

A large, abstract, grayscale graphic on the left side of the page, consisting of overlapping curved shapes and a thin white line that curves across the page.

# **Oracle Solaris Studio 12.2 DLight Tutorial**

September 2010

- “Introduction” on page 2
- “Building the Sample Application” on page 2
- “Starting DLight” on page 3
- “Running the Sample Application” on page 3
- “Using the Indicator Controls” on page 3
- “Exploring Thread Microstates” on page 5
- “Exploring CPU Usage” on page 13
- “Exploring Memory Usage” on page 16
- “Exploring Thread Usage” on page 18
- “Exploring I/O Usage” on page 21

## Introduction

DLight is an interactive graphical observability tool for C/C++ developers based on Oracle Solaris Dynamic Tracing (DTrace) technology. You can use DLight on Solaris platforms to analyze data from multiple DTrace scripts in a synchronized fashion to trace a runtime problem in an application to the root cause.

You must have root privileges on the system where you run DLight. If you are not logged in as root when you start DLight, you are prompted for your root password the first time you run your program.

DLight includes five profiling tools for C, C++, and Fortran programs:

- Thread Microstates
- CPU Usage
- Memory Usage
- Thread Usage
- I/O Usage

DLight also includes three profiling tools for processes in the AMP (Apache, MySQL, PHP) stack:

- Apache Monitor
- MySQL Monitor
- PHP Monitor

When you run an executable target or attach to a target process, each tool displays usage information in a graph in the Run Monitor window. Each graph includes a button to click for more information. Indicator controls at the bottom of the window let you control your view of the graphs.

## Building the Sample Application

The ProfilingDemo application is available in the `examples/dlight/ProfilingDemo` directory in your installed Oracle Solaris Studio 12.2.

1. Copy the contents of the ProfilingDemo directory to your own private working area:

```
cp -r installation_directory/examples/dlight/ProfilingDemo ~/ProfilingDemo
```

2. Build your own copy of the program:

```
cd ~/ProfilingDemo  
make
```


The program 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

If you do not already have DLight running, start it by typing:

```
installation_directory/bin/dLight
```

## Running the Sample Application

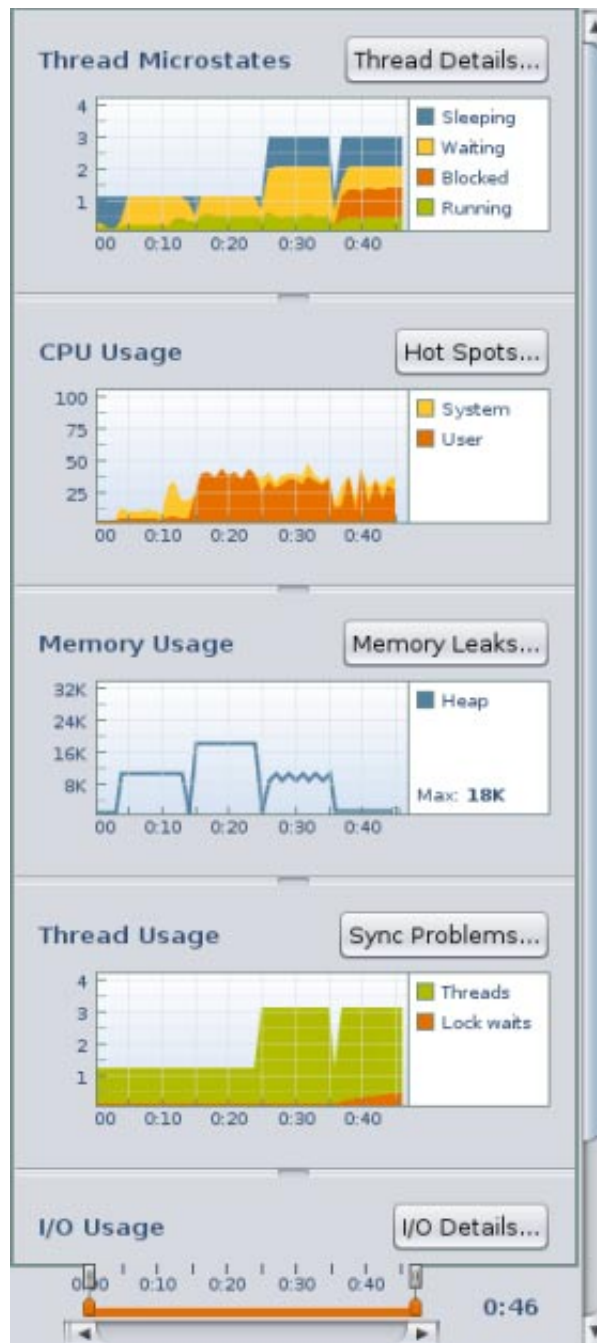
1. In DLight, click the New DLight Target button . In the target dialog box:
  - a. Click the Executable Target tab.
  - b. Type the pathname to your profilingdemo executable in the Run field, or click Browse to use the Open dialog box to browse to your profilingdemo executable file.
  - c. The sample application does not take any arguments, so leave the Arguments field blank.
  - d. You would only use the Trace field if you wanted to trace a child process of the executable, so leave that field blank also.
  - e. Click Run.
  - f. If you were not logged in as root when you started DLight, you are prompted for your root password.
2. The ProfilingDemo application starts executing and the Run Monitor window starts displaying dynamic graphs. 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.
3. Press Enter each time the program requests it until execution is finished.

## Using the Indicator Controls

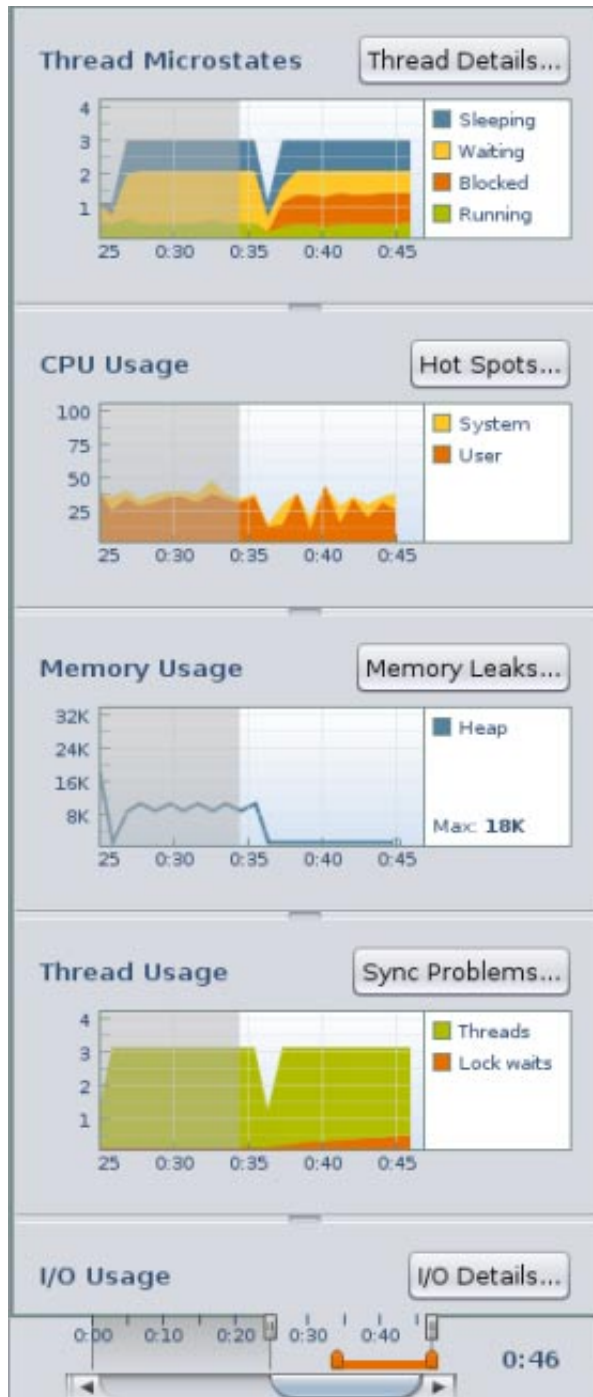
1. At the bottom of the Run Monitor window, you can see sliders for controlling your view of the graphs: View slider, Details slider, and Time slider. Place your mouse cursor over the end points of the sliders for information about the sliders.



