线程同步

2024年6月24日　　13:18

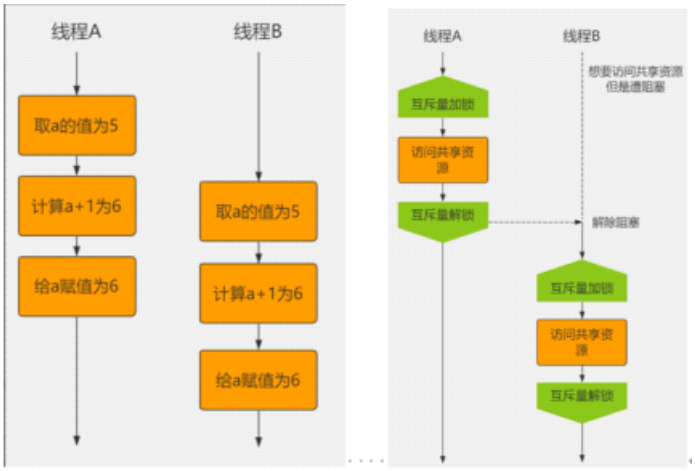
六、线程同步

子线程没有独立的地址空间，大部分数据都是共享的，如果同时访问数据，就会造成混乱，所以要进行控制，线程之间要协调好先后执行的顺序。**同步就是协同步调**，按预定的先后次序进行运行。如:你说完，我再说。**这里的同步千万不要理解成那个同时进行，应是指协同、协助、互相配合**。线程同步是指多线程通过特定的设置(如互斥量，条件变量等)来控制线程之间的执行顺序(即所谓的同步)也可以说是在线程之间通过同步建立起执行顺序的关系，如果没有同步，那线程之间是各自运行各自的!线程互斥是指对于共享的进程系统资源，在各单个线程访问时的排它性。当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。

6.1 互斥锁（互斥量）

线程的主要优势在于能够通过全局变量来共享信息。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由其他线程修改的变量

互斥量可以保护对共享变量的访问。



a++需要执行3个步骤:

1. 取a的值
2. 计算 a+1
3. a+1 赋值给 a

某一时刻，全局变量a的值为5，线程A和线程B中都要执行a++，如果线程A在执行a+1 赋值给a之前，这时线程B执行了a++，线程B取到a的值仍是5，这时，线程A和 B赋给a的值都是 6。值为5的变量a经过两次a++结果却是6，显然出现了错误。

多线程竞争操作共享变量的这段代码叫做临界区。多个进程同时操作临界区会产生错误所以这段代码应该互斥，当一个线程执行临界区时，应该阻止其他线程进入临界区。

为避免线程更新共享变量时出现问题，可以使用互斥量(mutex是mutualexclusion 的缩写)来**确保同时仅有一个线程可以访问某项共享资源。**

互斥量的作用类似于一个“锁”，当一个线程访问共享资源时，它必须先尝试获取互斥量的锁。如果互斥量当前没有被其他线程占用，那么该线程将成功获取锁，并可以安全地访问共享资源:如果互斥量已经被其他线程占用，那么该线程将被阻塞，直到互斥量的锁被放。

互斥量有两种状态:**已锁定(locked)**和**未锁定(unlocked)**。至多只有一个线程可以锁定该互斥量，试图对已经锁定的某一互斥量再次加锁将会阻塞线程。一旦线程锁定斥量，随即成为该互斥量的所有者，只有所有者才能给互斥量解锁。

pthread_mutexinit 函数

函数描述:

初始化一个互斥量

函数原型:

```
int_pthread_mutex init(pthread_mutex_t*mutex, const pthread_mutex_attr_t* mutexattr);
```

函数参数:

mutex:指向互斥量的指针，下面几个函数都有该参数，不一一介绍

mutexattr:指向定义互斥量属性的指针，取默认值传 NULL

函数返回值:

成功返回 0

失败返回错误号

pthread_mutex_lock和pthread_mutex_unlock函数

函数描述:

