多线程编程指南

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

文件号码 819–7051–10
2006 年 10 月

061017@15490

目录

前言

《**多线程编程指南**》介绍了 Solaris™ 操作系统 (Solaris Operating System, Solaris OS）中 POSIX*线程和 Solaris 线程的多线程编程接口。本指南将指导应用程序程序员如何创建新的多线程程序以及如何向现有的程序中添加多线程。

尽管本指南同时介绍了 POSIX 线程接口和 Solaris 线程接口，但大多数主题都以 POSIX 线程为重点。仅适用于 Solaris 线程的信息将专门在一章中介绍。

要理解本指南，读者必须熟悉并发编程的概念：

- UNIX® SVR4 系统－首选是 Solaris 发行版。
- C 编程语言－多线程接口由标准 C 库提供。
- 并发编程（与顺序编程相对）的原理。

注 – 本 Solaris 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。支持的系统可以在 `http://www.sun.com/bigadmin/hcl` 上的《Solaris 10 Hardware Compatibility List》中找到。本文档列举了在不同类型的平台上进行实现时的所有差别。

在本文档中，术语 "x86" 是指使用与 AMD64 或 Intel Xeon/Pentium 产品系列兼容的处理器生产的 64 位和 32 位系统。有关受支持的系统的信息，请参见《Solaris 10 Hardware Compatibility List》。

本指南的结构

第 1 章概述本发行版中线程实现的结构。

第 2 章讨论常规 POSIX 线程例程，其中重点介绍如何创建具有缺省属性的线程。

第 3 章介绍如何创建具有非缺省属性的线程。

第 4 章介绍线程同步例程。

第 5 章讨论为支持多线程而对操作环境进行的更改。

第 6 章介绍多线程的安全问题。

第 7 章介绍编译和调试多线程应用程序的基本信息。

第 8 章介绍 Solaris 线程（与 POSIX 线程相对）接口。

第 9 章讨论会影响程序员编写多线程应用程序的问题。

附录 A 说明如何为 POSIX 线程设计代码。

附录 B 举例说明如何在 Solaris 线程中构建屏障。

联机访问 Sun 文档

可以通过 docs.sun.com℠ Web 站点联机访问 Sun 技术文档。您可以浏览 docs.sun.com 文档库或查找某个特定的书名或主题。URL 为 http://docs.sun.com。

相关书籍

多线程技术要求以一种不同的方式来考虑函数交互。建议阅读以下书籍：

- 由 Alan Burns 和 Geoff Davies 合著的《*Concurrent Programming*》（Addison-Wesley 出版，1993）。
- 由 Michel Raynal 编著的《*Distributed Algorithms and Protocols*》（Wiley 出版，1998）。
- 由 Silberschatz、Peterson 和 Galvin 合著的《*Operating System Concepts*》（Addison-Wesley 出版，1991）。
- 由 M. Ben-Ari 编著的《*Principles of Concurrent Programming*》（Prentice-Hall 出版，1982）。
- 由 Steve Kleiman、Devang Shah 和 Bart Smalders 合著的《*Programming with Threads*》（Prentice Hall 出版，1996）。

印刷约定的含义

下表介绍了本书中的印刷约定。

表 P–1 印刷约定

字体或符号	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.`

表 P–1 印刷约定 （续）

字体或符号	含义	示例
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	要删除文件，请键入 **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。
新词术语强调	新词或术语以及要强调的词	必须成为**超级用户**才能执行此操作。
《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	`#`

◆ ◆ ◆ 第 1 章

1 多线程基础介绍

多线程一词可以解释为**多个控制线程**或**多个控制流**。虽然传统的 UNIX 进程包含单个控制线程，但多线程 (multithreading, MT) 会将一个进程分成许多执行线程，其中每个线程都可独立运行。

本章介绍了一些多线程的术语和概念及其所产生的益处。如果您已准备好开始使用多线程，请跳至第 2 章。

定义多线程术语

表 1–1 介绍了本书中所使用的一些术语。

表 1–1 多线程术语

术语	定义
Process（进程）	通过 `fork(2)` 系统调用创建的 UNIX 环境（如文件描述符和用户 ID 等），为运行程序而设置。
Thread（线程）	在进程上下文中执行的指令序列。
POSIX pthread	符合 POSIX 线程的线程接口。
Solaris thread（Solaris 线程）	不符合 POSIX 线程的 Sun Microsystems™ 线程接口，pthread 的前序节点。
single-threaded（单线程）	仅允许访问一个线程。
Multithreading（多线程）	允许访问两个或多个线程。

表 1–1 多线程术语 （续）

术语	定义
User-level or Application-level thread（用户级线程或应用程序级线程）	在用户空间（而非内核空间）中由线程调度例程管理的线程。
Lightweight process（轻量进程）	用来执行内核代码和系统调用的内核线程，又称作 LWP。从 Solaris 9 开始，每个线程都有一个专用的 LWP。
Bound thread（绑定线程）（过时的术语）	指的是在 Solaris 9 之前，和一个 LWP 永久绑定的用户级线程。从 Solaris 9 开始，每个线程都有一个专用的 LWP。
Unbound thread（非绑定线程）（过时的术语）	指的是在 Solaris 9 之前，无须和一个 LWP 绑定的用户级线程。从 Solaris 9 开始，每个线程都有一个专用的 LWP。
Attribute object（属性对象）	包含不透明数据类型和相关处理函数。这些数据类型和函数可以对 POSIX 线程一些可配置的方面，例如互斥锁 (mutex) 和条件变量，进行标准化。
Mutual exclusion lock（互斥锁）	用来锁定和解除锁定对共享数据访问的函数。
Condition variable（条件变量）	用来阻塞线程直到状态发生变化的函数。
Read-write lock（读写锁）	可用于对共享数据进行多次只读访问的函数，但是要修改共享数据则必须以独占方式访问。
Counting semaphore（计数信号量）	一种基于内存的同步机制。
Parallelism（并行性）	如果至少有两个线程正在同时**执行**，则会出现此情况。
Concurrency（并发性）	如果至少有两个线程**正在进行**，则会出现此情况。并发是一种更广义的并行性，其中可以包括分时这种形式的虚拟并行性。

符合多线程标准

多线程编程的概念至少可以回溯到二十世纪六十年代。多线程编程在 UNIX 系统中的发展是从八十年代中期开始的。虽然对多线程的定义以及对支持多线程所需要的功能存在共识，但是用于实现多线程的接口有很大不同。

在过去的几年内，POSIX（Portable Operating System Interface，可移植操作系统接口）1003.4a 工作小组一直致力于制定多线程编程标准。现在，该标准已得到认可。

该《多线程编程指南》基于 POSIX 标准 IEEE Std 1003.1 1996 版（又称作 ISO/IEC 9945–1 第二版）。最新修订版的 POSIX 标准 IEEE Std 1003.1:2001（又称作 ISO/IEC 9945:2002 和单一 UNIX 规范版本 3）中也提供了这些功能。

特定于 Solaris 线程的主题将在第 8 章中进行介绍。

多线程的益处

本节简要介绍多线程的益处。

在代码中实现多线程具有以下益处：

- 提高应用程序的响应
- 更有效地使用多处理器
- 改进程序结构
- 占用较少的系统资源

提高应用程序的响应

可以对任何一个包含许多相互独立的活动的程序进行重新设计，以便将每个活动定义为一个线程。例如，多线程 GUI 的用户不必等待一个活动完成即可启动另一个活动。

有效使用多处理器

通常，要求并发线程的应用程序无需考虑可用处理器的数量。使用额外的处理器可以明显提高应用程序的性能。

具有高度并行性的数值算法和数值应用程序（如矩阵乘法）在多处理器上通过多个线程实现时，运行速度会快得多。

改进程序结构

许多应用程序都以更有效的方式构造为多个独立或半独立的执行单元，而非整块的单个线程。多线程程序比单线程程序更能适应用户需求的变化。

占用较少的系统资源

如果两个或多个进程通过共享内存访问公用数据，则使用这些进程的程序可以实现对多个线程的控制。

但是，每个进程都有一个完整的地址空间和操作环境状态。每个进程用于创建和维护大量状态信息的成本，与一个线程相比，无论是在时间上还是空间上代价都更高。

此外，进程间所固有的独立性使得程序员需要花费很多精力来处理不同进程间线程的通信或者同步这些线程的操作。

结合线程和 RPC（远程过程调用）

通过将多个线程和一个远程过程调用 (remote procedure call, RPC) 结合起来，可以充分利用无共享内存的多处理器（如工作站集合）。这种结合将工作站集合视为一个多处理器，从而使应用程序的分布变得相对容易些。

例如，一个线程可以创建多个子线程，每个子线程随后可以请求远程过程调用，从而调用另一个工作站上的过程。尽管初始线程此时仅创建了一些并行运行的线程，但是这种并行性会涉及到其他计算机。

多线程概念

本节介绍多线程的基本概念。

并发性和并行性

在单个处理器的多线程进程中，处理器可以在线程之间切换执行资源，从而执行并发。

在共享内存的多处理器环境内的同一个多线程进程中，进程中的每个线程都可以在一个单独的处理器上并发运行，从而执行并行。如果进程中的线程数不超过处理器的数目，则线程的支持系统和操作环境可确保每个线程在不同的处理器上执行。例如，在线程数和处理器数目相同的矩阵乘法中，每个线程和每个处理器都会计算一行结果。

多线程结构一览

传统的 UNIX 已支持多线程的概念。每个进程都包含一个线程，因此对多个进程进行编程即是对多个线程进行编程。但是，进程同时也是一个地址空间，因此创建进程会涉及到创建新的地址空间。

创建线程比创建新进程成本低，因为新创建的线程使用的是当前进程的地址空间。相对于在进程之间切换，在线程之间进行切换所需的时间更少，因为后者不包括地址空间之间的切换。

在进程内部的线程间通信很简单，因为这些线程会共享所有内容，特别是地址空间。所以，一个线程生成的数据可以立即用于其他所有线程。

在 Solaris 9 和较早的 Solaris 发行版中，支持多线程的接口是通过特定的子例程库实现的。这些子例程库包括用于 POSIX 线程的 `libpthread` 和用于 Solaris 线程的 `libthread`。多线程通过将内核级资源和用户级资源分离来提供灵活性。在当前的发行版中，对于这两组接口的多线程支持是由标准 C 库提供的。

用户级线程

线程是多线程编程中的主编程接口。线程仅在进程内部是可见的，进程内部的线程会共享诸如地址空间、打开的文件等所有进程资源。

用户级线程状态

以下状态对于每个线程是唯一的。

- 线程 ID
- 寄存器状态（包括 PC 和栈指针）
- 栈
- 信号掩码
- 优先级
- 线程专用存储

由于线程可共享进程指令和大多数进程数据，因此一个线程对共享数据进行的更改对进程内其他线程是可见的。一个线程需要与同一个进程内的其他线程交互时，该线程可以在不涉及操作系统的情况下进行此操作。

注 – 顾名思义，用户级线程不同于内核级线程，只有系统程序员才能处理内核级线程。由于本书面向应用程序程序员，因此将不讨论内核级线程。

线程调度

POSIX 标准指定了三种调度策略：先入先出策略 (`SCHED_FIFO`)、循环策略 (`SCHED_RR`) 和自定义策略 (`SCHED_OTHER`)。`SCHED_FIFO` 是基于队列的调度程序，对于每个优先级都会使用不同的队列。`SCHED_RR` 与 FIFO 相似，不同的是前者的每个线程都有一个执行时间配额。

`SCHED_FIFO` 和 `SCHED_RR` 是对 POSIX Realtime 的扩展。`SCHED_OTHER` 是缺省的调度策略。

有关 `SCHED_OTHER` 策略的信息，请参见第 149 页中的 “LWP 和调度类”。

提供了两个调度范围：进程范围 (`PTHREAD_SCOPE_PROCESS`) 和系统范围 (`PTHREAD_SCOPE_SYSTEM`)。具有不同范围状态的线程可以在同一个系统甚至同一个进程中共存。进程范围只允许这种线程与同一进程中的其他线程争用资源，而系统范围则允许此类线程与系统内的其他所有线程争用资源。实际上，从 Solaris 9 发行版开始，系统就不再区分这两个范围。

线程取消

一个线程可以请求终止同一个进程中的其他任何线程。目标线程（要取消的线程）可以延后取消请求，并在该线程处理取消请求时执行特定于应用程序的清理操作。

通过 pthread 取消功能，可以对线程进行异步终止或延迟终止。异步取消可以随时发生，而延迟取消只能发生在所定义的点。延迟取消是缺省类型。

线程同步

使用同步功能，可以控制程序流并访问共享数据，从而并发执行多个线程。

共有四种同步模型：互斥锁、读写锁、条件变量和信号。

- **互斥锁**仅允许每次使用一个线程来执行特定的部分代码或者访问特定数据。
- **读写锁**允许对受保护的共享资源进行并发读取和独占写入。要修改资源，线程必须首先获取互斥写锁。只有释放所有的读锁之后，才允许使用互斥写锁。
- **条件变量**会一直阻塞线程，直到特定的条件为真。
- **计数信号量**通常用来协调对资源的访问。使用计数，可以限制访问某个信号的线程数量。达到指定的计数时，信号将阻塞。

使用 64 位体系结构

对于应用程序开发者，Solaris 64 位和 32 位环境的主要区别在于所使用的 C 语言数据类型的模型。64 位数据类型使用 LP64 模型，其中 `long` 和指针的宽度为 64 位，其他所有基础数据类型仍然与 32 位实现的数据类型相同。32 位数据类型使用 ILP32 模型，其中的 `int`、`long` 和指针宽度为 32 位。

以下简要概述了 64 位环境的主要特征以及使用该环境时的注意事项：

- 大虚拟地址空间

  在 64 位环境中，进程的虚拟地址空间最高可达 64 位（即 18 EB）。目前，32 位进程的最大地址空间为 4 GB，较大的虚拟地址空间大约是其 40 亿倍。但是由于硬件限制，某些平台可能并不支持完整的 64 位地址空间。

  大地址空间增加了可创建的具有缺省栈大小的线程数。在 32 位和 64 位系统中，栈的大小分别为 1 MB 和 2 MB。 在 32 位和 64 位系统中，具有缺省栈大小的线程数分别是大约 2000 个和 80000 亿个。

- 内核内存读取器

  内核是在内部使用 64 位数据结构的 LP64 对象。这意味着，使用 `libkvm`、`/dev/mem` 或 `/dev/kmem` 的现有 32 位应用程序不能正常工作，必须转换为 64 位程序。

- `/proc` 限制

  使用 `/proc` 的 32 位程序可以查看 32 位进程，但是无法识别 64 位进程。用来描述进程的现有接口和数据结构不够大，因此无法包含 64 位值。对于此类程序，必须将其重新编译为 64 位程序，使其可同时适用于 32 位进程和 64 位进程。

- 64 位库

  32 位库必须与 32 位应用程序进行链接，而 64 位库必须与 64 位应用程序进行链接。除已过时的库以外，所有的系统库都同时提供 32 位版本和 64 位版本。

- 64 位运算

  64 位运算早已在以前的 32 位 Solaris 发行版中提供。现在，64 位实现提供了完整的 64 位计算机寄存器，用于进行整数运算和参数传递。

- 大文件

  如果应用程序仅要求大文件支持，则可以保留 32 位并使用大文件接口。要充分利用 64 位功能，必须将应用程序转换为 64 位。

◆ ◆ ◆ 第 2 章

2 基本线程编程

本章介绍 POSIX 线程的基本线程编程例程。本章介绍**缺省线程**（即，具有缺省属性值的线程），这是多线程编程中最常用的线程。本章还介绍如何创建和使用具有非缺省属性的线程。

本章介绍的 POSIX 例程具有与最初的 Solaris 多线程库相似的编程接口。

线程库

下面简要论述了特定任务及其相关手册页。

创建缺省线程

如果未指定属性对象，则该对象为 NULL，系统会创建具有以下属性的缺省线程：

- 进程范围
- 非分离
- 缺省栈和缺省栈大小
- 零优先级

还可以用 pthread_attr_init() 创建缺省属性对象，然后使用该属性对象来创建缺省线程。有关详细信息，请参见第 50 页中的 “初始化属性”一节。

pthread_create 语法

使用 pthread_create(3C) 可以向当前进程中添加新的受控线程。

```
int    pthread_create(pthread_t *tid, const pthread_attr_t *tattr,

    void*(*start_routine)(void *), void *arg);
```

```
#include <pthread.h>


pthread_attr_t() tattr;

pthread_t tid;

extern void *start_routine(void *arg);

void *arg;

int ret;


/* default behavior*/

ret = pthread_create(&tid, NULL, start_routine, arg);


/* initialized with default attributes */

ret = pthread_attr_init(&tattr);

/* default behavior specified*/

ret = pthread_create(&tid, &tattr, start_routine, arg);
```

使用具有必要状态行为的 *attr* 调用 `pthread_create()` 函数。 *start_routine* 是新线程最先执行的函数。当 *start_routine* 返回时，该线程将退出，其退出状态设置为由 *start_routine* 返回的值。请参见第 23 页中的 “pthread_create 语法” 。

当 `pthread_create()` 成功时，所创建线程的 ID 被存储在由 *tid* 指向的位置中。

使用 `NULL` 属性参数或缺省属性调用 `pthread_create()` 时， `pthread_create()` 会创建一个缺省线程。在对 *tattr* 进行初始化之后，该线程将获得缺省行为。

pthread_create 返回值

`pthread_create()` 在调用成功完成之后返回零。其他任何返回值都表示出现了错误。如果检测到以下任一情况， `pthread_create()` 将失败并返回相应的值。

`EAGAIN`
描述: 超出了系统限制，如创建的线程太多。

`EINVAL`
描述: *tattr* 的值无效。

等待线程终止

pthread_join() 函数会一直阻塞调用线程，直到指定的线程终止。

pthread_join 语法

使用 pthread_join(3C) 等待线程终止。

```
int    pthread_join(thread_t tid, void **status);

#include <pthread.h>



pthread_t tid;

int ret;

void *status;



/* waiting to join thread "tid" with status */

ret = pthread_join(tid, &status);



/* waiting to join thread "tid" without status */

ret = pthread_join(tid, NULL);
```

指定的线程必须位于当前的进程中，而且不得是分离线程。有关线程分离的信息，请参见第 52 页中的 “设置分离状态” 。

当 *status* 不是 NULL 时，*status* 指向某个位置，在 pthread_join() 成功返回时，将该位置设置为已终止线程的退出状态。

如果多个线程等待同一个线程终止，则所有等待线程将一直等到目标线程终止。然后，一个等待线程成功返回。其余的等待线程将失败并返回 ESRCH 错误。

在 pthread_join() 返回之后，应用程序可回收与已终止线程关联的任何数据存储空间。

pthread_join 返回值

调用成功完成后，pthread_join() 将返回零。其他任何返回值都表示出现了错误。如果检测到以下任一情况，pthread_join() 将失败并返回相应的值。

ESRCH
描述: 没有找到与给定的线程 ID 相对应的线程。

EDEADLK
描述: 将出现死锁，如一个线程等待其本身，或者线程 A 和线程 B 互相等待。

EINVAL
描述: 与给定的线程 ID 相对应的线程是分离线程。

`pthread_join()` 仅适用于非分离的目标线程。如果没有必要等待特定线程终止之后才进行其他处理，则应当将该线程分离。

简单线程的示例

在示例 2–1 中，一个线程执行位于顶部的过程，该过程首先创建一个辅助线程来执行 `fetch()` 过程。`fetch()` 执行复杂的数据库查找操作，查找过程需要花费一些时间。

主线程将等待查找结果，但同时还执行其他操作。因此，主线程将执行其他活动，然后通过执行 `pthread_join()` 等待辅助线程。

将新线程的 *pbe* 参数作为栈参数进行传递。这个线程参数之所以能够作为栈参数传递，是因为主线程会等待辅助线程终止。不过，首选方法是使用 `malloc` 从堆分配存储，而不是传递指向线程栈存储的地址。如果将该参数作为地址传递到线程栈存储，则该地址可能无效或者在线程终止时会被重新分配。

示例 2–1 简单线程程序

```
void mainline (...)

{

        struct phonebookentry *pbe;

        pthread_attr_t tattr;

        pthread_t helper;

        void *status;



        pthread_create(&helper, NULL, fetch, &pbe);



            /* do something else for a while */
```

示例 2–1 简单线程程序 （续）

```
        pthread_join(helper, &status);
        /* it's now safe to use result */
}

void *fetch(struct phonebookentry *arg)
{
        struct phonebookentry *npbe;
        /* fetch value from a database */

        npbe = search (prog_name)
            if (npbe != NULL)
                *arg = *npbe;
        pthread_exit(0);
}

struct phonebookentry {
        char name[64];
        char phonenumber[32];
        char flags[16];
}
```

分离线程

pthread_detach(3C) 是 pthread_join(3C) 的替代函数，可回收创建时 *detachstate* 属性设置为 PTHREAD_CREATE_JOINABLE 的线程的存储空间。

pthread_detach 语法

```
int    pthread_detach(thread_t tid);

#include <pthread.h>



pthread_t tid;

int ret;



/* detach thread tid */

ret = pthread_detach(tid);
```

pthread_detach() 函数用于指示应用程序在线程 *tid* 终止时回收其存储空间。如果 *tid* 尚未终止，pthread_detach() 不会终止该线程。

pthread_detach 返回值

pthread_detach() 在调用成功完成之后返回零。其他任何返回值都表示出现了错误。如果检测到以下任一情况，pthread_detach() 将失败并返回相应的值。

EINVAL
描述: *tid* 是分离线程。

ESRCH
描述: *tid* 不是当前进程中有效的未分离的线程。

为线程特定数据创建键

单线程 C 程序有两类基本数据：局部数据和全局数据。对于多线程 C 程序，添加了第三类数据：**线程特定数据**。线程特定数据与全局数据非常相似，区别在于前者为线程专有。

线程特定数据基于每线程进行维护。TSD（特定于线程的数据）是定义和引用线程专用数据的唯一方法。每个线程特定数据项都与一个作用于进程内所有线程的键关联。通过使用 *key*，线程可以访问基于每线程进行维护的指针 (*void* *)。

pthread_key_create 语法

```
int    pthread_key_create(pthread_key_t *key,

    void (*destructor) (void *));

#include <pthread.h>



pthread_key_t key;

int ret;



/* key create without destructor */

ret = pthread_key_create(&key, NULL);



/* key create with destructor */

ret = pthread_key_create(&key, destructor);
```

可以使用 `pthread_key_create`(3C) 分配用于标识进程中线程特定数据的**键**。键对进程中的所有线程来说是全局的。创建线程特定数据时，所有线程最初都具有与该键关联的 `NULL` 值。

使用各个键之前，会针对其调用一次 `pthread_key_create()`。不存在对键（为进程中所有的线程所共享）的隐含同步。

创建键之后，每个线程都会将一个值绑定到该键。这些值特定于线程并且针对每个线程单独维护。如果创建该键时指定了 `destructor` 函数，则该线程终止时，系统会解除针对每线程的绑定。

当 `pthread_key_create()` 成功返回时，会将已分配的键存储在 *key* 指向的位置中。调用方必须确保对该键的存储和访问进行正确的同步。

使用可选的析构函数 `destructor` 可以释放过时的存储。如果某个键具有非 `NULL` `destructor` 函数，而线程具有一个与该键关联的非 `NULL` 值，则该线程退出时，系统将使用当前的相关值调用 `destructor` 函数。`destructor` 函数的调用顺序不确定。

pthread_key_create 返回值

`pthread_key_create()` 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，`pthread_key_create()` 将失败并返回相应的值。

EAGAIN
描述: *key* 名称空间已经用完。

ENOMEM
描述: 此进程中虚拟内存不足，无法创建新键。

删除线程特定数据键

使用 pthread_key_delete(3C) 可以销毁现有线程特定数据键。由于键已经无效，因此将释放与该键关联的所有内存。引用无效键将返回错误。Solaris 线程中没有类似的函数。

pthread_key_delete 语法

```
int    pthread_key_delete(pthread_key_t key);

#include <pthread.h>



pthread_key_t key;

int ret;



/* key previously created */

ret = pthread_key_delete(key);
```

如果已删除**键**，则使用调用 pthread_setspecific() 或 pthread_getspecific() 引用该键时，生成的结果将是不确定的。

程序员在调用删除函数之前必须释放所有线程特定资源。删除函数不会调用任何析构函数。反复调用 pthread_key_create() 和 pthread_key_delete() 可能会产生问题。如果 pthread_key_delete() 将键标记为无效，而之后 *key* 的值不再被重用，那么反复调用它们就会出现问题。对于每个所需的键，应当只调用 pthread_key_create() 一次。

pthread_key_delete 返回值

pthread_key_delete() 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下情况，pthread_key_delete() 将失败并返回相应的值。

EINVAL
描述: *key* 的值无效。

设置线程特定数据

使用 pthread_setspecific(3C) 可以为指定线程特定数据键设置线程特定绑定。

pthread_setspecific 语法

```
int    pthread_setspecific(pthread_key_t key, const void *value);

#include <pthread.h>



pthread_key_t key;

void *value;

int ret;



/* key previously created */

ret = pthread_setspecific(key, value);
```

pthread_setspecific 返回值

pthread_setspecific() 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，pthread_setspecific() 将失败并返回相应的值。

ENOMEM
描述: 虚拟内存不足。

EINVAL
描述: *key* 无效。

注 – 设置新绑定时，pthread_setspecific() 不会释放其存储空间。必须释放现有绑定，否则会出现内存泄漏。

获取线程特定数据

请使用 pthread_getspecific(3C) 获取调用线程的**键**绑定，并将该绑定存储在 *value* 指向的位置中。

pthread_getspecific 语法

```
void    *pthread_getspecific(pthread_key_t key);
```

```
#include <pthread.h>


pthread_key_t key;

void *value;


/* key previously created */

value = pthread_getspecific(key);
```

pthread_getspecific 返回值

pthread_getspecific 不返回任何错误。

全局和专用线程特定数据的示例

示例 2–2 显示的代码是从多线程程序中摘录出来的。这段代码可以由任意数量的线程执行，但该代码引用了两个全局变量：*errno* 和 *mywindow*。这些全局值实际上应当是对每个线程专用项的引用。

示例 2–2 线程特定数据－全局但专用

```
body() {

    ...


    while (write(fd, buffer, size) == -1) {

        if (errno != EINTR) {

            fprintf(mywindow, "%s\n", strerror(errno));

            exit(1);

        }

    }


    ...
```

示例 2–2 线程特定数据－全局但专用　　（续）

```
}
```

errno 引用应该从线程所调用的例程获取系统错误，而从其他线程所调用的例程获取系统错误。因此，线程不同，引用 errno 所指向的存储位置也不同。

mywindow 变量指向一个 stdio （标准 IO）流，作为线程专属的流窗口。因此，与 errno 一样，线程不同，引用 *mywindow* 所指向的存储位置也不同。最终，这个引用指向不同的流窗口。唯一的区别在于系统负责处理 errno，而程序员必须处理对 *mywindow* 的引用。

下一个示例说明对 *mywindow* 的引用如何工作。预处理程序会将对 *mywindow* 的引用转换为对 _mywindow() 过程的调用。

此例程随后调用 pthread_getspecific()。pthread_getspecific() 接收 *mywindow_key* 全局变量作为输入参数，以输出参数 *win* 返回该线程的窗口。

示例 2–3 将全局引用转化为专用引用

```
thread_key_t mywin_key;



FILE *_mywindow(void) {

    FILE *win;



    win = pthread_getspecific(mywin_key);

    return(win);

}



#define mywindow _mywindow()



void routine_uses_win( FILE *win) {

    ...

}
```

示例 2–3 将全局引用转化为专用引用　　（续）

```
void thread_start(...) {

    ...

    make_mywin();

    ...

    routine_uses_win( mywindow )

    ...

}
```

mywin_key 变量标识一类变量，对于该类变量，每个线程都有其各自的专用副本。这些变量是线程特定数据。每个线程都调用 `make_mywin()` 以初始化其时限并安排其 *mywindow* 实例以引用线程特定数据。

调用此例程之后，此线程可以安全地引用 *mywindow*，调用 `_mywindow()` 之后，此线程将获得对其专用时限的引用。引用 *mywindow* 类似于直接引用线程专用数据。

示例 2–4 说明如何设置引用。

示例 2–4 初始化线程特定数据

```
void make_mywindow(void) {

    FILE **win;

    static pthread_once_t mykeycreated = PTHREAD_ONCE_INIT;



    pthread_once(&mykeycreated, mykeycreate);



    win = malloc(sizeof(*win));

    create_window(win, ...);
```

示例 2–4 初始化线程特定数据　（续）

```
    pthread_setspecific(mywindow_key, win);

}



void mykeycreate(void) {

    pthread_key_create(&mywindow_key, free_key);

}



void free_key(void *win) {

    free(win);

}
```

首先，得到一个唯一的键值，*mywin_key*。此键用于标识线程特定数据类。第一个调用 `make_mywin()` 的线程最终会调用 `pthread_key_create()`，该函数将唯一的 *key* 赋给其第一个参数。第二个参数是 `destructor` 函数，用于在线程终止后将该线程的特定于该线程的数据项实例解除分配。

接下来为调用方的线程特定数据项的实例分配存储空间。获取已分配的存储空间，调用 `create_window()`，以便为该线程设置时限。*win* 指向为该时限分配的存储空间。最后，调用 `pthread_setspecific()`，将 *win* 与该键关联。

以后，每当线程调用 `pthread_getspecific()` 以传递全局 *key*，线程都会获得它在前一次调用 `pthread_setspecific()` 时设置的与该键关联的值）。

线程终止时，会调用在 `pthread_key_create()` 中设置的 `destructor` 函数。每个 `destructor` 函数仅在终止线程通过调用 `pthread_setspecific()` 为 *key* 赋值之后才会被调用。

获取线程标识符

请使用 `pthread_self`(3C) 获取调用线程的 `thread identifier`。

pthread_self 语法

```
pthread_t     pthread_self(void);
```

```
#include <pthread.h>


pthread_t tid;


tid = pthread_self();
```

pthread_self 返回值

pthread_self() 返回调用线程的 thread identifier。

比较线程 ID

请使用 pthread_equal(3C) 对两个线程的线程标识号进行比较。

pthread_equal 语法

```
int     pthread_equal(pthread_t tid1, pthread_t tid2);

#include <pthread.h>


pthread_t tid1, tid2;

int ret;


ret = pthread_equal(tid1, tid2);
```

pthread_equal 返回值

如果 *tid1* 和 *tid2* 相等，pthread_equal() 将返回非零值，否则将返回**零**。如果 *tid1* 或 *tid2* 是无效的线程标识号，则结果无法预测。

初始化线程

使用 pthread_once(3C)，可以在首次调用 pthread_once 时调用初始化例程。以后调用 pthread_once() 将不起作用。

pthread_once 语法

```
int     pthread_once(pthread_once_t *once_control,

    void (*init_routine)(void));

#include <pthread.h>



pthread_once_t once_control = PTHREAD_ONCE_INIT;

int ret;



ret = pthread_once(&once_control, init_routine);
```

once_control 参数用来确定是否已调用相关的初始化例程。

pthread_once 返回值

`pthread_once()` 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下情况，`pthread_once()` 将失败并返回相应的值。

`EINVAL`
描述: *once_control* 或 *init_routine* 是 `NULL`。

停止执行线程

使用 `sched_yield`(3RT)，可以使当前线程停止执行，以便执行另一个具有相同或更高优先级的线程。

sched_yield 语法

```
int     sched_yield(void);

#include <sched.h>



int ret;



ret = sched_yield();
```

sched_yield返回值

sched_yield() 在成功完成之后返回零。否则，返回 -1，并设置 *errno* 以指示错误状态。

ENOSYS
描述: 本实现不支持 sched_yield。

设置线程的优先级

请使用 pthread_setschedparam(3C) 修改现有线程的优先级。此函数对于调度策略不起作用。

pthread_setschedparam语法

```
int     pthread_setschedparam(pthread_t tid, int policy,

    const struct sched_param *param);

#include <pthread.h>



pthread_t tid;

int ret;

struct sched_param param;

int priority;



/* sched_priority will be the priority of the thread */

sched_param.sched_priority = priority;

policy = SCHED_OTHER;



/* scheduling parameters of target thread */

ret = pthread_setschedparam(tid, policy, &param);
```

pthread_setschedparam 返回值

pthread_setschedparam() 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，pthread_setschedparam() 函数将失败并返回相应的值。

EINVAL
描述:所设置属性的值无效。

ENOTSUP
描述:尝试将该属性设置为不受支持的值。

获取线程的优先级

pthread_getschedparam(3C) 可用来获取现有线程的优先级。

pthread_getschedparam 语法

```
int     pthread_getschedparam(pthread_t tid, int policy,

    struct schedparam *param);

#include <pthread.h>



pthread_t tid;

sched_param param;

int priority;

int policy;

int ret;



/* scheduling parameters of target thread */

ret = pthread_getschedparam (tid, &policy, &param);



/* sched_priority contains the priority of the thread */

priority = param.sched_priority;
```

pthread_getschedparam 返回值

pthread_getschedparam() 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

ESRCH
描述: *tid* 指定的值不引用现有的线程。

向线程发送信号

请使用 pthread_kill(3C) 向线程发送信号。

pthread_kill 语法

```
int     pthread_kill(thread_t tid, int sig);

#include <pthread.h>

#include <signal.h>



int sig;

pthread_t tid;

int ret;



ret = pthread_kill(tid, sig);
```

pthread_kill() 将信号 *sig* 发送到由 *tid* 指定的线程。*tid* 所指定的线程必须与调用线程在同一个进程中。*sig* 参数必须来自 signal(5) 提供的列表。

如果 *sig* 为零，将执行错误检查，但并不实际发送信号。此错误检查可用来检查 *tid* 的有效性。

pthread_kill 返回值

pthread_kill() 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，pthread_kill() 将失败并返回相应的值。

EINVAL
描述: *sig* 是无效的信号量。

ESRCH
描述: 当前的进程中找不到 *tid*。

访问调用线程的信号掩码

请使用 pthread_sigmask(3C) 更改或检查调用线程的信号掩码。

pthread_sigmask 语法

```
int    pthread_sigmask(int how, const sigset_t *new, sigset_t *old);

#include <pthread.h>

#include <signal.h>



int ret;

sigset_t old, new;



ret = pthread_sigmask(SIG_SETMASK, &new, &old); /* set new mask */

ret = pthread_sigmask(SIG_BLOCK, &new, &old); /* blocking mask */

ret = pthread_sigmask(SIG_UNBLOCK, &new, &old); /* unblocking */
```

how 用来确定如何更改信号组。*how* 可以为以下值之一：

- SIG_BLOCK。向当前的信号掩码中添加 *new*，其中 *new* 表示要阻塞的信号组。
- SIG_UNBLOCK。从当前的信号掩码中删除 *new*，其中 *new* 表示要取消阻塞的信号组。
- SIG_SETMASK。将当前的信号掩码替换为 *new*，其中 *new* 表示新的信号掩码。

当 *new* 的值为 NULL 时，*how* 的值没有意义，线程的信号掩码不发生变化。要查询当前已阻塞的信号，请将 NULL 值赋给 *new* 参数。

除非 *old* 变量为 NULL，否则 *old* 指向用来存储以前的信号掩码的空间。

pthread_sigmask 返回值

pthread_sigmask() 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下情况，pthread_sigmask() 将失败并返回相应的值。

EINVAL
描述: 未定义 *how* 的值。

安全地 Fork

请参见第 146 页中的 “解决方案： pthread_atfork”中有关 pthread_atfork(3C) 的论述。

pthread_atfork 语法

```
int pthread_atfork(void (*prepare) (void), void (*parent) (void),

    void (*child) (void) );
```

pthread_atfork 返回值

pthread_atfork() 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下情况，pthread_atfork() 将失败并返回相应的值。

ENOMEM
描述: 表空间不足，无法记录 Fork 处理程序地址。

终止线程

请使用 pthread_exit(3C) 终止线程。

pthread_exit 语法

```
void     pthread_exit(void *status);

#include <pthread.h>



void *status;



pthread_exit(status); /* exit with status */
```

pthread_exit() 函数可用来终止调用线程。将释放所有线程特定数据绑定。如果调用线程尚未分离，则线程 ID 和 *status* 指定的退出状态将保持不变，直到应用程序调用 pthread_join() 以等待该线程。否则，将忽略 *status*。线程 ID 可以立即回收。有关线程分离的信息，请参见第 52 页中的 “设置分离状态” 。

pthread_exit 返回值

调用线程将终止，退出状态设置为 *status* 的内容。

结束

线程可通过以下方法来终止执行：

- 从线程的第一个（最外面的）过程返回，使线程启动例程。请参见 pthread_create。
- 调用 pthread_exit()，提供退出状态。

- 使用 POSIX 取消函数执行终止操作。请参见 pthread_cancel()。

线程的缺省行为是拖延，直到其他线程通过 "joining" 拖延线程确认其已死亡。此行为与非分离的缺省 pthread_create() 属性相同，请参见 pthread_detach。join 的结果是 joining 线程得到已终止线程的退出状态，已终止的线程将消失。

有一个重要的特殊情况，即当初始线程（即调用 main() 的线程）从 main() 调用返回时或调用 exit() 时，整个进程及其所有的线程将终止。因此，一定要确保初始线程不会从 main() 过早地返回。

请注意，如果主线程仅仅调用了 pthread_exit，则仅主线程本身终止。进程及进程内的其他线程将继续存在。所有线程都已终止时，进程也将终止。

取消线程

取消操作允许线程请求终止其所在进程中的任何其他线程。不希望或不需要对一组相关的线程执行进一步操作时，可以选择执行取消操作。

取消线程的一个示例是异步生成取消条件，例如，用户请求关闭或退出正在运行的应用程序。另一个示例是完成由许多线程执行的任务。其中的某个线程可能最终完成了该任务，而其他线程还在继续运行。由于正在运行的线程此时没有任何用处，因此应当取消这些线程。

取消点

仅当取消操作安全时才应取消线程。pthreads 标准指定了几个取消点，其中包括：

- 通过 pthread_testcancel 调用以编程方式建立线程取消点。
- 线程等待 pthread_cond_wait 或 pthread_cond_timedwait(3C) 中的特定条件出现。
-
- 被 sigwait(2) 阻塞的线程。
- 一些标准的库调用。通常，这些调用包括线程可基于其阻塞的函数。有关列表，请参见 cancellation(5) 手册页。

缺省情况下将启用取消功能。有时，您可能希望应用程序禁用取消功能。如果禁用取消功能，则会导致延迟所有的取消请求，直到再次启用取消请求。

有关禁用取消功能的信息，请参见第 45 页中的 “pthread_setcancelstate 语法”。

放置取消点

执行取消操作存在一定的危险。大多数危险都与完全恢复不变量和释放共享资源有关。取消线程时一定要格外小心，否则可能会使互斥保留为锁定状态，从而导致死锁。或者，已取消的线程可能保留已分配的内存区域，但是系统无法识别这一部分内存，从而无法释放它。

标准C库指定了一个取消接口用于以编程方式允许或禁止取消功能。该库定义的**取消点**是一组可能会执行取消操作的点。该库还允许定义**取消处理程序**的范围，以确保这些处理程序在预期的时间和位置运行。取消处理程序提供的清理服务可以将资源和状态恢复到与起点一致的状态。

必须对应用程序有一定的了解，才能放置取消点并执行取消处理程序。互斥肯定不是取消点，只应当在必要时使之保留尽可能短的时间。

请将异步取消区域限制在没有外部依赖性的序列，因为外部依赖性可能会产生挂起的资源或未解决的状态条件。在从某个备用的嵌套取消状态返回时，一定要小心地恢复取消状态。该接口提供便于进行恢复的功能：`pthread_setcancelstate(3C)` 在所引用的变量中保留当前的取消状态，`pthread_setcanceltype(3C)` 以同样的方式保留当前的取消类型。

在以下三种不同的情况下可能会执行取消操作：

- 异步
- 执行序列中按标准定义的各个点
- 调用 `pthread_testcancel()` 时

缺省情况下，仅在POSIX标准可靠定义的点执行取消操作。

无论何时，都应注意资源和状态恢已复到与起点一致的状态。

取消线程

请使用 `pthread_cancel(3C)` 取消线程。

pthread_cancel 语法

```
int    pthread_cancel(pthread_t thread);
```

```
#include <pthread.h>
```

```
pthread_t thread;

int ret;
```

```
ret = pthread_cancel(thread);
```

取消请求的处理方式取决于目标线程的状态。状态由以下两个函数确定：`pthread_setcancelstate(3C)` 和 `pthread_setcanceltype(3C)`。

pthread_cancel 返回值

pthread_cancel() 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

ESRCH
描述: 没有找到与给定线程 ID 相对应的线程。

启用或禁用取消功能

请使用 pthread_setcancelstate(3C) 启用或禁用线程取消功能。创建线程时，缺省情况下线程取消功能处于启用状态。

pthread_setcancelstate 语法

```
int    pthread_setcancelstate(int state, int *oldstate);

#include <pthread.h>



int oldstate;

int ret;



/* enabled */

ret = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);



/* disabled */

ret = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);
```

pthread_setcancelstate 返回值

pthread_setcancelstate() 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下情况，pthread_setcancelstate() 函数将失败并返回相应的值。

EINVAL
描述: 状态不是 PTHREAD_CANCEL_ENABLE 或 PTHREAD_CANCEL_DISABLE。

设置取消类型

使用 pthread_setcanceltype(3C) 可以将取消类型设置为延迟或异步模式。

pthread_setcanceltype 语法

```
int    pthread_setcanceltype(int type, int *oldtype);

#include <pthread.h>



int oldtype;

int ret;



/* deferred mode */

ret = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype);



/* async mode*/

ret = pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &oldtype);
```

创建线程时，缺省情况下会将取消类型设置为延迟模式。在延迟模式下，只能在取消点取消线程。在异步模式下，可以在执行过程中的任意一点取消线程。因此建议不使用异步模式。

pthread_setcanceltype 返回值

pthread_setcanceltype() 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: 类型不是 PTHREAD_CANCEL_DEFERRED 或 PTHREAD_CANCEL_ASYNCHRONOUS。

创建取消点

请使用 pthread_testcancel(3C) 为线程建立取消点。

pthread_testcancel 语法

```
void pthread_testcancel(void);
```

```
#include <pthread.h>


pthread_testcancel();
```

当线程取消功能处于启用状态且取消类型设置为延迟模式时，pthread_testcancel() 函数有效。如果在取消功能处于禁用状态下调用 pthread_testcancel()，则该函数不起作用。

请务必仅在线程取消操作安全的序列中插入 pthread_testcancel()。除通过 pthread_testcancel() 调用以编程方式建立的取消点以外，pthread 标准还指定了几个取消点。有关更多详细信息，请参见第 43 页中的 “取消点” 。

pthread_testcancel 返回值

pthread_testcancel() 没有返回值。

将处理程序推送到栈上

使用清理处理程序，可以将状态恢复到与起点一致的状态，其中包括清理已分配的资源和恢复不变量。使用 pthread_cleanup_push(3C) 和 pthread_cleanup_pop(3C) 函数可以管理清理处理程序。

在程序的同一词法域中可以推送和弹出清理处理程序。推送和弹出操作应当始终匹配，否则会生成编译器错误。

pthread_cleanup_push 语法

请使用 pthread_cleanup_push(3C) 将清理处理程序推送到清理栈 (LIFO)。

```
void pthread_cleanup_push(void(*routine)(void *), void *args);

#include <pthread.h>


/* push the handler "routine" on cleanup stack */

pthread_cleanup_push (routine, arg);
```

pthread_cleanup_push 返回值

pthread_cleanup_push() 没有返回值。

从栈中弹出处理程序

请使用 pthread_cleanup_pop(3C) 从清理栈中弹出清理处理程序。

pthread_cleanup_pop 语法

```
void pthread_cleanup_pop(int execute);

#include <pthread.h>



/* pop the "func" out of cleanup stack and execute "func" */

pthread_cleanup_pop (1);



/* pop the "func" and DONT execute "func" */

pthread_cleanup_pop (0);
```

如果弹出函数中的参数为非零值，则会从栈中删除该处理程序并执行该处理程序。如果该参数为零，则会弹出该处理程序，而不执行它。

线程显式或隐式调用 pthread_exit(3C) 时，或线程接受取消请求时，会使用非零参数有效地调用 `pthread_cleanup_pop()`。

pthread_cleanup_pop 返回值

`pthread_cleanup_pop()` 没有返回值。

◆ ◆ ◆ 第 3 章

3 线程属性

前面一章介绍了使用缺省属性创建线程的基本原理。本章论述如何在创建线程时设置属性。

注 – 只有 pthreads 使用属性和取消功能。本章中介绍的 API 仅适用于 POSIX 线程。除此之外，Solaris 线程和 `pthreads` 的**功能**大致是相同的。有关相似和不同之处的更多信息，请参见第 8 章。

属性对象

通过设置属性，可以指定一种不同于缺省行为的行为。使用 `pthread_create(3C)` 创建线程时，或初始化同步变量时，可以指定属性对象。缺省值通常就足够了。

