第一章 线程基础知识

一．什么是线程

在一个程序里的多个执行路线就叫做线程。更准确的定义是：线程是“一个进程内部的一个控制序列”。

典型的 unix 进程可以看成只有一个控制线程：一个进程在同一时刻只做一件事情。有了多个控制线程以后，在程序设计时可以把进程设计成在同一时刻能够做不止一件事，每个线程处理各只独立的任务。

二．线程的优点

（1） 通过为每种事件类型的处理分配单独的线程，能够简化处理异步时间的代码。

（2） 多个线程可以自动共享相同的存储地址空间和文件描述符。

（3） 有些问题可以通过将其分解从而改善整个程序的吞吐量。

（4） 交互的程序可以通过使用多线程实现相应时间的改善，多线程可以把程序中处理用户输入输出的部分与其它部分分开。

三．线程的缺点

线程也有不足之处。编写多线程程序需要更全面更深入的思考。在一个多线程程序里，因时间分配上的细微偏差或者因共享了不该共享的变量而造成不良影响的可能性是很大的。调试一个多线程程序也比调试一个单线程程序困难得多

四．线程的结构

线程包含了表示进程内执行环境必需的信息，其中包括进程中标识线程的线程 ID，一组寄存器值、栈、调度优先级和策略、信号屏蔽子，errno 变量以及线程私有数据。进程的所有信息对该进程的所有线程都是共享的，包括可执行的程序文本，程序的全局内存和堆内存、栈以及文件描述符。

五．线程标识

就像每个进程有一个进程 ID 一样，每个线程也有一个线程 ID，进程 ID 在整个系统中是唯一的，但线程不同，线程 ID 只在它所属的进程环境中有效。线程 ID 用 pthread_t 数据类型来表示，实现的时候可以用一个结构来代表 pthread_t 数据类型，所以可以移植的操作系统不能把它作为整数处理。因此必须使用函数来对来对两个线程 ID 进行比较。

1．

名称::	pthread_equal
功能:	比较两个线程 ID
头文件:	#include <pthread.h>
函数原形:	int pthread_equal(pthread_t tid1,pthread_t tid2);
参数:	tid1 进程 1id tid2 进程 2id
返回值:	若相等返回非 0 值，否则返回 0

dddddd

2.

dd

名称::	pthread_self
功能:	获取自身线程的 id
头文件:	#include <pthread.h>
函数原形:	pthread_t pthread_self(void);
参数:	无
返回值:	调用线程的线程 id

六．线程的创建

3.

名称::	pthread_create
功能:	创建线程
头文件:	#include <pthread.h>
函数原形:	int pthread_create(pthread_t *restrict tidp,const pthread _attr_t *restrict attr,void *(*start_rtn)(void),void *restrict arg);
参数:	
返回值:	若成功返回则返回 0，否则返回错误编号

当 pthread_creat 成功返回时， tidp 指向的内存单元被设置为新创建线程的线程 ID。attr 参数用于定制各种不同的线程属性。可以把它设置为 NULL，创建默认的线程属性。

新创建的线程从 start_rtn 函数的地址开始运行，该函数只有一个无类型指针参数 arg,如果需要向 start_rtn 函数传递的参数不止一个，那么需要把这些参数放到一个结构中，然后把这个结构的地址作为 arg 参数传入。

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

```
void printids(const char *s)
{
      printf("%s pid:%u  tid:%u \n", s,getpid(),pthread_self());
}

void *thr_fn(void *arg)
{
      printids("new thread: ");
}

int main()
{
      int err;
      pthread_t tid;
      err=pthread_create(&tid,NULL,thr_fn,NULL);
      if(err=0)
          printf("can't create thread:%s\n",strerror(err));
      printids("main thread: ");
      sleep(1);
      exit(0);
}
```

关于进程的编译我们都要加上参数 –lpthread 否则提示找不到函数的错误。
具体编译方法是 cc –lpthread –o gettid gettid.c
运行结果为
main thread: pid 14954 tid 134529024
new thread: pid 14954 tid 134530048

七．.线程的终止

线程是依进程而存在的，当进程终止时，线程也就终止了。当然也有在不终止整个进程的情况下停止它的控制流。

（1）线程只是从启动例程中返回，返回值是线程的退出码。
（2）县城可以被同一进程中的其他线程取消。
（3）线程调用 pthread_exit.

4．

名称::	pthread_exit
功能:	终止一个线程
头文件:	#include <pthread.h>
函数原形:	void pthread_exit(void *rval_ptr);
参数:	
返回值:	无

rval_prt 是一个无类型指针，与传给启动例程的单个参数类似。进程中的其他线程可以调用 pthread_join 函数访问到这个指针。

名称::	pthread_join
功能:	获得进程的终止状态
头文件:	#include <pthread.h>
函数原形:	int pthread_join(pthread_t thread,void **rval_ptr);
参数:	
返回值:	若成功返回 0，否则返回错误编号。

5.

当一个线程通过调用 pthread_exit 退出或者简单地从启动历程中返回时，进程中的其他线程可以通过调用 pthread_join 函数获得进程的退出状态。调用 pthread_join 进程将一直阻塞，直到指定的线程调用 pthread_exit,从启动例程中或者被取消。
如果线程只是从它的启动历程返回，rval_ptr 将包含返回码。

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

void *thr_fn1(void *arg)
{
	printf(“thread 1 returning\n”);
	return((void *)1);
}

void *thr_fn2(void *arg)
{
	printf(“thread 2 exiting\n”);
	return((void *)2);
}

int main()
{
	pthread_t tid1,tid2;
	void *tret;

	pthread_create(&tid1,NULL,thr_fn1,NULL);
	pthread_create(&tid2,NULL,thr_fn2,NULL);
	pthread_join(tid1,&tret);
	printf(“thread 1 exit code %d\n”,(int)tret);
```

```
    pthread_join(tid2,&tret);
    printf("thread 2 exit code %d\n",(int)tret);
    exit(0);
}
```

运行结果是:

```
thread 1 returning
thread 2 exiting
thread 1 exit code 1
thread 2 exit code 2
```

6.

名称::	pthread_detach
功能:	使线程进入分离状态。
头文件:	#include <pthread.h>
函数原形:	int pthread_detach(pthread_t tid);
参数:	
返回值:	若成功则返回 0，否则返回错误编号。

在默认情况下，线程的终止状态会保存到对该线程调用 pthread_join,如果线程已经处于分离状态，线程的底层存储资源可以在线程终止时立即被收回。当线程被分离时，并不能用 pthread_join 函数等待它的终止状态。对分离状态的线程进行 pthread_join 的调用会产生失败，返回 EINVAL.pthread_detach 调用可以用于使线程进入分离状态。

7.

名称::	pthread_cancel
功能:	取消同一进程中的其他线程
头文件:	#include <pthread.h>
函数原形:	int pthread_cancel(pthread_t tid);
参数:	tid 线程 id
返回值:	若成功返回 0，否则返回错误编号。

在默认的情况下，pthread_cancel 函数会使由 tid 标识的线程的行为表现为如同调用了参数为 PTHEAD_CANCELED 的 pthread_exit 函数，但是，线程可以选择忽略取消方式和控制取消方式。pthread_cancel 并不等待线程终止，它仅仅提出请求。

名称::	pthread_cancel_push/ pthread_cancel_push_pop
功能:	线程清理处理程序
头文件:	#include <pthread.h>
函数原形:	void pthread_cancel_push(void (*rtn)(void *)，void *arg); void pthread_cancel_pop(int execute);
参数:	rtn 处理程序入口地址 arg 传递给处理函数的参数
返回值:	无

8.

线程可以安排它退出时需要调用的函数，这样的函数称为线程清理处理程序，线程可以建立多个清理处理程序。处理程序记录在栈中，也就是说它们的执行顺序与它们注册时的顺序相反。

要注意的是如果线程是通过从他的启动例程中返回而终止的，它的处理程序就不会调用。还要注意清理处理程序是按照与它们安装时相反的顺序调用的。

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

void cleanup(void *arg)
{
   printf(“cleanup: %s\n”,(char *)arg);
}

void *thr_fn(void *arg) /*线程入口地址*/
{
    printf(“thread start\n”);
    pthread_cleanup_push(cleanup,”thread first handler”);/*设置第一个线程处理程序*/
    pthread_cleanup_push(cleanup,”thread second handler”); /*设置第二个线程处理程序*/
    printf(“thread push complete\n”);
    pthread_cleanup_pop(0); /*取消第一个线程处理程序*/
    pthread_cleanup_pop(0); /*取消第二个线程处理程序*/
}

int main()
{
    pthread_t tid;
    void *tret;
```

```
    pthread_creat(&tid,NULL,thr_fn,(void *)1); /*创建一个线程*/
    pthread_join(tid,&tret); /*获得线程终止状态*/
    ptinrf("thread exit code %d\n",(int)tret);
}
```

八、一次性初始化

有时候我们需要对一些 posix 变量只进行一次初始化，如线程键（我下面会讲到）。如果我们进行多次初始化程序就会出现错误。

在传统的顺序编程中，一次性初始化经常通过使用布尔变量来管理。控制变量被静态初始化为 0，而任何依赖于初始化的代码都能测试该变量。如果变量值仍然为 0，则它能实行初始化，然后将变量置为 1。以后检查的代码将跳过初始化。

但是在多线程程序设计中，事情就变的复杂的多。如果多个线程并发地执行初始化序列代码，2 个线程可能发现控制变量为 0，并且都实行初始化，而该过程本该仅仅执行一次。初始化的状态必须由互斥量保护。

如果我们需要对一个 posix 变量静态的初始化，可使用的方法是用一个互斥量对该变量的初始话进行控制。但有时候我们需要对该变量进行动态初始化，pthread_once 就会方便的多。

9.

名称:	pthread_once
功能:	一次性初始化
头文件:	#include <pthread.h>
函数原形:	pthread_once_t once_control=PTHREAD_ONCE_INIT; int pthread_once(pthread_once_t *once_control,void(*init_routine)(void));
参数:	once_control 控制变量 init_routine 初始化函数
返回值:	若成功返回 0，若失败返回错误编号。

类型为 pthread_once_t 的变量是一个控制变量。控制变量必须使用 PTHREAD_ONCE_INIT 宏静态地初始化。

pthread_once 函数首先检查控制变量，判断是否已经完成初始化，如果完成就简单地返回；否则，pthread_once 调用初始化函数，并且记录下初始化被完成。如果在一个线程初始时，另外的线程调用 pthread_once，则调用线程等待，直到那个现成完成初始话返回。

下面就是该函数的程序例子:

```
#include <pthread.h>

pthread_once_t once=PTHREAD_ONCE_INIT;
```

```
pthread_mutex_t mutex;   /*互斥量，我们后面会讲到*/

void once_init_routine(void)   /*一次初始化函数*/
{
   int status;
     status=pthread_mutex_init(&mutex,NULL);/*初始化互斥量*/
     if(status==0)
        printf("Init success!,My id is %u",pthread_self());
}

void *child_thread(void *arg)
{
     printf("I'm child ,My id is %u",pthread_self());
   pthread_once(&once,once_init_routine); /*子线程调用一次性初始化函数*/
}

int main(int argc,char *argv[ ])
{
     pthread_t child_thread_id;

     pthread_create(&child_thread_id,NULL,child_thread,NULL);/*创建子线程*/
     printf("I'm father,my id is %u",pthread_self());
     pthread_once(&once,once_init_routine);/*父线程调用一次性初始化函数*/
     pthread_join(child_thread_id,NULL);
}
```

程序运行结果如下：

```
./once
I'm father,My id is 3086874304
Init success!,My id is 3086874304
I'm child, My id is 3086871472
```

从上面的结果可以看到当主函数初始化成功后，子函数初始化失败。

九、线程的私有数据

在进程内的所有线程共享相同的地址空间，任何声明为静态或外部的变量，或在进程堆声明的变量，都可以被进程所有的线程读写。那怎样才能使线程序拥有自己的私有数据呢。

posix 提供了一种方法，创建线程键。

10.

