LINUX 多线程

1. Linux 多线程概述

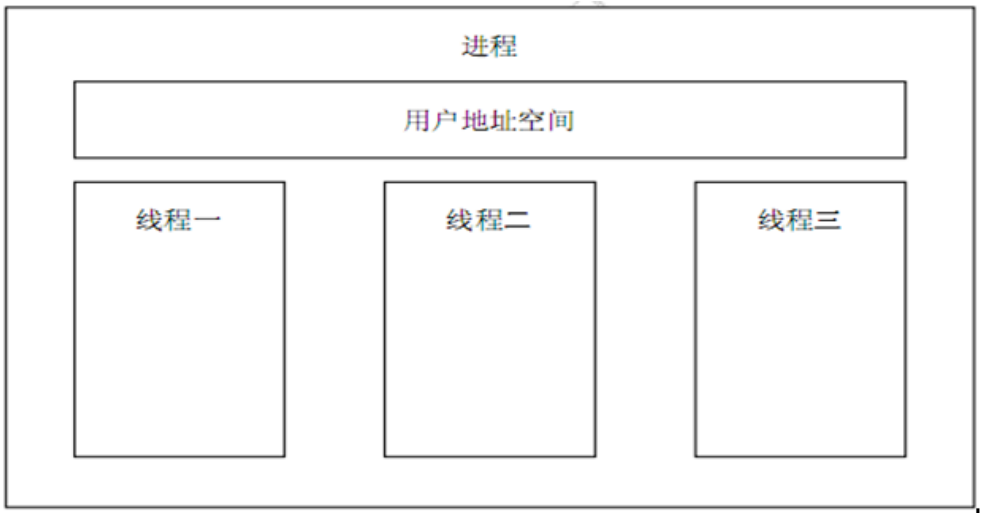
1.1. 概述

进程是系统中程序执行和资源分配的基本单位。每个进程有自己的数据段、代码段和堆栈段。这就造成进程在进行切换等操作时都需要有比较负责的上下文切换等动作。为了进一步减少处理器的空转时间支持多处理器和减少上下文切换开销，也就出现了线程。

线程是在共享内存空间中并行执行的多道执行路径，是一个更加接近于执行体的概念，拥有独立的执行序列，是进程的基本调度单元，每个进程至少都有一个 main 线程。它与同进程中的其他线程共享进程空间｛堆 代码 数据 文件描述符 信号等｝，只拥有自己的栈空间，大大减少了上下文切换的开销。

线程和进程在使用上各有优缺点：线程执行开销小，占用的 CPU 少，线程之间的切换快，但不利于资源的管理和保护；而进程正相反。从可移植性来讲，多进程的可移植性要好些。

同进程一样，线程也将相关的变量值放在线程控制表内。一个进程可以有多个线程，也就是有多个线程控制表及堆栈寄存器，但却共享一个用户地址空间。要注意的是，由于线程共享了进程的资源和地址空间，因此，任何线程对系统资源的操作都会给其他线程带来影响。



1.2. 线程分类

按调度者分为用户级线程和核心级线程

- 用户级线程：主要解决上下文切换问题，调度算法和调度过程全部由用户决定，在运行时不需要特定的内核支持。缺点是无法发挥多处理器的优势
- 核心级线程：允许不同进程中的线程按照同一相对优先调度方法调度，发挥多处理器的并发优势

现在大多数系统都采用用户级线程和核心级线程并存的方法。一个用户级线程可以对应一个或多个核心级线程，也就是“一对一”或“一对多”模型。

1.3. 线程创建的 Linux 实现

Linux 的线程是通过用户级的函数库实现的，一般采用 pthread 线程库实现线程的访问和控制。它用第 3 方 posix 标准的 pthread，具有良好的可移植性。 编译的时候要在后面加上 **–lpthread**

	创建	退出	等待
多进程	fork()	exit()	wait()
多线程	pthread_create	pthread_exit()	pthread_join()

2. 线程的创建和退出

创建线程实际上就是确定调用该线程函数的入口点，线程的创建采用函数 pthread_create。在线程创建以后，就开始运行相关的线程函数，在该函数运行完之后，线程就退出，这也是线程退出的一种方式。另一种线程退出的方式是使用函数 pthread_exit()函数，这是线程主动退出行为。这里要注意的是，在使用线程函数时，不能随意使用 exit 退出函数进行出错处理，由于 exit 的作用是使调用进程终止，往往一个进程包括了多个线程，所以在线程中通常使用 pthread_exit 函数来代替进程中的退出函数 exit。

由于一个进程中的多个线程是共享数据段的，因此通常在线程退出之后，退出线程所占用的资源并不会随着线程的终止而得到释放。正如进程之间可以通过 wait()函数系统调用来同步终止并释放资源一样，线程之间也有类似的机制，那就是 pthread_join 函数。pthread_join 函数可以用于将当前线程挂起，等待线程的结束。这个函数是一个线程阻塞函数，调用它的函数将一直等待直到被等待的线程结束为止，当函数返回时，被等待线程的资源被回收。

函数原型：

```
#include <pthread.h>
int pthread_create(pthread_t* thread, pthread_attr_t * attr, void *(*start_routine)(void *), void * arg);
void pthread_exit(void *retval);
pthread_t pthread_self(void);
```

通常的形式为：

```
pthread_t pthid;
pthread_create(&pthid,NULL,pthfunc,NULL);或 pthread_create(&pthid,NULL,pthfunc,(void*)3);
pthread_exit(NULL);或 pthread_exit((void*)3);//3 作为返回值被 pthread_join 函数捕获。
```

函数 pthread_create 用来创建线程。返回值：成功，则返回 0；失败，则返回对应错误码。各参数描述如下：

- 参数 thread 是传出参数，保存新线程的标识；
- 参数 attr 是一个结构体指针，结构中的元素分别指定新线程的运行属性,attr 可以用 pthread_attr_init 等函数设置各成员的值，但通常传入为 NULL 即可；
- 参数 start_routine 是一个函数指针，指向新线程的入口点函数，线程入口点函数带有一个 void *的参数由 pthread_create 的第 4 个参数传入；
- 参数 arg 用于传递给第 3 个参数指向的入口点函数的参数，可以为 NULL，表示不传递。

函数 pthread_exit 表示线程的退出。其参数可以被其它线程用 pthread_join 函数捕获。

示例：

```
#include <stdio.h>
#include <pthread.h>
void *ThreadFunc(void *pArg)   //参数的值为 123
{
    int i = 0;
```

```
        for(; i<10; i++)
        {
            printf("Hi,I'm child thread,arg is:%d\n", (int)pArg);
            sleep(1);
        }
        pthread_exit(NULL);
    }
    int main()
    {
        pthread_t thdId;
        pthread_create(&thdId, NULL, ThreadFunc, (void *)123 );
        int i = 0;
        for(; i<10; i++)
        {
            printf("Hi,I'm main thread,child thread id is:%x\n", thdId);
            sleep(1);
        }
        return 0;
    }
```

