Oracle® Developer Studio 12.5：线程分析器用户指南

文件号码 E71960
2016 年 6 月

ORACLE

文件号码 E71960

文档可访问性

有关 Oracle 对可访问性的承诺，请访问 Oracle Accessibility Program 网站 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc。

获得 Oracle 支持

购买了支持服务的 Oracle 客户可通过 My Oracle Support 获得电子支持。有关信息，请访问 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info；如果您听力受损，请访问 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs。

目录

使用本文档

- **概述** – 介绍线程分析器工具并提供了两套详尽的教程。其中一套重点介绍数据争用检测，另一套重点介绍死锁检测。本手册还有两个附录，分别介绍线程分析器可识别的 API 以及有用的提示。
- **目标读者** – 应用程序开发者、系统开发者、架构师、支持工程师
- **必备知识** – 编程经验、软件开发测试以及构建和编译软件产品的能力

产品文档库

有关该产品及相关产品的文档和资源，可从以下网址获得：http://www.oracle.com/pls/topic/lookup?ctx=E71940。

反馈

可以在 http://www.oracle.com/goto/docfeedback 上提供有关本文档的反馈。

♦ ♦ ♦ 第 1 章

1 什么是线程分析器及其执行什么操作？

线程分析器是一款 Oracle Developer Studio 工具，可用于分析多线程程序的执行情况。线程分析器可以检测多线程编程错误，例如用 POSIX 线程 API、Oracle Solaris 线程 API、OpenMP 指令或混用这几者编写的代码中的数据争用和死锁。

本章讨论以下主题：

线程分析器入门

线程分析器可以显示您专门创建的用来检查相应错误类型的实验中的数据争用和死锁，如本文档中所述。

线程分析器是性能分析器的一个专门视图，设计用于检查线程分析实验。有关详情，请参阅“线程分析器界面” [15]。

什么是数据争用？

线程分析器可检测多线程进程执行期间发生的数据争用。满足以下所有条件时，就会发生数据争用：

- 一个进程内的两个或多个线程同时访问同一内存位置
- 至少其中一个访问是用于写入
- 线程未使用任何互斥锁来控制它们对该内存的访问

这三个条件成立时，访问的顺序是不确定的，每次运行的计算结果都可能因该顺序而异。有些数据争用可能是良性的（例如，内存访问用于忙等待时），但许多数据争用都属于程序中的错误。

线程分析器适用于用 POSIX 线程 API、Oracle Solaris 线程 API、OpenMP 或混用这几者编写的多线程程序。

什么是死锁？

死锁描述的是两个或多个线程由于相互等待而永远被阻塞的情况。导致死锁的原因有很多。线程分析器可检测由于不恰当使用互斥锁而导致的死锁。此类死锁通常发生在多线程应用程序中。

满足以下所有条件时，包含两个或多个线程的进程可能会进入死锁状态：

- 已持有锁的线程请求新锁
- 同时发出对新锁的请求
- 两个或多个线程形成一个循环链，其中每个线程都在等待链中下一个线程持有的锁

以下是一个死锁情况的简单示例：

- 线程 1 持有锁 A 并请求锁 B
- 线程 2 持有锁 B 并请求锁 A

死锁可分为两种类型：潜在死锁或实际死锁。潜在死锁不一定在给定的运行过程中发生，但可能发生在程序的任何执行过程中，具体取决于线程的调度情况以及线程进行锁请求的时间。实际死锁是在程序执行过程中发生的死锁。实际死锁会导致所涉及的线程挂起，但可能会导致整个进程挂起。

线程分析器使用模型

下列步骤说明了使用线程分析器为多线程程序排除故障的过程。

1. 检测程序（如果进行数据争用检测）。
2. 创建数据争用检测实验或死锁检测实验。
3. 检查实验结果，并确定线程分析器所发现的多线程编程问题属于真实错误还是良性现象。
4. 修复真实的错误，并使用不同的因素（如不同的输入数据、不同的线程数、不同的循环调度，乃至不同的硬件）创建其他实验（上面的步骤 2）。这种重复有助于找出原因不确定的问题。

以下几节将介绍上述的步骤 1 到步骤 3。

检测数据争用的使用模型

必须执行下列三个步骤来检测数据争用：

1. 检测代码以允许数据争用检测
2. 基于检测后的代码创建实验
3. 检查实验中是否存在数据争用

检测代码以检测数据争用

要在应用程序中启用数据争用检测，必须首先检测代码来监视运行时内存访问，表示必须在代码中插入对运行时支持库 `libtha.so` 的调用以便监视运行时内存访问并确定是否存在任何数据争用。

可以在编译过程中在应用程序源代码级别检测代码，也可以通过对二进制代码运行其他工具在应用程序二进制代码级别检测代码。

源代码级别检测通过编译器完成，需要使用特殊选项。还可以指定优化级别以及要使用的其他编译器选项。由于编译器可以进行一些分析工作并检测较少的内存访问，因此源代码级别检测可使运行时速度更快。

源代码不可用时，可以使用二进制代码级别检测。有源代码时也可以使用二进制代码检测，但无法编译应用程序所使用的共享库。如果使用 `discover` 工具进行二进制代码检测，则可以检测二进制代码并可在所有共享库打开时检测所有这些共享库。

源代码级别检测

要在源代码级别进行检测，请使用特殊的编译器选项编译源代码：

```
-xinstrument=datarace
```

通过使用该编译器选项，将会对编译器生成的代码进行检测以便检测数据争用。

生成应用程序二进制代码时，还应使用 `-g` 编译器选项。该选项会导致生成额外数据，线程分析器通过该数据可以在报告数据争用时显示源代码和行号信息。

二进制代码级别检测

要在二进制代码级别进行检测，必须使用 `discover` 工具。如果二进制代码命名为 `a.out`，可以通过执行以下命令创建检测后的二进制代码 `a.outi`：

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

`discover` 工具在所有共享库打开时自动检测所有这些共享库（无论它们是静态链接到程序中的，还是通过 `dlopen`（） 动态打开的）。缺省情况下，检测后的库副本会缓存到 `$HOME/SUNW_Bit_Cache` 目录中。

下面显示了一些有用的 `discover` 命令行选项。有关详细信息，请参见 `discover`(1) 手册页。

`-o` *file*	将检测后的二进制代码输出到指定文件名
`-N` *lib*	不检测指定的库
`-T`	不检测任何库
`-D` *dir*	将高速缓存目录更改为 *dir*

基于检测后的应用程序创建实验

要创建数据争用检测实验，请使用带有 `-r race` 标志的 `collect` 命令运行应用程序并在进程执行期间收集实验数据。使用 `-r race` 选项时，收集到的数据包含构成争用的数据访问对。

检查实验中是否存在数据争用

可以使用 `tha` 命令检查数据争用检测实验，该命令会启动线程分析器图形用户界面。也可以使用 `er_print` 命令行界面。

检测死锁的使用模型

检测死锁包括两个步骤：

1. 创建死锁检测实验。
2. 检查实验中是否存在死锁。

创建用于检测死锁的实验

要创建死锁检测实验，请使用带有 `-r deadlock` 标志的 `collect` 命令运行应用程序并在进程执行期间收集实验数据。使用 `-r deadlock` 选项时，收集到的数据包含形成循环链的锁持有和锁请求。

检查实验中是否存在死锁

可以使用 `tha` 命令检查死锁检测实验，该命令会启动线程分析器图形用户界面。也可以使用 `er_print` 命令行界面。

检测数据争用和死锁的使用模型

如果要同时检测数据争用和死锁，请按照“检测数据争用的使用模型” [12]中的三个步骤来检测数据争用，但要使用带有 `r race,deadlock` 标志的 `-collect` 命令来运行应用程序。实验将同时包含争用检测数据和死锁检测数据。

线程分析器界面

可以使用 `tha` 命令启动线程分析器。

线程分析器界面是经过简化的性能分析器 (`analyzer`) 界面，适合进行多线程程序分析。现在不再显示常用的性能分析器视图，而是显示 "Races"（争用）、"Deadlocks"（死锁）和 "Dual Source"（双源）等视图。如果使用性能分析器查看多线程程序实验，将看到传统的性能分析器视图（如 "Functions"（函数）、"Callers-Callees"（调用方-被调用方）、"Disassembly"（反汇编））以及关于数据争用和死锁的视图。

♦ ♦ ♦ 第 2 章

2 数据争用教程

本教程详细介绍如何使用线程分析器检测和修复数据争用。

本教程分为以下几节：

数据争用教程源文件

本教程所依赖的两个程序都存在数据争用：

- 第一个程序查找质数。该程序是用 C 编写的，并使用 OpenMP 指令进行了并行化。该源文件称为 `prime_omp.c`。
- 第二个程序也查找质数，并且也是用 C 编写的。但是，它使用 POSIX 线程（而不是 OpenMP 指令）进行并行化。该源文件称为 `prime_pthr.c`。

获取数据争用教程源文件

您可以从 Oracle Developer Studio 开发者门户的下载区域 (`http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/index.html`)中下载本教程所用的源文件。

在下载并解压缩样例文件之后，您可以在 `OracleDeveloperStudio12.5-Samples/ThreadAnalyzer` 目录中找到样例。这些样例位于 `prime_omp` 和 `prime_pthr` 子目录中。每个样例目录都包含 `Makefile` 和 `DEMO` 说明文件，但是本教程并不遵循这些说明，也不使用 `Makefile`。本教程将逐步指导您执行命令。

为跟随本教程学习，您可以将 prime_omp.c 和 prime_pthr.c 文件从样例目录复制到其他目录，也可以创建自己的文件并从下面列出的代码内容中复制代码。

prime_omp.c 的源代码

本节列出了 prime_omp.c 的源代码，如下所示：

```
 1 /*
 2  * Copyright (c) 2006, 2011, Oracle and/or its affiliates. All Rights Reserved.
 3  */
 4
 5 #include <stdio.h>
 6 #include <math.h>
 7 #include <omp.h>
 8
 9 #define THREADS 4
10 #define N 10000
11
12 int primes[N];
13 int pflag[N];
14
15 int is_prime(int v)
16 {
17     int i;
18     int bound = floor(sqrt(v)) + 1;
19
20     for (i = 2; i < bound; i++) {
21         /* no need to check against known composites */
22         if (!pflag[i])
23             continue;
24         if (v % i == 0) {
25             pflag[v] = 0;
26             return 0;
27         }
28     }
29     return (v > 1);
30 }
31
32 int main(int argn, char **argv)
33 {
34     int i;
35     int total = 0;
36
37 #ifdef _OPENMP
38     omp_set_dynamic(0);
39     omp_set_num_threads(THREADS);
40 #endif
41
42     for (i = 0; i < N; i++) {
43         pflag[i] = 1;
44     }
```

```
45
46      #pragma omp parallel for
47      for (i = 2; i < N; i++) {
48          if ( is_prime(i) ) {
49              primes[total] = i;
50              total++;
51          }
52      }
53
54      printf("Number of prime numbers between 2 and %d: %d\n",
55             N, total);
56
57      return 0;
58  }
```

prime_pthr.c 的源代码

本节列出了 prime_pthr.c 的源代码，如下所示：

```
 1  /*
 2   * Copyright (c) 2006, 2011, Oracle and/or its affiliates. All Rights Reserved.
 3   */
 4
 5  #include <stdio.h>
 6  #include <math.h>
 7  #include <pthread.h>
 8
 9  #define THREADS 4
10  #define N 10000
11
12  int primes[N];
13  int pflag[N];
14  int total = 0;
15
16  int is_prime(int v)
17  {
18      int i;
19      int bound = floor(sqrt(v)) + 1;
20
21      for (i = 2; i < bound; i++) {
22          /* no need to check against known composites */
23          if (!pflag[i])
24              continue;
25          if (v % i == 0) {
26              pflag[v] = 0;
27              return 0;
28          }
29      }
30      return (v > 1);
31  }
32
33  void * work(void *arg)
```

```
34 {
35     int start;
36     int end;
37     int i;
38
39     start = (N/THREADS) * (*(int *)arg);
40     end = start + N/THREADS;
41     for (i = start; i < end; i++) {
42         if ( is_prime(i) ) {
43             primes[total] = i;
44             total++;
45         }
46     }
47     return NULL;
48 }
49
50 int main(int argn, char **argv)
51 {
52     int i;
53     pthread_t tids[THREADS-1];
54
55     for (i = 0; i < N; i++) {
56         pflag[i] = 1;
57     }
58
59     for (i = 0; i < THREADS-1; i++) {
60         pthread_create(&tids[i], NULL, work, (void *)&i);
61     }
62
63     i = THREADS-1;
64     work((void *)&i);
65
66     for (i = 0; i < THREADS-1; i++) {
67         pthread_join(tids[i], NULL);
68     }
69
70     printf("Number of prime numbers between 2 and %d: %d\n",
71            N, total);
72
73     return 0;
74 }
```

数据争用在 **prime_omp.c** 和 **prime_pthr.c** 中的效果

当代码包含争用情况时，内存访问的顺序是不确定的，因此每次运行的计算结果会不同。`prime_omp` 和 `prime_pthr` 程序中的正确答案为 1229。

通过编译并运行示例可以看出，由于代码中存在数据争用，执行 `prime_omp` 或 `prime_pthr` 时都会产生不正确且不一致的结果。

在下面的示例中，在提示符下键入命令以编译并运行 `prime_omp` 程序：

```
% cc -xopenmp=noopt -o prime_omp prime_omp.c -lm
%
% ./prime_omp
Number of prime numbers between 2 and 10000: 1229
% ./prime_omp
Number of prime numbers between 2 and 10000: 1228
% ./prime_omp
Number of prime numbers between 2 and 10000: 1180
```

