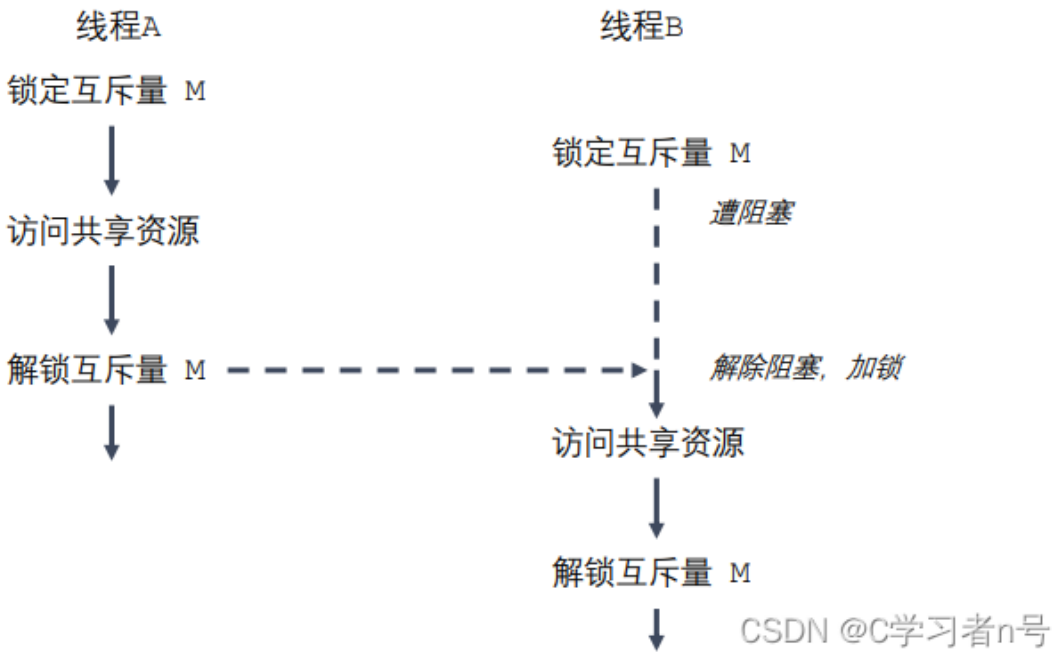
# 线程同步问题

## 概述

■ 线程的主要优势在于，能够通过全局变量来共享信息。不过，这种便捷的共享是有代价的：必须确保多个线程不会同时修改同一变量，或者某一线程不会读取正在由其他线程修改的变量。
■ 临界区是指访问某一共享资源的代码片段，并且这段代码的执行应为原子操作，也就是同时访问同一共享资源的其他线程不应终端该片段的执行。
■ 线程同步：即当有一个线程在对内存进行操作时，其他线程都不可以对这个内存地址进行操作，直到该线程完成操作，其他线程才能对该内存地址进行操作，而其他线程则处于等待状态。

## 互斥量（锁）

■ 如果多个线程试图执行这一块代码（一个临界区），事实上只有一个线程能够持有该互斥量（其他线程将遭到阻塞），即同时只有一个线程能够进入这段代码区域，如下图所示：




## 互斥量的相关操作函数

```
■ 互斥量的类型 pthread_mutex_t
/**
    在使用锁之前需要对锁进行初始化，使用完之后要销毁。即phtead_mutex_init和destroy函数
    pthread_mutex_lock ->加锁
    pthread_mutex_trylock ->尝试加锁
    pthread_mutex_unlock ->解锁
    在加锁和解锁的函数之间的代码则是临界区，对共享数据进行操作的代码
*/
■ int pthread_mutex_init(pthread_mutex_t *restrict mutex, const
pthread_mutexattr_t *restrict attr);
■ int pthread_mutex_destroy(pthread_mutex_t *mutex);

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

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

## 实例1 未加锁情况

```c
/*
    使用多线程实现买票的案例。
    有3个窗口，一共是100张票。
*/

#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

// 全局变量，所有的线程都共享这一份资源。
int tickets = 100;

void * sellticket(void * arg) {
    // 卖票
    while(tickets > 0) {
        usleep(6000);
        printf("%ld 正在卖第 %d 张门票\n", pthread_self(), tickets);
        tickets--;
    }
    return NULL;
}

int main() {

    // 创建3个子线程
    pthread_t tid1, tid2, tid3;
    pthread_create(&tid1, NULL, sellticket, NULL);
    pthread_create(&tid2, NULL, sellticket, NULL);
    pthread_create(&tid3, NULL, sellticket, NULL);