编译时需要带上线程库选项：

```
gcc  -o  a  a.c  -lpthread
```

3. 线程的等待退出

3.1. 等待线程退出

线程从入口点函数自然返回，或者主动调用 pthread_exit()函数，都可以让线程正常终止

线程从入口点函数自然返回时，函数返回值可以被其它线程用 pthread_join 函数获取

pthread_join 原型为：

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
```

1. 该函数是一个阻塞函数，一直等到参数 th 指定的线程返回；与多进程中的 wait 或 waitpid 类似。thread_return 是一个传出参数，接收线程函数的返回值。如果线程通过调用 pthread_exit()终止，则 pthread_exit()中的参数相当于自然返回值，照样可以被其它线程用 pthread_join 获取到。

2.thid 传递 0 值时，join 返回 ESRCH 错误。

Example：返回值的例子

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
void *ThreadFunc(void *pArg)
{
    int iArg = (int)pArg;   //将 void*转换为 int
    sleep(iArg);
    if(iArg < 3)
```

```
            return (void *)(iArg*2);
        else
            pthread_exit((void *)(iArg*2)); //和 reaturn 达到的效果一样，都可以用于正常返回
}
int main()
{
    pthread_t thdId;
    long iRet = 0;
    pthread_create(&thdId, NULL, ThreadFunc, (void *)2 ); //传递参数值为 2
    pthread_join(thdId,(void **)&iRet);   //接收子线程的返回值
    printf("The first child thread ret is:%d\n",iRet);
    pthread_create(&thdId, NULL, ThreadFunc, (void *)4 );
    pthread_join(thdId,(void **)&iRet);
    printf("The second child thread ret is:%d\n",iRet);
    return 0;
}
```

2. 该函数还有一个非常重要的作用，由于一个进程中的多个线程共享数据段，因此通常在一个线程退出后，退出线程所占用的资源并不会随线程结束而释放。如果 th 线程类型并不是自动清理资源类型的，则 th 线程退出后，线程本身的资源必须通过其它线程调用 pthread_join 来清除,这相当于多进程程序中的 **waitpid**。

Example：子线程释放空间

```
#include <stdio.h>
#include <pthread.h>
#include <malloc.h>
void* threadfunc(void *args)
{
    char *p = (char*)malloc(10);              //自己分配了内存
    int i = 0;
    for(; i < 10; i++)
    {
        printf("hello,my name is wangxiao!\n");
        sleep(1);
    }
    free(p);                    //如果父线程中没有调用 pthread_cancel，此处可以执行
    printf("p is freed\n");
    pthread_exit((void*)3);
}
int main()
{
    pthread_t pthid;
    pthread_create(&pthid, NULL, threadfunc, NULL);
    int i = 1;
    for(; i < 5; i++)      //父线程的运行次数比子线程的要少，当父线程结束的时候，如果没有
pthread_join 函数等待子线程执行的话，子线程也会退出。
    {
```

```
            printf("hello,nice to meet you!\n");
            sleep(1);
        //    if(i % 3 == 0)
        //        pthread_cancel(pthid);    //表示当 i%3==0 的时候就取消子线程，该函数将导致子线程直接退出，不会执行上面紫色的 free 部分的代码，即释放空间失败。要想释放指针类型的变量 p，此时必须要用 pthread_cleanup_push 和 pthread_cleanup_pop 函数释放空间，见后面的例子
        }
        int retvalue = 0;
        pthread_join(pthid,(void**)&retvalue); //等待子线程释放空间，并获取子线程的返回值
        printf("return value is :%d\n",retvalue);
        return 0;
}
```

pthread_join 不回收堆内存的，只回收线程的栈内存和内核中的 struct task_struct 结构占用的内存。

3.2. 线程的取消

线程也可以被其它线程杀掉，在 Linux 中的说法是一个线程被另一个线程取消(cancel)。

线程取消的方法是一个线程向目标线程发 cancel 信号，但是如何处理 cancel 信号则由目标线程自己决定，目标线程或者忽略、或者立即终止、或者继续运行至 cancelation-point(取消点)后终止。

取消点：

根据 POSIX 标准，pthread_join()、pthread_testcancel()、pthread_cond_wait()、pthread_cond_timedwait()、sem_wait()、sigwait()等函数以及 read()、write()等会引起阻塞的系统调用都是 Cancelation-point，而其他 pthread 函数都不会引起 Cancelation 动作。但是 pthread_cancel 的手册页声称，由于 Linux 线程库与 C 库结合得不好，因而目前 C 库函数都不是 Cancelation-point；但 CANCEL 信号会使线程从阻塞的系统调用中退出，并置 EINTR 错误码，因此可以在需要作为 Cancelation-point 的系统调用前后调用 pthread_testcancel()，从而达到 POSIX 标准所要求的目标，即如下代码段：

```
pthread_testcancel();
retcode = read(fd, buffer, length);
pthread_testcancel();
```

但是从 RedHat9.0 的实际测试来看，至少有些 C 库函数的阻塞函数是取消点，如 read(),getchar()等，而 sleep()函数不管线程是否设置了 pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,NULL)，都起到取消点作用。总之，线程的取消一方面是一个线程强行杀另外一个线程，从程序设计角度看并不是一种好的风格，另一方面目前 Linux 本身对这方面的支持并不完善，所以在实际应用中应该谨慎使用！！

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

增加 man 信息 ， apt-get install manpages-posix-dev

3.3. 线程终止清理函数

不论是可预见的线程终止还是异常终止，都会存在资源释放的问题，在不考虑因运行出错而退出的前提下，如何保证线程终止时能顺利的释放掉自己所占用的资源，特别是锁资源，就是一个必须考虑解决的问题。

