NightStarRT Tutorial

(Version 3.1)

0898009-050
March 2006

Preface

General Information

NightStar RT™ allows users running RedHawk Linux to schedule, monitor, debug and analyze the run-time behavior of their time-critical applications as well as the operating system kernel.

NightStar RT consists of the NightTrace™ event analyzer; the NightProbe™ data monitoring tool; the NightView™ symbolic debugger; the NightSim™ scheduler; the NightTune™ system and application tuner; the Data Monitoring API; and the Shmdefine shared memory utility.

Scope of Manual

This manual is a tutorial for NightStar RT.

Structure of Manual

This manual consists of five chapters which comprise the tutorial for NightStar RT.

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 and comments in code 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. Keywords also appear in `list` type.
emphasis	Words or phrases that require extra emphasis use emphasis type.
window	Keyboard sequences and window features such as push buttons, radio buttons, menu items, labels, and titles appear in window 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.
`{ }`	Braces enclose mutually exclusive choices separated by the pipe (`|`) character, where one choice must be selected. You do not type the braces or the pipe character with the choice.
`...`	An ellipsis follows an item that can be repeated.
`::=`	This symbol means *is defined as* in Backus-Naur Form (BNF).

Referenced Publications

The following publications are referenced in this document:

0898395	*NightView RT User's Guide*
0898398	*NightTrace RT User's Guide*
0898515	*NightTune RT User's Guide*
0898465	*NightProbe RT User's Guide*
0898480	*NightSim RT User's Guide*

Contents

1
Overview

NightStar RT™ is an integrated set of debugging tools for developing time-critical Linux® applications. NightStar RT tools run at application speed with minimal intrusion, thus preserving execution behavior and determinism. Users can quickly and easily debug, monitor, analyze, and tune their applications.

NightStar RT graphics-based tools reduce test time, increase productivity, and lower development costs. Time-critical applications require debugging tools that can handle the complexities of multiple processors, multi-task interaction, and multithreading. NightStar RT advanced features enable system builders to solve difficult problems quickly.

The NightStar RT tools consist of:

- NightView™ source-level debugger
- NightTrace™ event analyzer
- NightProbe™ data monitor
- NightTune™ system and application tuner
- NightSim™ scheduler

In this tutorial, we will integrate these tools into one cohesive example incorporating various scenarios which demonstrate their extensive functionality.

Getting Started

Certain activities in this tutorial require enhanced user privileges which are not granted to user accounts by default. You will need to run as the root user, where indicated within this tutorial, or obtain approprirate privileges as detailed in the "Setting Up User Privileges" on page 1-2.

Setting Up User Privileges

Linux provides a means to grant otherwise unprivileged users the authority to perform certain privileged operations. **`pam_capability(8)`**, the Pluggable Authentication Module, is used to manage sets of capabilities, called roles, required for various activities.

Linux systems should be configured with an `nightstar` role which provides the capabilities required by NightStar RT. In order to take full advantages of NightStar RT features, each user must be configured to use (at a minimum) the capabilities specified below.

Edit **`/etc/security/capability.conf`** and define the `nightstar` role (if it is not already defined) in the "ROLES" section:

```
role nightstar cap_sys_nice cap_ipc_lock
```

Additionally, for each NightStar RT user on the target system, add the following line at the end of the file:

`user` *username* `nightstar`

where *username* is the login name of the user.

If the user requires capabilities not defined in the `nightstar` role, add a new role which contains `nightstar` and the additional capabilities needed, and substitute the new role name for `nightstar` in the text above.

In addition to registering your login name in **`/etc/security/capability.conf`**, files under the **`/etc/pam.d`** directory must also be configured to allow capabilities to be activated.

To activate capabilities, add the following line to the end of selected files in **`/etc/pam.d`** if it is not already present:

```
session required pam_capability.so
```

The list of files to modify is dependent on the list of methods that will be used to access the system. The following table presents a recommended configuration that will grant capabilities to users of the services most commonly employed in accessing a system.

Table 1-1. Recommended /etc/pam.d Configuration

/etc/pam.d File	Affected Services	Comment
remote	telnet rlogin rsh (when used w/o a command)	Depending on your system, the **remote** file may not exist. Do not create the **remote** file, but edit it if it is present.
login	local login (e.g. console) telnet* rlogin* rsh* (when used w/o a command)	*On some versions of Linux, the presence of the **remote** file limits the scope of the **login** file to local logins. In such cases, the other services listed here with **login** are then affected solely by the **remote** configuration file.
rsh	rsh (when used with a command)	e.g. **rsh system_name a.out**
sshd	ssh	You must also edit **/etc/sshd/sshd_config** and ensure that the following line is present: **UsePrivilegeSeparation no**
gdm	gnome sessions	
kde	kde sessions	

If you modify **/etc/pam.d/sshd** or **/etc/sshd/sshd_config**, you must resetart the **sshd** service for the changes to take effect:

```
On RedHawk systems: service sshd restart
On SUSE systems: bash /etc/init.d/sshd restart
```

In order for the above changes to take effect, the user must log off and log back onto the target system.

NOTE

To verify that you have been granted capabilities, issue the following command:

```
/usr/sbin/getpcaps $$
```

The output from that command will list the roles currently assigned to you.

Creating a Tutorial Directory

We will start by creating a directory in which we will do all our work. Create a directory and position yourself in it:

- Use the **mkdir(1)** command to create a working directory.

We will name our directory **tutorial** using the following command:

```
mkdir tutorial
```

- Position yourself in the newly created directory using the **cd(1)** command:

```
cd tutorial
```

Source files, as well as configuration files for the various tools, are copied to **/usr/lib/NightStar-RT/tutorial** during the installation of NightStar RT. We will copy these tutorial-related files to our **tutorial** directory.

- Copy all tutorial-related files to our local directory.

```
cp /usr/lib/NightStar-RT/tutorial/* .
```

Building the Program

Our example uses a cyclic multi-threaded program which performs various tasks during each cycle. The cycle will be controlled by the main thread which uses a timeout with a configurable rate.

A portion of the main source file, **app.c**, is shown below:

```
main()
{
   pthread_t thread;
   pthread_attr_t attr;
   struct sembuf trigger = { 2, 0, 0 };

   trace_begin ("/tmp/data",NULL);
   trace_open_thread ("main");

   sema = semget (IPC_PRIVATE, 1, IPC_CREAT+0666);

   ptrace_attr_init (&attr);
   Pthread_create (&thread, &attr, sine_thread, &data[0]);

   ptrace_attr_init (&attr);
   Pthread_create (&thread, &attr, cosine_thread, &data[1]);

   ptrace_attr_init (&attr);
   Pthread_create (&thread, &attr, heap_thread, NULL);

   for (;;) {
      struct timespec delay = { 0, rate };
      nanosleep (&delay, NULL);
      semop (sema, trigger, 1);
   }

}
```

The program creates three threads and then enters a loop which cyclically activates each of two threads based on a common timeout. The third thread, `heap_thread`, runs asynchronously.

To build the executable

- From the local **`tutorial`** directory, enter the following command:

```
cc -g -o app app.c -lntrace_thr -lpthread -lm
```

NOTE

The NightStar RT tools require that the user application is built with DWARF debugging information in order to read symbol table information from user application program files. For this reason, the **`-g`** compile option is specified. However, the tools can be used to debug programs without symbols with reduced functionality.

2
Using NightView

NightView is a graphical source-level debugging and monitoring tool specifically designed for time-critical applications. NightView can monitor, debug, and patch multiple processes running on multiple processors with minimal intrusion.

NightView supports all the features you find in standard debuggers, including:

- breakpoints
- single stepping through statements
- single stepping over function calls
- full symbolic expression analysis
- conditions and ignore counts for breakpoints
- hardware-assisted address traps (watchpoints)
- assembly and symbolic debugging

In addition to standard debugging capabilities, NightView provides the following features:

- application-speed eventpoint conditions
- the ability to patch code to change program flow or modify memory or registers during program execution
- hot patch and eventpoint control
- synchronous data monitoring
- loadable modules
- support of multi-threaded programs
- debugging of multiple processes
- dynamic memory debugging

Invoking NightView

NightView can be launched by issuing:

```
nview &
```

at the command prompt or by double-clicking on the desktop icon.

When we launch NightView, a NightView Dialogue window is presented.

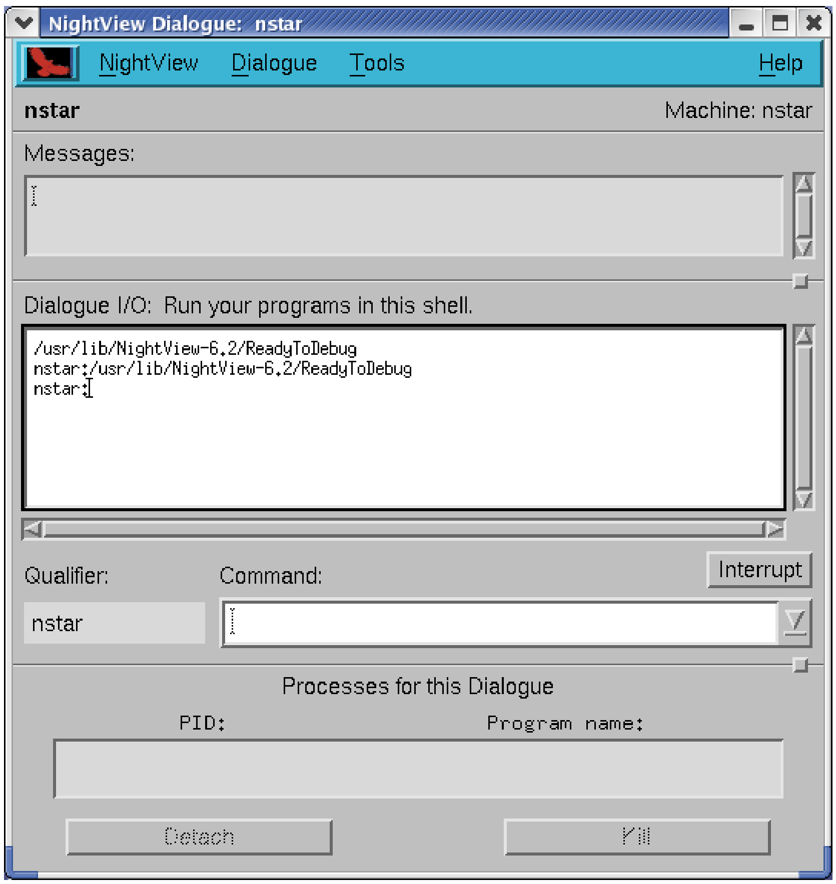


Figure 2-1. NightView Dialogue

Unlike other debuggers, the dialogue interface provides you a standard shell from which to execute user applications or other commands. By default, programs that are invoked from this shell come under the control of the debugger. Filters provide the capability to prevent specific programs or programs that match certain patterns from coming under the control of the debugger. By default, programs in **/usr/bin** and other common locations are ignored by NightView. This allows you to debug multiple applications that might have complex shell scripts required to start them.

In our example, we'll be debugging a single application.

- Invoke our tutorial application in the NightView Dialogue window by typing:

  ./app

  at the command prompt in the shell.

NOTE

If you have not yet created the **app** program, see "Building the Program" on page 1-4.

Any output generated by the program will appear in the dialogue window, just as it would in an **xterm** or similar program with an interior shell.

When the **app** program begins to execute, NightView stops the program and displays a Principal Debug Window from which most debugging operations are controlled.

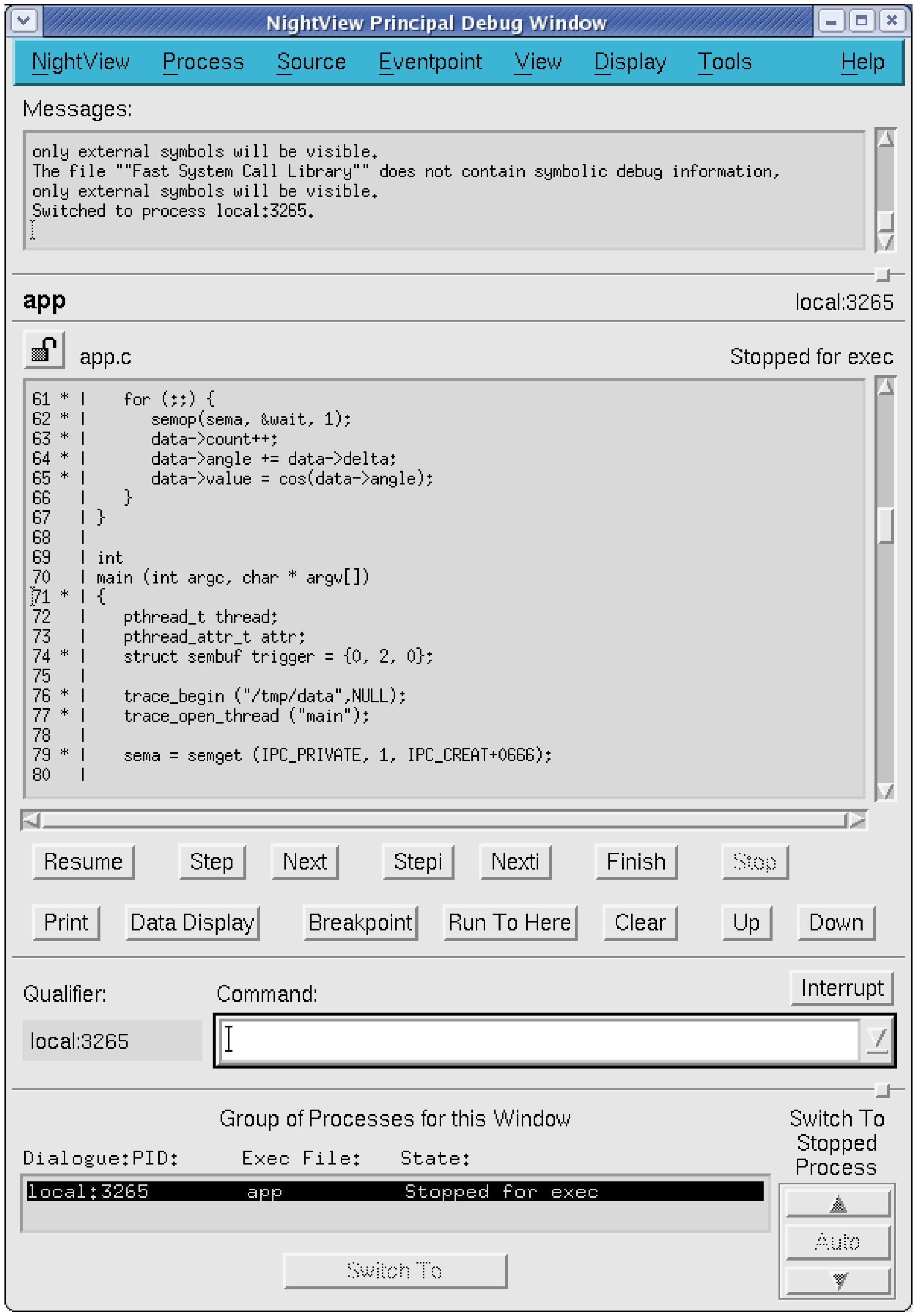


Figure 2-2. NightView Principal Debug Window

IMPORTANT

Do not resume execution of the program at this time.

NightView supports debugging multiple processes as well as single and multi-threaded processes. In this tutorial, you will be debugging a single process.

- To save screen space, hide the Process Group Area in the Principal Debug Window by clearing the Display Group Area checkbox in the View menu.

Heap Debugging

Debugging dynamic memory problems can be difficult and extremely time-consuming. The word *heap* refers to a collection of allocated and freed memory typically controlled by the `malloc()` and `free()` utilities in the C language.

NightView provides the unique ability to monitor and detect memory allocations, frees, and sets of user errors without requiring a non-standard allocator to be compiled or linked into your program.

One advantage of this is that often when you switch to a debugging allocator, the way blocks are allocated and freed changes -- often hiding the very bugs you're trying to find.

NightView offers a variety of settings and debugging levels that are useful in catching common heap-related errors. Some settings will change the behavior of the system allocator -- affecting the size of allocated blocks and ultimately, the address values returned.

Dynamic memory errors are detected in one of four ways:

- a check of the entire heap at a specified frequency when heap functions (e.g., `malloc`, `free`, `calloc`, etc.) are called
- a check of the entire heap when a **`heappoint`** is crossed
- a check of an individual allocated block when `free` or `realloc` is called
- a check of the entire heap when a **`heapcheck`** command is issued

The frequency setting of the **`heapdebug`** command controls how often NightView should check for heap errors when a utility routine is called. Setting the frequency to one causes NightView to check for heap errors on every heap operation.

A **`heappoint`** causes NightView to check for errors when the process executes instructions where the heappoint is inserted. An unlimited number of heappoints can be inserted into your program.

The check of an individual block when `free` or `realloc` is called is automatic.

All four mechanisms are useful. With the first three mechanisms, the heap error detection is executed at program application speed without context switching to the debugger.

Activating Heap Debugging

One limitation of heap debugging is that it requires that you activate the debugging before any allocations occur in your program. If you attempt to activate the heap debugging features after allocations have already occurred, NightView will inform you of its inability to satisfy your request.

NOTE

If you have mistakenly resumed execution of the program already, kill the program and restart it in the Dialogue window. Type `kill` in the Command area of the Principal Debug Window and press Enter. Go back to the Dialog window and type **`./app`** andn Enter in the dialog shell.

- Select the Debug Heap... menu option from the Process menu in the Principle Debug Window.

The Debug Heap window is shown.

- Select the On radio button in the Heap Debugger area.
- Press the 2(Medium) button in the Debugging Level area.
- Change the Check Heap frequency by typing 1 in the text field next to the Every button.

The Debug Heap window should look similar to the following figure:

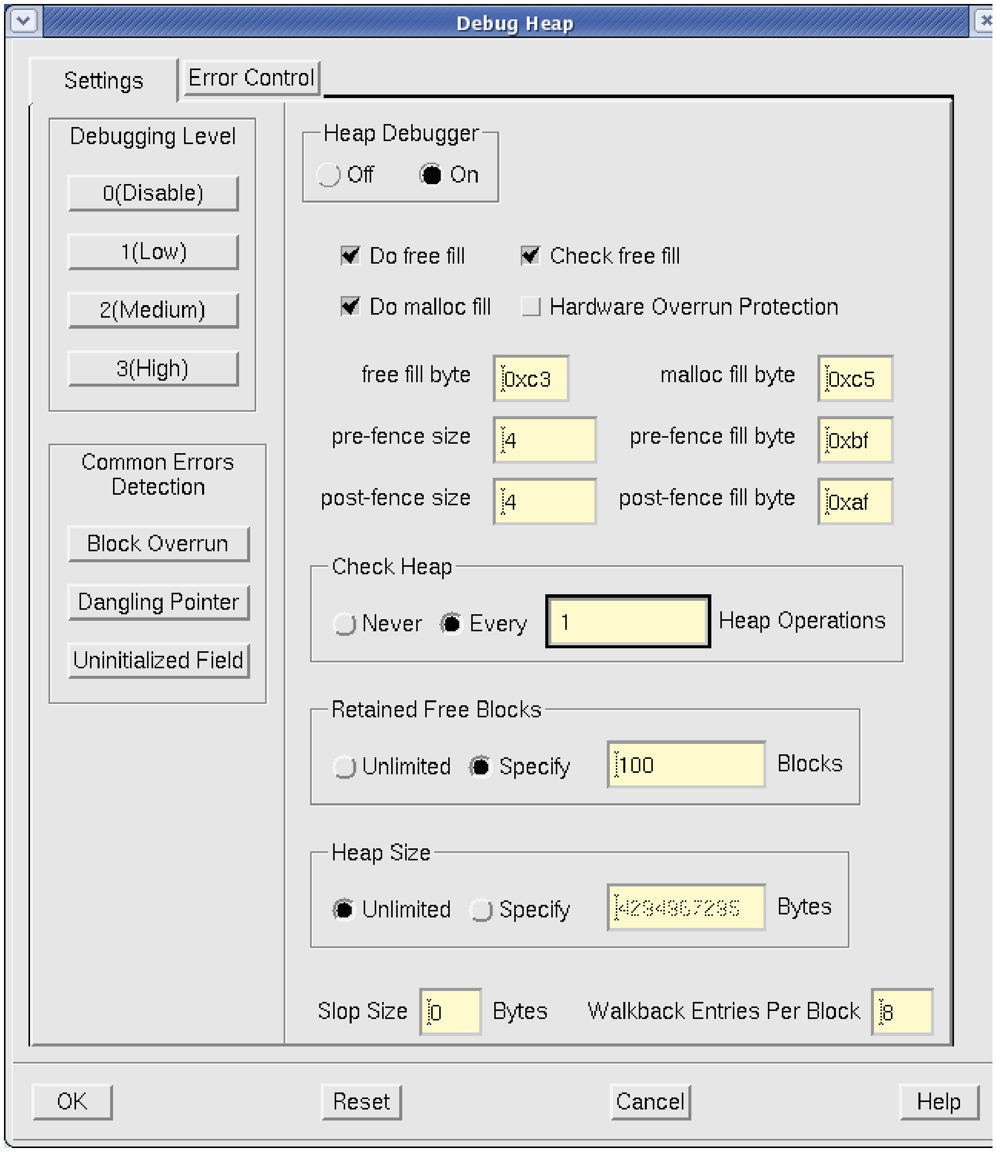


Figure 2-3. NightView Debug Heap Window

- Press the OK button to apply the changes and close the dialog.

These options instruct the debugger to activate heap debugging, retain freed blocks to detect certain kinds of errors, allocate some additional memory past the end of the requested size to detect errors, and stop the program when any heap error is detected.

Controlling the app Program

The third thread created by the main program executes a routine called `heap_thread`.

This routine iteratively executes various dynamic memory operations based on the setting of the `scenario` variable which are representative of common user errors relating to dynamic memory.

- Set a breakpoint on line 114:

  ```
  sleep(5);
  ```

  using either the Set Breakpoint option from the Eventpoint menu or the following command:

  ```
  break app.c:114
  ```

Scenario 1: Use of a Freed Pointer

A common error is to read or write a block of memory that has already been freed.

A way to detect this is to tell NightView to retain freed blocks and fill the freed blocks with a specific pattern. If the blocks are subsequently read, your application may more quickly discover the error since the contents are unexpected. If the blocks are subsequently written, NightView can detect this.

- Resume the process and let it reach the breakpoint on line 114:

```
resume
```

By default, the `heap_thread` will not actually execute any of the five scenarios.

- To cause it to execute scenario 1, set the variable `scenario` to 1:

```
set scenario=1
resume
```

This causes the following snippet of code to be executed after a delay of 5 seconds:

```
ptr = alloc_ptr(1024,3);
free_ptr (ptr,2);
memset (ptr, 47, 64);
```

The last line represents usage of dynamically allocated space that has already been freed.

NightView will detect this at a subsequent heap operation (based on the **frequency** setting of the **heapdebug** command) or at a heappoint inserted by the user, in this case on line 154.

NightView will stop the process once the heap error has been detected and issue a diagnostic similar to the following:

```
Heap error in process local:12345:
   free-fill modified in free block (value=0x804a4a8)
#0 0x8048b71 in heap_thread(void*unused=0) at app.c line 154
```

The error refers to the fact that locations within the freed block were modified by the process after the block was freed.

The Data Window is useful for displaying heap-related information as well as a variety of other attributes.

- Using the Display menu, select Heap Information... and press OK to add the item to the default Data Window.
- Likewise, select Local Variables... from the Display menu to add a list of local variables to the default Data Window.
- Expand the `Configuration` item under `Heap Information` in the Data Window to show the current **heapdebug** settings.
- Expand the `Totals` item under `Heap Information` to show summary statistics related to heap activity.

- Right-click on the box to the left of the first `Ever allocated` item and select Format and then Resize Label... from the pop-up menu. Type in a value of 40 in the text field and press OK.
- Increase the width of the Data Display window.

NOTE

> In general, all information in the Data Window is updated whenever the process being debugged stops.

- Collapse the `totals` and `configuration` items or expand the size of the Data Window so that the `Local Variables` item is shown.

The list of items underneath `Local Variables` changes each time the process stops to represent the local variables associated with the current frame being displayed. Note that the description of the variable `ptr` is displayed in red because it no longer contains a valid (allocated) heap address.

Expanding the `ptr` item reveals the (heap info) item. Expanding that item reveals additional information relating to the block that the pointer once referred to including:

- its state - freed, but retained (invalid)
- its address range
- its size
- walkback information relating to the routines which allocated and freed the block

The Data Window should appear similar to the following figure:

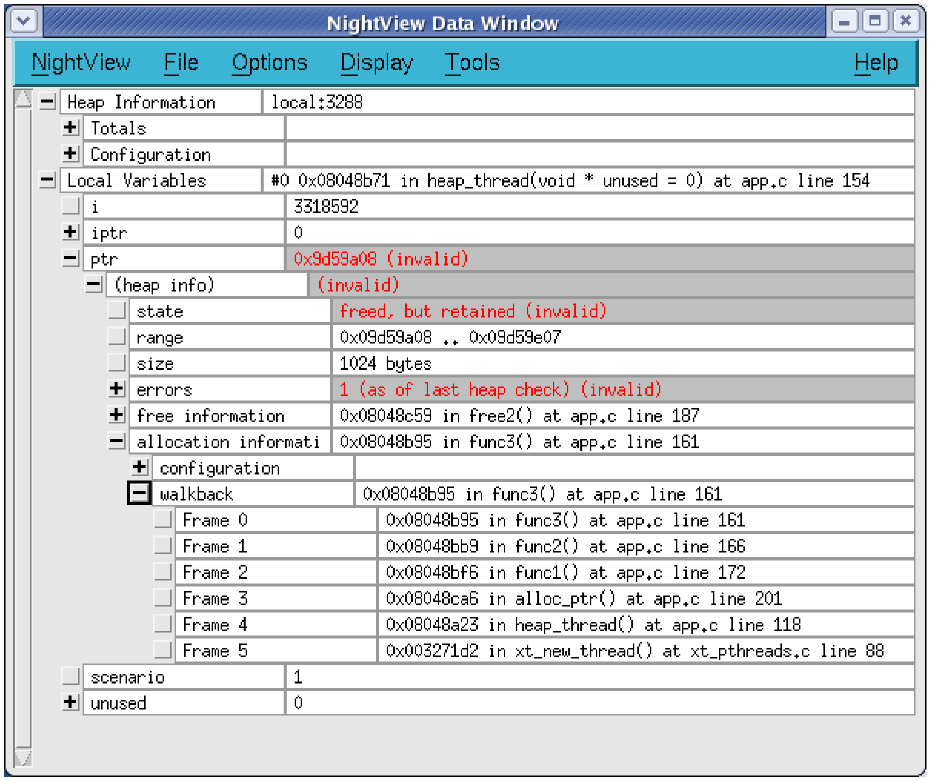


Figure 2-4. NightView Data Window

Scenario 2: Freeing an Invalid Pointer Value

Another common error is to free a pointer multiple times or to free a value which doesn't actually refer to a heap block.

- Resume the process and let it reach the breakpoint on line 114:

  `resume`

- Set the variable `scenario` to 2:

  ```
  set scenario=2
  resume
  ```

This causes the following snippet of code to be executed after a delay of 5 seconds:

```
ptr = alloc_ptr(1024,3);
free_ptr(ptr,2);
free(ptr);
```

NightView will detect the failure and print a diagnostic similar to the following:

```
Heap error in process local:12345: free called on freed or
unallocated block (value=0x804a5c8)
#0 0x804a5c8 in heap_thread(void*unused=0) at app.c line 126
```

Another way of obtaining information about the heap block in question is to use the **`info memory`** command. It provides textual output of the information available in the Data Window under the `ptr` item to the Messages area of the Principal Debug Window.

