Sun Studio 12: Thread Analyzer User's Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 820–0619

Copyright 2007 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more U.S. patents or pending patent applications in the U.S. and in other countries.

U.S. Government Rights – Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

This distribution may include materials developed by third parties.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, the Solaris logo, the Java Coffee Cup logo, docs.sun.com, Java, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

Products covered by and information contained in this publication are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical or biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

DOCUMENTATION IS PROVIDED “AS IS” AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 2007 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. Tous droits réservés.

Sun Microsystems, Inc. détient les droits de propriété intellectuelle relatifs à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et ce sans limitation, ces droits de propriété intellectuelle peuvent inclure un ou plusieurs brevets américains ou des applications de brevet en attente aux Etats-Unis et dans d'autres pays.

Cette distribution peut comprendre des composants développés par des tierces personnes.

Certaines composants de ce produit peuvent être dérivées du logiciel Berkeley BSD, licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays; elle est licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, le logo Solaris, le logo Java Coffee Cup, docs.sun.com, Java et Solaris sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui, en outre, se conforment aux licences écrites de Sun.

Les produits qui font l'objet de cette publication et les informations qu'il contient sont régis par la legislation américaine en matière de contrôle des exportations et peuvent être soumis au droit d'autres pays dans le domaine des exportations et importations. Les utilisations finales, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes chimiques ou biologiques ou pour le nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou réexportations vers des pays sous embargo des Etats-Unis, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière non exclusive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations des produits ou des services qui sont régis par la legislation américaine en matière de contrôle des exportations et la liste de ressortissants spécifiquement designés, sont rigoureusement interdites.

LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFACON.

070518@17466

Contents

Preface 5

1 **What is the Thread Analyzer and What Does It Do?** 11

1.1 Getting Started With the Thread Analyzer 11

1.2 What is a Data Race? 11

1.3 What is a Deadlock? 12

1.4 The Thread Analyzer Usage Model 12

2 **The Data-Race Tutorial** 15

2.1 Tutorial Source Files 15

2.1.1 Complete Listing of `omp_prime.c` 15

2.1.2 Complete Listing of `pthr_prime.c` 17

2.2 Creating Experiments 21

2.2.1 Instrument the Source Code 21

2.2.2 Create a Data-Race Detection Experiment 22

2.2.3 Examine the Data-Race Detection Experiment 22

2.3 Understanding the Experiment Results 23

2.3.1 Data Races in `omp_prime.c` 23

2.3.2 Data Races in `pthr_prime.c` 26

2.4 Diagnosing the Cause of a Data Race 29

2.4.1 Check Whether or Not the Data Race is a False Positive 29

2.4.2 Check Wether or Not the Data Race is Benign 30

2.4.3 Fix the Bug, Not the Data Race 30

2.5 False Positives 33

2.5.1 User-Defined Synchronizations 34

2.5.2 Memory That is Recycled by Different Threads 35

2.6 Benign Data-Races 36

2.6.1 A Program for Finding Primes 36
2.6.2 A Program that Verifies Array-Value Types 37
2.6.3 A Program Using Double-Checked Locking 38

3 The Deadlock Tutorial 39
3.1 The Dining Philosophers Source File 40
3.2 The Dining Philosophers Scenario 42
3.2.1 How the Philosophers Can Deadlock 43
3.2.2 Introducing a Sleep Time for Philosopher One 44
3.3 How to Use the Thread Analyzer to Find Deadlocks 46
3.3.1 Compile the Source Code 47
3.3.2 Create a Deadlock-Detection Experiment 47
3.3.3 Examine the Experiment Results 47
3.4 Understanding the Experiment Results 48
3.4.1 Examining Runs That Deadlock 48
3.4.2 Examining Runs That Complete Despite Deadlock Potential 52
3.5 Fixing the Deadlocks and Understanding False-Positives 55
3.5.1 Regulating the Philosophers With Tokens 56
3.5.2 An Alternative System of Tokens 60

A Thread Analyzer User API 65
A.1 The Thread-Analyzer's User-APIs 65
A.2 Other Recognized APIs 67
A.2.1 POSIX Thread APIs 67
A.2.2 Solaris Thread APIs 68
A.2.3 Memory-Allocation APIs 68
A.2.4 OpenMP APIs 68

B Thread Analyzer Frequently Asked Questions 69
B.1 FAQ 69

Index 73

Preface

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

Typographic Conventions

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

TABLE P–1 Typographic Conventions

Typeface	Meaning	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 the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

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

Supported Platforms

This Sun Studio release supports systems that use the SPARC® and x86 families of processor architectures: UltraSPARC®, SPARC64, AMD64, Pentium, and Xeon EM64T. The supported systems for the version of the Solaris Operating System you are running are available in the hardware compatibility lists at `http://www.sun.com/bigadmin/hcl`. These documents cite any implementation differences between the platform types.

In this document, these x86 related terms mean the following:

- "x86" refers to the larger family of 64-bit and 32-bit x86 compatible products.
- "x64' points out specific 64-bit information about AMD64 or EM64T systems.
- "32-bit x86" points out specific 32-bit information about x86 based systems.

For supported systems, see the hardware compatibility lists.

Accessing Sun Studio Documentation

You can access the documentation at the following locations:

- The documentation is available from the documentation index that is installed with the software on your local system or network at `file:/opt/SUNWspro/docs/index.html` on Solaris platforms and at `file:/opt/sun/sunstudio12/docs/index.html` on Linux platforms.

  If your software is not installed in the `/opt` directory on a Solaris platform or the `/opt/sun` directory on a Linux platform, ask your system administrator for the equivalent path on your system.
- Most manuals are available from the `docs.sun.com`sm web site. The following titles are available through your installed software on Solaris platforms only:

- *Standard C++ Library Class Reference*
- *Standard C++ Library User's Guide*
- *Tools.h++ Class Library Reference*
- *Tools.h++ User's Guide*

The release notes are available from the http://docs.sun.com web site.

- Online help for all components of the IDE is available through the Help menu, as well as through Help buttons on many windows and dialog boxes, in the IDE.

The http://docs.sun.com web site enables you to read, print, and buy Sun Microsystems manuals through the Internet. If you cannot find a manual, see the documentation index that is installed with the software on your local system or network.

Note – Sun is not responsible for the availability of third-party Web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Documentation in Accessible Formats

The documentation is provided in accessible formats that are readable by assistive technologies for users with disabilities. You can find accessible versions of documentation as described in the following table. If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Type of Documentation	Format and Location of Accessible Version
Manuals (except third-party manuals)	HTML at http://docs.sun.com
Third-party manuals: ■ *Standard C++ Library Class Reference* ■ *Standard C++ Library User's Guide* ■ *Tools.h++ Class Library Reference* ■ *Tools.h++ User's Guide*	HTML in the installed software on Solaris platforms through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Readmes	HTML on the Sun Developer Network portal at http://developers.sun.com/sunstudio/documentation/ss12

Man pages	HTML in the installed software through the documentation index at file:/opt/SUNWspro/docs/index.html on Solaris platforms, and at file:/opt/sun/sunstudio12/docs/index.html on Linux platforms,
Online help	HTML available through the Help menu and Help buttons in the IDE
Release notes	HTML at http://docs.sun.com

Related Sun Studio Documentation

The following table describes related documentation that is available at file:/opt/SUNWspro/docs/index.html and http://docs.sun.com. If your software is not installed in the /opt directory, ask your system administrator for the equivalent path on your system.

Document Title	Description
Performance Analyzer	Provides instructions for using the Performance Analyzer software to diagnose and tune software.
C User's Guide	Provides a reference of all compiler options, descriptions of supported ISO/IEC 9899:1999 (referred to as C99) features, implementation specifics such as pragmas and declaration specifiers, and complete information for using the lint code-checking program.
C++ User's Guide	Describes how to use the C++ compiler and provides detailed information on command-line compiler options, program organization, pragmas, templates, exception handing, using the cast operators, and using and building libraries.
Fortran Programming Guide	Describes how to write effective Fortran programs on Solaris environments; input/output, libraries, performance, debugging, and parallelization.
Fortran Library Reference	Details the Fortran library and intrinsics.
OpenMP API User's Guide	Summary of the OpenMP multiprocessing API, with specifics about the implementation.
Numerical Computation Guide	Describes issues regarding the numerical accuracy of floating-point computations.

Accessing Related Solaris Documentation

The following table describes related documentation that is available through the `docs.sun.com` web site.

Document Collection	Document Title	Description
Solaris Reference Manual Collection	See the titles of man page sections.	Provides information about the Solaris OS.
Solaris Software Developer Collection	*Linker and Libraries Guide*	Describes the operations of the Solaris link-editor and runtime linker.
Solaris Software Developer Collection	*Multithreaded Programming Guide*	Covers the POSIX and Solaris threads APIs, programming with synchronization objects, compiling multithreaded programs, and finding tools for multithreaded programs.

Resources for Developers

Visit the Sun Developer Network Sun Studio portal at `http://developers.sun.com/sunstudio` to find these frequently updated resources:

- Articles on programming techniques and best practices
- A knowledge base of short programming tips
- Documentation of the software, as well as corrections to the documentation that is installed with your software
- Information on support levels
- User forums
- Downloadable code samples
- New technology previews

The Sun Studio portal is one of a number of additional resources for developers at the Sun Developer Network website, `http://developers.sun.com`.

Contacting Sun Technical Support

If you have technical questions about this product that are not answered in this document, go to:

`http://www.sun.com/service/contacting`

Sending Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. Submit your comments to Sun at this URL:

`http://www.sun.com/hwdocs/feedback`

Please include the part number of the document in the subject line of your email. For example, the part number for this document is 820-0619.

