电子科技大学信息与软件工程学院

**实 验 报 告**

学 号 2015220103022

姓 名 张健顺

（实验） 课程名称 **操作系统基础**

理论教师 杨霞

实验教师 杨霞

**电子科技大学教务处制表**

**电 子 科 技 大 学**

**实 验 报 告**

**学生姓名：张健顺 学号：2015220103022 指导教师：杨霞实验地点：信软楼西304 实验时间：2017.04.20**

**一、实验名称：信号量经典问题的实现**

**二、实验学时：4**

**三、实验目的：**本实验分为两部分，首先实现哲学家就餐问题，要求不能出现死锁。通过本实验熟悉Linux系统的基本环境，了解Linux下进程和线程的实现。然后实现生产者/消费者问题，通过本实验掌握进程间的同步和互斥机制的使用。熟悉基于某操作系统进程和线程的编程。

1. **掌握进程、线程的概念，熟悉相关的控制原语。**
2. **掌握进程、线程间的同步原理和方法。**
3. **掌握进程、线程间的互斥原理和方法。**

**掌握使用信号量原语解决进程、线程间互斥和同步方法。**

**四、实验原理：**

由Dijkstra提出并解决的哲学家进餐问题(The Dinning Philosophers Problem)是典型的同步问题。该问题是描述有五个哲学家共用一张圆桌，分别坐在周围的五张椅子上，在圆桌上有五个碗和五只筷子，他们的生活方式是交替地进行思考和进餐。平时，一个哲学家进行思考，饥饿时便试图取用其左右最靠近他的筷子，只有在他拿到两只筷子时才能进餐。进餐完毕，放下筷子继续思考。

第二个问题考虑n个缓冲区的缓冲池作为一个共享资源，当生产者进程从数据源—文件中读取数据后将会申请一个缓冲区，并将此数据放入缓冲区中。消费者从一个缓冲区中取走数据，并将其中的内容打印输出。当生产者进程正在访问缓冲区时，消费者进程不能同时访问缓冲区，因此缓冲区是个互斥资源。

**五、实验内容：**

该实验分为两部分内容:

第一步：实现哲学家就餐问题

1）熟悉Ubuntu系统环境和命令；

2）熟悉Ubuntu系统下的多线程编程；

2）在Ubuntu系统下编程实现哲学家就餐问题。

实现教材2.5.2节中所描述的哲学家就餐问题。要求显示出每个哲学家的工作状态，如吃饭，思考。连续运行30次以上都未出现死锁现象。

第二步：实现生产者/消费者问题

1）有一群生产者进程在生产产品，并将这些产品提供给消费者进程去消费。为使生产者进程与消费者进程能并发执行，在两者之间设置了一个具有n个缓冲区的缓冲池：生产者进程从文件中读取一个数据，并将它存放到一个缓冲区中； 消费者进程从一个缓冲区中取走数据，并输出此数据。生产者和消费者之间必须保持同步原则：不允许消费者进程到一个空缓冲区去取产品；也不允许生产者进程向一个已装满产品且尚未被取走的缓冲区中投放产品。

2）、创建3进程（或者线程）作为生产者，4个进程（或者线程）作为消费者。创建一个文件作为数据源，文件中事先写入一些内容作为数据。

3）、生产者和消费者进程（或者线程）都具有相同的优先级。

**六、实验器材（设备、元器件）：**

1. 学生每人一台PC，安装Windows8操作系统。
2. 个人PC安装VMware虚拟机和Ubuntu系统。

**七、实验步骤：**

（一） 熟悉Ubuntu系统下的多线程编程。

1. 使用“Ctrl+Alt+T”打开终端；

2. 使用gedit或vim命令打开文本编辑器进行编码： “gedit 文件名.c”

3．编译程序：

“ gcc 文件名.c -o 可执行程序名 ”

（如果只输入 gcc 文件名.c，默认可执行程序名为a.out）

使用线程库时，gcc编译需要添加-lpthread

1. 执行程序：./可执行程序名

（二）信号量同步问题实现

步骤1：分配具有n个缓冲区的缓冲池，作为共享资源。

步骤2：定义两个资源型信号量empty 和full，empty信号量表示当前空的缓冲区数量，full表示当前满的缓冲区数量。

步骤3：定义互斥信号量mutex，当某个进程访问缓冲区之前先获取此信号量，在对缓冲区的操作完成后再释放此互斥信号量。以此实现多个进程对共享资源的互斥访问。

步骤4：创建3进程（或者线程）作为生产者，4个进程（或者线程）作为消费者。创建一个文件作为数据源，文件中事先写入一些内容作为内容。

步骤5 ：编写代码实现生产者进程的工作内容，即从文件中读取数据，然后申请一个empty信号量，和互斥信号量，然后进入临界区操作将读取的数据放入此缓冲区中。并释放empty信号量和互斥信号量。

