PowerMAX OS Programming Guide

CONCURRENT COMPUTER CORPORATION™

0890423-080

December 2001

Printed in U. S. A.

Revision History:	Level:	Effective With:
Original Release -- August 1994	000	PowerUX 1.1
Previous Release -- August 1999	070	PowerMAX OS 4.3
Current Release -- December 2001	080	PowerMAX OS 5.1

Preface

Scope of Manual

This manual provides information needed for application programming in the PowerMAX OS™[1] operating system environment. It describes the system services provided by the system calls and libraries for the C programming language. It focuses on such topics as process scheduling, memory management, interprocess communications, and threads programming.

Structure of Manual

This manual consists of 14 chapters, 1 appendix, a glossary, and an index. A brief description of the chapters and appendix is presented as follows:

- Chapter 1 provides an introduction to the manual.
- Chapter 2 introduces the system calls and other system services that you can use to develop application programs.
- Chapter 3 discusses the system file and record locking facility. It also describes the STREAMS mechanism as it relates to input/output operations.
- Chapter 4 explains process management.
- Chapter 5 provides an overview of process scheduling and describes:
    - System V scheduler classes
    - POSIX®[2] scheduling policies
    - Scheduler priorities.
- Chapter 6 provides an overview of primary memory and explains the procedures for using memory management facilities.
- Chapter 7 discusses the general terminal interface to control asynchronous communication ports. It also addresses the STREAMS mechanism as it relates to terminal device control.
- Chapter 8 describes the programming interface to the internationalization feature.
- Chapter 9 discusses file and directory management. Security considerations including privileges and device security are also described.

1. PowerMAX OS is a trademark of Concurrent Computer Corporation
2. POSIX is a registered trademark of the Institute of Electrical and Electronics Engineers, Inc.

- Chapter 9 provides a detailed explanation of three interprocess communication facilities:
  - signals
  - job control
  - pipes.
- Chapter 11 introduces the Threads Library, which provides facilities for concurrent programming.
- Chapter 12 describes the InterProcess Communications (IPC) package that allows processes to exchange data and synchronize execution. The system calls for three IPC mechanisms are described:
  - messages
  - semaphores
  - shared memory.

  This chapter also describes the POSIX shared memory facilities.
- Chapter 13 discusses the synchronous polling mechanism and asynchronous event notification within STREAMS. It also explains STREAMS input/output multiplexing.
- Chapter 14 describes how to package software applications.
- Appendix A provides guidelines for writing trusted software.

The glossary contains definitions of technical terms that are important to understanding the concepts presented in this book.

The index contains an alphabetical reference to key terms and concepts and numbers of pages where they occur in the text.

Syntax Notation

The following notation is used throughout this guide:

italic	Books, reference cards, and items that the user must specify appear in *italic* type. Special terms may also appear in *italic*.
`list bold`	User input appears in **`list bold`** type and must be entered exactly as shown. Names of directories, files, commands, options and man page references also appear in **`list bold`** type.
`list`	Operating system and program output such as prompts and messages and listings of files and programs appears in `list` type.
`[]`	Brackets enclose command options and arguments that are optional. You do not type the brackets if you choose to specify such option or arguments

Referenced Publications

The following publications are referenced in this document:

0890240	*hf77 Fortran Reference Manual*
0890424	*Character User Interface Programming*
0890429	*System Administration Volume 1*
0890430	*System Administration Volume 2*
0890431	*Concurrent C Reference Manual*
0890459	*Compilation Systems Volume 1 (Tools)*
0890460	*Compilation Systems Volume 2 (Concepts)*
0890466	*PowerMAX OS Real-Time Guide*

Contents

Chapter 14 Packaging Your Software Applications

Tables

Screens

1 Introduction

1
Introduction

Introduction

This book, *PowerMAX OS Programming Guide,* concentrates on how to use the system services provided by the operating system. It is designed to give you information about application programming in a UNIX® [1] system environment. It does not attempt to teach you how to write programs. Rather, it is intended to supplement texts on programming by concentrating on the other elements that are part of getting application programs into operation.

Audience and Prerequisite Knowledge

As the title suggests, this manual assumes that you are a software developer. No special level of programming involvement is assumed. Hopefully, this book will also be useful to you if you work on or manage large application development projects.

If you are a programmer in the expert class, or if you are engaged in developing system software, you may find that the *PowerMAX OS Programming Guide* lacks the depth of information that you need. In this case, refer to the on-line system manual pages.

Knowledge of terminal use, of a UNIX system editor, and of the UNIX system directory/file structure is assumed. If you feel shaky about your mastery of these basic tools, you may want to look over the *User's Guide* before tackling this one.

Related Books and Documentation

Throughout this book, you will find pointers and references to other guides and manuals where information is described in more detail. In particular, you will find references to other programming guides (this document being a part of the programming guide series) and reference manuals. Both of these document sets are described below.

Programming Books

The components of PowerMAX OS include the shell command line interface (CLI), the Application Program Interface (API), and the Device Driver Interface/Driver Kernel Inter-

1. UNIX is a registered trademark, licensed exclusively by X/Open Company Ltd.

face (DDI/DKI). This document is part of a series of programming guides that includes the following:

- *PowerMAX OS Real-Time Guide* — Provides an introduction to the real-time features of PowerMAX OS and describes techniques for improving response time and increasing determinism. It contains documentation for interfaces that are used primarily by real-time applications. These interfaces include those for interprocess synchronization tools, POSIX clocks and timers, user-level interrupt routines, synchronized I/O, and asynchronous I/O.

  The *PowerMAX OS Programming Guide* contains documentation for interfaces that are used generally by both real-time and secure applications (for example, process management facilities, POSIX scheduling interfaces, signal management facilities, memory management facilities, and Threads Library facilities). It is intended that these two guides be used together.

- *Concurrent C Reference Manual* — Provides an introduction to Concurrent C, describes Concurrent extensions to the C language, and explains use of the Concurrent C compiler, compiler optimization options, and compilation modes.

- Compilation Systems Manuals — *The Compilation Systems Volume 1 (Tools)* describes the features and use of several software development environment tools, analysis tools, and project-control tools. The *Compilation Systems Volume 2 (Concepts)* describes the concepts underlying compilation systems. Such concepts include those related to environments, performance analysis, and formats.

- *Character User Interface Programming* — Provides guidelines on how to develop a menu and form-based interface that operates on ASCII character terminals running on PowerMAX OS.

Reference Set

The on-line reference set contains manual pages that formally and comprehensively describe features of the PowerMAX OS operating system. References to this documentation can be found throughout this book. Therefore, the reference set is recommended as a companion set to the PowerMAX OS programming guides. It is available only in on-line form and is composed of the following:

- *Command Reference* — Describes all user and administrator commands in the system.

- *Operating System API Reference* — Describes system calls and C language library functions.

- *System Files and Devices Reference* — Describes file formats, special files (devices), and miscellaneous system facilities.

- *Device Driver Reference* — Describes functions used by device driver software.

The C Connection

The UNIX system supports many programming languages, and C compilers are available on many different operating systems. Nevertheless, the relationship between the UNIX operating system and C has always been and remains very close. Most of the code in the UNIX operating system is written in the C language, and over the years many organizations using the UNIX system have come to use C for an increasing portion of their application code. Thus, while the *PowerMAX OS Programming Guide* is intended to be useful to you no matter what language(s) you are using, you will find that, unless there is a specific language-dependent point to be made, the examples assume you are programming in C. *Concurrent C Reference Manual* gives you detailed information about C language programming in the UNIX environment.

Hardware/Software Dependencies

If some commands just don't seem to exist at all, they may be members of packages not installed on your system. If you do find yourself trying to execute a non-existent command, talk to the administrators of your system to find out what you have available.

Information in the Examples

While every effort has been made to present displays of information just as they appear on your terminal, it is possible that your system may produce slightly different output. Some displays depend on a particular machine configuration that may differ from yours. Changes between releases of the UNIX system software may cause small differences in what appears on your terminal.

Where complete code samples are shown, we have tried to make sure they compile and work as represented. Where code fragments are shown, while we can't say that they have been compiled, we have attempted to maintain the same standards of coding accuracy for them.

Notation Conventions

Whenever the text includes examples of output from the computer and/or commands entered by you, we follow the standard notation scheme that is common throughout PowerMAX OS documentation:

- All computer input and output is shown in a `constant-width` font. Commands that you type in from your terminal are shown in constant-width type. Text that is printed on your terminal by the computer is shown in constant-width type.
- Comments added to a display to show that part of the display has been omitted are shown in *italic* type and are indented to separate them from the text that represents computer output or input. Comments that explain the input or output are shown in the same type font as the rest of the display.

An italic font is used to show substitutable text elements, such as the word "*filename*" for example.

- Because you are expected to press the RETURN key after entering a command or menu choice, the RETURN key is not explicitly shown in these cases. If, however, during an interactive session, you are expected to press RETURN without having typed any text, the notation is shown.
- Control characters are shown by the string "CTRL-" followed by the appropriate character, such as "d" (this is known as "CTRL-d"). To enter a control character, hold down the key marked "CTRL" (or "CONTROL") and press the d key.
- The standard default prompt signs for an ordinary user and `root` are the dollar sign ($) and the pound sign (#).
- When the # prompt is used in an example, the command illustrated may be executed only by `root`.

Manual Page References

Manual pages are available only in on-line form and are referred to with the function name showing first in constant width font, followed by the section number appearing in parentheses; for example, the Executable and Linking Format Library (ELF) manual page appears as **`elf(3E)`**. Reference manuals are not referred to individually; however, individual sections are referred to as "Section 3E in the Reference Manuals."

Section (1)	*Command Reference*
Sections (2), (3)	*Operating System API Reference*
Sections (4), (5), (7), (8)	*System Files and Devices Reference*
Section (9)	*Manual Descriptions*

Note that the *Command Reference* describes commands appropriate for general users and system administrators as well as for programmers.

Application Programming in the UNIX System Environment

This section introduces application programming in a UNIX system environment. It briefly describes what application programming is and then moves on to a discussion on UNIX system tools and where you can read about them, and to languages supported in the UNIX system environment and where you can read about them.

Programmers working on application programs develop software for the benefit of other, nonprogramming users. Most large commercial computer applications involve a team of applications development programmers. They may be employees of the end-user organization or they may work for a software development firm. Some of the people working in this environment may be more in the project management area than working programmers.

Application programming has some of the following characteristics:

- Applications are often large and are developed by a team of people who write requirements, designs, tests, and end-user documents. This implies use of a project management methodology, including version control (described in *Compilation Systems Volume 1 (Tools)),* change requests, tracking, and so on.
- Applications must be developed more robustly.
  - They must be easy to use, implying character or graphical user interfaces.
  - They must check all incoming data for validity (for example, using the Data Validation Tools described in *Compilation Systems Volume 1 (Tools)* and *Compilation Systems Volume 2 (Concepts)).*
  - They should be able to handle large amounts of data.
- Applications must be easy to install and administer.

UNIX System Tools and Languages

What is meant by the term *UNIX system tools*? In simple terms, it means an existing piece of software used as a component in a new task. In a broader context, the term is used often to refer to elements of the UNIX system that might also be called features, utilities, programs, filters, commands, languages, functions, and so on. It gets confusing because any of the things that might be called by one or more of these names can be, and often are, used simply as components of the solution to a programming problem. The chapter's aim is to give you some sense of the situations in which you use these tools, and how the tools fit together. It refers you to other chapters in this book or to other documents for more details.

Facilities Covered and Not Covered in This Guide

The *PowerMAX OS Programming Guide* is about facilities used by application programs in a UNIX system environment. Some tools may or may not be covered in this book. Actually, some of the subjects not covered in this programming guide might be even more important to you than those that are covered. This book could not possibly cover everything you ever need to know about UNIX system tools in one volume.

Tools not covered in this text are as follows:

- The **`login`** procedure
- UNIX system editors and how to use them
- How the file system is organized and how you move around in it
- Shell programming

Information about these subjects can be found in the *User's Guide* and a number of commercially available texts.

Tools that are covered in this text apply to application software development. This text also covers tools for packaging application and device driver software and for customizing the administrative interface.

Programming Tools and Languages in the UNIX System Environment

This section describes a variety of programming tools supported in the UNIX system environment. *Programming tools* refers to those tools offered for use on a computer running a current release of PowerMAX OS. Since these are separately purchasable items, not all of them will necessarily be installed on your machine. On the other hand, you may have programming tools and languages available on your machine that came from another source and are not mentioned in this discussion.

The C Language

C is intimately associated with the UNIX system since it was originally developed for use in recoding the UNIX system kernel. If you need to use a lot of UNIX system function calls for low-level I/O, memory or device management, or interprocess communication, C is a logical first choice. Most programs, however, don't require such direct interfaces with the operating system, so the decision to choose C might better be based on one or more of the following characteristics:

- a variety of data types: characters, integers of various sizes, and floating point numbers
- low-level constructs (most of the UNIX system kernel is written in C)
- derived data types such as arrays, functions, pointers, structures, and unions
- multidimensional arrays
- scaled pointers and the ability to do pointer arithmetic
- bitwise operators
- a variety of flow-of-control statements: `if`, `if-else`, `switch`, `while`, `do-while`, and `for`
- a high degree of portability

Refer to the *Concurrent C Reference Manual* for complete details on C.

It takes fairly concentrated use of the C language over a period of several months to reach your full potential as a C programmer. If you are a casual programmer, you might make it easier for yourself if you choose a less demanding programming facility such as those described below.

Shell

You can use the shell to create programs (new commands). Such programs are also called shell procedures.

awk

The **awk** program (its name is an acronym constructed from the initials of its developers) scans an input file for lines that match pattern(s) described in a specification file. When **awk** finds a line that matches a pattern, it performs actions also described in the specification. It is not uncommon that an **awk** program can be written in a couple of lines to do functions that would take a couple of pages to describe in a programming language like FORTRAN or C. For example, consider a case where you have a set of records that consist of a key field and a second field that represents a quantity, and the task is to output the sum of the quantities for each key. The pseudocode for such a program might look like this:

```
SORT RECORDS
Read the first record into a hold area;
Read additional records until EOF;
{
If the key matches the key of the record in the hold area,
  add the quantity to the quantity field of the held record;
If the key does not match the key of the held record,
  write the held record,
  move the new record to the hold area;
}
At EOF, write out the last record from the hold area.
```

An **awk** program to accomplish this task would look like this:

```
      { qty[$1] += $2 }
  END { for (key in qty) print key, qty[key] }
```

This illustrates only one characteristic of **awk**; its ability to work with associative arrays. With **awk**, the input file does not have to be sorted, which is a requirement of the pseudoprogram.

lex

lex is a lexical analyzer that can be added to C or FORTRAN programs. A lexical analyzer is interested in the vocabulary of a language rather than its grammar, which is a system of rules defining the structure of a language. **lex** can produce C language subroutines that recognize regular expressions specified by the user, take some action when a regular expression is recognized, and pass the output stream on to the next program.

For detailed information on **lex**, see the “**lex**” chapter in *Compilation Systems Volume 1 (Tools)* and *Compilation Systems Volume 2 (Concepts)* and **lex(1)** in the *Command Reference.*

yacc

yacc (Yet Another Compiler Compiler) is a tool for describing an input language to a computer program. **yacc** produces a C language subroutine that parses an input stream according to rules laid down in a specification file. The **yacc** specification file establishes a set of grammatical rules together with actions to be taken when tokens in the input match the rules. **lex** may be used with **yacc** to control the input process and pass tokens to the parser that applies the grammatical rules.

For detailed information on **yacc**, see the "**yacc**" chapter in *Compilation Systems Volume 1 (Tools)* and **yacc(1)** in the *Command Reference*.

m4

m4 is a macro processor that can be used as a preprocessor for assembly language and C programs. For details, see the "**m4**" chapter of *Concurrent C Reference Manual* and **m4(1)** in the *Command Reference*.

bc and dc

bc enables you to use a computer terminal as you would a programmable calculator. You can edit a file of mathematical computations and call **bc** to execute them. The **bc** program uses **dc**. You can use **dc** directly, if you want, but it takes a little getting used to since it works with reverse Polish notation. **bc** and **dc** are described in Section 1 of the *Command Reference*.

Character User Interfaces

curses

Actually a library of C functions, `curses` is included in this list because the set of functions comprise a sublanguage for dealing with terminal screens. If you are writing programs that include interactive user screens, you will want to become familiar with this group of functions.

For detailed information on `curses`, see the *Character User Interface Programming*.

FMLI

The Form and Menu Language Interpreter (FMLI) is a high-level programming tool having two main parts:

- The Form and Menu Language, a programming language for writing scripts that define how an application will be presented to users. The syntax of the Form and Menu Language is very similar to that of the UNIX system shell programming language, including variable setting and evaluation, built-in commands and functions, use of and escape from special characters, redirection of input and output, conditional statements, interrupt signal handling, and the ability to set various terminal attributes. The Form and Menu Language also includes sets of “descriptors,” which are used to define or customize attributes of frames and other objects in your application.

- The Form and Menu Language Interpreter, **`fmli`**, which is a command interpreter that sets up and controls the video display screen on a terminal, using instructions from your scripts to supplement FMLI's predefined screen control mechanisms. FMLI scripts can also invoke UNIX system commands and C executables, either in the background or in full screen mode. The Form and Menu Language Interpreter operates similarly to the UNIX command interpreter **`sh`**. At run time it parses the scripts you have written, thus giving you the advantages of quick prototyping and easy maintenance.

FMLI provides a framework for developers to write applications and application interfaces that use menus and forms. It controls many aspects of screen management for you. This means that you do not have to be concerned with the low-level details of creating or placing frames, providing users with a means of navigating between or within frames, or processing the use of forms and menus. Nor do you need to worry about on which kind of terminal your application will be run. FMLI takes care of all that for you.

For details see the FMLI chapter in the *Character User Interface Programming.*

ETI

The Extended Terminal Interface (ETI) is a set of C library routines that promote the development of application programs displaying and manipulating windows, panels, menus, and forms and that run under the UNIX system. ETI consists of

- the low-level (`curses`) library
- the `panel` library
- the `menu` library
- the `form` library
- the TAM Transition library

The routines are C functions and macros; many of them resemble routines in the standard C library. For example, there's a routine **`printw`** that behaves much like **`printf`** and another routine **`getch`** that behaves like **`getc`**. The automatic teller program at your bank might use **`printw`** to print its menus and **`getch`** to accept your requests for withdrawals (or, better yet, deposits). A visual screen editor like the UNIX system screen editor **`vi`** might also use these and other ETI routines.

A major feature of ETI is cursor optimization. Cursor optimization minimizes the amount a cursor has to move around a screen to update it. For example, if you designed a screen editor program with ETI routines and edited the sentence

```
ETI is a great package for creating forms and menus
```

to read

```
ETI is the best package for creating forms and menus
```

the program would change only “`the best`” in place of “`a great`”. The other characters would be preserved. Because the amount of data transmitted—the output—is minimized, cursor optimization is also referred to as output optimization.

Cursor optimization takes care of updating the screen in a manner appropriate for the terminal on which an ETI program is run. This means that ETI can do whatever is required to update many different terminal types. It searches the `terminfo` database to find the correct description for a terminal.

How does cursor optimization help you and those who use your programs? First, it saves you time in describing in a program how you want to update screens. Second, it saves a user's time when the screen is updated. Third, it reduces the load on your UNIX system's communication lines when the updating takes place. Fourth, you don't have to worry about the myriad of terminals on which your program might be run.

Figure 1-1 shows a simple ETI program. It uses some of the basic ETI routines to move a cursor to the middle of a terminal screen and print the character string `BullsEye`. For now, just look at their names and you will get an idea of what each of them does:

```
#include <curses.h>

main()
{
   initscr();

   move( LINES/2 - 1, COLS/2 - 4 );
   addstr(“Bulls”);
   refresh();
   addstr(“Eye”);
   refresh();
   endwin();
}
```

Figure 1-1. A Simple ETI Program

For complete information on ETI, refer to the ETI chapter in the *Character User Interface Programming.*

Graphical User Interfaces

XWIN Graphical Windowing System

The XWIN Graphical Windowing System is a network-transparent window system. X display servers run on computers with either monochrome or color bitmap display hardware. The server distributes user input to and accepts output requests from various application programs (referred to as “clients”). Each client is located on either the same machine or on another machine in the network.

The clients use **`Xlib`**, a C library routine, to interface with the window system by means of a stream connection.

“Widgets” are a set of code and data that provide the look and feel of a user interface. The C library routines used for creating and managing widgets are called the X Intrinsics. They are built on top of the X Window System, monitor events related to user interactions, and dispatch the correct widget code to handle the display. Widgets can then call application-registered routines (called callbacks) to handle the specific application semantics of an interaction. The X Intrinsics also monitor application-registered, nongraphical events and dispatch application routines to handle them. These features allow programmers to use this implementation of an OPEN LOOK toolkit in data base management, network management, process control, and other applications requiring response to external events.

Clients sometimes use a higher level library of the X Intrinsics and a set of widgets in addition to **`Xlib`**. Refer to the *X Window System User’s Guide R5* for general information about the design of X and to the *OSF/Motif Documentation Set* for information about widgets.

System Calls and Libraries

This section describes the system services supplied by system calls and libraries for the C programming language. It introduces such topics as the process scheduler, virtual memory, interprocess communication, file and record locking, and symbolic links. The system calls and libraries that programs use to access these system services are described in detail later in this book.

File and Device Input/Output

UNIX system applications can do all I/O by reading or writing files, because all I/O devices, even a user's terminal, are files in the file system. Each peripheral device has an entry in the file system hierarchy, so that device names have the same structure as file names, and the same protection mechanisms apply to devices as to files. Using the same I/O calls on a terminal as on any file makes it easy to redirect the input and output of commands from the terminal to another file. Besides the traditionally available devices, names exist for disk devices regarded as physical units outside the file system, and for absolutely addressed memory.

STREAMS Input/Output

STREAMS is a general, flexible facility and a set of tools for development of UNIX system communication services. It supports the implementation of services ranging from complete networking protocol suites to individual device drivers. STREAMS defines standard interfaces for character input/output within the kernel, and between the kernel and the rest of the UNIX system. The associated mechanism is simple and open-ended. It consists of a set of system calls, kernel resources, and kernel routines.

The standard interface and mechanism enable modular, portable development and easy integration of high-performance network services and their components. STREAMS does not impose any specific network architecture. The STREAMS user interface is upwardly compatible with the character I/O user level functions such as **`open`**, **`close`**, **`read`**,

`write`, and **`ioctl`**. Benefits of STREAMS are discussed in more detail later in this chapter.

A "Stream" is a full-duplex processing and data transfer path between a STREAMS driver in kernel space and a process in user space.

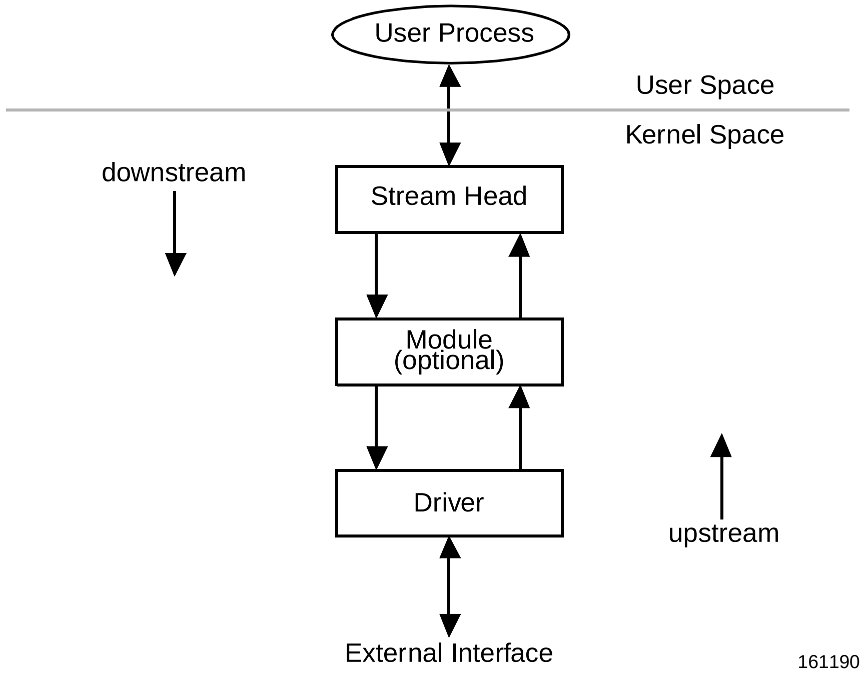


Figure 1-2. Simple Streams

In the kernel, a Stream is constructed by linking a Stream head, a driver, and zero or more modules between the Stream head and driver.The "Stream head" is the end of the Stream nearest to the user process. All system calls made by a user level process on a Stream are processed by the Stream head.

Pipes are also STREAMS-based. A STREAMS-based pipe is a full-duplex (bidirectional) data transfer path in the kernel.It implements a connection between the kernel and one or more user processes and also shares properties of STREAMS-based devices.

A STREAMS driver may be a device driver that provides the services of an external I/O device, or a software driver, commonly referred to as a pseudo-device driver. The driver typically handles data transfer between the kernel and the device and does little or no processing of data other than conversion between data structures used by the STREAMS mechanism and data structures that the device understands.

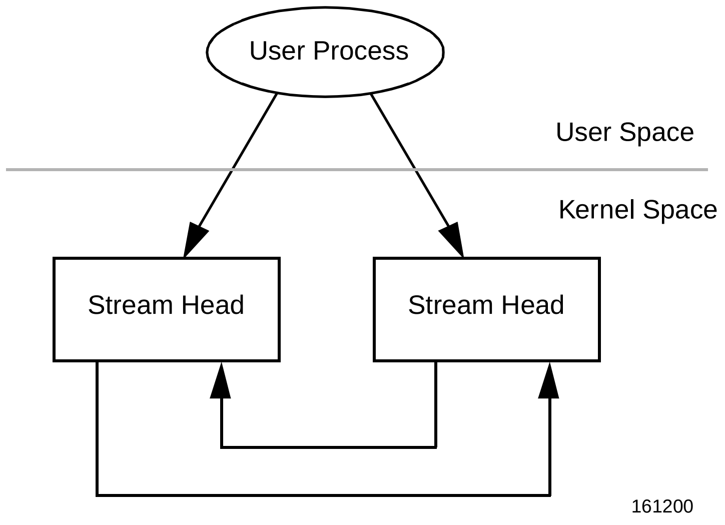


Figure 1-3. STREAMS-based Pipe

A STREAMS module represents processing functions to be performed on data flowing on the Stream. The module is a defined set of kernel-level routines and data structures used to process data, status, and control information. Data processing may involve changing the way the data is represented, adding/deleting header and trailer information to data, and/or packetizing/depacketizing data. Status and control information includes signals and input/output control information.

Each module is self-contained and functionally isolated from any other component in the Stream except its two neighboring components. The module communicates with its neighbors by passing messages. The module is not a required component in STREAMS, whereas the driver is, except in a STREAMS-based pipe where only the Stream head is required.

One or more modules may be inserted into a Stream between the Stream head and driver to perform intermediate processing of messages as they pass between the Stream head and driver.STREAMS modules are dynamically interconnected in a Stream by a user process.No kernel programming, assembly, or link editing is required to create the interconnection.

STREAMS uses queue structures to keep information about given instances of a pushed module or opened STREAMS device. A queue is a data structure that contains status information, a pointer to routines for processing messages, and pointers for administering the Stream. Queues are always allocated in pairs; one queue for the read-side and the other for the write-side. There is one queue pair for each driver and module, and the Stream head. The pair of queues is allocated whenever the Stream is opened or the module is pushed (added) onto the Stream.

Data is passed between a driver and the Stream head and between modules in the form of messages. A message is a set of data structures used to pass data, status, and control information between user processes, modules, and drivers. Messages that are passed from the Stream head toward the driver or from the process to the device, are said to travel downstream (also called write-side). Similarly, messages passed in the other direction, from the device to the process or from the driver to the Stream head, travel upstream (also called read-side).

A STREAMS message is made up of one or more message blocks.Each block consists of a header, a data block, and a data buffer. The Stream head transfers data between the data space of a user process and STREAMS kernel data space. Data to be sent to a driver from a user process is packaged into STREAMS messages and passed downstream. When a message containing data arrives at the Stream head from downstream, the message is processed by the Stream head, which copies the data into user buffers.

Within a Stream, messages are distinguished by a type indicator. Certain message types sent upstream may cause the Stream head to perform specific actions, such as sending a signal to a user process. Other message types are intended to carry information within a Stream and are not directly seen by a user process.

File and Record Locking

The provision for locking files, or portions of files, is primarily used to prevent the sort of error that can occur when two or more users of a file try to update information at the same time. The classic example is the airlines reservation system where two ticket agents each assign a passenger to Seat A, Row 5 on the 5 o'clock flight to Detroit. A locking mechanism is designed to prevent such mishaps by blocking Agent B from even seeing the seat assignment file until Agent A's transaction is complete.

File locking and record locking are really the same thing, except that file locking implies the whole file is affected; record locking means that only a specified portion of the file is locked. (Remember, in the UNIX system, file structure is undefined; a record is a concept of the programs that use the file.)

Two types of locks are available: read locks and write locks. If a process places a read lock on a file, other processes can also read the file but all are prevented from writing to it, that is, changing any of the data. If a process places a write lock on a file, no other processes can read or write in the file until the lock is removed. Write locks are also known as exclusive locks. The term shared lock is sometimes applied to read locks.

Another distinction needs to be made between mandatory and advisory locking. Mandatory locking means that the discipline is enforced automatically for the system calls that read, write, or create files. This is done through a permission flag established by the file's owner (or the superuser). Advisory locking means that the processes that use the file take the responsibility for setting and removing locks as needed. Thus, mandatory may sound like a simpler and better deal, but it isn't so. The principal weakness in the mandatory method is that the lock is in place only while the single system call is being made. It is extremely common for a single transaction to require a series of reads and writes before it can be considered complete. In cases like this, the term atomic is used to describe a transaction that must be viewed as an indivisible unit. The preferred way to manage locking in such a circumstance is to make certain the lock is in place before any I/O starts, and that it is not removed until the transaction is done. That calls for locking of the advisory variety.

Where to Find More Information

Chapter 3 discusses file and device I/O including file and record locking in detail with a number of examples. There is an example of file and record locking in the sample application in Chapter 2. The manual pages that specifically address file and record locking are **`fcntl(2)`**, **`lockf(3)`** and **`chmod(2)`** and **`fcntl(5)`**. **`fcntl(2)`** describes the system call for file and record locking (although it isn't limited to that only) **`fcntl(5)`** tells you the file control options. The subroutine **`lockf(3)`** can also be used to lock sections of a file or an entire file. Setting **`chmod`** so that all portions of a file are locked will ensure that parts of files are not corrupted.

Memory Management

The UNIX system includes a complete set of memory-mapping mechanisms. Process address spaces are composed of a vector of memory pages, each of which can be independently mapped and manipulated. The memory-management facilities

- unify the system's operations on memory
- provide a set of kernel mechanisms powerful and general enough to support the implementation of fundamental system services without special-purpose kernel support
- maintain consistency with the existing environment, in particular using the UNIX file system as the name space for named virtual-memory objects

The system's virtual memory consists of all available physical memory resources including local and remote file systems, processor primary memory, swap space, and other random-access devices. Named objects in the virtual memory are referenced through the UNIX file system. However, not all file system objects are in the virtual memory; devices that the UNIX system cannot treat as storage, such as terminal and network device files, are not in the virtual memory. Some virtual memory objects, such as private process memory and shared memory segments, do not have names.

The Memory Mapping Interface

The applications programmer gains access to the facilities of the virtual memory system through several sets of system calls.

- **`mmap`** establishes a mapping between a process's address space and a virtual memory object.
- **`mprotect`** assigns access protection to a block of virtual memory.
- **`munmap`** removes a memory mapping.
- **`getpagesize`** returns the system-dependent size of a memory page.
- **`mincore`** tells whether mapped memory pages are in primary memory.

Where to Find More Information

Chapter 6 gives a detailed description of the virtual memory system. Refer to the **`mmap(2)`**, **`mprotect(2)`**, **`munmap(2)`**, **`getpagesize(2)`** and **`mincore(2)`** system manual pages.

Process Management and Scheduling

Beginning with the operating system (OS), the schedulable entity is always a lightweight process (LWP). Scheduling priorities and classes are attributes of LWPs and not processes. When scheduling system calls accept a process on which to operate, the operation is applied to each LWP in the process. The UNIX system scheduler determines when LWPs run. It maintains priorities based on configuration parameters, process behavior, and user requests; it uses these priorities to assign LWPs to the CPU.

The OS gives users absolute control over the sequence in which certain LWPs run, and the amount of time each LWP may use the CPU before another LWP gets a chance

By default, the scheduler uses a time-sharing policy similar to the policy used in previous releases. A time-sharing policy adjusts priorities dynamically in an attempt to provide good response time to interactive LWPs and good throughput to CPU-intensive LWPs.

A fixed class scheduling policy is available, also. It is similar to the time-sharing policy except that the time slices given to fixed class processes or LWPs do not degrade over time.

The scheduler offers a fixed priority scheduling policy as well as a time-sharing policy. Fixed priority scheduling allows users to set fixed priorities on a per-process or LWP basis. The highest-priority fixed priority LWP always gets the CPU as soon as it is runnable, even if system processes are runnable. An application can therefore specify the exact order in which LWPs run. An application may also be written so that its fixed priority LWPs have a guaranteed response time from the system.

For most UNIX environments, the default scheduler configuration works well and no fixed priority LWPs are needed: administrators should not change configuration parameters and users should not change scheduler properties of their applications. However, for some applications with strict timing constraints, fixed priority LWPs are the only way to guarantee that the application's requirements are met.

Where to Find More Information

Chapter 5 gives detailed information on the process scheduler, along with relevant code examples. See also the **`priocntl(1)`**, **`priocntl(2)`**, and **`dispadmin(1M)`** system manual pages.

Interprocess Communications

Pipes, named pipes, and signals are all forms of interprocess communication. Business applications running on a UNIX system computer, however, often need more sophisticated methods of communication. In applications, for example, where fast response is critical, a number of processes may be brought up at the start of a business day to be constantly available to handle transactions on demand. This cuts out initialization time that can add seconds to the time required to deal with the transaction. To go back to the ticket reservation example again for a moment, if a customer calls to reserve a seat on the 5 o'clock flight to Detroit, you don't want to have to say, “Yes, sir; just hang on a minute while I start up the reservations program.” In transaction-driven systems, the normal mode of processing is to have all the components of the application standing by waiting for some sort of an indication that there is work to do.

To meet requirements of this type, the UNIX system offers a set of nine system calls and their accompanying header files, all under the umbrella name of interprocess communications (IPC).

The IPC system calls come in sets of three; one set each for messages, semaphores, and shared memory. These three terms define three different styles of communication between processes:

messages	Communication is in the form of data stored in a buffer. The buffer can be either sent or received.
semaphores	Communication is in the form of positive integers with a value between 0 and 32,767. Semaphores may be contained in an array the size of which is determined by the system administrator. The default maximum size for the array is 25.
shared memory	Communication takes place through a common area of main memory. One or more processes can attach a segment of memory and as a consequence can share whatever data is placed there.

The sets of IPC system calls are:

`msgget`	**`semget`**	**`shmget`**
`msgctl`	**`semctl`**	**`shmctl`**
`msgop`	**`semop`**	**`shmop`**

The “**`get`**” calls each return to the calling program an identifier for the type of IPC facility that is being requested.

The “**`ctl`**” calls provide a variety of control operations that include obtaining (`IPC_STAT`), setting (`IPC_SET`) and removing (`IPC_RMID`), the values in data structures associated with the identifiers picked up by the “**`get`**” calls.

The “**`op`**” manual pages describe calls that are used to perform the particular operations characteristic of the type of IPC facility being used. **`msgop`** has calls that send or receive messages. **`semop`** (the only one of the three that is actually the name of a system call) is used to increment or decrement the value of a semaphore, among other functions. **`shmop`** has calls that attach or detach shared memory segments.

Where to Find More Information

Chapter 12 gives a detailed description of IPC, with many code examples that use the IPC system calls. The system calls are described in Section 2 of the *Operating System API Reference.*

Symbolic Links

A symbolic link is a special type of file that represents another file. The data in a symbolic link consists of the path name of a file or directory to which the symbolic link file refers. The link that is formed is called symbolic to distinguish it from a regular (also called a hard) link. A symbolic link differs functionally from a regular link in three major ways.

- Files from different file systems may be linked.
- Directories, as well as regular files, may be symbolically linked by any user.
- A symbolic link can be created even if the file it represents does not exist

When a user creates a regular link to a file, a new directory entry is created containing a new filename and the inode number of an existing file. The link count of the file is incremented.

In contrast, when a user creates a symbolic link (using the **`ln(1)`** command with the **`-s`** option), both a new directory entry and a new inode are created. A data block is allocated to contain the path name of the file to which the symbolic link refers. The link count of the referenced file is not incremented.

Symbolic links can be used to solve a variety of common problems. For example, it frequently happens that a disk partition (such as **`root`**) runs out of disk space. With symbolic links, an administrator can create a link from a directory on that file system to a directory on another file system. Such a link provides extra disk space and is, in most cases, transparent to both users and programs.

Symbolic links can also help deal with the built-in path names that appear in the code of many commands. Changing the path names would require changing the programs and recompiling them. With symbolic links, the path names can effectively be changed by making the original files symbolic links that point to new files.

In a shared resource environment like NFS, symbolic links can be very useful. For example, if it is important to have a single copy of certain administrative files, symbolic links can be used to help share them. Symbolic links can also be used to share resources selectively. Suppose a system administrator wants to do a remote mount of a directory that contains sharable devices. These devices must be in **`/dev`** on the client system, but this system has devices of its own so the administrator does not want to mount the directory onto **`/dev`**. Rather than do this, the administrator can mount the directory at a location other than **`/dev`** and then use symbolic links in the **`/dev`** directory to refer to these remote devices. (This is similar to the problem of built-in path names since it is normally assumed that devices reside in the **`/dev`** directory.)

Finally, symbolic links can be valuable within the context of the virtual file system (VFS) architecture. With VFS, new services, such as higher performance files and network IPC, may be provided on a file system basis. Symbolic links can be used to link these services to home directories or to places that make more sense to the application or user. Thus, you might create a data base index file in a RAM-based file system type and symbolically link it to the place where the data base server expects it and manages it.

Where to Find More Information

Chapter 9 discusses symbolic links in detail. Refer to the **`symlink(2)`** system manual page for information on creating symbolic links. See also **`stat(2)`**, **`rename(2)`**, **`link(2)`**, **`readlink(2)`**, **`unlink(2)`**, and **`ln(1)`**.

2
System Calls and Libraries

2 System Calls and Libraries

Introduction

This chapter introduces the system calls and other system services you can use to develop application programs. Each application performs a different function, but goes through the same basic steps: input, processing, and output. For the input and output steps, most applications interact with an end user at a terminal. During the processing step, sometimes an application needs access to special services provided by the operating system (for example, to interact with the file system, control processes, manage memory, and more). Some of these services are provided through system calls and some through libraries of functions.

Libraries and Header Files

The standard libraries supplied by the C compilation system contain functions that you can use in your program to perform input/output, string handling, and other high-level operations that are not explicitly provided by the C language. Header files contain definitions and declarations that your program will need if it calls a library function. They also contain function-like macros that you can use in your program as you would a function.

In this part, we'll talk a bit more about header files and show you how to use library functions in your program. We'll also describe the contents of some of the more important standard libraries, and tell you where to find them in the *Operating System API Reference.* We'll close with a brief discussion of standard I/O.

Header Files

Header files serve as the interface between your program and the libraries supplied by the C compilation system. Because the functions that perform standard I/O, for example, very often use the same definitions and declarations, the system supplies a common interface to the functions in the header file **`<stdio.h>`**. By the same token, if you have definitions or declarations that you want to make available to several source files, you can create a header file with any editor, store it in a convenient directory, and include it in your program as described in the first part of this chapter.

Header files traditionally are designated by the suffix **`.h`**, and are brought into a program at compile time. The preprocessor component of the compiler does this because it interprets the `#include` statement in your program as a directive. The two most commonly used directives are `#include` and `#define`. As we have seen, the `#include` directive is used to call in and process the contents of the named file. The `#define` directive is used to define the replacement token string for an identifier. For example,

```
#define NULL   0
```

defines the macro `NULL` to have the replacement token sequence `0`. See the *Concurrent C Reference Manual* for information on preprocessing directives.

Many different **`.h`** files are named in the *Operating System API Reference.* Here we are going to list a number of them, to illustrate the range of tasks you can perform with header files and library functions. When you use a library function in your program, the manual page will tell you which header file, if any, needs to be included. If a header file is mentioned, it should be included before you use any of the associated functions or declarations in your program. It's generally best to put the `#include` right at the top of a source file.

`assert.h`	assertion checking
`ctype.h`	character handling
`errno.h`	error conditions
`float.h`	floating point limits
`limits.h`	other data type limits
`locale.h`	program's locale
`math.h`	mathematics
`setjmp.h`	nonlocal jumps
`signal.h`	signal handling
`stdarg.h`	variable arguments
`stddef.h`	common definitions
`stdio.h`	standard input/output
`stdlib.h`	general utilities
`string.h`	string handling
`time.h`	date and time
`unistd.h`	system calls

How to Use Library Functions

The manual page for each function describes how you should use the function in your program. Manual pages follow a common format; although, some manual pages may omit some sections:

- The **"NAME"** section names the component(s) and briefly states its purpose.
- The **"SYNOPSIS"** section specifies the C language programming interface(s).

- The **“DESCRIPTION”** section details the behavior of the component(s).
- The **“EXAMPLE”** section gives examples, caveats and guidance on usage.
- The **“FILES**” section gives the file names that are built into the program.
- The **“SEE ALSO**” section lists related component interface descriptions.
- The **“DIAGNOSTICS”** section outlines return values and error conditions.

The **“NAME”** section lists the names of components described in that manual page with a brief, one-line statement of the nature and purpose of those components.

The **“SYNOPSIS”** section summarizes the component interface by compactly representing the order of any arguments for the component, the type of each argument (if any) and the type of value the component returns.

The **“DESCRIPTION”** section specifies the functionality of components without stipulating the implementation; it excludes the details of how the OS implements these components and concentrates on defining the external features of a standard computing environment instead of the internals of the operating system, such as the scheduler or memory manager. Portable software should avoid using any features or side-effects not explicitly defined.

The **“SEE ALSO”** section refers the reader to other related manual pages in the PowerMAX OS reference manual set as well as other documents. The **“SEE ALSO**” section identifies manual pages by the title which appears in the upper corners of each page of a manual page.

Some manual pages cover several commands, functions or other PowerMAX OS components; thus, components defined along with other related components share the same manual page title. For example, references to the function **`calloc`** cite **`malloc(3)`** because the function **`calloc`** is described with the function malloc in the manual page entitled **`malloc(3)`**. As an example manual page, we'll look at the **`strcmp`** function, which compares character strings. The routine is described on the `string` manual page in Section 3, Subsection 3C, of the *Operating System API Reference.* Related functions are described there as well, but only the sections relevant to **`strcmp`** are shown in Figure 2-1.

As shown, the **“DESCRIPTION”** section tells you what the function or macro does. It's the **“SYNOPSIS”** section, though, that contains the critical information about how you use the function or macro in your program. Note that the first line in the **“SYNOPSIS”** is

```
#include <string.h>
```

That means that you should include the header file `<string.h>` in your program because it contains useful definitions or declarations relating to `strcmp`.

In fact, **`<string.h>`** contains the **`strcmp`** “function prototype” as follows:

```
extern int strcmp(const char *, const char *);
```

A function prototype describes the kinds of arguments expected and returned by a C language function. Function prototypes afford a greater degree of argument type checking than old-style function declarations, and reduce the chance of using the function incorrectly. Including **`<string.h>`**, assures that the C compiler checks calls to **`strcmp`** against the official interface. You can, of course, examine **`<string.h>`** in the standard place for header files on your system, usually the **`/usr/include`** directory.

NAME

`strcat`, **`strdup`**, **`strncat`**, **`strcmp`**, **`strncmp`**, **`strcpy`**, **`strncpy`**, **`strlen`**,
`strchr`, **`strrchr`**, **`strpbrk`**, **`strspn`**, **`strcspn`**,
`strtok` - string operations

SYNOPSIS

`#include` **`<string.h>`**
`. . .`
`int` **`strcmp`**`(const char *`*sptr1*`, const char *`*sptr2*`); . . .`

DESCRIPTION

. . . **`strcmp`** compares its arguments and returns an integer less than, equal to, or greater than 0, according as the first argument is lexicographically less than, equal to, or greater than the second . . .

Figure 2-1. Sample DIAGNOSTICS Section

The **“SYNOPSIS”** for a C library function closely resembles the C language declaration of the function and its arguments. The **“SYNOPSIS”** tells the reader:

- the type of value returned by the function;
- the arguments the function expects to receive when called, if any;
- the argument types.

For example, the **“SYNOPSIS”** for the macro **`feof`** is:

`#include` **`<stdio.h>`**
`int` **`feof`**`( FILE *`*sfp*` )`

The **“SYNOPSIS”** section for **`feof`** shows that:

- The macro **`feof`** requires the header file **`<stdio.h>`**
- The macro **`feof`** returns a value of type `int`
- The argument *sfp* is a pointer to an object of type `FILE`

To use **`feof`** in a program, you need only write the macro call, preceded at some point by the `#include` control line, as in the following:

```
#include <stdio.h>   /* include definitions */

main() {
   FILE *infile;     /* define a file pointer */
   while (!feof(infile)) {   /* until end-of-file */
      /* operations on the file */
   }
}
```

By way of further illustration, here is an example of how you might use **`strcmp`** in your own code. The following figure shows a program fragment that will find the bird of your choice in an array of birds.

```
#include <string.h>

/* birds must be in alphabetical order */
char *birds[] = { “albatross”,  “canary”, “cardinal”, “ostrich”, “penguin” };

/* Return the index of the bird in the array. */
/* If the bird is not in the array, return -1 */

int is_bird(const char *string)
{
    int low, high, midpoint;
    int cmp_value;

    /* use a binary search to find the bird */
    low = 0;
    high = sizeof(birds)/sizeof(char *) - 1;
    while(low <= high)
    {
        midpoint = (low + high)/2;
        cmp_value = strcmp(string, birds[midpoint]);
        if (cmp_value < 0)
            high = midpoint - 1;
        else if (cmp_value > 0)
            low = midpoint + 1;
        else /* found a match */
            return midpoint;
    }
    return -1;
}
```

Figure 2-2. How strcmp Is Used in a Program

The format of a **“SYNOPSIS”** section only resembles, but does not duplicate, the format of C language declarations. To show that some components take varying numbers of arguments, the **“SYNOPSIS”** section uses additional conventions not found in actual C function declarations:

- Text in `courier` represents source-code typed just as it appears.
- Text in *italic* usually represents substitutable argument prototypes.
- Square brackets [ ] around arguments indicate optional arguments.
- Ellipses . . . indicate that the previous arguments may repeat.
- If the type of an argument may vary, the **“SYNOPSIS”** omits the type.

For example, the **“SYNOPSIS”** for the function **`printf`** is:

```
#include <stdio.h>
int printf( char *fmt [ , arg . . . ] )
```

The **“SYNOPSIS”** section for **`printf`** shows that the argument **`arg`** is optional, may be repeated and is not always of the same data type. The **”DESCRIPTION”** section of the manual page provides any remaining information about the function **`printf`** and the arguments to it.

The **"DIAGNOSTICS"** section specifies return values and possible error conditions. The text in **"DIAGNOSTICS"** takes a conventional form which describes the return value in case of successful completion followed by the consequences of an unsuccessful completion, as in the following example:

Figure 2-3. Sample DIAGNOSTICS Section

On success, **`lseek`** returns the value of the resulting file-offset, as measured in bytes from the beginning of the file.

On failure, **`lseek`** returns `-1`, it does not change the file-offset, and `errno` equals:

`EBADF` if `fildes` is not a valid open file-descriptor.

`EINVAL` if `whence` is not `SEEK_SET`, `SEEK_CUR` or `SEEK_END`.

`ESPIPE` if `fildes` denotes a pipe or FIFO.

The **`<errno.h>`** header file defines symbolic names for error conditions which are described in **`intro(2)`** of the *Operating System API Reference.* For more information on error conditions, see the section entitled "System Call Error Handling" in this chapter.

C Library (libc)

In this section, we describe some of the more important routines in the standard C library. As we indicated in the first part of this chapter, **`libc`** contains the system calls described in Section 2 of the *Operating System API Reference,* and the C language functions described in Section 3, Subsections 3C and 3S. We'll explain what each of these subsections contains below. We'll look at system calls at the end of the section.

Subsection 3C Routines

Subsection 3C of the *Operating System API Reference* contains functions and macros that perform a variety of tasks:

- string manipulation
- character classification
- character conversion

Figure 2-4 lists string-handling functions that appear on the `string` page in Subsection 3C of the *Operating System API Reference.* Programs that use these functions should include the header file **`<string.h>`.**

>

Figure 2-4. String Operations

`strcat`	Append a copy of one string to the end of another.
`strncat`	Append no more than a given number of characters from one string to the end of another.
`strcmp`	Compare two strings. Returns an integer less than, greater than, or equal to 0 to show that one is lexicographically less than, greater than, or equal to the other.
`strncmp`	Compare no more than a given number of characters from the two strings. Results are otherwise identical to **`strcmp`**.
`strcpy`	Copy a string.
`strncpy`	Copy a given number of characters from one string to another. The destination string will be truncated if it is longer than the given number of characters, or padded with null characters if it is shorter.
`strdup`	Return a pointer to a newly allocated string that is a duplicate of a string pointed to.
`strchr`	Return a pointer to the first occurrence of a character in a string, or a null pointer if the character is not in the string.
`strrchr`	Return a pointer to the last occurrence of a character in a string, or a null pointer if the character is not in the string.
`strlen`	Return the number of characters in a string.
`strpbrk`	Return a pointer to the first occurrence in one string of any character from the second, or a null pointer if no character from the second occurs in the first.
`strspn`	Return the length of the initial segment of one string that consists entirely of characters from the second string.
`strcspn`	Return the length of the initial segment of one string that consists entirely of characters not from the second string.
`strstr`	Return a pointer to the first occurrence of the second string in the first string, or a null pointer if the second string is not found.
`strtok`	Break up the first string into a sequence of tokens, each of which is delimited by one or more characters from the second string. Return a pointer to the token, or a null pointer if no token is found.

Figure 2-5 lists functions and macros that classify 8-bit character-coded integer values. These routines appear on the **`conv(3C)`** and **`ctype(3C)`** pages in Subsection 3C of the *Operating System API Reference*. Programs that use these routines should include the header file **`<ctype.h>`**

.

Figure 2-5. Classifying 8-Bit Character-Coded Integer Values

isalpha	Is *c* a letter?
isupper	Is *c* an uppercase letter?
islower	Is *c* a lowercase letter?
isdigit	Is *c* a digit [0-9]?
isxdigit	Is *c* a hexadecimal digit [0-9], [A-F], or [a-f]?
isalnum	Is *c* alphanumeric (a letter or digit)?
isspace	Is *c* a space, horizontal tab, vertical tab, new-line, form-feed, or carriage return?
ispunct	Is *c* a punctuation character (neither control nor alphanumeric)?
isprint	Is *c* a printing character?
isgraph	Same as **isprint** except false for a space.
iscntrl	Is *c* a control character or a delete character?
isascii	Is *c* an ASCII character?
toupper	Change lower case to upper case.
_toupper	Macro version of toupper.
tolower	Change upper case to lower case.
_tolower	Macro version of tolower.
toascii	Turn off all bits that are not part of a standard ASCII character; intended for compatibility with other systems.

Figure 2-6 lists functions and macros in Subsection 3C of the *Operating System API Reference* that are used to convert characters, integers, or strings from one representation to another. The left-hand column contains the name that appears at the top of the manual page; the other names in the same row are related functions or macros described on the same manual page. Programs that use these routines should include the header file **<stdlib.h>**.

Figure 2-6. Converting Characters, Integers, or Strings

a64l	**l64a**			Convert between long integer and base-64 ASCII string.
ecvt	**fcvt**	**gcvt**		Convert floating point number to string.
l3tol	**ltol3**			Convert between 3-byte packed integer and long integer.
strtod	**atof**			Convert string to double-precision number.
strtol	**stroll atoll**	**atol**	**atoi**	Convert string to integer.
strtoul	**stroull**			Convert string to unsigned long.

Subsection 3S Routines

Subsection 3S of the *Operating System API Reference* contains the so-called standard I/O library for C programs. Frequently, one manual page describes several related functions or macros. In Figure 2-7,the left-hand column contains the name that appears at the top of the manual page; the other names in the same row are related functions or macros described on the same manual page. Programs that use these routines should include the header file **`<stdio.h>`**. We'll talk a bit more about standard I/O in the last subsection of this chapter.

Figure 2-7. Standard I/O Functions and Macros

`fclose`	**`fflush`**			Close or flush a stream.
`ferror`	**`feof`**	**`clearerr`**	**`fileno`**	Stream status inquiries.
`fopen fopen64 freopen64`	**`freopen`**	**`fdopen`**		Open a stream.
`fread`	**`fwrite`**			Input/output.
`fseek fgetpos fgetpos64 fseeko fsetpos fsetpos64 ftello ftello64`	**`rewind`**	**`ftell`**		Reposition a file pointer in a stream.
`getc`	**`getchar`**	**`fgetc`**	**`getw`**	Get a character or word from a stream.
`gets`	**`fgets`**			Get a string from a stream.
`popen`	**`pclose`**			Begin or end a pipe to/from a process.
`printf`	**`fprintf`**	**`sprintf`**		Print formatted output.
`putc`	**`putchar`**	**`fputc`**	**`putw`**	Put a character or word on a stream
`puts`	**`fputs`**			Put a string on a stream.
`scanf`	**`fscanf`**	**`sscanf`**		Convert formatted input.
`setbuf`	**`setvbuf`**			Assign buffering to a stream.
`system`				Issue a command through the shell.
`tmpfile`	**`tmpfile64`**			Create a temporary file.
`tmpnam`	**`tempnam`**			Create a name for a temporary file.
`ungetc`				Push character back into input stream.
`vprintf`	**`vfprintf`**	**`vsprintf`**		Print formatted output of a `varargs` argument list.

Math Library (libm)

The math library, **`libm`**, contains the mathematics functions supplied by the C compilation system. These appear in Subsection 3M of the *Operating System API Reference.* Here we describe some of the major functions, organized by the manual page on which they appear. Note that functions whose names end with the letter `f` are single-precision versions, which means that their argument and return types are `float`. Programs that use math functions should include the header file **`<math.h>`**.

The OS also provides an alternate math library:**`/usr/ccs/lib/libM.a`** Use of this library is recommended when the characteristics of the arguments are well-understood and higher performance is preferred to increased accuracy. It differs from the standard math library in the following ways:

- Arguments are not checked to ensure that they are valid IEEE floating-point numbers.
- Arguments are not checked for mathematical validity (for example, `sqrt(-2)`).
- For the single-precision functions, certain calculations that are performed in double precision in the standard library are performed in single precision in the alternate library. As a result, 1-bit errors can occur in some calculations.
- This alternate library uses large tables of constants as a repository of data for its calculations. Use of this library will require a larger address space than is needed with the standard library.

For additional information on use of the alternate math library, refer to *Compilation Systems Volume 1 (Tools).*

Table 2-1. Math Functions

`erf(3M)`		
`erf`		Compute the error function of *x*, defined as: $\frac{2}{\sqrt{\Pi}}\int_0^x e^{-t^2}dt$
`erfc`		Compute 1.0 - `erf(x)`, which is used because of the extreme loss of relative accuracy if **`erf`** is called for large *x* and the result subtracted from 1.0 (e.g., for *x* = 5, 12 places are lost).
`exp(3M)`		
`exp`	**`expf`**	Compute e^x.
`cbrt`		Compute the cube root of *x*.
`log`	**`logf`**	Compute the natural logarithm of *x*. The value of *x* must be positive.

Table 2-1. Math Functions (Cont.)

`log10`	`log10f`	Compute the base-ten logarithm of *x*. The value of *x* must be positive.
`pow`	`powf`	Compute x^y . If *x* is zero, *y* must be positive. If *x* is negative, *y* must be an integer.
`sqrt`	`sqrtf`	Compute the non-negative square root of *x*. The value of *x* must be non-negative.
`floor(3M)`		
`floor`	`floorf`	Compute the largest integer not greater than *x*.
`ceil`	`ceilf`	Compute the smallest integer not less than *x*.
`copysign`		Compute *x* but with the sign of *y*.
`fmod`	`fmodf`	Compute the floating point remainder of the division of *x* by *y*: *x* if *y* is zero, otherwise the number *f* with same sign as *x*, such that $x = iy + f$ for some integer *i*, and $\lvert f\rvert < \lvert y\rvert$.
`fabs`	`fabsf`	Compute $\lvert x\rvert$, the absolute value of *x*.
`rint`		Compute as a double-precision floating point number the integer value nearest the double-precision floating point argument *x*, and rounds the return value according to the currently set machine rounding mode.
`remainder`		Compute the floating point remainder of the division of *x* by *y*: `NaN` if y is zero, otherwise the value $r = x - yn$, where *n* is the integer value nearest the exact value of x/y , and *n* is even whenever $\lvert n - x/y\rvert = 1/2$.
`gamma(3M)`		
`gamma`	`lgamma`	Compute $\ln\lvert\Gamma(x)\rvert$, where $\Gamma(x)$ is defined as $\int_0^x e^{-t}t^{x-1}dt$
`hypot(3M)`		
`hypot`		Compute $\sqrt{x^2 + y^2}$, taking precautions against overflows.
`matherr(3)`		
`matherr`		Error handling.
`trig(3M)`		
`sin`	`sinf`	Compute the sine of *x*, measured in radians.
`cos`	`cosf`	Compute the cosine of *x*, measured in radians.
`tan`	`tanf`	Compute the tangent of *x*, measured in radians.

Table 2-1. Math Functions (Cont.)

asin	**asinf**	Compute the arcsine of *x*, in the range $[-\Pi/2,\Pi/2]$.
acos	**acosf**	Compute the arccosine of *x*, in the range $[0,\Pi]$.
atan	**atanf**	Compute the arctangent of *x*, in the range $(-\Pi/2,\Pi/2)$.
atan2	**atan2f**	Compute the arctangent of y/x, in the range $(-\Pi, \Pi]$, using the signs of both arguments to determine the quadrant of the return value.
sinh(3M)		
sinh	**sinhf**	Compute the hyperbolic sine of *x*.
cosh	**coshf**	Compute the hyperbolic cosine of *x*.
tanh	**tanhf**	Compute the hyperbolic tangent of *x*.
asinh		Compute the inverse hyperbolic sine of *x*.
acosh		Compute the inverse hyperbolic cosine of *x*.
atanh		Compute the inverse hyperbolic tangent of *x*.

General Purpose Library (libgen)

libgen contains general purpose functions, and functions designed to facilitate internationalization. These appear in Subsection 3G of the *Operating System API Reference.* Table 2-2 describes functions in **libgen**. The header files **<libgen.h>** and, occasionally, **<regexp.h>** should be included in programs that use these functions.

Table 2-2. libgen Functions

advance	**step**	Execute a regular expression on a string.
basename		Return a pointer to the last element of a path name.
bgets		Read a specified number of characters into a buffer from a stream until a specified character is reached.
bufsplit		Split the buffer into fields delimited by tabs and new-lines.
compile		Return a pointer to a compiled regular expression that uses the same syntax as **ed**.
copylist		Copy a file into a block of memory, replacing new-lines with null characters. It returns a pointer to the copy.
dirname		Return a pointer to the parent directory name of the file path name.

Table 2-2. libgen Functions (Cont.)

eaccess		Determine if the effective user ID has the appropriate permissions on a file.
gmatch		Check if name matches shell file name pattern.
isencrypt		Use heuristics to determine if contents of a character buffer are encrypted.
mkdirp		Create a directory and its parents.
p2open	**p2close**	**p2open** is similar to **popen** (see **popen(3S)**). It establishes a two-way connection between the parent and the child. **p2close** closes the pipe.
pathfind		Search the directories in a given path for a named file with given mode characteristics. If the file is found, a pointer is returned to a string that corresponds to the path name of the file. A null pointer is returned if no file is found.
regcmp		Compile a regular expression and return a pointer to the compiled form.
regex		Compare a compiled regular expression against a subject string.
rmdirp		Remove the directories in the specified path.
strccpy	**strcadd**	**strccpy** copies the input string to the output string, compressing any C-like escape sequences to the real character. **strcadd** is a similar function that returns the address of the null byte at the end of the output string.
strecpy		Copy the input string to the output string, expanding any non-graphic characters with the C escape sequence. Characters in a third argument are not expanded.
strfind		Return the offset of the first occurrence of the second string in the first string. `-1` is returned if the second string does not occur in the first.
strrspn		Trim trailing characters from a string. It returns a pointer to the last character in the string not in a list of trailing characters.
strtrns		Return a pointer to the string that results from replacing any character found in two strings with a character from a third string. This function is similar to the **tr** command.

Standard I/O Library

The functions in Subsection 3S of the *Operating System API Reference* constitute the standard I/O library for C programs. In this section, we want to discuss standard I/O in a bit more detail. First, let's briefly define what I/O involves. It has to do with

- reading information from a file or device to your program;
- writing information from your program to a file or device;
- opening and closing files that your program reads from or writes to.

Three Files You Always Have

Programs automatically start off with three open files: standard input, standard output, and standard error. These files with their associated buffering are called streams, and are designated **`stdin`**, **`stdout`**, and **`stderr`**, respectively. The shell associates all three files with your terminal by default.

This means that you can use functions and macros that deal with **`stdin`**, **`stdout`**, or **`stderr`** without having to open or close files. **`gets`**, for example, reads a string from **`stdin`**; **`puts`** writes a string to **`stdout`**. Other functions and macros read from or write to files in different ways: character at a time, **`getc`** and **`putc`**; formatted, **`scanf`** and **`printf`**; and so on. You can specify that output be directed to **`stderr`** by using a function such as **`fprintf`**. **`fprintf`** works the same way as **`printf`** except that it delivers its formatted output to a named stream, such as **`stderr`**.

Named Files

Any file other than standard input, standard output, and standard error must be explicitly opened by you before your program can read from or write to the file. You open a file with the standard library function **`fopen`**. **`fopen`** takes a path name, asks the system to keep track of the connection between your program and the file, and returns a pointer that you can then use in functions that perform other I/O operations.

The pointer is to a structure called `FILE`, defined in **`<stdio.h>`**, that contains information about the file: the location of its buffer, the current character position in the buffer, and so on. In your program, then, you need to have a declaration such as

```
FILE *fin;
```

which says that `fin` is a pointer to a `FILE`. The statement

```
fin = fopen(“filename”, “r”);
```

associates a `FILE` structure with `filename`, the path name of the file to open, and returns a pointer to it. The `“r”` means that the file is to be opened for reading. This argument is known as the mode. There are modes for reading, writing, and both reading and writing.

In practice, the file open function is often included in an `if` statement:

```
if ((fin = fopen(“filename”, “r”)) == NULL)
    (void)fprintf(stderr,“Cannot open input file %s\n”,
        “filename”);
```

which takes advantage of the fact that **`fopen`** returns a `NULL` pointer if it cannot open the file. To avoid falling into the immediately following code on failure, you can call **`exit`**, which causes your program to quit:

```
if ((fin = fopen(“filename”, “r”)) == NULL) {
    (void)fprintf(stderr,“Cannot open input file %s\n”,
        “filename”);
    exit(1);
}
```

If the file to be opened is larger than 2GB, or is likely to grow to be that size, you should use **`fopen64`**. See “Subsection 3S Routines” on page 2-9 for other Standard I/O functions for use with large files.

Once you have opened the file, you use the pointer `fin` in functions or macros to refer to the stream associated with the opened file:

```
int c;
c = getc(fin);
```

brings in one character from the stream into an integer variable called `c`. The variable `c` is declared as an integer even though we are reading characters because **`getc`** returns an integer. Getting a character is often incorporated in some flow-of-control mechanism such as

```
while ((c = getc(fin)) != EOF)
     .
     .
     .
```

that reads through the file until `EOF` is returned. `EOF`, `NULL`, and the macro getc are all defined in **`<stdio.h>`**. **`getc`** and other macros in the standard I/O package keep advancing a pointer through the buffer associated with the stream; the UNIX system and the standard I/O functions are responsible for seeing that the buffer is refilled if you are reading the file, or written to the output file if you are producing output, when the pointer reaches the end of the buffer.

Your program may have multiple files open simultaneously, 20 or more depending on system configuration. If, subsequently, your program needs to open more files than it is permitted to have open simultaneously, you can use the standard library function **`fclose`** to break the connection between the `FILE` structure in **`<stdio.h>`** and the path names of the files your program has opened. Pointers to `FILE` may then be associated with other files by subsequent calls to **`fopen`**. For output files, an **`fclose`** call makes sure that all output has been sent from the output buffer before disconnecting the file. **`exit`** closes all open files for you, but it also gets you completely out of your process, so you should use it only when you are sure you are finished.

How C Programs Communicate with the Shell

Information or control data can be passed to a C program as an argument on the command line, which is to say, by the shell. When you execute a C program, command line arguments are made available to the function **main** in two parameters, an argument count, conventionally called `argc`, and an argument vector, conventionally called `argv`. (Every C program is required to have an entry point named **main**.) `argc` is the number of arguments with which the program was invoked. `argv` is an array of pointers to character strings that contain the arguments, one per string. Since the command name itself is considered to be the first argument, or `argv[0]`, the count is always at least one. Here is the declaration for **main**:

```
int
main(int argc, char *argv[])
```

For two examples of how you might use run-time parameters in your program, see the last subsection of this chapter.

The shell, which makes arguments available to your program, considers an argument to be any sequence of non-blank characters. Characters enclosed in single quotes (`'abc def'`) or double quotes (`“abc def”`) are passed to the program as one argument even if blanks or tabs are among the characters. You are responsible for error checking and otherwise making sure that the argument received is what your program expects it to be.

In addition to `argc` and `argv`, you can use a third argument: `envp` is an array of pointers to environment variables. You can find more information on `envp` in the *Operating System API Reference* under **exec** in Section 2 and in the *System Files and Devices Reference* under `environ` in Section 5.

C programs exit voluntarily, returning control to the operating system, by returning from main or by calling the exit function. That is, a **return(***n***)** from main is equivalent to the call **exit(***n***)**. (Remember that main has type “function returning `int`.”) Your program should return a value to say whether it completed successfully or not. The value gets passed to the shell, where it becomes the value of the `$?` shell variable if you executed your program in the foreground. By convention, a return value of zero denotes success, a non-zero return value means some sort of error occurred. You can use the macros `EXIT_SUCCESS` and `EXIT_FAILURE`, defined in the header file **<stdlib.h>**, as return values from main or argument values for **exit**.

Passing Command Line Arguments

As described above, information or control data can be passed to a C program as an argument on the command line. When you execute the program, command line arguments are made available to the function **main** in two parameters, an argument count, conventionally called `argc`, and an argument vector, conventionally called `argv`. `argc` is the number of arguments with which the program was invoked. `argv` is an array of pointers to characters strings that contain the arguments, one per string. Since the command name itself is considered to be the first argument, or `argv[0]`, the count is always at least one.

If you plan to accept run-time parameters in your program, you need to include code to deal with the information. and show program fragments that illustrate two common uses of run-time parameters:

- Figure 2-8 shows how you provide a variable file name to a program, such that a command of the form

  $ **prog** *filename*

  will cause **prog** to attempt to open the specified file.

- Figure 2-9shows how you set internal flags that control the operation of a program, such that a command of the form

  $ **prog -opr**

  will cause **prog** to set the corresponding variables for each of the options specified. The **getopt** function used in the example is the most common way to process arguments in UNIX system programs. **getopt** is described in Subsection 3C of the *Operating System API Reference.*

```
#include <stdio.h>

int
main(int argc, char *argv[])
{
      FILE *fin;
      int ch;

      switch (argc)
      {
      case 2:
            if ((fin = fopen(argv[1], ”r”)) == NULL)
            {
                  /* First string (%s) is program name (argv[0]).  */
                  /* Second string (%s) is name of file that could */
                  /* not be opened (argv[1]). */

                  (void)fprintf(stderr, ”%s: Cannot open input file %s\n”,
                        argv[0], argv[1]);
                  return(2);
            }
            break;
      case 1:
            fin = stdin;
            break;

      default:
            (void)fprintf(stderr, ”Usage: %s [file]\n”, argv[0]);
            return(2);
      }

      while ((ch = getc(fin)) != EOF)
            (void)putchar(ch);

      return (0);

}
```

Figure 2-8. Using argv[1] to Pass a File Name

```
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
      int oflag = 0;
      int pflag = 0;/* Function flags */
      int rflag = 0;
      int ch;

      while ((ch = getopt(argc, argv, “opr”)) != -1)
      {
            /* For options present, set flag to 1.              */
            /* If unknown options present, print error message. */

            switch (ch)
            {
            case 'o':
                  oflag = 1;
                  break;
            case 'p':
                  pflag = 1;
                  break;
            case 'r':
                  rflag = 1;
                  break;
            default:
                  (void)fprintf(stderr, “Usage: %s [-opr]\n”, argv[0]);
                  return(2);
            }
      }
      /* Do other processing controlled by oflag, pflag, rflag. */
      return(0);
}
```

Figure 2-9. Using Command Line Arguments to Set Flags

System Calls

System calls are the interface between the kernel and the user programs that run on top of it. The kernel is the software on which everything else in the UNIX operating system depends. The kernel manages system resources, maintains file systems and supports system calls. **`read`**, **`write`** and the other system calls in Section 2 of the *Operating System API Reference* define what the UNIX system is. Everything else is built on their foundation. Strictly speaking, they are the only way to access such facilities as the file system, interprocess communication primitives, and multitasking mechanisms.

Of course, most programs do not need to invoke system calls directly to gain access to these facilities. If you are writing a C program, for example, you can use the library functions described in Section 3 of the *Operating System API Reference.* When you use these functions, the details of their implementation on the system are transparent to the program, for example, that the system call read underlies the **`fread`** implementation in the standard C library. In other words, the program will generally be portable to any system, UNIX or not, with a conforming C implementation. (See the *Concurrent C Reference Manual* guide for a discussion of the standard C library.)

In contrast, programs that invoke system calls directly are portable only to other UNIX systems or systems that are similar to UNIX systems; for that reason, you would not use **`read`** in a program that performed a simple input/output operation. Other operations, however, including most multitasking mechanisms, do require direct interaction with the UNIX system kernel. These operations are the subject of the first part of this book. This chapter lists the system calls in functional groups, and includes brief discussions of error handling. For details on individual system calls, see Section 2 of the *Operating System API Reference.*

A C program is automatically linked with the system calls you have invoked when you compile the program. The procedure may be different for programs written in other languages. Check the *Concurrent C Reference Manual* guide for details on the language you are using.

Input/Output and File System Calls

File and Device I/O

These system calls perform basic input/output operations on system files.

Table 2-3. File and Device I/O Functions

Function Name(s)		Purpose
`open`		open a file for reading or writing
`open64`		open a large file
`creat`		create a new file or rewrite an existing one
`creat64`		create a large file
`close`		close a file descriptor
`read`	**`write`**	transfer data from/onto a file or device
`getmsg`	**`putmsg`**	get/put message from/onto a stream
`lseek`		move file I/O pointer
`lseek64`		move file I/O pointer of a large file
`fcntl`		file I/O control
`ioctl`		device I/O control
`truncate64`	**`ftruncate64`**	set a large file to a specified length

Terminal Device Control

These system calls deal with a general terminal interface for the control of asynchronous communications ports.

Table 2-4. Terminal Device Control Functions

Function Name(s)		Purpose
tcgetattr	**tcsetattr**	get and set terminal attributes
tcdrain	**tcflush**	line control functions
tcflow	**tcsendbreak**	line control functions
cfgetispeed	**cfgetospeed**	get baud rate functions
cfsetispeed	**cfsetospeed**	set baud rate functions
tcgetsid		get terminal session ID
tcgetpgrp		get terminal foreground process group ID
tcsetpgrp		set terminal foreground process group ID

Directory and File System Control

These system calls allow creation of new directories (and other types of files), linking to existing files, obtaining or modifying file status information, and allow you to control various aspects of the file system.

Table 2-5. Directory and File System Control Functions

Function Name(s)			Purpose
link			link to a file
access			determine accessibility of a file
mknod			make a directory, special, or regular file
chmod	**fchmod**		change mode of file
chown	**fchown**	**lchown**	change owner and group of a file
utime	**lutime**		set file access and modification times
stat	**fstat**	**lstat**	get file status
pathconf	**fpathconf**		get configurable path name variables
getdents			read directory entries and put in file system-independent format
mkdir			make a directory
readlink			read the value of a symbolic link
rename			change the name of a file
rmdir			remove a directory
symlink			make a symbolic link to a file

Table 2-5. Directory and File System Control Functions (Cont.)

Function Name(s)			Purpose
`unlink`			remove directory entry
`ustat`			get file system statistics
`sync`			update super block
`mount`	**`umount`**		mount/unmount a file system
`statfs`	**`fstatfs`**		get file system information
`stat64`	**`fstat64`**	**`lstat64`**	get large file status
`sysfs`			get file system type information

Access Control System Calls

These system calls are used to obtain or modify security level information that is used by the system to mediate access control when the Enhanced Security Utilities are installed and running.

Only privileged processes can modify file or process security levels. See the individual system call manual pages and the **`intro(2)`** manual page in the *Operating System API Reference* for a description of the various privileges required by the system calls, and the effect of Mandatory Access Control levels on the access checking algorithm used by the system.

Table 2-6. Mandatory Access Control (MAC) System Calls

Function Name(s)	Purpose
`devstat, fdevstat`	get or set the security attributes of a device
`lvldom`	determine domination relationship of two levels
`lvlequal`	determine if two levels are equal
`lvlfile`	get or set the level of a directory, a named pipe or a regular or special file
`lvlipc`	manipulate an IPC object's level
`lvlproc`	get or set the level of the calling process
`lvlvfs`	get or set the level ceiling of a mounted file system
`mkmld`	create a multilevel directory
`mldmode`	get or set the mld mode of the calling process
`secadvise`	get kernel advisory access information

Process and Memory System Calls

Processes

These system calls control user processes.

Table 2-7. Process Management Functions

Function Name(s)			Purpose
fork			create a new process
execl	**execle**	**execlp**	execute a file with a list of arguments
execv	**execve**	**execvp**	execute a file with a variable list
exit	**_exit**		terminate process
wait	**waitpid**	**waitid**	wait for child process to change state
cpu_bias			get and set LWP's CPU bias or assignment
setuid	**setgid**		set user and group IDs
getpgrp	**setpgrp**		get and set process group ID
chdir	**fchdir**		change working directory
chroot			change root directory
nice			change priority of a process
getcontext	**setcontext**		get and set current user context
getgroups	**setgroups**		get or set supplementary group IDs
getpid	**getppid**	**getpgid**	get process and parent process IDs
getuid	**geteuid**		get real user and effective user
getgid	**getegid**		get real group and effective group
_lwp_global_self			get current LWP's global ID
pause			suspend process until signal
priocntl			process scheduler control

Table 2-7. Process Management Functions (Cont.)

Function Name(s)	Purpose
setpgid	set process group ID
setsid	set session ID
kill	send a signal to a process or group of processes

Signals

Signals are messages passed by the system to running processes.

Figure 2-10. Signal Management Functions

Function Name(s)		Purpose
sigaltstack		set/get signal alternate stack context
sigignore	**sigpause**	simplified signal management
sighold	**sigrelse**	simplified signal management
sigset	**signal**	simplified signal management
sigpending		examine blocked and pending signals
sigprocmask		change or examine signal mask
sigqueue		queue a signal to a process
sigsuspend		install a signal mask and suspend process
sigsend	**sigsendset**	send a signal to a process or group of processes

Basic Interprocess Communication

These system calls connect processes so they can communicate. `pipe` is the system call for creating an interprocess channel. `dup` is the call for duplicating an open file descriptor. (These IPC mechanisms are not applicable for processes on separate hosts.)

Figure 2-11. Basic Interprocess Communication Functions

Function Name(s)	Purpose
pipe	open file-descriptors for a pipe
dup	duplicate an open file-descriptor

Advanced Interprocess Communication

Some of these system calls support System V IPC messages, semaphores, and shared memory and are effective in data base management. (These IPC mechanisms are also not applicable for processes on separate hosts.) Others are interprocess synchronization tools that are provided by the OS for use in synchronizing cooperating processes' access to data in shared memory. For additional information on the System V IPC mechanisms, refer to Chapter 12. For an explanation of the interprocess synchronization tools, refer to the *PowerMAX OS Real-Time Guide.*

Figure 2-12. Advanced Interprocess Communication Functions

Function Name(s)	Purpose
msgget	get message queue
msgctl	message control operations
msgop	message operations
semget	get set of semaphores
semctl	semaphore control operations
semop	semaphore operations
shmget	get shared memory segment identifier
shmctl	shared memory control operations
shmop	shared memory operations
resched_cntl	rescheduling control operations
server_block	block the calling LWP only if no wake-up request has occurred since last return from **server_block**
server_wake1	wake a single server blocked in a **server_block** system call
server_wakevec	wake a group of servers blocked in a **server_block** system call
client_block	block the calling LWP (a client) and pass its priority to another LWP (a server)
client_wake1	wake a single client blocked in a **client_block** system call
client_wakechan	wake a group of clients blocked in a **client_block** system call

Memory Management

These system calls give you access to virtual memory facilities.

Figure 2-13. Memory Management Functions

Function Name(s)		Purpose
getpagesize		get system page size
memcntl		memory management control
mmap		map pages of memory
mmap64		map pages of memory for a large file
mprotect		set protection of memory mapping
munmap		unmap pages of memory
plock		lock process, text, or data in memory
userdma		prepare a buffer for DMA transfers
brk	**sbrk**	dynamically allocate memory space

Miscellaneous System Calls

These are system calls for such things as administration, timing, and other miscellaneous purposes.

Figure 2-14. Miscellaneous System Functions

Function Name(s)		Purpose
acct		enable or disable process accounting
alarm		set a process alarm clock
getrlimit	**setrlimit**	control maximum system resource consumption
getrlimit64	**setrlimit64**	control large file size limit
hrdclk		control hardclock interrupt handling
modload		loads dynamically loadable kernel module
moduload		unloads kernel module
modpath		change path from which modules are loaded
modadm		module administration
profil		execution time profile
sysconf		method for application's determination of value for system configuration

Figure 2-14. Miscellaneous System Functions (Cont.)

Function Name(s)		Purpose
syscx		machine-specific functions (available only on PowerMAX OS)
time	**stime**	get/set time
uadmin		administrative control
ulimit		get and set user limits
uname		get/set name of current system

System Call Error Handling

System calls that fail to complete successfully almost always return a value of `-1` to your program. (If you look through the system calls in Section 2, you will see that there are a few calls for which no return value is defined, but they are the exceptions.) In addition to the `-1` returned to the program, the unsuccessful system call places an integer in an externally declared variable, `errno`. In a C program, you can determine the value in `errno` if your program contains the following statement:

```
#include <errno.h>
```

The C language function **perror(3C)** can be used to print an error message (on **stderr**) based on the value of `errno`. The value in `errno` is not cleared on successful calls, so your program should check it only if the system call returned a `-1` indicating an error. Table 2-8 identifies the symbolic names defined in the **<errno.h>** header file and described in **intro(2)** of the *Operating System API Reference.*

Table 2-8. errno Values

Symbolic Name	Description
ENOENT	No such file or directory A file name is specified and the file should exist but fails to, or one of the directories in a path name fails to exist.
ESRCH	No such process No process can be found corresponding to the that specified by PID in the **kill** or **ptrace** routine.
EINTR	Interrupted system call An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system service routine. If execution is resumed after processing the signal, it will appear as if the interrupted routine call returned this error condition.

Table 2-8. errno Values (Cont.)

Symbolic Name	Description
`EIO`	I/O error Some physical I/O error has occurred. This error may in some cases occur on a call following the one to which it actually applies.
`ENXIO`	No such device or address I/O on a special file refers to a subdevice which does not exist, or exists beyond the limit of the device. It may also occur when, for example, a tape drive is not on-line or no disk pack is loaded on a drive.
`E2BIG`	Arg list too long An argument list longer than `ARG_MAX` bytes is presented to a member of the **`exec`** family of routines. The argument list limit is sum of the size of the argument list plus the size of the environment's exported shell variables.
`ENOEXEC`	Exec format error A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid format (see **`a.out(4)`**).
`EBADF`	Bad file number Either a file descriptor refers to no open file, or a **`read`** (respectively, **`write`**) request is made to a file that is open only for writing (respectively, reading).
`ECHILD`	No child processes A **`wait`** routine was executed by a process that had no existing or unwaited-for child processes.
`EAGAIN`	Resource is temporarily unavailable For example, the **`fork`** routine failed because the system's process table is full or the user is not allowed to create any more processes. Or a system call failed because of insufficient memory or swap space.
`ENOMEM`	Not enough space During execution of an **`exec`**, **`brk`**, or **`sbrk`** routine, a program asks for more space than the system is able to supply. This is not a temporary condition; the maximum size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers, or if there is not enough swap space during the **`fork`** routine.
`EACCES`	Permission denied An attempt was made to access a file in a way forbidden by the protection system.

Table 2-8. errno Values (Cont.)

Symbolic Name	Description
`EFAULT`	Bad address The system encountered a hardware fault in attempting to use an argument of a routine. For example, `errno` potentially may be set to `EFAULT` any time a routine that takes a pointer argument is passed an invalid address, if the system can detect the condition. Because systems will differ in their ability to reliably detect a bad address, on some implementations passing a bad address to a routine will result in undefined behavior.
`ENOTBLK`	Block device required A non-block file was mentioned where a block device was required (e.g., in a call to the **`mount`** routine).
`EBUSY`	Device busy An attempt was made to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, active text segment). It will also occur if an attempt is made to enable accounting when it is already enabled. The device or resource is currently unavailable.
`EEXIST`	File exists An existing file was mentioned in an inappropriate context (e.g., call to the **`link`** routine).
`EXDEV`	Cross-device link A link to a file on another device was attempted.
`ENODEV`	No such device An attempt was made to apply an inappropriate operation to a device (e.g., read a write-only device).
`ENOTDIR`	Not a directory A non-directory was specified where a directory is required (e.g., in a path prefix or as an argument to the **`chdir`** routine).
`EISDIR`	Is a directory An attempt was made to write on a directory.
`EINVAL`	Invalid argument An invalid argument was specified (e.g., unmounting a non-mounted device, mentioning an undefined signal in a call to the **`signal`** or **`kill`** routine. Also set by the functions described in the math package **`(3M)`**.
`ENFILE`	File table overflow The system file table is full (i.e., `SYS_OPEN` files are open, and temporarily no more files can be opened).
`EMFILE`	Too many open files No process may have more than `OPEN_MAX` file descriptors open at a time.

Table 2-8. errno Values (Cont.)

Symbolic Name	Description
ENOTTY	Not a typewriter A call was made to the **ioctl** routine specifying a file that is not a special character device.
ETXTBSY	Text file busy An attempt was made to execute a pure-procedure program that is currently open for writing. Also an attempt to open for writing or to remove a pure-procedure program that is being executed.
EFBIG	File too large The size of a file exceeded the maximum file size, FCHR_MAX (see **getrlimit(2)**).
ENOSPC	No space left on device While writing an ordinary file or creating a directory entry, there is no free space left on the device. In the **fcntl** routine, the setting or removing of record locks on a file cannot be accomplished because there are no more record entries left on the system.
ESPIPE	Illegal seek A call to the **lseek** routine was issued to a pipe.
EROFS	Read-only file system An attempt to modify a file or directory was made on a device mounted read-only.
EMLINK	Too many links An attempt to make more than the maximum number of links, LINK_MAX, to a file.
EPIPE	Broken pipe A write on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.
EDOM	Math argument out of domain of func The argument of a function in the math package **(3M)** is out of the domain of the function.
ERANGE	Math result not representable The value of a function in the math package (3M) is not representable within machine precision.
ENOMSG	No message of desired type An attempt was made to receive a message of a type not existing on the specified message queue (see **msgop(2)**).
EIDRM	Identifier removed This error is returned to processes that resume execution due to the removal of an identifier from the file system's name space (see **msgctl(2)**, **semctl(2)**, and **shmctl(2)**).
ECHRNG	Channel number out of range

Table 2-8. errno Values (Cont.)

Symbolic Name	Description
`EL2NSYNC`	Level 2 not synchronized
`EL3HLT`	Level 3 halted
`EL3RST`	Level 3 reset
`ELNRNG`	Link number out of range
`EUNATCH`	Protocol driver not attached
`ENOCSI`	No CSI structure available
`EL2HLT`	Level 2 halted
`EDEADLK`	Deadlock condition A deadlock situation was detected and avoided. This error pertains to file and record locking.
`ENOLCK`	No record locks available There are no more locks available. The system lock table is full (see **`fcntl(2)`**).
`ENOSTR`	Device not a stream. A `putmsg` or `getmsg` system call was attempted on a file descriptor that is not a STREAMS device.
`ENODATA`	No data available
`ETIME`	Timer expired The timer set for a STREAMS **`ioctl`** call has expired. The cause of this error is device specific and could indicate either a hardware or software failure, or perhaps a timeout value that is too short for the specific operation. The status of the **`ioctl`** operation is indeterminate.
`ENOSR`	Out of stream resources During a STREAMS `open`, either no STREAMS queues or no STREAMS head data structures were available. This is a temporary condition; one may recover from it if other processes release resources.
`ENOPKG`	Package not installed This error occurs when users attempt to use a system call from a package which has not been installed.
`EREMOTE`	Object is remote This error occurs when users try to advertise a resource that is not on the local machine or try to mount/unmount a device (or path name) that is on a remote machine.
`ENOLINK`	Link has been severed This error occurs when the link (virtual circuit) connecting to a remote machine is gone.
`EPROTO`	Protocol error Some protocol error occurred. This error is device specific, but is generally not related to a hardware failure.

Table 2-8. errno Values (Cont.)

Symbolic Name	Description
EBADMSG	Not a data message During a **read**, **getmsg**, or **ioctl I_RECVFD** system call to a STREAMS device, something has come to the head of the queue that can't be processed. That something depends on the system call: **read**—control information or a passed file descriptor, **getmsg**—passed file descriptor, **ioctl**—control or data information.
ENAMETOOLONG	File name too long The length of the path argument exceeds PATH_MAX, or the length of a path component exceeds NAME_MAX while _POSIX_NO_TRUNC is in effect; (see **limits(4)**).
EOVERFLOW	Value too large to be stored in data type. Attempting a **stat** or **lseek** using SEEK_END on a file whose size is too large to fit in a long integer. Other operations on large files may also return this value if the large file system calls are not used.
ENOTUNIQ	Name not unique on network Given log name not unique.
EBADFD	File descriptor in bad state. Either a file descriptor refers to no open file or a read request was made to a file that is open only for writing.
EREMCHG	Remote address changed.
ELIBACC	Cannot access a needed shared library Trying to **exec** an **a.out** that requires a shared library and the shared library doesn't exist or the user doesn't have permission to use it.
ELIBBAD	Accessing a corrupted shared library Trying to **exec** an **a.out** that requires a shared library (to be linked in) and **exec** could not load the shared library. The shared library is probably corrupted.
ELIBSCN	.lib section in **a.out** corrupted Trying to **exec** an **a.out** that requires a shared library (to be linked in) and there was erroneous data in the .lib section of the **a.out**. The .lib section tells **exec** what shared libraries are needed. The **a.out** is probably corrupted.
ELIBMAX	Attempting to link in more shared libraries than system limit Trying to **exec** an **a.out** that requires more static shared libraries than is allowed on the current configuration of the system. See the *System Administration, Volume 1*.
ELIBEXEC	Cannot exec a shared library directly Attempting to **exec** a shared library directly.
EILSEQ	Illegal byte sequence. Handle multiple characters as a single character.

Table 2-8. errno Values (Cont.)

Symbolic Name	Description
`ENOSYS`	Operation not applicable
`ELOOP`	Number of symbolic links encountered during path name traversal exceeds **`MAXSYMLINKS`**
`ERESTART`	Interrupted system call should be restarted.
`ESTRPIPE`	Streams pipe error (not externally visible).
`ENOTEMPTY`	Directory not empty
`EUSERS`	Too many users.
`ENOTSOCK`	Socket operation on non-socket Self-explanatory.
`EDESTADDRREQ`	Destination address required A required address was omitted from an operation on a transport endpoint. Destination address required.
`EMSGSIZE`	Message too long A message sent on a transport provider was larger than the internal message buffer or some other network limit.
`EPROTOTYPE`	Protocol wrong type for socket. A protocol was specified that does not support the semantics of the socket type requested.
`ENOPROTOOPT`	Protocol not available A bad option or level was specified when getting or setting options for a protocol.
`EPROTONOSUPPORT`	Protocol not supported The protocol has not been configured into the system or no implementation for it exists.
`ESOCKTNOSUPPORT`	Socket type not supported The support for the socket type has not been configured into the system or no implementation for it exists.
`EOPNOTSUPP`	Operation not supported on transport endpoint For example, trying to accept a connection on a datagram transport endpoint.
`EPFNOSUPPORT`	Protocol family not supported The protocol family has not been configured into the system or no implementation for it exists. Used for the Internet protocols.
`EAFNOSUPPORT`	Address family not supported by protocol family An address incompatible with the requested protocol was used.
`EADDRINUSE`	Address already in use User attempted to use an address already in use, and the protocol does not allow this.

Table 2-8. errno Values (Cont.)

Symbolic Name	Description
`EADDRNOTAVAIL`	Cannot assign requested address Results from an attempt to create a transport endpoint with an address not on the current machine.
`ENETDOWN`	Network is down Operation encountered a dead network.
`ENETUNREACH`	Network is unreachable Operation was attempted to an unreachable network.
`ENETRESET`	Network dropped connection because of reset The host you were connected to crashed and rebooted.
`ECONNABORTED`	Software caused connection abort A connection abort was caused internal to your host machine.
`ECONNRESET`	Connection reset by peer A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote host due to a timeout or a reboot.
`ENOBUFS`	No buffer space available An operation on a transport endpoint or pipe was not performed because the system lacked sufficient buffer space or because a queue was full.
`EISCONN`	Transport endpoint is already connected A connect request was made on an already connected transport endpoint; or, a `sendto` or `sendmsg` request on a connected transport endpoint specified a destination when already connected.
`ENOTCONN`	Transport endpoint is not connected A request to send or receive data was disallowed because the transport endpoint is not connected and (when sending a datagram) no address was supplied.
`ESHUTDOWN`	Cannot send after transport endpoint shutdown A request to send data was disallowed because the transport endpoint had already been shut down.
`ETOOMANYREFS`	Too many references: cannot splice
`ETIMEDOUT`	Connection timed out A connect or send request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.)
`ECONNREFUSED`	Connection refused No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the remote host.

Table 2-8. errno Values (Cont.)

Symbolic Name	Description
`EHOSTDOWN`	Host is down A transport provider operation failed because the destination host was down.
`EHOSTUNREACH`	No route to host A transport provider operation was attempted to an unreachable host.
`EALREADY`	Operation already in progress An operation was attempted on a non-blocking object that already had an operation in progress.
`EINPROGRESS`	Operation now in progress An operation that takes a long time to complete (such as a **`connect`**) was attempted on a non-blocking object.
`ESTALE`	Stale NFS file handle
`ENOLOAD`	Cannot load required module An attempt made to load a module failed.
`ERELOC`	Relocation error in loading module Symbolic referencing error.
`ENOMATCH`	No symbol is found matching the given spec
`EBADVER`	Version number mismatched The version number associated with a module is not supported by the kernel.
`ECONFIG`	Configured kernel resource exhausted
`EPERM`	Not superuser
`EBADE`	Invalid exchange
`EBADR`	Invalid request descriptor
`EXFULL`	Exchange full
`ENOANO`	No anode
`EBADRQC`	Invalid request code
`EBADSLT`	Invalid slot
`EDEADLOCK`	File locking deadlock error
`EBFONT`	Bad font file format
`ECLNRACE`	Non-clone open race with clone open
`EPROCLIM`	Too many processes
`EDQUOT`	Not superuser

Table 2-8. errno Values (Cont.)

Symbolic Name	Description
`ELKBUSY`	File/record lock request will block (only returned to NFS lock manager).
`EPOWERFAIL`	Power failure
`ECANCELED`	Asynchronous I/O request canceled

3
File and Device Input/Output

3
File and Device Input/Output

Introduction

This chapter discusses the UNIX system file and record locking facility. Mandatory and advisory file and record locking are both available on current releases of the UNIX system. The intent of this capability is to provide a synchronization mechanism for programs accessing the same stores of data simultaneously. Such processing is characteristic of many multiuser applications, and the need for a standard method of dealing with the problem has been recognized by standards advocates like **`/usr/group`**, an organization of UNIX System users from businesses and campuses across the country.

Advisory file and record locking can be used to coordinate self-synchronizing processes. In mandatory locking, the standard I/O subroutines and I/O system calls enforce the locking protocol. In this way, at the cost of a little efficiency, mandatory locking double checks the programs against accessing the data out of sequence.

Also included in this chapter is a description of how file and record locking capabilities can be used. Examples are given for the correct use of record locking. Misconceptions about the amount of protection that record locking affords are dispelled. Record locking should be viewed as a synchronization mechanism, not a security mechanism.

The remainder of this chapter describes the STREAMS mechanism as it relates to input/output operations.

Input/Output System Calls

The lowest level of I/O in provides no buffering or other such services, but it offers the most control over what happens. System calls that represent direct entries into the PowerMAX OS kernel control all user I/O. The OS keeps the system calls that do I/O simple, uniform and regular to eliminate differences between files, devices and styles of access. The same read and write system calls apply to ordinary disk files and I/O devices such as terminals, tape-drives and line-printers. They do not distinguish between "random" and "sequential" I/O, nor do they impose any logical record size on files. Thus, a single, uniform interface handles all communication between programs and peripheral devices, and programmers can defer specifying devices from program-development until program-execution time.

All I/O is done by reading or writing files, because all peripheral I/O devices, even a user's terminal, are files in the file system. Each supported device has an entry in the file system hierarchy, so that device names have the same structure as filenames, and the same protection mechanisms work on both devices and files.

A file is an ordered set of bytes of data on a I/O-device. The size of the file on input is determined by an end-of-file condition dependent on device-specific characteristics. The size of a regular file is determined by the position and number of bytes written on it, no predetermination of the size of a file is necessary or possible.

Besides the traditionally available devices, names exist for disk devices regarded as physical units outside the file system, and for absolutely addressed memory. The most important device in practice is the user's terminal. Treating a communication-device in the same way as any file by using the same I/O calls make it easy to redirect the input and output of commands from the terminal to another file; although, some differences are inevitable. For example, the OS ordinarily treats terminal input in units of lines because character-erase and line-delete processing cannot be completed until a full line is typed. Programs trying to read some large number of bytes from a terminal must wait until a full line is typed, and then may be notified that some smaller number of bytes were actually read. All programs must prepare for this eventuality in any case, because a read from any disk file returns fewer bytes than requested when it reaches the end of the file. Ordinarily, reads from a terminal are fully compatible with reads from a disk file.

File Descriptors

The OS File and Device I/O functions denote a file by a small positive integer called a “file-descriptor” and declared as follows:

```
int fildes
```

where `fildes` represents the file-descriptor, and the file-descriptor denotes an open file from which data is read or onto which data is written. The OS maintains all information about an open file; the user program refers to the file only by the file-descriptor. Any I/O on the file uses the file-descriptor instead of the filename to denote the file.

Multiple file-descriptors may denote the same file, and each file-descriptor has associated with it information used to do I/O on the file:

- a file-offset that shows which byte in the file to read or write next;
- file-status and access-modes (e.g., *read*, *write*, *read*/*write*) (see **`open(2)`**);
- the “close-on-exec” flag (see **`fcntl(2)`**).

Doing I/O on the user's terminal occurs commonly enough that special arrangements make this convenient. When the command interpreter (the “shell”) runs a program, it opens three files, called the standard input, the standard output and the standard error output, with file-descriptors `0`, `1` and `2`. All of these are normally connected to the terminal; thus, a program reading file-descriptor `0` and writing file-descriptors `1` and `2`, can do terminal I/O without opening the files. If I/O is redirected to and from files with `<` and `>`, as in:

```
prog <infile >outfile
```

the shell changes the default assignments for file-descriptors `0` and `1` from the terminal to the named files. Similar conventions hold for I/O on a pipe. Normally file-descriptor `2` remains attached to the terminal, so error messages can go there. In all cases, the shell changes the file assignments, the program does not. The program can ignore where its output goes, as long as it uses file-descriptor `0` for input and `1` and `2` for output.

Reading and Writing Files

The functions `read` and `write` do I/O on files. For both, the first argument is a file-descriptor, the second argument is a buffer in the user program where the data comes from or goes to and the third argument is the number of bytes of data to transfer. Each call returns a count of the number of bytes actually transferred. These calls look like:

> *n* = **`read`**(*fildes*, *buffer*, *count*);
> *n* = **`write`**(*fildes*, *buffer*, *count*);

Up to *count* bytes are transferred between the file denoted by *fildes* and the byte array pointed to by *buffer*. The returned value n is the number of bytes actually transferred.

For writing, the returned value is the number of bytes actually written; it is generally an error if this fails to equal the number of bytes requested. In the **`write`** case, *n* is the same as count except under exceptional conditions, such as I/O errors or end of physical medium on special files; in a **`read`**, however, *n* may without error be less than count.

For reading, the number of bytes returned may be less than the number requested, because fewer than count bytes remained to be read. If the file-offset is so near the end of the file that reading count characters would cause reading beyond the end, only sufficient bytes are transferred to reach the end of the file, also, typewriter-like terminals never return more than one line of input. (When the file is a terminal, **`read`** normally reads only up to the next new-line, which is generally less than what was requested.)

When a **`read`** call returns with *n* equal to zero, the end of the file has been reached. For disk files this occurs when the file-offset equals the current size of the file. It is possible to generate an end-of-file from a terminal by use of an escape sequence that depends on the device used. The function **`read`** returns `0` to signify end-of-file, and returns `-1` to signify an error.

The number of bytes to be read or written is quite arbitrary. The two most common values are `1`, which means one character at a time (“unbuffered”), and `512`, which corresponds to a physical block size on many peripheral devices. This latter size is most efficient, but even character at a time I/O is not overly expensive. Bytes written affect only those parts of a file implied by the position of the file-offset and the count; no other part of the file is changed. If the last byte lies beyond the end of the file, the file grows as needed.

A simple program using the **`read`** and **`write`** functions to copy its input to its output can copy anything, since the input and output can be redirected to any file or device.

```
#define BUFSIZE 512

main()   /* copy input to output */
{
   char buf[BUFSIZE];
   int  n;

   while ((n = read(0, buf, BUFSIZE)) > 0)
      write( 1, buf, n);
   exit(0);
}
```

If the file size is not a multiple of `BUFSIZE`, some **`read`** will return a smaller number of bytes to be written by **`write`**: the next call to **`read`** after that will return zero indicating end-of-file.

To see how **`read`** and **`write`** can be used to construct higher level functions like **`getchar`** and **`putchar`**, here is an example of **`getchar`** which does unbuffered input:

```
#define  CMASK   0377  /* for making char's > 0 */

getchar() /* unbuffered single character input */
{
   char c;

   return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

The variable `c` must be declared `char`, because **`read`** accepts a character pointer. The character returned must be masked with `0377` to ensure that it is positive; otherwise, sign extension may make it negative.

The second version of **`getchar`** does input in big chunks, and hands out the characters one at a time.

```
#define  CMASK   0377  /* for making char's > 0 */
#define  BUFSIZE  512

getchar()  /* buffered version */
{
   static char    buf[BUFSIZE];
   static char    *bufp = buf;
   static int     n = 0;

   if (n == 0)  {   /* buffer is empty */
      n = read(0, buf, BUFSIZE);
      bufp = buf;
   }
   return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

Opening, Creating and Closing Files

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. The functions that do this are: **open**, **open64**, **creat**, and **creat64** (see **open(2)**, **open64(2)**, **creat(2)**, and **creat64(2)** in the *Operating System API Reference*). To read or write a file assumed to exist already, it must be opened by the following call:

fildes = **open**(*name*, *oflag*);

The argument *name* is a character string that represents an OS file system pathname. The *oflag* argument indicates whether the file is to be read, written, or "updated", that is, read and written simultaneously. The returned value *fildes* is a file-descriptor used to denote the file in subsequent calls that read, write or otherwise manipulate the file. If **open** is used on a pre-existing large file, the operation fails and `errno` is set to `EOVERFLOW`. To open a large file, set `O_LARGEFILE` in *oflag* or use **open64**.

The function **open** resembles the function **fopen** in the Standard I/O Library, except that instead of returning a pointer to `FILE`, **open** returns a file-descriptor which is just an `int` (see **fopen(3S)** and **stdio(3S)** in the *Operating System API Reference*). Moreover, the values for the access mode argument *oflag* are different (the flags are found in **/usr/include/fcntl.h**):

- `O_RDONLY` for read access.
- `O_WRONLY` for write access.
- `O_RDWR` for read and write access.

The function `open` returns `-1` if any error occurs; otherwise it returns a valid open file-descriptor.

Trying to **open** a file that does not exist causes an error; hence, **creat** is used to create new files, or to re-write old ones. The **creat** system call creates the given file if it does not exist, or truncates it to zero length if it does exist; **creat** also opens the new file for writing and, like **open**, returns a file-descriptor. Calling **creat** as follows:

fildes = **creat**(*name*, *pmode*);

returns a file-descriptor if it created the file identified by the string *name*, and `-1` if it did not. Trying to **`creat`** a file that already exists does not cause an error, but if the file already exists, **`creat`** truncates it to zero length. If **`creat`** is used on a pre-existing large file, the operation fails and `errno` is set to `EOVERFLOW`. To create or re-write an existing large file, use **`creat64`**.

If the file is brand new, **`creat`** creates it with the protection mode specified by the `pmode` argument. The *OS* file system associates nine bits of protection information with a file, controlling *read*, *write* and *execute* permission for the *owner* of the file, for the owner's *group*, and for any *other* users. Thus, a three-digit octal number specifies the permissions most conveniently. For example, `0755` specifies *read*, *write* and *execute* permission for the *owner*, and *read* and *execute* permission for the *group* and all *other* users.

Figure 3-1 illustrates this with a simplified version of the OS utility **`cp`** (a program which copies one file to another):

```
#define  NULL 0
#define  BUFSIZE 512
#define  PMODE 0644 /* RW owner, R group & others */

main(argc, argv)    /* cp: copy fd1 to fd2 */
   int argc;
   char *argv[ ];
{
   int  fd1, fd2, n;
   char buf[BUFSIZE];

   if (argc != 3)
      error(“Usage: cp from to”, NULL);
   if ((fd1 = open(argv[1], 0)) == -1)
      error(“cp: can't open %s”, argv[1]);
   if ((fd2 = creat(argv[2], PMODE)) == -1)
      error(“cp: can't create %s”, argv[2]);

   while ((n = read(fd1, buf, BUFSIZE)) > 0)
      if (write(fd2, buf, n) != n)
         error(“cp: write error”, NULL);

   exit(0);
}

error(s1, s2)  /* print error message and die */
   char *s1, *s2;
{
   printf(s1, s2);
   printf(“\n”);

   exit(1);
}
```

Figure 3-1. Simplified Version of cp

The main simplification is that this version copies only one file, and does not permit the second argument to be a directory.

As stated earlier, there is a limit, `OPEN_MAX`, on the number of files which a process may have open simultaneously. Accordingly, any program which intends to process many files must be prepared to re-use file-descriptors. The function **`close`** breaks the connection between a file-descriptor and an open file, and frees the file-descriptor for use with some other file. Termination of a program via **`exit`** or return from the main program closes all open files.

Random Access — lseek

Normally, file I/O is sequential: each **`read`** or **`write`** proceeds from the point in the file right after the previous one. This means that if a particular byte in the file was the last byte written (or read), the next I/O call implicitly refers to the immediately following byte. For each open file, the OS maintains a file-offset that indicates the next byte to be read or written. If n bytes are read or written, the file-offset advances by n bytes. When necessary, however, a file can be read or written in any arbitrary order using **`lseek`** or **`lseek64`** (for files larger than 2GB) to move around in a file without actually reading or writing.

To do random (direct-access) I/O it is only necessary to move the file-offset to the appropriate location in the file with a call to **`lseek`**. Calling **`lseek`** as follows:

```
lseek(fildes, offset, whence);
```

or as follows:

```
location = lseek(fildes, offset, whence);
```

forces the current position in the file denoted by file-descriptor *fildes* to move to position offset as specified by *whence*. Subsequent reading or writing begins at the new position. To set the file offset beyond 2GB, **`lseek64`** should be used:

```
location = lseek64(fildes, offset, whence);
```

In this case location and offset should be declared as type `off64_t`.

The file-offset associated with *fildes* is moved to a position *offset* bytes from the beginning of the file, from the current position of the file-offset or from the end of the file, depending on *whence*; offset may be negative. For some devices (e.g., paper tape and terminals) **`lseek`** calls are ignored. The value of `location` equals the actual offset from the beginning of the file to which the file-offset was moved. The argument offset is of type `off_t` defined by the header file **`<types.h>`** as a `long`; fildes and whence are `int`'s. The argument whence can be `SEEK_SET`, `SEEK_CUR` or `SEEK_END` to specify that offset is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append a file, seek to the end before writing:

```
lseek(fildes, 0L, SEEK_END);
```

To get back to the beginning ("rewind"),

```
lseek(fildes, 0L, SEEK_SET);
```

Notice the `0L` argument; it could also be written as `(long) 0`.

If **`lseek`** is used, and the returned offset would overflow an item of type `off_t`, the error `EOVERFLOW` is returned. For example:

```
lseek(fildes, 0L, SEEK_END);
```

will fail with `errno` set to `EOVERFLOW` if the file is larger than 2GB.

With **`lseek`**, you can treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary point in a file:

```
get(fd, p, buf, n) /* read n bytes from position p */
   int fd, n;
   long p;
   char *buf;
{
   lseek(fd, p, SEEK_SET);  /* move to p */
   return(read(fd, buf, n));
}
```

File and Record Locking

Mandatory and advisory file and record locking are both available on current releases of the UNIX system. The intent of this capability to is provide a synchronization mechanism for programs accessing the same stores of data simultaneously. Such processing is characteristic of many multiuser applications, and the need for a standard method of dealing with the problem has been recognized by standards advocates like **/usr/group**, an organization of UNIX system users from businesses and campuses across the country.

Advisory file and record locking can be used to coordinate self-synchronizing processes. In mandatory locking, the standard I/O subroutines and I/O system calls enforce the locking protocol. In this way, at the cost of a little efficiency, mandatory locking double checks the programs against accessing the data out of sequence.

The remainder of this chapter describes how file and record locking capabilities can be used. Examples are given for the correct use of record locking. Misconceptions about the amount of protection that record locking affords are dispelled. Record locking should be viewed as a synchronization mechanism, not a security mechanism.

The manual pages for the **fcntl** system call, the **lockf** library function, and **fcntl** data structures and commands are referred to throughout this section (see **fcntl(5)**). You should read them before continuing.

An additional set of **fcntl** lock commands, `F_GETLK64`, `F_SETLK64`, and `F_SETLKW64`, is provided for use on large files.

To make use of these locking commands you must declare your lock structure as type `struct flock64`. The `l_start` and `l_len` members of this structure are expanded to long long data types.

As an alternative to using **fcntl**, **lockf64** can be used on large files. The **lockf64** library function has the same functionality as **lockf**, but takes a long argument for the length of the lock.

Terminology

Before discussing how to use record locking, a few terms need to be defined.

Record

A contiguous set of bytes in a file. The UNIX operating system does not impose any record structure on files. This may be done by the programs that use the files.

Cooperating Processes

Processes that work together in some well-defined fashion to accomplish the tasks at hand. Processes that share files must request permission to access the files before using them. File access permissions must be carefully set to restrict noncooperating processes from accessing those files. The term process will be used interchangeably with cooperating process to refer to a task obeying such protocols.

Read (Share) Locks

These are used to gain limited access to sections of files. When a read lock is put on a record, other processes may also read lock that record, in whole or in part. No other process, however, may have or obtain a write lock on an overlapping section of the file. If a process holds a read lock it may assume that no other process will be writing or updating that record at the same time. This access method also lets many processes read the given record. This might be necessary when searching a file, without the contention involved if a write or exclusive lock were used.

Write (Exclusive) Locks

These are used to gain complete control over sections of files. When a write lock is put on a record, no other process may read or write lock that record, in whole or in part. If a process holds a write lock it may assume that no other process will be reading or writing that record at the same time.

Advisory Locking

A form of record locking that does not interact with the I/O subsystem. Advisory locking is not enforced, for example, by **`creat`**, **`open`**, **`read`**, or **`write`**. The control over records is accomplished by requiring an appropriate record lock request before I/O operations. If appropriate requests are always made by all processes accessing the file, then the accessibility of the file will be controlled by the interaction of these requests. Advisory locking depends on the individual processes to enforce the record locking protocol; it does not require an accessibility check at the time of each I/O request.

Mandatory Locking

A form of record locking that does interact with the I/O subsystem. Access to locked records is enforced by the **`creat`**, **`open`**, **`read`** and **`write`** system calls. If a record is locked, then access of that record by any other process is restricted according to the type of lock on the record. The control over records should still be performed explicitly by requesting an appropriate record lock before I/O operations, but an additional check is made by the system before each I/O operation to ensure the record locking protocol is being honored. Mandatory locking offers an extra synchronization check, but at the cost of some additional system overhead.

File Protection

There are access permissions for UNIX system files to control who may read, write, or execute such a file. These access permissions may only be set by the owner of the file or by a process with the appropriate privilege. The permissions of the directory in which the file resides can also affect the ultimate disposition of a file. Note that if the directory permissions allow anyone to write in it, then files within the directory may be removed, even if those files do not have read, write or execute permission for that user. Any information that is worth protecting, is worth protecting properly. If your application warrants the use of record locking, make sure that the permissions on your files and directories are set properly. A record lock, even a mandatory record lock, will only protect the portions of the files that are locked. Other parts of these files might be corrupted if proper precautions are not taken.

Only a known set of programs and/or administrators should be able to read or write a data base. This can be done easily by setting the set-group-ID bit of the data base accessing programs (see **`chmod(1)`**). The files can then be accessed by a known set of programs that obey the record locking protocol. An example of such file protection, although record locking is not used, is the **`mail`** command. In that command only the particular user and the **`mail`** command can read and write in the unread mail files.

Opening a File for Record Locking

The first requirement for locking a file or segment of a file is having a valid open file descriptor. If read locks are to be done, then the file must be opened with at least read accessibility, and with write accessibility for write locks.

NOTE

Mapped files cannot be locked: if a file has been mapped, any attempt to use file or record locking on the file fails. See **`mmap(2).`**

For this example you will open your file for both read and write access:

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>

int fd;  /* file descriptor */
char *filename;

main(argc, argv)
int argc;
char *argv[];
{
      extern void exit(), perror();

      /* get data base file name from command line and open the
       * file for read and write access.
       */
      if (argc < 2) {
           (void) fprintf(stderr, “usage: %s filename\n”, argv[0]);
           exit(2);
       }
      filename = argv[1];
      fd = open(filename, O_RDWR);
      if (fd < 0) {
           perror(filename);
           exit(2);
           }
      .
      .
      .
```

The file is now open to perform both locking and I/O functions. You then proceed with the task of setting a lock.

Setting a File Lock

There are several ways to set a lock on a file. In part, these methods depend on how the lock interacts with the rest of the program. There are also questions of performance as well as portability. Two methods will be given here, one using the **fcntl** system call, the other using the **/usr/group** standards compatible **lockf** library function call.

Locking an entire file is just a special case of record locking. For both these methods the concept and the effect of the lock are the same. The file is locked starting at a byte offset of zero (0) until the end of the maximum file size. This point extends beyond any real end of the file so that no lock can be placed on this file beyond this point. To do this the value of the size of the lock is set to zero. The code using the **fcntl** system call is as follows:

```
#include <fcntl.h>
#define MAX_TRY10
int try;
struct flock lck;

try = 0;

/* set up the record locking structure, the address of which
 * is passed to the fcntl system call.
 */
lck.l_type = F_WRLCK;   /* setting a write lock */
lck.l_whence = 0;       /* offset l_start from beginning of file */
lck.l_start = 0L;
lck.l_len = 0L;         /* until the end of the file address space */

/* Attempt locking MAX_TRY times before giving up.
 */
while (fcntl(fd, F_SETLK, &lck) < 0) {
        if (errno == EAGAIN || errno == EACCES) {
                /* there might be other errors cases in which
                 * you might try again.
                 */
                if (++try < MAX_TRY) {
                        (void) sleep(2);
                        continue;
                }
                (void) fprintf(stderr,“File busy try again later!\n”);
                return;
        }
        perror(“fcntl”);
        exit(2);
}
        .
        .
        .
```

This portion of code tries to lock a file. This is attempted several times until one of the following things happens:

- the file is locked
- an error occurs
- it gives up trying because `MAX_TRY` has been exceeded

To perform the same task using the **`lockf`** function, the code is as follows:

```
#include <unistd.h>
#define MAX_TRY10
int try;
try = 0;

/* make sure the file pointer
 * is at the beginning of the file.
 */
lseek(fd, 0L, 0);

/* Attempt locking MAX_TRY times before giving up.
 */
while (lockf(fd, F_TLOCK, 0L) < 0) {
    if (errno == EAGAIN || errno == EACCES) {
        /* there might be other errors cases in which
         * you might try again.
         */
        if (++try < MAX_TRY) {
            sleep(2);
            continue;
        }
        (void) fprintf(stderr,“File busy try again later!\n”);
        return;
    }
    perror(“lockf”);
    exit(2);
}
    .
    .
    .
```

It should be noted that the **`lockf`** example appears to be simpler, but the **`fcntl`** example exhibits additional flexibility. Using the **`fcntl`** method, it is possible to set the type and start of the lock request simply by setting a few structure variables. **`lockf`** merely sets write (exclusive) locks; an additional system call, **`lseek`**, is required to specify the start of the lock.

Setting and Removing Record Locks

Locking a record is done the same way as locking a file except for the differing starting point and length of the lock. You will now try to solve an interesting and real problem. There are two records (these records may be in the same or different file) that must be updated simultaneously so that other processes get a consistent view of this information. (This type of problem comes up, for example, when updating the interrecord pointers in a doubly linked list.) To do this you must decide the following questions:

- What do you want to lock?
- For multiple locks, in what order do you want to lock and unlock the records?
- What do you do if you succeed in getting all the required locks?
- What do you do if you fail to get all the locks?

In managing record locks, you must plan a failure strategy if you cannot obtain all the required locks. It is because of contention for these records that you have decided to use record locking in the first place. Different programs might:

- wait a certain amount of time, and try again
- abort the procedure and warn the user
- let the process sleep until signaled that the lock has been freed
- some combination of the above

Now look at the example of inserting an entry into a doubly linked list. For the example, you will assume that the record after which the new record is to be inserted has a read lock on it already. The lock on this record must be changed or promoted to a write lock so that the record may be edited.

Promoting a lock (generally from read lock to write lock) is permitted if no other process is holding a read lock in the same section of the file. If there are processes with pending write locks that are sleeping on the same section of the file, the lock promotion succeeds and the other (sleeping) locks wait. Promoting (or demoting) a write lock to a read lock carries no restrictions. In either case, the lock is merely reset with the new lock type. Because the **`/usr/group lockf`** function does not have read locks, lock promotion is not applicable to that call. An example of record locking with lock promotion follows:

```
struct record {
    .
    .   /* data portion of record */
    .
    long prev;/* index to previous record in the list */
    long next;/* index to next record in the list */
};

/* Lock promotion using fcntl(2)
 * When this routine is entered it is assumed that there are read
 * locks on “here” and “next”.
 * If write locks on “here” and “next” are obtained:
 *    Set a write lock on “this”.
 *    Return index to “this” record.
 * If any write lock is not obtained:
 *    Restore read locks on “here” and “next”.
 *    Remove all other locks.
 *    Return a -1.
 */
long
set3lock (this, here, next)
long this, here, next;
{
    struct flock lck;

    lck.l_type = F_WRLCK;   /* setting a write lock */
    lck.l_whence = 0;       /* offset l_start from beginning of file */
    lck.l_start = here;
    lck.l_len = sizeof(struct record);

    /* promote lock on “here” to write lock */
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
          return (-1);
    }
    /* lock “this” with write lock */
    lck.l_start = this;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
          /* Lock on “this” failed;
           * demote lock on “here” to read lock.
           */
          lck.l_type = F_RDLCK;
          lck.l_start = here;
          (void) fcntl(fd, F_SETLKW, &lck);
          return (-1);
    }
    /* promote lock on “next” to write lock */
    lck.l_start = next;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
          /* Lock on “next” failed;
           * demote lock on “here” to read lock,
           */
           lck.l_type = F_RDLCK;
        lck.l_start = here;
           (void) fcntl(fd, F_SETLK, &lck);
           /* and remove lock on “this”.
            */
           lck.l_type = F_UNLCK;
           lck.l_start = this;
           (void) fcntl(fd, F_SETLK, &lck);
           return (-1);/* cannot set lock, try again or quit */
    }

    return (this);
}
```

The locks on these three records were all set to wait (sleep) if another process was blocking them from being set. This was done with the `F_SETLKW` command. If the `F_SETLK` command was used instead, the **`fcntl`** system calls would fail if blocked. The program would then have to be changed to handle the blocked condition in each of the error return sections.

Now look at a similar example using the **lockf** function. Since there are no read locks, all (write) locks will be referenced generically as locks.

```
/* Lock promotion using lockf(3)
 * When this routine is entered it is assumed that there are
 * no locks on “here” and “next”.
 * If locks are obtained:
 *    Set a lock on “this”.
 *    Return index to “this” record.
 * If any lock is not obtained:
 *    Remove all other locks.
 *    Return a -1.
 */

#include <unistd.h>

long
set3lock (this, here, next)
long this, here, next;

{

    /* lock “here” */
    (void) lseek(fd, here, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        return (-1);
    }
    /* lock “this” */
    (void) lseek(fd, this, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        /* Lock on “this” failed.
         * Clear lock on “here”.
         */
        (void) lseek(fd, here, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1);

    }

    /* lock “next” */
    (void) lseek(fd, next, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {

        /* Lock on “next” failed.
         * Clear lock on “here”,
         */
        (void) lseek(fd, here, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));

        /* and remove lock on “this”.
         */
        (void) lseek(fd, this, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1);/* cannot set lock, try again or quit */

    }


    return (this);
}
```

Locks are removed in the same manner as they are set, only the lock type is different (`F_UNLCK` or `F_ULOCK`). An unlock cannot be blocked by another process and will only affect locks that were placed by this process. The unlock only affects the section of the file defined in the previous example by `lck`. It is possible to unlock or change the type of lock on a subsection of a previously set lock. This may cause an additional lock (two locks for one system call) to be used by the operating system. This occurs if the subsection is from the middle of the previously set lock.

Getting Lock Information

You can determine which processes, if any, are blocking a lock from being set. This can be used as a simple test or as a means to find locks on a file. A lock is set up as in the previous examples and the `F_GETLK` command is used in the **`fcntl`** call. If the lock passed to **`fcntl`** would be blocked, the first blocking lock is returned to the process through the structure passed to **`fcntl`**. That is, the lock data passed to **`fcntl`** is overwritten by blocking lock information. This information includes two pieces of data that have not been discussed yet, `l_pid` and `l_sysid`, that are only used by `F_GETLK`. (For systems that do not support a distributed architecture the value in `l_sysid` should be ignored.) These fields uniquely identify the process holding the lock.

If a lock passed to **`fcntl`** using the `F_GETLK` command would not be blocked by another process's lock, then the `l_type` field is changed to `F_UNLCK` and the remaining fields in the structure are unaffected. You can use this capability to print all the segments locked by other processes. Note that if there are several read locks over the same segment only one of these will be found.

```
struct flock lck;

/* Find and print “write lock” blocked segments of this file. */
     (void) printf(“sysid   pid type    start   length\n”);
     lck.l_whence = 0;
     lck.l_start = 0L;
     lck.l_len = 0L;
     do {
          lck.l_type = F_WRLCK;
          (void) fcntl(fd, F_GETLK, &lck);
          if (lck.l_type != F_UNLCK) {
               (void) printf(“%5d %5d   %c  %8d %8d\n”,
                    lck.l_sysid,
                    lck.l_pid,
                    (lck.l_type == F_WRLCK) ? 'W' : 'R',
                    lck.l_start,
                    lck.l_len);
               /* if this lock goes to the end of the address
                * space, no need to look further, so break out.
                */
               if (lck.l_len == 0)
                    break;
               /* otherwise, look for new lock after the one
                * just found.
                */
               lck.l_start += lck.l_len;
          }
     } while (lck.l_type != F_UNLCK);
```

`fcntl` with the `F_GETLK` command will always return correctly (that is, it will not sleep or fail) if the values passed to it as arguments are valid.

The **`lockf`** function with the `F_TEST` command can also be used to test if there is a process blocking a lock. This function does not, however, return the information about where the lock actually is and which process owns the lock. A routine using **`lockf`** to test for a lock on a file follows:

```
/* find a blocked record. */
/* seek to beginning of file */
(void) lseek(fd, 0, 0L);
/* set the size of the test region to zero (0)
 * to test until the end of the file address space.
 */
if (lockf(fd, F_TEST, 0L) < 0) {
    switch (errno) {
        case EACCES:
        case EAGAIN:
        (void) printf(“file is locked by another process\n”);
        break;
        case EBADF:
        /* bad argument passed to lockf */
        perror(“lockf”);
        break;
        default:
        (void) printf(“lockf: unknown error <%d>\n”, errno);
        break;
        }
    }
```

When a process forks, the child receives a copy of the file descriptors that the parent has opened. The parent and child also share a common file pointer for each file. If the parent were to seek to a point in the file, the child's file pointer would also be at that location. This feature has important implications when using record locking. The current value of the file pointer is used as the reference for the offset of the beginning of the lock, as described by `l_start`, when using a `l_whence` value of `1`. If both the parent and child process set locks on the same file, there is a possibility that a lock will be set using a file pointer that was reset by the other process. This problem appears in the **`lockf`** function call as well and is a result of the **`/usr/group`** requirements for record locking. If forking is used in a record locking program, the child process should close and reopen the file if either locking method is used. This will result in the creation of a new and separate file pointer that can be manipulated without this problem occurring. Another solution is to use the **`fcntl`** system call with a `l_whence` value of `0` or `2`. This makes the locking function atomic, so that even processes sharing file pointers can be locked without difficulty.

Deadlock Handling

There is a certain level of deadlock detection/avoidance built into the record locking facility. This deadlock handling provides the same level of protection granted by the **`/usr/group`** standard **`lockf`** call. This deadlock detection is only valid for processes that are locking files or records on a single system. Deadlocks can only potentially occur when the system is about to put a record locking system call to sleep. A search is made for constraint loops of processes that would cause the system call to sleep indefinitely. If such a situation is found, the locking system call will fail and set `errno` to the deadlock error number. If a process wishes to avoid the use of the systems deadlock detection it should set its locks using `F_GETLK` instead of `F_GETLKW.`

Selecting Advisory or Mandatory Locking

The use of mandatory locking is not recommended for reasons that will be made clear in a subsequent section. Whether or not locks are enforced by the I/O system calls is determined at the time the calls are made by the permissions on the file (see **chmod(2)**). For locks to be under mandatory enforcement, the file must be a regular file with the set-group-ID bit on and the group execute permission off. If either condition fails, all record locks are advisory. Mandatory enforcement can be assured by the following code:

```
#include <sys/types.h>
#include <sys/stat.h>

int mode;
struct stat buf;
        .
        .
        .
    if (stat(filename, &buf) < 0) {
        perror(“program”);
        exit (2);
    }
    /* get currently set mode */
    mode = buf.st_mode;
    /* remove group execute permission from mode */
    mode &= ~(S_IEXEC>>3);
    /* set 'set group id bit' in mode */
    mode |= S_ISGID;
    if (chmod(filename, mode) < 0) {
        perror(“program”);
        exit(2);
    }
        .
        .
        .
```

Files that are to be record locked should never have any type of execute permission set on them. This is because the operating system does not obey the record locking protocol when executing a file.

The **chmod(1)** command can also be easily used to set a file to have mandatory locking. This can be done with the command:

chmod +l *file*

The **ls(1)** command shows this setting when you ask for the long listing format:

ls -l *file*

causes the following to be printed:

```
-rw---l---   1 user     group    size    mod_time    file
```

Caveat Emptor—Mandatory Locking

- Mandatory locking only protects those portions of a file that are locked. Other portions of the file that are not locked may be accessed according to normal UNIX system file permissions.
- If multiple reads or writes are necessary for an atomic transaction, the process should explicitly lock all such pieces before any I/O begins. Thus advisory enforcement is sufficient for all programs that perform in this way.

- As stated earlier, arbitrary programs should not have unrestricted access permission to files that are important enough to record lock.
- Advisory locking is more efficient because a record lock check does not have to be performed for every I/O request.

Record Locking and Future Releases of the UNIX System

Provisions have been made for file and record locking in a UNIX system environment. In such an environment the system on which the locking process resides may be remote from the system on which the file and record locks reside. In this way multiple processes on different systems may put locks upon a single file that resides on one of these or yet another system. The record locks for a file reside on the system that maintains the file. It is also important to note that deadlock detection/avoidance is only determined by the record locks being held by and for a single system. Therefore, it is necessary that a process only hold record locks on a single system at any given time for the deadlock mechanism to be effective. If a process needs to maintain locks over several systems, it is suggested that the process avoid the sleep-when-blocked features of **fcntl** or **lockf** and that the process maintain its own deadlock detection. If the process uses the sleep-when-blocked feature, then a timeout mechanism should be provided by the process so that it does not hang waiting for a lock to be cleared.

Basic STREAMS Operations

This section describes the basic set of operations for manipulating STREAMS entities.

A STREAMS driver is similar to a traditional character I/O driver in that it has one or more nodes associated with it in the file system, and it is accessed using the **open** system call. Typically, each file system node corresponds to a separate minor device for that driver. Opening different minor devices of a driver causes separate Streams to be connected between a user process and the driver. The file descriptor returned by the **open** call is used for further access to the Stream. If the same minor device is opened more than once, only one Stream is created; the first **open** call creates the Stream, and subsequent **open** calls return a file descriptor that references that Stream. Each process that opens the same minor device shares the same Stream to the device driver.

Once a device is opened, a user process can send data to the device using the **write** system call and receive data from the device using the **read** system call. Access to STREAMS drivers using **read** and **write** is compatible with the traditional character I/O mechanism.

The **close** system call closes a device and dismantles the associated Stream when the last open reference to the Stream is given up.

The following example shows how a simple Stream is used. In the example, the user program interacts with a communications device that provides point-to-point data transfer between two computers. Data written to the device transmitted over the communications line, and data arriving on the line can be retrieved by reading from the device.

```
#include <fcntl.h>

main()
{
    char buf[1024];
    int fd, count;

    if ((fd = open("/dev/comm/01", O_RDWR)) < 0) {
        perror("open failed");
        exit(1);
    }

    while ((count = read(fd, buf, 1024)) > 0) {
        if (write(fd, buf, count) != count) {
            perror("write failed");
            break;
        }
    }
    exit(0);
}
```

In the example, **`/dev/comm/01`** identifies a minor device of the communications device driver. When this file is opened, the system recognizes the device as a STREAMS device and connects a Stream to the driver. Figure 3-2 shows the state of the Stream following the call to **open**.

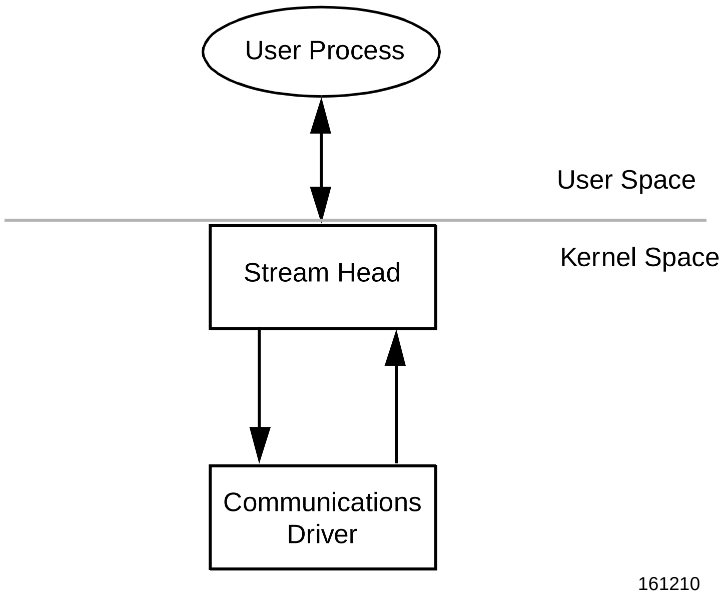


Figure 3-2. Stream to Communication Driver

This example illustrates a user reading data from the communications device and then writing the input back out to the same device. In short, this program echoes all input back over the communications line. The example assumes that a user sends data from the other side of the communications line. The program reads up to 1024 bytes at a time, and then writes the number of bytes just read.

The **`read`** call returns the available data, which may contain fewer than 1024 bytes. If no data is currently available at the Stream head, the **`read`** call blocks until data arrive.

Similarly, the **`write`** call attempts to send count bytes to **`/dev/comm/01`**. However, STREAMS implements a flow control mechanism that prevents a user from exhausting system resources by flooding a device driver with data.

Flow control controls the rate of message transfer among the modules, drivers, Stream head, and processes. Flow control is local to each Stream and advisory (voluntary). It limits the number of characters that can be queued for processing at any queue in a Stream, and limits buffers and related processing at any queue and in any one Stream, but does not consider buffer pool levels or buffer usage in other Streams. Flow control is not applied to high-priority messages.

If the Stream exerts flow control on the user, the **`write`** call blocks until flow control is relieved. The call does not return until it has sent count bytes to the device. **`exit`**, which is called to terminate the user process, also closes all open files, and thereby dismantling the Stream in this example.

Benefits of STREAMS

STREAMS provides the following benefits:

- A flexible, portable, and reusable set of tools for development of UNIX system communication services.
- Easy creation of modules that offer standard data communications services and the ability to manipulate those modules on a Stream.
- From user level, modules can be dynamically selected and interconnected; kernel programming, assembly, and link editing are not required to create the interconnection.

STREAMS also greatly simplifies the user interface for languages that have complex input and output requirements.

Standardized Service Interfaces

STREAMS simplifies the creation of modules that present a service interface to any neighboring application program, module, or device driver. A service interface is defined at the boundary between two neighbors. In STREAMS, a service interface is a specified set of messages and the rules that allow passage of these messages across the boundary. A module that implements a service interface receives a message from a neighbor and responds with an appropriate action (for example, sends back a request to retransmit) based on the specific message received and the preceding sequence of messages.

In general, any two modules can be connected anywhere in a Stream. However, rational sequences are generally constructed by connecting modules with compatible protocol service interfaces. For example, a module that implements an X.25 protocol layer, as shown in Figure 3-2, presents a protocol service interface at its input and output sides. In this case, other modules should only be connected to the input and output side if they have the compatible X.25 service interface.

Manipulating Modules

STREAMS provides the capabilities to manipulate modules from the user level, to interchange modules with common service interfaces, and to change the service interface to a STREAMS user process. These capabilities yield further benefits when implementing networking services and protocols, including:

- User level programs can be independent of underlying protocols and physical communication media.
- Network architectures and higher level protocols can be independent of underlying protocols, drivers, and physical communication media.
- Higher level services can be created by selecting and connecting lower level services and protocols.

The following examples show the benefits of STREAMS capabilities for creating service interfaces and manipulating modules. These examples are only illustrations and do not necessarily reflect real situations.

Protocol Portability

Figure 3-3 shows how the same X.25 protocol module can be used with different drivers on different machines by implementing compatible service interfaces. The X.25 protocol module interfaces are Connection Oriented Network Service (CONS) and Link Access Protocol - Balanced (LAPB).

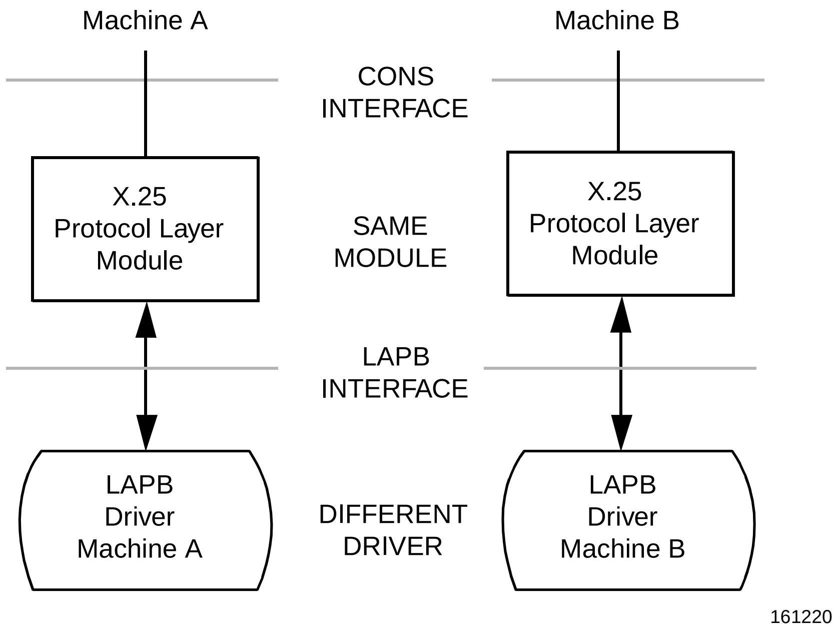


Figure 3-3. X.25 Multiplexing Stream

Protocol Substitution

Alternate protocol modules (and device drivers) can be interchanged on the same machine if they are implemented to an equivalent service interface.

Protocol Migration

Figure 3-4 illustrates how STREAMS can move functions between kernel software and front-end firmware. A common downstream service interface allows the transport protocol module to be independent of the number or type of modules below. The same transport module connects without change to either an X.25 module or X.25 driver that has the same service interface.

By shifting functions between software and firmware, developers can produce cost effective, functionally equivalent systems over a wide range of configurations. They can rapidly incorporate technological advances. The same transport protocol module can be used on a lower capacity machine, where economics may preclude the use of front-end hardware, and also on a larger scale system where a front-end is economically justified.

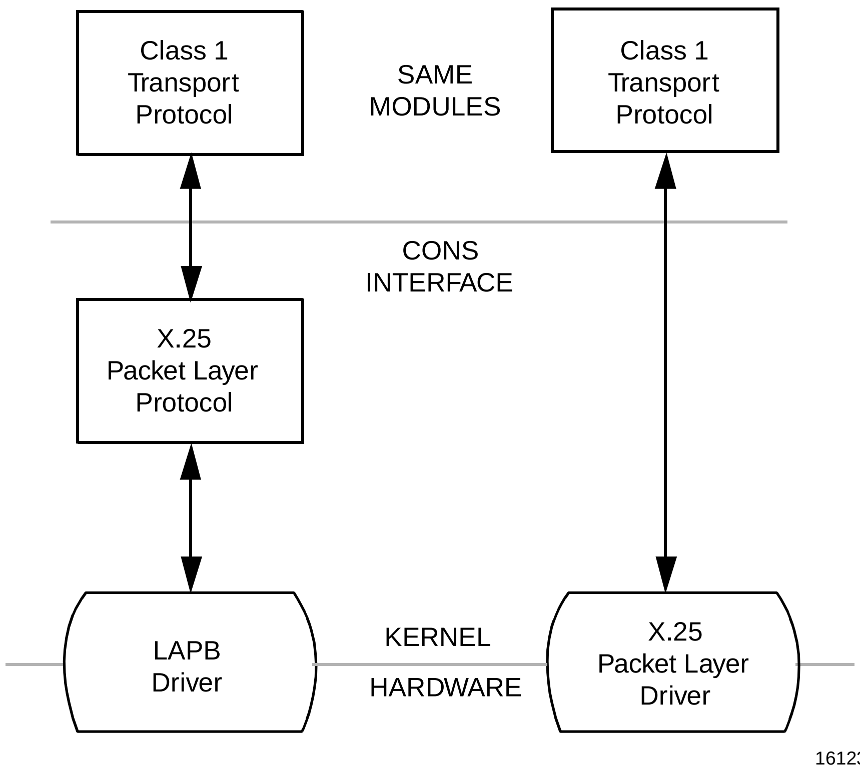


Figure 3-4. Protocol Migration

Module Reusability

Figure 3-5 shows the same canonical module (for example, one that provides delete and kill processing on character strings) reused in two different Streams. This module is typically implemented as a filter, with no downstream service interface. In both cases, a tty interface is presented to the Stream's user process because the module is nearest to the Stream head.

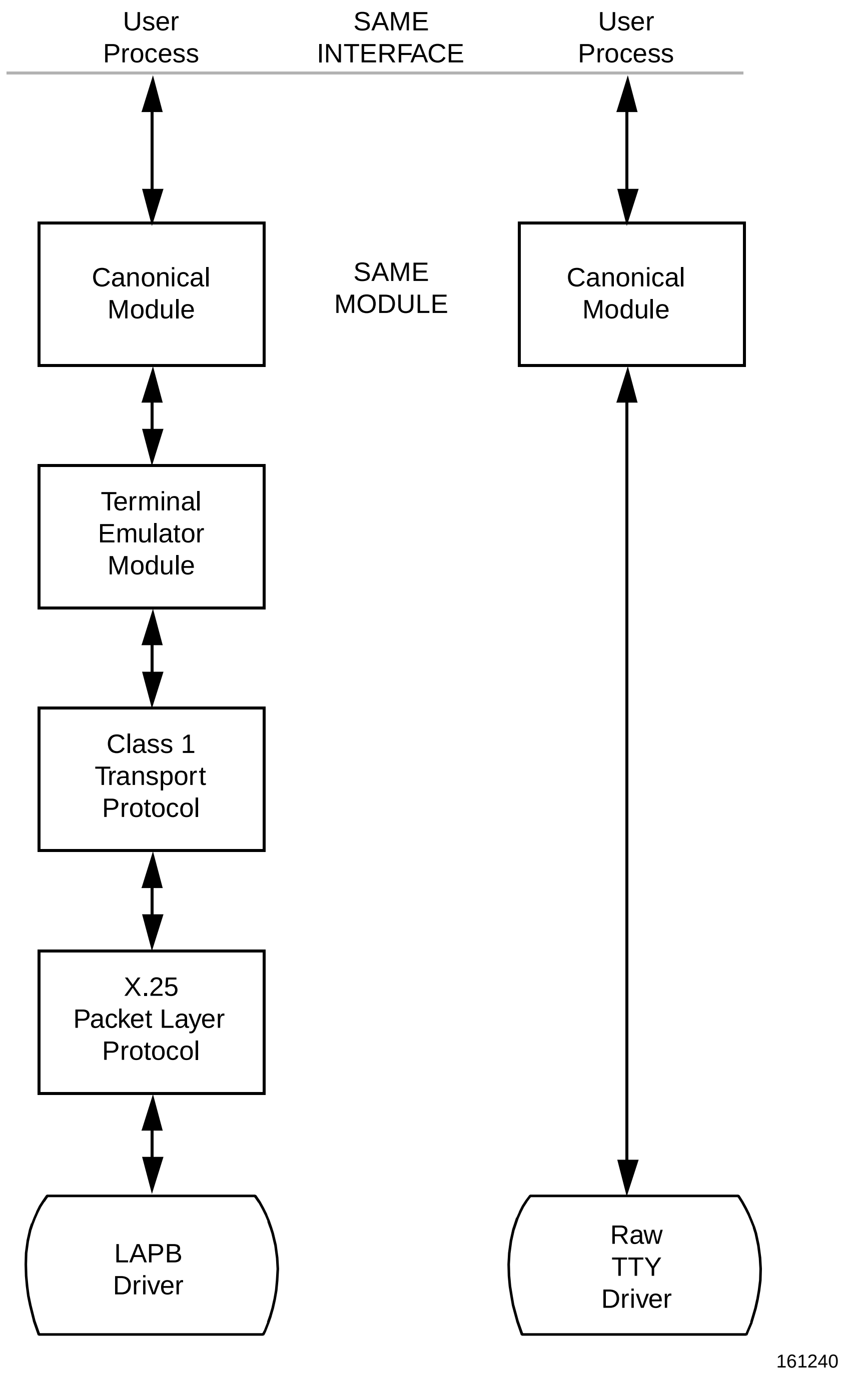


Figure 3-5. Module Reusability

STREAMS Mechanism

This chapter shows how to construct, use, and dismantle a Stream using STREAMS-related systems calls. General and STREAMS-specific system calls provide the user level facilities required to implement application programs. This system call interface is upwardly compatible with the traditional character I/O facilities. The **`open`** system call recognizes a STREAMS file and creates a Stream to the specified driver. A user process can receive and send data on STREAMS files using **`read`** and **`write`** in the same way as with traditional character files. The **`ioctl`** system call enables users to perform functions specific to a particular device. STREAMS **`ioctl`** commands (see **`streamio(7)`**)support a variety of functions for accessing and controlling Streams. The last **`close`** in a Stream dismantles a Stream.

In addition to the traditional **`ioctl`** commands and system calls, there are other system calls used by STREAMS. The **`poll`** system call enables a user to poll multiple Streams for various events. The **`putmsg`** and **`getmsg`** system calls enable users to send and receive STREAMS messages, and are suitable for interacting with STREAMS modules and drivers through a service interface.

STREAMS provides kernel facilities and utilities to support development of modules and drivers. The Stream head handles most system calls so that the related processing does not have to be incorporated in a module or driver.

STREAMS System Calls

The STREAMS-related system calls are as follows:

`open`	Open a Stream
`close`	Close a Stream
`read`	Read data from a Stream
`write`	Write data to a Stream
`ioctl`	Control a Stream
`getmsg`	Receive a message at the Stream head
`putmsg`	Send a message downstream
`poll`	Notify the application program when selected events occur on a Stream
`pipe`	Create a channel that provides a communication path between multiple processes

A STREAMS device responds to the standard character I/O system calls, such as **`read`** and **`write`**, by turning the request into a message. This feature ensures that STREAMS devices may be accessed from the user level in the same manner as non-STREAMS character devices. However, additional system calls provide other capabilities.

getmsg and putmsg

The **`putmsg`** and **`getmsg`** system calls enable a user process to send and receive STREAMS messages, in the same form the messages have in kernel modules and drivers. **`read`** and **`write`** are not designed to include the message boundaries necessary to encode messages.

The advantage of this capability is that a user process, as well as a STREAMS module or driver, can implement a service interface.

poll

The **`poll`** system call allows a user process to monitor a number of streams to detect expected I/O events. Such events might be the availability of a device for writing, input data arriving from a device, a hangup occurring, an error being detected, or the arrival of a priority message. See **`poll(2)`** in the *Operating System API Reference* for more information.

Opening a STREAMS Device File

One way to construct a Stream is to open (see **`open(2)`**) a STREAMS-based driver file.

If the **`open`** call is the initial file open, a Stream is created. (There is one Stream per major/minor device pair.)

If this is the initial open of this Stream, the driver open routine is called. If modules have been specified to be autopushed, they are pushed immediately after the driver open. When a Stream is already open, further opens of the same Stream result in calls to the open procedures of all pushable modules and the driver open. Note that this is done in the reverse order from the initial Stream open. In other words, the initial open processes from the Stream end to the Stream head, while later opens process from the Stream head to the Stream end.

Creating a STREAMS-based Pipe

In addition to opening a STREAMS-based driver, a Stream can be created by creating a pipe (see **`pipe(2)`**). Because pipes are not character devices, STREAMS creates and initializes a `streamtab` structure for each end of the pipe.

When the **`pipe`** system call is executed, two Streams are created. STREAMS follows the procedures similar to those of opening a driver; however, duplicate data structures are created. That is, two entries are allocated in the user's file table and two `vnode`s are created to represent each end of the pipe. The file table entries are initialized to point to the allocated `vnode`s and each `vnode` is initialized to specify a file of type `FIFO`.

Each Stream header represents one end of the pipe, and it points to the downstream half of each Stream head queue pair. Unlike STREAMS-based devices, however, the downstream portion of the Stream terminates at the upstream portion of the other Stream.

Adding and Removing Modules

As part of constructing a Stream, a module can be added (pushed) with an **ioctl I_PUSH** (see **streamio(7)**) system call. The push inserts a module beneath the Stream head. Because of the similarity of STREAMS components, the push operation is similar to the driver open. First, the address of the qinit structure for the module is obtained.

Next, STREAMS allocates a pair of queue structures and initializes their contents as in the driver open.

Then, q_next values are set and modified so that the module is interposed between the Stream head and its neighbor immediately downstream. Finally, the module open procedure (located using **qinit**) is called.

Each push of a module is independent, even in the same Stream. If the same module is pushed more than once on a Stream, there will be multiple occurrences of that module in the Stream. The total number of pushable modules that may be contained on any one Stream is limited by the kernel parameter NSTRPUSH.

An **ioctl I_POP** (see **streamio(7)**) system call removes (pops) the module immediately below the Stream head. The pop calls the module close procedure. On return from the module close, any messages left on the module's message queues are freed (deallocated). Then, STREAMS connects the Stream head to the component previously below the popped module and deallocates the module's queue pair. **I_PUSH** and **I_POP** enable a user process to alter dynamically the configuration of a Stream by pushing and popping modules as required. For example, a module may be removed and a new one inserted below the Stream head. Then the original module can be pushed back after the new module has been pushed.

Closing the Stream

The last **close** to a STREAMS file dismantles the Stream. Dismantling consists of popping any modules on the Stream and closing the driver. Before a module is popped, the **close** may delay to allow any messages on the write message queue of the module to be drained by module processing. Similarly, before the driver is closed, the **close** may delay to allow any messages on the write message queue of the driver to be drained by driver processing. If O_NDELAY (or O_NONBLOCK) is clear, **close** waits up to 15 seconds for each module to drain and up to 15 seconds for the driver to drain (see **open(2)**). If O_NDELAY (or O_NONBLOCK) is set, the pop is performed immediately and the driver is closed without delay. Messages can remain queued, for example, if flow control is inhibiting execution of the write queue service procedure. When all modules are popped and any wait for the driver to drain is completed, the driver close routine is called. On return from the driver close, any messages left on the driver's queues are freed, and the queue and stdata structures are deallocated.

NOTE

STREAMS frees only the messages contained on a message queue. Any message or data structures used internally by the driver or module must be freed by the driver or module close procedure.

Finally, the user's file table entry and the vnode are deallocated and the file is closed.

Stream Construction Example

The following example extends the previous communications device echoing example (see the section “Basic STREAMS Operations” in this chapter) by inserting a module in the Stream. The (hypothetical) module in this example can convert (change case, delete, and/or duplicate) selected alphabetic characters.

Inserting Modules

An advantage of STREAMS over the traditional character I/O mechanism stems from the ability to insert various modules into a Stream to process and manipulate data that pass between a user process and the driver. In the example, the character conversion module is passed a command and a corresponding string of characters by the user. All data passing through the module are inspected for instances of characters in this string; the operation identified by the command is performed on all matching characters. The necessary declarations for this program are shown below:

```
#include <string.h>
#include <fcntl.h>
#include <stropts.h>

#define BUFLEN 1024

/*
 * These defines would typically be
 * found in a header file for the module
 */
#define XCASE        1   /* change alphabetic case of char */
#define DELETE       2   /* delete char */
#define DUPLICATE    3   /* duplicate char */

main()
{
    char buf[BUFLEN];
    int fd, count;
    struct strioctl strioctl;
```

The first step is to establish a Stream to the communications driver and insert the character conversion module. The following sequence of system calls accomplishes the following display:

```
if ((fd = open("/dev/comm/01", O_RDWR)) < 0) {
    perror("open failed");
    exit(1);
}

if (ioctl(fd, I_PUSH, "chconv") < 0) {
    perror("ioctl I_PUSH failed");
    exit(2);
}
```

The **`I_PUSH ioctl`** call directs the Stream head to insert the character conversion module between the driver and the Stream head, creating the Stream shown in Figure 3-6 As with drivers, this module resides in the kernel and must have been configured into the system before it was booted, unless the system has an autoload capability.

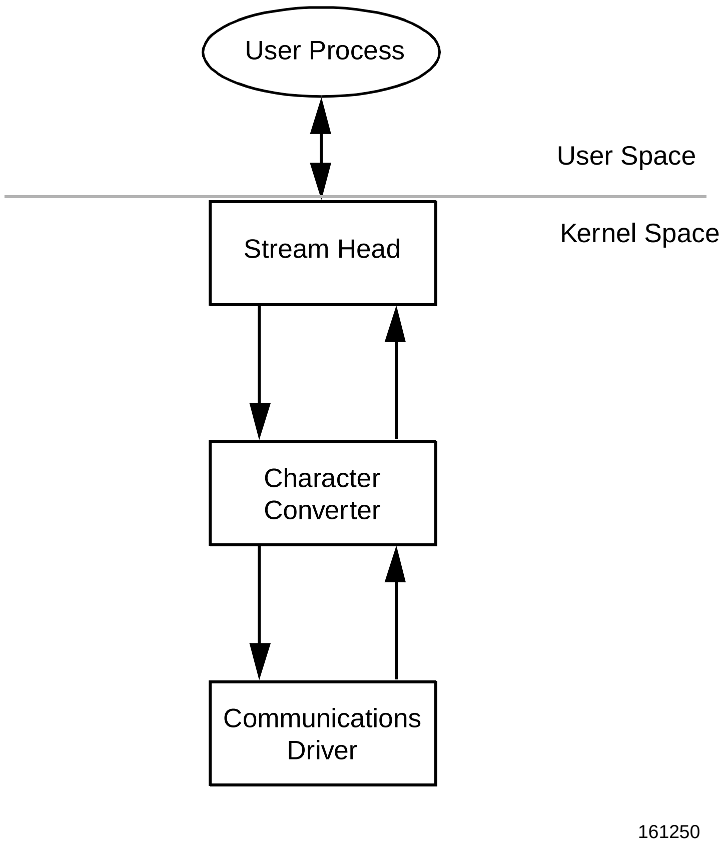


Figure 3-6. Case Converter Module

An important difference between STREAMS drivers and modules is illustrated here. Drivers are accessed through a node or nodes in the file system and may be opened just like any other device. Modules, on the other hand, do not occupy a file system node.

Instead, they are identified through a separate naming convention, and are inserted into a Stream using **`I_PUSH`**. The name of a module is defined by the module developer

Modules are pushed onto a Stream and removed from a Stream in Last-In-First-Out (LIFO) order. Therefore, if a second module was pushed onto this Stream, it would be inserted between the Stream head and the character conversion module

Module and Driver Control

The next step in this example is to pass the commands and corresponding strings to the character conversion module. This can be done by issuing **`ioctl`** calls to the character conversion module as follows:

```
    /* change all uppercase vowels to lowercase */
    strioctl.ic_cmd = XCASE;
    strioctl.ic_timout = 0;/* default timeout (15 sec) */
    strioctl.ic_dp = “AEIOU”;
    strioctl.ic_len = strlen(strioctl.ic_dp);

    if (ioctl(fd, I_STR, &strioctl) < 0) {
        perror(“ioctl I_STR failed”);
        exit(3);
    }

    /* delete all instances of the chars 'x' and 'X' */
    strioctl.ic_cmd = DELETE;
    strioctl.ic_dp = “xX”;
    strioctl.ic_len = strlen(strioctl.ic_dp);

    if (ioctl(fd, I_STR, &strioctl) < 0) {
        perror(“ioctl I_STR failed”);
        exit(4);
    }
```

`ioctl` requests are issued to STREAMS drivers and modules indirectly, using the **`I_STR`** **`ioctl`** call (see **`streamio(7)`**). The argument to **`I_STR`** must be a pointer to a `strioctl` structure, which specifies the request to be made to a module or driver. This structure is defined in **`<stropts.h>`** and has the following format:

```
struct strioctl {
    int  ic_cmd;      /* ioctl request */
    int  ic_timout;   /* ACK/NAK timeout */
    int  ic_len;      /* length of data argument */
    char *ic_dp;      /* ptr to data argument */
};
```

where `ic_cmd` identifies the command intended for a module or driver, `ic_timout` specifies the number of seconds an **`I_STR`** request should wait for an acknowledgment before timing out, `ic_len` is the number of bytes of data to accompany the request, and `ic_dp` points to that data.

In the example, two separate commands are sent to the character conversion module. The first sets `ic_cmd` to the command `XCASE` and sends as data the string “AEIOU”; it converts all uppercase vowels in data passing through the module to lowercase. The second sets `ic_cmd` to the command `DELETE` and sends as data the string “xX”; it deletes all occurrences of the characters ‘x’ and ‘X’ from data passing through the module. For each command, the value of `ic_timout` is set to zero, which specifies the system default timeout value of 15 seconds. The `ic_dp` field points to the beginning of the data for each command; `ic_len` is set to the length of the data.

`I_STR` is intercepted by the Stream head, which packages it into a message, using information contained in the `strioctl` structure, and sends the message downstream. Any module that does not understand the command in `ic_cmd` passes the message further downstream. The request will be processed by the module or driver closest to the Stream head that understands the command specified by `ic_cmd`. The **`ioctl`** call will block up to `ic_timout` seconds, waiting for the target module or driver to respond with either a positive or negative acknowledgment message. If an acknowledgment is not received in `ic_timout` seconds, the **`ioctl`** call will fail.

NOTE

Only one **`I_STR`** request can be active on a Stream at one time. Further requests will block until the active **`I_STR`** request is acknowledged and the system call completes.

The `strioctl` structure is also used to retrieve the results, if any, of an **`I_STR`** request. If data is returned by the target module or driver, `ic_dp` must point to a buffer large enough to hold that data, and `ic_len` will be set on return to show the amount of data returned:

