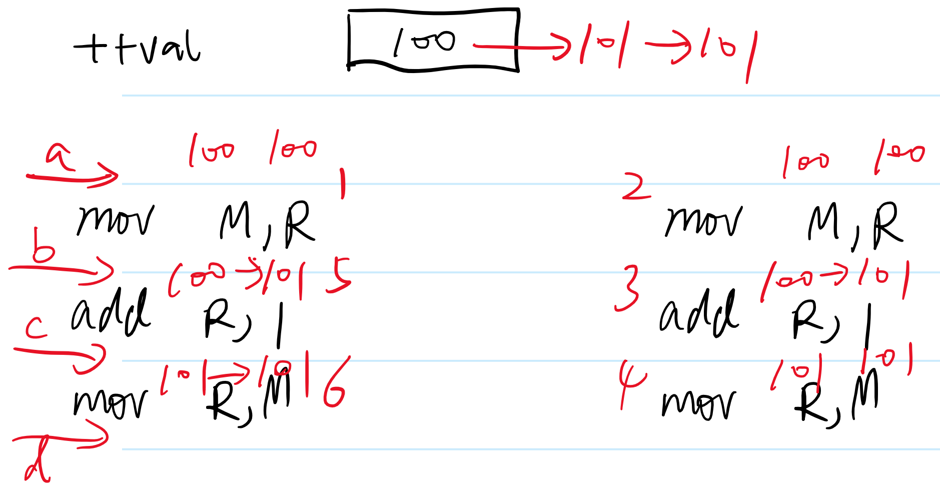
竞争条件

2023年8月17日 9:44

```c
#define NUM 10000000
void * threadFunc(void *arg){
    int * pval = (int *)arg;
    for(int i = 0; i < NUM; ++i){
        ++*pval;
    }
}
int main(int argc, char *argv[])
{
    int val = 0;
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,&val);

    for(int i = 0; i < NUM; ++i){
        ++val;
    }

    pthread_join(tid,NULL);
    printf("val = %d\n", val);
    return 0;
}
```



互斥

2023年8月17日 9:55

互相排斥 mutual exclusion

mutex

互斥锁 标志位

0	未锁
1	已锁.

原子操作

lock 测试并加锁

while(mutex 已锁)
    等待
}

→ 未锁 → 已锁.

正常执行.

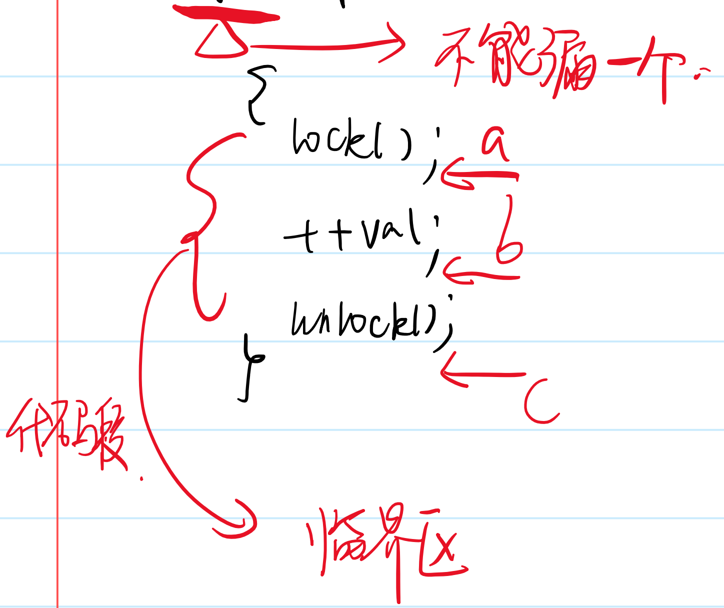
unlock 解锁.

使用互斥锁保护资源

2023年8月17日 10:04

① 弄一个共享的 mutex

② 每个线程 先lock 访问共享资源 再unlock.



```
{
    lock();
    ++val;
    unlock();
}
```

```
{
    lock();
    ++val;
    unlock();
}
```