步骤6：编写代码实现消费者者进程的工作内容，即先申请一个full信号量，和互斥信号量，然后进入临界区操作从缓冲区中读取数据并打印输出。

**八、实验结果与分析（含重要数据结果分析或核心代码流程分析）**

第一步：实现哲学家就餐问题

核心思想：为了避免产生死锁，可以采用多种方式来解决。而个人采用的是哲学家先取左筷子，取完左筷子后尝试着去取右筷子，如果失败，则放弃左筷子，释放左筷子，跳出循环，避免左筷子被该哲学家进程阻塞。如果拿到右筷子，则执行吃饭的动作。一段时间后，释放左右筷子。

程序源码：

#include <pthread.h>

#include <stdio.h>

#include <errno.h>

#include <stdlib.h>

#define N 6

pthread\_t thread[N];

char id[N];

pthread\_mutex\_t chopstick[N];

void\* thread\_start(void \*arg)

{

char philosopher =\*(char\*)arg;

int left,right;

//Acoording to the thread,give left and right chopstick number.

switch(philosopher){

case 'A' : left = 5; right = 1; break;

case 'B' : left = 1; right = 2; break;

case 'C' : left = 2; right = 3; break;

case 'D' : left = 3; right = 4; break;

case 'E' : left = 4; right = 5; break;

}

for(int i = 0; i < 30 ; i++){

// sleep for one seconds

sleep(1);

//catch left chopstick

pthread\_mutex\_lock(&chopstick[left]);

printf("Philosopher %c catches left chopsticks %d\n",philosopher,left);

// pthread\_mutex\_trylock() try to lock the resource

// if return EBUSY,it means lock failure

if(pthread\_mutex\_trylock(&chopstick[right]) == EBUSY){

//can't get right chopstick so release left chopstick

pthread\_mutex\_unlock(&chopstick[left]); printf("Philosopher %c drops left chopsticks %d\n",philosopher,left);

continue;

}

// At here,it means that lock the resource success!

printf("Philosopher %c catches right chopsticks %d\n",philosopher,right);

// When philosopher gets left chopstick and right chopstick

printf("Philosopher %c is having breakfast!\n",philosopher);

sleep(1);

// After eating,we should release the resource!

pthread\_mutex\_unlock(&chopstick[left]);

printf("Philosopher %c drops left chopsticks %d\n",philosopher,left);

pthread\_mutex\_unlock(&chopstick[right]);

printf("Philosopher %c drops right chopsticks %d\n",philosopher,right);

}

}

void\* thread\_create() //Create thread

{

int temp,i;

for(i=1;i<N;i++){

// use pthread\_create() to create threads with names

temp = pthread\_create(&thread[i],NULL,&thread\_start,&id[i]);

if(temp != 0 ){

printf("Philosopher thread %d create failure !!\n",i);

}else{

printf("Philosopher thread %d create success !!\n",i);

}

}

}

void\* thread\_wait()

{

for(int i=1;i<6;i++){

pthread\_join(thread[i],NULL);

printf("Phiolosopher thread %d ends !!!\n", i );

}

}

int main()

{

int i;

int b = 65;//ASCII 'A' convert number to letter

for(int i=1;i<6;i++){

id[i] = (char)b;

b++;

pthread\_mutex\_init(&chopstick[i],NULL);

}

printf("\n\n\n");

thread\_create();

thread\_wait();

return 0;

