# 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。

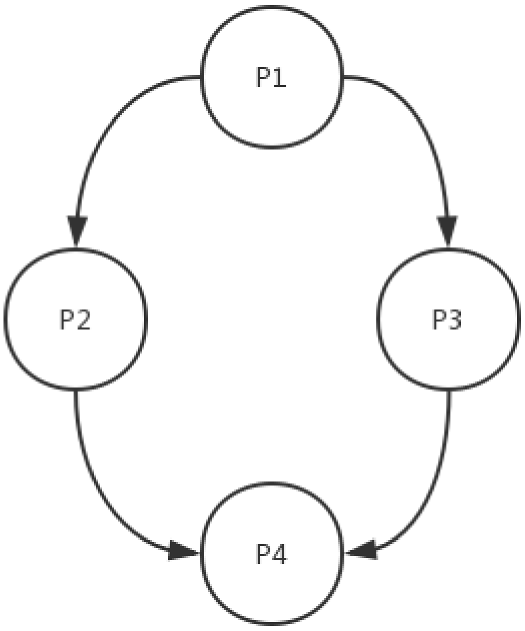
## 2. 实验题目

### 题目一

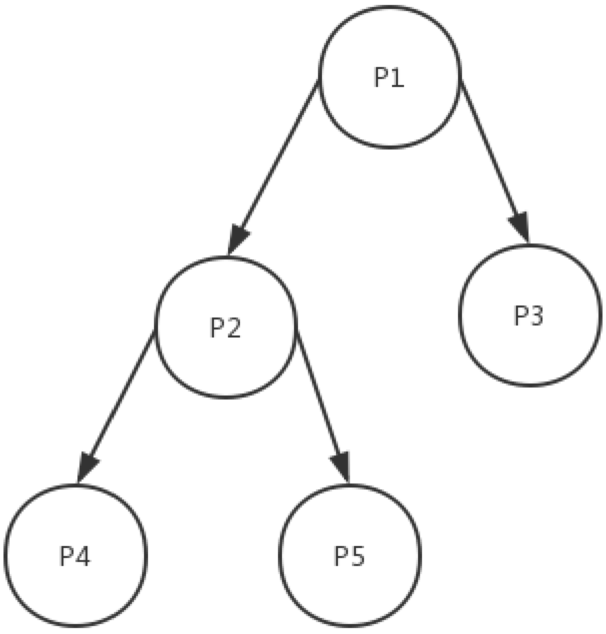
通过fork的方式，产生4个进程P1,P2,P3,P4，每个进程打印输出自己的名字，例如P1输出“I am the process P1”。要求P1最先执行，P2、P3互斥执行，P4最后执行。通过多次测试验证实现是否正确。

#### 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();
```

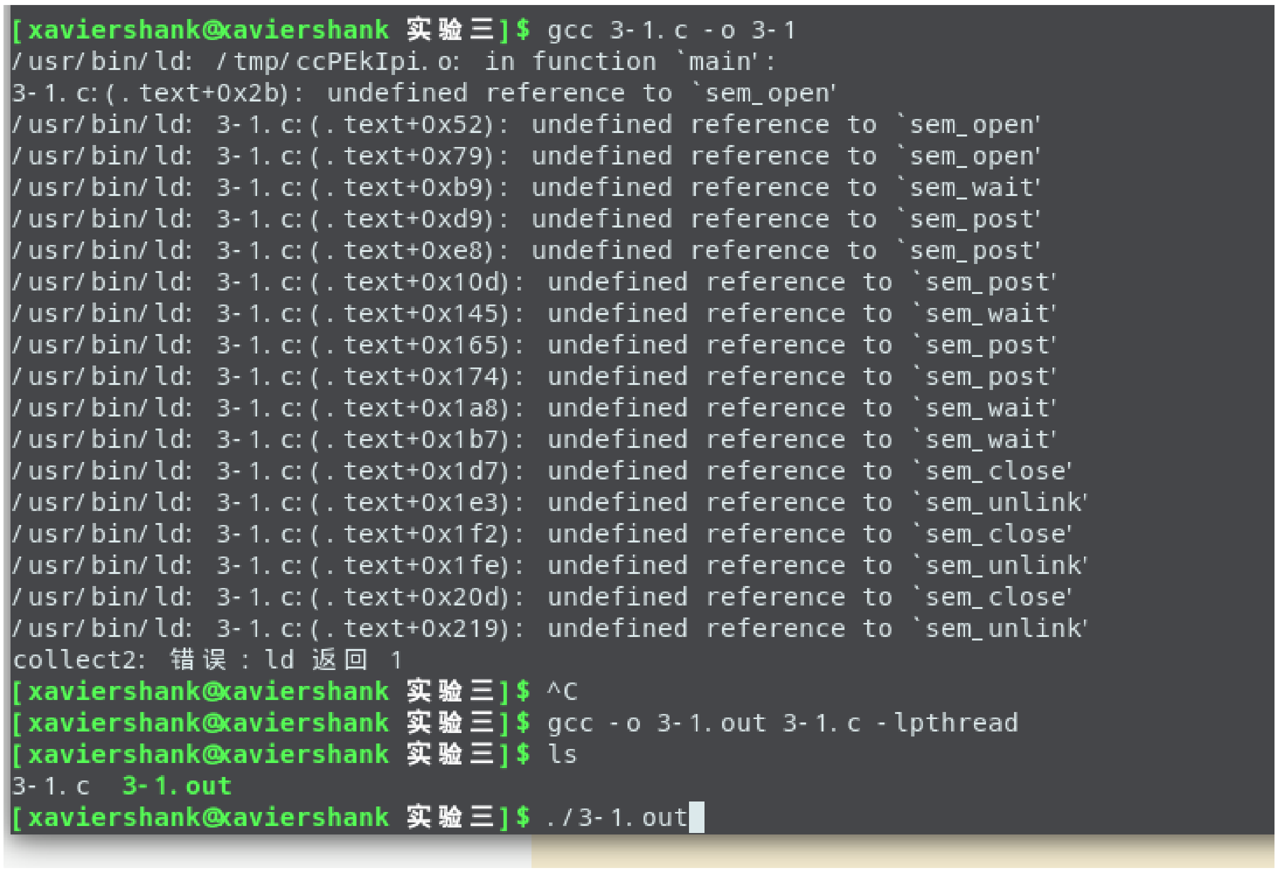
```
     if(pid < 0)
     {
         printf("进程为创建失败！");

     }


     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

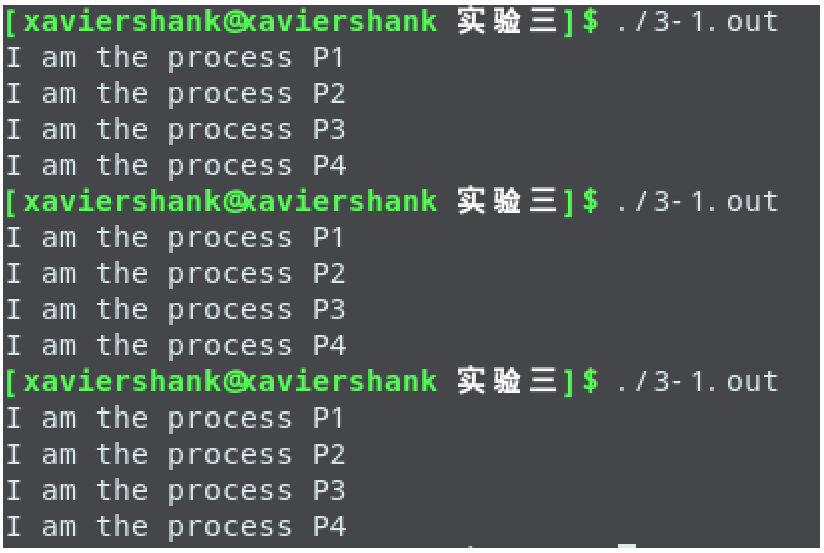
   运行指令：gcc -o 3-1.out 3-1.c -pthread

   **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：



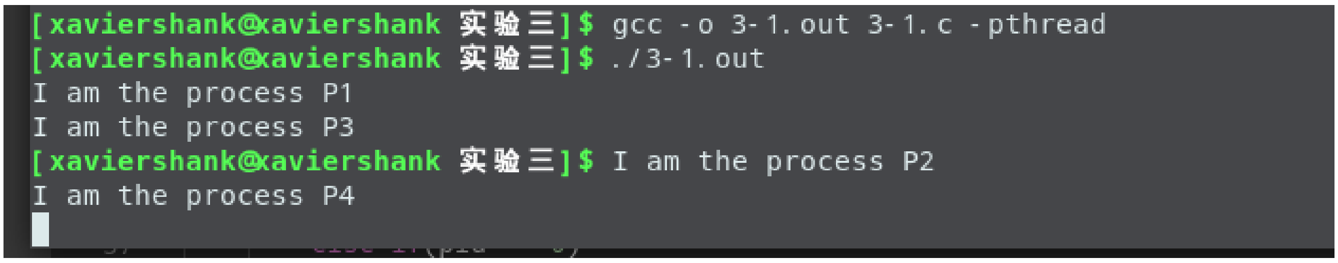
```
[xaviershank@xaviershank 实验三]$ ./3-1.out
I am the process P1
I am the process P2
I am the process P3
I am the process P4
[xaviershank@xaviershank 实验三]$ ./3-1.out
I am the process P1
I am the process P2
I am the process P3
I am the process P4
[xaviershank@xaviershank 实验三]$ ./3-1.out
I am the process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

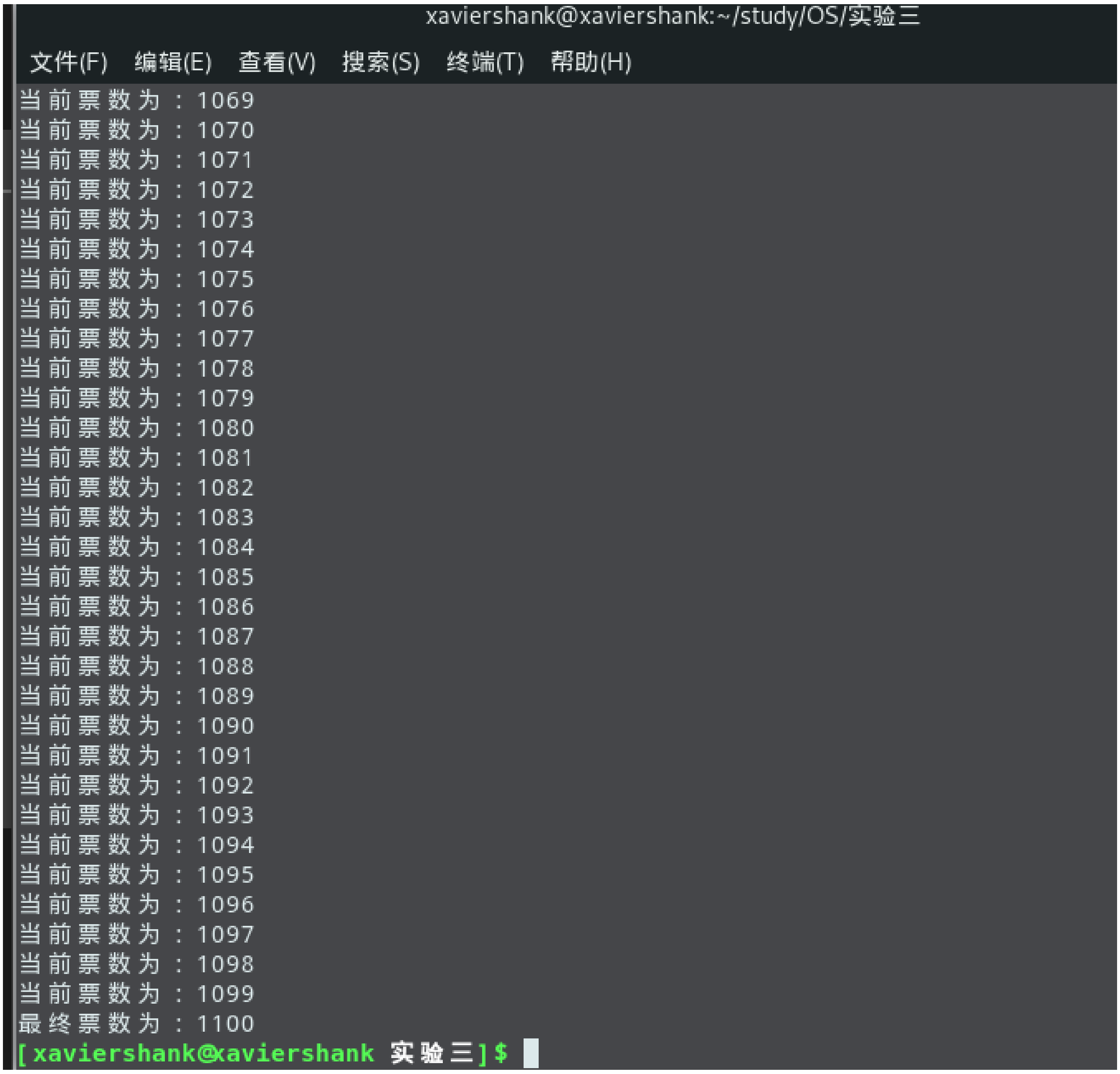
   情况一：

   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：

