多线程及其互斥与同步

课程目标

- 理解线程及其生命周期的含义。
- 理解linux线程管理、调度的基本含义。
- 掌握并理解多线程的相关函数。
- 掌握并理解互斥锁的相关函数。
- 掌握并理解信号量的相关函数。
- 掌握多线程的同步和互斥的模型。

课程实践

- 创建多线程程序。
- 利用互斥锁完成多线程间临界资源保护。
- 利用信号量实现多线程同步使用竞争资源。

多线程

理解线程

- 进程具有很好的独立性，每个进程都有独立的虚拟内存空间。
- 进程在其生命周期内与其他进程并行运行，接受操作系统的同一调度。

- 由于进程的地址空间是私有的，因此在进程间上下文切换时，系统开销比较大（恢复进程挂起时的各个寄存器的值，装载虚拟内存映射表）。

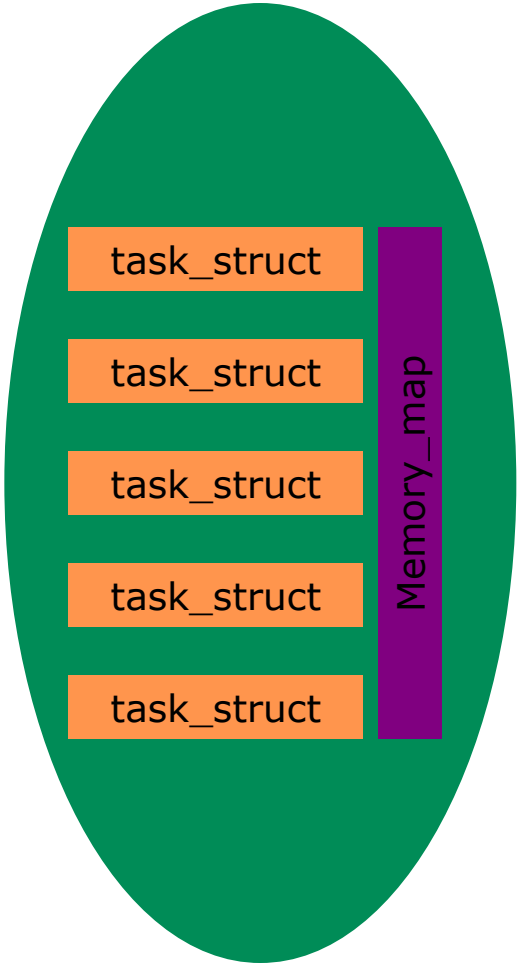
理解线程

- 为了提高系统的性能，许多操作系统规范里引入了轻量级进程的概念，也被称为线程。
- 在同一个进程中创建的线程共享该进程的地址空间。
- Linux里同样用task_struct来描述一个线程，线程和进程都参与统一的调度。

- 历史上fork主要为了竞争CPU，随着线程的引入，fork自己基本没有什么意思，所以一般用来和exec一起使用创建新的程序。
- 多线程担负起了竞争CPU的角色。

理解线程

- 进程逐渐演化成为一个静态的概念，只是为线程提供一个运行的环境，包含各种安全标识符，内存使用等。
- 线程是运行的实体，是CPU调度的基本单位。
- 一个进程至少要有一个线程存活，如果所有线程都退出了，进程也会退出。
- main函数为被当作主线程，如果主线程结束，其他线程也会被结束。



理解线程

- 一个进程中的多个线程共享以下资源
  - 代码段，所有的执行代码。
  - 数据段，所有的静态变量，部分局部变量。
  - 安全描述符，（例如用户，权限、umask）
  - 环境变量，（例如：当前工作目录）
- 一个进程中的多个线程独享以下资源
  - 线程的栈空间
  - 部分局部变量

理解线程

- 线程在代码编写方面看：其本质是一个函数，只是线程函数可以与main函数能并行运行。
- main函数只需用一种特别的方法将函数运行起来，而不需要等待函数的返回。

创建一个线程

所需头文件	**#include <pthread.h>**
函数原型	**int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void * (* routine)(void *), void *arg)**
函数参数	**thread：返回创建的线程的id**
	attr：指定线程的属性，NULL表示使用缺省属性
	routine：线程执行的函数
	arg: 传递给线程执行的函数的参数
函数返回值	**成功：0**
	出错：返回错误号