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 what 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. 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. Now let's focus our attention closer. 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.
10. Drag the start point of the View slider back to the beginning of the run so you can see all the data.

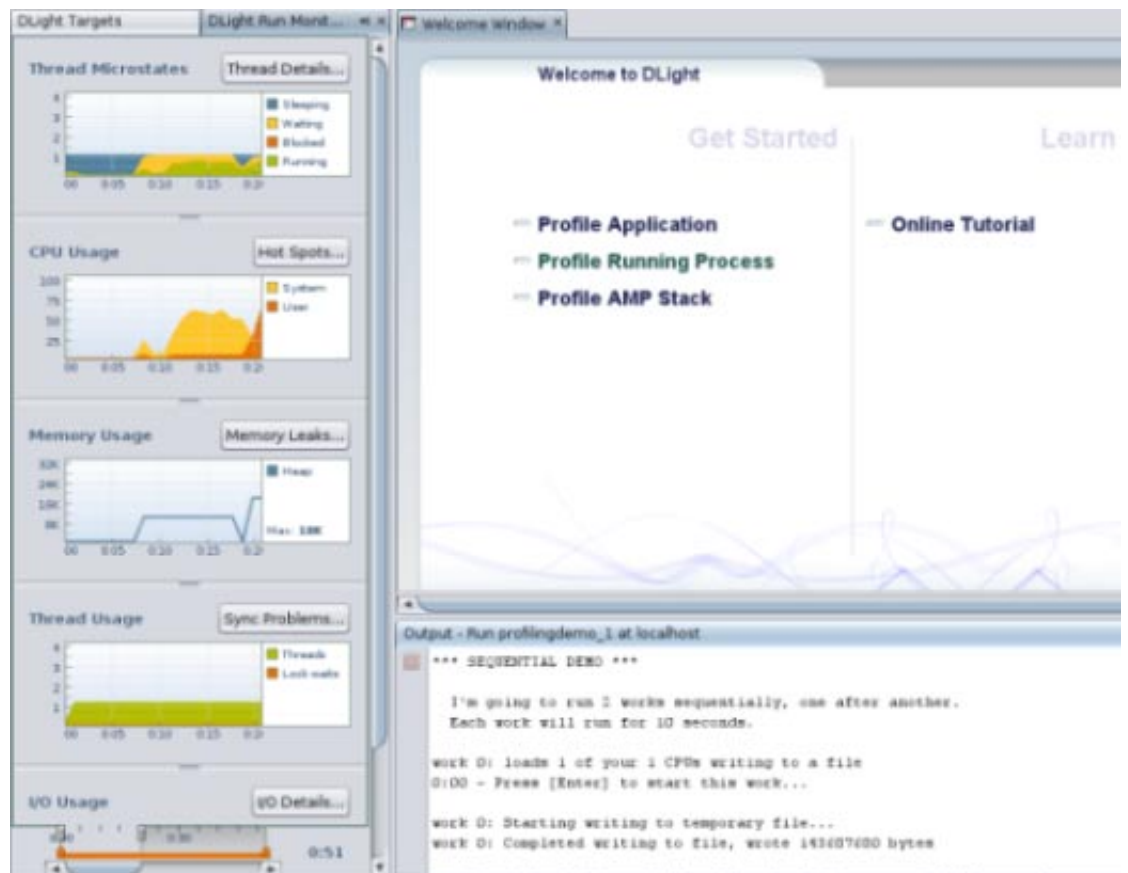
## Exploring Thread Microstates

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:

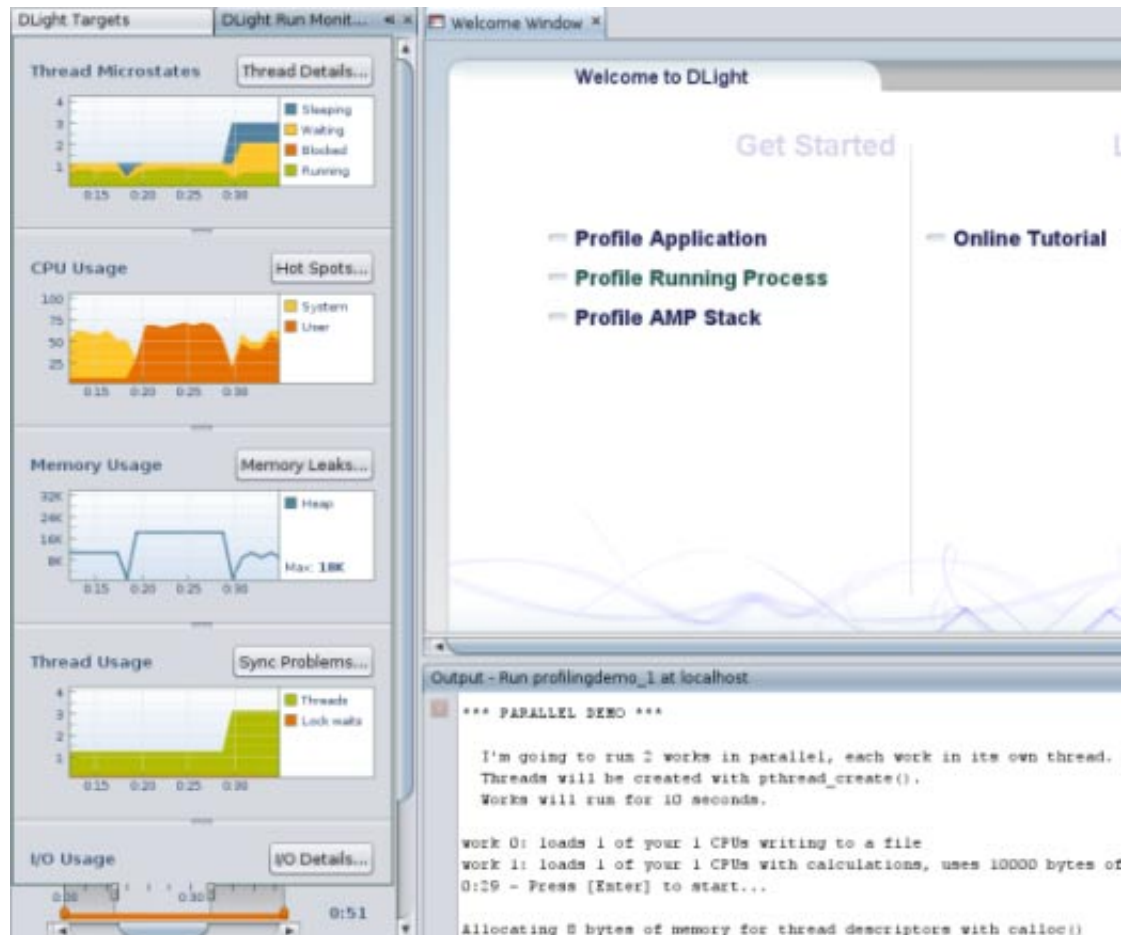
User Running	The percentage of time the process has spent in user mode
System Running	The percentage of time the process has spent in system mode
Other running	The percentage of time the process has spent in processing system traps and such
Text page fault	The percentage of time the process has spent 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.



- Click and drag the Time slider to the right until the number of threads shown in the Thread Microstates tool jumps to three.
- 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.