```c
#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：


xaviershank@xaviershank:~/study/OS/实验三
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

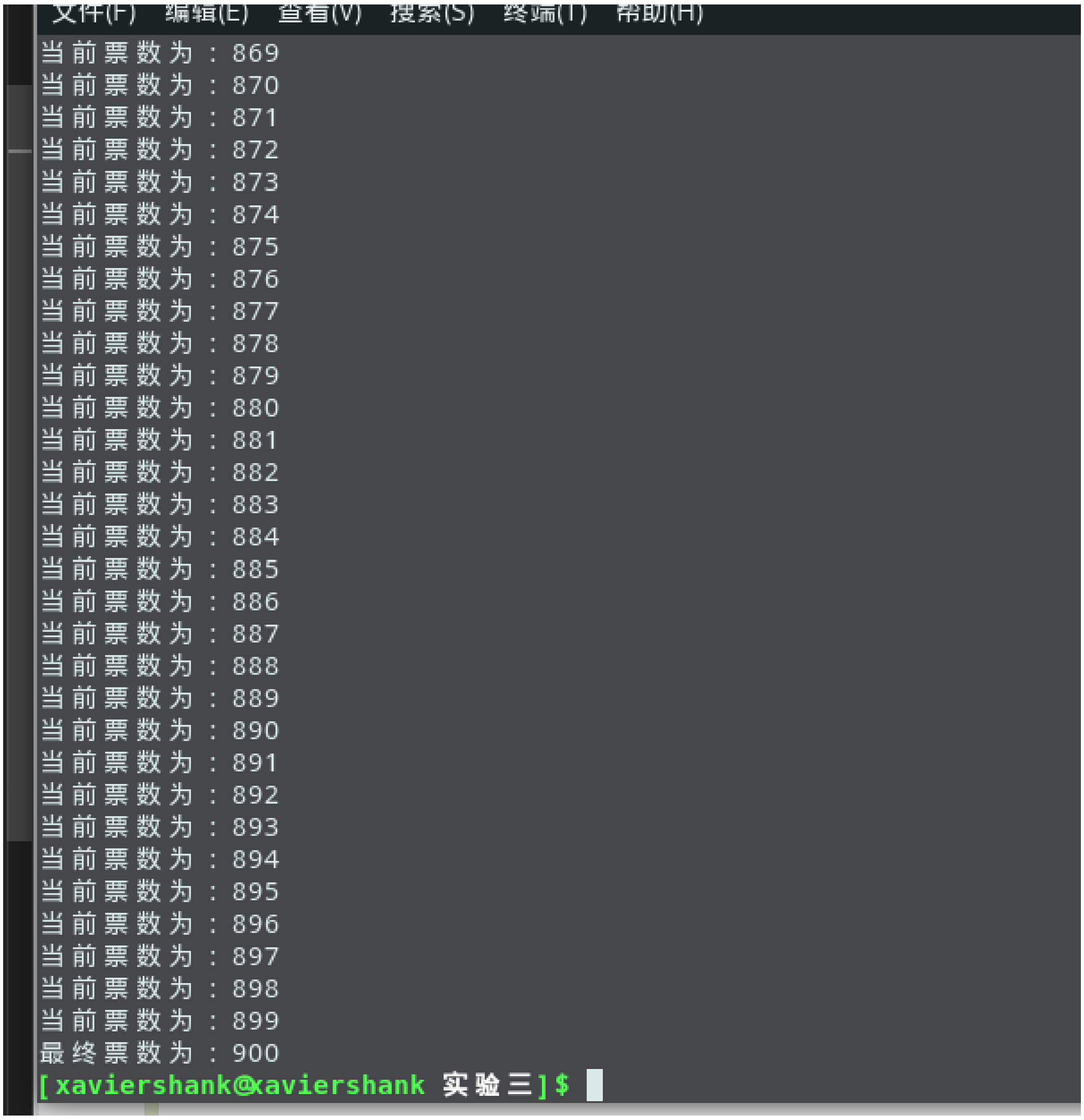
```
当前票数为：1069
当前票数为：1070
当前票数为：1071
当前票数为：1072
当前票数为：1073
当前票数为：1074
当前票数为：1075
当前票数为：1076
当前票数为：1077
当前票数为：1078
当前票数为：1079
当前票数为：1080
当前票数为：1081
当前票数为：1082
当前票数为：1083
当前票数为：1084
当前票数为：1085
当前票数为：1086
当前票数为：1087
当前票数为：1088
当前票数为：1089
当前票数为：1090
当前票数为：1091
当前票数为：1092
当前票数为：1093
当前票数为：1094
当前票数为：1095
当前票数为：1096
当前票数为：1097
当前票数为：1098
当前票数为：1099
最终票数为：1100
[xaviershank@xaviershank 实验三]$
```

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：



```
当前票数为 : 869
当前票数为 : 870
当前票数为 : 871
当前票数为 : 872
当前票数为 : 873
当前票数为 : 874
当前票数为 : 875
当前票数为 : 876
当前票数为 : 877
当前票数为 : 878
当前票数为 : 879
当前票数为 : 880
当前票数为 : 881
当前票数为 : 882
当前票数为 : 883
当前票数为 : 884
当前票数为 : 885
当前票数为 : 886
当前票数为 : 887
当前票数为 : 888
当前票数为 : 889
当前票数为 : 890
当前票数为 : 891
当前票数为 : 892
当前票数为 : 893
当前票数为 : 894
当前票数为 : 895
当前票数为 : 896
当前票数为 : 897
当前票数为 : 898
当前票数为 : 899
最终票数为 : 900
[xaviershank@xaviershank 实验三]$
```

