# Oracle® Solaris Studio 12.3: DLight Tutorial

December 2011

ORACLE®

The DLight Tutorial shows how to use DLight to launch an executable and use the DLight monitoring graphs and tools to examine the profiling data from the running program. The tutorial also shows how to attach to a process and examine profiling data from the process and all its child processes and threads. The tutorial includes the following topics:

# Introduction

DLight is an interactive graphical observability tool that uses Oracle Solaris Dynamic Tracing (DTrace) technology to gather information about programs running on Oracle Solaris. DLight enables you to analyze programs with DTrace, without needing to know how to use the D scripting language. DLight runs multiple DTrace scripts in a synchronized fashion, while showing you the output graphically, to help you trace a runtime problem in an application to the root cause.

The basic workflow of DLight is as follows:

1. Create a target to define the process or executable you want to profile
2. Run the target
3. Examine the data in the DLight tools that open when you run the target
4. Interact with the tools to view problem points in the code
5. Stop the target if necessary

When you create a target you can specify the type of target and the system where the target should run (local or remote system).

DLight Targets types are:

**Executable Target**      Program that is not yet running

**Process Target**      Running process

**Process Tree Target**      Running process and all child processes spawned by that process

When you run an Executable Target or Process Target, each tool displays usage information in a graph in the DLight Run Monitor window.

When you run the Process Tree target, tools display in Process Tree Profiling windows that offer more information about the activity of the multiple processes and threads.

The tools in the Run Monitor window and Process Tree Profiling window are described later in this document.

You must have specific DTrace-related Oracle Solaris privileges on the system where you run DLight, as described in the next section.

---

**Tip** – See Help ⇒ Help Contents for information about targets, profiling processes on remote hosts, and changing tool configuration.

---

# Setting Oracle Solaris Privileges for DLight

DLight requires the user who runs it to be assigned Oracle Solaris privileges: `dtrace_proc`, `dtrace_user`, and `dtrace_kernel`. These privileges control a user's access to DTrace features. If your username does not have these privileges assigned, DLight prompts you for the password for an administrator account such as root that can set privileges. This prompt occurs the first time you run an executable target or attach to a target process in the DLight session.

After you supply the correct password, the administrator account is used to assign the necessary privileges to the process that runs DLight for the duration of the DLight session. If you use DLight on a system where you have these privileges assigned to your username permanently or log in as an administrator, DLight does not prompt for an administrator password.

---

**Note** – These privileges should be used with care because they enable the user to look inside the running of the kernel. You should only enable these privileges for users on systems that are used for software development, and not on production systems.

---

To check your Solaris privileges, type the following at the command prompt:

$ **/bin/ppriv $$**

If your account has the required privileges, the `ppriv` command should return something similar to:

```
E: basic,dtrace_kernel,dtrace_proc,dtrace_user
I: basic,dtrace_kernel,dtrace_proc,dtrace_user
P: basic,dtrace_kernel,dtrace_proc,dtrace_user
L: all
```

The line that starts with `I:` is important because it specifies privileges that are inherited by programs started from your shell. If your account doesn't have the required inheritable privileges, and you do not have administrator privileges or root access to your system, you should ask your system administrator to add the `dtrace_proc`, `dtrace_user`, and `dtrace_kernel` inheritable privileges to your account.

If you have administrator privileges or root access to your system, you can grant your user account the required privileges as described below.

To permanently grant required DTrace privileges to a user account:

1. Make sure the user account whose privileges you want to modify is logged out of the system.

2. Log in as superuser (root) or another administrator user.

3. Type the following at the command prompt, and replace *username* with the user account name you are modifying:

    $ **usermod -K defaultpriv=basic,dtrace_kernel,dtrace_user,dtrace_proc** *username*

You can also assign privileges to a user's running shell process to temporarily grant these enhanced privileges to the user.

To temporarily grant required DTrace privileges to a user account:

1. Type the following at the user's command prompt to determine the process ID of the user's shell process:

    $ **echo $$**

2. Log in as superuser (root) or another administrator user in another terminal.

3. Type the following, and replace *process-ID* with the process ID that was returned from the `echo` command.

    $ **ppriv -s I+dtrace_user,dtrace_proc,dtrace_kernel** *process-ID*

    All commands typed in the shell specified by *process-ID* now inherit the required privileges. The user should start DLight in this shell.

# Downloading the Oracle Solaris Studio Sample Applications

This tutorial uses a sample program called `ProfilingDemo`.

The source code for the sample program is available in the sample applications zip file on the Oracle Solaris Studio 12.3 Sample Applications web page (`http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/solaris-studio-samples-1408618.html`).

If you have not already done so, download the sample applications zip file and unpack it in a directory of your choice. The sample application is located in the DLight subdirectory of the `SolarisStudioSampleApplications` directory.

# Building the Sample Application

This procedure assumes you have already downloaded the sample applications zip file and unpacked it in a directory of your choice.

1. Copy the `ProfilingDemo` directory to your own private working area, such as your home directory:

   ```
   % cp -r SolarisStudioSampleApplications/DLight/ProfilingDemo ~/ProfilingDemo
   ```
2. Build your copy of the program:

   ```
   % cd ~/ProfilingDemo
   % make
   ```

   The program `profilingdemo` is built using the `-g` option, which generates debug information. If you compile without this option, DLight can collect and display run time data for the program, but the features that let you display the source code for a function will not work.

# Starting DLight

Start DLight by typing:

```
% dlight
```

If you did not add the Oracle Solaris Studio directory to your path, you can start DLight by typing:

```
% /studio-installation-directory/bin/dlight
```

The default installation directory is `/opt/solarisstudio12.3/bin`.

# Profiling the Sample Application

The DLight Executable Target enables DLight to launch and profile an application that is not yet running. In this section, you create and run an Executable Target and observe the dynamic graphs displayed by DLight as the sample application runs.

1. In DLight, click the New DLight Target button ![icon]. The Create New Executable Target dialog box opens with the Executable Target tab selected.
2. In the Create New Executable Target dialog box:
   a. Type the pathname to your `profilingdemo` executable in the Run field, or click Browse to navigate to your `profilingdemo` executable file and open it.
   b. Click Run.

The new Executable Target is saved and listed in the DLight Targets window, and DLight runs the target.

c. If you do not have sufficient privileges for DTrace as explained in "Setting Oracle Solaris Privileges for DLight" on page 3, you are prompted for your root password. You can also use non-root administrator user's credentials if you replace the root username with the appropriate administrator name.

3. The profilingdemo application begins executing and the Run Monitor window displays dynamic graphs of profiling data gathered about the running application.

---

**Tip** – If you cannot see all the graphs, close the Host Info window. You can also use the scroll bars on the side of the Run Monitor window to view more of the graphs.

---

4. In the Output window, the profilingdemo program tells you what it is doing so you can match the output to the data represented in the graphs. Click the Output window and press Enter each time the program requests it until execution is finished.

## Using the Slider Controls

At the bottom of the DLight Run Monitor window, you can see sliders for controlling your view of the graphs: View slider, Details slider, and Time slider.



1. Place your mouse cursor over the end points of the sliders for tooltip information about each of the sliders.

2. Click and hold your mouse cursor on the Time slider and drag the slider to the left to see the beginning of the run. All the graphs slide in unison so you can see what happened in each area (CPU, memory, threads, I/O) at any given time, and see the relationships between them.

3. Drag the Time slider from left to right to see the complete run.

4. Move your mouse cursor to the View slider, the control that overlays the time units. The View slider controls let you select which part of the run time is displayed in the graphs.

5. Click and drag the left handle of the View slider, the start point, all the way to the beginning of the run at 0:00. The graphs now show the entire run at once. The effect is similar to zooming out as far as possible. Notice that the Time slider is not functional when you select the complete run time. You are already looking at all the data so there is nothing left to scroll.
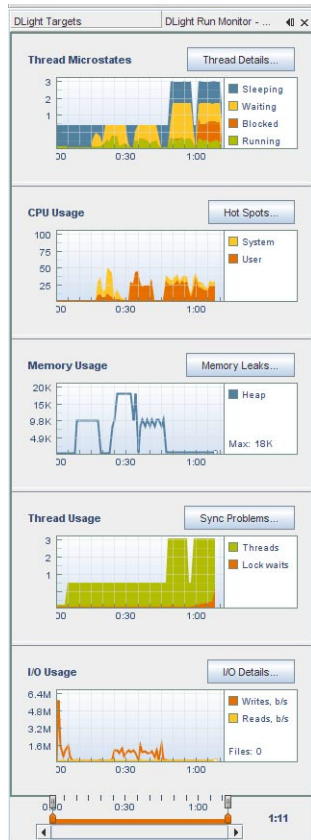
6. The View slider can be used for zooming. Drag the start point of the View slider to the right. As you drag the handle, the graphs zoom in to focus on the area toward the end of the run. Notice that the Time slider can again be used to scroll back and forth in the run time.

7. Place your mouse cursor over the end points of the orange colored Details slider for a description of how to use the slider. The Details slider controls enable you to select a portion of the run time to examine in detail.

8. Drag the start point of the Details slider to a later time than the start point of the View slider. Notice that the graphs are grayed out in the area in front of the start point, giving a highlight effect to the graph between the start and end points.
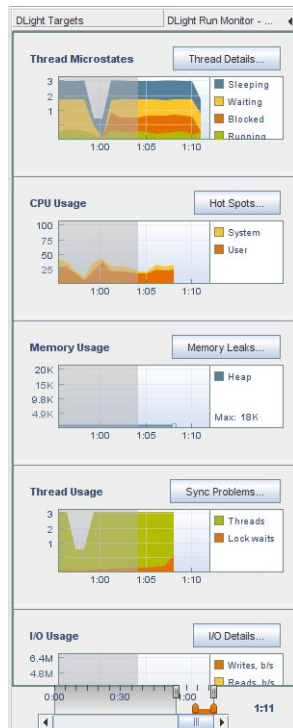
9. When you click on any of the graphs' detail buttons (Thread Details, Hot Spots, Memory Leaks, Sync Problems, or I/O Details), the data for the highlighted area is shown in a details tab. In other words, the Details slider is used to select the data to be shown in the Details tabs.

