多线程

一.多线程的概念

- 什么是多线程

  线程是从进程的内存资源中切割一部分内存，提供给线程使用，线程也能够实现多任务并发执行。

  Windows系统，linux系统，还有STM32它们都是支持多任务的。

  早期的操作系统中并没有线程的概念，进程是能拥有资源和独立运行的最小单位，也是程序执行的最小单位。
  任务调度采用的是时间片轮转的抢占式调度方式，而进程是任务调度的最小单位，每个进程有各自独立的一块内存，
  使得各个进程之间内存地址相互隔离（独立）。

  后来，随着计算机的发展，对CPU的要求越来越高，进程之间的切换开销较大，
  已经无法满足越来越复杂的程序的要求了。

  为了提高系统的性能，于是就发明了线程，许多操作系统规范里引用了轻量级进程的概念，
  也就是线程。而且线程是CPU系统调度的最小单位。

二.线程与进程的区别

定义上的区别：

进程：具有一个独立功能的程序，拥有独立的内存空间。

线程：它是比进程更小的能独立运行的基本单位，线程本身是不拥有系统资源的，
但是它可以和属于同一个进程的其他线程共享进程所拥有的全部资源。

调度区别：

进程是拥有资源的基本单位。

线程是调度和分派的基本单位。

共享地址空间：

进程：拥有各自独立的地址空间、资源，所以共享复杂，需要用IPC（进程间通信），但是同步简单。

线程：共享所属进程的资源，因此共享简单，但是同步复杂，需要用加锁等措施。

占用内存和cpu：

进程：占用内存多，切换复杂，cpu利用率低；

线程：占用内存少，切换简单，cpu利用率高。

互相影响：

进程之间不会互相影响；

一个线程挂掉可能会导致整个进程挂掉。

三.基本相关API

- **创建一条新线程**

```
NAME
       pthread_create - create a new thread

SYNOPSIS
       #include <pthread.h>

       int pthread_create(pthread_t *thread,
                          const pthread_attr_t *attr,
```

```
                          void *(*start_routine) (void *),
                         void *arg);

        Compile and link with -pthread.      //编译程序时,需要连接线程库.

//参数解析
    参数1 pthread_t *thread :  新线程TID.                    保存线程的id号.(正整数)
    参数2 const pthread_attr_t *attr  : 线程属性.            一般设置为NULL,表示使用系统默认的属性.(标准属性)
    参数3 void *(*start_routine) (void *)  :  线程例程.     (函数指针) 返回值为void* 参数为 void* 与 signal函数用法一致.
                                                            创建线程需要完成的任务,靠这个函数来实现.称之为线程任务函数.
    参数4 void *arg :  传递给线程的参数.

//返回值
    成 功 : 返回0;
    失 败 : 返回errno;

#注意
    在使用多线程函数的时,需要链接我们的线程库 -pthread.
    也就是在gcc example.c -o main -pthread.
    如果没有链接,在编译时则会报错,必须要链接我们的线程库.
```

```
基本使用示例:
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

void *fun(void * arg) //线程任务函数    接口为固定的除了函数名 与 参数名可以改变,其他的不允许改变
{
   while(1)
```

```
    {
            printf("11111111\n");
            sleep(1);
    }
}
int main(int argc, char const *argv[])  //执行main函数的线程 被称为主线程
{

    pthread_t thread_id;
    //1.创建一个子线程
    pthread_create(&thread_id,    //线程的ID号
                   NULL,          //表示系统标准属性,不额外设置线程属性
                   fun,           //线程将要执行的任务函数名
                   NULL);         //不传递额外参数给线程
    while(1)
    {
            printf("2222222222\n");
            sleep(1);
    }

    return 0;
}

```

- **退出当前线程**

```
NAME
       pthread_exit - terminate calling thread

```

```
SYNOPSIS
       #include <pthread.h>

       void pthread_exit(void *retval);

参数解析:
    void *retval : 保存线程的退出值.

    配合着 pthread_join()使用.


//获取线程的ID
SYNOPSIS
       #include <pthread.h>

       pthread_t pthread_self(void);
返回值:
       成功 : 线程ID号
       这个函数是永远不会失败的.
```