情况三：

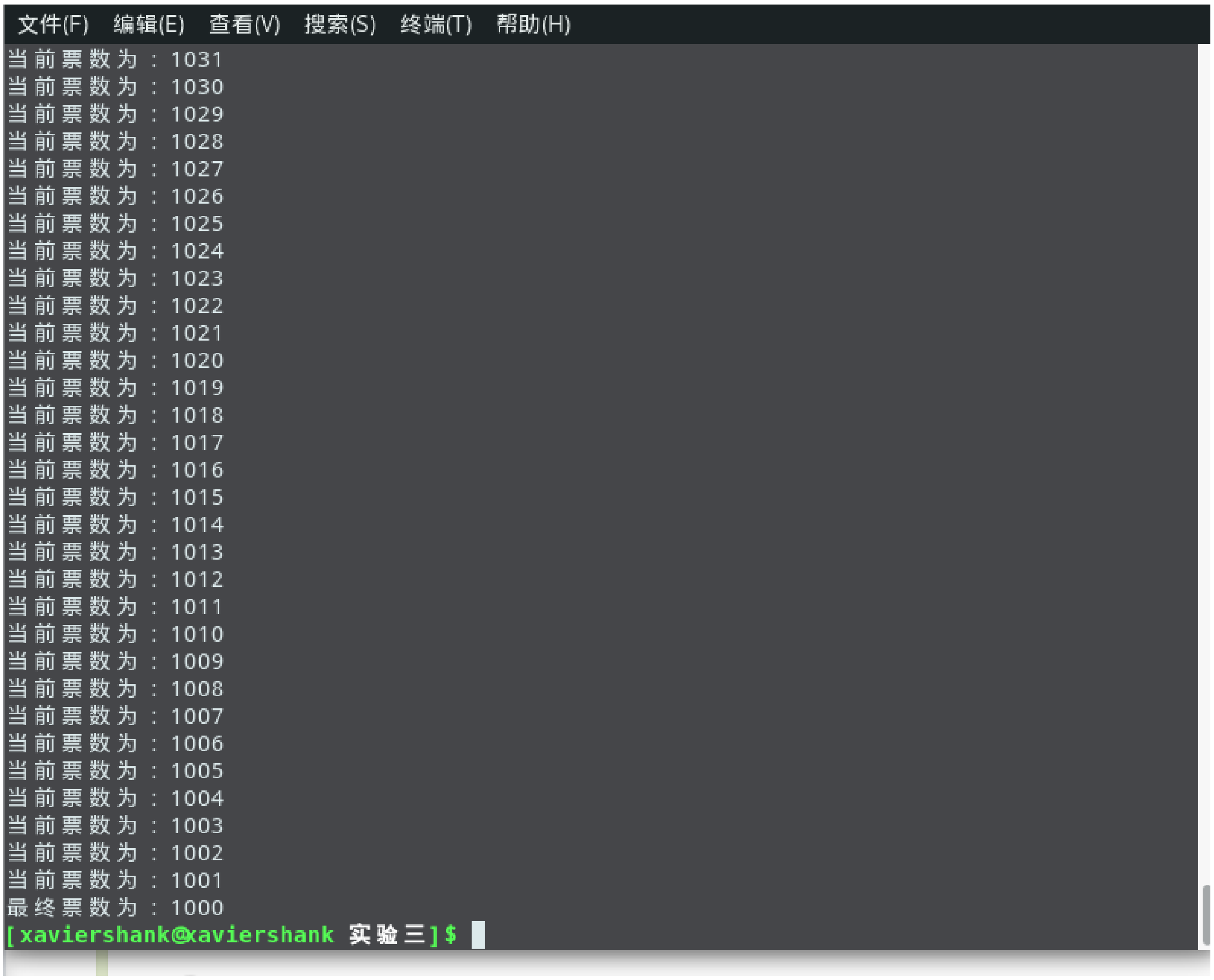
加入信号量机制：

**源代码**

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#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--)
    {
    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：



```
当前票数为：1031
当前票数为：1030
当前票数为：1029
当前票数为：1028
当前票数为：1027
当前票数为：1026
当前票数为：1025
当前票数为：1024
当前票数为：1023
当前票数为：1022
当前票数为：1021
当前票数为：1020
当前票数为：1019
当前票数为：1018
当前票数为：1017
当前票数为：1016
当前票数为：1015
当前票数为：1014
当前票数为：1013
当前票数为：1012
当前票数为：1011
当前票数为：1010
当前票数为：1009
当前票数为：1008
当前票数为：1007
当前票数为：1006
当前票数为：1005
当前票数为：1004
当前票数为：1003
当前票数为：1002
当前票数为：1001
最终票数为：1000
[xaviershank@xaviershank 实验三]$
```

## 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**

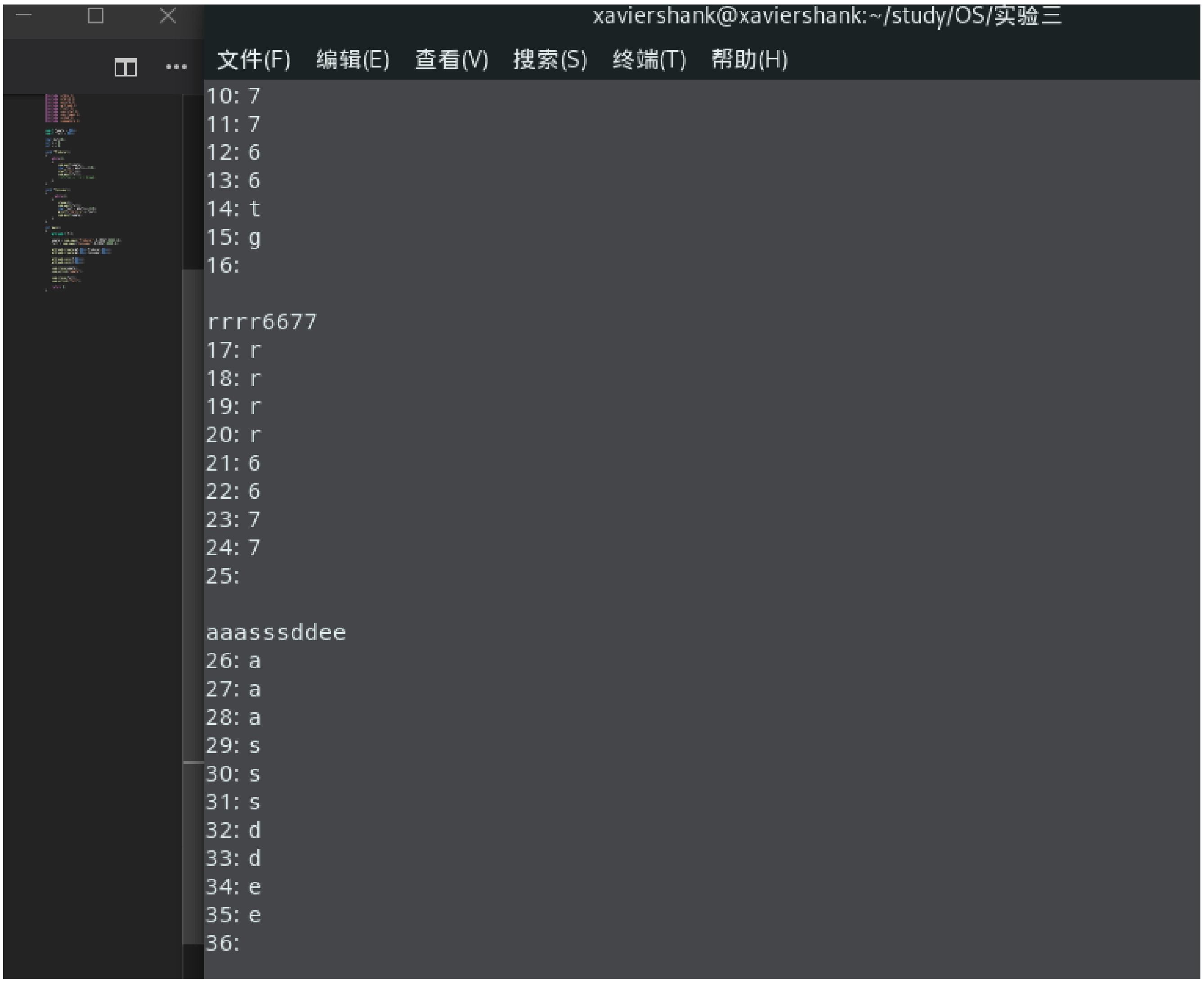
```
Producer
Repeat
    wait(empty);
    Read and put into buffer;
    signal(full);

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;
```

```c
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：


xaviershank@xaviershank:~/study/OS/实验三
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

```
10: 7
11: 7
12: 6
13: 6
14: t
15: g
16: 

rrrr6677
17: r
18: r
19: r
20: r
21: 6
22: 6
23: 7
24: 7
25: 

aaasssddee
26: a
27: a
28: a
29: s
30: s
31: s
32: d
33: d
34: e
35: e
36: 
```

## 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？

1. 源代码：

   发送者：

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

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

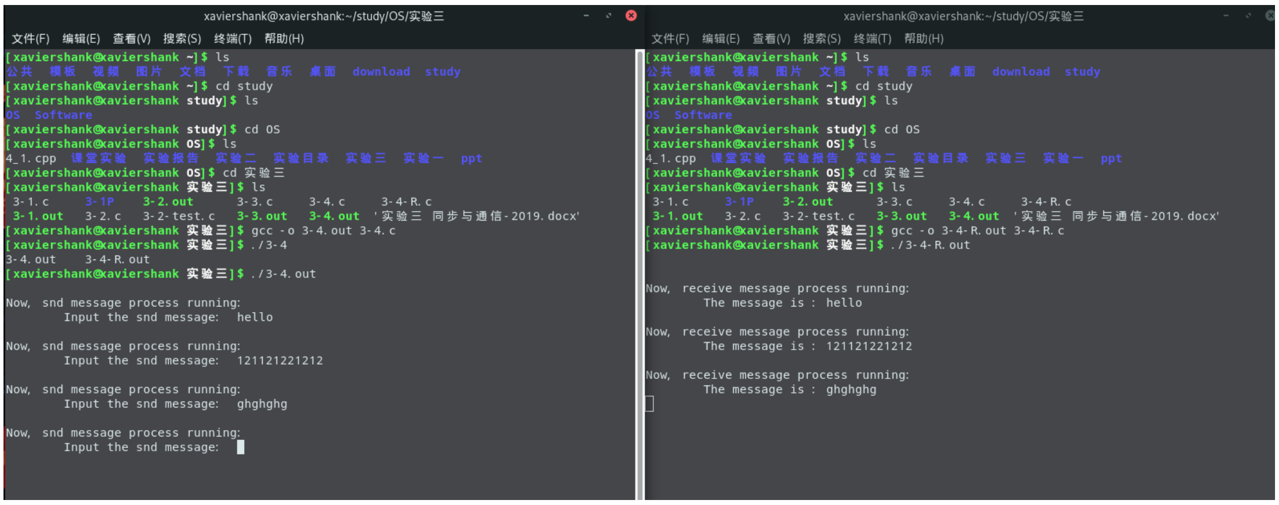
```c
#include <sys/shm.h>
#include <sys/types.h>
#include <string.h>

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(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;
}
```

接受者：

```
/*
 * 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
```

```c
        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;
}
```

2. 实验测试



```
xaviershank@xaviershank:~/study/OS/实验三
文件(F)  编辑(E)  查看(V)  搜索(S)  终端(T)  帮助(H)
[xaviershank@xaviershank ~]$ ls
公共  模板  视频  图片  文档  下载  音乐  桌面  download  study
[xaviershank@xaviershank ~]$ cd study
[xaviershank@xaviershank study]$ ls
OS  Software
[xaviershank@xaviershank study]$ cd OS
[xaviershank@xaviershank OS]$ ls
4_1.cpp  课堂实验  实验报告  实验二  实验目录  实验三  实验一  ppt
[xaviershank@xaviershank OS]$ cd 实验三
[xaviershank@xaviershank 实验三]$ ls
 3-1.c      3-1P    3-2.out       3-3.c      3-4.c      3-4-R.c
 3-1.out   3-2.c   3-2-test.c   3-3.out   3-4.out   '实验三 同步与通信-2019.docx'
[xaviershank@xaviershank 实验三]$ gcc -o 3-4.out 3-4.c
[xaviershank@xaviershank 实验三]$ ./3-4
3-4.out    3-4-R.out
[xaviershank@xaviershank 实验三]$ ./3-4.out

