实验三 同步与通信

16281035 计科1601

1. 实验目的

- 系统调用的进一步理解。
- 进程上下文切换。
- 同步与通信方法。

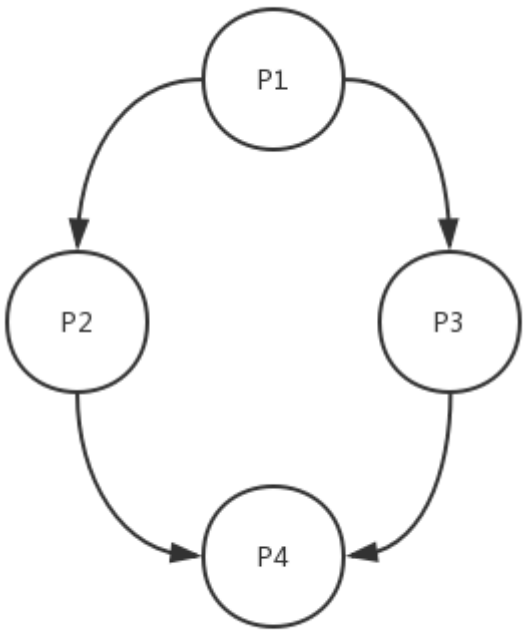
2. 实验题目

题目一

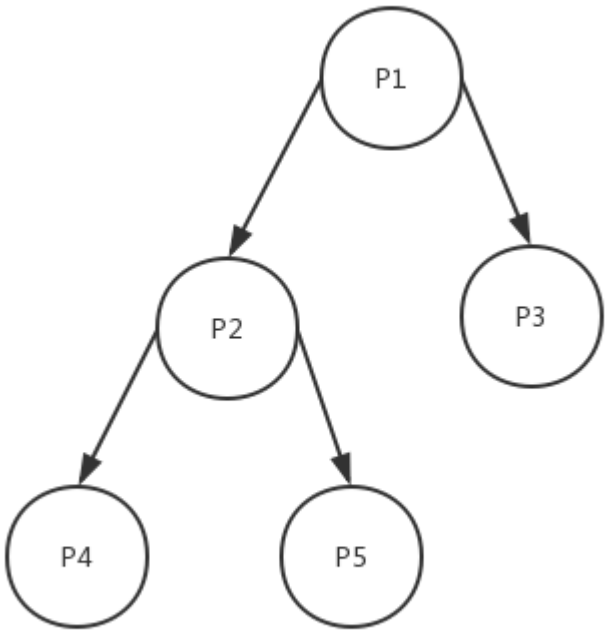
通过fork的方式，产生4个进程P1,P2,P3,P4，每个进程打印输出自己的名字，例如P1输出“I am the process P1”。要求P1最先执行，P2、P3互斥执行，P4最后执行。通过多次测试验证实现是否正确。

解答：

1. 将题意实现如图：



2. 之前做过用fork产生进程的实验，当时的要求如下：



3. 区别：之前的实验对两个进程之间是否互斥并无要求，现在要求P1最先执行，P2、P3互斥执行，P4最后执行，用到信号量机制。

4. 信号量的设计：
    1. P1执行完毕后，P2,P3才能执行，所以互斥信号量的**值只有1和0**。

       初始化P1_signal为0.
    2. P2,P3以互斥的方式实现，两者都等待wait(P1_signal)。
    3. P4在P2和P3都执行完才能执行，所以这时P2执行完应当有一个信号量P2_signal，P3执行完有一个信号量P3_signal，这两个信号量控制P4的执行且**互不相同，互相独立**。P4需要wait(P2_signal)以及wait(P3_signal)后才可以执行。

5. **伪代码**

```
Var P1_signal,P2_signal,P2_signal: semphore:=0,0,0;
begin
    parbegin
      begin P1; signal(P1_signal);end;//P1执行完毕后为其增加一个资源，此后P2、P3竞争。
      begin wait(P1_signal); P2; signal(P1_signal);signal(P2_signal); end;
    //取得信号量的进程消耗一个资源，等待执行完毕后，释放一个资源，供另一个进程执行。
      begin wait(P1_signal); P3; signal(P1_signal);signal(P3_signal); end;
      begin wait(P2_signal);wait(P3_signal);P4;end;
    parend
end
```

