# 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" \t "_blank" \o "Hadoop知识库)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

4.1Pthreads介绍

POSIX线程（POSIX threads），简称Pthreads，是线程的POSIX标准。该标准定义了创建和操纵线程的一整套API。在类Unix操作系统（Unix、Linux、Mac OS X等）中，都使用Pthreads作为操作系统的线程。

1. 常见的数据类型：

pthread\_t:线程id

pthread\_attr\_t:线程属性

1. 常见的操作函数：

pthread\_create()：创建一个线程

pthread\_exit()：终止当前线程

pthread\_cancel()：中断另外一个线程的运行

pthread\_join()：阻塞当前的线程，直到另外一个线程运行结束

pthread\_attr\_init()：初始化线程的属性

pthread\_attr\_getdetachstate()：获取脱离状态的属性

pthread\_attr\_destroy()：删除线程的属性

pthread\_kill()：向线程发送一个信号

4.2线程锁介绍

在线程实际运行过程中，我们经常需要多个线程保持同步。这时可以用Pthreads提供的互斥锁来完成任务；互斥锁的使用过程中，主要有pthread\_mutex\_init，pthread\_mutex\_destory，pthread\_mutex\_lock，pthread\_mutex\_unlock这几个函数以完成锁的初始化，锁的销毁，上锁和释放锁操作。

1. 锁的创建：

锁可以被动态或静态创建，可以用宏PTHREAD\_MUTEX\_INITIALIZER来静态的初始化锁，采用这种方式比较容易理解，互斥锁是pthread\_mutex\_t的结构体，而这个宏是一个结构常量，如下可以完成静态的初始化锁：

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

另外锁可以用pthread\_mutex\_init函数动态的创建，函数原型如下

int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \* attr)

1. 锁的属性：

互斥锁属性可以由pthread\_mutexattr\_init(pthread\_mutexattr\_t \*mattr);来初始化，然后可以调用其他的属性设置方法来设置其属性；

互斥锁的范围：可以指定是该进程与其他进程的同步还是同一进程内不同的线程之间的同步。可以设置为PTHREAD\_PROCESS\_SHARE和PTHREAD\_PROCESS\_PRIVATE。默认是后者，表示进程内使用锁。可以使用下面两个方法用来设置与获取锁的范围；

int pthread\_mutexattr\_setpshared(pthread\_mutexattr\_t \*mattr, int pshared)

int pthread\_mutexattr\_getshared(pthread\_mutexattr\_t \*mattr,int \*pshared)

互斥锁的类型：

PTHREAD\_MUTEX\_TIMED\_NP 缺省值，也就是普通锁

PTHREAD\_MUTEX\_RECURSIVE\_NP 嵌套锁

PTHREAD\_MUTEX\_ERRORCHECK\_NP 检错锁

PTHREAD\_MUTEX\_ADAPTIVE\_NP 适应锁

可以用下面两个方法获取或设置锁的类型：

pthread\_mutexattr\_settype(pthread\_mutexattr\_t \*attr , int type)

pthread\_mutexattr\_gettype(pthread\_mutexattr\_t \*attr , int \*type)

1. 锁的释放：

调用pthread\_mutex\_destory之后，可以释放锁占用的资源，但这有一个前提上锁当前是没有被锁的状态。

1. 锁的操作

对锁的操作主要包括加锁 pthread\_mutex\_lock()、解锁pthread\_mutex\_unlock()和测试加锁 pthread\_mutex\_trylock()三个。

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex)

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex)

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex)  
pthread\_mutex\_trylock()语义与pthread\_mutx\_lock()类似，不同的是在锁已经被占据时返回EBUSY而不是挂起等待。

1. 锁的使用

下面代码给出了锁的简单的使用：

#include <pthread.h>

#include <stdio.h>

pthread\_mutex\_t mutex ;

void \*print\_msg(void \*arg){

int i=0;

pthread\_mutex\_lock(&mutex);

for(i=0;i<15;i++){

printf("output : %d\n",i);

usleep(100);

}

pthread\_mutex\_unlock(&mutex);

}

int main(int argc,char\*\* argv){

pthread\_t id1;

pthread\_t id2;

pthread\_mutex\_init(&mutex,NULL);

pthread\_create(&id1,NULL,print\_msg,NULL);

pthread\_create(&id2,NULL,print\_msg,NULL);

pthread\_join(id1,NULL);

pthread\_join(id2,NULL);

pthread\_mutex\_destroy(&mutex);

return 1;