NOTE

> NightView optionally displays a Global Window which echoes all commands entered by the user as well as those initiated due to dialog usage. It also contains all output generated by NightView commands. To activate this window, select the Open Global Window menu item from the NightView menu.

- Issue the following command:

  `info memory ptr`

NightView will provide output similar to the following:

```
Memory map enclosing address 0x0887deb0 for process local:13433:

Virtual Address Range  No. bytes  Comments
---------------------  ---------  --------------------------
0x0887d000 0x0889dfff     135168  Readable,Writable

Allocator information for address 0x0887deb0 for process local:13433:

freed, but retained
in block 0x0887deb0 .. 0x0887e2af (1024 bytes)
no errors detected in block
free information:
   4 post-fence bytes with 0xaf (fence range 0x0887e2b0 .. 0x0887e2b3)
   4 pre-fence bytes with 0xbf (fence range 0x0887deac .. 0x0887deaf)
   free fill with 0xc3
   malloc fill with 0xc5
   walkback:
     0x08048c59 in free2() at app.c line 187
     0x08048c7d in free1() at app.c line 193
     0x08048cbf in free_ptr() at app.c line 206
     0x08048a72 in heap_thread() at app.c line 125
     0x00a911d2 in xt_new_thread() at xt_pthreads.c line 88
allocation information:
   4 post-fence bytes with 0xaf (fence range 0x0887e2b0 .. 0x0887e2b3)
   4 pre-fence bytes with 0xbf (fence range 0x0887deac .. 0x0887deaf)
   free fill with 0xc3
   malloc fill with 0xc5
   walkback:
     0x08048b95 in func3() at app.c line 161
     0x08048bb9 in func2() at app.c line 166
     0x08048bf6 in func1() at app.c line 172
     0x08048ca6 in alloc_ptr() at app.c line 201
     0x08048a5f in heap_thread() at app.c line 124
     0x00a911d2 in xt_new_thread() at xt_pthreads.c line 88
```

In this case, the walkback information associated with the actual free is useful as you can quickly locate what code segment actually freed the block.

Scenario 3: Writing Past the End of an Allocated Block

Another common error is to allocate insufficient space or to write past the end of an allocated block.

- Resume the process and let it reach the breakpoint on line 114:

  `resume`

- Set the variable `scenario` to 3:

  ```
  set scenario=3
  resume
  ```

This causes the following snippet of code to be executed after a delay of 5 seconds:

```
ptr = alloc_ptr(strlen(MyString),2);
strcpy (ptr, MyString); -- oops -- forgot the zero byte
```

NightView will detect the failure and print a diagnostic similar to the following:

```
Heap error in process local:12345:
   post-fence modified in allocated block (value=0x804a6a8)
#0 0x804aca8 in heap_thread(void*unused=0) at app.c line 154
```

Note that the description of the variable `ptr` in the `Local Variables` list in the **Data Window** does not indicate an invalid status. That is because `ptr` does point to a valid heap block.

Scenario 4: Use of Uninitialized Heap Blocks

Another common error is forgetting to initialize dynamically allocated memory before using it. Code segments may assume that dynamically allocated memory is initialized to zero, as is the case with `calloc()`, but not `malloc()`.

- Resume the process and let it reach the breakpoint on line 114:

    resume

- Set the variable `scenario` to 4:

```
handle sigsegv stop print pass
set scenario=4
resume
```

This causes the following snippet of code to be executed after a delay of 5 seconds:

```
iptr = (int**)alloc_ptr(sizeof(int*),2);
if (*iptr) **iptr = 2778;
```

NightView will detect the failure and print a diagnostic similar to the following:

```
Process local:12345 received SIGSEGV
#0 0x804aca8 in heap_thread(void*unused=0) at app.c line 137
```

The `malloc_fill` setting of the **heapdebug** command causes NightView to fill blocks being allocated with a specific byte pattern, in this case `0xc5`.

- Issue the following command to view the content of the uninitialized memory block:

    x/x iptr

A `SIGSEGV` signal is a fatal error so we must restart the process to continue the tutorial.

- Issue the following command:

    kill

  and then re-initiate the program in the Dialogue window by typing:

    ./app

  in the dialogue shell.

  Alternatively, you can issue the following command directly from the Principal Debug Window to initiate the process in the Dialogue shell:

    !./app

NOTE

NightView automatically re-applies all eventpoint and heapcontrol settings when it sees the subsequent execution of the program.

Scenario 5: Detection of Leaks

Another situation which may be indicative of error or inappropriate use of memory are leaks. In this instance, we define a leak as a dynamically allocated block of memory that is no longer referred to by any pointer in the program.

Detection of leaks is a ***very expensive*** process with respect to CPU utilization and intrusion on the user application. As such, leak detection is only executed when an explicit request is made from the user.

- Resume the process and let it reach the breakpoint on line 114:

  `resume`

- Set the variable `scenario` to 5:

  ```
  set scenario=5
  resume
  ```

This causes the following snippet of code to be executed after a delay of 5 seconds:

```
ptr = alloc_ptr(37,1);
ptr = 0;
```

NightView does detect the leak automatically, as mentioned above. The process will stop again when the breakpoint on line 114 is reached.

- At that time, specifically request a leak report by selecting Heap Leaks... from the Display menu and press OK to add the item to the default Data Window.

  This operation causes NightView to analyze the program for leaks and displays a `Leak Sets` item in the Data Window. On small programs, this operation may appear to be insignificant, but for larger programs it can take some significant time.

- Expand the `Leak Sets` item, if necessary.

  An additional item is displayed for every leak set with a matching block size that was allocated with a matching walkback. Expansion of individual sets provides the common walkback shown for each allocation as well as expandable descriptions of each individual leaked block.

- Expand the leak set item with size 37 and then expand the `walkback` item associated with it.

  Note the walkback indicating that it was allocated by the `heap_thread()` routine on line 141 of **`app.c`**.

NOTE

Unlike most items in the Data Window, the `leak sets` item is not automatically updated when the process stops. The description will remain static until you explicitly request a refresh operation. This can be accomplished by selecting Re-Evaluate from the pop-up menu launched when you right-click the box to the left of the `Leak Sets` item.

Scenario 6: Allocation Reports

NightView provides a detailed report of all allocated memory.

Construction of this report is a ***very expensive*** process with respect to CPU utilization and intrusion on the user application execution time. As such, allocation reports are only executed when an explicit request is made from the user.

- Set the variable `scenario` to 6:

```
set scenario=6
resume
```

This causes additional allocations to be made.

The process will stop again when the breakpoint on line 114 is reached.

- At that time, specifically request an allocation report by selecting Still Allocated Blocks... from the Display menu and press OK to add the item to the default Data Window.

This operation causes NightView to analyze the program and displays a `Still Allocated Blocks` item in the Data Window. On small programs, this operation may appear to be insignificant, but for larger programs it can take some significant time.

- Select the Resize Label... menu item from the Format option in the pop-up menu launched by right-clicking the box to the left of the `Still Allocated Sets` item. Type in a value of 30 in the text area, check the Apply change to children checkbox and press OK.
- Expand the `Still Allocated Sets` item, if necessary. An additional item is displayed for every allocation set with a matching block size that was allocated with a matching walkback. Expansion of individual sets provides the common walkback shown for each allocation as well as expandable descriptions of each individual leaked block.
- Expand the `allocated set` item with size 1048576 and then expand the `walkback` item associated with it.

  Note the walkback indicating that it was allocated by the `heap_thread()` routine on line 146 of **`app.c`**.

NOTE

Unlike most items in the Data Window, the `Still Allocated Sets` item is not automatically updated when the process stops. The description will remain static until you explicitly request a refresh operation. This can be accomplished by selecting Re-Evaluate from the pop-up menu launched when you right-click the box to the left of the `Still Allocated Sets` item.

Disabling Heap Debugging

To disable all overhead associated with heap debugging, issue the following command:

```
heapdebug off
```

This concludes the tutorial's topic on heap debugging. We will now continue on to other capabilities of NightView.

Debugging Multiple Threads

At this point in the tutorial the user application should be stopped at line 114 in **app.c**.

NOTE

If the application is not stopped at line 114, set a breakpoint on line 114 in **app.c** and resume the process until it stops on that line number. Refer to the previous section for instructions on setting breakpoints and resuming the process.

Our application consists of the main thread and three additional ones created by the main thread.

When the application hits a breakpoint or is otherwise stopped by NightView, all threads in the application will stop. Similarly, when NightView resumes execution of a thread, all threads will resume execution.

- Collapse the expanded items in the Data Window.
- Select the Threads... option from the Display menu and press OK to add the `Threads` list to the Data Window.
- Right-click on the box to the left of the `C Threads` item and select the Expand Tree... option from the pop-up menu. Type a 2 in the text area and press OK.

  This causes the list of threads to be expanded and shows the stack walkback for each individual thread.

Expanding an individual Frame in the walkback list shows all local variables for that frame. You can further expand composite and pointer variables in the local variables items.

The figure below shows such an expansion indicating that the main thread is in the `main()` routine; another thread is in the `sine_thread()` routine; another in the `cosine_thread()` routine; and finally, the last thread is in the `heap_thread()` routine.

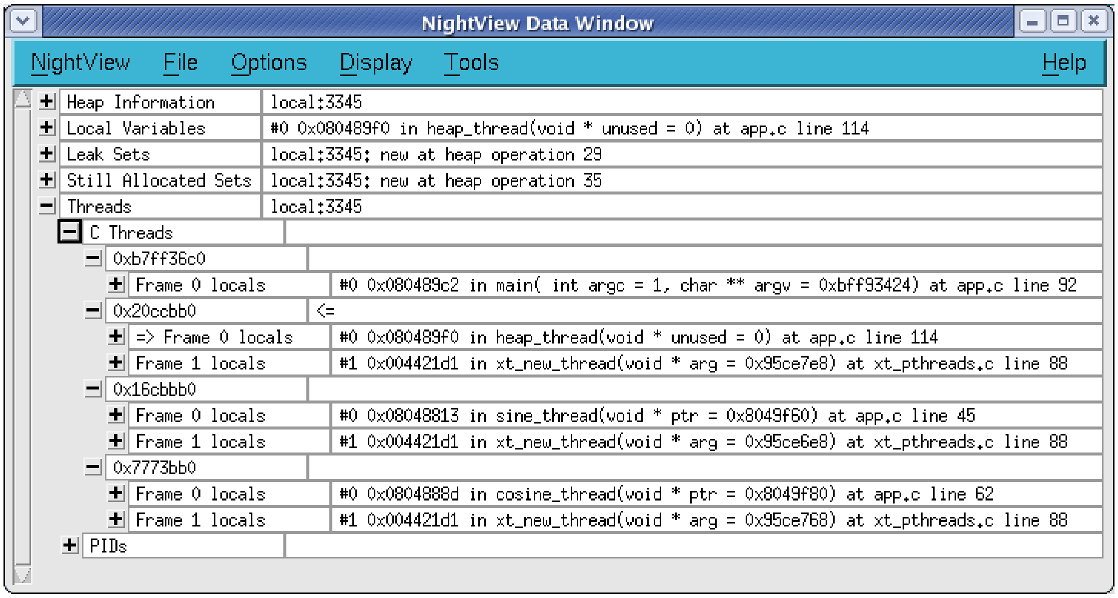


Figure 2-5. NightView Display Window - Threads

The context shown in the Principal Debug Window is that of the thread which caused the process to stop. You can tell which thread you are stopped in by looking for the "<=" indicator on the Data Window to the right of the thread number.

You can switch to the context of other threads by selecting the Select Frame option from the pop-up menu launched by right-clicking the box to the left of a thread of interest.

Alternatively, you can use the **`select-context`** command and specify the thread ID as shown in the `C Threads` display or from the output of the **`info threads`** command:

```
info threads /v
select-context thread=0x401ab4e0
```

- Switch to the context of the thread executing `sine_thread()` by selecting the Select Frame option from the pop-up menu launched by right-clicking the box to the left of the `Thread` item.

  The source displayed in the Principal Debug Window changes to line 45 on a call to `semop()`.

NOTE

It is possible that the context of the thread in question could be executing on any line in the range of 44-48.

The <> indicator near the line number in the Principal Debug Window represents the fact that we are positioned at a stack frame which is not the topmost stack frame and that the current frame is executing a subprogram call.

By default, NightView hides uninteresting frames. If you desire to see all frames for all routines, even those that have no debug information, you can set your **`interest threshold`** to the keyword `min`:

```
int thresh min
```

Once that command is issued, the walkback information shows all frames and you can position to any frame and debug at the assembly level if desired.

- Reset the **`interest threshold`** to zero via the following command:

  ```
  int thresh 0
  ```

- Remove the breakpoint on line 114 using the Summarize/Change... item from the Eventpoint menu or issue the following command:

  ```
  clear app.c:114
  ```

  before proceeding to the next section.

Using Monitorpoints

Monitorpoints provide a means of monitoring the values of variables in your program without stopping it. A monitorpoint is code inserted by the debugger at a specified location that will save the value of one or more expressions, which you specify. The saved values are then periodically displayed by NightView in a Monitor Window.

Unlike asynchronous sampling, monitorpoints allow you to view data which is synchronized with execution of a particular location in your application.

- Select the Set Monitorpoint... option from the Eventpoint menu to launch the Set a New Monitorpoint dialog.
- Ensure that the Location text field has app.c:45, correcting if it need be.
- Enter the following:

```
print id="sine count" data->count
print id="sine value" data->value
```

in the Commands text box and press OK.

A Monitor Window is opened containing an entry for the commands entered above.

- Likewise, set a monitorpoint on line 62 with the same commands as in the previous monitorpoint, substituting `cosine` for `sine` in the optional `id` parameter.

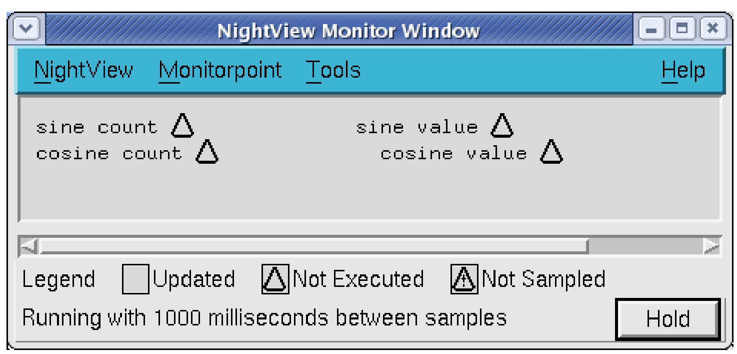


Figure 2-6. NightView Monitor Window

- Resume execution of the process.

  At this point, the data values in the Monitor Window change.

  The values are sampled whenever line 45 or 62 are executed. NightView displays the latest set of values in the Monitor Window at a user-selectable rate.

NOTE

A significant feature of the NightView is the ability to execute most debugging operations without having to stop execution of the process.

All subsequent debugging operations in this tutorial can be done without stopping the process!

Using Eventpoint Conditions and Ignore Counts

All eventpoints in NightView have optional `Condition` and `Ignore` attributes.

A `Condition` is a user-supplied boolean expression of arbitrary complexity which is evaluated before the eventpoint is executed. Conditions can involve function calls in the user application.

Similarly, the `Ignore` attribute is a count of the number of times to ignore an eventpoint before actually executing it.

Conditions and ignore counts are evaluated by the application itself via patched-in code and, as such, run at full application speed. Other debuggers evaluate the conditions and ignore counts from within the context of the debugger which takes significant time and can drastically affect the behavior of your program.

- To demonstrate these capabilities, select the Summarize/Change... option from the Eventpoint menu.
- Select the first eventpoint in the list and press Change... to launch the Change This Monitorpoint dialog.
- Enter 500 in the Ignore Count text field and press OK.
- Press Close in the Summarize and Change Eventpoints dialog.

The Monitor Window now indicates that the values for that monitorpoint have not been sampled by displaying an exclamation point enclosed within a triangle. When the ignore count reaches zero, the values will start updating again.

Finally, monitorpoints can include complex expressions that aren't just simple variables.

- Enter the following commands in the Principal Debug Window:

```
monitor app.c:92
   p FunctionCall()
end monitor
```

A new item is added to the Monitor Window which represents the result of the function call `FunctionCall()` as executed by the user application each time line 92 is crossed.

Using Patchpoints

Unlike breakpoints and monitorpoints, patchpoints allow you to modify the behavior of your program.

Patchpoints allow you to change program flow or modify variables or machine registers.

First, we will use a patchpoint to branch around some statements in our program.

NOTE

If the source file **app.c** is not displayed, issue the following command:

```
l app.c:47
```

- Scroll the source file displayed in the Principal Debug Window and click on line 47:

```
data->angle += data->delta
```

- Select the Set Patchpoint... option from the Eventpoint menu to launch the Set a New Patchpoint dialog.
- In the Location text area, ensure the text indicates app.c:47.
- Click on the Branch to a different location radiobutton in the lower portion of the dialog.
- In the Go to: text area, type:

```
app.c:48
```

  then press the OK button.

This will effectively cause the application to skip execution of line 47, where it updates the angle used in the subsequent `sin()` call.

Note that the `sine value` in the Monitor Window stops changing, yet the associated `sine count` value continues to change.

Alternatively, we can use patchpoints to change the value of expressions or variables.

- Type the following command in the Principal Debug Window:

```
patch app.c:48 eval data->count -= 2
```

Note that the value of `sine count` is decrementing.

We can disable the patchpoints without deleting them.

- Select the Summarize/Change... option from the Eventpoint menu. Select both active patchpoints (as indicated in the `Type` column by the letter `P`) and press Change...

Two dialogs will pop-up.

- Click the Disable radiobutton and press OK in both dialogs.
- Press Close to close the Summarize and Change Eventpoints dialog.

The patches are disabled and the values shown in the Monitor Window return to their original behavior.

Adding and Replacing Functions Dynamically

NightView provides the ability to dynamically add new functions to the application being debugged, as well as to replace existing functions.

- In a terminal session outside of NightView, compile the **`report.c`** source file which was copied into your current directory in the initial steps of this tutorial:

  `cc -g -c report.c`

- Load the new module into the program using the following command in the Principal Debug Window:

  `load report.o`

NOTE

The source displayed in the Principle Debug Window may change as a result of the load command. This annoyance will be addressed in the future.

We have added a simple function which prints information to **`stdout`**. The function could have been arbitrarily complex and referenced any variable in the application. The only limitation is that the function cannot reference symbols that are absent from the module being loaded and are not already in the user application.

- Issue the following command to see the source code for the function `report()`:

  `l report.c`

You will see that the `report()` function expects a char * descriptor and a double value.

- Go back to the application source file by issuing the following command:

  `l app.c`

We will install a new patchpoint which will call the newly added function.

- Issue the following command:

  `patch app.c:62 eval report("cos",data->value)`

See that the program is now generating output to **`stdout`** in the NightView Dialogue window as calls to the `report()` function are executed.

- Disable the patchpoint that was just added by issuing the following command:

  `disable .`

The *dot* parameter to the **`disable`** command is a short-hand notation for the last eventpoint created; in this case, the eventpoint created by the **`patch`** command above.

Finally, we will replace a function that already exists in the application.

- In a terminal session outside of NightView, list the contents of the source file **`function.c`** which was copied into your current directory in the initial steps of this tutorial, and compile it with the following commands:

```
cat function.c
cc -g -c function.c
```

- Now load the replacement code via the following command:

```
load function.o
```

Note how the Monitor Window value for the `FunctionCall()` value no longer pertains to the value computed by the application, but rather is a monotonically increasing number as per the source file **`function.c`**.

- Return the Principal Debug Window source display to the **`app.c`** source file via the following command:

```
l app.c:40
```

Using Tracepoints

The last portion of NightView we will cover in this tutorial is integration with NightTrace.

A tracepoint is a specialized eventpoint which essentially patches a call to log a trace event with optional arguments.

The current limitation on tracepoints is that the application must already have linked with the NightTrace API library and has made just two API calls to initiate tracing.

Our application satisfies this requirement.

Suppose that we were interested in measuring the performance of our cycles in the `sine_thread()` and `cosine_thread()` routines and that we also were interested in logging data values during the cycle.

- Select Set Tracepoint... from the Eventpoint menu to launch the Set a New Tracepoint dialog.
- In the Location: text field type in:

  app.c:46

  and Event ID: text field type:

  1

  and the press the OK button.

Similarly, we'll set additional tracepoints using the **tracepoint** command.

- Enter the following commands in the Principal Debug Window:

```
tracepoint 2 at app.c:45 value=data->value
tracepoint 3 at app.c:62 value=data->value
```

Trace events can now be logged with the NightTrace tool which is described in the next section of this tutorial.

- Launch NightTrace by selecting the NightTrace Analyzer menu item from the Tools menu of the Principal Debug Window.
- For clarity, minimize all NightView windows before proceeding to the next section.

Conclusion - NightView

This concludes the NightView portion of the NightTrace RT User's Guide.

NOTE

Do not exit NightView or stop the application. The next section uses the tracepoints that were inserted in the previous section (see "Using Tracepoints" on page 2-32).

3 Using NightTrace

NightTrace is a graphical tool for analyzing the dynamic behavior of single and multiprocessor applications. NightTrace can log user-defined application data events from simultaneous processes executing on multiple CPUs or even multiple systems. NightTrace can also log Linux kernel events such as individual system calls, context switches, machine exceptions, page faults and interrupts. By combining application events with Linux kernel events, NightTrace presents a synchronized view of the entire system. Furthermore, NightTrace allows users to zoom, search, filter, summarize, and analyze those events in a wide variety of ways.

Using NightTrace, users can manage multiple user and kernel NightTrace daemons simultaneously from a central location. NightTrace provides the user with the ability to start, stop, pause, and resume execution of any of the daemons under its management.

NightTrace users can define and save a “session” consisting of one or more daemon definitions. These definitions include daemon collection modes and settings, daemon priorities and CPU bindings, and data output formats, as well as the trace event types that are logged by that particular daemon.

Invoking NightTrace

NightTrace was invoked during the last step of the Using NightView section.

If you skipped the Using NightView section, execute the steps in “Using Tracepoints” on page 2-32 before beginning this section of the tutorial.

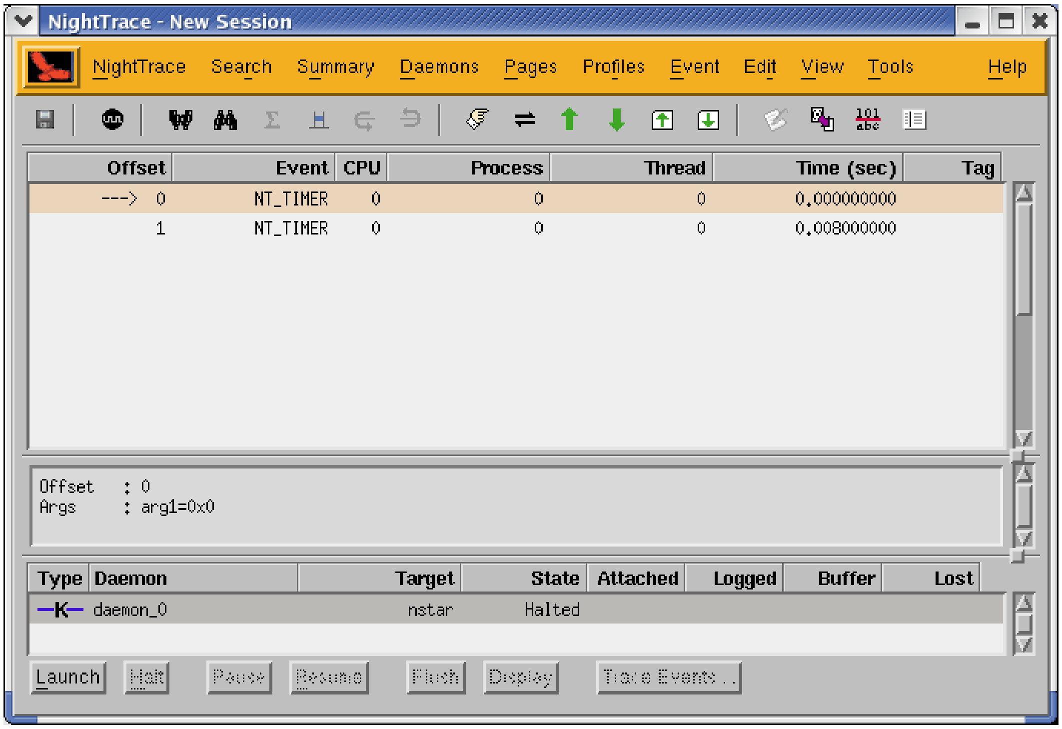


Figure 3-1. NightTrace main window

Configuring a User Daemon

NightTrace allows the user to configure a user daemon to collect user trace events.

User trace events are generated by user applications that use the NightTrace API.

We will configure a user daemon to collect the events that our **app** program logs.

To configure a user daemon

- From the Daemons menu on the NightTrace main window, select the New... menu item.

  The Daemon Definition dialog is displayed:

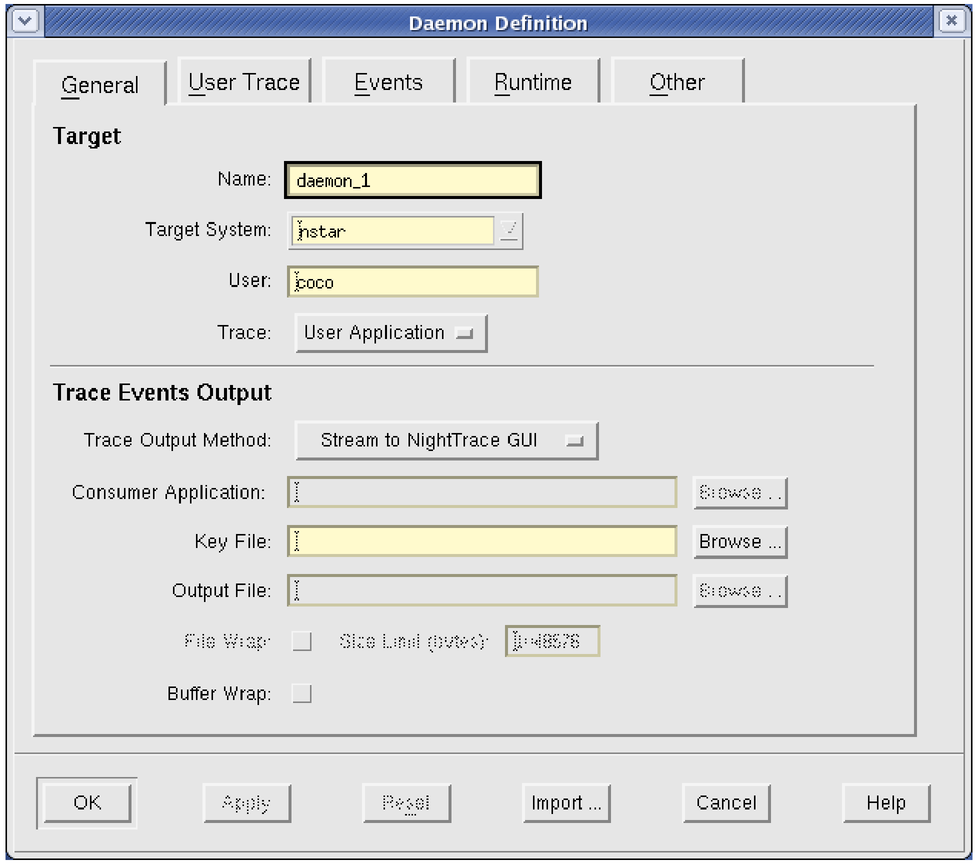


Figure 3-2. Daemon Definition dialog

- Press the Import... button at the bottom of the Daemon Definition dialog.

  You will be presented with a Login dialog.

  - Enter the name of the system on which the **app** application is running in the Target System field.
  - Enter your login name on that system in the User field.
  - Press the OK button.