Now,  snd message process running:
        Input the snd message:   hello

Now,  snd message process running:
        Input the snd message:   121121221212

Now,  snd message process running:
        Input the snd message:   ghghghg

Now,  snd message process running:
        Input the snd message:   
```

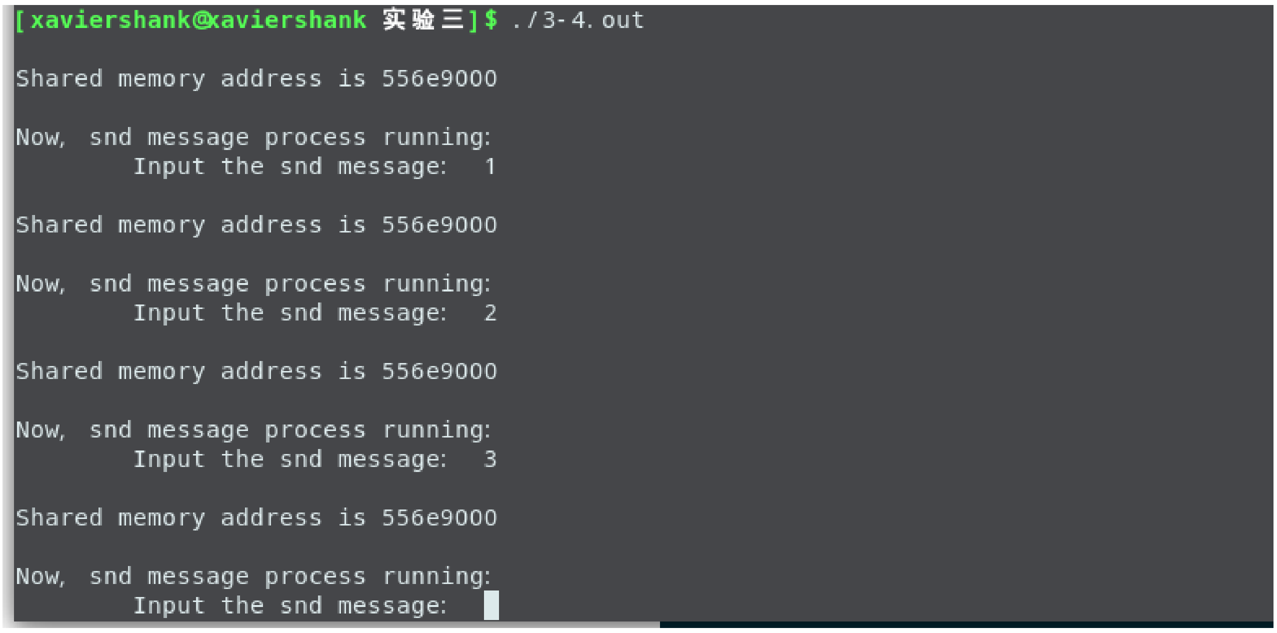
```
xaviershank@xaviershank:~/study/OS/实验三
文件(F)  编辑(E)  查看(V)  搜索(S)  终端(T)  帮助(H)
[xaviershank@xaviershank ~]$ ls
公共  模板  视频  图片  文档  下载  音乐  桌面  download  study
[xaviershank@xaviershank ~]$ cd study
[xaviershank@xaviershank study]$ ls
OS  Software
[xaviershank@xaviershank study]$ cd OS
[xaviershank@xaviershank OS]$ ls
4_1.cpp  课堂实验  实验报告  实验二  实验目录  实验三  实验一  ppt
[xaviershank@xaviershank OS]$ cd 实验三
[xaviershank@xaviershank 实验三]$ ls
 3-1.c      3-1P    3-2.out       3-3.c      3-4.c      3-4-R.c
 3-1.out   3-2.c   3-2-test.c   3-3.out   3-4.out   '实验三 同步与通信-2019.docx'
[xaviershank@xaviershank 实验三]$ gcc -o 3-4-R.out 3-4-R.c
[xaviershank@xaviershank 实验三]$ ./3-4-R.out


Now,  receive message process running:
        The message is :  hello

Now,  receive message process running:
        The message is :  121121221212

Now,  receive message process running:
        The message is :  ghghghg

```

结论：receiver能正确读出sender发送的字符串。

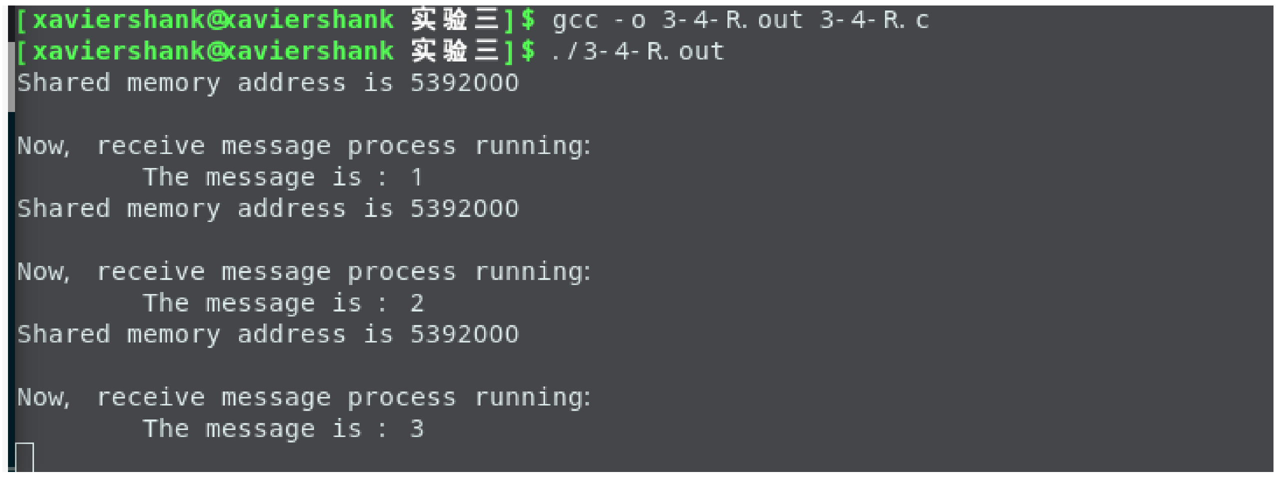
3. 删除代码中的互斥部分：

删除代码中的互斥部分，即删除信号量后发现，receiver在不停地读共享内存，并且速度很快，会出现重复读的现象。

4. 打印输出共享内存地址



```
[xaviershank@xaviershank 实验三]$ ./3-4.out

Shared memory address is 556e9000

Now, snd message process running:
        Input the snd message:  1

Shared memory address is 556e9000

Now, snd message process running:
        Input the snd message:  2

Shared memory address is 556e9000

Now, snd message process running:
        Input the snd message:  3

Shared memory address is 556e9000

Now, snd message process running:
        Input the snd message:
```



```
[xaviershank@xaviershank 实验三]$ gcc -o 3-4-R.out 3-4-R.c
[xaviershank@xaviershank 实验三]$ ./3-4-R.out
Shared memory address is 5392000

Now, receive message process running:
        The message is : 1
Shared memory address is 5392000

Now, receive message process running:
        The message is : 2
Shared memory address is 5392000

Now, receive message process running:
        The message is : 3
```

**惊奇地发现，sender和receiver的共享内存地址竟然不同！为什么呢？我的理解是因为操作系统给进程分配的是虚拟内存，相应每个进程会有自己的虚拟内存分配地址。又因为运行的两个进程在初始化的时候使用了shmat函数，此函数的作用是将共享内存空间挂载到进程中就是对进程的虚拟内存映射到共享内存的物理内存，从而实现内存的共享。所以虽然我们打印出来的内存地址不一样，但是它们实际映射的物理内存地址是一样的。**

## b）

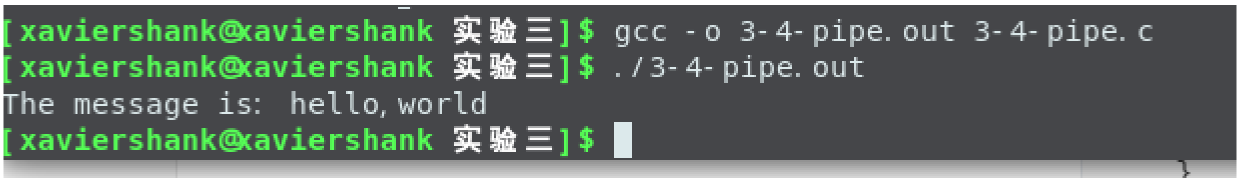
有名管道和无名管道通信系统调用是否已经实现了同步机制？通过实验验证，发送者和接收者如何同步的。比如，在什么情况下，发送者会阻塞，什么情况下，接收者会阻塞？

1. 源代码
    1. 无名管道 pipe.c

       多用于亲缘关系进程间通信，方向为**单向**；为**阻塞**读写；通信进程双方退出后自动消失

```
/*
 * Filename: pipe.c
 */
```

```
#include <stdio.h>
#include <unistd.h>     //for pipe()
#include <string.h>     //for memset()
#include <stdlib.h>     //for exit()

int main()
{
    int fd[2];
    char buf[20];
    if(-1 == pipe(fd))
    {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    write(fd[1], "hello,world", 12);
    memset(buf, '\0', sizeof(buf));

    read(fd[0], buf, 12);
    printf("The message is: %s\n", buf);

    return 0;
}
```



```
[xaviershank@xaviershank 实验三]$ gcc -o 3-4-pipe.out 3-4-pipe.c
[xaviershank@xaviershank 实验三]$ ./3-4-pipe.out
The message is: hello,world
[xaviershank@xaviershank 实验三]$ 
```

由实验结果可见，无名管道通信实现了同步机制。

2. 有名管道

   fifo_sen.c

   有名管道也实现了同步机制：

   对于以只读方式（O_RDONLY）打开的FIFO文件，如果open调用是阻塞的（即第二个参数为O_RDONLY），除非有一个进程以写方式打开同一个FIFO，否则它不会返回；如果open调用是非阻塞的的（即第二个参数为O_RDONLY | O_NONBLOCK），则即使没有其他进程以写方式打开同一个FIFO文件，open调用将成功并立即返回。

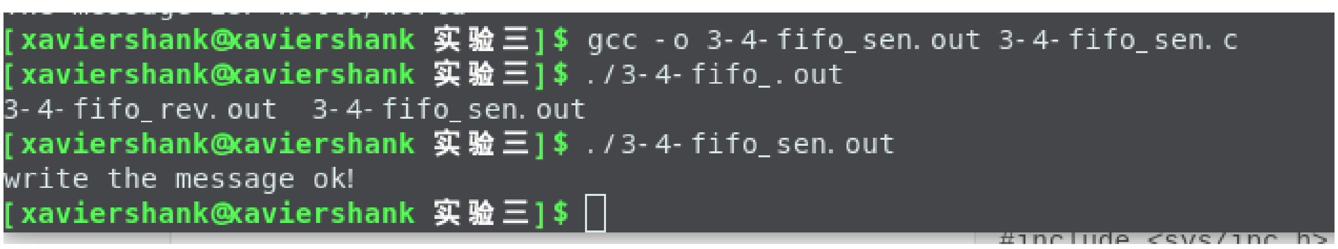
   对于以只写方式（O_WRONLY）打开的FIFO文件，如果open调用是阻塞的（即第二个参数为O_WRONLY），open调用将被阻塞，直到有一个进程以只读方式打开同一个FIFO文件为止；如果open调用是非阻塞的（即第二个参数为O_WRONLY | O_NONBLOCK），open总会立即返回，但如果没有其他进程以只读方式打开同一个FIFO文件，open调用将返回-1，并且FIFO也不会被打开。

以下是对实验的验证：

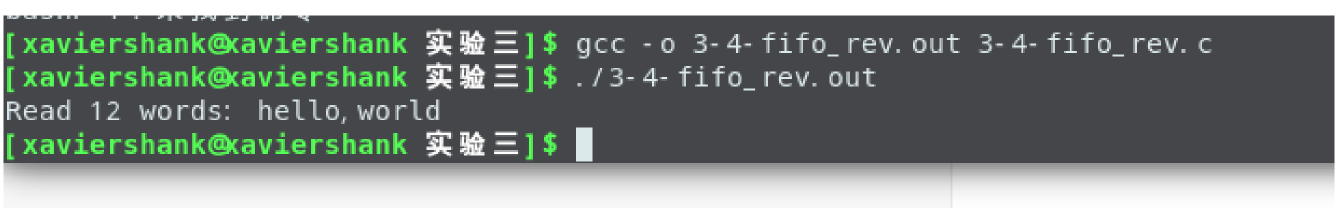
```c
/*
 *File: fifo_send.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <fcntl.h>


