Oracle® Solaris Studio 12.3: Thread Analyzer User's Guide

Part No: E21995
December 2011

ORACLE®

120118@25097

Contents

Preface

The *Thread Analyzer User's Guide* provides an introduction to the Thread Analyzer tool along with two detailed tutorials. One tutorial focuses on data race detection and the other focuses on deadlock detection. The manual also includes an appendix of APIs recognized by the Thread Analyzer and an appendix of useful tips.

Supported Platforms

This Oracle Solaris Studio release supports platforms that use the SPARC family of processor architectures running the Oracle Solaris operating system, as well as platforms that use the x86 family of processor architectures running Oracle Solaris or specific Linux systems.

This document uses the following terms to cite differences between x86 platforms:

- “x86” refers to the larger family of 64-bit and 32-bit x86 compatible products.
- “x64” points out specific 64-bit x86 compatible CPUs.
- “32-bit x86” points out specific 32-bit information about x86 based systems.

Information specific to Linux systems refers only to supported Linux x86 platforms, while information specific to Oracle Solaris systems refers only to supported Oracle Solaris platforms on SPARC and x86 systems.

For a complete list of supported hardware platforms and operating system releases, see the Oracle Solaris Studio 12.3 Release Notes.

Oracle Solaris Studio Documentation

You can find complete documentation for Oracle Solaris Studio software as follows:

- Product documentation is located at the Oracle Solaris Studio documentation web site, including release notes, reference manuals, user guides, and tutorials.
- Online help for the Code Analyzer, the Performance Analyzer, the Thread Analyzer, dbxtool, DLight, and the IDE is available through the Help menu, as well as through the F1 key and Help buttons on many windows and dialog boxes, in these tools.
- Man pages for command-line tools describe a tool's command options.

Resources for Developers

Visit the Oracle Technical Network web site to find these resources for developers using Oracle Solaris Studio:

- Articles on programming techniques and best practices
- Links to complete documentation for recent releases of the software
- Information on support levels
- User discussion forums

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–1 Typographic Conventions

Typeface	Description	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX system prompt and superuser prompt for shells that are included in the Oracle Solaris OS. Note that the default system prompt that is displayed in command examples varies, depending on the Oracle Solaris release.

TABLE P–2 Shell Prompts

Shell	Prompt
Bash shell, Korn shell, and Bourne shell	`$`
Bash shell, Korn shell, and Bourne shell for superuser	`#`
C shell	`machine_name%`
C shell for superuser	`machine_name#`

CHAPTER 1

What is the Thread Analyzer and What Does It Do?

The Thread Analyzer is an Oracle Solaris Studio tool that you can use to analyze the execution of a multithreaded program. The Thread Analyzer can detect multithreaded programming errors such as data races and deadlocks in code that is written using the POSIX thread API, the Solaris thread API, OpenMP directives, or a mix of these.

This chapter discusses the following topics:

1.1 Getting Started With the Thread Analyzer

The Thread Analyzer is a specialized view of the Performance Analyzer that is designed for examining thread analysis experiments. A separate command, `tha`, is used to start the Performance Analyzer with this specialized view, and the tool when started in this way is known as the Thread Analyzer.

The Thread Analyzer can show data races and deadlocks in experiments that you can generate specifically for examining these types of data, as explained in this document.

1.1.1 What is a Data Race?

The Thread Analyzer detects data races that occur during the execution of a multithreaded process. A data race occurs when all of the following are true:

- Two or more threads in a *single process* access the same memory location concurrently
- At least one of the accesses is for writing

- The threads are not using any exclusive locks to control their accesses to that memory

When these three conditions hold, the order of accesses is non-deterministic, and the computation may give different results from run to run depending on that order. Some data races may be benign (for example, when the memory access is used for a busy-wait), but many data races are bugs in the program.

The Thread Analyzer works on a multithreaded program written using the POSIX thread API, Solaris thread API, OpenMP, or a mix of these.

1.1.2 What is a Deadlock?

Deadlock describes a condition in which two or more threads are blocked forever because they are waiting for each other. There are many causes of deadlocks. The Thread Analyzer detects deadlocks that are caused by the inappropriate use of mutual exclusion locks. This type of deadlock is commonly encountered in multithreaded applications.

A process with two or more threads can deadlock when all of the following conditions are true:

- Threads that are already holding locks request new locks
- The requests for new locks are made concurrently
- Two or more threads form a circular chain in which each thread waits for a lock which is held by the next thread in the chain

Here is a simple example of a deadlock condition:

– Thread 1 holds lock A and requests lock B
– Thread 2 holds lock B and requests lock A

A deadlock can be of two types: A *potential* deadlock or an *actual* deadlock. A potential deadlock does not necessarily occur in a given run, but can occur in any execution of the program depending on the scheduling of threads and the timing of lock requests by the threads. An actual deadlock is one that occurs during the execution of a program. An actual deadlock causes the threads involved to hang, but may or may not cause the whole process to hang.

1.2 The Thread Analyzer Usage Model

The following steps show the process by which you can troubleshoot your multithreaded program with the Thread Analyzer.

1. Instrument the program, if doing data race detection.
2. Create a data-race-detection or deadlock-detection experiment.
3. Examine the experiment result and establish whether or not the multithreaded programming conflicts revealed by the Thread Analyzer are legitimate bugs or benign phenomena.

4. Fix the legitimate bugs and create additional experiments (step 2 above) with varied factors such as different input data, a different number of threads, varied loop schedules or even different hardware. This repetition helps locate non-deterministic problems.

Steps 1 through 3 above are described in the following sections.

1.2.1 Usage Model for Detecting Data Races

You must perform three steps to detect data races:

1. Instrument the code to enable data race detection
2. Create an experiment on the instrumented code
3. Examine the experiment for data races

1.2.1.1 Instrument the Code for Data Race Detection

To enable data race detection in an application, the code must first be instrumented to monitor memory accesses at runtime. You can instrument your code at the application source-level during compilation, or at the application binary-level by running an additional tool on the binary.

Source-level instrumentation is done by the compiler when you use a special option. You can also specify the optimization level and other compiler options to use. Source-level instrumentation can result in faster runtime since the compiler can do some analysis and instrument fewer memory accesses.

Binary-level instrumentation is useful when the source code is not available. You might also use binary instrumentation if you have the source code, but cannot compile shared libraries that are used by the application. Binary instrumentation using the `discover` tool instruments the binary as well as all shared libraries as they are opened.

Source-level Instrumentation

To instrument at the source level, compile the source code with the special compiler option:

`-xinstrument=datarace`

With this compiler option, the code generated by the compiler will be instrumented for data race detection.

The `-g` compiler option should also be used when building application binaries. This option causes extra data to be generated which allows the Thread Analyzer to display source code and line number information when reporting data races.

Binary-level Instrumentation

To instrument at the binary level, you must use the `discover` tool. If the binary is named `a.out`, you can create an instrumented binary `a.outi` by executing:

```
discover -i datarace -o a.outi a.out
```

The `discover` tool automatically instruments all shared libraries as they are opened, whether they are statically linked in the program or opened dynamically by `dlopen()`. By default, instrumented copies of libraries are cached in the directory `$HOME/SUNW_Bit_Cache`.

Some useful `discover` command line options are shown below. See the `discover`(1) man page for details.

`-o` *file*	Output the instrumented binary to the specified file name
`-N` *lib*	Do not instrument the specified library
`-T`	Do not instrument any libraries
`-D` *dir*	Change the cache directory to *dir*

For instrumenting a program's binary code to detect data races, the `discover` tool requires the input binary to be compiled under the following conditions:

- Operating system version must be at least Oracle Solaris 10 Update 5 or Oracle Solaris 11.
- Compiler must be from a release no earlier than Oracle Solaris Studio 12 Update 1.
- One of the compiler optimization flags `(-xO1, -xO2, -xO3, -xO4, -xO5)` must be used.
- The `-g` compiler option should also be used to enable the Thread Analyzer to display source code and line number information when reporting data races.

You might also be able to use the `discover` tool on an earlier Solaris version running on a SPARC-based system if the binary was compiled with the compiler option `-xbinopt=prepare`. See the `cc`(1), `CC`(1), or `f95`(1) man pages for information about this compiler option.

1.2.1.2 Create an Experiment on the Instrumented Application

To create a data-race-detection experiment, use the `collect` command with the `-r race` flag to run the application and collect experiment data during the execution of the process. When you use the `-r race` option, the collected data includes pairs of data accesses that constitute a race.

1.2.1.3 Examine the Experiment for Data Races

You can examine the data-race-detection experiment with the `tha` command, which starts the Thread Analyzer graphical user interface. You can also use the `er_print` command-line interface.

1.2.2 Usage Model for Detecting Deadlocks

Two steps are involved in detecting deadlocks:

1. Create a deadlock detection experiment.
2. Examine the experiment for deadlocks.

1.2.2.1 Create an Experiment for Detecting Deadlocks

To create a deadlock-detection experiment, use the `collect` command with the `-r deadlock` flag to run the application and collect experiment data during the execution of the process. When you use the `-r deadlock` option, the collected data includes lock holds and lock requests that form a circular chain.

1.2.2.2 Examine the Experiment for Deadlocks

You can examine the deadlock-detection experiment with the `tha` command, which starts the Thread Analyzer graphical user interface. You can also use the `er_print` command-line interface.

1.2.3 Usage Model for Detecting Data Races and Deadlocks

If you want to detect data races and deadlocks at the same time, follow the three steps in "1.2.1 Usage Model for Detecting Data Races" on page 13 for detecting data races, but use the `collect` command with the `-r race,deadlock` flag to run the application. The experiment will contain both race-detection and deadlock-detection data.

1.3 Thread Analyzer Interface

You can start the Thread Analyzer by using the `tha` command.

The Thread Analyzer interface is the Performance Analyzer interface that is streamlined for multithreaded program analysis. Instead of the usual Performance Analyzer tabs, you see the Races, Deadlocks, Dual Source, Race Details, and Deadlock Details tabs. If you use the Performance Analyzer to look at multithreaded program experiments you see the traditional Performance Analyzer tabs such as Functions, Callers-Callees, Disassembly, along with the tabs for data races and deadlocks.

CHAPTER 2

The Data Race Tutorial

The following is a detailed tutorial on how to detect and fix data races with the Thread Analyzer.

The tutorial is divided into the following sections:

2.1 Data Race Tutorial Source Files

This tutorial relies on two programs, both of which contain data races:

- The first program finds prime numbers. It is written with C and is parallelized with OpenMP directives. The source file is called `prime_omp.c`.
- The second program also finds prime numbers and is also written with C. However, it is parallelized with POSIX threads instead of OpenMP directives. The source file is called `prime_pthr.c`.

2.1.1 Getting the Data Race Tutorial Source Files

You can download the source files used in this tutorial from the samples download area (`http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/solaris-studio-samples-1408618.html`) of the Oracle Solaris Studio developer portal.

After you download and unpack the sample files, you can find the samples in the `SolarisStudioSampleApplications/ThreadAnalyzer` directory. The samples are located in the `prime_omp` and `prime_pthr` subdirectories. Each sample directory includes a `Makefile` and a `DEMO` file of instructions, but this tutorial does not follow those instructions or use the `Makefile`. Instead, you are instructed to execute commands individually.

To follow the tutorial, you can copy the `prime_omp.c` and `prime_pthr.c` files from the samples directories to a different directory, or you can create your own files and copy the code from the following code listings.

