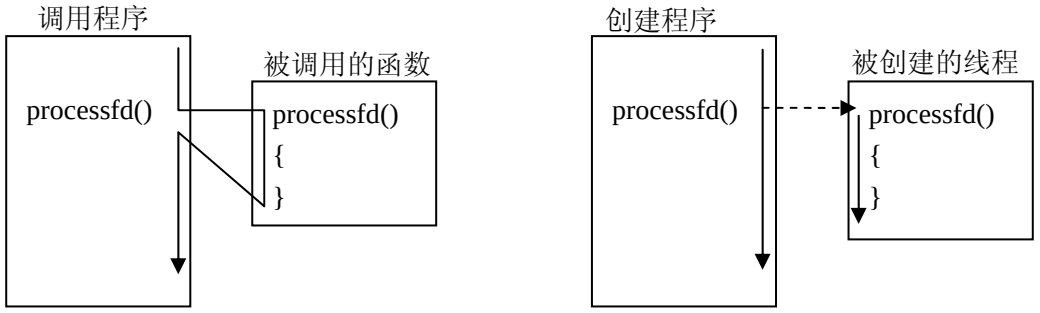
第六章 POSIX 线程

实现并行的一种方法是多个进程通过共享内存或消息传递来进行协作和同步。另一种方法是在单个地址空间中使用多个执行线程。

1、线程的基本概念

按照教科书上的定义，进程是资源管理的最小单位，线程是程序执行的最小单位。在操作系统设计上，从进程演化出线程，最主要的目的就是更好的支持 SMP 以及减小（进程/线程）上下文切换开销。SMP 是对称多处理器系统的缩写。



线程是进程内部的一个指令执行序列。一个进程至少有一个线程。对于支持线程编程的操作系统来讲，可以在一个进程内通过创建线程的方法来使进程具有多个指令执行序列（即线程）。此时，如果进程运行在 SMP 机器上，它就可以同时使用多个 CPU 来执行各个线程，达到最大程度的并行，以提高效率；同时，即使是在单 CPU 的机器上，采用多线程模型来设计程序，正如采用多进程模型代替单进程模型一样，也将使设计更简洁、功能更完备，程序的执行效率也更高。

可以这样理解，线程就是从某种程度上来说“更小的进程”，在早期的 Linux 中，线程就是通过轻量级进程来实现的。当进程调用 fork（）创建子进程时，这个新进程几乎复制了父进程的所有资源（尽管实现上是 copy on write，但理解上必须理解为全部复制）。但在进程中创建一个新线程时，新线程只独立拥有自己的栈（因而有自己的局部变量），而其它的资源都和创建者（父线程？）共享，包括全局变量、文件描述符、信号句柄和当前目录状态等。因此线程一方面可以共享资源，另一方面其调度更轻。

是否采用线程编程，不能一概而论，因为线程能做的事，进程也能做，只不过线程更轻

而已（对于调度而言），而且像 Linux 系统对进程的创建和调度就已经做得效率很高。一般而言，怯用线程更多的是其调试比较困难。我们完全可以针对具体的平台、具体的任务测试出到底该用进程还是该用线程。

下面通过一个例子先对线程有一个感性的认识：

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg);
char message[] = "Hello World";

int main()
{
    int res;
    pthread_t a_thread;
    void *thread_result;

    res = pthread_create(&a_thread, NULL, thread_function, (void
*)message);
    if (res != 0){
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    printf("Waiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0){
        perror("Thread join failed");
        exit(EXIT_FAILURE);
```

```
        }
        printf("Thread joined, it returned %s\n", (char
*)thread_result);
        printf("Message is now %s\n", message);
        exit(EXIT_SUCCESS);
    }

    void *thread_function(void *arg)
    {
        printf("Thread_function is runing, Argument was %s\n", (char
*)arg);
        sleep(3);
        strcpy(message, "Bye");
        pthread_exit("thank you for the CPU time");
    }
```

首先，程序包含了头文件 pthread.h，因为程序中要用线程库函数。其次，在 Linux 中编译这个程序要用以下命令（假定程序文件名为 thread1.c，输出文件为 a.out）：

gcc –D_REENTRANT thread1.c -lpthread

其中“–D_REENTRANT”表示在编译时要定义宏_REENTRANT，告诉编译器我们需要可重入功能。编译器将使用可重入版本的函数并使 errno 成为一个函数调用，它能够以一种安全的多线程方式来获取真正的 errno 值。“-lpthread”表示要用到库文件 libpthread.a 或 libpthread.so。

程序一开始就有一个执行线程（从语句 res=开始），然后该语句调用函数 pthread_create 创建一个新线程，新线程是执行函数 thread_function（）的执行线程，至此，该进程有两个线程，且我们认为它们并发执行（实际上要看具体平台）：新线程开始执行函数，而原线程开始执行函数 pthread_join，该函数类似于进程的 wait 函数，但这里意思是汇合，即原线程在等待新线程。等新线程执行结束后，原线程打印信息并结束程序。

如果创建一个新线程后原线程开始休息，则没有任何意义。下面的程序表明两个线程都在运行。你能说清楚两个线程的具体行为吗？

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg);
char message[] = "Hello World";
int run_now = 1;

int main()
{
    int res;
    pthread_t a_thread;
    void *thread_result;

    printf(“Message is now %s”, message);
    res = pthread_create(&a_thread, NULL, thread_function, (void
*)message);
    if (res != 0){
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    int print_count1 = 0;
    while(print_count1++ < 20){
        printf("I am parent\n");
        if (run_now == 1){
            printf("%d\n", print_count1);
            run_now = 2;
        }
```

```
            else
            sleep(2);
        }
        printf("Waiting for thread to finish...\n");
        res = pthread_join(a_thread, &thread_result);
        if (res != 0){
            perror("Thread join failed");
            exit(EXIT_FAILURE);
        }
        printf("Thread   joined,   it   returned   %s\n",   (char
*)thread_result);
        printf("Message is now %s\n", message);
        exit(EXIT_SUCCESS);
    }

    void *thread_function(void *arg)
    {
        int print_count2 = 0;

        while(print_count2++ < 20){
            printf("I am child\n");
            if (run_now == 2){
                printf("%dx\n", print_count2);
                run_now = 1;
            }
            else
            sleep(2);
        }
        strcpy(message, “Bye”)
```

```
}
```

最后需要说明一点的是线程没有父子概念，这意味着不仅旧线程可以 join 新线程，新线程也可以 join 旧线程。

线程的可移植是一个很重要的问题。各种线程的实现版本因厂商的不同而有所差异，Linux 系统在 1996 年就已经获得线程的支持，当时的线程库称为 LinuxThread。但 LinuxThread 本身存在的问题，特别是兼容性上的问题，严重阻碍了 Linux 上的跨平台应用，从而使得 Linux 上的线程应用一直保持在比较低的水平。目前有许多项目都在研究如何才能改善 Linux 对线程的支持。最为人看好的有两个项目，一个是 RedHat 公司牵头研发的 NPTL（Native Posix Thread Library），另一个则是 IBM 投资开发的 NGPT（Next Generation Posix Threading），二者都是围绕完全兼容 POSIX 1003.1c 来做。NPTL 已经用在 RedHat Linux 上。

POSIX 标准用 POSIX：THR 线程扩展中描述的 POSIX 线程来处理线程可移植性问题。本章讨论的代码都是基于 POSIX 标准的，因此适用于任何一种线程库。