}

程序分析：根据main()函数分析程序执行过程。

① 构造线程名组成的数组，并进行初始化；

② 为每一个筷子分配一把互斥锁并初始化互斥锁；

③ 调用线程创建函数：

1. 线程创建函数根据已有的参数和例程函数进行线程的创建
2. 线程例程函数中实现哲学家就餐

④ 调用pthread\_join函数，执行线程中的代码

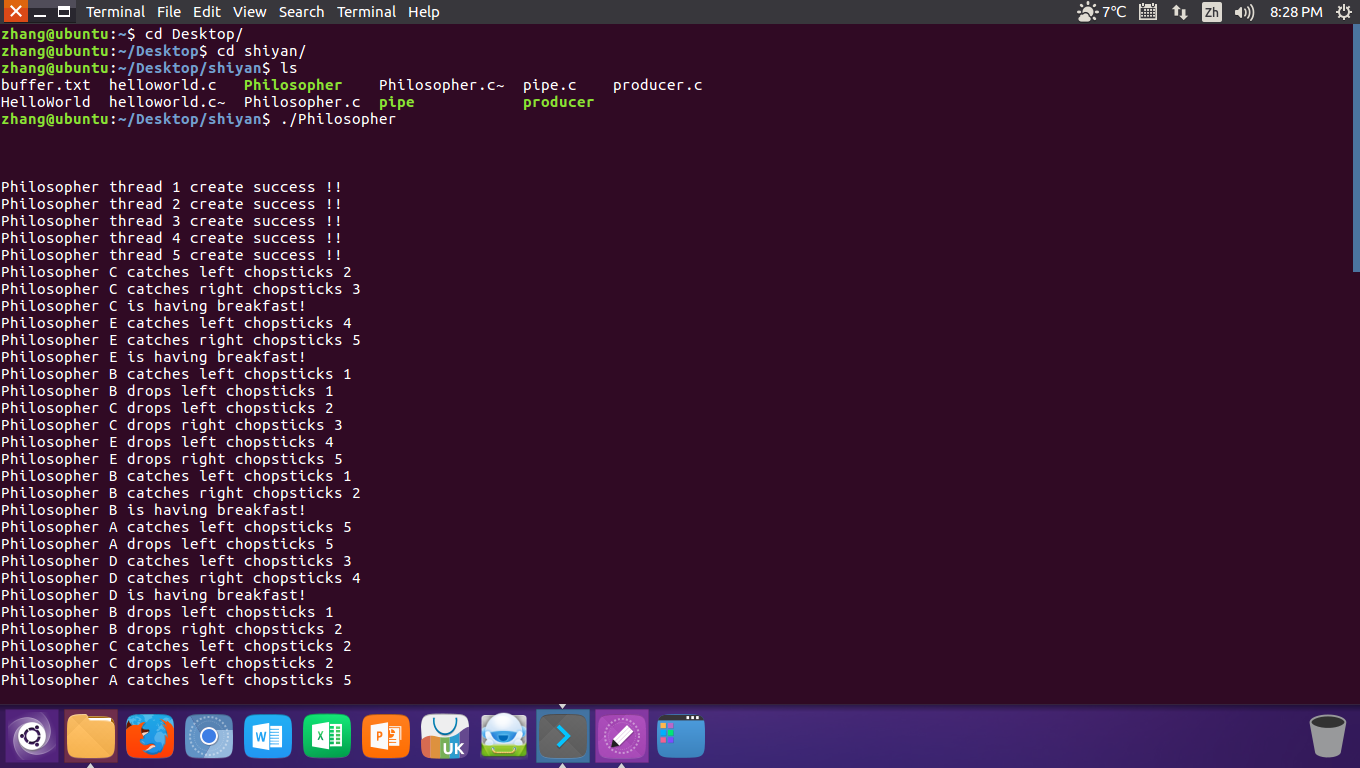
核心函数分析：

void\* thread\_start(void \*arg)

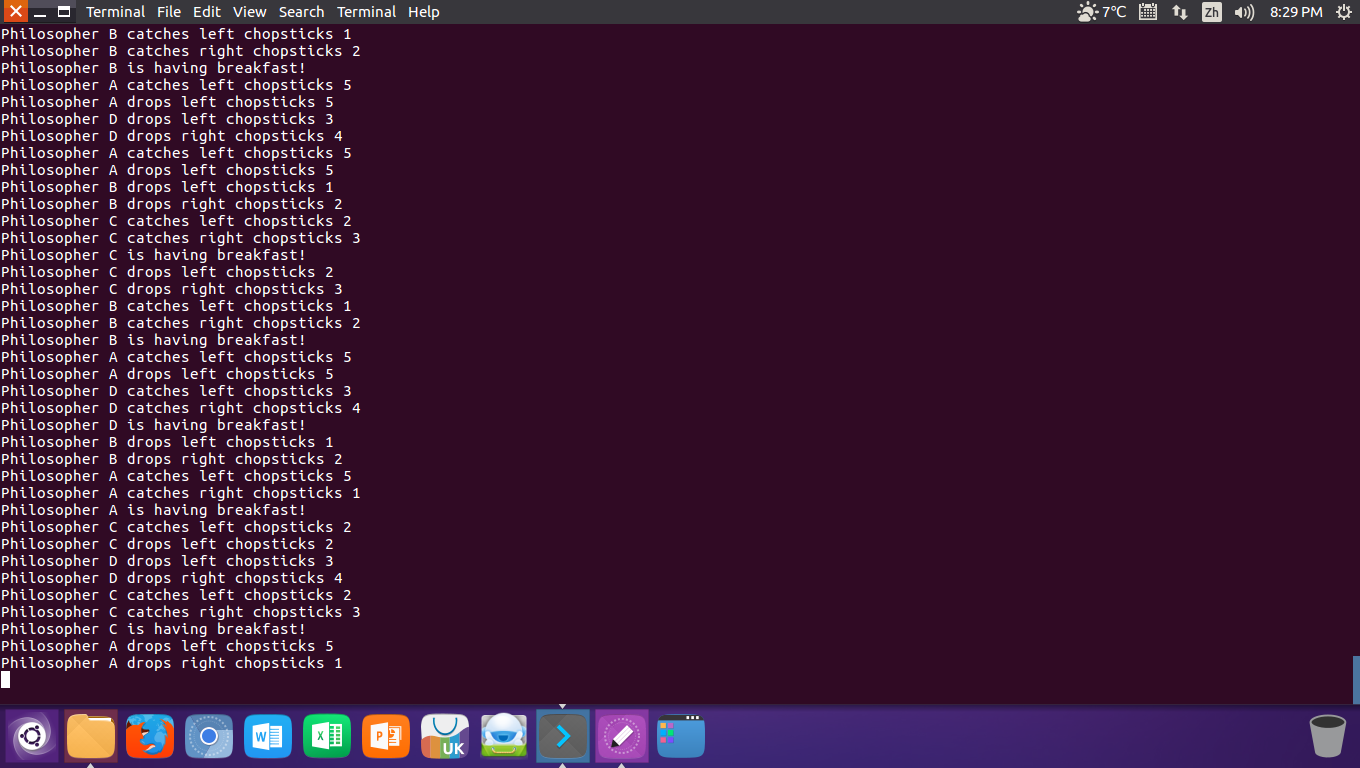
该线程例程函数先根据哲学家线程的名字分配左右筷子资源，然后循环执行哲学家拿筷子就餐的程序。哲学家先通过给左筷子加锁的方式拿到左筷子pthread\_mutex\_lock(),拿到左筷子后，尝试给右筷子进行加锁操作，调用pthread\_mutex\_trylock()函数，根据返回值判断是否加锁成功，若成功，则哲学家开始就餐，输出状态语句，就餐完毕后释放左右筷子供下一位哲学家就餐使用。若失败，则释放被哲学家占用的左筷子，并输出思考状态，跳出循环。