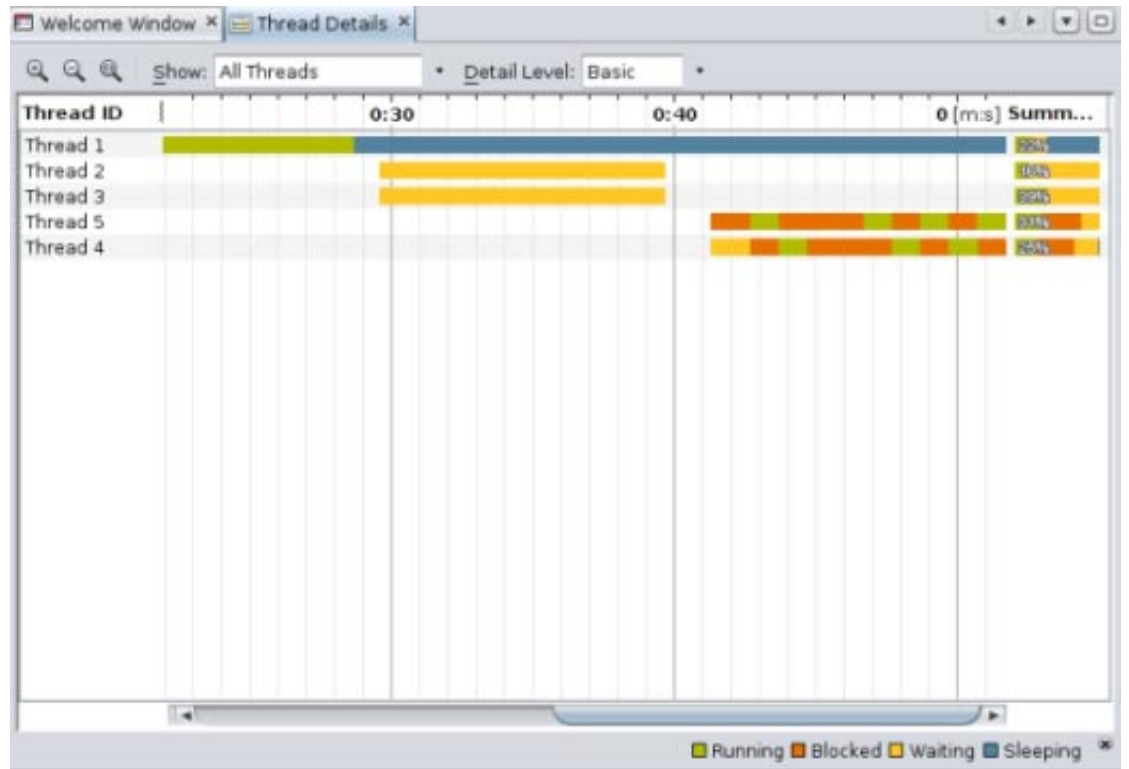


- Drag the Time slider all the way to the end of the run time. Notice that the Blocked microstate shown in orange appears at the point where the program enters the PTHREAD MUTEX DEMO portion, in which each thread uses mutual exclusion locks to prevent other threads from interfering at certain points. 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.



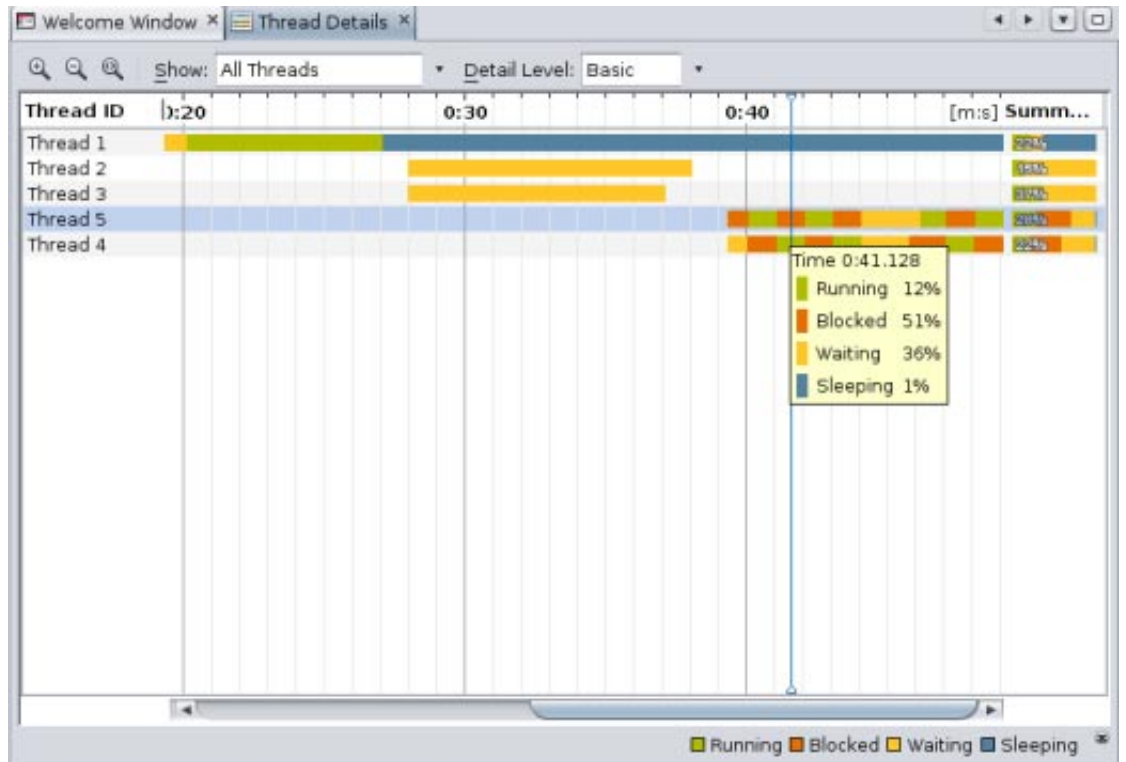
5. Click the Thread Details button to display details about the thread microstates. The Thread Details window opens to display a graphical timeline representation of all the threads run in the program along with detailed state information.




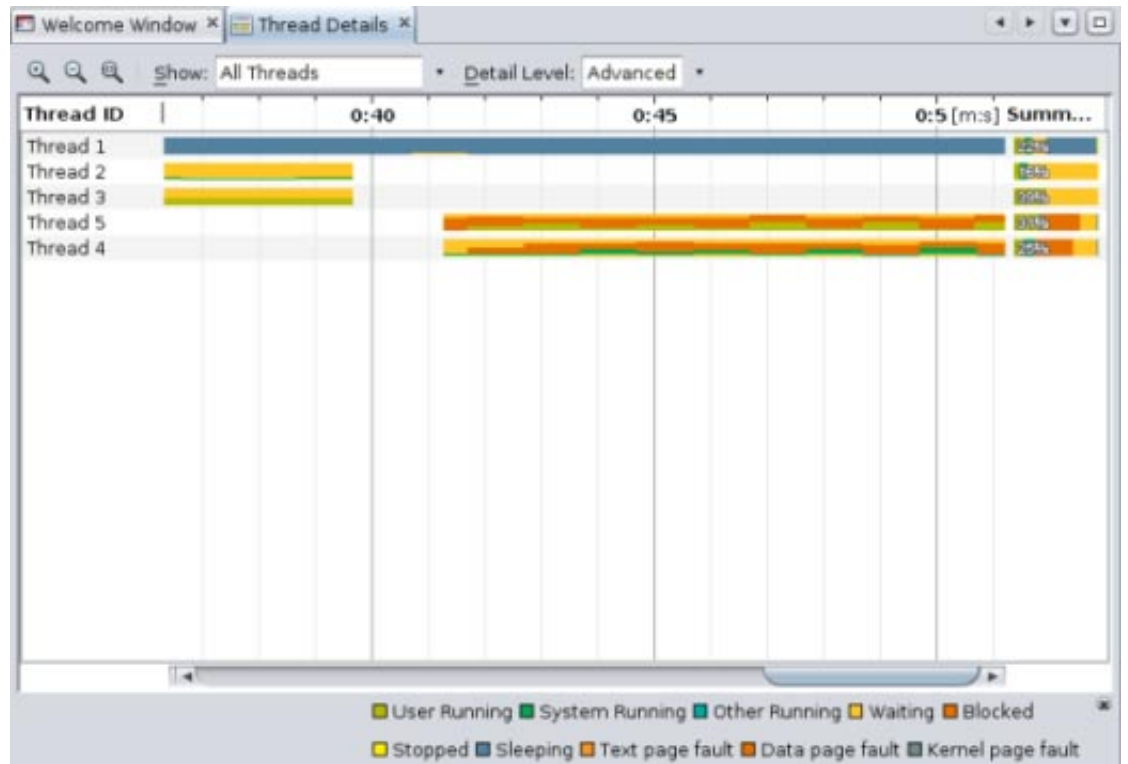





The Thread Details window shows the state transitions for each thread during the complete run time of the program.