pthread_mutex_init

2023年8月17日 10:15

pthread_mutex_t 锁的数据类型

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

动态 初始化

静态 初始化

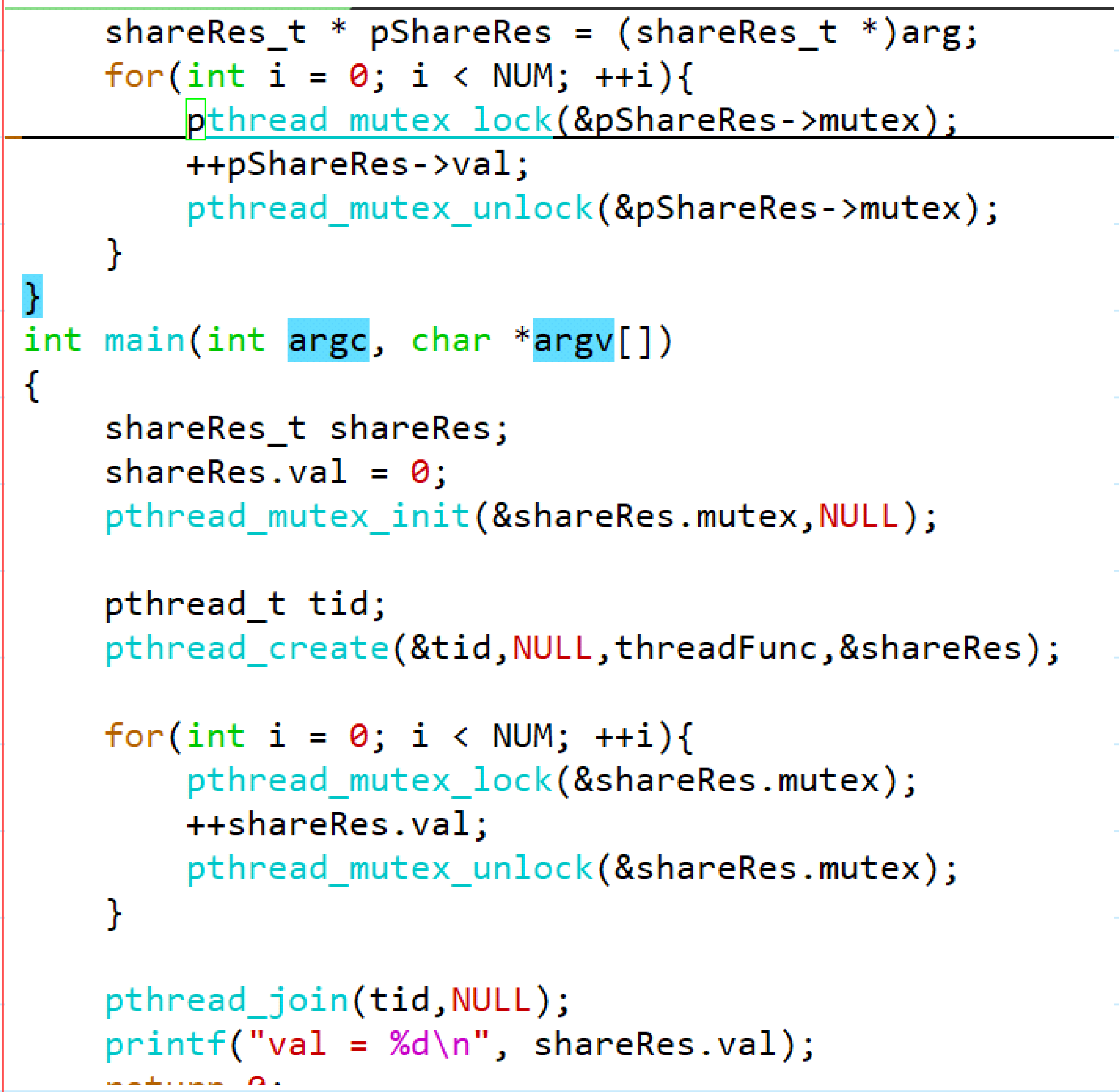
lock unlock

2023年8月17日 10:18

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

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

2023年8月17日 10:28



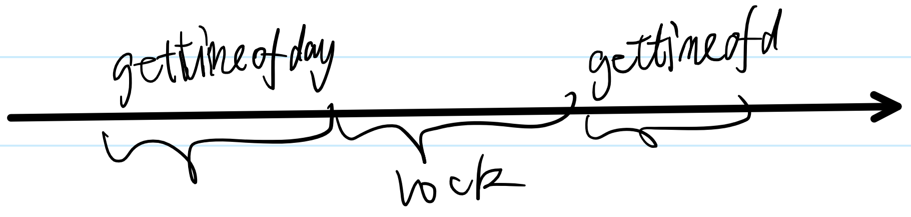
```
    shareRes_t * pShareRes = (shareRes_t *)arg;
    for(int i = 0; i < NUM; ++i){
        pthread_mutex_lock(&pShareRes->mutex);
        ++pShareRes->val;
        pthread_mutex_unlock(&pShareRes->mutex);
    }
}
int main(int argc, char *argv[])
{
    shareRes_t shareRes;
    shareRes.val = 0;
    pthread_mutex_init(&shareRes.mutex,NULL);

    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,&shareRes);

    for(int i = 0; i < NUM; ++i){
        pthread_mutex_lock(&shareRes.mutex);
        ++shareRes.val;
        pthread_mutex_unlock(&shareRes.mutex);
    }

    pthread_join(tid,NULL);
    printf("val = %d\n", shareRes.val);
```

gettimeofday

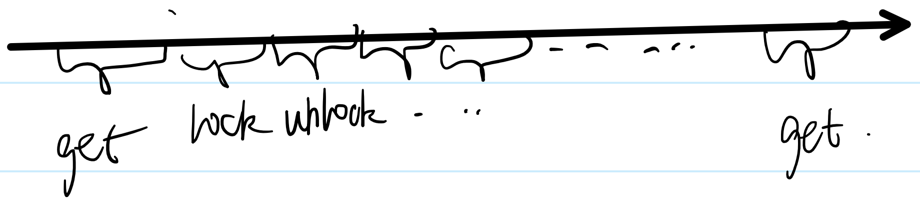
2023年8月17日 10:28

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

```
while(1){
    gettimeofday
    lock
    gettimeofday
```



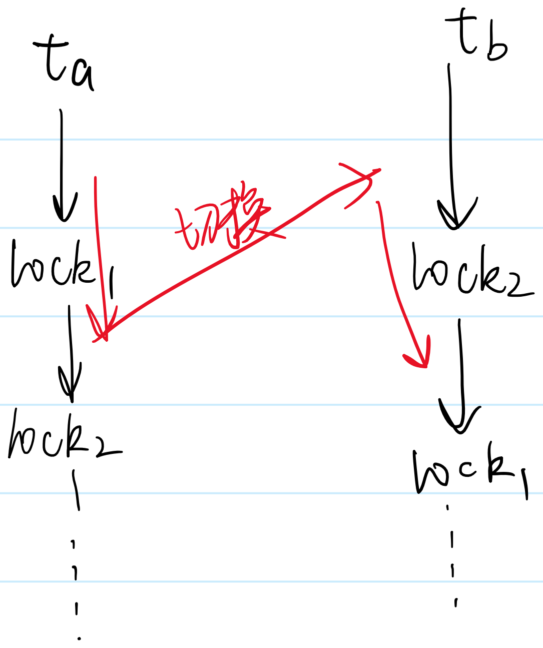
```
gettimeofday
while(1){

}
gettimeofday
```



第一种死锁

2023年8月17日 11:05

$mutex_1$ $mutex_2$



使用多个锁，加锁顺序不当.