CHAPTER 1

What is the Thread Analyzer and What Does It Do?

The Thread Analyzer is a tool that you can use to analyze the execution of a multi-threaded program. It can detect multi-threaded programming errors such as data races or deadlocks in code that is written using the POSIX thread API, the Solaris Operating System(R) thread API, OpenMP directives, Sun parallel directives, Cray(R) parallel directives, or a mix of these.

1.1 Getting Started With the Thread Analyzer

You can start the Thread Analyzer by using the new `tha` command. The Thread Analyzer interface is streamlined for multi-threaded program analysis so it does not display the traditional Analyzer tabs. Instead, you see the new Races, Deadlocks, Dual Source, Race Details, and Deadlock Details tabs. If you use the Analyzer to look at the same multi-threaded program experiments you will see the traditional Analyzer tabs such as Functions, Callers-Callees, Disassembly, along with the new tabs.

The Thread Analyzer supports the following hardware and operating systems:

- The SPARC(R) v8plus, v8plusa, v8plusb, v9, v9a, and v9b architectures
- The Intel(R) x86 and AMD(R) x64 platforms
- The Solaris 9 and Solaris 10 operating systems
- SuSE Linux Enterprise Server 9, and Red Hat Enterprise Linux 4 operating systems

1.2 What is a Data Race?

The Thread Analyzer detects data-races that occur during the execution of a multi-threaded process. A data race occurs when:

- two or more threads in a *single process* access the same memory location concurrently, and
- at least one of the accesses is for writing, and

- 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 multi-threaded program written using the POSIX thread API, Solaris thread API, OpenMP, Sun parallel directives, Cray parallel directives, or a mix of the above.

1.3 What is a Deadlock?

Deadlock describes a condition in which two or more threads are blocked (hung) 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 multi-threaded applications. A process with two or more threads can deadlock when the following 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. 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.

1.4 The Thread Analyzer Usage Model

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

1. Instrument the program. See “2.2.1 Instrument the Source Code” on page 21 for more information.

2. Perform an experiment and then repeat the experiment with varied factors such as different input data, a different number of threads, varied loop schedules or even different hardware. This repetition helps locate problems with non-deterministic roots.
3. Establish whether or not the multi-threaded programming-conflicts revealed by the Thread Analyzer are legitimate bugs or benign phenomenon.
4. Fix the legitimate bugs and repeat the experiment.
5. If the Thread Analyzer reports new multi-threaded programming-conflicts repeat the previous two steps.

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 Tutorial Source Files" on page 15
- "2.2 Creating Experiments" on page 21
- "2.3 Understanding the Experiment Results" on page 23
- "2.4 Diagnosing the Cause of a Data Race" on page 29
- "2.5 False Positives" on page 33
- "2.6 Benign Data-Races" on page 36

2.1 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 `omp_prime.c`.
- The second program also finds prime number and is also written with C. However, it is parallelized with POSIX threads instead of OpenMP directives. The source file is called `pthr_prime.c`.

2.1.1 Complete Listing of `omp_prime.c`

```
1   #include <stdio.h>
2   #include <math.h>
3   #include <omp.h>
4
5   #define THREADS 4
6   #define N 3000
7
8   int primes[N];
```

```
9   int pflag[N];
10
11  int is_prime(int v)
12  {
13      int i;
14      int bound = floor(sqrt ((double)v)) + 1;
15
16      for (i = 2; i < bound; i++) {
17          /* No need to check against known composites */
18          if (!pflag[i])
19              continue;
20          if (v % i == 0) {
21              pflag[v] = 0;
22              return 0;
23          }
24      }
25      return (v > 1);
26  }
27
28  int main(int argn, char **argv)
29  {
30      int i;
31      int total = 0;
32
33  #ifdef _OPENMP
34      omp_set_num_threads(THREADS);
35      omp_set_dynamic(0);
36  #endif
37
38      for (i = 0; i < N; i++) {
39          pflag[i] = 1;
40      }
41
42      #pragma omp parallel for
43      for (i = 2; i < N; i++) {
44          if ( is_prime(i) ) {
45              primes[total] = i;
46              total++;
47          }
48      }
49      printf("Number of prime numbers between 2 and %d: %d\n",
50             N, total);
51      for (i = 0; i < total; i++) {
52          printf("%d\n", primes[i]);
53      }
54
55  return 0;
56  }
```

2.1.2 Complete Listing of `pthr_prime.c`

```
1   #include <stdio.h>
2   #include <math.h>
3   #include <pthread.h>
4
5   #define THREADS 4
6   #define N 3000
7
8   int primes[N];
9   int pflag[N];
10  int total = 0;
11
12  int is_prime(int v)
13  {
14      int i;
15      int bound = floor(sqrt ((double)v)) + 1;
16
17      for (i = 2; i < bound; i++) {
18          /* No need to check against known composites */
19          if (!pflag[i])
20              continue;
21          if (v % i == 0) {
22              pflag[v] = 0;
23              return 0;
24          }
25      }
26      return (v > 1);
27  }
28
29  void *work(void *arg)
30  {
31      int start;
32      int end;
33      int i;
34
35      start = (N/THREADS) * (*(int *)arg) ;
36      end = start + N/THREADS;
37      for (i = start; i < end; i++) {
38          if ( is_prime(i) ) {
39              primes[total] = i;
40              total++;
41          }
42      }
43      return NULL;
44  }
45
46  int main(int argn, char **argv)
```

```
47  {
48      int i;
49      pthread_t tids[THREADS-1];
50
51      for (i = 0; i < N; i++) {
52          pflag[i] = 1;
53      }
54
55      for (i = 0; i < THREADS-1; i++) {
56          pthread_create(&tids[i], NULL, work, (void *)&i);
57      }
58
59      i = THREADS-1;
60      work((void *)&i);
61
62      printf("Number of prime numbers between 2 and %d: %d\n",
63             N, total);
64      for (i = 0; i < total; i++) {
65          printf("%d\n", primes[i]);
66      }
67
68      return 0;
69  }
```

2.1.2.1 Data Races in `omp_prime.c` and `pthr_prime.c`

As noted in the "2.1.1 Complete Listing of `omp_prime.c`" on page 15, the order of memory accesses is non-deterministic when code contains a race condition and the computation gives different results from run to run. Each execution of `omp_prime.c` produces incorrect and inconsistent results because of the data races in the code. An example of the output is shown below:

```
% cc -xopenmp=noopt omp_prime.c -lm
% a.out | sort -n
0
0
0
0
0
0
0
Number of prime numbers between 2 and 3000: 336
2
3
5
7
11
13
```

```
17
19
23
29
31
37
41
43
47
53
59
61
67
71
...
2971
2999

% a.out | sort -n
0
0
0
0
0
0
0
0
0
Number of prime numbers between 2 and 3000: 325
3
5
7
13
17
19
23
29
31
41
43
47
61
67
71
73
79
83
89
```

```
101
...
2971
2999
```

Similarly, as a result of data-races in `pthr_prime.c`, different runs of the program may produce incorrect and inconsistent results as shown below.

```
% cc pthr_prime.c -lm -mt
% a.out | sort -n
Number of prime numbers between 2 and 3000: 304
751
757
761
769
773
787
797
809
811
821
823
827
829
839
853
857
859
863
877
881
...
2999
2999

% a.out | sort -n
Number of prime numbers between 2 and 3000: 314
751
757
761
769
773
787
797
809
811
821
823
```

```
827
839
853
859
877
881
883
907
911
...
2999
2999
```

2.2 Creating Experiments

The Thread Analyzer follows the same "collect-analyze" model that the Sun Studio Performance Analyzer uses. There are three steps involved in using the Thread Analyzer:

- "2.2.1 Instrument the Source Code" on page 21
- "2.2.2 Create a Data-Race Detection Experiment" on page 22
- "2.2.3 Examine the Data-Race Detection Experiment" on page 22

2.2.1 Instrument the Source Code

In order to enable data-race detection in a program, the source files must first be compiled with a special compiler option. This special option for the C, C++, and Fortran languages is: `-xinstrument=datarace`

Add the `-xinstrument=datarace` compiler option to the existing set of options you use to compile your program. You can apply the option to only the source files that you suspect to have data-races.

Note – Be sure to specify `-g` when you compile your program. 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 callstacks, may be incorrect when a high optimization level is used.

The following are example commands for instrumenting the source code:

- `cc -xinstrument=datarace -g -mt pthr_prime.c`
- `cc -xinstrument=datarace -g -xopenmp=noopt omp_prime.c`

2.2.2 Create a Data-Race Detection Experiment

Use the `collect` command with the `-r` onflag 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. The following is an example command that creates a data-race experiment:

- `collect -r race./a.out`

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 in the different experiments.

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 former presents a simplified set of default tabs, but is otherwise identical to the Performance Analyzer.

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. 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 trace selected from the Races tab.

The `er_print` utility, on the other hand, presents a command-line interface. The following subcommands are useful for examining races with the `er_print` utility:

- `-races`: This reports any data races revealed in the experiment.
- `-rdetail race_id`: This displays detailed information about the data-race with the specified `race_id`. If the specified `race_id` is "all", then detailed information about all data-races will be displayed.
- `-header`: This displays descriptive information about the experiment, and reports any errors or warnings.

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 traces associated with the data-race. Each trace refers to the pair of thread callstacks at the time the two data-race accesses occurred. If you are using the GUI, the two callstacks will be displayed in the Race Details tab when an individual trace is selected. If you are using the `er_print` utility, the two callstacks will be displayed by the `rdetail` command.