    // 回收子线程的资源,阻塞
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_join(tid3, NULL);

    // 设置线程分离。
    // pthread_detach(tid1);
    // pthread_detach(tid2);
    // pthread_detach(tid3);

    pthread_exit(NULL); // 退出主线程

    return 0;
}
```

## 实例2 加锁

```
/*
    互斥量的类型 pthread_mutex_t
    int pthread_mutex_init(pthread_mutex_t *restrict mutex, const
pthread_mutexattr_t *restrict attr);
        - 初始化互斥量
        - 参数 :
            - mutex :  需要初始化的互斥量变量
            - attr :  互斥量相关的属性，NULL
        - restrict : C语言的修饰符，被修饰的指针，不能由另外的一个指针进行操作。
            pthread_mutex_t *restrict mutex = xxx;
            pthread_mutex_t * mutex1 = mutex;

    int pthread_mutex_destroy(pthread_mutex_t *mutex);
        - 释放互斥量的资源

    int pthread_mutex_lock(pthread_mutex_t *mutex);
        - 加锁，阻塞的，如果有一个线程加锁了，那么其他的线程只能阻塞等待

    int pthread_mutex_trylock(pthread_mutex_t *mutex);
        - 尝试加锁，如果加锁失败，不会阻塞，会直接返回。

    int pthread_mutex_unlock(pthread_mutex_t *mutex);
        - 解锁
*/
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

// 全局变量，所有的线程都共享这一份资源。
int tickets = 1000;

// 创建一个互斥量
pthread_mutex_t mutex;

void * selltticket(void * arg) {
    // 卖票
    while(1) {
        // 加锁
        pthread_mutex_lock(&mutex);

        if(tickets > 0) {
            usleep(6000);
            printf("%ld 正在卖第 %d 张门票\n", pthread_self(), tickets);
            tickets--;
        }else {
            // 解锁
            pthread_mutex_unlock(&mutex);
            break;
        }

