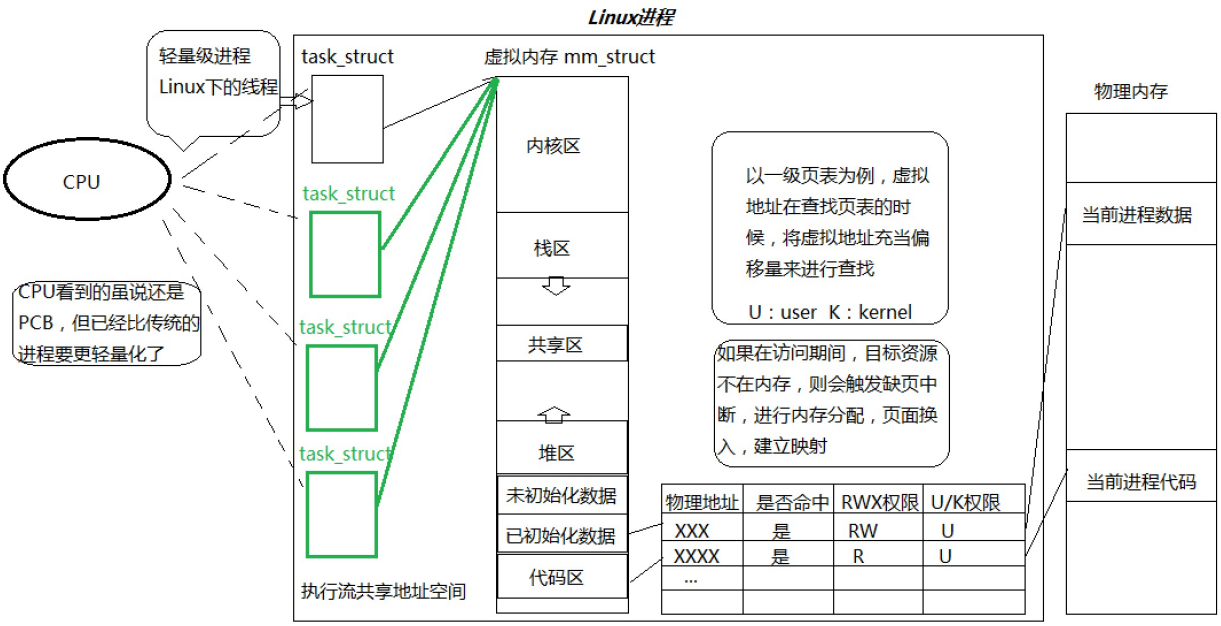
###本节重点：

> 1. 了解线程概念，理解线程与进程区别与联系。
> 2. 学会线程控制，线程创建，线程终止，线程等待。
> 3. 了解线程分离与线程安全概念。
> 4. 学会线程同步。
> 5. 学会使用互斥量，条件变量，posix信号量，以及读写锁。
> 6. 理解基于读写锁的读者写者问题。

##1. Linux线程概念

###什么是线程

> - 在一个程序里的一个执行路线就叫做线程（thread）。更准确的定义是：线程是“一个进程内部的控制序列”
> - 一切进程至少都有一个执行线程
> - 线程在进程内部运行，本质是在进程地址空间内运行
> - 在Linux系统中，在CPU眼中，看到的PCB都要比传统的进程更加轻量化
> - 透过进程虚拟地址空间，可以看到进程的大部分资源，将进程资源合理分配给每个执行流，就形成了线程执行流



线程的优点

> - 创建一个新线程的代价要比创建一个新进程小得多
> - 与进程之间的切换相比，线程之间的切换需要操作系统做的工作要少很多
> - 线程占用的资源要比进程少很多
> - 能充分利用多处理器的可并行数量
> - 在等待慢速I/O操作结束的同时，程序可执行其他的计算任务
> - 计算密集型应用，为了能在多处理器系统上运行，将计算分解到多个线程中实现
> - I/O密集型应用，为了提高性能，将I/O操作重叠。线程可以同时等待不同的I/O操作。

线程的缺点

- 性能损失
  - 一个很少被外部事件阻塞的计算密集型线程往往无法与共它线程共享同一个处理器。如果计算密集型线程的数量比可用的处理器多，那么可能会有较大的性能损失，这里的性能损失指的是增加了额外的同步和调度开销，而可用的资源不变。
- 健壮性降低
  - 编写多线程需要更全面更深入的考虑，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，换句话说线程之间是缺乏保护的。
- 缺乏访问控制
  - 进程是访问控制的基本粒度，在一个线程中调用某些OS函数会对整个进程造成影响。
- 编程难度提高
  - 编写与调试一个多线程程序比单线程程序困难得多

线程异常

- 单个线程如果出现除零，野指针问题导致线程崩溃，进程也会随着崩溃
- 线程是进程的执行分支，线程出异常，就类似进程出异常，进而触发信号机制，终止进程，进程终止，该进程内的所有线程也就随即退出

线程用途

- 合理的使用多线程，能提高CPU密集型程序的执行效率
- 合理的使用多线程，能提高IO密集型程序的用户体验（如生活中我们一边写代码一边下载开发工具，就是多线程运行的一种表现）

##2. Linux进程VS线程

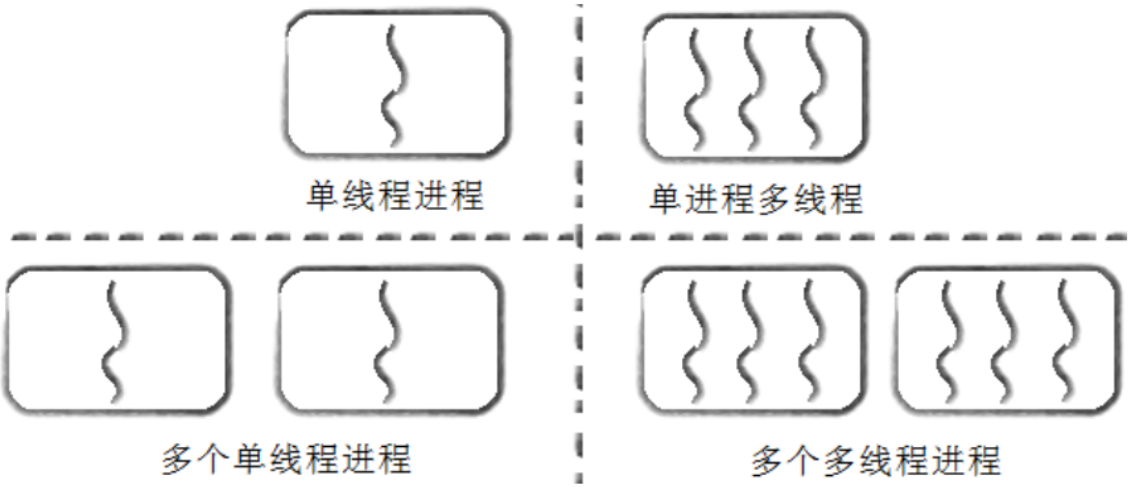
###进程和线程

- 进程是资源分配的基本单位
- 线程是调度的基本单位
- 线程共享进程数据，但也拥有自己的一部分数据:
  - 线程ID
  - 一组寄存器
  - 栈
  - errno
  - 信号屏蔽字
  - 调度优先级

###进程的多个线程共享 同一地址空间,因此Text Segment、Data Segment都是共享的,如果定义一个函数,在各线程中都可以调用,如果定义一个全局变量,在各线程中都可以访问到,除此之外,各线程还共享以下进程资源和环境:

- 文件描述符表
- 每种信号的处理方式(SIG_ IGN、SIG_ DFL或者自定义的信号处理函数)
- 当前工作目录
- 用户id和组id

进程和线程的关系如下图:



关于进程线程的问题

> - 如何看待之前学习的单进程？具有一个线程执行流的进程

##3. Linux线程控制

###POSIX线程库

> - 与线程有关的函数构成了一个完整的系列，绝大多数函数的名字都是以“pthread_”打头的
> - 要使用这些函数库，要通过引入头文<pthread.h>
> - 链接这些线程函数库时要使用编译器命令的“-lpthread”选项

###创建线程

```
功能：创建一个新的线程
原型
    int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *
(*start_routine)(void*), void *arg);
参数
    thread:返回线程ID
    attr:设置线程的属性，attr为NULL表示使用默认属性
    start_routine:是个函数地址，线程启动后要执行的函数
    arg:传给线程启动函数的参数
返回值：成功返回0；失败返回错误码
```

错误检查:

> - 传统的一些函数是，成功返回0，失败返回-1，并且对全局变量errno赋值以指示错误。
> - pthreads函数出错时不会设置全局变量errno（而大部分其他POSIX函数会这样做）。而是将错误代码通过返回值返回
> - pthreads同样也提供了线程内的errno变量，以支持其它使用errno的代码。对于pthreads函数的错误，建议通过返回值业判定，因为读取返回值要比读取线程内的errno变量的开销更小

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

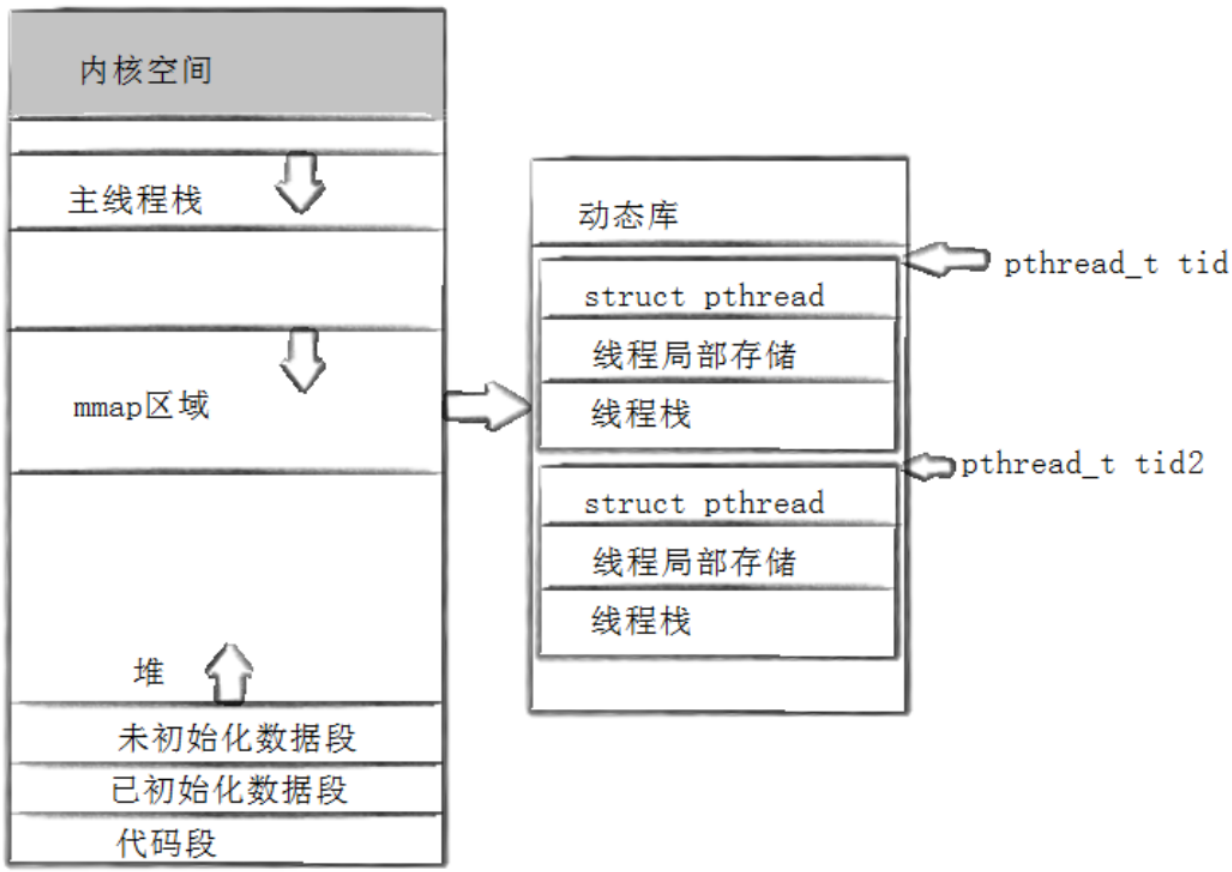
```c
#include <string.h>
#include <pthread.h>

void *rout(void *arg) {
    int i;
    for( ; ; ) {
        printf("I'am thread 1\n");
        sleep(1);
   }
}

int main( void )
{
    pthread_t tid;
    int ret;
    if ( (ret=pthread_create(&tid, NULL, rout, NULL)) != 0 ) {
        fprintf(stderr, "pthread_create : %s\n", strerror(ret));
        exit(EXIT_FAILURE);
    }

    int i;
    for(; ; ) {
        printf("I'am main thread\n");
        sleep(1);
    }
}
```

###线程ID及进程地址空间布局

> - pthread_ create函数会产生一个线程ID，存放在第一个参数指向的地址中。该线程ID和前面说的线程ID不是一回事。
> - 前面讲的线程ID属于进程调度的范畴。因为线程是轻量级进程，是操作系统调度器的最小单位，所以需要一个数值来唯一表示该线程。
> - pthread_ create函数第一个参数指向一个虚拟内存单元，该内存单元的地址即为新创建线程的线程ID，属于NPTL线程库的范畴。线程库的后续操作，就是根据该线程ID来操作线程的。
> - 线程库NPTL提供了pthread_ self函数，可以获得线程自身的ID：

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

`pthread_t`到底是什么类型呢？取决于实现。对于Linux目前实现的NPTL实现而言，pthread_t类型的线程ID，本质就是一个进程地址空间上的一个地址。



###线程终止

如果需要只终止某个线程而不终止整个进程,可以有三种方法:

> 1. 从线程函数return。这种方法对主线程不适用,从main函数return相当于调用exit。
> 2. 线程可以调用pthread_ exit终止自己。
> 3. 一个线程可以调用pthread_ cancel终止同一进程中的另一个线程。

pthread_exit函数

```
功能：线程终止
原型
    void pthread_exit(void *value_ptr);
参数
    value_ptr:value_ptr不要指向一个局部变量。
返回值：无返回值，跟进程一样，线程结束的时候无法返回到它的调用者（自身）
```

需要注意,pthread_exit或者return返回的指针所指向的内存单元必须是全局的或者是用malloc分配的,不能在线程函数的栈上分配,因为当其它线程得到这个返回指针时线程函数已经退出了。

pthread_cancel函数

```
功能：取消一个执行中的线程
原型
    int pthread_cancel(pthread_t thread);
参数
    thread:线程ID
返回值：成功返回0；失败返回错误码
```

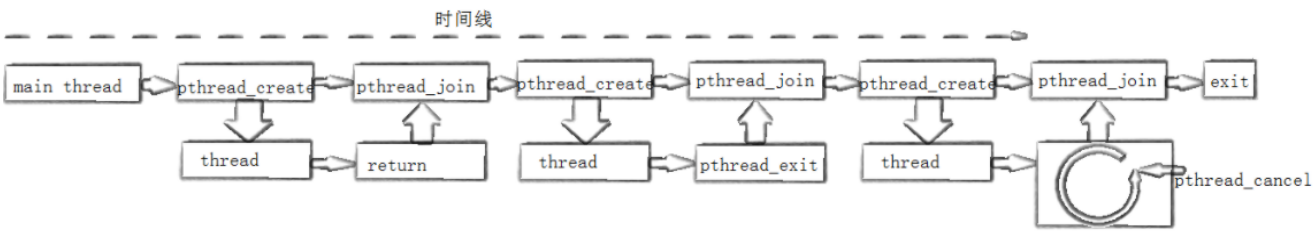
###线程等待 为什么需要线程等待?

