

实验三 实验报告

孙汉武 安全1601 16281047

实验三 实验报告

Task 1

- 1.1 实验要求
- 1.2 实验过程
- 1.3 实验结果
- 1.4 现象解释

Task 2

- 2.1 实验要求
- 2.2 实验过程
 - 2.2.1 未添加同步机制
 - 2.2.2 添加同步机制
- 2.3 实验结果
- 2.4 现象解释
 - 2.4.1 现象解释1
 - 2.4.2 现象解释2

Task 3

- 3.1 实验要求
- 3.2 实验过程
- 3.3 实验结果
 - 3.3.1 实验运行现象
 - 3.3.2 实验现象解释

Task 4

- 4.1 实验要求
- 4.2 实验过程
 - 4.2.1 内存共享
 - 4.2.2 管道通信
 - (1) 无名管道
 - (2) 有名管道
 - 4.2.3 消息队列

Task 5

Task 1

1.1 实验要求

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。

1.2 实验过程

1. 实验源码

Task1.c

```
1 #include<stdio.h>
2 #include<unistd.h>
3 #include<stdlib.h>
4 #include<pthread.h>
5 #include<semaphore.h>
6 #include<fcntl.h>
7 int main()
8 {
9     sem_t *P1_signal,*P2_signal,*P3_signal;
10    //主函数中的进程是P1
11    pid_t p2,p3,p4;
12    P1_signal=sem_open("P1_signal",O_CREAT,0666,0);
13    P2_signal=sem_open("P2_signal",O_CREAT,0666,0);
14    P3_signal=sem_open("P3_signal",O_CREAT,0666,0);
15
16    p2=fork(); //创建进程P2
17    if(p2<0)
18    {
19        perror("创建进程p2出错！");
20    }
21    if(p2==0)
22    {
23        sem_wait(P1_signal);
24        printf("I am the process P2!\n");
25        sem_post(P1_signal);
26        sem_post(P2_signal);
27    }
28    if(p2>0)
29    {
30        p3=fork();
31        if(p3<0)
32        {
33            perror("创建进程p出错！");
34        }
35        if(p3==0)
36        {
37            sem_wait(P1_signal);
38            printf("I am the process P3!\n");
39            sem_post(P1_signal);
```

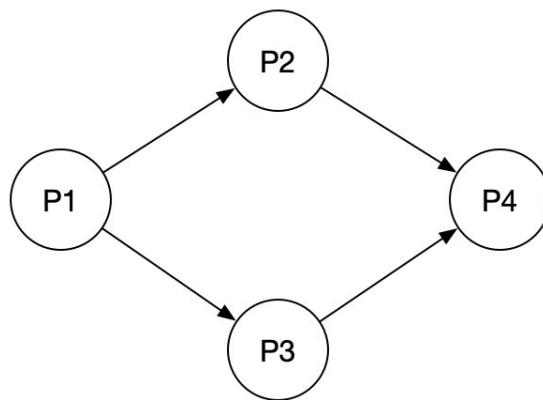
```

40         sem_post(P3_signal);
41     }
42     if(p3>0)
43     {
44         printf("I am the process P1!\n");
45         sem_post(P1_signal);
46         p4=fork();
47         if(p4<0)
48         {
49             perror("创建进程p4出错! ");
50         }
51         if(p4==0)
52         {
53             sem_wait(P2_signal);
54             sem_wait(P3_signal);
55             printf("I am the process P4!\n");
56             sem_post(P2_signal);
57             sem_post(P3_signal);
58         }
59     }
60 }
61 sem_close(P1_signal);
62 sem_close(P3_signal);
63 sem_close(P2_signal);
64 sem_unlink("P1_signal");
65 sem_unlink("P2_signal");
66 sem_unlink("P3_signal");
67 return 0;
68 }

```

2. 原理解释

- 前趋图



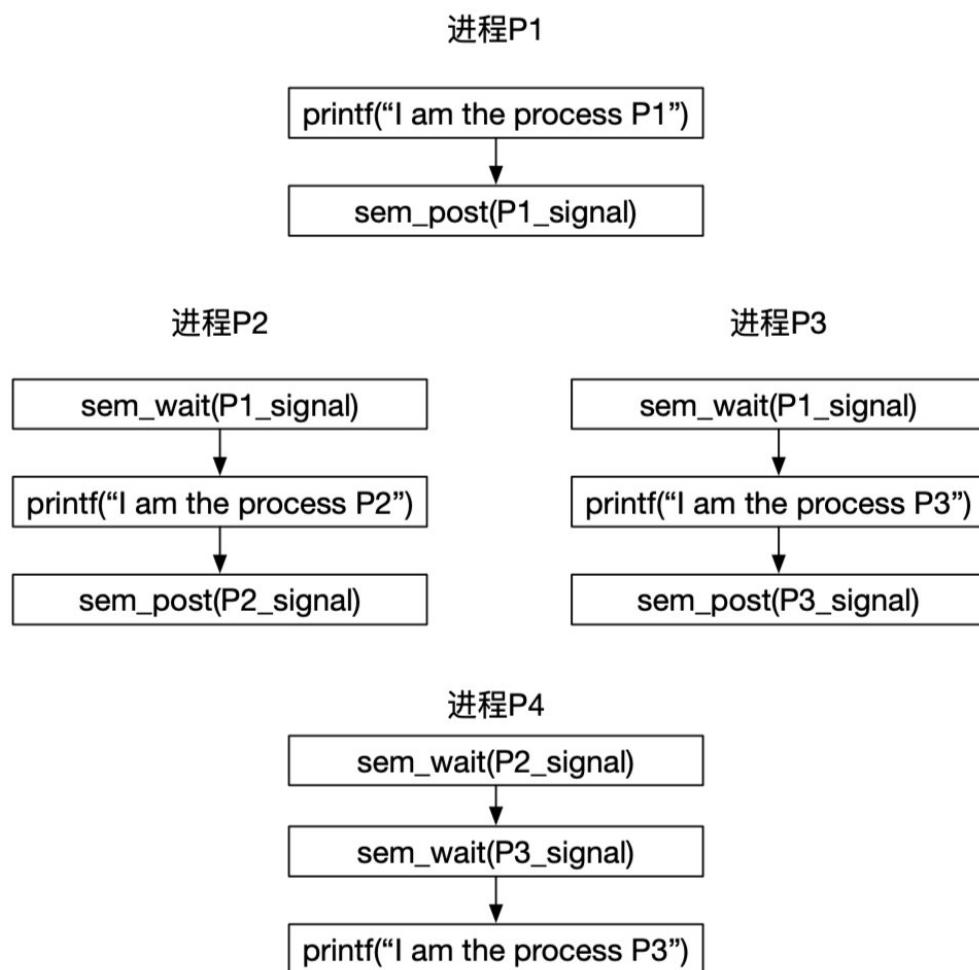
前驱关系： P1-->P2、 P1-->P3、 P2-->P4、 P3-->P4

- 前驱关系实现

题目要求产生的四个进程必须是P1最先执行，P2、P3在P1执行完后互斥执行，P4最后执行。于是根据要求有了上面的前驱关系和前驱图。但是如何实现这种进程间的前驱关系呢？比较自然的想到了是用信号量机制。如上面的代码所示，定义了三个信号量，P1_signal、P2_signal和P3_signal，其初值均为0

```
1 P1_signal=sem_open("P1_signal",O_CREAT,0666,0);  
2 P2_signal=sem_open("P2_signal",O_CREAT,0666,0);  
3 P3_signal=sem_open("P3_signal",O_CREAT,0666,0);
```