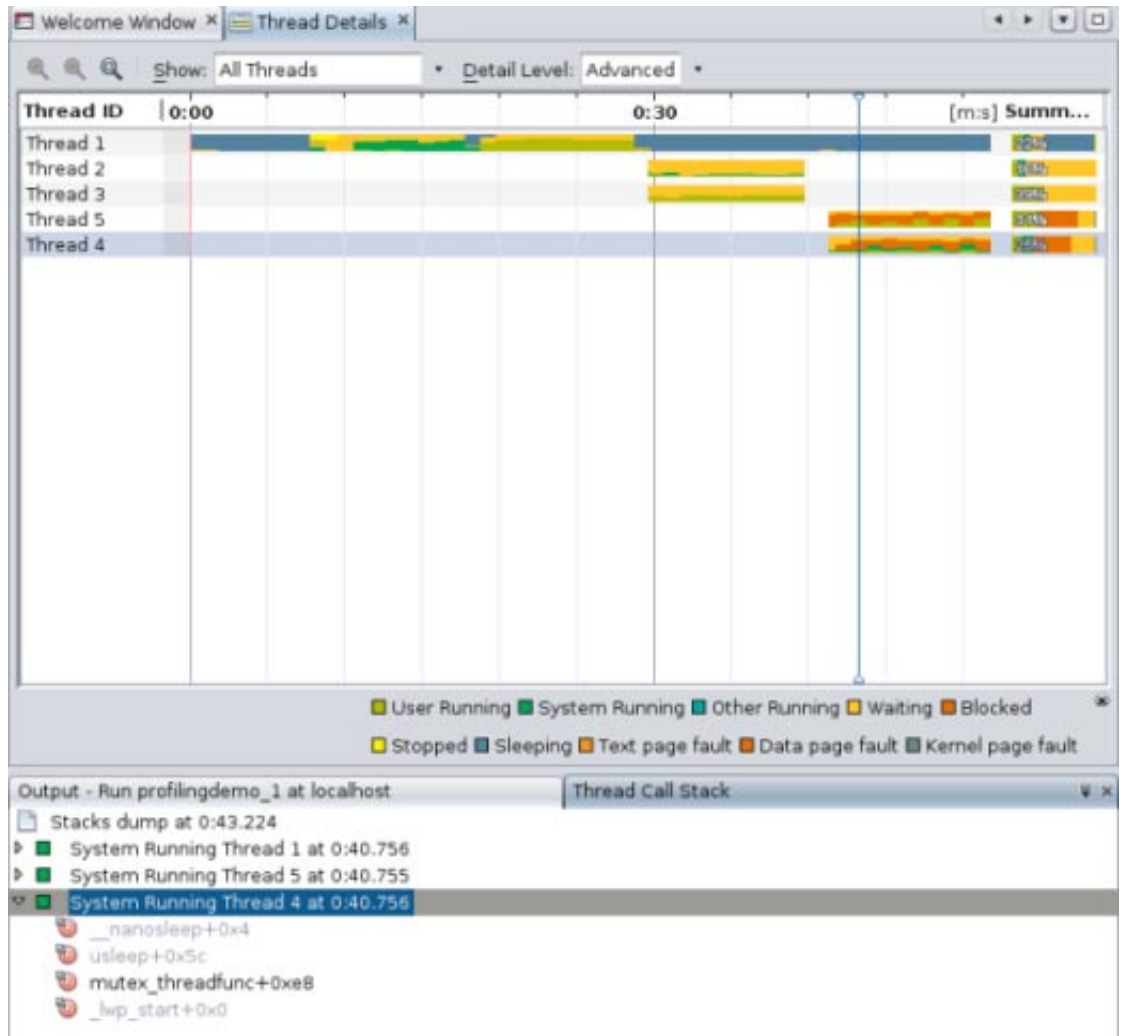
- Put your cursor on one of the colored areas of a thread. A popup window displays details about what is going on with that thread at that particular moment. Details include the time when the data was collected and the percentage of time spent in each thread state at that moment. When you put your cursor 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.



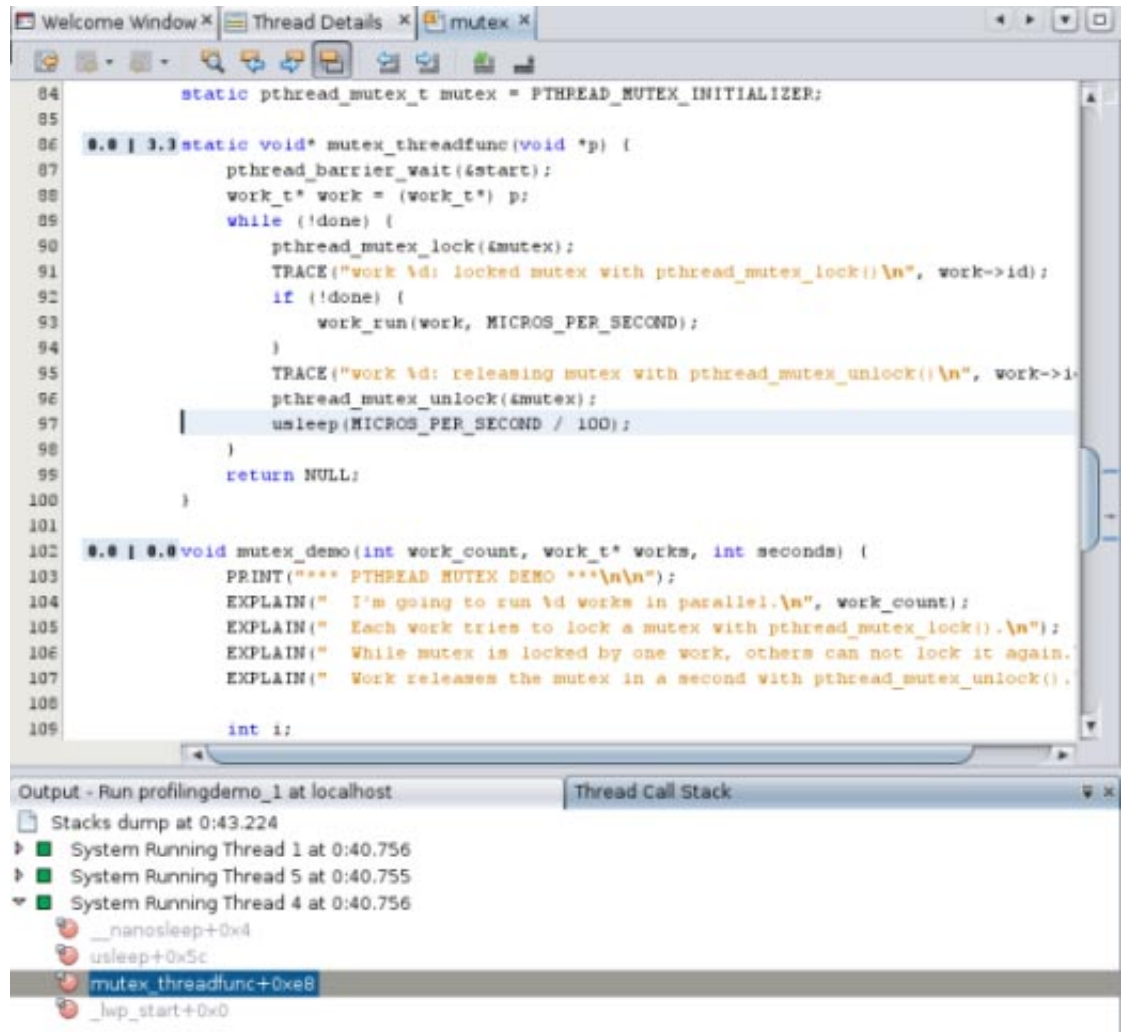
7. Try using the window's controls to change what is shown:
  - By default, the window shows all threads. Click the downward arrow to the right of the Show drop-down list. You can select Live Threads only to show only threads that have not terminated, or Finished Threads only to show only those threads 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.
  
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.)



- 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.

- 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.