2.3.1 Data Races in `omp_prime.c`

```
% cc -xopenmp=noopt omp_prime.c -lm -xinstrument=datarace

% collect -r race a.out | sort -n
0
0
0
0
0
0
0
0
0
0
...
0
0
Creating experiment database test.1.er ...
Number of prime numbers between 2 and 3000: 429
2
3
5
7
```

```
11
13
17
19
23
29
31
37
41
47
53
59
61
67
71
73
...
2971
2999

% er_print test.1.er
(er_print) races

Total Races:  4 Experiment:  test.1.er

Race #1, Vaddr: 0xffbfeec4
      Access 1: Read,  main -- MP doall from line 42 [_$d1A42.main] + 0x00000060,
                       line 45 in "omp_prime.c"
      Access 2: Write, main -- MP doall from line 42 [_$d1A42.main] + 0x0000008C,
                       line 46 in "omp_prime.c"
  Total Traces: 2

Race #2, Vaddr: 0xffbfeec4
      Access 1: Write, main -- MP doall from line 42 [_$d1A42.main] + 0x0000008C,
                       line 46 in "omp_prime.c"
      Access 2: Write, main -- MP doall from line 42 [_$d1A42.main] + 0x0000008C,
                       line 46 in "omp_prime.c"
  Total Traces: 1

Race #3, Vaddr: (Multiple Addresses)
      Access 1: Write, main -- MP doall from line 42 [_$d1A42.main] + 0x0000007C,
                       line 45 in "omp_prime.c"
      Access 2: Write, main -- MP doall from line 42 [_$d1A42.main] + 0x0000007C,
                       line 45 in "omp_prime.c"
  Total Traces: 1

Race #4, Vaddr: 0x21418
      Access 1: Read,  is_prime + 0x00000074,
```

```
                         line 18 in "omp_prime.c"
      Access 2: Write, is_prime + 0x00000114,
                         line 21 in "omp_prime.c"
  Total Traces: 1
(er_print)
```

The following screen-shot shows the races that were detected in `omp_primes.c` as displayed by the Thread Analyzer GUI. The command to invoke the GUI and load the experiment data is `tha test.1.er`.

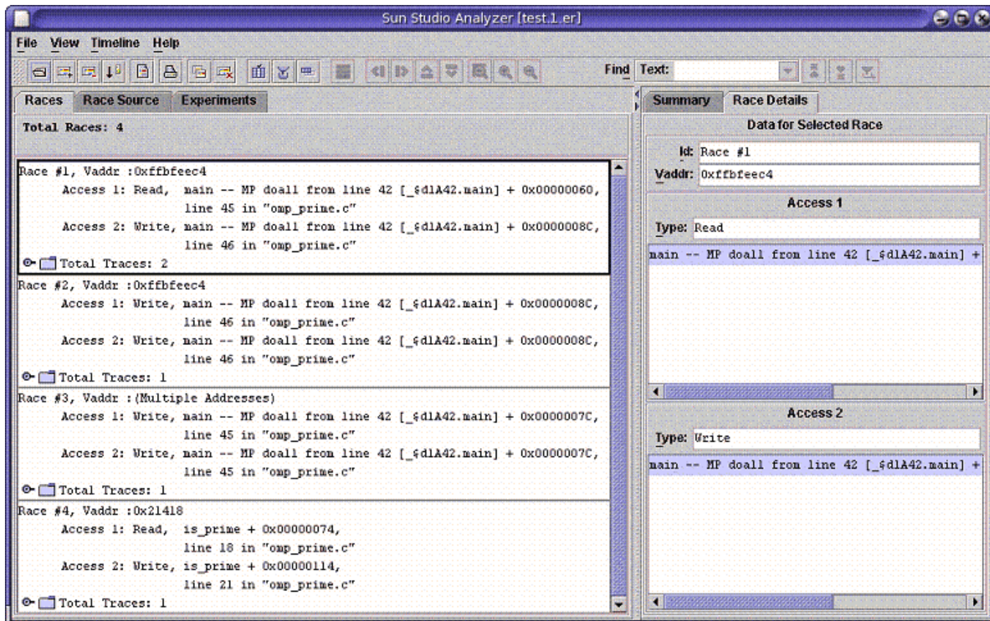


FIGURE 2–1 Data Races Detected in `omp_primes.c`

There are four data-races in `omp_primes.c`:

- Race number one: A data-race between a read from `total` on line 45 and a write to `total` on line 46.
- Race number two: A data-race between a write to `total` on line 46 and another write to `total` on the same line.
- Race number three: A data-race between a write to `primes[]` on line 45 and another write to `primes[]` on the same line.
- Race number four: A data-race between a read from `pflag[]` on line 18 and a write to `pflag[]` on line 21.

2.3.2 Data Races in `pthr_prime.c`

```
% cc pthr_prime.c -lm -mt -xinstrument=datarace                                .
% collect -r on a.out | sort -n

Creating experiment database test.2.er ...
of type "nfs", which may distort the measured performance.
0
0
0
0
0
0
0
0
0
0
...
0
0
Creating experiment database test.2.er ...
Number of prime numbers between 2 and 3000: 328
751
757
761
773
797
809
811
821
823
827
829
839
853
857
859
877
881
883
887
907
...
2999
2999

% er_print test.2.er
(er_print) races
```

```
Total Races:  6 Experiment:  test.2.er

Race #1, Vaddr: 0x218d0
      Access 1: Write, work + 0x00000154,
                       line 40 in "pthr_prime.c"
      Access 2: Write, work + 0x00000154,
                       line 40 in "pthr_prime.c"
  Total Traces: 3

Race #2, Vaddr: 0x218d0
      Access 1: Read,  work + 0x000000CC,
                       line 39 in "pthr_prime.c"
      Access 2: Write, work + 0x00000154,
                       line 40 in "pthr_prime.c"
  Total Traces: 3

Race #3, Vaddr: 0xffbfeec4
      Access 1: Write, main + 0x00000204,
                       line 55 in "pthr_prime.c"
      Access 2: Read,  work + 0x00000024,
                       line 35 in "pthr_prime.c"
  Total Traces: 2

Race #4, Vaddr: (Multiple Addresses)
      Access 1: Write, work + 0x00000108,
                       line 39 in "pthr_prime.c"
      Access 2: Write, work + 0x00000108,
                       line 39 in "pthr_prime.c"
  Total Traces: 1

Race #5, Vaddr: 0x23bfc
      Access 1: Write, is_prime + 0x00000210,
                       line 22 in "pthr_prime.c"
      Access 2: Write, is_prime + 0x00000210,
                       line 22 in "pthr_prime.c"
  Total Traces: 1

Race #6, Vaddr: 0x247bc
      Access 1: Write, work + 0x00000108,
                       line 39 in "pthr_prime.c"
      Access 2: Read,  main + 0x00000394,
                       line 65 in "pthr_prime.c"
  Total Traces: 1
(er_print)
```

The following screen-shot shows the races detected in `pthr_primes.c` as displayed by the Thread Analyzer GUI. The command to invoke the GUI and load the experiment data is `tha test.2.er`.

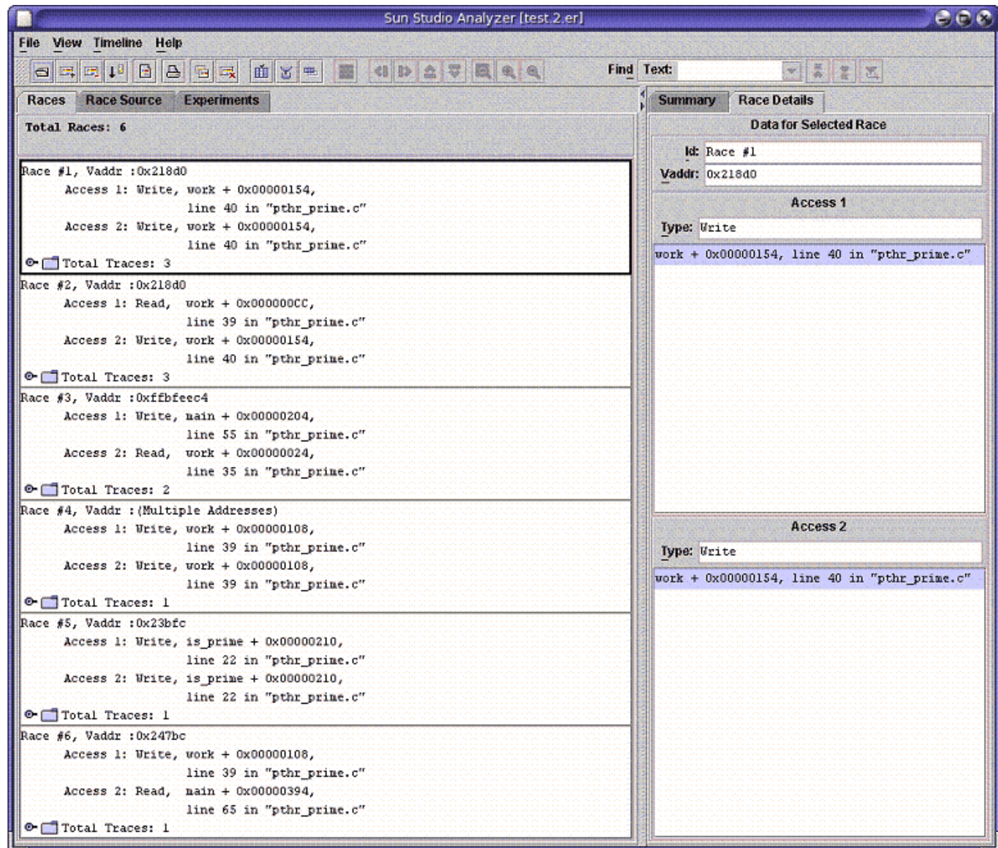


FIGURE 2–2 Data Races Detected in `pthr_primes.c`

There are six data-races in `pthr_prime.c`:

- Race number one: A data-race between a write to `total` on line 40 and another write to `total` on the same line.
- Race number two: A data-race between a read from `total` on line 39 and a write to `total` on line 40.
- Race number three: A data-race between a write to `i` on line 55 and a read from `i` on line 35.
- Race number four: A data-race between a write to `primes[]` on line 39 and another write to `primes[]` the same line.
- Race number five: A data-race between a write to `pflag[]` on line 22 and another write to `pflag[]` on the same line
- Race number six: A data-race between a write to `primes[]` on line 39 and a read from `primes[]` on line 65.