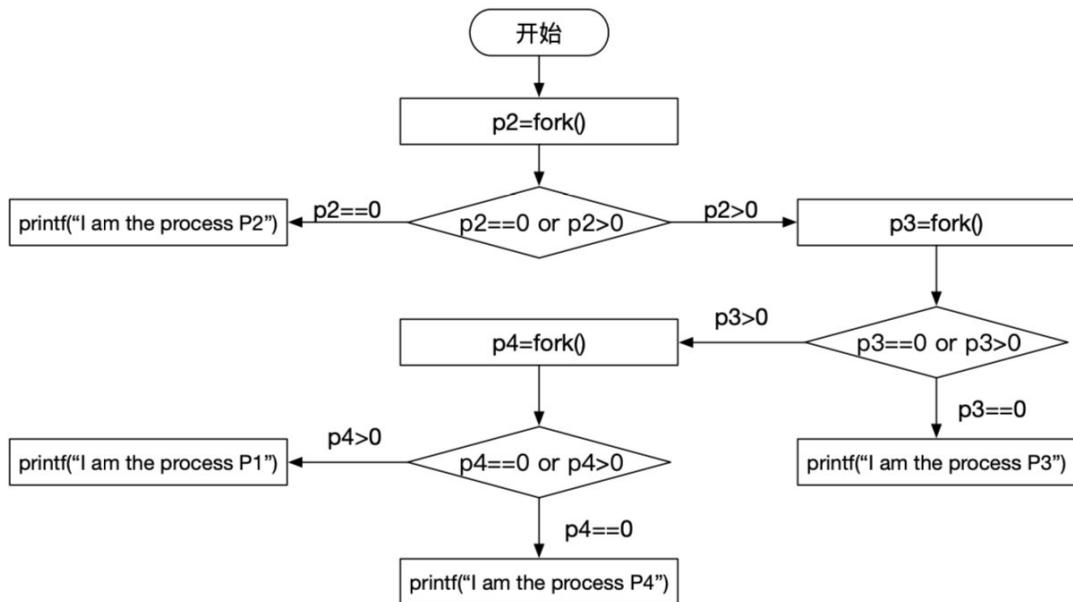
P1进程执行完打印任务之后对P1_signal信号量进行V操作，产生一个资源让等待P1_signal的进程P2和P3其中之一可以执行。由于P2和P3都是等待P1_signal信号量，但是P1进程只产生一个单位的信号，所以P2和P3的执行是互斥的，这样就满足了题目要求。最后在P2和P3执行完打印任务后对信号量P2_signal和P3_signal进行V操作从各产生一个单位的信号量，而进程P4会等待P2_signal和P3_signal，所以知道当P2和P3进程都完成才能进行P4进程。通过控制这三个信号量，这四个进程之间的前驱关系就满足了题目要求。



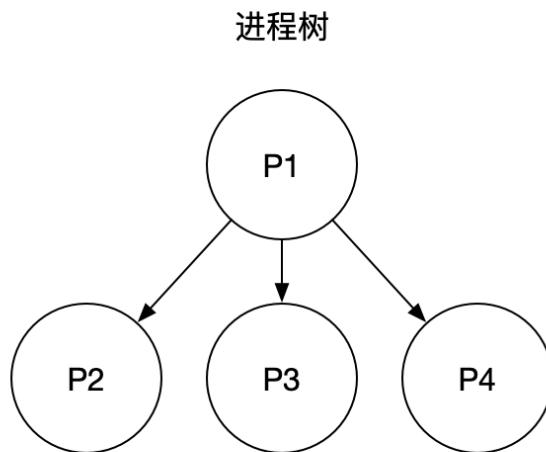
- 进程产生实现

根据题目要求，通过fork的方式产生四个进程。fork函数会从当前位置复制进程，并且在父进程中返回的pid为复制进程的真实pid，在子进程中返回的pid为0。了解这些知识之后可以得到如下的流程图：

下面的流程图仅表示进程间的关系，前驱关系的实现请看上一小节。



- 进程树



3. 编译源码

通过下面的命令编译源码，得到可执行程序

```
1 | gcc -g task1.c -o task1 -lpthread
```

1.3 实验结果

通过上面的实验已经得到满足实验要求的可执行程序task1,下面给出运行结果，经过多次测试，四个进程在屏幕上打印的顺序只有两种结果，分别如下：

1. 顺序1： P1-->P2-->P3-->P4

```
ubuntu@VM-0-13-ubuntu:~/study/OS/16281047_OperatingSystemExperiment/lab3$ master$ ./task1
I am the process P1!
I am the process P2!
I am the process P3!
I am the process P4!
```

2. 顺序2: P1-->P3--P2-->P4

```
ubuntu@VM-0-13-ubuntu:~/study/OS/16281047_OperatingSystemExperiment/lab3$ master$ ./task1
I am the process P1!
I am the process P3!
I am the process P2!
I am the process P4!
```

1.4 现象解释

测试的实验结果中出现两种执行顺序, 通过1.2节中的分析不难解释这种现象, 由于P1是P2和P3的前驱, 所以P1一定会在P2和P3之前执行, 但是P2和P3是互斥关系, 这两个进程谁先获得P1产生的信号量谁就先执行另一个进程等待。最后等P2和P3都执行完了再执行P4, 所以会出现上面的两种执行顺序。

Task 2

2.1 实验要求

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。

2.2 实验过程

2.2.1 未添加同步机制

1. 实验源码

task2_1.c:

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<pthread.h>
4 #include<semaphore.h>
5 #include<sys/stat.h>
6 #include<fcntl.h>
7 #include<string.h>
8 int ticketCount=1000;
9 void *SaleThread(void *arg)
10 {
11     int num,temp;
12     num=atoi(arg);
13     for(int i=0;i<num;i++)
14     {
15         if(i % 10 ==0)
16             printf("卖%d张票,剩余%d张票\n",i,ticketCount);
17         temp=ticketCount;
18         //放弃CPU, 强制切换到另外一个进程
```

```

19     pthread_yield();
20     temp=temp-1;
21     pthread_yield();
22     ticketCount=temp;
23 }
24 return NULL;
25 }

26
27 void *RefundThread(void *arg)
28 {
29     int num,temp;
30     num=atoi(arg);
31     for(int i=0;i<num;i++)
32     {
33         if(i % 10 ==0)
34             printf("退%d张票, 剩余%d张票\n",i,ticketCount);
35         temp=ticketCount;
36         pthread_yield();
37         temp=temp+1;
38         pthread_yield();
39         ticketCount=temp;
40     }
41     return NULL;
42 }
43 int main(int argc,char *argv[])
44 {
45     if(argc!=3)
46     {
47         printf("请正确输入参数! \n");
48         exit(0);
49     }
50     printf("初始票数为: %d\n",ticketCount);
51     pthread_t p1,p2;
52     /* printf("%s %s",argv[1],argv[2]); */
53     pthread_create(&p1,NULL,SaleThread,argv[1]);
54     pthread_create(&p2,NULL,RefundThread,argv[2]);
55     pthread_join(p1,NULL);
56     pthread_join(p2,NULL);
57     printf("最终票数为: %d\n",ticketCount);
58     return 0;
59 }

```

2. 程序解释

- 在main函数中创建两个线程，分别是模拟售票的线程 `SaleThread` 和模拟退票的线程 `RefundThread`，两个进程并发执行，不添加任何的同步机制。
- 模拟票数的变量 `ticketCount` 是全局变量
- 程序运行需要输入两个参数，第一个是售票数量，第二个数退票数量

3. 程序运行结果

编译上述程序，得到可执行程序 task2_1

```
1 | gcc -g task2_1.c -o task2_1 -lpthread
```

多次测试运行，运行结果可以分为两种类型，一种是售票数量比退票数量多，另一种是售票数量比退票数量少。两种情况的结果分别如下：

- 售票数量比退票数量多：

初始票数：1000 售票：100 退票：40

```
master ➤ ./task2_1 100 40
初始票数为：1000
退0张票，剩余1000张票
卖0张票，剩余1000张票
退10张票，剩余1010张票
卖10张票，剩余990张票
退20张票，剩余1020张票
卖20张票，剩余980张票
退30张票，剩余1030张票
卖30张票，剩余970张票
卖40张票，剩余960张票
卖50张票，剩余950张票
卖60张票，剩余940张票
卖70张票，剩余930张票
卖80张票，剩余920张票
卖90张票，剩余910张票
最终票数为：900
```

- 售票数量比退票数量少：

初始票数：1000 售票：50 退票：80

