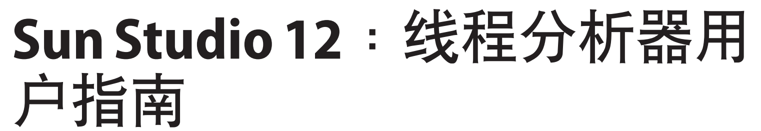


Sun Studio 12：线程分析器用户指南

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

文件号码 820–1220–10

版权所有 2007 Sun Microsystems, Inc.　4150 Network Circle, Santa Clara, CA 95054 U.S.A.　保留所有权利。

对于本文档中介绍的产品，Sun Microsystems, Inc. 对其所涉及的技术拥有相关的知识产权。需特别指出的是（但不局限于此），这些知识产权可能包含一项或多项美国专利，以及在美国和其他国家/地区申请的待批专利。

美国政府权利－商业软件。政府用户应遵循 Sun Microsystems, Inc. 的标准许可协议，以及 FAR（Federal Acquisition Regulations，即“联邦政府采购法规”）的适用条款及其补充条款。

本发行版可能包含由第三方开发的内容。

本产品的某些部分可能是从 Berkeley BSD 系统衍生出来的，并获得了加利福尼亚大学的许可。UNIX 是 X/Open Company, Ltd. 在美国和其他国家/地区独家许可的注册商标。

Sun、Sun Microsystems、Sun 徽标、Solaris 徽标、Java 咖啡杯徽标、docs.sun.com、Java 和 Solaris 是 Sun Microsystems, Inc. 在美国和其他国家/地区的商标或注册商标。所有 SPARC 商标的使用均已获得许可，它们是 SPARC International, Inc. 在美国和其他国家/地区的商标或注册商标。标有 SPARC 商标的产品均基于由 Sun Microsystems, Inc. 开发的体系结构。

OPEN LOOK 和 Sun™ 图形用户界面是 Sun Microsystems, Inc. 为其用户和许可证持有者开发的。Sun 感谢 Xerox 在研究和开发可视或图形用户界面的概念方面为计算机行业所做的开拓性贡献。Sun 已从 Xerox 获得了对 Xerox 图形用户界面的非独占性许可证，该许可证还适用于实现 OPEN LOOK GUI 和在其他方面遵守 Sun 书面许可协议的 Sun 许可证持有者。

本出版物所介绍的产品以及所包含的信息受美国出口控制法制约，并应遵守其他国家/地区的进出口法律。严禁将本产品直接或间接地用于核设施、导弹、生化武器或海上核设施，也不能直接或间接地出口给核设施、导弹、生化武器或海上核设施的最终用户。严禁出口或转口到美国禁运的国家/地区以及美国禁止出口清单中所包含的实体，包括但不限于被禁止的个人以及特别指定的国家/地区的公民。

本文档按“原样”提供，对于所有明示或默示的条件、陈述和担保，包括对适销性、适用性或非侵权性的默示保证，均不承担任何责任，除非此免责声明的适用范围在法律上无效。

070723@18135

目录

前言 5

1 什么是线程分析器？它有何作用？ 11

1.1 线程分析器入门 11

1.2 什么是数据争用? 11

1.3 什么是死锁? 12

1.4 线程分析器使用模型 12

2 数据争用教程 13

2.1 教程源文件 13

2.1.1 `omp_prime.c` 的完整列表 13

2.1.2 `pthr_prime.c` 的完整列表 15

2.2 创建实验 19

2.2.1 对源代码进行校验 19

2.2.2 创建数据争用检测实验 19

2.2.3 检查数据争用检测实验 20

2.3 了解实验结果 21

2.3.1 `omp_prime.c` 中的数据争用 21

2.3.2 `pthr_prime.c` 中的数据争用 23

2.4 诊断数据争用的原因 27

2.4.1 检查数据争用是否为误报 27

2.4.2 检查数据争用是否为良性 28

2.4.3 修复错误而不是修复数据争用 28

2.5 误报 31

2.5.1 用户定义的同步 31

2.5.2 由不同线程再循环的内存 32

2.6 良性数据争用 33

2.6.1 用于查找素数的程序 33
2.6.2 用于验证数组值类型的程序 34
2.6.3 使用双检锁的程序 35

3 死锁教程 37
3.1 哲人进餐源文件 37
3.2 哲人进餐方案 40
3.2.1 哲人怎样发生死锁 41
3.2.2 为哲人 1 引入休眠时间 41
3.3 如何使用线程分析器查找死锁 44
3.3.1 对源代码进行编译 44
3.3.2 创建死锁检测实验 44
3.3.3 检查实验结果 44
3.4 了解实验结果 45
3.4.1 检查出现死锁的运行 46
3.4.2 检查存在潜在死锁但仍可完成的运行 49
3.5 修复死锁并了解误报 52
3.5.1 使用令牌控制哲人 53
3.5.2 另一种令牌机制 57

A 线程分析器用户 API 61
A.1 线程分析器的用户 API 61
A.2 其他可识别的 API 62
A.2.1 POSIX 线程 API 62
A.2.2 Solaris 线程 API 63
A.2.3 内存分配 API 64
A.2.4 OpenMP API 64

B 线程分析器常见问题 65
B.1 常见问题 65

索引 69

前言

《线程分析器用户指南》介绍了线程分析器工具，并提供了两套详细的教程。一套教程着重讨论死锁检测，另一套主要讨论数据争用检测。本手册中还提供了常见问题解答和受支持 API 的附录。

印刷约定

下表介绍了本书中的印刷约定。

表 P–1 印刷约定

字体	含义	示例
`AaBbCc123`	命令、文件和目录的名称；计算机屏幕输出	编辑 `.login` 文件。 使用 `ls -a` 列出所有文件。 `machine_name% you have mail.`
`AaBbCc123`	用户键入的内容，与计算机屏幕输出的显示不同	`machine_name%` **`su`** `Password:`
aabbcc123	要使用实名或值替换的命令行占位符	删除文件的命令为 `rm` *filename*。
AaBbCc123	保留未译的新词或术语以及要强调的词	这些称为 *Class* 选项。 **注意**：有些强调的项目在联机时以粗体显示。
新词术语强调	新词或术语以及要强调的词	**高速缓存**是存储在本地的副本。 请**勿**保存文件。
《书名》	书名	阅读《用户指南》的第 6 章。

命令中的 shell 提示符示例

下表列出了 C shell、Bourne shell 和 Korn shell 的缺省 UNIX® 系统提示符和超级用户提示符。

表 P–2 Shell 提示符

Shell	提示符
C shell	`machine_name%`
C shell 超级用户	`machine_name#`
Bourne shell 和 Korn shell	`$`
Bourne shell 和 Korn shell 超级用户	`#`

受支持的平台

此 Sun Studio 发行版支持使用 SPARC® 和 x86 系列处理器体系结构的系统：UltraSPARC®、SPARC64、AMD64、Pentium 和 Xeon EM64T。可从以下位置获得硬件兼容性列表，在列表中可以查看您正在使用的 Solaris 操作系统版本所支持的系统：http://www.sun.com/bigadmin/hcl。这些文档中给出了平台类型间所有实现的区别。

在本文档中，与 x86 相关的术语的含义如下：

- "x86" 泛指 64 位和 32 位的 x86 兼容产品系列。
- "x64" 表示有关 AMD64 或 EM64T 系统的特定 64 位信息。
- “32 位 x86”表示有关基于 x86 的系统的特定 32 位信息。

有关受支持的系统，请参阅硬件兼容性列表。

访问 Sun Studio 文档

可以访问以下位置的文档：

- 可以通过随软件一起安装在本地系统或网络上的文档索引（在 Solaris 平台上为 `file:/opt/SUNWspro/docs/index.html`；在 Linux 平台上为 `file:/opt/sun/sunstudio12/docs/index.html`）获取文档。

  如果未将软件安装在 Solaris 平台上的 `/opt` 目录中或 Linux 平台上的 `/opt/sun` 目录中，请询问系统管理员以获取系统中的等效路径。
- 可以从 `docs.sun.com`™ Web 站点获取大多数手册。下列书目只能从 Solaris 平台上已安装的软件中获取：
  - 《标准 C++ 库类参考》

- 《标准 C++ 库用户指南》
- 《Tools.h++ 类库参考》
- 《Tools.h++ 用户指南》

  相应发行说明可从 http://docs.sun.com Web 站点获取。

- IDE 所有组件的联机帮助可通过 IDE 中的“帮助”菜单以及许多窗口和对话框上的“帮助”按钮获取。

可以通过 Internet 访问 http://docs.sun.com Web 站点，阅读、打印和购买 Sun Microsystems 的各种手册。如果找不到某手册，请参见随软件一起安装在本地系统或网络上的文档索引。

注 – Sun 对本文档中提到的第三方 Web 站点的可用性不承担任何责任。对于此类站点或资源中的（或通过它们获得的）任何内容、广告、产品或其他资料，Sun 并不表示认可，也不承担任何责任。对于因使用或依靠此类站点或资源中的（或通过它们获得的）任何内容、产品或服务而造成的或连带产生的实际或名义损坏或损失，Sun 概不负责，也不承担任何责任。

采用易读格式的文档

该文档以易读格式提供，以方便残障用户使用辅助技术进行阅读。可以按照下表所述找到文档的易读版本。如果该软件未安装在 `/opt` 目录中，请询问系统管理员以获取系统中的等效路径。