创建一个线程

- pthread_create将一个函数作为一个线程执行。
- 第三个参数是线程函数的名称，函数的返回值是void *，参数也是void *。
- 第四个参数是线程函数的参数，只能是一个指针类型，一般将需要传给函数的局部变量打包成一个结构体，将结构体的地址传给线程函数。

实验1

- 1、创建一个线程，主线程打印1000个“*”,其他线程打印1000个“-”；观察并理解线程函数并行运行。
- 2、创建一个线程，主线程打印100个“*”,其他线程打印1000个“-”；观察并理解主线程退出，从线程也退出。
- 3、将实验1.2改造一下，由主线程通过参数传递给子线程需要打印“-”的个数，学会使用参数传递。
- 4、创建一个多线程程序，打开一个文件，主线程在文件的奇数位写“*”,子线程在偶数位写“-”，各写1000个。

```c
void * fun()
{
    int i;
    for (i = 0; i < 1024; ++i)
    {
        printf("*");
        fflush(0);
        //usleep(1);
    }
    return NULL;
}
```

```c
int main()
{
    pthread_t tid;
    int i;
    pthread_create(&tid, NULL, fun, NULL);
    for (i = 0; i < 1024; ++i)
    {
        printf("-");
        fflush(0);
        //usleep(1);
    }
    sleep(1);
    return 0;
}
```

线程退出

- 1、主线程和子线程并行运行，如何得知子线程结束？
- 2、主线程能结束子线程吗？
- 3、线程函数调用的子函数能结束线程吗？
- 4、进程main函数调用的子函数能结束进程吗？

- 1、线程函数执行return。
- 2、线程崩溃，系统强制结束线程。
- 3、主线程写封修书，结束子线程。

等待一个线程退出

所需头文件	#include <pthread.h>
函数原型	int pthread_join(pthread_t thread, void **value_ptr)
函数参数	thread：要等待的线程
	value_ptr：指针*value_ptr指向线程返回的参数
函数返回值	成功：0
	出错：返回错误号

线程退出

所需头文件	#include <pthread.h>
函数原型	void pthread_exit(void *value_ptr)
函数参数	value_ptr：线程退出时返回的值

所需头文件	#include <pthread.h>
函数原型	int pthread_cancel(pthread_t thead)
函数参数	thread：要取消的线程
函数返回值	成功：0
	出错：返回错误号

多线程编程

```
pthread_create(&tid, NULL, fun, NULL);   // 创建线程
```

- tid: 用来保存线程的ID
- fun: 线程函数

sublinme要为编译增加-lpthread选项

运行：（记事本打开后编辑第二行，大括号是第一行）

```
gedit /home/ziyang/.config/sublime-text-3/Packages/User/gcc.sublime-build
```

改前：

```
"shell_cmd": "gcc \"${file}\" -o \"${file_path}/${file_base_name}\" -O2 -Wall -lm",
```

改后：

```
{
    "shell_cmd": "gcc \"${file}\" -o \"${file_path}/${file_base_name}\" -O2 -Wall -lm -lpthread",
```

编辑后Ctrl+s保存，点击左上角的关闭图标退出gedit

---

ziongz

实验2

- 1、将实验1.2改造正确。
- 2、编写一个多线程程序，在线程子函数中结束线程。
- 3、编写一个多线程程序，子线程得到用户的输入，主线程倒计时，如果十秒钟用户没有输入“I love you”，就打印“都老了，没戏了！”，否则打印“结婚吧，生娃吧！”，10秒钟后要结束输入线程。

- 多线程除了竞争CPU外，另一个常用的用途是处理阻塞事件。一个函数阻塞了，用另外一个线程做其他事情，不至于让整个进程停在那里。
- 基本每个阻塞的函数都有一种非阻塞的用法，但自从有了多线程，非阻塞的用法不再必须了，可以用阻塞函数监视事件变化，其他线程继续工作，不需要循环监视事件了。

线程属性

所需头文件	#include <pthread.h>
函数原型	int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void * (* routine)(void *), void *arg)
函数参数	thread：创建的线程
	attr：指定线程的属性，NULL表示使用缺省属性
	routine：线程执行的函数
	arg: 传递给线程执行的函数的参数
函数返回值	成功：0
	出错：返回错误号