```
master ➤ ./task2_1 50 80
初始票数为：1000
退0张票，剩余1000张票
卖0张票，剩余1000张票
退10张票，剩余1010张票
卖10张票，剩余990张票
退20张票，剩余1020张票
卖20张票，剩余980张票
退30张票，剩余1030张票
卖30张票，剩余970张票
退40张票，剩余1040张票
卖40张票，剩余960张票
退50张票，剩余1050张票
退60张票，剩余1060张票
退70张票，剩余1070张票
最终票数为：1080
```

4. 实验现象归纳

通过一系列的测试，归纳出的实现现象如下：

- 当售票数量大于退票数量的时候，最终票数等于总票数减去售票数
- 当售票数量小于退票数量的时候，最终票数等于总票数加上退票数

2.2.2 添加同步机制

1. 实验源码

task2_2.c

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<pthread.h>
4 #include<semaphore.h>
5 #include<sys/stat.h>
6 #include<fcntl.h>
7 #include<string.h>
8 volatile int ticketCount=1000;
9 sem_t *flag=NULL;
10 void *SaleThread(void *arg)
11 {
12     int num,temp;
13     num=atoi(arg);
14     for(int i=0;i<num;i++)
15     {
16         if(i % 10 ==0)
17             printf("卖%d张票,剩余%d张票\n",i,ticketCount);
18         sem_wait(flag);
19         temp=ticketCount;
20         //放弃CPU, 强制切换到另外一个进程
21         pthread_yield();
22         temp=temp-1;
23         ticketCount-=1;
24         pthread_yield();
25         ticketCount=temp;
26         sem_post(flag);
27     }
28     return NULL;
29 }
30
31 void *RefundThread(void *arg)
32 {
33     int num,temp;
34     num=atoi(arg);
35     for(int i=0;i<num;i++)
36     {
37         if(i % 10 ==0)
38             printf("退%d张票, 剩余%d张票\n",i,ticketCount);
39         sem_wait(flag);
40         temp=ticketCount;
41         pthread_yield();
42         temp=temp+1;
43         ticketCount+=1;
44         pthread_yield();
45         ticketCount=temp;
```

```

46     sem_post(flag);
47 }
48 return NULL;
49 }
50 int main(int argc,char *argv[])
51 {
52     if(argc!=3)
53     {
54         printf("请正确输入参数! \n");
55         exit(0);
56     }
57     flag=sem_open("flag",O_CREAT,0666,1);
58     printf("初始票数为: %d\n",ticketCount);
59     pthread_t p1,p2;
60     printf("%s %s",argv[1],argv[2]);
61     pthread_create(&p1,NULL,SaleThread,argv[1]);
62     pthread_create(&p2,NULL,RefundThread,argv[2]);
63     pthread_join(p1,NULL);
64     pthread_join(p2,NULL);
65     printf("最终票数为: %d\n",ticketCount);
66     sem_close(flag);
67     sem_unlink("flag");
68     return 0;
69
70 }

```

2. 程序解释

- task2_2.c在task2_1.c的基础上增加了同步机制，其他部分完全一致，通过信号量flag的控制，让售票线程和退票线程一次只能执行一个，在一个没有执行完成之前另一个不能进入执行，这样就保证了售票操作和退票操作的原子性，避免了脏数据的读取。
- flag初始值为设置为1，表示每次只允许一个线程操作ticketCount这个数据
- 售票线程和退票线程在进入操作之前都要sem_wait(flag)，等待信号量，在完成操作之后要sem_post(flag)，下图是售票线程中增加了信号量的操作：

```

18     sem_wait(flag);
19     temp=ticketCount;
20     //放弃CPU，强制切换到另外一个进程
21     pthread_yield();
22     temp=temp-1;
23     ticketCount-=1;
24     pthread_yield();
25     ticketCount=temp;
26     sem_post(flag);

```

3. 程序运行结果

编译上述程序，得到可执行程序task2_2

```
1 | gcc task2_2.c -o task2_2 -lpthread
```

多次测试运行，测试主要分为两种类型，一种是售票数量比退票数量多，另一种是售票数量比退票数量少。两种情况的结果分别如下：

- 售票数量比退票数量多：

初始票数：1000 售票：100 退票：40

```
ubuntu@VM-0-13-ubuntu:~/study/OS/16281047_OperatingSystemExperiment/lab  
3(master) > ./task2_2 100 40  
初始票数为：1000  
100 40退0张票，剩余1000张票  
卖0张票，剩余1000张票  
退10张票，剩余1010张票  
退20张票，剩余1020张票  
退30张票，剩余1030张票  
卖10张票，剩余1030张票  
卖20张票，剩余1020张票  
卖30张票，剩余1010张票  
卖40张票，剩余1000张票  
卖50张票，剩余990张票  
卖60张票，剩余980张票  
卖70张票，剩余970张票  
卖80张票，剩余960张票  
卖90张票，剩余950张票  
最终票数为：940
```

- 退票数量比售票数量多：

初始票数：1000 售票：50 退票：80

```
ubuntu@VM-0-13-ubuntu:~/study/OS/16281047_OperatingSystemExperiment/lab  
3(master) > ./task2_2 50 80  
初始票数为：1000  
50 80退0张票，剩余1000张票  
卖0张票，剩余1000张票  
退10张票，剩余1010张票  
退20张票，剩余1020张票  
退30张票，剩余1030张票  
退40张票，剩余1040张票  
退50张票，剩余1050张票  
退60张票，剩余1060张票  
退70张票，剩余1070张票  
卖10张票，剩余1070张票  
卖20张票，剩余1060张票  
卖30张票，剩余1050张票  
卖40张票，剩余1040张票  
最终票数为：1030
```

4. 实验现象归纳

- 在第一个测试样例中，初始票数为1000，售票100并且退票40，最终总票数为940，结果正确；
- 在第二个测试样例中，初始票数为1000，售票50并且退票80，最终总票数为1030，结果正确。
- 通过上面的测试结果可以看出，不论是售票数量比退票数量多还是少，都不会发生类似前面2.2.1的问题，最终的票数是期待得到的结果。
- 上面的实验证实了增加了同步机制之后的多线程并发程序有效的解决了脏数据的读取问题

2.3 实验结果

通过2.2节的对比实验可以看出，在执行多进程并发程序的时候，由于多进程的切换可能发生在某个进程的中间，会导致在一个进程处理的数据未写入ticketCount之前另外一个进程读取该数据，这样就导致了脏数据的读取，导致最终结果的不正确。

在2.2节的后半部分通过怎加同步机制，保证售票进程和退票进程的的原子性，就是指在某个进程操作的时候，在它完成操作之前另外一个进程无法操作共享变量ticketCount,这样就避免了脏数据的发生，得到了预期的正确结果。

2.4 现象解释

2.4.1 现象解释1

在2.2.1节的实验中，以售票线程为例（代码如下图所示），没有添加同步机制，并且在进行`temp=temp-1` 和 `temp=ticketCount` 的后面均加上了`pthread_yield`，这个函数的作用是放弃对CPU的使用权，切换到其他进程中，本实验中就是切换到退票进程中。

这样就能解释为什么2.2.1节中的实验现象，在2.2.1节中，不论售票数多还是退票数多，最终结果都是总票数加上或减去值比较大的那个数。通过分析可以得出解释，售票进程和退票进程同时进行，初始票数均为1000，售票进程完成一次是票数为999，售票进程开始下一次售票，但是在运行`temp=ticketCount`之前，退票进程处理的数据还没有写入到内存中，导致售票进程读取的还是自己之前计算的ticketCount值，而不是全局的值。退票进程也是同理。