属性对象是不透明的，而且不能通过赋值直接进行修改。系统提供了一组函数，用于初始化、配置和销毁每种对象类型。

初始化和配置属性后，属性便具有进程范围的作用域。使用属性时最好的方法即是在程序执行早期一次配置好所有必需的状态规范。然后，根据需要引用相应的属性对象。

使用属性对象具有两个主要优点。

- 使用属性对象可增加代码可移植性。

  即使支持的属性可能会在实现之间有所变化，但您不需要修改用于创建线程实体的函数调用。这些函数调用不需要进行修改，因为属性对象是隐藏在接口之后的。

  如果目标系统支持的属性在当前系统中不存在，则必须显式提供才能管理新的属性。管理这些属性是一项非常容易的移植任务，因为只需在明确定义的位置初始化属性对象一次即可。

- 应用程序中的状态规范已被简化。

  例如，假设进程中可能存在多组线程。每组线程都提供单独的服务。每组线程都有各自的状态要求。

在应用程序执行初期的某一时间，可以针对每组线程初始化线程属性对象。以后所有线程的创建都会引用已经为这类线程初始化的属性对象。初始化阶段是简单和局部的。将来就可以快速且可靠地进行任何修改。

在进程退出时需要注意属性对象。初始化对象时，将为该对象分配内存。必须将此内存返回给系统。pthreads 标准提供了用于销毁属性对象的函数调用。

初始化属性

请使用 pthread_attr_init(3C) 将对象属性初始化为其缺省值。存储空间是在执行期间由线程系统分配的。

pthread_attr_init 语法

```
int pthread_attr_init(pthread_attr_t *tattr);

#include <pthread.h>



pthread_attr_t tattr;

int ret;



/* initialize an attribute to the default value */

ret = pthread_attr_init(&tattr);
```

表 3–1 给出了属性 (*tattr*) 的缺省值。

表 3–1 *tattr* 的缺省属性值

属性	值	结果
scope	PTHREAD_SCOPE_PROCESS	新线程与进程中的其他线程发生竞争。
detachstate	PTHREAD_CREATE_JOINABLE	线程退出后，保留完成状态和线程 *ID*。
stackaddr	NULL	新线程具有系统分配的栈地址。
stacksize	0	新线程具有系统定义的栈大小。
priority	0	新线程的优先级为 0。

表 3–1 *tattr* 的缺省属性值 （续）

属性	值	结果
inheritsched	PTHREAD_EXPLICIT_SCHED	新线程不继承父线程调度优先级。
schedpolicy	SCHED_OTHER	新线程对同步对象争用使用 Solaris 定义的固定优先级。线程将一直运行，直到被抢占或者直到线程阻塞或停止为止。

pthread_attr_init 返回值

pthread_attr_init() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

ENOMEM
描述: 如果未分配足够的内存来初始化线程属性对象，将返回该值。

销毁属性

请使用 pthread_attr_destroy(3C) 删除初始化期间分配的存储空间。属性对象将会无效。

pthread_attr_destroy 语法

```
int    pthread_attr_destroy(pthread_attr_t *tattr);

#include <pthread.h>



pthread_attr_t tattr;

int ret;



/* destroy an attribute */

ret = pthread_attr_destroy(&tattr);
```

pthread_attr_destroy 返回值

pthread_attr_destroy() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: 指示 *tattr* 的值无效。

设置分离状态

如果创建分离线程 (PTHREAD_CREATE_DETACHED)，则该线程一退出，便可重用其线程 *ID* 和其他资源。如果调用线程不准备等待线程退出，请使用 pthread_attr_setdetachstate(3C)。

pthread_attr_setdetachstate(3C) 语法

```
int    pthread_attr_setdetachstate(pthread_attr_t *tattr,int detachstate);

#include <pthread.h>



pthread_attr_t tattr;

int ret;



/* set the thread detach state */

ret = pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
```

如果使用 PTHREAD_CREATE_JOINABLE 创建非分离线程，则假设应用程序将等待线程完成。也就是说，程序将对线程执行 pthread_join()。

无论是创建分离线程还是非分离线程，在所有线程都退出之前，进程不会退出。有关从 main() 提前退出而导致的进程终止的讨论，请参见第 42 页中的 “结束”。

注 – 如果未执行显式同步来防止新创建的分离线程失败，则在线程创建者从 pthread_create() 返回之前，可以将其线程 ID 重新分配给另一个新线程。

非分离线程在终止后，必须要有一个线程用 join 来等待它。否则，不会释放该线程的资源以供新线程使用，而这通常会导致内存泄漏。因此，如果不希望线程被等待，请将该线程作为分离线程来创建。

示例 3–1 创建分离线程

```
#include <pthread.h>



pthread_attr_t tattr;

pthread_t tid;

void *start_routine;
```

示例 3–1 创建分离线程 （续）

```
void arg

int ret;



/* initialized with default attributes */

ret = pthread_attr_init (&tattr);

ret = pthread_attr_setdetachstate (&tattr,PTHREAD_CREATE_DETACHED);

ret = pthread_create (&tid, &tattr, start_routine, arg);
```

pthread_attr_setdetachstate 返回值

`pthread_attr_setdetachstate()` 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

`EINVAL`
描述: 指示 *detachstate* 或 *tattr* 的值无效。

获取分离状态

请使用 `pthread_attr_getdetachstate`(3C) 检索线程创建状态（可以为分离或连接）。

pthread_attr_getdetachstate 语法

```
int    pthread_attr_getdetachstate(const pthread_attr_t *tattr,

    int *detachstate;

#include <pthread.h>



pthread_attr_t tattr;

int detachstate;

int ret;
```

```
/* get detachstate of thread */

ret = pthread_attr_getdetachstate (&tattr, &detachstate);
```

pthread_attr_getdetachstate 返回值

pthread_attr_getdetachstate() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: 指示 *detachstate* 的值为 NULL 或 *tattr* 无效。

设置栈溢出保护区大小

pthread_attr_setguardsize(3C) 可以设置 *attr* 对象的 *guardsize*。

pthread_attr_setguardsize(3C) 语法

```
#include <pthread.h>



int pthread_attr_setguardsize(pthread_attr_t *attr, size_t  guardsize);
```

出于以下两个原因，为应用程序提供了 guardsize 属性：

- 溢出保护可能会导致系统资源浪费。如果应用程序创建大量线程，并且已知这些线程永远不会溢出其栈，则可以关闭溢出保护区。通过关闭溢出保护区，可以节省系统资源。
- 线程在栈上分配大型数据结构时，可能需要较大的溢出保护区来检测栈溢出。

guardsize 参数提供了对栈指针溢出的保护。如果创建线程的栈时使用了保护功能，则实现会在栈的溢出端分配额外内存。此额外内存的作用与缓冲区一样，可以防止栈指针的栈溢出。如果应用程序溢出到此缓冲区中，这个错误可能会导致 SIGSEGV 信号被发送给该线程。

如果 *guardsize* 为零，则不会为使用 *attr* 创建的线程提供溢出保护区。如果 *guardsize* 大于零，则会为每个使用 *attr* 创建的线程提供大小至少为 *guardsize* 字节的溢出保护区。缺省情况下，线程具有实现定义的非零溢出保护区。

允许合乎惯例的实现，将 *guardsize* 的值向上舍入为可配置的系统变量 PAGESIZE 的倍数。请参见 sys/mman.h 中的 PAGESIZE。如果实现将 *guardsize* 的值向上舍入为 PAGESIZE 的倍数，则以 *guardsize*（先前调用 pthread_attr_setguardsize() 时指定的溢出保护区大小）为单位存储对指定 *attr* 的 pthread_attr_getguardsize() 的调用。

pthread_attr_setguardsize 返回值

如果出现以下情况，pthread_attr_setguardsize() 将失败：

EINVAL
描述: 参数 *attr* 无效，参数 *guardsize* 无效，或参数 *guardsize* 包含无效值。

获取栈溢出保护区大小

pthread_attr_getguardsize(3C) 可以获取 *attr* 对象的 *guardsize*。

pthread_attr_getguardsize 语法

```
#include <pthread.h>



int pthread_attr_getguardsize(const pthread_attr_t *attr,

                                        size_t  *guardsize);
```

允许一致的实现将 *guardsize* 中包含的值向上舍入为可配置系统变量 PAGESIZE 的倍数。请参见 sys/mman.h 中的 PAGESIZE。如果实现将 *guardsize* 的值向上舍入为 PAGESIZE 的倍数，则以 *guardsize*（先前调用 pthread_attr_setguardsize() 时指定的溢出保护区大小）为单位存储对指定 *attr* 的 pthread_attr_getguardsize() 的调用。

pthread_attr_getguardsize 返回值

如果出现以下情况，pthread_attr_getguardsize() 将失败：

EINVAL
描述: 参数 *attr* 无效，参数 *guardsize* 无效，或参数 *guardsize* 包含无效值。

设置范围

请使用 pthread_attr_setscope(3C) 建立线程的争用范围（PTHREAD_SCOPE_SYSTEM 或 PTHREAD_SCOPE_PROCESS）。使用 PTHREAD_SCOPE_SYSTEM 时，此线程将与系统中的所有线程进行竞争。使用 PTHREAD_SCOPE_PROCESS 时，此线程将与进程中的其他线程进行竞争。

注 – 只有在给定进程中才能访问这两种线程类型。

---

pthread_attr_setscope 语法

```
int    pthread_attr_setscope(pthread_attr_t *tattr,int scope);

#include <pthread.h>
```

```
pthread_attr_t tattr;

int ret;



/* bound thread */

ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);



/* unbound thread */

ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_PROCESS);
```

本示例使用三个函数调用：用于初始化属性的调用、用于根据缺省属性设置所有变体的调用，以及用于创建 pthreads 的调用。

```
#include <pthread.h>



pthread_attr_t attr;

pthread_t tid;

void start_routine;

void arg;

int ret;



/* initialized with default attributes */

ret = pthread_attr_init (&tattr);



/* BOUND behavior */

ret =  pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);

ret = pthread_create (&tid, &tattr, start_routine, arg);
```

pthread_attr_setscope 返回值

pthread_attr_setscope() **成功**完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: 尝试将 *tattr* 设置为无效的值。

获取范围

请使用 pthread_attr_getscope(3C) 检索线程范围。

pthread_attr_getscope 语法

```
int    pthread_attr_getscope(pthread_attr_t *tattr, int *scope);

#include <pthread.h>



pthread_attr_t tattr;

int scope;

int ret;



/* get scope of thread */

ret = pthread_attr_getscope(&tattr, &scope);
```

pthread_attr_getscope 返回值

pthread_attr_getscope() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *scope* 的值为 NULL 或 *tattr* 无效。

设置线程并行级别

针对标准符合性提供了 pthread_setconcurrency(3C)。应用程序使用 pthread_setconcurrency() 通知系统其所需的并发级别。对于 Solaris 9 发行版中引入的线程实现，此接口没有任何作用，所有可运行的线程都将被连接到 LWP。

pthread_setconcurrency 语法

```
#include <pthread.h>



int pthread_setconcurrency(int new_level);
```

pthread_setconcurrency 返回值

如果出现以下情况，pthread_setconcurrency() 将失败：

EINVAL
描述: *new_level* 指定的值为负数。

EAGAIN
描述: *new_level* 指定的值将导致系统资源不足。

获取线程并行级别

pthread_getconcurrency(3C) 返回先前调用 pthread_setconcurrency() 时设置的值。

pthread_getconcurrency 语法

```
#include <pthread.h>



int pthread_getconcurrency(void);
```

如果以前未调用 pthread_setconcurrency() 函数，则 pthread_getconcurrency() 将返回零。

pthread_getconcurrency 返回值

pthread_getconcurrency() 始终会返回先前调用 pthread_setconcurrency() 时设置的并发级别。如果从未调用 pthread_setconcurrency()，则 pthread_getconcurrency() 将返回零。

设置调度策略

请使用 pthread_attr_setschedpolicy(3C) 设置调度策略。POSIX 标准指定 SCHED_FIFO（先入先出）、SCHED_RR（循环）或 SCHED_OTHER（实现定义的方法）的调度策略属性。

pthread_attr_setschedpolicy(3C) 语法

```
int    pthread_attr_setschedpolicy(pthread_attr_t *tattr, int policy);
```

```
#include <pthread.h>



pthread_attr_t tattr;

int policy;

int ret;



/* set the scheduling policy to SCHED_OTHER */

ret = pthread_attr_setschedpolicy(&tattr, SCHED_OTHER);
```

- SCHED_FIFO

  如果调用进程具有有效的用户 ID 0，则争用范围为系统 (PTHREAD_SCOPE_SYSTEM) 的先入先出线程属于实时 (RT) 调度类。如果这些线程未被优先级更高的线程抢占，则会继续处理该线程，直到该线程放弃或阻塞为止。对于具有进程争用范围 (PTHREAD_SCOPE_PROCESS)) 的线程或其调用进程没有有效用户 ID 0 的线程，请使用 SCHED_FIFO。SCHED_FIFO 基于 TS 调度类。

- SCHED_RR

  如果调用进程具有有效的用户 ID 0，则争用范围为系统 (PTHREAD_SCOPE_SYSTEM)) 的循环线程属于实时 (RT) 调度类。如果这些线程未被优先级更高的线程抢占，并且这些线程没有放弃或阻塞，则在系统确定的时间段内将一直执行这些线程。对于具有进程争用范围 (PTHREAD_SCOPE_PROCESS) 的线程，请使用 SCHED_RR（基于 TS 调度类）。此外，这些线程的调用进程没有有效的用户 ID 0。

SCHED_FIFO 和 SCHED_RR 在 POSIX 标准中是可选的，而且仅用于实时线程。

有关调度的论述，请参见第 19 页中的 “线程调度”一节。

pthread_attr_setschedpolicy 返回值

pthread_attr_setschedpolicy() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EINVAL
: **描述:** 尝试将 *tattr* 设置为无效的值。

ENOTSUP
: **描述:** 尝试将该属性设置为不受支持的值。

获取调度策略

请使用 pthread_attr_getschedpolicy(3C) 检索调度策略。

pthread_attr_getschedpolicy 语法

```
int    pthread_attr_getschedpolicy(pthread_attr_t *tattr, int *policy);

#include <pthread.h>



pthread_attr_t tattr;

int policy;

int ret;



/* get scheduling policy of thread */

ret = pthread_attr_getschedpolicy (&tattr, &policy);
```

pthread_attr_getschedpolicy 返回值

pthread_attr_getschedpolicy() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: 参数 *policy* 为 NULL 或 *tattr* 无效。

设置继承的调度策略

请使用 pthread_attr_setinheritsched(3C) 设置继承的调度策略。

pthread_attr_setinheritsched 语法

```
int    pthread_attr_setinheritsched(pthread_attr_t *tattr, int inherit);

#include <pthread.h>



pthread_attr_t tattr;

int inherit;

int ret;
```

```
/* use the current scheduling policy */

ret = pthread_attr_setinheritsched(&tattr, PTHREAD_EXPLICIT_SCHED);
```

inherit 值 PTHREAD_INHERIT_SCHED 表示新建的线程将继承创建者线程中定义的调度策略。将忽略在 pthread_create() 调用中定义的所有调度属性。如果使用缺省值 PTHREAD_EXPLICIT_SCHED，则将使用 pthread_create() 调用中的属性。

pthread_attr_setinheritsched 返回值

pthread_attr_setinheritsched() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EINVAL
描述: 尝试将 *tattr* 设置为无效的值。

ENOTSUP
描述: 尝试将属性设置为不受支持的值。

获取继承的调度策略

pthread_attr_getinheritsched(3C) 将返回由 pthread_attr_setinheritsched() 设置的调度策略。

pthread_attr_getinheritsched 语法

```
int    pthread_attr_getinheritsched(pthread_attr_t *tattr, int *inherit);

#include <pthread.h>



pthread_attr_t tattr;

int inherit;

int ret;



/* get scheduling policy and priority of the creating thread */

ret = pthread_attr_getinheritsched (&tattr, &inherit);
```

pthread_attr_getinheritsched 返回值

pthread_attr_getinheritsched() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: 参数 *inherit* 为 NULL 或 *tattr* 无效。

设置调度参数

pthread_attr_setschedparam(3C) 可以设置调度参数。

pthread_attr_setschedparam 语法

```
int    pthread_attr_setschedparam(pthread_attr_t *tattr,

    const struct sched_param *param);

#include <pthread.h>



pthread_attr_t tattr;

int newprio;

sched_param param;

newprio = 30;



/* set the priority; others are unchanged */

param.sched_priority = newprio;



/* set the new scheduling param */

ret = pthread_attr_setschedparam (&tattr, &param);
```

调度参数是在 param 结构中定义的。仅支持优先级参数。新创建的线程使用此优先级运行。

pthread_attr_setschedparam 返回值

pthread_attr_setschedparam() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *param* 的值为 NULL 或 *tattr* 无效。

可以采用两种方式之一来管理 pthreads 优先级：

- 创建子线程之前，可以设置优先级属性
- 可以更改父线程的优先级，然后再将该优先级改回来

获取调度参数

pthread_attr_getschedparam(3C) 将返回由 pthread_attr_setschedparam() 定义的调度参数。

pthread_attr_getschedparam 语法

```
int    pthread_attr_getschedparam(pthread_attr_t *tattr,

    const struct sched_param *param);
```

```
#include <pthread.h>



pthread_attr_t attr;

struct sched_param param;

int ret;



/* get the existing scheduling param */

ret = pthread_attr_getschedparam (&tattr, &param);
```

使用指定的优先级创建线程

创建线程之前，可以设置优先级属性。将使用在 sched_param 结构中指定的新优先级创建子线程。此结构还包含其他调度信息。

创建子线程时建议执行以下操作：

- 获取现有参数

- 更改优先级
- 创建子线程
- 恢复原始优先级

创建具有优先级的线程的示例

示例 3–2 给出了使用不同于其父线程优先级的优先级创建子线程的示例。

示例 3–2 创建具有优先级的线程

```
#include <pthread.h>

#include <sched.h>


pthread_attr_t tattr;

pthread_t tid;

int ret;

int newprio = 20;

sched_param param;


/* initialized with default attributes */

ret = pthread_attr_init (&tattr);


/* safe to get existing scheduling param */

ret = pthread_attr_getschedparam (&tattr, &param);


/* set the priority; others are unchanged */

param.sched_priority = newprio;


/* setting the new scheduling param */
```

示例 3–2 创建具有优先级的线程　　（续）

```
ret = pthread_attr_setschedparam (&tattr, &param);



/* with new priority specified */

ret = pthread_create (&tid, &tattr, func, arg);
```

pthread_attr_getschedparam 返回值

`pthread_attr_getschedparam()` 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

`EINVAL`
描述: *param* 的值为 `NULL` 或 *tattr* 无效。

设置栈大小

`pthread_attr_setstacksize`(3C) 可以设置线程栈大小。

pthread_attr_setstacksize 语法

```
int    pthread_attr_setstacksize(pthread_attr_t *tattr,

                                 size_t size);

#include <pthread.h>



pthread_attr_t tattr;

size_t size;

int ret;



size = (PTHREAD_STACK_MIN + 0x4000);



/* setting a new size */

ret = pthread_attr_setstacksize(&tattr, size);
```

stacksize 属性定义系统分配的栈大小（以字节为单位）。*size* 不应小于系统定义的最小栈大小。有关更多信息，请参见第 67 页中的 “关于栈”。

size 包含新线程使用的栈的字节数。如果 *size* 为零，则使用缺省大小。在大多数情况下，零值最适合。

PTHREAD_STACK_MIN 是启动线程所需的栈空间量。此栈空间没有考虑执行应用程序代码所需的线程例程要求。

pthread_attr_setstacksize 返回值

pthread_attr_setstacksize() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *size* 值小于 PTHREAD_STACK_MIN，或超出了系统强加的限制，或者 *tattr* 无效。

获取栈大小

pthread_attr_getstacksize(3C) 将返回由 pthread_attr_setstacksize() 设置的栈大小。

pthread_attr_getstacksize 语法

```
int    pthread_attr_getstacksize(pthread_attr_t *tattr,

                                 size_t *size);

#include <pthread.h>



pthread_attr_t tattr;

size_t size;

int ret;



/* getting the stack size */

ret = pthread_attr_getstacksize(&tattr, &size);
```

pthread_attr_getstacksize 返回值

pthread_attr_getstacksize() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *tattr* 无效。

关于栈

通常，线程栈是从页边界开始的。任何指定的大小都被向上舍入到下一个页边界。不具备访问权限的页将被附加到栈的溢出端。大多数栈溢出都会导致将 SIGSEGV 信号发送到违例线程。将直接使用调用方分配的线程栈，而不进行修改。

指定栈时，还应使用 PTHREAD_CREATE_JOINABLE 创建线程。在该线程的 pthread_join(3C) 调用返回之前，不会释放该栈。在该线程终止之前，不会释放该线程的栈。了解这类线程是否已终止的唯一可靠方式是使用 pthread_join(3C)。

为线程分配栈空间

一般情况下，不需要为线程分配栈空间。系统会为每个线程的栈分配 1 MB（对于 32 位系统）或 2 MB（对于 64 位系统）的虚拟内存，而不保留任何交换空间。系统将使用 mmap() 的 MAP_NORESERVE 选项来进行分配。

系统创建的每个线程栈都具有红色区域。系统通过将页附加到栈的溢出端来创建红色区域，从而捕获栈溢出。此类页无效，而且会导致内存（访问时）故障。红色区域将被附加到所有自动分配的栈，无论大小是由应用程序指定，还是使用缺省大小。

注 – 对于库调用和动态链接，运行时栈要求有所变化。应绝对确定，指定的栈满足库调用和动态链接的运行时要求。

极少数情况下需要指定栈和/或栈大小。甚至专家也很难了解是否指定了正确的大小。甚至符合 ABI 标准的程序也不能静态确定其栈大小。栈大小取决于执行中特定运行时环境的需要。

生成自己的栈

指定线程栈大小时，必须考虑被调用函数以及每个要调用的后续函数的分配需求。需要考虑的因素应包括调用序列需求、局部变量和信息结构。

有时，您需要与缺省栈略有不同的栈。典型的情况是，线程需要的栈大小大于缺省栈大小。而不太典型的情况是，缺省大小太大。您可能正在使用不足的虚拟内存创建数千个线程，进而处理数千个缺省线程栈所需的数千兆字节的栈空间。

对栈的最大大小的限制通常较为明显，但对其最小大小的限制如何呢？必须存在足够的栈空间来处理推入栈的所有栈帧，及其局部变量等。

要获取对栈大小的绝对最小限制，请调用宏 PTHREAD_STACK_MIN。PTHREAD_STACK_MIN 宏将针对执行 NULL 过程的线程返回所需的栈空间量。有用的线程所需的栈大小大于最小栈大小，因此缩小栈大小时应非常谨慎。

```
#include <pthread.h>



pthread_attr_t tattr;

pthread_t tid;

int ret;



size_t size = PTHREAD_STACK_MIN + 0x4000;



/* initialized with default attributes */

ret = pthread_attr_init(&tattr);



/* setting the size of the stack also */

ret = pthread_attr_setstacksize(&tattr, size);



/* only size specified in tattr*/

ret = pthread_create(&tid, &tattr, start_routine, arg);
```

设置栈地址和大小

pthread_attr_setstack(3C) 可以设置线程栈地址和大小。

pthread_attr_setstack(3C) 语法

```
int    pthread_attr_setstack(pthread_attr_t *tattr,void *stackaddr,

                             size_t stacksize);
```

```
#include <pthread.h>



pthread_attr_t tattr;

void *base;

size_t size;

int ret;



base = (void *) malloc(PTHREAD_STACK_MIN + 0x4000);



/* setting a new address and size */

ret = pthread_attr_setstack(&tattr, base,PTHREAD_STACK_MIN + 0x4000);
```

stackaddr 属性定义线程栈的基准（低位地址）。*stacksize* 属性指定栈的大小。如果将 *stackaddr* 设置为非空值，而不是缺省的 NULL，则系统将在该地址初始化栈，假设大小为 *stacksize*。

base 包含新线程使用的栈的地址。如果 *base* 为 NULL，则 pthread_create(3C) 将为大小至少为 *stacksize* 字节的新线程分配栈。

pthread_attr_setstack(3C) 返回值

pthread_attr_setstack() 成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *base* 或 *tattr* 的值不正确。*stacksize* 的值小于 PTHREAD_STACK_MIN。

以下示例说明如何使用自定义栈地址和大小来创建线程。

```
#include <pthread.h>



pthread_attr_t tattr;

pthread_t tid;

int ret;
```

```
void *stackbase;

size_t size;


/* initialized with default attributes */

ret = pthread_attr_init(&tattr);


/* setting the base address and size of the stack */

ret = pthread_attr_setstack(&tattr, stackbase,size);


/* address and size specified */

ret = pthread_create(&tid, &tattr, func, arg);
```

获取栈地址和大小

pthread_attr_getstack(3C) 将返回由 pthread_attr_setstack() 设置的线程栈地址和大小。

pthread_attr_getstack 语法

```
int    pthread_attr_getstack(pthread_attr_t *tattr,void * *stackaddr,

                             size_t *stacksize);

#include <pthread.h>


pthread_attr_t tattr;

void *base;

size_t size;

int ret;
```

```
/* getting a stack address and size */

ret = pthread_attr_getstackaddr (&tattr, &base, &size);
```

pthread_attr_getstack返回值

pthread_attr_getstackaddr()成功完成后将返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *tattr*的值不正确。

◆ ◆ ◆ 第 4 章

用同步对象编程

本章介绍了可用于线程的同步类型，还说明了使用同步的时间和方法。

同步对象是内存中的变量，可以按照与访问数据完全相同的方式对其进行访问。不同进程中的线程可以通过放在由线程控制的共享内存中的同步对象互相通信。尽管不同进程中的线程通常互不可见，但这些线程仍可以互相通信。

同步对象还可以放在文件中。同步对象可以比创建它的进程具有更长的生命周期。

同步对象具有以下可用类型：

- 互斥锁
- 条件变量
- 读写锁
- 信号

同步的作用包括以下方面：

- 同步是确保共享数据一致性的唯一方法。
- 两个或多个进程中的线程可以合用一个同步对象。由于重新初始化同步对象会将对象的状态设置为**解除锁定**，因此应仅由其中的一个协作进程来初始化同步对象。
- 同步可确保可变数据的安全性。
- 进程可以映射文件并指示该进程中的线程获取记录锁。一旦获取了记录锁，映射此文件的任何进程中尝试获取该锁的任何线程都会被阻塞，直到释放该锁为止。

- 访问一个基本类型变量（如整数）时，可以针对一个内存负荷使用多个存储周期。如果整数没有与总线数据宽度对齐或者大于数据宽度，则会使用多个存储周期。尽管这种整数不对齐现象不会出现在 SPARC® Platform Edition 体系结构上，但是可移植的程序却可能会出现对齐问题。

注 – 在 32 位体系结构上，`long long` 不是原子类型。（**原子**操作不能划分成更小的操作。）`long long` 可作为两个 32 位值进行读写。类型 `int`、`char`、`float` 和指针在 SPARC Platform Edition 计算机和 Intel 体系结构的计算机上是原子类型。

互斥锁属性

使用互斥锁（互斥）可以使线程按顺序执行。通常，互斥锁通过确保一次只有一个线程执行代码的临界段来同步多个线程。互斥锁还可以保护单线程代码。

要更改缺省的互斥锁属性，可以对属性对象进行声明和初始化。通常，互斥锁属性会设置在应用程序开头的某个位置，以便可以快速查找和轻松修改。表 4–1 列出了用来处理互斥锁属性的函数。

表 4–1 互斥锁属性例程

操作	相关函数说明
初始化互斥锁属性对象	第 75 页中的 “`pthread_mutexattr_init` 语法”
销毁互斥锁属性对象	第 76 页中的 “`pthread_mutexattr_destroy` 语法”
设置互斥锁范围	第 77 页中的 “`pthread_mutexattr_setpshared` 语法”
获取互斥锁范围	第 78 页中的 “`pthread_mutexattr_getpshared` 语法”
设置互斥锁的类型属性	第 78 页中的 “`pthread_mutexattr_settype` 语法”
获取互斥锁的类型属性	第 79 页中的 “`pthread_mutexattr_gettype` 语法”
设置互斥锁属性的协议	第 80 页中的 “`pthread_mutexattr_setprotocol` 语法”
获取互斥锁属性的协议	第 82 页中的 “`pthread_mutexattr_getprotocol` 语法”
设置互斥锁属性的优先级上限	第 83 页中的 “`pthread_mutexattr_setprioceiling` 语法”
获取互斥锁属性的优先级上限	第 84 页中的 “`pthread_mutexattr_getprioceiling` 语法”
设置互斥锁的优先级上限	第 85 页中的 “`pthread_mutex_setprioceiling` 语法”
获取互斥锁的优先级上限	第 86 页中的 “`pthread_mutex_getprioceiling` 语法”
设置互斥锁的强健属性	第 86 页中的 “`pthread_mutexattr_setrobust_np` 语法”

表 4–1 互斥锁属性例程 （续）

操作	相关函数说明
获取互斥锁的强健属性	第 88 页中的 “pthread_mutexattr_getrobust_np 语法”

表 4–2 中显示了在定义互斥范围时 Solaris 线程和 POSIX 线程之间的差异。

表 4–2 互斥锁范围比较

Solaris	POSIX	定义
USYNC_PROCESS	PTHREAD_PROCESS_SHARED	用于同步该进程和其他进程中的线程
USYNC_PROCESS_ROBUST	无 POSIX 等效项	用于在进程间**可靠地**同步线程
USYNC_THREAD	PTHREAD_PROCESS_PRIVATE	用于仅同步该进程中的线程

初始化互斥锁属性对象

使用 pthread_mutexattr_init(3C) 可以将与互斥锁对象相关联的属性初始化为其缺省值。在执行过程中，线程系统会为每个属性对象分配存储空间。

pthread_mutexattr_init 语法

```
int    pthread_mutexattr_init(pthread_mutexattr_t *mattr);

#include <pthread.h>



pthread_mutexattr_t mattr;

int ret;



/* initialize an attribute to default value */

ret = pthread_mutexattr_init(&mattr);
```

调用此函数时，*pshared* 属性的缺省值为 PTHREAD_PROCESS_PRIVATE。该值表示可以在进程内使用经过初始化的互斥锁。

mattr 的类型为 opaque，其中包含一个由系统分配的属性对象。*mattr* 范围可能的值为 PTHREAD_PROCESS_PRIVATE 和 PTHREAD_PROCESS_SHARED。PTHREAD_PROCESS_PRIVATE 是缺省值。

对于互斥锁属性对象，必须首先通过调用 pthread_mutexattr_destroy(3C) 将其销毁，才能重新初始化该对象。pthread_mutexattr_init() 调用会导致分配类型为 opaque 的对象。如果未销毁该对象，则会导致内存泄漏。

pthread_mutexattr_init 返回值

pthread_mutexattr_init() 成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

ENOMEM
描述: 内存不足，无法初始化互斥锁属性对象。

销毁互斥锁属性对象

pthread_mutexattr_destroy(3C) 可用来取消分配用于维护 pthread_mutexattr_init() 所创建的属性对象的存储空间。

pthread_mutexattr_destroy 语法

```
int    pthread_mutexattr_destroy(pthread_mutexattr_t *mattr)

#include <pthread.h>

pthread_mutexattr_t mattr;

int ret;



/* destroy an attribute */

ret = pthread_mutexattr_destroy(&mattr);
```

pthread_mutexattr_destroy 返回值

pthread_mutexattr_destroy() 成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: 由 *mattr* 指定的值无效。

设置互斥锁的范围

pthread_mutexattr_setpshared(3C) 可用来设置互斥锁变量的作用域。

pthread_mutexattr_setpshared 语法

```
int    pthread_mutexattr_setpshared(pthread_mutexattr_t *mattr,

    int pshared);

#include <pthread.h>



pthread_mutexattr_t mattr;

int ret;



ret = pthread_mutexattr_init(&mattr);

/*

 * resetting to its default value: private

 */

ret = pthread_mutexattr_setpshared(&mattr,

     PTHREAD_PROCESS_PRIVATE);
```

互斥锁变量可以是进程专用的（进程内）变量，也可以是系统范围内的（进程间）变量。要在多个进程中的线程之间共享互斥锁，可以在共享内存中创建互斥锁，并将 *pshared* 属性设置为 PTHREAD_PROCESS_SHARED。此行为与最初的 Solaris 线程实现中 mutex_init() 中的 USYNC_PROCESS 标志等效。

如果互斥锁的 *pshared* 属性设置为 PTHREAD_PROCESS_PRIVATE，则仅有那些由同一个进程创建的线程才能够处理该互斥锁。

pthread_mutexattr_setpshared 返回值

pthread_mutexattr_setpshared() 成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: 由 *mattr* 指定的值无效。

获取互斥锁的范围

pthread_mutexattr_getpshared(3C) 可用来返回由 pthread_mutexattr_setpshared() 定义的互斥锁变量的范围。

pthread_mutexattr_getpshared 语法

```
int    pthread_mutexattr_getpshared(pthread_mutexattr_t *mattr,

    int *pshared);

#include <pthread.h>



pthread_mutexattr_t mattr;

int pshared, ret;



/* get pshared of mutex */

ret = pthread_mutexattr_getpshared(&mattr, &pshared);
```

此函数可为属性对象 *mattr* 获取 *pshared* 的当前值。该值为 PTHREAD_PROCESS_SHARED 或 PTHREAD_PROCESS_PRIVATE。

pthread_mutexattr_getpshared 返回值

pthread_mutexattr_getpshared() 成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: 由 *mattr* 指定的值无效。

设置互斥锁类型的属性

pthread_mutexattr_settype(3C) 可用来设置互斥锁的 *type* 属性。

pthread_mutexattr_settype 语法

```
#include <pthread.h>



int pthread_mutexattr_settype(pthread_mutexattr_t  *attr , int type);
```

类型属性的缺省值为 PTHREAD_MUTEX_DEFAULT。

type 参数指定互斥锁的类型。以下列出了有效的互斥锁类型：

PTHREAD_MUTEX_NORMAL
描述:此类型的互斥锁不会检测死锁。如果线程在不首先解除互斥锁的情况下尝试重新锁定该互斥锁，则会产生死锁。尝试解除由其他线程锁定的互斥锁会产生不确定的行为。如果尝试解除锁定的互斥锁未锁定，则会产生不确定的行为。

PTHREAD_MUTEX_ERRORCHECK
描述:此类型的互斥锁可提供错误检查。如果线程在不首先解除锁定互斥锁的情况下尝试重新锁定该互斥锁，则会返回错误。如果线程尝试解除锁定的互斥锁已经由其他线程锁定，则会返回错误。如果线程尝试解除锁定的互斥锁未锁定，则会返回错误。

PTHREAD_MUTEX_RECURSIVE
描述:如果线程在不首先解除锁定互斥锁的情况下尝试重新锁定该互斥锁，则可成功锁定该互斥锁。 与 PTHREAD_MUTEX_NORMAL 类型的互斥锁不同，对此类型互斥锁进行重新锁定时不会产生死锁情况。多次锁定互斥锁需要进行相同次数的解除锁定才可以释放该锁，然后其他线程才能获取该互斥锁。如果线程尝试解除锁定的互斥锁已经由其他线程锁定，则会返回错误。 如果线程尝试解除锁定的互斥锁未锁定，则会返回错误。

PTHREAD_MUTEX_DEFAULT
描述:如果尝试以递归方式锁定此类型的互斥锁，则会产生不确定的行为。对于不是由调用线程锁定的此类型互斥锁，如果尝试对它解除锁定，则会产生不确定的行为。对于尚未锁定的此类型互斥锁，如果尝试对它解除锁定，也会产生不确定的行为。允许在实现中将该互斥锁映射到其他互斥锁类型之一。对于 Solaris 线程，PTHREAD_PROCESS_DEFAULT 会映射到 PTHREAD_PROCESS_NORMAL。

pthread_mutexattr_settype 返回值

如果运行成功，pthread_mutexattr_settype 函数会返回零。否则，将返回用于指明错误的错误号。

EINVAL
描述:值为 *type* 无效。

EINVAL
描述: *attr* 指定的值无效。

获取互斥锁的类型属性

pthread_mutexattr_gettype(3C) 可用来获取由 pthread_mutexattr_settype() 设置的互斥锁的 *type* 属性。

pthread_mutexattr_gettype 语法

```
#include <pthread.h>
```

```
int pthread_mutexattr_gettype(pthread_mutexattr_t  *attr , int  *type);
```

类型属性的缺省值为 PTHREAD_MUTEX_DEFAULT。

type 参数指定互斥锁的类型。有效的互斥锁类型包括：

- PTHREAD_MUTEX_NORMAL
- PTHREAD_MUTEX_ERRORCHECK
- PTHREAD_MUTEX_RECURSIVE
- PTHREAD_MUTEX_DEFAULT

有关每种类型的说明，请参见第 78 页中的 “pthread_mutexattr_settype 语法”。

pthread_mutexattr_gettype 返回值

如果成功完成，pthread_mutexattr_gettype() 会返回 0。其他任何返回值都表示出现了错误。

设置互斥锁属性的协议

pthread_mutexattr_setprotocol(3C) 可用来设置互斥锁属性对象的协议属性。

pthread_mutexattr_setprotocol 语法

```
#include <pthread.h>


int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);
```

attr 指示以前调用 pthread_mutexattr_init() 时创建的互斥锁属性对象。

protocol 可定义应用于互斥锁属性对象的协议。

pthread.h 中定义的 *protocol* 可以是以下值之一：PTHREAD_PRIO_NONE、PTHREAD_PRIO_INHERIT 或 PTHREAD_PRIO_PROTECT。

- PTHREAD_PRIO_NONE

  线程的优先级和调度不会受到互斥锁拥有权的影响。

- PTHREAD_PRIO_INHERIT

  此协议值（如 thrd1）会影响线程的优先级和调度。如果更高优先级的线程因 thrd1 所拥有的一个或多个互斥锁而被阻塞，而这些互斥锁是用 PTHREAD_PRIO_INHERIT 初始化的，则 thrd1 将以高于它的优先级或者所有正在等待这些互斥锁（这些互斥锁是 thrd1 指所拥有的互斥锁）的线程的最高优先级运行。

  如果 thrd1 因另一个线程 (thrd3) 拥有的互斥锁而被阻塞，则相同的优先级继承效应会以递归方式传播给 thrd3。

使用 PTHREAD_PRIO_INHERIT 可以避免优先级倒置。低优先级的线程持有较高优先级线程所需的锁时，便会发生优先级倒置。只有在较低优先级的线程释放该锁之后，较高优先级的线程才能继续使用该锁。设置 PTHREAD_PRIO_INHERIT，以便按与预期的优先级相反的优先级处理每个线程。

如果为使用协议属性值 PTHREAD_PRIO_INHERIT 初始化的互斥锁定义了 _POSIX_THREAD_PRIO_INHERIT，则互斥锁的属主失败时会执行以下操作。属主失败时的行为取决于 pthread_mutexattr_setrobust_np() 的 *robustness* 参数的值。

- 解除锁定互斥锁。
- 互斥锁的下一个属主将获取该互斥锁，并返回错误 EOWNERDEAD。
- 互斥锁的下一个属主会尝试使该互斥锁所保护的状态一致。如果上一个属主失败，则状态可能会不一致。如果属主成功使状态保持一致，则可针对该互斥锁调用 pthread_mutex_init() 并解除锁定该互斥锁。

注 – 如果针对以前初始化的但尚未销毁的互斥锁调用 pthread_mutex_init()，则该互斥锁不会重新初始化。

- 如果属主无法使状态保持一致，**请勿**调用 pthread_mutex_init()，而是解除锁定该互斥锁。在这种情况下，所有等待的线程都将被唤醒。以后对 pthread_mutex_lock() 的所有调用将无法获取互斥锁，并将返回错误代码 ENOTRECOVERABLE。现在，通过调用 pthread_mutex_destroy() 来取消初始化该互斥锁，即可使其状态保持一致。调用 pthread_mutex_init() 可重新初始化互斥锁。
- 如果已获取该锁的线程失败并返回 EOWNERDEAD，则下一个属主将获取该锁及错误代码 EOWNERDEAD。

- PTHREAD_PRIO_PROTECT

  当线程拥有一个或多个使用 PTHREAD_PRIO_PROTECT 初始化的互斥锁时，此协议值会影响其他线程（如 thrd2）的优先级和调度。thrd2 以其较高的优先级或者以 thrd2 拥有的所有互斥锁的最高优先级上限运行。基于被 thrd2 拥有的任一互斥锁阻塞的较高优先级线程对于 thrd2 的调度没有任何影响。

如果某个线程调用 sched_setparam() 来更改初始优先级，则调度程序不会采用新优先级将该线程移到调度队列末尾。

- 线程拥有使用 PTHREAD_PRIO_INHERIT 或 PTHREAD_PRIO_PROTECT 初始化的互斥锁
- 线程解除锁定使用 PTHREAD_PRIO_INHERIT 或 PTHREAD_PRIO_PROTECT 初始化的互斥锁

一个线程可以同时拥有多个混合使用 PTHREAD_PRIO_INHERIT 和 PTHREAD_PRIO_PROTECT 初始化的互斥锁。在这种情况下，该线程将以通过其中任一协议获取的最高优先级执行。

pthread_mutexattr_setprotocol 返回值

如果成功完成，pthread_mutexattr_setprotocol() 会返回 0。其他任何返回值都表示出现了错误。

如果出现以下任一情况，pthread_mutexattr_setprotocol() 将失败并返回对应的值。

ENOSYS
描述: 选项 _POSIX_THREAD_PRIO_INHERIT 和 _POSIX_THREAD_PRIO_PROTECT 均未定义并且该实现不支持此函数。

ENOTSUP
描述: *protocol* 指定的值不受支持。

如果出现以下任一情况，pthread_mutexattr_setprotocol() 可能会失败并返回对应的值。

EINVAL
描述: *attr* 或 *protocol* 指定的值无效。

EPERM
描述: 调用方无权执行该操作。

获取互斥锁属性的协议

pthread_mutexattr_getprotocol(3C) 可用来获取互斥锁属性对象的协议属性。

pthread_mutexattr_getprotocol 语法

```
#include <pthread.h>



int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr,

                                  int *protocol);
```

attr 指示以前调用 pthread_mutexattr_init() 时创建的互斥锁属性对象。

protocol 包含以下协议属性之一：PTHREAD_PRIO_NONE、PTHREAD_PRIO_INHERIT 或 PTHREAD_PRIO_PROTECT。

pthread_mutexattr_getprotocol 返回值

如果成功完成，pthread_mutexattr_getprotocol() 会返回 0。其他任何返回值都表示出现了错误。

如果出现以下情况，pthread_mutexattr_getprotocol() 将失败并返回对应的值。

ENOSYS
描述: _POSIX_THREAD_PRIO_INHERIT 选项和 _POSIX_THREAD_PRIO_PROTECT 选项均未定义并且该实现不支持此函数。

如果出现以下任一情况，pthread_mutexattr_getprotocol() 可能会失败并返回对应的值。

EINVAL
描述: *attr* 指定的值无效。

EPERM
描述: 调用方无权执行该操作。

设置互斥锁属性的优先级上限

pthread_mutexattr_setprioceiling(3C) 可用来设置互斥锁属性对象的优先级上限属性。

pthread_mutexattr_setprioceiling 语法

```
#include <pthread.h>


int pthread_mutexattr_setprioceiling(pthread_mutexatt_t *attr,
                int prioceiling, int *oldceiling);
```

attr 指示以前调用 pthread_mutexattr_init() 时创建的互斥锁属性对象。

prioceiling 指定已初始化互斥锁的优先级上限。优先级上限定义执行互斥锁保护的临界段时的最低优先级。*prioceiling* 位于 SCHED_FIFO 所定义的优先级的最大范围内。要避免优先级倒置，请将 *prioceiling* 设置为高于或等于可能会锁定特定互斥锁的所有线程的最高优先级。

oldceiling 包含以前的优先级上限值。

pthread_mutexattr_setprioceiling 返回值

如果成功完成，pthread_mutexattr_setprioceiling() 会返回 0。其他任何返回值都表示出现了错误。

如果出现以下任一情况，pthread_mutexattr_setprioceiling() 将失败并返回对应的值。

ENOSYS
描述: 选项 _POSIX_THREAD_PRIO_PROTECT 未定义并且该实现不支持此函数。

如果出现以下任一情况，pthread_mutexattr_setprioceiling() 可能会失败并返回对应的值。

EINVAL
描述: *attr* 或 *prioceiling* 指定的值无效。

EPERM
描述: 调用方无权执行该操作。

获取互斥锁属性的优先级上限

pthread_mutexattr_getprioceiling(3C) 可用来获取互斥锁属性对象的优先级上限属性。

pthread_mutexattr_getprioceiling 语法

```
#include <pthread.h>


int pthread_mutexattr_getprioceiling(const pthread_mutexatt_t *attr,
                        int *prioceiling);
```

attr 指定以前调用 pthread_mutexattr_init() 时创建的属性对象。

注 – 仅当定义了 _POSIX_THREAD_PRIO_PROTECT 符号时，*attr* 互斥锁属性对象才会包括优先级上限属性。

---

pthread_mutexattr_getprioceiling() 返回 *prioceiling* 中已初始化互斥锁的优先级上限。优先级上限定义执行互斥锁保护的临界段时的最低优先级。*prioceiling* 位于 SCHED_FIFO 所定义的优先级的最大范围内。要避免优先级倒置，请将 *prioceiling* 设置为高于或等于可能会锁定特定互斥锁的所有线程的最高优先级。

pthread_mutexattr_getprioceiling 返回值

如果成功完成，pthread_mutexattr_getprioceiling() 会返回 0。其他任何返回值都表示出现了错误。

如果出现以下任一情况，pthread_mutexattr_getprioceiling() 将失败并返回对应的值。

ENOSYS
描述: _POSIX_THREAD_PRIO_PROTECT 选项未定义并且该实现不支持此函数。

如果出现以下任一情况，pthread_mutexattr_getprioceiling() 可能会失败并返回对应的值。

EINVAL
描述: *attr* 指定的值无效。

EPERM
描述: 调用方无权执行该操作。

设置互斥锁的优先级上限

pthread_mutexattr_setprioceiling(3C) 可用来设置互斥锁的优先级上限。

pthread_mutex_setprioceiling 语法

```
#include <pthread.h>


int pthread_mutex_setprioceiling(pthread_mutex_t *mutex,
                    int prioceiling, int *old_ceiling);
```