线程属性

在一般的多线程程序中，pthread_create的第三个参数pthread_attr_t *attr设为空，即使用系统默认属性即可。

在一些特殊情况下，我们需要设置这个参数。这个参数可设置子线程的一些属性：

线程属性

```
typedef struct
{
    int                 detachstate;    线程的分离状态
    int                 schedpolicy;    线程调度策略
    structsched_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;
```

线程属性

- 第一步：pthread_attr_t attr;
- 第二步：pthread_attr_init(&attr)
- 第三步：修改属性attr中默认的内容：
  - 设置：pthread_attr_setXXXX(); xxxx替换成属性元素名
  - 获取：pthread_attr_getXXXX(); xxxx替换成属性元素名
    - 例如：pthread_attr_getscope、pthread_attr_setscope
    - pthread_attr_getstacksize、pthread_attr_setstacksize
    - pthread_attr_getschedparam、pthread_attr_setschedparam
    - pthread_attr_getdetachstate、pthread_attr_setdetachstate
- 第四步：创建线程pthread_create(&thid, &attr, fun, &para)。
- 第五步： 销毁属性pthread_attr_destroy(&attr);

线程分离

- 线程的分离状态决定一个线程以什么样的方式来终止自己。在默认情况下线程是非分离状态的，这种情况下，原有的线程等待创建的线程结束。只有当pthread_join()函数返回时，创建的线程才算终止，才能释放自己占用的系统资源。
- 而分离线程运行结束了，线程也就终止了，马上释放系统资源。

思考：线程的分离和进程的分离的区别与联系，子进程如何与父进程分离呢？

线程分离

- 实际工作中，线程的大部分属性都可以系统默认的属性即可。
- 在守护进程等长时间运行的服务进程中，特别是需要不断创建线程的进程中，一般都会设置线程的分离属性，否则主线程要等待每一个线程的退出、手动负责回收内存。
- 所以线程分离是线程属性中唯一一个需要重点关注的属性。

线程分离

- 线程分离可以使用设置线程的属性的函数pthread_attr_setdetachstate（五步）
- 系统开发库给我们提供了一个更便捷的函数实现该功能：

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

- Man手册的例子：

```
pthread_detach(pthread_self());
```

实验3

- 1、每秒创建一个线程（使用默认属性），使用top –p查看进程的内存变化，（如果改为每毫秒创建一个线程呢？）
- 2、将上个实验的线程变成分离的，再观察内存情况。

- 如果一个程序要长时间运行，一定要管好自己的内存，内存泄露不能有，子进程、线程要分离，做好内存回收。（内存回收是java虚拟机的技术瓶颈，也是它无法取代C语言的原因之一）

互斥锁

线程互斥锁

- 有些时候，多线程不能同时使用某个资源，特定时刻只能有一个线程使用该资源，那么就需要一个变量，解决资源使用的问题，比如：12306买火车票选座位、聊天室里显示发言内容。
- 互斥锁保证同一时刻只有一个线程可以问该对象。

- 思考：观察下我们实验1.4生成的文件对吗，如果不对，为什么？

线程互斥锁

- 实验1.4、创建一个多线程程序，打开一个文件，主线程在文件的奇数位写“*”,子线程在偶数位写“-”，各写1000个。
- 主线程：　　子线程：

```
while(1){
    lseek(fd, i+1, SEEK_SET);
    if( 1 != write(fd, ".", 1)){
        perror("write");
    }
    i += 2;
    if(i > 10000)break;
}
pthread_join(thid, &retval);
return 0;
```

线程调度

```
void * thread_a(void * para)
{
    int i = 0;
    while(1){
        lseek(fd, i, SEEK_SET);
        if( 1 != write(fd, "*", 1)){
            perror("write");
        }
        i += 2;
        if(i > 10000)break;
    }
}
```