但是为什么刚好就是总票数加上或减去值比较大的那个呢？按照道理来说因该售票进程执行`temp=ticketCount`在退票进程写入`ticketCount`值之前发生是存在一定概率的，但是在目前为止的所有测试结果全部都是在写入之前读取`ticketCount`值，对此的解释是由于`ticketCount=temp`和`temp=ticketCount`之间没有加`pthread_yield`操作，而现代的处理器运算速度足够快，在退票进程放弃CPU控制权的那个时间片已经完成了这两步操作，所以相当于售票进程读取的`ticketCount`一直是自己本身的值，退票进程处理的数据对售票进程并没有影响。

```
10 void *SaleThread(void *arg)
11 {
12     int num,temp;
13     num=atoi(arg);
14     for(int i=0;i<num;i++)
15     {
16         if(i % 10 ==0)
17             printf("卖%d张票,剩余%d张票\n",i,ticketCount);
18         /* sem_wait(flag); */
19         temp=ticketCount;
20         //放弃CPU，强制切换到另外一个进程
21         pthread_yield();
22         temp=temp-1;
23         ticketCount-=1;
24         pthread_yield();
25         ticketCount=temp;
26         /* pthread_yield(); */
27         /* sem_post(flag); */
28     }
29     return NULL;
30 }
```

为了验证上面的猜想，在=如下图所示代码，在ticketCount之后增加一行代码，pthread_yield，放弃当前进程对CPU的控制权，即售票进程放弃对CPU的控制权转而交给退票进程，这个时候退票进程处理的数据就能写入到内存中，而当售票进程再次处理temp=ticketCount的时候，读取的就是退票进程已经写入的数据。如果猜想正确的话，期待的最终票数因该还会发生错误，但并不是像第一中那种恰好等于总票数加减数值大的那个数。

```
10 void *SaleThread(void *arg)
11 {
12     int num,temp;
13     num=atoi(arg);
14     for(int i=0;i<num;i++)
15     {
16         if(i % 10 ==0)
17             printf("卖%d张票,剩余%d张票\n",i,ticketCount);
18         /* sem_wait(flag); */
19         temp=ticketCount;
20         //放弃CPU，强制切换到另外一个进程
21         pthread_yield();
22         temp=temp-1;
23         ticketCount-=1;
24         pthread_yield();
25         ticketCount=temp;
26         pthread_yield();  
/* sem_post(flag); */
27     }
28     return NULL;
29 }
```

得到的结果如下，发现最终的票数不在是 $1000+50=1050$ ，而是分布在 $1000\sim 1050$ 之间的数值。猜想得到验证。

```
ubuntu@VM-0-13-ubuntu:~/study/OS/16281047_OperatingSystemExperiment/lab  
3 [master] $ ./task2_1 20 50
初始票数为： 1000
退0张票，剩余1000张票
卖0张票，剩余1000张票
卖10张票，剩余990张票
退10张票，剩余988张票
退20张票，剩余988张票
退30张票，剩余998张票
退40张票，剩余1008张票
最终票数为： 1018
```

针对上面的猜想（CPU运算速度过快，导致ticketCount=temp和temp=ticketCount两步操作在一个进程的时间片内完成导致的数据错误），另外的一种验证方式是将初始票数和售票退票数设置的足够大，当数据足够大的时候，就会存在一定概率出现在一个进程的ticketCount=temp和temp=ticketCount两步操作之间切换进程的问题，得到的结果就不会类似2.2.1中的那样，而是类似在ticketCount=temp下面加了pthread_yield那样。

```

8 volatile int ticketCount=100000000;
9 /* sem_t *flag=NULL; */
10 void *SaleThread(void *arg)
11 {
12     int num,temp;
13     num=atoi(arg);
14     for(int i=0;i<num;i++)
15     {
16         if(i % 100000 ==0)
17             printf("卖%d张票,剩余%d张票\n",i,ticketCount);
18         /* sem_wait(flag); */
19         temp=ticketCount;
20         //放弃CPU, 强制切换到另外一个进程
21         pthread_yield();
22         temp=temp-1;
23         ticketCount-=1;
24         pthread_yield();
25         ticketCount=temp;
26         /* pthread_yield(); */
27         /* sem_post(flag); */
28     }
29     return NULL;
30 }

```

再次运行，可以看到如下的实验结果：

```
t/lab3 ➤ master • ➤ ./task2_1 10000000 5000000
初始票数为: 100000000
退0张票, 剩余100000000张票
卖0张票, 剩余100000000张票
退1000000张票, 剩余101000000张票
卖1000000张票, 剩余99000000张票
退2000000张票, 剩余102000000张票
卖2000000张票, 剩余98000000张票
退3000000张票, 剩余103000000张票
卖3000000张票, 剩余97000000张票
退4000000张票, 剩余104000000张票
卖4000000张票, 剩余102452181张票
卖5000000张票, 剩余103016186张票
卖6000000张票, 剩余102016186张票
卖7000000张票, 剩余101016186张票
卖8000000张票, 剩余100016186张票
卖9000000张票, 剩余99016186张票
最终票数为: 98016186
```

2.4.2 现象解释2

在2.2.2节中，当给售票进程和退票进程都加上同步机制后，保证了每个线程操作的原子性，每个线程操作的过程中其他的线程不能对共享的ticketCount变量进程修改，这样的话最终的结果就是正确的结果。

Task 3

3.1 实验要求

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。

3.2 实验过程

1. 实验源码

task3.c

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<pthread.h>
4 #include<semaphore.h>
5 #include<sys/stat.h>
6 #include<fcntl.h>
7 char buf[10];
8 sem_t *empty=NULL;
9 sem_t *full=NULL;
10 void *worker1(void *arg)
11 {
12
13     for(int i=0;i<10;i++)
14     {
15         sem_wait(empty);
16         /* fflush(stdin); */
17         /* printf("输入: "); */
18         scanf("%c",&buf[i]);
19         sem_post(full);
20         if(i==9)
21         {
22             i=-1;
23         }
24     }
25     return NULL;
26 }
27 void *worker2(void *arg)
28 {
29     for(int i=0;i<10;i++)
30     {
31         sem_wait(full);
32         printf("输出: %c\n",buf[i]);
33         sem_post(empty);
34         sleep(1);
35         if(i==9)
36         {
```

```

37     i=-1;
38 }
39 }
40 return NULL;
41 }
42
43 int main(int argc,char *argv[])
44 {
45     empty=sem_open("empty_",O_CREAT,0666,10);
46     full=sem_open("full_",O_CREAT,0666,0);
47     pthread_t p1,p2;
48     pthread_create(&p1,NULL,worker1,NULL);
49     pthread_create(&p2,NULL,worker2,NULL);
50     pthread_join(p1,NULL);
51     pthread_join(p2,NULL);
52     sem_close(empty);
53     sem_close(full);
54     sem_unlink("empty_");
55     sem_unlink("full_");
56     return 0;
57 }
```

2. 程序解释

- work1是输入线程调用的函数， worker2是输出线程调用的函数。
- 设置两个信号量empty和ful来控制程序的执行， 其中empty信号量用于保证输入线程在写入数据到缓存的时候缓存中还有空余的位置， 保证写入线程后写入的数据不会把前面写入但是为输出的数据给覆盖掉， 其初始值为10， 表示最开始缓存中有10个空余的位置供给写入线程写入数据； full信号量是用于保证输出线程有数据输出， 避免在写入线程还没有写入数据的情况下输出线程输出随机数据， 其初始值为0， 表示初始状态下缓存中没有数据可以输出
- 输入线程在写入一个数据前要等待empty信号量， 进入后便消耗一个信号量； 完成写入数据操作之后post一个full信号量， 通知输出线程输出数据。
- 输出线程在输出一个数据之前哟啊等待full信号量， 进出输出操作后便消耗一个full信号量； 完成输出操作后post一个empty信号量， 通知写入线程缓存又多一个空余位置以供写入数据。
- 输出线程每输出一个字符等待一秒钟， 方便实验结果的查看。

3. 编译源代码

```
1 | gcc task3.c -o task3 -lpthread
```

3.3 实验结果

3.3.1 实验运行现象

1. 随机输入字母和数字（10个以内）：124365abc