文档类型	易读版本的格式和位置
手册（第三方手册除外）	HTML，位于 http://docs.sun.com
第三方手册： ■ 《标准 C++ 库类参考》 ■ 《标准 C++ 库用户指南》 ■ 《Tools.h++ 类库参考》 ■ 《Tools.h++ 用户指南》	HTML，位于 Solaris 平台上已安装软件中，可通过文档索引 (`file:/opt/SUNWspro/docs/index.html`) 获取
自述文件	HTML，位于 Sun Developer Network 门户 http://developers.sun.com/sunstudio/documentation/ss12

手册页	HTML，位于已安装软件中，可通过文档索引（在 Solaris 平台上为 file:/opt/SUNWspro/docs/index.html；在 Linux 平台上为 file:/opt/sun/sunstudio12/docs/index.html）获取
联机帮助	HTML，可通过 IDE 中的“帮助”菜单和“帮助”按钮获取
发行说明	HTML，位于 http://docs.sun.com

相关 Sun Studio 文档

下表列出了可通过 file:/opt/SUNWspro/docs/index.html 和 http://docs.sun.com 获取的相关文档。如果该软件未安装在 /opt 目录中，请询问系统管理员以获取系统中的等效路径。

文档标题	说明
《性能分析器》	介绍如何使用性能分析器软件诊断和调整软件。
《C 用户指南》	提供了所有编译器选项的参考、支持的 ISO/IEC 9899:1999（称为 C99）功能的说明、实现细节（如 pragma 和声明说明符）以及有关使用 lint 代码检查程序的完整信息。
《C++ 用户指南》	介绍了如何使用 C++ 编译器，还提供了有关命令行编译器选项、程序组织、pragma、模板、异常处理、使用强制类型转换运算符以及使用和生成库的详细信息。
《Fortran 编程指南》	介绍了如何在 Solaris 环境中编写高效 Fortran 程序；并介绍了输入/输出、库、性能、调试和并行处理。
《Fortran 库参考》	详细介绍 Fortran 库以及内函数。
《OpenMP API 用户指南》	概括介绍了 OpenMP 多重处理 API，并提供了有关实现的具体信息。
《数值计算指南》	介绍了与浮点计算的数值精度有关的问题。

访问 Solaris 相关文档

下表列出了可从 docs.sun.com Web 站点上获取的相关文档。

文档集合	文档标题	说明
Solaris Reference Manual Collection	请参见手册页各章节的标题。	提供 Solaris 操作系统的有关信息。
Solaris Software Developer Collection	《链接程序和库指南》	介绍了 Solaris 链接编辑器和运行时链接程序的操作。
Solaris Software Developer Collection	《多线程编程指南》	介绍了 POSIX 和 Solaris 线程 API、使用同步对象进行编程、编译多线程程序和多线程程序的查找工具。

开发者资源

访问 Sun Developer Network Sun Studio 门户 (http://developers.sun.com/sunstudio) 查看下列经常更新的资源：

- 有关编程技术和最佳做法的文章
- 有关编程小技巧的知识库
- 软件文档以及随软件一起安装的文档的更正信息
- 有关支持级别的信息
- 用户论坛
- 可下载的代码示例
- 新技术预览

Sun Studio 门户是 Sun Developer Network Web 站点 (http://developers.sun.com) 上面向开发者的众多其他资源之一。

联系 Sun 技术支持

如果您遇到通过本文档无法解决的技术问题，请访问以下 URL：

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

Sun 欢迎您提出意见

Sun 致力于提高其文档的质量，并十分乐意收到您的意见和建议。您可以通过以下 URL 向 Sun 提交您的意见：

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

请在电子邮件的主题行中注明文档的文件号码。例如，本文档的文件号码是 820-0619。

◆ ◆ ◆ 第 1 章

什么是线程分析器？它有何作用？

线程分析器是一种可用来分析多线程程序执行的工具。它可以检测多线程编程错误，如代码中的数据争用或死锁，其中代码是使用 POSIX 线程 API、Solaris 操作系统(R) 线程 API、OpenMP 指令、Sun 并行指令、Cray(R) 并行指令或混合使用前面各项编写的。

1.1 线程分析器入门

可以使用新增的 `tha` 命令启动线程分析器。线程分析器界面针对多线程程序分析进行了简化，从而不再显示传统的分析器选项卡，而是显示新的 Races（争用）、“死锁”、“双重数据源”、Race Details（争用详细信息）以及“死锁详细信息”选项卡。如果使用分析器查看相同的多线程程序实验，则将看到传统的分析器选项卡（如“函数”、“调用者与被调用者”、“反汇编”以及新增的选项卡。

线程分析器支持以下硬件和操作系统：

- SPARC(R) v8plus、v8plusa、v8plusb、v9、v9a 和 v9b 体系结构
- Intel(R) x86 和 AMD(R) x64 平台
- Solaris 9 和 Solaris 10 操作系统
- SuSE Linux Enterprise Server 9 和 Red Hat Enterprise Linux 4 操作系统

1.2 什么是数据争用？

线程分析器检测在执行多线程进程的过程中发生的数据争用。在以下情况下会发生数据争用：

- **单进程**中的两个或多个线程同时访问同一内存位置，且
- 至少一个访问用于写入，且
- 线程未使用任何互斥锁控制其对该内存的访问。

这三个条件成立时，访问顺序是不确定的，在不同运行中计算提供的结果可能随该顺序而异。有些数据争用可能是良性的（例如，当内存访问用于忙等待时），但是很多数据争用都是程序中的错误。

线程分析器适用于使用 POSIX 线程 API、Solaris 线程 API、OpenMP、Sun 并行指令、Cray 并行指令或上述项的混合编写的多线程程序。

1.3 什么是死锁？

死锁描述两个或多个线程因相互等待而被永远阻塞（挂起）的情况。导致死锁的原因有多种。线程分析器可检测到因不正确使用互斥锁而导致的死锁。这种类型的死锁在多线程应用程序中比较常见。以下条件成立时，具有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了一个循环链，其中每个线程等待链中下一线程持有的锁

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

死锁可能属于两种类型：**潜在**死锁或**实际**死锁。潜在死锁不一定在给定运行中发生，但是它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和线程的锁定请求的时限。实际死锁是在执行程序的过程中发生的死锁。实际死锁会导致所涉及的线程挂起，但是可能会也可能不会导致整个进程挂起。

1.4 线程分析器使用模型

以下步骤说明可以使用线程分析器解决多线程程序问题的过程。

1. 对程序进行校验。有关更多信息，请参见第 19 页中的 “2.2.1 对源代码进行校验”。
2. 进行试验，然后使用不同的因素重复试验，如不同的输入数据、不同数目的线程、不同的循环计划，甚至不同的硬件。此重复有助于找出根源不确定的问题。
3. 确定线程分析器揭示的多线程编程冲突是合法错误还是良性现象。
4. 修复合法错误并重复试验。
5. 如果线程分析器报告新的多线程编程冲突，请重复前面的两个步骤。

第 2 章

数据争用教程

以下是有关如何使用线程分析器检测和修复数据争用的详细教程。本教程由以下部分组成：

- 第 13 页中的 “2.1 教程源文件”
- 第 19 页中的 “2.2 创建实验”
- 第 21 页中的 “2.3 了解实验结果”
- 第 27 页中的 “2.4 诊断数据争用的原因”
- 第 31 页中的 “2.5 误报”
- 第 33 页中的 “2.6 良性数据争用”

2.1 教程源文件

本教程依赖两个包含数据争用的程序：

- 第一个程序查找素数。它是用 C 编写的，是使用 OpenMP 指令并行化的。源文件名为 omp_prime.c。
- 第二个程序也查找素数，并且也是用 C 编写的。但是， 它是使用 POSIX 线程而不是 OpenMP 指令并行化的。源文件名为 pthr_prime.c。

2.1.1 omp_prime.c 的完整列表

```
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 pthr_prime.c 的完整列表

```
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 omp_prime.c 和 pthr_prime.c 中的数据争用

如第 13 页中的 “2.1.1 omp_prime.c 的完整列表”所述，当代码包含争用情况且不同的运行提供不同的计算结果时，内存访问顺序是不确定的。由于代码中存在数据争用，所以每次执行 omp_prime.c 都会生成不正确且不一致的结果。下面显示了一个输出示例：

```
% 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
```