6. **源代码**

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<pthread.h>
#include <fcntl.h>
#include <sys/stat.h>
#include<semaphore.h>
#include<sys/types.h> //这个头文件不能少，否则pid_t没有定义
#include <sys/wait.h>


sem_t *P1_signal = NULL;
sem_t *P2_signal = NULL;
sem_t *P3_signal = NULL;
//sem_t sem_open(const char * name, int oflag, mode_t mode, unsigned int value)
//返回值sem_t 是一个结构，如果函数调用成功，则返回指向这个结构的指针，里面装着当前信号量的资源数。

int main(int argc, char *argv[])
{
    pid_t pid;
    P1_signal = sem_open("P1_signal",O_CREAT,0666,0);
    P2_signal = sem_open("P2_signal",O_CREAT,0666,0);
    P3_signal = sem_open("P3_signal",O_CREAT,0666,0);

    pid = fork();

    if(pid < 0)
    {
        printf("进程为创建失败！");

    }


    else if(pid == 0)
    {
        //sleep(1);
        sem_wait(P1_signal);
        printf("I am the process P2\n");
        sem_post(P1_signal);
        sem_post(P2_signal);

         pid = fork();

            if(pid < 0)
            {
                printf("进程为创建失败！");

            }

            else if(pid == 0)
            {
```

```
                sem_wait(P2_signal);
                sem_wait(P3_signal);
                printf("I am the process P4\n");

            }
    }

   else
    {
        printf("I am the process P1\n");
        sem_post(P1_signal);

        pid = fork();

        if(pid < 0)
        {
        printf("进程为创建失败！");

        }

        else if(pid == 0)
        {
            sem_wait(P1_signal);
            printf("I am the process P3\n");
            sem_post(P1_signal);
            sem_post(P3_signal);
            return 0;

        }
    }

    sem_close(P1_signal);
    sem_unlink("P1_signalname");
    sem_close(P2_signal);
    sem_unlink("P2_signalname");
    sem_close(P3_signal);
    sem_unlink("P3_signalname");
    return 0;

}
```

7. 运行结果

   运行指令：gcc -o 3-1.out 3-1.c -pthread

   一定注意pthread非linux系统的默认库， 需手动链接-线程库 -lpthread，否则会报以下错误：

```
[xaviershank@xaviershank 实验三]$ gcc 3-1.c -o 3-1
/usr/bin/ld: /tmp/ccPEkIpi.o: in function `main':
3-1.c:(.text+0x2b): undefined reference to `sem_open'
/usr/bin/ld: 3-1.c:(.text+0x52): undefined reference to `sem_open'
/usr/bin/ld: 3-1.c:(.text+0x79): undefined reference to `sem_open'
/usr/bin/ld: 3-1.c:(.text+0xb9): undefined reference to `sem_wait'
/usr/bin/ld: 3-1.c:(.text+0xd9): undefined reference to `sem_post'
/usr/bin/ld: 3-1.c:(.text+0xe8): undefined reference to `sem_post'
/usr/bin/ld: 3-1.c:(.text+0x10d): undefined reference to `sem_post'
/usr/bin/ld: 3-1.c:(.text+0x145): undefined reference to `sem_wait'
/usr/bin/ld: 3-1.c:(.text+0x165): undefined reference to `sem_post'
/usr/bin/ld: 3-1.c:(.text+0x174): undefined reference to `sem_post'
/usr/bin/ld: 3-1.c:(.text+0x1a8): undefined reference to `sem_wait'
/usr/bin/ld: 3-1.c:(.text+0x1b7): undefined reference to `sem_wait'
/usr/bin/ld: 3-1.c:(.text+0x1d7): undefined reference to `sem_close'
/usr/bin/ld: 3-1.c:(.text+0x1e3): undefined reference to `sem_unlink'
/usr/bin/ld: 3-1.c:(.text+0x1f2): undefined reference to `sem_close'
/usr/bin/ld: 3-1.c:(.text+0x1fe): undefined reference to `sem_unlink'
/usr/bin/ld: 3-1.c:(.text+0x20d): undefined reference to `sem_close'
/usr/bin/ld: 3-1.c:(.text+0x219): undefined reference to `sem_unlink'
collect2: 错误：ld 返回 1
[xaviershank@xaviershank 实验三]$ ^C
[xaviershank@xaviershank 实验三]$ gcc -o 3-1.out 3-1.c -lpthread
[xaviershank@xaviershank 实验三]$ ls
3-1.c  3-1.out
[xaviershank@xaviershank 实验三]$ ./3-1.out
```

运行结果：

```
[xaviershank@xaviershank 实验三]$ ./3-1.out
I am the process P1
I am the process P2
I am the process P3
I am the process P4
[xaviershank@xaviershank 实验三]$ ./3-1.out
I am the process P1
I am the process P2
I am the process P3
I am the process P4
[xaviershank@xaviershank 实验三]$ ./3-1.out
I am the process P1
I am the process P2
I am the process P3
I am the process P4
```

问题分析：

存在问题：每次P2都在P3前运行，似乎P2的竞争力强于P3?我猜想原因是P2线程要比P3早建立，所以有这种问题。

我们让P2 sleep(1)后：