One advantage of the GUI is that it allows you to see, side by side, the two source locations associated with a data-race. For example, select race number six for `pthr_prime.c` in the Races tab and then click on the Dual Source tab. You will see the following:

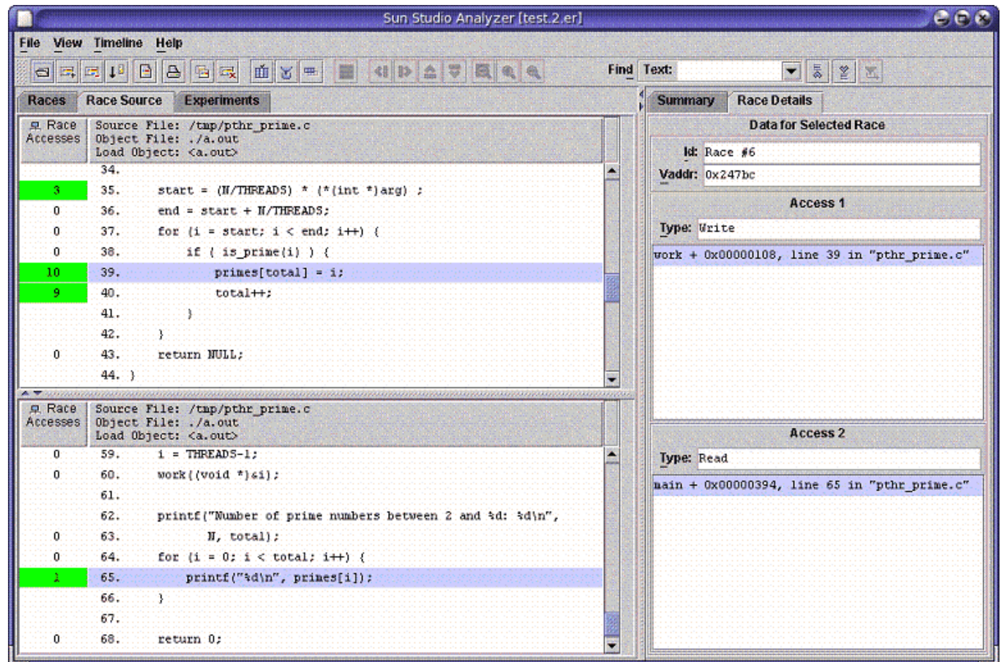


FIGURE 2–3 Source-Location Details of a Data Race

The first access for race number six (line 39) is shown in the top Race Source pane, while the second access for that data-race is shown in the bottom pane. Source lines 39 and 65, where the data-race accesses occurred, are highlighted. The default metric (Exclusive Race Accesses metric) is shown to the left of each source line. This metric gives a count of the number of times a data-race access was reported on that line.

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 33 for some examples of false positive data-races. For information on how to remove false positive data-races from the report, see “A.1 The Thread-Analyzer's User-APIs” on page 65.

2.4.2 Check Wether 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 multi-threaded 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 33 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. Fix the bug, not the data-race.

2.4.3.1 Fixing Bugs in `omp_prime.c`

Here's how to fix the bug in `omp_prime.c`. See “2.1.1 Complete Listing of `omp_prime.c`” on page 15 for a complete file listing.

Move lines 45 and 46 into a critical section in order to remove the data-race between the read from `total` on line 45 and the write to `total` on line 46. The critical section protects the two lines and prevents the data-race. Here is the corrected code:

```
42      #pragma omp parallel for         .
43      for (i = 2; i < N; i++) {
44          if ( is_prime(i) ) {
                  #pragma omp critical

                  {
 45                  primes[total] = i;
 46                  total++;
                  }
 47           }
 48       }
```

Note that the addition of a single critical section also fixes two other data races in omp_prime.c. It fixes the data-race on prime[] at line 45, as well as the data-race on total at line 46. The fourth data-race, between a read from pflag[] from line 18 and a write to pflag[] from line 21, is actually a benign race because it does not lead to incorrect results. It is not essential to fix benign data-races.

You could also move lines 45 and 46 into a critical section as follows, but this change fails to correct the program:

```
42      #pragma omp parallel for        .
43      for (i = 2; i < N; i++) {
44          if ( is_prime(i) ) {
                 #pragma omp critical
                 {
45                 primes[total] = i;
                 }
                 #pragma omp critical
                 {

46                 total++;
                 }
47          }
48      }
```

The critical sections around lines 45 and 46 get rid of the data-race because the threads are not using any exclusive locks to control their accesses to total. The critical section around line 46 ensures that the computed value of total is correct. However, the program is still incorrect. Two threads may update the same element of primes[] using the same value of total. Moreover, some elements in primes[] may not be assigned a value at all.

2.4.3.2 Fixing Bugs in pthr_prime.c

Here's how to fix the bug in pthr_prime.c. See "2.1.2 Complete Listing of pthr_prime.c" on page 17 for a complete file listing.

Use a single mutex to remove the data-race in pthr_prime.c between the read from total on line 39 and the write to total on line 40. This addition also fixes two other data races in pthr_prime.c: the data-race on prime[] at line 39, as well as the data-race on total at line 40.

The data-race between the write to i on line 55 and the read from i on line 35 and the data-race on pflag[] on line 22, reveal a problem in the shared-access to the variable i by different threads. The initial thread in pthr_prime.c creates the child threads in a loop (source lines 55-57), 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. This ensures that each thread has its own private copy of `i` with a unique value. To remove the data-race on `primes[]` between the write access on line 39 and the read access on line 65, we can protect line 65 with the same mutex lock as the one used above for lines 39 and 40. However, this is not the correct fix. The real problem is that the main thread may report the result (lines 50 through 53) while the child threads are still updating `total` and `primes[]` in function `work()`. Using mutex locks does not provide the proper ordering synchronization between the threads. One correct fix is to let the main thread wait for all child threads to join it before printing out the results.

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

```
1    #include <stdio.h>
2    #include <math.h>
3    #include <pthread.h>
4
5    #define THREADS 4
6    #define N 3000
7
8    int primes[N];
9    int pflag[N];
10    int total = 0;
11    pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
12
13    int is_prime(int v)
14    {
15        int i;
16        int bound = floor(sqrt(v)) + 1;
17
18        for (i = 2; i < bound; i++) {
19            /* no need to check against known composites */
20            if (!pflag[i])
21                continue;
22            if (v % i == 0) {
23                pflag[v] = 0;
24                return 0;
25            }
26        }
27        return (v > 1);
28    }
29
30    void *work(void *arg)
31    {
32        int start;
33        int end;
34        int i;
35
36        start = (N/THREADS) * ((int)arg) ;
37        end = start + N/THREADS;
```

```
38      for (i = start; i < end; i++) {
39          if ( is_prime(i) ) {
40              pthread_mutex_lock(&mutex);
41              primes[total] = i;
42              total++;
43              pthread_mutex_unlock(&mutex);
44          }
45      }
46      return NULL;
47  }
48
49  int main(int argn, char **argv)
50  {
51      int i;
52      pthread_t tids[THREADS-1];
53
54      for (i = 0; i < N; i++) {
55          pflag[i] = 1;
56      }
57
58      for (i = 0; i < THREADS-1; i++) {
59          pthread_create(&tids[i], NULL, work, (void *)i);
60      }
61
62      i = THREADS-1;
63      work((void *)i);
64
65      for (i = 0; i < THREADS-1; i++) {
66          pthread_join(tids[i], NULL);
67      }
68
69      printf("Number of prime numbers between 2 and %d: %d\n",
70             N, total);
71      for (i = 0; i < total; i++) {
72          printf("%d\n", primes[i]);
73      }
74  }
```

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 "2.5.1 User-Defined Synchronizations" on page 34 or "2.5.2 Memory That is Recycled by Different Threads" on page 35.

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. For example, the tool cannot recognize implementation of locks using CAS instructions, post and wait operations using busy-waits, etc. 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.

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 The Thread-Analyzer's User-APIs" on page 65 for more information.

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 multi-threaded 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 the following file, `omp_prime.c` check whether an integer is a prime number by executing the function `is_prime()`.

```
11 int is_prime(int v)
    12 {
    13     int i;
    14     int bound = floor(sqrt ((double)v)) + 1;
    15
    16     for (i = 2; i < bound; i++) {
    17         /* No need to check against known composites */
    18         if (!pflag[i])
    19             continue;
    20         if (v % i == 0) {
    21             pflag[v] = 0;
    22             return 0;
    23         }
    24     }
    25     return (v > 1);
    26 }
```

The Thread Analyzer reports that there is a data-race between the write to `pflag[]` on line 21 and the read of `pflag[]` on line 18. However, this data-race is benign as it does not affect the correctness of the final result. At line 18, 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 `v` is divisible by `i`; we 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 18) before it has been assigned the value zero, it then executes lines 20-23. If, in the meantime, another thread assigns zero to that same `pflag[]` element (line 21), the final result is not changed. Essentially, this means that the first thread executed lines 20-23 unnecessarily.

