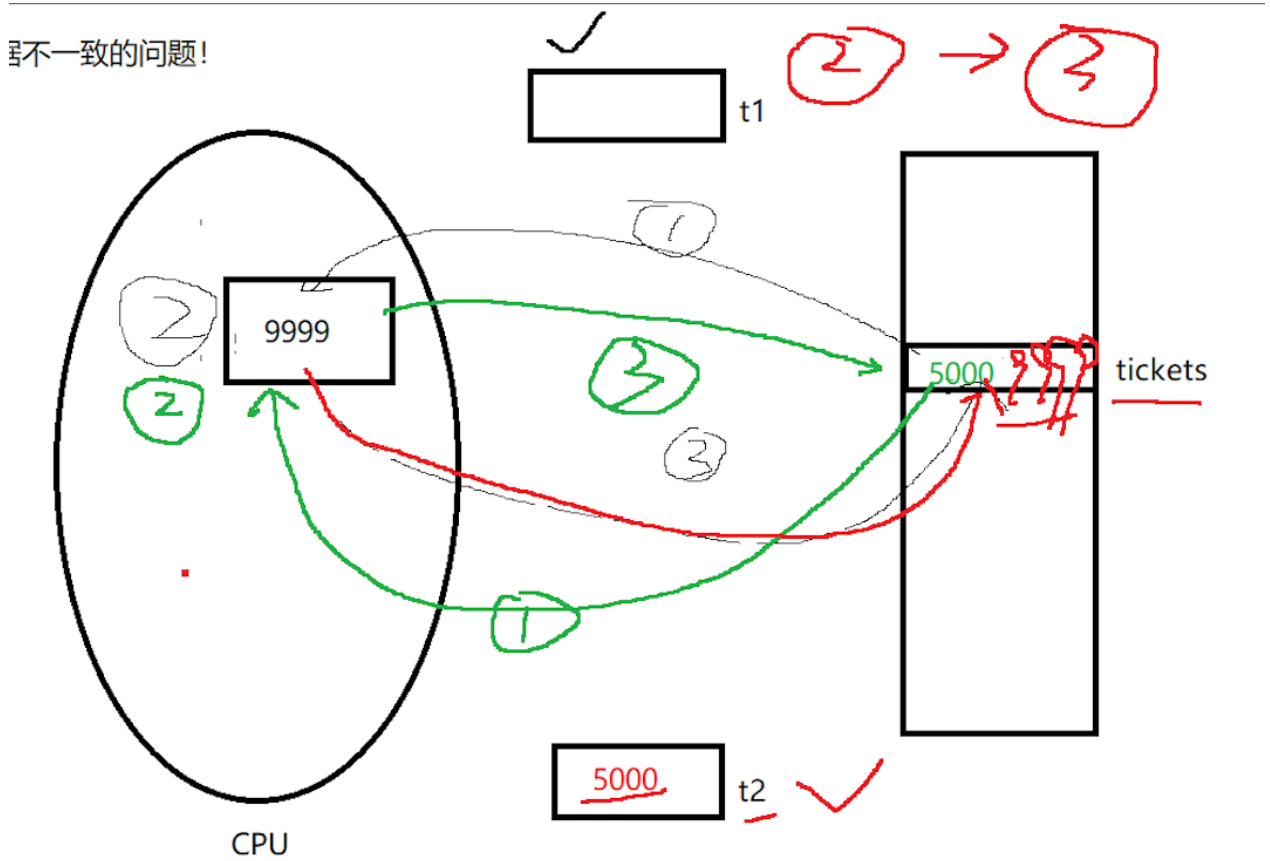
0112线程同步和互斥

抢票那份代码其实就可以体现：
代码本身没问题
但是多线程执行的时候出现问题

```
9
10  // 如果多线程访问同一个全局变量，并对它进行数据计算，多线程会互相影响吗?
11  int tickets = 10000; // 这里的10000就是临界资源
12                  临界资源
13  void *getTickets(void *args)
14  {
15      (void)args;
16      while (true)
17      {
18          if (tickets > 0) // 判断的本质也是计算的一种
19          {
20              usleep(1000);
21              printf("%p: %d\n", pthread_self(), tickets);
22              tickets--;
23          }
24          else
25          {
26              // 没有票了
```