```
        ......
        sleep(1);
        sem_wait(P1_signal);
        printf("I am the process P2\n");
        sem_post(P1_signal);
        sem_post(P2_signal);
       ......
```

```
[xaviershank@xaviershank 实验三]$ gcc -o 3-1.out 3-1.c -pthread
[xaviershank@xaviershank 实验三]$ ./3-1.out
I am the process P1
I am the process P3
[xaviershank@xaviershank 实验三]$ I am the process P2
I am the process P4
```

这样就人为地消除了这个问题。

题目二

火车票余票数ticketCount ，初始值为1000，有一个售票线程，一个退票线程，各循环执行多次。添加同步机制，使得结果始终正确。要求多次测试添加同步机制前后的实验效果。(说明：为了更容易产生并发错误，可以在适当的位置增加一些**pthread_yield()**，放弃**CPU**，并强制线程频繁切换，例如售票线程的关键代码：

temp=ticketCount;

pthread_yield();

temp=temp-1;

pthread_yield();

ticketCount=temp;

退票线程的关键代码：

temp=ticketCount;

pthread_yield();

temp=temp+1;

pthread_yield();

ticketCount=temp;

)

解答：

1. 售出票线程Sell();
2. 退回票线程Return();
3. 这两个是互斥事件，一个占用时，另一个必须等待。设置一个信号量flag，初始值为1；
4. **伪代码**如下：

   记录型信号量实现互斥：

```
Var flag: semaphore :=1;

Sell:
begin
    repeat
        wait(flag);
        ticketCount = ticketCount-1;
        signal(flag);
    until false;
end

return
begin
    repeat
        wait(flag);
        ticketcount = ticketCount+1;
        signal(flag);