在下面的示例中，在提示符处键入命令以编译并运行 `prime_pthr` 程序：

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

请注意每个程序的三次运行结果的不一致性。可能需要运行这些程序三次以上才能看到不一致的结果。

接下来将会检测代码并创建实验，以便可以找出发生数据争用的位置。

如何使用线程分析器找到数据争用

线程分析器沿用与 Oracle Developer Studio 性能分析器相同的“收集-分析”模型。

使用线程分析器的过程涉及三个步骤：

1. “检测代码” [21]
2. “创建数据争用检测实验” [22]
3. “检查数据争用检测实验” [23]

检测代码

为了在程序中检测数据争用，必须首先对代码进行检测以监视运行时的内存访问。检测可以在应用程序源代码或应用程序二进制文件上执行。本教程将介绍如何使用这两种检测程序的方法。

检测源代码

要检测源代码，必须使用特殊的编译器选项 `-xinstrument=datarace` 对应用程序进行编译。此选项会指示编译器对生成的代码进行检测，以便检测数据争用。

将 -xinstrument=datarace 编译器选项添加到用于编译程序的现有选项集中。

注 - 使用 -xinstrument=datarace 对程序进行编译时，务必还要指定 -g 选项，目的是生成其他信息以启用线程分析器的全部功能。对程序进行编译以检测数据争用时，不要指定高优化级别。请使用 -xopenmp=noopt 编译 OpenMP 程序。使用高优化级别时，报告的信息（如行号和调用堆栈）可能是错误的。

可以使用以下命令检测本教程的源代码：

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

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

请注意，本示例在结尾处使用了 _inst 指定输出文件，因此可以知道该二进制代码是检测后的二进制代码。不过，这不是必需的。

检测二进制代码

要检测程序的二进制代码而非源代码，需要使用 Oracle Developer Studio 中包含的 discover 工具，discover(1) 手册页和《Oracle Developer Studio 12.5：Discover 和 Uncover 用户指南》中对该工具进行了说明。

对于本教程示例，请键入以下命令编译代码：

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

% cc -g -O2 -o prime_pthr prime_pthr.c -lm
```

然后，对所创建的 prime_omp 和 prime_pthr 优化二进制代码运行 discover：

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

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

这些命令将创建检测后的二进制代码 prime_omp_disc 和 prime_pthr_disc，可以将这些二进制代码与 collect 一起使用，以创建可使用线程分析器进行检查的实验。

创建数据争用检测实验

使用带有 -r race 标志的 collect 命令来运行程序并在进程执行期间创建数据争用检测实验。对于 OpenMP 程序，确保使用的线程数多于一个。在本教程的样例中，使用了四个线程。

基于通过检测源代码创建的二进制代码创建实验：

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

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

基于通过使用 `discover` 工具创建的二进制代码创建实验：

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

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

为增大检测到数据争用的可能性，建议使用带有 `-r race` 标志的 `collect` 创建多个数据争用检测实验。对于每个实验，应使用不同的线程数和不同的输入数据。

例如，在 `prime_omp.c` 中，由以下行设置线程数：

```
#define THREADS 4
```

可通过将上面的 4 更改为大于 1 的其他某个整数（例如 8）来更改线程数。

`prime_omp.c` 中的以下行会将程序限制为查找 2 和 3000 之间的质数：

```
#define N 3000
```

可通过更改 `N` 的值提供不同的输入数据，从而使程序执行更多或更少的工作。

检查数据争用检测实验

可以使用线程分析器、性能分析器或 `er_print` 实用程序检查数据争用检测实验。线程分析器和性能分析器都提供 GUI 界面；线程分析器显示的是一组简化的缺省视图，但在其他方面与性能分析器完全相同。

使用线程分析器查看数据争用实验

要启动线程分析器，请键入以下命令：

```
% tha
```

第一次启动线程分析器时，将显示 "Welcome"（欢迎）屏幕。

线程分析器在左侧具有菜单栏、工具栏和垂直导航栏，因而您可以选择数据视图。

缺省情况下显示以下数据视图：

- "Overview"（概述）屏幕显示已装入实验的度量概述。
- "Races"（争用）视图显示程序中检测到的数据争用列表以及关联的调用堆栈跟踪。选择 "Races"（争用）视图中的项时，"Race Details"（争用详细信息）窗口会显示有关数据争用或所选调用堆栈跟踪的详细信息。

- "Dual Source"（双源）视图，显示与所选数据争用的两次访问相对应的两个源代码位置。发生数据争用访问的源代码行会突出显示。
- "Experiments"（实验）视图显示实验中的装入对象并列出错误和警告消息。

可以选择使用 "More Views"（更多视图）选项菜单查看其他视图。

使用 `er_print` 查看数据争用实验

`er_print` 实用程序提供命令行界面。可以在交互式会话中使用 `er_print` 实用程序并在该会话期间指定子命令。也可以使用命令行选项以非交互方式指定子命令。

使用 `er_print` 实用程序检查争用时，以下子命令非常有用：

- `-races`

  该选项会报告在实验中发现的所有数据争用。在 `(er_print)` 提示符下指定 `-races`，或者在 `er_print` 命令行上指定 `races`。
- `-rdetail` *race_id*

  该选项会显示具有指定 *race_id* 的数据争用的详细信息。在 `(er_print)` 提示符下指定 `-rdetail`，或者在 `er_print` 命令行上指定 `rdetail`。如果指定的 *race_id* 为 **`all`**，将显示所有数据争用的详细信息。否则，请指定单个争用编号，例如为第一个数据争用指定 **`1`**。
- `-header`

  该选项会显示有关实验的描述性信息并报告所有错误或警告。在 `(er_print)` 提示符下指定 `header`，或者在命令行上指定 `-header`。

有关更多信息，请参阅 `collect`(1)、`tha`(1)、`analyzer`(1) 和 `er_print`(1) 手册页。

了解实验结果

本节介绍如何使用 `er_print` 命令行和线程分析器显示检测到的每次数据争用的以下信息：

- 数据争用的唯一 ID。
- 与数据争用相关联的虚拟地址 `Vaddr`。如果不止一个虚拟地址，则会在括号中显示标签 `Multiple Addresses`（多个地址）。
- 两个不同线程对虚拟地址 `Vaddr` 的内存访问。将会显示访问的类型（读取或写入），以及源代码中发生访问的函数、偏移量和行号。
- 与数据争用相关联的调用堆栈跟踪总数。每个跟踪都引用发生两个数据争用访问时的一对线程调用堆栈。

如果使用的是线程分析器，则在 "Races"（争用）视图中选择单个调用堆栈跟踪时会在 "Race Details"（争用详细信息）窗口中显示这两个调用堆栈。如果使用的是 `er_print` 实用程序，`rdetail` 命令将会显示这两个调用堆栈。

`prime_omp.c` 中的数据争用

要检查 `prime_omp.c` 中的数据争用，可以使用在“创建数据争用检测实验” [22]中创建的实验之一。

要使用 `er_print` 显示 `prime_omp_instr.er` 实验中的数据争用信息，请键入以下命令。

```
% er_print prime_omp_inst.er
```

在 `(er_print)` 提示符下，键入 **`races`** 可看到类似如下的输出：

```
(er_print) races

Total Races:  4 Experiment:  prime_omp_inst.er

Race #1, Vaddr: 0x21850
         Access 1: Write, line 25 in "prime_omp.c",
                   is_prime
         Access 2: Read, line 22 in "prime_omp.c",
                   is_prime
 Total Callstack Traces: 1

Race #2, Vaddr: (Multiple Addresses)
         Access 1: Write, line 49 in "prime_omp.c",
                   main
         Access 2: Write, line 49 in "prime_omp.c",
                   main
 Total Callstack Traces: 1

Race #3, Vaddr: 0xffbff534
         Access 1: Write, line 50 in "prime_omp.c",
                   main
         Access 2: Read, line 49 in "prime_omp.c",
                   main
 Total Callstack Traces: 1

Race #4, Vaddr: 0xffbff534
         Access 1: Write, line 50 in "prime_omp.c",
                   main
         Access 2: Write, line 50 in "prime_omp.c",
                   main
Total Callstack Traces: 1
(er_print)
```

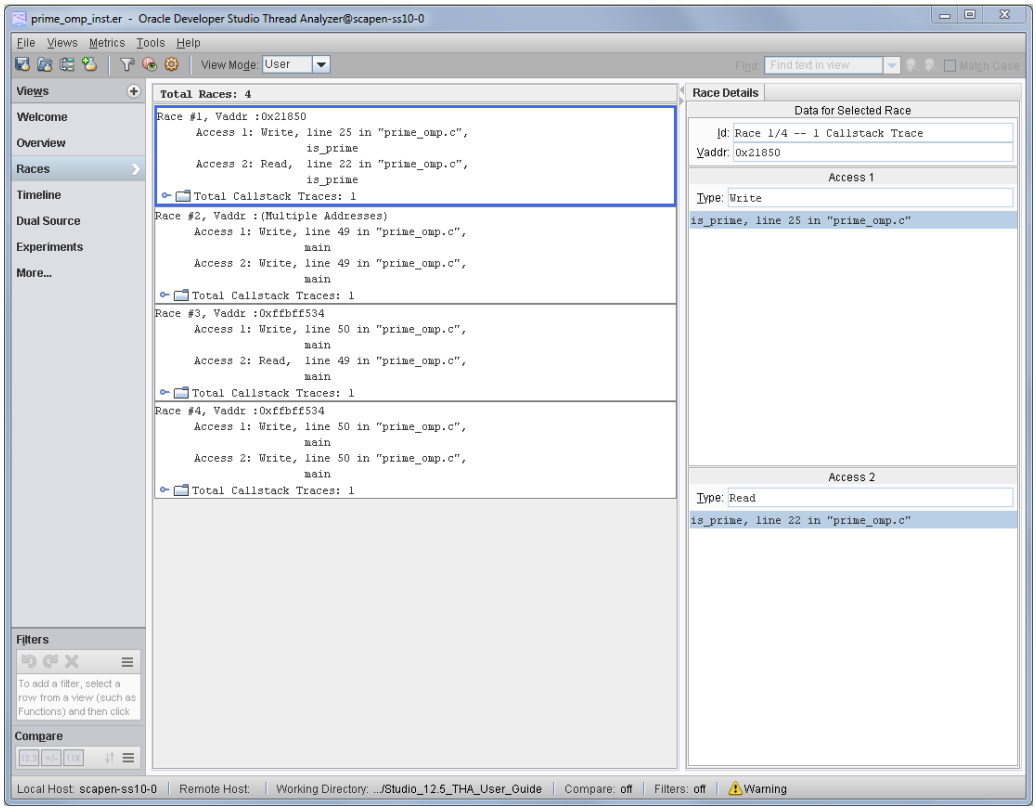
该程序的此次特定运行期间发生了四次数据争用。

要在线程分析器中打开 prime_omp_inst.er 实验，请键入以下命令：

```
% tha prime_omp_inst.er
```

以下屏幕抓图显示了在 prime_omp.c 中检测到的争用，如线程分析器所显示。

图 1 prime_omp.c 中检测到的数据争用



prime_omp.c 中显示了四次数据争用：

- Race #1（争用 1）显示了第 25 行函数 is_prime() 中的一次写入与第 22 行同一函数中的一次读取之间的争用。如果查看源代码，可以看到在这些行上，正在对 pflag[ ] 数组进行访问。在线程分析器中，通过单击 "Dual Source"（双源）视图可以很方便地查看位于两个行号位置的源代码，其中还有一些度量显示受影响的代码行上的争用访问次数。
- Race #2（争用 2）显示了对 main 函数第 49 行的两次写入之间的争用。单击 "Dual Source"（双源）视图可看到，多次尝试访问第 49 行上 primes [ ] 数组的元素。"

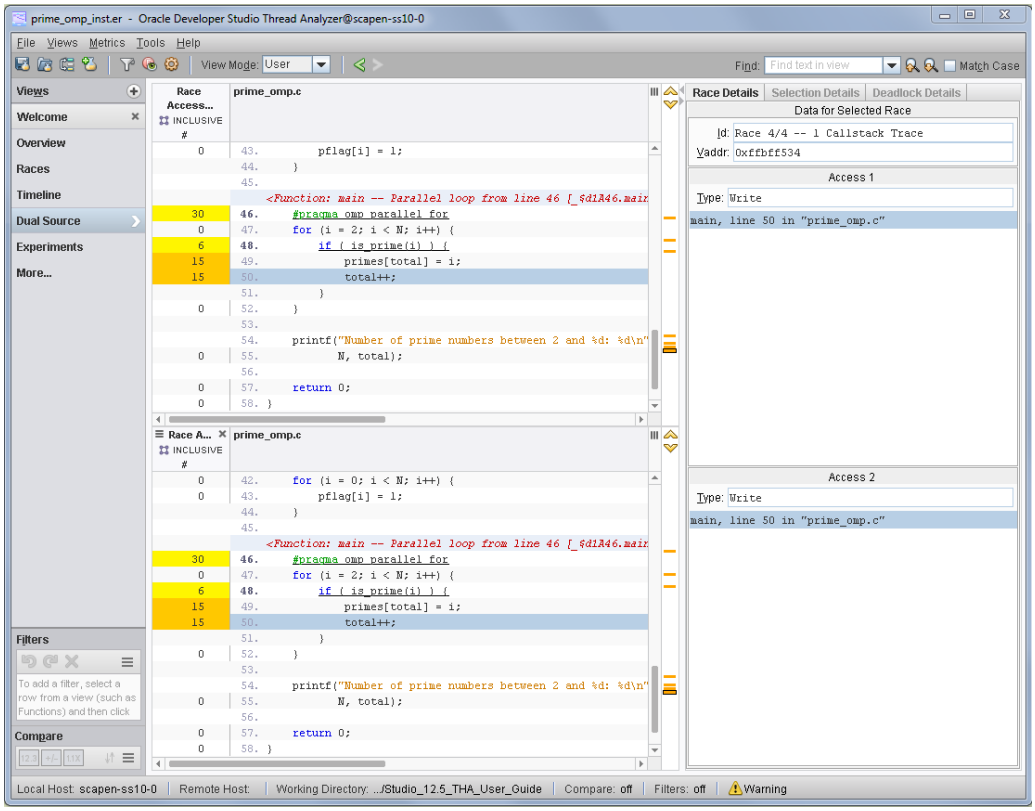
Race #2"（争用 2）表示在数组 `primes[ ]` 的不同元素中发生的一组数据争用。通过 Vaddr 显示 Multiple Addresses（多个地址）可以看出这一点。

- `Race #3`（争用 3）显示了第 50 行 main 函数中的一次写入与第 49 行同一函数中的一次读取之间的争用。如果查看源代码，可以看到在这些行上，正在对可变 total 进行访问。
- `Race #4`（争用 4）显示了对 main 函数第 50 行的两次写入之间的争用。如果查看源代码，可以看到在这些行上，正在对可变 total 进行更新。

  使用线程分析器中的 "Dual Source"（双源）视图，可以同时看到与数据争用相关联的两个源代码位置。例如，在 "Races"（争用）视图中为 `prime_omp.c` 选择 `Race #1`（争用 1），然后单击 "Dual Source"（双源）视图。将会看到类似如下的内容。

使用线程分析器中的 "Dual Source"（双源）视图，可以同时看到与数据争用相关联的两个源代码位置。例如，在 "Races"（争用）视图中为 `prime_pthr.c` 选择 `Race #3`（争用 3），然后单击 "Dual Source"（双源）视图。将会看到类似如下的内容。

图 2 `prime_omp.c` 中检测到的数据争用的源代码



prime_pthr.c 中的数据争用

要检查 prime_pthr.c 中的数据争用，可以使用在“创建数据争用检测实验” [22]中创建的实验之一。

要使用 er_print 显示 prime_pthr_instr.er 实验中的数据争用信息，请键入以下命令：