2、线程管理

线程包中通常包含了用于线程创建和线程销毁、调度、强制互斥和条件等待的函数。典型的线程包中还包括一个运行系统来对线程进行透明的管理（也就是说，用户是不知道运行系统的存在的）。线程被创建时，运行系统分配数据结构来装载线程 ID，栈和程序计数器值。线程的内部数据结构中可能还包括调度和使用信息。一个进程的各个线程共享那个进程的整个地址空间。它们可以修改全局变量，访问打开的文件描述符，并用其它的方式互相配合或互相干扰。

下表列出基本的 POSIX 线程管理函数：

POSIX 函数	描述
pthread_self	找出自己的线程 ID
pthread_equal	测试两个线程 ID 是否相等
pthread_create	创建一个线程
pthread_detach	设置线程以释放资源
pthread_join	等待一个线程
pthread_exit	退出线程，而不退出进程

pthread_cancel	终止另一个线程
pthread_setcancelstate	设置取消状态
pthread_setcanceltype	设置取消类型
pthread_testcancel	测试取消类型

如果成功，大多数线程都返回 0，如果不成功，大多数线程函数都会返回非零的错误码，它们不设置 errno，因此调用程序不能用 perror 来报告错误，但是可以用 strerror 来报告错误。POSIX 标准特别说明，没有任何一个 POSIX 线程函数会返回 EINTR，而且如果 POSIX 线程函数被信号中断，也不必重新启动。

POSIX 线程由一个 pthread_t 类型的 ID 来引用。线程可以通过调用 pthread_self 找出自己的 ID。

```
SYNOPSIS
#incluide <pthread.h>
pthread_t pthread_self(void);                    POSIX：THR
```

由于 pthread_t 可能是一个结构，可以用 pthread_equal 来比较线程 ID 是否相等。

```
SYNOPSIS
#incluide <pthread.h>
pthread_t pthread_equal(pthread_t t1, pthread_t t2);POSIX：THR
```

如果 t1 等于 t2，pthread_equal 返回一个非零值。如果线程 ID 不相等，则返回 0。

pthread_create 函数创建了一个线程。与有些线程工具不同，例如 Java 编程语言提供的那些线程工具不同，POSIX 的 pthread_create 会自动使线程成为可运行的，而不需要一个单独的启动操作。参数 thread 指向新创建的线程 ID；attr 表示一个封装了线程的各种属性的属性对象，如果 attr 为 NULL，线程就具有默认的属性；start_routine 是线程开始执行的时候调用的函数的名字；start_routine 有一个由 arg 指定的参数，这个参数是一个指向 void 的指针。start_routine 返回一个指向 void 的指针，这个返回值被 pthread_join 当作退出状态来处理。

```
SYNOPSIS
#incluide <pthread.h>
int pthread_create(pthread_t *restrict thread,
```

```
                    const pthread_attr_t *restrict attr,
                    void *(*start_routine)(void *)
                    void *restrict arg);        POSIX：THR
```

如果成功，pthread_create 返回 0，否则返回一个非零的错误码。

不要让 pthread_create 的原型吓住你，其实线程是很容易创建和使用的。

除非是一个分离线程，否则在线程退出时，它是不会释放它的资源的。pthread_detach 函数将线程分离，它设置线程的内部选项来说明线程退出后，线程的存储空间可以被重新收回。分离线程退出时不会报告它们的状态。没有分离的线程是可结合的，而且在另一个线程为它们调用 pthread_join 或者整个进程退出之前，这些线程不会释放它们所有的资源。

pthread_join 函数与进程级的 waitpid 类似，会使调用者等待特定的线程退出。为防止内存泄露，长时间运行的程序最终应该为每个线程调用 pthread_detach 或 pthread_join。pthread_join 函数将调用线程挂起，直到第一个参数指定的目标线程终止为止，参数 value_ptr 为指向返回值的指针提供了一个位置，这个返回值是由目标线程传递给 pthread_exit 或 return 的，如果 value_ptr 为 NULL，调用程序就不会对目标线程的返回状态进行检索了。

```
SYNOPSIS
#incluide <pthread.h>
int pthread_detach(pthread_t thread);
int pthread_join(pthread_t thread, void **value_ptr);POSIX：THR
```

如果成功，二者返回 0，否则返回一个非零的错误码。

pthread_join 并不是唯一一种可以使主线程一直阻塞到其它线程都结束的方法。

进程的终止可以通过直接调用 exit、执行 return、或者通过进程的某个其它线程调用 exit 来实现。在任何一种情况下，所有的线程都会终止。如果主线程在创建了其它线程之后没有工作可做，它就应该阻塞到所有线程都结束为止，或者调用 pthread_exit(NULL)。

调用 exit 会使整个进程终止，调用 pthread_exit 只会使调用线程终止。POSIX 没有为 pthread_exit 定义任何错误。

对一个成功的 pthread_join 来说，value_ptr 的值是可用的。但是 pthread_exit 中的 value_ptr 必须指向线程退出后仍然存在的数据，因此线程不应该为 value_ptr 使用指向自动局部数据的指针。

线程可以通过取消机制迫使其它线程返回。线程可以调用 pthread_cancel 来请求取消另

一个线程。结果由目标线程的类型和取消状态决定。pthread_cancel 函数没有阻塞调用进程，它在发出取消请求后就返回了。

```
SYNOPSIS
#incluide <pthread.h>
void pthread_exit(void *value_ptr);
int pthread_cancel(pthread_t thread);            POSIX：THR
```

如果成功，pthread_cancel 返回 0，否则返回一个非零的错误码。

线程收到一个取消请求时会发生什么情况取决于它的状态和类型。如果线程处于 PTHREAD_CANCEL_ENABLE 状态，它就接收取消请求。另一方面，如果线程处于 PTHREAD_CANCEL_DISABLE 状态，取消请求就会被保持在挂起状态。默认情况下线程处于 PTHREAD_CANCEL_ENABLE 状态。pthread_setcancelstate 函数用来改变调用线程的取消状态，state 说明要设置的新状态，oldstate 指向一个整数的指针，这个整数中装载了以前的状态。

```
SYNOPSIS
#incluide <pthread.h>
int pthread_setcancelstate(int state， int *oldstate); POSIX：THR
```

如果成功 pthread_setcancelstate 返回 0，否则返回一个非零的错误值。

作为一个通用的原则，改变了其取消状态或类型的函数应该在返回之前恢复它们的值，除非遵守了这个原则，否则调用程序无法对程序的行为做出可靠的假设。

如果线程有类似于锁或打开的文件描述符这样的在退出之前必须释放掉的资源，取消就有可能引起一些问题。尽管被取消的线程在退出之前可以执行一个清楚函数（这里不讨论），但在退出处理程序中释放资源有时候行不通，这样，在执行中的某些地方退出，会将程序置于一种无法接受的状态。当线程将退出作为对取消请求的响应时，取消类型允许线程控制它在什么地方退出。pthread_setcanceltype 函数根据它的 type 参数指定的值来修改线程的取消类型，参数 oldtype 是指向一个位置的指针，在这个位置上保存了以前的取消类型。当线程的取消类型为 PTHREAD_CANCEL_ASYNCHRONOUS 时，线程在任何时候都可以响应取消请求，与之相对，PTHREAD_CANCEL_DEFERRED 使得线程只能在特定的几个取消点上响应取消请求。默认情况下为后者。

对于 PTHREAD_CANCEL_DEFERRED 取消类型的线程，可以通过调用

pthread_testcancel 在代码特定的位置上设置一个取消点。

```
SYNOPSIS
#incluide <pthread.h>
int pthread_setcanceltype(int type， int *oldtype);
void pthread_testcancel(void);                    POSIX：THR
```