名称::	pthread_key_create
功能:	建立线程私有数据键
头文件:	#include <pthread.h>
函数原形:	int pthread_key_create(pthread_key_t *key,void(*destructor)(void *));
参数:	key 私有数据键 destructor 清理函数
返回值:	若成功返回 0，若失败返回错误编号。

第一个参数为指向一个键值的指针，第二个参数指明了一个 destructor 函数（清理函数），如果这个参数不为空，那么当每个线程结束时，系统将调用这个函数来释放绑定在这个键上的内存块。这个函数常和函数 pthread_once 一起使用，为了让这个键只被创建一次。函数 pthread_once 声明一个初始化函数，第一次调用 pthread_once 时它执行这个函数，以后的调用将被它忽略。

下面是程序例子:

```
#include <pthread.h>

pthread_key_t tsd_key;
pthread_once_t key_once=PTHREAD_ONCE_INIT;

void once_routine(void)
{
    int status;

    status=pthread_key_create(&tsd_key,NULL);/*初始化线程私有数据
键*/
    if(status=0)
       printf("Key create success! My id is %u\n",pthread_self());
}

void *child_thread(void *arg)
{
    printf("I'm child,My id is %u\n",pthread_self());
    pthread_once(&key_once,once_routine);/* 调用一次性初始化函数*/
}

int main(int argc,char *argv[ ])
{
    pthread_t child_thread_id;

    pthread_create(&child_thread_id,NULL,child_thread,NULL);
```

```
    printf("I'm father,my id is%u\n",pthread_self());
    pthread_once(&key_once,once_routine);
}
```

程序运行结果如下：

I'm father,My id is 3086231232
Key create success! My id is 3086231232
I'm child,My id is 2086228400

第二章 线程高级知识

一．线程属性

线程具有属性，用 pthread_attr_t 表示，在对该结构进行处理之前必须进行初始化，在使用后需要对其去除初始化。我们用 pthread_attr_init 函数对其初始化，用 pthread_attr_destroy 对其去除初始化。

1．

名称:：	pthread_attr_init/pthread_attr_destroy
功能：	对线程属性初始化/去除初始化
头文件：	#include <pthread.h>
函数原形：	int pthread_attr_init(pthread_attr_t *attr); int pthread_attr_destroy(pthread_attr_t *attr);
参数：	Attr 线程属性变量
返回值：	若成功返回 0，若失败返回-1。

调用 pthread_attr_init 之后，pthread_t 结构所包含的内容就是操作系统实现支持的线程所有属性的默认值。

如果要去除对 pthread_attr_t 结构的初始化，可以调用 pthread_attr_destroy 函数。如果 pthread_attr_init 实现时为属性对象分配了动态内存空间，pthread_attr_destroy 还会用无效的值初始化属性对象，因此如果经 pthread_attr_destroy 去除初始化之后的 pthread_attr_t 结构被 pthread_create 函数调用，将会导致其返回错误。

线程属性结构如下：

```
typedef struct
{
    int                 detachstate;   线程的分离状态
    int                 schedpolicy;  线程调度策略
    struct sched_param          schedparam;  线程的调度参数
    int                 inheritsched;  线程的继承性
    int                  scope;       线程的作用域
    size_t               guardsize;   线程栈末尾的警戒缓冲区大小
    int                  stackaddr_set;
```

```
    void *          stackaddr;  线程栈的位置
    size_t          stacksize;  线程栈的大小
}pthread_attr_t;
```

每个个属性都对应一些函数对其查看或修改。下面我们分别介绍。

二、线程的分离状态

线程的分离状态决定一个线程以什么样的方式来终止自己。在默认情况下线程是非分离状态的，这种情况下，原有的线程等待创建的线程结束。只有当 pthread_join（）函数返回时，创建的线程才算终止，才能释放自己占用的系统资源。

而分离线程不是这样子的，它没有被其他的线程所等待，自己运行结束了，线程也就终止了，马上释放系统资源。程序员应该根据自己的需要，选择适当的分离状态。所以如果我们在创建线程时就知道不需要了解线程的终止状态，则可以 pthread_attr_t 结构中的 detachstate 线程属性，让线程以分离状态启动。

名称::	pthread_attr_getdetachstate/pthread_attr_setdetachstate
功能:	获取/修改线程的分离状态属性
头文件:	#include <pthread.h>
函数原形:	int pthread_attr_getdetachstate(const pthread_attr_t * attr,int *detachstate); int pthread_attr_setdetachstate(pthread_attr_t *attr,int detachstate);
参数:	Attr 线程属性变量 Detachstate 线程的分离状态属性
返回值:	若成功返回 0，若失败返回-1。

2.

可以使用 pthread_attr_setdetachstate 函数把线程属性 detachstate 设置为下面的两个合法值之一：设置为 PTHREAD_CREATE_DETACHED,以分离状态启动线程；或者设置为 PTHREAD_CREATE_JOINABLE,正常启动线程。可以使用 pthread_attr_getdetachstate 函数获取当前的 datachstate 线程属性。

以分离状态创建线程

```
#iinclude <pthread.h>

void *child_thread(void *arg)
{
    printf("child thread run!\n");
}

int main(int argc,char *argv[ ])
{
    pthread_t tid;
    pthread_attr_t attr;
```

```
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
    pthread_create(&tid,&attr,fn,arg);
    pthread_attr_destroy(&attr);
    sleep(1);
}
```

三、线程的继承性

函数 pthread_attr_setinheritsched 和 pthread_attr_getinheritsched 分别用来设置和得到线程的继承性，这两个函数的定义如下：

3.

名称:：	pthread_attr_getinheritsched pthread_attr_setinheritsched
功能：	获得/设置线程的继承性
头文件：	#include <pthread.h>
函数原形：	Int pthread_attr_getinheritsched(const pthread_attr_t *attr,int *inheritsched); int pthread_attr_setinheritsched(pthread_attr_t *attr,int inheritsched);
参数：	attr 线程属性变量 inheritsched 线程的继承性
返回值：	若成功返回 0，若失败返回-1。

这两个函数具有两个参数，第 1 个是指向属性对象的指针，第 2 个是继承性或指向继承性的指针。继承性决定调度的参数是从创建的进程中继承还是使用在 schedpolicy 和 schedparam 属性中显式设置的调度信息。Pthreads 不为 inheritsched 指定默认值，因此如果你关心线程的调度策略和参数，必须先设置该属性。

继承性的可能值是 PTHREAD_INHERIT_SCHED（表示新线程将继承创建线程的调度策略和参数）和 PTHREAD_EXPLICIT_SCHED（表示使用在 schedpolicy 和 schedparam 属性中显式设置的调度策略和参数）。

如果你需要显式的设置一个线程的调度策略或参数，那么你必须在设置之前将 inheritsched 属性设置为 PTHREAD_EXPLICIT_SCHED.

下面我来讲进程的调度策略和调度参数。我会结合下面的函数给出本函数的程序例子。

四、线程的调度策略

函数 pthread_attr_setschedpolicy 和 pthread_attr_getschedpolicy 分别用来设置和得到线程的调度策略。

4.

名称::	pthread_attr_getschedpolicy pthread_attr_setschedpolicy
功能:	获得/设置线程的调度策略
头文件:	#include <pthread.h>
函数原形:	int pthread_attr_getschedpolicy(const pthread_attr_t *attr,int *policy); int pthread_attr_setschedpolicy(pthread_attr_t *attr,int policy);
参数:	attr 线程属性变量 policy 调度策略
返回值:	若成功返回 0，若失败返回-1。

这两个函数具有两个参数，第 1 个参数是指向属性对象的指针，第 2 个参数是调度策略或指向调度策略的指针。调度策略可能的值是先进先出（SCHED_FIFO）、轮转法（SCHED_RR）,或其它（SCHED_OTHER）。

SCHED_FIFO 策略允许一个线程运行直到有更高优先级的线程准备好，或者直到它自愿阻塞自己。在 SCHED_FIFO 调度策略下，当有一个线程准备好时，除非有平等或更高优先级的线程已经在运行，否则它会很快开始执行。

SCHED_RR(轮循)策略是基本相同的，不同之处在于：如果有一个 SCHED_RR
策略的线程执行了超过一个固定的时期(时间片间隔)没有阻塞，而另外的 SCHED_RR 或 SCHBD_FIPO 策略的相同优先级的线程准备好时，运行的线程将被抢占以便准备好的线程可以执行。

当有 SCHED_FIFO 或 SCHED_RR 策赂的线程在一个条件变量上等持或等持加锁同一个互斥量时，它们将以优先级顺序被唤醒。即，如果一个低优先级的 SCHED_FIFO 线程和一个高优先织的 SCHED_FIFO 线程都在等待锁相同的互斥且，则当互斥量被解锁时，高优先级线程将总是被首先解除阻塞。

五、线程的调度参数

函数 pthread_attr_getschedparam 和 pthread_attr_setschedparam 分别用来设置和得到线程的调度参数。

5.

名称::	pthread_attr_getschedparam pthread_attr_setschedparam
功能:	获得/设置线程的调度参数
头文件:	#include <pthread.h>
函数原形:	int pthread_attr_getschedparam(const pthread_attr_t *attr,struct sched_param *param); int pthread_attr_setschedparam(pthread_attr_t *attr,const struct sched_param *param);
参数:	attr 线程属性变量 param sched_param 结构
返回值:	若成功返回 0，若失败返回-1。

这两个函数具有两个参数，第 1 个参数是指向属性对象的指针，第 2 个参数是 sched_param 结构或指向该结构的指针。结构 sched_param 在文件/usr/include /bits/sched.h 中定义如下:

```
struct sched_param
{
        int sched_priority;
};
```

结构 sched_param 的子成员 sched_priority 控制一个优先权值，大的优先权值对应高的优先权。系统支持的最大和最小优先权值可以用 sched_get_priority_max 函数和 sched_get_priority_min 函数分别得到。

注意：如果不是编写实时程序，不建议修改线程的优先级。因为，调度策略是一件非常复杂的事情，如果不正确使用会导致程序错误，从而导致死锁等问题。如：在多线程应用程序中为线程设置不同的优先级别，有可能因为共享资源而导致优先级倒置。

名称::	sched_get_priority_max sched_get_priority_min
功能:	获得系统支持的线程优先权的最大和最小值
头文件:	#include <pthread.h>
函数原形:	int sched_get_priority_max(int policy); int sched_get_priority_min(int policy);
参数:	policy 系统支持的线程优先权的最大和最小值
返回值:	若成功返回 0，若失败返回-1。

6.