```
x sun@iZwz9evpg61guy89phiy9yZ ~/study/OS/16281047_OperatingSystemExperiment/lab3 master ./task3  
124365abc  
输出: 1  
输出: 2  
输出: 4  
输出: 3  
输出: 6  
输出: 5  
输出: a  
输出: b  
输出: c  
输出:
```

2. 随机输入字母和数字(10个以上): 123456789abcdefg

```
x sun@iZwz9evpg61guy89phiy9yZ ~/study/OS/16281047_OperatingSystemExperiment/lab3 master ./task3  
123456789abcdefg  
输出: 1  
输出: 2  
输出: 3  
输出: 4  
输出: 5  
输出: 6  
输出: 7  
输出: 8  
输出: 9  
输出: a  
输出: b  
输出: c  
输出: d  
输出: e  
输出: f  
输出: g  
输出:
```

3. 不间断输入:

```
x sun@iZwz9evpg61guy89phiy9yZ ~/study/OS/16281047_OperatingSystemExperiment/lab3 master ./task3  
123  
输出: 1  
输出: 2  
asd输出: 3  
输出:  
输出: a  
654输出: s  
输出: d  
ZXC  
输出:  
输出: 6  
输出: 5  
输出: 4  
输出:  
输出: z  
输出: x  
输出: c  
输出:
```

通过观察上面的实验现象，可以看到已经满足了实验题目的要求。

3.3.2 实验现象解释

- 在第一种类型的测试中，输入数据不大于10个字符的时候，由于empty的信号量初始值为10，所以输入进程会一直连续不断的向缓存中写入数据，每写入一个数据，便post一个full信号量，输出线程便能按序输出字符。
- 在第二种类型的测试中，输入数据大于10个字符的时候，由于empty的初始值为10，所以输入的字符中开始的时候只有前10个字符被写入缓存中，其他的在I/O缓冲区等待输入，当输出线程接收

到输入线程post的信号量的时候便会开始输出，每输出一个字符便会post一个empty信号量，当输入线程接收到empty信号量的时候有开始从I/O缓冲区读取数据写入到缓存中。

- 第三种测试和第二种类似，在输出的过程中间输入数据，原理其实是一样的。

Task 4

4.1 实验要求

1. 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？
2. 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？
3. 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？

4.2 实验过程

实验过程根据实验要求的三个部分，对应的过程也分为三个部分，具体如下所示

4.2.1 内存共享

1. 实验源码

内存共享实验的源码分为两个部分，分别是Sender.c和Receive.c，

Sender.c：

```
1  /*
2  * Filename: Sender.c
3  * Description:
4  */
5
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <unistd.h>
9 #include <sys/sem.h>
10 #include <sys/ipc.h>
11 #include <sys/shm.h>
12 #include <sys/types.h>
13 #include <string.h>
14
15 int main(int argc, char *argv[])
16 {
17     key_t key;
18     int shm_id;
19     int sem_id;
20     int value = 0;
```

```

1 //1. Product the key
2 key = ftok(".", 0xFF);
3 //2. Creat semaphore for visit the shared memory
4 sem_id = semget(key, 1, IPC_CREAT|0644);
5 if(-1 == sem_id)
6 {
7     perror("semget");
8     exit(EXIT_FAILURE);
9 }
10 //3. init the semaphore, sem=0
11 if(-1 == (semctl(sem_id, 0, SETVAL, value)))
12 {
13     perror("semctl");
14     exit(EXIT_FAILURE);
15 }
16 //4. Creat the shared memory(1K bytes)
17 shm_id = shmget(key, 1024, IPC_CREAT|0644);
18 if(-1 == shm_id)
19 {
20     perror("shmget");
21     exit(EXIT_FAILURE);
22 }
23 //5. attach the shm_id to this process
24 char *shm_ptr;
25 shm_ptr = shmat(shm_id, NULL, 0);
26 if(NULL == shm_ptr)
27 {
28     perror("shmat");
29     exit(EXIT_FAILURE);
30 }
31 //6. Operation procedure
32 struct sembuf sem_b;
33 sem_b.sem_num = 0;           //first sem(index=0)
34 sem_b.sem_flg = SEM_UNDO;
35 sem_b.sem_op = 1;           //Increase 1,make sem=1
36
37 while(1)
38 {
39     if(0 == (value = semctl(sem_id, 0, GETVAL)))
40     {
41         printf("\nNow, snd message process running:\n");
42         printf("\tInput the snd message: ");
43         scanf("%s", shm_ptr);
44
45         if(-1 == semop(sem_id, &sem_b, 1))
46         {
47             perror("semop");
48             exit(EXIT_FAILURE);
49         }
50     }
51 }

```

```

70     }
71     //if enter "end", then end the process
72     if(0 == (strcmp(shm_ptr , "end")))
73     {
74         printf("\nExit sender process now!\n");
75         break;
76     }
77 }
78 shmdt(shm_ptr);
79 return 0;
80 }
81

```

Receiver.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/sem.h>
5 #include <sys/ipc.h>
6 #include <sys/shm.h>
7 #include <sys/types.h>
8 #include <string.h>
9
10 int main(int argc, char *argv[])
11 {
12     key_t key;
13     int shm_id;
14     int sem_id;
15     int value = 0;
16     //1.Product the key
17     key = ftok(".", 0xFF);
18     //2. Creat semaphore for visit the shared memory
19     sem_id = semget(key, 1, IPC_CREAT|0644);
20     if(-1 == sem_id)
21     {
22         perror("semget");
23         exit(EXIT_FAILURE);
24     }
25     //3. init the semaphore, sem=0
26     if(-1 == (semctl(sem_id, 0, SETVAL, value)))
27     {
28         perror("semctl");
29         exit(EXIT_FAILURE);
30     }
31     //4. Creat the shared memory(1K bytes)
32     shm_id = shmemget(key, 1024, IPC_CREAT|0644);
33     if(-1 == shm_id)
34     {

```

```

35         perror("shmget");
36         exit(EXIT_FAILURE);
37     }
38     //5. attach the shm_id to this process
39     char *shm_ptr;
40     shm_ptr = shmat(shm_id, NULL, 0);
41     if(NULL == shm_ptr)
42     {
43         perror("shmat");
44         exit(EXIT_FAILURE);
45     }
46
47     //6. Operation procedure
48     struct sembuf sem_b;
49     sem_b.sem_num = 0;           //first sem(index=0)
50     sem_b.sem_flg = SEM_UNDO;
51     sem_b.sem_op = -1;          //Increase 1,make sem=1
52
53     while(1)
54     {
55         if(1 == (value = semctl(sem_id, 0, GETVAL)))
56         {
57             printf("\nNow, receive message process running:\n");
58             printf("\tThe message is : %s\n", shm_ptr);
59
60             if(-1 == semop(sem_id, &sem_b, 1))
61             {
62                 perror("semop");
63                 exit(EXIT_FAILURE);
64             }
65         }
66         //if enter "end", then end the process
67         if(0 == (strcmp(shm_ptr , "end")))
68         {
69             printf("\nExit the receiver process now!\n");
70             break;
71         }
72     }
73     shmdt(shm_ptr);
74     //7. delete the shared memory
75     if(-1 == shmctl(shm_id, IPC_RMID, NULL))
76     {
77         perror("shmctl");
78         exit(EXIT_FAILURE);
79     }
80     //8. delete the semaphore
81     if(-1 == semctl(sem_id, 0, IPC_RMID))
82     {
83         perror("semctl");

```

```
84         exit(EXIT_FAILURE);
85     }
86     return 0;
87 }
```

2. 程序解释

下面以sender.c为例解释一下如何创建共享内存并通过信号量机制实现互斥访问从而达到进程间通信的目的。

- 创建一个共享内存的ID,就是代码中的key

```
1 key_t key;
2 key = ftok(".", 0xFF);
```

通过ftok函数创建一个key_t类型的变量，作为共享内存的key，ftok函数的两个参数分别是文档名(一个存在的路径),上例中的路径是.表示当前路径，另一个参数是子序号