如果成功，pthread_setcanceltype 返回 0，否则返回非零错误值。pthread_testcancel 没有返回值。

下面的程序是一个创建了一个线程来拷贝文件的程序，该程序说明了如何传递一个指向数组的指针。前面的程序 ex_pthreadcopy.c 是主程序，它将包含两个打开的文件描述符的数组传递给一个运行 copyfilemalloc 函数的线程。后面的程序是函数 copyfilemalloc 的实现，这个函数从一个文件中读入并向另一个文件中输出。参数 arg 装载了一个指针，这个指针指向一对用来表示源和目标文件的打开的文件描述符。变量 bytesp、infd 和 outfd 是在 copyfilemalloc 的局部栈上分配的，不能被其它线程直接访问。

copyfilemalloc 还说明了从线程中返回值的策略。因为不允许线程返回一个指向它的局部变量的指针，所以线程为返回拷贝的字节数分配了内存空间。POSIX 要求 malloc 是线程安全的。copyfilemalloc 函数返回 bytesp 指针，这和调用 pthread_exit 的效果是一样的。使用完之后释放这个空间是调用程序（ex_pthreadcopy）的责任。在这种情况下，程序会终止，因此就不需要 free 了。

copyfilemalloc 本身调用了 copyfile 函数（该函数已作为.o 文件加载到 libmy.a 中），对 copyfilemalloc.c 用如下命令编译：

gcc –D_REENTRANT –c copyfilemalloc.c

然后用命令 ar 将其加载到 libmy.a 中：

ar –r libmy.a copyfilemalloc.o

将其函数申明添加到 myinclude.h 头文件中：

```
int makeargv(const char *s, const char *delimiters, char ***argvp);
int copyfile(int fromfd, int tofd);
void *copyfilemalloc(void *arg);
```

用如下命令编译 ex_pthreadcopy.c：

gcc ex_pthreadcopy.c –L. –lmy -lpthread

ex_pthreadcopy.c

```
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include "myinclude.h"
#define PERMS (S_IRUSR | S_IWUSR)
#define READ_FLAGS O_RDONLY
#define WRITE_FLAGS (O_WRONLY | O_CREAT | O_TRUNC)

int main(int argc, char *argv[])
{
   int *bytesptr;
   int error;
   int fds[2];
   pthread_t tid;

   if (argc != 3){
      fprintf(stderr, "Usage: %s fromfile tofile\n", argv[0]);
      return 1;
   }
   if (((fds[0] = open(argv[1], READ_FLAGS)) == -1) ||
      ((fds[1] = open(argv[2], WRITE_FLAGS, PERMS)) == -1)){
      perror("Failed to open the files");
      return 1;
   }
```

```
    if (error = pthread_create(&tid, NULL, copyfilemalloc, fds)){
        fprintf(stderr,   "Failed   to   create   thread:   %s\n",
strerror(error));
        return 1;
    }
    if (error = pthread_join(tid, (void**)&bytesptr)){
        fprintf(stderr,   "Failed   to   join   thread:   %s\n",
strerror(error));
        return 1;
    }
    printf("Number of bytes copied: %d\n", *bytesptr);
    return 0;
}
```

copyfilemalloc.c

```
#include <stdlib.h>
#include <unistd.h>

void *copyfilemalloc(void *arg)
{
    int *bytesp;
    int infd;
    int outfd;

    infd = *((int *)(arg));
    outfd = *((int *)(arg)+1);
    if ((bytesp = (int *)malloc(sizeof(int))) == NULL)
        return NULL;
    *bytesp = copyfile(infd, outfd);
    close(infd);
```

```
    close(outfd);
    return bytesp;
}
```

3、用户线程和内核线程

用户级线程（user-level thread）和内核级线程（kernel-level thread）是两种传统的线程控制模式。

用户级线程对内核来说是不可见的，用户级线程由一个属于进程的线程运行系统来调度，因此它们只能共享分配给它们的封装进程的资源，这种实现机制一方面使得用户级线程的开销较低，但另一方面由于分配给进程的处理器资源只能是一个 CPU，因此，用户级线程不能共享多处理器的优势。

内核线程对内核可见，内核了解每一个作为可调度实体的线程，这些线程可以在全系统范围内竞争处理器（而不仅限于某进程资源内）。内核级线程的调度开销比整个进程的调度开销要小，但比用户级线程的管理代价更高。

混合线程模型（hybrid thread model）通过提供两个级别的控制，具备了用户级和内核级模型的优点。此时，用户级线程被称为线程，而内核级线程被称为轻量级进程（lightweight process），例如在 Sun 的 Solaris 线程实现中。

POSIX 线程调度模型是一个混合模型，它很灵活，足以在标准的特定实现中支持用户级和内核级线程。模型中包括两级调度——线程级和内核实体级。

POSIX 引入了一个线程调度竞争范围（thread-scheduling contention scope）的概念，这个概念赋予了程序员一些控制权，使他们可以控制怎样将内核实体映射为线程，线程的 contenrionscope 属性可以是 PTHREAD_SCOPE_PROCESS，也可以是 PTHREAD_SCOPE_SYSTEM。带有属性 PTHREAD_SCOPE_PROCESS 的线程与它们所在的进程中的其它线程竞争处理器资源。带有 PTHREAD_SCOPE_SYSTEM 属性的线程很像内核级线程，它们在全系统范围内竞争处理器资源。

如果你的 POSIX 实现既支持 POSIX：THR 线程扩展，又支持 POSIX：TPS 线程执行调度扩展，那么你就可以用 pthread_attr_getscope 来获得范围，并用 pthread_sttr_setscope 来设置范围。

4、线程的属性

创建线程的函数 pthread_create 有 4 个参数，分别为新创建线程的 ID、新创建线程的属性、新创建线程的执行函数和向新创建线程传递的参数。在前面的例子中，对于创建线程的属性，我们用 NULL。现在我们来讨论线程的属性。

线程的属性主要包括线程的状态、线程栈和线程调度三方面的内容。POSIX 将这些属性封装到一个 pthread_attr_t 类型的对象中去，用面向对象的方式表示和设置这些属性。下表显示的是线程属性的可设置属性及其相关函数。

属性	函数
属性对象	pthread_attr_destroy
	pthread_attr_init
状态	pthread_attr_getdetachstate
	pthread_attr_setdetachstate
栈	pthread_attr_getguardsize
	pthread_attr_setguardsize
	pthread_attr_getstack
	pthread_attr_setstack
调度	pthread_attr_getinheritsched
	pthread_attr_setinheritsched
	pthread_attr_getschedparam
	pthread_attr_setschedparam
	pthread_attr_getschedpolicy
	pthread_attr_setschedpolicy
	pthread_attr_getscope
	pthread_attr_setscope

函数 pthread_attr_init 用默认值对一个线程属性对象进行初始化，pthread_attr_destroy 函数将属性对象的值设为无效的。

如果成功，pthread_attr_init 和 pthread_attr_destroy 都返回 0，否则返回非零错误码。

```
SYNOPSIS
```

```
#incluide <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t, *attr);  POSIX：THR
```

函数 pthread_attr_getdetachstate 函数用来查看一个属性对象的状态，而 pthread_attr_setdetachstate 函数用来设置一个属性对象的状态。线程状态的可能取值为 PTHREAD_CREATE_JOINABLE 和 PTHREAD_CREATE_DETACHED。参数 attr 是一个指向属性对象的指针，参数 detachstate 对应于要为 pthread_attr_setdetachstate 设置的值以及一个指向 pthread_attr_getdetachstate 检索所得的值的指针。

```
SYNOPSIS
#incluide <pthread.h>
int pthread_attr_getdetachstate(const pthread_attr_t, *attr,
                                int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t, *attr,
                                int detachstate); POSIX：THR
```