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 corrupt. Each thread checks a different section of the array. If a thread finds that an element is corrupt, 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 multi-threaded 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 };
 ...

 200 Singleton* Singleton::ptr_instance = 0;
 ...

 300 Singleton* Singleton::instance() {
 301     Singleton *tmp = ptr_instance;
 302     memory_barrier();
 303     if (tmp == NULL) {
 304         Lock();
 305         if (ptr_instance == NULL) {
 306             tmp = new Singleton;
 307             memory_barrier();
 308             ptr_instance = tmp;
 309         }
 310         Unlock();
 311     }
 312     return tmp;
 313 }
```

The read of `ptr_instance` (line 301) is intentionally not protected by a lock. This makes the check to determine whether or not the singleton has already been instantiated in a multi-threaded environment efficient. Notice that there is a data-race on variable `ptr_instance` between the read on line 301 and the write on line 308, but the program works correctly. However, writing a correct program that allows data-races is a difficult task. For example, in the above double-checked-locking code, the calls to `memory_barrier()` at lines 302 and 307 are used to ensure that the singleton and `ptr_instance` are set, and read, in the proper order. Consequently, all threads read them consistently. This programming technique will not work if the memory barriers are not used.

CHAPTER 3

The Deadlock Tutorial

This tutorial explains how to use the Thread Analyzer to detect potential, as well as actual, deadlocks in your multi-threaded program. The term 'deadlock' describes a condition in which two or more threads are blocked (hung) forever because they are waiting for each other. There are many causes of deadlocks such as erroneous program logic, inappropriate use of synchronizations and barriers. This tutorial focuses on deadlocks that are caused by the inappropriate use of mutual exclusion locks. This type of deadlock is commonly encountered in multi-threaded 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.

3.1 The Dining Philosophers Source File

The sample program which simulates the dining-philosophers problem is a C program that uses POSIX threads. The source file is called `din_philo.c`. The program can exhibit both potential and actual deadlocks. Here is the listing of the code which is followed by an explanation:

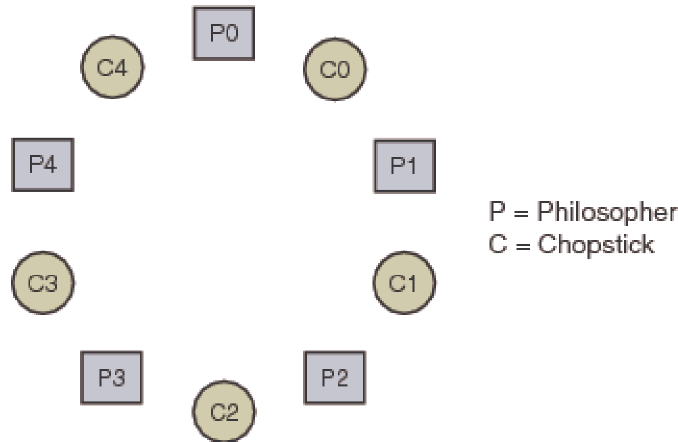
```
/* din_philo.c */
     1  #include <pthread.h>
     2  #include <stdio.h>
     3  #include <unistd.h>
     4  #include <stdlib.h>
     5  #include <errno.h>
     6  #include <assert.h>
     7
     8  #define PHILOS 5
     9  #define DELAY 5000
    10  #define FOOD 50
    11
    12    void *philosopher (void *id);
    13    void grab_chopstick (int,
    14                         int,
    15                         char *);
    16    void down_chopsticks (int,
    17                          int);
    18    int food_on_table ();
    19
    20    pthread_mutex_t chopstick[PHILOS];
    21    pthread_t philo[PHILOS];
    22    pthread_mutex_t food_lock;
    23    int sleep_seconds = 0;
    24
    25
    26    int
    27    main (int argn,
    28          char **argv)
    29    {
    30        int i;
    31
    32        if (argn == 2)
    33            sleep_seconds = atoi (argv[1]);
    34
    35        pthread_mutex_init (&food_lock, NULL);
    36        for (i = 0; i < PHILOS; i++)
    37            pthread_mutex_init (&chopstick[i], NULL);
    38        for (i = 0; i < PHILOS; i++)
    39            pthread_create (&philo[i], NULL, philosopher, (void *)i);
    40        for (i = 0; i < PHILOS; i++)
```

```
41            pthread_join (philo[i], NULL);
42        return 0;
43    }
44
45    void *
46    philosopher (void *num)
47    {
48        int id;
49        int i, left_chopstick, right_chopstick, f;
50
51        id = (int)num;
52        printf ("Philosopher %d is done thinking and now ready to eat.\n", id);
53        right_chopstick = id;
54        left_chopstick = id + 1;
55
56        /* Wrap around the chopsticks. */
57        if (left_chopstick == PHILOS)
58            left_chopstick = 0;
59
60        while (f = food_on_table ()) {
61
62            /* Thanks to philosophers #1 who would like to take a nap
63             * before picking up the chopsticks, the other philosophers
64             * may be able to eat their dishes and not deadlock.
65             */
66            if (id == 1)
67                sleep (sleep_seconds);
68
69            grab_chopstick (id, right_chopstick, "right ");
70            grab_chopstick (id, left_chopstick, "left");
71
72            printf ("Philosopher %d: eating.\n", id);
73            usleep (DELAY * (FOOD - f + 1));
74            down_chopsticks (left_chopstick, right_chopstick);
75        }
76
77        printf ("Philosopher %d is done eating.\n", id);
78        return (NULL);
79    }
80
81    int
82    food_on_table ()
83    {
84        static int food = FOOD;
85        int myfood;
86
87        pthread_mutex_lock (&food_lock);
88        if (food > 0) {
```

```
 89             food--;
 90         }
 91         myfood = food;
 92         pthread_mutex_unlock (&food_lock);
 93         return myfood;
 94     }
 95
 96     void
 97     grab_chopstick (int phil,
 98                     int c,
 99                     char *hand)
100     {
101         pthread_mutex_lock (&chopstick[c]);
102         printf ("Philosopher %d: got %s chopstick %d\n", phil, hand, c);
103     }
104
105     void
106     down_chopsticks (int c1,
107                      int c2)
108     {
109         pthread_mutex_unlock (&chopstick[c1]);
110         pthread_mutex_unlock (&chopstick[c2]);
111     }
```

3.2 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 left of each philosopher (as they sit facing the table) has the same number as that philosopher.



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.

3.2.1 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 zero is holding chopstick zero, but is waiting for chopstick one
Philosopher one is holding chopstick one, but is waiting for chopstick two
Philosopher two is holding chopstick two, but is waiting for chopstick three
Philosopher three is holding chopstick three, but is waiting for chopstick four
Philosopher four is holding chopstick four, but is waiting for chopstick zero

In this situation, nobody can eat and the philosophers are in a deadlock. Rerun the program a number of times and you will see that the program may sometimes hang, or run to completion at other times.

Run the dining philosophers program and see whether it completes or deadlocks. It may hang as shown in the following sample run:

```
prompt% cc din_phil.c -mt
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
```

3.2.2 Introducing a Sleep Time for Philosopher One

One possible solution to the deadlock potential is for philosopher one 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 one 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 one 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.

3.3 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 Sun 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.

3.3.1 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 callstacks, 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` for more information.

3.3.2 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.

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.

See `collect.1` and `collector.1` for more information.

3.3.3 Examine the Experiment Results

You can examine the deadlock-detection experiment with either the `tha` command, the `analyzer` command, or the `er_print` utility. Both the Thread Analyzer and the Analyzer present a GUI interface while er_print employes a command-line interface.

See `tha.1`, `analyzer.1`, and `er_print.1` for more information.

3.3.3.1 The Thread Analyzer Interface

The Thread Analyzer includes a menu bar, a tool bar, and a split pane that contains tabs for the various displays. The following three tabs are shown by default in the left-hand pane:

- 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 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 two 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.

3.3.3.2 The er_print Interface

In contrast to the left-hand pane, the right-hand pane contains the Deadlock Details tab which shows detailed information for the selected thread in the Deadlocks tab. The most useful subcommands for examining deadlocks with er_print are the following:

- `-deadlocks`

  The option reports any potential and actual deadlocks detected in the experiment.

- `-ddetail deadlock_id`

  This option returns detailed information about the deadlock with the specified `deadlock_id`. If you specify the value **`all`** as the `deadlock_id`, then er_print displays detailed information about all deadlocks.

- `-header`

  This option displays descriptive information about the experiment and reports any errors or warnings.

See `er_print.1` for more information.

3.4 Understanding the Experiment Results

This section explains how to use the Thread Analyzer to investigate the deadlocks in the dining philosopher program. We'll start by executing runs that result in actual deadlocks and then examine runs that terminate normally but have the potential for deadlocks.

3.4.1 Examining Runs That Deadlock

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

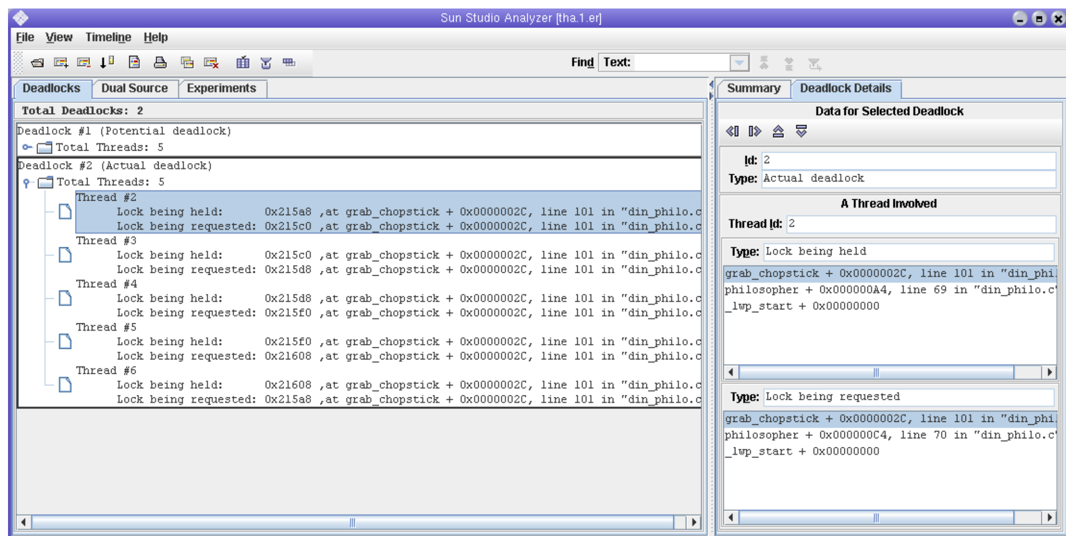
```
prompt% cc din_philo.c -mt -g