    until false;
end
```

5. **源代码**

情况一：

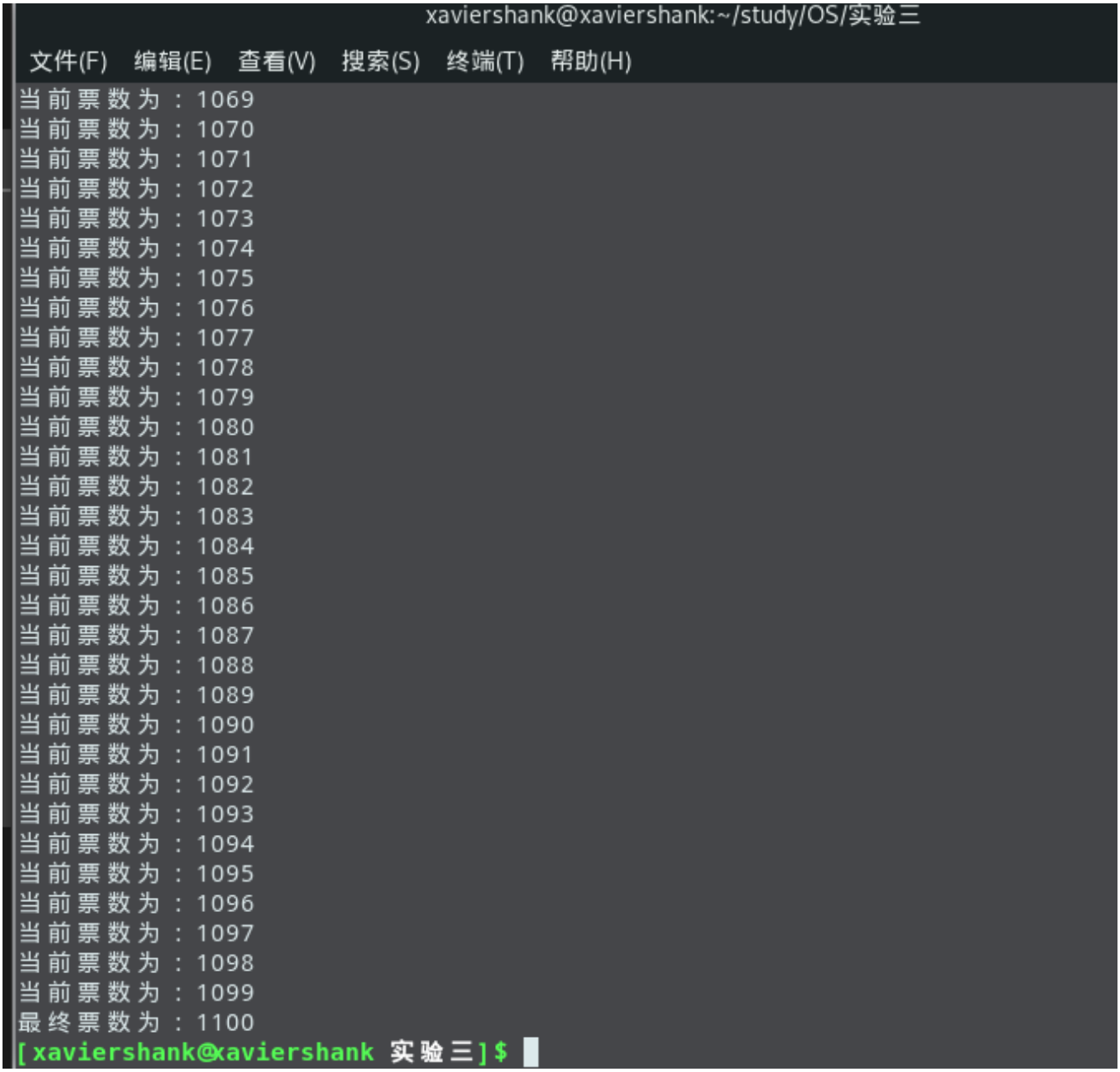
未加信号量机制，并且在卖票Sellticket( )的temp写回前加入sched_yield();

猜想：因为票数减少且未及时写回，而退票数目正常，会导致总票数增加：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sched.h>
#include <semaphore.h>

int ticketCount = 1000;
sem_t *flag = NULL;

void *SellTicket()
{
    for(int i=100;i>0;i--)
    {
    //wait(flag);
    int temp;
    printf("当前票数为：%d\n",ticketCount);
    temp = ticketCount;
    temp = temp - 1;
    sched_yield();
```

```
    ticketCount = temp;
    //signal(flag);
    }
}

void *ReturnTicket()
{
   for(int i=100;i>0;i--)
    {
    //wait(flag);
    printf("当前票数为：%d\n",ticketCount);
    int temp;
    temp = ticketCount;
    temp = temp + 1;
    //sched_yield();
    ticketCount = temp;
    //signal(flag);
    }

}

int main()
{
    pthread_t Sell,Return;
    flag = sem_open("flag",O_CREAT,0666,1);

    pthread_create(&Sell,NULL,SellTicket,NULL);
    pthread_join(Sell,NULL);

    pthread_create(&Return,NULL,ReturnTicket,NULL);
    pthread_join(Return,NULL);

    sem_close(flag);
    sem_unlink("flag");

    printf("最终票数为：%d \n",ticketCount);
    return 0;
}
```