zionyz

线程互斥锁

- 使用hd newfile查看文件内容，写1000次会发现由于这种调度而产生错误的概率相对大！

```
ziyang@ubuntu:~/Documents/5-2$ hd testfile
00000000  2a 2e 2a 2e 2a 2e 2a 2e  2a 2e 2a 2e 2a 2e 2a 2e  |*.*.*.*.*.*.*.*.|
*
000001b0  2a 2e 2a 2e 2a 2e 2a 2e  2a 2e 2a 2e 2a 00 2a 2e  |*.*.*.*.*.*.*.*.|
000001c0  2a 2e 2a 2e 2a 2e 2a 2e  2a 2e 2a 2e 2a 2e 2a 2e  |*.*.*.*.*.*.*.*.|
*
000002b0  2a 2e 2a 2e 2a 2e 2a 2e  2e 2e 2a 2e 2a 2e 2a 2e  |*.*.*.*...*.*.*.|
000002c0  2a 2e 2a 2e 2a 2e 2a 2e  2a 2e 2a 2e 2a 2e 2a 2e  |*.*.*.*.*.*.*.*.|
*
00001db0  2a 2e 2a 2e 2a 2e 2a 2e  2a 2e 2a 2e 2a 2e 2a 00  |*.*.*.*.*.*.*.*.|
00001dc0  2a 2e 2a 2e 2a 2e 2a 2e  2a 2e 2a 2e 2a 2e 2a 2e  |*.*.*.*.*.*.*.*.|
*
000020d0  2a 2e 2a 2e 2a 2e 2e 2e  2a 2e 2a 2e 2a 2e 2a 2e  |*.*.*...*.*.*.*.|
000020e0  2a 2e 2a 2e 2a 2e 2a 2e  2a 2e 2a 2e 2a 2e 2a 2e  |*.*.*.*.*.*.*.*.|
*
00002712
```

互斥锁

- 引入互斥(mutual exclusion)锁的目的是用来保证共享数据操作的完整性。
- 互斥锁主要用来保护临界资源
- 每个临界资源都由一个互斥锁来保护，任何时刻最多只能有一个线程能访问该资源
- 线程必须先获得互斥锁才能访问临界资源，访问完资源后释放该锁。如果无法获得锁，线程会阻塞直到获得锁为止

互斥锁

- 1、定义一个互斥锁
- 2、初始化一个互斥锁
- 3、竞争占用互斥锁
- 4、释放互斥锁资源
- 5、销毁互斥锁

初始化互斥锁

所需头文件	**#include <pthread.h>**
函数原型	**int pthread_mutex_init(pthread_mutex_t *mutex,** **pthread_mutexattr_t *attr)** **// 初始化互斥锁**
函数参数	**mutex：互斥锁** **attr：互斥锁属性 // NULL表示缺省属性**
函数返回值	**成功：0** **出错：-1**

占用和释放

所需头文件	#include <pthread.h>
函数原型	int pthread_mutex_lock(pthread_mutex_t *mutex) // 申请互斥锁
函数参数	mutex：互斥锁
函数返回值	成功：0
	出错：返回错误号

所需头文件	#include <pthread.h>
函数原型	int pthread_mutex_unlock(pthread_mutex_t *mutex) // 释放互斥锁
函数参数	mutex：互斥锁
函数返回值	成功：0
	出错：返回错误号

```c
pthread_mutex_t mutex;
int pthread_mutex_init(&mutex, NULL);

pthread_mutex_lock(&mutex);
pthread_mutex_unlock(&mutex);

int pthread_mutex_destroy(&mutex);
```

实验4

- 1、将实验1.4修改正确（经典的临界保护，互斥）。
  - 实验1.4、创建一个多线程程序，打开一个文件，主线程在文件的奇数位写“*”,子线程在偶数位写“-”，各写1000个。

- 2、将实验1.1中的多线程混乱的打印，改造成线程交替打印（经典的相互控制模型，同步）
  - 实验1.1、创建一个线程，主线程打印1000个“*”,其他线程打印1000个“-”；观察并理解线程函数并行运行。