同样，由于 pthr_prime.c 中存在数据争用，所以程序的不同运行可能生成不正确且不一致的结果，如下所示。

```
% 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 创建实验

线程分析器沿用与 Sun Studio 性能分析器相同的“收集分析”模型。使用线程分析器包括以下三个步骤：

- 第 19 页中的“2.2.1 对源代码进行校验”
- 第 19 页中的“2.2.2 创建数据争用检测实验”
- 第 20 页中的“2.2.3 检查数据争用检测实验”

2.2.1 对源代码进行校验

为了在程序中启用数据争用检测，必须首先使用特殊的编译器选项编译源文件。对 C、C++ 和 Fortran 语言来说，此特殊选项是：`-xinstrument=datarace`

将 `-xinstrument=datarace` 编译器选项添加到现有的用来编译程序的一组选项。只能将该选项应用于您怀疑有数据争用的源文件。

注 – 确保在编译程序时指定 `-g`。为检测争用而编译程序时，不要指定高优化级别。使用 `-xopenmp=noopt` 编译 OpenMP 程序。使用高优化级别时，报告的信息（如行号和调用栈）可能是不正确的。

以下是对源代码进行校验的示例命令：

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

2.2.2 创建数据争用检测实验

将 `collect` 命令与 `-r on` 标志一起使用，以运行程序并在执行过程中创建数据争用检测实验。对于 OpenMP 程序，请确保所用的线程超过一个。以下是创建数据争用实验的示例命令：

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

为增大检测到数据争用的可能性，建议将 `collect` 与 `r race` 标志一起使用，以创建若干个数据争用检测实验。在不同的实验中使用不同的线程数和不同的输入数据。

2.2.3 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；前者提供的是一组简化的缺省选项卡，但在其他方面与性能分析器完全相同。

线程分析器 GUI 具有菜单栏、工具栏和包含各种选项卡的拆分窗格（不同选项卡对应不同的显示）。在左窗格上，缺省情况下显示以下三个选项卡：

- Races（争用）选项卡显示在程序中检测到的数据争用的列表。缺省情况下此选项卡处于选中状态。
- “双重数据源”选项卡显示对应于所选数据争用的两次访问的两个源位置。突出显示其中发生数据争用访问的源代码行。
- “试验”选项卡显示实验中的装入对象，并列出错误和警告消息。

在线程分析器显示屏的右窗格上，显示以下两个选项卡：

- “摘要”选项卡显示有关在 Races（争用）选项卡中选择的数据争用访问的摘要信息。
- Race Details（争用详细信息）选项卡显示有关在 Races（争用）选项卡中选择的数据争用跟踪的详细信息。

另一方面，`er_print` 实用程序提供命令行界面。在使用 `er_print` 实用程序检查争用时，以下子命令很有用：

- `-races`：它报告实验所显示的任何数据争用。
- `-rdetail race_id`：它显示有关具有指定 `race_id` 的数据争用的详细信息。如果指定的 `race_id` 为 "all"，将显示有关所有数据争用的详细信息。
- `-header`：它显示有关实验的描述性信息，并报告所有错误或警告。

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

2.3 了解实验结果

本部分说明如何使用 er_print 命令行和线程分析器 GUI 显示有关检测到的每个数据争用的以下信息：

- 数据争用的唯一 ID。
- 与数据争用关联的虚拟地址 Vaddr。如果存在多个虚拟地址，则在圆括号中显示标签“多个地址”。
- 两个不同线程对虚拟地址 Vaddr 的内存访问。显示访问类型（读取或写入），以及源代码中发生访问处的函数、偏移量和行号。
- 与数据争用关联的跟踪总数。每个跟踪都引用发生两个数据争用访问时的线程调用栈对。如果使用 GUI，则选择单个跟踪时 Race Details（争用详细信息）选项卡中将显示这两个调用栈。如果使用 er_print 实用程序，则 rdetail 命令将显示这两个调用栈。

2.3.1 omp_prime.c 中的数据争用

```
% 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)
```

以下屏幕快照显示在 omp_primes.c 中检测到的争用，与线程分析器 GUI 显示的相同。调用 GUI 并装入实验数据的命令是 tha test.1.er。

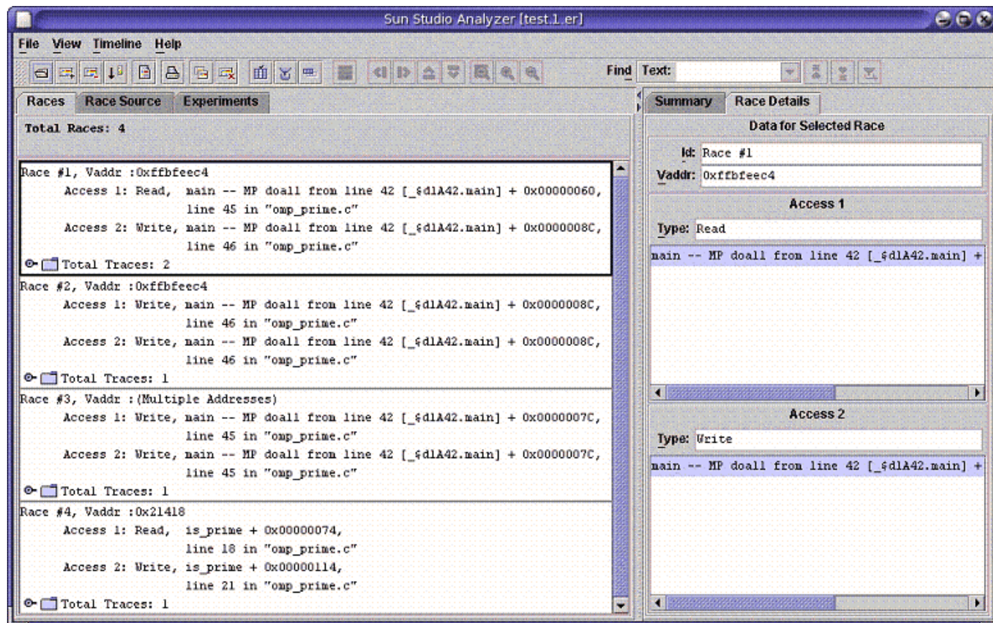


图 2–1 在 omp_primes.c 中检测到的数据争用

在 omp_primes.c 中有以下四个数据争用：

- 一号争用：第 45 行上 total 的读取和第 46 行上 total 的写入之间的数据争用。
- 二号争用：第 46 行上 total 的写入和同一行上 total 的另一写入之间的数据争用。
- 三号争用：第 45 行上 primes[] 的写入和同一行上 primes[] 的另一写入之间的数据争用。
- 四号争用：第 18 行上 pflag[] 的读取和第 21 行上 pflag[] 的写入之间的数据争用。

2.3.2 pthr_prime.c 中的数据争用