```
% er_print prime_pthr_inst.er
```

在 (er_print) 提示符下，键入 **races** 可看到类似如下的输出：

```
(er_print) races

Total Races:  5 Experiment:  prime_pthr_inst.er

Race #1, Vaddr: (Multiple Addresses)
Access 1: Write, line 26 in "prime_pthr.c",
          is_prime + 0x00000234
Access 2: Write, line 26 in "prime_pthr.c",
          is_prime + 0x00000234
Total Callstack Traces: 2

Race #2, Vaddr: 0xffbff6dc
Access 1: Write, line 59 in "prime_pthr.c",
          main + 0x00000208
Access 2: Read, line 39 in "prime_pthr.c",
          work + 0x00000070
Total Callstack Traces: 1

Race #3, Vaddr: 0x21620
Access 1: Write, line 44 in "prime_pthr.c",
          work + 0x000001C0
Access 2: Read, line 43 in "prime_pthr.c",
          work + 0x0000011C
Total Callstack Traces: 2

Race #4, Vaddr: 0x21620
Access 1: Write, line 44 in "prime_pthr.c",
          work + 0x000001C0
Access 2: Write, line 44 in "prime_pthr.c",
          work + 0x000001C0
Total Callstack Traces: 2

Race #5, Vaddr: (Multiple Addresses)
Access 1: Write, line 43 in "prime_pthr.c",
          work + 0x00000174
Access 2: Write, line 43 in "prime_pthr.c",
          work + 0x00000174
Total Callstack Traces: 2
(er_print)
```

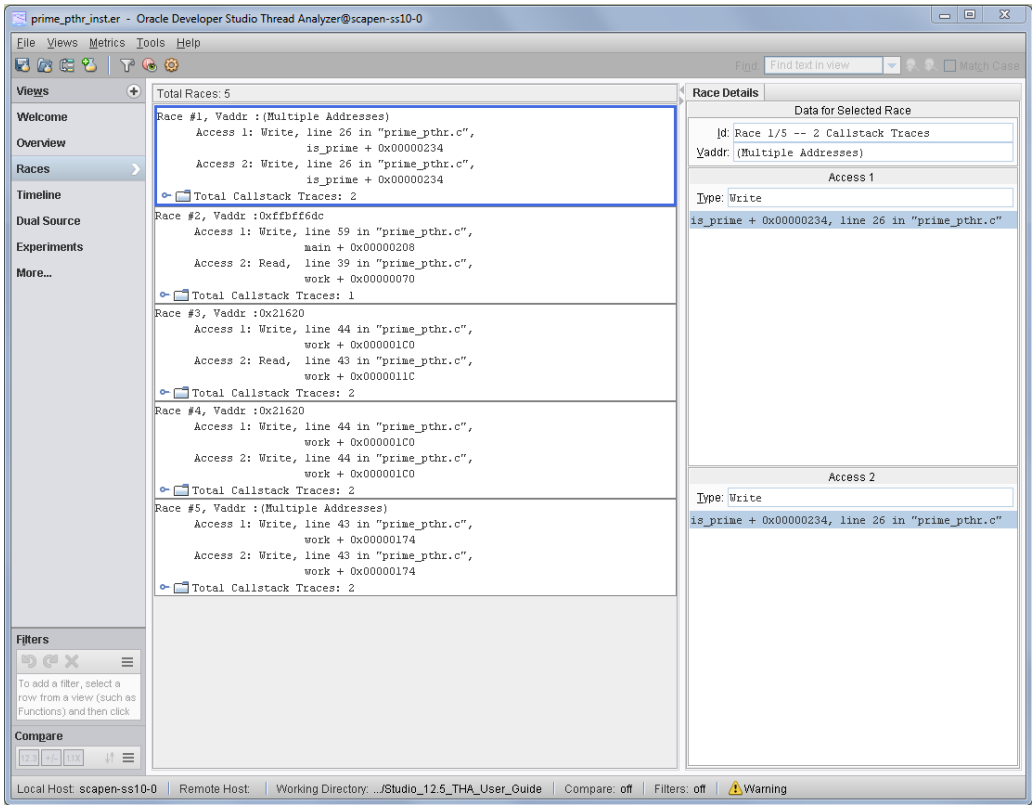
该程序的此次特定运行期间发生了五次数据争用。

要在线程分析器中打开 prime_pthr_inst.er 实验，请键入以下命令：

```
% tha prime_pthr_inst.er
```

以下屏幕抓图显示了在 prime_pthr.c 中检测到的争用，如线程分析器所显示。请注意，这些争用与 er_print 所显示的争用相同。

图 3 prime_pthr.c 中检测到的数据争用



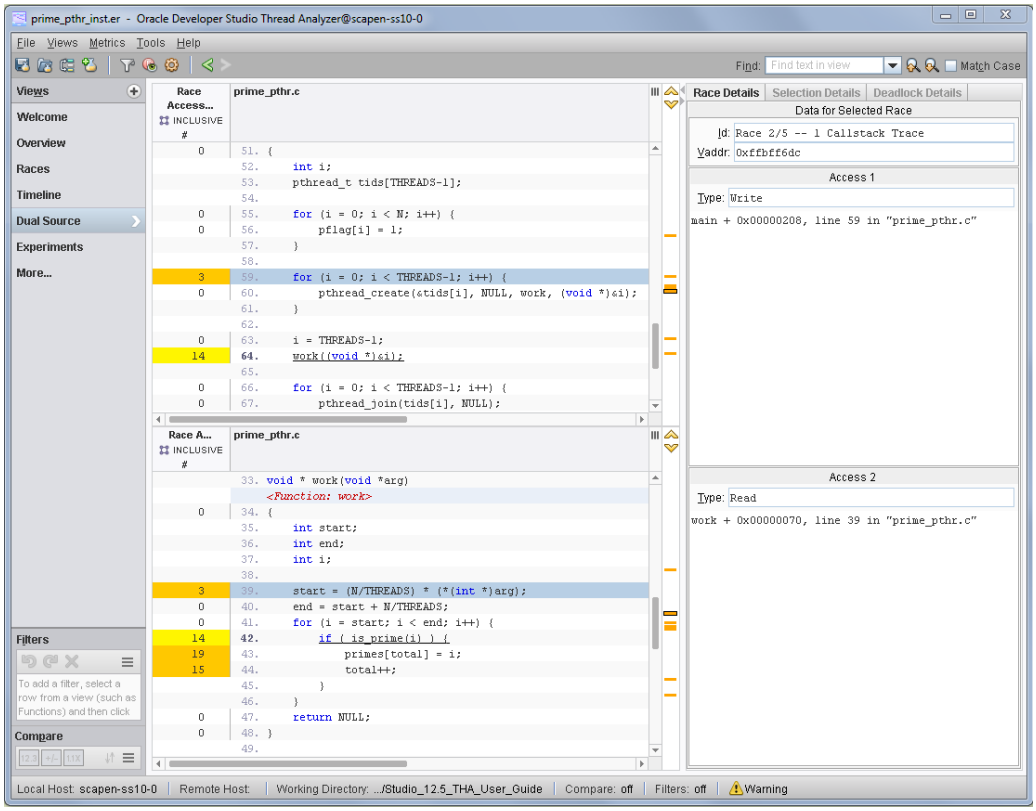
prime_pthr.c 中显示了五次数据争用：

- Race #1（争用 1）是第 26 行上函数 is_prime（） 中 pflag[ ] 数组元素的两次写入之间的数据争用。
- Race #2（争用 2）是第 59 行中对 main（） 中内存位置 i 的一次写入与第 39 行中对同一内存位置（在 work（） 中名为 *arg）的一次读取之间的数据争用。
- Race #3（争用 3）是第 44 行中对 total 的一次写入与第 43 行中对 total 的一次读取之间的数据争用。

- Race #4（争用 4）是第 44 行中对 total 的一次写入与同一行中对 total 的另一次写入之间的数据争用。
- Race #5（争用 5）是 main() 函数第 43 行的两次写入之间的数据争用。Race #5（争用 3）表示数组 primes[ ] 的不同元素中所发生的一组数据争用。通过 Vaddr 显示 Multiple Addresses（多个地址）可以看出这一点。

如果选择 Race #3（争用 3），然后单击 "Dual Source"（双源）视图，则会看到两个源代码位置，类似于以下屏幕抓图。

图 4 数据争用的源代码详细信息

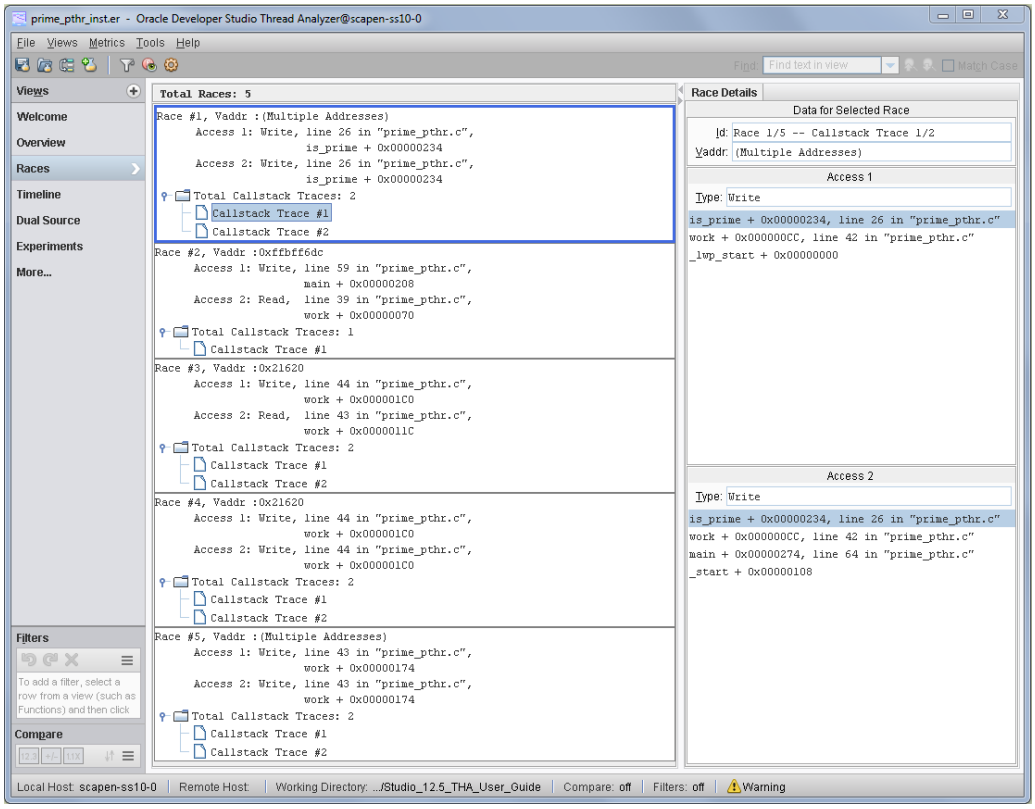


Race #2（争用 2）的第一次访问位于第 59 行，显示在顶部面板中。第二次访问位于第 39 行，显示在底部面板中。源代码的左侧会突出显示 "Race Accesses"（争用访问）度量。此度量显示了该行上报告的数据争用访问总次数。

数据争用的调用堆栈跟踪

线程分析器的 "Races"（争用）视图中列出的每次数据争用还有一个或多个相关联的调用堆栈跟踪。这些调用堆栈显示了代码中导致数据争用的执行路径。单击调用堆栈跟踪时，右侧面板中的 "Race Details"（争用详细信息）窗口会显示导致数据争用的函数调用。

图 5 "Races"（争用）视图（包含 `prime_omp.c` 的调用堆栈跟踪）



诊断数据争用的原因

本节介绍诊断数据争用原因的基本策略。

检查数据争用是否为误报

误报数据争用是指线程分析器已报告但实际并未发生的数据争用。换言之，这是“假警报”。线程分析器会尽可能减少误报数量，但在某些情况下，该工具无法完成精确的工作并可能误报数据争用。

由于误报的数据争用不是真正的数据争用，并不会影响程序的行为，因此可将其忽略。

有关误报数据争用的一些示例，请参见“误报” [35]。有关如何从报告中消除误报数据争用的信息，请参见“线程分析器用户 API” [69]。

检查数据争用是否为良性

良性数据争用是指特意允许的数据争用，其存在不会影响程序正确性。

一些多线程应用程序会特意使用可能导致数据争用的代码。由于这些数据争用是专门设计的，所以不需要进行修复。但在某些情况下，要使这样的代码正确运行会相当棘手。应仔细检查这些数据争用。

有关良性争用的更多详细信息，请参见“误报” [35]。

修复错误而不是修复数据争用

线程分析器可以帮助查找程序中的数据争用，但是它无法自动查找程序中的错误，也不会建议如何修复找到的数据争用。数据争用可能是由某个错误引入的。这种情况下，必须找到并修复该错误。仅仅消除数据争用并不是正确的做法，这样做可能会使进一步的调试更加困难。

修复 `prime_omp.c` 中的错误

本节介绍如何修复 `prime_omp.c` 中的错误。有关所列的完整文件内容，请参见“`prime_omp.c` 的源代码” [18]。

将第 49 行和第 50 行移动到 `critical`（临界）段中，以便消除数组 `primes[ ]` 的元素上的数据争用。

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