- 创建并初始化信号量

```
1 int sem_id;
2 sem_id = semget(key, 1, IPC_CREAT|0644);
3 if(-1 == sem_id)
4 {
5     perror("semget");
6     exit(EXIT_FAILURE);
7 }
8 if(-1 == (semctl(sem_id, 0, SETVAL, value)))
9 {
10    perror("semctl");
11    exit(EXIT_FAILURE);
12 }
```

通过semget()函数创建一个信号量，初始值为1，再通过semctl()函数初始化该信号量

- 创建共享内存并挂载在进程中

```
1 //4. Create the shared memory(1K bytes)
2 shm_id = shmget(key, 1024, IPC_CREAT|0644);
3 if(-1 == shm_id)
4 {
5     perror("shmget");
6     exit(EXIT_FAILURE);
7 }
8 //5. attach the shm_id to this process
9 char *shm_ptr;
10 shm_ptr = shmat(shm_id, NULL, 0);
11 if(NULL == shm_ptr)
12 {
```

```
13     perror("shmat");
14     exit(EXIT_FAILURE);
15 }
```

在这部分代码中，首先通过shmget()函数创建了一个大小为1000B的共享内存，然后通过shmat函数，将刚刚创建的共享内存以可读写的方式挂载在进程上，并且指定系统将自动选择一个合适的地址给共享内存，将挂载的共享内存地址赋值给char型指针shm_ptr

- Sender主循环

```
1 while(1)
2 {
3     if(0 == (value = semctl(sem_id, 0, GETVAL)))
4     {
5         printf("\nNow, snd message process running:\n");
6         printf("\tInput the snd message: ");
7         scanf("%s", shm_ptr);
8
9         if(-1 == semop(sem_id, &sem_b, 1))
10    {
11        perror("semop");
12        exit(EXIT_FAILURE);
13    }
14 }
15 //if enter "end", then end the process
16 if(0 == (strcmp(shm_ptr , "end")))
17 {
18     printf("\nExit sender process now!\n");
19     break;
20 }
21 }
```

主循环中首先判断表示共享内存访问情况的信号量是否为0(为0表示共享内存空闲)，如果为0的话提示用户输入想要输入的消息，并将用户输入的消息写入共享内存中，写完后通过semop函数将信号量加一，通知receiver读取消息。并且定义一个end命令表示退出当前进程。循环退出的时候取消共享内存的挂载

- Receiver主循环

```
1 while(1)
2 {
3     if(1 == (value = semctl(sem_id, 0, GETVAL)))
4     {
5         printf("\nNow, receive message process running:\n");
6         printf("\tThe message is : %s\n", shm_ptr);
7
8         if(-1 == semop(sem_id, &sem_b, 1))
9     {
```

```

10         perror("semop");
11         exit(EXIT_FAILURE);
12     }
13 }
14 //if enter "end", then end the process
15 if(0 == (strcmp(shm_ptr , "end")))
16 {
17     printf("\nExit the receiver process now!\n");
18     break;
19 }
20 }

```

主循环中首先判断表示共享内存访问情况的信号量是否为1(为1表示共享内存已经写入消息, 可以读取), 如果为1的话输出该消息, 输出后通过semop函数将信号量减1, 通知Sender可以再次写入消息。并且定义一个end命令表示退出当前进程。循环退出的时候取消共享内存的挂载

3. 实验现象

将上述源码编译后进行测试, 得到下面的结果。

```

x sun@iZwz9evpg61guy89phiy9yZ ~/study/OS/16281047_OperatingSystemExperiment/lab3 ➤ master ➤ ./Sender
Now, snd message process running:
Input the snd message: HelloWorld!
Now, snd message process running:
Input the snd message: e^H
Now, snd message process running:
Input the snd message: end
Exit sender process now!
sun@iZwz9evpg61guy89phiy9yZ ~/study/OS/16281047_OperatingSystemExperiment/lab3 ➤ master ➤

x sun@iZwz9evpg61guy89phiy9yZ ~/study/OS/16281047_OperatingSystemExperiment/lab3 ➤ master ➤ ./Receiver
Now, receive message process running:
The message is : HelloWorld!
Now, receive message process running:
The message is : e
Exit the receiver process now!
sun@iZwz9evpg61guy89phiy9yZ ~/study/OS/16281047_OperatingSystemExperiment/lab3 ➤ master ➤

```

可以看到sender进程发出的消息receiver进程均准确无误的收到

4. 删除互斥访问相关的代码

程序主要的代码没有变化, 只是在Sender和Receiver进程的主循环中将用于控制互斥访问共享内存的相关代码删除, 注释后的结果如下:

Sender_2.c:

```

1 while(1)
2 {
3     printf("\nNow, snd message process running:\n");
4     printf("\tInput the snd message:  ");
5     scanf("%s", shm_ptr);
6     //if enter "end", then end the process
7     if(0 == (strcmp(shm_ptr , "end")))
8     {
9         printf("\nExit sender process now!\n");
10        break;
11    }
12 }

```

Receiver_2.c:

```

1  while(1)
2  {
3      printf("\nNow, receive message process running:");
4      printf("\tThe message is : %s\n", shm_ptr);
5
6      //if enter "end", then end the process
7      if(0 == (strcmp(shm_ptr , "end")))
8      {
9          printf("\nExit the receiver process now!\n");
10         break;
11     }
12     sleep(3);
13 }

```

最后加一个sleep(1)用于控制打印的速度，便于观察现象

5. 删除互斥访问后的实验现象

<pre> sun@iZwz9evpg61guy89phiy9yZ ~:/study/OS/16281047_OperatingSystemExperiment/lab3 [master] ./Sender_2 Now, snd message process running: Input the snd message: wdjcghjvc Now, snd message process running: Input the snd message: </pre>	<pre> sun@iZwz9evpg61guy89phiy9yZ ~:/study/OS/16281047_OperatingSystemExperiment/lab3 [master] ./Receiver_2 Now, receive message process running: The message is : Now, receive message process running: The message is : wdjcghjvc Now, receive message process running: The message is : wdjcghjvc Now, receive message process running: The message is : wdjcghjvc Now, receive message process running: The message is : wdjcghjvc Now, receive message process running: The message is : wdjcghjvc Now, receive message process running: The message is : wdjcghjvc </pre>
---	---

实验现象解释：当删除互斥访问之后，两个进程便没有限制的访问共享内存，Sender进程由于受限于用户输入的速度，会停留一直等待用户输入数据，但是Receiver进程会一直输出共享内存中的消息。

6. 打印Sender和Receiver进程中共享内存的地址

在原始代码的基础上修改，具体代码文件分别是 Sender_3.c 和 Receiver_3.c，具体修改就是如下：

```

48 //5. attach the shm_id to this process
49 char *shm_ptr;
50 shm_ptr = shmat(shm_id, NULL, 0);
51 printf("Sender_3进程中共享内存地址为: %p",shm_ptr);

```

在挂载共享内存后打印挂载后的地址

7. 打印共享内存地址实验现象

<pre> sun@iZwz9evpg61guy89phiy9yZ ~:/study/OS/16281047_OperatingSystemExperiment/lab3 [master] ./Sender_3 Sender_3进程中共享内存地址为: 0x7f0f9c1a6000 Now, snd message process running: Input the snd message: jkgcbkhxbakj Now, snd message process running: Input the snd message: mhacshgs Now, snd message process running: Input the snd message: end Exit sender process now! </pre>	<pre> sun@iZwz9evpg61guy89phiy9yZ ~:/study/OS/16281047_OperatingSystemExperiment/lab3 [master] ./Receiver_3 Receiver_3进程中共享内存的地址为: 0x7f4f3a097000 Now, receive message process running: The message is : jkgcbkhxbakj Now, receive message process running: The message is : mhacshgs Exit the receiver process now! </pre>
---	---

可以看到实验现象，在两个进程中共享内存的地址不一样

现象解释：

通过上面的现象可以看到共享内存存在不同进程中是不相同的，总结有以下的原因导致共享内存存在不同进程中的地址不一样：

- 进程在挂载内存的时候使用的 `shmat()` 函数中的第二个参数使用的是NULL，NULL参数的含义是进程让系统分配给共享内存合适的地址。在 `shmat()` 函数中，第二个参数有三种选择，分别是：