给互斥量加锁和解锁，解锁的函数在解锁的同时唤醒阻塞在该互斥量上的线程，默认先阻塞的先唤醒。

函数原型:

```
int pthread_mutex_lock(pthread_mutex_t* mutex),
int thread_mutex_unlock(pthread_mutex_t* mutex),
```

函数返回值:

成功返回 0

出现错误返回错误号。加锁不成功，线程阻塞

pthread_mutex destroy 函数

函数描述:

销毁一个互斥量

函数原型:

```
int pthread_mutex_destroy(pthread_mutex t*mutex);
```

函数返回值:

成功返回 0

失败返回错误号

pthread_mutextrylock函数

函数描述:

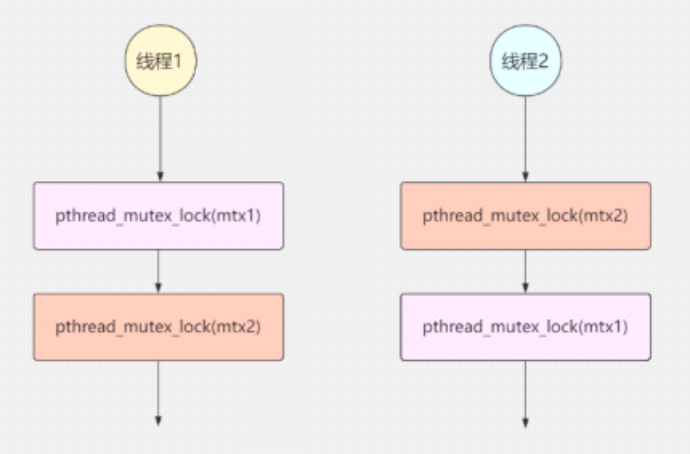
尝试给互斥量加锁，加锁不成功直接返回错误号(EBUSY)，不会阻塞，其他与pthread mutex lock相同。

```c
// 创建互斥量
pthread_mutex_t mtx;
int a = 0;
void* thread1(void* arg)
{
    printf("thread1 id = %lu\n",pthread_self());
    for(int i = 0; i < 10000000; i++)
    {
        pthread_mutex_lock(&mtx); // 添加锁
        a++;
        pthread_mutex_unlock(&mtx); // 解锁
    }
    return NULL;
}
int main(int argc, char* argv[])
{
```

```
    // 初始化互斥量
    pthread_mutex_init(&mtx,NULL);
    pthread_t mpd;
    pthread_t mpd_Arr[10];
    for(int i = 0; i < 10; i++)
    {
        pthread_create(&mpd_Arr[i],NULL,thread1,NULL);
    }
    // 连接线程，等待
    for(int i = 0; i < 10; i++)
    {
        pthread_join(mpd_Arr[i],NULL);
    }

    printf("main id = %lu\n",pthread_self());
    // sleep(10);
    printf("a = %d\n",a);


    pthread_mutex_destroy(&mtx); // 销毁锁
    return 0;
}
```

死锁现象

当多个线程中为了保护多个共享资源而使用了多个互斥锁，如果多个互斥锁使用不当，就可能造成，多个线程一直等待对方的锁释放，这就是死锁现象。



死锁产生的四个必要条件:

(1)互斥条件:资源只能同时被一个进程占用

(2)持有并等待条件:线程1已经持有资源 A，可以申请持有资源 B，如果资源B已经被

(3)线程 2 持有，这时线程1持有资源并等待资源 B不可剥夺条件:一个线程持有资源，只能自己释放后其他线程才能使用。其他线程不能强制收回该资源

(4)环路等待条件:多个线程互相等待资源，形成一个环形等待链

避免死锁:破坏其中一个必要条件就可以避免死锁，常用的方法如下:

(1)锁的粒度控制:破坏请求与保持条件，尽可能减少持有锁的时间，降低发生死锁的可能性

(2)资源有序分配:破坏环路等待条件，规定线程使用资源的顺序，按规定顺序给资源加锁

(3)重试机制:破坏不可剥夺条件，如果尝试获取资源失败，放弃已持有的资源后重试

```
pthread_mutex_t mtx1;
pthread_mutex_t mtx2;

void* thread1(void* arg)
{
    while (1)
    {
        printf("子线程获取 mtx1中。。。\n");
        pthread_mutex_lock(&mtx1);
        printf("子线程获取 mtx1成功! \n");
        sleep(1);
        printf("子线程获取 mtx2中。。。\n");
```

```
        int ret = pthread_mutex_trylock(&mtx2);
        if((ret != 0) && (ret = EBUSY))
        {
            printf("mtx2被占用，已成功释放！\n");
            pthread_mutex_unlock(&mtx2);
            sleep(1);
        }
        printf("子线程获取 mtx2成功！\n");
        pthread_mutex_unlock(&mtx2);
        pthread_mutex_unlock(&mtx1);
        return NULL;
    }

}
int main(int argc, char* argv[])
{
    pthread_mutex_init(&mtx1,NULL);
    pthread_mutex_init(&mtx2,NULL);
    pthread_t ptid;
    pthread_create(&ptid,NULL,thread1,NULL);
    printf("主线程获取 mtx2中。。。\n");
    pthread_mutex_lock(&mtx2);
    printf("主线程获取 mtx2成功！\n");
    sleep(1);
    printf("主线程获取 mtx1中。。。\n");
    pthread_mutex_lock(&mtx1);
    printf("主线程获取 mtx1成功！\n");
    pthread_mutex_unlock(&mtx1);
    pthread_mutex_unlock(&mtx2);

    printf("线程结束！！\n");
    while(1);

    return 0;
}
```

练习

模拟选座系统

有 10个空座，12个用户(线程)同时选座，系统从空座中随机选择
打印出当前用户的序号和选中座位序号(线程创建顺序就是用户序号顺序)。
输出正确结果应为:10个用户成功，2个失败，没有重复座位

```
#define SEAT_NUM 10 // 座位数量
#define USER_NUM 12 // 选座人数
pthread_mutex_t mtx;
// 座位数组
int seat[SEAT_NUM];
// 空座数量
int empty_seat_count =  SEAT_NUM;
int chooseSeat()
{
    if(empty_seat_count > 0)
    {
        // 生成随机座位号—座位数组的下标
        int rand_index = rand()%empty_seat_count;
        // 获取座位号
        int seat_num = seat[rand_index];
        // 更新座位数组
        seat[rand_index] = seat[empty_seat_count-1];
        // 更新空座数量
        empty_seat_count--;
        return seat_num;
    }
    else
    {
        return -1;
    }
}
void* thread1(void* arg)
```

```
{
    pthread_mutex_lock(&mtx); // 加锁
    int seat_num = chooseSeat(); // 选座
    pthread_mutex_unlock(&mtx); // 解锁
    if(seat_num==-1)
    {
        printf("用户 %d 选座失败，座位以售罄！ \n",*(int*)arg);
    }
    else
    {
        printf("用户 %d 选座成功，座位号是 %d\n",*(int*)arg,seat_num);
    }
}
int main(int argc, char* argv[])
{
    // 设置随机种子
    srand(time(NULL));
    // 初始化互斥锁
    pthread_mutex_init(&mtx,NULL);
    // 初始化座位号
    for(int i = 0; i < SEAT_NUM; i++)
    {
        seat[i]= i+1;
    }
    pthread_t pArr[USER_NUM]; // 存储线程id的数组
    for(int i = 0; i < USER_NUM; i++) // 给每个用户创建线程
    {
        int* pi = (int*)malloc(sizeof(int));
        *pi = i+1;
        pthread_create(&pArr[i],NULL,thread1,pi);
    }
    // 设置等待
    for(int i = 0; i < USER_NUM; i++)
    {
        pthread_join(pArr[i],NULL);
    }
    // 销毁互斥锁
    pthread_mutex_destroy(&mtx);
    return 0;
}
```