程序结果截图：

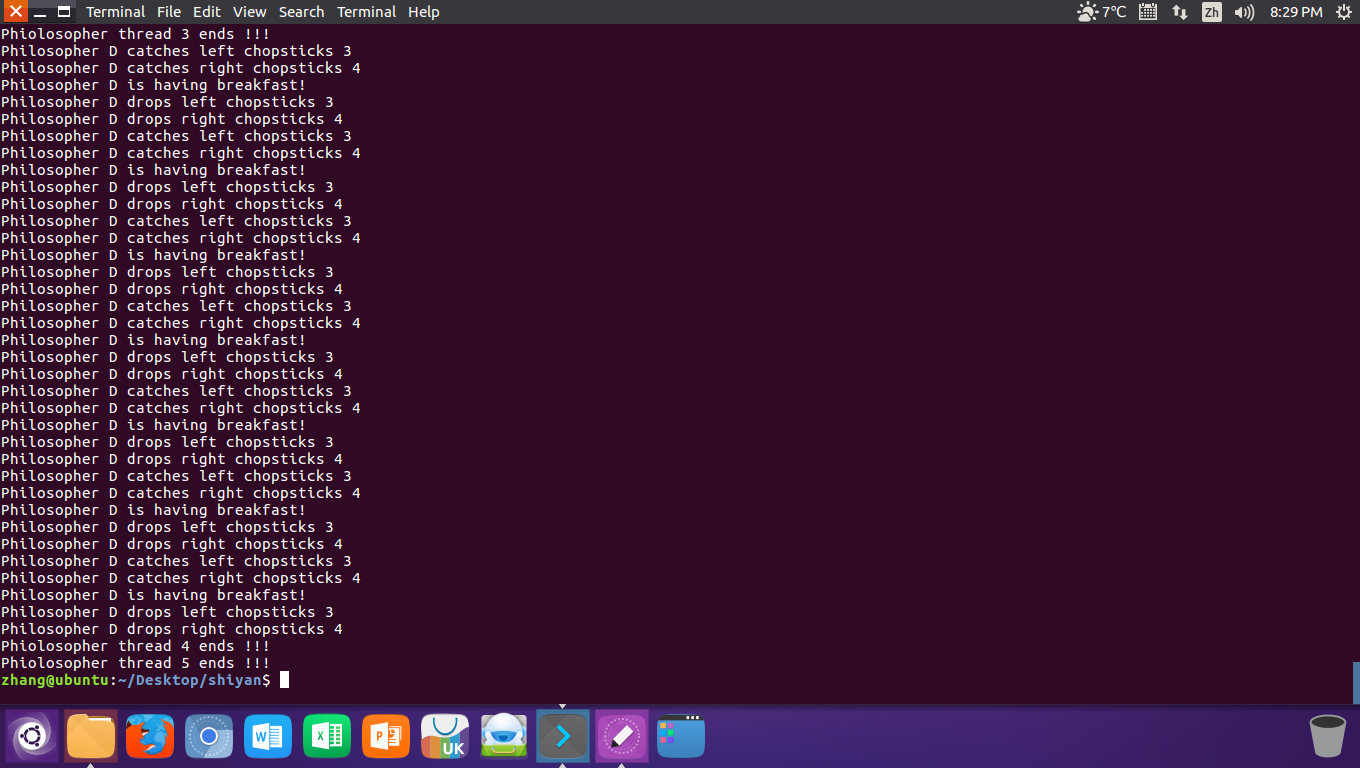
在控制台键入命令gcc Philosopher.c -o Philosopher -lpthread 之后会生成Philosopher的可执行文件，再键入./Philosopher执行。



线程创建成功，开始执行



线程执行过程中（循环过程中）



线程结束执行（循环三十次后结束执行）

第二步：实现生产者/消费者问题

核心思想：为了解决资源的同步问题，采用信号量的机制来实现。生产者首先对于空的缓冲区进行减一操作，也就是empty资源信号量减一，然后为这片缓冲区（资源）加锁，也就是所谓的互斥信号量减一，避免受到其他线程的影响，然后从文件中读取到数据存入到缓冲区中。生产操作完成后，释放该缓冲区的锁，full互斥信号量加一，然后缓冲区资源信号量加一。对于消费者，则是先对满的缓冲区进行减一操作，full资源信号量减一，之后对缓冲区进行加锁操作，互斥信号量减一，再进行资源消费，消费完以后释放该缓冲区的锁，互斥信号量加一，再释放缓冲区资源，empty资源信号量加一。