10. Drag the start point of the View slider back to the beginning of the run so you can see all the data.

## Exploring Thread Microstates

The Thread Microstates graph shows an overview of the program's threads as they enter various execution states during the program's run. The Solaris microstate accounting feature uses the DTrace facility to provide fine-grained information about the state of each thread as it enters and exits ten different execution states:

| | |
|---|---|
| **User running** | The percentage of time the process has spent in user mode |
| **System running** | The percentage of time the process has spent in system mode |
| **Other Running** | The percentage of time the process has spent in processing system traps and such |
| **Text page fault** | The percentage of time the process has spent in processing text page faults |
| **Data page fault** | The percentage of time the process has spent in processing data page faults |
| **Blocked** | The percentage of time the process has spent waiting for user locks |
| **Sleeping** | The percentage of time the process has spent sleeping |
| **Waiting** | The percentage of time the process has spent waiting for CPU |

The Thread Microstates tool graphically shows summarized state information for all the threads that are created during the program's run. Only four states are shown: Sleeping, Waiting, Blocked, and Running. These states represent a simplified or summary view of the ten possible microstates, and give an overview of the states of all the threads running in your program. For example, the time spent in the Running state represents all types of running states: running in user mode, running in system calls, running in page faults, running in traps.

1. Move the left handle of the View slider to the left until the graphs show about 20 seconds of the run time as illustrated below. This image shows the beginning of the program's run, during the SEQUENTIAL DEMO portion, when two tasks are run one after the other in a single thread. The points where the thread is sleeping correspond to the points where the program is waiting for the user to press Enter.
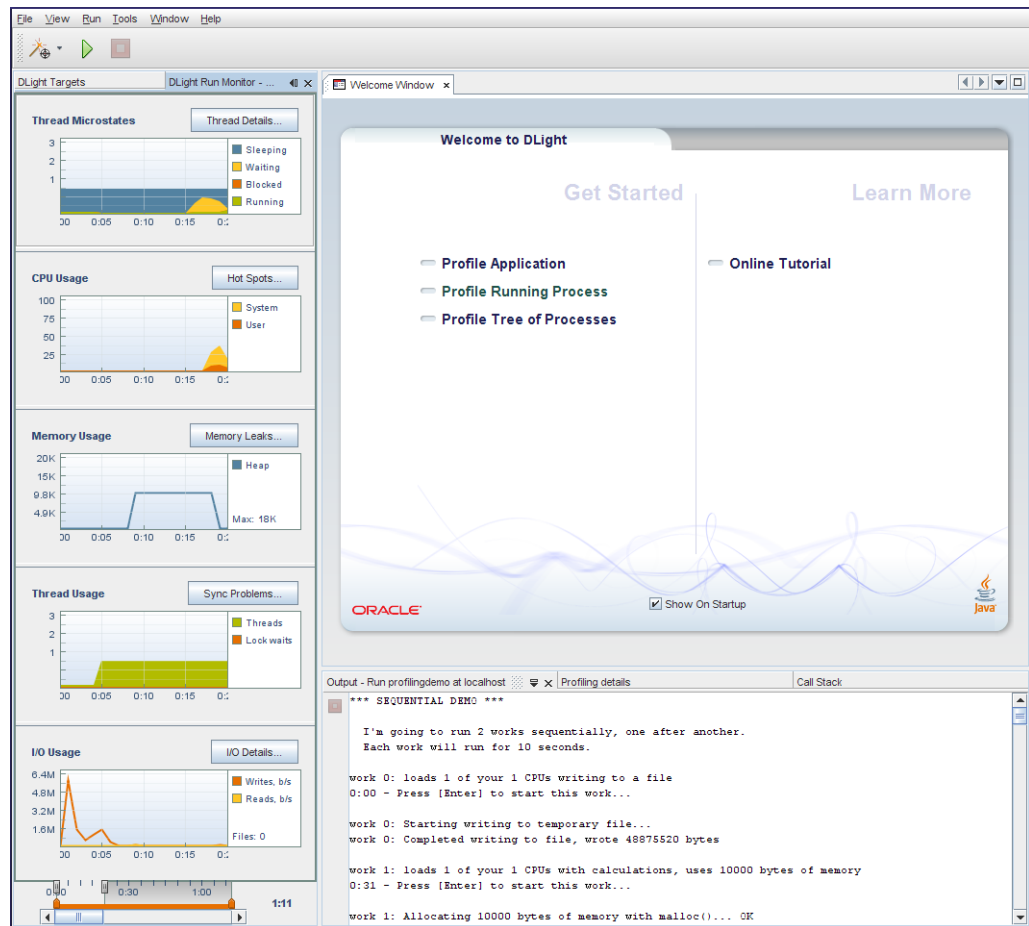
2. Click and drag the Time slider to the right until the number of threads shown in the Thread Microstates tool jumps to three.

3. The number of threads jumps to three as the program enters the PARALLEL DEMO portion. The main thread launches two additional threads to run two tasks in parallel, each in its own thread. You can see that considerable time is spent in the Waiting state (yellow) and the Sleeping state (blue), and not as much time spent in the Running state (green). No time is spent in the Blocked state (orange) during the PARALLEL DEMO portion because this portion of the program does not implement any thread synchronization tactics such as mutex locks that would block threads.
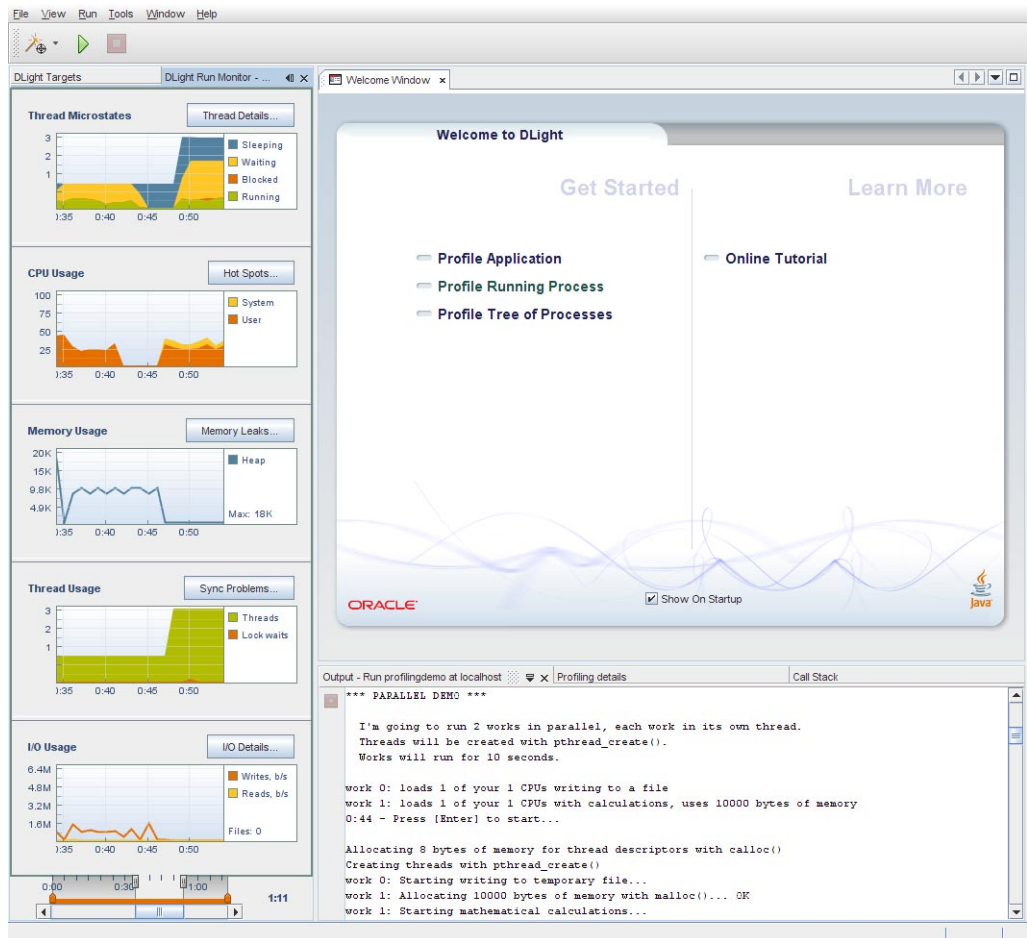
4. Drag the Time slider all the way to the end of the run time. Notice that the Blocked microstate shown in orange appears at the point where the program enters the PTHREAD MUTEX DEMO portion, in which each thread uses mutual exclusion locks to prevent other threads from interfering at certain points. Each thread can run actively only after it obtains the mutex lock. A thread is blocked when it tries to access a locked section of code when another thread owns the mutex lock. The use of mutual exclusion locks prevents the threads from entering a data race condition where threads have overlapping access to the same data.
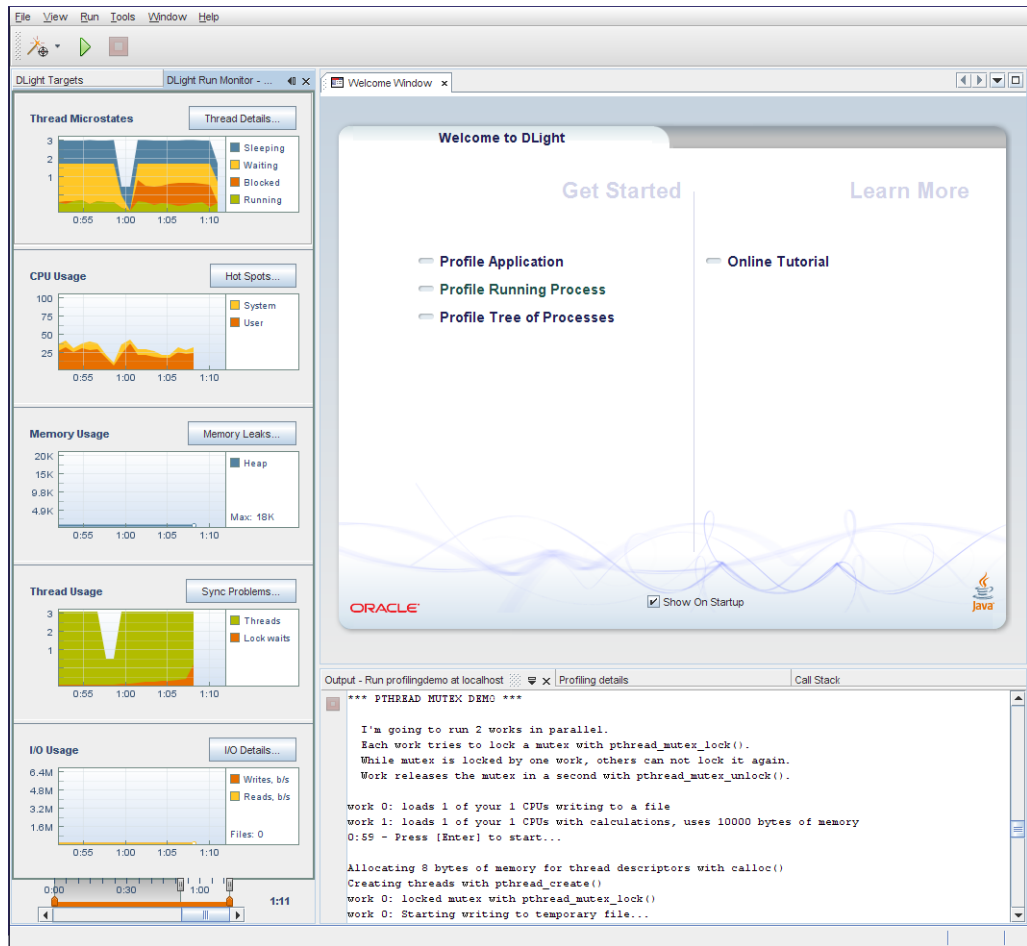
5. Click the Thread Details button to display details about the thread microstates. The Thread Details window opens to display a graphical timeline representation of all the threads run in the program along with detailed state information.
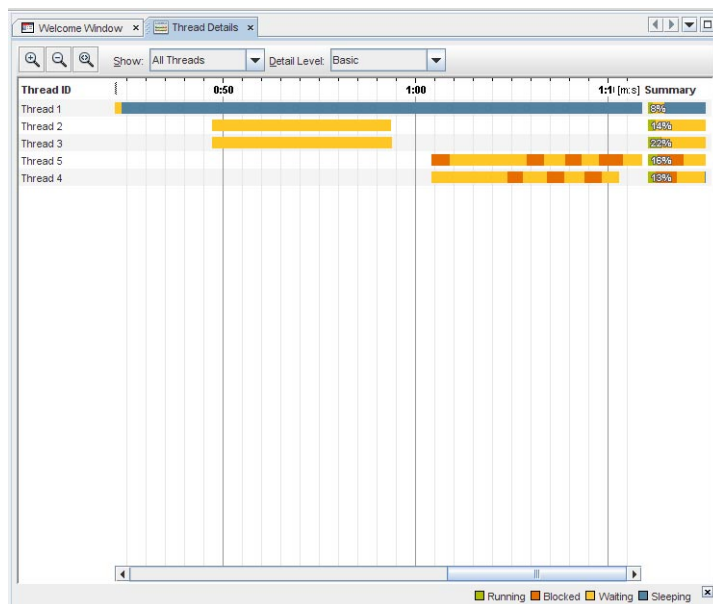
The Thread Details window shows the state transitions for each thread during the complete run time of the program.