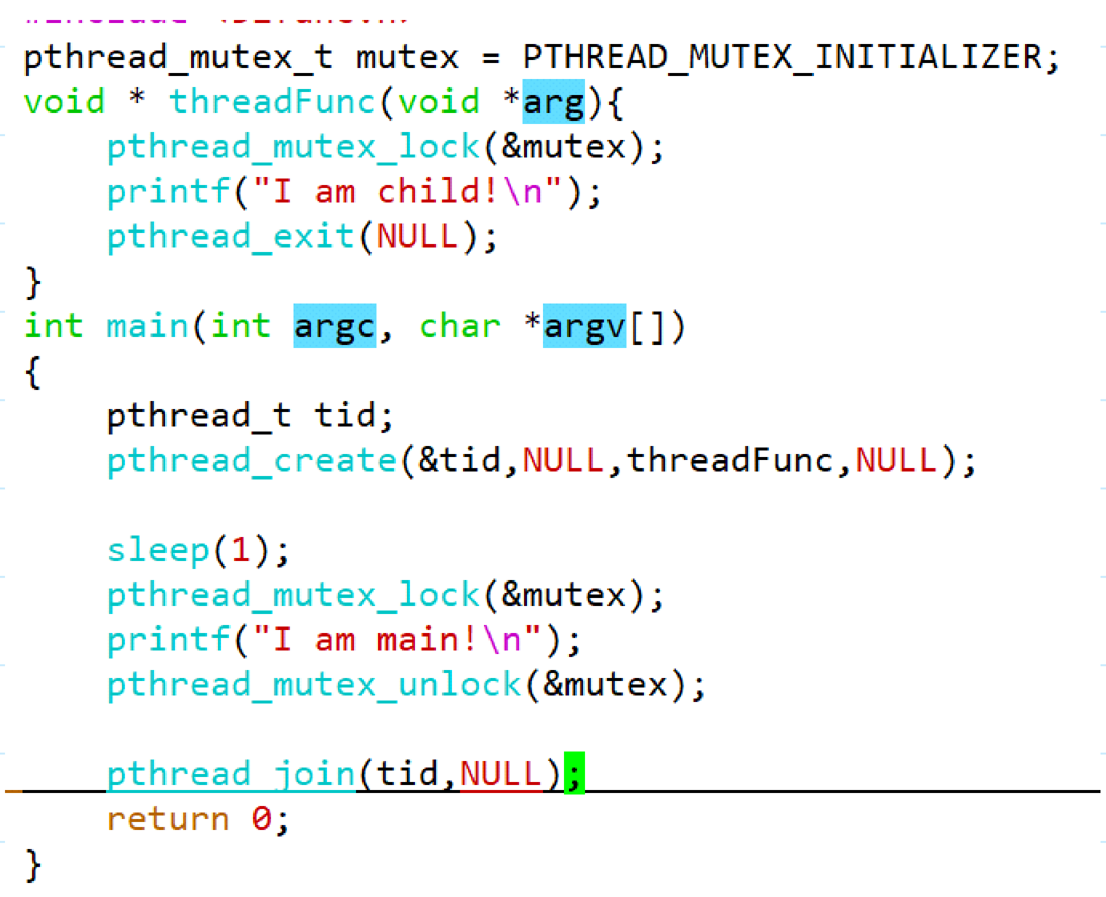
方案1. 调整顺序.
(可行性低)

方案2: 禁止使用多把锁

第二种死锁

2023年8月17日 11:12

持有锁的线程，在带锁情况下终止.



```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void * threadFunc(void *arg){
    pthread_mutex_lock(&mutex);
    printf("I am child!\n");
    pthread_exit(NULL);
}
int main(int argc, char *argv[])
{
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,NULL);

    sleep(1);
    pthread_mutex_lock(&mutex);
    printf("I am main!\n");
    pthread_mutex_unlock(&mutex);

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

解决方案

① 主动退出，退出前记得解锁

② 被取消，把解锁行为放入资源清理.

第三种死锁

2023年8月17日　11:20

持有锁的线程 对持有的锁再加锁

lock　未锁 → 已锁

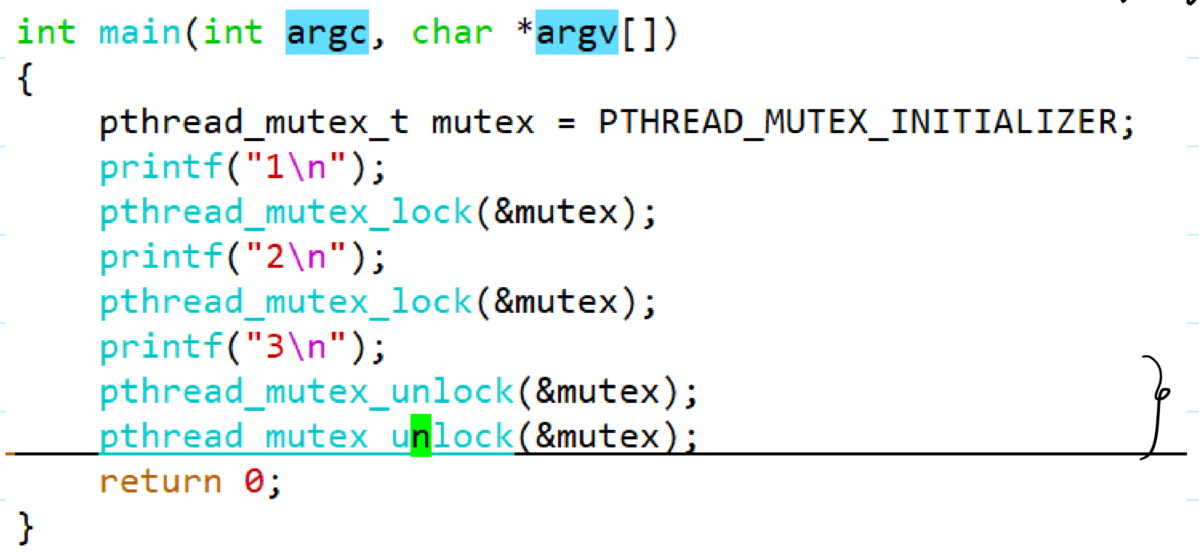
lock　已锁 → 等待

↓

不能继续

在加锁前考虑之前是否加过锁

```
while(1){
    lock()
    ...
    unlock()
}
```



```
int main(int argc, char *argv[])
{
    pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
    printf("1\n");
    pthread_mutex_lock(&mutex);
    printf("2\n");
    pthread_mutex_lock(&mutex);
    printf("3\n");
    pthread_mutex_unlock(&mutex);
    pthread_mutex_unlock(&mutex);
    return 0;
}
```

锁的属性

2023年8月17日 11:26

普通锁。 检错锁 可重入锁 → 解决第3种死锁。

检错锁 → 重复报错。

可重入锁 → 重复加锁，引用计数 +1

解锁 -1

检错锁

2023年8月17日 11:29

锁属性的类型

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

初始化

```
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

```
PTHREAD_MUTEX_NORMAL  PTHREAD_MUTEX_ERRORCHECK  PTHREAD_MUTEX_RECURSIVE  PTHREAD_MUTEX_DEFAULT
```

普通　检错.　可重入性　普通.