> - 已经退出的线程，其空间没有被释放，仍然在进程的地址空间内。
> - 创建新的线程不会复用刚才退出线程的地址空间。

```
功能：等待线程结束
原型
    int pthread_join(pthread_t thread, void **value_ptr);
参数
    thread:线程ID
    value_ptr:它指向一个指针，后者指向线程的返回值
返回值：成功返回0；失败返回错误码
```

调用该函数的线程将挂起等待,直到id为thread的线程终止。thread线程以不同的方法终止,通过pthread_join得到的终止状态是不同的，总结如下:

> 1. 如果thread线程通过return返回,value_ ptr所指向的单元里存放的是thread线程函数的返回值。
> 2. 如果thread线程被别的线程调用pthread_ cancel异常终掉,value_ ptr所指向的单元里存放的是常数PTHREAD_ CANCELED。
> 3. 如果thread线程是自己调用pthread_exit终止的,value_ptr所指向的单元存放的是传给pthread_exit的参数。
> 4. 如果对thread线程的终止状态不感兴趣,可以传NULL给value_ ptr参数。




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

void *thread1( void *arg )
{
    printf("thread 1 returning ... \n");
    int *p = (int*)malloc(sizeof(int));
    *p = 1;
    return (void*)p;
}
```

```c
void *thread2( void *arg )
{
    printf("thread 2 exiting ...\n");
    int *p = (int*)malloc(sizeof(int));
    *p = 2;
    pthread_exit((void*)p);
}

void *thread3( void *arg )
{
    while ( 1 ){ //
        printf("thread 3 is running ...\n");
        sleep(1);
    }
    return NULL;
}

int main( void )
{
    pthread_t tid;
    void *ret;

    // thread 1 return
    pthread_create(&tid, NULL, thread1, NULL);
    pthread_join(tid, &ret);
    printf("thread return, thread id %X, return code:%d\n", tid, *(int*)ret);
    free(ret);

    // thread 2 exit
    pthread_create(&tid, NULL, thread2, NULL);
    pthread_join(tid, &ret);
    printf("thread return, thread id %X, return code:%d\n", tid, *(int*)ret);
    free(ret);

    // thread 3 cancel by other
    pthread_create(&tid, NULL, thread3, NULL);
    sleep(3);
    pthread_cancel(tid);
    pthread_join(tid, &ret);
    if ( ret ==  PTHREAD_CANCELED )
        printf("thread return, thread id %X, return code:PTHREAD_CANCELED\n", tid);
    else
        printf("thread return, thread id %X, return code:NULL\n", tid);
}
```

运行结果:

```
    [root@localhost linux]# ./a.out
    thread 1 returning ...
    thread return, thread id 5AA79700, return code:1
    thread 2 exiting ...
    thread return, thread id 5AA79700, return code:2
    thread 3 is running ...
    thread 3 is running ...
```

```
 thread 3 is running ...
 thread return, thread id 5AA79700, return code:PTHREAD_CANCELED
```

##4. 分离线程

> - 默认情况下，新创建的线程是joinable的，线程退出后，需要对其进行pthread_join操作，否则无法释放资源，从而造成系统泄漏。
> - 如果不关心线程的返回值，join是一种负担，这个时候，我们可以告诉系统，当线程退出时，自动释放线程资源。

```
int pthread_detach(pthread_t thread);
```

可以是线程组内其他线程对目标线程进行分离，也可以是线程自己分离:

```
pthread_detach(pthread_self());
```

joinable和分离是冲突的，一个线程不能既是joinable又是分离的。

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

void *thread_run( void * arg )
{
    pthread_detach(pthread_self());
    printf("%s\n", (char*)arg);
    return NULL;
}

int main( void )
{
    pthread_t tid;
    if ( pthread_create(&tid, NULL, thread_run, "thread1 run...") != 0 ) {
        printf("create thread error\n");
        return 1;
    }

    int ret = 0;
    sleep(1);//很重要，要让线程先分离，再等待

    if ( pthread_join(tid, NULL ) == 0 ) {
        printf("pthread wait success\n");
        ret = 0;
    } else {
        printf("pthread wait failed\n");
        ret = 1;
    }
    return ret;
}
```

##5. Linux线程互斥

###进程线程间的互斥相关背景概念

> - 临界资源：多线程执行流共享的资源就叫做临界资源
> - 临界区：每个线程内部，访问临界资源的代码，就叫做临界区
> - 互斥：任何时刻，互斥保证有且只有一个执行流进入临界区，访问临界资源，通常对临界资源起保护作用
> - 原子性（后面讨论如何实现）：不会被任何调度机制打断的操作，该操作只有两态，要么完成，要么未完成

###互斥量mutex

> - 大部分情况，线程使用的数据都是局部变量，变量的地址空间在线程栈空间内，这种情况，变量归属单个线程，其他线程无法获得这种变量。
> - 但有时候，很多变量都需要在线程间共享，这样的变量称为共享变量，可以通过数据的共享，完成线程之间的交互。
> - 多个线程并发的操作共享变量，会带来一些问题。

```
// 操作共享变量会有问题的售票系统代码
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>

int ticket = 100;

void *route(void *arg)
{
    char *id = (char*)arg;
    while ( 1 ) {
        if ( ticket > 0 ) {
            usleep(1000);
            printf("%s sells ticket:%d\n", id, ticket);
            ticket--;
        } else {
            break;
        }
    }
}

int main( void )
{
    pthread_t t1, t2, t3, t4;

    pthread_create(&t1, NULL, route, "thread 1");
    pthread_create(&t2, NULL, route, "thread 2");
    pthread_create(&t3, NULL, route, "thread 3");
    pthread_create(&t4, NULL, route, "thread 4");

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
```

```
    pthread_join(t3, NULL);
    pthread_join(t4, NULL);
}

一次执行结果:
    thread 4 sells ticket:100
    ...
    thread 4 sells ticket:1
    thread 2 sells ticket:0
    thread 1 sells ticket:-1
    thread 3 sells ticket:-2
```

为什么可能无法获得争取结果?

> - `if` 语句判断条件为真以后，代码可以并发的切换到其他线程
> - `usleep` 这个模拟漫长业务的过程，在这个漫长的业务过程中，可能有很多个线程会进入该代码段
> - `--ticket` 操作本身就不是一个原子操作

```
取出ticket--部分的汇编代码
objdump -d a.out > test.objdump
152   40064b:   8b 05 e3 04 20 00       mov    0x2004e3(%rip),%eax        # 600b34 <ticket>
153   400651:   83 e8 01                sub    $0x1,%eax
154   400654:   89 05 da 04 20 00       mov    %eax,0x2004da(%rip)        # 600b34 <ticket>
```

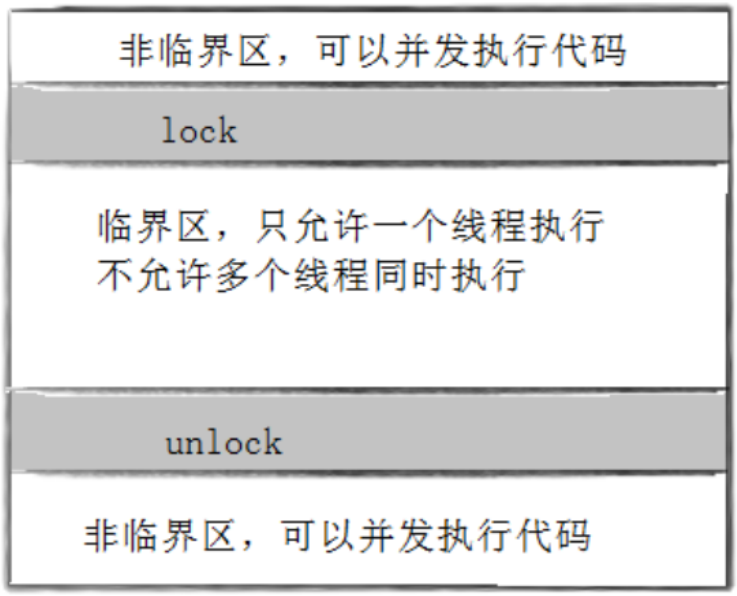
`--`操作并不是原子操作，而是对应三条汇编指令:

> - `load`：将共享变量ticket从内存加载到寄存器中
> - `update`:更新寄存器里面的值，执行-1操作
> - `store`：将新值，从寄存器写回共享变量ticket的内存地址

要解决以上问题，需要做到三点:

> - 代码必须要有互斥行为：当代码进入临界区执行时，不允许其他线程进入该临界区。
> - 如果多个线程同时要求执行临界区的代码，并且临界区没有线程在执行，那么只能允许一个线程进入该临界区。
> - 如果线程不在临界区中执行，那么该线程不能阻止其他线程进入临界区。

要做到这三点，本质上就是需要一把锁。Linux上提供的这把锁叫互斥量。



互斥量的接口

初始化互斥量
初始化互斥量有两种方法：

> - 方法1，静态分配：

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER
```

> - 方法2，动态分配：

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict
attr);
    参数：
        mutex：要初始化的互斥量
        attr：NULL
```

销毁互斥量
销毁互斥量需要注意：

> - 使用`PTHREAD_ MUTEX_ INITIALIZER`初始化的互斥量不需要销毁
> - 不要销毁一个已经加锁的互斥量
> - 已经销毁的互斥量，要确保后面不会有线程再尝试加锁

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

互斥量加锁和解锁

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
返回值:成功返回0,失败返回错误号
```

调用`pthread_ lock`时，可能会遇到以下情况：

> - 互斥量处于未锁状态，该函数会将互斥量锁定，同时返回成功
> - 发起函数调用时，其他线程已经锁定互斥量，或者存在其他线程同时申请互斥量，但没有竞争到互斥量，那么pthread_ lock调用会陷入阻塞(执行流被挂起)，等待互斥量解锁。

改进上面的售票系统:

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

int ticket = 100;
pthread_mutex_t mutex;

void *route(void *arg)
{
    char *id = (char*)arg;
    while ( 1 ) {
        pthread_mutex_lock(&mutex);
        if ( ticket > 0 ) {
            usleep(1000);
            printf("%s sells ticket:%d\n", id, ticket);
            ticket--;
            pthread_mutex_unlock(&mutex);
            // sched_yield(); 放弃CPU
        } else {
            pthread_mutex_unlock(&mutex);
            break;
        }
    }
}

int main( void )
{
    pthread_t t1, t2, t3, t4;

    pthread_mutex_init(&mutex, NULL);

    pthread_create(&t1, NULL, route, "thread 1");
    pthread_create(&t2, NULL, route, "thread 2");
    pthread_create(&t3, NULL, route, "thread 3");
    pthread_create(&t4, NULL, route, "thread 4");

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    pthread_join(t4, NULL);
    pthread_mutex_destroy(&mutex);
}
```

###互斥量实现原理探究

- 经过上面的例子，大家已经意识到单纯的`i++`或者`++i`都不是原子的，有可能会有数据一致性问题
- 为了实现互斥锁操作,大多数体系结构都提供了swap或exchange指令,该指令的作用是把寄存器和内存单元的数据相交换,由于只有一条指令,保证了原子性,即使是多处理器平台,访问内存的 总线周期也有先后,一个处理器上的交换指令执行时另一个处理器的交换指令只能等待总线周期。 现在我们把lock和unlock的伪代码改一下

```
lock:
        movb $0, %al
        xchgb %al, mutex
        if(al寄存器的内容 > 0){
                return 0;
        } else
                挂起等待;
        goto lock;

unlock:
        movb $1, mutex
        唤醒等待Mutex的线程;
        return 0;
```

###可重入VS线程安全

####概念

- 线程安全：多个线程并发同一段代码时，不会出现不同的结果。常见对全局变量或者静态变量进行操作，并且没有锁保护的情况下，会出现该问题。
- 重入：同一个函数被不同的执行流调用，当前一个流程还没有执行完，就有其他的执行流再次进入，我们称之为重入。一个函数在重入的情况下，运行结果不会出现任何不同或者任何问题，则该函数被称为可重入函数，否则，是不可重入函数。

####常见的线程不安全的情况

- 不保护共享变量的函数
- 函数状态随着被调用，状态发生变化的函数
- 返回指向静态变量指针的函数
- 调用线程不安全函数的函数

常见的线程安全的情况

- 每个线程对全局变量或者静态变量只有读取的权限，而没有写入的权限，一般来说这些线程是安全的
- 类或者接口对于线程来说都是原子操作
- 多个线程之间的切换不会导致该接口的执行结果存在二义性

常见不可重入的情况

- 调用了malloc/free函数，因为malloc函数是用全局链表来管理堆的
- 调用了标准I/O库函数，标准I/O库的很多实现都以不可重入的方式使用全局数据结构
- 可重入函数体内使用了静态的数据结构

常见可重入的情况

> - 不使用全局变量或静态变量
> - 不使用用malloc或者new开辟出的空间
> - 不调用不可重入函数
> - 不返回静态或全局数据，所有数据都有函数的调用者提供
> - 使用本地数据，或者通过制作全局数据的本地拷贝来保护全局数据

可重入与线程安全联系

> - 函数是可重入的，那就是线程安全的
> - 函数是不可重入的，那就不能由多个线程使用，有可能引发线程安全问题
> - 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。

可重入与线程安全区别

> - 可重入函数是线程安全函数的一种
> - 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。
> - 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。

6. 常见锁概念

死锁

> - 死锁是指在一组进程中的各个进程均占有不会释放的资源，但因互相申请被其他进程所站用不会释放的资源而处于的一种永久等待状态。

###死锁四个必要条件

> - 互斥条件：一个资源每次只能被一个执行流使用
> - 请求与保持条件：一个执行流因请求资源而阻塞时，对已获得的资源保持不放
> - 不剥夺条件:一个执行流已获得的资源，在末使用完之前，不能强行剥夺
> - 循环等待条件:若干执行流之间形成一种头尾相接的循环等待资源的关系

###避免死锁

> - 破坏死锁的四个必要条件
> - 加锁顺序一致
> - 避免锁未释放的场景
> - 资源一次性分配

###避免死锁算法

> - 死锁检测算法(了解)
> - 银行家算法（了解）

##7. Linux线程同步

###条件变量

> - 当一个线程互斥地访问某个变量时，它可能发现在其它线程改变状态之前，它什么也做不了。
> - 例如一个线程访问队列时，发现队列为空，它只能等待，只到其它线程将一个节点添加到队列中。这种情况就需要用到条件变量。

####同步概念与竞态条件

> - 同步：在保证数据安全的前提下，让线程能够按照某种特定的顺序访问临界资源，从而有效避免饥饿问题，叫做同步
> - 竞态条件：因为时序问题，而导致程序异常，我们称之为竞态条件。在线程场景下，这种问题也不难理解

####条件变量函数 *初始化*

```
int pthread_cond_init(pthread_cond_t *restrict cond,const pthread_condattr_t *restrict
attr);
参数:
    cond：要初始化的条件变量
    attr：NULL
```

销毁

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

等待条件满足

```
int pthread_cond_wait(pthread_cond_t *restrict cond,pthread_mutex_t *restrict mutex);
    参数:
        cond：要在这个条件变量上等待
        mutex：互斥量，后面详细解释
```

唤醒等待

```
   int pthread_cond_broadcast(pthread_cond_t *cond);
   int pthread_cond_signal(pthread_cond_t *cond);
```