6. Put your cursor on one of the colored areas of a thread. A popup window displays details about what is going on with that thread at that particular moment. Details include the time when the data was collected and the percentage of time spent in each thread state at that moment. When you put your cursor over the Summary area at the right side of the window, the popup window displays the percentage in each state for the thread's complete run.
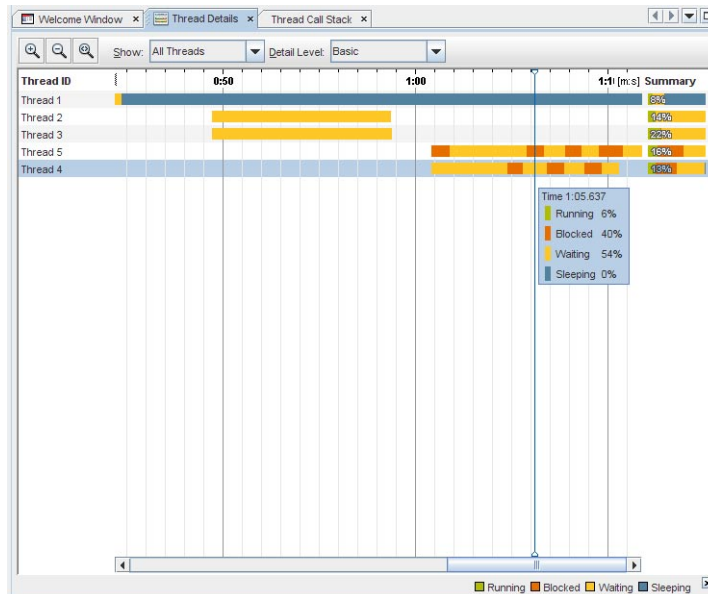


7. Try using the window's controls to change what is shown:

   ■ By default, the window shows all threads. Click the downward arrow to the right of the Show drop-down list. You can select Live Threads only to show only threads that have not terminated, or Finished Threads only to show only those threads that have terminated during the program run.

   ■ By default, the window shows only the four generalized execution states. Click the downward arrow to the right of the Detail Level drop-down list. You can select Moderate to display more detail on these states, or Advanced to display ten microstates.

   ■ Click an individual thread and notice that the thread is highlighted. If you press Shift and click another thread, a range of threads is selected. To select multiple threads that are not adjacent, press Ctrl before selecting the threads. When the threads you are interested in are highlighted, right-click and select Show Only Selected Threads. To see all the threads again, select All Threads from the Show drop-down list.

   ■ Right-click on any thread and select Thread Name ⇒ Thread Entry to change the displayed thread names to the functions entered when the threads began executing.

8. Zoom in on the thread graphs to take a closer look by clicking the Zoom In button ⊕ . This image shows the window zoomed in, with the Detail Level set to Advanced.
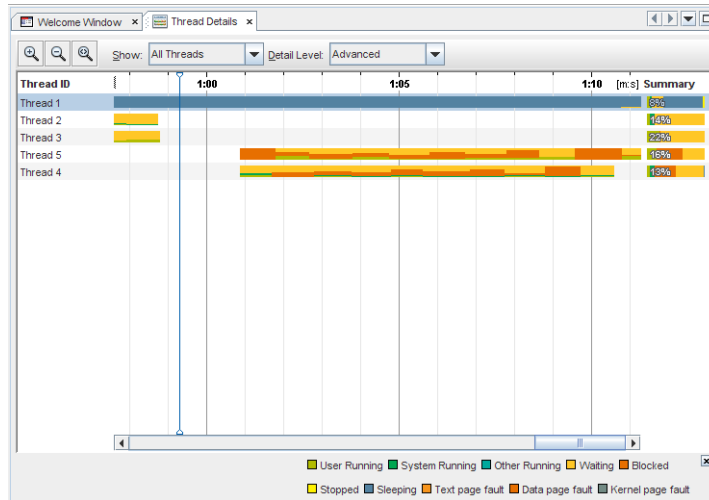
9. Click the Zoom Out button 🔍 to go back to the previous zoom level.

10. Click the Show Complete Run button 🔍 to display the complete run in the Thread Details window at once. You can click the button again 🔍 to return to the scrolling view of the thread details.

11. Click the second orange rectangle on thread 4. The Thread Call stack tab opens to show the call stack for the thread at this moment. You can expand a node in the stack to see the calls happening in that thread, or right-click the top node and choose Expand All to see the calls in all of the threads.
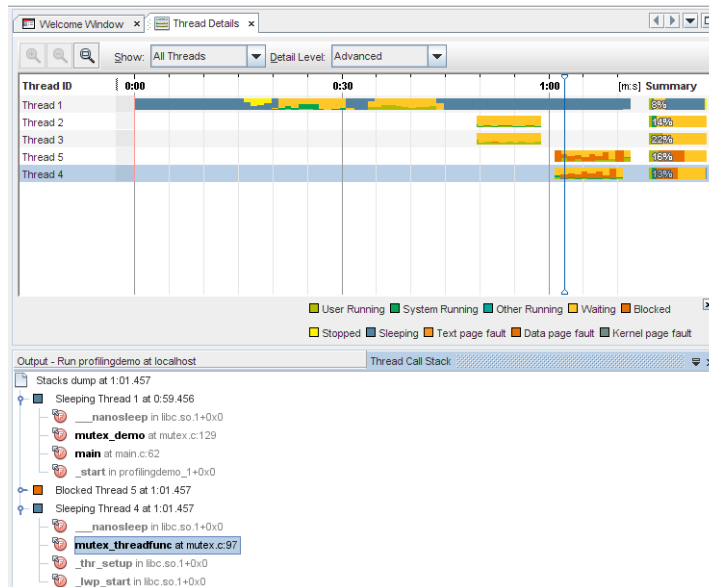


12. Double-click the `mutex_threadfunc` function to open the source file where the function is called. (You can display the calling source file for any function in the stack that is not grayed out.)
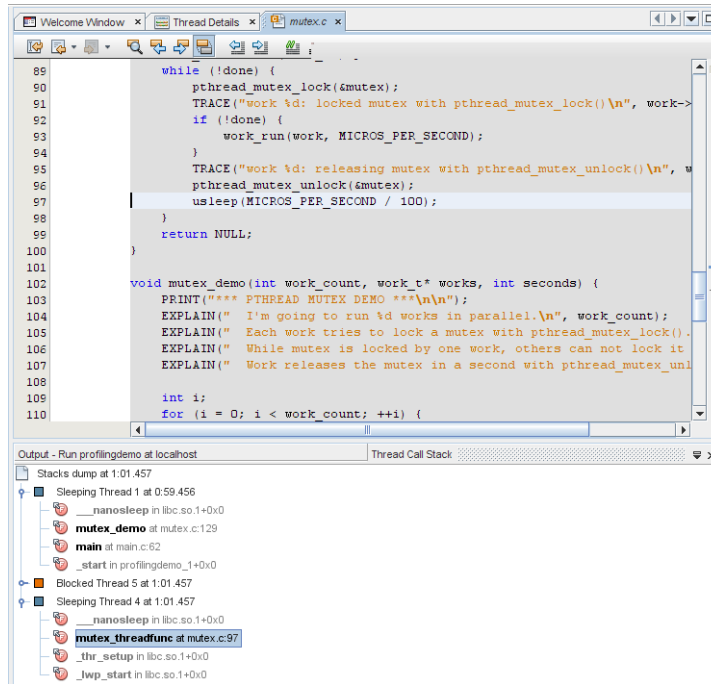
13. Click the Thread Details tab to return to the Thread Details window. Click on a thread. You can navigate along the thread's timeline using your mouse or your keyboard:

- With your mouse, right-click the thread and use the Navigate menu to move the focus to the left or right, set a point on the timeline and update the content of the Thread Call Stack tab, or switch focus to the Thread Call Stack tab.