Function Name	CPU Time (Exclusive)	CPU Time (Inclusive)
<b>work_run_usrcpu</b> at common.c:59	12.861	13.004
_write	3.651	3.651
_read	0.530	0.530
mutex_lock_impl	0.352	0.352
mutex_unlock_queue	0.187	0.187
nanosleep	0.015	0.015

- Click the header of the Function Name column to sort the functions alphabetically.
- Click the header of the CPU Time (Exclusive) column to sort the functions by the order of time used by the individual functions.
- 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.
- 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.
- 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. Some numbers are displayed in the left margin for this line.

```

46     time->tv_usec = micros % MICROS_PER_SECOND;
47     }
48     }
49     static int time_before(struct timeval* a, struct timeval* b) {
50         return (a->tv_sec < b->tv_sec) || (a->tv_sec == b->tv_sec && a->tv_usec <
51         )
52     }
53     static void work_run_idle(int work_id, long micros) {
54         TRACE("work %d: Sleeping for %ld microseconds with usleep()\n", work_id,
55         usleep(micros);
56         TRACE("work %d: Done sleeping\n", work_id);
57     }
58     }
59     12.9 | 13.0 static void work_run_usrcpu(int work_id, long micros) {
60         TRACE("work %d: Starting mathematical calculations...\n", work_id);
61         long i = 0, j = 0;
62         double pi = 0;
63         struct timeval curtime, endtime;
64         gettimeofday(&endtime, 0);
65         time_add(&endtime, micros);
66         for (;;) {
67             gettimeofday(&curtime, 0);
68             if (!time_before(&curtime, &endtime)) {
69                 break;
70             }
71             for (j = i + 1000; i < j; ++i) {

```

- 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.

The screenshot shows the Oracle Solaris Studio IDE with the `common.c` source file open. The `work_run_usrcpu` function is visible, and a tooltip is displayed over the line numbers in the left margin. The tooltip shows the following CPU time metrics:

```

12.9 | 13.0
CPU Time (Exclusive):12860558307 ns
CPU Time (Inclusive):13003914806 ns

```

The source code includes the following functions:

```

46     time->tv_usec = micros % MICROS_PER_SECOND;
47 }
48
49 static int time_before(struct timeval* a, struct timeval* b) {
50     return (a->tv_sec < b->tv_sec) || (a->tv_sec == b->tv_sec && a->tv_usec <
51 )
52 }
53
54 static void work_run_idle(int work_id, long micros) {
55     TRACE("work %d: Sleeping for %d microseconds with usleep()\n", work_id, r
56     usleep(micros);
57     TRACE("work %d: Done sleeping\n", work_id);
58 }
59
60 static void work_run_usrcpu(int work_id, long micros) {
61     TRACE("work %d: Starting mathematical calculations...\n", work_id);
62     struct timeval curtime, endtime;
63     gettimeofday(&endtime, 0);
64     time_add(&endtime, micros);
65     for (;;) {
66         gettimeofday(&curtime, 0);
67         if (!time_before(&curtime, &endtime)) {
68             break;
69         }

```

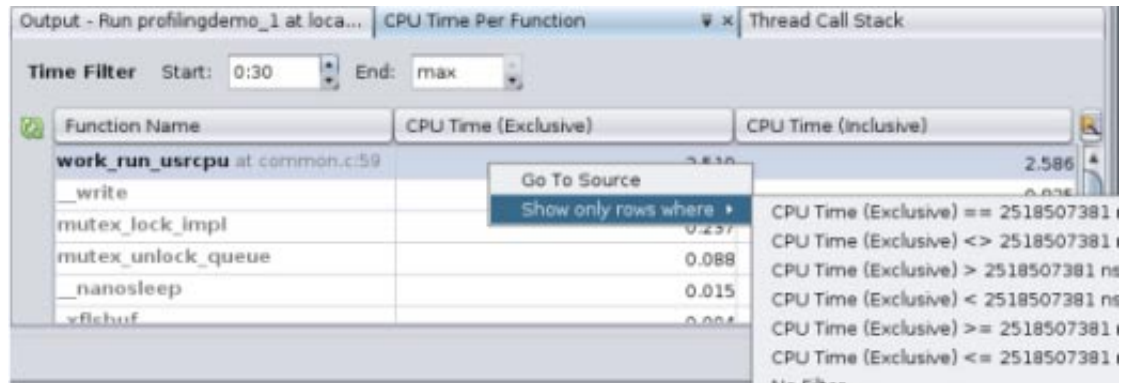
- Change the Time Filter start time in the CPU Time Per Function tab to 0:30 by either typing the time and pressing Enter, or using the arrows to scroll through the seconds. The graphs in the Run Monitor window change just as they did when you moved the handles on Details slider. If you drag the handles, 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.

```

46     time->tv_usec = micros % MICROS_PER_SECOND;
47 }
48
49 static int time_before(struct timeval* a, struct timeval* b) {
50     return (a->tv_sec < b->tv_sec) || (a->tv_sec == b->tv_sec && a->tv_usec <
51     b->tv_usec);
52 }
53
54 static void work_run_idle(int work_id, long micros) {
55     TRACE("work %d: Sleeping for %d microseconds with usleep()\n", work_id, r
56     usleep(micros);
57     TRACE("work %d: Done sleeping\n", work_id);
58 }
59
60 12.9 | 13.0 static void work_run_usrcpu(int work_id, long micros) {
61     TRACE("work %d: Starting mathematical calculations...\n", work_id);
62     CPU Time (Exclusive):12860558307 ns
63     CPU Time (Inclusive):13003914806 ns
64     struct timeval curtime, endtime;
65     gettimeofday(&endtime, 0);
66     time_add(&endtime, micros);
67     for (;;) {
68         gettimeofday(&curtime, 0);
69         if (!time_before(&curtime, &endtime)) {
70             break;
71         }
72     }
73 }

```

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.