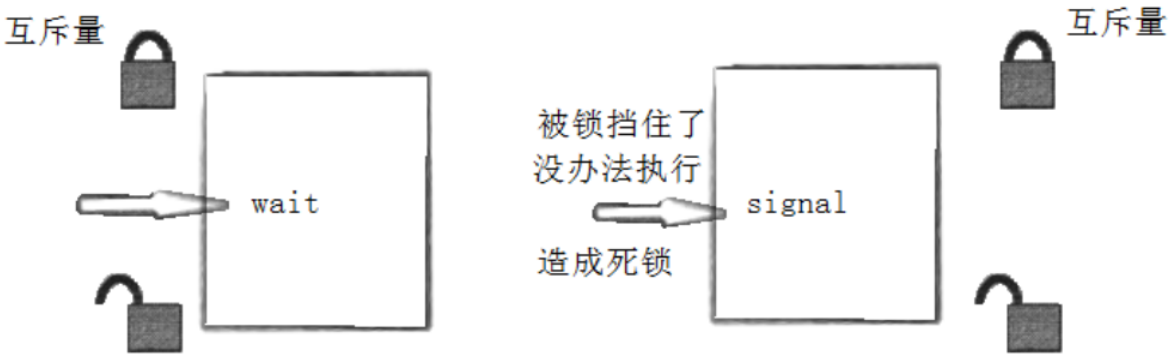
简单案例:

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

pthread_cond_t cond;
pthread_mutex_t mutex;

void *r1( void *arg )
{
    while ( 1 ){
        pthread_cond_wait(&cond, &mutex);
        printf("活动\n");
    }
}

void *r2(void *arg )
{
    while ( 1 ) {
```

```
        pthread_cond_signal(&cond);
        sleep(1);
    }
}

int main( void )
{
    pthread_t t1, t2;

    pthread_cond_init(&cond, NULL);
    pthread_mutex_init(&mutex, NULL);

    pthread_create(&t1, NULL, r1, NULL);
    pthread_create(&t2, NULL, r2, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);
}

[root@localhost linux]# ./a.out
    活动
    活动
    活动
```

为什么`pthread_cond_wait`需要互斥量?

> - 条件等待是线程间同步的一种手段，如果只有一个线程，条件不满足，一直等下去都不会满足，所以必须要有一个线程通过某些操作，改变共享变量，使原先不满足的条件变得满足，并且友好的通知等待在条件变量上的线程。
> - 条件不会无缘无故的突然变得满足了，必然会牵扯到共享数据的变化。所以一定要用互斥锁来保护。没有互斥锁就无法安全的获取和修改共享数据。



> - 按照上面的说法，我们设计出如下的代码：先上锁，发现条件不满足，解锁，然后等待在条件变量上不就行了，如下代码:

```
// 错误的设计
pthread_mutex_lock(&mutex);
while (condition_is_false) {
    pthread_mutex_unlock(&mutex);
    //解锁之后，等待之前，条件可能已经满足，信号已经发出，但是该信号可能被错过
    pthread_cond_wait(&cond);
    pthread_mutex_lock(&mutex);
}
pthread_mutex_unlock(&mutex);
```

> - 由于解锁和等待不是原子操作。调用解锁之后，`pthread_cond_wait`之前，如果已经有其他线程获取到互斥量，摒弃条件满足，发送了信号，那么`pthread_cond_wait`将错过这个信号，可能会导致线程永远阻塞在这个`pthread_cond_wait`。所以解锁和等待必须是一个原子操作。
> - `int pthread_cond_wait(pthread_cond_ t *cond,pthread_mutex_ t * mutex);`进入该函数后，会去看条件量等于0不？等于，就把互斥量变成1，直到cond_ wait返回，把条件量改成1，把互斥量恢复成原样。

###条件变量使用规范

> - 等待条件代码

```
pthread_mutex_lock(&mutex);
while (条件为假)
    pthread_cond_wait(cond, mutex);
修改条件
pthread_mutex_unlock(&mutex);
```

> - 给条件发送信号代码

```
pthread_mutex_lock(&mutex);
设置条件为真
pthread_cond_signal(cond);
pthread_mutex_unlock(&mutex);
```

8. 生产者消费者模型

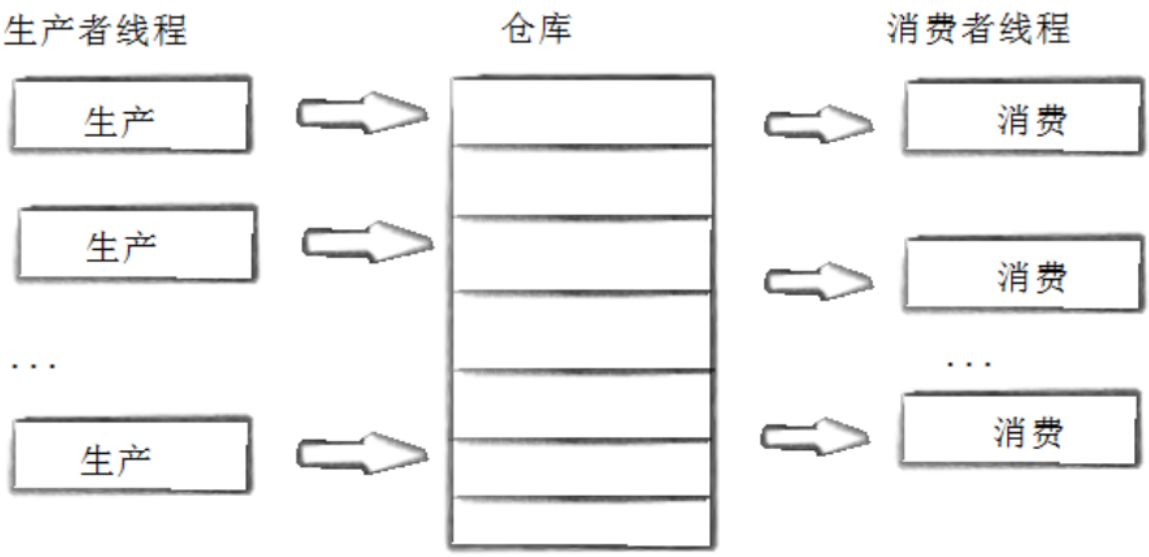
> - 321原则(便于记忆)

为何要使用生产者消费者模型

生产者消费者模式就是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。这个阻塞队列就是用来给生产者和消费者解耦的。

生产者消费者模型优点

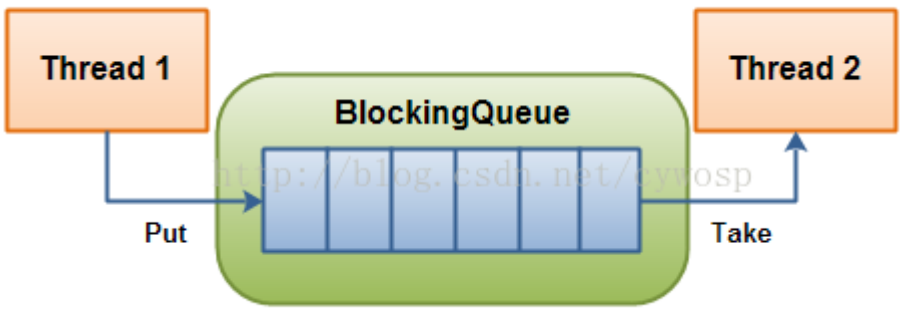
> - 解耦
> - 支持并发
> - 支持忙闲不均



基于BlockingQueue的生产者消费者模型

`BlockingQueue`

在多线程编程中阻塞队列(Blocking Queue)是一种常用于实现生产者和消费者模型的数据结构。其与普通的队列区别在于，当队列为空时，从队列获取元素的操作将会被阻塞，直到队列中被放入了元素；当队列满时，往队列里存放元素的操作也会被阻塞，直到有元素被从队列中取出(以上的操作都是基于不同的线程来说的，线程在对阻塞队列进程操作时会被阻塞)



C++ queue模拟阻塞队列的生产消费模型

代码:

> - 为了便于同学们理解，我们以单生产者，单消费者，来进行讲解。

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

#define NUM 8

class BlockQueue{
    private:
```

```
    std::queue<int> q;
    int cap;
    pthread_mutex_t lock;
    pthread_cond_t full;
    pthread_cond_t empty;

private:
    void LockQueue()
    {
        pthread_mutex_lock(&lock);
    }
    void UnLockQueue()
    {
        pthread_mutex_unlock(&lock);
    }
    void ProductWait()
    {
        pthread_cond_wait(&full, &lock);
    }
    void ConsumeWait()
    {
        pthread_cond_wait(&empty, &lock);
    }
    void NotifyProduct()
    {
        pthread_cond_signal(&full);
    }
    void NotifyConsume()
    {
        pthread_cond_signal(&empty);
    }
    bool IsEmpty()
    {
        return ( q.size() == 0 ? true : false );
    }
    bool IsFull()
    {
        return ( q.size() == cap ? true : false );
    }

public:
    BlockQueue(int _cap = NUM):cap(_cap)
    {
        pthread_mutex_init(&lock, NULL);
        pthread_cond_init(&full, NULL);
        pthread_cond_init(&empty, NULL);
    }
    void PushData(const int &data)
    {
        LockQueue();
        while(IsFull()){
            NotifyConsume();
            std::cout << "queue full, notify consume data, product stop." << std::endl;
```

```
                ProductWait();
            }
            q.push(data);
//          NotifyConsume();
            UnLockQueue();
        }
        void PopData(int &data)
        {
            LockQueue();
            while(IsEmpty()){
                NotifyProduct();
                std::cout << "queue empty, notify product data, consume stop." << std::endl;
                ConsumeWait();
            }
            data = q.front();
            q.pop();
//          NotifyProduct();
            UnLockQueue();
        }
        ~BlockQueue()
        {
            pthread_mutex_destroy(&lock);
            pthread_cond_destroy(&full);
            pthread_cond_destroy(&empty);
        }
};

void *consumer(void *arg)
{
    BlockQueue *bqp = (BlockQueue*)arg;
    int data;
    for( ; ; ){
        bqp->PopData(data);
        std::cout << "Consume data done : " << data << std::endl;
    }
}

//more faster
void *producter(void *arg)
{
    BlockQueue *bqp = (BlockQueue*)arg;
    srand((unsigned long)time(NULL));
    for( ; ; ){
        int data = rand() % 1024;
        bqp->PushData(data);
        std::cout << "Prodoct data done: " << data << std::endl;
    //  sleep(1);
    }
}

int main()
{
    BlockQueue bq;
```

```
    pthread_t c,p;
    pthread_create(&c, NULL, consumer, (void*)&bq);
    pthread_create(&p, NULL, producter, (void*)&bq);

    pthread_join(c, NULL);
    pthread_join(p, NULL);
    return 0;
}
```