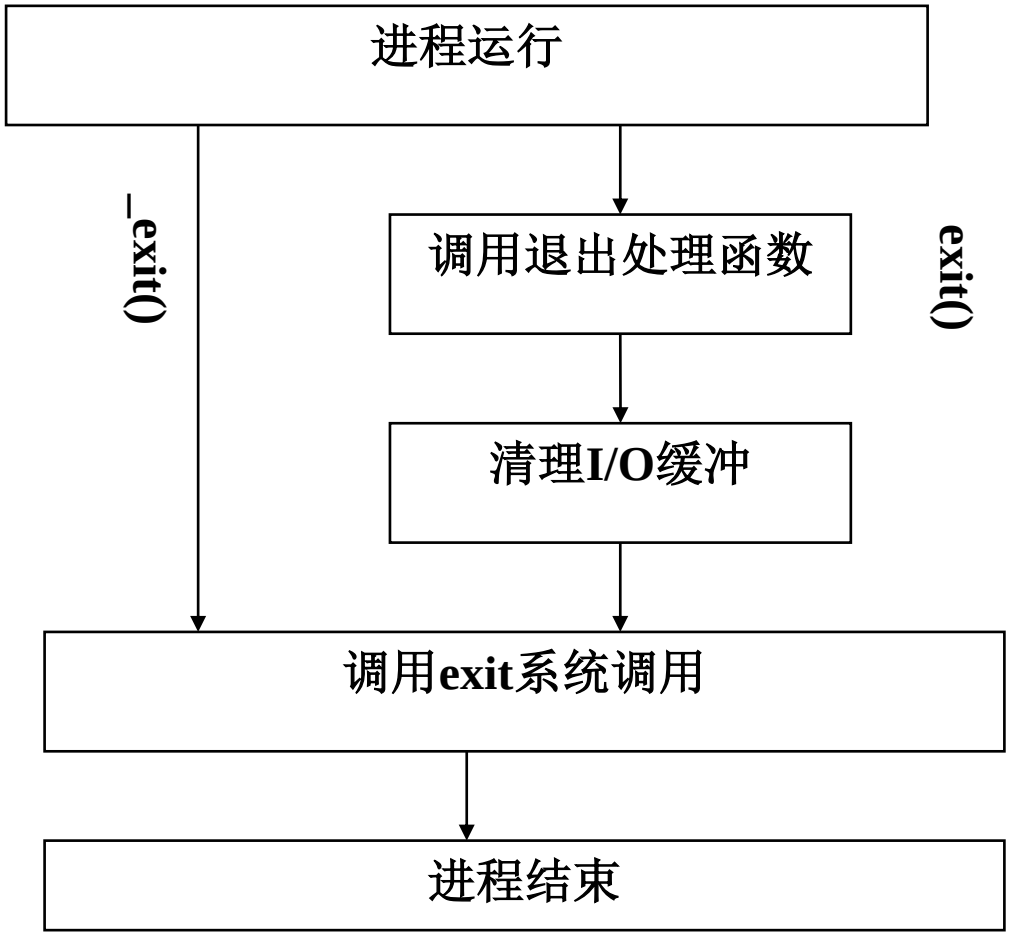
进程退出函数

- 线程中pthread_exit函数可以在任何函数中退出子线程。
- 进程中也存在一些函数可以在任何的子函数中退出整个进程：exit()和_exit()。

exit和_exit

所需头文件	exit：#include <stdlib.h>
	_exit：#include <unistd.h>
函数原型	exit：void exit(int status);
	_exit：void _exit(int status);
函数传入值	status是一个整型的参数，可以利用这个参数传递进程结束时的状态。通常0表示正常结束；其他的数值表示出现了错误，进程非正常结束。在实际编程时，可以用wait系统调用接收子进程的返回值，进行相应的处理。

exit和_exit



exit和_exit

- _exit()函数的作用最为简单：直接使进程终止运行，清除其使用的内存空间，并销毁其在内核中的各种数据结构；
- exit()函数则在这些基础上作了一些包装，在执行退出之前加了若干道工序。
- exit()函数在调用exit系统调用之前要检查文件的打开情况，把文件缓冲区中的内容写回文件，就是图中的"清理I/O缓冲"一项。
- 思考：exit， pthread_exit，return的区别和联系。

实验5

- 1、用exit在非main里面退出整个程序。

信号量

信号量

- 多线程共享同一个进程的地址空间

- 优点：线程间很容易进行通信
  - 通过全局变量实现数据共享和交换
- 缺点：多个线程同时访问共享对象时需要引入同步和互斥机制
  - 互斥锁，只有两种状态
  - 信号量，可以定义多种状态。

信号量

- 信号量代表某一类资源，其值表示系统中该资源的数量
- 信号量是一个受保护的变量，只能通过三种操作来访问
  - 初始化
  - P操作(申请资源)
  - V操作(释放资源)
- 信号量的值为非负整数

信号量

- P(S)含义如下:

```
if (信号量的值大于0) {
        申请资源的任务继续运行;
        信号量的值减一;
}
else {  申请资源的任务阻塞;  }
```