最经常出现的情形是资源独占锁的使用：线程为了**访问临界共享资源而为其加上锁**，但在访问过

程中该线程被外界取消，或者发生了中断，则该临界资源将永远处于锁定状态得不到释放。外界取消操作是不可预见的，因此的确需要一个机制来简化用于资源释放的编程。

在 POSIX 线程 API 中提供了一个 pthread_cleanup_push()/pthread_cleanup_pop()函数对用于自动释放资源--从 pthread_cleanup_push()的调用点到 pthread_cleanup_pop()之间的程序段中的终止动作都将执行 pthread_cleanup_push()所指定的清理函数。API 定义如下：

void pthread_cleanup_push(void (*routine) (void *), void *arg)

void pthread_cleanup_pop(int execute)

pthread_cleanup_push()/pthread_cleanup_pop()采用**先入后出**的栈结构管理

void routine(void *arg)函数在调用 pthread_cleanup_push()时压入清理函数栈，多次对 pthread_cleanup_push()的调用将在清理函数栈中形成一个函数链，在执行该函数链时按照压栈的相反顺序弹出。execute 参数表示执行到 pthread_cleanup_pop()时是否在弹出清理函数的同时执行该函数，为 0 表示不执行，非 0 为执行；这个参数并不影响异常终止时清理函数的执行。

pthread_cleanup_push()/pthread_cleanup_pop()是以宏方式实现的，这是 pthread.h 中的宏定义：

```
#define pthread_cleanup_push(routine,arg) \
{
    struct  _pthread_cleanup_buffer  _buffer; \
    _pthread_cleanup_push (&_buffer, (routine), (arg));
#define pthread_cleanup_pop(execute) \
    _pthread_cleanup_pop (&_buffer, (execute));
}
```

可见，pthread_cleanup_push()带有一个"{"，而 pthread_cleanup_pop()带有一个"}"，因此这两个函数必须成对出现，且必须位于程序的同一级别的代码段中才能通过编译。

pthread_cleanup_pop 的参数 execute 如果为非 0 值,则按栈的顺序注销掉一个原来注册的清理函数，并执行该函数；当 pthread_cleanup_pop()函数的参数为 0 时，仅仅在线程调用 pthread_exit 函数或者其它线程对本线程调用 pthread_cancel 函数时，才在弹出"清理函数"的同时执行该"清理函数"。

示例：

```
#include <stdio.h>
#include <pthread.h>
void CleanFunc(void *pArg)
{
    printf("CleanFunc(%d)\n",(int)pArg);
}
void *ThreadFunc(void *pArg)
{
    pthread_cleanup_push(CleanFunc,(void *)1);
    pthread_cleanup_push(CleanFunc,(void *)2);
    sleep(2);
    pthread_cleanup_pop(1);
    pthread_cleanup_pop(1);

}
int main()
{
    pthread_t thdId;
    pthread_create(&thdId, NULL, ThreadFunc, (void *)2);
    pthread_join(thdId,NULL);
```

```
        return 0;
    }
```

运行结果为：

```
    CleanFunc(2)
    CleanFunc(1)
```

如果将里面的两次 pthread_cleanup_pop(1);改为 pthread_cleanup_pop(0);推测一下结果是怎样？

→没有任何输出（此时 CleanFunc 函数得不到执行）

如果修改为 0 之后，再在 sleep(2)之后添加 pthread_exit(NULL);则此时的结果又是如何：

→跟 pthread_cleanup_pop(1);实现的结果一样了。

Example：用 pthread_cleanup_push 和 pthread_cleanup_pop 来释放子线程分配的内存空间

```
    #include <stdio.h>
    #include <pthread.h>
    #include <malloc.h>
    void freemem(void * args)
    {
        free(args);
        printf("clean up the memory!\n");
    }
    void* threadfunc(void *args)
    {
        char *p = (char*)malloc(10);                //自己分配了内存
        pthread_cleanup_push(freemem,p);
        int i = 0;
        for(; i < 10; i++)
        {
            printf("hello,my name is wangxiao!\n");
            sleep(1);
        }
        pthread_exit((void*)3);
        pthread_cleanup_pop(0);
    }
    int main()
    {
        pthread_t pthid;
        pthread_create(&pthid, NULL, threadfunc, NULL);
        int i = 1;
        for(; i < 5; i++)       //父线程的运行次数比子线程的要少，当父线程结束的时候，如果没有
    pthread_join 函数等待子线程执行的话，子线程也会退出，即子线程也只执行了 4 次。
        {
            printf("hello,nice to meet you!\n");
            sleep(1);
            if(i % 3 == 0)
                pthread_cancel(pthid);     //表示当 i%3==0 的时候就取消子线程，该函数将导致直接
    退出，不会执行上面紫色的 free 部分的代码，即释放空间失败。要想释放指针类型的变量 p，必
    须要用 pthread_cleanup_push 和 pthread_cleanup_pop 函数释放空间
        }
```

```
    int retvalue = 0;
    pthread_join(pthid,(void**)&retvalue); //等待子线程释放空间，并获取子线程的返回值
    printf("return value is :%d\n",retvalue);
    return 0;
}
```

4. 线程的同步与互斥

4.1. 线程的互斥

在 Posix Thread 中定义了一套专门用于线程互斥的 mutex 函数。mutex 是一种简单的加锁的方法来控制对共享资源的存取，这个互斥锁只有两种状态（上锁和解锁），可以把互斥锁看作某种意义上的全局变量。为什么需要加锁，就是因为多个线程共用进程的资源，要访问的是公共区间时（全局变量），当一个线程访问的时候，需要加上锁以防止另外的线程对它进行访问，实现资源的独占。在一个时刻只能有一个线程掌握某个互斥锁，拥有上锁状态的线程能够对共享资源进行操作。若其他线程希望上锁一个已经上锁了的互斥锁，则该线程就会挂起，直到上锁的线程释放掉互斥锁为止。

1. 创建和销毁锁

有两种方法创建互斥锁，静态方式和动态方式。

• 静态方式：

POSIX 定义了一个宏 PTHREAD_MUTEX_INITIALIZER 来静态初始化互斥锁，方法如下：

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

在 Linux Threads 实现中，pthread_mutex_t 是一个结构，而 PTHREAD_MUTEX_INITIALIZER 则是一个宏常量。

• 动态方式：

动态方式是采用 pthread_mutex_init()函数来初始化互斥锁，API 定义如下：

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)
```

其中 mutexattr 用于指定互斥锁属性（见下），如果为 NULL 则使用缺省属性。通常为 NULL

pthread_mutex_destroy()用于注销一个互斥锁，API 定义如下：

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

销毁一个互斥锁即意味着释放它所占用的资源，且要求锁当前处于开放状态。由于在 Linux 中，互斥锁并不占用任何资源，因此 Linux Threads 中的 pthread_mutex_destroy()除了检查锁状态以外（锁定状态则返回 EBUSY）没有其他动作。

2. 互斥锁属性设置

互斥锁的属性在创建锁的时候指定，不同的锁类型在试图对一个已经被锁定的互斥锁加锁时表现不同也就是是否阻塞等待。有三个值可供选择：

• PTHREAD_MUTEX_TIMED_NP，这是缺省值（直接写 NULL 就是表示这个缺省值），也就是普通锁(或快速锁)。当一个线程加锁以后，其余请求锁的线程将形成一个阻塞等待队列，并在解锁后按优先级获得锁。这种锁策略保证了资源分配的公平性

示例：初始化一个快速锁。