        // 解锁
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

```c
int main() {

    // 初始化互斥量
    pthread_mutex_init(&mutex, NULL);

    // 创建3个子线程
    pthread_t tid1, tid2, tid3;
    pthread_create(&tid1, NULL, sellticket, NULL);
    pthread_create(&tid2, NULL, sellticket, NULL);
    pthread_create(&tid3, NULL, sellticket, NULL);

    // 回收子线程的资源,阻塞
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_join(tid3, NULL);

    pthread_exit(NULL); // 退出主线程

    // 释放互斥量资源
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

# 死锁问题

## 概述

■ 有时，一个线程需要同时访问两个或更多不同的共享资源，而每个资源又都由不同的互斥量管理。当超过一个线程加锁同一组互斥量时，就有可能发生死锁。
■ 两个或两个以上的进程在执行过程中，因争夺共享资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁。
■ 死锁的几种场景：
1、忘记释放锁。最直接的产生方式，加了锁之后没有释放，即使是当前加锁的线程再次执行也无法获取这个锁。
2、重复加锁。就是一个线程加了一个x锁，但是在没有释放之前又加上了这个锁，但是这个x锁正在被当前线程占用，所以不管是谁想要加上这个锁都会失败。
3、 多线程多锁，抢占锁资源。正常情况下，每一个共享资源都会有一个锁进行同步控制，当有多个线程且对多个不同的共享资源进行操作，此时，多个线程之间可能就会抢占锁。比如，一个x锁被一个线程抢占了，继续执行时，又需要加另外一个y锁，但是在此之前y锁被另一个线程占用了，这个线程继续执行时又需要x锁，此时，就会造成死锁，线程之间相互等待。

## 实例 忘记释放锁

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

// 全局变量，所有的线程都共享这一份资源。
int tickets = 1000;

// 创建一个互斥量
pthread_mutex_t mutex;
```

```c
void * sellticket(void * arg) {

    // 卖票
    while(1) {

        // 加锁
        pthread_mutex_lock(&mutex);
        if(tickets > 0) {
            usleep(6000);
            printf("%ld 正在卖第 %d 张门票\n", pthread_self(), tickets);
            tickets--;
        }else {
            // 解锁
            pthread_mutex_unlock(&mutex);
            break;
        }
        //此处未释放锁，当其他线程执行时拿不到锁
    }

    return NULL;
}

int main() {

    // 初始化互斥量
    pthread_mutex_init(&mutex, NULL);

    // 创建3个子线程
    pthread_t tid1, tid2, tid3;
    pthread_create(&tid1, NULL, sellticket, NULL);
    pthread_create(&tid2, NULL, sellticket, NULL);
    pthread_create(&tid3, NULL, sellticket, NULL);

    // 回收子线程的资源,阻塞
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_join(tid3, NULL);

    pthread_exit(NULL); // 退出主线程

    // 释放互斥量资源
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

## 实例 重复加锁

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

// 全局变量，所有的线程都共享这一份资源。
int tickets = 1000;

// 创建一个互斥量
pthread_mutex_t mutex;
```

```c
void * sellticket(void * arg) {

    // 卖票
    while(1) {

        // 加锁
        pthread_mutex_lock(&mutex);
        pthread_mutex_lock(&mutex);

        if(tickets > 0) {
            usleep(6000);
            printf("%ld 正在卖第 %d 张门票\n", pthread_self(), tickets);
            tickets--;
        }else {
            // 解锁
            pthread_mutex_unlock(&mutex);
            break;
        }

        // 解锁
        pthread_mutex_unlock(&mutex);
        pthread_mutex_unlock(&mutex);
    }

    return NULL;
}

int main() {

    // 初始化互斥量
    pthread_mutex_init(&mutex, NULL);

    // 创建3个子线程
    pthread_t tid1, tid2, tid3;
    pthread_create(&tid1, NULL, sellticket, NULL);
    pthread_create(&tid2, NULL, sellticket, NULL);
    pthread_create(&tid3, NULL, sellticket, NULL);

    // 回收子线程的资源,阻塞
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_join(tid3, NULL);

    pthread_exit(NULL); // 退出主线程

    // 释放互斥量资源
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

## 实例 多线程多锁

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

// 创建2个互斥量
pthread_mutex_t mutex1, mutex2;

void * workA(void * arg) {

    pthread_mutex_lock(&mutex1);
    sleep(1);
    pthread_mutex_lock(&mutex2);

    printf("workA....\n");

    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    return NULL;
}


void * workB(void * arg) {
    pthread_mutex_lock(&mutex2);
    sleep(1);
    pthread_mutex_lock(&mutex1);

    printf("workB....\n");

    pthread_mutex_unlock(&mutex1);
    pthread_mutex_unlock(&mutex2);

    return NULL;
}

int main() {

    // 初始化互斥量
    pthread_mutex_init(&mutex1, NULL);
    pthread_mutex_init(&mutex2, NULL);

    // 创建2个子线程
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, workA, NULL);
    pthread_create(&tid2, NULL, workB, NULL);

    // 回收子线程资源
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    // 释放互斥量资源
    pthread_mutex_destroy(&mutex1);
    pthread_mutex_destroy(&mutex2);

    return 0;
}
```

# 读写锁

## 概述

■ 当有一个线程已经持有互斥锁时，互斥锁将所有试图进入临界区的线程都阻塞住。但是考虑一种情形，当前持有互斥锁的线程只是要读访问共享资源，而同时有其它几个线程也想读取这个共享资源，但是由于互斥锁的排它性，所有其它线程都无法获取锁，也就无法读访问共享资源了，但是实际上多个线程同时读访问共享资源并不会导致问题。
■ 在对数据的读写操作中，更多的是读操作，写操作较少，例如对数据库数据的读写应用。为了满足当前能够允许多个读出，但只允许一个写入的需求，线程提供了读写锁来实现。
■ 读写锁的特点：
1、如果有线程读数据，则允许其它线程执行读操作，但不允许写操作。
2、如果有线程写数据，则其它线程都不允许读、写操作。
3、写是独占的，写的优先级高。