`pthread_mutex_setprioceiling()` 可更改互斥锁 *mutex* 的优先级上限 *prioceiling*。`pthread_mutex_setprioceiling()` 可锁定互斥锁（如果未锁定的话），或者一直处于阻塞状态，直到 `pthread_mutex_setprioceiling()` 成功锁定该互斥锁，更改该互斥锁的优先级上限并将该互斥锁释放为止。锁定互斥锁的过程无需遵循优先级保护协议。

如果 `pthread_mutex_setprioceiling()` 成功，则将在 *old_ceiling* 中返回以前的优先级上限值。如果 `pthread_mutex_setprioceiling()` 失败，则互斥锁的优先级上限保持不变。

pthread_mutex_setprioceiling 返回值

如果成功完成，`pthread_mutex_setprioceiling()` 会返回 0。其他任何返回值都表示出现了错误。

如果出现以下情况，`pthread_mutexatt_setprioceiling()` 将失败并返回对应的值。

`ENOSYS`
描述: 选项 `_POSIX_THREAD_PRIO_PROTECT` 未定义并且该实现不支持此函数。

如果出现以下任一情况，`pthread_mutex_setprioceiling()` 可能会失败并返回对应的值。

`EINVAL`
描述: *prioceiling* 所请求的优先级超出了范围。

`EINVAL`
描述: *mutex* 指定的值不会引用当前存在的互斥锁。

`ENOSYS`
描述: 该实现不支持互斥锁的优先级上限协议。

`EPERM`
描述: 调用方无权执行该操作。

获取互斥锁的优先级上限

`pthread_mutexattr_getprioceiling`(3C) 可用来获取互斥锁的优先级上限。

pthread_mutex_getprioceiling 语法

```
#include <pthread.h>



int pthread_mutex_getprioceiling(const pthread_mutex_t *mutex,

                    int *prioceiling);
```

`pthread_mutex_getprioceiling()` 会返回 *mutex* 的优先级上限 *prioceiling*。

pthread_mutex_getprioceiling 返回值

如果成功完成，`pthread_mutex_getprioceiling()` 会返回 0。其他任何返回值都表示出现了错误。

如果出现以下任一情况，`pthread_mutexatt_getprioceiling()` 将失败并返回对应的值。

`ENOSYS`
描述: `_POSIX_THREAD_PRIO_PROTECT` 选项未定义并且该实现不支持此函数。

如果出现以下任一情况，`pthread_mutex_getprioceiling()` 可能会失败并返回对应的值。

`EINVAL`
描述: *mutex* 指定的值不会引用当前存在的互斥锁。

`ENOSYS`
描述: 该实现不支持互斥锁的优先级上限协议。

`EPERM`
描述: 调用方无权执行该操作。

设置互斥锁的强健属性

`pthread_mutexattr_setrobust_np`(3C) 可用来设置互斥锁属性对象的强健属性。

pthread_mutexattr_setrobust_np 语法

```
#include <pthread.h>



int pthread_mutexattr_setrobust_np(pthread_mutexattr_t *attr,

                                   int *robustness);
```

注 – 仅当定义了符号 _POSIX_THREAD_PRIO_INHERIT 时，pthread_mutexattr_setrobust_np() 才适用。

attr 指示以前通过调用 pthread_mutexattr_init() 创建的互斥锁属性对象。

robustness 定义在互斥锁的属主失败时的行为。pthread.h 中定义的 *robustness* 的值为 PTHREAD_MUTEX_ROBUST_NP 或 PTHREAD_MUTEX_STALLED_NP。缺省值为 PTHREAD_MUTEX_STALLED_NP。

- PTHREAD_MUTEX_ROBUST_NP

  如果互斥锁的属主失败，则以后对 pthread_mutex_lock() 的所有调用将以不确定的方式被阻塞。

- PTHREAD_MUTEX_STALLED_NP

  互斥锁的属主失败时，将会解除锁定该互斥锁。互斥锁的下一个属主将获取该互斥锁，并返回错误 EOWNWERDEAD。

  注 – 应用程序必须检查 pthread_mutex_lock() 的返回代码，查找返回错误 EOWNWERDEAD 的互斥锁。

  - 互斥锁的新属主应使该互斥锁所保护的状态保持一致。如果上一个属主失败，则互斥锁状态可能会不一致。
  - 如果新属主能够使状态保持一致，请针对该互斥锁调用 pthread_mutex_consistent_np()，并解除锁定该互斥锁。
  - 如果新属主**无法**使状态保持一致，**请勿**针对该互斥锁调用 pthread_mutex_consistent_np()，而是解除锁定该互斥锁。

    所有等待的线程都将被唤醒，以后对 pthread_mutex_lock() 的所有调用都将无法获取该互斥锁。返回代码为 ENOTRECOVERABLE。通过调用 pthread_mutex_destroy() 取消对互斥锁的初始化，并调用 pthread_mutex_int() 重新初始化该互斥锁，可使该互斥锁保持一致。

  如果已获取该锁的线程失败并返回 EOWNERDEAD，则下一个属主获取该锁时将返回代码 EOWNERDEAD。

pthread_mutexattr_setrobust_np 返回值

如果成功完成，pthread_mutexattr_setrobust_np() 会返回 0。其他任何返回值都表示出现了错误。

如果出现以下任一情况，pthread_mutexattr_setrobust_np() 将失败并返回对应的值。

ENOSYS
描述: 选项 _POSIX_THREAD_PRIO__INHERIT 未定义，或者该实现不支持 pthread_mutexattr_setrobust_np()。

ENOTSUP
描述: *robustness* 指定的值不受支持。

pthread_mutexattr_setrobust_np() 可能会在出现以下情况时失败：

EINVAL
描述: *attr* 或 *robustness* 指定的值无效。

获取互斥锁的强健属性

pthread_mutexattr_getrobust_np(3C) 可用来获取互斥锁属性对象的强健属性。

pthread_mutexattr_getrobust_np 语法

```
#include <pthread.h>



int pthread_mutexattr_getrobust_np(const pthread_mutexattr_t *attr,

                                                int *robustness);
```

注 – 仅当定义了符号 _POSIX_THREAD_PRIO_INHERIT 时，pthread_mutexattr_getrobust_np() 才适用。

---

attr 指示以前通过调用 pthread_mutexattr_init() 创建的互斥锁属性对象。

robustness 是互斥锁属性对象的强健属性值。

pthread_mutexattr_getrobust_np 返回值

如果成功完成，pthread_mutexattr_getrobust_np() 会返回 0。其他任何返回值都表示出现了错误。

如果出现以下任一情况，pthread_mutexattr_getrobust_np() 将失败并返回对应的值。

ENOSYS
描述: 选项 _POSIX_THREAD_PRIO__INHERIT 未定义，或者该实现不支持 pthread_mutexattr_getrobust_np()。

pthread_mutexattr_getrobust_np() 可能会在出现以下情况时失败：

EINVAL
描述: *attr* 或 *robustness* 指定的值无效。

使用互斥锁

表 4–3 列出了用来处理互斥锁的函数。

表 4–3 互斥锁的例程

操作	相关函数说明
初始化互斥锁	第 89 页中的 “pthread_mutex_init 语法”
使互斥锁保持一致	第 91 页中的 “pthread_mutex_consistent_np 语法”
锁定互斥锁	第 91 页中的 “pthread_mutex_lock 语法”
解除锁定互斥锁	第 93 页中的 “pthread_mutex_unlock 语法”
使用非阻塞互斥锁锁定	第 94 页中的 “pthread_mutex_trylock 语法”
销毁互斥锁	第 95 页中的 “pthread_mutex_destroy 语法”

缺省调度策略 SCHED_OTHER 不指定线程可以获取锁的顺序。如果多个线程正在等待一个互斥锁，则获取顺序是不确定的。出现争用时，缺省行为是按优先级顺序解除线程的阻塞。

初始化互斥锁

使用 pthread_mutex_init(3C) 可以使用缺省值初始化由 *mp* 所指向的互斥锁，还可以指定已经使用 pthread_mutexattr_init() 设置的互斥锁属性。*mattr* 的缺省值为 NULL。对于 Solaris 线程，请参见第 204 页中的 “mutex_init(3C) 语法” 。

pthread_mutex_init 语法

```
int    pthread_mutex_init(pthread_mutex_t *mp,

    const pthread_mutexattr_t *mattr);

#include <pthread.h>



pthread_mutex_t mp = PTHREAD_MUTEX_INITIALIZER;

pthread_mutexattr_t mattr;

int ret;



/* initialize a mutex to its default value */
```

```
ret = pthread_mutex_init(&mp, NULL);



/* initialize a mutex */

ret = pthread_mutex_init(&mp, &mattr);
```

如果互斥锁已初始化，则它会处于未锁定状态。互斥锁可以位于进程之间共享的内存中或者某个进程的专用内存中。

注 – 初始化互斥锁之前，必须将其所在的内存清零。

将 *mattr* 设置为 NULL 的效果与传递缺省互斥锁属性对象的地址相同，但是没有内存开销。

使用 PTHREAD_MUTEX_INITIALIZER 宏可以将以静态方式定义的互斥锁初始化为其缺省属性。

当其他线程正在使用某个互斥锁时，请勿重新初始化或销毁该互斥锁。如果任一操作没有正确完成，将会导致程序失败。如果要重新初始化或销毁某个互斥锁，则应用程序必须确保当前未使用该互斥锁。

pthread_mutex_init 返回值

pthread_mutex_init() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EBUSY
: **描述:** 该实现已检测到系统尝试重新初始化 *mp* 所引用的对象，即以前进行过初始化但 尚未销毁的互斥锁。

EINVAL
: **描述:** *mattr* 属性值无效。互斥锁尚未修改。

EFAULT
: **描述:** *mp* 所指向的互斥锁的地址无效。

使互斥保持一致

如果某个互斥锁的属主失败，该互斥锁可能会变为不一致。

使用 pthread_mutex_consistent_np 可使互斥对象 *mutex* 在其属主停止之后保持一致。

pthread_mutex_consistent_np 语法

```
#include <pthread.h>

int    pthread_mutex_consistent_np(pthread_mutex_t *mutex);
```

注 – 仅当定义了 _POSIX_THREAD_PRIO_INHERIT 符号时，pthread_mutex_consistent_np() 才适用，并且仅适用于使用协议属性值 PTHREAD_PRIO_INHERIT 初始化的互斥锁。

调用 pthread_mutex_lock() 会获取不一致的互斥锁。EOWNWERDEAD 返回值表示出现不一致的互斥锁。

持有以前通过调用 pthread_mutex_lock() 获取的互斥锁时可调用 pthread_mutex_consistent_np()。

如果互斥锁的属主失败，则该互斥锁保护的临界段可能会处于不一致状态。在这种情况下，仅当互斥锁保护的临界段可保持一致时，才能使该互斥锁保持一致。

针对互斥锁调用 pthread_mutex_lock()、pthread_mutex_unlock() 和 pthread_mutex_trylock() 会以正常方式进行。

对于**不**一致或者未持有的互斥锁，pthread_mutex_consistent_np() 的行为是不确定的。

pthread_mutex_consistent_np 返回值

pthread_mutex_consistent_np() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。

pthread_mutex_consistent_np() 会在出现以下情况时失败：

ENOSYS
描述: 选项 _POSIX_THREAD_PRIO_INHERIT 未定义，或者该实现不支持 pthread_mutex_consistent_np()。

pthread_mutex_consistent_np() 可能会在出现以下情况时失败：

EINVAL
描述: *mattr* 属性值无效。

锁定互斥锁

使用 pthread_mutex_lock(3C) 可以锁定 *mutex* 所指向的互斥锁。

pthread_mutex_lock 语法

```
int    pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
#include <pthread.h>



pthread_mutex_t mutex;

int ret;



ret = pthread_ mutex_lock(&mp); /* acquire the mutex */
```

当 pthread_mutex_lock() 返回时，该互斥锁已被锁定。调用线程是该互斥锁的属主。如果该互斥锁已被另一个线程锁定和拥有，则调用线程将阻塞，直到该互斥锁变为可用为止。对于 Solaris 线程，请参见第 207 页中的 “mutex_lock 语法” 。

如果互斥锁类型为 PTHREAD_MUTEX_NORMAL，则不提供死锁检测。尝试重新锁定互斥锁会导致死锁。如果某个线程尝试解除锁定的互斥锁不是由该线程锁定或未锁定，则将产生不确定的行为。

如果互斥锁类型为 PTHREAD_MUTEX_ERRORCHECK，则会提供错误检查。如果某个线程尝试重新锁定的互斥锁已经由该线程锁定，则将返回错误。如果某个线程尝试解除锁定的互斥锁不是由该线程锁定或者未锁定，则将返回错误。

如果互斥锁类型为 PTHREAD_MUTEX_RECURSIVE，则该互斥锁会保留锁定计数这一概念。线程首次成功获取互斥锁时，锁定计数会设置为 1。线程每重新锁定该互斥锁一次，锁定计数就增加 1。线程每解除锁定该互斥锁一次，锁定计数就减小 1。 锁定计数达到 0 时，该互斥锁即可供其他线程获取。如果某个线程尝试解除锁定的互斥锁不是由该线程锁定或者未锁定，则将返回错误。

如果互斥锁类型是 PTHREAD_MUTEX_DEFAULT，则尝试以递归方式锁定该互斥锁将产生不确定的行为。对于不是由调用线程锁定的互斥锁，如果尝试解除对它的锁定，则会产生不确定的行为。如果尝试解除锁定尚未锁定的互斥锁，则会产生不确定的行为。

pthread_mutex_lock 返回值

pthread_mutex_lock() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EAGAIN
描述: 由于已超出了互斥锁递归锁定的最大次数，因此无法获取该互斥锁。

EDEADLK
描述: 当前线程已经拥有互斥锁。

如果定义了 _POSIX_THREAD_PRIO_INHERIT 符号，则会使用协议属性值 PTHREAD_PRIO_INHERIT 对互斥锁进行初始化。此外，如果 pthread_mutexattr_setrobust_np() 的 *robustness* 参数是 PTHREAD_MUTEX_ROBUST_NP，则该函数将失败并返回以下值之一：

EOWNERDEAD
描述: 该互斥锁的最后一个属主在持有该互斥锁时失败。该互斥锁现在由调用方拥有。调用方必须尝试使该互斥锁所保护的状态一致。

如果调用方能够使状态保持一致，请针对该互斥锁调用 `pthread_mutex_consistent_np()` 并解除锁定该互斥锁。以后对 `pthread_mutex_lock()` 的调用都将正常进行。

如果调用方无法使状态保持一致，请勿针对该互斥锁调用 `pthread_mutex_init()`，但要解除锁定该互斥锁。以后调用 `pthread_mutex_lock()` 时将无法获取该互斥锁，并且将返回错误代码 `ENOTRECOVERABLE`。

如果获取该锁的属主失败并返回 `EOWNERDEAD`，则下一个属主获取该锁时将返回 `EOWNERDEAD`。

ENOTRECOVERABLE
描述: 尝试获取的互斥锁正在保护某个状态，此状态由于该互斥锁以前的属主在持有该锁时失败而导致不可恢复。尚未获取该互斥锁。如果满足以下条件，则可能出现此不可恢复的情况：

- 以前获取该锁时返回 `EOWNERDEAD`
- 该属主无法清除此状态
- 该属主已经解除锁定了该互斥锁，但是没有使互斥锁状态保持一致

ENOMEM
描述: 已经超出了可同时持有的互斥锁数目的限制。

解除锁定互斥锁

使用 `pthread_mutex_unlock`(3C) 可以解除锁定 *mutex* 所指向的互斥锁。 对于 Solaris 线程，请参见第 207 页中的 “mutex_unlock 语法” 。

pthread_mutex_unlock 语法

```
int    pthread_mutex_unlock(pthread_mutex_t *mutex);
```

```
#include <pthread.h>



pthread_mutex_t mutex;

int ret;



ret = pthread_mutex_unlock(&mutex); /* release the mutex */
```

pthread_mutex_unlock() 可释放 *mutex* 引用的互斥锁对象。互斥锁的释放方式取决于互斥锁的类型属性。如果调用 pthread_mutex_unlock() 时有多个线程被 *mutex* 对象阻塞，则互斥锁变为可用时调度策略可确定获取该互斥锁的线程。对于 PTHREAD_MUTEX_RECURSIVE 类型的互斥锁，当计数达到零并且调用线程不再对该互斥锁进行任何锁定时，该互斥锁将变为可用。

pthread_mutex_unlock 返回值

pthread_mutex_unlock() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EPERM
描述: 当前线程不拥有互斥锁。

使用非阻塞互斥锁锁定

使用 pthread_mutex_trylock(3C) 可以尝试锁定 *mutex* 所指向的互斥锁。对于 Solaris 线程，请参见第 208 页中的 “mutex_trylock 语法”。

pthread_mutex_trylock 语法

```
int    pthread_mutex_trylock(pthread_mutex_t *mutex);
```

```
#include <pthread.h>



pthread_mutex_t mutex;

int ret;



ret = pthread_mutex_trylock(&mutex); /* try to lock the mutex */
```

pthread_mutex_trylock() 是 pthread_mutex_lock() 的非阻塞版本。如果 *mutex* 所引用的互斥对象当前被任何线程（包括当前线程）锁定，则将立即返回该调用。否则，该互斥锁将处于锁定状态，调用线程是其属主。

pthread_mutex_trylock 返回值

pthread_mutex_trylock() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EBUSY
描述: 由于 *mutex* 所指向的互斥锁已锁定，因此无法获取该互斥锁。

EAGAIN
描述: 由于已超出了 *mutex* 的递归锁定最大次数，因此无法获取该互斥锁。

如果定义了 _POSIX_THREAD_PRIO_INHERIT 符号，则会使用协议属性值 PTHREAD_PRIO_INHERIT 对互斥锁进行初始化。此外，如果 pthread_mutexattr_setrobust_np() 的 *robustness* 参数是 PTHREAD_MUTEX_ROBUST_NP，则该函数将失败并返回以下值之一：

EOWNERDEAD
描述: 该互斥锁的最后一个属主在持有该互斥锁时失败。该互斥锁现在由调用方拥有。调用方必须尝试使该互斥锁所保护的状态一致。

如果调用方能够使状态保持一致，请针对该互斥锁调用 pthread_mutex_consistent_np() 并解除锁定该互斥锁。以后对 pthread_mutex_lock() 的调用都将正常进行。

如果调用方无法使状态保持一致，请勿针对该互斥锁调用 pthread_mutex_init()，而要解除锁定该互斥锁。以后调用 pthread_mutex_trylock() 时将无法获取该互斥锁，并且将返回错误代码 ENOTRECOVERABLE。

如果已获取该锁的属主失败并返回 EOWNERDEAD，则下一个属主获取该锁时返回 EOWNERDEAD。

ENOTRECOVERABLE
描述: 尝试获取的互斥锁正在保护某个状态，此状态由于该互斥锁以前的属主在持有该锁时失败而导致不可恢复。尚未获取该互斥锁。 以下条件下可能会出现此情况：

- 以前获取该锁时返回 EOWNERDEAD
- 该属主无法清除此状态
- 该属主已经解除锁定了该互斥锁，但是没有使互斥锁状态保持一致

ENOMEM
描述: 已经超出了可同时持有的互斥锁数目的限制。

销毁互斥锁

使用 pthread_mutex_destroy(3C) 可以销毁与 *mp* 所指向的互斥锁相关联的任何状态。 对于 Solaris 线程，请参见第 206 页中的 “mutex_destroy 语法”。

pthread_mutex_destroy 语法

```
int    pthread_mutex_destroy(pthread_mutex_t *mp);
```

```
#include <pthread.h>



pthread_mutex_t mp;
```

```
int ret;


ret = pthread_mutex_destroy(&mp); /* mutex is destroyed */
```

请注意，没有释放用来存储互斥锁的空间。

pthread_mutex_destroy 返回值

`pthread_mutex_destroy()` 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

`EINVAL`
描述: *mp* 指定的值不会引用已初始化的互斥锁对象。

互斥锁定的代码示例

示例 4–1 显示了使用互斥锁定的一些代码段。

示例 4–1 互斥锁示例

```
#include <pthread.h>


pthread_mutex_t count_mutex;

long long count;


void

increment_count()

{

        pthread_mutex_lock(&count_mutex);

    count = count + 1;

        pthread_mutex_unlock(&count_mutex);

}
```

示例 4–1 互斥锁示例　　（续）

```
long long

get_count()

{

    long long c;



    pthread_mutex_lock(&count_mutex);

        c = count;

    pthread_mutex_unlock(&count_mutex);

        return (c);

}
```

示例 4–1 中的两个函数将互斥锁用于不同目的。`increment_count()` 函数使用互斥锁确保对共享变量进行原子更新。`get_count()` 函数使用互斥锁保证以原子方式读取 64 位数量 *count*。在 32 位体系结构上，`long long` 实际上是两个 32 位数量。

读取整数值时执行的是原子运算，因为整数是大多数计算机中常见的字长。

锁分层结构的使用示例

有时，可能需要同时访问两个资源。您可能正在使用其中的一个资源，随后发现还需要另一个资源。如果两个线程尝试声明这两个资源，但是以不同的顺序锁定与这些资源相关联的互斥锁，则会出现问题。例如，如果两个线程分别锁定互斥锁 1 和互斥锁 2，则每个线程尝试锁定另一个互斥锁时，将会出现死锁。示例 4–2 说明了可能的死锁情况。

示例 4–2 死锁

线程 1	线程 2
`pthread_mutex_lock(&m1);`	`pthread_mutex_lock(&m2);`
/* use resource 1 */	/* use resource 2 */
`pthread_mutex_lock(&m2);`	`pthread_mutex_lock(&m1);`
/* use resources 1 and 2 */	/* use resources 1 and 2 */
`pthread_mutex_unlock(&m2);`	`pthread_mutex_unlock(&m1);`
`pthread_mutex_unlock(&m1);`	`pthread_mutex_unlock(&m2);`

避免此问题的最佳方法是，确保线程在锁定多个互斥锁时，以同样的顺序进行锁定。如果始终按照规定的顺序锁定，就不会出现死锁。此方法称为锁分层结构，它通过为互斥锁指定逻辑编号来对这些锁进行排序。

另外，请注意以下限制：如果您持有的任何互斥锁其指定编号大于 *n*，则不能提取指定编号为 *n* 的互斥锁。

但是，不能始终使用此方法。有时，必须按照与规定不同的顺序提取互斥锁。要防止在这种情况下出现死锁，请使用 `pthread_mutex_trylock()`。如果线程发现无法避免死锁时，该线程必须释放其互斥锁。

示例 4–3 条件锁定

线程 1	线程 2
`pthread_mutex_lock(&m1);` `pthread_mutex_lock(&m2);`	`for (; ;)`
/* no processing */	`{ pthread_mutex_lock(&m2);`
`pthread_mutex_unlock(&m2);`	`if(pthread_mutex_trylock(&m1)==0)`
`pthread_mutex_unlock(&m1);`	/* got it */
	`break;`
	/* didn't get it */
	`pthread_mutex_unlock(&m2);`
	`}`
	/* get locks; no processing */
	`pthread_mutex_unlock(&m1);`
	`pthread_mutex_unlock(&m2);`

在示例 4–3 中，线程 1 按照规定的顺序锁定互斥锁，但是线程 2 不按顺序提取互斥锁。要确保不会出现死锁，线程 2 必须非常小心地提取互斥锁 1。如果线程 2 在等待该互斥锁释放时被阻塞，则线程 2 可能刚才已经与线程 1 进入了死锁状态。

要确保线程 2 不会进入死锁状态，线程 2 需要调用 `pthread_mutex_trylock()`，此函数可在该互斥锁可用时提取它。如果该互斥锁不可用，线程 2 将立即返回并报告提取失败。此时，线程 2 必须释放互斥锁 2。线程 1 现在会锁定互斥锁 2，然后释放互斥锁 1 和互斥锁 2。

嵌套锁定和单链接列表的结合使用示例

示例 4–4 和示例 4–5 说明了如何同时提取三个锁。通过按照规定的顺序提取锁可避免出现死锁。

示例 4–4 单链接列表结构

```
typedef struct node1 {

    int value;

    struct node1 *link;

    pthread_mutex_t lock;

} node1_t;



node1_t ListHead;
```

本示例针对每个包含一个互斥锁的节点使用单链接列表结构。要将某个节点从列表中删除，请首先从 *ListHead* 开始搜索列表，直到找到所需的节点为止。*ListHead* 永远不会被删除。

要防止执行此搜索时产生并发删除，请在访问每个节点的任何内容之前先锁定该节点。由于所有的搜索都从 *ListHead* 开始，并且始终按照列表中的顺序提取锁，因此不会出现死锁。

因为更改涉及到两个节点，所以找到所需的节点之后，请锁定该节点及其前序节点。因为前序节点的锁总是最先提取，所以可再次防止出现死锁。示例 4–5 说明如何使用 C 代码来删除单链接列表中的项。

示例 4–5 单链接列表和嵌套锁定

```
node1_t *delete(int value)

{
```

示例 4–5 单链接列表和嵌套锁定 （续）

```
    node1_t *prev, *current;



    prev = &ListHead;

    pthread_mutex_lock(&prev->lock);

    while ((current = prev->link) != NULL) {

        pthread_mutex_lock(&current->lock);

        if (current->value == value) {

            prev->link = current->link;

            pthread_mutex_unlock(&current->lock);

            pthread_mutex_unlock(&prev->lock);

            current->link = NULL;

            return(current);

        }

        pthread_mutex_unlock(&prev->lock);

        prev = current;

    }

    pthread_mutex_unlock(&prev->lock);

    return(NULL);

}
```

嵌套锁定和循环链接列表的示例

示例 4–6 通过将以前的列表结构转换为循环列表来对其进行修改。由于不再存在用于标识的头节点，因该线程可以与特定的节点相关联，并可针对该节点及其邻居执行操作。锁分层结构在此处不适用，因为链接之后的分层结构明显是循环结构。

示例 4–6 循环链接列表结构

```
typedef struct node2 {

    int value;

    struct node2 *link;

    pthread_mutex_t lock;

} node2_t;
```

以下的 C 代码用来获取两个节点上的锁并执行涉及到这两个锁的操作。

示例 4–7 循环链接列表和嵌套锁定

```
void Hit Neighbor(node2_t *me) {

    while (1) {

        pthread_mutex_lock(&me->lock);

        if (pthread_mutex_lock(&me->link->lock)!= 0) {

            /* failed to get lock */

            pthread_mutex_unlock(&me->lock);

            continue;

        }

        break;

    }

    me->link->value += me->value;

    me->value /=2;

    pthread_mutex_unlock(&me->link->lock);

    pthread_mutex_unlock(&me->lock);

}
```

条件变量属性

使用条件变量可以以原子方式阻塞线程，直到某个特定条件为真为止。条件变量始终与互斥锁一起使用。

使用条件变量，线程可以以原子方式阻塞，直到满足某个条件为止。对条件的测试是在互斥锁（互斥）的保护下进行的。

如果条件为假，线程通常会基于条件变量阻塞，并以原子方式释放等待条件变化的互斥锁。如果另一个线程更改了条件，该线程可能会向相关的条件变量发出信号，从而使一个或多个等待的线程执行以下操作：

- 唤醒
- 再次获取互斥锁
- 重新评估条件

在以下情况下，条件变量可用于在进程之间同步线程：

- 线程是在可以写入的内存中分配的
- 内存由协作进程共享

调度策略可确定唤醒阻塞线程的方式。对于缺省值 SCHED_OTHER，将按优先级顺序唤醒线程。

必须设置和初始化条件变量的属性，然后才能使用条件变量。表 4–4 列出了用于处理条件变量属性的函数。

表 4–4 条件变量属性

操作	函数说明
初始化条件变量属性	第 103 页中的 “pthread_condattr_init 语法”
删除条件变量属性	第 104 页中的 “pthread_condattr_destroy 语法”
设置条件变量的范围	第 104 页中的 “pthread_condattr_setpshared 语法”
获取条件变量的范围	第 105 页中的 “pthread_condattr_getpshared 语法”

表 4–5 中显示了定义条件变量的范围时 Solaris 线程和 POSIX 线程之间的差异。

表 4–5 条件变量范围比较

Solaris	POSIX	定义
USYNC_PROCESS	PTHREAD_PROCESS_SHARED	用于同步该进程和其他进程中的线程
USYNC_THREAD	PTHREAD_PROCESS_PRIVATE	用于仅同步该进程中的线程

初始化条件变量属性

使用 pthread_condattr_init(3C) 可以将与该对象相关联的属性初始化为其缺省值。在执行过程中，线程系统会为每个属性对象分配存储空间。

pthread_condattr_init 语法

```
int    pthread_condattr_init(pthread_condattr_t *cattr);

#include <pthread.h>

pthread_condattr_t cattr;

int ret;



/* initialize an attribute to default value */

ret = pthread_condattr_init(&cattr);
```

调用此函数时，*pshared* 属性的缺省值为 PTHREAD_PROCESS_PRIVATE。*pshared* 的该值表示可以在进程内使用已初始化的条件变量。

cattr 的数据类型为 opaque，其中包含一个由系统分配的属性对象。*cattr* 范围可能的值为 PTHREAD_PROCESS_PRIVATE 和 PTHREAD_PROCESS_SHARED。PTHREAD_PROCESS_PRIVATE 是缺省值。

条件变量属性必须首先由 pthread_condattr_destroy(3C) 重新初始化后才能重用。pthread_condattr_init() 调用会返回指向类型为 opaque 的对象的指针。如果未销毁该对象，则会导致内存泄漏。

pthread_condattr_init 返回值

pthread_condattr_init() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

ENOMEM
描述: 分配的内存不足，无法初始化线程属性对象。

EINVAL
描述: *cattr* 指定的值无效。

删除条件变量属性

使用 pthread_condattr_destroy(3C) 可以删除存储并使属性对象无效。

pthread_condattr_destroy 语法

```
int    pthread_condattr_destroy(pthread_condattr_t *cattr);

#include <pthread.h>

pthread_condattr_t cattr;

int ret;



/* destroy an attribute */

ret

 = pthread_condattr_destroy(&cattr);
```

pthread_condattr_destroy 返回值

pthread_condattr_destroy() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *cattr* 指定的值无效。

设置条件变量的范围

pthread_condattr_setpshared(3C) 可用来将条件变量的范围设置为进程专用（进程内）或系统范围内（进程间）。

pthread_condattr_setpshared 语法

```
int    pthread_condattr_setpshared(pthread_condattr_t *cattr,

    int pshared);

#include <pthread.h>



pthread_condattr_t cattr;

int ret;
```

```
/* all processes */

ret = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_SHARED);



/* within a process */

ret = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_PRIVATE);
```

如果 *pshared* 属性在共享内存中设置为 PTHREAD_PROCESS_SHARED，则其所创建的条件变量可以在多个进程中的线程之间共享。此行为与最初的 Solaris 线程实现中 mutex_init() 中的 USYNC_PROCESS 标志等效。

如果互斥锁的 *pshared* 属性设置为 PTHREAD_PROCESS_PRIVATE，则仅有那些由同一个进程创建的线程才能够处理该互斥锁。PTHREAD_PROCESS_PRIVATE 是缺省值。PTHREAD_PROCESS_PRIVATE 所产生的行为与在最初的 Solaris 线程的 cond_init() 调用中使用 USYNC_THREAD 标志相同。PTHREAD_PROCESS_PRIVATE 的行为与局部条件变量相同。PTHREAD_PROCESS_SHARED 的行为与全局条件变量等效。

pthread_condattr_setpshared 返回值

pthread_condattr_setpshared() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *cattr* 或 *pshared* 的值无效。

获取条件变量的范围

pthread_condattr_getpshared(3C) 可用来获取属性对象 *cattr* 的 *pshared* 的当前值。

pthread_condattr_getpshared 语法

```
int    pthread_condattr_getpshared(const pthread_condattr_t *cattr,

    int *pshared);

#include <pthread.h>



pthread_condattr_t cattr;

int pshared;
```

```
int ret;


/* get pshared value of condition variable */

ret = pthread_condattr_getpshared(&cattr, &pshared);
```

属性对象的值为 PTHREAD_PROCESS_SHARED 或 PTHREAD_PROCESS_PRIVATE。

pthread_condattr_getpshared 返回值

pthread_condattr_getpshared() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *cattr* 的值无效。

使用条件变量

本节介绍如何使用条件变量。表 4–6 列出了可用的函数。

表 4–6 条件变量函数

操作	相关函数说明
初始化条件变量	第 107 页中的 “pthread_cond_init 语法”
基于条件变量阻塞	第 108 页中的 “pthread_cond_wait 语法”
解除阻塞特定线程	第 109 页中的 “pthread_cond_signal 语法”
在指定的时间之前阻塞	第 111 页中的 “pthread_cond_timedwait 语法”
在指定的时间间隔内阻塞	第 113 页中的 “pthread_cond_reltimedwait_np 语法”
解除阻塞所有线程	第 114 页中的 “pthread_cond_broadcast 语法”
销毁条件变量状态	第 116 页中的 “pthread_cond_destroy 语法”

初始化条件变量

使用 pthread_cond_init(3C) 可以将 *cv* 所指示的条件变量初始化为其缺省值，或者指定已经使用 pthread_condattr_init() 设置的条件变量属性。

pthread_cond_init 语法

```
int    pthread_cond_init(pthread_cond_t *cv,

    const pthread_condattr_t *cattr);

#include <pthread.h>



pthread_cond_t cv;

pthread_condattr_t cattr;

int ret;



/* initialize a condition variable to its default value */

ret = pthread_cond_init(&cv, NULL);



/* initialize a condition variable */

ret = pthread_cond_init(&cv, &cattr);
```

cattr 设置为 NULL。将 *cattr* 设置为 NULL 与传递缺省条件变量属性对象的地址等效，但是没有内存开销。对于 Solaris 线程，请参见第 209 页中的 “cond_init 语法” 。

使用 PTHREAD_COND_INITIALIZER 宏可以将以静态方式定义的条件变量初始化为其缺省属性。PTHREAD_COND_INITIALIZER 宏与动态分配具有 null 属性的 pthread_cond_init() 等效，但是不进行错误检查。

多个线程决不能同时初始化或重新初始化同一个条件变量。如果要重新初始化或销毁某个条件变量，则应用程序必须确保该条件变量未被使用。

pthread_cond_init 返回值

pthread_cond_init() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EINVAL
描述: *cattr* 指定的值无效。

EBUSY
描述: 条件变量处于使用状态。

EAGAIN
描述: 必要的资源不可用。

ENOMEM
描述: 内存不足，无法初始化条件变量。

基于条件变量阻塞

使用 pthread_cond_wait(3C) 可以以原子方式释放 *mp* 所指向的互斥锁，并导致调用线程基于 *cv* 所指向的条件变量阻塞。对于 Solaris 线程，请参见第 211 页中的 “cond_wait 语法” 。

pthread_cond_wait 语法

```
int    pthread_cond_wait(pthread_cond_t *cv,pthread_mutex_t *mutex);

#include <pthread.h>



pthread_cond_t cv;

pthread_mutex_t mp;

int ret;



/* wait on condition variable */

ret = pthread_cond_wait(&cv, &mp);
```

阻塞的线程可以通过 pthread_cond_signal() 或 pthread_cond_broadcast() 唤醒，也可以在信号传送将其中断时唤醒。

不能通过 pthread_cond_wait() 的返回值来推断与条件变量相关联的条件的值的任何变化。必须重新评估此类条件。

pthread_cond_wait() 例程每次返回结果时调用线程都会锁定并且拥有互斥锁，即使返回错误时也是如此。

该条件获得信号之前，该函数一直被阻塞。该函数会在被阻塞之前以原子方式释放相关的互斥锁，并在返回之前以原子方式再次获取该互斥锁。

通常，对条件表达式的评估是在互斥锁的保护下进行的。如果条件表达式为假，线程会基于条件变量阻塞。然后，当该线程更改条件值时，另一个线程会针对条件变量发出信号。这种变化会导致所有等待该条件的线程解除阻塞并尝试再次获取互斥锁。

必须重新测试导致等待的条件，然后才能从 pthread_cond_wait() 处继续执行。唤醒的线程重新获取互斥锁并从 pthread_cond_wait() 返回之前，条件可能会发生变化。等待线程可能并未真正唤醒。建议使用的测试方法是，将条件检查编写为调用 pthread_cond_wait() 的 while() 循环。

```
    pthread_mutex_lock();

        while(condition_is_false)

            pthread_cond_wait();

    pthread_mutex_unlock();
```

如果有多个线程基于该条件变量阻塞，则无法保证按特定的顺序获取互斥锁。

注 – pthread_cond_wait() 是取消点。如果取消处于暂挂状态，并且调用线程启用了取消功能，则该线程会终止，并在继续持有该锁的情况下开始执行清除处理程序。

---

pthread_cond_wait 返回值

pthread_cond_wait() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *cv* 或 *mp* 指定的值无效。

解除阻塞一个线程

对于基于 *cv* 所指向的条件变量阻塞的线程，使用 pthread_cond_signal(3C) 可以解除阻塞该线程。对于 Solaris 线程，请参见第 213 页中的 “cond_signal 语法” 。

pthread_cond_signal 语法

```
int    pthread_cond_signal(pthread_cond_t *cv);

#include <pthread.h>



pthread_cond_t cv;

int ret;
```

```
/* one condition variable is signaled */

ret = pthread_cond_signal(&cv);
```

应在互斥锁的保护下修改相关条件，该互斥锁用于获得信号的条件变量中。否则，可能在条件变量的测试和 pthread_cond_wait() 阻塞之间修改该变量，这会导致无限期等待。

调度策略可确定唤醒阻塞线程的顺序。对于 SCHED_OTHER，将按优先级顺序唤醒线程。

如果没有任何线程基于条件变量阻塞，则调用 pthread_cond_signal() 不起作用。

示例 4–8 使用 pthread_cond_wait() 和 pthread_cond_signal()

```
pthread_mutex_t count_lock;

pthread_cond_t count_nonzero;

unsigned count;



decrement_count()

{

    pthread_mutex_lock(&count_lock);

    while (count == 0)

        pthread_cond_wait(&count_nonzero, &count_lock);

    count = count - 1;

    pthread_mutex_unlock(&count_lock);

}



increment_count()

{

    pthread_mutex_lock(&count_lock);

    if (count == 0)

        pthread_cond_signal(&count_nonzero);
```

示例 4–8 使用 pthread_cond_wait() 和 pthread_cond_signal() （续）

```
    count = count + 1;

    pthread_mutex_unlock(&count_lock);

}
```

pthread_cond_signal 返回值

pthread_cond_signal() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *cv* 指向的地址非法。

示例 4–8 说明了如何使用 pthread_cond_wait() 和 pthread_cond_signal()。

在指定的时间之前阻塞

pthread_cond_timedwait(3C) 的用法与 pthread_cond_wait() 的用法基本相同，区别在于在由 *abstime* 指定的时间之后 pthread_cond_timedwait() 不再被阻塞。

pthread_cond_timedwait 语法

```
int    pthread_cond_timedwait(pthread_cond_t *cv,

    pthread_mutex_t *mp, const struct timespec *abstime);

#include <pthread.h>

#include <time.h>



pthread_cond_t cv;

pthread_mutex_t  mp;

timestruct_t abstime;

int ret;



/* wait on condition variable */
```

ret = pthread_cond_timedwait(&*cv*, &*mp*, &*abstime*);

pthread_cond_timewait() 每次返回时调用线程都会锁定并且拥有互斥锁，即使 pthread_cond_timedwait() 返回错误时也是如此。 对于 Solaris 线程，请参见第 211 页中的“cond_timedwait 语法”。

pthread_cond_timedwait() 函数会一直阻塞，直到该条件获得信号，或者最后一个参数所指定的时间已过为止。

注 – pthread_cond_timedwait() 也是取消点。

示例 4–9 计时条件等待

```
pthread_timestruc_t to;

pthread_mutex_t m;

pthread_cond_t c;

...

pthread_mutex_lock(&m);

to.tv_sec = time(NULL) + TIMEOUT;

to.tv_nsec = 0;

while (cond == FALSE) {

    err = pthread_cond_timedwait(&c, &m, &to);

    if (err == ETIMEDOUT) {

        /* timeout, do something */

        break;

    }

}

pthread_mutex_unlock(&m);
```

pthread_cond_timedwait 返回值

pthread_cond_timedwait() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EINVAL
描述: *cv* 或 *abstime* 指向的地址非法。

ETIMEDOUT
描述: *abstime* 指定的时间已过。

超时会指定为当天时间，以便在不重新计算值的情况下高效地重新测试条件，如示例 4–9 中所示。

在指定的时间间隔内阻塞

pthread_cond_reltimedwait_np(3C) 的用法与 pthread_cond_timedwait() 的用法基本相同，唯一的区别在于 pthread_cond_reltimedwait_np() 会采用相对时间间隔而不是将来的绝对时间作为其最后一个参数的值。

pthread_cond_reltimedwait_np 语法

```
int  pthread_cond_reltimedwait_np(pthread_cond_t *cv,

    pthread_mutex_t *mp,

   const struct timespec *reltime);

#include <pthread.h>

#include <time.h>



pthread_cond_t cv;

pthread_mutex_t mp;

timestruct_t reltime;

int ret;



/* wait on condition variable */

ret = pthread_cond_reltimedwait_np(&cv, &mp, &reltime);
```

pthread_cond_reltimedwait_np() 每次返回时调用线程都会锁定并且拥有互斥锁，即使 pthread_cond_reltimedwait_np() 返回错误时也是如此。对于 Solaris 线程，请参见 cond_reltimedwait(3C)。pthread_cond_reltimedwait_np() 函数会一直阻塞，直到该条件获得信号，或者最后一个参数指定的时间间隔已过为止。

注 – pthread_cond_reltimedwait_np() 也是取消点。

pthread_cond_reltimedwait_np 返回值

pthread_cond_reltimedwait_np() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EINVAL
描述: *cv* 或 *reltime* 指示的地址非法。

ETIMEDOUT
描述: *reltime* 指定的时间间隔已过。

解除阻塞所有线程

对于基于 *cv* 所指向的条件变量阻塞的线程，使用 pthread_cond_broadcast(3C) 可以解除阻塞所有这些线程，这由 pthread_cond_wait() 来指定。

pthread_cond_broadcast 语法

```
int    pthread_cond_broadcast(pthread_cond_t *cv);
```

```
#include <pthread.h>



pthread_cond_t cv;

int ret;



/* all condition variables are signaled */

ret = pthread_cond_broadcast(&cv);
```

如果没有任何线程基于该条件变量阻塞，则调用 pthread_cond_broadcast() 不起作用。对于 Solaris 线程，请参见第 213 页中的 “cond_broadcast 语法” 。

由于 pthread_cond_broadcast() 会导致所有基于该条件阻塞的线程再次争用互斥锁，因此请谨慎使用 pthread_cond_broadcast()。例如，通过使用 pthread_cond_broadcast()，线程可在资源释放后争用不同的资源量，如示例 4–10 中所示。

示例 4–10 条件变量广播

```
pthread_mutex_t rsrc_lock;

pthread_cond_t rsrc_add;

unsigned int resources;



get_resources(int amount)

{

    pthread_mutex_lock(&rsrc_lock);

    while (resources < amount) {

        pthread_cond_wait(&rsrc_add, &rsrc_lock);

    }

    resources -= amount;

    pthread_mutex_unlock(&rsrc_lock);

}



add_resources(int amount)

{

    pthread_mutex_lock(&rsrc_lock);

    resources += amount;

    pthread_cond_broadcast(&rsrc_add);

    pthread_mutex_unlock(&rsrc_lock);

}
```

请注意，在 `add_resources()` 中，首先更新 *resources* 还是首先在互斥锁中调用 `pthread_cond_broadcast()` 无关紧要。

应在互斥锁的保护下修改相关条件，该互斥锁用于获得信号的条件变量中。否则，可能在条件变量的测试和 pthread_cond_wait() 阻塞之间修改该变量，这会导致无限期等待。

pthread_cond_broadcast 返回值

pthread_cond_broadcast() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *cv* 指示的地址非法。

销毁条件变量状态

使用 pthread_cond_destroy(3C) 可以销毁与 *cv* 所指向的条件变量相关联的任何状态。对于 Solaris 线程，请参见第 210 页中的 “cond_destroy 语法”。

pthread_cond_destroy 语法

```
int    pthread_cond_destroy(pthread_cond_t *cv);

#include <pthread.h>



pthread_cond_t cv;

int ret;



/* Condition variable is destroyed */

ret = pthread_cond_destroy(&cv);
```

请注意，没有释放用来存储条件变量的空间。

pthread_cond_destroy 返回值

pthread_cond_destroy() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *cv* 指定的值无效。