POSIX信号量

POSIX信号量和SystemV信号量作用相同，都是用于同步操作，达到无冲突的访问共享资源目的。 但POSIX可以用于线程间同步。

初始化信号量

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
参数:
    pshared:0表示线程间共享，非零表示进程间共享
    value：信号量初始值
```

销毁信号量

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

等待信号量

```
功能：等待信号量，会将信号量的值减1
int sem_wait(sem_t *sem); //P()
```

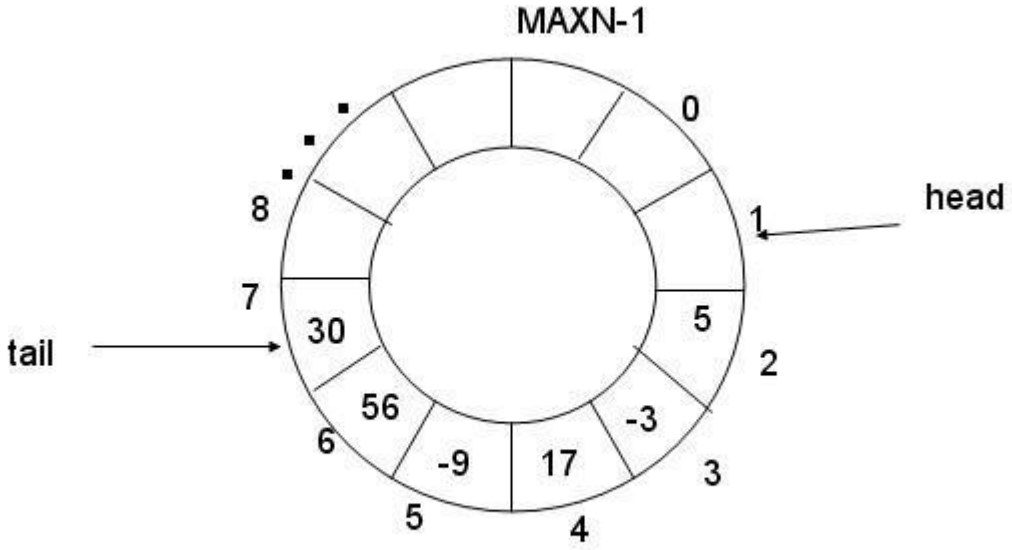
发布信号量

```
功能：发布信号量，表示资源使用完毕，可以归还资源了。将信号量值加1。
int sem_post(sem_t *sem);//V()
```

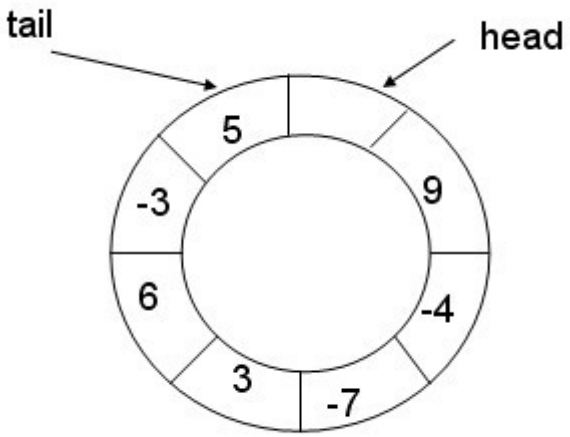
上一节生产者-消费者的例子是基于queue的,其空间可以动态分配,现在基于固定大小的环形队列重写这个程序（POSIX信号量）：

基于环形队列的生产消费模型

> - 环形队列采用数组模拟，用模运算来模拟环状特性



- 环形结构起始状态和结束状态都是一样的，不好判断为空或者为满，所以可以通过加计数器或者标记位来判断满或者空。另外也可以预留一个空的位置，作为满的状态



- 但是我们现在有信号量这个计数器，就很简单的进行多线程间的同步过程

```
#include <iostream>
#include <vector>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>

#define NUM 16

class RingQueue{
```

```
    private:
        std::vector<int> q;
        int cap;
        sem_t data_sem;
        sem_t space_sem;
        int consume_step;
        int product_step;

    public:
        RingQueue(int _cap = NUM):q(_cap),cap(_cap)
        {
            sem_init(&data_sem, 0, 0);
            sem_init(&space_sem, 0, cap);
            consume_step = 0;
            product_step = 0;
        }
        void PutData(const int &data)
        {
            sem_wait(&space_sem); // P
            q[consume_step] = data;
            consume_step++;
            consume_step %= cap;
            sem_post(&data_sem); //V
        }
        void GetData(int &data)
        {
            sem_wait(&data_sem);
            data = q[product_step];
            product_step++;
            product_step %= cap;
            sem_post(&space_sem);
        }
        ~RingQueue()
        {
            sem_destroy(&data_sem);
            sem_destroy(&space_sem);
        }
};
void *consumer(void *arg)
{
    RingQueue *rqp = (RingQueue*)arg;
    int data;
    for( ; ; ){
        rqp->GetData(data);
        std::cout << "Consume data done : " << data << std::endl;
        sleep(1);
    }
}