如果成功，这些函数返回 0，否则返回一个非零的错误码。

分离线程终止时，释放它们的资源，而可接合的线程应该用 pthread_join 来等待。分离线程不能用 pthread_join 来等待。默认情况下，线程是可接合的。你可以在创建线程之后调用 pthread_detach 函数来分离一个线程，也可以通过用带有线程状态 PTHREAD_CREATE_DETACHED 的属性对象来创建一个处于分离状态的线程。

下面的代码创建了一个分离线程来运行 processfd：

```
int error, fd;
pthread_attr_t tattr;
pthread_t tid;

if (error = pthread_attr_init(&tattr))
   fprintf(stderr, “Failed to create attribute object:%s\n”,
strerror(error));
else if (error = pthread_attr_setdetachstate(&tattr,
PTHREAD_CREATE_DETACHED))
```

```
        fprintf(stderr, “Failed to set attribute state to detached: %s\n”, strerror(error));
    else if (error = pthread_create(&tid, &tattr, processfd, &fd ))
        fprintf(stderr, “Failed to create thread: %s\n”, strerror(error));
```

线程有一个栈，用户可以设置栈的位置和大小。要为线程定义栈的布局和大小，就必须先用特定的栈属性来创建一个属性对象，然后用这个属性对象来调用 pthread_create。

pthread_attr_getstack 函数用来查看栈的参数，pthread_attr_setstack 函数用来设置一个属性对象的栈参数。每个函数的参数 attr 都是一个指向属性对象的指针。pthread_attr_setstack 函数将栈的地址和栈的大小作为额外的参数，pthread_attr_getstack 则将指向这些条目的指针当作参数。

```
SYNOPSIS
#incluide <pthread.h>
int pthread_attr_getstack(const pthread_attr_t, *restrict attr,
    void **restrict stackaddr, size_t *restrict stacksize);
int pthread_attr_setstack(pthread_attr_t, *attr,
     void *stackaddr, size_t stacksize); POSIX：THR
```

如果成功，这些函数就返回 0，否则返回一个非零的错误码。

如果用户还没有设置 stackaddr，POSIX 还提供了检查栈溢出或者为栈溢出设置警戒的函数，pthread_attr_getguardsize 函数用来查看警戒参数，pthread_attr_setguardsize 函数在一个属性对象中设置了用来控制栈溢出的警戒参数，如果参数 guardsize 为 0，栈就是无警戒的。实现为一个非零的 guardsize 至少分配了 guardsize 的额外内存。对这个额外内存的溢出会引发一个错误，而且可能会为线程产生一个 SIGSEGV 信号。

```
SYNOPSIS
#incluide <pthread.h>
int pthread_attr_getguardsize(const pthread_attr_t, *restrict attr, size_t *restrict guardsize);
int pthread_attr_setguardsize(pthread_attr_t, *attr,
```

```
                                    size_t guardsize); POSIX：THR,XSI
```

如果成功，函数就返回 0，否则返回一个非零的错误码。

线程的竞争范围（contention scope）控制了线程是在进程内部还是在系统级竞争调度资源。pthread_attr_getscope 用来查看竞争范围，pthread_attr_setscope 用来设置一个属性对象的竞争范围。参数 attr 是一个指向属性对象的指针。参数 contenrionscope 对应于要为 pthread_attr_setscope 设置的值，以及一个指向要从 pthread_attr_getscope 获得的值的指针。参数 contentionscope 的可能值是 PTHREAD_SCOPE_PROCESS 和 PTHREAD_SCOPE_SYSTEM。

```
SYNOPSIS
#incluide <pthread.h>
int pthread_attr_getscope(const pthread_attr_t, *restrict attr,
                          int *restrict contentionscope);
int pthread_attr_setscope(pthread_attr_t, *attr,
                          int contentionscope); POSIX：THR,TPS
```

如果成功，函数返回 0，否则返回一个非零的错误码。

下面的代码创建了一个争夺内核资源的线程（内核级线程）：

```
int error;
int fd;
pthread_attr_t tattr;
pthread_t tid;

if(error = pthread_attr_init(&tattr))
   fprintf(stderr,”Failed to create an attribution object:%s\n”,
strerror(error))
else     if     (error     =     pthread_attr_setscope(&tattr,
PTHREAD_SCOPE_SYSTEM))
   fprintf(stderr,”Failed   to   set   scope   to   system:%s\n”,
strerror(error))
else if (error = pthread_create(&tid, &tattr, proceddfd, &fd))
```

```
        fprintf(stderr, ”Failed to create a thread: %s\n”,
strerror(error))
```

POSIX 允许线程用不同的方式继承调度策略。pthread_attr_getinheritsched 函数负责查看调度继承策略，而 pthread_attr_setinheritsched 函数负责为一个属性对象设置调度继承策略。参数 attr 是一个指向属性对象的指针。参数 inheritsched 对应于要为 pthread_attr_setinheritsched 设置的值和一个小指向要从 pthread_attr_getinheritsched 获得的值的指针。inheritsched 的两个可能的值为 PTHREAD_INHERIT_SCHED 和 PTHREAD_EXPLICIT_SCHED，前者表示调度属性从创建线程中继承，后者表示线程使用的是这个属性对象中的调度属性。

```
    SYNOPSIS
    #incluide <pthread.h>
    int pthread_attr_getinheritsched(const pthread_attr_t, *restrict
attr,int *restrict inheritsched);
    int pthread_attr_setinheritsched(pthread_attr_t, *attr,
                          int inheritsched);      POSIX：THR,TPS
```

如果成功，函数就返回 0，否则返回一个非零的错误码。

pthread_attr_getschedparam 函数负责查看调度参数，而 pthread_attr_setschedparam 负责设置一个属性对象的调度参数。参数 attr 是一个指向属性对象的指针。参数 param 是一个指向 pthread_attr_setschedparam 所要设置的值的指针，或者是一个指向要从 pthread_attr_getschedparam 获得的值的指针。注意，与其他的 pthread_attr_set 函数不同，第二个参数是一个指针，因为它对应于一个结构而不是一个整数。

```
    SYNOPSIS
    #incluide <pthread.h>
    int pthread_attr_getschedparam(const pthread_attr_t, *restrict
attr,struct sched_partam *restrict param);
    int pthread_attr_setschedparam(pthread_attr_t, *attr,
      const struct sched_param *restrict param);   POSIX：THR,TPS
```

如果成功，就返回 0，否则返回一个非零的错误码。

调度参数取决于调度策略。它们被封装在一个 sched.h 中定义的 struct sched_param 结构

中。SCHED_FIFO 和 SCHED_RR 调度策略只需使用 struct sched_param 结构中的 sched_priority 成员。sched_priority 字段装载了一个 int 类型的优先级值，较大的优先级值对应较高的优先级。

下面的代码用指定的优先级创建了一个线程属性对象，并返回该对象的指针，如果失败，则返回 NULL：

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

pthread_attr_t *makepriority(int priority)
{
   pthread_attr_t *attr;
   int error;
   struct sched_param param;

   if ((attr = (pthread_attr_t *)malloc(sizeof(pthread_attr_t)))
== NULL)
      return NULL;
   if (!(error = pthread_attr_init(attr) &&
     !(error = pthread_attr_getschedparam(attr, &param))){
      param.sched_priority = priority;
      error = pthread_attr_setschedparam(attr, &param);
   }
   if (error){
      free(attr);
      errno = error;
      return NULL;
   }
   return attr;
```