```
% 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)
```

以下屏幕快照显示在 pthr_primes.c 中检测到的争用，与线程分析器 GUI 显示的相同。调用 GUI 和装入实验数据的命令是
tha test.2.er。

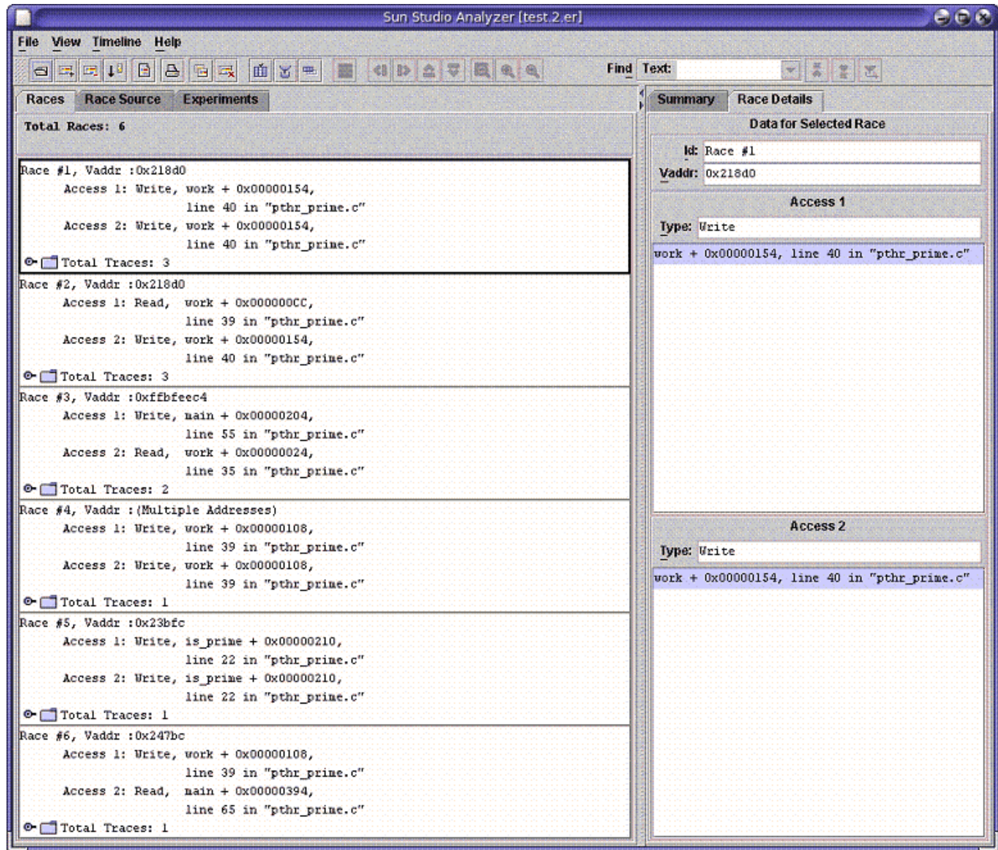


图 2–2 在 pthr_primes.c 中检测到的数据争用

在 pthr_prime.c 中有以下六个数据争用：

- 一号争用：第 40 行上 total 的写入和同一行上 total 的另一写入之间的数据争用。
- 二号争用：第 39 行上 total 的读取和第 40 行上 total 的写入之间的数据争用。
- 三号争用：第 55 行上 i 的写入和第 35 行上 i 的读取之间的数据争用。
- 四号争用：第 39 行上primes[] 的写入和同一行上primes[] 的另一写入之间的数据争用。
- 五号争用：第 22 行上 pflag[] 的写入和同一行上 pflag[] 的另一写入之间的数据争用
- 六号争用：第 39 行上 primes[] 的写入和第 65 行上 primes[] 的读取之间的数据争用。

GUI 的一个优势在于，它允许您并排查看与数据争用关联的两个源位置。例如，在 Races（争用）选项卡中选择 pthr_prime.c 的六号争用，然后单击“双重数据源”选项卡。您将看到以下内容：

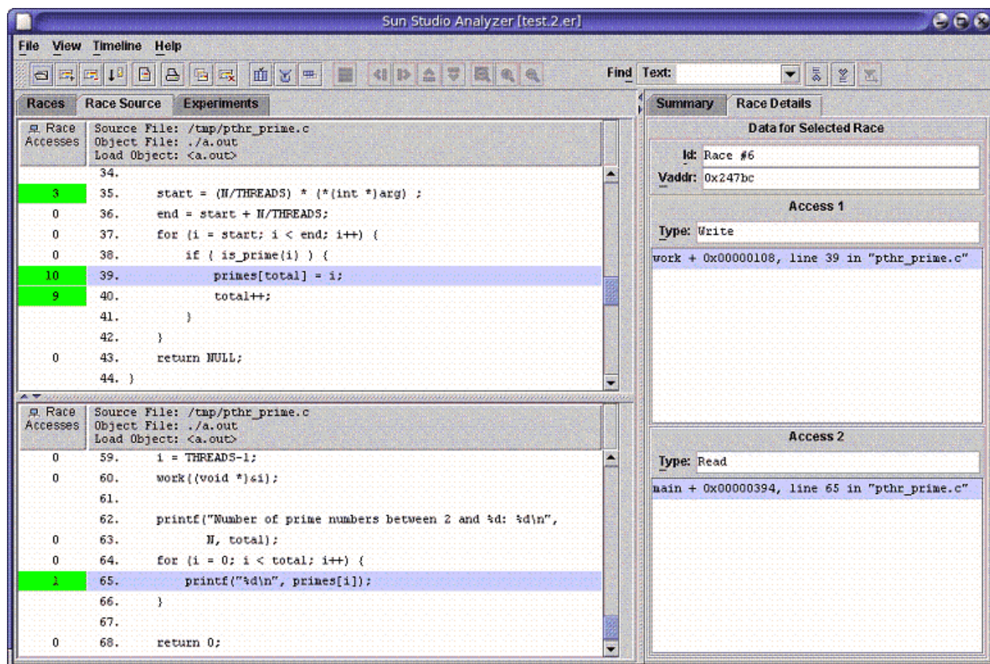


图 2–3 数据争用的源位置详细信息

在顶部 "Race Source" 窗格中显示六号争用（第 39 行）的第一次访问，在底部窗格中则显示该数据争用的第二次访问。突出显示其中发生数据争用访问的源代码（第 39 行和第 65 行）。在每个源代码行的左侧显示缺省度量（互斥争用访问度量）。该度量显示在该行上报告的数据争用访问次数。

2.4 诊断数据争用的原因

此部分提供诊断数据争用原因的基本策略。

2.4.1 检查数据争用是否为误报

误报数据争用是线程分析器报告了实际上未发生的数据争用。线程分析器尝试减少误报数。但是，存在该工具无法执行准确的作业并可能误报数据争用的情况。

可以忽略误报的数据争用，因为它不是真正的数据争用，因此不会影响程序的行为。

有关误报数据争用的一些示例，请参见 第 31 页中的 “2.5 误报” 。有关如何避免误报数据争用的信息，请参见第 61 页中的 “A.1 线程分析器的用户 API” 。

2.4.2 检查数据争用是否为良性

良性数据争用是指其存在不会影响程序正确性的有意数据争用。

有些多线程应用程序会有意使用可能导致数据争用的代码。由于那里的数据争用是设计使然，因此无需进行修复。但是，在某些情况下，使这样的代码正确运行是相当棘手的。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 第 31 页中的 “2.5 误报” 。

2.4.3 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，也无法建议如何修复所找到的数据争用。数据争用也可能是由错误引入的。找到并修复错误是很重要的。仅仅消除数据争用并不是正确的方法，这样做可能会使进一步调试变得更加困难。修复错误而不是修复数据争用。

2.4.3.1 修复 omp_prime.c 中的错误

以下说明如何修复 omp_prime.c 中的错误。有关完整的文件列表，请参见 第 13 页中的 “2.1.1 omp_prime.c 的完整列表” 。

将第 45 行和第 46 行移动到临界段中，以便消除第 45 行上 total 的读取和第 46 行上 total 的写入之间的数据争用。临界段保护这两行并防止数据争用。以下是更正后的代码：

```
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       }
```

请注意，添加单个临界段还修复了 omp_prime.c 中的两个其他数据争用。它修复了第 45 行上 prime[] 的数据争用，以及第 46 行上 total 的数据争用。第 18 行上 pflag[] 的读取和第 21 行上 pflag[] 的写入之间的第四个数据争用实际上是良性争用，因为它不会导致不正确的结果。修复良性数据争用不是必需的。

还可以将第 45 行和第 46 行移动到临界段中，如下所示，但是此更改将无法更正程序：

```
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      }
```

第 45 行和第 46 行周围的临界段消除了数据争用，因为线程不使用任何互斥锁控制其对 total 的访问。第 46 行周围的临界段确保 total 的计算值是正确的。但是，程序仍是不正确的。两个线程可能使用 total 的同一值更新 primes[] 的同一元素。此外，primes[ ] 中的某些元素可能根本未赋值。

2.4.3.2 修复 pthr_prime.c 中的错误

以下说明如何修复 pthr_prime.c 中的错误。有关完整的文件列表，请参见 第 15 页中的 "2.1.2 pthr_prime.c 的完整列表"。

使用单个互斥锁消除 pthr_prime.c 中第 39 行上 total 的读取和第 40 行上 total 的写入之间的数据争用。此添加还修复了 pthr_prime.c 中的两个其他数据争用：第 39 行上 prime[] 的数据争用以及第 40 行上 total 的数据争用。

第 55 行上 i 的写入和第 35 行上 i 的读取之间的数据争用以及第 22 行上 pflag[] 的数据争用显示了不同线程对变量 i 进行共享访问的问题。pthr_prime.c 中的初始线程在循环中创建子线程（源代码的第 55-57 行），并调度它们处理函数 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，而是将随初始线程递增循环索引而改变。由于不同线程使用 i 的同一值，因此发生了数据争用。

修复该问题的一种方法是将 i 按值传递到 work()。这确保每个线程都具有自己的带有唯一值的专用 i 副本。要消除第 39 行上写入访问和第 65 行上读取访问之间的 primes[] 数据争用，可以使用与前面第 39 行和第 40 行所用相同的互斥锁保护第 65 行。但是，这不是正确的修复方法。真正的问题是，主线程可能会在子线程仍在函数 work() 中更新 total 和 primes[] 的同时报告结果（第 50 行到第 53 行）。使用互斥锁并不提供线程之间的正确排序同步。一种正确的修复方法是在打印出结果之前让主线程等待所有子线程加入。

以下是更正后的 pthr_prime.c 版本：

```
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 误报

有些情况下，线程分析器可能会报告程序中未真正发生的假数据争用。这些情况称为误报。大多数情况下，误报是由第 31 页中的 “2.5.1 用户定义的同步”或第 32 页中的 “2.5.2 由不同线程再循环的内存”导致的。

2.5.1 用户定义的同步

线程分析器可以识别由 OpenMP、POSIX 线程和 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，而且在代码包含这样的同步时可能会报告假的数据争用。例如，该工具无法使用 CAS 指令识别锁的实现，无法使用忙等待识别张贴和等待操作，等等。以下是某类误报的典型示例，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);
103   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() 调用通常在测试谓词的循环中进行，以防止程序错误和虚假唤醒。谓词的测试和设置通常由互斥锁进行保护。在上面的代码中，线程 1 在第 100 行针对变量 data 生成值，在第 102 行将 ready_flag 的值设置为一以指示已生成数据，然后调用 pthread_cond_signal() 以唤醒使用方线程（即线程 2）。线程 2 在循环中测试谓词 (*!ready_flag*)。它在发现已设置标志时，将使用第 205 行上的数据。

第 102 行上 ready_flag 的写入和第 201 行上 ready_flag 的读取是由同一互斥锁保护的，因此在这两个访问之间不存在数据争用，而且该工具可正确地对此进行识别。

第 100 行上 data 的写入和第 205 行上 data 的读取不受互斥锁的保护。但是，在程序逻辑中，第 205 行上的读取始终发生在第 100 行上的写入之后，原因是存在标志变量 ready_flag。因此，在这两个访问数据之间不存在数据争用。但是，如果在运行时实际上未调用对 pthread_cond_wait()（第 202 行）的调用，该工具将报告在两个访问之间存在数据争用。如果曾在执行第 201 行之前执行第 102 行，则执行第 201 行时，循环项测试失败并跳过第 202 行。该工具监视 pthread_cond_signal() 调用和 pthread_cond_wait() 调用，并且可以使它们成对以派生同步。如果未调用第 202 行上的 pthread_cond_wait()，则该工具不知道第 100 行上的写入始终是在第 205 行上的读取之前执行的。因此，它认为它们是并发执行的，并报告它们之间的数据争用。

为了避免误报此类数据争用，线程分析器提供了一组 API，可以用来在执行用户定义的同步时通知该工具。有关更多信息，请参见第 61 页中的 “A.1 线程分析器的用户 API”。