- To use keyboard shortcuts to navigate the thread, press the following keys:

  - Ctrl+LeftArrow and Ctrl+RightArrow to scroll left and right in the thread timeline
  - Ctrl+DownArrow to select a point in the timeline, which will update the Thread Call Stack
  - Alt+DownArrow to focus input on the Thread Call Stack window

- In the Thread Call Stack you can use the arrow keys and Enter to open source files associated with the functions

## Exploring CPU Usage

The CPU Usage graph shows the percentage of the total CPU time used by your application during its run.

1. Click the Hot Spots button to display details about the CPU time. The CPU Time Per Function tab opens to display the functions of the program, along with the CPU time used by each function. The functions are listed in order of CPU time used, with the functions that use the most time listed first. If the program is still running, the time initially displayed is the amount of time consumed at the moment you clicked the button.



2. Click the header of the Function Name column to sort the functions alphabetically.

3. Click the header of the CPU Time (Exclusive) column to sort the functions by the order of time used by the individual functions.

4. Notice the difference between the two columns of CPU Time. CPU Time (Inclusive) shows the total CPU time spent from the time the function is entered until the time it is exited, including the time of all other functions that are called by the listed function. CPU Time (Exclusive) shows the time used only by the specific function, not including any functions that it calls.

5. Click the CPU Time (Inclusive) column header to place the most time-consuming function back at the top. Notice that the work_run_usrcpu function used slightly more CPU Time (Inclusive) than CPU Time (Exclusive), which means that a small amount of its CPU time was actually used by other functions that it calls, but that the work_run_usrcpu function itself used most of the time.

6. Some of the functions are shown in bold text. You can display the source files for those functions. Double-click the work_run_usrcpu function. The common.c file opens, with the cursor resting on the work_run_usrcpu function, line 59.
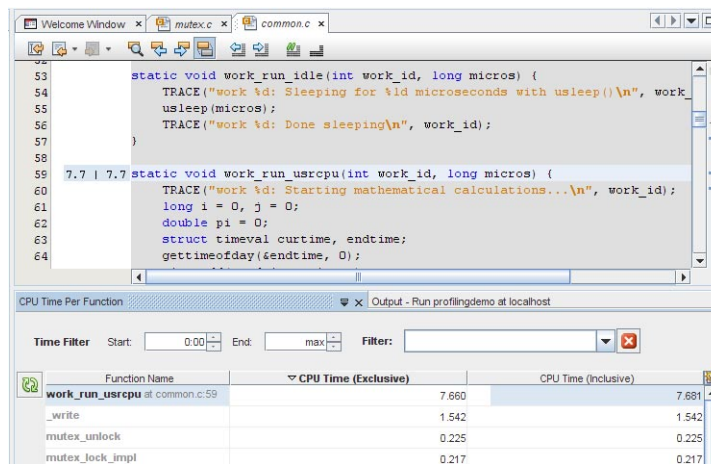


7. Place your mouse cursor over the numbers in the left margin. The numbers are the same metrics for exclusive and inclusive CPU time for the function that are displayed in the CPU Time Per Function tab. The metrics are rounded to use less space, but the unrounded values are shown when you mouse over them. Metrics for CPU-consuming lines such as the for loop that does calculations within the work_run_usrcpu function are also shown in the common.c source file.
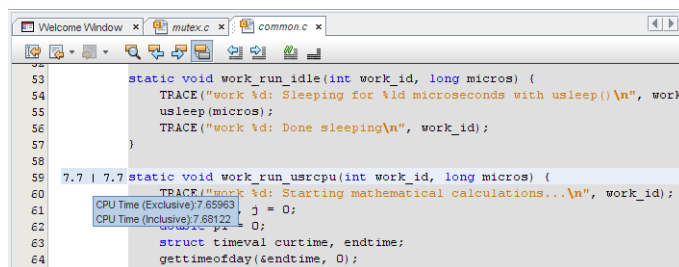


8. Change the Time Filter start time in the CPU Time Per Function tab to 0:30 by either typing the time and pressing Enter, or using the arrows to scroll through the seconds. The graphs in the Run Monitor window change just as they did when you moved the handles on Details slider. If you drag the handles, the Time Filter settings in the CPU Time Per Function tab are updated to match. More importantly, the data shown for the functions in the tab is updated to reflect the filter so only the CPU time used during that time period is shown.
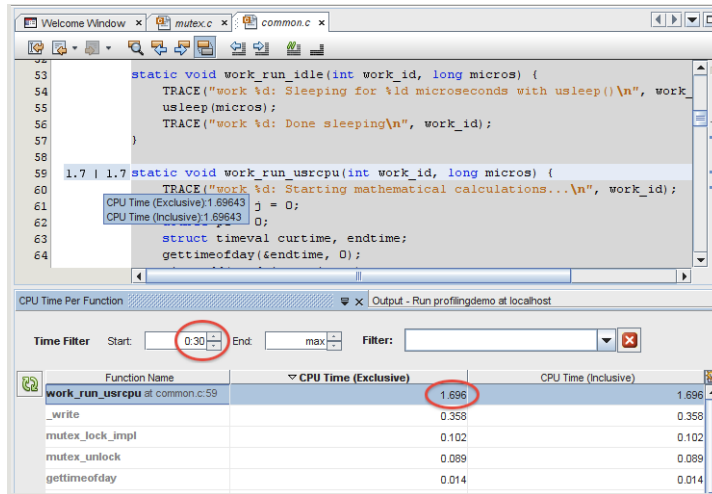
9. You can also filter for data that meets a certain metric. Right-click on the CPU Time (Exclusive) metrics for `work_run_usrcpu`. Select Show only rows where > CPU Time (Exclusive) == the metric shown for `work_run_usrcpu`. All the other rows are filtered away, and only the row whose exclusive CPU time is equal to that metric is displayed.
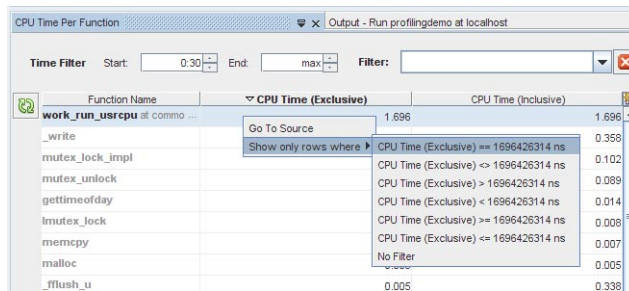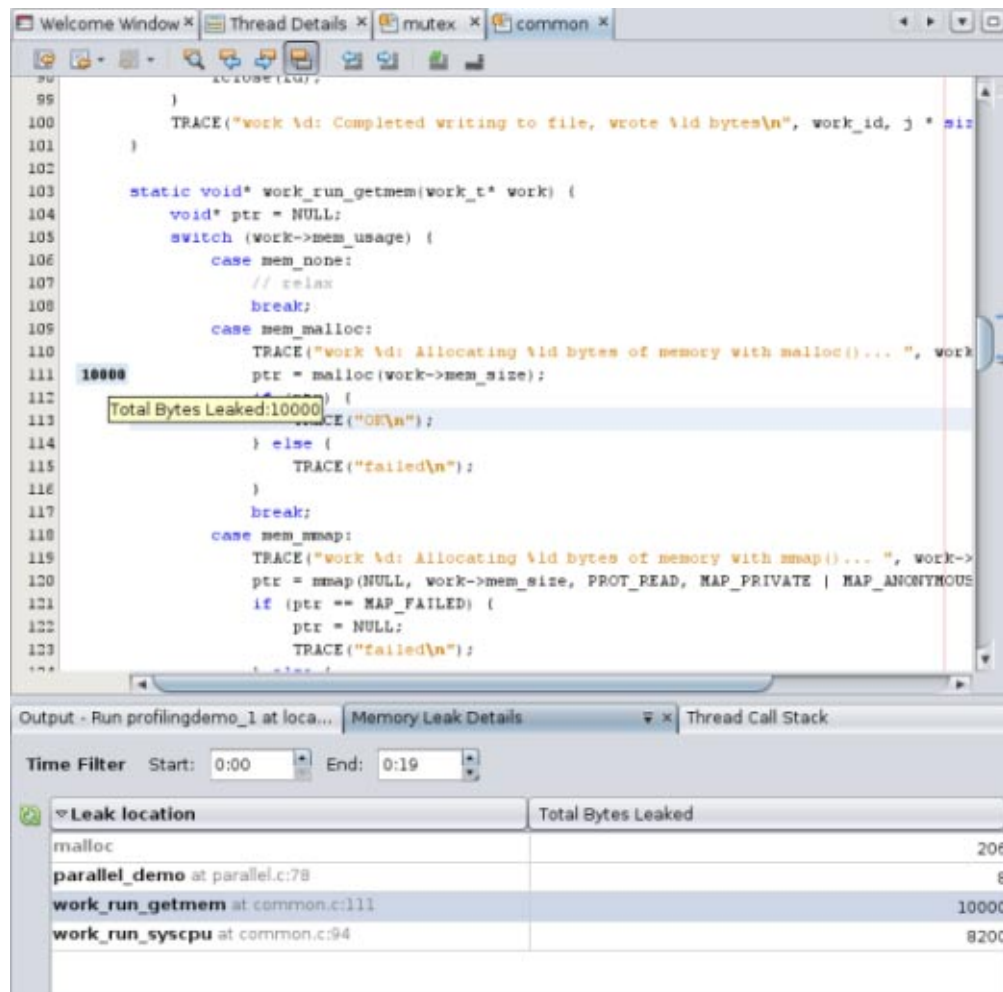


## Exploring Memory Usage

The Memory Usage tool shows how your program's memory heap changes over its run time. You can use it to identify memory leaks, which are points in your program where memory that is no longer needed fails to be released. Memory leaks can lead to increased memory consumption in your program. If a program with a memory leak runs long enough, it can eventually run out of usable memory.