--操作，其实相当于，

1. 把这个变量从内存load到cpu上
2. cpu进行计算
3. 把这个变量填回内存

而线程有可能在上面的任意一步中切换，因此对于不加保护的内存进行并行操作，就会出现问题

怎么避免这种现象发生呢？

我们对
tickets这样的全局数据进行保护
叫做：
加锁

我们最常用的：
pthread_mutex_init
互斥锁

PTHREAD_MUTEX_DESTROY(3P) POSIX Programmer's Manual PTHREAD_MUTEX_DESTROY(3P)

PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME

pthread_mutex_destroy, pthread_mutex_init - destroy and initialize a mutex

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
       const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

DESCRIPTION

The pthread_mutex_destroy() function shall destroy the mutex object referenced by mutex; the mutex object becomes, in effect, uninitialized. An implementation may cause pthread_mutex_destroy() to set the object referenced by mutex to an invalid value. A destroyed mutex object can be reinitialized using pthread_mutex_init(); the results of otherwise referencing the object after it has been destroyed are undefined.

It shall be safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex results in undefined behavior.

Manual page pthread_mutex_init(3p) line 1 (press h for help or q to quit)

对互斥锁的初始化有两种方法
第一种是调用初始化接口
第二种是直接把一个宏赋值给锁

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
       const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
8   #include <vector>
9
10  // 要对它进行加锁保护
11  pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER; // pthread_mutex_t 原生线程库提供的一个数据类型
12  int tickets = 10000;
13
14  void *getTickets(void *args)
15  {
16      (void)args;
17      while (true)
18      {
19          if (tickets > 0) // 判断的本质也是计算的一种
20          {
21              usleep(1000);
22              printf("%p: %d\n", pthread_self(), tickets);
23              tickets--;
24          }
25          else
26          {
```

这部分代码是会对临界资源
进行访问的

因此这部分区域叫做临界区

我们就是要对这一部分代码
进行加锁保护

pthread_mutex_lock

```
SYNOPSIS
       #include <pthread.h>

       int pthread_mutex_lock(pthread_mutex_t *mutex);
       int pthread_mutex_trylock(pthread_mutex_t *mutex);
       int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

```c
void *getTickets(void *args)
{
    (void)args;
    while (true)
    {
        pthread_mutex_lock(&mtx);
        if (tickets > 0) // 判断的本质也是计算的一种
        {
            usleep(1000);
            printf("%p: %d\n", pthread_self(), tickets);
            tickets--;
        }
        else
        {
            // 没有票了
            break;
        }
    }
}
```

这句话保证了
只有拿到锁的线程，才能向后执行
而这个接口保证
没有拿到锁的，锁在别人手上的线程
要在这里阻塞等待

加锁到解锁中间这部分代码
只能串行执行！

```c
(void)args;
while (true)
{
    pthread_mutex_lock(&mtx);
    if (tickets > 0) // 判断的本质也是计算的一种
    {
        usleep(1000);
        printf("%p: %d\n", pthread_self(), tickets);
        tickets--;
        pthread_mutex_unlock(&mtx);
    }
    else
    {
        // 没有票了
        pthread_mutex_unlock(&mtx);
        break;
    }
}
```

不要放到ifelse外面才解锁
这样如果中途break了
锁就一直没解开了

在加锁和解锁的中间这部分区域，叫做临界区！

```
0x7f758c0ca700: 11
0x7f758c0ca700: 10
0x7f758c0ca700: 9
0x7f758c0ca700: 8
0x7f758c0ca700: 7
0x7f758c0ca700: 6
0x7f758c0ca700: 5
0x7f758c0ca700: 4
0x7f758c0ca700: 3
0x7f758c0ca700: 2
0x7f758c0ca700: 1
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0112]$
```