pthread_mutex_init

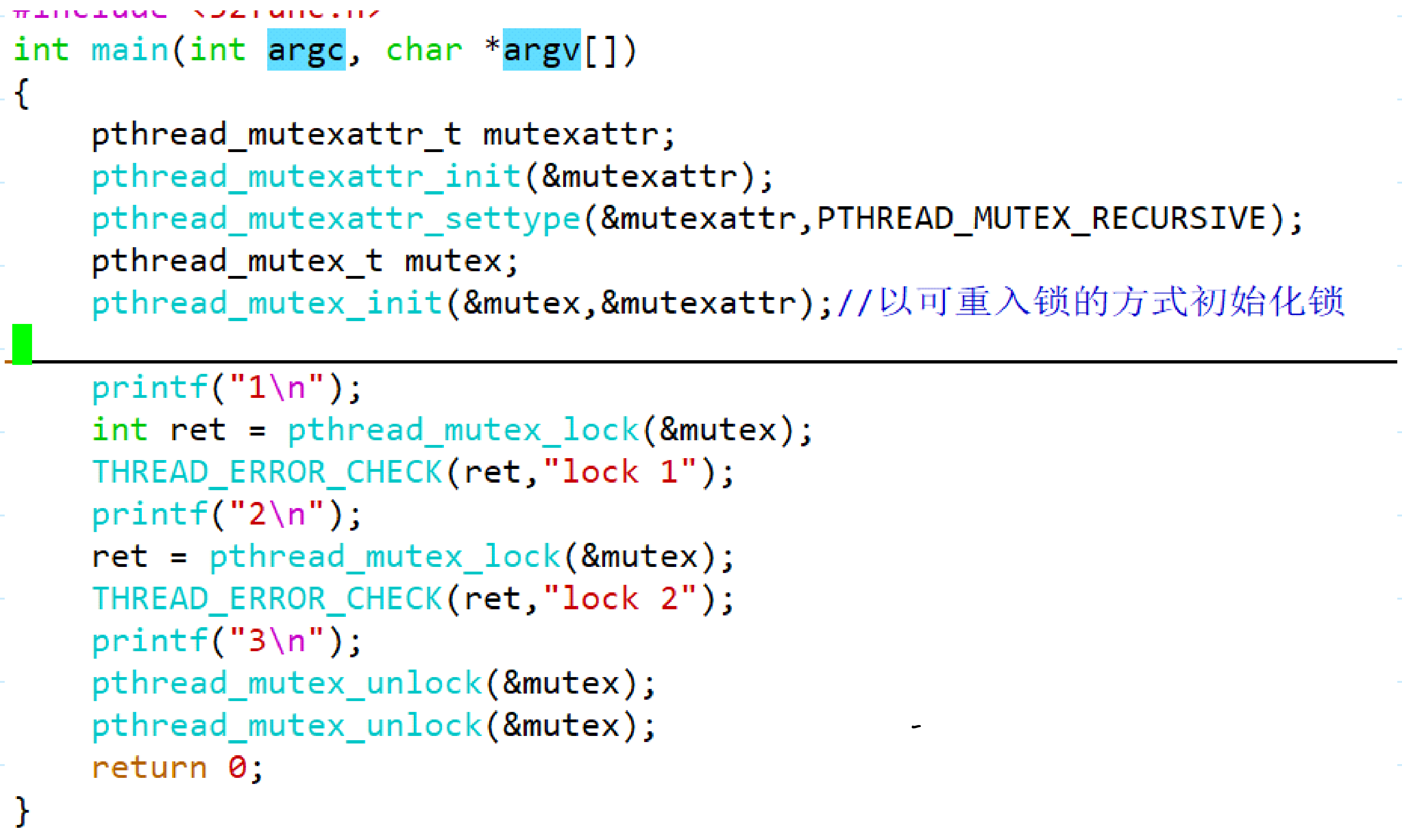
2023年8月17日　　11:36

```c
int main(int argc, char *argv[])
{
    pthread_mutexattr_t mutexattr;
    pthread_mutexattr_init(&mutexattr);
    pthread_mutexattr_settype(&mutexattr,PTHREAD_MUTEX_ERRORCHECK);
    pthread_mutex_t mutex;
    pthread_mutex_init(&mutex,&mutexattr);//以检错锁的方式初始化锁

    printf("1\n");
    int ret = pthread_mutex_lock(&mutex);
    THREAD_ERROR_CHECK(ret,"lock 1");
    printf("2\n");
    ret = pthread_mutex_lock(&mutex);
    THREAD_ERROR_CHECK(ret,"lock 2");
    printf("3\n");
    pthread_mutex_unlock(&mutex);
    pthread_mutex_unlock(&mutex);
    return 0;
}
```

可重入锁

2023年8月17日 11:37

把锁重复加锁，引用计数+1



```c
#include <32func.h>
int main(int argc, char *argv[])
{
    pthread_mutexattr_t mutexattr;
    pthread_mutexattr_init(&mutexattr);
    pthread_mutexattr_settype(&mutexattr,PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_t mutex;
    pthread_mutex_init(&mutex,&mutexattr);//以可重入锁的方式初始化锁

    printf("1\n");
    int ret = pthread_mutex_lock(&mutex);
    THREAD_ERROR_CHECK(ret,"lock 1");
    printf("2\n");
    ret = pthread_mutex_lock(&mutex);
    THREAD_ERROR_CHECK(ret,"lock 2");
    printf("3\n");
    pthread_mutex_unlock(&mutex);
    pthread_mutex_unlock(&mutex);
    return 0;
}
```

不同锁的特点

2023年8月17日 11:40

A已锁了一次.

普通锁

A再加锁 → 死锁

B加锁 → 阻塞

检错锁

A再加锁 → 报错

B加锁 → 阻塞.

可重入锁.