下面是上面几个函数的程序例子：

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

void *child_thread(void *arg)
{
    int policy;
    int max_priority,min_priority;
    struct sched_param param;
    pthread_attr_t attr;

    pthread_attr_init(&attr); /*初始化线程属性变量*/
    pthread_attr_setinheritsched(&attr,PTHREAD_EXPLICIT_SCHED); /*设置线
程继承性*/
    pthread_attr_getinheritsched(&attr,&policy); /*获得线程的继承性*/
    if(policy==PTHREAD_EXPLICIT_SCHED)
        printf("Inheritsched:PTHREAD_EXPLICIT_SCHED\n");
    if(policy==PTHREAD_INHERIT_SCHED)
        printf("Inheritsched:PTHREAD_INHERIT_SCHED\n");

    pthread_attr_setschedpolicy(&attr,SCHED_RR);/*设置线程调度策略*/
    pthread_attr_getschedpolicy(&attr,&policy);/*取得线程的调度策略*/
    if(policy==SCHED_FIFO)
        printf("Schedpolicy:SCHED_FIFO\n");
    if(policy==SCHED_RR)
        printf("Schedpolicy:SCHED_RR\n");
    if(policy==SCHED_OTHER)
        printf("Schedpolicy:SCHED_OTHER\n");

    sched_get_priority_max(max_priority);/*获得系统支持的线程优先权的最大值
*/
    sched_get_priority_min(min_priority);/* 获得系统支持的线程优先权的最小值
*/
    printf("Max priority:%u\n",max_priority);
    printf("Min priority:%u\n",min_priority);

    param.sched_priority=max_priority;
    pthread_attr_setschedparam(&attr,&param);/*设置线程的调度参数*/
    printf("sched_priority:%u\n",param.sched_priority);/*获得线程的调度参数*/
    pthread_attr_destroy(&attr);
}
```

```
int main(int argc,char *argv[ ])
{
    pthread_t child_thread_id;

    pthread_create(&child_thread_id,NULL,child_thread,NULL);
    pthread_join(child_thread_id,NULL);
}
```

六、线程的作用域

函数 pthread_attr_setscope 和 pthread_attr_getscope 分别用来设置和得到线程的作用域，这两个函数的定义如下：

7.

名称::	pthread_attr_setscope pthread_attr_getscope
功能:	获得/设置线程的作用域
头文件:	#include <pthread.h>
函数原形:	int pthread_attr_setscope(pthread_attr_t *attr,int scope); int pthread_attr_getscope(const pthread_attr_t *attr,int *scope);
参数:	attr 线程属性变量 scope 线程的作用域
返回值:	若成功返回 0，若失败返回-1。

这两个函数具有两个参数，第 1 个是指向属性对象的指针，第 2 个是作用域或指向作用域的指针，作用域控制线程是否在进程内或在系统级上竞争资源，可能的值是 PTHREAD_SCOPE_PROCESS（进程内竞争资源）PTHREAD_SCOPE_SYSTEM.（系统级上竞争资源）。

七、线程堆栈的大小

函数 pthread_attr_setstacksize 和 pthread_attr_getstacksize 分别用来设置和得到线程堆栈的大小，这两个函数的定义如下所示：

8.

名称::	pthread_attr_getdetstacksize pthread_attr_setstacksize
功能:	获得/修改线程栈的大小
头文件:	#include <pthread.h>
函数原形:	int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,size_t *restrict stacksize); int pthread_attr_setstacksize(pthread_attr_t *attr ,size_t *stacksize);
参数:	attr 线程属性变量 stacksize 堆栈大小
返回值:	若成功返回 0，若失败返回-1。

这两个参数具有两个参数，第 1 个是指向属性对象的指针，第 2 个是堆栈大小或指向堆栈大小的指针

如果希望改变栈的默认大小，但又不想自己处理线程栈的分配问题，这时使用 pthread_attr_setstacksize 函数就非常用用。

八、线程堆栈的地址

函数 pthread_attr_setstackaddr 和 pthread_attr_getstackaddr 分别用来设置和得到线程堆栈的位置，这两个函数的定义如下：

名称:：	pthread_attr_setstackaddr pthread_attr_getstackaddr
功能：	获得/修改线程栈的位置
头文件：	#include <pthread.h>
函数原形：	int pthread_attr_getstackaddr(const pthread_attr_t *attr,void **stackaddf); int pthread_attr_setstackaddr(pthread_attr_t *attr,void *stackaddr);
参数：	attr 线程属性变量 stackaddr 堆栈地址
返回值：	若成功返回 0，若失败返回-1。

9.

这两个函数具有两个参数，第 1 个是指向属性对象的指针，第 2 个是堆栈地址或指向堆栈地址的指针。

九、线程栈末尾的警戒缓冲区大小

函数 pthread_attr_getguardsize 和 pthread_attr_setguardsize 分别用来设置和得到线程栈末尾的警戒缓冲区大小，这两个函数的定义如下：

名称::	pthread_attr_getguardsize pthread_attr_setguardsize
功能:	获得/修改线程栈末尾的警戒缓冲区大小
头文件:	#include <pthread.h>
函数原形:	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);
参数:	
返回值:	若成功返回 0，若失败返回-1。

10.

线程属性 guardsize 控制着线程栈末尾之后以避免栈溢出的扩展内存大小。这个属性默认设置为 PAGESIZE 个字节。可以把 guardsize 线程属性设为 0，从而不允许属性的这种特征行为发生：在这种情况下不会提供警戒缓存区。同样地，如果对线程属性 stackaddr 作了修改，系统就会假设我们会自己管理栈，并使警戒栈缓冲区机制无效，等同于把 guardsize 线程属性设为 0。

第三章 Posix 有名信号灯

函数 sem_open 创建一个新的有名信号灯或打开一个已存在的有名信号灯。有名信号灯总是既可用于线程间的同步，又可以用于进程间的同步。

一、 posix 有名信号灯函数

1.

名称::	sem_open
功能:	创建并初始化有名信号灯
头文件:	#include <semaphore.h>
函数原形:	sem_t *sem_open(const char *name,int oflag,/*mode_t mode,unsigned int value*/);
参数:	name 信号灯的外部名字 oflag 选择创建或打开一个现有的信号灯 mode 权限位 value 信号灯初始值
返回值:	成功时返回指向信号灯的指针，出错时为 SEM_FAILED

oflag 参数可以是 0、O_CREAT（创建一个信号灯）或 O_CREAT|O_EXCL（如果没有指定的信号灯就创建），如果指定了 O_CREAT，那么第三个和第四个参数是需要的；其中 mode 参数指定权限位，value 参数指定信号灯的初始值，通常用来指定共享资源的书面。该初始不能超过 SEM_VALUE_MAX，这个常值必须低于为 32767。二值信号灯的初始值通常为 1，计数信号灯的初始值则往往大于 1。

如果指定了 O_CREAT（而没有指定 O_EXCL），那么只有所需的信号灯尚未存在时才初始化它。所需信号灯已存在条件下指定 O_CREAT 不是一个错误该标志的意思仅仅是“如果所需信号灯尚未存在，那就创建并初始化它”。但是所需信号灯等已存在条件下指定 O_CREAT|O_EXCL 却是一个错误。

sem_open 返回指向 sem_t 信号灯的指针，该结构里记录着当前共享资源的数目。

```
/*semopen.c*/
#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc,char **argv)
{
    sem_t *sem;

    if(argc!=2)
    {
        printf(“please input a file name!\n”);
        exit(1);
    }
    sem=sem_open(argv[1],O_CREAT,0644,1);
    exit(0);
}
```

#gcc –lpthread –o semopen semopen.c
#./semopen

名称::	sem_close
功能:	关闭有名信号灯
头文件:	#include <semaphore.h>
函数原形:	int sem_close(sem_t *sem);
参数:	sem 指向信号灯的指针
返回值:	若成功则返回 0，否则返回-1。

2.

一个进程终止时，内核还对其上仍然打开着的所有有名信号灯自动执行这样的信号灯关闭操作。不论该进程是自愿终止的还是非自愿终止的，这种自动关闭都会发生。

但应注意的是关闭一个信号灯并没有将它从系统中删除。这就是说，Posix 有名信号灯至少是随内核持续的：即使当前没有进程打开着某个信号灯，它的值仍然保持。

名称::	sem_unlink
功能:	从系统中删除信号灯
头文件:	#include <semaphore.h>
函数原形:	int sem_unlink(count char *name);
参数:	Name 信号灯的外部名字
返回值:	若成功则返回 0，否则返回-1。

3.

有名信号灯使用 sem_unlink 从系统中删除。

每个信号灯有一个引用计数器记录当前的打开次数，sem_unlink 必须等待这个数为 0 时才能把 name 所指的信号灯从文件系统中删除。也就是要等待最后一个 sem_close 发生。

```
/*semunlink.c*/
#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc,char **argv)
```

```
{
    sem_t *sem;
    int val;
    if(argc!=2)
    {
        printf("please input a file name!\n");
        exit(1);
    }
    if((sem_unlink(argv[1]))!=0)
        perror("sem_unlink");
    else
        printf("success");
    exit(0);
}
```

4.

名称::	sem_getvalue
功能:	测试信号灯
头文件:	#include <semaphore.h>
函数原形:	int sem_getvalue(sem_t *sem,int *valp);
参数:	sem 指向信号灯的指针
返回值:	若成功则返回 0，否则返回-1。

sem_getvalue 在由 valp 指向的正数中返回所指定信号灯的当前值。如果该信号灯当前已上锁，那么返回值或为 0，或为某个负数，其绝对值就是等待该信号灯解锁的线程数。

```
/*semgetvalue.c*/
#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc,char **argv)
{
    sem_t *sem;
    int val;

    if(argc!=2)
    {
        printf("please input a file name!\n");
```

```
        exit(1);
    }
    sem=sem_open(argv[1],0);
    sem_getvalue(sem,&val);
    printf("getvalue:value=%d\n",val);
    exit(0);
}
```

5.

名称::	sem_wait/sem_trywait
功能:	等待共享资源
头文件:	#include <semaphore.h>
函数原形:	int sem_wait(sem_t *sem); int sem_trywait(sem_t *sem);
参数:	sem 指向信号灯的指针
返回值:	若成功则返回 0，否则返回-1。

我们可以用 sem_wait 来申请共享资源，sem_wait 函数可以测试所指定信号灯的值，如果该值大于 0，那就将它减 1 并立即返回。我们就可以使用申请来的共享资源了。如果该值等于 0，调用线程就被进入睡眠状态，直到该值变为大于 0，这时再将它减 1，函数随后返回。sem_wait 操作必须是原子的。sem_wait 和 sem_trywait 的差别是：当所指定信号灯的值已经是 0 时，后者并不将调用线程投入睡眠。相反，它返回一个 EAGAIN 错误。

下面的程序我们先不去运行，稍后再运行。