#define FIFO "/tmp/my_fifo"

int main()
{
    char buf[] = "hello,world";

    //`. check the fifo file existed or not
    int ret;
    ret = access(FIFO, F_OK);
    if(ret == 0)    //file /tmp/my_fifo existed
    {
        system("rm -rf /tmp/my_fifo");
    }

    //2. creat a fifo file
    if(-1 == mkfifo(FIFO, 0766))
    {
        perror("mkfifo");
        exit(EXIT_FAILURE);
    }

    //3.Open the fifo file
    int fifo_fd;
    fifo_fd = open(FIFO, O_WRONLY);
    if(-1 == fifo_fd)
    {
        perror("open");
        exit(EXIT_FAILURE);

    }

    //4. write the fifo file
    int num = 0;
    num = write(fifo_fd, buf, sizeof(buf));
    if(num < sizeof(buf))
    {
        perror("write");
        exit(EXIT_FAILURE);
    }

    printf("write the message ok!\n");
    close(fifo_fd);

    return 0;
}
```

fifo_rev.c

```
/*
 *File: fifo_rcv.c
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <fcntl.h>


#define FIFO "/tmp/my_fifo"

int main()
{
    char buf[20] ;
    memset(buf, '\0', sizeof(buf));

    //`. check the fifo file existed or not
    int ret;
    ret = access(FIFO, F_OK);
    if(ret != 0)    //file /tmp/my_fifo existed
    {
        fprintf(stderr, "FIFO %s does not existed", FIFO);
        exit(EXIT_FAILURE);
    }

    //2.Open the fifo file
    int fifo_fd;
    fifo_fd = open(FIFO, O_RDONLY);
    if(-1 == fifo_fd)
    {
        perror("open");
        exit(EXIT_FAILURE);

    }

    //4. read the fifo file
    int num = 0;
    num = read(fifo_fd, buf, sizeof(buf));

    printf("Read %d words: %s\n", num, buf);
    close(fifo_fd);

    return 0;
}
```



```
[xaviershank@xaviershank 实验三]$ gcc -o 3-4-fifo_sen.out 3-4-fifo_sen.c
[xaviershank@xaviershank 实验三]$ ./3-4-fifo_.out
3-4-fifo_rev.out  3-4-fifo_sen.out
[xaviershank@xaviershank 实验三]$ ./3-4-fifo_sen.out
write the message ok!
[xaviershank@xaviershank 实验三]$ 
```



```
[xaviershank@xaviershank 实验三]$ gcc -o 3-4-fifo_rev.out 3-4-fifo_rev.c
[xaviershank@xaviershank 实验三]$ ./3-4-fifo_rev.out
Read 12 words: hello,world
[xaviershank@xaviershank 实验三]$ 
```

## 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？

1. 源代码（实验验证）
2. client.c

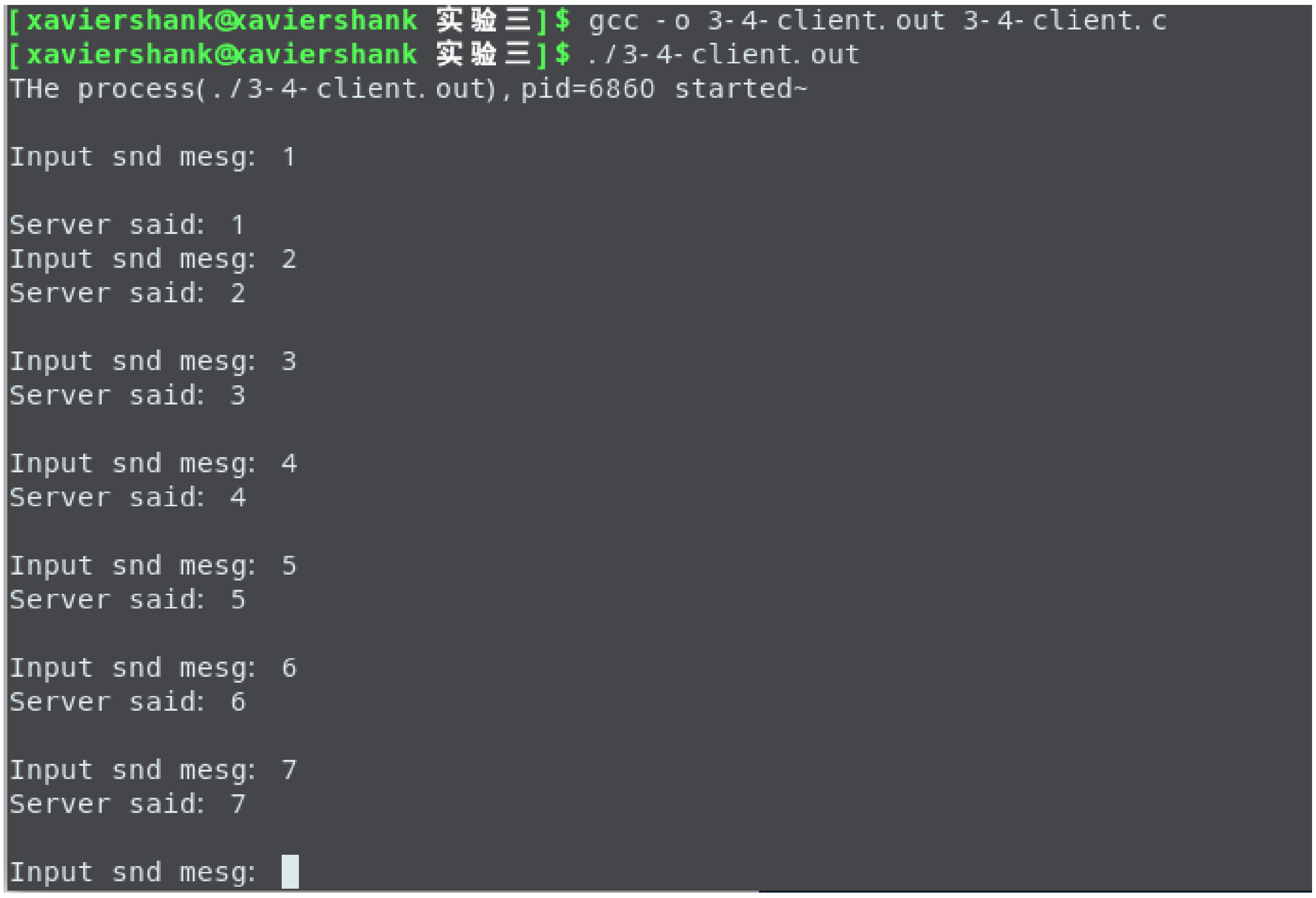
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include <signal.h>

#define BUF_SIZE 128

//Rebuild the strcut (must be)
struct msgbuf
{
    long mtype;
    char mtext[BUF_SIZE];
};


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

```
{
    //1. creat a mseg queue
    key_t key;
    int msgId;

    printf("THe process(%s),pid=%d started~\n", argv[0], getpid());

    key = ftok(".", 0xFF);
    msgId = msgget(key, IPC_CREAT|0644);
    if(-1 == msgId)
    {
        perror("msgget");
        exit(EXIT_FAILURE);
    }

    //2. creat a sub process, wait the server message
    pid_t pid;
    if(-1 == (pid = fork()))
    {
        perror("vfork");
        exit(EXIT_FAILURE);
    }

    //In child process
    if(0 == pid)
    {
        while(1)
        {
            alarm(0);
            alarm(100);     //if doesn't receive messge in 100s, timeout & exit
            struct msgbuf rcvBuf;
            memset(&rcvBuf, '\0', sizeof(struct msgbuf));
            msgrcv(msgId, &rcvBuf, BUF_SIZE, 2, 0);
            printf("Server said: %s\n", rcvBuf.mtext);
        }

        exit(EXIT_SUCCESS);
    }

    else    //parent process
    {
        while(1)
        {
            usleep(100);
            struct msgbuf sndBuf;
            memset(&sndBuf, '\0', sizeof(sndBuf));
            char buf[BUF_SIZE] ;
            memset(buf, '\0', sizeof(buf));

            printf("\nInput snd mesg: ");
            scanf("%s", buf);

            strncpy(sndBuf.mtext, buf, strlen(buf)+1);
            sndBuf.mtype = 1;

            if(-1 == msgsnd(msgId, &sndBuf, strlen(buf)+1, 0))
            {
                perror("msgsnd");
                exit(EXIT_FAILURE);
            }

            //if scanf "end~", exit
            if(!strcmp("end~", buf))
                break;
        }