6.2 读写锁

读写锁，由读锁和写锁两部分组成，读取资源时用读锁，修改资源时用写锁。其特性为:写独占，读共享(读优先锁)。

读写锁适合读多写少的场景。

读写锁的工作原理

没有线程持有写锁时，所有线程都可以一起持有读锁
有线程持有写锁时，所有的读锁和写锁都会阻塞
读优先锁:有线程持有锁，这时有一个读线程和一个写线程想要获取锁，读线程会优先获取锁，就是读优先锁，反过来就是写优先锁。

读写锁函数

```
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,const pthread_rwlockattr_t *restrict attr),
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock),
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock)
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock),
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock)
```

场景分析

(1)持有读锁时，申请读锁：全部直接加锁成功，不需要等待
(2)持有写锁时，申请写锁:申请的写锁阻塞等待，写锁释放再申请加锁
(3)持有读锁时，申请写锁:写锁阻塞
(4)持有写锁时，申请读锁:读锁阻塞
(5)持有写锁时申请写锁和读锁:申请的读锁和写锁都会阻塞，当持有的写锁释放时，读锁先加锁成功
(6)持有读锁时申请写锁和读锁:申请的写锁阻塞，读锁加锁成功，写锁阻塞到读锁全部解锁才能加锁在此期间可能一直有读锁申请，会导致写锁一直无法申请成功，造成饥饿

读优先锁:

线程 1要加写锁，线程2要加写锁，线程3要加读锁:
线程1直接成功加锁，线程2和线程3阻塞
线程1释放锁
由于是读优先锁，线程3加锁成功，线程2继续阻塞
线程1 要加读锁，线程2要加写锁，线程3要加读锁:
线程1成功加锁
由于是读优先锁，线程 3加锁成功，线程2继续阻塞条件变量

练习

使用读写锁模拟银行账户管理:

1. 有一个变量 balance 代表账户余额，存取钱就是对 balance 进行修改
2. 存5 次钱，在执行程序时传入5个整数代表钱的数额，sleep1 秒
3. 查询 10 次余额，查余额时随机 sleep1到3秒
4. 每一次的存钱和查余额操作都使用线程完成

```
/*
使用读写锁模拟银行账户管理
1. 有一个变量 balance 代表账户余额，存取钱就是对 balance 进行修改
2. 存5次钱，在执行程序时传入5个整数代表钱的数额，sleep1秒
3. 查询 10 次余额，查余额时随机 sleep1到3秒
4. 每一次的存钱和查余额操作都使用线程完成
*/
pthread_rwlock_t rwl;
int balance = 0;
// 存钱操作
int SetBalance(int money)
{
    sleep(1);
    balance += money;
    return balance;
}
// 查看操作
int GetBalance()
{
    int rand_time = rand()%3+1;
    sleep(rand_time);
    return balance;
}
// 存钱线程
void* threadSet(void* arg)
{
    pthread_rwlock_wrlock(&rwl);
    int money = atoi((char*)arg);
    int ret = SetBalance(money);
    printf("存钱成功，现有余额为: %d\n",ret);
    pthread_rwlock_unlock(&rwl);
    return NULL;
}
// 查看线程
void* threadGet(void* arg)
{
    pthread_rwlock_rdlock(&rwl);
    int ret = GetBalance();
    printf("读取余额成功，现有余额为: %d\n",ret);
```

```
    pthread_rwlock_unlock(&rwl);
    return NULL;
}
int main(int argc, char* argv[])
{
    if(argc !=6)
    {
        printf("输入的参数有误，请输入5个存取的数额！\n");
        exit(1);
    }
    // 设置随机种子
    srand(time(NULL));
    // 初始化读写锁
    pthread_rwlock_init(&rwl,NULL);
    pthread_t pArr[15]; // 存储查看线程id的数组
    for(int i = 0; i < 5; i++) // 给存取钱创建线程
    {
        pthread_create(&pArr[i],NULL,threadSet,argv[i+1]);
    }
    for(int i = 5; i < 15; i++) // 给查看操作创建线程
    {
        pthread_create(&pArr[i],NULL,threadGet,NULL);
    }
    for(int i = 0; i < 15; i++) // 设置等待
    {
        pthread_join(pArr[i],NULL);
    }
    pthread_rwlock_destroy(&rwl);


    return 0;
}
```