2.5.2 由不同线程再循环的内存

一些内存管理例程再循环线程释放的内存以供另一线程使用。线程分析器有时无法识别由不同线程使用的同一内存位置的使用期限不重叠。如果出现此情况，则该工具可能误报数据争用。以下示例说明此类误报。

```
/*----------*/                        /*----------*/
 /* 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 并发执行。每个线程都分配一个用作其专用内存的内存块。例程 `mymalloc()` 可能会将前一调用释放的内存提供给 `myfree()`。如果线程 2 在线程 1 调用 `myfree()` 之前调用 `mymalloc()`，则 `ptr1` 和 `ptr2` 将获取不同的值，因此这两个线程之间没有数据争用。但是，如果线程 2 在线程 1 调用 `myfree()` 之后调用 `mymalloc()`，则 `ptr1` 和 `ptr2` 可能具有相同值。由于线程 1 不再访问该内存，因此不存在数据争用。但是，如果该工具不知道 `mymalloc()` 正在再循环内存，则它将报告 `ptr1` 数据的写入和 `ptr2` 数据的写入之间存在数据争用。当 C++ 运行时库为临时变量再循环内存时，此类误报通常发生在 C++ 应用程序中。它通常还发生在实现自己的内存管理例程的用户应用程序中。当前，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。

2.6 良性数据争用

一些多线程应用程序有意允许数据争用，以便获得更佳性能。良性数据争用是指其存在不会影响程序正确性的有意数据争用。以下示例说明良性数据争用。

注 – 除了良性数据争用外，大量应用程序允许数据争用，因为它们依赖于很难正确设计的锁释放和等待释放算法。线程分析器可以帮助确定这些应用程序中存在数据争用的位置。

2.6.1 用于查找素数的程序

文件 `omp_prime.c` 中的线程通过执行函数 `is_prime()` 来检查一个整数是否为素数。

```
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 }
```

线程分析器报告在第 21 行上 pflag[] 的写入和第 18 行上 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 18 行上，线程检查对于 i 的给定值 pflag[i] 是否等于零。如果 pflag[i] 不等于零，则表明 i 是已知的合数（换句话说，知道 i 不是素数）。因此，无需检查 v 是否可被 i 整除；我们只需检查 v 是否可被某个素数整除。因此，如果 pflag[i] 等于零，则线程将继续执行 i 的下一个值。如果 pflag[i] 不等于零且 v 可被 i 整除，则线程将为 pflag[v] 分配零，以指示 v 不是素数。

从正确性方面看，多个线程检查同一 pflag[] 元素并同时向其写入是可以的。pflag[] 元素的初始值为一。当线程更新该元素时，它们为该元素分配零值。即，线程在该元素的同一内存字节的同一位中存储零。在当前的体系结构中，可以假定那些存储是原子的。这意味着，线程读取该元素时，读取的值要么为一，要么为零。如果线程在为其分配零值之前检查给定 pflag[] 元素（第 18 行），则它执行第 20-23 行。如果在此期间，另一线程为该同一 pflag[] 元素（第 21 行）分配零值，则最终结果不变。在本质上，这意味着第一个线程不必要地执行了第 20-23 行。

2.6.2 用于验证数组值类型的程序

一组线程并发调用 check_bad_array() 以检查数组 data_array 是否有元素已损坏。每个线程检查数组的不同部分。如果线程发现某元素已损坏，则它会将全局共享变量 is_bad 的值设置为 true。

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

 100  /*
 102   * 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 行上 is_bad 的读取和第 112 行上 is_bad 的写入之间存在数据争用。但是，该数据争用不会影响最终结果的正确性。

is_bad 的初始值为零。当线程更新 is_bad 时，它们为其分配值一。即，线程在 is_bad 的同一内存字节的同一位中存储一。在当前的体系结构中，可以假定那些存储是原子的。因此，当线程读取 is_bad 时，读取的值要么是零，要么是一。如果线程在为其分配值一之前检查 is_bad（第 108 行），则它继续执行 for 循环。如果在此期间，另一个线程为 is_bad（第 112 行）分配值一，则不会更改最终结果。这仅仅意味着，线程执行 for 循环的时间超过了所需时间。

2.6.3 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。双检锁是一种常用的有效方法，用于在多线程应用程序中初始化单件。以下代码说明这样的实现。

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

 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 }
```

ptr_instance 的读取（第 301 行）有意不受锁的保护。这使检查可以确定在多线程环境中是否已有效地实例化单件。请注意，在第 301 行上的读取和第 308 行上的写入之间存在对 ptr_instance 变量的数据争用，但是程序可正常工作。不过，编写允许数据争用的正确程序是一项艰难的任务。例如，在上面的双检锁代码中，第 302 行和第 307 行上对 memory_barrier() 的调用用于确保按正确的顺序设置和读取单件和 ptr_instance。因此，所有线程将一致地读取它们。如果未使用内存屏障，则此编程方法将失效。

◆ ◆ ◆ 第 3 章

死锁教程

本教程说明如何使用线程分析器在多线程程序中检测潜在的以及实际的死锁。术语“死锁”描述两个或更多线程由于互相等待而被永远阻塞（挂起）的情况。导致死锁的原因可能有多种，如错误的程序逻辑、不恰当使用同步和屏障等。此教程将重点讨论由于不恰当地使用互斥锁而造成的死锁。此类死锁通常出现在多线程应用程序中。以下三个条件成立时，具有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了一个循环链，其中每个线程等待链中下一线程持有的锁

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

死锁可以分为两种类型：**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运行中发生，但是它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和线程的锁定请求的时限。
- 实际死锁是在执行程序的过程中发生的死锁。实际死锁会导致所涉及的线程挂起，但是可能会也可能不会导致整个进程挂起。

3.1 哲人进餐源文件

模拟哲人进餐问题的样例程序是一个使用 POSIX 线程的 C 程序。源文件名为 `din_philo.c`。该程序可以同时展示潜在死锁和实际死锁。以下是代码列表，并附有解释：

```
/* din_philo.c */
     1  #include <pthread.h>
     2  #include <stdio.h>
```

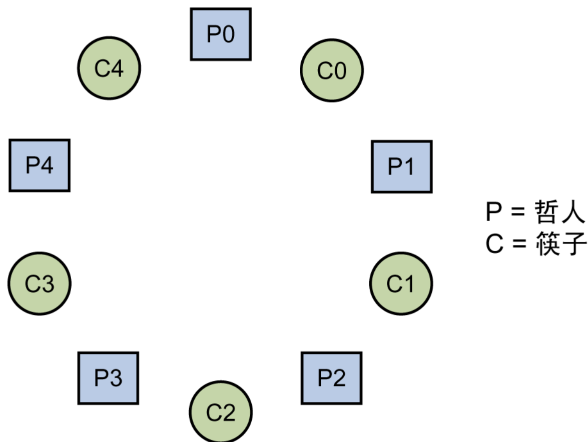
```
 3 #include <unistd.h>
 4 #include <stdlib.h>
 5 #include <errno.h>
 6 #include <assert.h>
 7
 8 #define PHILOS 5
 9 #define DELAY 5000
10 #define FOOD 50
11
12   void *philosopher (void *id);
13   void grab_chopstick (int,
14                        int,
15                        char *);
16   void 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 哲人进餐方案

哲人进餐方案是一个结构如下的经典方案：五个哲人，其编号为零到四，坐在圆桌旁思考。随着时间的推移，不同的个人感到饿了，决定去吃面条。桌子上有一个盛有面条的大浅盘，但是每个哲人只有一根筷子可以使用。为了吃面条，他们必须共用筷子。每个哲人左侧（哲人面向餐桌而坐）的筷子具有与该哲人相同的编号。



P = 哲人
C = 筷子

每个哲人首先去拿带有其编号的筷子。当他拿到指定给自己的筷子后，他将去拿指定给其邻桌的筷子。拿到两根筷子后，他就可以吃了。吃完后，他将筷子放回桌子上的原始位置，一侧放置一根。重复该过程，直到将面条吃完。

3.2.1 哲人怎样发生死锁

当每个哲人都持有他自己的筷子并等待邻座的筷子变为可用时，会发生实际死锁：

哲人 0 持有筷子 0，但等待筷子 1
哲人 1 持有筷子 1，但等待筷子 2
哲人 2 持有筷子 2，但等待筷子 3
哲人 3 持有筷子 3，但等待筷子 4
哲人 4 持有筷子 4，但等待筷子 0

在这种情况下，没有人可以吃，这些哲人处于死锁状态。多次重新运行该程序，您将看到该程序可能有时挂起，或有时运行直到完成。

运行哲人进餐程序，并查看它完成还是发生死锁。它可能挂起，如以下样例运行所示：

```
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 为哲人 1 引入休眠时间

潜在死锁的一种可能解决方案是，哲人 1 在拿他的筷子之前先等待。就代码而言，可以让他在拿自己的筷子之前休眠指定的时间 (`sleep_seconds`)。如果他休眠的时间足够长，则程序可能完成而不会发生实际死锁。您可以将他休眠的秒数作为可执行程序的参数进行指定。如果您不指定参数，则哲人不休眠。

下列伪代码显示了每个哲人的逻辑：

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

以下列表说明，如果哲人 1 在拿自己的筷子前等待 30 秒，程序是如何运行的。程序运行直到完成，所有五个哲人都吃完了。

```
% 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
```

尝试运行几次该程序，指定不同的休眠参数。如果哲人在拿自己的筷子前仅等待很短的时间，会发生什么情况？如果他等待更长的时间，又会怎么样？尝试为可执行的 a.out 指定不同的休眠参数。将该程序重新运行几次，使用或不使用休眠参数。有时程序会挂起，有时则可以完成运行。程序是否挂起取决于线程的调度，以及线程请求锁定的时间。