```
pthread_mutex_t    lock;
```

```
pthread_mutex_init(&lock, NULL);
```

• PTHREAD_MUTEX_RECURSIVE，嵌套锁，允许同一个线程对同一个锁成功获得多次，并通过多次 unlock 解锁。如果是不同线程请求，则在加锁线程解锁时重新竞争。

示例：初始化一个嵌套锁。

```
pthread_mutex_t    lock;
pthread_mutexattr_t    mutexattr;
pthread_mutexattr_init(&mutexattr)
pthread_mutexattr_settype(&mutexattr, PTHREAD_MUTEX_RECURSIVE);
pthread_mutex_init(&lock, &mutexattr);
```

•PTHREAD_MUTEX_ERRORCHECK，检错锁，如果同一个线程请求同一个锁，则返回 EDEADLK，否则与 PTHREAD_MUTEX_TIMED 类型动作相同。这样就保证当不允许多次加锁时不会出现最简单情况下的死锁。如果锁的类型是快速锁，一个线程加锁之后，又加锁，则此时就是死锁。

示例：初始化一个嵌套锁。

```
pthread_mutex_t    lock;
pthread_mutexattr_t    mutexattr;
pthread_mutexattr_settype(&mutexattr, PTHREAD_MUTEX_ERRORCHECK);
pthread_mutex_init(&lock, &mutexattr);
```

3．锁操作

锁操作主要包括

加锁	int pthread_mutex_lock(pthread_mutex_t *mutex)
解锁	int pthread_mutex_unlock(pthread_mutex_t *mutex)
测试加锁	int pthread_mutex_trylock(pthread_mutex_t *mutex)

• pthread_mutex_lock：加锁，不论哪种类型的锁，都不可能被两个不同的线程同时得到，而必须等待解锁。对于普通锁类型，解锁者可以是同进程内任何线程；而检错锁则必须由加锁者解锁才有效，否则返回 EPERM；对于嵌套锁，文档和实现要求必须由加锁者解锁，但实验结果表明并没有这种限制，这个不同目前还没有得到解释。在同一进程中的线程，如果加锁后没有解锁，则任何其他线程都无法再获得锁。

• pthread_mutex_unlock：根据不同的锁类型，实现不同的行为：

对于快速锁，pthread_mutex_unlock 解除锁定；

对于递规锁，pthread_mutex_unlock 使锁上的引用计数减 1；

对于检错锁，如果锁是当前线程锁定的，则解除锁定，否则什么也不做。

•pthread_mutex_trylock：语义与 pthread_mutex_lock()类似，不同的是在锁已经被占据时返回 EBUSY 而不是挂起等待。

Example：比较 pthread_mutex_trylock()与 pthread_mutex_lock()

```
#include <stdio.h>
#include <pthread.h>
pthread_mutex_t lock;
void* pthfunc(void *args)
{
    pthread_mutex_lock(&lock);        //先加一次锁
```

```
        pthread_mutex_lock(&lock);        //再用 lock 加锁，会挂起阻塞
        //pthread_mutex_trylock(&lock);   //用 trylock 加锁，则不会挂起阻塞
        printf("hello\n");
        sleep(1);
        pthread_exit(NULL);
}
main()
{
        pthread_t pthid = 0;
        pthread_mutex_init(&lock,NULL);
        pthread_create(&pthid,NULL,pthfunc,NULL);
        pthread_join(pthid,NULL);
        pthread_mutex_destroy(&lock);
}
```

4. 加锁注意事项

如果线程在加锁后解锁前被取消，锁将永远保持锁定状态，因此如果在关键区段内有取消点存在，则必须在退出回调函数 pthread_cleanup_push/pthread_cleanup_pop 中解锁。同时不应该在信号处理函数中使用互斥锁，否则容易造成死锁。

死锁是指多个进程因竞争资源而造成的一种僵局（互相等待），若无外力作用，这些进程都将无法向前推进。

死锁产生的原因

1. 系统资源的竞争

系统资源的竞争导致系统资源不足，以及资源分配不当，导致死锁。

2. 进程运行推进顺序不合适

进程在运行过程中，请求和释放资源的顺序不当，会导致死锁。

死锁的四个必要条件

互斥条件：一个资源每次只能被一个进程使用，即在一段时间内某 资源仅为一个进程所占有。此时若有其他进程请求该资源，则请求进程只能等待。

请求与保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源 已被其他进程占有，此时请求进程被阻塞，但对自己已获得的资源保持不放。

不可剥夺条件:进程所获得的资源在未使用完毕之前，不能被其他进程强行夺走，即只能 由获得该资源的进程自己来释放（只能是主动释放)。

循环等待条件: 若干进程间形成首尾相接循环等待资源的关系

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

死锁的预防

我们可以通过破坏死锁产生的 4 个必要条件来 预防死锁，由于资源互斥是资源使用的固有特性是无法改变的。

破坏“不可剥夺”条件：一个进程不能获得所需要的全部资源时便处于等待状态，等待期间他占有的资源将被隐式的释放重新加入到 系统的资源列表中，可以被其他的进程使用，而等待的进程只有重新获得自己原有的资源以及新申请的资源才可以重新启动，执行。

破坏”请求与保持条件“：第一种方法静态分配即每个进程在开始执行时就申请他所需要的全部资源。第二种是动态分配即每个进程在申请所需要的资源时他本身不占用系统资源。

破坏“循环等待”条件：采用资源有序分配其基本思想是将系统中的所有资源顺序编号，将紧缺的，稀少的采用较大的编号，在申请资源时必须按照编号的顺序进行，一个进程只有获得较小编号的进程才能申请较大编号的进程。

5. 互斥锁实例

Example：火车站售票（此处不加锁，则会出现卖出负数票的情况）

```
#include <stdio.h>
#include <pthread.h>
int ticketcount = 20;       //火车票，公共资源（全局）
void* salewinds1(void* args)        //售票口 1
{
    while(ticketcount > 0)          //如果有票,则卖票
    {
        printf("windows1 start sale ticket!the ticket is:%d\n",ticketcount);
        sleep(3);                   //卖一张票需要 3 秒的操作时间
        ticketcount --;             //出票
        printf("sale ticket finish!,the last ticket is:%d\n",ticketcount);
    }
}
void* salewinds2(void* args)        //售票口 2
{
    while(ticketcount > 0)          //如果有票,则卖票
    {
        printf("windows2 start sale ticket!the ticket is:%d\n",ticketcount);
        sleep(3);                   //卖一张票需要 3 秒的操作时间
        ticketcount --;             //出票
        printf("sale ticket finish!,the last ticket is:%d\n",ticketcount);
    }
}
int main()
{
    pthread_t pthid1 = 0;
    pthread_t pthid2 = 0;
    pthread_create(&pthid1,NULL,salewinds1,NULL);     //线程 1
    pthread_create(&pthid2,NULL,salewinds2,NULL);     //线程 2
    pthread_join(pthid1,NULL);
```

