0113线程同步

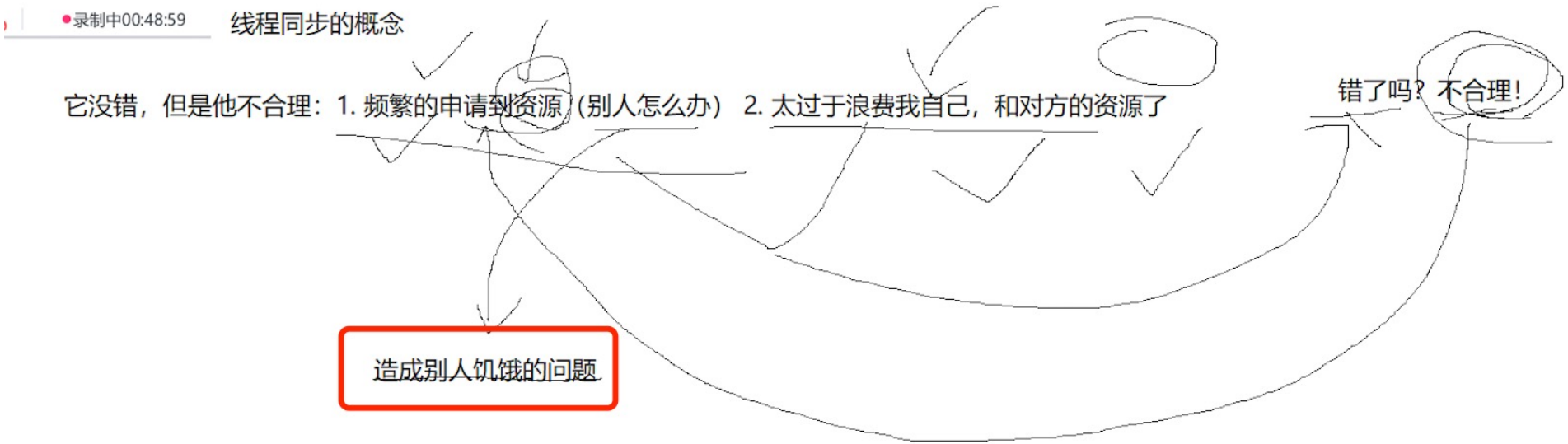
###避免死锁

> - 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）

线程同步



引入同步：
主要是为了解决访问临界资源和理性的问题

即：按照一定的顺序，进行临界资源的访问，这个叫做线程同步！

方案1：条件变量

当我们申请临界资源前：先要做临界资源是否存在的检测，要做检测的本质：也是访问临界资源！
因此，对临界资源的检测，也一定是需要在加锁和解锁之间的！

那么，有没有办法让我们的线程检测到资源部就绪的时候

1. 不要让线程再频繁的自己检测了，等待就行！
2. 当条件就绪的时候，通知对应的线程，让它来进行资源申请和访问

条件变量

PTHREAD_COND_DESTROY(3P) POSIX Programmer's Manual PTHREAD_COND_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_cond_destroy, pthread_cond_init - destroy and initialize condition variables

SYNOPSIS

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *restrict cond,
       const pthread_condattr_t *restrict attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

PTHREAD_COND_TIMEDWAIT(3P) POSIX Programmer's Manual PTHREAD_COND_TIMEDWAIT(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_cond_timedwait, pthread_cond_wait - wait on a condition

SYNOPSIS

```
#include <pthread.h>

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
       pthread_mutex_t *restrict mutex,
       const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond,
       pthread_mutex_t *restrict mutex);
```

```
PTHREAD_COND_BROADCAST(3P)                POSIX Programmer's Manual                PTHREAD_COND_BROADCAST(3P)

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

NAME
       pthread_cond_broadcast, pthread_cond_signal - broadcast or signal a condition

SYNOPSIS
       #include <pthread.h>

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

等待，唤醒就是上面这些接口

现在要写一个代码：
主线程能按照要求一次唤醒1，2，3，4的线程
主线程让你执行你就执行，不让你执行你就不要执行
我们要完成以上这种效果！

具体的代码要见服务器！

```
(base) [yufc@VM-12-12-centos:~/Core/BitCodeField/0113]$ ./mycond
Thread 1 A_running...
Thread 2 B_running...
Thread 3 C_running...
Thread 4 D_running...
Thread 1 A_running...
Thread 2 B_running...
Thread 3 C_running...
Thread 4 D_running...
Thread 1 A_running...Thread 2 B_running...