程序源码：

#include <stdio.h>

#include <semaphore.h>

#include <pthread.h>

#include <unistd.h>

#include <fcntl.h>

#define producersNumber 3

#define consumersNumber 4

#define bufferSize 10

int in=0;

int out=0;

int producerId = 0;

int consumerId = 0;

int bufferData;

int buffer[bufferSize];

sem\_t mutex;

sem\_t empty;

sem\_t full;

FILE\* file;

void\* producer()

{

printf("start to execute!!\n");

do{

sleep(1);

sem\_wait(&empty);

sem\_wait(&mutex);

if(fscanf(file,"%d",&bufferData) == EOF)

{

fseek(file,0,SEEK\_SET);

fscanf(file,"%d",&bufferData);

}

buffer[in] = bufferData;

printf("Producer produce product %d !!!\n",buffer[in]);

in = (in+1) % bufferSize;

sem\_post(&mutex);

sem\_post(&full);

}while(1);

}

void\* consumer()

{

sleep(1);

do{

sem\_wait(&full);

sem\_wait(&mutex);

printf("Consumer consume product %d !!! \n",buffer[out]);

buffer[out] = 0;

out = (out+1) % bufferSize;

sem\_post(&mutex);

sem\_post(&empty);

}while(1);

}

int main()

{

int i;

int status;

pthread\_t producers[producersNumber];

pthread\_t consumers[consumersNumber];

sem\_init(&empty,0,bufferSize);

sem\_init(&full,0,0);

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

if((file = fopen("buffer.txt","r")) == NULL)

{

printf("Can't open the file!!!\n");

return 0;

}

for(i = 0; i < producersNumber; i++)

{

pthread\_join(producers[i],NULL);

}

for(i = 0; i < consumersNumber; i++)

{

pthread\_join(consumers[i],NULL);

}

return 0;

}

for(i = 0; i < producersNumber; i++)

{

status = pthread\_create(&producers[i],NULL,producer,(void\*)&i);

if(status != 0 ){

printf("Producer thread %d create failure !!\n",i);

}else{

printf("Producer thread %d create success !!\n",i);

}

}

for(i = 0; i < consumersNumber; i++)

{

status = pthread\_create(&consumers[i],NULL,consumer,(void\*)&i);

if(status != 0 ){

printf("Consumer thread %d create failure !!\n",i);

}else{

printf("Consumer thread %d create success !!\n",i);

}

}

for(i = 0; i < producersNumber; i++)

{

pthread\_join(producers[i],NULL);

}

for(i = 0; i < consumersNumber; i++)

{

pthread\_join(consumers[i],NULL);

}

return 0;

}

程序分析：

根据main()函数，首先将empty、full和mutex信号量进行初始化，再打开文件，判断是否成功，再利用循环分别创建生产者和消费者线程，再利用循环分别调用生产者和消费者线程的join函数执行线程。

