# 实验五 学习多线程（Pthread）

## 1实验目的

* 需要理解线程概念
* 理解多线程与多进程的区别
* 理解生产者消费者问题
* 掌握Pthread提供的同步机制Pthread\_mutex

## 2实验内容

主要引入线程的概念，理解线程的相关概念，体会多进程与多线程的区别，然后利用Pthread提供的同步机制Pthread\_mutex实现1对1的生产者-消费者。

## 3实验原理

3.1线程介绍

操作系统引入进程的目的是为了描述和实现多个进程的并发执行，以改善资源利用率，提高系统的吞吐量。而进程作为资源的拥有者和系统调度的对象，需要花费系统较大的额外开销。所以系统中同时存在的进程数据不宜过多，进程切换的频率也不宜过高，这也就限制了并发度的进一步提高。因此，就引入了线程的概念。线程的引入可以减少程序并发执行时系统所付出的额外开销，使操作系统具有更好的并发性。

进程是一个拥有资源的独立单位同时也是一个可以独立调度的基本单位。引入了线程之后，将进程的资源申请和调度属性分开了，进程是拥有资源的独立单位，而线程则是资源调度的基本单位。线程自身基本上不拥有资源，只拥有少许运行必不可少的私有资源。线程可与同属于一个进程的其他线程共享进程的同步资源。

进程中的所有线程共享进程的状态。线程具有三种基本状态就绪、执行和阻塞。一个进程可以创建和撤销一个或多个线程，同一进程中的多个线程可以并发执行。

线程属性主要包括：作用域（scope）、栈尺寸（stack size）、栈地址（stack address）、优先级（priority）、分离的状态（detached state）、调度策略和参数（scheduling policy and parameters）。默认的属性为非绑定、非分离、缺省1M的堆栈、与父进程同样级别的优先级。

3.2用户线程与内核线程

用户级线程：用户级线程的创建、撤销及切换等操作全部由支持线程的应用程序完成，内核并不知道线程的存在。用户级线程的优点是：线程的调度不需要内核的直接参与，控制简单；可以在不支持线程的操作系统中实现；创建销毁的代价比内核级线程小；允许用户定义自己的调度算法，管理比较灵活等。缺点是资源调度是按照进程进行，多个处理机下，同一个进程中的线程只能在同一处理机下分时复用。

内核级线程：内核级线程的管理完全由系统内核完成，应用程序无权进行线程切换等操作，系统为应用程序提供相应的api。内核级线程的优点是：当有多个处理机时，一个进程的多个线程可以同时进行。缺点是：必须由内核进行调度。

用户级线程与内核级线程的区别：

* 1. 内核支持线程是OS内核可感知的，而用户级线程是OS内核不可感知的。
  2. 用户级线程的创建、撤消和调度不需要OS内核的支持，是在语言（如Java）这一级处理的；而内核支持线程的创建、撤消和调度都需OS内核提供支持，而且与进程的创建、撤消和调度大体是相同的。
  3. 用户级线程执行系统调用指令时将导致其所属进程被中断，而内核支持线程执行系统调用指令时，只导致该线程被中断。
  4. 在只有用户级线程的系统内，CPU调度还是以进程为单位，处于运行状态的进程中的多个线程，由用户程序控制线程的轮换运行；在有内核支持线程的系统内，CPU调度则以线程为单位，由OS的线程调度程序负责线程的调度。
  5. 用户级线程的程序实体是运行在用户态下的程序，而内核支持线程的程序实体则是可以运行在任何状态下的程序。

3.3多线程与多进程

**多线程的优点：**

1. 无需跨进程边界；
2. 程序逻辑和控制方式简单；
3. 所有线程可以直接共享内存和变量等；
4. 线程方式消耗的总资源比进程方式好；

**多线程的缺点：**

1. 每个线程与主程序共用地址空间，受限于2GB地址空间；
2. 线程之间的同步和加锁控制比较麻烦；
3. 一个线程的崩溃可能影响到整个程序的稳定性；
4. 到达一定的线程数程度后，即使再增加CPU也无法提高性能；
5. 线程能够提高的总性能有限，而且线程多了之后，线程本身的调度比较麻烦，需要消耗较多的CPU；