参数值	NULL	addr	addr
含义	系统将自动选择一个合适的地址	如果shmaddr非0 并且指定了SHM_RND 则此段连接到shmaddr - (shmaddr mod SHMLAB)所表示的地址上。	第三个参数如果在flag中指定了SHM_RDONLY位，则以只读方式连接此段，否则以读写的方式连接此段。

可以看到，当addr有具体的值的时候，便将共享内存挂载到指定的地址上

- 现代操作系统中都存在ASLR(地址空间随机化)，ASLR是一种针对缓冲区溢出的安全保护机制，具有ASLR机制的操作系统每次加载到内存的程序起始地址会随机变化。系统的这个随机化操作可能导致共享内存的地址不一致。

验证：

- 指定Sender_4.c和Receiver_4.c中共享内存的挂载地址为 `0x7fcc2c0bb000`

- 修改具体的代码如下：

```
48 //5. attach the shm_id to this process
49 char *shm_ptr;
50 shm_ptr = shmat(shm_id, 0x7fcc2c0bb000, 0);
51 printf("Sender_4进程中共享内存的地址为:%p", shm_ptr);
52 if(NULL == shm_ptr)
53 {
54     perror("shmat");
55     exit(EXIT_FAILURE);
56 }
```

- 运行结果：

```
sun@iZwz9evpg61guy89phiy9yZ ~]$ ./Sender_4
Sender_4进程中共享内存的地址为:0x7fcc2c0bb000
Now, send message process running:
Input the send message: hdcvjh

Now, send message process running:
Input the send message: end

Exit sender process now!
sun@iZwz9evpg61guy89phiy9yZ ~]$ ./Receiver_4
Receiver_4进程中共享内存的地址为:0x7fcc2c0bb000
Now, receive message process running:
The message is : hdcvjh
```

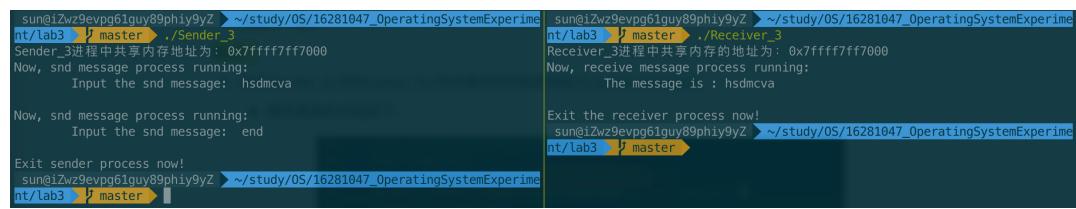
实验结果结果佐证了上面的现象解释，通过指定挂载共享内存的地址，可以使共享内存的地址一致，可以随意指定改地址

- 关闭系统的ASLR操作

- 具体的关闭命令如下：

```
1 | sudo su
2 | sysctl -w kernel.randomize_va_space=0
```

■ 运行结果：



The screenshot shows two terminal windows. The left window is titled 'Sender_3' and the right window is titled 'Receiver_3'. Both windows are running on a host named 'sun' with IP '127.0.0.1'. In the Sender window, the process ID is 1111. It shows the message 'Input the snd message: hsdmcva' being sent. In the Receiver window, the process ID is 1112. It shows the message 'The message is : hsdmcva' received. The command 'Exit sender process now!' is also visible in the Sender window.

这个实验现象也佐证了系统的ASLR也对导致挂载的共享内存地址不一样

4.2.2 管道通信

(1) 无名管道

1. 实验源码

pipe.c:

```
1 | #include <stdio.h>
2 | #include <unistd.h>      //for pipe()
3 | #include <string.h>       //for memset()
4 | #include <stdlib.h>       //for exit()
5 | int main()
6 | {
7 |     int fd[2];
8 |     char buf[20];
9 |     if(-1 == pipe(fd))
10 |    {
11 |        perror("pipe");
12 |        exit(EXIT_FAILURE);
13 |    }
14 |    write(fd[1], "hello,world", 12);
15 |    memset(buf, '\0', sizeof(buf));
16 |    read(fd[0], buf, 12);
17 |    printf("The message is: %s\n", buf);
18 |    return 0;
19 | }
```

2. 程序解释

- 通过pipe函数创建管道，函数传递一个整形数组fd，fd的两个整形数表示的是两个文件描述符，其中第一个用于读取数据，第二个用于写数据。两个描述符相当远管道的两端，一段负责写数据，一段负责读数据。
- pipe管道是半双工的工作模式，某一时刻只能读或者只能写
- 读写管道就和读写普通文件一样，使用write和read

3. 实验现象

```
sun@iZwz9evpg61guy89phiy9yZ ~/study/OS/16281047_OperatingSystemExperiment/lab3 master ./pipe
The message is: hello,world
```

4. 无名管道同步机制验证

为了验证无名管道的同步机制，在上述代码的基础上进行修改，得到如下的代码

pipe_2.c:

```
1 #include <stdio.h>
2 #include <unistd.h>      //for pipe()
3 #include <string.h>       //for memset()
4 #include <stdlib.h>       //for exit()
5 int main()
6 {
7     int fd[2];
8     char buf[200]={0};
9     pid_t child;
10    //创建管道
11    if(-1 == pipe(fd))
12    {
13        perror("pipe");
14        exit(EXIT_FAILURE);
15    }
16    //创建子进程
17    child=fork();
18    if(child== -1)
19    {
20        perror("fork");
21        exit("EXIT_FAILURE");
22    }
23    if(child==0)
24    {
25        //关闭子进程中不需要的写描述符
26        close(fd[1]);
27        while(1)
28        {
29            if(read(fd[0],buf,sizeof(buf))>0)
30                printf("子进程接收的消息是:%s\n",buf);
31            else
32                printf("子进程:管道中没有数据\n");
33                sleep(2);
34                if(strcmp(buf,"end")==0)
35                    break;
36                memset(buf,0,sizeof(buf));
37        }
38    }
39    if(child>0)
40    {
```

```

41     close(fd[0]);
42     while(1)
43     {
44         printf("父进程中-请输入消息:");
45         scanf("%s",buf);
46         write(fd[1],buf,strlen(buf));
47         if(strcmp(buf,"end")==0)
48             break;
49     }
50 }
51     return 0;
52 }
```

对于上述代码做出如下解释：父进程是消息的发送者，在父进程中创建了两个文件描述符，fork一个子进程的时候会复制这两个管道文件描述符，因此父进程和子进程都会将自己的那个用不到的文件描述符关闭。父进程中会持续向管道中写入用户输入的消息，子进程会一直输出管道中的消息，如果管道中没有消息就会阻塞等待。

5. 无名管道同步机制实验现象

```

ent/lab3 > master > ./pipe_2
父进程中-请输入消息:helloworld
父进程中-请输入消息:子进程接收的消息是:helloworld
gcbdhscjnksdjc
父进程中-请输入消息:子进程接收的消息是:gcbdhscjnksdjc
chjsbkncclkjd
父进程中-请输入消息:c子进程接收的消息是:chjsbkncclkjd
hjsdhbvcks
父进程中-请输入消息:子进程接收的消息是:chjsdhbvcks

dgfh
父进程中-请输入消息:子进程接收的消息是:dgfh
ghj
父进程中-请输入消息:hgjk
父进程中-请输入消息:子进程接收的消息是:ghjhgjk
gfg
父进程中-请输入消息:子进程接收的消息是:gfg

dgfd
父进程中-请输入消息:g
父进程中-请输入消息:dgd
父进程中-请输入消息:gd
父进程中-请输入消息:
dg子进程接收的消息是:dgfdgfdgfd
```

可以看到输出进程是按照输入进程输入的顺序输出数据，并且当输入进程没有数据输入，即管道中没有数据的时候，输出进程会阻塞。因此无名管道通信系统调用的时候已经实现了同步机制