}

在上面所示的代码中，print\_msg函数的功能是：首先取得线程锁，然后执行循环打印语句，循环结束之后释放线程锁。在mian函数中，创建两个线程，在线程中都调用这个print\_msg函数。某个线程获得线程锁以后会执行循环打印语句，另一个线程会因为未获得线程锁而等待，直到第一个线程之后过后它才能获得线程锁，才可以执行。至于两个线程哪个线程会获得线程锁是不固定的。

4.3信号量介绍

信号量的使用主要是用来保护共享资源，使得资源在一个时刻只有一个进程（线程）所拥有。信号量的值是正的时候，说明它处于空闲状态，若为0，说明它被占用，测试的线程要进入睡眠队列中，等待被唤醒。

信号量分为有名信号量以及无名信号量，有名信号量其值保存在文件中，它可以用于线程也可以用于进程间的同步，无名信号量，其值保存在内存之中。在本实验中，运用的是无名信号量。下面是对用到无名信号量时的一些常用函数的介绍：

1. 初始化信号量：

int sem\_init(sem\_t \*sem, int pshared, unsigned int value)

sem为指向信号量结构的一个指针；pshared不为0时此信号量在进程间共享，否则只能为当前进程的所有线程共享；value给出了信号量的初始值。

1. 获得信号量当前的值：

int sem\_getvalue(sem\_t \*sem, int \*sval);

取回信号量sem的当前值，把该值保存到sval中。若有1个或更多的线程或进程调用sem\_wait阻塞在该信号量上，该函数返回两种值：0或者是阻塞在该信号量上的进程或线程的数目，Linux采用返回的第一种策略。

1. 申请资源，即P操作：

int sem\_wait(sem\_t \*sem);

这是一个阻塞的函数。测试所指定信号量的值，它的操作是原子的。若sem>0，那么它减1并立即返回。若sem==0，则睡眠直到sem>0，此时立即减1，然后返回。

int sem\_trywait(sem\_t \*sem);

这是一个非阻塞的函数其他的行为和sem\_wait一样，除了若sem==0，不是睡眠，而是返回一个错误EAGAIN。

1. 释放资源，即V操作：

int sem\_post(sem\_t \*sem);

把指定的信号量sem的值加1；呼醒正在等待该信号量的任意线程。只有sem\_post是信号安全的函数，它是可重入函数。

1. 一个简单的使用无名信号量的例子

对于上面几点介绍的信号量的相关操作，这个例子的主要目的就是向同学们展示无名信号量在编程之中的具体应用，掌握这个例子很重要，后面的综合实验需要以这个小实验介绍的无名信号量的使用作为基础。

#include <pthread.h>

#include <semaphore.h>

#include <sys/types.h>

#include <stdio.h>

#include <unistd.h>

int number; // 被保护的全局变量

sem\_t sem\_id;

void\* thread\_one\_fun(void \*arg)

{

sem\_wait(&sem\_id);

printf("thread\_one have the semaphore\n");

number++; printf("number = %d\n",number);

sem\_post(&sem\_id);

}

void\* thread\_two\_fun(void \*arg)

{

sem\_wait(&sem\_id);

printf("thread\_two have the semaphore \n");

number--;

printf("number = %d\n",number);

sem\_post(&sem\_id);

}

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

{

number = 1;

pthread\_t id1, id2;

sem\_init(&sem\_id, 0, 1);

pthread\_create(&id1,NULL,thread\_one\_fun, NULL);

pthread\_create(&id2,NULL,thread\_two\_fun, NULL);

pthread\_join(id1,NULL);

pthread\_join(id2,NULL);

printf("main,,,\n");

return 0;

}

上面所示的例子主要展示了无名信号量的实际应用，可以用来进行线程之间的同步。在这个例子之中，创建了两个线程，一个线程执行的是将number进行“++”操作，一个线程执行的是将number进行“--”操作。线程中获得到这个信号量则进行操作，直到这个线程执行完另一个线程才开始执行。这里需要说明的是，如果设置的是线程之间的互斥，则多个线程只设置一个信号量sem，如果设置的是线程之间的同步，则需要设置多个信号量，安排不同的初值来实现他们的顺序进行，线程的同步情况如图1所示，线程的互斥情况如图2所示。上述的例子就是线程之间互斥的一个例子。