2.1.2 Source Code for `prime_omp.c`

The source code for `prime_omp.c` is shown below:

```
/*
 * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All Rights Reserved.
 * @(#)prime_omp.c 1.3 (Oracle) 10/03/26
 */

#include <stdio.h>
#include <math.h>
#include <omp.h>

#define THREADS 4
#define N 10000

int primes[N];
int pflag[N];

int is_prime(int v)
{
 int i;
 int bound = floor(sqrt(v)) + 1;

 for (i = 2; i < bound; i++) {
        /* no need to check against known composites */
        if (!pflag[i])
            continue;
        if (v % i == 0) {
            pflag[v] = 0;
            return 0;
        }
    }
    return (v > 1);
}

int main(int argn, char **argv)
{
    int i;
    int total = 0;

#ifdef _OPENMP
    omp_set_dynamic(0);
    omp_set_num_threads(THREADS);
#endif

```

```
    for (i = 0; i < N; i++) {
        pflag[i] = 1;
    }

    #pragma omp parallel for
    for (i = 2; i < N; i++) {
        if ( is_prime(i) ) {
            primes[total] = i;
            total++;
        }
    }

    printf("Number of prime numbers between 2 and %d: %d\n",
           N, total);

    return 0;
}
```

2.1.3 Source Code for `prime_pthr.c`

The source code for `prime_pthr.c` is shown below:

```
/*
 * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All Rights Reserved.
 * @(#)prime_pthr.c 1.4 (Oracle) 10/03/26
 */

#include <stdio.h>
#include <math.h>
#include <pthread.h>

#define THREADS 4
#define N 10000

int primes[N];
int pflag[N];
int total = 0;

int is_prime(int v)
{
    int i;
    int bound = floor(sqrt(v)) + 1;

    for (i = 2; i < bound; i++) {
        /* no need to check against known composites */
        if (!pflag[i])
            continue;
        if (v % i == 0) {
            pflag[v] = 0;
            return 0;
        }
    }
    return (v > 1);
}

void *work(void *arg)
```

```
{
    int start;
    int end;
    int i;

    start = (N/THREADS) * (*(int *)arg);
    end = start + N/THREADS;
    for (i = start; i < end; i++) {
        if ( is_prime(i) ) {
            primes[total] = i;
            total++;
        }
    }
    return NULL;
}

int main(int argn, char **argv)
{
    int i;
    pthread_t tids[THREADS-1];

    for (i = 0; i < N; i++) {
        pflag[i] = 1;
    }

    for (i = 0; i < THREADS-1; i++) {
        pthread_create(&tids[i], NULL, work, (void *)&i);
    }

    i = THREADS-1;
    work((void *)&i);

    for (i = 0; i < THREADS-1; i++) {
        pthread_join(tids[i], NULL);
    }

    printf("Number of prime numbers between 2 and %d: %d\n",
           N, total);

    return 0;
}
```

2.1.3.1 Effect of Data Races in `prime_omp.c` and `prime_pthr.c`

The order of memory accesses is non-deterministic when code contains a race condition and the computation gives different results from run to run. The correct answer in the `prime_omp` and `prime_pthr` programs is 1229.

You can compile and run the examples so you can see that each execution of `prime_omp` or `prime_pthr` produces incorrect and inconsistent results because of the data races in the code.

In the following example, type the commands in bold to compile and run the `prime_omp` program:

```
% cc -xopenmp=noopt  -o prime_omp prime_omp.c -lm
%
% ./prime_omp
```

```
Number of prime numbers between 2 and 10000: 1229
% ./prime_omp
Number of prime numbers between 2 and 10000: 1228
% ./prime_omp
Number of prime numbers between 2 and 10000: 1180
```

In the following example, type the commands in bold to compile and run the `prime_pthr` program:

```
% cc -mt -o prime_pthr prime_pthr.c -lm
%
% ./prime_pthr
Number of prime numbers between 2 and 10000: 1140
% ./prime_pthr
Number of prime numbers between 2 and 10000: 1122
% ./prime_pthr
Number of prime numbers between 2 and 10000: 1141
```

Notice the inconsistency of the results of the three runs of each program. You may need to run the programs more than three times to see inconsistent results.

Next you instrument the code and create experiments so you can find where the data races are occurring.

2.2 How to Use the Thread Analyzer to Find Data Races

The Thread Analyzer follows the same "collect-analyze" model that the Oracle Solaris Studio Performance Analyzer uses.

There are three steps involved in using the Thread Analyzer:

1. "2.2.1 Instrument the Code" on page 21
2. "2.2.2 Create a Data Race Detection Experiment" on page 23
3. "2.2.3 Examine the Data Race Detection Experiment" on page 23

2.2.1 Instrument the Code

In order to enable data race detection in a program, the code must first be instrumented to monitor memory accesses at runtime. The instrumentation can be done on application source code, or on application binaries that have been compiled with certain Oracle compiler optimization flags. The tutorial shows how to use both methods of instrumenting the programs.

2.2.1.1 To Instrument Source Code

To instrument the source code, you must compile the application with the special compiler option `-xinstrument=datarace`. This option instructs the compiler to instrument the generated code for data race detection.

Add the `-xinstrument=datarace` compiler option to the existing set of options you use to compile your program.

Note – Be sure to also specify the `-g` option when you compile your program with `-xinstrument=datarace` to generate additional information to enable the Analyzer's full capabilities. Do not specify a high level of optimization when compiling your program for race detection. Compile an OpenMP program with `-xopenmp=noopt`. The information reported, such as line numbers and call stacks, may be incorrect when a high optimization level is used.

You can use the following commands for instrumenting the source code for the tutorial:

```
% cc -xinstrument=datarace -g -xopenmp=noopt -o prime_omp_inst prime_omp.c -lm
```

```
% cc -xinstrument=datarace -g -o prime_pthr_inst prime_pthr.c -lm
```

Notice that in the example, the output file is specified with `_inst` at the end so that you can tell that the binary is the instrumented binary. This is not required.

2.2.1.2 To Instrument Binary Code

To instrument a program's binary code instead of the source code, you need to use the `discover` tool, which is included in Oracle Solaris Studio and is documented in the `discover`(1) man page and *Oracle Solaris Studio 12.3: Discover and Uncover User's Guide*.

See "Binary-level Instrumentation" on page 14 for more information about requirements for binary instrumentation.

For the tutorial examples, type the following command to compile the code with optimization level 3 to create binaries that can be used by `discover`.

```
% cc -xopenmp=noopt -g -o prime_omp_opt prime_omp.c -lm
```

```
% cc -g -O3 -o prime_pthr_opt prime_pthr.c -lm
```

Then run `discover` on the `prime_omp_opt` and `prime_pthr_opt` optimized binaries that you created:

```
% discover -i datarace -o prime_omp_disc prime_omp_opt
```

```
% discover -i datarace -o prime_pthr_disc prime_pthr_opt
```

These commands create instrumented binaries, `prime_omp_disc` and `prime_pthr_disc` that you can use with `collect` to create experiments that you can examine with the Thread Analyzer.

2.2.2 Create a Data Race Detection Experiment

Use the `collect` command with the `-r race` flag to run the program and create a data race detection experiment during the execution of the process. For OpenMP programs, make sure that the number of threads used is larger than one. In the tutorial samples, four threads are used.

To create experiments from the binaries that you created by instrumenting the source code:

```
% collect -r race -o prime_omp_inst.er prime_omp_inst
```

```
% collect -r race -o prime_pthr_inst.er prime_pthr_inst
```

To create experiments from the binaries that you created by using the `discover` tool:

```
% collect -r race -o prime_omp_disc.er prime_omp_disc
```

```
% collect -r race -o prime_pthr_disc.er prime_pthr_disc
```

To increase the likelihood of detecting data races, it is recommended that you create several data race detection experiments using `collect` with the `-r race` flag. Use a different number of threads and different input data for each experiment.

For example, in `prime_omp.c`, the number of threads is set by the following line:

```
#define THREADS 4
```

The number of threads can be changed by changing 4 in the above to some other integer larger than 1, for example 8.

The following line in `prime_omp.c` limits the program to look for prime numbers between 2 and 3000:

```
#define N 3000
```

You can provide different input data by changing the value of `N` to make the program do more or less work.

2.2.3 Examine the Data Race Detection Experiment

You can examine a data race detection experiment with the Thread Analyzer, the Performance Analyzer, or the `er_print` utility. Both the Thread Analyzer and the Performance Analyzer present a GUI interface; the Thread Analyzer presents a simplified set of default tabs, but is otherwise identical to the Performance Analyzer.

2.2.3.1 Using Thread Analyzer to View the Data Race Experiment

To start the Thread Analyzer, type the following command:

```
% tha
```

The Thread Analyzer GUI has a menu bar, a tool bar, and a split pane that contains tabs for the various displays.

On the left-hand pane, the following three tabs are shown by default:

- The Races tab shows a list of data races detected in the program, and associated call stack traces. This tab is selected by default.
- The Dual Source tab shows the two source locations corresponding to the two accesses of a selected data race. The source line where a data race access occurred is highlighted.
- The Experiments tab shows the load objects in the experiment, and lists error and warning messages.

On the right-hand pane of the Thread Analyzer display, the following two tabs are shown:

- The Summary tab shows summary information about a data race access selected from the Races tab.
- The Race Details tab shows detailed information about a data race or call stack trace selected from the Races tab.

2.2.3.2 Using `er_print` to View the Data Race Experiment

The `er_print` utility presents a command-line interface. You can use the `er_print` utility in an interactive session and specify sub-commands during the session. You can also use command-line options to specify sub-commands non-interactively.

The following sub-commands are useful for examining races with the `er_print` utility:

- `-races`

  This reports any data races revealed in the experiment. Specify `races` at the `(er_print)` prompt or `-races` on the `er_print` command line.
- `-rdetail` *race_id*

  This displays detailed information about the data race with the specified *race_id*. Specify `rdetail` at the `(er_print)` prompt or `-rdetail` on the `er_print` command line. If the specified *race_id* is **`all`**, then detailed information about all data races will be displayed. Otherwise, specify a single race number such as **`1`** for the first data race.
- `-header`

  This displays descriptive information about the experiment, and reports any errors or warnings. Specify `header` at the `(er_print)` prompt or `-header` on the command line.

Refer to the collect(1), tha(1), analyzer(1), and er_print(1) man pages for more information.

2.3 Understanding the Experiment Results

This section shows how to use both the er_print command line and the Thread Analyzer GUI to display the following information about each detected data race:

- The unique ID of the data race.
- The virtual address, Vaddr, associated with the data race. If there is more than one virtual address, then the label Multiple Addresses is displayed in parentheses.
- The memory accesses to the virtual address, Vaddr by two different threads. The type of the access (read or write) is shown, as well as the function, offset, and line number in the source code where the access occurred.
- The total number of call stack traces associated with the data race. Each trace refers to the pair of thread call stacks at the time the two data race accesses occurred. If you are using the GUI, the two call stacks are displayed in the Race Details tab when an individual call stack trace is selected. If you are using the er_print utility, the two call stacks will be displayed by the rdetail command.

2.3.1 Data Races in prime_omp.c

To examine data races in prime_omp.c, you can use one of the experiments you created in “2.2.2 Create a Data Race Detection Experiment” on page 23.

To show the data race information in the prime_omp_instr.er experiment with er_print, type the following command.

```
% er_print prime_omp_inst.er
```

At the (er_print) prompt, type **races** to see output similar to the following:

```
(er_print)races

Total Races:  2 Experiment:  prime_omp_inst.er

Race #1, Vaddr: 0x21ca8
      Access 1: Write, is_prime,
                       line 26 in "prime_omp.c"
      Access 2: Read,  is_prime,
                       line 23 in "prime_omp.c"
  Total Callstack Traces: 1

Race #2, Vaddr: (Multiple Addresses)
      Access 1: Write, main,
```

```
                             line 50 in "prime_omp.c"
          Access 2: Write, main,
                             line 50 in "prime_omp.c"
    Total Callstack Traces: 2
(er_print)
```

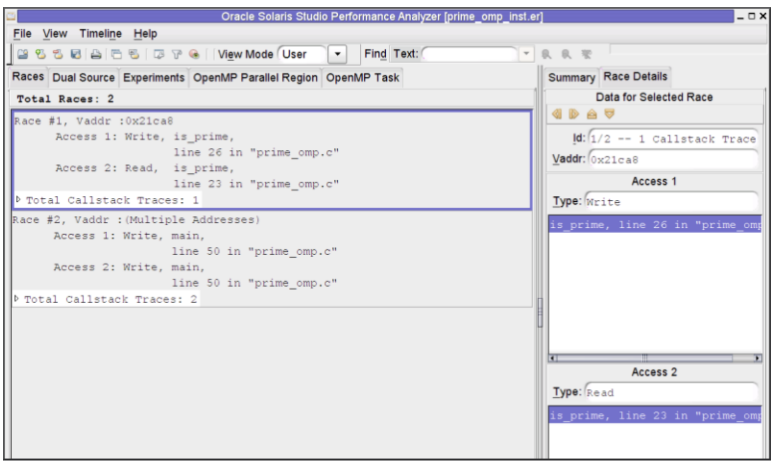
Two data races occurred during this particular run of the program.

To open the `prime_omp_inst.er` experiment in the Thread Analyzer, type the following command:

```
% tha prime_omp_inst.er
```

The following screen shot shows the races that were detected in `prime_omp.c` as displayed by the Thread Analyzer.