唤醒丢失问题

如果线程未持有与条件相关联的互斥锁，则调用 pthread_cond_signal() 或 pthread_cond_broadcast() 会产生**唤醒丢失**错误。

满足以下所有条件时，即会出现唤醒丢失问题：

- 一个线程调用 pthread_cond_signal() 或 pthread_cond_broadcast()
- 另一个线程已经测试了该条件，但是尚未调用 pthread_cond_wait()
- 没有正在等待的线程

  信号不起作用，因此将会丢失

仅当修改所测试的条件但未持有与之相关联的互斥锁时，才会出现此问题。只要仅在持有关联的互斥锁同时修改所测试的条件，即可调用 pthread_cond_signal() 和 pthread_cond_broadcast()，而无论这些函数是否持有关联的互斥锁。

生成方和使用者问题

并发编程中收集了许多标准的众所周知的问题，生成方和使用者问题只是其中的一个问题。此问题涉及到一个大小限定的缓冲区和两类线程（生成方和使用者），生成方将项放入缓冲区中，然后使用者从缓冲区中取走项。

生成方必须在缓冲区中有可用空间之后才能向其中放置内容。使用者必须在生成方向缓冲区中写入之后才能从中提取内容。

条件变量表示一个等待某个条件获得信号的线程队列。

示例 4–11 中包含两个此类队列。一个队列 (*less*) 针对生成方，用于等待缓冲区中出现空位置。另一个队列 (*more*) 针对使用者，用于等待从缓冲槽位的空位置中提取其中包含的信息。该示例中还包含一个互斥锁，因为描述该缓冲区的数据结构一次只能由一个线程访问。

示例 4–11 生成方和使用者的条件变量问题

```
typedef struct {

    char buf[BSIZE];

    int occupied;

    int nextin;

    int nextout;

    pthread_mutex_t mutex;
```

示例 4–11 生成方和使用者的条件变量问题 （续）

```
    pthread_cond_t more;

    pthread_cond_t less;

} buffer_t;



buffer_t buffer;
```

如示例 4–12 中所示，生成方线程获取该互斥锁以保护 `buffer` 数据结构，然后，缓冲区确定是否有空间可用于存放所生成的项。如果没有可用空间，生成方线程会调用 `pthread_cond_wait()`。`pthread_cond_wait()` 会导致生成方线程连接正在等待 *less* 条件获得信号的线程队列。*less* 表示缓冲区中的可用空间。

与此同时，在调用 `pthread_cond_wait()` 的过程中，该线程会释放互斥锁的锁定。正在等待的生成方线程依赖于使用者线程在条件为真时发出信号，如示例 4–12 中所示。该条件获得信号时，将会唤醒等待 *less* 的第一个线程。但是，该线程必须再次锁定互斥锁，然后才能从 `pthread_cond_wait()` 返回。

获取互斥锁可确保该线程再次以独占方式访问缓冲区的数据结构。该线程随后必须检查缓冲区中是否确实存在可用空间。如果空间可用，该线程会向下一个可用的空位置中进行写入。

与此同时，使用者线程可能正在等待项出现在缓冲区中。这些线程正在等待条件变量 *more*。刚在缓冲区中存储内容的生成方线程会调用 `pthread_cond_signal()` 以唤醒下一个正在等待的使用者。如果没有正在等待的使用者，此调用将不起作用。

最后，生成方线程会解除锁定互斥锁，从而允许其他线程处理缓冲区的数据结构。

示例 4–12 生成方和使用者问题：生成方

```
void producer(buffer_t *b, char item)

{

    pthread_mutex_lock(&b->mutex);



    while (b->occupied >= BSIZE)

        pthread_cond_wait(&b->less, &b->mutex);
```

示例 4–12 生成方和使用者问题：生成方 （续）

```
    assert(b->occupied < BSIZE);

    b->buf[b->nextin++] = item;

    b->nextin %= BSIZE;
    b->occupied++;

    /* now: either b->occupied < BSIZE and b->nextin is the index
       of the next empty slot in the buffer, or
       b->occupied == BSIZE and b->nextin is the index of the
       next (occupied) slot that will be emptied by a consumer
       (such as b->nextin == b->nextout) */

    pthread_cond_signal(&b->more);

    pthread_mutex_unlock(&b->mutex);
}
```

请注意 `assert()` 语句的用法。除非在编译代码时定义了 `NDEBUG`，否则 `assert()` 在其参数的计算结果为真（非零）时将不执行任何操作。如果参数的计算结果为假（零），则该程序会中止。在多线程程序中，此类断言特别有用。如果断言失败，`assert()` 会立即指出运行时问题。`assert()` 还有另一个作用，即提供有用的注释。

以 `/* now: either b->occupied ...` 开头的注释最好以断言形式表示，但是由于语句过于复杂，无法用布尔值表达式来表示，因此将用英语表示。

断言和注释都是不变量的示例。这些不变量是逻辑语句，在程序正常执行时不应将其声明为假，除非是线程正在修改不变量中提到的一些程序变量时的短暂修改过程中。当然，只要有线程执行语句，断言就应当为真。

使用不变量是一种极为有用的方法。即使没有在程序文本中声明不变量，在分析程序时也应将其视为不变量。

每次线程执行包含注释的代码时，生成方代码中表示为注释的不变量始终为真。如果将此注释移到紧挨 mutex_unlock() 的后面，则注释不一定仍然为真。如果将此注释移到紧跟 assert() 之后的位置，则注释仍然为真。

因此，不变量可用于表示一个始终为真的属性，除非一个生成方或一个使用者正在更改缓冲区的状态。线程在互斥锁的保护下处理缓冲区时，该线程可能会暂时声明不变量为假。但是，一旦线程结束对缓冲区的操作，不变量即会恢复为真。

示例 4–13 给出了使用者的代码。该逻辑流程与生成方的逻辑流程相对称。

示例 4–13 生成方和使用者问题：使用者

```
char consumer(buffer_t *b)

{

    char item;

    pthread_mutex_lock(&b->mutex);

    while(b->occupied <= 0)

        pthread_cond_wait(&b->more, &b->mutex);



    assert(b->occupied > 0);



    item = b->buf[b->nextout++];

    b->nextout %= BSIZE;

    b->occupied--;



    /* now: either b->occupied > 0 and b->nextout is the index

       of the next occupied slot in the buffer, or

       b->occupied == 0 and b->nextout is the index of the next

       (empty) slot that will be filled by a producer (such as
```

示例 4–13 生成方和使用者问题：使用者 （续）

```
        b->nextout == b->nextin) */


    pthread_cond_signal(&b->less);

    pthread_mutex_unlock(&b->mutex);


    return(item);

}
```

使用信号进行同步

信号是 E. W. Dijkstra 在二十世纪六十年代末设计的一种编程架构。Dijkstra 的模型与铁路操作有关：假设某段铁路是单线的，因此一次只允许一列火车通过。

信号将用于同步通过该轨道的火车。火车在进入单一轨道之前必须等待信号灯变为允许通行的状态。火车进入轨道后，会改变信号状态，防止其他火车进入该轨道。火车离开这段轨道时，必须再次更改信号的状态，以便允许其他火车进入轨道。

在计算机版本中，信号以简单整数来表示。线程等待获得许可以便继续运行，然后发出信号，表示该线程已经通过针对信号执行 `P` 操作来继续运行。

线程必须等到信号的值为正，然后才能通过将信号值减 1 来更改该值。完成此操作后，线程会执行 `V` 操作，即通过将信号值加 1 来更改该值。这些操作必须以原子方式执行，不能再将其划分成子操作，即，在这些子操作之间不能对信号执行其他操作。在 `P` 操作中，信号值在减小之前必须为正，从而确保生成的信号值不为负，并且比该值减小之前小 1。

在 `P` 和 `V` 操作中，必须在没有干扰的情况下进行运算。如果针对同一信号同时执行两个 `V` 操作，则实际结果是信号的新值比原来大 2。

对于大多数人来说，如同记住 Dijkstra 是荷兰人一样，记住 `P` 和 `V` 本身的含义并不重要。但是，从真正学术的角度来说，`P` 代表 prolagen，这是由 proberen te verlagen 演变而来的杜撰词，其意思是**尝试减小**。`V` 代表 verhogen，其意思是**增加**。Dijkstra 的技术说明 EWD 74 中介绍了这些含义。

`sem_wait`(3RT) 和 `sem_post`(3RT) 分别与 Dijkstra 的 `P` 和 `V` 操作相对应。`sem_trywait`(3RT) 是 `P` 操作的一种条件形式。如果调用线程不等待就不能减小信号的值，则该调用会立即返回一个非零值。

有两种基本信号：二进制信号和计数信号量。二进制信号的值只能是 0 或 1，计数信号量可以是任意非负值。二进制信号在逻辑上相当于一个互斥锁。

不过，尽管不会强制，但互斥锁应当仅由持有该锁的线程来解除锁定。因为不存在“持有信号的线程”这一概念，所以，任何线程都可以执行 V 或 sem_post(3RT) 操作。

计数信号量与互斥锁一起使用时的功能几乎与条件变量一样强大。在许多情况下，使用计数信号量实现的代码比使用条件变量实现的代码更为简单，如示例 4–14、示例 4–15 和示例 4–16 中所示。

但是，将互斥锁用于条件变量时，会存在一个隐含的括号。该括号可以清楚表明程序受保护的部分。对于信号则不必如此，可以使用并发编程当中的 *go to* 对其进行调用。信号的功能强大，但是容易以非结构化的不确定方式使用。

命名信号和未命名信号

POSIX 信号可以是未命名的，也可以是命名的。未命名信号在进程内存中分配，并会进行初始化。未命名信号可能可供多个进程使用，具体取决于信号的分配和初始化的方式。未命名信号可以是通过 fork() 继承的专用信号，也可以通过用来分配和映射这些信号的常规文件的访问保护功能对其进行保护。

命名信号类似于进程共享的信号，区别在于命名信号是使用路径名而非 *pshared* 值引用的。命名信号可以由多个进程共享。命名信号具有属主用户 ID、组 ID 和保护模式。

对于 open、retrieve、close 和 remove 命名信号，可以使用以下函数：sem_open、sem_getvalue、sem_close 和 sem_unlink。通过使用 sem_open，可以创建一个命名信号，其名称是在文件系统的名称空间中定义的。

有关命名信号的更多信息，请参见 sem_open、sem_getvalue、sem_close 和 sem_unlink 手册页。

计数信号量概述

从概念上来说，信号量是一个非负整数计数。信号量通常用来协调对资源的访问，其中信号计数会初始化为可用资源的数目。然后，线程在资源增加时会增加计数，在删除资源时会减小计数，这些操作都以原子方式执行。

如果信号计数变为零，则表明已无可用资源。计数为零时，尝试减小信号的线程会被阻塞，直到计数大于零为止。

表 4–7 信号例程

操作	相关函数说明
初始化信号	第 123 页中的 “sem_init 语法”
增加信号	第 125 页中的 “sem_post 语法”

表 4–7 信号例程 （续）

操作	相关函数说明
基于信号计数阻塞	第 126 页中的 “sem_wait 语法”
减小信号计数	第 126 页中的 “sem_trywait 语法”
销毁信号状态	第 127 页中的 “sem_destroy 语法”

由于信号无需由同一个线程来获取和释放，因此信号可用于异步事件通知，如用于信号处理程序中。同时，由于信号包含状态，因此可以异步方式使用，而不用象条件变量那样要求获取互斥锁。但是，信号的效率不如互斥锁高。

缺省情况下，如果有多个线程正在等待信号，则解除阻塞的顺序是不确定的。

信号在使用前必须先初始化，但是信号没有属性。

初始化信号

使用 sem_init(3RT) 可以将 *sem* 所指示的未命名信号变量初始化为 *value*。

sem_init 语法

```
int    sem_init(sem_t *sem, int pshared, unsigned int value);

#include <semaphore.h>


sem_t sem;

int pshared;

int ret;

int value;


/* initialize a private semaphore */

pshared = 0;

value = 1;

ret = sem_init(&sem, pshared, value);
```

如果 *pshared* 的值为零，则不能在进程之间共享信号。如果 *pshared* 的值不为零，则可以在进程之间共享信号。对于 Solaris 线程，请参见第 214 页中的 “sema_init 语法” 。

多个线程决不能初始化同一个信号。

不得对其他线程正在使用的信号重新初始化。

初始化进程内信号

pshared 为 0 时，信号只能由该进程内的所有线程使用。

```
#include <semaphore.h>


sem_t sem;

int ret;

int count = 4;


/* to be used within this process only */

ret = sem_init(&sem, 0, count);
```

初始化进程间信号

pshared 不为零时，信号可以由其他进程共享。

```
#include <semaphore.h>


sem_t sem;

int ret;

int count = 4;


/* to be shared among processes */

ret = sem_init(&sem, 1, count);
```

sem_init 返回值

`sem_init()` 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EINVAL
描述: 参数值超过了 SEM_VALUE_MAX。

ENOSPC
描述: 初始化信号所需的资源已经用完。到达信号的 SEM_NSEMS_MAX 限制。

ENOSYS
描述: 系统不支持 sem_init() 函数。

EPERM
描述: 进程缺少初始化信号所需的适当权限。

增加信号

使用 sem_post(3RT) 可以原子方式增加 *sem* 所指示的信号。

sem_post 语法

```
int    sem_post(sem_t *sem);

#include <semaphore.h>



sem_t sem;

int ret;



ret = sem_post(&sem); /* semaphore is posted */
```

如果所有线程均基于信号阻塞，则会对其中一个线程解除阻塞。对于 Solaris 线程，请参见第 215 页中的 “sema_post 语法” 。

sem_post 返回值

sem_post() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *sem* 所指示的地址非法。

基于信号计数进行阻塞

使用 sem_wait(3RT) 可以阻塞调用线程，直到 *sem* 所指示的信号计数大于零为止，之后以原子方式减小计数。

sem_wait 语法

```
int    sem_wait(sem_t *sem);

#include <semaphore.h>



sem_t sem;

int ret;



ret = sem_wait(&sem); /* wait for semaphore */
```

sem_wait 返回值

sem_wait() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EINVAL
描述: *sem* 所指示的地址非法。

EINTR
描述: 此函数已被信号中断。

减小信号计数

使用 sem_trywait(3RT) 可以在计数大于零时，尝试以原子方式减小 *sem* 所指示的信号计数。

sem_trywait 语法

```
int    sem_trywait(sem_t *sem);

#include <semaphore.h>



sem_t sem;
```

```
int ret;


ret = sem_trywait(&sem); /* try to wait for semaphore*/
```

此函数是 sem_wait() 的非阻塞版本。sem_trywait() 在失败时会立即返回。

sem_trywait 返回值

sem_trywait() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EINVAL
描述: *sem* 所指示的地址非法。

EINTR
描述: 此函数已被信号中断。

EAGAIN
描述: 信号已为锁定状态，因此该信号不能通过 sem_trywait() 操作立即锁定。

销毁信号状态

使用 sem_destroy(3RT) 可以销毁与 *sem* 所指示的未命名信号相关联的任何状态。

sem_destroy 语法

```
int    sem_destroy(sem_t *sem);

#include <semaphore.h>


sem_t sem;

int ret;


ret = sem_destroy(&sem); /* the semaphore is destroyed */
```

不会释放用来存储信号的空间。对于 Solaris 线程，请参见第 217 页中的 “sema_destroy(3C) 语法” 。

sem_destroy 返回值

sem_destroy() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EINVAL
描述: *sem* 所指示的地址非法。

使用信号时的生成方和使用者问题

示例 4–14 中的数据结构与示例 4–11 中所示的用于条件变量示例的结构类似。两个信号分别表示空缓冲区和满缓冲区的数目，通过这些信号可确保生成方等待缓冲区变空，使用者等待缓冲区变满为止。

示例 4–14 使用信号时的生成方和使用者问题

```
typedef struct {

    char buf[BSIZE];

    sem_t occupied;

    sem_t empty;

    int nextin;

    int nextout;

    sem_t pmut;

    sem_t cmut;

} buffer_t;



buffer_t buffer;



sem_init(&buffer.occupied, 0, 0);

sem_init(&buffer.empty,0, BSIZE);

sem_init(&buffer.pmut, 0, 1);

sem_init(&buffer.cmut, 0, 1);
```

示例 4–14 使用信号时的生成方和使用者问题 （续）

```
buffer.nextin = buffer.nextout = 0;
```

另一对二进制信号与互斥锁作用相同。在多个生成方使用多个空缓冲槽位，以及多个使用者使用多个满缓冲槽位的情况下，信号可用来控制对缓冲区的访问。在这种情况下，使用互斥锁可能会更好，但这里主要是为了演示信号的用法。

示例 4–15 生成方和使用者问题：生成方

```
void producer(buffer_t *b, char item) {

    sem_wait(&b->empty);

    sem_wait(&b->pmut);



    b->buf[b->nextin] = item;

    b->nextin++;

    b->nextin %= BSIZE;



    sem_post(&b->pmut);

    sem_post(&b->occupied);

}
```

示例 4–16 生成方和使用者问题：使用者

```
char consumer(buffer_t *b) {

    char item;



    sem_wait(&b->occupied);



    sem_wait(&b->cmut);
```

示例 4–16 生成方和使用者问题：使用者 （续）

```
    item = b->buf[b->nextout];

    b->nextout++;

    b->nextout %= BSIZE;


    sem_post(&b->cmut);


    sem_post(&b->empty);


    return(item);

}
```

读写锁属性

通过读写锁，可以对受保护的共享资源进行并发读取和独占写入。读写锁是可以在**读取**或**写入**模式下锁定的单一实体。要修改资源，线程必须首先获取互斥写锁。必须释放所有读锁之后，才允许使用互斥写锁。

有关 Solaris 线程所实现的读写锁，请参见第 186 页中的 “相似的同步函数－读写锁” 。

对数据库的访问可以使用读写锁进行同步。读写锁支持并发读取数据库记录，因为读操作不会更改记录的信息。要更新数据库时，写操作必须获取互斥写锁。

要更改缺省的读写锁属性，可以声明和初始化属性对象。通常，可以在应用程序开头的某个位置设置读写锁属性，设置在应用程序的起始位置可使属性更易于查找和修改。下表列出了本节中讨论的用来处理读写锁属性的函数。

表 4–8 读写锁属性例程

操作	相关函数说明
初始化读写锁属性	第 131 页中的 “pthread_rwlockattr_init 语法”
销毁读写锁属性	第 131 页中的 “pthread_rwlockattr_destroy 语法”

表 4–8 读写锁属性例程 （续）

操作	相关函数说明
设置读写锁属性	第 132 页中的 “pthread_rwlockattr_setpshared 语法”
获取读写锁属性	第 133 页中的 “pthread_rwlockattr_getpshared 语法”

初始化读写锁属性

pthread_rwlockattr_init(3C) 使用实现中定义的所有属性的缺省值来初始化读写锁属性对象 *attr*。

pthread_rwlockattr_init 语法

```
#include <pthread.h>


int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

如果调用 pthread_rwlockattr_init 来指定已初始化的读写锁属性对象，则结果是不确定的。读写锁属性对象初始化一个或多个读写锁之后，影响该对象的任何函数（包括销毁）不会影响先前已初始化的读写锁。

pthread_rwlockattr_init 返回值

如果成功，pthread_rwlockattr_init() 会返回零。否则，将返回用于指明错误的错误号。

ENOMEM
描述: 内存不足，无法初始化读写锁属性对象。

销毁读写锁属性

pthread_rwlockattr_destroy(3C) 可用来销毁读写锁属性对象。

pthread_rwlockattr_destroy 语法

```
#include <pthread.h>


int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

在再次调用 pthread_rwlockattr_init() 重新初始化该对象之前，使用该对象所产生的影响是不确定的。实现可以导致 pthread_rwlockattr_destroy() 将 *attr* 所引用的对象设置为无效值。

pthread_rwlockattr_destroy 返回值

如果成功，pthread_rwlockattr_destroy() 会返回零。否则，将返回用于指明错误的错误号。

EINVAL
描述: *attr* 指定的值无效。

设置读写锁属性

pthread_rwlockattr_setpshared(3C) 可用来设置由进程共享的读写锁属性。

pthread_rwlockattr_setpshared 语法

```
#include <pthread.h>



int pthread_rwlockattr_setpshared(pthread_rwlockattr_t  *attr,

                                          int  pshared);
```

读写锁属性可以为以下值之一：

PTHREAD_PROCESS_SHARED
描述: 允许可访问用于分配读写锁的内存的任何线程对读写锁进行处理。即使该锁是在由多个进程共享的内存中分配的，也允许对其进行处理。

PTHREAD_PROCESS_PRIVATE
描述: 读写锁只能由某些线程处理，这些线程与初始化该锁的线程在同一进程中创建。如果不同进程的线程尝试对此类读写锁进行处理，则其行为是不确定的。由进程共享的属性的缺省值为 PTHREAD_PROCESS_PRIVATE。

pthread_rwlockattr_setpshared 返回值

如果成功，pthread_rwlockattr_setpshared() 会返回零。否则，将返回用于指明错误的错误号。

EINVAL
描述: *attr* 或 *pshared* 指定的值无效。

获取读写锁属性

pthread_rwlockattr_getpshared(3C) 可用来获取由进程共享的读写锁属性。

pthread_rwlockattr_getpshared 语法

```
#include <pthread.h>


int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t  *attr,

                                                  int *pshared);
```

pthread_rwlockattr_getpshared() 从 *attr* 引用的已初始化属性对象中获取由进程共享的属性的值。

pthread_rwlockattr_getpshared 返回值

如果成功，pthread_rwlockattr_getpshared() 会返回零。否则，将返回用于指明错误的错误号。

EINVAL
描述: *attr* 或 *pshared* 指定的值无效。

使用读写锁

配置读写锁的属性之后，即可初始化读写锁。以下函数用于初始化或销毁读写锁、锁定或解除锁定读写锁或尝试锁定读写锁。下表列出了本节中讨论的用来处理读写锁的函数。

表 4–9 处理读写锁的例程

操作	相关函数说明
初始化读写锁	第 134 页中的 “pthread_rwlock_init 语法”
读取读写锁中的锁	第 134 页中的 “pthread_rwlock_rdlock 语法”
读取非阻塞读写锁中的锁	第 135 页中的 “pthread_rwlock_tryrdlock 语法”
写入读写锁中的锁	第 136 页中的 “pthread_rwlock_wrlock 语法”
写入非阻塞读写锁中的锁	第 136 页中的 “pthread_rwlock_trywrlock 语法”
解除锁定读写锁	第 137 页中的 “pthread_rwlock_unlock 语法”
销毁读写锁	第 138 页中的 “pthread_rwlock_destroy 语法”

初始化读写锁

使用 pthread_rwlock_init(3C) 可以通过 *attr* 所引用的属性初始化 *rwlock* 所引用的读写锁。

pthread_rwlock_init 语法

```
#include <pthread.h>



int pthread_rwlock_init(pthread_rwlock_t *rwlock,

                                const pthread_rwlockattr_t *attr);



pthread_rwlock_t  rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

如果 *attr* 为 NULL，则使用缺省的读写锁属性，其作用与传递缺省读写锁属性对象的地址相同。初始化读写锁之后，该锁可以使用任意次数，而无需重新初始化。成功初始化之后，读写锁的状态会变为已初始化和未锁定。如果调用 pthread_rwlock_init() 来指定已初始化的读写锁，则结果是不确定的。如果读写锁在使用之前未初始化，则结果是不确定的。对于 Solaris 线程，请参见第 186 页中的 “rwlock_init 语法”。

如果缺省的读写锁属性适用，则 PTHREAD_RWLOCK_INITIALIZER 宏可初始化以静态方式分配的读写锁，其作用与通过调用 pthread_rwlock_init() 并将参数 *attr* 指定为 NULL 进行动态初始化等效，区别在于不会执行错误检查。

pthread_rwlock_init 返回值

如果成功，pthread_rwlock_init() 会返回零。否则，将返回用于指明错误的错误号。

如果 pthread_rwlock_init() 失败，将不会初始化 *rwlock*，并且 *rwlock* 的内容是不确定的。

EINVAL
描述: *attr* 或 *rwlock* 指定的值无效。

获取读写锁中的读锁

pthread_rwlock_rdlock(3C) 可用来向 *rwlock* 所引用的读写锁应用读锁。

pthread_rwlock_rdlock 语法

```
#include <pthread.h>



int  pthread_rwlock_rdlock(pthread_rwlock_t *rwlock );
```

如果写入器未持有读锁，并且没有任何写入器基于该锁阻塞，则调用线程会获取读锁。如果写入器未持有读锁，但有多个写入器正在等待该锁时，调用线程是否能获取该锁是不确

定的。如果某个写入器持有读锁，则调用线程无法获取该锁。如果调用线程未获取读锁，则它将阻塞。调用线程必须获取该锁之后，才能从 pthread_rwlock_rdlock() 返回。如果在进行调用时，调用线程持有 *rwlock* 中的写锁，则结果是不确定的。

为避免写入器资源匮乏，允许在多个实现中使写入器的优先级高于读取器。例如，Solaris 线程实现中写入器的优先级高于读取器。 请参见第 188 页中的 “rw_rdlock 语法” 。

一个线程可以在 *rwlock* 中持有多个并发的读锁，该线程可以成功调用 pthread_rwlock_rdlock() *n* 次。该线程必须调用 pthread_rwlock_unlock() *n* 次才能执行匹配的解除锁定操作。

如果针对未初始化的读写锁调用 pthread_rwlock_rdlock()，则结果是不确定的。

线程信号处理程序可以处理传送给等待读写锁的线程的信号。从信号处理程序返回后，线程将继续等待读写锁以执行读取，就好像线程未中断一样。

pthread_rwlock_rdlock 返回值

如果成功，pthread_rwlock_rdlock() 会返回零。否则，将返回用于指明错误的错误号。

EINVAL
描述: *attr* 或 *rwlock* 指定的值无效。

读取非阻塞读写锁中的锁

pthread_rwlock_tryrdlock(3C) 应用读锁的方式与 pthread_rwlock_rdlock() 类似，区别在于如果任何线程持有 *rwlock* 中的写锁或者写入器基于 *rwlock* 阻塞，则 pthread_rwlock_tryrdlock() 函数会失败。对于 Solaris 线程，请参见第 188 页中的 “rw_tryrdlock 语法” 。

pthread_rwlock_tryrdlock 语法

```
#include <pthread.h>



int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

pthread_rwlock_tryrdlock 返回值

如果获取了用于在 *rwlock* 所引用的读写锁对象中执行读取的锁，则 pthread_rwlock_tryrdlock() 将返回零。如果没有获取该锁，则返回用于指明错误的错误号。

EBUSY
描述: 无法获取读写锁以执行读取，因为写入器持有该锁或者基于该锁已阻塞。

写入读写锁中的锁

pthread_rwlock_wrlock(3C) 可用来向 *rwlock* 所引用的读写锁应用写锁。

pthread_rwlock_wrlock 语法

```
#include <pthread.h>


int  pthread_rwlock_wrlock(pthread_rwlock_t *rwlock );
```

如果没有其他读取器线程或写入器线程持有读写锁 *rwlock*，则调用线程将获取写锁。否则，调用线程将阻塞。调用线程必须获取该锁之后，才能从 pthread_rwlock_wrlock() 调用返回。如果在进行调用时，调用线程持有读写锁（读锁或写锁），则结果是不确定的。

为避免写入器资源匮乏，允许在多个实现中使写入器的优先级高于读取器。（例如，Solaris 线程实现允许写入器的优先级高于读取器。请参见第 189 页中的 “rw_wrlock 语法” 。）

如果针对未初始化的读写锁调用 pthread_rwlock_wrlock()，则结果是不确定的。

线程信号处理程序可以处理传送给等待读写锁以执行写入的线程的信号。从信号处理程序返回后，线程将继续等待读写锁以执行写入，就好像线程未中断一样。

pthread_rwlock_wrlock 返回值

如果获取了用于在 *rwlock* 所引用的读写锁对象中执行写入的锁，则 pthread_rwlock_rwlock() 将返回零。如果没有获取该锁，则返回用于指明错误的错误号。

写入非阻塞读写锁中的锁

pthread_rwlock_trywrlock(3C) 应用写锁的方式与 pthread_rwlock_wrlock() 类似，区别在于如果任何线程当前持有用于读取和写入的 *rwlock*，则 pthread_rwlock_trywrlock() 函数会失败。对于 Solaris 线程，请参见 第 190 页中的 “rw_trywrlock 语法” 。

pthread_rwlock_trywrlock 语法

```
#include <pthread.h>


int pthread_rwlock_trywrlock(pthread_rwlock_t  *rwlock);
```

如果针对未初始化的读写锁调用 pthread_rwlock_trywrlock()，则结果是不确定的。

线程信号处理程序可以处理传送给等待读写锁以执行写入的线程的信号。从信号处理程序返回后，线程将继续等待读写锁以执行写入，就好像线程未中断一样。

pthread_rwlock_trywrlock 返回值

如果获取了用于在 *rwlock* 引用的读写锁对象中执行写入的锁，则 `pthread_rwlock_trywrlock()` 将返回零。否则，将返回用于指明错误的错误号。

`EBUSY`
描述: 无法为写入获取读写锁，因为已为读取或写入锁定该读写锁。

解除锁定读写锁

`pthread_rwlock_unlock`(3C) 可用来释放在 *rwlock* 引用的读写锁对象中持有的锁。

pthread_rwlock_unlock 语法

```
#include <pthread.h>



int pthread_rwlock_unlock (pthread_rwlock_t  *rwlock);
```

如果调用线程未持有读写锁 *rwlock*，则结果是不确定的。对于 Solaris 线程，请参见第 190 页中的 “rw_unlock 语法”。

如果通过调用 `pthread_rwlock_unlock()` 来释放读写锁对象中的读锁，并且其他读锁当前由该锁对象持有，则该对象会保持读取锁定状态。如果 `pthread_rwlock_unlock()` 释放了调用线程在该读写锁对象中的最后一个读锁，则调用线程不再是该对象的属主。如果 `pthread_rwlock_unlock()` 释放了该读写锁对象的最后一个读锁，则该读写锁对象将处于无属主、解除锁定状态。

如果通过调用 `pthread_rwlock_unlock()` 释放了该读写锁对象的最后一个写锁，则该读写锁对象将处于无属主、解除锁定状态。

如果 `pthread_rwlock_unlock()` 解除锁定该读写锁对象，并且多个线程正在等待获取该对象以执行写入，则通过调度策略可确定获取该对象以执行写入的线程。如果多个线程正在等待获取读写锁对象以执行读取，则通过调度策略可确定等待线程获取该对象以执行写入的顺序。如果多个线程基于 *rwlock* 中的读锁和写锁阻塞，则无法确定读取器和写入器谁先获得该锁。

如果针对未初始化的读写锁调用 `pthread_rwlock_unlock()`，则结果是不确定的。

pthread_rwlock_unlock 返回值

如果成功，`pthread_rwlock_unlock()` 会返回零。否则，将返回用于指明错误的错误号。

销毁读写锁

`pthread_rwlock_destroy`(3C) 可用来销毁 *rwlock* 引用的读写锁对象并释放该锁使用的任何资源。

pthread_rwlock_destroy 语法

```
#include <pthread.h>


int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);


pthread_rwlock_t  rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

在再次调用 pthread_rwlock_init() 重新初始化该锁之前，使用该锁所产生的影响是不确定的。实现可能会导致 pthread_rwlock_destroy() 将 *rwlock* 所引用的对象设置为无效值。如果在任意线程持有 *rwlock* 时调用 pthread_rwlock_destroy()，则结果是不确定的。尝试销毁未初始化的读写锁会产生不确定的行为。已销毁的读写锁对象可以使用 pthread_rwlock_init() 来重新初始化。销毁读写锁对象之后，如果以其他方式引用该对象，则结果是不确定的。对于 Solaris 线程，请参见第 191 页中的 “rwlock_destroy 语法”。

pthread_rwlock_destroy 返回值

如果成功，pthread_rwlock_destroy() 会返回零。否则，将返回用于指明错误的错误号。

EINVAL
描述: *attr* 或 *rwlock* 指定的值无效。

跨进程边界同步

每个同步元语都可以跨进程边界使用。通过确保同步变量位于共享内存段中，并调用适当的 init() 例程，可设置元语。元语必须已经初始化，并且其共享属性设置为在进程间使用。

生成方和使用者问题示例

示例 4–17 说明了位于不同进程中的生成方和使用者的问题。主例程将与其子进程共享的全零内存段映射到其地址空间。

创建子进程是为了运行使用者，父进程则运行生成方。

本示例还说明了生成方和使用者的驱动程序。producer_driver() 可从 stdin 读取字符并调用 producer()。consumer_driver() 通过调用 consumer() 来获取字符并将这些字符写入 stdout 中。

示例 4–17 中的数据结构与示例 4–4 中所示用于条件变量示例的结构类似。两个信号分别空缓冲区和满缓冲区的数量，通过这些信号可确保生成方等待缓冲区变空，使用者等待缓冲区变满为止。

示例 4–17 跨进程边界同步

```
main() {

    int zfd;

    buffer_t *buffer;

    pthread_mutexattr_t mattr;

    pthread_condattr_t cvattr_less, cvattr_more;


    zfd = open("/dev/zero", O_RDWR);

    buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),

        PROT_READ|PROT_WRITE, MAP_SHARED, zfd, 0);

    buffer->occupied = buffer->nextin = buffer->nextout = 0;


    pthread_mutex_attr_init(&mattr);

    pthread_mutexattr_setpshared(&mattr,

        PTHREAD_PROCESS_SHARED);


    pthread_mutex_init(&buffer->lock, &mattr);

    pthread_condattr_init(&cvattr_less);

    pthread_condattr_setpshared(&cvattr_less, PTHREAD_PROCESS_SHARED);

    pthread_cond_init(&buffer->less, &cvattr_less);

    pthread_condattr_init(&cvattr_more);

    pthread_condattr_setpshared(&cvattr_more,

        PTHREAD_PROCESS_SHARED);

    pthread_cond_init(&buffer->more, &cvattr_more);
```

示例 4–17 跨进程边界同步 （续）

```
    if (fork() == 0)
        consumer_driver(buffer);
    else
        producer_driver(buffer);
}

void producer_driver(buffer_t *b) {
    int item;

    while (1) {
        item = getchar();
        if (item == EOF) {
            producer(b, '\0');
            break;
        } else
            producer(b, (char)item);
    }
}

void consumer_driver(buffer_t *b) {
    char item;
```

示例 4–17 跨进程边界同步 （续）

```
    while (1) {

        if ((item = consumer(b)) == '\0')

            break;

        putchar(item);

    }

}
```

比较元语

线程中最基本的同步元语是互斥锁。因此，在内存使用和执行时间这两个方面，互斥锁都是最高效的机制。互斥锁的基本用途是按顺序访问资源。

线程中第二高效的元语是条件变量。条件变量的基本用途是基于状态的变化进行阻塞。条件变量可提供线程等待功能。请注意，线程在基于条件变量阻塞之前必须首先获取互斥锁，在从 `pthread_cond_wait()` 返回之后必须解除锁定互斥锁。线程还必须在状态发生改变期间持有互斥锁，然后才能对 `pthread_cond_signal()` 进行相应的调用。

信号比条件变量占用更多内存。由于信号变量基于状态而非控制来工作，因此在某些情况下更易于使用。与锁不同，信号没有属主。任何线程都可以增加已阻塞的信号。

通过读写锁，可以对受保护的资源进行并发读取和独占写入。读写锁是可以在**读取**或**写入**模式下锁定的单一实体。要修改资源，线程必须首先获取互斥写锁。必须释放所有读锁之后，才允许使用互斥写锁。

◆ ◆ ◆ 第 5 章

使用 Solaris 软件编程

本章介绍多线程与 Solaris 软件的交互方式以及软件经过更改后支持多线程的方式。

进程创建中的 fork 问题

Solaris 9 产品和更早 Solaris 发行版中处理 `fork()` 的缺省方式与在 POSIX 线程中处理 `fork()` 的方式稍有不同。 对于 Solaris 9 之后的 Solaris 发行版，在所有情况下，`fork()` 都会按照为 POSIX 线程指定的方式工作。

表 5–1 对在 Solaris 线程和 pthread 中处理 `fork()` 的相似与不同之处进行了比较。当可比较的接口在 POSIX 线程或 Solaris 线程中不可用时，‘—’ 字符将出现在表列中。

表 5–1 比较 POSIX 与 Solaris `fork()` 的处理

	Solaris 接口	POSIX 线程接口
Fork-one 模型	`fork1(2)`	`fork(2)`
Fork-all 模型	`forkall(2)`	`forkall(2)`
fork 安全性	—	`pthread_atfork(3C)`

Fork-One 模型

如表 5–1 所示，pthread `fork(2)` 函数的行为与 Solaris `fork1(2)` 函数的行为相同。pthread `fork(2)` 函数和 Solaris `fork1(2)` 函数都将创建新的进程，并将完整的地址空间复制到子进程中。但是，这两个函数都只将调用线程复制到子进程中。

当子进程直接调用 `exec()` 时，将调用线程复制到子进程中非常有用，大多数情况下，此操作发生在对 `fork()` 的调用之后。在这种情况下，子进程不需要复制 `fork()` 以外的任何线程。

在子进程中，调用 `fork()` 之后和调用 `exec()` 之前，请不要调用任何库函数。某个库函数可能会使用父进程在调用 `fork()` 时所持有的锁。调用某个 `exec()` 处理程序之前，子进程可能仅执行异步信号安全操作。

Fork-One 安全问题和解决方案

除了通常关注的问题（如锁定共享数据）以外，当只有 `fork()` 线程处于运行状态时，还应根据 fork 子进程的操作来处理库。问题在于子进程中的唯一线程可能会尝试获取由未复制到子进程中的线程持有的锁定。

大多数程序不可能遇到此问题。从 `fork()` 返回后，大多数程序都会调用子进程中的 `exec()`。但是，如果程序在调用 `exec()` 之前必须在子进程中执行操作，或永远不会调用 `exec()`，则子进程**可能**会遇到死锁。每个库编写者都应提供安全的解决方案，尽管提供一个非 fork 安全的库不是一个很大的问题。

例如，假设当 T2 fork 新进程时，T1 在进行打印，且对 `printf()` 持有锁定。在子进程中，如果唯一的线程 (T2) 调用 `printf()`，则 T2 将快速死锁。

POSIX `fork()` 或 Solaris `fork1()` 函数仅复制用于调用 `fork()` 或 `fork1()` 的线程。如果调用 Solaris `forkall()` 来复制所有线程，则此问题不是要关注的问题。

但是，`forkall()` 可能会导致其他问题，使用时应小心。例如，如果一个线程调用 `forkall()`，则将在子进程中复制对文件执行 I/O 的父线程。线程的两个副本都将继续对同一个文件执行 I/O，一个副本在父进程中，一个副本在子进程中，这将导致出现异常或文件损坏。

要防止在调用 `fork1()` 时出现死锁，请确保在执行 fork 时任何锁定都未被持有。防止死锁的最有效的方式就是让 fork 线程获取可能由子进程使用的所有锁定。由于无法获取对 `printf()` 的所有锁定（由于 `printf()` 由 `libc` 所有），因此必须确保在使用 `fork()` 时没有使用 `printf()`。

要管理库中的锁定，应执行以下操作：

- 确定库使用的所有锁定。
- 确定库所使用锁定的锁定顺序。如果没有使用严格的锁定顺序，则必须谨慎管理锁定获取。
- 安排在 fork 时获取所有锁定。

在以下示例中，库使用的锁定列表为 {`L1,...Ln`}。这些锁定的锁定顺序也为 `L1...Ln`。

```
mutex_lock(L1);

mutex_lock(L2);

fork1(...);

mutex_unlock(L1);

mutex_unlock(L2);
```

使用 Solaris 线程或 POSIX 线程时，应该在库的 `.init()` 部分中添加对 `pthread_atfork(f1, f2, f3)` 的调用。`f1()`、`f2()`、`f3()` 定义如下：

```
f1() /* This is executed just before the process forks. */

{

 mutex_lock(L1); |

 mutex_lock(...); | -- ordered in lock order

 mutex_lock(Ln); |

 } V



f2() /* This is executed in the child after the process forks. */

 {

 mutex_unlock(L1);

 mutex_unlock(...);

 mutex_unlock(Ln);

 }



f3() /* This is executed in the parent after the process forks. */

 {

 mutex_unlock(L1);

 mutex_unlock(...);
```

```
 mutex_unlock(Ln);

 }
```

虚拟 fork －vfork

标准 vfork(2) 函数在多线程程序中并不安全。vfork(2)（与 fork1(2) ¤@样）仅复制子进程中的调用线程。就像在非线程实现中一样，vfork() 不会复制子进程的地址空间。

请注意，子进程中的线程在调用 exec(2) 之前不会更改内存。vfork() 为子进程提供父地址空间。子进程调用 exec() 或退出之后，父进程将取回其地址空间。子进程不得更改父进程的状态。

例如，如果在对 vfork() 的调用与对 exec() 的调用之间创建新的线程，则会出现灾难性问题。

解决方案： pthread_atfork

使用 Fork-One 模型时，请使用 pthread_atfork() 来防止死锁。