//more faster
void *producter(void *arg)
{
    RingQueue *rqp = (RingQueue*)arg;
```

```
    srand((unsigned long)time(NULL));
    for( ; ; ){
        int data = rand() % 1024;
        rqp->PutData(data);
        std::cout << "Prodoct data done: " << data << std::endl;
    //  sleep(1);
    }
}

int main()
{
    RingQueue rq;
    pthread_t c,p;
    pthread_create(&c, NULL, consumer, (void*)&rq);
    pthread_create(&p, NULL, producter, (void*)&rq);

    pthread_join(c, NULL);
    pthread_join(p, NULL);
}
```

9. 线程池

```
/*threadpool.h*/
/*  线程池:
 *      一种线程使用模式。线程过多会带来调度开销，进而影响缓存局部性和整体性能。而线程池维护着多个线程，等待着监督管理者分配可并发执行的任务。这避免了在处理短时间任务时创建与销毁线程的代价。线程池不仅能够保证内核的充分利用，还能防止过分调度。可用线程数量应该取决于可用的并发处理器、处理器内核、内存、网络sockets等的数量。
 *  线程池的应用场景:
 *      1.  需要大量的线程来完成任务，且完成任务的时间比较短。 WEB服务器完成网页请求这样的任务，使用线程池技术是非常合适的。因为单个任务小，而任务数量巨大，你可以想象一个热门网站的点击次数。 但对于长时间的任务，比如一个Telnet连接请求，线程池的优点就不明显了。因为Telnet会话时间比线程的创建时间大多了。
 *      2.  对性能要求苛刻的应用，比如要求服务器迅速响应客户请求。
 *      3.  接受突发性的大量请求，但不至于使服务器因此产生大量线程的应用。突发性大量客户请求，在没有线程池情况下，将产生大量线程，虽然理论上大部分操作系统线程数目最大值不是问题，短时间内产生大量线程可能使内存到达极限，出现错误.
 *  线程池的种类:
 *  线程池示例:
 *      1.  创建固定数量线程池，循环从任务队列中获取任务对象，
 *      2.  获取到任务对象后，执行任务对象中的任务接口
 */

/*threadpool.hpp*/
#ifndef __M_TP_H__
#define __M_TP_H__
#include <iostream>
#include <queue>
#include <pthread.h>

#define MAX_THREAD  5
typedef bool (*handler_t)(int);
class ThreadTask
{
```

```
    private:
        int _data;
        handler_t _handler;
    public:
        ThreadTask():_data(-1), _handler(NULL) {}
        ThreadTask(int data, handler_t handler) {
            _data= data;
            _handler = handler;
        }
        void SetTask(int data, handler_t handler) {
            _data = data;
            _handler = handler;
        }
        void Run() {
            _handler(_data);
        }
};

class ThreadPool
{
    private:
        int _thread_max;
        int _thread_cur;
        bool _tp_quit;
        std::queue<ThreadTask *> _task_queue;
        pthread_mutex_t _lock;
        pthread_cond_t _cond;
    private:
        void LockQueue() {
            pthread_mutex_lock(&_lock);
        }
        void UnLockQueue() {
            pthread_mutex_unlock(&_lock);
        }
        void WakeUpOne() {
            pthread_cond_signal(&_cond);
        }
        void WakeUpAll() {
            pthread_cond_broadcast(&_cond);
        }
        void ThreadQuit() {
            _thread_cur--;
            UnLockQueue();
            pthread_exit(NULL);
        }
        void ThreadWait(){
            if (_tp_quit) {
                ThreadQuit();
            }
            pthread_cond_wait(&_cond, &_lock);
        }
        bool IsEmpty() {
            return _task_queue.empty();
```

```
    }
    static void *thr_start(void *arg) {
        ThreadPool *tp = (ThreadPool*)arg;
        while(1) {
            tp->LockQueue();
            while(tp->IsEmpty()) {
                tp->ThreadWait();
            }
            ThreadTask *tt;
            tp->PopTask(&tt);
            tp->UnLockQueue();
            tt->Run();
            delete tt;
        }
        return NULL;
    }
public:
    ThreadPool(int max=MAX_THREAD):_thread_max(max), _thread_cur(max),
    _tp_quit(false) {
        pthread_mutex_init(&_lock, NULL);
        pthread_cond_init(&_cond, NULL);
    }
    ~ThreadPool() {
        pthread_mutex_destroy(&_lock);
        pthread_cond_destroy(&_cond);
    }
    bool PoolInit() {
        pthread_t tid;
        for (int i = 0; i < _thread_max; i++) {
            int ret = pthread_create(&tid, NULL, thr_start, this);
            if (ret != 0) {
                std::cout<<"create pool thread error\n";
                return false;
            }
        }
        return true;
    }
    bool PushTask(ThreadTask *tt) {
        LockQueue();
        if (_tp_quit) {
            UnLockQueue();
            return false;
        }
        _task_queue.push(tt);
        WakeUpOne();
        UnLockQueue();
        return true;
    }
    bool PopTask(ThreadTask **tt) {
        *tt = _task_queue.front();
        _task_queue.pop();
        return true;
    }
```

```
        bool PoolQuit() {
            LockQueue();
            _tp_quit = true;
            UnLockQueue();
            while(_thread_cur > 0) {
                WakeUpAll();
                usleep(1000);
            }
            return true;
        }
};
#endif


/*main.cpp*/
bool handler(int data)
{
    srand(time(NULL));
    int n = rand() % 5;
    printf("Thread: %p Run Tast: %d--sleep %d sec\n", pthread_self(), data, n);
    sleep(n);
    return true;
}
int main()
{
    int i;

    ThreadPool pool;
    pool.PoolInit();
    for (i = 0; i < 10; i++) {
        ThreadTask *tt = new ThreadTask(i, handler);
        pool.PushTask(tt);
    }

    pool.PoolQuit();
    return 0;
}
```

```
g++ -std=c++0x test.cpp -o test -pthread -lrt
```

10. 线程安全的单例模式

什么是单例模式

单例模式是一种 "经典的, 常用的, 常考的" **设计模式**.

什么是设计模式

IT行业这么火, 涌入的人很多. 俗话说林子大了啥鸟都有. 大佬和菜鸡们两极分化的越来越严重. 为了让菜鸡们不太拖大佬的后腿, 于是大佬们针对一些经典的常见的场景, 给定了一些对应的解决方案, 这个就是 **设计模式**

单例模式的特点

某些类, 只应该具有一个对象(实例), 就称之为单例.

例如一个男人只能有一个媳妇.

在很多服务器开发场景中, 经常需要让服务器加载很多的数据 (上百G) 到内存中. 此时往往要用一个单例的类来管理这些数据.

饿汉实现方式和懒汉实现方式

[洗完的例子]

```
吃完饭，立刻洗碗，这种就是饿汉方式．因为下一顿吃的时候可以立刻拿着碗就能吃饭．
吃完饭，先把碗放下，然后下一顿饭用到这个碗了再洗碗，就是懒汉方式．
```

懒汉方式最核心的思想是 "延时加载". 从而能够优化服务器的启动速度.

饿汉方式实现单例模式

```
template <typename T>
class Singleton {
  static T data;
public:
  static T* GetInstance() {
    return &data;
  }
};
```

只要通过 Singleton 这个包装类来使用 T 对象, 则一个进程中只有一个 T 对象的实例.

懒汉方式实现单例模式

```
template <typename T>
class Singleton {
  static T* inst;
public:
  static T* GetInstance() {
    if (inst == NULL) {
      inst = new T();
    }
    return inst;
  }
};
```

存在一个严重的问题, 线程不安全.

第一次调用 GetInstance 的时候, 如果两个线程同时调用, 可能会创建出两份 T 对象的实例.

但是后续再次调用, 就没有问题了.

懒汉方式实现单例模式(线程安全版本)