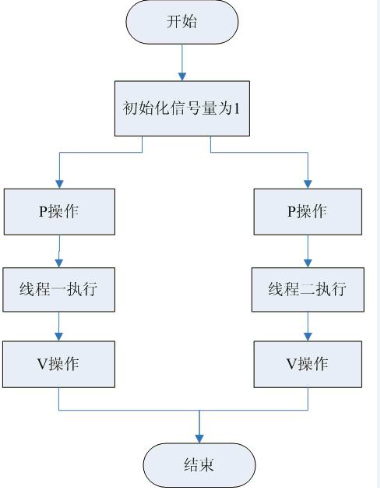


图 1 线程互斥情况

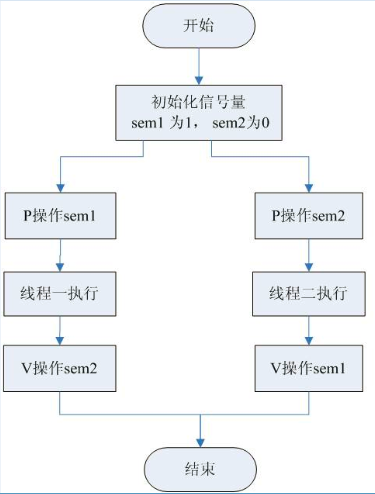


图 2 线程同步情况

4.3生产者消费者问题代码实现

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdint.h>

#include <stdbool.h>

#include <errno.h>

#include <unistd.h>

#include <pthread.h>

#include <semaphore.h>

#include <sys/types.h>

#define ERR\_EXIT(m)

do {

perror(m);

} while(0)

#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;

}

上述代码给出了生产者-消费者问题的一种实现，主要采用了信号量，通过PV操作，来进行生产者线程和消费者线程的同步过程。主要注意的是，生产者需要先执行，产生物品之后，消费者才可以执行，当缓冲区满了之后，生产者不再继续执行。如果是多个生产者与多个消费者，则在运用信号量进行PV操作的同时，还需运用线程锁，这样才能够解决发生的资源冲突情况。

## 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。

## 6参考文献

可以在网上搜索Pthread的相关API，会有更多更加详细的关于线程操作的介绍。

## 7实验扩展

在本次实验中介绍了1个生产者1个消费者的线程同步问题，同学们可以在课后试着去尝试着用Pthreads的相关操作去实现多个生产者多个消费者问题。

### 实验六: 多进程、多线程综合实验

## 1实验目的

综合实验，主要是巩固之前所学的知识，让学生分别实验一个多进程、多线程的生产者-消费者。

## 2实验内容

2.1 编写程序使用多进程实现生产者-消费者问题。

编写程序，使用多进程解决生产者-消费者问题，通过系统函数shmget实现进程间内存共享，通过系统函数semget、semctl、semop实现进程间信号量的同步与互斥。

2.2 编写程序使用多线程实现生产者-消费者问题。

编写程序，使用多线程解决生产者-消费者问题，使用数据结构sem\_t及其相关系统函数，实现线程同步，使用数据结构pthread\_mutex\_t 及其相关系统函数，实现线程互斥。

## 3实验原理

3.1 生产者-消费者问题

生产者和消费者问题是多个相互合作的进程（线程）之间的一种抽象。生产者和消费者之间的关系:

1.  对缓冲区的访问是互斥的。由于两者都会修改缓冲区，因此，一方修改缓冲区时，另一方不能修改，这就是互斥。

2.  一方的行为影响另一方。缓冲区不空，才能消费，何时不空？生产了就不空；缓冲区满，就不能生产，何时不满？消费了就不满。这是同步关系。

为了描述这种关系，一方面，使用共享内存代表缓冲区；另一方面，使用互斥信号量控制对缓冲区的访问，使用同步信号量描述两者的依赖关系。

本实验需要的三个信号量的功能详述：

第一个信号量用于限制生产者必须在缓冲区不满时才能生产，是同步信号量。

第二个信号量用于限制消费者必须在缓冲区有产品时才消费，是同步信号量。

第三个信号量用于限制生产者和消费者在访问缓冲区时必须互斥，是互斥信号量。

3.2 多进程解决生产者-消费者问题

1.共享内存：

共享存储是进程间通信的一种手段，通常，使用信号量同步或互斥访问共享存储。共享存储的原理是将进程的地址空间映射到一个共享存储段。在LINUX下，通过使用 shmget 函数创建或者获取共享内存。