- **等待进程退出**

```

NAME
       pthread_join - join with a terminated thread

SYNOPSIS
       #include <pthread.h>
函数原型 :
```

```
        int pthread_join(pthread_t thread, void **retval);  //阻塞等待子线程退出

        该函数会使调用者阻塞等待，直到所指定的线程退出为止。
        该返回时系统将回收退出线程的资源，调用线程可以获得退出线程的返回值。

        Compile and link with -pthread.

参数解析 :
        pthread_t thread : 线程的ID号,(指定等待某个线程退出);
        void **retval    : 储存线程退出值的指针.(保留线程退出时所返回的数据);

返回值:
         成功 : 返回0;
         失败 : 返回errno值;
```

```
使用示例:
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <strings.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```c
#include <sys/shm.h>
#include <pthread.h>

void *fun(void * arg) //线程任务函数    接口为固定的除了函数名 与 参数名可以改变,其他的不允许改变
{

    int a = *(int *)arg;
    printf("%d \n", a);
     //不需要返回值写法
     //pthread_exit(NULL);
     //需要返回值写法
    pthread_exit("abcd");
}

int main(int argc, char const *argv[])  //执行main函数的线程 被称为主线程
{

    int a = 100;

    pthread_t thread_id;
    //1.创建一个子线程
    pthread_create(&thread_id,   //线程的ID号
                   NULL,          //表示系统标准属性,不额外设置线程属性
                   fun,           //线程将要执行的任务函数名
                   &a);           //不传递额外参数给线程

     //只等待,不需要返回值
     //pthread_join(thread_id,NULL);

    //等待以上线程退出,并获取退出值
```

```
    void *ret;
    pthread_join(thread_id, &ret);
    printf("%s\n", (char *)ret);
    return 0;
}
```

- **线程的取消**

```
SYNOPSIS
       #include <pthread.h>

       int pthread_cancel(pthread_t thread);
参数解析 :
        pthread_t thread : 需要取消的线程ID号.


返回值:
        成功 : 0;
        失败 : 错误码.
```

多线程属性设置

```
使用示例:
```

```c

/*
    线程取消
    pthread_cancel()函数使用示例
*/

void *fun(void *arg)
{
    while(1)
    {
            printf("线程正在执行...\n");
            sleep(1);
    }
}

int main(int argc, char **argv)
{
    pthread_t tid;

    pthread_create( &tid, NULL, fun, NULL);

    //主线程 隔5秒 取消线程.
    sleep(5);
    pthread_cancel(tid);
    pthread_join(tid, NULL);

    return 0;
}
```

- 设置线程取消属性

```
NAME
       pthread_setcancelstate, pthread_setcanceltype - set cancelability state and type

SYNOPSIS
       #include <pthread.h>

       int pthread_setcancelstate(int state, int *oldstate); //设置线程取消的状态
       int pthread_setcanceltype(int type, int *oldtype);    //设置线程取消的类型


       Compile and link with -pthread.

       线程取消类型有两种 : 延时取消 与 立即取消
       线程取消状态有两种 : 可取消 与 不可取消
//参数解析:

    int state :
                PTHREAD_CANCEL_ENABLE        //可以取消,默认是可以被取消的.
                PTHREAD_CANCEL_DISABLE       //不可取消

    int *oldstate : 用于保存原本线程的取消状态,不想保存则设置为NULL;


    int type  :
     //延时取消,当线程被取消时,该线程不会立即退出,会继续执行直到遇到取消函数点时,线程才会结束
     //取消点函数,是Linux中已经规定好的一系列函数. shell命令查看 man 7 pthreads

     //默认的取消状态为延迟取消
```

```
                PTHREAD_CANCEL_DEFERRED
    //当线程被取消的时候,该线程会立即退出.
                PTHREAD_CANCEL_ASYNCHRONOUS   //立即取消

   int *oldtype  :  用于保存原本线程的取消类型,不想保存则设置为NULL;
```

```c
/*
    函数使用示例
    设置线程不可取消
    int pthread_setcancelstate(int state, int *oldstate);
*/
void *fun(void *arg)
{
    //设置当前线程不可取消,不保存当前线程取消状态
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);

    while(1)
    {
            printf("线程正在执行...\n");
            sleep(1);
    }
}

int main(int argc, char **argv)
{
    pthread_t tid;

    pthread_create( &tid, NULL, fun, NULL);