A再加锁 → 引用计数+1

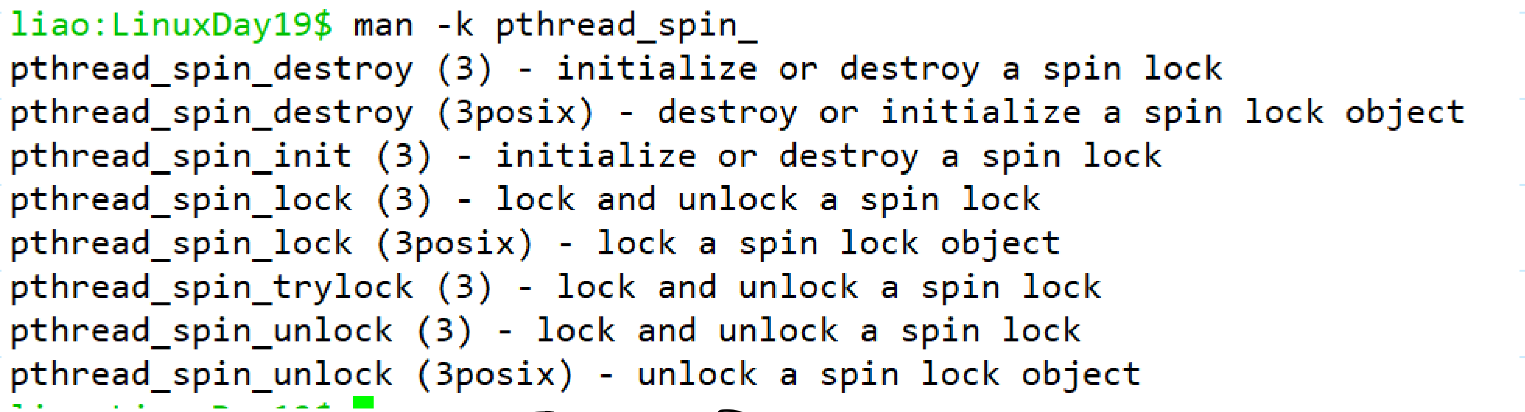
B加锁 → 阻塞

锁的类型

2023年8月17日 14:30

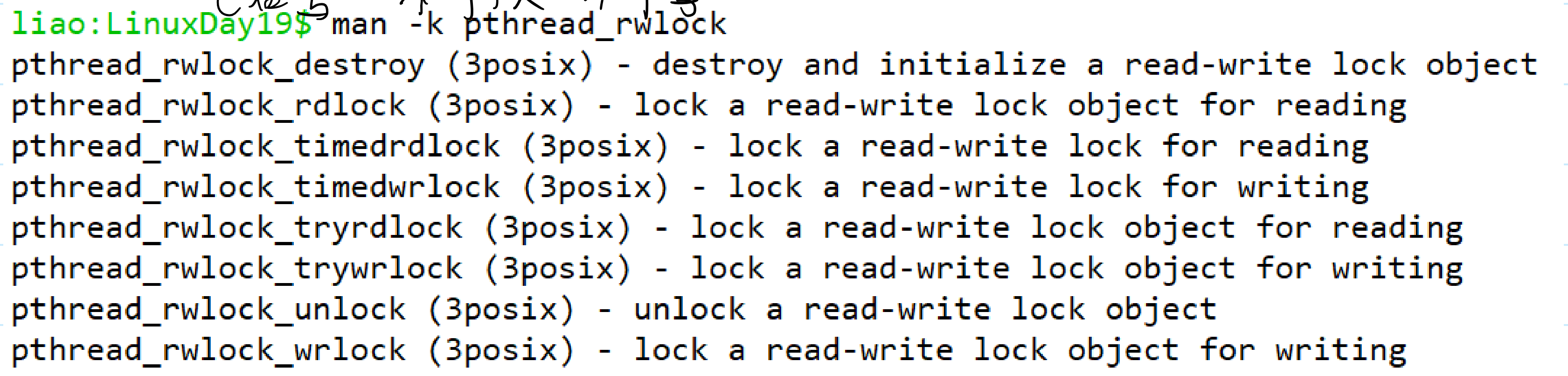
mutex 互斥锁 阻塞式.

spin 自旋锁. cpu 死循环.



```
liao:LinuxDay19$ man -k pthread_spin_
pthread_spin_destroy (3) - initialize or destroy a spin lock
pthread_spin_destroy (3posix) - destroy or initialize a spin lock object
pthread_spin_init (3) - initialize or destroy a spin lock
pthread_spin_lock (3) - lock and unlock a spin lock
pthread_spin_lock (3posix) - lock a spin lock object
pthread_spin_trylock (3) - lock and unlock a spin lock
pthread_spin_unlock (3) - lock and unlock a spin lock
pthread_spin_unlock (3posix) - unlock a spin lock object
```

读写锁 正在读 可读 不可写
正在写 不可读 不可写

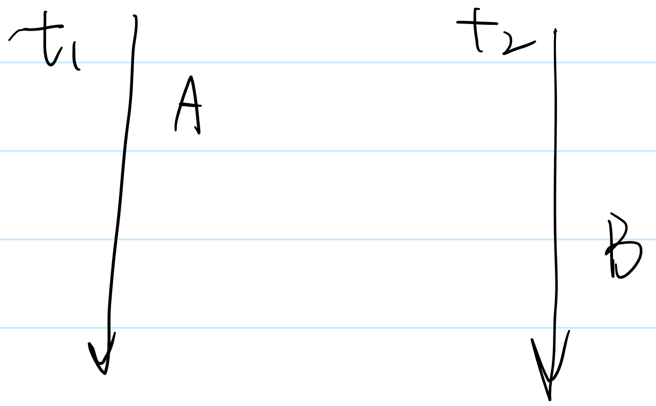


```
liao:LinuxDay19$ man -k pthread_rwlock
pthread_rwlock_destroy (3posix) - destroy and initialize a read-write lock object
pthread_rwlock_rdlock (3posix) - lock a read-write lock object for reading
pthread_rwlock_timedrdlock (3posix) - lock a read-write lock for reading
pthread_rwlock_timedwrlock (3posix) - lock a read-write lock for writing
pthread_rwlock_tryrdlock (3posix) - lock a read-write lock object for reading
pthread_rwlock_trywrlock (3posix) - lock a read-write lock object for writing
pthread_rwlock_unlock (3posix) - unlock a read-write lock object
pthread_rwlock_wrlock (3posix) - lock a read-write lock object for writing
```

同步

2023年8月17日 14:39

flag 条件 $\begin{cases} 0 & \text{A未完成} \\ 1 & \text{A已完成} \end{cases}$



永远 A先B后. 同步

```
A();
lock();
flag -> 1
unlock();
```

```
while(1){
    lock()
    if(flag 为1){
        unlock()
        break;
    }
    unlock()
}
B()
```

同步的代码

2023年8月17日 14:54

```c
int main(int argc, char *argv[])
{
    shareRes_t shareRes;
    shareRes.flag = 0;
    pthread_mutex_init(&shareRes.mutex,NULL);

    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,&shareRes);

    A();
    pthread_mutex_lock(&shareRes.mutex);
    shareRes.flag = 1;
    pthread_mutex_unlock(&shareRes.mutex);

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

```c
void *threadFunc(void *arg){
    shareRes_t * pShareRes = (shareRes_t *)arg;
    while(1){
        pthread_mutex_lock(&pShareRes->mutex);
        if(pShareRes->flag != 0){
            pthread_mutex_unlock(&pShareRes->mutex);
            break;
        }
        pthread_mutex_unlock(&pShareRes->mutex);
    }
    B();
}
```

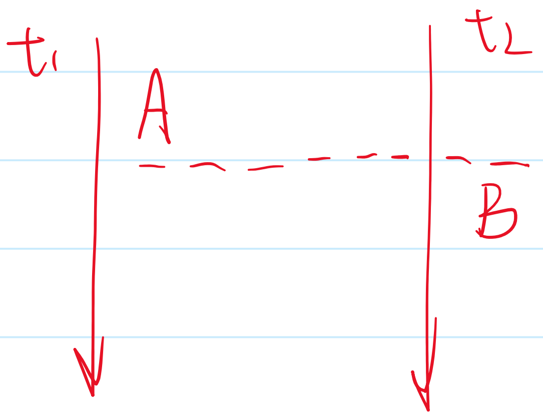
死循环阻止t2继续B.

浪费cpu资源

条件变量 condition variable.

2023年8月17日 15:01

无竞争的同步机制：等待－唤醒
wait signal



flag：条件.