        printf("THe process(%s),pid=%d exit~\n", argv[0], getpid());
    }

    return 0;
}
```

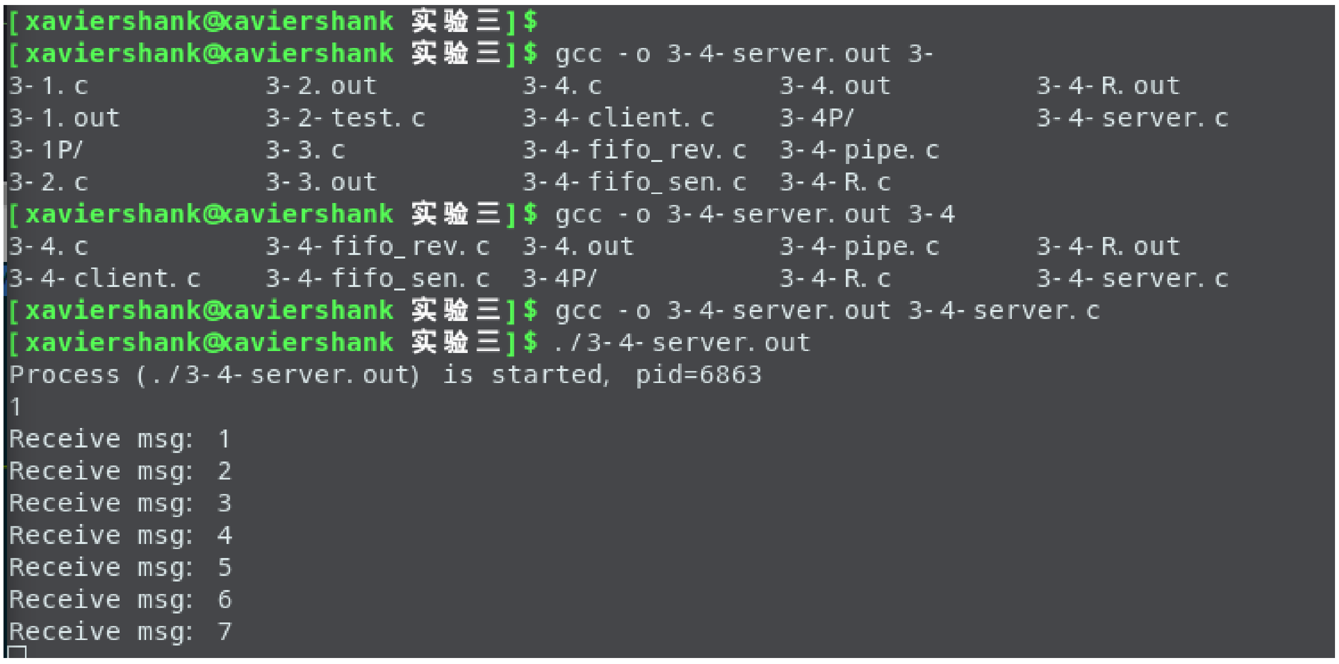
2. server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include <signal.h>

#define BUF_SIZE 128

//Rebuild the strcut (must be)
struct msgbuf
{
    long mtype;
    char mtext[BUF_SIZE];
};


int main(int argc, char *argv[])
{
    //1. creat a mseg queue
    key_t key;
    int msgId;

    key = ftok(".", 0xFF);
    msgId = msgget(key, IPC_CREAT|0644);
    if(-1 == msgId)
    {
        perror("msgget");
        exit(EXIT_FAILURE);
    }
    printf("Process (%s) is started, pid=%d\n", argv[0], getpid());

    while(1)
    {
        alarm(0);
        alarm(600);     //if doesn't receive messge in 600s, timeout & exit
        struct msgbuf rcvBuf;
        memset(&rcvBuf, '\0', sizeof(struct msgbuf));
        msgrcv(msgId, &rcvBuf, BUF_SIZE, 1, 0);
        printf("Receive msg: %s\n", rcvBuf.mtext);

        struct msgbuf sndBuf;
        memset(&sndBuf, '\0', sizeof(sndBuf));

        strncpy((sndBuf.mtext), (rcvBuf.mtext), strlen(rcvBuf.mtext)+1);
        sndBuf.mtype = 2;

        if(-1 == msgsnd(msgId, &sndBuf, strlen(rcvBuf.mtext)+1, 0))
        {
            perror("msgsnd");
            exit(EXIT_FAILURE);
        }

        //if scanf "end~", exit
        if(!strcmp("end~", rcvBuf.mtext))
             break;
    }

    printf("THe process(%s),pid=%d exit~\n", argv[0], getpid());

    return 0;
}
```

2.运行结果



```
[xaviershank@xaviershank 实验三]$ gcc -o 3-4-client.out 3-4-client.c
[xaviershank@xaviershank 实验三]$ ./3-4-client.out
THe process(./3-4-client.out),pid=6860 started~

Input snd mesg: 1

Server said: 1
Input snd mesg: 2
Server said: 2

Input snd mesg: 3
Server said: 3

Input snd mesg: 4
Server said: 4

Input snd mesg: 5
Server said: 5

Input snd mesg: 6
Server said: 6

Input snd mesg: 7
Server said: 7

Input snd mesg: 
```

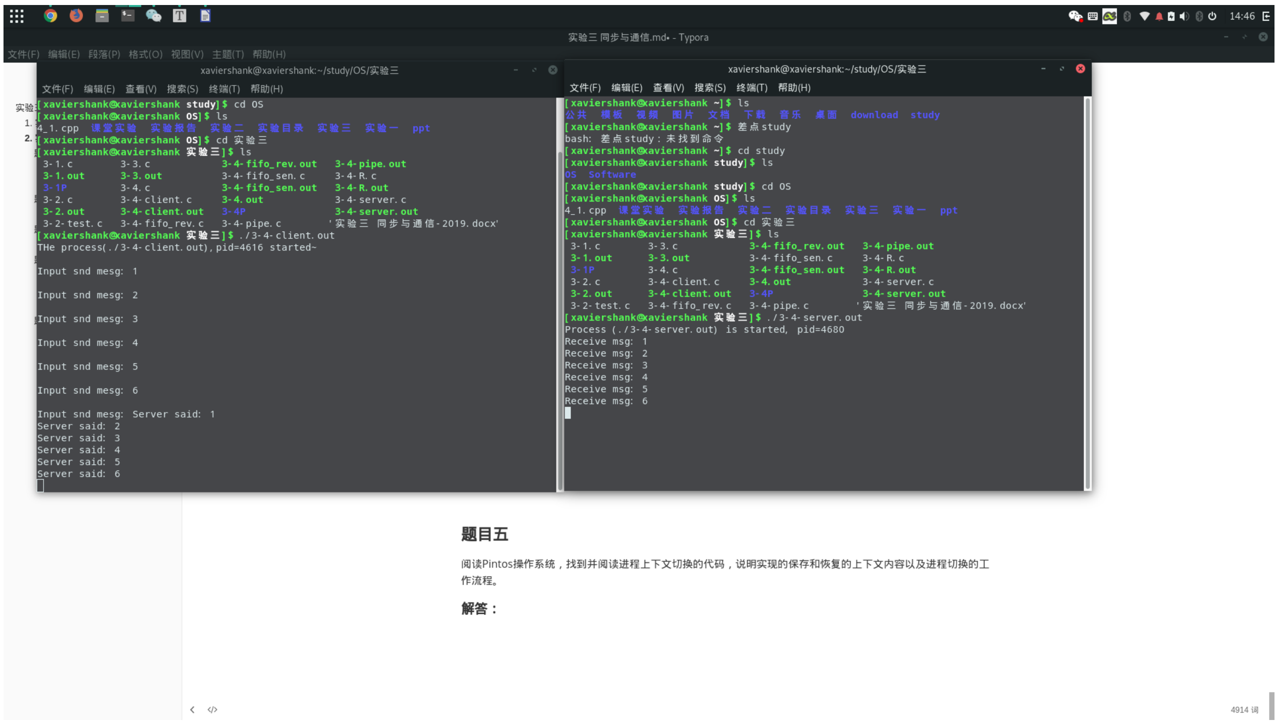


```
[xaviershank@xaviershank 实验三]$
[xaviershank@xaviershank 实验三]$ gcc -o 3-4-server.out 3-
3-1.c           3-2.out         3-4.c           3-4.out         3-4-R.out
3-1.out         3-2-test.c      3-4-client.c    3-4P/           3-4-server.c
3-1P/           3-3.c           3-4-fifo_rev.c  3-4-pipe.c
3-2.c           3-3.out         3-4-fifo_sen.c  3-4-R.c
[xaviershank@xaviershank 实验三]$ gcc -o 3-4-server.out 3-4
3-4.c           3-4-fifo_rev.c  3-4.out         3-4-pipe.c      3-4-R.out
3-4-client.c    3-4-fifo_sen.c  3-4P/           3-4-R.c         3-4-server.c
[xaviershank@xaviershank 实验三]$ gcc -o 3-4-server.out 3-4-server.c
[xaviershank@xaviershank 实验三]$ ./3-4-server.out
Process (./3-4-server.out) is started, pid=6863
1
Receive msg: 1
Receive msg: 2
Receive msg: 3
Receive msg: 4
Receive msg: 5
Receive msg: 6
Receive msg: 7
```

3. 验证阻塞？

在此机制中，发送端传送的消息都会加入一个消息队列。写进程在此机制中不会被阻塞，其写入的字符串会一直被添加至队列的末端，而读进程会从队列的首端一直读取消息，消息节点一旦被读取便会移除队列。当队列中不含其需要类型的消息时便会阻塞。

为验证我们的阻塞规则，可以先只开启客户端进行多条信息传输，再开启服务端观察结果。



```
[xaviershank@xaviershank study]$ cd OS
[xaviershank@xaviershank OS]$ ls
4_1.cpp  课堂实验  实验报告  实验二  实验目录  实验三  实验一  ppt
[xaviershank@xaviershank OS]$ cd 实验三
[xaviershank@xaviershank 实验三]$ ls
 3-1.c       3-3.c            3-4-fifo_rev.out  3-4-pipe.out
 3-1.out     3-3.out          3-4-fifo_sen.c    3-4-R.c
 3-1P        3-4.c            3-4-fifo_sen.out  3-4-R.out
 3-2.c       3-4-client.c     3-4.out           3-4-server.c
 3-2.out     3-4-client.out   3-4P              3-4-server.out
 3-2-test.c  3-4-fifo_rev.c   3-4-pipe.c       '实验三 同步与通信-2019.docx'
[xaviershank@xaviershank 实验三]$ ./3-4-client.out
THe process(./3-4-client.out),pid=4616 started~

Input snd mesg: 1

Input snd mesg: 2

Input snd mesg: 3

Input snd mesg: 4

Input snd mesg: 5

Input snd mesg: 6

Input snd mesg: Server said: 1
Server said: 2
Server said: 3
Server said: 4
Server said: 5
Server said: 6
```