  The Import Daemon Definition dialog is presented:

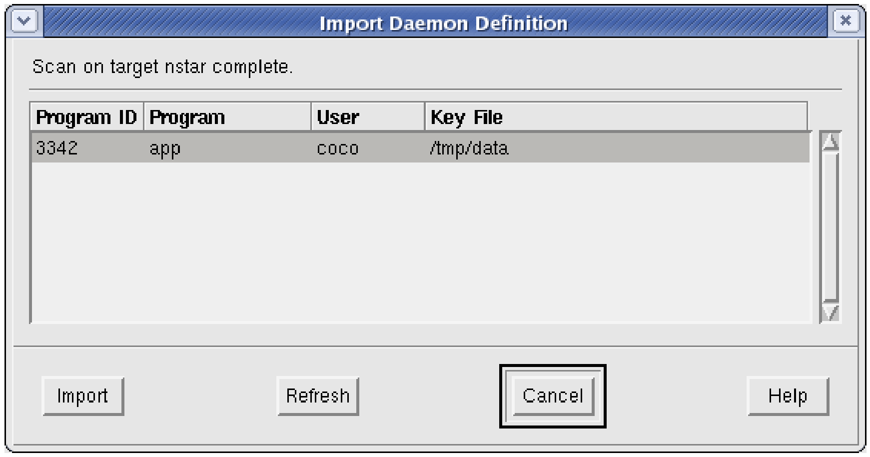


Figure 3-3. Import Daemon Definition dialog

The Import Daemon Definition dialog allows the user to define daemon attributes based on a running user application containing NightTrace API calls.

- Select the entry corresponding to the **app** application.
- Press the Import button.

The Import Daemon Definition dialog closes and the Daemon Definition dialog is populated with the imported attributes.

- Press OK on the Daemon Definition dialog to complete the configuration of the user application daemon.

Streaming Live Data to the NightTrace GUI

NightTrace allows you to use a daemon to capture trace events and store them in a file for subsequent analysis or to stream the events directly into the graphical interface for live analysis.

Our daemon is configured for live streaming.

- Select the daemon labelled `daemon_1` from the bottom of the Daemon Display Area in the NightTrace main window.
- Press the Launch button.
- Press the Resume button.

The daemon is now collecting events which are being generated by the **app** program from the tracepoints we inserted in “Using Tracepoints” on page 2-32.

NOTE

If you plan to leave the turorial for an extended period of time before returning, press the Pause button to temporarily prevent the collection of trace points. When you return, press the Resume button.

NOTE

An additional window is launched with Launch is pressed. This window is a automatically customized display page which we will use later on in the tutorial. The description immediately below refers to the NightTrace Main window.

The statistics on the Daemon Display Area indicate the number of raw events in the shared memory buffer used between the daemon and the user application and the number of raw events written to NightTrace by the daemon (under the Buffer and Logged columns, respectively).

The Trace Segment Display Area indicates the number of processed events that are currently available for immediate analysis through the Event Display Area and other display pages.

You can force events to be flushed from the daemon buffer and output stream to be brought into the segment area for immediate viewing by a variety of methods:

- pressing the Flush button in the Daemon Display Area
- sliding the scroll bar in the Event Display Area all the way to the bottom
- sliding the scroll bar in display windows all the way to the right
- zooming all the way out in display windows

Bring in data for analysis with the following actions:

- Press the Flush button.
- Slide the scrollbar in the Events list all the way down to the bottom and then release.

Using the Main Window for Textual Analysis

The NightTrace main window is used for controlling daemons and data segments and textual analysis of trace event information. The events shown in the Events list are synchronized with the events shown in display pages. The event with a salmon-colored background and the “--->” indicator indicates the current timeline.

- Click on a line in the Events list.
- Press the Down Arrow key to advance to the next event.

- Press the Up Arrow key to advance to the previous event.

Whenever an event is selected or the current event line moves, the Text Display Area shows additional information about the event, if applicable.

- Press the PageDown key to advance one page.
- Press the PageUp key to advance to the previous page.

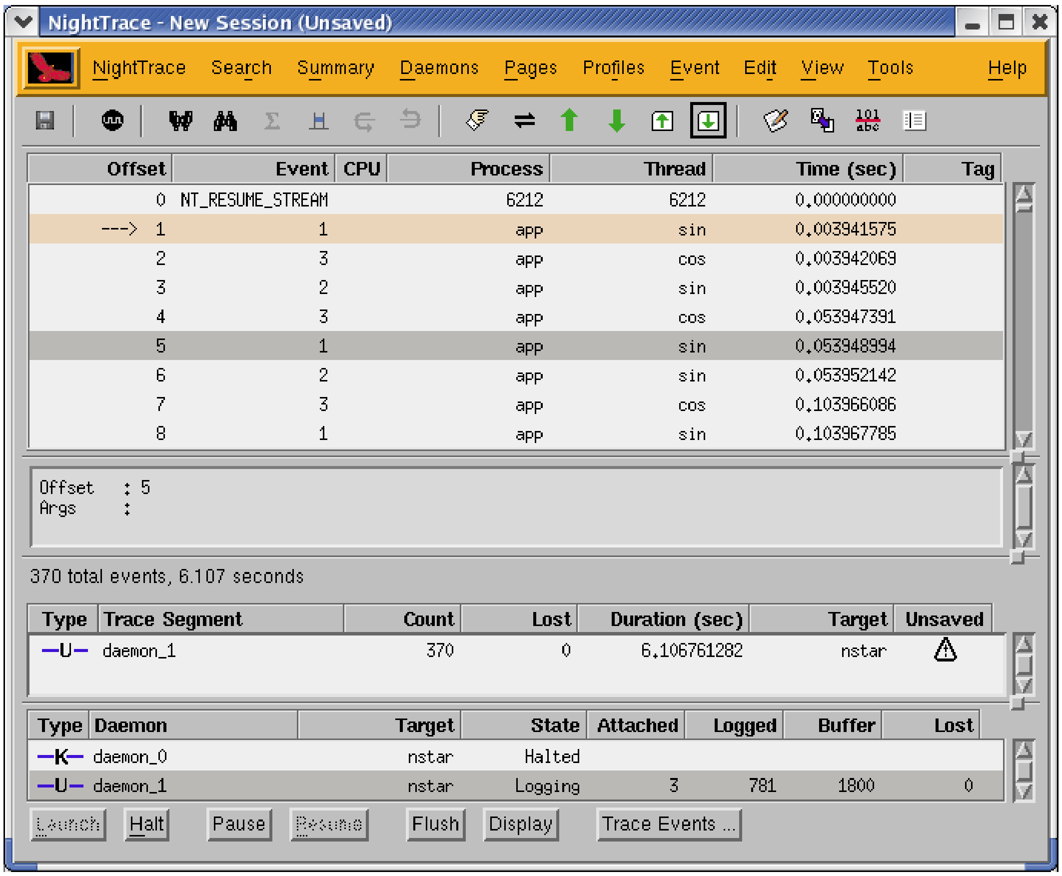


Figure 3-4. NightTrace Main Window - Events List

Customizing Event Descriptions

The event values we logged with the tracepoint commands in NightView were event IDs 1 and 2. We will customize the description of these events using the Edit String Table dialog.

- Press the Edit Events icon in the tool bar 

  The Edit String Table dialog is displayed:

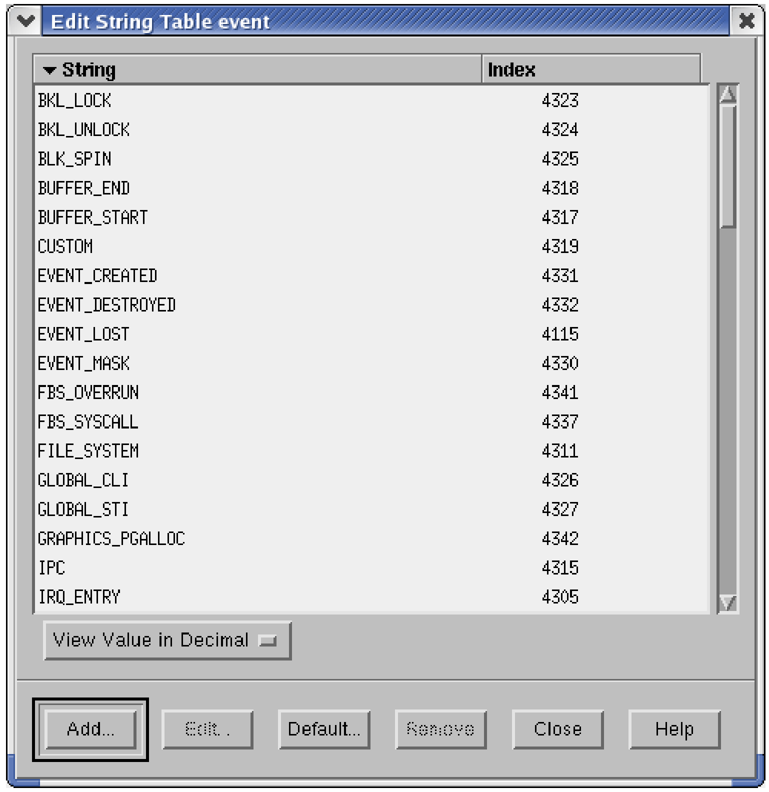


Figure 3-5. Edit String Table dialog

- Press the `Add...` button.

  The `Edit Event Map Entry` dialog is displayed:

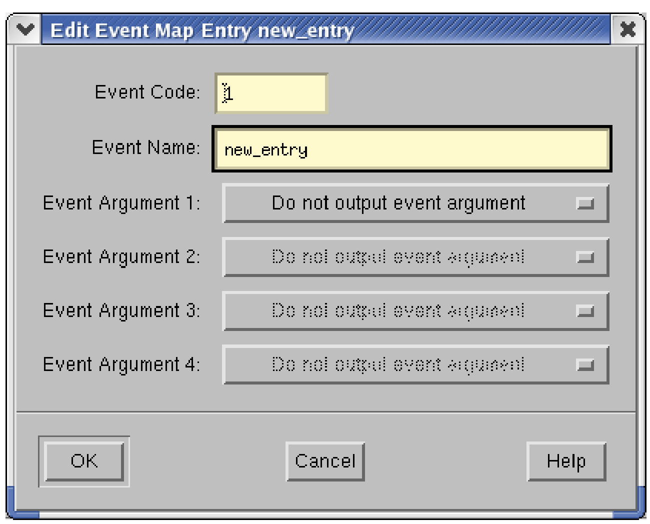


Figure 3-6. Edit Event Map Entry dialog

- Enter:

  `cycle_start`

  in the Event Name text field.

- Press OK.
- Press the Add... button again.
- Type in 2 in the Event Code text field.
- Enter:

  `cycle_end`

  in the Event Name text field.

- Select “Output event argument as float” from the Event Argument 1 option list.
- Press the OK button.
- Press Close to close the Edit String Table dialog.

The description of the events in the Events list now correspond to the textual identifiers we encoded in the previous dialogs. Additionally, when a `cycle_end` event is selected, the textual description includes the value of argument 1 formatted as a floating point value. This value, the result of the `sin()` calculation in the `sine_thread()` routine, is logged with the event from the tracepoint inserted via NightView (see “Using Tracepoints” on page 2-32).

Searching the Events List

We can use the search capabilities of the Profiles dialog to search for a specific occurrence of an event, or condition relating to an event or its arguments.

- Select the Search... menu item from the Search menu in the NightTrace main window or press Ctrl+F.

  The Profiles dialog is displayed:

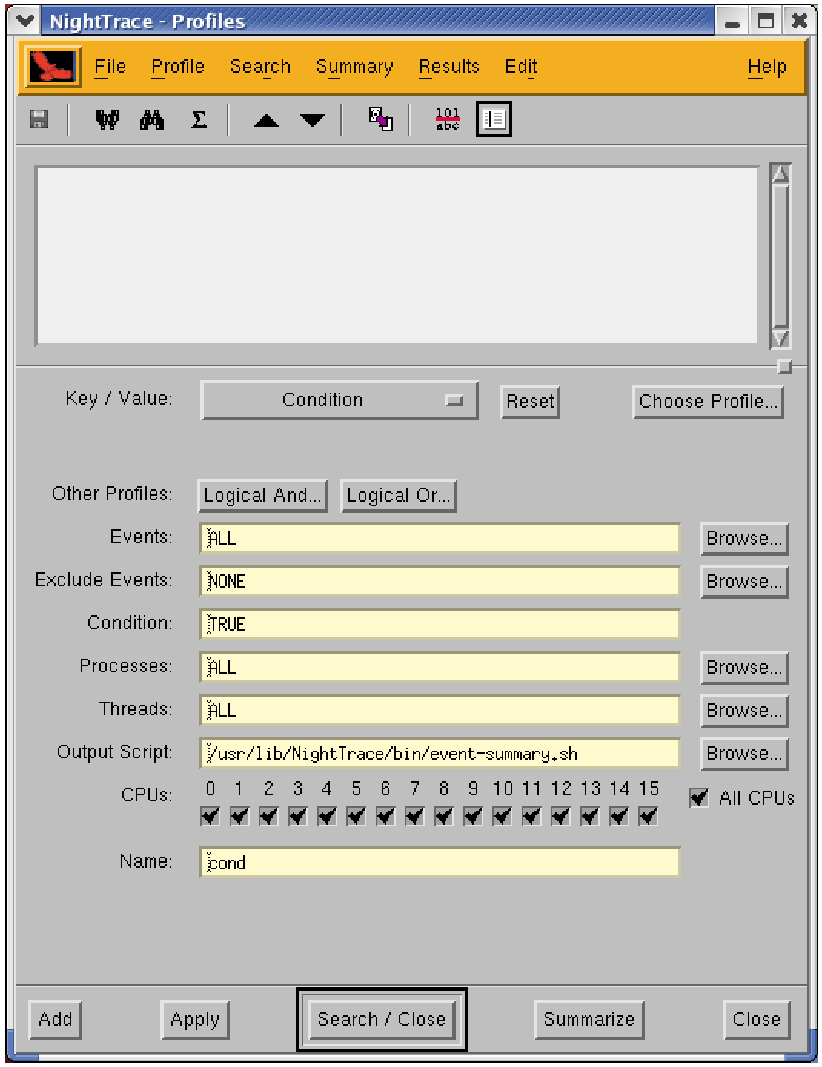


Figure 3-7. Searching using the Profiles dialog

- Enter:

  `cycle_end`

  in the Events text field.

- Enter:

  `arg1_dbl > 0.8`

  in the Condition text field.

- Enter:

  `obtuse`

  in the Name text field.

- Press the Search / Close button.

These actions have two effects:

1. A profile called `obtuse` is now defined and appears in a Profiles list in the NightTrace main window.
2. The current timeline was moved to the first event that matched the search criteria, that being the end of a cycle when the sine value exceeded 0.8.

Figure 3-8. NightTrace Main Window - obtuse profile

See that the Text Display Area indicates `arg1` with a value exceeding 0.8.

NOTE

It is possible that the search will fail if an insufficient number of events have been brought into live analysis. If this occurs, bring in more events using the Event list scroll bar and retry the search by pressing the forward search icon on the tool bar.

Halting the Daemon

Since the NightTrace portion of the tutorial is rather lengthy and may likely be a new experience for many users, we will halt the daemon to reduce memory usage.

Examine the daemon statistics in the Daemon Display Area. If the application has logged over 100,000 events, halt the daemon by pressing the Halt button to reduce memory usage as we slowly move through the NightTrace portion of the tutorial.

If it has not reached this stage yet, you may leave the daemon running and occasionally glance at the statistics. If NightTrace becomes unresponsive or slows down as the event counts reach into the millions, halt the daemon. NightTrace has a configurable memory consumption limit that will automatically halt the daemon when the limit is reached; a dialog will be presented informing the user when this occurs.

Using NightTrace Display Pages

When we initially launched the user daemon, NightTrace created a default user display page.

- Bring this display page to the foreground.

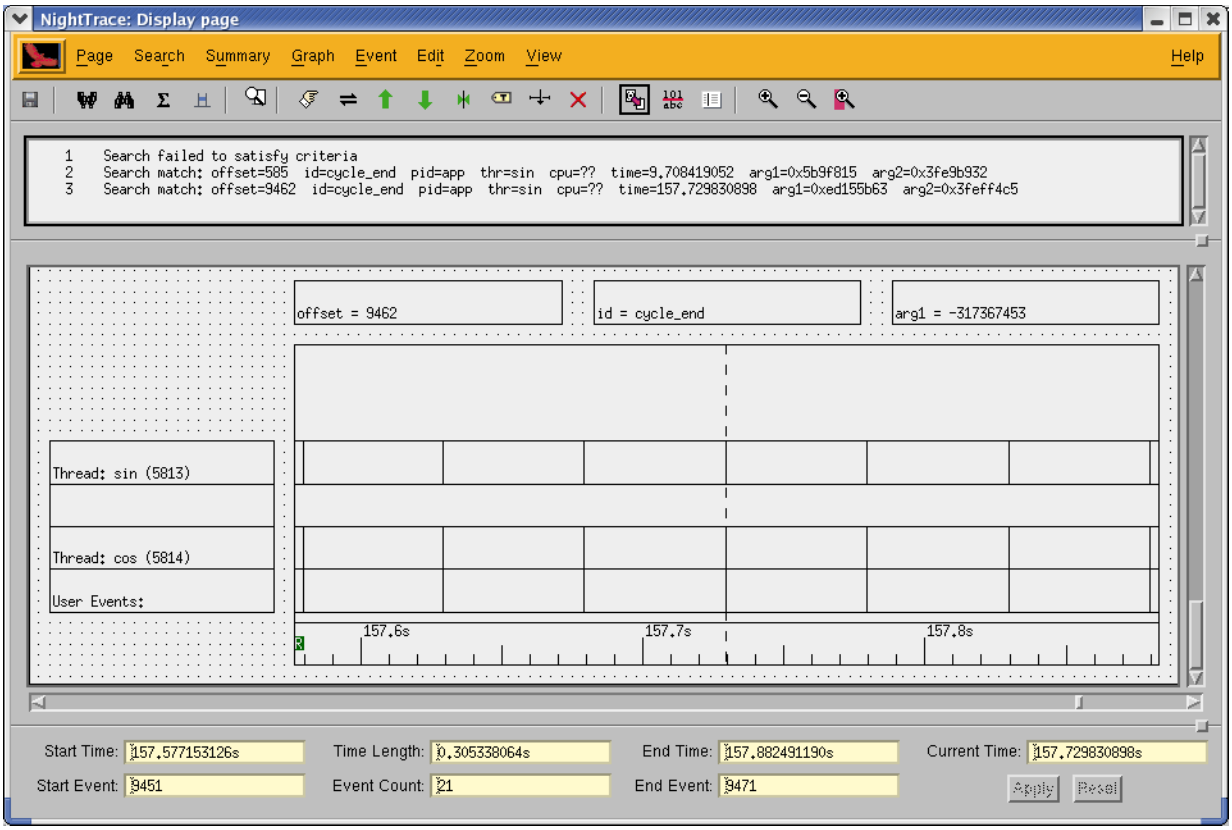


Figure 3-9. NightTrace Display Page

A display page is divided into five main areas:

1. menu bar
2. tool bar
3. text area
4. graphical display area
5. interval control area

The text area displays brief information about events that are results of searches or event-information requests. It also displays textual summary results as well as diagnostics.

The interval control area describes and defines the range of events shown in the display area. You can adjust these values as desired.

The display area contains static and dynamic labels and event and state graphs.

By default, NightTrace detects the threads that have registered themselves through NightTrace API calls and creates individual graphs for each thread. The user events graph shows events for all threads.

In “Using Tracepoints” on page 2-32 in the Using NightView section, we inserted tracepoints into the sine thread, which registered itself with the string “`sin`”.

Each vertical line in the graph represents at least one event. You can zoom in and zoom out to adjust the level of detail.

The vertical dashed line is the current timeline and is directly connected to the salmon-highlighted event in the NightTrace main window.

Left-clicking the mouse in the display area moves the current timeline. The three data boxes above the graphs change to reflect the event closest to the left of the current timeline.

Changing a Data Box Configuration

By default, the third data box displays the first argument for every event. We will change its configuration to tailor its display for our data set.

- Select the Edit Mode option from the Edit menu or click the Edit icon on the toolbar 

  This puts the window into *edit mode* and allows for configuration changes to the display window.

- Double-click on the third data box which contains text describing the value of `arg1`.

  A Data Box configuration dialog appears as shown below.

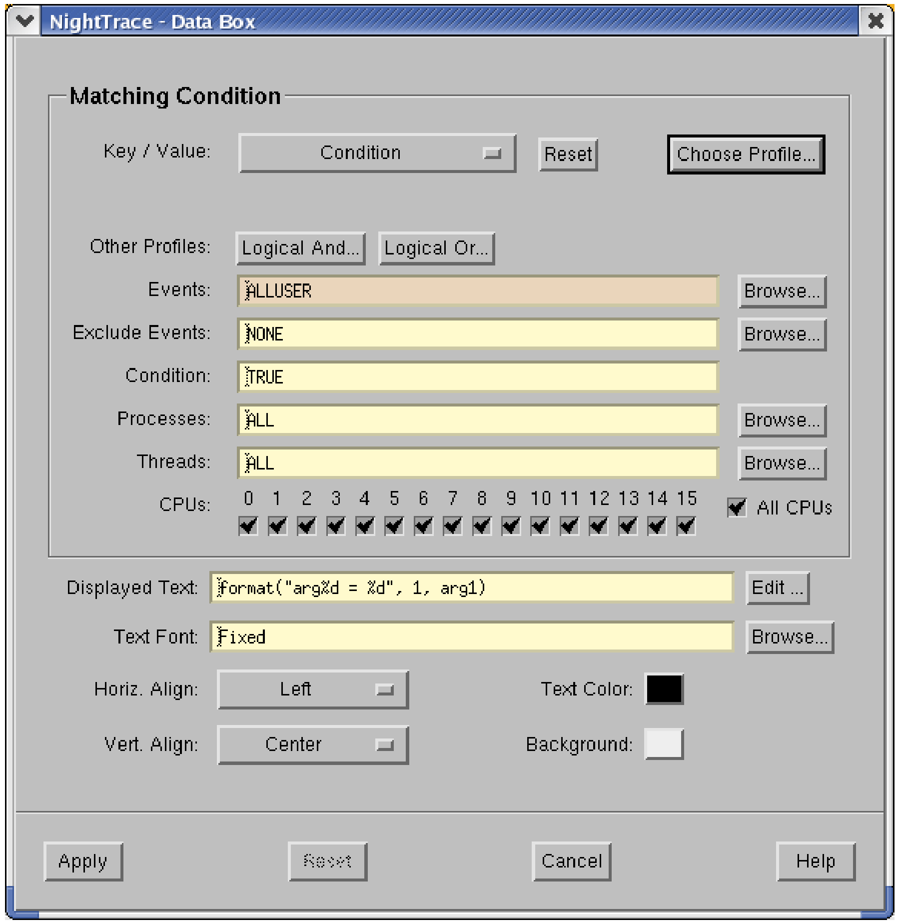


Figure 3-10. NightTrace Data Box dialog

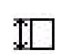
- Change the text in the Events field to:

  `cycle_end`

- Change the text in the Displayed Text field to:

  `format("sine value = %f", arg1_dbl)`

- Press the Apply button.

- Return to *view mode* by pressing the Edit icon on the toolbar 

The data box has been changed to describe only the `cycle_end` event and to properly display the sine value.

Configuring a State

In addition to event graphs, NightTrace can display states.

- Select the New Profile... option from the Edit menu or press the Profiles icon on the toolbar (sixth icon from the left).

  The Profiles dialog is displayed:

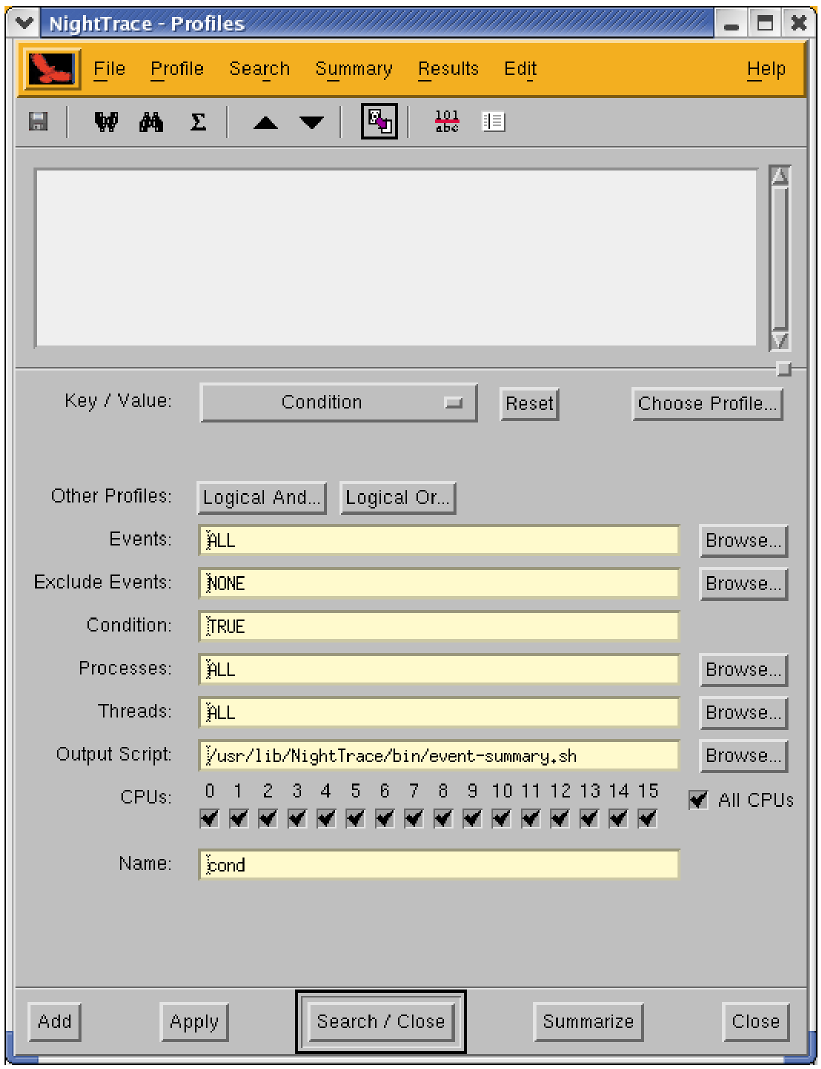


Figure 3-11. Profiles dialog

- Select State in the Key / Value option list.
- Enter:

  `cycle_start`

  in the Start Events text area

- Enter:

  `cycle_end`

  in the End Events text field.

- Enter:

  `sin`

  in the Threads text field.

- Enter:

  `sine`

  in the Name text field.

- Press the Add button.
- Press the Close button.

A state named `sine` has now been defined and occurrences can be displayed in the graphs in the display page.