```

```
    //主线程 隔5秒 取消线程.
    sleep(5);
    pthread_cancel(tid);
    pthread_join(tid, NULL);

    return 0;
}
```

```
/*
    函数使用示例
    设置线程取消的类型 为立即取消
    int pthread_setcanceltype(int type, int *oldtype);    //设置线程取消的类型
*/
void *fun(void *arg)
{
    printf("我将要被取消了...\n");
    //设置当前线程立即取消,不保存当前线程取消类型
    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
    printf("被取消了\n");

    while(1)
    {
            printf("线程正在执行...\n");
            sleep(1);
    }
}

int main(int argc, char **argv)
```

```
{
    pthread_t tid;

    pthread_create( &tid, NULL, fun, NULL);

    pthread_cancel(tid);
    pthread_join(tid, NULL);

    return 0;
}
```

- 线程的属性设置

```
线程的属性结构体
pthread_attr_t 类型变量,是Linux中用于存放线程属性的结构体类型定义如下.
typedef struct
{
     int detachstate;        //线程的分离状态
     int schedpolicy;        //线程调度策略
     struct sched_param schedparam;  //线程的调度参数
     int inheritsched;       //线程的继承性
     int scope;              //线程的作用域
    size_t guardsize;       //线程栈末尾的警戒缓冲区大小
    int stackaddr_set;
    void* stackaddr;        //线程栈的位置
    size_t stacksize;       //线程栈的大小
}pthread_attr_t;

线程的属性,有很多,设置思路都一样.以设置线程分离属性为例.

```

线程的分离属性分为两种 : 可分离 与 不可分离

可分离 : 1.线程创建以后,不需要主线程进行回收(想回收也回收不了),线程结束时,会被系统自动回收.

2. 在主线程中join回收线程时,主线程将不再阻塞等待该线程.

不可分离 : 线程创建以后,需要主进程去主动回收.(线程被创建后,该线程被系统默认为不可分离的状态).

线程属性设置的基本步骤.

1.定义线程属性变量并初始化.

```
int pthread_attr_init(pthread_attr_t *attr);
```

2.根据你想设置的属性,选择对应的函数进行设置. //详情参考书籍 5.4章节 Linux 线程入门 468页

```
pthread_attr_setxxxx();

//设置线程分离属性
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

3.创建线程,使用刚才初始化好的程属性变量.填写到pthread_create()函数的第二个参数当中.

4.销毁属性变量

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

参数解析:

pthread_attr_t *attr : 为 pthread_attr_t 类型变量.

int detachstate : 宏定义.

```
PTHREAD_CREATE_DETACHED   //表示可分离
PTHREAD_CREATE_JOINABLE   //表示不可分离
```

线程属性分离示例代码:

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

```c
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <string.h>
#include <strings.h>
#include <errno.h>

#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>

#include <pthread.h>
/*
    设置线程可分离.
*/
void *fun(void *arg)
{
    sleep(1);
    printf("线程运行起来了...\n");
    pthread_exit("线程退出了..");
}

int main(int argc, char **argv)
{
    void * arg = NULL;//用于接收线程的退出值
    pthread_t tid;
    //1.定义线程属性变量 并 初始化
    pthread_attr_t myattr;
    pthread_attr_init(&myattr);
```