结果：因为是读优先锁，因此，一旦有读锁进来，就得一直读。

```
weihong@weihong:~/linux/Day_613$ ./main 100 100 100 100 -100
存钱成功，现有余额为：100
读取余额成功，现有余额为：100
读取余额成功，现有余额为：100
读取余额成功，现有余额为：100
读取余额成功，现有余额为：100
读取余额成功，现有余额为：100
读取余额成功，现有余额为：100
读取余额成功，现有余额为：100
读取余额成功，现有余额为：100
读取余额成功，现有余额为：100
读取余额成功，现有余额为：100
存钱成功，现有余额为：200
存钱成功，现有余额为：300
存钱成功，现有余额为：400
存钱成功，现有余额为：300
weihong@weihong:~/linux/Day_613$
```

6.3 条件变量

条件变量，通知状态的改变。条件变量允许一个线程就某个共享变量的状态变化通知其他线程，并让其他线程等待(阻塞于)这一通知。**条件变量总是结合互斥量使用**。条件变量就共享变量的状态改变发出通知，而互斥量则提供对该共享变量访问的互斥。

演示程序：

定义一个变量 a，线程1当a为0时对 a+1，线程2当a为1时对 a-1，循环 10次，a的结果应该是 0和1交替。

两个线程需要不断的轮询结果，造成 CPU浪费。可以使用条件变量解决:线程1不满足运行条件时，先休眠等待，其他线程运行到满足线程1的运行条件时，通知并唤醒线程2继续执行。

条件变量函数

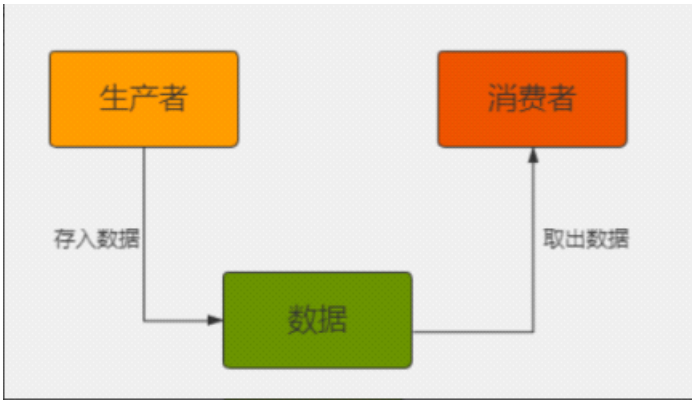
```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond _attr),
    参数 attr 表示条件变量的属性，默认值传 NULL

int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t*cond, pthread_mutex_t *mutex);
    阻塞等待一个条件变量，释放持有的互斥锁，这两步是原子操作
    被唤醒时，函数返回，解除阻塞并重新申请获取互斥锁
int pthread_cond_signal(pthread_cond_t *cond);
    唤醒一个阻塞在条件变量上的线程
int pthread_cond_broadcast(pthread_cond_t*cond),
    唤醒全部阻塞在条件变量上的线程
int pthread_cond_timedwait(pthread_cond_t*cond,
                           pthread_mutext*mutex
                           const_struct_timespec *abstime),
    限时等待一个条件变量
    参数 abstime 是一个 timespec 结构体，以秒和纳秒表示的绝对时间
```

```c
// 0-1交替 -- 条件变量
pthread_mutex_t mtx;
pthread_cond_t cont;
int a = 0;
void* thread1(void* arg)
{
    for(int i = 0;i < 10;i++)
    {
        pthread_mutex_lock(&mtx);
        // 这里只能用while，不能用if
        // 因为如果是if的话，他只会判断一次，就阻塞在锁下面了。
        // 如果，多个线程同时执行的话，因为不在判断条件（a=1），就有可能之间抢到锁，
        // 接着++ 导致a=2，不是0-1交替了。
        // 因此这里只能使用while判断，加锁失败，阻塞之后，想再次加锁，得再次判断是否满足条件
        while(a!=0)
        {
            // a != 0 时，阻塞等待，并将 thread1解锁释放，然后thread2执行
            // 如果被唤醒，则重新加锁。
            pthread_cond_wait(&cont,&mtx);
        }
        printf("a = %d\n",++a);
        pthread_cond_broadcast(&cont); //  唤醒
        pthread_mutex_unlock(&mtx);

    }
    return NULL;
}
void* thread2(void* arg)
{
    for(int i = 0;i < 10;i++)
    {
        pthread_mutex_lock(&mtx);
        while(a!=1)
        {
            // a != 1 时，阻塞等待，并将 thread2解锁释放，然后thread1执行
            // 如果被唤醒，则重新加锁。
            pthread_cond_wait(&cont,&mtx);
        }

        printf("a = %d\n",--a);
        pthread_cond_broadcast(&cont);//  唤醒
        pthread_mutex_unlock(&mtx);
    }
    return NULL;
}

int main(int argc, char* argv[])
{
    pthread_mutex_init(&mtx,NULL);
    pthread_cond_init(&cont,NULL);
```

```
    pthread_t  ptid1;
    pthread_t  ptid2;
    pthread_t  ptid3;
    pthread_t  ptid4;
    pthread_create(&ptid1,NULL,thread1,NULL);
    pthread_create(&ptid2,NULL,thread2,NULL);
    pthread_create(&ptid3,NULL,thread1,NULL);
    pthread_create(&ptid4,NULL,thread2,NULL);
    pthread_join(ptid1,NULL);
    pthread_join(ptid2,NULL);
    pthread_join(ptid3,NULL);
    pthread_join(ptid4,NULL);
    return 0;
}
```