```
/*semwait.c*/
#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc,char **argv)
{
    sem_t *sem;
    int val;

    if(argc!=2)
    {
        printf("please input a file name!\n");
        exit(1);
    }
```

```
    sem=sem_open(argv[1],0);
    sem_wait(sem);
    sem_getvalue(sem,&val);
    printf("pid %ld has semaphore,value=%d\n",(long)getpid(),val);
    pause();
    exit(0);
}
```

6.

名称::	sem_post
功能:	挂出共享资源
头文件:	#include <semaphore.h>
函数原形:	int sem_post(sem_t *sem); int sem_getvalue(sem_t *sem,int *valp);
参数:	sem 指向信号灯的指针
返回值:	若成功则返回 0，否则返回-1。

当一个线程使用完某个信号灯时，它应该调用 sem_post 来告诉系统申请的资源已经用完。本函数和 sem_wait 函数的功能正好相反，它把所指定的信号灯的值加 1，然后唤醒正在等待该信号灯值变为正数的任意线程。

下面的程序我们先不去运行，稍后再运行。

```
/*sempost.c*/
#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc,char **argv)
{
    sem_t *sem;
    int val;

    if(argc!=2)
    {
        printf("please input a file name!\n");
        exit(1);
    }
    sem=sem_open(argv[1],0);
    sem_post(sem);
```

```
    sem_getvalue(sem,&val);
    printf("value=%d\n", val);
    exit(0);
}
```

二、关于 **posix** 有名信号灯使用的几点注意

我们要注意以下几点：

1.Posix 有名信号灯的值是随内核持续的。也就是说，一个进程创建了一个信号灯，这个进程结束后，这个信号灯还存在，并且信号灯的值也不会改变。

下面我们利用上面的几个程序来证明这点

#./semopen test

#./semgetvalue test

value=1 信号的值仍然是 1

2。当持有某个信号灯锁的进程没有释放它就终止时，内核并不给该信号灯解锁。

#./semopen test

#./semwait test&

pid 1834 has semaphore,value=0

#./semgetvalue test

value=0 信号量的值变为 0 了

三、**posix** 有名信号灯应用于多线程

```
#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <thread.h>

void *thread_function(void *arg); /*线程入口函数*/
void print(pid_t); /*共享资源函数*/
sem_t *sem; /*定义 Posix 有名信号灯*/
int val; /*定义信号灯当前值*/

int main(int argc,char *argv[])
{
    int n=0;

    if(argc!=2)
    {
        printf("please input a file name!\n");
        exit(1);
    }
    sem=sem_open(argv[1],O_CREAT,0644,3); /*打开一个信号灯*/
```

```
    while(n++<5) /*循环创建 5 个子线程，使它们同步运行*/
    {
        if（（pthread_create(&a_thread,NULL,thread_function,NULL)）!=0）
       {
            perror("Thread creation failed");
            exit(1);
        }
    }
    pthread_join(a_thread,NULL);
    sem_close(bin_sem);
    sem_unlink(argv[1]);
}

void *thread_function(void *arg)
{
          sem_wait(sem); /*申请信号灯*/
        print(); /*调用共享代码段*/
        sleep(1);
        sem_post(sem); /*释放信号灯*/
        printf("I'm finished,my tid is %d\n",pthread_self());
}

void print()
{
    printf("I get it,my tid is %d\n",pthread_self());
    sem_getvalue(sem,&val);
    printf("Now the value have %d\n",val);
}
```

程序用循环的方法建立 5 个线程，然后让它们调用同一个线程处理函数 thread_function，在函数里我们利用信号量来限制访问共享资源的线程数。共享资源我们用 print 函数来代表，在真正编程中它有可以是一个终端设备（如打印机）或是一段有实际意义的代码。

运行结果为：

```
#gcc –lpthread –o 8_1 8_1.c
#./8_1 test
I get it,my tid is 1082330304
Now the value have 2
Iget it,my pid is 1894
Now the value have 1
Iget it,my pid is 1895
```

Now the value have 0
I’m finished,my pid is 1893
I’m finished,my pid is 1894
I’m finished,my pid is 1895
I get it,my pid is 1896
Now the value have 2
I get it,mypid is 1897
Now the value have 1
I’m finished,my pid is 1896
I’m finished,my pid is 1897

四、**posix** 有名信号灯应用于多进程

下面就是应用 Posix 有名信号灯的一个小程序。用它来限制访问共享代码的进程数目。

```
#include <semaphore.h>

#include <unistd.h>

#include <stdio.h>

#include <fcntl.h>

void print(pid_t);

sem_t *sem; /*定义 Posix 有名信号灯*/

int val; /*定义信号灯当前值*/

int main(int argc,char *argv[])

{

    int n=0;

    if(argc!=2)

    {

        printf(“please input a file name!\n”);

        exit(1);
```

```
    }
    sem=sem_open(argv[1],O_CREAT,0644,2); /*打开一个信号灯，初值设为 2*/
    while(n++<5) /*循环创建 5 个子进程，使它们同步运行*/
    {
       if(fork()==0)
       {
          sem_wait(sem); /*申请信号灯*/
          print(getpid()); /*调用共享代码段*/
          sleep(1);
          sem_post(sem); /*释放信号灯*/
          printf("I'm finished,my pid is %d\n",getpid());
          return 0;
       }
    wait(); /*等待所有子进程结束*/
    sem_close(sem);
    sem_unlink(argv[1]);
    exit(0);
}
void print(pid_t pid)
{
    printf("I get it,my pid is %d\n",pid);
    sem_getvalue(sem,&val);
```

```
    printf(“Now the value have %d\n”,val);
}
```

程序编译后运行会得到如下结果：

```
#./8_2 8_2.c
I get it,my tid is 1082330304
Now the value have 1
I get it,my tid is 1090718784
Now the value have 0
I finished,my pid is 1082330304
I finished,my pid is 1090718784
I get it,my tid is 1099107264
Now the value have 1
I get it,my tid is 1116841120
Now the value have 0
I finished,my pid is 1099107264
I finished,my pid is 1116841120
I get it,my tid is 1125329600
Now the value have 1
I finished,my pid is 1125329600
```

五、基于内存的信号灯

前面已经介绍了 Posix 有名信号灯。这些信号灯由一个 name 参数标识，它通常指代文件系统中的某个文件。然而 Posix 也提供基于内存的信号灯，它们由应用程序分配信号灯的内存空间，然后由系统初始化它们的值。

7.

名称::	sem_init/sem_destroy
功能:	初始化/关闭信号等
头文件:	#include <semaphore.h>

函数原形:	int sem_init(sem_t *sem,int shared,unsigned int value);
	
	int sem_getvalue(sem_t *sem);

参数:	sem　指向信号灯的指针 shared　作用范围 value　信号灯初始值
返回值:	若成功则返回 0，否则返回-1。

基于内存的信号灯是由 sem_init 初始化的。sem 参数指向必须由应用程序分配的 sem_t 变量。如果 shared 为 0，那么待初始化的信号灯是在同一进程的各个线程共享的，否则该信号灯是在进程间共享的。当 shared 为零时，该信号灯必须存放在即将使用它的所有进程都能访问的某种类型的共享内存中。跟 sem_open 一样，value 参数是该信号灯的初始值。

使用完一个基于内存的信号灯后，我们调用 sem_destroy 关闭它。

除了 sem_open 和 sem_close 外，其它的 poisx 有名信号灯函数都可以用于基于内存的信号灯。

注意：posix 基于内存的信号灯和 posix 有名信号灯有一些区别，我们必须注意到这些。

1.sem_open 不需要类型与 shared 的参数，有名信号灯总是可以在不同进程间共享的。

2.sem_init 不使用任何类似于 O_CREAT 标志的东西，也就是说，sem_init 总是初始化信号灯的值。因此，对于一个给定的信号灯，我们必须小心保证只调用一次 sem_init。

3.sem_open 返回一个指向某个 sem_t 变量的指针，该变量由函数本身分配并初始化。但 sem_init 的第一个参数是一个指向某个 sem_t 变量的指针，该变量由调用者分配，然后由 sem_init 函数初始化。

4.posix 有名信号灯是通过内核持续的，一个进程创建一个信号灯，另外的进程可以通过该信号灯的外部名（创建信号灯使用的文件名）来访问它。posix 基于内存的信号灯的持续性却是不定的，如果基于内存的信号灯是由单个进程内的各个线程共享的，那么该信号灯就是随进程持续的，当该进程终止时它也会消失。如果某个基于内存的信号灯是在不同进程间同步的，该信号灯必须存放在共享内存区中，这要只要该共享内存区存在，该信号灯就存在。

5.基于内存的信号灯应用于线程很麻烦（待会你会知道为什么），而有名信号灯却很方便，基于内存的信号灯比较适合应用于一个进程的多个线程。

下面是 posix 基于内存的信号灯实现一个进程的各个线程间的互次。

```
#include <semaphore.h>

#include <unistd.h>

#include <stdio.h>

#include <fcntl.h>

#include <pthread.h>

#incude <stdlib.h>

void *thread_function(void *arg); /*线程入口函数*/

void print(void); /*共享资源函数*/

sem_t bin_sem; /*定义信号灯*/

int value; /*定义信号量的灯*/

int main()

{

    int n=0;

    pthread_t a_thread;

    if((sem_init(&bin_sem,0,2))!=0) /*初始化信号灯，初始值为 2*/

    {

       perror(“sem_init”);

       exit(1);

    }

    while(n++<5) /*循环创建 5 个线程*/
```

```
    {
       if((pthread_create(&a_thread,NULL,thread_function,NULL))!=0)
         {
            perror("Thread creation failed");
            exit(1);
         }
    }
    pthread_join(a_thread,NULL);/*等待子线程返回*/
}
void *thread_function(void *arg)
{
    sem_wait(&bin_sem); /*等待信号灯*/
    print();
    sleep(1);
    sem_post(&bin_sem); /*挂出信号灯*/
    printf("I finished,my pid is %d\n",pthread_self());
    pthread_exit(arg);
}
void print()
{
    printf("I get it,my tid is %d\n",pthread_self());
    sem_getvalue(&bin_sem,&value); /*获取信号灯的值*/
```

```
    printf("Now the value have %d\n",value);
}
```

posix 基于内存的信号灯和有名信号灯基本是一样的，上面的几点区别就可以了。

下面是运行结果：

#gcc –lpthread –o seminitthread seminitthread.c

#./seminitthread

I get it,my tid is 1082330304

Now the value have 1

I get it,my tid is 1090718784

Now the value have 0

I finished,my pid is 1082330304

I finished,my pid is 1090718784

I get it,my tid is 1099107264

Now the value have 1

I get it,my tid is 1116841120

Now the value have 0

I finished,my pid is 1099107264

I finished,my pid is 1116841120

I get it,my tid is 1125329600

Now the value have 1

I finished,my pid is 1125329600

下面是经典的生产者消费者问题：

```
#include <stdio.h>
```

```
#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#include <errno.h>

#include <sys/ipc.h>

#include <semaphore.h>

#include <fcntl.h>

#define FIFO "myfifo"