```
33
34      //2.调用线程设置分离属性函数,设置为可分离.
35      pthread_attr_setdetachstate(&myattr,PTHREAD_CREATE_DETACHED); //设置线程分离属性
36
37      //3.创建线程,使用刚才初始化好的程属性变量,设置为线程属性分离.
38      pthread_create( &tid, &myattr, fun, NULL);
39
40      //pthread_create( &tid, NULL, fun, NULL);  //可尝试两种写法的对比.
41
42      //回收线程
43      printf("主函数准备回收线程...\n");
44      pthread_join(tid, &arg);  //join函数本是阻塞等待线程退出后,主线程才会退出.
45                                //设置好了属性分离后,此时的join函数,不再进行阻塞.因为它没有线程可回收
46
47      printf("我回收的线程退出信息为 : %s\n", (char *)arg);
48
49      //4.销毁线程分离属性.
50      pthread_attr_destroy(&myattr);
51
52      return 0;
53  }
```

- 练习

  根据线程分离属性步骤，尝试设置线程调度策略.

互斥锁

一、前序

在我们使用单线程(main主线程)时，我们在调用函数的时候，这个程序一定是从上往下执行的，
而我们的多线程与单线程是完全不一样的，多线程与进程一样，它们是一个并发的状态。
既然是并发的状态，那么难免也会和我们的进程一样，多个任务去争夺某种临界资源。

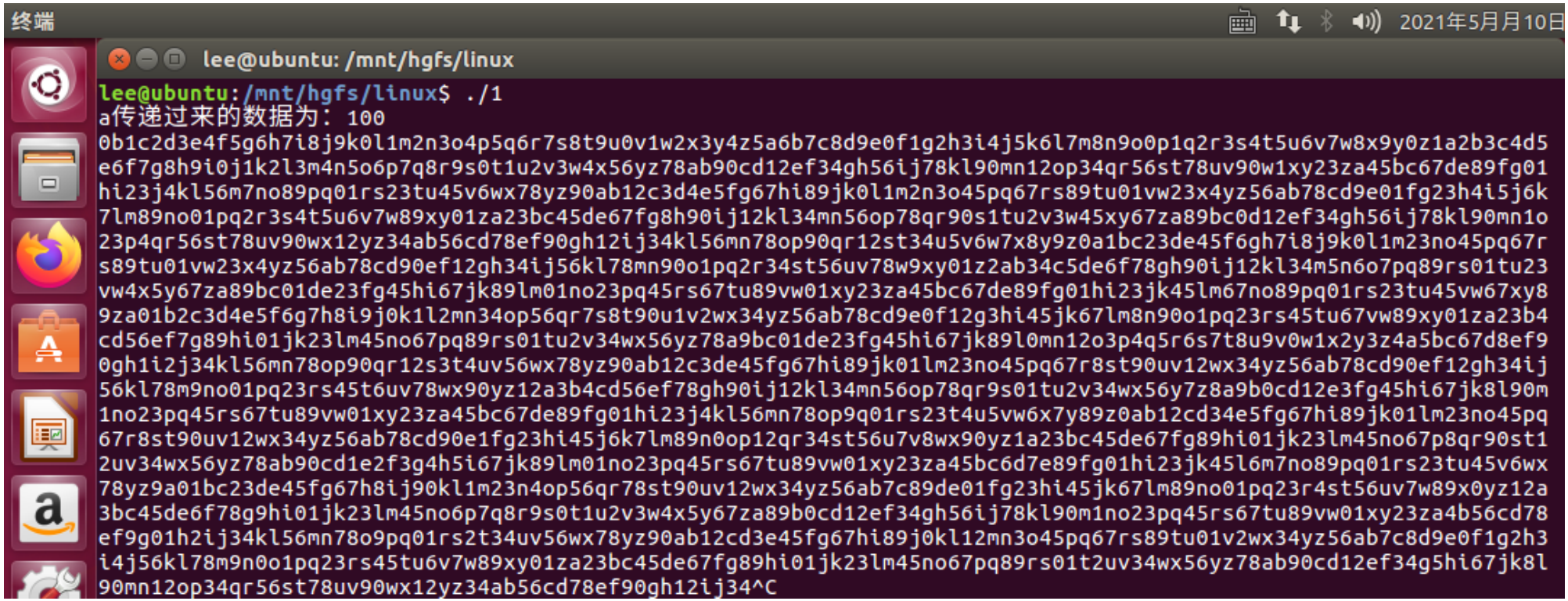
这样的话，我们就需要有互斥锁，读写锁，条件变量等相关代码，对这些临界资源进行保护。
首先我们来看一下，两个线程取争夺同一个共享资源会发生什么样的情况.