```
#include <pthread.h>



int pthread_atfork(void (*prepare) (void), void (*parent) (void),

    void (*child) (void) );
```

pthread_atfork() 函数声明了在调用 fork() 的线程的上下文中的 fork() 前后调用的 fork() 处理程序。

- 在 fork() 启动前调用 *prepare* 处理程序。
- 在父进程中返回 fork() 后调用 *parent* 处理程序。
- 在子进程中返回 fork() 后调用 *child* 处理程序。

可以将任何处理程序参数都设置为 NULL。对 pthread_atfork() 进行连续调用的顺序非常重要。

例如，*prepare* 处理程序可能会获取所有需要的互斥。然后，*parent* 和 *child* 处理程序可能会释放互斥。获取所有需要的互斥的 *prepare* 处理程序可确保在对进程执行 fork **之前**，所有相关的锁定都由调用 fork 函数的线程持有。此技术可防止子进程中出现死锁。

pthread_atfork 返回值

调用成功完成后，pthread_atfork() 将返回零。其他任何返回值都表示出现了错误。如果检测到以下情况，pthread_atfork() 将失败并返回对应的值。

ENOMEM
描述: 用于记录 fork 处理程序地址的表空间不足。

Fork-all 模型

Solaris forkall(2) 函数可以复制地址空间以及子进程中的所有线程。地址空间复制非常有用，例如，在子进程永远不调用 exec(2) 但会使用其父地址空间的副本时。

当进程中的某个线程调用 Solaris forkall(2) 时，在可中断的系统调用中阻塞的线程将返回 EINTR。

请注意，不要创建同时由父进程和子进程持有的锁定。通过调用包含 MAP_SHARED 标志的 mmap() 在共享内存中分配锁定时，会出现父进程和子进程同时持有锁定的情况。如果使用 Fork-One 模型，则不会出现此问题。

选择正确的 Fork

从 Solaris 10 发行版开始，对 fork() 的调用与对 fork1() 的调用相同。具体来说，在子进程中仅复制调用线程。此行为与 POSIX fork() 的行为相同。

在以前的 Solaris 软件发行版中，fork() 的行为取决于应用程序是否与 POSIX 线程库相链接。如果与 -lthread（Solaris 线程）链接，但没有与 -lpthread（POSIX 线程）链接，则 fork() 与 forkall() 相同。如果与 -lpthread 链接，无论 fork() 是否还与 -lthread 链接，fork() 都与 fork1() 相同。

从 Solaris 10 发行版开始，多线程不需要 -lthread 和 -lpthread。标准 C 库为两组应用程序程序接口提供所有的线程支持。需要**复制所有** fork 语义的应用程序必须调用 forkall()。

调用任何 fork() 函数后，使用全局状态时要非常小心。

例如，当一个线程连续读取文件，而进程中的另一个线程成功 fork 时，每个进程都包含读取该文件的线程。由于在调用 fork() 后会共享文件描述符的查找指针，因此在子线程获取数据的同时，父进程会获取不同的数据。由于父线程和子线程将获取不同的数据，因此会给连续读取访问带来间隙。

进程创建：exec 和 exit 问题

exec(2) 和 exit(2) 系统调用的工作方式与这些函数在单线程进程中的工作方式相同，但以下情况例外。在多线程应用程序中，这些函数将销毁地址空间中的所有线程。销毁所有执行资源和所有活动线程之前，这两个调用将阻塞。

exec() 重新生成进程时，exec() 将创建单个轻量进程 (lightweight process, LWP)。进程启动代码将生成初始线程。通常，如果初始线程返回，则该线程将调用 exit()，且进程将被销毁。

当进程中的所有线程都退出时，进程将退出。从包含多个线程的进程中调用任何 exec() 函数时将终止所有线程，并装入和执行新的可执行映像。但不会调用 destructor 函数。

计时器、报警与剖析

在 Solaris 2.5 发行版中，已针对每 LWP 计时器和每线程报警声明了“生命周期结束时间”。请参见 timer_create(3RT)、alarm(2) 或 setitimer(2) 手册页。现在，每 LWP 计时器和每线程报警都被替换为每进程变体，这些内容在本节中加以介绍。

最初，每个 LWP 都有唯一的实时时间间隔计时器和报警，与 LWP 绑定的线程可以使用该计时器和报警。当计时器或报警过期时，会向线程传送一个信号。

每个 LWP 还有一个虚拟时间计时器或配置文件时间间隔计时器，与 LWP 绑定的线程可以使用这些计时器。当时间间隔计时器过期时，系统会根据需要将 SIGVTALRM 或 SIGPROF 发送到拥有该时间间隔计时器的 LWP。

每 LWP POSIX 计时器

在 Solaris 2.3 和 2.4 发行版中，timer_create(3RT) 函数返回一个包含计时器 ID 的计时器对象，该 ID 仅在调用 LWP 中有意义。到期信号将被传送到该 LWP。由于返回的计数器对象的行为，只有绑定线程可以使用 POSIX 计时器工具。

即使使用受到限制，Solaris 2.3 和 2.4 发行版中多线程应用程序的 POSIX 计时器也不可靠。这些计时器不能可靠地屏蔽生成的信号，也不能可靠地传送 sigvent 结构中的关联值。

在 Solaris 2.5 发行版中引入的应用程序可以创建每进程计时器。编译应用时定义了 _POSIX_PER_PROCESS_TIMERS 宏，或通过使用大于或等于 199506L 的值定义宏 _POSIX_C_SOURCE 来编译应用程序。

从 Solaris 9 发行版起生效，所有的计时器都针对每个进程，但虚拟时间计时器和配置文件时间间隔计时器除外，它们仍然针对每个 LWP。有关 ITIMER_VIRTUAL 和 ITIMER_PROF，请参见 setitimer(2)。

每进程计时器的计时器 ID 在任何 LWP 中都可用。系统将针对进程（而非针对特定 LWP）生成到期信号。

只能通过 timer_delete(3RT) 或在进程终止时删除每进程计时器。

每线程报警

在 Solaris 2.3 和 2.4 发行版中，对 alarm(2) 或 setitimer(2) 的调用仅在调用 LWP 中有意义。LWP 创建终止时，将自动删除这类计时器。由于此行为，只有 alarm() 或 setitimer() 可以使用绑定线程。

即使限于使用绑定线程，Solaris 2.3 和 2.4 多线程应用程序中的 alarm() 和 setitimer() 计时器也不可靠。特别是，在从发出这些调用的绑定线程屏蔽信号方面，alarm() 和 settimer() 计时器不可靠。如果不需要这类屏蔽，则这两个系统调用可在绑定线程中可靠地工作。

在 Solaris 2.5 发行版中，调用 alarm() 时，与 -lpthread (POSIX) 线程链接的应用程序将获取每个进程传送的 SIGALRM。alarm() 生成的 SIGALRM是针对进程生成，而不是针对特定 LWP 生成。另外，进程终止时，将重置报警。

使用 Solaris 2.5 之前的发行版编译的应用程序或没有与 -lpthread 链接的应用程序将继续查看每个 LWP 传送的信号，这些信号是由 alarm() 和 setitimer() 生成的。

对 alarm() 或 setitimer(ITIMER_REAL) 的调用将导致生成的 SIGALRM 信号被发送到进程（从 Solaris 9 发行版开始生效）。

剖析多线程程序

在 Solaris 2.6 以前的 Solaris 发行版中，在多线程程序中调用 profil() 仅影响调用 LWP。创建 LWP 时不会继承配置文件状态。要使用全局配置文件缓冲区来配置多线程程序，线程启动时每个线程都需要调用 profil()。此外，每个线程必须为绑定线程。这些限制很麻烦。它们不能顺利支持动态打开和关闭剖析。

在 Solaris 2.6 以及更高发行版中，对多线程进程的 profil() 系统调用具有全局影响力。对 profil() 的调用将影响进程中的所有 LWP 和线程。profil() 可能会使与以前的每 LWP 语义相关的应用程序中断。但是，预计调用 profil() 可以改进需要在运行时动态打开和关闭剖析的多线程程序。

非本地转向：setjmp 和 longjmp

setjmp() 和 longjmp() 的范围限于一个线程，该线程在大多数时间是可接受的。但是，限制的范围意味着只有在同一个线程中执行 setjmp() 时，处理信号的线程才能执行 longjmp()。

资源限制

资源限制是对整个进程设置的，且通过添加进程中所有线程的资源使用来加以确定。超过软资源限制时，会向违例线程发送相应的信号。可通过 getrusage(3C) 使用进程中所使用的所有资源。

LWP 和调度类

Solaris 内核具有三种调度类。优先级最高的调度类是实时 (RT) 类。优先级居中的调度类是 system。不能将 system 类应用于用户进程。优先级最低的调度类为分时 (TS) 类，也是缺省类。

系统将针对每个 LWP 维护调度类。创建进程时，初始 LWP 将继承调度类和在父进程中创建 LWP 的优先级。随着所创建进程数目的增多，其关联的 LWP 也会继承此调度类和优先级。

线程具有其基础 LWP 的调度类和优先级。进程中的每个 LWP 都有内核可见的唯一调度类和优先级。

线程优先级可以控制同步对象的争用情况。缺省情况下，LWP 处于分时类中。对于与计算绑定的多线程，线程优先级不是非常有用。对于使用 MT 库频繁执行同步的多线程应用程序，线程优先级更有意义。

调度类是由 priocntl(2) 设置的。指定前两个参数的方式确定只有调用 LWP 还是一个或多个进程的所有 LWP 受影响。priocntl() 的第三个参数是命令，可以是以下命令之一。

- PC_GETCID — 获取特定类的类 ID 和类属性。
- PC_GETCLINFO — 获取特定类的类名称和类属性。
- PC_GETPARMS — 获取进程、包含进程的 LWP 或一组进程的类标识符和类特定调度参数。
- PC_SETPARMS — 设置进程、包含进程的 LWP 或一组进程的类标识符和类特定调度参数。

请注意，priocntl() 会影响与调用线程关联的 LWP 的调度。对于未绑定线程，返回对 priocntl() 的调用后，无法保证调用线程与受影响的 LWP 关联。

分时调度

分时调度可以在分时调度类的 LWP 中公平地分布处理资源。内核的其他部分可以在短时间内独占处理器，而不会缩短用户察觉的响应时间。

priocntl(2) 调用可以设置一个或多个进程的 nice(2) 级别。priocntl() 调用还会影响进程中所有分时类 LWP 的 nice() 级别。nice() 级别的范围通常为 0 到 +20，对于具有超级用户权限的进程，该范围为 -20 到 +20。值越低，优先级越高。

分时 LWP 的分发优先级是根据 LWP 的即时 CPU 使用率及其 nice() 级别计算出来的。nice() 级别指示 LWP 相对于分时调度程序的优先级。

nice() 值越大的 LWP 获得的总处理份额越小，但都为非零值。接收处理量较大的 LWP 的优先级与接收处理量很少或没有接收任何处理的 LWP 的优先级要低。

实时调度

可以将实时类 (RT) 应用于整个进程或应用于进程中的一个或多个 LWP。 必须具有超级用户权限才能使用实时类。

与分时类的 nice(2) 级别不同，可以分别或联合为分类为实时类的 LWP 指定优先级。priocntl(2) 调用将影响进程中所有实时 LWP 的属性。

调度程序始终会分发优先级最高的实时 LWP。当优先级较高的 LWP 可以运行时，优先级高的实时 LWP 优先于优先级较低的 LWP。优先的 LWP 置于其级别队列的开头。

实时 LWP 始终控制着处理器，直到优先处理了 LWP、LWP 暂停或更改了其实时优先级为止。RT 类的 LWP 绝对优先于 TS 类中的进程。

新的 LWP 将继承父进程或 LWP 的调度类。RT 类 LWP 将继承父进程的时间¤ ù，无论是有限的还是无限的。

有限的时间片 LWP 将始终运行，直到 LWP 终止、中断了 I/O 事件、优先级较高的可运行实时进程优先于该 LWP 执行或时间片到期为止。

只有在 LWP 终止、中断或其他实时进程优先于该 LWP 执行，时间片无限的 LWP 才停止执行操作。

公平共享调度程序

公平共享调度程序 (fair share scheduler, FSS) 调度类允许根据份额来分配 CPU 时间。

缺省情况下，FSS 调度类与 TS 和交互式 (interactive, IA) 调度类使用相同的优先级范围（0 到 59）。进程中的所有 LWP 必须在同一调度类中运行。FSS 类将调度单个 LWP，而不是整个进程。因此，混合使用 FSS 和 TS/IA 类中的进程可能会导致在这两种情况下出现意外的调度行为。

TS/IA 或 FSS 调度类进程不会争用相同的 CPU。处理器集可以在系统中混合 TS/IA 与 FSS。但是，每个处理器集中的所有进程都必须属于 TS/IA 调度类或 FSS 调度类。

固定优先级调度

FX（固定优先级）调度类可以指定没有为适应资源占用而调整的固定优先级和时间量程。进程优先级只能由指定优先级的进程或具有适当权限的进程进行更改。有关 FX 的更多信息，请参见 `priocntl`(1) 和 `dispadmin`(1M) 手册页。

此类中的线程与 TS 和交互式 (interactive, IA) 调度类共享相同的优先级范围（0 到 59）。TS 通常为缺省调度类。FX 通常与 TS 结合使用。

扩展传统信号

传统的 UNIX 信号模型通过相当自然的方式扩展到线程。关键特征是信号是在进程范围内部署的，而信号掩码是针对每个进程部署的。信号的进程范围部署是使用传统的机制（`signal`(3C)、`sigaction`(2) 等）建立的。

当信号处理程序标记为 `SIG_DFL` 或 `SIG_IGN` 时，将对整个接收进程执行信号接收操作。这些信号包括退出、核心转储、停止、继续和忽略。系统将针对进程中的所有线程执行这些信号的接收操作。因此，不存在哪个线程拾取信号的问题。退出、核心转储、停止、继续和忽略信号都没有处理程序。有关信号的基本信息，请参见 `signal(5)`。

每个线程都有自己的信号掩码。当线程使用的内存或状态同时也被信号处理程序使用时，可通过信号掩码来阻塞某些信号。进程中的所有线程都共享由 `sigaction`(2) 及其变体设置的一组信号处理程序。

一个进程中的线程不能将信号发送到另一个进程中的特定线程。通过 `kill`(2)、`sigsend`(2) 或 `sigqueue`(3RT) 发送到进程的信号由该进程中任何接收线程来处理。

信号被分为以下类别：陷阱、异常和中断。异常是以同步方式生成的信号。陷阱和中断是以异步方式生成的信号。

就像在传统的 UNIX 中一样，如果信号处于暂挂状态，则该信号的其他实例通常没有其他影响。暂挂信号由位表示，而不是由计数器表示。但是，通过 `sigqueue`(3RT) 接口发送的信号允许在进程中对同一信号的多个实例排队。

对于单线程进程而言，线程接收信号时如果被阻塞在系统调用中，则该线程可能很早就会返回。如果线程很早就返回，则该线程会返回 EINTR 错误代码，或在 I/O 调用中传输的字节数比请求的字节数要少。

对多线程程序特别重要的一点就是信号对 pthread_cond_wait(3C) 产生的影响。此调用通常仅针对 pthread_cond_signal(3C) 或 pthread_cond_broadcast(3C) 返回零，而不会出现任何错误。但是，如果等待线程接收传统的 UNIX 信号，则 pthread_cond_wait() 将返回零，即使唤醒是虚假的也是如此。在这种情况下，Solaris 线程 cond_wait(3C) 函数将返回 EINTR。有关更多信息，请参见第 160 页中的 “中断对条件变量的等待”。

同步信号

陷阱（如 SIGILL、SIGFPE 和 SIGSEGV）是由于对线程执行操作引起的，如除以零或引用不存在的内存。陷阱仅由导致陷阱的线程处理。进程中的多个线程可以同时生成和处理同种类型的陷阱。

可以很容易地针对同时生成的信号将信号扩展到各个线程。可以针对生成同步信号的线程调用处理程序。

但是，如果进程选择不建立相应的信号处理程序，则出现陷阱时将执行缺省操作。即使针对生成的信号阻塞违例线程，也会执行缺省操作。这类信号的缺省操作是终止进程，可能还会进行核心转储。

这类同步信号通常意味着整个进程出现严重问题，而不仅仅是线程出现问题。在这种情况下，终止进程通常是很好的选择。

异步信号

中断（如 SIGINT 和 SIGIO）与任何线程都是异步的，而且是由进程外的某些操作引起的。这些中断可能是其他进程显式发送的信号，也可能代表外部操作（如用户键入 Ctrl-C 组合键）。

中断可由信号掩码允许中断的任何线程来处理。即使有多个线程可以接收中断，也只能选择一个线程。

将同一信号的多个实例发送到进程时，每个实例都可由单独的线程来处理。但是，可用线程不得屏蔽信号。当所有的线程都屏蔽信号时，信号将被标记为**暂挂**，而由取消屏蔽信号的第一个线程来处理信号。

延续语义

延续语义是传统的处理信号的方式。信号处理程序返回时，将控制恢复进程在中断时所处的位置。此控制恢复非常适合于单线程进程中的异步信号，如示例 5–1 所示。

在其他编程语言（如 PL/1）中，此控制恢复还用作异常处理机制。

示例 5–1 延续语义

```
unsigned int nestcount;


unsigned int A(int i, int j) {
    nestcount++;


    if (i==0)
        return(j+1)
    else if (j==0)
        return(A(i-1, 1));
    else
        return(A(i-1, A(i, j-1)));
}


void sig(int i) {
    printf("nestcount = %d\n", nestcount);
}


main() {
    sigset(SIGINT, sig);
    A(4,4);
}
```

对信号执行的操作

本节介绍对信号执行的操作。

设置线程的信号掩码

pthread_sigmask(3C) 对线程执行 sigprocmask(2) 对进程所执行的操作。pthread_sigmask() 可以设置 thread 的信号掩码。创建新的线程时，其初始掩码是从其创建者继承的。

在多线程进程中对 sigprocmask() 执行调用等效于对 pthread_sigmask() 执行调用。有关更多信息，请参见 sigprocmask(2) 手册页。

将信号发送到特定线程

pthread_kill(3C) 是 kill(2) 的线程模拟。pthread_kill() 可以将信号发送到特定线程。发送到指定线程的信号不同于发送到进程的信号。将信号发送到进程时，信号可由该进程中的任何线程来处理。通过 pthread_kill() 发送的信号只能由指定线程来处理。

可以使用 pthread_kill() 将信号仅发送到当前进程中的线程。由于 *thread_t* 类型的线程标识符的范围是本地，因此不能指定当前进程范围以外的线程。

通过目标线程接收信号时，调用的操作（处理程序 SIG_DFL 或 SIG_IGN）通常为全局操作。如果将 SIG*XXX* 发送到线程，且 SIG*XXX* 的作用是中止进程，则目标线程接收信号时将中止整个进程。

等待指定信号

对于多线程程序，sigwait(2) 是可供使用的首选接口，因为 sigwait() 可以很好地处理异步生成的信号。

sigwait() 将导致调用线程等待，直到由其设置参数标识的任何信号被传送到该线程为止。线程等待的同时，系统将取消屏蔽由设置参数标识的信号，但调用返回时将恢复原始掩码。

由设置参数标识的所有信号必定会在所有线程（包括调用线程）上受到阻塞。否则，sigwait() 可能无法正常工作。

使用 sigwait() 将线程与异步信号分离。创建一个侦听异步信号的线程时，可同时创建其他线程来阻塞为此进程设置的任何异步信号。

从 Solaris 2.5 发行版开始，可以使用 sigwait() 的两个版本：Solaris 2.5 版本和 POSIX 标准版本。新的应用程序和新的库应该使用 POSIX 标准接口，因为 Solaris 版本在未来的发行版中可能不可用。

以下示例显示了 sigwait() 的两个版本的语法：

```
#include <signal.h>



/* the Solaris 2.5 version*/

int sigwait(sigset_t *set);



/* the POSIX standard version */

int sigwait(const sigset_t *set, int *sig);
```

传送信号时，POSIX sigwait() 将清除暂挂信号，并将信号数字置于 *sig* 中。许多线程可以同时调用 sigwait()，但是针对每个接收的信号仅返回一个线程。

借助 sigwait()，可以同时处理异步信号。信号到达后，处理这类信号的线程将立即调用 sigwait() 并返回。通过确保所有线程（包括 sigwait() 的调用程序）都屏蔽异步信号，可确保信号仅由预期处理程序处理，且安全地进行处理。

通过始终屏蔽所有线程中的所有信号并在必要时调用 sigwait()，可以使应用程序中依赖于信号的线程的安全性大大提高。

通常，可以创建一个或多个为等待信号而调用 sigwait() 的线程。由于 sigwait() 甚至会检索屏蔽的信号，因此一定要阻塞所有其他线程中的重要信号，以便不会意外传送这些信号。

信号到达时，线程将从 sigwait() 返回，处理信号，并再次调用 sigwait() 以等待更多信号。信号处理线程并不限于使用异步信号安全函数。信号处理线程可采用通常的方式与其他线程同步。第 167 页中的 “MT 接口安全级别”定义了异步信号安全类别。

注 – sigwait() 不能接收同步生成的信号。

在给定时间内等待指定的信号

sigtimedwait(3RT) 类似于 sigwait(2)，但在指定的时间内没有收到信号时，sigtimedwait() 将失败并返回错误。

定向于线程的信号

借助定向于线程的信号的概念，对 UNIX 信号机制得到了扩展。定向于线程的信号就像普通的异步信号一样，但定向于线程的信号是被发送到特定线程，而不是进程。

与安装用于处理信号的信号处理程序相比，使用等待异步信号的单独线程可能更安全且更简单。

一种处理异步信号的更好方式是同步处理这些信号。通过调用 sigwait(2)，线程可以一直等待，直到信号出现为止。请参见第 154 页中的 “等待指定信号”。

示例 5–2 异步信号和 sigwait(2)

```
main() {

    sigset_t set;

    void runA(void);

    int sig;



    sigemptyset(&set);

    sigaddset(&set, SIGINT);

    pthread_sigmask(SIG_BLOCK, &set, NULL);

    pthread_create(NULL, 0, runA, NULL, PTHREAD_DETACHED, NULL);



    while (1) {

        sigwait(&set, &sig);

        printf("nestcount = %d\n", nestcount);

        printf("received signal %d\n", sig);

    }

}



void runA() {

    A(4,4);

    exit(0);

}
```

本示例将修改示例 5–1 的代码。主例程将屏蔽 SIGINT 信号，创建一个子线程（用于调用前一个示例的函数 A），并发出 sigwait() 来处理 SIGINT 信号。

请注意，信号在计算线程中将被屏蔽，因为计算线程将从主线程继承其信号掩码。当且仅当主线程在 sigwait() 内部不受阻塞时，才能受到保护，而不去处理 SIGINT。

另外，请注意，使用 sigwait() 时不存在中断系统调用的危险。

完成语义

另一种处理信号的方式是使用完成语义。

当信号指明发生灾难性情况，导致没有理由继续执行当前代码块时，请使用完成语义。信号处理程序将代替其余有问题的块运行。换句话说，信号处理程序将完成该块。

在示例 5–3 中，所讨论的块是 if 语句的 then 部分的主体。对 setjmp(3C) 的调用会在 *jbuf* 中保存程序的当前寄存器状态并返回 0，从而执行块。

示例 5–3 完成语义

```
sigjmp_buf jbuf;

void mult_divide(void) {

    int a, b, c, d;

    void problem();



    sigset(SIGFPE, problem);

    while (1) {

        if (sigsetjmp(&jbuf) == 0) {

            printf("Three numbers, please:\n");

            scanf("%d %d %d", &a, &b, &c);

            d = a*b/c;

            printf("%d*%d/%d = %d\n", a, b, c, d);

        }

    }
```

示例 5–3 完成语义 （续）

```
}


void problem(int sig) {

    printf("Couldn't deal with them, try again\n");

    siglongjmp(&jbuf, 1);

}
```

如果出现 SIGFPE 浮点异常，则系统将调用信号处理程序。

信号处理程序将调用 siglongjmp(3C)（用于恢复 *jbuf* 中保存的寄存器状态），进而导致程序再次从 sigsetjmp() 返回。保存的寄存器包括程序计数器和栈指针。

但是，此时 sigsetjmp(3C) 将返回 siglongjmp() 的第二个参数（值为 1）。请注意，块将被跳过，而仅在下一次迭代 while 循环期间执行。

可以在多线程程序中使用 sigsetjmp(3C) 和 siglongjmp(3C)。请注意，一个线程永远不会执行使用另一个线程的 sigsetjmp() 结果的 siglongjmp()。

此外，sigsetjmp() 和 siglongjmp() 可以恢复和保存信号掩码，而 setjmp(3C) 和 longjmp(3C) 不会执行这些操作。

使用信号处理程序时，请使用 sigsetjmp() 和 siglongjmp()。

完成语义通常用于处理异常。需要特别指出的是，Sun Ada™ 编程语言就使用此模型。

注 – 请记住，**不得**将 sigwait(2) 与同步信号一同使用。

信号处理程序和异步信号安全

与线程安全类似的概念就是异步信号安全。异步信号安全操作可保证不会干扰正被中断的操作。

当信号处理程序操作干扰正被中断的操作时，就会引发异步信号安全问题。

例如，假设程序正在调用 printf(3S)，且其调用程序调用 printf() 时出现了信号。在这种情况下，两个 printf() 语句的输出彼此关联。要避免关联输出，当 printf() 可能被信号中断时，处理程序不应直接调用 printf()。

无法使用同步元语来解决此问题。 在信号处理程序与正被同步的操作之间执行的任何同步尝试都将立即产生死锁现象。

假设 printf() 借助互斥来保护自身。现在，假设调用 printf() 进而通过互斥锁持有锁定的线程被信号中断。

如果处理程序调用 printf()，则通过互斥锁持有锁定的线程将尝试再次利用互斥锁。尝试利用互斥锁将导致瞬间死锁。

为避免处理程序与操作之间出现干扰，请确保这种情况永远不会发生。或许您可以在关键时候屏蔽信号，或从内部信号处理程序中仅调用异步信号安全操作。

表 5–2 中列出了 POSIX 可确保异步信号安全的仅有例程。任何信号处理程序都可以安全地调用这些函数之一。

表 5–2 异步信号安全函数

_exit()	fstat()	read()	sysconf()
access()	getegid()	rename()	tcdrain()
alarm()	geteuid()	rmdir()	tcflow()
cfgetispeed()	getgid()	setgid()	tcflush()
cfgetospeed()	getgroups()	setpgid()	tcgetattr()
cfsetispeed()	getpgrp()	setsid()	tcgetpgrp()
cfsetospeed()	getpid()	setuid()	tcsendbreak()
chdir()	getppid()	sigaction()	tcsetattr()
chmod()	getuid()	sigaddset()	tcsetpgrp()
chown()	kill()	sigdelset()	time()
close()	link()	sigemptyset()	times()
creat()	lseek()	sigfillset()	umask()
dup2()	mkdir()	sigismember()	uname()
dup()	mkfifo()	sigpending()	unlink()
execle()	open()	sigprocmask()	utime()
execve()	pathconf()	sigsuspend()	wait()
fcntl()	pause()	sleep()	waitpid()
fork()	pipe()	stat()	write()

中断对条件变量的等待

将捕获到的已取消屏蔽的信号传送到等待条件变量的线程时，线程将从虚假唤醒的信号处理程序中返回。 虚假唤醒是指不是由其他线程中的条件信号调用导致的唤醒。在这种情况下，Solaris 线程接口 `cond_wait()` 和 `cond_timedwait()` 将返回 `EINTR`，而 POSIX 线程接口 `pthread_cond_wait()` 和 `pthread_cond_timedwait()` 将返回 0。在所有情况下，从条件等待返回之前都将重新获取关联的互斥锁定。

重新获取关联的互斥锁定并不暗示线程在执行信号处理程序的同时，互斥处于锁定状态。未定义信号处理程序中的互斥状态。

由于在 Solaris 9 发行版之前的 Solaris 软件发行版中实现了 `libthread`，因而保证了在处于信号处理程序中时保留了互斥。依赖此原有行为的应用程序需要修改 Solaris 9 发行版以及后续发行版。

示例 5–4 将对处理程序清除加以说明。

示例 5–4 条件变量和中断的等待

```
int sig_catcher() {

    sigset_t set;

    void hdlr();



    mutex_lock(&mut);



    sigemptyset(&set);

    sigaddset(&set, SIGINT);

    sigsetmask(SIG_UNBLOCK, &set, 0);



    if (cond_wait(&cond, &mut) == EINTR) {

        /* signal occurred and lock is held */

        cleanup();

        mutex_unlock(&mut);

        return(0);
```

示例 5–4 条件变量和中断的等待 （续）

```
    }

    normal_processing();

    mutex_unlock(&mut);

    return(1);

}



void hdlr() {

    /* state of the lock is undefined */

    ...

}
```

假设 SIGINT 信号在进入 sig_catcher() 时在所有线程中受到阻塞。此外，还假设已通过调用 sigaction(2) 建立了 hdlr()（作为 SIGINT 信号的处理程序）。如果在线程处于 cond_wait() 中时将捕获到的已取消屏蔽的 SIGINT 信号实例传送到该线程，该线程将调用 hdlr()。然后，线程将返回到 cond_wait() 函数（如有必要，将在此处重新获取互斥锁定），并从 cond_wait() 返回 EINTR。

是否已针对 sigaction() 将 SA_RESTART 指定为标志在此处无影响。cond_wait(3C) 不是系统调用，也不会自动重新启动。如果在 cond_wait() 中阻塞线程时出现捕获的信号，则调用将始终返回 EINTR。

I/O 问题

多线程编程的最大优点之一就是可以提升 I/O 性能。传统的 UNIX API 在这方面给您提供的帮助极少。要么就使用文件系统的功能，要么就整个跳过文件系统。

本节说明如何使用线程通过 I/O 并发性和多缓冲来获得更多灵活性，此外，本节还论述了包含线程的同步 I/O 与包含和不包含线程的异步 I/O 的各种方式之间的差异和相似之处。

I/O 作为远程过程调用

在传统的 UNIX 模型中，I/O 看上去是同步的，就像对 I/O 设备进行远程过程调用一样。调用返回后，I/O 即完成，或者至少看上去已完成。例如，写入请求可能仅导致将数据传输到操作环境中的缓冲区。

此模型的优点在于过程调用的概念是为用户所熟知的。

在传统 UNIX 系统中未使用的一种替代方法是异步模型，在此模型中，I/O 请求仅启动操作。程序必须以某种方式了解操作完成的时间。

异步模型不像同步模型那样简单。但是，异步模型的优点是允许并发 I/O 和在传统单线程 UNIX 进程中的处理。

人为的异步性

通过在多线程程序中使用同步 I/O，可以获得异步 I/O 的大多数优势。使用异步 I/O 时，可以发出请求并随后检查以确定 I/O 完成的时间。可以改用单独的线程同步执行 I/O。随后，主线程或许会在以后的某个时间通过调用 `pthread_join`(3C) 来检查是否完成了操作。

异步 I/O

在多数情况下，不需要异步 I/O，因为其效果可借助线程得以实现，每个线程执行同步 I/O。但是，在少数情况下，线程不能实现异步 I/O 可以实现的效果。

最直观的示例就是写入磁带机以形成磁带机流。流形式可防止在向磁带机写入内容的同时磁带机停止运行。磁带将高速向前移动，同时提供写入磁带的连续不断的数据流。

要支持流形式，内核中的磁带机应使用线程。内核中的磁带机对中断做出响应时，该磁带机一定会发出排队的写入请求。中断指示以前的磁带写入操作已完成。

线程不能保证会对异步写入进行排序，因为线程执行的顺序是不确定的。例如，您无法指定写入磁带的顺序。

异步 I/O 操作

```
#include <sys/asynch.h>



int aioread(int fildes, char *bufp, int bufs, off_t offset,

    int whence, aio_result_t *resultp);



int aiowrite(int filedes, const char *bufp, int bufs,

    off_t offset, int whence, aio_result_t *resultp);
```

```
aio_result_t *aiowait(const struct timeval *timeout);


int aiocancel(aio_result_t *resultp);
```

aioread(3AIO) 和 aiowrite(3AIO) 在格式上类似于 pread(2) 和 pwrite(2)，但是添加了最后一个参数。对 aioread() 和 aiowrite() 的调用导致启动 I/O 操作或将该操作排入队列。

调用将顺利返回，而不被阻塞，而且调用的状态将在 *resultp* 所指向的结构中返回。*resultp* 是 aio_result_t 类型的项，其中包含以下值：

```
int aio_return;

int aio_errno;
```

当调用立即失败时，可以在 aio_errno 中找到失败代码。否则，此字段将包含 AIO_INPROGRESS，意味着已成功地将操作排入队列。

等待 I/O 操作完成

通过调用 aiowait(3AIO)，可以等待未完成的异步 I/O 操作完成。aiowait() 将返回指向 aio_result_t 结构（随原始 aioread(3AIO) 或原始 aiowrite(3) 调用一同提供）的指针。

此时，aio_result_t 将包含 read(2) 或 write(2) ?ò a?|^ao?H?S，前提是要调用其中一个函数而不是异步版本。如果 read() 或 write() 成功，则 aio_return 包含已读取或写入的字节数。如果 read() 或 write() 不成功，则 aio_return 为 -1，且 aio_errno 包含错误代码。

aiowait() 将使用 *timeout* 参数，该参数指示调用程序将等待的时间。此处的 NULL 指针表示调用程序将无限期等下去。指向包含零值的结构的指针表示调用程序根本不会等待。

您可能会启动异步 I/O 操作，执行某项工作，然后调用 aiowait() 以等待请求完成。或者，可以使用 SIGIO 在操作完成时以异步方式得到通知。

最后，可通过调用 aiocancel() 来取消暂挂的异步 I/O 操作。此例程是使用结果区域地址作为参数来进行调用的。此结果区域标识哪项操作将被取消。

共享的 I/O 和新的 I/O 系统调用

多个线程执行具有相同文件描述符的并发 I/O 操作时，您可能会发现传统的 UNIX I/O 接口不是线程安全的。在 lseek(2) 系统调用设置了文件偏移的位置，会出现不连续 I/O 问题。随后将在接下来的 read(2) 或 write(2) 调用中使用该文件偏移，以指示操作应在文件中的哪个位置开始。当两个或更多线程向同一文件描述符发出 lseeks() 时，将产生冲突。

为避免此冲突，请使用 pread(2) 和 pwrite(2) 系统调用。

```
#include <sys/types.h>

#include <unistd.h>
```

```
ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);


ssize_t pwrite(int filedes, void *buf, size_t nbyte,

    off_t offset);
```

pread(2) 和 pwrite(2) 的行为方式与 read(2) 和 write(2) 非常类似，但是 pread(2) 和 pwrite(2) 多使用了一个参数（即文件偏移）。可以使用此参数来指定偏移，而无需使用 lseek(2)，因此多个线程可以安全地使用这些例程来处理对同一文件描述符的 I/O。

getc 和 putc 的替代项

标准 I/O 也会出现问题。程序员习惯使用例程（如 getc(3C) 和 putc(3C)），这些例程以宏方式实现，且速度非常快。由于 getc(3C) 和 putc(3C) 的速度较快，因此可以在程序的内部循环中使用这些宏，而不必担心效率。

但是，将 getc(3C) 和 putc(3C) 设为线程安全时，宏的使用代价会突然变高。现在，宏至少需要两个内部子例程调用来锁定和解除锁定互斥。

为避开此问题，提供了这些例程的替代版本 getc_unlocked(3C) 和 putc_unlocked(3C)。

getc_unlocked(3C) 和 putc_unlocked(3C) 不会获得对互斥的锁定。这些宏的速度像原始非线程安全版本的 getc(3C) 和 putc(3C) 一样快。

但是，要采用线程安全方式使用这些宏，必须使用 flockfile(3C) 和 funlockfile(3C) 显式锁定和释放保护标准 I/O 流的互斥。对其中靠后例程的调用是在循环外进行的。对 getc_unlocked() 或 putc_unlocked() 的调用是在循环内进行的。

◆ ◆ ◆ 第 6 章

安全和不安全的接口

本章定义函数和库的 MT 安全级别。本章论述以下主题：

线程安全

线程安全可以避免数据竞争。不管数据值设置的正确与否，都会出现数据争用的情况，具体取决于多个线程访问和修改数据的顺序。

不需要共享时，请为每个线程提供一个专用的数据副本。如果共享非常重要，则提供显式同步，以确保程序以确定的方式操作。

当某个过程由多个线程同时执行时，如果该过程在逻辑上是正确的，则认为该过程是线程安全的。实际上，一般可分为以下几种安全性级别。

- 不安全
- 线程安全，可串行化
- 线程安全，MT 安全

通过将过程包含在语句中来锁定和解除锁定互斥，可以使不安全过程变成线程安全过程，而且可以进行串行化。示例 6–1 说明了 `fputs()` 的三个简化实现，最初线程是不安全的。

下面是此例程的可串行化版本，它使用单一互斥来防止过程出现并发执行问题。实际上，单一互斥比通常需要的同步效果更强。当两个线程使用 `fputs()` 将输出发送到不同文件时，一个线程无需等待另一个线程。只有在共享输出文件时，线程才需要进行同步。

最新版本是 MT 安全的。此版本对每个文件都使用一个锁定，允许两个线程同时指向不同的文件。因此，只要例程是线程安全的，该例程就是 MT 安全的，而且例程的执行不会对性能造成负面影响。

示例6–1线程安全程度

```
/* not thread-safe */
fputs(const char *s, FILE *stream) {
    char *p;
    for (p=s; *p; p++)
        putc((int)*p, stream);
}

/* serializable */
fputs(const char *s, FILE *stream) {
    static mutex_t mut;
    char *p;
    mutex_lock(&m);
    for (p=s; *p; p++)
        putc((int)*p, stream);

    mutex_unlock(&m);
}

/* MT-Safe */
mutex_t m[NFILE];
fputs(const char *s, FILE *stream) {
    static mutex_t mut;
    char *p;
```

示例 6–1 线程安全程度 （续）

```
    mutex_lock(&m[fileno(stream)]);

    for (p=s; *p; p++)

        putc((int)*p, stream);

    mutex_unlock(&m[fileno(stream)]0;

}
```

MT 接口安全级别

线程手册页 `man(3C)` 使用表 6–1 中列出的安全级别类别来描述接口对线程的支持程度。这些类别在 `Intro(3)` 手册页中进行了完整说明。

表 6–1 接口安全级别

类别	说明
安全	可以从多线程应用程序中调用此代码
安全（包含异常）	有关异常的说明，请参见对应手册页的 NOTES 部分。
不安全	此接口与多线程应用程序结合使用时是不安全的，除非应用程序一次仅安排在库中执行一个线程。
MT 安全	此接口已完全做好准备，可以执行多线程访问。此接口是**安全的**，而且支持一定的并发性。
MT 安全（包含异常）	有关异常的列表，请参见《man pages section 3: Basic Library Functions》中对应页中的 NOTES 部分。
异步信号安全	可以从信号处理程序中安全地调用此例程。执行异步信号安全例程的线程在被信号中断时，不会自行死锁。
Fork1–安全	每次调用 Solaris `fork1(2)` 或 POSIX `fork(2)` 时，此接口都会释放持有的锁定。

有关库例程的安全级别，请参见参考手册页的第 3 部分。

出于以下原因，特意未将某些函数设为安全的。

- 设置为 MT 安全的接口对单线程应用程序的性能有负面影响。
- 库具有不安全的接口。例如，函数可能会返回指向栈中缓冲区的指针。可以对其中的某些函数使用可重复执行的对应函数。可重复执行的函数名称是在原始函数名称后附加 "_r"。

注意 – 确定名称不以 "_r" 结尾的函数是否是 MT 安全的唯一方法就是查看该函数的手册页。必须使用同步设备或通过限制初始线程来保护对标识为非 MT 安全的函数的使用。

不安全接口的可重复执行函数

对于包含不安全接口的大多数函数而言，存在例程的 MT 安全版本。新的 MT 安全例程的名称始终为原有不安全例程的名称附加 "_r" 后的形式。Solaris 环境中提供表 6–2 "_r" 例程。

表 6–2 可重复执行函数

asctime_r(3c)	gethostbyname_r(3n)	getservbyname_r(3n)
ctermid_r(3s)	gethostent_r(3n)	getservbyport_r(3n)
ctime_r(3c)	getlogin_r(3c)	getservent_r(3n)
fgetgrent_r(3c)	getnetbyaddr_r(3n)	getspent_r(3c)
fgetpwent_r(3c)	getnetbyname_r(3n)	getspnam_r(3c)
fgetspent_r(3c)	getnetent_r(3n)	gmtime_r(3c)
gamma_r(3m)	getnetgrent_r(3n)	lgamma_r(3m)
getauclassent_r(3)	getprotobyname_r(3n)	localtime_r(3c)
getauclassnam_r(3)	getprotobynumber_r(3n)	nis_sperror_r(3n)
getauevent_r(3)	getprotoent_r(3n)	rand_r(3c)
getauevnam_r(3)	getpwent_r(3c)	readdir_r(3c)
getauevnum_r(3)	getpwnam_r(3c)	strtok_r(3c)
getgrent_r(3c)	getpwuid_r(3c)	tmpnam_r(3s)
getgrgid_r(3c)	getrpcbyname_r(3n)	ttyname_r(3c)
getgrnam_r(3c)	getrpcbynumber_r(3n)	
gethostbyaddr_r(3n)	getrpcent_r(3n)	

异步信号安全函数

可以从信号处理程序中安全调用的函数就是**异步信号安全**函数。POSIX 标准定义并列出了异步信号安全函数（IEEE Std 1003.1-1990, 3.3.1.3 (3)(f)，第 55 页）。除 POSIX 异步信号安全函数外，Solaris 线程接口中的以下函数也是异步信号安全函数：

- sema_post(3C)
- thr_sigsetmask(3C)，类似于 pthread_sigmask(3C)

- `thr_kill(3C)`，类似于 `pthread_kill(3C)`

库的 MT 安全级别

所有可能由线程从多线程程序中调用的例程都应该是 MT 安全的。因此，例程的两项或多项激活操作必须能够同时**正确**执行。这样，多线程程序使用的每个库接口都必须是 MT 安全级别。

目前，并非所有库都是 MT 安全的。表 6–3 中列出了常用的 MT 安全库。其他的库最终会被修改为 MT 安全的。

表 6–3 部分 MT 安全的库

库	注释
`lib/libc`	不安全的接口具有 `*_r` 形式的线程安全接口，通常包含不同的语义。
`lib/libdl_stubs`	支持静态切换编译
`lib/libintl`	国际化库
`lib/libm`	符合 System V Interface Definition, Edition 3, X/Open and ANSI C 的数学库
`lib/libmalloc`	空间有效内存分配库，请参见 `malloc(3X)`
`lib/libmapmalloc`	基于 `mmap` 的备选内存分配库，请参见 `mapmalloc(3X)`
`lib/libnsl`	TLI 接口、XDR、RPC 客户机和服务器、`netdir`、`netselect` 以及 `getXXbyYY` 接口都不是安全的，但都具有 `getXXbyYY_r` 形式的线程安全接口
`lib/libresolv`	线程特定 errno 支持
`lib/libsocket`	用于执行网络连接的套接字库
`lib/libw`	支持多字节语言环境的宽字符和宽字符串函数
`lib/straddr`	名称到地址的网络转换库
`lib/libX11`	X11 Windows 库例程
`lib/libC`	C++ 运行时共享对象

不安全库

只有在单线程调用时，多进程程序才能安全地调用库中无法保证是 MT 安全级别的例程。

◆ ◆ ◆ 第 7 章

7 编译和调试

本章介绍如何编译和调试多线程程序。本章论述以下主题：

编译多线程应用程序

许多选项可用于头文件、定义标志和链接。

为编译做准备

编译和链接多线程程序时，需要以下项。Solaris 软件应包括除 C 编译器以外的所有项。

- 标准 C 编译器
- 包括以下文件:
  - `<thread.h>` 和 `<pthread.h>`
  - `<errno.h>`, `<limits.h>`, `<signal.h>` , `<unistd.h>`
- 常规 Solaris 链接程序 `ln`(1)
- Solaris 线程库 (`libthread`)、POSIX 线程库 (`libpthread`)，可能还有信号的 POSIX 实时库 (`librt`)（仅限于 Solaris 9 和以前的发行版）
- MT 安全库，如 `libc`、`libm`、`libw`、`libintl`、`libnsl`、`libsocket`、`libmalloc`、`libmapmalloc` 等

选择 Solaris 语义或 POSIX 语义

某些函数在 POSIX 标准中的语义与在 Solaris 2.4 发行版中的语义是不同的，Solaris 2.4 发行版基于早期的 POSIX 草案。函数定义是在编译时选择的。有关预期参数和返回值中差异的说明，请参见手册页 section 3: Library Interfaces and Headers。以下是具有不同语义的函数：

- asctime_r(3C)
- ctime_r(3C)
- ftrylockfile(3C)（新增）
- getgrgid_r(3C)
- getgrnam_r(3C)
- getlogin_r(3C)
- getpwnam_r(3C)
- getpwuid_r(3C)
- readdir_r(3C)
- sigwait(2)
- ttyname_r(3C)

在 Solaris 9 和以前的发行版中，Solaris fork(2) 函数可以复制所有的线程 *fork-all* 行为。POSIX fork(2) 函数仅复制调用线程 *fork-one* 行为，与 Solaris fork1() 函数是一样的。

从 Solaris 10 发行版开始，fork() 的行为在未链接到 -lpthreaad 时可能会发生更改，以与 POSIX 版本保持一致。需要特别指出的是，fork() 被重新定义为fork1()。因此，fork() 将复制子进程中的调用线程。所有 Solaris 发行版中都支持 fork1() 的行为。新函数 forkall() 可以针对需要将父进程的所有线程复制到子进程中的应用程序提供此行为。

包括 <thread.h> 或 <pthread.h>

包括文件 <thread.h> 可以编译与早期的 Solaris 软件发行版向上兼容的代码。此文件包含 Solaris 线程接口的声明。要使用 POSIX 线程调用 thr_setconcurrency(3C)，程序需要包括 <thread.h>。

包括文件 <pthread.h>（与 -lpthread 库结合使用）可以编译符合 POSIX 标准定义的多线程接口的代码。为了与 POSIX 完全符合，应该将功能测试宏 _POSIX_C_SOURCE 的值 (long) 设置为 ≥ 199506。请参见 standards(5) 手册页。

对于 1996 版 POSIX 标准：

```
cc89 -D_POSIX_C_SOURCE=199506L [flags] file
```

对于 2001 版 POSIX 标准：

```
cc99 -D_POSIX_C_SOURCE=200112L [flags] file ... [-l rt]
```

可以在同一个应用程序中混合使用 Solaris 线程与 POSIX 线程。请在应用程序中同时包括 <thread.h> 和 <pthread.h>。

如果二者混合使用，则当使用 -D_REENTRANT 编译时，将采用 Solaris 语义，而当使用 -D_POSIX_C_SOURCE 编译时，将采用 POSIX 语义。

定义 _REENTRANT 或 _POSIX_C_SOURCE

对于 POSIX 行为，请使用 -D_POSIX_C_SOURCE 标志集 ≥ 199506L 来编译应用程序。对于 Solaris 行为，请使用 -D_REENTRANT 标志来编译多线程程序。这些编译器标志适用于应用程序的每个模块。

对于混合的应用程序，具有 POSIX 语义的 Solaris 线程使用 -D_REENTRANT 和 -D_POSIX_PTHREAD_SEMANTICS 标志进行编译。

要编译单线程应用程序，请不要定义 -D_REENTRANT 标志，也不要定义 -D_POSIX_C_SOURCE 标志。不存在这些标志时，errno、stdio 等的所有原有定义仍然生效。

注 – 请在不使用 -D_REENTRANT 标志的条件下编译单线程应用程序。使用这种方式编译单线程应用程序，以避免将宏（如 putc(3s)）转换为可重复执行函数调用时引起的性能降低。

总之，定义 -D_POSIX_C_SOURCE 的 POSIX 应用程序将获取例程的 POSIX 语义。仅定义 -D_REENTRANT 的应用程序将获取这些例程的 Solaris 语义。定义 -D_POSIX_PTHREAD_SEMANTICS 的 Solaris 应用程序将获取这些例程的 POSIX 语义，但仍然可以使用 Solaris 线程接口。

同时定义 -D_POSIX_C_SOURCE 和 -D_REENTRANT 的应用程序将获取 POSIX 语义。

使用 libthread 或 libpthread 链接

对于 POSIX 线程行为（在 Solaris 9 和以前的发行版中），请装入 libpthread 库。对于 Solaris 线程行为，请装入 libthread 库。有的 POSIX 程序员可能想使用 -lthread 进行链接，以保留 fork() 与 fork1() 之间的 Solaris 区别。-lpthread 库使 fork() 的行为方式与 Solaris fork1() 调用的行为方式相同。

在 Solaris 10 和后续发行版中，两个线程库都不再是必需的，但是仍然可以为了实现兼容而指定库。所有的线程功能都已被移入标准 C 库中。要使用 libthread，请在 ld 命令行的 lc 前面指定 -lthread，或在 cc 命令行的末尾指定 -lthread。

要使用 libthread，请在 ld 命令行的 -lc 前面指定 -lthread，或在 cc 命令行的末尾指定 -lthread。

要使用 libpthread，请在 ld 命令行的 -lc 前面指定 -lpthread，或在 cc 命令行的末尾指定 -lpthread。

在 Solaris 9 发行版之前，不应使用 -lthread 或 -lpthread 来链接非线程程序。这样做将在链接时建立在运行时启动的多线程机制。这些机制将使单线程应用程序的速度变慢，浪费系统资源，而且会在调试代码时产生误导性结果。

在 Solaris 9 和后续发行版中，使用 -lthread 或 -lpthread 链接非线程应用程序时不会为程序产生语义差异。也不会创建额外的线程或额外的 LWP。只有主线程会像传统的单线程进程一样执行操作。对程序的唯一影响就是使系统库锁定成为实际锁定，与伪函数调用相反。您必须为获取无竞争锁定付出代价。

在 Solaris 10 发行版之前，如果应用程序没有链接 -lthread 或 -lpthread，则对 libthread 和 libpthread 的所有调用都为空操作指令。运行时库 libc 具有许多预定义 libthread 和 libpthread 存根，这些存根都是空过程。当应用程序同时链接了 libc 和线程库时，将通过 libthread 或 libpthread 插入实际过程。

注 – 对于使用线程的 C++ 程序，请使用 -mt 选项（而不是 -lthread）来编译和链接应用程序。-mt 选项与 libthread 链接，并且能确保正确的库链接顺序。-lthread 可能会导致程序进行核心转储。

在 POSIX 环境中链接

对于 1996 版 POSIX 标准，请使用以下选项来编译和链接应用程序：