只需要共享存储的 ID 就可以通过  shmat  函数获得共享存储所占用的实际地址。因此，可以在父进程的栈中用变量存放指向共享存储的指针，那么 fork 之后，子进程就可以很方便地通过这个指针访问共享存储了。

2.创建获取信号量：

使用函数semget(IPC\_PRIVATE, NUM\_OF\_SEM, FLAGS )和semget(SEM\_KEY, NUM\_OF\_SEM,FLAGS);

3初始化信号量

使用函数semctl(int semSetId , int semIdx , int cmd, union semun su);

4对信号量实行P、V操作

使用函数semop(int semid,struct sembuf\*sops,unsign ednsops);

3.3 多线程解决生产者-消费者问题

1.共享内存

使用数组int buff[M] = {0}。

2三个信号量

sem\_t empty\_sem; // 同步信号量， 当满了时阻止生产者放产品

sem\_t full\_sem; // 同步信号量， 当没产品时阻止消费者消费

pthread\_mutex\_t mutex; // 互斥信号量， 一次只有一个线程访问缓冲

3对信号量的操作

使用函数sem\_wait，和sem\_post对同步信号量进行PV操作。

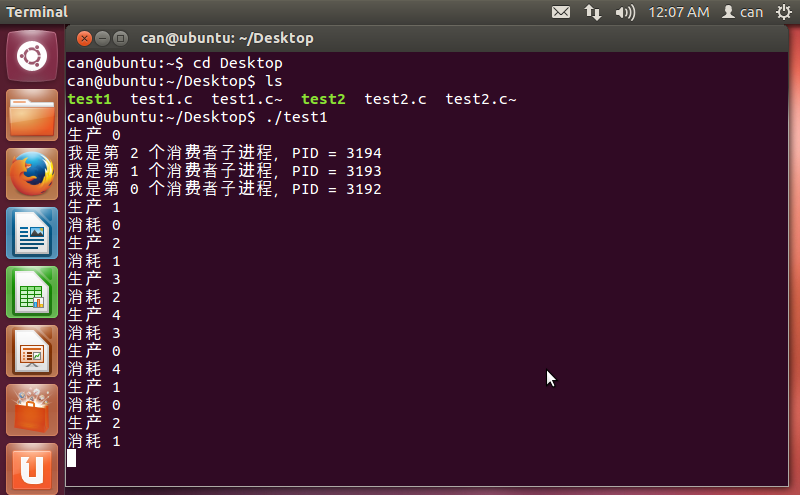
使用函数pthread\_mutex\_unlock对互斥信号量进行PV操作。

## 4实验步骤

4.1多进程解决生产者-消费者问题

a.编写函数test1。(见参考源码4.3.1)

b.编译执行查看结果。随机结果如图4.1所示。

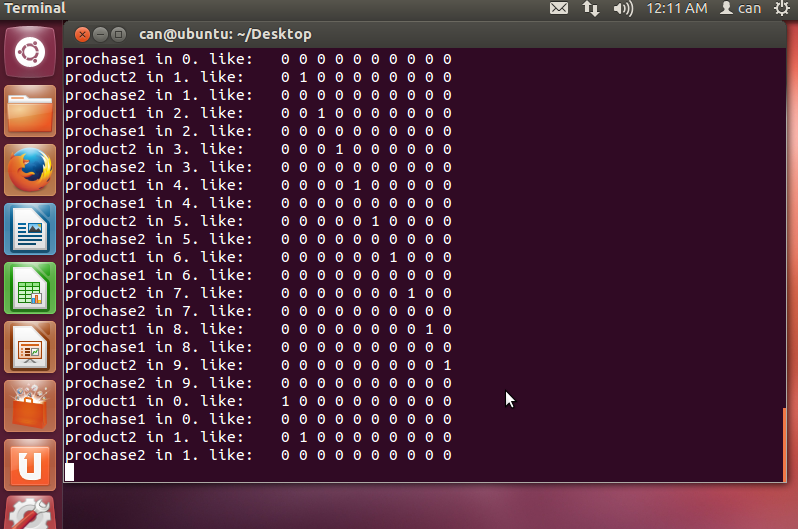


**图4.1**

4.2 多线程解决生产者-消费者问题

a.编写函数test2（见参考源码4.3.2）。

b.编译执行查看结果。随机结果如图4.2所示



**图4.2**

4.3 参考源码：