```
}
```

上面的程序还说明了修改参数的常规策略——先读入当前值，只修改你需要修改的那些值。

具有相同优先级的线程根据它们的调度策略指定的方式来竞争处理器资源。头文件 sched.h 为先进先出的调度策略定义了 SCHED_FIFO，为轮转调度定义了 SCHED_RR，并为一些其他的策略定义了 SCHED_OTHER。

先进先出的调度策略（如 SCHED_FIFO）为一个特定的优先级中处于可运行状态的线程使用了一个队列。转成可运行状态的阻塞线程被放入与它们的优先级相对应的队列的末尾，而被抢占的运行线程则放在队列的前面。

在轮转调度（如 SCHED_RR）中，当运行线程运行完它的时间片之后，就被放入它的优先级队列的末尾，除此之外，它的行为与先进先出类似。sched_rr_get_interval 函数用来返回时间片。

分散调度与先进先出类似，它用了两个参数（重装周期和执行容量）来控制运行在一个指定优先级的线程的数目。它的规则相当复杂，但这种策略允许程序更容易地将竞争处理器的线程数目作为可用资源的一个函数来管理。

抢占优先级策略是 SCHED_OTHER 最常见的一种实现方式。其实现的具体行为取决于调度范围和其他的因素。

pthread_attr_getschedpolicy 函数负责获取调度策略，pthread_attr_setschedpolicy 函数负责设置一个属性对象的调度策略。参数 attr 是一个指向属性对象的指针。对设置函数来说，参数 polocy 是一个要设置的值，对获取函数来说，是一个指向要获取的值的指针。

```
SYNOPSIS
#incluide <pthread.h>
int pthread_attr_getschedpolicy(const pthread_attr_t, *restrict
attr, int *restrict policy);
int pthread_attr_setschedpolicy(pthread_attr_t, *attr ,
                                int policy);   POSIX：THR
```

如果成功，这些函数返回 0，否则返回一个非零的错误码。

练习 6-1：并行文件拷贝

（1）、编写一个被称为 copydirectory 的函数，函数原型如下所示。

void *copydirectory(void *arg);

copydirectory 函数从一个目录中将所有文件都拷贝到另一个目录中去。目录名作为两个连续的字符串（用一个空格分隔）在 arg 中传递。假设源目录和目标目录都存在，且只拷贝普通文件，而忽略子目录。为每一个要拷贝的文件创建一个线程来完成拷贝。在创建下一个线程之前等待每个线程的结束。

（2）、编写一个 main 程序，它用两个命令行参数来表示源目录和目标目录。main 程序创建了一个线程来运行 copydirectory，然后执行 pthread_join 来等待 copydirectory 线程的结束。用这个程序来测试 copydirectory 的第一个版本。

（3）、修改 copydirectory 函数，如果目标目录不存在，copydirectory 就创建这个目录。对这个版本进行测试。

（4）、修改 copydirectory，在它创建了一个线程来拷贝一个文件之后，继续创建线程来拷贝其他的文件。用与下面的结构类似的节点结构将每个 copydirectory 线程的线程 ID 和打开的文件没描述符保存在一个链表中。

```
type struct copy_struct {
   char *namestring;
   int sourcefd;
   int destinationfd;
   int bytescopied;
   pthread_t tid;
   struct copy_struct *next;
} copyinfo_t;
copyinfo_t *head = NULL;
copyinfo_t *tail = NULL;
```

当 copydirectory 函数创建了线程，拷贝了目录中所有的文件之后，就为它的列表中的每个线程执行 pthread_join，并释放 copyinfo_t 结构。

（5）、修改 copydirectory，如果文件是一个目录而不是一个文件，就创建一个进程来运行 copydirectory。

（6）、设计一个执行计时的方法，对普通拷贝和线程化的拷贝进行比较。

（7）、如果运行在一个很大的目录中，程序试图打开的文件描述符或线程的数目可能

会比允许一个进程打开的数目多。设计一种方法来处理这种情况。

（8）、如果线程的范围是 PTHREAD_SCOPR_SYSTEM 而不是 PTHREAD_SCOPE_PROCESS 的话，运行时间会有什么变化，试解释之。

5、线程同步

本节主要讨论互斥量、条件变量和读-写锁等线程的同步机制，下表总结了 POSIX：THR 扩展中可用的同步函数。每种同步机制都提供了一个初始化函数和一个销毁对象的函数。可以对互斥锁和条件变量进行静态初始化。这三种类型的同步机制都有与之相关的属性对象，但这里仅使用带有默认属性的同步对象。

描述	POSIX 函数
互斥量	pthread_mutex_destory
	pthread_mutex_init
	pthread_mutex_lock
	pthread_mutex_trylock
	pthread_mutex_unlock
条件变量	pthread_cond_broadcast
	pthread_cond_destory
	pthread_cond_init
	pthread_cond_signal
	pthread_cond_timedwait
	pthread_cond_wait
读-写锁	pthread_rwlock_destory
	pthread_rwlock_init
	pthread_rwlock_rdlock
	pthread_rwlock_timedrdlock
	pthread_rwlock_timedwrlock
	pthread_rwlock_tryrdlock
	pthread_rwlock_trywrlock

5.1、互斥量

互斥量（mutex）是最简单也是最有效的线程同步机制。程序用互斥量来保护临界区，以获得对资源的排他性访问权。

互斥量是一种特殊的变量，它可以处于锁定（locked）状态，也可以处于解锁（unlocked）状态。如果互斥量是锁定的，就有一个特定的线程持有（hold）这个互斥量，如果没有线程持有这个互斥量，我们就说这个互斥量处于解锁状态。每个互斥量还有一个等待持有该互斥量的线程队列。互斥量的等待队列中的线程获得互斥量的顺序由线程调度策略确定。

当互斥量处于解锁状态，并且有一个线程试图获取这个互斥量时，那个线程就可以非阻塞的获得这个互斥量。否则，互斥函数被阻塞。互斥函数不是线程取消点，也不能被信号中断。除非进程终止了、（从信号处理程序中）用 pthread_exit 终止了线程、或者异步取消了线程（通常不用这种方法），否则，等待互斥锁的线程不能被逻辑地中断。

互斥量只能被短时间的持有。出现等待输入这样的持续时间不确定的情况时，用条件变量来同步。

POSIX 使用 pthread_mutex_t 类型的变量来表示互斥量。互斥量使用之前通常都必须对其进行初始化。对静态分配的互斥量来说，只要将 PTHREAD_MUTEX_INITIALIZER 赋给变量就行了。对动态分配或没有默认互斥属性的互斥量来说，则要调用 pthread_mutex_init 来执行初始化工作。

pthread_mutex_init 的参数 mutex 是一个指向要初始化的互斥量的指针。向 pthread_mutex_init 的参数 attr 传递 NULL 来初始化一个带有默认属性的互斥量。否则，就要用类似于线程属性对象所用的方式，先创建，然后再初始化互斥属性对象。

```
   SYNOPSIS
   #incluide <pthread.h>
   int pthread_mutex_init(pthread_mutex_t, *restrict mutex,
              const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  POSIX：THR
```

如果成功，函数返回 0，否则返回一个非零的错误码。

下面的代码通过静态初始化程序，用默认属性对互斥量 mylock 进行了初始化：

pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER；

静态初始化程序通常比 pthread_mutex_init 更有效，而且在线程开始执行之前确保它们正好会被执行一次。

POSIX 中明确指出，如果线程试图去初始化一个已经被初始化了的互斥量，则程序的行为是未知的，因此要避免这种情况。

pthread_mutex_destory 销毁了它的参数所引用的互斥量。参数 mutex 是一个指向要销毁的互斥量的指针。可以用 pthread_mutex_init 对一个被销毁的互斥量重新进行初始化。