Thread 3 C_running...
Thread 4 D_running...
```

先把多线程并行运行的架构写好

```
23
24  // ==================== 线程要执行的函数 ==================== //
25  #if 1
26  void func1(const std::string &name, pthread_mutex_t *pmtx, pthread_cond_t *pcond)
27  {
28      while (1)
29      {
30          pthread_cond_wait(pcond, pmtx); //默认该线程再执行的时候，wait代码被执行，当前线程会立即被阻塞!
31          std::cout << name << " A_running..." << std::endl;
32          sleep(1);
33      }
34  }
35  void func2(const std::string &name, pthread_mutex_t *pmtx, pthread_cond_t *pcond)
36  {
37      while (1)
```

我们给每个线程都加入这句话
那么，如果主线程不唤醒
这四个线程都执行不了，都被挂起了！

那么，如果我按照一定规则，唤醒这些线程
我们就可以很明确的看到，线程的执行是有规律的！！

```
93          ThreadData *td = new ThreadData(name, funcs[i], &mtx
94          pthread_create(tids + i, nullptr, Entry, (void *)td)
95      }
96
97      //控制对应的线程按照一定规则执行
98      while(true)
99      {
100         pthread_cond_signal(&cond);
101         sleep(1);
102     }
103
```

此时，每隔一秒唤醒一个
在特定cond下
的其中一个
线程

```
(base) [yufc@VM-12-12-centos:~/Core/BitCodeField/0113]$ make
g++ -std=c++11 -o mycond mycond.cc -lpthread -g
(base) [yufc@VM-12-12-centos:~/Core/BitCodeField/0113]$ ./mycond
new threads generate success
main thread begin to control all new threads ...
Thread 2 B_running...
Thread 4 D_running...
Thread 3 C_running...
Thread 1 A_running...
```

```
(base) [yufc@VM-12-12-centos:~/Core/BitCodeField/0113]$ ./mycond
new threads generate success
main thread begin to control all new threads ...
Thread 1 A_running...
Thread 2 B_running...
Thread 3 C_running...
Thread 4 D_running...
Thread 1 A_running...
Thread 2 B_running...
Thread 3 C_running...
Thread 4 D_running...
Thread 1 A_running...
Thread 2 B_running...
Thread 3 C_running...
Thread 4 D_running...
Thread 1 A_running...
```

我们发现
有序的！

因为只有符合 cond的线程才会被唤醒

如果一个线程发现自己的cond不满足是
就会在该cond的队列中进行排队等待
排队，不就是有序吗？

PTHREAD_COND_BROADCAST(3P) POSIX Programmer's Manual PTHREAD_COND_BROADCAST(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_cond_broadcast, pthread_cond_signal - broadcast or signal a condition

SYNOPSIS

```
#include <pthread.h>

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

下面那个是唤醒一个符合cond的线程
上面那个是唤醒所有符合cond的线程，即唤醒一批线程

我们现在设置
当主线程倒数完cnt之后

所有线程退出
我们应该就能看到
所有线程的join信息

但是，为什么到最后
没有join信息，而是卡住了？
signal 和 broadcast都一样，为什么？

```
g++ -std=c++11 -o mycond mycond.cc -lpthread -g
(base) [yufc@VM-12-12-centos:~/Core/BitCodeField/0113/demo]$ ./mycond
new threads generate success
main thread begin to control all new threads ...
Thread 1 A_running...
Thread 2 B_running...
Thread 3 C_running...
Thread 4 D_running...
Thread 1 A_running...
Thread 2 B_running...
Thread 3 C_running...
Thread 4 D_running...
Thread 1 A_running...
Thread 2 B_running...
^C
(base) [yufc@VM-12-12-centos:~/Core/BitCodeField/0113/demo]$
```

为什么会出现上面的问题？

是因为我们的wait没有处理好！

我们的wait本质是在检测线程资源是否就绪
那么检测这个动作，本身就应该在临界区里面进行！

这里很不好理解，所以我们先阐述结论：
pthread_cond_wait一定要在加锁和解锁之间！

```
void func1(const std::string &name, pthread_mutex_t *pmtx, pthread_cond_t *pcond)
{
    while (!quit) // 当quit = true之后，所有线程退出
    {
        //wait一定在加锁和解锁之间!
        pthread_mutex_lock(pmtx);
        //这里要做的一件事其实就是:
        //if(临界资源就绪吗? 不就绪)
        //   刚开始学的做法是: break, 然后重新检测
        //   但是现在我不想让线程一直检测了，那么就让它去等！等到就绪为止
        //所以！其实我们访问了临界资源!
        pthread_cond_wait(pcond, pmtx); //默认该线程再执行的时候，wait代码被执行，当前线程会立即被阻塞!
        std::cout << name << " A_running..." << std::endl;
        // sleep(1);
        pthread_mutex_unlock(pmtx);
    }
}
```

重要！

生产者消费者模型（重要）

1. 条件满足的时候，我们再唤醒制定的进程 --- 我怎么知道条件是否满足呢？
2. mutex的意义？

---

1. 例子说明，生产消费的基本组成概念
2. 用基本工程师思维，重新理解CP