生产者消费者模型

线程同步典型的案例即为生产者消费者**模型**，而借助条件变量来实现这一模型，是比较常见的一种方法。假定有两个线程，一个模拟生产者行为，一个模拟消费者行为。两个线程同时操作一个共享资源(一般称之为汇聚)，生产者向其中添加产品，消费者从中消费掉产品。



相较于互斥量而言，条件变量可以减少竞争。如直接使用互斥量，除了生产者、消费者之间要竞争互斥量以外，消费者之间也需要竞争互斥量，但如果汇聚(链表)中没有数据消费者之间竞争互斥量是无意义的。有了条件变量机制以后，只有生产者完成生产，才会引起消费者之间的竞争。提高了程序效率。

场景:一个线程产生随机数放入链表中，一个线程从链表中取出一个随机数打印

```
pthread_mutex_t mtx;
pthread_cond_t cond;
// 生产者消费者模型
// 场景:一个线程产生随机数放入链表中，一个线程从链表中取出一个随机数打印
// 定义结构体
typedef struct Node
{
    int number;
    struct Node* next;
}Node;
// 定义头节点
Node* head;
// 添加节点的函数
Node* AddNode(int number)
{
    // 申请新的节点
    Node* new_node = (Node*)malloc(sizeof(Node));
    // 将传入的数据，给到节点
    new_node->number = number;
    // 将指针域指向头节点
    new_node->next = head;
    // 将头节点指向申请的节点
    head = new_node;
    return head;
`
```

```
}
// 获取头结点元素的函数
int PopNode()
{
    if(head == NULL)
    {
        return -1;
    }
    Node* temp = head;
    int number = temp->number;
    head = temp->next;
    free(temp);
    return number;
}

// 生产者生成随机数，添加到链表
void* producer(void* arg)
{
    while(1)
    {
        pthread_mutex_lock(&mtx);  // 加锁
        int number = rand() % 500 + 1; // 生成随机数
        AddNode(number);    // 添加节点
        pthread_cond_signal(&cond);// 条件信号，添加完节点后，通知消费者读取数据
        pthread_mutex_unlock(&mtx);
        sleep(1);
    }
    return NULL;
}
// 消费者，取出头节点的随机数
void* consumer(void* arg)
{
    while(1)
    {
        pthread_mutex_lock(&mtx);

        while(head == NULL)
        {
            // 如果链表为空，阻塞等待，并释放持有的锁，让生产者去加锁，添加数据
            pthread_cond_wait(&cond,&mtx);
        }
        int num = PopNode();
        printf("num = %d\n",num);
        pthread_mutex_unlock(&mtx);
    }
    return NULL;
}
int main(int argc, char* argv[])
{
    // 设置随机种子
    srand(time(NULL));
    // 初始化读写锁
    pthread_mutex_init(&mtx,NULL);
    pthread_cond_init(&cond,NULL);
    pthread_t  ptid1;
    pthread_t  ptid2;
    pthread_create(&ptid1,NULL,producer,NULL);
    pthread_create(&ptid2,NULL,consumer,NULL);
    pthread_join(ptid1,NULL);
    pthread_join(ptid2,NULL);
    free(head);
    return 0;
}
```