```
    while ((count = read(fd, buf, BUFLEN)) > 0) {
        if (write(fd, buf, count) != count) {
            perror(“write failed”);
            break;
        }
    }
    exit(0);
}
```

Note that the character conversion processing was realized with no change to the communications driver.

The **`exit`** system call dismantles the Stream before terminating the process. The character conversion module is removed from the Stream automatically when it is closed. Alternatively, modules may be removed from a Stream using the **`I_POP`** **`ioctl`** call described in **`streamio(7)`**. This call removes the topmost module on the Stream, and enables a user process to alter the configuration of a Stream dynamically, by popping modules as needed.

A few of the important **`ioctl`** requests supported by STREAMS have been discussed. Several other requests are available to support operations such as determining if a given module exists on the Stream, or flushing the data on a Stream. These requests are described fully in **`streamio(7)`**.

4
Process Management

4
Process Management

Introduction

A process is the execution of a program; most OS commands execute as separate processes. Each process is a distinct entity, able to execute and terminate independently of all other processes. Each user can have many processes in the system simultaneously. In fact, it is not always necessary for the user to be logged into the system while those processes are executing.

The OS supports a schedulable entity called a *lightweight process* (LWP). Each process contains one or more LWPs. LWPs allow multiple threads of control within a single process. The Threads Library provides interfaces with which applications may be multithreaded. See Chapter 11, entitled, “Programming with the Threads Library” for information about threads and LWPs. When a process does not explicitly create any new LWPs, it contains one LWP and has the same semantics that a process had in previous releases.

Whenever you execute a command in a UNIX system, you are initiating a process that is numbered and tracked by the operating system. A flexible feature of a UNIX system is that processes can be generated by other processes. This happens more than you might ever be aware of. For example, when you log in to your system you are running a process, very probably the shell. If you then use an editor such as **`vi`**, take the option of invoking the shell from **`vi`**, and execute the **`ps`** command, you will see a display something like the one in the following figure (which shows the results of a **`ps -f`** command):

```
UID      PID      PPID     CLS  PRI  STIME        TTY      TIME     COMD
abc      24210    1        TS   70   06:13:14     tty29    0:05     -sh
abc      24631    24210    TS   70   06:59:07     tty29    0:13     vi c2.uli
abc      28441    28358    TS   70   09:17:22     tty29    0:01     ps -f
abc      28358    24631    TS   70   09:15:14     tty29    0:01     sh -i
```

Figure 4-1. Process Status

As you can see, user `abc` (who went through the steps described above) now has four processes active. It is an interesting exercise to trace the chain that is shown in the Process ID (PID) and Parent Process ID (PPID) columns. The shell that was started when user abc logged on is process 24210; its parent is the initialization process (process ID 1). Process 24210 is the parent of process 24631, and so on.

The four processes in the example above are all UNIX system shell-level commands, but you can spawn new processes from your own program. You might think, “Well, it's one thing to switch from one program to another when I'm at my terminal working interactively with the computer; but why would a program want to run other programs, and if one does, why wouldn't I just put everything together into one big executable module?"

Overlooking the case where your program is itself an interactive application with diverse choices for the user, your program may need to run one or more other programs based on conditions it encounters in its own processing. (If it's the end of the month, go do a trial balance, for example.) The usual reasons why it might not be practical to create one large executable are:

- The load module may get too big to fit in the maximum process size for your system.
- You may not have control over the object code of all the other modules you want to include.

Suffice it to say, there are legitimate reasons why this creation of new processes might need to be done. There are two ways to do it:

- **`exec(2)`**—stop this process and start another
- **`fork(2)`**—start an additional copy of this process

Program Execution and Process Creation

Program Execution—exec

Overlays, performed by the family of **`exec`** system calls, can change the executing program, but cannot create new processes. Processes are created (or spawned) by the system call **`fork`**, which is discussed later.

`exec` is the name of a family of functions that includes **`execl`**, **`execv`**, **`execle`**, **`execve`**, **`execlp`**, and **`execvp`**. They all have the function of transforming the calling process into a new process. The reason for the variety is to provide different ways of pulling together and presenting the arguments of the function. An example of one version (**`execl`**) might be:

```
execl("/usr/bin/prog2", "prog", progarg1,
        progarg2, (char *)0);
```

For **`execl`** the argument list is

`/usr/bin/prog2`	path name of the new process file
`prog`	the name the new process gets in its `argv[0]`
`progarg1`, `progarg2`	arguments to `prog2` as `char *`'s
`(char *)0`	a null `char` pointer to mark the end of the arguments

Check the **`exec(2)`** manual page in the *Operating System API Reference* for the rest of the details. The key point of the **`exec`** family is that there is no return from a successful execution: the new process overlays the process that makes the **exec** system call. The new process also takes over the process ID and other attributes of the old process. If the call to **exec** is unsuccessful, control is returned to your program with a return value of `-1`. You can check `errno` to learn why it failed.

The system call **execl** executes another program, *without returning*; thus, to print the date as the last action of a running program, use:

```
    execl("/bin/date", "date", NULL);
```

The first argument to **execl** is the *filename* of the command; you have to know where it is found in the file system. The second argument is conventionally the program name (that is, the last component of the filename), but this is seldom used except as a placeholder. If the command takes arguments, they are strung out after this; the end of the list is marked by a `NULL` argument.

The **execl** call overlays the existing program with the new one, runs that, then exits, without returning to the original program.

NOTE

> When a multithreaded process calls **exec**, the new process will be created with a single thread (and LWP), effectively terminating all other threads (and LWPs) in the process. If **exec** fails, no threads (or LWPs) are terminated.

The one exception to the rule that the original program never gets control back occurs when there is an error, for example if the file can't be found or is not executable. If you don't know where `date` is located, say:

```
    execl("/bin/date", "date", NULL);
    execl("/usr/bin/date", "date", NULL);
    printf(stderr, "Someone stole 'date'\n");
```

A variant of **execl** called **execv** is useful when you don't know in advance how many arguments there are going to be. The call is:

    execv(*filename*, *argp*);

Where *argp* is an array of pointers to the arguments; the last pointer in the array must be `NULL` so `execv` can tell where the list ends. As with execl, *filename* is the file in which the program is found, and *argp[0]* is the name of the program. (This arrangement is identical to the *argv* array for C program arguments.)

Neither of these functions provides the niceties of normal command execution. There is no automatic search of multiple directories; you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like "<", ">", "*", "?" and "`[]`" in the argument list. If you want these, use `execl` to invoke the shell **sh**, which then does all the work. Construct a string `cmdline` that contains the complete command as it would have been typed at the terminal, then say:

```
    execl("/bin/sh", "sh", "-c", cmdline, NULL);
```

The shell is assumed to be at a fixed place, **/bin/sh**. Its argument "-c" says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in cmdline.

To summarize:

- Any process may **exec** (cause execution of) a file.
- Doing an **exec** does not change the process ID; the process that did the **exec** persists, but after the **exec** it is executing a different program.
- Files that were open before the **exec** remain open afterwards.

Many programs want to regain control after **exec**ing another program; these should use a combination of **fork** and **exec** (see the next section). However, a program with two or more phases that communicate only through temporary files might use an **exec** function without a **fork**. Here it is natural to make the second pass simply an **execl** call from the first. For example, the first pass of a compiler might overlay itself with the second pass of the compiler. This is analogous to a "goto" in programming.

Process Creation—fork

If a process wishes to regain control after **exec**ing a second program, it should fork a child process, have the child **exec** the second program, and the parent **wait** for the child. This is analogous to a "call" except that the fork system call creates a new process that is an exact copy of the calling process. The following figure depicts what is involved in executing a program with a typical **fork** as the first step:

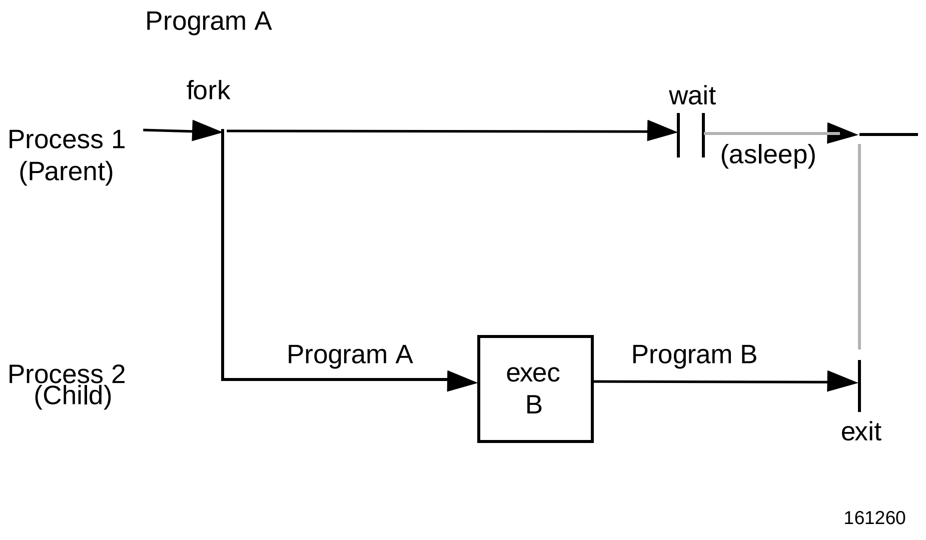


Figure 4-2. Process Primitives

Because the **exec** functions simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new overlaying program to finish. The system call **`fork`** does the splitting as in the following call:

```
    proc_id = fork();
```

The newly created process, known as the *child process*, is a copy of the image of the original process, called the *parent process*. The system call **`fork`** splits the program into two copies, both of which continue to run, and which differ only in the value returned in `proc_id`. In the child process, `proc_id` equals zero; in the parent process, `proc_id` equals a non-zero value that is the process number of the child process. Thus, the basic way to call, and return from, another program is:

```
    if (fork() == 0)   /* in child */
       execl(“/bin/sh”, “sh”, “-c”, cmd, NULL);
```

And in fact, except for handling errors, this is sufficient. The **`fork`** is zero, so it calls **`execl`**, which does the *cmd* and then dies. In the parent, **`fork`** returns non-zero so it skips the **`execl`**. (If there is any error, **`fork`** returns `-1`.)

A child inherits its parent's permissions, working-directory, root-directory, open files, etc. This mechanism permits processes to share common input streams in various ways. Files that were open before the **`fork`** are shared after the **`fork`**. The processes are informed through the return value of **`fork`** as to which is the parent and which is the child. In any case the child and parent differ in three important ways:

- The child has a different process ID.
- The child has a different parent process ID.
- All accounting variables are reset to appropriate values in the child.

NOTE

The functionality of **`fork(2)`** in a multithreaded program differs depending upon whether POSIX threads or PowerMAX OS threads are being used. See the “Using **`fork(2)`**” section in the “Programming with the Threads Library” chapter for more details.

The **`fork`** system call creates a child process with code and data copied from the parent process that created the child process. Once the copying is completed, the new (child) process is placed on the runnable queue to be scheduled. Each child process executes independently of its parent process, although the parent may explicitly wait for the termination of that child or any of its children. Usually the parent waits for the death of its child at some point because this **`wait`** call is used to free the process-table entry used by the child. See the discussion under “Process Termination” for more detail.

Calling **`fork`** creates a new process that is an exact copy of the calling process. The one major difference between the two processes is that the child gets its own unique process ID. When the **`fork`** process has completed successfully, it returns a `0` to the child process and the child's process ID to the parent. If the idea of having two identical processes seems a little funny, consider this:

- Because the return value is different between the child process and the parent, the program can contain the logic to determine different paths.
- The child process could say, "Okay, I'm the child; I'm supposed to issue an **exec** for an entirely different program."
- The parent process could say, "My child is going to **exec** a new process; I'll issue a **wait** until I get word that the new process is finished."

Your code might include statements like the following:

```
#include <errno.h>

pid_t ch_pid;
int ch_stat, status;
char *p_arg1, *p_arg2;
void exit();

    if ((ch_pid = fork()) < 0) {

        /* Could not fork... check errno */

    }
    else if (ch_pid == 0) {/* child */
        (void)execl("/usr/bin/prog2", "prog", p_arg1, p_arg2, (char *)NULL);
        exit(2);/* execl() failed */
    }
    else {      /* parent */
        while ((status = wait(&ch_stat)) != ch_pid) {
            if (status < 0 && errno == ECHILD)
                break;
            errno = 0;
        }
    }
```

Figure 4-3. Example of fork

Because the new **exec**'d process takes over the child process ID, the parent knows the ID. What this boils down to is a way of leaving one program to run another, returning to the point in the first program where processing left off.

Keep in mind that the fragment of code above includes minimal checking for error conditions, and has potential for confusion about open files and which program is writing to a file. Leaving out the possibility of named files, the new process created by the **fork** or **exec** has the three standard files that are automatically opened: `stdin`, `stdout`, and `stderr`. If the parent has buffered output that should appear before output from the child, the buffers must be flushed before the fork. Also, if the parent and the child processes both read input from a stream, whatever is read by one process will be lost to the other. That is, once something has been delivered from the input buffer to a process the pointer has moved on.

Process creation is essential to the basic operation of the OS because each command run by the Shell executes in its own process. In fact, execution of a Shell command or Shell procedure involves both a **fork** and an overlay. This scheme makes a number of services easy to provide. I/O redirection, for example, is basically a simple operation; it is performed entirely in the child process that executes the command, and thus no memory in the Shell parent process is required to rescind the change in standard input and output.

Background processes likewise require no new mechanism; the Shell merely refrains from waiting for commands executing in the background to complete. Finally, recursive use of the Shell to interpret a sequence of commands stored in a file is in no way a special operation.

Control of Processes—fork and wait

A parent process can suspend its execution to wait for termination of a child process with **`wait`** or **`waitpid`**. More often, the parent wants to wait for the child to terminate before continuing itself as follows:

```
int status;


if (fork() == 0)
   execl( ... );
wait(&status);
```

The previous code fragment avoids handling any abnormal conditions, such as a failure of the **`execl`** or **`fork`**, or the possibility that there might be more than one child running simultaneously. (The function **`wait`** returns the process-id of the terminated child, which can be checked against the value returned by **`fork`**.) In addition, this fragment avoids dealing with any funny behavior on the part of the child (which is reported in `status`).

The low-order eight bits of the value returned by **`wait`** encodes the termination status of the child process; `0` signifies normal termination and non-zero to signify various kinds of abnormalities. The next higher eight bits are taken from the argument of the call to **`exit`** that caused a normal termination of the child process. It is good coding practice for all programs to return meaningful status.

When a program is called by the shell, the three file-descriptors are available for use. When this program calls another one, correct etiquette suggest making sure the same conditions hold. Neither **`fork`** nor the **`exec`** calls affects open files in any way. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the **`execl`**. Conversely, if a caller buffers an input stream, the called program loses any information that has been read by the caller.

Process Termination

Processes terminate in one of two ways:

- Normal termination occurs by a return from **`main`** or when requested by an explicit call to **`exit`** or **`_exit`**.
- Abnormal termination occurs as the default action of a signal or when requested by **`abort`**.

On receiving a signal, a process looks for a signal-handling function. Failure to find a signal-handling function forces the process to call **`exit`**, and therefore to terminate. The functions **`_exit`**, **`exit`** and **`abort`** terminate a process with the same effects except that

abort makes available to **wait** or **waitpid** the status of a process terminated by the signal SIGABRT (see **exit(2)** and **abort(2)**).

As a process terminates, it can set an eight-bit exit status code available to its parent. Usually, this code indicates success (zero) or failure (non-zero), but it can be used in any manner the user wishes. If a signal terminated the process, the system first tries to dump an image of core, then modifies the exit code to indicate which signal terminated the process and whether core was dumped. This is provided that the signal is one that produces a core dump (see **signal(5)**). Next, all signals are set to be ignored, and resources owned by the process are released, including open files and the working directory. The terminating process is now a “zombie” process, with only its process-table entry remaining; and that is unavailable for use until the process has finally terminated. Next, the process-table is searched for any child or zombie processes belonging to the terminating process. Those children are then adopted by **init** by changing their parent process ID to 1). This is necessary because there must be a parent to record the death of the child. The last actions of **exit** are to record the accounting information and exit code for the terminated process in the zombie process-table entry and to send the parent the death-of-child signal, SIGCHLD. (see Chapter 10, “Signals, Job Control, and Pipes”).

If the parent wants to wait until a child terminates before continuing execution, the parent can call **wait**, which causes the parent to sleep until a child zombie is found (meaning the child terminated). When the child terminates, the death-of-child signal is sent to the parent although the parent ignores this signal. (Ignore is the default disposition. Applications that fork children and need to know the return status should set this signal to other than ignore.) The search for child zombies continues until the terminated child is found; at which time, the child's exit status and accounting information is reported to the parent (remember the call to **exit** in the child put this information in the child's process-table entry) and the zombie process-table entry is freed. Now the parent can wake up and continue executing.

Managing Processors and Processes

Processor Administration Information

Processors are identified with a processor ID number that gives them a unique tag within the system. The state of the processors in your system can be examined by using the **psrinfo(1M)** command or the **processor_info(2)** system call. They report whether the processor is on line or off line.

Binding Processes to Processors

By default, an LWP can execute on any processor in the system. Every LWP has a bit mask, or CPU bias, that determines the processor or processors on which it can be scheduled. An LWP inherits its CPU bias from its creator during a **fork(2)** or an **_lwp_create(2)** but may thereafter change it. The kernel assigns an LWP to a CPU in the LWP’s CPU bias. If an LWP’s CPU bias identifies more than one processor, the LWP’s CPU assignment may change several times during its execution. In making CPU assignments, the kernel attempts to balance the load on the CPUs and avoid unnecessary migration between CPUs.

The commands and program interfaces that allow you to set an LWP's CPU bias accept the following types of identifiers for the target LWP(s):

LWP ID	specifies a particular LWP
Process ID	specifies all LWPs in the process
Process group ID	specifies all LWPs in each process in the group
User ID	specifies all LWPs in each process owned by the user

You can set the CPU bias for one or more LWPs by using one of the following methods:

1. Invoke the **`cpu_bias(2)`** system call from a program, and specify the **`CPU_SETBIAS`** command.
2. Invoke the **`mpadvise(3C)`** library routine from a program, and specify the **`MPA_PRC_SETBIAS`** command.
3. Invoke the **`run(1)`** or **`rerun(1)`** command from the shell, and specify the **`–b`** *bias* option.

   If you wish to run a program with a particular CPU bias, use the **`run`** command. If you wish to change the CPU bias of a program that is already running, use the **`rerun`** command.

With **`mpadvise`**, **`cpu_bias`**, and **`rerun`**, the following conditions must be met:

- The real or effective user ID of the calling process must match the real or saved user ID of the LWP for which the bias is being set, or the calling process must have the `P_OWNER` privilege.
- To add a CPU to an LWP's CPU bias, the calling process must have the `P_CPUBIAS` privilege.

  Note that the `P_CPUBIAS` privilege is also required to use the **`run`** command for this purpose.

If the Enhanced Security Utilities are installed and running, the following additional conditions must be met:

The Mandatory Access Control (MAC) level of the calling process must equal the MAC level of the target process, or the calling process must have the `P_MACWRITE` privilege.

You can also change the CPU assignment of one or more LWPs by invoking **`cpu_bias`**, **`mpadvise`**, and **`rerun`**. Procedures are as follows:

1. Invoke the **`cpu_bias(2)`** system call from a program, specify the **`CPU_SETRUN`** command, and set only <u>one</u> bit in **mask*.
2. Invoke the **`mpadvise(3C)`** library routine from a program, specify the **`MPA_PRC_SETRUN`** command, and set only <u>one</u> bit in **mask*.
3. Invoke the **`rerun(1)`** command from the shell, and specify the **`–c`** *cpu_id* option.

To change an LWP's CPU assignment, the following conditions must be met:

- The real or effective user ID of the calling process must match the real or saved user ID of the LWP for which the CPU assignment is being changed, or the calling process must have the `P_OWNER` privilege.

If the Enhanced Security Utilities are installed and running, the following additional conditions must be met:

> The Mandatory Access Control (MAC) level of the calling process must equal the MAC level of the target process, or the calling process must have the `P_MACWRITE` privilege.

The System V **`processor_bind(3C)`** library routine and **`pbind(1M)`** utility allow you to assign a process (all of the associated LWPs) or LWP to a single processor. Both also allow you to remove a process's or LWP's previous CPU assignment. Once a process's or LWP's assignment has been removed, the process or LWP can execute on any processor in the system. To use **`processor_bind`** or **`pbind`** for these purposes, the real or effective user ID of the calling process must match the real or saved user ID of the target process or LWP, or the calling process must have the `P_OWNER` privilege.

For additional information on the use of **`cpu_bias(2)`**, **`mpadvise(3C)`**, **`run(1)`**, **`rerun(1)`**, **`processor_bind(3C)`**, and **`pbind(1M)`**, refer to the corresponding system manual pages.

Local Memory Considerations

On NightHawk platforms with more than one local memory pool, there is an additional restriction that must be observed in order to successfully migrate a process or LWP between two different CPU boards.

A process may not have any writable data pages of its address space locked into memory if they are located in a local memory pool when attempting to migrate that process, or a LWP within that process, to another CPU board. (For more information on NUMA policies, see Chapter 6 Memory Management and the **`memory(7)`** man page.)

Therefore, applications that wish to migrate across CPU boards should perform the migration before memory locking any writable data pages into local memory, or temporarily unlock those pages until the migration completes.

The **`memcntl(2)`**, **`mlock(3C)`**, **`mlockall(3C)`** and **`plock(2)`** functions and system services may be used to lock pages into memory. (For more information on memory page locking, see Chapter 6, “Memory Management”, in this manual.)

Also note that rescheduling variables (**`resched_cntl(2)`**) will cause a private writable data user page to be locked down in memory when a rescheduling variable is setup. (For more information on rescheduling variables, see Chapter 6 “Interprocess Synchronization” of the *PowerMAX OS Real-Time Guide*.)

5
Process Scheduling and Management

5
Process Scheduling and Management

This chapter provides an overview of process scheduling on PowerMAX OS systems. It explains how the process scheduler works and describes System V scheduler classes, POSIX scheduling policies, and scheduler priorities. It explains the procedures for using the program interfaces and commands that support process scheduling and management and describes scheduler interaction with such functions as **`fork`**, **`exec`**, and **`init`**. It also highlights performance issues.

Process Scheduling

In the OS, the schedulable entity is always a lightweight process (LWP). Scheduling priorities and classes are attributes of LWPs and not processes. When scheduling program interfaces accept a process on which to operate, the operation is applied to each LWP in the process. The system scheduler determines when LWPs run. It maintains priorities based on configuration parameters, process behavior, and user requests; it uses these priorities as well as other factors to assign LWPs to the CPU.

The OS gives users absolute control over the sequence in which certain LWPs run and the amount of time each LWP may use the CPU before another LWP gets a chance.

By default, the scheduler uses the time-sharing class. The time-sharing class adjusts priorities dynamically in an attempt to provide good response time to interactive LWPs and good throughput to CPU-intensive LWPs. The fixed class is similar to the time-sharing class except that priorities and time slices are not dynamically adjusted over time.

The scheduler offers a fixed-priority scheduling class as well as a time-sharing and a fixed class. Fixed-priority scheduling allows users to set fixed priorities on a per-process or LWP basis. The highest-priority fixed-priority LWP always gets the CPU as soon as it is runnable, even if system processes are runnable. An application can therefore specify the exact order in which LWPs run. An application may also be written so that its fixed-priority LWPs have a guaranteed response time from the system.

For system environments in which real-time performance is not required, the default scheduler configuration works well, and no fixed-priority LWPs are needed: administrators should not change configuration parameters, and users should not change scheduler properties of their applications. However, for real-time applications or applications with strict timing constraints, fixed-priority LWPs are the only way to guarantee that the application's requirements are met.

NOTE

Fixed-priority LWPs used carelessly can have a dramatic negative effect on the performance of time-sharing LWPs and the system in general.

This chapter is addressed to programmers who need more control over order of process and LWP execution than they get using default scheduler parameters.

Because changes in scheduler administration can affect scheduler behavior, programmers may also need to know about scheduler administration. For administrative information on the scheduler, see the *System Administration Volume 2* manual. There are also the following system manual pages with information on scheduler administration:

`dispadmin(1M)`	tells how to change scheduler configuration in a running system
`ts_dptbl(4)`	describes the time-sharing dispatcher parameter table that is used to configure the scheduler
`fc_dptbl(4)`	describes the fixed-class dispatcher parameter table that is used to configure the scheduler
`fp_dptbl(4)`	describes the fixed-priority dispatcher parameter tables that are used to configure the scheduler

How the Process Scheduler Works

Figure 5-1 shows how the OS process and LWP scheduler works. Fixed-class priorities overlap the default time-sharing priorities

When a process or LWP is created, it inherits its scheduler parameters, including scheduler class and a priority within that class. A process or LWP changes class only as a result of a user request. The system manages the priority of an LWP based on user requests and the scheduler class of the LWP.

In the default configuration, the initialization process belongs to the time-sharing class. Because processes inherit their scheduler parameters, all user login shells begin as time-sharing processes in the default configuration.

The scheduler converts class-specific priorities into global priorities. The global priority of an LWP determines when it runs—the scheduler always runs the runnable LWP with highest global priority. Numerically higher priorities run first. Once the scheduler assigns an LWP to the CPU, the LWP runs until it uses up its time slice, sleeps, or is preempted by a higher-priority LWP. LWPs with the same priority run round-robin.

Administrators specify default time slices in the configuration tables, but users may assign time slices to fixed-priority LWPs by using the **`priocntl(1)`** command or the **`priocntl(2)`** system call.

You can display the global priority of a process or LWP with the **`ps(1)`** command. You can display configuration information about class-specific priorities with the **`priocntl(1)`** command and the **`dispadmin(1M)`** command.

By default, all fixed-priority processes or LWPs have higher priorities than any system process, and all system processes have higher priorities than any time-sharing process.

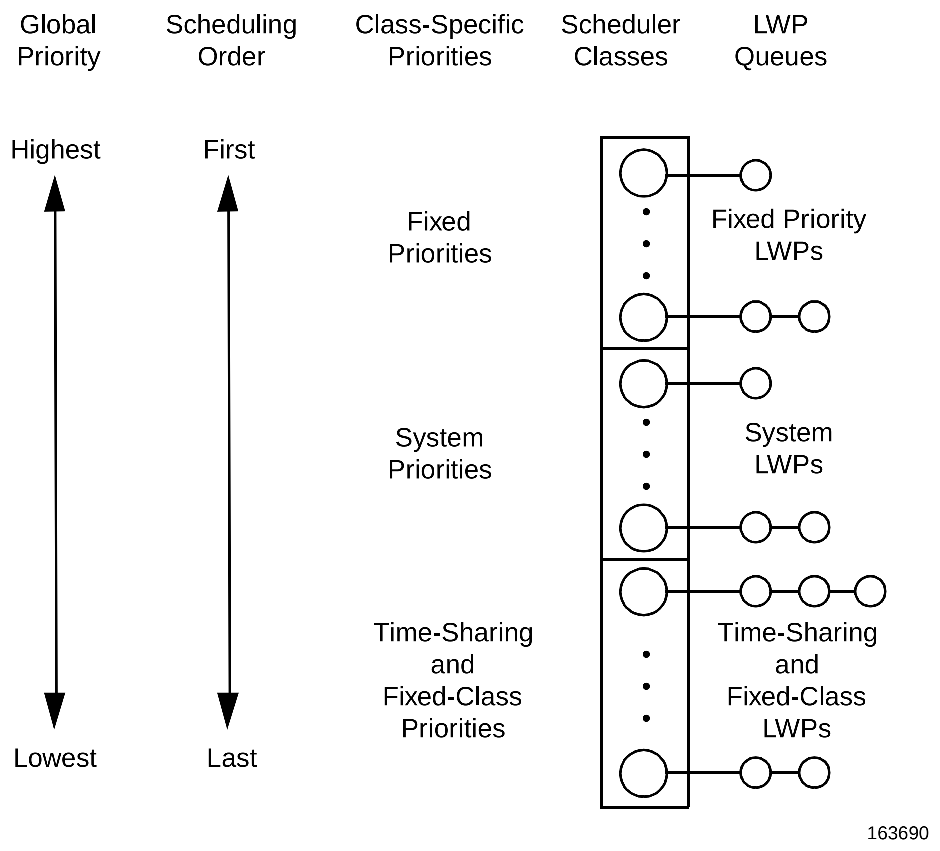


Figure 5-1. The PowerMAX OS Scheduler

NOTE

As long as a runnable fixed-priority process or LWP is available for a particular processor, no system process and no time-sharing process will run on that processor.

The sections that follow describe System V scheduler classes, POSIX scheduling policies, and scheduler priorities.

System V Scheduler Classes

System V defines four scheduler classes: the time-sharing class, the fixed class, the system class, and the fixed-priority class. PowerMAX OS defines an additional class reserved for the internal use of the ADA runtime environment. The sections that follow describe each of the classes.

Time-Sharing Class and Fixed Class

The goal of the time-sharing class is to provide good response time to interactive processes and LWPs and good throughput to CPU-bound processes and LWPs. The scheduler switches CPU allocation frequently enough to provide good response time, but not so frequently that it spends too much time doing the switching. Time slices are typically a few hundred milliseconds.

The time-sharing class changes priorities dynamically and assigns time slices of different lengths. The scheduler raises the priority of an LWP that sleeps after only a little CPU use (an LWP sleeps, for example, when it starts an I/O operation such as a terminal read or a disk read); frequent sleeps are characteristic of interactive tasks such as editing and running simple shell commands. On the other hand, the time-sharing class lowers the priority of an LWP that uses the CPU for long periods without sleeping.

The default time-sharing class gives larger time slices to LWPs with lower priorities. An LWP with a low priority is likely to be CPU-bound. Other LWPs get the CPU first, but when a low-priority LWP finally gets the CPU, it gets a bigger chunk of time. If a higher-priority LWP becomes runnable during a time slice, however, it preempts the running process or LWP.

The scheduler manages time-sharing processes and LWPs using configurable parameters in the time-sharing parameter table **`ts_dptbl(4).`** This table contains information specific to the time-sharing class.

The default fixed class is similar to the default time-sharing class except that the priorities and time slices given to fixed-class processes or LWPs are not dynamically changed over time. The **`fc_dptbl(4)`** parameter table contains information specific to the fixed class.

System Class

The system class uses fixed priorities to run kernel processes such as servers and housekeeping processes such as the paging daemon. The system class is reserved for use by the kernel; users may neither add a process to nor remove a process from the system class. Priorities for system class processes are set up in the kernel code for those processes; once established, the priorities of system processes do not change. (User processes and LWPs running in kernel mode are not in the system class.)

Fixed-Priority Class

With the fixed-priority class, critical processes and LWPs can run in predetermined sequence. Fixed priorities never change except when a user requests a change. Contrast this fixed-priority class with the time-sharing class, for which the system changes priorities to provide good interactive response time.

Privileged users can use the **`run(1)`**, **`rerun(1)`**, and **`priocntl(1)`** commands or the **`sched_setscheduler(3C)`** and **`priocntl(2)`** program interfaces to assign an LWP to the fixed-priority class. (See “Scheduler Commands,” p. 5-31, for information on the commands and “POSIX Scheduling Routines” and “System V Scheduling System Calls and Commands,” pp. 5-8 and 5-16, for information on the program interfaces.)

The scheduler manages fixed-priority processes and LWPs using configurable parameters in the fixed-priority parameter table **`fp_dptbl(4).`** This table contains information specific to the fixed-priority class.

ADA Priority Class

PowerMAX OS reserves the AD scheduling class for the internal use of the ADA runtime environment; applications should not directly use it.

POSIX Scheduling Policies

POSIX defines three types of scheduling policies that control the way a process is scheduled:

SCHED_FIFO	first–in–first–out (FIFO) scheduling policy
SCHED_RR	round–robin (RR) scheduling policy
SCHED_OTHER	time-sharing scheduling policy

Each of the POSIX scheduling policies is associated with one of the System V scheduler classes described in the preceding sections.

The **SCHED_FIFO** and **SCHED_RR** policies are associated with the fixed-priority class. These policies are almost identical. The only difference is that a process scheduled under the **FIFO** policy does not have an associated time quantum. As a result, as long as a process scheduled under the **FIFO** policy is the highest priority process scheduled on a particular CPU, it will continue to execute until it voluntarily blocks. A process that is scheduled under the **SCHED_RR** policy has an associated time quantum; the system default time quantum is defined in the fixed-priority parameter table **`fp_dptbl(4)`**.

The **SCHED_OTHER** policy is associated with the time-sharing class.

Each POSIX scheduling policy has a range of priority values associated with it. The priority range for the **SCHED_FIFO** and **SCHED_RR** policies is the same as that for the fixed-priority class. The priority range for the **SCHED_OTHER** policy is the same as that for the time-sharing class. The “Scheduler Priorities” section that follows provides an overview of scheduler priorities and explains the class-specific priority ranges.

A set of library routines that is based on IEEE Standard 1003.1b provides you with direct access to a process’s scheduling policy and priority. Included in the set are routines that allow processes to obtain or set a process’s scheduling policy and priority; obtain the minimum and maximum priorities associated with a particular scheduling policy; and obtain the time quantum associated with a process scheduled under the **SCHED_RR** scheduling policy. You may alter the scheduling policy and the scheduling parameters associated with that policy for an LWP or process by using the **`run(1)`** or **`rerun(1)`** command. Procedures for using the POSIX scheduling routines are explained in “POSIX Scheduling Routines” (p. 5-8). Procedures for using the **`run`** and **`rerun`** commands are explained in “Scheduler Commands” (p. 5-31).

Scheduler Priorities

Figure 5-2 presents a programmer's view of default LWP priorities. Fixed-class priorities overlap the default time-sharing priorities.

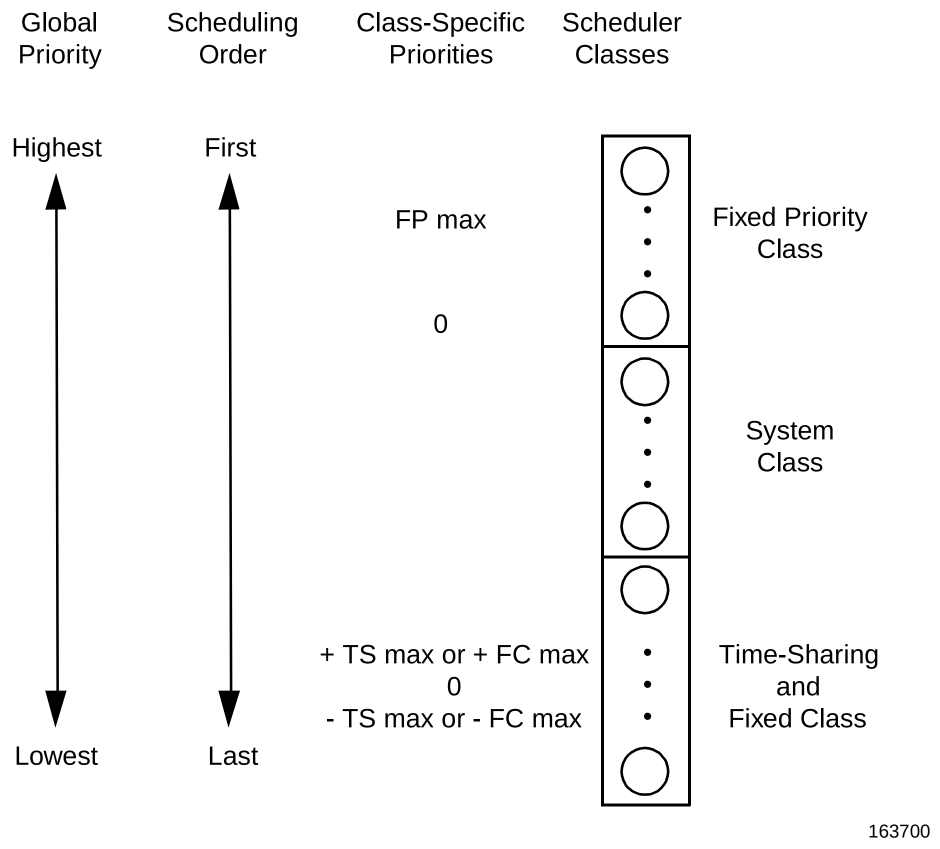


Figure 5-2. Process Priorities (Programmer's View)

From a user or programmer's point of view, a process or LWP priority has meaning only in the context of a scheduler class. You specify an LWP priority by specifying a class and a class-specific priority value. The class and class-specific value are mapped by the system into a global priority that the system uses to schedule LWPs.

- Fixed priorities run from zero to a configuration-dependent maximum. The system maps them directly into global priorities. They never change except when a user changes them.
- System priorities are controlled entirely in the kernel. Users cannot affect them.
- Time-sharing priorities have a user-controlled component (the *user priority*) and a component controlled by the system. The system does not change the user priority except as the result of a user request. The system changes the system-controlled component dynamically on a per-process or LWP basis to provide good overall system performance; users cannot affect the system-controlled component. The scheduler combines these two components to get the process or LWP global priority.

  The user priority runs from the negative of a configuration-dependent maximum to the positive of that maximum. A process or LWP inherits its user priority. Zero is the default initial user priority.

The *user priority limit* is the configuration-dependent maximum value of the user priority. You may set a user priority to any value below the user priority limit. With the `P_TSHAR` privilege, you may raise the user priority limit. Zero is the default user priority limit. You can raise the user priority limit by using the **`priocntl(2)`** system call (see “The priocntl System Call,” p. 5-17).

You may lower the user priority of a process or LWP to give the process or LWP reduced access to the CPU, or you may raise the user priority to get better service. Because you cannot set the user priority above the user priority limit, you must raise the user priority limit before you raise the user priority if both have their default values of zero. Note that you must have the `P_TSHAR` privilege to raise the user priority limit.

An administrator configures the maximum user priority independent of global time-sharing priorities. In the default configuration, for example, a user may set a user priority only in the range from -20 to +20, but 60 time-sharing global priorities are configured.

- Fixed-class priorities have a user-controlled component (the *user priority*) and a component controlled by the system. The system does not change the user priority except as the result of a user request. The scheduler combines these two components to get the process or LWP global priority.

  The user priority runs from the negative of a configuration-dependent maximum to the positive of that maximum. A process or LWP inherits its user priority. Zero is the default initial user priority.

  The *user priority limit* is the configuration-dependent maximum value of the user priority. You may set a user priority to any value below the user priority limit. With the `P_TSHAR` privilege, you may raise the user priority limit. Zero is the default user priority limit. You can raise the user priority limit by using the **`priocntl(2)`** system call (see “The priocntl System Call,” p. 5-17).

  You may lower the user priority of a process or LWP to give the process or LWP reduced access to the CPU, or you may raise the user priority to get better service. Because you cannot set the user priority above the user priority limit, you must raise the user priority limit before you raise the user priority if both have their default values of zero. Note that you must have the `P_TSHAR` privilege to raise the user priority limit.

  An administrator configures the maximum user priority independent of global fixed-class priorities. In the default configuration, for example, a user may set a user priority only in the range from -20 to +20, but 60 fixed-class global priorities are configured.

A system administrator's view of priorities is different from that of a user or programmer. When configuring scheduler classes, an administrator deals directly with global priorities. The system maps priorities supplied by users into these global priorities. See *System Administration Volume 2* for additional detail.

POSIX Scheduling Routines

The sections that follow explain the procedures for using the POSIX scheduling routines. These routines are briefly described as follows:

`sched_setscheduler`	set a process's scheduling policy and priority
`sched_getscheduler`	obtain a process's scheduling policy
`sched_setparam`	change a process's scheduling priority
`sched_getparam`	obtain a process's scheduling priority
`sched_yield`	relinquish the CPU
`sched_get_priority_min`	obtain the lowest priority associated with a scheduling policy
`sched_get_priority_max`	obtain the highest priority associated with a scheduling policy
`sched_rr_get_interval`	obtain the time quantum associated with a process scheduled under the **SCHED_RR** scheduling policy

NOTE

The POSIX scheduling routines that are listed above should only be used for singled-threaded applications or for multi-LWP applications that directly call **`_lwp_create(2)`** to create the LWPs within the process.

Multi-threaded applications that are linked with the thread library should make use of the thread library routines that get or set a thread's scheduling class and priority, instead of the routines listed above.

When a process (all of the associated LWPs) is scheduled under the **SCHED_FIFO** or **SCHED_RR** scheduling policy, it is assigned to the fixed-priority scheduler class. The priority value specified on the call to the POSIX scheduling routine is the same priority value that would be specified if a **`priocntl(2)`** call were made to set the process's priority within the fixed-priority class.

When a process (all of the associated LWPs) is scheduled under the **SCHED_OTHER** scheduling policy, it is assigned to the time-sharing scheduler class. The priority value specified on the call to the POSIX scheduling routine is the same priority value that would be specified if a **`priocntl(2)`** call were made to set the process's priority within the time-sharing class.

The sched_setscheduler Routine

The **`sched_setscheduler(3C)`** library routine allows you to set the scheduling policy and priority of a specified process.

It is important to note that to use the **`sched_setscheduler`** call to (1) change a process's scheduling policy to the **SCHED_FIFO** or the **SCHED_RR** policy or (2) change the priority of a process scheduled under the **SCHED_FIFO** or the **SCHED_RR** policy, the following conditions must be met:

- The calling process must have the `P_RTIME` privilege
- The effective user ID of the calling process must match the effective user ID of the target process (the process for which the scheduling policy and priority are being set), or the calling process must have the `P_OWNER` privilege.

If the Enhanced Security Utilities are installed and running, the following additional conditions must be met:

> The Mandatory Access Control (MAC) level of the calling process must equal the MAC level of the target process, or the calling process must have the `P_MACWRITE` privilege.

To use **`sched_setscheduler`** to raise the priority of a process scheduled under the **SCHED_OTHER** policy above a per-process or LWP limit, the following conditions must be met:

- The calling process must have the `P_TSHAR` privilege.
- The effective user ID of the calling process must match the effective user ID of the target process (the process for which the scheduling priority is being set), or the calling process must have the `P_OWNER` privilege.

If the Enhanced Security Utilities are installed and running, the following additional conditions must be met:

> The Mandatory Access Control (MAC) level of the calling process must equal the MAC level of the target process, or the calling process must have the `P_MACWRITE` privilege.

The specifications required for making the **`sched_setscheduler`** call are as follows:

```
#include <sched.h>

int sched_setscheduler(pid, policy, param)

pid_t pid;
int policy;
struct sched_param *param;
```

The arguments are defined as follows:

pid	the process identification number (PID) of the process for which the scheduling policy and priority are being set. To specify the current process, set the value of *pid* to zero.

policy	a scheduling policy as defined in the file <**`sched.h`**>. The value of *policy* must be one of the following:

SCHED_FIFO	first–in–first–out (FIFO) scheduling policy
SCHED_RR	round–robin (RR) scheduling policy. Note that a process cannot be scheduled under this policy on a CPU on which servicing of the 60 Hz clock interrupt has been disabled. In such cases, the process will behave as though it were scheduled under the **SCHED_FIFO** policy.
SCHED_OTHER	time-sharing scheduling policy

param	a pointer to a structure that specifies the scheduling priority of the process identified by *pid*. The priority is an integer value that lies in the range of priorities defined for the scheduler class associated with the specified policy. You can determine the range of priorities associated with that policy by invoking the **`run(1)`** command from the shell and not specifying any options or arguments (see the corresponding system manual page for an explanation of this command). You can also do so by invoking one of the following routines: **`sched_get_priority_min`** or **`sched_get_priority_max`** (for an explanation of these routines, see pages 5-14 and 5-14, respectively).

If the scheduling policy and priority of the specified process are successfully set, the **`sched_setscheduler`** routine returns the process's previous scheduling policy. A return value of **`–1`** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`sched_setscheduler(3C)`** system manual page for a listing of the types of errors that may occur. If an error occurs, the process's scheduling policy and priority are not changed.

It is important to note that when you change a process's scheduling policy, you also change its time quantum to the default time quantum that is defined for the scheduler class associated with the new policy and the priority. You can change the time quantum for a process or LWP scheduled under the **SCHED_RR** scheduling policy by using the **`run(1)`** or **`rerun(2)`** command (see p.5-36 for information on these commands).

NOTE

The sched_setscheduler routine should not be used to modify the LWPs in a multithreaded process. See the **`sched_setscheduler(3C)`** man page for more details on this subject.

The sched_getscheduler Routine

The **`sched_getscheduler(3C)`** library routine allows you to obtain the scheduling policy for a specified process. Scheduling policies are defined in the file <**`sched.h`**> as follows:

SCHED_FIFO	first–in–first–out (FIFO) scheduling policy
SCHED_RR	round–robin (RR) scheduling policy

SCHED_OTHER	time-sharing scheduling policy

The specifications required for making the **`sched_getscheduler`** call are as follows

```
#include <sched.h>

int sched_getscheduler(pid)

pid_t pid;
```

The argument is defined as follows:

pid	the process identification number (PID) of the process for which you wish to obtain the scheduling policy. To specify the current process, set the value of *pid* to zero.

If the call is successful, **`sched_getscheduler`** returns the scheduling policy of the specified process. A return value of **`–1`** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`sched_getscheduler(3C)`** system manual page for a listing of the types of errors that may occur.

Note

Multi-threaded applications should use the thread library function calls for obtaining a thread's scheduling policy.

The sched_setparam Routine

The **`sched_setparam(3C)`** library routine allows you to change the scheduling priority of a specified process.

It is important to note that to use the **`sched_setparam`** call to change the scheduling priority of a process scheduled under the **SCHED_FIFO** or the **SCHED_RR** policy, the following conditions must be met:

- The calling process must have the `P_RTIME` privilege.
- The effective user ID of the calling process must match the effective user ID of the target process (the process for which the scheduling policy and priority are being set), or the calling process must have the `P_OWNER` privilege.

If the Enhanced Security Utilities are installed and running, the following additional conditions must be met:

The Mandatory Access Control (MAC) level of the calling process must equal the MAC level of the target process, or the calling process must have the `P_MACWRITE` privilege.

If you wish to raise the priority of a process scheduled under the **SCHED_OTHER** policy above a per-process or LWP limit, the following conditions must be met:

- The calling process must have the `P_TSHAR` privilege.
- The effective user ID of the calling process must match the effective user ID of the target process (the process for which the scheduling priority is being set), or the calling process must have the `P_OWNER` privilege.

If the Enhanced Security Utilities are installed and running, the following additional conditions must be met:

The Mandatory Access Control (MAC) level of the calling process must equal the MAC level of the target process, or the calling process must have the `P_MACWRITE` privilege.

The specifications required for making the **`sched_setparam`** call are as follows:

```
#include <sched.h>

int sched_setparam(pid, param)

pid_t pid;
struct sched_param *param;
```

The arguments are defined as follows:

pid	the process identification number (PID) of the process for which the scheduling priority is being changed. To specify the current process, set the value of *pid* to zero.
param	a pointer to a structure that specifies the scheduling priority of the process identified by *pid*. The priority is an integer value that lies in the range of priorities associated with the process's current scheduling policy. High numbers represent more favorable priorities and scheduling. You can obtain a process's scheduling policy by invoking the **`sched_getscheduler(3C)`** routine (see p. 5-9 for an explanation of this routine). You can determine the range of priorities associated with that policy by invoking the **`run(1)`** command from the shell and not specifying any options or arguments (see the corresponding system manual page for an explanation of this command). You can also obtain the range of priorities associated with a policy by invoking the **`sched_get_priority_min(3C)`** and **`sched_get_priority_max(3C)`** routines (see p.5-14 for explanations of these routines).

A return value of `0` indicates that the scheduling priority of the specified process has been successfully changed. A return value of `-1` indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`sched_setparam(3C)`** system manual page for a listing of the types of errors that may occur. If an error occurs, the process's scheduling priority is not changed.

NOTE

The sched_setparam routine should not be used to modify the LWPs in a multithreaded process. See the **`sched_setparam(3C)`** man page for more details on this subject.

The sched_getparam Routine

The **`sched_getparam(3C)`** library routine allows you to obtain the scheduling priority of a specified process.

The specifications required for making the **`sched_getparam`** call are as follows:

```
#include <sched.h>

int sched_getparam(pid, param)

pid_t pid;
struct sched_param *param;
```

The arguments are defined as follows:

pid	the process identification number (PID) of the process for which you wish to obtain the scheduling priority. To specify the current process, set the value of *pid* to zero.
param	a pointer to a structure to which the scheduling priority of the process identified by *pid* will be returned

A return value of **`0`** indicates that the call to **`sched_getparam`** has been successful. The scheduling priority of the specified process is returned in the structure to which *param* points. A return value of – **`1`** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`sched_getparam(3C)`** system manual page for a listing of the types of errors that may occur.

NOTE

Multi-threaded applications should use the thread library function calls for obtaining a thread's scheduling priority.

The sched_yield Routine

The **`sched_yield(3C)`** library routine allows the calling process to relinquish the CPU until it again becomes the highest priority process that is ready to run. Note that a call to **`sched_yield`** is effective only if a process whose priority is equal to that of the calling process is ready to run. This routine cannot be used to allow a process whose priority is lower than that of the calling process to execute.

The specifications required for making the **`sched_yield`** call are as follows:

```
#include <sched.h>

void sched_yield()
```

The **sched_yield** routine does not return a value.

The sched_get_priority_min Routine

The **sched_get_priority_min(3C)** library routine allows you to obtain the lowest (least favorable) priority associated with a specified scheduling policy.

The specifications required for making the **sched_get_priority_min** call are as follows:

```
#include <sched.h>
#include <timers.h>

int sched_get_priority_min(policy)

int policy;
```

The argument is defined as follows:

policy a scheduling policy as defined in the file <**sched.h**>. The value of *policy* must be one of the following:

SCHED_FIFO	first–in–first–out (FIFO) scheduling policy
SCHED_RR	round–robin (RR) scheduling policy
SCHED_OTHER	time-sharing scheduling policy

If the call is successful, **sched_get_priority_min** returns the lowest priority associated with the specified scheduling policy. A return value of **–1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the system manual page for **sched_get_priority_max(3C)** to obtain a listing of the types of errors that may occur.

The sched_get_priority_max Routine

The **sched_get_priority_max(3C)** library routine allows you to obtain the highest (most favorable) priority associated with a specified scheduling policy.

The specifications required for making the **sched_get_priority_max** call are as follows:

```
#include <sched.h>
#include <timers.h>

int sched_get_priority_max(policy)

int policy;
```

The argument is defined as follows:

policy	a scheduling policy as defined in the file <**sched.h**>. The value of *policy* must be one of the following:

SCHED_FIFO	first–in–first–out (FIFO) scheduling policy
SCHED_RR	round–robin (RR) scheduling policy
SCHED_OTHER	time-sharing scheduling policy

If the call is successful, **sched_get_priority_max** returns the highest priority associated with the specified scheduling policy. A return value of **–1** indicates that an error has occurred; **errno** is set to indicate the error. For a listing of the types of errors that may occur, refer to the **sched_get_priority_max(3C)** system manual page.

The sched_rr_get_interval Routine

The **sched_rr_get_interval(3C)** library routine allows you to obtain the time quantum for a process that is scheduled under the **SCHED_RR** scheduling policy. The time quantum is the fixed period of time for which the kernel allocates the CPU to a process. When the process to which the CPU has been allocated has been running for its time quantum, a scheduling decision is made. If another process of the same priority is ready to run, that process will be scheduled. If not, the other process will continue to run.

The specifications required for making the **sched_rr_get_interval** call are as follows:

```
#include <sched.h>
#include <timers.h>

int sched_rr_get_interval(pid, min)

pid_t pid;
struct timespec *min;
```

The arguments are defined as follows:

pid	the process identification number (PID) of the process for which you wish to obtain the time quantum. To specify the current process, set the value of *pid* to zero.
min	a pointer to a structure to which the time quantum of the process identified by *pid* will be returned.

A return value of **0** indicates that the call to **sched_rr_get_interval** has been successful. The time quantum of the specified process is returned in the structure to which *min* points. A return value of **–1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **sched_getpriority_max(3C)** system manual page for a listing of the types of errors that may occur.

System V Scheduling System Calls and Commands

The sections that follow explain the procedures for using the System V system calls and commands related to scheduling. These system calls are briefly described as follows:

`priocntl`	system call that obtains or sets the scheduler parameters of one or more processes or LWPs
`priocntllist`	system call that obtains or sets the scheduler parameters of a list of LWPs
`priocntlset`	system call that obtains or sets the scheduler parameters of a set of running processes
`setrun`	system command that defines the scheduling environment in which a command executes

These system calls set or retrieve scheduler parameters for processes and LWPs. Steps that you use to set priorities are similar for the first three functions (**`setrun`** works a little differently):

1. Specify the target processes and LWPs.
2. Specify the scheduler parameters you want for those processes and LWPs.
3. Do the command or system call to set the parameters for the processes and LWPs.

You specify the target processes and LWPs using an ID type and an ID. The ID type tells how to interpret the ID. (This concept of a set of processes and LWPs applies to signals as well as to the scheduler; see **`sigsend(2)`**). The valid ID types that you may specify are as follows.

- LWP ID
- Process ID
- Parent process ID
- Process group ID
- Session ID
- Class ID
- Effective user ID
- Effective group ID
- All processes

These IDs are basic properties of UNIX processes and LWPs. (See **`intro(2)`**). The class ID refers to the scheduler class of the process or LWP. The **`priocntl`** system call works only for the time-sharing, fixed, and fixed-priority classes, not for the system class.

Processes in the system class have fixed priorities assigned when they are started by the kernel. Such processes include system daemons. The only way that you can affect the priority of a system daemon is to change the value of the system tunable parameter associated with that daemon's scheduling priority. You can examine and modify the values of system tunable parameters associated with system daemons by using the **`config(1M)`** utility. For an explanation of the procedures for using this utility, refer to the "Configuring and Building the Kernel" chapter of *System Administration Volume 2.* Note that after changing a tunable parameter, you must rebuild the kernel and then reboot your system. If you wish to change the scheduling priority of all of the system daemons, you may use the **`daemon_tune(1M)`** command. In this case also, you must rebuild the kernel and then reboot your system.

NOTE

Global priorities and user-supplied priorities are in ascending order: numerically higher priorities run first.

The priocntl System Call

```
#include <sys/types.h>
#include <sys/procset.h>
#include <sys/priocntl.h>
#include <sys/fppriocntl.h>
#include <sys/tspriocntl.h>
#include <sys/fcpriocntl.h>

long priocntl(idtype_t idtype, id_t id, int cmd, void *arg);
```

The **`priocntl`** system call gets or sets scheduler parameters of a set of processes or LWPs. The input arguments are defined as follows:

- *idtype* is the type of ID you are specifying.
- *id* is the ID.
- *cmd* specifies which **`priocntl`** function to perform. The functions are listed in Table 5-2.
- *arg*, in most cases, is a pointer to a structure that depends on *cmd*. The type of *arg* for each value of *cmd* is listed in Table 5-2.

Table 5-1 presents the valid values for *idtype* that are defined in **`<sys/procset.h>`** and the corresponding interpretations of *id.*

Table 5-1. priocntl(2) idtype Values

idtype	Interpretation of *id*
`P_PID`	Process ID (of a single process)
`P_PPID`	Parent process ID
`P_PGID`	Process group ID
`P_LWPID`	LWP ID
`P_SID`	Session ID

Table 5-1. priocntl(2) idtype Values (Cont.)

idtype	Interpretation of *id*
`P_CID`	Scheduling class ID
`P_UID`	Effective user ID
`P_GID`	Effective group ID
`P_ALL`	All processes and LWPs

Table 5-2 shows the valid values for *cmd* as defined in <**`sys/priocntl.h`**>, their meanings, and the type of *arg.*

Table 5-2. priocntl(2) Commands

cmd	*arg* Type	Function
`PC_GETCID`	`pcinfo_t`	Get class ID and attributes
`PC_GETCLINFO`	`pcinfo_t`	Get class name and attributes
`PC_SETPARMS`	`pcparms_t`	Set class and scheduling parameters
`PC_GETPARMS`	`pcparms_t`	Get class and scheduling parameters
`PC_GETTQ`	`int`	Get time quantum
`PC_SETTQ`	`int`	Set time quantum

Following are the values **`priocntl`** returns on success:

- The `GETCID` and `GETCLINFO` commands return the number of configured scheduler classes.
- `PC_SETPARMS` returns `0`.
- `PC_GETPARMS` returns the process ID of the process or LWP whose scheduler properties it is returning.

On failure, **`priocntl`** returns `-1` and sets `errno` to indicate the reason for the failure. See **`priocntl(2)`** for the complete list of error conditions.

The PC_GETCID and PC_GETCLINFO Commands

The `PC_GETCID` and `PC_GETCLINFO` commands retrieve scheduler parameters for a class based on the class ID or class name. Both commands use the `pcinfo` structure to send arguments and receive return values:

```
typedef struct pcinfo {
   id_t  pc_cid;                 /* class id */
   char  pc_clname[PC_CLNMSZ];    /* class name */
   long  pc_clinfo[PC_CLINFOSZ];  /* class information */
} pcinfo_t;
```

The PC_GETCID command gets scheduler class ID and parameters given the class name. The class ID is used in some of the other **priocntl** commands to specify a scheduler class. The valid class names are TS for time-sharing, FC for fixed class, and FP for fixed priority.

For the fixed-priority class, pc_clinfo contains an fpinfo structure, which holds fp_maxpri, the maximum valid fixed priority; in the default configuration, this is the highest priority any process or LWP can have. The minimum valid fixed priority is zero. fp_maxpri is a configurable value; the “Process Scheduling” chapter of *System Administration Volume 2* explains how to configure process and LWP priorities.

```
typedef struct fpinfo {
   short  fp_maxpri;  /* maximum fixed priority */
} fpinfo_t;
```

For the time-sharing class, pc_clinfo contains a tsinfo structure, which holds ts_maxupri, the maximum time-sharing user priority. The minimum time-sharing user priority is -ts_maxupri. ts_maxupri is also a configurable value.

```
typedef struct tsinfo {
   short  ts_maxupri;  /*limits of user priority range */
} tsinfo_t;
```

For the fixed class, pc_clinfo contains an fcinfo structure, which holds fc_maxupri, the maximum fixed class user priority. The minimum fixed class user priority is -fc_maxupri. fc_maxupri is also a configurable value.

```
typedef struct fcinfo {
   short  fc_maxupri;  /*limits of user priority range */
} fcinfo_t;
```

The program shown in Screen 5-1 is a cheap substitute for **priocntl -l**; it gets and prints the range of valid priorities for the time-sharing and fixed-priority scheduler classes.

Screen 5-2 shows the output of the program shown in Screen 5-1. The program is called **getcid** in this example.

```
/*
 *  Get scheduler class IDs and priority ranges.
 */

#include <sys/types.h>
#include <sys/procset.h>
#include <sys/priocntl.h>
#include <sys/fppriocntl.h>
#include <sys/tspriocntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>

main ()
{
        pcinfo_t        pcinfo;
        tsinfo_t        *tsinfop;
        fpinfo_t        *fpinfop;
        short           maxtsupri, maxfppri;

   /* time sharing */
        (void) strcpy (pcinfo.pc_clname, “TS”);
        if (priocntl (0L, 0L, PC_GETCID, &pcinfo) == -1L) {
                perror (“PC_GETCID failed for time-sharing class”);
                exit (1);
        }
        tsinfop = (struct tsinfo *) pcinfo.pc_clinfo;
        maxtsupri = tsinfop->ts_maxupri;
        (void) printf(“Time sharing: ID %ld, priority range -%d through %d\n”,
                pcinfo.pc_cid, maxtsupri, maxtsupri);

   /* fixed priority */
        (void) strcpy(pcinfo.pc_clname, “FP”);
        if (priocntl (0L, 0L, PC_GETCID, &pcinfo) == -1L) {
                perror (“PC_GETCID failed for fixed priority class”);
                exit (2);
        }
        fpinfop = (struct fpinfo *) pcinfo.pc_clinfo;
        maxfppri = fpinfop->fp_maxpri;
        (void) printf(“Fixed priority:    ID %ld, \
                          priority range 0 through %d\n”,
                          pcinfo.pc_cid, maxfppri);


        return (0);
}
```

Screen 5-1. Obtaining the Range of Priorities for Scheduler Classes

```
$ getcid
Time sharing: ID 1, priority range -20 through 20
Fixed priority:    ID 2, priority range 0 through 59
```

Screen 5-2. Output from the getcid Program

The function shown in Screen 5-3 is useful in the examples presented in subsequent screens. Given a class name, it uses `PC_GETCID` to return the class ID and maximum priority in the class.

NOTE

All of the examples in the screens that follow omit the lines that include header files. The examples compile with the same header files as presented in Screen 5-1.

```
/*
 *  Return class ID and maximum priority.
 *  Input argument name is class name.
 *  Maximum priority is returned in *maxpri.
 */

id_t
schedinfo (name, maxpri)
        char *name;
        short *maxpri;
{
        pcinfo_t        info;
        tsinfo_t        *tsinfop;
        fpinfo_t        *fpinfop;

        (void) strcpy(info.pc_clname, name);
        if (priocntl (0L, 0L, PC_GETCID, &info) == -1L) {
                return (-1);
        }
        if (strcmp(name, “TS”) == 0) {
                tsinfop = (struct tsinfo *) info.pc_clinfo;
                *maxpri = tsinfop->ts_maxupri;
        } else if (strcmp(name, “FP”) == 0) {
                fpinfop = (struct fpinfo *) info.pc_clinfo;
                *maxpri = fpinfop->fp_maxpri;
        } else {
                return (-1);
        }
        return (info.pc_cid);
}
```

Screen 5-3. Obtaining the Class ID and the Maximum Priority

The `PC_GETCLINFO` command gets a scheduler class name and parameters given the class ID. This command makes it easy to write applications that make no assumptions about what classes are configured.

The program shown in Screen 5-4 uses `PC_GETCLINFO` to get the class name of a process or LWP based on the process ID. This program assumes the existence of a function `getclassID,` which retrieves the class ID of a process or LWP given the process ID; this function is shown in Screen 5-5.

```
/*  Get scheduler class name given process ID. */

main (argc, argv)
        int argc;
        char *argv[];
{
        pcinfo_t        pcinfo;
        id_t            pid, classID;
        id_t            getclassID();

        if ((pid = atoi(argv[1])) <= 0) {
                perror (“bad pid”);
                exit (1);
        }
        if ((classID = getclassID(pid)) == -1) {
                perror (“unknown class ID”);
                exit (2);
        }
        pcinfo.pc_cid = classID;
        if (priocntl (0L, 0L, PC_GETCLINFO, &pcinfo) == -1L) {
                perror (“PC_GETCLINFO failed”);
                exit (3);
        }
        (void) printf(“process ID %d, class %s\n”, pid, pcinfo.pc_clname);
}
```

Screen 5-4. Obtaining a Process’s Scheduler Class

The PC_GETPARMS and PC_SETPARMS Commands

The `PC_GETPARMS` command gets and the `PC_SETPARMS` command sets scheduler parameters for processes and LWPs. Both commands use the `pcparms` structure to send arguments or receive return values:

```
typedef struct pcparms {
   id_t  pc_cid;             /* process or LWP class */
   long  pc_clparms[PC_CLPARMSZ]; /* class specific */
} pcparms_t;
```

Ignoring class-specific information for the moment, a simple function for returning the scheduler class ID of a process is shown in Screen 5-5.

```
/*
 *  Return scheduler class ID of process with ID pid.
 */

getclassID (pid)
        id_t pid;
{
        pcparms_t       pcparms;

        pcparms.pc_cid = PC_CLNULL;
        if (priocntl(P_PID, pid, PC_GETPARMS, &pcparms) == -1) {
                return (-1);
        }
        return (pcparms.pc_cid);
}
```

Screen 5-5. Obtaining a Process's Scheduler Parameters

For the fixed-priority class, `pc_clparms` contains an `fpparms` structure. `fpparms` holds scheduler parameters specific to the fixed-priority class:

```
typedef struct fpparms {
   short   fp_pri;      /* fixed priority */
   ulong_t fp_tqsecs;   /* seconds in time quantum */
   long    fp_tqnsecs;  /* additional nsecs in quantum */
} fpparms_t;
```

`fp_pri` is the fixed priority. `fp_tqsecs` is the number of seconds, and `fp_tqnsecs` is the number of additional nanoseconds in a time slice; that is, `fp_tqsecs` seconds plus `fp_tqnsecs` nanoseconds is the interval an LWP may use the CPU without sleeping before the scheduler gives another LWP a chance at the CPU.

For the time-sharing class, `pc_clparms` contains a `tsparms` structure. `tsparms` holds the scheduler parameters specific to the time-sharing class:

```
typedef struct tsparms {
   short  ts_uprilim;  /* user priority limit */
   short  ts_upri;     /* user priority */
} tsparms_t;
```

`ts_uprilim` is the user priority limit, the maximum user priority a process or LWP may set for itself without being a privileged user. `ts_upri` is the user priority, the user-controlled component of a time-sharing priority. These values are described as follows:

- The user priority is the user-controlled component of a time-sharing or fixed-class priority. The scheduler calculates the global priority of a time-sharing or fixed-class process or LWP by combining this user priority with a system-controlled component that depends on process or LWP behavior. The user priority has the same effect as a value set by **`nice`** (except that **`nice`** uses higher numbers for lower priority).
- The user priority limit is the maximum user priority a process or LWP may set for itself without being a privileged user. By default, the user priority limit is 0. You must have the `P_TSHAR` privilege to set a user priority limit above 0.

Both the user priority and the user priority limit must be within the user priority range reported by the **priocntl -l** command. This range is also reported by the PC_GETCID and PC_GETCLINFO commands to the **priocntl** system call (see p. 5-18). The default range is -20 to +20 for both the time-sharing and fixed class.

There is no limit for the number of times a process or LWP may lower and raise its user priority, as long as the value is below its user priority limit. As a courtesy to other users, lower your user priority for big chunks of low-priority work. However, remember that if you lower your user priority limit, you must have the P_TSHAR privilege to raise it. A typical use of the user priority limit is to reduce permanently the priority of child processes or LWPs or another set of low-priority processes or LWPs.

The user priority can never be greater than the user priority limit. If you set the user priority limit below the user priority, the user priority is lowered to the new user priority limit. If you attempt to set the user priority above the user priority limit, the user priority is set to the user priority limit.

For the fixed class, pc_clparms contains an fcparms structure. fcparms holds scheduler parameters specific to the fixed class:

```
typedef struct fcparms {
   short  fc_uprilim;/* user priority limit */
   short  fc_upri;   /* user priority */
   long   fc_timeleft/* time-left for this lwp */
   short  fc_cpupri; /* assigned cpu priority */
   short  fc_umdpri; /* computed user mode priority */
} fcparms_t;
```

fc_uprilimit is the user priority limit, the maximum user priority a process or LWP may set for itself without being a privileged user. fc_upri is the user priority, the user-controlled component of a fixed-class priority. These values are described in the preceding paragraphs. fc_timeleft is the remaining amount of the time quantum. fc_cpupri is the system-controlled component of the fixed-class priority. fc_umdpri is the computed priority that is based on the value of fc_upri and fc_cpupri.

The PC_GETPARMS command gets the scheduler class and parameters of a single process or LWP. The return value of the **priocntl(2)** call is the process ID of the process or LWP whose parameters are returned in the pcparms structure. The process or LWP that is chosen depends on the *idtype* and *id* arguments to **priocntl** and on the value of pcparms.pc_cid, which contains a class ID returned by PC_GETCID or the constant **PC_CLNULL**. Table 5-3 shows the type of information that a PC_GETPARMS priocntl call returns in various circumstances.

Table 5-3. Information Returned by PC_GETPARMS

Number of Processes Selected by *idtype* and *id*	`pc_cid`			
	FP Class ID	TS Class ID	FC Class ID	PC_CLNULL
1	FP parameters of process or LWP selected	TS parameters of process or LWP selected	FC parameters of process or LWP selected	Class and parameters of process or LWP selected
More than 1	FP parameters of highest-priority FP process or LWP	TS parameters of process or LWP with highest user priority	FC parameters of process or LWP with highest user priority	(error)

If *idtype* and *id* select a single process or LWP and `pc_cid` does not conflict with the class of that process or LWP, **`priocntl`** returns the scheduler parameters of the process or LWP. If they select more than one process or LWP of a single scheduler class, **`priocntl`** returns parameters using class-specific criteria as shown in Table 5-3. **`priocntl`** returns an error in the following cases:

- The *idtype* and *id* arguments select one or more processes or LWPs, and none is in the class specified by `pc_cid.`
- The *idtype* and *id* arguments select more than one process or LWP, and `pc_cid` is PC_CLNULL.
- The *idtype* and *id* arguments select no processes or LWPs.

The program shown in Screen 5-6 takes a process ID as its input and prints the scheduler class and class-specific parameters of the highest priority LWP within that process.

The `PC_SETPARMS` command sets the scheduler class and parameters of a set of processes or LWPs. The *idtype* and *id* input arguments specify the processes or LWPs to be changed. The `pcparms` structure contains the new parameters: `pc_cid` contains the ID of the scheduler class to which the processes or LWPs are to be assigned, as returned by `PC_GETCID`; `pc_clparms` contains the class-specific parameters:

- If `pc_cid` is the fixed-priority class ID, `pc_clparms` contains an `fpparms` structure in which `fp_pri` contains the fixed priority and `fp_tqsecs` plus `fp_tqnsecs` contains the time slice to be assigned to the processes or LWPs.
- If `pc_cid` is the time-sharing class ID, `pc_clparms` contains a `tsparms` structure in which `ts_uprilim` contains the user priority limit and `ts_upri` contains the user priority to be assigned to the processes or LWPs.

```
/*
 *  Get scheduler class and parameters of
 *  process whose pid is input argument.
 */

main (argc, argv)
        int argc;
        char *argv[];
{
        pcparms_t       pcparms;
        fpparms_t       *fpparmsp;
        tsparms_t       *tsparmsp;
        id_t            pid, fpID, tsID;
        id_t            schedinfo();
        short           priority, tsmaxpri, fpmaxpri;
        ulong           secs;
        long            nsecs;

        pcparms.pc_cid = PC_CLNULL;
        fpparmsp = (fpparms_t *) pcparms.pc_clparms;
        tsparmsp = (tsparms_t *) pcparms.pc_clparms;
        if ((pid = atoi(argv[1])) <= 0) {
                perror (“bad pid”);
                exit (1);
        }

   /* get scheduler properties for this pid */
        if (priocntl(P_PID, pid, PC_GETPARMS, &pcparms) == -1) {
                perror (“GETPARMS failed”);
                exit (2);
        }

   /* get class IDs and maximum priorities for TS and FP */
        if ((tsID = schedinfo (“TS”, &tsmaxpri)) == -1) {
                perror (“schedinfo failed for TS”);
                exit (3);
        }
        if ((fpID = schedinfo (“FP”, &fpmaxpri)) == -1) {
                perror (“schedinfo failed for FP”);
                exit (4);
        }

   /* print results */

        if (pcparms.pc_cid == fpID) {
                priority = fpparmsp->fp_pri;
                secs = fpparmsp->fp_tqsecs;
                nsecs =  fpparmsp->fp_tqnsecs;
                (void) printf (“process %d: FP priority %d\n”,
                        pid, priority);
                (void) printf (“time slice %ld secs, %ld nsecs\n”,
                        secs, nsecs);
        } else if (pcparms.pc_cid == tsID) {
                priority = tsparmsp->ts_upri;
                (void) printf (“process %d: TS priority %d\n”,
                        pid, priority);
        } else {
                printf (“Unknown scheduler class %d\n”,
                        pcparms.pc_cid);
                exit (5);
        }
        return (0);
}
```

Screen 5-6. Obtaining a Process’s Scheduler Class and Parameters

- If `pc_cid` is the fixed class ID, `pc_clparms` contains an `fcparms` structure in which `fc_uprilim` contains the user priority limit; `fc_upri` contains the user priority to be assigned to the processes or LWPs; `fc_timeleft` contains the time quantum left (cannot be altered); `fc_cpupri` contains the system-controlled component of the fixed-class priority; and fc_umdpri contains the computed priority that is based on the value of fc_upri and fc_cpupri.

The program shown in Screen 5-7 takes a process ID as input, makes the process or LWP a fixed-priority process or LWP with the highest valid priority minus 1, and gives it the default time slice for that priority. The program calls the schedinfo function shown in Screen 5-3 to get the fixed-priority class ID and maximum priority.

```
/*
 *  Input arg is proc ID.  Make process or LWP a fixed priority
 *  process or LWP with highest priority minus 1.
 */

main (argc, argv)
        int argc;
        char *argv[];
{
        pcparms_t       pcparms;
        fpparms_t       *fpparmsp;
        id_t            pid, fpID;
        id_t            schedinfo();
        short           maxrtpri;
        short           maxfppri;

        if ((pid = atoi(argv[1])) <= 0) {
                perror (“bad pid”);
                exit (1);
        }

   /* Get highest valid FP priority. */
        if ((fpID = schedinfo (“FP”, &maxfppri)) == -1) {
                perror (“schedinfo failed for FP”);
                exit (2);
        }

   /*  Change proc to FP, highest prio - 1, default time slice */
        pcparms.pc_cid = fpID;
        fpparmsp = (struct fpparms *) pcparms.pc_clparms;
        fpparmsp->fp_pri = maxfppri - 1;
        fpparmsp->fp_tqnsecs = FP_TQDEF;

        if (priocntl(P_PID, pid, PC_SETPARMS, &pcparms) == -1) {
                perror (“PC_SETPARMS failed”);
                exit (3);
        }
}
```

Screen 5-7. Changing a Process’s Scheduler Class and Priority

The following table lists the special values fp_tqnsecs can take when PC_SETPARMS is used on fixed-priority processes and LWPs. When any of these is used, fp_tqsecs is ignored. These values are defined in the header file <**sys/fppriocntl.h>**.

fp_tqnsecs	Time Slice
FP_TQINF	Infinite
FP_TQDEF	Default
FP_NOCHANGE	Unchanged

FP_TQINF specifies an infinite time slice. FP_TQDEF specifies the default time slice configured for the fixed priority being set with the SETPARMS call. FP_NOCHANGE specifies no change from the current time slice; this value is useful, for example, when you change process or LWP priority but do not want to change the time slice. (You can also use `FP_NOCHANGE` in the `fp_pri` field to change a time slice without changing the priority.)

The PC_GETTQ and PC_SETTQ Commands

The `PC_GETTQ` command gets the time quantum associated with a single LWP that has been assigned to the fixed-priority, time-sharing, or fixed scheduler class. The LWP that is selected depends on the values of the *idtype* and *id* arguments. If *idtype* and *id* specify more than one LWP, **`priocntl`** returns the time quantum using the following class-specific criteria:

Fixed-priority class	quantum of the fixed-priority LWP with the highest priority
Time-sharing class	quantum of the time-sharing LWP with the highest user priority
Fixed class	quantum of the fixed-class LWP with the highest user priority

The **`priocntl`** `PC_GETTQ` call returns a positive integer value that represents the quantum in 60 Hz clock ticks <u>or</u> a negative integer value that indicates an infinite quantum.

The `PC_SETTQ` command sets the time quantum associated with processes or LWPs that have been assigned to the fixed-priority or the time-sharing scheduler class. The *idtype* and *id* input arguments specify the processes or LWPs for which the quantum is to be set. The *arg* argument specifies a positive integer value that represents the desired quantum in 60 Hz clock ticks <u>or</u> one of the symbolic constants that has been defined for the scheduler class as presented in Table 5-4.

Table 5-4. Symbolic Constants for Specifying Quantum

Quantum	Class-Specific Constants	
	Time-Sharing	Fixed-Priority
Infinite	`TS_TQINF`	`FP_TQINF`
Default	`TS_TQDEF`	`FP_TQDEF`
Unchanged	`TS_NOCHANGE`	`FP_NOCHANGE`

The constants for the time-sharing class are defined in the file <**`sys/tspriocntl.h`**>; the constants for the fixed-priority class are defined in the file <**`sys/fppriocntl.h`**>.

It is important to note that to change the time quantum for processes or LWPs assigned to the <u>fixed-priority</u> class, the following conditions must be met:

- The calling process must have the `P_RTIME` privilege
- The effective user ID of the calling process must match the effective user ID of the target process (the process for which the scheduling policy and priority are being set), or the calling process must have the `P_OWNER` privilege.

If the Enhanced Security Utilities are installed and running, the following additional conditions must be met:

> The Mandatory Access Control (MAC) level of the calling process must equal the MAC level of the target process, or the calling process must have the `P_MACWRITE` privilege.

The priocntllist System Call

```
#include <sys/types.h>
#include <sys/procset.h>
#include <sys/priocntl.h>
#include <sys/fppriocntl.h>
#include <sys/tspriocntl.h>

long priocntllist(lwpid_t *lwpidp, int idcnt, int cmd, void *arg);
```

The **`priocntllist`** system call provides the programming interface to scheduler classes and class-specific parameters for an arbitrary list of LWPs within the calling process. **`priocntllist`** has the same functions as the **`priocntl`** system call but offers a more general way of specifying the set of LWPs whose scheduling properties are to be changed. The input argument *lwpidp* points to an array in user memory of LWP IDs that identify the LWPs to which the system call applies, and *idcnt* is the number of elements in the array. *cmd* specifies the function to be performed, and *arg* is a pointer to a structure whose type depends on *cmd*.

The priocntlset System Call

```
#include <sys/types.h>
#include <sys/signal.h>
#include <sys/procset.h>
#include <sys/priocntl.h>
#include <sys/fppriocntl.h>
#include <sys/tspriocntl.h>

long priocntlset(procset_t *psp, int cmd, void *arg);
```

The **`priocntlset`** system call changes scheduler parameters of a set of processes or LWPs, just as **`priocntl(2)`** does. **`priocntlset`** has the same command set as **`priocntl`**; the *cmd* and *arg* input arguments are the same. But while **`priocntl`** applies to a set of processes or LWPs specified by a single *idtype/id* pair, **`priocntlset`** applies to a set of processes or LWPs that results from a logical combination of two *idtype/id* pairs. The input argument *psp* points to a `procset` structure that specifies the two *idtype/id* pairs and the logical operation to perform. This structure is defined in **`procset.h`**:

```
typedef struct procset {
        idop_t    p_op;          /* operator connecting */
                                 /* left and right sets */
   /* left set:  */
        idtype_t  p_lidtype;     /* left ID type */
        id_t      p_lid;         /* left ID */

   /* right set:  */
        idtype_t  p_ridtype;     /* right ID type */
        id_t      p_rid;         /* right ID */
} procset_t;
```

`p_lidtype` and `p_lid` specify the ID type and ID of one (**`left set`**) set of processes or LWPs; `p_ridtype` and `p_rid` specify the ID type and ID of a second (**`right set`**) set of processes or LWPs. `p_op` specifies the operation to perform on the two sets of processes or LWPs to get the set of processes or LWPs to operate on. The valid values for `p_op` and the processes or LWPs they specify are:

POP_DIFF	set difference—processes or LWPs in left set and not in right set
POP_AND	set intersection—processes or LWPs in both left and right sets
POP_OR	set union—processes or LWPs in either left or right sets or both
POP_XOR	set exclusive-or—processes or LWPs in left or right set but not in both

The following macro, also defined in **`procset.h`**, offers a convenient way to initialize a `procset` structure:

```
#define setprocset(psp, op, ltype, lid, rtype, rid) \
        ((psp)->p_op      = (op), \
        (psp)->p_lidtype  = (ltype), \
        (psp)->p_lid      = (lid), \
        (psp)->p_ridtype  = (rtype), \
        (psp)->p_rid      = (rid))
```

Here is a situation where **`priocntlset`** can be useful: an application has both fixed-priority and time-sharing processes that run under a single user ID. If the application wants to change the priority of only its fixed-priority processes without changing the time-sharing processes to fixed-priority processes, it can do so as shown in Screen 5-8 (This example uses the function `schedinfo,` which is defined in Screen 5-3.)

`priocntl` offers a simple scheduler interface that is adequate for many applications; applications that need a more powerful way to specify sets of processes or LWPs can use **`priocntlset.`**

```
/*
 *  Change fixed priorities of this uid
 *  to highest fixed priority minus 1.
 */

main (argc, argv)
        int argc;
        char *argv[];
{
        procset_t       procset;
        pcparms_t       pcparms;
        struct fpparms  *fpparmsp;
        id_t            fpclassID;
        id_t            schedinfo();
        short           maxfppri;

   /* left set: select processes with same uid as this process */
        procset.p_lidtype = P_UID;
        procset.p_lid = getuid();

   /* get info on fixed priority class */
        if ((fpclassID = schedinfo ("FP", &maxfppri)) == -1) {
                perror ("schedinfo failed");
                exit (1);
        }

   /* right set: select fixed priority processes */
        procset.p_ridtype = P_CID;
        procset.p_rid = fpclassID;

   /* select only my FP processes */
        procset.p_op = POP_AND;

   /* specify new scheduler parameters */
        pcparms.pc_cid = fpclassID;
        fpparmsp = (struct fpparms *) pcparms.pc_clparms;
         fpparmsp->fp_pri = maxfppri - 1;
        fpparmsp->fp_tqnsecs = FP_NOCHANGE;
        if (priocntlset (&procset, PC_SETPARMS, &pcparms) == -1) {
                perror ("priocntlset failed");
                exit (2);
        }
}
```

Screen 5-8. Changing the Scheduler Class for Selected Processes

Scheduler Commands

The sections that follow explain the procedures for using the following commands for scheduling purposes:

`priocntl` Displays or sets the scheduler parameters of processes or LWPs

This command provides easy access to the major services provided by the **`priocntl(2)`** system call.

`run` Executes a program in the specified environment

Options allow you to specify a POSIX scheduling policy and priority. You can also specify the time quantum for a program scheduled under the **SCHED_RR** policy.

`rerun`	Alters the execution environment of a running process or LWP Options allow you to specify a POSIX scheduling policy and priority. You can also specify the time quantum for a process or LWP scheduled under the **SCHED_RR** policy.
`setrun(1)`	Combines the functions of several system functions into one interface that: - Lets the application define the scheduling environment in which a specified command executes - Can either bind or exclusively bind a process to a processor - Can interpret the run-time environment and the command to execute from either a file or the command line.

In addition to these commands, you can also use the **`ps -cel`** command to obtain global priorities for all active processes and LWPs.

The priocntl Command

The **`priocntl(1)`** command sets or retrieves scheduler parameters for processes and LWPs. It reports the class-specific priorities that users and programmers use.

The steps used to set priorities with **`priocntl(1)`** are similar to those used for the **`priocntl(2)`**, **`priocntllist(2)`**, and **`priocntlset(2)`** system calls (see p. 5-16 for a list of these steps). The way in which you specify the target processes and LWPs by using an ID type and ID is also similar (see p. 5-16 for an explanation of ID types and IDs and a list of the valid ID types that you can specify).

The **`priocntl`** command comes in four forms:

`priocntl -l`	display configuration information
`priocntl -d`	display the scheduler parameters of processes and LWPs
`priocntl -s`	set the scheduler parameters of processes and LWPs
`priocntl -e`	execute a command with the specified scheduler parameters

Screen 5-9 shows the output of the **`-l`** option for the default configuration.

```
$ priocntl -l
CONFIGURED CLASSES
==================

SYS (System Class)

AD (ADA Class)
        Maximum Configured AD Priority: 160

TS (Time Sharing)
        Configured TS User Priority Range: -20 through 20

FP (Fixed Priority)
        Maximum Configured FP Priority: 59

FC (Fixed Class)
        Configured FC User Priority Range: -20 through 20
```

Screen 5-9. Output from the priocntl -l Command

Note

PowerMAX OS reserves the AD scheduling class for the internal use of the ADA runtime environment; applications should not use it.

The **`-d`** option displays the scheduler parameters of a process or LWP or a set of processes or LWPs. The syntax for this option is

`priocntl -d -i` *idtype idlist*

The *idtype* argument tells what kind of IDs are in *idlist*. *idlist* is a list of IDs separated by white space. Table 5-5 shows the valid values for *idtype* and their corresponding ID types in *idlist*:

Table 5-5. Idtype and idlist Values

idtype	*idlist*
`lwpid`	LWP IDs
`pid`	Process IDs
`ppid`	Parent process IDs
`pgid`	Process group IDs
`sid`	Session IDs
`class`	Class names (`TS`, `FC`, or `FP`)
`uid`	Effective user IDs
`gid`	Effective group IDs
`all`	All processes and LWPs

Screen 5-10 shows some examples of the **`-d`** option of **`priocntl`**:

```
$ # display info on all processes and LWPs
$ priocntl -d -i all
     .
     .
     .
$ # display info on all time-sharing processes and LWPs:
$ priocntl -d -i class TS
     .
     .
     .
$ # display info on all processes and LWPs with user ID 103 or 6626
$ priocntl -d -i uid 103 6626
     .
     .
     .
```

Screen 5-10. Output from the priocntl -d Command

The **`-s`** option sets scheduler parameters for a process or LWP or a set of processes or LWPs. The syntax for this option is

`priocntl -s -c` *class class_option(s)* **`-i`** *idtype idlist*

The *idtype* and *idlist* arguments are the same as for the **`-d`** option described previously.

The value of *class* is **`TS`** for time-sharing, **`FC`** for fixed class, or **`FP`** for fixed priority. You must have the `P_RTIME` privilege to change a process's or LWP's class to fixed priority. You must have the `P_TSHAR` privilege to raise a time-sharing or fixed-class user priority limit. Class options are class-specific as shown in Table 5-6:

Table 5-6. Class Specific Options for priocntl

class	**`-c`** *class*	*options*	Meaning
fixed priority	`FP`	**`-p`** *pri*	Priority
		`-t` *tslc*	Time slice
		`-r` *res*	Resolution
time-sharing	`TS`	**`-p`** *upri*	User priority
		`-m` *uprilim*	User priority limit
fixed class	`FC`	**`-p`** *upri*	User priority
		`-m` *uprilim*	User priority limit

For a fixed-priority process or LWP, you may assign a priority and a time slice.

- The priority is a number from `0` to the fixed-priority maximum as reported by **`priocntl -l;`** the default maximum is 59.

- To specify the time slice, you use the **`-t`** *tslc* option to specify a number of clock intervals; in addition, you have the option of using the **`-r`** *res* option to specify the resolution of the interval. Resolution is specified in intervals per second. The time slice, therefore, is *tslc/res* seconds. To specify a time slice of one-tenth of a second, for example, you would specify a *tslc* of **`1`** and a *res* of **`10`**. If you specify a time slice without specifying a resolution, millisecond resolution (a *res* of 1000) is assumed.

If you change a time-sharing or fixed-class process or LWP into a fixed-priority process or LWP, it gets a default priority and time slice if you do not specify one. If you want to change only the priority of a fixed-priority process or LWP and leave its time slice unchanged, omit the **`-t`** option. If you want to change only the time slice of a fixed-priority process or LWP and leave its priority unchanged, omit the **`-p`** option.

For a time-sharing or fixed-class process or LWP, you may assign a user priority and a user priority limit. The user priority is the user-controlled component of a time-sharing or fixed-class priority. The user priority limit is the maximum user priority a process or LWP may set for itself without being a privileged user. These values are described in detail in the section that explains use of the **`priocntl(2)`** system call (p. 5-17).

Both the user priority and the user priority limit must be within the user priority range reported by the **`priocntl -l`** command; this range is also reported by the PC_GETCID and PC_GETCLINFO commands to the **`priocntl(2)`** system call.

There is no limit for the number of times a process or LWP may lower and raise its user priority, as long as the value is below its user priority limit. As a courtesy to other users, lower your user priority for big chunks of low-priority work. However, remember that if you lower your user priority limit, you must have the P_TSHAR privilege to raise it. A typical use of the user priority limit is to reduce permanently the priority of child processes or LWPs or another set of low-priority processes or LWPs.

The user priority can never be greater than the user priority limit. If you set the user priority limit below the user priority, the user priority is lowered to the new user priority limit. If you attempt to set the user priority above the user priority limit, the user priority is set to the user priority limit.

Screen 5-11 shows examples of the **`-s`** option of **`priocntl`**:

```
# # make process with ID 24668 a fixed priority process with default
parameters:
# priocntl -s -c FP -i pid 24668

# # make 3608 FP with priority 55 and a one-fifth second time slice:
# priocntl -s -c FP -p 55 -t 1 -r 5 -i pid 3608

# # change all processes or LWPs into time-sharing processes or LWPs:
# priocntl -s -c TS -i all

# # for uid 1122, reduce TS user priority and user priority limit to -10:
# priocntl -s -c TS -p -10 -m -10 -i uid 1122
```

Screen 5-11. Output from the priocntl -s Command

The **-e** option sets scheduler parameters for a specified command and executes the command. The syntax for this option is

priocntl -e -c *class class_option(s) command* [*command arguments*]

The class and class options are the same as for the **-s** option described previously. Screen 5-12 shows examples of the **-e** option of **priocntl**:

```
# # start a fixed priority shell with default fixed priority:
# priocntl -e -c FP /bin/sh

# # run make with a time-sharing  user priority of -10:
# priocntl -e -c TS -p -10 make bigprog
```

Screen 5-12. Output from the priocntl -e Command

The **priocntl** command includes the function of **nice**, which continues to work as in previous releases. **nice** works only on time-sharing processes and LWPs and uses higher numbers to assign lower priorities. The example shown in Screen 5-12 is equivalent to using **nice** to set an increment of 10:

```
nice -10 make bigprog
```

The run and rerun Commands

The **run(1)** command allows you to run a program under a specified POSIX scheduling policy and at a specified priority (see p.5-5 for a complete explanation of POSIX scheduling policies). It also allows you to set the time quantum for a program scheduled under the **SCHED_RR** policy. The **rerun(1)** command allows you to change the scheduling policy and priority of one or more running processes or LWPs. It also allows you to change the time quantum for a process or LWP scheduled under the **SCHED_RR** policy.

To set a program's scheduling policy and priority, invoke the **run** command from the shell, and specify the **–s** *scheduling_policy* and **–P** *priority* options. The value of *scheduling_policy* must be one of the keywords presented in the following table:

Table 5-7. Acceptable Keywords for the -s Option

Scheduling Policy	Keywords
First-in-first-out (FIFO) policy	**- SCHED_FIFO** **- fifo**
Round-robin (RR) policy	**- SCHED_RR** **- rr**
Time-sharing policy	**- SCHED_OTHER** **- other**

The value of *priority* is (1) an integer that lies within the range of scheduling priorities defined for the scheduler class associated with the specified scheduling policy or (2) the keyword **`max`**. You can obtain the allowable range of priorities by invoking the **`run`** command from the shell and not specifying any options or arguments or by invoking the **`sched_get_priority_min(3C)`** and **`sched_get_priority_max(3C)`** library routines (see p. 5-14 for explanations of these routines). The keyword **`max`** specifies the highest (most favorable) priority defined for the scheduler class associated with the scheduling policy.

If you specify the **`-s`** *scheduling_policy* option without also specifying the **`-P`** *priority* option, the program's priority is set to zero, which is the default initial user priority.

To set the time quantum for a program being scheduled under the **SCHED_RR** scheduling policy, also specify the **`-q`** *quantum* option. The *quantum* specifies the time that an LWP may use the CPU before the scheduler preempts it to allow another LWP of the same priority to use the CPU. The time is specified in clock ticks, where each clock tick is 1/**HZ** of a second (**HZ** is a constant that is defined in <**`sys/param.h`**>).

It is important to note that to (1) change a process's scheduling policy to the **SCHED_FIFO** or the **SCHED_RR** policy, (2) change the priority of a process scheduled under the **SCHED_FIFO** or the **SCHED_RR** policy, or (3) change the time quantum for a process scheduled under the **SCHED_RR** policy, the following conditions must be met:

- The calling process must have the `P_RTIME` privilege.
- The effective user ID of the calling process must match the effective user ID of the target process (the process for which the scheduling policy and priority are being set), or the calling process must have the `P_OWNER` privilege.

If the Enhanced Security Utilities are installed and running, the following additional conditions must be met:

The Mandatory Access Control (MAC) level of the calling process must equal the MAC level of the target process, or the calling process must have the `P_MACWRITE` privilege.

To raise the priority of a process scheduled under the **SCHED_OTHER** policy above a per-process or LWP limit, the following conditions must be met:

- The calling process must have the `P_TSHAR` privilege.
- The effective user ID of the calling process must match the effective user ID of the target process (the process for which the scheduling priority is being set), or the calling process must have the `P_OWNER` privilege.

If the Enhanced Security Utilities are installed and running, the following additional conditions must be met:

The Mandatory Access Control (MAC) level of the calling process must equal the MAC level of the target process, or the calling process must have the `P_MACWRITE` privilege.

To change the scheduling policy and priority of one or more running processes or LWPs, invoke the **`rerun`** command from the shell, and specify the **–s** *scheduling_policy* and **–P** *priority* options. The value of *scheduling_policy* must be one of the keywords presented in Table 5-7.

Note that the specified priority of the process(es) or LWP(s) for which the scheduling policy is being changed must lie within the range of priorities defined for the scheduler class associated with *scheduling_policy*. If it does not, an error message will be displayed; the process's scheduling policy and priority will not be changed.

Note that when you change a process's scheduling policy, you also change its time quantum to the default time quantum that is defined for the scheduler class associated with the new policy and the priority. To change the time quantum for a running process or LWP scheduled under the **SCHED_RR** scheduling policy, specify the **`-q`** *quantum* option.

The privileges required for changing a running process's scheduling policy, priority, and quantum are the same as those presented previously for use of the **`run`** command.

For additional information on use of the **`run(1)`** and **`rerun(1)`** commands for scheduling purposes, refer to the corresponding system manual pages.

The setrun(1) command

You can use the System V scheduling classes without embedding the routines in your application by using **`setrun(1)`**. **`setrun(1)`** is a system command that sets the scheduling and execution environment for a process. **`setrun(1)`** combines the functions of several system functions into one interface that:

- Lets the application define the scheduling environment in which a specified command executes
- Can either bind or exclusively bind a process to a particular processor
- Can interpret the run-time environment and the command to execute from either a file or the command line.
- Can lock the application in memory.

See the **`setrun(1)`** man page for more information..

The syntax is:

`setrun -e` `[`**`-c`** *class*`]` `[`*class-specific options*`]` `[`*command*`]` `[`*argument*`]`

`setrun -f` `[`*filename*`]` `[`*command*`]` `[`*argument*`]`

`setrun -d` `[`**`-i`** `[`*idtype*`]]`

where:

`-e` Indicates that the specified *command* (and its arguments) shall execute in the environment specified.

`-f` Specifies that the file *filename* contains the execution environment. The command to be executed can be specified either on the command line or in the file.

`-d`	Displays information about the specified processes. This information is also available through **`ps(1)`**, **`pbind(1M)`**, **`pexbind(1M)`**, and **`psrinfo(1M)`**.

Scheduling classes

The **`-c`** option specifies the System V scheduling class:

`TS`	time-sharing
`FC`	fixed class
`FP`	fixed priority

Each class has class-specific options. See the **`setrun(1)`** man page for information.

Display options

The **`-i`** option specifies the type of process the **`-d`** option reports on:

`pid`	process
`lwp`	lightweight process

Examples

The following lines in a file instruct **`setrun(1)`** to run the program **`/user/tests/test`** as a fixed priority process with a priority of 56 on processor 2:

```
prog = /user/tests/test
class = FP
priority = 56
bind = 2
```

Additional keywords are **`prilim`**, **`quantum`**, and **`xbind`**.

The following command reads in all arguments from *filename* and runs the command listed in the file:

`setrun -f` *filename*

The following command executes *command* with arguments *arguments* as a time-sharing command with a priority of -8 and a priority limit of 0:

`setrun -e -c TS -m 0 -p -8` *command arguments*

The **`setrun(1)`** man page describes all of the options listed in this example.

Scheduler Interaction with Other Functions

This section describes scheduler interaction with kernel processes and such functions as **fork** and **exec**, **nice**, and **init**.

Kernel Processes

The kernel assigns its daemon and housekeeping processes to the system scheduler class. Users may not add processes or LWPs to this class, remove processes or LWPs from this class, or change the priorities of these processes or LWPs. The command **ps -el** lists the scheduler class of all processes or LWPs. Processes in the system class are identified by a SYS entry in the CLS column.

If the workload on a machine contains fixed-priority processes or LWPs that use too much CPU time, they can lock out system processes; doing so can lead to trouble if these fixed-priority processes depend on the services of a system daemon running on the same processor.

fork, exec

Scheduler class, priority, and other scheduler parameters are inherited across the **fork(2), _lwp_create(2)**, and **exec(2)** system calls.

nice

The **nice(1)** command and the **nice(2)** system call work as in previous versions of the UNIX operating system. They allow you to change the priority of only a time-sharing process or LWP. You still use lower numeric values to assign higher time-sharing priorities with these functions.

To change the scheduler class of a process or LWP or to specify a fixed priority, you must use one of the **priocntl** functions or the POSIX scheduling functions. Use higher numeric values to assign higher priorities with these functions.

init

The **init** process (process ID 1) may be assigned to any class configured in the system. However, **init** should be assigned to the time-sharing class unless there are compelling reasons to do otherwise. You can assign the **init** process to another class by using the **config(1M)** utility to change the value of the system tunable parameter INITCLASS. Refer to the corresponding system manual page and the “Configuring and Building the Kernel” chapter of *System Administration Volume 2* for an explanation of the procedures for doing so.

Scheduler Performance

Because the scheduler determines when and for how long LWPs run, it has an overriding importance in the performance and perceived performance of a system.

By default, all processes and LWPs are time-sharing processes or LWPs. A process or LWP changes class only as a result of one of the **`priocntl`** functions or the POSIX scheduling functions.

In the default configuration, all fixed-priority process priorities are above any time-sharing process priority. This implies that as long as any fixed-priority process or LWP is runnable, no time-sharing process or LWP or system process ever runs. So if a fixed-priority application is not written carefully, it can completely lock out users and essential kernel housekeeping.

Besides controlling process and LWP class and priorities, a fixed-priority application must also control several other factors that influence its performance. The most important factors in performance are CPU power, amount of primary memory, and I/O throughput. These factors interact in complex ways. For more information, see the chapter on performance management in the *System Administration Volume 2* manual. In particular, the **`sar(1)`** command has options for reporting on all the factors discussed in this section.

LWP State Transition

Applications that have strict fixed-priority constraints may need to prevent processes and LWPs from being swapped or paged out to secondary memory. Figure 5-3 presents a simplified overview of UNIX system LWP states and the transitions between states:

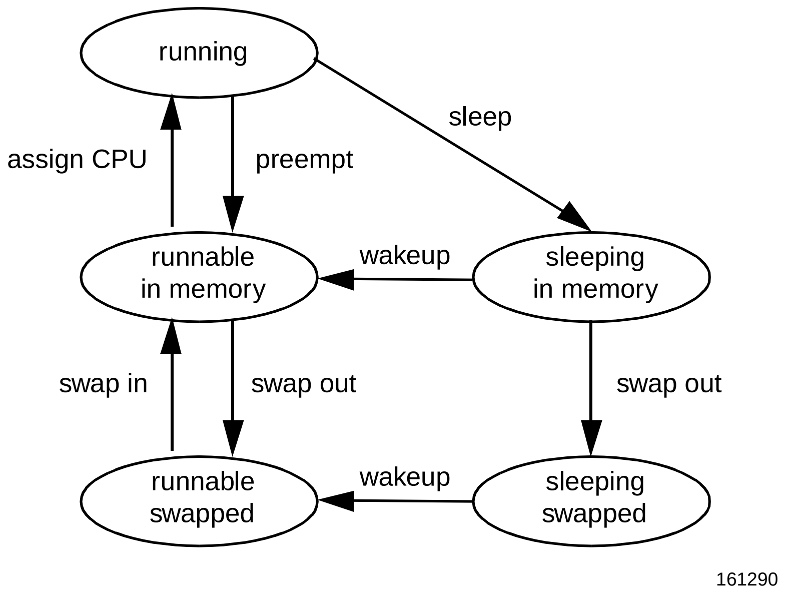


Figure 5-3. LWP State Transition Diagram

An active LWP is normally in one of the five states in the diagram. The arrows show how it changes states.

- An LWP is running if it is assigned to a CPU. An LWP is preempted—that is, removed from the running state—by the scheduler if an LWP with a higher priority becomes runnable. An LWP is also preempted if it consumes its entire time slice and an LWP of equal priority is runnable.
- An LWP is runnable in memory if it is in primary memory and ready to run but is not assigned to a CPU.
- An LWP is sleeping in memory if it is in primary memory but is waiting for a specific event before it can continue execution; for example, an LWP is sleeping if it is waiting for an I/O operation to complete, for a locked resource to be unlocked, or for a timer to expire. When the event occurs, the process is sent a wakeup; if the reason for its sleep is gone, the LWP becomes runnable.
- An LWP is runnable and swapped if it is not waiting for a specific event but has had its whole address space written to secondary memory to make room in primary memory for other LWPs.

- An LWP is sleeping and swapped if it is both waiting for a specific event and has had its whole address space written to secondary memory to make room in primary memory for other processes or LWPs.

If a machine does not have enough primary memory to hold all its active processes and LWPs, it must page or swap some address space to secondary memory:

- When the system is short of primary memory, it writes individual pages of some processes and LWPs to secondary memory but leaves those processes and LWPs runnable. When an LWP runs, if it accesses those pages, it must sleep while the pages are read back into primary memory.
- When the system gets into a more serious shortage of primary memory, it writes all the pages of some processes and LWPs to secondary memory and marks those processes and LWPs as swapped. Such processes and LWPs get back into a schedulable state only by being chosen by the system scheduler daemon process, then read back into memory.

Both paging and swapping, and especially swapping, introduce delay when a process or LWP is ready to run again. For processes and LWPs that have strict timing requirements, this delay can be unacceptable. To avoid swapping delays, fixed-priority processes and LWPs are never swapped, though parts of them may be paged. An application can prevent paging and swapping by locking its text and data into primary memory. For more information see the **`memcntl(2)`** system manual page. Of course, how much can be locked is limited by how much memory is configured. Also, locking too much can cause intolerable delays to processes and LWPs that do not have their text and data locked into memory. Trade offs between performance of fixed-priority processes and LWPs and performance of other processes and LWPs depend on local needs. On some systems, process locking may be required to guarantee the necessary fixed-priority response.

6
Memory Management

6 Memory Management

Overview of Primary Memory

This section provides an overview of primary memory from the hardware and software perspectives. Primary memory on Model 6800 systems consists of different types of memory pools that vary in their proximity to the system's processors. Because some memory pools are closer to some processors than others, the system architecture has non-uniform primary memory access times. Such architectures are called NUMA architectures. This section shows how you can influence the operating system's page placement decisions by selecting policies that govern the type of memory in which different portions of a process's address space may be located. It describes some of the performance benefits associated with the use of local memory.

Hardware Features

Primary memory includes *global memory* and *local memory*. Global memory is a pool of memory that is physically located on a memory board. The global memory pool is shared by all of the processors in the system. Each processor has access to global memory via the system bus. Local memory is a pool of memory that is located on a processor board. A local memory pool is shared by all of the processors that reside on that board. Each processor has a data path to its local memory pool that does not require use of the system bus and does not incur any system bus arbitration delay. Data from global or local memory can be stored in the processor's data or instruction cache. Figure 6-1 shows the organization of primary memory.

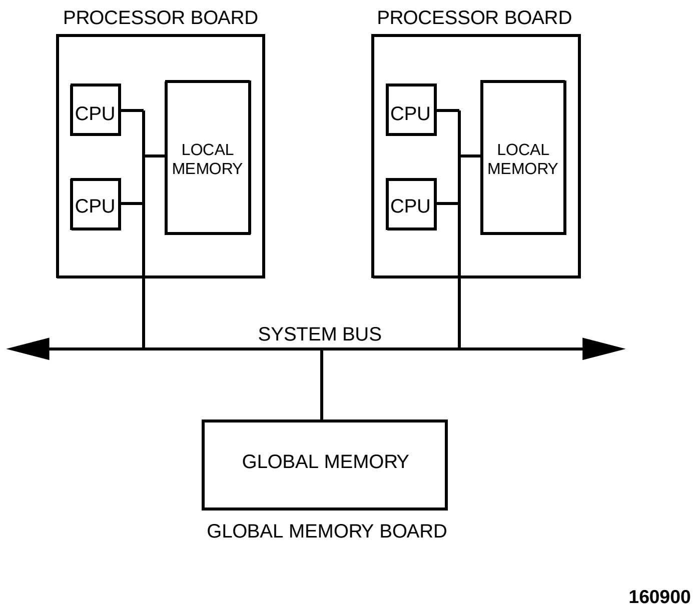


Figure 6-1. Logical Organization of Primary Memory

Local memory is an architectural feature that is designed to improve the performance of multiprocessor systems. The primary advantages of local memory include the following:

1. Reduced system bus traffic and contention

   Multiprocessor systems can easily utilize the entire bandwidth of the system bus, leading to a saturation condition. When this occurs, the memory requests of CPUs and I/O controllers must be queued and wait until the bus is free. Because local memory accesses do not use the system bus, overall bus traffic and contention are reduced. This contention has been one of the factors that has traditionally limited the number of processors that can be effectively utilized in a tightly coupled multiprocessing system.

2. Faster memory access times when a cache miss occurs

   Faster memory access times are due in part to the absence of the need to access the system bus, but they can also be due to different memory chip technology being used for local memory.

3. More predictable memory access times

   That memory access times are more predictable is an outgrowth of the lack of system bus contention on local memory accesses.

A processor can access the local memory on another processor board. Such accesses, which are called *foreign* or *remote* memory accesses, are slow. Whereas data from global or local memory can be stored in the processor's data cache, data from remote memory cannot. Hence, remote memory accesses always bypass the caches and contend for the system bus. Executing out of remote memory locations should be avoided. In addition, use of atomic synchronizing instructions (**`lwarx`** or **`stwcx.`**, for example) on remote memory locations should be avoided because the indivisibility of their read-modify-write cycle is not guaranteed (for information on **`lwarx`** and **`stwcx.`**, refer to the *PowerPC Microprocessor Family: the Programming Environments*). For these reasons, the kernel restricts the use of remote memory. (On Series 6000 and Power MAXION™[1] systems, use of atomic synchronizing instructions should also be avoided on cache-inhibited locations.)

Software Features

The operating system provides support for obtaining information about the memory resources of your system. The **`run(1)`** command and the **`mpadvise(3C)`** library routine allow you to determine which CPUs on your system have local memory. The **`mpstat(1)`** command provides a graphical display of information about all of the system's memory pools. The **`syscx(2)`** system call allows you to obtain information about a particular memory pool. The **`hwstat(1M)`** command provides a wide range of information about your system's hardware configuration; included is information about each memory pool. For additional details on the use of these interfaces and the types of information they provide, refer to the corresponding system manual pages.

"Memory Pools" describes memory pools from the software perspective, and "NUMA Policies" explains the NUMA policies that govern placement of portions of a process's address space in those pools. "Guidelines for Determining the Appropriate Default NUMA Policy" provides guidelines for determining the appropriate NUMA policy for different parts of an applications's address space. "Memory Pools and Process Memory Locking" explains how process memory locking is handled.

Memory Pools

By default, the operating system itself resides entirely in global memory, leaving all local memory for use by applications. Kernel text, however, can be replicated in selected local memory pools by setting the value of the corresponding `KTEXTLOCAL`*n* system tunable parameter to `1` (*n* denotes a number ranging from **`1`** to **`4`**, where **`1`** represents the first local memory pool, **2** the second, and so on, in order according to processor board slot number). You can examine and modify the values of system tunable parameters by using the **`config(1M)`** utility. For an explanation of the procedures for using this utility, refer to the "Configuring and Building the Kernel" chapter of *System Administration Volume 2*. After changing a tunable parameter, you must rebuild the kernel and then reboot your system. Note that when kernel text is replicated in local memory, the original copy of kernel text remains in global memory.

A local memory pool is shared by all of the processors that reside on the processor board on which the pool is located. Processes using local memory can easily migrate between the processors on the same board; however, a process using local memory cannot migrate

1. Power MAXION is a trademark of Concurrent Computer Corporation.

to another processor board without also migrating its local memory pages (in order to avoid costly remote memory accesses). Because cross-board migrations are expensive, they are never initiated by the kernel's dynamic load balancer. Therefore, when processes are using local memory, it is possible for a processor on one board to idle while processors on other boards are quite busy. For this and other reasons, the operating system allows applications to control their use of local memory.

When an application begins execution, there is one LWP in the process. If the process is using local memory, the dynamic load balancer confines the LWP to a single processor board. When additional LWPs are created by invoking **`_lwp_create(2)`**, they are confined to the same processor board as their creator. Thus, in a process that is using local memory, the default behavior is for all LWPs to run on the same processor board. You can override the default by using the **`_lwp_migrate(2)`** system call to migrate a particular LWP to another processor board. If there are multiple LWPs in the process, **`_lwp_migrate`** distributes the address space by creating a new virtual-to-physical address translation table for the LWP to use on the new processor board. Using multiple translation tables makes it possible for LWPs running on different processor boards to fetch instructions from their respective local memory pools. A process that is using local memory and that has multiple LWPs running on different processor boards is called a *distributed process*.

The operating system manages primary memory as a file cache. To keep the data in this cache coherent, the operating system enforces the rule that writable pages cannot exist in more than one memory pool at a time. Read-only pages, however, can be replicated in different memory pools in order to provide the smallest possible access times to the largest possible number of processes. In order to enforce the cache coherence rule, if a read-only page later becomes writable, replicas of the page are destroyed until only a single copy remains, Also, in order to avoid remote memory accesses, if a writable page in a local memory pool is referenced from a remote processor, the page may migrate to the global memory pool.

NUMA Policies

Beyond the cache coherence rule mentioned in the preceding section, there is some flexibility in the outcome of the kernel's page placement decisions. You can influence the kernel's page placement decisions by selecting NUMA policies for different portions of a process's virtual address space. It is important to note that NUMA policies are attributes of processes or mappings; they are not attributes of LWPs. All of the LWPs in a process use the same NUMA policy for a given page. Four policies are available: global, soft-local, hard-local, and any-pool. They are described as follows:

Global policy	Places the process's pages in global memory
Soft-local policy	Places the process's pages in local memory if possible and global memory if not The soft-local policy is available in two forms: floating and anchored. The terms *floating* and *anchored* are used to differentiate the times at which the kernel's page-placement decisions are made. With the floating soft-local policy, the decision is made when the page is referenced in memory. With the anchored soft-local policy, the decision is made when a mapping is created. (see "Virtual Memory, Address Spaces, and Mapping," p. 6-13).

These terms are described in further detail following presentation of the four policies.

Hard-local policy

Attempts to place the process's pages in local memory and if the local memory pool is full, waits for the pages to be available

When the local memory pool is full, the hard-local policy causes a process to block until the pages become available. When there is one LWP in the process and the hard-local policy is used for process-private stack and heap pages, it can guarantee that those pages will be placed in a local memory pool. However, the operating system's enforcement of the cache coherence rule and avoidance of remote memory accesses cannot guarantee that file pages will be placed in a local memory pool. File pages will not be placed in a local memory pool, for example, if they are writable and are being accessed from different processor boards. They will be placed in the global memory pool instead.

Use of the hard-local NUMA policy can cause a local memory pool to thrash. For this reason, you are advised to select this policy only when the application mix is well-known and the use of local memory is strictly controlled.

The hard-local policy is available in two forms: floating and anchored. The terms *floating* and *anchored* are used to differentiate the times at which the kernel's page-placement decisions are made. With the floating hard-local policy, the decision is made when the page is referenced in memory. With the anchored hard-local policy, the decision is made when a mapping is created (see "Virtual Memory, Address Spaces, and Mapping," p. 6-13). These terms are described in greater detail following presentation of the four policies.

Any-pool policy

Makes it possible for the process's pages to be placed in any of the system's memory pools

This policy is similar to the soft-local NUMA policy in that it prefers to use local rather than global page frames, but it is different from the other policies in that it considers remote page frames acceptable as a last resort. With the other policies, a remote page will migrate to or be replicated in a pool that is more acceptable. With the any-pool policy, the remote frame will be used directly.

Note that the any-pool NUMA policy is available only through use of the **`memcntl(2)`** and **`mmap(2)`** system calls. You are advised to use it sparingly.

The terms floating and anchored are further differentiated by considering the perspective from which the page-placement decision is made. With a floating local policy, the perspective is that of the CPU that is fetching the page. That perspective may be different for different LWPs in the same process because the LWPs may be executing on different processor boards. A floating local policy is useful for read-only pages (text pages, for example) in distributed processes because it causes the pages to be replicated. It is also useful for other processes because it causes the pages to follow the process when it migrates to another processor board.

With an anchored local policy, the perspective is that of the CPU that has created the mapping. That perspective is the same for all of the LWPs in a process, regardless of where they are executing. An anchored local policy is useful for writable pages that have a lopsided reference pattern (stack pages, for example). It causes the kernel to place the pages in a particular local memory pool—a pool that is close to the CPUs that generate the majority of the accesses. The operating system permits remote memory accesses from other processor boards.

Every process has a set of default NUMA policies for different parts of its address space. A process inherits its defaults from its parent during a **`fork(2)`** system call. Because **`init`** is the ancestor of all processes, its default NUMA policies are, in effect, system-wide defaults. A process can change its default NUMA policies by using the **`memdefaults(2)`** system call or the **`run(1)`** and **`rerun(1)`** commands. Procedures for using **`memdefaults`** are explained in “Using the memdefaults System Call.” Procedures for using **`run`** and **`rerun`** are explained in “Using the run and rerun Commands.” Default NUMA policies affecting all processes running on a system may be set by the system administrator by altering the NUMA policies of the **`init`** process as set in the **`rerun`** script in the **`/etc/init.d`** directory. Note that anchored local policies are not permitted as defaults.

A process can select a different default NUMA policy for its *text*, *private data*, *shared data*, and *U-block*. These categories are defined as follows:

Text	Refers to pages in private (**`mmap(2)`** **MAP_PRIVATE**) mappings that belong to a file in a file system (the traditional text segment, for example)
Private data	Refers to pages in private mappings that do not belong to a file in a file system (the traditional stack and data segments, for example) Note that the first time that a process writes to a page in a private, writable mapping to a file, the page will move from the text category to the private data category.
Shared data	Refers to shared (**`mmap(2)`** **MAP_SHARED**) mappings other than System V shared memory segments
U-block	Refers to a kernel data structure that is associated with each LWP; it contains the kernel-mode stack used during system calls, the register save area used during context switches, and information about the LWP.

A process’s default NUMA policies can be overridden by use of the **`mmap(2)`** and **`memcntl(2)`** system calls. **`Mmap`** allows a process to select any of the previously enumerated NUMA policies or no policy at all. If a policy is selected, it applies only to the mapping established by **`mmap`**. If no policy is selected, the defaults are used. **`Memcntl`** allows a process to change the NUMA policy associated with a range of addresses in its address space. Procedures for using **`mmap`** are explained in “Creating and Using Mappings.” Information necessary for using **`memcntl`** is provided in the corresponding system manual page.

The NUMA policies for System V shared memory segments are established when a shared memory segment is created. System V shared memory segments are created by using the **`shmget(2)`** system call or the **`shmdefine(1)`** or **`shmconfig(1M)`** utilities. The global NUMA policy is selected for a shared memory segment by default. If one of the local policies is selected, the local memory pool used for the shared memory segment will be that which belongs to the CPU of the calling process; the segment will not be available to processes executing on a different processor board. Procedures for using the OS shared memory facilities are explained in Chapter 12, "Interprocess Communication."

Guidelines for Determining the Appropriate Default NUMA Policy

Selection of NUMA policies for different parts of an application's address space may require some experimentation. The following analysis may help you to determine the policies that are appropriate for your application:

Text pages are not writable, so binding text pages to local memory can cause the pages to be replicated when the same program is run on multiple processor boards. Page replication is a space versus time trade off that may or may not be beneficial for a given application. Furthermore, text references are cached in the instruction cache and are not snooped (which means faster access to memory in the event of a cache miss). So if the hit rate on the instruction cache is high and the local memory pools are small, it may not be worthwhile to bind text pages to local memory. Writable pages cannot be replicated, so binding them to local memory can cause some page migration. In general though, placing private writable pages in local memory gives the best performance improvement.

A program that makes very frequent system calls may benefit from locating the U-block in local memory; the benefit comes from more predictable and possibly reduced context switch and system call times. Locating the program's private data in local memory will usually also lead to some reduction in system time because arguments to system calls and buffered I/O transfers must be copied between kernel space and the user program's data space.

Processes attempting to attach a shared memory segment that is bound to a local memory pool should be executing on a processor that has access to that pool. The operating system discourages a process from attaching to a shared memory segment that is bound to the local memory of another processor board. If such an attempt is made, the call to **`shmat(2)`** to attach the segment will fail and return an `EACCES` error. If a shared memory segment is to be used by processes executing on different processor boards, the global NUMA policy should be selected for that segment.

It is important to note that a process may attach a shared memory segment that is located in remote memory by invoking the **`shmat`** system call with the **SHM_FLMEM** bit set in the *shmflg* argument. Setting the **SHM_FLMEM** bit is not generally recommended. The reasons are as follows:

- All caching of the specified shared memory segment's pages will be inhibited on the CPU or CPUs that are making the remote memory references.
- Accesses to remote memory are much slower than references to global or local memory.
- Atomic synchronizing instructions are not supported when executed on a semaphore that resides in remote memory.

Memory Pools and Process Memory Locking

When an application locks pages in memory using the facilities described in "Memory Page Locking" and "Address Space Locking," the operating system attempts to preprocess all possible faults on those pages in order to guarantee that no faults will occur later when the application accesses those pages. It is not possible, however, for the operating system to guarantee that no faults will occur on some types of pages locked in a local memory pool. If a writable page locked in a local memory pool is referenced from a remote CPU with a conflicting NUMA policy (that is, one that specifies a different memory pool), the page will migrate to global memory, and the LWPs accessing the page will fault. Furthermore, if a read-only page locked in a local memory pool later becomes writable, replicas of the page are destroyed until only a single copy remains; the LWPs accessing those replicas will fault. Note, however, that this problem does not exist for process-private pages, pages locked in global memory, or shared memory pages.

When a process's NUMA policy or CPU assignment changes, some of the locked pages in the process's address space may need to move to a new memory pool. Before such an operation begins, it is very difficult to guarantee that all necessary page migrations can be completed successfully. If a page migration fails, the operation must be aborted. It is also not possible to guarantee that after the operation has been aborted, the process's original state can be restored. Aborting the operation can leave pages that should be locked unlocked. The OS provides features (for example, user-level interrupts and control over rescheduling) for real-time applications that require firm guarantees about the properties of locked pages. To ensure adherence to the guarantees, the following rule is enforced.

> A process or LWP migration or a NUMA policy change that requires locked pages to move between memory pools will fail (that is, will not be performed) without affecting the locked state of the pages in the address space.

When developing an application, you are advised to migrate LWPs and set their NUMA policies before locking their pages. Failures caused by locked pages are identified by setting **`errno`** to EBUSY.

Using the memdefaults System Call

The **`memdefaults(2)`** system call allows you to control the default NUMA policies of one or more processes.

Note that to change the default NUMA policies of a process, the real or effective user ID of the calling process must match the real or effective user ID of the receiving process, or the calling process must have the P_ OWNER privilege.

The specifications required for making the **`memdefaults`** call are as follows:

```
#include <sys/procset.h>
#include <sys/mman.h>

int memdefaults(cmd, idtype, id, arg)

int cmd;
```

```
idtype_t idtype;
id_t id;
void *arg;
```

The arguments are defined as follows:

cmd	the operation to be performed
idtype	the type of identifier specified by the *id* argument The *idtype* and *id* arguments together specify the process or set of processes for which the operation is to be performed. Acceptable values for the *idtype* argument and the corresponding interpretations of *id* are as follows:

P_PID	process ID
P_PPID	parent process ID
P_PGID	process group ID
P_SID	session ID
P_CID	scheduler class ID
P_UID	user ID
P_GID	group ID
P_ALL	all processes and LWPs In this case, the value of the *id* argument is ignored.

id	the identifier for the process or set of processes for which the operation is to be performed The value of this argument depends on the value of *idtype* as explained above. You may specify a value of **P_MYID** to be used with the value of *idtype* to specify the calling process's process ID, parent process ID, process group ID, session ID, scheduler class ID, user ID, or group ID.
arg	a pointer to an argument whose value depends on the value of *cmd*

Cmd can be one of the following. The values of arg that are associated with each command are indicated.

MDF_SETNUMA Sets the default NUMA policies of all of the specified processes to the values specified by *arg*. *Arg* is a pointer to an integer value that sets one or more of the following bits:

MDF_TEXT_GLOBAL	global text
MDF_TEXT_SOFTLOCAL	floating soft-local text
MDF_TEXT_HARDLOCAL	floating hard-local text
MDF_PRDATA_GLOBAL	global private data

MDF_PRDATA_SOFTLOCAL	floating soft-local private data
MDF_PRDATA_HARDLOCAL	floating hard-local private data
MDF_SHDATA_GLOBAL	global shared data
MDF_SHDATA_SOFTLOCAL	floating soft-local shared data
MDF_SHDATA_HARDLOCAL	floating hard-local shared data
MDF_UBLOCK_GLOBAL	global U-blocks
MDF_UBLOCK_SOFTLOCAL	floating soft-local U-blocks
MDF_UBLOCK_HARDLOCAL	floating hard-local U-blocks

Note that you may specify only one value from each of the text, private data, shared data, and U-block categories. If you do not specify a value for a particular category, the NUMA policy assigned to that category is not changed.

MDF_GETNUMA Returns the logical sum of the NUMA policies of the specified process(es) in the location referenced by *arg*

A return value of **0** indicates that the call to **`memdefaults`** has been successful. A return value of **-1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`memdefaults(2)`** system manual page for a listing of the types of errors that may occur.

The following C program segment shows how to use the **`memdefaults`** system call to set the default NUMA policy for the U-block and private data to hard-local and the default NUMA policy for text to soft-local.

```
#include <sys/types.h>
#include <sys/procset.h>
#include <sys/mman.h>
#include <stdio.h>
#include <errno.h>

void
main()
{
    int cmd;
    idtype_t idtype;
    id_t id;
    uint_t flags;

    cmd = MDF_SETNUMA;
    idtype = P_PID;
    id = P_MYID;
    flags = MDF_PRDATA_HARDLOCAL | MDF_UBLOCK_HARDLOCAL | MDF_TEXT_SOFTLOCAL;

    if (memdefaults(cmd,idtype,id,&flags) == -1) {
        printf (“memdefaults failed, errno = %d\n”, errno);
        exit (1);
    }

    printf (“default NUMA policies set successfully\n”);
    exit (0);
}
```

It is important to note that use of **`memdefaults`** to change NUMA policies may cause those parts of the program's address space that are bound to one memory pool to migrate to a new pool. Migration may be from local to global memory or from global to local memory. If a page requiring migration to another memory pool is locked, **`memdefaults`** will fail, and **`errno`** will be set to EBUSY. If the program is migrated to a CPU on a different processor board through use of the **`mpadvise(3C)`** library routine, the **`cpu_bias(2)`** system call, or the **`rerun(1)`** command, the parts of its address space that are bound to local memory will be migrated to the local memory pool on that board.

For additional information on use of the **`memdefaults(2)`** system call, refer to the corresponding system manual page.

Using the run and rerun Commands

The **`run(1)`** command executes a specified command in an environment that is set by a specified list of options. Such options include processor bias, memory bindings, scheduling policy and priority, and time quantum. The **`rerun(1)`** command alters the execution environment of one or more running processes. Aspects of the environment that may be altered include processor bias, processor assignment, memory bindings, scheduling policy and priority, and time quantum.

To select the NUMA policies for a command or one or more running processes, invoke the **`run`** or **`rerun`** command from the shell, and specify the **`-m`** *NUMA* option. *NUMA* specifies one or more keywords that select the NUMA policies for parts of the process's address space. Keywords must be separated by commas. The valid keywords are described as follows:

`global`	Selects the global NUMA policy for all categories
`local`	Selects the floating soft-local NUMA policy for all categories
`hard`	Selects the floating hard-local NUMA policy for all categories
`text_global`	Selects the global NUMA policy for text
`text_local`	Selects the floating soft-local NUMA policy for text
`text_hard`	Selects the floating hard-local NUMA policy for text
`prdata_global`	Selects the global NUMA policy for private data
`prdata_local`	Selects the floating soft-local NUMA policy for private data
`prdata_hard`	Selects the floating hard-local NUMA policy for private data
`shdata_global`	Selects the global NUMA policy for shared data
`shdata_local`	Selects the floating soft-local NUMA policy for shared data
`shdata_hard`	Selects the floating hard-local NUMA policy for shared data

`ublock_global`	Selects the global NUMA policy for the U-block
`ublock_local`	Selects the floating soft-local NUMA policy for the U-block
`ublock_hard`	Selects the floating hard-local NUMA policy for the U-block

The following example shows how to invoke the **`run`** command to set the NUMA policy for *x_cmd*'s U-block and private data to the floating soft-local NUMA policy:

```
run -m ublock_local,prdata_local x_cmd
```

The following example shows how to invoke the **`rerun`** command to run *proc_n* on CPU 4 and set the NUMA policy for the process's pages to the floating hard-local NUMA policy:

```
rerun -b 4 -m hard proc_n
```

Note that use of the **`rerun`** command to change NUMA policies or CPU bias may cause those parts of the program's address space that are bound to local memory to migrate to a new memory pool. If the program is migrated to a different processor board, the parts of the program that are bound to local memory will be migrated to the local memory pool on that board.

The **`run`** command also allows you to determine the current NUMA policies for the current shell. To do so, invoke **`run`** from the shell without specifying any options. The **`run`** command provides the current NUMA memory bindings.

For additional information on use of the **`run(1)`** and **`rerun(1)`** commands, refer to the corresponding system manual pages.

Memory Management Facilities

The UNIX system provides a complete set of memory management mechanisms, providing applications complete control over the construction of their address space and permitting a wide variety of operations on both process address spaces and the variety of memory objects in the system. Process address spaces are composed of a vector of memory pages, each of which can be independently mapped and manipulated. Typically, the system presents the user with mappings that simulate the traditional UNIX process memory environment, but other views of memory are useful as well.

The memory-management facilities:

- Unify the system's operations on memory.
- Provide a set of kernel mechanisms powerful and general enough to support the implementation of fundamental system services without special-purpose kernel support.
- Maintain consistency with the existing environment, in particular using the UNIX file system as the name space for named virtual-memory objects.

Virtual Memory, Address Spaces, and Mapping

The system's virtual memory (VM) consists of all available physical memory resources. Examples include local and remote file systems, processor primary memory, swap space, and other random-access devices. Named objects in the virtual memory are referenced through the UNIX file system. However, not all file system objects are in the virtual memory; devices such as terminal and network device files that cannot be treated as storage are not in the virtual memory. Some virtual memory objects, such as private process memory and System V shared memory segments, do not have names.

A process's address space is defined by mappings onto objects in the system's virtual memory (usually files). Each mapping is constrained to be sized and aligned with the page boundaries of the system on which the process is executing. Each page may be mapped (or not) independently. Only process addresses that are mapped to some system object are valid, for there is no memory associated with processes themselves—all memory is represented by objects in the system's virtual memory.

Each object in the virtual memory has an object address space defined by some physical storage. A reference to an object address accesses the physical storage that implements the address within the object. The virtual memory's associated physical storage is thus accessed by transforming process addresses to object addresses, and then to the physical store.

A given process page may map to only one object although a given object address may be the subject of many process mappings. An important characteristic of a mapping is that the object to which the mapping is made is not affected by the mere existence of the mapping. Thus, it cannot, in general, be expected that an object has an awareness of having been mapped or of which portions of its address space are accessed by mappings; in particular, the notion of a page is not a property of the object. Establishing a mapping to an object simply provides the potential for a process to access or change the object's contents.

The establishment of mappings provides an access method that renders an object directly addressable by a process. Applications may find it advantageous to access the storage resources they use directly rather than indirectly through **`read`** and **`write.`** Potential advantages include efficiency (elimination of unnecessary data copying between the kernel and the application) and reduced complexity (single-step updates rather than the *read*-*modify* buffer-*write* cycle). The ability to access an object and have it retain its identity over the course of the access is unique to this access method and facilitates the sharing of common code and data.

Networking, Heterogeneity and Integrity

VM is designed to fit well with the larger UNIX heterogeneous environment. This environment makes extensive use of networking to access file systems—file systems that are now part of the system's virtual memory. Networks are not constrained to consist of similar hardware or to be based upon a common operating system; in fact, the opposite is encouraged, for such constraints create serious barriers to accommodating heterogeneity. While a given set of processes may apply a set of mechanisms to establish and maintain the properties of various system objects—properties such as page sizes and the ability of objects to synchronize their own use—a given operating system should not impose such mechanisms on the rest of the network.

As it stands, the access method view of a virtual memory maintains the potential for a given object (say a text file) to be mapped by systems running the UNIX memory management system and also to be accessed by systems for which virtual memory and storage management techniques such as paging are totally foreign, such as PC-DOS. Such systems can continue to share access to the object, each using and providing its programs with the access method appropriate to that system. The unacceptable alternative would be to prohibit access to the object by less capable systems.

Another consideration arises when applications use an object as a communications channel, or otherwise try to access it simultaneously. In both cases, the object is shared; thus, applications must use some synchronization mechanism to maintain the integrity of their actions on it. The scope and nature of the synchronization mechanism is best left to the application. For example, file access on systems that do not support virtual memory access methods must be indirect, by way of **`read`** and **`write.`** Applications sharing files on such systems must coordinate their access using semaphores, file locking, or some application-specific protocols. What is required in an environment where mapping replaces **`read`** and **`write`** as the access method is an operation, such as **`fsync`**, that supports atomic update operations.

The nature and scope of synchronization over shared objects is application-defined from the outset. If the system tried to impose automatic semantics for sharing, it might prohibit other useful forms of mapped access that have nothing to do with communication or sharing. By providing the mechanism to support integrity and by leaving it to cooperating applications to apply the mechanism, the needs of applications are met without eliminating diversity. Note that this design does not prohibit the creation of libraries that provide abstractions for common application needs. Not all abstractions on which an application builds need be supplied by the operating system.

Memory Management Interfaces

The applications programmer gains access to VM facilities through several sets of system calls. The next sections summarize these calls and provide examples of their use.

Creating and Using Mappings

The memory mapping facilities allow processes to access the data in memory objects directly by mapping portions of their address spaces onto the objects. A *memory object* is defined as an object that contains an array of bytes. It has an address space that begins at zero and extends through the length of the object minus one. Each byte of data in the object can be identified by its offset in the object. Additionally, a memory object is an object for which a file descriptor can be obtained. It includes POSIX shared memory objects, regular files, and some devices (for an explanation of POSIX shared memory objects, see Chapter 12). One of the files to which a mapping can be established is the **`/proc/`***pid***`/as`** file, which is described following this section. One of the devices to which a mapping can be established is **`/dev/zero`**, which is also described following this section.

The data in a memory object are shared if multiple processes establish mappings to the same portion of the object. If the mappings permit shared write access, data written to the object through the address space of one process are visible in the address spaces of the other processes.

The operating system performs memory mapping operations on whole pages. Consequently, the length of a mapping is rounded up to the next multiple of the system's page size. The length of a process's mapping to a memory object may exceed the length of the object at the time of the call to **`mmap`**. The only requirement when creating a mapping is that the addresses, lengths, and offsets specified in the operation be possible (that is, within the range permitted for the object in question), not that they exist at the time the mapping is created (or subsequently). After the process's mapping has been established, the length of the object may increase or decrease at any time because it can be manipulated by other processes. Such changes may affect processes' accesses to the mapped area.

If the size of a memory object is not a multiple of the system's page size, then the operating system treats the page that contains the end of the object in a special way: it fills the locations that are beyond the end of the object but within this last page with zeros. If a process that has established a mapping to the memory object writes to the zero-filled portion of the last page, the operating system will not write the modified data to secondary storage. If the process references a page that is within the mapping but beyond the last page associated with the memory object, a `SIGBUS` signal will be sent to the process. A `SIGBUS` signal may also be sent to the process if a reference to a page will cause an error in the mapped object (an out of space condition, for example). If the process references a page that is not within the mapping, a `SIGSEGV` signal will be sent to the process. A `SIGSEGV` signal will also be sent to the process if it attempts to (1) write to a page within the mapping that has been mapped without write access or (2) access a page within the mapping that has been mapped with no access.

In general, if a program makes a reference to an address that is inconsistent with the mapping (or lack of a mapping) established at that address, the system will respond with a `SIGSEGV` violation. However, if a program makes a reference to an address consistent with how the address is mapped but that address does not evaluate at the time of the access to allocated storage in the object being mapped, then the system will respond with a `SIGBUS` violation. In this manner, a program (or user) can distinguish between whether it is the mapping or the object that is inconsistent with the access and take appropriate remedial action.

The mapping between a process's address space and a memory object remains until the process removes it by invoking the **`munmap(2)`** system call (see "Removing Mappings," p. 6-26, for an explanation of this call). If the process invokes the **`exec(2)`** or **`exit(2)`** system calls, the operating system removes the mapping.

The **`mmap(2)`** system call establishes a mapping between a process's address space and an object in the system's virtual memory. All other system functions that contribute to the definition of an address space are built from **`mmap`**, the system's most fundamental function for defining the contents of an address space.

The specifications required for making the **`mmap`** call are as follows:

```
#include <sys/mman.h>

void *mmap(addr, len, prot, flags, fildes, off)

void *addr;
size_t len;
int prot;
int flags;
int fildes;
off_t off;
```

The format for invoking **mmap** from an application program is presented as follows:

```
pa = mmap(addr, len, prot, flags, fildes, off);
```

The **mmap** call establishes a mapping from the process's address space at an address *pa* for *len* bytes to the object specified by *fildes* at offset *off* for *len* bytes. A successful call to **mmap** returns *pa* as its result; *pa* is an implementation-dependent function of the argument *addr* and the value of *flags* as described in the paragraphs that follow. The address range [*pa, pa* + *len*) must be valid for the address space of the process and the range [*off*, *off* + *len*) must be valid for the virtual memory object. (The notation [*start*, *end*) denotes the interval from *start* to *end,* including *start* but excluding *end.*)

NOTE

The mapping established by **mmap** replaces any previous mappings for the process's pages in the range [*pa, pa* + *len*).

To memory map a region of a file whose size exceeds 2GB, the **mmap64** function must be used as follows:

```
pa = mmap64(addr, len, prot, flags, fildes, off);
```

In this case the file offset, *off*, must be declared as type *off64_t*. The other arguments have the same types and meaning as with **mmap**. As with **mmap**, memory mappings using **mmap64** can be removed using **munmap**.

The arguments to **mmap** are defined as follows:

addr if the **MAP_FIXED** option is specified in the *flags* argument, specifies the starting address of the portion of the calling process’s virtual address space that is to be mapped to the memory object specified by *fildes*. The specified address must be a multiple of the system page size. The system page size is available to an application through use of the **sysconf(3C)** library routine. Note that when you specify the **MAP_FIXED** option, the mapping will replace any existing mapping at the address specified by *addr.*

If the **MAP_FIXED** option is not specified, the value of *addr* may be zero or a nonzero value that specifies an appropriate starting address for the mapping. Specifying a value of zero grants the operating system complete freedom in selecting an address for the mapping. Specifying a nonzero value provides the operating system with a suggested address near which the mapping may be placed. The operating system will select an address that is suitable for mapping *len* bytes to the specified memory object. In selecting an address, the operating system will not place a mapping at address **0**, replace an existing mapping, or map areas that are considered part of the process's potential data or stack segments.

len	the length in bytes of the mapping. This value is not required to be a multiple of the system's page size. Because the operating system performs memory mapping operations on whole pages, the length of the mapping may be rounded up to the next multiple of the system's page size.

prot an integer value that specifies one or more of the following options and determines the access permissions associated with the data to which the calling process's address space is being mapped. This value is either

PROT_NONE	permits no access to the data in the memory object

or

the bitwise inclusive OR of one or more of the following:

PROT_READ	permits read access to the data in the memory object
PROT_WRITE	permits write access to the data in the memory object
PROT_EXEC	permits execute access to the data in the memory object

It is important to note that regardless of the value specified by *prot*, the memory object specified by *fildes* must have been opened with read permission. If **PROT_WRITE** is specified and the **MAP_SHARED** option is specified in the *flags* argument, the memory object must have been opened with write permission.

A write access must fail if **PROT_WRITE** has not been set although the behavior of the write can be influenced by setting **MAP_PRIVATE** in the *flags* argument. The *flags* argument provides other information about the handling of mapped pages as described in the following paragraphs

flags provides information about the mapping. The value specified by *flags* is the bitwise inclusive OR of the following options.

Note that **MAP_SHARED** and **MAP_PRIVATE** are mutually exclusive. One of them must be specified.

MAP_SHARED indicates that changes to data in the memory object are to be shared

Note that when **MAP_SHARED** is specified, a store to a page will modify the memory object such that it is changed for all processes that have mapped a portion of their address spaces to it. The act of storing into a `MAP_SHARED` mapping is equivalent to doing a **`write`** system call.

After a **`fork(2)`** system call, the child process has the memory object mapped to the same address range as the parent and has it mapped **MAP_SHARED**. A store performed by either the parent or the child to a **MAP_SHARED** page will modify the underlying memory object.

MAP_PRIVATE indicates that changes to data in the memory object are private to the calling process

When **MAP_PRIVATE** is specified, the calling process's first store to a page creates a private copy of the page and redirects the mapping and the initial and successive write references to the copy; changes to the copy do not affect the underlying memory object.
This operation is sometimes referred to as copy-on-write and occurs invisibly to the process causing the store. Only pages actually modified have copies made in this manner.

Until the calling process's first store to a **MAP_PRIVATE** page, changes made by other processes that have mapped the page **MAP_SHARED** will be visible. If an application needs isolation from changes made by other processes, it should use **`read`** to make a copy of the data it wishes to keep isolated. After the store has occurred, no other process can modify the page because the calling process has a private copy of it.

After a **`fork(2)`** system call, the child process has the memory object mapped to the same address range as the parent and has it mapped **MAP_PRIVATE**. The first store performed by either the parent or the child to a **MAP_PRIVATE** page creates a private copy of the page. Changes to either process's copy do not affect the underlying memory object. Changes made by the parent are not visible to the child. Changes made by the child are not visible to the parent.

MAP_PRIVATE mappings are used by system functions such as **`exec(2)`** when mapping files containing programs for execution. This permits operations by programs such as debuggers to modify the text (code) of the program without affecting the file from which the program is obtained.

MAP_FIXED indicates that the specified memory object must be mapped to the calling process's virtual address space at the exact location specified by *addr*. You are advised to use caution in specifying this option because it may prevent the system from making the most effective use of system resources. The system strives to choose alignments for mappings that maximize the performance of its hardware resources.

Note that when you specify this option, the mapping will replace any existing mapping in the address range specified by the *addr* and *len* arguments.

The following values set the NUMA policy for the pages being mapped (NUMA refers to non-uniform memory access as described in "Overview of Primary Memory"). Specifying one of these values is optional. If you do not specify a value, the default NUMA policy is the policy that has been established previously by invoking the **`memdefaults(2)`** system call. Only one of the following values may be specified:

MAP_GLOBAL	selects the global NUMA policy, which indicates that the pages are to be placed in global memory
MAP_FLOATSOFT	selects the floating soft-local NUMA policy, which indicates that the pages are to be placed in local memory if possible and global memory if not (for an explanation of the term floating, refer to p. 6-5)
MAP_FLOATHARD	selects the floating hard-local NUMA policy, which indicates that the pages are to be placed only in local memory (for an explanation of the term floating, refer to p. 6-5). If the necessary pages are not available, the process blocks until the pages become available. It is important to note that selection of this policy for memory mapped files does not guarantee that the pages will be placed in local memory. Pages will not be placed in local memory if they are writable and are being accessed from different processor boards; they will be placed in global memory instead. Use of the hard-local NUMA policy can cause a local memory pool to thrash. For this reason, you are advised to select this policy only when the application mix is well-known and the use of local memory is strictly controlled.
MAP_ANCHORSOFT	selects the anchored soft-local NUMA policy, which indicates that the pages are to be placed in local memory if possible and global memory if not (for an explanation of the term anchored, refer to p. 6-5)
MAP_ANCHORHARD	selects the anchored hard-local NUMA policy, which indicates that the pages are to be placed only in local memory (for an explanation of the term anchored, refer to p. 6-5). If the necessary pages are not available, the process blocks until the pages become available.

MAP_ANYPOOL selects the any-pool NUMA policy, which indicates that the pages may be placed in any of the system's memory pools

This policy is similar to the soft-local NUMA policy in that it prefers to use local rather than global page frames, but it is different from the other policies in that it considers remote page frames acceptable as a last resort. With the other policies, a remote page will migrate to or be replicated in a pool that is more acceptable. With the any-pool policy, the remote frame will be used directly.

The any pool NUMA policy is available only through use of the **`memcntl`** and **`mmap`** system calls. You are advised to use it sparingly.

The following values set the cache policy for the pages being mapped. Specifying one of these values is optional. If you do not specify a value, the default cache policy is copyback. Only one of the following values may be specified:

MAP_NOCPUCACHE selects the no-cache CPU cache policy, which indicates that accesses to the pages are to bypass the CPU's data cache

MAP_CBCPUCACHE selects the copyback CPU cache policy, which indicates that accesses to the pages are to be cached in copyback mode

In copyback mode, a CPU write transaction usually updates the data cache only; it does not immediately update memory. Later when the cache line is displaced or invalidated, the data are written to memory.

fildes the file descriptor identifying the memory object to which the process's address space is to be mapped. A process must have obtained this descriptor previously by invoking the **`open(2)`** or **`creat(2)`** system call or the **`shm_open(3C)`** library routine (see Chapter 12 for an explanation of this routine).

The file descriptor used in a **`mmap`** call need not be kept open after the mapping is established. If it is closed, the mapping will remain until such time as it is replaced by another call to **`mmap`** that explicitly specifies the addresses occupied by this mapping or until the mapping is removed

off the byte offset in the memory object where the mapping is to begin. The value of *off* must be a multiple of the system page size. The system page size is available to an application through use of the **`sysconf(3C)`** library routine. The range beginning at *off* and extending for the number of bytes specified by *len* must be valid for the possible (not necessarily current) offsets in the memory object. The size of a memory object cannot be extended by using **`mmap`**.

If the call is successful, **`mmap`** returns the virtual address at which the mapping of the memory object begins. A return value of **`-1`** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`mmap(2)`** system manual page for a listing of the types of errors that may occur.

Using **`mmap`** to access system memory objects can simplify programs in a variety of ways. Keeping in mind that **`mmap`** can really be viewed as just a means to access memory objects, it is possible to program using **`mmap`** in many cases where you might program with **`read`** or **`write.`** However, it is important to realize that **`mmap`** can be used only to gain access to memory objects—those objects that can be thought of as randomly accessible storage. Thus, terminals and network connections cannot be accessed with **`mmap`** because they are not memory. Magnetic tapes, even though they are memory devices, cannot be accessed with **`mmap`** because storage locations on the tape can be addressed only sequentially. Some examples of situations that can be thought of as candidates for use of **`mmap`** over more traditional methods of file access include:

- Random access operations—either map the entire file into memory or if the address space cannot accommodate the file or the file size is variable, create "windows" of mappings to the object.
- Efficiency—even in situations where access is sequential, if the object being accessed can be accessed via **`mmap`**, an efficiency gain may be obtained by avoiding the copying operations inherent in accesses via **`read`** or **`write.`**
- Structured storage— if the storage being accessed is collected as tables or data structures, algorithms can be more conveniently written if access to the file is treated as though the tables were in memory. Previously, programs could not simply make storage or table alterations in memory and save them for access in subsequent runs; however, when the addresses of a table are defined by mappings to a file, then changes to that storage are changes to the file and are, thus, automatically recorded in it.
- Scattered storage—if a program requires scattered regions of storage, such as multiple heaps or stack areas, such areas can be defined by mapping operations during program operation.

In some cases, devices or files are useful only if accessed via mapping. An example of this is frame buffer devices used to support bit-mapped displays, where display management algorithms function best if they can operate randomly on the addresses of the display directly.

Not all device drivers support memory mapping. If you try to map a device that does not support mapping, **`mmap`** fails.

Finally, it is important to remember that mappings can be operated upon at the granularity of a single page. Even though a mapping operation may define multiple pages of an address space, there is no restriction that subsequent operations on those addresses must operate on the same number of pages. For instance, an **`mmap`** operation defining ten pages of an address space may be followed by subsequent **`munmap`** operations that remove every other page from the address space, leaving five mapped pages, each followed by an unmapped page (for information on **`munmap`**, see "Removing Mappings," p. 6-26). Those unmapped pages may subsequently be mapped to different locations in the same or different objects, or the whole range of pages (or any partition, superset, or subset of the pages) may be used in other **mmap** or other memory management operations. Further, it must be noted that any mapping operation that operates on more than a single page can partially succeed in that some parts of the address range can be affected even though the call returns a failure. Thus, an **mmap** operation that replaces another mapping, if it fails, may have deleted the previous mapping and failed to replace it. Similarly, other operations (unless specifically stated otherwise) may process some pages in the range successfully before operating on a page where the operation fails.

The sections that follow explain the special cases of establishing a mapping to a target process's address space and establishing a mapping to **/dev/zero**.

Establishing a Mapping to a Target Process's Address Space

For each running process, the **/proc** file system provides a file that represents the address space of the process. The name of this file is **/proc/***pid***/as**, where *pid* denotes the process ID of the process whose address space is represented. A process can open a **/proc/***pid***/as** file and use the **read(2)** and **write(2)** system calls to read and modify the contents of another process's address space. As mentioned previously, a process can use **mmap** to map a portion of its address space to a **/proc/***pid***/as** file and directly access the contents of another process's address space. A process that establishes a mapping to a **/proc/***pid***/as** file is hereinafter referred to as a *monitoring process*. A process whose address space is being mapped is referred to as a *target process*.

To establish a mapping to a **/proc/***pid***/as** file, the following requirements must be met:

- The file must have been opened with read permission. If you intend to modify the target process's address space, the file must also have been opened with write permission.
- On the call to **mmap** to establish the mapping, the *flags* argument must specify the **MAP_SHARED** option. It may not specify any of the options that set the NUMA policy or the cache policy for the pages being mapped. The NUMA policy and cache policy associated with the mapping are those set by the target process.

The following C program segment shows how to establish a mapping to a **/proc/***pid***/as** file.

```
/*
 *  Header files
 */
#include <sys/types.h>
#include <sys/mman.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

     . . .

/*
 *  Data declarations
 */
     unsigned long  pagemask   /* for adjusting to page boundary */
     caddr_t        address;   /* target address to map */
     long           offset;    /* page address to map */
     caddr_t        mapaddr;   /* resulting page address of map */
     unsigned long  delta;     /* offset of target within its page */
     size_t         nbytes;    /* number of bytes to map */
     pid_t          pid;       /* process id of target */
     int            fd;        /* file descriptor for /proc file */
     char           name[16];  /* "/proc/00000/as" */

     . . .

/*
 * Assign values to address and pid
 */

        . . .

/*
 *  mmap requires the target address to be on a page boundary.
 *  Round down the address value and increase the nbytes accordingly.
 */

     pagemask = ~ (getpagesize() - 1);
     offset = (long) ((unsigned long) address & pagemask);
     delta = (unsigned long) address - (unsigned long) offset;
     nbytes += delta;

/*
 *  Open the address space file in the /proc file system.
 */
     sprintf(name, "/proc/%d/as", pid);
     fd = open(name, O_RDWR, 0);
     if (fd < 0) {
          perror(name);
          return;
     }


/*
 *  Map the target memory.  The mapping is to nbytes bytes starting at
 *  offset (which is on a page boundary) of the address space memory
 *  object identified by fd.  The resulting virtual address of offset
 *  is returned in mapaddr.  The virtual mapped address of the original
 *  address value is mappaddr+delta.
 */

     mapaddr = (caddr_t) mmap((caddr_t)0, nbytes,
          PROT_READ|PROT_WRITE, MAP_SHARED, fd, offset);

/*
 *  After mapping, the proc file can be closed.
 */
     close(fd);

/*
 *  Test the return value for a failed mmap attempt.
 */
     if ((int) mapaddr == -1) {
          perror("mmap");
          return;
     }
```

It is important to note that a monitoring process's mapping is to the memory object to which the target process's pages in the range [*off*, *off* + *len*) are mapped. As a result, a monitoring process's mapping to a target process's address space can become invalid if the target's mapping changes. In such circumstances, the monitoring process retains a mapping to the underlying memory object, but the mapping is no longer shared with the target process. Because a monitoring process cannot detect that a mapping is no longer valid, you must make provision in your application for controlling the relationship between the monitoring process and the target.

Circumstances in which a mapping to a target process's address space becomes invalid are as follows:

- The target process terminates.
- The target process unmaps a page in the range [*off*, *off* + *len*).
- The target process maps a page in the range [*off*, *off* + *len*) to a different memory object.
- The target process invokes **`fork(2)`** and stores into an unlocked, private, writable page in the range [*off*, *off* + *len*) before the child process does.

  In this case, the target process receives a private copy of the page, and its mapping and write reference are redirected to the copy (see p. 6-18 for an explanation of **MAP_PRIVATE** mappings and the **`fork`** system call). The monitoring process retains a mapping to the original memory object.
- The target process invokes **`fork(2)`** and locks in memory a private, writable page in the range [*off*, *off* + *len*) that is being shared with the child process pending a copy-on-write.

  In this case, the process that performs the lock operation receives a private copy of the page (as though it has performed the first store to the page). If it is the parent, or target, process that locks the page, then the monitoring process's mapping is no longer valid.
- The target process invokes **`mprotect(2)`** to enable write permission on a locked, private, read-only page in the range [*off*, *off* + *len*) that is being shared with the child process pending a copy-on-write.

  In this case, the target process receives a private copy of the page. The monitoring process retains a mapping to the original memory object.

If your application is expected to be the target of address space mapping by a monitoring process, you are advised to:

- Perform memory-locking operations in the target process before its address space is mapped by the monitoring process.
- Prior to invoking **`fork`**, lock in memory any pages for which mappings by the parent and the monitoring process need to be retained.

If your application is not expected to be the target of address space mapping, you may wish to postpone locking pages in memory until after invoking **`fork`**.

Finally, if a monitoring process attempts to write to a private, writable page that the target process is sharing with a child process pending a copy-on-write, the operation will fail, and a `SIGSEGV` signal will be sent to the monitoring process.

It is important to note that the OS also supports the **`usermap(3rt)`** and **`usermap(3F77rt)`** library routines, which allow a target process's address space to be mapped onto the virtual address space of another process. For information on the use of these routines, refer to the corresponding system manual pages. It is recommended that you consider the following prior to using these routines.

- The **mmap** system call is a standard System V interface although the capability of using it to establish mappings to **/proc/***pid***/as** files is a Concurrent extension. The **usermap** routines are wholly Concurrent extensions.
- **Mmap** provides control over the page protections and the location of mappings in the calling process. The **usermap** routines do not.
- Based on the assumption that you have identified the pages that are to be mapped, **mmap** provides independent mappings (two requests for a mapping to the same page result in two separate mappings). The **usermap** routines are intended to be used for mappings to particular data items and so can avoid redundant mappings to the same target page.
- When invoking **mmap**, you specify an open file descriptor. It is your responsibility to open and close the target memory object at appropriate times. When invoking one of the **usermap** routines, you specify a process identifier. **Usermap** opens the corresponding **/proc/***pid***/as** file. Because **usermap** is expected to be called multiple times, the file descriptor remains open. Leaving the file descriptor open may not be appropriate in all cases.

Establishing a Mapping to /dev/zero

As mentioned previously, one of the devices to which a process can establish a mapping is a pseudo-device called **/dev/zero**. A process can read from or write to **/dev/zero**. If it uses the **read(2)** system call to read from **/dev/zero**, it obtains a buffer full of zeros. If it uses the **write(2)** system call to write to **/dev/zero**, the write operation succeeds, but the data that are written are ignored.

A process can use **mmap** to map a portion of its address space to **/dev/zero**. Each time a process establishes a mapping to **/dev/zero**, it creates a mapping to a new, unnamed memory object that is filled with zeros. If two processes map **/dev/zero**, each has a mapping to a different zero-filled memory object. If one process maps **/dev/zero** twice, it has mappings to two different zero-filled memory objects. When a process has **/dev/zero** mapped **MAP_SHARED**, then a child process created through a subsequent call to **fork(2)** will share the object with the parent. A store performed by either the parent or the child will be visible to both. When a process has **/dev/zero** mapped **MAP_PRIVATE**, then a child process created through a subsequent call to **fork** will get a copy of the object in its address space.

The following code fragment demonstrates a use of **/dev/zero** to create a block of scratch storage in a program at an address of the system's choosing.