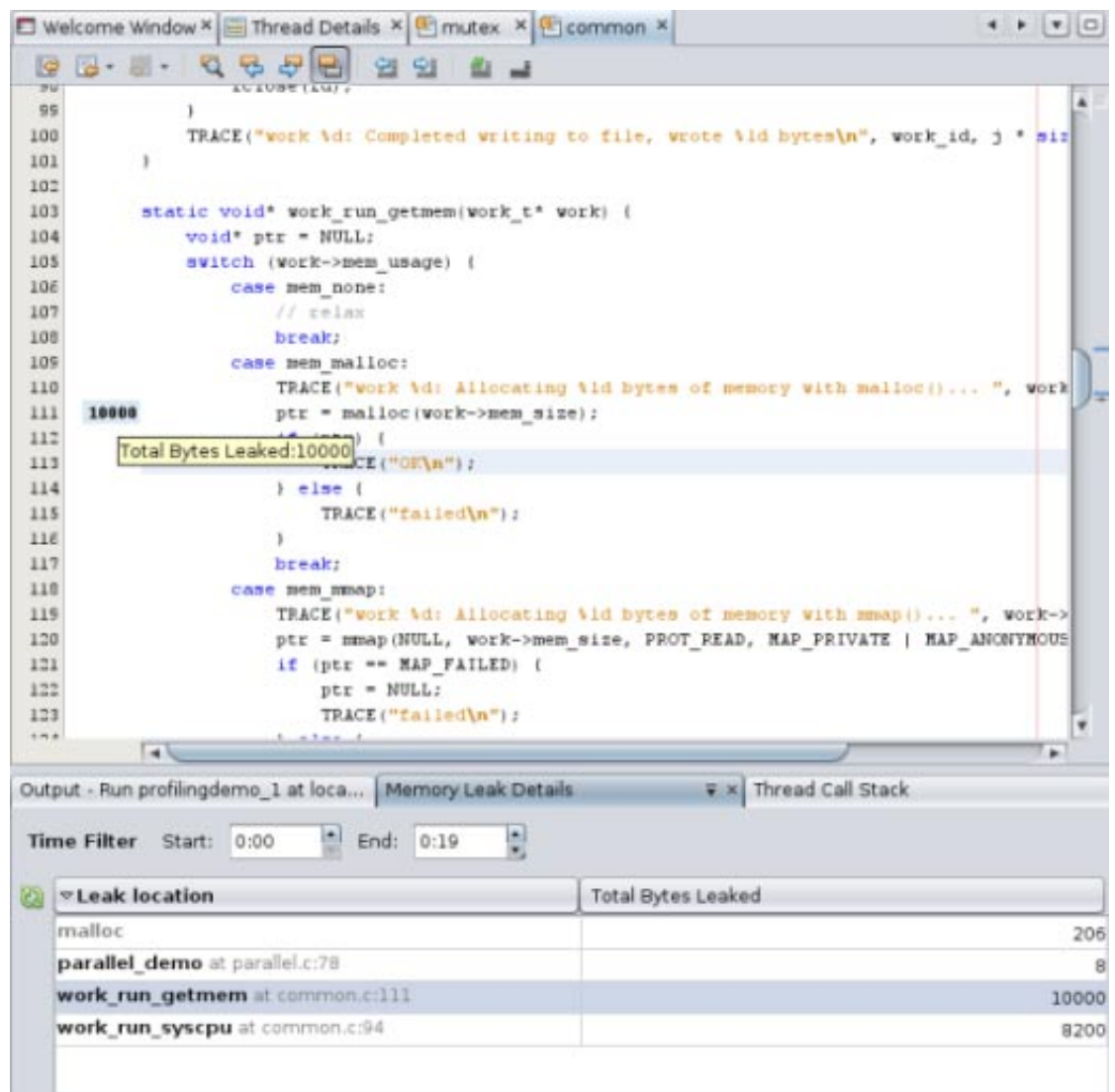
## Exploring Memory Usage

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 our 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.



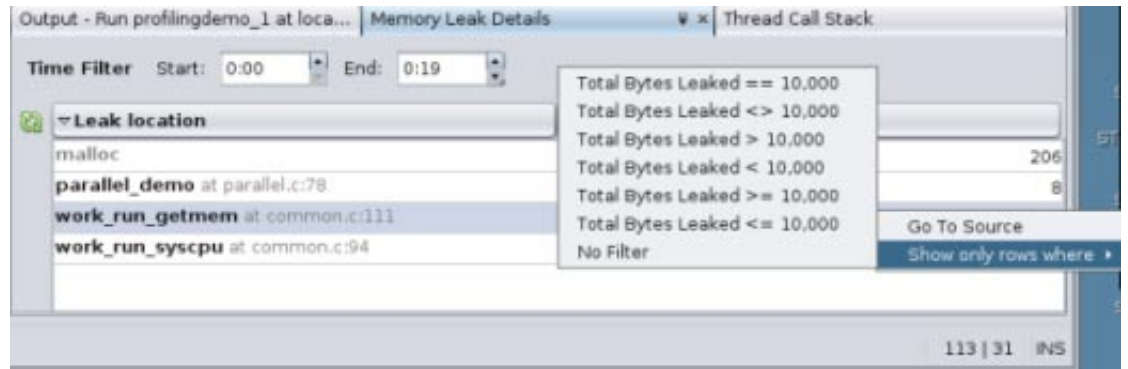
2. Click the Memory Leaks button to display the Memory Leak Details tab, which shows details about which functions exhibit memory leaks. Only functions that are producing memory leaks are listed in the tab. If your program is running when you click the button, the leak locations shown are those that 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, 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.



The screenshot shows the Oracle Solaris Studio IDE. The top window displays the source code for `common.c`. A memory leak is highlighted at line 111, where `ptr = malloc(work->mem_size);` is called. A tooltip shows "Total Bytes Leaked:10000". Below the code editor, the "Memory Leak Details" tab is active, showing a table of memory leaks.

Leak location	Total Bytes Leaked
malloc	206
parallel_demo at parallel.c:78	8
work_run_getmem at common.c:111	10000
work_run_syscpu at common.c:94	8200

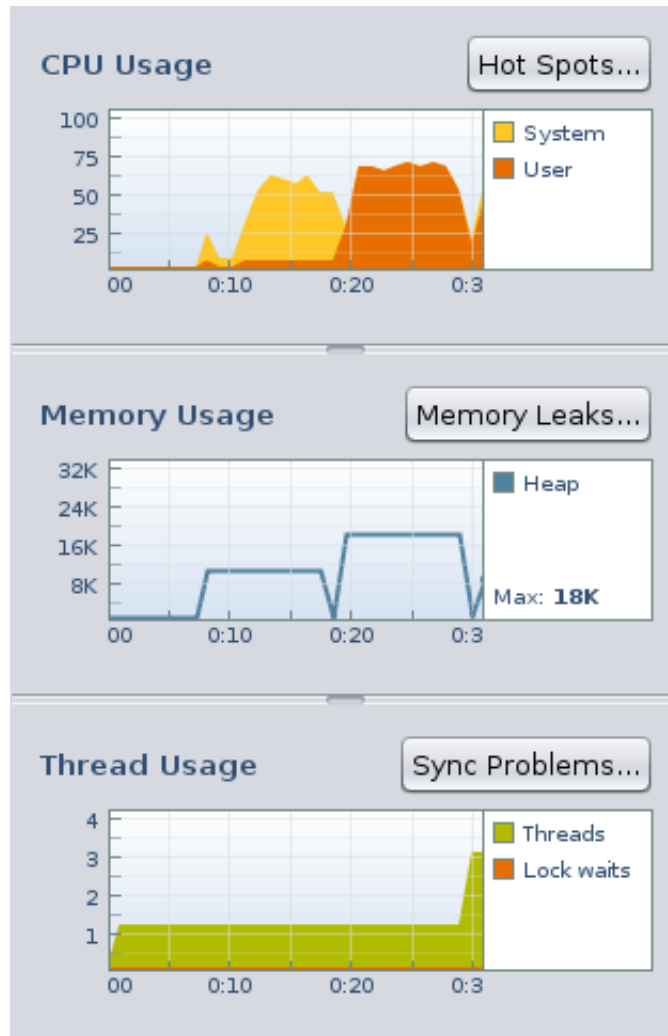
6. As with the CPU Time Per Function tab, you can right-click the metrics in the Memory Leak Details tab and select a criterion for filtering the data.



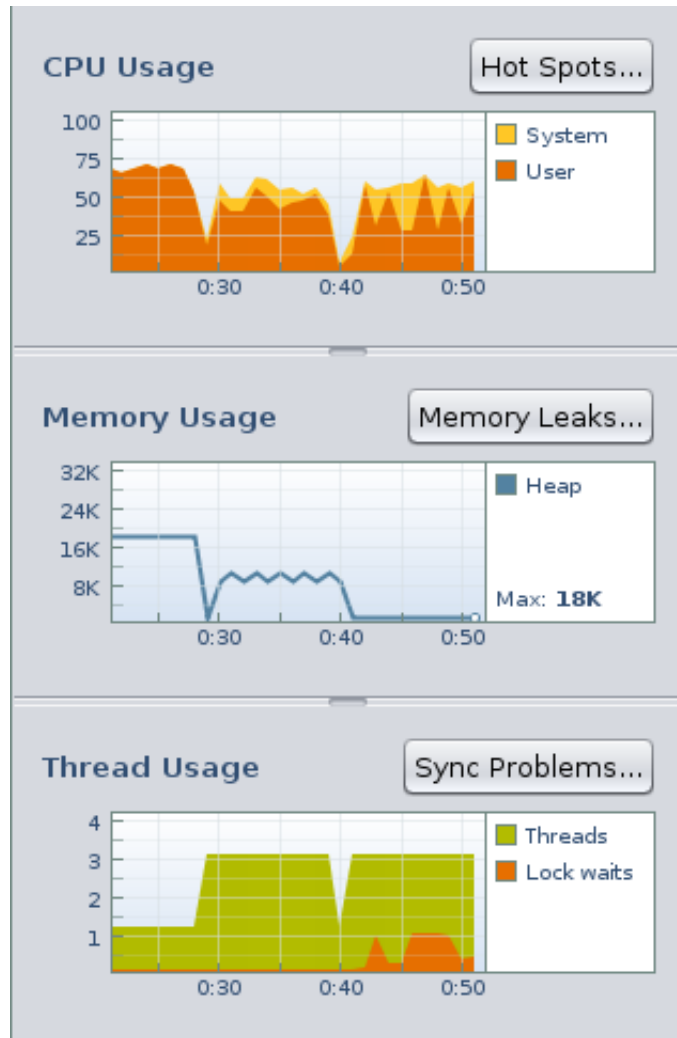
## Exploring Thread Usage


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.