**多进程的优点：**

1. 每个进程互相独立，不影响主程序的稳定性，子进程崩溃没关系；
2. 通过增加CPU，就可以容易扩充性能；
3. 可以尽量减少线程加锁/解锁的影响，极大提高性能，就算是线程运行的模块算法效率低也没关系；
4. 每个子进程都有2GB地址空间和相关资源，总体能够达到的性能上限非常大；

**多进程的缺点：**

1. 逻辑控制复杂，需要和主程序交互；
2. 需要跨进程边界，如果有[大数据](http://lib.csdn.net/base/20)量传送，就不太好，适合小数据量传送、密集运算；
3. 多进程调度开销比较大；

3.4线程的同步与互斥

线程共享进程的资源和地址空间，对这些资源进行操作时，必须考虑线程间同步与互斥问题，线程的同步机制主要包括三种：互斥锁、信号量、条件变量。这其中，互斥锁更适合同时可用的资源是唯一的情况。信号量更适合同时可用的资源为多个的情况。

（1）互斥锁：用简单的加锁方法控制对共享资源的原子操作，只有两种状态: 上锁、解锁两个操作。可把互斥锁看作某种意义上的全局变量，在同一时刻只能有一个线程掌握某个互斥锁，拥有上锁状态的线程能够对共享资源进行操作。若其他线程希望上锁一个已经被上锁的互斥锁,则该线程就会挂起，直到上锁的线程释放掉互斥锁为止。互斥锁保证让每个线程对共享资源按顺序进行原子操作。

互斥锁分类区别在于其他未占有互斥锁的线程在希望得到互斥锁时是否需要阻塞等待：

快速互斥锁：调用线程会阻塞直至拥有互斥锁的线程解锁为止。默认为快速互斥锁。

检错互斥锁：为快速互斥锁的非阻塞版本，它会立即返回并返回一个错误信息。

（2）信号量：操作系统中所用到的PV原子操作，广泛用于进程或线程间的同步与互斥。信号量本质上是一个非负的整数计数器，被用来控制对公共资源的访问。

PV原子操作：对整数计数器信号量sem的操作：一次P操作使sem减一，而一次V操作使sem加一。进程（或线程）根据信号量的值来判断是否对公共资源具有访问权限。当信号量sem的值大于等于零时，该进程（或线程）具有公共资源的访问权限。当信号量sem的值小于零时，该进程（或线程）就将阻塞直到信号量sem的值大于等于0为止。