- Enter *edit mode* by clicking the Edit icon on the toolbar.
- Double-click on the state graph assocated with the row labelled “Thread: sin”

  The State Graph configuration dialog is displayed as shown below:

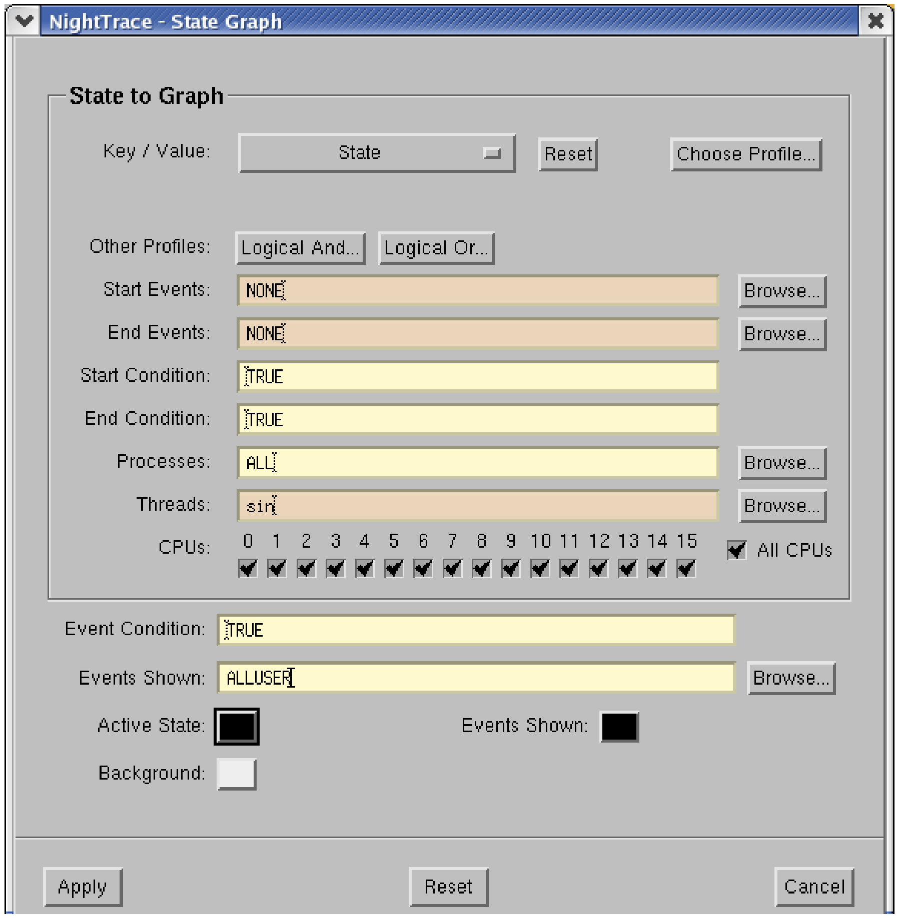


Figure 3-12. NightTrace State Graph dialog

- Select State from the Key / Value option list.
- Press the Choose Profile... button.
- Select the `sine` state from the list
- Make sure the Import Reference to Profile checkbox is checked
- Press Select.
- Press the black box to the right of the Active State label.
- Use the slide bars in the Choose Color dialog to select a color and press OK.
- Press the Apply button.
- Return to *view mode* by pressing the Edit icon on the toolbar.

The graph has now been configured to display the sine state as a solid bar in the lower portion of the state graph. Events will still be displayed as vertical black lines that extend over the entire vertical height of the graph.

It is likely that the display page has not changed in a significant way.

This is because the `cycle_start` and `cycle_end` events occur so close together in time that you cannot distinguish them at the current zoom setting.

- Click in the middle of the state graph.
- Zoom in using the **Zoom In** icon on the toolbar or the **Down Arrow** key until the two events can be distinguished and a state bar is shown.

  You may need to readjust the current timeline as you zoom in.

NOTE

If the Down Arrow key has no effect, press the Num Lock key and try again.

The figure below displays an instance of the sine state:

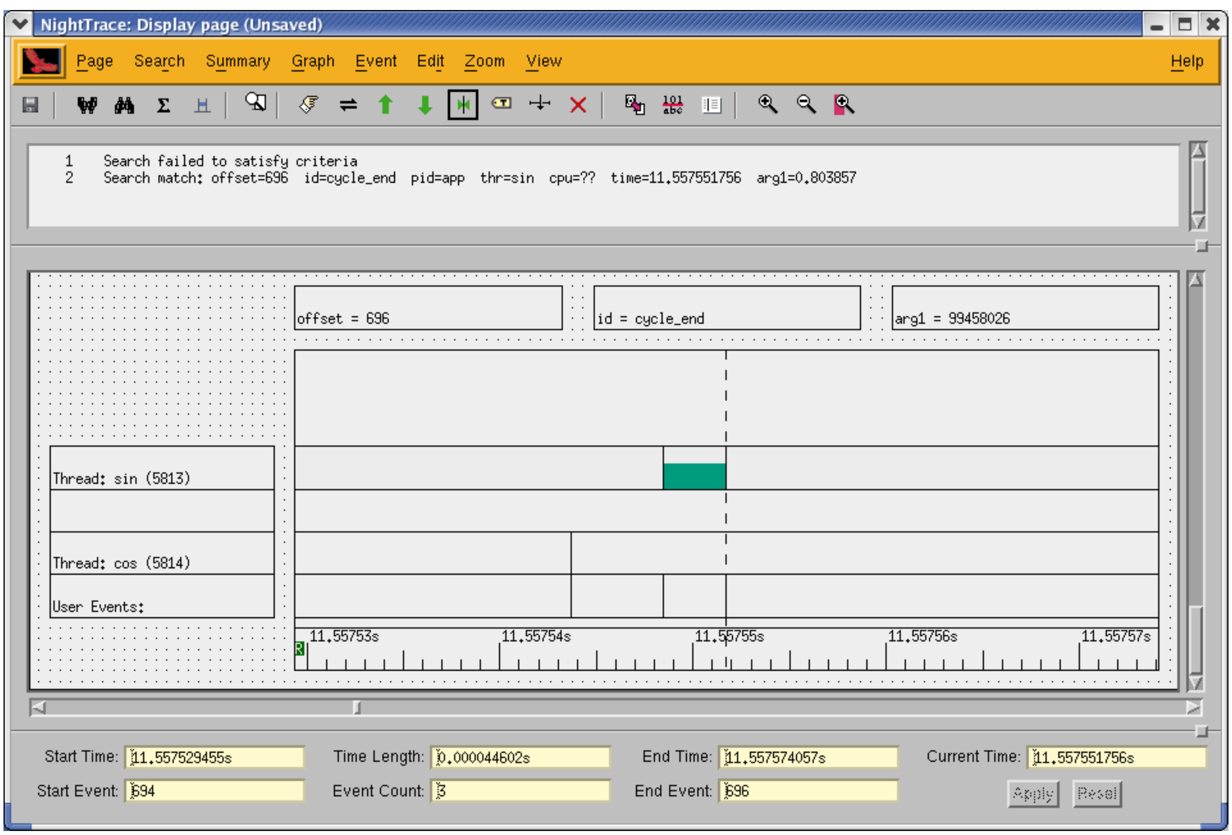


Figure 3-13. Display Page - sine state graph

Displaying State Duration

The duration of the most recently completed state can be displayed via a data box.

- Enter *edit mode* by pressing the Edit icon on the toolbar.
- Select the Data Box option from the Graph menu.

  The cursor will turn into a + character.

- Using the left mouse button, click an area in the display page on the grid (outside of any currently displayed graph or data box -- i.e. only on an available area whose background shows the dotted grid) and drag the mouse to create the outline of the new data box
- Double-click the data box.
- Enter the following into the Displayed Text field:

  `format ("cycle = %f ms", state_dur(sine)*1000.0)`

- Press the Apply button.
- Enter *view mode* by pressing the Edit icon on the toolbar.

The data box now displays the length of the most recently completed instance of the sine state in seconds.

Using the Summary Dialog

In addition to obtaining detailed information about specific events and states, summary information is easily generated.

- Select the Summary... menu item from the Summary menu.
- Use the solid black up and down arrows in the toolbar to select the profile matching the `sine` state

  It is likely that the `sine` profile is already selected. Check the name text area near the bottom of the dialog

- Press the Summarize button.

  A textual summary is displayed in the text area at the top of the Profiles dialog and a State Matches dialog is launched.

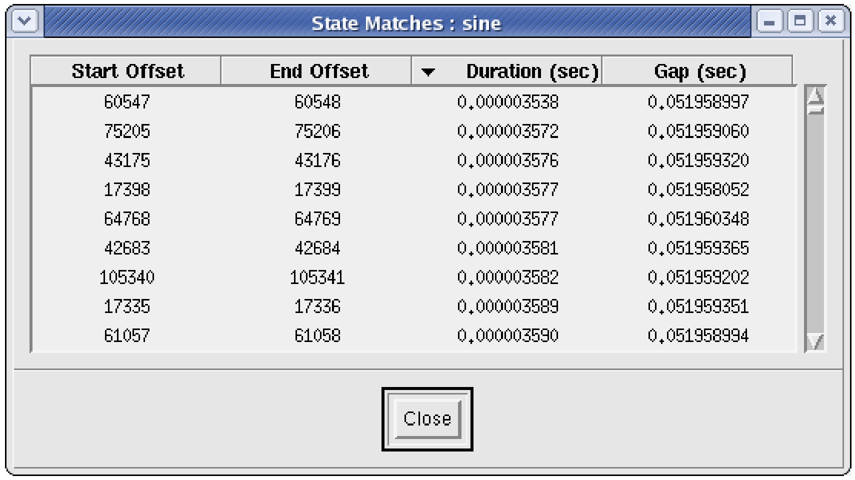


Figure 3-14. Profiles dialog - State Matches dialog

The State Matches dialog provides four columns of information: the state's starting and ending offsets, the state's duration, and the gap between a state an its more recent previous occurrence. You can click on the column headers to control how the list is sorted.

Double-clicking on a row in the list positions the current timeline to the beginning of that instance of the state.

- Press the Close button to close the State Matches dialog.

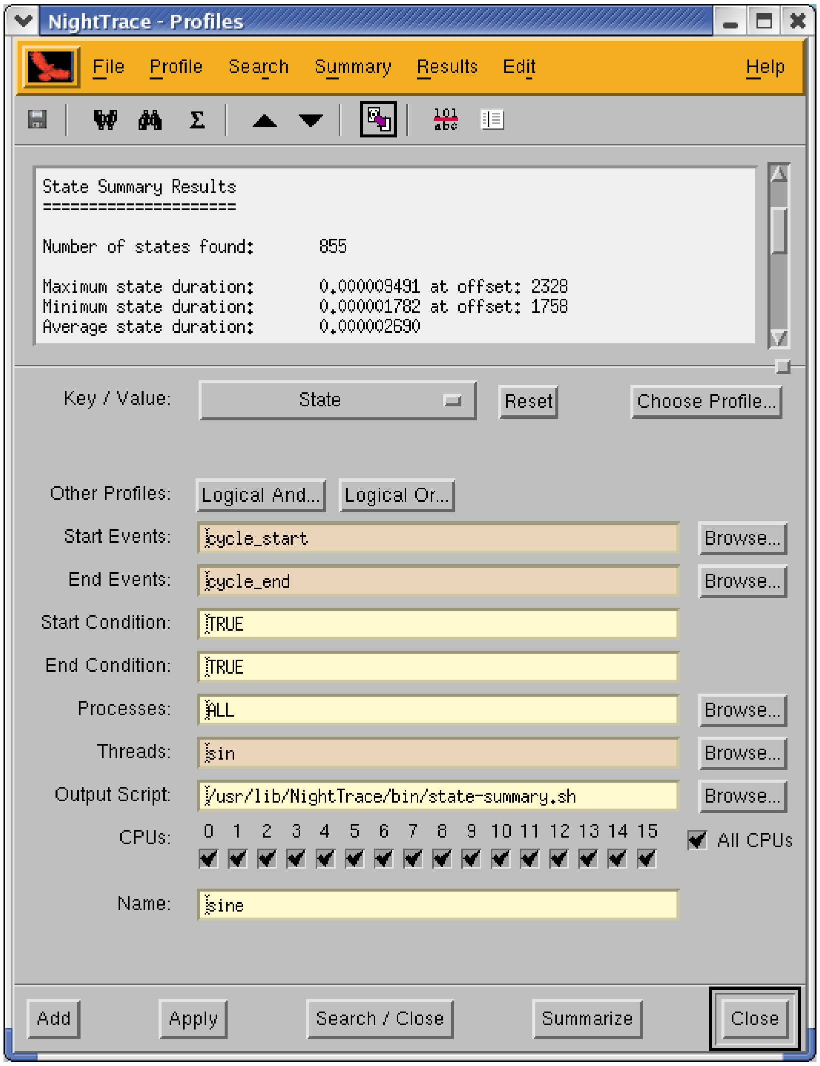


Figure 3-15. Profiles dialog - Summary results

- In the Profiles, use the scroll bar to see the summary results and ensure that the line with:

  ```
  Minimum state duration
  ```

  is displayed.

- Double-click the number representing the event offset of the minimum state duration.

The display page is automatically changed to place the current timeline at the end of the state associated with that offset.

The minimum and maximum state occurrences are often of interest. However, a graphical display of state durations can be more enlightening.

- Select the Options... item from the Summary menu in the Profiles dialog.
- Select the Durations item from the State Summary Graph option list.
- Select the Scroll to longest duration item from the State Summary Action option list.
- Press the Apply button.
- Press the Summarize button.
- Press the Close button on the Profiles dialog.

A customized display page is created which summarizes the `sine` state.

The display includes a data graph with vertical lines representing the value of the duration of each instance of the state.

A narrow state graph is displayed directly above the data graph.

- Zoom out all the way to display all of the instances of the `sine` state by pressing Alt+Up Arrow.

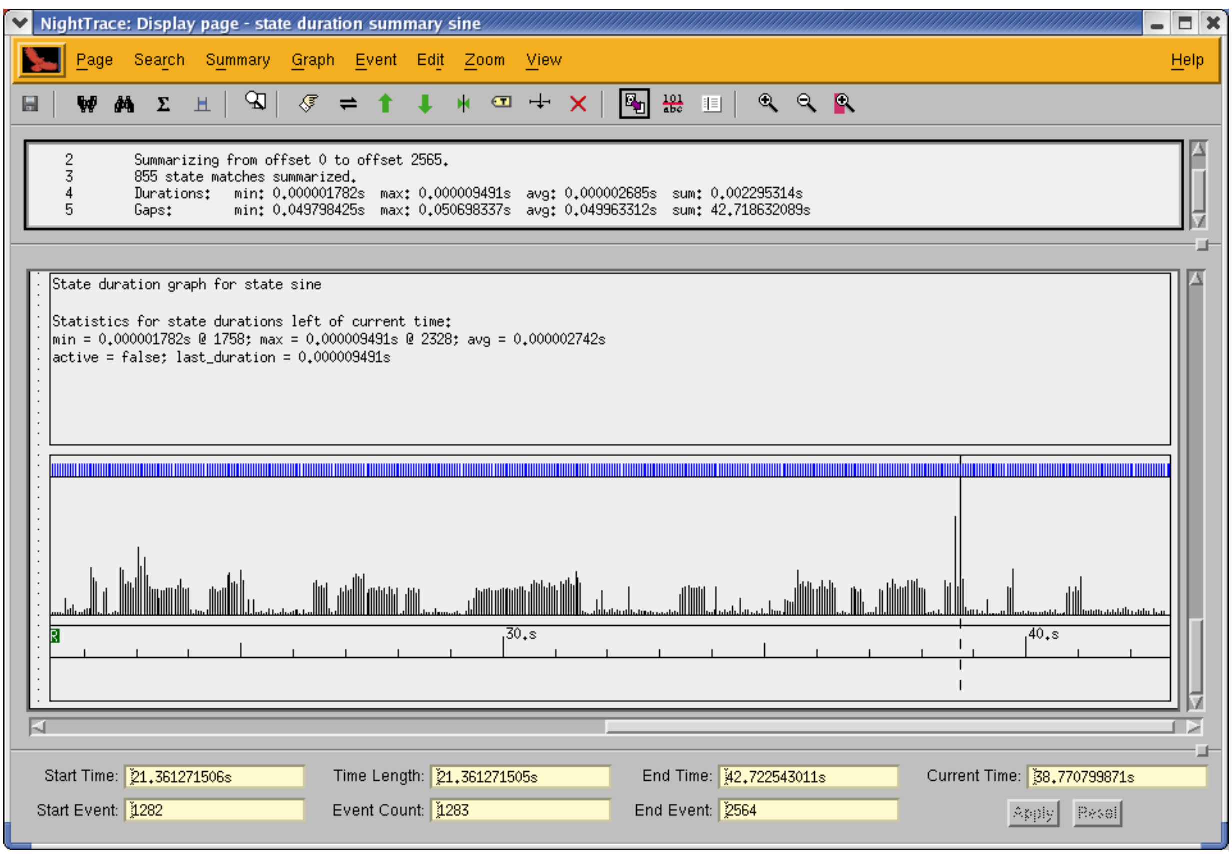


Figure 3-16. Display Page - sine state summary

Depending on the actual variations in state duration, most of the state durations may appear as tiny vertical lines.

- Enter *edit mode* by pressing the Edit icon on the toolbar.
- Double-click in the middle of the data graph.
- Change the Max Graph Value text field to a value about 10 times the average duration listed in the text area above the graph.
- Press the Apply button.
- Enter *view mode* by pressing the Edit icon on the toolbar

Values exceeding the maximum value set in the dialog will appear as vertical lines spanning the entire height of the data graph, but smaller durations are graphed according to their actual value.

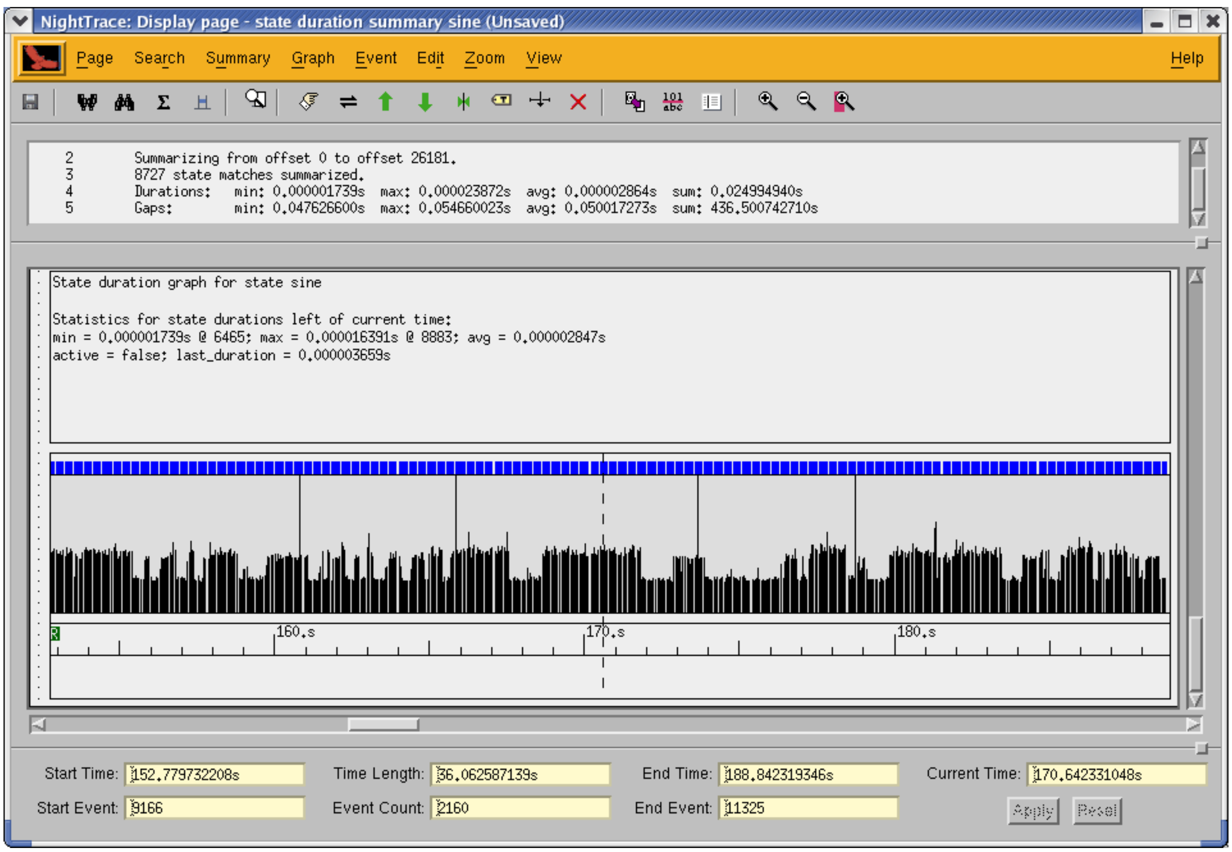


Figure 3-17. Display Page - sine state summary - adjusted

All display pages are linked together with the current timeline. You can easily identify spikes of interest using the summary graph, click to change the current timeline to that location, then switch to another display page for more analysis of the new location.

- Select the Close menu item from the Page menu in the display page containing the state summary.

NOTE

A warning dialog may appear indicating the changes to the summary graph display page will be lost. Press OK.

Defining a Data Graph

The page displaying user events has a blank area in the main display box (which is called a *column*).

- Enter *edit mode* by clicking the Edit icon on the toolbar.
- Select the Data Graph menu item from the Graph menu.

  The cursor changes to a + character.
- Using the left mouse button, click inside the column near the upper left-hand corner and drag the mouse downward and release just before reaching the top of the first event graph.

NOTE

Make sure that you click inside the column and not on the lines defining the border of the column

- Double-click in the middle of the data graph you just inserted.
- Enter:

  `cycle_end`

  in the Events text field.
- Enter:

  `arg1_dbl`

  in the Graph Value text field.
- Click on the black box to the right of the Fill Color label to select a color for the data graph. Click OK to close the Choose Color dialog.
- Press the Apply button to close the Data Graph dialog.
- Return to *view mode* by pressing the Edit icon on the toolbar.
- Zoom the display to see the sine wave generated by the program.

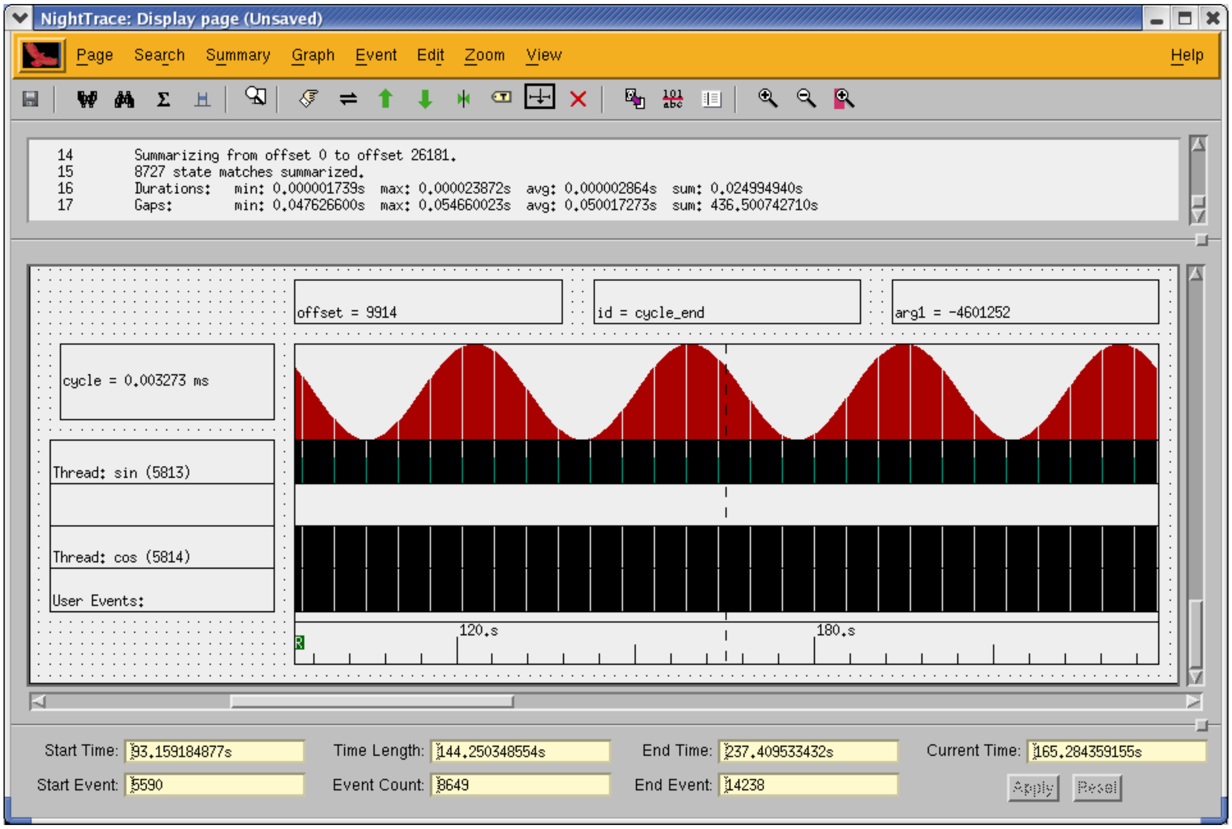


Figure 3-18. Display Page - sine wave graph

Kernel Tracing

Kernel tracing provides amazing insight into the activities of the system and how applications interact with each other and the kernel.

In order to use kernel tracing you must be running a trace-enabled kernel.

By default, three RedHawk kernels are installed on RedHawk systems. Kernels names ending in **`-trace`** and **`-debug`** have kernel tracing enabled. You may check to see which kernel is running by using the following command:

```
uname -r
```

If you are not running a trace-enabled kernel, reboot now and select it from the GRUB menu at boot time. If you are unable to reboot your system at this time, please follow the tutorial and load the pre-recorded kernel data as instructed.

- Minimize the user display page.
- Ensure the user daemon is stopped by pressing Halt on the NightTrace main window.
- Select the `daemon_1` segment in the Trace Segment Area of the NightTrace main window.
- Select the Close Trace Segments option from the NightTrace menu.

NOTE

If the trace segment was not removed it is likely that you selected the `daemon_1` line from the Daemon Definition Area and not the Trace Segment Area which is above it.

Obtaining Kernel Trace Data

If you are not running a trace-enabled kernel, skip this section and refer to the section Using Prerecorded Kernel Data.

- Double-click on the `daemon_0` entry in the daemon list in the NightTrace main window.
- Check the Buffer Wrap checkbox on the Daemon Definition dialog.
- Press OK.

The kernel daemon is now configured to run in bufferwrap mode. This means that kernel events are collected in kernel memory buffers and are not passed to NightTrace except by explicit flush operations.

Depending on system activity, huge amounts of kernel trace data can be generated in a relatively short period of time. Since operation of NightTrace is likely a new experience for many users, we will restrict the data flow to a managable size for new users.

- Ensure that `daemon_0` is selected in the Daemon Display Area.
- Press the Launch button.
- Press the Resume button
- Watch the daemon statistics in the Daemon Display Area; once 50,000 events or so are present in the Buffer column, press the Flush button.

Skip the next section and jump directly to “Analyzing Kernel Data” on page 3-26.

Using Prerecorded Kernel Data

This section is provided only for those using the tutorial that have not booted a trace-enabled kernel.

If you collected live kernel trace data in the preceding section, skip to Analyzing Kernel Data.

The NightStar RT **tutorial** directory contains some pre-recorded kernel data which can be used in the section titled “Analyzing Kernel Data” on page 3-26.

- Select the Open Trace File... menu item from the NightTrace menu in the NightTrace main window.
- Type the following into the file dialog in the Selection text field:

  /usr/lib/NightStar-RT/tutorial/.kernel-data

- Press the OK button.

Proceed to the next section.

Analyzing Kernel Data

NightTrace automatically generates a default kernel display page that is customized to the system from which the kernel data was captured.

- Resize the kernel display page so that information for all CPUs can be seen.
- Zoom out until the data and state graphs are populated with events.
- Click in an active area and zoom in until detail can be seen.