```
cc89 -D_POSIX_C_SOURCE=199506L [flags] file ... [-l rt]
```

对于 2001 版 POSIX 标准，请使用以下选项来编译和链接应用程序：

```
cc99 -D_POSIX_C_SOURCE=200112L [flags] file ... [-l rt]
```

在 Solaris 环境中链接

在 Solaris 线程环境中，请使用以下选项来编译和链接应用程序：

```
cc -D_REENTRANT -D POSIX_THREAD_SEMANTICS [flags] file ... [-l rt]
```

在混合环境中链接

在混合环境中，请使用以下选项来编译和链接应用程序：

```
cc -D_REENTRANT [flags] file ... [-l rt]
```

在混合使用时，需要包括 thread.h 和 pthread.h。

与 POSIX 信号的 -lrt 链接

Solaris 信号例程 sema_*(3C) 包含在标准的 C 库中。相对而言，您可以链接 -lrt 库，从而获取第 121 页中的 “使用信号进行同步”中所述的标准 sem_*(3R) POSIX 信号例程。

将原有模块与新模块链接

表 7–1 说明，在将多线程对象模块与原有对象模块链接时应格外小心。

表 7–1 使用或不使用 _REENTRANT 标志进行编译

文件类型	编译	参考	返回值
原有对象文件（非线程）和新对象文件	**不使用** _REENTRANT 或 _POSIX_C_SOURCE 标志	静态存储	传统的 *errno*
新对象文件	**使用** _REENTRANT 或 _POSIX_C_SOURCE 标志	__errno，新的二进制入口点	线程的 errno 定义地址
在 libnsl 中使用 TLI 的程序要获取 TLI 全局错误变量，请包括 tiuser.h。	**使用** _REENTRANT 或 _POSIX_C_SOURCE 标志（必需）	__t_errno，新的入口点	线程的 t_errno 定义地址。

备用线程库

Solaris 8 发行版引入了备用的线程库实现，位于目录 /usr/lib/lwp (32 位) 和 /usr/lib/lwp/64 (64 位) 中。 在 Solaris 9 发行版中，此实现成为标准的线程实现（位于 /usr/lib 和 /usr/lib/64 中）。从 Solaris 10 发行版开始生效，所有的线程功能都已被移入 libc 中，不再需要任何单独的线程库。

调试多线程程序

下面论述的内容介绍了一些可能在多线程程序中导致错误的特征。

多线程程序中常见的疏忽性问题

以下列表指出了在多线程程序中可能导致错误的一些经常被疏忽的问题。

- 将指针作为新线程的参数传递给调用方栈。
- 在没有同步机制保护的情况下访问全局内存的共享可更改状态。
- 两个线程尝试轮流获取对同一对全局资源的权限时导致死锁。 其中一个线程控制第一种资源，另一个线程控制第二种资源。其中一个线程放弃之前，任何一个线程都无法继续操作。
- 尝试重新获取已持有的锁（递归死锁）。
- 在同步保护中创建隐藏的间隔。 如果受保护的代码段包含的函数释放了同步机制，而又在返回调用方之前重新获取了该同步机制，则将在保护中出现此间隔。结果具有误导性。对于调用方，表面上看全局数据已受到保护，而实际上未受到保护。
- 将 UNIX 信号与线程混合时，使用 sigwait(2) 模型来处理异步信号。
- 调用 setjmp(3C) 和 longjmp(3C)，然后长时间跳跃，而不释放互斥锁。
- 从对 *_cond_wait() 或 *_cond_timedwait() 的调用中返回后无法重新评估条件。

- 忘记已创建缺省线程 PTHREAD_CREATE_JOINABLE 并且必须使用 pthread_join(3C) 来进行回收。请注意，pthread_exit(3C) 不会释放其存储空间。
- 执行深入嵌套、递归调用以及使用大型自动数组可能会导致问题，因为多线程程序与单线程程序相比对栈大小的限制更多。
- 指定不适当的栈大小，或使用非缺省栈。

多线程程序（特别是那些包含错误的程序）经常在两次连续运行中的行为方式不同，即使输入相同也是如此。此行为是由线程调度顺序的差异所导致的。

一般情况下，多线程错误是统计得出的，不具有确定性。通常，与基于断点的调试相比，跟踪是用于查找执行顺序问题的一种更有效的方法。

使用 TNF 实用程序跟踪和调试

请使用 TNF 实用程序跟踪、调试和收集应用程序和库中的性能分析信息。TNF 实用程序将内核以及多个用户进程和线程中的跟踪信息整合在一起。TNF 实用程序对于多线程代码特别有用。TNF 实用程序包括在 Solaris 软件中，是该软件的一部分。

使用 TNF 实用程序，可以轻松跟踪和调试多线程程序。有关使用 prex(1) 和 tnfdump(1) 的详细信息，请参见 TNF 手册页。

使用 truss

有关跟踪系统调用、信号和用户级别函数调用的信息，请参见 truss(1) 手册页。

使用 mdb

有关 mdb 的信息，请参见《Solaris Modular Debugger Guide》。

可以使用下面的 mdb 命令来访问多线程程序的 LWP。

$l	如果目标为用户进程，则将列显有代表性的线程的 LWP ID。
$L	如果目标为用户进程，则将列显目标中每个 LWP 的 LWP ID。
pid::attach	附加到编号为 *pid* 的进程。
::release	释放以前附加的进程或核心转储文件。随后可以由 prun(1) 继续处理进程，或者可通过应用 MDB 或其他调试器来恢复进程。
address::context	上下文切换到指定进程。

这些用于设置条件断点的命令通常很有用。

[*addr*] ::bp [+/-dDestT] [-c *cmd*] [-n *count*] *sym* . . .
在指定的位置设置断点。

addr ::delete [*id* | all]
删除包含给定 ID 编号的事件说明符。

使用 dbx

使用 dbx 实用程序，可以调试和执行使用 C++、ANSI C 和 FORTRAN 编写的源代码程序。dbx 与调试器接受同样的命令，但使用标准的终端 (TTY) 接口。dbx 和调试器都支持调试多线程程序。有关如何启动 dbx 的说明，请参见 dbx(1) 手册页。有关 dbx 的概述，请参见《Debugging a Program With dbx》。调试器功能在 dbx 的调试器 GUI 的联机帮助中介绍。

表 7–2 中列出的所有 dbx 选项均可支持多线程应用程序。

表 7–2 MT 程序的 dbx 选项

选项	操作
cont at line [-sig signo id]	在包含信号 *signo* 的 *line* 中继续执行操作。*id*（如果存在）指定哪个线程或 LWP 继续操作。缺省值为 *all*。
lwp	显示当前的 LWP。切换到给定 LWP [lwpid]。
lwps	列出当前进程中所有的 LWP。
next ... tid	单步执行给定线程。跳过函数调用时，所有的 LWP 都会在该函数调用期间隐式恢复。不能单步执行非活动线程。
next ... lid	单步执行给定 LWP。跳过函数时不会隐式恢复所有 LWP。所含的给定线程处于活动状态的 LWP。跳过函数时不会隐式恢复所有 LWP。
step... tid	单步执行给定线程。跳过函数调用时，所有的 LWP 都会在该函数调用期间隐式恢复。不能单步执行非活动线程。
step... lid	单步执行给定 LWP。跳过函数时不会隐式恢复所有 LWP。
stepi... lid	给定的 LWP。
stepi... tid	所含的给定线程处于活动状态的 LWP。
thread	显示当前线程。切换到线程 *tid*。在下面所有的变体中，可选的 *tid* 表示当前线程。
thread -info [tid]	列显有关给定线程的所有已知信息。
thread -blocks [tid]	列显阻塞其他线程的给定线程持有的所有锁定。
thread -suspend [tid]	使给定线程进入暂停状态。
thread -resume [tid]	取消暂停给定线程。

表 7–2 MT 程序的 dbx 选项 （续）

选项	操作
`thread -hide [ tid ]`	**隐藏**给定线程或当前线程。该线程不会出现在通用的 `threads` 列表中。
`thread -unhide [ tid ]`	**取消隐藏**给定线程或当前线程。
`thread -unhide all`	**取消隐藏**所有线程。
`threads`	列显所有已知线程的列表。
`threads -all`	列显通常不会列显的线程（僵线程）。
`threads -mode all|filter`	控制在缺省情况下，`threads` 是列显所有线程，还是过滤线程。
`threads -mode auto|manual`	实现线程列表的自动更新。
`threads -mode`	回显当前模式。线程或 LWP ID 可以按照以前的任何形式来追溯指定实体。

◆ ◆ ◆ 第 8 章

Solaris 线程编程

本章比较了 Solaris 线程和 POSIX 线程的应用程序编程接口 (application programming interface, API)，并介绍了 POSIX 线程中没有的 Solaris 功能。本章讨论以下主题：

比较 Solaris 线程和 POSIX 线程的 API

Solaris 线程 API 和 pthread API 是同一问题的两种不同解决方案，即在应用程序软件中建立并行性。尽管每个 API 都是完整的，但是可以安全地在同一程序中混合使用 Solaris 线程函数和 pthread 函数。

不过，这两个 API 并不完全匹配。Solaris 线程支持 pthread 中没有的函数，而 pthread 中则包括 Solaris 接口不支持的函数。对于那些**匹配**的函数，尽管信息内容实际相同，但是关联参数可能并不相同。

通过合并这两个 API，可以使用仅存在于其中一个 API 中的功能来增强另一个 API。同样，在同一个系统中还可以同时运行仅使用 Solaris 线程的应用程序和仅使用 pthread 的应用程序。

API 的主要差异

Solaris 线程和 pthread 在 API 操作和语法方面非常相似。表 8–1 中列出了两者之间的主要差异。

表 8–1 Solaris 线程和 pthread 的独有功能

Solaris 线程	POSIX 线程
使用 thr_ 前缀表示线程函数的名称，使用 sema_ 前缀表示信号函数的名称	使用 pthread_ 前缀表示 pthread 函数的名称，使用 sem_ 前缀表示信号函数的名称
能够创建“守护进程”线程	取消语义
暂停和继续执行线程	调度策略

函数比较表

下表对 Solaris 线程函数和 pthread 函数进行了比较。请注意，即使 Solaris 线程函数和 pthread 函数看上去完全相同，但是其参数可以不同。

如果某个进行比较的接口在 pthread 或 Solaris 线程中不可用，则会在相应的列中显示一个连字符 "-"。pthread 列中后跟 "POSIX.1b" 的各项属于 POSIX 实时标准规范，而不属于 pthread。

表 8–2 Solaris 线程和 POSIX pthread 的比较

Solaris 线程	pthread
thr_create()	pthread_create()
thr_exit()	pthread_exit()
thr_join()	pthread_join()
thr_yield()	sched_yield() POSIX.1b
thr_self()	pthread_self()
thr_kill()	pthread_kill()
thr_sigsetmask()	pthread_sigmask()
thr_setprio()	pthread_setschedparam()
thr_getprio()	pthread_getschedparam()
thr_setconcurrency()	pthread_setconcurrency()
thr_getconcurrency()	pthread_getconcurrency()
thr_suspend()	-
thr_continue()	-
thr_keycreate()	pthread_key_create()
-	pthread_key_delete()

表 8–2 Solaris 线程和 POSIX pthread 的比较 （续）

Solaris 线程	pthread
thr_setspecific()	pthread_setspecific()
thr_getspecific()	pthread_getspecific()
-	pthread_once()
-	pthread_equal()
-	pthread_cancel()
-	pthread_testcancel()
-	pthread_cleanup_push()
-	pthread_cleanup_pop()
-	pthread_setcanceltype()
-	pthread_setcancelstate()
mutex_lock()	pthread_mutex_lock()
mutex_unlock()	pthread_mutex_unlock()
mutex_trylock()	pthread_mutex_trylock()
mutex_init()	pthread_mutex_init()
mutex_destroy()	pthread_mutex_destroy()
cond_wait()	pthread_cond_wait()
cond_timedwait()	pthread_cond_timedwait()
cond_reltimedwait()	pthread_cond_reltimedwait_np()
cond_signal()	pthread_cond_signal()
cond_broadcast()	pthread_cond_broadcast()
cond_init()	pthread_cond_init()
cond_destroy()	pthread_cond_destroy()
rwlock_init()	pthread_rwlock_init()
rwlock_destroy()	pthread_rwlock_destroy()
rw_rdlock()	pthread_rwlock_rdlock()
rw_wrlock()	pthread_rwlock_wrlock()
rw_unlock()	pthread_rwlock_unlock()
rw_tryrdlock()	pthread_rwlock_tryrdlock()

表 8–2 Solaris 线程和 POSIX pthread 的比较 （续）

Solaris 线程	pthread
rw_trywrlock()	pthread_rwlock_trywrlock()
-	pthread_rwlockattr_init()
-	pthread_rwlockattr_destroy()
-	pthread_rwlockattr_getpshared()
-	pthread_rwlockattr_setpshared()
sema_init()	sem_init() POSIX.1b
sema_destroy()	sem_destroy() POSIX.1b
sema_wait()	sem_wait() POSIX.1b
sema_post()	sem_post() POSIX.1b
sema_trywait()	sem_trywait() POSIX.1b
fork1()	fork()
-	pthread_atfork()
forkall()，多线程副本	-
-	pthread_mutexattr_init()
-	pthread_mutexattr_destroy()
mutex_init() 中的 type 参数	pthread_mutexattr_setpshared()
-	pthread_mutexattr_getpshared()
-	pthread_mutex_attr_settype()
-	pthread_mutex_attr_gettype()
-	pthread_condattr_init()
-	pthread_condattr_destroy()
cond_init() 中的 type 参数	pthread_condattr_setpshared()
-	pthread_condattr_getpshared()
-	pthread_attr_init()
-	pthread_attr_destroy()
thr_create() 中的 THR_BOUND 标志	pthread_attr_setscope()
-	pthread_attr_getscope()
-	pthread_attr_setguardsize()

表 8–2 Solaris 线程和 POSIX pthread 的比较 （续）

Solaris 线程	pthread
-	pthread_attr_getguardsize()
thr_create() 中的 stack_size 参数	pthread_attr_setstacksize()
-	pthread_attr_getstacksize()
thr_create() 中的 stack_addr 参数	pthread_attr_setstack()
-	pthread_attr_getstack()
thr_create() 中的 THR_DETACH 标志	pthread_attr_setdetachstate()
-	pthread_attr_getdetachstate()
-	pthread_attr_setschedparam()
-	pthread_attr_getschedparam()
-	pthread_attr_setinheritsched()
-	pthread_attr_getinheritsched()
-	pthread_attr_setsschedpolicy()
-	pthread_attr_getschedpolicy()

要使用本章中介绍的用于 Solaris 9 和以前发行版的 Solaris 线程函数，必须使用 Solaris 线程库 -lthread 进行链接。

对于 Solaris 线程和 pthread 来说，即使函数名或参数可能会有所不同，但是操作实际上是相同的。此处仅提供了一个简单的示例，其中包括正确的头文件和函数原型。如果没有为 Solaris 线程函数提供返回值，请参见《man pages section 3: Basic Library Functions》中的相应页以获取函数的返回值。

有关 Solaris 相关函数的更多信息，请参见相关 pthread 文档中以类似方式命名的函数。

如果 Solaris 线程函数所提供的功能在 pthread 中不可用，则会提供这些函数的完整说明。

Solaris 线程的独有函数

本节介绍 Solaris 线程的独有函数：用于暂停执行线程和继续执行暂停的线程。

暂停执行线程

thr_suspend(3C) 可用来立即暂停执行 *target_thread* 所指定的线程。如果从 thr_suspend() 成功返回，则将不再执行暂停的线程。

因为 thr_suspend() 在暂停目标线程时不会考虑该线程可能持有的锁，所以，在使用 thr_suspend() 时一定要格外小心。如果要暂停的线程调用的函数需要由已暂停的目标线程拥有的锁，则将产生死锁。

thr_suspend 语法

```
#include <thread.h>



int thr_suspend(thread_t tid);
```

线程暂停之后，以后调用 thr_suspend() 将不起任何作用。信号无法唤醒暂停的线程。线程恢复执行之前，信号将一直保持暂挂状态。

在以下概要中，pthread 中定义的 pthread_t *tid* 与 Solaris 线程中定义的 thread_t *tid* 相同。这两个 *tid* 值可以通过赋值或通过使用强制转换来互换使用。

```
thread_t tid; /* tid from thr_create() */



/* pthreads equivalent of Solaris tid from thread created */

/* with pthread_create() */

pthread_t ptid;



int ret;



ret = thr_suspend(tid);



/* using pthreads ID variable with a cast */

ret = thr_suspend((thread_t) ptid);
```

thr_suspend 返回值

thr_suspend() 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下情况，thr_suspend() 将失败并返回对应的值。

ESRCH
描述: 当前的进程中找不到 *tid*。

继续执行暂停的线程

thr_continue(3C) 可用来恢复执行暂停的线程。继续执行暂停的线程之后，以后调用 thr_continue() 将不起任何作用。

thr_continue 语法

```
#include <thread.h>



int thr_continue(thread_t tid);
```

信号无法唤醒暂停的线程。thr_continue() 继续执行暂停的线程之前，信号将一直保持暂挂状态。

在 pthread 中定义的 pthread_t *tid* 与在 Solaris 线程中定义的 thread_t *tid* 相同。这两个 *tid* 值可以通过赋值或通过使用强制转换来互换使用。

```
thread_t tid; /* tid from thr_create()*/



/* pthreads equivalent of Solaris tid from thread created */

/* with pthread_create()*/

pthread_t ptid;



int ret;



ret = thr_continue(tid);



/* using pthreads ID variable with a cast */

ret = thr_continue((thread_t) ptid)
```

thr_continue 返回值

thr_continue() 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下情况，thr_continue() 将失败并返回对应的值。

ESRCH
描述: 当前的进程中找不到 *tid*。

相似的同步函数－读写锁

读写锁允许多个线程同时进行读取访问，但限制每次只能对一个线程进行写入访问。本节讨论了以下主题：

- 初始化读取器/写入器锁
- 获取读锁
- 尝试获取读锁
- 获取写锁
- 尝试获取写锁
- 解除锁定读取器/写入器锁
- 销毁读取器/写入器锁的状态

一个线程持有读锁时，其他线程也可以获取读锁，但要获取写锁则必须等待。如果一个线程持有写锁或者正在等待获取写锁，则其他线程必须等待才能获取读锁或写锁。

相比互斥锁，读写锁速度较慢。但是，如果许多并发线程读取使用锁保护的数据，但不经常进行写入，则使用读写锁可以提高性能。

使用读写锁可以同步此进程和其他进程中的线程。读写锁是在可以写入并在协作进程之间共享的内存中分配的。有关针对此行为映射读写锁的信息，请参见 mmap(2) 手册页。

缺省情况下，多个线程正在等待读写锁时，获取锁的顺序是不确定的。但是，为了避免写入器资源匮乏，对于优先级相同的写入器和读取器，Solaris 线程软件包中写入器往往优先于读取器。

读写锁在使用之前必须先初始化。

初始化读写锁

使用 rwlock_init(3C) 可以初始化 *rwlp* 所指向的读写锁并将锁的状态设置为未锁定。

rwlock_init 语法

```
#include <synch.h>  （或 #include <thread.h>）


int rwlock_init(rwlock_t *rwlp, int type, void * arg);
```

type 可以是以下值之一：

- USYNC_PROCESS。读写锁可用来同步此进程和其他进程中的线程。*arg* 会被忽略。

- USYNC_THREAD。读写锁可用来仅同步此进程中的线程。*arg* 会被忽略。

同一个读写锁不能同时由多个线程初始化。读写锁还可以通过在清零的内存中进行分配来初始化，在这种情况下假设 type 为 USYNC_THREAD。对于其他线程可能正在使用的读写锁，不得重新初始化。

对于 POSIX 线程，请参见第 134 页中的 “pthread_rwlock_init 语法”。

初始化进程内读写锁

```
#include <thread.h>



rwlock_t rwlp;

int ret;



/* to be used within this process only */

ret = rwlock_init(&rwlp, USYNC_THREAD, 0);
```

初始化进程间读写锁

```
#include <thread.h>



rwlock_t rwlp;

int ret;



/* to be used among all processes */

ret = rwlock_init(&rwlp, USYNC_PROCESS, 0);
```

rwlock_init 返回值

rwlock_init() 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EINVAL
描述: 参数无效。

EFAULT
描述: *rwlp* 或 *arg* 指向的地址非法。

获取读锁

使用 rw_rdlock(3C) 可以获取 *rwlp* 所指向的读写锁中的读锁。

rw_rdlock 语法

```
#include <synch.h> （或 #include <thread.h>）


int rw_rdlock(rwlock_t *rwlp);
```

如果读写锁中的写锁已经锁定，则调用线程将阻塞，直到释放写锁为止。否则，将获取读锁。对于 POSIX 线程，请参见第 134 页中的 “pthread_rwlock_rdlock 语法”。

rw_rdlock 返回值

rw_rdlock() 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EINVAL
描述: 参数无效。

EFAULT
描述: *rwlp* 指向的地址非法。

尝试获取读锁

使用 rw_tryrdlock(3C) 可以尝试获取 *rwlp* 所指向的读写锁中的读锁。

rw_tryrdlock 语法

```
#include <synch.h> （或 #include <thread.h>）


int rw_tryrdlock(rwlock_t *rwlp);
```

如果读写锁中的写锁已经锁定，则 rw_tryrdlock() 将返回错误。否则，将获取读锁。对于 POSIX 线程，请参见第 135 页中的 “pthread_rwlock_tryrdlock 语法”。

rw_tryrdlock返回值

rw_tryrdlock() 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EINVAL
描述: 参数无效。

EFAULT
描述: *rwlp* 指向的地址非法。

EBUSY
描述: *rwlp* 所指向的读写锁已经锁定。

获取写锁

使用 rw_wrlock(3C) 可以获取 *rwlp* 所指向的读写锁中的写锁。

rw_wrlock语法

```
#include <synch.h>  （或 #include <thread.h>）


int rw_wrlock(rwlock_t *rwlp);
```

如果读写锁中的读锁或写锁已经锁定，则调用线程将阻塞，直到释放所有的读锁和写锁为止。读写锁中的写锁一次只能由一个线程持有。对于 POSIX 线程，请参见第 136 页中的“pthread_rwlock_wrlock 语法”。

rw_wrlock返回值

rw_wrlock() 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EINVAL
描述: 参数无效。

EFAULT
描述: *rwlp* 指向的地址非法。

尝试获取写锁

使用 rw_trywrlock(3C) 可以尝试获取 *rwlp* 所指向的读写锁中的写锁。

rw_trywrlock 语法

```
#include <synch.h>  （或 #include <thread.h>）


int rw_trywrlock(rwlock_t *rwlp);
```

如果读写锁上的读锁或写锁已经锁定，则 rw_trywrlock() 将返回错误。对于 POSIX 线程，请参见第 136 页中的 “pthread_rwlock_trywrlock 语法”。

rw_trywrlock 返回值

rw_trywrlock() 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EINVAL
描述: 参数无效。

EFAULT
描述: *rwlp* 指向的地址非法。

EBUSY
描述: *rwlp* 所指向的读写锁已经锁定。

解除锁定读写锁

使用 rw_unlock(3C) 可以解除锁定 *rwlp* 所指向的读写锁。

rw_unlock 语法

```
#include <synch.h>  （或 #include <thread.h>）


int rw_unlock(rwlock_t *rwlp);
```

读写锁必须处于锁定状态，并且调用线程必须持有读锁或写锁。如果还有其他线程正在等待读写锁成为可用，则其中一个线程将被解除阻塞。对于 POSIX 线程，请参见第 137 页中的 “pthread_rwlock_unlock 语法”。

rw_unlock 返回值

rw_unlock() 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EINVAL
描述: 参数无效。

EFAULT
描述: *rwlp* 指向的地址非法。

销毁读写锁的状态

使用 rwlock_destroy(3C) 可以销毁与 *rlwp* 所指向的读写锁相关联的任何状态。

rwlock_destroy 语法

```
#include <synch.h>  （或 #include <thread.h>）


int rwlock_destroy(rwlock_t *rwlp);
```

用来存储读写锁的空间不会释放。对于 POSIX 线程，请参见第 138 页中的“pthread_rwlock_destroy 语法”。

示例 8–1 使用银行帐户来说明读写锁。尽管该程序可能会允许多个线程对帐户余额进行并行只读访问，但是仅允许使用一个写入器。请注意，get_balance() 函数需要使用该锁才能确保以原子方式添加支票帐户余额和储蓄帐户余额。

示例 8–1 读写银行帐户

```
rwlock_t account_lock;

float checking_balance = 100.0;

float saving_balance = 100.0;

...

rwlock_init(&account_lock, 0, NULL);

...



float

get_balance() {

    float bal;
```

示例 8–1 读写银行帐户 （续）

```
    rw_rdlock(&account_lock);
    bal = checking_balance + saving_balance;
    rw_unlock(&account_lock);
    return(bal);
}

void
transfer_checking_to_savings(float amount) {
    rw_wrlock(&account_lock);
    checking_balance = checking_balance - amount;
    saving_balance = saving_balance + amount;
    rw_unlock(&account_lock);
}
```

rwlock_destroy 返回值

rwlock_destroy() 在成功完成之后返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EINVAL
描述: 参数无效。

EFAULT
描述: *rwlp* 指向的地址非法。

相似的 Solaris 线程函数

表 8–3 相似的 Solaris 线程函数

操作	相关函数说明
创建线程	第 193 页中的 “thr_create 语法”
获取最小的栈大小	第 195 页中的 “thr_min_stack 语法”
获取线程标识符	第 196 页中的 “thr_self 语法”
停止执行线程	第 196 页中的 “thr_yield 语法”
向线程发送信号	第 197 页中的 “thr_kill 语法”
访问调用线程的信号掩码	第 197 页中的 “thr_sigsetmask 语法”
终止线程	第 198 页中的 “thr_exit 语法”
等待线程终止	第 198 页中的 “thr_join 语法”
创建线程特定的数据键	第 200 页中的 “thr_keycreate 语法”
设置线程特定数据	第 201 页中的 “thr_setspecific 语法”
获取线程特定数据	第 202 页中的 “thr_getspecific 语法”
设置线程优先级	第 202 页中的 “thr_setprio 语法”
获取线程优先级	第 203 页中的 “thr_getprio 语法”

创建线程

thr_create(3C) 例程是 Solaris 线程接口中最详细的所有例程其中之一。

使用 thr_create(3C) 可以向当前的进程中增加新的受控线程。对于 POSIX 线程，请参见第 23 页中的 “pthread_create 语法”。

thr_create 语法

```
#include <thread.h>


int thr_create(void *stack_base, size_t stack_size,
    void *(*start_routine) (void *), void *arg, long flags,
    thread_t *new_thread);
```

```
size_t thr_min_stack(void);
```

请注意，新线程不会继承暂挂的信号，但确实会继承优先级和信号掩码。

stack_base。包含新线程所使用的栈的地址。如果 *stack_base* 为 NULL，则 thr_create() 会为新线程分配一个至少为 *stack_size* 字节的栈。

stack_size。包含新线程所使用的栈的大小（以字节数表示）。如果 *stack_size* 为零，则使用缺省大小。在大多数情况下，零值最适合。如果 *stack_size* 不为零，则 *stack_size* 必须大于 thr_min_stack() 返回的值。

通常，无需为线程分配栈空间。系统会为每个线程的没有保留交换空间的栈分配 1 MB 的虚拟内存。系统使用 mmap(2) 的 -MAP_NORESERVE 选项来进行分配。

start_routine。包含用以开始执行新线程的函数。start_routine() 返回时，该线程将退出，并且其退出状态会设置为 *start_routine* 返回的值。请参见第 198 页中的 “thr_exit 语法”。

arg。可以是 void 所描述的任何变量，通常是任何大小为 4 字节的值。对于较大的值，必须通过将该参数指向对应的变量来间接传递。

请注意，仅可以提供一个参数。要在程序中采用多个参数，请将多个参数编码为单个参数，如通过将这些参数放在一个结构中。

flags。指定所创建的线程的属性。在大多数情况下，最适合使用零值。

flags 中的值是通过对以下参数执行按位或运算（包含边界值）来构造的：

- THR_SUSPENDED。暂停新线程，并且不执行 *start_routine*，直到 thr_continue() 启动该线程为止。使用 THR_SUSPENDED 可以在运行该线程之前对其执行操作，如更改其优先级。
- THR_DETACHED。分离新线程，以便在该线程终止之后，可以立即重用其线程 ID 和其他资源。如果不想等待线程终止，可以设置 THR_DETACHED。

注 – 如果没有明确分配同步，则未暂停的分离线程会失败。如果失败，则从 thr_create() 返回该线程的创建者之前，会将该线程 ID 重新指定给另一个新线程。

- THR_BOUND。将新线程永久绑定到 LWP。新线程是**绑定线程**。从 Solaris 9 发行版开始，系统不再区分绑定线程和非绑定线程，所有的线程均视为绑定线程。
- THR_DAEMON。将新线程标记为守护进程。守护进程线程始终处于分离状态。THR_DAEMON 表示 THR_DETACHED。所有非守护进程线程退出时，该进程也会随之退出。守护进程线程不会影响进程的退出状态，并且在退出对线程数进行计数时会被忽略。

  进程的退出方法有两种，一是调用 exit()，二是使用进程中不是通过 THR_DAEMON 标志创建的每个线程来调用 thr_exit(3C)。进程所调用的应用程序或库可以创建一个或多个线程，在确定是否退出时应当忽略（不计数）这些线程。THR_DAEMON 标志可标识在进程退出条件中不计数的线程。

new_thread。如果 *new_thread* 不为 NULL，则它将指向 thr_create() 成功时用来存储新线程 ID 的位置。调用方负责提供该参数所指向的存储空间。线程 ID 仅在调用进程中有效。

如果对该标识符不感兴趣，请向 *new_thread* 提供一个 NULL 值。

thr_create 返回值

thr_create() 函数在成功完成之后返回零。其他任何返回值都表示出现了错误。如果检测到以下情况之一，thr_create() 将失败并返回对应的值。

EAGAIN
描述: 超出了系统限制，如创建的 LWP 过多。

ENOMEM
描述: 可用内存不足，无法创建新线程。

EINVAL
描述: *stack_base* 不为 NULL，并且 *stack_size* 小于 thr_min_stack() 返回的值。

获取最小栈大小

使用 thr_min_stack(3C) 可以获取线程的最小栈大小。

Solaris 线程中的栈行为通常与 pthread 中的栈行为相同。有关设置和操作栈的更多信息，请参见第 67 页中的 “关于栈”。

thr_min_stack 语法

```
#include <thread.h>



size_t thr_min_stack(void);
```

thr_min_stack() 会返回执行空线程所需的空间量。创建空线程的目的是执行空过程。由于有用线程需要的栈要大于绝对最小栈，因此在减小栈大小时请务必小心。

执行非空过程的线程所分配的栈大小应大于 thr_min_stack() 的大小。

如果某个线程是借助于用户提供的栈创建的，则用户必须保留足够的空间才能运行该线程。动态链接的执行环境会增加确定线程最小栈要求的难度。

可以通过两种方法来指定自定义栈。第一种方法是为栈位置提供 NULL，从而要求运行时库为该栈分配空间，但是向 thr_create() 提供 stacksize 参数中所需的大小。

另一种方法是全面负责栈管理的各个方面，并向 thr_create() 提供一个指向该栈的指针。这意味着不但需要负责分配栈，还需要负责取消分配栈。线程终止时，必须安排对该线程的栈进行处理。

当您分配自己的栈时，请确保通过调用 mprotect(2) 在该栈末尾附加一个红色区域。

大多数用户都不应当通过用户提供的栈来创建线程。用户提供的栈之所以存在，只是为了支持要求对其执行环境进行完全控制的应用程序。

相反，用户应当由系统来管理对栈的分配。系统提供的缺省栈应当能够满足所创建的任何线程的要求。

thr_min_stack 返回值

未定义任何错误。

获取线程标识符

使用 thr_self(3C) 可以获取调用线程的 ID。对于 POSIX 线程，请参见第 35 页中的“pthread_self 语法”。

thr_self 语法

```
#include <thread.h>



thread_t thr_self(void);
```

thr_self 返回值

未定义任何错误。

停止执行线程

thr_yield(3C) 会导致当前的线程停止执行，以便执行另一个具有相同或更高优先级的线程。否则，thr_yield() 不起任何作用。但是，调用 thr_yield() 无法保证当前的线程会停止执行。

thr_yield 语法

```
#include <thread.h>



void thr_yield(void);
```

thr_yield 返回值

thr_yield() 不会返回任何内容并且不会设置 errno。

向线程发送信号

thr_kill(3C) 可用来向线程发送信号。对于 POSIX 线程，请参见第 35 页中的“pthread_self 语法”。

thr_kill 语法

```
#include <thread.h>

#include <signal.h>

int thr_kill(thread_t target_thread, int sig);
```

thr_kill 返回值

如果成功完成，thr_kill() 将返回 0。如果检测到以下任一情况，thr_kill() 将失败并返回对应的值。如果失败，则不发送任何信号。

ESRCH
描述: 未找到与 *thread* ID 所指定的线程相关联的线程。

EINVAL
描述: *sig* 参数值不为零。*sig* 无效或者是不支持的信号编号。

访问调用线程的信号掩码

使用 thr_sigsetmask (3C) 可以更改或检查调用线程的信号掩码。

thr_sigsetmask 语法

```
#include <thread.h>

#include <signal.h>

int thr_sigsetmask(int how, const sigset_t *set, sigset_t *oset);
```

thr_sigsetmask() 可用来更改或检查调用线程的信号掩码。每个线程都有自己的信号掩码。新线程会继承调用线程的信号掩码和优先级，但不会继承暂挂的信号。新线程的暂挂信号将为空。

如果参数 *set* 的值不为 NULL，则 *set* 会指向一组信号，这组信号可用来修改当前阻塞的信号组。如果 *set* 的值为 NULL，则 *how* 的值没有意义，并且不会修改线程的信号掩码。使用此行为可了解有关当前阻塞的信号的情况。

how 的值可指定更改信号组的方法。*how* 可以为以下值之一。

- SIG_BLOCK。*set* 对应于一组要阻塞的信号。这些信号会添加到当前的信号掩码中。

- SIG_UNBLOCK。*set* 对应于一组要解除阻塞的信号。这些信号会从当前的信号掩码中删除。
- SIG_SETMASK。*set* 对应于新的信号掩码。当前的信号掩码将替换为 *set*。

thr_sigsetmask 返回值

如果成功完成，thr_sigsetmask() 将返回 0。如果检测到以下任一情况，thr_sigsetmask() 将失败并返回对应的值。

EINVAL
描述: *set* 不为 NULL，并且 *how* 的值未定义。

终止线程

使用 thr_exit(3C) 可以终止线程。对于 POSIX 线程，请参见第 42 页中的 “pthread_exit 语法”。

thr_exit 语法

```
#include <thread.h>


void thr_exit(void *status);
```

thr_exit 返回值

thr_exit() 不返回到其调用方。

等待线程终止

使用 thr_join(3C) 可以等待目标线程终止。对于 POSIX 线程，请参见第 25 页中的 “pthread_join 语法”。

thr_join 语法

```
#include <thread.h>


int thr_join(thread_t tid, thread_t *departedid, void **status);
```

目标线程必须是当前进程的成员，而不能是分离线程或守护进程线程。

多个线程不能等待同一个线程完成，否则仅有一个线程会成功完成。其他线程将终止，并返回 ESRCH 错误。

如果目标线程已经终止，thr_join() 将不会阻塞对调用线程的处理。

thr_join，加入特定线程

```
#include <thread.h>


thread_t tid;

thread_t departedid;

int ret;

void *status;


/* waiting to join thread "tid" with status */

ret = thr_join(tid, &departedid, &status);


/* waiting to join thread "tid" without status */

ret = thr_join(tid, &departedid, NULL);


/* waiting to join thread "tid" without return id and status */

ret = thr_join(tid, NULL, NULL);
```

如果 *tid* 为 (thread_t)0，则 thread_join() 将等待进程中的任何非分离线程终止。换句话说，如果未指定线程标识符，则任何未分离的线程退出都将导致返回 thread_join()。

thr_join，加入任何线程

```
#include <thread.h>


thread_t tid;

thread_t departedid;

int ret;
```

```
void *status;


/* waiting to join any non-detached thread with status */

ret = thr_join(0, &departedid, &status);
```

通过在 Solaris `thr_join()` 中使用 0 来表示线程 ID，进程中的任何非分离线程退出时都将执行加入操作。*departedid* 表示现有线程的线程 ID。

thr_join 返回值

`thr_join()` 在成功运行后返回 0。如果检测到以下任一情况，`thr_join()` 将失败并返回对应的值。

ESRCH
描述: 未找到与目标线程 ID 对应的非分离线程。

EDEADLK
描述: 检测到死锁，或者目标线程的值指定了调用线程。

创建线程特定的数据键

使用 `thr_keycreate`(3C) 可分配键，用于标识进程中线程特定数据。键可全局应用于进程中的所有线程。创建键时，每个线程都会将一个值与其绑定。

除了函数的名称和参数以外，Solaris 线程的线程特定数据与 POSIX 线程的线程特定数据完全相同。本节概述了 Solaris 函数。 对于 POSIX 线程，请参见第 29 页中的“pthread_key_create 语法”。

thr_keycreate 语法

```
#include <thread.h>


int thr_keycreate(thread_key_t *keyp,
    void (*destructor) (void *value));
```

keyp 为每个绑定线程单独维护特定的值。所有的线程最初都会绑定到专用元素 *keyp*，该元素可用于访问其线程特定数据。创建键时，对于所有活动线程，将为新键赋予值 NULL。此外在创建线程时，还会为以前在新线程中创建的所有键赋予值 NULL。

destructor 函数是可选的，可以将其与每个 *keyp* 相关联。线程退出时，如果 *keyp* 具有非 NULL 的 *destructor*，并且线程具有与 *keyp* 相关联的非 NULL *value*，则 *destructor* 将用当前的关联 *value* 进行调用。如果线程退出时存在多个 *destructor* 与其相关，则 destructor 的调用顺序是不确定的。

thr_keycreate 返回值

`thr_keycreate()` 在成功运行后返回 0。如果检测到以下任一情况，`thr_keycreate()` 将失败并返回对应的值。

`EAGAIN`
描述: 系统资源不足，无法创建另一个线程特定的数据键，或者键数目超过了 PTHREAD_KEYS_MAX 的每进程限制。

`ENOMEM`
描述: 可用内存不足，无法将 *value* 与 *keyp* 相关联。

设置线程特定的数据值

`thr_setspecific`(3C) 可用来将 *value* 绑定到线程特定的数据键（对于调用线程来说为 *key*）。对于 POSIX 线程，请参见第 31 页中的 “pthread_setspecific 语法”。

thr_setspecific 语法

```
#include <thread.h>



int thr_setspecific(thread_key_t key, void *value);
```

thr_setspecific 返回值

`thr_setspecific()` 在成功运行后返回 0。如果检测到以下任一情况，`thr_setspecific()` 将失败并返回对应的值。

`ENOMEM`
描述: 可用内存不足，无法将 *value* 与 *keyp* 相关联。

`EINVAL`
描述: *keyp* 无效。

获取线程特定的数据值

`thr_getspecific`(3C) 可用来将当前绑定到调用线程的 *key* 的值存储到 *valuep* 所指向的位置。对于 POSIX 线程，请参见第 31 页中的 “pthread_getspecific 语法”。

thr_getspecific 语法

```
#include <thread.h>


int thr_getspecific(thread_key_t key, void **valuep);
```

thr_getspecific 返回值

thr_getspecific() 在成功运行后返回 0。如果检测到以下任一情况，thr_getspecific() 将失败并返回对应的值。

ENOMEM
描述: 可用内存不足，无法将 *value* 与 *keyp* 相关联。

EINVAL
描述: *keyp* 无效。

设置线程的优先级

在 Solaris 线程中，对于在创建时与父线程具有不同优先级的线程，可以在 SUSPEND 模式下创建。在暂停之后，可以通过使用 thr_setprio(3C) 函数调用来修改线程的优先级。thr_setprio() 完成之后，线程可恢复执行。

争用同步对象时，高优先级的线程优先于低优先级的线程。

thr_setprio 语法

thr_setprio(3C) 可用来将当前进程内 *tid* 所指定线程的优先级更改为 *newprio* 所指定的优先级。对于 POSIX 线程，请参见第 38 页中的 “pthread_setschedparam 语法” 。

```
#include <thread.h>


int thr_setprio(thread_t tid, int newprio)
```

缺省情况下，线程是基于范围从 0（最不重要）到 127（最重要）的固定优先级来调度的。

```
thread_t tid;

int ret;

int newprio = 20;
```

```
/* suspended thread creation */

ret = thr_create(NULL, NULL, func, arg, THR_SUSPENDED, &tid);



/* set the new priority of suspended child thread */

ret = thr_setprio(tid, newprio);



/* suspended child thread starts executing with new priority */

ret = thr_continue(tid);
```

thr_setprio 返回值

thr_setprio() 在成功运行后返回 0。如果检测到以下任一情况，thr_setprio() 将失败并返回对应的值。

ESRCH
描述: *tid* 指定的值不引用现有的线程。

EINVAL
描述: *priority* 的值对于与 *tid* 相关联的调度类没有意义。

获取线程的优先级

使用 thr_getprio(3C) 可以获取线程的当前优先级。每个线程都从其创建者继承优先级。thr_getprio() 会将当前的优先级 *tid* 存储到 *newprio* 所指向的位置。对于 POSIX 线程，请参见第 39 页中的 “pthread_getschedparam 语法”。

thr_getprio 语法

```
#include <thread.h>



int thr_getprio(thread_t tid, int *newprio)
```

thr_getprio 返回值

thr_getprio() 在成功运行后返回 0。如果检测到以下情况，thr_getprio() 将失败并返回对应的值。

ESRCH
描述: *tid* 指定的值不会引用现有的线程。

相似的同步函数－互斥锁

- 初始化互斥锁
- 销毁互斥锁
- 获取互斥锁
- 释放互斥锁
- 尝试获取互斥锁

初始化互斥锁

使用 mutex_init(3C) 可以初始化 *mp* 所指向的互斥锁。对于 POSIX 线程，请参见第 89 页中的 “初始化互斥锁” 。

mutex_init(3C) 语法

```
#include <synch.h>

#include <thread.h>



int mutex_init(mutex_t *mp, int type, void *arg));
```

type 可以是以下值之一：

- USYNC_PROCESS。互斥锁可用来同步此进程和其他进程中的线程。*arg* 会被忽略。
- USYNC_PROCESS_ROBUST。可使用互斥来在此进程和其他进程中**可靠地**同步线程。*arg* 会被忽略。
- USYNC_THREAD。互斥锁可用来仅同步此进程中的线程。*arg* 会被忽略。

如果进程在持有 USYNC_PROCESS 锁时失败，该锁以后的请求者都将挂起。对于与客户机进程共享锁的系统，此行为会产生问题，因为客户机进程会异常中止。为了避免出现停用进程所持有的锁定挂起的问题，请使用 USYNC_PROCESS_ROBUST 来锁定互斥锁。
USYNC_PROCESS_ROBUST 增加了两个功能：

- 如果进程停用，则系统将解除锁定该进程所持有的全部锁。
- 请求失败进程所拥有的任何锁的下一个请求者将获取该锁。但是，拥有该锁时会返回一个错误，指明以前的属主在拥有该锁时失败。

互斥锁还可以通过在清零的内存中进行分配来初始化，在这种情况下假定 *type* 为 USYNC_THREAD。

多个线程决不能同时初始化同一个互斥锁。如果其他线程正在使用互斥锁，则不得将该互斥锁重新初始化。

进程内的互斥锁

```
#include <thread.h>



mutex_t mp;

int ret;



/* to be used within this process only */

ret = mutex_init(&mp, USYNC_THREAD, 0);
```

进程间的互斥锁

```
#include <thread.h>



mutex_t mp;

int ret;