```
/*
 * Function to allocate a block of zeroed storage.  Parameter
 * is the number of bytes desired.  The storage is mapped as
 * MAP_SHARED so that if a fork occurs, the child process
 * will be able to access and modify the storage.  If we wished
 * to cause the child's modifications (as well as those by the
 * parent) to be invisible to the ancestry of processes, we
 * would use MAP_PRIVATE.
 */
caddr_t
get_zero_storage(int len);
{
    int fd;
    caddr_t result;

    if ((fd = open("/dev/zero", O_RDWR)) == -1)
        return ((caddr_t)-1);
    result = mmap(0, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    (void) close(fd);
    return (result);
}
```

As written, this function permits a hierarchy of processes to use the area of allocated storage as a region of communication (for implicit interprocess communication purposes).

Removing Mappings

The **munmap(2)** system call removes all mappings for pages in a specified range from the address space of the calling process. It is not an error to remove mappings from addresses that do not have them, and any mapping, regardless of how it has been established, can be removed with **munmap.** The **munmap** call does not in any way affect the objects that have been mapped at those addresses.

If the call to **munmap** is successful, further references to the previously mapped pages will result in delivery of a SIGSEGV signal to the process. Any changes to pages that have been mapped **MAP_PRIVATE** will be discarded. Any memory locks associated with the mapped pages will be removed (see "Memory Page Locking", p.6-27, and "Address Space Locking", p.6-33, for information on memory locking facilities).

The specifications required for making the **munmap** call are as follows:

```
#include <sys/mman.h>

int munmap(addr, len)

void *addr;
size_t len;
```

The arguments are defined as follows:

addr — the starting address of the portion of the calling process's virtual address space that is to be unmapped. The specified address must be a multiple of the system page size. The system page size is available to an application through use of the **sysconf(3C)** library routine.

len — the length in bytes of the portion of the process's address space that is to be unmapped

A return value of **0** indicates that the call to **munmap** has been successful. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **munmap(2)** system manual page for a listing of the types of errors that may occur.

Cache Control

The UNIX memory management system can be thought of as a form of cache management in which a processor's primary memory is used as a cache for pages from objects from the system's virtual memory. Thus, there are several operations that control or interrogate the status of this cache, as described in this section.

Memory Cache Control

```
int
memcntl(caddr_t addr, size_t len, int cmd, caddr_t arg, int attr, int mask);
```

The **memcntl(2)** system call provides several control operations over mappings in the range [*addr*, *addr* + *len*), including locking pages into physical memory, unlocking them, and writing pages to secondary storage. The functions described in the rest of this section offer simplified interfaces to the **memcntl** operations.

Memory Page Locking

Using the mlock and munlock Library Routines

The **mlock(3C)** and **munlock(3C)** library routines allow a process to lock and unlock pages within its virtual address space. The interfaces to these routines are based on UNIX System V Release 4 and IEEE Standard 1003.1b-1993. They are explained in the paragraphs that follow.

NOTE

To use these routines, the calling process must have the P_PLOCK privilege (for additional information on privileges, refer to the "Security Considerations" section of Chapter 9 and the **intro(2)** system manual page).

If you wish to be able to perform DMA (Direct Memory Access) transfers to or from the virtual address space of an application program, you must use the **userdma(2)** system call rather than the **mlock(3C)** library routine. Requirements and procedures are explained in "Using the userdma System Call."

As an alternative to embedding library calls in your application, you can use the multipurpose **setrun(1)** command to run a command so that all current and future mappings are locked and the process is immune to page stealing, page aging, and page swapping.

The syntax relating to process memory locking is:

```
setrun -e -l [command] [argument]
```

See the **setrun(1)** man page for a full description.

The **mlock** routine causes a specified range of the calling process's virtual pages to be locked in physical memory. The size of the area that is locked in memory is a multiple of the system page size. References to these pages (through other mappings in this or other processes) will not result in page faults that require an I/O operation to obtain the data needed to satisfy the reference. Because this operation ties up physical system resources and has the potential to disrupt normal system operation, use of this facility is restricted to users who have the P_PLOCK privilege. The system prohibits more than a configuration-dependent limit of pages to be locked in memory simultaneously, the call to **mlock** will fail if this limit is exceeded.

The pages will be resident until the process unlocks them by invoking the **munlock(3C)** or the **munlockall(3C)** library routine, invokes the **exec(2)** system call, or exits. Multiple lock operations performed by the process on the same range of virtual pages will be removed with a single call to **munlock**. If the process invokes **fork(2)**, the virtual pages within the child process will not be locked in physical memory.

CAUTION

The **fork** system call normally uses the copy-on-write technique to reduce the number of pages of the parent process that must be copied to the child. In this case, the copy is not made until the first write to the page is performed. To avoid the copy-on-write protection faults, copies of locked pages are made at the time of the call to **fork**. The performance of the **fork** operation will be significantly slower if the **fork** is performed when a large portion of an application's address space is locked.

The specifications required for using the **mlock** call are as follows:

```
#include <mman.h>

int mlock(addr, len)

void *addr;
size_t len;
```

The arguments are defined as follows:

addr	the starting address of the range of virtual address space that is to be locked in memory
len	the length in bytes of the range of virtual address space that is to be locked in memory

If the call to **mlock(3C)** is successful, the range of pages between *addr* and *addr* + *len* - **1** is locked in memory; a value of zero is returned. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error.

munlock releases the locks on physical pages. If multiple **mlock** calls are made through the same mapping, only a single **munlock** call will be required to release the locks (in other words, locks on a given mapping do not nest.) However, if different mappings to the same pages are processed with **mlock,** then the pages will stay locked until the locks on all the mappings are released.

Locks are also released when a mapping is removed, either through being replaced with an **mmap** operation or removed explicitly with **munmap.** A lock will be transferred between pages on the "copy-on-write" event associated with a `MAP_PRIVATE` mapping, thus locks on an address range that includes `MAP_PRIVATE` mappings will be retained transparently along with the copy-on-write redirection (see **mmap** above for a discussion of this redirection).

The **munlock(3C)** library routine allows the calling process to unlock a specified range of its virtual pages. The size of the area that is unlocked is a multiple of the system page size.

The specifications required for using the **munlock** call are as follows:

```
#include <mman.h>

int munlock(addr, len)

void *addr;
size_t len;
```

The arguments are defined as follows:

addr	the starting address of the range of virtual address space that is to be unlocked
len	the length in bytes of the range of virtual address space that is to be unlocked

Upon successful completion, the **munlock(3C)** routine returns a value of zero. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error.

Using the userdma System Call

The **userdma** system call allows you to use an I/O controller's DMA (Direct Memory Access) capabilities directly from user mode. It prepares an I/O buffer located in a user process's virtual address space for DMA transfers.

Standard DMA hardware operates at the physical memory level; it bypasses memory management units and sometimes data caches. To be able to perform DMA transfers to or from the virtual address space of an application program, the following requirements must be met:

- The application's buffer must be locked in physical memory; that is, the buffer must be resident, and the virtual to physical mappings must not be allowed to change.
- The application must know the physical location of the buffer.
- CPU access and I/O access to the buffer must be coherent.

- The virtual pages containing the buffer must be marked “used” and for DMA read operations, “modified.” The **`userdma(2)`** system call ensures that all of these requirements are met.

The specifications required for using the **`userdma`** call are as follows:

```
#include <sys/types.h>
#include <sys/mman.h>

int userdma(addr, len, cmd, flags, vec, nvec)

caddr_t addr;
size_t len;
int cmd;
int flags;
struct dmavec *vec;
int nvec;
```

Arguments are defined as follows:

addr	the virtual address of the first byte of the user’s I/O buffer
len	the length in bytes of the I/O buffer
cmd	the operation to be performed on the I/O buffer
flags	an integer value that indicates how the I/O buffer is to be used. The value of *flags* depends upon the operation specified by *cmd*.
vec	the null pointer constant or a pointer to an array of **`dmavec`** structures to which the physical locations of the buffer fragments are returned. A **`dmavec`** structure contains the following fields:

```
paddr_t    dma_paddr; /*Physical address and ... */
uint_t     dma_plen; /*length of buffer fragment.*/
```

nvec	zero or the number of elements in the array pointed to by *vec*

Cmd must be one of the following. The values of *flags* that are associated with each command are indicated.

USERDMA_LOCK — fault the pages in the address range specified by the *addr* and *len* arguments into physical memory and ensure that physical memory and CPU data caches are coherent. A page may be locked multiple times through different mappings; however, within a mapping, locks on a page are not nested. Multiple lock operations on the same address in the same process will be removed with a single unlock operation. An unlock operation will be performed on a locked I/O buffer if a mapping is removed or a page is deleted when a file is removed or truncated.

It is important to note that the physical location of a locked buffer can change under the following circumstances:

- If a read-only, **MAP_PRIVATE** mapping is made writable by a call to **`mprotect(2)`**
- If the buffer resides in a local memory pool and is mapped to a file and another process attempts to access the file

As a result, it is advisable to use process private pages for locked I/O buffers.

The value of the *flags* argument may be one or both of the following or zero:

USERDMA_WRITE	indicates that the buffer is to be used to write to a device. The pages containing the buffer will be checked for read access and marked as "used."
USERDMA_READ	indicates that the buffer is to be used to read from a device. The pages containing the buffer will be checked for write access and marked as "used" and "modified."

If one or both of these flags are specified, the cache modes of the pages containing the buffer are altered as necessary to keep memory and cache coherent. If the value of *flags* is zero, the I/O buffer is locked in physical memory, but the cache modes of the pages containing the buffer are <u>not</u> modified.

If the value of *nvec* is greater than zero, **`userdma`** returns the physical location of the I/O buffer in the array pointed to by *vec*. Each element in the array describes a contiguous physical buffer fragment. The return value of **`userdma`** is the number of array elements used to describe the I/O buffer. If the value of *nvec* is zero, **`userdma`** does not return the physical location of the I/O buffer.

Note that to use this command, the calling process must have the P_PLOCK privilege (for additional information on privileges, refer to the "Security Considerations" section of Chapter 9 and the **`intro(2)`** system manual page).

USERDMA_UNLOCK unlock all of the pages in the address range specified by the *addr* and *len* arguments

The value of the *flags* argument should be the same as the value that was supplied on a corresponding USERDMA_LOCK call. The value of *vec* must be **NULL**; the value of *nvec* must be zero.

Note that to use this command, the calling process must have the P_PLOCK privilege (for additional information on

	privileges, refer to the "Security Considerations" section of Chapter 9 and the **intro(2)** system manual page).
USERDMA_VTOP	return the physical location of the I/O buffer located in the address range specified by the *addr* and *len* arguments in the same way that it is returned by a USERDMA_LOCK **userdma** call The value of the *flags* argument must be zero. Note that this command performs virtual to physical address translation only; it does not lock the I/O buffer in physical memory. You must lock the I/O buffer in physical memory prior to making a USERDMA_VTOP **userdma** call.

Upon successful completion of a USERDMA_LOCK or a USERDMA_VTOP call, the return value of **userdma** is the number of **dmavec** structures used to describe the physical location of an I/O buffer. The number of structures can range from one to as many as one per page. In the best case, when the entire buffer is physically contiguous, the number will be one. In the worst case, when none of the pages are contiguous, the number will be one per page. In any case, **userdma** takes advantage of any physical contiguity in describing a physical buffer fragment. Upon successful completion of a USERDMA_UNLOCK call, the return value of **userdma** is **0**. If an error occurs on a **userdma** call, the return value is **-1**, and **errno** is set to indicate the error.

You can compute the number of pages in the address range specified by the *addr* and *len* arguments (where *len* > **0**) by using the following formula:

```
nvec = ((unsigned)(addr + len - 1) / nbpp) - ((unsigned)(addr) / nbpp) + 1;
```

The variable **nbpp** represents the number of bytes per page on your system. You can obtain this value by using the **sysconf(3C)** library routine (see p. 6-36 for additional information on this routine).

You can use the value obtained from the formula to determine how large the array of **dmavec** structures needs to be. You can then supply a pointer to the array and the number of elements that it contains on a USERDMA_LOCK **userdma** call as follows:

```
nfrag = userdma(addr, len, USERDMA_LOCK, 0, vec, nvec);
```

This call locks the I/O buffer in physical memory and returns its physical location in the array that is pointed to by `vec`. Upon return from the call, `nfrag` contains the number of contiguous physical buffer fragments.

To unlock the I/O buffer, you can invoke **userdma** as follows:

```
userdma(addr, len, USERDMA_UNLOCK, 0, 0, 0);
```

For additional information on use of the **userdma(2)** system call, refer to the corresponding system manual page. User-level device drivers use **userdma**. An overview of user-level device drivers is provided in *Device Driver Programming.*

Address Space Locking

The **mlockall(3C)** and **munlockall(3C)** library routines are similar in purpose and restriction to **mlock(3C)** and **munlock(3C)** (see p. 6-27) except that they operate on entire address spaces.

NOTE

To use these routines, the calling process must have the P_PLOCK privilege (for additional information on privileges, refer to the “Security Considerations” section of Chapter 9 and the **intro(2)** system manual page).

If you wish to be able to perform DMA (Direct Memory Access) transfers to or from the virtual address space of an application program, you must use the **userdma(2)** system call rather than the **mlockall(3C)** library routine. Requirements and procedures are explained in “Using the userdma System Call:.”

The **mlockall** routine locks all of the calling process’s virtual address space in physical memory--that is, text, data, stack, and shared memory segments, memory-mapped files, and shared libraries. The address space remains resident until the process unlocks it by invoking the **munlockall** routine, invokes the **exec(2)** system call, or exits. Multiple lock operations performed by the process on its virtual address space will be removed with a single call to **munlockall**. If the process invokes **fork(2)**, the address space of the child process will not be locked in physical memory.

CAUTION

The **fork** system call normally uses the copy-on-write technique to reduce the number of pages that must be copied from the parent process to the child. With this technique, a page is not copied until the first write to the page is performed by either the parent or the child. However, to avoid the copy-on-write protection faults that result, copies of locked pages are made at the time of the call to **fork**. The performance of the **fork** operation will be significantly slower when a large portion of the parent’s address space is locked.

The specifications required for using the **mlockall** call are as follows:

```
#include <mman.h>

int mlockall(flags)

int flags;
```

The argument is defined as follows:

flags an integer value that sets one or both of the following bits:

MCL_CURRENT	causes all of the process's current virtual address space to be locked in physical memory
MCL_FUTURE	causes the new virtual pages to be locked in physical memory if the process's virtual address space is expanded in the future

If *flags* is (**MCL_CURRENT**|**MCL_FUTURE**), the lock is to affect everything that is currently in the process's address space and everything that is added in the future (both current and future mappings).

Upon successful completion, the **`mlockall(3C)`** routine returns a value of zero. A return value of **-1** indicates that an error has occurred; **`errno`** is set to indicate the error.

One call to the **`munlockall`** routine removes all locks on all pages in the process's virtual address space, whether the locks have been established by **`mlock, mlockall`**, **`plock`**, or **`userdma`**.

The specifications required for using the **`munlockall`** call are as follows:

```
#include <mman.h>

int munlockall()
```

The **`munlockall(3C)`** routine always returns a value of zero.

Memory Cache Synchronization

The **`msync(3C)`** routine supports applications that require assertions about the integrity of data in the storage backing their mapping, either for correctness or for coherent communications in a distributed environment. The **`msync`** routine causes all modified copies of pages over the range [*addr*, *addr* + *len*) to be flushed to the objects mapped by those addresses. In the cache analogy discussed in "Cache Control" (see p. 6-27), **`msync`** is the cache "write-back," or flush, operation. It is similar in purpose to the **`fsync(2)`** operation for files. The **`msync`** routine optionally invalidates such cache entries so that further references to the pages cause the system to obtain them from their permanent storage locations.

Secondary storage for the portion of a process's address space that has been mapped **MAP_SHARED** is the file to which it has been mapped. If a portion of a process's address space has been mapped **MAP_PRIVATE** and has been modified, it has no permanent secondary storage--it temporarily uses the swap area.

The specifications for making the **`msync`** call are as follows:

```
#include <sys/mman.h>

int msync(addr, len, flags)

void *addr;
size_t len;
int flags;
```

The arguments are defined as follows:

addr	the starting address of the memory-mapped file that contains modified data to be written to secondary storage. The specified address must be a multiple of the system page size. The system page size is available to an application through use of the **`sysconf(3C)`** library routine.
len	the length in bytes of the memory-mapped file that contains modified data to be written to secondary storage
flags	an integer value that sets one or more of the following bits:

Note that **MS_ASYNC** and **MS_SYNC** are mutually exclusive bits. <u>One</u> of them must be specified.

MS_ASYNC	indicates that the write operations are to be asynchronous. In this case, the **`msync`** routine will return as soon as all of the write operations have been queued.
MS_SYNC	indicates that the write operations are to be synchronous. In this case, the **`msync`** routine will not return until all of the write operations have been completed as defined for synchronized I/O data integrity completion (for an explanation of POSIX synchronized I/O, refer to the *Real-Time Programming Guide*).
MS_INVALIDATE	indicates that copies of pages in memory are to be invalidated. If these pages are referenced subsequently, the system will obtain them from secondary storage.

A return value of **0** indicates that the call to **`msync`** has been successful. A return value of **-1** indicates that an error has occurred; **`errno`** is set to indicate the error.

Memory Page Residency

```
int
mincore(caddr_t addr, size_t len, char *vec);
```

The **`mincore(2)`** system call determines the residency of the memory pages in the address space covered by mappings in the range [*addr, addr + len*). Using the cache concept described in “Cache Control” (see p. 6-27), this function can be viewed as an operation that interrogates the status of the cache and returns an indication of what is currently resident in the cache. The status is returned as a char-per-page in the character array referenced by **vec* (which the system assumes to be large enough to encompass all of the pages in the address range). Each character contains either a 1 (indicating that the page is resident in the system's primary storage) or a 0 (indicating that the page is not resident in primary storage.) Other bits in the character are reserved for possible future expansion—therefore, programs testing residency should test only the least significant bit of each character.

The **`mincore`** call returns residency information that is accurate at an instant in time. Because the system may frequently adjust the set of pages in memory, this information may quickly be outdated. Only locked pages are guaranteed to remain in memory.

Other Mapping Functions

Given the **_SC_PAGESIZE** argument, the **`sysconf(3C)`** routine returns the system-dependent size of a memory page. For portability, applications should not embed any constants specifying the size of a page, and instead should make use of **`sysconf`** to obtain that information. Note that it is not unusual for page sizes to vary even among implementations of the same instruction set, increasing the importance of using this function for portability.

The **`mprotect(2)`** system call allows the calling process to change the access permissions associated with a mapping that it has established to a memory object. **`Mprotect`** has the effect of assigning protection *prot* to all pages in the range [*addr*, *addr* + *len*). The protection assigned cannot exceed the permissions allowed on the underlying object. For instance, a read-only mapping to a file that has been opened for read-only access cannot be set to be writable with **`mprotect`** (unless the mapping is of the **MAP_PRIVATE** type, in which case the write access is permitted because the writes will modify copies of pages from the object, and not the object itself).

A process may not change the permissions associated with a mapping that has been established by another process.

The specifications for making the **`mprotect`** call are as follows:

```
#include <sys/mman.h>

int mprotect(addr, len, prot)

void *addr;
size_t len;
int prot;
```

The arguments are defined as follows:

addr	the starting address of the mapping for which the permissions are to be changed
len	the length in bytes of the mapping for which the permissions are to be changed
prot	an integer value that specifies one or more of the following options and determines the access permissions to be associated with the mapped data. This value is either

PROT_NONE	permits no access to the data

or

the bitwise inclusive OR of one or more of the following:

PROT_READ	permits read access to the data
PROT_WRITE	permits write access to the data

PROT_EXEC	permits execute access to the data

Generally a process should not attempt to read, write, or execute data for which the corresponding permission has not been granted. If **PROT_NONE** is specified, any attempt to access the data will fail; a `SIGSEGV` signal will be sent to the process. If **PROT_WRITE** is not specified, any attempt to write to the data will fail; a `SIGSEGV` signal will be sent to the process. If **PROT_WRITE** is specified and the mapping has been established through a call to **`mmap(2)`** with the **MAP_SHARED** option specified, the underlying memory object must have been opened with write permission.

It is important to note that on Model 6800 systems, the caches may not be coherent if either of the following occurs:

- Only **PROT_READ** and **PROT_EXEC** are specified, and a process attempts to read the data. In this case, the data cache will not be coherent with memory.
- **PROT_WRITE** and **PROT_EXEC** are specified, and a process attempts to execute the data. In this case, the instruction cache may not be coherent with memory.

If you wish to modify data and then execute the modified data, use the following steps: (1) map the data with the *prot* option set to **PROT_WRITE**; (2) modify the data; (3) change the permissions associated with the data by invoking **`mprotect`** with the *prot* option set to **PROT_EXEC**; and (4) execute the data.

A return value of **0** indicates that the call to **`mprotect`** has been successful. A return value of **-1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`mprotect(2)`** system manual page for a listing of the types of errors that may occur.

Address Space Layout

Traditionally, the address space of a UNIX process has consisted of exactly three segments: one each for write-protected program code (text), a heap of dynamically allocated storage (data), and the process's stack. Text is read-only and shared, while the data and stack segments are private to the process.

The OS still uses text, data, and stack segments although these should be thought of as constructs provided by the programming environment rather than by the operating system. As such, it is possible to construct processes that have multiple segments of each type, or of types of arbitrary semantic value—no longer are programs restricted to being built only from objects the system was capable of representing directly. For instance, a process's address space may contain multiple text and data segments, some belonging to specific programs and some shared among multiple programs. Text segments from shared libraries, for example, typically appear in the address spaces of many processes. A process's address space is simply a vector of pages, and there is no necessary division between different address-space segments. Process text and data spaces are simply groups of pages mapped in ways appropriate to the function they provide the program.

While the system may have multiple areas that can be considered data segments, for programming convenience, the system maintains operations to operate on an area of storage associated with a process's initial heap storage area. A process can manipulate this area by calling **`brk(2)`** and **`sbrk(2)`**:

```
caddr_t
brk(caddr_t addr);

caddr_t
sbrk(int incr);
```

The **`brk`** system call sets the system's idea of the lowest data segment location not used by the caller to *addr* (rounded up to the next multiple of the system's page size).

The alternate function, **`sbrk`** adds *incr* bytes to the caller's data space and returns a pointer to the start of the new data area.

A process's address space is usually sparsely populated, with data and text pages intermingled. The precise mechanics of the management of stack space is machine-dependent. By convention, page 0 is not used. Process address spaces are often constructed through dynamic linking when a program is **`exec'`**ed. Operations such as **`exec`** and dynamic linking build upon the mapping operations described previously. Dynamic linking is described further in the *Concurrent C Reference Manual*.

Managing Misaligned Data

This section describes the data alignment requirements of Series 6000 systems. It explains the procedures for handling misaligned data exceptions on those systems and describes the underlying implementation details.

Alignment

On Series 6000 systems, data must align according to the following requirements:

Table 6-1. Byte Alignment Requirements

Data Type	Byte Alignment
half-word	2
word	4
single-precision float	4
double-precision float	8

Programs that try to access data not aligned according to these requirements cause a Series 6000 system to generate a bus error.

Certain coding practices can cause misaligned data exceptions. In a FORTRAN application, for example, `EQUIVALENCE` statements and `COMMON` blocks can be used to override the compiler's normal alignment procedure. If a double-precision variable is forced to be aligned on a boundary that is not a multiple of eight, a misaligned data exception occurs.

To ensure that a program can access misaligned data without aborting, you must make provisions for handling misaligned data exceptions in your program. The following section explains procedures for handling these exceptions.

Exceptions

Handle misaligned data exceptions on a Series 6000 system using one of the following methods:

- Write a signal-handling routine to deal with the `SIGBUS` signal.
- Compile and link the source program with the **`misalign.o`** object designed to handle such exceptions by silently emulating the access.

`misalign.o` provides a signal handler to decide if the exception is a misaligned data exception. If it is, the handler works around the problem by accessing the data byte by byte at a significant cost in performance.

To compile and link a C or FORTRAN source program with **`misalign.o`**, specify its name when invoking the compiler. Use the following examples for command line instructions with C and FORTRAN programs:

```
cc -p source_file.c /lib/misalign.o
f77 source_file.f /lib/misalign.o
```

The following section presents implementation details related to the system object files.

Implementation

`misalign.o` deals with misaligned data exceptions by using the C global variable `_handle_misaligned_accesses` and the C routines **`_misalign_init()`** and **`_sigbus_handler()`**. The C start-up routines **`crt0.o`**, **`mcrt0.o`**, and **`gcrt0.o`**, check the value of the variable `_handle_misaligned_accesses`. If the value is non-zero, the start-up routines call the **`_misalign_init()`** routine. This routine then makes the following **`sigaction(2)`** system call to specify the address of a structure that identifies a signal-handling routine to deal with the `SIGBUS` signal:

```
sigaction (SIGBUS, &action, 0)
```

(See the **`sigaction(2)`** system manual page and "The sigaction Structure" on page 10-9 for information on using this call.)

The **`_sigbus_handler()`** routine queries the `SIGBUS` signal and emulates the misaligned data access. To handle misaligned data exceptions, the signal handler must deal with the `SIGBUS` signal and recognize that the exception block argument is `BUS_ADRALN`.

NOTE

User programs can replace system-provided versions of variables and functions by naming them identically. This keeps the system-provided versions from linking into the executable modules.

7
Terminal Device Control

7
Terminal Device Control

Introduction

This chapter discusses the general terminal interface to control asynchronous communication ports. The functions on the **`termio(7)`** manual page are used to access and configure the hardware interface to a terminal.

Also included in this chapter is a discussion of the mechanisms involved with opening and closing a terminal device file, as well as input/output processing.

The remainder of this chapter addresses the STREAMS mechanism as it relates to terminal device control. The STREAMS-based terminal subsystem provides a uniform interface for implementing character I/O devices and networking protocols in the kernel. Also discussed here is the notion of the STREAMS-based pseudo-terminal subsystem which provides the user with an identical interface to the STREAMS-based terminal subsystem.

Terminal Device Control Functions

General Terminal Interface

Terminal Device Control functions offer a general terminal interface for controlling asynchronous communication-ports in a device-independent manner using parameters stored in the `termios` structure which is defined by the **`<termios.h>`** header file (see **`termios(7)`**). The OS also uses `termios` to control the operation of network-connections. Table 7-1 gives an overview of this interface:

Table 7-1. Terminal Device Control Functions

Feature/Function Description	Interface
General Terminal Characteristics	
- get output baud-rate	**`cfgetospeed`**
- set output baud-rate	**`cfsetospeed`**
- get input baud-rate	**`cfgetispeed`**
- set input baud-rate	**`cfsetispeed`**

Table 7-1. Terminal Device Control Functions (Cont.)

General Terminal Control Functions	
- get state of terminal	**tcgetattr**
- set state of terminal	**tcsetattr**
- line control function	**tcsendbreak**
- line control function	**tcdrain**
- line control function	**tcflush**
- line control function	**tcflow**
- get foreground process-group-id	**tcgetpgrp**
- set foreground process-group-id	**tcsetpgrp**

The `termios` structure stores the values of settable terminal I/O parameters used by functions to control terminal I/O characteristics and the operation of a terminal-device-file. The **`<termios.h>`** header file defines the `termios` structure to contain at least the following members (see **`termios(7)`**):

```
tcflag_t  c_iflag;      /* input modes */
tcflag_t  c_oflag;      /* output modes */
tcflag_t  c_cflag;      /* control modes */
tcflag_t  c_lflag;      /* local modes */
cc_t      c_cc[NCCS];   /* control chars */
```

The **`<termios.h>`** header file defines the type `tcflag_t` as `long`, the type `cc_t` as `char`. The **`<termios.h>`** header file also defines the symbolic-constant `NCCS` as the size of the control-character array.

Baud Rates

The structure `termios` stores the input and output baud-rates in `c_cflag`. Table 7-2 below shows symbolic names defined in **`<termios.h>`** and the baud-rate each represents:

Table 7-2. Baud-Rates Definitions

Symbolic Constant	Associated baud-rate
`B0`	hang up
`B50`	50 baud
`B75`	75 baud
`B110`	110 baud
`B134`	134.5 baud
`B150`	150 baud
`B200`	200 baud

Table 7-2. Baud-Rates Definitions (Cont.)

Symbolic Constant	Associated baud-rate
`B300`	300 baud
`B600`	600 baud
`B1200`	1200 baud
`B1800`	1800 baud
`B2400`	2400 baud
`B4800`	4800 baud
`B9600`	9600 baud
`B19200`	19200 baud
`B38400`	38400 baud

Note that the zero baud-rate, `B0`, is used to terminate the connection. If `B0` is specified, the modem control lines are no longer asserted; normally, this disconnects the line (see **`cfsetospeed(2)`** and **`tcsetattr(2)`**):

The `termios` structure members `c_iflag`, `c_oflag`, `c_cflag` and `c_lflag` take as values the bitwise inclusive-OR of bitwise distinct masks with symbolic names defined by the **`<termios.h>`** header file (see **`termios(7)`**).

Input Modes

The input-modes field `c_iflag` specifies treatment of terminal input. Calling **`read`** on a terminal-device-file works as described in “Input Processing and Reading Data” and the value of `c_iflag` along with the value of `c_lflag` determine how to process input read from the terminal (see **`termios(7)`**).

Output Modes

The output-modes field `c_oflag` specifies treatment of terminal output. Calling **`write`** on a terminal-device-file works as described in “Writing Data and Output Processing” and the value of `c_oflag` determines how to process output written to the terminal (see **`termios(7)`**).

Control Modes

The control-modes field `c_cflag` specifies communication control for terminals. The value of `c_cflag` controls characteristics of the communications-port to a terminal-device, but the underlying hardware may fail to support all `c_cflag` values (see **`termios(7)`**). A communication-port other than an asynchronous serial connection may ignore some of the control-modes; for example, if an attempt is made to set the baud-rate on a network-connection to a terminal on another host, the baud-rate may or may not be set on the connection between the terminal and the machine it is directly connected to.

Local Modes and Line Disciplines

The local-modes field `c_lflag` specifies the *line-discipline* for the terminal. The line-discipline works as described in “Canonical Mode Input Processing” and “Non-Canonical Mode Input Processing” and the value of `c_lflag` along with the value of `c_iflag` determine how the line-discipline acts on input from a terminal-device-file (see **`termios(7)`**).

Special Control Characters

The array `c_cc` specifies the special control-characters that affect the operation of the communication-port and the processing of terminal input and output as described in the “Special Characters” section below. For each entry of the control-character array `c_cc`, the following are typical default values:

Table 7-3. Terminal Device Control Character Array

Subscript Value	Subscript Name	Character Value	Character Description
0	VINTR	ASCII DEL	INTR character
1	VQUIT	ASCII FS	QUIT character
2	VERASE	#	ERASE character
3	VKILL	@	KILL character
4	VEOF	ASCII EOT	EOF character
5	VEOL	ASCII NUL	EOL character
6	reserved		
7	reserved		
8	VSTART	ASCII DC1	START character
9	VSTOP	ASCII DC3	STOP character
10	VSUSP	ASCII SUB	SUSP character

The subscript values are unique, except that the `VMIN` and `VTIME` subscripts may have the same value as the `VEOF` and `VEOL` subscripts respectively. The **`<termios.h>`** header file defines the relative positions, subscript names and default values for the control-character array `c_cc` (see **`termios(7)`**).

The `NL` and `CR` character cannot be changed. The `INTR,QUIT,ERASE, KILL,EOF, EOL, SUSP, STOP` and `START` characters can be changed as follows:

```
struct termios term;

term.c_cc[VINTR]    = 'a';
term.c_cc[VQUIT]    = 'b';
term.c_cc[VERASE]   = 'c';
term.c_cc[VKILL]    = 'd';
```

```
    term.c_cc[VEOF]      = 'e';
    term.c_cc[VEOL]      = 'f';
    term.c_cc[VSUSP]     = 'g';
    term.c_cc[VSTOP]     = 'h';
    term.c_cc[VSTART]    = 'i';
```

where *a, b, c, d, e, f, g, h* and *i* are the `INTR, QUIT, ERASE, KILL, EOF, EOL, SUSP, STOP` and `START` characters respectively.

Implementations which prohibit changing the `START` and `STOP` characters may ignore the character values in the `c_cc` array indexed by the `VSTART` and `VSTOP` subscripts when **`tcsetattr`** is called, but return the character value when **`tcsetattr`** is called (see **`tcsetattr(2)`**).

If `_POSIX_VDISABLE` is defined for the terminal-device-file, and the value of one of the changeable special control-characters equals `_POSIX_VDISABLE`, that function is disabled; that is, the special character is ignored on input and is not recognized (see "Special Characters" section below). If `ICANON` is clear, the value of `_POSIX_VDISABLE` lacks any special meaning for the `VMIN` and `VTIME` entries of the `c_cc` array.

Opening a Terminal Device File

When a terminal-device-file is opened, it normally causes the process to wait until a connection is established. In practice, application-programs seldom open such files; instead, at system-initialization time special-programs open terminal-device-files as the *standard input*, *standard output* and *standard error* files (see **`stdio(4)`**).

Opening a terminal-device-file with the flag `O_NONBLOCK` clear on the **`open`** system call causes the process to block until the terminal-device is ready and available (see **`open(2)`**). The flag `CLOCAL` can also affect the **`open`** system call (see **`termios(7)`**).

Input Processing and Reading Data

A terminal-device accessed through an open terminal-device-file ordinarily operates in full-duplex mode. This means data may arrive at any time, even while output is occurring. Each terminal-device-file has associated with it an "input-queue," into which the system stores incoming data before the process reads that data. The system imposes a limit of `MAX_INPUT`, the maximum allowable number of bytes of input data, on the number of bytes of data that it stores in the input-queue. Data is lost only when the input-queue becomes completely full, or when an input line exceeds `MAX_INPUT`. The behavior of the system when this limit is exceeded is implementation-dependent.

In the OS, if the data in the terminal-device-file input-queue exceeds `MAX_INPUT` and `IMAXBEL` is clear, all the bytes of data saved up to that point are discarded without any notice, but if `IMAXBEL` is set and the data in the terminal-device-file input-queue exceeds `MAX_INPUT`, the ASCII BEL character is echoed. Further input is not stored, and any data already present in the input-queue remains undisturbed.

Two general kinds of input processing are available, determined by whether the terminal-device-file is operating in canonical mode or non-canonical mode. These modes are described in “Canonical Mode Input Processing” and “Non-Canonical Mode Input Processing”. Additionally, input is processed according to the `c_iflag` and `c_lflag` fields (see **`termios(7)`**). Such processing can include echoing, which in general means transmitting input data bytes immediately back to the terminal when they are received from the terminal. This is useful for terminals that can operate in full-duplex mode.

The way a process reading from a terminal-device-file gets data depends on whether the terminal-device-file is operating in canonical mode or non-canonical mode. How **`read`** operates on a terminal-device-file also depends on how **`open(2)`** or **`fcntl(2)`** sets the flag `O_NONBLOCK` for the file:

- If `O_NONBLOCK` and `O_NDELAY` are clear, **`read`** blocks until data is available or a signal interrupts the **`read`** operation.
- If `O_NONBLOCK` is set, **`read`** completes, without blocking, in one of the following three ways:
    1. If enough bytes of data are available to satisfy the entire request, **`read`** completes successfully and returns the number of bytes it transferred.
    2. If too few bytes of data are available to satisfy the entire request, **`read`** completes successfully, having transferred as much data as it could, and returns the number of bytes it actually transferred.
    3. If *no* data is available, **`read`** returns `-1` and `errno` equals `EAGAIN`.

When data become available depends on whether the input-processing mode is canonical or non-canonical. The following sections, “Canonical Mode Input Processing” and “Non-Canonical Mode Input Processing,”describe each of these input-processing modes.

Canonical Mode Input Processing

In canonical mode input processing, terminal input is processed in units of lines. A line is delimited by the new-line (`'\n'`) character, end-of-file (`EOF`) character or end-of-line (`EOL`) character (see “Special Characters”section below for more information on `EOF` and `EOL`).

Processing terminal input in units of lines means that a program attempting a **`read`** from a terminal-device-file is suspended until an entire line is typed, or a signal is received. Also, no matter how many bytes of data a **`read`** may request from a terminal-device-file, it transfers at most one line of input. It is not, however, necessary to read the entire line at once; a **`read`** may request any number of bytes of data, even one, without losing any data remaining in the line of input.

If `MAX_CANON` is defined for this terminal-device, it is a limit on the number of bytes in a line. The behavior of the system when this limit is exceeded is implementation-dependent. If `MAX_CANON` is not defined for this terminal-device, there is no such limit.

It should be noted that there is a possible inherent deadlock if the program and the implementation conflict on the value of `MAX_CANON`. With both `ICANON` and `IXOFF` set when more than `MAX_CANON` characters transmitted without a line-feed, transmission is stopped, the line-feed (or carriage-return if `ICRLF` is set) never arrives, and the **`read`** is never satisfied.

A program should never set IXOFF if it is using canonical-mode unless it knows that (even in the face of a transmission error) the conditions described previously cannot be met or unless it is prepared to deal with the possible deadlock in some other way, such as time-outs.

NOTE

This would only occur if the transmitting side was a communications device (for example, an asynchronous port). This normally will not happen since the transmitting side is a user at a terminal.

It should also be noted that this can be made to happen in non-canonical-mode if the number of characters received that would cause IXOFF to be sent is less than VMIN when VTIME equals zero.

With the OS, if the data in the line-discipline buffer exceeds MAX_CANON in canonical mode and IMAXBEL is clear, all the bytes of data saved in the buffer up to that point are discarded without any notice, but if IMAXBEL is set and the data in the line-discipline buffer exceeds MAX_INPUT, the ASCII BEL character is echoed. Further input is not stored, and any data already present in the input-queue remains undisturbed.

During input, *erase* and *kill* processing occurs whenever either of two special characters, the ERASE and KILL characters is received (see “Special Characters”). This processing affects data in the input-queue that has yet to be delimited by a new-line, EOF or EOL character. This un-delimited data makes up the current line. The ERASE character deletes the last character (if any) in the current line; it does not erase beyond the beginning of the line. The KILL character deletes all data (if any) in the current line; it optionally outputs a new-line character. The ERASE and KILL characters have no effect if the current line lacks any data.

Both the ERASE and KILL characters operate on a key-stroke basis independently of any backspacing or tabbing. Typically, # is the default ERASE character, and @ is the default KILL character. The ERASE and KILL characters themselves are not placed in the input-queue.

Non-Canonical Mode Input Processing

In non-canonical input processing, input bytes are not assembled into lines, and erase and kill processing does not occur. The values of the MIN and TIME members of the c_cc array determine how to process any data received.

MIN is the minimum number of bytes of data that a **read** should return when it completes successfully. If MIN exceeds MAX_INPUT, the response to the request is implementation-defined. With the OS, the maximum value that can be stored for MIN in c_cc[VMIN] is 256, less than MAX_INPUT which equals 512; thus, the MIN value can never exceed MAX_INPUT. TIME is a read-timer with a 0.10 second granularity used to time-out burst and short-term data transmissions. The four possible interactions between MIN and TIME follow:

1. (MIN>0, TIME>0).

Because TIME>0, it serves as an inter-byte timer activated on receipt of the first byte of data, and reset on receipt of each byte of data. MIN and TIME interact as follows:

- As soon as a byte of data is received, the inter-byte timer starts (remember that the timer is reset on receipt of each byte)
- If MIN bytes of data are received before the inter-byte timer expires, the **read** completes successfully.
- If the inter-byte timer expires before MIN bytes of data are received, the **read** transfers any bytes received up until then.

When TIME expires, a **read** transfers at least one byte of data because the inter-byte timer is enabled if and only if a byte of data was received. A program using this case must wait for at least one byte of data to be read before proceeding. In case (MIN>0, TIME>0), a **read** blocks until receiving a byte of data activates MIN and TIME, or a signal interrupts the **read**. Thus, the **read** transfers at least one byte of data.

2. (MIN>0, TIME=0).

   Because TIME=0, the timer plays no role and only MIN is significant. A **read** completes successfully only on receiving MIN bytes of data (i.e., the pending **read** blocks until MIN bytes of data are received) or a signal interrupts the **read**. Use these values only when the program cannot continue until a predetermined number of bytes of data are read. A program using this case to do record-based terminal I/O may block indefinitely in a **read**.

3. (MIN=0, TIME>0).

   Because MIN=0, TIME no longer serves as an inter-byte timer, but now serves as a read-timer activated when a **read** is processed (in canon). A **read** completes successfully as soon as any bytes of data are received or the read-timer expires. A **read** does not transfer any bytes of data if the read-timer expires. If the read-timer does not expire, a **read** completes successfully if and only if some bytes of data are received. In case (MIN=0, TIME>0), the **read** does not block indefinitely waiting for a byte of data. If no bytes of data are received within TIME*0.10 seconds after the **read** starts, it returns 0 having read no data. If the buffer holds data when a **read** starts, the read-timer starts as if it received data immediately. MIN and TIME are useful when a program can assume that data is not available after a TIME interval and other processing can be done before data is available.

4. (MIN=0, TIME=0).

   Without waiting for more bytes of data to be received, a **read** returns the minimum of either the number of bytes of data requested or the number of bytes of data currently available. In this case, a **read** immediately transfers any bytes of data present, or if no bytes of data are available, it returns 0 having read no data. In case (MIN=0, TIME=0), **read** operates identically to the O_NDELAY flag in canonical mode.

MIN/TIME interactions serve different purposes and thus do not parallel one another. In case [2]: (MIN>0, TIME=0), TIME lacks effect, but with the conditions reversed in case [3]: (MIN=0, TIME>0), both MIN and TIME play a role in that receiving a single byte satisfies the `MIN` criteria. Furthermore, in case [3]: (`MIN=0`, `TIME>0`), `TIME` represents a read-timer, while in case [1]: (`MIN>0`, `TIME>0`), `TIME` represents an inter-byte timer,

Cases [1] and [2], where `MIN>0`, handle burst mode activity (e.g., file-transfers), where programs need to process at least `MIN` bytes of data at a `time`. In case [1], the inter-byte timer acts as a safety measure; in case [2], the timer is turned off.

Cases [3] and [4] handle single byte, timed transfers like those used by screen-based programs that need to know if a byte of data is present in the input-queue before refreshing the screen. In case [3], the **`read`** is timed, while in case [4], it is not.

One should also note that `MIN` is always just a minimum, and does not define a record length. Thus, if a program tries a **`read`** of 20 bytes when 25 bytes of data are present and `MIN` is 10, the **`read`** returns 20 bytes of data. In the special case of `MIN=0`, this still applies: if more than one byte of data is available, all data is returned immediately.

Writing Data and Output Processing

When a process writes data onto a terminal-device-file, `c_oflag` controls how to process those bytes (see **`termios(7)`**). The OS provides buffering such that a call to **`write`** schedules data for transfer to the device, but has not necessarily completed the transfer when the call returns (see **`write(2)`** for the effects of `O_NONBLOCK` on **`write`**).

Closing a Terminal Device File

The last process to close a terminal-device-file causes any output remaining to be sent to the device and any input remaining to be discarded. Following these actions, if the flag `HUPCL` is set in the control-modes and the communication-port supports a disconnect function, the terminal-device does a disconnect.

Because the POSIX.1 standard is silent on whether a **`close`** blocks waiting for transmission to drain, or even if a **`close`** might flush any pending output, a program concerned about how data in terminal input and output-queues are handled should call the appropriate functions such as **`tcdrain`** to ensure the desired behavior (see **`close(2)`** and **`tcdrain(2)`**).

Special Characters

Certain characters have special functions on input or output or both. These functions and their typical default character values are summarized below:

`INTR` (typically, rubout or ASCII DEL) sends an *interrupt* signal, `SIGINT`, to all processes in the foreground process-group for which the terminal is the controlling-terminal. Receiving the signal `SIGINT` normally forces a process to terminate, but a process may arrange to ignore the signal or to call a signal-catching function (see **`sigaction(2)`**).

If `ISIG` is set, the `INTR` character is recognized and acts as a special character on input and is discarded when processed (see **`termios(7)`**).

`QUIT`	(typically, `control-\` or ASCII FS) sends a *quit* signal, `SIGQUIT`, to all processes in the foreground process-group for which the terminal is the controlling-terminal. Receiving the signal `SIGQUIT` normally forces a process to terminate just as the signal `SIGINT` does except that, unless a receiving process makes other arrangements, it not only terminates but a core image file (called `CORE`) will be created in the current working directory of the process (see **`sigaction(2)`**). If `ISIG` is set, the `QUIT` character is recognized and acts as a special character on input and is discarded when processed (see **`termios(7)`**).
`ERASE`	(typically, the character #) erases the most recently input character in the current line (see “Canonical Mode Input Processing”). It does not erase beyond the start of a line. If `ICANON` is set, the `ERASE` character is recognized and acts as a special character on input and is discarded when processed (see **`termios(7)`**).
`KILL`	(typically, the character `@`) deletes the entire line, as delimited by an EOF, `EOL` or `NL` character. If `ICANON` is set, the `KILL` character is recognized and acts as a special character on input and is discarded when processed (see **`termios(7)`**).
`EOF`	(typically, `control-d` or ASCII EOT) generates an `EOF`, from a terminal. On receiving `EOF`, a **`read`** immediately passes any bytes of data it holds to the process without waiting for a new-line, and discards the `EOF`. If `EOF` occurred at the beginning of a line, a **`read`** holds no bytes of data, and returns a byte count of zero, the standard end-of-file indication. If `ICANON` is set, the `EOF` character is recognized and acts as a special character on input and is discarded when processed (see **`termios(7)`**).
`NL`	(ASCII LF) is the normal line delimiter, (`'\n'`), which can not be changed or escaped. If `ICANON` is set, the `NL` character is recognized and acts as a special character on input (see **`termios(7)`**).
`EOL`	(typically, ASCII NUL) is an additional line delimiter, like the `NL` character. `EOL` is not normally used. If `ICANON` is set, the `EOL` character is recognized and acts as a special character on input (see **`termios(7)`**).
`SUSP`	(typically, `control-d` or ASCII SUB) sends an *stop* signal, `SIGTSTP`, to all processes in the foreground process-group for which the terminal is the controlling-terminal. If job-control is supported and `ISIG` is set, the `SUSP` character is recognized and acts as a special character on input and is discarded when processed (see **`termios(7)`**).

`STOP` (typically, `control-s` or ASCII DC3) temporarily suspends output. It is useful with CRT terminals to prevent output from disappearing before it can be seen. While output is suspended, `STOP` characters are ignored not read. The `STOP` character can be changed through the `c_cc` array (see **`termios(7)`**).

If `IXON` (output control) is set or `IXOFF` (input control) is set, the `STOP` character is recognized and acts as a special character on both input and output. If `IXON` is set, the `STOP` character is discarded when processed (see **`termios(7)`**).

`START` (typically, `control-q` or ASCII DC1) resumes output suspended by a `STOP` character. While output is not suspended, `START` characters are ignored and not read. The `START` character can be changed through the `c_cc` array (see **`termios(7)`**).

If `IXON` (output control) is set or `IXOFF` (input control) is set, the `START` character is recognized and acts as a special character on both input and output. If `IXON` is set, the `START` character is discarded when processed (see **`termios(7)`**).

`CR` (ASCII CR) is a line delimiter, (`'\r'`), which is translated into the NL character, and it has the same effect as the `NL` character if `ICANON` and `ICRNL` are set and `IGNCR` is clear.

If `ICANON` is set, the `NL` character is recognized and acts as a special character on input (see **`termios(7)`**).

`MIN` controls terminal I/O during raw mode (`ICANON` off) processing (see “Canonical Mode Input Processing”).

`TIME` controls terminal I/O during raw mode (`ICANON` off) processing (see “Non-Canonical Mode Input Processing”).

The `NL` and `CR` character cannot be changed. The `INTR, QUIT, ERASE, KILL, EOF, EOL, SUSP, STOP` and `START` characters can be changed through the `c_cc` array (see **`termios(7)`**).

The `ERASE`, `KILL` and `EOF` characters may be entered literally (their special meaning escaped) by preceding them with the escape character (`'\'`). In this case, no special function is done and the escape character is not read as input.

The Controlling-Terminal and Process-Groups

A terminal may belong to a process as its *controlling-terminal*, which is a terminal uniquely associated with one session. Each process of a session with a controlling-terminal has the same controlling-terminal assigned to it. Each session may have at most one controlling-terminal associated with it and vice versa. A terminal may be assigned to at most one session as the controlling-terminal. Certain input sequences from the controlling-terminal cause signals to be sent to all processes in the process-group for the controlling-terminal (see **`termios(7)`**). The controlling-terminal plays a special role in handling *quit* and *interrupt* signals (see “Special Characters”).

The controlling-terminal for a session is acquired by the *session-leader*, which is the process that created the session; the *session-id* of a session equals the process-id of the session-leader. When a session-leader acquires a controlling-terminal for its session, it thereby becomes the c*ontrolling-process* of that session (see **`setsid(2)`**). Should the terminal later cease to be a controlling-terminal for the session of the session-leader, the session-leader ceases to be a controlling-process.

When a session-leader without a controlling-terminal opens a terminal-device-file and the flag `O_NOCTTY` is clear on **`open`**, that terminal becomes the controlling-terminal assigned to the session-leader if the terminal is not already assigned to some session (see **`open(2)`**). When any process other than a session-leader opens a terminal-device-file, or the flag `O_NOCTTY` is set on **`open`**, that terminal does not become the controlling-terminal assigned to the calling-process.

A controlling-terminal distinguishes one of the process-groups in the session assigned to it as the *foreground* process-group; all other process-groups in the session are *background* process-groups. By default, when the session-leader acquires a controlling-terminal, the process-group of the session-leader becomes the foreground process-group of the controlling-terminal. The foreground process-group plays a special role in handling signal-generating input characters (see “Special Characters” above).

A new process inherits the controlling-terminal through the **`fork`** operation (see **`fork(2)`**). When a process calls **`setsid`** to create a new session, the process relinquishes its controlling-terminal; other processes remaining in the old session with that terminal as their controlling-terminal continue to have it (see **`setsid(2)`**). When all file-descriptors that denote the controlling-terminal in the system are closed (whether or not it is in the current session), it is unspecified whether all processes that had that terminal as their controlling-terminal cease to have any controlling-terminal. Whether and how a session-leader can reacquire a controlling-terminal after the controlling-terminal is relinquished in this fashion is unspecified. A process does not relinquish its controlling-terminal simply by closing all of its file-descriptors that denote the controlling-terminal if other processes continue to have it open.

When a session-leader terminates, the current session relinquishes the controlling-terminal allowing a new session-leader to acquire it. Any further attempts to access the terminal by other processes in the old session may be denied and treated as if modem-disconnect was detected on the terminal.

Session Management and Job Control

If `_POSIX_JOB_CONTROL` is defined, the OS supports job-control and command interpreter processes supporting job-control can assign the terminal to different jobs, or process-groups, by placing related processes in a single process-group and assigning the process-group with the terminal. A process may examine or change the foreground process-group of a terminal assuming the process has the required permissions (see **`tcgetpgrp(2)`** and **`tcsetpgrp(2)`**). The `termios` facility aids in this assignment by restricting access to the terminal by processes outside of the foreground process-group (see Chapter 10. “Signals, Job Control, and Pipes” in this guide).

When there is no longer any process whose process-id or process-group-id matches the process-group-id of the foreground process-group, the terminal lacks any foreground process-group. It is unspecified whether the terminal has a foreground process-group when there is no longer any process whose process-group-id matches the process-group-id of the foreground process-group, but there is a process whose process-id matches the process-group-id of the foreground process-group. Only a successful call to **`tcsetpgrp`** or assignment of the controlling-terminal as described can make a process-group the foreground process-group of a terminal (see **`tcsetpgrp(2)`**).

Background process-groups in the session of the session-leader are subject to a job-control line-discipline when they attempt to access their controlling-terminal. Typically, they are sent a signal that causes them to stop, unless they have made other arrangements (see **`signal(4)`**). An exception is made for processes that belong to a orphaned process-group, which is a process-group none of whose members have a parent in another process-group within the same session and thus share the same controlling-terminal. When these processes attempt to access their controlling-terminal, they return errors, because there is no process to continue them if they should stop (see chapter “Signals, Job Control, and Pipes” in this guide).

Improving Terminal I/O Performance

For user-level programs that read and write to terminals, the TTY subsystem the OS provides a flexible interface, known as the `termio` facility. The flexibility of the `termio` facility enables users to perform efficient TTY I/O in a wide range of applications. However, the improper use of this `termio` can result in inefficient user programs. This section discusses writing programs that use `termio` and focuses on the topics of buffer size, canonical mode, raw mode and flow control and provides several code examples.

User programs that read from terminal devices must read from TTYs in either canonical mode or raw mode.

TTY in Canonical Mode

In canonical mode, characters are read from the device and processed before being returned. This processing translates kill and erase characters. Characters are not returned until a new line (`NL`), end of file (`EOF`), or end of line (`EOL`) is read, which means that characters are returned a line at a time. Canonical mode is usually associated with terminals.

An important factor to consider when using canonical mode is what to do when reading from a TTY device for which characters are not available. If the `O_NDELAY` flag has been set for the TTY, then such **`read`**s return a `0`, indicating that no characters are available. Otherwise, **`read`**s will not return until a character is available. If a program can perform other processing when characters are not available from a TTY, then the `O_NDELAY` flag should be set for the TTY. This might require programs to be more complicated, but the complication are offset by an increase in efficiency.

The following function shown in Screen 7-1 opens a TTY device for reading or writing (line 12), places it in canonical mode (line 23), and sets the `O_NDELAY` option so that **`read`**s are not blocked when characters are not available (line 12).

```
#include <fcntl.h>
#include <termio.h>

extern struct termio old_term;

setup1(TTY)
char *TTY;
{
      int fid;
      struct termio new_term;

      if ((fid = open(TTY, O_RDWR|O_NDELAY)) == -1)
      {
              printf("open failed.\n");
              exit(1);
      }
              else if (ioctl(fid, TCGETA, &old_term) == -1)
                   {
                      printf("ioctl get failed.\n");
                      exit(1);
                   }
      new_term = old_term;
      new_term.c_lflag |= ICANON;
      if (ioctl(fid, TCSETA, &new_term) == -1)
      {
              printf("ioctl set failed.\n");
              exit(1);
      }
      return fid;
}
```

Screen 7-1. Improving TTY Performance Canonical Mode

TTY in Raw Mode

In raw mode, characters are read and returned as is; that is, without being processed. Reading from a TTY device in raw mode is faster than reading from a TTY device in canonical mode. In the interest of efficiency, raw mode should be used when characters do not need to be canonically processed.

Just as in canonical mode, TTY devices that are in raw mode must deal with the problem of what to do when reading from a device for which characters are not available. The `O_NDELAY` flag only applies to TTY devices that are in canonical mode. The same function is provided by the `MIN` and `TIME` values for raw TTY devices. By choosing appropriate values of `MIN` and `TIME`, a programmer can help maximize efficiency when reading from TTY devices in raw mode.

The following function shown in Screen 7-2 inputs a TTY that has previously been opened in raw mode and sets the `MIN` and `TIME` options to be `0` so that **`read`**s will not be blocked when characters are not available.

```
#include <termio.h>

extern struct termio old_term;

setup2(fid)
int fid;
{
      struct termio new_term;

     if (ioctl(fid, TCGETA, &old_term) == -1)
      {
              printf(“ioctl get failed.\n”);
              exit(1);
      }

      new_term = old_term;
      new_term.c_lflag &= ~ICANON;
      new_term.c_cc[VMIN] = 0;
      new_term.c_cc[VTIME] = 0;

      if (ioctl(fid, TCSETA, &new_term) == -1)
      {
              printf(“ioctl set failed.\n”);
              exit(1);
      }
}
```

Screen 7-2. Improving TTY Performance Raw Mode

TTY Flow Control

Flow control becomes a problem when a program that reads from a TTY device that cannot keep up with the number of characters that are coming into the TTY. If this happens, characters are over-written in the TTY input queue before they can be read by the program.

Conversely, when a program writes to a TTY, the device might not be able to keep up with the TTY. When this happens, characters that are written by a program to a TTY are not being seen by the appropriate device.

The **`termio`** facility provides a mechanism called software flow control to solve this problem. If a program cannot keep up with the characters coming into a TTY, the TTY sends a `STOP` character to the originator. The originator, upon receipt of the `STOP` character, stops sending characters to the TTY until it received a `START` character. The TTY sends the `START` character when the program has sufficiently emptied its input queue.

If a device cannot keep up with a TTY, the device sends a `STOP` character to the TTY. Upon receipt of the `STOP` character, the TTY stops sending characters to the terminal until it receives a `START` character. The terminal sends the `START` character when it has sufficiently emptied its input queue. The TTY then blocks writes to the TTY until the TTY's output has sufficiently emptied.

Three different options are provided for flow control: `IXON`, `IXOFF`, and `IXANY`. If `IXOFF` is set, then software flow control is enabled on the TTY's input queue. The TTY transmits a `STOP` character when the program cannot keep up with its input queue and transmits a `START` character when its input queue in nearly empty again.

If `IXON` is set, software flow control is enabled on the TTY's output queue. The TTY blocks writes by the program when the device to which it is connected cannot keep up with it. If `IXANY` is set, then any character received by the TTY from the device restarts the output that has been suspended.

The following function shown in Screen 7-3 sets the `IXANY`, `IXOFF`, and `IXANY` options for a TTY device that has previously been opened so that software flow control is enabled for both input and output.

```
#include <termio.h>

extern struct termio old_term;

setup3(fid)
int fid;
{
         struct termio new_term;

        if (ioctl(fid, TCGETA, &old_term) == -1)
        {
                printf(“ioctl get failed.\n”);
                exit(1);
        }

        new_term = old_term;
        new_term.c_iflag |= IXON | IXOFF | IXANY;

        if (ioctl(fid, TCSETA, &new_term) == -1)
        {
                printf(“ioctl set failed.\n”);
                exit(1);
        }
}
```

Screen 7-3. Improving TTY Performance Flow Control

When you design programs that read and write for the TTY subsystem, remember to address buffer size, canonical/raw mode and flow control concerns to ensure programming efficiency. For further information, see the following references:

- **`termio(7)`** in the *System Files and Devices Reference.*
- **`open(2)`**, **`read(2)`**, and **`ioctl(2)`** in the *Operating System API Reference.*
- **`termio(BA_ENV)`** in the *System V Interface Definition.*

STREAMS-Based Terminal Subsystem

The OS implements the terminal subsystem in STREAMS. The STREAMS-based terminal subsystem (see Figure 7-1) provides many benefits:

- Reusable line discipline modules. The same module can be used in many STREAMS where the configuration of these STREAMS may be different.

- Line discipline substitution. Although the OS provides a standard terminal line discipline module, another one conforming to the interface may be substituted. For example, a remote login feature may use the terminal subsystem line discipline module to provide a terminal interface to the user.
- Internationalization. The modularity and flexibility of the STREAMS-based terminal subsystem enables an easy implementation of a system that supports multiple byte characters for internationalization. This modularity also allows easy addition of new features to the terminal subsystem.
- Easy customizing. Users may customize their terminal subsystem environment by adding and removing modules of their choice.
- The pseudo-terminal subsystem. The pseudo-terminal subsystem can be easily supported.
- Merge with networking. By pushing a line discipline module on a network line, you can make the network look like a terminal line.

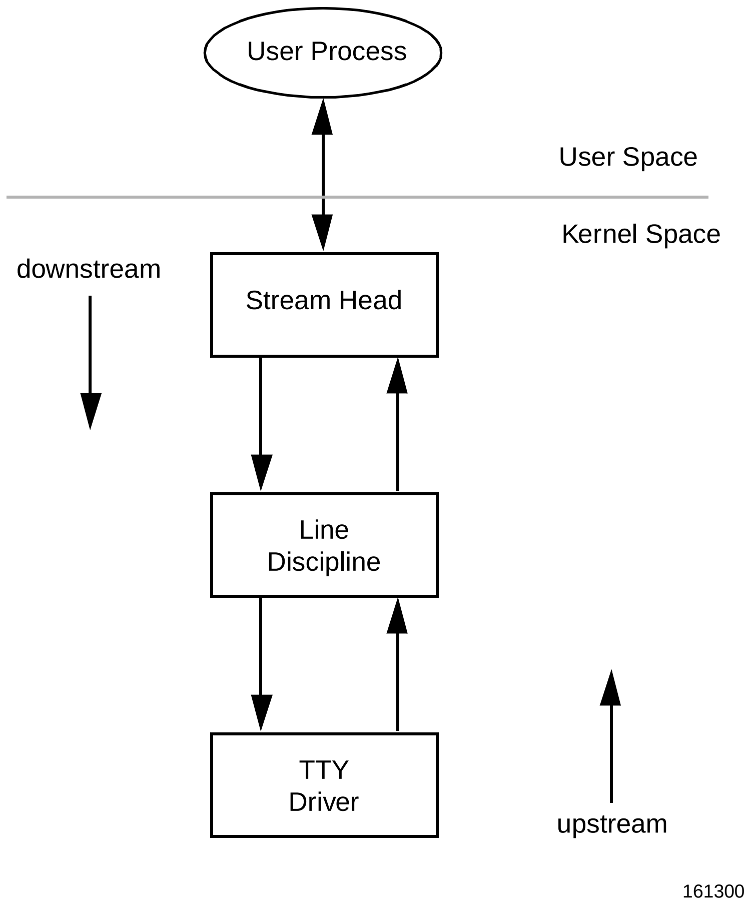


Figure 7-1. STREAMS-based Terminal Subsystem

The initial setup of the STREAMS-based terminal subsystem is handled with the **ttymon(1M)** command within the framework of the Service Access Facility (SAF) or the autopush facility.

The STREAMS-based terminal subsystem supports **termio**, the termios specification of the POSIX standard, multiple byte characters for internationalization, the interface to asynchronous hardware flow control and peripheral controllers for asynchronous terminals (see **termio(7)**, **termios(7)** and **termiox(7)**).

To use shl with the STREAMS-based terminal subsystem, the sxt driver is implemented as a STREAMS-based driver. However, the sxt feature is being phased out and users are encouraged to use the job control mechanism. Note that both shl and job control should not be run simultaneously.

Line Discipline Module

A STREAMS line discipline module called **ldterm** (see **ldterm(7)**) is a key part of the STREAMS-based terminal subsystem. Throughout this chapter, the terms "line discipline" and **ldterm** are used interchangeably and refer to the STREAMS version of the standard line discipline and not the traditional character version. **ldterm** performs the standard terminal I/O processing that was traditionally done through the linesw mechanism.

The **termio** and termios specifications describe four flags that are used to control the terminal: c_iflag (defines input modes), c_oflag (defines output modes), c_cflag (defines hardware control modes), and c_lflag (defines terminal functions used by **ldterm**). To process these flags elsewhere (for example, in the firmware or in another process), a mechanism is in place to turn on and off the processing of these flags. When **ldterm** is pushed, it sends an M_CTL message downstream, which asks the driver which flags the driver will process. The driver sends back that message in response if it needs to change **ldterm**'s default processing. By default, **ldterm** assumes that it must process all flags except c_cflag unless it receives a message telling otherwise.

Default Settings

When **ldterm** is pushed on the Stream, the open routine initializes the settings of the **termio** flags. The default settings are:

```
c_iflag = BRKINT|ICRNL|IXON|ISTRIP|IXANY
c_oflag = OPOST|ONLCR|TAB3
c_cflag = 0
c_lflag = ISIG|ICANON|ECHO|ECHOK
```

In canonical mode (ICANON flag in c_lflag is turned on), **read** from the terminal file descriptor is in message nondiscard (RMSGN) mode (see **streamio(7)**). This implies that in canonical mode, **read** on the terminal file descriptor always returns at most one line regardless of how many characters have been requested. In noncanonical mode, **read** is in byte-stream (RNORM) mode.

Open and Close Routines

The open routine of the **ldterm** module allocates space for holding state information.

The **ldterm** module establishes a controlling tty for the line when an M_SETOPTS message (so_flags is set to SO_ISTTY) is sent upstream. The Stream head allocates the controlling tty on the open, if one is not already allocated.

To maintain compatibility with existing application-programs that use the O_NDELAY flag, the **open** routine sets the SO_NDELON flag on in the so_flags field of the stroptions structure in the M_SETOPTS message.

The open routine fails if there is insufficient space for allocating the state structure, or when an interrupt occurs while the open is sleeping until memory becomes available.

The close routine frees all the outstanding buffers allocated by this Stream. It also sends an M_SETOPTS message to the Stream head to undo the changes made by the open routine. The **ldterm** module also sends M_START and M_STARTI messages downstream to undo the effect of any previous M_STOP and M_STOPI messages.

Read-Side Processing

The **ldterm** module's read-side processing has **put** and **service** procedures. **ldterm** can send the following messages upstream:

M_DATA, M_BREAK, M_PCSIG, M_SIG, M_FLUSH, M_ERROR, M_IOCACK, M_IOCNAK, M_HANGUP, M_CTL, M_SETOPTS, M_COPYOUT, and M_COPYIN.

The **ldterm** module's read-side processes M_BREAK, M_DATA, M_CTL, M_FLUSH, M_HANGUP, and M_IOCACK messages. All other messages are sent upstream unchanged.

The **put** procedure scans the message for flow control characters (IXON), signal generating characters, and after (possible) transformation of the message, queues the message for the **service** procedure. Echoing is handled completely by the **service** procedure.

In canonical mode if the ICANON flag is on in c_lflag, canonical processing is performed. If the ICANON flag is off, noncanonical processing is performed (see **termio(7)** for more details). Handling VMIN/VTIME in the STREAMS environment is somewhat complicated, because **read** needs to activate a timer in the **ldterm** module in some cases; hence, read notification becomes necessary. When a user issues an **ioctl** to put **ldterm** in noncanonical mode, the **ldterm** module sends an M_SETOPTS message to the Stream head to register read notification. Further reads on the terminal file descriptor causes the Stream head to issue an M_READ message downstream and data are sent upstream in response to the M_READ message. With read notification, buffering of raw data is performed by **ldterm**. It is possible to canonize the raw data when the user has switched from raw to canonical mode. However, the reverse is not possible.

To summarize, in noncanonical mode, the **ldterm** module buffers all data until a request for the data arrives in the form of an M_READ message. The number of bytes sent upstream is the argument of the M_READ message.

Input flow control is regulated by the **ldterm** module by generating M_STARTI and M_STOPI high-priority messages. When sent downstream, receiving drivers or modules take appropriate action to regulate the sending of data upstream. Output flow control is activated when **ldterm** receives flow control characters in its data stream. The **ldterm** module then sets an internal flag indicating that output processing is to be restarted/stopped and sends an M_START/M_STOP message downstream.

Write-Side Processing

Write-side processing of the **ldterm** module is performed by the write-side **put** and **service** procedures. The **ldterm** module supports the following **ioctl**s:

TCSETA, **TCSETAW**, **TCSETAF**, **TCSETS**, **TCSETSW**, **TCSETSF**, **TCGETA**, **TCGETS**, **TCXONC**, **TCFLSH**, **TCSBRK**, **TIOCSWINSZ**, **TIOCGWINSZ**, and **JWINSIZE.**

All **ioctl**s not recognized by the **ldterm** module are passed downstream to the neighboring module or driver. BSD functionality is turned off by IEXTEN (see **termio(7)** for more details).

The following messages can be received on the write-side:

M_DATA, M_DELAY, M_BREAK, M_FLUSH, M_STOP, M_START, M_STOPI, M_STARTI, M_READ, M_IOCDATA, M_CTL, and M_IOCTL.

On the write-side, the **ldterm** module processes M_FLUSH, M_DATA, M_IOCTL, and M_READ messages, and all other messages are passed downstream unchanged.

An M_CTL message is generated by **ldterm** as a query to the driver for an intelligent peripheral and to determine the functional split for **termio** processing. If all or part of **termio** processing is done by the intelligent peripheral, **ldterm** can turn off this processing to avoid computational overhead. This is done by sending an appropriate response to the M_CTL message, as follows: (see also **ldterm(7)**).

- If all the **termio** processing is done by the peripheral hardware, the driver sends an M_CTL message back to **ldterm** with ioc_cmd of the structure iocblk set to MC_NO_CANON. If **ldterm** is to handle all **termio** processing, the driver sends an M_CTL message with ioc_cmd set to MC_DO_CANON. Default is MC_DO_CANON.
- If the peripheral hardware handles only part of the **termio** processing, it informs **ldterm** in the following way:

  The driver for the peripheral device allocates an M_DATA message large enough to hold a termios structure. The driver then turns on those c_iflag, c_oflag, and c_lflag fields of the termios structure that are processed on the peripheral device by ORing the flag values. The M_DATA message is then attached to the b_cont field of the M_CTL message it received. The message is sent back to **ldterm** with ioc_cmd in the data buffer of the M_CTL message set to MC_PART_CANON.

The **ldterm** module does not check if write-side flow control is in effect before forwarding data downstream. It expects the downstream module or driver to queue the messages on its queue until flow control is lifted.

EUC Handling in ldterm

The idea of letting post-processing (the `o_flags`) happen off the host processor is not recommended unless the board software is prepared to deal with international (EUC) character sets properly. The reason for this is that post-processing must take the EUC information into account. **`ldterm`** knows about the screen width of characters (that is, how many columns are taken by characters from each given code set on the current physical display) and it takes this width into account when calculating tab expansions. When using multibyte characters or multicolumn characters **`ldterm`** automatically handles tab expansion (when `TAB3` is set) and does not leave this handling to a lower module or driver.

By default, multibyte handling by **`ldterm`** is turned off. When **`ldterm`** receives an `EUC_WSET` **`ioctl`** call, it turns multibyte processing on, if it is essential to handle properly the indicated code set. Thus, if one is using single byte 8-bit codes and has no special multicolumn requirements, the special multicolumn processing is not used at all. This means that multibyte processing does not reduce the processing speed or efficiency of **`ldterm`** unless it is actually used.

The following describes how the EUC handling in **`ldterm`** works:

First, the multibyte and multicolumn character handling is only enabled when the `EUC_WSET` **`ioctl`** indicates that one of the following conditions is met:

- Code set consists of more than one byte (including the `SS2` and/or `SS3`) of characters.
- Code set requires more than one column to display on the current device, as indicated in the `EUC_WSET` structure.

Assuming that one or more of the above conditions, EUC handling is enabled. At this point, a parallel array, used for other information, is allocated. When a byte with the high bit arrives, it is checked to see if it is `SS2` or `SS3`. If so, it belongs to code set 2 or 3. Otherwise, it is a byte that comes from code set 1. Once the extended code set flag has been set, the input processor retrieves the subsequent bytes, as they arrive, to build one multibyte character. A counter field tells the input processor how many bytes remain to be read for the current character. The parallel array holds the display width of each logical character in the canonical buffer. During erase processing, positions in the parallel array are consulted to figure out how many backspaces need to be sent to erase each logical character. (In canonical mode, one backspace of input erases one logical character, no matter how many bytes or columns that character consumes.) This greatly simplifies erase processing for EUC.

The `t_maxeuc` field holds the maximum length, in memory bytes, of the EUC character mapping currently in use. The `eucwioc` field is a substructure, which holds information about each extended code set.

The `t_eucign` field aids in output post-processing (tab expansion). When characters are output, **`ldterm`** keeps a column to show the current cursor column. When it sends the first byte of an extended character, it adds the number of columns required for that character to the output column. It then subtracts one from the total width in memory bytes of that character and stores the result in `t_eucign`. This field tells **`ldterm`** how many bytes to ignore for the purposes of column calculation. (**`ldterm`** calculates the appropriate number of columns when it sees the first byte of the character.)

The field `t_eucwarn` is a counter for occurrences of bad extended characters. It is mostly useful for debugging. After receiving a certain number of invalid EUC characters (perhaps because of some problem on the line or with declared values), a warning is given on the system console.

There are two relevant files for handling multibyte characters: **`<euc.h>`** and **`<eucioctl.h>`**. The **`<eucioctl.h>`** header contains the structure that is passed with `EUC_WSET` and `EUC_WGET` calls. The normal way to use this structure is to get `CSWIDTH` (see note below) from the locale using a mechanism such as **`getwidth`** or **`setlocale`** and then copy the values into the structure in **`<eucioctl.h>`**, and send the structure using an **`I_STR`** **`ioctl`** call. The `EUC_WSET` call informs the **`ldterm`** module about the number of bytes in extended characters and how many columns the extended characters from each set consume on the screen. This allows **`ldterm`** to treat multibyte characters as single entities for erase processing and to calculate correctly tab expansions for multibyte characters.

NOTE

`LC_CTYPE` (instead of `CSWIDTH`) should be used in the environment in PowerMAX OS systems. See **`chrtbl(1M)`** for more information.

The file **`<euc.h>`** has the structure with fields for EUC width, screen width, and wide character width. The following functions are used to set and get EUC widths (these functions assume the environment where the `eucwidth_t` structure is needed and available):

```
#include <eucioctl.h> /* need some other things too, like stropts.h */

struct eucioc eucw;   /* for EUC_WSET/EUC_WGET to line discipline */
eucwidth_t width;     /* return struct from _getwidth() */

/*
 * set_euc   Send EUC code widths to line discipline.
 */

set_euc(e)
    set_euc(struct eucioc *e)
    {
    struct strioctl sb;

    sb.ic_cmd = EUC_WSET;
    sb.ic_timout = 15;
    sb.ic_len = sizeof(struct eucioc);
    sb.ic_dp = (char *) e;

    if (ioctl(0, I_STR, &sb) < 0)
        fail();
    }
/*
 * euclook   Get current EUC code widths from line discipline.
 */

euclook(e)
    euclook(struct eucioc *e)
    {
    struct strioctl sb;

    sb.ic_cmd = EUC_WGET;
    sb.ic_timout = 15;
    sb.ic_len = sizeof(struct eucioc);
    sb.ic_dp = (char *) e;
    if (ioctl(0, I_STR, &sb) < 0)
        fail();
    printf("CSWIDTH=%d:%d,%d:%d,%d:%d,
                    e->eucw[1], e->scrw[1],
                    e->eucw[2], e->scrw[2],
                    e->eucw[3], e->scrw[3]);
    }
```

The brief discussion of multiple byte character handling by the **`ldterm`** module was provided here for those interested in internationalization applications in the OS.

Support of termiox

The OS includes the extended general terminal interface (see **`termiox(7)`**) that supplements the **`termio(7)`** general terminal interface by adding for asynchronous hardware flow control, isochronous flow control and clock modes, and local implementations of additional asynchronous features. **`termiox(7)`** is handled by hardware drivers if the board supports it.

Hardware flow control supplements the **`termio(7)`** `IXON`, `IXOFF`, and `IXANY` character flow control. The **`termiox(7)`** interface allows for both unidirectional and bidirectional hardware flow control. Isochronous communication is a variation of asynchronous communication where two communicating devices provide transmit and/or receive clock to each other. Incoming clock signals can be taken from the baud rate generator on the local isochronous port controller. Outgoing signals are sent on the receive and transmit baud rate generator on the local isochronous port controller.

Terminal parameters are specified in the `termiox` structure that is defined in the **`<termiox.h>`**.

Hardware Emulation Module

If a Stream supports a terminal interface, a driver or module that understands all **`ioctls`** to support terminal semantics (specified by **`termio`** and `termios`) is needed. If there is no hardware driver that understands all **`ioctl`** commands downstream from the **`ldterm`** module, a hardware emulation module must be placed downstream from the **`ldterm`** module. The function of the hardware emulation module is to understand and acknowledge the **`ioctl`**s that may be sent to the process at the Stream head and to mediate the passage of control information downstream. The combination of the **`ldterm`** module and the hardware emulation module behaves as if there were a terminal on that Stream.

The hardware emulation module is necessary whenever there is no tty driver at the end of the Stream. For example, it is necessary in a pseudo-tty situation where there is process-to-process communication on one system and in a network situation where a **`termio`** interface is expected (for example, remote login) but there is no tty driver on the Stream.

Most actions taken by the hardware emulation module are the same regardless of the underlying architecture. However, some actions differ depending on whether the communication is local or remote and whether the underlying transport protocol supports the remote connection.

Each hardware emulation module has an open, close, read queue **`put`** procedure, and write queue **`put`** procedure.

The hardware emulation module does the following:

- Processes, if appropriate, and acknowledges receipt of the following **`ioctl`**s on its write queue by sending an `M_IOCACK` message back upstream: **TCSETA**, **TCSETAW**, **TCSETAF**, **TCSETS**, **TCSETSW**, **TCSETSF**, **TCGETA**, **TCGETS**, and **TCSBRK**.
- Acknowledges the Extended UNIX Code (EUC) **`ioctl`**s.
- If the environment supports windowing, it acknowledges the windowing **`ioctl`**s **TIOCSWINSZ**, **TIOCGWINSZ**, and `JWINSIZE`. If the environment does not support windowing, an `M_IOCNAK` message is sent upstream.
- If any other **`ioctl`**s are received on its write queue, it sends an `M_IOCNAK` message upstream.
- When the hardware emulation module receives an `M_IOCTL` message of type **TCSBRK** on its write queue, it sends an `M_IOCACK` message upstream and the appropriate message downstream. For example, an `M_BREAK` message could be sent downstream.
- When the hardware emulation module receives an `M_IOCTL` message on its write queue to set the baud rate to 0 (**TCSETAW** with `CBAUD` set to `B0`), it sends an `M_IOCACK` message upstream and an appropriate message downstream; for networking situations this probably is an `M_PROTO` mes-

sage, which is a TPI `T_DISCON_REQ` message requesting the transport provider to disconnect.

- All other messages (`M_DATA`, and so forth) not mentioned here are passed to the next module or driver in the Stream.

The hardware emulation module processes messages in a way consistent with the driver that exists below.

STREAMS-based Pseudo-Terminal Subsystem

The pseudo-terminal subsystem (pseudo-tty) supports a pair of STREAMS-based devices called the "master" device and "slave" device. The slave device provides processes with an interface that is identical to the terminal interface. However, where all devices that provide the terminal interface have some hardware device behind them, the slave device has another process manipulating it through the master half of the pseudo terminal. Anything written on the master device is given to the slave as an input and anything written on the slave device is presented as an input on the master-side.

Figure 7-2 illustrates the architecture of the STREAMS-based pseudo-terminal subsystem. The master driver called `ptm` is accessed through the clone driver (see **`clone(7)`**) and is the controlling part of the system. The slave driver called `pts` works with the **`ldterm`** module and the hardware emulation module to provide a terminal interface to the user process. An optional packetizing module called `pckt` is also provided. It can be pushed on the master-side to support packet mode.

The number of pseudo-tty devices that can be installed on a system depends on available memory.

Line Discipline Module

In the pseudo-tty subsystem (see Figure 7-2), the line discipline module **`ldterm`** is pushed on the slave side to present the user with the terminal interface.

`ldterm` may turn off the processing of the `c_iflag`, `c_oflag`, and `c_lflag` fields to allow processing to take place elsewhere. The **`ldterm`** module may also turn off all canonical processing when it receives an `M_CTL` message with the `MC_NO_CANON` command to support remote mode. Although **`ldterm`** passes through messages without processing them, the appropriate flags are set when a "get" **`ioctl`**, such as **`TCGETA`** or **`TCGETS`**, is issued to show that canonical processing is being performed.

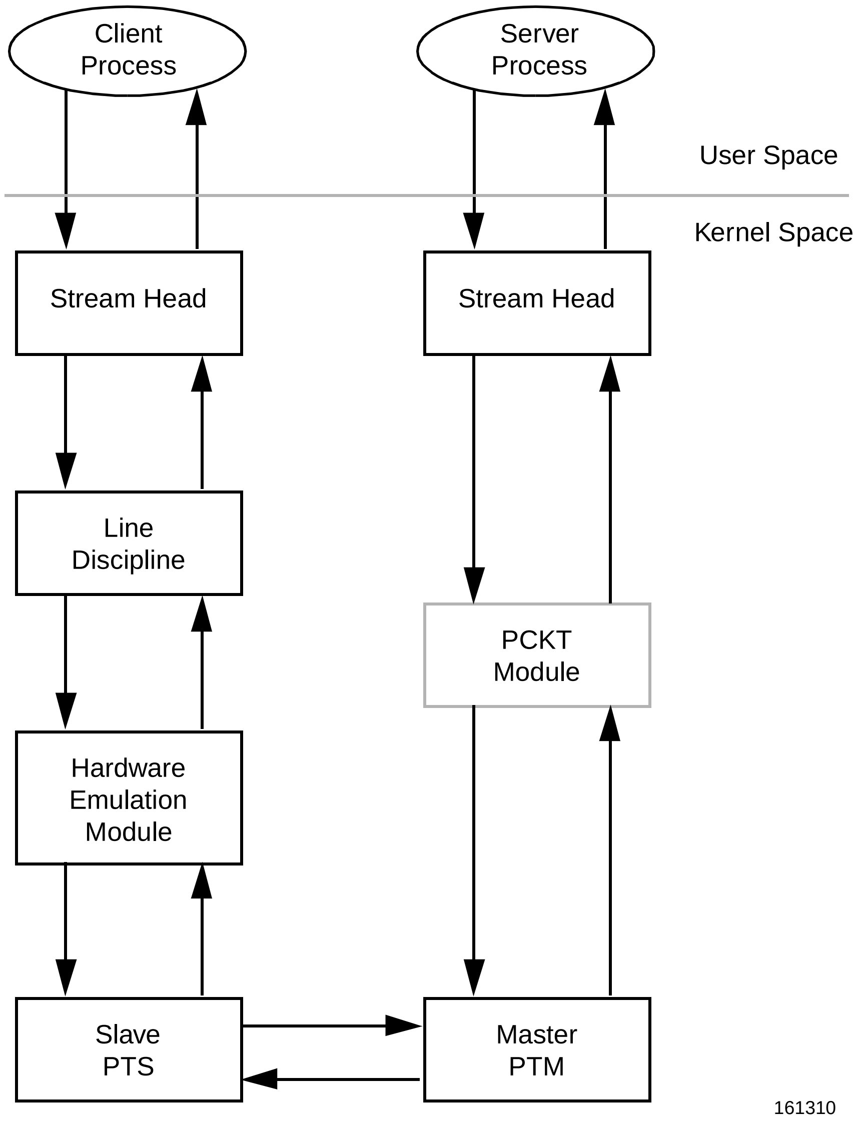


Figure 7-2. Pseudo-tty Subsystem Architecture

Pseudo-tty Emulation Module — ptem

Because the pseudo-tty subsystem has no hardware driver downstream from the **`ldterm`** module to process the terminal **`ioctl`** calls, another module that understands the **`ioctl`** commands is placed downstream from the **`ldterm`**. This module, known as `ptem`, processes all the terminal **`ioctl`** commands and mediates the passage of control information downstream.

ldterm and ptem together behave like a real terminal. Because there is no real terminal or modem in the pseudo-tty subsystem, some of the **ioctl** commands are ignored and cause only an acknowledgment of the command. The ptem module keeps track of the terminal parameters set by the various "set" commands such as TCSETA or TCSETAW but does not usually perform any action. For example, if one of the "set" **ioctl**s is called, none of the bits in the c_cflag field of **termio** has any effect on the pseudo-terminal except if the baud rate is set to 0. When setting the baud rate to 0, it has the effect of hanging up the pseudo-terminal.

The pseudo-terminal has no concept of parity so none of the flags in the c_iflag that control the processing of parity errors have any effect. The delays specified in the c_oflag field are not also supported.

The ptem module does the following:

- Processes, if appropriate, and acknowledges receipt of the following **ioctl**s on its write queue by sending an M_IOCACK message back upstream:

  TCSETA, **TCSETAW**, **TCSETAF**, **TCSETS**, **TCSETSW**, **TCSETSF**, **TCGETA**, **TCGETS**, and **TCSBRK.**

- Keeps track of the window size; information needed for the **TIOCSWINSZ**, **TIOCGWINSZ**, and **JWINSIZE** **ioctl** commands.

- When it receives any other **ioctl** on its write queue, it sends an M_IOCNAK message upstream.

- It passes downstream the following **ioctl**s after processing them:

  TCSETA, **TCSETAW**, **TCSETAF**, **TCSETS**, **TCSETSW**, **TCSETSF**, **TCSBRK**, and **TIOCSWINSZ.**

- ptem frees any M_IOCNAK messages it receives on its read queue in case the pckt module is not on the pseudo-terminal subsystem and the above **ioctl**s get to the master's Stream head, which then sends an M_IOCNAK message.

- In its open routine, the ptem module sends an M_SETOPTS message upstream requesting allocation of a controlling tty.

- When the ptem module receives an M_IOCTL message of type TCSBRK on its read queue, it sends an M_IOCACK message downstream and an M_BREAK message upstream.

- When it receives an **ioctl** message on its write queue to set the baud rate to 0 (**TCSETAW** with CBAUD set to B0), it sends an M_IOCACK message upstream and a 0-length message downstream.

- When it receives an M_IOCTL of type **TIOCSIGNAL** on its read queue, it sends an M_IOCACK downstream and an M_PCSIG upstream where the signal number is the same as in the M_IOCTL message.

- When the ptem module receives an M_IOCTL of type **TIOCREMOTE** on its read queue, it sends an M_IOCACK message downstream and the appropriate M_CTL message upstream to enable/disable canonical processing.

- When it receives an M_DELAY message on its read or write queue, it discards the message and does not act on it.
- When it receives an M_IOCTL message with type JWINSIZE on its write queue and if the values in the jwinsize structure of ptem are not zero, it sends an M_IOCACK message upstream with the jwinsize structure. If the values are zero, it sends an M_IOCNAK message upstream.
- When it receives an M_IOCTL message of type **TIOCGWINSZ** on its write queue and if the values in the winsize structure are not zero, it sends an M_IOCACK message upstream with the winsize structure. If the values are zero, it sends an M_IOCNAK message upstream. It also saves the information passed to it in the winsize structure and sends a STREAMS signal message for signal SIGWINCH upstream to the slave process if the size changed.
- When the ptem module receives an M_IOCTL message with type **TIOCGWINSZ** on its read queue and if the values in the winsize structure are not zero, it sends an M_IOCACK message downstream with the winsize structure. If the values are zero, it sends an M_IOCNAK message downstream. It also saves the information passed to it in the winsize structure and sends a STREAMS signal message for signal SIGWINCH upstream to the slave process if the size changed.
- All other messages not mentioned above are passed to the next module or driver.

Remote Mode

A feature known as remote mode is available with the pseudo-tty subsystem. This feature is used for applications that perform the canonical function normally done by the **ldterm** module and tty driver. The remote mode allows applications on the master-side to turn off the canonical processing. An **ioctl** **TIOCREMOTE** with a nonzero parameter (ioctl(fd, TIOCREMOTE, 1)) is issued on the master-side to enter the remote mode. When this occurs, an M_CTL message with the command MC_NO_CANON is sent to the **ldterm** module indicating that data should be passed when received on the read-side and no canonical processing is to take place. The remote mode may be disabled by ioctl(fd, TIOCREMOTE, 0).

Packet Mode

The STREAMS-based pseudo-terminal subsystem also supports a feature called packet mode. This is used to inform the process on the master-side when “state” changes have occurred in the pseudo-tty. Packet mode is enabled by pushing the pckt module on the master-side. Data written on the master-side is processed normally. When data is written on the slave-side or when other messages are encountered by the pckt module, a header is added to the message so it can be retrieved later by the master-side with a **getmsg** operation.

The pckt module does the following:

- When a message is passed to this module on its write queue, the module does no processing and passes the message to the next module or driver.
- The `pckt` module creates an `M_PROTO` message when one of the following messages is passed to it:

  `M_DATA, M_IOCTL, M_PROTO/M_PCPROTO, M_FLUSH, M_START/M_STOP, M_STARTI/M_STOPI, and M_READ.`

  All other messages are passed through. The `M_PROTO` message is passed upstream and retrieved when the user issues **`getmsg(2)`**.
- If the message is an `M_FLUSH` message, `pckt` does the following:

  If the flag is `FLUSHW`, it is changed to `FLUSHR` (because `FLUSHR` was the original flag before the `pts` driver changed it), packetized into an `M_PROTO` message, and passed upstream. To prevent the Stream head's read queue from being flushed, the original `M_FLUSH` message must not be passed upstream.

  If the flag is `FLUSHR`, it is changed to `FLUSHW`, packetized into an `M_PROTO` message, and passed upstream. To flush the write queues properly, an `M_FLUSH` message with the `FLUSHW` flag set is also sent upstream.

  If the flag is `FLUSHRW`, the message with both flags set is packetized and passed upstream. An `M_FLUSH` message with the `FLUSHW` flag set is also sent upstream.

Pseudo-tty Drivers — ptm and pts

In order to use the pseudo-tty subsystem, a node for the master-side driver **`/dev/ptmx`** and *N* number of slave drivers must be installed. (*N* is determined at installation time.) The names of the slave devices are **`/dev/pts/`***M* where *M* has the values 0 through *N*-1. A user accesses a pseudo-tty device through the master device (called `ptm`) that in turn is accessed through the clone driver (see **`clone(7)`**). The master device is set up as a clone device where its major device number is the major for the clone device and its minor device number is the major for the `ptm` driver.

The master pseudo-driver is opened by the **`open`** system call with **`/dev/ptmx`** as the device to be opened. The clone open finds the next available minor device for that major device; a master device is available only if it and its corresponding slave device are not already open. There are no nodes in the file system for master devices.

When the master device is opened, the corresponding slave device is automatically locked out. No user may open that slave device until it is unlocked. A user may invoke a function **`grantpt`** that will change the owner of the slave device to that of the user who is running this process, change the group ID to `tty`, and change the mode of the device to `0620`. Once the permissions have been changed, the device may be unlocked by the user. Only the owner or superuser can access the slave device. The user must then invoke the **`unlockpt`** function to unlock the slave device. Before opening the slave device, the user must call the **`ptsname`** function to obtain the name of the slave device. The functions **`grantpt`**, **`unlockpt`**, and **`ptsname`** are called with the file descriptor of the master device. The user may then invoke the **`open`** system call with the name that was returned by the **`ptsname`** function to open the slave device.

The following example shows how a user may invoke the pseudo-tty subsystem:

```
int fdm fds;
char *slavename;
extern char *ptsname();

fdm = open("/dev/ptmx", O_RDWR);  /* open master */
grantpt(fdm);                      /* change permission of slave */
unlockpt(fdm);                    /* unlock slave */
slavename = ptsname(fdm);         /* get name of slave */
fds = open(slavename, O_RDWR);    /* open slave */
ioctl(fds, I_PUSH, "ptem");       /* push ptem */
ioctl(fds, I_PUSH, "ldterm");     /* push ldterm */
```

Unrelated processes may open the pseudo-device. The initial user may pass the master file descriptor using a STREAMS-based pipe or a slave name to another process to enable it to open the slave. After the slave device is open, the owner is free to change the permissions.

NOTE

Certain programs such as **`write`** and **`wall`** are set group-ID (`setgid`) to `tty` and are also able to access the slave device.

After both the master and slave have been opened, the user has two file descriptors that provide full-duplex communication using two Streams. The two Streams are automatically connected. The user may then push modules onto either side of the Stream. The user also needs to push the `ptem` and **`ldterm`** modules onto the slave-side of the pseudo-terminal subsystem to get terminal semantics.

The master and slave drivers pass all STREAMS messages to their adjacent queues. Only the `M_FLUSH` needs some processing. Because the read queue of one side is connected to the write queue of the other, the `FLUSHR` flag is changed to `FLUSHW` flag and vice versa.

When the master device is closed, an `M_HANGUP` message is sent to the slave device that will render the device unusable. The process on the slave-side gets the `errno` `ENXIO` when attempting to write on that Stream but it will be able to read any data remaining on the Stream head read queue. When all the data has been read, **`read`** returns `0` indicating that the Stream can no longer be used.

On the last close of the slave device, a 0-length message is sent to the master device. When the application on the master-side issues a **`read`** or **`getmsg`** and `0` is returned, the user of the master device decides whether to issue a **`close`** that dismantles the pseudo-terminal subsystem. If the master device is not closed, the pseudo-tty subsystem will be available to another user to open the slave device.

Because 0-length messages are used to indicate that the process on the slave-side has closed and should be interpreted that way by the process on the master-side, applications on the slave-side should not **`write`** 0-length messages. If that occurs, the **`write`** returns `0`, and the 0-length message is discarded by the `ptem` module.

The standard STREAMS system calls can access the pseudo-tty devices. The slave devices support the `O_NDELAY` and `O_NONBLOCK` flags. Because the master-side does not act like the terminal, if `O_NONBLOCK` or `O_NDELAY` is set, **`read`** on the master side returns `-1` with `errno` set to `EAGAIN` if no data is available, and **`write`** returns `-1` with `errno` set to `EAGAIN` if there is internal flow control.

The master driver supports the **`ISPTM`** and **`UNLKPT`** **`ioctl`**s that are used by the functions **`grantpt`**, **`unlockpt`**, and **`ptsname`** (see **`grantpt(3C)`**, **`unlockpt(3C)`**, **`ptsname(3C)`**). The **`ioctl`** **`ISPTM`** determines whether the file descriptor is that of an open master device. On success, it returns the major/minor number (type `dev_t`) of the master device that can be used to determine the name of the corresponding slave device. The **`ioctl`** **`UNLKPT`** unlocks the master and slave devices. It returns `0` on success. On failure, the `errno` is set to `EINVAL` indicating that the master device is not open.

The format of these commands is:

`int` **`ioctl`** `(int` *fd*`, int` *command*`, int` *arg*`)`

where *command* is either **`ISPTM`** or **`UNLKPT`** and *arg* is `0`. On failure, `-1` is returned.

When data is written to the master-side, the entire block of data written is treated as a single line. The slave-side process reading the terminal receives the entire block of data. Data is not input-edited by the **`ldterm`** module regardless of the terminal mode. The master-side application is responsible for detecting an interrupt character and sending an interrupt signal `SIGINT` to the process in the slave-side. This can be done as follows:

`ioctl` `(`*fd*`, TIOCSIGNAL, SIGINT)`

where `SIGINT` is defined in the file **`<signal.h>`**. When a process on the master-side issues this **`ioctl`**, the argument is the number of the signal that should be sent. The specified signal is then sent to the process group on the slave-side.

To summarize, the master driver and slave driver have the following characteristics:

- Each master driver has a one-to-one relationship with a slave device based on major/minor device numbers.
- Only one open is allowed on a master device. Multiple opens are allowed on the slave device according to standard file mode and ownership permissions.
- Each slave driver minor device has a node in the file system.
- An open on a master device automatically locks out an open on the corresponding slave driver.
- A slave cannot be opened unless the corresponding master is open and has unlocked the slave.
- To provide a tty interface to the user, the **`ldterm`** and `ptem` modules are pushed on the slave-side.
- A **`close`** on the master sends a hang-up to the slave and renders both Streams unusable after all data has been consumed by the process on the slave side.
- The last **`close`** on the slave-side sends a 0-length message to the master but does not sever the connection between the master and slave drivers.

grantpt

The **`grantpt`** function changes the mode and the ownership of the slave device that is associated with the given master device. Given a file descriptor *fd*, **`grantpt`** first checks that the file descriptor is that of the master device. If so, it obtains the name of the associated slave device and sets the user ID to that of the user running the process and the group ID to `tty`. The mode of the slave device is set to `0620`.

If the process is already running as root, the permission of the slave can be changed directly without invoking this function. The interface is:

```
grantpt (int fd)
```

The **`grantpt`** function returns `0` on success and `-1` on failure. It fails if one or more of the following occurs: *fd* is not an open file descriptor, *fd* is not associated with a master device, the corresponding slave could not be accessed, or a system call failed because no more processes could be created.

unlockpt

The **`unlockpt`** function clears a lock flag associated with a master/slave device pair. Its interface is:

```
unlockpt (int fd)
```

The **`unlockpt`** returns `0` on success and `-1` on failure. It fails if one or more of the following occurs: *fd* is not an open file descriptor or *fd* is not associated with a master device.

ptsname

The **`ptsname`** function returns the name of the slave device that is associated with the given master device. It first checks that the file descriptor is that of the master. If it is, it then determines the name of the corresponding slave device **`/dev/pts/`***M* and returns a pointer to a string containing the null-terminated pathname. The return value points to static data whose content is overwritten by each call. The interface is:

```
char *ptsname (int fd)
```

The **`ptsname`** function returns a non-`NULL` pathname on success and a `NULL` pointer upon failure. It fails if one or more of the following occurs: *fd* is not an open file descriptor or *fd* is not associated with the master device.

8 Internationalization

8 Internationalization

Introduction

This chapter describes the programming interface to the PowerMAX OS internationalization feature. Its primary audience is the application programmer in C, although it may be of interest to system programmers and, to a lesser extent, administrators. It is assumed that readers are experienced in the UNIX system and the C language.

The chapter consists of a discussion of the programming interface, and covers only as much of the interface as programmers will need to get started. Much of the details can be found in the manual pages of the reference set. A list of UNIX system commands that have been enhanced for internationalization is provided in this chapter.

For the most part, the discussion concentrates on the System V implementation of ANSI standard C functions. These routines are supported in turn by the X/Open consortium, of which many System V vendors are members. To provide as realistic a view as possible, we give the locations of files used by these functions as they would be installed on a System V target implementation. You should not assume that these will be their locations on other X/Open or ANSI C-conforming systems, nor should you assume that these locations are permanent even on System V installations. In other words, the path names we provide should not be hardcoded in programs intended to be portable across UNIX or C language implementations. Similarly, although some type of “extended character set” will be supported on every X/Open and ANSI C-conforming system, the discussion below of “extended UNIX code” (EUC) is specific to System V, and should not be taken to describe the character encoding elsewhere.

Of course, both System V and X/Open go beyond the ANSI C standard in various other ways, most importantly in providing facilities for handling program messages in international contexts. In this regard, note that System V offers two distinct approaches to message handling, only one of which is standard to X/Open. Both approaches are described below, but keep in mind that the X/Open method is employed throughout much of Europe, so you can generally count on wider support for it than for the System V-specific method. By and large, System V internationalization is aligned with the X/Open *Portability Guide Issue 4* and the ISO/IEC 9945-2 (POSIX.2) specification.The only significant departures from this guide is that full support for internationalized regular expressions is not provided and the names and syntaxes of the commands used to build the locale specific files do not match the standards.

Discussion

This chapter describes C language functions that you can use to write UNIX applications that will process input and generate output in a user's native language or cultural environment. It shows you how to use these functions and some associated commands to create programs that make no assumptions about the language environments in which they will be run, and so are portable across these environments. We'll also look at a STREAMS module called `kbd` (for "keyboard display") that can be programmed to alter or supplement data as it flows between the physical terminal and a user process to produce language-dependent effects: for example, characters that cannot be entered from terminal keyboards, for instance, or overstriking sequences on printers.

The basic idea behind the internationalization interface is that at any time a C program has a current "locale": a collection of information on which it relies for language- or culture-dependent processing. This information is supplied by implementations and seen by the program only at run time. Because the information is stored externally to the program, applications need not make — and should not make if they mean to be portable — any assumptions about

- the *code sets* used by the implementation in which they are executed. The 7-bit US ASCII code set, for example, cannot represent every member of the Spanish character set; the 8-bit code sets used for most European languages cannot represent every ideogram and phonogram in the Japanese language.
- the *cultural and language conventions* of the application's users. The same date is formatted in the United States as `6/14/90`, in Great Britain as `14/6/90`, in Germany as `14.6.90`. Similar problems arise in formatting numeric and monetary values. By language conventions we mean, for instance, that the sharp `s` "β" (bühne) in German is collated as `ss`; the character `ch` in Spanish is collated after all other character sequences starting with `c`.
- the *language of the messages* in which the program communicates with the user. Interactive applications in an English-speaking setting usually will query users at some point for a `yes` or `no` response; in a German-language setting the responses will be `ja` or `nein`; in a French one `oui` or `non`. Program error messages will differ much more widely than that across languages: `File not found`, `Fichier inexistant`, and so on.

A typical locale, then, consists of an encoding scheme; databases that describe the conventions appropriate to some nationality, culture, and language; and a file which you supply, that contains your program's message strings in whatever language the locale implements.

Organization

The discussion is organized in terms of these three elements of a locale. "Character Representation" describes the character encoding used by System V implementations that support the internationalization feature, and the ANSI C library functions that perform codeset-dependent tasks. It also discusses the sequences of bytes, or "multibyte characters," that are needed to encode Asian-language ideograms. "Cultural and Language

Conventions” looks at ANSI C functions that collate strings and format cultural information in locale-dependent ways. “Message Handling” describes the functions you use to generate program messages in a user's native language. The “`kbd`” section outlines the function of the STREAMS module used as the keyboard display interface. Before we turn to this material, there's some background we need to give on how C programs determine their locales.

NOTE

For the relationship of System V internationalization to the ANSI C and X/Open standards, see the “Introduction” section in the beginning of this chapter.

Locales

One or more locales is provided by every UNIX system implementation that supports the internationalization feature. Each OS program begins in the `"C"` locale, which causes all library functions to behave as they have historically. The “`POSIX`” locale is another name for the default `"C"` locale. Either name can be used to specify the default locale. Any other locale will cause certain functions to behave in the appropriate language- or culture-dependent ways. Locales can have names that are strings — `"french"`, `"german"`, and so forth (or `"fr"` and `"de"`, following ISO conventions) — but only `"C", "POSIX"` and `""` are guaranteed. When given as the second argument to the ANSI C `setlocale` function, the string `""` tells the program to change its current locale to the one set by the user, or the system administrator for all users, in the UNIX system shell environment. Any other argument will cause the program to change its current locale to the one specified by the string. See the **`environ(5)`** manual page for further information.

Locales are partitioned into categories:

`LC_CTYPE`	character representation information
`LC_TIME`	date and time printing information
`LC_MONETARY`	currency printing information
`LC_NUMERIC`	numeric printing information
`LC_COLLATE`	sorting information
`LC_MESSAGES`	message information

In the implementation's view, these categories are files in directories named for each locale it supports; the directories themselves are kept in **`/usr/lib/locale`**. In the user's view, the categories are environment variables that can be set to given locales:

```
$ LC_COLLATE=german export LC_COLLATE
$ LC_CTYPE=french export LC_CTYPE
$ LC_MESSAGES=french export LC_MESSAGES
```

In the program's view, the categories are macros that can be passed as the first argument to **`setlocale`** to specify that it change the program's locale for just that category. That is,

```
setlocale(LC_COLLATE, “”);
```

tells the program to use the sorting information for the locale specified in the environment, in this case, `german`, but leaves the other categories unchanged.

`LC_ALL` is the macro that specifies the entire locale. Given the environment setup above, the code

```
setlocale(LC_ALL, ““);
```

would allow a user to work in a French interface to a program while sorting German text files. When the `LC_ALL` variable is set, it overrides all other `LC_` variables, as well as the `LANG` setting; setting it to `spanish`, for instance, causes all the categories to be set to `spanish` in the environment. The `LANG` environment variable is checked after the environment variables for individual categories, so a user could set a category to `french` and use `LANG` to set the other categories to `spanish`.

`setlocale`, then, is the interface to the program's locale. Any program that has a need to use language or cultural conventions should put a call such as

```
#include <locale.h>
/*...*/
setlocale(LC_ALL, ““);
```

early in its execution path. You'll generally want to use `“”` as the second argument to **`setlocale`** so that your application will change locales correctly for whatever language environment in which it is run. Occasionally, though, you may want to change the locale or a portion of it for a limited duration in a way that's transparent to the user.

Suppose, for example, there are parts of your program that need only the ASCII upper- and lowercase characters guaranteed by ANSI C in the **`<ctype.h>`** header. In these parts, in other words, you want the program to see the character classification information in `LC_CTYPE` for the `“C”` locale. Since the user of the program in a non-ASCII environment will presumably have set `LC_CTYPE` to a locale other than `“C”`, and will not be able to change its setting mid-program, you'll have to arrange for the program to change its `LC_CTYPE` locale whenever it is in those parts. **`setlocale`** returns the name of the current locale for a given category and serves in an inquiry-only capacity when its second argument is a null pointer. So you might want to use code something like this:

```
char *oloc;
/*...*/
oloc = setlocale(LC_CTYPE, NULL);
if (setlocale(LC_CTYPE, “C”) != 0)
{
   /* use temporarily changed locale */
   (void)setlocale(LC_CTYPE, oloc);
}
```

The **`setlocale(3C)`** function is described in section (3C) of the reference manual set.

Character Representation

Every System V implementation that supports the internationalization feature can represent up to four code sets concurrently in an 8-bit byte stream. The code sets are configured in a scheme called “extended UNIX code,” or EUC. As shown in Table 8-1, the primary code set (code set 0) is always 7-bit US ASCII. Each byte of any character in a supplementary code set (code sets 1,2, or 3) has the high-order bit set; code sets 2 and 3 are distinguished from code set 1 and each other by their use of a special “shift byte” before each character.

Table 8-1. EUC Code Set Representations

Code Set	EUC Representation
0	0xxxxxxx
1	1xxxxxxx [1xxxxxxx]
2	`SS2` 1xxxxxxx [1xxxxxxx]
3	`SS3` 1xxxxxxx [1xxxxxxx]

There are two shift bytes: `SS2` and `SS3`. `SS2` is represented in hexadecimal by `0x8e`; `SS3` is represented by `0x8f`.

EUC is provided mainly to support the huge number of ideograms needed for I/O in an Asian-language environment. To work within the constraints of usual computer architectures, these ideograms are encoded as sequences of bytes, or “multibyte characters.” Because single-byte characters (the digits 0-9, say) can be intermixed with multibyte characters, the sequence of bytes needed to encode an ideogram must be self-identifying: regardless of the supplementary code set used, each byte of a multibyte character will have the high-order bit set; if code sets 2 or 3 are used, each multibyte character will also be preceded by a shift byte. In a moment, we'll take a closer look at multibyte characters and at the implementation-defined integral type `wchar_t` that lets you manipulate variable width characters as uniformly sized data objects called “wide characters.” We'll also discuss the functions you use to manage multibyte and wide characters.

Of course, programmers developing applications for less complex linguistic environments need not concern themselves with the details of multibyte or wide character processing. In Europe, for instance, a single 8-bit code set can hold all the characters of the major languages. In these environments, at least one 8-bit character set will be represented in the EUC code sets, usually code sets 0 and 1. Other character sets may be represented simultaneously, in various combinations. Applications will work correctly with any standard 7- or 8-bit character set, provided (1) they are “8-bit clean” — they make no assumptions about the contents of the high-order bit when processing characters; and (2) they use correctly the functions supplied by the interface for codeset-dependent tasks — character classification and conversion, in other words. We'll take a brief look at these issues now.

"8-bit Clean"

UNIX system applications written for 7-bit US ASCII environments have sometimes assumed that the high-order bit is available for purposes other than character processing. In data communications, for instance, it was often used as a parity bit. On receipt and after a parity check, the high-order bit was stripped either by the line discipline or the program to obtain the original 7-bit character:

```
char c;
/* bitwise AND with octal value 177 strips high-order bit
*/
c &= 0177;
```

Other programs used the high-order bit as a private data storage area, usually to test a flag:

```
char c;
/*...*/
c |= 0200;/* bitwise OR with octal value 200 sets flag */
/*...*/
c &= 0177;/* bitwise AND removes flag */
/*...*/
if (c & 0200)/* test if flag set */
{
/*...*/
}
c &= 0177;/* original character */
```

Neither of these practices will work with 8-bit or larger code sets. To show you how to store data in a codeset-independent way, we'll look at code fragments from a UNIX system program before and after it was made 8-bit clean. In the first fragment, the program sets the high-order bit of characters quoted on the command line:

```
#define LITERAL '\xd3
#define QUOTE 0200
register int c;
register char *argp = arg->argval;

if (c == LITERAL)/* character is a single quote */
{
    /* get next character until next single quote */
    while ((c = getc()) && c != LITERAL)
    {
        *argp++ = (c | QUOTE);
    }
}
```

In the next fragment, the same data is stored by internally placing backslashes before quoted characters in the command string:

```
#define LITERAL '\xd3
register int c;
register unsigned char *argp = arg->argval;

if (c == LITERAL)
{
    while ((c = getc()) && c != LITERAL)
    {
    /* precede each character within single quotes with a backslash */
        *argp++ = '\\';
        *argp++ = c;
    }
}
```

Because the data is stored in 8-bit character values rather than the high-order bit of the quoted characters, the program will work correctly with code sets other than US ASCII. Note, by the way, the use of the type `unsigned char` in the declaration of the character pointer in the second fragment. We'll discuss the reasons why you use it in the next section.

Character Classification and Conversion

The ANSI C functions declared in the **`<ctype.h>`** header file classify or convert character-coded integer values according to type and conversion information in the program's locale. The tables used by these functions are stored in the `LC_CTYPE` file in the locale’s directory. The **`chrtbl(1M)`** and **`wchrtbl(1M)`** commands are used to build these locale specific tables. All the classification functions except **`isdigit`** and **`isxdigit`** can return nonzero (true) for single-byte supplementary code set characters when the `LC_CTYPE` category of the current locale is other than `“C”` or `“POSIX”.`In a Spanish locale, `isalpha('~')` should be true. Similarly, the case conversion functions **`toupper`** and **`tolower`** will appropriately convert any single-byte supplementary code set characters identified by the **`isalpha`** function.

The point of these functions is to let you determine a character's type or case without reference to its numeric value in a given code set. Whereas a program written for a US ASCII environment might test whether a character is printable with the code.

```
if ( c <= 037 || c == 0177 )
```

a codeset-independent program will use **`isprint`**:

```
if ( !isprint(c) )
```

Similarly,

```
c = toupper(c);
```

will do the same thing as

```
if( c >= 'a' && c <= 'z')
    c += 'a' -'A';
```

without relying on the fact that upper- and lowercase characters are numerically contiguous in the US ASCII code set.

The **`<ctype.h>`** functions are almost always macros that are implemented using table lookups indexed by the character argument. Their behavior is changed by resetting the table(s) to the new locale's values. The classification functions are described on the **`ctype(3C)`** manual page, the conversion functions on the **`conv(3C)`** page. Both single- and multibyte character classification and conversion routines are declared in the **`<wctype.h>`** header, and described on the pages **`wctype(3W)`** and **`wconv(3W)`**. Note that the multibyte routines are not part of the ANSI C standard, nor are the single-byte functions **`isascii`** and **`toascii`**.

Sign Extension

In some C language implementations, character variables that are not explicitly declared `signed` or `unsigned` are treated as nonnegative quantities with a range typically from 0 to 255. In other implementations, they are treated as signed quantities with a range typically from -128 to 127. When a signed object of type `char` is converted to a wider integer, the machine is obliged to propagate the sign, which is encoded in the high-order bit of the new integer object. If the character variable holds an eight-bit character with the high-order bit set, the sign bit will be propagated the full width of an object of type `int` or `long`, producing a negative value.

You can avoid this problem (which typically occurs with the `ctype` functions) by declaring as `unsigned` any object of type `char` that is liable to be converted to a wider integer. In the example we showed earlier, for instance, the declaration of the character pointer as of type `unsigned char` would guarantee that on any implementation the values pointed at will be nonnegative. On this system, objects of type `char` are by default unsigned.

Characters Used as Indices

A related problem arises when characters are used as indices into arrays and tables. If a table has been defined to contain only 128 possible characters, the amount of allocated memory will be exceeded if an eight-bit character whose value is greater than 127 is used as an index. Moreover, if the character is signed, the index may be negative.

The solution, at least when dealing with 8-bit code sets, is obviously to increase the size of the table from the 7-bit maximum of 128 to the 8-bit maximum of 256. And again, to declare the object that will hold the character as type `unsigned char`.

Wide Characters

Earlier in this section we looked at the encoding scheme used for the multibyte characters that are needed to represent Asian-language ideograms. We noted that because single-byte characters can be intermixed with multibyte characters, the sequence of bytes needed to encode an ideogram must be self-identifying: regardless of the supplementary code set used, each byte of a multibyte character will have the high-order bit set. In this way, any byte of a multibyte character can always be distinguished from a member of the primary, 7-bit US ASCII code set, whose high-order bit is not set (or “0”). If code sets 2 or 3 are used, each multibyte character will also be preceded by a shift byte; that is, if code set 1 were dedicated to a single-byte character set, either of code sets 2 or 3 could be used to represent multibyte characters. Given some set of these encodings, then any program interested in the next character will be able to determine whether the next byte represents a single-byte character or the first byte of a multibyte character. If the latter, then the program will have to retrieve bytes until the character is complete. A maximum of two bytes per multibyte character is supported in the supplementary code sets (exclusive of the single shift bytes used for code sets 2 and 3).

Some of the inconvenience of handling multibyte characters would be eliminated, of course, if all characters were a uniform number of bytes. ANSI C provides the implementation-defined integral type `wchar_t` to let you manipulate variable-width characters as uniformly sized data objects called wide characters. Since there can be thousands or tens of thousands of ideograms in an Asian-language set, programs must use a 32-bit sized integral value to hold all members. `wchar_t` is defined in the headers **`<stdlib.h>`** and **`<widec.h>`** as a `typedef` declaration of `long`.

Implementations provide appropriate libraries with functions that you can use to manage multibyte and wide characters. We'll look at these functions below.

For each wide character there is a corresponding EUC representation and vice versa; the wide character that corresponds to a regular single-byte character is required to have the same numeric value as its single-byte value, including the null character. There is no guarantee that the value of the macro `EOF` can be stored in a `wchar_t`, just as `EOF` might not be representable as a `char`.

Table 8-2. EUC and Corresponding 32-bit Wide-Character Representation

Code Set	EUC Code Representation	Wide-character Representation
0	0xxxxxxx	0000000000000000000000000xxxxxxx
1	1xxxxxxx 1xxxxxxx1xxxxxxx	0011000000000000000000000xxxxxxx 001100000000000000xxxxxxxxxxxxxx
2	SS2 1xxxxxxx SS2 1xxxxxxx1xxxxxxx	0001000000000000000000000xxxxxxx 000100000000000000xxxxxxxxxxxxxx
3	SS3 1xxxxxxx SS3 1xxxxxxx1xxxxxxx	0010000000000000000000000xxxxxxx 001000000000000000xxxxxxxxxxxxxx

Most of the functions provided let you convert multibyte characters into wide characters and back again. Before we turn to the functions, we should note that most application programs will not need to convert multibyte characters to wide characters in the first place. Programs such as **`diff`**, for example, will read in and write out multibyte characters, needing only to check for an exact byte-for-byte match. More complicated programs such as **`grep`**, that use regular expression pattern matching, may need to understand multibyte characters, but only the common set of functions that manages regular expressions needs this knowledge. The program **`grep`** itself requires no other special multibyte character handling. Finally, note that except for **`libc`**, the libraries described below are archives, not shared objects. They cannot be dynamically linked with your program.

Multibyte and Wide-character Conversion

ANSI C provides five library functions that manage multibyte and wide characters:

mblen	length of next multibyte character
mbtowc	convert multibyte character to wide character
wctomb	convert wide character to multibyte character
mbstowcs	convert multibyte character string to wide character string
wcstombs	convert wide character string to multibyte character string

The first three functions are described on the **mbchar(3C)** manual page, the last two on the **mbstring(3C)** page.

Input/Output

Since most programs will convert between multibyte and wide characters just before or after performing I/O, **libc** provides routines that let you manage the conversion within the I/O function itself. **getwc**, for instance, reads bytes from a stream until a complete EUC character has been seen and returns it in its wide-character representation. **getws** does the same thing for strings; **putwc** and **putws** are the corresponding write versions. Of course, these routines and others are functionally similar to the **stdio(3S)** functions; they differ only in their handling of EUC representations. Check the 3W manual pages for details. Here is a look at how you can expect the functions to work.

Given the following declarations:

```
#include <stdio.h>
#include <widec.h>


wchar_t s1[BUFSIZ];  /* declare array s1 to store wide characters*/
char    s2[BUFSIZ];  /* declare array s2 of characters
                     for EUC representation */
```

a multibyte string can be input into `s1` using **getws**:

```
getws(s1);           /* read EUC string from stdin and convert
                     to process code string in s1 */
```

gets and **strtows**:

```
gets(s2);            /* read EUC string from stdin into s2 */
strtows(s1, s2);     /* convert EUC string in s2 to process
                     code string in s1 */
```

the `%ws` conversion specifier for **scanf**:

```
scanf("%ws", s1);    /* read EUC string from stdin and convert to
                     process code string in s1 */
```

the `%s` conversion specifier for **scanf** and **strtows**:

```
scanf("%s", s2);        /* read EUC string from stdin into s2 */
strtows(s1, s2);        /* convert EUC string in s2 to process
                        code string in s1 */
```

You can use **`putws`**, **`wstostr`**, and the `%ws` conversion specifier for **`printf`** in the same way for output.

Character Classification and Conversion

Single- and multibyte character classification and conversion functions are provided in **`libc`**. You can use these routines to test 7-bit US ASCII characters, for instance, in their wide-character representations, or to determine whether multibyte characters are ideograms, phonograms, or the like. See the **`wctype(3W)`** and **`wconv(3W)`** manual pages for details. The **`towupper(3C)`** and **`towlower(3C)`** functions provide conversion of wide characters between upper and lower case.

As noted, these routines are declared in the **`<wchar.h>`** header file.

curses Support

32-bit versions of certain `curses` functions are provided in `libcurses` and declared in **`<curses.h>`**. Check the 3X manual pages especially **`curses(3X)`**, for some of the things you need to look out for in using these functions.

C Language Features

To give even more flexibility to the programmer in an Asian environment, ANSI C provides 32-bit wide character constants and wide string literals. These have the same form as their non-wide versions except that they are immediately prefixed by the letter `L`:

`'x'`	regular character constant
`'¥'`	regular character constant
`L'x'`	wide character constant
`L'¥'`	wide character constant
`"abc¥xyz"`	regular string literal
`L"abc¥xyz"`	wide string literal

Note that multibyte characters are valid in both the regular and wide versions. The sequence of bytes necessary to produce the ideogram ¥ is encoding-specific, but if it consists of more than one byte, the value of the character constant `¥'` is implementation-defined, just as the value of `'ab'` is implementation-defined. A regular string literal contains exactly the bytes (except for escape sequences) specified between the quotes, including the bytes of each specified multibyte character. Of course, programs using this feature will probably not be portable.

When the compilation system encounters a wide character constant or wide string literal, each multibyte character is converted (as if by calling the **`mbtowc`** function) into a wide character. Thus the type of `L'¥'` is `wchar_t` and the type of `L“abc¥xyz”` is array of `wchar_t` with length eight. (Just as with regular string literals, each wide string literal has an extra zero-valued element appended, but in these cases it is a `wchar_t` with value zero.)

Just as regular string literals can be used as a short-hand method for character array initialization, wide string literals can be used to initialize `wchar_t` arrays:

```
wchar_t *wp = L“a¥z”;
wchar_t x[] = L“a¥z”;
wchar_t y[] = {L'a', L'¥', L'z', 0};
wchar_t z[] = {'a', L'¥', 'z', '\0'};
```

In the above example, the three arrays `x`, `y` and `z` as well as the array pointed to by **`wp`**, have the same length and all are initialized with identical values.

Adjacent wide string literals will be concatenated, just as with regular string literals. Adjacent regular and wide string literals produce undefined behavior.

System-defined Words

The UNIX system uses a number of special words to identify system resources, user and group names, process IDs, peripherals, and other information. The following should be specified only with characters from the primary 7-bit ASCII code set:

- process ID numbers
- message queue, semaphore, and shared memory identifiers
- external symbol names and fill patterns for the **`cc`** and **`as`** commands
- layer names

Although the following can be specified with supplementary code set characters, we recommend against it:

- user names
- group names
- passwords
- names of devices, terminals, and special devices
- printer names and printer class names
- system names
- disk pack, diskette, and tape label/volume names
- names visible to other machines on a network
- environment variable names

The following can be specified with primary or supplementary code set characters, subject to length limitations imposed by the file system:

- file names
- directory names
- command names
- file system names

File name prefixes of the form `s.`, or suffixes of the form `.c`, must be specified with characters from the primary code set.

Cultural and Language Conventions

In this section we'll look at how programs interpret or print the formatted date and time, or formatted numeric and monetary values, in locale-dependent ways. We'll also look at the functions you use to collate strings according to the rules of the language the locale implements.

Date and Time

The ANSI C function **`strftime`** provides a **`sprintf`**-like formatting of the values in a `struct tm`, along with some date and time representations that depend on the `LC_TIME` category of the current locale. (**`strftime`** supersedes **`ctime`** and **`ascftime`**, although, for the sake of compatibility with older systems, these routines format the date and time correctly for a given locale.) Unlike the other `LC_` categories, there is no special command that must be used to generate the `LC_TIME` file. It is an ASCII file that is prepared using one of the standard editors. See the **`strftime(3C)`** and **`strftime(4)`** manual pages for information on the format of this file. Here is how you might use **`strftime`** to print the current date in a locale-dependent way:

```
#include <stdio.h>
#include <locale.h>
#include <time.h>

main()
{
    time_t tval;
    struct tm *tmptr;
    char buf [BUFSIZ];

    tval = time(NULL);
    tmptr = localtime(&tval);

    setlocale(LC_ALL, “”);

    strftime(buf, BUFSIZE, “%x”, tmptr);
    puts(buf);
}
```

In this case, **`strftime`** puts characters into the array pointed to by `buf`, as controlled by the string pointed to by `%x`. `%x` is a directive that provides an implementation-defined date representation appropriate to the locale. In a Spanish locale, for example, the current date

June 14, 1990, might be represented as `14 Junio 1990` or `14/6/90` or any other way the implementation deems appropriate to the locale. No particular format is guaranteed. Use the `%X` directive to obtain the locale's appropriate time representation:

```
strftime(buf, BUFSIZE, “%X”, tmptr)
```

or `%c` to obtain both the date and time representation. Check the **`strftime(3C)`** manual page for the other directives.

Although it requires a bit more work, you can control the format of the date and time for different locales by using **`printf`** with the message retrieval functions **`gettxt`** or **`catgets`**. Suppose, for example, you want the current date June 14, 1990, to be displayed in a British locale as `14/6/90`, in a German locale as `14.6.90`, and in a U.S. locale as `6/14/90`. What you need, in other words, is some way to switch the arguments to **`printf`** depending on the program's current locale. The `%`*n*`$` form of conversion specification lets you convert the *n*th argument in a **`printf`** argument list rather than the next unused argument. That is,

```
printf(gettxt(“progmsgs:9”, “%d/%d/%d\n”),
    tm->tm.mon,
    tm->tm.mday,
    tm->tm.year);
```

will produce the locale-dependent date displays we want, so long as the string whose index is `9` in the message file **`progmsgs`** reads, in the British locale

```
“%2$d/%1$d/%3$d\n”
```

in the German locale

```
“%2$d.%1$d.%3$d\n”
```

and in the U.S. locale

```
“%1$d/%2$d/%3$d\n” /* or simply “%d/%d/%d\n” */
```

You can use **`scanf`** in a similar way to interpret formatted dates in the input:

```
int month, day, year;
scanf(gettxt(“progmsgs:9”, “%d/%d/%d\n”),
    &month, &day, &year);
```

Note that the `%`*n*`$` form of conversion specification has a wider application than the one we've described here, as we'll show in the “Message Handling” section below. There, too, we'll take a closer look at **`gettxt`** and **`catgets`**. Detailed information concerning **`printf(3S)`**, **`scanf(3S)`**, **`gettxt(3C)`** and **`catgets(3C)`** can be found in their respective manual pages.

Numeric and Monetary Information

The ANSI C **`localeconv`** function returns a pointer to a structure containing information useful for formatting numeric and monetary information appropriate to the current locale's `LC_NUMERIC` and `LC_MONETARY` categories. (This is the only function whose behavior depends on more than one category.) For numeric values the structure describes the decimal-point (radix) character, the thousands separator, and where the separator(s) should be located. Other structure members describe how to format monetary values, as in the following, somewhat contrived example. Assuming **`setlocale`** has been called, the code

```
int thousands = 1;
int rest = 234;
int frac = 56;

struct lconv *lptr;
lptr = localeconv();

printf(“%s%d%c%d%c%d\n”,
     lptr->currency_symbol,
     thousands, lptr->mon_thousands_sep[0], rest,
     lptr->mon_decimal_point[0], frac);
```

will print `kr1.234,56` in a Norwegian locale, `F 1.234,56` in a Dutch locale, and `SFrs.1,234.56` in a Swiss locale. Check the **`localeconv(3C)`** manual page for details.

`localeconv` aside, functions that write or interpret printable floating values — **`printf`** and **`scanf`**, for example — may use a decimal-point character other than a period (`.`) when the `LC_NUMERIC` category of the current locale is other than `“C”` or `“POSIX”.` There is no provision for converting numeric values to printable form with thousands separator-type characters, but when converting from a printable form to an internal form, implementations are allowed to accept such additional forms, again in other than the `“C”` or `“POSIX”` locale. Functions that make use of the decimal-point character are the **`printf`** and **`scanf`** families, **`atof`**, and **`strtod`**. Functions that are allowed implementation-defined extensions for the thousands separator are **`atof`**, **`atoi`**, **`atol`**, **`strtod`**, **`strtol`**, **`strtoul`**, and the **`scanf`** family.

The **`chrtbl(1M)`** and **`wchrtbl(1M)`** commands are used to build the locale specific `LC_NUMERIC` category table. The **`montbl(1M)`** command is used to build the locale specific `LC_MONETARY` category table.

String Collation

ANSI C provides two functions for locale-dependent string compares. **`strcoll`** is analogous to **`strcmp`** except that the two strings are compared according to the `LC_COLLATE` category of the current locale. (see **`strcoll(3C)`** and **`strcmp(3C)`**). The locale specific `LC_COLLATE` table is built using the **`colltbl(1M)`** command. Conceptually, collation occurs in two passes to obtain an appropriate ordering of accented characters, two-character sequences that should be treated as one (the Spanish character `ch`, for example), and single characters that should be treated as two (the sharp `s` in German “β” (bühne), for instance). Since this comparison is not necessarily as inexpensive as **`strcmp`**, the **`strxfrm`** function is provided to transform a string into another. Therefore, any two such after-translation strings can be passed to **`strcmp`** to get an ordering identical to what **`strcoll`** would have returned if passed the two pre-translation strings. You are responsible for keeping track of the strings in their translated and printable forms. Generally, you should use **`strxfrm`** when a string will be compared a number of times.

The following example uses **`qsort(3C)`** and **`strcoll(3C)`** to sort lines in a text file:

Assuming **`malloc`** succeeds, the return value of **`compare`** (*s1*, *s2*) should correspond to the return value of **`strcoll`**(*s1*, *s2*). Although it is too complicated to show here, it would probably be better to hold onto the strings for subsequent comparisons rather than transforming them each time the function is called. Details of **`strcoll(3C)`** and **`strxfrm(3C)`** can be found in their respective manual pages.

```
#include <stdio.h>
#include <string.h>
#include <locale.h>

char table [ELEMENTS] [WIDTH];

main(argc, argv)
int argc;
char **argv;
{
    FILE *fp;
    int nel, i;

    setlocale(LC_ALL, ““);

    if ((fp = fopen(argv[1], ”r”)) == NULL) {
        fprintf(stderr, gettxt(“progmsgs:2”,
            “Can't open %s\n”, argv[1]);
        exit(2);
    }
    for (nel = 0; nel < ELEMENTS &&
        fgets(table[nel], WIDTH, fp); ++nel);

    fclose(fp);

    if (nel >= ELEMENTS) {
        fprintf(stderr, gettxt(“progmsgs:3”,
            “File too large\n”);
        exit(3);
    }
    qsort(table, nel, WIDTH, strcoll);
    for (i = 0; i < nel; ++i)
        fputs(table(i), stdout);
    return(0);
}
```

The next example does the same thing with a function that uses **`strxfrm`**:

```
compare (s1, s2)
char *s1, *s2;
{
    char *tmp;
    int result;
    size_t n1 = strxfrm(NULL, s1, 0) + 1;
    size_t n2 = strxfrm(NULL, s2, 0) + 1;

    if ((tmp = malloc(n1 + n2)) == NULL)
        return strcmp(s1, s2);

    (void)strxfrm(tmp, s1, n1);
    (void)strxfrm(tmp + n1 + 1, s2, n2);

    result = strcmp(tmp, tmp + n1 + 1);
    free(tmp);
    return(result);
}
```

Message Handling

As the examples in earlier sections may have suggested, the general approach behind the message handling feature is to separate messages from program source code, replacing hard-coded character strings with function calls that fetch the strings from a file. You supply the file, which contains your program's messages in whatever language the locale implements. You can adapt your applications to different locales, then, without having to change and recompile source code.

In this section we'll look at the System V-specific and X/Open message handling facilities as they might be used to adapt an “English-speaking” program to a French locale. The code fragment below queries the English-speaking user for an affirmative or negative response, and reads the response:

```
#include <stdio.h>

main()
{
    int yes();

    while(1)
    {
        puts("Choose (y/n)");
        if (yes())
            puts("yes");
        else
            puts("no");
    }
}

static int
yes()
{
    int i, b;

    i = b = getchar();
    while (b != '\n' && b != '\0' && b != EOF)
        b = getchar();
    return(i == 'y');
}
```

mkmsgs and gettxt (System V-specific)

You use the **`mkmsgs(1)`** command to store the strings for a given locale in a file that can be read by the message retrieval function **`gettxt(3C)`**. **`mkmsgs`** accepts an input file consisting of text strings separated by new lines. If the file **`fr.str`** contains

```
Votre choix (o/n)
oui
non
```

the command:

```
$ mkmsgs -o -i french fr.str progmsgs
```

will generate a file called **`progmsgs`** that, when installed in the directory **`/usr/lib/locale/french/LC_MESSAGES`**, can be read by **`gettxt`** such that

```
puts(gettxt("progmsgs:1", "Choose (y/n)"));
```

will display:

```
Votre choix (o/n)
```

in a French locale. **`gettxt`** takes as its first argument the name of the file created by **`mkmsgs`** and the number of the desired string in the file, counting from 1. You hard-code the second argument, not necessarily in English, in case **`gettxt`** fails to retrieve the message string from the current locale, or the default `"C"("POSIX")` locale.

exstr and srchtxt (System V-specific)

Once you have created the message files for the different locales, you can use the **`exstr(l)`** command to extract the strings from the original source code and replace them with calls to **`gettxt`**. If the name of the source file is **`prog.c`**, the command

```
$ exstr -e prog.c > prog.strings
```

will produce the following output in **`prog.strings`**:

```
prog.c:9:8:::Choose (y/n)
prog.c:11:8:::yes
prog.c:13:8:::no
```

The first three fields in each entry are the file name, the line number in which the string appears in the file, and the character position of the string in the line. You fill in the next two fields with the name of the message file and the index of the string in the file:

```
prog.c:9:8:progmsgs:1:Choose (y/n)
prog.c:11:8:progmsgs:2:yes
prog.c:13:8:progmsgs:3:no
```

Now the command

```
$ exstr -rd prog.c < prog.strings > intl.c
```

will produce in **`intl.c`**

```
#include <stdio.h>

extern char *gettxt();
main()
{
    int yes();

    while(1)
    {
        puts(gettxt(“progmsgs:1”, “Choose (y/n)”));
        if (yes())
            puts(gettxt(“progmsgs:2”, “yes”));
        else
            puts(gettxt(“progmsgs:3”, “no”));
    }
}

static int
yes()
{
    int i, b;

    i = b = getchar();
    while (b != '\n' && b != '\0' && b != EOF)
        b = getchar();
    return(i == 'y');
}
```

The next step in the conversion of these routines is to change the test for the affirmative response. This can also be accomplished using **`gettxt(3c)`** to provide the locale specific affirmative response. This conversion step must be completed manually.

The completed source code would look like this:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <locale.h>
#define RESPLEN 16

char yesstr[RESPLEN];/* assumed to be long enough */
extern char *gettxt();
main()
{
    int yes();

    setlocale(LC_ALL, “”);

    /* save local yes string for subsequent comparisons */
    strcpy(yesstr, gettxt(“progmsgs:2”, “yes”));

    while(1)
    {
        puts(gettxt(“progmsgs:1”, “Choose (y/n)”));
        if (yes())
            puts(yesstr);
        else
            puts(gettxt(“progmsgs:3”, “no”));
    }
}

static int
yes()
{
    int i, b;

    i = b = getchar();
    while (b != '\n' && b != '\0' && b != EOF)
        b = getchar();
    return(i == (int) yesstr[0]);
}
```

The **`srchtxt`** command lets you display or search for text strings in message files installed in a given locale. Among other ways, you might want to use it to see how other programs have translated messages similar to yours. Details of the **`mkmsgs(1)`**, **`exstr(1)`**, **`srchtxt(1)`** and **`gettxt(3C)`** commands can be found in their respective manual pages.

catopen and catclose (X/Open)

As noted in the “Introduction” section at the beginning of this chapter, the X/Open messaging interface is the de facto standard throughout much of Europe, so you can generally count on wider support for it than for the System V-specific version. The principal difference between the interfaces lies in where your message files, or message catalogs, to use the X/Open terminology, are located on the target system. System V-specific message files must be installed in the standard place (**`/usr/lib/locale/`***locale*/**`LC_MESSAGES`**). X/Open message catalogs can be installed anywhere on the system, which means that programs must search their environments for the location of message catalogs at run time.

Users specify message catalog search paths with the `NLSPATH` environment variable. The value of `NLSPATH` is used by the function **`catopen(3C)`** to locate the message catalog named in its first argument. Users will almost always find it convenient to use the `%L` and `%N` substitution fields when setting `NLSPATH`:

```
$ NLSPATH="%L/%N" export NLSPATH
```

In this example, the value of the `LC_MESSAGES` locale category is substituted for `%L`. The value of the first argument to **`catopen`** is substituted for `%N`. So if the name of the catalog given to **`catopen`** is `progmsgs`, and if the environment variable `LC_MESSAGES` is set to `french`, then the value of `NLSPATH` would be **`/usr/lib/locale/french/LC_MESSAGES/progmsgs`** on a System V implementation. For more on `NLSPATH`, see the **`catopen(3C)`** manual page.

The call to **`catopen`** would look like:

```
nl_catd catd;
catd = catopen("progmsgs", NL_CAT_LOCALE);
```

where **`catopen`** and the type `nl_catd` are defined in the header **`<nl_types.h>`**. `catd` is a message catalog descriptor that can be passed as an argument to subsequent calls of the **`catgets`** and **`catclose`** functions. We'll look at **`catgets`** in the next section; **`catclose`** closes the message catalog identified by `catd`. The second argument to **`catopen`** is used as a flag. When set to `NL_CAT_LOCALE`, the `LC_MESSAGES` category is used to locate the message catalog. When this flag is zero, the environment variable `LANG` locates the message catalog.

gencat and catgets (X/Open)

You use the **`gencat(l)`** command to store the strings for a given locale in a catalog that can be read by the message retrieval function **`catgets(3C)`**. The **`gencat`** input file for our example would be:

```
$set
1 votre choix (o/n)
2 oui
3 non
```

The **`$set`** directive specifies that the three messages are members of set 1. A subsequent **`$set`** directive would mean that the following messages are members of set 2, and so on. The messages for each module of an application, then, can be assigned to different sets, making it easier to keep track of message numbers across source files: the messages for any given module will always be numbered consecutively from 1. Note that each message in a **`gencat`** input file must be numbered. For details of the input file syntax, see the **`gencat(1)`** manual page.

If the **`gencat`** input file is named **`fr.str`**, the command

```
$ gencat progmsgs fr.str
```

will generate a catalog called **`progmsgs`** that, when installed in the appropriate directory, can be read by **`catgets`** such that

```
puts(catgets(catd, 1, 1, "Choose (y/n)"));
```

will display

```
Votre choix (o/n)
```

in a French locale. `catd` is the message catalog descriptor returned by the earlier call to **`catopen`**; the second and third arguments are the set and message numbers, respectively, of the string in the catalog. Again, you hard-code the final argument in case **`catgets`** fails. Details on **`gencat(1)`**, **`catgets(3C)`** and **`catopen(3C)`** can be found in their respective manual pages.

The X/Open version of our example follows:

```
#include <stdio.h>
#include <nl_types.h>
#include <string.h>
#include <locale.h>
#define RESPLEN 16

char yesstr[RESPLEN];/* assumed to be long enough */
extern char *catgets();
main()
{
    int yes();
    nl_catd catd;
    setlocale(LC_ALL, “”);
    catd = catopen(“progmsgs”, 0);

    /* save local yes string for subsequent comparisons */
    strcpy(yesstr, catgets(catd, 1, 2, “yes”));

    while(1)
    {
        puts(catgets(catd, 1, 1, “Choose (y/n)”));
        if (yes())
            puts(yesstr);
        else
            puts(catgets(catd, 1, 3, “no”));
    }
}

static int
yes()
{
    int i, b;
    i = b = getchar();
    while (b != '\n' && b != '\0' && b != EOF)
        b = getchar();
    return(i == (int) yesstr[0]);
}
```

%*n*$ Conversion Specifications

Earlier we noted that the `%`*n*`$` form of conversion specification lets you convert the *n*th argument in a **`printf`** or **`scanf`** argument list rather than the next unused argument. We showed you how you could use the feature to control the format of the date and time in different locales, and suggested that `%`*n*`$` had a wider application than that. What we had in mind were cases in which the rules of a given language were built into print statements such as

```
printf( “%s %s\n”,
func == MAP ? “Can't map” : “Can't create”,  pathname);
```

The problem with this code is that it assumes that the verb precedes the object of the sentence, which is not the case in many languages. In other words, even if we rewrote the fragment to use **`gettxt`**, and stored translations of the strings in message files in the appropriate locales, we would still want to use the `%`*n*`$` conversion specification to switch the arguments to **`printf`** depending on the locale. That is, the **`printf`** format string

```
“%1$s %2$s\n”
```

in an English-language locale would be written

```
“%2$s %1$s\n”
```

in a locale in which the object of the sentence precedes the predicate.

kbd

As noted, `kbd` is a STREAMS module that can be programmed to alter or supplement data as it flows between the physical terminal and a user process to produce language-dependent effects. It translates strings in the input stream according to instructions given in tables compiled with the **`kbdcomp`** command. In a European environment these instructions might describe how to compose characters that cannot be entered from terminal keyboards (so-called compose and dead keys), or how to map one key to another (a German user of a `QWERTY` keyboard, for instance, will want the `y` and `z` keys swapped). In an Asian-language environment, where the number of ideograms far exceeds the number of keys on most keyboards, `kbd` might be used to implement a dictionary lookup scheme that converts single-byte input to multibyte characters.

The compiled tables are loaded with the **`kbdload`** command, and attached to user processes with the **`kbdset`** command. Public tables, which are loaded when the system is first brought up, are retained in memory across invocations and made available to all users. Private tables can be defined and loaded by users, but do not remain resident in memory. `kbd` also supports the use of external kernel-resident functions as if they were tables. These functions, which must be registered with the `alp` (“algorithm pool management”) module, are needed for code set conversions that would be difficult or impossible with normal `kbd` tables.

In this section, we'll take a brief look at how you might build a `kbd` table. We provide this material for background only. Most programmers will not have occasion to use `kbd`. For more on the STREAMS facility, see the *STREAMS Modules and Drivers*. Detailed information concerning **`alp(7)`**, **`alpq(1)`**, **`kbd(7)`**, **`kbdcomp(1M)`**, **`kbdload(1M)`**, **`kbdpipe(1)`**, and **`kbdset(1)`** can be found in their respective manual pages.

Building kbd Tables

A `kbd` table typically consists of a map declaration of the form

```
map (name) {
    expressions
}
```

The expressions we'll look at here have the forms

```
keylist (string string)
define (word value)
word (extension result)
```

In the following example of a map for a German-language environment

```
map(german) {
    keylist(yzYZ zyZY)
    define(umlaut '\042')
    umlaut(a '\0344')
    umlaut(o '\0366')
    umlaut(u '\0374')
    define(sharp '\044)
    sharp(ss '\0315')
}
```

the `keylist` expression causes the `y` and `z` keys to be swapped by defining `y` as `z` and vice versa in the lookup table generated by `kbdcomp` for this map. The first `define` expression causes the double quote key (octal 042 in the code set being used) to be defined as a dead key such that whenever it is followed by an `a`, `o` or `u` in the input, it will produce the umlaut version of that character in the code set. The second `define` does the same thing with the sharp key and the characters `ss` to produce the German sharp `s`. Check the **`kbdcomp(1)`** manual page for details. The mappings are summarized below:

Input	Output
y	z
z	y
“a	a
“o	o
“u	u
#ss	b

Internationalization Facilities

Interface Standards

The functions discussed in this chapter are listed below by task in Table 8-3 and Table 8-4. In Table 8-3, pages describing utilities compatible with both the ANSI C and X/Open standards are denoted by an asterisk (*****); pages describing utilities compatible with the X/Open standard only are denoted by a dagger (**†**).

Table 8-3. Routines for Application Programming

Task	Functions
locale specification	**`setlocale(3C)*, environ(5)`**
character classification	**`conv(3C)*, ctype(3C)*`**
multibyte/wide character conversion	**`mbchar(3C)*, mbstring(3C)*`**
wide character handling	**`all (3W) †`**
curses wide character handling	**`all (3X)`**

date and time	**`strftime(3C)*, strftime(4)*`**
	`nl_langinfo(3C) †, langinfo(5) †`
	`getdate(3C)`

numeric and monetary conventions	**`localeconv(3C)*`**
	`nl_langinfo(3C) †, langinfo(5) †`

string collation	**`strcoll(3C)*, strxfrm(3C)*`**
formatted input/output	**`printf(3S)*, scanf(3S)*`**

message handling	**`gencat(1) †, catgets(3C) †,`**
	`catopen(3C) †, nl_types(5) †`
	`exstr(1), gettxt(1), mkmsgs(1),`
	`srchtxt(1), gettxt(3C)`

message management and monitoring	**`lfmt(1), pfmt(1), addsev(3C),`**
	`lfmt(3C), pfmt(3C), setcat(3C),`
	`setlabel(3C)`

Table 8-4. Routines for System Programming and Administration

Task	Functions
character tables	**chrtbl(1M), wchrtbl(1M)**
monetary tables	**montbl(1M)**
collation tables	**colltbl(1M)**
date and time databases	**strftime(4)**
STREAMS	**alpq(1), kbdpipe(1), kbdset(1), pseudo(1), kbdcomp(1M), kbdload(1M), eucioctl(5), iconv(5), alp(7),kbd(7)**

Enhanced Commands

All System V commands are “8-bit clean.” They make no assumptions about the contents of the high-order bit when processing characters. Accordingly, they will work correctly with any standard 7- or 8-bit character set, provided the environment variables LC_CTYPE or LANG have been set to a locale in which the character set is implemented. Similar arrangements have been made for commands that use locale-dependent date and time representations and collation.

Many of these commands have been further enhanced to process multibyte characters, again, provided the environment variables LC_CTYPE or LANG have been set to a locale in which the multibyte character set is implemented. In the manual pages, these characters are described as “supplementary code set characters” in reference to their EUC representation. Check the manual pages for the degree of multibyte support provided.

Finally, many commands have been enhanced to produce locale-specific message output, provided the environment variables LC_MESSAGES or LANG have been set to a locale in which the message output is stored. Note that commands that produce localized output messages use the System V-specific messaging interface.

Table 8-5. Enhanced Commands

Command Name	Multibyte Support	Message Facility
accept	y	y
admin	y	y
ar		y
as		y
at		y
atq		y
atrm		y
awk	y	y

Table 8-5. Enhanced Commands (Cont.)

Command Name	Multibyte Support	Message Facility
`banner`		y
`basename`		y
`batch`		y
`bc`		y
`bfs`	y	
`cal`	y	y
`calendar`	y	y
`cancel`	y	y
`cat`	y	y
`cb`	y	
`cc`	y	y
`cd`		y
`cflow`	y	
`chgrp`		y
`chmod`		y
`chown`		y
`cksum`		y
`cmp`		y
`col`		y
`comm`		y
`compress`		y
`cp`		y
`cpio`	y	y
`cron`		y
`crontab`		y
`csh`	y	y
`csplit`	y	y
`ctccpio`	y	
`cu`	y	y
`cut`	y	y
`cxref`	y	
`date`	y	y

Table 8-5. Enhanced Commands (Cont.)

Command Name	Multibyte Support	Message Facility
`dc`		y
`dd`	y	y
`delta`	y	y
`devattr`		y
`devnm`		y
`df`		y
`diff`	y	y
`diff3`		y
`dircmp`	y	y
`dirname`		y
`disable`	y	
`du`		y
`echo`	y	
`ed`	y	y
`edit`	y	y
`egrep`	y	y
`enable`	y	
`env`	y	y
`ex`	y	y
`expr`	y	y
`fgrep`	y	y
`file`	y	y
`find`	y	y
`fmt`	y	y
`fold`	y	y
`fsdb`	y	
`fuser`		y
`gencat`		y
`get`		y
`getconf`		y
`getopt`	y	y
`getopts`	y	

Table 8-5. Enhanced Commands (Cont.)

Command Name	Multibyte Support	Message Facility
`gettxt`		y
`getty`		y
`grep`	y	y
`head`	y	y
`iconv`		y
`id`		y
`installf`		y
`join`	y	y
`jsh`	y	y
`kill`		y
`ksh`	y	y
`lex`	y	
`line`		y
`ln`		y
`logger`		y
`login`		y
`logname`		y
`lp`	y	y
`lpadmin`		y
`lpfilter`		y
`lpforms`		y
`lpmove`		y
`lpsched`		y
`lpshut`		y
`lpstat`	y	y
`lpusers`		y
`ls`	y	y
`m4`	y	y
`mail`	y	y
`mailx`	y	y
`make`		y
`mesg`		y

Table 8-5. Enhanced Commands (Cont.)

Command Name	Multibyte Support	Message Facility
mkdir		y
mkfifo		y
mkmsgs	y	y
more	y	y
mv		y
mvdir		y
nawk	y	y
newform	y	
newgrp		y
news		y
nl	y	y
nlsadmin	y	
nohup		y
od	y	y
pack		y
page	y	y
passwd		y
paste	y	y
pathchk		y
pcat		y
pg	y	y
pkgadd		y
pkgask		y
pkgchk		y
pkginfo		y
pkgmk		y
pkgparam		y
pkgproto		y
pkgrm		y
pkgtrans		y
pr	y	y
printf		y

Table 8-5. Enhanced Commands (Cont.)

Command Name	Multibyte Support	Message Facility
prs		y
ps		y
pwd		y
read		y
red	y	y
regcmp	y	
reject	y	
removef		y
rfuadmin	y	
rm		y
rmdel		y
rmdir		y
rsh	y	y
sdb	y	
sdiff	y	
sed	y	y
sh	y	y
shl	y	y
sleep		y
sort	y	y
split		y
srchtxt	y	
strip		y
stty		y
sttydefs		y
su		y
sum		y
sysadm	y	
tabs		y
tail	y	y
tar		y
tee		y

Table 8-5. Enhanced Commands (Cont.)

Command Name	Multibyte Support	Message Facility
test	y	y
touch		y
tr	y	y
tty		y
ttyadm		y
ttymon		y
umask		y
uname		y
uncompress		y
unget		y
uniq	y	y
unpack		y
uucleanup	y	
uucp	y	y
uudecode		y
uuencode		y
uulog	y	y
uuname	y	y
uupick		y
uustat		y
uuto		y
uux	y	y
vedit	y	y
vi	y	y
view	y	y
wait		y
wall	y	y
wc	y	y
who		y
write	y	y

Table 8-5. Enhanced Commands (Cont.)

Command Name	Multibyte Support	Message Facility
`xargs`		y
`yacc`	y	y
`zcat`		y

9 Directory and File Management

9 Directory and File Management

Introduction

PowerMAX OS File System functions create and remove files and directories, and inspect and modify their characteristics. Processes use these functions to access files and directories for subsequent I/O operations. One of the most important services provided by an operating system is to maintain a consistent, orderly and easily accessed file system. The file system contains directories of files arranged in a tree-like structure. The file system is simple in structure; nevertheless, it is more powerful and general than those often found even in considerably larger operating systems.

All OS files have a consistent structure to conceal physical properties of the device storing the file, such as the size of a disk track. It is not necessary, nor even possible, to preallocate space for a file. The size of a file is the number of bytes in it, with the last byte determined by the high-water mark of writes to the file. The OS presents each file as a featureless, randomly addressable sequence of bytes arranged as a one-dimensional array of bytes ending with `EOF`.

The file system organizes files and directories into a tree-like structure of directories with files attached anywhere (and possibly multiply) into this hierarchy of directories. Files can be accessed by a “full pathname” or “relative pathname”, have independent protection modes, are automatically allocated and de-allocated, and can be linked across directories.

In the hierarchically arranged directory tree-structure, each directory contains a list of names (character strings) and the associated file index, which implicitly refers to the same device as does the directory. Because directories are themselves files, the naming structure is potentially an arbitrary directed graph. Administrative rules restrict it to have the form of a tree, except that non-directory-files may have several names (entries in various directories).

The same non-directory-file may appear in several directories under possibly different names. This feature is called *linking*; a directory-entry for a file is sometimes called a *link*. PowerMAX OS differs from other systems in which linking is permitted in that all links to a file have equal status. That is, a file does not exist within a particular directory; the directory-entry for a file consists merely of its name and a pointer to the information actually describing the file. Thus, a file exists independently of any directory-entry, although in practice a file is removed along with the last link to it.

Structure of the File System

Types of Files

From the point of view of the user, there are three types of files:

1. regular files
2. directory files
3. special files

The user and user application programs access all three types of files simply as a string of bytes, and must interpret the file appropriately. With the OS, files normally reside on a disk.

Regular Files

Regular files contain whatever information users write onto them (for example, character data, source programs or binary objects). Any file other than a special file or a directory file is a regular file. Every file is a (one-dimensional) array of bytes; the OS imposes no further structure on the contents of files. A file of text consists simply of a string of characters, with the new-line character delimiting lines. Binary files are sequences of words as they appear in memory when the file executes. Some programs operate on files with more structure; for example, the assembler generates, and the loader expects, object files in a specific format. The programs that use files dictate their structure, not the system.

Directory Files

Directory files (also called "directories") provide the mapping (paths) between the names of files and the files themselves. Directories induce a tree-like structure on the file system as a whole to create a hierarchical system of files with directories as the nodes in the hierarchy. A directory is a file that catalogs the files, including directories (sub-directories), directly beneath it in the hierarchy.

Each user owns a directory of files, and may also create sub-directories to contain groups of files conveniently treated together. A directory behaves exactly like a regular file except that only the operating system can write onto it. The OS controls the contents of directories; however, users with permission may read a directory just like any other file.

The operating system maintains several directories for its own use. One of these is the *root-directory*. Each file in the file system can be found by tracing a path from the root-directory through a chain of directories until the desired file is reached. Other system directories contain any programs provided for general use; that is, all *commands*; however, it is by no means necessary that a program reside in one of these directories for it to be executed.

Entries in a directory file are called *links*. A link associates a file-identifier with a filename. Each directory has at least two links, "`.`" (dot) and "`..`" (dot-dot). The link dot refers to the directory itself; while dot-dot refers to the parent of the directory in which dot-dot appears. Programs may read the current-directory using "`.`" without knowing its complete pathname.

The root-directory, which is the top-most node of the hierarchy, has itself as its parent-directory; thus, "`/`" is the pathname of both the root-directory and the parent-directory of the root-directory.

The directory structure is constrained to have the form of a rooted tree. Except for the special entries `.` and `..`, each directory must appear as an entry in exactly one other directory, which is its parent. The reason for this is to simplify the writing of programs that visit sub-trees of the directory structure, and more important, to avoid the separation of portions of the hierarchy. If arbitrary links to directories were permitted, it would be quite difficult to detect when the last connection from the root-directory to a directory was severed.

Special Files

Special files constitute the most unusual feature of the PowerMAX OS file system. Each supported I/O device is associated with at least one special file. Special files are read and written just like regular files, but requests to read or write result in activation of the associated device-handler (driver) rather than the normal file mechanism.

An entry for each special file resides under the directory **`/dev`** although a link may be made to one of these files just as it may to a regular file. For example, to write on magnetic tape one may write on the file **`/dev/mt`**. Special files exist for peripheral devices such as terminal ports, communication links, disk drives, tape drives and for physical main memory. Of course, the active disks and memory special files are protected from indiscriminate access by appropriate read and write permissions.

There are several advantages to treating I/O devices this way:

- file and device I/O are as similar as possible; all I/O is treated uniformly, and the same system calls work on all types of files.
- file and device names have the same syntax and meaning, so that a program expecting a filename as a parameter can be passed a device name.
- the same protection mechanism works on special files, directory files and regular files.

Organization of Files

The file system is made up of a set of regular files, special files, symbolic links, and directories. These components provide a way to organize, retrieve, and manage information electronically. The "File and Device Input/Output" chapter introduced some of the properties of directories and files; this section will review them briefly before discussing how to use them.

- A regular file is a collection of characters stored on a disk. It may contain text for a report or code for a program.
- A special file represents a physical device, such as a terminal or disk.
- A symbolic link is a file that points to another file.

- A directory is a collection of files and other directories (sometimes called subdirectories). Use directories to group files together on the basis of any criteria you choose. For example, you might create a directory for each product that your company sells or for each of your student's records.

The set of all the directories and files is organized into a tree shaped structure. Figure 9-1 shows a sample file structure with a directory called root (**/**) as its source. By moving down the branches extending from root, you can reach several other major system directories. By branching down from these, you can, in turn, reach all the directories and files in the file system.

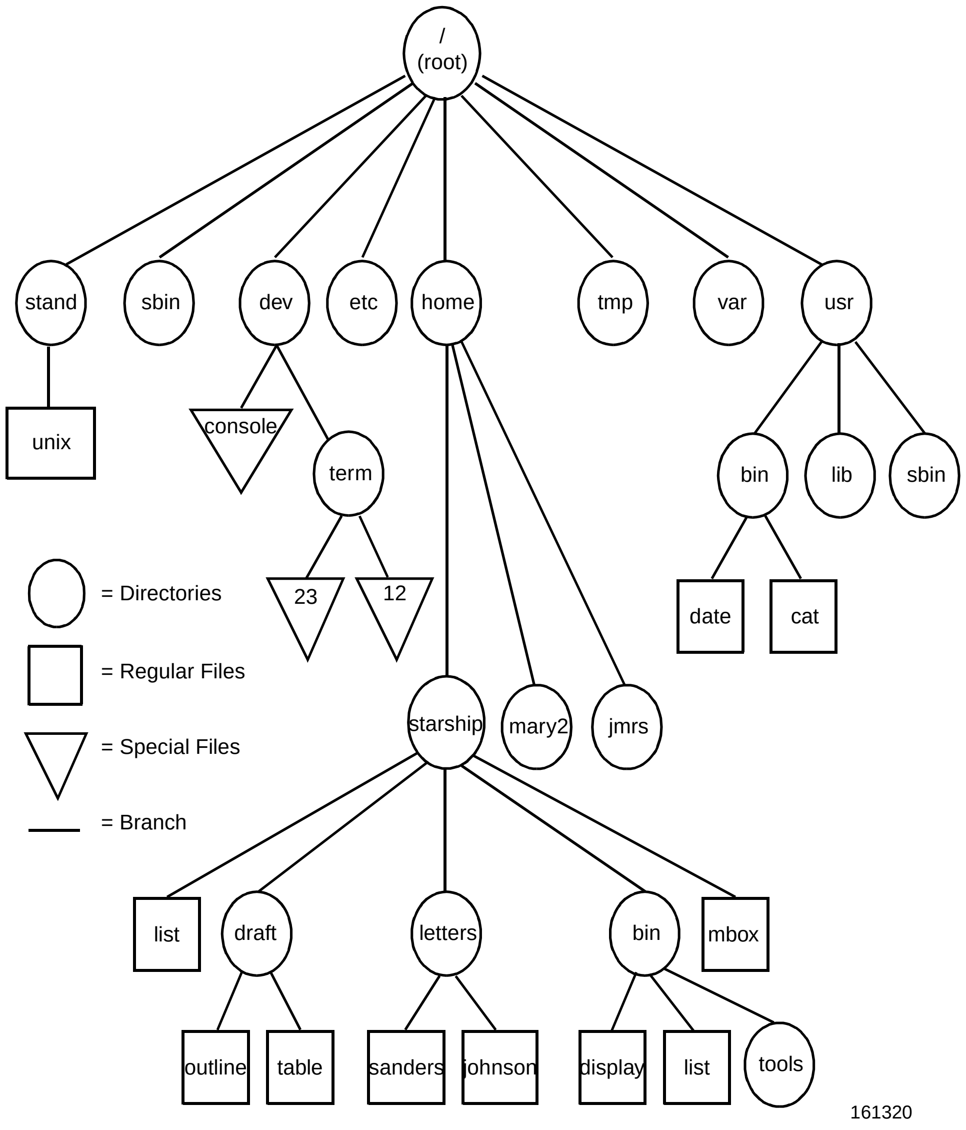


Figure 9-1. A Sample File System

In this hierarchy, files and directories that are subordinate to a directory have what is called a parent/child relationship. This type of relationship is possible for many layers of files and directories. In fact, there is no limit to the number of files and directories you may create in any directory that you own. Neither is there a limit to the number of layers of directories that you may create. Thus, you have the capability to organize your files in a variety of ways, as shown in the preceding figure.

File Naming

Strings of 1 to `{NAME_MAX}` characters may be used to name a regular file, directory file or special file. `{NAME_MAX}` must be at least 14, and the characters may be any from the set of all character values excluding `NULL` and slash, '`/`'. The following are examples of legal directory or file names:

```
memoMEMOsection2ref:list
file.dchap3+4item1-10outline
```

A regular file, special file or directory may have any name that conforms to the following rules:

- All characters other than `/` are legal.
- Non-printing characters including space, tab and backspace, are best avoided. If you use a space or tab in a directory or filename, you must enclose the name in quotation-marks on the command-line.
- Note that it is generally unwise to use "`*`", "`?`", "`!`", "`[`" or "`]`" as part of filenames because of the special meaning given these characters for filename expansion by the command interpreter (see **`system(2)`**). Other characters to avoid are the hyphen, "`<`", "`>`", backslash, single and double quotes, accent grave, vertical bar, caret, curly braces and parentheses.
- Avoid using a `+`, `-` or `.` as the first character in a filename.
- Upper case and lower case characters are distinct to the UNIX system. For example, the system considers a directory (or file) named **`draft`** to be different from one named **`DRAFT`**.