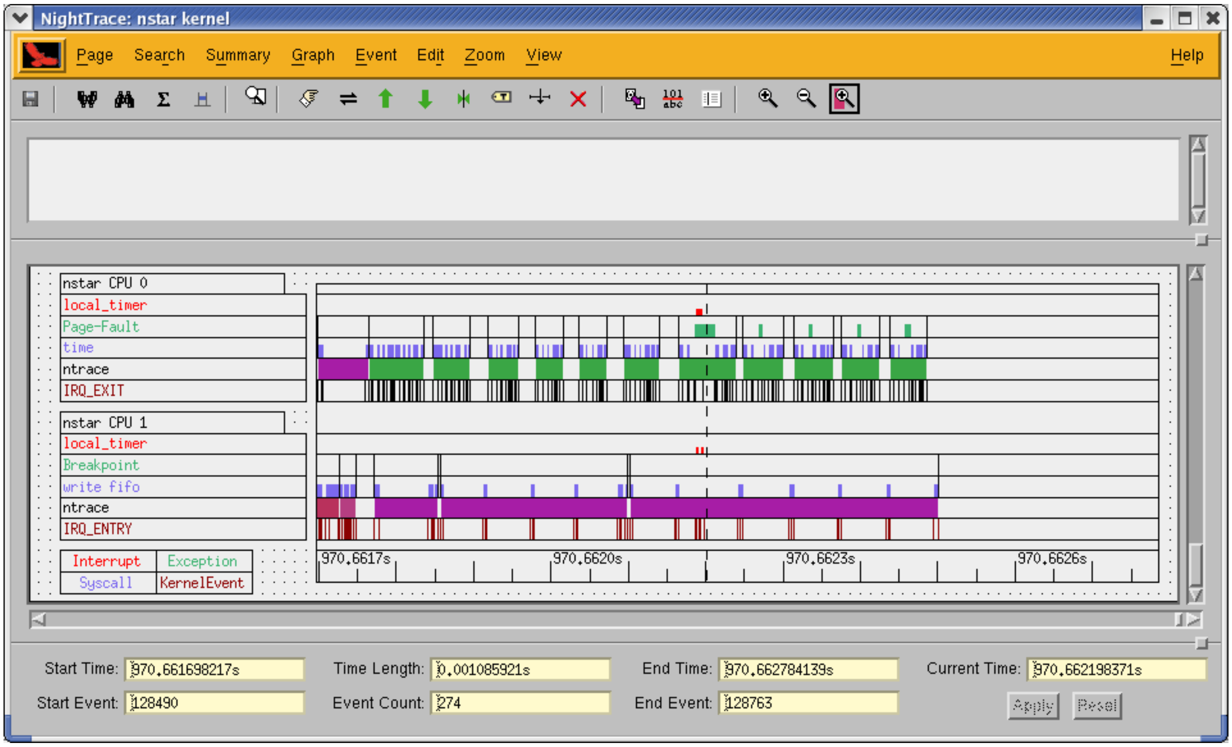


Figure 3-19. Kernel Display Page

NOTE

> Your display page may look significantly different if you have a different number of CPUs. Additional system activity can make the display vary as well. Do not be concerned about such differences at this step.

For each CPU, the following information is displayed:

- interrupt activity
- machine exception activity
- system call activity
- per-process CPU utilization
- detailed kernel events

The data boxes on the left hand side of the display page are color coded to match the information they describe. Their contents change dynamically based on the position of the current timeline.

- Press `Ctrl+F` to open a Profiles/Search dialog.
- Press the `Browse...` button to the right of the `Processes` text field.

- Select the **app** process from the list of known processes.
- Press the Select button to close the process list.
- Select the System Call Enter Events option from the Key / Value option list.
- Select `nanosleep` from the list of system calls shown.
- Change the list of events in the Events text field to include only `SYSCALL_RESUME`
- Press the Select button to close the system call list.
- Press the Search / Close button.

The current timeline is changed to the next occurrence of the `nanosleep` system call in process **app**.

Zoom in until detailed information is visible, similar to what is shown below:

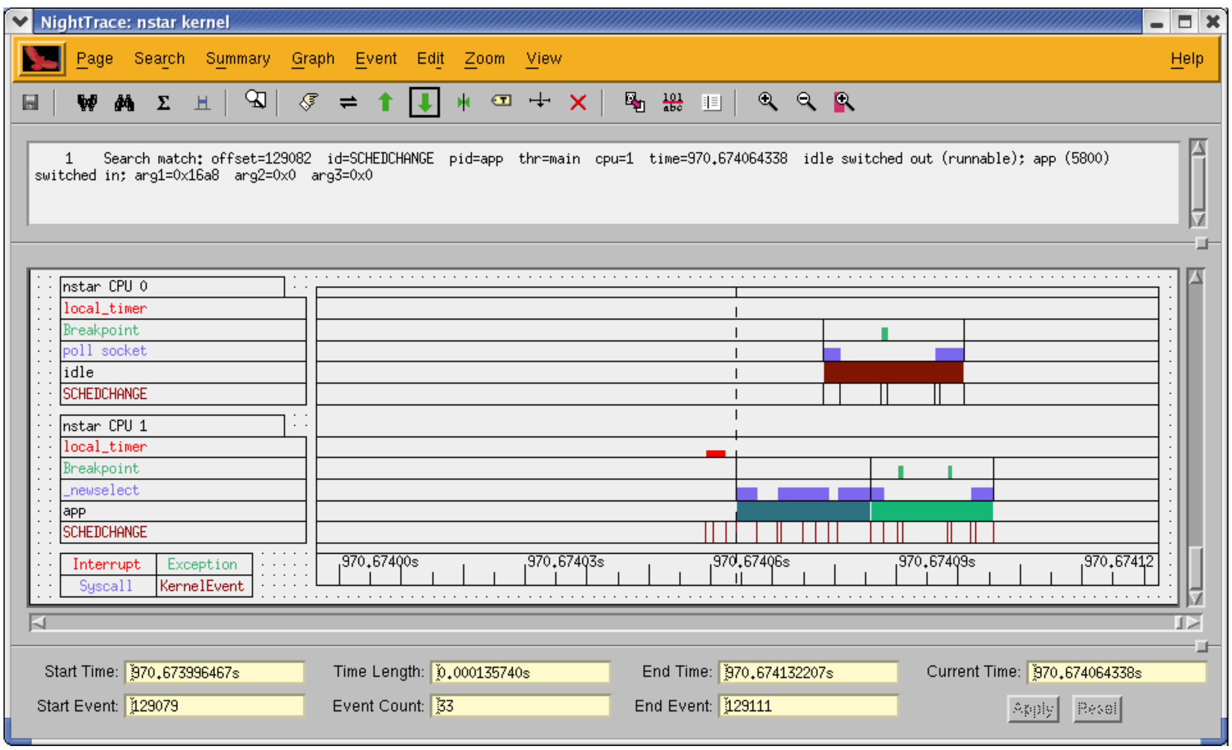


Figure 3-20. Kernel Display Page

NOTE

> Your display page may look significantly different if you have a different number of CPUs. Additional system activity can make the display vary as well. Repeat the search a few times to find an occurrence that looks similar to the row which indicates the **`app`** process. You can repeat the last search by pressing the forward search icon on the tool bar or by pressing the > key (no shift is required).

The red bar to the left of the current timeline indicates that an interrupt occurred. In this case, it was a `local_timer` interrupt.

The tall vertical black line represents a context switch.

- Raise the NightTrace main window.

NOTE

> You can raise the NightTrace main window from any display page by using Alt+P, M keystrokes. Alternatively you can select the NightTrace Main Window... option from the Page menu of any display page.

Look at the Events list.

The salmon-shaded event is the event at the current timeline, which should be `SYSCALL_RESUME`.

- Select that event

The text area of the Main window describes the event in more detail, in this case, text similar to the following:

```
Offset  : 3581
Detail  : Resuming system call nanosleep
Args     : arg1=0 arg2=162 arg3=0
```

- While the `SYSCALL_RESUME` event is selected, press the Up Arrow key

The current timeline is changed to the preceding event and the text description indicates a context switch with text similar to the following:

```
Offset : 3580
Detail  : idle switched out (runnable); app (5336) switched in
Args    : arg1=0x14d8 arg2=0x0 arg3=0x0
```

- Raise the kernel display page

The blue bar represents sysem call activity. The data box to the left will describe the system call name for the system call at or to the left of the current time line.

- Press Shift+Period key to advance back to the `SYSCALL_RESUME` event

In this case, the main thread is exiting the `nanosleep` call on line 92 of **`app.c`**. It then enters an `ipc` system call to execute the `semop` library call on line 93.

In the example shown above, one of the other threads shortly wakes up and begins to execute and then blocks again in an `ipc` system call.

Mixing Kernel and User Data

If you are not running a trace-enabled kernel, skip this section.

- Raise the NightTrace main window.
- Press the Halt button to stop the kernel daemon.
- Double-click the `daemon_0` line in the Daemon Display Area to edit the daemon definition.
- Clear the Buffer Wrap checkbox.
- Press the OK button to close the dialog.
- Select both daemons in the Daemon Display Area using by holding Shift while selecting them.
- Press the Launch button.

Read the next four steps before proceeding, then execute them in order.

- Press the Resume button.
- Wait about 2 seconds.
- Press the Flush button.
- Press the Halt button.

Data from both the user application and the kernel have been captured and brought into NightTrace.

- Select the `sine` profile from the Profiles list at the top of the NightTrace main window.

  You may need to scroll the list of profiles to locate `sine`.
- Press the Summary icon on the toolbar (the fifth icon from the left).

The last action caused a summary of the `sine` state defined in "Using the Summary Dialog" on page 3-18.

The current timeline was automatically positioned to the longest instance of the state.

- Raise the kernel display page.
- Zoom in until you can clearly see the detail relating to the sine thread's cycle.

In the graphic shown below, the sine thread was preempted by a kernel processing of an `i8042` interrupt.

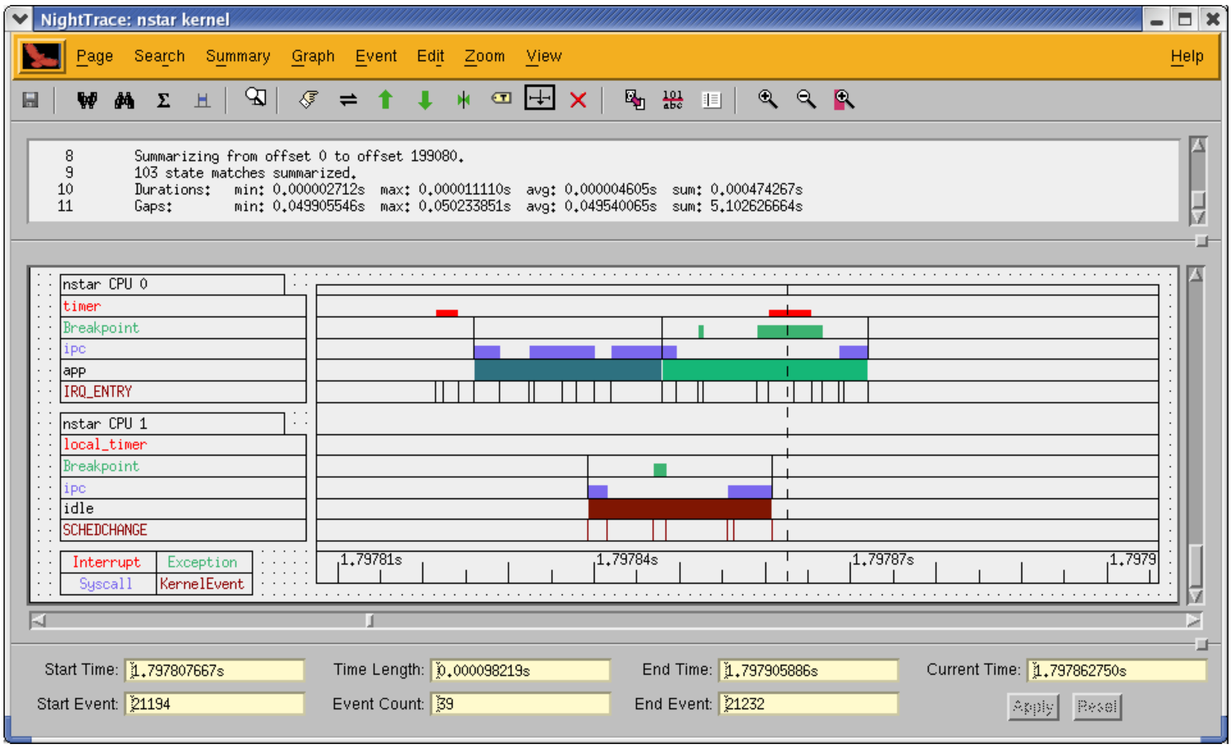


Figure 3-21. Kernel Display Page

The reason for the extended cycle in your trace data may be due to other circumstances.

- Was the `sine_thread()` preempted by another process?
- Did an interrupt occur during the cycle?
- Was there signifcant activity on the hyper-threaded sibling CPU which stole cycles from the CPU where the sine thread was executing?
- Did the application get a page fault or other machine exception?

Machine exceptions include information detailing the type of exception, the faulting address (when applicable), and the PC at which the exception occurred.

- Type Ctrl+F while the kernel display page is selected.
- Select Exception All Events from the Key / Value option list.
- Select `Page-Fault` from the list of exceptions.
- Press the Select button.
- Press the Search / Close button.

The current timeline is moved to the next occurrence of a page fault. The text area at the top of the kernel display page includes detailed information about the exception, including the PC at which the fault occurred and the faulting address.

You can use NightView to see the actual line number of programs (if they have debugging information) based on the PC information.

Using the NightTrace Analysis API

NightTrace provides a powerful API which allows user applications to analyze pre-recorded trace data or to monitor and analyze live trace data.

Users can write programs that defines states and conditions and process events as they occur.

In this tutorial, we will instruct NightTrace to build an API program automatically.

- Raise the NightTrace main window.
- Select the `sine` profile from the Profiles list.
- Select the Export... menu item from the Profiles menu.

The following dialog is displayed:

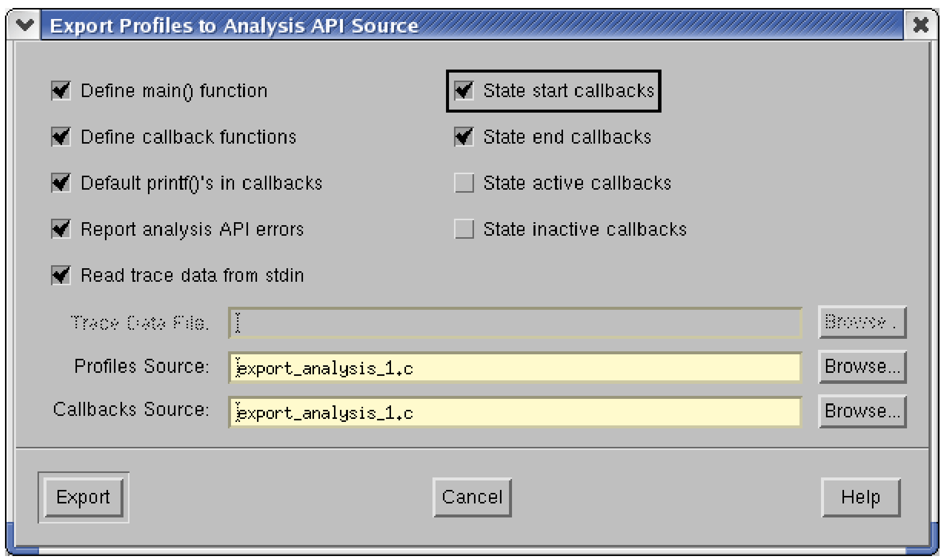


Figure 3-22. Export Profiles to Analysis API Source dialog

- Clear the State start callbacks checkbox.
- Press the Export button.
- Select the Exit Immediately menu item from the NightTrace menu to exit NightTrace.

NightTrace has created an API program which listens for occurrences of the state defined by the **`sine`** profile and prints out some information for each instance.

- Build the API program using the following command:

```
cc -g export_analysis_0.c -lntrace_analysis
```

This program expects to consume live trace data.

You can configure a user daemon with the NightTrace GUI and have NightTrace launch the analysis program automatically.

Alternatively, you can use the command line user daemon program **ntraceud** to achieve the same effect.

- Type the following command:

```
ntraceud --stream --join /tmp/data | ./a.out
```

This command instructs **ntraceud** to start capturing trace data from a running application which is using the file **/tmp/data** as a handle. The **--stream** option indicates that instead of logging the data to the named file, it should be sent to **stdout**.

The application program may not immediately begin generating output because the data rate is fairly low and buffering is involved.

- To flush the current buffers for immediate consumption by the applicaiton, issue the following command in a different terminal session:

```
ntraceud --flush /tmp/data
```

NOTE

You may need to repeat that command several times over a period of a few seconds to allow the data to pass through system buffers.

Data similar to the following will appear on **stdout** in the terminal session where the analysis program was launched:

```
sine (end)offset 665 occur 333 code 2 pid 3399 time 16.628649 duration 0.000003
sine (end)offset 667 occur 334 code 2 pid 3399 time 16.678631 duration 0.000003
sine (end)offset 669 occur 335 code 2 pid 3399 time 16.728655 duration 0.000003
sine (end)offset 671 occur 336 code 2 pid 3399 time 16.778676 duration 0.000003
sine (end)offset 673 occur 337 code 2 pid 3399 time 16.828693 duration 0.000003
sine (end)offset 675 occur 338 code 2 pid 3399 time 16.878716 duration 0.000004
sine (end)offset 677 occur 339 code 2 pid 3399 time 16.928745 duration 0.000003
sine (end)offset 679 occur 340 code 2 pid 3399 time 16.978760 duration 0.000003
sine (end)offset 681 occur 341 code 2 pid 3399 time 17.028779 duration 0.000003
```

- Issue the following command to terminate the daemon:

```
ntraceud --quit-now /tmp/data
```

If you are not running a trace-enabled kernel daemon, skip the remaining of this section.

Several sample API programs are provided with NightTrace.

- Type the following commands to build the watchdog example program:

```
cp /usr/lib/NightTrace/examples/watchdog.c .
cc -g -o watchdog watchdog.c -lntrace_analysis
```

This simple sample program watches for context switches on a specific CPU and prints the name of the process that is switching in.

This time the **ntracekd** kernel daemon will be used to capture 5 seconds of kernel data and stream the output to the **watchdog** program.

- Issue the following command:

```
ntracekd --stream --wait=5 /tmp/x | ./watchdog 1
```

The program will generate output similar to the following:

```
context switch: 4.979350027     4 ksoftirqd/0
context switch: 4.979358275  2846 X
context switch: 4.983906074     0 idle
context switch: 4.983960385  2846 X
context switch: 4.994892976  3167 firefox-bin
context switch: 4.994989171  4492 ntfilterl
context switch: 4.995070736  4489 watchdog
context switch: 4.995092415  4492 ntfilterl
context switch: 4.995173214  4489 watchdog
context switch: 4.995188096  4492 ntfilterl
context switch: 4.995256175  4489 watchdog
context switch: 4.995270824  4492 ntfilterl
context switch: 4.995332743  4489 watchdog
context switch: 4.995355783  2846 X
context switch: 5.000351519     4 ksoftirqd/0
context switch: 5.000360675  2846 X
```

Conculsion - NightTrace

This concludes the NightTrace portion of the NightTrace RT User’s Guide.

4
Using NightProbe

NightProbe is a graphical tool for viewing and modifying data from independently executing programs as well as recording data for subsequent analysis.

This chapter assumes you have already built the **app** program and it is running under the control of NightView. If you have not built the program, do so using the instructions in “Building the Program” on page 1-4 and execute the application via the following command before proceeding:

```
./app
```

Invoking NightProbe

Programs to be probed do not need to be instrumented with any special API calls. However, in order for NightProbe to refer to symbolic variable names, program should be compiled with debug information (typically the **-g** compilation option).

NightProbe takes advantage of significant performance capabilities of the RedHawk kernel, making intrusion on the process almost undetectable. NightProbe samples and modifies variables in other programs using direct memory fetches and stores.

Invoke NightProbe using the NightProbe desktop icon or type the following command:

```
nprobe &
```

The NightProbe Main window is displayed.

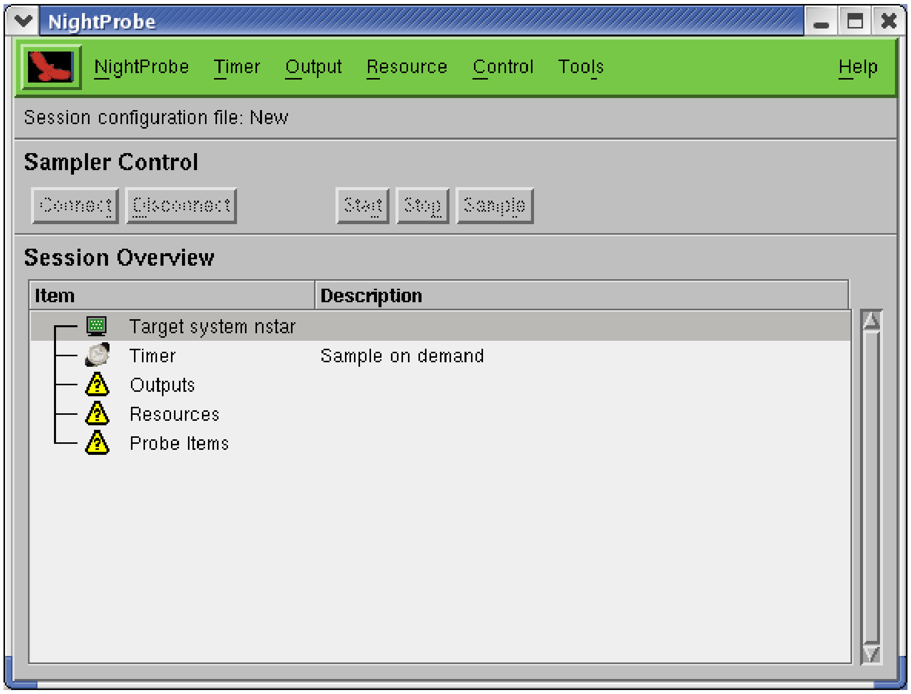


Figure 4-1. NightProbe Main Window

Selecting Processes and Variables

NightProbe has the ability to probe several kinds of resources, including programs, shared memory segments, and other memory mapped entities.

- Right-click the Resources icon in the Session Overview area and select the Add Program... option.

The Program Window selection dialog is shown

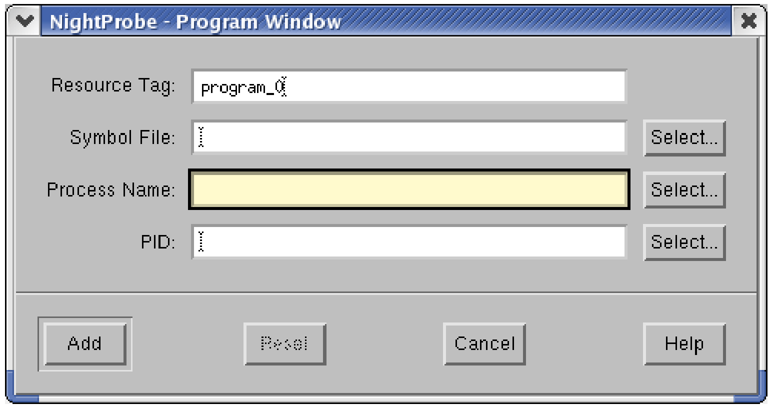


Figure 4-2. NightProbe Program Window

- Press the Select... button to the right of the PID row

The Select Process ID dialog will appear.

- Scroll the display to locate the **app** program and select it
- Press the Select button

The process ID associated with the **app** program has been placed in the PID text field and the Process Name and Symbol File text fields have been updated.

- If the text in the Symbol File text field is a relative pathname, change the text to the full pathname of the program)
- Press the Add button

The **app** program has been added to the list of resources to be probed.

- Right-click the app icon in the Resources list and select Add item from program... menu option

The Item Browser dialog is displayed.

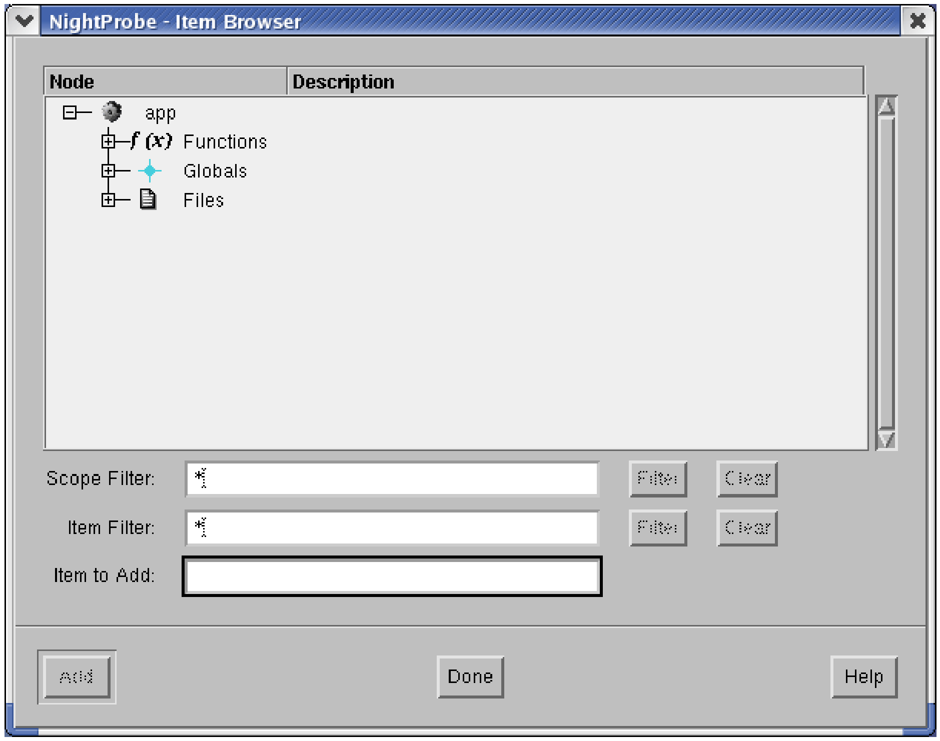


Figure 4-3. NightProbe Item Browser Window

- Expand the list of global variables by pressing the + icon to the left of the Globals label

The list of global variables in the **app** program are displayed. The `data` variable is a composite object and can be expanded.

- Expand the `data` variable by pressing the + icon to the left of the data label
- Expand both structures displayed, `data[0]` and `data[1]`
- Double-click the `angle`, `count`, and `value` fields from both `data[0]` and `data[1]` structures
- Double-click the `rate` variable
- Press the Done button

Double-clicking an item causes the color to turn reddish-orange and adds it to the list of program items to be probed.

The Main window should now include the selected variables as shown below:

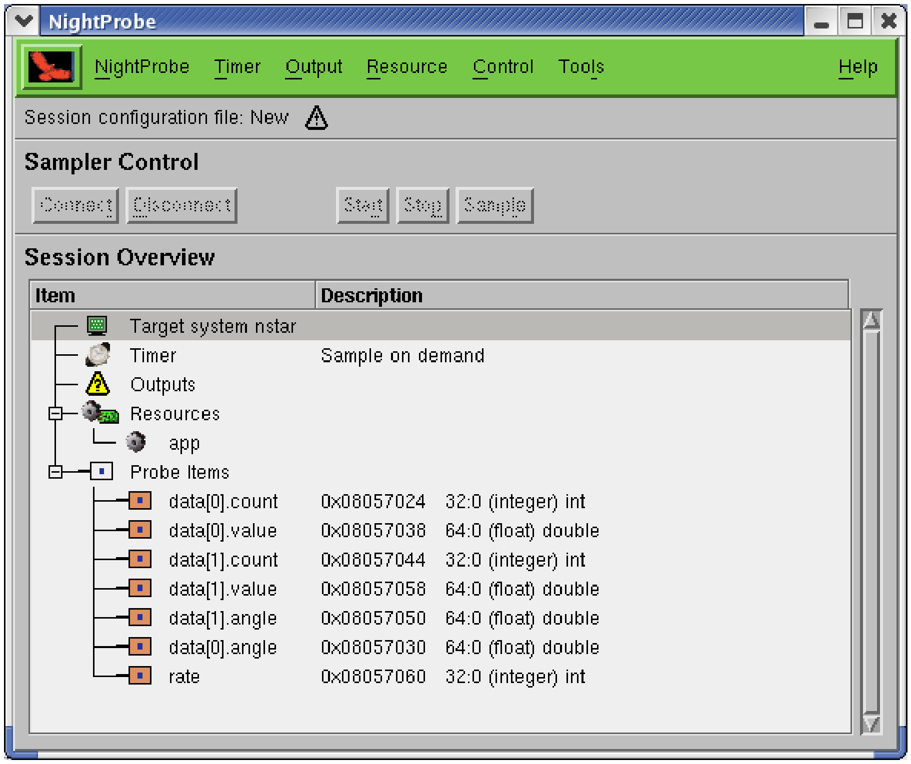


Figure 4-4. NightProbe Main Window with selected items

If the seven items shown above are not in your list, repeat the steps above to add them.