```
SYNOPSIS
#incluide <pthread.h>
int pthread_mutex_destory(pthread_mutex_t, *mutex);POSIX:THR
```

如果成功，函数返回 0，否则返回一个非零的错误码。

POSIX 中有两个可以用来获取互斥量的函数，pthread_mutex_lock 会一直阻塞直到互斥量可用为止，而 pthread_mutex_trylock 则会立即返回。pthread_mutex_unlock 负责释放指定的互斥量。

```
SYNOPSIS
#incluide <pthread.h>
int pthread_mutex_lock(pthread_mutex_t, *mutex);
int pthread_mutex_trylock(pthread_mutex_t, *mutex);
int pthread_mutex_unlock(pthread_mutex_t, *mutex); POSIX:TH
```

如果成功，这些函数就返回 0，否则返回一个非零的错误码。

互斥量的应用范围比较广泛，在程序中既可以用互斥量来保护一个临界区（临界区可以是一个变量，也可以是一个数据结构），又可以用互斥量来保护不安全的函数（有些函数不是线程安全函数）。

因为互斥量必须能被所有需要同步的线程访问，所以它们通常会以全局变量的形式出现（内部或外部链接）。尽管 C 不是面向对象的，但是对象的组织方式通常是很有用的。我们应该为那些不需要从指定文件外部访问的对象使用内部链接。通过封装，被保护的对象及其互斥量都可以使用内部链接。下面的例子说明了使用内部链接的方法。

下面的代码显示的是一个线程安全的计数器实例，在线程化程序中可以用这个计数器对引用进行计数。锁定机制隐藏在函数中，调用程序不用为使用互斥量而担心。变量 count 和 countlock 都具有 static 属性，因此只能从 counter.c 内部引用这些变量。根据 POSIX 线程

库的模式，如果成功，函数就返回 0，否则返回一个非零的错误码。

```
#include <pthread.h>
static int count = 0;
static pthread_mutex_t countlock = PTHREAD_MUTEX_INITIALIZER;

int increment(void)
{  /* increment the counter */
   int error;
   if (error = pthread_mutex_lock(&countlock))
      return error;
   count++;
   return pthread_mutex_unlock(&countlock);
}

int decrement(void)
{  /* decrement the counter */
   int error;
   if (error = pthread_mutex_lock(&countlock))
      return error;
   count--;
   return pthread_mutex_unlock(&countlock);
}

int getcount(int *countp)
{  /* retrieve the counter */
   int error;
   if (error = pthread_mutex_lock(&countlock))
      return error;
   *countp = count;
```

```
    return pthread_mutex_unlock(&countlock);
}
```

下面的代码显示了如何用互斥量保护不安全的库函数。POSIX 中把 C 库函数 rand 列为对线程化应用程序不安全的函数，而 randsafe 则线程安全：

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

int randsafe(double *ranp)
{
    static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
    int error;
    if (error = pthread_mutex_lock(&lock))
        return error;
    *ranp = (rand() + 0.5)/(RAND_MAX + 1.0);
    return pthread_mutex_unlock(&lock);
}
```

如果互斥量没有被静态初始化，程序就必须在使用所有其他的互斥函数之前调用 pthread_mutex_init。有些程序在创建其他线程之前有一个定义得很明确的初始化阶段，对这些程序来说，主线程可以用来执行初始化工作。但并不是所有的问题都适合这种结构。在任何线程访问互斥量之前调用 pthread_mutex_init 必须非常小心，让每个线程都去初始化互斥量的效果是未知的。

单次初始化的概念非常重要，POSIX 甚至提供了一个 pthread_once 函数来确保这个语义的实现。

5.2、条件变量

假设有两个变量 x 和 y 被多个线程共享，我们希望一个线程一直等到 x 和 y 相等为止。典型的不正确的忙等解决方法是：

while(x != y)

由于 x 和 y 是共享变量，因此线程在访问之前要先拥有互斥量，而一旦该线程拥有互斥量，则其他线程就无法访问共享变量 x 和 y，从而陷入死循环。

正确的非忙等策略是：

（1）获取互斥量；

（2）测试条件 x==y；

（3）如果为真，释放互斥量，并退出循环；

（4）如果为假，释放互斥量，将线程挂起。

为实现这种策略，我们需要一种新的数据类型，一种与等待 x==y 这样的任意条件为真的线程队列相关的数据类型，称其为条件变量（condition variable）。

POSIX 用 pthread_cond_t 类型的变量来表示条件变量。程序必须在使用条件变量之前对其进行初始化。对那些静态分配的、带有默认属性的条件变量来说，简单地将 PTHREAD_COND_INITIALIZER 赋给变量就可以完成初始化。对那些动态分配的或不具有默认属性的条件变量，就要用 pthread_cond_init 来执行初始化，如果将 NULL 传递给 attr，则表示用默认属性来初始化一个条件变量。否则就要用与线程属性对象类似的方法，先创建，然后再初始化一个条件变量属性对象。

```
SYNOPSIS
#incluide <pthread.h>
int pthread_cond_init(pthread_cond_t, *restrict cond,
                  const pthread_condattr_t *restrict attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;   POSIX:THR
```

如果成功，pthread_cond_init 就返回 0，否则返回一个非零的错误码。

函数 pthread_cond_destory 销毁由它的参数 cond 引用的条件变量。被销毁的条件变量可以重新初始化。

```
SYNOPSIS
#incluide <pthread.h>
int pthread_cond_destory(pthread_cond_t, *cond);    POSIX:THR
```

如果成功，函数返回 0，否则返回一个非零的错误码。

函数 pthread_cond_wait 将一个条件变量和一个互斥量作为参数，它原子地挂起调用线程并释放互斥量。可以认为它将线程放入了一个队列，队列中的线程都在等待条件发生变化的通知。线程收到通知时，函数会带着重新获得的互斥量返回，在继续执行之前，线程必须再次对条件进行测试。函数 pthread_cond_timedwait 可以用来等待一段有限的时间。

```
SYNOPSIS
#incluide <pthread.h>
int pthread_cond_timedwait(pthread_cond_t, *restrict cond,
                           pthread_mutex_t, *restrict mutex,
                       const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t, *restrict cond,
                      pthread_mutex_t, *restrict mutex);POSIX:THR
```

如果成功，函数返回 0，否则返回一个非零的错误码。如果 abstime 指定的时间到了，pthread_cond_timedwait 就返回 ETIMEOUT。

函数 pthread_cond_wait 只能由拥有互斥量的线程调用，当函数返回时，线程就再次拥有了互斥量。由于挂起的线程在调用 pthread_cond_wait 之前就拥有互斥量，并且在 pthread_cond_wait 之后也拥有互斥量，所以给人一种拥有的互斥量从未中断过的感觉，但实际上，在挂起的过程中，其他线程是可以获取互斥量的。

修改 x 或 y 的线程可以调用 pthread_cond_signal 函数将它所做的修改通知给其它线程。该函数将一个条件变量作为参数，并试图至少唤醒一个在相应队列中等待的线程。pthread_cond_broadcast 函数试图唤醒所有阻塞在相应队列中等待的线程。

```
SYNOPSIS
  #incluide <pthread.h>
  int pthread_cond_broadcast(pthread_cond_t, *cond);
  int pthread_cond_signal(pthread_cond_t, *cond);    POSIX:THR
```

如果成功，函数返回 0，否则返回一个非零的错误码。

下面的代码段显示了如何用 POSIX 条件变量 v 和互斥量 m 来等待条件 x==y：

```
pthread_mutex_lock(&m);
while(x != y)
   pthread_cond_wait(&v, &m);
/* modify x or y if nessary */
pthread_mutex_unlock(&m);
```