3.3 如何使用线程分析器查找死锁

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 Sun Studio 性能分析器相同的“收集分析”模型。使用线程分析器包括以下三个步骤：

- 对源代码进行编译。
- 创建死锁检测实验。
- 检查实验结果。

3.3.1 对源代码进行编译

编译代码，并确保指定了 -g。不要指定高优化级别，因为在高优化级别中可能会错误地报告行号和调用栈等信息。使用 -g -xopenmp=noopt 编译 OpenMP 程序，仅使用 -g -mt 编译 POSIX 线程程序。

有关更多信息，请参见 cc.1、CC.1 或 f95.1。

3.3.2 创建死锁检测实验

使用线程分析器的带有 -r deadlock 选项的 collect 命令。此选项将在执行程序的过程中创建死锁检测实验。

可通过创建多个死锁检测实验提高检测到死锁的可能性。为不同的实验使用不同的线程数和不同的输入数据。

有关更多信息，请参见 collect.1 和 collector.1。

3.3.3 检查实验结果

可以使用 tha 命令、analyzer 命令或 er_print 实用程序检查死锁检测实验。当 er_print 使用命令行界面时，线程分析器和分析器均代表 GUI 界面。

有关详细信息，请参见 tha.1、analyzer.1 和 er_print.1。

3.3.3.1 线程分析器界面

线程分析器包含菜单栏、工具栏和包含各种选项卡的拆分窗格（不同选项卡对应不同的显示）。在左窗格上，缺省情况下显示以下三个选项卡：

- “死锁”选项卡

  此选项卡显示线程分析器在程序中检测到的潜在死锁和实际死锁列表。缺省情况下此选项卡处于选中状态。显示每个死锁涉及的线程。这些线程构成了一个循环链，其中每个线程都占用一个锁，并请求使用链中下一个线程占用的另一个锁。

- “双重数据源”选项卡

  在循环链中选择线程，并单击“双重数据源”选项卡。“双重数据源”选项卡显示线程占用锁的源位置，以及同一线程请求锁的源位置。突出显示线程占用锁和请求锁的源代码行。

- “实验”选项卡

  此选项卡显示实验中的装入对象，并列出错误和警告消息。在线程分析器显示屏的右窗格上，显示以下两个选项卡：

  - “摘要”选项卡显示在“死锁”选项卡中选择的死锁的摘要信息。
  - “死锁详细信息”选项卡显示“死锁”选项卡中选择的线程上下文的详细信息。

3.3.3.2 er_print 界面

与左窗格相对应，右窗格包含了“死锁详细信息”选项卡，该选项卡中显示了“死锁”选项卡中选择的死锁的详细信息。使用 er_print 检查死锁的最有用的子命令如下：

- `-deadlocks`

  此选项报告在实验中检测到的任意潜在死锁和实际死锁。

- `-ddetail deadlock_id`

  此选项返回具有指定 `deadlock_id` 的死锁的详细信息。如果指定值 **`all`** 作为 `deadlock_id`，则 er_print 将显示所有死锁的详细信息。

- `-header`

  此选项显示有关实验的描述性信息，并报告所有错误或警告。

有关更多信息，请参见 `er_print.1`。

3.4 了解实验结果

本部分将详细说明如何使用线程分析器来检查哲人进餐程序中的死锁。首先将执行导致实际死锁的运行，然后将检查正常终止但有潜在死锁的运行。

3.4.1 检查出现死锁的运行

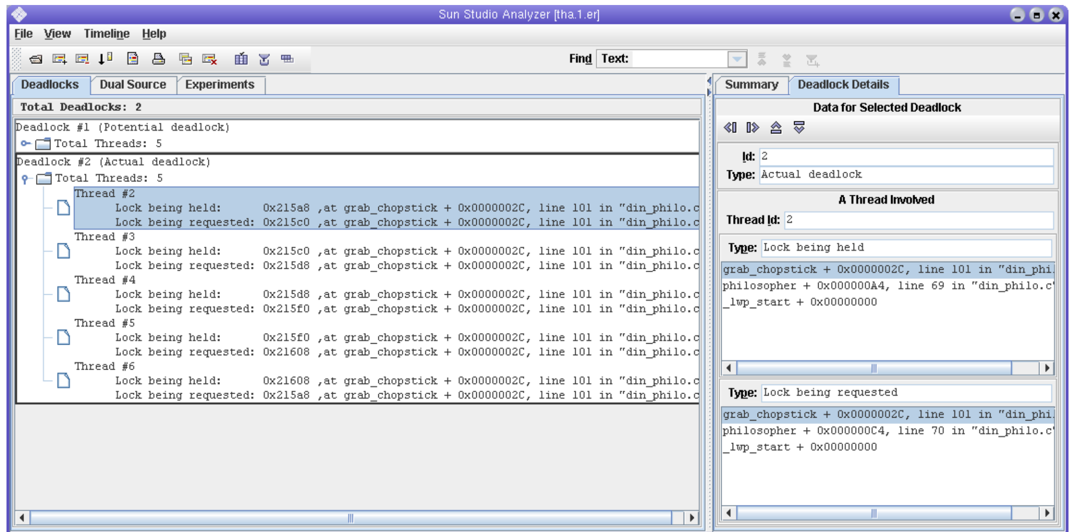
以下列表显示了导致实际死锁的哲人进餐程序的一个运行。

```
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)
```

下面的屏幕抓图给出了线程分析器中显示的死锁信息：

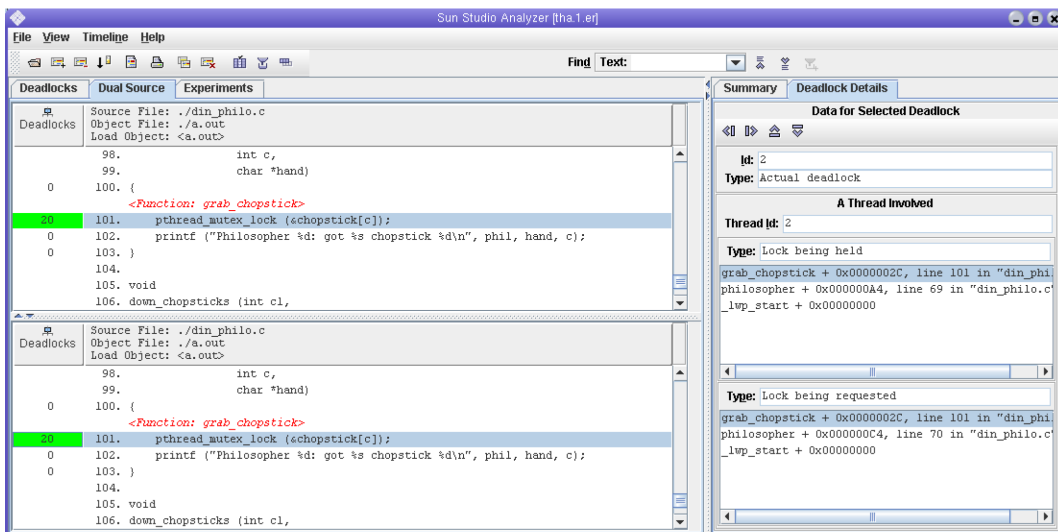


线程分析器报告了 `din_philo.c` 的两个死锁，一个潜在死锁和一个实际死锁。通过进一步地检查，我们发现这两个死锁是相同的。死锁中涉及的循环链如下：

线程 2：
在地址 `0x215a8` 占用锁，在地址 `0x215c0` 请求锁
线程 3：
在地址 `0x215c0` 占用锁，在地址 `0x215d8` 请求锁
线程 4：
在地址 `0x215d8` 占用锁，在地址 `0x215f0` 请求锁
线程 5：
在地址 `0x215f0` 占用锁，在地址 `0x21608` 请求锁

线程 6：
在地址 `0x21608` 占用锁，在地址 `0x215a8` 请求锁

选择链中的第一个线程（线程 #2），然后单击“双重数据源”选项卡，查看在源代码中的何处，线程 #2 占用锁（在地址 `0x215a8`），以及在何处请求锁（在地址 `0x215c0`）。以下屏幕抓图显示了 2 号线程的“双重数据源”选项卡。在每个源代码行的左侧显示了缺省度量（互斥死锁度量）。该度量提供（死锁中所涉及的）锁定占用或锁定请求操作在该代码行上报告的次数。



3.4.2 检查存在潜在死锁但仍可完成的运行

在提供足够大的休眠参数时，哲人进餐程序可避免实际死锁，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。它仅仅表示在一次特定的运行中，占用和请求的锁定不会构成死锁链。如果在其他运行中计时发生更改，则还会发生实际死锁。以下列表显示了正常终止的哲人进餐程序的一次运行。但是 er_print 实用程序和线程分析器会报告潜在死锁。

