第四章：多线程编程

一、多线程基础

1> 多线程的引入：也是为了实现多任务并发执行的问题的，**能够实现多个阻塞任务同时执行**

2> 多线程（LWP轻量版进程）：**线程是粒度更小的任务执行单元**

3> **进程是资源分配的基本单位，而线程是任务器进行任务调度的最小单位**

4> **一个进程可以拥有多个线程，同一个进程中的多个线程共享进程的资源**

5> 由于线程是共用进程的资源，所以对于线程的切换而言，开销较小

6> 由于多个线程使用的是同一个进程的资源，那么，就会导致每个进程使用资源时，产生资源抢占问题，没有多进程安全

7> 每个进程至少有一个线程：主线程

8> 只要有一个线程中退出了进程，那么所有的线程也就结束了

主线程结束后，整个进程也就结束了

9> 多个线程执行顺序：没有先后顺序，按时间片轮询，上下文切换，抢占CPU的方式进行

二、多线程编程（重点）

由于C库没有提供有关多线程的相关操作，对于多线程编程要依赖于第三方库

头文件：#include<pthread.h>

编译时：需要加上 -lpthread 选项，链接上对于的线程支持库

2.1 创建线程：pthread_create

```
        #include <pthread.h>      //头文件

        int pthread_create(pthread_t *thread, const pthread_attr_t *attr,void *
(*start_routine) (void *), void *arg);
        功能：创建一个分支线程
        参数1：线程号，通过参数返回，用法：在外部定义一个该类型的变量，将地址传递入函数，调用结束后，该变量中即是线程号
        参数2：线程属性，一般填NULL，让系统使用默认属性创建一个线程
        参数3：是一个回调函数，一个函数指针，需要向该参数中传递一个函数名，作为线程体执行函数
                该函数由用户自己定义，参数是void*类型，返回值也是void *类型
        参数4：  是参数3的参数，如果不想向线程体内传递数据，填NULL即可
        返回值：成功返回0，失败返回一个错误码（非linux内核的错误码，是线程支持库中定义的一个错误码）


        Compile and link with -pthread.     //编译时需要加上 -pthread选项
```

1> 不向线程体传递数据

```c
#include<myhead.h>

//定义线程体函数
void *task(void *arg)
{
    while(1)
    {
    printf("我是分支线程\n");
    sleep(1);
    }


}

/**************主程序**********************************/
int main(int argc, const char *argv[])
{
    pthread_t tid = -1;    //用于存储线程号的变量
    if(pthread_create(&tid, NULL, task, NULL) != 0)
    //参数2：表示让系统使用默认属性创建一个线程
    //参数3：线程体函数名
    //参数4：表示向线程体中传递的数据
    {
        printf("tid create error\n");
        return -1;
    }

    printf("pthread_create success\n");
    while(1)
    {
    printf("我是主线程\n");
    sleep(1);
    }

    while(1);          //防止主线程结束


    return 0;
}
```

2> 向线程体中传递单个数据

```c
#include<myhead.h>

//定义线程体函数
void *task(void *arg)
{
    //arg --> &num  但是arg是一个void*类型的变量，需要转换为具体指针进行操作
    //(int*)arg    --->将其转换为整型的指针
    //*(int *)arg  --->num的值
    int key = *(int*)arg;


    printf("我是分支线程：num = %d\n", key);        //1314
```

```
}

/**************主程序**********************************/
int main(int argc, const char *argv[])
{
    pthread_t tid = -1;    //用于存储线程号的变量
    int num = 520;

    if(pthread_create(&tid, NULL, task, &num) != 0)
    //参数2：表示让系统使用默认属性创建一个线程
    //参数3：线程体函数名
    //参数4：表示向线程体中传递的数据
    {
        printf("tid create error\n");
        return -1;
    }

    printf("pthread_create success,tid = %#lx\n", tid);
    printf("我是主线程\n");

    num = 1314;           //主线程中更改数据
    printf("主线程中num = %d\n", num);


    while(1);          //防止主线程结束


    return 0;
}
```