prompt% collect -r deadlock a.out
Creating experiment database tha.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 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
```

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

Deadlock #1, Potential deadlock
    Thread #2
        Lock being held:    0x215a8, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
        Lock being requested:    0x215c0, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
    Thread #3
        Lock being held:    0x215c0, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
        Lock being requested:    0x215d8, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
    Thread #4
        Lock being held:    0x215d8, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
        Lock being requested:    0x215f0, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
    Thread #5
        Lock being held:    0x215f0, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
        Lock being requested:    0x21608, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
    Thread #6
        Lock being held:    0x21608, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
        Lock being requested:    0x215a8, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"

Deadlock #2, Actual deadlock
    Thread #2
        Lock being held:    0x215a8, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
        Lock being requested:    0x215c0, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
    Thread #3
        Lock being held:    0x215c0, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
        Lock being requested:    0x215d8, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
    Thread #4
        Lock being held:    0x215d8, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
```

```
        Lock being requested:    0x215f0, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
    Thread #5
        Lock being held:    0x215f0, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
        Lock being requested:    0x21608, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
    Thread #6
        Lock being held:    0x21608, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
        Lock being requested:    0x215a8, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"

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

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

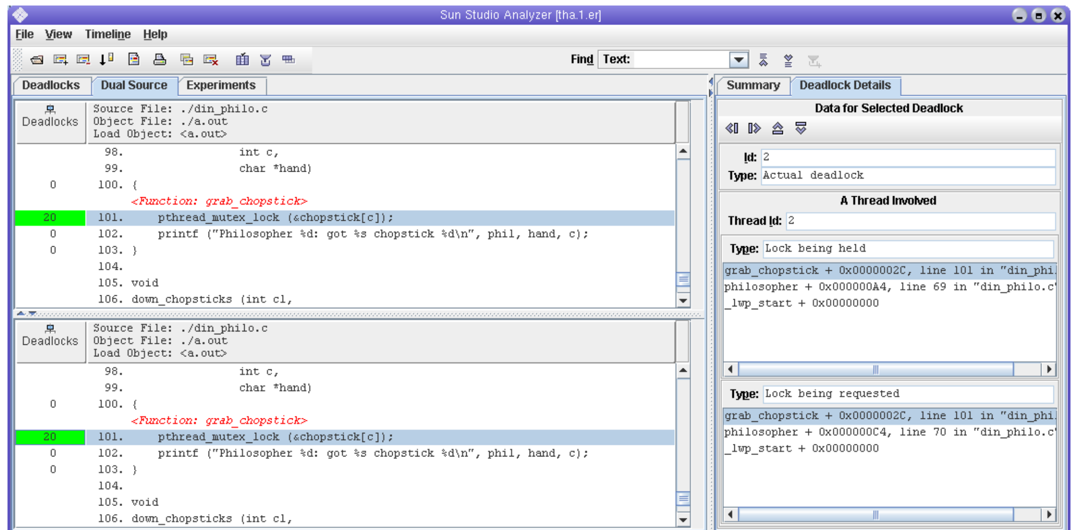


The Thread Analyzer reports two deadlocks for `din_philo.c`, one potential and the other actual. On closer inspection, we find that the two deadlocks are identical. The circular chain involved in the deadlock is as follows:

Thread 2: holds lock at address `0x215a8`, requests lock at address `0x215c0`
Thread 3: holds lock at address `0x215c0`, requests lock at address `0x215d8`
Thread 4: holds lock at address `0x215d8`, requests lock at address `0x215f0`
Thread 5: holds lock at address `0x215f0`, requests lock at address `0x21608`
Thread 6: holds lock at address `0x21608`, requests lock at address `0x215a8`

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 held the lock at address `0x215a8`, and where in the source code it requested the lock at address `0x215c0`. The following screen-shot shows the Dual Source tab for thread number two. The default metric (Exclusive Deadlocks metric) is shown to the left

of each source line. This metric gives a count of the number of times a lock-hold or lock-request operation, which was involved in a deadlock, was reported on that line.



3.4.2 Examining Runs That Complete Despite Deadlock Potential

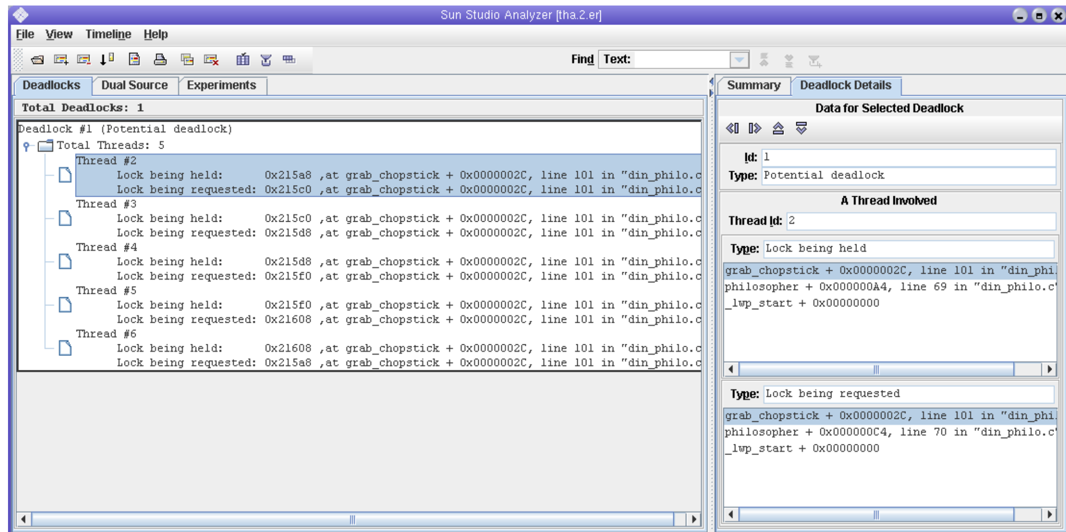
The dining philosophers program can avoid actual deadlock and terminate normally if you supply a large enough sleep argument. Normal termination, however, does not mean the program is safe from deadlocks. It simply means that the locks held and requested did not form a deadlock chain during a given run. 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. However, the er_print utility and the Thread Analyzer report potential deadlocks.

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


% collect -r deadlock a.out 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 right  chopstick 0
Philosopher 4: got right  chopstick 4
Philosopher 2: got left chopstick 3
Philosopher 2: eating.
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 3: got right  chopstick 3
Philosopher 2: got right  chopstick 2
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 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 4: got right  chopstick 4
Philosopher 2: got left chopstick 3
Philosopher 2: eating.
Philosopher 4: got left chopstick 0
Philosopher 4: eating.
Philosopher 3: got right  chopstick 3
Philosopher 2: got right  chopstick 2
Philosopher 0: got right  chopstick 0
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 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 4: got right  chopstick 4
Philosopher 3: got right  chopstick 3
Philosopher 4: got left chopstick 0
Philosopher 4: eating.
Philosopher 0: got right  chopstick 0
Philosopher 3: got left chopstick 4
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 3: eating.
Philosopher 0: got right  chopstick 0
Philosopher 2: got left chopstick 3
Philosopher 2: eating.
Philosopher 0: got left chopstick 1
```

```
Philosopher 0: 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


% er_print tha.2.er
(er_print) deadlocks

Deadlock #1, Potential deadlock
    Thread #2
        Lock being held:    0x215a8, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
        Lock being requested:    0x215c0, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
    Thread #3
        Lock being held:    0x215c0, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
        Lock being requested:    0x215d8, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
    Thread #4
        Lock being held:    0x215d8, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
        Lock being requested:    0x215f0, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
    Thread #5
        Lock being held:    0x215f0, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
        Lock being requested:    0x21608, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
    Thread #6
        Lock being held:    0x21608, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"
        Lock being requested:    0x215a8, at: grab_chopstick + 0x0000002C, line 101 in "din_philo.c"

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

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



3.5 Fixing the Deadlocks and Understanding False-Positives

In addition to the strategy of philosophers waiting before they start to eat, we can use a system of tokens in which 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.

3.5.1 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`:

```
1    #include <pthread.h>
 2    #include <stdio.h>
 3    #include <unistd.h>
 4    #include <stdlib.h>
 5    #include <errno.h>
 6    #include <assert.h>
 7
 8    #define PHILOS 5
 9    #define DELAY 5000
10    #define FOOD 50
11
12    void *philosopher (void *id);
13    void grab_chopstick (int,
14                         int,
15                         char *);
16    void down_chopsticks (int,
17                          int);
18    int food_on_table ();
19    int get_token ();
20    void return_token ();
21
22    pthread_mutex_t chopstick[PHILOS];
23    pthread_t philo[PHILOS];
24    pthread_mutex_t food_lock;
25    pthread_mutex_t num_can_eat_lock;
26    int sleep_seconds = 0;
27    uint32_t num_can_eat = PHILOS - 1;
28
29
30    int
31    main (int argn,
32          char **argv)
33    {
34        int i;
35
36        pthread_mutex_init (&food_lock, NULL);
37        pthread_mutex_init (&num_can_eat_lock, NULL);
38        for (i = 0; i < PHILOS; i++)
39            pthread_mutex_init (&chopstick[i], NULL);
40        for (i = 0; i < PHILOS; i++)
41            pthread_create (&philo[i], NULL, philosopher, (void *)i);
```

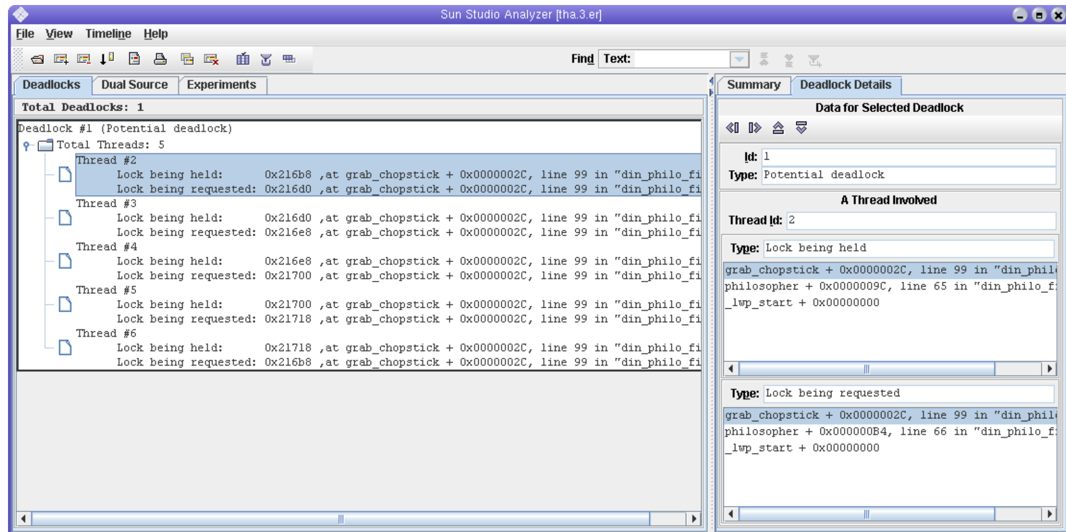
```
42      for (i = 0; i < PHILOS; i++)
43          pthread_join (philo[i], NULL);
44      return 0;
45  }
46
47  void *
48  philosopher (void *num)
49  {
50      int id;
51      int i, left_chopstick, right_chopstick, f;
52
53      id = (int)num;
54      printf ("Philosopher %d is done thinking and now ready to eat.\n", id);
55      right_chopstick = id;
56      left_chopstick = id + 1;
57
58      /* Wrap around the chopsticks. */
59      if (left_chopstick == PHILOS)
60          left_chopstick = 0;
61
62      while (f = food_on_table ()) {
63          get_token ();
64
65          grab_chopstick (id, right_chopstick, "right ");
66          grab_chopstick (id, left_chopstick, "left");
67
68          printf ("Philosopher %d: eating.\n", id);
69          usleep (DELAY * (FOOD - f + 1));
70          down_chopsticks (left_chopstick, right_chopstick);
71
72          return_token ();
73      }
74
75      printf ("Philosopher %d is done eating.\n", id);
76      return (NULL);
77  }
78
79  int
80  food_on_table ()
81  {
82      static int food = FOOD;
83      int myfood;
84
85      pthread_mutex_lock (&food_lock);
86      if (food > 0) {
87          food--;
88      }
89      myfood = food;
```

```
 90       pthread_mutex_unlock (&food_lock);
 91       return myfood;
 92   }
 93
 94   void
 95   grab_chopstick (int phil,
 96                   int c,
 97                   char *hand)
 98   {
 99       pthread_mutex_lock (&chopstick[c]);
100       printf ("Philosopher %d: got %s chopstick %d\n", phil, hand, c);
101   }
102
103   void
104   down_chopsticks (int c1,
105                    int c2)
106   {
107       pthread_mutex_unlock (&chopstick[c1]);
108       pthread_mutex_unlock (&chopstick[c2]);
109   }
110
111
112   int
113   get_token ()
114   {
115       int successful = 0;
116
117       while (!successful) {
118           pthread_mutex_lock (&num_can_eat_lock);
119           if (num_can_eat > 0) {
120               num_can_eat--;
121               successful = 1;
122           }
123           else {
124               successful = 0;
125           }
126           pthread_mutex_unlock (&num_can_eat_lock);
127       }
128   }
129
130   void
131   return_token ()
132   {
133       pthread_mutex_lock (&num_can_eat_lock);
134       num_can_eat++;
135       pthread_mutex_unlock (&num_can_eat_lock);
136   }
```

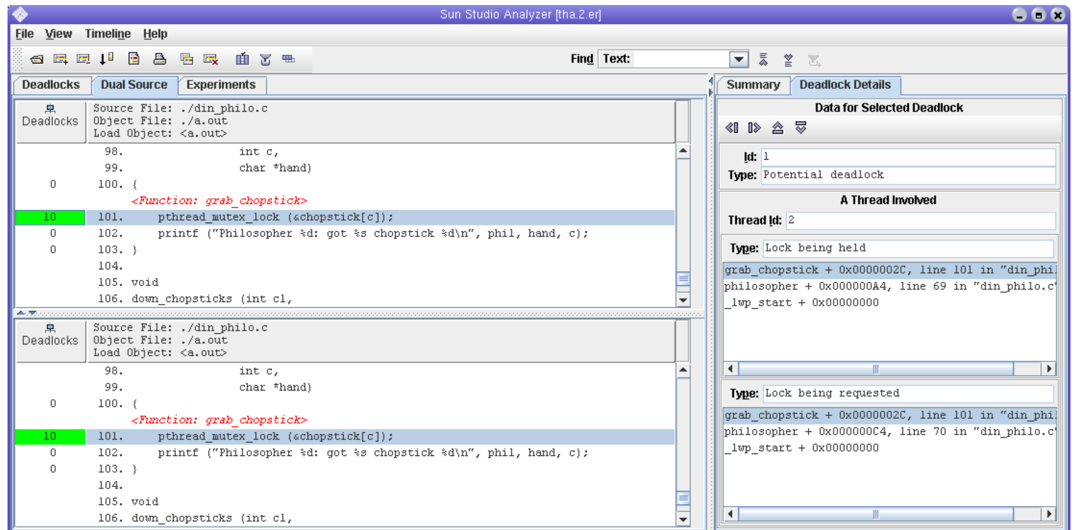
Try compiling and running this fixed version of the dining philosophers program and running it several times. The system of tokens limits the number of diners attempting to use the chopsticks and thus avoids actual and potential deadlocks.

3.5.1.1 A False-Positive Report

In spite of 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:



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 `0x215a8`, and where in the source code it requested the lock at address `0x215c0`. The following screen-shot shows the Dual Source tab for Thread #2.



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.

3.5.2 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`.

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

The following listing details `din_philo_fix2.c`:

```
1    #include <pthread.h>
 2    #include <stdio.h>
 3    #include <unistd.h>
```

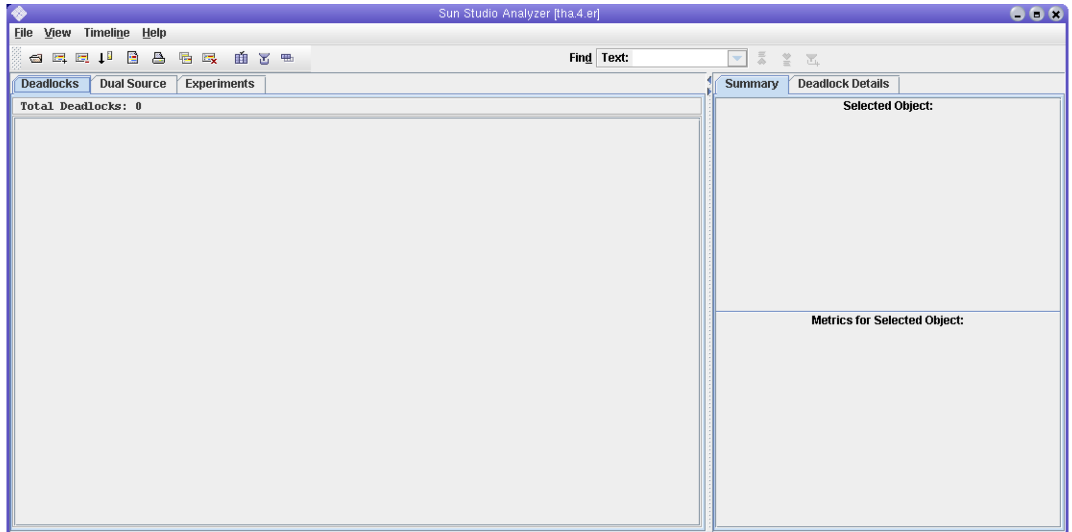
```
 4    #include <stdlib.h>
 5    #include <errno.h>
 6    #include <assert.h>
 7    #include <semaphore.h>
 8
 9    #define PHILOS 5
10    #define DELAY 5000
11    #define FOOD 50
12
13    void *philosopher (void *id);
14    void grab_chopstick (int,
15                         int,
16                         char *);
17    void down_chopsticks (int,
18                          int);
19    int food_on_table ();
20    int get_token ();
21    void return_token ();
22
23    pthread_mutex_t chopstick[PHILOS];
24    pthread_t philo[PHILOS];
25    pthread_mutex_t food_lock;
26    int sleep_seconds = 0;
27    sem_t num_can_eat_sem;
28
29
30    int
31    main (int argn,
32          char **argv)
33    {
34        int i;
35
36        pthread_mutex_init (&food_lock, NULL);
37        sem_init(&num_can_eat_sem, 0, PHILOS - 1);
38        for (i = 0; i < PHILOS; i++)
39            pthread_mutex_init (&chopstick[i], NULL);
40        for (i = 0; i < PHILOS; i++)
41            pthread_create (&philo[i], NULL, philosopher, (void *)i);
42        for (i = 0; i < PHILOS; i++)
43            pthread_join (philo[i], NULL);
44        return 0;
45    }
46
47    void *
48    philosopher (void *num)
49    {
50        int id;
51        int i, left_chopstick, right_chopstick, f;
```