```
52      }
```

也可以按照如下所示将第 49 行和第 50 行移动到两个 critical（临界）段中，但此更改方式无法更正程序：

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

包含第 49 行和第 50 行的临界段将会消除数据争用，因为线程会使用互斥锁控制它们对 primes[ ] 数组的访问。但是，程序仍是错误的。两个线程可能会使用同一 total 值更新 primes[ ] 的同一元素，而 primes[ ] 的某些元素可能根本不会被赋值。

第二个数据争用（第 22 行中从 pflag[ ] 的读取与第 25 行中对 pflag[ ] 的写入之间的数据争用）实际上是良性争用，因为它并不会导致错误结果。没有必要修复良性数据争用。

修复 prime_pthr.c 中的错误

本节介绍如何修复 prime_pthr.c 中的错误。有关所列的完整文件内容，请参见“prime_pthr.c 的源代码” [19]。

要删除第 50 行中对 prime[ ] 的数据争用以及第 44 行中对 total 的数据争用，请围绕这两行添加互斥锁/解锁，以便每次只有一个线程可以更新 prime[ ] 和 total。

第 59 行中对 i 的写入与第 39 行中同一内存位置（名为 *arg）的读取之间的数据争用表明不同的线程对变量 i 进行的共享访问存在问题。prime_pthr.c 中的初始线程在第 59-61 行通过循环方式创建子线程，并调度它们执行函数 work（）。循环索引 i 按地址传递到 work（）。由于所有线程都访问 i 的同一内存位置，因此每个线程的 i 值不会保持唯一，而会随着初始线程对循环索引的递增而改变。由于不同的线程使用同一个 i 值，因此就会发生数据争用。修复该问题的一种方法是按值（而不是按地址）将 i 传递给 work（）。

以下代码是 prime_pthr.c 的更正版本：

```
1 /*
2  * Copyright (c) 2006, 2011, Oracle and/or its affiliates. All Rights Reserved.
3  */
4
5 #include <stdio.h>
```

```
 6 #include <math.h>
 7 #include <pthread.h>
 8
 9 #define THREADS 4
10 #define N 10000
11
12 int primes[N];
13 int pflag[N];
14 int total = 0;
15 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
16
17 int is_prime(int v)
18 {
19     int i;
20     int bound = floor(sqrt(v)) + 1;
21
22     for (i = 2; i < bound; i++) {
23         /* no need to check against known composites */
24         if (!pflag[i])
25             continue;
26         if (v % i == 0) {
27             pflag[v] = 0;
28             return 0;
29         }
30     }
31     return (v > 1);
32 }
33
34 void * work(void *arg)
35 {
36     int start;
37     int end;
38     int i;
39
40     start = (N/THREADS) * ((int)arg) ;
41     end = start + N/THREADS;
42     for (i = start; i < end; i++) {
43         if ( is_prime(i) ) {
44             pthread_mutex_lock(&mutex);
45             primes[total] = i;
46             total++;
47             pthread_mutex_unlock(&mutex);
48         }
49     }
50     return NULL;
51 }
52
53 int main(int argn, char **argv)
54 {
55     int i;
56     pthread_t tids[THREADS-1];
57
58     for (i = 0; i < N; i++) {
59         pflag[i] = 1;
```

```
60      }
61
62      for (i = 0; i < THREADS-1; i++) {
63          pthread_create(&tids[i], NULL, work, (void *)i);
64      }
65
66      i = THREADS-1;
67      work((void *)i);
68
69      for (i = 0; i < THREADS-1; i++) {
70          pthread_join(tids[i], NULL);
71      }
72
73      printf("Number of prime numbers between 2 and %d: %d\n",
74             N, total);
75
76      return 0;
77  }
```

误报

有时，线程分析器可能会报告实际在程序中并未发生的数据争用。我们将这些称为误报。大多数情况下，误报是由用户定义的同步导致的，或由不同线程回收的内存导致的。有关更多信息，请参见“用户定义的同步” [35]和“由不同线程回收的内存” [36]。

用户定义的同步

线程分析器可以识别由 OpenMP、POSIX 线程和 Oracle Solaris 线程提供的大多数标准同步 API 和构造。但是，该工具无法识别用户定义的同步，如果代码中包含这样的同步，可能会误报数据争用。

注 - 为了避免报告此类误报的数据争用，线程分析器提供了一组 API，可用于在执行用户定义的同步时通知该工具。有关更多信息，请参见“线程分析器用户 API” [69]。

为了说明为何需要使用这些 API，请考虑以下内容。线程分析器无法使用 CAS 指令来识别锁的实现，也无法使用忙等待来识别发布和等待操作，等等。下面是一类误报的典型示例，其中程序利用的是 POSIX 线程条件变量的常见用法：

```
/* 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;
```

pthread_cond_wait（） 调用通常在循环内进行，该循环的作用是对谓词进行测试以防止发生程序错误和虚假唤醒。谓词的测试和设置通常由互斥锁加以保护。在以上代码中，线程 1 在第 100 行生成变量 data 的值，在第 102 行将 ready_flag 的值设置为 1 以表明已生成该数据，然后调用 pthread_cond_signal（） 以唤醒使用者线程（即线程 2）。线程 2 在循环内测试谓词 (!ready_flag)。当它发现已设置该标志时，将在第 205 行使用数据。

第 102 行 ready_flag 的写入和第 201 行 ready_flag 的读取由同一互斥锁加以保护，因此这两次访问之间不存在数据争用，并且该工具可以正确识别该情况。

第 100 行 data 的写入和第 205 行 data 的读取不受互斥锁的保护。但是，在程序逻辑中，第 205 行的读取总是发生在第 100 行的写入之后，原因是存在标志变量 ready_flag。因此，这两次数据访问之间不存在数据争用。但是，如果在运行时未实际对 pthread_cond_wait（） 进行调用（第 202 行），该工具将报告这两次访问之间存在数据争用。如果在执行第 201 行之前执行了第 102 行，则在执行第 201 行时，循环条目测试就会失败，因此将跳过第 202 行。该工具会监视 pthread_cond_signal（） 调用和 pthread_cond_wait（） 调用，并可使它们成对以形成同步。如果未调用第 202 行的 pthread_cond_wait（），该工具并不知道第 100 行的写入总是在第 205 行的读取之前执行。因此，该工具认为它们是同时执行的，并报告它们之间的数据争用。

libtha(3C) 手册页和“线程分析器用户 API” [69]说明了如何使用 API 避免报告此类误报的数据争用。

由不同线程回收的内存

有些内存管理例程负责回收由一个线程释放的内存，以供另一个线程使用。由不同线程使用的同一内存位置的使用期限不会重叠，线程分析器有时无法认识这一点。发生这种情况时，该工具可能会误报数据争用。以下示例将说明此类误报。

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

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

线程 1 和线程 2 同时执行。每个线程都会分配一个用作专用内存的内存块。mymalloc（） 例程可能会提供先前由 myfree（） 调用所释放的内存。如果线程 2 在线程 1 调用 myfree（） 之前调用 mymalloc（），则 ptr1 和 ptr2 会获得不同的值，并且在这两个线程之间不存在数据争用。但是，如果线程 2 在线程 1 调用 myfree（） 之后调用 mymalloc（），则 ptr1 和 ptr2 可能会具有相同的值。由于线程 1 不再访问该内存，所以不存在数据争用。但是，如果该工具不知道 mymalloc（） 正在回收内存，它会报告 ptr1 数据的写入与 ptr2 数据的写入之间存在数据争用。在 C++ 应用程序中，当 C++ 运行时库回收临时变量的内存时，经常发生此类误报。在实现自己的内存管理例程的用户应用程序中，也经常发生此类误报。目前，线程分析器能够识别通过标准 malloc（）、calloc（） 和 realloc（） 接口执行的内存分配和释放操作。

良性数据争用

为了获得更好的性能，某些多线程应用程序会特意允许数据争用。良性数据争用是指特意允许的数据争用，其存在不会影响程序正确性。以下示例将说明良性数据争用。

注 - 除了良性数据争用之外，有很大一类的应用程序允许数据争用，因为它们依赖于锁无关 (lock-free) 和等待无关 (wait-free) 算法，不过很难正确设计这些算法。线程分析器可以帮助确定这些应用程序中的数据争用位置。

用于查找质数的程序

prime_omp.c 中的线程通过执行函数 is_prime（） 来检查某个整数是否为质数。

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

```
28      }
29      return (v > 1);
30  }
```

线程分析器会报告第 25 行中对 pflag[ ] 的写入与第 22 行中对 pflag[ ] 的读取之间存在数据争用。但是，此数据争用是良性的，因为它不会影响最终结果的正确性。在第 22 行，线程将检查对于给定的 i 值，pflag[i] 是否等于 0。如果 pflag[i] 等于 0，则表明 i 是已知的合数（换句话说，知道 i 不是质数）。因此，无需检查 v 是否可以被 i 整除；只需检查 v 是否可以被某个质数整除。因此，如果 pflag[i] 等于 0，该线程会继续处理 i 的下一个值。如果 pflag[i] 不等于 0 并且 v 可以被 i 整除，该线程会将 0 分配给 pflag[v] 以表明 v 不是质数。

从正确性方面来说，多个线程检查同一 pflag[ ] 元素并同时向其写入是可以的。pflag[ ] 元素的初始值为 1。当线程更新该元素时，它们会为该元素分配值 0。也就是说，这些线程会在该元素的同一内存字节的同一位中存储 0。在当前的体系结构中，可以放心地认为这些存储是原子操作。这意味着，某个线程读取该元素时，读取的值要么为 1，要么为 0。如果某个线程在给定的 pflag[ ] 元素（第 22 行）被分配值 0 之前对该元素进行检查，则该线程会执行第 24–26 行。在此期间，如果另一个线程也将 0 分配给同一 pflag[ ] 元素（第 25 行），最终结果不变。在本质上，这意味着第一个线程不必要地执行了第 24–26 行，但最终结果是相同的。

用于检验数组值类型的程序

一组线程同时调用 check_bad_array() 以检查数组 data_array 中是否存在任何“错误”元素。每个线程检查数组的不同部分。如果线程发现某个元素有错误，它会将全局共享变量 is_bad 的值设置为真。

```
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  }
```

第 108 行中对 is_bad 的读取与第 112 行中对 is_bad 的写入之间存在数据争用。但是，该数据争用不影响最终结果的正确性。

is_bad 的初始值为 0。当线程更新 is_bad 时，它们会为其分配值 1。也就是说，这些线程会在 is_bad 的同一内存字节的同一位中存储 1。在当前的体系结构中，可以放心地认为这些存储是原子操作。因此，某个线程读取 is_bad 时，读取的值要么为 0，要么为 1。如果某个线程在 is_bad（第 108 行）被分配值 1 之前对其进行检查，则该线程会继续执行 for 循环。在此期间，如果另一个线程也将值 1 分配给 is_bad（第 112 行），这并不会改变最终结果。这仅仅意味着该线程执行 for 循环的时间超出了必要时间。

使用双检锁的程序

单件可确保在整个程序中只有一个特定类型的对象存在。双检锁是在多线程应用程序中对单件进行初始化的一种常见的有效方法。以下代码将说明这样的实现。

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

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

ptr_instance 的读取（第 202 行）特意未用锁进行保护。这样就可以通过检查来确定 Singleton（单件）是否已经在多线程环境中进行有效实例化。请注意，第 202 行的读取与第 212 行的写入之间存在对变量 ptr_instance 的数据争用，但程序可正常工作。不过，编写允许数据争用的正确程序时，需要格外小心。例如，在以上双检锁定代码

中，第 209 行中对 `memory_barrier（）` 的调用用来确保线程不会看到 `ptr_instance` 为非空，直至构造 `Singleton`（单件）的所有写入已完成。

♦ ♦ ♦ 第 3 章

3 死锁教程

本教程介绍如何使用线程分析器在多线程程序中检测潜在的死锁和实际的死锁。

本教程涵盖以下主题：

关于死锁

术语死锁描述的是两个或多个线程由于相互等待而永远被阻塞的情况。导致死锁的原因有很多，例如程序逻辑错误以及不恰当使用同步（如锁和屏障）。本教程重点介绍由于不恰当使用互斥锁而导致的死锁。此类死锁通常发生在多线程应用程序中。

以下三个条件成立时，包含两个或多个线程的进程可能会进入死锁状态：

- 已持有锁的线程请求新锁
- 同时发出对新锁的请求
- 两个或多个线程形成一个循环链，其中每个线程都在等待链中下一个线程持有的锁

以下是一个死锁情况的简单示例：

- 线程 1 持有锁 A 并请求锁 B
- 线程 2 持有锁 B 并请求锁 A

死锁可分为两种类型：潜在死锁或实际死锁，二者的区别如下：

- 潜在死锁不一定在给定的运行过程中发生，但可能发生在程序的任何执行过程中，具体取决于线程的调度情况以及线程进行锁请求的时间。
- 实际死锁是在程序执行过程中发生的死锁。实际死锁会导致所涉及的线程挂起，但可能会导致整个进程挂起。

获取死锁教程源文件

您可以从 Oracle Developer Studio 开发者门户的下载区域 (http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/index.html)中下载本教程所用的源文件。

在下载并解压缩样例文件之后，您可以在 OracleDeveloperStudio12.5-Samples/ThreadAnalyzer 目录中找到样例。这些样例位于 din_philo 子目录中。din_philo 目录包含了 Makefile 和 DEMO 说明文件，但是本教程并不遵循这些说明，也不使用 Makefile。本教程将逐步指导您执行命令。

为跟随本教程学习，您可以将 din_philo.c 文件从 OracleDeveloperStudio12.5-Samples/ThreadAnalyzer/din_philo 目录复制到其他目录，也可以创建自己的文件并从下面列出的代码内容中复制代码。

模拟哲学家就餐问题的 din_philo.c 样例程序是一个使用 POSIX 线程的 C 程序。该程序可以同时展示潜在死锁和实际死锁。

din_philo.c 的源代码内容

din_philo.c 的源代码如下所示：