#define N 5

int lock_var;

time_t end_time;

char buf_r[100];

sem_t mutex,full,avail;/*定义 3 个信号量，full 标识缓冲区是否为满，avail 标识缓冲区是否为空．*/

int fd;

void pthread1(void *arg);

void pthread2(void *arg);

void consumer(void *arg);

void productor(void *arg);

int main(int argc, char *argv[])

{

    pthread_t id1,id2;
```

```
pthread_t mon_th_id;
int ret;
end_time = time(NULL)+30;
if((mkfifo(FIFO,O_CREAT|O_EXCL)<0)&&(errno!=EEXIST))
    printf("cannot create fifoserver\n");
printf("Preparing for reading bytes...\n");
memset(buf_r,0,sizeof(buf_r));
fd=open(FIFO,O_RDWR|O_NONBLOCK,0);
if(fd==-1)
{
    perror("open");
    exit(1);
}
ret=sem_init(&mutex,0,1);
ret=sem_init(&avail,0,N);
ret=sem_init(&full,0,0);
if(ret!=0)
{
    perror("sem_init");
}
ret=pthread_create(&id1,NULL,(void *)productor, NULL);
if(ret!=0)
```

```
        perror("pthread cread1");
    ret=pthread_create(&id2,NULL,(void *)consumer, NULL);
    if(ret!=0)
        perror("pthread cread2");
    pthread_join(id1,NULL);
    pthread_join(id2,NULL);
    exit(0);
}
void productor(void *arg)
{
    int i,nwrite;
    while(time(NULL) < end_time){
        sem_wait(&avail);
        sem_wait(&mutex);
        if((nwrite=write(fd,"hello",5))==-1)
     {
            if(errno==EAGAIN)
                printf("The FIFO has not been read yet.Please try later\n");
     }
        else
            printf("write hello to the FIFO\n");
        sem_post(&full);
```

```
        sem_post(&mutex);
        sleep(1);
    }
}
void consumer(void *arg)
{
    int nolock=0;
    int ret,nread;
    while(time(NULL) < end_time){
        sem_wait(&full);
        sem_wait(&mutex);
        memset(buf_r,0,sizeof(buf_r));
        if((nread=read(fd,buf_r,100))==-1){
            if(errno==EAGAIN)
                printf("no data yet\n");
     }
        printf("read %s from FIFO\n",buf_r);
        sem_post(&avail);
        sem_post(&mutex);
        sleep(1);
    }
}
```

下面是 posix 基于内存的信号灯实现各进程间的互斥。但要注意它并不能得到我们想要的结果。

```
#include <semaphore.h>

#include <unistd.h>

#include <stdio.h>

#include <fcntl.h>

void print(pid_t);

sem_t *sem; /*定义 Posix 有名信号灯*/

int val; /*定义信号灯当前值*/

int main(int argc,char *argv[])

{

    int n=0;

    sem=sem_open(argv[1],O_CREAT,0644,3); /*打开一个信号灯*/

    sem_getvalue(sem,&val); /*查看信号灯的值*/

    printf("The value have %d\n",val);

    while(n++<5) /*循环创建 5 个子进程，使它们同步运行*/

    {

       if(fork()==0)

       {

          sem_wait(sem); /*申请信号灯*/

          print(getpid()); /*调用共享代码段*/

          sleep(1);

          sem_post(sem); /*释放信号灯*/
```

```
            printf("I'm finished,my pid is %d\n",getpid());
            return 0;
        }
    wait(); /*等待所有子进程结束*/
    return 0;
}
void print(pid_t pid)
{
    printf("I get it,my pid is %d\n",pid);
    sem_getvalue(sem,&val);
    printf("Now the value have %d\n",val);
}
```

下面是运行结果：

```
#cc –lpthread –o sem sem.c
#./sem
The value have 3
I get it,my pid is 2236
Now the value have 2
I get it,my pid is 2237
Now the value have 2
I get it,my pid is 2238
Now the value have 2
I get it,my pid is 2239
```

Now the value have 2

Iget it,my pid is 2240

Now the value have 2

I’m finished,my pid is 2236

I’m finished,my pid is 2237

I’m finished,my pid is 2238

I’m finished,my pid is 2239

I’m finished,my pid is 2240

问题在于 sem 信号灯不在共享内存区中。fork 出来的子进程通常不共享父进程的内存空间。子进程是在父进程内存空间的拷贝上启动的，它跟共享内存不是一回事。

第四章 互斥量

一、什么是互斥锁

另一种在多线程程序中同步访问手段是使用互斥量。程序员给某个对象加上一把“锁”，每次只允许一个线程去访问它。如果想对代码关键部分的访问进行控制，你必须在进入这段代码之前锁定一把互斥量，在完成操作之后再打开它。

互斥量函数有
pthread_mutex_init 初始化一个互斥量
pthread_mutex_lock 给一个互斥量加锁
pthread_mutex_trylock 加锁，如果失败不阻塞
pthread_mutex_unlock 解锁

可以通过使用 pthread 的互斥接口保护数据，确保同一时间只有一个线程访问数据。互斥量从本质上说是一把锁，在访问共享资源前对互斥量进行加锁，在访问完成后释放互斥量上的锁。对互斥量进行加锁以后，任何其他试图再次对互斥量加锁的线程将会被阻塞直到当前线程释放该互斥锁。如果释放互斥锁时有多个线程阻塞，所以在该互斥锁上的阻塞线程都会变成可进行状态，第一个变成运行状态的线程可以对互斥量加锁，其他线程在次被阻塞，等待下次运行状态。

互斥量用 pthread_mutex_t 数据类型来表示，在使用互斥量以前，必须首先对它进行初始化，可以把它置为常量 PTHREAD_MUTEX_INITIALIZER(只对静态分配的互斥量)，也可以通过调用 pthread_mutex_init 函数进行初始化，如果动态地分配互斥量，那么释放内存前需要调用 pthread_mutex_destroy.

二、初始化/回收互斥锁

名称::	pthread_mutexattr_init
功能:	初始化互斥锁。
头文件:	#include <pthread.h>
函数原形:	int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutex_t *attr);
参数:	mutex 互斥量 attr 互斥锁属性
返回值:	若成功则返回 0，否则返回错误编号。

1.

mutex 是我们要锁住的互斥量，attr 是互斥锁的属性，可用相应的函数修改，我们在下章介绍，要用默认的属性初始化互斥量，只需把 attr 设置为 NULL。

名称::	pthread_mutex_destroy
功能:	释放对互斥变量分配的资源
头文件:	#include <pthread.h>
函数原形:	int pthread_mutex_destroy(pthread_mutex_t *mutex);
参数:	
返回值:	若成功则返回 0，否则返回错误编号。

2.

三、对互斥量加减锁

名称::	pthread_mutex_lock/ pthread_mutex_trylock/ pthread_mutex_unlock
功能:	对互斥量加/减锁
头文件:	#include <pthread.h>

函数原形:	int pthread_mutex_lock(pthread_mutex_t *mutex);
	int pthread_mutex_trylock(pthread_mutex_t *mutex);
	int pthread_mutex_unlock(pthread_mutex_t *mutex);

参数:	
返回值:	若成功则返回 0，否则返回错误编号。

3.

对互斥量进行加锁，需要调用 pthread_mutex_lock,如果互斥量已经上锁，调用线程阻塞直至互斥量解锁。对互斥量解锁，需要调用 pthread_mutex_unlock.

如果线程不希望被阻塞，他可以使用 pthread_mutex_trylock 尝试对互斥量进行加锁。如果调用 pthread_mutex_trylock 时互斥量处于未锁住状态，那么 pthread_mutex_trylock 将锁住互斥量，否则就会失败，不能锁住互斥量，而返回 EBUSY。

下面试例子可以证明对互斥量加锁的必要性：
我们先来看不加锁的程序。

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

viid *thread_function(void *arg);

int run_now=1;   /*用 run_now 代表共享资源*/

int main()
{
    int print_count1=0; /*用于控制循环*/
    prhread_t a_thread;

    if(pthread_create(&a_thread,NULL,thread_function,NULL)!=0)   /*创建一个进
程*/
    {
        perror(“Thread createion failed”);
        exit(1);
    }

    while(print_count1++<5)
    {
        if(run_now==1)  /主线程：如果 run_now 为 1 就把它修改为 2*/
        {
            printf(“main thread is run\n”);
            run_now=2;
        }
        else
        {
            printf(“main thread is sleep\n”);
            sleep(1);
        }
    }
    pthread_join(a_thread,NULL); /*等待子线程结束*/
    exit(0);
}

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

```
    while(print_count2++<5)
    {
        if(run_now==2)  /子线程：如果 run_now 为 1 就把它修改为 1*/
        {
            printf("function thread is run\n");
            run_now=1;
        }
        else
        {
            printf("function thread is sleep\n");
            sleep(1);
        }
    }
    pthread_exit(NULL);
}
```

运行上面程序的运行结果为:

```
function thread is sleep
main thread is run
main thread is sleep
main thread is sleep
function thread is run
function thread is sleep
main thread is run
main thread is sleep
function thread is run
function thread is sleep
```

我们可以看到 main 线程和 function 线程是交替运行的。它们都可以对 run_now 进行操作。

下面是加锁的程序。

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

viid *thread_function(void *arg);

int run_now=1;  /*用 run_now 代表共享资源*/
pthread_mutex_t work_mutex;  /*定义互斥量*/

int main()
{
```

```
    int res;
    int print_count1=0;
    prhread_t a_thread;

    if(pthread_mutex_init(&work_mutex,NULL)!=0)  /*初始化互斥量*/
    {
        perror("Mutex init faied");
        exit(1);
    }

    if(pthread_create(&a_thread,NULL,thread_function,NULL)!=0)  /*创建新线程*
/
    {
        perror("Thread createion failed");
        exit(1);
    }

    if(pthread_mutex_lock(&work_mutex)!=0)  /*对互斥量加锁*/
    {
        preeor("Lock failed");
        exit(1);
    }
    else
        printf("main lock\n");

    while(print_count1++<5)
    {
        if(run_now==1)  /主线程：如果 run_now 为 1 就把它修改为 2*/
        {
            printf("main thread is run\n");
            run_now=2;
        }
        else
        {
            printf("main thread is sleep\n");
            sleep(1);
        }
    }

    if(pthread_mutex_unlock(&work_mutex)!=0) /*对互斥量解锁*/
    {
        preeor("unlock failed");
        exit(1);
```

```
    }
    else
        printf("main unlock\n");

    pthread_mutex_destroy(&work_mutex); /*收回互斥量资源*/
    pthread_join(a_thread,NULL); /*等待子线程结束*/
    exit(0);
}

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

    sleep(1);
    if(pthread_mutex_lock(&work_mutex)!=0)
    {
        perror("Lock failed");
        exit(1);
    }
    else
        printf("function lock\n");
    while(print_count2++<5)
    {
        if(run_now==2)  /分进程：如果 run_now 为 1 就把它修改为 1*/
        {
            printf("function thread is run\n");
            run_now=1;
        }
        else
        {
            printf("function thread is sleep\n");
            sleep(1);
        }
    }

    if(pthread_mutex_unlock(&work_mutex)!=0)  /*对互斥量解锁*/
    {
        perror("unlock failed");
        exit(1);
    }
    else
        printf("function unlock\n");
    pthread_exit(NULL);
```

```
}
```