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

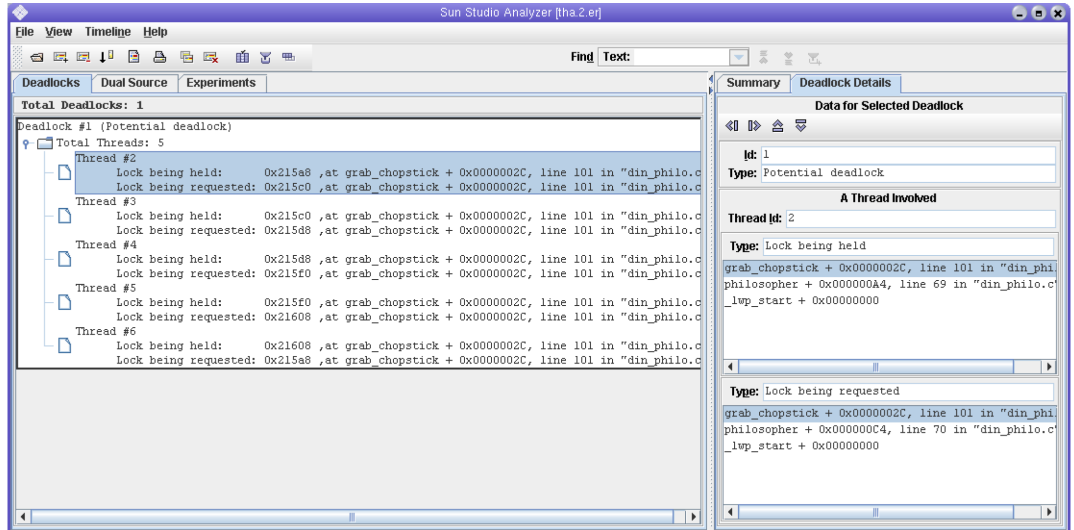
```
% collect -r deadlock a.out 40
Creating experiment database tha.2.er ...
Philosopher 0 is done thinking and now ready to eat.
Philosopher 2 is done thinking and now ready to eat.
Philosopher 1 is done thinking and now ready to eat.
Philosopher 3 is done thinking and now ready to eat.
Philosopher 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)
```

下面的屏幕抓图显示了线程分析器界面的潜在死锁信息：



3.5 修复死锁并了解误报

除了哲人在开始进餐前等待的策略，我们还可以使用令牌机制，在该机制中，哲人必须在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人的数量。当哲人收到令牌后，他可以按照餐桌规则尝试进餐。进餐后，每个哲人返还令牌并重复该过程。下列伪代码显示了使用令牌机制时每个哲人的逻辑：

```
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
  }
```

以下部分将详细说明令牌机制的两种不同实现方法。

3.5.1 使用令牌控制哲人

下面列出了使用令牌机制的哲人进餐程序的修正版本。该解决方案结合使用了 4 个令牌，比用餐者的数量少一，因此能够同时尝试进餐的哲人不会超过 4 个。该版本程序称为 `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++)
```

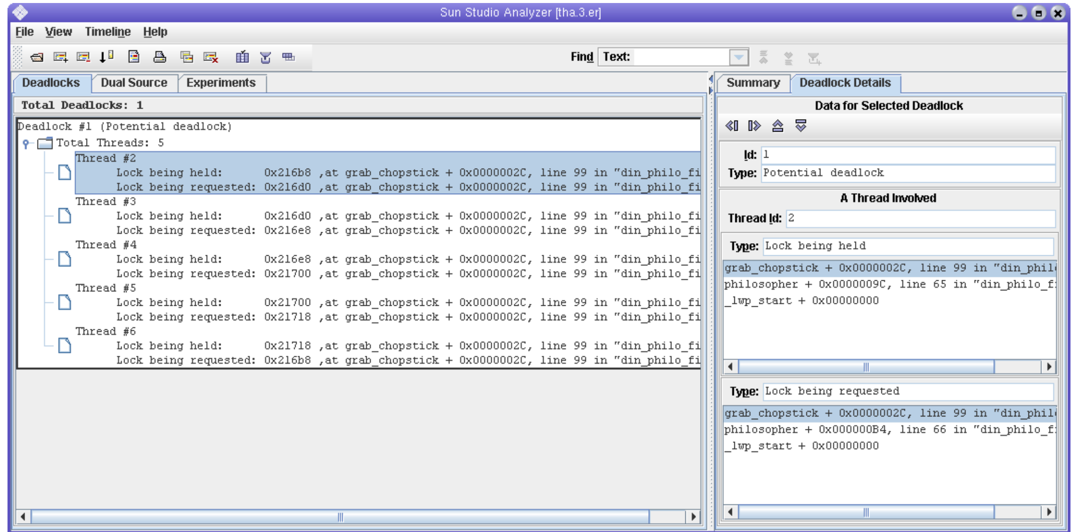
```
43          pthread_join (philo[i], NULL);
44      return 0;
45  }
46
47  void *
48  philosopher (void *num)
49  {
50      int id;
51      int i, left_chopstick, right_chopstick, f;
52
53      id = (int)num;
54      printf ("Philosopher %d is done thinking and now ready to eat.\n", id);
55      right_chopstick = 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     }
```

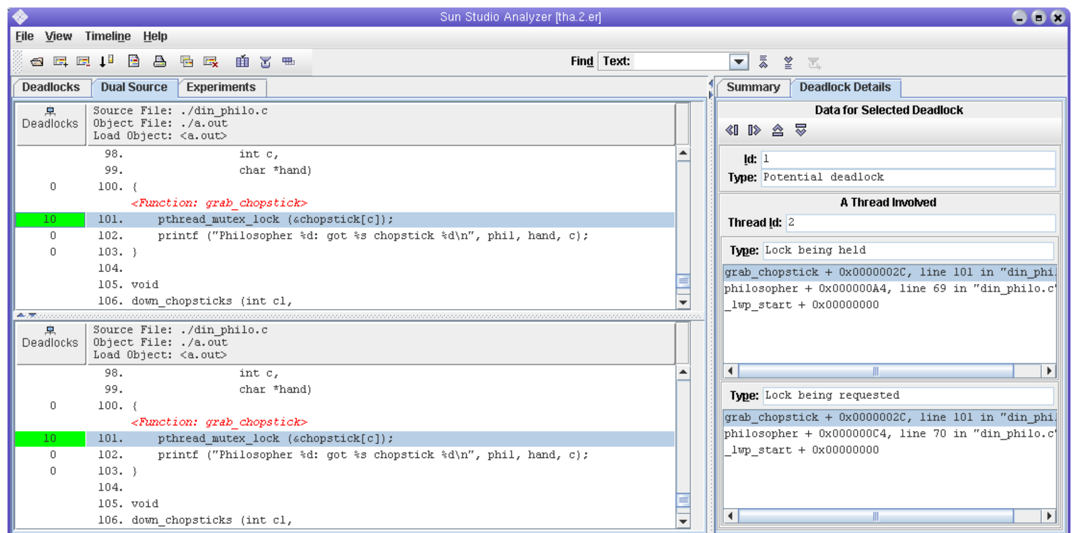
尝试编译并运行这一哲人进餐程序的修正版本，并多运行几次。令牌机制限制了尝试使用筷子的用餐者的数量，从而可避免实际死锁和潜在死锁。

3.5.1.1 误报的报告

尽管使用了令牌机制，线程分析器仍为此实现报告了一个潜在死锁（虽然并不存在死锁）。这是一个误报。考虑以下包含潜在死锁详细信息的屏幕抓图：



选择该链中的第一个线程（线程 #2），然后单击“双重数据源”选项卡，查看线程 #2 在源代码中占用锁的位置（在地址 `0x215a8`）以及请求锁的位置（在地址 `0x215c0`）。以下屏幕抓图显示了线程 #2 的“双重数据源”选项卡。



din_philo_fix1.c 中的 get_token() 函数使用 while 循环同步线程。在成功获得一个令牌（当 num_can_eat 大于 0 时将发生这种情况）之前，线程不会离开 while 循环。while 循环将同时进餐的人数限制为 4。但是线程分析器无法识别 while 循环实现的同步。它假定所有 5 个哲人都尝试同时取筷子并进餐，因此报告了一个潜在死锁。下一部分将详细说明如何使用线程分析器可识别的同步限制同时进餐的人数。

3.5.2 另一种令牌机制

以下列表显示了令牌机制的另一种实现。该实现仍使用 4 个令牌，因此同时尝试进餐的人数不会超过 4 个。但是，该实现使用 sem_wait() 和 sem_post() 信号例程来限制进餐的哲人数量。该版本的源文件称为 din_philo_fix2.c。

注 – 必须使用 -lrt 编译 din_philo_fix2.c，以链接正确的信号例程。

以下列表详细说明了 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
```

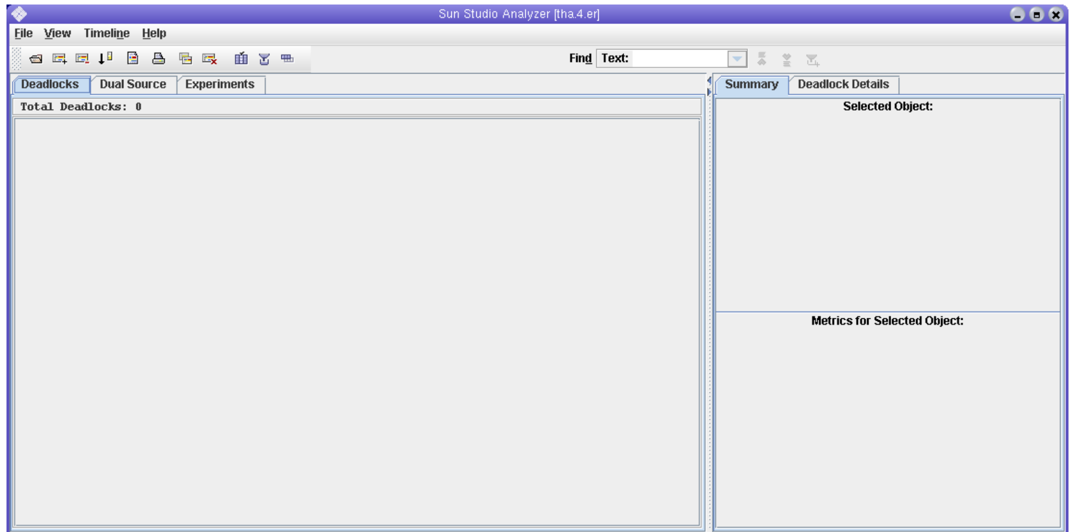
```
29
30  int
31  main (int argn,
32        char **argv)
33  {
34      int i;
35
36      pthread_mutex_init (&food_lock, NULL);
37      sem_init(&num_can_eat_sem, 0, PHILOS - 1);
38      for (i = 0; i < PHILOS; i++)
39          pthread_mutex_init (&chopstick[i], NULL);
40      for (i = 0; i < PHILOS; 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    }
```