- 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 these 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         work_t* work = (work_t*) p;
89         while (!done) {
90             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->id);
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;

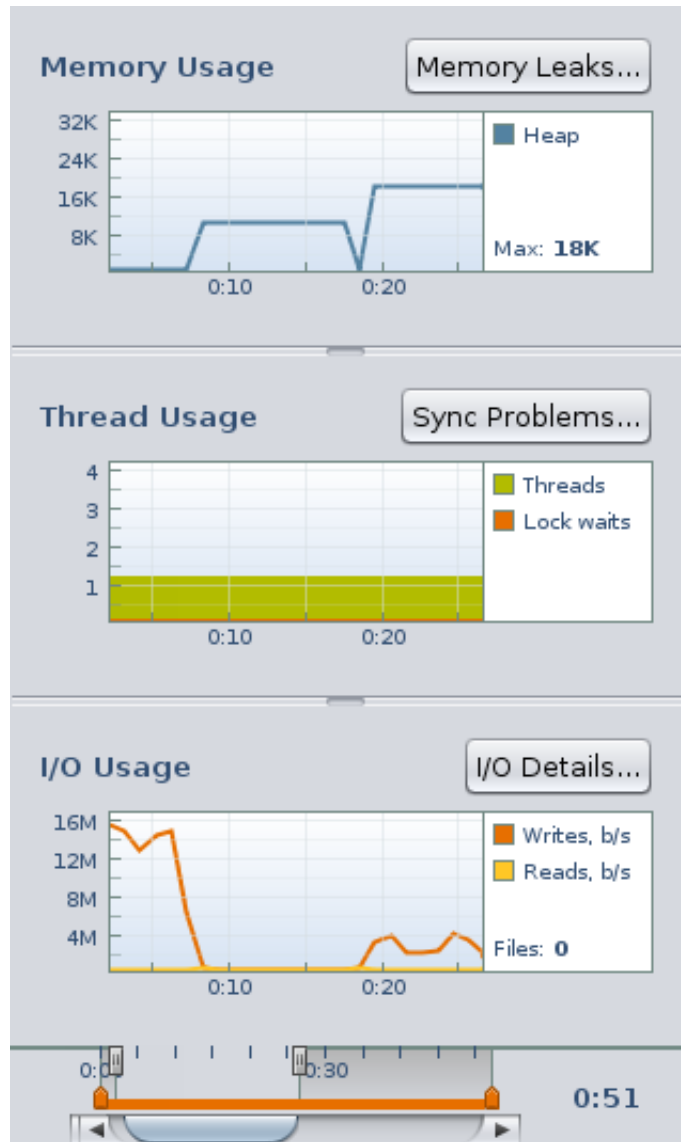
```

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.

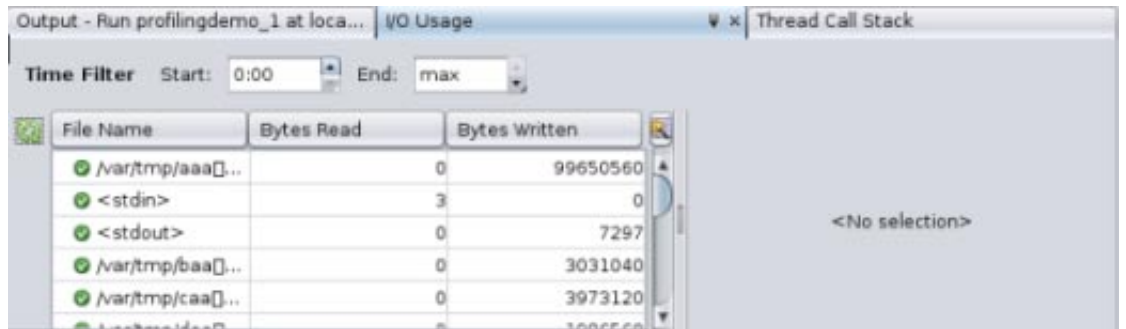
## Exploring I/O Usage

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

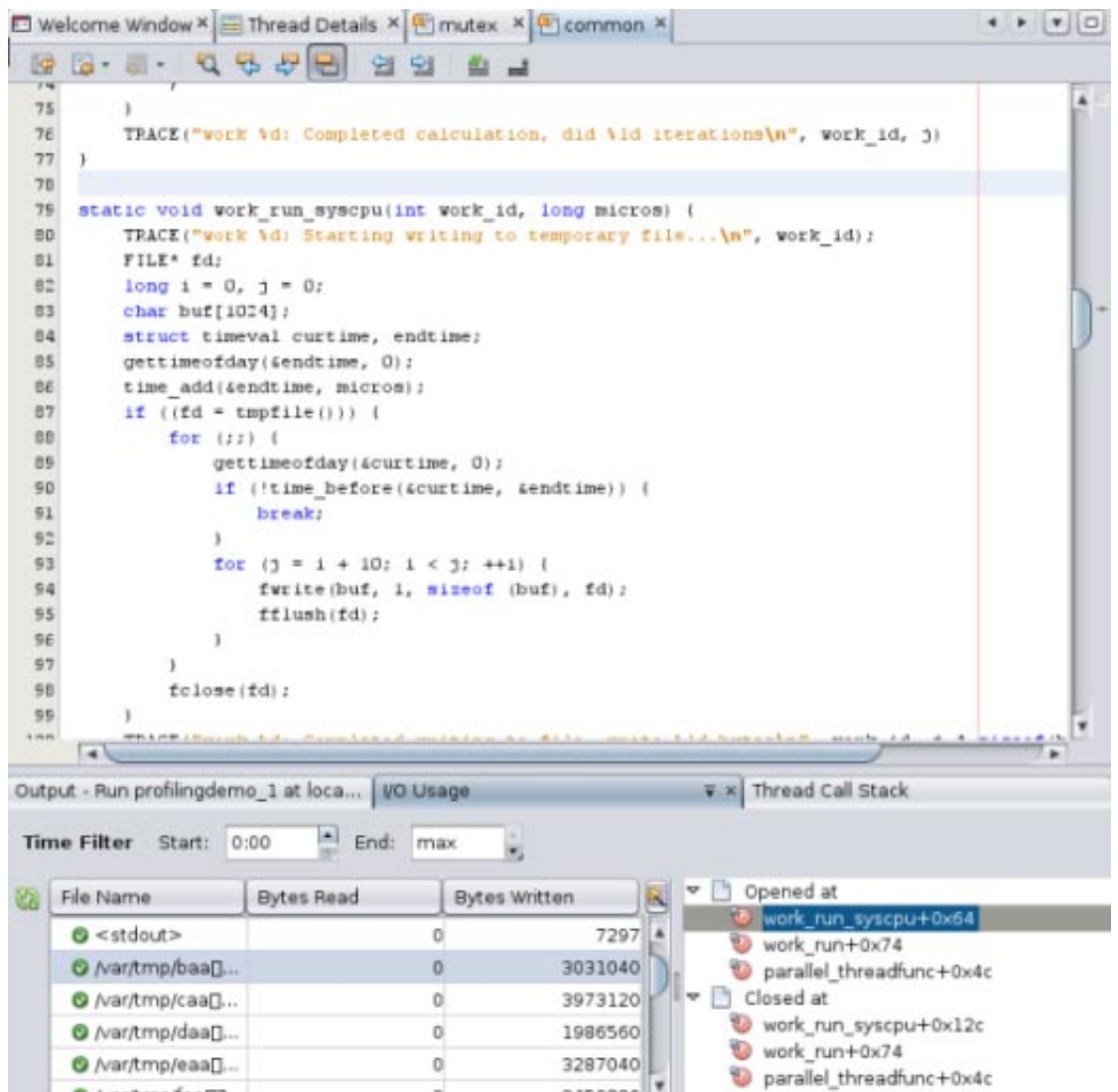
1. The following image shows the I/O usage at the beginning of the run, during the SEQUENTIAL DEMO portion in which two tasks are run one after the other in a single thread. In the first few seconds the program started, and then it was waiting for the user to press Enter. When the user pressed Enter,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



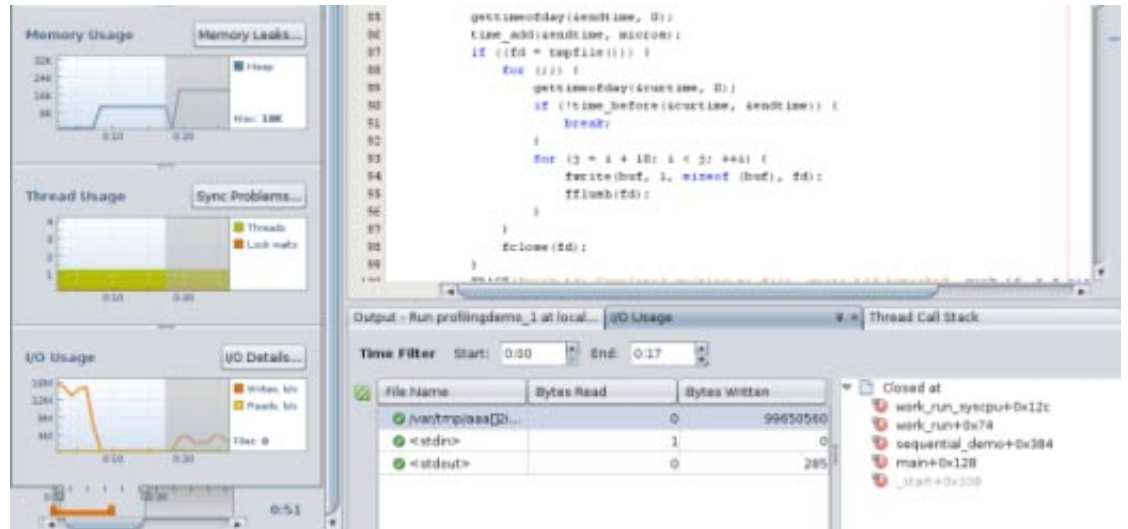
2. Notice the following:
  - The orange line shows the number of bytes written in the last second. The Profiling Demo program itself reports into the Output window that it writes a total of 79984640 bytes or about 76.3M during this run of the SEQUENTIAL DEMO. If you added up all of the data points shown 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 checkmarks 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.

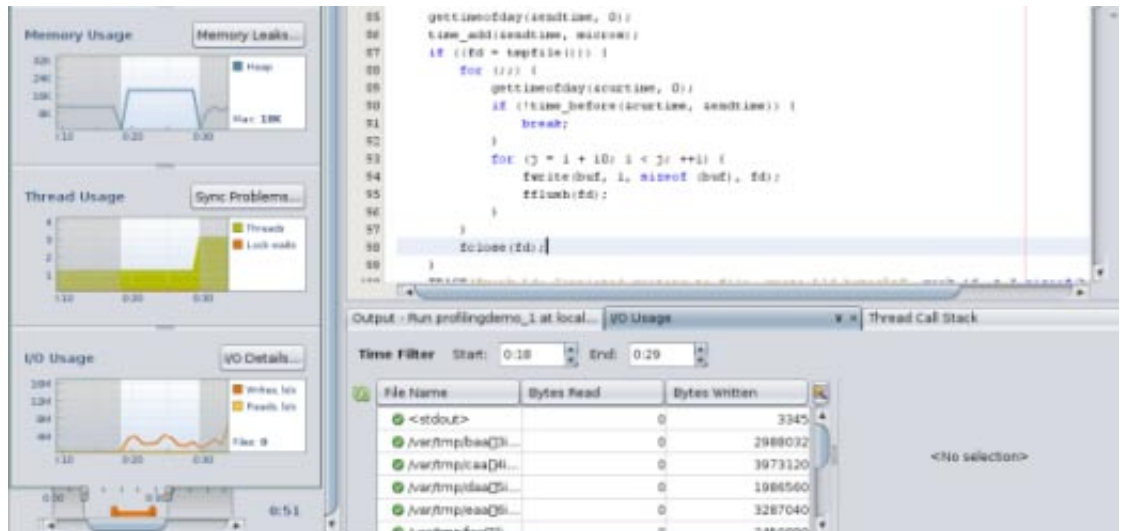