FIGURE 2–1 Data Races Detected in `prime_omp.c`



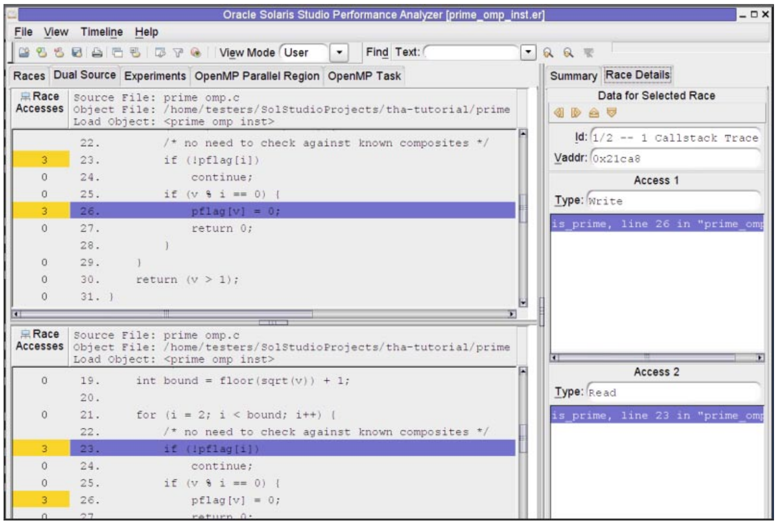
Two data races are shown in `prime_omp.c`:

- `Race #1` shows a race between a write in the function `is_prime` on line 26 and a read in the same function on line 23. If you look at the source code you can see that on these lines, the `pflag[ ]` array is being accessed. In the Thread Analyzer, you can click the Dual Source tab to easily see the source code at both line numbers along with metrics showing the number of race accesses on the affected lines of code.
- `Race #2` shows a race between two writes to line 50 of the `main` function. Click the Dual Source tab to see that there are multiple attempts to access the value of the `primes [ ]` array in line 50.

  `Race #2` represents a group of data races that occur in different elements of the array `primes[ ]`. This is indicated by the `Vaddr` specified as `Multiple Addresses`.

The Dual Source tab in the Thread Analyzer allows you to see the two source locations associated with a data race at the same time. For example, select `Race #2` for `prime_pthr.c` in the Races tab and then click on the Dual Source tab. You will see something similar to the following.

FIGURE 2–2 Source Code of Data Races Detected in `prime_omp.c`



Tip – You might need to drag the mouse on the header of each source panel to see the Race Accesses metrics in the left margin of the Dual Source tab.

2.3.2 Data Races in `prime_pthr.c`

To examine data races in `prime_pthr.c`, you can use one of the experiments you created in "2.2.2 Create a Data Race Detection Experiment" on page 23.

To show the data race information in the `prime_pthr_instr.er` experiment with `er_print`, type the following command:

```
% er_print prime_pthr_inst.er
```

At the `(er_print)` prompt, type **`races`** to see output similar to the following:

```
(er_print) races

Total Races:  4 Experiment:  prime_pthr_inst.er
```

```
Race #1, Vaddr: (Multiple Addresses)
      Access 1: Write, is_prime + 0x00000270,
                       line 27 in "prime_pthr.c"
      Access 2: Write, is_prime + 0x00000270,
                       line 27 in "prime_pthr.c"
  Total Callstack Traces: 2

Race #2, Vaddr: 0xffbfe714
      Access 1: Write, main + 0x0000025C,
                       line 60 in "prime_pthr.c"
      Access 2: Read,  work + 0x00000070,
                       line 40 in "prime_pthr.c"
  Total Callstack Traces: 1

Race #3, Vaddr: (Multiple Addresses)
      Access 1: Write, work + 0x00000150,
                       line 44 in "prime_pthr.c"
      Access 2: Write, work + 0x00000150,
                       line 44 in "prime_pthr.c"
  Total Callstack Traces: 2

Race #4, Vaddr: 0x21a90
      Access 1: Write, work + 0x00000198,
                       line 45 in "prime_pthr.c"
      Access 2: Write, work + 0x00000198,
                       line 45 in "prime_pthr.c"
  Total Callstack Traces: 2
(er_print)
```

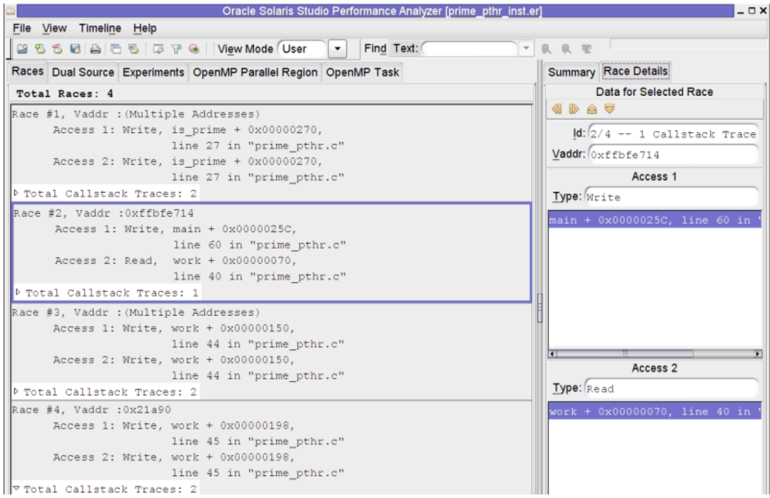
Four data races occurred during this particular run of the program.

To open the `prime_pthr_inst.er` experiment in the Thread Analyzer, type the following command:

```
% tha prime_pthr_inst.er
```

The following screen shot shows the races detected in `prime_pthr.c` as displayed by the Thread Analyzer. Notice that they are the same as the races shown by `er_print`.

FIGURE 2–3 Data Races Detected in `prime_pthr.c`

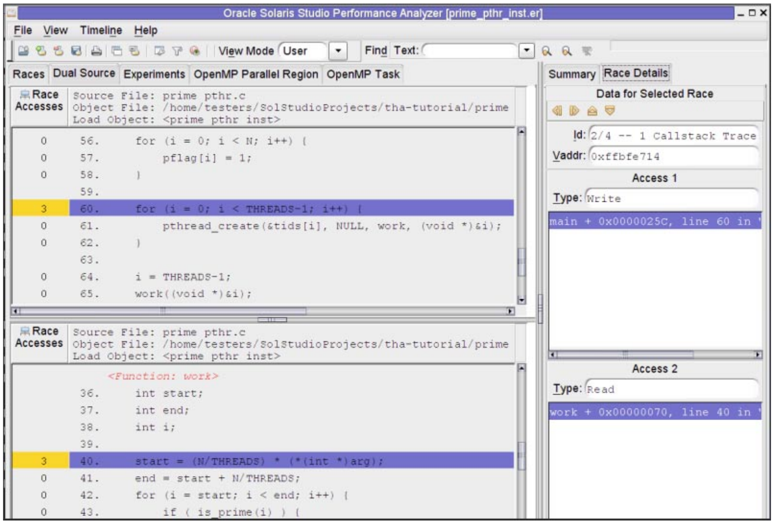


Four data races are shown in `prime_pthr.c`:

- `Race #1` is a data race between a write to the `pflag[ ]` array in function `is_prime` on line 27 and another write to `pflag[ ]` on the same line.
- `Race #2` is a data race between a write on line 60 to the memory location named `i` in `main()` and a read on line 40 from the same memory location (named `*arg` in `work()`).
- `Race #3` is a data race between a write to `primes[total]` on line 44 and another write to `primes[total]` the same line.
- `Race #4` is a data race between a write to `total` on line 45 and another write to `total` on the same line.

If you select `Race #2` and then click the Dual Source tab, you see the two source locations, similar to the following screen shot.

FIGURE 2–4 Source Code Details of a Data Race

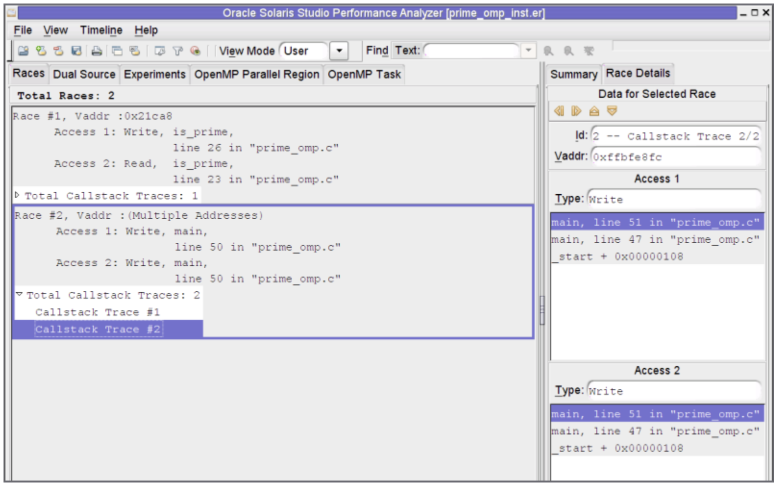


The first access for Race #2 is at line 60 and is shown in the top panel. The second access is at line 40 and is shown in the bottom panel. The Race Accesses metric is highlighted at the left of the source code. This metric gives a count of the number of times a data race access was reported on that line.

2.3.3 Call Stack Traces of Data Races

Each data race listed in the Races tab of the Thread Analyzer also has one or more associated Call Stack Traces. The call stacks show the execution paths through the code that lead to a data race. When you click a Call Stack Trace, the Race Details tab in the right panel shows the function calls that lead to the data race.

FIGURE 2–5 Races Tab with Call Stack Traces for `prime_omp.c`



2.4 Diagnosing the Cause of a Data Race

This section provides a basic strategy to diagnosing the cause of data races.

2.4.1 Check Whether or Not the Data Race is a False Positive

A false positive data race is a data race that is reported by the Thread Analyzer, but has actually not occurred. The Thread Analyzer tries to reduce the number of false positives reported. However, there are cases where the tool is not able to do a precise job and may report false positive data races.

You can ignore a false positive data race because it is not a genuine data race and, therefore, does not affect the behavior of the program.

See "2.5 False Positives" on page 35 for some examples of false positive data races. For information on how to remove false positive data races from the report, see "A.1 Thread Analyzer User APIs" on page 67.

2.4.2 Check Whether or Not the Data Race is Benign

A benign data race is an intentional data race whose existence does not affect the correctness of the program.

Some multithreaded applications intentionally use code that may cause data races. Since the data races are there by design, no fix is required. In some cases, however, it is quite tricky to get such codes to run correctly. These data races should be reviewed carefully.

See “2.5 False Positives” on page 35 for more detailed information about benign races.

2.4.3 Fix the Bug, Not the Data Race

The Thread Analyzer can help find data races in the program, but it cannot automatically find bugs in the program nor suggest ways to fix the data races found. A data race may have been introduced by a bug. It is important to find and fix the bug. Merely removing the data race is not the right approach, and could make further debugging even more difficult.

2.4.3.1 Fixing Bugs in `prime_omp.c`

Here's how to fix the bug in `prime_omp.c`. See “2.1.2 Source Code for `prime_omp.c`” on page 18 for a complete file listing.

Move lines 50 and 51 into a `critical` section in order to remove the data race on elements of the array `primes[ ]`.

```
47      #pragma omp parallel for
48      for (i = 2; i < N; i++) {
49          if ( is_prime(i) ) {
                 #pragma omp critical

                 {
50                  primes[total] = i;
51                  total++;
                 }
52          }
53     }
```

You could also move lines 50 and 51 into two `critical` sections as follows, but this change fails to correct the program:

```
47      #pragma omp parallel for
48      for (i = 2; i < N; i++) {
49          if ( is_prime(i) ) {
                 #pragma omp critical
                 {
50                  primes[total] = i;
                 }
                 #pragma omp critical
                 {
51                  total++;
                 }
52          }
53     }
```

The critical sections around lines 50 and 51 get rid of the data race because the threads are using exclusive locks to control their accesses to the `primes[ ]` array. However, the program is still incorrect. Two threads might update the same element of `primes[ ]` using the same value of `total`, and some elements of `primes[ ]` may not be assigned a value at all.

The second data race, between a read from `pflag[ ]` from line 23 and a write to `pflag[ ]` from line 26, is actually a benign race because it does not lead to incorrect results. It is not essential to fix benign data races.

2.4.3.2 Fixing Bugs in `prime_pthr.c`

Here's how to fix the bug in `prime_pthr.c`. See “2.1.3 Source Code for `prime_pthr.c`” on page 19 for a complete file listing.

Use a single mutex to remove the data race on `prime[ ]` at line 44, as well as the data race on `total` at line 45.

The data race between the write to `i` on line 60 and the read from the same memory location (named `*arg`) on line 40, as well as the data race on `pflag[ ]` on line 27, reveal a problem in the shared access to the variable `i` by different threads. The initial thread in `prime_pthr.c` creates the child threads in a loop in lines 60-62, and dispatches them to work on the function `work()`. The loop index `i` is passed to `work()` by address. Since all threads access the same memory location for `i`, the value of `i` for each thread will not remain unique, but will change as the initial thread increments the loop index. As different threads use the same value of `i`, the data races occur. One way to fix the problem is to pass `i` to `work()` by value, instead of by address.