先拥有互斥量（此时就有可能被阻塞），然后判断条件不满足时，自己挂起在条件变量 v 对应得队列上，并释放互斥量 m（这样，别的线程才有机会修改 x 和 y），当别的线程修改

x 或 y 之后，就会调用 pthread_cond_signal 或者 pthread_cond_broadcast，从而唤醒该线程，此时，又回到 while 语句（因为即使别的线程修改了 x 或 y，也可能仍不满足条件），若满足则向下走（修改 x 或 y 后释放互斥量，或直接释放互斥量）；若不满足，继续挂起。

假设 v 是一个条件变量而 m 是一个互斥量。如果 test_condition()定义的断言为真，那么下面就是用条件变量来访问资源的一种正确方法。

```
static pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t v = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&m);
while(!test_condition())
   pthread_cond_wait(&v, &m);    /* get resource */
/* do critical section, possibly changing test_condition() */
pthread_cond_signal(&v);         /* inform another thread */
pthread_mutex_unlock(&m);
```

即如果修改了条件，就要通知其他线程（用 pthread_cond_signal 或 broadcast）。

下面的程序通过使用条件变量实现了一个线程安全的路障。变量 limit 说明了在从路障中释放线程之前，应该有多少个线程到达路障（执行 waitbarrier）。变量 count 说明当前有多少线程在路障上等待。这两个变量都用 static 属性声明，这样就只有通过 initbarrier 和 waitbarrier 才能对它们进行访问。如果成功，initbarrier 和 waitbarrier 函数就返回 0，否则返回一个非零的错误码。

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

static pthread_cond_t bcond = PTHREAD_COND_INITIALIZER;
static pthread_mutex_t bmutex = PTHREAD_MUTEX_INITIALIZER;
static int count = 0;
static int limit = 0;

int initbarrier(int n) /* initialize the barrier to be size n */
```

```
{
    int error;

    if (error = pthread_mutex_lock(&bmutex))
        return error;    /* couldn't lock give up */
    if (limit != 0){    /* barrier can only be initialized once */
        pthread_mutex_unlock(&bmutex);
        return EINVAL;
    }
    limit = n;
    return pthread_mutex_unlock(&bmutex);
}

int waitbarrier(void)
{  /* wait at the barrier until all n threads arrive */
    int berror = 0;
    int error;

    if (error = pthread_mutex_lock(&bmutex))
        return error;    /* couldn't lock, give up */
    if (limit <= 0){    /* make sure barrier initialized */
        ptherad_mutex_unlock(&bmutex);
        return EINVAL;
    }
    count++;
    while ((count < limit) && (!berror))
        berror = pthread_cond_wait(&bcond, &bmutex);
    if (!berror)
        berror = pthread_cond_broadcast(&bcond);/* wake up all */
```

```
        error = pthread_mutex_unlock(&bmutex);
        if (berror)
            return berror;
        return error;
    }
```

练习 6-2：试编写一程序，能够演示出上面路障的例子。

5.3、读者和写者

读-写者问题指的是这样一种情况，在这种情况下，允许对资源进行两种类型的访问（读和写），一种类型的访问必须确保是排他的（如写操作），但是另一种类型的访问可以是共享的（如读操作）。例如，任意数量的进程都可以毫无困难的从同一个文件中读出，但一次只能有一个进程对文件进行修改。

处理读-写者同步的两种常见的策略被称为强读者同步和强写者同步。在强读者同步中，总是给读者以优先权，只要写者当前没有进行写操作，读者就可以获得访问权。在强写者同步中，通常将优先权交给写者，而将读者延迟到所有等待的或活动的写者都完成了为止。例如，由于读者都需要最新的信息，所以航线预定系统优先使用强写者同步；另一方面，图书馆的参考数据可能希望将优先权赋予读者。

POSIX 读-写锁由 pthread_rwlock_t 类型的变量表示。程序在用 pthread_rwlock_t 变量进行同步之前，必须调用 pthread_rwlock_init 来初始化这个变量。参数 rwlock 是一个指向读-写锁的指针。将 NULL 传递给 pthread_rwlock_init 的参数 attr，以便用默认属性来初始化读-写锁。否则，就要使用与线程属性对象类似的方法，先创建，然后再初始化读-写锁属性对象。

```
SYNOPSIS
  #incluide <pthread.h>
  int pthread_rwlock_init(pthread_rwlock_t, *restrict rwlock,
        const pthread_rwlockattr_t *restrict attr);
  int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);POSIX:THR
```

如果成功，函数返回 0，否则返回一个非零的错误码。

pthread_rwlock_destroy 函数销毁了它的参数引用的读-写锁。参数 rwlock 是一个指向读-写锁的指针，销毁了的读-写锁可以重新进行初始化。

pthread_rwlock_rdlock 和 pthread_rwlock_tryrdlock 函数允许线程为读操作获取一个读-写锁，pthread_rwlock_wrlock 和 pthread_rwlock_trywrlock 函数允许线程为写操作获取一个读-写锁。pthread_rwlock_unlock 函数会将读-写锁释放掉。这些函数要求将一个指向读-写锁的指针作为参数传递。

```
SYNOPSIS
  #incluide <pthread.h>
  int pthread_rwlock_rdlock(pthread_rwlock_t, *rwlock);
  int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
  int pthread_rwlock_wrlock(pthread_rwlock_t, *rwlock);
  int pthread_rwlock_trywrlock(pthread_rwlock_t, *rwlock);
  int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);POSIX:THR
```

如果成功，这些函数就返回 0，否则返回一个非零的错误码。如果因为锁已经被持有而无法获取，两个 try 函数就返回 EBUSY。

6、信号处理与线程

进程中所有线程都共享进程的信号处理程序，但每个线程都有它自己的信号掩码。由于线程的操作可以异步于信号，所以线程与信号的交互会比较复杂，下表总结了三种类型的信号及其相应的传递方法。

类型	传递动作
异步	传递给某些解除了对该信号阻塞的线程
同步	传递给引发（该信号）的线程
定向的	传递给标识了的线程（pthread_kill）

异步信号在不可预测的时间产生，也不与特定的线程相关。因为信号处理程序是进程范围的，也就是说如果某进程有信号产生，则进程中的线程都可能会遇到该信号，一般情况下，线程运行系统从解除阻塞信号的线程中挑选一个来处理信号。

下面的程序演示了对异步信号的线程化处理。函数 signal_function 是信号处理程序，该函数只输出当前线程的线程 ID。函数 thread_function 是线程执行体，只是不停的睡眠。主程序先安装信号 SIGUSR1 的信号处理程序，然后产生 10 个线程并输出线程 ID，最后输出

主线程 ID，然后不停的睡眠。由于没有做任何有关信号阻塞的操作，因此主线程及其其它 10 个线程都继承了进程对所有信号的缺省响应。编译运行程序，在另一个终端用命令

ps aux

查看该程序的进程 ID，然后用命令

ps －s SIGUSR1 进程 ID

向该进程传送 SIGUSR1 信号，则结果显示，在这种情况下，线程运行系统总是将信号传递给主线程。

```
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

static void signal_function(int signo)
{
   printf("thread ID is %u\n", pthread_self());
}

void *thread_function(void *arg)
{
   while(1)
       sleep(*((int *)(arg)));
}

int main(void)
{
   struct sigaction act;
   pthread_t a_thread;
   int i;
```

```
    act.sa_handler = signal_function;
    act.sa_flags = 0;
    if ((sigemptyset(&act.sa_mask) == -1) ||
        (sigaction(SIGUSR1, &act, NULL) == -1)){
        perror("Failed to set SIGUSR1 handles");
        return 1;
    }
    for (i = 0; i < 10; i++){
        if (pthread_create(&a_thread, NULL, thread_function, &i)){
                  perror("Thread creation failed");
                  return 1;
           }
        fprintf(stderr,"%u\n", a_thread);
    }
    fprintf(stderr, "%u\n", pthread_self());
    while(1){
        sleep(1);
    }
    return 0;
}
```

如果进程在创建 10 个线程之后阻塞 SIGUSR1 信号，即让主线程阻塞信号 SIGUSR1。则运行结果显示线程运行系统顺序选择 10 个线程中的第一个线程传递信号。程序如下：