我们发现上面的票都被一个线程抢走了
也是可能的
毕竟这是调度器调度的
如果我们想让大家都抢到，可以让休眠时间随机一点，或者让程序周期长一些，放多点票

为什么加锁不加在while前面
解锁放在while结束之后？

如果是这样，整个抢票的逻辑就完全串行了
这和没有多线程有什么区别？

所以我们加锁的时候，一定要保证加锁的粒度，越小越好！

```
#include <time.h>
// 要对它进行加锁保护
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER; // pthread_mutex_t 原生线程库提供的一个数据类型
int tickets = 10000;
```

如果锁定义在全局区域
就可以这样用宏定义

如果是定义锁为局部变量
就要用init接口定义

具体的实现见代码
通过传结构体的形式把锁传到线程的函数里面去

加了锁之后，线程在临界区中，是否会切换，会有问题吗？原子性的体现
加了锁之后，线程在临界区中，是否会切换？会切换，会有问题吗？不会！

虽然被切换，但是我们是持有锁被切换的！
其他执行流想要执行这部分代码，要申请锁，因此其他执行流申请锁会失败

加锁就是串行执行了吗？
是的！执行临界区代码一定是串行的

要访问临界资源，每一个线程都必须申请锁
前提是，每一个线程都必须先看到同一把锁 && 去访问它
那么，锁本身是不是一种共享临界资源？
谁来保证锁的安全呢？
所以为了保证锁的安全，申请和释放锁必须是原子的！！！

如何保证？锁究竟是什么？锁是如何实现的？

swap或exchange指令：
如果我们在汇编的角度，只有一条汇编语句，我们就认为该汇编语句的执行是原子的！

这句话，就保证锁只能被一个执行流获取

-swap或exchange指令 以一条汇编的方式，将内存和CPU内寄存区数据进行交换

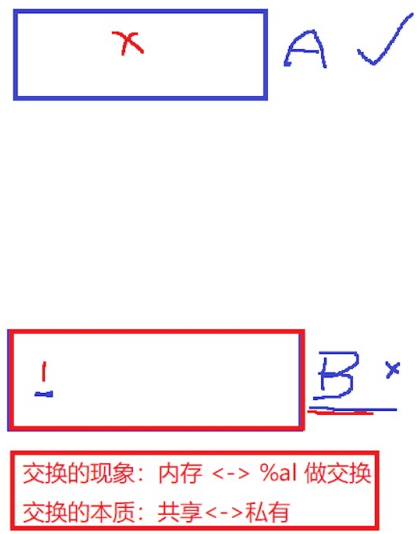
如果我们在汇编的角度，只有一条汇编语句，我们就认为该汇编语句的执行是原子的！

在执行流视角，是如何看待CPU上面的寄存器的？CPU内部的寄存器，本质，叫做当前执行流的上下文！！寄存器们，空间是被所有的执行流共享的，但是寄存器的内容，是被每一个执行流私有的！上下文！

```
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;
```

pthread_mutex_lock();

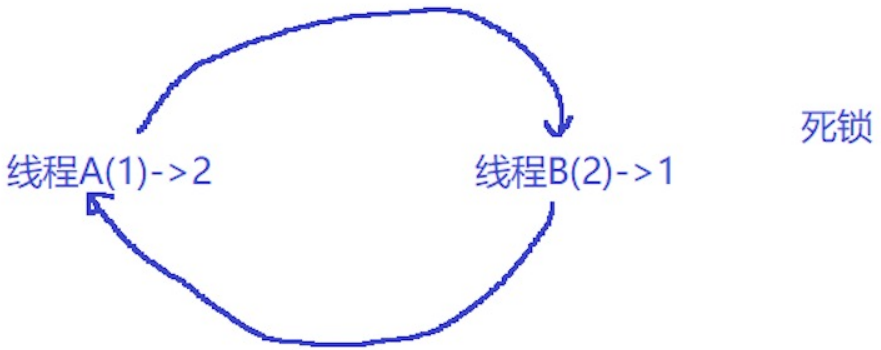
```
unlock:
        movb $1, mutex
        唤醒等待Mutex的线程;
        return 0;
```



####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

死锁



线程A需要2锁，此时2锁在线程B手上
线程B需要1锁，此时1锁在线程A手上

###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，在未使用完之前，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