运行结果：

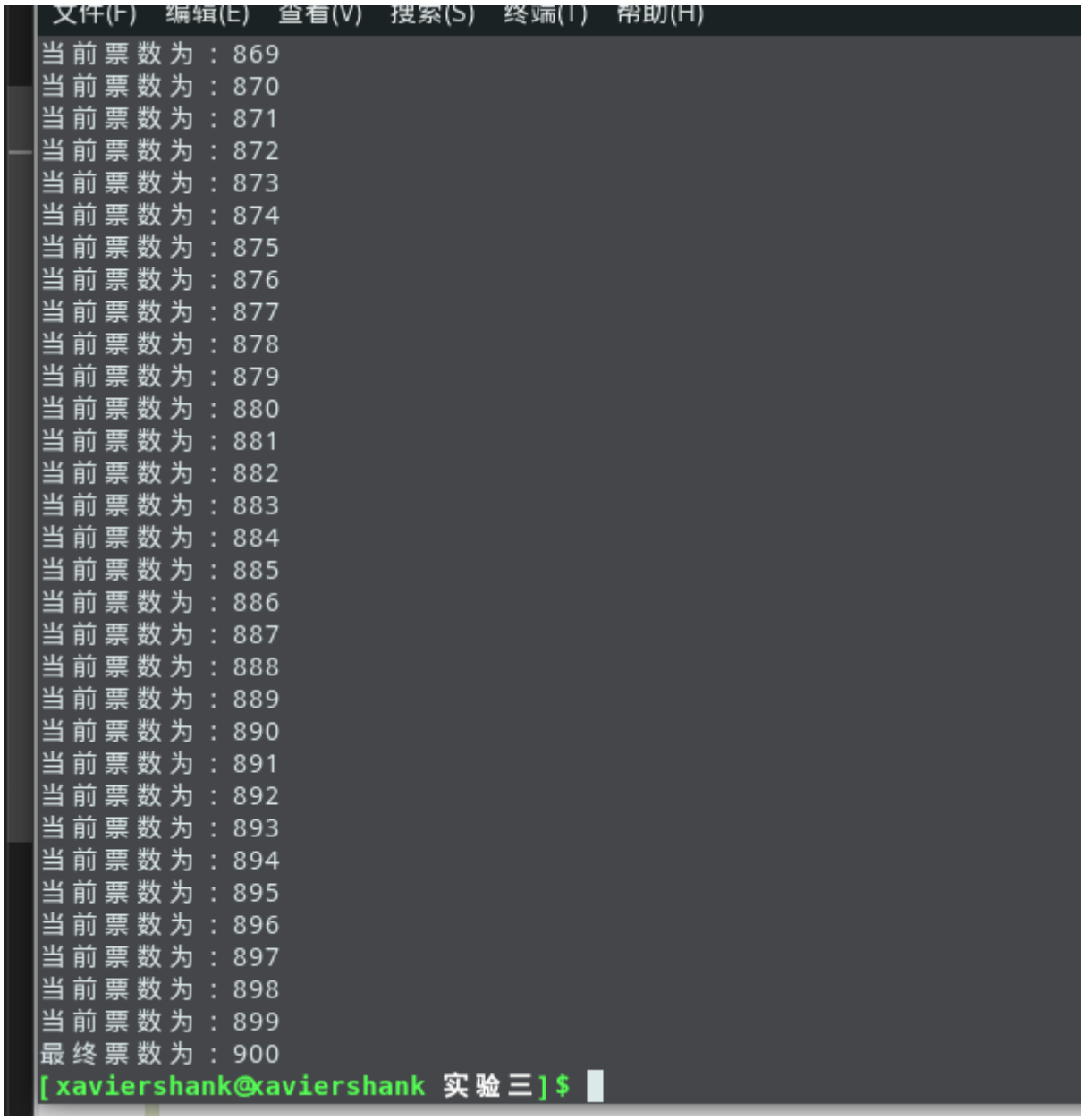


情况二：

未加信号量机制，并且在退票Returnticket( )的temp写回前加入sched_yield();