```
        pthread_join(pthid2,NULL);
        return 0;
    }
```

Example：加锁之后的火车售票

```
#include <stdio.h>
#include <pthread.h>
int ticketcount = 20;
pthread_mutex_t lock;
void* salewinds1(void* args)
{
    while(1)
    {
        pthread_mutex_lock(&lock);   //因为要访问全局的共享变量，所以就要加锁
        if(ticketcount > 0)   //如果有票
        {
            printf("windows1 start sale ticket!the ticket is:%d\n",ticketcount);
            sleep(3);                   //卖一张票需要 3 秒的操作时间
            ticketcount --;             //出票
            printf("sale ticket finish!,the last ticket is:%d\n",ticketcount);
        }
        else                    //如果没有票
        {
            pthread_mutex_unlock(&lock);     //解锁
            pthread_exit(NULL);              //退出线程
        }
        pthread_mutex_unlock(&lock);         //解锁
        sleep(1);       //要放到锁的外面，让另一个有时间锁
    }
}
void* salewinds2(void* args)
{
    while(1)
    {
        pthread_mutex_lock(&lock);   //因为要访问全局的共享变量，所以就要加锁
        if(ticketcount>0)   //如果有票
        {
            printf("windows2 start sale ticket!the ticket is:%d\n",ticketcount);
            sleep(3);                   //卖一张票需要 3 秒的操作时间
            ticketcount --;             //出票
            printf("sale ticket finish!,the last ticket is:%d\n",ticketcount);
        }
        else                    //如果没有票
        {
            pthread_mutex_unlock(&lock);     //解锁
            pthread_exit(NULL);              //退出线程
        }
```

```
            pthread_mutex_unlock(&lock);          //解锁
            sleep(1);       //要放到锁的外面，让另一个有时间锁
        }

    }
    int main()
    {
        pthread_t pthid1 = 0;
        pthread_t pthid2 = 0;
        pthread_mutex_init(&lock,NULL);    //初始化锁
        pthread_create(&pthid1,NULL,salewinds1,NULL);     //线程 1
        pthread_create(&pthid2,NULL,salewinds2,NULL);     //线程 2
        pthread_join(pthid1,NULL);
        pthread_join(pthid2,NULL);
        pthread_mutex_destroy(&lock);       //销毁锁
        return 0;
    }
```

总结：线程互斥 mutex：加锁步骤如下：

1. 定义一个全局的 pthread_mutex_t lock; 或者用
   pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER; //则 main 函数中不用 init
2. 在 main 中调用 pthread_mutex_init 函数进行初始化
3. 在子线程函数中调用 pthread_mutex_lock 加锁
4. 在子线程函数中调用 pthread_mutex_unlock 解锁
5. 最后在 main 中调用 pthread_mutex_destroy 函数进行销毁

4.2. 线程的同步

4.2.1 条件变量

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）。为了防止竞争，条件变量的使用总是和一个互斥锁结合在一起。

1、 创建和注销

条件变量和互斥锁一样，都有静态、动态两种创建方式：

静态方式使 PTHREAD_COND_INITIALIZER 常量，如下：

```
pthread_cond_t    cond = PTHREAD_COND_INITIALIZER;
```

动态方式调用 pthread_cond_init()函数，API 定义如下：

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
```

尽管 POSIX 标准中为条件变量定义了属性，但在 Linux Threads 中没有实现，因此 cond_attr 值通常为 NULL，且被忽略。

注销一个条件变量需要调用 pthread_cond_destroy()，只有在没有线程在该条件变量上等待的时候能注销这个条件变量，否则返回 EBUSY。因为 Linux 实现的条件变量没有分配什么资源，所以注销动作只包括检查是否有等待线程。API 定义如下：

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

2、 等待和激发

等待条件有两种方式：无条件等待 pthread_cond_wait()和计时等待 pthread_cond_timedwait():

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,

const struct timespec *abstime);

线程解开 mutex 指向的锁并被条件变量 cond 阻塞。其中计时等待方式表示经历 abstime 段时间后，即使条件变量不满足，阻塞也被解除。无论哪种等待方式，都必须和一个互斥锁配合，以防止多个线程同时请求 pthread_cond_wait()（或 pthread_cond_timedwait()，下同）的竞争条件（Race Condition）。mutex 互斥锁必须是普通锁（PTHREAD_MUTEX_TIMED_NP），且在调用 pthread_cond_wait()前必须由本线程加锁（pthread_mutex_lock()），而在更新条件等待队列以前，mutex 保持锁定状态，并在线程挂起进入等待前解锁。在条件满足从而离开 pthread_cond_wait()之前，mutex 将被重新加锁，以与进入 pthread_cond_wait()前的加锁动作对应。（也就是说在做 pthread_cond_wait 之前，往往要用 pthread_mutex_lock 进行加锁，而调用 pthread_cond_wait 函数会将锁解开，然后将线程挂起阻塞。直到条件被 pthread_cond_signal 激发，再将锁状态恢复为锁定状态，最后再用 pthread_mutex_unlock 进行解锁）。

激发条件有两种形式，pthread_cond_signal()激活一个等待该条件的线程，存在多个等待线程时按入队顺序激活其中一个；而 pthread_cond_broadcast()则激活所有等待线程

前时间 + 多长时间超时

超时是指 当

3、 其他

pthread_cond_wait()和 pthread_cond_timedwait()都被实现为取消点，也就是说如果 pthread_cond_wait()被取消，则退出阻塞，然后将锁状态恢复，则此时 mutex 是保持锁定状态的，而当前线程已经被取消掉，那么解锁的操作就会得不到执行，此时锁得不到释放，就会造成死锁，因而需要定义退出回调函数来为其解锁。

以下示例集中演示了互斥锁和条件变量的结合使用，以及取消对于条件等待动作的影响。在例子中，有两个线程被启动，并等待同一个条件变量，如果不使用退出回调函数（见范例中的注释部分），则 tid2 将在 pthread_mutex_lock()处永久等待。如果使用回调函数，则 tid2 的条件等待及主线程的条件激发都能正常工作。