3> 向线程体传入多个数据

```
#include<myhead.h>

//定义信息结构体，用于向线程体传递数据
struct Info
{
    int num;
    char name[20];
    double score;
};


//定义线程体函数
void *task(void *arg)
{
    Info buf = *((Info*)arg);          //将结构体指针转换为结构体变量

    printf("分支线程中：num = %d, name = %s, score = %.2lf\n", buf.num, buf.name,
buf.score);

}

/**************主程序**********************************/
int main(int argc, const char *argv[])
{
```

```
    pthread_t tid = -1;     //用于存储线程号的变量
    int num = 520;
    char name[20] = "zhangsan";
    double score = 99.5;

    //需求：将上面的三个数据全部传入线程体中
    Info buf = {num, "zhangsan", score};



    if(pthread_create(&tid, NULL, task, &buf) != 0)
    //参数2：表示让系统使用默认属性创建一个线程
    //参数3：线程体函数名
    //参数4：表示向线程体中传递的数据
    {
        printf("tid create error\n");
        return -1;
    }

    printf("pthread_create success,tid = %#lx\n", tid);
    printf("我是主线程\n");


    while(1);          //防止主线程结束


    return 0;
}
```

2.2 线程号的获取：pthread_self

```
#include <pthread.h>

pthread_t pthread_self(void);
功能：获取当前线程的线程号
参数：无
返回值：返回调用线程的id号，不会失败
```

2.3 线程的退出函数：pthread_exit

```
#include <pthread.h>

void pthread_exit(void *retval);
功能：退出当前线程
参数：表示退出时的状态，一般填NULL
返回值：无
```

2.4 线程的资源回收：pthread_join

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
功能：阻塞回收指定线程的资源
参数1：要回收的线程线程号
参数2：线程退出时的状态，一般填NULL
返回值：成功返回0，失败返回一个错误码
```

2.5 线程分离态：pthread_detach

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
功能：将指定线程设置成分离态，被设置成分离态的线程，退出后，资源由系统自动回收
参数：要分离的线程号
返回值：成功返回0，失败返回一个错误码
```

```c
#include <myhead.h>

//定义线程体函数
void *task(void *arg)
{
    printf("分支线程,tid = %#x\n", pthread_self());      //2、调用函数，输出当前线程的线程号


    sleep(3);
    //3、退出线程
    pthread_exit(NULL);
}


/**********************主程序*****************************/
int main() {

    //1、定义一个线程号变量
    pthread_t tid = -1;
    //创建一个线程
    if(pthread_create(&tid, NULL, task, NULL) != 0)
    {
        printf("pthread_create error\n");
        return -1;
    }

    printf("主线程,tid = %#x\n", tid);


    //4、回收分支线程的资源
    //pthread_join(tid, NULL);          //前3秒，主线程处于休眠等待分支线程的结束
                                        //分支线程处于休眠状态

    //将线程设置成分离态(非阻塞)
    pthread_detach(tid);

    sleep(5);
```

```
    // while(1);                        //防止主线程结束
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

练习：使用多线程完成两个文件的拷贝，线程1拷贝前一半内容，线程2拷贝后一半内容，主线程用于回收两个分支线程的资源

```
#include <myhead.h>

//定义要向线程体函数中出入数据的结构体类型
struct Info
{
    const char *srcfile;    //要拷贝的原文件
    const char *destfile;   //目标文件
    int start;              //起始位置
    int len;                //要拷贝的长度
};

//定义获取文件长度的函数
int get_file_len(const char *srcfile, const char *destfile)
{
    //定义两个文件描述符，分别作为源文件和目标文件的句柄
    int sfd, dfd;
    //以只读的形式打开源文件
    if((sfd = open(srcfile, O_RDONLY)) == -1)
    {
        perror("open srcfile error");
        return -1;
    }

    //以只写的形式打开目标文件
    if((dfd = open(destfile, O_RDWR|O_CREAT|O_TRUNC, 0664)) == -1)
    {
        perror("open destfile error");
        return -1;
    }

    //获取源文件的长度
    int len = lseek(sfd, 0, SEEK_END);

    //关闭文件
    close(sfd);
    close(dfd);

    return len;
}

//定义线程体函数
void *task(void *arg)
{
    //将传入的数据解析出来
    const char *srcfile = ((struct Info*)arg)->srcfile;
    const char *destfile = ((struct Info*)arg)->destfile;
    int start = ((struct Info*)arg)->start;
    int len = ((struct Info*)arg)->len;
```