1. Slide the Time slider in the Run Monitor to the left and right to see how the memory heap increases and decreases over time. There are four spikes in usage in the run of the program. The first two occur during the SEQUENTIAL DEMO, the third occurs during the PARALLEL DEMO, and the fourth occurs during the PTHREAD MUTEX DEMO.

2. Click the Memory Leaks button to display the Memory Leak Details tab, which shows details about which functions exhibit memory leaks. Only functions that are producing memory leaks are listed in the tab. If your program is running when you click the button, the leak locations shown are those that exist at the moment you clicked the button. More leaks may occur as time goes on, so click the Refresh button  to update the list. If no memory leaks are detected by the end of the run, the Memory Leak Details tab indicates that no memory leaks were found.

3. You can filter the data by changing the Start and End times in the Memory Leak Details tab, or by using the Details slider in the Run Monitor window.

4. In this run, the `ProfilingDemo` program shows a memory leak associated with a `work_run_getmem` function. Double-click the `work_run_getmem` function and the `common.c` file opens with the cursor at the line where the memory leak occurs in the function.

5. The memory leak metrics are displayed in the left margin. Mouse over them to display the details as you did with the CPU Usage metrics.



6. As with the CPU Time Per Function tab, you can right-click the metrics in the Memory Leak Details tab and select a criterion for filtering the data.
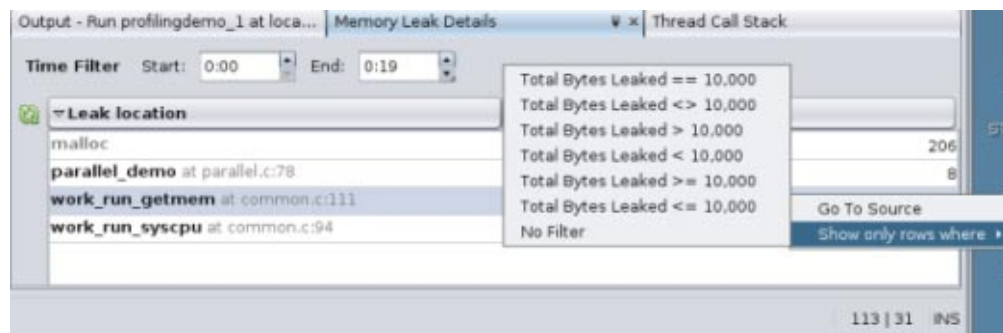


## Exploring Thread Usage

The Thread Usage Tool shows the number of threads in use by your program, and any moments where a thread has to wait to get a lock in order to proceed with its task. This data is useful for multithreaded applications, which must perform thread synchronization in order to avoid expensive wait times.

1. Slide the Time slider to the beginning of the run and notice, as you did for the Thread Microstates graph, that the number of threads is one during the `SEQUENTIAL DEMO` portion of the program but increases to three as the program enters the `PARALLEL DEMO` portion.

2. Move the endpoint handle of the View slider so that you can see the data from the beginning of the run until just before the two additional threads are launched.

3. Look at the CPU Usage and Memory Usage graphs for the same time period and notice that the single thread is performing some activity that uses CPU time and memory. This period corresponds to the `SEQUENTIAL DEMO` portion of the program, in which the main thread writes to the file and then performs some calculations sequentially. CPU and memory usage both decrease while the program waits for the user to press Enter, and the number of threads remains at one.
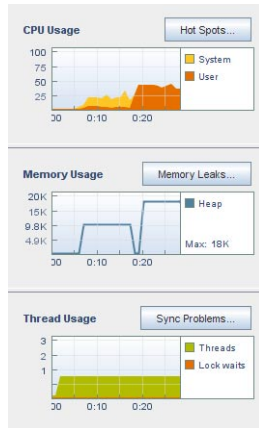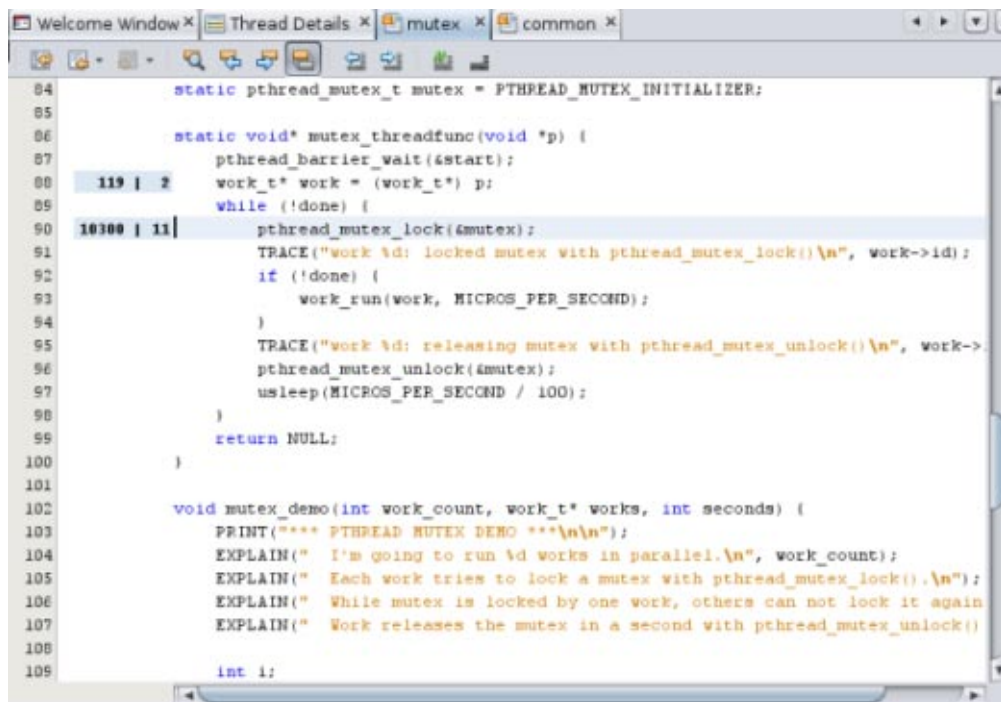


4. Slide the Time slider to the right so that you can see two points where the threads increase to three. The first increase in threads corresponds to the `PARALLEL DEMO` portion of the program run, where the main thread starts two additional threads to do the work of writing to a file and performing calculations, in parallel. Notice that the memory usage and CPU usage are a bit higher during this portion, but the two tasks are completed in much less time than in the `SEQUENTIAL DEMO` portion.
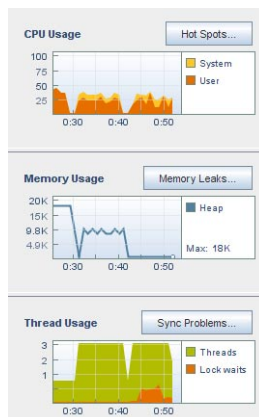


5. Notice that the number of threads returns to one after the `PARALLEL DEMO` threads finish, and the main thread waits for the user to press Enter.

6. The thread count increases to three again as the `PTHREAD MUTEX DEMO` portion of the program runs. Notice that shortly after the thread count increases to three, a lock wait appears, shown in orange. The `PTHREAD MUTEX DEMO` uses mutual exclusion locks to prevent overlapping access to certain functions by multiple threads, which causes the threads to wait to obtain a lock.

7. Click the Sync Problems button to display details about thread locks. The Thread Synchronization Details tab opens and lists functions that had to wait to obtain a mutex lock. Also displayed are metrics for the number of milliseconds that the function spent waiting and the number of times the functions had to wait for a lock.

8. If you click the Sync Problems button while the program is running, you might need to click the Refresh button ⟳ to update the display with the latest thread lock.

9. Click the header of the Wait Time column to sort the functions in order of time spent waiting.

10. Click the header of the Lock Waits column to sort the functions by the number of times a thread was waiting in the functions.

11. Double-click the `mutex_threadfunc` function, which has the most lock waits. The `mutex.c` source file opens with the cursor at the line where the `pthread_mutex_lock` function is called. This function is responsible for locking a memory location before the location is read or written, and must wait until no other thread has a lock on that memory location.

```
84              static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
85
86              static void* mutex_threadfunc(void *p) {
87                      pthread_barrier_wait(&start);
88      119 | 2        work_t* work = (work_t*) p;
89                      while (!done) {
90    10300 | 11            pthread_mutex_lock(&mutex);
91                          TRACE("work %d: locked mutex with pthread_mutex_lock()\n", work->id);
92                          if (!done) {
93                              work_run(work, MICROS_PER_SECOND);
94                          }
95                          TRACE("work %d: releasing mutex with pthread_mutex_unlock()\n", work->
96                          pthread_mutex_unlock(&mutex);
97                          usleep(MICROS_PER_SECOND / 100);
98                      }
99                      return NULL;
100             }
101
102             void mutex_demo(int work_count, work_t* works, int seconds) {
103                     PRINT("**** PTHREAD MUTEX DEMO ***\n\n");
104                     EXPLAIN("  I'm going to run %d works in parallel.\n", work_count);
105                     EXPLAIN("  Each work tries to lock a mutex with pthread_mutex_lock().\n");
106                     EXPLAIN("  While mutex is locked by one work, others can not lock it again
107                     EXPLAIN("  Work releases the mutex in a second with pthread_mutex_unlock()
108
109                     int i;
```

12. The metrics for Wait Time and Lock Waits are displayed in the left column margin of the source file. Place your mouse cursor over the metrics to see the details, which match what is shown in the Thread Synchronization Details tab.

13. Right-click on the metrics and deselect Show Profiler Metrics. The metrics are no longer displayed in the source editor for any of the profiling tools.

14. Choose View ⟹ Show Profiler Metrics to display the metrics again.

## Exploring I/O Usage

The I/O Usage tool shows an overview of the program's read and write activity during the run.