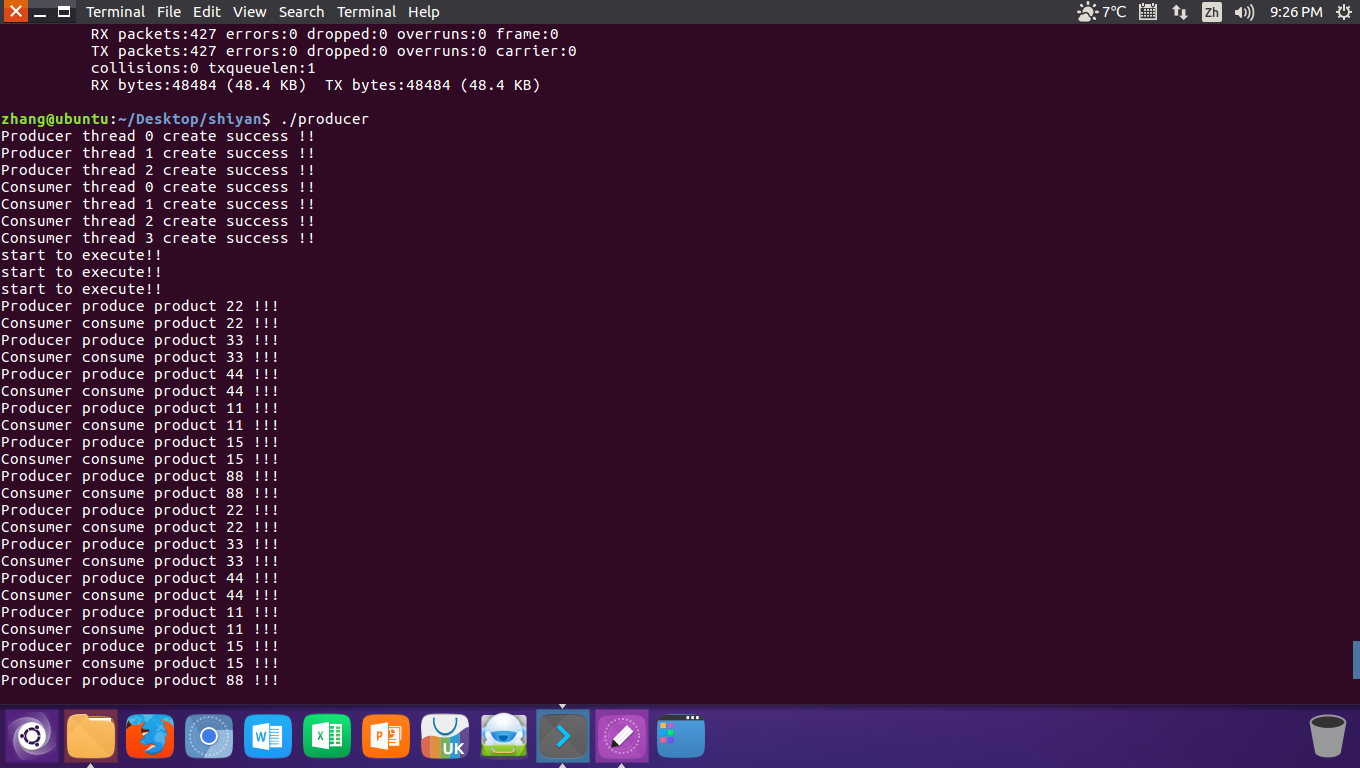
关键函数分析：

void\* consumer();

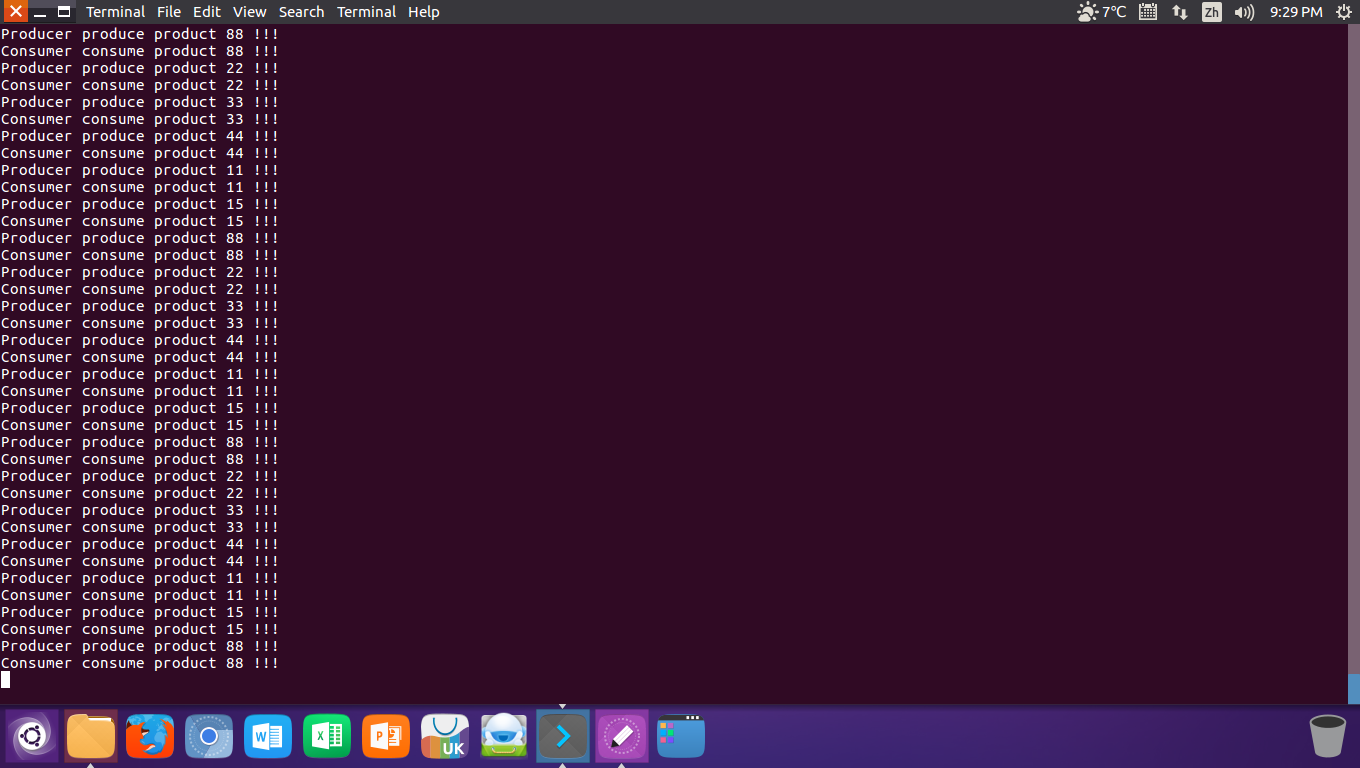
void\* producer();