```
    //准备拷贝工作
    //定义两个文件描述符，分别作为源文件和目标文件的句柄
    int sfd, dfd;
    //以只读的形式打开源文件
    if((sfd = open(srcfile, O_RDONLY)) == -1)
    {
        perror("open srcfile error");
        return NULL;
    }

    //以只写的形式打开目标文件
    if((dfd = open(destfile, O_RDWR)) == -1)
    {
        perror("open destfile error");
        return NULL;
    }

    //偏移指针
    lseek(sfd, start, SEEK_SET);
    lseek(dfd, start, SEEK_SET);

    //拷贝工作
    int ret = 0;          //记录每次读取的数据
    int count = 0;         //记录拷贝的总个数
    char buf[128] = "";          //数据搬运工
    while(1)
    {
        ret = read(sfd, buf, sizeof(buf));
        //将读取的数据放入到count中
        count += ret;
        if(count >= len)
        {
            //说明该部分的内容拷贝结束，还剩最后一次
            write(dfd, buf, ret - (count-len));
            break;
        }

        //其余的正常拷贝
        write(dfd, buf, ret);
    }

    //关闭文件描述符
    close(dfd);
    close(sfd);

}



int main(int argc, const char *argv[])
{
    //判断传入的文件个数是否正确
    if(argc != 3)
    {
        printf("input file error\n");
        printf("usage:./a.out srcfile destfile\n");
        return -1;
```

```
    }

    //获取原文件的长度,顺便将目标文件创建出来
    int len = get_file_len(argv[1], argv[2]);

    //创建两个线程
    pthread_t tid1, tid2;

    //定义向线程体函数传参的变量
    struct Info buf[2] = {{argv[1], argv[2], 0, len/2}, \
                          {argv[1], argv[2], len/2, len-len/2}};

    if(pthread_create(&tid1, NULL, task, &buf[0]) != 0)
    {
        printf("线程创建失败\n");
        return -1;
    }

    if(pthread_create(&tid2, NULL, task, &buf[1]) != 0)
    {
        printf("线程创建失败\n");
        return -1;
    }

    //主线程中完成对两个分支线程资源的回收
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("拷贝成功\n");


    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

三、线程的同步互斥机制（难点）

3.1 线程同步互斥机制的引入

1> 由于同一个进程的多个线程会共享进程的资源，这些被共享的资源称为临界资源

2> 多个线程对公共资源的抢占问题，访问临界资源的代码段称为临界区

3> 多个线程抢占进程资源的现象称为竞态

4> 为了解决竞态，我们引入了同步互斥机制

3.2 线程的互斥机制之互斥锁

1> 互斥锁的本质也是一个特殊的临界资源，当该临界资源被某个线程所拥有后，其他线程就不能拥有该资源，直到，拥有该资源的线程释放掉互斥锁后，其他线程才能进行抢占（同一时刻，一个互斥锁只能被一个线程所拥有）