1. The following image shows the I/O usage at the beginning of the run, during the SEQUENTIAL DEMO portion in which two tasks are run one after the other in a single thread. In the first few seconds the program started, and then it was waiting for the user to press Enter. When the user pressed Enter, the program wrote characters to a temporary file. This activity is reflected in the graph as the orange line showing bytes written.
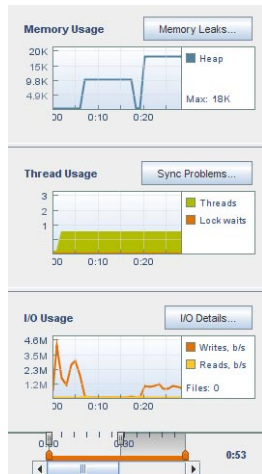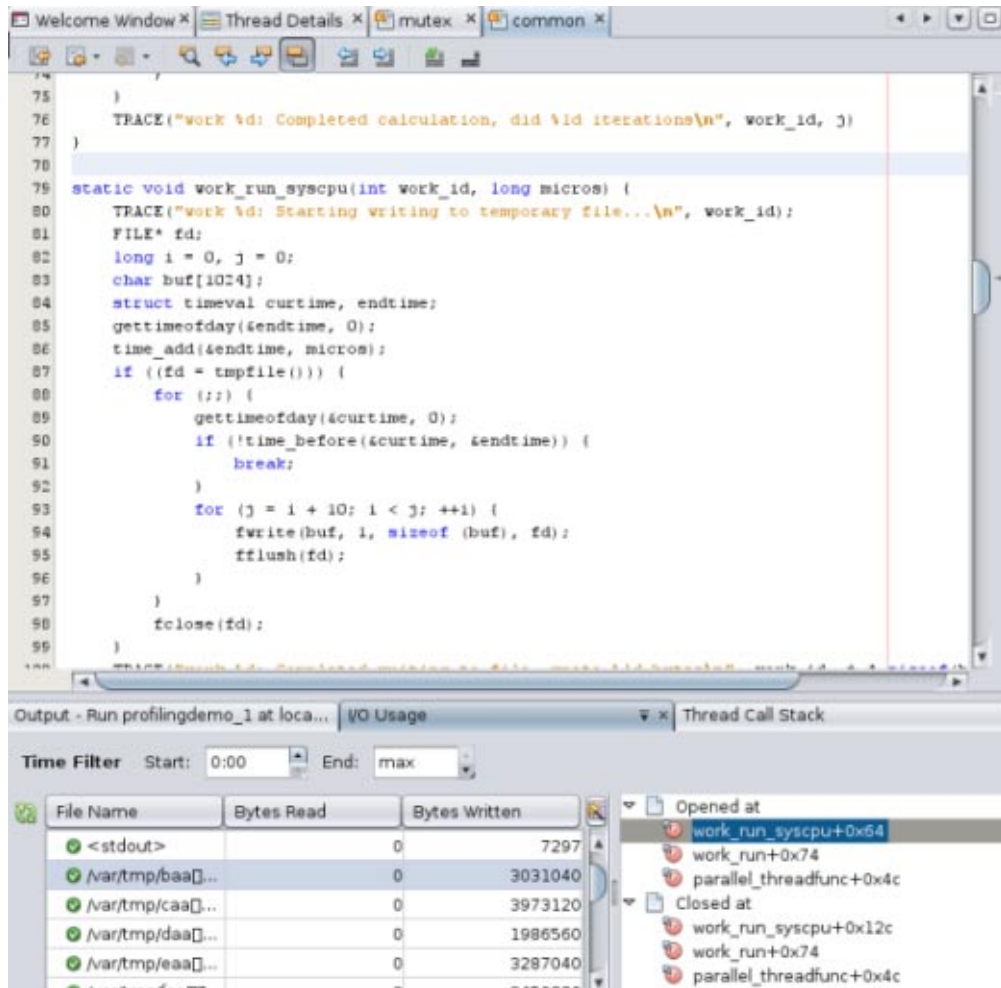
2. Notice the following:

   - The orange line shows the number of bytes written in the last second. The Profiling Demo program itself reports in the Output window that it writes a total of 79984640 bytes or about 76.3M during this run of the SEQUENTIAL DEMO. If you added up all of the data points shown on the orange line, the total would be close to 76.3M.

   - The System time shown in the CPU Usage tool is high during this phase because the program makes use of system calls to generate data and write it to the disk.

   - The Memory Usage tool shows a steady 8K bytes of memory heap allocated.

3. Click the I/O Details button. The I/O Usage details tab opens, displaying standard input, standard output, and temporary files that the program reads from and writes to. The files with check marks have been closed. The files that are marked with a yellow icon are still open for read and write actions.



4. Notice that this program requires input from pressing the Enter key only occasionally, so the Bytes Read column is not very useful. You can close the Bytes Read column by clicking the Change Visible Columns button to the right of the column headers. In the Change Visible Columns dialog box, click the checkbox to deselect Bytes Read and click OK.

5. Suppose that you want more details about how this program uses all of these temporary files. Click the /var/tmp/baa[]... file in the I/O Usage details tab to see the functions that opened and closed the file. The functions are listed in the panel to the right of the file list.

6. Double-click the work_run_syscpu function in the function list to open the source file for the function.

```
75        }
76        TRACE("work %d: Completed calculation, did %ld iterations\n", work_id, j)
77   }
78
79   static void work_run_syscpu(int work_id, long micros) {
80        TRACE("work %d: Starting writing to temporary file...\n", work_id);
81        FILE* fd;
82        long i = 0, j = 0;
83        char buf[1024];
84        struct timeval curtime, endtime;
85        gettimeofday(&endtime, 0);
86        time_add(&endtime, micros);
87        if ((fd = tmpfile())) {
88            for (;;) {
89                gettimeofday(&curtime, 0);
90                if (!time_before(&curtime, &endtime)) {
91                    break;
92                }
93                for (j = i + 10; i < j; ++i) {
94                    fwrite(buf, 1, sizeof (buf), fd);
95                    fflush(fd);
96                }
97            }
98            fclose(fd);
99        }
```

7. In the I/O Usage details tab, as in the CPU Time Per Function tab and Thread Synchronization Details tab, you can specify a time interval to look at. In the I/O Usage graph in the Run Monitor window, the write activity starts very near the beginning of the run, at the point where the SEQUENTIAL DEMO portion of the program begins writing to a temporary file.

8. Isolate the SEQUENTIAL DEMO portion of the run by typing the time that it ended (0:17 in the following image) in the End field. The data is filtered in the Run Monitor window and the I/O Usage details tab so that you can focus on the input and output of the SEQUENTIAL DEMO phase.
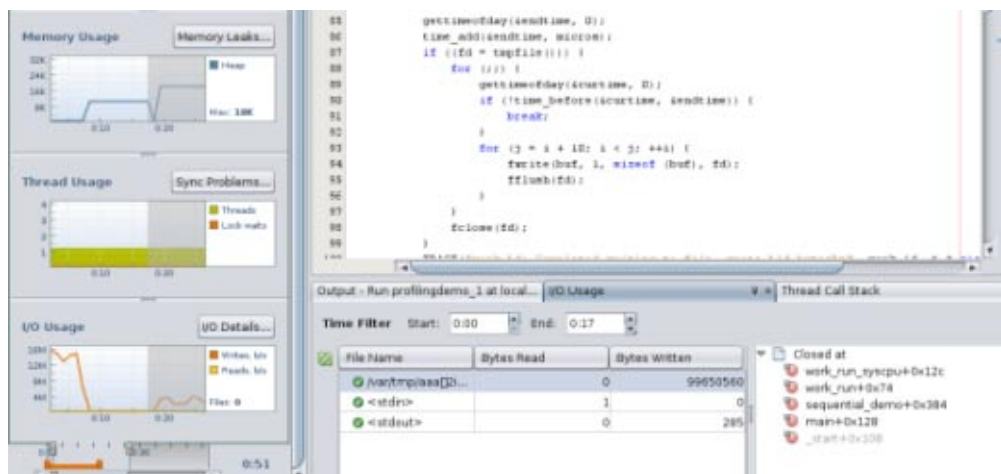
9. Notice that one temporary file is shown in the I/O Usage details tab during the sequential demo. A single thread opens, writes to the file, and then closes it. Click the file to see the functions that access it, and double-click a function to see its source code.

10. In the Run Monitor window, move the Time slider to the right until you see write activity begin again after a pause. The pause in writing occurs during the second task in the sequential demo, which is a calculation task during which no disk writes take place. The renewed write activity shows that the program is entering the PARALLEL DEMO portion, where the user presses Enter to launch the tasks, and the writing to disk and calculation occur simultaneously in separate threads.

11. Isolate the parallel demo activity by typing the time it starts in the Start field and the time it ends in the End field. For the run shown in this image, the start time is 0:18 and the end time is 0:29.
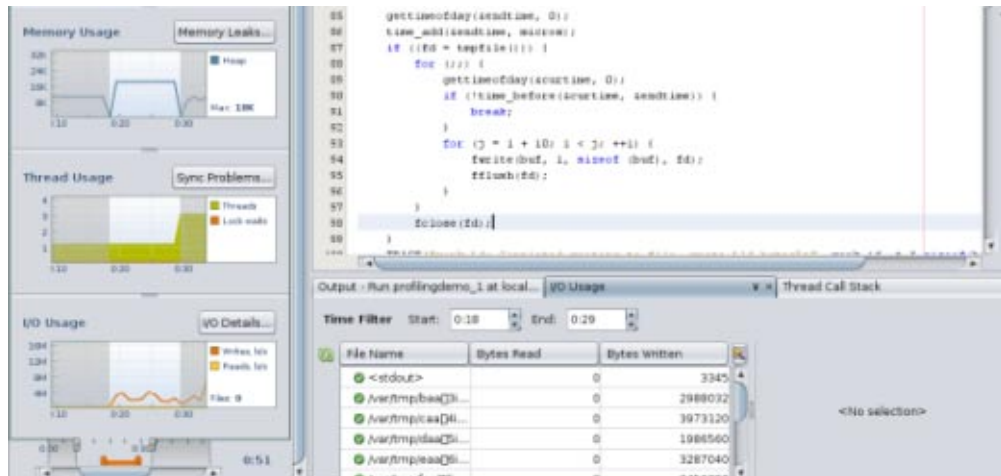


12. Notice that several temporary files are shown in the I/O Usage details tab during the PARALLEL DEMO portion of the run. Only one thread is writing to the files because each second it switches to a new file. The calculating task does not write to disk.

