一、互斥锁--mutex 函数

Linux 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）。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，以实现资源的独占。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才能够对共享资源进行操作。

1、互斥锁的创建

有两种方法创建互斥锁，静态方式和动态方式。

2、静态创建

利用宏定义：

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_t 是一个结构，而 PTHREAD_MUTEX_INITIALIZER 则是一个宏常量代表创建快速互斥锁。

3、动态创建

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)
```

参数 1：创建的互斥锁，上锁解锁操作的就是次参数

参数 2：创建锁的类型，通常指定为 NULL，也有一下几种选择

PTHREAD_MUTEX_INITIALIZER：创建快速互斥锁

PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP：创建递归互斥锁

PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP： 创建检错互斥锁

其中，互斥锁可以分为快速互斥锁、递归互斥锁和检错互斥锁。这 3 种锁的区别主要在于其他未占有互斥锁的线程在希望得到互斥锁时是否需要阻塞等待。快速锁是指调用线程会阻塞直至拥有互斥锁的线程解锁为止。递归互斥锁能够成功地返回，并且增加调用线程在互斥上加锁的次数，而检错互斥锁则为快速互斥锁的非阻塞版本，它会立即返回并返回一个错误信息。默认属性为快速互斥锁。

4、注销锁

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（成功返回 0 出错返回错误码）没有其他动作。

5、加锁和解锁

加锁 int pthread_mutex_lock(pthread_mutex_t *mutex)

解锁 int pthread_mutex_unlock(pthread_mutex_t *mutex)

注意：如果线程在加锁后解锁前被取消，锁将永远保持锁定状态。因此如果在前边区段内有加锁存在，则必须在退出回调函数 pthread_cleanup_push/pthread_cleanup_pop 中解锁。

6、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，及时解锁
5. 最后在 main 中调用 pthread_mutex_destroy 函数进行销毁

我们来模拟没有锁机制的操作：

```
#include <stdio.h>
#include <pthread.h>
int ticketcount = 20;   //火车票，公共资源（全局）
```

```
void* salewinds1(void* args)
{
    while(ticketcount > 0)
    {
        printf("窗口 1 正在售票，当前有%d 张票\n",ticketcount);
        sleep(3);
        ticketcount --;
        printf("窗口 1 售票完，当前剩余%d 张票\n",ticketcount);
    }
}
void* salewinds2(void* args)
{
    while(ticketcount > 0)
    {
        printf("窗口 2 正在售票，当前有%d 张票\n",ticketcount);
        sleep(3);
        ticketcount --;
        printf("窗口 2 售票完，当前剩余%d 张票\n",ticketcount);
    }
}
int main()
{
    pthread_t pthid1 = 0;
    pthread_t pthid2 = 0;

    //创建窗口 1
    pthread_create(&pthid1,NULL,salewinds1,NULL);

    //创建窗口 2
    pthread_create(&pthid2,NULL,salewinds2,NULL);

    //等待子线程退出
    pthread_join(pthid1,NULL);
    pthread_join(pthid2,NULL);
    return 0;
}
```

这样的程序运行的结果为：

```
窗口2正在售票，当前有2张票
窗口1售票完，当前剩余1张票
窗口1正在售票，当前有1张票
窗口2售票完，当前剩余0张票
窗口1售票完，当前剩余-1张票
[root@localhost text]#
```

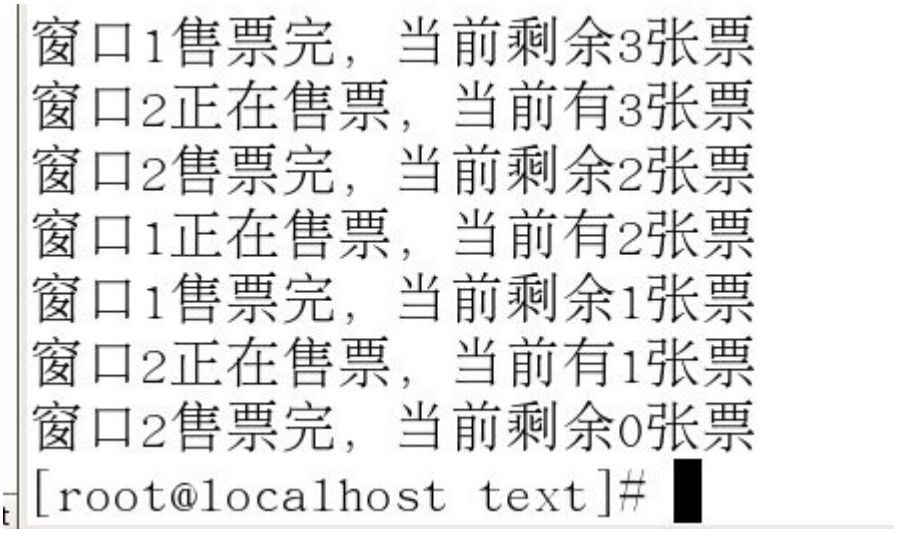
此时发现有负的出现，这个在实际运用中肯定是不允许的，这就是线程操作不加保护同步机制的后果，

所以我们线程操作一定要加上保护和同步资源的机制。加锁程序如下。