这一新实现使用信号 num_can_eat_sem 来限制同时进餐的哲人数量。信号 num_can_eat_sem 初始化为 4，比哲人的数量少一。在尝试进餐前，哲人将调用

get_token()，从而会调用 sem_wait(&num_can_eat_sem)。调用 sem_wait() 将导致发出调用的哲人进入等待状态，直到信号值为正，然后通过从值中减 1 来更改信号的值。当一位哲人进完餐后，他将调用 return_token()，从而会调用 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，并确定并非所有哲人都会同时尝试进餐。

如果多次运行这一新的程序实现，您将发现该程序每次都会正常终止，不会挂起。此外，还将发现线程分析器不会报告任何实际死锁或潜在死锁，如下面的屏幕抓图所示：



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。

附录 A

线程分析器用户 API

线程分析器可以识别 OpenMP 指令、POSIX 线程和 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，而且在您使用这样的同步时可能报告假的数据争用。例如，该工具无法识别通过手动编码汇编语言代码实现的旋转锁定。

如果代码包括用户定义的同步，则将线程分析器支持的用户 API 插入到程序中以标识那些同步。此标识允许线程分析器识别同步并减少误报数。下面列出了用户 API：

A.1 线程分析器的用户 API

表 A–1 线程分析器用户 API

`tha_notify_acquire_lock(id)()`	在程序尝试获取用户定义的锁之前立即插入。
`tha_notify_lock_acquired(id)()`	在成功获取用户定义的锁之后立即插入。
`tha_notify_writelock_acquired(id)()`	在写入模式下成功获取用户定义的读写锁之后立即插入。
`tha_notify_readlock_acquired(id)()`	在读取模式下成功获取用户定义的读写锁之后立即插入。
`tha_notify_lock_released(id)()`	在成功释放用户定义的锁（包括读写锁）之后立即插入。
`tha_notify_sync_post_begin(id)()`	在执行用户定义的后同步之前立即插入。
`tha_notify_sync_post_end(id)()`	在执行用户定义的后同步之后立即插入。
`tha_notify_sync_wait_begin(id)()`	执行用户定义的等待同步之前立即插入。
`tha_notify_sync_wait_end(id)()`	执行用户定义的等待同步之后立即插入。

提供了 API 的 C/C++ 版本和 Fortran 版本。每个 API 调用都采用单个参数 id，其值应该唯一标识同步对象。

在 API 的 C/C++ 版本中，参数类型为 `uintptr_t`，在 32 位模式下其长度为 4 字节，在 64 位模式下其长度为 8 字节。调用任何 API 时，都需要将 `#include <tha_interface.h>` 添加到 C/C++ 源文件中。

在 API 的 Fortran 版本中，参数类型为整型 `tha_sobj_kind`，在 32 位和 64 位模式下其长度都为 8 字节。调用任何 API 时，都需要将 `"tha_finterface.h"` 添加到 Fortran 源文件中。要唯一标识同步对象，每个不同同步对象的参数 ID 应具有不同的值。一种执行此操作的方法是将同步对象的地址值用作 ID。以下代码示例说明如何使用 API 避免误报的数据争用：

```
# 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。

A.2 其他可识别的 API

以下几部分详细介绍线程分析器可识别的 API：

A.2.1 POSIX 线程 API

```
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 线程 API

```
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 内存分配 API

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

A.2.4 OpenMP API

有关更多信息，请参见 《Sun Studio 12: OpenMP API User's Guide》。

附录 B

线程分析器常见问题

本节包括常见问题及其解答的列表。有关对此常见问题的最新更新，请参见Sun Developer Network (`http://developers.sun.com/sunstudio/index.jsp`)。

B.1 常见问题

问题: 为什么行号信息不正确?

答案: 尝试关闭优化，或指定级别 `-xO3` 或更低。编译器的优化变换可能会使行号信息失真，并使实验结果变得难以阅读。

问题: 我是否真的需要安装 `collect` 命令缺少的修补程序?

答案: 是。确保实验系统安装了所有必需修补程序。如果缺少任何必需的修补程序，则实验结果可能会不正确。

问题: 是否可以使用我的代码链接 `malloc()` 库的归档版本?

答案: 否。线程分析器放在 `malloc()` 例程上，因此链接 `malloc()` 库的归档版本可能会导致误报数据争用。

问题: 线程分析器是否可以检测 OpenMP 应用程序中的数据争用? 对于 POSIX 或 Solaris 线程应用程序又怎么样呢?

答案: 线程分析器可以检测在使用 POSIX 线程 API、Solaris 操作系统(R) 线程 API、OpenMP 指令、Sun 并行指令、Cray(R) 并行指令或这些项的混合编写的代码中发生的数据争用。

问题: 线程分析器是否可以检测不同进程之间的数据争用?

答案: 还不可以。当前，它仅检测从一个进程产生的不同线程之间的数据争用。

问题: 线程分析器是否能够找出所有数据争用?

答案: 否。线程分析器在运行时检测数据争用，应用程序的确切运行时行为取决于输入数据集。特定的输入数据集可能并不导致数据争用。线程分析器按高级别模拟线程之间的并发性，以便最大限度地减少操作系统调度的影响。但是，操作系统调度仍可以影响内存分配和存储重用，而这会使潜在的数据争用发生变化。

将线程分析器与不同数目的线程和不同的输入数据集一起使用，并对单个数据集重复实验，以最大限度地增加该工具检测数据争用的可能性。

问题: 为什么在不同的运行中线程分析器提供的数据争用结果是不同的?

答案: 之所以出现此现象，是因为运行之间存在计时差异。在不同运行中线程按不同的顺序访问内存时，将会报告不同的数据争用结果。

问题: 为什么线程分析器报告在我的应用程序中不存在的数据争用? 如何删除它们?

答案: 在一些情况下，线程分析器可能会报告在程序中从未实际发生过的数据争用。这种情况称为误报，它们通常在使用用户实现的同步或在线程之间再循环内存时发生。例如，如果代码包括实现旋转锁的手动编码程序集，则线程分析器将无法识别这些同步点。有关误报的详细说明以及如何通过 API 调用删除它们的示例，请参见教程。

问题: 什么是 `librdthooks.so`? 它有何作用?

答案: `librdthooks.so` 是满足数据争用检测校验调用和用户 API 调用的入口点的库。编译程序并将其与 `-xinstrument=datarace` 链接时，将自动链接它。有关更多信息，请参见 `librdthooks(3)` 手册页。

问题: 我如何知道可执行文件或库是否已经过校验?

答案: 使用 `nm`。有关更多详细信息，请参见 `nm(1)` 手册页。如果找到全局未定义符号 `__rdt_src_read` 或 `__rdt_src_write`，则可执行文件或库已经过校验。

问题: 是否可以使用分析器读取数据争用实验?

答案: 可以，分析器显示所有的传统性能分析选项卡，以及新增的 Races（争用）、Races Source（争用源）和 Race Details（争用详细信息）选项卡。线程分析器界面经过了简化，不显示传统的分析器选项卡。

问题: 为什么将它与 C、C++ 或 F90 一起使用时，出现一条错误消息，指出编译器选项 `-xinstrument=datarace` 是错误的?

答案: 您使用的是不支持线程分析器的旧版本 Sun Studio。通过输入以下内容，检查您所使用的 Sun Studio 版本：`cc -Version`。必须使用 2006 年 6 月以后的版本。

问题: 为什么使用 `er_print` 实用程序时出现一条错误消息，指出争用命令无效?

答案: 您使用的是不支持线程分析器的旧版本 Sun Studio。通过输入以下内容，检查您所使用的 Sun Studio 版本：`er_print -V`。必须使用不早于 2006 年 6 月的版本。

问题: 为什么在运行 `collect -r` 时出现一条错误消息，指出无法识别 -r?

答案: 您使用的是不支持线程分析器的旧版本 Sun Studio。通过输入 `collect -V` 检查您所使用的 Sun Studio 版本。必须使用不早于 2006 年 6 月的版本。

问题: 如何报告错误或与他人共享我的线程分析器体验?

答案: 将您的反馈与线程分析器工程师和用户共享的最佳方法是，读取并张贴到 Sun Studio 工具论坛 (`http://developers.sun.com/sunstudio/community/forums.jsp`)。您可能会发现已经解答了您的问题。

索引

文
文档, 访问，6-8, 8
文档索引，6

线
线程分析器, 入门，19

易
易读文档，7-8