Selection of Outputs

NightProbe provides various output methods for the probed data.

File Output

The data samples taken by NightProbe may be written to a file for subsequent analysis.

- Right-click the Outputs icon in the Session Overview area and select the File Output... menu option

A File Output file selection dialog is presented.

- Type in the following in the Output File text area

/tmp/nprobe-data

- Press the Add button

List Window Output

A List Window is a simple listing dialog which shows the values of each data sample along with the individual data item names.

- Right-click the Outputs icon in the Session Overview area and select the List Window Output menu option

A List Window is displayed.

Spreadsheet Output

Data values may be displayed in a simplified spreadsheet window which provides for customized placement and formatting as well as for modification of variable values.

- Right-click the Outputs icon in the Session Overview area and select the Spreadsheet Output menu option

The Spreadsheet Viewer window is displayed.

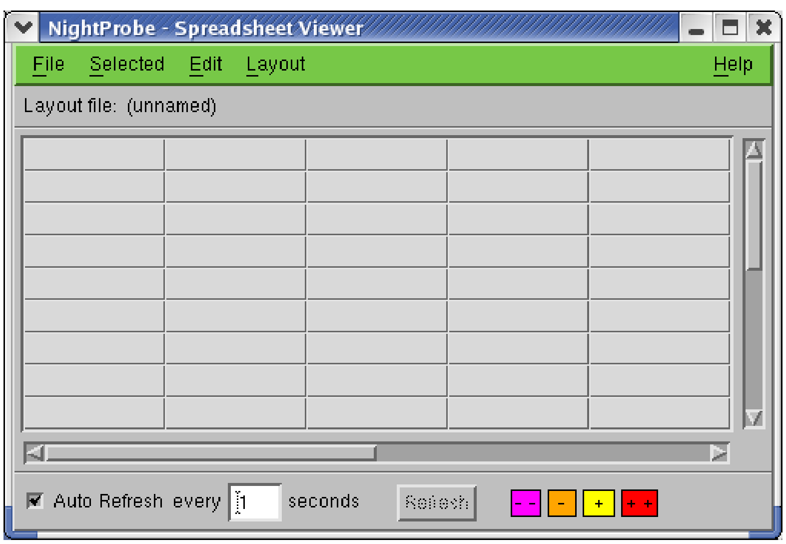


Figure 4-5. NightProbe Spreadsheet Viewer

- Select the Place Variables menu option from the Selected menu

The Spreadsheet Variables selection window is displayed.

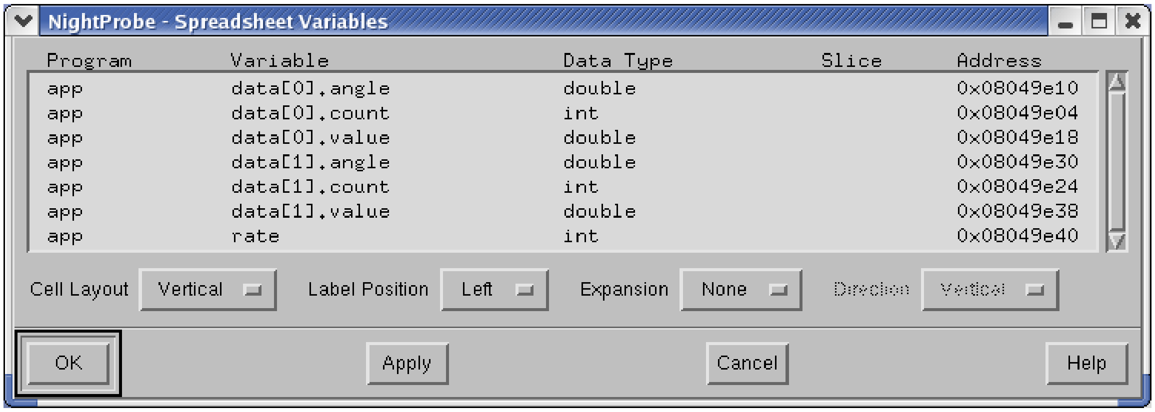


Figure 4-6. NightProbe Spreadsheet Variables selection dialog

- Select all the variables displayed using a Shift+Click selection operation
- Press the OK button

All seven variables have now been added to the spreadsheet.

Widen the column displays using the following actions:

- Select both of the first two columns
- Select the Column Width menu option from the Layout menu
- Change the width to 20
- Press the Set button

Probing Variables

Data probing begins when we connect to the target resources and request data samples.

- Press the Connect button in the Sampler Control area
- Press the Sample button in the Sampler Control area

The default Timer setting is Sample on demand.

Each time you press the Sample button, NightProbe samples all data items and sends them to the output items you have selected.

The List Viewer and Spreadsheet Viewer windows update to display the values of each sample.

- Press the Disconnect button

- Right-click the Timer icon in the Session Overview area and select the System Clock... menu option
- Change the units of time to msec from the option list
- Change the sampling rate to 100
- Press the Set Timer button
- Press the Connect button in the Sampler Control area
- Press the Start button in the Sampler Control area

NightProbe is now collecting data samples every tenth of a second automatically.

Every sample is written to the output file you selected.

The most recent samples are displayed in the List Viewer and Spreadsheet Viewer windows, at a user-selectable rate which default to once per second.

- Press the Disconnect button

Save the current NightProbe session.

- Select the Save Session... menu option from the NightProbe menu in the Main window
- Type in a filename that identifies the session
- Press the Select button
- Press Yes to also save the spreadsheet layout in the Warning dialog that appears

Viewing Recorded Data

The List Viewer can be used to view all samples of data recorded via NightProbe.

- Select the Open Data File... menu option from the File menu of the List Viewer window.
- Enter the name of the output file selected in “File Output” on page 4-5.

  `/tmp/nprobe-data`

- Press the OK button

A textual description of every data sample is shown in the window.

You can save the textual description to a file using the Save As Text... menu option of the File menu.

Viewing Data with NightTrace

Probed data can be sent to NightTrace for live analysis.

- Right-click the Outputs icon in the Session Overview area and select the NightTrace Output... menu option

The NightTrace Output dialog is displayed.

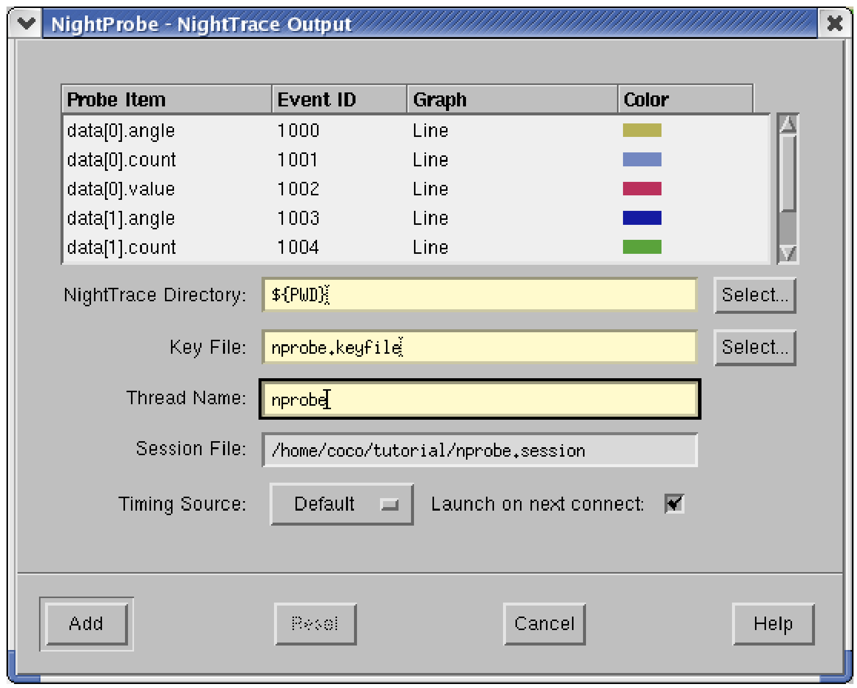


Figure 4-7. The NightTrace Output Selection Window

By default, all items are selected for graphing with NightTrace.

- Resize the window so that all seven data items can be seen
- Select all entries in the list using a Shift+Click selection operation
- Right-click and select the Do not graph menu option
- Right click the Do not graph text for the `data[0].value` variable
- Select Graph with line from the graph options
- Press the Add button to close the NightTrace Output dialog

NightTrace will be sent all data items in the sample but has been configured to generate a data graph for the value `data[0].value`.

- Press the Connect button in the Sampler Control area

- Press the Start button in the Sampler Control area of the NightProbe Main window

At this point in the tutorial, it is assumed that you are familiar with the basic operation of NightTrace. If not, please review the chapter on “Using NightTrace” on page 3-1 before proceeding.

- Press Launch in the Main NightTrace window
- Press Resume in the Main NightTrace window
- Raise the user display page

The top row of the column graphs the value of data[0].value.

The bottom rows in the column contain an event indication for every variable from every data sample.

The various data boxes in the user display page indicate the name of the variable associated with the current NightTrace event and its value.

- Zoom out and scroll to the right to see the sine wave generated by `data[0].value`

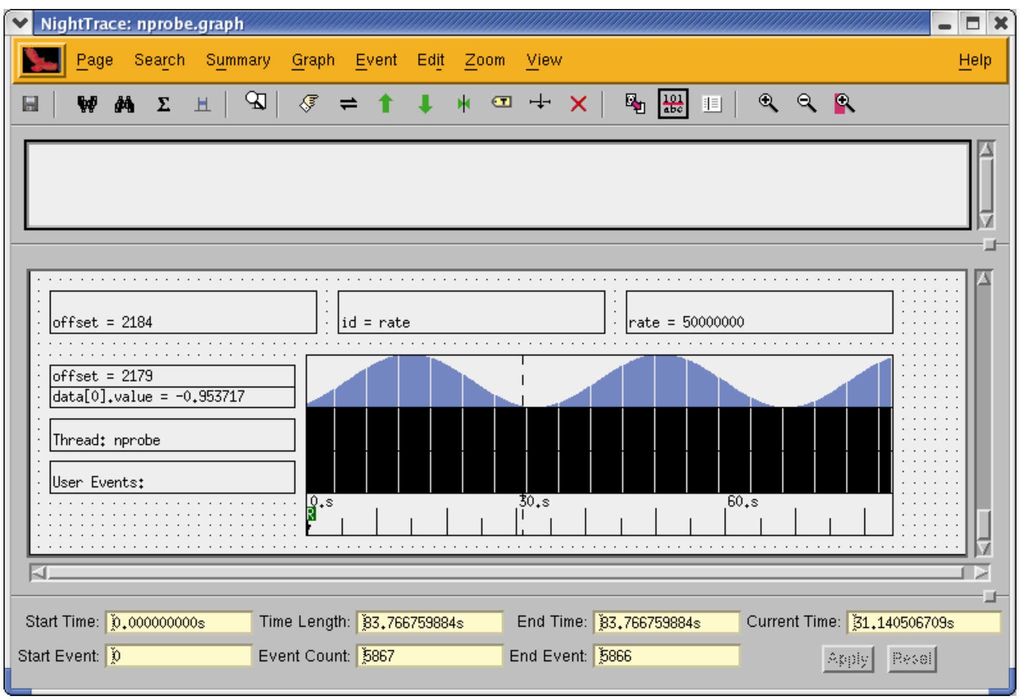


Figure 4-8. NightTrace User Display Page

NightProbe allows you to change the value of variables as well as to view their current value.

- Raise the NightProbe Spreadsheet Viewer window

- Click in the cell which shows the value of `data[0].angle`

The background changes to blue and the value stops updating

- Backspace over the existing value shown and enter

    `0.0`

- Hit the <enter> key

Now return to the NightTrace user display page.

- Scroll to the right until the latest data appears

Note the break in the continuous sine wave when we reset the angle to zero.

Terminate execution of NightTrace using the following actions:

- Press Halt in the Main window
- Select Exit Immediately from the NightTrace menu in the Main window

Using the NightProbe API

NightProbe includes a simple API which allows you to unpack data samples from pre-recorded data files or to analyze streaming data samples in a live environment.

Included in the /usr/lib/NightStar-RT/tutorial directory is a simple program which reads data samples from `stdin` using the NightProbe API and outputs a streaming graphical value of two of the data items in each sample.

- Press the Disconnect button in the NightProbe Main window
- Right-click the Outputs icon in the Session Overview area and select the Program Output... menu option

The Program Output dialog is displayed.

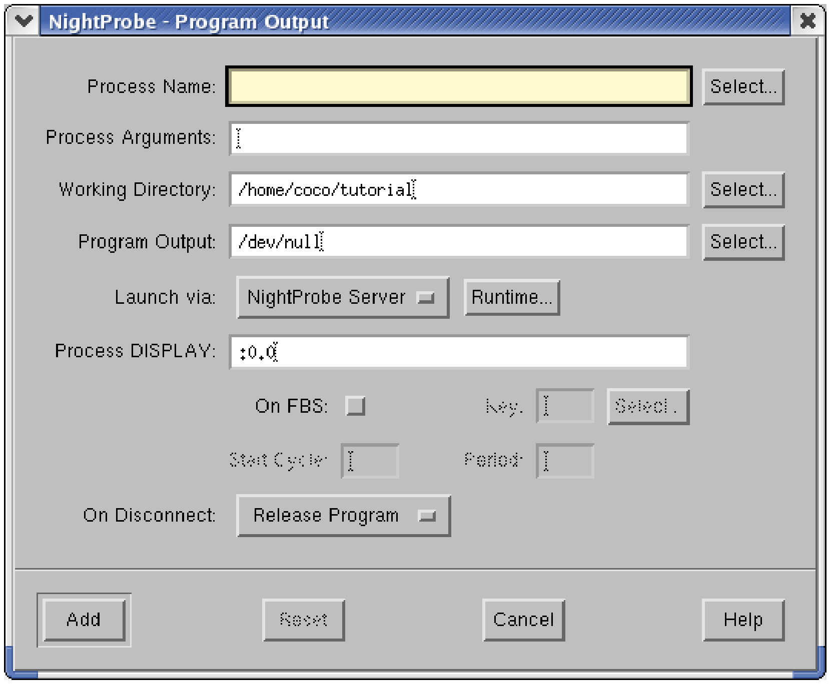


Figure 4-9. The NightProbe Program Output Window

- In the Process Name field, enter the full pathname to the graph program copied into your tutorial directory in "Getting Started" on page 1-2.

  /home/*user***/tutorial/graph**

- Ensure the Working Directory path is correct
- Press the Add button

The graph program will be launched when we next connect to the target resources.

- Press the Connect button in the Sampler Control area of the Main window

A small window entitled graph is displayed.

If a diagnostic window appears with text similar to the following, dismiss it.

```
Server Error: (14) Xlib: extension "GLX" missing on
display ":0.0".
```

When data sampling begins, the graph program will begin to display sine and cosine waves based on the values of `data[0].value` and `data[1].value` in the data samples sent to the program from NightProbe.

- Press the Start button in the Sampler Control area of the NightProbe Main window

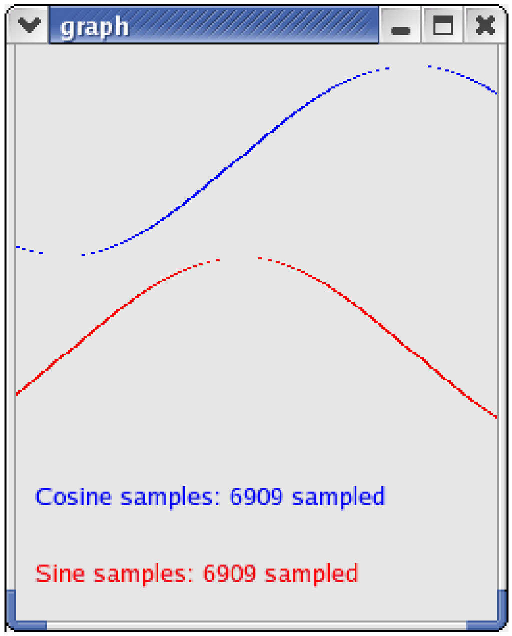


Figure 4-10. Example Output of Graph Program

Change the rate at which the **app** program runs using the following steps:

- Raise the NightProbe Spreadsheet Viewer window
- Click in the cell displaying the value of the `rate` variable

The background turns to blue and the value stops updating.

- Backspace over the displayed value and type in

  5000000

- Hit the <enter> key

The shape of the sine and cosine waves change as shown in the graph window.

For more information on the NightTrace API, refer to the *NightProbe API* chapter in the *NightProbe User's Guide* (0890465).

Conclusion - NightProbe

To terminate NightProbe operations, execute the following steps:

- Press the Disconnect button in the Sampler Control area of the Main window

- Select the Exit menu option from the NightProbe menu
- Select No from the warning dialog
- Close the graph window of the **graph** program

This concludes the NightProbe portion of the NightTrace RT User's Guide.

5
Using NightTune

NightTune is a graphical tool for analyzing and adjusting system activities.

This chapter assumes you have already built the **app** program and it is running. If you have not built the program, do so using the instructions in "Building the Program" on page 1-4 and execute the application via the following command before proceeding:

```
./app &
```

Invoking NightTune

NightTune can be launched with the following command at a command prompt:

```
ntune &
```

Or it may be launched by double-clicking on the NightTune desktop icon.

For some aspects of this tutorial, it will be necessary to execute NightTune as the **root** user or to ensure that your user account has appropriate privileges. See the "Setting Up User Privileges" on page 1-2 for more information.

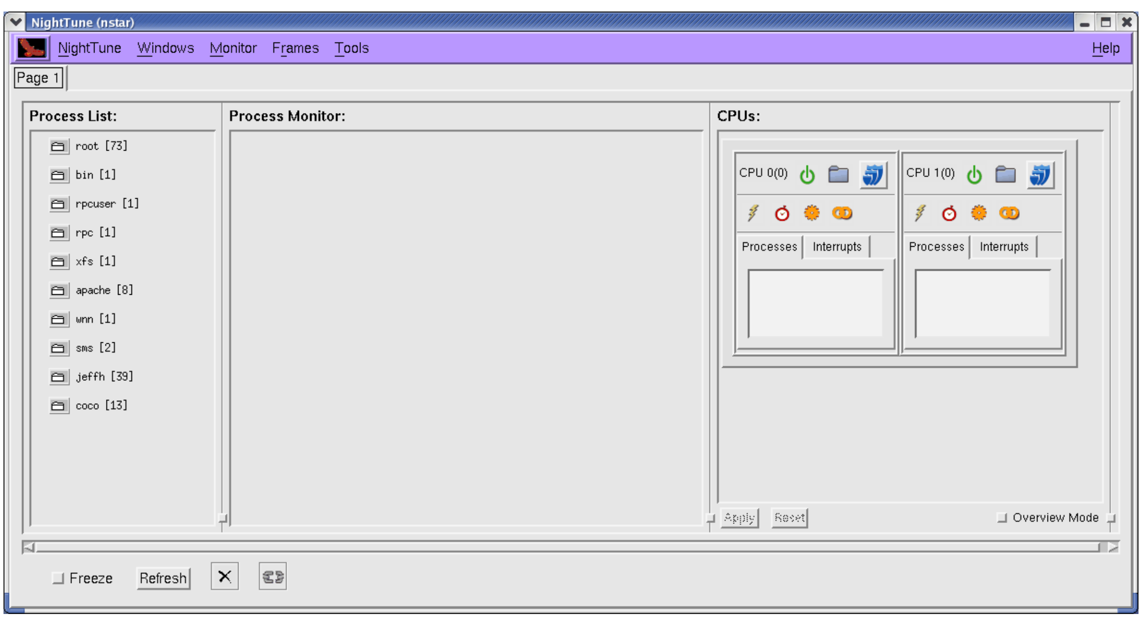


Figure 5-1. NightTune initial panels

Monitoring a Process

First monitor the running **app** process.

- In the Process List panel on the left side of the window, click on the user running that process.
- Click on the **app** process that appears under that user.

Its checkbox will be checked, and that process will appear in the Process Monitor panel in the middle of the window.

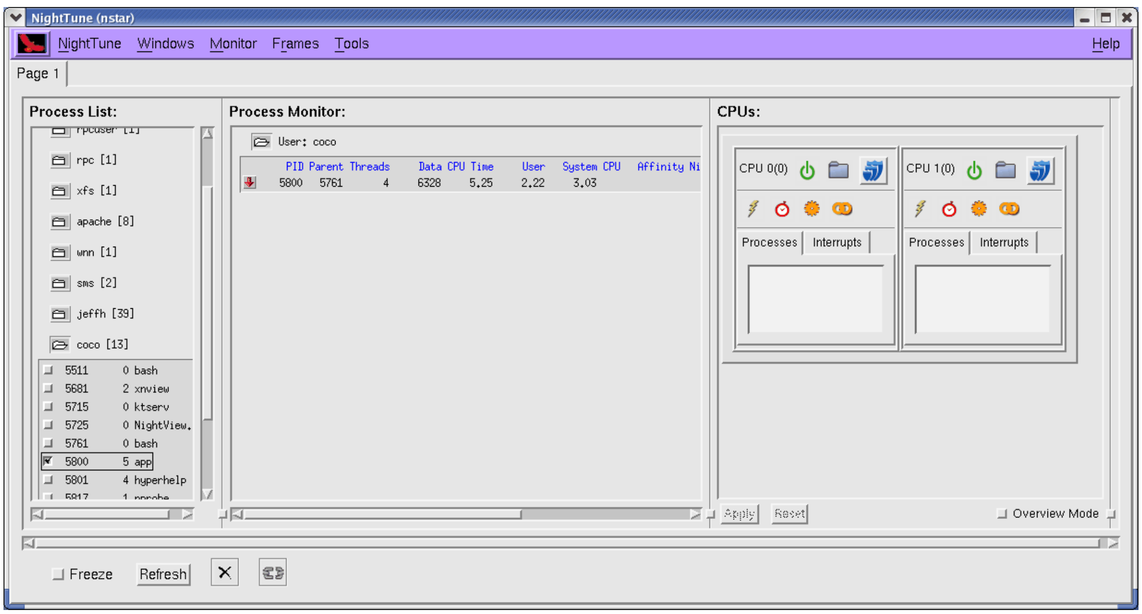


Figure 5-2. NightTune Process Monitor panel

- Resize the Process Monitor panel by clicking the sash (small box) to the right of its scrollbar and dragging it until you can see all the columns displayed for the **app** process in the Process Monitor panel.

- To monitor each thread in the **app** process individually, press the down arrow button to the left of that process in the Process Monitor panel.

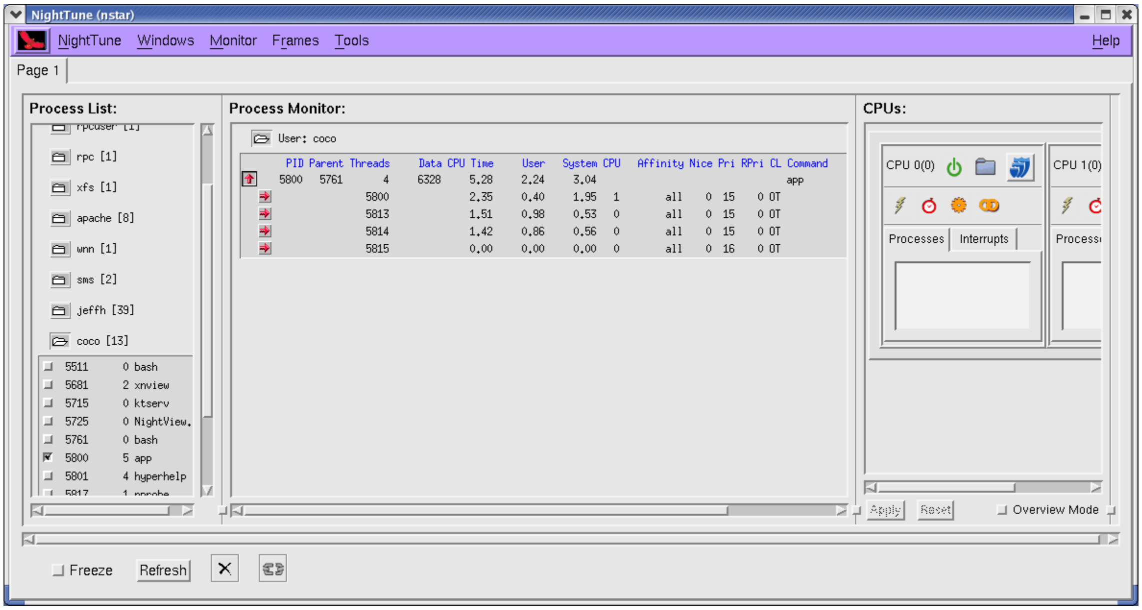


Figure 5-3. NightTune Process Monitor panel with threads

The Process Monitor panel shows characteristics of each thread and of the entire process. In particular, they include:

- memory usage of the process, broken down by virtual memory size, residence memory size, and data memory (including only data memory by default)
- amount of time used by each thread or the whole process, broken down by system time, user time, and total time.
- CPU on which each thread ran most recently
- CPU affinity for each thread (the set of CPUs on which the thread is allowed to run)
- scheduling characteristics of each thread

The set of columns displayed can be modified by clicking the Frames menu, then on the Display Fields menu item, and then choosing individual fields by checking or unchecking their menu items.

Changing Process Scheduling Parameters

It may be desirable to change the scheduling properties of a thread or process while it is running to see how that changes the behavior of an application. For instance, perhaps one thread is being starved of CPU time by other threads. You may wish to change its scheduling class to a real-time class and/or its priority to a higher priority.

- Click on the right arrow button to the left of one of the threads in the process **app**.

The Process Scheduler dialog will appear.

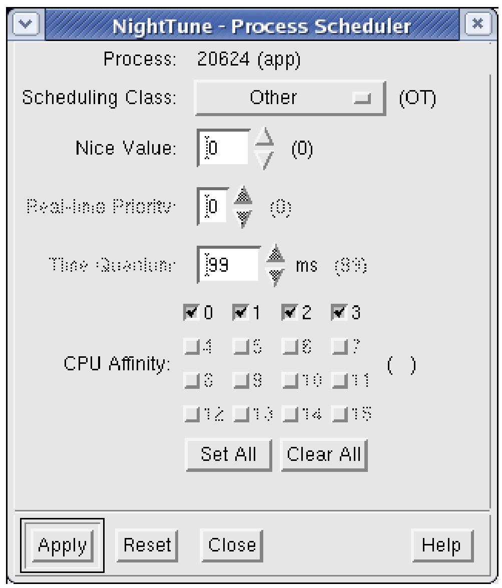


Figure 5-4. Process Scheduler dialog

In this dialog, it is possible to change the Scheduling Class, Nice Value, Real-time Priority, and/or Time Quantum. On multi-processor systems, it also is possible to change the CPU Affinity. For each CPU on which the process or thread is allowed to run, the checkbox with the number of that CPU should be checked. See “Setting Process CPU Affinity” on page 5-5 for more on this topic.

- Change the Scheduling Class to Round Robin by selecting that from a drop down list.
- Change the Real-time Priority to 3 by entering that value into the field.
- Press the Apply button.
- Dismiss the Process Scheduler dialog by pressing the Close button.