6.4 信号量

信号量是操作系统提供的一种协调共享资源访问的方法。信号量是由内核维护的整型变量(sem)，**它的值表示可用资源的数量。**

互斥量就是可用资源数量为一的信号量

对信号量的原子操作:

P 操作:

如果有可用资源(sem>0)，占用一个资源(sem-1)

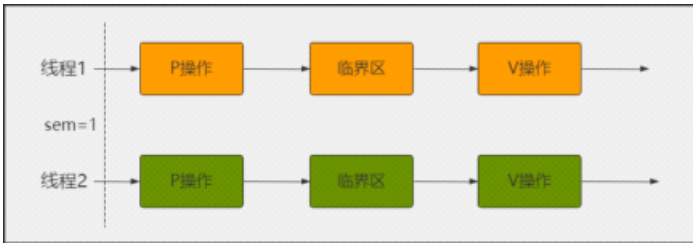
如果没有可用资源(sem=0)，进程或线程阻塞，直到有资源

V 操作:

如果没有进程或线程在等待资源，释放一个资源(sem+1)

如果有进程或线程在等待资源，唤醒一个进程或线程

P 操作和V操作成对出现，进入临界区前进行P操作，离开临界区后进行V操作



POSIX提供两种信号量，命名信号量和无名信号量，命名信号量一般是用在进程间同步，无名信号量一般用在线程间同步。

命名信号量和无名信号量通用函数:

```
int sem_wait(sem_t*sem);
    P操作，信号量大于零将信号量减一。否则进程阻塞
int sem_post(sem_t*sem);
    V操作，有阻塞进程会唤醒阻塞进程。否则信号量加一
int sem_getvalue(sem_t*sem, int *sval);
```

无名信号量函数:

```
int sem_init(sem_t*sem, int pshared, unsigned int value);
    sem:要进行初始化的信号量
    pshared:等于0用于同一进程下多线程的同步
    pshared:大于0用于多个相关进程间的同步(即fork产生的)
    ovalue:信号量的初始值
int sem_destroy(sem_t* sem);
```

命名信号量函数:

```
sem tsem_open(const char * name, int oflag, mode_t mode, unsigned int value).
    打开或创建信号量
    参数 name:信号量名字
    参数 oflag:
        oflag为 0:打开信号量
        oflag为 O_CREAT:如果 name 不存在就创建一个信号量
        oflag为 O_CREATIO EXCL:如果 name 存在，会失败
    参数mode和value:参数 oflag有 O_CREAT时，需要传这两个参数,mode 代表权限value 代表信号量的初始值。
    返回值:指向 sem_t值的指针，后续通过这个指针操作新打开或创建的信号量。

sem_close(sem_t*sem);
int sem_unlink(const char*name);
```

例子1：a自增

```c
// 无名信号量-自增1000000次
sem_t s;
int a = 0;
void* thread1(void* arg)
{
    for(int i = 0; i <10000000;i++)
    {
        sem_wait(&s); // p操作
        a++;
```

```
        sem_post(&s); // v操作
    }
    return NULL;
}
int main(int argc, char* argv[])
{
    // 无名信号量
    // 0 : 同一进程下的信号量
    // 1 : 信号量的初始值
    sem_init(&s,0,1);
    pthread_t  ptid1;
    pthread_t  ptid2;
    pthread_create(&ptid1,NULL,thread1,NULL);
    pthread_create(&ptid2,NULL,thread1,NULL);
    pthread_join(ptid1,NULL);
    pthread_join(ptid2,NULL);
    printf("a = %d\n",a);
    return 0;
}
```