PV操作主要用于线程间的同步和互斥：

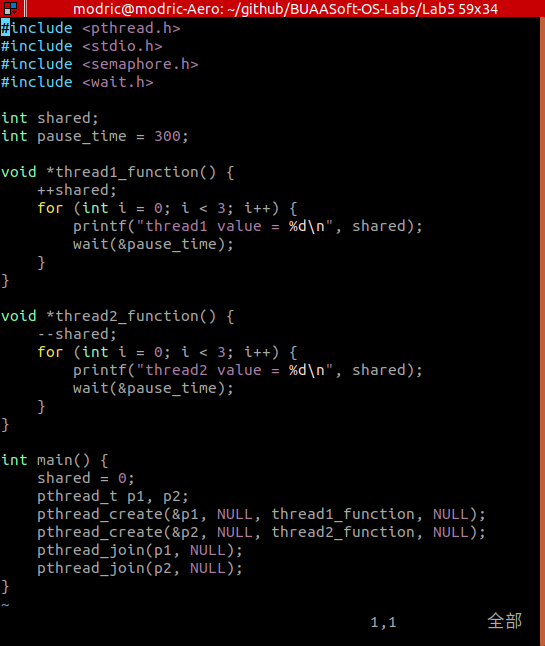
　　互斥，几个线程只设置一个信号量sem

　　同步，会设置多个信号量，安排不同初值来实现它们之间的顺序执行

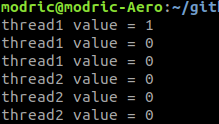
## 4实验步骤

4.1用pthread\_create()新生成一个线程 。实验两个线程间共享全局变量，也就是用全局变量通信。

代码如下：

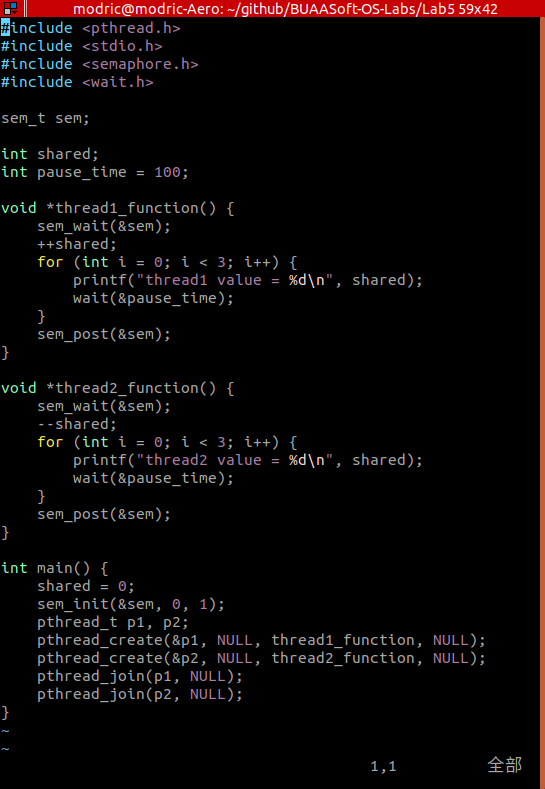


结果如下：

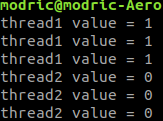


可以看出，未加锁是危险的，可能会导致全局变量在使用过程中被改变

4.2尝试使用匿名信号量加锁



输出结果：

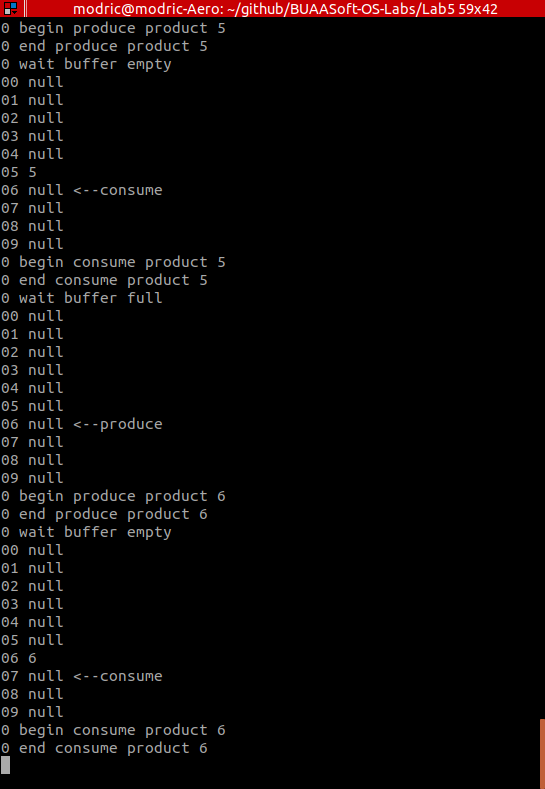


加锁以后，全局变量的值是安全的，不会被其他线程改变。

4.3生产者消费者问题代码实现

#include **<stdio.h>**#include **<stdlib.h>**#include **<string.h>**#include **<unistd.h>**#include **<pthread.h>**#include **<semaphore.h>**#define CONSUMERS\_COUNT 1  
#define PRODUCERS\_COUNT 1  
#define BUFFSIZE 10  
  
**int** g\_buffer[BUFFSIZE];  
  
**unsigned short** in = 0;  
**unsigned short** out = 0;  
**unsigned short** produce\_id = 0;  
**unsigned short** consume\_id = 0;  
  
sem\_t g\_sem\_full;  
sem\_t g\_sem\_empty;  
pthread\_mutex\_t g\_mutex;  
  
pthread\_t g\_thread[CONSUMERS\_COUNT + PRODUCERS\_COUNT];  
  