2> 互斥锁的相关API函数接口

```
1、创建一个互斥锁：只需定义一个pthread_mutex_t 类型的变量即创建了一个互斥锁
```

```
    pthread_mutex_t mutex;
2、初始化互斥锁
    pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;       //静态初始化
    int pthread_mutex_init(pthread_mutex_t *restrict mutex,const
pthread_mutexattr_t *restrict attr);
    功能：初始化互斥锁变量
    参数1：互斥锁变量的地址，属于地址传递
    参数2：互斥锁属性，一般填NULL，让系统自动设置互斥锁属性
    返回值：成功返回0，失败返回错误码

3、获取锁资源
    int pthread_mutex_lock(pthread_mutex_t *mutex);
    功能：获取锁资源,如果要获取的互斥锁已经被其他线程锁定，那么该函数会阻塞，直到能够获取锁资源
    参数：互斥锁地址，属于地址传递
    返回值：成功返回0，失败返回一个错误码

4、释放锁资源
    int pthread_mutex_unlock(pthread_mutex_t *mutex);
    功能：释放对互斥锁资源的拥有权
    参数：互斥锁变量的地址
    返回值：成功返回0，失败返回一个错误码

5、销毁互斥锁
    int pthread_mutex_destroy(pthread_mutex_t *mutex);
    功能：销毁互斥锁
    参数：互斥锁变量的地址
    返回值：成功返回0，失败返回一个错误码
```

```c
#include <myhead.h>

//11、创建一个互斥锁
pthread_mutex_t mutex;

//定义一个全局资源
int num = 520;              //临界资源

//定义分支线程1
void *task1(void *arg)
{
    while(1)
    {
         sleep(1);

        //33、获取锁资源
        pthread_mutex_lock(&mutex);

        num -= 10;          //线程1将临界资源减少10

        printf("张三取了10，剩余%d\n", num);

        //44、释放锁资源
        pthread_mutex_unlock(&mutex);
    }
}
```

```c
//定义分支线程2
void *task2(void *arg)
{
    while(1)
    {
        sleep(1);

        //33、获取锁资源
        pthread_mutex_lock(&mutex);

        num -= 20;          //线程1将临界资源减少10

        printf("李四取了20，剩余%d\n", num);

        //44、释放锁资源
        pthread_mutex_unlock(&mutex);
    }
}

/*****************************主线程*****************************/
int main() {

    //22、初始化互斥锁,参数NULL表示让系统自动分配互斥锁属性
    pthread_mutex_init(&mutex, NULL);

    //1、创建两个分支线程
    pthread_t tid1,tid2;
    if(pthread_create(&tid1, NULL, task1, NULL) != 0)
    {
        printf("tid1 create error\n");
        return -1;
    }
    if(pthread_create(&tid2, NULL, task2, NULL) != 0)
    {
        printf("tid2 create error\n");
        return -1;
    }

    printf("主线程：tid1 = %#x, tid2 = %#x\n", tid1, tid2);

    //2、阻塞等待线程结束
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    //55、释放锁资源
    pthread_mutex_destroy(&mutex);

    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

3.3 线程同步之无名信号量

1> 线程同步：就是多个线程之间有先后顺序得执行，这样在访问临界资源时，就不会产生抢占现象了

2> 同步机制常用于生产者消费者模型：消费者任务要想执行，必须先执行生产者线程，多个任务有顺序执行

3> 无名信号量：本质上也是一个特殊的临界资源，内部维护了一个value值，当某个进行想要执行之前，先申请该无名信号量的value资源，如果value值大于0，则申请资源函数接触阻塞，继续执行后续操作。如果value值为0，则当前申请资源函数会处于阻塞状态，直到其他线程将该value值增加到大于0

3> 无名信号量的相关API函数接口

```
1、创建无名信号量：只需定义一个sem_t 类型的变量即可
    sem_t sem;

2、初始化无名信号量
        #include <semaphore.h>

        int sem_init(sem_t *sem, int pshared, unsigned int value);
        功能：初始化无名信号量，最主要是初始化value值
        参数1：无名信号量的地址
        参数2：判断进程还是线程的同步
                0：表示线程间同步
                非0：表示进程间同步，需要创建在共享内存段中
         参数3：无名信号量的初始值
         返回值：成功返回0，失败返回-1并置位错误码

3、申请无名信号量的资源（P操作）
        #include <semaphore.h>

        int sem_wait(sem_t *sem);
        功能：阻塞申请无名信号量中的资源，成功申请后，会将无名信号量的value进行减1操作，如果当前无名信号量的value为0，则阻塞
        参数：无名信号量的地址
        返回值：成功返回0，失败返回-1并置位错误码

4、释放无名信号量的资源（V操作）
        #include <semaphore.h>

        int sem_post(sem_t *sem);
        功能：将无名信号量的value值增加1操作
        参数：无名信号量地址
        返回值：成功返回0，失败返回-1并置位错误码

5、销毁无名信号量
        #include <semaphore.h>

        int sem_destroy(sem_t *sem);
        功能：销毁无名信号量
        参数：无名信号量地址
        返回值：成功返回0，失败返回-1并置位错误码
```

```
#include <myhead.h>

//11、创建无名信号量
sem_t sem;

//创建生产者线程
void *task1(void *arg)
{
```

```
    int num = 5;
    while(num--)
    {
        sleep(1);
        printf("我生产了一辆特斯拉\n");