猜想：因为票数增加且未及时写回，而卖票数目正常，会导致总票数减少：

运行结果：



情况三：

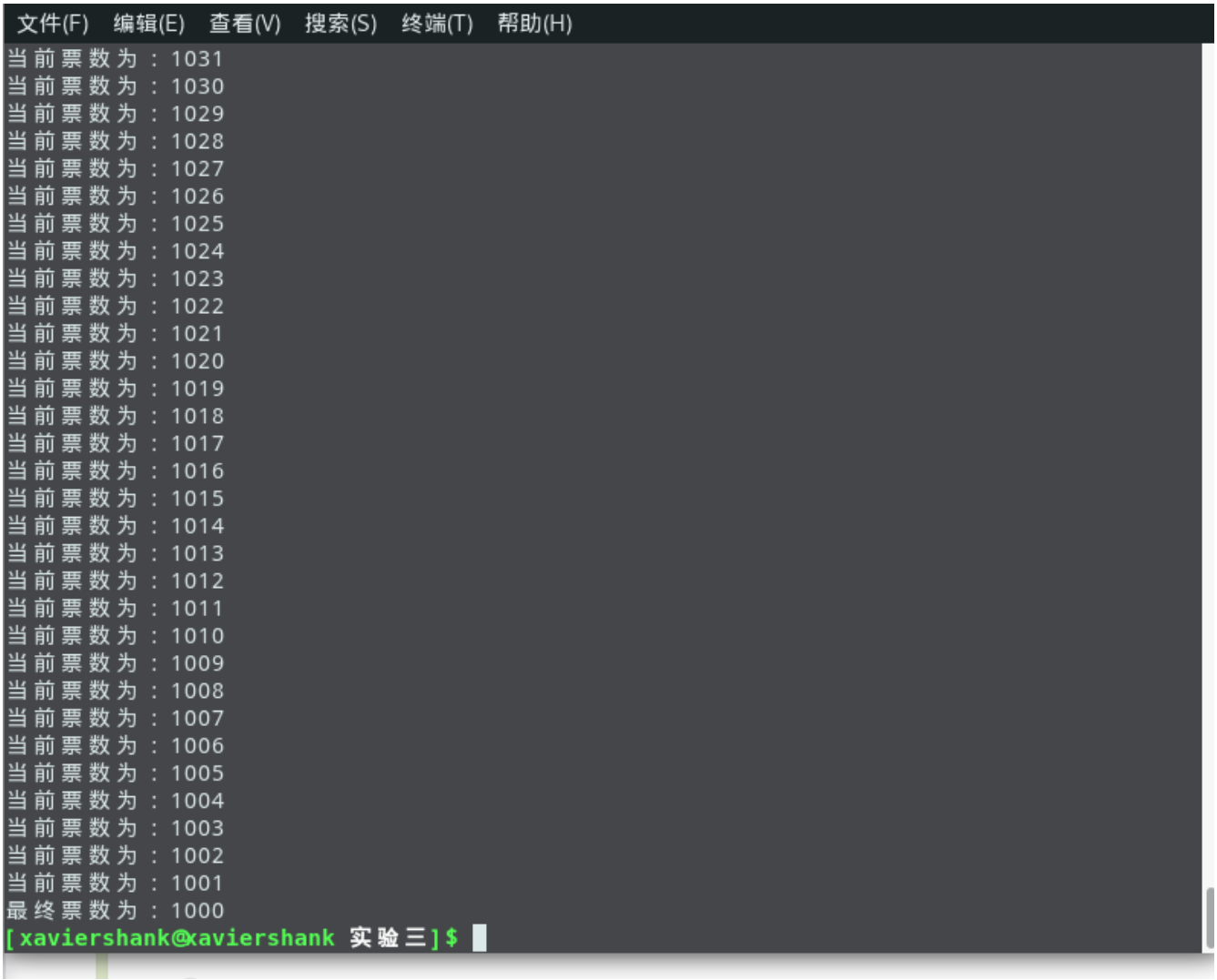
加入信号量机制：

源代码

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sched.h>
#include <semaphore.h>
```

```c
int ticketCount = 1000;
sem_t *flag = NULL;

void *SellTicket()
{
    for(int i=100;i>0;i--)
    {
    sem_wait(flag);
    int temp;
    printf("当前票数为：%d\n",ticketCount);
    temp = ticketCount;
    temp = temp - 1;
    //sched_yield();
    ticketCount = temp;
    sem_post(flag);
    }
}

void *ReturnTicket()
{
   for(int i=100;i>0;i--)
    {
    sem_wait(flag);
    printf("当前票数为：%d\n",ticketCount);
    int temp;
    temp = ticketCount;
    temp = temp + 1;
    //sched_yield();
    ticketCount = temp;
    sem_post(flag);
    }

}

int main()
{
    pthread_t Sell,Return;
    flag = sem_open("flag",O_CREAT,0666,1);

    pthread_create(&Sell,NULL,SellTicket,NULL);
    pthread_join(Sell,NULL);

    pthread_create(&Return,NULL,ReturnTicket,NULL);
    pthread_join(Return,NULL);

    sem_close(flag);
    sem_unlink("flag");