```
 1 /*
 2  * Copyright (c) 2006, 2011, Oracle and/or its affiliates. All Rights Reserved.
 3  */
 4
 5 #include <pthread.h>
 6 #include <stdio.h>
 7 #include <unistd.h>
 8 #include <stdlib.h>
 9 #include <errno.h>
10 #include <assert.h>
11
12 #define PHILOS 5
13 #define DELAY 5000
14 #define FOOD 100
15
16 void *philosopher (void *id);
17 void grab_chopstick (int,
18                      int,
19                      char *);
20 void down_chopsticks (int,
21                       int);
22 int food_on_table ();
23
24 pthread_mutex_t chopstick[PHILOS];
25 pthread_t philo[PHILOS];
26 pthread_mutex_t food_lock;
27 int sleep_seconds = 0;
```

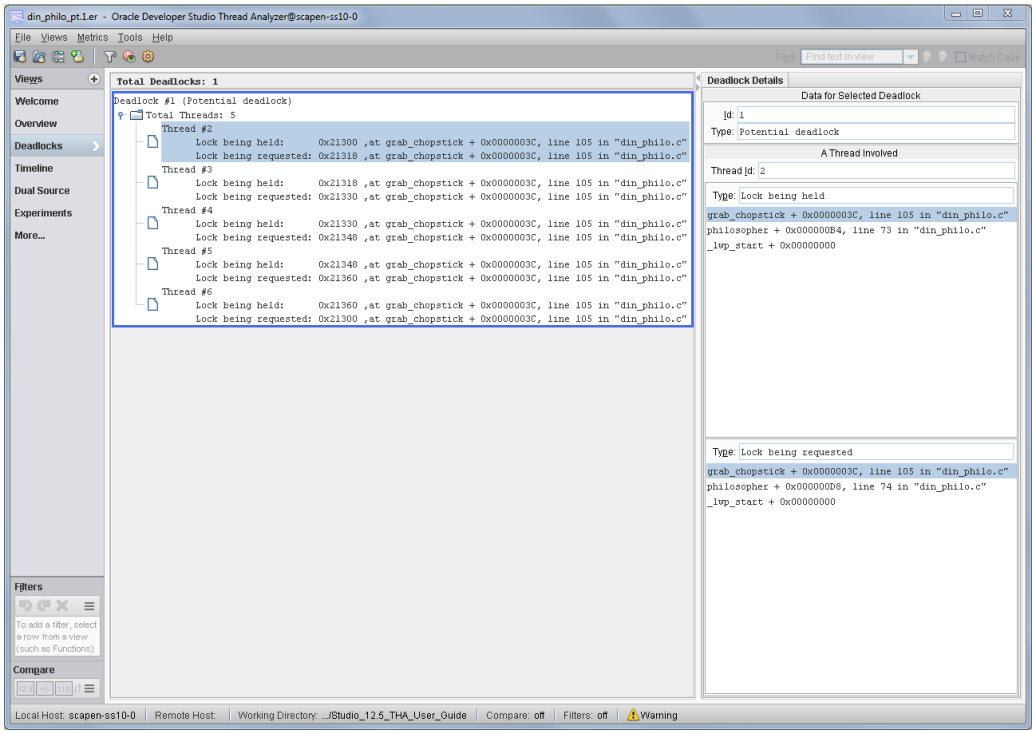
```
28
29
30 int
31 main (int argn,
32       char **argv)
33 {
34     int i;
35
36     if (argn == 2)
37         sleep_seconds = atoi (argv[1]);
38
39     pthread_mutex_init (&food_lock, NULL);
40     for (i = 0; i < PHILOS; i++)
41         pthread_mutex_init (&chopstick[i], NULL);
42     for (i = 0; i < PHILOS; i++)
43         pthread_create (&philo[i], NULL, philosopher, (void *)i);
44     for (i = 0; i < PHILOS; i++)
45         pthread_join (philo[i], NULL);
46     return 0;
47 }
48
49 void *
50 philosopher (void *num)
51 {
52     int id;
53     int i, left_chopstick, right_chopstick, f;
54
55     id = (int)num;
56     printf ("Philosopher %d is done thinking and now ready to eat.\n", id);
57     right_chopstick = id;
58     left_chopstick = id + 1;
59
60     /* Wrap around the chopsticks. */
61     if (left_chopstick == PHILOS)
62         left_chopstick = 0;
63
64     while (f = food_on_table ()) {
65
66         /* Thanks to philosophers #1 who would like to take a nap
67          * before picking up the chopsticks, the other philosophers
68          * may be able to eat their dishes and not deadlock.
69          */
70         if (id == 1)
71             sleep (sleep_seconds);
72
73         grab_chopstick (id, right_chopstick, "right ");
74         grab_chopstick (id, left_chopstick, "left");
75
76         printf ("Philosopher %d: eating.\n", id);
77         usleep (DELAY * (FOOD - f + 1));
78         down_chopsticks (left_chopstick, right_chopstick);
79     }
80
81     printf ("Philosopher %d is done eating.\n", id);
```

```
 82     return (NULL);
 83 }
 84
 85 int
 86 food_on_table ()
 87 {
 88     static int food = FOOD;
 89     int myfood;
 90
 91     pthread_mutex_lock (&food_lock);
 92     if (food > 0) {
 93         food--;
 94     }
 95     myfood = food;
 96     pthread_mutex_unlock (&food_lock);
 97     return myfood;
 98 }
 99
100 void
101 grab_chopstick (int phil,
102                 int c,
103                 char *hand)
104 {
105     pthread_mutex_lock (&chopstick[c]);
106     printf ("Philosopher %d: got %s chopstick %d\n", phil, hand, c);
107 }
108
109 void
110 down_chopsticks (int c1,
111                  int c2)
112 {
113     pthread_mutex_unlock (&chopstick[c1]);
114     pthread_mutex_unlock (&chopstick[c2]);
115 }
```

哲学家就餐方案

哲学家就餐方案是一个结构如下的经典方案。编号从 0 到 4 的五位哲学家坐在圆桌旁思考。随着时间的推移，每个人开始饥饿并决定进餐。餐桌上有一盘面条，但是每位哲学家只有一根筷子可用。为了吃到面条，他们必须共用筷子。每位哲学家右边（他们面向餐桌而坐）的筷子的编号与该哲学家的编号相同。

图 6 哲学家就餐



每位哲学家首先拿到与自己编号相同的筷子。他拿到分配给自己的筷子后，他将去拿分配给邻座的筷子。拿到两根筷子后，他就可以进餐了。吃完后，他将筷子放回桌上的原来位置，一边一根。该过程一直重复，直到把面条吃完。

哲学家如何发生死锁

每位哲学家都拿到自己的筷子并等待使用其邻座的筷子时，就会发生实际的死锁。

- 编号为 0 的哲学家拿着编号为 0 的筷子，但在等待编号为 1 的筷子
- 编号为 1 的哲学家拿着编号为 1 的筷子，但在等待编号为 2 的筷子
- 编号为 2 的哲学家拿着编号为 2 的筷子，但在等待编号为 3 的筷子
- 编号为 3 的哲学家拿着编号为 3 的筷子，但在等待编号为 4 的筷子
- 编号为 4 的哲学家拿着编号为 4 的筷子，但在等待编号为 0 的筷子

在这种情况下，没人可以吃到东西，于是哲学家们陷入了死锁之中。多次运行该程序，您将看到该程序有时会挂起，而有时又可以运行至完成。该程序挂起的情况如以下运行过程样例所示：

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

Execution terminated by pressing CTRL-C
```

为 1 号哲学家引入一段休眠时间

一种避免死锁的方法是让 1 号哲学家在拿起他的筷子之前等待一段时间。就代码而言，可让他在拿起自己的筷子之前休眠指定的一段时间 (`sleep_seconds`)。如果他休眠的时间足够长，该程序就可能会完成而不发生任何实际死锁。可以将他休眠的秒数指定为可执行程序的参数。如果不指定参数，该哲学家就不会休眠。

以下伪代码显示了每位哲学家的逻辑：

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

下面列出了该程序的一次运行，在这次运行中 1 号哲学家在拿起他的筷子之前等待 30 秒。该程序运行至完成，所有五位哲学家都吃到了面条。

```
% 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

尝试多次运行该程序，并指定不同的休眠参数。如果 1 号哲学家在拿起他的筷子之前仅等待很短的一段时间，那么发生什么情况呢？如果他等待的时间更长一些，又会怎么样？尝试为可执行程序 `a.out` 指定不同的休眠参数。在使用或不使用休眠参数的情况下多次重新运行该程序。该程序有时会挂起，而有时又可以运行至完成。程序是否挂起取决于线程的调度情况以及线程进行锁请求的时间。

如何使用线程分析器找到死锁

可以使用线程分析器检查程序中的潜在死锁和实际死锁。线程分析器沿用与 Oracle Developer Studio 性能分析器相同的收集-分析模型。

使用线程分析器的过程涉及三个步骤：

- 编译源代码。
- 创建死锁检测实验。
- 检查实验结果。

编译源代码

编译代码并务必指定 -g。不要指定高优化级别，因为在高优化级别时报告的信息（如行号和调用堆栈）可能是错误的。应使用 -g -xopenmp=noopt 编译 OpenMP 程序，并仅使用 -g -mt 编译 POSIX 线程程序。

有关这些编译器选项的更多信息，请参见 cc(1)、CC(1) 或 f95(1) 手册页。

对于本教程，请使用以下命令编译代码：

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

创建死锁检测实验

使用带有 -r deadlock 选项的 collect 命令。此选项将在程序的执行期间创建死锁检测实验。

对于本教程，请使用以下命令创建名为 din_philo.1.er 的死锁检测实验：

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

collect -r 命令接受以下选项，创建死锁检测实验时这些选项非常有用：

terminate	如果检测到不可修复的错误，将终止程序。
abort	如果检测到不可修复的错误，则终止带有信息转储的程序。
continue	如果检测到不可修复的错误，将允许程序继续。

缺省行为是 terminate。

可以将前面的任何选项与 collect -r 命令配合使用来得到您需要的行为。例如，要在发生实际死锁时导致程序终止并具有信息转储，请使用以下 collect -r 命令。

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

要在发生实际死锁时导致程序挂起，请使用以下 collect -r 命令：

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

可以通过创建多个死锁检测实验来提高检测到死锁的可能性。对于各个实验，应使用不同的线程数和不同的输入数据。例如，在 din_philo.c 代码中，可以更改以下行中的值：