Path Names

The name of a file may take the form of a *pathname*, which is a sequence of directory names separated from one another by "`/`" and ending in a filename. A pathname is a null-terminated character-string starting with an optional slash, "`/`", followed by zero or more directory-names separated by slashes and optionally followed by a filename.

More precisely, a pathname is a null-terminated character-string as follows:

```
<path_name>::=<file_name><path_prefix><file_name>/...
<path_prefix>::=<rtprefix>/<rtprefix> empty
<rtprefix>::=<dirname>/<rtprefix><dirname>/
```

where `<file_name>` is a string of 1 to **`{NAME_MAX}`** significant characters (other than slash and null), and `<dirname>` is a string of 1 to **`{NAME_MAX}`** significant characters (other than slash and null) that names a directory. The result of names not produced by the grammar are undefined. A null string is undefined and may be considered an error. As a limiting case, the pathname "`/`" refers to the root-directory itself. An attempt to create or delete the pathname slash by itself is undefined and may be considered an error. The meanings of "`.`" and "`..`" are defined earlier under the heading "Directory Files."

The sequence of directories preceding the filename is called a *path-prefix*, and if the path-prefix begins with a slash, the search begins in the root-directory. This is called a full pathname.

Full Pathnames

A full pathname (sometimes called an "absolute pathname") starts in the root directory and leads down through a unique sequence of directories to a particular directory or file. Because a full pathname always starts at the root of the file system, its leading character is always a `/` (slash). The final name in a full pathname can be either a file name or a directory name. All other names in the path must be directories. You can use a full pathname to reach any file or directory in the UNIX system in which you are working.

To understand how a full pathname is constructed and how it directs you, consider the following example. Suppose you are working in the **`starship`** directory, located in **`/home`**. You issue the **`pwd`** command and the system responds by printing the full pathname of your working directory: **`/home/starship`**.

Figure 9-2 describes the elements of this pathname:

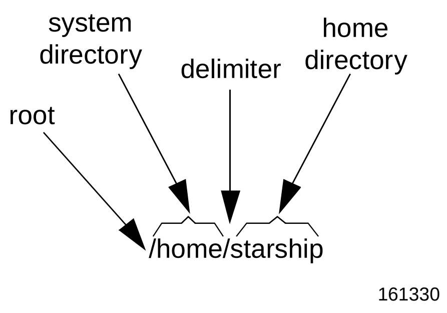


Figure 9-2. Diagram of a Full Pathname

The key for this figure is as follows:

`/` (leading)	=	the slash that appears as the first character in the pathname is the root of the file system
`home`	=	system directory one level below root in the hierarchy to which root points or branches
`/` (subsequent)	=	the next slash separates or delimits the directory names **`home`** and **`starship`**
`starship`	=	current working directory

The following pathname:

```
/usr/bin/send
```

causes a search of the root-directory for directory "**`usr`**", then a search of "**`usr`**" for "**`bin`**", finally to find "**`send`**" in "**`bin`**". The file "**`send`**" may be a directory, regular or special file. A null-prefix (or for that matter, any path-prefix without an initial "**`/`**") causes the search to begin in the current-directory of the user. Thus, the simplest form of pathname, "**`alpha`**", refers to a file found in the current-directory, and the pathname "**`alpha/beta`**" specifies the file named "**`beta`**" in sub-directory "**`alpha`**" of the current-directory. This *relative* pathname allows a user to quickly specify a sub-directory without needing to know (or input) the full pathname.

The dashed lines in Figure 9-3 trace the full path to **`/home/starship`**.

Relative Pathnames

A relative pathname gives directions that start in your current working directory and lead you up or down through a series of directories to a particular file or directory. By moving down from your current directory, you can access files and directories you own.

For example, suppose you are in the directory **`starship`** in the sample system and **`starship`** contains directories named **`draft`**, **`letters`**, and **`bin`** and a file named **`mbox`**. The relative pathname to any of these is simply its name, such as **`draft`** or **`mbox`**. Figure 9-4 traces the relative path from **`starship`** to **`draft`**.

The **`draft`** directory belonging to **`starship`** contains the files **`outline`** and **`table`**. The relative pathname from **`starship`** to the file **`outline`** is **`draft/outline`**.

Figure 9-5 traces this relative path. Notice that the slash in this pathname separates the directory named **`draft`** from the file named **`outline`**. Here, the slash is a delimiter showing that **`outline`** is subordinate to **`draft`**; that is, **`outline`** is a child of its parent, **`draft`**.

So far, the discussion of relative pathnames has covered how to specify names of files and directories that belong to, or are children of, the current directory. You can move down the system hierarchy level by level until you reach your destination. You can also, however, ascend the levels in the system structure or ascend and subsequently descend into other files and directories.

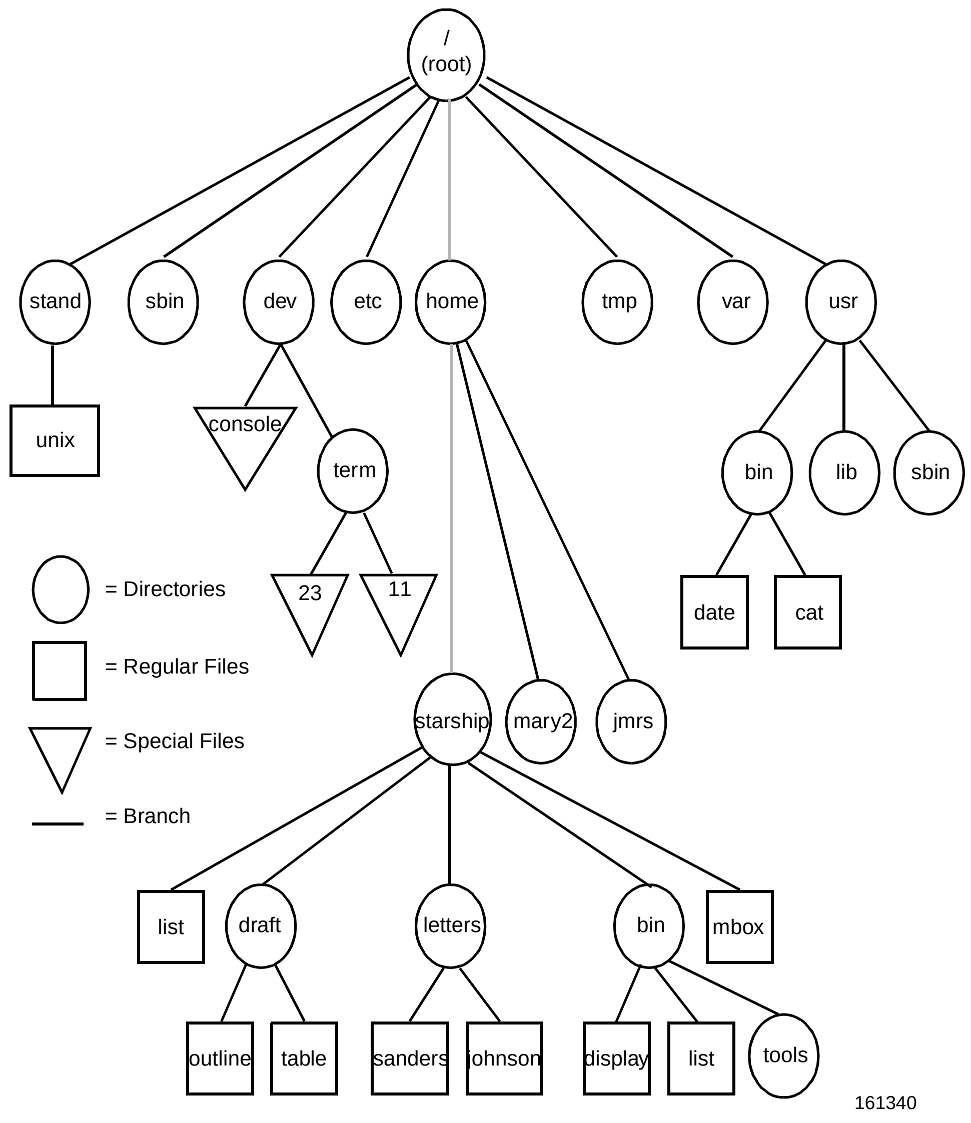


Figure 9-3. Full Pathname of the /home/starship Directory

By moving up from your current directory, you pass through layers of parent directories to the grandparent of all system directories, root. From there you can move anywhere in the file system.

The relative pathname is just one of the mechanisms built into the file system to alleviate the need to use full pathnames. By convention, the path-prefix “ **. .** ” refers to the parent-directory (that is, the directory containing the current-directory), and the path-prefix “ **.** ” refers to the current-directory.

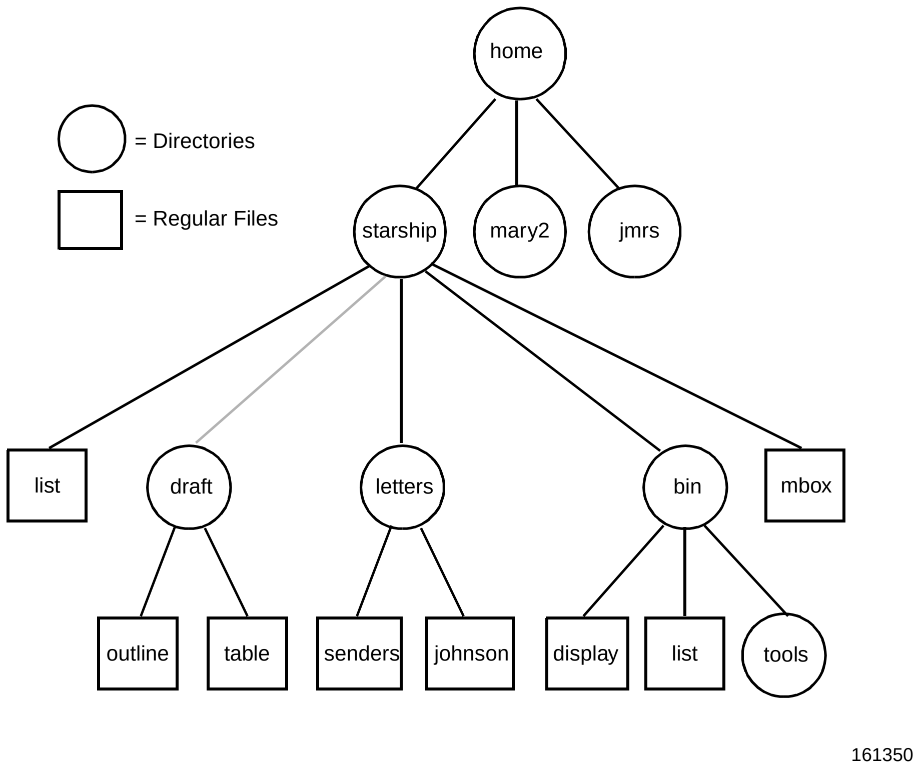


Figure 9-4. Relative Pathname of the draft Directory

A relative pathname begins with one of the following: a directory or file name; a “ `.` ” (pronounced dot), which is a shorthand notation for your current directory; or a “ `..` ” (pronounced dot dot), which is a shorthand notation for the directory immediately above your current directory in the file system hierarchy. The directory represented by “ `..` ” (dot dot) is called the parent directory of `.` (your current directory).

To ascend to the parent of your current directory, you can use the “ `..` ” notation. This means that if you are in the directory named “**`draft`**” in the sample file system, “ `..` ” is the pathname to “**`starship`**”, and “ `../..` ” is the pathname to “**`starship`**”'s parent directory, “**`home`**”.

From “**`draft`**”, you can also trace a path to the file “**`sanders`**” by using the pathname “**`../letters/sanders`**”. The “ `..` ” brings you up to “**`starship`**”. Then the names “**`letters`**” and “**`sanders`**” take you down through the “**`letters`**” directory to the “**`sanders`**” file.

Keep in mind that you can always use a full pathname in place of a relative one.

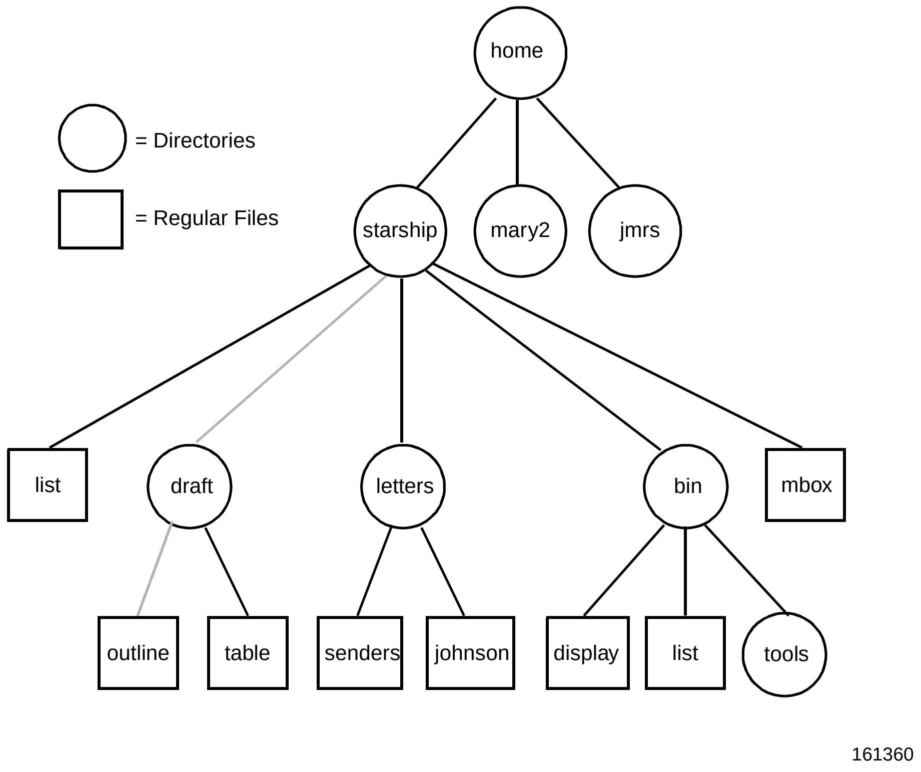


Figure 9-5. Relative Pathname from “starship” to “outline”

Figure 9-6 shows some examples of full and relative pathnames.

Figure 9-6. Example Pathnames

Path Name	Meaning
`/`	full pathname of the root directory
`/usr/bin`	full pathname of the **`bin`** directory that belongs to the **`usr`** directory that belongs to **`root`** (contains most executable programs and utilities)

Figure 9-6. Example Pathnames (Cont.)

Path Name	Meaning
`/home/starship/bin/tools`	full pathname of the **`tools`** directory belonging to the **`bin`** directory that belongs to the **`starship`** directory belonging to **`home`** that belongs to root
`bin/tools`	relative pathname to the file or directory **`tools`** in the directory **`bin`** If the current directory is **`/`**, then the UNIX system searches for **`/usr/bin/tools`**. However, if the current directory is **`starship`**, then the system searches the full path **`/home/starship/bin/tools`**.
`tools`	relative pathname of a file or directory **`tools`** in the current directory.

Moving files to the directory “ **`.`** ” moves them into the current-directory. In addition, files can be linked across directories. Linking a file to the current-directory obviates the need to supply a path-prefix when accessing the file. When created, a process has one current-directory and one root-directory associated with it, which can differ for other processes. See the chapter entitled “Process Management” for more detail on processes.

Symbolic Links

A symbolic link is a special type of file that represents another file. The data in a symbolic link consists of the pathname of a file or directory to which the symbolic link file is linked. The link that is formed is called symbolic to distinguish it from a regular (also called a hard) link such as can be created by using the **`ln(1)`** command. A symbolic link differs functionally from a regular link in three major ways: files from different file systems may be linked together; directories as well as regular files may be symbolically linked by any user; and a symbolic link can be created even if the file it represents does not exist.

In order to understand how a symbolic link works, it is necessary to understand how the UNIX operating system views files. (The following description pertains to files that belong to the standard System V file system type.) The internal representation of a file is contained in an inode, which contains a description of the layout of the file data on disk as well as information about the file, such as the file owner, the access permissions, and the access times. Every file has one inode, but a file may have several names, all of which point to the inode. Each name is called a regular (or hard) link.

When a file is created, an inode is allocated for it, the file contents are stored in data blocks, and an entry is created in a directory. A directory is a file whose data is a sequence of entries, each consisting of an inode number and the name of a file. The inode initially has a link count of one, which means that this file has one name (or one link to it).

We are now in a position to understand the difference between the creation of a regular and a symbolic link. When a user creates a regular link to a file with the **`ln(1)`** command, a new directory entry is created containing a new file name and the inode number of an existing file. The link count of the file is incremented.

In contrast, when a user creates a symbolic link both a new directory entry and a new inode are created. A data block is allocated to contain the pathname of the file to which the symbolic link refers. The link count of the referenced file is not incremented.

Symbolic links can be used to solve a variety of common problems. For example, it frequently happens that a disk partition (such as root) runs out of disk space. With symbolic links, an administrator can create a link from a directory on that file system to a directory on another file system. Such a link provides extra disk space and is, in most cases, transparent to both users and programs.

Symbolic links can also help deal with the built-in pathnames that appear in the code of many commands. Changing the pathnames would require changing the programs and recompiling them. With symbolic links, the pathnames can effectively be changed by making the original files symbolic links that point to new files.

In a shared resource environment like NFS, symbolic links can be very useful. For example, if it is important to have a single copy of certain administrative files, symbolic links can be used to help share them. Symbolic links can also be used to share resources selectively. Suppose a system administrator wants to do a remote mount of a directory that contains sharable devices. These devices must be in **`/dev`** on the client system, but this system has devices of its own so the administrator does not want to mount the directory onto **`/dev`**. Rather than do this, the administrator can mount the directory at a location other than **`/dev`** and then use symbolic links in the **`/dev`** directory to refer to these remote devices. (This is similar to the problem of built-in pathnames since it is normally assumed that devices reside in the **`/dev`** directory.)

Finally, symbolic links can be valuable within the context of the virtual file system (VFS) architecture. With VFS new services, such as higher performance files, events, and network IPC, may be provided on a file system basis. Symbolic links can be used to link these services to home directories or to places that make more sense to the application or user. Thus one might create a database index file in a RAM-based file system type and symbolically link it to the place where the database server expects it and manages it.

NOTE

The phrases "following symbolic links" and "not following symbolic links" as they are used in this document refer to the evaluation of the last component of a pathname. In the evaluation of a pathname, if any component other than the last is a symbolic link, the symbolic link is followed and the referenced file is used in the pathname evaluation. However, if the last component of a pathname is a symbolic link, the link may or may not be followed.

Properties of Symbolic Links

This section summarizes some of the essential characteristics of symbolic links. Succeeding sections describe how symbolic links may be used, based on the characteristics outlined here.

As we have seen above, a symbolic link is a new type of file that represents another file. The file to which it refers may be of any type; a regular file, a directory, a character-special, block-special, or FIFO-special file, or another symbolic link. The file may be on the local system or on a remote system. In fact, the file to which a symbolic link refers does not even have to exist. In particular, the file does not have to exist when the symbolic link is created or when it is removed.

Creation and removal of a symbolic link follow the same rules that apply to any file. To do either, the user must have write permission in the directory that contains the symbolic link. The ownership and the access permissions (mode) of the symbolic link are ignored for all accesses of the symbolic link. It is the ownership and access permissions of the referenced file that are used.

If the Enhanced Security Utilities are installed and running, Mandatory Access Control checks depend on the security level of the referenced file when following a symbolic link, while the level of the symbolic link is ignored. This means that a file is protected at its security level, whether it is accessed directly or through a symbolic link. When access is made to the symbolic link itself without following the link, on the other hand, the level of the symbolic link inode is used to determine access. For example, to display information on a symbolic link using **`ls -l`**, your login level must dominate that of the symbolic link, while the level of the referenced file is ignored. In this case the contents of the symbolic link, which consists of the pathname of the referenced file, is displayed. Since symbolic links can be created to nonexistent files, this does not give away any information about the referenced file itself.

A symbolic link cannot be opened or closed and its contents cannot be changed once it has been created.

If **`/usr/jan/junk`** is a symbolic link to the file **`/etc/passwd`**, in effect the file name **`/etc/passwd`** is substituted for **`junk`** so that when the user executes

```
cat /usr/jan/junk
```

it is the contents of the file **`/etc/passwd`** that are printed.

Similarly, if **`/usr/jan/junk`** is a symbolic link to the file **`../junk2`**, executing

```
cat /usr/jan/junk
```

is the same as executing

```
cat /usr/jan/../junk2
```

or

```
cat /usr/junk2
```

When a symbolic link is followed and brings a user to a different part of the file tree, we may distinguish between where the user really is (the physical path) and how the user got there (the virtual path). The behavior of **`/usr/bin/pwd`**, the shell built-in **`pwd`**, and **`..`** are all based on the physical path. In practical terms this means that there is no way for the user to retrace the path which brought the user to the current position in the file tree.

CAUTION

Other shells may use the virtual path. For example, by default the Korn shell **pwd** uses the virtual path, though there is an option allowing the user to make it use the physical path.

Consider the case shown in Figure 9-7 where **/usr/include/sys** is a symbolic link to **/usr/src/uts/sys.** Here if a user enters

```
    cd /usr/include/sys
```

and then enters **pwd**, the result is

```
    /usr/src/uts/sys
```

If the user then enters `cd ..` followed by **pwd**, the result is

```
    /usr/src/uts
```

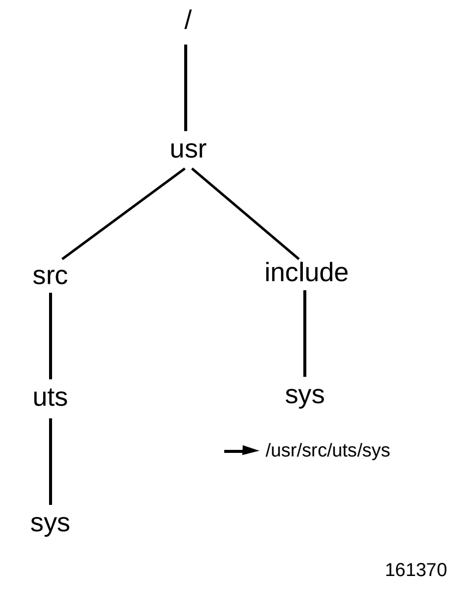


Figure 9-7. File Tree with Symbolic Link

Using Symbolic Links

This section discusses creating, removing, accessing, copying, and linking symbolic links.

Creating Symbolic Links

To create a symbolic link, the new system call **symlink(2)** is used and the owner must have write permission in the directory where the link will reside. The file is created with the user's user-id and group-id but these are subsequently ignored. The mode of the file is created as 0777.

If the Enhanced Security Utilities are installed and running, the file's security level is set to the level of the parent directory. This allows a user who has access permission for the directory to also have access to the symbolic link file.

CAUTION

> No checking is done when a symbolic link is created. There is nothing to stop a user from creating a symbolic link that refers to itself or to an ancestor of itself or several links that loop around among themselves. Therefore, when evaluating a pathname, it is important to put a limit on the number of symbolic links that may be encountered in case the evaluation encounters a loop. The variable `MAXSYMLINKS` is used to force the error `ELOOP` after `MAXSYMLINKS` symbolic links have been encountered. The value of `MAXSYMLINKS` should be at least 20.

To create a symbolic link, the **`ln`** command is used with the **`-s`** option (see **`ln(1)`**). If the **`-s`** option is not used and a user tries to create a link to a file on another file system, a symbolic link will not be created and the command will fail.

The syntax for creating symbolic links is as follows:

> **`ln -s`** *sourcefile1* `[` *sourcefile2* `... ]` *target*

With two arguments:

- *sourcefile1* may be any pathname and need not exist.
- *target* may be an existing directory or a non-existent file.
- If *target* is an existing directory, a file is created in directory *target* whose name is the last component of *sourcefile1* (\`**`basename`** *sourcefile1*\`). This file is a symbolic link that references *sourcefile1*.
- If *target* does not exist, a file with name *target* is created and it is a symbolic link that references *sourcefile1*.
- If *target* already exists and is not a directory, an error is returned.
- *sourcefile1* and *target* may reside on different file systems.

With more than two arguments:

- For each *sourcefile*, a file is created in *target* whose name is *sourcefile* or its last component (\`**`basename`** *sourcefile*\`) and is a symbolic link to *sourcefile*.
- If *target* is not an existing directory, an error is returned.
- Each *sourcefile* and *target* may reside on different file systems.

Examples

The following examples show how symbolic links may be created.

```
ln -s /usr/src/uts/sys /usr/include/sys
```

In this example **/usr/include** is an existing directory. But file **sys** does not exist so it will be created as a symbolic link that refers to **/usr/src/uts/sys**. The result is that when file **/usr/include/sys/x** is accessed, the file **/usr/src/uts/sys/x** will actually be accessed.

This kind of symbolic link may be used when files exist in the directory **/usr/src/uts/sys** but programs often refer to files in **/usr/include/sys**. Rather than creating corresponding files in **/usr/include/sys** that are hard links to files in **/usr/src/uts/sys**, one symbolic link can be used to link the two directories. In this example **/usr/include/sys** becomes a symbolic link that links the former **/usr/include/sys** directory to the **/usr/src/uts/sys** directory.

```
ln -s /etc/group .
```

In this example the *target* is a directory (the current directory), so a file called **group** (**`basename /etc/group`**) is created in the current directory that is a symbolic link to **/etc/group**.

```
ln -s /fs1/jan/abc /var/spool/abc
```

In this example we imagine that **/fs1/jan/abc** does not exist at the time the command is issued. Nevertheless, the file **/var/spool/abc** is created as a symbolic link to **/fs1/jan/abc**. Later, **/fs1/jan/abc** may be created as a directory, regular file, or any other file type.

The following example illustrates the use of more than two arguments:

```
ln -s /etc/group /etc/passwd .
```

The user would like to have the **group** and **passwd** files in the current directory but cannot use hard links because **/etc** is a different file system. When more than two arguments are used, the last argument must be a directory; here it is the current directory. Two files, **group** and **passwd**, are created in the current directory, each a symbolic link to the associated file in **/etc**.

Removing Symbolic Links

Normally, when accessing a symbolic link, one follows the link and actually accesses the referenced file. However, this is not the case when one attempts to remove a symbolic link. When the **rm(1)** command is executed and the argument is a symbolic link, it is the symbolic link that is removed; the referenced file is not touched.

Accessing Symbolic Links

Suppose **abc** is a symbolic link to file **def**. When a user accesses the symbolic link **abc**, it is the file permissions (ownership and access) of file **def** that are actually used; the permissions of **abc** are always ignored. If file **def** is not accessible (that is, either it does not exist or it exists but is not accessible to the user because of access permissions) and a user tries to access the symbolic link **`abc,`** the error message will refer to **`abc,`** not file **`def`**.

Copying Symbolic Links

This section describes the behavior of the **`cp(1)`** command when one or more arguments are symbolic links. With the **`cp(1)`** command, if any argument is a symbolic link, that link is followed. Then the semantics of the command are as described in the *Command Reference*. Suppose the command line is

```
cp sym file3
```

where `sym` is a symbolic link that references a regular file **`test1`** and **`file3`** is a regular file. After execution of the command, **`file3`** gets overwritten with the contents of the file **`test1`**.

If the last argument is a symbolic link that references a directory, then files are copied to that directory. Suppose the command line is:

```
cp file1 sym symd
```

where **`file1`** is a regular file, **`sym`** is a symbolic link that references a regular file **`test1`**, and **`symd`** is a symbolic link that references a directory **`DIR`**. After execution of the command, there will be two new files, **`DIR/file1`** and **`DIR/sym`** that have the same contents as **`file1`** and **`test1`**.

Linking Symbolic Links

This section describes the behavior of the **`ln(1)`** command when one or more arguments are symbolic links. To understand the difference in behavior between this and the **`cp(1)`** command, it is useful to think of a copy operation as dealing with the contents of a file while the link operation deals with the name of a file.

Let us look at the case where the source argument to **`ln`** is a symbolic link. If the **`-s`** option is specified to **`ln`**, the command calls the **`symlink`** system call (see **`symlink(2)`**). **`symlink`** does not follow the symbolic link specified by the source argument and creates a symbolic link to it. If **`-s`** is not specified, **`ln`** invokes the **`link(2)`** system call. **`link`** follows the symbolic link specified by the source argument and creates a hard link to the file referenced by the symbolic link.

For the target argument, **`ln`** invokes a **`stat`** system call (see **`stat(2)`**). If **`stat`** indicates that the target argument is a directory, the files are linked in that directory. Otherwise, if the target argument is an existing file, it is overwritten. This means that if the second argument is a symbolic link to a directory, it is followed, but if it is a symbolic link to a regular file, the symbolic link is overwritten.

For example, if the command line is

```
ln sym file1
```

where **`sym`** is a symbolic link that references a regular file **`foo`**, and **`file1`** is a regular file, **`file1`** is overwritten and hard-linked to **`foo`**. Thus a hard link to a regular file has been created.

If the command is

```
ln -s sym file1
```

where the files are the same as in first example, **file1** is overwritten and becomes a symbolic link to **sym**.

If the command is

```
ln file1 sym
```

where the files are the same as in the first example, **sym** is overwritten and hard-linked to **file1**.

When the last argument is a directory as in

```
ln file1 sym symd
```

where **symd** is a symbolic link to a directory **DIR**, and **file1** and **sym** are the same as in the first example, the file **DIR/file1** is hard-linked to **file1** and **DIR/sym** is hard-linked to **foo**.

Moving Symbolic Links

This section describes the behavior of the **mv(1)** command. Like the **ln(1)** command, **mv(1)** deals with file names rather than file contents. With two arguments, a user invokes the **mv(1)** command to rename a file. Therefore, one would not want to follow the first argument if it is a symbolic link because it is the name of the file that is to be changed rather than the file contents. Suppose that **sym** is a symbolic link to **/etc/passwd** and **abc** is a regular file. If the command

```
mv sym abc
```

is executed, the file **sym** is renamed **abc** and is still a symbolic link to **/etc/passwd**. If **abc** existed (as a regular file or a symbolic link to a regular file) before the command was executed, it is overwritten.

Suppose the command is

```
mv sym1 file1 symd
```

where **sym1** is a symbolic link to a regular file **foo**, **file1** is a regular file, and **symd** is a symbolic link that references a directory **DIR**. When the command is executed, the files **sym1** and **file1** are moved from the current directory to the **DIR** directory so that there are two new files, **DIR/sym1**, which is still a symbolic link to **foo**, and **DIR/file1**.

With the OS, the **mv(1)** command uses the **rename(2)** system call. If the first argument to **rename(2)** is a symbolic link, **rename(2)** does not follow it; instead it renames the symbolic link itself. In previous releases, a file was moved using the **link(2)** system call followed by the **unlink(2)** system call. Since **link(2)** and **unlink(2)** do not follow symbolic links, the result of those two operations is the same as the result of a call to **rename(2)**.

File Ownership and Permissions

The system calls **chmod**, **chown** and **chgrp** are used to change the mode and ownership of a file. If the argument to **chmod**, **chown** or **chgrp** is a symbolic link, the mode and ownership of the referenced file rather than of the symbolic link itself will be changed. (See the section on “Symbolic Links” that follows in this chapter). In such cases, the link is followed.

Once a symbolic link has been created, its permissions cannot be changed. By default, the **chown(1)** and **chgrp(1)** commands change the owner and group of the referenced file. However, a new **-h** option enables the user to change the owner and group of the symbolic link itself. This is useful for removing files from sticky directories.

Using Symbolic Links with NFS

When using symbolic links in an NFS environment, it is important to understand how pathnames are evaluated. The rule by which evaluations are performed is simple. Symbolic links that a client encounters on the server are interpreted in accordance with the client's view of the file tree.

Users on a server system must keep this rule in mind when they create symbolic links in order to avoid problems. The examples that follow illustrate situations in which failure to consider the client's view of the file tree can lead to problems.

In the example shown in Figure 9-8, the server advertises its **/usr** file system as USR. If the server creates the symbolic link **/usr/include/sys** as an absolute pathname to **/usr/src/uts/sys,** evaluation of the link will work as intended as long as a client mounts USR as **/usr**. Another way of saying this is that if the file tree naming conventions are the same on the client and the server, things will work as intended. However, if the client mounts USR as **/mnt/usr**, when the symbolic link **/usr/src/uts/sys** is evaluated, the evaluation will be done with respect to the client's view of the file tree and will not cross the mount point back to the server but will remain on the client. Thus the client will not access the file intended. In this situation the server should create the symbolic link as a relative pathname, **../src/uts/sys**, so that evaluation will produce the desired results regardless of where the client mounts USR.

Figure 9-9 shows another potential problem situation in which the server advertises its **/usr** file system as USR. But in this case the server has a symbolic link from **/usr/src/uts/sys/new.h** to **/foo/usr/src/uts/sys/new.h**. Because the referenced file, **/foo/usr/src/uts/sys/new.h**, is outside of the advertised resource, users on the server can access this file but users on the client cannot. In this example, it would make no difference if the symbolic link was a relative rather than an absolute pathname, because the directory **/foo** on the server is not part of the client's name space. When the system evaluates the symbolic link, it will look for the file on the client and will not follow the link as intended.

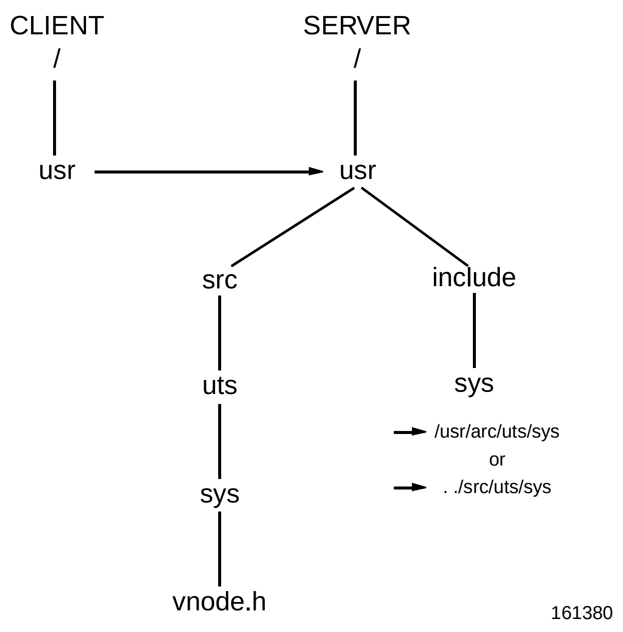


Figure 9-8. Symbolic Links with NFS: Example 1

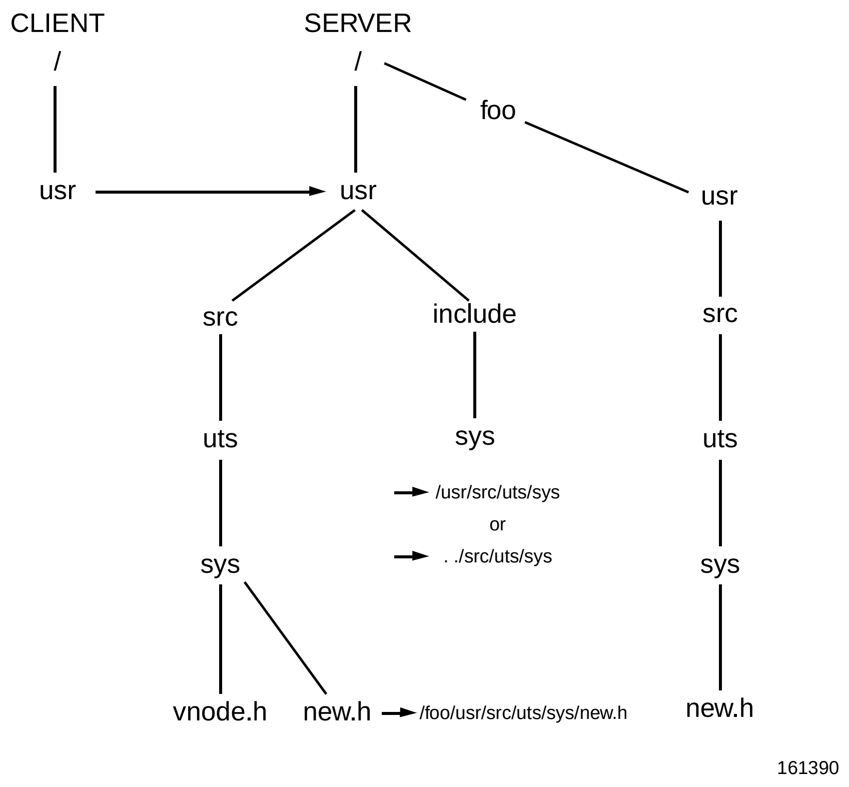


Figure 9-9. Symbolic Links with NFS: Example 2

Archiving Commands

The **`cpio(1)`** command copies file archives usually to or from a storage medium such as tape, disk, or diskette. By default, **`cpio`** does not follow symbolic links, unless the **`-L`** option is used with the **`-o`** and **`-p`** options to indicate that symbolic links should be followed. Note that this option is not valid with the **`-i`** option.

Normally, a user invokes the **`find(1)`** command to produce a list of filenames and pipes this into the **`cpio(1)`** command to create an archive of the files listed. The **`find(1)`** command also has a new option **`-follow`** to indicate that symbolic links should be followed. If a user invokes **`find(1)`** with the **`-follow`** option, then **`cpio(1)`** must also be invoked with its new option **`-L`** to indicate that it too should follow symbolic links.

When evaluating the output from **`find(1)`**, following or not following symbolic links only makes a difference when a symbolic link to a directory is encountered. For example, if **`/usr/jan/symd`** is a symbolic link to the directory **`../joe/test`** and files **`test1`** and **`test2`** are in directory **`/usr/joe/test`**, the output of a **`find`** starting from **`/usr/jan`** includes the file **`/usr/jan/symd`** if symbolic links are not followed, but includes **`/usr/jan/symd/test1`** and **`/usr/jan/symd/test2`** as well as **`/usr/jan/symd`** if symbolic links are followed.

If the user wants to preserve the structure of the directories being archived, it is recommended that symbolic links not be followed on both commands. (This is the default.) When this is done symbolic links will be preserved and the directory hierarchy will be duplicated as it was. If the user is more concerned that the contents of the files be saved, then the user should use the **`-L`** option to **`cpio(1)`** and the **`-follow`** option to **`find(1)`** to follow symbolic links.

CAUTION

The user should take care not to mix modes, that is, the user should either follow or not follow symbolic links for both **`cpio(1)`** and **`find(1).`** If modes are mixed, an archive will be created but the resulting hierarchy created by **`cpio -i`** may exhibit unexpected and undesirable results.

The **`-i`** option to **`cpio(1)`** copies symbolic links as is.

Summary of UNIX System Files & Directories

UNIX system files are organized in a hierarchy; their structure is often described as an inverted tree. At the top of this tree is the root directory, the source of the entire file system. It is designated by a / (slash). All other directories and files descend and branch out from root, as shown in Figure 9-10:

The following section provides brief descriptions of the root directory and the system directories under it, as shown in an earlier figure.

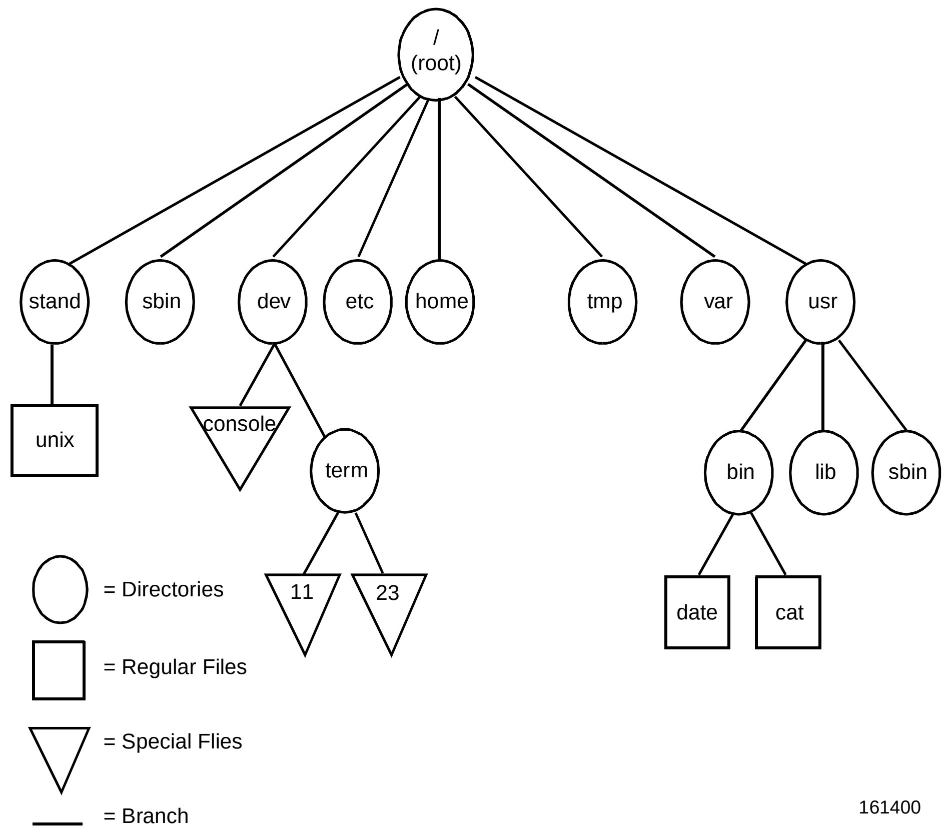


Figure 9-10. Directory Tree from root

UNIX System Directories

`/`	the source of the file system (called the root directory)
`/stand`	contains programs and data files used in the booting process
`/sbin`	contains essential executables used in the booting process and in manual system recovery
`/dev`	contains special files that represent peripheral devices, such as:

`console`	console
`lp`	line printer
`term/*`	user terminal(s)
`dsk/*`	disks

`/etc`	contains machine-specific administrative configuration files and system administration databases
`/home`	the root of a subtree for user directories
`/tmp`	contains temporary files, such as the buffers created for editing a file
`/var`	the root of a subtree for varying files such as log files
`/usr`	contains other directories, including **`lib`** and **`bin`**
`/usr/bin`	contains many executable programs and utilities, including the following: **`cat`**, **`date`**, **`login`**, **`grep`**, **`mkdir`**, **`who`**.
`/usr/lib`	contains libraries for programs and languages

Directories and Files

This section describes:

- Directories and files that are important for administering a system
- Directories that are new for this software release
- The reorganization of the directory structure introduced in this release
- The new organization of the root file system, and significant directories mounted on root

CAUTION

To maintain a secure environment, do not change the file or directory permissions from those assigned at the time of installation.

Directories in root

The **`/`** (root) file system contains executables and other files necessary to boot and run the system. The directories of the root file system are explained next.

`/bck`

The **`/bck`** directory is used to mount a backup file system for restoring files.

`/dev`

The **`/dev`** directory contains block and character special files that are usually associated with hardware devices or STREAMS drivers.

/etc

The **/etc** directory contains machine-specific configuration files and system administration databases.

/export

The **/export** directory contains the default root of the exported file system tree.

/home

The **/home** directory contains user directories.

/install

The **/install** directory is used by the packaging commands to mount add-on packages for installation and removal (**/install** file system).

/lost+found

The **/lost+found** directory is used by **fsck** to save disconnected files and directories.

/mnt

The **/mnt** directory is used to mount file systems for temporary use.

/opt

The **/opt** directory is the mount point from which add-on application packages are installed.

/proc

The **/proc** directory is the mount point of the **proc** file system which provides information on the system's processes.

/save

The **/save** directory is used by packaging commands for saving data.

/sbin

The **/sbin** directory contains executables used in the booting process and in manual recovery from a system failure.

/stand

The **/stand** directory contains standalone programs, which include **boot**.

/tmp

The **/tmp** directory contains temporary files.

/usr

The **/usr** directory is the mount point of the **usr** file system.

/var

The **/var** directory is the mount point of the **var** file system. It contains those files and directories that vary from machine to machine, such as **tmp**, **spool** and **mail**. The **/var** file system also contains administrative directories such as **/var/adm** and **/var/opt**, the latter is installed by application packages.

Directories in /etc

This section describes the directories under the **/etc** directory, which contain machine-specific configuration files and system administration databases.

/etc/bkup

This directory contains machine-specific files and directories for the extended backup and restore operations. Also contained here are files and directories that allow restore operations to be performed from single-user mode (system state 1).

/etc/bkup/method

This directory contains files that describe all the extended backup and restore methods currently used on your computer.

/etc/conf

The **/etc/conf** directory contains files that define the hardware drivers, software drivers, and system parameters used to build the UNIX system file **/stand/unix**.

/etc/cron.d

This directory contains administrative files for controlling and monitoring **cron** activities.

/etc/default

This directory contains files that assign default values to certain system parameters.

/etc/fs

This directory contains file-type specific utilities.

/etc/init.d

This directory contains executable files used in upward and downward transitions to all system states. These files are linked to files beginning with **S** (start) or **K** (stop) in **/etc/rc***n***.d**, where *n* is the appropriate system state. Files are executed from the **/etc/rc***n***.d** directories.

/etc/lp

This directory contains the configuration files and interface programs for the LP print service.

/etc/mail

This directory contains files used in administering the electronic mail system.

/etc/mail/lists

This directory contains files, each of which contains a mail alias. The name of each file is the name of the mail alias that it contains. (See the **mailx(1)** command for a description of the mail alias format.)

/etc/rc.d

This directory contains executable files that perform the various functions needed to initialize the system to system state 2. The files are executed when **/usr/sbin/rc2** is run. (Files contained in this directory before UNIX System V Release 3.0 were moved to **/etc/rc2.d**. This directory is maintained only for compatibility reasons.)

/etc/rc0.d

This directory contains files executed by **/usr/sbin/rc0** for transitions to system states 0, 5, and 6. Files in this directory are linked from the **/etc/init.d** directory, and begin with either a K or an S. K shows processes that are stopped, and S shows processes that are started when entering system states 0, 5, or 6.

/etc/rc1.d

This directory contains files executed by **/usr/sbin/rc1** for transitions to system state 1. Files in this directory are linked from the **/etc/init.d** directory, and begin with either a K or an S. K shows processes that should be stopped, and S shows processes that should be started when entering system state 1.

/etc/rc2.d

This directory contains files executed by **/usr/sbin/rc2** for transitions to system state 2. Files in this directory are linked from the **/etc/init.d** directory, and begin with either a K or an S. K shows processes that should be stopped, and S shows processes that should be started when entering system state 2.

/etc/rc3.d

This directory contains files executed by **/usr/sbin/rc3** for transitions to system state 3 (multi-user mode). Files in this directory are linked from the **/etc/init.d** directory, and begin with either a K or an S. K shows processes that should be stopped, and S shows processes that should be started when entering system state 3.

/etc/saf

This directory contains files and subdirectories used by the Service Access Facility. The following commands in **/usr/sbin** use **/etc/saf** subdirectories for data storage and retrieval: **nlsadmin**, **pmadm** and **sacadm**. The following files are included:

_sactab	A list of port monitors to be started by the Service Access Controller (SAC). Each port monitor listed in this table has a **_pmtab** file in the **/etc/saf/***pmtag* directory, where *pmtag* is the tag of this port monitor (such as **/etc/saf/starlan** for the starlan port monitor).
_sysconfig	The configuration script used to modify the environment for the Service Access Facility.

/etc/save.d

This directory contains files used by the **sysadm** command for backing up data on floppy diskettes. The following files are included:

except	A list of the directories and files that should not be copied as part of a backup is maintained in this file.
timestamp/...	The date and time of the last backup (volume or incremental) is maintained for each file system in the **/etc/save.d/timestamp** directory.

Files in /etc

The following files are used in machine-specific configuration and system administration databases.

/etc/bkup/bkexcept.tab

This file contains a list of files to be excluded from an incremental backup.

/etc/bkup/bkhist.tab

This file contains information about the success of all backup attempts.

/etc/bkup/bkreg.tab

This file contains instructions to the system for performing backup operations on your computer.

/etc/bkup/bkstatus.tab

This file contains the status of backup operations currently taking place.

/etc/bkup/rsmethod.tab

This file contains descriptions of the types of objects that may be restored using the full or partial restore method.

/etc/bkup/rsnotify.tab

This file contains the electronic mail address of the operator to be notified whenever restore requests require operator intervention.

/etc/bkup/rsstatus.tab

This file contains a list of all restore requests made by users of your computer.

/etc/bkup/rsstrat.tab

This file specifies a strategy for selecting archives when handling restore requests. In completing restore operations for these requests, the backup history log is used to navigate through the backup tape to find the desired files and or directories.

/etc/bupsched

This file contains the backup schedule.

/etc/d_passwd

This file contains a list of programs that will require dial-up passwords when run from **login.** Each line in the file is formatted as

program:*encrypted_password*:

where *program* is the full path to any programs into which a user can log in and run. The password referred to in the *encrypted_password* is the one that will be used by the dial-up password program. This password must be entered before the user is given the login prompt. It is used in conjunction with the file **/etc/dialups.**

/etc/default/cron

This file contains the default status (enable or disable) for the CRONLOG operation.

/etc/default/login

This file may contain the following parameters that define a user's login environment:

ALTSHELL	Alternate shell status available to users (yes or no).
CONSOLE	Root login allowed only at the console terminal.
HZ	Number of clock ticks per second.
IDLEWEEKS	Number of weeks a password may remain unchanged before the user is denied access to the system.
PASSREQ	Password requirement on logins (yes or no).
PATH	User's default PATH.
SUPATH	Root's default PATH.

`TIMEOUT`	Number of seconds allowed for logging in before a timeout occurs.
`TIMEZONE`	Time zone used within the user's environment.
`ULIMIT`	File size limit (`ulimit`).
`UMASK`	User's value for `umask`.

`/etc/default/passwd`

This file contains the following information about the length and aging of user passwords:

`MINWEEKS`	Minimum number of weeks before a password can be changed.
`MAXWEEKS`	Maximum number of weeks a password can be unchanged.
`PASSLENGTH`	Minimum number of characters in a password.
`WARNWEEKS`	Number of weeks before a password expires that the user is to be warned.
`SULOG`	A pathname that identifies a file in which a log of all **`su`** attempts may be created.
`CONSOLE`	Pathnames of the console on which are broadcast messages notifying you whenever someone attempts to **`su`** `root`.
`PATH`	`PATH` used for **`su`** users.
`SUPATH`	`PATH` used for **`su`** `root` users.

`/etc/device.tab`

This file is the device table. It lists the device alias, path to the vnode, and special attributes of every device connected to the computer.

`/etc/devlock.tab`

This file is created at run time and lists the reserved (locked) devices. Device reservations do not remain intact across system reboots.

`/etc/saf/`*pmtag*`/_config`

This file contains a configuration script used to customize the environment for the port monitor tagged as *pmtag* (such as **`/etc/saf/starlan/_config`** for the starlan port monitor). Port monitor configuration scripts are optional.

`/etc/dgroup.tab`

This file lists the group or groups to which a device belongs.

/etc/dialups

This file contains a list of terminal devices that cannot be accessed without a dial-up password. It is used in conjunction with the file **/etc/d_passwd**.

/etc/group

This file describes each user group to the system. An entry is added for each new group with the **groupadd** command.

/etc/inittab

This file contains instructions for the **/sbin/init** command. The instructions define the processes created or stopped for each initialization state. Initialization states are called system states or run states. By convention, system state 1 (or S or s) is single-user mode; system states 2 and 3 are multi-user modes. (See **inittab(4)** in the *System Files and Devices Reference* for additional information.)

/etc/mail/mailcnfg

This file permits per-site customizing of the mail subsystem. See the **mailcnfg(4)** manual page in the *System Files and Devices Reference* and in the *Network Administration* manuals.

/etc/mail/mailsurr

This file lists actions to be taken when mail containing particular patterns is processed by **mail**. This can include routing translations and logging. See the **mailsurr(4**) manual page in the *System Files and Devices Reference.*

/etc/mail/mailx.rc

This file contains defaults for the **mailx** program. It may be added by the system administrator. See **mailx(1)**.

/etc/mail/notify and **/etc/mail/notify.sys**

These files are used by the **notify** program to determine the location of users in a networked environment and to establish systems to use in case of file error.

/etc/motd

This file contains the message of the day. The message of the day is displayed on a user's screen after that user has successfully logged in. (The commands that produce this output on the screen are in the **/etc/profile** file.) This message should be kept short and to the point. The **/var/news** files should be used for lengthy messages.

/etc/passwd

This file identifies each user to the system. An entry is automatically added for each new user with the **useradd** command, removed with the **userdel** command, and modified with the **usermod** command.

/etc/profile

This file contains the default profile for all users. The standard (default) environment for all users is established by the instructions in the **/etc/profile** file. The system administrator can change this file to set options for the `root` login. For example, the six lines of code shown in Figure 9-11 can be added to the **/etc/profile**. This code defines the erase character, automatically identifies the terminal type, and sets the `TERM` variable when the login ID is `root`.

```
if [ ${LOGNAME} = root ]
    then
        stty echoe
        echo “Terminal: c”; read TERM           export TERM
    fi
```

Figure 9-11. Excerpt from /etc/profile

/etc/dfs/dfstab

This file specifies the distributed file system resources from your machine that are automatically shared to remote machines when entering system state 3. Each entry in this file should be a **share(1M)** command line.

/etc/saf/*pmtag*/_pmtab

This is the administrative file for the port monitor tagged as *pmtag*. It contains an entry for each service available through the *pmtag* port monitor.

/etc/saf/_sactab

This file contains information about all port monitors for which the Service Access Controller (SAC) is responsible.

/etc/saf/_sysconfig

This file contains a configuration script to customize the environments for all port monitors on the system. This per-system configuration file is optional.

/etc/TIMEZONE

This file sets the time zone shell variable `TZ`. The `TZ` variable is initially established for the system via the **sysadm setup** command. The `TZ` variable in the **TIMEZONE** file is changed by the **sysadm timezone** command. The `TZ` variable can be redefined on a user (login) basis by setting the variable in the associated **.profile**. The **TIMEZONE** file is executed by **/usr/sbin/rc2**. (See **timezone(4)** in the *System Files and Devices Reference* for more information.)

/etc/ttydefs

This file contains information used by ttymon port monitor to set the terminal modes and baud rate for a TTY port.

/etc/vfstab

This file provides default values for file systems and remote resources. The following information can be stored in this file:

- The block and character devices on which file systems reside
- The resource name
- The location where a file system is usually mounted
- The file system type
- Information on special mounting procedures

These defaults do not override command line arguments that have been entered manually. (See **mountall(1M)** in the *Command Reference* for additional information.) Figure 9-12 shows a sample of this file.

```
#special           fsckdev        mountp              fstype    fsckpass   automnt      mntopts
/dev/root          /dev/rroot     /                   ufs       0          yes          -
/dev/swap          -              -                   swap      -          yes          -
/dev/usr           /dev/rusr      /usr                ufs       1          yes          -
/dev/var           /dev/rvar      /var                ufs       0          yes          -
/proc              -              /proc               proc      -          no           -
/dev/fd            -              /dev/fd             fdfs      -          no           -
/system/processor  -              /system/processor   profs     -          no           -
```

Figure 9-12. Sample /etc/vfstab File

Directories in /usr

This section describes the directories in the **/usr** file system. The **/usr** file system contains architecture-dependent and architecture-independent files and system administration databases that can be shared.

/usr/adm

This directory contains administrative files.

/usr/bin

This directory contains public commands and system utilities.

/usr/ccs

This directory contains compilation systems executables, libraries, and miscellaneous files.

/usr/include

This directory contains public header files for C programs.

`/usr/lib`

This directory contains public libraries, daemons, and architecture dependent databases.

`/usr/lib/lp`

This directory contains the directories and files used in processing requests to the LP print service.

`/usr/lib/mail`

This directory contains directories and files used in processing mail.

`/usr/lib/mail/surrcmd`

This directory contains programs necessary for mail surrogate processing.

`/usr/lost+found`

This directory contains orphaned files found by **`fsck(1M)`**.

`/usr/sadm`

This directory contains **`sysadm`** administration files.

`/usr/sadm/bkup`

This directory contains executables for the extended backup and restore services.

`/usr/sbin`

This directory contains executables used for system administration.

`/usr/share`

This directory contains architecture independent files that can be shared.

`/usr/share/lib`

This directory contains architecture independent databases.

`/usr/share/man`

This directory contains system manual pages.

`/usr/sadm/skel`

This directory contains the files and directories built when using the **`useradd`** command with the **`-m`** argument. All directories and files under this location are built under the **`$HOME`** location for the new user.

Files in /usr

This section describes the files in the **/usr** directories, which contain architecture-dependent and architecture-independent files and system administrative databases that can be shared.

/usr/sbin/rc0

This file contains a shell script executed by **/usr/sbin/shutdown** for transitions to single-user state, and by **/sbin/init** for transitions to system states 0, 5, and 6. Files in the **/etc/shutdown.d** and **/etc/rc0.d** directories are executed when **/usr/sbin/rc0** is run. The file **K00ANNOUNCE** in **/etc/rc0.d** prints the message `System services are now being stopped.` Any task that you want executed when the system is taken to system states 0, s, 5, or 6 is done by adding a file to the **/etc/rc0.d** directory.

/usr/sbin/rc1

This file contains a shell script executed by **/sbin/init** for transitions to system state 1 (single-user state). Executable files in the **/etc/rc.d** directory and any executable files beginning with S or K in the **/etc/rc1.d** directories are executed when **/usr/sbin/rc1** is run. All files in **rc1.d** are linked from files in the **/etc/init.d** directory. Other files may be added to the **/etc/rc1.d** directory as a function of adding hardware or software to the system.

/usr/sbin/rc2

This file contains a shell script executed by **/sbin/init** for transitions to system state 2 (multi-user state). Executable files in the **/etc/rc.d** directory and any executable files beginning with S or K in the **/etc/rc2.d** directories are executed when **/usr/sbin/rc2** is run. All files in **rc2.d** are linked from files in the **/etc/init.d** directory. Other files may be added to the **/etc/rc2.d** directory as a function of adding hardware or software to the system.

/usr/sbin/rc3

This file is executed by **/sbin/init**. It executes the shell scripts in **/etc/rc3.d** for transitions to a distributed file system mode (system state 3).

/usr/sbin/rc6

This shell script is run for transitions to system state 6 (for example, using **shutdown -i6**). If the kernel needs reconfiguring, the **/sbin/buildsys** script is run. If reconfiguration succeeds, **/usr/sbin/rc6** reboots without running diagnostics. If reconfiguration fails, it spawns a shell.

/usr/sbin/shutdown

This file contains a shell script to shut down the system gracefully in preparation for a system backup or scheduled downtime. After stopping all nonessential processes, the **shutdown** script executes files in the **/etc/shutdown.d** directory by calling **/usr/sbin/rc0** for transitions to system state

1 (single-user state). For transitions to other system states, the **`shutdown`** script calls **`/sbin/init`**.

`/usr/share/lib/mailx/mailx.help, /usr/share/lib/mailx/mailx.help.`

Help files for **`mailx`**. The file **`mailx.help.~`** contains help messages for **`mailx`**'s tilde commands. See **`mailx(1)`** in the *Command Reference.*

Directories in /var

This section describes the directories of the **`/var`** directory, which contain files and directories that vary from machine to machine.

`/var/adm`

This directory contains system logging and accounting files.

`/var/crashfiles`

This directory contains the system crash files dumped by the **`savecore(1M)`** utility.

`/var/cron`

This directory contains the **`cron`** log file.

`/var/iaf`

This directory contains log files for the identification and authentication facility.

`/var/lost+found`

This directory contains orphaned files found by **`fsck(1M)`**.

`/var/lp`

This directory contains log files for the LP print service.

`/var/mail`

This directory contains subdirectories and mail files that users access with the **`mail(1)`** and **`mailx(1)`** commands.

`/var/mail/:saved`

This directory contains temporary storage for mail messages while **`mail`** is running. Files are named with the user's ID while they are in **`/var/mail`**.

`/var/news`

This directory contains news files. The file names are descriptive of the contents of the files; they are analogous to headlines. When a user reads the news, using the **`news`** command, an empty file named **`.news_time`** is created in his or her login directory. The date (time) of this file is used by the **`news`** command to determine if a user has read the latest news file(s).

/var/opt

This directory is created and used by application packages.

/var/options

This directory contains a file (or symbolic link to a file) that identifies each utility installed on the system. This directory also contains information created and used by application packages (such as temporary files and logs).

/var/preserve

This directory contains backup files for **vi** and **ex**.

/var/sadm

This directory contains logging and accounting files for the backup and restore services, software installation utilities, and package management facilities.

/var/sadm/pkg

This directory contains data directories for installed software packages.

/var/sa

This directory contains data files used by **sar(1M)**.

/var/saf

This directory contains log files for the Service Access Facility.

/var/spool

This directory contains temporary spool files.

/var/spool/cron/crontabs

This directory contains **crontab** files for the **adm**, **root** and **sys** logins. Users whose login IDs are in the **/etc/cron.d/cron.allow** file can establish their own **crontab** file using the **crontab** command. If the **cron.allow** file does not exist, the **/etc/cron.d/cron.deny** file is checked to determine if the user should be denied the use of the **crontab** command.

As **root**, you can use the **crontab** command to make the desired entries. Revisions to the file take effect at the next reboot. The file entries support the **calendar** reminder service and the Basic Networking Utilities. Remember, you can use the **cron** command to decrease the number of tasks you perform with the **sysadm** command; include recurring and habitual tasks in your **crontab** file. (See **crontab(1)** in the *Command Reference* for additional information.)

/var/spool/lp

This directory contains temporary print job files.

/var/spool/smtpq

This directory contains Simple Mail Transfer Protocol (SMTP) directories and log files. Directories named ***host*** contain messages spooled to be sent to that host. Files named **LOG.***n* contain the logs from the past seven days (Sunday's log is called **log.0**). The current day's log is simply **LOG**.

/var/spool/uucp

This directory contains files to be sent by **uucp**.

/var/spool/uucppublic

This directory contains files received by **uucp**.

/var/tmp

This directory contains temporary files.

/var/uucp

This directory contains logging and accounting files for **uucp**.

/var/yp

This directory contains utilities and data used for the Network Information Service (NIS).

Files in /var

This section describes the files in the **/var** directories, which contain information that varies from machine to machine.

/var/adm/errfile

This file contains the error log for **errdemon(1M)**.

/var/adm/lastlog

This file contains the last login time for all users.

/var/adm/spellhist

If the Spell Utility is installed, this file contains a history of all words that the **spell** command fails to match. Periodically, this file should be reviewed for words that you can add to the dictionary. Clear the **spellhist** file after reviewing it. (Refer to **spell(1)** for information on adding words to the dictionary, cleaning up the **spellhist** file, and other commands that can be used with the Spell Utility.)

/var/adm/syslog

This is the default log file from the **syslogd** (system message daemon). It is set up in **/etc/syslog.conf**.

/var/adm/utmp

This file contains information on the current system state. This information is accessed with the **who** command.

/var/adm/utmpx

This file contains information similar to that in the **/var/adm/utmp** file, along with a record of the remote host.

/var/adm/wtmp

This file contains a history of system logins. The owner and group of this file must be **adm**, and the access permissions must be 664. Each time **login** is run this file is updated. As the system is accessed, this file increases in size. Periodically, this file should be cleared or truncated. The command line **>/var/adm/wtmp** when executed by **root** creates the file with nothing in it. The following command lines limit the size of the **/var/adm/wtmp** file to the last 3600 characters in the file:

```
# tail -3600c /var/adm/wtmp > /var/tmp/wtmp
# mv /var/tmp/wtmp /var/adm/wtmp
#
```

The **/usr/sbin/cron**, **/usr/sbin/rc0**, or **/usr/sbin/rc2** command can be used to clean up the **wtmp** file. You can add the appropriate command lines to the **/var/spool/cron/crontabs/root** file or add shell command lines to directories such as **/etc/rc2.d**, **/etc/rc3.d**, and so on.

/var/adm/wtmpx

This file contains information similar to that in the **/var/adm/wtmp** file, along with a record of the remote host.

/var/adm/loginlog

If this file exists, it is a text file that contains one entry for each group of five consecutive unsuccessful attempts to log in to the system.

/var/adm/sulog

This file contains a history of substitute user (**su**) command usage. As a security measure, this file should not be readable by `others`. The **/var/adm/sulog** file should be truncated periodically to keep the size of the file within a reasonable limit. The **/usr/sbin/cron**, the **/usr/sbin/rc0**, or the **/usr/sbin/rc2** command can be used to clean up the **sulog** file. You can add the appropriate command lines to the **/var/spool/cron/crontabs/root** file or add shell command lines to directories such as **/etc/rc2.d**, **/etc/rc3.d**, and so on. The following command lines limit the size of the log file to the last 100 lines in the file:

```
# tail -100 /var/adm/sulog > /var/tmp/sulog
# mv /var/tmp/sulog /var/adm/sulog
# /var/cron/log
```

This file contains a history of all actions taken by **/usr/sbin/cron**. The **/var/cron/log** file should be truncated periodically to keep the size of the file within a reasonable limit. The **/usr/sbin/cron**, **/usr/sbin/rc0**, or **/usr/sbin/rc2** command can be used to clean up the **/var/cron/log** file. You can add the appropriate command lines to the **/var/spool/cron/crontabs/root** file or add shell command lines in the following directories (as applicable): **/etc/rc2.d**, **/etc/rc3.d**, (and so on). The following command lines limit the size of the log file to the last 100 lines in the file:

```
# tail -100 /var/cron/log > /var/tmp/log
# mv /var/tmp/log /var/cron/log
#
```

This file contains a process log used when troubleshooting a backup operation.

/var/sadm/bkup/logs/bkrs

This file contains a process log used when troubleshooting a backup or restore operation for which a method was not specified.

/var/sadm/bkup/logs/rslog

This file contains a process log used when troubleshooting a restore operation.

/var/sadm/bkup/toc

This file contains table of contents entries created by a backup method.

File Access Controls

When the **ls -l** command displays the contents of a directory, the first column of output describes the "mode" of the file. This information tells you not only what type of file it is, but who has permission to access it. This first field is 10 characters long. The first character defines the file type and can be one of the following types:

Figure 9-13. File Types

Type	Symbol
Text, programs, etc.	-
Directories	d
Character special	c
Block special	b
FIFO (named pipe) special	p
Symbolic links	l

Using this key to interpret the previous screen, you can see that the **`starship`** directory contains three directories and two regular disk files.

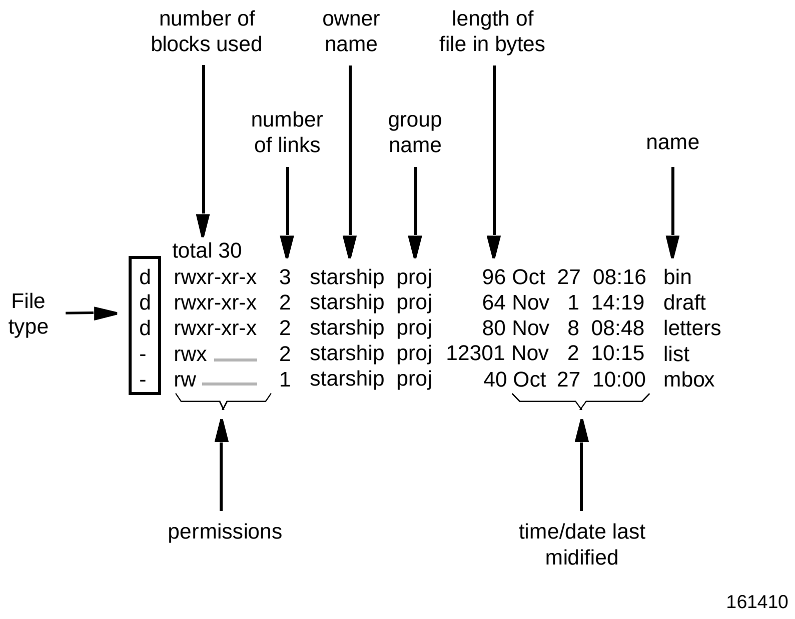
The next several characters, which are either letters or hyphens, identify who has permission to read and use the file or directory.

The following number is the link count. For a file, this equals the number of users linked to that file. For a directory, this number shows the number of directories immediately under it plus two (for the directory itself and its parent directory).

Next, the login name of the file's owner appears (here it is **`starship`**), followed by the group name of the file or directory (**`project`**).

The following number shows the length of the file or directory entry measured in units of information (or memory) called bytes. The month, day, and time that the file was last modified is given next. Finally, the last column shows the name of the directory or file.

Figure 9-14 identifies each column in the rows of output from the **`ls -l`** command.



```
 total 30
d rwxr-xr-x  3  starship  proj     96 Oct  27  08:16  bin
d rwxr-xr-x  2  starship  proj     64 Nov   1  14:19  draft
d rwxr-xr-x  2  starship  proj     80 Nov   8  08:48  letters
- rwx        2  starship  proj  12301 Nov   2  10:15  list
- rw         1  starship  proj     40 Oct  27  10:00  mbox
```

Figure 9-14. Description of Output Produced by the ls -l Command

File Protection

Because the UNIX operating system is a multi-user system, you usually do not work alone in the file system. System users can follow pathnames to various directories and read and use files belonging to one another, as long as they have permission to do so.

If you own a file, you can decide who has the right to read it, write in it (make changes to it), or, if it is a program, to execute it. You can also restrict permissions for directories. When you grant execute permission for a directory, you allow the specified users to change directory to it and list its contents with the **`ls`** command (see **`ls(1)`**). Only the owner or a privileged user can define the following:

- which users have permission to access data
- which types of permission they have (that is, how they are allowed to use the data)

This section introduces access-permissions for files and discusses file protection.

File Permissions

The OS defines access-control and privilege mechanisms to allow for extended-security-controls that implement security policies different from those in PowerMAX OS, but which avoid altering or overriding the defined semantics of any functions in the OS. Although quite simple, the access-control scheme has some unusual features. Each user has a unique user-identification (user-id) number, as well as a shared group-identification (group-id) number. A file is tagged with the user-id and group-id of its owner, and a set of access-permission-bits when created by **`open`**, **`creat`**, **`mkdir`**, **`mknod`** and **`mkfifo`** (see **`open(2)`**, **`creat(2)`**, **`mkdir(2)`**, **`mknod(2)`** and **`mkfifo(2)`**). The OS file-access-control uses the access-permission-bits to specify independent read, write and execute permissions for the *owner* of the file, for any members of the owner's *group* and for any *other* users. For directories, execute permission means *search* permission. These access-permission-bits are changed by **`chmod`**, and are read by **`stat`** and **`fstat`**(see **`chmod(2)`**, **`stat(2)`** and **`fstat(2)`**).

When a process requests file-access-permission for read, write or execute/search, access is determined as follows:

1. If the effective-user-id of the process is a user with appropriate access-permissions (such as a privileged user).
    a. If read, write or directory search permission is requested, access is granted.
    b. If execute permission is requested, access is granted if execute permission is granted to at least one user by the file-permission-bits or by an alternate-access-control mechanism; otherwise, access is denied.
2. Otherwise:
    a. The read, write and execute/search access-permissions on a file are granted to a process if one or more of the following are true (see **`chmod(2)`**):
        - The appropriate access-permission-bit of the owner portion of the file-mode is set and the effective-user-id of the process matches the user-id of the owner of the file
        - The appropriate access-permission-bit of the group portion of the file-mode is set, the effective-group-id of the process matches the group-id of the file and the effective-user-id of the process fails to match the user-id of the owner of the file.

- The appropriate access-permission-bit of the other portion of the file-mode is set, the effective-group-id of the process fails to match the group-id of the file and the effective-user-id of the process fails to match the user-id of the owner of the file.

Otherwise, the corresponding access-permissions on a file are denied to the process.

b. Access is granted if an alternate-access-control mechanism is not enabled and the requested access-permission-bit is set for the class to which the process belongs, or if an alternate-access-control mechanism is enabled and it allows the requested access; otherwise, access is denied.

Implementations may provide additional-file-access-control or alternate-file-access-control mechanisms, or both. An additional-access-control mechanism only further restricts the file-access-permissions defined by the file-permission-bits. An alternate-access-control mechanism shall:

1. specify file-permission-bits for the file-owner-class, file-group-class and file-other-class of the file, corresponding to the access-permissions, that **`stat`** and **`fstat`** return.
2. Be enabled only by explicit user action, on a per-file basis by the file-owner or a user with the appropriate-privilege.
3. Be disabled for a file after the file-permission-bits are changed for that file with **`chmod`**. The disabling of the alternate mechanism need not disable any additional mechanisms defined by an implementation.

Setting Default Permissions

When a file is created its default permissions are set. These default settings may be changed by placing an appropriate **`umask`** command in the system profile (**`/etc/profile`**).

Figure 9-15. Umask(1) Settings for Different Security Levels

Level of Security	umask	Disallows
Permissive	0002	`w` for others
Moderate	0027	`w` for group, `rwx` for others
Severe	0077	`rwx` for group and others

How to Determine Existing Permissions

You can determine what permissions are currently in effect on a file or a directory by using **`ls -l`** to produce a long listing of a directory's contents.

In the first field of the **`ls -l`** output, the next nine characters are interpreted as three sets of three bits each. The first set refers to the owner's permissions; the next to permissions of members in the file's group; and the last to all others. Within each set, the three characters show permission to read, to write, and to execute the file as a program, respectively. For a directory, “execute” permission is interpreted to mean permission to search the directory for a specified file. For example, typing **`ls -l`** while in the directory named **`starship/bin`** in the sample file system produces the following output:

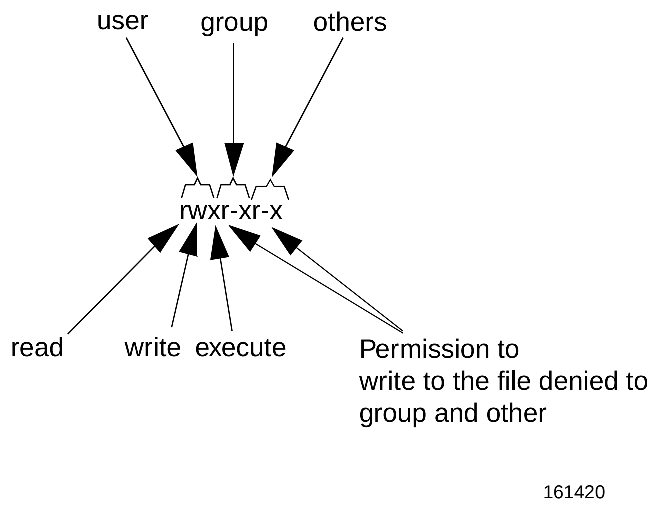
```
$ ls -l
total 35
-rwxr-xr-x   1 starship      project        9346  Nov 1  08:06  display
-rw-r--r--   1 starship      project        6428  Dec 2  10:24  list
drwx--x--x   2 starship      project          32  Nov 8  15:32  tools
$
```

Permissions for the **`display`** and **`list`** files and the **`tools`** directory are shown on the left of the screen under the line `total 35`, and appear in this format:

```
-rwxr-xr-x      (for the display file)
-rw-r--r--      (for the list file))
drwx--x--x      (for the tools directory)
```

After the initial character, which describes the file type (for example, a `-` (dash) symbolizes a regular file and a `d` a directory), the other nine characters that set the permissions comprise three sets of three characters. The first set refers to permissions for the owner, the second set to permissions for group members, and the last set to permissions for all other system users. Within each set of characters, the `r`, `w` and `x` show the permissions currently granted to each category. If a dash appears instead of an `r`, `w` or `x` permission to read, write or execute is denied.

The following diagram summarizes this breakdown for the file named **`display`**.



As you can see, the owner has `r`, `w`, and `x` permissions and members of the group and other system users have `r` and `x` permissions.

There are two exceptions to this notation system. Occasionally the letter `s` or the letter `l` may appear in the permissions line, instead of an `r`, `w` or `x`. The letter `s` (short for set user ID or set group ID) represents a special type of permission to execute a file. It appears where you normally see an `x` (or `-`) for the user or group (the first and second sets of permissions). From a user's point of view it is equivalent to an `x` in the same position; it implies that execute permission exists. It is significant only for programmers and system administrators. (See the *System Administration, Volume 1*, manual for details about setting the user or group ID.) The letter `l` indicates that locking will occur when the file is accessed. It does not mean that the file has been locked. The permissions are as follows:

Figure 9-16. File Access Permissions

Symbol	Explanation
`r`	The file is readable.
`w`	The file is writable.
`x`	The file is executable.
`-`	This permission is <u>not</u> granted.
`l`	Mandatory locking will occur during access. (The set-group-ID bit is on and the <u>group</u> execution bit is off.)
`s`	The `s`et-user-ID or `s`et-group-ID bit is on, and the corresponding <u>user</u> or <u>group</u> execution bit is also on.
`S`	The set-user-ID bit is on and the <u>user</u> execution bit is off.
`t`	The sticky and the execution bits for <u>other</u> are on.
`T`	The sticky bit is turned on, and the execution bit for <u>other</u> is off.

Figure 9-17. Directory Access Permissions

Symbol	Explanation
`r`	The directory is readable.
`w`	The directory may be altered (files may be added or removed).
`x`	The directory may be searched. (This permission is required to **`cd`** to the directory.)
`t`	File removal from a writable directory is limited to the owner of the directory or file unless the file is writable.

How to Change Existing Permissions

After you have determined what permissions are in effect, you can change them by calling the **`chmod`** command in the following format:

`chmod` *who+permission file(s)*

or

chmod *who=permission file(s)*

The following list defines each component of this command line.

chmod	name of the program

who	one of three user groups (u, g or o)
	u = user
	g = group
	o = others

+ or -	instruction that grants (+) or denies (-) permission

permission	any combination of three authorizations (r, w and x)
	r = read
	w = write
	x = execute

file(s)	file (or directory) name(s) listed; assumed to be branches from your current directory, unless you use full pathnames.

NOTE

The **chmod** command will not work if you type a space(s) between *who*, the instruction that gives (+) or denies (-) permission, and the *permission.*

The following examples show a few possible ways to use the **chmod** command. As the owner of **display**, you can read, write, and run this executable file. You can protect the file against being accidentally changed by denying yourself write (w) permission. To do this, type the command line:

chmod u-w display

After receiving the prompt, type **ls -l** and press the RETURN key to verify that this permission has been changed, as shown in the following screen.

```
$ chmod u-w display
$ ls -l
total 35
-r-xr-xr-x   1 starship      project        9346  Nov 1  08:06  display
rw-r--r--    1 starship      project        6428  Dec 2  10:24  list
drwx--x--x   2 starship      project          32  Nov 8  15:32  tools
$
```

As you can see, you no longer have permission to write changes into the file. You will not be able to change this file until you restore write permission for yourself.

Now consider another example. Notice that permission to write into the file **`display`** has been denied to members of your group and other system users. However, they do have read permission. This means they can copy the file into their own directories and then make changes to it. To prevent all system users from copying this file, you can deny them read permission by typing:

```
chmod go-r display
```

The `g` and `o` stand for group members and all other system users, respectively, and the **`-r`** denies them permission to read or copy the file. Check the results with the **`ls -l`** command.

```
$ chmod go-r display
$ ls -l
total 35
-rwx--x--x   1 starship      project        9346  Nov 1  08:06  display
rw-r--r--    1 starship      project        6428  Dec 2  10:24  list
drwx--x--x   2 starship      project          32  Nov 8  15:32  tools
$
```

For more information, refer to **`ls(1)`** and **`chmod(1)`** in the *Command Reference.*

A Note on Permissions and Directories

You can use the **`chmod`** command to grant or deny permission for directories as well as files. Simply specify a directory name instead of a file name on the command line.

However, consider the impact on various system users of changing permissions for directories. For example, suppose you grant read permission for a directory to yourself (`u`), members of your group (`g`), and other system users (`o`). Every user who has access to the system will be able to read the names of the files contained in that directory by running the **`ls -l`** command. Similarly, granting write permission allows the designated users to create new files in the directory and remove existing ones. Granting permission to execute the directory allows designated users to move to that directory (and make it their current directory) by using the **`cd`** command.

An Alternative Method

There are two methods by which the **`chmod`** command can be executed. The method described above, in which symbols such as `r`, `w` and `x` are used to specify permissions, is called the symbolic method.

An alternative method is the octal method. Its format requires you to specify permissions using three octal numbers, ranging from 0 to 7. (The octal number system is different from the decimal system that we typically use on a day-to-day basis.) To learn how to use the octal method, see the **`chmod(1)`** entry in the *Command Reference.*

Security Considerations

This section gives the software developer information on various security features and their impact on writing applications. While many of the security features, like Mandatory Access Control, are available only if the Enhanced Security Utilities are installed and running, it is to your advantage to program your application so that it will run on the OS with and without the Enhanced Security Utilities installed. This way, you can avoid programming the same application for each environment.

What Security Means to Programmers

As a programmer on PowerMAX OS, you need a general understanding of how security affects you and protects your files on the computer system. You also need to understand the difference between basic security and enhanced security. Finally, you need to understand the term Trusted Computing Base (TCB), an all-encompassing term which describes the mechanisms used to enforce Enhanced Security.

What Is Security?

Security for a computing system means that the information on the system is protected from unauthorized disclosure or modification. If each user had a personal non-networked computing system that was kept locked up, each user's files would be secure. But isolation and physical security are not practical in most circumstances.

On a computer system that many people share, the simplest security mechanism would be to allow only the owner of a file to access that file. That would be inconvenient, however, since one of the benefits of a computer system is the sharing of resources. For example, it would be wasteful for each user to have a private copy of each command. Commands are usually shared, but users often want to restrict access to the contents of data files.

On a secure system, each user has a unique identity and a level of authorization associated with that identity. For security to work, the computer system must have some way of identifying users, their level of authorization, and their files. For the most part, while you are logged in, all data you enter, create, and process belongs to you. Data is stored in named files on the computer system. Each file you own is kept separate from the rest of your files and from the files belonging to other users.

As a programmer, you are also concerned with the impact of security on users who run your programs.

A secure computer system must have a mechanism that makes access decisions, that is, one that decides who can access what, based upon user identity and authorization.

There are many ways in which the security of a computer system can be violated. Unauthorized access to read or write files can be the result of:

- The abuse of privileges by users or administrators
- Malicious programs that surreptitiously gain privileges or access to files

- Idle browsing of files that are inadequately protected

Most computer systems provide some degree of basic security. However, the mechanisms supplied by the TCB and the Enhanced Security Utilities available with the OS provide specific, enhanced protection against these and other potential security hazards.

A review of basic security will provide a background for understanding the Enhanced Security Utilities available with the OS.

How Basic Security Works

An operating system stores and processes information in the form of electronic data. In doing so, it provides an interface between you, the user of the computer, and the computer. An operating system provides you with commands, library routines, functions, and programs that allow you to tell the computer how to store and process the information that belongs to you.

A computer system enforces basic security by making access decisions, that is, by deciding who can access what. In order to make access decisions, a computer system uniquely identifies each user on the system and stores information in named files, each of which belongs to a single user on the system. It would be a potential violation of security if users could access any files at will.

The OS supplies basic security through the use of the **`login`** and **`passwd`** (password) mechanisms, which identify you to the system and put you in control of your data. Also included in basic security are `access mode bits,` which give users some control over which other users can access their files. It is not a violation of basic security for users to have the ability to share individual files with specific other users.

Additionally, you may attach specific privileges to executable files and processes. This allows programs to perform privileged operations without having to become root.

Privileged users need to perform sensitive tasks, but because privileges are associated with processes and executable files, not user IDs (except for the special case of UID=0 when using the SUM privilege policy module), it is not possible to grant privileges to users directly. The Trusted Facility Management (TFM) tools provide an interface between users and privileges. The TFM tools maintain a database of users and the commands they may execute with privilege. The **`tfadmin(1)`** command invokes the requested command, regulating the privileges based on the TFM database information. If privileges are assigned to a user's shell, they will be inherited by all commands executed by that shell.

How Enhanced Security Works

The mechanisms that provide enhanced security appear to be simply more restrictive versions of the mechanisms that provide basic security. They are that, but they are much more. The mechanisms available with the OS are designed to protect sensitive information. Sensitive information must be specially protected according to the rules of a security policy because its unauthorized disclosure, loss, or alteration will cause damage or harm to someone or something. The Enhanced Security mechanisms that protect sensitive information are part of the Trusted Computing Base, or TCB.

Because the information on a computer system can be easily shared and potentially stolen, the TCB can be thought of as the mechanisms that control the sharing of information. That is, the TCB controls who has access to what. By controlling access, the TCB can protect your files and your programs from being seen or accessed by other computer users.

The security policy for the OS prescribes a relationship between access rules and access attributes. The access rules allow the TCB to define several distinct levels of authorization, and the access attributes provide the mechanism for the TCB to prevent unauthorized access to sensitive information.

NOTE

Note that the levels of authorization defined by the TCB are similar to but much more complicated and restrictive than the access control provided by the access mode bits.

More specifically, the security policy for a computing system running on the OS describes the relationships among five elements. The first two elements are the subjects and objects on a computer system that interact with each other.

- Subjects cause information to flow among objects or they change the system status. A user is represented on the system as a subject.
- Objects are those parts of a computing system that contain or receive information. Examples of objects are data files, program files, memory, terminals, line printers, disks, tapes, and processes.

NOTE

Note that a process is a subject when it requests an action and an object when it receives information.

Typical interactions are for subjects to create, read, or write objects.

The remaining three elements define the ways in which subjects and objects interact. These elements are access attributes, access rules, and process privileges.

- The access attributes of a subject or an object define its position within the separation scheme that the TCB uses to segregate computer users and information on the computer system.
- The access rules embody the policy that segregates information for the system. The TCB determines whether a subject can access a given object by comparing the access attributes of the subject with the access attributes that are required to access the object. Only if a subject passes all relevant access checks can it access an object.
- Privileges are assigned to executables. When a subject has access to that executable, that subject can perform override access checks to perform sensitive system operations.

NOTE

The security policy requires that a process has only the privileges it needs to perform its task and that it relinquish a privilege when it is no longer needed. Most users execute processes without privileges. If your programs need privileges, contact a system administrator.

As a very simplified approximation, you can think of the security policy as segregating the levels of authorization in a hierarchy, that is, some levels are conceptually “higher” than other levels. A subject can read an object if and only if the subject's level is higher than the level of the object. A subject can write to an object if and only if the subject's level is equal to the level of the object.

Actually, the relationship between security levels is much more complex than this simple approximation. Security levels can be disjoint. For more detailed explanations of security levels, refer to *System Administration, Volume 1*.

In enforcing the security policy, the TCB assigns access permissions to subjects and objects according to the local security policy as instituted by the system administrators, and then uses the access rules to ensure that subjects do not access objects for which the subjects do not have the proper access attributes.

The TCB further restricts the use of certain commands and system calls to subjects (processes) that have the proper privileges.

Thus, users are limited in their ability to allow access, and the TCB makes access decisions. Security is enhanced because the ability to grant access is enforced by the TCB, not by individual users.

Since the TCB restricts the use of certain commands and system calls to subjects (processes) which have the proper privileges, you will likely want to read more detail about the security features that affect applications programming in the next section.

Privileges

Privilege, in the simplest terms, is the ability to override system restrictions on the actions of users. All operating systems allow users to exercise special privilege, under certain conditions, to perform sensitive system operations. Sensitive system operations are those which affect the configuration of the system or its availability to users.

Most users cannot, for example, execute commands affecting the hardware or software configuration of the system. Activities such as mounting and checking file systems, adding users, modifying user profiles, adding and removing peripherals, installing application software, password administration, and administration of the user terminal lines, are restricted to certain users.

In previous UNIX releases, the restriction of privilege is implemented by designating a special user identifier (UID) of `0`; the login name historically associated with this UID is `root`.

When a person logs in as `root`, that person has unrestricted access to every file on the system, and the ability to alter system operation. Commands that execute sensitive system operations check to see whether the effective UID of the process requesting the operation is `0`. If it is, the user process is given unlimited access to the system.

The `root` login in previous UNIX releases possesses, in effect, the one privilege necessary to override all system restrictions on command execution and access: the superuser privilege.

The OS provides an alternative privilege mechanism that is more flexible to suit the needs of the user community. Now, rather than investing the power to issue any command on the system to one user, you can give partial superuser power to several users. By assigning privileges linked to specific tasks, you essentially assign a role to each such user.

This privilege mechanism checks the invoking process for the presence of one or more of a discrete set of privileges corresponding to each sensitive system operation.

The privileges inherited by a new process are derived from the calling process's privileges and the privileges set on the file being executed. This type of privilege mechanism is called a file-based privilege mechanism. It is actually a combination of the old UID functionality supported in the UNIX operating system for over 20 years, and new, discrete privilege functionality.

The most important advantage of this privilege mechanism over the pure UID-based privilege mechanism is the fine granularity with which it can apportion system privileges to executing processes. For example, you might assign someone to the role of mail administrator. That person would have all the privileges necessary to oversee maintenance and troubleshooting of the mail subsystem, but no others; he or she wouldn't be able to add and delete user accounts, reorganize file systems, or do any other administrative work unrelated to electronic mail.

The superuser privilege is replaced by a list of discrete privileges based on the categorization of sensitive system operations into groups of operations exercising the same kind of privilege. In other words, many different commands might need to override discretionary read access restrictions on files to perform their functions; defining a privilege such as `P_DACREAD`.

While the privilege mechanism provides the means by which a system can apportion and control process privileges, the privilege policy module provides the rules by which the system grants privileges to processes.

The system is delivered with a Super User Module (SUM) privilege policy module that provides the same functionality as provided by the superuser privilege in previous UNIX releases. It also provides additional flexibility by allowing fixed privileges on executable files.

The Enhanced Security Utilities are delivered with a Least Privilege Module (LPM), that provides a more restrictive privilege policy. This policy specifies that processes execute with only the amount of privilege necessary to perform their given function, and no more. The superuser is not automatically granted privileges as it is when using the SUM privilege policy module.

By default, the SUM module is used unless the Enhanced Security Utilities are installed and you specifically include the LPM module.

Both privilege policy modules use:

- a list of system privileges
- a working and maximum set of privileges for each process
- a fixed set of privileges for each executable file

The LPM policy module also supports an inheritable set of privileges for executable files in addition to the fixed set.

It is important to recognize that the list of system privileges, fixed privileges on files, and the inheritance mechanism are all part of the basic privilege mechanism provided by the operating system and are present even when the Enhanced Security Utilities are not installed.

The ability to override system restrictions with privileges is vested in three ways:

- to any user whose effective identity is root (SUM policy module only)
- through fixed privileges assigned to executable files
- by way of the Trusted Facility Management (TFM) utilities

When using the SUM privilege policy module, any process with an effective user-ID of "0" (root) is considered omnipotent and has all privileges assigned to it.

The second and third approaches provide methods of giving a "little bit of root" to a user or command. They introduce the idea of discrete privileges that are associated with executable files and processes. A process has privilege only when it is executing a privileged command.

The **`filepriv(1M)`** command assigns fixed privileges to executable files. The fixed privileges become a file attribute. The executable file will always run with privilege for those users who can access the file. The fixed privileges for a file will become invalid if any attributes of the file are changed (checksum, file size, ctime, mode, etc). You may want to assign fixed privileges to commands when you are more concerned that users are accessing a version of a command that you can verify is secure, than with the identities of the users using the command.

Privileged users need to perform sensitive tasks, but because privileges are associated with processes and executable files, not user IDs (except for "root" when using the SUM privilege policy module), it is not possible to grant privileges to users directly. The Trusted Facility Management (TFM) utilities provide a way to associate users with the commands that they can invoke with privilege. Commands that can have a wide-ranging or destructive effect on the system can be restricted to one user. Unlike file privileges, the TFM database privileges for an executable file are not invalidated if the file is modified.

The system administrator retains the most control with the TFM tools by specifying exactly which commands can be executed with privilege by each user. However, this can become a burden for the administrator to have to add new commands to the TFM database every time users need to execute additional commands with privilege. Privileges can be assigned to the user's shell instead of assigning privileges to individual commands. Because privileges are inherited by child processes, any privileges given to the user's shell will be inherited by every command the user executes under that shell. See the "Trusted Facility Management" chapter in the *System Administrators Volume 1* manual for additional information on the TFM utilities.

Privileges Associated with a File

For every executable file there may be a set of privileges that are acquired when that program is executed via an **`exec`** system call. This set of privileges is known as fixed privileges: they are always given to the new program, independent of the privileges of the parent or calling-process. Each executable file can have two sets of privileges associated with it that are propagated when that program is executed via an **`exec`** system call:

- Fixed privileges are always given to the new program, independent of the calling or parent process's privileges.
- Inheritable privileges will exist in the new program only if they existed in the previous program. Inheritable privileges are given to the new program only if they exist in the calling process's privilege set. Inheritable privileges are only used by the LPM privilege module, not by the SUM privilege module. The SUM module considers all of the privileges in the calling process’s maximum privilege set to be inheritable

These sets are disjoint, that is, a privilege cannot be defined as both fixed and inheritable for the same file. If an executable file does not require any privileges, both sets are empty.

CAUTION

Privileges associated with a file are removed when the validity information for the file changes (for example, when the file is opened for writing or when the modes of the file change). This removes the file from the Trusted Computing Base; the privileges must be set again in order for the command to run with privilege. This validity checking can be disabled. See the following section and the **`initprivs(1M)`** manual page for further details.

Due to the default values of VAL_CKSUM, VAL_SIZE, and VAL_VALIDITY in the file **`/etc/default/privcmds`**, if a file’s attributes are modified through any of the above stated system calls, then the kernel privilege tables will retain the privilege information that has been established on that file. On most SUM systems this may not be a problem; however, if one wishes to use the privilege mechanisms provided, then a problem may occur where the vnode of a file which has been removed and is subsequently re-acquired as a new file name already has the previous file’s privilege attributes established. To resolve this problem the system administrator should set the values of VAL_CKSUM, VAL_SIZE, and VAL_VALIDITY to Yes in **`/etc/default/privcmds`**. This assures that when a vnode is acquired validity checking will be performed and the file will be created with no privilege attributes established.

Giving privileges to non-evaluated programs violates the B2 security rating (with the Enhanced Security Utilities installed.) Great care must be taken when writing software that requires privileges. See the appendix "Guidelines for Writing Trusted Software" in this guide.

Manipulating File Privileges

The kernel maintains a table of file privileges in memory. It is initialized at system startup by the **initprivs(1M)** command using the file privilege entries in the Privilege Data File (PDF), **/etc/security/tcb/privs**.

NOTE

If the PDF is missing, an error results and the system is halted. The system should then be rebooted from tape.

Entries in the PDF are added, deleted, or modified using the **filepriv(1M)** command. When the **filepriv** command adds a file to the PDF, it records checksum, size and last updated time information about the file in addition to the file privileges. **initprivs** compares this validity information to the current values for the file. By default, if these do not match, the file will not be granted privileges by initprivs and the entry will not be passed to the kernel to add to the kernel privilege table. This validity checking can be disabled by resetting flags contained in the file **/etc/default/privcmds**. On systems where the SUM module is configured (the Enhanced Security Utilities are not installed), disabling validity checking in **initprivs** also disables the validity checking performed by the kernel when a file is executed with **exec(2)**. See **initprivs(1M)** for further information.

File privilege entries in the kernel privilege table can be set, retrieved or counted using the **filepriv(2)** system call. The **filepriv(2)** system call does not modify the PDF entry for the file. Privileges that are changed with **filepriv(2)** are valid only until the next reboot, at which time the changes are lost and the privileges are as defined in the PDF.

The **filepriv(1M)** command updates the kernel privilege table (using the **filepriv(2)** system call) each time additions, deletions or changes are made to entries in the PDF.

The **filepriv(2)** system call has three command types:

- PUTPRV sets the fixed and inheritable privileges associated with a file. This is an absolute setting; the specified privileges replace any previously existing privileges for the file.
- GETPRV retrieves the fixed and inheritable privileges associated with a file.
- CNTPRV returns the number of privileges associated with a file.

intro(2) lists the names and descriptions of each privilege. **privilege(5)** lists the name of the privileges include file as well as some other important items, including macros to manipulate privilege descriptors. **priv(5)** lists some functions that can be used to prepare the arguments to the privilege system calls.

Some of the above command types require a list of privileges or return such a list. `PUTPRV` requires an array of privilege descriptors that lists the privileges to be set. A privilege descriptor is an integral data type that is assigned a value defining the privilege and the set it is in. Functions such as **`pm_inher`** and **`pm_fixed`** have been defined to make this task simpler. Use `pm_inher` to indicate an inheritable privilege. For example, `pm_fixed(P_DACREAD)` would indicate the `P_DACREAD` privilege in the fixed set. Similarly `pm_inher(P_MACREAD)` would indicate the `P_MACREAD` privilege in the inheritable set.

Screen 9-1 shows a code fragment that sets file privileges in the kernel privilege table. The inheritable privilege set indicated in this example may or may not exist or be valid for your particular system.

```
#include <priv.h>

priv_t privd[3];
/*
 * Set P_DACREAD and P_DACWRITE as inheritable and
 * P_SETUID as fixed for file /sbin/testprog.
 * This process must have P_SETFPRIV, P_DACREAD, P_DACWRITE, and
 * P_SETUID in its maximum set.
 */
privd[0] = pm_inher(P_DACREAD);
privd[1] = pm_inher(P_DACWRITE);
privd[2] = pm_fixed(P_SETUID);
if (filepriv(“/sbin/testprog”, PUTPRV, privd, 3) == -1) {
    /* Some error occurred, display the error and exit. */
    perror(“filepriv PUTPRV error”);
    exit(1);
}
```

Screen 9-1. Setting File Privileges in Kernel Privilege Table

In this example, privileges are being set for the executable file **`/sbin/testprog`**. The privileges `P_DACREAD` and `P_DACWRITE` are made inheritable, while `P_SETUID` is made fixed. `pm_inher` and `pm_fixed` are used to assign values to the privilege descriptors; the `pm_inher` function marks `P_DACREAD` and `P_DACWRITE` as inheritable while `pm_fixed` marks `P_SETUID` as fixed. The call to `filepriv` using `PUTPRV` will set the indicated privileges for the file in the kernel privilege table and are valid only until the next reboot. If an error occurred, `perror` is called to display an error message (see **`perror(3C)`**) and the program terminates.

The **`filepriv(1M)`** command must be used to make these changes permanent in the Privilege Data File (PDF) that is used to initialize the kernel privilege table on the next reboot.

NOTE

A privilege that is being set for a file must exist in the maximum set of the process making the **`filepriv`** system call.

Since the **`PUTPRV`** command for **`filepriv`** is a privileged operation, a process using this system call must have the appropriate privilege in its working set. See **`intro(2)`** for a list of privileges.

Use the **`GETPRV`** command for the **`filepriv`** system call to determine the fixed and inheritable privileges associated with a file. This command also requires a pointer to an array of privilege descriptors. You must ensure that the array is large enough to contain all the privileges associated with the file.

Screen 9-2 shows a code fragment that will retrieve the privileges associated with a file.

```
#include <priv.h>

priv_t *privp;
int cnt;
/*
 * Determine the number of privileges for /sbin/testprog.
 */
if ((cnt = filepriv(“/sbin/testprog”, CNTPRV, (priv_t *)0, 0)) == -1) {
     /* filepriv failed; display error and exit. */
     perror(“filepriv CNTPRV error”);
     exit(1);
}
if (cnt > 0) {
     /*
      * malloc some memory and get the privileges.
      */
     if ((privp = (priv_t *)malloc(cnt * sizeof(priv_t))) == NULL) {
          exit(1);   /* Couldn't malloc so exit. */
     }
     if (filepriv(“/sbin/testprog”, GETPRV, privp, cnt) == -1) {
          /* filepriv failed; display error and exit. */
          perror(“filepriv GETPRV error”);
          exit(1);
     }
}
```

Screen 9-2. Retrieving File Privileges

In this example, the **`CNTPRV`** command is used to determine the number of privileges. This number is then used to determine the amount of memory to request when calling **`malloc`** for an array large enough to contain all the privileges. (see **`malloc(3C)`**). **`filepriv`** is then called with the **`GETPRV`** command to retrieve the actual privileges.

Privileges Associated with a Process

After a **`fork`**, the privileges of the parent and child processes are identical. However, when an **`exec`** system call is performed, the privileges of the new program are determined from those of the program performing the **`exec`** and from the privileges associated with the executable file.

Each process has two sets of privileges:

- The maximum set contains all the privileges granted to the process, either as fixed or inherited privileges.
- The working set contains all the privileges currently being used by the process.

How the privileges for a new process are determined is specific to the privilege policy module installed. The “Privilege Policy Modules” section in the “Administering Privilege” chapter of the *System Administration Volume 1* manual provides further information on the privilege policy modules.

Manipulating Process Privileges

Use the **`procpriv`** system call to add, put, remove, retrieve, or count privileges associated with the calling process. This system call has five command types:

- `SETPRV` adds the requested privileges to the working set for the current process. Privileges already in the working set are not affected; they remain in the set. Requested privileges not in the current maximum set are ignored.
- `PUTPRV` sets the working and maximum sets for the current process. This is an absolute setting; the specified privileges replace the current working and maximum sets. Privileges requested which are not in the current maximum set are ignored.
- `CLRPRV` removes the requested privileges from either the working or maximum set. If a privilege is removed from the maximum set, it is also removed from the working set if it exists there, since the working set is always a subset of the maximum set.
- `GETPRV` retrieves the working and maximum privilege sets for the current process.
- `CNTPRV` returns the number of privileges associated with the current process.

Screen 9-3 shows a code fragment that does a **`setuid`** and uses **`procpriv`** to set and clear the appropriate privilege as needed.

The first call to **`procpriv`** sets the `P_SETUID` privilege in the process's working set. Note that the count of 1 in the system call indicates that only one (the first) element of the array `privd` is to be used. Once the privilege is in the working set, **`setuid`** is called. Since `P_SETUID` will not be required by the program any more, **`procpriv`** is again called, this time with the **`CLRPRV`** command.

Note in this case that the count of 2 indicates that both elements of array `privd` are to be used, thus removing the privilege from both the maximum and working sets. Note that if the privilege had only been removed from the maximum set, the system would have also removed it from the working set, since the working set must be a subset of the maximum set, that is, the working set cannot contain privileges which are not in the maximum set.

Use the **PUTPRV** command for **procpriv** similarly to **SETPRV,** but remember that the setting is absolute, that is, the indicated privileges replace both the current working and maximum sets. The privileges you request must exist in the current maximum set.

```
#include <priv.h>

priv_t privd[2];
int uid;

privd[0] = pm_work(P_SETUID);
privd[1] = pm_max(P_SETUID);
/*
 * Add P_SETUID to the working set of the current process.  P_SETUID
 * must be in the maximum working set to be successful.
 */
if (procpriv(SETPRV, privd, 1) == -1) {
    /* It failed, so display error and exit. */
    perror(“procpriv SETPRV error”);
    exit(1);
}
/*
 * Change to user id “uid” (previously initialized)
 */
if (setuid(uid) == -1) {
    /*
     * It failed, perhaps P_SETUID wasn't in our maximum working
     * set.  Display error and exit.
     */
    perror(“setuid error”);
    exit(1);
}
/*
 * We don't need P_SETUID any more so remove it from the working
 * and maximum sets.
 */
if (procpriv(CLRPRV, privd, 2) == -1) {
    /*
     * It failed, so display error and exit.
     */
    perror(“procpriv CLRPRV error”);
    exit(1);
}
```

Screen 9-3. Adding and Clearing Process Privileges

Screen 9-4 shows a code fragment that uses the **PUTPRV** command to set the maximum and working sets.

```
#include <priv.h>

priv_t privd[2];

privd[0] = pm_max(P_SETUID);
/*
 * Set the maximum set to P_SETUID.  The working set is empty since
 * it is not set here.
 */
if (procpriv(PUTPRV, privd, 1) == -1) {
    /* It failed, so display error and exit. */
    perror(“procpriv PUTPRV error”);
    exit(1);
}
```

Screen 9-4. Setting Process Privileges Using PUTPRV

In this example, the privilege descriptor is set to `P_SETUID` in the maximum set. If `P_SETUID` is already in the maximum set, **`procpriv`** causes the new maximum set to contain only `P_SETUID.` The new working set will be empty, since no privileges are defined for it.

The **`GETPRV`** and **`CNTPRV`** commands work in a manner similar to their counterparts in the **`filepriv`** system call. Screen 9-5 shows a code fragment that will retrieve the privileges associated with a process.

```
#include <priv.h>

priv_t *privp;
int cnt;

/*
 * Determine the number of privileges for this process.
 */
if ((cnt = procpriv(CNTPRV, (priv_t *)0, 0)) == -1) {
    /* procpriv failed; display error and exit. */
    perror(“procpriv CNTPRV error”);
    exit(1);
}
if (cnt > 0) {
    /*
     * malloc some memory and get the privileges.
     */
    if ((privp = (priv_t *)malloc(cnt * sizeof(priv_t)) == NULL) {
        /* Couldn't malloc so exit. */
        exit(1);
    }
    if (procpriv(GETPRV, privp, cnt) == -1) {
        /* procpriv failed; display error and exit. */
        perror(“procpriv GETPRV error”);
        exit(1);
    }
}
```

Screen 9-5. Retrieving Process Privileges

In this example, the number of privileges returned by the **`CNTPRV`** command to **`procpriv`** is used to determine the amount of memory to request when calling **`malloc.`** **`procpriv`** is then called with the **`GETPRV`** command to retrieve the actual privileges.

With proper use, the privilege mechanism provides a way to restrict execution of sensitive system functions and improves the security of the system. See “Guidelines for Writing Trusted Software” in this guide.

Device Security

On a system with multilevel security, all devices that store data need to be protected by a range of security levels, which restrict what data can be stored on the device. These restrictions ensure that a device receives only data that is appropriate for its location and configuration. For example, you would not want to print highly sensitive information on a printer located in a public area, accessible to everyone on the site.

Information on security characteristics of devices is stored in the Device Database (DDB), which is a collection of three files. The file **`/etc/device.tab`** contains one entry per device alias that defines the following:

- all device attributes not related to security
- the **`secdev`** attribute

The file **`/etc/security/ddb/ddb_sec`** contains definitions for all security-related attributes of devices.

The file **`/etc/security/ddb/ddb_dsfmap`** contains one entry for each block or character device special file, indicating to which device alias it maps. The device special files are checked to see that they are unique and that each one maps to only one device alias.

A set of commands is provided to administer information in the DDB. The **`putdev`** command is used to add information to the DDB or to modify existing information. The **`getdev`** command can be used to list the devices defined in the DDB. The **`devattr`** command can be used to list the values of device attributes, including the security attributes. These are administrative commands that can be used only by an appropriate administrator.

The security attributes in the DDB define the characteristics the device will have when it is allocated for use by the operating system's kernel. Upon allocating the device, the kernel checks the information in the DDB and uses it, along with any information passed to the device allocation routine, to set security characteristics for the device. Once the device is allocated, the security characteristics are maintained in kernel data structures. The **`devalloc`** routine can be used to either get or set the security attributes of a device. When used to set security attributes, **`devalloc`** invokes the **`devstat`** system call to set the device attributes. The **`devdealloc`** routine is used to deallocate a device whose attributes were set by **`devalloc.`**

To write application programs that control devices, you will need to understand the DDB and the routines and system calls used for secure device allocation. The rest of this section presents information on those topics. Information on secure device management can also be found in *System Administration, Volume 1*. You should also read carefully the “Guidelines for Writing Trusted Software” chapter of this guide before using the features described below.

Any programs that use the features described in this section will need the appropriate privilege to execute properly. See *System Administration, Volume 1*, for information on the privileges required.

Device Database

Before any device can be allocated for use by the kernel, it must be defined in the Device Database. Information on the Device Database and secure device attributes can be found in *System Administration, Volume 1*. You should read this section before attempting to write applications that allocate devices.

For secure devices, three essential attributes (`range`, `state`, and `mode`) must be defined for each device listed in the DDB. You can use the **`putdev`** command to define these and other attributes for a device; see **`putdev(1M)`** in the *Command Reference* for complete information.

Kernel Device Allocation

The kernel maintains several security-related attributes that are associated with each device special file. These attributes provide a mechanism to regulate access to devices.

The following security attributes are associated with each device special file.

release flag	The release flag indicates whether the device is allocated and the way in which it is allocated. The flag can have one of three values:

`DEV_PERSISTENT`	This value indicates that the security attributes are set explicitly and remain associated with the device special file while the system is running or until the attributes are explicitly changed.
`DEV_LASTCLOSE`	This value indicates that the security attributes are set explicitly and remain associated with the device until the last reference to the device is closed.
`DEV_SYSTEM`	This value indicates that the security attributes are set by the system. This value is used with the device driver flags (described in the next section) to handle several special cases. If (1) the release flag is `DEV_SYSTEM,` (2) the device state (defined below) is `private`, and (3) there are no open connections to the device (through open file descriptors or the **`mmap`** system call), then the device is not currently allocated.

state	The state attribute must either be `private` or `public`. A device state of `private` indicates that the device is a private TCB resource and that unprivileged access to the device is denied. A device state of `public` indicates that unprivileged access to the device is allowed. The state is changed from `private` to `public` when the device is allocated for unprivileged access and is changed from `public` to `private` when the device is deallocated.
level range	The device level range constrains the allowed values for the security level of the device and should be based on the physical constraints of the device (such as device location). The high level of the device level range must dominate the low level of the device level range. The device level (as set in the device special files for the device) must be contained in the level range.

mode	The device mode is always `static`. The other possible value for the device mode, `dynamic`, is provided only for those sites that are upgrading from a previous release of a secure UNIX system. If you are upgrading, please consult the documentation provided as part of that upgrade for the correct use of the device mode. If the device mode is `static`, then changing the MAC level of the device is prohibited if the device state is static and there are active I/O connections to the device. For all other cases, MAC level change is allowed. When the device mode is `dynamic`, the MAC access to the device is checked for each I/O operation. Thus, if the level of the device is changed, the process that is accessing the device must continue to pass MAC access checks.

Device Driver Flags

The kernel uses three new flags associated with each device driver to determine MAC access. The flags are used to handle special cases. If no flag is specified for the driver, the normal MAC checks (write equal and read down) are performed. The device driver flags are:

`NOSPECMACDATA`	This flag indicates that no MAC access checks will be done by the kernel for data transfers and no inode update access time changes.
`INITPUB`	This flag indicates that when the device has the release flag set to `DEV_SYSTEM`, the device is in `public` state. A device that has the `INITPUB` flag set in the driver will by default be accessible by non-privileged processes.
`RDWREQ`	This flag indicates that all accesses (both read and write) require strict equality.

Device Allocation Routines

In application programs, devices can be allocated with the **`devalloc`** routine and deallocated (set back to system configuration values) with the **`devdealloc`** routine. The **`devalloc`** routine can be used to set the security attributes defined above or to retrieve information about them. The use of these two routines is described briefly in the following subsections. Both these routines call the **`devstat`** system call to perform their functions. For more detailed information, see the **`devalloc(3X)`**, **`devdealloc(3X)`**, and **`devstat(2)`** manual pages in the *Operating System API Reference.*

The devalloc Routine

The **`devalloc`** routine can be used to get or set the security attributes of a device. The routine takes three arguments: a *device*, which can be either a pathname of a block or character special device or a device alias defined in the Device Database; a *cmd*, which is either `DEV_SET` or `DEV_GET`; and *bufp*, which is a pointer to a `dev_alloca` structure. This structure contains the following security attributes:

`dev_range`	The security level range that will be assigned to the device. It is expressed as a pair of levels, with the high level given first.
`dev_state`	The device state, which is either `DEV_PRIVATE` or `DEV_PUBLIC`.
`dev_mode`	The device mode, which is either `DEV_DYNAMIC` or `DEV_STATIC`.
`dev_relflag`	The release flag, which indicates how the security attributes can be released. The flag can be either `DEV_PERSISTENT`, `DEV_LASTCLOSE`, or `DEV_SYSTEM`.
`uid`	The user ID, which is used to check authorization to access the device. The type of the argument if `uid_t`.

The **`devalloc`** routine calls the `devstat` system call to get or set these values. If **`devstat`** fails, the **`devalloc`** routine undoes any work it has done.

The devdealloc Routine

The **`devdealloc`** routine is used to deallocate a device; it clears any security attributes set with **`devalloc`** and returns the device to the “system configuration” state. The values of this state are

range	*hilevel*=*lolevel*=`SYS_PUBLIC` (LID value 0)
state	`private`
mode	`static`
release flag	`DEV_SYSTEM`

The routine takes a *device* as an argument, which can be either the absolute pathname of a block or character special device or a device alias defined in the Device Database. **`devdealloc`** resets the values by invoking the **`devstat`** system call with a release flag set to `DEV_SYSTEM`. If **`devstat`** fails, **`devdealloc`** undoes any work it has done.

10
Signals, Job Control, and Pipes

10
Signals, Job Control, and Pipes

Introduction

The kernel provides several means by which processes can communicate with each other. This chapter provides a detailed discussion on three of these facilities: signals, pipes, and job control.

Signals are a communications mechanism between processes and the kernel. They notify a process that a certain event has occurred, and they can be sent to a process or a group of processes. Based on the type of signal received, a process may take some necessary action. Included in this chapter is a discussion of the types of signals, signal handlers, the way in which signals are sent, and the signal stack feature.

Job control provides a means of managing processes during a login session. The discussion here includes an overview of job control and STREAMS-based job control.

Also included in this chapter are sections devoted to pipes and STREAMS-based pipes and FIFOs. A pipe is a mechanism that provides a means of passing information from one running process to another. With the OS, pipes and FIFOs have become STREAMS-based for network applications. For completeness, a discussion of this subject has also been included.

Signals

A *signal* is an asynchronous notification of an event; it is said to be *generated* for (or *sent* to) a process when the event that causes the signal first occurs. A signal may be sent to a process by another process, from the terminal, or by the system itself. A signal can be generated in several ways, which include the following:

- An error during a system call
- An error caused by an LWP (a reference to memory that does not exist, for example)
- Some condition raised at the controlling-terminal of a process (such as break or hangup)
- An explicit system call to **`kill(2)`** or **`sigsend(2)`** or a call to one of the following POSIX interfaces: **`sigqueue(2)`**, **`mq_notify(3C)`**, **`aio_read(3C)`**, **`aio_write(3C)`**, **`lio_listio(3C)`**, **`aio_fsync(3C)`**, and **`timer_create(3C)`**.

- Expiration of the alarm clock timer or the generation of the trap signal during process tracing

Signals are the most frequently used means to notify a process of the occurrence of some event that may have an effect on that process. In some circumstances, the same event generates signals for multiple processes. A process may request a detailed notification of the source of the signal and the reason that it was generated (see **`sigaction(2)`**). All signals have the same priority. When multiple unblocked signals with different signal numbers are pending delivery to a process, they are delivered in order according to signal number; the unblocked pending signal with the lowest signal number is delivered first.

Multithreading brings additional complexity and additional capabilities to signal management. Signal semantics for multithreaded applications are described in the chapter entitled, “Programming with the Threads Library.” That chapter also describes the recommended paradigm for signal management in multithreaded programs.

Signal Types

There are two categories of signals, those generated externally (a break from a terminal) and those generated internally (a process fault). Both types are treated identically. The file **`/usr/include/signal.h`** defines the signals that may be delivered to a process.

The OS supports the following signals required by POSIX:

Table 10-1. POSIX Signals

Symbolic Name	Signal Event Description
`SIGABRT`	Abnormal termination (see **`abort(2)`**)
`SIGALRM`	Alarm time out (see **`alarm(2)`**)
`SIGFPE`	Floating-Point Exception / Erroneous Arithmetic Operation
`SIGHUP`	Hangup on controlling-terminal (see **`termios(2)`**)
`SIGILL`	Illegal hardware instruction / Invalid function image
`SIGINT`	Interactive attention *interrupt* (see **`termios(2)`**)
`SIGKILL`	Termination (cannot be caught or ignored)
`SIGPIPE`	Write onto pipe without readers (see **`write(2)`**)
`SIGQUIT`	Interactive termination *quit* (see **`termios(2)`**)
`SIGSEGV`	Invalid memory (segmentation) reference
`SIGTERM`	Termination
`SIGUSR1`	Reserved as application-defined signal 1
`SIGUSR2`	Reserved as application-defined signal 2

The OS supports the following job control signals:

Table 10-2. Job Control Signals

Symbolic Name	Signal Event Description
`SIGCHLD`	Child status changed
`SIGCONT`	Continue process execution
`SIGSTOP`	Stop process execution
`SIGTSTP`	Interactive stop (see **`termios(2)`**)
`SIGTTIN`	Stop tty input(see **`termios(2)`**)
`SIGTTOU`	Stop tty output (see **`termios(2)`**)

The OS supports the following real-time signals.

Table 10-3. Real-Time Signals

Symbolic Name	Signal Event Description
`SIGRT1`	Real-time signal 1
`SIGRT2`	Real-time signal 2
`SIGRT3`	Real-time signal 3
`SIGRT4`	Real-time signal 4
`SIGRT5`	Real-time signal 5
`SIGRT6`	Real-time signal 6
`SIGRT7`	Real-time signal 7
`SIGRT8`	Real-time signal 8

The OS supports the following additional signals:

Table 10-4. Additional Signals

Symbolic Name	Signal Event Description
`SIGBUS`	Bus error
`SIGEMT`	Emulation trap
`SIGPOLL`	Pollable event (see **`streamio(7)`**)
`SIGPWR`	Power fail / Restart
`SIGSYS`	Bad system call
`SIGTRAP`	Trace / Breakpoint trap
`SIGWINCH`	Window size change

Table 10-4. Additional Signals (Cont.)

Symbolic Name	Signal Event Description
`SIGXCPU`	CPU time limit exceeded (see **`getrlimit(2)`**)
`SIGXFSZ`	File size limit exceeded (see **`getrlimit(2)`**)
`SIGWAITING`	All LWPs blocked (for the Threads Library)
`SIGLWP`	Virtual interprocessor interrupt for the Threads Library
`SIGAIO`	Asynchronous I/O
`SIGURG`	Urgent condition on I/O channel
`SIGIO`	I/O possible or completed
`SIGPROF`	Profiling time alarm (see **`setitimer(3C)`**)
`SIGVTALRM`	Virtual time alarm (see **`setitimer(3C)`**)
`SIGPRE`	Programming exception
`SIGRESCHED`	An LWP has blocked rescheduling for longer than contracted

The signals fall into one of the following classes:

- Hardware conditions
- Software conditions
- Input/output notification
- Job control
- Resource control

Hardware signals are derived from exceptional conditions that may occur during execution. Such signals include `SIGBUS` for accesses that result in hardware-related errors, `SIGFPE` representing floating-point and other arithmetic exceptions, `SIGILL` for invalid instruction execution, and `SIGSEGV` for addresses outside the currently assigned area of memory or for accesses that violate memory protection constraints. Other, more CPU-specific hardware signals such as `SIGABRT, SIGEMT` and `SIGTRAP` may be defined by a specific implementation.

Software signals reflect interrupts generated by user request: `SIGINT` for the normal interrupt signal; `SIGQUIT` for the more powerful quit signal that normally causes a core image to be generated; `SIGHUP` and `SIGTERM` that cause graceful process termination, either because a user has “hung-up” or by user or program request; and `SIGKILL,` a more powerful termination signal that a process cannot catch or ignore. Programs may define their own asynchronous events using `SIGUSR1` and `SIGUSR2.` Other software signals such as `SIGALRM, SIGVTALRM, SIGPROF` indicate the expiration of interval timers.

A process can request notification via the signal `SIGPOLL` when input or output is possible on a file descriptor or when a nonblocking operation completes. A process may request to receive the signal `SIGURG` when an urgent condition arises.

A process may be stopped by a signal sent to it or the members of its process group (see **`termios(2)`**). The signal `SIGSTOP` is a powerful stop signal because it cannot be caught. Other stop signals—`SIGTSTP`, `SIGTTIN` and `SIGTTOU`—are used when a user request, input request, or output request, respectively, is the reason for stopping the process. The signal `SIGCONT` is sent to a process when it is continued from a stopped state. Processes may receive notification with the signal `SIGCHLD` when a child process changes state either by stopping or by terminating (see **`wait(2)`**).

Exceeding resource limits may cause signals to be generated. `SIGXCPU` occurs when a process nears its CPU time limit and `SIGXFSZ` warns that the limit on file size limit has been reached.

Three system signals, `SIGLWP`, `SIGWAITING`, and `SIGAIO`, are generated by the operating system for internal use by the Threads Library.

Signal Actions

Signals interrupt the normal flow of control in a process. They can affect a process in both the user mode and kernel mode. While a signal can interrupt the kernel mode only at certain points, the user mode must be prepared to handle a signal at any time. Signals do not direct the execution of a process, but rather request that the process take some action.

Associated with each signal is a default action, which is one of the following:

- Terminate the process with all of the consequences outlined in **`exit(2)`**
- Generate a **`core`** file, and terminate the process with all of the consequences outlined in **`exit(2)`**
- Stop the process
- Ignore the signal

The **`signal(5)`** system manual page contains a complete list of the OS signals and the corresponding default actions.

A user process specifies the action that is to be taken upon receipt of a particular signal. That action may be one of the following: take the default action for the signal, ignore the signal, or execute a user–specified signal–handling routine. The chosen action for each signal is known as the signal's *disposition*.

An interrupt signal may be sent, for example, by pressing an appropriate key on the terminal (Delete, Break, or Rubout). The action taken depends on the requirements of the specific program being executed. Examples are provided by the following:

- The shell invokes most commands in such a way that they stop executing immediately (die) when an interrupt is received; for example, the **`pr`** (print) command normally dies, allowing the user to stop unwanted output.
- The shell itself ignores interrupts when reading from the terminal because the shell should continue execution even when the user stops a command like **`pr`**.
- The editor **`ed`** chooses to catch interrupts so that it can halt its current action (especially printing) without allowing itself to be terminated.

A signal is *delivered* to a process when the default action is taken or the process's signal–handling routine is executed. Signal delivery resembles the occurrence of a hardware interrupt: the signal is normally blocked from further occurrence; the current process context is saved; and a new one is built. A signal is *caught* when it is delivered to the process's signal–handling routine.

A process can *block* one or more signals from delivery. Each process has a *signal set* that lists the signals whose delivery is currently being blocked. During the period of time between a signal's generation and its delivery, the signal is marked *pending*. If a blocked signal has been generated and is not being ignored by the process, it remains pending until the process either unblocks it or requests that it be ignored. The operating system selects one of the pending unblocked signals and delivers it to the process. If the process requests that a blocked signal be ignored, the signal is discarded as soon as it is generated.

The occurrence of a signal is recorded in the process table entry of the receiving process and is later recognized and acted upon by that process. During the posting of a signal, if the receiving process is sleeping, it is made runnable. If the signal is to be ignored, no action is taken, and the process continues sleeping. Signals are posted when they occur and are handled when the receiving process finds them. It is possible that the signal cannot be found until the completion of a system call, the occurrence of a process fault, or the resumption of a preempted user mode. When the process finds a signal, execution may be interrupted immediately; or, if the process is sleeping with a low enough priority, it may prematurely return from **`sleep`**, as explained previously, and branch directly to a signal-handling routine.

Real-Time Signal Behavior

To support real-time requirements, POSIX specifies real-time signal behavior to include the following:

- Specification of a range of real–time signal numbers
- Support for FIFO queuing of multiple occurrences of a particular signal when the signal is generated by using selected POSIX interfaces
- Specification of additional parameters to the signal–handling routine
- Support for specification of an application–defined value when a signal is generated by selected POSIX interfaces to allow for differentiation among multiple occurrences of signals of the same type
- Support for a signal code that indicates the reason for a signal and definition of code values that allow an application to determine whether an application–defined value is present

Each aspect of real-time signal behavior is described in the paragraphs that follow.

Certain signal numbers are reserved for use by real–time applications. The number of these signals is established by the value of RTSIG_MAX. This value is set in the file <**`limits.h`**>. It is available to an application through use of the **`sysconf(2)`** system call.

The range of real–time signal numbers is bound by the values of two symbolic constants: SIGRTMIN, which establishes the lowest real–time signal number, and SIGRTMAX, which establishes the highest real–time signal number. This range does not overlap the range of other signal numbers that are defined in the system. The values of SIGRTMIN and SIGRT-MAX and a corresponding range of real–time signal numbers for the OS are defined in the file <**`signal.h`**>. The default action for the real–time signals in the range SIGRTMIN to SIGRTMAX is to terminate the process.

Normally, multiple occurrences of the same signal number are indistinguishable—that is, the pending state of a signal merely indicates that one or more instances of that signal have been generated. An application may request that multiple occurrences of a signal that it receives be queued, with each occurrence being delivered independently. The circumstances under which it can do so are described in the paragraphs that follow.

A process declares an action to be taken upon receipt of a signal by invoking the **`sigaction(2)`** system call and supplying a pointer to a **`sigaction`** structure (see "The sigaction System Call," p. 10-17, and "The sigaction Structure," p. 10-9, respectively, for an explanation of this system call and a description of this structure). The **`sigaction`** structure contains a flags field that allows a process to modify the delivery of the signal. One of the flags that may be set is SA_SIGINFO. Setting the SA_SIGINFO flag requires that the signal–handling routine have three arguments: the signal number; a pointer to a **`siginfo_t`** structure, which provides information about the signal; and a pointer to a **`ucontext_t`** structure, which defines the user context of the process receiving the signal. An explanation of the interface to the signal–handling routine is provided in "The Signal–Handling Routine" (p. 10-29). The **`siginfo_t`** and **`ucontext_t`** structures are presented in "Signal Structures" (p. 10-8).

A process sets the SA_SIGINFO flag to indicate (1) that the signal–handling routine is to be passed a **`siginfo_t`** structure providing information about the signal and (2) that when a subsequent occurrence of a pending signal is generated, another **`siginfo_t`** structure is to be attached to the signal queue with that instance of the signal. The **`siginfo_t`** structure contains such information as the signal number and a code that indicates the reason for the signal. The supported codes are as follows: **SI_USER**, **SI_QUEUE**, **SI_TIMER**, **SI_ASYNCIO**, and **SI_MESGQ**. In certain cases, the **`siginfo_t`** structure also contains an application–defined value sent with the signal.

In order for a signal to be queued, the following condition must be met: the function that is generating the signal must be one that sends queued signals. The functions that send queued signals are the POSIX interfaces that allow a process to queue a signal and receive notification when a message arrives at a message queue, an asynchronous I/O operation is completed, and a POSIX timer expires. These interfaces include the **`sigqueue(2)`** system call and the **`aio_read(3C)`**, **`aio_write(3C)`**, **`lio_listio(3C)`**, **`aio_fsync(3C)`**, and **`timer_create(3C)`** library routines. See the *PowerMAX OS Real-Time Guide* for information on **`mq_notify`**, **`aio_read`**, **`aio_write`**, **`lio_listio`**, **`aio_fsync`**, and **`timer_create`**. See "The sigqueue System Call" (p. 10-27) for information on **`sigqueue`**. When these interfaces are used, one of the following codes is passed to the signal–handling routine in the **`siginfo_t`** structure: **SI_QUEUE**, **SI_MESQ**, **SI_ASYNCIO**, **or** **SI_TIMER**. An application–defined value is also passed in the structure.

Support for specification of an application–defined value when a signal is sent to a process is provided by the **`sigevent`** structure. A process uses this structure to indicate the type of notification mechanism to be used when an asynchronous event occurs, the number of the signal that is to be generated, and the application–defined value that is to be used to differentiate one occurrence of the signal from another. A **`sigval`** union is defined for specification of the application–defined value so that either an integer or a pointer can be passed to the signal–handling routine. The **`sigevent`** structure and **`sigval`** union are presented in “The sigval and sigevent Structures” (p. 10-11). An application–defined value may be associated with any signal that is defined in the OS.

When the conditions necessary for queuing a signal are met, subsequent occurrences of any signal that is defined in the OS are queued to the receiving process in FIFO (first–in–first–out) order. The limit on the number of queued signals that a process may send and that are still pending at the receiver(s) at any time is 32. A process can obtain this value by using the **`sysconf(2)`** system call.

A process may invoke **`sigaction(2)`** with the SA_SIGINFO flag set and receive signals that are not queued. Such signals include those generated by the **`kill(2)`** and **`sigsend(2)`** system calls and those generated by the kernel. When a signal is generated by a **`kill`** or **`sigsend`** system call, only one occurrence of the specified signal is ever pending—for example, if the process sending the signal sends two signals before the receiving process can receive one of them, the receiving process will receive only one instance of the signal. When these interfaces are used, a **`siginfo_t`** structure is queued to mark the signal pending. The code that is passed to the signal–handling routine in this structure is **SI_USER**. A value is not passed because these interfaces do not allow the sending process to specify one. See “The kill System Call” (p. 10-16) for information on **`kill`** and “System V Signal System Calls” (p. 10-28) for information on **`sigsend`**.

Signals generated by the kernel include those for which the code passed to the signal–handling routine in the **`siginfo_t`** structure is a positive number. A positive number is passed for signals that provide additional information in the **`siginfo_t`** structure; this number can be used with the number of the signal that has been delivered to determine the cause of the signal (for additional information, refer to the **`signal(5)`** system manual page). When the code is a positive number, a value is not passed.

When the SA_SIGINFO flag is not set on a call to **`sigaction(2)`**, the arguments that are passed to the signal–handling routine are different from those that are passed when SA_SIGINFO is set. They include the signal number and possibly one or more additional arguments that vary according to architecture (for a description of these arguments, refer to the section on signal handlers in the **`signal(5)`** system manual page).

Signal Structures

The structures and unions that are used by the POSIX signal–management facilities include the following: **`sigset_t`**, **`sigaction`**, **`sigval`**, **`sigevent`**, **`siginfo_t`**, and **`ucontext_t`**. The **`sigset_t`** structure is presented in “The sigset_t Structure.” The **`sigaction`** structure is presented in “The sigaction Structure.” The **`sigval`** union and **`sigevent`** structure are presented in “The sigval and sigevent Structures.” The **`siginfo_t`** structure is presented in “The siginfo_t Structure.” The **`ucontext_t`** structure is presented in “The ucontext_t Structure.”

The sigset_t Structure

The **`sigset_t`** structure specifies a set of signals. It is defined in the file <**`signal.h`**>. A process defines and manipulates a set of signals by using the group of library routines described in the **`sigsetops(3C)`** system manual page. These routines are presented in “The sigsetops Library Routines” (p. 10-17).

You supply a pointer to a **`sigset_t`** structure when you invoke a number of the system calls and routines that allow you to manage signals. Such calls and routines include **`sigaction(2)`**, **`sigprocmask(2)`**, **`sigpending(2)`**, **`sigsuspend(2)`**, **`sigtimedwait(2)`** and those described in **`sigsetops(2)`**. All are described in "POSIX Signal System Calls" (p. 10-16).

The sigaction Structure

The **`sigaction`** structure defines the action to be associated with a particular signal. You supply a pointer to this structure when you invoke the **`sigaction(2)`** system call to specify or obtain an action for a signal (for information on this call, see "The sigaction System Call," p. 10-17).

The **`sigaction`** structure is defined in **`<signal.h>`** as follows:

```
struct sigaction {
  union {
    void (*sa_handler);
    void (*sa_sigaction) (int, siginfo_t *, void *);
  } sa_func;
  sigset_t sa_mask;
  int sa_flags;
};
```

The fields in the structure are described as follows.

sa_func — one of two special values, SIG_DFL or SIG_IGN, or a pointer to a process's signal–handling routine. SIG_DFL specifies the default action for the signal (for information on the default action for a particular signal, refer to the **`signal(5)`** system manual page). SIG_IGN specifies that the signal is to be ignored. The **`signal.h`** header file expands each of these special values into a distinct constant expression of the type `(void(*)())`, whose value matches no declarable function.

sa_mask — specifies a set of signals that is to be blocked while the signal–handling routine is active. On entry to the signal–handling routine, this set of signals will be added to the set of signals already being blocked when the signal is delivered. The signal for which the signal–handling routine is invoked will also be blocked unless the SA_NODEFER bit is set in the **`sa_flags`** field. Note that the system does not allow the SIGSTOP or the SIGKILL signal to be blocked.

sa_flags — contains zero or an integer value that sets one or more of the following bits:

SA_ONSTACK — causes the signal to be delivered on an alternate stack if the signal is caught (delivered to a signal–handling routine) and an alternate signal stack has been defined by using the **`sigaltstack(2)`** system call. If this bit is not set, the signal is delivered on the same stack as that on which the main process is executing.

SA_RESETHAND — causes the action for the signal to be reset to SIG_DFL if the signal is caught. Note that the system does not allow the SIGILL, SIGTRAP, and SIGPWR signals to be reset automatically (for additional information on these signals, refer to the **signal(5)** system manual page).

SA_NODEFER prevents the signal from being automatically blocked by the kernel while it is being caught

SA_RESTART if the signal is caught, causes a system call that is interrupted by execution of the signal–handling routine to be restarted

SA_SIGINFO if the signal specified by *sig* on the call to **sigaction(2)** is caught, causes the signal number and two additional arguments to be passed to the associated signal–handling routine: a pointer to a **siginfo_t** structure and a pointer to a **ucontext_t** structure. The **siginfo_t** structure contains a code that identifies the reason for the signal. The **ucontext_t** structure contains the context of the receiving process at the time that the signal was delivered. For additional information on these structures, see “The siginfo_t Structure” (p. 10-13) and “The ucontext_t Structure” (p. 10-15), respectively. If the signal is generated by an interface that sends queued signals, then subsequent occurrences of the signal are queued to the receiving process in FIFO (first–in–first–out) order; with each occurrence, another **siginfo_t** structure is queued (see “The siginfo_t Structure,” p. 10-13, for a more detailed explanation of the use of this flag).

If the signal is caught and this bit is <u>not</u> set, the arguments that are passed to the signal–handling routine are different from those just described. They include the signal number and possibly one or more additional arguments that vary according to architecture. For an explanation of the handler interface to be used on your system, refer to the section on signal handlers in the **signal(5)** system manual page.

SA_NOCLDWAIT if the signal specified by *sig* on the call to **sigaction(2)** is set to SIGCHLD, prevents the system from creating zombie processes when children of the calling process exit. If the calling process subsequently invokes the **wait(2)** system call, **wait(2)** does not return until all of the child processes terminate. When **wait(2)** does return, it returns a value of **–1**; **errno** is set to ECHILD.

SA_NOCLDSTOP if the signal specified by *sig* on the call to **sigaction(2)** is set to SIGCHLD, prevents that signal from being sent to the calling process when its child processes stop or continue

The sigval and sigevent Structures

The POSIX routines use two structures to specify an application–defined value to be delivered with a signal. The **sigval** union and **sigevent** structure are defined in **<sys/siginfo.h>**. The **sigval** union provides a means of specifying an application–defined value as either an integer or a pointer. The **sigevent** structure provides a means of passing a signal number and a value to a function that will cause a signal to be sent upon the occurrence of some event. These structures are used by the POSIX routines that allow you to specify a signal and the value that is to be passed to the signal–handling routine when the signal is delivered.

The **sigevent** structure contains a **notifyinfo** union. The **notifyinfo** union provides support for the call-back mechanism—a faster, more deterministic mechanism for asynchronous event notification than delivery of a signal. The call-back mechanism is a Concurrent extension that is not POSIX-compliant.

The **notifyinfo** union allows you to specify a routine that is to be called when a certain type of event occurs and to specify an application-defined value that is to be passed to that routine. The functions that support use of the call-back mechanism are the POSIX interfaces that allow a process to receive notification when a message arrives at a queue, an asynchronous I/O operation is completed, and a POSIX timer expires. These interfaces are as follows: **mq_notify(3C)**, **aio_read(3C)**, **aio_write(3C)**, **lio_listio(3C)**, **aio_fsync(3C)**, and **timer_create(3C)**. Procedures for using these interfaces are fully described in the *PowerMAX OS Real-Time Guide.*

The **sigval** union is presented as follows:

```
union sigval {
       int sival_int;
       void *sival_ptr;
};
```

The fields in the union are described as follows.

sival_int	an application–defined value of type integer. When the signal is delivered, this value is to be passed to the signal–handling routine as the **si_value** component of the **siginfo_t** structure (see "The siginfo_t Structure," p. 10-13, for an explanation of this structure and "The Signal–Handling Routine," p. 10-29, for an explanation of the procedures for defining the signal–handling routine).
sival_ptr	an application–defined value of type pointer. When the signal is delivered, this value is to be passed to the signal–handling routine as the **si_value** component of the **siginfo_t** structure (see "The siginfo_t Structure," p. 10-13, for an explanation of this structure and "The Signal–Handling Routine," p. 10-29, for an explanation of the procedures for defining the signal–handling routine).

The **notifyinfo** union is presented as follows:

```
union notifyinfo {
       int nisigno;
       void (*nifunc)(union sigval);
};
```

The fields in the union are described as follows.

sigev_signo	the number of the signal that is to be sent to a process. A set of symbolic constants has been defined to assist you in specifying signal numbers. These constants are defined in the file <**`signal.h`**>.
sigev_notify	a pointer to a process's call-back routine

The **`sigevent`** structure is presented as follows:

```
struct sigevent {
        int              sigev_notify;
        union notifyinfo sigev_notifyinfo;
        union sigval     sigev_value;
};

#define sigev_func       sigev_notifyinfo.nifunc
#define sigev_signo      sigev_notifyinfo.nisigno
```

The fields in the structure are described as follows.

sigev_notify — an integer value that specifies the notification mechanism to be used when an asynchronous event occurs. This value must be one of the following:

SIGEV_NONE	indicates that no asynchronous notification is to be delivered when the event of interest occurs
SIGEV_SIGNAL	indicates that a queued signal with an application–defined value is to be generated when the event of interest occurs
SIGEV_CALLBACK	indicates that an application-defined call-back routine is to be called when the event of interest occurs The call-back mechanism is a Concurrent extension that provides a more efficient mechanism for asynchronous event notification. When the event of interest occurs, a bound daemon thread, which is blocked in the kernel, is wakened. This thread immediately returns to user space to execute the application-defined call-back routine. Note that the call-back mechanism is <u>not</u> POSIX-compliant.

sigev_signo	if the value of the **sigev_notify** field is **SIGEV_SIGNAL**, the number of the signal that is to be sent to a process. A set of symbolic constants has been defined to assist you in specifying signal numbers. These constants are defined in the file <**signal.h**>.
sigev_func	if the value of the **sigev_notify** field is **SIGEV_CALLBACK**, a pointer to an application-defined call-back routine
sigev_value	an application–defined value that is to be used by the signal–handling routine for the signal specified by **sigev_signo** or an application-defined value that is to be passed as an argument to the call-back routine.specified by **sigev_func** This value may be an integer or a pointer.

The siginfo_t Structure

The **siginfo_t** structure contains information about a signal that has been delivered to a process. Such information includes the signal number, a code that indicates the reason for the signal, and an error number associated with the signal. The structure also contains the PID and user ID of the sending process if the signal has been generated by a user process using a POSIX function that sends queued signals or by a user process using the **kill(2)** or **sigsend(2)** system call. Note that in the case of **kill** or **sigsend**, when a signal of the same signal number is already pending delivery to the receiving process, that process will receive only one instance of the signal. If the signal has been generated by the kernel, the structure contains additional information that is specific to the signal—that is, the address of the faulting instruction in the case of a SIGILL signal, the PID of the child process in the case of a SIGCHLD signal, and so on.

If the SA_SIGINFO flag is set when the **sigaction(2)** system call is invoked to declare a handling routine for a particular signal, a pointer to a **siginfo_t** structure is supplied as an argument to the signal–handling routine. For information on use of the **sigaction(2)** system call, see "The sigaction System Call" (p. 10-17). For an explanation of the interface to the signal–handling routine, see "The Signal–Handling Routine" (p. 10-29).

The **siginfo_t** structure is defined in **<sys/siginfo.h>**. The following fields are among those defined in the structure. Note that the only fields that the program can examine are those that are defined for the current signal number and signal code value.

```
int                     si_signo;
int                     si_code;
int                     si_errno;
pid_t                   si_pid;
uid_t                   si_uid;
int                     si_value;
int                         si_status;
caddr_t                     si_addr;
```

These fields are described as follows.

si_signo contains the number of the signal that has been delivered to the process

si_code contains an integer value that indicates the reason that the signal defined by the **si_signo** field has been generated. Signal codes are defined in the file <**sys/siginfo.h**>. If the value of **si_code** is one of the following, a user process has generated the signal:

SI_USER	indicates that the signal has been sent by the **kill(2)** or **sigsend(2)** system call (see "The kill System Call," p. 10-16, and "System V Signal System Calls," p. 10-28, for explanations of these calls)
SI_QUEUE	indicates that the signal has been sent by the **sigqueue(2)** system call (see "The sigqueue System Call," p. 10-27, for an explanation of this call)
SI_TIMER	indicates that the signal has been generated by the expiration of a POSIX timer set by the **timer_settime(3P4)** library routine (refer to the *PowerMAX OS Real-Time Guide* for information on POSIX clocks and timers)
SI_ASYNCIO	indicates that the signal has been generated by the completion of an asynchronous I/O operation (refer to the *PowerMAX OS Real-Time Guide* for information on asynchronous I/O)
SI_MESGQ	indicates that the signal has been generated by the arrival of a message at an empty message queue (refer to the *PowerMAX OS Real-Time Guide* for information on POSIX message queues)

If the value of **si_code** is different from these, the signal has been generated by the kernel. In this case, the **si_code** may be a positive number. A positive number indicates that the signal is one that provides additional information in the **siginfo_t** structure; the number can be used with the number of the signal that has been delivered to determine the reason for the signal (refer to the file <**siginfo.h**> for a brief statement of the reason). It is likely that the signal has resulted from a hardware exception; such signals include SIGFPE, SIGILL, SIGSEGV, SIGBUS, and SIGTRAP.

si_errno contains zero or an error number associated with the signal specified by **si_signo**. Error numbers are defined in the file <**errno.h**>. Currently, there is no condition that results in the return of an error number.

si_pid	if a user process has generated the signal, contains the process identification number (PID) of that process. If the value of **si_signo** is SIGCHLD, this field contains the PID of the child process.
si_uid	if a user process has generated the signal, contains the user ID of that process
si_value	if the value of **si_code** is **SI_QUEUE**, **SI_TIMER**, or **SI_MESGQ**, contains an application–defined value that was specified by the sender of the signal. If the value of **si_code** is **SI_ASYNCIO**, **si_value** contains the identifier for the completed asynchronous I/O operation.
si_status	if the value of **si_signo** is SIGCHLD, contains the child process's exit value or the number of the signal that has terminated the child process
si_addr	if the value of **si_signo** is SIGILL, contains the virtual address of the faulting instruction. If the value of **si_signo** is SIGSEGV, **si_addr** contains the virtual address of the faulting memory reference.

The ucontext_t Structure

The **ucontext_t** structure shows the user context of a process at the point at which a particular signal is delivered. Its contents include the process's signal set, execution stack, and machine registers.

If the SA_SIGINFO flag is set when the **sigaction(2)** system call is invoked to declare a signal–handling routine for a particular signal, a pointer to this structure is supplied as an argument to the handling routine. (For information on use of the **sigaction(2)** system call, see "The sigaction System Call," p. 10-17. For an explanation of the interface to the signal–handling routine, see "The Signal–Handling Routine," p. 10-29)

The **ucontext_t** structure is defined in **<sys/ucontext.h>** as follows:

```
typedef struct ucontext {

  . . .

  sigset_t   uc_sigmask;
  stack_t    uc_stack;

  . . .

  mcontext_t uc_mcontext;

  . . .

} ucontext_t;
```

The fields in the structure are described as follows.

uc_sigmask	specifies the set of signals whose delivery is blocked
uc_stack	if an alternate stack has been defined by using the **sigaltstack(2)** system call, specifies an alternate user

	stack on which signals are to be processed (for information on the **sigaltstack(2)** system call, refer to “System V Signal System Calls,” p. 10-28).
uc_mcontext	contains an array of the saved set of machine registers. Note that portable applications should not access or modify this array.

POSIX Signal System Calls

POSIX.1 defines a number of functions that facilitate signal management. They are briefly described as follows:

kill(2)	send a signal to a process or a group of processes
sigsetops(3C)	manipulate a signal set
sigaction(2)	define or obtain an action for a signal
sigprocmask(2)	obtain or modify a signal set
sigpending(2)	identify pending signals
sigsuspend(2)	wait for a signal
sigtimedwait(2)	wait a specified period of time for receipt of a signal
sigwaitinfo(2)	wait indefinitely for receipt of a signal
sigqueue(2)	queue a signal and an application–defined value to a process

Each of these functions is described in detail in the sections that follow.

The kill System Call

A process can send a signal to another process or group of processes by using **kill(2)**:

```
kill(pid, signo

pid_t  pid;
int    signo;
```

Unless the process sending the signal is privileged, its real or effective user ID must be equal to the receiving process's real or saved user ID. If the Enhanced Security Utilities are installed, then the unprivileged sending process's security level must be equal to the receiving process's level

As explained in “Real-Time Signal Behavior” (p. 10-6), the **kill** system calls does not send queued signals.

The sigsetops Library Routines

A set of C library routines, which is described in the **`sigsetops(3C)`** system manual page, allows a calling process to manipulate a signal set. A signal set is a list of signals. Among other things, a process may use it to define the signals whose delivery is to be blocked.

A process supplies a pointer to a signal set as an argument to a number of the system calls that are used to manage signals—for example, those that allow a process to define an action for a signal (**`sigaction(2)`**), wait for one or more signals (**`sigsuspend(2)`** or **`sigtimedwait(2)`**), and identify pending signals (**`sigpending(2)`**).

The routines in the **`sigsetops`** set are briefly described as follows:

`sigemptyset`	initialize a signal set by excluding all of the signals that are defined for the system
`sigfillset`	initialize a signal set by including all of the signals that are defined for the system
`sigaddset`	add a specified signal to an existing signal set
`sigdelset`	delete a specified signal from an existing signal set
`sigismember`	determine whether or not a specified signal is included in a signal set

For the specifications required for making these calls, the return values, and other details, refer to the **`sigsetops(3C)`** system manual page.

It is important to note that you must initialize a signal set prior to using it. Typically an application first invokes **`sigemptyset`** and then invokes **`sigaddset`** one or more times to define a particular set of signals. If a process has not initialized a signal set prior to specifying it as an argument to a system call, the results of the call are undefined.

The sigaction System Call

The **`sigaction(2)`** system call allows a process to specify an action to be taken upon receipt of a particular signal and to obtain information about the action that has previously been specified for a signal.

The specifications required for making the **`sigaction`** call are as follows:

```
#include <signal.h>

int sigaction(sig, act, oact)

int sig;
struct sigaction *act;
struct sigaction *oact;
```

The arguments are defined as follows:

sig	the number of the signal for which the action is being specified. A set of symbolic constants has been defined to assist you in specifying signal

numbers. These constants are defined in the file <**`signal.h`**>, which must always be included when signals are used.

act the null pointer constant or a pointer to a structure that defines the action for the specified signal. If the value of *act* is **NULL**, the action that is currently defined for the signal is not changed; it may be returned in the structure pointed to by *oact*. The **`sigaction`** structure is described in "The sigaction Structure" (p. 10-9).

oact the null pointer constant or a pointer to a structure to which information about the action previously associated with the specified signal is returned. The **`sigaction`** structure is described in "The sigaction Structure" (p. 10-9).

A return value of **0** indicates that the call has been successful. A return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. If an error occurs, a signal–handling routine is not installed. The action that has previously been defined for the signal is not changed. Refer to the **`sigaction(2)`** system manual page for a listing of the types of errors that may occur.

Initially, all signals are set to SIG_DFL or SIG_IGN prior to entry of the function **`main`** (see **`exec(2)`**). Once an action is established for a specific signal, it usually remains established until another action is explicitly established by a call to **`signal(2)`**, **`sigset(2)`**, **`sigignore(2)`**, or **`sigaction(2)`** or until the process calls **`fork(2)`** or **`exec(2)`**. A child process inherits the actions of the parent for the defaulted and ignored signals. Caught signals are reset to the default action in the child process. This is necessary because the address linkage for signal-handling routines specified in the parent are no longer appropriate in the child. When a process invokes **`exec`**, all signals set to catch the signal are reset to SIG_DFL. Alternatively, a process may request that the action for a signal automatically be reset to SIG_DFL after catching it.

In the example shown in Screen 10-1, the first call to **`sigaction`** causes interrupts to be ignored; while the second call to **`sigaction`** restores the default action for interrupts, which is to terminate the process. In both cases, **`sigaction`** returns the previous signal action in the final argument `old_act`.

```
#include <signal.h>

main() {
   struct sigaction new_act, old_act;

   new_act.sa_handler = SIG_IGN;
   sigaction(SIGINT, &new_act, &old_act);

   /* do processing */

   new_act.sa_handler = SIG_DFL;
   sigaction(SIGINT, &new_act, &old_act);
}
```

Screen 10-1. Example Specifying SIG_IGN or SIG_DFL

Instead of the special values SIG_IGN or SIG_DFL, the second argument to **`sigaction`** may specify a pointer to a signal-handling routine; in this case, the specified routine is called when the signal occurs. Most commonly this facility is used to allow the program to clean up unfinished business before terminating—to delete a temporary file, for example, as illustrated in Screen 10-2:

```
#include <signal.h>

main() {
   struct sigaction new_act, old_act;
   void on_intr();

   new_act.sa_handler = SIG_IGN;
   sigaction(SIGINT, &new_act, &old_act);

   if (old_act.sa_handler != SIG_IGN) {
      new_act.sa_handler = on_intr;
      sigaction(SIGINT, &new_act, &old_act);
   }

   /* do processing */

   exit(0);       /* exit with normal status */
}

void on_intr() {

   unlink(tempfile);

   exit(1);       /* exit with interrupted status */
}
```

Screen 10-2. Example Specifying a Pointer to a Handler

Before establishing **`on_intr`** as the signal handler for `SIGINT`, the program tests the state of interrupt handling and continues to ignore interrupts if they are already being ignored. This is needed because `SIGINT` is sent to all processes started from a specific terminal. Accordingly, when a program is initiated with an ampersand (**`&`**) to run without any interaction in the background, the shell turns off interrupts for it so that it will not be stopped by interrupts intended for foreground processes. If this program began by setting **`on_intr`** to catch all interrupts regardless, that would undo the shell's efforts to protect it when run in the background. The solution, shown in Screen 10-2, is to call **`sigaction`** for `SIGINT` first to get the signal action currently established for the interrupt signal, which is returned in the third argument to **`sigaction`**. If interrupt signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught. In that case, the second call to **`sigaction`** for `SIGINT` establishes a new signal action that specifies **`on_intr`** as the signal handler.

A more sophisticated program may wish to intercept and interpret `SIGINT` as a request to stop what it is doing and return to its own command processing loop. Think of a text editor: interrupting a long printout should not cause it to terminate and lose the work already done. The outline of the code for this case is probably best written as illustrated in Screen 10-3:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf sjbuf;

main() {
   struct sigaction new_act, old_act;
   void on_intr();

   new_act.sa_handler = SIG_IGN;
   sigaction(SIGINT, &new_act, &old_act);

   setjmp(sjbuf);   /* save current stack position */

   if (old_act.sa_handler != SIG_IGN) {
      new_act.sa_handler = on_intr;
      sigaction(SIGINT, &new_act, &old_act);
   }
/*
 * main command processing loop
 */
   exit(0)
}

void on_intr() {

   printf("\nInterrupt\n");   /* print message */

   longjmp(sjbuf);   /* return to saved state */
}
```

Screen 10-3. Example Detecting SIGINT Signal

The **<setjmp.h>** header file declares the type `jmp_buf` for a buffer in which the state can be saved, and the program shown in Screen 10-3 declares `sjbuf` to be of type `jmp_buf`, which is an array of some type. The function **setjmp** saves the current context of the user process in `sjbuf`. When an interrupt occurs, a call to the function **on_intr** is forced, which prints a message and can set flags or do something else. The function **longjmp** takes as argument an object stored into by **setjmp**, and restores control to the location after the call to **setjmp**, so control (and the stack level) pops back to the place in the program **main** where the signal is set up and the main loop entered. Notice, by the way, that the signal gets set again after an interrupt occurs. This is necessary; most signals are automatically reset to their default action when they occur.

Some programs that want to detect signals simply cannot be stopped at an arbitrary point—in the middle of updating a linked list for example. If the function called on occurrences of a signal sets a flag and then returns instead of calling **exit** or **longjmp**, execution resumes at the exact point at which it was interrupted. The interrupt flag can then be tested later.

This approach has the following difficulty. Suppose the program is reading the terminal when the interrupt is sent. The specified function is duly called; it sets its flag and returns. If it were really true, as said earlier, that execution resumes at the exact point at which it was interrupted, the program would continue reading the terminal until the user typed another line. This behavior might well be confusing because the user might not know the program is reading and, presumably, would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the **read** from the terminal when execution resumes after the signal, with **read** returning an error code (`EINTR`) that indicates the interruption.

As a consequence, programs that catch signals and resume execution afterward should be prepared for errors caused by interrupted system calls. (The ones to watch out for in particular are **`wait`** and **`pause`** as well as any **`read`** from the terminal.)

A program whose **`on_intr`** function just sets `intflag`, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input or reads directly from a terminal device.

```
if (getchar() == EOF)
   if (intflag)
      /* EOF caused by interrupt */
   else
      /* actual end-of-file */
```

A final subtlety to keep in mind becomes important when signal handling is combined with execution of other programs. Suppose a program handles interrupts, and also includes a method (like “ ! ” in the editor) whereby other programs can be executed. Then the code should look something like the following:

```
if (fork() == 0)
   exec( ... );
new_act.sa_handler = SIG_IGN; /* ignore interrupts */
sigaction(SIGINT, &new_act, &old_act);
wait(&status);         /* until the child completes */
new_act.sa_handler = on_intr; /* restore interrupts */
sigaction(SIGINT, &new_act, &old_act);
```

Why is this? Again, it is not obvious but not really difficult. Suppose the program called catches its own interrupts. When this subprogram gets interrupted, it receives the signal, returns to its main loop, and probably tries to read the terminal. But the calling program also pops out of its wait for the subprogram and tries to read the terminal. If two processes try to read the terminal, it is very unfortunate because the system randomly decides which should get each line of input. A simple solution is for the parent to ignore interrupts until the child completes. This reasoning is reflected in the function **`system`** that Screen 10-4 illustrates.