```
A();
lock();
flag → 1
signal();
unlock();
```

```
lock();
if(flag != 1){
    wait()      等 lock解锁
                醒来 加锁.
}
unlock
B();
```

条件变量的接口

2023年8月17日 15:10

条件变量的数据类型

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

← 动态初始化

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

← 等待.

保护flag的.

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

← 唤醒.

2023年8月17日 15:31

```c
void *threadFunc(void *arg){
    sleep(10);
    shareRes_t * pShareRes = (shareRes_t *)arg;
    // 后事件的线程有机会wait --> 观察flag的值决定是否wait --> 用mutex加锁再观察flag
    pthread_mutex_lock(&pShareRes->mutex);
    if(pShareRes->flag != 1){
        //在等待期间，wait要把保护flag的锁解掉，其他线程才有机会修改flag
        pthread_cond_wait(&pShareRes->cond,&pShareRes->mutex);
        //在醒来会wait会把锁加回来
    }
    pthread_mutex_unlock(&pShareRes->mutex);
    B();
};
int main(int argc, char *argv[])
{
    shareRes_t shareRes;
    shareRes.flag = 0;
    pthread_mutex_init(&shareRes.mutex,NULL);
    pthread_cond_init(&shareRes.cond,NULL);
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,&shareRes);
    A();
    pthread_mutex_lock(&shareRes.mutex);
    shareRes.flag = 1;
    pthread_cond_signal(&shareRes.cond);
    pthread_mutex_unlock(&shareRes.mutex);

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

加锁
判断flag
wait

初始化

先事件 cond_signal

底层

2023年8月17日 15:51

cond 唤醒队列

pthread_cond_wait.

① 检查一下有没有mutex
② 把本线程移入队列
③ 解锁并陷入等待

原子操作

上半部

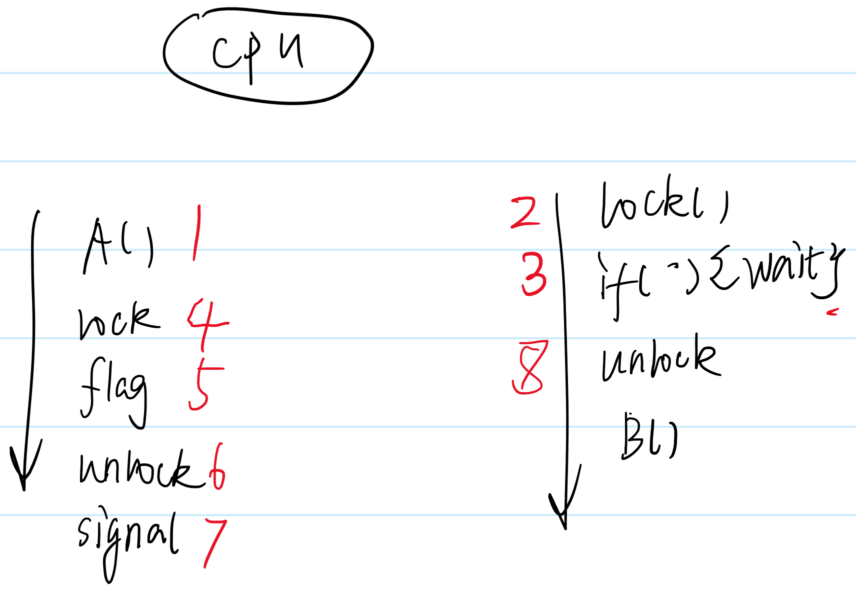
④ 被唤醒
⑤ 加锁.

下半部

⑥ 返回 后续代码带锁执行.

如果CPU这样切换

2023年8月17日 16:05



整理了使用条件变量的流程

2023年8月17日 16:10

① 准备共享数据. flag., mutex, cond

② a. 先事件. 将要 pthread_cond_signal.

事件 → 临界区中改flag → pthread_cond_signal

b. 后事件 将要 pthread_cond_wait

加锁 → 判断条件, 满足条件才 pthread_cond_wait

↳ 解锁 → 事件

火车票

2023年8月17日 16:18

20张.

window1

window2

就算是读共享资源,也要放入临界区

有问题的代码

切换

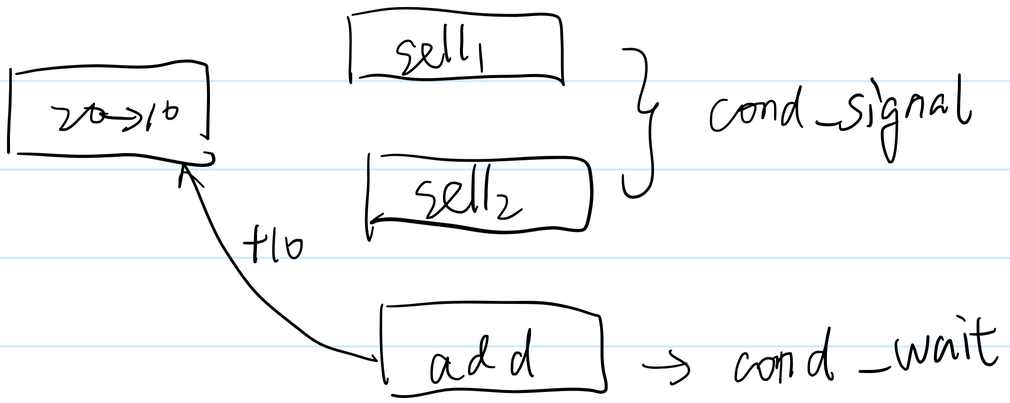
```
while(pShareRes->ticket > 0){
    pthread_mutex_lock(&pShareRes->mutex);
    printf("before window1, ticket = %d\n", pShareRes->ticket);
    --pShareRes->ticket;
    printf("after window1, ticket = %d\n", pShareRes->ticket);
    pthread_mutex_unlock(&pShareRes->mutex);
    //sleep(1);
}
```