13. Click one of the files and see that the parallel_threadfunc function is opening and closing the files.
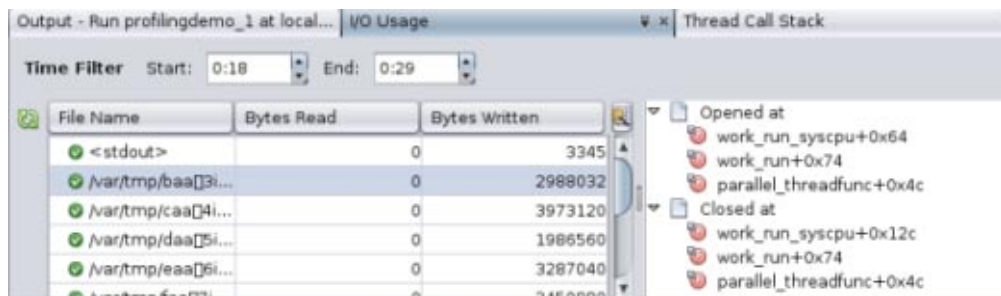


14. Click and drag the right handle of the Details slider to the end of the run. In the following image, you can see that the I/O activity changes again when the program enters the PTHREAD MUTEX DEMO portion and the user presses Enter. Filter the data for this portion of the run by changing the start time to the end time of the parallel demo, in this case 0:24.
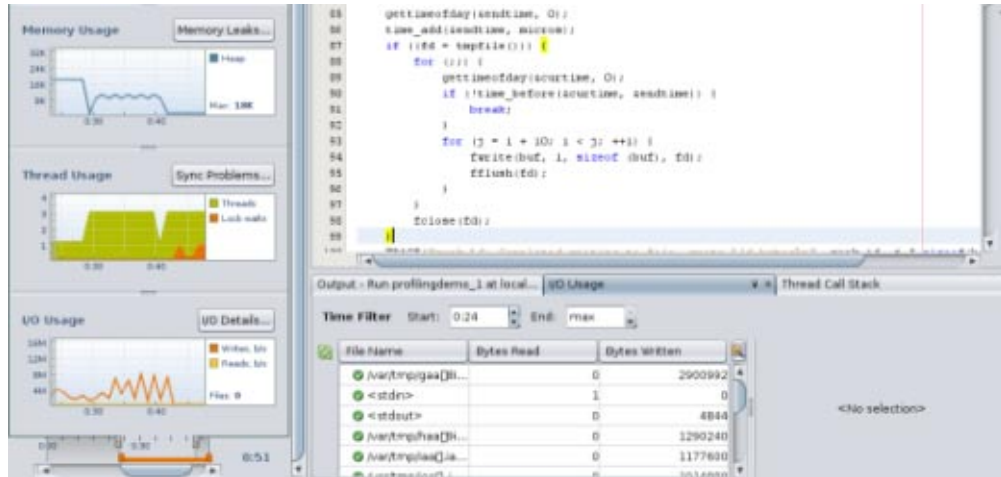
15. The I/O Usage details tab shows that multiple files are open during the PTHREAD MUTEX DEMO portion of the run. One thread is writing to a file, and every second it switches the file it writes to, as in the PARALLEL DEMO portion. However, because mutex locks are used, sometime the writing thread is blocked by the calculating thread and cannot continuously write to files.

This concludes the demonstration of the Run Monitor window's tools for Execute Targets. These tools also apply to the Process Target.

# Profiling a Process Tree

When you run a Process Tree Target, you attach DLight to a running process. DLight profiles the specified process and all child processes that it spawns or forks. Profiling a running process and its child processes can help you detect issues that occur between the processes as well as among the threads of a process.

If you want to capture the activity from when a process launches, you can start profiling the shell you use to execute the program, and the activity of the shell and the program's main process and all its child processes will be recorded. You can also target the program's already running process if you are not interested in the startup activity.

This section shows generally how to use the Process Tree target and does not require use of a sample application.

1. In a terminal window, type the following command to determine the PID of the shell.

   ```
   % echo $$
   ```

2. In DLight, click the down-arrow next to the New Target button and select Process Tree Target.

   The Create New Target dialog box opens with the Process Tree Target tab selected. The tab shows a list of processes running on the system.

3. In the Process Tree Target tab's Filter field, type the PID that was returned by the echo command to find the PID in the list of processes.

   The list of processes should refresh automatically to display only the processes containing the string you typed in the Filter field.

4. Select the process running with the PID and click Save.

   The target is listed in the DLight Targets tab with the label Process Tree: [pid] [localhost]. The target is saved permanently to run on this PID, no matter what program you run from the shell running under this PID.

5. Select the Process Tree target and click the Run button: ▷ .

The Process Tree Profiling tab opens in DLight, and displays graphs labelled Thread Microstates of Profiled Processes and CPU Usage of Profiled Processes. At first, only the shell is profiled and not much activity is occurring.

6.  In the terminal window where the targeted shell is running, start the program that you want to profile.

    The following figure shows the Process Tree Profiling tab with data being generated on a program called `locker` which starts many child processes.
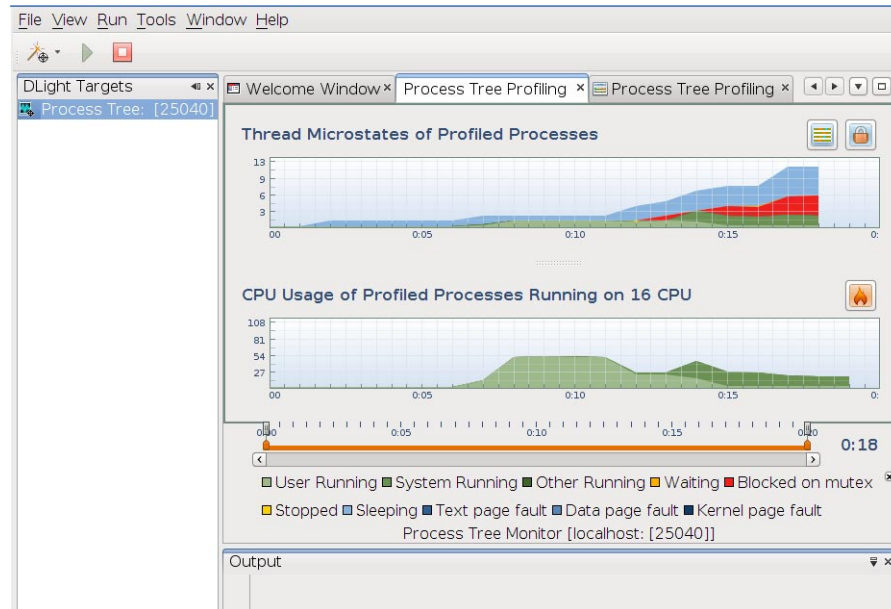


These graphs together can be used to determine how your application's multiple processes and multiple threads are working together. You can see points where threads are blocked and the effect on CPU usage, and narrow the problems down to the lines of code where they occur.

7.  Examine the Thread Microstates of Profiled Processes graph at the top of the Process Tree Profiling tab.

    In the Thread Microstates of Profiled Processes graph, the vertical axis shows the number of threads running and the horizontal axis shows the wall clock time elapsed while the processes ran. However, the graph shows an aggregation of the microstates of all the threads of all the processes that DLight profiled with the Process Tree Target. You can use the graph for a broad view of the behavior of the processes.

8.  Examine the CPU Usage of Profiled Processes graph.

    The CPU Usage of Profiled Processes graph's vertical axis shows the percent of CPU time, and the horizontal axis shows the wall clock time elapsed while the processes ran. However, the data shown is an aggregation of the CPU usage of all the threads in all the processes profiled across all the CPUs on the system. The number of CPUs is noted in the title of the graph and also in the Host Info window on the left side of the DLight window.

    Along with the Thread Microstates graph, you can use the CPU Usage graph to determine where there are locking issues.

9.  Click the Thread Map button  to open a new Process Tree Profiling tab that shows detailed information about the microstates of processes and threads.

    The Process Tree Microstate Details window shows the microstate transitions in the form of a timeline for each thread of the target process and its child processes during the period that DLight is monitoring the target.

    When the window opens initially, each profiled process is shown. The target process is the top line and it shows the aggregation of microstates for all the threads executing in that process. The child process timelines also show the microstates for the threads they execute.
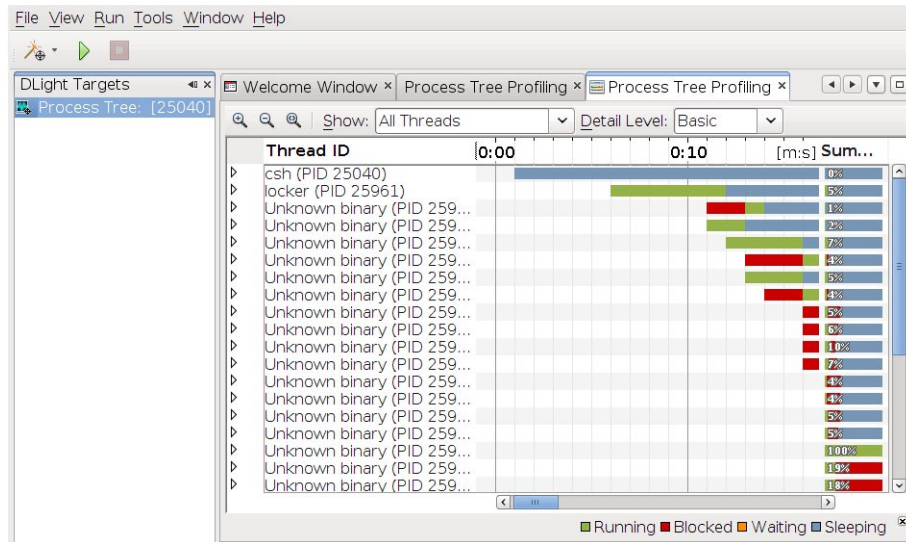
In the figure above, the top line shows the `csh` shell process that was specified in the DLight target, and it is sleeping while the program runs. The second line is the `locker` program's main process, and the rest of the "Unknown binary" processes were started by the `locker` program as forks of the original process. This is a known issue that DLight sometimes cannot determine the process name.