实例：

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

pthread_mutex_t mutex;
pthread_cond_t cond;

void ThreadClean(void *arg)
{
    pthread_mutex_unlock(&mutex);
}

void * child1(void *arg)
{
```

```
    //pthread_cleanup_push(ThreadClean,NULL);    //1
    while(1){
        printf("thread 1 get running \n");
        printf("thread 1 pthread_mutex_lock returns %d\n", pthread_mutex_lock(&mutex));
        pthread_cond_wait(&cond,&mutex); //等待父进程发送信号
        printf("thread 1 condition applied\n");
        pthread_mutex_unlock(&mutex);
        sleep(5);
    }
    //pthread_cleanup_pop(0);      //2
    return 0;
}

void *child2(void *arg)
{
    while(1){
        sleep(3);       //3
        printf("thread 2 get running.\n");
        printf("thread 2 pthread_mutex_lock returns %d\n", pthread_mutex_lock(&mutex));
        pthread_cond_wait(&cond,&mutex);
        printf("thread 2 condition applied\n");
        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
}

int main(void)
{
    pthread_t tid1,tid2;
    printf("hello, condition variable test\n");
    pthread_mutex_init(&mutex,NULL);
    pthread_cond_init(&cond,NULL);
    pthread_create(&tid1,NULL,child1,NULL);
    pthread_create(&tid2,NULL,child2,NULL);
    while(1){    //父线程
        sleep(2);                    //4
        pthread_cancel(tid1);        //5
        sleep(2);                    //6
        pthread_cond_signal(&cond);
    }
    sleep(10);
    return 0;
}
```

不做注释 1，2 则导致 child1 中的 unlock 得不到执行，锁一直没有关闭，而 child2 中的锁不能执行 lock，则会一直在 pthread_mutex_lock()处永久等待。如果不做注释 5 的 pthread_cancel()动作，即使

没有那些 sleep()延时操作，child1 和 child2 都能正常工作。注释 3 和注释 4 的延迟使得 child1 有时间完成取消动作，从而使 child2 能在 child1 退出之后进入请求锁操作。如果没有注释 1 和注释 2 的回调函数定义，系统将挂起在 child2 请求锁的地方，因为 child1 没有释放锁；而如果同时也不做注释 3 和注释 4 的延时，child2 能在 child1 完成取消动作以前得到控制，从而顺利执行申请锁的操作，但却可能挂起在 pthread_cond_wait()中，因为其中也有申请 mutex 的操作。child1 函数给出的是标准的条件变量的使用方式：回调函数保护，等待条件前锁定，pthread_cond_wait()返回后解锁。

条件变量机制和互斥锁一样，不能用于信号处理中，在信号处理函数中调用 pthread_cond_signal()或者 pthread_cond_broadcast()很可能引起死锁。

Example：火车售票，利用条件变量，当火车票卖完的时候，再重新设置票数为 10；

```
#include<pthread.h>
#include<stdio.h>
int ticketcount = 10;
pthread_mutex_t lock;    //互斥锁
pthread_cond_t cond;     //条件变量
void* salewinds1(void* args)
{
    while(1)
    {
        pthread_mutex_lock(&lock);   //因为要访问全局的共享变量 ticketcount，所以就要加锁
        if(ticketcount > 0)   //如果有票
        {
            printf("windows1 start sale ticket!the ticket is:%d\n",ticketcount);
            ticketcount --;//则卖出一张票
            if(ticketcount == 0)
                pthread_cond_signal(&cond); //通知没有票了
            printf("sale ticket finish!,the last ticket is:%d\n",ticketcount);
        }
        else     //如果没有票了，就解锁退出
        {
            pthread_mutex_unlock(&lock);
            break;
        }
        pthread_mutex_unlock(&lock);
        sleep(1);         //要放到锁的外面
    }
}
void* salewinds2(void* args)
{
    while(1)
    {
        pthread_mutex_lock(&lock);
        if(ticketcount > 0)
        {
            printf("windows2 start sale ticket!the ticket is:%d\n",ticketcount);
            ticketcount --;
```

```
                if(ticketcount == 0)
                    pthread_cond_signal(&cond);   //发送信号
                printf("sale ticket finish!,the last ticket is:%d\n",ticketcount);
            }
            else
            {
                pthread_mutex_unlock(&lock);
                break;
            }
            pthread_mutex_unlock(&lock);
            sleep(1);
        }
}
void *setticket(void *args)   //重新设置票数
{
        pthread_mutex_lock(&lock);              //因为要访问全局变量 ticketcount，所以要加锁
        if(ticketcount > 0)
            pthread_cond_wait(&cond,&lock); //如果有票就解锁并阻塞，直到没有票就执行下面的
        ticketcount = 10;    //重新设置票数为 10
        pthread_mutex_unlock(&lock);   //解锁
        sleep(1);
        pthread_exit(NULL);
}
main()
{
        pthread_t pthid1,pthid2,pthid3;
        pthread_mutex_init(&lock,NULL);   //初始化锁
        pthread_cond_init(&cond,NULL);    //初始化条件变量
        pthread_create(&pthid1,NULL, salewinds1,NULL);   //创建线程
        pthread_create(&pthid2,NULL, salewinds2,NULL);
        pthread_create(&pthid3,NULL, setticket,NULL);
        pthread_join(pthid1,NULL);      //等待子线程执行完毕
        pthread_join(pthid2,NULL);
        pthread_join(pthid3,NULL);
        pthread_mutex_destroy(&lock);   //销毁锁
        pthread_cond_destroy(&cond);    //销毁条件变量
}
```

4.3. 生产者消费者问题

Example1、链表实现如下：

```
#include <pthread.h>
#include <unistd.h>
#include<stdio.h>
#include<malloc.h>
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
```

```
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
struct node {
        int n_number;
        struct node *n_next;
} *head = NULL;

static void cleanup_handler(void *arg)
{
        printf("Cleanup handler of second thread\n");
        free(arg);
        pthread_mutex_unlock(&mtx);
}

static void *thread_func(void *arg)//消费者
{
        struct node *p = NULL;
        pthread_cleanup_push(cleanup_handler, p);
        while (1)
        {
            If(
            pthread_mutex_lock(&mtx);
            while (head == NULL || (flag=0)){ pthread_cond_wait(&cond,&mtx);}
            p = head;
            head = head->n_next;
            printf("Got %d from front of queue\n", p->n_number);
            free(p);
            pthread_mutex_unlock(&mtx);
            else
        }
        pthread_exit(NULL);
        pthread_cleanup_pop(0);     //必须放在最后一行
}
int main(void)
{
        pthread_t tid;
        int i;
        struct node *p;
        pthread_create(&tid, NULL, thread_func, NULL);
        for (i = 0; i < 10; i++)//生产者
        {
            p = (struct node*)malloc(sizeof(struct node));
            p->n_number = i;

            pthread_mutex_lock(&mtx);//因为 head 是共享的，访问共享数据必须要加锁
            p->n_next = head;
            head = p;
```