Here is the corrected version of `prime_pthr.c`:

```
 1  /*
 2   * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All Rights Reserved.
 3   * @(#)prime_pthr_fixed.c 1.3 (Oracle) 10/03/26
 4   */
 5
 6  #include <stdio.h>
 7  #include <math.h>
 8  #include <pthread.h>
 9
10  #define THREADS 4
11  #define N 10000
12
13  int primes[N];
14  int pflag[N];
15  int total = 0;
16  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
17
18  int is_prime(int v)
19  {
20      int i;
21      int bound = floor(sqrt(v)) + 1;
22
23      for (i = 2; i < bound; i++) {
24          /* no need to check against known composites */
25          if (!pflag[i])
26              continue;
27          if (v % i == 0) {
28              pflag[v] = 0;
29              return 0;
```

```
        }
    }
    return (v > 1);
}

void *work(void *arg)
{
    int start;
    int end;
    int i;

    start = (N/THREADS) * ((int)arg) ;
    end = start + N/THREADS;
    for (i = start; i < end; i++) {
        if ( is_prime(i) ) {
            pthread_mutex_lock(&mutex);
            primes[total] = i;
            total++;
            pthread_mutex_unlock(&mutex);
        }
    }
    return NULL;
}

int main(int argn, char **argv)
{
    int i;
    pthread_t tids[THREADS-1];

    for (i = 0; i < N; i++) {
        pflag[i] = 1;
    }

    for (i = 0; i < THREADS-1; i++) {
        pthread_create(&tids[i], NULL, work, (void *)i);
    }

    i = THREADS-1;
    work((void *)i);

    for (i = 0; i < THREADS-1; i++) {
        pthread_join(tids[i], NULL);
    }

    printf("Number of prime numbers between 2 and %d: %d\n",
           N, total);

    return 0;
}
```

2.5 False Positives

Occasionally, the Thread Analyzer may report data races that have not actually occurred in the program. These are called false positives. In most cases, false positives are caused by user-defined synchronizations or by memory that is recycled by different threads. For more information, see “2.5.1 User-Defined Synchronizations” on page 35 and “2.5.2 Memory That is Recycled by Different Threads” on page 36.

2.5.1 User-Defined Synchronizations

The Thread Analyzer can recognize most standard synchronization APIs and constructs provided by OpenMP, POSIX threads, and Solaris threads. However, the tool cannot recognize user-defined synchronizations, and may report false data races if your code contains such synchronizations.

Note – In order to avoid reporting this kind of false positive data race, the Thread Analyzer provides a set of APIs that can be used to notify the tool when user-defined synchronizations are performed. See “A.1 Thread Analyzer User APIs” on page 67 for more information.

To illustrate why you might need to use the APIs, consider the following. The Thread Analyzer cannot recognize implementation of locks using CAS instructions, post and wait operations using busy-waits, and so on. Here is a typical example of a class of false positives where the program employs a common way of using POSIX thread condition variables:

```
/* Initially ready_flag is 0 */

/* Thread 1: Producer */
100   data = ...
101   pthread_mutex_lock (&mutex);
102   ready_flag = 1;
103   pthread_cond_signal (&cond);
104   pthread_mutex_unlock (&mutex);
...
/* Thread 2: Consumer */
200   pthread_mutex_lock (&mutex);
201   while (!ready_flag) {
202       pthread_cond_wait (&cond, &mutex);
203   }
204   pthread_mutex_unlock (&mutex);
205   ... = data;
```

The `pthread_cond_wait()` call is usually made within a loop that tests the predicate to protect against program errors and spurious wake-ups. The test and set of the predicate is often protected by a mutex lock. In the above code, Thread 1 produces the value for the variable `data` at line 100, sets the value of `ready_flag` to one at line 102 to indicate that the data has been produced, and then calls `pthread_cond_signal()` to wake up the consumer thread, Thread 2. Thread 2 tests the predicate (*!ready_flag*) in a loop. When it finds that the flag is set, it consumes the data at line 205.

The write of `ready_flag` at line 102 and read of `ready_flag` at line 201 are protected by the same mutex lock, so there is no data race between the two accesses and the tool recognizes that correctly.

The write of `data` at line 100 and the read of `data` at line 205 are not protected by mutex locks. However, in the program logic, the read at line 205 always happens after the write at line 100 because of the flag variable `ready_flag`. Consequently, there is no data race between these two accesses to data. However, the tool reports that there is a data race between the two accesses if the call to `pthread_cond_wait()` (line 202) is actually not called at run time. If line 102 is executed before line 201 is ever executed, then when line 201 is executed, the loop entry test fails and line 202 is skipped. The tool monitors `pthread_cond_signal()` calls and `pthread_cond_wait()` calls and can pair them to derive synchronization. When the `pthread_cond_wait()` at line 202 is not called, the tool does not know that the write at line 100 is always executed before the read at line 205. Therefore, it considers them as executed concurrently and reports a data race between them.

The `libtha`(3C) man page and "A.1 Thread Analyzer User APIs" on page 67 explain how to use the APIs to avoid reports of this kind of false positive data race.

2.5.2 Memory That is Recycled by Different Threads

Some memory management routines recycle memory that is freed by one thread for use by another thread. The Thread Analyzer is sometimes not able to recognize that the life span of the same memory location used by different threads do not overlap. When this happens, the tool may report a false positive data race. The following example illustrates this kind of false positive.

```
/*----------*/                   /*----------*/
/* Thread 1 */                   /* Thread 2 */
/*----------*/                   /*----------*/
 ptr1 = mymalloc(sizeof(data_t));
 ptr1->data = ...
 ...
 myfree(ptr1);

                                 ptr2 = mymalloc(sizeof(data_t));
                                 ptr2->data = ...
                                 ...
                                 myfree(ptr2);
```

Thread 1 and Thread 2 execute concurrently. Each thread allocates a chunk of memory that is used as its private memory. The routine `mymalloc()` may supply the memory freed by a previous call to `myfree()`. If Thread 2 calls `mymalloc()` before Thread 1 calls `myfree()`, then `ptr1` and `ptr2` get different values and there is no data race between the two threads. However, if Thread 2 calls `mymalloc()` after Thread 1 calls `myfree()`, then `ptr1` and `ptr2` may have the same value. There is no data race because Thread 1 no longer accesses that memory. However, if the tool does not know `mymalloc()` is recycling memory, it reports a data race between the write of `ptr1` data and the write of `ptr2` data. This kind of false positive often happens in C++ applications when the C++ runtime library recycles memory for temporary variables. It also often happens in user applications that implement their own memory management routines. Currently, the Thread Analyzer is able to recognize memory allocation and free operations performed with the standard `malloc()`, `calloc()`, and `realloc()` interfaces.

2.6 Benign Data Races

Some multithreaded applications intentionally allow data races in order to get better performance. A benign data race is an intentional data race whose existence does not affect the correctness of the program. The following examples demonstrate benign data races.

Note – In addition to benign data races, a large class of applications allow data races because they rely on lock-free and wait-free algorithms which are difficult to design correctly. The Thread Analyzer can help determine the locations of data races in these applications.

2.6.1 A Program for Finding Primes

The threads in `prime_omp.c` check whether an integer is a prime number by executing the function `is_prime()`.

```
16  int is_prime(int v)
17  {
18      int i;
19      int bound = floor(sqrt(v)) + 1;
20
21      for (i = 2; i < bound; i++) {
22          /* no need to check against known composites */
23          if (!pflag[i])
24              continue;
25          if (v % i == 0) {
26              pflag[v] = 0;
27              return 0;
28          }
29      }
30      return (v > 1);
31  }
```

The Thread Analyzer reports that there is a data race between the write to `pflag[ ]` on line 26 and the read of `pflag[ ]` on line 23. However, this data race is benign as it does not affect the correctness of the final result. At line 23, a thread checks whether or not `pflag[i]`, for a given value of `i` is equal to zero. If `pflag[i]` is equal to zero, that means that `i` is a known composite number (in other words, `i` is known to be non-prime). Consequently, there is no need to check whether or not `v` is divisible by `i`; you only need to check whether or not `v` is divisible by some prime number. Therefore, if `pflag[i]` is equal to zero, the thread continues to the next value of `i`. If `pflag[i]` is not equal to zero and `v` is divisible by `i`, the thread assigns zero to `pflag[v]` to indicate that `v` is not a prime number.

It does not matter, from a correctness point of view, if multiple threads check the same `pflag[ ]` element and write to it concurrently. The initial value of a `pflag[ ]` element is one. When the threads update that element, they assign it the value zero. That is, the threads store zero in the same bit in the same byte of memory for that element. On current architectures, it is safe to assume that those stores are atomic. This means that, when that element is read by a thread, the value read is either one or zero. If a thread checks a given `pflag[ ]` element (line 23) before it has been assigned the value zero, it then executes lines 25–28. If, in the meantime, another thread assigns zero to that same `pflag[ ]` element (line 26), the final result is not changed. Essentially, this means that the first thread executed lines 25–28 unnecessarily, but the final result is the same.

2.6.2 A Program that Verifies Array-Value Types

A group of threads call `check_bad_array()` concurrently to check whether any element of array `data_array` is "bad". Each thread checks a different section of the array. If a thread finds that an element is bad, it sets the value of a global shared variable `is_bad` to true.

```
20  volatile int is_bad = 0;
 ...

 100  /*
 101   * Each thread checks its assigned portion of data_array, and sets
 102   * the global flag is_bad to 1 once it finds a bad data element.
 103   */
 104  void check_bad_array(volatile data_t *data_array, unsigned int thread_id)
 105  {
 106     int i;
 107     for (i=my_start(thread_id); i<my_end(thread_id); i++) {
 108          if (is_bad)
 109              return;
 110          else {
 111              if (is_bad_element(data_array[i])) {
 112                  is_bad = 1;
 113                  return;
 114              }
 115          }
 116     }
 117  }
```

There is a data race between the read of `is_bad` on line 108 and the write to `is_bad` on line 112. However, the data race does not affect the correctness of the final result.

The initial value of `is_bad` is zero. When the threads update `is_bad`, they assign it the value one. That is, the threads store one in the same bit in the same byte of memory for `is_bad`. On current architectures, it is safe to assume that those stores are atomic. Therefore, when `is_bad` is read by a thread, the value read will either be zero or one. If a thread checks `is_bad` (line 108) before it has been assigned the value one, then it continues executing the `for` loop. If, in the meantime, another thread has assigned the value one to `is_bad` (line 112), that does not change the final result. It just means that the thread executed the `for` loop longer than necessary.

2.6.3 A Program Using Double-Checked Locking

A singleton ensures that only one object of a certain type exists throughout the program. Double-checked locking is a common, efficient way to initialize a singleton in multithreaded applications. The following code illustrates such an implementation.

```
100 class Singleton {
101     public:
102     static Singleton* instance();
103     ...
104     private:
105     static Singleton* ptr_instance;
106 };
107
108 Singleton* Singleton::ptr_instance = 0;
...

200 Singleton* Singleton::instance() {
201     Singleton *tmp;
202     if (ptr_instance == 0) {
203         Lock();
204         if (ptr_instance == 0) {
205             tmp = new Singleton;
206
207             /* Make sure that all writes used to construct new
208                Singleton have been completed. */
209             memory_barrier();
210
211             /* Update ptr_instance to point to new Singleton. */
212             ptr_instance = tmp;
213
214         }
215         Unlock();
216     }
217     return ptr_instance;
```

The read of `ptr_instance` on line 202 is intentionally not protected by a lock. This makes the check to determine whether or not the `Singleton` has already been instantiated in a multithreaded environment more efficient. Notice that there is a data race on variable `ptr_instance` between the read on line 202 and the write on line 212, but the program works correctly. However, writing a correct program that allows data races requires extra care. For example, in the above double-checked-locking code, the call to `memory_barrier()` at line 209 is used to ensure that `ptr_instance` is not seen to be non-null by the threads until all writes to construct the `Singleton` have been completed.

CHAPTER 3

The Deadlock Tutorial

This tutorial explains how to use the Thread Analyzer to detect potential deadlocks and actual deadlocks in your multithreaded program.

The tutorial covers the following topics:

3.1 About Deadlocks

The term *deadlock* describes a condition in which two or more threads are blocked forever because they are waiting for each other. There are many causes of deadlocks such as erroneous program logic and inappropriate use of synchronizations such as locks and barriers. This tutorial focuses on deadlocks that are caused by the inappropriate use of *mutexes*, or mutual exclusion locks. This type of deadlock is commonly encountered in multithreaded applications.

A process with two or more threads can enter deadlock when the following three conditions hold:

- Threads that are already holding locks request new locks
- The requests for new locks are made concurrently
- Two or more threads form a circular chain in which each thread waits for a lock which is held by the next thread in the chain

Here is a simple example of a deadlock condition:

– Thread 1 holds lock A and requests lock B
– Thread 2 holds lock B and requests lock A

A deadlock can be of two types: A *potential* deadlock or an *actual* deadlock and they are distinguished as follows:

- A potential deadlock does not necessarily occur in a given run, but can occur in any execution of the program depending on the scheduling of threads and the timing of lock requests by the threads.
- An actual deadlock is one that occurs during the execution of a program. An actual deadlock causes the threads involved to hang, but may or may not cause the whole process to hang.