        //44、释放无名信号量资源
        sem_post(&sem);
    }

    //退出线程
    pthread_exit(NULL);
}

//创建消费者线程
void *task2(void *arg)
{
    int num = 5;
    while(num--)
    {
        //33、申请无名信号量的资源
        sem_wait(&sem);

        printf("我消费了一辆特斯拉，很开心\n");
    }

    //退出线程
    pthread_exit(NULL);
}


/*******************************主程序*************************/
int main() {

    //22、初始化无名信号量,第一个0表示用于线程间通信，第二个0表示初始值为0
    sem_init(&sem, 0, 0);

    //1、创建两个分支线程
    pthread_t tid1,tid2;
    if(pthread_create(&tid1, NULL, task1, NULL) != 0)
    {
        printf("tid1 create error\n");
        return -1;
    }
    if(pthread_create(&tid2, NULL, task2, NULL) != 0)
    {
        printf("tid2 create error\n");
        return -1;
    }

    printf("主线程：tid1 = %#x, tid2 = %#x\n", tid1, tid2);

    //2、阻塞等待线程结束
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    //55、销毁无名信号量
    sem_destroy(&sem);
```

```
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

练习：使用无名信号量完成，定义三个任务，任务1打印A，任务2打印B，任务3打印C，最终输出的结果为ABCABCABCABC...

```
#include <myhead.h>

//1、定义三个无名信号量
sem_t sem1, sem2, sem3;

//定义三个任务，分别打印A\B\C
void *task1(void *arg)
{
    while(1)
    {
        //申请sem1的资源
        sem_wait(&sem1);

        sleep(1);
        printf("A");
        fflush(stdout);        //刷新缓冲区

        //释放sem2的资源
        sem_post(&sem2);
    }
}

void *task2(void *arg)
{
    while(1)
    {
        //申请sem2的资源
        sem_wait(&sem2);

        sleep(1);
        printf("B");
        fflush(stdout);        //刷新缓冲区

        //释放sem3的资源
        sem_post(&sem3);
    }
}

void *task3(void *arg)
{
    while(1)
    {
        //申请sem3的资源
        sem_wait(&sem3);

        sleep(1);
        printf("C");
```

```
        fflush(stdout);       //刷新缓冲区

        //释放sem1的资源
        sem_post(&sem1);
    }
}




/********************主程序***************** */
int main(int argc, const char *argv[])
{

    //初始化无名信号量
    sem_init(&sem1, 0, 1);
    sem_init(&sem2, 0, 0);
    sem_init(&sem3, 0, 0);

    //准备三个任务
    pthread_t tid1, tid2, tid3;

    //创建三个任务
    if(pthread_create(&tid1, NULL, task1, NULL) != 0)
    {
        printf("tid1 create error\n");
        return -1;
    }
    if(pthread_create(&tid2, NULL, task2, NULL) != 0)
    {
        printf("tid2 create error\n");
        return -1;
    }
    if(pthread_create(&tid3, NULL, task3, NULL) != 0)
    {
        printf("tid3 create error\n");
        return -1;
    }

    //阻塞等待分支线程的结束
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_join(tid3, NULL);

    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

3.4 线程同步之条件变量

1>　条件变量本质上也是一个临界资源，他维护了一个队列，当消费者线程想要执行时，先进入队列中等待生产者的唤醒。执行完生产者，再由生产者唤醒在队列中的消费者，这样就完成了生产者和消费者之间的同步关系。

2> 但是，多个消费者在进入休眠队列的过程是互斥的，所以，在消费者准备进入休眠队列时，需要使用互斥锁来进行互斥操作

3> 条件变量的API函数接口

```
1、创建一个条件变量，只需定义一个pthread_cond_t类型的全局变量即可
    pthread_cond_t cond;

2、初始化条件变量
            pthread_cond_t cond = PTHREAD_COND_INITIALIZER;      //静态初始化
    int pthread_cond_init(pthread_cond_t *restrict cond,const pthread_condattr_t
*restrict attr);
    功能：初始化条件变量
    参数1：条件变量的起始地址
    参数2：条件变量的属性，一般填NULL
    返回值：成功返回0，失败返回一个错误码

3、消费者线程进入等待队列
    int pthread_cond_wait(pthread_cond_t *restrict cond,pthread_mutex_t
*restrict mutex);
    功能：将线程放入休眠等待队列，等待其他线程的唤醒
    参数1：条件变量的地址
    参数2：互斥锁，由于多个消费者线程进入等待队列时会产生竞态，为了解决竞态，需要使用一个互斥锁
    返回值：成功返回0，失败返回错误码