```
                pthread_cond_signal(&cond);
                pthread_mutex_unlock(&mtx);
                sleep(1);
            }
            printf("thread 1 wanna end the line.So cancel thread 2.\n");
            pthread_cancel(tid);
            pthread_join(tid, NULL);
            printf("All done------exiting\n");
            return 0;
        }
```

Example2、队列实现如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#define BUFFER_SIZE 16          //表示一次最多可以不间断的生产 16 个产品
struct prodcons {
    int buffer[BUFFER_SIZE];    /* 数据 */
    pthread_mutex_t lock;       /* 加锁 */
    int readpos, writepos;      /* 读 pos   写位置 */
    pthread_cond_t notempty;    /* 不空，可以读 */
    pthread_cond_t notfull;     /* 不满，可以写 */
};
/* 初始化*/
void init(struct prodcons * b)
{
    pthread_mutex_init(&b->lock, NULL);       //初始化锁
    pthread_cond_init(&b->notempty, NULL);    //初始化条件变量
    pthread_cond_init(&b->notfull, NULL);     //初始化条件变量
    b->readpos = 0;                           //初始化读取位置从 0 开始
    b->writepos = 0;                          //初始化写入位置从 0 开始
}
/* 销毁操作 */
void destroy(struct prodcons *b)
{
    pthread_mutex_destroy(&b->lock);
    pthread_cond_destroy(&b->notempty);
    pthread_cond_destroy(&b->notfull);
}
void put(struct prodcons * b, int data)//生产者
{
    pthread_mutex_lock(&b->lock);
    while ((b->writepos + 1) % BUFFER_SIZE == b->readpos) {//判断是不是满了
        printf("wait for not full\n");
        pthread_cond_wait(&b->notfull, &b->lock); //此时为满，不能生产，等待不满的信号
```

```
        }

        //下面表示还没有满，可以进行生产
        b->buffer[b->writepos] = data;
        b->writepos++;       //写入点向后移一位
        if (b->writepos >= BUFFER_SIZE)    b->writepos = 0; //如果到达最后，就再转到开头
        pthread_cond_signal(&b->notempty);   //此时有东西可以消费，发送非空的信号
        pthread_mutex_unlock(&b->lock);
}
int get(struct prodcons * b)//消费者
{
        pthread_mutex_lock(&b->lock);
        while (b->writepos == b->readpos) {//判断是不是空
                printf("wait for not empty\n");
                pthread_cond_wait(&b->notempty, &b->lock); //此时为空，不能消费，等待非空信号
        }
        //下面表示还不为空，可以进行消费
        int data = b->buffer[b->readpos];
        b->readpos++;        //读取点向后移一位
        if (b->readpos >= BUFFER_SIZE)    b->readpos = 0; //如果到达最后，就再转到开头
        pthread_cond_signal(&b->notfull);       //此时可以进行生产，发送不满的信号
        pthread_mutex_unlock(&b->lock);
        return data;
}
/*-----------------------------------------------------*/
#define OVER (-1)        //定义结束标志
struct prodcons buffer;    //定义全局变量
/*-----------------------------------------------------*/
void * producer(void * data)
{
        int n = 0;
        for (; n < 50; n++) {
                printf(" put-->%d\n", n);
                put(&buffer, n);
        }
        put(&buffer, OVER);
        printf("producer stopped!\n");
        pthread_exit(NULL);
}
/*-----------------------------------------------------*/
void * consumer(void * data)
{
        while (1) {
                int d = get(&buffer);
                if (d == OVER ) break;
                printf(" %d-->get\n", d);
```

```
        }

        printf("consumer stopped!\n");
        pthread_exit(NULL);
}
/*--------------------------------------------------------*/
int main(void)
{
        pthread_t th_a, th_b;
        init(&buffer);
        pthread_create(&th_a, NULL, producer, 0);
        pthread_create(&th_b, NULL, consumer, 0);
        pthread_join(th_a, NULL);
        pthread_join(th_b, NULL);
        destroy(&buffer);
        return 0;
}
```

5. 线程安全和线程属性

5.1. 线程安全

先来看一段代码：

```
void* threadFunc(void* p1)
{
        time_t now;
        time(&now);
        char *p=ctime(&now);
        printf("I am child thread p=%s\n",p);
        sleep(3);
        printf("I am child thread p=%s\n",p);
}
int main(int argc,char* argv[])
{
        pthread_t pthid;
        pthread_create(&pthid,NULL,threadFunc,NULL);
        sleep(1);
        time_t now;
        time(&now);
        char *p=ctime(&now);
        printf("I am main thread p=%s\n",p);
        pthread_join(pthid,NULL);
        return 0;
}
```

线程安全：如果一个函数能够安全的**同时被多个线程**调用而得到正确的结果，那么，我们说这个函数是线程安全的。简单来说线程安全就是多个线程并发同一段代码时，不会出现不同的结果，我们就可以说该线程是安全的；

线程安全产生的原因：大多是因为对全局变量和静态变量的操作。

可重入函数：访问时有可能因为重入而造成错乱,像这样的函数称为 不可重入函数,反之,如果一个函数只访问自己的局部变量或参数,则称为可重入函数。简而言之，可重入函数，描述的是函数被多次调用但是结果具有可再现性。

可重入，并不一定要是多线程的。可重入只关注一个结果可再现性。在 APUE 中，可函数可重入的概念最先是在讲 signal 的 handler 的时候提出的。此时进程(线程)正在执行函数 fun()，在函数 fun()还未执行完的时候，突然进程接收到一个信号 sig，此时，需要暂停执行 fun(),要转而执行 sig 信号的处理函数 sig_handler()，那么，如果在 sig_handler()中，也恰好调用了函数 fun().信号的处理是以软终端的形式进行的，那么，当 sig_handler()执行完返回之后，CPU 会继续从 fun()被打断的地方往下执行。这里讲的比较特殊，最好的情况是，进程中调用了 fun()，函数，信号处理函数 sig_handle()中也调用了 fun()。如果 fun()函数是可重入的，那么，多次调用 fun()函数就具有可再现性。从而，两次调用 fun()的结果是正确的预期结果。非可重入函数，则恰好相反。

(1)可重入概念只和函数访问的变量类型有关，和是否使用锁没有关系。
(2)线程安全，描述的是函数能同时被多个线程安全的调用，并不要求调用函数的结果具有可再现性。也就是说，多个线程同时调用该函数，允许出现互相影响的情况，这种情况的出现需要某些机制比如互斥锁来支持，使之安全。
(3)可重入函数是线程安全函数的一种，其特点在于它们被多个线程调用时，不会引用任何共享数据。
(4)线程安全是在多个线程情况下引发的，而可重入函数可以在只有一个线程的情况下来说。
(5)线程安全不一定是可重入的，而可重入函数则一定是线程安全的。
(6)如果一个函数中有全局变量，那么这个函数既不是线程安全也不是可重入的。
(7)如果将对临界资源的访问加上锁，则这个函数是线程安全的，但如果这个重入函数若锁还未释放则会产生死锁，因此是不可重入的。
(8)线程安全函数能够使不同的线程访问同一块地址空间，而可重入函数要求不同的执行流对数据的操作互不影响使结果是相同的。