```
13  #define PHILOS 5
14  #define DELAY 5000
15  #define FOOD 100
```

然后可以像以前一样进行编译，并收集其他实验。

有关更多信息，请参见 collect(1) 和 collector(1) 手册页。

检查死锁检测实验

可以使用线程分析器、性能分析器或 er_print 实用程序检查死锁检测实验。线程分析器和性能分析器都提供 GUI 界面；线程分析器显示的是一组简化的缺省视图，但在其他方面与性能分析器完全相同。

使用线程分析器查看死锁检测实验

要启动线程分析器并打开 din_philo.1.er 实验，请键入以下命令：

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

线程分析器在左侧具有菜单栏、工具栏和垂直导航栏，因而您可以选择数据视图。

打开收集的死锁检测实验时，缺省情况下显示以下数据视图：

- "Overview"（概述）屏幕显示已装入实验的度量概述。
- "Deadlocks"（死锁）视图显示线程分析器在程序中检测到的潜在死锁和实际死锁的列表。er_print din_philo.1.er 显示了每个死锁所涉及的线程。这些线程形成一个循环链，其中每个线程都持有一个锁并请求该链中的下一个线程持有的另一个锁。

  选择死锁后，右侧面板中的 "Deadlock Details"（死锁详细信息）窗口会显示有关涉及的线程的详细信息。
- "Dual Source"（双源）视图会显示该线程持有锁的源代码位置，以及同一线程请求锁的源代码位置。该线程持有锁和请求锁的源代码行会突出显示。要显示此视图，请在 "Deadlocks"（死锁）视图上选择循环链中的线程，然后单击 "Dual Source"（双源）视图。
- "Experiments"（实验）视图显示实验中的装入对象并列出所有错误和警告消息。

可以选择使用 "More Views"（更多视图）选项菜单查看其他视图。

使用 er_print 查看死锁检测实验

er_print 实用程序提供命令行界面。可以在交互式会话中使用 er_print 实用程序并在该会话期间指定子命令。也可以使用命令行选项以非交互方式指定子命令。

使用 er_print 实用程序检查死锁时，以下子命令非常有用：

- -deadlocks

  该选项会报告实验中检测到的所有潜在死锁和实际死锁。在 (er_print) 提示符下指定 -deadlocks，或者在 er_print 命令行上指定 deadlocks。
- -ddetail *deadlock-ID*

该选项会返回具有指定 *deadlock-ID* 的死锁的详细信息。在 (er_print) 提示符下指定 -ddetail，或者在 er_print 命令行上指定 ddetail。如果指定的 *deadlock-ID* 为 **all**，将显示所有死锁的详细信息。否则，请指定单个死锁编号，例如为第一个死锁指定 **1**。

- -header

  该选项会显示有关实验的描述性信息并报告所有错误或警告。在 (er_print) 提示符下指定 header，或者在命令行上指定 -header。

有关更多信息，请参阅 collect(1)、tha(1)、analyzer(1) 和 er_print(1) 手册页。

了解死锁实验结果

本节说明如何使用线程分析器检查哲学家就餐程序中的死锁。

检查出现死锁的运行

下面列出了导致实际死锁的哲学家就餐程序的一次运行。

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

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

Execution terminated by pressing CTRL-C

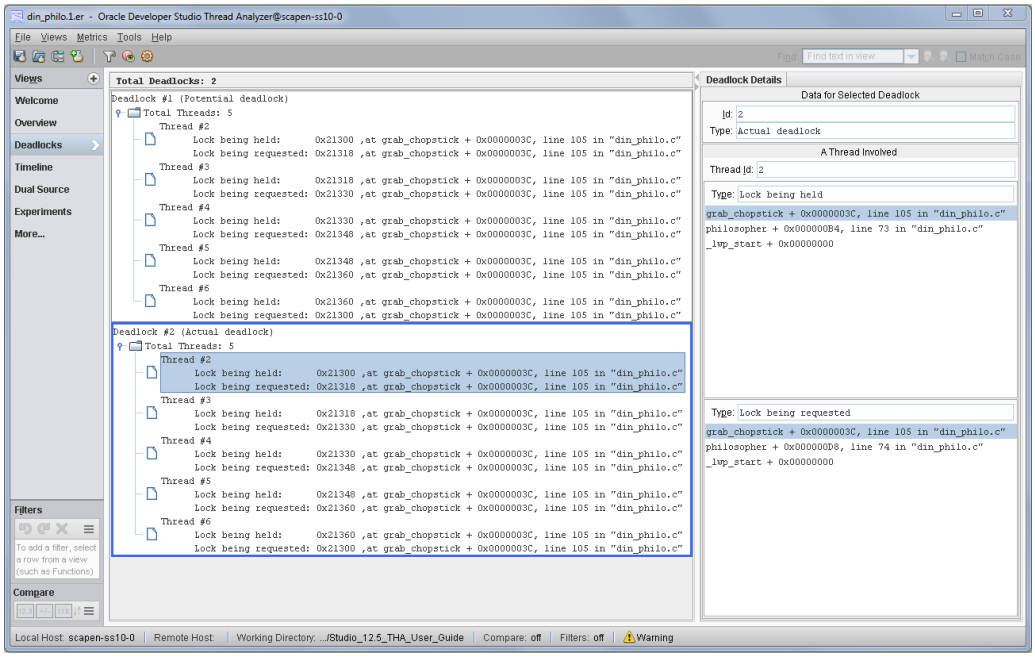
键入以下命令以使用 `er_print` 实用程序检查实验：

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

Deadlock #1, Potential deadlock
     Thread #2
          Lock being held: 0x21300, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"
          Lock being requested: 0x21318, at: grab_chopstick + 0x0000003C, line 105 in
 "din_philo.c"
     Thread #3
          Lock being held: 0x21318, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"
          Lock being requested: 0x21330, at: grab_chopstick + 0x0000003C, line 105 in
 "din_philo.c"
     Thread #4
          Lock being held: 0x21330, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"
          Lock being requested: 0x21348, at: grab_chopstick + 0x0000003C, line 105 in
 "din_philo.c"
     Thread #5
          Lock being held: 0x21348, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"
          Lock being requested: 0x21360, at: grab_chopstick + 0x0000003C, line 105 in
 "din_philo.c"
     Thread #6
          Lock being held: 0x21360, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"
          Lock being requested: 0x21300, at: grab_chopstick + 0x0000003C, line 105 in
 "din_philo.c"

Deadlock #2, Actual deadlock
     Thread #2
          Lock being held: 0x21300, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"
          Lock being requested: 0x21318, at: grab_chopstick + 0x0000003C, line 105 in
 "din_philo.c"
```

```
    Thread #3
         Lock being held: 0x21318, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"
         Lock being requested: 0x21330, at: grab_chopstick + 0x0000003C, line 105 in
 "din_philo.c"
    Thread #4
         Lock being held: 0x21330, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"
         Lock being requested: 0x21348, at: grab_chopstick + 0x0000003C, line 105 in
 "din_philo.c"
    Thread #5
         Lock being held: 0x21348, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"
         Lock being requested: 0x21360, at: grab_chopstick + 0x0000003C, line 105 in
 "din_philo.c"
    Thread #6
         Lock being held: 0x21360, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"
         Lock being requested: 0x21300, at: grab_chopstick + 0x0000003C, line 105 in
 "din_philo.c"

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

以下屏幕抓图显示了线程分析器中显示的死锁信息。

图 7 din_philo.c 中检测到的死锁



线程分析器报告了 din_philo.c 的两个死锁，一个是潜在死锁，另一个是实际死锁。通过进一步检查，可以发现两个死锁是相同的。

该死锁涉及的循环链如下：

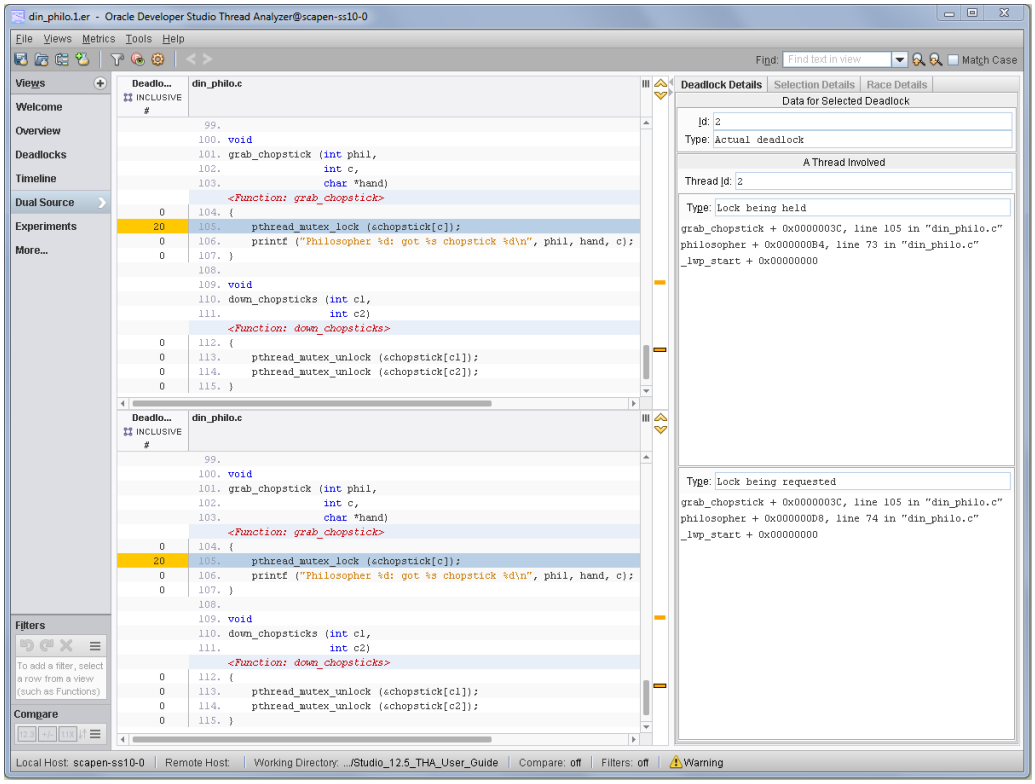
线程 2：在地址 0x21300 持有锁，在地址 0x21318 请求锁
线程 3：在地址 0x21318 持有锁，在地址 0x21330 请求锁
线程 4：在地址 0x21330 持有锁，在地址 0x21348 请求锁
线程 5：在地址 0x21348 持有锁，在地址 0x21360 请求锁
线程 6：在地址 0x21360 持有锁，在地址 0x21300 请求锁

选择循环链中的第一个线程（线程 2），然后单击 "Dual Source"（双源）视图，可以看到线程 2 在地址 0x21430 持有锁的对应源代码位置，以及该线程在 0x21448 请求锁的对应源代码位置。下图显示了对应于线程 2 的 "Dual Source"（双源）视图。

以下屏幕抓图显示了对应于线程 2 的 "Dual Source"（双源）视图。屏幕抓图的上半部分显示了线程 2 通过在第 105 行调用 pthread_mutex_lock（） 获取了地址 0x21300 处的锁。屏幕抓图的下半部分显示了同一线程通过在第 105 行调用 pthread_mutex_lock 请求了地址 0x21318 处的锁。在对 pthread_mutex_lock 的两次调用中，分别使用了不同的锁作为参数。通常，锁获取和锁请求操作可能不在同一源代码行上。

在屏幕抓图中每个源代码行的左侧会显示缺省的度量："Exclusive Deadlocks"（互斥死锁）度量。此度量显示了该源代码行上报告的（死锁涉及的）锁获取或锁请求操作的总次数。只有属于死锁循环链一部分的源代码行才具有此度量的值，并且该值大于零。

图 8 din_philo.c 中的潜在死锁



检查存在潜在死锁但仍可完成的运行

如果提供足够大的休眠参数，哲学家就餐程序就可避免实际死锁并正常终止。但是，正常终止并不意味着程序可以完全避免死锁。它仅仅表示在给定的一次运行中，持有和请求的锁未形成死锁链。如果在其他运行中出现了时间变化，则仍可能会发生实际死锁。下面列出了由于引入 40 秒休眠时间（作为可执行文件的参数提供）而正常终止的哲学家就餐程序的一次运行。

```
% cc -g -o din_philo_pt din_philo.c
% collect -r deadlock -o din_philo_pt.1.er din_philo_pt 40
Creating experiment database tha.2.er ...
Philosopher 0 is done thinking and now ready to eat.
Philosopher 2 is done thinking and now ready to eat.
Philosopher 1 is done thinking and now ready to eat.
Philosopher 3 is done thinking and now ready to eat.
```

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

Execution terminated normally

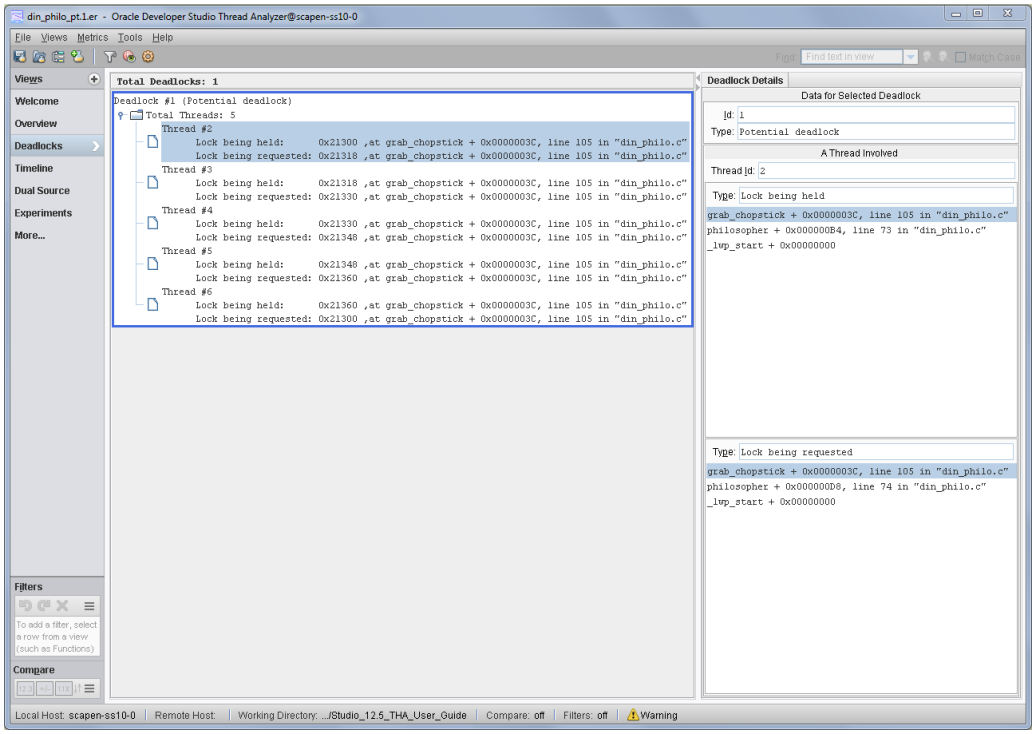
键入提示符下显示的以下命令以使用 `er_print` 实用程序检查实验：

```
% er_print din_philo_pt.1.er
(er_print) deadlocks
Deadlock #1, Potential deadlock

Thread #2
     Lock being held: 0x21300, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"
     Lock being requested: 0x21318, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"
Thread #3
     Lock being held: 0x21318, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"
     Lock being requested: 0x21330, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"
Thread #4
     Lock being held: 0x21330, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"
     Lock being requested: 0x21348, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"
Thread #5
```

```
     Lock being held: 0x21348, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"
     Lock being requested: 0x21360, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"
Thread #6
     Lock being held: 0x21360, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"
     Lock being requested: 0x21300, at: grab_chopstick + 0x0000003C, line 105 in "din_philo.c"