下面是运行结果：

```
main lock
main thread is run
main thread is sleep
main thread is sleep
main thread is sleep
main thread is sleep
main unlock
function lock
function thread is run
function thread is sleep
function thread is sleep
function thread is sleep
function thread is sleep
function unlock
```

我们从运行结果可以看到，当主线程把互斥量锁住后，子线程就不能对共享资源进行操作了。

四、互斥锁属性

线程和线程的同步对象（互斥量，读写锁，条件变量）都具有属性。在修改属性前都需要对该结构进行初始化。使用后要把该结构回收。我们用 pthread_mutexattr_init 函数对 pthread_mutexattr 结构进行初始化，用 pthread_mutexattr_destroy 函数对该结构进行回收。

4

名称:：	pthread_mutexattr_init/pthread_mutexattr_destroy
功能：	初始化/回收 pthread_mutexattr_t 结构
头文件：	#include <pthread.h>
函数原形：	int pthread_mutexattr_init(pthread_mutexattr_t *attr); int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
参数：	Attr　pthread_mutexattr_t 结构变量
返回值：	若成功返回 0，若失败返回错误编号。

pthread_mutexattr_init 将属性对象的值初始化为缺省值。并分配属性对象占用的内存空间。

attr 中 pshared 属性表示用这个属性对象创建的互斥锁的作用域，它的取值可以是 PTHREAD_PROCESS_PRIVATE(缺省值，表示由这个属性对象创建的互斥锁只能在进程内使用)或 PTHREAD_PROCESS_SHARED。

互斥量属性分为共享互斥量属性和类型互斥量属性。两种属性分别由不同的函数得到并由不同的函数进行修改。pthread_mutexattr_getpshared 和 pthread_mutexattr_setpshared 函数可以获得和修改共享互斥量属性 。pthread_mutexattr_gettype 和 pthread_mutexattr_settype 函数可以获得和修改类型互斥量属性。下面我们分别介绍。

名称::	pthread_mutexattr_getpshared/pthread_mutexattr_setpshared
功能:	获得/修改共享互斥量属性
头文件:	#include <pthread.h>
函数原形:	int pthread_mutexattrattr_ getpshared (const pthread_attr_t *restrict attr,int*restrict pshared); int pthread_mutexattrattr_ setpshared (const pthread_attr_t *restrict attr,int pshared);
参数:	
返回值:	若成功返回 0，若失败返回错误编号。

5

共享互斥量属性用于规定互斥锁的作用域。互斥锁的域可以是进程内的也可以是进程间的。pthread_mutexattrattr_ getpshared 可以返回属性对象的互斥锁作用域属性。可以是以下值:
PTHREAD_PROCESS_SHARED，PTHREAD_PROCESS_PRIVATE。如果互斥锁属性对象的 pshared 属性被置 PTHREAD_PROCESS_SHARED。那么由这个属性对象创建的互斥锁将被保存在共享内存中，可以被多个进程中的线程共享。如果 pshared 属性被置为 PTHREAD_PROCESS_PRIVATE，那么只有和创建这个互斥锁的线程在同一个进程中的线程才能访问这个互斥锁。

名称::	pthread_mutexattr_gettype/pthread_mutexattr_settype
功能:	获得/修改类型互斥量属性
头文件:	#include <pthread.h>
函数原形:	int pthread_mutexattrattr_ getpshared (const pthread_attr_t *restrict attr,int*restrict pshared); int pthread_mutexattrattr_ setpshared (const pthread_attr_t *restrict attr,int pshared);
参数:	
返回值:	若成功返回 0，若失败返回错误编号。

6

pthread_mutexattr_gettype 函数可以获得互斥锁类型属性。缺省的互斥锁类型属性是 PTHREAD_MUTEX_DEFAULT。
合法的类型属性值有：
PTHREAD_MUTEX_NORMAL；
PTHREAD_MUTEX_ERRORCHECK；
PTHREAD_MUTEX_RECURSIVE；
PTHREAD_MUTEX_DEFAULT。
类型说明：
PTHREAD_MUTEX_NORMAL
这种类型的互斥锁不会自动检测死锁。如果一个线程试图对一个互斥锁重复锁定将会引起这个线程的死锁。如果试图解锁一个由别的线程锁定的互斥锁会引发不可预料的结果。如果一个线程试图解锁已经被解锁的互斥锁也会引发不可预料的结果。

PTHREAD_MUTEX_ERRORCHECK
这种类型的互斥锁会自动检测死锁。如果一个线程试图对一个互斥锁重复锁定，将会返回一个错误代码。如果试图解锁一个由别的线程锁定的互斥锁将会返回一个错误代码。如果一个线程试图解锁已经被解锁的互斥锁也将会返回一个错误代码。

PTHREAD_MUTEX_RECURSIVE
如果一个线程对这种类型的互斥锁重复上锁，不会引起死锁，一个线程对这类互斥锁的多次重复上锁必须由这个线程来重复相同数量的解锁，这样才能解开这个互斥锁，别的线程才能得到这个互斥锁。如果试图解锁一个由别的线程锁定的互斥锁将会返回一个错误代码。如果一个线程试图解锁已经被解锁的互斥锁也将会返回一个错误代码。这种类型的互斥锁只能是进程私有的（作用域属性为 PTHREAD_PROCESS_PRIVATE）。

PTHREAD_MUTEX_DEFAULT
这种类型的互斥锁不会自动检测死锁。如果一个线程试图对一个互斥锁重复锁定将会引起不可预料的结果。如果试图解锁一个由别的线程锁定的互斥锁会引发不可预料的结果。如果一个线程试图解锁已经被解锁的互斥锁也会引发不可预料的结果。POSIX 标准规定，对于某一具体的实现，可以把这种类型的互斥锁定义为其他类型的互斥锁。

五、 应用互斥量需要注意的几点

1、互斥量需要时间来加锁和解锁。锁住较少互斥量的程序通常运行得更快。所以，互斥量应该尽量少，够用即可，每个互斥量保护的区域应则尽量大。

2、互斥量的本质是串行执行。如果很多线程需要领繁地加锁同一个互斥量，则线程的大部分时间就会在等待，这对性能是有害的。如果互斥量保护的数据(或代码)包含彼此无关的片段，则可以特大的互斥量分解为几个小的互斥量来提高性能。这样，任意时刻需要小互斥量的线程减少，线程等待时间就会减少。所以，互斥量应该足够多(到有意义的地步)，每个互斥量保护的区域则应尽量的少。

第五章 条件变量

一、什么是条件变量

与互斥锁不同，条件变量是用来等待而不是用来上锁的。条件变量用来自动阻塞一个线程，直到某特殊情况发生为止。通常条件变量和互斥锁同时使用。

条件变量使我们可以睡眠等待某种条件出现。条件变量是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待"条件变量的条件成立"而挂起；另一个线程使"条件成立"（给出条件成立信号）。

条件的检测是在互斥锁的保护下进行的。如果一个条件为假，一个线程自动阻塞，并释放等待状态改变的互斥锁。如果另一个线程改变了条件，它发信号给关联的条件变量，唤醒一个或多个等待它的线程，重新获得互斥锁，重新评价条件。如果两进程共享可读写的内存，条件变量可以被用来实现这两进程间的线程同步。

使用条件变量之前要先进行初始化。可以在单个语句中生成和初始化一个条件变量如：

pthread_cond_t my_condition=PTHREAD_COND_INITIALIZER;（用于进程间线程的通信）。

也可以利用函数 pthread_cond_init 动态初始化。

二、条件变量函数

名称：	pthread_cond_init
目标：	条件变量初始化
头文件：	#include < pthread.h>
函数原形：	int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
参数:	cptr 条件变量 attr 条件变量属性
返回值：	成功返回 0，出错返回错误编号。

1.

pthread_cond_init 函数可以用来初始化一个条件变量。他使用变量 attr 所指定的属性来初始化一个条件变量，如果参数 attr 为空，那么它将使用缺省的属性来设置所指定的条件变量。

2.

名称：	pthread_cond_destroy
目标：	条件变量摧毁
头文件：	#include < pthread.h>
函数原形：	int pthread_cond_destroy(pthread_cond_t *cond);
参数:	cptr 条件变量
返回值：	成功返回 0，出错返回错误编号。

pthread_cond_destroy 函数可以用来摧毁所指定的条件变量，同时将会释放所给它分配的资源。调用该函数的进程也并不要求等待在参数所指定的条件变量上。

名称：	pthread_cond_wait/pthread_cond_timedwait
目标：	条件变量等待
头文件：	#include < pthread.h>
函数原形：	int pthread_cond_wait(pthread_cond_t *cond,pthread_mutex_t *mutex); int pthread_cond_timedwait(pthread_cond_t *cond,pthread_mutex_t mytex,const struct timespec *abstime);
参数:	cond 条件变量 mutex 互斥锁
返回值：	成功返回 0，出错返回错误编号。

3.

第一个参数*cond 是指向一个条件变量的指针。第二个参数*mutex 则是对相关的互斥锁的指针。函数 pthread_cond_timedwait 函数类型与函数 pthread_cond_wait,区别在于，如果达到或是超过所引用的参数*abstime,它将结束并返回错误 ETIME.pthread_cond_timedwait 函数的参数*abstime 指向一个 timespec 结构。该结构如下：

```
typedef struct timespec{
        time_t tv_sec;
        long tv_nsex;
}timespec_t;
```

3.

名称：	pthread_cond_signal/pthread_cond_broadcast
目标：	条件变量通知
头文件：	#include < pthread.h>
函数原形：	int pthread_cond_signal(pthread_cond_t *cond); int pthread_cond_broadcast(pthread_cond_t *cond);
参数:	cond 条件变量
返回值：	成功返回 0，出错返回错误编号。

参数*cond 是对类型为 pthread_cond_t 的一个条件变量的指针。当调用 pthread_cond_signal 时一个在相同条件变量上阻塞的线程将被解锁。如果同时有多个线程阻塞，则由调度策略确定接收通知的线程。如果调用 pthread_cond_broadcast,则将通知阻塞在这个条件变量上的所有线程。一旦被唤醒线程仍然会要求互斥锁。如果当前没有线程等待通知，则上面两种调用实际上成为一个空操作。如果参数*cond 指向非法地址，则返回值 EINVAL。

下面是一个简单的例子，我们可以从程序的运行来了解条件变量的作用。

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

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;/*初始化互斥锁*/
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;/*初始化条件变量*/

void *thread1(void *);
void *thread2(void *);

int i=1;
int main(void)
{
   pthread_t t_a;
   pthread_t t_b;

   pthread_create(&t_a,NULL,thread2,(void *)NULL);/*创建进程 t_a*/
   pthread_create(&t_b,NULL,thread1,(void *)NULL); /*创建进程 t_b*/
   pthread_join(t_b, NULL);/*等待进程 t_b 结束*/
   pthread_mutex_destroy(&mutex);
   pthread_cond_destroy(&cond);
   exit(0);
}

void *thread1(void *junk)
{
   for(i=1;i<=9;i++)
   {
      pthread_mutex_lock(&mutex);/*锁住互斥量*/
      if(i%3==0)
          pthread_cond_signal(&cond);/*条件改变，发送信号，通知 t_b 进程*/
```