NOTE

To change the Scheduling Class to Round Robin and change the Real-time Priority, it is necessary that NightTune be run by the **root** user or that your user account has appropriate privileges as described in “Setting Up User Privileges” on page 1-2.

The Process Monitor panel now reflects these changes to the thread.

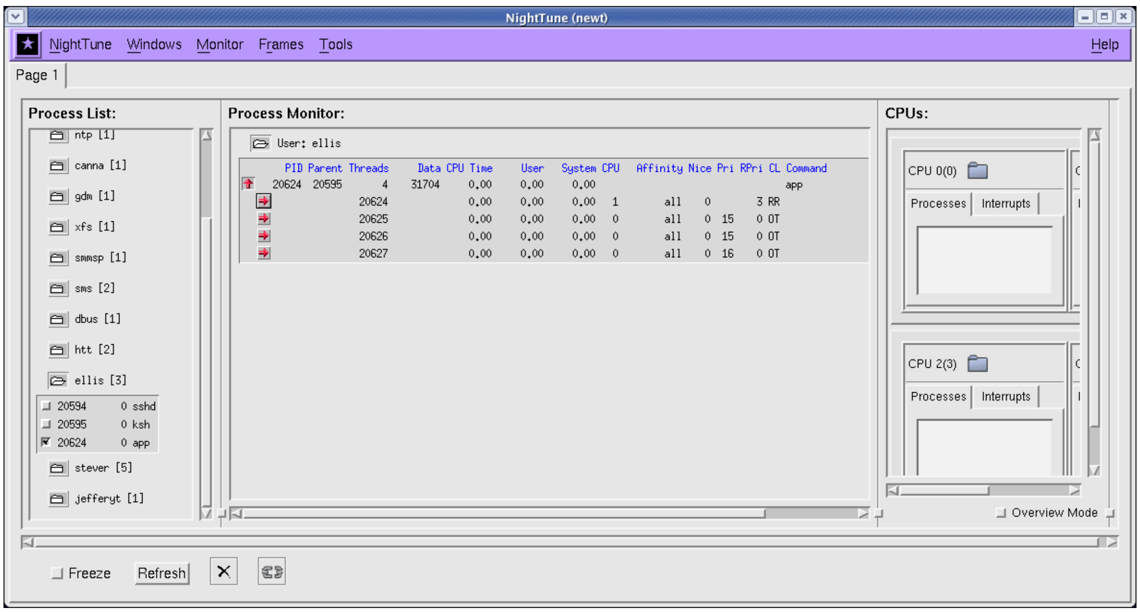


Figure 5-5. NightTune Process Monitor with modified thread

For the modified thread, the CL (Scheduling Class) field displays the value RR (Round Robin), and the RPri (Real-time Priority) field displays the value 3.

Setting Process CPU Affinity

This section only is applicable if the system running NightTune is a multi-processor system. If not, skip to “Monitoring Processor Usage” on page 5-11.

The Process List panel no longer is necessary, so close it.

- Click on the Monitor menu, and then on the Process List menu item.

The Process List panel will disappear.

The CPU Status panel (labeled CPUs) may still be only partially visible. To the right of the panel at the bottom is a sash.

- Press and hold the sash and then drag it to the right. This will resize the CPU Status panel. Make the panel wide enough so that all CPUs are visible.

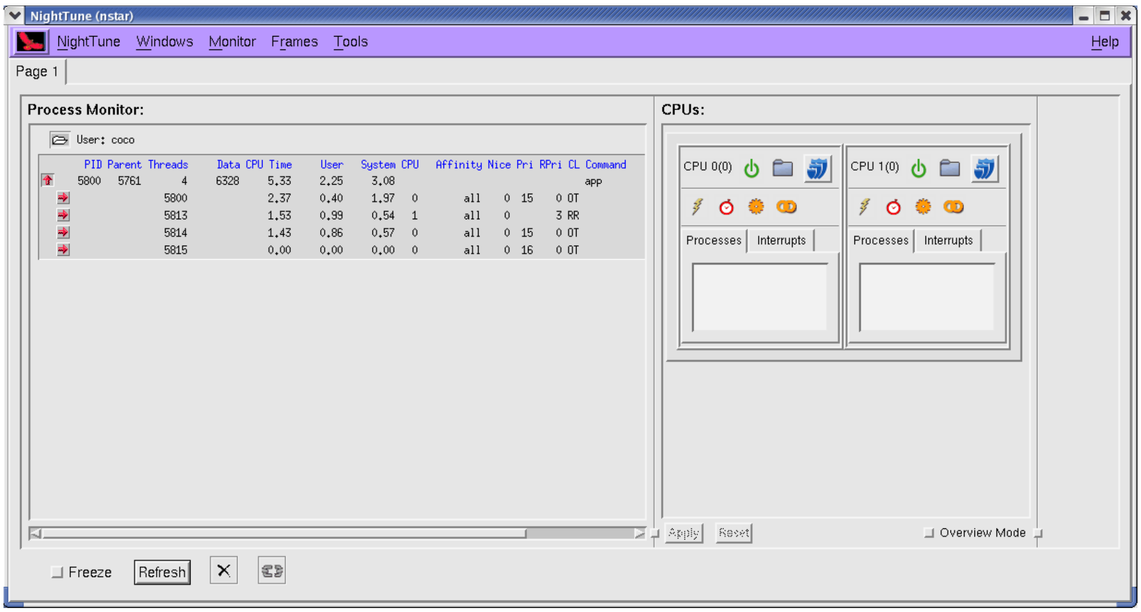


Figure 5-6. NightTune with CPU Status panel

A process or thread has a CPU affinity, which determines the set of CPUs on which it may execute. It may even be restricted such that it may run on only a single CPU. Often this is called *binding* the process or thread. “Changing Process Scheduling Parameters” on page 5-4 described one way to change the CPU affinity. In addition, the CPU Status panel can be used to bind a process or thread quickly.

- While the cursor is positioned over one of the threads in the **app** process, press and hold the *middle* mouse button, then drag the thread to one of the CPUs in the CPU Status panel.

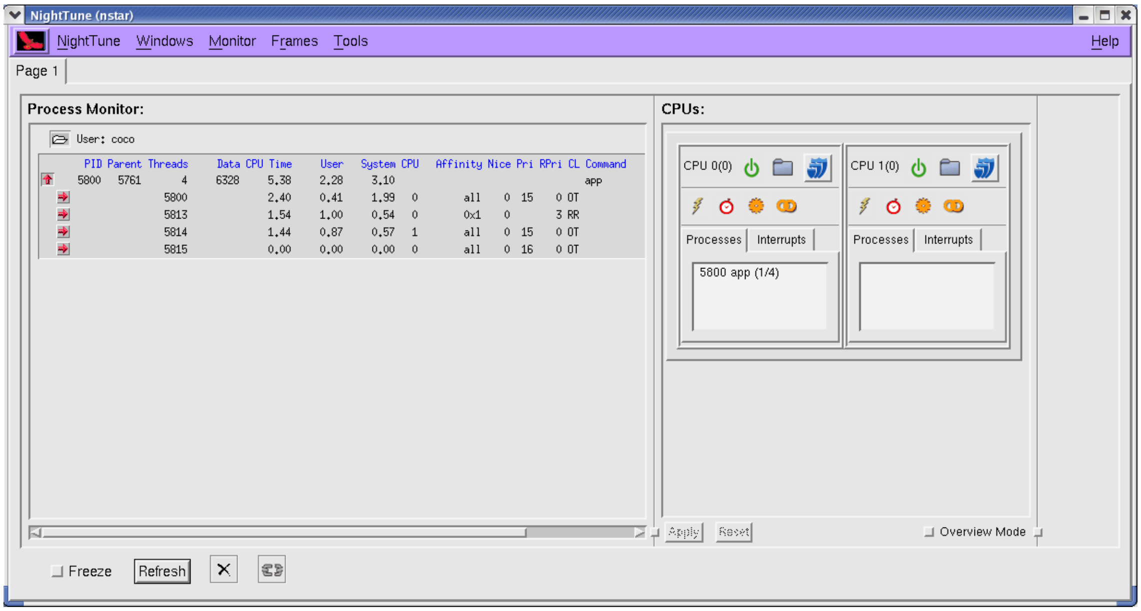


Figure 5-7. NightTune with bound thread

This action binds the selected thread to the particular CPU. That is, its CPU affinity is set to include only that single CPU. When a process' or thread's CPU affinity contains only a single CPU, that process or thread is listed in the CPU Status panel under the particular CPU's Processes tab. In this case, there is one entry under CPU 1. Because only one thread was bound to CPU 1 in this example, the entry includes the suffix (1/4), indicating that only 1 of the 4 threads is bound to that CPU.

The thread's new CPU affinity also is reflected in the Affinity field of the Process Monitor panel. That field displays a bit mask in hexadecimal, where the low order bit represents CPU 0, the next bit represents CPU 1, etc. In this case, the value 0x2 has only the second lowest bit turned on, indicating CPU 1.

NightTune also can unbind a process quickly.

- While the cursor is over the thread entry in the CPU Status panel, press and hold the *middle* mouse button, then drag the item to the Unbind destination panel at the bottom left of the window (appearing like a broken chain link).

The Process Monitor panel will reflect that the thread is unbound once again.

Setting Interrupt CPU Affinity

The functionality described in this section only is available if NightTune was executed by the **root** user or your user account has appropriate privileges as described in "Setting Up User Privileges" on page 1-2. If this is not the case, skip to "Monitoring Processor Usage" on page 5-11.

In addition to being able to set the CPU affinity of a process, NightTune can control the CPU affinity of an interrupt.

It may be desirable to change the CPU affinity of an interrupt. For instance, an interrupt may be occurring frequently and, depending on the CPU which handles it, may be interfering with an application running on that same CPU.

- Close the Process Monitor panel by clicking on the Monitor menu and then the Process Monitor menu item.

- In its place, open the Interrupt Activity panel by clicking on the Monitor menu and then the Interrupt Activity menu item.

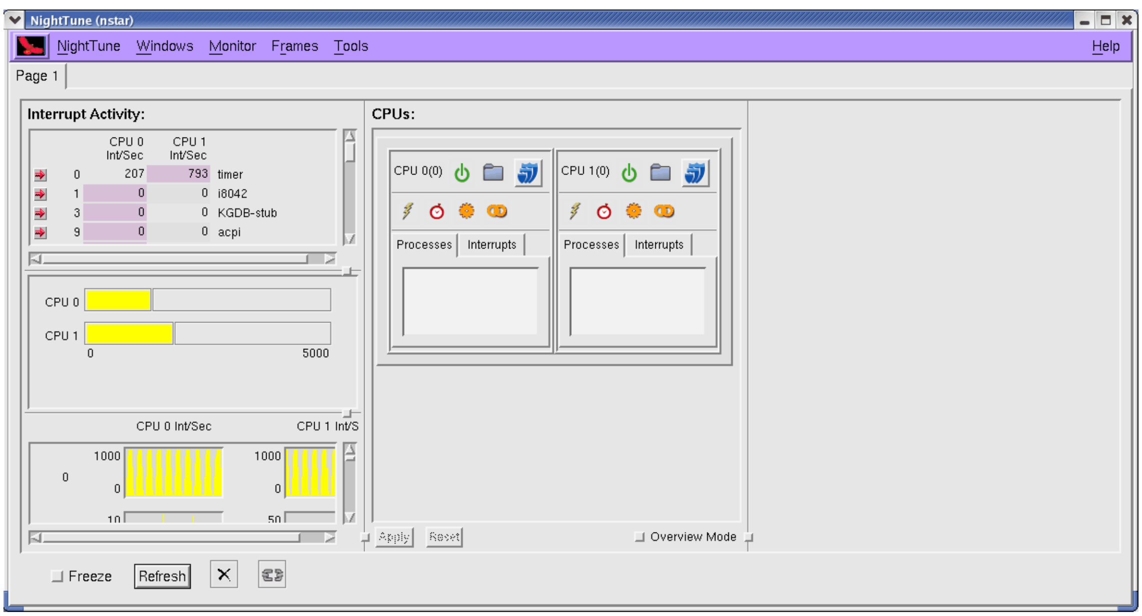


Figure 5-8. NightTune with Interrupt Activity panel

The Interrupt Activity panel is comprised of 3 sub-panels which display interrupt activity in 3 different formats. The top sub-panel shows the number of interrupts per second for each interrupt as handled on each CPU (if on a multi-processor system). The middle sub-panel shows the total number of interrupts per second for each CPU using bar graphs. The bottom sub-panel shows that the number of interrupts per second over a recent span of time for each interrupt as handled on each CPU using line graphs.

For the purpose of this tutorial, hide all but the top sub-panel

- Within the Interrupt Activity panel, press and hold the *right* mouse button so that a drop-down menu appears, and select Hide bargraph display.
- Then do the same, but select Hide linegraph display.
- Depending on the number of CPUs, it also may be necessary to widen the Interrupt Activity panel. To do this, over the sash to the lower right of the panel, press and hold the left mouse button and drag it to the right until the whole panel is visible.
- After doing that, it may be necessary to widen the CPU Status panel similarly.
- Depending on the number of interrupt sources, it may also be necessary to resize the window vertically.
- Click on the Interrupts tabs in each CPU within the CPU Status panel.

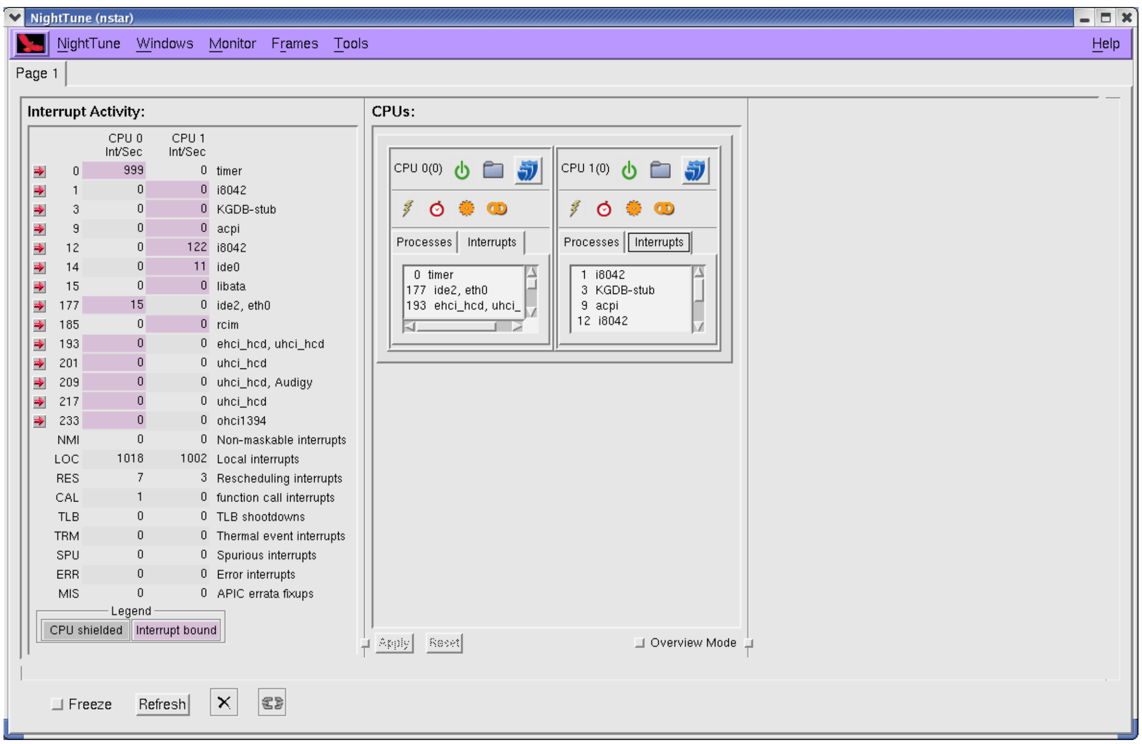


Figure 5-9. NightTune with resized Interrupt Activity panel

The colored boxes in the Interrupt Activity panel indicate that an interrupt may be handled by that particular CPU. However, if an interrupt may be handled by all CPUs, then no colored boxes appear for that interrupt. The same information is displayed in the Interrupts tabs for each CPU in the CPU Status panel.

To bind an interrupt to a single CPU, it may be dragged in much the same way as a process.

While the cursor is over an interrupt in the Interrupt Activity panel, you may press and hold the *middle* mouse button, then drag the interrupt to the particular CPU in the CPU Status panel. Similarly, while the cursor is over an interrupt in the Interrupts tab of a CPU in the CPU Status panel, you may press and hold the middle mouse button, then drag the interrupt to a different CPU in the CPU Status panel.

To change an interrupt's affinity to allow multiple CPUs, but possibly exclude one or more, click on the right arrow to the left of a particular interrupt, such as interrupt 225 in this example, and the Interrupt Affinity dialog will appear:

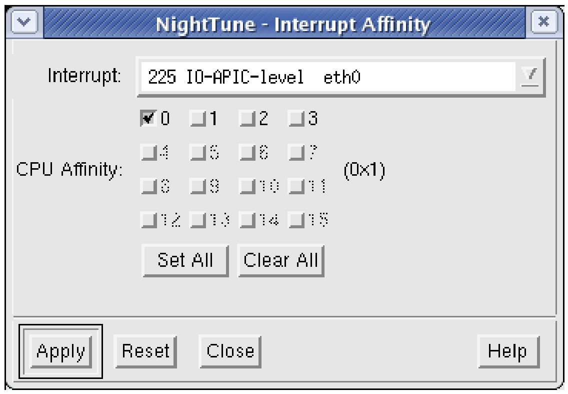


Figure 5-10. Interrupt Affinity dialog

For each CPU on which the interrupt is allowed to be handled, the checkbox with the number of that CPU should be checked. The changes take effect when the Apply button is pressed.

- Using any of these techniques, change the CPU affinity such that no interrupts may be handled on CPU 1.

NOTE

For certain interrupts, such as NMI, it is impossible to control their CPU affinity.

Figure 5-11. NightTune with no interrupts on CPU 1

Finally, it is possible to unbind an interrupt in much the same way as a process. While the cursor is over the interrupt entry in the CPU Status panel, press and hold the *middle* mouse button, then drag the item to the Unbind destination panel at the bottom left of the window (appearing like a broken chain link).

Monitoring Processor Usage

This section describes how to monitor the utilization of each CPU on the system. The concepts herein also apply to monitoring context switches, virtual memory, disk activity, and network activity.

Panels left open from previous sections no longer are necessary, so close them.

- Click on the Monitor menu, and then on any menu item with a checked checkbox to its left. The panel with that name will close. Repeat this until no panels remain.
- In their place, open the Processor Usage panel by clicking on the Monitor menu and then the Processor Usage menu item.

Depending on the number of CPUs, it may be necessary to widen the Processor Usage panel.

- To do this, over the sash to the lower right of the panel, press and hold the left mouse button and drag it to the right until all the columns of graphs in the bottom sub-panel are visible.

- It may be necessary to resize the window vertically, and then heighten the 3 sub-panels using the sashes on the horizontal separators to the lower right of each sub-panel.

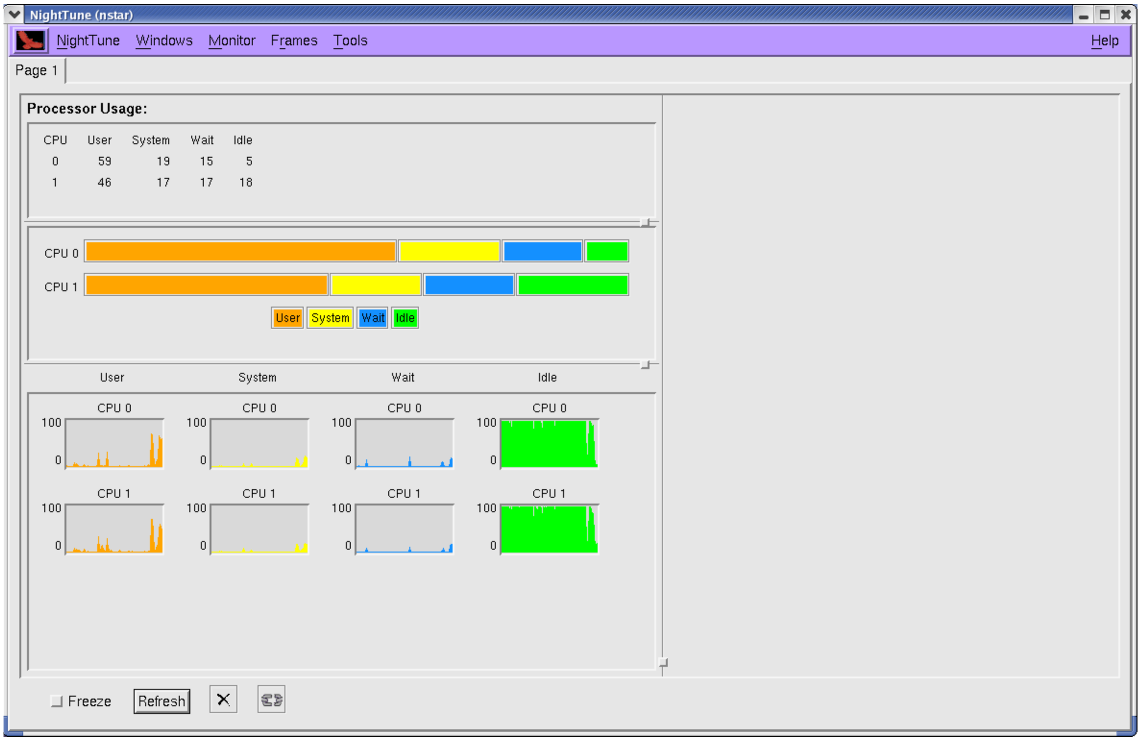


Figure 5-12. NightTune Processor Usage panel

The Processor Usage panel is comprised of 3 sub-panels which display processor usage percentages in 3 different formats. The top sub-panel shows the processor usage percentage spent doing each of these activities:

- executing user code (including kernel daemons which handle post-interrupt processing)
- executing system code (i.e. executing in the kernel)
- waiting for an I/O operation
- waiting in the idle loop

These percentages are shown separately for each CPU (if on a multi-processor system).

The middle sub-panel shows the same information using bar graphs. A legend at the bottom of that sub-panel shows how each color represents each of the activities listed above.

The bottom sub-panel shows that the same information over a recent span of time using line graphs. The legend from the middle sub-panel applies to it as well.

Conclusion - NightTune

To terminate NightTune, click on the NightTune menu and then the Exit menu item.

This concludes the NightTune portion of the NightTrace RT User's Guide.

6
Using NightSim

NightSim is a graphical tool for scheduling multiple processes in a synchronized manner and monitoring their execution.

NightSim provides a graphical interface to the Frequency Based Scheduler utilities which are part of the RedHawk Linux operating system.

If you don't have the Frequency Based Scheduler installed on your system, this portion of the tutorial isn't applicable to you. Use the following command to see if the Frequency Based Scheduler is installed:

```
rpm -q ccur-fbsched
```

This chapter of the tutorial also uses a real-time clock interrupt source from the Real-Time Clock and Interrupt Module (RCIM) which is standard equiptment on all iHawk systems. If your system does not include an RCIM device, this portion of the tutorial isn't applicatble to you. Use the following command to see if an RCIM is installed:

```
cat /proc/driver/rcim/status
```

If the file shown above does not exist, an RCIM does not exist on your system or your kernel has had the RCIM support removed.

For some aspects of this section, it will be necessary to execute NightSim and NightTune as the **root** user or to ensure that your user account has appropriate privileges. See the "Setting Up User Privileges" on page 1-2 for more information.

Invoking NightSim

A NightSim configuration file has been prepared for this tutorial and should have been copied to your current working directory during the activities in the section entitled "Creating a Tutorial Directory" on page 1-3 .

Launch NightSim specifying the configuration file, as show below:

```
nsim -f nsim.config -offline &
```

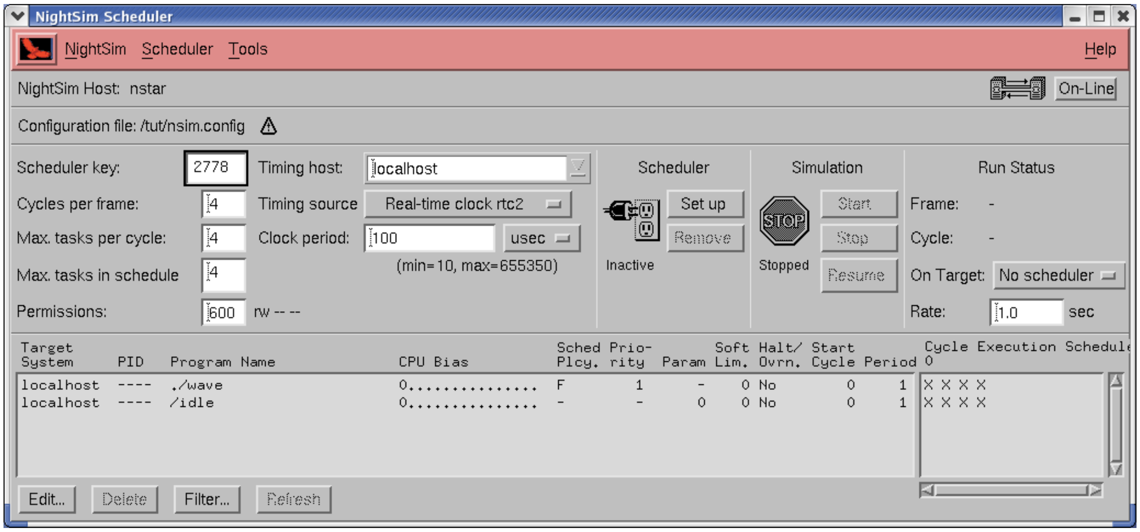


Figure 6-1. NightSim initial window

Creating a Scheduler

NightSim allows you to define the scheduling of multiple processes, using the following parameters:

- The scheduling source (usually an external interrupt)
- The rate at which the interrupts occur (for clock-based interrupts)
- The period at which a process is scheduled
- The CPU affinity, scheduling policy and priority of scheduled processes

Collectively, these parameters define a *scheduler*.

A cycle is defined as the time between the scheduling sources (interrupts).

A frame is defined by a fixed number of cycles. Frames are useful concepts in many cyclic applications where a series of discrete steps must be executed (cycles) in order before the entire algorithm repeats (frame).

The scheduler configured by the nsim.config file specified on the command line above defines a scheduler with the following attributes:

- Cycles Per Frame -- four cycles per frame
- Timing Source - an interrupt source using RTC2 of the Real-time Clock and Interrupt Module device (RCIM)
- Clock Period -- a cycle time of 100 microseconds
- Processes -- a single process, wave, schedule to run on every cycle of the frame

To view the details of the attributes of the scheduled process, select the **`./wave`** process in the process area at the bottom portion of the NightSim window and then press the Edit... button on the lower left hand area of the window.

The Edit Process window is displayed.