10. Click the handle of a process timeline to show timelines for each thread created by the process.
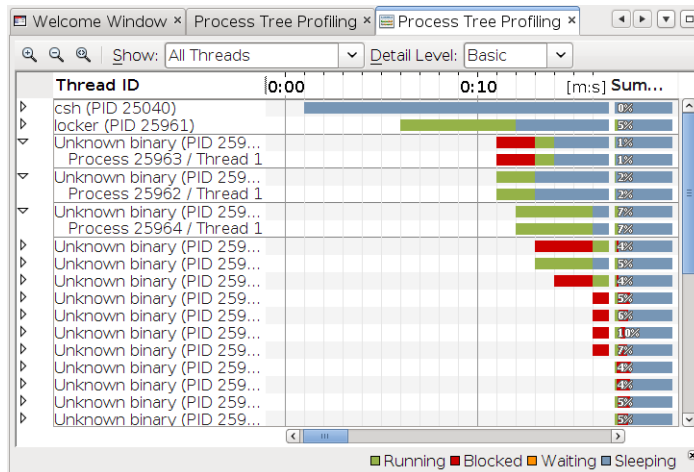


If the processes are single-threaded as shown in this document, the process timelines and thread timelines look the same. However, the thread timelines show more information when you click them as described later. If the processes were multithreaded, expanding the process would show multiple threads.

11. Place your mouse cursor over the Summary area to the right.

The Summary area shows the percent of time spent in microstates for the entire run of the thread or process. When you place the mouse cursor over the Summary, a legend is displayed to indicate the meaning of the colors and the percentage of time for the process or thread. Similar pop-ups are displayed when you place the mouse cursor over the processes and threads timelines themselves.
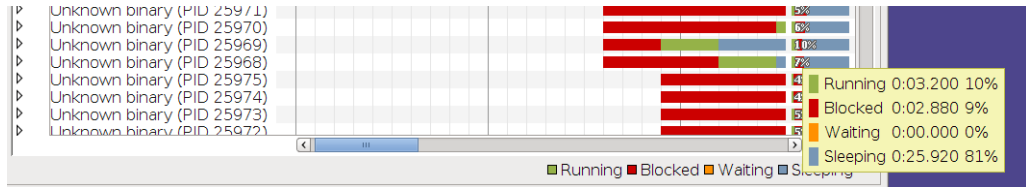
In the figure above, the mouse cursor was placed over the Summary area of a thread.

12. Click the handle of a process timeline to show its thread timelines, then click some point in a thread timeline and the Thread Call Stack window opens to show the call stack that was executing at that moment in time.
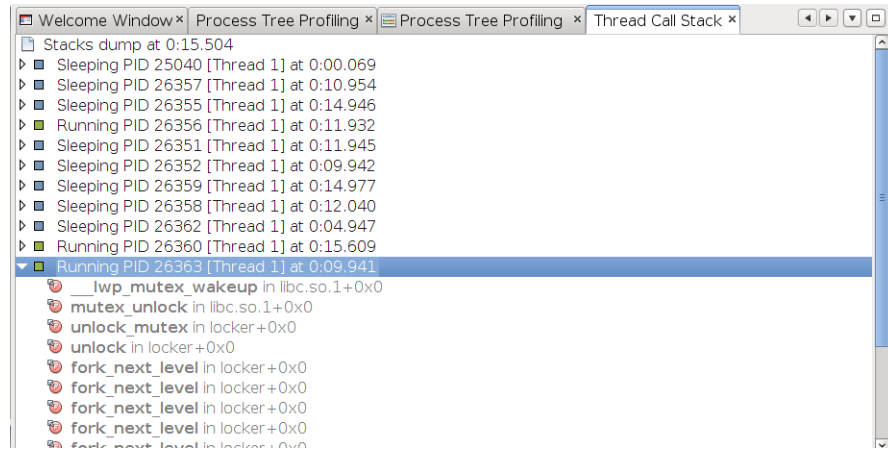


In the call stack, if the program's source code is available, the highlighted functions indicate functions that occur in your source. Double-click the highlighted functions to see the source in a new window in DLight. If the executable was built with debugging information (using the -g compiler option) then DLight can also provide source code navigation.

For more information about viewing microstate timelines, see the DLight help by choosing Help ⇒ Help Contents from the DLight menu bar.

13. Click the main Process Tree Profiling tab to return to the Thread Microstates of Profiled Processes and CPU Usage of Profiled Processes graphs.

14. Click the Lock Stat button  to open a new tab that shows detailed information about the microstates of processes and threads.

A new Process Tree Profiling tab opens and shows locking statistics for the process and its children, including the amount of blocked time for threads that are blocked, threads that hold locks, mutexes, and lines of code where threads were blocked.
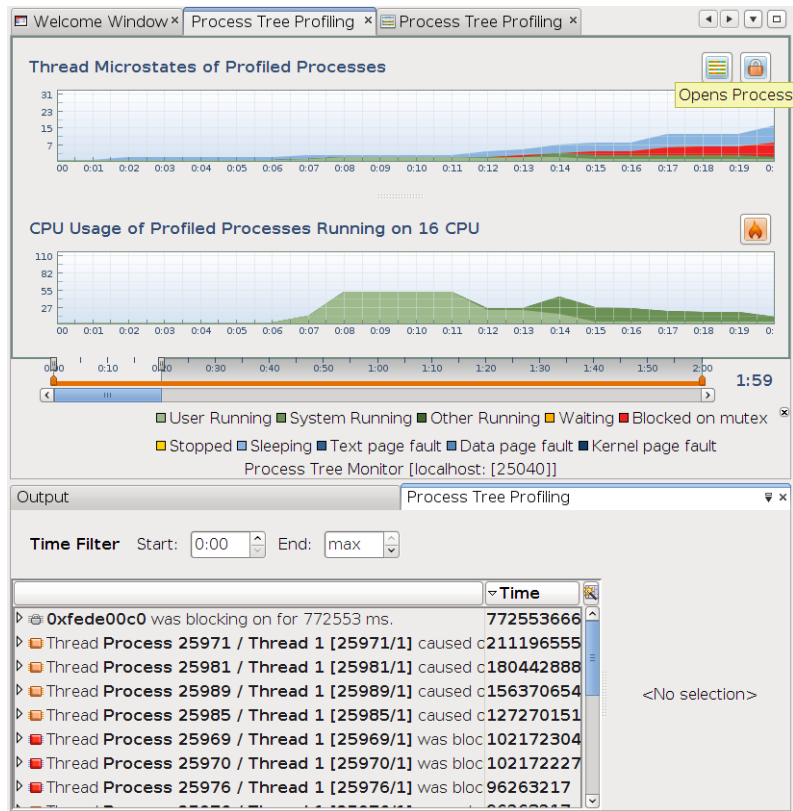
15. Click the handles on the left to display the relationships between the threads, functions, and lines of code that are blocking or being blocked.

16. Click an item in the list to open a thread call stack in a side window.
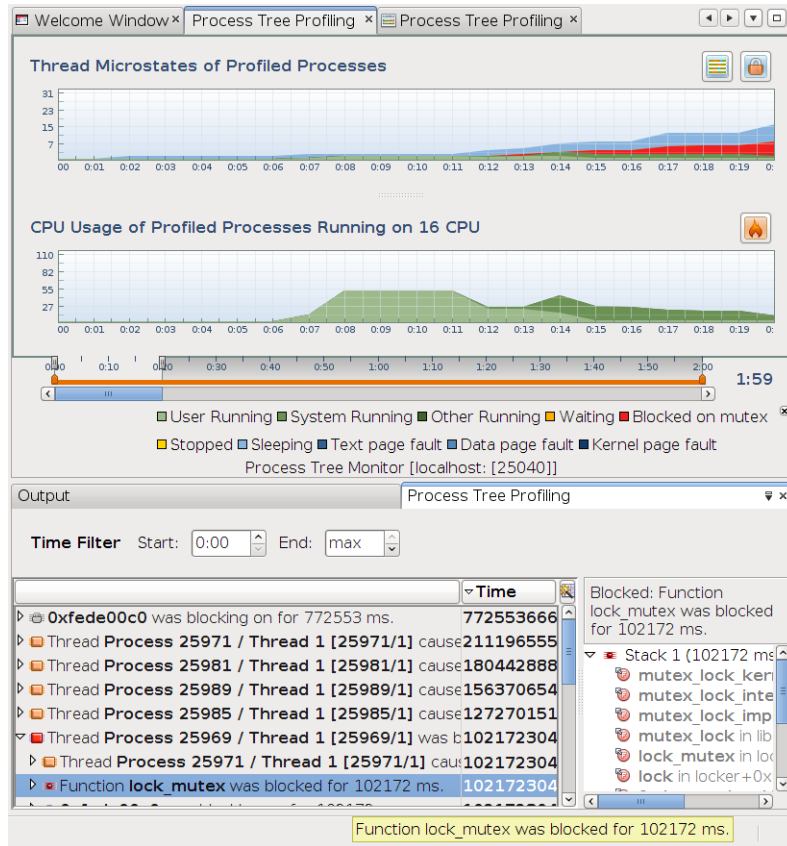
If multiple call stacks were affected by a block, each stack is listed in this window. The blocked time shown for an item in the list is an aggregate of all the calls to the item. In the call stack window, you can see how much of the block time was used by each call stack. The most time-consuming call stack is listed first. You might see multiple call stacks even when one line of code is responsible for all the block time if the line is executed in different paths through the code.

17. Notice that some of the functions in the call stack are shown in bold. Click the bold function name to open the source file at the listed line number in DLight.

    Because DLight has access to the source code, the functions are listed in bold and the source file name and line number is shown.

18. Click the main Process Tree Profiling tab to return to the graphs Thread Microstates of Profiled Processes and CPU Usage of Profiled Processes.

19. Click the Hot Spots button  in the CPU Usage of Profiled Processes panel to open a new tab to help you identify the CPU-intensive areas (hot spots) in your program's process tree.

The tab displays the functions in your program with their exclusive CPU Time metric. This is the time used only by the specific function, not including the CPU time used by any functions that it calls.

Notice that some of the functions in the list are shown in bold.

20. Double-click the bold function names to open the source file at the listed line numbers.

If the source file is available to DLight, a new window opens to show the source.

# For More Information

For more information about using the DLight graphical tools to identify performance problems in your applications, choose Help ⟹ Help Contents from the DLight menu bar.

The help includes instructions for topics not covered in this document, including:

- Profiling on a remote host so you can run DLight locally and profile an application that is running on a remote system.
- Specifying a tool configuration to create a subset of the DLight tools so you see only the ones you are interested in.
- Detailed usage information for each tool to show the tasks you can perform with each tool.

**Oracle Corporation 500 Oracle Parkway, Redwood City, CA 94065 U.S.A.**

**ORACLE**®