```
        else
            printf("thead1:%d\n",i);
        pthread_mutex_unlock(&mutex);/*解锁互斥量*/
            sleep(1);
      }
}

void *thread2(void *junk)
{
   while(i<9)
   {
      pthread_mutex_lock(&mutex);
          if(i%3!=0)
          pthread_cond_wait(&cond,&mutex);/*等待*/
      printf("thread2:%d\n",i);
      pthread_mutex_unlock(&mutex);
          sleep(1);
      }
}
```

程序创建了 2 个新线程使他们同步运行，实现进程 t_b 打印 20 以内 3 的倍数，t_a 打印其他的数，程序开始线程 t_b 不满足条件等待，线程 t_a 运行使 a 循环加 1 并打印。直到 i 为 3 的倍数时，线程 t_a 发送信号通知进程 t_b，这时 t_b 满足条件，打印 i 值。

下面是运行结果：

```
#cc –lpthread –o cond cond.c
#./cond
thread1:1
thread1:2
thread2:3
thread1:4
thread1:5
thread2:6
thread1:7
thread1:8
thread2:9
```

下面的程序是经典的生产者/消费者的例证。

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX 5

pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER; /*初始化互斥锁*/
pthread_cond_t=PTHREAD_CODE_INITIALIZER; /*初始化条件变量*/
```

```
typedef struct{
      char buffer[MAX];
      int how_many;
}BUFFER;

BUFFER share={“”,0};
char ch=’A’;/*初始化 ch*/

void *read_some(void *);
void *write_some(void *);

int main(void)
{
      pthread_t t_read;
      pthread_t t_write;

      pthread_create(&t_read,NULL,read_some,(void *)NULL); /*创建进程 t_a*/
      pthread_create(&t_write,NULL,write_some,(void *)NULL); /*创建进程 t_b*/
      pthread_join(t_write,(void **)NULL);
      pthread_mutex_destroy(&mutex);
      pthread_cond_destroy(&cond);
      exit(0);
}

void *read_some(void *junk)
{
      int n=0;

      printf(“R %2d: starting\n”,pthread_self());

      while(ch!=’Z’)
      {
          pthread_mutex_lock(&lock_it);/*锁住互斥量*/
          if(share.how_many!=MAX)
          {

              share.buffer[share.how_many++]=ch++;/*把字母读入缓存*/
              printf(“R %2d:Got char[%c]\n”,pthread_self(),ch-1);/*打印读入字母*/
              if(share.how_many==MAX)
              {
                  printf(“R %2d:signaling full\n”,pthread_self());
                  pthread_cond_signal(&write_it);/*如果缓存中的字母到达了最大值
```

```
就发送信号*/
                }
            pthread_mutex_unlock(&lock_it);/*解锁互斥量*/
        }
        sleep(1);
        printf("R %2d:Exiting\n",pthread_self());
        return NULL;
}

void *write_some(void *junk)
{
        int i;
        int n=0;
        printf("w %2d: starting\n",pthread_self());

        while(ch!='Z')
        {
            pthread_mutex_lock(&lock_it);/*锁住互斥量*/
            printf("\nW %2d:Waiting\n",pthread_self());
            while(share.how_many!=MAX)/*如果缓存区字母不等于最大值就等待*/
                pthread_cond_wait(&write_it,&lock_it);
            printf("W %2d:writing buffer\n",pthread_self());
            for(i=0;share.buffer[i]&&share.how_many;++i,share.how_many--)
                putchar(share.buffer[i]); /*循环输出缓存区字母*/
            pthread_mutex_unlock(&lock_it);/*解锁互斥量*/
        }
        printf("W %2d:exiting\n",pthread_self());
        return NULL;
}
```

程序每读入 5 个字母，打印一遍，并清空缓存区，循环执行直到 Y 为止。

程序运行结果如下：

```
#cc –lpthread –o readandwrite readandwrite.c
#./readandwrire
R 1082330304: stareing
W 1090718784:string
W 1090718784:Waiting
R 1082330304:Got char[A]
R 1082330304:Got char[B]
R 1082330304:Got char[C]
R 1082330304:Got char[D]
R 1082330304:Got char[E]
R 1082330304:signaling full
W 1090718784:wring buffer
```

ABCDE
W 1090718784:Waiting
R 1082330304:Got char[F]
……

三、条件变量属性

使用条件变量之前要先进行初始化。可以像我们前面那样可静态初始化 pthread_cond_t my_condition=PTHREAD_COND_INITIALIZER;也可以利用函数 pthread_cond_init 动态初始化。条件变量属性类型为 pthread_condattr_t，它们由以下函数初始化或摧毁。

名称::	pthread_condattr_init/pthread_condattr_destroy
功能:	初始化/回收 pthread_condattr_t 结构
头文件:	#include <pthread.h>
函数原形:	int pthread_condattr_init(pthread_condattr_t *attr); int pthread_condattr_destroy(pthread_condattr_t *attr);
参数:	
返回值:	若成功返回 0，若失败返回错误编号。

11.

一旦某个条件变量对象被初始化了，我们就可以利用下面函数来查看或修改特定属性了。

名称::	pthread_condattr_getpshared/pthread_condattr_setpshared
功能:	查看或修改条件变量属性
头文件:	#include <pthread.h>
函数原形:	int pthread_condattr_init(const pthread_condattr_t *restrict attr); int pthread_condattr_destroy(pthread_rwlockattr_t *attr,int pshared);
参数:	
返回值:	若成功返回 0，若失败返回错误编号。

12.

pthread_condattr_getpshared 函数在由 valptr 指向的整数中返回这个属性的当前值，pthread_condattr_setpshared 则根据 value 的值设置这个属性的当前值。value 的值可以是 PTHREAD_PROCESS_PRIVATE 或 PTHREAD_PROCESS_SHARED(进程间共享).

四、条件变量与互斥锁、信号量的区别

到这里，我们把 posix 的互斥锁、信号量、条件变量都接受完了，下面我们来比较一下他们。

1.互斥锁必须总是由给它上锁的线程解锁，信号量的挂出即不必由执行过它的等待操作的同一进程执行。一个线程可以等待某个给定信号灯，而另一个线程可以挂出该信号灯。

2.互斥锁要么锁住，要么被解开（二值状态，类型二值信号量）。

3.由于信号量有一个与之关联的状态（它的计数值），信号量挂出操作总是被记住。然而当向一个条件变量发送信号时，如果没有线程等待在该条件变量上，那么该信号将丢失。

4.互斥锁是为了上锁而优化的，条件变量是为了等待而优化的，信号灯即可用于上锁，也可用于等待，因而可能导致更多的开销和更高的复杂性。

第六章 共享内存

一、什么是共享内存区

共享内存区是最快的可用 IPC 形式。它允许多个不相关的进程去访问同一部分逻辑内存。如果需要在两个运行中的进程之间传输数据，共享内存将是一种效率极高的解决方案。一旦这样的内存区映射到共享它的进程的地址空间，这些进程间数据的传输就不再涉及内核。这样就可以减少系统调用时间，提高程序效率

共享内存是由 IPC 为一个进程创建的一个特殊的地址范围，它将出现在进程的地址空间中。其他进程可以把同一段共享内存段“连接到”它们自己的地址空间里去。所有进程都可以访问共享内存中的地址。如果一个进程向这段共享内存写了数据，所做的改动会立刻被有访问同一段共享内存的其他进程看到。

要注意的是共享内存本身没有提供任何同步功能。也就是说，在第一个进程结束对共享内存的写操作之前，并没有什么自动功能能够预防第二个进程开始对它进行读操作。共享内存的访问同步问题必须由程序员负责。可选的同步方式有互斥锁、条件变量、读写锁、纪录锁、信号灯。

二、**mmap**

在将共享内存前我们要先来介绍下面几个函数。

mmap 函数把一个文件或一个 Posix 共享内存区对象映射到调用进程的地址空间。使用该函数有三个目的：

1.使用普通文件以提供内存映射 I/O

2.使用特殊文件以提供匿名内存映射。

3.使用 shm_open 以提供无亲缘关系进程间的 Posix 共享内存区。

1.

名称::	mmap
功能:	把 I/O 文件映射到一个存储区域中
头文件:	#include <sys/mman.h>
函数原形:	void *mmap(void *addr,size_t len,int prot,int flag,int filedes,off_t off);
参数:	addr 指向映射存储区的起始地址 len 映射的字节 prot 对映射存储区的保护要求 flag flag 标志位 filedes 要被映射文件的描述符 off 要映射字节在文件中的起始偏移量
返回值:	若成功则返回映射区的起始地址，若出错则返回 MAP_FAILED

addr 参数用于指定映射存储区的起始地址。通常将其设置为 NULL，这表示由系统选择该映射区的起始地址。

filedes 指要被映射文件的描述符。在映射该文件到一个地址空间之前，先要打开该文件。len 是映射的字节数。

off 是要映射字节在文件中的起始偏移量。通常将其设置为 0。

prot 参数说明对映射存储区的保护要求。可将 prot 参数指定为 PROT_NONE,或者是 PROT_READ(映射区可读),PROT_WRITE（映射区可写）,PROT_EXEC（映射区可执行）任意组合的按位或，也可以是 PROT_NONE(映射区不可访问)。对指定映射存储区的保护要求不能超过文件 open 模式访问权限。

flag 参数影响映射区的多种属性:

MAP_FIXED 返回值必须等于 addr.因为这不利于可移植性，所以不鼓励使用此标志。

MAP_SHARED 这一标志说明了本进程对映射区所进行的存储操作的配置。此标志指定存储操作修改映射文件。

MAP_PRIVATE 本标志导致对映射区建立一个该映射文件的一个私有副本。所有后来对该映射区的引用都是引用该副本，而不是原始文件。
要注意的是必须指定 MAP_FIXED 或 MAP_PRIVATE 标志其中的一个，指定前者是对存储映射文件本身的一个操作，而后者是对其副本进行操作。

mmap 成功返回后，fd 参数可以关闭。该操作对于由 mmap 建立的映射关系没有影响。为从某个进程的地址空间删除一个映射关系，我们调用 munmap.

名称::	munmap
功能:	解除存储映射
头文件:	#include <sys/mman.h>
函数原形:	int munmap(caddr_t addr,size_t len);
参数:	addr 指向映射存储区的起始地址 len 映射的字节
返回值:	若成功则返回 0，若出错则返回-1

2.

其中 addr 参数是由 mmap 返回的地址，len 是映射区的大小。再次访问这些地址导致向调用进程产生一个 SIGSEGV 信号。
如果被映射区是使用 MAP_PRIVATE 标志映射的，那么调用进程对它所作的变动都被丢弃掉。

内核的虚存算法保持内存映射文件（一般在硬盘上）与内存映射区（在内存中）的同步（前提它是 MAP_SHARED 内存区）。这就是说，如果我们修改了内存映射到某个文件的内存区中某个位置的内容，那么内核将在稍后某个时刻相应地更新文件。然而有时候我们希望确信硬盘上的文件内容与内存映射区中的文件内容一致，于是调用 msync 来执行这种同步。

名称::	msync
功能:	同步文件到存储器
头文件:	#include <sys/mman.h>
函数原形:	int msync(void *addr,size_t len,int flags);
参数:	addr 指向映射存储区的起始地址 len 映射的字节 prot flags
返回值:	若成功则返回 0，若出错则返回-1

3.