```
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

static void signal_function(int signo)
{
```

```
    printf("thread ID is %u\n", pthread_self());
}

void *thread_function(void *arg)
{
    while(1)
        sleep(*((int *)(arg)));
}

int main(void)
{
    struct sigaction act;
    sigset_t newsigset;
    pthread_t a_thread;
    int i;

    act.sa_handler = signal_function;
    act.sa_flags = 0;
    if ((sigemptyset(&act.sa_mask) == -1) ||
        (sigaction(SIGUSR1, &act, NULL) == -1)){
        perror("Failed to set SIGUSR1 handles");
        return 1;
    }
    for (i = 0; i < 10; i++){
        if (pthread_create(&a_thread, NULL, thread_function, &i)){
                perror("Thread creation failed");
                return 1;
           }
        fprintf(stderr,"%u\n", a_thread);
```

```
    }
    fprintf(stderr, "%u\n", pthread_self());
    if ((sigemptyset(&newsigset) == -1) ||
        (sigaddset(&newsigset, SIGUSR1) == -1)){
        perror("Failed to initialize the signal mask");
        return 1;
    }
    if (sigprocmask(SIG_BLOCK, &newsigset, NULL) == -1){
        perror("Failed to set mask for SIGUSR1");
        return 1;
    }
    while(1){
        sleep(1);
    }
    return 0;
}
```

显然如果进程在创建线程之前就阻塞 SIGSUR1 信号，则所有的线程都不理睬该信号。请修改程序验证之。

同步信号的产生是可以预测的。例如 SIGFPE（浮点异常）这样的信号就是同步于引发它们的线程的（也就是说，它们通常在线程执行的什么位置上产生是可以预知的）。下面的程序帮助我们理解同步信号的概念。程序首先安装信号 SIGFPE 的信号处理程序，因为是浮点异常，该信号处理程序在输出线程 ID 之后，知趣的线程退出。主线程然后创建 10 个线程，每个线程在睡眠适当的时间（1 秒、11 秒、21 秒、…、91 秒）后，主动产生 SIGFPE 信号。主线程最后等待所有其它线程后结束程序：

```
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
```

```
static void signal_function(int signo)
{
   printf("thread ID is %u\n", pthread_self());
   pthread_exit(NULL);
}

void *thread_function(void *arg)
{
   int x = 10, y = 0;
      sleep(10*(*((int *)(arg))) + 1);
      x = x / y;
}

int main(void)
{
   struct sigaction act;
   pthread_t a_thread[10];
   int i;

   act.sa_handler = signal_function;
   act.sa_flags = 0;
   if ((sigemptyset(&act.sa_mask) == -1) ||
      (sigaction(SIGFPE, &act, NULL) == -1)){
      perror("Failed to set SIGFPE handles");
      return 1;
   }
   for (i = 0; i < 10; i++){
      if (pthread_create(&a_thread[i], NULL, thread_function,
&i)){
```

```
                perror("Thread creation failed");
                return 1;
            }
        fprintf(stderr,"%u\n", a_thread[i]);
    }
    fprintf(stderr, "%u\n", pthread_self());
    for (i = 0; i < 10; i++)
        pthread_join(a_thread[i], NULL);
    return 0;
}
```

pthread_kill 函数要求产生信号码为 sig 的信号，并将其传送到 thread 指定的线程中去。

```
SYNOPSIS
  #incluide <pthread.h>
  int pthread_kill(pthread_t thread, int sig);      POSIX:THR
```

如果成功，函数就返回 0，否则返回一个非零的错误码。

下面的代码段会使线程将它自己和整个进程都杀死：

```
if (pthread_kill(pthread_self(), SIGKILL))
    fprintf(stderr, “Failed to commit suicide\n”);
```

上例说明了一个与 pthread_kill 有关的重要的问题，尽管 pthread_kill 将信号传递给了一个特定的线程，但处理信号的行为可能会影响整个进程。一种常见的概念混淆是假定 pthread_kill 总是会使进程终止的，但实际上并不是这样的。pthread_kill 仅仅为线程产生了一个信号，由于 SIGKILL 不能被捕捉、阻塞或忽略，所以才使进程终止。除非进程忽略、阻塞或捕捉到信号，否则任何默认行为是终止进程的信号都回终止进程。

下面的程序是一个 pthread_kill 函数的例子，运行程序，可以看到 SIGUSR1 被传递到 a[i]线程中：

```
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
```

```
static void signal_function(int signo)
{
   printf("thread ID is %u\n", pthread_self());
}

void *thread_function(void *arg)
{
   while(1)
      sleep(10*(*((int *)(arg))));
}

int main(void)
{
   struct sigaction act;
   pthread_t a_thread;
   int i;
   unsigned long int a[10];

   act.sa_handler = signal_function;
   act.sa_flags = 0;
   if ((sigemptyset(&act.sa_mask) == -1) ||
      (sigaction(SIGUSR1, &act, NULL) == -1)){
      perror("Failed to set SIGUSR1 handles");
      return 1;
   }
   for (i = 0; i < 10; i++){
      if (pthread_create(&a_thread, NULL, thread_function, &i)){
               perror("Thread creation failed");
```

```
                return 1;
            }
        fprintf(stderr,"%u\n", a_thread);
        a[i] = a_thread;
    }
    fprintf(stderr, "%u\n", pthread_self());
    pthread_kill(a[6], SIGUSR1);
    while(1){
        sleep(1);
    }
    return 0;
}
```

虽然信号处理程序是进程范围的，但是每个线程都有它自己的信号掩码，线程可以通过 pthread_sigmask 函数来检查或设置它的信号掩码，这个函数是 sigprocmask 在线程化程序中的推广。当进程中有多个线程时，就不应该使用 sigprocmask 函数，但在创建其它的线程之前，它可以被主线程调用。参数 how 和 set 指出了修改信号掩码的方式。如果 oset 不为 NULL，pthread_sigmask 函数就将*oset 设置为线程的前一个信号掩码。how 的值 SIG_SETMASK 会使线程现在阻塞 set 中所有的信号，而不阻塞任何其它的信号。SIG_BLOCK 使线程阻塞 set 中的信号（添加到线程当前的信号掩码中），SIG_UNBLOCK 从线程当前的信号掩码中将 set 中当前被阻塞的信号删除（不再阻塞了）。

```
SYNOPSIS
  #incluide <pthread.h>
  #include <signal.h>
  int pthread_sigmask(int how, const sigset_t *restrict set,
                      sigset_t *restrict oset);    POSIX:THR
```

如果成功，函数就返回 0，否则返回一个非零的错误码。如果 how 无效，返回 EINVAL。

信号处理程序是进程范围的，与在单线程的进程中一样，可以用 sigaction 调用来安装它们。在线程化程序中，进程范围的信号处理程序和线程特定的信号掩码之间的区别是很重要的。在多线程的进程中进行信号处理的一种推荐策略是：为信号处理程序使用特定的线程。

主线程在创建线程之前阻塞所有的信号，由于信号掩码是从创建线程的线程中继承的，因此所有的线程将同样阻塞所有信号。然后，专门用来处理信号的线程对那个信号执行 sigwait，或者线程可以用 pthread_sigmask 来解除对信号的阻塞。使用 sigwait 的好处是线程不局限于异步信号安全的函数。