```
#include <signal.h>

system(cmd_str) /* run command string */
   char *cmd_str;
{
   int status;
   pid_t wpid, xpid;
   struct sigaction sig_act, i_stat, q_stat;

   if ((xpid=fork()) == 0) {
      execl(“/bin/sh”, “sh”, “-c”, cmd_str, 0);
      _exit(127);
   }

   sig_act.sa_handler = SIG_IGN;
   sigaction(SIGINT, &sig_act, &i_stat);

   sig_act.sa_handler = SIG_IGN;
   sigaction(SIGQUIT, &sig_act, &q_stat);

   while ( ((wpid=wait(&status)) != xpid) && (wpid != -1) )
      ;
   if (wpid == -1)
      status = -1;

   sigaction(SIGINT, &i_stat, &sig_act);
   sigaction(SIGQUIT, &q_stat, &sig_act);

   return(status);
}
```

Screen 10-4. System() Function

The sigprocmask System Call

The **sigprocmask(2)** system call allows the calling process to obtain or modify the set of signals whose delivery is currently being blocked.

The specifications required for making the **sigprocmask** call are as follows:

```
#include <signal.h>

int sigprocmask(how, set, oset)

int how;
sigset_t *set;
sigset_t *oset;
```

The arguments are defined as follows:

how an integer value that indicates the way in which the set of signals currently being blocked is to be changed. The value of *how* must be one of the following:

SIG_BLOCK	adds the signals to which *set* points to the set of signals currently being blocked
SIG_UNBLOCK	removes the signals to which *set* points from the set of signals currently being blocked

SIG_SETMASK	replaces the set of signals currently being blocked with the signals to which *set* points

set the null pointer constant or a pointer to a structure that specifies the set of signals that is to be used to change the set of signals currently being blocked. Note that the SIGKILL and SIGSTOP signals are ignored if they are included in the signal set. If the value of *set* is **NULL**, the set of signals currently being blocked is not changed; this set may be returned in the structure pointed to by *oset*. The **sigset_t** structure is described in "The sigset_t Structure" (p. 10-8).

oset the null pointer constant or a pointer to a structure to which the set of signals currently being blocked is returned. The **sigset_t** structure is described in "The sigset_t Structure" (p. 10-8).

A return value of **0** indicates that the call has been successful. A return value of **–1** indicates that an error has occurred; **errno** is set to indicate the error. If an error occurs, the process's signal set is not changed. Refer to the **sigprocmask(2)** system manual page for a listing of the types of errors that may occur.

The **sigprocmask** system call provides a mechanism whereby critical sections of code may protect themselves against the occurrence of specified signals.

To block a section of code against one or more signals, the following call may be used to add a set of signals to the existing mask and return the old mask:

```
sigprocmask (SIG_BLOCK, &new_set, &old_set);
```

The old mask can then be restored later with the following call:

```
sigprocmask (SIG_UNBLOCK, &new_set, &old_set);
```

It is possible to check conditions with some signals blocked, pause waiting for a signal, and then restore the mask by using a call to **sigsuspend(2)**. For more information on this call, refer to "The sigsuspend System Call" (p. 10-24).

The sigpending System Call

The **sigpending(2)** system call allows the calling process to obtain the set of signals that have been sent but are currently being blocked from delivery.

The specifications required for making the **sigpending** call are as follows:

```
#include <signal.h>

int sigpending(set)

sigset_t *set;
```

The argument is defined as follows:

set a pointer to a location to which the set of currently pending signals is returned. The **sigset_t** structure is described in "The sigset_t Structure" (p. 10-8).

A return value of **0** indicates that the call has been successful. A return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`sigpending(2)`** system manual page for a listing of the types of errors that may occur.

The sigsuspend System Call

The **`sigsuspend(2)`** system call allows the calling process to suspend execution until delivery of a signal whose associated action is termination or execution of a signal–handling routine. It also allows the process to replace the current signal set with a different set. If the action associated with a signal is execution of a signal–handling routine, the **`sigsuspend`** call returns after the signal–handling routine returns, and it restores the signal set that existed prior to the call to **`sigsuspend`**.

The specifications required for making the **`sigsuspend`** call are as follows:

```
#include <signal.h>

int sigsuspend(set)

sigset_t *set;
```

The argument is defined as follows:

set	a pointer to the set of signals that is to replace the set whose delivery is currently being blocked

Because the **`sigsuspend`** system call suspends process execution indefinitely, it does not return a value to indicate that the call has been successful. A return value of **–1** indicates that the calling process has caught a signal and that control has returned from the signal–handling routine. **`Errno`** is set accordingly. Refer to the **`sigsuspend(2)`** system manual page for additional information.

The sigtimedwait System Call

The **`sigtimedwait(2)`** system call allows the calling process to wait for a signal in a particular signal set for a specified period of time. If any of the signals in the set are pending at the time of the call, **`sigtimedwait`** selects the signal with the lowest number and returns such information as the signal number, the reason that the signal has been generated, and, if applicable, an application–defined value that has been queued with the signal. If none of the signals in the set are pending at the time of the call, **`sigtimedwait`** suspends the process until one of the following occurs:

- One or more of the signals in the set are generated or pending delivery.

  In this case, **`sigtimedwait`** returns the number of the selected signal and the reason that it has been generated. The signal–handling routine for that signal is not invoked.

- The specified period of time elapses.

  In this case, **`sigtimedwait`** returns a value of **–1** and sets **`errno`** to EAGAIN.

- The process is interrupted by a signal that is not in the set.

  In this case, the signal–handling routine for the signal is invoked if the signal is being caught; **`sigtimedwait`** returns a value of **–1** and sets **`errno`** to EINTR. If the signal is not being caught, the process is terminated.

The **`sigtimedwait`** and **`sigpending`** system calls are similar in that both notify a process that a signal has been sent. You may wish to use **`sigtimedwait`** instead of **`sigpending`** because it returns information about the signal and does not require that a signal–handling routine be called.

The specifications required for making the **`sigtimedwait`** call are as follows:

```
#include <sys/siginfo.h>
#include <signal.h>
#include <sys/timers.h>

int sigtimedwait(set, info, timeout)

const sigset_t *set;
siginfo_t *info;
const struct timespec *timeout;
```

The arguments are defined as follows:

set	a pointer to a structure that specifies the signal(s) for which the process is to wait.
info	the null pointer constant or a pointer to a structure to which information about the selected signal is returned. If the value of *info* is not **NULL**, on return the **`si_signo`** component contains the number of the selected signal, and the **`si_code`** component contains the code that identifies the reason for the signal. The signal numbers that may be returned are defined in the file <**`signal.h`**>; the signal codes are defined in the file <**`sys/siginfo.h`**>. If an application–defined value has been queued to the process with that signal, the **`si_value`** component contains that value; if not, the content of the **`si_value`** component is undefined.
timeout	the null pointer constant or a pointer to a structure that specifies the length of time that the process is to wait for a signal. If the value of *timeout* is **NULL**, the process will wait indefinitely. If the structure to which *timeout* points contains zeros and none of the signals specified by the *set* argument are pending, an `EAGAIN` error occurs; the **`sigtimedwait`** call returns immediately.

If one of the signals specified by the *set* argument is generated or is pending delivery, **`sigtimedwait`** returns the number of the selected signal in the structure to which *info* points. A return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`sigtimedwait(2)`** system manual page for a listing of the types of errors that may occur.

The sigwaitinfo System Call

The **`sigwaitinfo(2)`** system call allows the calling process to wait indefinitely for a signal in a particular signal set. If any of the signals in the set are pending at the time of the call, **`sigwaitinfo`** selects the signal with the lowest number and returns such information as the signal number, the reason that the signal has been generated, and, if applicable, an application–defined value that has been queued with the signal. If none of the signals in the set are pending at the time of the call, **`sigwaitinfo`** suspends the process until one of the following occurs:

- One or more of the signals in the set are generated or pending delivery

  In this case, **`sigwaitinfo`** returns the number of the selected signal and the reason that it has been generated. The signal–handling routine for that signal is not invoked.

- The process is interrupted by a signal that is not in the set

  In this case, the signal–handling routine for the signal is invoked if the signal is being caught; **`sigwaitinfo`** returns a value of **–1** and sets **`errno`** to EINTR. If the signal is not being caught, the process is terminated.

The specifications required for making the **`sigwaitinfo`** call are as follows:

```
#include <sys/siginfo.h>
#include <signal.h>

int sigwaitinfo(set, info)

const sigset_t *set;
siginfo_t *info;
```

The arguments are defined as follows:

set	a pointer to a signal set that specifies the signal(s) for which the process is to wait.
info	the null pointer constant or a pointer to a structure to which information about the selected signal is returned. If the value of *info* is not NULL, on return the **`si_signo`** component contains the number of the selected signal, and the **`si_code`** component contains the code that identifies the reason for the signal. The signal numbers that may be returned are defined in the file <**`signal.h`**>; the signal codes are defined in the file <**`sys/siginfo.h`**>. If an application–defined value has been queued to the process with the signal, the **`si_value`** component contains that value; if not, the content of the **`si_value`** component is undefined.

If one of the signals specified by the *set* argument is generated or is pending delivery, **`sigwaitinfo`** returns the number of the selected signal in the structure to which *info* points. A return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`sigtimedwait(2)`** system manual page for a listing of the types of errors that may occur.

The sigqueue System Call

The **sigqueue(2)** system call allows the calling process to queue a signal and a value to itself or another process.

In order for multiple occurrences of a signal to be queued and an application–defined value to be passed with the signal, the receiving process must have asked for queueing by setting the SA_SIGINFO flag on a call to **sigaction(2)** to specify the signal action (for information on the SA_SIGINFO flag and the **sigaction(2)** system call, see “The sigaction Structure,” p. 10-9, and “The sigaction System Call,” p. 10-17, respectively).

The **sigqueue** system call differs from the **kill(2)** system call in three respects: it queues a unique instance of a **siginfo_t** structure to be delivered to the specified process with a signal; it allows a sending process to specify a *value* argument; and it does not allow a process to specify a negative value for the *pid* argument (which signifies a signal broadcast).

Unless the sending process is sending the SIGCONT signal to a process that is a member of the same session, the following conditions must be met in order to use the **sigqueue** system call:

- The real or effective user ID of the sending process must match the real or saved user ID of the receiving process unless the process sending the signal is privileged.
- If the Enhanced Security Utilities are installed, then the unprivileged sending process's security level must be equal to the receiving process's level.

The specifications required for making the **sigqueue** call are as follows:

```
#include <sys/siginfo.h>
#include <signal.h>

int sigqueue(pid, signo, value)

pid_t pid;
int signo;
const union sigval value;
```

The arguments are defined as follows:

pid the process identification number (PID) of the process to which the signal is to be sent. If the value of *pid* is the PID of the calling process and the signal specified by *signo* is not blocked, *signo* or at least one pending unblocked signal will be delivered to the calling process before the **sigqueue(2)** call returns.

signo the number of the signal that is to be sent to the process specified by *pid.* A set of symbolic constants has been defined to assist you in specifying signal numbers. These constants are defined in the file <**signal.h**>.

If the value of *signo* is zero (the null signal), **sigqueue** checks the validity of the specified PID but does not send a signal.

value an application–defined value that is to be used by a signal–handling routine defined by the receiving process. This value may be a pointer or an integer. It is available to the receiving process if that process has defined a signal–handling routine for the signal specified by *signo* and has set the SA_SIGINFO flag on a call to **sigaction(2)** to declare the handling routine.

This value is presented in the `si_value` component of the `siginfo_t` argument to the signal–handling routine. The signal code that is presented in the `si_code` component of this argument is **SI_QUEUE**. For an explanation of the procedures for using the **sigaction** system call, see "The sigaction System Call" (p. 10-17). For a description of the **siginfo_t** structure, see "The siginfo_t Structure" (p. 10-13). For an explanation of the procedures for defining the signal–handling routine, see "The Signal–Handling Routine" (p. 10-29).

A return value of **0** indicates that the specified signal has been successfully queued. A return value of **–1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **sigqueue(2)** system manual page for a listing of the types of errors that may occur.

System V Signal System Calls

The OS also supports the System V **sigsend(2)** and **sigaltstack(2)** system calls. The **sigsend** call allows a process to send a signal to another process or group of processes:

```
sigsend(idtype, id, signo);
  idtype_t idtype;
  id_t id;
  int signo;
```

Unless the process sending the signal is privileged, its real or effective user ID must be equal to the receiving process's real or saved user ID. If the Enhanced Security Utilities are installed, then the unprivileged sending process's security level must be dominated by the receiving process's level.

As explained in "Real-Time Signal Behavior" (p. 10-6), the **sigsend** system call does not send queued signals.

Signals can also be sent from a terminal device to the process group or session leader associated with the terminal (see **termio(7)**).

The **sigaltstack** system call allows a process to define an alternate stack area on which signals are to be processed. Applications that maintain complex or fixed-size stacks can use the call:

```
struct sigaltstack {
  caddr_t                    ss_sp;
  int                        ss_size;
  int                        ss_flags;
};
```

```
sigaltstack(ss, oss)
  struct sigaltstack *ss;
  struct sigaltstack *oss;
```

to provide the system with a stack based at `ss_sp` of size `ss_size` for delivery of signals. The system automatically adjusts for direction of stack growth. The member `ss_flags` indicates whether the process is currently on the signal stack and whether the signal stack is disabled. The **`sigaltstack`** structure is defined in <**`signal.h`**>.

When a signal is to be delivered and the process has requested that it be delivered on the alternate stack, the system checks whether the process is currently executing on that stack (see **`sigaction(2)`**). If it is not, then the process is switched to the alternate signal stack for delivery of the signal. The return from the signal is arranged to restore the previous stack.

If the process wishes to take a nonlocal exit from the signal-handling routine or run code from the signal stack that uses a different stack, a **`sigaltstack`** call should be used to reset the signal stack (see **`sigaltstack(2)`**).

The Signal–Handling Routine

If you wish to obtain information about the reason that a signal has been generated and the user context of the process that has received it, you must (1) define the signal–handling routine with a particular interface and (2) declare the routine as the handler for the signal by invoking the **`sigaction(2)`** system call and setting the SA_SIGINFO bit (see "The sigaction System Call," p. 10-17, for an explanation of this call). The interface that you must use for the signal–handling routine is presented as follows:

```
handler(sig, infop, ucp)
int sig;
siginfo_t *infop;
ucontext_t *ucp;
```

The arguments to the routine are defined as follows:

sig	the signal number
infop	the null pointer constant or a pointer to a structure that contains information about the signal specified by *sig*. Such information includes the signal number and the reason that the signal has been generated. For a detailed description of the contents of this structure, see "The siginfo_t Structure" (p. 10-13).
ucp	a pointer to a structure that defines the user context for the process prior to the delivery of the signal *sig*. The user context includes the process's signal mask, execution stack, and machine registers. It will be used to restore the process's context upon return from the signal handler. For a detailed description of the contents of this structure, see "The ucontext_t Structure" (p. 10-15).

You may modify certain components of the user context within a signal–handling routine, but you are advised to use caution in doing so. You may modify the general purpose registers, instruction address registers, signal mask, and stack flags, but you are not allowed to modify the address or size of the stack.

Job Control and Session Management

An overview of Job Control is provided here for completeness and because it interacts with the STREAMS-based terminal subsystem. This section describes how to use a Stream as a controlling terminal. More information on Job Control can be obtained from the following manual pages: **`exit(2)`**, **`getpgid(2)`**, **`getpgrp(2)`**, **`getsid(2)`**, **`kill(2)`**, **`setpgid(2)`**, **`setpgrp(2)`**, **`setsid(2)`**, **`sigaction(2)`**, **`signal(2)`**, **`sigsend(2)`**, **`termios(2)`**, **`waitid(2)`**, **`waitpid(3C)`**, **`signal(5)`**, **`and termio(7)`**.

Overview of Job Control

Job Control is a feature supported by the BSD UNIX operating system. It is also an optional part of the IEEE P1003.1 POSIX standard. Job Control breaks a login session into smaller units called jobs. Each job consists of one or more related and cooperating processes. One job, the foreground job, is given complete access to the controlling terminal. The other jobs, called background jobs, are denied read access to the controlling terminal and given conditional write and **`ioctl`** access to it. The user may stop an executing job and resume the stopped job either in the foreground or in the background.

Under Job Control, background jobs do not receive events generated by the terminal and are not informed with a hangup indication when the controlling process exits. Background jobs that linger after the login session has been dissolved are prevented from further access to the controlling terminal, and do not interfere with the creation of new login sessions.

The OS supports job-control and command interpreter processes supporting job-control can assign the terminal to different jobs, or process-groups, by placing related processes in a single process-group and assigning the process-group with the terminal. A process may examine or change the foreground process-group of a terminal assuming the process has the required permissions (see **`tcgetpgrp(2)`** and **`tcsetpgrp(2)`**). The **`termios`** facility aids in this assignment by restricting access to the terminal by processes outside of the foreground process-group (see “Terminal Access Control”).

When there is no longer any process whose process-id or process-group-id matches the process-group-id of the foreground process-group, the terminal lacks any foreground process-group. It is unspecified whether the terminal has a foreground process-group when there is no longer any process whose process-group-id matches the process-group-id of the foreground process-group, but there is a process whose process-id matches the process-group-id of the foreground process-group. Only a successful call to **`tcsetpgrp`** or assignment of the controlling terminal as described can make a process-group the foreground process-group of a terminal (see **`tcsetpgrp(2)`**).

Background process-groups in the session of the session-leader are subject to a job-control line-discipline when they attempt to access their controlling terminal. Typically, they are sent a signal that causes them to stop, unless they have made other arrangements (see

`signal(4)`). An exception is made for processes that belong to a orphaned process-group, which is a process-group none of whose members have a parent in another process-group within the same session and thus share the same controlling terminal. When these processes attempt to access their controlling terminal, they return errors because there is no process to continue them if they should stop (see “Terminal Access Control”).

Job Control Terminology

The following defines terms associated with Job Control:

- Background Process-group—a process-group that is a member of a session that established a connection with a controlling terminal and is not the foreground process-group.
- Controlling Process—a session leader that established a connection to a controlling terminal.
- Controlling Terminal—a terminal that is associated with a session. Each session may have at most one controlling terminal associated with it and a controlling terminal may be associated with at most one session. Certain input sequences from the controlling terminal cause signals to be sent to the process-groups in the session associated with the controlling terminal.
- Foreground Process Group—each session that establishes a connection with a controlling terminal distinguishes one process-group of the session as a foreground process-group. The foreground process-group has certain privileges that are denied to background process-groups when accessing its controlling terminal.
- Orphaned Process Group—a process-group in which the parent of every member in the group is either a member of the group, or is not a member of the process-group's session.
- Process Group—each process in the system is a member of a process-group that is identified by a process-group ID. Any process that is not a process-group leader may create a new process-group and become its leader. Any process that is not a process-group leader may join an existing process-group that shares the same session as the process. A newly created process joins the process-group of its creator.
- Process Group Leader —a process whose process ID is the same as its process group ID.
- Process Group Lifetime—a time period that begins when a process-group is created by its process-group leader and ends when the last process that is a member in the group leaves the group.
- Process ID—a positive integer that uniquely identifies each process in the system. A process ID may not be reused by the system until the process lifetime, process-group lifetime, and session lifetime ends for any process ID, process-group ID, and session ID sharing that value.
- Process Lifetime—a time period that begins when the process is forked and ends after the process exits, when its termination has been acknowledged by its parent process.

- Session—each process-group is a member of a session that is identified by a session ID.
- Session ID—a positive integer that uniquely identifies each session in the system. It is the same as the process ID of its session leader.
- Session Leader—a process whose session ID is the same as its process and process-group ID.
- Session Lifetime—a time period that begins when the session is created by its session leader and ends when the lifetime of the last process-group that is a member of the session ends.

Job Control Signals

The following signals manage Job Control (see also **`signal(5)`**)

`SIGCONT`	Sent to a stopped process to continue it.
`SIGSTOP`	Sent to a process to stop it. This signal cannot be caught or ignored.
`SIGTSTP`	Sent to a process to stop it. It is typically used when a user requests to stop the foreground process.
`SIGTTIN`	Sent to a background process to stop it when it attempts to read from the controlling terminal.
`SIGTTOU`	Sent to a background process to stop it when one attempts to write to or modify the controlling terminal.

The Controlling Terminal and Process-Groups

A session may be allocated a controlling terminal. For every allocated controlling terminal, Job Control elevates one process group in the controlling process's session to the status of foreground process group. The remaining process-groups in the controlling process's session are background process-groups. A controlling terminal gives a user the ability to control execution of jobs within the session. Controlling-terminals play a central role in Job Control. A user may cause the foreground job to stop by typing a predefined key on the controlling terminal. A user may inhibit access to the controlling terminal by background jobs. Background jobs that attempt to access a terminal that has been so restricted will be sent a signal that typically causes the job to stop. (See the section titled “Accessing the Controlling Terminal” section later in this chapter.)

Terminal Access Control

If a process is in the foreground process-group of its controlling terminal, **`read`** works as described in the “System Calls and Libraries” chapter. If any process in a background process-group attempts to read from its controlling terminal when job-control is supported, the signal `SIGTTIN` is sent to its process-group unless one of these special cases apply.

- If the reading process either ignores or blocks the signal SIGTTIN or if the reading process is a member of an orphaned process-group, attempting to read the controlling terminal fails without sending the signal SIGTTIN, the **read** returns -1 and errno equals EIO.

The default action of the signal SIGTTIN is to stop the process to which it is sent (see **signal(4)**).

If a process is in the foreground process-group of its controlling terminal, **write** works as described in "Writing Data and Output Processing." If any process in a background process-group attempts to write onto its controlling terminal when the flag TOSTOP is set in the c_lflag field of the **termios** structure, the signal SIGTTOU is sent to the process-group unless one of these special cases apply:

- If the writing process either ignores or blocks the signal SIGTTOU, attempting to write the controlling terminal proceeds without sending the signal SIGTTOU.
- If the writing process neither ignores nor blocks the signal SIGTTOU and if the writing process is a member of an orphaned process-group, attempting to write the controlling terminal fails without sending the signal SIGTTOU, the **write** returns -1 and errno equals EIO.

If the flag TOSTOP is clear, attempting to write the controlling terminal proceeds without sending the signal SIGTTOU.

Certain calls that set terminal parameters are treated the same as **write** calls, except that the flag TOSTOP is ignored; thus, the effect is the same as terminal **write** calls when the flag TOSTOP is set (see **tcgetattr(2)** and **tcsetattr(2)**).

If the implementation supports job-control, unless otherwise noted, processes in a background process-group are restricted in their use of the terminal-control-functions (see **tcdrain(2)**, **tcflow(2)**, **tcflush(2)**, **tcgetattr(2)**, **tcgetpgrp(2)**, **tcsendbreak(2)**, **tcsetattr(2)**, **tcsetsid(2)**, **tcsetpgrp(2)**). Attempts to perform these functions cause the process-group to be sent the signal SIGTTOU. If the calling process either ignores or blocks the signal SIGTTOU, attempting to perform a control-function proceeds without sending the signal SIGTTOU.

The default action of the signal SIGTTOU is to stop the process to which it is sent (see **signal(4)**).

All terminal-control-functions operate on an open file-descriptor and they affect the underlying terminal-device-file denoted by the file-descriptor, not the open-file-description that represents it.

If a member of a background process-group attempts to invoke an **ioctl** on its controlling terminal, and that **ioctl** modifies terminal parameters (e.g., **TIOCSPGRP**, **TCSETA**, **TCSETAW** or **TCSETAF**) its process-group is sent SIGTTOU, which normally causes the members of that process-group to stop.

- If the calling process either ignores or blocks the signal SIGTTOU, attempting to perform a terminal-control-function on the controlling terminal proceeds without sending the signal SIGTTOU.
- If the calling process neither ignores nor blocks the signal SIGTTOU and if the calling process is a member of an orphaned process-group, attempting

to perform a terminal-control-function on the controlling terminal fails without sending the signal `SIGTTOU`, the **`ioctl`** returns `-1` and **`errno`** equals `EIO`.

The terminal access controls described in this section apply only to a process accessing its controlling terminal because these controls are for the purpose of job-control, not security, and job-control relates only to a controlling terminal for a process. Normal file-access-permissions handle security. A process accessing a terminal other than the controlling terminal is effectively treated the same as a member of the foreground process-group.

If a process in a background orphaned process-group calls **`read`** or **`write`**, stopping the process-group is undesirable, as it is no longer under the control of a job-control shell that can put it into foreground again. Accordingly, calls to **`read`** and **`write`** by such processes receive an immediate return error.

The terminal-driver must repeatedly do a foreground/background/orphaned process-group check until either the process-group of the calling process is orphaned or the calling process moves into the foreground. If a calling process is in the background and should receive a job-control signal, the terminal-driver sends the appropriate signal (`SIGTTIN` or `SIGTTOU`) to every process in the process-group of the calling process then lets the calling process receive the signal immediately, usually by blocking the process so it reacts to the signal right away. Note, however, that after the process catches the signal and the terminal-driver regains control, the driver must repeat the foreground/background/orphaned process-group check. The process may still be in the background, either because a job-control shell continued the process in the background, or because the process caught the signal and did nothing.

The terminal-driver repeatedly does the foreground/background/orphaned process-group check whenever a process tries to access the terminal. For **`write`** or the line-control functions, the check is done on entering the function. For **`read`**, the check is done not only on entering the function but also after blocking the process to wait for input data (if necessary). If the process calling **`read`** is in the foreground, the terminal-driver tries to get data from the input-queue, and if the queue is empty, blocks the process to wait for data. When data are input and the terminal-driver regains control, it must repeat the foreground/background/orphaned process-group check again because the process may have moved to the background from the foreground while it blocked to wait for input data. (see the documentation on job control in the “Glossary”).

Modem Disconnect

The following arrangements are made to allow processes that read from a terminal-device-file and test for end-of-file to terminate appropriately when a modem-disconnect is detected on the terminal-device:

- All processes with that terminal as the controlling terminal receive a hang-up signal, `SIGHUP`, if `CLOCAL` is clear in the `c_cflags` for the terminal (see **`termios(4)`**). Unless other arrangements are made, the signal `SIGHUP` forces the processes to terminate (see **`signal(4)`** and **`sigaction(2)`**). If the signal `SIGHUP` is ignored or caught by a signal-catching function, any subsequent **`read`** returns `0` to indicate end-of-file until the terminal-device-file is closed (see **`read(2)`**).
- If the controlling process is not in the foreground process group of the terminal, the signal `SIGTSTP` is sent to all processes in the foreground pro-

cess group for which the terminal is the controlling terminal. Unless other arrangements are made, the signal SIGTSTP forces the processes to terminate (see **signal(4)** and **sigaction(2)**).

- Processes in background process groups that try a **read** or a **write** of the controlling terminal after a modem-disconnect while the terminal is still assigned to the session receive the appropriate signal, SIGTTIN or SIGTTOU respectively (see **read(2)** and **write(2)**). Unless other arrangements are made, the signal SIGTTIN or SIGTTOU forces the processes to terminate (see **signal(4)** and **sigaction(2)**).

STREAMS-based Job Control

Job Control requires support from a line discipline module on the controlling terminal's Stream. The TCSETA, TCSETAW, and TCSETAF commands of **termio(7)** allow a process to set the following line discipline values relevant to Job Control:

SUSP character	A user defined character that, when typed, causes the line discipline module to request that the Stream head sends a SIGTSTP signal to the foreground process with an M_PCSIG message, which by default stops the members of that group. If the value of SUSP is zero, the SIGTSTP signal is not sent, and the SUSP character is disabled.
TOSTOP flag	If TOSTOP is set, background processes are inhibited from writing to their controlling terminal.

A line discipline module must record the SUSP suspend character and notify the Stream head when the user has typed it, and record the state of the TOSTOP bit and notify the Stream head when the user has changed it.

Allocation and Deallocation

A Stream is allocated as a controlling terminal for a session if

- The Stream is acting as a terminal
- The Stream is not already allocated as a controlling terminal
- The Stream is opened by a session leader that does not have a controlling terminal.

Drivers and modules can inform the Stream head to act as a terminal Stream by sending an M_SETOPTS message with the SO_ISTTY flag set upstream. This state may be changed by sending an M_SETOPTS message with the SO_ISNTTY flag set upstream.

Controlling-terminals are allocated with the **open(2)** system call. A Stream head must be informed that it is acting as a terminal by an M_SETOPTS message sent upstream before or while the Stream is being opened by a potential controlling process. If the Stream head is opened before receiving this message, the Stream is not allocated as a controlling terminal.

Hung-up Streams

When a Stream head receives an `M_HANGUP` message, it is marked as hung-up. Streams that are marked as hung-up are allowed to be reopened by their session leader if they are allocated as a controlling terminal, and by any process if they are not allocated as a controlling terminal. This way, the hangup error can be cleared without forcing all file descriptors to be closed first.

If the reopen is successful, the hung-up condition is cleared.

Hangup Signals

When the `SIGHUP` signal is generated by an `M_HANGUP` message (instead of an `M_SIG` or `M_PCSIG` message), the signal is sent to the controlling process instead of the foreground process-group because the allocation and deallocation of controlling terminals to a session is the responsibility of that process-group.

Accessing the Controlling Terminal

If a process attempts to access its controlling terminal after it has been deallocated, access is denied. If the process is not holding or ignoring `SIGHUP`, it is sent a `SIGHUP` signal. Otherwise, the access fails with an `EIO` error.

Members of background process-groups have limited access to their controlling terminals:

- If the background process is ignoring or holding the `SIGTTIN` signal or is a member of an orphaned process-group, an attempt to read from the controlling terminal fails with an `EIO` error. Otherwise, the process is sent a `SIGTTIN` signal, which by default stops the process.
- If the process is attempting to write to the terminal and if the terminal's `TOSTOP` flag is clear, the process is allowed access.

  The `TOSTOP` flag is set on reception of an `M_SETOPTS` message with the `SO_TOSTOP` flag set in the `so_flags` field. It is cleared on reception of an `M_SETOPTS` message with the `SO_TONSTOP` flag set.
- If the terminal's `TOSTOP` flag is set and a background process is attempting to write to the terminal, the write succeeds if the process is ignoring or holding `SIGTTOU`. Otherwise, the process stops except when it is a member of an orphaned process-group, in which case, it is denied access to the terminal and it is returned an `EIO` error.
- If a background process is attempting to perform a destructive **`ioctl`** (an **`ioctl`** that modifies terminal parameters), the **`ioctl`** call succeeds if the process is ignoring or holding `SIGTTOU`. Otherwise, the process will stop except when the process is a member of the orphaned process-group. In that case, the access to the terminal is denied and an `EIO` error is returned.

Basic Interprocess Communication Pipes

The system call **`pipe`** creates a pipe, a type of unnamed FIFO (First In First Out) file used as an I/O channel between two cooperating processes: one process writes onto the pipe, while the other reads from it. Most pipes are created by the shell, as in:

```
ls | pr
```

which connects the standard output of **`ls`** to the standard input of **`pr`**. Sometimes, however, it is most convenient for a process to set up its own plumbing; this section illustrates how to establish and use the pipe connection.

Because a pipe is both for reading and writing, **`pipe`** returns two file-descriptors as follows:

```
int  fd[2];
stat = pipe(fd);
if (stat == -1)
    /* there was an error ... */
```

where `fd` is an array of two file-descriptors, with `fd[0]` for the read end of the pipe and `fd[1]` for the write end of the pipe. These may be used in **`read`**, **`write`** and **`close`** calls just like any other file-descriptors.

Implementation of pipes consists of implied **`lseek`** operations before each **`read`** or **`write`** in order to implement first-in-first-out. The system looks after buffering the data and synchronizing the two processes to prevent the writer from grossly out-producing the reader and to prevent the reader from overtaking the writer. If a process reads a pipe that is empty, it will wait until data arrive; if a process writes into a pipe that is full, it will wait until the pipe empties somewhat. If the write end of the pipe is closed, a subsequent **`read`** will encounter end-of-file.

To illustrate the use of pipes in a realistic setting, consider a function **`popen(`***cmd*,*mode*`)`, which creates a process *cmd*, and returns a file-descriptor that will either read or write that process, according to *mode*; thus, the call

```
fout = popen(“pr”, WRITE);
```

creates a process that executes the **`pr`** command; subsequent **`write`** calls using the file-descriptor **`fout`** send data to that process through the pipe.

```
#include <stdio.h>

#define   READ   0
#define   WRITE  1
#define   tst(a, b) (mode == READ ? (b) : (a))
static    int popen_pid;

popen(cmd, mode)
   char *cmd;
   int  mode;
{
   int p[2];

   if (pipe(p) < 0)
      return(NULL);

   if ((popen_pid = fork( )) == 0) {
      close(tst(p[WRITE], p[READ]));
      close(tst(0, 1));
      dup(tst(p[READ], p[WRITE]));
      close(tst(p[READ], p[WRITE]));
      execl("/bin/sh", "sh", "-c", cmd, 0);
      _exit(1) /* disaster occurred if we got here */
   }
   if (popen_pid == -1)
      return(NULL);

   close(tst(p[READ], p[WRITE]));
   return(tst(p[WRITE], p[READ]));
}
```

Screen 10-5. popen

As shown in Screen 10-5, the function **popen** first calls **pipe** to create a pipe, then calls **fork** to create two copies of itself. The child decides whether it is supposed to read or write, closes the other end of the pipe, then calls the shell (via **execl**) to run the desired process. The parent likewise closes the end of the pipe it does not use. These **close** operations are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never encounter the end-of-file on the pipe, just because there is one writer potentially active. The sequence of **close** operations in the child is a bit tricky. Suppose that the task is to create a child process that will read data from the parent. Then the first **close** closes the write end of the pipe, leaving the read end open.

To associate a pipe with the standard input of the child, use the following:

```
close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));
```

The **close** call closes file-descriptor `0`, the standard input, then the **dup** call returns a duplicate of the open file-descriptor. File-descriptors are assigned in increasing order and **dup** returns the first available one, so the **dup** call effectively copies the file-descriptor for the pipe (read end) to file-descriptor `0` making the read end of the pipe the standard input. (Although somewhat tricky, it's a standard idiom.) Finally, the old read end of the pipe is closed. A similar sequence of operations takes place when the child process must write to the parent process instead of reading from it. To finish the job we need a function **pclose** to close a pipe created by **popen**.

```
#include <signal.h>

pclose(fd)   /* close pipe descriptor */
   int fd;
{
   struct sigaction o_act, h_act, i_act, q_act;
   extern pid_t popen_pid;
   pid_t c_pid;
   int   c_stat;

   close(fd);

   sigaction(SIGINT, SIG_IGN, &i_act);
   sigaction(SIGQUIT, SIG_IGN, &q_act);
   sigaction(SIGHUP, SIG_IGN, &h_act);

   while ((c_pid=wait(&c_stat))!=-1 && c_pid!=popen_pid);
   if (c_pid == -1)
      c_stat = -1;

   sigaction(SIGINT, &i_act, &o_act);
   sigaction(SIGQUIT, &q_act, &o_act);
   sigaction(SIGHUP, &h_act, &o_act);

   return(c_stat);
}
```

Figure 10-1. Pclose

The main reason for using a separate function rather than **close** is that it is desirable to wait for the termination of the child process. First, the return value from **pclose** indicates whether the process succeeded. Equally important when a process creates several children is that only a bounded number of unwaited-for children can exist, even if some of them have terminated; performing the **wait** lays the child to rest. The calls to **sigaction** make sure that no interrupts, etc., interfere with the waiting process (see **sigaction(2)**).

The routine as written has the limitation that only one pipe may be open at once because of the single shared variable popen_pid; it really should be an array indexed by file-descriptor. A **popen** function, with slightly different arguments and return value is available as part of the Standard I/O Library (see **stdio(3S)**).

STREAMS-Based Pipes and FIFOs

A pipe in the UNIX system is a mechanism that provides a communication path between multiple processes. Before Release 4, the OS had “standard” pipes and named pipes (also called FIFOs). With standard pipes, one end was opened for reading and the other end for writing, thus data flow was unidirectional. FIFOs had only one end; typically, one process opened the file for reading and another process opened the file for writing. Data written into the FIFO by the writer could then be read by the reader.

To provide greater support and development flexibility for networked applications, pipes and FIFOs have become STREAMS-based in the OS. The basic interface remains the same but the underlying implementation has changed. Pipes now provide a bidirectional mechanism for process communication. When a pipe is created by the **pipe** system call, two Streams are opened and connected together, thus providing a full-duplex mechanism. Data flow is on a FIFO basis. Previously, pipes were associated with character devices and the creation of a pipe was limited to the capacity and configuration of the device. STREAMS-based pipes and FIFOs are not attached to STREAMS-based character devices, eliminating configuration constraints and the number of opened pipes to the number of file descriptors for that process.

NOTE

The remainder of this chapter uses the terms “pipe” and “STREAMS-based pipe” interchangeably.

Creating and Opening Pipes and FIFOs

FIFOs, which are created by **`mknod(2)`** or **`mkfifo(3C)`** behave like regular file system nodes but are distinguished from other file system nodes by the `p` in the first column when the **`ls -l`** command is executed. Data written to the FIFO or read from the FIFO flow up and down the Stream in STREAMS buffers. Data written by one process can be read by another process.

NOTE

If the Enhanced Security Utilities are installed, the Mandatory Access Control (MAC) security level of a fifonode is inherited from the level of the creating process. A privileged process can change the level using the **`lvlfile`** system call. See **`lvlfile(2)`** in the *Operating System API Reference* for details.

MAC write access is required for either a read to or a write from a FIFO. In other words, a process must be at the same level as the fifonode for either read or write. This is because a read modifies the FIFO. If you have installed the Enhanced Security Utilities, see the chapter “Directory and File Management” in this guide for a general discussion of security levels and Mandatory Access Controls.

FIFOs are opened in the same way as other file system nodes using the **`open`** system call. Any data written to the FIFO can be read from the same file descriptor in a FIFO manner. Modules can also be pushed on the FIFO. See **`open(2)`** for the restrictions that apply when opening a FIFO.

A STREAMS-based pipe is created by the **`pipe`** system call that returns two file descriptors, `fd[0]` and `fd[1]`. Both file descriptors are opened for reading and writing. Data written to `fd[0]` becomes data read from `fd[1]` and vice versa.

Each end of the pipe has knowledge of the other end through internal data structures. Subsequent reads, writes, and closes are aware of whether the other end of the pipe is open or closed. When one end of the pipe is closed, the internal data structures provide a way to access the Stream for the other end so that an `M_HANGUP` message can be sent to its Stream head.

NOTE

If the Enhanced Security Utilities are installed, the security level of a pipe is inherited from the level of the creating process and cannot be changed. MAC write access is required for either a read to or a write from a pipe; in other words, a process must be at the same level as the pipe.

After successful creation of a STREAMS-based pipe, `0` is returned. If **`pipe`** is unable to create and open a STREAMS-based pipe, it will fail with **`errno`** set as follows:

`ENFILE`	File table is overflowed.
`EMFILE`	Cannot allocate more file descriptors for the process.
`ENOSR`	Could not allocate resources for both Stream heads.
`EINTR`	Signal was caught while creating the Stream heads.

STREAMS modules can be added to a STREAMS-based pipe with the **`ioctl I_PUSH`**. A module can be pushed onto one or both ends of the pipe (see Figure 10-2). However, a pipe maintains the concept of a midpoint so that if a module is pushed onto one end of the pipe, that module cannot be popped from the other end.

Accessing Pipes and FIFOs

STREAMS-based pipes and FIFOs can be accessed through the operating system routines **`read(2)`**, **`write(2)`**, **`ioctl(2)`**, **`close(2)`**, **`putmsg(2)`**, **`getmsg(2)`**, and **`poll(2)`**. If FIFOs, **`open`** is also used.

Reading from a Pipe or FIFO

The **`read`** (or **`getmsg`**) system call is used to read from a pipe or FIFO. A user reads data from a Stream (not from a data buffer as was done prior to Release 4). Data can be read from either end of a pipe.

On success, the **`read`** returns the number of bytes read and placed in the buffer. When the end of the data is reached, the **`read`** returns `0`.

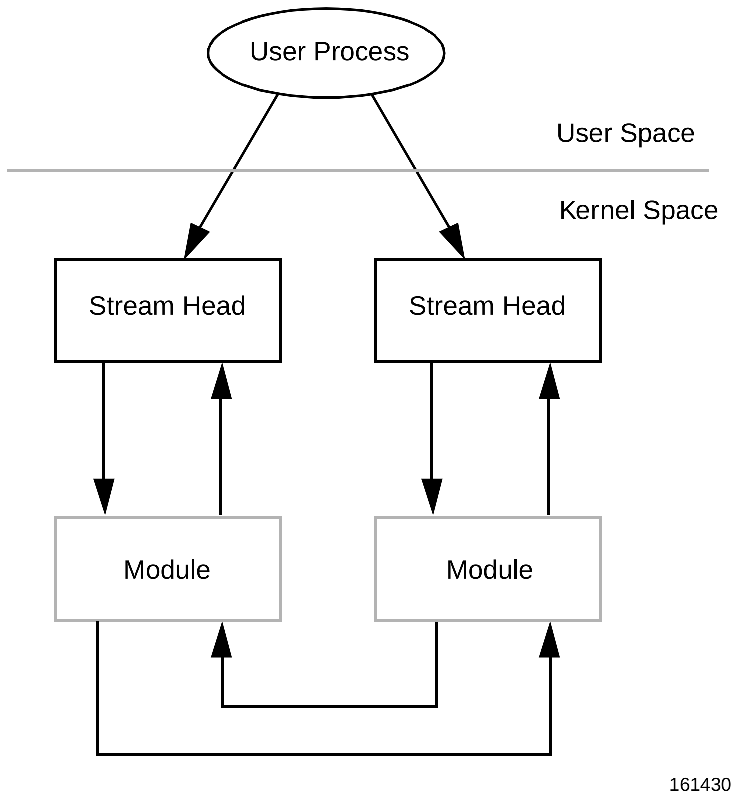


Figure 10-2. Pushing Modules on a STREAMS-based Pipe

When a user process attempts to read from an empty pipe (or FIFO), the following will happen:

- If one end of the pipe is closed, `0` is returned indicating the end of the file.
- If no process has the FIFO open for writing, **`read`** returns `0` to indicate the end of the file.
- If some process has the FIFO open for writing, or both ends of the pipe are open, and `O_NDELAY` is set, **`read`** returns `0`.
- If some process has the FIFO open for writing, or both ends of the pipe are open, and `O_NONBLOCK` is set, **`read`** returns `-1` and sets **`errno`** to `EAGAIN`.
- If `O_NDELAY` and `O_NONBLOCK` are not set, the **`read`** call blocks until data is written to the pipe, until one end of the pipe is closed, or the FIFO is no longer open for writing.

Writing to a Pipe or FIFO

When a user process calls the **`write`** system call, data is sent down the associated Stream. If the pipe or FIFO is empty (no modules pushed), data written is placed on the read queue of the other Stream for STREAMS-based pipes, and on the read queue of the same Stream for FIFOs. Because the size of a pipe is the number of unread data bytes, the written data is reflected in the size of the other end of the pipe.

Zero Length Writes

If a user process issues **`write`** with `0` as the number of bytes to send down a STREAMS-based pipe or FIFO, `0` is returned, and by default no message is sent down the Stream. However, if a user requires that a `0`-length message be sent downstream, an **`ioctl`** call may be used to change this default behavior. The flag `SNDZERO` supports this. If `SNDZERO` is set in the Stream head, **`write`** requests of l bytes generate a `0`-length message and send the message down the Stream. If `SNDZERO` is not set, no message is generated and `0` is returned to the user.

To toggle the `SNDZERO` bit, the **`ioctl`** **`I_SWROPT`** is used. If *arg* in the **`ioctl`** call is set to `SNDZERO` and the `SNDZERO` bit is off, the bit is turned on. If *arg* is set to `0` and the `SNDZERO` bit is on, the bit is turned off.

The **`ioctl`** **`I_GWROPT`** is used to return the current write settings.

Atomic Writes

If multiple processes simultaneously write to the same pipe, data from one process can be interleaved with data from another process, if modules are pushed on the pipe or the write is greater than `PIPE_BUF`. The sequence of data written is not necessarily the sequence of data read. To ensure that writes of less than `PIPE_BUF` bytes are not be interleaved with data written from other processes, any modules pushed on the pipe should have a maximum packet size of at least `PIPE_BUF`.

NOTE

`PIPE_BUF` is an implementation-specific constant that specifies the maximum number of bytes that are atomic in a write to a pipe. When writing to a pipe, write requests of `PIPE_BUF` or less bytes are not interleaved with data from other processes doing writes on the same pipe. However, write requests greater than `PIPE_BUF` bytes may have data interleaved on arbitrary byte boundaries with writes by other processes whether the `O_NONBLOCK` or `O_NDELAY` flag is set.

If the module packet size is at least the size of `PIPE_BUF`, the Stream head packages the data in such a way that the first message is at least `PIPE_BUF` bytes. The remaining data may be packaged into smaller or larger blocks depending on buffer availability. If the first module on the Stream cannot support a packet of `PIPE_BUF`, atomic writes on the pipe cannot be guaranteed.

Closing a Pipe or FIFO

The **`close`** system call closes a pipe or FIFO and dismantles its associated Streams. On the last close of one end of a pipe, an `M_HANGUP` message is sent upstream to the other end of the pipe. Later **`read`** or **`getmsg`** calls on that Stream head return the number of bytes read and `0` when there is no more data. Later **`write`** or **`putmsg`** requests will fail with **`errno`** set to `EIO`. If the pipe has been mounted using **`fattach`**, the pipe must be unmounted before calling **`close`**; otherwise, the Stream will not be dismantled. If the other end of the pipe is mounted, the last close of the pipe will force it to be unmounted.

Flushing Pipes and FIFOs

When the flush request is initiated from a user **`ioctl`** or from a **`flushq`** routine, the `FLUSHR` and/or `FLUSHW` bits of an `M_FLUSH` message have to be switched. The point of switching the bits is the point where the `M_FLUSH` message is passed from a write queue to a read queue. This point is also known as the midpoint of the pipe.

The midpoint of a pipe is not always easily detectable, especially if there are numerous modules pushed on either end of the pipe. In that case, there needs to be a mechanism to intercept all messages passing through the Stream. If the message is an `M_FLUSH` message and it is at the Streams midpoint, the flush bits need to switched.

This bit switching is handled by the `pipemod` module. `pipemod` should be pushed onto a pipe or FIFO where flushing of any kind takes place. The `pipemod` module can be pushed on either end of the pipe. The only requirement is that it is pushed onto an end that previously did not have modules on it. That is, `pipemod` must be the first module pushed onto a pipe so that it is at the midpoint of the pipe itself.

The `pipemod` module handles only `M_FLUSH` messages. All other messages are passed on to the next module by the **`putnext`** utility routine. If an `M_FLUSH` message is passed to o `pipemod` and the `FLUSHR` and `FLUSHW` bits are set, the message is not processed but is passed to the next module by the **`putnext`** routine. If only the `FLUSHR` bit is set, the `FLUSHR` bit is turned off and the `FLUSHW` bit is set. The message is then passed to the next module by **`putnext`**. Similarly, if the `FLUSHW` bit is the only bit set in the `M_FLUSH` message, the `FLUSHW` bit is turned off and the `FLUSHR` bit is turned on. The message is then passed to the next module on the Stream.

The `pipemod` module can be pushed on any Stream that desires the bit switching. It must be pushed onto a pipe or FIFO if any form of flushing must take place.

Named Streams

Some applications may want to associate a Stream or STREAMS-based pipe with an existing node in the file system name space. For example, a server process may create a pipe, name one end of the pipe, and allow unrelated processes to communicate with it over that named end.

fattach

A STREAMS file descriptor can be named by attaching that file descriptor to a node in the file system name space. The routine **`fattach`** (see also **`fattach(3C)`**) is used to name a STREAMS file descriptor. **`fattach(3C)`**. Its format is

```
    int fattach (int fildes, char *fildes)
```

where *fildes* is an open file descriptor that refers to either a STREAMS-based pipe or a STREAMS device driver (or a pseudo device driver), and *path* is an existing node in the file system name space (for example, regular file, directory, character special file, and so forth).

The *path* cannot have a Stream already attached to it. It cannot be a mount point for a file system nor the root of a file system. A user must be an owner of the *path* with write permission or a user with the appropriate privileges to attach the file descriptor.

If the *path* is in use when the routine **`fattach`** is executed, those processes accessing the *path* are not interrupted and any data associated with the *path* before the call to the **`fattach`** routine will continue to be accessible by those processes.

NOTE

> If the Enhanced Security Utilities are installed, **`fattach(3C),`** and Mandatory Access Control the process calling `fattach` must be at the same MAC security level as *path*. This restriction can be overridden by processes with appropriate privileges. The device or pipe associated with the STREAM pointed to by *fildes* must also be at the same security level as *path*; there is no privilege to override this restriction.

After a Stream is named, all subsequent operations (for example, **`open(2)`**) on the *path* operate on the named Stream. Thus, it is possible that a user process has one file descriptor pointing to the data originally associated with the *path* and another file descriptor pointing to a named Stream.

Once the Stream has been named, the **`stat`** system call on *path* shows information for the Stream. If the named Stream is a pipe, the **`stat(2)`** information shows that *path* is a pipe. If the Stream is a device driver or a pseudo-device driver, *path* appears as a device. The initial modes, permissions, and ownership of the named Stream are taken from the attributes of the *path*. The user can issue the system calls **`chmod`** and **`chown`** to alter the attributes of the named Stream and not affect the original attributes of the *path*, nor the original attributes of the STREAMS file.

The size represented in the **`stat`** information reflects the number of unread bytes of data currently at the Stream head. This size is not necessarily the number of bytes written to the Stream.

A STREAMS-based file descriptor can be attached to many different *path*s at the same time (that is, a Stream can have many names attached to it). The modes, ownership, and permissions of these *path*s may vary, but operations on any of these *path*s access the same Stream.

Named Streams can have modules pushed on them, be polled, be passed as file descriptors, and be used for any other STREAMS operation.

fdetach

A named Stream can be disassociated from a file with the **`fdetach`** routine (see also **`fdetach(3C)`**), which has the following format:

```
int fdetach (char *path)
```

where *path* is the name of the previously named Stream. Only the owner of *path* or the user with the appropriate privileges may disassociate the Stream from its name. The Stream may be disassociated from its name while processes are accessing it. If these processes have the named Stream open at the time of the **`fdetach`** call, the processes do not get an error, and continue to access the Stream. However, after the disassociation, later operations on *path* access the underlying file rather than the named Stream.

If only one end of the pipe is named, the last close of the other end causes the named end to be automatically detached. If the named Stream is a device and not a pipe, the last close does not cause the Stream to be detached.

If there is no named Stream or the user does not have access permissions on *path* or on the named Stream, **`fdetach`** returns `-1` with `errno` set to `EINVAL`. Otherwise, **`fdetach`** returns `0` for success.

A Stream remains attached with or without an active server process. If a server aborted, the only way a named Stream is cleaned up is if the server executed a clean up routine that explicitly detached and closed down the Stream.

If the named Stream is that of a pipe with only one end attached, clean up occurs automatically. The named end of the pipe is forced to be detached when the other end closes down. If there are no other references after the pipe is detached, the Stream is deallocated and cleaned up. Thus, a forced detach of a pipe end occurs when the server is aborted.

If both ends of the pipe are named, the pipe remains attached even after all processes have exited. In order for the pipe to become detached, a server process has to explicitly invoke a program that executes the **`fdetach`** routine.

To eliminate the need for the server process to invoke the program, the **`fdetach(1M)`** command can be used. This command accepts a pathname that is a path to a named Stream. When the command is invoked, the Stream is detached from the path. If the name is the only reference to the Stream, the Stream is also deallocated.

A user invoking the **`fdetach(1M)`** command must be an owner of the named Stream or a user with the appropriate permissions.

isastream

The function **`isastream`** (see also **`isastream(3C)`**) may be used to determine if a file descriptor is associated with a STREAMS device. Its format is

```
    int isastream (int fildes)
```

where *fildes* refers to an open file. **`isastream`** returns `1` if *fildes* represents a STREAMS file, and `0` if not. On failure, **`isastream`** returns `-1` with **`errno`** set to `EBADF`.

This function is useful for client processes communicating with a server process over a named Stream to check whether the file has been overlaid by a Stream before sending any data over the file.

File Descriptor Passing

Named Streams are useful for passing file descriptors between unrelated processes. A user process can send a file descriptor to another process by invoking the **`ioctl`** **`I_SENDFD`** on one end of a named Stream. This sends a message containing a file pointer to the

Stream head at the other end of the pipe. Another process can retrieve that message containing the file pointer by invoking the **ioctl I_RECVFD** on the other end of the pipe.

Unique Connections

With named pipes, client processes may communicate with a server process by using a module called **connld** that enables a client process to gain a unique, nonmultiplexed connection to a server. The **connld** module can be pushed onto the named end of the pipe. If **connld** is pushed on the named end of the pipe and that end is opened by a client, a new pipe is created. One file descriptor for the new pipe is passed back to a client (named Stream) as the file descriptor from the **open** call and the other file descriptor is passed to the server. The server and the client may now communicate through a new pipe.

Figure 10-3 illustrates a server process that has created a pipe and pushed the **connld** module on the other end. The server then invokes the **fattach** routine to name the other end **/usr/toserv**.

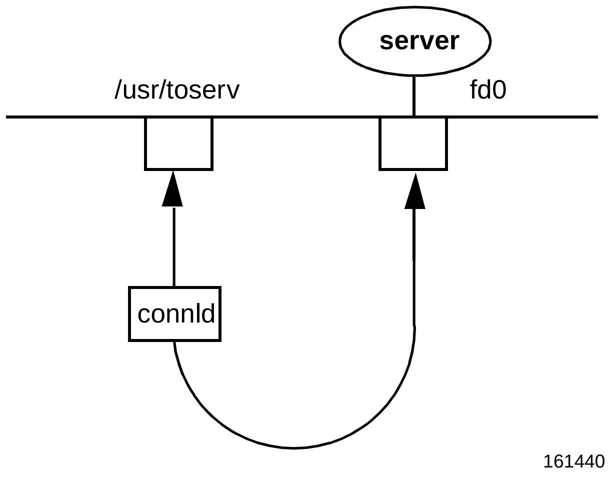


Figure 10-3. Server Sets Up a Pipe

When process X (**procx**) opens **/usr/toserv**, it gains a unique connection to the server process that was at one end of the original STREAMS-based pipe. When process Y (**procy**) does the same, it also gains a unique connection to the server. Figure 10-4 shows that the server process has access to three separate STREAMS-based pipes using three file descriptors.

connld is a STREAMS-based module that has an **open**, **close**, and **put** procedure. **connld** is opened when the module is pushed onto the pipe for the first time and whenever the named end of the pipe is opened. The **connld** module distinguishes between these two opens with the `q_ptr` field of its read queue. On the first **open**, this field is set to `1` and the routine returns without further processing. On later **opens**, the field is checked for `1` or `0`. If the `1` is present, the **connld** module creates a pipe and sends the file descriptor to a client and a server. When the named Stream is opened, the open routine of **connld** is called. The **connld** open fails if

- The pipe ends cannot be created.
- A file pointer and file descriptor cannot be allocated.
- The Stream head cannot stream the two pipe ends.
- A failure occurs while sending the file descriptor to the server.

The open is not complete until the server process receives the file descriptor using the **ioctl I_RECVFD**.

NOTE

If the Enhanced Security Utilities are installed, a MAC check is performed to make sure the server has MAC access to the received file descriptor before the server receives the file descriptor. If this fails, the file descriptor is put back on the queue so that another process that passes MAC checks can receive it.

The setting of the O_NDELAY or O_NONBLOCK flag has no affect on the open.

The **connld** module does not process messages. All messages are passed to the next object in the Stream. The read and write **put** routines call **putnext** to send the message up or down the Stream.

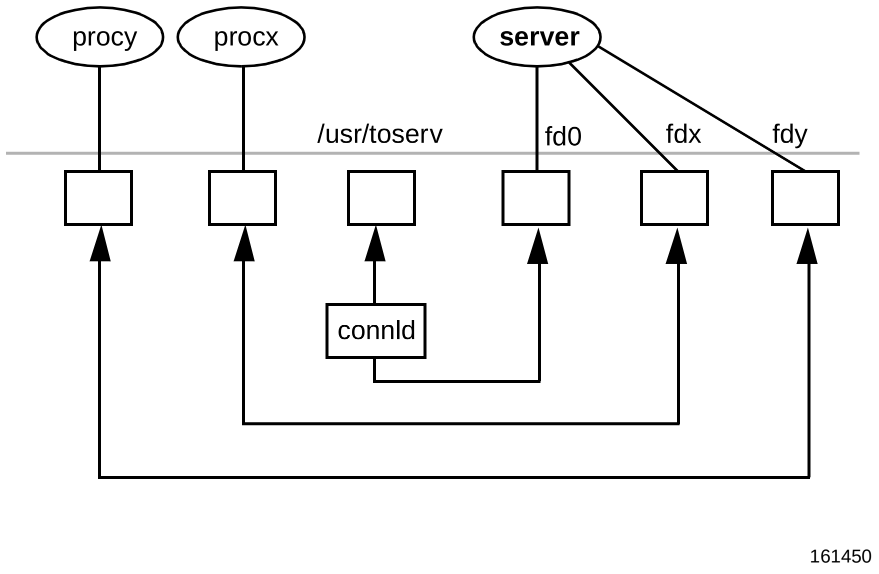


Figure 10-4. Processes X and Y Open /usr/toserv

11
Programming with the Threads Library

11
Programming with the Threads Library

Introduction

This chapter introduces the Threads Library, which provides facilities for concurrent programming. Before describing the routines included in the Threads Library, this chapter first discusses concepts and terminology of concurrent programming in general and of the Threads Library in particular.

The Threads Library provides two classes of routines: thread management routines and synchronization routines. The thread management routines are discussed in "Basic Threads Management." These include routines to create threads, terminate threads, wait for threads, and adjust threads' scheduling characteristics. In addition, this section discusses how signals interact with multithreaded programs, how threads are scheduled, and the relationship between threads and lightweight processes. The synchronization routines are discussed in "Synchronizing Threads." This includes an overview of the various types of locks, semaphores, barriers, and condition variables, used to synchronize threads that are sharing data.

In addition to the PowerMAX OS Threads Library routines, the Threads Library also includes the POSIX threads function calls. These POSIX threads function calls conform to the IEEE POSIX 1003.1 1996 Edition. Customers that want to write multi-threaded applications that are POSIX-compliant are highly encouraged to use these POSIX threads routines. Mixing POSIX threads function calls with non-POSIX threads function calls is highly discouraged. Generally speaking, the POSIX threads function call names begin with the 'pthread_' prefix, while the non-POSIX (hereafter referred to as the PowerMAX OS Threads implementation) threads function call names begin with the 'thr_' prefix.

Since most of the POSIX threads function calls are very similar in functionality and interface to the PowerMAX OS threads routines, discussion of the POSIX threads function calls within this chapter are grouped together with the discussions of their PowerMAX OS threads function call counter-parts.

The section entitled, "Development Environment," discusses the compilation environment. Finally, "Examples" gets you started with some basic threads programs.

This chapter is not intended to replicate all the information covered in the system manual pages for the threads library routines. Refer to the individual pages on line for details such as error returns. The overview pages, **`thread(3thread)`**, **`pthread(3pthread)`**, and **`synch(3synch)`**, list all the available routines.

What Is Concurrent Programming?

Historically, most programs are examples of sequential programming. That is, they consist of a series of operations that are carried out one at a time. With *concurrent* programming the programmer can specify sets of instructions that potentially can be executed in parallel and still provide correct results.

The advantages of this style of programming are:

- A powerful programming paradigm.

  Programs are often written to emulate or respond to events in the real world. In the real world, concurrency is common and purely sequential events are the exception. Modeling such behavior is facilitated if the programming environment supports the notion of concurrency.

- Possible performance improvement.

  If multiple processors are available, the program might be executed in less real time (than sequential execution) if more than one processor is working simultaneously. This is called *true concurrency.*

  Even on uniprocessor machines, there may be some performance gain from designing greater concurrency into the program. While one activity is blocked, others might still be executing.

  Thus, there is an advantage to concurrent programming even if the resources (processors) are not available to provide true concurrency and the application is only *logically* concurrent.

Concurrent programming has been available in the UNIX system since its inception via the process model. In the UNIX system problems are solved not just by running programs but by running sets of programs (a running program is called a process) — sometimes pre-existing tools or commands; sometimes specifically written programs — that work together (often concurrently) to solve the problem. Processes can communicate and synchronize with each other by mechanisms that include:

- pipes (named and unnamed)
- files and file/record locks
- signals
- messages
- shared memory
- semaphores

What Are Threads?

The PowerMAX OS provides vastly expanded capabilities for concurrent programming via the Threads Library. These capabilities include:

- Facilities to define multiple threads of control to be run concurrently within a single process. Each *thread* is a set of instructions that is itself sequential but can be executed concurrently with other threads.
- A new, rich set of software mechanisms for coordinating and synchronizing the activities of the process's threads.

  PowerMAX OS thread functions include:

  - mutual exclusion locks (mutexes), both recursive and not, both blocking and spinning.
  - reader-writer locks
  - counting semaphores (<u>not</u> the IPC semaphore system calls)
  - condition variables
  - barriers

  POSIX thread functions include:

  - blocking, non-recursive mutual exclusion locks (mutexes),
  - condition variables
- Features to control the level of concurrency and the scheduling of threads.
- Underlying operating system kernel support that enables the library to provide true concurrency (on multiprocessor architectures), not just logical concurrency for threads.

General characteristics of threads programming:

- Each thread starts executing at a programmer-specified address of a function.
  - A given, common function can be the starting point for several unique threads.
- A thread has many features that are analogous to process features. For example,
  - Each thread is an individually schedulable entity.
  - Threads can be preempted; consequently, a thread cannot assume uninterrupted access to common data unless special synchronizing arrangements (for example, locking) are made.
  - Threads execute logically in parallel, exhibit logical concurrency and possibly true concurrency.
  - A thread will go through many states during its lifetime such as:

    - executing
    - ready to run but not currently executing
    - waiting for some resource
    - terminated thread with unreported exit status
    - stopped from running
- Threads can receive signals; consequently, asynchronous programming is still possible. (See“Threads and Signals,” page 11-19.)
- In this implementation, most of the features of the Threads Library are implemented by user-level library code that is dynamically linked with the application program at run time. The underlying operating system kernel is not aware of the threads of a process.
- The operating system kernel supports a scheduling abstraction called the *lightweight process* (LWP). An LWP is not the same as a thread. It is a facility that is used by the Threads Library to provide true concurrency for threads. (See “Managing Threads Concurrency,” page 11-32.)

Each thread of the process has access to all of the resources of the process including:

- The entire address space.

  Any thread can access any memory location in the process’s address space. By using threads for concurrency, the programmer sacrifices the address space protection that the operating system maintains (with support of hardware features) between processes—for example, one thread might use an incorrect data pointer to write data at a location that would corrupt another thread’s data.

  On the other hand, thread-to-thread data sharing is easy and efficient. By default, all data is available to all threads. Thread-to-thread communication avoids the system call overhead and typical data copying of process-to-process communications.

- Resources maintained by the operating system, including:
    - Open files, file pointer offsets, file/record locks, and current directory.
    - Access rights (to files, IPC facilities, and so on) and Enhanced Security privileges.
    - Resource limits such as `ulimit`, `umask`, and file descriptor limit.
- Process identity (such as process ID number, parent process ID, process group number)

The features unique to each thread include:

- Program context (that is, register values)
- Stack

- Scheduling information (such as scheduling class, current priority)
- Timers

  The interval timers the **`getitimer(3c)`** and **`setitimer(3c)`** functions provide are unique to each thread. All threads in the process share the POSIX per-process timers provided by the **`timer_gettime(3c)`** and **`timer_settime(3c)`** functions.
- Signal handling

  Actually, some signal handling features are maintained per thread and some are maintained at the process level. The relationship between the two (and how to use them) will be discussed later. (See “Threads and Signals,” page 11-19.)
- Thread ID number and thread-private data.

On the whole, there is a much more intimate relationship between the threads of a process than between processes of an application. This gives the programmer much greater flexibility and potentially better performance.

- With this intimacy, there is a greater potential for introducing subtle errors, and that implies a greater demand on the programmer's skill to produce correct code.
- Moreover, the proper design of a concurrent program requires certain disciplines that do not often arise in sequential programming. For example, inappropriate use of the Threads Library facilities for synchronizing threads may result in a program that is incorrect, inefficient, or both.

The Threads Library functions are best-suited for medium- to coarse-grained concurrency. Using the Threads Library can be inefficient when the scale of concurrent tasks is a different order of magnitude than the scale of a single function:

- Much larger, such as an entire program. Use **`fork(2)`**/**`exec(2)`** instead.
- Much smaller, when the Threads Library entails too much overhead.

Threads Illustrated

Figure 11-1 illustrates the relation of threads to LWPs to processes to processors. The terms “multiplexed” threads and “bound” threads will be discussed later in this chapter.

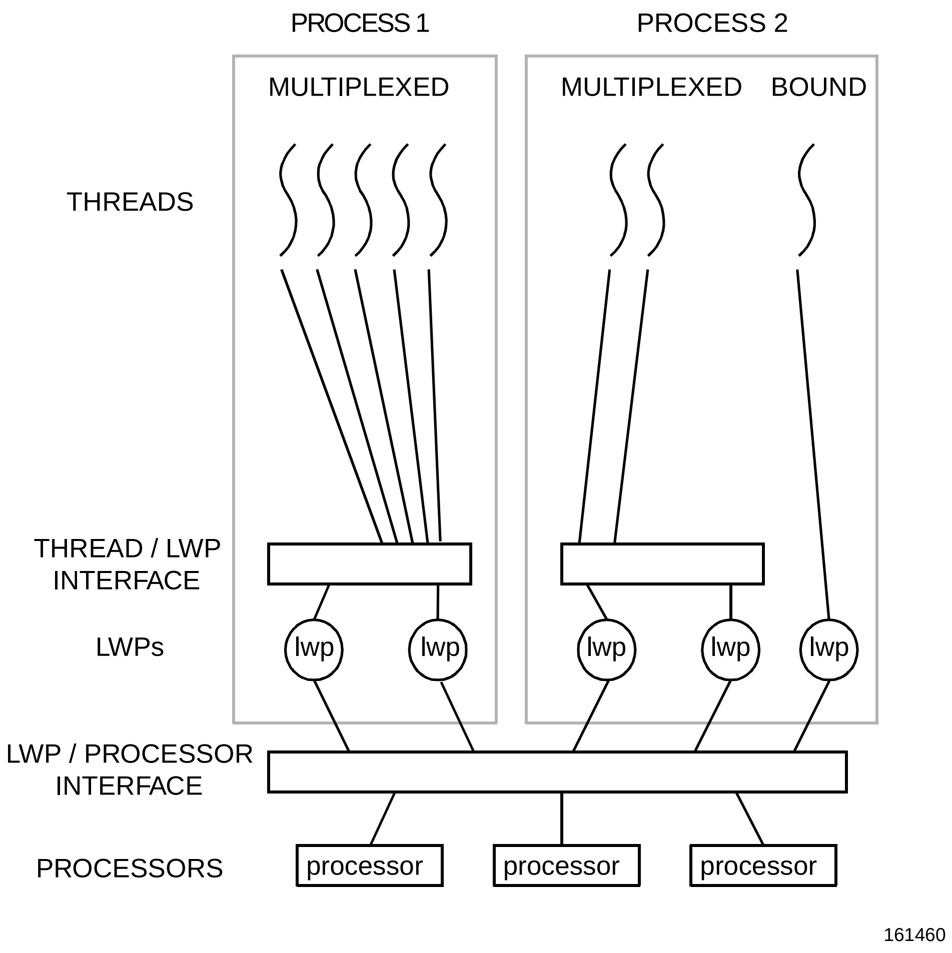


Figure 11-1. Overview of Threads

Basic Threads Management

The basic operations on threads are *conceptually* similar to certain operations on processes.

Operation	Process Method	Thread Method

Creation	**fork(2)/exec(2)**	**thr_create(3thread)**
		pthread_create(3pthread)

Termination	**exit(2)**	**thr_exit(3thread)**
		pthread_exit(3pthread)

Synchronization	**wait(2)**	**thr_join(3thread)**
		pthread_join(3pthread)

Creating a New Thread

Processes that are linked with the thread library contain just one user thread within the process at the point when the process begins execution of a new program (on entry to **`main()`** following a successful call to **`exec(2)`**).

The thread that executes the **`main()`** routine is known as the initial, or primordial, thread. By default, this thread is created internally by the thread library as a multiplexing thread. Therefore, all the characteristics that are associated with other multiplexing threads also apply to this initial thread.

Creating a PowerMAX OS Thread

New threads can be created via the **`thr_create(3thread)`** routine

```
int thr_create(
    void     *stack_address,
    size_t   stack_size,
    void     *(*start_routine)(void *arg),
    void     *arg,
    long     flags,
    thread_t *new_thread
);
```

which takes the following parameters:

stack_address and *stack_size*

These define the stack space for the new thread. (This space is used for function call transactions and for automatic variables in functions called by the thread.)

The stack of the traditional UNIX process has *autogrow* support by the operating system. That is, if the stack grows beyond its initial size the operating system automatically increases its size as needed (or until it runs into some other defined segment). However, threads (other than the initial thread) use stacks that do not have autogrow support; consequently, the stacks should be allocated to meet the maximum needs of the thread.

As a convenience, the Threads Library will implicitly allocate a reasonably-sized stack if *stack_address* and *stack_size* are set to `NULL` and `0`, respectively,

- The programmer can specify other sizes if needed. The value must not be less than that returned by **`thr_minstack(3thread)`**. Note that *stack_address* should point to the base address (lowest) of the allocated space.
- In this implementation, the Threads Library manages the process address space so that stack overflows will result in an addressing error (`SIGSEGV`). For most applications, this is a desirable behavior. It is better to discover such errors as soon as they occur rather than have one stack corrupt another.

start_routine and *arg*

These parameters define the starting condition of the newly created thread. *start_routine* is the function address where the new thread's execution will begin and *arg* is the argument that *start_routine* will receive.

start_routine takes a single parameter of type `(void *)` and returns a value of the same type. These values can be used (with type casts) to pass values (or aggregations of values in structures) of any type.

NOTE

For portability, do not cast an `int` to `(void *)`, and then cast it back to `int`. These values should only be used as pointers; otherwise, information can be lost.

Of course, a thread need not be entirely defined by a single function. That initial function will typically call other functions (hence the thread's need for a separate stack).

flags

These flags will be discussed as their respective topics arise later in this chapter. These flags are not mutually exclusive; they can be combined with a bitwise inclusive OR. For each flag, the relevant section is shown:

Flag	Section
`THR_SUSPENDED`	Managing Thread Scheduling
`THR_BOUND`	Managing Concurrency Level
`THR_DETACHED`	Waiting for Thread Termination
`THR_INCR_CONC`	Managing Concurrency Level
`THR_DAEMON`	Terminating a Thread

new_thread

The *thread ID* of the newly created thread is delivered to the creator thread at this address.

- This value can be used in other functions to influence that thread.
- The scope of the value is limited to the enclosing process; it is not relevant to threads in other processes.
- A thread can learn its own thread ID number by the **`thr_self(3thread)`** function.

NOTE

`thr_create(3thread)` and the other functions in the Threads Library return `0` on success. On failure, instead of setting the `errno` global variable, they *return* the error code as the function's value.

Creating a POSIX Thread

POSIX thread creation is done via the **`pthread_create(3pthread)`** routine:

```
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine)(void *),
    void *arg
);
```

which takes the following parameters:

thread
: The thread ID of the newly created thread is returned to the creator thread at this address.

attr
: Pointer to a thread attributes structure. When this structure is not NULL, then the attributes in this structure are used when creating the new thread. When NULL, the default values are used to create the new thread.

start_routine and *arg*
: The new thread is created executing *start_routine* with *arg* as its sole argument. If the *start_routine* returns, the effect shall be as if there was an implicit call to **`pthread_exit(3pthread)`** using the return value of *start_routine* as the exit status.

POSIX Thread Creation Attributes

The **`pthread_create(3pthread)`** function allows the caller to pass a thread attributes structure, which may be used to specify certain attributes for the thread about to be created. Once a thread attributes structure has been properly setup, it may be used on multiple **`pthread_create(3pthread)`** function calls, with or without modifications between each **`pthread_create(3pthread)`** call.

The following POSIX thread function call provides an application with the ability to initialize a threads attribute structure to default values. Additional POSIX thread function calls provide the ability to modify these default values.

```
int pthread_attr_init(pthread_attr_t *attr);
```

This function call initializes the caller's thread attributes object, pointed to by *attr.* After a **`pthread_attr_init(3pthread)`** function call, the specified thread attributes object will be initialized to the following default attribute values. Note that these default values provide the same results as specifying a NULL *attr* thread attributes pointer would on a **`pthread_create(3pthread)`** call:

```
        stack address = NULL
        stack size = 0
```

The created thread will have a stack allocated by the threads library, with a default stack size.

```
    detach state = PTHREAD_CREATE_JOINABLE
```

The created thread will be in the joinable state. Such a thread is eligible to participate in a **`pthread_join(3pthread)`** operation.

```
    contention scope = PTHREAD_SCOPE_SYSTEM
```

The created thread will have `PTHREAD_SCOPE_SYSTEM` scheduling contention scope; that is, the thread will be bound to a new lightweight process (LWP) and will be scheduled exclusively to this LWP.

```
    inherit scheduling = PTHREAD_INHERIT_SCHED
```

The created thread will inherit its scheduling policy and associated attributes from the creating thread. The scheduling attributes in the thread attributes object will be ignored.

```
    scheduling policy = SCHED_OTHER
```

`SCHED_OTHER` is the default scheduling policy for newly created threads. It is defined as a time-sharing policy.

```
    scheduling parameter = DEFAULTMUXPRI
```

This is the scheduling priority parameter. The default value is defined in `<thread.h>`, and applies to multiplexed threads only. With the default settings, it is ignored at thread creation time. For multiplexed threads, priority is inherited from the creator thread. For bound threads, priority is inherited from the creator thread's LWP.

It should be noted that several PowerMAX OS thread creation attributes do not have equivalent counterparts in the POSIX thread interface. The attributes that are not available through **`pthread_create(3pthread)`** via the threads attribute structure are:

- Suspending a thread upon creation (`THR_SUSPENDED`)
- Increasing the level of concurrency (`THR_INCR_CONC`)
- Creating a daemon thread (`THR_DAEMON`)

Modifying POSIX Thread Creation Attributes

The following POSIX thread functions may be used to modify or obtain the current value of an attribute within a threads attribute structure:

`int` **`pthread_attr_destroy`**`(pthread_attr_t *`*attr*`);`

This function call will set the specified thread attributes structure to a destroyed state[PTHREAD_DESTROYED]. After this call, the thread attributes may no longer be used on **`pthread_create(3pthread)`** calls, without an intervening **`pthread_attr_init(3pthread)`**call to re-initialize the structure.

```
int pthread_attr_getstackaddr(const pthread_attr_t
    *attr, void **stackaddr);
int pthread_attr_getstacksize(const pthread_attr_t *attr,
    size_t *stacksize);
```

These two routines will return the stack address and stack size attributes that are in the specified thread attributes structure.

```
int pthread_attr_setstackaddr(pthread_attr_t *attr,
    void *stackaddr);
int pthread_attr_setstacksize(pthread_attr_t *attr,
    size_t stacksize);
```

These two routines will set the current stack address and stack size attributes, within the specified thread attributes structure.

As is the case for **thr_create(3thread)**, a default value of NULL for the stack address and a zero value for the stack size will cause the Thread Library to implicitly create a reasonable-sized stack for the new thread.

```
int pthread_attr_getdetachstate(const pthread_attr_t
    *attr, int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr,
    int detachstate);
```

These two function calls are used to get and set the detached state threads attribute. Threads created with a detached state of PTHREAD_CREATE_JOINABLE are joinable via **pthread_join(3pthread)**, while PTHREAD_CREATE_DETACHED threads may not be joined.

```
int pthread_attr_getscope(pthread_attr_t *attr,
    int *contentionscope);
int pthread_attr_setscope(pthread_attr_t *attr,
    int contentionscope);
```

These two function calls are used to get and set the contention scope threads attribute. A thread created with a PTHREAD_SCOPE_SYSTEM contention scope will be created as a bound thread, while a thread created with a PTHREAD_SCOPE_PROCESS contention scope will be created as a multiplexing thread.

```
int pthread_attr_getinheritsched(const pthread_attr_t
    *attr, int *inheritsched);
int pthread_attr_setinheritsched(pthread_attr_t *attr,
    int inheritsched);
```

These two function calls are used to get and set the inherit scheduling attribute in the threads attribute structure. When **inheritsched** is PTHREAD_INHERIT_SCHED, the created thread inherits its scheduling policy and scheduling parameters from the creating thread. When **inheritsched** is PTHREAD_EXPLICIT_SCHED, the thread is created with the scheduling policy and scheduling parameters contained in the thread attributes structure.

```
int pthread_attr_getschedpolicy(const pthread_attr_t
    *attr, int *policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr,
    int policy);
```

These two function calls may be used to get and set the scheduling policy attribute in the thread attributes structure. The policy may be `SCHED_OTHER`, `SCHED_FIFO` (first-in-first-out) or `SCHED_RR` (round robin). Only bound (`PTHREAD_SCOPE_SYSTEM`) threads may be created with the `SCHED_FIFO` or `SCHED_RR` scheduling policies.

```
int pthread_attr_getschedparam(const pthread_attr_t *attr,
    struct sched_param *param);
int pthread_attr_setschedparam(pthread_attr_t *attr,
    const struct sched_param *param);
```

These two function calls are used to get and set the scheduling parameters (currently only the priority) of the threads attribute structure. This attribute is ignored unless the inherit scheduling attribute of the threads attribute structure is set to `PTHREAD_EXPLICIT_SCHED`.

If a bound thread (`PTHREAD_SCOPE_SYSTEM`) with explicit scheduling (`PTHREAD_EXPLICIT_SCHED`) is being created, then **`pthread_attr_setschedparam(3pthread)`** should always be called to set the priority of the new thread, since the default priority value in the thread attributes structure only pertains to multiplexing **(**`PTHREAD_SCOPE_PROCESS`**)** threads.

Creating a Thread From a Thread

The creation of one thread by another is conceptually similar but not identical to the creation of a new process by another process via the **`fork(2)`** system call. Some differences are:

- After a **`fork(2)`** system call both the creator (parent) and created (child) processes resume from the same point of computation — the return from **`fork(2)`**.

  In contrast, a new thread starts execution at the *start_function* specified by the creator (in some respects similar to the **`exec(2)`** system call), while the creating thread returns from **`thr_create(3thread)`** or **`pthread_create(3pthread)`**.

- The operating system maintains a parent/child relationship between creating and created processes that affects later interactions at process termination (for example, **`wait(2)`** semantics). In contrast, there is no innate hierarchy among threads. Each is a *sibling* of the other. Thus, the creator might wait for the newly created thread to terminate or, just as easily, the new thread can wait for its creator to terminate. (See discussion of **`thr_join(3thread)`** and **`pthread_join(3pthread)`** below.)

Terminating a Thread

PowerMAX OS Thread Termination

A thread can terminate itself by using the **`thr_exit(3thread)`** function:

void **thr_exit**(void **status*);

where:

status	is a pointer to the exit value of the terminating thread. The *status* will be returned to one of any sibling threads that call **`thr_join(3thread)`**.

The call to **`thr_exit`** initiates automatic clean-up for thread resources:

- Recovery of stack allocated by the Threads Library (see above).

NOTE

The stack for a non-detached thread will not be recovered until after another thread calls **`thr_join`** to obtain the *status* for the thread, which is stored on the stack. Likewise, an explicitly-allocated stack should not be recovered until after another thread calls **`thr_join`**.

- Invocation of the *destructor* function for each *key* value that the thread has used. (See “Thread-Specific Data,” page 11-17.)

The Threads Library arranges for a simple return from the *start_routine* to be equivalent to a call to **`thr_exit(3thread)`** (except for the initial thread, see “Termination of the Process,” page 11-14).

The **`thr_exit(3thread)`** function allows one thread to return a value called *status* to another; however, this mechanism is more general than the exit status returned by a child process to its parent. The argument to **`exit(2)`** is limited to a small range of integers. The *status* returned by **`thr_exit(3thread)`** is a general pointer that can be used (with type casts) to direct the receiver to objects of greater complexity such as structures, arrays, and linked lists. Of course, both the terminating and receiving threads should be coded to employ the same convention.

POSIX Thread Terminations

A POSIX thread can terminate itself by using the **`pthread_exit(3pthread)`** function:

void **pthread_exit**(void **status*);

where:

status is a pointer to the exit value of the terminating thread. The status will be returned to one of any sibling threads that call **pthread_join(3pthread)**.

The call to **pthread_exit** initiates automatic clean-up for thread resources:

- Recovery of stack allocated by the Threads Library.

NOTE

The stack for a non-detached (PTHREAD_CREATE_JOINABLE) thread will not be recovered until after another thread calls **pthread_join(3pthread)** to obtain the status for the thread. Likewise, an explicitly allocated stack should not be recovered until after another thread calls **pthread_join**.

- Invocation of the destructor function for each key value that the thread has used. (See "Thread-Specific Data" on page 11-17.)

A simple return from the *start_routine* is equivalent to a call to **pthread_exit(3pthread)**, with the status value being set to the value returned by the *start_routine*.

As was the case for **thr_exit(3thread)**, the status value is not limited to be a small range of integers, but may be used as a general pointer to objects such as structures or arrays.

Termination of the Process

PowerMAX OS Process Termination

The termination of the last *non-daemon* thread of the process will terminate the process (**thr_exit** passes the *status* to **exit(2)** when terminating the process).

- The Threads Library categorizes a thread as either a *daemon* thread or a non-daemon thread. In practice, daemon threads are used to provide services for other threads. Although they can terminate themselves or be terminated, there is no need to do so. By being distinguished as daemons, they will be implicitly terminated when there are no other threads (non-daemons) that might need their services.

  A thread is categorized as a daemon thread at the time of its creation by use of the THR_DAEMON flag to **thr_create(3thread)**.

- There are some special semantics for the initial thread. If the initial thread executes a **return** statement or if it implicitly returns from **main**, the process will be terminated. However, a **thr_exit(3thread)** by the initial thread will terminate only the initial thread. The process continues to execute as long as there are other non-daemon threads.

- Finally, any thread can terminate the process by calling the **exit(2)** system call.

POSIX Process Termination

The termination of the last thread in the process terminates the process (**pthread_exit(3pthread)** passes a zero status value to **exit(2)** when terminating the process.)

- There are some special semantics for the initial (primordial) thread. If the initial thread executes a return statement or if it implicitly returns from **main()**, the process will be terminated with the return value from **main()** being used as the value that is passed to **exit(2)**. However, a **pthread_exit(3pthread)** call by the initial thread will only terminate the initial thread. The process continues to execute as long as there are other threads in the process.
- Any thread can terminate the process by calling the **exit(2)** system service directly.

Waiting for Thread Termination

PowerMAX OS Thread Joining

One thread can suspend itself to wait for the termination of another thread with the **thr_join(3thread)** function

```
int thr_join(
    thread_t wait_for,
    thread_t *departed,
    void     **status
);
```

where the parameters have the following meaning:

wait_for	The ID of the thread of interest, that is, the thread whose termination the caller will await. A `(thread_t)0` indicates interest in the next thread to terminate (or one that has already terminated, but has not been joined), whatever its ID happens to be.
departed	**thr_join(3thread)** will deposit the thread ID of the terminated thread at this address.
status	**thr_join(3thread)** will deposit at this address the value given as an argument by the terminated thread when it called **thr_exit(3thread)**. That value should be the address at which the terminated thread left its return value (exit status).

POSIX Thread Joining

One thread can suspend itself to wait for the termination of another thread with the **pthread_join(3thread)** function:

```
int pthread_join(
    pthread_t thread,
    void **value_ptr
);
```

where the parameters have the following meaning:

thread	The ID of the thread of interest. This parameter must contain a valid thread ID; it may not be NULL.
status	**pthread_join(3pthread)** will store, at this location, the value passed on the **pthread_exit(3pthread)** call that was made by the terminated thread.

Note that unlike **thr_join(3thread)**, the **pthread_join(3pthread)** interface only provides the ability to wait for one specific thread to terminate.

If the thread of interest has already terminated, **thr_join(3thread)** or **pthread_join(3pthread)** will return immediately; otherwise, the calling thread will block.

If there is more than one thread waiting for the termination of some particular thread:

- The thread of interest will be joined to only one of the waiting threads. The choice is not predictable.
- All other waiting threads will return with the ESRCH error code.

If a thread receives a catchable signal while blocked in either **thr_join(3thread)** or **pthread_join(3pthread)**:

- The signal is handled.
- The **thr_join(3thread)** function is transparently restarted.

NOTE

This is analogous to the autorestart option for blocking system calls. (See description of the SA_RESTART flag to the **sigaction(2)** system call.)

The resources of the terminated thread (for example, a stack allocated by the Threads Library) will not be fully recovered by the Threads Library until some other thread has called **thr_join(3thread)** or **pthread_join(3pthread)** and received the terminated thread's exit status.

NOTE

Beware of lingering zombies!

Detached Threads

If the programmer knows at thread creation time that no other thread will attempt to join with the new thread via **`thr_join(3thread)`** or **`pthread_join(3pthread)`**, then the new thread should be created as a detached thread. When a detached thread terminates, its resources may be recovered immediately. In fact, it is not valid to use **`thr_join(3thread)`** or **`pthread_join(3pthread)`** on a detached thread.

By default, new threads are not created as detached threads.

To create a detached thread, set the `THR_DETACHED` flag on a **`thr_create(3thread)`** call, or set the `PTHREAD_CREATE_DETACHED` attribute in the threads attribute structure that is passed on a subsequent **`pthread_create(3pthread)`** call.

In addition to making a thread detached at thread creation time, POSIX threads may also be dynamically detached after they have already been created.The **`pthread_detach(3pthread)`** function call may be used for this purpose.

There are several reasons why a method for dynamically detaching a thread may be useful:

1. In order to detach the **`initial/main()`** thread.
2. **`pthread_join(3pthread)`** call may be interrupted due to a thread cancellation (see the section on POSIX thread cancellations). This canceled thread could have a **`pthread_detach(3pthread)`** call in its cancellation routine which would detach the target thread, since the thread being terminated would no longer be able to join with the other target thread.

NOTE

If a thread is blocked in **`pthread_join(3thread)`** waiting for the specified target thread to exit, and that target thread dynamically becomes detached, then **`pthread_join()`** will unblock the calling thread and return an error of ESRCH.

Thread-Specific Data

Historically, programs have used the `static` or `extern` storage classes to save data that must be preserved between function calls. This practice is no longer valid when many threads in the same process may run a given function concurrently and reference one `static` or `extern` variable by name. Values will not be preserved across function calls if one thread modifies a value left by another.

NOTE

In contrast, since each thread gets a unique stack, variables of the `auto` storage class are implicitly unique.

The facility for thread-specific data provides a solution to this problem.

- Data can be stored and retrieved by a key value.
- The same key value can be used to store data by many threads.
- The key is a virtual variable name that will resolve to the correct data for the calling thread when using the following access functions:
  - PowerMAX OS:

    `thr_setspecific(3thread)`
    `thr_getspecific(3thread)`
  - POSIX:

    `pthread_setspecific(3pthread)`
    `pthread_getspecific(3pthread)`

NOTE

Analogously, the file name **`/dev/tty`** can be used by any process to access its particular controlling terminal.

- The data is specific to each thread, but as with any other part of the process address space, the data is not protected from access or change by other threads.

PowerMAX OS Thread-Specific Data Functions

The access functions have the following syntax:

```
int thr_setspecific(thread_key_t key, void *data);
void *thr_getspecific(thread_key_t key);
```

A key value must be created by the **`thr_keycreate(3thread)`** function

```
int thr_keycreate(
    thread_key_t *key,
    void         (*destructor)(void *data)
);
```

where:

key	specifies the address where the newly created key value will be deposited
destructor	specifies a function that will be called on the exit of any thread that has used the key for data storage. This function should recover any space that has been used to store thread-specific data. When called, this function receives one argument, the *data* address that the thread gave as the second argument to **`thr_setspecific(3thread)`**.

POSIX Thread-Specific Data Functions

The POSIX thread-specific data access functions have the following syntax:

```
int pthread_setspecific(pthread_key_t key,
    const void *value);
void *pthread_getspecific(pthread_key_t key);
```

These functions may be used to set a thread-specific *value* with a given *key*, or to get the thread-specific value that is associated with the given key. Initially, the corresponding *value* for a given thread and key is NULL until that thread sets *value* to a non-NULL value.

A key value must first be created with the **pthread_create(3pthread)** function:

```
int pthread_key_create(pthread_key_t *key,
    void (*destructor)(void *));
```

where:

key specifies the location to which the calling thread returns the newly created key value.

destructor specifies a function that will be called during **pthread_exit(3pthread)** processing for any thread that has a non-NULL value associated with *key*. The intent is that this *destructor* function should recover any space allocated for storing thread-specific data. When called, this function receives one argument, the *value* parameter that was passed on a previous **pthread_setspecific(3pthread)** call.

NOTE

The key can be created (or later removed) by threads other than those that use the key for data storage. The using threads need only have access to the key value by function argument, global variable, or other means.

If a particular key value is needed for only a particular phase of a program (perhaps initialization), it can be deallocated by **thr_keydelete(3thread)** or **pthread_keydelete(3pthread)**.

For efficiency, it is best to minimize the number of keys used in an application.

Threads and Signals

When a process receives a signal of some type (for example, `SIGINT` type) the process can either take the default response, ignore the signal (the kernel does not actually deliver the signal), or catch the signal. When the signal is caught, the system will call a handler function when the signal is delivered. This response is called the disposition for the signal type. In the PowerMAX OS, that disposition is common to all of the threads of a process.

If the disposition for a signal type is:

termination	such signals will terminate all threads, and the process will terminate.
ignore	such signals will be ignored by all threads.
catch	any thread responding to such signals will enter the same handler function.

Moreover, if any thread changes the disposition (by calling **`sigaction(2)`**, for example), the new disposition is in effect for all threads.

NOTE

System signal types `SIGLWP` and `SIGWAITING` are used internally by the Threads Library. The Threads Library prevents modification of the disposition or masking of those signal types.

On the other hand, *signal masks* (the set of signal types being blocked) are maintained per thread.

PowerMAX OS Thread Signal Masks

A thread inherits the signal mask of its creating thread. A thread can alter its mask with the **`thr_sigsetmask(3thread)`** routine.

```
int thr_sigsetmask(
    int          how,
    const sigset_t *set,
          sigset_t *oset
);
```

where:

set	Defines a set of signal types.
how	Specifies how *set* will be used. *how* can be one of the following:

`SIG_SETMASK`	Discard the old mask; make *set* the new mask
`SIG_BLOCK`	Add the types in *set* to the existing mask
`SIG_UNBLOCK`	Remove the types in *set* from the existing mask

oset	Can be used to save the prior value of the thread's signal mask.

POSIX Thread Signal Masks

Like PowerMAX OS threads, POSIX threads also inherit the signal mask of their creating thread. A thread may alter its signal mask with the **`pthread_sigmask(3pthread)`** routine:

```
int pthread_sigmask(
        int how,
        const sigset_t *set,
        sigset_t *oset
    );
```

where the parameters have the exact same functionality as the **thr_sigsetmask(3thread)** routine.

NOTE

> The syntax of **thr_sigsetmask(3thread)** and **pthread_sigmask(3pthread)** is nearly identical to that of the **sigprocmask(2)** system call.

Signals can be categorized as being asynchronously generated or synchronously generated. A synchronously-generated signal is one that arises from the action of a particular thread or process. For example, alarm signals, signals resulting from an illegal memory reference, and signals resulting from an illegal arithmetic operation are all synchronously-generated signals. An asynchronously-generated signal is one that is sent from outside the thread (or process); its delivery is unpredictable. Interruptions and termination signals are usually asynchronously generated.

Asynchronously-Generated Signals

When a signal is delivered to a process, if it is being caught, it will be handled by one, and only one, of the threads meeting either of the following conditions:

- A thread blocked in a **sigwait(2)** system call whose argument does include the type of the caught signal.
- A thread whose signal mask does not include the type of the caught signal.

Additional considerations:

- A thread blocked in **sigwait(2)** is given preference over a thread not blocking the signal type.
- If more than one thread meets these requirements (perhaps two threads are calling **sigwait(2)**), then one of them will be chosen by the Threads Library. This choice is not predictable by application programs.
- If no thread is eligible, the signal will remain pending at the process level until some thread becomes eligible.

Asynchronously-Generated Signals — Paradigm

One useful paradigm for managing signals originating outside of the process is to have all threads include the caught signals in their signal mask and specifically create one daemon thread to handle the signals. If that thread uses the **sigwait(2)** system call, the signals can be handled in a synchronous style.

```
thr_sigsetmask(mask);
while( (signo = sigwait(mask)) > 0){
      handle signal type signo
}
```

Note that it is not only valid to wait for masked signals with **`sigwait`**, but it is important to mask out the signal types of interest before calling **`sigwait`**. Otherwise, the arrival of one such signal between calls to **`sigwait`** will be handled according to the current process disposition. By default, that will terminate the entire process. **`sigwait`** effectively unmasks any masked signals while blocked, then masks them again before returning.

Even if a handler function is specified, it will not be executed if a signal is delivered to a thread blocked in **`sigwait`**; **`sigwait`** bypasses any handler.

Since all threads are masking out the same set of signals, one can predict that the signals in that set will be handled by the single thread using **`sigwait`**. This paradigm is advantageous because:

- It reduces the complexity of the program.
- Only one thread need allocate stack space for signal handling. If there are several eligible threads, each must have sufficient stack for the handler.

NOTE

Alternate signal handling stacks (see **`sigaltstack(2)`**) are not supported by the Threads Library.

- Signals are handled in a synchronous style, which is usually easier to write and understand than an asynchronous style.

NOTE

The thread that handles the signals should be a <u>bound</u> thread. Bound threads are introduced in a later section, under “Threads Concurrency Level.”

Synchronously-Generated Signals

A caught, non-masked signal that is caused by a particular thread will be handled by that thread. Examples include:

- Signals arising from an invalid memory reference or illegal arithmetic operation. This allows the offending thread to correct its error.
- Alarm or timer signals requested by the thread.

  The Threads Library arranges for such signals to always be delivered to the requesting thread even if that (multiplexed) thread is no longer held by the same LWP as at the time of the request.

NOTE

Multiplexed threads are formally introduced in the section titled "Managing Threads Concurrency."

Each thread will use the common handler function.

Thread-to-Thread Signaling

PowerMAX OS Thread Signaling

One thread can signal another thread with the **thr_kill(3thread)** function:

```
int thr_kill(thread_t tid,int signo);
```

where:

tid	The thread ID of the target thread.
signo	The type of signal to send.

POSIX Thread Signaling

One POSIX thread can signal another thread with the **pthread_kill(3pthread)** function:

```
int pthread_kill(
    pthread_t tid,
    int sig
};
```

where:

tid	The POSIX thread ID of the target thread
sig	The signal number of the signal to be delivered to the target thread.

A thread catching a signal cannot distinguish between a signal originating from another thread of the process or from outside of the process.

The process disposition for the sent signal type (*signo*) is also applied for thread-to-thread signaling. As usual, the response will be to ignore the signal, to call the handler function, or to take the default response (usually, process termination).

This facility allows one thread to influence (perhaps "reset" or terminate) another thread asynchronously.

POSIX Thread Cancellations

Programs traditionally have used the signal mechanism combined with either **`longjmp()`** or polling to cancel/abort operations. Programmers may often have trouble using these facilities to solve their problems efficiently in a single-threaded process. With the introduction of threads, these solutions have become even more difficult to use.

Therefore, a POSIX thread cancellation mechanism has been provided for allowing a thread to terminate the execution of any other thread within the process in a controlled manner. The target thread (the one being canceled) is allowed to hold cancellation requests pending and to specify application-specific cleanup processing routines which will be executed when and if a thread cancellation occurs.

Each thread maintains its own cancelability state. The cancellation state has two attributes:

`State` This attribute may be either enabled or disabled. When the state is disabled, then cancellation requests are held pending. The **`pthread_setcancelstate(3pthread)`** function may be used to set the cancellation state to either *PTHREAD_CANCEL_ENABLE* (the default) or *PTHREAD_CANCEL_DISABLE*.

`Type` This attribute may be either deferred, or asynchronous. The **`pthread_setcanceltype(3pthread)`** function may be used to set the cancellation type to either *PTHREAD_CANCEL_DEFERRED* (the default) or *PTHREAD_CANCEL_ASYNCHRONOUS.*

When a thread's cancellation type is asynchronous, then the thread may be canceled at any time. Use of asynchronous cancelability while holding resources (or calling a library routine that acquires internal locks, etc.) may result in resource loss or indeterminate state of data structures, such as a mutex lock left in a locked state.

It should be mentioned that three functions are async-cancel safe: **`pthread_cancel(3pthread)`**, **`pthread_setcancelstate(3pthread)`** and **`pthread_setcanceltype(3pthread)`**. Asynchronous cancellations will not be processed while a thread is executing within these functions. However, asynchronous cancellation processing can and will be handled immediately upon return from these functions, when appropriate.

When a thread's cancellation type is deferred, then cancellation requests are held pending until a cancellation point is reached. Cancellation points occur within a specific set of functions, which are listed below:

`aio_suspend()`	**`pause()`**	**`sigwait()`**
`close()`	**`pthread_cond_timedwait()`**	**`sigwaitinfo()`**
`creat()`	**`pthread_cond_wait()`**	**`sleep()`**
`fsync()`	**`pthread_join()`**	**`system()`**
`mq_receive()`	**`pthread_testcancel()`**	**`tcdrain()`**
`mq_send()`	**`read()`**	**`wait()`**
`msync()`	**`sem_wait()`**	**`waitpid()`**

`nanosleep()`	**`sigsuspend()`**	**`write()`**
`open()`	**`sigtimedwait()`**	**`fcntl()`**

The cancellation point occurs in **`fcntl(2)`** only when the cmd argument is `F_SETLKW`.

Cancellation Point Function Considerations

The main intent of defining cancellation points within certain function calls is to allow a thread to be canceled while it is blocked indefinitely. To that end, when the calling thread's cancellation state and type are enabled and deferred, then these routines will honor any pending (or new) cancellation request if the thread is about to block indefinitely, or if the thread is already in an indefinitely blocked state. If the thread does not reach a point within the function where it is to be indefinitely blocked, then the routine will not necessarily check for and/or honor any pending cancellation request (except for **`pthread_testcancel(3pthread)`**, which will always honor any pending cancellation request).

Note that some of the above functions are invoked from within other system library functions. In these situations, the cancellation point processing in the above functions will still occur. For example, a commonly used function that internally calls a cancellation point routine is **`printf(3S)`**, which internally calls **`write(2)`**.

NOTE

Due to the fact that the C library will use an internal lock to serialize access to **`printf()`** operations, is it recommended that a calling thread set its cancellation state to *PTHREAD_CANCEL_DISABLE* if the thread may be sent a cancellation request while it is within the **`printf()`** routine.

A side effect of acting upon a cancellation request while in a condition variable wait, such as **`pthread_cond_wait()`**, is that the associated mutex is reacquired before the thread cancellation processing is initiated. Therefore, applications where cancellation requests are issued to threads that make condition wait function calls with cancellations enabled should usually use **`pthread_cleanup_push()`** (see the next section) to specify a cleanup handler that will unlock the associated mutex lock.

Cancellation Cleanup Handlers

When a thread is canceled, the cancellation code will call all cancellation handlers that are currently defined (pushed) for that thread. These cancellation handlers are maintained on a per thread basis. Note that the thread library will disable cancellations for the thread before calling its cancellation handlers.

The cleanup handler push and pop function interfaces are shown below:

```
void pthread_cleanup_push(void (*routine)(void *), void *arg);

void pthread_cleanup_pop(int execute);
```

The **`pthread_cleanup_push(3pthread)`** function pushes the specified thread-specific cancellation cleanup handler 'routine' onto the cancellation cleanup stack of the calling thread, while the **`pthread_cleanup_pop(3pthread)`** function removes the 'routine' at the top of the cancellation cleanup stack of the calling thread and optionally invokes it if the 'execute' argument is nonzero.

When a cancellation request is acted upon, the currently pushed cancellation routines are invoked one by one in LIFO (last-in-first-out) order. The thread invokes the cancellation handler with cancellation disabled until the last cancellation cleanup handler returns. If the last cancellation cleanup handler returns, thread execution is terminated and a status of `PTHREAD_CANCELED` is made available to any threads joining with the canceled thread via **`pthread_join(3pthread)`**.

Note that any currently pushed cancellation handlers are also invoked when a thread voluntarily calls **`pthread_exit(3pthread)`**.

These functions MUST appear as statement pairs within the same lexical scope; that is, **`pthread_cleanup_push()`** may be thought to expand to a token list whose first token is "{" with **`pthread_cleanup_pop()`** expanding to a token list whose last token is the corresponding "}". Failure to properly pair these two functions together within the same lexical scope will result in compilation errors.

Issuing a Cancellation Request

The **`pthread_cancel(3pthread)`** function may be used to issue a cancellation request to another thread within the same process:

```
int pthread_cancel(pthread_t thread);
```

where the 'thread' argument specifies the target thread to be canceled.

As previously mentioned, the cancellation state and type of the target thread determines when the cancellation takes effect. Note that the cancellation processing in the target thread occurs asynchronously with respect to the calling thread returning from **`pthread_cancel(3pthread)`**.

Note that it is also valid to specify the caller's thread id on the **`pthread_cancel(3pthread)`** call; this will cause a cancellation request to be issued for the calling thread.

Testing for Cancellation Requests

When a thread's cancellation type is deferred, and its cancellation state is enabled, then the thread may call **`pthread_testcancel(3pthread)`** to check for a cancellation request. In this case, if a cancellation request is pending for the calling thread, then cancellation processing for the calling thread will begin immediately, and the calling thread will not return from this function.

This function call is provided as a way to allow a thread to be canceled in a controlled fashion, even when that thread is not blocking inside of one of the other cancellation point routines that were previously listed.

If a cancellation request for the calling thread is not pending, or if the calling thread's cancellation state is currently disabled, then this function will have no effect.

Cancellation Cleanup Handler Example

The following example shows how one thread is canceled by another thread while it is blocked inside a **`pthread_cond_wait()`** call. The canceled thread pushes a cancellation handler which unlocks the mutex that is associated with the condition variable.

Most of the error checking that would normally be done on function calls has been left out in order to make the code easier to read.

```
#include <pthread.h>

/* Set when target thread has called pthread_cond_wait() */
int is_waiting;

/* Set when target thread has called its cancellation routine */
int was_canceled;

/* routine definitions */
void new_thread(void *);
void cond_cleanup(void *);

/* for blocking and synchronizations. */
pthread_cond_t pcond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t pmutex = PTHREAD_MUTEX_INITIALIZER;

main()
{
    int status;
    pthread_attr_t attr;
    pthread_t pthr_id;
    void *exit_status;


    /* Setup attributes for thread creation, and create the thread
    * that will be canceled.  (A bound thread will be created
    * in this example.)
    */
    (void) pthread_attr_init(&attr);
    (void) pthread_create(&pthr_id, &attr,
        (void *(*) (void *))new_thread, (void *)NULL);

    /* Wait for other thread to get blocked in pthread_cond_wait().
    */
    while (1) {
        (void) pthread_mutex_lock(&pmutex);
        if (is_waiting) {
            (void) pthread_mutex_unlock(&pmutex);
            break;
        }
            (void) pthread_mutex_unlock(&pmutex);
            while (sleep(1)) ;
    }

    /* Send a cancellation to the target thread, and
    * wait for it to be canceled.
    */
    (void) pthread_cancel(pthr_id);
    while (!was_canceled)
            sleep(1);

    /* Should be able to join with terminated thread.
    * The exit status value should be PTHREAD_CANCELED.
    */
    (void) pthread_join(pthr_id, &exit_status);
    if (exit_status != PTHREAD_CANCELED) {
        printf("ERROR: invalid exit status 0x%x\n",
                exit_status);
        exit(1);
    }

    /* Check that the mutex is unlocked.
     */
    status = pthread_mutex_trylock(&pmutex);
    if (status) {
        if (status == EBUSY)
            printf("ERROR: mutex still locked.\n");
        else
            printf("ERROR:pthread_mutex_trylock() returned %d\n",
                    status);
        exit(1);
    }
    (void) pthread_mutex_unlock(&pmutex);

    * All done.
    */
    pthread_exit((void *)0);
}


/*
 * Cleanup routine for the pthread_cond_wait() call.
 */
void
cond_cleanup(void *arg)
{
    int status;

    /* When canceled, should be called with the mutex locked.
    */
    status = pthread_mutex_trylock(&pmutex);
    if (status != EBUSY) {
        printf("ERROR: mutex not locked?\n");
        exit(1);
    }
    /* Unlock the mutex before exiting.
    */
    (void) pthread_mutex_unlock(&pmutex);

    was_canceled++;
}


/*
 * pthread_create() thread starts here.
 */
void
new_thread(void *arg)
{
    int status;

    (void) pthread_mutex_lock(&pmutex);

    is_waiting++;/* main thread waits for this count */

    /* Push a cleanup routine and block on the condition variable.
    */
    pthread_cleanup_push(cond_cleanup, (void *)NULL);

    (void) pthread_cond_wait(&pcond, &pmutex);

    /* The main thread should have canceled us, so we shouldn't
    * be here.  Note that the pthread_cleanup_pop() is necessary
    * for proper compilation.
    */
    printf("ERROR: Shouldn't be here. Not canceled.\n");
    exit(1);

    pthread_cleanup_pop(0);
}
```

Disabled Cancellation Example

This example shows how a thread may temporarily disable cancellations until it is ready to allow a cancellation to possibly occur. In this example, the thread being canceled disables cancellations until the mutex associated with the condition variable is unlocked. This example also shows how **`pthread_testcancel()`** may be used to cancel the calling thread.

Most of the error checking that would normally be done on function calls has been left out in order to make the code easier to read.

```
#include <pthread.h>

/* Set when target thread has called pthread_cond_wait() */
int is_waiting;

void new_thread(void *);
```

```
/* for blocking and synchronizations. */
pthread_cond_t pcond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t pmutex = PTHREAD_MUTEX_INITIALIZER;

main()
{
    int status;
    pthread_attr_t attr;
    pthread_t pthr_id;
    void *exit_status;


    /* Setup attributes for thread creation, and create the thread
    * that will be canceled.  (A bound thread will be created
    * in this example.)
    */
    (void) pthread_attr_init(&attr);
    (void) pthread_create(&pthr_id, &attr,
        (void *(*) (void *))new_thread, (void *)NULL);

    /* Wait for other thread to get blocked in pthread_cond_wait().
    */
    while (1) {
        (void) pthread_mutex_lock(&pmutex);
        if (is_waiting) {
                (void) pthread_mutex_unlock(&pmutex);
                break;
        }
        (void) pthread_mutex_unlock(&pmutex);
        while (sleep(1)) ;
    }

    /* Send a cancellation to the target thread.
    * It will not be immediately processed, since the target
    * thread has cancellations disabled.
    */
    (void) pthread_cancel(pthr_id);

    /* Now wakeup the target thread.
    */
    (void) pthread_cond_signal(&pcond);

    /* Should be able to join with terminated thread.
    * The exit status value should be PTHREAD_CANCELED.
    */
    (void) pthread_join(pthr_id, &exit_status);
    if (exit_status != PTHREAD_CANCELED) {
        printf("ERROR: invalid exit status 0x%x\n", exit_status);
        exit(1);
    }

    /* All done.
    */
    pthread_exit((void *)0);
}


/*
 * Target thread starts execution here.
 */
void
new_thread(void *arg)
{
    int status, oldstate;


    /* Disable cancellations.
    */
    (void) pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);

    /* Call the condition wait.
    */
    (void) pthread_mutex_lock(&pmutex);
    is_waiting++;

    /* main thread waits for this count */
    (void) pthread_cond_wait(&pcond, &pmutex);

    /* Let go of the mutex while we still have cancellations disabled.
    */
    (void) pthread_mutex_unlock(&pmutex);

    /* Enable cancellations.
     */
    (void) pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);

    /* We will now process our pending cancellation request.
    */
    (void) pthread_testcancel();

    printf("ERROR: Was not canceled.\n");
    exit(1);
}
```

Threads Concurrency Level

Lightweight Processes

The operating system kernel is not aware of the multithreading of any process using the Threads Library. The kernel supports an entity known as a lightweight process (LWP).

- There may be many LWPs associated with a single process.

- Each LWP of a process shares the process address space with its sibling LWPs.
- Each LWP has its own scheduling context.
- On multiprocessor machines, several LWPs of a process might be running on different processors simultaneously (true concurrency).
- Each LWP has access to all of the resources of the process such as open file descriptors, access rights and privileges, and resource limits.

Many of these are features of threads as well. This is no coincidence. Threads have many of these features because a thread only executes once it has been picked up by an LWP. However, a process typically has more threads than LWPs.

Conceptually, an LWP is a dearer resource than a thread. The Threads Library will typically maintain a pool of LWPs that are shared by the set of runnable threads in a process.

NOTE

Analogously, the operating system arranges for the sharing of a relatively small number of (hardware) processors among a much greater number of processes or LWPs.

Multiplexed Threads

The Threads Library multiplexes threads among the pool of available LWPs for the process. For threads with real-time scheduling deadlines, the additional overhead of the threads library-level scheduler can cause unacceptable delays in thread scheduling. Threads with real-time scheduling deadlines should instead be bound threads (see “Bound Threads,” page 11-34).

- An LWP can pick up and run only one thread at a time.
- After a time the LWP will put down (stop running) its current thread and pick up another.
- Some time later the thread will be picked up again; not necessarily by the previous LWP.
- The algorithm by which a thread is associated with an LWP and later preempted is covered in the section on Multiplexed Thread Scheduling below.
- On multiprocessor systems, a larger number of LWPs implies a greater *chance* that different threads of the process will be executed simultaneously (that is, true, not logical concurrency).

Managing Threads Concurrency

The size of the pool of available LWPs (the actual concurrency level) will vary over time. The Threads Library manages the size of this pool automatically and dynamically according to rules outlined below. The programmer can influence the algorithm by changing the requested concurrency level (with **`thr_setconcurrency`**, see below); at times, the actual concurrency level may be either greater than or less than the requested level.

The rules governing implicit changes to the actual concurrency level are:

- Initially, for each program, there is a single LWP available for execution of threads.
- The size of the pool is incremented when a thread is created with the *THR_INCR_CONC* flag to **`thr_create(3thread)`**. This may increase the actual level of concurrency above the current requested level of concurrency.

NOTE

> The newly created thread is not necessarily picked up by that newly created LWP. In fact, the new LWP is created asynchronously by a housekeeping thread that the Threads Library creates for each process.

- If all of the LWPs of a process are blocked in system calls, then the process cannot execute any threads. However, the kernel sends a `SIGWAITING` type signal to the process when this condition occurs. Additional LWPs are created if there are additional runnable threads.
- The number of LWPs should not exceed the number of threads — at least not for long — that would be wasteful. An LWP that remains unassigned to a thread for a certain time (5 minutes) is said to have aged and will be terminated (**`_lwp_exit(2)`**). Aging will terminate LWPs until the size of the pool equals the lesser of
  - requested level of concurrency
  - number of active (running or runnable) threads.

  Thus, if there are few threads, the actual number of LWPs may be less than the requested level.

A thread can use the **`thr_setconcurrency(3thread)`** function to change the requested concurrency level mentioned in the algorithm above. The syntax is:

```
int thr_setconcurrency(
    int new_level
);
```

This request is serviced asynchronously.

The rules governing the explicit changes in actual concurrency by **`thr_setconcurrency(3thread)`** are:

- When the level is increased, the number of LWPs is increased asynchronously by a housekeeping thread.

  One implication is that certain errors (for example, `EAGAIN` — system limit on user (for LWPs) exceeded) may not be reported because they occur after **`thr_setconcurrency`** returns.

- A request to lower the level of concurrency does not have an immediate effect: no LWP is terminated, nor is any thread preempted. Instead, the actual level of concurrency becomes lower by the LWP aging described above.
- Setting *new_level* to `0` requests the default level of concurrency.
- The programmer can retrieve the current value of the requested level of concurrency with the **`thr_getconcurrency(3thread)`** function.
- There is no mechanism to return the current, actual level of concurrency.

NOTE

The POSIX specification does not provide any POSIX Thread interface or function for direct control of thread concurrency levels.

Bound Threads

A thread may become runnable at a time when all LWPs of the process are already executing threads. That thread will be made runnable and enqueued until an LWP becomes available. This implies some latency between thread awakening and execution. There may be circumstances where this behavior is not acceptable. (Perhaps the thread must respond to a signal in a timely manner.)

If a thread is created with either the `THR_BOUND` flag to **`thr_create(3thread)`** or the `PTHREAD_SCOPE_SYSTEM` attribute (the default) is defined on a **`pthread_create(3pthread)`** call, then:

- Both a thread and a new LWP are created.
- The new LWP picks up the new thread.
- That association remains in effect for the life of the thread.
- Such threads are called bound threads.

Bound threads are not counted in the algorithm that manages the level of concurrency.

NOTE

Bound threads are not guaranteed to gain processor time whenever they are ready to execute; the LWP on which a bound thread runs must be scheduled to run on a processor by the system scheduler. See “Bound Thread Scheduling,” on page 11-37. Nevertheless, bound threads have a performance advantage over multiplexed threads.

The Initial (Primordial) Thread

Note

Information provided in this section is only applicable 4.3P10 and beyond, and for 5.1SR3 and beyond.

As was previously mentioned, the thread that executes the main() routine is known as the initial, or primordial, thread. By default, this thread is created internally by the thread library as a multiplexing thread. Therefore, all the characteristics that are associated with other multiplexing threads also apply to this initial thread.

In some situations, applications may wish to have their initial thread be created as a bound thread. An application may indicate to the thread library initialization code that it desires a bound thread as the primordial thread, by placing the following global variable declaration and static initialization somewhere in their application code:

```
int __primordial_bound = 1;
```

The above line will cause the thread library to setup the primoridal thread as a bound thread instead of a multiplexing thread. Note that the *__primordial_bound* global variable MUST be statically initialized to a value of 1.

Thread Scheduling

Thread scheduling governs the competition among threads for various system resources.

- Multiplexed threads vie for a limited number of LWPs.
    - Bound threads are spared this competition; each maintains its association with its LWP for its lifetime.
- LWPs are, in turn, assigned by the kernel to a limited number of (hardware) processors for execution.
- To coordinate their activities, threads often make use of various synchronization mechanisms. At times there may be more than one thread waiting for a given event (for example, the unlocking of a semaphore). The Threads Library must decide which thread will receive the resource.

  This last category of thread scheduling will be covered in the section entitled “Synchronizing Threads.”

Multiplexed Thread Scheduling

Multiplexed threads are subjected to two levels of scheduling:

- Threads Library Scheduling: The Threads Library scheduler assigns multiplexed threads to LWPs for execution and, at times, preempts them so the LWP can pick up another thread.

- System Scheduling: The kernel assigns LWPs to (hardware) processors and later preempts them.

The Threads Library maintains a *priority level* for each multiplexed thread. This value plays a role in the selection of a thread for assignment to an LWP.

Priority for PowerMAX OS Threads

The priority value of a PowerMAX OS multiplexed thread can be modified with the **`thr_setprio(3thread)`** function.

```
int thr_setprio(thread_t tid,int prio);
```

Priority for POSIX Threads

The priority of an existing POSIX multiplexed thread can be modified with the **`pthread_setschedparam(3pthread)`** function:

```
int pthread_setschedparam(pthread_t thread, int policy, const
struct sched_param *param);
```

For multiplexed threads, the *policy* parameter should always be set to *SCHED_OTHER*, and the priority value should be stored into the **`policy_params[0]`** location of the **`sched_param`** *param* structure.

It is also possible to specify the priority of a POSIX multiplexed thread at thread creation time. See “POSIX Thread Creation Attributes” on page 11-9.

Runnable, multiplexed threads are scheduled for execution in a round-robin manner within each priority level.

- A thread with a higher priority value will be scheduled to run before a thread with a lower value.
- The valid range of priorities is `0` to `MAXINT-1`; however, the Threads Library is optimized for a maximum priority of `126` (or less).

The Threads Library must select a thread for assignment to an LWP on the following occasions:

- When an LWP becomes available, a runnable multiplexed thread will be assigned to it.

  For example, an LWP becomes available when a thread exits, or when a multiplexed thread blocks on a thread synchronization mechanism (discussed later), or when the concurrency level is increased.

- When a multiplexed thread becomes runnable (perhaps a mutex has been released by one thread and acquired by another), it can preempt a multiplexed thread of a lower priority.
- When an executing thread calls **`thr_yield(3thread)`**, it deliberately surrenders its LWP to a higher priority thread (if any). (POSIX threads should use the POSIX-compliant **`sched_yield(3C)`** instead.)

Threads Library scheduling and system scheduling are independent of each other.

- The Threads Library can assign a thread to an LWP but cannot say when that LWP will actually execute.
- The kernel is unaware that the Threads Library is using LWPs to implement (user-level) threads. The kernel maintains its own scheduling context (for example, current priority, “nice value,” priority class) that is separate from similar features that the Threads Library maintains for threads.

The interaction of these two levels of scheduling can produce some interesting effects:

- LWPs of the time-sharing priority class will have their kernel priority adjusted dynamically according to processor usage and other factors.

NOTE

See the **`priocntl(2)`** manual page for further details of the time-sharing priority class. Note that using the **`priocntl(2)`** system call directly from a multiplexed thread should be avoided because it may interfere with thread scheduling by the Threads Library.

Consequently, a thread picked up by an LWP may run with a kernel priority determined by the activity of the *prior* thread on that LWP.

- It is possible for a thread of high priority from the point of view of the Threads Library to be picked up by an LWP of relatively low priority to the kernel.

Additional points to consider:

- A thread that is blocked in a system call will remain with its LWP until that system call returns. The Threads Library is unaware of such suspensions.
- Each LWP in the pool used for multiplexed LWPs is of the same kernel scheduling class (that is, time-sharing or fixed priority). That class is determined by the scheduling class (that is, time-sharing or fixed priority) of the LWP running the initial thread of the program.
- One part of associating a thread with an LWP is to make the signal mask of the LWP agree with that of the thread. On each thread context switch there is a check for agreement. If the mask of the new thread differs from that of the prior thread, there is a system call to update the mask of the LWP. One implication of this is that using threads with a wide variety of signal masks can add to the cost of switching threads.

Bound Thread Scheduling

The semantics of bound thread scheduling differs considerably from that for multiplexed threads.

- Bound threads are permanently attached to their LWPs; consequently, they are exempt from that level of scheduling by the Threads Library.

- A bound thread executes whenever the kernel schedules its underlying LWP.
- The Threads Library supports the concept of “scheduling policy” as well as “priority level” for bound threads. When the programmer specifies these characteristics, the Threads Library applies them to the LWP holding the thread.

  These characteristics can be modified with the **`thr_setscheduler(3thread)`** function. POSIX threads should use **`pthread_setschedparam(3pthread)`** or initially create the thread with the desired scheduling priority and scheduling policy. See “POSIX Thread Creation Attributes” on page 11-9.
- The available scheduling policies for bound threads are:

  `SCHED_TS` or `SCHED_OTHER`
  : The two values are synonymous. The bound thread is run by an LWP of the kernel time-sharing scheduling class.

NOTE

Technically, multiplexed threads are also categorized as having the `SCHED_TS` policy even though they are not necessarily run by LWPs in the kernel time sharing class. The Threads Library algorithm for scheduling multiplexed threads (round robin) bears a closer resemblance to the kernel's fixed priority class than the kernel's time sharing class.

`SCHED_FIFO` or `SCHED_RR`
: The thread will be run on an LWP of the fixed-priority scheduling class. `SCHED_FIFO` means that the LWP will have an infinite time quantum (not preempted) whereas `SCHED_RR` (round-robin) uses a fixed priority with a finite time slice.

  The `SCHED_FIFO` and `SCHED_RR` policies can be used only by bound threads.

NOTE

Appropriate privilege is required to set the policy of a thread to `SCHED_FIFO` or `SCHED_RR`. See the system manual page for **`sched_setscheduler(3C).`**

A bound thread with real-time constraints can further improve response time by using **`processor_bind(2)`** to bind its LWP to a processor. It can use **`_lwp_self(2)`** to find the ID of the LWP to which it is bound, and pass that as an argument to `processor_bind`.

NOTE

Multiplexed threads should not use `processor_bind`.

Managing Thread Scheduling

The initial thread of a newly executing program (a process returning from **`exec(2)`**) is always a multiplexed thread running under the `SCHED_TS` policy. The scheduling characteristics of new threads are generally derived from the creator thread. (There are some interesting variations when a bound thread creates a multiplexed thread and *vice versa*. See the **`thr_create(3thread)`** or **`pthread_create(3pthread)`** manual page for details.)

PowerMAX OS Thread Scheduling

To create a thread with different scheduling characteristics the programmer can:

1. Create a new thread with **`thr_create(3thread)`** using the `THR_SUSPENDED` flag. This will create a new thread but not allow it to execute.
2. Use the returned thread ID to change the characteristics of the new thread with either of the following functions: **`thr_setprio(3thread)`** or **`thr_setscheduler(3thread)`**.
3. Use the **`thr_continue(3thread)`** function to make the new thread runnable.

Alternatively, a thread can use **`thr_setscheduler(3thread)`** or **`thr_setprio(3thread)`** to modify its own scheduling class or priority.

POSIX Thread Scheduling

To create a POSIX thread with different scheduling characteristics, the programmer can:

1. Initialize a POSIX thread attributes structure with **`pthread_attr_init(3pthread)`**.
2. Set the desired scheduling policy attribute in the thread attributes structure with **`pthread_attr_setschedpolicy(3pthread)`**.
3. Set the desired scheduling priority attribute in the thread attributes structure with **`pthread_attr_setschedparam(3pthread)`**.
4. Set the inherit scheduling thread attribute in the thread attributes structure to PTHREAD_EXPLICIT_SCHED with **`pthread_attr_setinheritsched(3pthread)`**.
5. Create the thread with **`pthread_create(3pthread)`**, passing the threads attributes structure as the *attr* parameter.

Alternatively, a POSIX thread can use **`pthread_setschedparam(3pthread)`** to modify its own or an existing scheduling priority and policy.

Using fork(2)

The functionality provided by **fork(2)** differs between POSIX thread and PowerMAX OS thread applications.

For PowerMAX OS applications, two variations of **fork(2)**, **forkall(2)** and **fork1(2)**, are provided. **forkall(2)** (which is a synonym for **fork(2)**) duplicates in the new process the set of threads and underlying LWPs that exist in the calling process. **fork1(2)**, on the other hand, creates a new process with a single thread and a single LWP. **fork1(2)** should be used by multithreaded processes that will have the new process call **exec(2)**. Because **exec(2)** will terminate all but one thread (and LWP), there is no need to duplicate all threads with **forkall(2)**.

Contrastingly, when a POSIX thread calls **fork(2)**, only the calling thread (and underlying LWP) of the parent process is duplicated in the new child process. When the child process returns from the **fork(2)** call, the returning thread will be the only user thread that exists within the new child process. Note that both the **forkall(2)** and **fork1(2)** services are not POSIX-compliant services and therefore, a POSIX-compliant application should not usually make use of these services.

When a POSIX thread calls **fork(2)**, the new process will contain a replica of the calling thread and its entire address space, possibly including the state of mutexes, condition variables, and other resources.

The POSIX function, **pthread_atfork(3pthread)**, may be used to establish fork handlers whose intended purpose is to maintain data structure consistency across **fork(2)** calls:

```
int pthread_atfork(
        void (*prepare(void), void (*parent)(void), void (*child)(void));
```

One or more **pthread_atfork(3pthread)** function calls may be made by a process before a subsequent f**ork(2)** call is made by one or more of the threads in the parent process. The **pthread_atfork(3pthread)** calls establish fork handler routines that are to be called before and after the **fork(2)** actually occurs.

When the **fork(2)** call is made, the prepare fork handler(s) are called by the calling thread in the parent process, before the **fork(2)** processing commences.

Upon successful return from the **fork(2)** system service call, the parent fork handler(s) are called by the thread that made the f**ork(2)** call within the parent process. The child fork handler(s) are called by the child process's thread upon return from the **fork(2)** system service call.

If no handler is desired at one or more of these three points, then the corresponding fork handler addresses may be set to NULL.

Note that once any atfork handlers have been established by a **pthread_atfork(3pthread)** function call, all subsequent **fork(2)** calls will make use of these atfork handler routines for as long as that process exists. Also note that a child process will inherit the set of atfork handlers from the parent process, if any existed at the time that the child process was created via f**ork(2)**.

The ordering of **pthread_atfork(3pthread)** calls is significant. The parent and child fork handlers are called in the same order in which they were established by calls to

pthread_atfork(3pthread). For example, the first parent and/or child handler called after the **fork(2)** processing completes will be the handler that was previously established on the first **pthread_atfork(3pthread)** call.

However, the prepare fork handlers are called in the opposite order; that is, the last handler established on the last **pthread_atfork** call will be the first prepare handler called.

For example, an application can supply a prepare routine that acquires the necessary mutexes, a parent routine that releases those same mutexes, and a child routine that re initializes those same mutexes, thus ensuring that both the parent and child processes get consistent snapshots of the mutex states. Note that the child process will only contain one thread, and therefore any private (PTHREAD_PROCESS_PRIVATE) mutexes may be safely re initialized. Furthermore, re initializing rather than unlocking the mutex is the recommended procedure in this situation, since the calling child thread of a **pthread_mutex_unlock(3pthread)** call would not be the owner of the mutex; the owner would be the parent process's thread that originally called **fork(2)**.

A pthread_atfork() Example

A sample program is show below. This program shows how the state of two mutexes, mutex1 and mutex2, can be properly maintained across a **fork(2)** call. For clarity, the error returns are not coded, and additional threads in the parent process that could also be using mutex1 and mutex2 are not shown.

Note that if a locking hierarchy between mutex1 and mutex2 did exist, then this example is coded such that mutex2 would usually be acquired BEFORE mutex1, and mutex1 would be unlocked before unlocking mutex2.

The example below could have been coded with just one set of prepare, parent and child *atfork* handler routines that would operate on both mutexes, but two sets of handler routines were used in order to demonstrate the calling order of these handlers.

```
#include <pthread.h>

/* Internal routines.
 */
void child_routine(void);
void prepare1(void), prepare2(void);
void parent1(void), parent2(void);
void child1(void), child2(void);

/* Example mutexes that are acquired and released or re-initialized
 * around the fork(2) call.
 */
pthread_mutex_t mutex1, mutex2;

/* Counters that verify/demonstrate the calling order of the
 * prepare, parent and child atfork handlers.
 */
int prepare;
int parent;
int child;

main()
```

```
{
    int status;
    pid_t pid;

    /* Initialize the mutexes to default attributes.
    * These mutexes are PTHREAD_PROCESS_PRIVATE.
    */
    (void) pthread_mutex_init(&mutex1, (pthread_mutexattr_t *)NULL);
    (void) pthread_mutex_init(&mutex2, (pthread_mutexattr_t *)NULL);

    /* Push two sets of atfork handlers.
    */
    (void) pthread_atfork(prepare1, parent1, child1);
    (void) pthread_atfork(prepare2, parent2, child2);

    /* Fork off a new process.
    */
    pid = fork();
    if (pid == -1) {
        printf("fork(2) errno %d\n", errno);
        exit(1);
    }

    /* Upon return from the fork(2) call for both the child and
    * parent processes, neither of the two mutexes should be locked.
    */
    status = pthread_mutex_trylock(&mutex1);
    if (status) {
        printf("ERROR: trylock 1 %d\n", status);
        exit(1);
    }

    status = pthread_mutex_trylock(&mutex2);
    if (status) {
        printf("ERROR: trylock 2 %d\n", status);
        exit(1);
    }

    /* The fact that the two prepare handlers were called should be
    * visible to both the parent and child processes.
    */
    if (prepare != 2) {
        printf("error on prepare %d\n", prepare);
        exit(1);
    }

    if (pid) {
        /* The parent process.
        * Both parent handlers should have been called.
        */
        if (parent != 2) {
                    printf("error on parent %d\n", parent);
                    exit(1);
        }
        /* The child handlers should not have been called within
        * this parent process.
        */
        if (child) {
                    printf("error on child routine execution %d\n", child);
                    exit(1);
        }
    }
    else {
        /* Both child handlers should have been called in this
        * child process.
        */
        if (child != 2) {
                    printf("error on child %d\n", child);
                    exit(1);
        }

        /* The parent handlers should not have been called within
        * this child process.
        */
        if (parent) {
                    printf("error on parent routine execution %d\n",
                                parent);
                    exit(1);
        }
    }

    /* All done.
    */
    pthread_exit((void *)0);
}

void
prepare1()
{
    /* mutex1 is locked AFTER mutex2 has been locked.
    */
    (void) pthread_mutex_lock(&mutex1);

    if (pthread_mutex_trylock(&mutex2) != EBUSY) {
        printf("prepare1() mutex2 not locked.\n");
        exit(1);
    }

    /* The prepare2() routine should have already been called,
    * and no child or parent handlers should have been called.
    */
    if ((prepare != 1) || child || parent) {
        printf("prepare1() wrong order\n");
        exit(1);
    }
    prepare++;
}
```

```
void
prepare2()
{
    /* mutex2 is the 1st mutex to be locked.
    */
    (void) pthread_mutex_lock(&mutex2);

    /* prepare2() is the 1st prepare handler to be called.
    */
    if (prepare || child || parent) {
        printf("prepare2() wrong order\n");
        exit(1);
    }
    prepare++;
}

void
parent1()
{

    /* Unlock mutex1 first, since it was locked last.
    */
    (void) pthread_mutex_unlock(&mutex1);

    /* This should be the 1st parent handler called, and both
    * prepare handlers should already have been called.
    */
    if (parent || (prepare != 2)) {
        printf("parent1() wrong order\n");
        exit(1);
    }
    parent++;
}

void
parent2()
{
    /* mutex2 is unlocked last, since it was the first mutex to be locked.
    */
    (void) pthread_mutex_unlock(&mutex2);

    /* This should be the 2nd parent handler to be called.
    */
    if (parent != 1) {
        printf("parent2() wrong order\n");
        exit(1);
    }
    parent++;

}
```

```
void
child1()
{
    /* Re-initialize the mutex mutex1.
    */
    (void) pthread_mutex_init(&mutex1, (pthread_mutexattr_t *)NULL);

    /* This should be the 1st child handler to be called.
    * Both prepare handlers should have already been called.
    */
    if (child || (prepare != 2)) {
        printf("child1() wrong order\n");
        exit(1);
    }
    child++;
}

void
child2()
{
    /* Re-initialize the mutex mutex2.
    */
    (void) pthread_mutex_init(&mutex2, (pthread_mutexattr_t *)NULL);

    /* child1() should already have been called.
    */
    if (child != 1) {
        printf("child2() wrong order\n");
        exit(1);
    }
    child++;
}
```

Synchronizing Threads

In general, each thread must take special care in using resources that might be concurrently used by another thread.

NOTE

The definition of *resource* will vary with applications. Typically, resources are manifested as some organization of data relevant to the application in process memory (perhaps a linked list or other data structure) or in files.

Unless their actions are synchronized, threads may encounter logically inconsistent linked lists or partially updated structures in common process memory. Synchronization may also be needed for concurrent actions on commonly held external resources such as file descriptors and message queues.

- There is no automatic, implicit mechanism to protect each thread from the actions of other threads. The correctness of a multithreaded program must be incorporated into the design by having each thread cooperate with the others.
- The Threads Library provides a suite of functions with several categories of synchronization semantics. The categories are:
  - Locks
  - Semaphores
  - Barriers
  - Condition Variables

  Most of these categories contain several variants.
- The programmer has the responsibility to:
  - use the correct number and type of synchronization mechanism(s)
  - use them where needed
  - enforce synchronization on *every* thread using the common resource
  - avoid deadlock and starvation conditions
- Other than programmer discipline, there is nothing to stop any thread from using common resources without obeying the synchronization protocol being used by the others.
- The general procedure for using these mechanisms is:
  1. Allocate a synchronization data structure for the resource to be protected (for example, to use a mutual exclusion lock, allocate a structure of type `mutex_t`). The address of that structure becomes an argument for all subsequent operations on this instance of the mechanism.
  2. Initialize the mechanism.
  3. Use the mechanism.
  4. Deallocate the mechanism when it is no longer needed — perhaps when the resource being protected is deallocated.
- Source code that uses the PowerMAX OS synchronization routines in the Threads Library should include the following line

  ```
  #include <synch.h>
  ```
- Source code that uses the POSIX thread synchronization routines (3pthread) in the Threads Library should include the following line:

  ```
  #include <pthread.h>
  ```

Locks

The semantics of a *lock* allow the resource to be used by only one thread at a time. The Threads Library supports several types of locks:

- mutual exclusion locks (“mutexes”)
- spin locks
- recursive mutual exclusion locks (rmutexes)
- reader-writer locks (these allow non-exclusive access for readers)

A thread that successfully locks a resource is said to hold the lock or to have acquired the lock. Unlocking is also known as releasing the lock.

NOTE

The POSIX Thread interface only supports the mutual exclusion locks. It does not support the other 3 types of locks listed above. However, the programmer may construct the other 3 types of locks by using the available POSIX Thread locking functions. See the following sections for details on this topic.

Mutual Exclusion Locks

A *mutual exclusion lock*, or mutex, allows only one thread at any time to access the resource being protected.

PowerMAX OS Mutex Lock Interface

The lock is acquired by the **`mutex_lock(3synch)`** function.

```
int mutex_lock(mutex_t *mutex);
```

If the lock is already held by some other thread, the calling thread will block in **`mutex_lock(3synch)`**.

A non-blocking attempt to acquire a mutex lock can be done with the **`mutex_trylock(3synch)`** function:

```
int mutex_trylock(mutex_t *mutex);
```

If the lock can be acquired, then this function will return 0. Otherwise, when the lock is already locked by another thread, the caller will NOT block, but instead, EBUSY will be immediately returned.

When the thread holding the lock calls **`mutex_unlock(3synch)`**, some waiting thread (if any) will be made runnable.

```
int mutex_unlock(mutex_t *mutex);
```

The Threads Library does not enforce any notion of ownership of a lock by a thread. The thread unlocking a mutex need not be the same thread that locked the mutex.

POSIX Mutex Lock Interface

The mutex lock can be acquired with the **`pthread_mutex_lock(3pthread)`** function.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

If the lock is already held by some other thread, the calling thread will block in **`pthread_mutex_lock(3pthread)`**.

A non-blocking attempt to acquire a mutex lock can be done with the **`pthread_mutex_trylock(3pthread)`** function:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

If the lock can be acquired, then this function will return 0. Otherwise, when the lock is already locked by another thread, the caller will NOT block, but instead, EBUSY will be immediately returned.

When the thread holding the lock calls **`pthread_mutex_unlock(3pthread)`**, some waiting thread (if any) will be made runnable.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

It is recommended that the owner of the thread holding the lock be the one to call **`pthread_mutex_unlock(3pthread)`** to unlock the mutex, although this restriction is not currently enforced.

POSIX Priority Ceiling Protocol Mutexes

POSIX mutual exclusion locks also contain a protocol attribute which is associated with each `pthread_mutex_t mutex`. By changing the default value of this protocol attribute, the resulting POSIX mutex can greatly reduce, or eliminate potential priority inversions that are not otherwise prevented by traditional mutex usage.

In a priority-driven environment, direct use of traditional primitives like mutexes and condition variables can lead to unbounded priority inversion, where a higher priority thread can be blocked by a lower priority thread, or set of threads, for an unbounded duration of time. As a result, it becomes impossible to guarantee thread deadlines. Priority inversion can be bounded and minimized by the use of priority inheritance protocols. This allows thread deadlines to be guaranteed even in the presence of synchronization requirements.

The IEEE POSIX 1003.1 1996 Edition specifies a Priority Ceiling Protocol Emulation protocol, governed by the `_POSIX_THREAD_PRIO_PROTECT` option in <unistd.h>, where each mutex may have a priority ceiling, usually setup to be equal to the priority of the highest priority thread that will lock the mutex. When a thread is executing inside critical sections, its priority is unconditionally increased to the highest of the priority ceilings of all the mutexes currently owned (locked) by the thread. The Priority Ceiling Protocol is supported for POSIX threads in the thread library and therefore, the `_POSIX_THREAD_PRIO_PROTECT` option is set to a value of '1' in <**`unistd.h`**>.

NOTE

It should be mentioned that the IEEE POSIX 1003.1 1996 Edition also defines the Basic Priority Inheritance protocol, governed by the `_POSIX_THREAD_PRIO_INHERIT` option in <unistd.h>. Under this protocol, a thread executes at the priority of the highest priority thread that is currently blocked on the mutex. This protocol is not currently supported in the thread library and the corresponding `_POSIX_THREAD_PRIO_INHERIT` option is set to a value of '0' in <unistd.h>.

Priority Ceiling Mutex Restrictions

Applications wishing to use the Priority Ceiling Protocol for a given mutex may do so by setting the protocol mutex attribute to a value of `PTHREAD_PRIO_PROTECT`, using the **`pthread_mutexattr_setprotocol(3pthread)`** function. However, programmers should be aware of the fact that there are certain restrictions regarding the use of this protocol.

The `PTHREAD_PRIO_PROTECT` mutex attribute will only be enforced for bound (`PTHREAD_SCOPE_SYSTEM`) threads in the `SCHED_RR or SCHED_FIFO` scheduling classes. Threads in the `SCHED_OTHER` (`SCHED_TS`) scheduling class will not have their priority altered when locking a `PTHREAD_PRIO_PROTECT` mutex. This applies to all multiplexed threads and bound threads that are in the `SCHED_OTHER` scheduling class.

For efficiency and performance reasons, the thread library makes use of internally cached information about a thread's scheduling class and priority. Internally saving off this information allows decisions about raising or lowering a thread's priority to be made without making a system service call at every mutex lock or unlock point in order to determine the current thread's priority and scheduling class. However, as a result, processes should not modify the scheduling class or priority of threads in other processes that are using `PTHREAD_PRIO_PROTECT` mutexes. Otherwise, the thread library will not be using accurate priority scheduling information for determining the correct action to take when a thread locks or unlocks a `PTHREAD_PRIO_PROTECT` mutex.

Note that it is permissible for a thread to change the scheduling class and/or priority of itself or any other thread within the process, using **`pthread_setschedparam (3pthread)`**; the thread library will have knowledge of these changes and it will update its internal information accordingly. However, if the thread being modified currently owns a `PTHREAD_PRIO_PROTECT` mutex (holds the lock), then unspecified results will occur when that thread unlocks the mutex.

Other methods for modifying the scheduling class and/or priority within or without a thread's own process, such as **`priocntl(2)`**, **`priocntllist(2)`** and **`priocntlset(2)`**, should be avoided when `PTHREAD_PRIO_PROTECT` mutexes are being used.

Also note that the **`sched_setscheduler(3C)`** and **`sched_setparam(3C)`** functions are generally not recommended for multithreaded processes. See the **`sched_setscheduler(3C)`** or **`sched_setparam(3C)`** man pages for more information on this subject.

Initializing PTHREAD_PRIO_PROTECT Mutexes

By default, a mutex is initialized to have a protocol attribute of `PTHREAD_PRIO_NONE.` The `PTHREAD_PRIO_NONE` protocol attribute is set dynamically on a **`pthread_mutex_init(3pthread)`** function call, or statically with the `PTHREAD_MUTEX_INITIALIZER` macro (see the **`pthread_mutex_init(3pthread)`** man page for more details). A `PTHREAD_PRIO_NONE` mutex behaves in the traditional fashion; no adjustment of priority will occur when a thread locks or unlocks the mutex.

A Priority Ceiling Protocol mutex may be created by using the **`pthread_mutexattr_setprotocol(3pthread)`** function, and specifying the `PTHREAD_PRIO_PROTECT` value as the protocol parameter.

The ceiling priority value associated with a `PTHREAD_PRIO_PROTECT` mutex should be specified with the **`pthread_mutexattr_setprioceiling(3pthread)`** function call, where the priority value must be within the maximum range of priorities defined by `SCHED_FIFO/SCHED_RR` (the range is 0 to 59, since these classes correspond to the kernel's fixed priority (FP) scheduling class). In order to avoid priority inversion, the priority ceiling of the mutex should be set to a priority that is higher than or equal to the highest priority of all the threads that may lock that mutex. Also note when more than one `PTHREAD_PRIO_PROTECT` mutex may be held at the same time by one thread, the associated priority ceiling attribute of each mutex must be at equal or ascending priority values as each additional mutex is acquired; it is not valid for a thread to lock a mutex with a lower priority ceiling attribute than the current thread's priority value.

As an example, the sequence for initializing a Priority Ceiling Protocol mutex, with a priority ceiling value of 50, is shown below:

```
pthread_mutexattr_t mattr;
pthread_mutex_t mutex;

pthread_mutexattr_init(&mattr);
pthread_mutexattr_setprotocol(&mattr, PTHREAD_PRIO_PROTECT);
pthread_mutexattr_setprioceiling(&mattr, 50);
pthread_mutex_init(&mutex, &mattr);
```

In addition to the **`pthread_mutexattr_setprotocol()`** and **`pthread_mutexattr_setprioceiling()`** functions, there are corresponding **`pthread_mutexattr_getprotocol(3pthread)`** and **`pthread_mutexattr_getprioceiling(3pthread)`** functions that may be used to obtain the protocol and priority ceiling values, respectively, within a specified mutex attributes structure.

It is also possible to dynamically change the priority ceiling value associated with an already initialized `PTHREAD_PRIO_PROTECT` mutex by using the **`pthread_mutex_setprioceiling(3pthread)`** function call. In this case, the mutex is first acquired by the calling thread (the calling thread will block, if the mutex is already locked), the priority ceiling attribute is modified, and then the mutex is released/unlocked. It should be noted that the locking of the mutex on the **`pthread_mutex_setprioceiling()`** call will NOT follow the `PTHREAD_PRIO_PROTECT` protocol; the priority of the calling thread will not be raised on this function call.

Using PTHREAD_PRIO_PROTECT Mutexes

When a bound thread in the SCHED_RR or SCHED_FIFO scheduling classes locks a PTHREAD_PRIO_PROTECT mutex with the **pthread_mutex_lock(3pthread)** or **pthread_mutex_trylock(3pthread)** functions, the scheduling priority of the thread will be raised to the prioceiling mutex attribute that is associated with that mutex. If the locking thread's priority is already equal to the priority ceiling value, then no priority modification will be performed.

When a mutex has a protocol attribute of PTHREAD_PRIO_PROTECT, and the calling thread's priority is already higher than the mutex's prioceiling attribute, **pthread_mutex_lock()** or **pthread_mutex_trylock()** will return EINVAL, and the mutex will not be locked.

The priority of the thread will remain raised until that thread makes a **pthread_mutex_unlock(3pthread)** call. If additional PTHREAD_PRIO_PROTECT mutexes are acquired, then the thread's priority may be raised to higher priority levels until each of those additional mutexes are released/unlocked.

In the unlikely event that raising the priority of a thread that is locking a PTHREAD_PRIO_PROTECT mutex fails on a **pthread_mutex_lock()** or **pthread_mutex_trylock()** call, the these functions will return EPERM, with the mutex not locked. Since SCHED_RR and SCHED_FIFO threads should normally be able to raise their own priority, this error most likely would indicate that some external manipulation of the thread's or the process's scheduling parameters has been done by another process.

Note that when a thread calls **pthread_cond_wait(3pthread)** or **pthread_cond_timedwait(3pthread)** and the associated mutex is a PTHREAD_PRIO_PROTECT mutex, the calling thread's priority will be lowered back to its previous value while the mutex is unlocked and the thread is blocked on the condition variable. When the thread becomes unblocked and re-acquires the mutex, its priority will be raised back to the prioceiling priority mutex attribute value of the mutex before returning back from the conditional wait function call.

For PTHREAD_PRIO_NONE mutexes, it is highly recommended that the calling thread of **pthread_mutex_unlock()** be the thread that currently owns the mutex. For performance reasons, this recommendation is not currently enforced by the thread library.

However, for PTHREAD_PRIO_PROTECT mutexes where the priority of the owning thread has been raised while the mutex was locked, the calling thread of the corresponding **pthread_mutex_unlock()** call MUST be the thread that currently owns the mutex; otherwise, an error of EPERM will be returned, and the mutex will remain locked.

Priority Protect Mutex Example

The following coding example uses four mutexes, three of which are PTHREAD_PRIO_PROTECT mutexes, and one mutex that is a PTHREAD_PRIO_NONE mutex. Four threads are created, and each thread acquires all four mutexes, in ascending priority order, and then releases the mutexes in reverse order.

Each of the four threads are different: one is in the SCHED_FIFO class, one is in the SCHED_RR class, one is a bound thread in the SCHED_OTHER class, and one thread is a MUX thread in the SCHED_OTHER class.

The threads in the SCHED_OTHER class do not have their priority modified when locking these mutexes, but the SCHED_RR and SCHED_FIFO threads have their priority adjusted when locking and unlocking the PTHREAD_PRIO_PROTECT mutexes.

The example code checks after each mutex lock and mutex unlock operation to verify that the thread's priority and scheduling class are at the appropriate values.

Note that the priority ceiling values are based off of the kernel's fix priority (FP) scheduling class.

The main thread in this example is the thread that initializes the mutexes, creates the four additional threads, and then waits for the other threads to finish.

The appropriate error return checks from function calls were left out in order to make the code easier to read.

```
#include <pthread.h>
#include <sys/procset.h>
#include <sys/priocntl.h>
#include <sys/fppriocntl.h>

/* Internal routines.
 */
void setup_mutexes(void);
void get_max_priority(void);
void create_threads(void);
void thread_start(void *);
void run_fifo_rr(int, int);
void run_other(int);
void wait_for_done(void);

#define NUM_THREADS  4   /* uses four additional threads */
int num_ready;           /* incremented when each thread is ready to run the
test */
int num_done;            /* incremented when each thread is through testing */

/* Arg value passed to thread_start() routine.
 */
#define ARG_PRIO_RR  0   /* bound thread in SCHED_RR */
#define ARG_PRIO_FIFO1   /* bound thread in SCHED_FIFO */
#define ARG_PRIO_OTHER2  /* bound thread in SCHED_OTHER */
#define ARG_PRIO_MUX 3   /* mux thread */

int arg_array[NUM_THREADS] =
                  { ARG_PRIO_RR, ARG_PRIO_FIFO, ARG_PRIO_OTHER, ARG_PRIO_MUX };

/* Keep all the mutexes in one array.
 */
#define NUM_MUTEXES  4   /* number of mutexes to go through */
#define NUM_NONE_MUTEX 2 /* index of PTHREAD_PRIO_NONE */

pthread_mutex_t pmutex_array[NUM_MUTEXES];
int prio_array[NUM_MUTEXES];/* The associated priorities */

/* The main thread handles timeouts.
```

```
 */
#define TIMEOUT_WAIT 10  /* seconds */

/* Maximum mutex prioceiling priority value.
 */
int max_prioceiling;

/* Condition variable used to wait for all threads to get ready.
 * It's associated mutex is just a normal mutex.
 */
pthread_mutex_t pmutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t pcond = PTHREAD_COND_INITIALIZER;
int posted;         /* set just before the broadcast wakeup */


main(argc, argv)
int argc;
char **argv;
{
    /* Initialize the mutexes.
    */
    setup_mutexes();

    /* Create all the other threads and let them run the test.
    */
    create_threads();

    /* Wait for the threads to finish.
    * Timeout if some sort of lockup failure.
    */
    wait_for_done();

    /* Test is good if we reach this point without termination.
    */
    pthread_exit((void *)0);
}


/*
 * The main thread calls this routine to initialize the mutexes.
 */
void
setup_mutexes()
{
    int i, status, priority, *priop, protocol;
    pthread_mutex_t *pmp;
    pthread_mutexattr_t mattr;


    /* Get the maximum priority ceiling value.
    */
    get_max_priority();

    /* Initialize a mutex attributes structure.
    */
    (void) pthread_mutexattr_init(&mattr);

    /* Set the protocol of the mutex attributes to prio protect.
    */
    (void) pthread_mutexattr_setprotocol(&mattr, PTHREAD_PRIO_PROTECT);

    /* Use the top available priorities for the mutexes.
    * One of the mutexes is not a prioprotect mutex.
    */
    priority = max_prioceiling - NUM_MUTEXES + 2;

    for (i = 0, pmp = pmutex_array, priop = prio_array;
            i < NUM_MUTEXES; i++, pmp++, priop++)
    {
            if (i == NUM_NONE_MUTEX) {
                    (void) pthread_mutex_init(pmp,
                                (pthread_mutexattr_t *)NULL);

                    /* Save off the priority into the array.
                    * -1 is used to indicate no priority change.
                    */
                    *priop = -1;
                    printf("mutex %d no priority protocol\n", i);
                    continue;
            }

            /* Set the priority attribute of this mutex.
            */
            (void) pthread_mutexattr_setprioceiling(&mattr, priority);

            /* Initialize the mutex.
            */
            (void) pthread_mutex_init(pmp, &mattr);

            /* Save off the priority into the array.
            */
            printf("mutex %d priority %d\n", i, priority);
            *priop = priority;
            priority++;
    }
}


/*
 * Called by the main thread from setup_mutexes().
 * This routine gets the kernel's fixed priority (FP) class's
 * maximum priority value.  This is the largest value allowed
 * for a prioceiling mutex attribute.
 */
void
get_max_priority()
{
    int status;
    pcinfo_t pcinfo;
    fpinfo_t *fpp;

    strcpy(pcinfo.pc_clname, "FP");
    (void) priocntl(0, 0, PC_GETCID, &pcinfo);
    fpp = (struct fpinfo *)pcinfo.pc_clinfo;
    max_prioceiling = fpp->fp_maxpri;
    printf("FP maxpri %d\n", max_prioceiling);
}


/*
 * The main thread calls this routine to create the additional threads.
 */
void
create_threads()
{
    int i, status, timeout;
    pthread_attr_t attr;
    pthread_t id;


    /* Initialize a thread attributes structure.
    * Bound threads by default.
    */
    (void) pthread_attr_init(&attr);

    /* Create the other threads.
    */
    for (i = 0; i < NUM_THREADS; i++) {
            if (i == ARG_PRIO_MUX) {
                    /* Make the last thread a mux thread. */
                    (void) pthread_attr_setscope(&attr,
                                 PTHREAD_SCOPE_PROCESS);
            }
            (void) pthread_create(&id, &attr,
                    (void *(*)(void *))thread_start, (void *)&arg_array[i]);
    }

    /* Wait for the other threads to run and initialize themselves.
    */
    timeout = 0;
    while (1) {
            if (num_ready == NUM_THREADS)
                    break;
            if (timeout == TIMEOUT_WAIT) {
                    printf("ERROR: timeout waiting for threads.\n");
                    exit(1);
            }
            timeout++;
            while ( sleep(1) ) ;
    }

    /* Wakeup the other threads to let them run the test.
    */
    posted = 1;
    (void) pthread_cond_broadcast(&pcond);
}


/*
 * The main thread calls this routine in order to wait while the
 * other threads run and finish their testing.
 */
void
wait_for_done()
{
    int i;

    for (i = 0; i < TIMEOUT_WAIT; i++) {
            if (num_done == NUM_THREADS)
                    return;
            while (sleep(1));
    }
    printf("ERROR: timeout on locking: num_done %d num_thread %d\n",
            num_done, NUM_THREADS);
    exit(1);
}


/*
 * Created threads start here.
 * 'arg' points to an integer value that defines the type of
 * scheduling class and scope (bound/mux) to be used for the thread.
 */
void
thread_start(void *arg)
{
    int class, which = *(int *)arg, status;
    struct sched_param param;


    switch (which) {
    case ARG_PRIO_FIFO:
    case ARG_PRIO_RR:
            /* Use a low priority so that the prio protect
            * priority changes are noticeable.
            */
            param.sched_priority = 0;

            if (which == ARG_PRIO_FIFO)
                    class = SCHED_FIFO;
            else
                    class = SCHED_RR;
            (void) pthread_setschedparam(pthread_self(), class, &param);
            break;

    case ARG_PRIO_OTHER:
```

```
    case ARG_PRIO_MUX:
            /* MUX thread and SCHED_OTHER bound thread.  Get our
            * scheduling priority.  (Class is SCHED_OTHER).
            */
            void) pthread_getschedparam(pthread_self(), &class, &param);
            break;

        default:
            printf("ERROR: unexpected which arg %d tid %d\n",
                    which, pthread_self());
            exit(1);
    }

    /* Sync up with all the other threads.
    */
    (void) pthread_mutex_lock(&pmutex);
    num_ready++;
    while (!posted) {
            (void) pthread_cond_wait(&pcond, &pmutex);
    }
    (void) pthread_mutex_unlock(&pmutex);

    /* Go do the locks and unlocks and check priorities.
    */
    switch (which) {
    case ARG_PRIO_FIFO:
    case ARG_PRIO_RR:
            run_fifo_rr(param.sched_priority, class);
            break;

    case ARG_PRIO_OTHER:
    case ARG_PRIO_MUX:
            run_other(param.sched_priority);
            break;
    }

    /* Let main thread know that we're finished and then exit.
    */
    (void) pthread_mutex_lock(&pmutex);
    num_done++;
    (void) pthread_mutex_unlock(&pmutex);
    pthread_exit((void *)0);
}


/*
 * The SCHED_FIFO and SCHED_RR threads call this routine to do the locking
 * and unlocking.  The priority of these threads should be modified at each
 * lock and unlock point, except for the one mutex that has not been setup as
 * a prioprotect mutex.
 *
 * orig_pri      contains the current priority value of the thread
 * orig_class    contains the current scheduling class of the calling thread
 */
```

```
void
run_fifo_rr(int orig_pri, int orig_class)
{
    int i, class, status;
    int our_prio[NUM_MUTEXES];
    int *ourpriop = our_prio;
    int *priop = prio_array;
    pthread_mutex_t *pmp = pmutex_array;
    struct sched_param param;


    /* Push current priority into our own priority stack.
     * We save off up to the next to last mutex's resulting priority
     * change.  Used during unlocking for checking priority.
     */
    *ourpriop = orig_pri;
    ourpriop++;

    /* Loop through the mutex array, locking each mutex and
     * checking for the proper scheduling priority adjustment.
     */
    for (i = 0; i < NUM_MUTEXES; i++, pmp++, priop++, ourpriop+
            /* Get the lock. */
            (void) pthread_mutex_lock(pmp);

            /* Get our priority and class */
            (void) pthread_getschedparam(pthread_self(), &class, &param);

            /* Check the class.  Should not have changed. */
            if (class != orig_class) {
                    printf("ERROR: class changed during lock.\n");
                    exit(1);
            }

            /* Check the priority. */
            if (*priop == -1) {
                    /* Priority should not have been changed
                    * for the PTHREAD_PRIO_NONE mutex.
                    */
                    ourpriop--; /* back up to previous priority */
                    if (*ourpriop != param.sched_priority) {
                                printf("ERROR: priority changed.\n");
                                exit(1);
                    }
                    ourpriop++;
            }
            else if (param.sched_priority != *priop) {
                    /* Priority was not raised to the proper value.
                     */
                    printf("ERROR: priority not raised.\n");
                    exit(1);
            }

            /* Save current priority for unlocks
```

```
         * unless we're at the last lock.
         */
        if ((i + 1) != NUM_MUTEXES)
                *ourpriop = param.sched_priority;

        /* Communicate with the outside world.
         */
        printf("thread id %d locked mutex %d priority %d class %d\n",
                pthread_self(), i, param.sched_priority, orig_class);
    }

    /* Now unlock the mutexes in the reverse order.
     */
    ourpriop = &our_prio[NUM_MUTEXES - 1];
    pmp = &pmutex_array[NUM_MUTEXES - 1];

    for (i = 0; i < NUM_MUTEXES; i++, pmp--, ourpriop--) {
        /* Unlock the lock. */
        (void) pthread_mutex_unlock(pmp);

        /* Get our priority and class */
        (void) pthread_getschedparam(pthread_self(), &class, &param);

        /* Check the class.  Should not have changed. */
        if (class != orig_class) {
                printf("ERROR: class changed during unlock.\n");
                exit(1);
        }

        /* Check the priority. */
        if (*ourpriop != param.sched_priority) {
                printf("ERROR: invalid priority at unlock.\n");
                exit(1);
        }

        /* Communicate with the outside world. */
        printf("thread id %d unlocked mutex %d priority %d class %d\n",
                pthread_self(), NUM_MUTEXES - i - 1,
                param.sched_priority, orig_class);
    }
}


/*
 * The bound and MUX SCHED_OTHER threads call this routine to do the
 * locking and unlocking.
 *
 * 'orig_pri' contains the current priority value of the thread.
 * The priority of these threads should not change during the locking
 * and unlocking of these locks.
 */
void
run_other(int orig_pri)
{
    int i, class, status;
    pthread_mutex_t *pmp = pmutex_array;
    struct sched_param param;


    /* Loop through the mutex array, locking each mutex and
    * checking the scheduling priority and class.
    */
    for (i = 0; i < NUM_MUTEXES; i++, pmp++) {
            /* Get the lock. */
            (void) pthread_mutex_lock(pmp);

            /* Get our priority and class */
            (void) pthread_getschedparam(pthread_self(), &class, &param);

            /* Check the class.  Should not have changed. */
            if (class != SCHED_OTHER) {
                    printf("ERROR: class changed during lock.\n");
                    exit(1);
            }

            /* Check the priority. */
            if (orig_pri != param.sched_priority) {
                    printf("ERROR: priority changed during lock.\n");
                    exit(1);
            }

            /* Communicate with the outside world. */
            printf("thread id %d locked mutex %d priority %d class %d\n",
                    pthread_self(), i, param.sched_priority, SCHED_OTHER);
    }

    /* Now unlock the mutexes in the reverse order.
    */
    pmp = &pmutex_array[NUM_MUTEXES - 1];

    for (i = 0; i < NUM_MUTEXES; i++, pmp--) {
            /* Unlock the lock. */
            (void) pthread_mutex_unlock(pmp);

            /* Get our priority and class */
            (void) pthread_getschedparam(pthread_self(), &class, &param);

            /* Check the class.  Should not have changed. */
            if (class != SCHED_OTHER) {
                    printf("ERROR: class change during unlock.\n");
                    exit(1);
            }

            /* Check the priority. */
            if (orig_pri != param.sched_priority) {
                    printf("ERROR: priority change during unlock.\n");
                    exit(1);
            }
            /* Communicate with the outside world. */
            printf("thread id %d unlocked mutex %d priority %d class %d\n",
                    pthread_self(), NUM_MUTEXES - i - 1,
                    param.sched_priority, SCHED_OTHER);
    }
}
```

Spin Locks

A spin lock is also used for mutually exclusive access to some resource. The PowerMAX OS **`_spin_lock(3synch)`** function differs from **`mutex_lock(3synch)`** in implementation. If a spin lock is not available, the calling thread is not blocked, instead the caller busy waits (or *spins*) until the lock becomes available.

`_spin_lock``(spin_t *`*lock*`);`
`_spin_unlock``(spin_t *`*lock*`);`

Considerations for the use of spin locks:

- The busy waiting prevents the LWP from being used by another thread.
- This facility is intended for use when the delay is expected to be smaller than the time to context switch to another thread and back.
- Use of this facility is not recommended on uniprocessor machines or if only one processor of a multiprocessor machine might be available. In those circumstances the spinning thread prevents the possible execution of the thread that is holding the lock, thereby delaying, possibly deadlocking, itself.

CAUTION

Extreme care should be exercised in using spin locks. Deadlocks are always possible; therefore, in this release, most applications cannot use these interfaces.

POSIX Spin Locks

While POSIX Thread interfaces do not directly support spin locks, they can be built using **`pthread_mutex_trylock(3pthread)`**. The following example omits error checking for clarity:

```
/* spinlock.h */

struct spinlock {
    pthread_mutex_t s_mutex;
};


/* spinlock.c */
```

```
void
spinlock_init(struct spinlock *s)
{
    (void) pthread_mutex_init(&s->s_mutex,
    (pthread_mutexattr_t *)NULL);
}

void
spinlock_lock(struct spinlock *s)
{
    while (pthread_mutex_trylock(&s->s_mutex)) ;
}

void
spinlock_unlock(struct spinlock *s)
{
    (void) pthread_mutex_unlock(&s->s_mutex);
}
```

Recursive Mutual Exclusion

The regular mutex lock (shown earlier) will deadlock the calling thread on attempts to re-lock a lock that it already holds. A recursive mutually exclusive lock (recursive mutex or rmutex) allows the holder of a lock to re-lock without deadlock; other threads will block normally.