例子2：0-1交替

```
// 01交替--信号量
/*
    这里加锁是对 ++和--操作加锁，所以需要两个信号量
    而不是对a进行就加锁。
    如果是对a进行加锁，那么++a之后，就自动解锁了，还能接着++a，就不是0-1交替了
    正确的做法是
    加操作的信号量（s_add）初始值为1，可以进行++a。（因为a的初始值为0，得先进行加操作）
    减操作的信号量（s_sub）初始值为0，开始就阻塞，等加操作结束。

    此时加操作进行p操作。s_add = 0;之后就不能进行加操作了。
    ++a结束之后，对s_sub 进行v操作，s_sub =1;
    此时 加操作阻塞，减操作的信号量为1，可以执行减操作。减操作信号量进行p操作，s_sub = 0;
    --a结束之后，对s_add 进行 v操作，s_add = 1; 阻塞的加操作可以执行。

*/
sem_t s_add; // 加操作信号量
sem_t s_sub; // 减操作信号量
int a = 0;
void* add(void* arg)
{
    while(1)
    {
        sem_wait(&s_add); // 加操作执行 p 操作，s_add = 0;
        printf("a = %d\n",++a); // ++a
        sleep(1);
        sem_post(&s_sub); // 减操作执行 v 操作，s_add = 1;

    }
    return NULL;
}
void* sub(void* arg)
{
    while(1)
    {
        sem_wait(&s_sub); // 减操作执行 p 操作，s_sub = 0;
        printf("a = %d\n",--a);
        sleep(1);
        sem_post(&s_add); // 加操作执行 v 操作，s_add = 1;

    }
    return NULL;
}
int main(int argc, char* argv[])
{
    sem_init(&s_add,0,1); // 加操作信号量初始值为1
    sem_init(&s_sub,0,0); // 减操作信号量初始值为0
    pthread_t  ptid1;
```

```
    pthread_t  ptid2;
    pthread_create(&ptid1,NULL,add,NULL);
    pthread_create(&ptid2,NULL,sub,NULL);
    pthread_join(ptid1,NULL);
    pthread_join(ptid2,NULL);
    return 0;
}
```

例子3:信号量实现生产者消费者模型，生产者线程产生随机数存入数组，消费者线程从数组取出一个随机数打印。

```
// 例子3:信号量实现生产者消费者模型,
// 生产者线程产生随机数存入数组，消费者线程从数组取出一个随机数打印。
sem_t arr; // 数组的信号量
sem_t s; // 消费者的信号量
int num_Arr[3]; // 数组
void* producer(void* arg)
{
    int index = 0;
    while(1)
    {
        sem_wait(&arr); // 数组信号量加锁
        int rand_num = rand()%500; // 产生随机数
        num_Arr[index%3] = rand_num; // 赋值
        printf("producer num_Arr[%d] = %d\n",index%3,num_Arr[index%3]);
        index++; // 改变下标
        sleep(1);
        sem_post(&s);

    }
    return NULL;
}
void* consumer(void* arg)
{
    int index = 0;
    while(1)
    {
        sem_wait(&s);
        printf("consumer num_Arr[%d] = %d\n",index%3,num_Arr[index%3]);
        num_Arr[index%3] = -1;
        index++;
        sleep(1);
        sem_post(&arr);
    }

    return NULL;
}
int main(int argc, char* argv[])
{
    srand((unsigned int)time(NULL));
    sem_init(&arr,0,3);
    sem_init(&s,0,0);
    pthread_t ptid1;
    pthread_t ptid2;
    pthread_create(&ptid1,NULL,producer,NULL);
    pthread_create(&ptid2,NULL,consumer,NULL);
    pthread_join(ptid1,NULL);
    pthread_join(ptid2,NULL);
    sem_destroy(&arr);
    sem_destroy(&s);
    return 0;
}
```