```c
//两个线程争夺屏幕资源代码.(不添加任何保护机制)
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <strings.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <pthread.h>

void *fun(void * arg) //线程任务函数    接口为固定的除了函数名 与 参数名可以改变,其他的不允许改变
{

    int a = *(int *)arg;
    printf("传递过来的数据为: %d \n", a);

```

```
    int num = '0';
    for (int i = 0; ; ++i, i %= 10)
    {
            fprintf(stderr,"%c", num + i ); //不缓冲方式输出到屏幕
            usleep(10*1000);
    }
}

int main(int argc, char const *argv[])  //执行main函数的线程 被称为主线程
{

    int a = 100;

    pthread_t thread_id;
    //1.创建一个子线程
    pthread_create(&thread_id,   //线程的ID号
                   NULL,          //表示系统标准属性,不额外设置线程属性
                   fun,           //线程将要执行的任务函数名
                   &a);           //不传递额外参数给线程

    char ch = 'a';
    for (int i = 0; ; ++i, i %= 26)
    {
            fprintf(stderr,"%c", ch+i);  //不缓冲方式输出到屏幕
            usleep(10*1000);
    }

    //等待以上线程退出,并获取退出值
    pthread_join(thread_id, NULL);

```

```
53     return 0;
54 }
```

通过上列代码运行，我们可以得到一下的结果，我们可以发现，当两条线程在争夺屏幕资源时，没有对屏幕资源进行写保护，数字与字符交错出现，它们的数据是无序的错乱的，那这样的效果显然是满足不了我们的需求。



假如，我们想要得到一些有序的数据，那么我们就需要借助 互斥锁，读写锁，条件变量等相关代码，对这些临界资源进行保护。

- **互斥锁概念**

  "锁"是一种形象的说法，就像是我们平常去银行使用自动存储机，银行里会有那种带有玻璃门，防窥防抢的自助自动存储机一样，一个自动存储机一次只允许一个人进行操作，进去了的人，门会自动锁上，外面的人只能排队等候，任何人都不可以再进入，等进去的人取完钱，打开反锁的玻璃门，下一个人才能继续去使用这个自动存储机。

那实际上相当于一个临界资源(共享资源)，在任意时刻最多只能有一个线程在访问，这样的逻辑叫 "互斥"。

```
互斥锁的操作有以下几种：

//定义一个互斥锁变量
pthread_mutex_t m;

//初始化锁，在任何线程使用该互斥锁之前都必须要先初始化
pthread_mutex_init(&m,NULL);

//访问临界资源之前 上锁
pthread_mutex_lock(&m);

//使用完临界资源后 解锁
pthread_mutex_unlock(&m);

//不需要使用后，销毁锁
pthread_mutex_destroy(&m)
```

```
//互斥锁使用示例代码
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>

```

```c
#include <errno.h>
#include <pthread.h>

//定义互斥锁变量
pthread_mutex_t m;

void output(const char *msg)
{
    while(*msg != '\0')
    {
            putc(*msg, stdout);  //不缓冲单个字符输出到屏幕.
            usleep(100);
            msg += 1;
    }
    return;
}

void *routine(void *arg)
{
    pthread_mutex_lock(&m);          //加锁
    output("info output by sub-thread.\n");
    pthread_mutex_unlock(&m);    //解锁
    pthread_exit(NULL);
}

int main(void)
{
    // 初始化
    pthread_mutex_init(&m, NULL/* 定义一个标准的互斥锁 */);

```

```
37      //创建线程
38      pthread_t tid;
39      pthread_create(&tid, NULL, routine, NULL);
40
41      // 在进入临界区之前，加锁
42      pthread_mutex_lock(&m);
43
44      // 访问了共享资源（临界资源）的代码，称为临界区代码
45      output("message delivered by main thread.\n");
46
47      // 在离开临界区的时候，解锁
48      pthread_mutex_unlock(&m);
49
50      //等待线程退出
51      pthread_join(tid, NULL);
52
53      //线程退出后 销毁锁
54      pthread_mutex_destroy(&m);
55
56      pthread_exit(NULL);
57  }
```

互斥锁使用非常简便，但是也有不适合它的场合，例如:

要保护的共享资源在绝大数的情况下是读操作，就会导致这些本来就可以一起读的线程阻塞在互斥锁上，资源得不到最大的利用，这时互斥锁会导致整个程序效率较低。

读写锁

- **概念**

  互斥锁的低效率，是因为没有更加细致的区分如何访问共享资源，无论在任何时候都只允许一条线程访问共享资源，
  而事实情况是读操作可以同时进行，不加以限制，只有写操作才需要互斥，因此如果能够根据访问的目的(读或写)，
  来分别加读锁(可以重复添加)或写锁(一次只允许操作一个)，就能极大地提高效率，尤其是在大量操作读操作的情况下，。

  读写锁的操作几乎与互斥锁一样，唯一的区别是在加锁的时候可以选择加读锁 或 写锁。