```
// 懒汉模式，线程安全
template <typename T>
class Singleton {
  volatile static T* inst;  // 需要设置 volatile 关键字，否则可能被编译器优化.
  static std::mutex lock;
public:
  static T* GetInstance() {
    if (inst == NULL) {     // 双重判定空指针，降低锁冲突的概率，提高性能.
      lock.lock();          // 使用互斥锁，保证多线程情况下也只调用一次 new.
      if (inst == NULL) {
        inst = new T();
      }
      lock.unlock();
    }
    return inst;
  }
};
```

注意事项:

1. 加锁解锁的位置
2. 双重 if 判定, 避免不必要的锁竞争
3. volatile关键字防止过度优化

11. STL,智能指针和线程安全

STL中的容器是否是线程安全的?

不是.

原因是, STL 的设计初衷是将性能挖掘到极致, 而一旦涉及到加锁保证线程安全, 会对性能造成巨大的影响.

而且对于不同的容器, 加锁方式的不同, 性能可能也不同(例如hash表的锁表和锁桶).

因此 STL 默认不是线程安全. 如果需要在多线程环境下使用, 往往需要调用者自行保证线程安全.

智能指针是否是线程安全的?

对于 unique_ptr, 由于只是在当前代码块范围内生效, 因此不涉及线程安全问题.

对于 shared_ptr, 多个对象需要共用一个引用计数变量, 所以会存在线程安全问题. 但是标准库实现的时候考虑到了这个问题, 基于原子操作(CAS)的方式保证 shared_ptr 能够高效, 原子的操作引用计数.

12. 其他常见的各种锁

> - 悲观锁：在每次取数据时，总是担心数据会被其他线程修改，所以会在取数据前先加锁（读锁，写锁，行锁等），当其他线程想要访问数据时，被阻塞挂起。
> - 乐观锁：每次取数据时候，总是乐观的认为数据不会被其他线程修改，因此不上锁。但是在更新数据前，会判断其他数据在更新前有没有对数据进行修改。主要采用两种方式：版本号机制和CAS操作。
> - CAS操作：当需要更新数据时，判断当前内存值和之前取得的值是否相等。如果相等则用新值更新。若不等则失败，失败则重试，一般是一个自旋的过程，即不断重试。
> - 自旋锁，公平锁，非公平锁?

13. 读者写者问题［选学］

读写锁

在编写多线程的时候，有一种情况是十分常见的。那就是，有些公共数据修改的机会比较少。相比较改写，它们读的机会反而高的多。通常而言，在读的过程中，往往伴随着查找的操作，中间耗时很长。给这种代码段加锁，会极大地降低我们程序的效率。那么有没有一种方法，可以专门处理这种多读少写的情况呢？ 有，那就是读写锁。

读写锁的行为

当前锁状态	读锁请求	写锁请求
无锁	可以	可以
读锁	可以	阻塞
写锁	阻塞	阻塞

> - **注意：写独占，读共享，读锁优先级高**

读写锁接口

设置读写优先

```
int pthread_rwlockattr_setkind_np(pthread_rwlockattr_t *attr, int pref);
/*
pref 共有 3 种选择

PTHREAD_RWLOCK_PREFER_READER_NP （默认设置） 读者优先，可能会导致写者饥饿情况

PTHREAD_RWLOCK_PREFER_WRITER_NP 写者优先，目前有 BUG，导致表现行为和
PTHREAD_RWLOCK_PREFER_READER_NP 一致

PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP 写者优先，但写者不能递归加锁
*/
```

初始化

```
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,const pthread_rwlockattr_t
*restrict attr);
```

销毁

```
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

加锁和解锁

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

读写锁案例：

```
#include <vector>
#include <sstream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <unistd.h>
#include <pthread.h>


volatile int ticket = 1000;
pthread_rwlock_t rwlock;


void * reader(void * arg)
{
    char *id = (char *)arg;
    while (1) {
        pthread_rwlock_rdlock(&rwlock);
        if (ticket <= 0) {
            pthread_rwlock_unlock(&rwlock);
            break;
        }
        printf("%s: %d\n", id, ticket);
        pthread_rwlock_unlock(&rwlock);
        usleep(1);
    }

    return nullptr;
}


void * writer(void * arg)
{
    char *id = (char *)arg;
    while (1) {
        pthread_rwlock_wrlock(&rwlock);
        if (ticket <= 0) {
            pthread_rwlock_unlock(&rwlock);
            break;
        }
        printf("%s: %d\n", id, --ticket);
        pthread_rwlock_unlock(&rwlock);
        usleep(1);
    }

    return nullptr;
}


struct ThreadAttr
{
    pthread_t tid;
    std::string id;
```

```
};


std::string create_reader_id(std::size_t i)
{
    // 利用 ostringstream 进行 string 拼接
    std::ostringstream oss("thread reader ", std::ios_base::ate);
    oss << i;
    return oss.str();
}


std::string create_writer_id(std::size_t i)
{
    // 利用 ostringstream 进行 string 拼接
    std::ostringstream oss("thread writer ", std::ios_base::ate);
    oss << i;
    return oss.str();
}


void init_readers(std::vector<ThreadAttr>& vec)
{
    for (std::size_t i = 0; i < vec.size(); ++i) {
        vec[i].id = create_reader_id(i);

        pthread_create(&vec[i].tid, nullptr, reader, (void *)vec[i].id.c_str());
    }
}


void init_writers(std::vector<ThreadAttr>& vec)
{
    for (std::size_t i = 0; i < vec.size(); ++i) {
        vec[i].id = create_writer_id(i);

        pthread_create(&vec[i].tid, nullptr, writer, (void *)vec[i].id.c_str());
    }
}


void join_threads(std::vector<ThreadAttr> const& vec)
{
    // 我们按创建的 逆序 来进行线程的回收
    for (std::vector<ThreadAttr>::const_reverse_iterator it = vec.rbegin(); it !=
vec.rend(); ++it) {
        pthread_t const& tid = it->tid;
        pthread_join(tid, nullptr);
    }
}


void init_rwlock()
```

```
{
#if 0   // 写优先
    pthread_rwlockattr_t attr;
    pthread_rwlockattr_init(&attr);
    pthread_rwlockattr_setkind_np(&attr, PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP);
    pthread_rwlock_init(&rwlock, &attr);
    pthread_rwlockattr_destroy(&attr);
#else   // 读优先，会造成写饥饿
    pthread_rwlock_init(&rwlock, nullptr);
#endif
}


int main()
{
    // 测试效果不明显的情况下，可以加大 reader_nr
    // 但也不能太大，超过一定阈值后系统就调度不了主线程了
    const std::size_t reader_nr = 1000;
    const std::size_t writer_nr = 2;

    std::vector<ThreadAttr> readers(reader_nr);
    std::vector<ThreadAttr> writers(writer_nr);

    init_rwlock();

    init_readers(readers);
    init_writers(writers);

    join_threads(writers);
    join_threads(readers);

    pthread_rwlock_destroy(&rwlock);
}
```

```
main: main.cpp
    g++ -std=c++11 -Wall -Werror $^ -o $@ -lpthread
```

```
thread reader 180: 1000
thread reader 929: 1000
thread reader 285: 1000
thread reader 916: 1000
thread reader 93: 1000
thread reader 172: 1000
thread reader 752: 1000
thread reader 589: 1000
thread reader 727: 1000
thread reader 701: 1000
thread reader 375: 1000
thread reader 255: 1000
thread reader 213: 1000
thread reader 499: 1000
thread reader 318: 1000
thread reader 126: 1000
thread reader 158: 1000
thread reader 256: 1000
thread reader 444: 1000
thread reader 829: 1000
```