两个函数都是先等待资源信号量，然后等待互斥信号量对缓冲区进行加锁，对缓冲区执行相关操作后，再释放资源信号量，释放互斥信号量，以保证同步。

结果截图:

****

创建生产者消费者线程

****

生产者消费者线程开始执行

1. **总结及心得体会：**

通过本次实验，了解了linux系统下的相关操作命令，了解了ubuntu下的多线程编程，对于线程间的通信和同步有了一个更深层次的理解。也感受到了多线程带来的效率上的显著提高。与此同时，巩固了C语言编程的基础，熟悉了linux下的一些线程操作函数，理解了线程的执行过程，理解了信号量的意义以及相关操作原语，也感受到了linux相比于windows的独到之处。对于以后解决同步和互斥的相关问题奠定了基础。

1. **对本实验过程及方法、手段的改进建议：**

希望能增加实验时长从而寻求更多种解决同步互斥问题的方案，并一一编程实现，同时希望能从源代码的角度出发深入理解linux系统的相关知识，同时将linux和windows系统进行比较，寻找到各个操作系统之间的异同点。

**报告评分：**

**指导教师签字：**

**电 子 科 技 大 学**

**实 验 报 告**

**学生姓名：张健顺 学号：2015220103022 指导教师：杨霞**

**实验地点：信软楼西304 实验时间：2017.04.27**

**一、实验名称：利用管道实现两个进程的通信**

**二、实验学时：4**

**三、实验目的：**

1. **熟悉Linux下的应用程序开发**
2. **熟悉Linux的进程控制原语的使用**
3. **掌握Linux操作系统的进程间通信机制管道的使用。**
4. **掌握Linux操作系统中父进程与子进程的同步。**

**四、实验原理：**

首先创建两个子进程，注意Linux下使用fork()函数创建进程的方法。父进程和两个子进程间需要同步，使用waitpid（）函数实现父进程等待子进程运行完毕后从管道中读取数据并打印，只有子进程将数据写入管道后，父进程才能够执行打开管道操作。

由于fork函数让子进程完整地拷贝了父进程的整个地址空间，所以子进程都有管道的读端和写端。所以在相关进程中最好关掉不用的那一端。根据要求，“父进程先接收子进程P1发来的消息，然后再接收子进程P2发来的消息。”存在两个同步问题，两个子进程和父进程之间（先子写后父读）同步、子进程1和子进程2之间（先1写，再2写）

**五、实验内容：**

在Linux系统中使用系统调用fork()创建两个子进程，使用系统调用pipe()建立一个管道，两个子进程分别向管道各写一句话：

Child process 1 is sending a message!

Child process 2 is sending a message!

而父进程则从管道中读出来自于两个子进程的信息，显示在屏幕上。然后分别结束两个子进程的运行。

要求：

1.父进程先接收子进程P1发来的消息，然后再接收子进程P2发来的消息。

2.在Linux平台下实现。

**六、实验器材（设备、元器件）：**

1. 学生每人一台PC，安装Windows8操作系统。
2. 个人PC安装VMware虚拟机和Ubuntu系统。

**七、实验步骤：**

步骤1：创建一个管道。

步骤2：创建子进程1，向管道中写入“Child process 1 is sending a message!”，并做好跟父进程的同步执行。

步骤3：创建子进程2，向管道中写入“Child process 2 is sending a message!”，并做好跟父进程的同步执行。

步骤4：父进程从管道中读取数据，并打印输出。

**八、实验结果与分析（含重要数据结果分析或核心代码流程分析）**

核心思想：

使用管道实现两个进程之间的通信，管道有两端，读端和写端，两个进程之间的通信则可以采用分别调用读端和写端来实现，多个子进程分别向缓冲区中写入信息。同时为了保证先写后读，父进程需要等待子进程的执行。将会使用到相关的wait函数。进程将数据写入管道后，父进程才能够执行打开管道操作。在读写数据时最好是将不是用的一端给关闭掉。

程序源码：

#include <stdio.h>

#include <unistd.h>

#include <string.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

int main()

{

//fileDest[0] is the reader of the pipe

//fileDest[1] is the writer of the pipe

int fileDest[2];

//define the buffer

char write\_buffer[200],read\_buffer[200];

int length;

//pipe() is to create pipe

pipe(fileDest);

pid\_t child1;

pid\_t child2;

//use fork() to creat progress

child1 = fork();

child2 = fork();

if(child1 == 0)

{

//close the reader of the pipe

close(fileDest[0]);

//sprintf(var,"...") is to write the ... to the var

sprintf(write\_buffer,"Child process 1 is sending a message!\n");

length = strlen(write\_buffer);

//writes the data (length) from the write\_buffer to fileDest[1](the writer of the pipe)

write(fileDest[1],write\_buffer,length);

return 0;

}

else if(child1 > 0)

{

//father progress waits the child1 progress to run

waitpid(child1,NULL,0);

if(child2 == 0)

{

close(fileDest[0]);

sprintf(write\_buffer,"Child process 2 is sending a message!\n");

length = strlen(write\_buffer);

write(fileDest[1],write\_buffer,length);

return 0;