```
// 定义一个读写锁变量
pthread_rwlock_t rwlock;

// 初始化读写锁
pthread_rwlock_init(&rwlock, NULL);

//访问共享资源之前添加写锁，写锁与任何锁都是互斥的，
pthread_rwlock_wrlock(&rwlock);

//访问完毕共享资源解开写锁，不然会形成死锁，死锁也就意味着这个共享资源是无法被其他的线程访问
pthread_rwlock_unlock(&rwlock);

//在访问共享资源时,读锁是可以重复的,是不被加以限制的,可以被多个线程同时访问,所以读锁也被称为共享锁
//而且解锁时,也是可以通过解开写锁的方式 进行解锁.
pthread_rwlock_rdlock(&rwlock);

//在程序结束之前,应该销毁读写锁
pthread_rwlock_destroy(&rwlock);
```

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

// 定义了一个读写锁
pthread_rwlock_t rwlock;

// 全局变量，这是典型的共享资源
int global = 0;

void *routine1(void *arg)
{
    //============ WRITE lock ============//
    // 游戏规则：每当你要对临界资源进行写操作的时候，必须加写锁
    // 加了写锁之后，不可加任何其他的锁，写锁就是互斥锁
    pthread_rwlock_wrlock(&rwlock);
    global += 1;
    printf("I am %s, now global=%d\n", (char *)arg, global);
    sleep(1);

    // 在结束对共享资源的访问之后，一定要解锁，否则会引起死锁
    pthread_rwlock_unlock(&rwlock);
    //============ WRITE unlock ============//

    pthread_exit(NULL);
}

```

```c
void *routine2(void *arg)
{
    //============ WRITE lock ============//
    // 游戏规则：每当你要对临界资源进行写操作的时候，必须加写锁
    // 加了写锁之后，不可加任何其他的锁，写锁就是互斥锁
    pthread_rwlock_wrlock(&rwlock);
    global  = 100;
    printf("I am %s, now global=%d\n", (char *)arg, global);
    sleep(1);
    pthread_rwlock_unlock(&rwlock);
    //============ WRITE unlock ============//

    pthread_exit(NULL);
}

void *routine3(void *arg)
{
    //============ READ lock ============//
    // 游戏规则：每当你要对临界资源进行读操作的时候，必须加读锁
    // 加了读锁之后，可以加其他的读锁，读锁就是共享锁
    pthread_rwlock_rdlock(&rwlock);
    printf("I am %s, now global=%d\n", (char *)arg, global);
    sleep(1);
    pthread_rwlock_unlock(&rwlock);
    //============ READ unlock ============//

    pthread_exit(NULL);
}

int main (int argc, char *argv[])
```

```
59 {
60     // 初始化
61     pthread_rwlock_init(&rwlock, NULL);
62 
63     pthread_t t1, t2, t3;
64 
65     pthread_create(&t1, NULL, routine1, "thread 1");
66     pthread_create(&t2, NULL, routine2, "thread 2");
67     pthread_create(&t3, NULL, routine3, "thread 3");
68 
69     pthread_join(t1, NULL);
70     pthread_join(t2, NULL);
71     pthread_join(t3, NULL);
72 
73     pthread_rwlock_destroy(&rwlock);
74 
75     return 0;
76 }
77 
```