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

以下屏幕抓图显示了线程分析器中的潜在死锁信息。

图 9 din_philo.c 中的潜在死锁



修复死锁和了解误报

一种消除潜在死锁和实际死锁的方法是使用一种令牌机制，该机制要求哲学家必须收到令牌后才能尝试进餐。可用令牌的数量必须少于餐桌前哲学家的人数。当哲学家收到令牌后，他可以根据餐桌规则尝试进餐。每位哲学家在进餐后归还令牌，然后重复该过程。以下伪代码显示了使用令牌机制时每位哲学家的逻辑。

```
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
   }
```

以下几节将详细介绍令牌机制的两种不同实现。

使用令牌控制哲学家

下面列出了使用令牌机制的哲学家就餐程序的修正版本。该解决方案结合了四个令牌，比就餐人数少一个，因此最多有四位哲学家可同时尝试进餐。该版本的程序称为 `din_philo_fix1.c`：

提示 - 下载样例应用程序之后，可以从 `OracleDeveloperStudio12.5-Samples/ThreadAnalyzer/din_philo` 目录复制 `din_philo_fix1.c` 文件。

```
 1 /*
 2  * Copyright (c) 2006, 2011, Oracle and/or its affiliates. All Rights Reserved.
 3  */
 4
 5 #include <pthread.h>
 6 #include <stdio.h>
 7 #include <unistd.h>
 8 #include <stdlib.h>
 9 #include <errno.h>
10 #include <assert.h>
11
12 #ifdef __linux__
13 #include <stdint.h>
14 #endif
15
16 #define PHILOS 5
17 #define DELAY 5000
18 #define FOOD 100
19
20 void *philosopher (void *id);
21 void grab_chopstick (int,
22                      int,
23                      char *);
24 void down_chopsticks (int,
25                       int);
26 int food_on_table ();
27 int get_token ();
28 void return_token ();
```

```
29
30 pthread_mutex_t chopstick[PHILOS];
31 pthread_t philo[PHILOS];
32 pthread_mutex_t food_lock;
33 pthread_mutex_t num_can_eat_lock;
34 int sleep_seconds = 0;
35 uint32_t num_can_eat = PHILOS - 1;
36
37
38 int
39 main (int argn,
40       char **argv)
41 {
42     int i;
43
44     pthread_mutex_init (&food_lock, NULL);
45     pthread_mutex_init (&num_can_eat_lock, NULL);
46     for (i = 0; i < PHILOS; i++)
47         pthread_mutex_init (&chopstick[i], NULL);
48     for (i = 0; i < PHILOS; i++)
49         pthread_create (&philo[i], NULL, philosopher, (void *)i);
50     for (i = 0; i < PHILOS; i++)
51         pthread_join (philo[i], NULL);
52     return 0;
53 }
54
55 void *
56 philosopher (void *num)
57 {
58     int id;
59     int i, left_chopstick, right_chopstick, f;
60
61     id = (int)num;
62     printf ("Philosopher %d is done thinking and now ready to eat.\n", id);
63     right_chopstick = id;
64     left_chopstick = id + 1;
65
66     /* Wrap around the chopsticks. */
67     if (left_chopstick == PHILOS)
68         left_chopstick = 0;
69
70     while (f = food_on_table ()) {
71         get_token ();
72
73         grab_chopstick (id, right_chopstick, "right ");
74         grab_chopstick (id, left_chopstick, "left");
75
76         printf ("Philosopher %d: eating.\n", id);
77         usleep (DELAY * (FOOD - f + 1));
78         down_chopsticks (left_chopstick, right_chopstick);
79
80         return_token ();
81     }
82
```

```
 83     printf ("Philosopher %d is done eating.\n", id);
 84     return (NULL);
 85 }
 86
 87 int
 88 food_on_table ()
 89 {
 90     static int food = FOOD;
 91     int myfood;
 92
 93     pthread_mutex_lock (&food_lock);
 94     if (food > 0) {
 95         food--;
 96     }
 97     myfood = food;
 98     pthread_mutex_unlock (&food_lock);
 99     return myfood;
100 }
101
102 void
103 grab_chopstick (int phil,
104                 int c,
105                 char *hand)
106 {
107     pthread_mutex_lock (&chopstick[c]);
108     printf ("Philosopher %d: got %s chopstick %d\n", phil, hand, c);
109 }
110
111
112
113 void
114 down_chopsticks (int c1,
115                  int c2)
116 {
117     pthread_mutex_unlock (&chopstick[c1]);
118     pthread_mutex_unlock (&chopstick[c2]);
119 }
120
121
122 int
123 get_token ()
124 {
125     int successful = 0;
126
127     while (!successful) {
128         pthread_mutex_lock (&num_can_eat_lock);
129         if (num_can_eat > 0) {
130             num_can_eat--;
131             successful = 1;
132         }
133         else {
134             successful = 0;
135         }
136         pthread_mutex_unlock (&num_can_eat_lock);
```

```
137     }
138 }
139
140 void
141 return_token ()
142 {
143     pthread_mutex_lock (&num_can_eat_lock);
144     num_can_eat++;
145     pthread_mutex_unlock (&num_can_eat_lock);
146 }
```

尝试编译这一修正版本的哲学家就餐程序，并多次运行该程序。令牌机制限制了尝试使用筷子的就餐人数，从而可避免实际死锁和潜在死锁。

要进行编译，请使用以下命令：

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

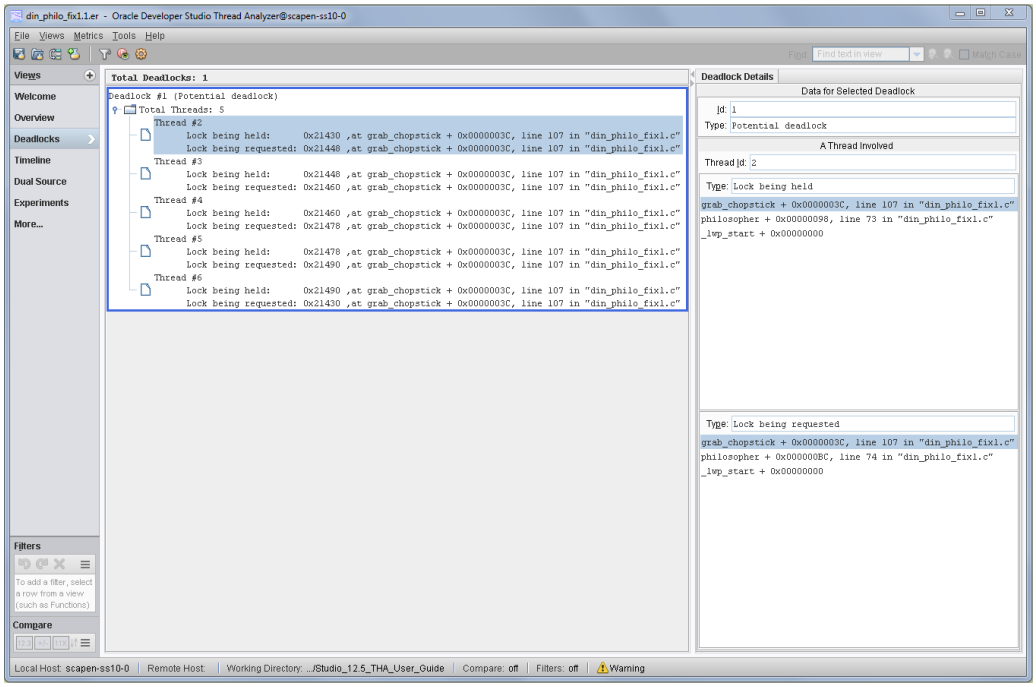
收集实验：

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

误报的报告

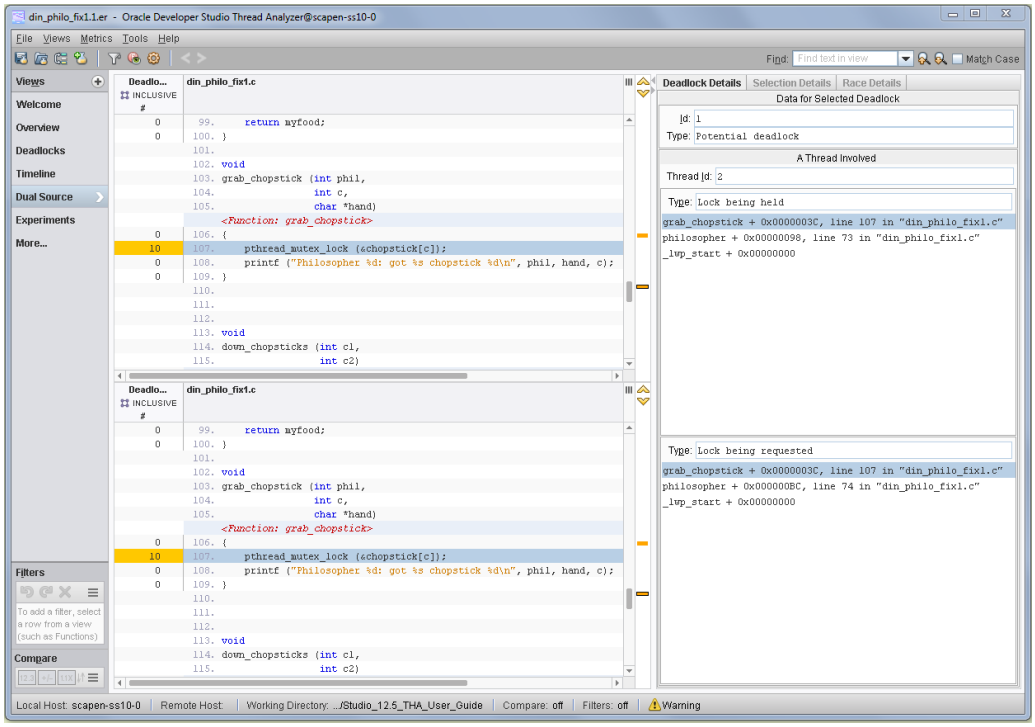
即使是使用令牌机制，线程分析器也会在不存在死锁时报告该实现存在潜在死锁。这是一个误报。请参考以下包含潜在死锁详细信息的屏幕抓图。

图 10 误报的潜在死锁



选择循环链中的第一个线程（线程 2），然后单击 "Dual Source"（双源）视图，可以看到线程 2 在地址 `0x216a8` 持有锁的对应源代码位置，以及该线程在 `0x216c0` 请求锁的对应源代码位置。下图显示了对应于线程 2 的 "Dual Source"（双源）视图。

图 11 误报的潜在死锁的源代码



`din_philo_fix1.c` 中的 `get_token（）` 函数使用 `while` 循环来同步线程。线程在成功获得令牌之前不会离开 `while` 循环（当 `num_can_eat` 大于 0 时可成功获得令牌）。`while` 循环将同时就餐的人数限制为 4。但是，线程分析器无法识别由 `while` 循环实现的同步。它认为所有五位哲学家同时尝试抓取筷子并进餐，因此报告了一个潜在死锁。下一节将详细说明如何使用线程分析器可识别的同步来限制同时就餐的人数。

另一种令牌机制

下面列出了令牌机制的另一种实现。该实现仍然使用四个令牌，所以最多有四个就餐者可同时尝试进餐。但是，该实现使用 `sem_wait（）` 和 `sem_post（）` 信号例程限制就餐的哲学家人数。该版本的源文件称为 `din_philo_fix2.c`。

提示 - 下载样例应用程序之后，可以从 `OracleDeveloperStudio12.5-Samples/ThreadAnalyzer/din_philo` 目录复制 `din_philo_fix2.c` 文件。

下面列出了详细的 din_philo_fix2.c：

```
 1 /*
 2  * Copyright (c) 2006, 2011, Oracle and/or its affiliates. All Rights Reserved.
 3  */
 4
 5 #include <pthread.h>
 6 #include <stdio.h>
 7 #include <unistd.h>
 8 #include <stdlib.h>
 9 #include <errno.h>
10 #include <assert.h>
11 #include <semaphore.h>
12
13 #define PHILOS 5
14 #define DELAY 5000
15 #define FOOD 100
16
17 void *philosopher (void *id);
18 void grab_chopstick (int,
19                      int,
20                      char *);
21 void down_chopsticks (int,
22                       int);
23 int food_on_table ();
24 int get_token ();
25 void return_token ();
26
27 pthread_mutex_t chopstick[PHILOS];
28 pthread_t philo[PHILOS];
29 pthread_mutex_t food_lock;
30 int sleep_seconds = 0;
31 sem_t num_can_eat_sem;
32
33
34 int
35 main (int argn,
36       char **argv)
37 {
38     int i;
39
40     pthread_mutex_init (&food_lock, NULL);
41     sem_init(&num_can_eat_sem, 0, PHILOS - 1);
42     for (i = 0; i < PHILOS; i++)
43         pthread_mutex_init (&chopstick[i], NULL);
44     for (i = 0; i < PHILOS; i++)
45         pthread_create (&philo[i], NULL, philosopher, (void *)i);
46     for (i = 0; i < PHILOS; i++)
47         pthread_join (philo[i], NULL);
48     return 0;
49 }
50
51 void *
52 philosopher (void *num)
```

```
53 {
54     int id;
55     int i, left_chopstick, right_chopstick, f;
56 
57     id = (int)num;
58     printf ("Philosopher %d is done thinking and now ready to eat.\n", id);
59     right_chopstick = id;
60     left_chopstick = id + 1;
61 
62     /* Wrap around the chopsticks. */
63     if (left_chopstick == PHILOS)
64         left_chopstick = 0;
65 
66     while (f = food_on_table ()) {
67         get_token ();
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         return_token ();
77     }
78 
79     printf ("Philosopher %d is done eating.\n", id);
80     return (NULL);
81 }
82 
83 int
84 food_on_table ()
85 {
86     static int food = FOOD;
87     int myfood;
88 
89     pthread_mutex_lock (&food_lock);
90     if (food > 0) {
91         food--;
92     }
93     myfood = food;
94     pthread_mutex_unlock (&food_lock);
95     return myfood;
96 }
97 
98 void
99 grab_chopstick (int phil,
100                 int c,
101                 char *hand)
102 {
103     pthread_mutex_lock (&chopstick[c]);
104     printf ("Philosopher %d: got %s chopstick %d\n", phil, hand, c);
105 }
106 
```

```
107 void
108 down_chopsticks (int c1,
109                   int c2)
110 {
111     pthread_mutex_unlock (&chopstick[c1]);
112     pthread_mutex_unlock (&chopstick[c2]);
113 }
114
115
116 int
117 get_token ()
118 {
119     sem_wait(&num_can_eat_sem);
120 }
121
122 void
123 return_token ()
124 {
125     sem_post(&num_can_eat_sem);
126 }
```