```
int rmutex_lock(rmutex_t *rmutex);
int rmutex_unlock(rmutex_t *rmutex);
```

Considerations for the use of recursive mutex locks:

- The holder must unlock the lock for each time it was locked.
- This facility is useful for
  - The implementation of recursive algorithms.
  - Situations where the code locking a resource cannot know which locks have already been acquired. This may arise in the implementation of library functions where generally the activities of the callers are not known.
- Recursive mutexes only prevent deadlock of a thread with itself for a single resource. It is still possible for a thread to become deadlocked even with recursive mutexes. Two (or more) threads can deadlock by each acquiring multiple locks in an unfortunate order.
- Recursive mutexes provide exclusivity but they sacrifice “atomicity.” A resource protected by an rrmutex will be used by only one thread at a time; however, that use must be designed to be reentrant because that thread might reacquire the resource in the midst of using it.

POSIX Thread Recursive Mutexes

While the POSIX Thread interfaces do not directly support recursive mutual exclusion locks, such locks can be built using **`pthread_mutex_lock()`** and **`pthread_mutex_unlock()`**. The following example omits error checking for clarity:

```
/* rmutexlock.h */

struct rmutexlock {
    pthread_mutex_t r_mutex;
    pthread_t r_id;
    int r_depth;
};


/* rmutexlock.c */

void
rmutexlock_init(struct rmutexlock *r)
{
    (void) pthread_mutex_init(&r->r_mutex,
    (pthread_mutexattr_t *)NULL);
    r->r_depth = 0;
    r->r_id = (pthread_t)-1;
}

void
rmutexlock_lock(struct rmutexlock *r)
{
    if (r->r_id == pthread_self()) {
        r->r_depth++;
        return;
    }
    (void) pthread_mutex_lock( &r->r_mutex );
    r->r_depth = 1;
    r->r_id = pthread_self();
}

void
rmutexlock_unlock(struct rmutexlock *r)
{
    r->r_depth--;
    if (r->r_depth == 0) {
        r->r_id = (pthread_t)-1;
        (void) pthread_mutex_unlock( &r->r_mutex );
    }
}
```

Reader-Writer Locks

Whereas the locks for mutual exclusion allow only one thread to use a resource at a time, the reader-writer facility supports a more complicated model of resource use. Mutual exclusion is needed only when the resource is being modified; otherwise, access need not be denied to multiple threads.

Such locks can be held in either read mode (a read lock) or write mode (a write lock).

- In read mode, there is no limit on the number of threads using the resource. By convention, each thread with such a lock assumes that the resource is stable while the lock is held. That assumption is reasonable provided no thread will modify the resource until it acquires a write lock and that is not possible while at least one read lock is being held. As usual, these assumptions are not enforced by any mechanism other than programming discipline.
- In practice, a lock held in read mode should bar only writers while a lock held in write mode should bar all readers and all other writers. Read and write locks are acquired by the **rw_rdlock(3synch)** and **rw_wrlock(3synch)** functions, respectively.

  ```
  int rw_rdlock(rwlock_t *rwlock);
  int rw_wrlock(rwlock_t *rwlock);
  ```

  If one or more threads are blocked waiting for a write lock, then any threads requesting read locks will be blocked to wait for the writer. This prevents a sequence of readers from indefinitely blocking a waiting writer.
- The order of access is strictly first-in-first-out (FIFO). This ordering is obeyed even if the readers have higher priority than the writer. This is an exception to the algorithm used to awaken threads by the other thread synchronization mechanisms. (See “Further Considerations for Synchronization Mechanisms,” page 11-74.)
- It is not possible to promote in place a read lock to a write lock. The read lock must be released and a write lock acquired in a separate operation.

POSIX Thread Reader-Writer Locks

While the POSIX Thread interfaces do not provide the reader-writer lock mechanism, it can be built using POSIX Thread features and functions. The condition variables used in the example below are described in the next section, POSIX Thread Condition Variables.

Unlike PowerMAX OS thread reader-writer locks, the code below does not implement strict FIFO access to the lock when the threads have differing scheduling policies and/or priorities. Implementing FIFO access to a reader-writer lock would require a FIFO queue of dynamically allocated condition variable structures, one for each thread that is waiting. This approach is not part of the following example, which also omits error recovery for clarity:

```
/* rwlock.h */

struct rwl_lock {
    pthread_mutex_t rw_mutex;/* for synchronization */
    pthread_cond_t rw_readcond;/* waiting readers */
    pthread_cond_t rw_writecond;/* waiting writers */
    int rw_writing;/* non-zero if write locked */
    int rw_readcnt;/* number of current readers */
```

```
    int rw_readwcnt;/* number of waiting readers */
    int rw_writewcnt;/* number of waiting writers */
};


/* rwlock.c */

/*
 * Initialize the read/write lock structure.
 */
void
rwl_init(struct rwl_lock *r)
{
    (void) pthread_mutex_init(&(r->rw_mutex),
        (pthread_mutexattr_t *)NULL);
    (void) pthread_cond_init(&(r->rw_readcond),
        (pthread_condattr_t *)NULL);
    (void) pthread_cond_init(&(r->rw_writecond),
        (pthread_condattr_t *)NULL);

    r->rw_readwcnt = 0;
    r->rw_writewcnt = 0;
    r->rw_writing = 0;
    r->rw_readcnt = 0;
}


/*
 * Read lock the rw lock.
 */
void
rwl_read(struct rwl_lock *r)
{
    (void) pthread_mutex_lock(&r->rw_mutex);

    while (r->rw_writing || r->rw_writewcnt) {
        /*
         * Wait if write locked or waiting writers
         */
        r->rw_readwcnt++;
        (void) pthread_cond_wait(&r->rw_readcond,
        &r->rw_mutex);
        r->rw_readwcnt--;
    }
    r->rw_readcnt++;

    (void) pthread_mutex_unlock(&r->rw_mutex);
}


/*
 * Write lock the rw lock.
 */
void
```

```
                  rwl_write(struct rwl_lock *r)
                  {
                      (void) pthread_mutex_lock(&r->rw_mutex);

                      while (r->rw_writing || r->rw_readcnt) {
                          /*
                           * Wait if already read or write locked.
                           */
                          r->rw_writewcnt++;
                          (void) pthread_cond_wait(&r->rw_writecond,
                          &r->rw_mutex);
                          r->rw_writewcnt--;
                      }
                      r->rw_writing = 1;

                      (void) pthread_mutex_unlock(&r->rw_mutex);
                  }


                  /*
                   * Unlock the rw lock.
                   */
                  void
                  rwl_unlock(struct rwl_lock *r)
                  {
                      (void) pthread_mutex_lock(&r->rw_mutex);

                      if (r->rw_writing) {
                          /* Writer unlock */
                          r->rw_writing = 0;

                          /* Wakeup any waiting writer, or
                           * wakeup any waiting reader if no waiting writers.
                           */
                          if (r->rw_writewcnt)
                              (void) pthread_cond_signal(&r->rw_writecond);
                          else if (r->rw_readwcnt)
                              (void) pthread_cond_broadcast(&r->rw_readcond);
                          }
                      else {
            /* Reader unlock. */
            r->rw_readcnt--;

            /* Wakeup any waiting writer. */
            if ((r->rw_readcnt == 0) && r->rw_writewcnt)
                (void) pthread_cond_signal(&r->rw_writecond);
    }
    (void) pthread_mutex_unlock(&r->rw_mutex);
}
```

Condition Variables

Condition variables are a general mechanism by which one thread can delay its execution until some condition is true and another thread can announce when some condition is true.

NOTE

The semantics of condition variables are analogous to those of the sleep-wakeup idiom historically used in kernel and device driver code.

The condition variable (of type `cond_t`) is part of the mechanism by which this synchronization occurs but that variable is not the condition itself. This condition is a somewhat abstract concept (as is resource) that is represented by other code in the program. Some hypothetical examples of conditions are:

- A message has arrived.
- Data is available for processing.
- Space is available to buffer output.

The association between condition and the condition variable arises from the programmer's usage of the feature.

One distinguishing feature of the conditional variable mechanism is that two different types of data structures are employed, not just one. A mutual exclusion lock (type `mutex_t`) *must* be used in concert with the condition variable (type `cond_t`) itself. By convention, a thread that evaluates, modifies, or acts on the condition must acquire the associated mutex lock beforehand and release that lock afterward.

The following pseudo-code shows the protocol for a thread that is making some condition true and announcing the change.

```
mutex_lock(&mutex);
make condition true;
cond_signal(&cond);  awaken thread (if any) waiting for condition
mutex_unlock(&mutex);
```

When the thread announces the change of the condition (to being true), it has a choice of awakening either a single thread waiting for that condition or all threads waiting for that condition. The syntax is:

```
int cond_signal(   cond_t *cond);  awaken one thread
int cond_broadcast(cond_t *cond);  awaken all threads
```

In either case, there is no problem if there are no waiting threads at the time of announcement.

NOTE

> Do not confuse the term “signal” in the sense of calling **`cond_signal(3synch)`** and “signal” in the sense of **`thr_kill(3thread)`**. They are different mechanisms with different semantics. (The latter provides asynchronous influence, the former does not.)

A thread wanting to delay itself until the condition is true must first acquire the associated mutex before evaluating the condition. If the condition is true, there is no need for delay and the thread can proceed; otherwise, the thread must call **`cond_wait(3synch)`** to wait for the condition to become true. The following pseudo-code illustrates the programming idiom.

```
mutex_lock(&mutex)
while(condition is false)
    cond_wait(&cond, &mutex);
act on the condition; possibly invalidate it
mutex_unlock(&mutex);
```

The mutex and condition variable used here must be the same data structures as those used in the places where the condition is made true.

If the condition is true when **`cond_wait`** is called, it returns immediately. If the condition is false, **`cond_wait`** will:

- Implicitly unlock the specified mutex.

  If the mutex remained locked (and the stated conventions were obeyed) no thread could enter the critical section to make the condition true.

- Block the calling thread until some other thread makes the condition true and announces that change with **`cond_signal(3synch)`** or **`cond_broadcast(3synch)`**.
- Implicitly re-acquire the specified mutex before returning.

  If this were not done, the thread could neither validly re-evaluate the condition (part of the `while` loop), nor validly act on the condition.

The semantics of condition variables require that a thread re-test the condition on any return from **`cond_wait(3synch)`**.

- The calling of either **`cond_signal(3synch)`** or **`cond_broadcast(3synch)`** implies that the condition was set true at some time.
- A thread waiting for that condition is made runnable, but there may be some delay until it actually executes and returns from **`cond_wait(3synch)`**.
- During that delay some other thread may be chosen to return from **`cond_wait(3synch)`** and invalidate the condition. This other thread might be:

  - Another thread in the “gang” awakened by **`cond_broadcast(3synch)`**.
  - A thread of higher priority that concurrently called **`cond_wait(3synch)`**.
- Consequently, for correctness every thread must re-test the condition on return from **`cond_wait(3synch)`**.

Other features of condition variables are:

- Blocked threads can be awakened by signals. The handler will be called and **`cond_wait(3synch)`** returns with an `EINTR` condition.
- There is a time-limited variant of `cond_wait` called **`cond_timedwait(3synch)`**.
- The separateness of the variable (type `cond_t`) used for signaling and for mutual exclusion (type `mutex_t`) means that several different conditions can be managed within one critical section.

POSIX Thread Condition Variables

The POSIX Thread interface provides condition variable functions almost identical to those provided by PowerMAX OS, with a few differences discussed below. All other POSIX condition variables work the same as PowerMAX OS condition variables.

POSIX condition variables support an attributes object used when initializing a condition variable. Therefore, the following functions exist to initialize and modify this attributes structure:

`int pthread_condattr_init(pthread_condattr_t *`*attr*`);`

Initializes the specified condition variable attributes structure to default values. This attributes structure can be used on a subsequent **`pthread_cond_init(3pthread)`** call. The only attribute currently defined in the condition variable attributes structure is the shared attribute. The shared attribute is initialized to `PTHREAD_PROCESS_PRIVATE` by this routine. The `PTHREAD_PROCESS_PRIVATE` value indicates that any condition variable initialized with this attributes structure can be used only by the threads within the process.

`int pthread_condattr_getpshared(const pthread_condattr_t *`*attr*`, int *`*pshared*`);`

`int pthread_condattr_setpshared(pthread_condattr_t *`*attr*`, int `*pshared*`);`

These routines get and set the shared attribute of the condition variable attributes object. The shared attribute can be set to a value of `PTHREAD_PROCESS_SHARED` if the condition variable nitialized with this attributes object is used between threads in different processes.

`int pthread_condattr_destroy(pthread_condattr_t *`*attr*`);`

Used to set a condition variable attributes object to a destroyed state.

int pthread_cond_init(pthread_cond_t ****cond*, **const pthread_condattr_t ****attr*);**

Initializes a POSIX condition variable. The *attr* parameter can be either specified or set to NULL. If NULL, then the condition variable has a shared attribute of `PTHREAD_PROCESS_PRIVATE`.

The other POSIX Thread condition variable functions so resemble PowerMAX OS condition variables functions that little additional discussion is needed. These functions include:

- **pthread_cond_wait(3pthread)**
- **pthread_cond_timedwait(3pthread)**
- **pthread_cond_signal(3pthread)**
- **pthread_cond_broadcast(3pthread)**
- **pthread_cond_destroy(3pthread)**

Only one minor difference between the condition variable wait functions needs mention. The PowerMAX OS functions, **cond_timedwait(3synch)** and **cond_wait(3synch)**, return a value of EINTR when they return prematurely due to a signal. The POSIX functions, **pthread_cond_timedwait(3pthread)** and **pthread_cond_wait(3pthread)**, return a value of 0 in the same situation. Since the semantics of condition variables require that a thread re-test the condition upon any return from a condition variable wait, this is an insignificant difference.

Semaphores

Semaphores are a facility well-suited to managing the allocation and deallocation of identical resources.

- The semaphore can be initialized to the number of resources.
- A thread needing a resource should atomically decrement the associated semaphore with the **sema_wait(3synch)** function.
- If the resource is not available (semaphore count non-positive) the caller will block in **sema_wait(3synch)** until one becomes available.

  `int` **sema_wait**`(sema_t *`*sema*`);`

- When a resource is no longer in use, the thread releasing the resource should increment the associated semaphore with the **sema_post(3synch)** function.

  `int` **sema_post**`(sema_t *`*sema*`);`

- If any threads are blocked on that semaphore, the call to `sema_post` will make one runnable so that it can (implicitly) decrement the semaphore and return from **sema_wait(3synch)**.

Additional considerations:

- This mechanism lacks the following features of the IPC style semaphore facility:

  - Increment/decrement by values greater than `1`.
  - Operations on semaphore sets.
  - The ability to, "block while count is non-zero" instead of the usual rule, "block only when count is zero."
  - The ability to automatically release semaphores on termination (`SEM_UNDO` flag).

- A semaphore initialized to `1` is almost equivalent to a mutex. In such cases, use the mutex facilities; they are more efficient, having been optimized for that case.

NOTE

If the design of the application calls for signal handlers to use the synchronization operations, then use semaphores, which are asynchronous-safe, and can be used to communicate between signal handlers and base level code. For example, a signal handler can safely call `sema_post`, but it should not try to lock or unlock a mutex used in the base code.

- The **`sema_post(3synch)`** function can validly be used to increase a semaphore count above that defined by **`sema_init(3synch)`**.

POSIX Thread Semaphores

The POSIX Thread interfaces within the thread library do not provide thread-specific support for POSIX semaphores. Instead, all POSIX compliant applications, whether they are single-threaded or POSIX multi-threaded applications, should make use of the POSIX semaphore interfaces: **`sem_wait(3)`**, **`sem_post(3)`**, etc.

Barriers

In a sense, a barrier is the logical inverse of a lock. Whereas a lock allows only one thread at a time to proceed (to use a resource), a barrier allows no thread to proceed until an entire group of them are ready to proceed.

- The number of threads expected to gather at a barrier is specified in the barrier structure when it is initialized with **`barrier_init(3synch)`**. (See "Initialization of Synchronization Mechanisms," page 11-74.)
- A thread declares its arrival at the barrier with the **`barrier_wait(3synch)`** function.

```
int barrier_wait(barrier_t *barrier);
```

- The barrier mechanism has no facility to authenticate the threads calling **`barrier_wait`**; it simply counts the arriving threads.

- If the number of threads at the barrier is less than the initialized value, the thread calling **barrier_wait** is suspended.
- If the arriving thread brings the count to the requisite value, all of the waiting threads are made runnable and eventually return from **barrier_wait**.
- When the threads are released, the count of threads at the barrier is reset to zero. That same barrier can be reused without re-initialization.
- A barrier should not be re-initialized while there are waiting threads.
- There is also a “spinning” variant of barriers called **_barrier_spin(3synch)**. The considerations for usage are similar to those for spin locks given above.

POSIX Thread Barriers

The POSIX Thread interfaces do not provide support for barriers. However, a set of barrier routines can be coded by using standard POSIX Thread constructs and routines. The example code shown below does not include error recovery, for clarity's sake:

```
/* pbarrier.h */

struct pbarrier {
    pthread_mutex_t pb_mutex;/* synchronization lock */
    pthread_cond_t pb_cond;/* for waiting */
    int pb_count;/* barrier count */
    int pb_generation;/* for determining wakeups */
    int pb_waiting;/* number of threads waiting */
};


/* pbarrier.c */

/*
 * Initialize the barrier structure.
 */
void
pbar_init(struct pbarrier *b, int count)
{
    (void) pthread_mutex_init(&b->pb_mutex,
        (pthread_mutexattr_t *)NULL);
    (void) pthread_cond_init(&b->pb_cond,
        (pthread_condattr_t *)NULL);
    b->pb_count = count;
    b->pb_generation = 0;
    b->pb_waiting = 0;
}

/*
* Barrier wait.
*/
void
```

```
pbar_wait(struct pbarrier *b)
{
    int my_generation;

    (void) pthread_mutex_lock(&b->pb_mutex);

    b->pb_waiting++;
    if (b->pb_waiting >= b->pb_count)
    {
        b->pb_waiting = 0;
        b->pb_generation++;
        (void) pthread_cond_broadcast(&b->pb_cond);
    }
    else
    {
        my_generation = b->pb_generation;
        while (my_generation == b->pb_generation)
                (void) pthread_cond_wait(&b->pb_cond,
                &b->pb_mutex);
    }
    (void) pthread_mutex_unlock(&b->pb_mutex);
}
```

Awakening Threads for Synchronization Mechanisms

When only one thread is to be awakened for a newly available synchronization mechanism, the selection is made by the following general rule.

a. Preference is given to bound threads over multiplexed threads.

b. If there is still more than one candidate for awakening, the thread with the highest (Threads Library) priority is chosen.

c. If there is still more than one candidate for awakening, the thread that blocked first is selected.

   FIFO ordering of threads of the same priority is generally true but not guaranteed. In this implementation, there are race conditions in which the ordering is not strictly FIFO.

There are some exceptions to this algorithm:

- For a broadcast on a condition variable and for barriers, more than one thread is awakened. Conceptually, these are awakened simultaneously.
- For reader-writer locks, the order of awakening is strictly FIFO, regardless of priority or other factors.

Further Considerations for Synchronization Mechanisms

- There is no protection against priority inversion. When a thread holds a lock, it keeps its priority even if a higher priority thread is waiting for that lock. Therefore, a low priority thread can prevent a thread of higher priority from running.
- **`thr_exit(3thread)`** and **`pthread_exit(3pthread)`** do not release any locks a thread might have acquired.
- There is no automatic protection from deadlock (except for the limited protection provided by recursive mutexes).
- If a caught signal is received by a thread while blocked on a synchronization mechanism (other than a condition variable):
  - The signal handler is called.
  - The blocked function call is transparently re-started.
  - The function does not return with `EINTR`.
  - Condition variables are the single exception. A call to **`cond_wait`** or **`cond_timedwait`** will be abnormally terminated on receipt of a signal, and `EINTR` will be returned. **`pthread_cond_wait`** and **`pthread_cond_timedwait`** will also be abnormally terminated on receipt of a signal, but will instead return 0.
- Each mechanism (except barriers) has a conditional `_try` variant that will not block when the resource is unavailable; error condition `EBUSY` is returned instead.

Initialization of Synchronization Mechanisms

PowerMAX OS Synchronization Mechanisms

Some general characteristics of the initialization functions are:

- The first argument is a pointer to the locking structure to be initialized.
- The *type* argument can take on the values of either:

`USYNC_THREAD`	thread-to-thread synchronization .
`USYNC_PROCESS`	interprocess synchronization. For such use, the synchronization data structures must reside in memory that is shared between the processes, using either IPC shared memory or the mapped file feature.

  The *type* argument is not available for the two spinning type locks.

- Two mechanism types (barriers and semaphore) require an initial value (*count*).

- The last argument is of type `(void *)`, is reserved for future use, and should be set to `NULL` for future compatibility.

The syntax of these functions is given below.

```
int sema_init(sema_t*sema,int count,int type,void *arg);
int barrier_init(barrier_t*barrier,int count,int type,void *arg);
int _barrier_spin_init(barrier_spin_t*barrier,int count,void *arg);
int _spin_init(spin_t*lock,void *arg);
int cond_init(cond_t*cond, int type,void *arg);
int mutex_init(mutex_t*mutex,int type,void *arg);
int rmutex_init(rmutex_t*rmutex,int type,void *arg);
int rwlock_init(rwlock_t*rwlock,int type,void *arg);
```

POSIX Initialization Mechanisms

POSIX thread mutexes and condition variables can both be initialized with or without a user-specified attributes structure.

Both the mutex and condition variable attributes structures have one attribute, the shared attribute. The value of this attribute may be set to:

```
PTHREAD_PROCESS_PRIVATE
        thread-to-thread synchronization
PTHREAD_PROCESS_SHARED
        interprocess synchronization.
```

When an attributes structure is not specified on a mutex or condition variable initialization call, then this attribute defaults to `PTHREAD_PROCESS_PRIVATE`.

The syntax of the mutex and condition variable initialization functions are shown below:

```
int pthread_condattr_init(pthread_condattr_t *attr);
int pthread_condattr_getpshared(const pthread_condattr_t
    *attr, int *pshared);
int pthread_condattr_setpshared(pthread_condattr_t *attr,
    int pshared);
int pthread_cond_init(pthread_cond_t *cond, const
    pthread_condattr_t *attr);
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_getpshared(const pthread_mutexattr_t
    *attr, int *pshared);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
    int pshared);
int pthread_mutex_init(pthread_mutex_t *mutex, const
    pthread_mutexattr_t *attr);
```

Alternative Initialization

In this implementation, it is valid to use statically initialized (zero-filled) data structures for the synchronization mechanisms.

For most of the mechanisms, a zero-filled data structure is taken to be unlocked and of type USYNC_THREAD. The mechanisms that take a count argument have the following additional interpretations:

- A zero-filled *sema_t* structure represents zero available resources. A **sema_wait(3synch)** on that structure will block.
- A zero-filled barrier_t structure is valid but meaningless.

This technique is *not* recommended for re-initialization of synchronization structures. In general, it is incorrect to re-initialize a synchronization structure while in use. Some of the initialization functions (shown above) return EBUSY if called for an active data structure (one on which threads are blocked). Zero-filling the data structure bypasses that check.

POSIX Static Initializations

Both pthread_mutex_t and pthread_cond_t structures can be statically initialized, when the default attribute values for these structures is appropriate. The PTHREAD_MUTEX_INITIALIZER and PTHREAD_COND_INITIALIZER macros may be used for this purpose:

- **pthread_mutex_t mutex** = PTHREAD_MUTEX_INITIALIZER;
- **pthread_cond_t cond** = PTHREAD_COND_INITIALIZER;

Note that dynamically zero-filling POSIX thread mutex or condition variables as a method for initializing these structures is NOT appropriate behavior for POSIX-compliant applications.

Invalidation of Synchronization Mechanisms

The syntax of the functions that invalidate synchronization structures is even more regular than that of the initializing functions.

- The first and only argument is a pointer to the mechanism-specific structure to be invalidated.

The syntax is:

```
int sema_destroy           (sema_t           *sema);
int _spin_destroy          (spin_t           *lock);
int barrier_destroy        (barrier_t        *barrier);
int _barrier_spin_destroy  (barrier_spin_t   *barrier);
int cond_destroy           (cond_t           *cond);
int mutex_destroy          (mutex_t          *mutex);
int rmutex_destroy         (rmutex_t         *rmutex);
int rwlock_destroy         (rwlock_t         *rwlock);
int pthread_cond_destroy   (pthread_cond_t   *cond);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Each function can fail as follows:

EINVAL Invalid argument specified.

`EBUSY`	Mechanism currently in use.

The effect of these functions is:

- To mark the structure as being invalid for further use (unless re-initialized).
- To allow the recovery of any Threads Library internal resources that may have been allocated when the synchronization mechanism was initialized.
- Though these **`_destroy`** functions recover underlying resources, the space for the synchronization structure itself remains. If the space is to be recovered (say the structure will no longer be used) that must be done separately. For example, a space acquired from **`malloc(3C)`** should be recovered with **`free(3C)`**.

Development Environment

Compilation Environment

Source code that uses Threads Library functions should include one of the following lines:

- PowerMAX OS: `#include <thread.h>`
- POSIX: `#include <pthread.h>`

and should be compiled and linked with the Threads Library. You may link this program either statically or dynamically. To compile and link it dynamically, use the following command-line options:

`cc` [*options*] **`-D_REENTRANT`** *file* **`-lthread`**

This will create a dynamically linked executable. For linking statically, see the **`hc(1)`** and **`ld(1)`** system manual pages and the “Link Editor and Linking” chapter in *Compilation Systems Volume 1 (Tools)*.

The **`-lthread`** flag links in the Threads Library. The **`-D_REENTRANT`** flag is needed for an application to be thread-safe and to access reentrant routines in standard libraries (see below).

Error Returns

None of the thread management or synchronization routines in the Threads Library set `errno` to indicate an error; most return an error number if an error is encountered.

- The error numbers returned correspond to `errno` numbers.
- This discourages use of `errno`, which is not reentrant and is inefficient in a multithreaded environment.
- The Threads Library does not guarantee preservation of `errno` across calls.

NOTE

The asynchronous I/O routines, which are included in the Threads Library, do set `errno`. See the aio system manual pages.

However, threads may call routines that do set `errno`. If all threads in a process accessed a global `errno`, no thread could be sure that the global value resulted from a system call it had made, it might have resulted from another system call made by another thread. Therefore, the Threads Library maintains a private copy of `errno` for each thread. When a thread references `errno`, it will get the value of its private copy, not the global variable.

There is one exception: the initial thread (the thread running **`main`**) accesses the global `errno` via its private copy. Therefore, the initial thread can safely call into non-reentrant code (such as an old object file compiled before PowerMAX OS), and have correct *errno* semantics. Threads other than the initial thread should not make calls into old object files that set `errno`. The mixing of reentrant and non-reentrant object files is discouraged, and should only be done as an interim measure until applications are made reentrant.

Applications that reference `errno` should include the following line:

```
#include <errno.h>
```

Thread-Safe Libraries

In previous releases of the system, libraries freely used global and static data. In a multithreaded program, different sibling threads running concurrently could corrupt global or static data. Therefore, in the PowerMAX OS, standard libraries have been made thread-safe. When an application is compiled with the `-D_REENTRANT` flag to **`cc(1)`**, standard libraries will synchronize threads' use of global and static data. (As this synchronization has a performance cost to single-threaded applications, it is only enabled when the `-D_REENTRANT` flag is used.)

In addition, new, reentrant versions of some library routines have been added. The names of these routines are suffixed with `_r`; for example, the reentrant version of **`strtok(3S)`** is **`strtok_r(3S)`**. Multithreaded applications should use the reentrant versions of library routines.

The PowerMAX OS supplies thread-safe versions of the following libraries:

- **`libc`**
- **`libm`**
- **`libnsl`**
- **`libsocket`**
- **`libresolv`**
- **`resolv`**
- **`tcpip`**
- **`libud`**

Applications using other libraries that have not been made thread-safe must synchronize access to global data.

System Call Wrappers

The Threads Library provides wrappers for the system calls and library routines listed below. A wrapper is a routine with the same name and interface as another routine—in this case, a standard system call or library routine. Wrappers usually do something to modify the behavior of the standard routine, then call the standard routine, and perhaps do something further when the standard routine returns. Many of the Threads Library wrappers cause the system call to affect a single thread instead of the entire process or LWP. See the on-line system manual pages for these routines for details.

When you compile with **`-D_REENTRANT`** and **`-lthread`**, references to these routines will automatically access the Threads Library wrapper versions.

Table 11-1. System Call Wrappers

`alarm(2)`	**`close(2)`**	**`creat(2)`**
`fcntl(2)`	**`fork(2)`**	**`forkall(2)`**
`fsync(2)`	**`getitimer(3C)`**	**`iconnect(3C)`**
`ienable(3C)`	**`msync(3C)`**	**`open(2)`**
`pause(2)`	**`posix_timers(2)`**	**`read(2)`**
`raise(3C)`	**`sched_yield(3C)`**	**`setcontext(2)`**
`setitimer(3C)`	**`sigaction(2)`**	**`sighold(2)`**
`sigignore(2)`	**`signal(2)`**	**`sigpause(2)`**
`sigpending(2)`	**`sigprocmask(2)`**	**`sigrelse(2)`**
`sigset(2)`	**`sigsuspend(2)`**	**`sigwait(2)`**
`sigtimedwait(2)`	**`sigwaitinfo(2)`**	**`sleep(2)`**
`tcdrain(2)`	**`timer_create(3C)`**	**`timer_delete(3C)`**
`wait(2)`	**`waitpid(2)`**	**`write(2)`**

Timers

The Threads Library provides facilities that allow multiplexed threads to use alarms and real interval timers without requiring that the threads tie up LWPs between the initiation and expiration of the call. For this purpose, the Threads Library supplies wrappers for **`alarm(2)`**, **`getitimer(3C)`**, **`setitimer(3C)`**, and **`sleep(2)`**. When a bound thread calls one of these routines, it has access to the full functionality as described in the system manual page. However, when a multiplexed thread calls one of these functions, it will use the Threads Library version of the function, and, in some cases, the functionality will be limited; for example, a multiplexed thread can use only real timers, not virtual or profiling timers.

In addition, the wrapper versions of these functions have per-thread semantics rather than per-LWP semantics. For example, **`alarm(2)`** sets an alarm clock. When the set time expires, the caller receives a `SIGALRM` signal. The wrapper function ensures that the `SIGALRM` is delivered to the calling thread (rather than the calling LWP), regardless of whether the calling thread is running on the same LWP on which it was running when it issued the call to `alarm`.

See the system manual pages for these routines for details.

POSIX Timers

The Threads Library provides thread synchronization for the per-process POSIX timers. As a result, multiplexed threads may create, set, and delete timers that are shared among all threads within the process. For this purpose, the Threads Library supplies wrappers for **`timer_create(3C)`** and **`timer_delete(3C)`**.

In addition, the wrapper versions of these routines provide the **SIGEV_CALLBACK** timer expiration notification mechanism. This mechanism is a Concurrent Computer Corporation extension to the POSIX specification. It is provided as a higher performance, more deterministic method for providing notification of timer expiration.

User-Level Interrupts

The Threads Library provides wrapper versions of the **`iconnect(3C)`** and **`ienable(3C)`** routines. In addition to providing thread synchronization among the multiplexed threads within a process, these wrappers provide the ability to connect to more than one external interrupt within the same process.

Examples

This section presents several small programs to illustrate use of the Threads Library.

Hello, world

Figure 11-2 shows the traditional first program written when one enters a new regime of the UNIX programming environment. In this example, one thread is created to output "`hello`" and a separate thread to output "`world.\n`" Despite its brevity, this example illustrates several points about programming with the Threads Library:

- The argument types and the return type of the **`printf(3S)`** function disqualify it as the starting point of a thread. A "wrapper" function (**`print`**) had to be devised.
- There is no need for the initial thread to wait for the completion of the two threads running **`print`**. The process is automatically terminated after both of the printing threads complete.

NOTE

The use of **`thr_exit(3thread)`** is important. The use of `return` from **`main`** or allowing **`main`** to run off the closing brace is translated to a call to the **`exit(2)`** system call. That system call generally terminates the process before the printing threads can produce their output.

The order of the output is not guaranteed. In most cases the thread that is created first will be able to output “`hello,`” before the following thread outputs “`world.\n`” Occasionally, the order is reversed.

```
#include<stdio.h>
#include<stdlib.h>
#include<thread.h>
static void*print(void*);
int main()
{
    int okend = EXIT_SUCCESS;
    (void)thr_create(0,0, print, (void *)“hello, “,  0L,0);
    (void)thr_create(0,0, print, (void *)“world.\n”, 0L,0);
    thr_exit(&okend);
    /*NOTREACHED*/
}
static void *print(void *s)
{
    (void)printf(s);
    return NULL;
}
```

Figure 11-2. Hello, World

Basic Threads Management

The following examples on threads management will use (either explicitly or implicitly) the function **`sometask`** that appears in Figure 11-3. This function will call **`sleep(3C)`** to represent some arbitrary activity by the thread. Features to note in this example are:

- The simulated action for each thread will be different since each sleeps for a different, random period of time. The seed for the random number generator depends on current process ID, current time, and thread ID.
- The activity period is at least one second plus a random component between zero and nine seconds.
- The random number is generated with **`rand_r(3C),`** the thread-safe version of **`rand(3C)`**.

Calls to **`sleep(3C)`** by each thread will put only the calling thread to sleep, as arranged by the wrapper version of **`sleep`** provided by the Threads Library.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <thread.h>
#define RANGE10
/* ARGSUSED */

void *sometask(void *dummy)
{
    thread_t thrID= thr_self();
    unsigned seed= getpid() * time(NULL) * (thrID + 1);
    unsigned naptime=
        (unsigned)(1 + RANGE*((double)rand_r(&seed)/(double)RAND_MAX));
    setbuf(stdout,NULL);
    (void)printf(“thread %ld entering sometask\n”,thrID);
    (void)printf(“thread %ld naptime %d\n”,thrID, naptime);
    (void)sleep(naptime);
    (void)printf(“thread %ld leaving  sometask\n”,thrID);
    return NULL;
}
```

Figure 11-3. Sometask

The program in Figure 11-4 creates one or more threads as follows

- The number of threads to be created is determined by a (validated) command line parameter.
- Each new thread runs **`sometask`**.

The initial thread waits for the termination of each thread that it creates.

```
#include<stdio.h>
#include<stdlib.h>
#include<thread.h>
extern void*sometask(void *);

main(int argc, char **argv)
{
     int Nthreads, i; thread_t threadID;
     if(argc != 2){
    (void)fprintf( stderr,”%s: usage: %s Nthreads\nwhere Nthreads > 0\n”,
            argv[0], argv[0]);
    return 1;
     }
     if( (Nthreads = atoi(argv[1])) <= 0 ){
    (void)fprintf( stderr,”%s: usage: %s Nthreads\nwhere Nthreads > 0\n”,
            argv[0], argv[0]);
    return 1;
     }
     for(i = 0; i < Nthreads; i++)
    (void)thr_create(NULL, 0, sometask, NULL, 0, NULL);
     for(i = 0; i < Nthreads; i++){
    (void)thr_join(0, &threadID, NULL);
    (void)printf(“thread %ld is gone\n”, threadID);
     }
     return 0;
}
```

Figure 11-4. Multiple Threads

The example in Figure 11-5 is a variation of that in Figure 11-4. In this case:

- Each thread that is created (running **`repeatask`**) calls **`sometask`** repeatedly.
- The created threads coordinate their activity into cycles by the barrier facility of the Threads Library.
- Output of this program shows a flurry of activity as the barrier count is reached and the set of threads is unleashed for the next cycle.

There is no need for the initial thread to persist; consequently, the initial thread terminates itself with **`thr_exit(3thread)`**. The process continues until the user terminates it manually.

```
#include<stdio.h>
#include<stdlib.h>
#include<thread.h>
#include<synch.h>
extern void*sometask (void *);
static void*repeatask(void *);
static barrier_tcommon_wall;

main(int argc, char **argv)
{
     int Nthreads, i;
     if(argc != 2){
    (void)fprintf(stderr,"%s: usage: %s Nthreads\nwhere Nthreads > 0\n",
            argv[0], argv[0]);
    return 1;
     }
     if((Nthreads = atoi(argv[1]))>0){
    (void)barrier_init(&common_wall, Nthreads, USYNC_THREAD, NULL);
     } else {
    (void)fprintf(stderr,"%s: usage: %s Nthreads\nwhere Nthreads > 0\n",
            argv[0], argv[0]);
    return 1;
     }
     for(i = 0; i < Nthreads; i++)
    (void)thr_create(NULL, 0, repeatask, NULL, 0, NULL);
     thr_exit(NULL);
     /*NOTREACHED*/
}
/* ARGSUSED */
static void *repeatask(void *dummy)
{
    for(;;){
        (void)printf("thread %ld at wall\n", thr_self());
        (void)barrier_wait(&common_wall);
        (void)sometask(NULL);
    }
}
```

Figure 11-5. Barrier_wait

Dining Philosophers

The program in Figure 11-6 shows an implementation of the classic "dining philosophers" problem using the facilities of the Threads Library.

In this problem there are *N* philosophers sitting at a round table eating and thinking. Each has a plate of spaghetti and there is a single fork (a total of *N* forks) between each pair. Each philosopher must use two forks to eat the spaghetti. Each philosopher puts down both forks to think. This simple problem illustrates many of the issues in concurrent programming, such as the need for synchronization to prevent deadlock. The philosophers represent processes that require shared resources (the forks).

Some features of this program are:

- Each philosopher is represented by a thread.
- Each fork is represented by a semaphore.

```
#include<stdio.h>
#include<thread.h>
#include<synch.h>
#define NPHIL5
static sema_t    forks[NPHIL];
typedef struct {
    int id, left_fork, right_fork;
} philo_t;
static philo_t   philo_args[NPHIL];
static void  *philo(void*);
extern void  *sometask(void*);
main()
{
    int i;
    for(i = 0; i < NPHIL; i++){
        (void)sema_init(&forks[i], 1, USYNC_THREAD, NULL);
        philo_args[i].id= i;
        philo_args[i].left_fork= i;
        philo_args[i].rght_fork= (i+1)%NPHIL;
    }
    for(i = 0; i < NPHIL; i++)
        (void)thr_create(NULL, 0, philo, &philo_args[i], 0, NULL);
    thr_exit(NULL);
    /*NOTREACHED*/
}
static void *philo(void *philo_arg)
{
    philo_t*argp= (philo_t *)philo_arg;
    int id   = argp->id;
    int left= argp->left_fork;
    int rght= argp->rght_fork;
    (void)printf(“thrID %ld id %d left %d rght %d\n”,
            thr_self(), id, left, rght);
    for(;;){
        (void)sema_wait(&forks[left]);
        (void)sema_wait(&forks[rght]);
        (void)printf(“philo %d eating w. %d and %d\n”, id, left, rght);
        (void)sometask(NULL); /* eating */
        (void)printf(“philo %d done   w. %d and %d\n”, id, left, rght);
        (void)sema_post(&forks[left]);
        (void)sema_post(&forks[rght]);
        (void)sometask(NULL); /* think */
    } /* NOTREACHED */
}
```

Figure 11-6. Dining Philosophers

- Each thread runs the same code (**`philo`**) but needs different arguments. That is each philosopher being simulated will follow the same rules but is assigned to use a distinct pair of forks. The example shows how to assemble several items of information into a structure and inform the thread of where to find its arguments.

- This implementation can deadlock. If each philosopher picks up his or her left fork before any picks up his or her right fork, they will deadlock, waiting to pick up their right forks. One way to solve this deadlock would be to allow no more than *N*-1 philosophers to eat at the same time. That way, one of the philosophers will always be able to pick up two forks.
- As in other examples, there is no need for the initial thread to persist. The simulation continues until manually terminated.

Producer/Consumer

The program in Figure 11-7 shows a simple producer/consumer example implemented using the condition variables facilities of the Threads Library.

- There are two threads, each running different functions. One runs **`producer`**, the other runs **`consumer`**.
- There are two threads, each running different functions. One runs **`producer`**, the other runs **`consumer`**.
- The item being produced and consumed is data in a common buffer.
  - The producer obtains that data with **`fgets(3C)`** and places the data in the common buffer.
  - The consumer reads the data from the buffer. That data is output with **`fputs(3S)`** so that its actions can be confirmed.
- The actions of producer and consumer threads are coordinated by the condition variable facility so that they run in strict alternation.
  - Nothing will be output (consumed) until something is placed in the buffer (produced).
  - Data in the buffer will not be overwritten until it is output.
- Note that this example differs from the pseudo-code shown earlier. The use of the condition variables are not bracketed by calls to **`mutex_lock(3thread)`** and **`mutex_unlock(3thread)`**.
  - This curiosity arises because the actions of each of these threads is organized in a loop.
  - The semantics of **`cond_wait(3synch)`** guarantee that the named mutex will be released while a thread is waiting and reacquired before return from that function.
  - The condition (`DataInBuff`) is tested (for different values) by each thread only when the thread holds the mutex and the use of **`cond_wait`** by each threads allows the other to acquire the mutex.
- The initial calls to **`mutex_lock(3thread)`** by each thread and the initial state of `DataInBuff` are organized so that:
  - Proper conditions are achieved for the initial pass by each thread.

.

```
#include        <stdio.h>
#include        <stdlib.h>
#include        <string.h>
#include        <thread.h>
#include        <synch.h>
#define TRUE    1
#define FALSE   0
static  void    *producer(void*);
static  void    *consumer(void*);
static  char    Buff[BUFSIZ];
static  cond_t  Buff_cond;
static  mutex_t Buff_mutex;
static  int     DataInBuff = FALSE;
main()
{
        (void)mutex_init(&Buff_mutex, USYNC_THREAD,  NULL);
        (void)cond_init (&Buff_cond,  USYNC_THREAD,  NULL);
        (void)thr_create(NULL, 0, producer, NULL, 0, NULL);
        (void)thr_create(NULL, 0, consumer, NULL, 0, NULL);
        thr_exit(NULL);
        /*NOTREACHED*/
}
/*ARGSUSED*/
static void *producer(void *dummy)
{
    (void)mutex_lock(&Buff_mutex);
        for(;;){
                while(DataInBuff == TRUE)
                        cond_wait(&Buff_cond, &Buff_mutex);
         /* At this point,
          * the buffer is empty (contents have been output).
          * (Re)fill the buffer.
          */
                if(fgets(Buff, sizeof(Buff), stdin) == NULL)
                        exit(EXIT_SUCCESS);
                DataInBuff = TRUE;
                cond_signal (&Buff_cond);
        }
        /*NOTREACHED*/
}
/*ARGSUSED*/
static void *consumer(void *dummy)
{
        (void)mutex_lock(&Buff_mutex);
        for(;;){
                while(DataInBuff == FALSE)
                        cond_wait(&Buff_cond, &Buff_mutex);
         /* At this point,
          * the buffer has data to be output
          */
                (void)fputs(Buff, stdout);
                DataInBuff = FALSE;
                cond_signal(&Buff_cond);
        }
        /*NOTREACHED*/
}
```

Figure 11-7. Producer/Consumer

- The program will work correctly no matter which thread acquires the mutex on the first pass.

• The producer thread terminates the process with the **`exit(2)`** system call when it can obtain no more data from **`fgets(3S)`**.

12
Interprocess Communication

12
Interprocess Communication

Introduction

The OS provides several mechanisms that allow processes to exchange data. These mechanisms include the following: POSIX shared memory facilities that are based on IEEE Standard 1003.1b-1993, the System V Interprocess Communication (IPC) package, and such simple mechanisms as pipes, named pipes, and signals.

"POSIX Shared Memory" describes the POSIX shared memory facilities and shows how they can be used by cooperating processes.

The System V IPC package consists of three mechanisms for interprocess communication: messages, semaphores, and shared memory. "System V IPC Package" (p. 12-6) provides an overview of these mechanisms and the facilities for using them. "Security Enhancements for IPC Objects" (p. 12-7) describes the enhanced Discretionary Access Control (DAC) and Mandatory Access Control (MAC) that are provided if the Enhanced Security Utilities are installed on your system.

Pipes, named pipes, and signals are the simplest mechanisms for interprocess communication, but they are limited in the following ways:

- Pipes do not allow unrelated processes to communicate.
- Named pipes allow unrelated processes to communicate, but they cannot provide private channels for pairs of communicating processes; that is, any process with appropriate permission may read from or write to a named pipe.
- Sending signals, via the **`kill`** system call, allows arbitrary processes to communicate, but the message consists only of the signal number and one word of data.

Chapter 10, "Signals, Job Control, and Pipes," describes these mechanisms and explains the procedures for using them.

Shared Memory Alternatives

Although you may create areas of shared memory by using either the POSIX shared memory interfaces or the System V IPC shared memory interfaces, the two types of interfaces are quite different syntactically. The procedures that cooperating processes must follow in order to use the interfaces to share data are also different. (For additional details on syntax and procedures, refer to "POSIX Shared Memory," p.12-2, and "System V Shared Memory," p. 12-53, respectively.) The following recommendations may assist you in determining whether to use the POSIX interfaces or the System V interfaces in your application:

- It is recommended that you use a System V shared memory area in an application in which data placed in shared memory are temporary and do not need to exist following a reboot of the system. Data in a System V shared memory area are kept only in memory. No disk file is associated with that memory. Data in a System V shared memory area will not contribute to the disk traffic that is generated by the **`sync(2)`** system call. Note that disk traffic generated by **`sync`** can cause significant system overhead.

  If the POSIX shared memory interfaces are used to share data, those data are mapped to a disk file in the **`/var/tmp`** directory. If this directory is mounted on a **`memfs`** file system, then no extra disk traffic is generated to flush the shared data during the **`sync`** system call. If this directory is mounted on a regular disk partition, then disk traffic will be generated during the **`sync`** system call to keep the shared data updated in the mapped disk file. Whether the data that are written to POSIX shared memory are saved in a file or not, those data do not persist following a reboot of the system. For information on the **`memfs`** file system type, refer to *System Administration Volume 2.*

- If a shared memory area is to be associated with a section of physical memory, it is recommended that you use System V shared memory and the facilities for binding an area of shared memory to a section of physical memory (see "Using shmget and shmbind," p.12-73, and "Using shmconfig," p.12-84).

  An alternative to using System V shared memory is to use the **`mmap(2)`** system call to map a portion of the **`/dev/mem`** file. For information on the **`mmap`** system call, refer to Chapter 6, "Memory Management." For information on the **`/dev/mem`** file, refer to the **`mem(7)`** system manual page.

- When the user desires a shared memory area to be placed in local memory, the System V shared memory interfaces are more useful. With the System V shared memory interfaces, the NUMA policy is forced to be the same for all processes that attach to the shared memory segment. It is also possible to create a process that accesses the shared memory segment as a foreign local memory reference. Each process that attaches to a POSIX shared memory object can select a different NUMA policy. This means that the shared memory object could be migrated to global memory when a process that is not running on the processor board where the shared memory is currently residing attaches to the shared memory object.

POSIX Shared Memory

The POSIX shared memory interfaces allow cooperating processes to share data and more efficiently communicate through the use of a shared memory object. A *shared memory object* is defined as a named region of storage that is independent of the file system and can be mapped to the address space of one or more processes to allow them to share the associated memory.

The interfaces are briefly described as follows:

`shm_open`	create a shared memory object and establish a connection between the shared memory object and a file descriptor
`shm_unlink`	remove the name of a shared memory object

Procedures for using the **`shm_open`** routine are presented in "Using the shm_open Routine." Procedures for using the **`shm_unlink`** routine are presented in "Using the shm_unlink Routine."

In order for cooperating processes to use these interfaces to share data, one process completes the following steps. Note that the order in which the steps are presented is typical, but it is not the only order that you can use.

STEP 1:	Create a shared memory object and establish a connection between that object and a file descriptor by invoking the **`shm_open`** library routine, specifying a unique name, and setting the **O_RDWR** bit to open the shared memory object for reading and writing.
STEP 2:	Set the size of the shared memory object by invoking the **`ftruncate(2)`** system call and specifying the file descriptor obtained in Step 1. This system call requires that the memory object be open for writing. For additional information on **`ftruncate(2)`**, refer to the corresponding system manual page.
STEP 3:	Map a portion of the process's virtual address space to the shared memory object by invoking the **`mmap(2)`** system call and specifying the file descriptor obtained in Step 1 (see Chapter 6, "Memory Management," for an explanation of this system call).

To use the shared memory object, any other cooperating process completes the following steps. Note that the order in which the steps are presented is typical, but it is not the only order that you can use.

STEP 1:	Establish a connection between the shared memory object created by the first process and a file descriptor by invoking the **`shm_open`** library routine and specifying the same name that was used to create the object.
STEP 2:	If the size of the shared memory object is not known, obtain the size of the shared memory object by invoking the **`fstat(2)`** system call and specifying the file descriptor obtained in Step 1 and a pointer to a **`stat`** structure (this structure is defined in <**`sys/stat.h`**>). The size of the object is returned in the **`st_size`** field of the **`stat`** structure. Access permissions associated with the object are returned in the **`st_modes`** field. For additional information on **`fstat(2)`**, refer to the corresponding system manual page.

STEP 3:	Map a portion of the process's virtual address space to the shared memory object by invoking **mmap** and specifying the file descriptor obtained in Step 1 (see Chapter 6, "Memory Management," for an explanation of this system call).

Using the shm_open Routine

The **shm_open(3C)** routine allows the calling process to create a POSIX shared memory object and establish a connection between that object and a file descriptor. A process subsequently uses the file descriptor that is returned by **shm_open** to refer to the shared memory object on calls to **ftruncate(2)**, **fstat(2)**, and **mmap(2)**. After a process creates a shared memory object, other processes can establish a connection between the shared memory object and a file descriptor by invoking **shm_open** and specifying the same name.

After a shared memory object is created, all data in the shared memory object remain until every process removes the mapping between its address space and the shared memory object by invoking **munmap(2)**, **exec(2)**, or **exit(2)** and one process removes the name of the shared memory object by invoking **shm_unlink(3C)** (see Chapter 6, "Memory Management," and "Using the shm_unlink Routine," respectively, for explanations of these routines). Neither the shared memory object nor its name is valid after your system is rebooted.

The specifications required for making the **shm_open** call are as follows:

```
#include <sys/mman.h>

int shm_open(name, oflag, mode)

char *name;
int oflag;
mode_t mode;
```

The arguments are defined as follows:

name a pointer to a null–terminated string that specifies the name of the shared memory object. Note that this string may contain a maximum of 255 characters. It may contain a leading slash (/) character, but it may not contain embedded slash characters. Note that this name is not a part of the file system; neither a leading slash character nor the current working directory affects interpretation of it (**/shared_obj** and **shared_obj** are interpreted as the same name). If you wish to write code that can be ported to any system that supports POSIX interfaces, however, it is recommended that *name* begin with a slash character.

oflag an integer value that sets one or more of the following bits:

Note that **O_RDONLY** and **O_RDWR** are mutually exclusive bits; one of them must be set.

O_RDONLY causes the shared memory object to be opened for reading only

O_RDWR causes the shared memory object to be opened for reading and writing. Note that the process that creates the shared memory object must open it for writing in order to be able to set its size by invoking **`ftruncate(2)`**.

O_CREAT causes the shared memory object specified by *name* to be created if it does not exist. The memory object's user ID is set to the effective user ID of the calling process; its group ID is set to the effective group ID of the calling process; and its permission bits are set as specified by the *mode* argument.

If the shared memory object specified by *name* exists, setting **O_CREAT** has no effect except as noted for **O_EXCL**.

O_EXCL causes **`shm_open`** to fail if **O_CREAT** is set and the shared memory object specified by *name* exists. If **O_CREAT** is not set, this bit is ignored.

O_TRUNC causes the length of the shared memory object specified by *name* to be truncated to zero if the object exists and has been opened for reading and writing. The owner and the mode of the specified shared memory object are unchanged.

mode an integer value that sets the permission bits of the shared memory object specified by *name* with the following exception: bits set in the process's file mode creation mask are cleared in the shared memory object's mode (refer to the **`umask(2)`** and **`chmod(2)`** system manual pages for additional information). If bits other than the permission bits are set in *mode*, they are ignored. A process specifies the *mode* argument only when it is creating a shared memory object.

If the call is successful, **`shm_open`** creates a shared memory object of size zero and returns a file descriptor that is the lowest file descriptor not open for the calling process. The **FD_CLOEXEC** file descriptor flag is set for the new file descriptor; this flag indicates that the file descriptor identifying the shared memory object will be closed upon execution of the **`exec(2)`** system call (refer to the **`fcntl(2)`** system manual page for additional information).

A return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`shm_open(3C)`** system manual page for a listing of the types of errors that may occur.

Using the shm_unlink Routine

The **`shm_unlink(3C)`** routine allows the calling process to remove the name of a shared memory object. If one or more processes have a portion of their address space mapped to the shared memory object at the time of the call, the name is removed before **`shm_unlink`** returns, but data in the shared memory object are not removed until the last process removes its mapping to the object. The mapping is removed if a process invokes **`munmap(2)`**, **`exec(2)`**, or **`exit(2)`**.

The specifications required for making the **`shm_unlink`** call are as follows:

```
#include <sys/mman.h>

int shm_unlink(name)

char *name;
```

The argument is defined as follows:

name	a pointer to a null–terminated string that specifies the shared memory object name that is to be removed. Note that this string may contain a maximum of 255 characters. It may contain a leading slash (/) character, but it may not contain embedded slash characters. Note that this name is not a part of the file system; neither a leading slash character nor the current working directory affects interpretation of it (**`/shared_obj`** and **`shared_obj`** are interpreted as the same name). If you wish to write code that can be ported to any system that supports POSIX interfaces, however, it is recommended that *name* begin with a slash character.

A return value of **`0`** indicates that the call to **`shm_unlink`** has been successful. A return value of **–1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`shm_unlink(3C)`** system manual page for a listing of the types of errors that may occur. If an error occurs, the call to **`shm_unlink`** does not change the named shared memory object.

System V IPC Package

The mechanisms contained in the System V IPC package are described as follows:

- Messages allow processes to send formatted data streams to arbitrary processes.
- Semaphores allow processes to synchronize execution.
- Shared memory allows processes to share parts of their virtual address space.

When implemented as a unit, these three mechanisms share such common properties as the following:

- Each mechanism contains a “get” system call to create a new entry or retrieve an existing one.
- Each mechanism contains a “control” system call to query the status of an entry, to set status information, or to remove the entry from the system.
- Each mechanism contains an “operations” system call to perform various operations on an entry.

This chapter describes the system calls for each of these three forms of IPC. “System V Messages” (p.12-9) describes those for creating message queues and sending and receiving messages. “System V Semaphores” (p.12-31) describes those for creating semaphores and performing semaphore operations. “System V Shared Memory” (p.12-53) describes the system calls and utilities for creating shared memory segments, attaching them to and detaching them from a process’s virtual address space, and binding them to a section of physical memory.

This information is for programmers who write multiprocess applications. These programmers should have a general understanding of what semaphores are and how they are used.

Information from other sources would also be helpful. See the manual pages **`ipcs(1)`** and **`ipcrm(1)`** and the following manual pages*:*

```
intro(2)    aclipc(2)   lvlipc(2)
msgget(2)   msgctl(2)   msgop(2)
semget(2)   semctl(2)   semop(2)
shmget(2)   shmctl(2)   shmop(2)
```

Included in this chapter are several example programs that show the use of these System V IPC system calls. Since there are many ways to accomplish the same task or requirement, keep in mind that the example programs were written for clarity and not for program efficiency. Usually, system calls are embedded within a larger user-written program that makes use of a particular function provided by the calls.

Security Enhancements for IPC Objects

The Enhanced Security Utilities, if installed, enhance the UNIX system's ability to control access to a message queue, semaphore, or shared memory segment, by providing enhanced Discretionary Access Control (DAC) and Mandatory Access Control (MAC). Enhanced Discretionary Access Control allows the owner or creator of an IPC object to control access to it on a single-user basis through the use of Access Control Lists. Mandatory Access Control further restricts access to IPC objects through the security levels assigned to each process and IPC object on the system.

Discretionary Access Control

The IPC data structure for each specific type of IPC contains the `ipc_perm` data structure, around which is based a simple, easy-to-use, but somewhat limited DAC mechanism. The system call to manipulate this structure varies slightly for each type of IPC object, and is described separately for each type.

For cases where greater control is needed a more sophisticated mechanism, based on an Access Control List (ACL) is provided. Using an ACL, you can specify exactly what access (read, write, or both) is to be granted to specific users or groups.

The following discussion describes how to manipulate the ACL information for an IPC object. The system uses the information in the ACL, together with the information in the `ipc_perm` structure, to determine access to the IPC object. The evaluation that the system makes to determine access to IPC objects is nearly identical to that used to determine access to a file.

The ACL for an IPC object is controlled with the **`aclipc`** system call, which can perform three tasks:

- Get the ACL information for an IPC object (`GETACL`).
- Set the ACL information of an IPC object, replacing any existing ACL (`SETACL`).
- Get the number of ACL entries in the ACL of an IPC object (`GETACLCNT`).

The system call must be passed the type of IPC object being manipulated and the identifier of the specific object, together with the ACL itself.

The synopsis found in the **`aclipc(2)`** system manual page is as follows:

```
#include <sys/types.h>
#include <acl.h>

int aclipc(int type, int id, int cmd, int nentries,
            struct acl *aclbufp);
```

The `acl` structure is defined as:

```
struct acl {
        int     a_type;
        uid_t   a_id;
        ushort  a_perm;
}
```

The `type` variable must be one of `IPC_SHM`, `IPC_SEM`, or `IPC_MSG`; and *id* must be a valid identifier of an IPC object of the specified type.

The value of *cmd* must be one of the following:

`GETACL`	The ACL information for the IPC object specified by *type* and *id* is copied into the user-supplied `aclbufp`. The value of *nentries* specifies the number of ACL entries which will fit into `aclbufp`.The user must have read access to the IPC object
`SETACL`	The ACL for the IPC object specified by *type* and *id* is set to the ACL entries in the user-supplied buffer `aclbufp`. The value of *nentries* specifies the number of ACL entries in aclbufp. The contents of `aclbufp` must be a valid ACL. For a description of what makes up a valid ACL, see the chapter “Directory and File Management” of this guide. Only the owner or creator of an IPC object can set the ACL.
`GETACLCNT`	Returns the number of ACL entries for the IPC object specified by *type* and *id*. The values of *nentries* and *aclbufp* are ignored. The user must have read access to the IPC object.

A process with the appropriate privileges can override DAC restrictions. The `P_DACWRITE` privilege is used to override DAC for write operations; `P_DACREAD` overrides DAC for read operations. Some IPC read operations modify the object being read, and therefore require both privileges in order to override DAC restrictions. The `P_OWNER` privilege overrides the restriction on setting an IPC object's ACL.

Mandatory Access Control

Mandatory access control (MAC) is based on the security levels of subjects and objects. The MAC policy for IPC is the same as the policy for files and directories. A process's level must dominate that of an IPC object for read operations, and must equal the object's level for write operations.

In some cases reading an IPC object modifies the object. For example, when a process reads a message queue, the message is removed from the queue. Such operations require MAC write permission; in other words, the process must be at the same level as the IPC object.

An IPC object (message queue, semaphore, or shared memory segment) inherits its security level from the creating process. The security level of the IPC object cannot be changed.

A process with the appropriate privileges can override MAC restrictions. The `P_MACWRITE` privilege is used to override MAC for write operations; `P_MACREAD` overrides MAC for read operations. Both `P_MACWRITE` and `P_MACREAD` are needed to override MAC for read operations that alter IPC objects or to change the attributes of an IPC object existing at a different security level.

The examples shown throughout this chapter assume that the process does not possess any privileges. Unless a process needs information located at different security levels, a process may not notice any changes to the IPC mechanism. Minor differences are noted in the following sections.

System V Messages

The message type of IPC allows processes (executing programs) to communicate through the exchange of data stored in buffers. This data is transmitted between processes in discrete portions called messages. Processes using this type of IPC can send and receive messages.

If the Enhanced Security Utilities are installed and running, a process must be at the same security level as the message queue to send or receive messages. Thus, communication between processes must occur at identical security levels

Before a process can send or receive a message, it must have the UNIX operating system generate the necessary software mechanisms to handle these operations. A process does this using the **`msgget`** system call. In doing this, the process becomes the owner/creator of a message queue and specifies the initial operation permissions for all processes, including itself. Subsequently, the owner/creator can relinquish ownership or change the operation permissions using the **`msgctl`** system call. However, the creator remains the creator as long as the facility exists. Other processes with permission can use **`msgctl`** to perform various other control functions.

Processes which have permission and are attempting to send or receive a message can suspend execution if they are unsuccessful at performing their operation. That is, a process which is attempting to send a message can wait until it becomes possible to post the message to the specified message queue; the receiving process isn't involved (except indirectly, for example, if the consumer isn't consuming, the queue space will eventually be exhausted) and vice versa. A process which specifies that execution is to be suspended is performing a "blocking message operation." A process which does not allow its execution to be suspended is performing a "nonblocking message operation."

A process performing a blocking message operation can be suspended until one of three conditions occurs:

- It is successful.
- It receives a signal.
- The message queue is removed from the system.

System calls make these message capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns applicable information. Otherwise, a known error code (`-1`) is returned to the process, and an external error number variable, `errno`, is set accordingly.

Using Messages

Before a message can be sent or received, a uniquely identified message queue and data structure must be created. The unique identifier is called the message queue identifier (`msqid`); it is used to identify or refer to the associated message queue and data structure. This identifier is accessible by any process in the system, subject to normal access restrictions.

The message queue is used to store (header) information about each message being sent or received. This information, which is for internal use by the system, includes the following for each message:

- pointer to the next message on queue
- message type
- message text size
- message text address

There is one associated data structure for the uniquely identified message queue. This data structure contains the following information related to the message queue:

- operation permissions data (operation permission structure)
- pointer to first message on the queue
- pointer to last message on the queue

- current number of bytes on the queue
- number of messages on the queue
- maximum number of bytes on the queue
- process identification (PID) of last message sender
- PID of last message receiver
- last message send time
- last message receive time
- last change time

NOTE

> All C header files discussed in this chapter are located in the **`/usr/include`** or **`/usr/include/sys`** directories.

The definition for the associated message-queue data structure `msqid_ds` includes the following members as shown in Figure 12-1:

```
struct msqid_ds
{
struct ipc_permmsg_perm;       /* operation permission struct*/
struct msg  *msg_first;        /* ptr to first message on q */
struct msg *msg_last;          /* ptr to last message on q */
ulong_t    msg_cbytes;         /* current # bytes on q */
msgqnum_t  msg_qnum;           /* # of messages on Q */
msglen_t   msg_qbytes;         /* max # of bytes on q */
pid_t      msg_lspid;          /* pid of last msgsnd */
pid_t      msg_lrpid;          /* pid of last msgrcv */
time_t     msg_stime;          /* last msgsnd time */
long       msg_pad1;           /* reserved for time_t expansion */
time_t     msg_rtime;          /* last msgrcv time */
long       msg_pad2;           /* reserved for time_t expansion */
time_t     msg_ctime;          /* last change time */
long       msg_pad3;           /* time_t expansion */
long       msg_pad4(MSG_PAD];  /* reserve area */
};
```

Figure 12-1. Definition of msqid_ds Structure

The C programming language data structure definition for the message-queue data structure `msqid_ds` is located in the **`sys/msg.h`** header file.

With the OS, the definition of the `ipc_perm` data structure is as shown in Figure 12-2.

```
struct ipc_perm
{
uid_t          uid;      /*   owner's user id                    */
gid_t          gid;      /*   owner's group id                   */
uid_t          cuid;     /*   creator's user id*/
gid_t          cgid;     /*   creator's group id */
mode_t         mode;     /*   access modes */
ulong          seq;      /*   slot usage sequence number *
key_t          key;      /*   key */
struct ipc_sec*secp;     /*   security structure ptr */
long           pad[IPC_PERM_PAD];
                          /* reserve area */
};
```

Figure 12-2. Definition of ipc_perm Structure

The C programming language data structure definition for the interprocess communication permissions data structure `ipc_perm` is located in the **`sys/ipc.h`** header file and is common to all IPC facilities.

The data structure `ipc_sec` is used to hold the Discretionary and Mandatory Access Control (DAC and MAC) information used by the Enhanced Security Utilities. The `ipc_sec` data structure is shown in Figure 12-3 and is located in the **`sys/ipcsec.h`** header file.

```
struct ipc_sec
{
struct ipc_dac  *dacp;     /*  ptr */
lid_t           ipc_lid;   /* MAC level identifier */
};
```

Figure 12-3. Definition of ipc_sec Structure

The `ipc_sec` data structure contains both DAC and MAC information. DAC information is contained in the `ipc_dac` data structure. The MAC security level is stored in the `ipc_lid` field. The type `lid_t` is defined as an unsigned long integer in the header file **`sys/types.h.`**

The **`msgget`** system call is used to perform one of two tasks:

- to get a new message queue identifier and create an associated message queue and data structure for it
- to return an existing message queue identifier that already has an associated message queue and data structure

Both tasks require a `key` argument passed to the **`msgget`** system call. For the first task, if the `key` is not already in use for an existing message queue identifier, a new identifier is returned with an associated message queue and data structure created for the `key.` If the Enhanced Security Utilities are installed, the new `msqid` inherits the security level of the creating process.

This occurs as long as no system-tunable parameters would be exceeded and a control command `IPC_CREAT` is specified in the *msgflg* argument passed in the system call.

There is also a provision for specifying a `key` of value zero, known as the private `key` (`IPC_PRIVATE`). When specified, a new identifier is always returned with an associated message queue and data structure created for it unless a system-tunable parameter would be exceeded. The **`ipcs`** command will show the `key` field for the `msqid` as all zeros.

For the second task, if a message queue identifier exists for the `key` specified, the value of the existing identifier is returned. If you do not want to have an existing message queue identifier returned, a control command (`IPC_EXCL`) can be specified (set) in the *msgflg* argument passed to the system call (see “Using shmget” for how to use this system call).

When the Enhanced Security Utilities are installed and running, keys are kept on a per-level basis. `Key`s within a security level are unique; however, the same `key` may exist at different security levels. Each `key` references a different message queue and data structure at each security level where the `key` exists. As mentioned before, a message queue and data set inherit the security level of the creating process. While the security level of the message queue cannot be changed, a process with appropriate privilege may perform multilevel operations on message queues. Refer to the “Multilevel Operation On Messages” section of this chapter for details.

When performing the first task, the process that calls **`msgget`** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed but the creating process always remains the creator. The message queue creator also determines the initial operation permissions for it.

Once a uniquely identified message queue and data structure are created, **`msgop`** (message operations) and **`msgctl`** (message control) can be used.

Message operations, as mentioned before, consist of sending and receiving messages. The **`msgsnd`** and **`msgrcv`** system calls are provided for each of these operations (see “Operations for Messages” for details of the **`msgsnd`** and **`msgrcv`** system calls.

The **`msgctl`** system call permits you to control the message facility in the following ways:

- by retrieving the data structure associated with a message queue identifier (`IPC_STAT`)
- by changing operation permissions for a message queue (`IPC_SET`)
- by changing the size (`msg_qbytes`) of the message queue for a particular message queue identifier (`IPC_SET`)
- by removing a particular message queue identifier from the UNIX operating system along with its associated message queue and data structure (`IPC_RMID`)

See the section “Controlling Message Queues” for details of the **`msgctl`** system call.

Getting Message Queues

This section describes how to use the **`msgget`** system call. The accompanying program illustrates its use.

Using msgget

The synopsis found in the **`msgget(2)`** system manual page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (key_t key, int msgflg);
```

All of these `#include` files are located in the **`/usr/include/sys`** directory of the UNIX operating system.

The following line in the synopsis:

```
int msgget (key_t key, int msgflg);
```

informs you that **`msgget`** is a function that returns an integer-type value. It also declares the types of the two formal arguments: *key* is of type `key_t,` and *msgflg* is of type `int.` `key_t` is defined by a `typedef` in the **`sys/types.h`** header file to be an integral type.

The integer returned from this function upon successful completion is the message queue identifier that was discussed earlier. Upon failure, the external variable `errno` is set to indicate the reason for failure, and the value `-1` (which is not a valid `msqid`) is returned.

As declared, the process calling the **`msgget`** system call must supply two arguments to be passed to the formal *key* and *msgflg* arguments.

A new `msqid` with an associated message queue and data structure is provided if either

- *key* is equal to `IPC_PRIVATE`,

or

- *key* is a unique integer and the control command `IPC_CREAT` is specified in the *msgflg* argument.

If the Enhanced Security Utilities are installed, the *key* is an integer that is not yet associated with a message queue at the security level of the calling process; and the new message queue and its data structure inherit the security level of the creating process.

The value passed to the *msgflg* argument must be an integer-type value that will specify the following:

- operations permissions
- control fields (commands)

Operation permissions determine the operations that processes are permitted to perform on the associated message queue. “Read” permission is necessary for receiving messages or for determining queue status by means of a **`msgctl`** `IPC_STAT` operation. “Write” permission is necessary for sending messages.

If the Enhanced Security Utilities are installed and running, a process must also pass Mandatory Access Control (MAC) checks to send or receive messages or query queue status. The MAC policy requires that the process and message queue be at identical security levels to send or receive messages. The level of the process must dominate that of the message queue for determining queue status by means of a **`msgctl`** `IPC_STAT` operation.

Table 12-1 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Table 12-1. Operation Permissions Codes

Operation Permissions	Octal Value
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

A specific value is derived by adding or bitwise ORing the octal values for the operation permissions wanted. That is, if read by user and read/write by others is desired, the code value would be 00406 (00400 plus 00006). There are constants located in the **`sys/msg.h`** header file which can be used for the user operations permissions. They are as follows:

```
MSG_W   0200    /* write permissions by owner */
MSG_R   0400    /* read permissions by owner */
```

Control flags are predefined constants (represented by all upper-case letters). The flags which apply to the **`msgget`** system call are `IPC_CREAT` and `IPC_EXCL` and are defined in the **`sys/ipc.h`** header file.

The value for *msgflg* is therefore a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This is accomplished by adding or bitwise ORing (|) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible.

The *msgflg* value can easily be set by using the flag names in conjunction with the octal operation permissions value:

```
msqid = msgget (key, (IPC_CREAT | 0400));
msqid = msgget (key, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the **`msgget(2)`** system manual page, success or failure of this system call depends upon the argument values for *key* and *msgflg* or system-tunable parameters. The system call will attempt to return a new message queue identifier if one of the following conditions is true:

- *key* is equal to `IPC_PRIVATE`
- *key* does not already have a message queue identifier associated with it and (*msgflg* and `IPC_CREAT`) is “true” (not zero).

If the Enhanced Security Utilities are installed, the *key* is an integer that is not yet associated with a message queue at the security level of the calling process; and the new message queue and its data structure inherit the security level of the creating process.

The *key* argument can be set to `IPC_PRIVATE` like this:

```
msqid = msgget (IPC_PRIVATE, msgflg);
```

The system call will always be attempted. Exceeding the `MSGMNI` system-tunable parameter always causes a failure. The `MSGMNI` system-tunable parameter determines the systemwide number of unique message queues that may be in use at any given time.

`IPC_EXCL` is another control command used in conjunction with `IPC_CREAT`. It will cause the system call to return an error if a message queue identifier already exists for the specified *key* (at the security level of the calling process, if the Enhanced Security Utilities are installed and running). This is necessary to prevent the process from thinking that it has received a new identifier when it has not. In other words, when both `IPC_CREAT` and `IPC_EXCL` are specified, a new message queue identifier is returned if the system call is successful.

Refer to the **`msgget(2)`** manual page for specific, associated data structure initialization for successful completion. The specific failure conditions and their error names are contained there also.

Example Program

The example program that is presented at the end of this section is a menu-driven program. It allows all possible combinations of using the **`msgget`** system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-8) by including the required header files as specified by the **`msgget(2)`** system manual page. Note that the **`<errno.h>`** header file is included as opposed to declaring `errno` as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self explanatory. These names make the programs more readable are perfectly valid since they are local to the program.

The variables declared for this program and what they are used for are as follows:

key	used to pass the value for the desired *key*
`opperm`	used to store the desired operation permissions
`flags`	used to store the desired control commands (flags)
`opperm_flags`	used to store the combination from the logical ORing of the `S` and `flags` variables; it is then used in the system call to pass the *msgflg* argument
`msqid`	used for returning the message queue identification number for a successful system call or the error code (`-1`) for an unsuccessful one.

The program begins by prompting for a hexadecimal *key*, an octal operation permissions code, and finally for the control command combinations (flags) which are selected from a menu (lines 15-32). All possible combinations are allowed even though they might not be viable. This allows errors to be observed for invalid combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored in the `opperm_flags` variable (lines 36-51).

The system call is made next, and the result is stored in the `msqid` variable (line 53).

Since the `msqid` variable now contains a valid message queue identifier or the error code (`-1`), it is tested to see if an error occurred (line 55). If `msqid` equals `-1`, a message indicates that an error resulted, and the external `errno` variable is displayed (line 57).

If no error occurred, the returned message queue identifier is displayed (line 61).

The example program for the **`msgget`** system call follows. We suggest you name the program file **`msgget.c`** and the executable file **`msgget`**.

```
 1    /* This is a program to illustrate
 2     **the message get, msgget(),
 3     **system call capabilities.*/
 4    #include    <stdio.h>
 5    #include    <sys/types.h>
 6    #include    <sys/ipc.h>
 7    #include    <sys/msg.h>
 8    #include    <errno.h>
 9    /*Start of main C language program*/
10    main()
11    {
12        key_t key;
13        int opperm, flags;
14        int msqid, opperm_flags;
15        /*Enter the desired key*/
16        printf("Enter the desired key in hex = ");
17        scanf("%x", &key);
18        /*Enter the desired octal operation
19          permissions.*/
20        printf("\nEnter the operation\n");
21        printf("permissions in octal = ");
22        scanf("%o", &opperm);
23        /*Set the desired flags.*/
24        printf("\nEnter corresponding number to\n");
25        printf("set the desired flags:\n");
printf(“No flags                  = 0\n”);
printf(“IPC_CREAT                 = 1\n”);
printf(“IPC_EXCL                  = 2\n”);
printf(“IPC_CREAT and IPC_EXCL    = 3\n”);
printf(“            Flags         = “);
/*Get the flag(s) to be set.*/
scanf(“%d”, &flags);
/*Check the values.*/
printf (“\nkey =0x%x, opperm = 0%o, flags = 0%o\n”,
    key, opperm, flags);
/*Incorporate the control fields (flags) with
  the operation permissions*/
switch (flags)
{
case 0:    /*No flags are to be set.*/
    opperm_flags = (opperm | 0);
    break;
case 1:    /*Set the IPC_CREAT flag.*/
    opperm_flags = (opperm | IPC_CREAT);
    break;
case 2:    /*Set the IPC_EXCL flag.*/
    opperm_flags = (opperm | IPC_EXCL);
    break;
case 3:    /*Set the IPC_CREAT and IPC_EXCL flags.*/
    opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
}
/*Call the msgget system call.*/
msqid = msgget (key, opperm_flags);
/*Perform the following if the call is unsuccessful.*/
if(msqid == -1)
{
printf (“\nThe msgget call failed, error number = %d\n”,
errno);
}
/*Return the msqid upon successful completion.*/
else
    printf (“\nThe msqid = %d\n”, msqid);
exit(0);
  }
```

Controlling Message Queues

This section describes how to use the **msgctl** system call. The accompanying program illustrates its use.

Using msgctl

The synopsis found in the **msgctl(2)** system manual page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (msqid, cmd, buf)
int              msqid, cmd;
struct msqid_ds *buf;
```

The **`msgctl`** system call requires three arguments to be passed to it; it returns an integer-type value.

When successful, it returns a zero value; when unsuccessful, it returns a `-1`

The *msqid* variable must be a valid, non-negative, integer value. In other words, it must have already been created by using the **`msgget`** system call.

The *cmd* argument can be any one of the following values:

`IPC_STAT`	return the status information contained in the associated data structure for the specified message queue identifier, and place it in the data structure pointed to by the `buf` pointer in the user memory area.
`IPC_SET`	for the specified message queue identifier, set the effective user and group identification, operation permissions, and the number of bytes for the message queue to the values contained in the data structure pointed to by the `buf` pointer in the user memory area.
`IPC_RMID`	remove the specified message queue identifier along with its associated message queue and data structure.

To perform an `IPC_SET` or `IPC_RMID` control command, a process must have:

- an effective user id of OWNER/CREATOR, or
- the `P_OWNER` privilege.

and, if the Enhanced Security Utilities are installed, must be at the same security level as the message queue identifier.

Note that a message queue can also be removed by using the **`ipcrm(1)`** command and specifying the **`-q`** *msqid* or the **`-Q`** *msgkey* option, where *msqid* specifies the identifier for the message queue and the *msgkey* argument specifies the key associated with the message queue. To use this command, a process must have the same privileges as those required for performing an `IPC_RMID` control command. See the **`ipcrm(1)`** system manual page for additional information on the use of this command.

Read permission is required to perform the `IPC_STAT` control command.

The details of this system call are discussed in the following example program. If you need more information on the logic manipulations in this program, read the **`msgget(2)`** system manual page; it goes into more detail than would be practical for this document.

Example Program

The example program that is presented at the end of this section is a menu-driven program. It allows all possible combinations of using the **`msgctl`** system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **`msgctl(2)`** system manual page. Note in this program that `errno` is declared as an external variable, and therefore, the **`<errno.h>`** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self explanatory. These names make the program more readable and are perfectly valid since they are local to the program.

The variables declared for this program and what they are used for are as follows:

`uid`	used to store the `IPC_SET` value for the effective user identification
`gid`	used to store the `IPC_SET` value for the effective group identification
`mode`	used to store the `IPC_SET` value for the operation permissions
`bytes`	used to store the `IPC_SET` value for the number of bytes in the message queue (`msg_qbytes`)
`rtrn`	used to store the return integer value from the system call
`msqid`	used to store and pass the message queue identifier to the system call
`command`	used to store the code for the desired control command so that subsequent processing can be performed on it
`choice`	used to determine which member is to be changed for the `IPC_SET` control command
`msqid_ds`	used to receive the specified message queue identifier's data structure when an `IPC_STAT` control command is performed
`buf`	a pointer passed to the system call which locates the data structure in the user memory area where the `IPC_STAT` control command is to place its return values or where the `IPC_SET` command gets the values to set

Note that the `msqid_ds` data structure in this program (line 16) uses the data structure, located in the **`sys/msg.h`** header file of the same name, as a template for its declaration.

The next important thing to observe is that although the `buf` pointer is declared to be a pointer to a data structure of the `msqid_ds` type, it must also be initialized to contain the address of the user memory area data structure (line 17). Now that all of the required declarations have been explained for this program, this is how it works.

First, the program prompts for a valid message queue identifier which is stored in the `msqid` variable (lines 19, 20). This is required for every **`msgctl`** system call.

Then the code for the desired control command must be entered (lines 21-27) and stored in the command variable. The code is tested to determine the control command for subsequent processing.

If the `IPC_STAT` control command is selected (code 1), the system call is performed (lines 37, 38) and the status information returned is printed out (lines 39-46); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful (line 106), the status information of the last successful call is printed out. In addition, an error message is displayed and the `errno` variable is printed out (line 108). If the system call is successful, a message indicates this along with the message queue identifier used (lines 110-113).

If the `IPC_SET` control command is selected (code 2), the first thing is to get the current status information for the message queue identifier specified (lines 50-52). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 53-59). This code is stored in the choice variable (line 60). Now, depending upon the member picked, the program prompts for the new value (lines 66-95). The value is placed into the appropriate member in the user memory area data structure, and the system call is made (lines 96-98). Depending upon success or failure, the program returns the same messages as for `IPC_STAT` above.

If the `IPC_RMID` control command (code 3) is selected, the system call is performed (lines 100-103), and the `msqid` along with its associated message queue and data structure are removed from the UNIX operating system. Note that the `buf` pointer is ignored in performing this control command, and its value can be zero or `NULL`. Depending upon the success or failure, the program returns the same messages as for the other control commands.

The example program for the **`msgctl`** system call follows. We suggest that you name the source program file **`msgctl.c`** and the executable file **`msgctl`**.

```
 /* This is a program to illustrate
 **the message control, msgctl(),
 **system call capabilities.
*/
 /*Include necessary header files.*/
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/msg.h>
 /*Start of main C language program*/
 main()
 {
     extern int errno;
     int uid, gid, mode, bytes;
     int rtrn, msqid, command, choice;
     struct msqid_ds msqid_ds, *buf;
     buf = &msqid_ds;
     /* Get the msqid, and command.*/
     printf("Enter the msqid = ");
     scanf("%d", &msqid);
     printf("\nEnter the number for\n");
     printf("the desired command:\n");
     printf("IPC_STAT    =  1\n");
     printf("IPC_SET     =  2\n");
     printf("IPC_RMID    =  3\n");
     printf("Entry       =  ");
     scanf("%d", &command);
      /* Check the values.*/
     printf ("\nmsqid =%d, command = %d\n",
         msqid, command);
     switch (command)
     {
     case 1:    /* Use msgctl() to duplicate
                 the data structure for
                 msqid in the msqid_ds area pointed
                 to by buf and then print it out.*/
         rtrn = msgctl(msqid, IPC_STAT,
             buf);
         printf ("\nThe USER ID = %d\n",
             buf->msg_perm.uid);
         printf ("The GROUP ID = %d\n",
             buf->msg_perm.gid);
         printf ("The operation permissions = 0%o\n",
             buf->msg_perm.mode);
         printf ("The msg_qbytes = %d\n",
             buf->msg_qbytes);
         break;
     case 2:    /* Select and change the desired
                 member(s) of the data structure.*/
         /*Get the original data for this msqid
               data structure first.*/
         rtrn = msgctl(msqid, IPC_STAT, buf);
         printf("\nEnter the number for the\n");
         printf("member to be changed:\n");
         printf("msg_perm.uid   = 1\n");
         printf("msg_perm.gid   = 2\n");
         printf("msg_perm.mode  = 3\n");
         printf("msg_qbytes     = 4\n");
         printf("Entry          = ");
         scanf("%d", &choice);
         /*Only one choice is allowed per
           pass as an invalid entry will
               cause repetitive failures until
           msqid_ds is updated with
               IPC_STAT.*/
```

```
        switch(choice){
        case 1:
            printf(“\nEnter USER ID = “);
            scanf (“%ld”, &uid);
            buf->msg_perm.uid =(uid_t)uid;
            printf(“\nUSER ID = %d\n”,
                buf->msg_perm.uid);
            break;
        case 2:
            printf(“\nEnter GROUP ID = “);
            scanf(“%d”, &gid);
            buf->msg_perm.gid = gid;
            printf(“\nGROUP ID = %d\n”,
                buf->msg_perm.gid);
            break;
        case 3:
            printf(“\nEnter MODE = “);
            scanf(“%o”, &mode);
            buf->msg_perm.mode = mode;
            printf(“\nMODE = 0%o\n”,
                buf->msg_perm.mode);
            break;
        case 4:
            printf(“\nEnter msq_bytes = “);
            scanf(“%d”, &bytes);
            buf->msg_qbytes = bytes;
            printf(“\nmsg_qbytes = %d\n”,
                buf->msg_qbytes);
            break;
        }
        /*Do the change.*/
        rtrn = msgctl(msqid, IPC_SET,
            buf);
        break;
    case 3:    /*Remove the msqid along with its
                  associated message queue
                  and data structure.*/
        rtrn = msgctl(msqid, IPC_RMID, (struct msqid_ds *) NULL);
    }
    /*Perform the following if the call is unsuccessful.*/
    if(rtrn == -1)
    {
printf (“\nThe msgctl call failed, error number = %d\n”, errno);
    }
    /*Return the msqid upon successful completion.*/
    else
        printf (“\nMsgctl was successful for msqid = %d\n”,
            msqid);
    exit (0);
 }
```

Operations for Messages

This section describes how to use the **`msgsnd`** and **`msgrcv`** system calls. The accompanying program illustrates their use.

If the Enhanced Security Utilities are installed and running, communicating processes must be at the same security level.

Using Message Operations: msgsnd and msgrcv

The synopsis found in the **msgop(2)** system manual page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (msqid, msgp, msgsz, msgflg)
int             msqid;
struct msgbuf   *msgp;
int             msgsz, msgflg;

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int             msqid;
struct msgbuf   *msgp;
int             msgsz;
long            msgtyp;
int             msgflg;
```

Sending a Message

The **msgsnd** system call requires four arguments to be passed to it. It returns an integer value.

When successful, it returns a zero value; when unsuccessful, **msgsnd** returns a -1.

The *msqid* argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **msgget** system call.

The *msgp* argument is a pointer to a structure in the user memory area that contains the type of the message and the message to be sent.

The *msgsz* argument specifies the length of the character array in the data structure pointed to by the *msgp* argument. This is the length of the message. The maximum size of this array is determined by the MSGMAX system-tunable parameter.

The *msgflg* argument allows the "blocking message operation" to be performed if the IPC_NOWAIT flag is not set ((*msgflg* and IPC_NOWAIT)= = 0); the operation would block if the total number of bytes allowed on the specified message queue are in use (msg_qbytes or MSGMNB), or the total system-wide number of messages on all queues is equal to the system- imposed limit (MSGTQL). If the IPC_NOWAIT flag is set, the system call will fail and return a -1.

The value of the msg_qbytes data structure member can be lowered from MSGMNB by using the **msgctl** IPC_SET control command, but only a process with the P_SYSOPS privilege can raise it afterwards.

Further details of this system call are discussed in the following program. If you need more information on the logic manipulations in this program, read “Using msgget.” It goes into more detail than would be practical for every system call.

Receiving Messages

The **`msgrcv`** system call requires five arguments to be passed to it; it returns an integer value.

When successful, it returns a value equal to the number of bytes received; when unsuccessful it returns a `-1`.

The *msqid* argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **`msgget`** system call.

The *msgp* argument is a pointer to a structure in the user memory area that will receive the message type and the message text.

The *msgsz* argument specifies the length of the message to be received. If its value is less than the message in the array, an error can be returned if desired (see the *msgflg* argument below).

The *msgtyp* argument is used to pick the first message on the message queue of the particular type specified. If it is equal to zero, the first message on the queue is received; if it is greater than zero, the first message of the same type is received; if it is less than zero, the lowest type that is less than or equal to its absolute value is received.

The *msgflg* argument allows the “blocking message operation” to be performed if the `IPC_NOWAIT` flag is not set ((*msgflg* and `IPC_NOWAIT`) == 0); the operation would block if there is not a message on the message queue of the desired type (`msgtyp`) to be received. If the `IPC_NOWAIT` flag is set, the system call will fail immediately when there is not a message of the desired type on the queue. **`msgflg`** can also specify that the system call fail if the message is longer than the `size` to be received; this is done by not setting the `MSG_NOERROR` flag in the *msgflg* argument ((*msgflg* and `MSG_NOERROR`)) == 0). If the `MSG_NOERROR` flag is set, the message is truncated to the length specified by the `msgsz` argument of **`msgrcv.`**

Further details of this system call are discussed in the following program. If you need more information on the logic manipulations in this program, read “Using msgget.” It goes into more detail than would be practical for every system call.

Example Program

The example program that is presented at the end of this section is a menu-driven program. It allows all possible combinations of using the **`msgsnd`** and **`msgrcv`** system calls to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **`msgop(2)`** system manual page. Note that in this program `errno` is declared as an external variable; therefore, the **`<errno.h>`** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self explanatory. These names make the program more readable and are perfectly valid since they are local to the program.

The variables declared for this program and what they are used for are as follows:

`sndbuf`	used as a buffer to contain a message to be sent (line 13); it uses the `msgbuf1` data structure as a template (lines 10-13). The `msgbuf1` structure (lines 10-13) is a duplicate of the `msgbuf` structure contained in the **`sys/msg.h`** header file, except that the size of the character array for `mtext` is tailored to fit this application. The `msgbuf` structure should not be used directly because `mtext` has only one element that would limit the size of each message to one character. Instead, declare your own structure. It should be identical to `msgbuf` except that the size of the `mtext` array should fit your application.
`rcvbuf`	used as a buffer to receive a message (line 13); it uses the `msgbuf1` data structure as a template (lines 10-13)
`msgp`	used as a pointer (line 13) to both the `sndbuf` and `rcvbuf` buffers
`i`	used as a counter for inputing characters from the keyboard, storing them in the array, and keeping track of the message length for the **`msgsnd`** system call; it is also used as a counter to output the received message for the **`msgrcv`** system call
`c`	used to receive the input character from the **`getchar`** function (line 50)
`flag`	used to store the code of `IPC_NOWAIT` for the **`msgsnd`** system call (line 61)
`flags`	used to store the code of the `IPC_NOWAIT` or `MSG_NOERROR` flags for the **`msgrcv`** system call (line 117)
`choice`	used to store the code for sending or receiving (line 30)
`rtrn`	used to store the return values from all system calls
`msqid`	used to store and pass the desired message queue identifier for both system calls
`msgsz`	used to store and pass the `size` of the message to be sent or received
`msgflg`	used to pass the value of flag for sending or the value of flags for receiving
`msgtyp`	used for specifying the message type for sending or for picking a message type for receiving.

Note that a `msqid_ds` data structure is set up in the program (line 21) with a pointer initialized to point to it (line 22); this will allow the data structure members affected by message operations to be observed. They are observed by using the **`msgctl`** (`IPC_STAT`) system call to get them for the program to print them out (lines 80-92 and lines 160-167).

The first thing the program prompts for is whether to send or receive a message. A corresponding code must be entered for the desired operation; it is stored in the choice variable (lines 23-30). Depending upon the code, the program proceeds as in the following **`msgsnd`** or **`msgrcv`** sections.

msgsnd

When the code is to send a message, the `msgp` pointer is initialized (line 33) to the address of the send data structure, `sndbuf.` Next, a message type must be entered for the message; it is stored in the variable `msgtyp` (line 42), and then (line 43) it is put into the `mtype` member of the data structure pointed to by `msgp.`

The program now prompts for a message to be entered from the keyboard and enters a loop of getting and storing into the `mtext` array of the data structure (lines 48-51). This will continue until an end-of-file is recognized which, for the **`getchar`** function, is a CTRL-d immediately following a carriage return (RETURN).

The message is immediately echoed from the `mtext` array of the `sndbuf` data structure to provide feedback (lines 54-56).

The next and final thing that must be decided is whether to set the `IPC_NOWAIT` flag. The program does this by requesting that a code of a 1 be entered for yes or anything else for no (lines 57-65). It is stored in the flag variable. If a 1 is entered, `IPC_NOWAIT` is logically ORed with `msgflg`; otherwise, `msgflg` is set to zero.

The **`msgsnd`** system call is performed (line 69). If it is unsuccessful, a failure message is displayed along with the error number (lines 70-72). If it is successful, the returned value is printed and should be zero (lines 73-76).

Every time a message is successfully sent, three members of the associated data structure are updated. They are:

`msg_qnum`	represents the total number of messages on the message queue; it is incremented by one.
`msg_lspid`	contains the process identification (PID) number of the last process sending a message; it is set accordingly.
`msg_stime`	contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) of the last message sent; it is set accordingly.

These members are displayed after every successful message send operation (lines 79-92).

msgrcv

When the code is to receive a message, the program continues execution as in the following paragraphs.

The `msgp` pointer is initialized to the `rcvbuf` data structure (line 99).

Next, the message queue identifier of the message queue from which to receive the message is requested; it is stored in `msqid` (lines 100-103).

The message type is requested; it is stored in `msgtyp` (lines 104-107).

The code for the desired combination of control flags is requested next; it is stored in flags (lines 108-117). Depending upon the selected combination, `msgflg` is set accordingly (lines 118-131).

Finally, the number of bytes to be received is requested; it is stored in `msgsz` (lines 132-135).

The **`msgrcv`** system call is performed (line 142). If it is unsuccessful, a message and error number is displayed (lines 143-145). If successful, a message indicates so, and the number of bytes returned and the `msg` type returned (because the value returned may be different from the value requested) is displayed followed by the received message (lines 150-156).

When a message is successfully received, three members of the associated data structure are updated. They are:

`msg_qnum`	contains the number of messages on the message queue; it is decremented by one.
`msg_lrpid`	contains the PID of the last process receiving a message; it is set accordingly.
`msg_rtime`	contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) that the last process received a message; it is set accordingly.

The sample code for the **`msgop`** system call follows. We suggest that you put the program into a source file called **`msgop.c`** and then compile it into an executable file called **`msgop.`**

```
 1    /*This is a program to illustrat
 2    **the message operations, msgop(),
 3    **system call capabilities.
 4    */
 5    /*Include necessary header files.*/
 6    #include    <stdio.h>
 7    #include    <sys/types.h>
 8    #include    <sys/ipc.h>
 9    #include    <sys/msg.h>
10    struct msgbuf1 {
11        long    mtype;
12        char    mtext[8192];
13    } sndbuf, rcvbuf, *msgp;
14    /*Start of main C language program*/
15    main()
16    {
17        extern int errno;
18        int i, c, flag, flags, choice;
19        int rtrn, msqid, msgsz, msgflg;
20        long mtype, msgtyp;
21        struct msqid_ds msqid_ds, *buf;
22        buf = &msqid_ds;
23        /*Select the desired operation.*/
24        printf("Enter the corresponding\n");
25        printf("code to send or\n");
26        printf("receive a message:\n");
27        printf("Send           =  1\n");
28        printf("Receive        =  2\n");
29        printf("Entry          =  ");
```

```
    scanf("%d", &choice);
    if(choice == 1) /*Send a message.*/
    {
        msgp = &sndbuf; /*Point to user send structure.*/
        printf("\nEnter the msqid of\n");
        printf("the message queue to\n");
        printf("handle the message = ");
        scanf("%d", &msqid);
        /*Set the message type.*/
        printf("\nEnter a positive integer\n");
        printf("message type (long) for the\n");
        printf("message = ");
        scanf("%ld", &msgtyp);
        msgp->mtype = msgtyp;
        /*Enter the message to send.*/
        printf("\nEnter a message: \n");
        /*A control-d (^d) terminates as
          EOF.*/
        /*Get each character of the message
          and put it in the mtext array.*/
        for(i = 0; ((c = getchar()) != EOF); i++)
            sndbuf.mtext[i] = c;
        /*Determine the message size.*/
        msgsz = i;
        /*Echo the message to send.*/
        for(i = 0; i < msgsz; i++)
            putchar(sndbuf.mtext[i]);
        /*Set the IPC_NOWAIT flag if
          desired.*/
        printf("\nEnter a 1 if you want \n");
        printf("the IPC_NOWAIT flag set:  ");
        scanf("%d", &flag);
        if(flag == 1)
            msgflg = IPC_NOWAIT;
        else
            msgflg = 0;
        /*Check the msgflg.*/
        printf("\nmsgflg = 0%o\n", msgflg);
        /*Send the message.*/
        rtrn = msgsnd(msqid, (const void*) msgp, msgsz, msgflg);
        if(rtrn == -1)
        printf("\nMsgsnd failed.  Error = %d\n",
                errno);
        else {
            /*Print the value of test which
                  should be zero for successful.*/
            printf("\nValue returned = %d\n", rtrn);
            /*Print the size of the message
              sent.*/
            printf("\nMsgsz = %d\n", msgsz);
            /*Check the data structure update.*/
            msgctl(msqid, IPC_STAT, buf);
            /*Print out the affected members.*/
            /*Print the incremented number of
              messages on the queue.*/
            printf("\nThe msg_qnum = %d\n",
                buf->msg_qnum);
            /*Print the process id of the last sender.*/
            printf("The msg_lspid = %d\n",
                buf->msg_lspid);
            /*Print the last send time.*/
            printf("The msg_stime = %d\n",
                buf->msg_stime);
        }
    }
```

```
        if(choice == 2)  /*Receive a message.*/
        {
            /*Initialize the message pointer
              to the receive buffer.*/
            msgp = &rcvbuf;
            /*Specify the message queue which contains
                   the desired message.*/
            printf(“\nEnter the msqid = “);
            scanf(“%d”, &msqid);
            /*Specify the specific message on the queue
                  by using its type.*/
            printf(“\nEnter the msgtyp = “);
            scanf(“%ld”, &msgtyp);
            /*Configure the control flags for the
                  desired actions.*/
            printf(“\nEnter the corresponding code\n”);
            printf(“to select the desired flags: \n”);
            printf(“No flags                    =  0\n”);
            printf(“MSG_NOERROR                 =  1\n”);
            printf(“IPC_NOWAIT                  =  2\n”);
            printf(“MSG_NOERROR and IPC_NOWAIT  =  3\n”);
            printf(“                Flags       =  “);
            scanf(“%d”, &flags);
            switch(flags) {
            case 0:
                msgflg = 0;
                break;
            case 1:
                msgflg = MSG_NOERROR;
                break;
            case 2:
                msgflg = IPC_NOWAIT;
                break;
            case 3:
                msgflg = MSG_NOERROR | IPC_NOWAIT;
                break;
            }
            /*Specify the number of bytes to receive.*/
            printf(“\nEnter the number of bytes\n”);
            printf(“to receive (msgsz) = “);
            scanf(“%d”, &msgsz);
            /*Check the values for the arguments.*/
            printf(“\nmsqid =%d\n”, msqid);
            printf(“\nmsgtyp = %ld\n”, msgtyp);
            printf(“\nmsgsz = %d\n”, msgsz);
            printf(“\nmsgflg = 0%o\n”, msgflg);
            /*Call msgrcv to receive the message.*/
            rtrn = msgrcv(msqid, (void*), msgp, msgsz, msgtyp,msgflg);
            if(rtrn == -1)  {
                printf(“\nMsgrcv failed., Error = %d\n”, errno);
            }
            else {
                printf (“\nMsgctl was successful\n”);
                printf(“for msqid = %d\n”,
                    msqid);
                /*Print the number of bytes received,
                  it is equal to the return
                  value.*/
                printf(“Bytes received = %d\n”, rtrn);
                /*Print the received message.*/
                for(i = 0; i<rtrn; i++)
                    putchar(rcvbuf.mtext[i]);
            }
            /*Check the associated data structure.*/
            msgctl(msqid, IPC_STAT, buf);
            /*Print the decremented number of messages.*/
            printf("\nThe msg_qnum = %d\n", buf->msg_qnum);
            /*Print the process id of the last receiver.*/
            printf("The msg_lrpid = %d\n", buf->msg_lrpid);
            /*Print the last message receive time*/
            printf("The msg_rtime = %d\n", buf->msg_rtime);
        }
    }
```

Multilevel Operation On Messages

If the Enhanced Security Utilities are installed and running, it may be desirable for a privileged process to communicate with a process running at another Mandatory Access Control (MAC) level. Multilevel operation on messages is allowed for privileged processes.

For a process to write to a message queue at a different security level, the `P_MACWRITE` privilege is required. Both `P_MACREAD` and `P_MACWRITE` are required for a process to receive a message from a queue at a different security level, since reading changes the queue. Both privileges are also required to change the attributes of a message queue existing at a different security level.

Even though a privileged process may access information at many different security levels, a `key` specified in a **`msgget`** system call will return a `msqid` with an associated message queue and data structure having a security level identical to that of the calling process. Once a privileged process has obtained a `msqid,` the process may perform any of the possible operations from any security level. Unlike `key`s, `msqid`s are not unique to a security level but to the entire system.

There is no defined interface to obtain the `msqid` for multilevel operation. A process may obtain the `msqid` via the **`msgget`** system call when invoked from a specific security level. A privileged user may obtain a `msqid` and security level information about the message queue by invoking the **`ipcs`** command. See the manual page **`ipcs(1)`** for details on the use of **`ipcs.`**

The **`lvlipc`** system call reports the security level of a message queue associated with the specified `msqid.` This system call is of little use to an unprivileged process, since a message queue created and used by the unprivileged process always has a security level equal to that of the process. The process with the `P_MACREAD` privilege, though, may use this system call to find the security level of any existing message queue on the system. The process must also have discretionary read access to the message queue.

For a detailed discussion of process privileges, see the individual system call manual pages and the **`intro(2)`** manual page.

System V Semaphores

The semaphore type of IPC allows processes (executing programs) to communicate through the exchange of semaphore values. Since many applications require the use of more than one semaphore, the operating system has the ability to create sets or arrays of semaphores. A semaphore set can contain one or more semaphores up to a limit set by the system administrator. The tunable parameter, `SEMMSL`, has a default value of `25`. Semaphore sets are created by using the **`semget`** (semaphore get) system call.

The process performing the **`semget`** system call becomes the owner/creator, determines how many semaphores are in the set, and sets the initial operation permissions for all processes, including itself. This process can subsequently relinquish ownership of the set or change the operation permissions using the **`semctl`** (semaphore control) system call. The creating process always remains the creator as long as the facility exists. Other processes with permission can use **`semctl`** to perform other control functions.

Any process can manipulate the semaphore(s) if the owner of the semaphore grants permission. If the Enhanced Security Utilities are installed and running, the semaphore and the manipulating process must also have identical security levels. Privileged processes can access semaphores at different security levels.

Each semaphore within a set can be incremented and decremented with the **`semop`** system call (documented in the corresponding system manual page*)*.

To increment a semaphore, an integer value of the desired magnitude is passed to the **`semop`** system call. To decrement a semaphore, a minus (`-`) value of the desired magnitude is passed.

The operating system ensures that only one process can manipulate a semaphore set at any given time. Simultaneous requests are performed sequentially in an arbitrary manner.

A process can test for a semaphore value to be greater than a certain value by attempting to decrement the semaphore by one more than that value. If the process is successful, then the semaphore value is greater than that certain value. Otherwise, the semaphore value is not. While doing this, the process can have its execution suspended (`IPC_NOWAIT` flag not set) until the semaphore value would permit the operation (other processes increment the semaphore), or the semaphore facility is removed.

The ability to suspend execution is called a “blocking semaphore operation.” This ability is also available for a process which is testing for a semaphore equal to zero; only read permission is required for this test; it is accomplished by passing a value of zero to the `semop` (semaphore operation) system call.

On the other hand, if the process is not successful and did not request to have its execution suspended, it is called a “nonblocking semaphore operation.” In this case, the process is returned a known error code (`-1`), and the external `errno` variable is set accordingly.

The blocking semaphore operation allows processes to communicate based on the values of semaphores at different points in time. Remember also that IPC facilities remain in the operating system until removed by a permitted process or until the system is reinitialized.

Operating on a semaphore set is done by using the **`semop`** system call.

When a set of semaphores is created, the first semaphore in the set is semaphore number zero. The last semaphore number in the set is numbered one less than the total in the set.

A single system call can be used to perform a sequence of these “blocking/nonblocking operations” on a set of semaphores. When performing a sequence of operations, the blocking/nonblocking operations can be applied to any or all of the semaphores in the set. Also, the operations can be applied in any order of semaphore number. However, no operations are done until they can all be done successfully. For example, if the first three of six operations on a set of ten semaphores could be completed successfully, but the fourth operation would be blocked, no changes are made to the set until all six operations can be performed without blocking. Either the operations are successful and the semaphores are changed, or one (“nonblocking”) operation is unsuccessful and none are changed. In short, the operations are “atomically performed.”

Remember, any unsuccessful nonblocking operation for a single semaphore or a set of semaphores causes immediate return with no operations performed at all. When this occurs, an error code (`-1`) is returned to the process, and the external variable `errno` is set accordingly.

System calls make these semaphore capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (`-1`) is returned to the process, and the external variable `errno` is set accordingly.

Using Semaphores

Before semaphores can be used (operated on or controlled) a uniquely identified data structure and semaphore set (array) must be created. The unique identifier is called the semaphore set identifier (`semid`); it is used to identify or refer to a particular data structure and semaphore set. This identifier is accessible by any process in the system, subject to normal access restrictions.

The semaphore set contains a predefined number of structures in an array, one structure for each semaphore in the set. The number of semaphores (`nsems`) in a semaphore set is user selectable. The following members are in each structure within a semaphore set:

- semaphore value
- PID performing last operation
- number of processes waiting for the semaphore value to become greater than its current value
- number of processes waiting for the semaphore value to equal zero

There is one associated data structure for the uniquely identified semaphore set. This data structure contains the following information related to the semaphore set:

- operation permissions data (operation permissions structure)
- pointer to first semaphore in the set (array)
- number of semaphores in the set
- last semaphore operation time
- last semaphore change time

The definition for the semaphore set (array member) `sem` is as shown in Figure 12-4:

```
struct sem {
    ushort_t   semval;          /* semaphore value */
    pid_t      sempid;          /* pid of last operation */
    ushort_t   semncnt;         /* # awaiting semval > cval */
    ushort_t   semzcnt;         /* # awaiting semval = 0 */
    sv_t       semn_sv;         /* synch variable for semncnt */
    sv_t       semz_sv;         /* synch variable for semzcnt */
}:
```

Figure 12-4. Definition of sem Structure

Likewise, the definition for the associated semaphore data structure `semid_ds` is as shown in Figure 12-5:

```
struct semid_ds {
    struct ipc_perm    sem_perm;     /* operation permission struct */
    struct sem         *sem_base;    /* ptr to first semaphre in set*/
    char               sem_pad0[2];  /* expansion */
    ushort_t           sem_nsems;    /* # of semaphores in set */
    time_t             sem_otime;    /* last semop time */
    long               sem_pad1;/* reserved for time_t expansion */
    time_t             sem_ctime;         /* last change time */
    long               sem_pad2;          /* time_t expansion */
    long               sem_pad3[SEM_PAD]; /* reserve area */
};
```

Figure 12-5. Definition of semid_ds Structure

The C programming language data structure definition for the semaphore set (array member) and for the `semid_ds` data structure are located in the **`sys/sem.h`** header file.

Note that the `sem_perm` member of this structure uses `ipc_perm` as a template.

The `ipc_perm` data structure is the same for all IPC facilities; it is located in the **`sys/ipc.h`** header file and is shown in the “System V Messages” section.

The **`semget`** system call is used to perform two tasks:

- to get a new semaphore set identifier and create an associated data structure and semaphore set for it
- to return an existing semaphore set identifier that already has an associated data structure and semaphore set

The task performed is determined by the value of the *key* argument passed to the `semget` system call. For the first task, if the *key* is not already in use for an existing `semid` and the `IPC_CREAT` flag is set, a new `semid` is returned with an associated data structure and semaphore set created for it provided no system tunable parameter would be exceeded.

When the Enhanced Security Utilities are installed and running, `key`s are kept on a per-level basis. `Key`s within a security level are unique; however, the same `key` may exist at different security levels. Each `key` references a different semaphore and data structure at each security level where the `key` exists. As mentioned before, a semaphore and data set inherit the security level of the creating process. While the security level of the semaphore cannot be changed, a process with appropriate privilege may perform multilevel operations on semaphores. Refer to the "Multilevel Operation On Semaphores" section of this chapter for details.

There is also a provision for specifying a `key` of value zero (0), which is known as the private `key` (`IPC_PRIVATE`). When this `key` is specified, a new identifier is always returned with an associated data structure and semaphore set created for it, unless a system-tunable parameter would be exceeded. The **`ipcs`** command will show the `key` field for the `semid` as all zeros.

When performing the first task, the process which calls **`semget`** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator (see the "Controlling Semaphores" section). The creator of the semaphore set also determines the initial operation permissions for the facility.

For the second task, if a semaphore set identifier exists for the `key` specified, the value of the existing identifier is returned. If you do not want to have an existing semaphore set identifier returned, a control command (`IPC_EXCL`) can be specified (set) in the *semflg* argument passed to the system call. The system call will fail if it is passed a value for the number of semaphores (`nsems`) that is greater than the number actually in the set; if you do not know how many semaphores are in the set, use `0` for `nsems.` (see "Using semget" for how to use this system call).

If the Enhanced Security Utilities are installed and running, `key`s are unique within security levels. The same `key` may exist at different security levels; however, the `key` will reference a different semaphore set and data structure at each security level where the `key` exists. As mentioned before, a semaphore set and data structure inherit the security level of the creating process; the security level of the semaphore set cannot be changed.

Once a uniquely identified semaphore set and data structure are created, **`semop`** (semaphore operations) and **`semctl`** (semaphore control) can be used.

Semaphore operations consist of incrementing, decrementing, and testing for zero. The **`semop`** system call is used to perform these operations (see "Operations On Semaphores" for details of the **`semop`** system call.

The **`semctl`** system call permits you to control the semaphore facility in the following ways:

- by returning the value of a semaphore (`GETVAL`)
- by setting the value of a semaphore (`SETVAL`)
- by returning the PID of the last process performing an operation on a semaphore set (`GETPID`)
- by returning the number of processes waiting for a semaphore value to become greater than its current value (`GETNCNT`)

- by returning the number of processes waiting for a semaphore value to equal zero (`GETZCNT`)
- by getting all semaphore values in a set and placing them in an array in user memory (`GETALL`)
- by setting all semaphore values in a semaphore set from an array of values in user memory (`SETALL`)
- by retrieving the data structure associated with a semaphore set (`IPC_STAT`)
- by changing operation permissions for a semaphore set (`IPC_SET`)
- by removing a particular semaphore set identifier from the operating system along with its associated data structure and semaphore set (`IPC_RMID`)

See the section “Controlling Semaphores” for details of the **`semctl`** system call.

Getting Semaphores

This section describes how to use the **`semget`** system call. The accompanying program illustrates its use.

Using semget

The synopsis found in the **`semget(2)`** system manual page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (key, nsems, semflag)
key_t key;
int nsems, semflag;
```

The following line in the synopsis:

```
int semget (key, nsems, semflg)
```

informs you that **`semget`** is a function with three formal arguments that returns an integer-type value. The next two lines:

```
key_t key;
int nsems, semflg;
```

declare the types of the formal arguments. `key_t` is defined by a `typedef` in the **`sys/types.h`** header file to be an integer.

The integer returned from this system call upon successful completion is the semaphore set identifier that was discussed above.

The process calling the **semget** system call must supply three actual arguments to be passed to the formal *key*, *nsems*, and *semflg* arguments.

A new `semid` with an associated semaphore set and data structure is created if either

- *key* is equal to `IPC_PRIVATE`,

or

- *key* is a unique integer and `semflg` ANDed with `IPC_CREAT` is "true."

If the Enhanced Security Utilities are installed, a unique *key* is an integer that is not yet associated with a message queue at the security level of the calling process; the new message queue and its data structure inherit the security level of the creating process.

The value passed to the *semflg* argument must be an integer that will specify the following:

- operation permissions
- control fields (commands)

Table 12-2 shows the numeric values (expressed in octal notation) for the valid operation permissions codes.

Table 12-2. Operation Permissions Codes

Operation Permissions	Octal Value
Read by User	00400
Alter by User	00200
Read by Group	00040
Alter by Group	00020
Read by Others	00004
Alter by Others	00002

A specific value is derived by adding or bitwise ORing the values for the operation permissions wanted. That is, if read by user and read/alter by others is desired, the code value would be 00406 (00400 plus 00006). There are constants **#define**'d in the **sys/sem.h** header file which can be used for the user (OWNER). They are as follows:

```
SEM_A    0200    /* alter permission by owner */
SEM_R    0400    /* read permission by owner */
```

Control flags are predefined constants (represented by all upper-case letters). The flags that apply to the `semget` system call are `IPC_CREAT` and `IPC_EXCL` and are defined in the **sys/ipc.h** header file.

The value for `semflg` is, therefore, a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This specification is accomplished by adding or bitwise ORing (|) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible.

The `semflg` value can easily be set by using the flag names in conjunction with the octal operation permissions value:

```
semid = semget (key, nsems, (IPC_CREAT | 0400));
semid = semget (key, nsems, (IPC_CREAT | IPC_EXCL|0400));
```

As specified by the **`semget(2)`** system manual page, success or failure of this system call depends upon the actual argument values for *key*, *nsems*, and *semflg*, and system-tunable parameters. The system call will attempt to return a new semaphore set identifier if one of the following conditions is true:

- *key* is equal to `IPC_PRIVATE`
- *key* does not already have a semaphore set identifier associated with it and (`semflg & IPC_CREAT`) is “true” (not zero).

If the Enhanced Security Utilities are installed, the *key* is an integer that is not yet associated with a message queue at the security level of the calling process; and the new message queue and its data structure inherit the security level of the creating process.

The *key* argument can be set to `IPC_PRIVATE` like this:

```
semid = semget(IPC_PRIVATE, nsems, semflg);
```

Exceeding the `SEMMNI, SEMMNS`, or `SEMMSL` system-tunable parameters will always cause a failure. The `SEMMNI` system-tunable parameter determines the maximum number of unique semaphore sets (`semid`'s) that may be in use at any given time. The `SEMMNS` system-tunable parameter determines the maximum number of semaphores in all semaphore sets system wide. The `SEMMSL` system-tunable parameter determines the maximum number of semaphores in each semaphore set.

`IPC_EXCL` is another control command used in conjunction with `IPC_CREAT.` It will cause the system call to return an error if a semaphore set identifier already exists for the specified *key* provided. (If the Enhanced Security Utilities are installed and running, it will cause the system call to return an error if a semaphore set identifier already exists at the security level of the calling process for the specified *key* provided.) This is necessary to prevent the process from thinking that it has received a new (unique) identifier when it has not. In other words, when both `IPC_CREAT` and `IPC_EXCL` are specified, a new semaphore set identifier is returned if the system call is successful. Any value for *semflg* returns a new identifier if the *key* equals zero (`IPC_PRIVATE`) and no system-tunable parameters are exceeded.

Refer to the **`semget(2)`** manual page for specific associated data structure initialization for successful completion. The specific failure conditions and their error names are contained there also.

Example Program

The example program that is presented at the end of this section is a menu-driven program. It allows all possible combinations of using the `semget` system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-8) by including the required header files as specified by the **`semget(2)`** system manual page. Note that the **`<errno.h>`** header file is included as opposed to declaring **`errno`** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis. Their declarations are self explanatory. These names make the program more readable and are perfectly valid since they are local to the program.

The variables declared for this program and what they are used for are as follows:

`key`	used to pass the value for the desired key
`opperm`	used to store the desired operation permissions
`flags`	used to store the desired control commands (flags)
`opperm_flags`	used to store the combination from the logical ORing of the `opperm` and `flags` variables; it is then used in the system call to pass the `semflg` argument
`semid`	used for returning the semaphore set identification number for a successful system call or the error code (`-1`) for an unsuccessful one.

The program begins by prompting for a hexadecimal *key*, an octal operation permissions code, and the control command combinations (flags) which are selected from a menu (lines 15-32). All possible combinations are allowed even though they might not be viable. This allows observing the errors for invalid combinations.

Next, the menu selection for the flags is combined with the operation permissions; the result is stored in `opperm_flags` (lines 36-52).

Then, the number of semaphores for the set is requested (lines 53-57); its value is stored in `nsems`.

The system call is made next; the result is stored in the `semid` (lines 60, 61).

Since the `semid` variable now contains a valid semaphore set identifier or the error code (`-1`), it is tested to see if an error occurred (line 63). If `semid` equals `-1`, a message indicates that an error resulted and the external `errno` variable is displayed (line 65). Remember that the external `errno` variable is only set when a system call fails; it should only be examined immediately following system calls.

If no error occurred, the returned semaphore set identifier is displayed (line 69).

The example program for the **`semget`** system call follows. We suggest that you name the source program file **`semget.c`** and the executable file **`semget`**.

```
1    /*This is a program to illustrate
2    **the semaphore get, semget(),
3    **system call capabilities.*/
4    #include    <stdio.h>
5    #include    <sys/types.h>
6    #include    <sys/ipc.h>
7    #include    <sys/sem.h>
```

```
#include    <errno.h>
/*Start of main C language program*/
main()
{
    key_t key;     /*declare as long integer*/
    int opperm, flags, nsems;
    int semid, opperm_flags;
    /*Enter the desired key*/
    printf(“\nEnter the desired key in hex = “);
    scanf(“%x”, &key);
    /*Enter the desired octal operation
          permissions.*/
    printf(“\nEnter the operation\n”);
    printf(“permissions in octal = “);
    scanf(“%o”, &opperm);
    /*Set the desired flags.*/
    printf(“\nEnter corresponding number to\n”);
    printf(“set the desired flags:\n”);
    printf(“No flags                  = 0\n”);
    printf(“IPC_CREAT                 = 1\n”);
    printf(“IPC_EXCL                  = 2\n”);
    printf(“IPC_CREAT and IPC_EXCL    = 3\n”);
    printf(“            Flags         = “);
    /*Get the flags to be set.*/
    scanf(“%d”, &flags);
    /*Error checking (debugging)*/
    printf (“\nkey =0x%x, opperm = 0%o, flags = %d\n”,
        key, opperm, flags);
    /*Incorporate the control fields (flags) with
          the operation permissions.*/
    switch (flags)
    {
    case 0:    /*No flags are to be set.*/
        opperm_flags = (opperm | 0);
        break;
    case 1:    /*Set the IPC_CREAT flag.*/
        opperm_flags = (opperm | IPC_CREAT);
        break;
    case 2:    /*Set the IPC_EXCL flag.*/
        opperm_flags = (opperm | IPC_EXCL);
        break;
    case 3: /*Set the IPC_CREAT and IPC_EXCL
                  flags.*/
        opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
    }
    /*Get the number of semaphores for this set.*/
    printf(“\nEnter the number of\n”);
    printf(“desired semaphores for\n”);
    printf(“this set (25 max) = “);
    scanf(“%d”, &nsems);
    /*Check the entry.*/
    printf(“\nNsems = %d\n”, nsems);
    /*Call the semget system call.*/
    semid = semget(key, nsems, opperm_flags);
    /*Perform the following if the call is unsuccessful.*/
    if(semid == -1)
    {
printf(“The semget call failed, error number = %d\n”, errno);
    }
    /*Return the semid upon successful completion.*/
    else
        printf(“\nThe semid = %d\n”, semid);
    exit(0);
}
```

Controlling Semaphores

This section describes how to use the **semctl** system call. The accompanying program illustrates its use.

Using semctl

The synopsis found in the **semctl(2)** system manual page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (semid, semnum, cmd, arg)
int         semid, cmd;
int         semnum;
union semun
{
        int val;
        struct semid_ds *buf;
        ushort *array;
} arg;
```

The **semctl** system call requires four arguments to be passed to it, and it returns an integer value.

The *semid* argument must be a valid, non-negative, integer value that has already been created by using the **semget** system call.

The *semnum* argument is used to select a semaphore by its number. This relates to sequences of operations (atomically performed) on the set. When a set of semaphores is created, the first semaphore is number 0, and the last semaphore is numbered one less than the total in the set.

The *cmd* argument can be replaced by one of the following values:

`GETVAL`	return the value of a single semaphore within a semaphore set
`SETVAL`	set the value of a single semaphore within a semaphore set
`GETPID`	return the PID of the process that performed the last operation on the semaphore within a semaphore set
`GETNCNT`	return the number of processes waiting for the value of a particular semaphore to become greater than its current value
`GETZCNT`	return the number of processes waiting for the value of a particular semaphore to be equal to zero
`GETALL`	return the value for all semaphores in a semaphore set
`SETALL`	set all semaphore values in a semaphore set

`IPC_STAT`	return the status information contained in the associated data structure for the specified `semid`, and place it in the data structure pointed to by the `buf` pointer in the user memory area; **`arg.buf`** is the union member that contains pointer
`IPC_SET`	for the specified semaphore set (`semid`), set the effective user/group identification and operation permissions
`IPC_RMID`	remove the specified semaphore set (`semid`) along with its associated data structure.

To perform an `IPC_SET` or `IPC_RMID` control command, a process must have:

- an effective user id of OWNER/CREATOR, or
- the `P_OWNER` privilege,

and, if the Enhanced Security Utilities are installed, must be at the same security level as the `semid.`

Note that a semaphore set can also be removed by using the **`ipcrm(1)`** command and specifying the **`-s`** *semid* or the **`-S`** *semkey* option, where *semid* specifies the identifier for the semaphore set and the *semkey* argument specifies the key associated with the semaphore set. To use this command, a process must have the same privileges as those required for performing an `IPC_RMID` control command. See the **`ipcrm(1)`** system manual page for additional information on the use of this command.

The remaining control commands require either read or write permission, as appropriate.

If the Enhanced Security Utilities are installed and running, the process and the semaphore must be at the same security level and the process must pass DAC write access checks to execute commands that alter the semaphore set. To execute commands that do not alter the semaphore set, the process's level must dominate that of the semaphore set, and the process must pass DAC read access checks. A process with the `P_MACREAD` and/or `P_MACWRITE` privileges may override the security level restriction.

The *arg* argument is used to pass the system call the appropriate union member for the control command to be performed. For some of the control commands, the *arg* argument is not required and is simply ignored.

- *arg*`.val` required: `SETVAL`
- *arg*`.buf` required: `IPC_STAT`, `IPC_SET`
- *arg*`.array` required: `GETALL`, `SETALL`
- *arg* ignored: `GETVAL, GETPID, GETNCNT, GETZCNT, IPC_RMID`

The details of this system call are discussed in the following program. If you need more information on the logic manipulations in this program, read “Using semget.” It goes into more detail than would be practical for every system call.

Example Program

The example program that is presented at the end of this section is a menu-driven program. It allows all possible combinations of using the **`semctl`** system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **`semctl(2)`** system manual page. Note that in this program `errno` is declared as an external variable, and therefore the **`<errno.h>`** header file does not have to be included.

Variable, structure, and union names have been chosen to be as close as possible to those in the synopsis. Their declarations are self explanatory. These names make the program more readable and are perfectly valid since they are local to the program.

The variables declared for this program and what they are used for are as follows:

`semid_ds`	used to receive the specified semaphore set identifier's data structure when an `IPC_STAT` control command is performed
`c`	used to receive the input values from the **`scanf`** function (line 119) when performing a `SETALL` control command
`i`	used as a counter to increment through the union `arg.array` when displaying the semaphore values for a `GETALL` (lines 98-100) control command, and when initializing the `arg.array` when performing a `SETALL` (lines 117-121) control command
`length`	used as a variable to test for the number of semaphores in a set against the `i` counter variable (lines 98, 117)
`uid`	used to store the `IPC_SET` value for the user identification
`gid`	used to store the `IPC_SET` value for the group identification
`mode`	used to store the `IPC_SET` value for the operation permissions
`retrn`	used to store the return value from the system call
`semid`	used to store and pass the semaphore set identifier to the system call
`semnum`	used to store and pass the semaphore number to the system call
`cmd`	used to store the code for the desired control command so that subsequent processing can be performed on it
`choice`	used to determine which member (`uid`, `gid`, `mode`) for the `IPC_SET` control command is to be changed
`semvals[]`	used to store the set of semaphore values when getting (`GETALL`) or initializing (`SETALL`)
`arg.val`	used to pass the system call a value to set, or to store a value returned from the system call, for a single semaphore (union member)
`arg.buf`	a pointer passed to the system call which locates the data structure in the user memory area where the `IPC_STAT` control command is to place its return values, or where the `IPC_SET` command gets the values to set (union member)

`arg.array`	a pointer passed to the system call which locates the array in the user memory where the `GETALL` control command is to place its return values, or when the `SETALL` command gets the values to set (union member)

Note that the `semid_ds` data structure in this program (line 14) uses the data structure located in the **`sys/sem.h`** header file of the same name as a template for its declaration.

Note that the `semvals` array is declared to have 25 elements (0 through 24). This number corresponds to the maximum number of semaphores allowed per set (`SEMMSL`), a system-tunable parameter.

Now that all of the required declarations have been presented for this program, this is how it works.

First, the program prompts for a valid semaphore set identifier, which is stored in the `semid` variable (lines 24-26). This is required for all **`semctl`** system calls.

Then, the code for the desired control command must be entered (lines 17-42), and the code is stored in the `cmd` variable. The code is tested to determine the control command for subsequent processing.

If the `GETVAL` control command is selected (code 1), a message prompting for a semaphore number is displayed (lines 48, 49). When it is entered, it is stored in the `semnum` variable (line 50). Then, the system call is performed, and the semaphore value is displayed (lines 51-54). Note that the *arg* argument is not required in this case, and the system call will simply ignore it. If the system call is successful, a message indicates this along with the semaphore set identifier used (lines 197, 198); if the system call is unsuccessful, an error message is displayed along with the value of the external `errno` variable (lines 194, 195).

If the `SETVAL` control command is selected (code 2), a message prompting for a semaphore number is displayed (lines 55, 56). When it is entered, it is stored in the `semnum` variable (line 57). Next, a message prompts for the value to which the semaphore is to be set; it is stored as the *arg*`.val` member of the union (lines 58, 59). Then, the system call is performed (lines 60, 62). Depending upon success or failure, the program returns the same messages as for `GETVAL` above.

If the `GETPID` control command is selected (code 3), the system call is made immediately since all required arguments are known (lines 63-66), and the PID of the process performing the last operation is displayed. Note that the *arg* argument is not required in this case, and the system call will simply ignore it. Depending upon success or failure, the program returns the same messages as for `GETVAL` above.

If the `GETNCNT` control command is selected (code 4), a message prompting for a semaphore number is displayed (lines 67-71). When entered, it is stored in the `semnum` variable (line 73). Then, the system call is performed and the number of processes waiting for the semaphore to become greater than its current value is displayed (lines 73-76). Note that the *arg* argument is not required in this case, and the system call will simply ignore it. Depending upon success or failure, the program returns the same messages as for `GETVAL` above.

If the `GETZCNT` control command is selected (code 5), a message prompting for a semaphore number is displayed (lines 77-80). When it is entered, it is stored in the `semnum` variable (line 81). Then the system call is performed and the number of processes waiting for the semaphore value to become equal to zero is displayed (lines 82-85). Depending upon success or failure, the program returns the same messages as for `GETVAL` above.

If the `GETALL` control command is selected (code 6), the program first performs an `IPC_STAT` control command to determine the number of semaphores in the set (lines 87-93). The length variable is set to the number of semaphores in the set (line 93). The *arg*`.array` union member is set to point to the `semvals` array where the system call is to store the values of the semaphore set (line 96). Now, a loop is entered which displays each element of the *arg*`.array` from zero to one less than the value of length (lines 98-104). The semaphores in the set are displayed on a single line, separated by a space. Depending upon success or failure, the program returns the same messages as for `GETVAL` above.

If the `SETALL` control command is selected (code 7), the program first performs an `IPC_STAT` control command to determine the number of semaphores in the set (lines 107-110). The length variable is set to the number of semaphores in the set (line 113). Next, the program prompts for the values to be set and enters a loop which takes values from the keyboard and initializes the `semvals` array to contain the desired values of the semaphore set (lines 115-121). The loop puts the first entry into the array position for semaphore number zero and ends when the semaphore number that is filled in the array equals one less than the value of length. The *arg*`.array` union member is set to point to the `semvals` array from which the system call is to obtain the semaphore values. The system call is then made (lines 122-125). Depending upon success or failure, the program returns the same messages as for `GETVAL` above.

If the `IPC_STAT` control command is selected (code 8), the system call is performed (line 129), and the status information returned is printed out (lines 130-141); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful, the status information of the last successful one is printed out. In addition, an error message is displayed, and the `errno` variable is printed out (line 194).

If the `IPC_SET` control command is selected (code 9), the program gets the current status information for the semaphore set identifier specified (lines 145-149). This is necessary because this example program provides for changing only one member at a time, and the `semctl` system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 150-156). This code is stored in the `choice` variable (line 157). Now, depending upon the member picked, the program prompts for the new value (lines 158-181). The value is placed into the appropriate member in the user memory area data structure, and the system call is made (line 184). Depending upon success or failure, the program returns the same messages as for `GETVAL` above.

If the `IPC_RMID` control command (code 10) is selected, the system call is performed (lines 186-188). The semaphore set identifier along with its associated data structure and semaphore set is removed from the operating system. Depending upon success or failure, the program returns the same messages as for the other control commands.

The example program for the **`semctl`** system call follows. We suggest that you name the source program file **`semctl.c`** and the executable file **`semctl`**.

```
1    /*This is a program to illustrate
2    **the semaphore control, semctl(),
3    **system call capabilities.
4    */
```

```
/*Include necessary header files.*/
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>
/*Start of main C language program*/
main()
{
    extern int errno;
    struct semid_ds semid_ds;
    int c, i, length;
    int uid, gid, mode;
    int retrn, semid, semnum, cmd, choice;
    ushort semvals[25];
    union semun  {
        int val;
        struct semid_ds *buf;
        ushort *array;
    } arg;
    /*Enter the semaphore ID.*/
    printf(“Enter the semid = “);
    scanf(“%d”, &semid);
    /*Choose the desired command.*/
    printf(“\nEnter the number for\n”);
    printf(“the desired cmd:\n”);
    printf(“GETVAL      =  1\n”);
    printf(“SETVAL      =  2\n”);
    printf(“GETPID      =  3\n”);
    printf(“GETNCNT     =  4\n”);
    printf(“GETZCNT     =  5\n”);
    printf(“GETALL      =  6\n”);
    printf(“SETALL      =  7\n”);
    printf(“IPC_STAT    =  8\n”);
    printf(“IPC_SET     =  9\n”);
    printf(“IPC_RMID    =  10\n”);
    printf(“Entry       =  “);
    scanf(“%d”, &cmd);
    /*Check entries.*/
    printf (“\nsemid =%d, cmd = %d\n\n”,
        semid, cmd);
    /*Set the command and do the call.*/
    switch (cmd)
    {
    case 1: /*Get a specified value.*/
        printf(“\nEnter the semnum = “);
        scanf(“%d”, &semnum);
        /*Do the system call.*/
        retrn = semctl(semid, semnum, GETVAL, arg);
        printf(“\nThe semval = %d”, retrn);
        break;
    case 2: /*Set a specified value.*/
        printf(“\nEnter the semnum = “);
        scanf(“%d”, &semnum);
        printf(“\nEnter the value = “);
        scanf(“%d”, &arg.val);
        /*Do the system call.*/
        retrn = semctl(semid, semnum, SETVAL, arg);
        break;
    case 3: /*Get the process ID.*/
        retrn = semctl(semid, 0, GETPID, arg);
        printf(“\nThe sempid = %d”, retrn);
        break;
    case 4: /*Get the number of processes
        waiting for the semaphore to
        become greater than its current
```

```
            value.*/
            printf(“\nEnter the semnum = “);
            scanf(“%d”, &semnum);
            /*Do the system call.*/
            retrn = semctl(semid, semnum, GETNCNT, arg);
            printf(“\nThe semncnt = %d”, retrn);
            break;
        case 5: /*Get the number of processes
            waiting for the semaphore
            value to become zero.*/
            printf(“\nEnter the semnum = “);
            scanf(“%d”, &semnum);
            /*Do the system call.*/
            retrn = semctl(semid, semnum, GETZCNT, arg);
            printf(“\nThe semzcnt = %d”, retrn);
            break;
        case 6: /*Get all of the semaphores.*/
            /*Get the number of semaphores in
              the semaphore set.*/
            arg.buf = &semid_ds;
            retrn = semctl(semid, 0, IPC_STAT, arg);
            if(retrn == -1)
                goto ERROR;
            length = arg.buf->sem_nsems;
            /*Get and print all semaphores in the
              specified set.*/
            arg.array = semvals;
            retrn = semctl(semid, 0, GETALL, arg);
            for (i = 0; i < length; i++)
            {
                printf(“%d”, semvals[i]);
                /*Separate each
                  semaphore.*/
                printf(“ “);
            }
            break;
        case 7: /*Set all semaphores in the set.*/
            /*Get the number of semaphores in
              the set.*/
            arg.buf = &semid_ds;
            retrn = semctl(semid, 0, IPC_STAT, arg);
            if(retrn == -1)
                goto ERROR;
            length = arg.buf->sem_nsems;
            printf(“Length = %d\n”, length);
            /*Set the semaphore set values.*/
            printf(“\nEnter each value:\n”);
            for(i = 0; i < length ; i++)
            {
                scanf(“%d”, &c);
                semvals[i] = c;
            }
            /*Do the system call.*/
            arg.array = semvals;
            retrn = semctl(semid, 0, SETALL, arg);
            break;
        case 8: /*Get the status for the semaphore set.*/
            /*Get and print the current status values.*/
            arg.buf = &semid_ds;
            retrn = semctl(semid, 0, IPC_STAT, arg);
            printf (“\nThe USER ID = %d\n”,
                arg.buf->sem_perm.uid);
            printf (“The GROUP ID = %d\n”,
                arg.buf->sem_perm.gid);
            printf (“The operation permissions = 0%o\n”,
                arg.buf->sem_perm.mode);
            printf (“The number of semaphores in set = %d\n”,
                arg.buf->sem_nsems);
            printf (“The last semop time = %d\n”,
                arg.buf->sem_otime);
            printf (“The last change time  = %d\n”,
                arg.buf->sem_ctime);
            break;
        case 9:    /*Select and change the desired
                      member of the data structure.*/
            /*Get the current status values.*/
            arg.buf = &semid_ds;
            retrn = semctl(semid, 0, IPC_STAT, arg.buf);
            if(retrn == -1)
                goto ERROR;
            /*Select the member to change.*/
            printf(“\nEnter the number for the\n”);
            printf(“member to be changed:\n”);
            printf(“sem_perm.uid   = 1\n”);
            printf(“sem_perm.gid   = 2\n”);
            printf(“sem_perm.mode  = 3\n”);
            printf(“Entry          = “);
            scanf(“%d”, &choice);
            switch(choice){
            case 1: /*Change the user ID.*/
                printf(“\nEnter USER ID = “);
                scanf (“%d”, &uid);
                arg.buf->sem_perm.uid = uid;
                printf(“\nUSER ID = %d\n”,
                    arg.buf->sem_perm.uid);
                break;
            case 2: /*Change the group ID.*/
                printf(“\nEnter GROUP ID = “);
                scanf(“%d”, &gid);
                arg.buf->sem_perm.gid = gid;
                printf(“\nGROUP ID = %d\n”,
                    arg.buf->sem_perm.gid);
                break;
            case 3: /*Change the mode portion of
                  the operation
                              permissions.*/
                printf(“\nEnter MODE in octal = “);
                scanf(“%o”, &mode);
                arg.buf->sem_perm.mode = mode;
                printf(“\nMODE = 0%o\n”,
                    arg.buf->sem_perm.mode);
                break;
            }
            /*Do the change.*/
            retrn = semctl(semid, 0, IPC_SET, arg);
            break;
        case 10:    /*Remove the semid along with its
                      data structure.*/
            retrn = semctl(semid, 0, IPC_RMID, arg);
        }
        /*Perform the following if the call is unsuccessful.*/
        if(retrn == -1)
        {
    ERROR:
    printf (“\nThe semctl call failed!,error number =  %d\n”, errno);
            exit(0);
        }
        printf (“\n\nThe semctl system call was successful\n”);
        printf (“for semid = %d\n”, semid);
```

```
        exit (0);
}
```

Operations On Semaphores

This section describes how to use the **semop** system call. The accompanying program illustrates its use.

Using semop

The synopsis found in the **semop(2)** system manual page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (semid, sops, nsops)
int             semid;
struct sembuf   *sops;
unsigned        nsops;
```

The **semop** system call requires three arguments to be passed to it and returns an integer value which will be zero for successful completion or `-1` otherwise.

The *semid* argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **semget** system call.

The *sops* argument points to an array of structures in the user memory area that contains the following for each semaphore to be changed

- the semaphore number (`sem_num`)
- the operation to be performed (`sem_op`)
- the control flags (`sem_flg`)

The **sops* declaration means that either an array name (which is the address of the first element of the array) or a pointer to the array can be used. `sembuf` is the *tag* name of the data structure used as the template for the structure members in the array; it is located in the **sys/sem.h** header file.

The *nsops* argument specifies the length of the array (the number of structures in the array). The maximum size of this array is determined by the `SEMOPM` system-tunable parameter. Therefore, a maximum of `SEMOPM` operations can be performed for each **semop** system call.

The semaphore number (`sem_num`) determines the particular semaphore within the set on which the operation is to be performed.

The operation to be performed is determined by the following:

- if `sem_op` is positive, the semaphore value is incremented by the value of `sem_op`

- if sem_op is negative, the semaphore value is decremented by the absolute value of sem_op
- if sem_op is zero, the semaphore value is tested for equality to zero

The following operation commands (flags) can be used:

- IPC_NOWAIT—this operation command can be set for any operations in the array. The system call will return unsuccessfully without changing any semaphore values at all if any operation for which IPC_NOWAIT is set cannot be performed successfully. The system call will be unsuccessful when trying to decrement a semaphore more than its current value, or when testing for a semaphore to be equal to zero when it is not.
- SEM_UNDO—this operation command is used to tell the system to undo the process's semaphore changes automatically when the process exits; it allows processes to avoid deadlock problems. To implement this feature, the system maintains a table with an entry for every process in the system. Each entry points to a set of undo structures, one for each semaphore used by the process. The system records the net change.

Example Program

The example program that is presented at the end of this section is a menu-driven program. It allows all possible combinations of using the **semop** system call to be exercised. From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **shmop(2)** system manual page. Note that in this program errno is declared as an external variable; therefore, the **<errno.h>** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self explanatory. These names make the program more readable and are perfectly valid since they are local to the program.

The variables declared for this program and what they are used for are as follows:

sembuf[10]	used as an array buffer (line 14) to contain a maximum of ten sembuf type structures; ten is the standard value of the tunable parameter SEMOPM, the maximum number of operations on a semaphore set for each **semop** system call
sops	used as a pointer (line 14) to the sembuf array for the system call and for accessing the structure members within the array
string[8]	used as a character buffer to hold a number entered by the user
rtrn	used to store the return value from the system call
flags	used to store the code of the IPC_NOWAIT or SEM_UNDO flags for the **semop** system call (line 59)
sem_num	used to store the semaphore number entered by the user for each semaphore operation in the array

`i`	used as a counter (line 31) for initializing the structure members in the array, and used to print out each structure in the array (line 78)
`semid`	used to store the desired semaphore set identifier for the system call
`nsops`	used to specify the number of semaphore operations for the system call; must be less than or equal to `SEMOPM`

First, the program prompts for a semaphore set identifier that the system call is to perform operations on (lines 18-21). `semid` is stored in the `semid` variable (line 22).

A message is displayed requesting the number of operations to be performed on this set (lines 24-26). The number of operations is stored in the `nsops` variable (line 27).

Next, a loop is entered to initialize the array of structures (lines 29-76). The semaphore number, operation, and operation command (flags) are entered for each structure in the array. The number of structures equals the number of semaphore operations (`nsops`) to be performed for the system call, so `nsops` is tested against the `i` counter for loop control. Note that `sops` is used as a pointer to each element (structure) in the array, and `sops` is incremented just like `i. sops` is then used to point to each member in the structure for setting them.

After the array is initialized, all of its elements are printed out for feedback (lines 77-84).

The `sops` pointer is set to the address of the array (lines 85, 86). `sembuf` could be used directly, if desired, instead of `sops` in the system call.

The system call is made (line 88), and depending upon success or failure, a corresponding message is displayed. The results of the operation(s) can be viewed by using the **`semctl`** `GETALL` control command.

The example program for the **`semop`** system call follows. We suggest that you name the source program file **`semop.c`** and the executable file **`semop`**.

```
 1   /*This is a program to illustrate
 2    **the semaphore operations, semop(),
 3    **system call capabilities.
 4   */
 5   /*Include necessary header files.*/
 6   #include    <stdio.h>
 7   #include    <sys/types.h>
 8   #include    <sys/ipc.h>
 9   #include    <sys/sem.h>
10   /*Start of main C language program*/
11   main()
12   {
13       extern int errno;
14       struct sembuf sembuf[10], *sops;
15       char string[8];
16       int retrn, flags, sem_num, i, semid;
17       unsigned nsops;
18       /*Enter the semaphore ID.*/
19       printf("\nEnter the semid of\n");
20       printf("the semaphore set to\n");
21       printf("be operated on = ");
22       scanf("%d", &semid);
23       printf("\nsemid = %d", semid);
```

```
/*Enter the number of operations.*/
printf(“\nEnter the number of semaphore\n”);
printf(“operations for this set = “);
scanf(“%d”, &nsops);
printf(“\nsops = %d”, nsops);
/*Initialize the array for the
  number of operations to be performed.*/
for(i = 0, sops = sembuf; i < nsops; i++, sops++)
{
    /*This determines the semaphore in
      the semaphore set.*/
    printf(“\nEnter the semaphore\n”);
    printf(“number (sem_num) = “);
    scanf(“%d”, &sem_num);
    sops->sem_num = sem_num;
    printf(“\nThe sem_num = %d”, sops->sem_num);
    /*Enter a (-)number to decrement,
      an unsigned number (no +) to increment,
      or zero to test for zero.  These values
      are entered into a string and converted
      to integer values.*/
    printf(“\nEnter the operation for\n”);
    printf(“the semaphore (sem_op) = “);
    scanf(“%s”, string);
    sops->sem_op = atoi(string);
    printf(“\nsem_op = %d\n”, sops->sem_op);
    /*Specify the desired flags.*/
    printf(“\nEnter the corresponding\n”);
    printf(“number for the desired\n”);
    printf(“flags:\n”);
    printf(“No flags                  = 0\n”);
    printf(“IPC_NOWAIT                = 1\n”);
    printf(“SEM_UNDO                  = 2\n”);
    printf(“IPC_NOWAIT and SEM_UNDO   = 3\n”);
    printf(“            Flags         = “);
    scanf(“%d”, &flags);
    switch(flags)
    {
    case 0:
        sops->sem_flg = 0;
        break;
    case 1:
        sops->sem_flg = IPC_NOWAIT;
        break;
    case 2:
        sops->sem_flg = SEM_UNDO;
        break;
    case 3:
        sops->sem_flg = IPC_NOWAIT | SEM_UNDO;
        break;
    }
    printf(“\nFlags = 0%o\n”, sops->sem_flg);
}
/*Print out each structure in the array.*/
for(i = 0; i < nsops; i++)
{
    printf(“\nsem_num = %d\n”, sembuf[i].sem_num);
    printf(“sem_op = %d\n”, sembuf[i].sem_op);
    printf(“sem_flg = 0%o\n”, sembuf[i].sem_flg);
    printf(“ “);
}
sops = sembuf; /*Reset the pointer to
                 sembuf[0].*/
/*Do the semop system call.*/
retrn = semop(semid, sops, nsops);
```

```
        if(retrn == -1)  {
            printf(“\nSemop failed, error = %d\n”, errno);
        }
        else {
            printf (“\nSemop was successful\n”);
            printf(“for semid = %d\n”, semid);
            printf(“Value returned = %d\n”, retrn);
        }
    }
```

Multilevel Operation On Semaphores

If the Enhanced Security Utilities are installed and running, it may be desirable for a privileged process to communicate with a process running at another Mandatory Access Control (MAC) level. Multilevel operation on semaphores is allowed for privileged processes.

For a process to alter a semaphore at a different security level, the `P_MACWRITE` privilege is required. `P_MACREAD` is required for a process to read semaphore set at a different security level. Both privileges are required to change the attributes of a semaphore set existing at a different security level.

Even though a privileged process may access information at many different security levels, a `key` specified in a **`semget`** system call will return a `semid` with an associated semaphore set and data structure having a security level identical to that of the calling process. Once a privileged process has obtained a `semid,` the process may perform any of the possible operations from any security level. Unlike `keys, semids` are not unique to a security level but to the entire system.

There is no defined interface to obtain the `semid` for multilevel operation. A process may obtain the `semid` via the **`semget`** system call when invoked from a specific security level. A privileged user may also manually obtain a `semid` and security level information about the semaphore set by invoking the **`ipcs`** command. The privileged process must have the `P_MACREAD` privilege when invoking **`ipcs.`**

See the manual page **`ipcs(1)`** for details on the use of **`ipcs.`**

The **`lvlipc`** system call reports the security level of a semaphore set associated with the specified `semid.` This system call is of little use to an unprivileged process, since a semaphore set created and used by the unprivileged process always has a security level equal to that of the process. The process with the `P_MACREAD` privilege, though, may use this system call to find out the security level of any existing semaphore set on the system. The process must also have discretionary read access to the semaphore set.

For a detailed discussion of process privileges, see the individual system call manual pages and **`intro(2).`**

System V Shared Memory

The shared memory type of IPC allows two or more processes to share memory and, consequently, the data contained therein. This is done by allowing processes to set up access to a common virtual memory address space. This sharing occurs on a segment basis, which is memory management hardware-dependent.

A process initially creates a shared memory segment using the **`shmget`** system call. Upon creation, this process sets the overall operation permissions for the shared memory segment, sets its size in bytes, and can specify that the shared memory segment is for reference only (read-only) upon attachment. If the Enhanced Security Utilities are installed and running, the shared memory segment inherits the security level of the creating process.

If the memory segment is not specified to be for reference only, all other processes with appropriate operation permissions can read from or write to the memory segment. If the Enhanced Security Utilities are installed and running, the process must also be at the same security level as the segment.

`shmat` (shared memory attach) and **`shmdt`** (shared memory detach) can be performed on a shared memory segment.

`shmat` allows processes to associate themselves with the shared memory segment if they have permission. They can then read or write as allowed.

`shmdt` allows processes to disassociate themselves from a shared memory segment. Therefore, they lose the ability to read from or write to the shared memory segment.

The original owner/creator of a shared memory segment can relinquish ownership to another process using the **`shmctl`** system call. However, the creating process remains the creator until the facility is removed or the system is reinitialized. Other processes with permission can perform other functions on the shared memory segment using the **`shmctl`** system call.

A process can bind a shared memory segment to a section of physical memory by using the **`shmbind`** system call.

To facilitate use of shared memory by cooperating programs, a utility called **`shmdefine(1)`** is provided. Procedures for using this utility are explained in “Using shmdefine” (see p. 12-78). To assist you in creating a shared memory segment and binding it to a section of physical memory, a utility called **`shmconfig(1M)`** is also provided. Procedures for using this utility are explained in “Using shmconfig” (see p. 12-84).

Using Shared Memory

Sharing memory between processes occurs on a virtual segment basis. There is only one copy of each individual shared memory segment existing in the operating system at any time.

Before sharing of memory can be realized, a uniquely identified shared memory segment and data structure must be created. The unique identifier created is called the shared memory identifier (`shmid`); it is used to identify or refer to the associated data structure. This identifier is available to any process in the system, subject to normal access restrictions

The data structure includes the following for each shared memory segment:

- Operation permissions
- Segment size

- Segment descriptor (for internal system use only)
- PID performing last operation
- PID of creator
- Current number of processes attached
- Last attach time
- Last detach time
- Last change time

With the OS, the definition of the associated shared-memory segment data structure `shmid_ds` is as shown in Figure 12-6:

```
struct shmid_ds {
   struct ipc_perm shm_perm;   /* operation permission struct */
   int             shm_segsz;  /* size of segment in bytes */
   _VOID           *shm_pad0;/* placeholder for historical shm_amp */
   ushort_t        shm_lkcnt;/* number of times it is being locked */
   char            pad[SHM_PAD];/* expansion */
   pid_t           shm_lpid;   /* pid of last shmop */
   pid_t           shm_cpid;   /* pid of creator */
   shmatt_t        shm_nattch; /* used only for shminfo */
   ulong_t         shm_cnattch;/* used only for shminfo */
   time_t          shm_atime;  /* last shmat time */
   long            shm_pad1;   /* reserved for time_t expansion */
   time_t          shm_dtime;  /* last shmdt time */
   long            shm_pad2;   /* reserved for time_t expansion */
   time_t          shm_ctime;  /* last change time */
   long            shm_pad3;   /* reserved for time_t expansion */
   long            shm_pad4[SHM_PAD1];/* reserve area */
};
```

Figure 12-6. Definition of shmid_ds Structure

The C programming language data structure definition for the shared memory segment data structure `shmid_ds` is located in the **`sys/shm.h`** header file.

Note that the `shm_perm` member of this structure uses `ipc_perm` as a template. The `ipc_perm` data structure is the same for all IPC facilities; it is located in the **`sys/ipc.h`** header file.

The **`shmget`** system call performs two tasks:

- It gets a new shared memory identifier and creates an associated shared memory segment data structure.
- It returns an existing shared memory identifier that already has an associated shared memory segment data structure.

The task performed is determined by the value of the *key* argument passed to the **`shmget`** system call.

The *key* can be an integer that you select, or it can be an integer that you have generated by using the **ftok** subroutine. The **ftok** subroutine generates a key that is based upon a path name and identifier that you supply. By using **ftok**, you can obtain a unique key and control users' access to the key by limiting access to the file associated with the path name. If you wish to ensure that a key can be used only by cooperating processes, it is recommended that you use **ftok**. This subroutine is specified as follows:

```
    key_t ftok( path_name, id )
```

The *path_name* argument specifies a pointer to the path name of an existing file that should be accessible to the calling process. The *id* argument specifies a character that uniquely identifies a group of cooperating processes. **Ftok** returns a key that is based on the specified *path_name* and *id*. Additional information on the use of **ftok** is provided in the system manual page **stdipc(3C)**

When the Enhanced Security Utilities are installed and running, *key*s are kept on a per-level basis. *Key*s within a security level are unique; however, the same *key* may exist at different security levels. Each *key* references a different shared memory segment and data structure at each security level where the *key* exists. As mentioned before, a shared memory segment and data set inherit the security level of the creating process. While the security level of the shared memory segment cannot be changed, a process with appropriate privilege may perform multilevel operations on shared memory segments. Refer to the "Multilevel Operation On Shared Memory Segments" section of this chapter for details.

For the first task, if the *key* is not already in use for an existing shared memory identifier at the security level of the calling process and the `IPC_CREAT` flag is set in `shmflg,` a new identifier is returned with an associated shared memory segment data structure created for it provided no system-tunable parameters would be exceeded.