7. In the I/O Usage details tab, as in the CPU Time Per Function tab and Thread Synchronization Details tab, you can specify a time interval to look at. In the I/O Usage graph in the Run Monitor window, the write activity starts very near the beginning of the run, at the point where the SEQUENTIAL DEMO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



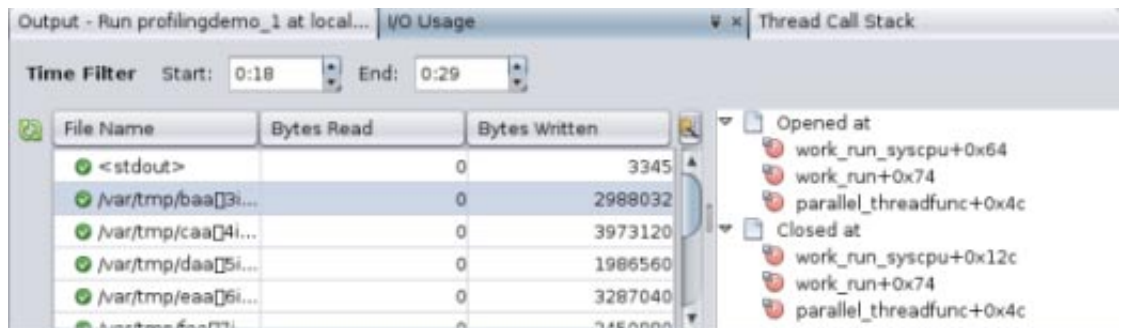
9. Notice that one temporary file is shown in the I/O Usage details tab during the sequential demo. A single thread opens, writes to the file, and then closes it. Click the file to see the functions that access it, and double-click a function to see its source code.
10. In the Run Monitor window, move 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.



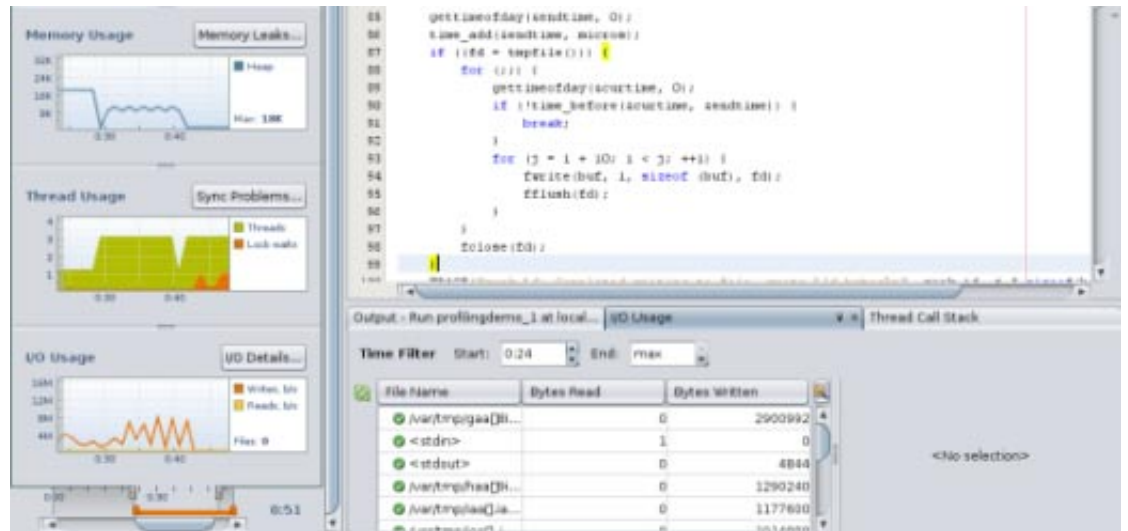


12. Notice that several temporary files are shown in the I/O Usage details tab during the PARALLEL DEMO portion of the run. Only one thread is writing to the files because each second it switches to a new file. The calculating task does not write to disk.

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.

Copyright ©2010 This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related software documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are “commercial computer software” or “commercial technical data” pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007).

Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. UNIX is a registered trademark licensed through X/Open Company, Ltd.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

821-2126

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