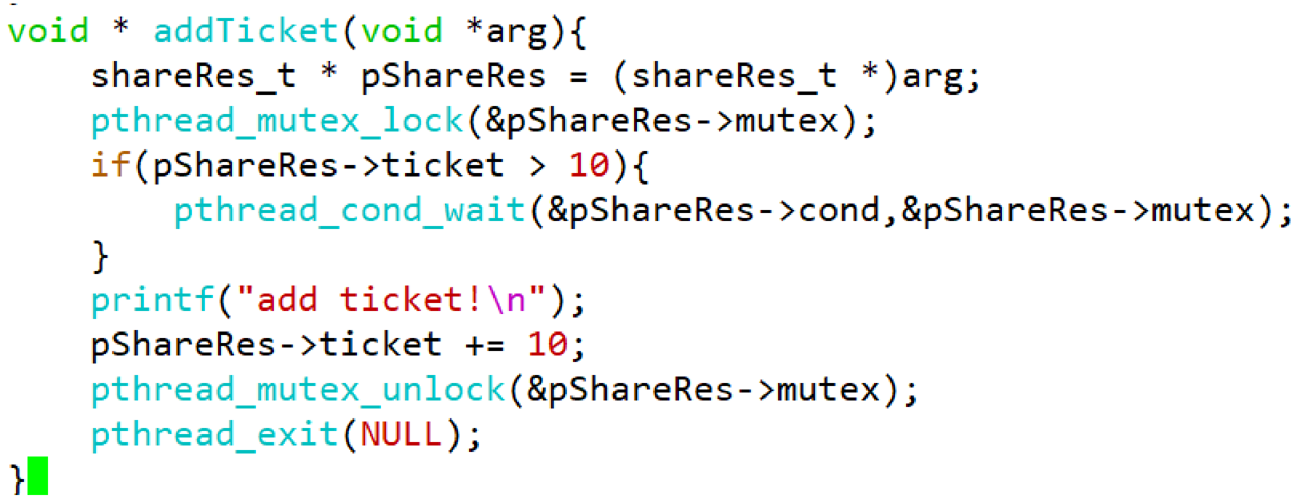
2023年8月17日　　16:34

```c
void * sell1(void *arg){
    shareRes_t * pShareRes = (shareRes_t *)arg;
    while(1){
        pthread_mutex_lock(&pShareRes->mutex);
        if(pShareRes->ticket <= 0){
            // 终止线程前记得解锁
            pthread_mutex_unlock(&pShareRes->mutex);
            break;
        }
        printf("before window1, ticket = %d\n", pShareRes->ticket);
        --pShareRes->ticket;
        printf("after window1, ticket = %d\n", pShareRes->ticket);
        pthread_mutex_unlock(&pShareRes->mutex);
        //sleep(1);
    }
    pthread_exit(NULL);
}
```

火车票

2023年8月17日　　16:34





```
void * addTicket(void *arg){
    shareRes_t * pShareRes = (shareRes_t *)arg;
    pthread_mutex_lock(&pShareRes->mutex);
    if(pShareRes->ticket > 10){
        pthread_cond_wait(&pShareRes->cond,&pShareRes->mutex);
    }
    printf("add ticket!\n");
    pShareRes->ticket += 10;
    pthread_mutex_unlock(&pShareRes->mutex);
    pthread_exit(NULL);
}
```

pthread_cond_timedwait

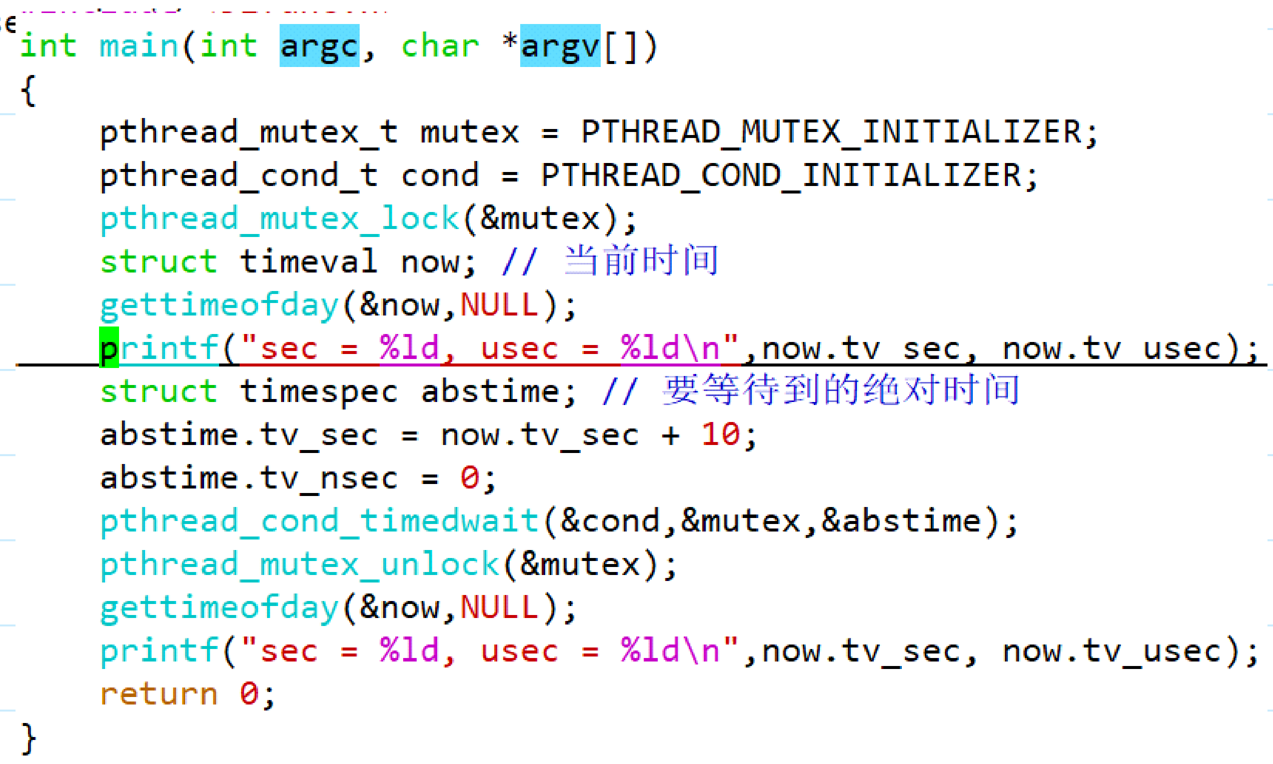
2023年8月17日 17:08

一般不用来实现同步，高精度定时.