}else if(child2 > 0)

{

waitpid(child2,NULL,0);

close(fileDest[1]);

//read the data (length) from the write\_buffer to fileDest[0](the reader of the pipe)

read(fileDest[0],read\_buffer,length);

printf("%s\n",read\_buffer);

return 0;

}

}

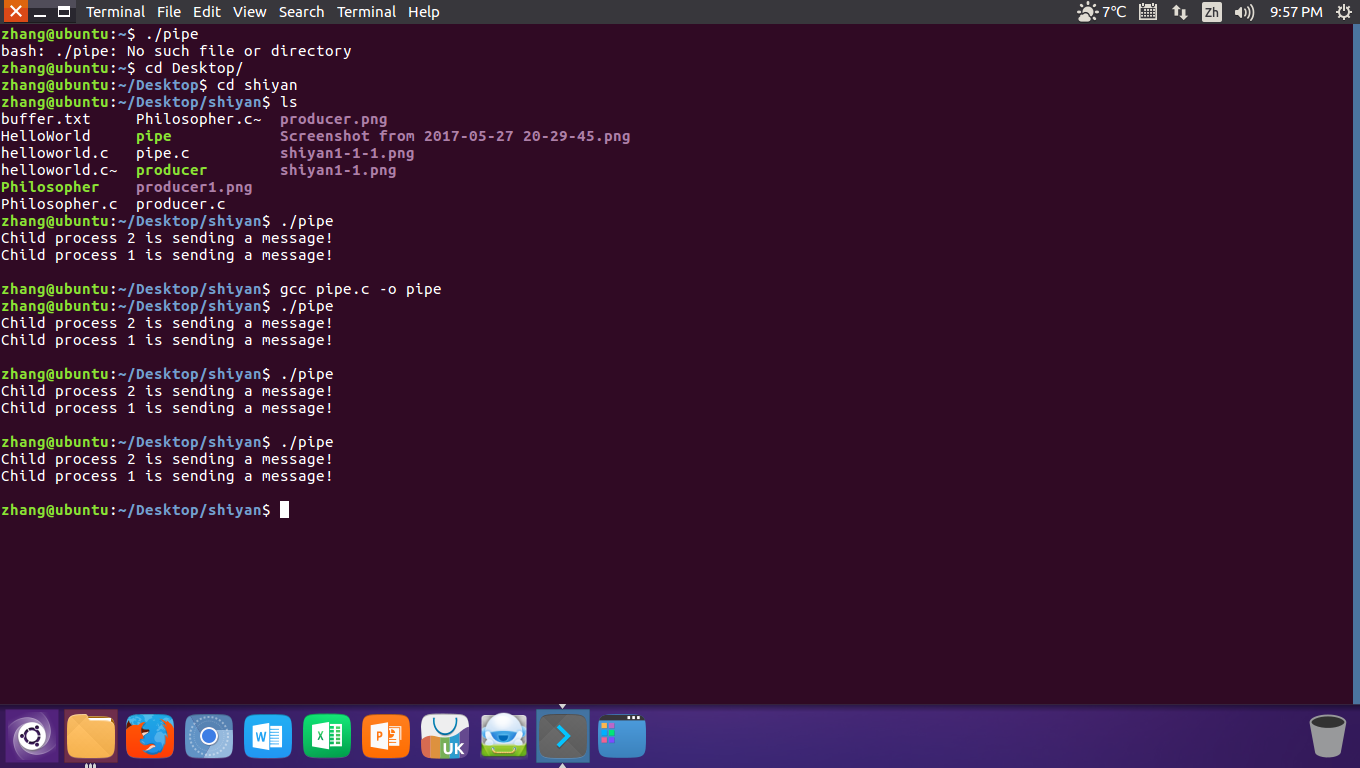
return 0;

}

程序分析：

在完成相关管道、读写缓冲区、子进程的创建以后，判断子进程创建的状态（是否成功），若子进程1创建成功，则向管道中写入相关数据，又因为父进程和两个子进程间需要同步，使用waitpid()函数实现父进程等待子进程运行完毕后从管道中读取数据并打印，只有子进程将数据写入管道后，父进程才能够执行打开管道操作。在读写数据时，为了避免占用相同的地址空间，在读写时关闭管道中不需要使用的一端。

结果截图：



父进程从管道中读出消息

**九、总结及心得体会：**

通过本次实验，了解了linux下的进程创建方法，以及一些进程操作的相关指令。同时也感受到了进程之间相互通信的应用场景，而管道又是一种进程间相互通信的方式。理解了管道的含义以及管道的使用策略。与此同时，加深了对进程间的同步问题的理解。除此之外，也回顾了C语言的读写操作的相关知识。自己也尝试着实现linux和windows文件共享，了解了一些linux下的基本的局域网配置的命令原语和相关知识。

1. **对本实验过程及方法、手段的改进建议：**

希望能通过更多的创建进程的方式（函数）来创建进程，同时理解他们之间的异同以及各自的应用场景。对于进程间的通信和同步，希望能有一些更为生动实际的例子来进行详细阐述。

**报告评分：**

**指导教师签字：**