其中 addr 和 len 参数通常指代内存中的整个内存映射区，不过也可以指定该内存区的一个子集。flags 参数为 MS_ASYNC(执行异步写)，MS_SYNC（执行同步写），MS_INVALIDATE（使高速缓存的数据实效）。其中 MS_ASYNC 和 MS_SYNC 这两个常值中必须指定一个，但不能都指定。它们的差别是，一旦写操作已由内核排入队列，MS_ASYNC 即返回，而 MS_SYNC 则要等到写操作完成后才返回。如果还指定了 MS_INVALIDATE，那么与其最终拷贝不一致的文件数据的所有内存中拷贝都失效。后续的引用将从文件取得数据。

名称::	memcpy
功能:	复制映射存储区
头文件:	#include <string.h>
函数原形:	void *memcpy(void *dest,const void *src,size_t n);
参数:	dest 待复制的映射存储区 src 复制后的映射存储区 n 待复制的映射存储区的大小
返回值:	返回 dest 的首地址

4.

memcpy 拷贝 n 个字节从 dest 到 src。

下面就是利用 mmap 函数影射 I/O 实现的 cp 命令。

```
/*mycp.c*/
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(int argc,char *argv[])
{
    int fdin,fdout;
```

```
    void *arc,dst;
    struct stat statbuf;

    if(argc!=3)
    {
        printf("please input two file!\n");
        exit(1);
    }
    if((fdin=open(argv[1],O_RDONLY))<0) /*打开原文件*/
        perror(argv[1]);
    if((fdout=open(argv[2],O_RDWR|O_CREAT|O_TRUNC))<0)/*创建并打开目标文件*/
        perror(argv[2]);

    if(fstat(fdin,&statbuf)<0) /*获得文件大小信息*/
        printf("fstat error");

    if(lseek(fdout,statbuf.st_size-1,SEEK_SET)==-1)/*初始化输出映射存储区*/
        printf("lseek error");
    if(write(fdout,"1")!=1)
        printf("write error");

    if((src=mmap(0,statbuf.st_size,PROT_READ,MAP_SHARED,fdin,0))==MAP_FAILED)
        /*映射原文件到输入的映射存储区*/
        printf("mmap error);
    if((dst=mmap(0,statbuf.st_size,PROT_READ|PROT_WRITE,MAP_SHARED,fdout,0))
==MAP_FAILED) /*映射目标文件到输出的映射存储区*/
        printf("mmap error);
    memcpy(dst,src,statbuf.st_size);/*复制映射存储区*/
    munmap(src,statbuf.st_size); /*解除输入映射*/
    munmap(dst,statbuf.st_size); /*解除输出映射*/
    close(fdin);
    close(fdout);
}
```

下面是运行结果：

```
#cc –o mycp mycp.c
#./mycp test1 test2
```

三、 **posix** 共享内存函数

posix 共享内存区涉及两个步骤：

1、指定一个名字参数调用 shm_open,以创建一个新的共享内存区对象或打开一个以存在的共享内存区对象。

2、调用 mmap 把这个共享内存区映射到调用进程的地址空间。传递给 shm_open 的名字参数随后由希望共享该内存区的任何其他进程使用。

名称::	shm_open
功能:	打开或创建一个共享内存区
头文件:	#include <sys/mman.h>
函数原形:	int shm_open(const char *name,int oflag,mode_t mode);
参数:	name　共享内存区的名字 cflag　标志位 mode　权限位
返回值:	成功返回 0，出错返回-1

5.

oflag 参数必须含有 O_RDONLY 和 O_RDWR 标志，还可以指定如下标志：O_CREAT,O_EXCL 或 O_TRUNC.

mode 参数指定权限位，它指定 O_CREAT 标志的前提下使用。

shm_open 的返回值是一个整数描述字，它随后用作 mmap 的第五个参数。

名称::	shm_unlink
功能:	删除一个共享内存区
头文件:	#include <sys/mman.h>
函数原形:	int shm_unlink(const char *name);
参数:	name　共享内存区的名字
返回值:	成功返回 0，出错返回-1

6.

shm_unlink 函数删除一个共享内存区对象的名字，删除一个名字仅仅防止后续的 open,mq_open 或 sem_open 调用取得成功。

下面是创建一个共享内存区的例子：

```
/*shm_open.c 创建共享内存区*/
#include <sys/mman.h>
#include <stdio.h>
```

```
#include <fcntl.h>

int main(int argc,char **argv)
{
      int shm_id;

      if(argc!=2)
      {
          printf("usage:shm_open <pathname>\n");
          exit(1);
      }
      shm_id=shm_open(argv[1],O_RDWR|O_CREAT,0644);
      printf("shmid:%d\n",shm_id);
      shm_unlink(argv[1]);
}
```

下面是运行结果，注意编译程序我们要加上"-lrt"参数。

```
#cc –lrt –o shm_open shm_open.c
#./shm_open test
shm_id:3
```

四、 ftruncate 和 fstat 函数

普通文件或共享内存区对象的大小都可以通过调用 ftruncate 修改。

7.

名称::	ftruncate
功能:	调整文件或共享内存区大小
头文件:	#include <unistd.h>
函数原形:	int ftruncate(int fd,off_t length);
参数:	fd 描述符 length 大小
返回值:	成功返回 0，出错返回-1

当打开一个已存在的共享内存区对象时，我们可调用 fstat 来获取有关该对象的信息。

名称::	fstat
功能:	获得文件或共享内存区的信息
头文件:	#include <unistd.h> #include <sys/types.h> #include <sys/stat.h>
函数原形:	int stat(const char *file_name,struct stat *buf);
参数:	file_name 文件名 buf stat 结构
返回值:	成功返回 0，出错返回-1

8.

对于普通文件 stat 结构可以获得 12 个以上的成员信息，然而当 fd 指代一个共享内存区对象时，只有四个成员含有信息。

```
struct stat{
      mode_t st_mode;
      uid_t st_uid;
      gid_t st_gid;
      off_t st_size;
      };
```

```
/*shm_show.c 显示共享区信息*/
#include <unistd.h>
#include <sys/type.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/mman.h>

int main(int argc,char **argv)
{
   int shm_id;
      struct stat buf;
```

```
    if(argc!=2)
    {
        printf("usage:shm_open <pathname>\n");
        exit(1);
    }
    shm_id=shm_open(argv[1],O_RDWR|O_CREAT,0644);/*创建共享内存*/
    ftruncate(shm_id,100);/*修改共享内存的打开*/
    fstat(shm_id,&buf); /*把共享内存的信息记录到 buf 中*/
    printf("uid_t:%d\n",buf.st_uid); /*共享内存区所有者 ID*/
    printf("git_t:%d\n",buf.st_gid); /*共享内存区所有者组 ID*/
    printf("size :%d\n",buf.st_size); /*共享内存区大小*/
}
```

下面是运行结果：

```
#cc –lrt –o shm_show shm_show.c
#./shm_show test
uid_t:0
git_t:0
size:100
```

五、共享内存区的写入和读出

上面我们介绍了 mmap 函数，下面我们就可以通过这些函数，把进程映射到共享内存区。

然后我们就可以通过共享内存区进行进程间通信了。

下面是共享内存区写入的例子：

```
/*shm_write.h 写入/读出共享内存区*/
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>

int main(int argc,char **argv)
{
    int shm_id;
    struct stat buf;
    char *ptr;

    if(argc!=2)
    {
        printf("usage:shm_open <pathname>\n");
        exit(1);
    }
    shm_id=shm_open(argv[1],O_RDWR|O_CREAT,0644);/*创建共享内存区*/
    ftruncate(shm_id,100);/*修改共享区大小*/
```

```
    fstat(shm_id,&buf);
    ptr=mmap(NULL,buf.st_size,PROT_READ|
PROT_WRITE,MAP_SHARED,shm_id,0);/*连接共享内存区*/
    strcpy(ptr,"hello linux");/*写入共享内存区*/
    printf("%s\n",ptr);/*读出共享内存区*/
    shm_unlink(argv[1]);/*删除共享内存区*/
}
```

下面是运行结果：

```
#cc –lrt –o shm_write shm_write.c
#./shm_write test
hello linux
```

六、 程序例子

下面是利用 pisix 共享内存区实现进程间通信的例子：服务器进程读出共享内存区内容，然后清空。客户进程向共享内存区写入数据。直到用户输入"q"程序结束。程序用 posix 信号量实现互斥。

```
/*server.c 服务器程序*/
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <semaphore.h>

int main(int argc,char **argv)
{
    int shm_id;
    char *ptr;
    sem_t *sem;

    if(argc!=2)
    {
        printf("usage:shm_open <pathname>\n");
        exit(1);
    }
    shm_id=shm_open(argv[1],O_RDWR|O_CREAT,0644);/*创建共享内存区*/
    ftruncate(shm_id,100);/*调整共享内存区大小*/
    sem=sem_open(argv[1],O_CREAD,0644,1);/*创建信号量*/
    ptr=mmap(NULL,100,PROT_READ|
PROT_WRITE,MAP_SHARED,shm_id,0);/*连接共享内存区*/
    strcpy(ptr,"\0");
    while(1)
    {
        if((strcmp(ptr,"\0"))==0)/*如果为空，则等待*/
```

```
            continue;
        else
        {
            if((strcmp(ptr,”q\n”))==0)/*如果内存为 q\n 退出循环*/
                break;
            sem_wait(sem);/*申请信号量*/
            printf(“server:%s”,ptr);/*输入共享内存区内容*/
            strcpy(ptr,”\0”);/*清空共享内存区*/
            sem_pose(sem);/*释放信号量*/
         }
         sem_unlink(argv[1]);/*删除信号量*/
         shm_unlink(argv[1]);/*删除共享内存区*/
    }
}
```

```
/*server.c 服务器程序*/
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <semaphore.h>
#include <stdio.h>

int main(int argc,char **argv)
{
      int shm_id;
      char *ptr;
      sem_t *sem;

      if(argc!=2)
      {
          printf(“usage:shm_open <pathname>\n”);
          exit(1);
      }
      shm_id=shm_open(argv[1],0);/*打开共享内存区
      sem=sem_open(argv[1],0);/*打开信号量*/
      ptr=mmap(NULL,100,PROT_READ|
PROT_WRITE,MAP_SHARED,shm_id,0);/*连接共享内存区*/
      while(1)
      {
         sem_wait(sem);/*申请信号量*/
         fgets(ptr,10,stdin);/*从键盘读入数据到共享内存区*/
```

```
        printf("user:%s",ptr);
        if((strcmp(ptr,"q\n"))==0)
            exit(0);
        sem_pose(sem);/*释放信号量*/
        sleep(1);
    }
    exit(0);
}
```

#cc –lrt –o server server.c
#cc –lrt –o user user.c
#./server test&
#./user test
输入:abc
user:abc
server:abc
输入:123
user:123
server:123
输入:q
user:q