## 读写锁API

```
■ 读写锁的类型 pthread_rwlock_t
/**
    读写锁也需要初始化和销毁。
    rdlock和tryrdlock表示加读锁和尝试加读锁;
    wrlock和trywrlock表示加写锁和尝试加写锁
*/
■ int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,const
pthread_rwlockattr_t *restrict attr);
■ int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

■ int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
■ int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
■ int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
■ int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);

■ int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

## 实例

说明：8个线程操作同一个全局变量。3个线程不定时写这个全局变量，5个线程不定时的读这个全局变量。

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

// 创建一个共享数据
int num = 1;
// pthread_mutex_t mutex;
pthread_rwlock_t rwlock;

void * writeNum(void * arg) {

    while(1) {
        pthread_rwlock_wrlock(&rwlock);
        num++;
        printf("++write, tid : %ld, num : %d\n", pthread_self(), num);
```

```
        pthread_rwlock_unlock(&rwlock);
        usleep(100);
    }

    return NULL;
}

void * readNum(void * arg) {

    while(1) {
        pthread_rwlock_rdlock(&rwlock);
        printf("===read, tid : %ld, num : %d\n", pthread_self(), num);
        pthread_rwlock_unlock(&rwlock);
        usleep(100);
    }

    return NULL;
}

int main() {

   pthread_rwlock_init(&rwlock, NULL);

    // 创建3个写线程，5个读线程
    pthread_t wtids[3], rtids[5];
    for(int i = 0; i < 3; i++) {
        pthread_create(&wtids[i], NULL, writeNum, NULL);
    }

    for(int i = 0; i < 5; i++) {
        pthread_create(&rtids[i], NULL, readNum, NULL);
    }

    // 设置线程分离
    for(int i = 0; i < 3; i++) {
       pthread_detach(wtids[i]);
    }

    for(int i = 0; i < 5; i++) {
         pthread_detach(rtids[i]);
    }

    pthread_exit(NULL);

    pthread_rwlock_destroy(&rwlock);

    return 0;
}
```