这一新实现使用信号 num_can_eat_sem 来限制可同时就餐的哲学家人数。信号 num_can_eat_sem 初始化为 4，比哲学家人数少一个。尝试进餐前，哲学家调用 get_token（），后者又调用 sem_wait(&num_can_eat_sem)。调用 sem_wait（） 会导致执行调用的哲学家进入等待状态，直到信号值变为正值，然后将信号值减 1。哲学家进餐完毕后，他将调用 return_token（），后者又调用 sem_post(&num_can_eat_sem)。调用 sem_post（） 可将信号值加 1。线程分析器可识别对 sem_wait（） 和 sem_post（） 的调用，并判定并非所有的哲学家都同时尝试进餐。

注 - 必须使用 -lrt 编译 din_philo_fix2.c 以链接相应的信号例程。

要编译 din_philo_fix2.c，请使用以下命令：

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

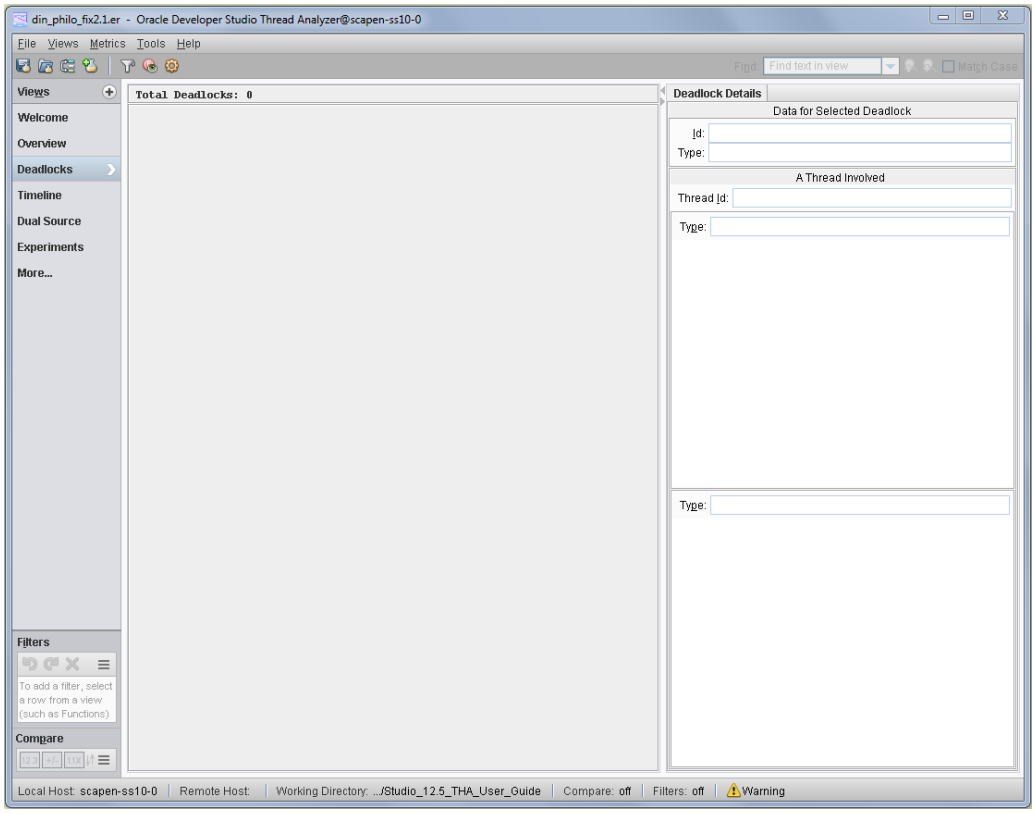
如果多次运行程序 din_philo_fix2 的这一新实现，将会发现它每次都正常终止，不会挂起。

在这一新的二进制代码上创建实验：

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

您将发现线程分析器不报告 din_philo_fix2.1.er 实验中的任何实际死锁或潜在死锁，如下图所示。

图 12 未报告 din_philo_fix2.c 中的任何死锁



有关线程分析器可识别的线程和内存分配 API 的列表，请参见附录 A, 线程分析器可识别的 API。

♦ ♦ ♦ 附 录 A

线程分析器可识别的 API

线程分析器可以识别由 OpenMP、POSIX 线程和 Oracle Solaris 线程提供的大多数标准同步 API 和构造。但是，该工具无法识别用户定义的同步，如果使用了这样的同步，可能会误报数据争用。例如，该工具无法识别通过手编汇编语言代码实现的自旋锁。

线程分析器用户 API

如果代码包含用户定义的同步，请将线程分析器支持的用户 API 插入该程序中来标识这些同步。通过此标识，线程分析器可识别这些同步并减少误报的数量。下面列出了在 `libtha.so` 中定义的线程分析器用户 API。

表 1 线程分析器用户 API

例程名称	描述
`tha_notify_acquire_lock ()`	可以在程序尝试获取用户定义的锁之前立即调用此例程。
`tha_notify_lock_acquired ()`	可以在成功获取用户定义的锁之后立即调用此例程。
`tha_notify_acquire_writelock ()`	可以在程序尝试以写入模式获取用户定义的读/写锁之前立即调用此例程。
`tha_notify_writelock_acquired ()`	可以在以读取模式成功获取用户定义的读/写锁之后立即调用此例程。
`tha_notify_acquire_readlock ()`	可以在程序尝试以读取模式获取用户定义的读/写锁之前立即调用此例程。
`tha_notify_readlock_acquired ()`	可以在以读取模式成功获取用户定义的读/写锁之后立即调用此例程。
`tha_notify_release_lock ()`	可以在释放用户定义的锁或读/写锁之前立即调用此例程。
`tha_notify_lock_released ()`	可以在成功释放用户定义的锁或读/写锁之后立即调用此例程。
`tha_notify_sync_post_begin ()`	可以在执行用户定义的后期同步之前立即调用此例程。
`tha_notify_sync_post_end ()`	可以在执行用户定义的后期同步之后立即调用此例程。
`tha_notify_sync_wait_begin ()`	可以在执行用户定义的等待同步之前立即调用此例程。
`tha_notify_sync_wait_end ()`	可以在执行用户定义的等待同步之后立即调用此例程。
`tha_check_datarace_mem ()`	该例程指示线程分析器在执行数据争用检测时监视或忽略对指定内存块的访问。
`tha_check_datarace_thr ()`	该例程指示线程分析器在执行数据争用检测时监视或忽略一个或多个线程进行的内存访问。

提供了 C/C++ 版本和 Fortran 版本的 API。每次 API 调用都采用单个参数 ID，其值应该唯一地标识同步对象。

在 C/C++ 版本的 API 中，参数的类型为 `uintptr_t`，在 32 位模式下长度为 4 个字节，在 64 位模式下长度为 8 个字节。在调用此版本的任何 API 时，都需要将 `#include <tha_interface.h>` 添加到 C/C++ 源文件中。

在 Fortran 版本的 API 中，参数的类型为整型 `tha_sobj_kind`，在 32 位模式和 64 位模式下长度均为 8 个字节。在调用此版本的任何 API 时，都需要将 `#include "tha_finterface.h"` 添加到 Fortran 源文件中。

为了唯一地标识同步对象，对于每个不同的同步对象，参数 ID 应具有不同的值。其中一种做法是将同步对象的地址值用作 ID。以下代码示例说明了如何使用 API 避免误报数据争用。

例 1　　使用线程分析器 API 避免误报数据争用的示例

```
# 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;
```

有关用户 API 的更多信息，请参见 `libtha`(3) 手册页。

其他可识别的 API

以下几节将详细介绍线程分析器可识别的线程 API。

POSIX 线程 API

有关这些 API 的更多信息，请参见 Oracle Solaris 文档中的《多线程编程指南》。

```
pthread_detach ()
pthread_mutex_init ()
pthread_mutex_lock ()
pthread_mutex_timedlock ()
pthread_mutex_reltimedlock_np ()
pthread_mutex_timedlock ()
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_rwlock_init ()
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 ()
```

Oracle Solaris 线程 API

有关这些 API 的更多信息，请参见 Oracle Solaris 文档中的《多线程编程指南》。

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

内存分配 API

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

有关内存分配 API 的信息，请参见 `malloc`(3C) 手册页。

内存操作 API

`memcpy ()`
`memccpy ()`
`memmove ()`

```
memchr ()
memcmp ()
memset ()
```

有关内存操作 API 的信息，请参见 `memcpy`(3C) 手册页。

字符串操作 API

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

有关字符串操作 API 的信息，请参见 `strcat`(3C) 手册页。

实时库 API

```
sem_post ()
sem_wait ()
sem_trywait ()
sem_timedwait ()
```

原子操作 (`atomic_ops`) API

```
atomic_add ()
```

```
atomic_and ()
atomic_cas ()
atomic_dec ()
atomic_inc ()
atomic_or ()
atomic_swap ()
```

OpenMP API

线程分析器可识别 OpenMP 同步，例如屏障、锁、临界区域、原子区域和任务等待 (taskwait)。

有关更多信息，请参见《Oracle Developer Studio 12.5：OpenMP API 用户指南》。

♦ ♦ ♦ 附 录 B

使用线程分析器的提示

本附录介绍关于使用线程分析器的一些提示信息。

编译应用程序

关于在收集实验之前编译应用程序的提示：

- 生成应用程序二进制代码时，使用 -g 编译器选项。通过该选项，线程分析器可以报告数据争用和死锁的行号信息。
- 生成应用程序二进制代码时，使用低于 -xO3 的优化级别进行编译。编译器转换可能会误报行号信息并使结果难以理解。
- 线程分析器会插入“内存分配 API” [72]中所示的例程。链接到内存分配库的归档版本可能会导致误报数据争用。

检测应用程序以检测数据争用

关于在收集实验之前检测应用程序以检测数据争用的提示：

- 如果没有检测二进制代码以进行数据争用检测，`collect -r race` 命令会发出一条如下所示的警告：

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

- 通过使用 `nm` 命令以及查找对 `tha` 例程的调用，可以确定是否检测了二进制代码以进行数据争用检测。如果显示名称以 `__tha_` 开头的例程，则表明已检测二进制代码。示例输出如下所示。

  源代码级检测：

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

```
  [53] | 135468| 0|FUNC |GLOB |0 |UNDEF |__tha_src_read_w_frame
  [61] | 135444| 0|FUNC |GLOB |0 |UNDEF |__tha_src_write_w_frame
```

二进制代码级检测：

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

使用 collect 命令运行应用程序

关于运行检测后的应用程序以检测数据争用和死锁的提示。

- 确保 Oracle Solaris 系统已安装了所有必需的修补程序。`collect` 命令可列出所有缺少的必需修补程序。对于 OpenMP 应用程序，需要安装最新版本的 `libmtsk.so`。
- 检测可能会导致执行时间显著延长（高达 50 倍甚至更多），还会导致内存消耗增加。可以尝试通过使用较小的数据集缩短执行时间。也可以尝试通过增加线程数缩短执行时间。
- 为检测数据争用，请确保应用程序当前使用的线程不止一个。对于 OpenMP，您可以指定线程数，方法是将环境变量 `OMP_NUM_THREADS` 设置为所需的线程数并将环境变量 `OMP_DYNAMIC` 设置为 `FALSE`。

报告数据争用

关于报告数据争用的提示：

- 线程分析器检测运行时的数据争用。应用程序的运行时行为取决于使用的输入数据集和操作系统调度。请通过使用不同的线程数以及使用不同的输入数据集，在 `collect` 下运行应用程序。还应使用单个数据集重复进行实验，以最大限度地提高该工具检测到数据争用的可能性。
- 线程分析器检测从单个进程产生的不同线程之间的数据争用。它不检测不同进程之间的数据争用。
- 线程分析器不报告数据争用中访问的变量的名称。但是，若要确定变量的名称，可以检查发生两次数据争用访问的源代码行，再找出这些源代码行上写入的变量和读取的变量。
- 在某些情况下，线程分析器可能会报告实际在程序中并未发生的数据争用。这些数据争用称为误报。使用由用户实现的同步时或在线程之间回收内存时，通常会发生这种情况。例如，如果代码中包含实现自旋锁的手编程序集，线程分析器将无法识别这些同步点。通过在源代码中插入对线程分析器用户 API 的调用，可让线程分析器知道

有关用户定义的同步的情况。有关更多信息，请参见“误报” [35]和附录 A, 线程分析器可识别的 API。

- 使用源代码级别检测报告的数据争用与使用二进制代码级别检测报告的数据争用可能不相同。在二进制代码级别检测中，缺省情况下会在打开共享库时检测这些共享库（无论它们是静态链接到程序中的，还是通过 `dlopen`（） 动态打开的）。在源代码级检测中，仅当使用 `-xinstrument=datarace` 编译库的源代码时，才会对库进行检测。