- V(S)含义如下:

```
if (没有任务在等待该资源) {  信号量的值加一;  }
else {  唤醒第一个等待的任务，让其继续运行}
```

信号量

- P、V操作在互斥锁上的实现为pthread_mutex_lock、pthread_mutex_unlock。
- P、V操作在信号量上的实现为sem_wait、sem_post。

信号量

- sem_init()
- sem_wait()/sem_trywait()
- sem_post()
- sem_destroy()
- sem_open()
- sem_close()

信号量

所需头文件	**#include <semaphore.h>**
函数原型	**int sem_init(sem_t *sem, int pshared, unsigned int value)** **// 初始化信号量**
函数参数	**sem：初始化的信号量**
	pshared：信号量共享的范围(0: 线程间使用 非0:进程间使用)
	value：信号量初值
函数返回值	**成功：0**
	出错：-1

信号量

所需头文件	#include <semaphore.h>
函数原型	int sem_wait(sem_t *sem) // P操作
函数参数	sem：信号量
函数返回值	成功：0
	出错：-1

所需头文件	#include <semaphore.h>
函数原型	int sem_post(sem_t *sem) // V操作
函数参数	sem：信号量
函数返回值	成功：0
	出错：-1

信号量

- **int sem_init(sem_t *sem, int pshared, unsigned int value)**
- **如果 pshared 设为0，即线程间使用；并且value设为1，那么信号量就互斥锁很像。**
- **sem_wait类似lock操作，sem_post类似unlock操作。**

- **信号量的实质就是一个类似整数型的全局变量，可通过sem_getvalue ()得到这个值。**
- **sem_wait()类似于：（实际实现要复杂的多）**

```
while(sem == 0){
    usleep(1);
}
sem--;
//do something
```

实验6

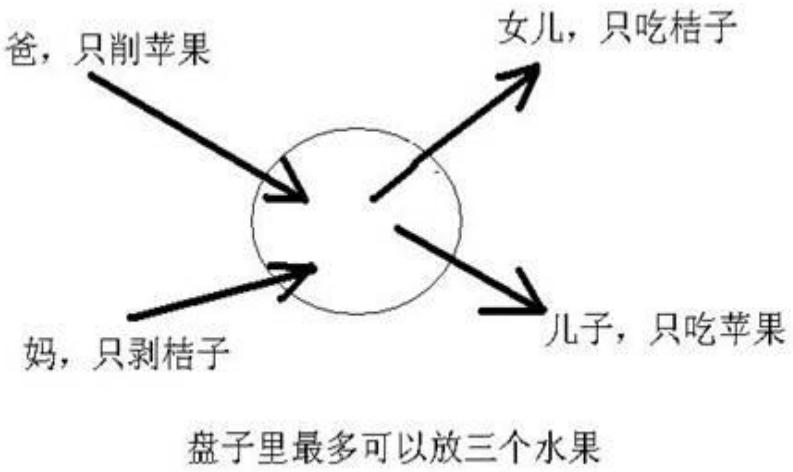
- 1、将实验1. 4用信号量（经典的临界保护，互斥）。
  - 实验1.4、创建一个多线程程序，打开一个文件，主线程在文件的奇数位写“*”,子线程在偶数位写“-”，各写1000个。

- 2、将实验1.1中的多线程混乱的打印，改造成用信号量控制线程交替打印（经典的相互控制模型，同步）
  - 实验1.1、创建一个线程，主线程打印1000个“*”,其他线程打印1000个“-”；观察并理解线程函数并行运行。

信号量

- value大于1的时候处理竞争资源的使用。

- 思考：如何设计信号量使用模型，处理上图的资源竞争问题？写出伪代码逻辑。
  - S1，盘子里空间容量，S1 = 3
  - S2，盘子里桔子的数量，S2 = 0
  - S3，盘子里苹果的数量，S3= 0
  - 4个人用四个线程表示。



一般用于硬件资源的分配，如内存，硬件解码器、硬件解密模块，如wifi设备最大接入设备数的限制，网络并发数量限制。

回顾

- 多线程创建、等待、取消、分离，线程属性。
- 互斥锁初始化、lock、unlock、destory。
- 信号量初始化，P、V操作、destroy
- 经典的互斥模型。
- 经典的同步模型。