    printf("最终票数为：%d \n",ticketCount);
    return 0;
}
```

运行结果：



题目三

一个生产者一个消费者线程同步。设置一个线程共享的缓冲区，char buf[10]。一个线程不断从键盘输入字符到buf,一个线程不断的把buf的内容输出到显示器。要求输出的和输入的字符和顺序完全一致。（在输出线程中，每次输出睡眠一秒钟，然后以不同的速度输入测试输出是否正确）。要求多次测试添加同步机制前后的实验效果。

解答：

1. 输入线程 Producer();
2. 输出线程 Consumer();

3. 信号量设置：
    1. 信号量empty,适用于Producer()，查看数组中是否有空位，有则输入。

       输入后，signal(full)，因为新加入了数据。

       对于输入超过数组界限的问题，我采用取模mod10，这样会覆盖前面的内容，但不会产生数组越界的问题。
    2. 信号量full,适用于Consumer()，查看数组中是否有数据，有则读取，没有则等待。

       输出后，signal(empty)，读走了数据，增加空位。
4. **伪代码**

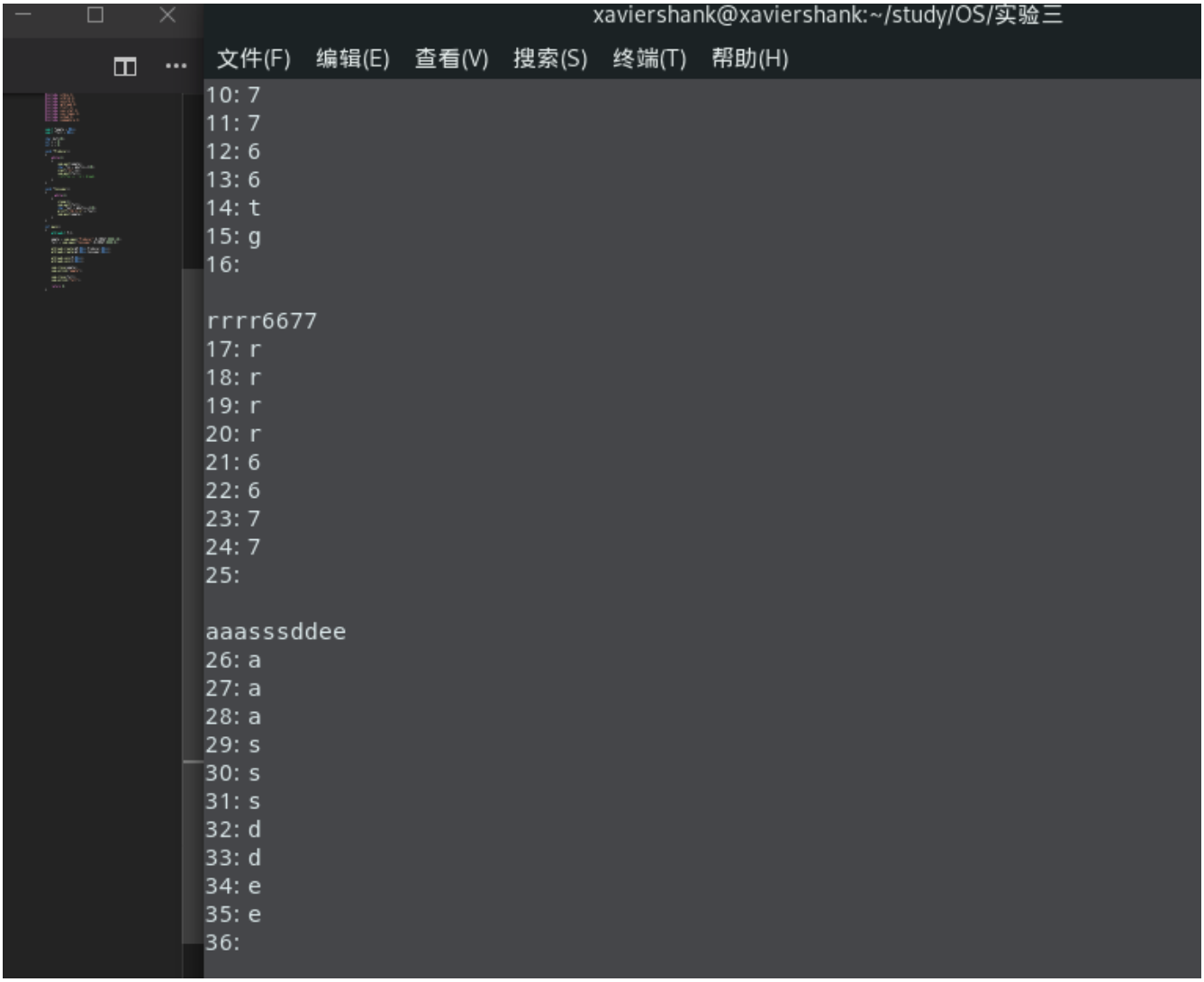
```
Producer
Repeat
    wait(empty);
    Read and put into buffer;
    signal(full);

Consumer
Repeat
    wait(full);
    Read from buffer;
    signal(empty);
```

5.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sched.h>
#include <semaphore.h>


sem_t *empty = NULL;
sem_t *full = NULL;

char buf[10];
int i = 0;
int j = 0;

void *Producer()
{
    while(1)
    {
        sem_wait(empty);
        char *in = &buf[i++%10];
        scanf("%c",in);
        sem_post(full);
```

```
        if(*in == '@') break;
    }
}

void *Consumer()
{
      while(1)
    {
        sleep(1);
        sem_wait(full);
        char *out = &buf[j++%10];
        printf("%d:%c\n",j,*out);
        sem_post(empty);
    }
}

int main()
{
    pthread_t P,C;

    empty = sem_open("Producer",O_CREAT,0666,10);
    full = sem_open("Consumer",O_CREAT,0666,0);

    pthread_create(&P,NULL,Producer,NULL);
    pthread_join(P,NULL);

    pthread_create(&C,NULL,Consumer,NULL);
    pthread_join(C,NULL);

    sem_close(empty);
    sem_unlink("empty");

    sem_close(full);
    sem_unlink("full");

    return 0;
}
```