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

绝对时间

```
struct timespec {
    time_t tv_sec;        /* seconds */
    long   tv_nsec;       /* nanose
};
```



```
int main(int argc, char *argv[])
{
    pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
    pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
    pthread_mutex_lock(&mutex);
    struct timeval now; // 当前时间
    gettimeofday(&now,NULL);
    printf("sec = %ld, usec = %ld\n",now.tv_sec, now.tv_usec);
    struct timespec abstime; // 要等待到的绝对时间
    abstime.tv_sec = now.tv_sec + 10;
    abstime.tv_nsec = 0;
    pthread_cond_timedwait(&cond,&mutex,&abstime);
    pthread_mutex_unlock(&mutex);
    gettimeofday(&now,NULL);
    printf("sec = %ld, usec = %ld\n",now.tv_sec, now.tv_usec);
    return 0;
}
```

广播

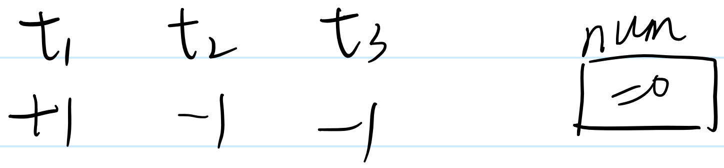
2023年8月17日 17:18

唤醒队列中的所有线程,

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

多个线程 可以同时因为 pthread_cond_wait, 陷入等待

唤醒所有线程不会引发竞争条件

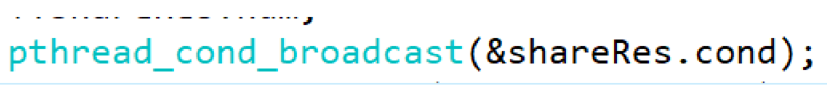


使用广播的场景

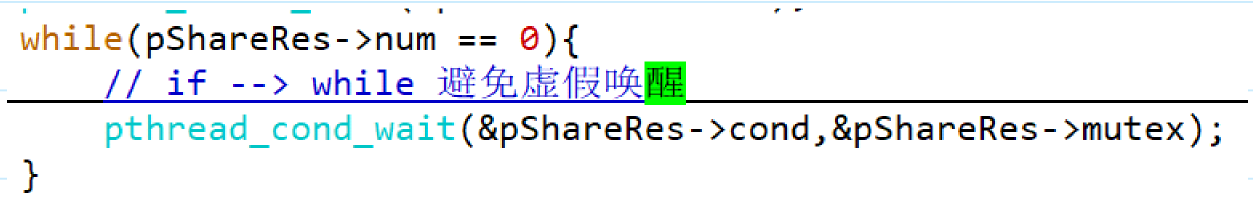
2023年8月17日　　17:42

多个线程 因为不同的原因, 等待在同一个条件变量中。

某一种资源增加 ——> 唤醒所有线程 ——> 线程醒来后重新检查条件.



```c
pthread_cond_broadcast(&shareRes.cond);
```



```c
while(pShareRes->num == 0){
    // if --> while 避免虚假唤醒
    pthread_cond_wait(&pShareRes->cond,&pShareRes->mutex);
}
```

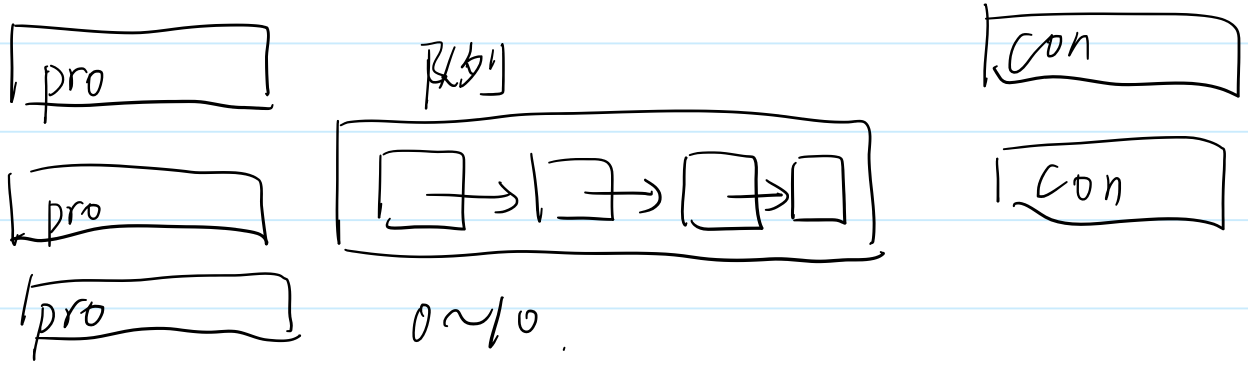
2023年8月17日 17:47

```c
int main(int argc, char *argv[])
{
    shareRes_t shareRes;
    shareRes.num = 0;
    pthread_mutex_init(&shareRes.mutex,NULL);
    pthread_cond_init(&shareRes.cond,NULL);

    pthread_t tid1,tid2;
    pthread_create(&tid1,NULL,threadFunc,&shareRes);
    pthread_create(&tid2,NULL,threadFunc,&shareRes);

    sleep(1);
    pthread_mutex_lock(&shareRes.mutex);
    ++shareRes.num;
    pthread_cond_broadcast(&shareRes.cond);
    pthread_mutex_unlock(&shareRes.mutex);

    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    return 0;
}
```

```c
void *threadFunc(void *arg){
    shareRes_t * pShareRes = (shareRes_t *)arg;
    pthread_mutex_lock(&pShareRes->mutex);
    while(pShareRes->num == 0){
        // if --> while 避免虚假唤醒
        pthread_cond_wait(&pShareRes->cond,&pShareRes->mutex);
    }
    printf("before num = %d\n", pShareRes->num);
    --pShareRes->num;
    printf("after num = %d\n", pShareRes->num);
    pthread_mutex_unlock(&pShareRes->mutex);
    pthread_exit(NULL);
}
```

生产者 消费者

2023年8月17日 17:50



线程的属性

2023年8月17日　　17:56

```
pthread_attr_t *attr,

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

PTHREAD_CREATE_DETACHED
       Threads that are

PTHREAD_CREATE_JOINABLE
```

```
void *threadFunc(void *arg){
    pthread_exit(NULL);
}
int main(int argc, char *argv[])
{
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    pthread_t tid;
    //pthread_create(&tid,&attr,threadFunc,NULL);
    pthread_create(&tid,NULL,threadFunc,NULL);

    int ret = pthread_join(tid,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_join");
    return 0;
}
```