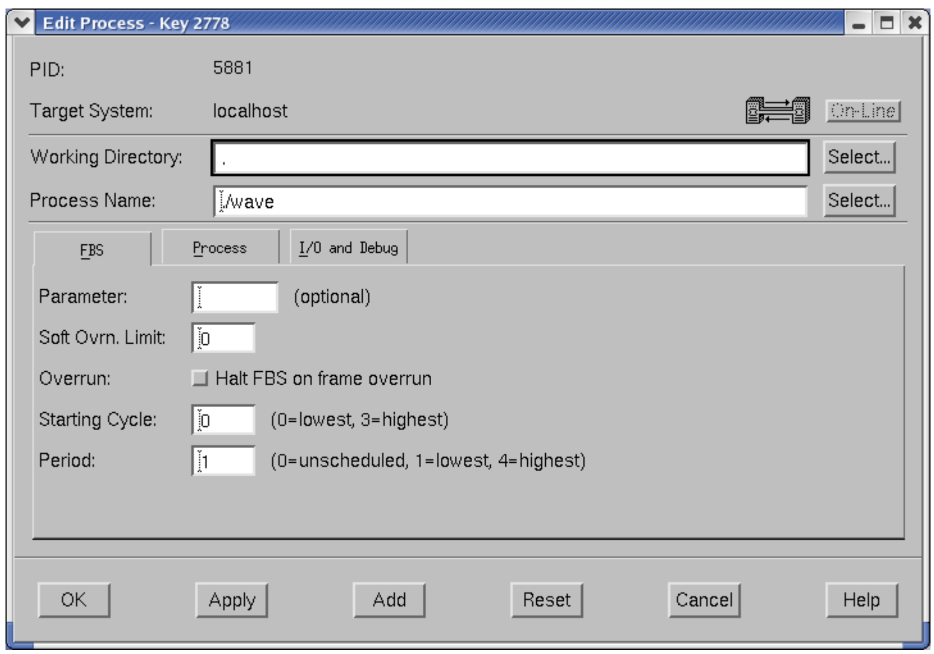


Figure 6-2. NightSim Edit Process Window

The Edit Process window shows the starting cycle and period of the **`wave`** process. The Staring Cycle defines the cycle within the frame where the process will begin its execution. The Period defines how the frequency of execution, in cycles.

Click on the Process tab of the Edit Process window.

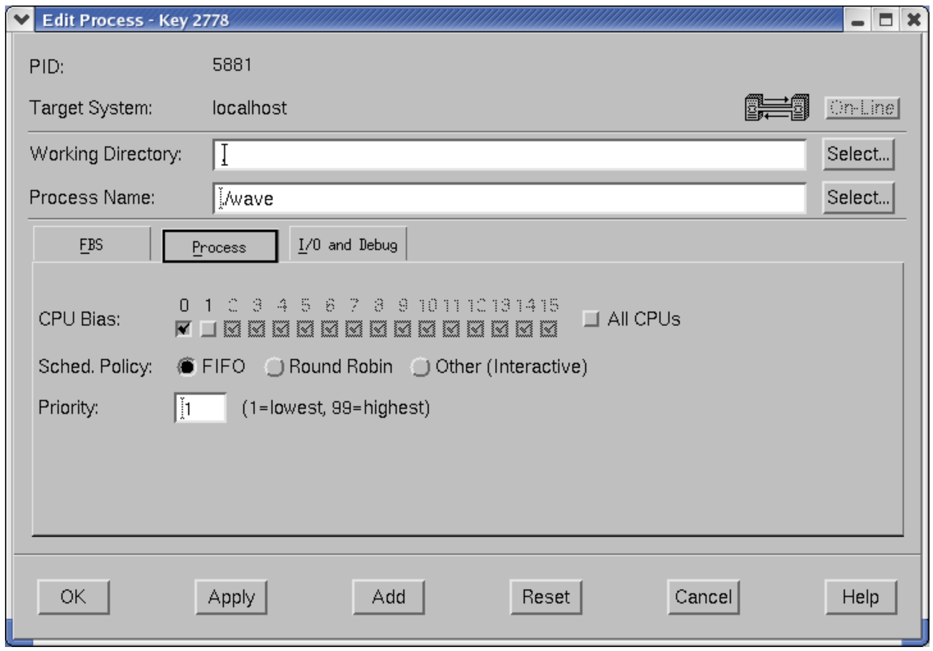


Figure 6-3. NightSim Edit Process Window -- Process Tab

The Process tab allows you to chose the CPU on which execution is allowed, the scheduling policy, and the scheduling priority of the process.

Close the Edit Process window by pressing the Close button.

Notice that in addition to the **wave** process, the **/idle** process is listed in the scheduling area of the NightSim window. We have registered the **/idle** process so that we may subsequently monitor the amount of idle time available for each cycle. The **/idle** process is not a process that is scheduled, but rather it is a placeholder used to represent idle cycles.

Application Source Coding

It is trivial to modify cyclic application so that they may be scheduled via NightSim.

A single API call is required.

The source code for our simplistic wave application follows:

```
#include <fbsched.h>
int workload = 1000;
main()
{
   int data = 0;
```

```
        int i;
        volatile double d = 1.0;
        while (fbswait()==0) {
            data = !data;
            for (i=0; i<workload; ++i) d = d/d;
        }
    }
```

The call to `fbswait()` causes the process to block until its next scheduled cycle at which point it returns. The process then performs its workload and then loops to block in `fbswait()` until its next scheduled cycle.

Compile and link the application using the following command:

```
cc -g -o wave wave.c -lccur_fbsched -lccur_rt
```

Running the Scheduler

To start the scheduling of the process, press the Setup button followed by the Start button in the NightSim window.

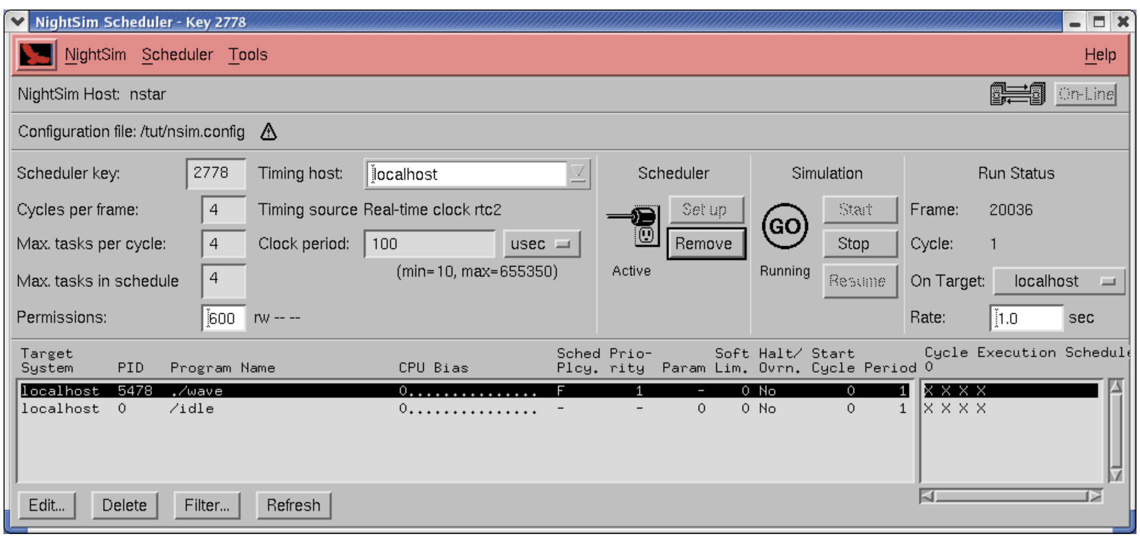


Figure 6-4. NightSim Window -- Scheduling has begun

Note the Frame count begins to increase under the Run Status area of the NightSim window as the Cycle oscillates between 0 and 3.

To monitor the execution of the process, launch a monitor window by selecting the Create Monitor Window menu item of the NightSim menu.

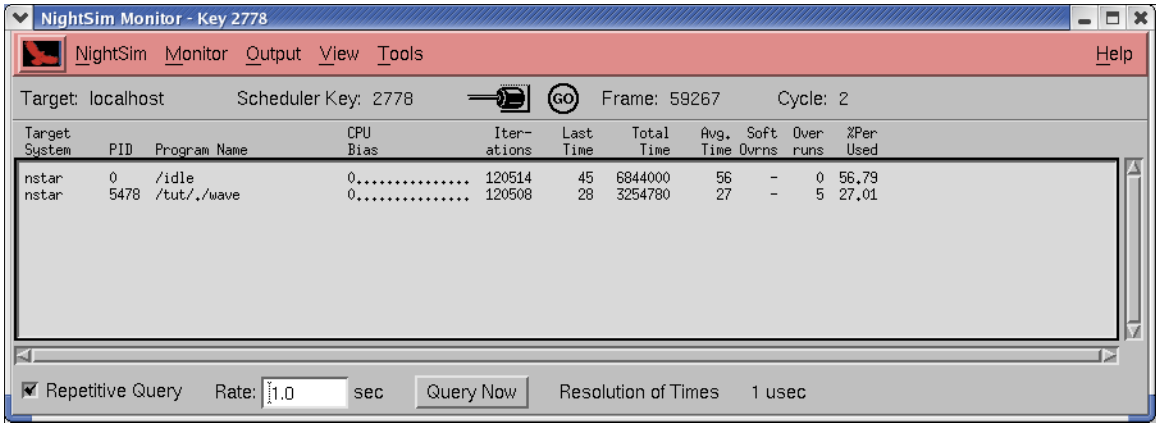


Figure 6-5. NightSim Monitor window

The NightSim Monitor window provides statistics about each individual process on the scheduler. It includes the process ID, process name, CPU affinity, number of cycles executes, and the times for each cycle execution. Additional statistics can be displayed via the Display Fields menu item of the View menu.

Watch the Last Time column. The values displayed are the CPU time used by each process for their last cycle's execution in microseconds. The values attributed to the **/idle** process indicate the remaining CPU time available within the cycle.

We will adjust the workload of the **wave** process and see the effects shown in the NightSim Monitor window.

Using Datamon to Modify Program Variables

The Data Monitoring Application Programming Interface is part of the NightStar RT tool set.

Data Monitoring allows you to specify executable programs that contain Ada, C, or Fortran variables to be monitored, obtain and modify the values of selected variables by specifying their names, and obtain information about the variables such as their addresses, types, and sizes.

Data Monitoring is a powerful capability with a rich API. However, for our purposes, we will write a very simple program which changes the value of a single variable.

Refer to the *Data Monitoring Reference Manual* for more information about Data Monitoring.

The source code for our **set_workload** program follows:

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

#define check(x) if((x)) {fprintf(stderr, "%s\n",
dm_get_error_string());exit(1);}

main(int argc, char * argv[])
{
   program_descriptor_t pgm;
   object_descriptor_t   obj;
   char buffer[100];

   if (argc != 2) {
      fprintf (stderr, "Usage: set_workload integer-value\n");
      exit(1);
   }

   check(dm_open_program("wave",0,&pgm));
   check(dm_get_descriptor("workload",0,pgm,&obj));
   check(dm_get_value(&obj,buffer,sizeof(buffer)));
   check(dm_set_value(&obj,argv[1]));

   printf ("workload: old_value=%s, new_value=%s\n", buffer, argv[1]);
}
```

The `dm_open_program` function initializes Data Monitoring on the specified process name and PID (in this case zero, which instructs the call to use any process matching the specified name).

The `dm_get_descriptor` call looks for the specified variable name and returns information about the variable. It also maps the underlying memory page of the variable in the wave process into the monitoring process.

The `dm_get_value` and `dm_set_value` routines return and set the value of the variable using direct memory reads and writes; the **wave** process is not affected in any other way than having the value of the workload variable changed.

The **set_workload.c** source file was copied into the current working directory during the activities in the section entitled “Creating a Tutorial Directory” on page 1-3 .

Compile the program using the following command:

```
cc -g -o set_workload set_workload.c -ldatamon -lccur_fbsched -lccur_rt
```

Change the value of the `workload` variable in the **wave** process by issuing the following command:

```
./set_workload 0
```

As shown in the source code above, the program prints the previous value of the `workload` variable and then sets it to the value specified as an argument to **set_workload**.

The Last Time field for the workload process is affected by the reduced workload as shown in the NightSim Monitor window.

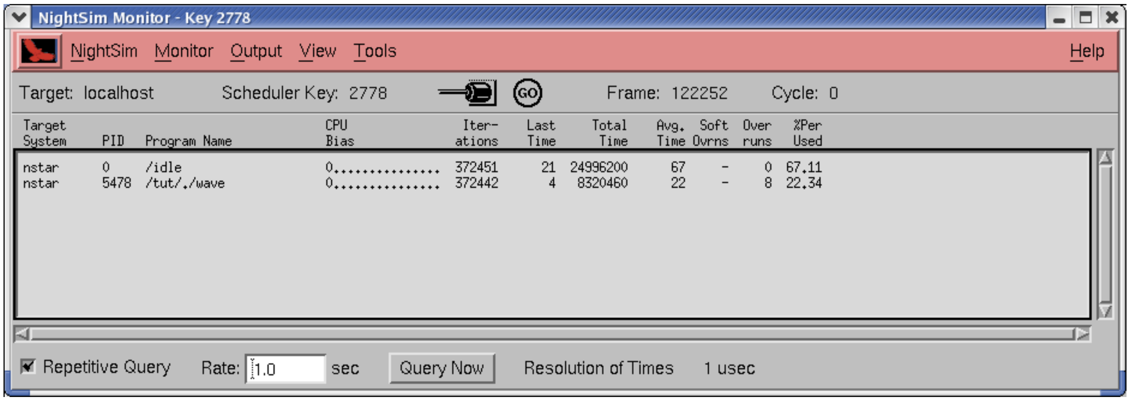


Figure 6-6. NightSim Monitor Window -- Reduced Workload

Experiment with various values of `workload` using the **`set_workload`** program until the average Last Cycle time is approximately 50 microseconds.

Overrun Detection and System Tuning

A scheduling *overrun* occurs when a process's next cycle begins but it has not yet finished execution of its previous cycle.

The NightSim Monitor window includes overrun counts for each process.

It is likely that several overruns have occurred for the wave process.

NOTE

If overruns have not yet occurred, place some additional load on the system. Running the following command in a separate terminal session should have the desired effect:

```
find / -print
```

The NightTrace tool, as described in a previous chapter, is well suited for determining the specific cause of process overruns. NightTrace kernel tracing provides a detailed view of system activity on all CPUs, including process context switches, interrupts, system calls, and machine exceptions.

For brevity, we will assume that the cause of the overruns is due to additional activities unrelated to the scheduler are occurring on the CPU where **`wave`** executes.

We will use NightTune to shield the CPU associated with our scheduler from other activities.

> Launch NightTune using the supplied configuration file which was copied into the current directory during the activities in the section entitled "Creating a Tutorial Directory" on page 1-3 **`ntune -config ntune.config`**

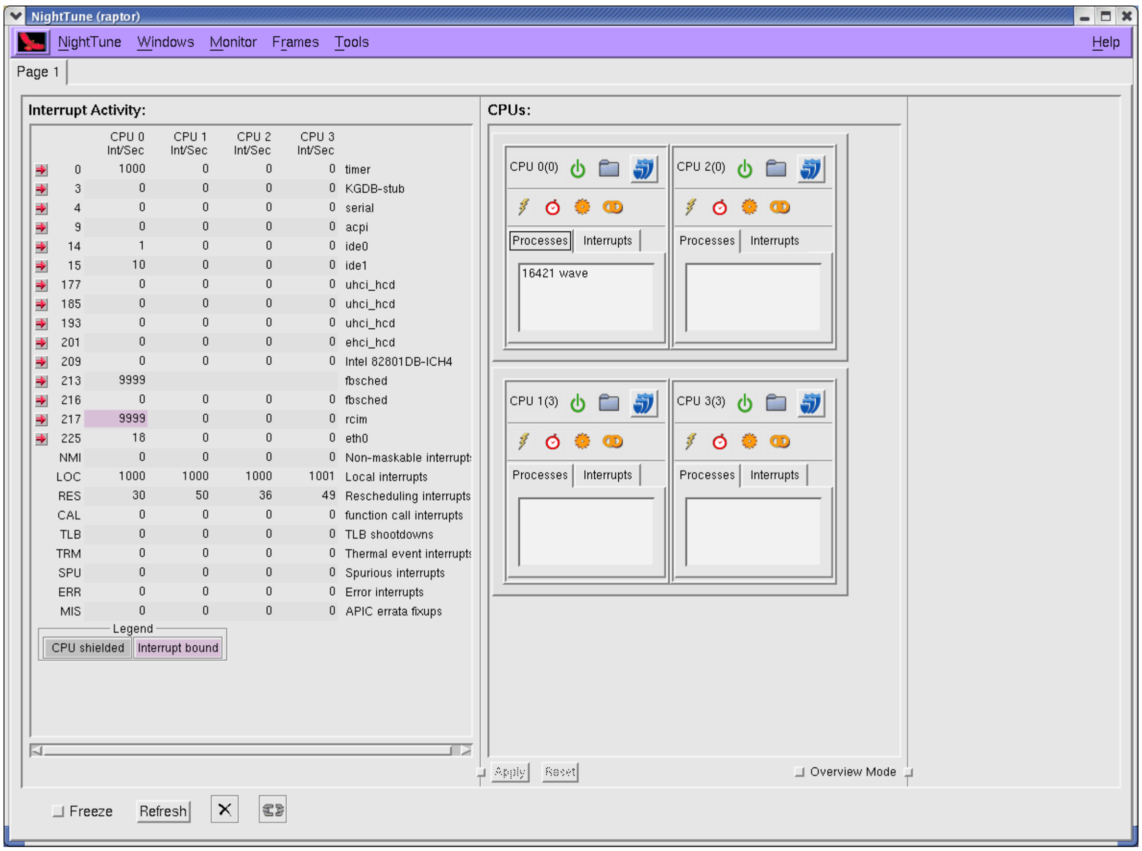


Figure 6-7. NightTune with Interrupt Activity and CPU Panels

A NightTune window appears which displays interrupt activity and the shielding and bound status of all CPUs.

Note that wave process is listed in the Processes area of CPU 0.

Take the following actions to bind the RCIM interrupt to CPU 0 and shield CPU 0 from all other activities:

- Drag the red arrow associated with the row describing the rcim interrupt into the CPU 0 box and release it.
- Click the Maximum Shield button in the CPU 0 box (the maximum shield button is the upper-rightmost button with three overlapping shield figures).

The CPU 0 box changes its display to indicate that all processes and interrupts other than save and rcim will be shielded from CPU 0. Additionally, the sibling hyper-threaded CPU (in this case CPU 2 as shown to the right of CPU 0) is marked down so that hyper-threaded execution on CPU 2 does not interfere with CPU 0.

NOTE:

> Your system may not have hyper-threading enabled in which case the CPUs are displayed in a single column. Furthermore, the hyperthreaded sibling of CPU 0 may be a logical CPU number other than CPU 2.

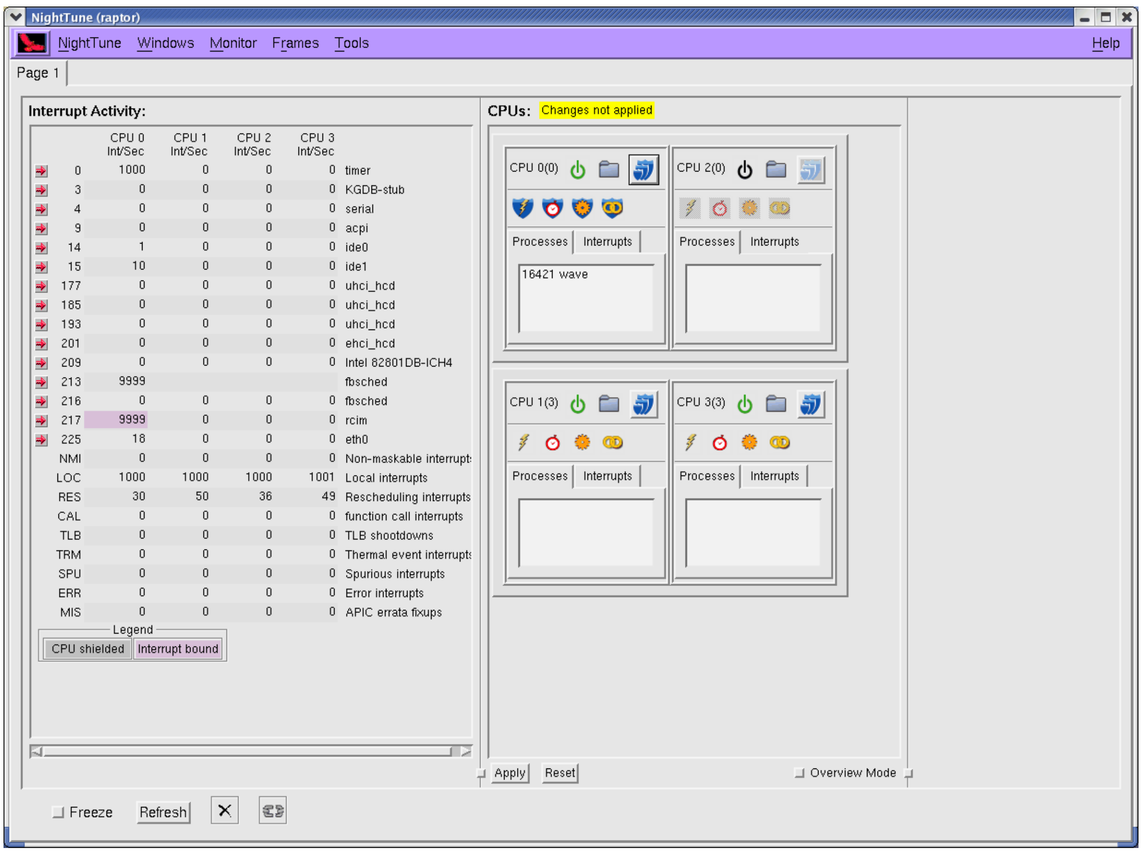


Figure 6-8. NightTune with Shielding Actions Pending

Press the Apply button to activate the shielding changes.

Return to the NightSim Monitor window and watch the Overrun column. It is likely that overruns have ceased to occur. Clear the overrun count by selecting the Clear Performance Values menu item from the Monitor menu and press OK when the CPU dialog is presented. This action resets all the statistics to zero.

Watch the Overrun column to see if any overruns still occur.

Experiment with the **./set_workload** program to make the workload of the **./wave** application such that 40 microseconds are left for **/idle** processing.

If the RedHawk system is properly configured, the scheduler should continue to execute without any overruns on the shielded CPU.

Shutting Down the Scheduler

Return to the NightSim Main window and press the Remove button to terminate the scheduler. Press OK when presented with the dialog which asks whether to kill the processes associated with the scheduler.

Exit NightSim by selecting the Exit menu item from the NightSim menu on the NightSim Main window.

You may also wish to clear the shielding attributes for CPU 0 and return the system to its previous state using NightTune.

This concludes the NightSim portion of the NightTrace RT User's Guide.

A
Tutorial Files

The following sections show the source listings for the files used in the *NightTrace RT User's Guide*.

app.c

```
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>
#include <ntrace.h>
#include <math.h>
#include <sys/ipc.h>
#include <sys/sem.h>

static void * heap_thread (void * ptr);

typedef struct {
   char * name;
   int    count;
   double delta;
   double angle;
   double value;
} control_t;

control_t data[2] = { { "sin", 0, M_PI/360.0, 0.0, 0.0 },
                      { "cos", 0, M_PI/360.0, 0.0, 0.0 } };
int rate = 50000000;

int sema;

extern
double
FunctionCall(void)
{
   return data[0].value + data[1].value;
}

static
void *
sine_thread (void * ptr)
{
   control_t * data = (control_t *)ptr;
   struct sembuf wait = {0, -1, 0};

   trace_open_thread (data->name);

   for (;;) {
      semop(sema, &wait, 1);
      data->count++;
      data->angle += data->delta;
      data->value = sin(data->angle);
   }
}

static
void *
cosine_thread (void * ptr)
{
   control_t * data = (control_t *)ptr;
   struct sembuf wait = {0, -1, 0};

   trace_open_thread (data->name);

   for (;;) {
```

```
      semop(sema, &wait, 1);
      data->count++;
      data->angle += data->delta;
      data->value = cos(data->angle);
   }
}

int
main (int argc, char * argv[])
{
   pthread_t thread;
   pthread_attr_t attr;
   struct sembuf trigger = {0, 2, 0};

   trace_begin ("/tmp/data",NULL);
   trace_open_thread ("main");

   sema = semget (IPC_PRIVATE, 1, IPC_CREAT+0666);

   pthread_attr_init(&attr);
   Pthread_create (&thread, &attr, sine_thread, &data[0]);

   pthread_attr_init(&attr);
   Pthread_create (&thread, &attr, cosine_thread, &data[1]);

   pthread_attr_init(&attr);
   Pthread_create (&thread, &attr, heap_thread, NULL);

   for (;;) {
      struct timespec delay = { 0, rate } ;
      nanosleep(&delay,NULL);
      semop(sema,&trigger,1);
   }

   trace_end () ;
}

void * ptrs[5];

static
void *
heap_thread (void * unused)
{
   int i;
   int scenario = -1;
   void * ptr;
   int * * iptr;

   extern void * alloc_ptr (int size, int swtch);
   extern void free_ptr (void * ptr, int swtch);

   for (;;) {
      sleep (5);
      switch (scenario) {
      case 1:
         // Use of freed pointer
         ptr = alloc_ptr(1024,3);
         free_ptr(ptr,2);
         memset (ptr, 47, 64);
         break;
      case 2:
         // Double-free
         ptr = alloc_ptr(1024,3);
         free_ptr(ptr,2);
         free(ptr);
```

```
         break;
      case 3:
         // Overwriting past end of an allocated block
#define MyString "mystring"
         ptr = alloc_ptr(strlen(MyString),2);
         strcpy (ptr,MyString); // oops -- forgot the zero-byte
         break;
      case 4:
         // Uninitialized use
         iptr = (int * *) alloc_ptr(sizeof(void*),2);
         if (*iptr) **iptr = 2778;
         break;
      case 5:
         // Leak -- all references to block removed
         ptr = alloc_ptr(37,1);
         ptr = 0;
         break;
      case 6:
         // Some more allocations we'll check on...
         ptrs[0] = alloc_ptr(1024*1024,3);
         ptrs[1] = alloc_ptr(1024,2);
         ptrs[2] = alloc_ptr(62,1);
         ptrs[3] = alloc_ptr(4564,3);
         ptrs[4] = alloc_ptr(8177,3);
         break;
      }

      (void) malloc(1);
      scenario = 0;
   }
}

void * func3 (int size, int count)
{
   return malloc(size);
}

void * func2 (int size, int count)
{
   if (--count > 0) return func3(size,count);
   return malloc(size);
}

void * func1 (int size, int count)
{
   if (--count > 0) return func2(size,count);
   return malloc(size);
}

void free3 (void * ptr, int count)
{
   free(ptr);
}

void free2 (void * ptr, int count)
{
   if (--count > 0) {
      free3(ptr,count);
      return;
   }
   free(ptr);
}

void free1 (void * ptr, int count)
{
```

```
   if (--count > 0) {
      free2(ptr,count);
      return;
   }
   free(ptr);
}

void * alloc_ptr (int size, int count)
{
   return func1(size,count);
}

void free_ptr (void * ptr, int count)
{
   free1(ptr,count);
}
```

report.c

```
void report (char * caller, double value)
{
   static int count;

   if (++count % 40) printf ("The value from %s is %f\n", caller, value);
}
```

function.c

```
double
FunctionCall(void)
{
   static double counter;
   return counter++;
}
```

wave.c

```
#include <fbsched.h>

int workload = 1000;

main()
{
   int data = 0;
   int i;
   volatile double d = 1.0;

   while (fbswait()==0) {
      data = !data;
      for (i=0; i<workload; ++i) d = d/d;
```

```
   }

}
```

set_workload.c

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

#define check(x) if((x)) {fprintf(stderr, "%s\n",
dm_get_error_string());exit(1);}

main(int argc, char * argv[])
{
   program_descriptor_t pgm;
   object_descriptor_t   obj;
   char buffer[100];

   if (argc != 2) {
      fprintf (stderr, "Usage: set_workload integer-value\n");
      exit(1);
   }

   check(dm_open_program("wave",0,&pgm));
   check(dm_get_descriptor("workload",0,pgm,&obj));
   check(dm_get_value(&obj,buffer,sizeof(buffer)));
   check(dm_set_value(&obj,argv[1]));

   printf ("workload: old_value=%s, new_value=%s\n", buffer, argv[1]);
}
```