3.2 Getting the Deadlock Tutorial Source Files

You can download the source files used in this tutorial from the samples download area (`http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/solaris-studio-samples-1408618.html`) of the Oracle Solaris Studio developer portal.

After you download and unpack the sample files, you can find the samples in the `SolarisStudioSampleApplications/ThreadAnalyzer` directory. The samples are located in the `din_philo` subdirectory. The `din_philo` directory includes a `Makefile` and a `DEMO` file of instructions, but this tutorial does not follow those instructions or use the `Makefile`. Instead, you are instructed to execute commands individually.

To follow the tutorial, you can copy the `din_philo.c` file from the `SolarisStudioSampleApplications/ThreadAnalyzer/din_philo` directory to a different directory, or you can create your own file and copy the code from the following code listing.

The `din_philo.c` sample program which simulates the dining-philosophers problem is a C program that uses POSIX threads. The program can exhibit both potential and actual deadlocks.

3.2.1 Source Code Listing for `din_philo.c`

The source code for `din_philo.c` is shown below:

```
/*
 * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All Rights Reserved.
 * @(#)din_philo.c 1.4 (Oracle) 10/03/26
 */

#include <pthread.h>
```

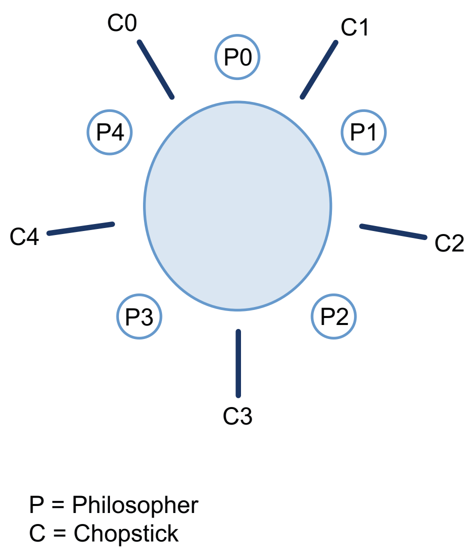
```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <assert.h>

#define PHILOS 5
#define DELAY 5000
#define FOOD 100

void *philosopher (void *id);
void grab_chopstick (int,
                     int,
                     char *);
void down_chopsticks (int,
                      int);
int food_on_table ();

pthread_mutex_t chopstick[PHILOS];
pthread_t philo[PHILOS];
pthread_mutex_t food_lock;
int sleep_seconds = 0;


int
main (int argn,
      char **argv)
{
    int i;

    if (argn == 2)
        sleep_seconds = atoi (argv[1]);

    pthread_mutex_init (&food_lock, NULL);
    for (i = 0; i < PHILOS; i++)
        pthread_mutex_init (&chopstick[i], NULL);
    for (i = 0; i < PHILOS; i++)
        pthread_create (&philo[i], NULL, philosopher, (void *)i);
    for (i = 0; i < PHILOS; i++)
        pthread_join (philo[i], NULL);
    return 0;
}

void *
philosopher (void *num)
{
    int id;
    int i, left_chopstick, right_chopstick, f;

    id = (int)num;
    printf ("Philosopher %d is done thinking and now ready to eat.\n", id);
    right_chopstick = id;
    left_chopstick = id + 1;

    /* Wrap around the chopsticks. */
    if (left_chopstick == PHILOS)
        left_chopstick = 0;

    while (f = food_on_table ()) {
```

```

        /* Thanks to philosophers #1 who would like to take a nap
         * before picking up the chopsticks, the other philosophers
         * may be able to eat their dishes and not deadlock.
         */
        if (id == 1)
            sleep (sleep_seconds);

        grab_chopstick (id, right_chopstick, "right ");
        grab_chopstick (id, left_chopstick, "left");

        printf ("Philosopher %d: eating.\n", id);
        usleep (DELAY * (FOOD - f + 1));
        down_chopsticks (left_chopstick, right_chopstick);
    }

    printf ("Philosopher %d is done eating.\n", id);
    return (NULL);
}

int
food_on_table ()
{
    static int food = FOOD;
    int myfood;

    pthread_mutex_lock (&food_lock);
    if (food > 0) {
        food--;
    }
    myfood = food;
    pthread_mutex_unlock (&food_lock);
    return myfood;
}

void
grab_chopstick (int phil,
                int c,
                char *hand)
{
    pthread_mutex_lock (&chopstick[c]);
    printf ("Philosopher %d: got %s chopstick %d\n", phil, hand, c);
}

void
down_chopsticks (int c1,
                 int c2)
{
    pthread_mutex_unlock (&chopstick[c1]);
    pthread_mutex_unlock (&chopstick[c2]);
}
```

3.3 The Dining Philosophers Scenario

The dining philosophers scenario is a classic which is structured as follows. Five philosophers, numbered zero to four, are sitting at a round table, thinking. As time passes, different individuals become hungry and decide to eat. There is a platter of noodles on the table but each philosopher only has one chopstick to use. In order to eat, they must share chopsticks. The chopstick to the right of each philosopher (as they sit facing the table) has the same number as that philosopher.

FIGURE 3–1 Dining Philosophers



Each philosopher first reaches for his own chopstick which is the one with his number. When he has his assigned chopstick, he reaches for the chopstick assigned to his neighbor. After he has both chopsticks, he can eat. After eating, he returns the chopsticks to their original positions on the table, one on either side. The process is repeated until there are no more noodles.

3.3.1 How the Philosophers Can Deadlock

An actual deadlock occurs when every philosopher is holding his own chopstick and waiting for the one from his neighbor to become available:

- Philosopher 0 is holding chopstick 0, but is waiting for chopstick 1
- Philosopher 1 is holding chopstick 1, but is waiting for chopstick 2
- Philosopher 2 is holding chopstick 2, but is waiting for chopstick 3
- Philosopher 3 is holding chopstick 3, but is waiting for chopstick 4

– Philosopher 4 is holding chopstick 4, but is waiting for chopstick 0

In this situation, nobody can eat and the philosophers are in a deadlock. Run the program a number of times. You will see that the program might hang sometimes, and run to completion at other times. The program might hang as shown in the following sample run:

```
prompt% cc din_philo.c
prompt% a.out
Philosopher 0 is done thinking and now ready to eat.
Philosopher 2 is done thinking and now ready to eat.
Philosopher 2: got right  chopstick 2
Philosopher 2: got left chopstick 3
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 4 is done thinking and now ready to eat.
Philosopher 4: got right  chopstick 4
Philosopher 2: eating.
Philosopher 3 is done thinking and now ready to eat.
Philosopher 1 is done thinking and now ready to eat.
Philosopher 0: got right  chopstick 0
Philosopher 3: got right  chopstick 3
Philosopher 2: got right  chopstick 2
Philosopher 1: got right  chopstick 1
(hang)
```

Execution terminated by pressing CTRL-C

3.3.2 Introducing a Sleep Time for Philosopher 1

One way to avoid deadlocks is for Philosopher 1 to wait before reaching for his chopstick. In terms of the code, he can be put to sleep for a specified amount of time (`sleep_seconds`) before reaching for his chopstick. If he sleeps long enough, then the program may finish without any actual deadlock. You can specify the number of seconds he sleeps as an argument to the executable. If you do not specify an argument, the philosopher does not sleep.

The following pseudo-code shows the logic for each philosopher:

```
   while (there is still food on the table)
      {
        if (sleep argument is specified and I am philosopher #1)
           {
             sleep specified amount of time
           }

        grab right fork
        grab left fork
        eat some food
        put down left fork
        put down right fork
      }
```

The following listing shows one run of the program in which Philosopher 1 waits 30 seconds before reaching for his chopstick. The program runs to completion and all five philosophers finish eating.

```
% a.out 30
Philosopher 0 is done thinking and now ready to eat.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 4 is done thinking and now ready to eat.
Philosopher 4: got right  chopstick 4
Philosopher 3 is done thinking and now ready to eat.
Philosopher 3: got right  chopstick 3
Philosopher 0: eating.
Philosopher 2 is done thinking and now ready to eat.
Philosopher 2: got right  chopstick 2
Philosopher 1 is done thinking and now ready to eat.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
...
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
```

```
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0 is done eating.
Philosopher 4: got left chopstick 0
Philosopher 4: eating.
Philosopher 4 is done eating.
Philosopher 3: got left chopstick 4
Philosopher 3: eating.
Philosopher 3 is done eating.
Philosopher 2: got left chopstick 3
Philosopher 2: eating.
Philosopher 2 is done eating.
Philosopher 1: got right  chopstick 1
Philosopher 1: got left chopstick 2
Philosopher 1: eating.
Philosopher 1 is done eating.
%
```

Execution terminated normally

Try running the program several times and specifying different sleep arguments. What happens when Philosopher 1 waits only a short time before reaching for his chopstick? How about when he waits longer? Try specifying different sleep arguments to the executable `a.out`. Rerun the program with or without a sleep argument several times. Sometimes the program hangs, while it runs to completion at other times. Whether the program hangs or not depends on the scheduling of threads and the timings of requests for locks by the threads.

3.4 How to Use the Thread Analyzer to Find Deadlocks

You can use the Thread Analyzer to check for potential and actual deadlocks in your program. The Thread Analyzer follows the same collect-analyze model that the Oracle Solaris Studio Performance Analyzer uses.

There are three steps involved in using the Thread Analyzer:

- Compile the source code.
- Create a deadlock-detection experiment.
- Examine the experiment results.

3.4.1 Compile the Source Code

Compile your code and be sure to specify `-g`. Do not specify a high-level of optimization because information such as line numbers and call stacks, may be reported incorrectly at a high optimization level. Compile an OpenMP program with `-g -xopenmp=noopt`, and compile a POSIX threads program with just `-g -mt`.

See `cc`(1), `CC`(1), or `f95`(1) man pages for more information about these options.

For this tutorial, compile the code using the following command:

```
% cc -g -o din_philo din_philo.c
```

3.4.2 Create a Deadlock Detection Experiment

Use the Thread Analyzer's `collect` command with the `-r deadlock` option. This option creates a deadlock-detection experiment during the execution of the program.

For this tutorial, create a deadlock detection experiment named `din_philo.1.er` using the following command:

```
% collect -r deadlock -o din_philo.1.er din_philo
```

You can increase the likelihood of detecting deadlocks by creating several deadlock-detection experiments. Use a different number of threads and different input data for the various experiments. For example, in the `din_philo.c` code, you could change the values in the following lines:

```
#define PHILOS 5
#define DELAY 5000
#define FOOD 100
```

You could then compile as before and collect another experiment.

See `collect`(1) and `collector`(1) man pages for more information.

3.4.3 Examine the Deadlock Detection Experiment

You can examine a deadlock detection experiment with the Thread Analyzer, the Performance Analyzer, or the `er_print` utility. Both the Thread Analyzer and the Performance Analyzer present a GUI interface; the Thread Analyzer presents a simplified set of default tabs, but is otherwise identical to the Performance Analyzer.

3.4.3.1 Using Thread Analyzer to View the Deadlock Detection Experiment

To start the Thread Analyzer and open the `din_philo.1.er` experiment , type the following command:

```
% tha din_philo.1.er
```

The Thread Analyzer includes a menu bar, a tool bar, and a split pane that contains tabs for the various displays.

The following tabs are shown by default in the left-hand pane when you open an experiment that was collected for deadlock detection:

- The Deadlocks tab

  This tab shows a list of potential and actual deadlocks that the Thread Analyzer detected in the program. This tab is selected by default. The threads involved for each deadlock are shown. These threads form a circular chain where each thread holds a lock and requests another lock that the next thread in the chain holds.
- The Dual Source tab

  Select a thread in the circular chain on the Deadlocks tab and then click on the Dual Source tab. The Dual Source tab shows the source location where the thread held a lock, and the source location where the same thread requested a lock. The source lines where the thread held and requested locks are highlighted.
- The Experiments tab

  This tab shows the load objects in the experiment, and lists any error and warning messages.

The following tabs are shown on the right-hand pane of the Thread Analyzer display:

- The Summary tab which shows summary information about a deadlock selected from the Deadlocks tab.
- The Deadlock Details tab which shows detailed information about a thread context selected from the Deadlocks tab.

3.4.3.2 Using `er_print` to View the Deadlock Detection Experiment

The `er_print` utility presents a command-line interface. You can use the `er_print` utility in an interactive session and specify sub-commands during the session. You can also use command-line options to specify sub-commands non-interactively.

The following sub-commands are useful for examining deadlocks with the `er_print` utility:

- `-deadlocks`

  This option reports any potential and actual deadlocks detected in the experiment. Specify `deadlocks` at the `(er_print)` prompt or `-deadlocks` on the `er_print` command line.
- `-ddetail` *deadlock_id*

This option returns detailed information about the deadlock with the specified *deadlock_id*. Specify `ddetail` at the `(er_print)` prompt or `-ddetail` on the `er_print` command line. If the specified *deadlock_id* is **`all`**, then detailed information about all deadlocks is displayed. Otherwise, specify a single deadlock number such as **`1`** for the first deadlock.