/* to be used among all processes */

ret = mutex_init(&mp, USYNC_PROCESS, 0);
```

强健的进程间互斥

```
#include <thread.h>



mutex_t mp;

int ret;
```

```
/* to be used among all processes */

ret = mutex_init(&mp, USYNC_PROCESS_ROBUST, 0);
```

mutex_init 返回值

mutex_init() 在成功运行后返回 0。如果检测到以下任一情况，mutex_init() 将失败并返回对应的值。

EFAULT
描述: *mp* 指向的地址非法。

EINVAL
描述: *mp* 指定的值无效。

ENOMEM
描述: 系统内存不足，无法初始化互斥锁。

EAGAIN
描述: 系统资源不足，无法初始化互斥锁。

EBUSY
描述: 系统检测到重新初始化活动互斥锁的尝试。

销毁互斥锁

使用 mutex_destroy(3C) 可以销毁与 *mp* 所指向的互斥锁相关联的任何状态。用来存储该互斥锁的空间不会释放。对于 POSIX 线程，请参见第 95 页中的 “pthread_mutex_destroy 语法”。

mutex_destroy 语法

```
#include <thread.h>



int mutex_destroy (mutex_t *mp);
```

mutex_destroy 返回值

mutex_destroy() 在成功运行后返回 0。如果检测到以下情况，mutex_destroy() 将失败并返回对应的值。

EFAULT
描述: *mp* 指向的地址非法。

获取互斥锁

使用 mutex_lock(3C) 可以锁定 *mp* 所指向的互斥锁。如果该互斥锁已经锁定，调用线程将会阻塞，直到该互斥锁成为可用为止。调用线程会在具有优先级的队列中等待。对于 POSIX 线程，请参见第 91 页中的 “pthread_mutex_lock 语法”。

mutex_lock 语法

```
#include <thread.h>



int mutex_lock(mutex_t *mp);
```

mutex_lock 返回值

mutex_lock() 在成功运行后返回 0。如果检测到以下任一情况，mutex_lock() 将失败并返回对应的值。

EFAULT
描述: *mp* 指向的地址非法。

EDEADLK
描述: 互斥锁已经锁定并且由调用线程拥有。

释放互斥锁

使用 mutex_unlock(3C) 可以解除锁定 *mp* 所指向的互斥锁。该互斥锁必须锁定。调用线程必须是最后一个锁定该互斥锁的线程，即该互斥锁的属主。对于 POSIX 线程，请参见第 93 页中的 “pthread_mutex_unlock 语法”。

mutex_unlock 语法

```
#include <thread.h>



int mutex_unlock(mutex_t *mp);
```

mutex_unlock 返回值

mutex_unlock() 在成功运行后返回 0。如果检测到以下任一情况，mutex_unlock() 将失败并返回对应的值。

EFAULT
描述: *mp* 指向的地址非法。

EPERM
描述: 调用线程不拥有该互斥锁。

尝试获取互斥锁

使用 mutex_trylock(3C) 可以尝试锁定 *mp* 所指向的互斥锁。此函数是 mutex_lock() 的非阻塞版本。对于 POSIX 线程，请参见第 94 页中的 “pthread_mutex_trylock 语法”。

mutex_trylock 语法

```
#include <thread.h>



int mutex_trylock(mutex_t *mp);
```

mutex_trylock 返回值

mutex_trylock() 在成功运行后返回 0。如果检测到以下任一情况，mutex_trylock() 将失败并返回对应的值。

EFAULT
描述: *mp* 指向的地址非法。

EBUSY
描述: 系统检测到重新初始化活动互斥锁的尝试。

相似的同步函数：条件变量

- 初始化条件变量
- 销毁条件变量
- 等待条件
- 等待绝对时间
- 等待时间间隔
- 解除阻塞一个线程
- 解除阻塞所有线程

初始化条件变量

使用 cond_init(3C) 可以初始化 *cv* 所指向的条件变量。

cond_init 语法

```
#include <thread.h>


int cond_init(cond_t *cv, int type, int arg);
```

type 可以是以下值之一：

- USYNC_PROCESS。条件变量可用来同步此进程和其他进程中的线程。*arg* 会被忽略。
- USYNC_THREAD。条件变量可用来仅同步此进程中的线程。*arg* 会被忽略。

条件变量还可以通过在清零的内存中进行分配来初始化，在这种情况下假设 type 为 USYNC_THREAD。

多个线程决不能同时初始化同一个条件变量。对于其他线程可能正在使用的条件变量，不得重新初始化。

对于 POSIX 线程，请参见第 103 页中的 “pthread_condattr_init 语法”。

进程内条件变量

```
#include <thread.h>


cond_t cv;

int ret;


/* to be used within this process only */

ret = cond_init(cv, USYNC_THREAD, 0);
```

进程间条件变量

```
#include <thread.h>


cond_t cv;

int ret;
```

```
/* to be used among all processes */

ret = cond_init(&cv, USYNC_PROCESS, 0);
```

cond_init 返回值

cond_init() 在成功运行后返回 0。如果检测到以下任一情况，cond_init() 将失败并返回对应的值。

EFAULT
描述: *cv* 指向的地址非法。

EINVAL
描述: *type* 不是可以识别的类型。

销毁条件变量

使用 cond_destroy(3C) 可以销毁与 *cv* 所指向的条件变量相关联的状态。用来存储该条件变量的空间不会释放。对于 POSIX 线程，请参见第 104 页中的 “pthread_condattr_destroy 语法” 。

cond_destroy 语法

```
#include <thread.h>



int cond_destroy(cond_t *cv);
```

cond_destroy 返回值

cond_destroy() 在成功运行后返回 0。如果检测到以下任一情况，cond_destroy() 将失败并返回对应的值。

EFAULT
描述: *cv* 指向的地址非法。

EBUSY
描述: 系统检测到销毁活动条件变量的尝试。

等待条件

使用 cond_wait(3C) 可以原子方式释放 *mp* 所指向的互斥锁，并导致调用线程基于 *cv* 所指向的条件变量阻塞。阻塞的线程可以由 cond_signal() 或 cond_broadcast() 唤醒，也可以在信号传送或 fork() 将其中断时唤醒。

cond_wait() 每次返回时，互斥锁均处于锁定状态并由调用线程拥有，即使返回错误时也是如此。

cond_wait 语法

```
#include <thread.h>


int cond_wait(cond_t *cv, mutex_t *mp);
```

cond_wait 返回值

cond_wait() 在成功运行后返回 0。如果检测到以下任一情况，cond_wait() 将失败并返回对应的值。

EFAULT
描述: *cv* 指向的地址非法。

EBUSY
描述: 等待过程已被信号或 fork() 中断。

等待绝对时间

cond_timedwait(3C) 与 cond_wait() 非常相似，区别在于 cond_timedwait() 经过 *abstime* 指定的时间之后不会阻塞。对于 POSIX 线程，请参见第 111 页中的 “pthread_cond_timedwait 语法”。

cond_timedwait 语法

```
#include <thread.h>


int cond_timedwait(cond_t *cv, mutex_t *mp, timestruct_t abstime);
```

cond_timedwait() 每次返回时，互斥锁均会锁定并由调用线程拥有，即使返回错误时也是如此。

cond_timedwait() 函数会一直阻塞，直到该条件获得信号，或者经过最后一个参数所指定的时间为止。超时以具体的时间指定，这样即可在不重新计算超时值的情况下高效地重新测试条件。

cond_timedwait 返回值

cond_timedwait() 在成功运行后返回 0。如果检测到以下任一情况，cond_timedwait() 将失败并返回对应的值。

EFAULT
描述: *cv* 指向的地址非法。

ETIME
描述: 由 *abstime* 指定的时间已过期。

EINVAL
描述: *abstime* 无效。

等待时间间隔

cond_reltimedwait(3C) 与 cond_timedwait() 非常相似，区别在于第三个参数的值不同。cond_reltimedwait() 的第三个参数采用相对时间间隔值，而不是绝对时间值。对于 POSIX 线程，请参见 pthread_cond_reltimedwait_np(3C) 手册页。

cond_reltimedwait() 每次返回时，互斥锁均会锁定并由调用线程拥有，即使返回错误时也是如此。cond_reltimedwait() 函数一直阻塞，直到该条件获得信号，或者经过最后一个参数所指定的时间间隔为止。

cond_reltimedwait 语法

```
#include <thread.h>



int cond_reltimedwait(cond_t *cv, mutex_t *mp,

    timestruct_t reltime);
```

cond_reltimedwait 返回值

cond_reltimedwait() 在成功运行后返回 0。如果检测到以下任一情况，cond_reltimedwait() 将失败并返回对应的值。

EFAULT
描述: *cv* 指向的地址非法。

ETIME
描述: 由 *reltime* 指定的时间已过期。

解除阻塞一个线程

对于基于 *cv* 所指向的条件变量阻塞的线程，使用 cond_signal(3C) 可以解除阻塞该线程。如果没有线程基于该条件变量阻塞，则调用 cond_signal() 不起任何作用。

cond_signal 语法

```
#include <thread.h>



int cond_signal(cond_t *cv);
```

cond_signal 返回值

cond_signal() 在成功运行后返回 0。如果检测到以下情况，cond_signal() 将失败并返回对应的值。

EFAULT
描述: *cv* 指向的地址非法。

解除阻塞所有线程

对于基于 *cv* 所指向的条件变量阻塞的全部线程，使用 cond_broadcast(3C) 可以解除阻塞这些线程。如果没有线程基于该条件变量阻塞，则调用 cond_broadcast() 不起任何作用。

cond_broadcast 语法

```
#include <thread.h>



int cond_broadcast(cond_t *cv);
```

cond_broadcast 返回值

cond_broadcast() 在成功运行后返回 0。如果检测到以下情况，cond_broadcast() 将失败并返回对应的值。

EFAULT
描述: *cv* 指向的地址非法。

相似的同步函数：信号

信号操作在 Solaris 操作环境和 POSIX 环境中均相同。 Solaris 操作环境中的函数名 sema_ 在 pthread 中会更改为 sem_。本节讨论了以下主题：

- 初始化信号
- 增加信号
- 基于信号计数阻塞
- 减小信号计数
- 销毁信号状态

初始化信号

使用 sema_init(3C) 可以通过 *count* 值来初始化 *sp* 所指向的信号变量。

sema_init 语法

```
#include <thread.h>



int sema_init(sema_t *sp, unsigned int count, int type,

    void *arg);
```

type 可以是以下值之一：

- USYNC_PROCESS。信号可用来同步此进程和其他进程中的线程。信号只能由一个进程来初始化。*arg* 会被忽略。
- USYNC_THREAD。信号可用来仅同步此进程中的线程。*arg* 会被忽略。

多个线程决不能同时初始化同一个信号。不得对其他线程正在使用的信号重新初始化。

进程内信号

```
#include <thread.h>



sema_t sp;

int ret;

int count;

count = 4;
```

```
/* to be used within this process only */

ret = sema_init(&sp, count, USYNC_THREAD, 0);
```

进程间信号

```
#include <thread.h>



sema_t sp;

int ret;

int count;

count = 4;



/* to be used among all the processes */

ret = sema_init (&sp, count, USYNC_PROCESS, 0);
```

sema_init 返回值

sema_init() 在成功运行后返回 0。如果检测到以下任一情况，sema_init() 将失败并返回对应的值。

EINVAL
描述: *sp* 引用的信号无效。

EFAULT
描述: *sp* 或 *arg* 指向的地址非法。

增加信号

使用 sema_post(3C) 可以原子方式增加 *sp* 所指向的信号。如果多个线程基于该信号阻塞，则系统会解除阻塞其中一个线程。

sema_post 语法

```
#include <thread.h>
```

```
int sema_post(sema_t *sp);
```

sema_post 返回值

sema_post() 在成功运行后返回 0。如果检测到以下任一情况，sema_post() 将失败并返回对应的值。

EINVAL
描述: *sp* 引用的信号无效。

EFAULT
描述: *sp* 指向的地址非法。

EOVERFLOW
描述: *sp* 指向的信号值超过了 SEM_VALUE_MAX。

基于信号计数阻塞

使用 sema_wait(3C) 可以一直阻塞调用线程，直到 *sp* 所指向的信号的计数变得大于零为止。计数变得大于零时，系统会以原子方式减小计数。

sema_wait 语法

```
#include <thread.h>


int sema_wait(sema_t *sp);
```

sema_wait 返回值

sema_wait() 在成功运行后返回 0。如果检测到以下任一情况，sema_wait() 将失败并返回对应的值。

EINVAL
描述: *sp* 引用的信号无效。

EINTR
描述: 等待过程已被信号或 fork() 中断。

减小信号计数

使用 sema_trywait(3C) 可以在计数大于零时，以原子方式减小 *sp* 所指向的信号的计数。此函数是 sema_wait() 的非阻塞版本。

sema_trywait 语法

```
#include <thread.h>


int sema_trywait(sema_t *sp);
```

sema_trywait 返回值

sema_trywait() 在成功运行后返回 0。如果检测到以下任一情况，sema_trywait() 将失败并返回对应的值。

EINVAL
描述: *sp* 指向的信号无效。

EBUSY
描述: *sp* 所指向的信号的计数为零。

销毁信号状态

使用 sema_destroy(3C) 可以销毁与 *sp* 所指向的信号相关联的任何状态。不会释放用来存储该信号的空间。

sema_destroy(3C) 语法

```
#include <thread.h>


int sema_destroy(sema_t *sp);
```

sema_destroy(3C) 返回值

sema_destroy() 在成功运行后返回 0。如果检测到以下情况，sema_destroy() 将失败并返回对应的值。

EINVAL
描述: *sp* 指向的信号无效。

跨进程边界同步

每个同步元语都可以设置为跨进程边界使用。可通过以下方法来设置跨边界同步：确保同步变量位于共享内存段中，并在 type 设置为 USYNC_PROCESS 的情况下调用相应的 init 例程。

如果 type 设置为 USYNC_PROCESS，则针对同步变量执行的操作与 *type* 为 USYNC_THREAD 时针对变量执行的操作相同。

```
mutex_init(&m, USYNC_PROCESS, 0);

rwlock_init(&rw, USYNC_PROCESS, 0);

cond_init(&cv, USYNC_PROCESS, 0);

sema_init(&s, count, USYNC_PROCESS, 0);
```

生成方和使用者问题示例

示例 8–2 说明了生成方和使用者位于不同进程时的生成方和使用者问题。主例程将与其子进程共享的全零内存段映射到其地址空间。请注意，必须调用 mutex_init() 和 cond_init()，因为同步变量的 type 为 USYNC_PROCESS。

创建子进程是为了运行使用者，父进程则运行生成方。

此示例还说明了生成方和使用者的驱动程序。producer_driver 从 stdin 读取字符并调用 producer。consumer_driver 通过调用 consumer 来获取字符并将这些字符写入 stdout 中。

示例 8–2 的数据结构与用于解析条件变量的数据结构相同。请参见第 99 页中的 “嵌套锁定和单链接列表的结合使用示例”。

示例 8–2 使用 USYNC_PROCESS 时的生成方和使用者问题

```
main() {

    int zfd;

    buffer_t *buffer;



    zfd = open(“/dev/zero”, O_RDWR);

    buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),

        PROT_READ|PROT_WRITE, MAP_SHARED, zfd, 0);
```

示例 8–2 使用 USYNC_PROCESS 时的生成方和使用者问题 （续）

```
    buffer->occupied = buffer->nextin = buffer->nextout = 0;


    mutex_init(&buffer->lock, USYNC_PROCESS, 0);
    cond_init(&buffer->less, USYNC_PROCESS, 0);
    cond_init(&buffer->more, USYNC_PROCESS, 0);
    if (fork() == 0)
        consumer_driver(buffer);
    else
        producer_driver(buffer);
}


void producer_driver(buffer_t *b) {
    int item;


    while (1) {
        item = getchar();
        if (item == EOF) {
            producer(b, '\0');
            break;
        } else
            producer(b, (char)item);
    }
}
```

示例 8–2 使用 USYNC_PROCESS 时的生成方和使用者问题 （续）

```
void consumer_driver(buffer_t *b) {
    char item;

    while (1) {
        if ((item = consumer(b)) == '\0')
            break;
        putchar(item);
    }
}
```

创建子进程是为了运行使用者，父进程则运行生成方。

fork() 和 Solaris 线程的特殊问题

在 Solaris 10 发行版之前，Solaris 线程和 POSIX 线程以不同的方式定义 fork() 的行为。有关 fork() 问题的详细讨论，请参见第 147 页中的 “进程创建：exec 和 exit 问题”。

Solaris libthread 同时支持 fork() 和 fork1()。fork() 调用具有 "Fork-All" 语义。fork() 可用来复制进程中的所有内容（包括线程和 LWP），从而创建父进程的准确克隆。fork1() 调用所创建的克隆中仅有一个线程，它可复制进程状态和地址空间，但是仅克隆调用线程。

POSIX libpthread 仅支持 fork()，该函数与 Solaris 线程中的 fork1() 具有相同语义。

fork() 具有 "Fork-All" 语义还是 "Fork-One"语义取决于所使用的库。使用 -lthread 进行链接可以赋予 fork() "Fork-All" 语义；使用 -lpthread 进行链接可以赋予 fork() "Fork-One"语义。

从 Solaris 10 发行版开始，fork() 在 Solaris 线程和 POSIX 线程中具有相同的语义。具体来说，fork1() 语义仅复制调用方。对于需要使用 “复制全部” 语义的应用程序，提供了一个新函数 forkall()。

有关更多详细信息，请参见第 173 页中的 “使用 libthread 或 libpthread 链接”。

◆ ◆ ◆ 第 9 章

9 编程原则

本章提供有关使用线程进行编程的一些建议。大多数建议同时适用于 Solaris 和 POSIX 线程，但效用会有所不同，需要注意其行为。本章着重讲解从单线程思维到多线程思维的转变。本章讨论以下主题：

重新考虑全局变量

以前，大多数代码都是为单线程程序设计的。此代码设计特别适合于大多数从 C 程序调用的库例程。对于单线程代码，进行了以下隐含假设：

- 写入全局变量，随后又从该变量中读取时，读取的内容就是写入的内容。
- 写入非全局静态存储，随后又从变量中读取时，所读取的内容恰好就是写入的内容。
- 不需要进行同步，因为不会调用对变量的并发访问。

以下示例论述了由于这些假设而在多线程程序中引发的一些问题，以及如何处理这些问题。

传统的单线程 C 和 UNIX 通常会处理在系统调用中检测到的错误。系统调用可将任何内容作为函数值返回。例如，`write()` 返回已传输的字节数。但是，会保留值 `-1` 以表明出现了错误。因此，当系统调用返回 `-1` 时，即表明调用失败。

示例 9–1 全局变量和 *errno*

```
extern int errno;
```

示例 9–1 全局变量和 *errno* （续）

```
...

if (write(file_desc, buffer, size) == -1) {

    /* the system call failed */

    fprintf(stderr, “something went wrong, “

        “error code = %d\n”, errno);

    exit(1);

}

...
```

错误代码将被置于全局变量 errno 中，而不是返回可能与正常返回值混淆的实际错误代码。系统调用失败时，可以查看 errno 以了解错误所在。

现在，请考虑在多线程环境中，当两个线程几乎同时失败而出现的错误不同时会发生的情况。两个线程都期望在 errno 中找到其错误代码，但 errno 的一个副本不能同时包含两个值。此全局变量方法不适用于多线程程序。

线程可通过概念上的新存储类来解决此问题：线程特定数据。此类存储似于全局存储。可从正在运行的线程的任何过程访问线程特定数据。但是，线程特定数据专门用于线程。当两个线程引用同名称的线程特定数据位置时，这些线程引用的是两个不同的存储区域。

因此，使用线程时，对 errno 的每个引用都是特定于线程的，因为每个线程都具有专用的 errno 副本。在此实现方案中，通过使 errno 成为可扩展到函数调用的宏来实现特定于线程的 errno 调用。

提供静态局部变量

示例 9–2 说明了与 errno 问题类似的问题，但涉及的是静态存储，而不是全局存储。函数 gethostbyname(3NSL) 是将计算机名称作为其参数进行调用的。返回值是一个指向结构的指针，该结构包含通过网络通信联系计算机所需的信息。

示例 9–2 gethostbyname() 问题

```
struct hostent *gethostbyname(char *name) {

    static struct hostent result;

        /* Lookup name in hosts database */
```

示例 9–2 gethostbyname() 问题 （续）

```
        /* Put answer in result */

    return(&result);

}
```

通常情况下，使用返回到局部变量的指针不是一个好办法。在本示例中使用指针有效，是因为变量是静态的。但是，当两个线程同时使用不同的计算机名称调用此变量时，使用静态存储会发生冲突。

与 errno 问题一样，可以使用线程特定数据来替换静态存储。但是，此替换涉及到动态分配存储，并且会增加调用开支。

处理该问题的更好方法是使 gethostbyname() 的调用方为调用结果提供存储。调用方可通过例程的其他输出参数来提供存储。其他输出参数需要 gethostbyname() 函数的新接口。

在线程中常使用此技术来解决许多问题。在大多数情况下，新接口的名称就是原有名称附加 "_r"，如 gethostbyname_r(3NSL)。

同步线程

共享数据和进程资源时，应用程序中的线程必须彼此协作并进行同步。

多个线程调用处理同一对象的函数时，会引发问题。在单线程环境中，同步对这类对象的访问不是问题。但是，如示例 9–3 所示，同步对于多线程代码是个问题。请注意，对于多线程程序，可以安全调用 printf(3S) 函数。本示例说明当 printf() 不安全时可能会发生的情况。

示例 9–3 printf() 问题

```
/* thread 1: */

    printf("go to statement reached");




/* thread 2: */

    printf("hello world");
```

示例 9–3 printf() 问题　（续）

```
printed on display:

    go to hello
```

单线程策略

一个策略是，只要应用程序中的任何线程处于运行状态并在必须阻塞之前被释放，即可获取单个应用程序范围的互斥锁。由于无论何时都只能有一个线程可以访问共享数据，因此每个线程都有一致的内存视图。

由于此策略仅对单线程非常有效，因此此策略的使用范围非常小。

可重复执行函数

更好的方法是利用模块化和数据封装的原理。可重复执行函数可以在同时被多个线程调用的情况下正确执行。要编写可重复执行函数，需要大致了解**正确操作**对此特定函数的意义。

必须使被多个线程调用的函数可重复执行。要使函数可重复执行，可能需要对函数接口或实现进行更改。

访问全局状态（如内存或文件）的函数具有可重复执行问题。这些函数需要借助线程提供的相应同步机制来保护其全局状态的使用。

使模块中的函数可重复执行的两个基本策略是代码锁定和数据锁定。

代码锁定

代码锁定是在函数调用级别执行的，而且可保证函数在锁定保护下完全执行。该假设针对通过函数对数据执行的所有访问。共享数据的函数应该在同一锁定下执行。

某些并行编程语言提供一种构造，称为**监视程序**。监视程序可以对监视程序范围内定义的函数隐式执行代码锁定。还可以通过互斥锁来实现监视。

可保证受同一互斥锁保护或同一监视程序中的函数相对于监视程序中的其他函数以原子方式执行。

数据锁定

数据锁定可保证一直维护对数据**集合**的访问。对于数据锁定，代码锁定概念仍然存在，但代码锁定只是对共享（全局）数据的轮流引用。对于互斥锁，在每个数据集合的临界段中只能有一个线程。

另外，在多个读取器、单个写入器协议中，允许每个数据集合或一个写入器具有多个读取器。当多个线程对不同数据集合执行操作时，这些线程可以在单个模块中执行。需要特别指出的是，对于多个读取器、单个写入器协议，这些线程不会在单个集合上发生冲突。因此，数据锁定通常比代码锁定具备的并发性更多。

使用锁定时应使用哪种策略，在程序中实现互斥、条件变量还是信号？是要尝试通过仅在必要时锁定并在不必要时尽快解除锁定来实现最大并行性（这种方法称作“细粒度锁定 (fine-grained locking)”）？还是要长期持有锁定，以使使用和释放锁的开销降到最低程度（这种方法称作“粗粒度锁定 (coarse-grained locking)”）？

锁定的粒度取决于锁定所保护的数据量。粒度非常粗的锁定可能是单一锁定，目的是保护所有数据。划分由适当数目的锁定保护数据的方式非常重要。锁定粒度过细可能会降低性能。当应用程序包含太多锁定时，与获取和释放锁关联的开销可能会变得非常大。

常见的明智之举是先使用粗粒度方法，确定瓶颈，并在必要时添加细粒度锁定来缓解瓶颈。此方法听起来是很合理的建议，但是您应该自行判断如何在最大化并行性与最小化锁定开销之间找到平衡。

不变量和锁定

对于代码锁定和数据锁定，**不变量**对于控制锁定复杂性非常重要。不变量指始终为真的条件或关系。

对于并发执行，其定义修改如下（在上述定义的基础上稍加修改即可得到此定义）：不变量是在设置关联锁定时为真的条件或关系。设置锁定后，不变量可能为假。但是，在释放锁之前，持有锁的代码必须重新建立不变量。

不变量还可以是设置锁定时为真的条件或关系。条件变量可以被认为含有一个不变量，而这个不变量就是这个条件。

示例 9–4 使用 assert(3X) 测试不变量

```
mutex_lock(&lock);

while((condition)==FALSE)

    cond_wait(&cv,&lock);

assert((condition)==TRUE);

  .

  .
```

示例 9–4 使用 assert(3X) 测试不变量 （续）

```
        .

    mutex_unlock(&lock);
```

`assert()` 语句用于测试不变量。`cond_wait()` 函数不保留不变量，这就是在线程返回时必须重新评估不变量的原因所在。

另一个示例就是用于管理双重链接的元素列表的模块。对于该列表中的每一项，良好的不变量是列表中前一项的向前指针。向前指针还应与向前项的向后指针指向同一元素。

假设此模块使用基于代码的锁定，进而受到单个全局互斥锁的保护。删除或添加项时，将获取互斥锁，正确处理指针，而且会释放互斥锁。显然，在处理指针的某一时间点，不变量为假，但在释放互斥锁之前，需要重新建立不变量。

避免死锁

死锁是指永久阻塞一组争用一组资源的线程。仅因为某个线程可以继续执行，并不表示不会在某个其他位置发生死锁。

导致死锁的最常见错误是**自死锁**或**递归死锁**。在自死锁或递归死锁中，线程尝试获取已被其持有的锁。递归死锁是在编程时很容易犯的错误。

例如，假设代码监视程序让每个模块函数在调用期间都获取互斥锁。随后，由互斥锁保护的模块内函数间的任何调用都会立即死锁。函数调用模块外的代码时，如果迂回调入任何受同一互斥锁保护的方法，则该函数也会发生死锁。

这种死锁的解决方案就是避免调用模块外可能通过某一路径依赖此模块的函数。需要特别指出的是，应避免在未重新建立不变量的情况下调用回调入模块的函数，而且在执行调用之前不要删除所有的模块锁。当然，在调用完成和重新获取锁定之后，必须验证状态，以确保预期的操作仍然有效。

另一种死锁的示例就是当两个线程（线程 1 和线程 2）分别获取互斥锁 A 和 B 时的情况。假设，线程 1 尝试获取互斥锁 B，线程 2 尝试获取互斥锁 A。如果线程 1 在等待互斥锁 B 时受到阻塞，而无法继续执行。线程 2 在等待互斥锁 A 时受到阻塞也无法继续执行。任何情况都不会发生变化。因此，在这种情况下，将永久阻塞线程，即出现死锁现象。

通过建立获取锁定的顺序（**锁定分层结构**），可以避免这种死锁。当所有线程始终按指定的顺序获取锁定时，即可避免这种死锁。

遵循严格的锁定获取顺序并不总是非常理想。例如，线程 2 具有许多有关在持有互斥锁 B 时模块状态的假设。放弃互斥锁 B 以获取互斥锁 A，然后按相应的顺序重新获取互斥锁 B 将导致线程放弃其假设。必须重新评估模块的状态。

阻塞同步元语通常具有变体，这些变体将尝试获取锁定，并在无法获取锁定时失败。例如 `mutex_trylock()`。元语变体的这种行为允许线程在不出现争用时破坏锁定分层结构。出现争用时，通常必须放弃持有的锁定，并按顺序重新获取锁定。

与调用相关的死锁

由于不能保证获取锁定的顺序，因此如果特定线程永远不能获取锁定就会出现问题。

持有锁的线程释放锁，一小段时间后重新获取锁定时，通常会出现此问题。由于锁被释放，因此其他线程似乎理应可以获取锁。但是，持有锁的线程未被阻塞。因此，从线程释放锁到重新获取锁定的时间内，该线程将持续运行。这样，就不会运行其他线程。

通常，通过在进行重新获取锁定的调用前调用 `thr_yield(3C)`，可以解决此类型的问题。`thr_yield()` 允许其他线程运行并获取锁定。

由于应用程序的时间片要求是可变的，因此系统不会强加任何要求。可通过调用 `thr_yield()` 来使线程根据需要进行分时操作。

锁定原则

请遵循以下的简单锁定原则。

- 请勿尝试在可能会对性能造成不良影响的长时间操作（如 I/O）中持有锁。
- 请勿在调用模块外且可能重进入模块的函数时持有锁。
- 一般情况下，请先使用粗粒度锁定方法，确定瓶颈，并在必要时添加细粒度锁定来缓解瓶颈。大多数锁定都是短期持有，而且很少出现争用。因此，请仅修复测得争用的那些锁定。
- 使用多个锁定时，通过确保所有线程都按相同的顺序获取锁定来避免死锁。

线程代码的一些基本原则

- 了解要导入的内容并了解其是否安全。

  线程程序不能随意输入非线程代码。
- 线程代码只能从初始线程中安全引用不安全的代码。

  以此方式引用不安全的代码确保了仅该线程使用与初始线程关联的静态存储。
- 使库可以安全地用于多线程时，请勿通过线程执行全局进程操作。

  请不要将全局操作或具有全局负面影响的操作更改为以线程方式执行。例如，如果将文件 I/O 更改为每线程操作，则线程无法在访问文件时进行协作。

  对于线程特定的行为或**线程识别**的行为，请使用线程功能。例如，终止 `main()` 时，应该仅终止将退出 `main()` 的线程。

```
        thr_exit();

         /*NOTREACHED*/
```

- 除非明确说明 Sun 提供的库是**安全的**，否则假定这些库**不安全**。

  如果参考手册项没有明确声明接口是 *MT-Safe*，则假定接口**不安全**。
- 请使用编译标志来管理二进制不兼容源代码更改。有关完整的说明，请参见第 7 章。
  - `-D_REENTRANT` 启用多线程。
  - `-D_POSIX_C_SOURCE` 提供 POSIX 线程行为。
  - `-D_POSIX_PTHREADS_SEMANTICS` 启用 Solaris 线程和 pthread 接口。当这两个接口发生冲突时，将优先使用 POSIX 接口。

创建和使用线程

线程软件包会对线程数据结构和栈进行高速缓存，以使重复创建线程的代价较为合理。但是，与管理等待独立工作的线程池相比，在需要线程时创建和销毁线程的代价通常会更高。RPC 服务器就是一个很好的示例，该服务器可以为每个请求创建一个线程，并在传送回复时销毁该线程。

创建线程的开销比创建进程的开销要少。但是，与创建几个指令的成本相比，创建线程并不是最经济的。请在至少持续处理数千个计算机指令时创建线程。

使用多处理器

借助多线程，可以充分利用多处理器，主要是通过并行性和可伸缩性。程序员应该了解多处理器与单处理器的内存模块之间的差异。

内存一致性与询问内存的处理器直接相关。对于单处理器，内存显然是一致的，因为只有一个处理器查看内存。

要提高多处理器性能，应放宽内存一致性。不应始终假设由一个处理器对内存所做的更改立即反映在该内存的其他处理器视图中。

使用共享变量或全局变量时，可借助同步变量来避免这种复杂性。

屏障同步有时是控制多处理器上并行性的一种有效方式。可以在附录 B 中找到屏障示例。

另一个多处理器问题是在线程必须等到所有线程都到达执行的共同点时进行有效同步。

注 – 始终使用线程同步元语访问共享内存位置时，此处讨论的问题并不重要。

基础体系结构

线程使用线程同步例程来同步对共享存储位置的访问。使用线程同步时，在共享内存多处理器上运行程序与在单处理器上运行程序的效果相同。

但是，在许多情况下，程序员可能很想利用多处理器，并使用“技巧”来避免同步例程。正如示例 9–5 和示例 9–6 所示，这类技巧可能是危险的。

了解常见多处理器体系结构支持的内存模块有助于了解这类危险。

主要的多处理器组件为：

- 运行程序的处理器
- 存储缓冲区，将处理器连接至其高速缓存
- **高速缓存**，存放最近访问或修改的存储位置的内容
- 内存，主存储区且供所有处理器共享

在简单的传统模型中，多处理器的行为方式就像将处理器直接连接到内存一样：当一个处理器存储在一个位置，而另一个处理器从同一位置直接装入时，第二个处理器将装入第一个处理器存储的内容。

可以使用高速缓存来加快平均内存访问速度。当该高速缓存与其他高速缓存保持一致时，即可实现所需的语义。

该简单方法存在的问题是，必须经常延迟处理器以确定是否已实现所需的语义。许多现代的多处理器使用各种技术来防止这类延迟，遗憾的是，这会更改内存模型的语义。

下面的两个示例说明了这两种方法及其效果。

“共享内存”多处理器

请考虑示例 9–5 中显示的对生成方和使用者问题的假设解决方案。

尽管此程序是在当前基于 SPARC 的多处理器上工作，但该程序假设所有的多处理器都有强秩序存储器 (strongly ordered memory)。因此，此程序不是可移植的。

示例 9–5 生成方和使用者问题：共享内存多处理器

```
                    char buffer[BSIZE];

                    unsigned int in = 0;

                    unsigned int out = 0;



void                                char

producer(char item) {                 consumer(void) {
```

示例 9–5 生成方和使用者问题：共享内存多处理器　　（续）

```
                                    char item;

    do

        ;/* nothing */                do

    while                                ;/* nothing */

        (in - out == BSIZE);            while

                                          (in - out == 0);

    buffer[in%BSIZE] = item;             item = buffer[out%BSIZE];

    in++;                               out++;

}                                   }
```

当此程序确实具有一个生成方和一个使用者，而且在共享内存多处理器上运行时，该程序看上去是正确的。*in* 与 *out* 之间的差异就是缓冲区中的项数目。

生成方通过重复计算此差异一直等待，直到缓冲区中有可用空间来存放新项为止。使用者一直等到缓冲区中存在项为止。

严格排序的内存可以对一个处理器上可供其他处理器直接使用的内存进行修改。对于强秩序存储器，该解决方案是正确的，即使考虑到 *in* 和 *out* 最终会溢出也是如此。只要 `BSIZE` 小于以单个词表示的最大整数，就会发生溢出。

共享内存多处理器不一定具有强秩序存储器。由一个处理器对内存所做的更改不一定可直接用于其他处理器。请了解一个处理器对不同内存位置进行两次更改时发生的具体情况。其他处理器不一定会按照更改顺序检测更改，因为不会立即修改内存。

首先，会将更改存储在对于高速缓存不可见的**存储缓冲区**中。

处理器将检查这些存储缓冲区，以确保程序具有一致的视图。但是，由于存储缓冲区对于其他处理器是不可见的，因此在某个处理器写入高速缓存之前，该处理器写入的内容不可见。

同步元语使用刷新存储缓冲区的特殊指令来执行高速缓存。请参见第 4 章。因此，对共享数据使用锁定可确保内存的一致性。

如果内存排序非常宽松，则示例 9–5 存在问题。使用者可能发现，在其查看对相应缓冲槽位所做的更改之前生成方已经增大了 *in*。

此排序称为**弱排序**，因为由一个处理器执行的存储在由另一个处理器执行时顺序可能会被打乱。但是，内存在同一个处理器中始终是一致的。要解决此不一致性，代码应使用互斥来刷新高速缓存。

由于趋势朝着放宽内存顺序方向发展，因此程序员在对所有的全局数据或共享数据使用锁定时变得越来越谨慎。

如示例 9–5 和示例 9–6 所示，锁定是最基本的。

Peterson 算法

示例 9–6 中的代码是 Peterson 算法的实现，该算法可以处理两个线程之间的互斥。此代码尝试保证临界段中只有一个线程。当线程调用 `mut_excl()` 时，线程会在某一时间“**快速**”进入临界段。

此处假设在进入临界段后线程很快就退出了。

示例 9–6 两个线程是否互斥？

```
void mut_excl(int me /* 0 or 1 */) {

    static int loser;

    static int interested[2] = {0, 0};

    int other; /* local variable */



    other = 1 - me;

    interested[me] = 1;

    loser = me;

    while (loser == me && interested[other])

        ;



    /* critical section */

    interested[me] = 0;

}
```

如果多处理器具有强秩序存储器，则此算法可工作一段时间。

某些多处理器（包括某些基于 SPARC 的多处理器）具有存储缓冲区。线程发出存储指令时，数据即被放入存储缓冲区中。缓冲区内容最终会被发送到高速缓存，但不一定会立即发送。每个处理器上的高速缓存都可以维护一致的内存视图，但修改的数据不会立即到达高速缓存。

在存储到多个内存位置时，更改将按正确顺序到达高速缓存和内存，但可能会在一定延迟后到达。具备此属性的基于 SPARC 的多处理器被称为具有全存储序顺序 (total store order, TSO)。

假设您面临的情况是，一个处理器存储到位置 A 并从位置 B 装入。另一个处理器存储到位置 B 并从位置 A 装入。第一个处理器将从位置 B 提取新修改的值，或第二个处理器将从位置 A 提取新修改的值，或者同时发生这两种情况。但是，两个处理器装入原有值的情况不会发生。

此外，由于装入和存储缓冲区导致的延迟，可能会发生“不可能的情况”。

使用 Peterson 算法时可能发生的情况是在不同处理器上运行的两个线程可以同时进入临界段。每个线程都可以存储到其各自的特定数组槽中，然后从其他槽装入。两个线程可以读取原有的值 (0)，每个线程都假设对方不存在，而且都可以进入临界段。测试程序时可能不会检测出这种问题，这种问题只会在稍后发生。

要避免此问题，请使用线程同步元语，其实现会发出特殊指令，从而强制将存储缓冲区写入高速缓存。

在共享内存并行计算机上并行化循环

在许多应用程序（特别是数值应用程序）中，一部分算法可以并行化，而其他部分却具有固有的顺序性，如下表所示。

表 9–1

顺序执行	并行执行
Thread 1	Thread 2 through Thread n
while(many_iterations) {	while(many_iterations) {
sequential_computation	
--- Barrier ---	--- Barrier ---
parallel_computation	parallel_computation
}	}

例如，您可能会使用严格的线性计算产生一组矩阵，并对使用并行算法的矩阵执行操作。随后，可以使用这些操作的结果来产生另一组矩阵，并行在这些矩阵上执行操作等。

这类计算的并行算法特征是计算期间很少需要执行同步。但是，为确保在并行计算开始之前完成顺序计算，需要对所有线程进行同步。

屏障将强制所有执行并行计算的线程一直等待，直到所有涉及到的线程都达到屏障为止。所有线程到达屏障后，即释放线程并同时开始计算。

线程程序示例

本指南介绍了各种重要线程编程问题。附录 A 提供了使用许多已论述功能和样式的 pthread 程序示例。附录 B 提供了使用 Solaris 线程的程序示例。

需要进一步阅读的内容

有关多线程的更详细信息，请参见 Steve Kleiman、Devang Shah 和 Bart Smaalders 合编的《*Programming with Threads*》（Prentice-Hall 出版，1995）。

◆ ◆ ◆ 附 录 A

样例应用程序：多线程 grep

本附录提供样例程序 tgrep，其中显示了 find(1) 与 grep(1) 组合后的多线程版本。

tgrep 的说明

tgrep 支持除常规 grep 的 -w 单词搜索选项和可独占使用选项以外的所有选项。

缺省情况下，tgrep 搜索与以下命令类似：

```
find . -exec grep [options] pattern {} \;
```

对于大型目录分层结构，tgrep 比 find 命令获取结果的速度更快，具体速度取决于可用的处理器数目。在单处理器计算机上，tgrep 的速度大约是 find 的两倍；在包含四个处理器的计算机上，tgrep 的速度大约是 find 的四倍。

-e 选项可用于更改 tgrep 解释模式字符串的方式。通常，在不使用 -e 选项的情况下，tgrep 将使用文字字符串匹配。如果使用 -e 选项，tgrep 将使用正则表达式处理程序的 MT-Safe（多线程安全）公共域版本。正则表达式方法的速度比较慢。

tgrep -B 选项可以使用 TGLIMIT 环境变量的值来限制搜索期间使用的线程的数目。如果未设置 TGLIMIT，则此选项没有效果。由于 tgrep 可以使用许多系统资源，因而使用 -B 选项是在分时系统上适当运行 tgrep 的一种方式。

注 – Catalyst 开发者的 CD 中包括 tgrep 的源代码。要了解如何获取副本，请与销售代表联系。

此处仅出现了多线程 main.c 模块。可以在 Catalyst 开发者的 CD 中找到其他模块（包括用于处理正则表达式的其他模块），以及文档和 make 程序的描述文件。

示例 A–1 tgrep 程序的源代码

```
/* Copyright (c) 1993, 1994  Ron Winacott                                */
```

示例 A–1 tgrep 程序的源代码 （续）

```
/* This program may be used, copied, modified, and redistributed freely */
/* for ANY purpose, so long as this notice remains intact.              */

#define _REENTRANT

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <errno.h>
#include <ctype.h>
#include <sys/types.h>
#include <time.h>
#include <sys/stat.h>
#include <dirent.h>

#include "version.h"

#include <fcntl.h>
#include <sys/uio.h>
#include <pthread.h>
#include <sched.h>
```

示例 A–1 tgrep 程序的源代码 （续）

```
#ifdef MARK
#include <prof.h> /* to turn on MARK(), use -DMARK to compile (see man prof5)*/
#endif

#include "pmatch.h"

#define PATH_MAX                1024 /* max # of characters in a path name */
#define HOLD_FDS                6  /* stdin,out,err and a buffer */
#define UNLIMITED               99999 /* The default tglimit */
#define MAXREGEXP               10  /* max number of -e options */

#define FB_BLOCK                0x00001
#define FC_COUNT                0x00002
#define FH_HOLDNAME             0x00004
#define FI_IGNCASE              0x00008
#define FL_NAMEONLY             0x00010
#define FN_NUMBER               0x00020
#define FS_NOERROR              0x00040
#define FV_REVERSE              0x00080
#define FW_WORD                 0x00100
#define FR_RECUR                0x00200
#define FU_UNSORT               0x00400
```

示例 A–1 tgrep 程序的源代码 （续）

```
#define FX_STDIN                0x00800

#define TG_BATCH                0x01000

#define TG_FILEPAT              0x02000

#define FE_REGEXP               0x04000

#define FS_STATS                0x08000

#define FC_LINE                 0x10000

#define TG_PROGRESS             0x20000


#define FILET                   1

#define DIRT                    2


typedef struct work_st {

    char                *path;

    int                 tp;

    struct work_st      *next;

} work_t;


typedef struct out_st {

    char                *line;

    int                 line_count;

    long                byte_count;

    struct out_st       *next;

} out_t;
```

示例 A–1 tgrep 程序的源代码　（续）

```
#define ALPHASIZ        128

typedef struct bm_pattern {     /* Boyer - Moore pattern                */

        short            p_m;           /* length of pattern string     */

        short            p_r[ALPHASIZ]; /* "r" vector                   */

        short           *p_R;           /* "R" vector                   */

        char            *p_pat;         /* pattern string               */

} BM_PATTERN;


/* bmpmatch.c */

extern BM_PATTERN *bm_makepat(char *p);

extern char *bm_pmatch(BM_PATTERN *pat, register char *s);

extern void bm_freepat(BM_PATTERN *pattern);

BM_PATTERN      *bm_pat;  /* the global target read only after main */


/* pmatch.c */

extern char *pmatch(register PATTERN *pattern, register char *string, int *len);

extern PATTERN *makepat(char *string, char *metas);

extern void freepat(register PATTERN *pat);

extern void printpat(PATTERN *pat);

PATTERN         *pm_pat[MAXREGEXP];  /* global targets read only for pmatch */


#include "proto.h"  /* function prototypes of main.c */
```

示例 A–1 tgrep 程序的源代码 （续）

```
/* local functions to POSIX threads only */
void pthread_setconcurrency_np(int con);
int pthread_getconcurrency_np(void);
void pthread_yield_np(void);

pthread_attr_t  detached_attr;
pthread_mutex_t output_print_lk;
pthread_mutex_t global_count_lk;

int             global_count = 0;

work_t          *work_q = NULL;
pthread_cond_t  work_q_cv;
pthread_mutex_t work_q_lk;
pthread_mutex_t debug_lock;

#include "debug.h"  /* must be included AFTER the
                       mutex_t debug_lock line */

work_t          *search_q = NULL;
pthread_mutex_t search_q_lk;
pthread_cond_t  search_q_cv;
```

示例 A–1 tgrep 程序的源代码 （续）

```
int             search_pool_cnt = 0;    /* the count in the pool now */
int             search_thr_limit = 0;   /* the max in the pool */

work_t          *cascade_q = NULL;
pthread_mutex_t cascade_q_lk;
pthread_cond_t  cascade_q_cv;
int             cascade_pool_cnt = 0;
int             cascade_thr_limit = 0;

int             running = 0;
pthread_mutex_t running_lk;

pthread_mutex_t stat_lk;
time_t          st_start = 0;
int             st_dir_search = 0;
int             st_file_search = 0;
int             st_line_search = 0;
int             st_cascade = 0;
int             st_cascade_pool = 0;
int             st_cascade_destroy = 0;
int             st_search = 0;
int             st_pool = 0;
int             st_maxrun = 0;
```

示例 A–1 tgrep 程序的源代码 （续）