## 实例

使用多进程来比较互斥锁和读写锁的性能比较

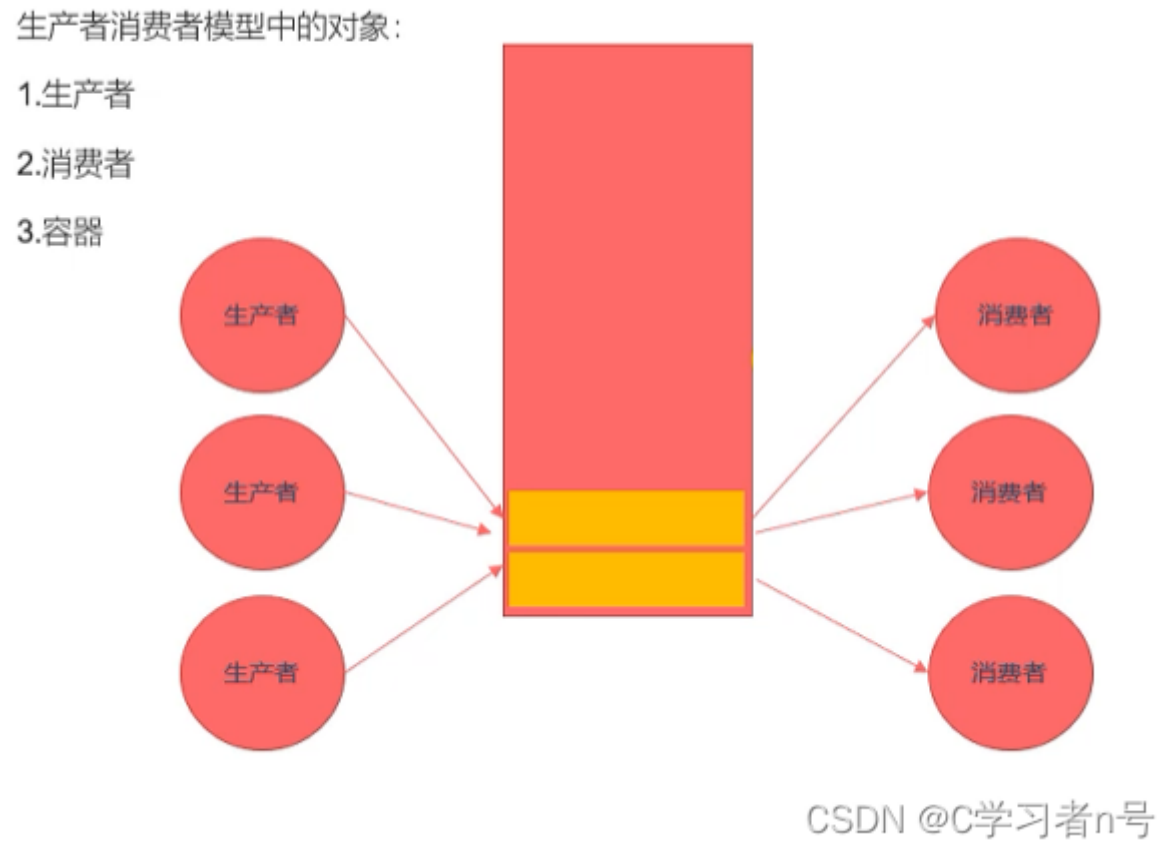
```
/*
    此程序是为了比较
    互斥锁和读写锁的在 高并发读操作上的效率差异
    创建两个进程，父进程中使用互斥锁读取，子进程使用读写锁读取
```

```
    每个进程中创建10个读线程，对全局变量区的某个变量进行读取
    每个线程读取 COUNT 次，总计完成10*COUNT次读取
    利用 gettimeofday 函数，对整个读取过程精确到微秒的计时
*/
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <time.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <stdlib.h>

//创建互斥锁
pthread_mutex_t mutex;
//创建读写锁
pthread_rwlock_t rwlock;
//全局变量,父子进程、线程间都是相同的
int data = 123456;
//读取次数
long COUNT = 10000000;

//互斥读函数
void *mFunc(void *arg)
{
    int read = 0;
    for (long i = 0; i < COUNT; ++i)
    {
        //加锁
        pthread_mutex_lock(&mutex);
        read = data;
        //解锁
        pthread_mutex_unlock(&mutex);
    }
    //结束线程
    pthread_exit(NULL);
}

//读写锁读
void *rwFunc(void *arg)
{
    int read = 0;
    for (long i = 0; i < COUNT; ++i)
    {
        //加锁
        pthread_rwlock_rdlock(&rwlock);
        read = data;
        //解锁
        pthread_rwlock_unlock(&rwlock);
    }
    //结束线程
    pthread_exit(NULL);
}
int main()
{
    //创建两个进程，父进程中使用互斥锁读取，子进程使用读写锁读取
    int pid = fork();
```

```
    if (pid > 0)
    {
        //parent process
        //初始化互斥锁
        pthread_mutex_init(&mutex, NULL);
        //创建十个线程，并开始计时
        pthread_t mtids[10];
        struct timeval start;
        gettimeofday(&start, NULL);
        for (int i = 0; i < 10; ++i)
        {
            pthread_create(&mtids[i], NULL, mFunc, NULL);
        }
        //在主线程中，调用join函数，回收线程，线程回收完成后，结束计时
        for (int i = 0; i < 10; ++i)
        {
            pthread_join(mtids[i], NULL);
        }
        struct timeval end;
        gettimeofday(&end, NULL);
        long timediff = (e***_sec - start.tv_sec) * 1000000 + e***_usec -
start.tv_usec;
        printf("互斥锁 读全部线程执行完毕，总耗时: %ld us\n", timediff);

        //回收子进程
        wait(NULL);
    }
    else if (pid == 0)
    {
        //子进程
        //初始化读写锁
        pthread_rwlock_init(&rwlock, NULL);
        //创建
        //创建十个线程，并开始计时
        pthread_t rwtids[10];
        struct timeval start;
        gettimeofday(&start, NULL);
        for (int i = 0; i < 10; ++i)
        {
            pthread_create(&rwtids[i], NULL, rwFunc, NULL);
        }
        //在主线程中，调用join函数，回收线程，线程回收完成后，结束计时
        for (int i = 0; i < 10; ++i)
        {
            pthread_join(rwtids[i], NULL);
        }
        struct timeval end;
        gettimeofday(&end, NULL);
        long timediff = (e***_sec - start.tv_sec) * 1000000 + e***_usec -
start.tv_usec;
        printf("读写锁 读全部线程执行完毕，总耗时: %ld us\n", timediff);
        //结束进程
        exit(0);
    }
    return 0;
}
```