5.2. 线程的属性

pthread_create 的第二个参数 attr 是一个结构体指针，结构中的元素分别指定新线程的运行属性,各成员属性为：

__detachstate 表示新线程是否与进程中其他线程脱离同步，如果置位则新线程不能用 pthread_join()来同步，且在退出时自行释放所占用的资源。缺省为 PTHREAD_CREATE_JOINABLE 状态。这个属性也可以在线程创建并运行以后用 pthread_detach()来设置，而一旦设置为 PTHREAD_CREATE_DETACHED 状态（不论是创建时设置还是运行时设置）则不能再恢复到 PTHREAD_CREATE_JOINABLE 状态。

__schedpolicy，表示新线程的调度策略，主要包括 SCHED_OTHER（正常、非实时）、SCHED_RR（实时、轮转法）和 SCHED_FIFO（实时、先入先出）三种，缺省为 SCHED_OTHER，后两种调度策

略仅对超级用户有效。运行时可以用过 pthread_setschedparam()来改变。

__schedparam，一个 sched_param 结构，目前仅有一个 sched_priority 整型变量表示线程的运行优先级。这个参数仅当调度策略为实时（即 SCHED_RR 或 SCHED_FIFO）时才有效，并可以在运行时通过 pthread_setschedparam()函数来改变，缺省为 0。

__inheritsched，有两种值可供选择：PTHREAD_EXPLICIT_SCHED 和 PTHREAD_INHERIT_SCHED，前者表示新线程使用显式指定调度策略和调度参数（即 attr 中的值），而后者表示继承调用者线程的值。缺省为 PTHREAD_EXPLICIT_SCHED。

__scope，表示线程间竞争 CPU 的范围，也就是说线程优先级的有效范围。POSIX 的标准中定义了两个值：PTHREAD_SCOPE_SYSTEM 和 PTHREAD_SCOPE_PROCESS，前者表示与系统中所有线程一起竞争 CPU 时间，后者表示仅与同进程中的线程竞争 CPU。目前 Linux 仅实现了 PTHREAD_SCOPE_SYSTEM 一值。

属性设置是由一些函数来完成的，通常调用 pthread_attr_init 函数进行初始化。设置绑定属性的函数为 pthread_attr_setscope,设置分离属性的函数是 pthread_attr_setdetachstate,设置线程优先级的相关函数 pthread_attr_getscehdparam（获取线程优先级）和 pthread_attr_setschedparam（设置线程优先级）。再设置完成属性后，调用 pthread_creat 函数创建线程。

- 线程属性初始化：

  int pthread_attr_init (pthread_attr_t *attr);

attr：传出参数，表示线程属性，后面的线程属性设置函数都会用到。

返回值：成功 0，错误-1。

- 设置绑定属性：

  pthread_attr_setscope(pthread_attr_t *attr, init scope);

attr：线程属性

scope：PTHREAD_SCOPE_SYSTEM(绑定)　　PTHREAD_SCOPE_PRCESS(非绑定)

返回值：成功 0，错误-1。

- 设置分离属性:

  pthread_attr_setdetachstate(pthread_attr_t　*attr, init detachstate);

attr：线程属性

detachstate ：PTHREAD_CREAT_DETACHED(分离)　PTHREAD_CREAT_JOINABLE(非分离)

返回值：成功 0，错误-1。

- 获取线程优先级：

  int pthread_attr_getschedparam(pthread_attr_t *attr, struct sched_param *param);

attr：线程属性

param：线程优先级

返回值：成功 0，错误-1。

- 设置线程优先级：

  int pthread_attr_setschedparam(pthread_attr_t *attr,　struct sched_param *param);

attr：线程属性

param：线程优先级

返回值：成功 0，错误-1。

实例：

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

```
void *thread_function(void *arg);

char message[] = "Hello World";
int thread_finished = 0;

int main()
{
    int res = 0;
    pthread_t a_thread;
    void *thread_result;
    pthread_attr_t    thread_attr;   //定义属性
    struct sched_param scheduling_value;
    res = pthread_attr_init(&thread_attr);    //属性初始化
    if (res != 0)
    {
        perror("Attribute creation failed");
        exit(EXIT_FAILURE);   // EXIT_FAILURE    -1
    }

    //设置调度策略
    res = pthread_attr_setschedpolicy(&thread_attr, SCHED_OTHER);
    if (res != 0)
    {
        perror("Setting schedpolicy failed");
        exit(EXIT_FAILURE);
    }

    //设置脱离状态
    res = pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_DETACHED);

    //创建线程
    res = pthread_create(&a_thread, &thread_attr, thread_function, (void *)message);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }



    //获取最大优先级别
    int    max_priority = sched_get_priority_max(SCHED_OTHER);
    //获取最小优先级
    int    min_priority = sched_get_priority_min(SCHED_OTHER);

    //重新设置优先级别
    scheduling_value.sched_priority = min_priority + 5;
```

```
    //设置优先级别
    res = pthread_attr_setschedparam(&thread_attr, &scheduling_value);

    pthread_attr_destroy(&thread_attr);

    while(!thread_finished)
    {
        printf("Waiting for thread to say it's finished...\n");
        sleep(1);
    }
    printf("Other thread finished, bye!\n");
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg)
{
    printf("thread_function is running. Argument was %s\n", (char *)arg);
    sleep(4);
    printf("Second thread setting finished flag, and exiting now\n");
    thread_finished = 1;
    pthread_exit(NULL);
}
```

额外作业：(快速文件 copy)

大家都使用过迅雷，迅雷有一个技术是多点下载，也就是不同的服务器为你提供种子，一部电影，可以同时从 10 个地方同时下载，这样下载的速度就大大提升，因为多点可以保证当一个点或者几个点的网络不稳定，或者对应提供给你下载的机器关机时，其他的点仍可以继续进行文件下载。

当然我们在多线程之后才会讲解网络编程，现在是你在本地有一个 1G 多的电影或者大文件，你需要同时启动 10 个线程，将文件切成 10 段，每个线程都有一个起始点，为了模拟下载模式，每个线程每次从文件中读取 64 个字节，然后写入一个新文件，当所有的线程均将自己的内容写入完毕后，主线程 join 所有子线程，然后打印文件 copy 成功。

提示：1.使用 mmap，mmap 不能直接写空文件，需要用 ftruncate 先建立一个和待复制文件一样大的全 0 文件，然后再映射

2.如果只使用 read，write，记得如果你只打开一个 fd，一个 fd 是只能记录一个偏移的，切记。