There is also a provision for specifying a *key* of value zero which is known as the private *key* (`IPC_PRIVATE`); when specified, a new `shmid` is always returned with an associated shared memory segment data structure created for it unless a system-tunable parameter would be exceeded. The **ipcs** command will show the *key* field for the `shmid` as all zeros.

For the second task, if a `shmid` exists for the *key* specified, the value of the existing `shmid` is returned. If it is not desired to have an existing `shmid` returned, a control command (`IPC_EXCL`) can be specified (set) in the `shmflg` argument passed to the system call. "Using shmget" discusses how to use this system call.

When performing the first task, the process that calls **shmget** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator (see "Controlling Shared Memory"). The creator of the shared memory segment also determines the initial operation permissions for it.

Once a uniquely identified shared memory segment data structure is created, **shmbind** (shared memory segment binding to physical memory), **shmctl** (shared memory control), and **shmop** (shared memory segment operations) can be used.

The **shmbind** system call allows you to bind a shared memory segment to a section of physical memory. It requires that you first reserve a section of physical memory of the desired size. See the section "Binding a Shared Memory Segment to Physical Memory" for details of the **shmbind** system call and procedures for reserving a segment of physical memory.

The **shmctl** system call permits you to control the shared memory facility in the following ways:

- by retrieving the data structure associated with a shared memory segment (IPC_STAT)
- by changing operation permissions for a shared memory segment (IPC_SET)
- by removing a particular shared memory segment from the operating system along with its associated shared memory segment data structure (IPC_RMID)
- by locking a shared memory segment in memory (SHM_LOCK)
- by unlocking a shared memory segment (SHM_UNLOCK)

See the section "Controlling Shared Memory" for details of the **shmctl** system call.

Shared memory segment operations consist of attaching and detaching shared memory segments. **shmat** and **shmdt** are provided for each of these operations (see "Operations for Shared Memory" for details of the **shmat** and **shmdt** system calls).

It is important to note that the **shmdefine(1)** and **shmconfig(1M)** utilities also allow you to create shared memory segments. See the section "Using Shared Memory Utilities" for details on the use of these utilities.

Getting Shared Memory Segments

This section describes how to use the **shmget** system call. The accompanying program illustrates its use.

Using shmget

The synopsis found in the **shmget(2)** system manual page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key, size, shmflg)
key_t   key;
int     size, shmflg;
```

Upon successful completion, this function returns the shared memory identifier (shmid) that was discussed earlier.

As declared, the process calling the **shmget** system call must supply three arguments to be passed to the formal *key*, *size*, and *shmflg* arguments.

A new shmid with an associated shared memory data structure is provided if either

- *key* is equal to IPC_PRIVATE,

or

- *key* is a unique integer and *shmflg* ANDed with `IPC_CREAT` is “true” (not zero).

If the Enhanced Security Utilities are installed, the *key* is an integer that is not yet associated with a `shmid` at the security level of the calling process; the new memory segment and its data structure inherit the security level of the creating process.

The value passed to the *shmflg* argument must be an integer-type value and will specify the following:

- operations permissions
- control fields (commands)

Access permissions determine the read/write attributes and modes determine the user/group/other attributes of the *shmflg* argument. They are collectively referred to as “operation permissions.”

Table 12-3 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Table 12-3. Operation Permissions Codes

Operation Permissions	Octal Value
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

A specific octal value is derived by adding or bitwise ORing the octal values for the operation permissions desired. That is, if read by user and read/write by others is desired, the code value would be 00406 (00400 plus 00006). The `SHM_R` and `SHM_W` constants that are located in **`<sys/shm.h>`** can be used to define read and write permission for the owner.

The flags that apply to the **`shmget`** system call are `IPC_CREAT` and `IPC_EXCL` and are defined in <**`sys/ipc.h>`**.

The OS defines additional flags that set the NUMA policy and cache policy associated with a shared memory segment on Series 6000 and Power MAXION systems. These flags are defined in <**`sys/shm.h`**>. They are described in the paragraphs that follow.

The global NUMA policy is selected for a shared memory segment by default. You may select one of the local NUMA policies by setting one of the following flags:

`SHM_LOCAL`	selects the anchored soft-local NUMA policy. Selecting a soft-local policy indicates that the pages are to be placed in local memory if possible and global memory if not
`SHM_HARD`	selects the anchored hard-local NUMA policy. Selecting a hard-local policy indicates that the pages are to be placed only in local memory. If the necessary pages are not available, the process blocks until the pages become available.

When one of these flags is set, the local memory pool used for the shared memory segment will be that which belongs to the CPU of the calling process.

If a shared memory identifier already exists for the specified key, then the NUMA policy that is specified in the *shmflg* argument will not affect the existing shared memory segment (that is, it will not cause the segment to be migrated). For additional information on NUMA policies, refer to Chapter 6, “Memory Management.”

The copyback cache policy is selected for a shared memory segment by default. You may select a cache policy for the shared memory segment by setting one of the following flags:

`SHM_NCACHE` selects the no-cache CPU cache policy, which indicates that accesses to the pages are to bypass the CPU’s data cache

CAUTION

Synchronizing instructions are <u>not</u> atomic when they operate on cache-inhibited locations.

`SHM_COPYBACK` selects the copyback CPU cache policy, which indicates that accesses to the pages are to be cached in copyback mode

In copyback mode, a CPU write transaction usually updates the data cache only; it does not immediately update memory. Later when the cache line is displaced or invalidated, the data are written to memory.

The value for *shmflg* is a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This is accomplished by adding or bitwise ORing (|) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible.

The *shmflg* value can easily be set by using the names of the flags in conjunction with the octal operation permissions value:

```
shmid = shmget (key, size, (IPC_CREAT | 0400));
shmid = shmget (key, size,
                (IPC_CREAT | IPC_EXCL | 0400));
```

The system call will attempt to return a new `shmid` if one of the following conditions is true:

- *key* is equal to `IPC_PRIVATE`.
- *key* does not already have a `shmid` associated with it and (`shmflg & IPC_CREAT`) is “true” (not zero).

If the Enhanced Security Utilities are installed and the *key* is an integer that is not yet associated with a `shmid` at the security level of the calling process, the new memory segment and its data structure inherit the security level of the creating process.

The `key` argument can be set to `IPC_PRIVATE` like this:

```
shmid = shmget(IPC_PRIVATE, size, shmflg);
```

The `SHMMNI` system tunable parameter determines the maximum number of unique shared memory segments (`shmid`s) that may be in use at any given time. If the maximum number of shared memory segments is already in use, an attempt to create an additional segment will fail. You can examine and modify the values of system tunable parameters by using the **`config(1M)`** utility. For an explanation of the procedures for using this utility, refer to the "Configuring and Building the Kernel" chapter of *System Administration Volume 2*.

`IPC_EXCL` is another control command used in conjunction with `IPC_CREAT.` It will cause the system call to return an error if a shared memory identifier already exists for the specified *key* provided. (If the Enhanced Security Utilities are installed and running, it will cause the system call to return an error if a shared memory identifier already exists at the security level of the calling process for the specified *key* provided.) This is necessary to prevent the process from thinking that it has received a new (unique) `shmid` when it has not. In other words, when both `IPC_CREAT` and `IPC_EXCL` are specified, a unique shared memory identifier is returned if the system call is successful. Any value for *shmflg* returns a new identifier if the *key* equals zero (`IPC_PRIVATE`).

The system call will fail if the value for the `size` argument is less than `SHMMIN` or greater than `SHMMAX.` These system tunable parameters specify the minimum and maximum shared memory segment sizes.

Refer to the **`shmget(2)`** manual page for the specific error conditions.

Example Program

The example program that is presented at the end of this section is a menu-driven program. It allows all possible combinations of using the `shmget` system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self explanatory. These names make the program more readable and are perfectly valid since they are local to the program.

The variables declared for this program and what they are used for are as follows:

`key`	used to pass the value for the desired `key`
`opperm`	used to store the desired operation permissions
`flags`	used to store the desired control commands (flags)
`shmid`	used for returning the message queue identification number for a successful system call or the error code (`-1`) for an unsuccessful one

`size`	used to specify the shared memory segment size
`opperm_flags`	used to store the combination from the logical ORing of the `opperm` and `flags` variables; it is then used in the system call to pass the `shmflg` argument

The program begins by prompting for a hexadecimal `key`, an octal operation permissions code, and finally for the control command combinations (flags) which are selected from a menu (lines 14-31). All possible combinations are allowed even though they might not be viable. This allows observing the errors for invalid combinations.

Next, the menu selection for the flags is combined with the operation permissions; the result is stored in the `opperm_flags` variable (lines 35-50).

A display then prompts for the size of the shared memory segment; it is stored in the `size` variable (lines 51-54).

The system call is made next; the result is stored in the `shmid` variable (line 56).

Since the `shmid` variable now contains a valid message queue identifier or the error code (`-1`), it is tested to see if an error occurred (line 58). If `shmid` equals `-1`, a message indicates that an error resulted and the external `errno` variable is displayed (line 60).

If no error occurred, the returned shared memory segment identifier is displayed (line 64).

The example program for the **`shmget`** system call follows. We suggest that you name the source program file **`shmget.c`** and the executable file **`shmget`**.

```
 1   /*This is a program to illustrate
 2   **the shared memory get, shmget(),
 3   **system call capabilities.*/
 4   #include    <sys/types.h>
 5   #include    <sys/ipc.h>
 6   #include    <sys/shm.h>
 7   #include    <errno.h>
 8   /*Start of main C language program*/
 9   main()
10   {
11       key_t key;             /*declare as long integer*/
12       int opperm, flags;
13       int shmid, size, opperm_flags;
14       /*Enter the desired key*/
15       printf("Enter the desired key in hex = ");
16       scanf("%x", &key);
17       /*Enter the desired octal operation
18         permissions.*/
19       printf("\nEnter the operation\n");
20       printf("permissions in octal = ");
21       scanf("%o", &opperm);
22       /*Set the desired flags.*/
23       printf("\nEnter corresponding number to\n");
24       printf("set the desired flags:\n");
25       printf("No flags                  = 0\n");
26       printf("IPC_CREAT                 = 1\n");
27       printf("IPC_EXCL                  = 2\n");
28       printf("IPC_CREAT and IPC_EXCL    = 3\n");
29       printf("           Flags          = ");
30       /*Get the flag(s) to be set.*/
31       scanf("%d", &flags);
32       /*Check the values.*/
33       printf ("\nkey =0x%x, opperm = 0%o, flags = %d\n",
        key, opperm, flags);
    /*Incorporate the control fields (flags) with
      the operation permissions*/
    switch (flags)
    {
    case 0:    /*No flags are to be set.*/
        opperm_flags = (opperm | 0);
        break;
    case 1:    /*Set the IPC_CREAT flag.*/
        opperm_flags = (opperm | IPC_CREAT);
        break;
    case 2:    /*Set the IPC_EXCL flag.*/
        opperm_flags = (opperm | IPC_EXCL);
        break;
    case 3:    /*Set the IPC_CREAT and IPC_EXCL flags.*/
        opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
    }
    /*Get the size of the segment in bytes.*/
    printf (“\nEnter the segment”);
    printf (“\nsize in bytes = “);
    scanf (“%d”, &size);
    /*Call the shmget system call.*/
    shmid = shmget (key, size, opperm_flags);
    /*Perform the following if the call is unsuccessful.*/
    if(shmid == -1)
    {
printf (“\nThe shmget call failed, error number = %d\n”, errno);
    }
    /*Return the shmid upon successful completion.*/
    else
        printf (“\nThe shmid = %d\n”, shmid);
    exit(0);
}
```

Controlling Shared Memory

This section describes how to use the **shmctl** system call. The accompanying program illustrates its use.

Using shmctl

The synopsis found in the **shmctl(2)** system manual page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (shmid, cmd, buf)
int shmid, cmd;
struct shmid_ds *buf;
```

The **shmctl** system call requires three arguments to be passed to it. It returns an integer value which will be zero for successful completion or `-1` otherwise.

The *shmid* variable must be a valid, non-negative, integer value. It must have been created previously by using the **`shmget`** system call.

The *cmd* argument can be replaced by one of following values:

`IPC_STAT`	return the status information contained in the associated data structure for the specified `shmid` and place it in the data structure pointed to by the `buf` pointer in the user memory area
`IPC_SET`	for the specified `shmid,` set the effective user and group identification, and operation permissions
`IPC_RMID`	remove the specified `shmid` with its associated shared memory segment data structure
`SHM_LOCK`	lock the specified shared memory segment in memory; must have appropriate privileges to perform this operation `SHM_LCCK` locks only one of two resources that are needed to provide fault-free access to memory. `SHM_LOCK` locks the pages but does not build and lock the virtual-to-physical translations to those pages. Use **`mlock(3C)`** to obtain fault-free access to a shared memory segment (refer to Chapter 6, “Memory Management,” for an explanation of this routine).
`SHM_UNLOCK`	unlock the shared memory segment from memory; must have appropriate privileges to perform this operation.

To perform an `IPC_SET` or `IPC_RMID` control command, a process must have:

- An effective user ID that is equal to that of the owner/creator of the shared memory segment

  or

- The `P_OWNER` privilege

If the Enhanced Security Utilities are installed and running, the following conditions must also be met:

- The calling process and the shared memory segment must have identical security levels, or the process must have both `P_MACREAD` and `P_MACWRITE` privileges.

Note that a shared memory segment can also be removed by using the **`ipcrm(1)`** command and specifying the **`-m`** *shmid* or the **`-M`** *shmkey* option, where *shmid* specifies the identifier for the shared memory segment and the *shmkey* argument specifies the key associated with the segment. To use this command, a process must have the same privileges as those required for performing an `IPC_RMID` control command. See the **`ipcrm(1)`** system manual page for additional information on the use of this command.

A process with the `P_SYSOPS` privilege can perform a `SHM_LOCK` or `SHM_UNLOCK` control command.

A process must have read permission to perform the `IPC_STAT` control command. To have read permission when the Enhanced Security Utilities are installed and running, the security level of the process must dominate that of the shared memory segment and the operation permissions must allow access. A process with the `P_MACREAD` privilege may override the security level restriction and perform the `IPC_STAT` control command successfully.

The details of this system call are discussed in the example program. If you need more information on the logic manipulations in this program, read “Using shmget.” It goes into more detail than would be practical for every system call.

Example Program

The example program that is presented at the end of this section is a menu-driven program. It allows all possible combinations of using the **`shmctl`** system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self explanatory. These names make the program more readable and are perfectly valid since they are local to the program.

The variables declared for this program and what they are used for are as follows:

`uid`	used to store the `IPC_SET` value for the user identification
`gid`	used to store the `IPC_SET` value for the group identification
`mode`	used to store the `IPC_SET` value for the operation permissions
`rtrn`	used to store the return integer value from the system call
`shmid`	used to store and pass the shared memory segment identifier to the system call
`command`	used to store the code for the desired control command so that subsequent processing can be performed on it
`choice`	used to determine which member for the `IPC_SET` control command is to be changed
`shmid_ds`	used to receive the specified shared memory segment identifier's data structure when an `IPC_STAT` control command is performed
`buf`	a pointer passed to the system call which locates the data structure in the user memory area where the `IPC_STAT` control command is to place its return values or where the `IPC_SET` command gets the values to set.

Note that the `shmid_ds` data structure in this program (line 16) uses the data structure of the same name located in the **`sys/shm.h`** header file as a template for its declaration.

The next important thing to observe is that although the `buf` pointer is declared to be a pointer to a data structure of the `shmid_ds` type, it must also be initialized to contain the address of the user memory area data structure (line 17).

First, the program prompts for a valid shared memory segment identifier which is stored in the `shmid` variable (lines 18-20). This is required for every **`shmctl`** system call.

Then, the code for the desired control command must be entered (lines 21-29); it is stored in the command variable. The code is tested to determine the control command for subsequent processing.

If the `IPC_STAT` control command is selected (code 1), the system call is performed (lines 39, 40) and the status information returned is printed out (lines 41-71). Note that if the system call is unsuccessful (line 139), the status information of the last successful call is printed out. In addition, an error message is displayed and the `errno` variable is printed out (lines 141). If the system call is successful, a message indicates this along with the shared memory segment identifier used (lines 143-147).

If the `IPC_SET` control command is selected (code 2), the first thing done is to get the current status information for the shared memory identifier specified (lines 88-90). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 91-96). This code is stored in the choice variable (line 97). Now, depending upon the member picked, the program prompts for the new value (lines 98-120). The value is placed in the appropriate member in the user memory area data structure, and the system call is made (lines 121-128). Depending upon success or failure, the program returns the same messages as for `IPC_STAT` above.

If the `IPC_RMID` control command (code 3) is selected, the system call is performed (lines 125-128), and the `shmid` along with its associated message queue and data structure are removed from the operating system. Note that the `buf` pointer is ignored in performing this control command and its value can be zero or NULL. Depending upon the success or failure, the program returns the same messages as for the other control commands.

If the `SHM_LOCK` control command (code 4) is selected, the system call is performed (lines 130,131). Depending upon the success or failure, the program returns the same messages as for the other control commands.

If the `SHM_UNLOCK` control command (code 5) is selected, the system call is performed (lines 133-135). Depending upon the success or failure, the program returns the same messages as for the other control commands.

The example program for the **`shmctl`** system call follows. We suggest that you name the source program file **`shmctl.c`** and the executable file **`shmctl`**.

```
 1    /*This is a program to illustrate
 2    **the shared memory control, shmctl(),
 3    **system call capabilities.
 4    */
 5    /*Include necessary header files.*/
 6    #include    <stdio.h>
 7    #include    <sys/types.h>
 8    #include    <sys/ipc.h>
 9    #include    <sys/shm.h>


10    /*Start of main C language program*/
11    main()
```

```
12    {
13        extern int errno;
14        int uid, gid, mode;
15        int rtrn, shmid, command, choice;
16        struct shmid_ds shmid_ds, *buf;
17        buf = &shmid_ds;
18        /*Get the shmid, and command.*/
19        printf(“Enter the shmid = “);
20        scanf(“%d”, &shmid);
21        printf(“\nEnter the number for\n”);
22        printf(“the desired command:\n”);
23        printf(“IPC_STAT    =  1\n”);
24        printf(“IPC_SET     =  2\n”);
25        printf(“IPC_RMID    =  3\n”);
26        printf(“SHM_LOCK    =  4\n”);
27        printf(“SHM_UNLOCK  =  5\n”);
28        printf(“Entry       =  “);
29        scanf(“%d”, &command);
30        /*Check the values.*/
31        printf (“\nshmid =%d, command = %d\n”,
32            shmid, command);
33        switch (command)
34        {
35        case 1:    /*Use shmctl() to get
36                   the data structure for
37                   shmid in the shmid_ds area pointed
38                   to by buf and then print it out.*/
39            rtrn = shmctl(shmid, IPC_STAT,
40                buf);
41            printf (“\nThe USER ID = %d\n”,
42                buf->shm_perm.uid);
43            printf (“The GROUP ID = %d\n”,
44                buf->shm_perm.gid);
45            printf (“The creator's ID = %d\n”,
46                buf->shm_perm.cuid);
47            printf (“The creator's group ID = %d\n”,
48                buf->shm_perm.cgid);
49            printf (“The operation permissions = 0%o\n”,
50                buf->shm_perm.mode);
51            printf (“The slot usage sequence\n”);
52            printf (“number = 0%x\n”,
53                buf->shm_perm.seq);
54            printf (“The key= 0%x\n”,
55                buf->shm_perm.key);
56            printf (“The segment size = %d\n”,
57                buf->shm_segsz);
58            printf (“The pid of last shmop = %d\n”,
59                buf->shm_lpid);
60            printf (“The pid of creator = %d\n”,
61                buf->shm_cpid);
62            printf (“The current # attached = %d\n”,
63                buf->shm_nattch);
64            printf(“The last shmat time = %ld\n”,
65                buf->shm_atime);
66            printf(“The last shmdt time = %ld\n”,
67                buf->shm_dtime);
68            printf(“The last change time = %ld\n”,
69                buf->shm_ctime);
70            break;/* Lines 71 - 85 deleted */


86        case 2:    /*Select and change the desired
87                      member(s) of the data structure.*/
88            /*Get the original data for this shmid
89                  data structure first.*/
```

```
            rtrn = shmctl(shmid, IPC_STAT, buf);
            printf("\nEnter the number for the\n");
            printf("member to be changed:\n");
            printf("shm_perm.uid   = 1\n");
            printf("shm_perm.gid   = 2\n");
            printf("shm_perm.mode  = 3\n");
            printf("Entry          = ");
            scanf("%d", &choice);
            switch(choice){
            case 1:
                printf("\nEnter USER ID = ");
                scanf ("%d", &uid);
                buf->shm_perm.uid = uid;
                printf("\nUSER ID = %d\n",
                    buf->shm_perm.uid);
                break;
            case 2:
                printf("\nEnter GROUP ID = ");
                scanf("%d", &gid);
                buf->shm_perm.gid = gid;
                printf("\nGROUP ID = %d\n",
                    buf->shm_perm.gid);
                break;
            case 3:
                printf("\nEnter MODE in octal = ");
                scanf("%o", &mode);
                buf->shm_perm.mode = mode;
                printf("\nMODE = 0%o\n",
                    buf->shm_perm.mode);
                break;
            }
            /*Do the change.*/
            rtrn = shmctl(shmid, IPC_SET,
                buf);
            break;
        case 3:    /*Remove the shmid along with its
                      associated
                      data structure.*/
            rtrn = shmctl(shmid, IPC_RMID, (struct shmid_ds *) NULL);
            break;
        case 4: /*Lock the shared memory segment*/
            rtrn = shmctl(shmid, SHM_LOCK, (struct shmid_ds *) NULL);
            break;
        case 5: /*Unlock the shared memory
                      segment.*/
            rtrn = shmctl(shmid, SHM_UNLOCK, (struct shmid_ds *)NULL);
            break;
        }
        /*Perform the following if the call is unsuccessful.*/
        if(rtrn == -1)
        {
      printf ("\nThe shmctl call failed, error number = %d\n", errno);
        }
        /*Return the shmid upon successful completion.*/
        else
            printf ("\nShmctl was successful for shmid = %d\n",
                shmid);
        exit (0);
    }
```

Binding a Shared Memory Segment to Physical Memory

The OS allows you to bind a shared memory segment to a section of physical memory. The procedures for doing so are as follows:

1. Define a reserved section of physical memory.
2. Create a shared memory segment, and bind it to a section of physical memory.
3. Attach the shared memory segment to the user's virtual address space
4. Detach the shared memory segment from the user's virtual address space

Procedures for defining a reserved section of physical memory are explained in "Reserving Physical Memory."

You can create a shared memory segment and bind it to a section of physical memory by:

- Using the **`shmget(2)`** system call to obtain an identifier for a shared memory segment and the **`shmbind(2)`** system call to bind the segment to a particular section of physical memory. It is recommended that you use this method if you are creating a segment that will be used only by the parent process and its children. Procedures for using these system calls are explained in "Using shmget and shmbind."

  or

- Invoking the **`shmconfig(1M)`** utility with the appropriate parameters. If you wish to create a segment that will be accessed by a controlled group of users, it is recommended that you use this utility. Procedures for using this utility are explained in "Using Shared Memory Utilities."

You can attach a shared memory segment to and detach it from the user's virtual address space by using the **`shmat`** and **`shmdt`** system calls. Procedures for using these system calls are explained in "Operations for Shared Memory."

Reserving Physical Memory

If you wish to bind a shared memory segment to a section of physical memory in the local or global memory pool, it is required that you first reserve the section of physical memory. Note that it is <u>not</u> necessary to reserve memory if the physical address resides in I/O memory space. There are several methods for reserving a section of physical memory.

One method allows you to reserve a section of physical memory statically; it requires that you first place an entry in an array in the configuration-dependent **`space.c`** file and then rebuild the kernel and reboot the system. This method is explained in "Initializing the res_sects Array." This method is recommended if you wish the section of physical memory to be reserved at an exact starting address and you wish to ensure that the section is reserved each time the system is rebooted.

The other methods allow you to reserve a section of physical memory dynamically—that is, on a running system. With these methods, it is not necessary to rebuild the kernel and reboot the system. These methods include use of the **`physmalloc(3C)`** library routine, the **physconfig(1M)** utility, or the **shmconfig(1M)** utility. Procedures for using these methods are explained in “Using physmalloc” (see p. 12-69), “Using physconfig” (see p. 12-71), and “Using shmconfig” (see p. 12-84).

Initializing the res_sects Array

To reserve a section of physical memory statically, you must initialize the `res_sects[]` array in the **/etc/conf/pack.d/mm/space.c** file. Your entry will describe the starting address and the desired length of the reserved section of memory. Initially the `res_sects[]` array appears as follows:

```
struct res_sect res_sects[] = {
/*   r_start,   r_len,   r_flags */
     { 0, 0, 0 } /* This must be the last line, DO NOT change it.*/
};
```

For each section of physical memory that you wish to reserve, place an entry in the `res_sects[]` array. The `r_start` field specifies the starting physical address, and the `r_len` field specifies the length in bytes. The `r_flags` field must always be zero.

To assist you in determining appropriate values for your entry, the **space.c** file provides the following examples:

```
/* struct res_sect res_sects[] = {
* {0x1000000, 0x20000,0 },
* {0x1000000, 0x40000,0 },
* {0x1000000, 0x80000,0 },
* { 0, 0, 0 }
* };
* reserves:
*    0x01000000 - 0x0101ffff (128 Kb),
*    0x01040000 - 0x0107ffff (256 Kb),
*    0x01080000 - 0x010fffff (512 Kb)
*/
```

These examples assume a system with at least 32 megabytes of global memory and a running kernel that occupies no more than the first 16 megabytes of memory. Note that the address and length values presented here are simply examples and may not be appropriate for every system. Multiple sections of physical memory may be reserved by adding additional lines.

After changing the **space.c** file, you must rebuild the kernel using **idbuild(1M)** and reboot your system before the changes take effect.

Using physmalloc

The **physmalloc(3C)** library routine allows the calling process to reserve a section of physical memory.

Note that to use this routine, the calling process must have the `P_PLOCK` privilege.

The specifications required for making the **physmalloc** call are as follows:

```
#include <sys/types.h>
#include <sys/physmem.h>

int physmalloc (floor, ceiling, len, rpaddr)
```

```
paddr_t floor;
paddr_t ceiling;
off_t len;
paddr_t *rpaddr;
```

The arguments are defined as follows:

floor	the desired starting address of the section of physical memory to be reserved. This address may reside in local or global memory. If it resides in global memory, the value specified by *ceiling* must also reside within global memory. If it resides in a local memory pool, the value specified by *ceiling* must reside in the same local memory pool.
ceiling	the highest desired ending address of the section of physical memory to be reserved
len	the length in bytes of the section of physical memory to be reserved
rpaddr	a pointer to a location to which the actual starting physical address of the reserved section of memory is returned

The **physmalloc** routine will try to reserve a section of memory that is *len* bytes in length within the boundaries defined by the values of *floor* and *ceiling*. If it is successful, the resulting reserved section will be aligned with a page boundary, be *len* bytes in length, and reside somewhere within the boundaries specified by *floor* and *ceiling.*

Upon successful completion, **physmalloc** returns a value of zero, and *rpaddr* contains the actual starting physical address of the reserved section of memory. A return value of **-1** indicates that an error has occurred; **errno** is set to indicate the error. Refer to the **physmalloc(3C)** system manual page for a listing of the types of errors that may occur.

To free a section of physical memory reserved with **physmalloc**, use the **physfree(3C)** library routine. Note that to use this routine, the calling process must have the `P_PLOCK` privilege.

The specifications required for making this call are as follows:

```
#include <sys/types.h>
#include <sys/physmem.h>

int physfree(paddr)

paddr_t paddr;
```

The argument is defined as follows:

paddr	the starting address of the reserved section of memory to be freed

In order to free a reserved section, there must be no bindings between shared memory segments and any address within the reserved area.

Upon successful completion, **`physfree`** returns a value of zero. A return value of **-1** indicates that an error has occurred; **`errno`** is set to indicate the error. Refer to the **`physmalloc(3C)`** system manual page for a listing of the types of errors that may occur.

Using physconfig

The **`/usr/sbin/physconfig`** command allows you to perform the following functions: (1) view the sections of physical memory that have been reserved; (2) reserve a section of physical memory; and (3) free a section of physical memory that has been reserved by using **`physconfig`** or **`shmconfig(1M)`**. Procedures for performing these functions are explained in the sections that follow.

To display sections that have been reserved, use the following format:

`/usr/sbin/physconfig -v`

This option is used to display sections of memory that have been reserved statically or dynamically. It displays the starting physical address of the section, the length, and the number of bindings between shared memory segments and any address within the section. Sample output follows:

```
Static Reserves:
Starting Address              Length                Bindings
----------------       --------------------         --------
0x10000000             128 KB (   32 pages)             2

Dynamic Reserves:
Starting Address              Length                Bindings
----------------       --------------------         --------
0x0072b000              64 KB (   16 pages)             1
0x0073b000              64 KB (   16 pages)             0
```

To reserve a section of physical memory, use the following format:

`/usr/sbin/physconfig -r -s` *size* **`[-b`** *begin***`] [-e`** *end***`]`**

Note that to use **`physconfig`** to reserve memory, you must have the `P_PLOCK` privilege.

Options for reserving memory are described in Table 12-4.

Table 12-4. Options Specified for Reserving Memory

Option	Description
`-r`	Specifies that you wish to reserve a section of physical memory When specifying this option, you must also specify the *size* argument; the *begin* and *end* arguments are optional.
`-s` *size*	Specifies the size in bytes of the section of physical memory to be reserved

Table 12-4. Options Specified for Reserving Memory (Cont.)

Option	Description
-b *begin*	Specifies the desired starting physical address of the reserved section of memory. This address may reside in local or global memory. If you do not this specify this option, the kernel will use the starting address of global memory as the starting address of the reserved section. It will then try to reserve the first available contiguous section of memory of *size* bytes that it finds in global memory. If the reserved section of memory must start at a certain physical address, you must also specify the **-e***end* option. The address specified by *end* should be equal to the value of *begin* plus the value of *size*.
-e *end*	Specifies the desired ending address of the section of physical memory to be reserved. This address must reside in the same memory pool as the starting physical address of the section. If you do not specify this option, the system will consider any physical address that is higher than *begin* and within the same memory pool as *begin* as memory that can satisfy the reservation request. Note that if you specify this option, you must also specify the **-b***begin* option.

It is recommended that the address range specified by *begin* and *end* be larger than the value of *size* in case the kernel cannot reserve a section of memory starting at *begin.*

The following example shows how to invoke **physconfig** to reserve a one-megabyte section of physical memory that resides somewhere between address 0x1000000 and 0x1C00000 on a Model HN6800 system In this example, it is assumed that the system has 32 MB of global memory.

```
physconfig -r -s 0x100000 -b 0x1000000 -e 0x1C00000
```

NOTE

Do not attempt to reserve a section in the last two megabytes of global memory. The kernel allocates space there for the console driver. Refer to the *HN6800 Architecture Manual* for the memory map for the Model HN6800 system.

Use the following format to free a section of physical memory that has been reserved by using **physconfig** or **shmconfig(1M)**:

/usr/sbin/physconfig -d *paddr*

The *paddr* argument is defined as follows:

paddr	the starting physical address of the section of reserved memory that is to be freed

Note that to use **`physconfig`** to free memory, you must have the `P_PLOCK` privilege.

In order to free a reserved section, there must be no bindings between shared memory segments and any address within the reserved area. Use the **`-v`** option to determine the starting physical addresses of the reserved sections and the number of bindings to each.

Using shmget and shmbind

The **`shmget(2)`** system call is invoked first to create a shared memory segment. The specification for making this call is as follows:

`int shmget(`*key*`, `*size*`, `*shmflg*`)`

Upon successful completion of the call, a shared memory segment of *size* bytes is created, and an identifier for the segment is returned. Complete information on the use of this call is provided in "Getting Shared Memory Segments" (p. 12-57).

After you have created a shared memory segment, you can bind it to a section of physical memory by using the **`shmbind(2)`** system call. Note that to use this call, you must have the `P_SHMBIND` privilege.

The section of physical memory is defined by its starting address and the size of the shared memory segment to which it is being bound. The starting address must be aligned with a page boundary. The size of the shared memory segment has been established by specifying the *size* argument on the call to **`shmget`**. If you have created a shared memory segment of 1024 bytes, for example, and you wish to bind it to a section of physical memory that starts at location 0x2000000 (hexadecimal representation), the bound section of physical memory will include memory locations 0x2000000 <u>through</u> 0x2000BFF.

The specifications required for making the call to **`shmbind`** are as follows:

```
int shmbind(shmid, paddr)

int     shmid;
paddr_t paddr;
```

Arguments are defined as follows:

shmid	the identifier for the shared memory segment that you wish to bind to a section of physical memory
paddr	the starting physical address of the section of memory to which you wish to bind the specified shared memory segment

Operations for Shared Memory

This section describes how to use the **`shmat`** and **`shmdt`** system calls. The accompanying program illustrates their use.

Using Shared Memory Operations: shmat and shmdt

The synopsis found in the **shmop(2)** system manual page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void *shmat (shmid, shmaddr, shmflg)
int      shmid;
void     *shmaddr;
int      shmflg;

int shmdt (shmaddr)
void     *shmaddr;
```

Attaching a Shared Memory Segment

The **shmat** system call requires three arguments to be passed to it. It returns a character pointer value. Upon successful completion, this value will be the address in memory where the process is attached to the shared memory segment and when unsuccessful, the value will be -1.

The *shmid* argument must be a valid, non-negative, integer value. It must have been created previously by using the **shmget** system call.

The *shmaddr* argument can be zero or user supplied when passed to the **shmat** system call. If it is zero, the operating system selects the address where the shared memory segment will be attached. If it is user-supplied, the address must be a valid address within the program's address space.

The following illustrates some typical address ranges.

```
0xc00c0000
0xc00e0000
0xc0100000
0xc0120000
```

Allowing the operating system to select addresses improves portability.

The shmflg argument is used to pass the SHM_RND and SHM_RDONLY flags to the **shmat** system call.

Detaching Shared Memory Segments

The **shmdt** system call requires one argument to be passed to it. It returns an integer value which will be zero for successful completion or -1 otherwise.

Further details on **shmat** and **shmdt** are discussed in the example program. If you need more information on the logic manipulations in this program, read "Using shmget." It goes into more detail than would be practical for every system call.

Example Program

The example program that is presented at the end of this section is a menu-driven program. It allows all possible combinations of using the **`shmat`** and **`shmdt`** system calls to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self explanatory. These names make the program more readable and are perfectly valid since they are local to the program.

The variables declared for this program and what they are used for are as follows:

`addr`	used to store the address of the shared memory segment for the **`shmat`** and **`shmdt`** system calls and to receive the return value from the **`shmat`** system call
`laddr`	used to store the desired attach/detach address entered by the user
`flags`	used to store the codes of the `SHM_RND` or `SHM_RDONLY` flags for the **`shmat`** system call
`i`	used as a loop counter for attaching and detaching
`attach`	used to store the desired number of attach operations
`shmid`	used to store and pass the desired shared memory segment identifier
`shmflg`	used to pass the value of flags to the **`shmat`** system call
`retrn`	used to store the return values from the **`shmdt`** system call
`detach`	used to store the desired number of detach operations

This example program combines both the **`shmat`** and **`shmdt`** system calls. The program prompts for the number of attachments and enters a loop until they are done for the specified shared memory identifiers. Then, the program prompts for the number of detachments to be performed and enters a loop until they are done for the specified shared memory segment addresses.

shmat

The program prompts for the number of attachments to be performed, and the value is stored at the address of the attach variable (lines 19-23).

A loop is entered using the attach variable and the `i` counter (lines 23-72) to perform the specified number of attachments.

In this loop, the program prompts for a shared memory segment identifier (lines 26-29); it is stored in the `shmid` variable (line 30). Next, the program prompts for the address where the segment is to be attached (lines 32-36); it is stored in the `laddr` variable (line 37) and converted to a pointer (line 39). Then, the program prompts for the desired flags to be used for the attachment (lines 40-47), and the code representing the flags is stored in the `flags` variable (line 48). The `flags` variable is tested to determine the code to be stored for the `shmflg` variable used to pass them to the **`shmat`** system call (lines 49-60). The system call is executed (line 63). If successful, a message stating so is displayed along with the attach address (lines 68-70). If unsuccessful, a message stating so is displayed and the error code is displayed (line 65). The loop then continues until it finishes.

shmdt

After the attach loop completes, the program prompts for the number of detach operations to be performed (lines 73-77) and the value is stored in the detach variable (line 76).

A loop is entered using the detach variable and the `i` counter (lines 80-98) to perform the specified number of detachments.

In this loop, the program prompts for the address of the shared memory segment to be detached (lines 81-85); it is stored in the `laddr` variable (line 86) and converted to a pointer (line 88). Then, the **`shmdt`** system call is performed (line 89). If successful, a message stating so is displayed along with the address that the segment was detached from (lines 95, 96). If unsuccessful, the error number is displayed (line 92). The loop continues until it finishes.

The example program for the **`shmop`** system calls follows. We suggest that you name the source program file **`shmop.c`** and the executable file **`shmop`**.

```
 1    /*This is a program to illustrate
 2    **the shared memory operations, shmop(),
 3    **system call capabilities.
 4    */
 5    /*Include necessary header files.*/
 6    #include    <stdio.h>
 7    #include    <sys/types.h>
 8    #include    <sys/ipc.h>
 9    #include    <sys/shm.h>
10    /*Start of main C language program*/
11    main()
12    {
13        extern int errno;
14        void *addr;
15        long laddr;
16        int flags, i, attach;
17        int shmid, shmflg, retrn, detach;
18        /*Loop for attachments by this process.*/
19        printf(“Enter the number of\n”);
20        printf(“attachments for this\n”);
21        printf(“process (1-4).\n”);
22        printf(“       Attachments = “);
23        scanf(“%d”, &attach);
24        printf(“Number of attaches = %d\n”, attach);
25        for(i = 1; i <= attach; i++) {
26            /*Enter the shared memory ID.*/
27            printf(“\nEnter the shmid of\n”);
28            printf(“the shared memory segment to\n”);
29            printf(“be operated on = “);
30            scanf(“%d”, &shmid);
31            printf(“\nshmid = %d\n”, shmid);
32            /*Enter the value for shmaddr.*/
33            printf(“\nEnter the value for\n”);
34            printf(“the shared memory address\n”);
35            printf(“in hexadecimal:\n”);
36            printf(“           Shmaddr = “);
```

```
        scanf(“%lx”, &laddr);
        addr = (void*) laddr;
        printf(“The desired address = 0x%lx\n”, (long)addr);
        /*Specify the desired flags.*/
        printf(“\nEnter the corresponding\n”);
        printf(“number for the desired\n”);
        printf(“flags:\n”);
        printf(“SHM_RND                = 1\n”);
        printf(“SHM_RDONLY             = 2\n”);
        printf(“SHM_RND and SHM_RDONLY = 3\n”);
        printf(“            Flags      = “);
        scanf(“%d”, &flags);
        switch(flags)
        {
        case 1:
            shmflg = SHM_RND;
            break;
        case 2:
            shmflg = SHM_RDONLY;
            break;
        case 3:
            shmflg = SHM_RND | SHM_RDONLY;
            break;
        }
        printf(“\nFlags = 0%o\n”, shmflg);
        /*Do the shmat system call.*/
        addr = shmat(shmid, addr, shmflg);
        if(addr == (char*) -1) {
            printf(“\nShmat failed, error = %d\n”, errno);
        }
        else {
            printf (“\nShmat was successful\n”);
            printf(“for shmid = %d\n”, shmid);
            printf(“The address = 0x%lx\n”, (long)addr);
        }
    }
    /*Loop for detachments by this process.*/
    printf(“Enter the number of\n”);
    printf(“detachments for this\n”);
    printf(“process (1-4).\n”);
    printf(“       Detachments = “);
    scanf(“%d”, &detach);
    printf(“Number of attaches = %d\n”, detach);
    for(i = 1; i <= detach; i++) {
        /*Enter the value for shmaddr.*/
        printf(“\nEnter the value for\n”);
        printf(“the shared memory address\n”);
        printf(“in hexadecimal:\n”);
        printf(“            Shmaddr = “);
        scanf(“%lx”, &laddr);
        addr = (void*) laddr;
        printf(“The desired address = 0x%lx\n”, (long)addr);
        /*Do the shmdt system call.*/
        retrn = shmdt(addr);
        if(retrn == -1)  {
            printf(“Error = %d\n”, errno);
        }
        else {
            printf (“\nShmdt was successful\n”);
            printf(“for address  = 0x%lx\n”, (long)addr);
        }
    }
}
```

Using Shared Memory Utilities

The OS provides two utilities that facilitate use of shared memory segments. The **shmdefine(1)** utility allows you to create one or more shared memory segments that are to be used by cooperating programs. The **shmconfig(1M)** utility allows you to create a shared memory segment and bind it to a section of physical memory. The sections that follow provide detailed explanations of the procedures for using **shmdefine** and **shmconfig**, respectively.

Using shmdefine

The **shmdefine** utility is designed to facilitate the use of shared memory by a set of cooperating programs. Although you may have a number of programs that will cooperate in using one or more shared memory segments, it is necessary to invoke the utility only once. Because **shmdefine** produces object files that must be linked to the source object file, you must invoke it prior to linking

The format for executing the **shmdefine** utility is as follows:

shmdefine [-b *base_name*] [**-U**] [*files*]

The object files that **shmdefine** produces are an initialization file and a linker command file. The initialization file contains an executable function that accesses shared memory services at program start-up time. The default name for this file is **shm_init.sm.c**. The linker command file describes the shared memory segments to the linker. The default name for this file is **shm_init.sm.ld**. The base name of each of these files is **shm_init**. The **-b** option enables you to supply a base name of your choice

The initialization file also takes care of data initialization. Data initialization of variables that are associated with a shared memory segment via **shmdefine** occurs when a newly executed program attaches to the corresponding shared memory segment--if and only if it is the only program currently attached to that segment.

The **shmdefine** utility converts uppercase Fortran COMMON block names to lowercase. The **-U** option enables you to prevent this conversion.

Input to the **shmdefine** utility defines the shared memory segment or segments that are to be used by cooperating programs. You may use the standard input to define the segments, or you may specify one or more files that contain the definitions. Although input in either case may be free-form, the general format for defining a shared memory segment is as follows:

SHARED REGION *segment_name*

[*attribute1, attribute2, ...*]
variable1, variable2, ...

END SHARED REGION

Note that blanks, tabs, and newlines are recognized only as separators. The hash character (#) can be used to indicate that the rest of the line is a comment.

Attributes that can be specified are presented in Table 12-5.

.

Table 12-5. Attributes

Attribute	Purpose
ADDRESS	Enables you to specify a starting virtual address for the shared memory segment
IPC	Enables you to set the control flags for the segment
SHM_LOCAL	Enables you to set the NUMA policy for the shared memory segment to the anchored soft-local policy. A soft-local policy allows pages to be allocated from global memory when pages are not available for allocation from the local memory pool. This option has no effect on systems without local memory. For complete information on NUMA policies, refer to Chapter 6, “Memory Management.”
SHM_HARD	Enables you to set the NUMA policy for the shared memory segment to the anchored hard-local policy. A hard-local policy causes a process to wait for local memory pages to become available if the pages cannot be allocated from the local memory pool when needed. This option has no effect on systems without local memory. For complete information on NUMA policies, refer to Chapter 6, “Memory Management.”
KEY	Enables you to specify a user-chosen identifier for the segment
MODE	Enables you to set the permissions that are associated with the segment
SHM_RDONLY	Enables you to prevent a process from writing to the segment

Variables must be either Fortran COMMON blocks or C external variables. These variables can be associated with Ada variables via the implementation-defined pragma **interface_shared_object** in the Ada source program. If you are using the Concurrent Fortran compiler, COMMON blocks must be declared VOLATILE in the Fortran source program. If you are using the Concurrent C compiler, external variables must be declared with the type qualifier **volatile** in the C source program. The volatile declaration informs the compiler that the values of the variables may be modified in a way that is unknown to the compiler. It is important to note that space in the shared memory segment is allocated to variables in the same order in which the variables are specified in the input to **`shmdefine`**.

Additional information that is needed to specify attributes and variables is provided in the system manual page **`shmdefine(1)`**.

Making provisions for programs to share data requires the following steps:

1. If you wish to bind the shared memory segment to a particular section of physical memory, add a line with the appropriate parameters to the **shmconfig** script in the **/etc/init.d** directory. Procedures for doing so are presented in “Using shmconfig” (p. 12-84).
2. Create source programs, and include in them a volatile type declaration for each program variable that is to reside in shared memory.
3. Create the **shmdefine** input file(s).
4. Execute **shmdefine** with the desired options.
5. Compile the initialization output file that is produced by **shmdefine**.
6. Compile and link the source programs with the **shmdefine** initialization object file and the **shmdefine** link command output file.

Use of these steps to enable a C program, a Fortran program, and two Ada programs to cooperate in using a shared memory segment is illustrated by the C shell dialogue that follows. When executed, the C program named **generate** places data into a shared memory segment; the Fortran program named **process** performs a computation on each item of data stored in the segment; the Ada program named **init** initializes the **iready** and **oready** variables; and the Ada program named **output** writes the result of each computation to the standard output.

These programs do not require that the shared memory segment be bound to a section of physical memory; therefore, Step 1 is not required. To perform Step 2, create the C, Fortran, and Ada source programs, using a text editor of your choice. The following listing shows the files that have been created:

```
% ls -C
generate.c   output.a   process.f
% cat generate.c
 /* This program creates 10 integer values and passes the
     data to the cooperating programs using the shared memory
     structure sm_data.
*/

 #include <stdio.h>

 volatile struct sm_data {
   int   ain, aout ;
   int    iready, oready ;
} shared_data__ ;

 void main () {
   int    i ;
   int    accum = 2 ;

    for (i = 1 ; i <= 10 ; ++i) {
      while (shared_data__.iready > 0) {
         sleep(1) ;
      }
      shared_data__.ain = accum ;
      shared_data__.iready = i ;
      accum *= 2 ;
   }
}
% cat process.f
C   This program processes the input data and places the results of
C   the calculations in another shared memory segment for output.  On
C   input, it waits for the iready variable to be equal to its count
C   of the data.  When it has processed that datum, it negates the
C   iready variable to tell the input program it is ready for another
C   one. A similar scheme is used for communicating to the output
C   program.

      PROGRAM process
      COMMON /shared_data/ ain, aout, iready, oready
      INTEGER ain, aout
      INTEGER iready, oready
      VOLATILE shared_data
      INTEGER   i

       DO i = 1,10
         DO WHILE (iready .NE. i .OR. oready .GE. 0)
            CALL sleep(1)
         END DO
         aout =  - ain
         iready = - iready
         oready = i
      END DO
      END
```

```
% cat output.a
---- This file contains the source for two programs.  ”Init”
---- initializes the iready and oready variables.  ”Output”
---- writes the results from the process program.  It waits
---- for the oready variable to be equal to its count of the
---- the data.  When that happens, it writes the results and
---- negates the oready variable.

package external_data is
----
     type data_items is new integer range -10 .. 10 ;
     subtype data_item_id is data_items range 1..10 ;

    type common_block is
      record
         ain, aout : integer ;
         iready, oready : data_items ;
      end record ;
   shared_data : common_block ;

    pragma interface_shared_object (shared_data, ”shared_data__”) ;
----
end external_data ;

with external_data ;
procedure init is
   use external_data ;
begin
   shared_data.iready := data_items’first ;
   shared_data.oready := data_items’first ;
end init ;

with external_data ;
with text_io ;
procedure output is
----
   use text_io ;
   use external_data ;
   package int_io is new integer_io (data_items) ;
   package item_io is new integer_io (integer) ;
----
begin
----
     for id in data_item_id loop
   ----
      ---- Wait for the [next] data item
      ---- to be ready for output.
      while (shared_data.oready /= id) loop
         delay (1.0) ;
      end loop ;

       ---- Print the data.
      put (”result ”) ;
      int_io.put (id) ;
      put (” = ”) ;
      item_io.put (shared_data.aout) ;
      new_line ;

      ---- Inform ”process” that the data item has been output.
      shared_data.oready := -shared_data.oready ;
   ----
   end loop ;
----
end output ;
```

To perform Step 3, create the **shmdefine** input file, using a text editor of your choice. Specify the **KEY** attribute with a path name to ensure that a unique identifier for the shared memory segment is obtained and that access to the segment is limited to the cooperating programs. Specify the Fortran COMMON block **shared_data** as the variable. Display the input file:

```
% cat shmdef
SHARED REGION input_output
   KEY="./generate.c"
   Fortran COMMON shared_data
END SHARED REGION
```

To perform Step 4, execute **shmdefine** using **shmdef** as the input file. Also specify the **-b** option to supply **shmdef** as the base name for the object files produced by the utility.

```
% shmdefine -b shmdef shmdef
```

Display a listing of your files. Note that the listing now includes the **shmdef** input file that you have created and the initialization and linker command files that **shmdefine** has produced.

```
% ls -C
generate.c    process.f     shmdef.sm.c
output.a      shmdef        shmdef.sm.ld
```

To perform Step 5, compile the initialization file by invoking the C compiler. Note that a subsequent listing of your files will include the object file produced by the compiler.

```
% hc -c shmdef.sm.c
% ls -C
generate.c    process.f     shmdef.sm.c   shmdef.sm.o
output.a      shmdef        shmdef.sm.ld
```

To perform Step 6, compile and link the source programs. To compile and link the program to generate the data, invoke the C compiler, and specify the initialization object file and the link command file. Note that a subsequent listing of your files will include the executable file **generate**.

```
% hc -o generate generate.c shmdef.sm.o -Wl,-M shmdef.sm.ld
% ls -C
generate*     output.a      shmdef        shmdef.sm.ld
generate.c    process.f     shmdef.sm.c   shmdef.sm.o
```

To compile and link the program to process the data, invoke the Fortran compiler, and specify the initialization object file and the link command file. Note that a subsequent listing of your files will include the executable file **process**.

```
% hf77 -o process -M shmdef.sm.ld process.f shmdef.sm.o
process.f:
% ls -C
generate*     output.a      process.f     shmdef.sm.c   shmdef.sm.o
generate.c    process*      shmdef        shmdef.sm.ld
```

To compile and link the programs to initialize the **iready** and **oready** variables and write the results of the computations, invoke the Ada compiler, and specify the initialization object file and the link command file. Note that a subsequent listing of your files will include the executable files **output** and **init** as well as files generated by the Ada compiler (**GVAS_table**, **ada.lib**, and **gnrx.lib**).

```
% /usr/hapse/bin/a.mklib
% /usr/hapse/bin/ada output.a
% /usr/hapse/bin/a.ld -o output output shmdef.sm.o shmdef.sm.ld
% /usr/hapse/bin/a.ld -o init   init   shmdef.sm.o shmdef.sm.ld
% ls -C
GVAS_table    generate.c    output*       process.f     shmdef.sm.ld
ada.lib       gnrx.lib      output.a      shmdef        shmdef.sm.o
generate*     init*         process*      shmdef.sm.c
```

You are now ready to run the programs. Note that each program performs its operations asynchronously. The appearance of the prompt (%) in the midst of the listing of output data, for example, indicates that **generate** has completed execution prior to output.

```
% init
% output &
[1] 5515
% process &
[2] 5526
% generate
RESULT    1 =       -2
RESULT    2 =       -4
RESULT    3 =       -8
RESULT    4 =       -16
RESULT    5 =       -32
RESULT    6 =       -64
RESULT    7 =       -128
RESULT    8 =       -256
RESULT    9 =       -512
% RESULT    10 =      -1024
[2] + Done                 process &
[1] + Done                 output &
```

Following the link editing of a program, shared variables in a section are undefined. In programs created with the aid of the **shmdefine** utility, shared variables that were initialized at compile time will have these initial values when control is transferred to the main procedure of the program. If multiple processes are simultaneously attached to a shared memory segment, however, only the first attached process will contribute its initial values.

If you wish to compile a C or Fortran program, it is important to note that the Concurrent C and Fortran compilers are available on all systems. The C compiler is called either **hc** or **cc**. The Fortran compiler is called either **hf77** or **f77**.

For information specific to the C programming language, refer to the *Concurrent C Reference Manual*. For information specific to Fortran and the use of shared memory, refer to the *hf77 Fortran Reference Manual*. For information specific to Ada and the use of shared memory, refer to the *HAPSE Reference Manual*.

Using shmconfig

The **/usr/sbin/shmconfig** command has been developed to assist you in creating a shared memory segment that is associated with a certain key and in binding it to a particular section of physical memory. This command also allows you to reserve the section of physical memory to which you wish a shared memory segment to be bound.

Options that you can specify with the command make it possible for you to create shared memory segments that are located in global memory, local memory, or I/O memory. If you create a segment that is to lie in global or local memory, you can specify whether its pages, when resident, can be located anywhere in the corresponding memory space or must be bound to a particular range of addresses in that space. If you create a segment that is to lie in I/O memory, you must specify the range of addresses to which it is to be bound.

If you do not wish to bind the segment to a particular section of physical memory, use the following format to specify the **`/usr/sbin/shmconfig`** command:

`/usr/sbin/shmconfig` {**`-L`***cpu* | **`-H`***cpu*} **`-s`***size* [**`-u`***user*] [**`-g`***group*] [**`-m`***mode*] *key*

Options are described in Table 12-6. .

Table 12-6. Options Specified for a Virtual Segment

Option	Description
`-L`*cpu*	Specifies that the NUMA policy for the shared memory segment is to be the anchored soft-local policy. A soft-local policy allows pages to be allocated from global memory when pages are not available for allocation from the local memory pool. This option has no effect on systems without local memory. The value of *cpu* can range from **0** to **7**, where the number specifies the CPU ID of the processor from whose local memory the segment's pages are to be allocated. For complete information on NUMA policies, refer to Chapter 6, "Memory Management."
`-H`*cpu*	Specifies that the NUMA policy for the shared memory segment is to be the anchored hard-local policy. A hard-local policy causes a process to wait for local memory pages to become available if the pages cannot be allocated from the local memory pool when needed. This option has no effect on systems without local memory. The value of *cpu* can range from **0** to **7**, where the number specifies the CPU ID of the processor from whose local memory the segment's pages are to be allocated. For complete information on NUMA policies, refer to Chapter 6, "Memory Management."
`-s`*size*	Specifies the size of the segment in bytes.
`-u`*user*	Specifies the login name of the owner of the shared memory segment
`-g`*group*	Specifies the name of the group to which group access to the segment is applicable.
`-m`*mode*	Specifies *mode* as the set of permissions governing access to the shared memory segment. You must use the octal method to specify the permissions.

If you do wish the segment to be bound to a specified section of physical memory, use the following format:

`/usr/sbin/shmconfig` {**`-L`***cpu* | **`-H`***cpu* } **`-p`***paddr* **`-s`***size* [**`-u`***user*] [**`-g`***group*] \
[**`-m`***mode*] *key*

Note that you must have the `P_SHMBIND` privilege to perform this operation.

Use this format in the following circumstances:

- If the physical address range has been dynamically reserved with **`physmalloc(3C)`** or **`physconfig(1M)`**
- If the physical address range has been statically reserved in the kernel
- If the physical address range resides in I/O memory space

Options are described in Table 12-7..

Table 12-7. Options Specified for a Bound Segment

Option	Description
`-L`*cpu*	Specifies that the NUMA policy for the shared memory segment is to be the anchored soft-local policy. A soft-local policy causes pages to be allocated from global memory when pages are not available for allocation from the local memory pool. This option has no effect on systems without local memory. The value of *cpu* can range from **0** to **7**, where the number specifies the CPU ID of the processor from whose local memory the segment's pages are to be allocated. For complete information on NUMA policies, refer to Chapter 6, "Memory Management."
`-H`*cpu*	Specifies that the NUMA policy for the shared memory segment is to be the anchored hard-local policy. A hard-local policy causes a process to wait for local memory pages to become available if the pages cannot be allocated from the local memory pool when needed. This option has no effect on systems without local memory. The value of *cpu* can range from **0** to **7**, where the number specifies the CPU ID of the processor from whose local memory the segment's pages are to be allocated. For complete information on NUMA policies, refer to Chapter 6, "Memory Management."
`-p`*paddr*	Specifies *paddr* as the starting address of the section of physical memory to which the segment is to be bound.
`-s`*size*	Specifies the size in bytes of the section of physical memory to which the segment is to be bound.

Table 12-7. Options Specified for a Bound Segment (Cont.)

Option	Description
-u*user*	Specifies the login name of the owner of the shared memory segment.
-g*group*	Specifies the name of the group to which group access to the segment is applicable.
-m*mode*	Specifies *mode* as the set of permissions governing access to the shared memory segment. You must use the octal method to specify the permissions.

If you wish to reserve a section of physical memory and then bind the shared memory segment to it, use the following format:

/usr/sbin/shmconfig {**-L***cpu* | **-H***cpu* | } **-b***begin* [**-e***end*] **-s***size* [**-u***user*] \
[**-g***group*] [**-m***mode*] *key*

Note that you must have the `P_SHMBIND` and the `P_PLOCK` privileges to perform these operations.

When using the **-b***begin* option, the **shmconfig** utility reserves a section of physical memory of *size* bytes within the address range specified by **-b***begin* and optionally **-e***end*. It then creates a shared memory segment identified by *key* and binds it to the reserved section of physical memory.

Use this method if the physical address range resides in local or global memory and has not been previously reserved using other methods.

Options are described in Table 12-8..

Table 12-8. Options for Reserving Memory Prior to Binding

Option	Description
-L*cpu*	Specifies that the NUMA policy for the shared memory segment is to be the anchored soft-local policy. A soft-local policy causes pages to be allocated from global memory when pages are not available for allocation from the local memory pool. This option has no effect on systems without local memory. The value of *cpu* can range from **0** to **7**, where the number specifies the CPU ID of the processor from whose local memory the segment's pages are to be allocated. For complete information on NUMA policies, refer to Chapter 6, "Memory Management."

Table 12-8. Options for Reserving Memory Prior to Binding (Cont.)

Option	Description
-H*cpu*	Specifies that the NUMA policy for the shared memory segment is to be the anchored hard-local policy. A hard-local policy causes a process to wait for local memory pages to become available if the pages cannot be allocated from the local memory pool when needed. This option has no effect on systems without local memory. The value of *cpu* can range from **0** to **7**, where the number specifies the CPU ID of the processor from whose local memory the segment's pages are to be allocated. For complete information on NUMA policies, refer to Chapter 6, "Memory Management."
-b*begin*	Specifies the desired starting address of the section of physical memory to be reserved. This address may reside in local or global memory. If you wish the kernel to use the starting address of global memory as the starting address of the reserved section, specify a value of zero. If not, specify the desired starting address. Starting at the address specified by *begin*, the kernel will attempt to reserve the first available contiguous section of memory of *size* bytes that it finds. If the reserved section of memory must start at a certain physical address, you must also specify the **-e***end* option. The address specified by *end* should be equal to the value of *begin* plus the value of *size*.
-e*end*	Specifies the highest desired ending address of the section of physical memory to be reserved. This address must reside in the same memory pool as the starting address specified by *begin*. If you do not specify this option, the system will consider any physical address that is higher than *begin* and within the same memory pool as *begin* as memory that can satisfy the reservation request. Note that if you specify this option, you must also specify the **-b***begin* option. It is recommended that the address range specified by *begin* and *end* be larger than the value of *size* in case the kernel cannot reserve a contiguous section of memory starting at *begin*.
-s*size*	Specifies the size in bytes of the section of physical memory to which the segment is to be bound.
-u*user*	Specifies the login name of the owner of the shared memory segment
-g*group*	Specifies the name of the group to which group access to the segment is applicable.

Table 12-8. Options for Reserving Memory Prior to Binding (Cont.)

Option	Description
-m*mode*	Specifies *mode* as the set of permissions governing access to the shared memory segment. You must use the octal method to specify the permissions.

It is important to note that the size of a segment as specified by the **-s***size* argument must match the size of the data that will be placed there. If the **shmdefine** utility is being used, the size of the segment must match the size of the variables that are declared to be a part of the shared segment. Specifying a larger size will work. (For information on **shmdefine**, see “Using shmdefine.”)

It is recommended that you specify the **-u**, **-g**, and **-m** options to identify the user and group associated with the segment and to set the permissions controlling access to it. If you do not, the default user ID and group ID of the segment are those of the owner; the default mode is 644.

The *key* argument represents a user-chosen identifier for a shared memory segment. This identifier can be either an integer or a standard UNIX path name that refers to an existing file.

If you use the **/usr/sbin/shmconfig** command to bind a section of I/O memory to a shared memory segment, it is recommended that you use the **badaddr(2)** system call to verify that the specified I/O memory location is valid. For additional information on the use of this call, refer to the corresponding system manual page.

When the **/usr/sbin/shmconfig** command is executed, an internal data structure and shared memory segment are created for the specified key; whether the created shared memory segment is bound to a contiguous section of physical memory depends upon the options that you have specified.

To access the shared memory segment that has been created by the **/usr/sbin/shmconfig** command, processes must first call **shmget(2)** to obtain the identifier for the segment. This identifier is required by other system calls for manipulating shared memory segments. The specification for **shmget** is as follows:

int shmget(*key*, *size*, **0)**

The value of *key* is determined by the value of the *key* argument specified with the **/usr/sbin/shmconfig** command. If the value of the *key* argument was an integer, that integer must be specified as *key* on the call to **shmget**. If the value of the *key* argument was a path name, you must first call the **ftok** subroutine to obtain an integer value that is based on the path name to specify as *key* on the call to **shmget**. It is important to note that the value of the *id* argument on the call to **ftok** must be zero because **/usr/sbin/shmconfig** calls **ftok** with an *id* of zero when it converts the path name to a key. The value of *size* must be equal to the number of bytes specified by the **-s***size* argument to the **/usr/sbin/shmconfig** command. A value of **0** is specified as the *flag* argument because the shared memory segment has already been created.

For complete information on use of the **shmget** system call, see “Getting Shared Memory Segments.” For assistance in using **ftok**, see “Using Shared Memory” and the system manual page for **stdipc(3C)**. When you are creating areas of mapped memory that are to be treated as global system resources, you may find it helpful to invoke **`/usr/sbin/shmconfig`** by adding a line to the **`shmconfig`** script in the **`/etc/init.d`** directory. Doing so allows you to reserve the IPC key before noncooperating processes have an opportunity to use it, and it enables you to establish the binding between the shared memory segment and physical memory before cooperating processes need to use the segment. Add a line similar to the following example:

```
/usr/sbin/shmconfig -p 0xf00000 -s 0x10000 -u root -g sys -m 0666 key
```

If you need additional information on the use of **`/usr/sbin/shmconfig`**, refer to the system manual page **`shmconfig(1M)`**.

Multilevel Operation On Shared Memory Segments

If the Enhanced Security Utilities are installed and running, it may be desirable for a privileged process to communicate with a process running at another Mandatory Access Control (MAC) level. Multilevel operation on shared memory segments is allowed for privileged processes.

For a process to attach to a shared memory segment at a different security level (using the **`shmat`** operation of the **`shmop`** system call), both the `P_MACWRITE` and `P_MACREAD` privileges are required. The process can then read from or write to the segment if it passes DAC checks. Both privileges are also required to change the attributes of a shared memory segment at a different security level.

Even though a privileged process may access information at many different security levels, a `key` specified in a **`shmget`** system call will return a `shmid` with an associated shared memory segment and data structure having a security level identical to that of the calling process. Once a privileged process has obtained a `shmid`, the process may perform any of the possible operations from any security level. Unlike `keys, shmids` are not unique to a security level but to the entire system.

There is no defined interface to obtain the `shmid` for multilevel operation. A process may obtain the `shmid` via the **`shmget`** system call when invoked from a specific security level. A privileged user may also manually obtain a `shmid` and security level information about the message queue by invoking the **`ipcs`** command. The privileged process must have the `P_MACREAD` privilege when invoking **`ipcs.`** See the manual page **`ipcs(1)`** for details on the use of **`ipcs.`**

The **`lvlipc`** system call reports the security level of a shared memory segment associated with the specified **`shmid.`** This system call is of little use to an unprivileged process, since a shared memory segment created and used by the unprivileged process always has a security level equal to that of the process. The process with the `P_MACREAD` privilege, though, may use this system call to find the security level of any existing shared memory segment on the system. The process must also have discretionary read access to the memory segment.

For a detailed discussion of process privileges, see “Privileges” in Chapter 9 of this guide.

13
STREAMS Polling and Multiplexing

13
STREAMS Polling and Multiplexing

Introduction

This chapter describes how STREAMS allows user processes to monitor, control, and poll Streams to allow an effective utilization of system resources. The synchronous polling mechanism and asynchronous event notification within STREAMS is discussed. STREAMS signal handling between modules and/or drivers and user processes is also discussed.

The remainder of this chapter is devoted to STREAMS input/output multiplexing. It defines a STREAMS multiplexor, and describes multiplexing drivers. A discussion of how STREAMS multiplexing configurations are created, is included. Code examples are included to illustrate using both the polling and multiplexing mechanisms.

STREAMS Input/Output Polling

This section describes the synchronous polling mechanism and asynchronous event notification within STREAMS.

User processes can efficiently monitor and control multiple Streams with two system calls: **`poll`** and the **`I_SETSIG`** **`ioctl`** command. These calls allow a user process to detect events that occur at the Stream head on one or more Streams, including receipt of data or messages on the read queue and cessation of flow control.

To monitor Streams with **`poll`**, a user process issues that system call and specifies the Streams to be monitored, the events to look for, and the amount of time to wait for an event. The **`poll`** system call blocks the process until the time expires or until an event occurs. If an event occurs, it returns the type of event and the Stream on which the event occurred.

Instead of waiting for an event to occur, a user process may want to monitor one or more Streams while processing other data. It can do so by issuing the **`I_SETSIG`** **`ioctl`** command, specifying one or more Streams and events (as with **`poll`**). This **`ioctl`** does not block the process and force the user process to wait for the event but returns immediately and issues a signal when an event occurs. The process must specify a signal handler to catch the resultant `SIGPOLL` signal.

If any selected event occurs on any of the selected Streams, STREAMS causes the `SIGPOLL` catching function to be executed in all associated requesting processes. However, the process(es) will not know which event occurred, nor on what Stream the event occurred. A process that issues the **I_SETSIG** can get more detailed information by issuing a **poll** after it detects the event.

Synchronous Input/Output

The **poll** system call provides a mechanism to identify those Streams over which a user can send or receive data. For each Stream of interest, users can specify one or more events about which they should be notified. The types of events that can be polled are as follows:

POLLIN	A message other than an M_PCPROTO is at the front of the Stream head read queue. This event is maintained for compatibility with the previous releases of the UNIX System V.
POLLRDNORM	A normal (nonpriority) message is at the front of the Stream head read queue.
POLLRDBAND	A priority message (band > 0) is at the front of the Stream head queue.
POLLPRI	A high-priority message (M_PCPROTO) is at the front of the Stream head read queue.
POLLOUT	The normal priority band of the queue is writable (not flow controlled).
POLLWRNORM	The same as POLLOUT.
POLLWRBAND	A priority band greater than 0 of a queue downstream exists and is writable.
POLLMSG	An M_SIG or M_PCSIG message containing the SIGPOLL signal has reached the front of the Stream head read queue.

Some of the events may not be applicable to all file types. For example, it is not expected that the POLLPRI event will be generated when polling a regular file. POLLIN, POLLRDNORM, POLLRDBAND, and POLLPRI are set even if the message is of zero length.

The **poll** system call examines each file descriptor for the requested events and, on return, shows which events have occurred for each file descriptor. If no event has occurred on any polled file descriptor, **poll** blocks until a requested event or timeout occurs. **poll** takes the following arguments:

- An array of file descriptors and events to be polled.
- The number of file descriptors to be polled.
- The number of milliseconds **poll** should wait for an event if no events are pending (-1 specifies wait forever).

The following example shows the use of **poll**. Two separate minor devices of the communications driver are opened, thereby establishing two separate Streams to the driver.

The `pollfd` entry is initialized for each device. Each Stream is polled for incoming data. If data arrives on either Stream, it is read and then written back to the other Stream.

```
#include <fcntl.h>
#include <poll.h>

#define NPOLL 2 /* number of file descriptors to poll */

main()
{
    struct pollfd pollfds[NPOLL];
    char buf[1024];
    int count, i;

    if ((pollfds[0].fd = open(“/dev/comm/01”, O_RDWR|O_NDELAY)) < 0) {
        perror(“open failed for /dev/comm/01”);
        exit(1);
    }

    if ((pollfds[1].fd = open(“/dev/comm/02”, O_RDWR|O_NDELAY)) < 0) {
        perror(“open failed for /dev/comm/02”);
        exit(2);
    }
```

The variable `pollfds` is declared as an array of the `pollfd` structure that is defined in **`<poll.h>`** and has the following format:

```
struct pollfd {
    int     fd;     /* file descriptor */
    shortevents;    /* requested events */
    shortrevents;   /* returned events */
}
```

For each entry in the array, **`fd`** specifies the file descriptor to be polled and `events` is a bitmask that contains the bitwise inclusive `OR` of events to be polled on that file descriptor. On return, the `revents` bitmask indicates which of the requested events has occurred.

The example continues to process incoming data as follows:

```
    pollfds[0].events = POLLIN; /* set events to poll */
    pollfds[1].events = POLLIN; /* for incoming data */
    pollfds[0].revents = 0;
    pollfds[1].revents = 0;


    while (1) {
        /* poll and use -1 timeout (infinite) */
        if (poll(pollfds, NPOLL, -1) < 0) {
            perror("poll failed");
            exit(3);
        }
        for (i = 0; i < NPOLL; i++) {
            switch (pollfds[i].revents) {

            case 0:                       /* no events */
                break;

            case POLLIN:
                /* echo incoming data on "other" Stream */
                while ((count = read(pollfds[i].fd, buf, 1024)) > 0)
                    /*
                     * the write loses data if flow control
                     * prevents the transmit at this time.
                     */
                if (write(pollfds[(i+1)%2].fd, buf, count) != count)
                        fprintf(stderr,"writer lost data\n");
                pollfds[i].revents = 0;
                break;

            default:                      /* default error case */
                perror("error event");
                exit(4);
            }
        }
    }
}
```

The user specifies the polled events by setting the `events` field of the `pollfd` structure to `POLLIN`. This requested event directs **poll** to notify the user of any incoming data on each Stream. The bulk of the example is an infinite loop, where each iteration polls both Streams for incoming data.

The second argument to the **poll** system call specifies the number of entries in the `pollfds` array (2 in this example). The third argument is a *timeout* value indicating the number of milliseconds **poll** should wait for an event if none occurs. On a system where millisecond accuracy is not available, *timeout* is rounded up to the nearest value available on that system. If the value of *timeout* is `0`, **poll** returns immediately. Here, the value of *timeout* is `-1`, specifying that **poll** should block until a requested event occurs or until the call is interrupted.

If the **poll** call succeeds, the program looks at each entry in the `pollfds` array. If `revents` is set to `0`, no event has occurred on that file descriptor. If `revents` is set to `POLLIN`, incoming data is available. In this case, all data is read from the polled minor device and written to the other minor device.

If `revents` is set to a value other than `0` or `POLLIN`, an error event must have occurred on that Stream, because `POLLIN` was the only requested event. The following are **poll** error events:

`POLLERR`	A fatal error has occurred in some module or driver on the Stream associated with the specified file descriptor. Further system calls will fail.
`POLLHUP`	A hangup condition exists on the Stream associated with the specified file descriptor. This event and `POLLOUT` are mutually exclusive; a Stream cannot be writable if a hangup has occurred.
`POLLNVAL`	The specified file descriptor is not valid

These events may not be polled by the user, but will be reported in `revents` whenever they occur. As such, they are only valid in the `revents` bitmask.

The example attempts to process incoming data as quickly as possible. However, when writing data to a Stream, the **`write`** call may block if the Stream is exerting flow control. To prevent the process from blocking, the minor devices of the communications driver were opened with the `O_NDELAY` (or `O_NONBLOCK`, see note) flag set. The **`write`** will not be able to send all the data if flow control is exerted and `O_NDELAY` (`O_NONBLOCK`) is set. This can occur if the communications driver is unable to keep up with the user's rate of data transmission. If the Stream becomes full, the number of bytes the **`write`** sends will be less than the requested `count`. For simplicity, the example ignores the data if the Stream becomes full, and a warning is printed to `stderr`.

NOTE

For conformance with the IEEE operating system interface standard, POSIX, it is recommended that new applications use the `O_NONBLOCK` flag, which behaves the same as `O_NDELAY` unless otherwise noted.

This program continues until an error occurs on a Stream, or until the process is interrupted.

Asynchronous Input/Output

The **`poll`** system call enables a user to monitor multiple Streams in a synchronous fashion. The **`poll`** call normally blocks until an event occurs on any of the polled file descriptors. In some applications, however, it is desirable to process incoming data asynchronously. For example, an application may want to do some local processing and be interrupted when a pending event occurs. Some time-critical applications cannot afford to block, but must have immediate indication of success or failure.

The **`I_SETSIG`** **`ioctl`** call (see **`streamio(7)`**) is used to request that a `SIGPOLL` signal be sent to a user process when a specific event occurs. Listed below are events for the **`ioctl`** **`I_SETSIG`**. These are similar to those described for **`poll`**.

`S_INPUT`	A message other than an `M_PCPROTO` is at the front of the Stream head read queue. This event is maintained for compatibility with previous releases.

S_RDNORM	A normal (nonpriority) message is at the front of the Stream head read queue.
S_RDBAND	A priority message (band > 0) is at the front of the Stream head read queue.
S_HIPRI	A high-priority message (M_PCPROTO) is present at the front of the Stream head read queue.
S_OUTPUT	A write queue for normal data (priority band = 0) is no longer full (not flow controlled). This notifies a user that there is room on the queue for sending or writing normal data downstream.
S_WRNORM	The same as S_OUTPUT.
S_WRBAND	A priority band greater than 0 of a queue downstream exists and is writable. This notifies a user that there is room on the queue for sending or writing priority data downstream.
S_MSG	An M_SIG or M_PCSIG message containing the SIGPOLL flag has reached the front of Stream head read queue.
S_ERROR	An M_ERROR message reaches the Stream head.
S_HANGUP	An M_HANGUP message reaches the Stream head.
S_BANDURG	When used with S_RDBAND, SIGURG is generated instead SIGPOLL when a priority message reaches the front of the Stream head read queue.

S_INPUT, S_RDNORM, S_RDBAND, and S_HIPRI are set even if the message is of zero length. A user process may choose to handle only high-priority messages by setting the *arg* to S_HIPRI.

Signals

STREAMS allows modules and drivers to cause a signal to be sent to user process(es) through an M_SIG or M_PCSIG message. The first byte of the message specifies the signal for the Stream head to generate. If the signal is not SIGPOLL (see **signal(2)**), the signal is sent to the process group associated with the Stream. If the signal is SIGPOLL, the signal is only sent to processes that have registered for the signal by using the **I_SETSIG ioctl**.

An M_SIG message can be used by modules or drivers that want to insert an explicit inband signal into a message Stream. For example, this message can be sent to the user process immediately before a particular service interface message to gain the immediate attention of the user process. When the M_SIG message reaches the head of the Stream head read queue, a signal is generated and the M_SIG message is removed. This leaves the service interface message as the next message to be processed by the user. Use of the M_SIG message is typically defined as part of the service interface of the driver or module.

Extended Signals

To enable a process to obtain the band and event associated with SIGPOLL more readily, STREAMS supports extended signals. For the given events, a special code is defined in **<siginfo.h>** that describes the reason SIGPOLL was generated. Table 13-1 describes the data available in the **siginfo_t** structure passed to the signal handler.

Table 13-1. siginfo_t Data Available to the Signal Handler

Event	si_signo	si_code	si_band	si_errno
S_INPUT	SIGPOLL	POLL_IN	band readable	unused
S_OUTPUT	SIGPOLL	POLL_OUT	band writable	unused
S_MSG	SIGPOLL	POLL_MSG	band signaled	unused
S_ERROR	SIGPOLL	POLL_ERR	unused	Stream error
S_HANGUP	SIGPOLL	POLL_HUP	unused	unused
S_HIPRI	SIGPOLL	POLL_PRI	unused	unused

STREAMS Input/Output Multiplexing

This section describes how STREAMS multiplexing configurations are created and also discusses multiplexing drivers.

Earlier, Streams were described as linear connections of modules, where each invocation of a module is connected to at most one upstream module and one downstream module. While this configuration is suitable for many applications, others require the ability to multiplex Streams in a variety of configurations. Typical examples are terminal window facilities, and internetworking protocols (which might route data over several subnetworks).

Figure 13-1 shows an example of a multiplexor that multiplexes data from several upper Streams over a single lower Stream. An upper Stream is one that is upstream from a multiplexor, and a lower Stream is one that is downstream from a multiplexor. A terminal windowing facility might be implemented in this fashion, where each upper Stream is associated with a separate window.

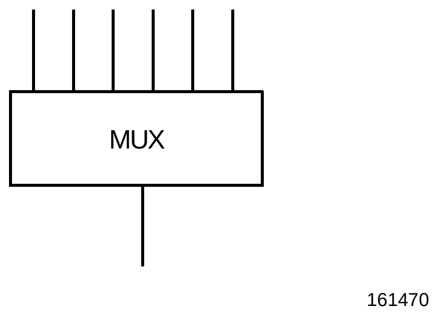


Figure 13-1. Many-to-One Multiplexor

Figure 13-2 shows a second type of multiplexor that might route data from a single upper Stream to one of several lower Streams. An internetworking protocol could take this form, where each lower Stream links the protocol to a different physical network.

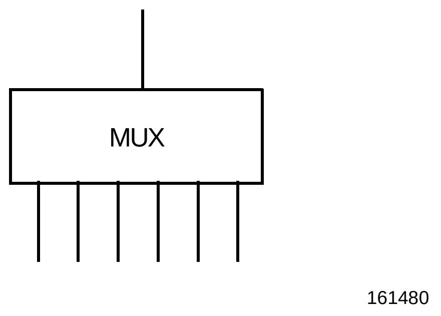


Figure 13-2. One-to-Many Multiplexor

Figure 13-3 shows a third type of multiplexor that might route data from one of many upper Streams to one of many lower Streams.

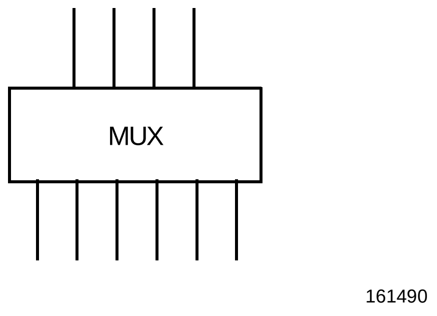


Figure 13-3. Many-to-Many Multiplexor

The STREAMS mechanism supports the multiplexing of Streams through special pseudo-device drivers. Using a linking facility, users can dynamically build, maintain, and dismantle multiplexed Stream configurations. Simple configurations like the ones shown in Figure 13-1 through Figure 13-3 can be further combined to form complex, multilevel, multiplexed Stream configurations.

STREAMS multiplexing configurations are created in the kernel by interconnecting multiple Streams. Conceptually, there are two kinds of multiplexors: upper and lower multiplexors. Lower multiplexors have multiple lower Streams between device drivers and the multiplexor, and upper multiplexors have multiple upper Streams between user processes and the multiplexor.

Figure 13-4 is an example of the multiplexor configuration that typically occurs where internetworking functions are included in the system. This configuration contains three hardware device drivers. The IP (Internet Protocol) is a multiplexor.

The IP multiplexor switches messages among the lower Streams or sends them upstream to user processes in the system. In this example, the multiplexor expects to see the same interface downstream to Module 1, Module 2, and Driver 3.

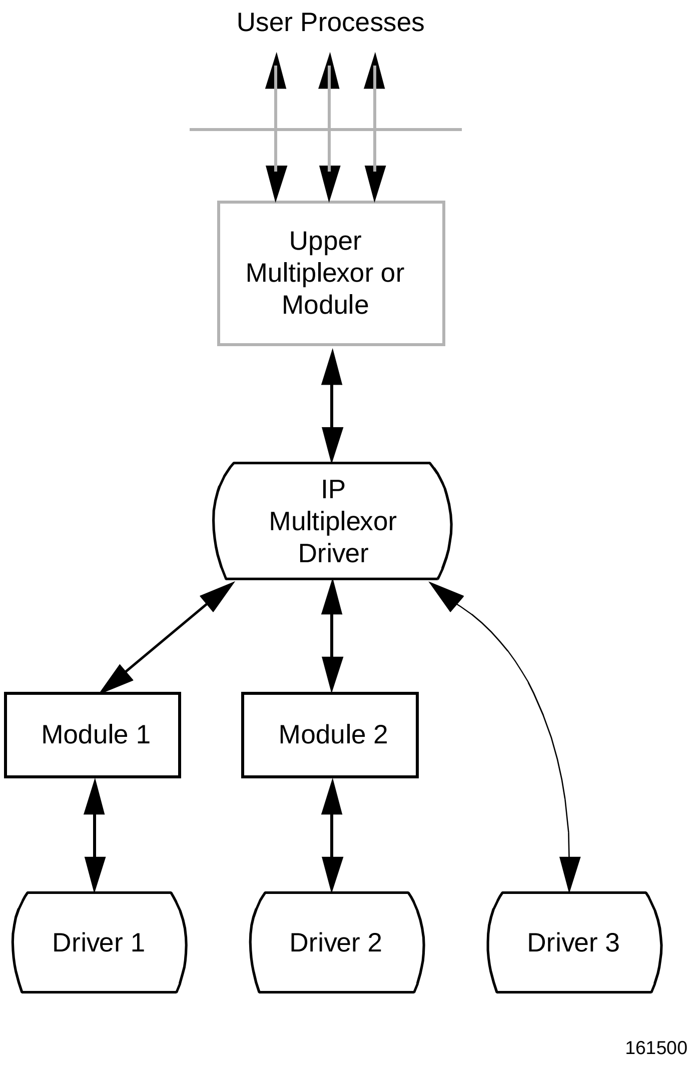


Figure 13-4. Internet Multiplexing Stream

Figure 13-4 depicts the IP multiplexor as part of a larger configuration. The multiplexor configuration, shown in the dashed rectangle, generally has an upper multiplexor and additional modules. Multiplexors can also be cascaded below the IP multiplexor driver if the device drivers are replaced by multiplexor drivers.

Figure 13-5 shows a multiplexor configuration where the multiplexor (or multiplexing driver) routes messages between the lower Stream and one upper Stream. This Stream performs X.25 multiplexing to multiple independent Switched Virtual Circuit (SVC) and Permanent Virtual Circuit (PVC) user processes. Upper multiplexors are a specific application of standard STREAMS facilities that support multiple minor devices in a device driver. This figure also shows that more complex configurations can be built by having one or more multiplexed drivers below and multiple modules above an upper multiplexor.

Developers can choose either upper or lower multiplexing, or both, when designing their applications. For example, a window multiplexor would have a similar configuration to the

X.25 configuration of Figure 13-5, with a window driver replacing the Packet Layer, a tty driver replacing the driver XYZ, and the child processes of the terminal process replacing the user processes. Although the X.25 and window multiplexing Streams have similar configurations, their multiplexor drivers would differ significantly. The IP multiplexor in Figure 13-4 has a different configuration than the X.25 multiplexor, and the driver would implement its own set of processing and routing requirements in each configuration.

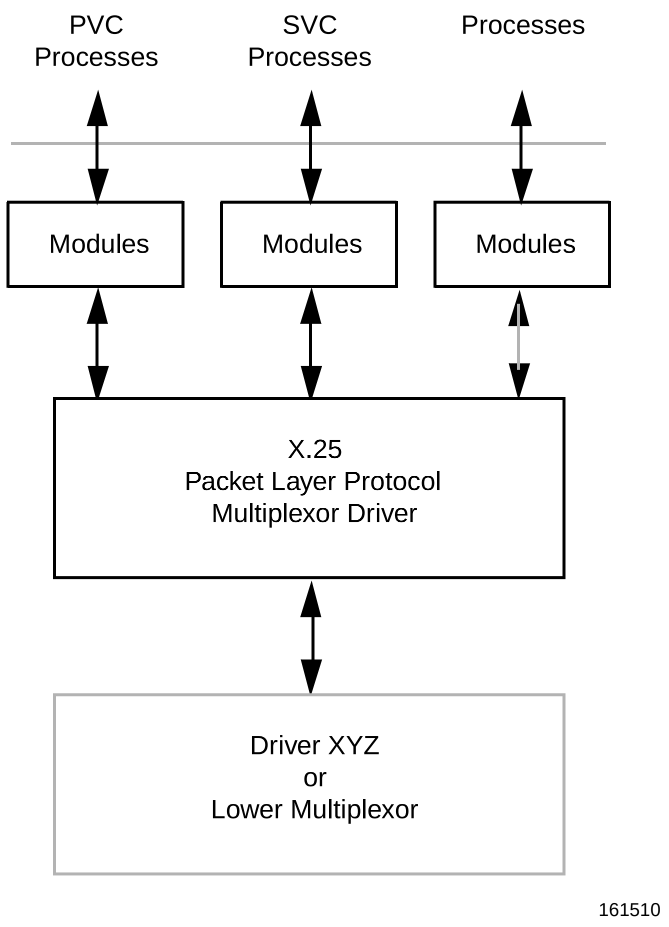

PVC Processes
SVC Processes
Processes
Modules
Modules
Modules
X.25 Packet Layer Protocol Multiplexor Driver
Driver XYZ or Lower Multiplexor
161510

Figure 13-5. X.25 Multiplexing Stream

In addition to upper and lower multiplexors, you can create more complex configurations by connecting Streams containing multiplexors to other multiplexor drivers. With such a diversity of needs for multiplexors, it is not possible to provide general purpose multiplexor drivers. Rather, STREAMS provides a general purpose multiplexing facility, which allows users to set up the intermodule/driver plumbing to create multiplexor configurations of generally unlimited interconnection.

STREAMS Multiplexors

A STREAMS multiplexor is a driver with multiple Streams connected to it. The primary function of the multiplexing driver is to switch messages among the connected Streams. Multiplexor configurations are created at user level by system calls.

STREAMS-related system calls set up the “plumbing,” or Stream interconnections, for multiplexing drivers. The subset of these calls that allows a user to connect (and disconnect) Streams below a driver is referred to as the multiplexing facility. This type of connection is referred to as a 1-to-M, or lower, multiplexor configuration. This configuration must always contain a multiplexing driver, which is recognized by STREAMS as having special characteristics.

Multiple Streams can be connected above a driver by **`open`** calls. There is no difference between the connections to these drivers, only the functions performed by the driver are different. In the multiplexing case, the driver routes data between multiple Streams. In the device driver case, the driver routes data between user processes and associated physical ports. Multiplexing with Streams connected above is referred to as an N-to-1, or upper, multiplexor. STREAMS does not provide any facilities beyond **`open`** and **`close`** to connect or disconnect upper Streams for multiplexing purposes.

From the driver's perspective, upper and lower configurations differ only in how they are initially connected to the driver. The implementation requirements are the same: route the data and handle flow control. All multiplexor drivers require special developer-provided software to perform the multiplexing data routing and to handle flow control. STREAMS does not directly support flow control among multiplexed Streams.

M-to-N multiplexing configurations are implemented by using both of the above mechanisms in a driver.

The multiple Streams that represent minor devices are actually distinct Streams in which the driver keeps track of each Stream attached to it. The STREAMS subsystem does not recognize any relationship between the Streams. The same is true for STREAMS multiplexors of any configuration. The multiplexed Streams are distinct and the driver must be implemented to do most of the work.

In addition to upper and lower multiplexors, more complex configurations can be created by connecting Streams containing multiplexors to other multiplexor drivers. With such a diversity of needs for multiplexors, it is not possible to provide general-purpose multiplexor drivers. Rather, STREAMS provides a general purpose multiplexing facility that allows users to set up the intermodule/driver plumbing to create multiplexor configurations of generally unlimited interconnection.

Building a Multiplexor

This section builds a protocol multiplexor with the multiplexing configuration shown in Figure 13-6. To free users from the need to know about the underlying protocol structure, a user-level daemon process is built to maintain the multiplexing configuration. Users can then access the transport protocol directly by opening the transport protocol (TP) driver device node.

An internetworking protocol driver (IP) routes data from a single upper Stream to one of two lower Streams. This driver supports two STREAMS connections beneath it. These connections are to two distinct networks; one for the IEEE 802.3 standard with the 802.3 driver, and the other to the IEEE 802.4 standard with the 802.4 driver. The TP driver multiplexes upper Streams over a single Stream to the IP driver.

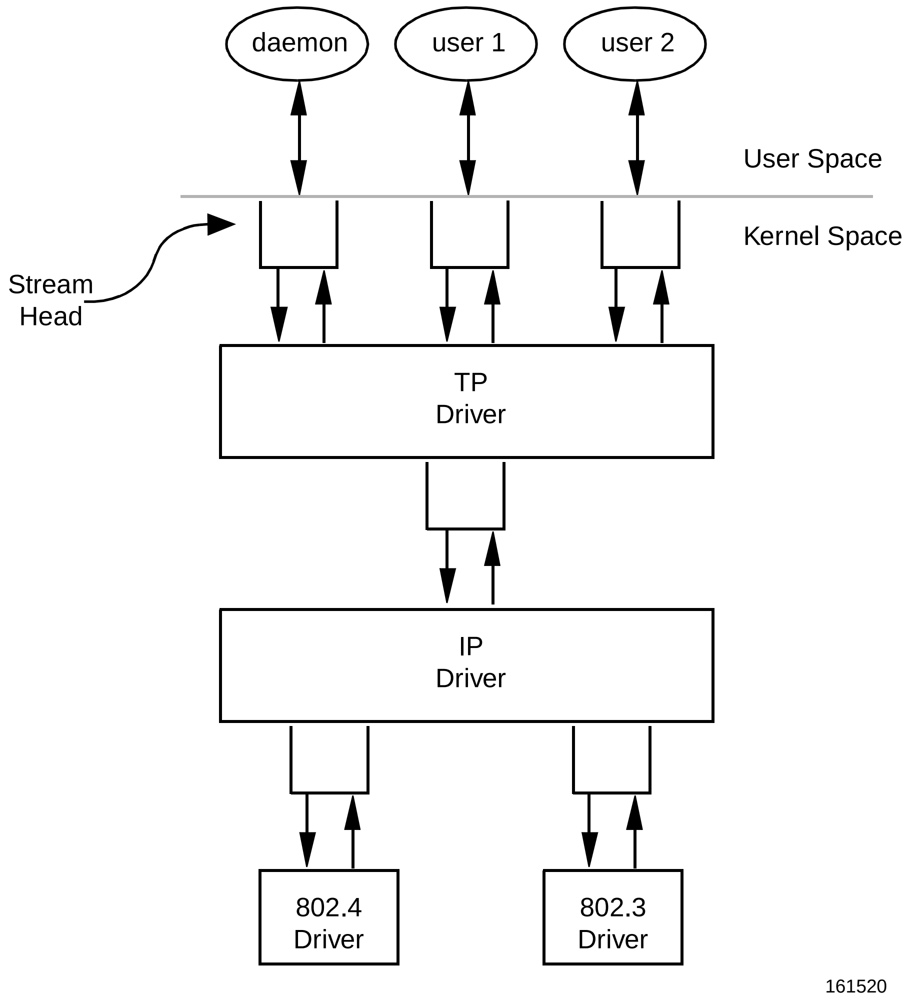


Figure 13-6. Protocol Multiplexor

The following example shows how this daemon process sets up the protocol multiplexor.The necessary declarations and initialization for the daemon program are as follows:

```
#include <fcntl.h>
#include <stropts.h>

main()
{
    int fd_802_4,
        fd_802_3,
        fd_ip,
        fd_tp;

    /* daemon-ize this process */

    switch (fork()) {
    case 0:
        break;
    case -1:
        perror("fork failed");
        exit(2);
    default:
        exit(0);
    }
    setsid();
```

This multilevel multiplexed Stream configuration is built from the bottom up. Therefore, the example begins by first constructing the Internet Protocol (IP) multiplexor. This multiplexing device driver is treated like any other software driver. It owns a node in the UNIX file system and is opened just like any other STREAMS device driver.

The first step is to open the multiplexing driver and the 802.4 driver, thus creating separate Streams above each driver as shown in Figure 13-7. The Stream to the 802.4 driver may now be connected below the multiplexing IP driver using the **`I_LINK`** **`ioctl`** call.

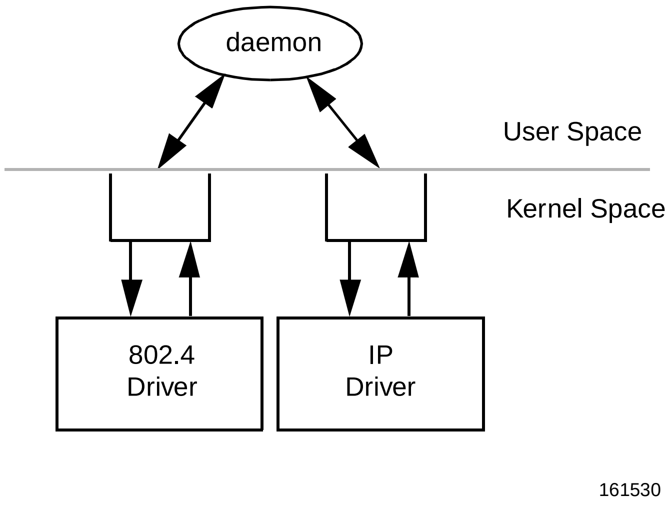


Figure 13-7. Before Link

The sequence of instructions to this point is

```
if ((fd_802_4 = open("/dev/802_4", O_RDWR)) < 0) {
    perror("open of /dev/802_4 failed");
    exit(1);
}

if ((fd_ip = open("/dev/ip", O_RDWR)) < 0) {
    perror("open of /dev/ip failed");
    exit(2);
}

/* now link 802.4 to underside of IP */

if (ioctl(fd_ip, I_LINK, fd_802_4) < 0) {
    perror("I_LINK ioctl failed");
    exit(3);
}
```

`I_LINK` takes two file descriptors as arguments. The first file descriptor, **`fd_ip`**, must reference the Stream connected to the multiplexing driver, and the second file descriptor, `fd_802_4`, must reference the Stream to be connected below the multiplexor. Figure 13-8 shows the state of these Streams following the **`I_LINK`** call. The complete Stream to the 802.4 driver has been connected below the IP driver. The Stream head's queues of the 802.4 driver is used by the IP driver to manage the lower half of the multiplexor.

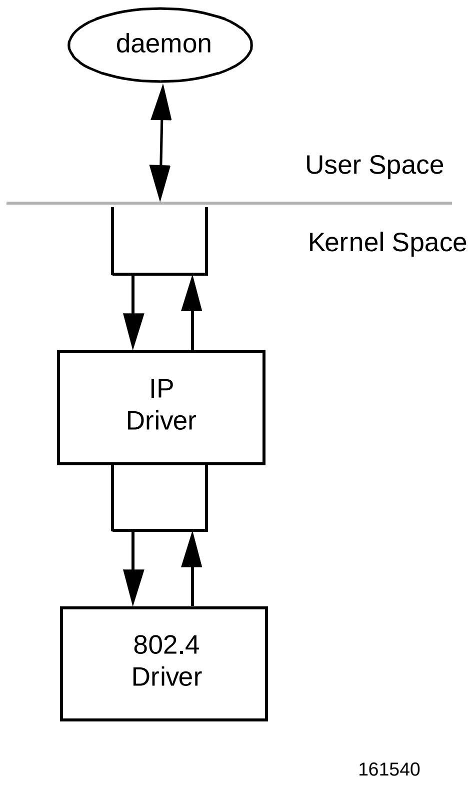


Figure 13-8. IP Multiplexor after First Link

`I_LINK` returns an integer value, called `muxid`, which is used by the multiplexing driver to identify the Stream just connected below it. This `muxid` is ignored in the example, but is useful for dismantling a multiplexor or routing data through the multiplexor. Its significance is discussed later.

The following sequence of system calls is used to continue building the internetworking protocol multiplexor (IP):

```
if ((fd_802_3 = open(“/dev/802_3”, O_RDWR)) < 0) {
     perror(“open of /dev/802_3 failed”);
     exit(4);
}

if (ioctl(fd_ip, I_LINK, fd_802_3) < 0) {
     perror(“I_LINK ioctl failed”);
     exit(5);
}
```

All links below the IP driver have now been established, giving the configuration in Figure 13-9.

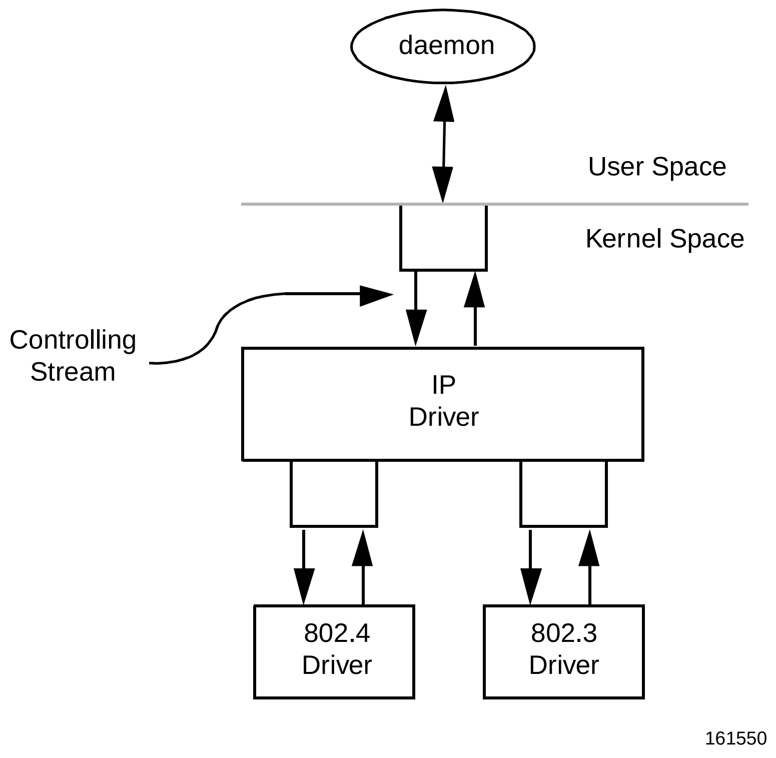


Figure 13-9. IP Multiplexor

The Stream above the multiplexing driver used to establish the lower connections is the controlling Stream and has special significance when dismantling the multiplexing configuration. This will be illustrated later in this section. The Stream referenced by `fd_ip` is the controlling Stream for the IP multiplexor.

NOTE

> The order in which the Streams in the multiplexing configuration are opened is unimportant. If it is necessary to have intermediate modules in the Stream between the IP driver and media drivers, these modules must be added to the Streams associated with the media drivers (using **`I_PUSH`**) before the media drivers are attached below the multiplexor.

The number of Streams that can be linked to a multiplexor is restricted by the design of the particular multiplexor. The manual page describing each driver (typically found in Section 7) describes such restrictions. However, only one **`I_LINK`** operation is allowed for each lower Stream; a single Stream cannot be linked below two multiplexors simultaneously.

Continuing with the example, the IP driver is now linked below the transport protocol (TP) multiplexing driver. As seen earlier in Figure 13-6, only one link is supported below the transport driver. This link is formed by the following sequence of system calls:

```
if ((fd_tp = open("/dev/tp", O_RDWR)) < 0) {
    perror("open of /dev/tp failed");
    exit(6);
}

if (ioctl(fd_tp, I_LINK, fd_ip) < 0) {
    perror("I_LINK ioctl failed");
    exit(7);
}
```

The multilevel multiplexing configuration shown in Figure 13-10 has now been created.

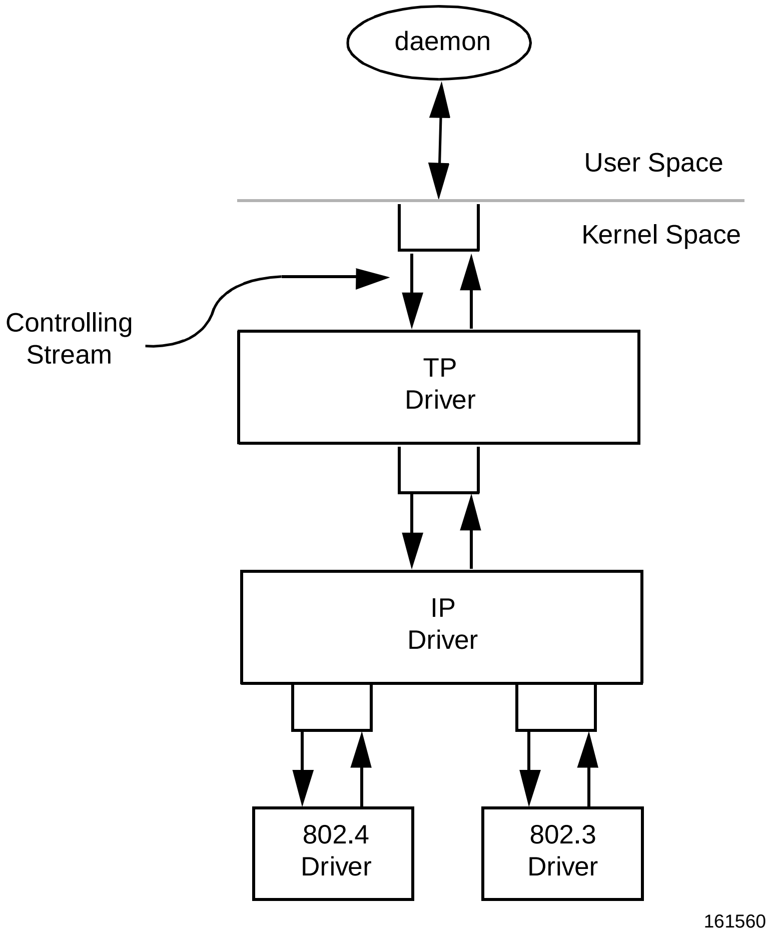


Figure 13-10. TP Multiplexor

Because the controlling Stream of the IP multiplexor has been linked below the TP multiplexor, the controlling Stream for the new multilevel multiplexor configuration is the Stream above the TP multiplexor.

At this point, the file descriptors associated with the lower drivers can be closed without affecting the operation of the multiplexor. If these file descriptors are not closed, all later **`read`**, **`write`**, **`ioctl`**, **`poll`**, **`getmsg`**, and **`putmsg`** system calls issued to them will fail because **`I_LINK`** associates the Stream head of each linked Stream with the multiplexor, so the user may not access that Stream directly for the duration of the link.

The following sequence of system calls completes the daemon example:

```
    close(fd_802_4);
    close(fd_802_3);
    close(fd_ip);

    /* Hold multiplexor open forever */
    pause();
}
```

To summarize, Figure 13-10 shows the multilevel protocol multiplexor. The transport driver supports several simultaneous Streams. These Streams are multiplexed over the single Stream connected to the IP multiplexor. The mechanism for establishing multiple Streams above the transport multiplexor is actually a by-product of the way in which Streams are created between a user process and a driver. By opening different minor devices of a STREAMS driver, separate Streams are connected to that driver. Of course, the driver must be designed with the intelligence to route data from the single lower Stream to the appropriate upper Stream.

The daemon process maintains the multiplexed Stream configuration through an open Stream (the controlling Stream) to the transport driver. Meanwhile, other users can access the services of the transport protocol by opening new Streams to the transport driver; they are freed from the need for any unnecessary knowledge of the underlying protocol configurations and subnetworks that support the transport service.

Multilevel multiplexing configurations should be assembled from the bottom up because the passing of **`ioctl`**s through the multiplexor is determined by the multiplexing driver and cannot generally be relied on.

Dismantling a Multiplexor

Streams connected to a multiplexing driver from above with **`open`**, can be dismantled by closing each Stream with **`close`**. The mechanism for dismantling Streams that have been linked below a multiplexing driver is less obvious, and is described below.

The **`I_UNLINK`** **`ioctl`** call disconnects each multiplexor link below a multiplexing driver individually. This command has the form:

`ioctl``(`*fd*`, `**`I_UNLINK`**`, `*muxid*`);`

where *fd* is a file descriptor associated with a Stream connected to the multiplexing driver from above, and *muxid* is the identifier that was returned by **`I_LINK`** when a driver was linked below the multiplexor. Each lower driver may be disconnected individually in this way, or a special *muxid* value of `-1` may disconnect all drivers from the multiplexor simultaneously.

In the multiplexing daemon program presented earlier, the multiplexor is never explicitly dismantled because all links associated with a multiplexing driver are automatically dismantled when the controlling Stream associated with that multiplexor is closed. Because the controlling Stream is open to a driver, only the final call of **`close`** for that Stream closes it. In this case, the daemon is the only process that opens the controlling Stream, so the multiplexing configuration is dismantled when the daemon exits.

For the automatic dismantling mechanism to work in the multilevel, multiplexed Stream configuration, the controlling Stream for each multiplexor at each level must be linked under the next higher level multiplexor. In the example, the controlling Stream for the IP driver was linked under the TP driver, which resulted in a single controlling Stream for the full, multilevel configuration. Because the multiplexing program relied on closing the controlling Stream to dismantle the multiplexed Stream configuration instead of using explicit **`I_UNLINK`** calls, the *muxid* values returned by **`I_LINK`** could be ignored.

An important side-effect of automatic dismantling on the close is that it is not possible for a process to build a multiplexing configuration with **`I_LINK`** and then exit. This is because **`exit`** closes all files associated with the process, including the controlling Stream. To keep the configuration intact, the process must exist for the life of that multiplexor. That is the motivation for implementing the example as a daemon process.

However, if the process uses persistent links with the **`I_PLINK`** **`ioctl`** call, the multiplexor configuration remains intact after the process exits. Persistent links are described later in this section.

Routing Data through a Multiplexor

As shown, STREAMS provides a mechanism for building multiplexed Stream configurations. However, the criteria on which a multiplexor routes data is driver-dependent. For example, the protocol multiplexor shown before might use address information found in a protocol header to determine over which subnetwork data should be routed. It is the multiplexing driver's responsibility to define its routing criteria.

One routing option available to the multiplexor is to use the *muxid* value to determine to which Stream data should be routed (remember that each multiplexor link is associated with a *muxid*). **`I_LINK`** passes the *muxid* value to the driver and returns this value to the user. The driver can therefore specify that the *muxid* value must accompany data routed through it. For example, if a multiplexor routed data from a single upper Stream to one of several lower Streams (as did the IP driver), the multiplexor could require the user to insert the *muxid* of the desired lower Stream into the first four bytes of each message passed to it. The driver could then match the *muxid* in each message with the *muxid* of each lower Stream, and route the data accordingly.

Persistent Links

With **`I_LINK`** and **`I_UNLINK`** **`ioctl`**s, the file descriptor associated with the Stream above the multiplexor used to set up the lower multiplexor connections must remain open for the duration of the configuration. Closing the file descriptor associated with the controlling Stream dismantles the whole multiplexing configuration. Some applications may not want to keep a process running merely to hold the multiplexor configuration together. Therefore, "free-standing" links below a multiplexor are needed. A persistent link is such a link. It is similar to a STREAMS multiplexor link, except that a process is not needed to hold the links together. After the multiplexor has been set up, the process may close all file descriptors and exit, and the multiplexor remains intact.

Two **`ioctl`**s, **`I_PLINK`** and **`I_PUNLINK`**, are used to create and remove persistent links that are associated with the Stream above the multiplexor. **`close`** and **`I_UNLINK`** are not able to disconnect the persistent links.

The format of **`I_PLINK`** is

> **`ioctl`**`(`*fd0*`,` **`I_PLINK`**`,` *fd1*`)`

The first file descriptor, *fd0*, must reference the Stream connected to the multiplexing driver and the second file descriptor, *fd1*, must reference the Stream to be connected below the multiplexor. The persistent link can be created in the following way:

```
upper_stream_fd = open("/dev/mux", O_RDWR);
lower_stream_fd = open("/dev/driver", O_RDWR);
muxid = ioctl(upper_stream_fd, I_PLINK, lower_stream_fd);
/*
 * save muxid in a file
 */
exit(0);
```

Figure 13-11 shows how **`open`** establishes a Stream between the device and the Stream head.

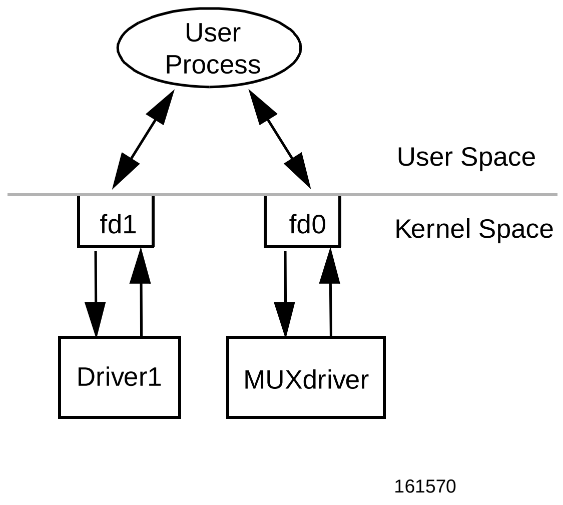


Figure 13-11. Open of MUXdriver and Driver1

The persistent link can still exist even if the file descriptor associated with the upper Stream to the multiplexing driver is closed. The **`I_PLINK ioctl`** returns an integer value, `muxid`, that can be used for dismantling the multiplexing configuration. If the process that created the persistent link still exists, it may pass the `muxid` value to some other process to dismantle the link, if the dismantling is desired, or it can leave the `muxid` value in a file so that other processes may find it later. Figure 13-12 shows a multiplexor after **`I_PLINK`**.

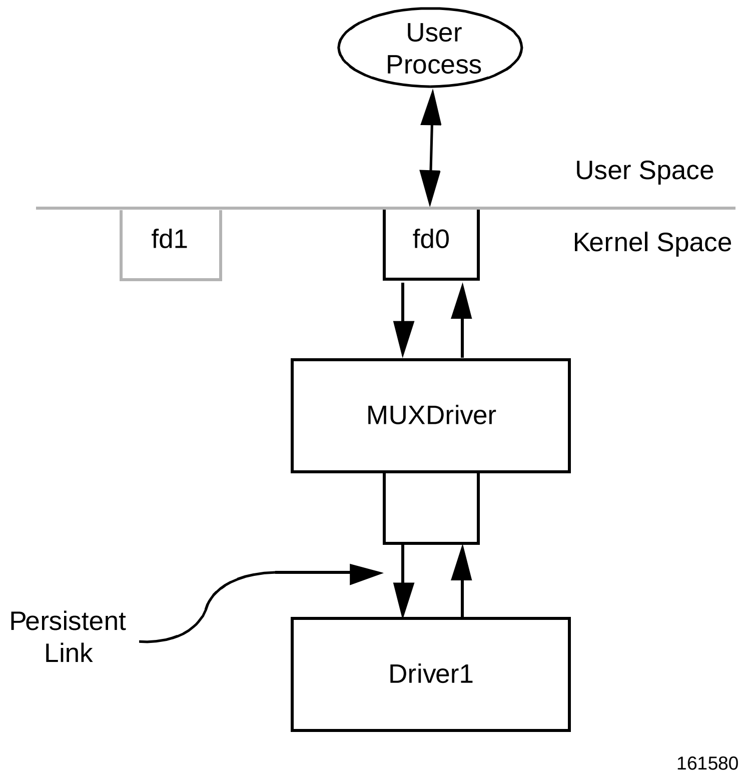


Figure 13-12. Multiplexor after I_PLINK

Several users can open the MUXdriver and send data to Driver1 since the persistent link to Driver1 remains intact. This is shown in Figure 13-13.

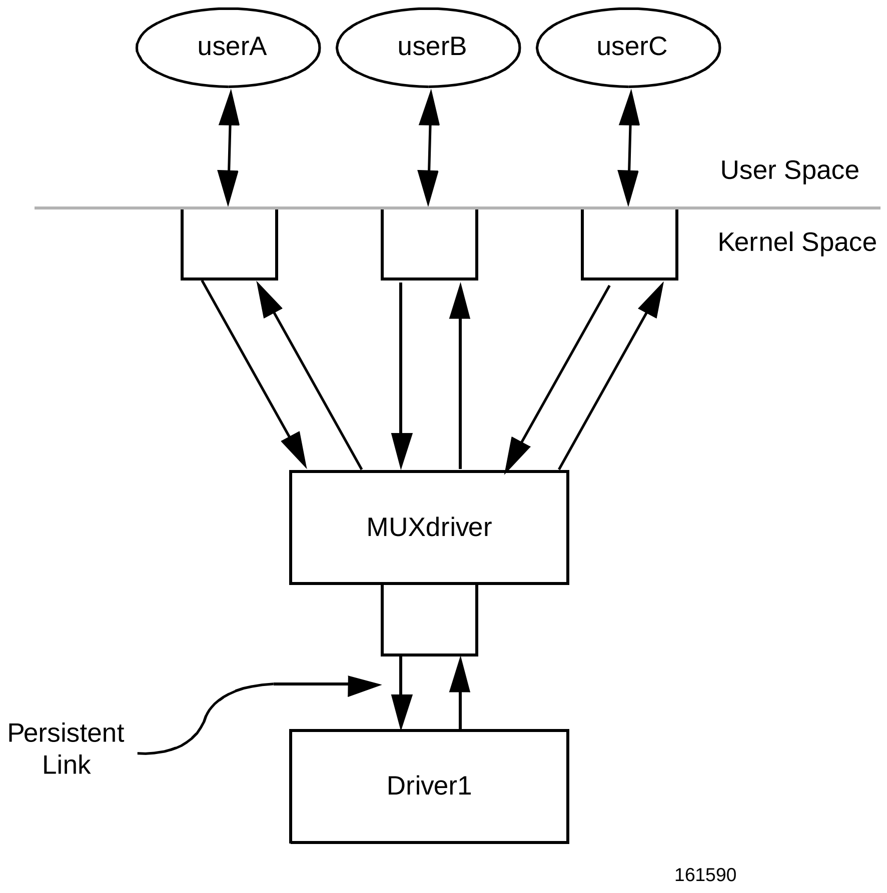


Figure 13-13. Other Users Opening a MUXdriver

The **`I_PUNLINK`** **`ioctl`** is used for dismantling the persistent link. Its format is

> **`ioctl`**`(`*fd0*`,` **`I_PUNLINK`**`,` *muxid*`)`

where the *fd0* is the file descriptor associated with Stream connected to the multiplexing driver from above. The *muxid* is returned by the **`I_PLINK`** **`ioctl`** for the Stream that was connected below the multiplexor. The **`I_PUNLINK`** removes the persistent link between the multiplexor referenced by the *fd0* and the Stream to the driver designated by the *muxid*. Each of the bottom persistent links can be disconnected individually. An **`I_PUNLINK`** **`ioctl`** with the *muxid* value of `MUXID_ALL` removes all persistent links below the multiplexing driver referenced by *fd0*.

The following dismantles the previously given configuration:

```
fd = open("/dev/mux", O_RDWR);
/*
 * retrieve muxid from the file
 */
ioctl(fd, I_PUNLINK, muxid);
exit(0);
```

The use of the **`ioctl`**s **`I_PLINK`** and **`I_PUNLINK`** should not be intermixed with **`I_LINK`** and **`I_UNLINK`**. Any attempt to unlink a regular link with **`I_PUNLINK`** or to unlink a persistent link with **`I_UNLINK`** **`ioctl`** causes the `errno` value of `EINVAL` to be returned.

Because multilevel multiplexing configurations are allowed in STREAMS, it is possible to have a situation where persistent links exist below a multiplexor whose Stream is connected to the above multiplexor by regular links. Closing the file descriptor associated with the controlling Stream removes the regular link but not the persistent links below it. On the other hand, regular links are allowed to exist below a multiplexor whose Stream is connected to the above multiplexor with persistent links. In this case, the regular links are removed if the persistent link above is removed and no other references to the lower Streams exist.

The construction of cycles is not allowed when creating links. A cycle could be constructed by creating a persistent link of multiplexor 2 below multiplexor 1 and then closing the controlling file descriptor associated with the multiplexor 2 and reopening it again and then linking the multiplexor 1 below the multiplexor 2, but this is not allowed. The operating system prevents a multiplexor configuration from containing a cycle to ensure that messages cannot be routed infinitely, thus creating an infinite loop or overflowing the kernel stack.

14 Packaging Your Software Applications

14
Packaging Your Software Applications

An Overview of Software Packaging

This chapter describes how to package software. A packaging tool, the **`pkgmk`** command, is provided to help automate package creation. It gathers the components of a package on the development machine, copies them onto the installation medium, and places them into a structure that **`pkgadd`** recognizes.

This chapter also describes the installation tool, the **`pkgadd`** command, which copies the package from the installation medium onto a system and performs system housekeeping routines that concern the package. This tool is primarily for the installer but is described here to provide you with a background on the environment into which your packages will be placed and to help you test-install your packages.

The next two sections describe what a package consists of and gives an overview of the structural life cycle of a package (how its structure on your development machine relates to its structure on the installation medium and on the installation machine).

The remaining sections familiarize you with all of the tools, files, and scripts involved in creating a package, provide suggestions for how to approach software packaging, and describe some specific procedures.

A section on set packaging is at the end of this chapter. It describes the new concept of "sets," how packages are related to them, and how some of the packaging commands have been enhanced to support sets.

After reading this chapter, you should study the section entitled "Package Installation Case Studies" which provides case studies using the tools and techniques described in this chapter.

Packaging Tools and the Enhanced Security Utilities

If you have installed, or are planning to install, the Enhanced Security Utilities, you should execute the packaging tools in single-user mode only. This will ensure that privileges and Mandatory Access Control levels are handled correctly when you install, remove, and change packages and their associated files and scripts. On a system running the Enhanced Security Utilities, only the administrator responsible for maintaining the system configuration (the Trusted Systems Programmer) should be executing commands in single-user mode that alter the installed software base of the system.

If your package requires MLDs when the Enhanced Security Utilities are installed, your installation script must update the file **`/etc/security/MLD`** with the names of MLDs required for the package to function properly when the Enhanced Security Utilities are installed. When the Enhanced Security Utilities are installed, a startup script in **`/etc/rc2.d`** reads this file and creates the MLDs listed, commenting out the lines for the MLDs created so that they will be ignored the next time the script is run (that is, on the next transition to the multi-user state).

It is recommended that you write your packaging scripts and data files to take advantage of the Enhanced Security Utilities so that you do not need to maintain two sets of packaging files. The Enhanced Security features you need to consider in your packaging files are Mandatory Access Control levels, fixed and inheritable privileges, and MLDs (mentioned above). Levels and privileges are populated via fields in the **`pkgmap`** and **`prototype`** files.

Contents of a Package

A software package is made up of a group of components that together create the software. These components naturally include the executables that comprise the software, but they also include at least two information files and can optionally include other information files and scripts.

As shown in Figure 14-1, a package's contents fall into three categories:

- required components (the **`pkginfo`** file, the **`prototype`** file, package objects)
- optional package information files
- optional packaging scripts

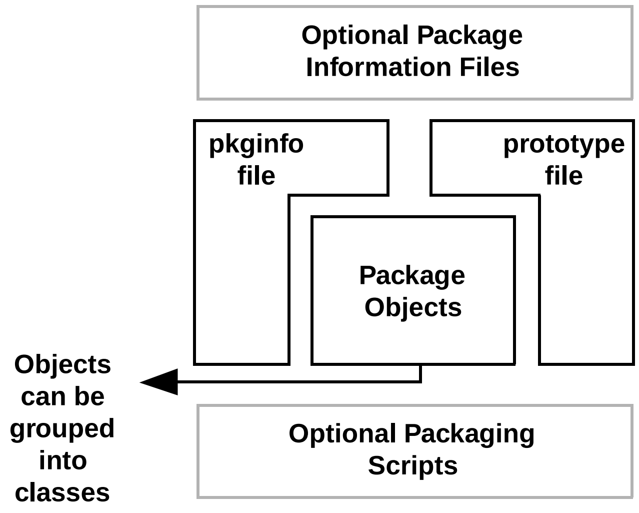

Optional Package Information Files
pkginfo file
prototype file
Package Objects
Objects can be grouped into classes
Optional Packaging Scripts

Figure 14-1. The Contents of a Package

Required Components

At the very least, a package must contain the following components:

- Package Objects

  These are the objects that make up the software. They can be files (executable or data), directories, or named pipes. Objects can be manipulated in groups during installation by placing them into classes. You will learn more about classes when reading the section “Step 3. Placing Objects into Classes”

- The **`pkginfo`** File

  The **`pkginfo`** file is a required package information file defining parameter values that describe a package. For example, this file defines values for the package abbreviation, the full package name, and the package architecture.

- The **`prototype`** File

  The **`prototype`** file is a required package information file that lists the contents of the package. There is one entry for each deliverable object and this entry consists of several fields of information describing the object. All package components, including the **`pkginfo`** file, must be listed in the **`prototype`** file.

Both required package information files are described further in “The Package Information Files” section and on their respective manual pages.

Optional Package Information Files

There are four optional package information files that you can add to your package:

- The **`compver`** File

  Defines previous versions of the package that are compatible with this version.

- The **depend** File

  Defines any software dependencies associated with this package.

- The **`space`** File

  Defines disk space requirements for the target environment beyond that used by objects defined in the **`prototype`** file (for example, files that will be dynamically created at installation time).

- The **`copyright`** File

  Defines the text for a copyright message that will be printed on the terminal at the time of package installation or removal.

Every package information file used must have an entry in the **`prototype`** file. All of these files are described further in the “The Package Information Files” section and on their respective manual pages.

Optional Installation Scripts

Your package can use three types of installation scripts, although no scripts are required. An installation script must be executable by **sh** (for example, a shell script or executable program). The three script types are the request script (solicits installer input), class action script (defines a set of actions to perform on a group of objects), and the procedure script (defines actions that will occur at particular points during installation).

Packaging scripts are described in detail in “The Installation Scripts” section. Example scripts can be found in the Case Studies.

The Structural Life Cycle of a Package

The material covered in this chapter talks about package object pathnames. While reading, keep in mind that a package object resides in three places while being packaged and installed. To help you avoid confusion, consider which of the three possible locations are being discussed:

- On a development machine

  Packages originate on a development machine. They can be in the same directory structure on your machine as they will be placed on the installation machine. Or **pkgmk** can locate components on the development machine and give them different pathnames on the installation machine.

- On the installation media

  When **pkgmk** copies the package components from the development machine to the installation medium, it places them into the structure you have defined in your **prototype** file and a format that **pkgadd** recognizes.

- On the installation machine

  pkgadd copies a package from the installation medium and places it in the structure defined in your **prototype** file. Package objects can be defined as relocatable, meaning the installer can define the actual location of these package objects on the installation machine during installation. Objects with fixed locations are copied to their predefined path.

The Package Creation Tools

The packaging tools are provided to automate package creation and to remove the burden of packaging from the developer. There are three packaging tools:

- From the components of a package on the development machine, **pkgmk** creates a package image in directory structure format.
- **pkgtrans** translates an installable package from one package format to another. The two format types are directory structure and datastream. For

example, after having used **pkgmk** to create a package in directory structure format, you might use **pkgtrans** to translate it into datastream format.

- **pkgproto** generates a **prototype** file based on the directory structure of your development area.

Each of these commands is described in the following text and on its manual page.

The pkgmk Command

This command takes all of the package objects residing on the development machine, optionally compresses them, copies them onto the installation medium, and places them into a fixed directory structure. You are not required to know the details of the fixed directory structure since **pkgmk** takes care of the formatting.

Files can be unstructured on the development machine and **pkgmk** will structure them correctly on the medium based on information supplied in the **prototype** file. The installation medium onto which a package is formatted can be what is typically thought of as a medium (cartridge tape, for example) or it can be a directory on a machine.

pkgmk requires the presence of two information files on the development machine, the **prototype** and the **pkginfo** file (other package information files may be present). The **pkginfo** file defines the values for a number of package parameters, such as the package abbreviation and the package name. The **prototype** file provides a complete list of the package contents. **pkgmk** creates the **pkgmap** file, which is the package contents file on the installation medium, by processing the **prototype** file and then adding three fields to each entry.

pkgmk follows these steps when processing a package:

1. Processes all of the command lines in the input **prototype** file. (**prototype** command lines can tell **pkgmk** where to look for package objects, merge other **prototype** files into this one, define default *mode owner group* for package objects, and place parameter values in the packaging environment.)
2. Copies the objects of a package onto the installation medium, using the **prototype** file as a listing of contents. If desired, the objects placed on the installation medium may be compressed.
3. Puts the package objects into the proper format.
4. Divides a package into pieces and distributes those pieces on multiple volumes, if necessary.
5. Creates the **pkgmap** file. (the content listing file that is placed on the installation medium). It resembles the **prototype** file except that all command lines are processed, and the `volno, size, cksum, and modtime` fields are added to each entry.

The pkgtrans Command

This command translates a package already created with **`pkgmk`** from one package format to another. It can make the following translations:

- a fixed directory structure to a datastream
- a datastream to a fixed directory structure
- a fixed directory structure to a fixed directory structure

Note that a package in a fixed directory structure can be in a directory on disk (for example, in a spooling directory) or on a removable device such as a cartridge tape. A datastream can be on any device; for example, on a disk or a tape.

The pkgproto Command

This command generates a **`prototype`** file. It scans the paths specified on the command line and creates description line entries for these paths. If the pathname is a directory, an entry for each object in the directory is generated. You can use the **`-c`** option of the **`pkgproto`** command to place objects into a particular class.

When you create a **`prototype`** file with an editor, it does not matter how package components are organized on your development machine. You use the *path1=path2* pathname format to define where the files reside on your development machine and where they should be placed on the installation machine. However, when you use **`pkgproto`** to create your file, your development area must be structured exactly as you want your package to be structured.

The Installation Tools

The installation tools provide capabilities to install and remove packages, create responses to prompts during installation of packages, check the accuracy of installed packages, and display information about software packages. These tools are introduced to you here so that you can understand the environment into which your package will be placed. Manual pages for these tools are provided in the back of this book. The installation tools are:

- **`pkgadd`** installs a package.
- **`pkgrm`** removes a package.
- **`pkgask`** stores answers to an interactive package (one with a request script) in a **`response`** file. Later, when installing the package, this file may be specified on the **`pkgadd`** command line so that the package may be installed in noninteractive mode.
- **`pkgchk`** checks the content and attribute information for an installed package to ensure that it was not corrupted during installation.
- **`pkginfo`** and **`pkgparam`** display information about packages.

The system administrator can set parameters that control various aspects of installation in an administration file called the **`admin`** file. Refer to the manual pages for more information on these commands and on the **`admin`** file.

The Package Information Files

Each of the six package information files will be described in the following pages. All of these files can be created using any editor. File formats are described in the following text and in full detail in their respective manual pages.

The six package information files are:

- the **`pkginfo`** file
- the **`prototype`** file
- the **`compver`** file
- the **`copyright`** file
- the **`depend`** file
- the **`space`** file

This section also describes the system-generated **`pkgmap`** file, which **`pkgmk`** creates and places on the installation medium. It is similar to the **`prototype`** file.

The pkginfo File

This required package information file defines parameter values that describe characteristics of the package, such as the package abbreviation, full package name, package version, and package architecture. The definitions in this file can set values for all of the installation parameters defined in the **`pkginfo`** manual page.

Each entry in the file uses the following format to establish the value of a parameter:

PARAM="value"

Figure 14-2 shows an example **`pkginfo`** file.

```
PKG="pkgA"
NAME="My Package A"
ARCH="nh68000"
RELEASE="4.0"
VERSION="2"
VENDOR="MYCOMPANY"
HOTLINE="1-800-245-6453"
VSTOCK="0122c3f5566"
CATEGORY="application"
ISTATES="S 2"
RSTATES="S 2"
```

Figure 14-2. Sample pkginfo File

The **`pkginfo`** and **`pkgparam`** commands can be used to access information in a **`pkginfo`** file.

NOTE

Before defining the `PKG`, `ARCH`, and `VERSION` parameters, you need to know how **`pkgadd`** defines a package instance and the rules associated with naming a package. Refer to the section, "Step 2. Defining a Package Instance" before assigning values to these parameters.

The prototype File

This required package information file contains a list of the package contents. The **`pkgmk`** command uses the **`prototype`** file to identify the contents of a package and their location on the development machine when building the package.

You can create this file in two ways. As with all the package information files, you can use an editor to create a file named **`prototype`**. It should contain entries following the description given later in this chapter. You can also use the **`pkgproto`** command to generate the file automatically. To make use of the second method, you must have a copy of your package on your development machine that is structured exactly as you want it structured on the installation machine and all modes and permissions must be correct. If you want to put the attributes `mac`, `fixed`, and `inherited` on any of the files in the prototype file, you must add them to the file manually after executing **`pkgproto`**. If you are not going to use **`pkgproto`**, you do not need a structured copy of your package.

There are two types of entries in the **`prototype`** file: description lines and command lines.

The Description Lines

You must create one description line for each deliverable object that consists of several fields describing the object. This entry describes such information as mode, owner, and group for the object. You can also use this entry to accomplish the tasks listed below.

- You can override **`pkgmk`**'s placement of an object on a multiple-part package. (Refer to the section "Step 11. Distributing Packages Over Multiple Volumes" for more details.)
- You can place objects into classes. (Refer to the section "Step 3. Placing Objects into Classes" for details.)
- You can tell **`pkgmk`** where to find an object in your development directory structure and map that name to the correct placement on the installation machine. (Refer to the section "Mapping Development Pathnames to Installation Pathnames" for details.)
- You can define an object as relocatable. (Refer to the section "Step 4. Making Package Objects Relocatable" for details.)
- You can define links. (Refer to the section "Step 9. Creating the prototype File" for details.)

The generic format of the descriptive line is:

[part] ftype class pathname [major minor] [mode owner group]
[part] ftype class pathname [major minor] [mode owner group] [mac fixed inherited]

Definitions for each field are as follows:

part	Designates the part in which an object should be placed. A package can be divided into a number of parts. A part is a collection of files and is the atomic unit by which a package is processed. A developer can choose the criteria for grouping files into a part (for example, by class). If not defined, **`pkgmk`** decides in which part the object will be placed.
ftype	Designates the file type of an object. Example file types are `f` (a standard executable or data file), `d` (a directory), `l` (a linked file), and `i` (a package information file). (Refer to the **`prototype`** manual page for a complete list of file types.)
class	Defines the class to which an object belongs. All objects must belong to a class. If the object belongs to no special class, this field should be defined as `none`.
pathname	Defines the pathname which an object should have on the installation machine. If you do not begin this name with a slash, the object is considered to be relocatable. You can use the form `path1=path2` to map the location of an object on your development machine to the pathname it should have when installed on a machine.

major/minor	Defines the major and minor numbers for a block or character special device.
mode/owner/group	Defines the mode, owner, and group for an object. The mode, owner, and group must be defined or packaging will fail. If not defined, the defaults defined with the **`default`** command are assigned.
mac	Defines the Mandatory Access Control (MAC) Level Identifier (LID), an integer value that specifies a combination of a hierarchical classification and zero or more non-hierarchical categories. This field can only be applied to a file on a sfs-type filesystem and is not used for linked files or packaging information files.
fixed	Defines a comma separated list of valid mnemonic fixed privilege names as defined for the **`filepriv`** command. The string `NULL` is used in place of the comma separated list when privilege is not to be specified. This field is not used for linked files or packaging information files.
inherited	Defines a comma separated list of valid mnemonic inherited privilege names as defined for the **`filepriv`** command. The string `NULL` is used in place of the comma separated list when privilege is not to be specified. This field is not used for linked files or packaging information files.

Figure 14-3 shows an example of this file with only description lines.

```
i pkginfo
i request
d bin /ncmpbin 0755 root other
f bin /ncmpbin/dired=/usr/ncmp/bin/dired 0755 root other
f bin /ncmpbin/less=/usr/ncmp/bin/less 0755 root other
f bin /ncmpbin/ttype=/usr/ncmp/bin/ttype 0755 root other
```

Figure 14-3. Sample #1 `prototype` File

The Command Lines

There are four types of commands that can be embedded in the **`prototype`** file. They are

`search` *pathnames*	Specifies a list of directories (separated by white space) in which **`pkgmk`** should search when looking for package objects. *pathnames* is prepended to the basename of each object in the **`prototype`** file until the object is located.

NOTE

The **search** command will not work when invoking **pkgmk** with the **-c** option specified to compress all noninformation package files.

include *filename* — Specifies the pathname of another **prototype** file that should be merged into this one during processing. (Note that **search** requests do not span **include** files. Each **prototype** file should have its own **search** command defined, if one is needed.)

default *mode owner group*

[mac fixed inherited]

Defines the default *mode owner group* that should be used if this information is not supplied in a **prototype** entry that requires the information. (The defaults do not apply to entries in any **include** files. Each **prototype** file should have its own **default** command defined, if one is needed.) The *mac, fixed,* and *inherited* fields can also be supplied.

param=value — Places the indicated parameter in the packaging environment. This allows you to expand a variable pathname so that **pkgmk** can locate the object without changing the actual object pathname. (This assignment will not be available in the installation environment.)

A command line must always begin with an exclamation point (`!`). Commands may have variable substitutions embedded within them.

Figure 14-4 shows an example **prototype** file with both description and command lines.

```
!PROJDIR=/usr/myname
!search /usr/myname/bin /usr/myname/src /usr/myname/hdrs
!include $PROJDIR/src/prototype
i pkginfo
i request
d bin ncmpbin 0755 root other
f bin ncmpbin/dired=/usr/ncmp/bin/dired 0755 root other
f bin ncmpbin/less=/usr/ncmp/bin/less 0755 root other
f bin ncmpbin/ttype=/usr/ncmp/bin/ttype 0755 root other
!default 755 root bin
```

Figure 14-4. Sample #2 prototype File

The compver File

This package information file defines previous (or future) versions of the package that are compatible with this version. Each line in the file consists of a string defining a version of the package with which the current version is compatible. Since some packages may require installation of a particular version of another software package, compatibility information is extremely crucial. If a package “A” requires version “1.0” of application “B” as a prerequisite, but the customer installing “A” has a new and improved version of “1.3” of “B”, the **compver** file for “B” must indicate that the new version is compatible with version “1.0” in order for the customer to install package “A”. The string must match the definition of the `VERSION` parameter in the **pkginfo** file of the package considered to be compatible. Figure 14-5 shows an example of this file.

```
Version 1.3
Version 1.0
```

Figure 14-5. Sample compver File

The copyright File

This package information file contains the text of a copyright message that will be printed on the terminal at the time of package installation or removal. The display is exactly as shown in the file. Figure 14-6 shows an example of this file

```
Copyright (c) 1989 HCSC
All Rights Reserved.


THIS PACKAGE CONTAINS UNPUBLISHED PROPRIETARY SOURCE CODE OF HCSC.

The copyright notice above does not evidence any
actual or intended publication of such source code.
```

Figure 14-6. Sample copyright File

This package information file is an optional file that contains interactive scripts for package installation. The **request** file contains a procedure script for situations that the Installation Tools do not handle. This request script will be described in the “The Installation Scripts” section of this chapter.

The depend File

This package information file defines software dependencies associated with the package. You can define three types of package dependencies with this file.

- a prerequisite package (meaning this package depends on the existence of another package)
- a reverse dependency (meaning another package depends on the existence of this package)
- an incompatible package (meaning your package is incompatible with this one)

The generic format of a line in this file is:

```
type pkg name
     (arch)version
     (arch)version
```

Definitions for each field are as follows:

`type`	Defines the dependency type. `P` indicates the named package is a prerequisite for installation. `I` indicates the named package is incompatible. `R` indicates a reverse dependency, that is, the named package requires that this package be on the system.
`pkg`	Indicates the package abbreviation for the package.
`name`	Specifies the full package name (used for display purposes only).
`(arch)version`	Defines a particular instance of a package by defining the architecture and version. If `(arch)version` is not supplied, it means the entry refers to any instance of the package.

Figure 14-7 shows an example of this file.

```
P acu    Advanced C Utilities
         Issue 4 Version 1
P cc     C Programming Language
         Issue 4 Version 1 (nh6800)
R vpkg   Another Vendor Package
```

Figure 14-7. Sample depend File

The space File

This package information file defines disk space requirements for the target environment beyond that which is used by objects defined in the **prototype** file—for example, files that will be dynamically created at installation time. It should define the maximum amount of additional space that a package will require.

The generic format of a line in this file is:

pathname blocks inodes

Definitions for each field are as follows:

`pathname`	Names a directory in which there are objects that will require additional space. The pathname may be the mount point for a file-system. Pathnames that do not begin with a slash (/) indicate relocatable directories.
`blocks`	Defines the number of 512 byte disk blocks required for installation of the files and directory entries contained in the pathname. (Do not include file-system dependent disk usage.)
`inodes`	Defines the number of inodes required for installation of the files and directory entries contained in `name`.

Numbers of blocks or inodes can be negative to indicate that the package will ultimately (after processing by scripts, and so on) take up less space than the installation tool would calculate.

Figure 14-8 shows an example of this file.

```
# extra space required by config data which is
# dynamically loaded onto the system
data 500 1
```

Figure 14-8. Sample space File

The pkgmap File

The **pkgmk** command creates the **pkgmap** file when it processes the **prototype** file. This new file contains all of the information in the **prototype** file plus three new fields for each entry. These fields are `size` (file size in bytes), `cksum` (checksum of file), and `modtime` (last time of modification). All command lines defined in the **prototype** file are executed as **pkgmk** creates the **pkgmap** file. The **pkgmap** file is placed on the installation medium. The **prototype** file is not. Refer to the **pkgmap** manual page for more details about this file.

The Installation Scripts

The **`pkgadd`** command automatically performs all of the actions necessary to install a package, using the package information files as input. As a result, you do not have to supply any packaging scripts. However, if you want to customize the installation procedures for your package needs, the following three types of scripts can be used:

request script	Solicits administrator interaction during package installation for the purpose of assigning or redefining environment parameter assignments.
class action scripts	Define an action or set of actions that should be applied to a class of files during installation or removal. You define your own classes or you can use one of three standard classes (**`sed, awk,`** and **`build`**). See the section “Step 3. Placing Objects into Classes” for details on how to define a class.
procedure scripts	Specifies a procedure to be invoked before or after the installation or removal of a package. The four procedure scripts are `preinstall, postinstall, preremove, and postremove.`

You decide which type of script to use based on when you want the script to execute. To help you with this assessment, script processing is discussed next, followed by a description of parameters available to packaging scripts, how to get information about a package for your scripts, and script exit codes. After that, each type of script is described in detail.

NOTE

All installation scripts must be executable by **`sh`** (for example, a shell script or an executable program).

Script Processing

You can customize the actions taken during installation by delivering installation scripts with your package. The decision on which type of script to use to meet a need depends upon when the action is needed during the installation process. As a package is installed, **`pkgadd`** performs the following steps:

- Executes the request script.

  This is the only point at which your package can solicit input from the installer.

- Executes the `preinstall` script.
- Installs the package objects.

Installation occurs class-by-class and class action scripts are executed accordingly. The list of classes operated upon and the order in which they should be installed is initially defined with the `CLASSES` parameter in your **`pkginfo`** file. However, your request script can change the value of `CLASSES`.

- Executes the `postinstall` script.

When a package is being removed, **`pkgrm`** performs these steps:

- Executes the `preremove` script.
- Executes the removal class action scripts.

  Removal also occurs class-by-class. As with the installation class action scripts, if more than one removal script exists, they are processed in the reverse order in which the classes were listed in the `CLASSES` parameter at the time of installation.
- Executes the `postremove` script.

The request script is not processed at the time of package removal. However, its output (a list of parameter values) is saved and so is available to removal scripts.

Installation Parameters

The following four groups of parameters are available to all installation scripts. Some of the parameters can be modified by a request script, others cannot be modified at all.

- The four system parameters that are generated by the installation software (see below for a description of these). None of these parameters can be modified by a package.
- The 21 standard installation parameters defined in the **`pkginfo`** file. Of these, a package can only modify the `CLASSES` parameter. (The standard installation parameters are described in detail in the **`pkginfo`** manual page.
- You can define your own installation parameters by assigning a value to them in the **`pkginfo`** file. Such a parameter must be alphanumeric with an initial capital letter. Any of these parameters can be changed by a request script.
- Your request script can define new parameters by assigning values to them and placing them into the installation environment, as shown in Figure 14-9.

```
# make parameters available to installation service
# and any other packaging script we might have
cat >$1 <<!
CLASSES='$CLASSES'
NCMPBIN='$NCMPBIN'
EMACS='$EMACS'
NCMPMAN='$NCMPMAN'
!
```

Figure 14-9. Placing Parameters into the Installation Environment

The four installation parameters that are generated by installation software are described below:

`PATH`	Specifies the search list used by **`sh`** to find commands; `PATH` is set to **`/sbin:/usr/sbin:/usr/bin:/usr/sadm/install/bin`** upon script invocation.
`UPDATE`	Indicates that the current installation is intended to update the system. Automatically set to `true` if the package being installed is overwriting a version of itself.
`PKGINST`	Specifies the instance identifier of the package being installed. If another instance of the package is not already installed, the value will be the package abbreviation. Otherwise, it is the package abbreviation followed by a suffix, such as `pkg.1`. (Multiple variations of the same package can reside simultaneously on the installation medium, as well as on the installation machine. Each variation is known as a package instance and assigned an instance identifier. See section “Step 2. Defining a Package Instance” for more details.)
`PKGSAV`	Specifies the directory where files can be saved for use by removal scripts or where previously saved files may be found.

Getting Package Information for a Script

There are two commands that can be used from your scripts to solicit information about a package.

The **`pkginfo`** command returns information about software packages, such as the instance identifier and package name.

The **`pkgparam`** command returns values for all parameters or only for the parameters specified.

The **`pkginfo(1)`** and **`pkgparam(1)`** manual pages give details for these tools.

Exit Codes for Scripts

Each script must exit with one of the following exit codes:

`0`	Successful completion of script.
`1`	Fatal error. Installation process is terminated at this point.
`2`	Warning or possible error condition. Installation will continue. A warning message will be displayed at the time of completion.
`3`	Script was interrupted and possibly left unfinished. Installation terminates at this point.
`10`	System should be rebooted when installation of all selected packages is completed. (This value should be added to one of the single-digit exit codes described above.)
`20`	The system should be rebooted immediately upon completing installation of the current package. (This value should be added to one of the single-digit exit codes described above.)

See the Case Studies for examples of exit codes in installation scripts.

The Request Script

The request script solicits interaction during installation and is the only place where your package can interact directly with the installer. It can be used, for example, to ask the installer if optional pieces of a package should be installed.

The output of a request script must be a list of parameters and their values. This list can include any of the parameters you created in the **`pkginfo`** file (not including the 21 standard parameters) and the `CLASSES` parameter. The list can also introduce parameters that have not been defined elsewhere.

When your request script assigns values to a parameter, it must then make those values available to the installation environment for use by **`pkgadd`** and also by other packaging scripts. The following example shows a request script segment that performs this task for the four parameters `CLASSES, NCMPBIN, EMACS, and NCMPMAN.`

Request Script Naming Conventions

There can only be one request script per package and it must be named `request`.

Request Script Usage Rules

1. The request script is executed as `uid=root` and `gid=other`. (Note that this does not conform to the Application Binary Interface requirement that `uid=install`.)

2. The request script should not modify any files, with the exception of the “response” file (described below) which is the output of the request script. It is intended only to interact with users and to create a list of parameter assignments based upon that interaction.
3. **`pkgadd`** calls the request script with one argument that names the file to which the output of this script will be written. This file is referred to as the **`response`** file.
4. The parameter assignments should be added to the installation environment for use by **`pkgadd`** and other packaging scripts (as shown in Figure 14-9).
5. System parameters and standard installation parameters, except for the `CLASSES` parameter, cannot be modified by a request script. Any of the other parameters available can be changed.
6. The format of the output list should be *PARAMETER*=“*value*”. For example:

   ```
   CLASSES=“none class1”
   ```
7. The list should be written to the file named as the argument to the request script.
8. The user's terminal is defined as standard input to the request script.
9. The request script is not executed during package removal. However, the parameter values assigned in the script are saved and are available during removal.

Soliciting User Input in Request Scripts

A tool is provided in the base package for generating full-screen menus for handling user input. This tool is called **`menu`**, and should be used when user input is solicited in a request script. Using a form description file [see **`menu(4)`**], **`menu`** generates a full-screen form that can be used for displaying information, entering a selection from a numbered list, or filling out a more complex, multiple-field form. The **`menu`** tool may also contain help text so that a user can get more information about the current step in the installation.

The **`menu`** tool output is a file that contains Bourne/Korn shell statements of the form:

```
VARIABLE=“value”
```

After the user completes the menu, this output file can be read in and executed in the request script using the shell '.' command. The values obtained may be used to generate the response file, the output of the request script.

The **`menu`** tool should be used when soliciting user input from the request script. The **`menu`** tool can also be used in the postinstall script for a package to inform the user about the status of the installation at completion. (See **`menu(1)`** and **`menu(4)`** for more information.)

The Class Action Script

The class action script defines a set of actions to be executed during installation or removal of a package. The actions are performed on a group of pathnames based on their class definition. (See the "Package Installation Case Studies" section for examples of class action scripts.)

Class Action Script Naming Conventions

The name of a class action script is based on which class it should operate and whether those actions should occur during package installation or removal. The two name formats are:

- *i.class* (operates on pathnames in the indicated class during package installation)
- *r.class* (operates on pathnames in the indicated class during package removal)

For example, the name of the installation script for a class named `class1` would be `i.class1` and the removal script would be named `r.class1`.

Class Action Script Usage Rules

1. Class action scripts are executed as `uid=root` and `gid=other`.
2. If a package spans more than one volume, the class action script will be executed once for each volume that contains at least one file belonging to the class. Consequently, each script must be "multiply executable." This means that executing a script any number of times with the same input must produce the same results as executing the script only once.

NOTE

The installation service relies upon this condition being met.

3. The script is executed only if there are files in the given class existing on the current volume.
4. **`pkgadd`** (and **`pkgrm`**) creates a list of all objects listed in the **`pkgmap`** file that belong to the class. As a result, a class action script can only act upon pathnames defined in the **`pkgmap`** and belonging to a particular class.
5. A class action script should never add, remove, or modify a pathname or system attribute that does not appear in the list generated by **`pkgadd`** unless by use of the **`installf`** or **`removef`** command.

   (See the manual pages for details on these two commands and the "Package Installation Case Studies" section for examples of them in use.)

6. When the class action script executes for the last time (meaning the input pathname is the last path on the last volume containing a file of this class), it is executed with the keyword argument `ENDOFCLASS`. This flag allows you to include post-processing actions into your script.

Installation of Classes

The following steps outline the system actions that occur when a class is installed. The actions are repeated once for each volume of a package as that volume is being installed.

1. **`pkgadd`** creates a pathname list.

   `pkgadd` creates a list of pathnames upon which the action script will operate. Each line of this list consists of source and destination pathnames, separated by white space. The source pathname indicates where the object to be installed resides on the installation volume and the destination pathname indicates the location on the installation machine where the object should be installed. The contents of the list is restricted by the following criteria:

   - The list contains only pathnames belonging to the associated class.
   - Directories, named pipes, character/block devices, and symbolic links are included in the list with the source pathname set to **`/dev/null`**. They are automatically created by **`pkgadd`** (if not already in existence) and given proper attributes (mode, owner, group) given proper attributes (mode, owner, group, mac, fixed, inherited) as defined in the **`pkgmap`** file.
   - Linked files are not included in the list, that is, files where **`ftype`** is `l`. (**`ftype`** defines the file type and is defined in the prototype file.) Links in the given class are created in Step 4.
   - If a pathname already exists on the target machine and its contents are no different from the one being installed, the pathname will not be included in the list.

     To determine this, **`pkgadd`** compares the `cksum`, `modtime`, and `size` fields in the installation software database with the values for those fields in your **`pkgmap`** file. If they are the same, it then checks the actual file on the installation machine to be certain it really has those values. If the field values are the same and are correct, the pathname for this object will not be included in the list.

2. If there is no class action script, the files associated with the pathnames are copied to the target machine.

   If no class action script is provided for installation of a particular class, the files in the generated pathname list will simply be copied from the volume to the appropriate target location.

3. If there is a class action script, the script is executed.

   The class action script is invoked with standard input containing the list generated in Step 1. If this is the last volume of the package and there are no more objects in this class, the script is executed with the single argument of `ENDOFCLASS`.

4. **`pkgadd`** performs a content and attribute audit and creates links.

   After successfully executing Step 2 or 3, an audit of both content and attribute information is performed on the list of pathnames. **`pkgadd`** creates the links associated with the class automatically. Detected attribute inconsistencies are corrected for all pathnames in the generated list.

Removal of Classes

Objects are removed class-by-class. Classes that exist for a package, but are not listed in the `CLASSES` parameter are removed last (for example, an object installed with the **`installf`** command). Classes that are listed in the `CLASSES` parameter are removed in reverse order. The following steps outline the system actions that occur when a class is removed:

1. **`pkgrm`** creates a pathname list.

   `pkgrm` creates a list of installed pathnames that belong to the indicated class. Pathnames referenced by another package are excluded from the list unless their ftype is `e` (meaning the file may be edited upon installation or removal).

   If a pathname is referenced by another package, it will not be removed from the system. However, if it is of ftype `e`, it may be modified to remove information placed in it by the package being removed. The modification should be performed by the removal class action script.

2. If there is no class action script, the pathnames are removed.

   If your package has no removal class action script for the class, all of the pathnames in the list generated by **`pkgrm`** will be removed.

NOTE

You should always assign a class for files with an **`ftype`** of `e` (editable) and have an associated class action script for that class. Otherwise, they will be removed at this point, even if the pathname is shared with other packages.

3. If there is a class action script, the script is executed.

   `pkgrm` invokes the class action script with standard input containing the list generated in Step 1.

4. **`pkgrm`** performs an audit.

   Upon successful execution of the class action script, knowledge of the pathnames is removed from the system unless a pathname is referenced by another package.

The Special System Classes

The system provides three special classes. They are:

- The **sed** class (provides a method for using **sed** instructions to edit files upon installation and removal).
- The **awk** class (provides a method for using **awk** instructions to edit files upon installation and removal).
- The **build** class (provides a method to construct a file dynamically during installation).

The sed Class Script

The **sed** installation class provides a method of installing and removing objects that require modification to an existing object on the target machine. (The file must have already been installed by another package.) A **sed** class action script delivers **sed** instructions in the format shown in Figure 14-10. You can give instructions that will be executed during either installation or removal. Two commands indicate when instructions should be executed. **sed** instructions that follow the **!install** command are executed during package installation and those that follow the **!remove** command are executed during package removal. It does not matter in which order the commands are used in the file.

The **sed** class action script executes automatically at installation time if a file belonging to class **sed** exists. The name of the **sed** class file should be the same as the name of the file upon which the instructions will be executed.

```
# comment, which may appear on any line in the file
!install
# sed(1) instructions which are to be invoked during
# installation of the object
[address [,address]] function [arguments]
   . . .

!remove
# sed(1) instructions to be invoked during the removal process
[address [,address]] function [arguments]
   . . .
```

Figure 14-10. sed Script Format

address, function, and arguments are as defined in the **sed(1)** manual page. See “Package Installation Case Studies” Case #5a and Case #5b for examples of **sed** class action scripts.

The awk Class Script

The **awk** installation class provides a method of installing and removing objects that require modification to an existing object on the target machine (the object must have been previously installed from another package installation). Modifications are delivered as **awk** instructions in an **awk** class action script.

The **awk** class action script executes automatically at the time of installation if a file belonging to class **awk** exists. Such a file contains instructions for the **awk** class script in the format shown in Figure 14-11. Two commands indicate when instructions should be executed. **awk** instructions that follow the **!install** command are executed during package installation and those that follow the **!remove** command are executed during package removal. It does not matter in which order the commands are used in the file.