# 生产者消费者模型

## 演示




## 代码

```c
/*
    生产者消费者模型（粗略的版本）
*/
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

// 创建一个互斥量
pthread_mutex_t mutex;

//相当于一个产品
struct Node{
    int num;
    struct Node *next;
};

// 头结点。代码中使用头插法的方式添加产品
struct Node * head = NULL;

void * producer(void * arg) {

    // 不断的创建新的节点，添加到链表中
    while(1) {
        pthread_mutex_lock(&mutex);
```

```
        struct Node * newNode = (struct Node *)malloc(sizeof(struct Node));
        newNode->next = head;
        head = newNode;
        newNode->num = rand() % 1000;
        printf("add node, num : %d, tid : %ld\n", newNode->num, pthread_self());

        pthread_mutex_unlock(&mutex);
        usleep(100);
    }

    return NULL;
}

void * customer(void * arg) {

    while(1) {
        pthread_mutex_lock(&mutex);
        // 保存头结点的指针
        struct Node * tmp = head;

        // 判断是否有数据
        if(head != NULL) {
            // 有数据
            head = head->next;
            printf("del node, num : %d, tid : %ld\n", tmp->num, pthread_self());
            free(tmp);
            pthread_mutex_unlock(&mutex);
            usleep(100);
        } else {
            // 没有数据
            pthread_mutex_unlock(&mutex);
        }
    }
    return  NULL;
}

int main() {

    pthread_mutex_init(&mutex, NULL);

    // 创建5个生产者线程，和5个消费者线程
    pthread_t ptids[5], ctids[5];

    for(int i = 0; i < 5; i++) {
        pthread_create(&ptids[i], NULL, producer, NULL);
        pthread_create(&ctids[i], NULL, customer, NULL);
    }

    for(int i = 0; i < 5; i++) {
        pthread_detach(ptids[i]);
        pthread_detach(ctids[i]);
    }

    while(1) {  //为了后面释放锁
        sleep(10);
    }
    //释放锁
```

```
    pthread_mutex_destroy(&mutex);

    pthread_exit(NULL);

    return 0;
}
```

# 条件变量

## 概述

1、条件变量起到一种通知唤醒的作用。上面的生产者消费者代码虽然没有错，但是生产者线程与消费者线程还是会轮流切换。如果此时没有没有产品，那么多次切换到消费者线程无疑是没有意义的。
2、条件变量的作用就是如果切换到了消费者线程且当时没有任何产品消费，让这个线程阻塞等待（wait），且会暂时解开这个锁，以便其他线程能够继续执行，当前线程陷入等待中；而生产者线程每生产一个产品就会发送一个signal通知生成了一个产品，此时陷入等待中的线程就会被唤醒，并且加上之前的锁，继续执行下面的代码。
3、条件变量无法解决同步问题，所以要保证数据同步还是要用锁。通常条件变量和互斥锁是混合使用的。

## 相关API

```
■ 条件变量的类型 pthread_cond_t

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

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

## 代码

```
/*
    条件变量的类型 pthread_cond_t

    int pthread_cond_init(pthread_cond_t *cond,pthread_condattr_t *cond_attr);
        功能：初始化一个条件变量
        参数：
            cond：指定要初始化的条件变量（指向结构pthread_cond_t的指针）
            cond_attr：NULL 默认的（用于设置条件变量是进程内还是进程间的）
        返回值：0 成功
                非0 错误
    int pthread_cond_destroy(pthread_cond_t *cond);
        功能：销毁一个条件变量
        参数：
            cound：指定要销毁的条件变量
        返回值：0 成功
                非0 错误
    int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);  //等待期
间不会占用cpu，先解锁，然后进入睡眠状态，等待接受信号
```

```
        功能：等待条件变量为真（无条件等待）
        参数：
            cond：指定等待的条件变量
            mutex：等待之前需要解开的锁
        返回值：0 成功
                非0 错误
    int   pthread_cond_timedwait(pthread_cond_t *cond,pthread_mutex_t *mutex,
const struct timespec *abstime); //超时等待
        功能：超时等待，超时返回错误（计时等待）
        参数：
            cond：指定等待的条件变量
            mutex：等待之前需要解开的锁
            abstime：指定等待的时间
        返回值：0 成功
                非0 错误

    int pthread_cond_signal(pthread_cond_t *cond);
        功能：启动在等待条件变量变为真的一个线程，每次最多可以给一个线程发送
        参数：
            cond：指定条件变量
        返回值：0 成功
                非0 错误
    int pthread_cond_broadcast(pthread_cond_t *cond);
        功能：启动所有的等待条件变量为真的线程
        参数：
          cond：指定条件变量
        返回值：0 成功
                非0 错误
*/