4、生产者线程唤醒休眠队列中的任务
       int pthread_cond_broadcast(pthread_cond_t *cond);
       功能：唤醒条件变量维护的队列中的所有消费者线程
       参数：条件变量的地址
       返回值：成功返回0，失败返回错误码

       int pthread_cond_signal(pthread_cond_t *cond);
       功能：唤醒条件变量维护的队列中的第一个进入队列的消费者线程
       参数：条件变量的地址
       返回值：成功返回0，失败返回错误码

5、销毁条件变量
       int pthread_cond_destroy(pthread_cond_t *cond);
       功能：销毁一个条件变量
       参数：条件变量的地址
       返回值：成功返回0，失败返回错误码
```

```c
#include <myhead.h>

//11、定义一个条件变量
pthread_cond_t cond;
//111、定义一个互斥锁
pthread_mutex_t mutex;

//创建生产者线程
void *task1(void *arg)
{
    /*
    int num = 3;
    while(num--)
    {
```

```
        sleep(1);
        printf("%#x:生产了一辆特斯拉\n", pthread_self());

        //44、唤醒一个消费者进行消费
        pthread_cond_signal(&cond);
    }
    */
   sleep(3);
   printf("我生产了3辆特斯拉\n");
   //44、唤醒所有消费者线程
    pthread_cond_broadcast(&cond);

    //退出线程
    pthread_exit(NULL);
}

//创建消费者线程
void *task2(void *arg)
{
    //333、获取锁资源
    pthread_mutex_lock(&mutex);

    //33、进入休眠队列，等待生产者的唤醒
    pthread_cond_wait(&cond, &mutex);

    printf("%#x:消费了一辆特斯拉，很开心\n", pthread_self());

    //444、释放锁资源
    pthread_mutex_unlock(&mutex);

    //退出线程
    pthread_exit(NULL);
}




int main()
{

    //22、初始化条件变量
    pthread_cond_init(&cond, NULL);
    //222、初始化互斥锁
    pthread_mutex_init(&mutex, NULL);

    //1、创建两个分支线程
    pthread_t tid1,tid2,tid3,tid4;
    if(pthread_create(&tid1, NULL, task1, NULL) != 0)
    {
        printf("tid1 create error\n");
        return -1;
    }
    if(pthread_create(&tid2, NULL, task2, NULL) != 0)
    {
        printf("tid2 create error\n");
        return -1;
    }
    if(pthread_create(&tid3, NULL, task2, NULL) != 0)
```

```
    {
        printf("tid2 create error\n");
        return -1;
    }
    if(pthread_create(&tid4, NULL, task2, NULL) != 0)
    {
        printf("tid2 create error\n");
        return -1;
    }

    printf("主线程：tid1 = %#x, tid2 = %#x, tid3 = %#x, tid4 = %#x\n", tid1,
tid2, tid3, tid4);

    //2、阻塞等待线程结束
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_join(tid3, NULL);
    pthread_join(tid4, NULL);

    //55、销毁条件变量
    pthread_cond_destroy(&cond);
    ///555、销毁互斥锁
    pthread_mutex_destroy(&mutex);


    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

四、C++11中的多线程

1> C++中的线程支持库，是在C++11后引入的，是系统提供的类模板库

2> 编译时，需要引入 -std=c++11,并且也要链接线程支持库 -pthread

3> 需要引入头文件：#include<thread.h>

4> C++11关于线程的相关操作

```
1、创建线程，只需要构造一个 thread类的对象即可，使用有参构造完成
    例如：thread th(func);
2、阻塞回收线程资源：join
3、线程分离态：detach
```

```
#include <myhead.h>
#include<thread>          //线程支持库的头文件
using namespace std;       //引入名字空间

//定义全局函数作为线程体函数
void task(int a, int b, int c)
{
    int num = 5;
    while(num--)
    {
        sleep(1);
```

```
        printf("我是分支线程, a = %d, b = %d, c= %d\n", a, b, c);
    }
}




int main()
{
    //实例化一个线程对象
    thread th(task, 1,2,3);

    //阻塞等待分支线程的结束，并回收资源
    //th.join();

    //将线程设置成分离态
    th.detach();

    sleep(3);

    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```