条件变量

一.前序

条件变量是在互斥锁的基础上添加了更为灵活的方法，但是条件变量是不可以单独使用，它必须要配合互斥锁进行使用。

满足某个条件的时候，条件变量可以帮助我们阻塞线程或解除线程的阻塞。

如果有这么一种情况，

我们大家都去过火车站售票处，在这个售票处会有很多售票窗口，

黄金假期时，你需要购买一张回家乡或外出的车票，这个时候的窗口售票点，会让车票数量急速下降，

而我们的铁道部门，也不会一次性的把所有的票都放出来。

假如你需要购买的票，车次是 G123, 铁道部会一次放10张票出来，然后在各个售票点进行销售。

那我们可以通过代码来进行模拟这么一种情况。

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

/*
    创建3个线程代表3个售票窗口
    票都是由铁道部分批发放,某一批票卖完了,
    窗口就要暂停售票总的票全部卖完,所有的窗口就下班
*/

// 全局变量
int alltickтnum = 100;    //表示该趟列车总票数
int curticktnum = 10;     //表示当前铁道部放出来的票数
```

```
15
16 //定义互斥锁
17 pthread_mutex_t mutex_lock;
18
19 void *win1(void *arg)
20 {
21     while(1)
22     {
23             pthread_mutex_lock(&mutex_lock);
24             curticktnum--;
25             printf("窗口1 卖出一张票，还剩下 %d 张票!\n", curticktnum);
26             sleep(1);
27             pthread_mutex_unlock(&mutex_lock);
28             usleep(100);
29     }
30
31     pthread_exit(NULL);
32 }
33
34 void *win2(void *arg)
35 {
36     while(1)
37     {
38             pthread_mutex_lock(&mutex_lock);
39             curticktnum--;
40             printf("窗口2 卖出一张票，还剩下 %d 张票!\n", curticktnum);
41             sleep(1);
42             pthread_mutex_unlock(&mutex_lock);
43             usleep(100);
44     }
```

```
45
46      pthread_exit(NULL);
47  }
48
49  void *win3(void *arg)
50  {
51      while(1)
52      {
53              pthread_mutex_lock(&mutex_lock);
54              curticktnum--;
55              printf("窗口3 卖出一张票，还剩下 %d 张票!\n", curticktnum);
56              sleep(1);
57              pthread_mutex_unlock(&mutex_lock);
58              usleep(100);
59      }
60      pthread_exit(NULL);
61  }
62
63  int main (int argc, char *argv[])
64  {
65      //创建3个线程,代表3个售票窗口
66      pthread_t t1, t2, t3;
67
68      //初始化互斥锁
69      pthread_mutex_init(&mutex_lock,NULL);
70
71
72      pthread_create(&t1, NULL, routine1, NULL);
73      pthread_create(&t2, NULL, routine2, NULL);
74      pthread_create(&t3, NULL, routine3, NULL);
```

```
75
76
77      //回收3条进程
78      pthread_join(t1, NULL);
79      pthread_join(t2, NULL);
80      pthread_join(t3, NULL);
81
82
83      return 0;
84  }
```

- **初始化、销毁条件变量**

```
1   头文件
2       #include <semaphore.h>
3   原型
4       int pthread_cond_init(pthread_cond_t *restrict cond,
5                             const pthread_condattr_t *restrict attr);
6       int pthread_cond_destroy(pthread_cond_t *cond);
7
8
9   参数解析:
10      pthread_cond_t *restrict cond : 条件变量
11      const pthread_condattr_t *restrict attr : 条件变量的属性，一般始设置为NULL,系统默认属性.
12
13  返回值:
14      成功 : 0
15      失败 : -1
16
```

```
#注意点
    跟其他的同步互斥机制一样，条件变量的开始使用之前也必须初始化。
    初始化函数中的属性参数 attr一般不使用，设置为NULL即可。
    当使用pthread_cond_destroy()销毁一个条件变量之后,他的值变得不确定,再使用必须重新初始化。

```

- **使用条件变量阻塞某条线程 解开阻塞（类似于上锁，解锁）**

```
头文件
#include <semaphore.h>
原型
        //阻塞一条线程
        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);    //只唤醒某个等待中的线程。解除线程的阻塞
        int pthread_cond_broadcast(pthread_cond_t *cond); //唤醒全部的线程

参数解析:
        pthread_cond_t *restrict cond   : 条件变量
        pthread_mutex_t *restrict mutex : 互斥锁
        const struct timespec *restrict abstime : 超时时间限制


返回值
    成功 : 0
```

```
    失败 : -1;

    pthread_cond_wait()与pthread_cond_timedwait()两个函数功能是一样的，
    区别是pthread_cond_timedwait()可以设置超时时间。

    着重要注意的是:
        pthread_cond_wait()，互斥锁mutex将会立即被解锁,然后该线程才会进入到阻塞状态.