- `-header`

  This option displays descriptive information about the experiment and reports any errors or warnings. Specify `header` at the `(er_print)` prompt or `-header` on the command line.

Refer to the `collect`(1), `tha`(1), `analyzer`(1), and `er_print`(1) man pages for more information.

3.5 Understanding the Deadlock Experiment Results

This section explains how to use the Thread Analyzer to investigate the deadlocks in the dining philosopher program.

3.5.1 Examining Runs That Deadlock

The following listing shows a run of the dining philosophers program that results in an actual deadlock.

```
% cc -g -o din_philo din_philo.c
% collect -r deadlock -o din_philo.1.er din_philo
Creating experiment database din_philo.1.er ...
Philosopher 1 is done thinking and now ready to eat.
Philosopher 2 is done thinking and now ready to eat.
Philosopher 3 is done thinking and now ready to eat.
Philosopher 0 is done thinking and now ready to eat.
Philosopher 1: got right  chopstick 1
Philosopher 3: got right  chopstick 3
Philosopher 0: got right  chopstick 0
Philosopher 1: got left chopstick 2
Philosopher 3: got left chopstick 4
Philosopher 4 is done thinking and now ready to eat.
Philosopher 1: eating.
Philosopher 3: eating.
Philosopher 3: got right  chopstick 3
Philosopher 4: got right  chopstick 4
Philosopher 2: got right  chopstick 2
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 1: got right  chopstick 1
Philosopher 4: got left chopstick 0
Philosopher 4: eating.
Philosopher 0: got right  chopstick 0
Philosopher 3: got left chopstick 4
Philosopher 3: eating.
Philosopher 4: got right  chopstick 4
```

```
Philosopher 2: got left chopstick 3
Philosopher 2: eating.
Philosopher 3: got right  chopstick 3
Philosopher 1: got left chopstick 2
Philosopher 1: eating.
Philosopher 2: got right  chopstick 2
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 1: got right  chopstick 1
Philosopher 4: got left chopstick 0
Philosopher 4: eating.
Philosopher 0: got right  chopstick 0
Philosopher 3: got left chopstick 4
Philosopher 3: eating.
...
Philosopher 4: got right  chopstick 4
Philosopher 2: got left chopstick 3
Philosopher 2: eating.
Philosopher 2: got right  chopstick 2
Philosopher 3: got right  chopstick 3
(hang)
```

Execution terminated by pressing CTRL-C

Type the following commands to examine the experiment with `er_print` utility:

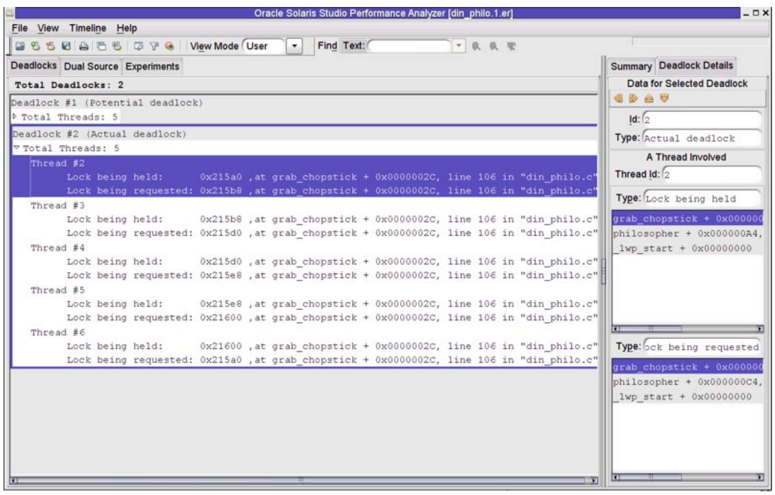
```
% er_print din_philo.1.er
(er_print) deadlocks
```

```
Deadlock #1, Potential deadlock
  Thread #2
    Lock being held:        0x215a0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested:   0x215b8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #3
    Lock being held:        0x215b8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested:   0x215d0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #4
    Lock being held:        0x215d0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested:   0x215e8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #5
    Lock being held:        0x215e8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested:   0x21600, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #6
    Lock being held:        0x21600, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested:   0x215a0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"

Deadlock #2, Actual deadlock
  Thread #2
    Lock being held:        0x215a0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested:   0x215b8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #3
    Lock being held:        0x215b8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested:   0x215d0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #4
    Lock being held:        0x215d0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested:   0x215e8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #5
    Lock being held:        0x215e8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
```

```
    Lock being requested:   0x21600, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #6
    Lock being held:        0x21600, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested:   0x215a0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"

Deadlocks List Summary: Experiment: din_philo.1.er Total Deadlocks: 2
(er_print)
```

The following screen shot shows the Thread Analyzer's presentation of the deadlock information.

FIGURE 3–2 Deadlock Detected in `din_philo.c`



The Thread Analyzer reports two deadlocks for `din_philo.c`, one potential and the other actual. On closer inspection, you find that the two deadlocks are identical.

The circular chain involved in the deadlock is as follows:

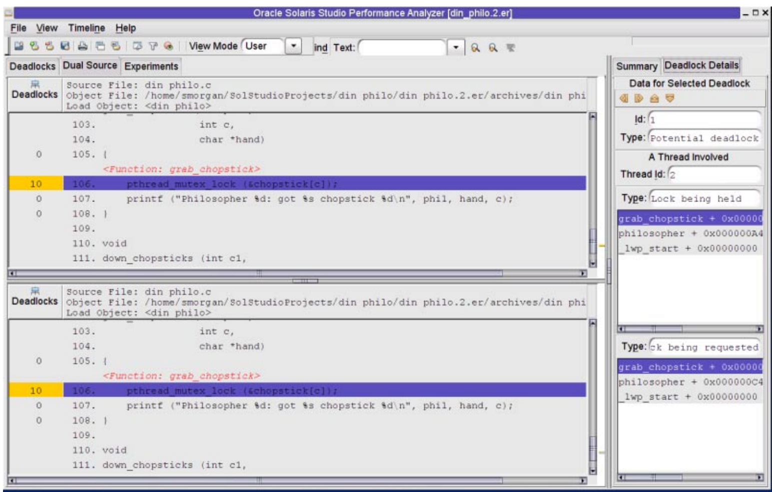
Thread 2: holds lock at address `0x215a0`, requests lock at address `0x215b8`
Thread 3: holds lock at address `0x215b8`, requests lock at address `0x215d0`
Thread 4: holds lock at address `0x215d0`, requests lock at address `0x215e8`
Thread 5: holds lock at address `0x215e8`, requests lock at address `0x21600`
Thread 6: holds lock at address `0x21600`, requests lock at address `0x215a0`

Select the first thread in the chain (Thread #2) and then click on the Dual Source tab to see where in the source code Thread #2 acquired the lock at address `0x215a0`, and where in the source code it requested the lock at address `0x215b8`.

The following screen shot shows the Dual Source tab for Thread #2. The top half of the screen shot shows that Thread #2 acquired the lock at address `0x215a0` by calling `pthread_mutex_lock()` on line 106. The bottom half of the screen shot shows that the same thread requested the lock at address `0x215b8` by calling `pthread_mutex_lock()` on line 106. Each of the two calls to `pthread_mutex_lock()` used a different lock as the argument. In general, the lock-acquire and lock-request operations may not be on the same source line.

The default metric (Exclusive Deadlocks metric) is shown to the left of each source line in the screen shot. This metric gives a count of the number of times a lock-acquire or lock-request operation, which was involved in a deadlock, was reported on that source line. Only source lines that are part of a deadlock chain would have a value for this metric that is larger than zero.

FIGURE 3–3 Potential Deadlock in `din_philo.c`



3.5.2 Examining Runs That Complete Despite Deadlock Potential

The dining philosophers program can avoid actual deadlock and terminate normally if you supply a large enough sleep argument. Normal termination, however, does not mean the program is safe from deadlocks. It simply means that the locks that were held and requested did not form a deadlock chain during a given run. If the timing changes in other runs, an actual deadlock can occur. The following listing shows a run of the dining philosophers program that terminates normally because of the 40 second sleep time. However, the `er_print` utility and the Thread Analyzer report potential deadlocks.

```
% cc -g -o din_philo_pt din_philo.c
% collect -r deadlock -o din_philo_pt.1.er din_philo_pt 40
```

```
Creating experiment database tha.2.er ...
Philosopher 0 is done thinking and now ready to eat.
Philosopher 2 is done thinking and now ready to eat.
Philosopher 1 is done thinking and now ready to eat.
Philosopher 3 is done thinking and now ready to eat.
Philosopher 2: got right  chopstick 2
Philosopher 3: got right  chopstick 3
Philosopher 0: got right  chopstick 0
Philosopher 4 is done thinking and now ready to eat.
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 3: got left chopstick 4
Philosopher 3: eating.
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 2: got left chopstick 3
Philosopher 2: eating.
...
Philosopher 4: got right  chopstick 4
Philosopher 3: got right  chopstick 3
Philosopher 2: got right  chopstick 2
Philosopher 4: got left chopstick 0
Philosopher 4: eating.
Philosopher 4 is done eating.
Philosopher 3: got left chopstick 4
Philosopher 3: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 3 is done eating.
Philosopher 2: got left chopstick 3
Philosopher 2: eating.
Philosopher 0 is done eating.
Philosopher 2 is done eating.
Philosopher 1: got right  chopstick 1
Philosopher 1: got left chopstick 2
Philosopher 1: eating.
Philosopher 1 is done eating.
%
```

Execution terminated normally

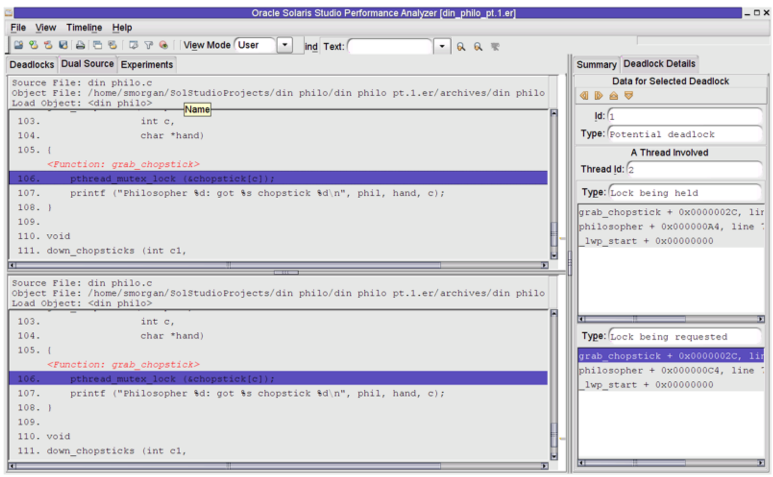
Type the following commands shown in bold to examine the experiment with `er_print` utility:

```
% er_print din_philo_pt.1.er
(er_print) deadlocks
Deadlock #1, Potential deadlock
  Thread #2
    Lock being held:      0x215a0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested: 0x215b8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #3
    Lock being held:      0x215b8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested: 0x215d0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #4
    Lock being held:      0x215d0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested: 0x215e8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #5
```

```
    Lock being held:      0x215e8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested: 0x21600, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #6
    Lock being held:      0x21600, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested: 0x215a0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"

Deadlocks List Summary: Experiment: din_philo_pt.1.er Total Deadlocks: 1
(er_print)
```

The following screen shot shows the potential deadlock information in the Thread Analyzer interface.

FIGURE 3–4 Potential deadlock in din_philo.c



3.6 Fixing the Deadlocks and Understanding False Positives

One way to remove potential and actual deadlocks is to use a system of tokens so that a philosopher must receive a token before attempting to eat. The number of available tokens must be less than the number of philosophers at the table. After a philosopher receives a token, he can attempt to eat in accordance with the rules of the table. After eating, each philosopher returns the token and repeats the process. The following pseudo-code shows the logic for each philosopher when using the token system.

```
while (there is still food on the table)
   {
     get token
     grab right fork
     grab left fork
     eat some food
     put down left fork
     put down right fork
```

```
  return token
}
```

The following sections detail two different implementations for the system of tokens.