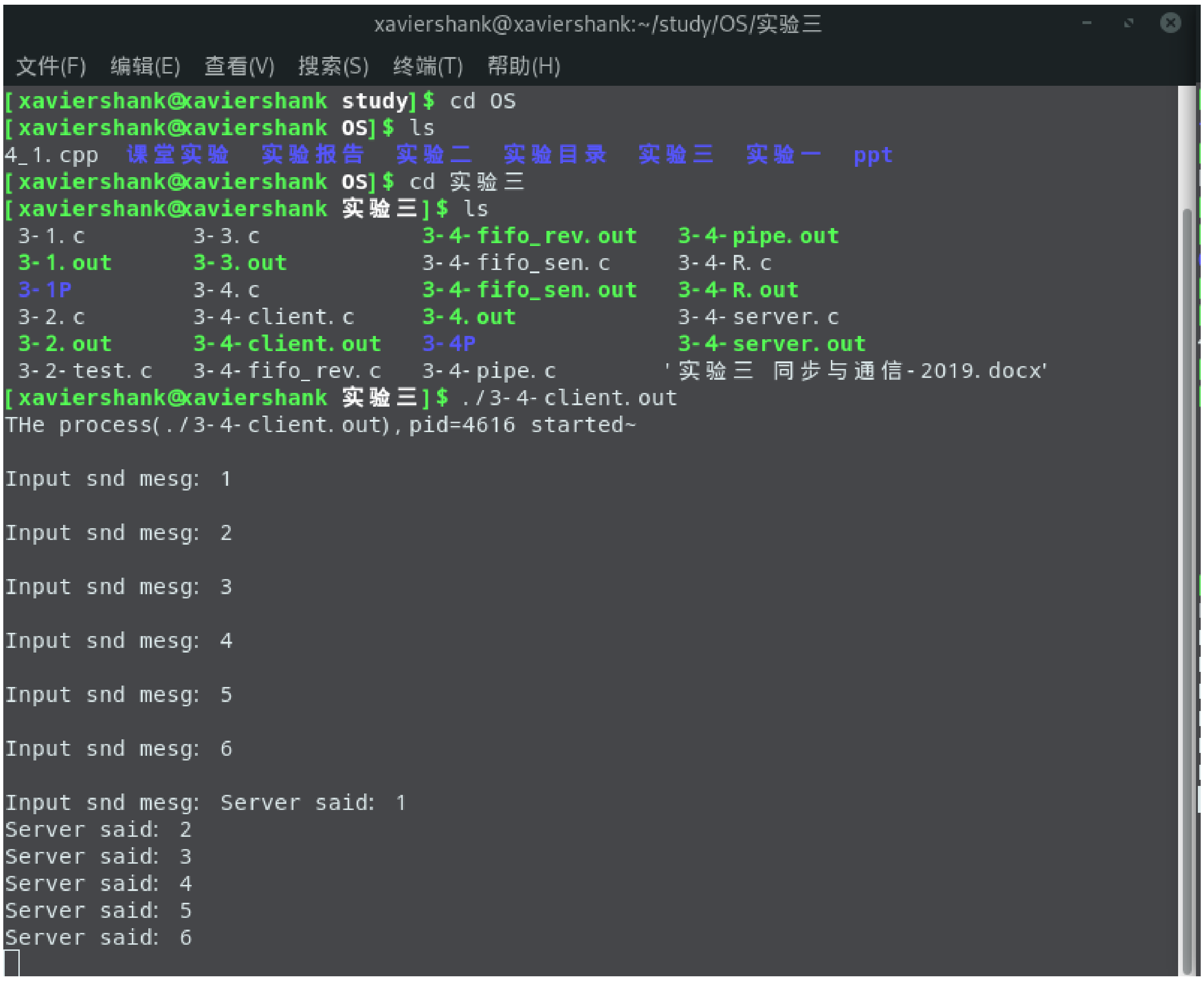
```
[xaviershank@xaviershank ~]$ ls
公共  模板  视频  图片  文档  下载  音乐  桌面  download  study
[xaviershank@xaviershank ~]$ 差点study
bash: 差点study：未找到命令
[xaviershank@xaviershank ~]$ cd study
[xaviershank@xaviershank study]$ ls
OS  Software
[xaviershank@xaviershank study]$ cd OS
[xaviershank@xaviershank OS]$ ls
4_1.cpp  课堂实验  实验报告  实验二  实验目录  实验三  实验一  ppt
[xaviershank@xaviershank OS]$ cd 实验三
[xaviershank@xaviershank 实验三]$ ls
 3-1.c       3-3.c            3-4-fifo_rev.out  3-4-pipe.out
 3-1.out     3-3.out          3-4-fifo_sen.c    3-4-R.c
 3-1P        3-4.c            3-4-fifo_sen.out  3-4-R.out
 3-2.c       3-4-client.c     3-4.out           3-4-server.c
 3-2.out     3-4-client.out   3-4P              3-4-server.out
 3-2-test.c  3-4-fifo_rev.c   3-4-pipe.c       '实验三 同步与通信-2019.docx'
[xaviershank@xaviershank 实验三]$ ./3-4-server.out
Process (./3-4-server.out) is started, pid=4680
Receive msg: 1
Receive msg: 2
Receive msg: 3
Receive msg: 4
Receive msg: 5
Receive msg: 6
```

**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。

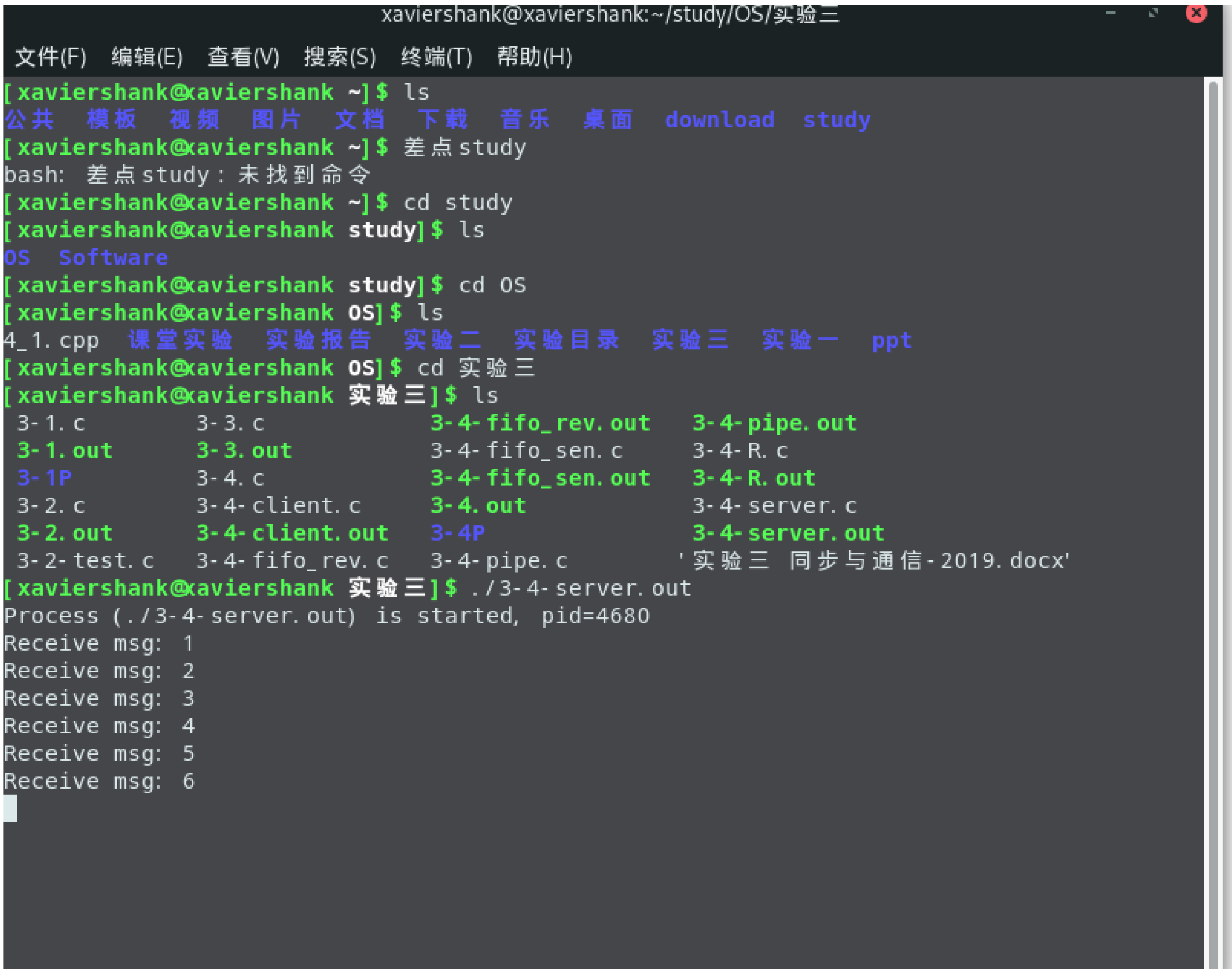
**解答：**

实验三 同步与通信.md• - Typora
文件(F) 编辑(E) 段落(P) 格式(O) 视图(V) 主题(T) 帮助(H)
xaviershank@xaviershank:~/study/OS/实验三
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
14:46
4914 词



```
[xaviershank@xaviershank study]$ cd OS
[xaviershank@xaviershank OS]$ ls
4_1.cpp  课堂实验  实验报告  实验二  实验目录  实验三  实验一  ppt
[xaviershank@xaviershank OS]$ cd 实验三
[xaviershank@xaviershank 实验三]$ ls
 3-1.c       3-3.c            3-4-fifo_rev.out  3-4-pipe.out
 3-1.out     3-3.out          3-4-fifo_sen.c    3-4-R.c
 3-1P        3-4.c            3-4-fifo_sen.out  3-4-R.out
 3-2.c       3-4-client.c     3-4.out           3-4-server.c
 3-2.out     3-4-client.out   3-4P              3-4-server.out
 3-2-test.c  3-4-fifo_rev.c   3-4-pipe.c       '实验三 同步与通信-2019.docx'
[xaviershank@xaviershank 实验三]$ ./3-4-client.out
THe process(./3-4-client.out),pid=4616 started~

Input snd mesg: 1

Input snd mesg: 2

Input snd mesg: 3

Input snd mesg: 4

Input snd mesg: 5

Input snd mesg: 6

Input snd mesg: Server said: 1
Server said: 2
Server said: 3
Server said: 4
Server said: 5
Server said: 6
```

xaviershank@xaviershank:~/study/OS/实验三
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)


xaviershank@xaviershank:~/study/OS/实验三
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

```
[xaviershank@xaviershank ~]$ ls
公共  模板  视频  图片  文档  下载  音乐  桌面  download  study
[xaviershank@xaviershank ~]$ 差点study
bash: 差点study: 未找到命令
[xaviershank@xaviershank ~]$ cd study
[xaviershank@xaviershank study]$ ls
OS  Software
[xaviershank@xaviershank study]$ cd OS
[xaviershank@xaviershank OS]$ ls
4_1.cpp  课堂实验  实验报告  实验二  实验目录  实验三  实验一  ppt
[xaviershank@xaviershank OS]$ cd 实验三
[xaviershank@xaviershank 实验三]$ ls
 3-1.c        3-3.c            3-4-fifo_rev.out   3-4-pipe.out
 3-1.out      3-3.out          3-4-fifo_sen.c     3-4-R.c
 3-1P         3-4.c            3-4-fifo_sen.out   3-4-R.out
 3-2.c        3-4-client.c     3-4.out            3-4-server.c
 3-2.out      3-4-client.out   3-4P               3-4-server.out
 3-2-test.c   3-4-fifo_rev.c   3-4-pipe.c        '实验三 同步与通信-2019.docx'
[xaviershank@xaviershank 实验三]$ ./3-4-server.out
Process (./3-4-server.out) is started, pid=4680
Receive msg: 1
Receive msg: 2
Receive msg: 3
Receive msg: 4
Receive msg: 5
Receive msg: 6
```