```
# comment, which may appear on any line in the file
!install
# awk(1) program to install changes
   . . . (awk program)

!remove
# awk1(1) program to remove changes
   . . . (awk program)
```

Figure 14-11. awk Script Format

The name of the **awk** class file should be the same as the name of the file upon which the instructions will be executed.

The file to be modified is used as input to **awk** and the output of the script ultimately replaces the original object. Parameters may not be passed to **awk** using this syntax.

See “Package Installation Case Studies” “Case #5a”, for example **awk** class action scripts.

The build Class Script

The **build** class installs or removes objects by executing instructions that create or modify the object file. These instructions are delivered as a **build** class action script.

The name of the instruction file should be the same as the name of the file upon which the instructions will be executed.

The **build** class action script executes automatically at installation time if a file belonging to class **build** exists.

A **build** script must be executable by **sh**. The script's output becomes the new version of the file as it is built.

See “Package Installation Case Studies” “Case #5c”, for an example **build** class action script.

The Procedure Script

The procedure script gives a set of instructions that are performed at particular points in installation or removal. Four possible procedure scripts are described below. (The Case Studies in the back of the chapter show examples of procedure scripts.)

Naming Conventions for Procedure Scripts

The four procedure scripts must use one of the names listed below, depending on when these instructions are to be executed.

- `preinstall` (executes before class installation begins)
- `postinstall` (executes after all volumes have been installed)
- `preremove` (executes before class removal begins)
- `postremove` (executes after all classes have been removed)

Procedure Script Usage Rules

1. Procedure scripts are executed as `uid=root` and `gid=other`.
2. Each installation procedure script must use the **`installf`** command to notify **`pkgadd`** that it will add or modify a pathname. After all additions or modifications are complete, this command should be invoked with the **`-f`** option to indicate all additions and modifications are complete. (See the **`installf`** manual page and the “Package Installation Case Studies” section for details and examples.)
3. Each removal procedure script must use the **`removef`** command to notify **`pkgrm`** that it will remove a pathname. After removal is complete, this command should be invoked with the **`-f`** option to indicate all removals have been completed. (See the **`removef`** manual page and the “Package Installation Case Studies” section for details and examples.)

NOTE

The **`installf`** and **`removef`** commands must be used for the following reasons. If a procedure script physically removes an object from the system, the system's contents database will still contain an entry for that object until the **`removef`** command is used to remove the entry. Similarly, if a procedure script places an object on the system, it will not be registered in the contents database until the **`installf`** command is used to register the object.

Basic Steps of Packaging

What steps you take to create a package depend on how customized your package will be; therefore, it is difficult to give you a step-by-step guide on how to proceed. Your first step should be to plan your packaging. For example, you must decide on which package information files and scripts your package needs.

The following list outlines some of the steps you might use in a packaging scenario. Not all of these steps are required and there exists no mandated order for their execution (although you must have all of your package objects together before executing **`pkgmk`**). The remainder of this chapter gives procedural information for each step.

NOTE

This list, and the following procedures, are intended only as guidelines. These guidelines can not substitute for reading the rest of this chapter to learn what options are available to your package, and do your own individualized planning.

1. Assign a package abbreviation.

   Every package installed must have a package abbreviation.

2. Define a package instance.

   You must decide on values for the three package parameters that will make each package instance unique. (You need to understand what a package instance is, how it is defined, what the instance identifier is, and how to use that identifier. All of this is covered in the section “Step 2. Defining a Package Instance.”)

3. Place your objects into classes.

   You must decide on what installation classes you are going to use before you can create the **`prototype`** file and also before you can write your class action scripts.

4. Set up a package and its objects as relocatable.

   Package objects can be delivered with either fixed locations, meaning that their location is defined by the package and cannot be changed, or with relocatable locations, meaning that they have no absolute location requirements. All of a package or parts of a package can be defined as relocatable. You should decide if package objects will have fixed locations or be relocatable before you write any installation scripts and before you create the **`prototype`** file.

5. Decide which installation scripts your package needs.

   You must assess the needs of your package beyond the actions provided by **`pkgadd`** and decide on which type of installation scripts will allow you to deliver your customized actions.

6. Define package dependencies

   You must decide if your package has dependencies on other packages and if any other packages depend on yours.

7. Write a copyright message.

   You must decide if your package requires a copyright message to appear as it is being installed (and removed) and, if so, you must write that message.

8. Create the **`pkginfo`** file.

   You must create a **`pkginfo`** file before executing **`pkgmk`**. It defines basic information concerning the package and can be created with any editor as long as it follows the format described earlier in this chapter and in the **`pkginfo`** manual page.

   Create the **`prototype`** file.

   This file is required and must be created before you execute **`pkgmk`**. It lists all of the objects that belong to a package and information about each object (such as its file type and to which class it belongs). You can create it using any editor and you must follow the format described earlier in this chapter and in the **`prototype`** manual page. You can also use the **`pkg-proto`** command to generate a **`prototype`** file.

9. Distribute packages over multiple volumes.

   `pkgmk` automatically distributes packages over multiple volumes. You must decide if you want to leave those calculations up to **`pkgmk`** or customize package placement on multiple volumes.

10. Create the package.

    Create the package using the **`pkgmk`** command, which copies objects from the development machine to the installation medium, puts them into the proper structure, and automatically spans them across multiple volumes, if necessary.

    This is always the last step of packaging, unless you want to create a datastream structure for your package. If so, you must execute **`pkgtrans`** after creating a package with **`pkgmk`**.

Step 1. Assigning a Package Abbreviation

Each package installed must have a package abbreviation assigned to it. This abbreviation is defined with the `PKG` parameter in the **`pkginfo`** file.

A valid package abbreviation must meet the criteria defined below:

- It must start with an alphabetic character.

- Additional characters may be alphanumeric and contain the two special characters `+` and `-`.
- It cannot be longer than nine characters.
- Reserved names are `install`, `new`, and `all`.

Step 2. Defining a Package Instance

The same software package can differ by version or architecture or both. Multiple variations of the same package can reside simultaneously on the same machine. Each variation is known as a package instance. **`pkgadd`** assigns a package identifier to each package instance at the time of installation. The package identifier is the package abbreviation with a numerical suffix. This identifier distinguishes an instance from any other package, including other instances of the same package.

Identifying a Package Instance

Three parameters defined in the **`pkginfo`** file combine to identify each instance uniquely. You cannot assign identical values for all three parameters for two instances of the same package installed in the same target environment. These parameters are:

- `PKG` (defines the software package abbreviation and remains constant for every instance of a package)
- `VERSION` (defines the software package version)
- `ARCH` (defines the software package architecture)

For example, you might identify two identical versions of a package that run on different hardware as:

Instance #1	Instance #2
PKG="*abbr*"	PKG="*abbr*"
VERSION="release 1"	VERSION="release 1"
ARCH="hn5800"	ARCH="nh6800"

Two different versions of a package that run on the same hardware might be identified as:

Instance #1	Instance #2
PKG="*abbr*"	PKG="*abbr*"
VERSION="release 1"	VERSION="release 2"
ARCH="nh6800"	ARCH="nh6800"

The instance identifier, assigned by **pkgadd**, maps the three pieces of information that identify an instance to one name consisting of the package abbreviation plus a suffix. The first instance of a package installed on a system does not have a suffix and so its instance identifier will be the package abbreviation. Subsequent instances receive a suffix, beginning with `.2`. An instance is given the lowest integer extension available and so may not correspond to the order in which a package was installed. For example, if **mypkg.2** was deleted after **mypkg.3** was installed, the next instance to be added will be named **mypkg.2**. Because the number of instances of a particular package can vary from machine to machine, the instance identifier can also vary.

NOTE

pkgmk also assigns an instance identifier to a package as it places it on the installation medium if one or more instances of a package already exists. That identifier bears no relationship to the identifier assigned to the same package on the installation machine.

Accessing the Instance Identifier in Your Scripts

Because the instance identifier is assigned at the time of installation and will differ from machine to machine, you should use the `PKGINST` system parameter to reference your package in your installation scripts.

Step 3. Placing Objects into Classes

Installation classes allow a series of actions to be performed on a group of package objects at the time of their installation or removal. You place objects into a class in the **prototype** file. All package objects must be given a class, although the class of `none` may be used for objects that require no special action.

The installation parameter `CLASSES`, defined in the **pkginfo** file, is a list of classes to be installed (including the `none` class). Objects defined in the **prototype** file that belong to a class not listed in this parameter will not be installed. The actions to be performed on a class (other than simply copying the components to the installation machine) are defined in a class action script. These scripts are named after the class itself.

For example, to define and install a group of objects belonging to a class named `class1`, follow these steps:

1. Define the objects belonging to `class1` as such in their **prototype** file entry. For example,

   ```
   f class1 /usr/src/myfile
   f class1 /usr/src/myfile2
   ```

2. Ensure that the `CLASSES` parameter in the **pkginfo** file has an entry for `class1`. For example,

   ```
   CLASSES=“class1 class2 none”
   ```

NOTE

Package objects cannot be removed by class.

3. Ensure that a class action script exists for this class. An installation script for a class named `class1` would be named `i.class1` and a removal script would be named `r.class1`.

   If you define a class but do not deliver a class action script, the only action taken for that class will be to copy components from the installation medium to the installation machine.

In addition to the classes that you can define, the system provides three standard classes for your use. The **`sed`** class provides a method for using **`sed`** instructions to edit files upon package installation and removal. The **`awk`** class provides a method for using **`awk`** instructions to edit files upon package installation and removal. The **`build`** class provides a method to construct a file dynamically during package installation.

Step 4. Making Package Objects Relocatable

Package objects can be delivered either with fixed locations, meaning that their location on the installation machine is defined by the package and cannot be changed, or as relocatable, meaning that they have no absolute location requirements on the installation machine. The location for relocatable package objects is determined during the installation process.

You can define two types of relocatable objects: collectively relocatable and individually relocatable. All collectively relocatable objects are placed relative to the same directory once the relocatable root directory is established. Individually relocatable objects are not restricted to the same directory location as collectively relocatable objects.

Defining Collectively Relocatable Objects

Follow these steps to define package objects as collectively relocatable:

1. Define a value for the `BASEDIR` parameter.

   Put a definition for the `BASEDIR` parameter in your **`pkginfo`** file. This parameter names a directory where relocatable objects will be placed by default. If you supply no value for `BASEDIR`, no package objects will be considered as collectively relocatable.

2. Define objects as collectively relocatable in the **`prototype`** file.

   An object is defined as collectively relocatable by using a relative pathname in its entry in the **`prototype`** file. A relative pathname does not begin with a slash. For example, **`src/myfile`** is a relative pathname, while **`/src/myfile`** is a fixed pathname.

NOTE

A package can deliver some objects with relocatable locations and others with fixed locations.

All objects defined as collectively relocatable will be put under the same root directory on the installation machine. The root directory value will be one of the following (and in this order):

- if the **`admin`** file contains **`basedir=ask`** and **`pkgadd`** was not invoked in non-interactive mode, then the installer's response to **`pkgadd`** when asked where relocatable objects should be installed (this overrides the value for `BASEDIR` in the package's **`pkginfo`** file, if any)
- the value of `BASEDIR` as it is defined in the **`admin`** file used during the **`pkgadd`** process (the `BASEDIR` value assigned in the **`admin`** file overrides the value of the **`pkginfo`** file)
- the value of `BASEDIR` as it is defined in your **`pkginfo`** file (this value is used only as a default in case the other two possibilities have not supplied a value)
- if the **`admin`** file contains `basedir=default` and no `BASEDIR` is set in the package's **`pkginfo`** file, then `BASEDIR` defaults to `/`

Defining Individually Relocatable Objects

A package object is defined as individually relocatable by using a variable in its pathname definition in the **`prototype`** file. Your request script must query the installer on where such an object should be placed and assign the response value to the variable. **`pkgadd`** will expand the pathname based on the output of your request script at the time of installation. Case Study 1 shows an example of the use of variable pathnames and the request script needed to solicit a value for the base directory.

Step 5. Writing Your Installation Scripts

You should read the section “The Installation Scripts” to learn what types of scripts you can write and how to write them. You can also look at the Case Studies to see how the various scripts can be utilized and to see examples. Remember, you are not required to write any installation scripts for a package. Remember, you are not required to write any installation scripts for a package, though you must do so to take advantage of certain features, such as Multilevel Directories (MLDs). The **`pkgadd`** command performs all of the actions necessary to install your package, using the information you supply with the package information files. Any installation script that you write will be used to perform customized actions beyond those executed by **`pkgadd`**.

For example, if your package requires MLDs when the Enhanced Security Utilities are installed, your installation script must update the file **`/etc/security/MLD`** with the names of MLDs required for the package to function properly when the Enhanced Security Utilities are installed. When the Enhanced Security Utilities are installed, a startup script in **`/etc/rc2.d`** reads this file and creates the MLDs listed, commenting out the lines for the MLDs created so that they will be ignored the next time the script is run (that is, on the next transition to the multi-user state).

NOTE

Be certain that every installation script being delivered with your package has an entry in the **`prototype`** file. The file type should be `i`.

Reserving Additional Space on the Installation Machine

`pkgadd` assures that there is enough disk space to install your package, based on the object definitions in the **`pkgmap`** file. However, sometimes your package will require additional disk space beyond that needed by the objects defined in the **`pkgmap`** file. For example, your package might create a file during installation. **`pkgadd`** checks for additional space when you deliver a **`space`** file with your package. Refer to the section "The space File" earlier in this chapter or the **`space`** manual page for details on the format of this file.

NOTE

Be certain that your **`space`** file has an entry in the **`prototype`** file. Its file type should be `i` (for package information file).

Step 6. Defining Package Dependencies

Package dependencies and incompatibilities can be defined with two of the optional package information files. Delivering a **`compver`** file lets you name versions of your package that are compatible with the one being installed. Delivering a **depend** file lets you define three types of dependencies associated with your package. These dependency types are:

- a prerequisite package (meaning your package depends on the existence of another package)
- a reverse dependency (meaning another package depends on the existence of your package)

NOTE

This type should only be used when a pre-UNIX System V Release 4 package (that cannot deliver a **depend** file) relies on the newer package.

- an incompatible package (meaning your package is incompatible with this one)

Refer to the sections “The depend File” and “The compver File” earlier in this chapter, or the manual pages **`depend`** and **`compver`** for details on the formats of these files.

NOTE

Be certain that your **`depend`** and **`compver`** files have entries in the **`prototype`** file. The file type should be `i` (for package information file).

Step 7. Writing a Copyright Message

To deliver a copyright message, you must create a copyright file named **`copyright`**. The message will be displayed exactly as it appears in the file (no formatting) as the package is being installed and as it is being removed. Refer to the section “The copyright File” earlier in this chapter or the **`copyright`** manual page for more detail.

NOTE

Be certain that your **`copyright`** file has an entry in the **`prototype`** file. Its file type should be `i` (for package information file).

Step 8. Creating the pkginfo File

The **`pkginfo`** file establishes values for parameters that describe the package and is a required package component. The format for an entry in this file is:

PARAM=“value”

PARAM can be any of the 21 standard parameters described in the **`pkginfo`** manual page. You can also create your own package parameters simply by assigning a value to them in this file. Your parameter names must begin with a capital letter followed by either upper or lowercase letters.

The following five parameters are required:

- `PKG` (package abbreviation)
- `NAME` (full package name)
- `ARCH` (package architecture)
- `VERSION` (package version)
- `CATEGORY` (package category)

The `CLASSES` parameter dictates which classes are installed and the order of installation. Although the parameter is not required, no classes will be installed without it. Even if you have no class action scripts, the `none` class must be defined in the `CLASSES` parameter before objects belonging to that class will be installed.

NOTE

You can choose to define the value of `CLASSES` with a request script and not to deliver a value in the **`pkginfo`** file.

Step 9. Creating the prototype File

The **`prototype`** file is a list of package contents and is a required package component.

You can create the **`prototype`** file by using any editor and following the format described in the section “The prototype File” and in the **`prototype`** manual page. You can also use the **`pkgproto`** command to create one automatically.

Creating the File Manually

While creating the **`prototype`** file, you must at the very least supply the following three pieces of information about an object:

- The object's type

  All of the possible object types are defined in the **`prototype`** manual page. `f` (for a data file), `l` (for a linked file), and `d` (for a directory) are examples of object types.

- The object's class

  All objects must be assigned a class. If no special handling is required, you can assign the class `none`.

- The object`s pathname

  The pathname can define a fixed pathname such as **`/mypkg/src/filename`**, a collectively relocatable pathname such as **`src/filename,`** and an individually relocatable pathname such as **`$BIN/filename`** or **`/opt/$PKGINST/filename`**.

Creating Links

To define links you must do the following in the **`prototype`** entry for the linked object:

1. Define its **`ftype`** as `l` (a link) or `s` (a symbolic link).
2. Define its pathname with the format *path1=path2* where *path1* is the destination and *path2* is the source file.

Mapping Development Pathnames to Installation Pathnames

If your development area is in a different structure than you want the package to be in on the installation machine, you can use the **prototype** entry to map one pathname to the other. You use the *path1=path2* format for the pathname as is used to define links. However, since the **ftype** is not defined as `l` or `s`, *path1* is interpreted as the pathname you want the object to have on the installation machine, and *path2* is interpreted as the pathname the object has on your development machine.

For example, your project might require a development structure that includes a project root directory and numerous **src** directories. However, on the installation machine you might want all files to go under a package root directory and for all **src** files to be in one directory. So, a file on your machine might be named **/projdir/srcA/filename**. If you want that file to be named **/pkgroot/src/filename** on the installation machine, your **prototype** entry for this file might look like this

```
f class1 /pkgroot/src/filename=/projdir/srcA/filename
```

Defining Objects for pkgadd to Create

You can use the **prototype** file to define objects that are not actually delivered on the installation medium. **pkgadd** creates objects with the following **ftype**s if they do not already exist at the time of installation:

- `d` (directories)
- `x` (exclusive directories)
- `l` (linked files)
- `s` (symbolically linked files)
- `p` (named pipes)
- `c` (character special device)
- `b` (block special device)

To request that one of these objects be created on the installation machine, you should add an entry for it in the **prototype** file using the appropriate **ftype**.

For example, if you want a directory created on the installation machine, but do not want to deliver it on the installation medium, an entry for the directory in the **prototype** file is sufficient. An entry such as the one shown below will cause the directory to be created on the installation machine, even if it does not exist on the installation medium.

```
d none /directoryA 644 root other
```

Using the Command Lines

There are four types of commands that you can put into your **prototype** file. They allow you to do the following:

- Nest **prototype** files (the **include** command)
- Define directories for **pkgmk** to look in when attempting to locate objects as it creates the package (the **search** command)

NOTE

This will not work if **pkgmk** is instructed to compress the package.

- Set a default value for `mode owner group mac fixed inherited` (the **default** command). If all or most of your objects have the same values, using the **default** command will keep you from having to define these values for every entry in the **prototype** file.
- Assign a temporary value for variable pathnames to tell **pkgmk** where to locate these relocatable objects on your machine (with *param=value)*

Creating the File Using pkgproto

The **pkgproto** command scans your directories and generates a **prototype** file. **pkgproto** cannot assign **ftype**s of `v` (volatile files), `e` (editable files), or `x` (exclusive directories). You can edit the **prototype** file and add these **ftype**s, as well as perform any other fine-tuning you require (for example, adding command lines or classes).

pkgproto writes its output to the standard output. To create a file, you should redirect the output to a file. The examples shown in this section do not perform redirection in order to show you what the contents of the file would like.

Creating a Basic prototype

The standard format of **pkgproto** is

`pkgproto` *path* `[ . . . ]`

where *path* is the name of one or more paths to be included in the **prototype** file. If *path* is a directory, then entries are created for the contents of that directory as well (everything below that directory).

With this form of the command, all objects are placed into the `none` class and are assigned the same `mode owner group` as exists on your machine. The following example shows **pkgproto** being executed to create a file for all objects in the directory **/home/pkg**:

```
$ pkgproto /home/pkg
d none /home/pkg 755 bin bin
f none /home/pkg/file1 755 bin bin
f none /home/pkg/file2 755 bin bin
f none /home/pkg/file3 755 bin bin
f none /home/pkg/file4 755 bin bin
f none /home/pkg/file5 755 bin bin
$
```

To create a **prototype** file that contains the output of the example above, you would execute **pkgproto /home/pkg > prototype**

NOTE

If no pathnames are supplied when executing **`pkgproto`**, standard in (stdin) is assumed to be a list of paths. Refer to the **`pkgproto`** manual page for details on this usage.

Assigning Objects to a Class

You can use the *-c class* option of **`pkgproto`** to assign objects to a class other than `none`. When using this option, you can only name one class. To define multiple classes in a **`prototype`** file created by **`pkgproto`**, you must edit the file after its creation.

The following example is the same as above except the objects have been assigned to `class1`.

```
$ pkgproto -c class1 /home/pkg
d class1 /home/pkg 755 bin bin
f class1 /home/pkg/file1 755 bin bin
f class1 /home/pkg/file2 755 bin bin
f class1 /home/pkg/file3 755 bin bin
f class1 /home/pkg/file4 755 bin bin
f class1 /home/pkg/file5 755 bin bin
$
```

Renaming Pathnames with pkgproto

You can use a *path1=path2* format on the **`pkgproto`** command line to give an object a different pathname in the **`prototype`** file than it has on your machine. You can, for example, use this format to define relocatable objects in a **`prototype`** file created by **`pkgproto`**.

The following example is like the others shown in this section, except that the objects are now defined as `bin` (instead of **`/usr/bin`**) and are thus relocatable.

```
$ pkgproto -c class1 /home/pkg=bin
d class1 bin 755 bin bin
f class1 bin/file1 755 bin bin
f class1 bin/file2 755 bin bin
f class1 bin/file3 755 bin bin
f class1 bin/file4 755 bin bin
f class1 bin/file5 755 bin bin
$
```

pkgproto and Links

`pkgproto` detects linked files and creates entries for them in the **`prototype`** file. If multiple files are linked together, it considers the first path encountered the source of the link.

If you have symbolic links established on your machine but want to generate an entry for that file with an **`ftype`** of `f` (file), then use the **`-i`** option of **`pkgproto`**. This option creates a file entry for all symbolic links.

Step 11. Distributing Packages Over Multiple Volumes

As a packager, you need not worry about placing package components on multiple volumes. **`pkgmk`** performs the calculations and actions necessary to organize a multiple volume package.

However, you can use the optional `part` field in the **`prototype`** file to define in which part you want an object to be placed. A number in this field overrides **`pkgmk`** and forces the placement of the component into the part given in the field. Note that there is a one-to-one correspondence between parts and volumes for removable media.

Step 12. Creating a Package with pkgmk

`pkgmk` takes all of the objects on your machine (as defined in the **`prototype`** file), puts them in the fixed directory format and copies everything to the installation medium.

To package your software, execute

```
pkgmk [-d device] [-f filename]
```

You must use the **`-d`** option to name the device onto which the package should be placed. *device* can be a directory pathname or the identifier for a disk. The default device is the installation spool directory.

`pkgmk` looks for a file named **`prototype`**. You can use the **`-f`** option to specify a package contents file named something other than **`prototype`**. This file must be in the **`prototype`** format.

For example, executing **`pkgmk -d`** diskette1 creates a package based on a file named **`prototype`** in your current directory. The package will be formatted and copied to the diskette in the device `diskette1`.

Package File Compression

In Release 4.2, the **`pkgmk`** command has been enhanced to optionally compress package files. If the **`-c`** option is specified, **`pkgmk`** will compress all non-information files. The following exceptions apply.

If, as a result of compression, the size of the file is not reduced, **`pkgmk`** will not compress the file. If the pathname for the file in the package's **`prototype`** file is a relative pathname, (for example, **`../mypkg/foo`**), the file will not be compressed.

Creating a Package Instance

`pkgmk` will create a new instance of a package if one already exists on the device to which it is writing. It will assign the package an instance identifier. Use the **`-o`** option of **`pkgmk`** to overwrite an existing instance of a package rather than to create a new one.

Helping pkgmk Locate Package Contents

The following list describes situations that might require supplying **`pkgmk`** with extra information and an explanation of how to do so:

- Your development area is not structured in the same way that you want your package structured.

  You should use the *path1=path2* pathname format in your **`prototype`** file.

- You have relocatable objects in your package.

  You can use the *path1=path2* pathname format in your **`prototype`** file, with *path1* as a relocatable name and *path2* a full pathname to that object on your machine.

  You can use the **`search`** command in your **`prototype`** file to tell **`pkgmk`** where to look for objects. (You can not use the **`-c`** option with **`search`**, however.)

  You can use the **`-b`** *basedir* option to define a pathname that informs **`pkgmk`** where to find relocatable object names while creating the package. It does this by prepending `basedir` to relocatable object names while creating the package. For example, executing

  pkgmk **`-d`** /dev/diskette **`-b`** usr2/myhome/reloc

  would look in the directory **`/usr2/myhome/reloc`** for any relocatable object in your package.

- You have variable object names.

  You can use the **`search`** command in your **`prototype`** file to tell **`pkgmk`** where to look for objects. (You can not use the **`-c`** option with **`search`**, however.)

  You can use the *param="value"* command in your **`prototype`** file to give **`pkgmk`** a value to use for the object name variables as it creates your package.

  You can use the *variable=value* option on the **`pkgmk`** command line to define a temporary value for variable names.

- The root directory on your machine differs from the root directory described in the **`prototype`** file (and that will be used on the installation machine).

You can use the *-r rootpath* option to tell **`pkgmk`** to ignore the destination pathnames in the **`prototype`** file. Instead, **`pkgmk`** prepends *rootpath* to the source pathnames in order to find objects on your machine.

Step 13. Creating a Package with pkgtrans

`pkgtrans` performs the following package translations:

- a fixed directory structure to a datastream
- a datastream to a fixed directory structure

To perform one of these translations, execute

`pkgtrans` **`-s`** *device1 device2* [*pkg1*[,*pkg2*[. . .]]]

where **`-s`** is the option to translate to datastream, *device1* is the name of the device or directory where the package currently resides, *device2* is the name of the device onto which the translated package will be placed, and [*pkg1*[*pkg2* . . .]] is one or more package names. If no package names are given, a menu of all packages residing in *device1* will be displayed and the user asked for a selection.

Creating a Datastream Package

Creating a datastream package requires two steps:

1. Create a package using **`pkgmk`**.

   Use the default device (the installation spool directory) or name a directory into which the package should be placed. **`pkgmk`** creates a package in a fixed directory format. Specify the capacity of the device where the datastream will be placed as an argument to the **`-l`** option.

2. After the software is formatted in fixed directory format and is residing in a spool directory, execute **`pkgtrans`**.

   This command translates the fixed directory format to the datastream format and places the datastream on the specified medium.

For example, the two steps shown below will create a datastream package.

1. **`pkgmk -d`** spooldir **`-l`** 1400

   (Formats a package into a fixed directory structure and places it in a directory named **`spooldir`**. Each part of the package will require no more than 1400 blocks.)

2. **`pkgtrans -s`** spooldir 9track package1

   (Translates the fixed directory format of `package1` residing in the directory **`spooldir`** into a datastream format. Places the datastream package on the medium in a device named `9track`.)

OR

3. **`pkgtrans -s`** spooldir diskette package1

   Similar to number 2 above, except that it places the datastream package on the medium in a device named `diskette`. **`pkgtrans`** will prompt for additional volumes if the package requires more than one diskette.

Translating a Package Instance

When an instance of the package being translated to fixed directory format already exists on *device2,* **`pkgtrans`** will not perform the translation. You can use the **`-o`** option to tell **`pkgtrans`** to overwrite any existing instances on the destination device and the **`-n`** option to tell it to create a new instance if one already exists. Note that the above does not apply when *device2* contains a datastream format.

Set Packaging

Sets provide a method of grouping packages together as one installable entity. Usually this is used to group packages that provide a particular feature or set of features. To enable the set capability in UNIX System V Release 4.2, several packaging commands have been enhanced. They are

- **`pkgadd(1M)`**
- **`pkginfo(1M)`**
- **`pkgrm(1M)`**

Set Installation

For sets, a special-purpose package referred to as a Set Installation Package (SIP) is used. The SIP is used to control the installation of a set's member packages. The SIP's name and package instance name are always the same as those used to identify the set itself. For instance, the SIP controlling the installation of the Foundation Set (**`fnd`**) is also named Foundation Set (**`fnd`**). A SIP is distinguished from other packages by the `CATEGORY` parameter “set” in its **`pkginfo`** file and by the presence of a special type of package information file named **`setinfo(4)`**. This file is used to convey information about a set's member packages to the software installation tools.

When **`pkgadd`** recognizes that a SIP is being processed, it sets up special environment variables and makes them available to the SIP's procedure scripts. This allows for a well-defined interface between the scripts and **`pkgadd`** that enables the SIP scripts to do most of the work when processing set member package selection and interaction. The SIP's request and preinstall scripts are especially designed to use this environment.

Among other things, the SIP's request script utilizes these environment variables to access the **`setinfo`** file and access the set member packages' request and default response files (if any). After the request script has finished processing, the SIP's preinstall script is then used to pass back to **pkgadd** a list of set member packages selected for installation as part of the set (see Case 7 in the “Package Installation Case Studies” section of this guide for examples of these scripts).

The following is a list of the environment variables made available to a SIP's procedure scripts.

- `$SETINFO`
  This environment variable is used to access the **setinfo** file.
- `$REQDIR`
  This environment variable provides the directory where the set member packages' request and default response files, if any, reside.
- `$RESPDIR`
  This environment variable contains the name of the directory where processed response files are to be placed. This response file could be the result of having run a set member package's request script (in the case of custom installation) or simply a copy of the default response file provided with the SIP (in the case of automatic installation).
- `$SETLIST`
  This environment variable is used to pass back to **pkgadd** the list of packages selected for installation as part of the set.

After it has processed a SIP, **pkgadd** adds the set member packages selected (it gets this from the file represented by `$SETLIST` in the installation environment) to the list of packages to be installed and proceeds to install them.

Set Removal

When the package instance specified to the **pkgrm** command is a SIP, **pkgrm** will remove all of the SIP's set member packages in reverse dependency order (opposite the order in which they were installed). After all of its member packages have been removed, the SIP itself is removed from the system.

Set Information Display

The **pkginfo** command displays information about sets. Since the name of the set is the same as its SIP, **pkginfo** must distinguish between when it is being requested to provide information on the SIP, from a request asking for information on the set's member packages (not including the SIP).

If **-c** set is specified, **pkginfo** will display information about the SIP, if one was specified on the command line, or on all SIPs if none was specified. If the category **set** is not specified, **pkginfo** will display information about all packages except those whose category is **set.** If the name of a set is specified on the command line, but **-c** set is not, **pkginfo** will display information on all set member packages belonging to that set except for the SIP itself.

The setsize File

The **`setsize`** file is a set information file that defines disk space requirements for the target environment. It contains information about all of the packages in the set. This file describes the disk space taken up by installed files as well as extra space needed for dynamically created files, as described in each package's **`space`** file.

The generic format of a line in this file is:

pkg:pathname blocks inodes

Definitions for each field are as follows:

- *pkg*

  The short, or abbreviated, name of a package in the set. This name describes which package of the set requires the amount of space described by the rest of the data on this line in the **`setsize`** file.

- *pathname*

  Names a directory in which there are objects that will be installed or that will require additional space. The name may be the mount-point for a file system. Names that do not begin with a slash (/) indicate relocatable directories.

- *blocks*

  Defines the number of 512-byte disk blocks required for installation of the files and directory entries contained in the pathname. (Do not include file-system-dependent disk usage).

- *inodes*

  Defines the number of inodes required for the installation of the files and directory entries contained in the pathname.

At installation time, the set installation calls **`setsizecvt`**, which reduces the **`setsize`** file for a set to a **`space`** file containing entries for only the packages that are selected. It is this resulting **`space`** file against which space checking for the set is performed.

The setsizecvt Command

The **`setsizecvt`** command generates files in the **`space`** format for sets. Before sets were included as packaging objects, the installation tools used **`space`** files to specify any additional space the packages required, in addition to that listed in the entries in that package's **`pkgmap`** file.

The **`setsizecvt`** command was designed to work as simply as possible; the packaging tools process the sets in much the same way they process packages.

Executing in a set's installation directory, **`setsizecvt`** collects the **`space`** file (if it exists) and the **`setsize`** file from each of the packages included in that set. The **`setsize`** file is a file whose entries are formatted as follows:

pkg:/path/name   #blks   #inodes

where *pkg* is the short form of the package name, and the rest is the directory and number of blocks and inodes used in that directory. This **`setsize`** file is created when the sets are created. Typically, the **`setsize`** file for a given set would be created from the **`pkgmap`** files for all of the packages in that set.

`setsizecvt` selects those entries in the **`setsize`** file for packages (in the current set) that the user wants to install. Those entries are then collected in a new file called **`space`**.

`pkgadd` uses the **`space`** file to see if there is enough space on the disk to install the set. The **`space`** file for a set is treated the same way as it is in a package.

Quick Reference to Packaging Procedures

Before beginning any packaging procedure, you must first have planned your packaging needs based on the information presented in this chapter. The section “Basic Steps of Packaging” gives a comprehensive list of possible packaging steps and considerations. This section only covers the required steps.

1. Create a **`prototype`** file.

   - Create one manually using any editor. There must be one entry for every package component. The format for a **`prototype`** file entry is:

     [volno] ftype class pathname [major minor] [mode owner group] [mac fixed inherited]

     volno designates the medium volume number on which the object should be placed. If no *volno* is given, **`pkgmk`** distributes package components across volumes automatically.

     ftype must be one of these object file types:

```
f (standard executable or data file)
e (file to be edited upon installation or removal)
v (volatile file, contents will change)
d (directory)
x (exclusive directory)
l (linked file)
p (named pipe)
c (character special device)
b (block special device)
i (installation script or package information file)
s (symbolic link)
```

class defines the class to which the object belongs. Place an object into the class of `none` if no special handling is required.

pathname defines the pathname of an object. It can be in one of these formats:

- fixed pathname: */src/myfile*
- collectively relocatable pathname: *src/myfile* (no beginning slash)
- individually relocatable pathname: *$BIN/myfile*

This pathname defines where the component should reside on the installation medium and also tells **pkgmk** where to find it on your machine. If these names differ, use the *path1=path2* format for *pathname*, where *path1* is the name it should have on the installation machine and *path2* is the name it has on your machine.

major minor defines the major and minor numbers for a block or character special device.

mode owner group defines the mode, owner and group for the object. If not defined, the value of the **default** command is used. If no default value is defined, `644 root other` is assigned.

mac fixed inherited defines the Mandatory Access Control (MAC) level, fixed privilege(s), and inheritable privilege(s) and for the object. If not defined, the value of the **default** command is used. If no default value is defined, the equivalent of specifying `4 NULL NULL` is assigned (that is, a MAC level of USER_PUBLIC, with no fixed or inheritable privileges). MAC levels are installed only on **sfs**-type file-systems.

You can use four types of command lines in a **prototype** file:

search *pathnames* (defines a search path for **pkgmk** to use when creating your package)

include *filename* (nests **prototype** files)

default *mode owner group mac fixed inherited* (defines a default `mode owner group mac fixed inherited` for objects defined in this **prototype** file)

default *mode owner group* (defines a default `mode owner group` for objects defined in this **prototype** file)

param=value (defines parameter values for **pkgmk**)

All command lines must begin with an exclamation point (`!`).

- Create one using **pkgproto**.

  pkgproto [**-i**] [**-c** *class*] [*path1*[=*path2*] . . .] > *filename* where **`-i`** tells **`pkgproto`** to record symbolic links with an **`ftype`** of `f` (not `s`), **`-c`** defines the class of all objects as *class*, and *path1* defines the object pathname (or names) to be included in the **`prototype`** file. If *path1* is a directory, entries for all objects in that directory will be generated.

Use the *path1=path2* format to give an object a different pathname in the **`prototype`** file than it has on your machine. *path1* is the pathname where objects can be located on your machine and *path2* is the pathname that should be substituted for those objects.

`pkgproto` writes its output to the standard output. To create a file, you should redirect the output to a file. That file can be named **`prototype`** (although it is not required).

2. Create a **`pkginfo`** file.

   Use any editor. Define one entry per line per parameter in this format:

   PARAM=*"value"*

   where *PARAM* is the name of one of the standard installation parameters defined in the **`pkginfo`** manual page and *value* is the value you assign to it.

   You can also define values for your own installation parameters using the same format. Names for parameters that you create must begin with a capital letter and be followed by only lower-case letters.

   The following five parameters are required in every **`pkginfo`** file: `PKG`, `NAME`, `ARCH`, `VERSION` and `CATEGORY`. No other restrictions apply concerning which parameters or how many parameters you define.

   The `CLASSES` parameter dictates which classes are installed and the order of installation. Although the parameter is not required, no classes will be installed without it. Even if you have no class action scripts, the `none` class must be defined in the `CLASSES` parameter before objects belonging to that class will be installed.

3. Execute **`pkgmk`**.

   `pkgmk [`**`-d`** *device*`] [`**`-r`** *rootpath*`] [`**`-b`** *basedir*`] [`**`-f`** *filename*`]`

   where **`-d`** specifies that the package should be copied onto *device*, **`-r`** requests that the root directory *rootpath* be used to locate objects on your machine, **`-b`** requests that *basedir* be prepended to relocatable paths when searching for them on your machine, and **`-f`** names a file, *filename*, to be used as your **`prototype`** file. (Other options are described in the **`pkgmk`** manual page.)

Refer to the procedures in this chapter for details on other, optional packaging steps (including how to use **`pkgtrans`** to create a package in datastream structure).

Package Installation Case Studies

This section presents packaging case study in order to show packaging techniques such as installing objects conditionally, determining at run time how many files to create, and how to modify an existing data file during package installation and removal.

Each case begins with a description of the study, followed by a list of the packaging techniques it uses and a narrative description of the approach taken when using those techniques. After this material, sample files and scripts associated with the case study are shown.

Case #1

This package has three types of objects. The installer may choose which of the three types to install and where to locate the objects on the installation machine.

Techniques

This case study shows examples of the following techniques:

- using variables in object pathnames
- using the request script to solicit input from the installer
- setting conditional values for an installation parameter

Approach

To set up selective installation, you must:

- Define a class for each type of object which can be installed.

  In this case study, the three object types are the package executables, the manual pages, and the emacs executables. Each type has its own class: bin, man, and emacs, respectively. Notice in the **`prototype`** file, shown in Figure 14-13, that all of the object files belong to one of these three classes.

- Initialize the `CLASSES` parameter in the **`pkginfo`** file as null.

  Normally when you define a class, you want the `CLASSES` parameter to list all classes that will be installed. Otherwise, no objects in that class will be installed. For this example, the parameter is initially set to null. `CLASSES` will be given values by the request script, based on the package pieces chosen by the installer. This way, `CLASSES` is set to only those object types that the installer wants installed. Figure 14-12 shows the **`pkginfo`** file associated with this package. Notice that the `CLASSES` parameter is set to null.

- Define object pathnames in the **`prototype`** file with variables.

  These variables will be set by the request script to the value which the installer provides. **`pkgadd`** resolves these variables at installation time and so knows where to install the package.

  The three variables used in this example are:

  - `$NCMPBIN` (defines location for object executables)
  - `$NCMPMAN` (defines location for manual pages)
  - `$EMACS` (defines location for emacs executables)

  Look at the example **`prototype`** file (Figure 14-13) to see how to define the object pathnames with variables.

- Create a request script to ask the installer which parts of the package should be installed and where they should be placed.

  The request script for this package, shown in Figure 14-14, asks two questions:

  - Should this part of the package be installed?

    When the answer is yes, then the appropriate class name is added to the `CLASSES` parameter. For example, when the question “Should the manual pages associated with this package be installed” is answered yes, the class `man` is added to the `CLASSES` parameter.

  - If so, where should that part of the package be placed?

    The appropriate variable is given the value of the response to this question. In the manual page example, the variable `$NCMPMAN` is set to this value.

  These two questions are repeated for each of the three object types.

  At the end of the request script, the parameters are made available to the installation environment for **`pkgadd`** and any other packaging scripts. In the case of this example, no other scripts are provided.

  When looking at the request script for this example, notice that the questions are generated by the data validation tools **`ckyorn`** and **`ckpath`**.

Sample Files

```
PKG='ncmp'
NAME='NCMP Utilities'
CATEGORY='applications,tools'
ARCH='nh6800'
VERSION='Release 1.0, Issue 1.0'
CLASSES=”
```

Figure 14-12. Case #1 `pkginfo` File

```
i pkginfo
i request
x bin $NCMPBIN 0755 root other
f bin $NCMPBIN/dired=/usr/ncmp/bin/dired 0755 root other
f bin $NCMPBIN/less=/usr/ncmp/bin/less 0755 root other
f bin $NCMPBIN/ttype=/usr/ncmp/bin/ttype 0755 root other
f emacs $NCMPBIN/emacs=/usr/ncmp/bin/emacs 0755 root other
x emacs $EMACS 0755 root other
f emacs $EMACS/ansii=/usr/ncmp/lib/emacs/macros/ansii 0644 root other
f emacs $EMACS/box=/usr/ncmp/lib/emacs/macros/box 0644 root other
f emacs $EMACS/crypt=/usr/ncmp/lib/emacs/macros/crypt 0644 root other
f emacs $EMACS/draw=/usr/ncmp/lib/emacs/macros/draw 0644 root other
f emacs $EMACS/mail=/usr/ncmp/lib/emacs/macros/mail 0644 root other
f emacs $NCMPMAN/man1/emacs.1=/usr/ncmp/man/man1/emacs.1 0644 root other
d man $NCMPMAN 0755 root other
d man $NCMPMAN/man1 0755 root other
f man $NCMPMAN/man1/dired.1=/usr/ncmp/man/man1/dired.1 0644 root other
f man $NCMPMAN/man1/ttype.1=/usr/ncmp/man/man1/ttype.1 0644 root other
f man $NCMPMAN/man1/less.1=/usr/ncmp/man/man1/less.1 0644 inixmr other
```

Figure 14-13. Case #1 `prototype` File

```
trap 'exit 3' 15

# determine if and where general executables should be placed
ans=`ckyorn -d y \
        -p “Should executables included in this package be installed”
` || exit $?
if [ “$ans” = y ]
then
        CLASSES=“$CLASSES bin”
        NCMPBIN=`ckpath -d /usr/ncmp/bin -aoy \
                -p “Where should executables be installed”
        ` || exit $?
fi

# determine if emacs editor should be installed, and if it should
# where should the associated macros be placed
ans=`ckyorn -d y \
        -p “Should emacs editor included in this package be installed”
` || exit $?
if [ “$ans” = y ]
then
        CLASSES=“$CLASSES emacs”
        EMACS=`ckpath -d /usr/ncmp/lib/emacs -aoy \
                -p “Where should emacs macros be installed”
        ` || exit $?
fi

# determine if and where manual pages should be installed
ans=`ckyorn \
        -d y \
        -p “Should manual pages associated with this package be installed”
` || exit $?
if [ “$ans” = y ]
then
        CLASSES=“$CLASSES man”
        NCMPMAN=`ckpath -d /usr/ncmp/man -aoy \
                -p “Where should manual pages be installed”
        ` || exit $?
fi

# make parameters available to installation service,
# and so to any other packaging scripts
cat >$1 <<!
CLASSES='$CLASSES'
NCMPBIN='$NCMPBIN'
EMACS='$EMACS'
NCMPMAN='$NCMPMAN'
!

exit 0
```

Figure 14-14. Case Study #1 Request Script

Case #2

This package installs a driver. A set of device nodes associated with that driver needs to be created, but the installer will decide how many nodes to create. After installation, the system needs to be rebooted so that the driver is properly configured.

Techniques

This case study shows examples of the following techniques:

- installing a driver with a postinstall script
- using an exit code to reboot the system
- allowing the installer to define how many device nodes to create at installation time

Approach

To install a driver at the time of installation, you must:

- Include the object and master files for the driver in the **`prototype`** file.

  In this example, the object file for the driver is a data file named **`qz.o`**. This is the file on which the standard UNIX system driver install command, **`drvinstall`**, operates. The **`master.d`** file is named **`qz`** and is used by **`drvinstall`** to help configure the driver.

  Looking at Figure 14-15 (the **`prototype`** file for this example), notice the following:

  - Since no special treatment is required for these files, you can put them into the standard `none` class. The `CLASSES` parameter is set to `none` in the **`pkginfo`** file (Figure 14-16).
  - The pathname for **`qz.o`** begins with the variable `$BOOTDIR`. This variable will be set in the request script and allows the administrator to decide where the object file should be installed. The default directory will be **`/boot`**.
  - There is an entry for the postinstall script (the script that will perform the driver installation).

- Create a request script.

  The request script, shown in Figure 14-17, has two major functions:

  - to determine how many device nodes to create for this driver

    This is accomplished by questioning the installer and then assigning the answer to the parameter `$NDEVICES`. Notice that the data validation tool **`ckrange`** is used and that it limits the response to a number between 0 and 32. It sets the default number to 8.

    If the installer chooses not to install any devices, the `CLASSES` parameter is set to null. This means that no classes are defined and therefore no objects will be installed.

  - to determine where the installer wants the driver objects to be installed

This is accomplished by questioning the installer and assigning the answer to the `$BOOTDIR` parameter.

The script ends with a routine to make the three parameters `CLASSES`, `NDEVICES`, and `BOOTDIR` available to the installation environment and so to the postinstall script.

- Create a postinstall script.

  The postinstall script, shown in Figure 14-18, actually performs the driver installation. It is executed after the two files **`qz`** and **`qz.o`** have been installed. The postinstall shown for this example performs the following actions:

  - checks to see if any devices should be installed (if not, it exits)
  - creates the **`/dev/qz`** directory using the **`installf`** command (this directory could also be created by putting an entry for it in the **`prototype`** file)
  - executes the **`drvinstall`** command using the two files installed with this package (the major number is returned to the script at this time)
  - calculates the minor numbers for installed devices
  - installs the device using **`installf`**
  - creates a link for the device also using **`installf`**
  - finalizes the installation using **`installf -f`**

- Reboot the system upon installation.

  This is accomplished by exiting from the postinstall script with an exit code of `10`, meaning that the system should be rebooted upon completing an error-free installation.

Sample Files

```
i pkginfo
i request
i postinstall
f none $BOOTDIR/qz.o 444 root root
f none /etc/master.d/qz 444 root root
```

Figure 14-15. Case #2 `prototype` File

```
PKG='qzdev'
NAME='qz Devices'
CATEGORY='system'
ARCH='nh6800'
VERSION='Software Issue #19'
CLASSES='none'
```

Figure 14-16. Case #2 `pkginfo` File

```
trap 'exit 3' 15

# determine if and where general executables should be placed
NDEVICES=`ckrange -l0 -u32 -d 8 \
        -p “How many qz devices do you want configured”
` || exit $?

# if user chose to install no devices, don't install anything
if [ $NDEVICES -eq 0 ]
then
        CLASSES=
else
        # determine where driver object should be placed; location
        # must be an absolute pathname which is an existing directory
        BOOTDIR=`ckpath -aoy -d /boot \
                -p “Where do you want driver object installed”
        ` || exit $?
fi

# make parameters available to installation service,
# and so to any other packaging scripts
cat >$1 <<!
CLASSES='$CLASSES'
NDEVICES='$NDEVICES'
BOOTDIR='$BOOTDIR'
!
exit 0
```

Figure 14-17. Case #2 Request Script

```
# PKGINST parameter provided by installation service
# NDEVICES parameter provided by 'request' script
# BOOTDIR parameter provided by 'request' script

[ $NDEVICES -eq 0 ] && exit 0

err_code=1  # an error is considered fatal

# need to create the /dev/qz directory
installf $PKGINST /dev/qz d 755 root sys ||
        exit $err_code

# install the driver object and determine major device number
majno=`/usr/sbin/drvinstall -m /etc/master.d/qz -d $BOOTDIR/qz.o -v1.0` ||
        exit $err_code

i=00
while [ $i -lt $NDEVICES ]
do
        for j in 0 1 2 3 4 5 6 7
        do
                # calculate minor number based on loop variables
                minno=`expr $i \\\* 8 + $j` || exit $err_code

                # install character device with appropriate major/minor
                # device numbers and correct permissions (installf will
                # do all of work here - you need only provide the info!)
                installf $PKGINST /dev/qz/$i$j c $majno $minno 644 root sys ||
                        exit $err_code

                # create a link from /dev/qz/xx to /dev/qzxx
                installf $PKGINST /dev/qz$i$j=/dev/qz/$i$j ||
                        exit $err_code
        done
        i=`expr $i + 1`

        # add leading zero if necessary
        [ $i -le 9 ] && i="0$i"
done

# finalize installation; the installf command will now
# attempt to create the links that was requested above
installf -f $PKGINST || exit $err_code

exit 10  # requests a reboot from user
```

Figure 14-18. Case #2 Postinstall Script

Case #3

This study creates a database file at the time of installation and saves a copy of the database when the package is removed.

Techniques

This case study shows examples of the following techniques:

- using classes and class action scripts to perform special actions on different sets of objects

- using the **`space`** file to inform **`pkgadd`** that extra space will be required to install this package properly
- using the **`installf`** command

Approach

To create a database file at the time of installation and save a copy on removal, you must:

- Create three classes.

  This package requires three classes:

  - the standard class of `none` (contains a set of processes belonging in the subdirectory **`bin`**)
  - the **`admin`** class (contains an executable file **`config`** and a directory containing data files)
  - the **`cfgdata`** class (contains a directory)

- Make the package collectively relocatable.

  Notice in the **`prototype`** file (Figure 14-20) that none of the pathnames begin with a slash or a variable. This indicates that they are collectively relocatable.

- Calculate the amount of space the database file will require and create a **`space`** file to deliver with the package. This file notifies **`pkgadd`** that this package requires extra space and how much extra space. Figure 14-21 shows the **`space`** file for this package.

- Create an installation class action script for the **`admin`** class.

  The script, shown in Figure 14-22, initializes a database using the data files belonging to the **`admin`** class. To perform this task, it:

  - copies the source data file to its proper destination
  - creates an empty file named **`config.data`** and assigns it to a class of **`cfgdata`**
  - executes the **`bin/config`** command (delivered with the package and already installed) to populate the database file **`config.data`** using the data files belonging to the **`admin`** class
  - executes **`installf -f`** to finalize installation

  No special action is required for the **`admin`** class at removal time so no removal class action script is created. This means that all files and directories in the **`admin`** class will simply be removed from the system.

- Create a removal class action script for the **`cfgdata`** class.

  The script, shown in Figure 14-23, makes a copy of the database file before it is deleted during package removal. No special action is required for this class at installation time, so no installation class action script is needed.

Remember that the input to a removal script is a list of pathnames to remove. Pathnames always appear in lexical order with the directories appearing first. This script captures directory names so that they can be acted upon later and copies any files to a directory named **/tmp**. When all of the pathnames have been processed, the script then goes back and removes all directories and files associated with the **cfgdata** class.

The outcome of this removal script is to copy **config.data** to **/tmp** and then remove the **config.data** file and the data directory.

Sample Files

```
PKG='krazy'
NAME='KrAzY Applications'
CATEGORY='applications'
ARCH='nh6800'
VERSION='Version 1'
CLASSES='none cfgdata admin'
```

Figure 14-19. Case #3 `pkginfo` File

```
i pkginfo
i i.admin
i r.cfgdata
d none bin 555 root sys
f none bin/process1 555 root other
f none bin/process2 555 root other
f none bin/process3 555 root other
f none bin/config 500 root sys
d admin cfg 555 root sys
f admin cfg/datafile1 444 root sys
f admin cfg/datafile2 444 root sys
f admin cfg/datafile3 444 root sys
f admin cfg/datafile4 444 root sys
d cfgdata data 555 root sys
```

Figure 14-20. Case #3 `prototype` File

```
# extra space required by config data which is
# dynamically loaded onto the system
data 500 1
```

Figure 14-21. Case #3 space File

```
# PKGINST parameter provided by installation service
# BASEDIR parameter provided by installation service

while read src dest
do
        # the installation service provides '/dev/null' as the
        # pathname for directories, pipes, special devices, etc
        # which it knows how to create
        [ “$src” = /dev/null ] && continue

        cp $src $dest || exit 2
done

# if this is the last time this script will
# be executed during the installation, do additional
# processing here
if [ “$1” = ENDOFCLASS ]
then
        # our config process will create a data file based on any changes
        # made by installing files in this class; make sure
        # the data file is in class 'cfgdata' so special rules can apply
        # to it during package removal
        installf -c cfgdata $PKGINST $BASEDIR/data/config.data f 444 root sys
||
                exit 2
        $BASEDIR/bin/config > $BASEDIR/data/config.data ||
                exit 2
        installf -f -c cfgdata $PKGINST ||
                exit 2
fi
exit 0
```

Figure 14-22. Case #3 Installation Class Action Script (`i.admin`)

```
# the product manager for this package has suggested that
# the configuration data is so valuable that it should be
# backed up to /tmp before it is removed!

while read path
do
        # pathnames appear in lexical order, thus directories
        # will appear first; you can't operate on directories
        # until done, so just keep track of names until
        # later
        if [ -d $path ]
        then
                dirlist=“$dirlist $path”
                continue
        fi
        mv $path /tmp || exit 2
done
if [ -n “$dirlist” ]
then
        rm -rf $dirlist || exit 2
fi
exit 0
```

Figure 14-23. Case #3 Removal Class Action Script (`r.cfgdata`)

Case #4

This package uses the optional packaging files to define package compatibilities and dependencies and to present a copyright message during installation.

Techniques

This case study shows examples of the following techniques:

- using the **`copyright`** file
- using the **`compver`** file
- using the **depend** file

Approach

To meet the requirements in the description, you must:

- Create a **`copyright`** file.

  A **`copyright`** file contains the ASCII text of a copyright message. The message shown in Figure 14-25 will be displayed on the screen during package installation (and also during package removal).

- Create a **`compver`** file.

  The **`pkginfo`** file shown in Figure 14-24 defines this package version as version 3.0. The **`compver`** file, shown in Figure 14-26, defines version 3.0 as being compatible with versions 2.3, 2.2, 2.1, 2.1.1, 2.1.3 and 1.7.

- Create a **depend** file.

  Files listed in a **depend** file must already be installed on the system when a package is installed. The example shown in Figure 14-27 has 11 packages which must already be on the system at installation time.

Sample Files

```
PKG='case4'
NAME='Case Study #4'
CATEGORY='application'
ARCH='nh6800'
VERSION='Version 3.0'
CLASSES='none'
```

Figure 14-24. Case #4 `pkginfo` File

```
Copyright (c) 1989 AT&T
All Rights Reserved.

THIS PACKAGE CONTAINS UNPUBLISHED PROPRIETARY SOURCE CODE OF AT&T.

The copyright notice above does not evidence any
actual or intended publication of such source code.
```

Figure 14-25. Case #4 `copyright` File

```
Version 2.3
Version 2.2
Version 2.1
Version 2.1.1
Version 2.1.3
Version 1.7
```

Figure 14-26. Case #4 `compver` File

```
P acu    Advanced C Utilities
         Issue 4 Version 1
P cc     C Programming Language
         Issue 4 Version 1
P dfm    Directory and File Management Utilities
P ed     Editing Utilities
P esg    Extended Software Generation Utilities
         Issue 4 Version 1
P graph  Graphics Utilities
P rfs    Remote File Sharing Utilities
         Issue 1 Version 1
P rx     Remote Execution Utilities
P sgs    Software Generation Utilities
         Issue 4 Version 1
P shell  Shell Programming Utilities
P sys    System Header Files
         Release 3.1
```

Figure 14-27. Case #4 depend File

Case #5a

This study modifies a file which exists on the installation machine during package installation. It uses one of three modification methods. The other two methods are shown in Case #5b and Case #5c. The file modified is **`/etc/inittab`**.

Techniques

This case study shows examples of the following techniques:

- using the **`sed`** class
- using a postinstall script

Approach

To modify **`/etc/inittab`** at the time of installation, you must:

- Add the **`sed`** class script to the **`prototype`** file.

  The name of a script must be the name of the file that will be edited. In this case, the file to be edited is **`/etc/inittab`** and so our **`sed`** script is named **`/etc/inittab`**. There are no requirements for the `mode owner group` of a sed script (represented in the sample **`prototype`** by question marks). The file type of the sed script must be `e` (indicating that it is editable). The **`prototype`** file for this case study is shown in Figure 14-29

  NOTE

  Since the pathname of the **`sed`** class action script is exactly the same as the file it is intended to edit, these two may not coexist in the same package.

- Set the `CLASSES` parameter to include **`sed`**.

  In the case of the example shown in Figure 14-28, **`sed`** is the only class being installed. However, it could be one of any number of classes.

- Create a **`sed`** class action script.

  You cannot deliver a copy of **`/etc/inittab`** that looks the way you need for it to, since **`/etc/inittab`** has already been installed and is a dynamic file. Because of this, you have no way of knowing how it will look at the time of package installation. Using a **`sed`** script allows us to modify the **`/etc/inittab`** file during package installation.

  As already mentioned, the name of a **`sed`** script should be the same as the name of the file it will edit. A **`sed`** script contains **`sed`** commands to remove and add information to the file. See Figure 14-30 for an example **`sed`** script.

- Create a postinstall script.

  You need to inform the system that **`/etc/inittab`** has been modified by executing **`init q`**. The only place you can perform that action in this example is in a postinstall script. Looking at the example postinstall script, shown in Figure 14-31, you will see that its only purpose is to execute **`init q`**.

This approach to editing **`/etc/inittab`** during installation has two drawbacks. First of all, you have to deliver a full script (the postinstall script) simply to perform **`init q`**. In addition to that, the package name at the end of each comment line is hard-coded. It would be nice if this value could be based on the package instance so that you could distinguish between the entries you add for each package.

Sample Files

```
PKG='case5a'
NAME='Case Study #5a'
CATEGORY='applications'
ARCH='nh6800'
VERSION='Version 1d05'
CLASSES='sed'
```

Figure 14-28. Case #5a `pkginfo` File

```
i pkginfo
i postinstall
e sed /etc/inittab=/home/mypkg/inittab.sed ? ? ?
```

Figure 14-29. Case #5a `prototype` File

```
!remove
# remove all entries from the table that are associated
# with this package, though not necessarily just
# with this package instance
/^[^:]*:[^:]*:[^:]*:[^#]*#ROBOT$/d

!install
# remove any previous entry added to the table
# for this particular change
/^[^:]*:[^:]*:[^:]*:[^#]*#ROBOT$/d

# add the needed entry at the end of the table;
# sed(1) does not properly interpret the '$a'
# construct if you previously deleted the last
# line, so the command
#        $a\
#        rb:023456:wait:/usr/robot/bin/setup #ROBOT
# will not work here if the file already contained
# the modification.  Instead, you will settle for
# inserting the entry before the last line!
$i\
rb:023456:wait:/usr/robot/bin/setup #ROBOT
```

Figure 14-30. Case #5a sed Script (`/home/mypkg/inittab.sed`)

```
# make init re-read inittab
/sbin/init q ||
        exit 2
exit 0
```

Figure 14-31. Case #5a Postinstall Script

Case #5b

This study modifies a file which exists on the installation during package installation. It uses one of three modification methods. The other two methods are shown in Case #5a and Case #5c. The file modified is **`/etc/inittab`**.

Techniques

This case study shows examples of the following techniques:

- creating classes
- using installation and removal class action scripts

Approach

To modify **`/etc/inittab`** during installation, you must:

- Create a class.

  Create a class called **`inittab`**. You must provide an installation and a removal class action script for this class. Define the **`inittabl`** class in the `CLASSES` parameter in the **`pkginfo`** file (as shown in Figure 14-32).

- Create an **`inittab`** file.

  This file contains the information for the entry that you will add to **`etc/inittab`**. Notice in the **`prototype`** file (Figure 14-33) that **`inittab`** is a member of the **`inittab`** class and has a file type of `e` for editable. Figure 14-36 shows what **`inittab`** looks like.

- Create an installation class action script.

  Since class action scripts must be multiply executable (meaning you get the same results each time they are executed), you can't just add our text to the end of the file. The script, shown in Figure 14-34, performs the following procedures:

  - checks to see if this entry has been added before
  - if it has, removes any previous versions of the entry
  - edits the **`inittab`** file and adds the comment lines so you know where the entry is from
  - moves the temporary file back into **`/etc/inittab`**
  - executes **`init q`** when it receives the end-of-class indicator

  Note that **`init q`** can be performed by this installation script. A one-line postinstall script is not needed by this approach.

- Create a removal class action script.

  The removal script, shown in Figure 14-35, is very similar to the installation script. The information added by the installation script is removed and **`init q`** is executed.

This case study resolves the drawbacks to Case #5a. You can support multiple package instances since the comment at the end of the **`inittab`** entry is now based on package instance. Also, you no longer need a one-line postinstall script. However, this case has a drawback of its own. You must deliver two class action scripts and the **`inittab`** file to add one line to a file. Case #5c shows a more streamlined approach to editing **`/etc/inittab`** during installation.

Sample Files

```
PKG='case5b'
NAME='Case Study #5b'
CATEGORY='applications'
ARCH='nh6800'
VERSION='Version 1d05'
CLASSES='inittab'
```

Figure 14-32. Case #5b `pkginfo` File

```
i pkginfo
i i.inittab
i r.inittab
e inittab /etc/inittab ? ? ?
```

Figure 14-33. Case #5b `prototype` File

```
# PKGINST parameter provided by installation service

while read src dest
do
        # remove all entries from the table that are
        # associated with this PKGINST
        sed -e “/^[^:]*:[^:]*:[^:]*:[^#]*#$PKGINST$/d” $dest > /tmp/$$itab ||
                exit 2

        sed -e “s/$/#$PKGINST” $src >> /tmp/$$itab ||
                exit 2

        mv /tmp/$$itab $dest ||
                exit 2
done
if [ “$1” = ENDOFCLASS ]
then
        /sbin/init q ||
                exit 2
fi
exit 0
```

Figure 14-34. Case #5b Installation Class Action Script (`i.inittab`)

```
# PKGINST parameter provided by installation service

while read src dest
do
        # remove all entries from the table that
        # are associated with this PKGINST
        sed -e “/^[^:]*:[^:]*:[^:]*:[^#]*#$PKGINST$/d” $dest > /tmp/$$itab ||
                exit 2

        mv /tmp/$$itab $dest ||
                exit 2
done
/sbin/init q ||
        exit 2
exit 0
```

Figure 14-35. Case #5b Removal Class Action Script (`r.inittab`)

```
rb:023456:wait:/usr/robot/bin/setup
```

Figure 14-36. Case #5b `inittab` File

Case #5c

This study modifies a file which exists on the installation machine during package installation. It uses one of three modification methods. The other two methods are shown in Case #5a and Case #5b. The file modified is **`/etc/inittab`**.

Techniques

This case study shows examples of the following technique:

- using the **`build`** class

Approach

This approach to modifying **`/etc/inittab`** uses the **`build`** class. A **`build`** class file is executed as a shell script and its output becomes the new version of the file for which it is named. In other words, the file **`inittab`** that is delivered with this package will be executed and the output of that execution will become **`/etc/inittab`**.

The **`build`** class file is executed during package installation and package removal. The argument **`install`** is passed to the file if it is being executed at installation time. Notice in the sample **`build`** file in that installation actions are defined by testing for this argument.

To edit **`/etc/inittab`** using the **`build`** class, you must:

- Define the build file in the **`prototype`** file.

  The **`prototype`** file entry for the build class file should be of class **`build`** and file type `e`. Be certain that the `CLASSES` parameter in the **`pkginfo`** file includes **`build`**. Figure 14-37 shows the **`pkginfo`** file for this example and Figure 14-38 shows the **`prototype`** file.

- Create the **`build`** file.

  The **`build`** file shown in Figure 14-39 performs the following procedures:

  - Edits **`/etc/inittab`** to remove any changes already existing for this package. Notice that the filename **`/etc/inittab`** is hard-coded into the **`sed`** command.
  - If the package is being installed, adds the new line to the end of **`/etc/inittab`**. A comment tag is included in this new entry to remind us from where that entry came.
  - Executes **`init q`**.

This solution addresses the drawbacks in case studies Case #5a and Case #5b. Only one file is needed (beyond the **`pkginfo`** and **`prototype`** files), that file is short and simple, it works with multiple instances of a package since the `$PKGINST` parameter is used, and no postinstall script is required since **`init q`** can be executed from the **`build`** file.

Sample Files

```
PKG='case5c'
NAME='Case Study #5c'
CATEGORY='applications'
ARCH='nh6800'
VERSION='Version 1d05'
CLASSES='build'
```

Figure 14-37. Case #5c `pkginfo` File

```
i pkginfo
e build /etc/inittab=/home/case5c/inittab.build ? ? ?
```

Figure 14-38. Case #5c `prototype` File

```
# PKGINST parameter provided by installation service

# remove all entries from the existing table that
# are associated with this PKGINST
sed -e "/^[^:]*:[^:]*:[^:]*:[^#]*#$PKGINST$/d" /etc/inittab ||
        exit 2

if [ "$1" = install ]
then
        # add the following entry to the table
        echo "rb:023456:wait:/usr/robot/bin/setup #$PKGINST" ||
                exit 2
fi
/sbin/init q ||
        exit 2
exit 0
```

Figure 14-39. Case #5c `build` Script (`/home/case5c/inittab.build`)

Case #6

This case study modifies a number of **`crontab`** files during package installation.

Techniques

This case study shows examples of the following techniques:

- using classes and class action scripts
- using the **`crontab`** command within a class action script

Approach

You could use the **`build`** class and follow the approach shown for editing **`/etc/inittab`** in Case #5c study except that you want to edit more than one file. If you used the **`build`** class approach, you would need to deliver one for each **`cron`** file edited. Defining a **`cron`** class provides a more general approach. Figure 14-40 shows the **`pkginfo`** file for this example and Figure 14-41 shows the **`prototype`** file. To edit a **`crontab`** file with this approach, you must:

- Define the **`cron`** files that will be edited in the **`prototype`** file.

  Create an entry in the **`prototype`** file for each **`crontab`** file which will be edited. Define their class as **`cron`** and their file type as `e`. Use the actual name of the file to be edited, as shown in Figure 14-41.

- Create the **`crontab`** files that will be delivered with the package.

  These files contain the information you want added to the existing

crontab files of the same name. See Figure 14-44 and Figure 14-45 for examples of what these files look like.

- Create an installation class action script for the **cron** class.

  The **i.cron** script (Figure 14-42) performs the following procedures:

  - Calculates the user id.This is done by setting the variable *user* to the base name of the **cron** class file being processed. That name equates to the user id. For example, the basename of **/var/spool/cron/crontabs/root** is root (which is also the user id).
  - Executes **crontab** using the user id and the **-l** option. Using the **-l** options tells **crontab** to send the standard output the contents of the **crontab** for the defined user.
  - Pipes the output of the **crontab** command to a **sed** script that removes any previous entries that have been added using this installation technique.
  - Puts the edited output into a temporary file.
  - Adds the data file for the root user id (that was delivered with the package) to the temporary file and adds a tag so that you will know from where these entries came.
  - Executes **crontab** with the same user id and give it the temporary file as input.

- Create a removal class action script for the **cron** class.

  The removal script, shown in Figure 14-43, is the same as the installation script except that there is no procedure to add information to the **crontab** file.

These procedures are performed for every file in the **cron** class.

Sample Files

```
PKG='case6'
NAME='Case Study #6'
CATEGORY='application'
ARCH='nh6800'
VERSION='Version 1.0'
CLASSES='cron'
```

Figure 14-40. Case #6 pkginfo File

```
i pkginfo
i i.cron
i r.cron
e cron /var/spool/cron/crontabs/root ? ? ?
e cron /var/spool/cron/crontabs/sys ? ? ?
```

Figure 14-41. Case #6 `prototype` File

```
# PKGINST parameter provided by installation service

while read src dest
do
        user=`basename $dest` ||
                exit 2

        (crontab -l $user |
        sed -e "/#$PKGINST$/d" > /tmp/$$crontab) ||
                exit 2

        sed -e "s/$/#$PKGINST/" $src >> /tmp/$$crontab ||
                exit 2

        crontab $user < /tmp/$$crontab ||
                exit 2
        rm -f /tmp/$$crontab
done
exit 0
```

Figure 14-42. Case #6 Installation Class Action Script (`i.cron`)

```
# PKGINST parameter provided by installation service

while read path
do
        user=`basename $path` ||
                exit 2

        (crontab -l $user |
        sed -e "/#$PKGINST$/d" > /tmp/$$crontab) ||
                exit 2

        crontab $user < /tmp/$$crontab ||
                exit 2
        rm -f /tmp/$$crontab
done
exit 0
```

Figure 14-43. Case #6 Removal Class Action Script (`r.cron`)

```
41,1,21 * * * * /usr/lib/uucp/uudemon.hour > /dev/null
45 23 * * * ulimit 5000; /usr/bin/su uucp -c “/usr/lib/uucp/uudemon.cleanup” >
/dev/null 2>&1
11,31,51 * * * * /usr/lib/uucp/uudemon.poll > /dev/null
```

Figure 14-44. Case #6 Root `crontab` File (Delivered with Package)

```
0 * * * 0-6 /usr/lib/sa/sa1
20,40 8-17 * * 1-5 /usr/lib/sa/sa1
5 18 * * 1-5 /usr/lib/sa/sa2 -s 8:00 -e 18:01 -i 1200 -A
```

Figure 14-45. Case #6 Sys `crontab` File (Delivered with Package)

Case #7a

This case study shows an example of creating a Set Installation Package (SIP) that is used to control the installation of a set of packages.

Techniques

This case study shows examples of the following:

- creating a **`setinfo`** file (Figure 14-46)
- creating a `request` script that processes set member packages selection and type of installation (custom and default, if applicable) (Figure 8-49)
- including the set member packages' `request` and default `response` files in the **`prototype`** file as file type **`i`** files, if any, as part the SIP so that all interaction with the installer is done only during SIP processing (Figure 14-47)
- using the **`preinstall`** script to pass the selected packages back to **pkgadd** (Figure 14-48)

Approach

- Create a request script to ask the installer how the set should be installed (Figure 14-49).
- Should default installation be performed on this set?

When the answer is yes, if any of the set's member packages require interaction and if default responses for that interaction have been provided, install the set using the default responses.

When the answer is no, for each package in the set, prompt as to whether this package should be installed.

When the answer is yes, if it is interactive (the package has a request script), should default installation of the package be performed?

If yes, use the default **`response`** file.

If no, execute the package's `request` script to obtain the responses.

- When the request script has completed, the `PKGLIST` variable should contain the list of selected set member packages that will be installed on the system. At this time
  - **`pkgadd`** runs the SIP's `preinstall` script which places the selected set member packages for installation (`$PKGLIST`) into the **`setlist`** file referenced via the `$SETLIST` variable.
  - **`pkgadd`** then reads the **`setlist`** file and inserts the packages listed there into the list of packages to be installed on the system.
  - As each of these packages is processed, if the package is interactive, **`pkgadd`** will use the response file created earlier so that no prompting for user input occurs except during SIP processing.

Sample Files

```
# Format for the setinfo file.  Field separator is: <tab>
# pkg partsdefaultcategorypackage_name
# abbr   y/n

pkgw1   y   system   Package W
pkgx1   y   system   Package X
pkgy2   n   system   Package Y
pkgz1   y   system   Package Y
```

Figure 14-46. Case #7a `setinfo` File

```
# set packaging files
i pkginfo
i preinstall
i request
i setinfo
i copyright

i pkgw/request=pkgw.request
i pkgw/response=pkgw.response
i pkgx/request=pkgx.request
i pkgx/response=pkgx.response
```

Figure 14-47. Case #7a `prototype` File

```
for PKG in $PKGLIST
do
    echo “$PKG” >>$SETLIST
done
```

Figure 14-48. Case #7a `preinstall` Script File

```
# If <DELETE> is pressed, make sure we exit 77 so pkgadd knows
# no packages were selected for installation.  In this case,
# pkgadd will also not install the SIP itself.
trap 'EXITCODE=77; exit' 2
trap 'exit $EXITCODE' 0

while read pkginst parts default category package_name
do
    echo $pkginst >>/tmp/order$$
    if [ "$default" = "y" ]
    then
        echo $pkginst >>/tmp/req$$
    else
        echo $pkginst >>/tmp/opt$$
    fi
done <$SETINFO

REQUIRED=`cat /tmp/req$$ 2>/dev/null`
OPTIONAL=`cat /tmp/opt$$ 2>/dev/null`
ORDER=`cat /tmp/order$$ 2>/dev/null`
rm -f /tmp/opt$$ /tmp/req$$ /tmp/order$$

HELPMSG="Enter 'y' to run default set installation or enter
'n' to run custom set installation."

PROMPT="Do you want to run default set installation?"

ANS=`ckyorn -d y -p "$PROMPT" -h "$HELPMSG"`|| exit $?

if [ "$ANS" = "y" ]
then
    # Default installation
    for PKG in $REQUIRED
    do
        PKGLIST="$PKGLIST $PKG"
        if [ -f $REQDIR/$PKG/response ]
        then
            cp $REQDIR/$PKG/response $RESPDIR/$PKG
        fi
    done
    echo "PKGLIST=$PKGLIST" >> $1
else
    # Custom installation of required packages
    for PKG in $REQUIRED
    do
        PKGLIST="$PKGLIST $PKG"
        if [ -f $REQDIR/$PKG/request ]
        then
            PROMPT="Do you want default installation for $PKG?"
            RANS=`ckyorn -d y -p "$PROMPT" -h "$HELPMSG"` || exit $?
            if [ "$RANS" = "y" ]
            then
```

Figure 14-49. Case #7a `request` Script File

```
                 cp $REQDIR/$PKG/request $RESPDIR/$PKG
             else
                 sh $REQDIR/$PKG/request $RESPDIR/$PKG
             fi
         fi
     done

     # Select which optional packages in set are to be installed
     for PKG in $OPTIONAL
     do
     HELPMSG="Enter 'y' to install $PKG as part of this set
     installation or 'n' to skip installation."
     PROMPT="Do you want to install $PKG?"
     PANS=`ckyorn -d y -p "$PROMPT" -h "$HELPMSG"` || exit $?

     if [ "$PANS" = "y" -o "$PANS" = "" ]
     then
         PKGLIST="$PKGLIST $PKG"
         if [ -f $REQDIR/$PKG/request ]
         then
            PROMPT="Do you want default installation for $PKG?"
            RANS=`ckyorn -d y -p "$PROMPT" -h "$HELPMSG"` || exit $?
            if [ "$RANS" = "y" ]
            then
                cp $REQDIR/$PKG/request $RESPDIR/$PKG
            else
                sh $REQDIR/$PKG/request $RESPDIR/$PKG
            fi
         fi
     fi
     done
     echo "PKGLIST=$PKGLIST" >> $1
fi

if [ "$PKGLIST" = "" ]
then
     EXITCODE=77
fi
export SETPKGS
```

Figure 14-49. Case #7a `request` Script File (Cont.)

Case #7b

This study shows an example of how to split one set into two new sets (see the following illustrations: Figure 14-50, Figure 14-51, Figure 14-52, Figure 14-53, Figure 14-54, and Figure 14-55).

Techniques

This case study shows examples of the following:

- breaking up a **`setinfo`** file into two
- splitting the set member packages' **`request`** and default **`response`** files in the original SIP **`prototype`** file into two

Approach

From the SIP's **`setinfo`** file

- create two separate **`setinfo`** files for the two new sets being created
- create two separate **`prototype`** files for the two new sets being created

Sample Files

```
# Format for the setinfo file.  Field separator is: <tab>
# pkg partsdefaultcategorypackage_name
# abbr   y/n

pkgw 1    y    system   Package W
pkgx 1    y    system   Package X
pkgy 2    n    system   Package Y
pkgz 1    y    system   Package Y
```

Figure 14-50. Case #7b Original `setinfo` File

```
# Format for the setinfo file.  Field separator is: <tab>
# pkg partsdefaultcategorypackage_name
# abbr   y/n

pkgw 1    y    system   Package W
pkgz 1    y    system   Package Y
```

Figure 14-51. Case #7b SIP One New `setinfo` File

```
# Format for the setinfo file.  Field separator is: <tab>
# pkg partsdefaultcategorypackage_name
# abbr   y/n

pkgx 1    y    system   Package W
pkgz 1    n    system   Package Y
```

Figure 14-52. Case #7b SIP Two New `setinfo` Files

```
# set packaging files
i pkginfo
i preinstall
i request
i setinfo
i copyright

i pkgw/request=pkgw.request
i pkgx/response=pkgx.response
i pkgy/request=pkgy.request
i pkgz/response=pkgz.response
```

Figure 14-53. Case #7b Original `prototype` File

```
# set packaging files
i pkginfo
i preinstall
i request
i setinfo
i copyright

i pkgw/request=pkgw.request
i pkgz/response=pkgz.response
```

Figure 14-54. Case #7b SIP One New `prototype` File

```
# set packaging files
i pkginfo
i preinstall
i request
i setinfo
i copyright

i pkgx/request=pkgx.request
i pkgy/response=pkgy.response
```

Figure 14-55. Case #7b SIP Two New `prototype` Files

A
Guidelines for Writing Trusted Software

Writing Trusted Software

As a programmer you need to be aware of the special care you need to exercise when designing and writing software for any system. You want to ensure that the software you write and install for local applications is trusted.

The concept of trusting software is applicable to any system, regardless of the level of security implemented; the process of trusting software will lead to a more secure installation.

CAUTION

PowerMAX OS is a secure system, evaluated and rated by the National Computer Security Center.

Programmers will likely have to write software for individual installations. When this software is added to the rated system, however, the system in effect loses its rating.

Despite this, it is possible to maintain the security of the system if the locally written software is itself secure and trusted. This chapter provides an overview that explains how to write trusted software.

If a locally written piece of trusted software that has privileges was added to the system and is found to not deserve that trust, the system must be brought down and rebooted, with all files reloaded from copies saved prior to running that software.

Trust is the belief that a system element upholds the security policy of an operating system. If this belief is founded on blind faith, disasters are likely to happen, so it makes sense to assign trust only when a system element has been shown to deserve that trust.

The Enhanced Security Utilities available with the OS is designed to be compliant with a B1 and B2 level of security as defined by the Department of Defense and the National Computer Security Center. The software that constitutes the rated system strictly follows the security policy, which controls what software can do by assigning authorization and by limiting privileges. When you add software to the released system, you need to make sure that these local customizations and contributions have only the authorization and privileges they need to do their jobs.

For user-level software, this means making sure that a command or library routine works as advertised, and prevents unauthorized users from circumventing access controls or mechanisms that protect sensitive system operations. In this section, trust refers not to blind faith, but to confirmed trustworthiness.

Trust is key to the Enhanced Security Utilities available with the OS and to adding software to it.

The Enhanced Security Utilities available with the OS are designed to protect sensitive information from unauthorized access. Software that runs on a system that contains sensitive information must be trusted to maintain this protection. The rest of this chapter discusses trust and how to include it in software added to the rated system.

The OS without the Enhanced Security Utilities installed is secure, but it is not compliant with the B2 level of security.

Scope of Trust

The first step in assigning trust to a command or library routine is to determine whether it has enough access to the system to require trust. Some commands do not require privilege or access to sensitive information. Such commands need not be trusted, since they pose no threat.

Other commands either occasionally or routinely obtain access to sensitive operations, or create that access for themselves through mechanisms like the **`setuid-on-exec`** feature. These commands must be trusted, since they operate in a sensitive environment.

The rules dictating which commands need trust and which commands do not are straightforward, but matching a command to a rule may not be. The following command classes must always be trusted:

- commands used by administrative personnel
- commands invoked by other trusted commands
- commands that use privilege (see the *System Administration, Volume 1* for an explanation of privilege)
- commands that set their user or group identity to an administrative one on execution (**`set-id`**)

Deciding whether a command is “used by administrative personnel” or “uses privilege” can be difficult, since this distinction often varies from site to site and administrator to administrator. The Enhanced Security Utilities defines a Mandatory Access Control (MAC) isolation policy that alleviates this confusion, but the decision must nonetheless be made.

Library routines have similar rules, but these routines are so pervasive the most reasonable rule is: each library routine must be trusted unless it can be shown not to be used by trusted code. This principle means that every element of a trusted command must itself be trusted. This principle includes the private routines within the command as well as all library routines used by the command.

How Trust Is Achieved

The rules for trust are different for commands and library routines. These rules are described in detail in the remaining sections of this chapter.

Trust is achieved by following all rules that pertain to writing a given piece of software and by documenting the methods used to follow those rules. This documentation must be supplied with every piece of trusted software. It describes the circumstances under which it is trusted, the methods used to make it trusted, and warnings about any practices that might jeopardize the trust placed in the software.

As with all code that is to be incorporated in a running system, trusted software needs to be reviewed and tested before it is installed. You can have reviewers and testers read this chapter so that they can familiarize themselves with the special requirements for trusted software.

How to Use This Chapter

This chapter is divided into sections describing the procedures needed to produce and install trusted software. You may want to read the *System Administration, Volume 1* for background information.

It is a good idea to become familiar with the background material first, then proceed with reading the sections of this chapter that explain how to ensure trust in the kind of software you are writing. Reading the entire chapter is useful, but not essential. Many rules for ensuring trust are also good general programming practices, so they may also benefit any programming you do.

Finally, be aware that this chapter does not contain the definitive explanation of trust. Writing software is as much an art as it is a science, and the rules presented here are only guidelines to gain an understanding of the issues involved. It is by no means a guarantee that you will produce trusted software if you blindly obey the rules and dutifully mark the checklists. However, reading the advice here is a good beginning to learning how to write trusted software.

Trust and Security

Any discussion of software trust must be based on fundamental understanding of the security-related system elements. In the system security provided by the Standard Package with the OS, these elements are:

- Privileges
- Trusted Facility Management (TFM)

- Discretionary Access Controls (DAC)
- DAC Isolation Mechanism

In the Enhanced Security Utilities package, these elements are:

- Mandatory Access Controls (MAC)
- MAC Isolation Mechanism

The next subsections give a general explanation of these elements of security and trust. There are other descriptions in the *System Administration, Volume 1* to which you may want to refer for other perspectives and information.

Privilege

Privilege means “the ability to override system restrictions.” This ability is vested in three ways:

- in any user whose effective identity is `root`
- by way of the Trusted Facility Management (TFM) feature
- through fixed privileges assigned to executable files

There is a problem with the first approach to overriding system restrictions. A user (or command) allowed a reasonably mundane privileged action (for example, reading a protected file without explicit permission) also has permission to perform every other privileged action on the system, including the permission to overwrite all files on the system, add users, kill processes, start and stop network services, mount and unmount file systems, and many other sensitive operations. There is no restriction because there is no way to give a “little bit of `root`” to a user or command. Any process with an effective user-ID of “0” (`root`) is considered omnipotent.

The second and third approaches provide methods of giving a “little bit of `root`” to a user or command, and thus address the problem with the first approach. These approaches can be thought of as “Least Privilege” since they introduce the idea of discrete privileges that are associated with executable files and processes.

The basic security mechanism divides the ability to override system restrictions into discrete privileges, each overriding a specific action or class of actions. With the least privilege mechanism, a command can get the privileges it needs without getting the privileges it does not need. That is, the command gets the least amount of privilege required for its task.

The second and third approaches dissolve the bond between user identity and privilege, making privilege a process and executable file attribute instead of a user attribute. This approach makes sense because command behavior is much easier to describe and regulate than user behavior. Specifically, a process has privilege only when it is executing a privileged command.

Process privileges are contained in two sets, “working” and “maximum.” The working set contains the privileges in effect at any particular instant. This set controls the restrictions that the process can override at the moment. The **procpriv(2)** system call allows a command to set or clear privileges in the calling process's working set.

The maximum set represents the upper limit of privileges that a process can have in its working set. These privileges have no effect unless they are also in the working set, but they are held in reserve for the command to assert at any time. Using the **procpriv** system call, a command can clear a privilege in the maximum set but cannot set one.

The privilege set associated with a command's executable file determine what is put in the working and maximum privilege sets when a process executes the command. The two possible file privilege sets are fixed and inheritable. Fixed privileges are useful for commands that do privileged things for ordinary users because they are granted unconditionally upon execution. The unconditional nature of fixed privileges, however, means that any program that uses them must strictly enforce all system policies it can override.

A privilege in the inheritable privilege set of a command is granted only if the invoking process had that privilege in its maximum set before the **exec** system call [see **exec(2)**]. If a privilege is not in either the inheritable or fixed privilege sets of a command, the command cannot get that privilege even if the process had the privilege before the **exec** system call. The inheritable privilege set for executable files is not supported when the SUM policy module is used. The SUM module considers all of the calling process's maximum privilege set to be inheritable.

This privilege propagation mechanism provides strong protection against Trojan Horse attacks by preventing commands from obtaining privileges they do not need or do not use correctly.

Trusted Facility Management

The privilege inheritance mechanism described in the "Privilege" section ensures that only commands that need privilege and that use it correctly ever get it.

One type of command that can never be shown to use privilege correctly is an interactive shell, because an interactive shell depends on the user to define its behavior. This causes a problem because most privileged commands have inheritable privileges, and inheritable privileges need a process from which they can be inherited. The solution to this problem is the Trusted Facility Management (TFM) mechanism. TFM provides an interface between users (not privileged) and commands (privileged). The primary elements of TFM are the **tfadmin(1M)** command and the TFM database.

The **tfadmin** command is invoked with the desired command line as its arguments as in the following example:

```
tfadmin mount /dev/mydsk /my_mnt_point
```

The fixed privilege set of the **tfadmin** command file contains all privileges, so the **exec** system call turns on all privileges in the resulting process.

But the **tfadmin** command cannot be executed successfully by every user. To open it to such free access would be a violation of trust. When **tfadmin** is invoked, the first thing it does is to find out the real identity (real UID) of the invoking user. It then uses that identity to find the user's entry in the TFM Database.

The TFM database contains three types of information:

- the list of privileged commands assigned to each user
- the list of roles to which each user is assigned
- the list of privileged commands that define specific roles

A trusted system may define administrative roles for selected system administrators. Each role is likely to be filled by a different administrator in order that all sensitive administrative functions not be handled by a single person. This division of administrative duties into separate roles reduces the chances for misuse of administrative power. All trusted administrators will be associated with at least one role and/or set of privileged commands; a very few administrators may be associated with more than one role, especially at small sites. But most users are not associated with any role.

When **`tfadmin`** finds the user's entry, it looks for the requested command in the list of specific commands, and if it does not find it, in the list of roles. Once the command is found and the user's entry verifies that the user is assigned to a role that has the authorization to use that command, **`tfadmin`** turns on the correct privileges (found in the database entry for the command) in its maximum set and executes the command. If the executable file has any fixed privileges associated with it (via the **`filepriv(1M)`** command) they will be added to the privileges obtained from the TFM database by the **`exec(2)`** system call. All privileges obtained from the TFM database and the executable file’s fixed privileges are propagated across the chain of execution of any child processes.

In order for a shell script to propagate privileges whether they are acquired by way of **`tfadmin`** or **`filepriv`**, the script file must begin with a line of the form:

```
#! pathname [arg]
```

where pathname is the path of the interpreter (usually a shell), and arg is an optional argument.

By providing a single point of privileged access to administrative commands and by basing that access on the real identity of the requesting user, **`tfadmin`** eliminates the need for privileged ID's and enhances administrative accountability.

Mandatory Access Control

The Mandatory Access Control (MAC) mechanism associates a “level” with each process and data object (file, pipe, device, IPC, and so on.) under the Enhanced Security Utilities. A MAC level defines the sensitivity and topic of a piece of data, and prevents it from being disclosed or altered by an unauthorized user.

Unlike the traditional access control attributes (the user, group, other mode bits), the MAC level is set automatically by the system and cannot be changed by anyone except an administrator responsible for maintaining MAC.

As noted above, the MAC level represents both the sensitivity and the topic of a data object. The sensitivity is represented by a classification value. The higher this value, the more sensitive the information is. The topic is represented by a set of categories. If, for example, a file contains proprietary information about future development projects for a hypothetical company and its product, the file might have the level:

```
    PROPRIETARY:firefly,freedonia,sylvania
```

The classification (degree of sensitivity) of the file is `PROPRIETARY`, and the categories (topics) are `firefly` (contains development information on the firefly project), `freedonia` (contains information about marketing firefly in the country of Freedonia) and `sylvania` (contains information about marketing firefly in Sylvania.)

The MAC level of a process is both the level put on any data object created by the process and the level used to determine whether the process has access to any particular object.

Access is based on level dominance. A level (A) dominates another level (B) if the classification of level A is equal to or greater than that of level B and all categories in level B are contained in level A. Two levels are equal if their categories and classification are exactly the same.

A process is allowed to read a data object only if the level of the process dominates the level of the object. An object is allowed to write a data object only if the level of the process equals the level of the object.

MAC Isolation Policy

Most commands base some decisions on data from outside sources. Commands that enforce security policies based on external data need correct data to make sane decisions. Take, for example, the **`login`** procedure. This procedure obtains user attributes and password information from a set of data files. If one of these data files were replaced or masked somehow by an attacker, **`login`** could not make rational decisions about attempts to log into the system. In its confusion, it might allow an attacker to enter the system under false pretenses and steal information or damage data.

Commands also fall prey to this attack. A classic infiltration method is to replace a command with a new command that does something extra, like create a set-uid shell. When an unsuspecting user invokes the command, that user gets the normal command and the unfriendly act.

The first line of defense against an attack on sensitive system data and commands is to prevent untrusted individuals from changing them. This is good, but not good enough, since there are often ways to mask or replace a data object without actually changing it.

An example of this kind of attack is the well known trick of planting a command named **`ls`** in a directory and waiting for someone with the current directory first in `$PATH` to list that directory. The **`ls`** command has not changed at all, but it has been effectively replaced by a different command. The only defense against this attack is to restrict a sensitive command's or a user's ability to use potentially bogus information.

Finally, there are always data objects that contain information that must not be disclosed to the general public since they might give clues to attackers about ways to infiltrate the system. An example of this kind of information is the system password list. Even if the names and passwords in this list are encrypted, disclosing this information exposes the system to password guessing.

In general, any information used solely to administer a system should be hidden from non-administrators, since they have no need to use the information, and attackers are clever and persistent. Information used both by administrators and users should be seen by both groups but should be changed only by administrators. Information used only by users should not be visible to administrators (to prevent administrators from using bad data or commands).

The access lattice defined by the set of MAC levels can be subdivided into three sections representing the three classes of information described above. Purely administrative information falls under the `SYSTEM:PRIVATE` section and has a level prohibiting reads or writes by non-administrators. Information shared by administrators and users falls under the `SYSTEM:PUBLIC` section and has a level prohibiting writes by non-administrators, but allowing reads by anyone. Pure user information falls under the `USER` section, and has a level prohibiting reads or writes by administrators.

Discretionary Access Control

Discretionary Access Control (DAC) on a file defines the permissible access to it by its owner, the owner's group, and all others. It is discretionary because the protection on this data object is set at the discretion of the owner of the object.

When the Enhanced Security Utilities are installed, DAC also includes Access Control Lists (ACLs).

Discretionary Access Isolation

Even though the access to sensitive system information is well regulated by the MAC access isolation mechanism, discretionary controls provide flexibility and more granular protection within that framework. Furthermore, since MAC exists only when the Enhanced Security Utilities are installed, and sensitive files are not limited to those systems, “a” DAC isolation mechanism is needed to protect files on base systems.

A review of the limitations and pitfalls of discretionary protection is in order. First, the discretion to change permissions on data resides with the owner. If ownership of a piece of data is obtained by a malicious or incompetent user, nothing can prevent that user from destroying all discretionary protections. Second, discretionary access controls cannot be used to prevent sensitive software or users from reading bad data, because the owner of a file can always make its data readable by the world, and the world includes sensitive people. Finally, discretionary access is based on effective user and group identity. Effective identities change whenever a set-id-on-exec command runs, and they remain changed until the command sets them back to the real identities or exits. Thus, sensitive discretionary access (and ownership) can be passed from a trusted command to an untrusted one by accident, exposing the system to attack.

The OS protects sensitive data files by setting the ownership of all such files to `root` and supplying **`setuid-on-exec`** commands to give users controlled access to these files. This method provides protection because it makes protected files accessible only to the most restricted user.

This protection is adequate for most systems, but it is inadequate for protecting sensitive information on secure systems, because in practice, this has led to a proliferation of **`setuid-on-exec`** to **`root`** commands, some of which might be less careful than they should about propagating the `root` user identity to other commands. As a result, not only did the file protection begin to fail, but what had been the most restricted user identity suddenly became much easier to obtain.

The next attempt was to set up "ghost" user identities other than `root` to own sensitive files. Ghost user identities are user ID's in the system that are inaccessible as a valid user account (i.e. no one can login with this ID). Programmers using this technique managed to protect `root` somewhat better, but still left open the risk of Trojan Horse attacks on the files they were trying to protect. Finally, it became clear that giving away ownership to files made attacks too easy. Giving away group access was preferable. True, it was still possible to gain unauthorized access through imperfect system commands, but at least that access was limited to reading and writing.

The currently recommended DAC isolation method calls for the existence of a "ghost" owner: `sys`. This owner has a locked password entry, to make logging in as that user impossible. In addition, no commands can set their user identity to `sys` upon execution. This makes it impossible for a non-privileged process to obtain this user identity. Groups are defined to provide protection isolated according to the kinds of commands and users needing access to protected files. Administrators are assigned multiple group lists that allow direct access to protected files while normal users may gain access only through set-gid commands. All files protected by this mechanism are owned by `sys` and have the appropriate system group identity.

Writing Trusted Commands

The following sections describe how to write trusted commands.

User Documentation

The first line of defense against system damage is accurate and complete documentation. Before a command can be trusted, its use, behavior, options, and influence over the system must be fully described. In addition to a full description of the command, any potentially harmful behavior should be noted, to allow users to avoid such hazards.

Parameter and Process Attribute Checking

The parameters given to a command at execution are the primary external influences over the behavior of the command. All parameters passed into a command at execution, therefore, must be checked and shown to be consistent by the command before processing starts. This means that a command that has, for example, two mutually exclusive modes of operation based on command line options must ensure that only one of these modes is requested at a time. This is particularly important when one operation might negate the other or cause an inconsistency in the system, or when the interfaces for two operations are similar enough to interact in a way that might be misinterpreted by the command.

Process attributes are also important, but, with rare exception, should not be checked explicitly by a command. The reason for this is that most process attributes are intended to be checked by the operating system itself and will cause identifiable errors if they are not right. It is unwise to make assumptions about the way a particular operating system decision will come out based on potentially flawed knowledge of how the decision is made. Some exceptions to this rule are the process `umask`, which should be set as needed by all trusted commands, and the process `ulimit`, which, if too small, may lead a trusted command to an error from which it cannot gracefully recover.

Privilege and Special Access

There are two forms of special access in PowerMAX OS. The first is the access granted by the set-id feature, and the second is privilege. In the past these have been bound together through the `root` effective user identity, and they continue to be bound in superuser-based versions of PowerMAX OS.

In least privilege-based versions, however, a distinction has been drawn. Regulating the use of these special permissions is central to trust, so some rules are needed to show how that regulation should be done.

Set-id Commands

Commands that use the set-id feature to obtain access to files not otherwise available to an invoking user must carefully control not only their own use of these access permissions, but how these permissions are granted to other commands. There is always the possibility of a Trojan Horse when a command executes another command so care must be taken (see “Executing Other Commands”). In this section, the issue is incorrect use of special access rights. In general, the best protection against either incorrect use or a Trojan Horse is to reset the effective user and group identity immediately on entry to a command and only use the special identities where they are explicitly needed. The code excerpt in Figure A-1 illustrates the procedure.

Privileged Commands

Adding least privilege with the Enhanced Security Utilities installed changes the handling of privileged system calls. Since privilege is no longer guaranteed to depend on the effective user identity, a mechanism is now provided to regulate the specific use of privilege. The mechanism is available in both least privilege and superuser based systems and is reached through the **`procpriv(2)`** system call or the **`procprivl(3C)`** library routine. The rules for controlling process privilege are similar to those for controlling effective user and group identity. The first thing a trusted command must do is turn off all working privileges, to ensure that the command can not use or pass on a privilege unintentionally. When the command needs to use a privilege, it turns on that privilege alone, using **`procpriv`** or **`procprivl`**, and makes the sensitive system call or library routine. After the call, the command turns off all privileges. Notice that the period in which privileges are asserted is as short as possible. Given the choice between turning on privilege in the main routine before calling a routine that uses privilege, or turning on and off the same privilege within the lower level routine, the latter choice should be taken. There are exceptions, but exceptions should be made for solid technical reasons, not convenience. The code excerpt in Figure A-2 illustrates proper privilege use:

```
static  uid_teff_uid, real_uid;
static  uid_teff_gid, real_gid;
        .
        .
        .
main(argc, argv)
int     argc;
char    *argv[];
{
        /*Variable declarations*/
        eff_uid = geteuid();
        eff_gid = getegid();
        real_uid = getuid();
        real_gid = getgid();
        if(seteuid(real_uid) < 0){  /*Set the effective UID to the real*/
                error(“Cannot reset UID.”); /*Report error and exit*/
        }
        if(setegid(real_gid) < 0){  /*Set the effective GID to the real*/
                error(“Cannot reset GID.”); /*Report error and exit*/
        }
              .
              .
              .
        if(setegid(eff_gid) < 0){            /*Assert the effective GID*/
                error(“Cannot assert GID.”);/*Report error and exit*/
        }
        fd = open(“/etc/security_file”, O_RDWR);
        if(setegid(real_gid) < 0){  /*Set the effective GID to the real*/
                cleanup();                  /*Restore consistency*/
                error(“Cannot reset GID.”); /*Report error and exit*/
        }
        if(fd < 0){
                error(“Cannot open file.”); /*Report error and exit*/
        }
        /*Process data*/
              .
              .
              .
        close(fd);
}
```

Figure A-1. Correct Regulation of Access in C Programs

Privilege and Special Access in Shared Private Routines

A group of related commands occasionally share routines from a common object module. Such routines may provide database access, device setup and release, data conversion, etc. The desire to centralize these utility functions leads to creation of private “libraries.” Although these are not usually libraries in the archive sense, they are collections of useful routines stored in a place that makes them accessible to a controlled group of commands. Since these routines are private, they are treated as subsections of the commands that use them. These routines are designed to cooperate closely with their calling programs, so they are expected to regulate privilege internally.

```
#include <priv.h>
section of , argv)
int     argc;
char    *argv[];
{
        /*Variable declarations*/
        if(procprivl(CLRPRV,pm_work(P_ALLPRIVS),0) < 0){
                error("Cannot clear privileges."); /*Report error and exit*/
        }
              .
              .
              .
        if(procprivl(SETPRV,pm_work(P_DACREAD),pm_work(P_DACWRITE),0) < 0){
                error("Cannot set privileges."); /*Report error and exit*/
        }
        fd = open("/etc/security_file", O_RDWR);
        if(procprivl(CLRPRV,pm_work(P_ALLPRIVS),0)){
                cleanup();                         /*Restore consistency*/
                error("Cannot clear privileges."); /*Report error and exit*/
        }
        if(fd < 0){
         error("Cannot open file.");        /*Report error and exit*/
        }
        /*Process data*/
              .
              .
              .
        close(fd);
}
```

Figure A-2. Correct Use of Privilege in a C Program

Exceptions to this rule occur when different commands have different views of the same routine or when the designer of a routine believes the routine may be added to a public library. A private database library may contain a routine to open and position the database. A command that only needs to query the database might want to assert only read access override privileges while a command that changes the database might want to assert both read and write access override privileges. Such a routine should make no assumptions about what privileges the calling routine wants to use, but should simply assume that the correct privileges are in place.

A library routine might also have broad enough usefulness to be a candidate for public use. The reasons why such a routine might not be placed in a public library range from a desire to keep the published interface as small as possible to name conflicts or even lack of staff to make the change. If a programmer believes that a routine is useful enough to merit consideration for a public library, the programmer should follow the rules for writing public library routines, even if the routine is initially private.

These guidelines apply equally well to special access permissions obtained through the set-id mechanism as they do to privilege. Wherever these access permissions are used instead of privilege, they should be turned on and off as though they were individual privileges, using the **seteuid** and **setegid** system calls as shown in Figure A-1.

Error Checking

Almost every system call or library routine can, somehow, encounter an error during its operation. While many of these occur only because of programmer error, each such problem indicates a failure of either the system, the calling program or a transient parameter like access permission or available memory. If a programmer chooses to ignore a reported error, the result is a command that, should some basic assumption of the system fail, could corrupt its environment. For trusted commands, therefore, every possible error return must be checked and reported. This rule is not always followed to the letter, since in some cases it is more efficient to detect the error case downstream from the actual failure. Ignoring errors is risky and should not be done without strong justification.

Signal Handling

Signals pose a problem in trusted software because they are not predictable. There are two main areas of concern when it comes to handling signals:

1. maintaining system integrity when a trusted command receives a signal
2. use of privilege and special permission inside signal handling functions

If a signal is received by a trusted command, that command must not simply exit and leave the system in an inconsistent or insecure state. If a command contains critical sections that cannot be interrupted, every effort must be made to prevent signals from interrupting those sections.

On the other hand, a signal usually means either that a system problem has occurred (like memory exhaustion, an addressing error, or invalid operation) or that the user has decided to abort the operation. Regardless, it is not correct for a command to continue processing as though nothing had happened.

A system-generated signal usually signifies a flaw in the command and almost certainly means that further processing will be based on corrupt data. A user-generated signal signifies a change of heart by the requesting user and should be honored where possible by restoring the system to the state it was in before the command was invoked. If a command receives a signal after it is committed to a change, the command should finish any steps necessary to ensure consistency and exit.

Attempts to write signal-safe commands must take into account the possibility of unforeseen signals and signals that cannot be caught. On any given system, the set of possible signals is constant, but in general, systems are allowed to have their own implementation-specific signals.

It is better to keep the critical sections of a command as small as possible than to try to protect large critical sections against interruption. This principle means, for example, a command that changes a system database should make all changes on a copy of any sensitive part of the database (for example an index file) before replacing the original. This limits opportunity for an unknown signal to interrupt the sensitive part of the command.

When a trusted command is using privilege or some other extraordinary access and receives a signal, the command may enter a signal handler. Because signals are unpredictable, it is not a good idea for a command to change the privileges or other access attributes of its process inside a signal handler. When the handler returns to the main stream of processing, these attributes must be the same as they were before the signal occurred, or unpredictable processing will result.

Since signal handlers are not allowed to change process attributes, they should never do anything that might take advantage of privileges or special access. In general, a signal handler should set a flag and return or long jump away. Once the flag is set, the command can recognize the signal and respond to it in an orderly fashion.

Handling Sensitive Data

While it is important that trusted commands always protect the integrity of the data they manipulate, they must also prevent information disclosure that might damage system security. If commands are used exclusively by administrators or never gain access to sensitive information, then they are mostly exempt from this concern, but some commands are regularly used by non-administrators and use privilege or special access to read secret information.

An example is the **`passwd`** command. The **`passwd`** command retrieves information from the system password list (not normally readable by users) and reports (and sometimes changes) that information. In the process of obtaining the information, **`passwd`** must scan through records that are not intended for the eyes of the invoking user. If a signal were to cause **`passwd`** to write a core image with one or more records buffered, it would be possible for an enterprising programmer to extract secret information from the core image.

It is best to eliminate this possibility by designing databases and commands to handle only the sensitive information they are authorized to disclose. When it is impossible to eliminate the risk, programmers should limit the vulnerability of the command by clearing the contents of any sensitive buffers as soon as they cease to be needed.

Executing Other Commands

Whenever a command executes another command, it must first set its effective user and group identities to its real user and group identities unless the executed command needs the special access to do its job. If the executed command needs the special access, the executing command must take every possible step to ensure that it executes the correct command with proper parameters and cannot be misled into executing a Trojan Horse.

A Trojan Horse is a command that imposes itself on a process by looking like the needed command. It inherits permissions and other attributes (like file descriptors, environment, and so on), from the executing command, and can use these capabilities to disrupt the system. Measures to prevent Trojan Horse intrusion include the following:

- using full pathnames for execution
- avoiding the **`system`** and **`popen`** library routines, which use the shell to interpret command lines
- carefully making sure the `$PATH`, `$IFS`, and other environment variables are set to safe values whenever the shell must be used

- never allowing special-access rights or file descriptors to survive across an execution of a user-supplied command name

Using Library Routines

A trusted command must never use an untrusted library routine. This restriction means that a trusted command must never use a library routine that has an untrusted call anywhere in its calling sequence, nor a library routine that causes an untrusted command to be executed. The information derived from the untrusted command might influence the behavior of the trusted command, or the command might give away extraordinary access to the untrusted command; neither action is acceptable.

Trusting Shell Scripts

With the introduction of support for multiple file formats in the OS, it is possible to have set-id and privileged shell scripts. In addition, there have always been shell scripts that are used by administrators. If a shell script can get administrative access to the system it must be trusted, so rules for trusting shell scripts are needed as well.

The primary rule of trusted shell scripts is: any shell script that uses privilege or special access rights is subject to spoofing and must not be available to non-administrators.

This rule means that shell scripts that use privilege must be `SYSTEM:PRIVATE` under the Enhanced Security Utilities and must be executable only by an administrative group or user in the base system.

User Documentation

The documentation needed for a trusted shell script is the same as that for any other trusted command. See the “User Documentation” part of the “Writing Trusted Commands” section.

Privilege and Special Access

The shell offers no way to control special access rights granted by the set-id feature. Without this control, such a shell script must be extremely simple before it can be trusted. In general, it is not a good idea to use the set-id mechanisms for shell scripts. Only trusted commands should be used in shell scripts.

The “trusted” shell provided with the Enhanced Security Utilities has the ability to regulate privilege through the new built-in **`priv`** command.

```
#! /sbin/sh -p
priv -allprivs max         #Turn off all working privileges
if [ $? -ne 0 ]
then                        #The priv command will report the error
        exit $?
fi
        .
        .
        .
priv +mount max
if [ $? -ne 0 ]
then                        #The mount command will report the error
    exit $?
fi
/sbin/mount /dev/mydsk /mnt
priv -allprivs max
if [ $? -ne 0 ]
then                        #The priv command will report the error
        exit $?
fi
        .
        .
        .
```

Figure A-3. Correct Use of Privilege in a Shell Script

Executing Commands

Shell scripts consist mainly of commands, which makes them especially vulnerable to spoofing attacks. Only trusted commands should be used in shell scripts. Also, all commands that are not known to be built into the shell itself must be executed either by their full pathname or through the **`/sbin/tfadmin`** command provided by the TFM feature.

Sometimes, a script will need to use a command with privilege regardless of TFM data. When this situation occurs, privileges are assigned to the script by way of TFM. Fixed privileges are assigned by way of the **`filepriv`** command. In this case, the script should turn on only the needed privileges and execute the command using a full pathname (see Figure A-3).

Another way of executing a privileged command is through the **`/sbin/tfadmin`** command, since this allows the TFM mechanisms to decide whether the user of the script should have the privilege. In this instance, all commands to be executed in the script must exist in the TFM database, and all users who execute the script must have access to them. This case is illustrated in Figure A-4.

```
#! /sbin/sh
if [ $? -ne 0 ]
then                            #The command will report the error
        exit $?
fi
        .
        .
        .
if [ $? -ne 0 ]
then                            #The command will report the error
    exit $?
fi
tfadmin mount /dev/mydsk /mnt
if [ $? -ne 0 ]
then                            #The command will report the error
        exit $?
fi
        .
        .
        .
```

Figure A-4. Shell Script Using Commands From TFM Database

In order for a script to propagate privileges whether they are acquired by way of **`tfadmin`** or **`filepriv`**, the `#!` line must be the first line of the script.

Error Checking

Most commands report the errors they encounter and exit with a non-zero return code on failure. Shell scripts, therefore, usually do not need to bother reporting errors. Nonetheless, shell scripts should check for errors. A command that fails and reports an error indicates a problem in the shell script. If that error might cause the system to be left in an inconsistent state by the script, the error must be caught and handled. Whether the error is specially reported depends on the particular circumstances.

For example, if the failing command redirects its standard error output to a file or to **`/dev/null`**, the shell script must report an error to avoid failing silently.

If, on the other hand, the command does nothing to redirect messages, then the command's error message should be enough to tell the user what happened.

Trusting Public Library Routines

While commands obtain their privilege and special access through kernel mechanisms, library routines obtain their access rights and privileges from the commands that call them. Additionally, library routines usually serve a single purpose instead of offering a spectrum of options. These differences dictate the rules for library routines described below.

Documentation

The most important aspect of trusting a library routine is the documentation used by a programmer to decide how and when that routine should be used. This description should include basic elements such as the interface to the routine, what the routine does, and what error conditions might be encountered by the routine. Additionally, any privileged routine should have a description of the privileges it can use and the reason it might use each privilege. Also, any interesting side effects of the routine should be detailed. These include opening, closing, deleting or creating files, executing commands, setting global variables, allocating heap storage, changing process attributes, sending signals, or any other behavior that is not immediately obvious to the reader.

Finally, the description should include a section describing any non-trusted uses of the routine. If, for example, a user can cause the routine to fill past the end of a buffer by feeding it too much data, this possibility should be stated in the description. By supplying as much information as possible to the programmer who will use the routine, the documenter allows the programmer to choose routines wisely and use them correctly.

Privilege and Special Access

Public libraries provide many useful functions, such as file IO buffering, memory allocation, and mathematical processing. These routines are intended for use by a wide variety of applications, with a wide variety of needs and goals.

A library routine, therefore, should not try to guess the intent of the calling program. It should simply do its job and return. The rule for public library routines and privilege or special access is: no public routine should change the privilege or access environment of a process unless that is its primary purpose. There should be no exceptions to this rule, since a trusted command must always be in full control of its privileges and special access rights.

Reporting Errors

The only way a command can detect and recover from an error is to use the information reported by the system calls and library routines that encountered the error. A library routine, therefore, must report every possible error case as informatively as possible to the calling program. Where several different failure modes are possible, each should be reported uniquely so that the calling program can take any necessary corrective action or can restore system integrity before exiting. It is not correct for a library routine to cause a process to exit as the result of an error, since the calling program may need to clean up before exiting. The rule is: library routines must report all errors as accurately as possible.

Handling Sensitive Data

Library routines sometimes need to retrieve sensitive data for a trusted command. The designer of such routines must be aware of the risk that this data might be accidentally disclosed in a core file or some other unprotected data object. For a more detailed discussion of this problem and its solutions, see the "Handling Sensitive Data" section of "Writing Trusted Commands."

Executing Commands

Whenever a library routine executes a shell level command it must take great care to ensure that the command is executed correctly and with the right parameters. For library routines that handle requests to execute a command this requirement is limited to making sure the request is followed exactly as issued. Library routines (like **`system`** or **`popen`**) that execute commands independently of the specific request must use full pathnames, and be certain that the commands they execute are themselves trusted.

Installing Trusted Commands and Data

The access isolation and privilege mechanisms described in the *System Administration, Volume 1* depend on the software installation procedures. Defining special levels and group identities serves no purpose if those levels and groups are not used correctly. Defining a set of privileges and kernel level mechanisms to enforce and control them serves no purpose if every command gets all fixed privileges. As much care must be put into defining the installation parameters of a command and its data objects as goes into writing the command and designing its data. This section establishes principles upon which installation decisions can be made.

Assigning Access Controls

All trusted data must be protected from unauthorized changes. Working in the enhanced security environment, this protection is achieved using the MAC Isolation policy. Any trusted data object containing information that must be visible to the entire system (for example, **`/etc/passwd`**) gets the `SYSTEM:PUBLIC` level alias. Any other trusted data object gets the `SYSTEM:PRIVATE` level alias, since only administrators ever need to read this data. This decision is based on the question "does any non-administrator need to use this information?" not "is this information too sensitive for non-administrators to see?"

MAC isolation for trusted commands is the same as MAC isolation for data. Trusted commands used both by administrators and non-administrators (for example, **`cat, cup, me, an)`** are kept at `SYSTEM:PUBLIC` while commands with only administrative usefulness (for example, **`uadmin, useradd, usermod, userdel)`** are kept at `SYSTEM:PRIVATE`. By keeping administrative commands out of reach of non-administrators, this policy prevents users from accidentally or intentionally executing privileged system services incorrectly or in manner that violates the security policy. (See the section on *"Privilege"* in the "Trusted Facility Management" chapter of this guide.)

Discretionary access controls are used within the `SYSTEM:PUBLIC` and `SYSTEM:PRIVATE` levels to provide a finer access granularity. These permissions should be assigned based on logical groupings of data according to the needs of a set of commands and administrators. Since the discretionary controls are the only protections available to the base system, they should be assigned as though they were protecting a system on which all files are public and writable unless restricted by DAC.

The actual permissions placed on a given file depend entirely on the needs of the commands that use the file. The group bits, however, should be used instead of the owner bits to grant controlled access to files. This methodology allows the designer to use set-uid `root` for non-access related privilege and still take advantage of DAC controls on a least privilege system.

Assigning Privileges and Special Permissions

Privileges are assigned to executable files (commands) based on the needs of the command and the knowledge that the command will not misuse the privileges. These two factors are equally important: Even though a programmer knows that a command will not abuse a particular privilege, the command must need that privilege or it does not get it. Furthermore, even though a command needs a privilege, it must be shown to use the privilege properly or it does not get it.

After determining what privileges a command can have, the next step is to determine whether the command needs privileges that are propagated through **`tfadmin`**, or fixed privileges. Inheritable privileges are assigned to commands that are either intended to be run only by administrators or other trusted commands or need privilege only when used in these ways. Fixed privileges are assigned to commands that need privilege when used by any user and cannot obtain the needed access any other way.

Using fixed privileges calls for extremely careful programming. A command with fixed privilege must never use untrusted data for security-relevant decision making. This means that a shell script can never have fixed privilege, since the environment a shell script inherits is untrusted and influences the shell's behavior (a command that uses the **`system`** or **`popen`** library routines can never have fixed privilege for the same reason). Other possible disqualifications are the following:

- commands that are controlled by user-supplied script files
- commands that are controlled by data from standard input

Privileges acquired through **`tfadmin`** are more carefully controlled, so they do not require the extensive limitations placed on fixed privilege. Any privileged command, however, must uphold system policies when it uses privilege and must obey both the spirit and the letter of the rules of trust described in these guidelines.

Special access rights should be used in favor of privileges wherever possible. A program that needs discretionary access to a well-defined set of files should be **`setgid`** to the group to which those files belong. The files should be as accessible as necessary to their group. If, for example, a command needs to read a file **`foo`** and read and write a file **`bar`** and the group of the files **`foo`** and **`bar`** is **`sys`**, the command should be **`setgid`** to **`sys`**. The file **`foo`** should be readable by group while the file **`bar`** should be both readable and writable by group. The `P_DACREAD` and `P_DACWRITE` privileges should not be used for this purpose, since they give too much access to the command.

Summary

Trusting a command or library routine requires a solid understanding of the risks encountered by the command or library, the policies of the system, and the principles of trust. These guidelines offer a brief look at the policies of the Enhanced Security Utilities available with the OS, and a discussion of the principles of trust. The risks encountered by a particular command or library must be determined by the programmer attempting to make it trusted.

While some of the rules presented here may seem overly exacting, or even clumsy, the strenuousness of the rules is the price paid for a secure system. Every rule and principle described in these guidelines originates from some aspect of an observed attack on a computer system. The programmer who ignores these rules does so, not at his or her own risk, since the programmer is unlikely to be affected by the attack, but at the risk of everyone who uses that programmer's software. The responsibility of writing trusted software, therefore, must not be taken lightly.

Glossary

The following terms are used throughout the programming series. This glossary includes terms found in:

- *PowerMAX OS Programming Guide*
- *Compilation Systems Volume 1 (Tools)*
- *Compilation Systems Volume 2 (Concepts)*
- *Character User Interface Programming*
- *Concurrent C Reference Manual*

a.out

`a.out`, historically for "assembler output," is the default file name for an executable program produced by the C compilation system.

abortive release

An abrupt termination of a transport connection, which may result in the loss of data.

access permissions

Access checking is performed whenever a subject (a process) tries to access an object (such as a file or directory). Permission to access an object is granted or denied on the basis of "mode bits and Access Control Lists (ACLs); ACLs mode bits, Access Control Lists (ACLs), and Mandatory Access Control (MAC) levels; ACLs and MAC levels are supported only if the Enhanced Security Utilities are installed and running, and only if the file system on which the object to be accessed resides is of type `sfs`.

ADJUST

The mouse button or keyboard equivalent used to adjust a selection (cf. **SELECT**); usually the middle button on a right hand mouse.

alias file

A script which contains alias definitions, each on a separate line. An alias file is optional, but if one is written, it must be named as an argument when **`fmli`** is invoked.

alias

A short name that can be used in FMLI scripts in place of a long pathname or a list of paths to search. An FMLI developer defines aliases in an alias file. Alias definitions have the format *alias*=*pathname*.

alternate keystrokes

A sequence of keystrokes, usually beginning with a `CTRL` key and consisting entirely of keys that are standard on all keyboards, which cause the same action to occur that occurs when a named key is pressed. Alternate keystrokes are necessary because many keyboards do not have a complete set of the named keys used by FMLI applications. For example, when the named key ¦is not available on a keyboard, users can type the alternate keystrokes `CTRL-u`.

anchor

Either end of a `Scrollbar` widget or a `Slider` widget. The part of the widget that remains fixed while the *elevator* or *drag box* moves along.

ANSI

ANSI is an acronym for the American National Standards Institute. ANSI establishes standards in the computing industry from the definition of ASCII (see below) to the measurement of overall datacom system performance. ANSI standards have been established for the Ada, FORTRAN, and C programming languages.

API

Application programmer interface.

application

An executable program, usually unique to one type of users' work, such as an accounting application. Applications are frequently interactive environments in which the user can perform various related tasks. See **FMLI application**.

archive

An archive, or statically linked library, is a collection of object files each of which contains the code for a function or a group of related functions in the library. When you call a library function in your program, and specify a static linking option on the **cc** command line, a copy of the object file that contains the function is incorporated in your executable at link time. For further information, see the *Concurrent C Reference Manual.*

argument

A character string or number that follows a command and controls its execution in some way. There are two types of arguments: options and operands. Options change the execution or output of the command. Operands provide data that will be operated on by the command. Arguments to the **open** command are saved in built-in variables readable (only) by the frame opened. Options are also called flags. Operands specify files or directories to be operated on by the program. For example, in the command line:

```
$ cc -o hello hello.c
```

all the elements after the **cc** command are arguments. For further information of how command line arguments are passed to C programs, see the *Concurrent C Reference Manual*.

In the C language, function arguments are enclosed in a pair of parentheses immediately following the function name. You can find formal definitions of the functions supplied with the C compilation system in **`cc(1)`**.

ASCII

An acronym for American Standard Code for Information Interchange. ASCII code uses one byte of computer memory to represent each character. Each alphanumeric and special character has an ASCII equivalent. When files and directories are printed according to the ASCII code equivalent of the first letter of their names, the order is called ASCII collating sequence. The order is special characters first, numbers second, then upper case and lower case letters.

assembler

Assembly language is a programming language that uses symbolic names to represent the machine instructions of a given computer. An assembler is a program that accepts instructions written in the assembly language of the computer and translates them into a binary representation of the corresponding machine instructions. Because each assembly language instruction usually has a one-to-one correspondence with a machine instruction, programs written in assembly language are not portable to different machines.

asynchronous execution

The mode of execution in which Transport Interface routines will never block while waiting for specific asynchronous events to occur, but instead will return immediately if the event is not pending.

automatic data

Data that is persistent only during the invocation of a procedure. It describes data belonging to a process. Automatic data occupies the stack segment. See **static data**.

background process group

Any process group that is not the foreground process group of a session that has established a connection with a controlling terminal.

backquoted expression

A command line enclosed in backquotes, whose output is returned as a value. The output of the command replaces the backquotes and the command line within the backquotes. In FMLI, this output can be used as an argument for another command, assigned to a variable, or assigned to a descriptor.

banner line

The top line of the screen in FMLI applications, used to display the application's title and a `Working` message that indicates when the application is busy.

bottom level

Lowest of the four lower RPC levels; programs written to this level can control many transport-specific details.

buffer

A buffer is a space in computer memory where data is stored temporarily in convenient units for system operations. Buffers are often used by programs such as editors that access and alter text or data frequently. When you edit a file, for instance, a copy of its contents are read into a buffer; the copy is what you change. For your changes to become part of the permanent file, you must write the buffer's contents back into the permanent file. This replaces the contents of the file with the contents of the buffer. When you quit the editor, the contents of the buffer are flushed.

button

Generic term for any of several widgets, specifically `RectButton` widgets and `OblongButton` widgets. The RectButtons are implicitly defined in *flattened widgets*, as well. A button, when pressed usually initiates certain actions, like popping up a menu or executing an application routine.

cable

In a `Scrollbar` widget, the cable is the “line” on which the *elevator* moves. One end of the cable is connected to the *anchor* and the other is connected to the *elevator.*

callback

A callback routine is a routine written by an application programmer and associated with a specific widget *resource*. The callback routine is invoked as a result of a specific activity associated with that widget (that is, the widget calls back the program via that routine). For example, the `XtNselect` *resource* contains the name of the callback routine that is entered when a *button* is pushed or when a `CheckBox` is selected; the `XtNverification` resource contains the name of the callback routine to invoke when a `TextField` widget is exited. The act of associating the name of a callback routine with a widget resource is called *registration.*

cast

An expression which describes the nature or use of that which follows it to the interpreter. In FMLI, casts are used: (1) to describe whether a file is a menu definition file, a form definition file, or a text frame definition file; (2) to indicate how often to evaluate a descriptor.

character class table

A character class table is used for character classification and conversion. The table is built by the commands **`chrtbl(1M)`** and **`wchrtbl(1M)`**, and located in the file **`usr/lib/locale/LC_CTYPE`**.

child process

See **`fork`**.

choices menu

A menu that can be provided to show a list of possible entries to a form field. An FMLI application developer defines choices where appropriate through the use of the `rmenu` descriptor.

click

The act of pressing and releasing a mouse button without moving the mouse *pointer* more than a few pixels.

click-move-click

A method of user interaction with a set of objects where the user clicks MENU to display the objects, moves the pointer over the one of interest, then clicks MENU or SELECT to select or activate the object.

client

The transport user in connection-mode that requests a transport connection.

CLTS

Connectionless Transport Service

command line

The next-to-the-last line on the screen in FMLI applications, where users can enter an application's commands without using the menus provided in the application.

command menu

A menu provided automatically in FMLI applications that lists a sub-set of the FMLI built-in commands and any application-specific commands that have been defined in a commands file. Users can execute a command in the Command Menu by selecting it, as in any menu. The Command Menu can be made current by pressing the CMD-MENU function key.

command

one of a set of executables built into FMLI, such as **`open`** and **`close,`** to which descriptors of type command must evaluate. A command line consists of the command followed by its arguments. For example:

```
$ cc file1.c file2.c
```

instructs the operating system to execute the C compiler program, which is stored in the file **`cc`**, and to use the source files **`file1.c`** and **`file2.c`** as input. A command line can extend over multiple terminal lines.

commands file

A script in which an FMLI developer can redefine or disable FMLI built-in commands, and define new, application-specific commands. The contents of a commands file are reflected in the Command Menu. Users can execute a command by selecting it from the Command Menu, or by typing it on the FMLI command line. A commands file is optional, but if one is written, it must be named as an argument when **`fmli`** is invoked.

compiler

A compiler is a program that translates a source program written in a higher-level language into the assembly language of the computer the program is to run on. An assembler translates the assembly language code into the machine instructions of the computer. In the C compilation system, these instructions are stored in object files that correspond to each of your source files. Each object file contains a binary representation of the C language code in the corresponding source file. The link editor links these object files with each other, and with any library functions you have used in your source code, to produce an executable program called **`a.out`** by default. For further information, see the *Concurrent C Reference Manual*.

composite widget

See **widget**. A widget that is a parent of other widgets, that physically contains other widgets.

connection establishment

The phase in connection-mode that enables two transport users to create a transport connection between them.

connection-mode

A circuit-oriented mode of transfer in which data is passed from one user to another over an established connection in a reliable, sequenced manner.

connection-oriented transport

Connection-oriented transports are reliable and support byte-stream deliveries of unlimited data size.

connectionless transport

Connectionless transports have less overhead than connection-oriented transports but are less reliable and maximum data transmissions are limited by buffer sizes.

container

A widget that defines a region that holds zero or more sub-objects of a given type.

control area

The area located directly under the header of a *window*. It is used to display “command buttons,” if the application in the window provides them.

controlling process

A session leader that established a connection to a controlling terminal.

controlling terminal

A terminal that is associated with a session. Each session may have, at most, one controlling terminal associated with it and a controlling terminal may be associated with only one session. Certain input sequences from the controlling terminal cause signals to be sent to process groups in the session associated with the controlling terminal; see **`termio(7)`**.

conversation

The negotiation and the data transfer between *Source* and *Destination*. Both tasks are accomplished through *selection mechanism*.

core image

A core image is a copy of the memory image of a process. A file named **`core`** is created in your current directory when the UNIX operating system aborts an executing program. The file contains the core image of the process at the time of the failure. For further information, see the *Concurrent C Reference Manual.*

current

The frame, menu item, form field, or activity in which the cursor is positioned. An element of the FMLI screen which is current is usually distinguished in some way from other screen elements being displayed—the current frame, for example, may be shown in bright video, while non-current frames may be shown in half-bright video. User input is processed by, or applies to, the current frame, item, and so on.

daemon

A background process that performs a system-wide public function. The UNIX system process **`init`** may spawn daemon processes that exist throughout the lifetime of the system. Daemons (often) continue to run after their parents terminate. An example of a daemon process is **`calendar(1).`**

data symbol

A data symbol names a variable that may or may not be initialized. Normally, these variables reside in read/write memory during execution. Compare "**text symbol**."

data transfer

The phase in connection-mode or connectionless-mode that supports the transfer of data between two transport users.

datagram transport

See *connectionless transport.*

datagram

A unit of data transferred between two users of the connectionless-mode service.

debugging

Debugging is the process of locating and correcting errors in executable programs.

default

A default is the way a program will perform a task in the absence of other instructions, that is, in default of your specifying something else.

descriptor

An element of the Form and Menu Language that defines some aspect of the look (appearance or location of an element of your application), or feel (an action to take in response to user input). A descriptor is coded in the format *dname*=*value*, where *dname* is one of the set of Form and Menu Language descriptors and *value* is, or generates, an expression of a type appropriate for the particular descriptor. Each Form and Menu Language descriptor is only meaningful in a particular context (that is, a menu frame, a form frame, and so on).

deserializing

Converting data from XDR format to a machine-specific representation.

destination

The ending point of the drag-and-drop operation. It is also referred as the requester.

dimmed

A visual effect on an object. A control, such as a *button*, is dimmed if its visible manifestation represents the state of just one of several objects that are in inconsistent states. When such a control is manipulated (for example, by clicking SELECT over the button), it is no longer dimmed because the manipulation sets the state for all the objects.

directory

A directory is a type of file used to group and organize other files or directories. A subdirectory is a directory that is pointed to by a directory one level above it in the file system. A directory name is a string of characters that identifies the directory. It can be a simple directory name, a relative path name, or a full path name. For further information, see the *User's Guide.*

display width

Display width is the width in screen columns required to display the characters of a particular code set. Display width is defined in the character class table.

dominate

Domination is a relationship between Mandatory Access Control (MAC) levels. Level S1 is said to dominate security level S2 if the hierarchical classification number of S1 is greater than or equal to that of S2, and if the nonhierarchical categories of S1 are a superset of the categories included in S2. Conversely, level S2 can be said to be dominated by

S1. The **`lvlname`** command lists the levels, classifications, and categories currently defined on the system (see **`lvlname(1M)`**).

double click

To press and release a mouse button twice in succession.

downstream

In a stream, the direction from stream head to driver.

drag area

In a `Scrollbar` widget, the drag area is the center portion of the *elevator* that is moved by the mouse.

drag box

In a `Slider` widget, the drag box is the portion of the slider that is moved by the mouse.

drag-and-drop

A single atomic action to achieve a *Conversation* between *Source* and *Destination.*

dragging

The act of moving the *pointer* while a mouse button or keyboard equivalent is pressed.

driver

In a stream, the driver provides the interface between peripheral hardware and the stream. A driver can also be a pseudo-driver, such as a multiplexor or log driver (see **`log(7)`**), which is not associated with a hardware device.

DTM

Desktop manager.

dynamic frame

A frame whose contents are determined at run-time.

dynamic linking

Dynamic linking refers to the process in which external references in a program are linked with their definitions when the program is executed. For further information, see the *Concurrent C Reference Manual.*

effective group ID
effective user ID

An active process has an effective user ID and an effective group ID that are used to determine file access permissions. The effective user ID and effective group ID are equal to the process's real user ID and real group ID respectively, unless the process or one of its ancestors evolved from a file that had the set-user-ID bit or set-group ID bit set (see **`exec(2)`**).

elevator

The center portion of a `Scrollbar` widget; that part which moves along the *cable.*

ELF

ELF is an acronym for the executable and linking format of the object files produced by the C compilation system. For further information, see the *Concurrent C Reference Manual.*

environment

A set of UNIX system shell variables created and assigned values by the system when a user logs in. The system executes programs that set these variables based on information it gets from **`/etc/profile,`** the shell, **`login(1),`** and the user's **`.profile`** file. In FMLI, variables can be added to the environment with the **`set(1F)`** built-in utility, and removed from the environment with the **`unset(1F)`** utility. FMLI also defines a local environment that contains variables known only to the FMLI application.

ETSDU

Expedited Transport Service Data Unit.

EUC

Extended UNIX system code. See *Programming with UNIX System Calls*.

executable program

On the UNIX operating system, an executable program is a compiled and linked program or a shell program. The command to execute either is the name of the file containing the program. A compiled and linked program is called an executable object file. Compare "**object file**."

executable

A program that can be processed or executed by the computer without any further translation; a file that has execute permission, such as an **`a.out`** file, or a shell script.

exit

The **`exit`** function causes a process to terminate. **`exit`** closes any open files and cleans up most other information and memory used by the process. An exit status, or return code, is an integer value that your program returns to the operating system to say whether it completed successfully or not. For further information, see the *Concurrent C Reference Manual*.

expedited data

Data that is considered urgent. The specific semantics of expedited data is defined by the transport protocol that provides the transport service.

expedited transport service data

The amount of expedited user data the identity of which is preserved from one end of a transport connection to the other.

expert level

Second-lowest of the four lower RPC levels. Programs written to this level can control client and server characteristics, interface with **`rpcbind`** and manipulate service dispatch.

expression

An expression is a mathematical or logical symbol or meaningful combination of symbols.

FALSE

A value to which a Boolean descriptor can evaluate. FALSE must be the word “false,” irrespective of case, or a non-zero return code.

File Class Database

Contains file class definitions where each definition consists of a file class name and a list of properties. The properties define the visual and metaphor behavior of files belonging to the file class.

file descriptor

A file descriptor is an integer value assigned by the operating system to a file when the file is opened by a process.

file system type

Each different file system implementation that is incorporated into the VFS architecture is referred to as a file system type. A file system type may support different file types. The traditional System V file system type, a secure file system type, a high performance file system type, and an MS-DOS file system type are examples of potential file system types.

file system

A UNIX file system is a hierarchical collection of directories and other files that are organized in a tree structure. The base of the structure is the root (/) directory; other directories, all subordinate to root, are branches. The collection of files can be mounted on a block special file. Each file of a file system appears exactly once in the inode list of the file system and is accessible via a single, unique path from the root directory of the file system. For further information, see the *User's Guide.*

file type

The general expected characteristics of a file are determined by its file type. File types include regular file, character special file, block special file, FIFO, directory, and symbolic link. Each file type is supported within some file system type

file

A file is a potential source of input or a potential destination for output; at some point, then, an identifiable collection of information. A file is known to the UNIX operating system as an inode plus the information the inode contains that tells whether the file is a plain file, a special file, or a directory. A plain file contains text, data, programs, or other information that forms a coherent unit. A special file is a hardware device or portion thereof, such as a disk partition. A directory is a type of file that contains the names and inode addresses of other plain, special, or directory files. For further information, see the *User's Guide.*

filter

A filter is a program that reads information from the standard input, acts on it in some way, and sends its result to the standard output. It is called a filter because it can be used in a pipeline (see **pipe**) to transform the output of another program. Filters are different from editors in that they do not change the contents of a file. Examples of UNIX operating system filters are **`sort`**, which sorts the input, and **`wc`**, which counts the number of words, characters, and lines in the input. See **`sort(1)`** and **`wc(1)`** for more information.

flag

See **argument**.

flat widget

See **widget**. A single widget that maintains a collection of similar user-interface components that together give the appearance and behavior of many widgets.

flattened widget

Same as **flat widget**.

FMLI application

An application developed using the Form and Menu Language Interpreter (FMLI) to provide and maintain a user interface relying only on standard characters. An FMLI application can provide access to other applications.

focus

To specify a particular area of the screen. (See **input focus** and **keyboard focus**).

folder

A folder represents a directory in a file system. A folder can contain other folders and files.

foreground process group

Each session that has established a connection with a controlling terminal will distinguish one process group of the session as the foreground process group of the controlling terminal. This group has certain privileges when accessing its controlling terminal that are denied to background process groups.

fork

`fork` is a system call that splits one process into two, the parent process and the child process, with separate, but initially identical, text, data, and stack segments. See **`fork(2)`** for more information.

form field

An area of a form consisting of a field label and a field input area into which a user can enter input.

form

A visual element of an FMLI application displayed in a frame. A form is made up of fields that allow a user to provide input to the application.

frame definition file

A file in which the contents, appearance, functionality, and placement of a menu, form, or text frame are defined using the Form and Menu Language.

frame ID number

A number assigned by FMLI to a frame when it is opened. A frame ID number appears at the left in the title bar of a frame. The frame ID number allows users to navigate among frames by number.

frame

An independently-scrollable, bordered region of the screen, used to display FMLI forms, menus, and text. A frame includes a title bar, frame border, contents, and—for frames containing more than three lines—a scroll box.

fundamental block size

The minimal file allocation unit. In the case of disk-based file systems this is a disk sector or a multiple of disk sectors, smaller than or equal to the preferred block size (see below).

gadget

A windowless object; an object that could be defined as a widget but, instead, is defined as having its parent's window resources.

grab

To position the mouse pointer on a *resize corner* and take hold of it for the purpose of resizing the window.

hard key

A physical key on a computer's keyboard. For example, the “Return” or “Enter” key is illustrated as `Return`.

header file

A header file is a file that usually contains shared data declarations that are to be copied into source files by the compiler. Header file names conventionally end with the characters **`.h`**. Header files are also called include files, for the C language **`#include`** directive by which they are made available to source files. For further information, see the *Concurrent C Reference Manual.*

I/O

I/O stands for input/output, the process by which information enters (input) and leaves (output) a computer system. For further information, see the *Concurrent C Reference Manual.*

icon

A graphical representation of an object. The visual consists of a glyph and a label centered below the glyph. In FMLI, it is a symbol used to indicate an available function. For example, the caret (`^`) is an icon displayed in a frame's border to indicate the contents can be scrolled upwards.

ideogram

An ideogram is a language symbol usually based on a pictorial representation of an object or concept. An ideogram may or may not have a phonetic value.

include file

See **header file**.

initial frame

The frame, or frames, named as arguments when the **`fmli`** command is invoked. Initial frames are displayed automatically when an FMLI application is started, and remain on display in the work area until the FMLI session is terminated.

initialization file

A script in which an FMLI developer can define global attributes of an application using the Form and Menu Language. Such things as a transient introductory frame, a customized banner line, colors for various display elements, and restrictions on user access to the UNIX system can be defined. An initialization file is optional, but if one is written it must be named as an argument when **`fmli`** is invoked.

input focus

To have the cursor on a particular field, designating that field as “next.”.

instance

A specific *realization* of a widget; one particular widget as opposed to a class of widgets.

intermediate level

Second-highest of the four lower RPC levels; programs written to this level specify the transport they require.

interpreter

A program that allows you to communicate with the operating system. It reads the commands you enter and interprets them as requests to execute other programs, access files, or provide output.

interrupt

A signal to stop the execution of a process. From the keyboard, interrupts are usually initiated by pressing the `DELETE` or `BREAK` key. **`stty(1)`** will report the interrupt key for your session as `intr.` In FMLI, the ability of users to interrupt a process defined in an `action` or `done` descriptor can be enabled or disabled through the use of the `interrupt` descriptor.

interrupt

A signal to stop the execution of a process. From the keyboard, interrupts are usually initiated by pressing the `DELETE` or `BREAK` key. **`stty(1)`** will report the interrupt key for your session as `intr.` In FMLI, the ability of users to interrupt a process defined in an `action` or `done` descriptor can be enabled or disabled through the use of the `interrupt` descriptor.

ISO

ISO is an acronym for the International Standards Organization. ISO establishes standards in the computing industry for international markets.

kernel

The kernel is the basic resident software of the UNIX operating system. The kernel is responsible for most system operations: scheduling and managing the work done by the computer, maintaining the file system, and so forth. The kernel has its own text, data, and stack areas.

keyboard focus

The area of the *screen* that will accept the next input from the keyboard.

lexical analysis

Lexical analysis is the process by which a stream of characters (often comprising a source program) is broken up into its elementary words and symbols, called tokens. The tokens can include the reserved words of a programming language, its identifiers and constants, and special symbols such as `=`, `:=`, and `;`. Lexical analysis enables you to recognize, for instance, that the stream of characters `printf("hello, world\n");` is a series of tokens beginning with `printf` and not with, say, `printf("h`. In compilers, a lexical analyzer is often called by a syntactic analyzer, or parser, that analyzes the grammatical form of tokens passed to it by the lexical analyzer. For further information, see *UNIX Software Development Tools*

library

A library is a file that contains object code for a group of commonly used functions. Rather than write the functions yourself, you arrange for the functions to be linked with your program when an executable is created (see "**archive**"), or when it is run (see **shared object**).

line discipline

The line discipline is a STREAMS module that processes line data in the I/O stream to control the format and flow of data into and out of the system—erase and kill character handling, for example. See also **stream**.

link editing

Link editing refers to the process in which a symbol referenced in one module of a program is connected with its definition in another. With the C compilation system, programs are linked statically, when an executable is created, or dynamically, when it is run. For further information, see the *Concurrent C Reference Manual*.

local management

The phase in either connection-mode or connectionless-mode in which a transport user establishes a transport endpoint and binds a transport address to the endpoint. Functions in this phase perform local operations, and require no transport layer traffic over the network.

makefile

A **`makefile`** is a file that is used with the program **`make`** to keep track of the dependencies between modules of a program, so that when one module is changed, dependent ones are brought up to date. For further information, see *UNIX Software Development Tools.*

menu frame

A screen display showing a number or choices from which a user can make a selection(s), and which invokes some action when a selection is made.

MENU

The mouse button or keyboard equivalent used to display (*pop up*) a menu.

menu

When unqualified, any of the three states of a GUI menu: *popup menu*, *stay-up menu*, or *pinned menu*.

message line

The third line from the bottom of the screen in FMLI applications, used to display one-line messages and instructions to the user.

message queue identifier

A message queue identifier (`msqid`) is a unique positive integer created by a `msgget` system call. Each `msqid` has a message queue and a data structure associated with it.

message queue

In a stream, a linked list of messages awaiting processing by a module or driver.

message

In a stream, one or more blocks of data or information, with associated STREAMS control structures. Messages can be of several defined types, which identify the message contents. Messages are the only means of transferring data and communicating within a stream.

metacharacters

Metacharacters are ASCII characters with special meanings during pattern processing.

module

A module is a program component that typically contains a function or a group of related functions. Source files and libraries are modules.

multi-select menu

A menu which allows the user to mark one or more items and then select all marked items.

multiplexor

A multiplexor is a driver that allows streams associated with several user processes to be connected to a single driver, or several drivers to be connected to a single user process. STREAMS provides facilities for constructing multiplexors and for connecting multiplexed configurations of streams.

named key

A keyboard key which has a name indicating the function it performs. For example, TAB, DELETE, or ENTER.

network client

A process that makes remote procedure calls to services.

network service

A collection of one or more remote programs.

non-current

A frame, or other element on display which is not the element in which the cursor is currently positioned.

null pointer

In the C language, a null pointer is a C pointer with a value of 0.

object file

An object file contains a binary representation of programming language code. A relocatable object file contains references to symbols that have not yet been linked with their definitions. An executable object file is a linked program. Compare **source file**.

optimizer

An optimizer improves the efficiency of the assembly language code generated by a compiler. That, in turn, will speed the execution time of your object code. For further information, see the *Concurrent C Reference Manual.*

option

See argument.

orderly release

A procedure for gracefully terminating a transport connection with no loss of data.

orphaned process group

A process group in which the parent of every member in the group is either itself a member of the group, or is not a member of the process group's session.

pane

The rectangular area within a window where an application displays text or graphics.

parent process ID

A new process is created by a currently active process (see **`fork(2)`**). The parent process ID of a process is the process ID of its creator.

parent process

See **`fork`**.

parser

A parser, or syntactic analyzer, analyzes the grammatical form of tokens passed to it by a lexical analyzer (see **lexical analysis**). For further information, see *UNIX Software Development Tools*.

path name

A path name designates the location of a file in the file system. It is made up of a series of directory names that proceed down the hierarchical path of the file system. The directory names are separated by a slash character (`/`). The last name in the path is the file. If the path name begins with a slash, it is called an absolute, or full, path name; the initial slash means that the path begins at the root directory. A path name that does not begin with a slash is known as a relative path name, meaning relative to your current directory. For further information, see the *User's Guide.*

peer user

The user with whom a given user is communicating above the Transport Interface.

permissions

Permissions define a right to access a file in the file system. Permissions are granted separately to you, your group, and all others. There are three basic permissions: read, write, and execute. For further information, see the *User's Guide.*

ping

A call to procedure `0` of an RPC program. Pinging is used to verify the existence and accessibility of a remote program. Pinging can also be used to time network communications.

pipe

A pipe causes the output of one program to be used as the input to another program, so that the programs run in sequence. You create a pipeline by preceding each command after the first command with the pipe symbol (|) which indicates that the output from the process on the left should be routed to the process on the right.

```
$ who | wc -l
```

causes the output of the **`who`** command, which lists the users who are logged in to the system, to be used as the input of the **`wc`**, or word count, command with the **`-l`** option. The result is the number of users logged in to the system. See **`who(1)`** and **`wc(1)`** for more information.

pixel

An addressable point on the *screen*.

pixmap

A bitmap of an area of the screen stored within the program. A “pixmap” is also a defined data type in the Xt Internists.

pointer

The *screen* representation of the location of the mouse or equivalent.

portability

Portability refers to the degree of ease with which a program can be moved, or ported, to a different operating system or machine.

post

The FMLI activity of reading and interpreting a frame definition file, displaying the frame described therein, and making that frame current.

preference

Synonymous with property settings or options. This document uses the term preference to avoid confusion with properties that mean name-value pairs.

preferred block size

The unit of transfer for block devices in read/write operations (also known as "logical block size").

preprocessor

A preprocessor is a program that prepares an input file for another program. The preprocessor component of the C compiler performs macro expansion, conditional compilation, and file inclusion.

press

The act of pressing a mouse button or keyboard key. This is distinct from the act of releasing the button or key, so that both can be discussed separately. Thus *"press"* SELECT means to press, but not release, the SELECT mouse button or keyboard equivalent key.

press-drag-release

A method of user interaction with a set of objects where the user presses MENU to display the objects, *drags* the pointer over the objects until it is over the one of interest, then releases MENU to select or activate the object.

primitive widget

See *widget.* A widget that does not have any child widgets; one that either performs a specific action, allows input or allows output.

privilege

Having appropriate privilege means having the capability to perform sensitive system operations (see **`procpriv(2)`**).

process group ID

Each active process is a member of a process group and is identified by a positive integer called the process group ID. This ID is the process ID of the group leader. This grouping permits the signaling of related processes (see **`kill(2)`**).

process group leader

A process group leader is a process whose process ID is the same as its process group ID.

process group lifetime

A process group lifetime begins when the process group is created by its process group leader, and ends when the lifetime of the last process in the group ends or when the last process in the group leaves the group.

process group

Each process in the system is a member of a process group that is identified by a process group ID. Any process that is not a process group leader may create a new process group and become its leader. Any process that is not a process group leader may join an existing process group that shares the same session as the process. A newly created process joins the process group of its parent.

process lifetime

A process lifetime begins when the process is forked and ends after it exits, when its termination has been acknowledged by its parent process. See **`wait(2)`**.

process

An instance of a program being executed. A number that identifies an active process. In the UNIX System, it incorporates the concept of an execution environment, including contents of memory, register values, name of the current directory, status of files, and various other information. See **`ps(1)`** for more information on how to determine the process ID of any process currently active on your system.

program

A set of instructions and data kept in an ordinary file.

push a button

The act of moving the *pointer* to a *button widget* and then *selecting* the button.

pushpin

A *screen* object that is part of a *popup menu*. It can be pointed to and selected. When it is first selected, it is “pushed in” and causes the menu to stay up after the user moves out of it. When it is again selected, it is pulled out and the menu pops down.

quota

A mechanism for restricting the amount of file system resources that a user can obtain. The quota mechanism sets limits on both the number of files and the number of disk blocks that a user may allocate. Implemented by UFS.

read queue

In a stream, the message queue in a module or driver containing messages moving upstream.

real group ID
real user ID

Each user allowed on the system is identified by a positive integer (0 to `UID_MAX`) called a real user ID. Each user is also a member of a group. The group is identified by a positive integer called the real group ID. An active process has a real user ID and real group ID that are set to the real user ID and real group ID, respectively, of the user responsible for the creation of the process.

realized

In the context of the X Toolkit Intrinsics, the point at which all the data structures of a widget have been allocated. Windows and other information are not created when the widget is created with the `XtCreateWidget` routine, but are created in a later call to `XtRealizeWidget` on the widget itself or on an ancestor widget.

register, registration

To make a routine name known to the API. When the application programmer develops a *callback* routine, that routine needs to be registered when the widget is created so that it can be properly invoked.

regular expression

A regular expression is a string of alphanumeric characters and special characters that describes, in a shorthand way, a pattern to be searched for in a file. For further information, see *UNIX Software Development Tools*.

release

The act of releasing a pressed button or keyboard key, as in "release MENU."

remote program

Software that implements one or more remote procedures.

resize corners

Hollow, L-shaped symbols located on all four corners of a *window* which, when *grabbed*, are used to change the size of the *window*.

resource translation

The mechanism by which resource values are made accessible to widgets. The list of resources is contained in the **`app-defaults`** files. Each entry in these files consists of a resource name/value pair of the form: *app_name*.*resource_name*: *value*. Using an asterisk in place of the *app_name* makes the entry available to any application that recognizes the *resource_name*. Any hardcoded value takes precedence over what is set in the resource file.

resource

An attribute of a widget or a widget class. A resource is a named data value in the defining structure of a widget.

root directory/current directory

Each process has associated with it a concept of a root directory and a current directory for the purpose of resolving pathname searches. The root directory of a process need not be the root directory of the root file system.

routine

A routine is another name for a function.

RPC language

A C-like programming language recognized by the `rpcgen` compiler.

RPC Package

The collection of software and documentation used to implement and support remote procedure calls in System V. The RPC Package implements and is a superset of the functionality of the RPC Protocol.

RPC Protocol

The message-passing protocol that is the basis of the RPC package.

RPC/XDR

See *RPC language*.

saved group ID
saved user ID

The saved user ID and saved group ID are the values of the effective user ID and effective group ID prior to an **`exec`** of a file (see **`exec(2)`**).

screen

The surface on your computer monitor where information is displayed.

screen-labeled keys

The eight function keys, `F1` through `F8`, found on many keyboards, to which the labels displayed on the last line of the screen in FMLI applications correspond. The screen-labels indicate the operations assigned to the function keys.

script

A file which contains the definition of a frame (a frame definition file), the definition of global attributes of an FMLI application (an initialization file), the definitions of application specific commands (a commands file), a list of aliases for pathnames (an alias file), or UNIX system shell commands.

scroll indicators

Symbols contained in the scroll box of FMLI frames, to indicate that additional material is available above or below the current frame borders. The up symbol is a caret (^) or up-arrow character, and the down indicator is a `v` or down-arrow character.

scrolling

An attribute of FMLI frames which allows a fixed-size frame to accommodate a larger amount of information than can be displayed in it at one time. The first frameful of information is displayed when the frame is opened, and users can press named keys or their alternate keystrokes to move forward to a new frameful of information, or to move back to a previous frameful.

SELECT

The mouse button or keyboard equivalent used to select and move an object, manipulate a control, or set the input focus.

select

To move the *pointer* to an object and press the `SELECT` mouse button. The result is to initiate either an application action or a change in the window content or structure.

Selection Mechanism

The primary mechanism that X11 defines for clients that want to exchange information. Refer to both Xlib and Inter-Client Communication Manual (ICCCM, [5]) documents for more details.

semaphore identifier

A semaphore identifier (`semid`) is a unique positive integer created by a **`semget`** system call. Each `semid` has a set of semaphores and a data structure associated with it.

serializing

Converting data from a machine-specific representation to XDR format.

server

The transport user in connection-mode that offers services to other users (clients) and enables these clients to establish a transport connection to it.

service indication

The notification of a pending event generated by the provider to a user of a particular service.

service primitive

The unit of information passed across a service interface that contains either a service request or service indication.

service request

A request for some action generated by a user to the provider of a particular service.

session ID

Each session in the system is uniquely identified during its lifetime by a positive integer called a session ID, the process ID of its session leader.

session Leader

A session leader is a process whose session ID is the same as its process and process group ID.

session lifetime

A session lifetime begins when the session is created by its session leader, and ends when the lifetime of the last process that is a member of the session ends, or when the last process that is a member in the session leaves the session.

session

A session is a group of processes identified by a common ID called a session ID, capable of establishing a connection with a controlling terminal. Any process that is not a process group leader may create a new session and process group, becoming the session leader of the session and process group leader of the process group. A newly created process joins the session of its creator.

shared memory identifier

A shared memory identifier (`shmid`) is a unique positive integer created by a **`shmget`** system call. Each `shmid` has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. (Note that these shared memory segments must be explicitly removed by the user after the last reference to them is removed.)

shared object

A shared object, or dynamically linked library, is a single object file that contains the code for every function in the library. When you call a library function in your program, and specify a dynamic linking option on the **cc** command line, the entire contents of the shared object are mapped into the virtual address space of your process at run time. As its name implies, a shared object contains code that can be used simultaneously by different programs at run time. For further information, see the *Concurrent C Reference Manual.*

shell

The shell is the UNIX system program that handles communication between you and the system. The shell is known as a command interpreter because it translates your commands into a language understandable by the system. A shell normally is started for you when you log in to the system. A shell program calls the shell to read and execute commands contained in an executable file. For further information, see the *User's Guide,* and the **sh(1)** page.

signal

A signal is a message you send to a process or that processes send to one another. You might use a signal, for example, to initiate an interrupt (see above). A signal sent by a running process is usually a sign of an exceptional occurrence that has caused the process to terminate or divert from the normal flow of control.

simplified interface

The simplest level of the RPC package.

single-select menu

A menu from which a user can select only one item at a time.

SLK

See screen-labeled keys.

source file

Source files contain the programming language version of a program. Before a computer can execute the program, the source code must be translated by a compiler and assembler into the machine language of the computer. Compare “**object file**.”

source

The starting point of the drag-and-drop operation. It is also referred as the holder.

special processes

The process with ID 0 and the process with ID 1 are special processes referred to as **proc0** and **proc1**; see **kill(2)**. **proc0** is the process scheduler. **proc1** is the initialization process (**init**); **proc1** is the ancestor of every other process in the system and is used to control the process structure.

standard error

Standard error is an output stream from a program that normally is used to convey error messages. On the UNIX operating system, the default case is to associate standard error with the user's terminal.

standard input

Standard input is an input stream to a program. On the UNIX operating system, the default case is to associate standard input with the user's terminal.

standard output

Standard output is an output stream from a program. On the UNIX operating system, the default case is to associate standard output with the user's terminal.

static data

Static represents a condition persistent throughout a process. Static data occupies the data segment and the bss segment.

static linking

Static linking refers to the process in which external references in a program are linked with their definitions when an executable is created. See the *Concurrent C Reference Manual*.

stream head

In a stream, the stream head is the end of the stream that provides the interface between the stream and a user process. The principal functions of the stream head are processing STREAMS-related system calls, and passing data and information between a user process and the stream.

stream

A stream is a full-duplex data path within the kernel between a user process and driver routines. The primary components are a stream head, a driver and zero or more modules between the stream head and driver. A stream is analogous to a shell pipeline except that data flow and processing are bidirectional.

STREAMS

A set of kernel mechanisms that support the development of network services and data communication drivers. It defines interface standards for character input/output within the kernel and between the kernel and user level processes. The STREAMS mechanism is composed of utility routines, kernel facilities and a set of data structures.

string

A string is a contiguous sequence of characters treated as a unit. In the C language, a character string is an array of characters terminated by the null character, `\0`.

sub-object

A sub-object is the equivalent of a *primitive widget* contained in a *flattened widget*. In a Flat Exclusives or F NonExclusives widget, the sub-objects are the equivalents of `RectButtons`. In a Flat CheckBox, the sub-objects are the equivalents of `CheckBox` widgets.

SVID

System V Interface Definition, which defines the standard interface for SVR4 and is the basis of other UNIX operating system standards.

syntax

Command syntax is the order in which commands and their arguments must be put together. The command always comes first. The order of arguments varies from command to command. Language syntax is the set of rules that describes how the elements of a programming language may legally be used.

system call

A system call is a request from a program for an action to be performed by the UNIX operating system kernel. For further information, see the *Programming in Standard* manual.

templates

Files that are used to be the initial structure and/or content of newly created files. Template files are specified for each file class by the `TEMPLATES` class property. (OEMs and ISVs can add new templates).

terminal attributes

Characteristics of the video screen which can be manipulated by an FMLI application developer to provide visual cues to the application's functionality. They include underlining, half-bright, bright, and blinking display of characters, an alternate character set for line drawing, and others.

text frame

a visual element of an FMLI application displayed in a frame. A text frame displays lines of text; for example, help on how to fill in a form field.

text symbol

A text symbol names a program instruction. Instructions reside in read-only memory during execution. Compare “**data symbol**.”

toggle

This is an action performed on an object with two states; it is the switching from one state to the other.

top level

Highest of the four lower RPC levels; programs written to this level specify the type of transport they require

translation

See **resource translation**.

transport address

The identifier used to differentiate and locate specific transport endpoints in a network.

transport connection

The communication circuit that is established between two transport users in connection-mode.

transport endpoint

The local communication channel between a transport user and a transport provider.

transport interface

The library routines and state transition rules that support the services of a transport protocol.

transport provider

The transport protocol that provides the services of the Transport Interface.

transport service data unit

The amount of user data whose identity is preserved from one end of a transport connection to the other.

transport user

The user-level application or protocol that accesses the services of the Transport Interface.

TRUE

A value to which a Boolean descriptor can evaluate. Any value other than those defined for FALSE is interpreted as TRUE.

TSDU

Transport Service Data Unit.

UFS

The Unified File System, a derivative of the 4.2BSD file system. It offers file hardening, supports large and fragmented block allocations for files, and distributed inode and free block management. Additionally, it supports quotas (see above).

universal address

A machine-independent representation of a network address.

upstream

In a stream, the direction from driver to stream head.

user ID

A user ID is an integer value, usually associated with a login name, that the system uses to identify owners of files and directories. The user ID of a process becomes the owner of files created by the process and by descendent processes (see **`fork`**).

utility

A software tool of general programming usefulness built-in to FMLI, such as `fmlgrep` or `message,` which can be used inside backquoted expressions, and which is executed when the backquoted expression is evaluated. A built-in utility has a performance advantage over a UNIX shell utility in that it does not fork a new process.

variable

In a program, a variable is an object whose value may change during the execution of the program or from one execution to the next. A variable in the shell is a name representing a string of characters.

virtual circuit transport

See *connection-oriented transport.*

virtual circuit

A transport connection established in connection-mode.

Vnode

The operating system's internal representation of a file (previously known as a file-system-independent inode).

white space

One or more space, tab, and/or newline characters. White space is normally used to separate strings of characters, and is required to separate a command from its arguments when it is invoked. For this toolkit, these characters are space, tab, newline, and `Return`.

widget class

A collection of code and data structures that provides a generic implementation of a part of a look-and-feel.

widget

A specific example or realization of a *widget class*.

window

A work area on the screen that you use to run and display an application.

word wrapping

An attribute of text frames which prevents words from being split across two lines when the text frame is displayed. Word wrapping can be turned on or off by the developer in the text frame definition file.

work area

In FMLI applications, the area of the screen running from the second line from the top to the fourth line from the bottom. The work area is used to display menus, forms, and text frames.

wrapping

An attribute of frames which allows a user to navigate through a list of menu items or form fields as if it were a circular list. Forward or backward navigation keys always cause movement to the next logical item or field. The next logical item or field may differ according to the navigation key being used.

write queue

In a stream, the message queue in a module or driver containing messages moving downstream.

X/Open

X/Open is short for the X/Open Company Limited, a consortium of computer firms dedicated to achieving open UNIX systems.

XDR language

A protocol specification language for data representation. RPC language builds on and is a superset of XDR.

XDR

eXternal Data Representation. Provides an architecture independent representation of data.

zombie

A process that has executed the **`exit`** system call and no longer exists, but which leaves a record containing an exit code and some timing statistics for its parent to collect. The zombie state is the final state of a process.

Index

Spine for 2” Binder

Product Name: 0.5" from top of spine, Helvetica, 36 pt, Bold

Volume Number (if any): Helvetica, 24 pt, Bold

Volume Name (if any): Helvetica, 18 pt, Bold

Manual Title(s): Helvetica, 10 pt, Bold, centered vertically within space above bar, double space between each title

Bar: 1" x 1/8" beginning 1/4" in from either side

Part Number: Helvetica, 6 pt, centered, 1/8" up

PowerMAX OS

Programmer

Programming Guide

0890423