    注意:
    被唤醒的线程并不能立即从pthread_cond_wait()中返回，而是必须要先获得配套的互斥锁。
```

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

/*
   pthread_cond_wait() 函数特点分析代码.
*/

// 全局变量
int allticktnum = 100;    //表示该趟列车总票数
int curticktnum = 10;     //表示当前铁道部放出来的票数

//定义互斥锁
pthread_mutex_t mutex_lock;

//定义条件变量
```

```c
pthread_cond_t Cond;


void *win1(void *arg)
{
    while(1)
    {
            pthread_mutex_lock(&mutex_lock);
            curticktnum--;
            printf("窗口1 卖出一张票，还剩下 %d 张票!\n", curticktnum);

            if (curticktnum <= 0) //当目前的票数小于等于0时,暂停卖票
            {
                    printf("当前票数已售完,窗口1 阻塞等待相关部门放票...\n");
                    pthread_cond_wait(&Cond, &mutex_lock);
            }
            sleep(1);
            pthread_mutex_unlock(&mutex_lock);
            usleep(100);
    }

    pthread_exit(NULL);
}

void *win2(void *arg)
{
    while(1)
    {
            pthread_mutex_lock(&mutex_lock);
            curticktnum--;
```

```
48              printf("窗口2 卖出一张票，还剩下 %d 张票!\n", curticktnum);
49              if (curticktnum <= 0) //当目前的票数小于等于0时,暂停卖票
50              {
51                      printf("当前票数已售完,窗口2 阻塞等待相关部门放票...\n");
52                      pthread_cond_wait(&Cond, &mutex_lock);
53              }
54              sleep(1);
55              pthread_mutex_unlock(&mutex_lock);
56              usleep(100);
57      }
58
59      pthread_exit(NULL);
60  }
61
62  void *win3(void *arg)
63  {
64      while(1)
65      {
66              pthread_mutex_lock(&mutex_lock);
67              curticktnum--;
68              printf("窗口3 卖出一张票，还剩下 %d 张票!\n", curticktnum);
69              if (curticktnum <= 0) //当目前的票数小于等于0时,暂停卖票
70              {
71                      printf("当前票数已售完,窗口1 阻塞等待相关部门放票...\n");
72                      pthread_cond_wait(&Cond, &mutex_lock);
73              }
74              sleep(1);
75              pthread_mutex_unlock(&mutex_lock);
76              usleep(100);
77      }
```

```
78      pthread_exit(NULL);
79  }
80
81  int main (int argc, char *argv[])
82  {
83      //创建3个线程,代表3个售票窗口
84      pthread_t t1, t2, t3;
85
86      //初始化互斥锁 mutex_lock , 设置为为系统默认属性
87      pthread_mutex_init(&mutex_lock,NULL);
88
89      //初始化条件变量Cond, 设置为为系统默认属性
90      pthread_cond_init(&Cond,NULL);
91
92      pthread_create(&t1, NULL, routine1, NULL);
93      pthread_create(&t2, NULL, routine2, NULL);
94      pthread_create(&t3, NULL, routine3, NULL);
95
96      //回收3条进程
97      pthread_join(t1, NULL);
98      pthread_join(t2, NULL);
99      pthread_join(t3, NULL);
100
101     return 0;
102 }
103
```

作业：

1.使用线程实现双向通信.(消息队列).

2.使用线程实现文件拷贝，要求使用目录检索，去寻找一个指定的文件，

在主线程中通过线程的参数传递给 线程1，

线程1，获取到参数，把获取到的数据，写入到TXT文本当中.