## 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：

**thread.h**

在thread.h中寻找关于进程的相关代码：

```
#ifndef THREADS_THREAD_H
#define THREADS_THREAD_H
#include "threads/synch.h"
#include <fixedpoint.h>
#include <debug.h>
#include <list.h>
#include <stdint.h>
/* States in a thread's life cycle. */
enum thread_status
{
THREAD_RUNNING, /* Running thread. */
THREAD_READY, /* Not running but ready to run. */
THREAD_BLOCKED, /* Waiting for an event to trigger. */
THREAD_DYING /* About to be destroyed. */
};
/* Thread identifier type.
You can redefine this to whatever type you like. */
typedef int tid_t;
#define TID_ERROR ((tid_t) -1) /* Error value for tid_t. */
/* Thread priorities. */
#define PRI_MIN 0 /* Lowest priority. */
#define PRI_DEFAULT 31 /* Default priority. */
#define PRI_MAX 63 /* Highest priority. */
/* A kernel thread or user process.
Each thread structure is stored in its own 4 kB page. The
thread structure itself sits at the very bottom of the page
(at offset 0). The rest of the page is reserved for the
thread's kernel stack, which grows downward from the top of
the page (at offset 4 kB). Here's an illustration:
4 kB +---------------------------------+
| kernel stack |
| | |
| | |
| V |
| grows downward |
| |
| |
| |
| |
| |
| |
| |
| |
+---------------------------------+
| magic |
| : |
| : |
| name |
| status |
0 kB +---------------------------------+
The upshot of this is twofold:
1. First, `struct thread' must not be allowed to grow too
big. If it does, then there will not be enough room for
the kernel stack. Our base `struct thread' is only a
few bytes in size. It probably should stay well under 1
kB.
2. Second, kernel stacks must not be allowed to grow too
large. If a stack overflows, it will corrupt the thread
state. Thus, kernel functions should not allocate large
structures or arrays as non-static local variables. Use
dynamic allocation with malloc() or palloc_get_page()
instead.
The first symptom of either of these problems will probably be
an assertion failure in thread_current(), which checks that
the `magic' member of the running thread's `struct thread' is
```

```
set to THREAD_MAGIC. Stack overflow will normally change this
value, triggering the assertion. */
/* The `elem' member has a dual purpose. It can be an element in
the run queue (thread.c), or it can be an element in a
semaphore wait list (synch.c). It can be used these two ways
only because they are mutually exclusive: only a thread in the
ready state is on the run queue, whereas only a thread in the
blocked state is on a semaphore wait list. */
struct thread
{
/* Owned by thread.c. */
tid_t tid; /* Thread identifier. */
enum thread_status status; /* Thread state. */
char name[16]; /* Name (for debugging purposes). */
uint8_t *stack; /* Saved stack pointer. */
int priority; /* Priority. */
int64_t wakeup_time; /* Thread wakeup time in ticks. */
struct semaphore timer_sema;
struct list_elem timer_elem; /* List element for timer_wait_list. */
/* Thread statistics. */
int niceness;
fp_t recent_cpu;
struct list_elem allelem; /* List element for all threads list. */
/* Shared between thread.c and synch.c. */
struct list_elem elem; /* List element. */
#ifdef USERPROG
/* Owned by userprog/process.c. */
uint32_t *pagedir; /* Page directory. */
#endif
/* Owned by thread.c. */
unsigned magic; /* Detects stack overflow. */
};
/* If false (default), use round-robin scheduler.
If true, use multi-level feedback queue scheduler.
Controlled by kernel command-line option "-o mlfqs". */
extern bool thread_mlfqs;
void thread_init (void);
void thread_start (void);
void thread_tick (void);
void thread_print_stats (void);
typedef void thread_func (void *aux);
tid_t thread_create (const char *name, int priority, thread_func *, void *);
void thread_block (void);
void thread_unblock (struct thread *);
struct thread *thread_current (void);
tid_t thread_tid (void);
const char *thread_name (void);
void thread_exit (void) NO_RETURN;
void thread_yield (void);
/* Performs some operation on thread t, given auxiliary data AUX. */
typedef void thread_action_func (struct thread *t, void *aux);
void thread_foreach (thread_action_func *, void *);
void thread_foreach_ready (thread_action_func *, void *);
```

```
int thread_get_priority (void);
void thread_set_priority (int);
int thread_get_nice (void);
void thread_set_nice (int);
int thread_get_recent_cpu (void);
int thread_get_load_avg (void);
/* Compare two threads by their wakeup_time. If wakeup_time
same, compare thread priorities to break the tie.
If true, first thread has earlier wakeup_time and in case of
a tie, higher priority. */
bool less_wakeup (const struct list_elem *left,
const struct list_elem *right, void *aux UNUSED);
/* Comparison function that prefers the threas with higher priority. */
bool more_prio (const struct list_elem *left,
const struct list_elem *right, void *aux UNUSED);
#endif /* threads/thread.h */
```

(1) Pintos中定义了一个thread的结构体用于存储线程的信息（包括优先级和状态），就在以上的代码thread.h中。

四个状态：

```
enum thread_status
{
THREAD_RUNNING, /* Running thread. */
THREAD_READY, /* Not running but ready to run. */
THREAD_BLOCKED, /* Waiting for an event to trigger. */
THREAD_DYING /* About to be destroyed. */
};
/* Thread identifier type.
You can redefine this to whatever type you like. */
```

(2) thread结构体

```
struct thread
{
/* Owned by thread.c. */
tid_t tid; /* Thread identifier. */
enum thread_status status; /* Thread state. */
char name[16]; /* Name (for debugging purposes). */
uint8_t *stack; /* Saved stack pointer. */
int priority; /* Priority. */
int64_t wakeup_time; /* Thread wakeup time in ticks. */
struct semaphore timer_sema;
struct list_elem timer_elem; /* List element for timer_wait_list. */
/* Thread statistics. */
int niceness;
fp_t recent_cpu;
struct list_elem allelem; /* List element for all threads list. */
/* Shared between thread.c and synch.c. */
struct list_elem elem; /* List element. */
#ifdef USERPROG
/* Owned by userprog/process.c. */
uint32_t *pagedir; /* Page directory. */
#endif
/* Owned by thread.c. */
unsigned magic; /* Detects stack overflow. */
}
```

**thread.c**

在这个文件中寻找thread的基本操作（函数）：

1. thread_block()用于将current_thread 终止变成blocked状态，不能跑，只有在调用
2. thread_unblock()后进入就绪队列。
3. thread_yield()用于直接把current_thread进入就绪队列，在任意时刻可再次被调用。 list_entry返回一个线程。
4. idle_thread平常是不在ready_list中的。

   1.在thread_start()即系统刚开始时，idle_thread在ready_list中。 2.在ready_list为空时，调用idle_thread。 idle() 在thread_start调用 { 1.把初始化为0的信号量设为1，即将idle_thread放入就绪队列。 2.把idle_thread的状态设为blocked。

}

5. running_thread()返回一个running thread
6. thread_current()是加上一个check的running_thread
7. init_thread() { 1.初始化thread 2.状态设为blocked 3.把栈的空间变小 4.magic值 5.放入all_list }
8. next_thread_to_run() { 是否就绪队列为空？返回idle_thread：返回就绪队列的第一个，并移除 }
9. thread_schedule_tail(prev) { current_thread的thread_tick清0 激活process 如果prev已经dying，则销毁。 }
10. **schedule()** { 调用这个的函数是thread_block,thread_exit,thread_yield.因为前提是current_thread状态不能是running。

}

**进程切换的核心函数**

其中最重要的是schedule( )这个函数，它专门负责线程切换，执行了以后会把当前线程放进队列里并调度下一个线程。

**schedule( )详解**

```
/* Schedules a new process.  At entry, interrupts must be off and
   the running process's state must have been changed from
   running to some other state.  This function finds another
   thread to run and switches to it.

   It's not safe to call printf() until thread_schedule_tail()
   has completed. */
static void
schedule (void)
{
  struct thread *cur = running_thread ();
  struct thread *next = next_thread_to_run ();
  struct thread *prev = NULL;

  ASSERT (intr_get_level () == INTR_OFF);
  ASSERT (cur->status != THREAD_RUNNING);
  ASSERT (is_thread (next));

  if (cur != next)
    prev = switch_threads (cur, next);
  thread_schedule_tail (prev);
}
```

1. 发现有三个thread结构体的指针，cur指向正在运行的线程的返回值，next指向下一个要运行的线程这个函数的返回值。
2. 再看schedule ();的后半部分，调用switch_threads (cur, next)将当前进程和下一个进程进行切换。此函数用汇编编写，存放在siwth.S中。这是将当前堆栈的指针保存到cur线程的堆栈，**接着从next线程的堆栈中恢复当前堆栈的指针，也就是寄存器esp的操作。由此我们可以确定进程的保存与恢复就是利用CPU栈顶指针的变化进行的，进程的状态则是保存在自身的堆栈当中。**

   ```
   #include "threads/switch.h"

   #### struct thread *switch_threads (struct thread *cur, struct thread *next);
   ####
   #### Switches from CUR, which must be the running thread, to NEXT,
   #### which must also be running switch_threads(), returning CUR in
   #### NEXT's context.
   ####
   #### This function works by assuming that the thread we're switching
   #### into is also running switch_threads().  Thus, all it has to do is
   #### preserve a few registers on the stack, then switch stacks and
   #### restore the registers.  As part of switching stacks we record the
   #### current stack pointer in CUR's thread structure.

   .globl switch_threads
   .func switch_threads
   switch_threads:
           # Save caller's register state.
           #
           # Note that the SVR4 ABI allows us to destroy %eax, %ecx, %edx,
           # but requires us to preserve %ebx, %ebp, %esi, %edi.  See
           # [SysV-ABI-386] pages 3-11 and 3-12 for details.
           #
           # This stack frame must match the one set up by thread_create()
           # in size.
           pushl %ebx
           pushl %ebp
           pushl %esi
           pushl %edi

           # Get offsetof (struct thread, stack).
   .globl thread_stack_ofs
   ```

```
        mov thread_stack_ofs, %edx

        # Save current stack pointer to old thread's stack, if any.
        movl SWITCH_CUR(%esp), %eax
        movl %esp, (%eax,%edx,1)

        # Restore stack pointer from new thread's stack.
        movl SWITCH_NEXT(%esp), %ecx
        movl (%ecx,%edx,1), %esp

        # Restore caller's register state.
        popl %edi
        popl %esi
        popl %ebp
        popl %ebx
        ret
.endfunc
```

3. thread_schedule_tail (prev) 这个函数通过激活新线程的页表来完成线程切换。

```
void
thread_schedule_tail (struct thread *prev)
{
  struct thread *cur = running_thread ();

  ASSERT (intr_get_level () == INTR_OFF);

  /* Mark us as running. */
  cur->status = THREAD_RUNNING;

  /* Start new time slice. */
  thread_ticks = 0;

#ifdef USERPROG
  /* Activate the new address space. */
  process_activate ();
#endif

  /* If the thread we switched from is dying, destroy its struct
     thread.  This must happen late so that thread_exit() doesn't
     pull out the rug under itself.  (We don't free
     initial_thread because its memory was not obtained via
     palloc().) */
  if (prev != NULL && prev->status == THREAD_DYING && prev != initial_thread)
    {
      ASSERT (prev != cur);
      palloc_free_page (prev);
    }
}
```

## 参考文献

作者：xiazdong 来源：CSDN 原文：https://blog.csdn.net/xiazdong/article/details/6327840