3.6.1 Regulating the Philosophers With Tokens

The following listing shows the fixed version of the dining philosophers program that uses the token system. This solution incorporates four tokens, one less than the number of diners, so no more than four philosophers can attempt to eat at the same time. This version of the program is called `din_philo_fix1.c`:

Tip – If you downloaded the sample applications, you can copy the `din_philo_fix1.c` file from the `SolarisStudioSampleApplications/ThreadAnalyzer/din_philo` directory.

```
/*
 * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All Rights Reserved.
 * @(#)din_philo_fix1.c 1.3 (Oracle) 10/03/26
 */

#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <assert.h>

#define PHILOS 5
#define DELAY 5000
#define FOOD 100

void *philosopher (void *id);
void grab_chopstick (int,
                     int,
                     char *);
void down_chopsticks (int,
                      int);
int food_on_table ();
void get_token ();
void return_token ();

pthread_mutex_t chopstick[PHILOS];
pthread_t philo[PHILOS];
pthread_mutex_t food_lock;
pthread_mutex_t num_can_eat_lock;
int sleep_seconds = 0;
uint32_t num_can_eat = PHILOS - 1;


int
main (int argn,
      char **argv)
```

```
{
    int i;

    pthread_mutex_init (&food_lock, NULL);
    pthread_mutex_init (&num_can_eat_lock, NULL);
    for (i = 0; i < PHILOS; i++)
        pthread_mutex_init (&chopstick[i], NULL);
    for (i = 0; i < PHILOS; i++)
        pthread_create (&philo[i], NULL, philosopher, (void *)i);
    for (i = 0; i < PHILOS; i++)
        pthread_join (philo[i], NULL);
    return 0;
}

void *
philosopher (void *num)
{
    int id;
    int i, left_chopstick, right_chopstick, f;

    id = (int)num;
    printf ("Philosopher %d is done thinking and now ready to eat.\n", id);
    right_chopstick = id;
    left_chopstick = id + 1;

    /* Wrap around the chopsticks. */
    if (left_chopstick == PHILOS)
        left_chopstick = 0;

    while (f = food_on_table ()) {
        get_token ();

        grab_chopstick (id, right_chopstick, "right ");
        grab_chopstick (id, left_chopstick, "left");

        printf ("Philosopher %d: eating.\n", id);
        usleep (DELAY * (FOOD - f + 1));
        down_chopsticks (left_chopstick, right_chopstick);

        return_token ();
    }

    printf ("Philosopher %d is done eating.\n", id);
    return (NULL);
}

int
food_on_table ()
{
    static int food = FOOD;
    int myfood;

    pthread_mutex_lock (&food_lock);
    if (food > 0) {
        food--;
    }
    myfood = food;
    pthread_mutex_unlock (&food_lock);
    return myfood;
```

```
}

void
grab_chopstick (int phil,
                int c,
                char *hand)
{
    pthread_mutex_lock (&chopstick[c]);
    printf ("Philosopher %d: got %s chopstick %d\n", phil, hand, c);
}



void
down_chopsticks (int c1,
                 int c2)
{
    pthread_mutex_unlock (&chopstick[c1]);
    pthread_mutex_unlock (&chopstick[c2]);
}


void
get_token ()
{
    int successful = 0;

    while (!successful) {
        pthread_mutex_lock (&num_can_eat_lock);
        if (num_can_eat > 0) {
            num_can_eat--;
            successful = 1;
        }
        else {
            successful = 0;
        }
        pthread_mutex_unlock (&num_can_eat_lock);
    }
}

void
return_token ()
{
    pthread_mutex_lock (&num_can_eat_lock);
    num_can_eat++;
    pthread_mutex_unlock (&num_can_eat_lock);
}
```

Try compiling this fixed version of the dining philosophers program and running it several times. The system of tokens limits the number of diners attempting to use the chopsticks and thus avoids actual and potential deadlocks.

To compile, use the following command:

```
cc -g -o din_philo_fix1 din_philo_fix1.c
```

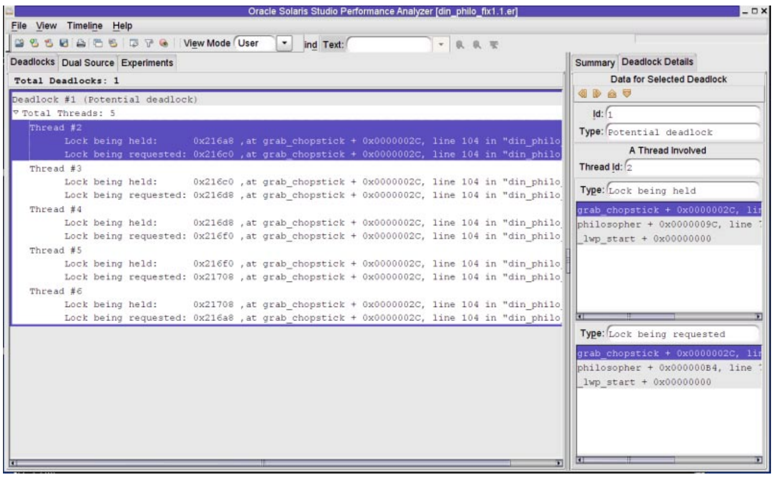
To collect an experiment:

```
collect -r deadlock din_philo_fix1 -o din_philo_fix1.1.er
```

3.6.1.1 A False Positive Report

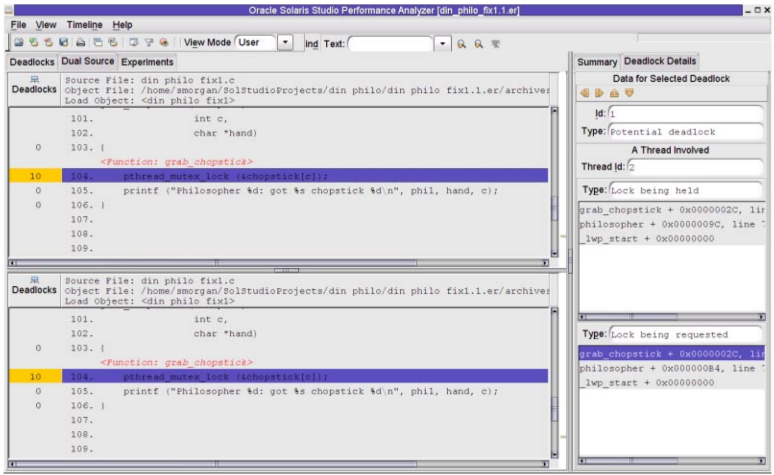
Even when using the system of tokens, the Thread Analyzer reports a potential deadlock for this implementation even though none exists. This is a false positive. Consider the following screen shot which details the potential deadlock.

FIGURE 3–5 False Positive Report of a Potential Deadlock



Select the first thread in the chain (Thread #2) and then click on the Dual Source tab to see the source code location in which Thread #2 held the lock at address `0x216a8`, and where in the source code it requested the lock at address `0x216c0`. The following figure shows the Dual Source tab for Thread #2.

FIGURE 3–6 False Positive Potential Deadlock's Source



The `get_token()` function in `din_philo_fix1.c` uses a `while` loop to synchronize the threads. A thread will not leave the `while` loop until it successfully gets a token (this occurs when `num_can_eat` is greater than zero). The `while` loop limits the number of simultaneous diners to four. However, the synchronization implemented by the `while` loop is not recognized by the Thread Analyzer. It assumes that all five philosophers attempt to grab the chopsticks and eat concurrently, so it reports a potential deadlock. The following section details how to limit the number of simultaneous diners by using synchronizations which the Thread Analyzer recognizes.

3.6.2 An Alternative System of Tokens

The following listing shows an alternative implementation of the system of tokens. This implementation still uses four tokens, so no more than four diners attempt to eat at the same time. However, this implementation uses the `sem_wait()` and `sem_post()` semaphore routines to limit the number of eating philosophers. This version of the source file is called `din_philo_fix2.c`.

Tip – If you downloaded the sample applications, you can copy the `din_philo_fix2.c` file from the `SolarisStudioSampleApplications/ThreadAnalyzer/din_philo` directory.

The following listing details `din_philo_fix2.c`:

```
/*
 * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All Rights Reserved.
 * @(#)din_philo_fix2.c 1.3 (Oracle) 10/03/26
 */
```

```

#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <assert.h>
#include <semaphore.h>

#define PHILOS 5
#define DELAY 5000
#define FOOD 100

void *philosopher (void *id);
void grab_chopstick (int,
                     int,
                     char *);
void down_chopsticks (int,
                      int);
int food_on_table ();
void get_token ();
void return_token ();

pthread_mutex_t chopstick[PHILOS];
pthread_t philo[PHILOS];
pthread_mutex_t food_lock;
int sleep_seconds = 0;
sem_t num_can_eat_sem;


int
main (int argn,
      char **argv)
{
    int i;

    pthread_mutex_init (&food_lock, NULL);
    sem_init(&num_can_eat_sem, 0, PHILOS - 1);
    for (i = 0; i < PHILOS; i++)
        pthread_mutex_init (&chopstick[i], NULL);
    for (i = 0; i < PHILOS; i++)
        pthread_create (&philo[i], NULL, philosopher, (void *)i);
    for (i = 0; i < PHILOS; i++)
        pthread_join (philo[i], NULL);
    return 0;
}

void *
philosopher (void *num)
{
    int id;
    int i, left_chopstick, right_chopstick, f;

    id = (int)num;
    printf ("Philosopher %d is done thinking and now ready to eat.\n", id);
    right_chopstick = id;
    left_chopstick = id + 1;

    /* Wrap around the chopsticks. */
```

```
    if (left_chopstick == PHILOS)
        left_chopstick = 0;

    while (f = food_on_table ()) {
        get_token ();

        grab_chopstick (id, right_chopstick, "right ");
        grab_chopstick (id, left_chopstick, "left");

        printf ("Philosopher %d: eating.\n", id);
        usleep (DELAY * (FOOD - f + 1));
        down_chopsticks (left_chopstick, right_chopstick);

        return_token ();
    }

    printf ("Philosopher %d is done eating.\n", id);
    return (NULL);
}

int
food_on_table ()
{
    static int food = FOOD;
    int myfood;

    pthread_mutex_lock (&food_lock);
    if (food > 0) {
        food--;
    }
    myfood = food;
    pthread_mutex_unlock (&food_lock);
    return myfood;
}

void
grab_chopstick (int phil,
                int c,
                char *hand)
{
    pthread_mutex_lock (&chopstick[c]);
    printf ("Philosopher %d: got %s chopstick %d\n", phil, hand, c);
}

void
down_chopsticks (int c1,
                 int c2)
{
    pthread_mutex_unlock (&chopstick[c1]);
    pthread_mutex_unlock (&chopstick[c2]);
}


void
get_token ()
{
    sem_wait(&num_can_eat_sem);
}

```

```
void
return_token ()
{
    sem_post(&num_can_eat_sem);
}
```

This new implementation uses the semaphore `num_can_eat_sem` to limit the number of philosophers who can eat at the same time. The semaphore `num_can_eat_sem` is initialized to four, one less than the number of philosophers. Before attempting to eat, a philosopher calls `get_token()` which in turn calls `sem_wait(&num_can_eat_sem)`. The call to `sem_wait()` causes the calling philosopher to wait until the semaphore's value is positive, then changes the semaphore's value by subtracting one from the value. When a philosopher is done eating, he calls `return_token()` which in turn calls `sem_post(&num_can_eat_sem)`. The call to `sem_post()` changes the semaphore's value by adding one. The Thread Analyzer recognizes the calls to `sem_wait()` and `sem_post()`, and determines that not all philosophers attempt to eat concurrently.

Note – You must compile `din_philo_fix2.c` with `-lrt` to link with the appropriate semaphore routines.

To compile `din_philo_fix2.c`, use the following command:

```
cc -g -lrt -o din_philo_fix2 din_philo_fix2.c
```

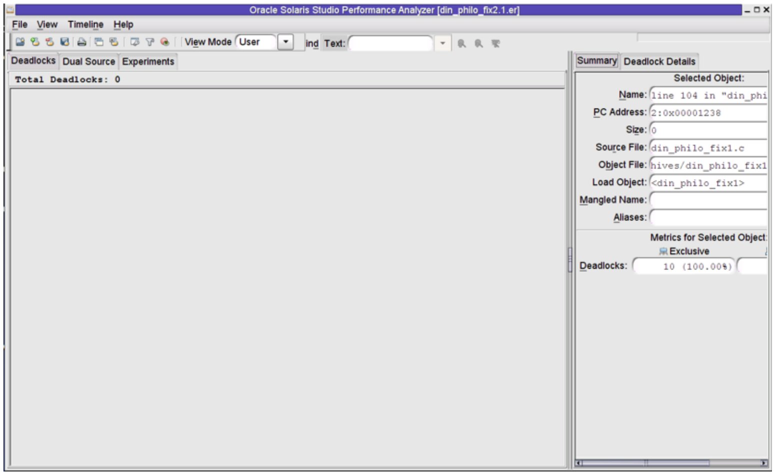
If you run this new implementation of the program `din_philo_fix2` several times, you will find that it terminates normally each time and does not hang.

To create an experiment on this new binary:

```
collect -r deadlock -o din_philo_fix2.1.er din_philo_fix2
```

You will find that the Thread Analyzer does not report any actual or potential deadlocks in the `din_philo_fix2.1.er` experiment, as the following figure shows.