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

// 创建一个互斥量
pthread_mutex_t mutex;
// 创建条件变量
pthread_cond_t cond;

struct Node{
    int num;
    struct Node *next;
};

// 头结点
struct Node * head = NULL;

void * producer(void * arg) {

    // 不断的创建新的节点，添加到链表中
    while(1) {
        pthread_mutex_lock(&mutex);
        struct Node * newNode = (struct Node *)malloc(sizeof(struct Node));
        newNode->next = head;
        head = newNode;
        newNode->num = rand() % 1000;
        printf("add node, num : %d, tid : %ld\n", newNode->num, pthread_self());
```

```
        // 只要生产了一个，就通知消费者消费
        pthread_cond_signal(&cond);

        pthread_mutex_unlock(&mutex);
        usleep(100);
    }

    return NULL;
}

void * customer(void * arg) {

    while(1) {
        pthread_mutex_lock(&mutex);
        // 保存头结点的指针
        struct Node * tmp = head;
        // 判断是否有数据
        if(head != NULL) {
            // 有数据
            head = head->next;
            printf("del node, num : %d, tid : %ld\n", tmp->num, pthread_self());
            free(tmp);
            pthread_mutex_unlock(&mutex);
            usleep(100);
        } else {
            // 没有数据，需要等待
            // 当这个函数调用阻塞的时候，会对互斥锁进行解锁，当不阻塞的，继续向下执行，会重新加锁。
            pthread_cond_wait(&cond, &mutex);
            pthread_mutex_unlock(&mutex);
        }
    }
    return  NULL;
}

int main() {

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

    // 创建5个生产者线程，和5个消费者线程
    pthread_t ptids[5], ctids[5];

    for(int i = 0; i < 5; i++) {
        pthread_create(&ptids[i], NULL, producer, NULL);
        pthread_create(&ctids[i], NULL, customer, NULL);
    }

    for(int i = 0; i < 5; i++) {
        pthread_detach(ptids[i]);
        pthread_detach(ctids[i]);
    }

    while(1) {
        sleep(10);
    }
```

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

    pthread_exit(NULL);

    return 0;
}
```

# 信号量

## 概述

1、和互斥锁类似，信号量本质也是一个全局变量。不同之处在于，互斥锁的值只有 2 个（加锁 "lock" 和解锁 "unlock"），而信号量的值可以根据实际场景的需要自行设置（取值范围为 ≥0）。更重要的是，信号量还支持做“加 1”或者 “减 1”运算，且修改值的过程以“原子操作”的方式实现。
2、多线程程序中，使用信号量需遵守以下几条规则：
a、信号量的值不能小于 0；
b、有线程访问资源时，信号量执行“减 1”操作，访问完成后再执行“加 1”操作；
c、当信号量的值为 0 时，想访问资源的线程必须等待，直至信号量的值大于 0，等待的线程才能开始访问。
3、原子操作
原子操作是指当多个线程试图修改同一个信号量的值时，各线程修改值的过程不会互相干扰。例如信号量的初始值为 1，此时有 2 个线程试图对信号量做“加 1”操作，则信号量的值最终一定是 3，而不会是其它的值。反之若不以“原子操作”方式修改信号量的值，那么最终的计算结果还可能是 2（两个线程同时读取到的值为 1，各自在其基础上加 1，得到的结果即为 2）。

```
■ 信号量的类型 sem_t
/**
    int sem_init(sem_t *sem, int pshared, unsigned int value);
    各个参数的含义分别为：
    sem：表示要初始化的目标信号量；
    pshared：表示该信号量是否可以和其他进程共享，pshared 值为 0 时表示线程间，非0是进程间
    value：设置信号量的初始值。
    当 sem_init() 成功完成初始化操作时，返回值为 0，否则返回 -1。
*/
/**
    sem_post() 函数的功能是：将信号量的值“加 1”，同时唤醒其它等待访问资源的线程；
    当信号量的值大于 0 时，sem_wait() 函数会对信号量做“减 1”操作；当信号量的值为 0 时，
sem_wait() 函数会阻塞当前线程，直至有线程执行 sem_post() 函数（使信号量的值大于 0），暂停的
线程才会继续执行；
    sem_trywait() 函数的功能和 sem_wait() 函数类似，唯一的不同在于，当信号量的值为 0 时，
sem_trywait() 函数并不会阻塞当前线程，而是立即返回 -1；
    sem_destory() 函数用于手动销毁信号量。
*/
■ int sem_init(sem_t *sem, int pshared, unsigned int value);
■ int sem_destroy(sem_t *sem);
■ int sem_wait(sem_t *sem);
■ int sem_trywait(sem_t *sem);
■ int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
■ int sem_post(sem_t *sem);
■ int sem_getvalue(sem_t *sem, int *sval);
```