```
#include <stdio.h>
#include <pthread.h>
int ticketcount = 20;
pthread_mutex_t lock;
void* salewinds1(void* args)
{
    while(1)
    {
        //因为要访问全局的共享变量，所以就要加锁
        pthread_mutex_lock(&lock);
        if(ticketcount > 0)  //如果有票
        {
            printf("窗口 1 正在售票，当前有%d 张票\n",ticketcount);
            sleep(3);
            ticketcount --;
            printf("窗口 1 售票完，当前剩余%d 张票\n",ticketcount);
        }
        else                //如果没票
        {
            pthread_mutex_unlock(&lock); //没票就解锁
            pthread_exit(NULL);          //退出子线程
        }
        pthread_mutex_unlock(&lock);     //如果有票，卖完票要解锁
        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);   //加锁
        if(ticketcount>0)            //如果有票
        {
            printf("窗口 2 正在售票，当前有%d 张票\n",ticketcount);
            sleep(3);
            ticketcount --;
            printf("窗口 2 售票完，当前剩余%d 张票\n",ticketcount);
        }
        else                //如果没票
        {
            pthread_mutex_unlock(&lock); //没票就解锁
            pthread_exit(NULL);          //退出子线程
        }
        pthread_mutex_unlock(&lock);     //如果有票，卖完票要解锁
        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;

    //初始化锁
    pthread_mutex_init(&lock,NULL);

    //创建窗口 1
    pthread_create(&pthid1,NULL,salewinds1,NULL);

    //创建窗口 2
    pthread_create(&pthid2,NULL,salewinds2,NULL);

    //等待子线程退出
    pthread_join(pthid1,NULL);
    pthread_join(pthid2,NULL);

    //销毁锁
    pthread_mutex_destroy(&lock);
    return 0;
}
```

程序运行结果如下：



此时我们发现就没有负的存在了。

二、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

静态方式：pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

动态方式：int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);

第二个参数是条件变量的属性，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才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

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

3、激活

pthread_cond_signal()激活一个等待该条件的线程

int pthread_cond_signal(pthread_cond_t *cond);

示例：

示例 1：

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
pthread_mutex_t mutex;//定义了一个互斥锁变量
pthread_cond_t cond;//定义了一个条件变量
void * child1(void *arg)
{
printf("线程 1 开始运行！ \n");
printf("线程 1 上锁了 : %d\n", pthread_mutex_lock(&mutex));
printf("线程 1 开始等待条件被激活\n");
pthread_cond_wait(&cond,&mutex); //等待父线程发送信号
printf("条件满足后，停止阻塞！ \n");
pthread_mutex_unlock(&mutex);
return 0;
}
int main(void)
{
pthread_t tid1;
printf("测试条件变量！ \n");
pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond,NULL);
pthread_create(&tid1,NULL,child1,NULL);
```

```
sleep(5); //6
pthread_cond_signal(&cond);//让条件满足，即激活条件
sleep(3);
pthread_cond_destroy(&cond);//注销条件变量
pthread_mutex_destroy(&mutex);//注销互斥锁变量
return 0;
}
```

测试结果：

```
测试条件变量！
线程 1 开始运行！
线程 1 上锁了 ： 0
线程 1 开始等待条件被激活
条件满足后，停止阻塞！
[root@localhost text]#
```

三、信号量

线程当中也有自己保护资源的信号量操作机制，他和进程当中的信号量的一样的，只是相关 API 不一样。

1、初始化一个信号量

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

第一个参数：代表信号量的数据类型其实就是一个长整型数

第二个参数：是一个非负的数，0 代表用于线程间，1 代表用于进程间

第三个参数：代表信号量的初始值

2、p 操作也就是信号量减一的操作

```
int sem_wait(sem_t *sem);
```

若减一的时候信号量为 0，则程序就会阻塞，等待信号量为正才可以执行

3、v 操作也就是信号量加一的操作

```
int sem_post(sem_t *sem);
```

将信号量的值加一。

通过信号量可以对共享资源进行保护。把信号量的值初始化成共享资源的数量，当要使用共享资源时，可以使用 wait 操作进行申请，只有当有剩余资源的时候才能够申请成功，否则要进行等待。而当使用完资源时，需要使用 post 操作进行释放，使资源的计数值加一，如果释放时有人在等待资源，则将其唤醒。

以上保护同步资源的操作一定要记得，所有进程或者线程书写的时候都要遵循次规则，如果有进程或者线程不遵守此规则就没有意义了。

互斥锁：忙等待
信号量：睡眠等待

硬件中断：while、延时