FIGURE 3–7 Deadlocks Not Reported in `din_philo_fix2.c`



See Appendix A, "APIs Recognized by the Thread Analyzer," for a listing of the threading and memory allocation APIs that the Thread Analyzer recognizes.

APPENDIX A

APIs Recognized by the Thread Analyzer

The Thread Analyzer can recognize most standard synchronization APIs and constructs provided by OpenMP directives, POSIX threads, and Solaris threads. However, the tool cannot recognize user-defined synchronizations, and may report false positive data races if you employ such synchronizations. For example, the tool cannot recognize spin locking that is implemented through hand-coded assembly-language code.

A.1 Thread Analyzer User APIs

If your code includes user-defined synchronizations, insert user APIs supported by the Thread Analyzer into the program to identify those synchronizations. This identification allows the Thread Analyzer to recognize the synchronizations and reduce the number of false positives. The Thread Analyzer user APIs are defined in `libtha.so` and are listed below.

TABLE A–1 Thread Analyzer User APIs

`tha_notify_acquire_lock()`	This routine can be called immediately before the program tries to acquire a user-defined lock.
`tha_notify_lock_acquired()`	This routine can be called immediately after a user-defined lock is successfully acquired.
`tha_notify_acquire_writelock()`	This routine can be called immediately before the program tries to acquire a user-defined read/write lock in write mode.
`tha_notify_writelock_acquired()`	This routine can be called immediately after a user-defined read/write lock is successfully acquired in write mode.
`tha_notify_acquire_readlock()`	This routine can be called immediately before the program tries to acquire a user-defined read/write lock in read mode.
`tha_notify_readlock_acquired()`	This routine can be called immediately after a user-defined read/write lock is successfully acquired in read mode.

TABLE A–1 Thread Analyzer User APIs *(Continued)*

`tha_notify_release_lock()`	This routine can be called immediately before a user-defined lock or read/write lock is to be released.
`tha_notify_lock_released()`	This routine can be called immediately after a user-defined lock or read/write lock is successfully released.
`tha_notify_sync_post_begin()`	This routine can be called immediately before a user-defined post synchronization is performed.
`tha_notify_sync_post_end()`	This routine can be called immediately after a user-defined post synchronization is performed.
`tha_notify_sync_wait_begin()`	This routine can be called immediately before a user-defined wait synchronization is performed.
`tha_notify_sync_wait_end()`	This routine can be called immediately after a user-defined wait synchronization is performed.
`tha_check_datarace_mem()`	This routine instructs the Thread Analyzer to monitor or ignore accesses to a specified block of memory when doing data race detection.
`tha_check_datarace_thr()`	This routine instructs the Thread Analyzer to monitor or ignore memory accesses by one or more threads when doing data race detection.

A C/C++ version and a Fortran version of the APIs are provided. Each API call takes a single argument id, whose value should uniquely identify the synchronization object.

In the C/C++ version of the APIs, the type of the argument is `uintptr_t`, which is 4 bytes long in 32-bit mode and 8 bytes long in 64-bit mode. You need to add `#include <tha_interface.h>` to your C/C++ source file when calling any of the APIs.

In the Fortran version of the APIs, the type of the argument is integer of kind `tha_sobj_kind` which is 8-bytes long in both 32-bit and 64–bit mode. You need to add `#include "tha_finterface.h"` to your Fortran source file when calling any of the APIs.

To uniquely identify a synchronization object, the argument id should have a different value for each different synchronization object. One way to do this is to use the value of the address of the synchronization object as the id. The following code example shows how to use the API to avoid a false positive data race.

EXAMPLE A–1 Example Using Thread Analyzer APIs to Avoid False Positive Data Races

```
# include <tha_interface.h>
...
/* Initially, the ready_flag value is zero */
...
/* Thread 1: Producer */
100   data = ...
```

EXAMPLE A–1 Example Using Thread Analyzer APIs to Avoid False Positive Data Races *(Continued)*

```
      pthread_mutex_lock (&mutex);
      tha_notify_sync_post_begin ((uintptr_t) &ready_flag);
      ready_flag = 1;
      tha_notify_sync_post_end ((uintptr_t) &ready_flag);

      pthread_cond_signal (&cond);
      pthread_mutex_unlock (&mutex);


/* Thread 2: Consumer */
      pthread_mutex_lock (&mutex);
      tha_notify_sync_wait_begin ((uintptr_t) &ready_flag);
      while (!ready_flag) {
          pthread_cond_wait (&cond, &mutex);
      }
      tha_notify_sync_wait_end ((uintptr_t) &ready_flag);
      pthread_mutex_unlock (&mutex);
      ... = data;
```

For more information on the user APIs, see the `libtha`(3) man page.

A.2 Other Recognized APIs

The following sections detail the threading APIs which the Thread Analyzer recognizes.

A.2.1 POSIX Thread APIs

See the *Multithreaded Programming Guide* in the Oracle Solaris documentation for more information about these APIs.

```
pthread_mutex_lock()
pthread_mutex_trylock()
pthread_mutex_unlock()
pthread_rwlock_rdlock()
pthread_rwlock_tryrdlock()
pthread_rwlock_wrlock()
pthread_rwlock_trywrlock()
pthread_rwlock_unlock()
pthread_create()
pthread_join()
pthread_cond_signal()
pthread_cond_broadcast()
pthread_cond_wait()
```

```
pthread_cond_timedwait()
pthread_cond_reltimedwait_np()
pthread_barrier_init()
pthread_barrier_wait()
pthread_spin_lock()
pthread_spin_unlock()
pthread_spin_trylock()
pthread_mutex_timedlock()
pthread_mutex_reltimedlock_np()
pthread_rwlock_timedrdlock()
pthread_rwlock_reltimedrdlock_np()
pthread_rwlock_timedwrlock()
pthread_rwlock_reltimedwrlock_np()
sem_post()
sem_wait()
sem_trywait()
sem_timedwait()
sem_reltimedwait_np()
```

A.2.2 Solaris Thread APIs

See the *Multithreaded Programming Guide* in the Oracle Solaris documentation for more information about these APIs.

```
mutex_lock()
mutex_trylock()
mutex_unlock()
rw_rdlock()
rw_tryrdlock()
rw_wrlock()
rw_trywrlock()
rw_unlock()
thr_create()
thr_join()
cond_signal()
cond_broadcast()
cond_wait()
cond_timedwait()
cond_reltimedwait()
sema_post()
sema_wait()
sema_trywait()
```

A.2.3 Memory-Allocation APIs

```
calloc()
malloc()
realloc()
valloc()
memalign()
```

See the `malloc`(3C) man page for information about the memory allocation APIs.

A.2.4 Memory Operations APIs

```
memcpy()
memmove()
memchr()
memcmp()
memset()
```

See the `memcpy`(3C) man page for information about the memory operations APIs.

A.2.5 String Operations APIs

```
strcat()
strncat()
strlcat()
strcasecmp()
strncasecmp()
strchr()
strrchr()
strcmp()
strncmp()
strcpy()
strncpy()
strlcpy()
strcspn()
strspn()
strdup()
strlen()
strpbrk()
strstr()
strtok()
```

See the `strcat`(3C) man page for information about the string operations APIs.

A.2.6 OpenMP APIs

The Thread Analyzer recognizes OpenMP synchronizations, such as barriers, locks, critical regions, atomic regions, and taskwait.

See the *Oracle Solaris Studio 12.3: OpenMP API User's Guide* for more information.

APPENDIX B

Useful Tips

This appendix includes some tips for using the Thread Analyzer.

B.1 Compiling the Application

Tips for compiling an application before collecting an experiment:

- Use the `-g` compiler option when building application binaries. This will allow the Thread Analyzer to report line number information for data races.
- Compile with an optimization level less than `-xO3` when building application binaries. Compiler transformations may distort line number information and make the results difficult to understand.
- The Thread Analyzer interposes on the memory-allocation routines shown in "A.2.3 Memory-Allocation APIs" on page 71. Linking to archive versions of memory allocation libraries may result in false positive data races being reported.

B.2 Instrumenting the Application for Data Race Detection

Tips for instrumenting an application for data race detection before collecting an experiment:

- If you get an error message from the compiler saying that the compiler option `-xinstrument=datarace` is illegal, you are using an older version of the Sun Studio compiler that does not support the Thread Analyzer. You can check the version of the compiler you are using by typing the command `cc -Version`. The earliest version that supports Thread Analyzer is dated June 2006.
- The `collect -r race` command issues a warning if the binary is not instrumented for data race detection, as shown here:

```
% collect -r races a.out
  WARNING: Target 'a.out' is not instrumented for datarace
  detection; reported datarace data may be misleading
```

- You can determine whether a binary is instrumented for data race detection by using the `nm` command and looking for calls to `tha` routines. If routines whose names begin with `__tha_` are shown, the binary is instrumented. Example output is shown below.

  Source-level instrumentation:

  ```
  % cc -xopenmp -g -xinstrument=datarace source.c
  % nm a.out | grep __tha_
    [71] | 135408| 0|FUNC |GLOB |0 |UNDEF |__tha_get_stack_id
    [53] | 135468| 0|FUNC |GLOB |0 |UNDEF |__tha_src_read_w_frame
    [61] | 135444| 0|FUNC |GLOB |0 |UNDEF |__tha_src_write_w_frame
  ```

  Binary-level instrumentation:

  ```
  % cc -xopenmp -g source.c
  % discover -i datarace -o a.out.i a.out
  % nm a.out.i | grep __tha_
    [88] | 0| 0|NOTY |GLOB |0 |UNDEF |__tha_read_w_pc_frame
    [49] | 0| 0|NOTY |GLOB |0 |UNDEF |__tha_write_w_pc_frame
  ```

- To use `discover`, the input binary must be compiled with one of the compiler optimization flags (`-xO1, -xO2, -xO3, -xO4, -xO5`). The compiler must be from a release no earlier than Oracle Solaris Studio 12 Update 1. The operating system must be at least Oracle Solaris 10 Update 5 or Oracle Solaris 11. Otherwise, you may get a warning as shown here.

  ```
  % discover -i datarace -o a.out.i a.out
    discover (warning): a.out has no annotations. Results may be
    incomplete. See discover documentation for compiler flag/OS
    recommendation
  ```

  You might also be able to use the `discover` tool on an earlier Solaris version running on a SPARC-based system if the binary was compiled with the compiler option `-xbinopt=prepare`. See the `cc`(1), `CC`(1), or `f95`(1) man pages for information about this compiler option.

B.3 Running the Application With collect

Tips for running an instrumented application to detect data races and deadlocks.

- Make sure that the Oracle Solaris system has all the required patches installed. The `collect` command lists any missing required patches. For OpenMP applications, the latest version of `libmtsk.so` is required.
- If you get an error message from `collect` saying that the `-r race` or `-r deadlock` argument is not recognized, you are using an older version of `collect` that does not support the Thread Analyzer. You can check the version of `collect` you are using by typing the `collect -Version` command. The earliest version that supports Thread Analyzer is dated June 2006.
- Instrumentation might cause a significant slowdown in execution time, 50 times or more, and an increase in memory consumption. You can try reducing the execution time by using a smaller data set. You can also try reducing the execution time by increasing the number of threads.

- To detect data races, make sure that the application is using more than one thread. For OpenMP, the number of threads can be specified by setting the environment variable `OMP_NUM_THREADS` to the desired number of threads, and setting the environment variable `OMP_DYNAMIC` to `FALSE`.

B.4 Reporting of Data Races

Tips for reporting of data races:

- The Thread Analyzer detects data races at runtime. The runtime behavior of an application depends on the input data set used and operating system scheduling. Run the application under `collect` with different numbers of threads and with different input data sets. Also repeat experiments with a single data set to maximize the tool's chance of detecting data races.
- The Thread Analyzer detects data races between different threads that are spawned from a single process. It does not detect data races between different processes.
- The Thread Analyzer does not report the name of the variable accessed in a data race. However, you can determine the name of the variable by inspecting the source lines where the two data race accesses occurred, and determining which variables are written to and read from on those source lines.
- In some cases, the Thread Analyzer might report data races that did not actually occur in the program. These data races are called false positives. This usually happens when a user-implemented synchronization is used or when memory is recycled between threads. For example, if your code includes hand-coded assembly that implements spin locks, the Thread Analyzer will not recognize these synchronization points. Insert calls to Thread Analyzer user APIs in your source code to notify the Thread Analyzer about user-defined synchronizations. See “2.5 False Positives” on page 35 and Appendix A, “APIs Recognized by the Thread Analyzer,” for more information.
- Data races reported using source-level instrumentation and binary-level instrumentation might not be identical. In binary-level instrumentation, shared libraries are instrumented by default as they are opened, whether they are statically linked in the program or opened dynamically by `dlopen()`. In source-level instrumentation, libraries are instrumented only if their sources are compiled with `-xinstrument=datarace`.