运行结果：



题目四

a）

通过实验测试，验证共享内存的代码中，receiver能否正确读出sender发送的字符串？如果把其中互斥的代码删除，观察实验结果有何不同？如果在发送和接收进程中打印输出共享内存地址，他们是否相同，为什么？

```
/*
 * Filename: Sender.c
 * Description:
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <string.h>
```

```c
int main(int argc, char *argv[])
{
    key_t  key;
    int shm_id;
    int sem_id;
    int value = 0;

    //1.Product the key
    key = ftok(".", 0xFF);

    //2. Creat semaphore for visit the shared memory
    sem_id = semget(key, 1, IPC_CREAT|0644);
    if(-1 == sem_id)
    {
        perror("semget");
        exit(EXIT_FAILURE);
    }

    //3. init the semaphore, sem=0
    if(-1 == (semctl(sem_id, 0, SETVAL, value)))
    {
        perror("semctl");
        exit(EXIT_FAILURE);
    }

    //4. Creat the shared memory(1K bytes)
    shm_id = shmget(key, 1024, IPC_CREAT|0644);
    if(-1 == shm_id)
    {
        perror("shmget");
        exit(EXIT_FAILURE);
    }

    //5. attach the shm_id to this process
    char *shm_ptr;
    shm_ptr = shmat(shm_id, NULL, 0);
    if(NULL == shm_ptr)
    {
        perror("shmat");
        exit(EXIT_FAILURE);
    }

    //6. Operation procedure
    struct sembuf sem_b;
    sem_b.sem_num = 0;      //first sem(index=0)
    sem_b.sem_flg = SEM_UNDO;
    sem_b.sem_op = 1;           //Increase 1,make sem=1

    while(1)
    {
        if(0 == (value = semctl(sem_id, 0, GETVAL)))
        {
```

```
            printf("\nNow, snd message process running:\n");
            printf("\tInput the snd message:  ");
            scanf("%s", shm_ptr);

            if(-1 == semop(sem_id, &sem_b, 1))
            {
                perror("semop");
                exit(EXIT_FAILURE);
            }
        }

        //if enter "end", then end the process
        if(0 == (strcmp(shm_ptr ,"end")))
        {
            printf("\nExit sender process now!\n");
            break;
        }
    }

    shmdt(shm_ptr);

    return 0;
}
```

```
/*
 * Filename: Receiver.c
 * Description:
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <string.h>

int main(int argc, char *argv[])
{
    key_t  key;
    int shm_id;
    int sem_id;
    int value = 0;

    //1.Product the key
    key = ftok(".", 0xFF);

    //2. Creat semaphore for visit the shared memory
    sem_id = semget(key, 1, IPC_CREAT|0644);
    if(-1 == sem_id)
    {
        perror("semget");
```

```
        exit(EXIT_FAILURE);
    }

    //3. init the semaphore, sem=0
    if(-1 == (semctl(sem_id, 0, SETVAL, value)))
    {
        perror("semctl");
        exit(EXIT_FAILURE);
    }

    //4. Creat the shared memory(1K bytes)
    shm_id = shmget(key, 1024, IPC_CREAT|0644);
    if(-1 == shm_id)
    {
        perror("shmget");
        exit(EXIT_FAILURE);
    }

    //5. attach the shm_id to this process
    char *shm_ptr;
    shm_ptr = shmat(shm_id, NULL, 0);
    if(NULL == shm_ptr)
    {
        perror("shmat");
        exit(EXIT_FAILURE);
    }

    //6. Operation procedure
    struct sembuf sem_b;
    sem_b.sem_num = 0;      //first sem(index=0)
    sem_b.sem_flg = SEM_UNDO;
    sem_b.sem_op = -1;           //Increase 1,make sem=1

    while(1)
    {
        if(1 == (value = semctl(sem_id, 0, GETVAL)))
        {
            printf("\nNow, receive message process running:\n");
            printf("\tThe message is : %s\n", shm_ptr);

            if(-1 == semop(sem_id, &sem_b, 1))
            {
                perror("semop");
                exit(EXIT_FAILURE);
            }
        }

        //if enter "end", then end the process
        if(0 == (strcmp(shm_ptr ,"end")))
        {
            printf("\nExit the receiver process now!\n");
            break;
        }
```

```
    }

    shmdt(shm_ptr);
    //7. delete the shared memory
    if(-1 == shmctl(shm_id, IPC_RMID, NULL))
    {
        perror("shmctl");
        exit(EXIT_FAILURE);
    }

    //8. delete the semaphore
    if(-1 == semctl(sem_id, 0, IPC_RMID))
    {
        perror("semctl");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

b)有名管道和无名管道通信系统调用是否已经实现了同步机制？通过实验验证，发送者和接收者如何同步的。比如，在什么情况下，发送者会阻塞，什么情况下，接收者会阻塞？

c) 消息通信系统调用是否已经实现了同步机制？通过实验验证，发送者和接收者如何同步的。比如，在什么情况下，发送者会阻塞，什么情况下，接收者会阻塞？

题目五

阅读Pintos操作系统，找到并阅读进程上下文切换的代码，说明实现的保存和恢复的上下文内容以及进程切换的工作流程。

解答：