**void** \*consume(**void** \*arg) {  
 **int** inx = \*((**int** \*) arg);  
 free(arg);  
 **int** i;  
 **while** (1) {  
 printf(**"%d wait buffer empty\n"**, inx);  
 sem\_wait(&g\_sem\_empty);  
 pthread\_mutex\_lock(&g\_mutex);  
 *//消费产品  
 //打印仓库当前状态* **for** (i = 0; i < BUFFSIZE; ++i) {  
 printf(**"%02d "**, i);  
 **if** (-1==g\_buffer[i]) {  
 printf(**"null"**);  
 } **else** {  
 printf(**"%d "**, g\_buffer[i]);  
 }  
 **if** (i==in) {  
 printf(**"\t<--consume"**);  
 }  
 printf(**"\n"**);  
 }  
 *//消费产品* consume\_id = g\_buffer[out];  
 printf(**"%d begin consume product %d\n"**, inx, consume\_id);  
 g\_buffer[out] = -1;  
 out = (out + 1)%BUFFSIZE;  
 printf(**"%d end consume product %d\n"**, inx, consume\_id);  
 pthread\_mutex\_unlock(&g\_mutex);  
 sem\_post(&g\_sem\_full);  
 sleep(1);  
 }  
 **return** NULL;  
}  
  
**void** \*produce(**void** \*arg) {  
 **int** inx = \*((**int** \*) arg);  
 free(arg);  
 **int** i;  
 **while** (1) {  
 printf(**"%d wait buffer full\n"**, inx);  
 sem\_wait(&g\_sem\_full);  
 pthread\_mutex\_lock(&g\_mutex);  
 *//生产产品  
 //打印仓库当前状态* **for** (i = 0; i < BUFFSIZE; ++i) {  
 printf(**"%02d "**, i);  
 **if** (-1==g\_buffer[i]) {  
 printf(**"null"**);  
 } **else** {  
 printf(**"%d "**, g\_buffer[i]);  
 }  
 **if** (i==in) {  
 printf(**"\t<--produce"**);  
 }  
 printf(**"\n"**);  
 }  
 printf(**"%d begin produce product %d\n"**, inx, produce\_id);  
 g\_buffer[in] = produce\_id;  
 in = (in + 1)%BUFFSIZE;  
 printf(**"%d end produce product %d\n"**, inx, produce\_id++);  
 pthread\_mutex\_unlock(&g\_mutex);  
 sem\_post(&g\_sem\_empty);  
 sleep(1);  
 }  
 **return** NULL;  
}  
  
**int** main(**int** argc, **char** \*argv[]) {  
 *//初始化一个匿名的POSIX信号量* sem\_init(&g\_sem\_full, 0, BUFFSIZE);  
 sem\_init(&g\_sem\_empty, 0, 0);  
 *//互斥锁* pthread\_mutex\_init(&g\_mutex, NULL);  
 **int** ret, i, \*p;  
 *//初始化仓库* **for** (i = 0; i < BUFFSIZE; ++i) {  
 g\_buffer[i] = -1;  
 }  
 *//创建线程* **for** (i = 0; i < CONSUMERS\_COUNT; ++i) {  
 p = (**int** \*) malloc(**sizeof**(**int**));  
 \*p = i;  
 ret = pthread\_create(&g\_thread[i], NULL, consume, p);  
 **if** (ret!=0) {  
 fprintf(stderr, **"pthread\_create:%s\n"**, strerror(ret));  
 exit(EXIT\_FAILURE);  
 }  
 }  
 **for** (i = 0; i < PRODUCERS\_COUNT; ++i) {  
 p = (**int** \*) malloc(**sizeof**(**int**));  
 \*p = i;  
 ret = pthread\_create(&g\_thread[CONSUMERS\_COUNT + i],  
 NULL, produce, p);  
 **if** (ret!=0) {  
 fprintf(stderr, **"pthread\_create:%s\n"**, strerror(ret));  
 exit(EXIT\_FAILURE);  
 }  
 }  
 *//主线程最好是等待子线程结束* **for** (i = 0; i < CONSUMERS\_COUNT + PRODUCERS\_COUNT; ++i) {  
 ret = pthread\_join(g\_thread[i], NULL);  
 **if** (ret!=0) {  
 fprintf(stderr, **"pthread\_join:%s\n"**, strerror(ret));  
 exit(EXIT\_FAILURE);  
 }  
 }  
 sem\_destroy(&g\_sem\_full);  
 sem\_destroy(&g\_sem\_empty);  
 pthread\_mutex\_destroy(&g\_mutex);  
 **return** 0;  
}

运行结果(部分)：



## 5实验总结

本次实验引入线程的概念，需要同学们理解线程的相关概念，掌握用户线程与内核线程的区别，掌握多进程与多线程的区别，对生产者消费者问题有更深一步的认识，能够初步的掌握Pthreads进行线程的相关操作。