```
int             st_worknull = 0;

int             st_workfds = 0;

int             st_worklimit = 0;

int             st_destroy = 0;


int             all_done = 0;

int             work_cnt = 0;

int             current_open_files = 0;

int             tglimit = UNLIMITED;    /* if -B limit the number of

                                   threads */

int             progress_offset = 1;

int             progress = 0;  /* protected by the print_lock ! */

unsigned int    flags = 0;

int             regexp_cnt = 0;

char            *string[MAXREGEXP];

int             debug = 0;

int             use_pmatch = 0;

char            file_pat[255];  /* file patten match */

PATTERN         *pm_file_pat; /* compiled file target string (pmatch()) */


/*

 * Main: This is where the fun starts

 */
```

示例 A–1 tgrep 程序的源代码　（续）

```
int

main(int argc, char **argv)

{

    int         c,out_thr_flags;

    long        max_open_files = 0l, ncpus = 0l;

    extern int  optind;

    extern char *optarg;

    int         prio = 0;

    struct stat sbuf;

    pthread_t tid,dtid;

    void        *status;

    char        *e = NULL, *d = NULL; /* for debug flags */

    int         debug_file = 0;

    struct sigaction sigact;

    sigset_t    set,oset;

    int         err = 0, i = 0, pm_file_len = 0;

    work_t      *work;

    int         restart_cnt = 10;


    /* NO OTHER THREADS ARE RUNNING */

    flags = FR_RECUR;  /* the default */


    while ((c = getopt(argc, argv, "d:e:bchilnsvwruf:p:BCSZzHP:")) != EOF) {
```

示例 A-1 tgrep 程序的源代码 （续）

```
        switch (c) {
#ifdef DEBUG
        case 'd':
            debug = atoi(optarg);
            if (debug == 0)
                debug_usage();

            d = optarg;
            fprintf(stderr,"tgrep: Debug on at level(s) ");
            while (*d) {
                for (i=0; i<9; i++)
                    if (debug_set[i].level == *d) {
                        debug_levels |= debug_set[i].flag;
                        fprintf(stderr,"%c ",debug_set[i].level);
                        break;
                    }
                d++;
            }
            fprintf(stderr,"\n");
            break;
        case 'f': debug_file = atoi(optarg); break;
#endif      /* DEBUG */
```

示例 A–1 tgrep 程序的源代码 （续）

```
        case 'B':
            flags |= TG_BATCH;
#ifndef __lock_lint
        /* locklint complains here, but there are no other threads */
            if ((e = getenv("TGLIMIT"))) {
                tglimit = atoi(e);
            }
            else {
                if (!(flags & FS_NOERROR))  /* order dependent! */
                    fprintf(stderr,"env TGLIMIT not set, overriding -B\n");
                flags &= ~TG_BATCH;
            }
#endif
            break;
        case 'p':
            flags |= TG_FILEPAT;
            strcpy(file_pat,optarg);
            pm_file_pat = makepat(file_pat,NULL);
            break;
        case 'P':
            flags |= TG_PROGRESS;
            progress_offset = atoi(optarg);
            break;
```

示例 A–1 tgrep 程序的源代码 （续）

```
        case ’S’: flags |= FS_STATS;    break;
        case ’b’: flags |= FB_BLOCK;    break;
        case ’c’: flags |= FC_COUNT;    break;
        case ’h’: flags |= FH_HOLDNAME; break;
        case ’i’: flags |= FI_IGNCASE;  break;
        case ’l’: flags |= FL_NAMEONLY; break;
        case ’n’: flags |= FN_NUMBER;   break;
        case ’s’: flags |= FS_NOERROR;  break;
        case ’v’: flags |= FV_REVERSE;  break;
        case ’w’: flags |= FW_WORD;     break;
        case ’r’: flags &= ~FR_RECUR;   break;
        case ’C’: flags |= FC_LINE;     break;
        case ’e’:
            if (regexp_cnt == MAXREGEXP) {
                fprintf(stderr,"Max number of regexp’s (%d) exceeded!\n",
                        MAXREGEXP);
                exit(1);
            }
            flags |= FE_REGEXP;
            if ((string[regexp_cnt] =(char *)malloc(strlen(optarg)+1))==NULL){
                fprintf(stderr,"tgrep: No space for search string(s)\n");
                exit(1);
            }
```

示例 A–1 tgrep 程序的源代码 （续）

```
            memset(string[regexp_cnt],0,strlen(optarg)+1);
            strcpy(string[regexp_cnt],optarg);
            regexp_cnt++;
            break;
        case 'z':
        case 'Z': regexp_usage();
            break;
        case 'H':
        case '?':
        default : usage();
        }
    }
    if (flags & FS_STATS)
        st_start = time(NULL);

    if (!(flags & FE_REGEXP)) {
       if (argc - optind < 1) {
            fprintf(stderr,"tgrep: Must supply a search string(s) "
                    "and file list or directory\n");
            usage();
        }
        if ((string[0]=(char *)malloc(strlen(argv[optind])+1))==NULL){
            fprintf(stderr,"tgrep: No space for search string(s)\n");
```

示例 A–1 tgrep 程序的源代码 （续）

```
            exit(1);
        }
        memset(string[0],0,strlen(argv[optind])+1);
        strcpy(string[0],argv[optind]);
        regexp_cnt=1;
        optind++;
    }

    if (flags & FI_IGNCASE)
        for (i=0; i<regexp_cnt; i++)
            uncase(string[i]);

    if (flags & FE_REGEXP) {
        for (i=0; i<regexp_cnt; i++)
            pm_pat[i] = makepat(string[i],NULL);
        use_pmatch = 1;
    }
    else {
        bm_pat = bm_makepat(string[0]); /* only one allowed */
    }

    flags |= FX_STDIN;
```

示例 A–1 tgrep 程序的源代码 （续）

```
    max_open_files = sysconf(_SC_OPEN_MAX);
    ncpus = sysconf(_SC_NPROCESSORS_ONLN);
    if ((max_open_files - HOLD_FDS - debug_file) < 1) {
        fprintf(stderr,"tgrep: You MUST have at least ONE fd "
                "that can be used, check limit (>10)\n");
        exit(1);
    }
    search_thr_limit = max_open_files - HOLD_FDS - debug_file;
    cascade_thr_limit = search_thr_limit / 2;
    /* the number of files that can be open */
    current_open_files = search_thr_limit;

    pthread_attr_init(&detached_attr);
    pthread_attr_setdetachstate(&detached_attr,
        PTHREAD_CREATE_DETACHED);

    pthread_mutex_init(&global_count_lk,NULL);
    pthread_mutex_init(&output_print_lk,NULL);
    pthread_mutex_init(&work_q_lk,NULL);
    pthread_mutex_init(&running_lk,NULL);
    pthread_cond_init(&work_q_cv,NULL);
    pthread_mutex_init(&search_q_lk,NULL);
```

示例 A-1 tgrep 程序的源代码 （续）

```
    pthread_cond_init(&search_q_cv,NULL);
    pthread_mutex_init(&cascade_q_lk,NULL);
    pthread_cond_init(&cascade_q_cv,NULL);

    if ((argc == optind) && ((flags & TG_FILEPAT) || (flags & FR_RECUR))) {
        add_work(".",DIRT);
        flags = (flags & ~FX_STDIN);
    }
    for ( ; optind < argc; optind++) {
        restart_cnt = 10;
        flags = (flags & ~FX_STDIN);
      STAT_AGAIN:
        if (stat(argv[optind], &sbuf)) {
            if (errno == EINTR) { /* try again !, restart */
                if (--restart_cnt)
                    goto STAT_AGAIN;
            }
            if (!(flags & FS_NOERROR))
                fprintf(stderr,"tgrep: Can't stat file/dir %s, %s\n",
                        argv[optind], strerror(errno));
            continue;
        }
        switch (sbuf.st_mode & S_IFMT) {
```

示例 A–1 tgrep 程序的源代码 （续）

```
        case S_IFREG :
            if (flags & TG_FILEPAT) {
                if (pmatch(pm_file_pat, argv[optind], &pm_file_len))
                    DP(DLEVEL1,("File pat match %s\n",argv[optind]));
                    add_work(argv[optind],FILET);
            }
            else {
                add_work(argv[optind],FILET);
            }
            break;
        case S_IFDIR :
            if (flags & FR_RECUR) {
                add_work(argv[optind],DIRT);
            }
            else {
                if (!(flags & FS_NOERROR))
                    fprintf(stderr,"tgrep: Can't search directory %s, "
                            "-r option is on. Directory ignored.\n",
                            argv[optind]);
            }
            break;
        }
    }
```

示例 A–1 tgrep 程序的源代码 （续）

```
    pthread_setconcurrency_np(3);

    if (flags & FX_STDIN) {
        fprintf(stderr,"tgrep: stdin option is not coded at this time\n");
        exit(0);                        /* XXX Need to fix this SOON */
        search_thr(NULL);
        if (flags & FC_COUNT) {
            pthread_mutex_lock(&global_count_lk);
            printf("%d\n",global_count);
            pthread_mutex_unlock(&global_count_lk);
        }
        if (flags & FS_STATS)
            prnt_stats();
        exit(0);
    }

    pthread_mutex_lock(&work_q_lk);
    if (!work_q) {
        if (!(flags & FS_NOERROR))
            fprintf(stderr,"tgrep: No files to search.\n");
        exit(0);
    }
```

示例 A–1 tgrep 程序的源代码 （续）

```
    pthread_mutex_unlock(&work_q_lk);


    DP(DLEVEL1,("Starting to loop through the work_q for work\n"));


    /* OTHER THREADS ARE RUNNING */
    while (1) {
        pthread_mutex_lock(&work_q_lk);
        while ((work_q == NULL || current_open_files == 0 || tglimit <= 0) &&
               all_done == 0) {
            if (flags & FS_STATS) {
                pthread_mutex_lock(&stat_lk);
                if (work_q == NULL)
                    st_worknull++;
                if (current_open_files == 0)
                    st_workfds++;
                if (tglimit <= 0)
                    st_worklimit++;
                pthread_mutex_unlock(&stat_lk);
            }
            pthread_cond_wait(&work_q_cv,&work_q_lk);
        }
        if (all_done != 0) {
            pthread_mutex_unlock(&work_q_lk);
```

示例 A–1 tgrep 程序的源代码 （续）

```
            DP(DLEVEL1,("All_done was set to TRUE\n"));
            goto OUT;
        }
        work = work_q;
        work_q = work->next;  /* maybe NULL */
        work->next = NULL;
        current_open_files--;
        pthread_mutex_unlock(&work_q_lk);

        tid = 0;
        switch (work->tp) {
        case DIRT:
            pthread_mutex_lock(&cascade_q_lk);
            if (cascade_pool_cnt) {
                if (flags & FS_STATS) {
                    pthread_mutex_lock(&stat_lk);
                    st_cascade_pool++;
                    pthread_mutex_unlock(&stat_lk);
                }
                work->next = cascade_q;
                cascade_q = work;
                pthread_cond_signal(&cascade_q_cv);
                pthread_mutex_unlock(&cascade_q_lk);
```

示例 A–1 tgrep 程序的源代码 （续）

```
                DP(DLEVEL2,("Sent work to cascade pool thread\n"));
            }
            else {
                pthread_mutex_unlock(&cascade_q_lk);
                err = pthread_create(&tid,&detached_attr,cascade,(void *)work);
                DP(DLEVEL2,("Sent work to new cascade thread\n"));
                if (flags & FS_STATS) {
                    pthread_mutex_lock(&stat_lk);
                    st_cascade++;
                    pthread_mutex_unlock(&stat_lk);
                }
            }
            break;
        case FILET:
            pthread_mutex_lock(&search_q_lk);
            if (search_pool_cnt) {
                if (flags & FS_STATS) {
                    pthread_mutex_lock(&stat_lk);
                    st_pool++;
                    pthread_mutex_unlock(&stat_lk);
                }
                work->next = search_q;  /* could be null */
                search_q = work;
```

示例 A–1 tgrep 程序的源代码　（续）

```
                pthread_cond_signal(&search_q_cv);
                pthread_mutex_unlock(&search_q_lk);
                DP(DLEVEL2,("Sent work to search pool thread\n"));
            }
            else {
                pthread_mutex_unlock(&search_q_lk);
                err = pthread_create(&tid,&detached_attr,
                                     search_thr,(void *)work);
                pthread_setconcurrency_np(pthread_getconcurrency_np()+1);
                DP(DLEVEL2,("Sent work to new search thread\n"));
                if (flags & FS_STATS) {
                    pthread_mutex_lock(&stat_lk);
                    st_search++;
                    pthread_mutex_unlock(&stat_lk);
                }
            }
            break;
        default:
            fprintf(stderr,"tgrep: Internal error, work_t->tp not valid\n");
            exit(1);
        }
        if (err) {  /* NEED TO FIX THIS CODE. Exiting is just wrong */
            fprintf(stderr,"Could not create new thread!\n");
```

示例 A–1 tgrep 程序的源代码　（续）

```
            exit(1);
        }
    }


 OUT:
    if (flags & TG_PROGRESS) {
        if (progress)
            fprintf(stderr,".\n");
        else
            fprintf(stderr,"\n");
    }
    /* we are done, print the stuff. All other threads are parked */
    if (flags & FC_COUNT) {
        pthread_mutex_lock(&global_count_lk);
        printf("%d\n",global_count);
        pthread_mutex_unlock(&global_count_lk);
    }
    if (flags & FS_STATS)
        prnt_stats();
    return(0); /* should have a return from main */
}


/*
```

示例 A-1 tgrep 程序的源代码 （续）

```
 * Add_Work: Called from the main thread, and cascade threads to add file
 * and directory names to the work Q.
 */
int
add_work(char *path,int tp)
{
    work_t      *wt,*ww,*wp;


    if ((wt = (work_t *)malloc(sizeof(work_t))) == NULL)
        goto ERROR;
    if ((wt->path = (char *)malloc(strlen(path)+1)) == NULL)
        goto ERROR;


    strcpy(wt->path,path);
    wt->tp = tp;
    wt->next = NULL;
    if (flags & FS_STATS) {
        pthread_mutex_lock(&stat_lk);
        if (wt->tp == DIRT)
            st_dir_search++;
        else
            st_file_search++;
        pthread_mutex_unlock(&stat_lk);
```

示例 A–1 tgrep 程序的源代码 （续）

```
    }
    pthread_mutex_lock(&work_q_lk);
    work_cnt++;
    wt->next = work_q;
    work_q = wt;
    pthread_cond_signal(&work_q_cv);
    pthread_mutex_unlock(&work_q_lk);
    return(0);
 ERROR:
    if (!(flags & FS_NOERROR))
        fprintf(stderr,"tgrep: Could not add %s to work queue. Ignored\n",
                path);
    return(-1);
}

/*
 * Search thread: Started by the main thread when a file name is found
 * on the work Q to be serached. If all the needed resources are ready
 * a new search thread will be created.
 */
void *
search_thr(void *arg) /* work_t *arg */
{
```

示例 A–1 tgrep 程序的源代码 （续）

```
    FILE        *fin;
    char        fin_buf[(BUFSIZ*4)];  /* 4 Kbytes */
    work_t      *wt,std;
    int         line_count;
    char        rline[128];
    char        cline[128];
    char        *line;
    register char *p,*pp;
    int            pm_len;
    int         len = 0;
    long        byte_count;
    long        next_line;
    int         show_line;  /* for the -v option */
    register int slen,plen,i;
    out_t       *out = NULL;    /* this threads output list */


    pthread_yield_np();
    wt = (work_t *)arg; /* first pass, wt is passed to use. */


    /* len = strlen(string);*/  /* only set on first pass */


    while (1) {  /* reuse the search threads */
        /* init all back to zero */
```

示例 A–1 tgrep 程序的源代码 （续）

```
        line_count = 0;

        byte_count = 0l;

        next_line = 0l;

        show_line = 0;


        pthread_mutex_lock(&running_lk);

        running++;

        pthread_mutex_unlock(&running_lk);

        pthread_mutex_lock(&work_q_lk);

        tglimit--;

        pthread_mutex_unlock(&work_q_lk);

        DP(DLEVEL5,("searching file (STDIO) %s\n",wt->path));


        if ((fin = fopen(wt->path,"r")) == NULL) {

            if (!(flags & FS_NOERROR)) {

                fprintf(stderr,"tgrep: %s. File \"%s\" not searched.\n",

                        strerror(errno),wt->path);

            }

            goto ERROR;

        }

        setvbuf(fin,fin_buf,_IOFBF,(BUFSIZ*4));  /* XXX */

        DP(DLEVEL5,("Search thread has opened file %s\n",wt->path));

        while ((fgets(rline,127,fin)) != NULL) {
```

示例 A–1 tgrep 程序的源代码 （续）

```
            if (flags & FS_STATS) {
                pthread_mutex_lock(&stat_lk);
                st_line_search++;
                pthread_mutex_unlock(&stat_lk);
            }
            slen = strlen(rline);
            next_line += slen;
            line_count++;
            if (rline[slen-1] == '\n')
                rline[slen-1] = '\0';
            /*
            ** If the uncase flag is set, copy the read in line (rline)
            ** To the uncase line (cline) Set the line pointer to point at
            ** cline.
            ** If the case flag is NOT set, then point line at rline.
            ** line is what is compared, rline is what is printed on a
            ** match.
            */
            if (flags & FI_IGNCASE) {
                strcpy(cline,rline);
                uncase(cline);
                line = cline;
            }
```

示例 A–1 tgrep 程序的源代码 （续）

```
            else {
                line = rline;
            }
            show_line = 1;  /* assume no match, if -v set */
            /* The old code removed */
            if (use_pmatch) {
                for (i=0; i<regexp_cnt; i++) {
                    if (pmatch(pm_pat[i], line, &pm_len)) {
                        if (!(flags & FV_REVERSE)) {
                            add_output_local(&out,wt,line_count,
                                             byte_count,rline);
                            continue_line(rline,fin,out,wt,
                                          &line_count,&byte_count);
                        }
                        else {
                            show_line = 0;
                        } /* end of if -v flag if / else block */
                        /*
                        ** if we get here on ANY of the regexp targets
                        ** jump out of the loop, we found a single
                        ** match so do not keep looking!
                        ** If name only, do not keep searcthing the same
                        ** file, we found a single match, so close the file,
```

示例 A–1 tgrep 程序的源代码 （续）

```
                    ** print the file name and move on to the next file.
                    */
                    if (flags & FL_NAMEONLY)
                        goto OUT_OF_LOOP;
                    else
                        goto OUT_AND_DONE;
                } /* end found a match if block */
            } /* end of the for pat[s] loop */
        }
        else {
            if (bm_pmatch( bm_pat, line)) {
                if (!(flags & FV_REVERSE)) {
                    add_output_local(&out,wt,line_count,byte_count,rline);
                    continue_line(rline,fin,out,wt,
                                  &line_count,&byte_count);
                }
                else {
                    show_line = 0;
                }
                if (flags & FL_NAMEONLY)
                    goto OUT_OF_LOOP;
            }
        }
```

示例 A–1 tgrep 程序的源代码　（续）

```
          OUT_AND_DONE:
            if ((flags & FV_REVERSE) && show_line) {
                add_output_local(&out,wt,line_count,byte_count,rline);
                show_line = 0;
            }
            byte_count = next_line;
        }
      OUT_OF_LOOP:
        fclose(fin);
        /*
        ** The search part is done, but before we give back the FD,
        ** and park this thread in the search thread pool, print the
        ** local output we have gathered.
        */
        print_local_output(out,wt);  /* this also frees out nodes */
        out = NULL; /* for the next time around, if there is one */
    ERROR:
        DP(DLEVEL5,("Search done for %s\n",wt->path));
        free(wt->path);
        free(wt);

        notrun();
        pthread_mutex_lock(&search_q_lk);
```

示例 A–1 tgrep 程序的源代码 （续）

```
        if (search_pool_cnt > search_thr_limit) {
            pthread_mutex_unlock(&search_q_lk);
            DP(DLEVEL5,("Search thread exiting\n"));
            if (flags & FS_STATS) {
                pthread_mutex_lock(&stat_lk);
                st_destroy++;
                pthread_mutex_unlock(&stat_lk);
            }
            return(0);
        }
        else {
            search_pool_cnt++;
            while (!search_q)
                pthread_cond_wait(&search_q_cv,&search_q_lk);
            search_pool_cnt--;
            wt = search_q;  /* we have work to do! */
            if (search_q->next)
                search_q = search_q->next;
            else
                search_q = NULL;
            pthread_mutex_unlock(&search_q_lk);
        }
    }
```

示例 A–1 tgrep 程序的源代码 （续）

```
    /*NOTREACHED*/
}


/*
 * Continue line: Special case search with the -C flag set. If you are
 * searching files like Makefiles, some lines might have escape char's to
 * contine the line on the next line. So the target string can be found, but
 * no data is displayed. This function continues to print the escaped line
 * until there are no more "\" chars found.
 */
int
continue_line(char *rline, FILE *fin, out_t *out, work_t *wt,
              int *lc, long *bc)
{
    int len;
    int cnt = 0;
    char *line;
    char nline[128];

    if (!(flags & FC_LINE))
        return(0);

    line = rline;
```

示例 A–1 tgrep 程序的源代码 （续）

```
  AGAIN:
    len = strlen(line);
    if (line[len-1] == '\\') {
        if ((fgets(nline,127,fin)) == NULL) {
            return(cnt);
        }
        line = nline;
        len = strlen(line);
        if (line[len-1] == '\n')
            line[len-1] = '\0';
        *bc = *bc + len;
        *lc++;
        add_output_local(&out,wt,*lc,*bc,line);
        cnt++;
        goto AGAIN;
    }
    return(cnt);
}


/*
 * cascade: This thread is started by the main thread when directory names
 * are found on the work Q. The thread reads all the new file, and directory
 * names from the directory it was started when and adds the names to the
```

示例 A–1 tgrep 程序的源代码 （续）

```
 * work Q. (it finds more work!)
 */

void *
cascade(void *arg)  /* work_t *arg */
{
    char        fullpath[1025];
    int         restart_cnt = 10;
    DIR         *dp;

    char        dir_buf[sizeof(struct dirent) + PATH_MAX];
    struct dirent *dent = (struct dirent *)dir_buf;
    struct stat   sbuf;
    char        *fpath;
    work_t      *wt;
    int         fl = 0, dl = 0;
    int         pm_file_len = 0;

    pthread_yield_np();  /* try toi give control back to main thread */
    wt = (work_t *)arg;

    while(1) {
        fl = 0;
```

示例 A–1 tgrep 程序的源代码 （续）

```
        dl = 0;
        restart_cnt = 10;
        pm_file_len = 0;

        pthread_mutex_lock(&running_lk);
        running++;
        pthread_mutex_unlock(&running_lk);
        pthread_mutex_lock(&work_q_lk);
        tglimit--;
        pthread_mutex_unlock(&work_q_lk);

        if (!wt) {
            if (!(flags & FS_NOERROR))
                fprintf(stderr,"tgrep: Bad work node passed to cascade\n");
            goto DONE;
        }
        fpath = (char *)wt->path;
        if (!fpath) {
            if (!(flags & FS_NOERROR))
                fprintf(stderr,"tgrep: Bad path name passed to cascade\n");
            goto DONE;
        }
        DP(DLEVEL3,("Cascading on %s\n",fpath));
```

示例 A–1 tgrep 程序的源代码 （续）

```
        if (( dp = opendir(fpath)) == NULL) {
            if (!(flags & FS_NOERROR))
                fprintf(stderr,"tgrep: Can't open dir %s, %s. Ignored.\n",
                        fpath,strerror(errno));
            goto DONE;
        }
        while ((readdir_r(dp,dent)) != NULL) {
            restart_cnt = 10;  /* only try to restart the interupted 10 X */

            if (dent->d_name[0] == '.') {
                if (dent->d_name[1] == '.' && dent->d_name[2] == '\0')
                    continue;
                if (dent->d_name[1] == '\0')
                    continue;
            }

            fl = strlen(fpath);
            dl = strlen(dent->d_name);
            if ((fl + 1 + dl) > 1024) {
                fprintf(stderr,"tgrep: Path %s/%s is too long. "
                        "MaxPath = 1024\n",
                        fpath, dent->d_name);
                continue;  /* try the next name in this directory */
```

示例 A–1 tgrep 程序的源代码 （续）

```
    }
    strcpy(fullpath,fpath);
    strcat(fullpath,"/");
    strcat(fullpath,dent->d_name);

RESTART_STAT:
    if (stat(fullpath,&sbuf)) {
        if (errno == EINTR) {
            if (--restart_cnt)
                goto RESTART_STAT;
        }
        if (!(flags & FS_NOERROR))
            fprintf(stderr,"tgrep: Can't stat file/dir %s, %s. "
                    "Ignored.\n",
                    fullpath,strerror(errno));
        goto ERROR;
    }

    switch (sbuf.st_mode & S_IFMT) {
    case S_IFREG :
        if (flags & TG_FILEPAT) {
            if (pmatch(pm_file_pat, dent->d_name, &pm_file_len)) {
                DP(DLEVEL3,("file pat match (cascade) %s\n",
```

示例 A–1 tgrep 程序的源代码 （续）

```
                                    dent->d_name));
                        add_work(fullpath,FILET);
                    }
                }
                else {
                    add_work(fullpath,FILET);
                    DP(DLEVEL3,("cascade added file (MATCH) %s to Work Q\n",
                                fullpath));
                }
                break;

            case S_IFDIR :
                DP(DLEVEL3,("cascade added dir %s to Work Q\n",fullpath));
                add_work(fullpath,DIRT);
                break;
            }
        }

    ERROR:
        closedir(dp);

    DONE:
        free(wt->path);
```

示例 A-1 tgrep 程序的源代码 （续）

```
        free(wt);
        notrun();
        pthread_mutex_lock(&cascade_q_lk);
        if (cascade_pool_cnt > cascade_thr_limit) {
            pthread_mutex_unlock(&cascade_q_lk);
            DP(DLEVEL5,("Cascade thread exiting\n"));
            if (flags & FS_STATS) {
                pthread_mutex_lock(&stat_lk);
                st_cascade_destroy++;
                pthread_mutex_unlock(&stat_lk);
            }
            return(0); /* pthread_exit */
        }
        else {
            DP(DLEVEL5,("Cascade thread waiting in pool\n"));
            cascade_pool_cnt++;
            while (!cascade_q)
                pthread_cond_wait(&cascade_q_cv,&cascade_q_lk);
            cascade_pool_cnt--;
            wt = cascade_q;  /* we have work to do! */
            if (cascade_q->next)
                cascade_q = cascade_q->next;
            else
```

示例 A–1 tgrep 程序的源代码 （续）

```
                cascade_q = NULL;
            pthread_mutex_unlock(&cascade_q_lk);
        }
    }
    /*NOTREACHED*/
}

/*
 * Print Local Output: Called by the search thread after it is done searching
 * a single file. If any oputput was saved (matching lines), the lines are
 * displayed as a group on stdout.
 */
int
print_local_output(out_t *out, work_t *wt)
{
    out_t       *pp, *op;
    int         out_count = 0;
    int         printed = 0;

    pp = out;
    pthread_mutex_lock(&output_print_lk);
    if (pp && (flags & TG_PROGRESS)) {
        progress++;
```

示例 A-1 tgrep 程序的源代码 （续）

```
        if (progress >= progress_offset) {
            progress = 0;
            fprintf(stderr,".");
        }
    }
    while (pp) {
        out_count++;
        if (!(flags & FC_COUNT)) {
            if (flags & FL_NAMEONLY) {  /* Pint name ONLY ! */
                if (!printed) {
                    printed = 1;
                    printf("%s\n",wt->path);
                }
            }
            else {  /* We are printing more then just the name */
                if (!(flags & FH_HOLDNAME))
                    printf("%s :",wt->path);
                if (flags & FB_BLOCK)
                    printf("%ld:",pp->byte_count/512+1);
                if (flags & FN_NUMBER)
                    printf("%d:",pp->line_count);
                printf("%s\n",pp->line);
            }
```

示例 A–1 tgrep 程序的源代码 （续）

```
        }
        op = pp;
        pp = pp->next;
        /* free the nodes as we go down the list */
        free(op->line);
        free(op);
    }

    pthread_mutex_unlock(&output_print_lk);
    pthread_mutex_lock(&global_count_lk);
    global_count += out_count;
    pthread_mutex_unlock(&global_count_lk);
    return(0);
}

/*
 * add output local: is called by a search thread as it finds matching lines.
 * the matching line, its byte offset, line count, etc. are stored until the
 * search thread is done searching the file, then the lines are printed as
 * a group. This way the lines from more then a single file are not mixed
 * together.
 */
```

示例 A–1 tgrep 程序的源代码 （续）

```
int
add_output_local(out_t **out, work_t *wt,int lc, long bc, char *line)
{
    out_t       *ot,*oo, *op;

    if (( ot = (out_t *)malloc(sizeof(out_t))) == NULL)
        goto ERROR;
    if (( ot->line = (char *)malloc(strlen(line)+1)) == NULL)
        goto ERROR;

    strcpy(ot->line,line);
    ot->line_count = lc;
    ot->byte_count = bc;

    if (!*out) {
        *out = ot;
        ot->next = NULL;
        return(0);
    }
    /* append to the END of the list; keep things sorted! */
    op = oo = *out;
    while(oo) {
        op = oo;
```

示例 A–1 tgrep 程序的源代码 （续）

```
        oo = oo->next;
    }
    op->next = ot;
    ot->next = NULL;
    return(0);

 ERROR:
    if (!(flags & FS_NOERROR))
        fprintf(stderr,"tgrep: Output lost. No space. "
                "[%s: line %d byte %d match : %s\n",
                wt->path,lc,bc,line);
    return(1);
}

/*
 * print stats: If the -S flag is set, after ALL files have been searched,
 * main thread calls this function to print the stats it keeps on how the
 * search went.
 */

void
prnt_stats(void)
{
```

示例 A–1 tgrep 程序的源代码 （续）

```
    float a,b,c;

    float t = 0.0;

    time_t  st_end = 0;

    char    tl[80];


    st_end = time(NULL); /* stop the clock */

    printf("\n----------------- Tgrep Stats. --------------------\n");

    printf("Number of directories searched:           %d\n",st_dir_search);

    printf("Number of files searched:                 %d\n",st_file_search);

    c = (float)(st_dir_search + st_file_search) / (float)(st_end - st_start);

    printf("Dir/files per second:                     %3.2f\n",c);

    printf("Number of lines searched:                 %d\n",st_line_search);

    printf("Number of matching lines to target:       %d\n",global_count);


    printf("Number of cascade threads created:        %d\n",st_cascade);

    printf("Number of cascade threads from pool:      %d\n",st_cascade_pool);

    a = st_cascade_pool; b = st_dir_search;

    printf("Cascade thread pool hit rate:             %3.2f%%\n",((a/b)*100));

    printf("Cascade pool overall size:                %d\n",cascade_pool_cnt);

    printf("Number of search threads created:         %d\n",st_search);

    printf("Number of search threads from pool:       %d\n",st_pool);

    a = st_pool; b = st_file_search;

    printf("Search thread pool hit rate:              %3.2f%%\n",((a/b)*100));
```

示例 A–1 tgrep 程序的源代码 （续）

```
    printf("Search pool overall size:                %d\n",search_pool_cnt);
    printf("Search pool size limit:                  %d\n",search_thr_limit);
    printf("Number of search threads destroyed:      %d\n",st_destroy);


    printf("Max # of threads running concurrenly:    %d\n",st_maxrun);
    printf("Total run time, in seconds.              %d\n",
           (st_end - st_start));


    /* Why did we wait ? */
    a = st_workfds; b = st_dir_search+st_file_search;
    c = (a/b)*100; t += c;
    printf("Work stopped due to no FD's:  (%.3d)       %d Times, %3.2f%%\n",
           search_thr_limit,st_workfds,c);
    a = st_worknull; b = st_dir_search+st_file_search;
    c = (a/b)*100; t += c;
    printf("Work stopped due to no work on Q:        %d Times, %3.2f%%\n",
           st_worknull,c);
    if (tglimit == UNLIMITED)
        strcpy(tl,"Unlimited");
    else
        sprintf(tl,"   %.3d   ",tglimit);
    a = st_worklimit; b = st_dir_search+st_file_search;
    c = (a/b)*100; t += c;
```

示例 A–1 tgrep 程序的源代码 （续）

```
    printf("Work stopped due to TGLIMIT:  (%.9s) %d Times, %3.2f%%\n",
           tl,st_worklimit,c);
    printf("Work continued to be handed out:          %3.2f%%\n",100.00-t);
    printf("--------------------------------------------------\n");
}
/*
 * not running: A glue function to track if any search threads or cascade
 * threads are running. When the count is zero, and the work Q is NULL,
 * we can safely say, WE ARE DONE.
 */
void
notrun (void)
{
    pthread_mutex_lock(&work_q_lk);
    work_cnt--;
    tglimit++;
    current_open_files++;
    pthread_mutex_lock(&running_lk);
    if (flags & FS_STATS) {
        pthread_mutex_lock(&stat_lk);
        if (running > st_maxrun) {
            st_maxrun = running;
            DP(DLEVEL6,("Max Running has increased to %d\n",st_maxrun));
```

示例 A–1 tgrep 程序的源代码 （续）

```
        }
        pthread_mutex_unlock(&stat_lk);
    }
    running--;
    if (work_cnt == 0 && running == 0) {
        all_done = 1;
        DP(DLEVEL6,("Setting ALL_DONE flag to TRUE.\n"));
    }
    pthread_mutex_unlock(&running_lk);
    pthread_cond_signal(&work_q_cv);
    pthread_mutex_unlock(&work_q_lk);
}


/*
 * uncase: A glue function. If the -i (case insensitive) flag is set, the
 * target strng and the read in line is converted to lower case before
 * comparing them.
 */
void
uncase(char *s)
{
    char        *p;
```

示例 A–1 tgrep 程序的源代码 （续）

```
    for (p = s; *p != NULL; p++)
        *p = (char)tolower(*p);
}

/*
 * usage: Have to have one of these.
 */

void
usage(void)
{
    fprintf(stderr,"usage: tgrep <options> pattern <{file,dir}>...\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"Where:\n");
#ifdef DEBUG
    fprintf(stderr,"Debug     -d = debug level -d <levels> (-d0 for usage)\n");
    fprintf(stderr,"Debug     -f = block fd's from use (-f #)\n");
#endif
    fprintf(stderr,"          -b = show block count (512 byte block)\n");
    fprintf(stderr,"          -c = print only a line count\n");
    fprintf(stderr,"          -h = Do NOT print file names\n");
    fprintf(stderr,"          -i = case insensitive\n");
    fprintf(stderr,"          -l = print file name only\n");
```

示例 A–1 tgrep 程序的源代码 （续）

```
    fprintf(stderr,"          -n = print the line number with the line\n");
    fprintf(stderr,"          -s = Suppress error messages\n");
    fprintf(stderr,"          -v = print all but matching lines\n");
#ifdef NOT_IMP
    fprintf(stderr,"          -w = search for a \"word\"\n");
#endif
    fprintf(stderr,"          -r = Do not search for files in all "
                               "sub-directories\n");
    fprintf(stderr,"          -C = show continued lines (\"\\\")\n");
    fprintf(stderr,"          -p = File name regexp pattern. (Quote it)\n");
    fprintf(stderr,"          -P = show progress. -P 1 prints a DOT on stderr\n"
                   "               for each file it finds, -P 10 prints a DOT\n"
                   "               on stderr for each 10 files it finds, etc...\n");
    fprintf(stderr,"          -e = expression search.(regexp) More then one\n");
    fprintf(stderr,"          -B = limit the number of threads to TGLIMIT\n");
    fprintf(stderr,"          -S = Print thread stats when done.\n");
    fprintf(stderr,"          -Z = Print help on the regexp used.\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"Notes:\n");
    fprintf(stderr,"      If you start tgrep with only a directory name\n");
    fprintf(stderr,"      and no file names, you must not have the -r option\n");
    fprintf(stderr,"      set or you will get no output.\n");
    fprintf(stderr,"      To search stdin (piped input), you must set -r\n");
```

示例 A–1 tgrep 程序的源代码 （续）

```
    fprintf(stderr,"      Tgrep will search ALL files in ALL \n");
    fprintf(stderr,"      sub-directories. (like */* */*/* */*/*/* etc..)\n");
    fprintf(stderr,"      if you supply a directory name.\n");
    fprintf(stderr,"      If you do not supply a file, or directory name,\n");
    fprintf(stderr,"      and the -r option is not set, the current \n");
    fprintf(stderr,"      directory \".\" will be used.\n");
    fprintf(stderr,"      All the other options should work \"like\" grep\n");
    fprintf(stderr,"      The -p patten is regexp; tgrep will search only\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"      Copy Right By Ron Winacott, 1993-1995.\n");
    fprintf(stderr,"\n");
    exit(0);
}


/*
 * regexp usage: Tell the world about tgrep custom (THREAD SAFE) regexp!
 */
int
regexp_usage (void)
{
    fprintf(stderr,"usage: tgrep <options> -e \"pattern\" <-e ...> "
            "<{file,dir}>...\n");
    fprintf(stderr,"\n");
```

示例 A–1 tgrep 程序的源代码 （续）

```
    fprintf(stderr,"metachars:\n");
    fprintf(stderr,"    . - match any character\n");
    fprintf(stderr,"    * - match 0 or more occurrences of previous char\n");
    fprintf(stderr,"    + - match 1 or more occurrences of previous char.\n");
    fprintf(stderr,"    ^ - match at beginning of string\n");
    fprintf(stderr,"    $ - match end of string\n");
    fprintf(stderr,"    [ - start of character class\n");
    fprintf(stderr,"    ] - end of character class\n");
    fprintf(stderr,"    ( - start of a new pattern\n");
    fprintf(stderr,"    ) - end of a new pattern\n");
    fprintf(stderr,"    @(n)c - match <c> at column <n>\n");
    fprintf(stderr,"    | - match either pattern\n");
    fprintf(stderr,"    \\ - escape any special characters\n");
    fprintf(stderr,"    \\c - escape any special characters\n");
    fprintf(stderr,"    \\o - turn on any special characters\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"To match two diffrerent patterns in the same command\n");
    fprintf(stderr,"Use the or function. \n"
            "ie: tgrep -e \"(pat1)|(pat2)\" file\n"
            "This will match any line with \"pat1\" or \"pat2\" in it.\n");
    fprintf(stderr,"You can also use up to %d -e expressions\n",MAXREGEXP);
    fprintf(stderr,"RegExp Pattern matching brought to you by Marc Staveley\n");
    exit(0);
```

示例 A–1 tgrep 程序的源代码　（续）

```
}


/*
 * debug usage: If compiled with -DDEBUG, turn it on, and tell the world
 * how to get tgrep to print debug info on different threads.
 */


#ifdef DEBUG
void
debug_usage(void)
{
    int i = 0;


    fprintf(stderr,"DEBUG usage and levels:\n");
    fprintf(stderr,"-------------------------------------------------\n");
    fprintf(stderr,"Level                   code\n");
    fprintf(stderr,"-------------------------------------------------\n");
    fprintf(stderr,"0                 This message.\n");
    for (i=0; i<9; i++) {
        fprintf(stderr,"%d                %s\n",i+1,debug_set[i].name);
    }
    fprintf(stderr,"-------------------------------------------------\n");
    fprintf(stderr,"You can or the levels together like -d134 for levels\n");
```

示例 A–1 tgrep 程序的源代码 （续）

```
    fprintf(stderr,"1 and 3 and 4.\n");
    fprintf(stderr,"\n");
    exit(0);
}
#endif

/* Pthreads NP functions */

#ifdef __sun
void
pthread_setconcurrency_np(int con)
{
    thr_setconcurrency(con);
}

int
pthread_getconcurrency_np(void)
{
    return(thr_getconcurrency());
}

void
pthread_yield_np(void)
```

示例 A–1 tgrep 程序的源代码 （续）

```
{
/*     In Solaris 2.4, these functions always return - 1 and set errno to ENOSYS */
    if (sched_yield())  /* call UI interface if we are older than 2.5 */
        thr_yield();
}

#else
void
pthread_setconcurrency_np(int con)
{
    return;
}

int
pthread_getconcurrency_np(void)
{
    return(0);
}

void
pthread_yield_np(void)
{
    return;
```

示例 A–1 tgrep 程序的源代码 （续）

```
}

#endif
```

◆ ◆ ◆ 附 录 B

Solaris 线程示例： barrier.c

barrier.c 程序演示了 Solaris 线程的屏障实现。有关屏障的定义，请参见第 232 页中的 “在共享内存并行计算机上并行化循环” 。

示例 B–1 Solaris 线程示例： barrier.c

```
#define _REENTRANT


/* Include Files        */


#include    <thread.h>

#include    <errno.h>


/* Constants & Macros   *


/* Data Declarations    */


typedef struct {

     int     maxcnt;     /* maximum number of runners */

     struct _sb {

        cond_t  wait_cv;   /* cv for waiters at barrier */
```

示例 B–1 Solaris 线程示例： barrier.c （续）

```
        mutex_t wait_lk;   /* mutex for waiters at barrier */
        int     runners;   /* number of running threads */
    } sb[2];
    struct _sb   *sbp;    /* current sub-barrier */
} barrier_t;




/*
 * barrier_init - initialize a barrier variable.
 *
 */

int
barrier_init( barrier_t *bp, int count, int type, void *arg ) {
    int n;
    int i;

    if (count < 1)
        return(EINVAL);

    bp->maxcnt = count;
```

示例 B–1 Solaris 线程示例： barrier.c （续）

```
     bp->sbp = &bp->sb[0];


     for (i = 0; i < 2; ++i) {
#if defined(__cplusplus)
     struct barrier_t::_sb *sbp = &( bp->sb[i] );
#else
     struct _sb *sbp = &( bp->sb[i] );
#endif
     sbp->runners = count;


     if (n = mutex_init(&sbp->wait_lk, type, arg))
             return(n);


     if (n = cond_init(&sbp->wait_cv, type, arg))
             return(n);
     }
     return(0);
}


/*
 * barrier_wait - wait at a barrier for everyone to arrive.
 *
 */
```

示例 B–1 Solaris 线程示例： barrier.c （续）

```
int
barrier_wait(register barrier_t *bp) {
#if defined(__cplusplus)
     register struct barrier_t::_sb *sbp = bp->sbp;
#else
     register struct _sb *sbp = bp->sbp;
#endif
     mutex_lock(&sbp->wait_lk);

     if (sbp->runners == 1) {   /* last thread to reach barrier */
           if (bp->maxcnt != 1) {
           /* reset runner count and switch sub-barriers */
                  sbp->runners = bp->maxcnt;
                  bp->sbp = (bp->sbp == &bp->sb[0])
                                       ? &bp->sb[1] : &bp->sb[0];

                  /* wake up the waiters          */
                  cond_broadcast(&sbp->wait_cv);
            }
        } else {
           sbp->runners--;         /* one less runner  */
```

示例 B–1 Solaris 线程示例： barrier.c （续）

```
            while (sbp->runners != bp->maxcnt)
                    cond_wait( &sbp->wait_cv, &sbp->wait_lk);
        }

     mutex_unlock(&sbp->wait_lk);

     return(0);
}

/*
 * barrier_destroy - destroy a barrier variable.
 *
 */

int
barrier_destroy(barrier_t *bp) {
        int     n;
        int     i;

        for (i=0; i < 2; ++ i) {
                if (n = cond_destroy(&bp->sb[i].wait_cv))
                        return( n );
```

示例 B–1 Solaris 线程示例： barrier.c （续）

```
                if (n = mutex_destroy( &bp->sb[i].wait_lk))
                        return(n);
        }

        return(0);
}



#define NTHR    4
#define NCOMPUTATION 2
#define NITER   1000
#define NSQRT   1000

        void *
compute(barrier_t *ba )
{
    int count = NCOMPUTATION;

    while (count--) {
        barrier_wait( ba );
        /* do parallel computation */
    }
}
```

示例 B–1 Solaris 线程示例： barrier.c （续）

```
main( int argc, char *argv[] ) {
        int             i;
        int             niter;
        int             nthr;
        barrier_t       ba;
        double          et;
        thread_t        *tid;

        switch ( argc ) {
          default:
          case 3 :      niter   = atoi( argv[1] );
                        nthr    = atoi( argv[2] );
                        break;

          case 2 :      niter   = atoi( argv[1] );
                        nthr    = NTHR;
                        break;

          case 1 :      niter   = NITER;
                        nthr    = NTHR;
                        break;
        }
```

示例 B–1 Solaris 线程示例： barrier.c （续）

```
        barrier_init( &ba, nthr + 1, USYNC_THREAD, NULL );
        tid = (thread_t *) calloc(nthr, sizeof(thread_t));

        for (i = 0; i < nthr; ++i) {
                int     n;

                if (n = thr_create(NULL, 0,
                        (void *(*)( void *)) compute,
                        &ba, NULL, &tid[i])) {
                    errno = n;
                    perror("thr_create");
                    exit(1);
                }
        }

    for (i = 0; i < NCOMPUTATION; i++) {
        barrier_wait(&ba );
    /* do parallel algorithm */
    }

    for (i = 0; i < nthr; i++) {
        thr_join(tid[i], NULL, NULL);
```

示例 B–1 Solaris 线程示例： barrier.c （续）

```
    }


}
```

索引

N

P

R

S

T

U

V

W

X

安

绑

报

比

编

变

标

并

不

同

完

未

系

细

线