## 代码

```
/*
    信号量的类型 sem_t
    int sem_init(sem_t *sem, int pshared, unsigned int value);
        - 初始化信号量
        - 参数:
            - sem : 信号量变量的地址
            - pshared : 0 用在线程间 ，非0 用在进程间
            - value : 信号量中的值

    int sem_destroy(sem_t *sem);
        - 释放资源

    int sem_wait(sem_t *sem);
        - 对信号量加锁，调用一次对信号量的值-1，如果值为0，就阻塞

    int sem_trywait(sem_t *sem);

    int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
    int sem_post(sem_t *sem);
        - 对信号量解锁，调用一次对信号量的值+1

    int sem_getvalue(sem_t *sem, int *sval);

    sem_t psem;
    sem_t csem;
    init(psem, 0, 8);
    init(csem, 0, 0);

    producer() {
        sem_wait(&psem);
        sem_post(&csem)
    }

    customer() {
        sem_wait(&csem);
        sem_post(&psem)
    }
以下代码执行的一种情景：
多个消费者线程先与生产者执行，此时执行到sem_wait(&csem)，由于csem的初始值为0，所以所有线程都处于阻塞等待状态；之后生产者线程被执行，首先把执行sem_wait(&psem)，把信号量-1，（初始值为0），然后某一个生产者线程拿到了一把锁，那么它就能继续往下执行业务，执行完业务释放锁资源。如果接下来执行sem_post(&csem)将csem信号量+1，相当于通知消费者线程中阻塞在sem_wait(&csem)处的线程可以执行了，但是只有一个线程能够去执行业务，因为接下来要看哪个消费者线程拿到了锁资源。

*/

#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>

// 创建一个互斥量
pthread_mutex_t mutex;
// 创建两个信号量
```

```c
sem_t psem;
sem_t csem;

struct Node{
    int num;
    struct Node *next;
};

// 头结点
struct Node * head = NULL;

void * producer(void * arg) {

    // 不断的创建新的节点，添加到链表中
    while(1) {
        sem_wait(&psem);    //psem信号量-1操作
        pthread_mutex_lock(&mutex);
        struct Node * newNode = (struct Node *)malloc(sizeof(struct Node));
        newNode->next = head;
        head = newNode;
        newNode->num = rand() % 1000;
        printf("add node, num : %d, tid : %ld\n", newNode->num, pthread_self());
        pthread_mutex_unlock(&mutex);
        sem_post(&csem);    //csem信号量+1操作
    }

    return NULL;
}

void * customer(void * arg) {

    while(1) {
        sem_wait(&csem);    //csem信号量-1操作，若是当前csem == 0 则等待
sem_pos(&csem)的操作发送信号+1
        pthread_mutex_lock(&mutex);
        // 保存头结点的指针
        struct Node * tmp = head;
        head = head->next;
        printf("del node, num : %d, tid : %ld\n", tmp->num, pthread_self());
        free(tmp);
        pthread_mutex_unlock(&mutex);
        sem_post(&psem);    //psem信号量-1

    }
    return  NULL;
}

int main() {

    pthread_mutex_init(&mutex, NULL);
    sem_init(&psem, 0, 8);
    sem_init(&csem, 0, 0);

    // 创建5个生产者线程，和5个消费者线程
    pthread_t ptids[5], ctids[5];

    for(int i = 0; i < 5; i++) {
        pthread_create(&ptids[i], NULL, producer, NULL);
```

```
        pthread_create(&ctids[i], NULL, customer, NULL);
    }

    for(int i = 0; i < 5; i++) {
        pthread_detach(ptids[i]);
        pthread_detach(ctids[i]);
    }

    while(1) {
        sleep(10);
    }

    pthread_mutex_destroy(&mutex);

    pthread_exit(NULL);

    return 0;
}
```