```
52
53      id = (int)num;
54      printf ("Philosopher %d is done thinking and now ready to eat.\n", id);
55      right_chopstick = id;
56      left_chopstick = id + 1;
57
58      /* Wrap around the chopsticks. */
59      if (left_chopstick == PHILOS)
60          left_chopstick = 0;
61
62      while (f = food_on_table ()) {
63          get_token ();
64
65          grab_chopstick (id, right_chopstick, "right ");
66          grab_chopstick (id, left_chopstick, "left");
67
68          printf ("Philosopher %d: eating.\n", id);
69          usleep (DELAY * (FOOD - f + 1));
70          down_chopsticks (left_chopstick, right_chopstick);
71
72          return_token ();
73      }
74
75      printf ("Philosopher %d is done eating.\n", id);
76      return (NULL);
77  }
78
79  int
80  food_on_table ()
81  {
82      static int food = FOOD;
83      int myfood;
84
85      pthread_mutex_lock (&food_lock);
86      if (food > 0) {
87          food--;
88      }
89      myfood = food;
90      pthread_mutex_unlock (&food_lock);
91      return myfood;
92  }
93
94  void
95  grab_chopstick (int phil,
96                  int c,
97                  char *hand)
98  {
99      pthread_mutex_lock (&chopstick[c]);
```

```
100         printf ("Philosopher %d: got %s chopstick %d\n", phil, hand, c);
101     }
102 
103     void
104     down_chopsticks (int c1,
105                      int c2)
106     {
107         pthread_mutex_unlock (&chopstick[c1]);
108         pthread_mutex_unlock (&chopstick[c2]);
109     }
110 
111 
112     int
113     get_token ()
114     {
115         sem_wait(&num_can_eat_sem);
116     }
117 
118     void
119     return_token ()
120     {
121         sem_post(&num_can_eat_sem);
122     }
```

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.

If you run this new implementation of the program several times, you will find that it terminates normally each time and does not hang. You will also find that the Thread Analyzer does not report any actual or potential deadlocks, as the following screen-shot shows:



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

APPENDIX A

Thread Analyzer User API

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.

If your code includes user-defined synchronizations, insert the 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 user APIs are listed below:

A.1 The Thread-Analyzer's User-APIs

TABLE A–1 Thread-Analyzer User-APIs

`tha_notify_acquire_lock(id)()`	Insert immediately before the program tries to acquire a user-defined lock.
`tha_notify_lock_acquired(id)()`	Insert immediately after a user-defined lock is successfully acquired.
`tha_notify_writelock_acquired(id)()`	Insert immediately after a user-defined read-write lock is successfully acquired in write mode.
`tha_notify_readlock_acquired(id)()`	Insert immediately after a user-defined read-write lock is successfully acquired in read mode.
`tha_notify_lock_released(id)()`	Insert immediately after a user-defined lock (including a read-write lock) is successfully released.

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

`tha_notify_sync_post_begin(id)()`	Insert immediately before a user-defined post synchronization is performed.
`tha_notify_sync_post_end(id)()`	Insert immediately after a user-defined post synchronization is performed.
`tha_notify_sync_wait_begin(id)()`	Insert immediately before a user-defined wait synchronization is performed.
`tha_notify_sync_wait_end(id)()`	Insert immediately after a user-defined wait synchronization is performed.

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:

```
# include <tha_interface.h>
...
/* Initially, the ready_flag value is zero */
...
/* Thread 1: Producer */
100   data = ...
101   pthread_mutex_lock (&mutex);
      tha_notify_sync_post_begin ((uintptr_t) &ready_flag);
102   ready_flag = 1;
      tha_notify_sync_post_end ((uintptr_t) &ready_flag);

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


/* Thread 2: Consumer */
200   pthread_mutex_lock (&mutex);
      tha_notify_sync_wait_begin ((uintptr_t) &ready_flag);
201   while (!ready_flag) {
202       pthread_cond_wait (&cond, &mutex);
```

```
203   }
      tha_notify_sync_wait_end ((uintptr_t) &ready_flag);
204   pthread_mutex_unlock (&mutex);
205   ... = 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

```
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

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()

A.2.4 OpenMP APIs

See the *Sun Studio 12: OpenMP API User's Guide* for more information.

APPENDIX B

Thread Analyzer Frequently Asked Questions

This section includes a list of frequently asked questions and their answers. See the Sun Developer Network (http://developers.sun.com/sunstudio/index.jsp) for the latest updates to this FAQ.

B.1 FAQ

Question: Why is the line-number information incorrect?

Answer: Try turning off optimization or specifying level `-x03` or lower. The compiler's optimization transformations can distort line number information and make the experiment result difficult to read.

Question: Do I really need to install the patches that the `collect` command is complaining about?

Answer: Yes. make sure the experiment system has all the required patches installed. Experiment results may be incorrect if any required patches are missing.

Question: Is it alright to link archive versions of `malloc()` libraries with my code?

Answer: No. The Thread Analyzer interposes on `malloc()` routines so linking archive versions of `malloc()` libraries can result in false-positive data races.

Question: Can the Thread Analyzer detect data-races in OpenMP applications? What about POSIX or Solaris thread applications?

Answer: The Thread Analyzer can detect data-races that occur in code that is written using the POSIX thread API, the Solaris Operating System(R) thread API, OpenMP directives, Sun parallel directives, Cray(R) parallel directives, or a mix of these.

Question: Can the Thread Analyzer detect data-races between different processes?

Answer: Not yet. It currently only detects data-races between different threads spawned from a single process.

Question: Is the Thread Analyzer able to find all data-races?

Answer: No. The Thread Analyzer detects data-races at run time and the exact runtime behavior of an application depends on the set of input data. A given input-data set may not lead to a data-race. The Thread Analyzer models the concurrency between threads at a high level in order to minimize the impact of scheduling by the operating system. However, the operating system scheduling can still affect memory allocation and storage reuse which changes the potential for data-races.

Use the Thread Analyzer with different numbers of threads and with different input data-sets and repeat experiments with a single data set to maximize the tool's chance of detecting data-races.

Question: Why does the Thread Analyzer give me different data-race results in different runs?

Answer: This occurs because of timing differences between runs. As the threads access memory in a different order from run to run, different data-race results will be reported.

Question: Why does the Thread Analyzer report data-races that do not exist in my application? How do I remove them?

Answer: In some cases, the Thread Analyzer may report data-races that never actually occur in the program. These are called false positives, which usually happen 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. See the tutorial for a detailed description of false positives and examples of how to remove them through API calls.

Question: What is `librdthooks.so` and what does it do?

Answer: `librdthooks.so` is a library that satisfies the entry points for the data-race-detection instrumentation calls and user API calls. It is linked automatically when a program is compiled and linked with `-xinstrument=datarace`. See the `librdthooks(3)` man page for more information.

Question: How do I know whether an executable or a library has been instrumented?

Answer: Use `nm`. See the `nm(1)` man page for more details. If you find a global undefined symbol of either `__rdt_src_read` or `__rdt_src_write`, then the executable or library is instrumented.

Question: Can I use the Analyzer to read data-race experiments?

Answer: Yes, the Analyzer displays all of the traditional performance analysis tabs as well as the new Races, Race Source, and Race Detail tabs. The Thread Analyzer interface is streamlined and does not display the traditional Analyzer tabs.

Question: Why do I get an error message saying that the compiler option `-xinstrument=datarace` is wrong when I use it with C, C++ or F90?

Answer: You are using an older version of Sun Studio that does not support the Thread Analyzer. Check the version of Sun Studio that you are using by entering: `cc -Version`. You must use a version that is no older than June 2006.

Question: Why do I get an error message when I use the `er_print` utility which says the races command is invalid?

Answer: You are using an older version of Sun Studio that does not support the Thread Analyzer. Check the version of Sun Studio that you are using by entering: `er_print -V`. You must use a version that is no older then June 2006.

Question: Why do I get an error message saying that `-r` is not recognized when I run `collect -r` on?

Answer: You are using an older version of Sun Studio that does not support the Thread Analyzer. Check the version of Sun Studio that you are using by entering `collect -V`. You must use a version that is no older then June 2006.

Question: How do I report a bug or share my Thread Analyzer experience with others?

Answer: The best resource for sharing your feedback with the Thread Analyzer engineers and users is by reading and posting to the Sun Studio Tools forum (`http://developers.sun.com/sunstudio/community/forums.jsp`). You may find that your question has already been answered.

Index

A
accessible documentation, 7-8

D
documentation, accessing, 6-9
documentation index, 6

T
Thread Analyzer, getting started, 21