6. 无名管道同步机制原理

通过上面的实验和查阅相关资料，得到无名管道如下的同步机制：

- 管道的读写通过两个系统调用write和read实现
- 发送者在向管道内存中写入数据之前，首先检查内存是否被读进程锁定和内存中是否还有剩

余空间，如果这两个要求都满足的话write函数会对内存上锁，然后进行写入数据，写完之后解锁；否则就会等待(阻塞)。

- 写进程在读取管道中的数据之前，也会检查内存是否被读进程锁定和管道内存中是否有数据，如果满足这两个条件，read函数会对内存上锁，读取数据后在解锁；否则会等到(阻塞)

(2) 有名管道

1. 实验代码

有名管道实验中设计两个代码文件 `fifo_send.c` 和 `fifo_recv.c`

`fifo_send.c:`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/stat.h>
5 #include <sys/ipc.h>
6 #include <fcntl.h>
7 #define FIFO "./my_fifo"
8
9 int main()
10 {
11     char buf[] = "hello,world";
12     //1. check the fifo file existed or not
13     int ret;
14     ret = access(FIFO, F_OK);
15     if(ret != 0)      //file /tmp/my_fifo existed
16     {
17         if(-1 == mkfifo(FIFO, 0766))
18         {
19             perror("mkfifo");
20             exit(EXIT_FAILURE);
21         }
22     }
23
24     //3.Open the fifo file
25     int fifo_fd;
26     fifo_fd = open(FIFO, O_WRONLY);
27     if(-1 == fifo_fd)
28     {
29         perror("open");
30         exit(EXIT_FAILURE);
31     }
32     //4. write the fifo file
33     int num = 0;
34     num = write(fifo_fd, buf, sizeof(buf));
35     if(num < sizeof(buf))
36     {
```

```

38         perror("write");
39         exit(EXIT_FAILURE);
40     }
41     printf("write the message ok!\n");
42
43     close(fifo_fd);
44
45     return 0;
46 }
```

`fifo_rcv.c:`

```

1  /*
2  *File: fifo_rcv.c
3  */
4
5 #include <stdio.h>
6 #include <string.h>
7 #include <stdlib.h>
8 #include <unistd.h>
9 #include <sys/stat.h>
10 #include <sys/ipc.h>
11 #include <fcntl.h>
12
13
14 #define FIFO "./my_fifo"
15
16 int main()
17 {
18     char buf[20] ;
19     memset(buf, '\0', sizeof(buf));
20
21     //^. check the fifo file existed or not
22     int ret;
23     ret = access(FIFO, F_OK);
24     if(ret != 0)      //file /tmp/my_fifo existed
25     {
26         if(-1==mkfifo(FIFO,0766))
27         {
28             perror("mkfifo");
29             exit("EXIT_FAILURE");
30         }
31     }
32
33 // 2.Open the fifo file
34     int fifo_fd;
35     fifo_fd = open(FIFO, O_RDONLY);
36     if(-1 == fifo_fd)
37     {
```

```

38         perror("open");
39         exit(EXIT_FAILURE);
40     }
41     //4. read the fifo file
42     int num = 0;
43     num = read(fifo_fd, buf, sizeof(buf));
44     printf("Read %d words: %s\n", num, buf);
45     close(fifo_fd);
46     return 0;
47 }
```

2. 程序解释

- 写进程fifo_send分为四个步骤执行，首先判断当前目录下是否已经存在my_fifo文件，不存在的话在当前目录下通过mkfifo()函数创建FIFO类型的文件my_fifo；再通过open()函数打开my_fifo文件，最后向文件中写入消息；
- 读进程的过程和写进程的类似，只没有了创建fifo文件的过程而已

3. 实验现象

```

sun@iZwz9evpg61guy89phiy9yZ ~/study/OS/16281047_OperatingSystemExperiment/lab3 $ ./fifo_send
write the message ok!
sun@iZwz9evpg61guy89phiy9yZ ~/study/OS/16281047_OperatingSystemExperiment/lab3 $ ./fifo_rcv
Read 12 words: hello,world
sun@iZwz9evpg61guy89phiy9yZ ~/study/OS/16281047_OperatingSystemExperiment/lab3 $
```

现象描述：在仅仅只运行fifo_send进程的时候，没有任何输出，进程一直阻塞，直到fifo_rcv进程运行，两个进程才开始输出信息。

当写进程和读进程都设置成阻塞状态的时候，不论先执行那个进程，先执行的进程都会阻塞等待，待另一个进程执行后两个进程才正常执行。

4. 探究有名管道的同步和阻塞机制

通过 `fifo_fd=open(FIFO,O_RDONLY | O_NONBLOCK)` 设置为非阻塞状态，`fifo_fd=open(FIFO,O_RDONLY)` 设置为阻塞状态，对应四个进程分别为fifo_send(阻塞)、fifo_rcv(阻塞)、fifo_send_1(非阻塞)、fifo_rcv_1(非阻塞)

- 读进程阻塞、写进程阻塞
 - 先执行fifo_send后执行fifo_rcv，结果正确
 - 截图请见上面的实验现象
- 先执行fifo_rcv后执行fifo_send，结果正确

```

sun@iZwz9evpg61guy89phiy9yZ ~/study/OS/16281047_OperatingSystemExperiment/lab3 $ ./fifo_rcv
Read 12 words: hello,world
sun@iZwz9evpg61guy89phiy9yZ ~/study/OS/16281047_OperatingSystemExperiment/lab3 $ ./fifo_send
write the message ok!
sun@iZwz9evpg61guy89phiy9yZ ~/study/OS/16281047_OperatingSystemExperiment/lab3 $
```

具体的原因是读进程在open FIFO的时候由于没有s

通过查阅资料得到了FIFO管道的阻塞机制如下：

对于设置了阻塞的读进程而言：

- 读进程阻塞的原因有三种：FIFO 中没有数据、有其他的读进程正在读取这些数据、没有写进程打开FIFO文件
- 不论是哪种原因引起的阻塞，解开阻塞的原因都是FIFO有新的数据写入
- 如果一个读进程有多个read操作，那么只会阻塞第一个read，其他的不会发生阻塞

对于设置了阻塞的写进程而言：

1. 当写入的数据量小于PIPE_BUF时，Linux保证写入原子性。如果此时管道中的空闲位置不足以容纳要写入的数据，泽写进程阻塞，直到管道中空间足够，一次性写入所有数据
2. 当写入的数据量大于PIPE_BUF时，Linux不再保证写入的原子性。一旦管道中有空闲位置便尝试写入数据，直到所有数据写入完成后返回。

○ 读进程阻塞，写进程非阻塞

- 先执行fifo_send_1后执行fifo_rcv，写进程open函数返回-1

```
x sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master] ./fifo_send_1
open: No such device or address
x sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master]
x sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master] ./fifo_rcv
~
```

- 先执行fifo_rcv后执行fifo_send_1，结果正常

```
x sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master] ./fifo_rcv
Read 12 words: hello,world
sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master]
x sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master] ./fifo_send_1
write the message ok!
sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master]
```

○ 读进程非阻塞，写进程阻塞

- 先执行fifo_send后执行fifo_rcv_1,结果正常

```
x sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master] ./fifo_send
write the message ok!
sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master]
x sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master] ./fifo_rcv_1
Read 12 words: hello,world
sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master]
```

- 先执行fifo_rcv_1后执行fifo_send，程序崩溃

```
x sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master] ./fifo_send
|
x sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master] ./fifo_rcv_1
Read 0 words:
sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master]
```

○ 读写进程都是非阻塞

- 先执行fifo_send_1后执行fifo_rcv_1

```
x sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master] ./fifo_send_1
open: No such device or address
x sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master]
x sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master] ./fifo_rcv_1
Read 0 words:
sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master]
```

- 先执行fifo_rcv_1后执行fifo_send_1

```
x sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master] ./fifo_send_1
open: No such device or address
x sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master]
x sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master] ./fifo_rcv_1
Read 0 words:
sun@iZwz9evpg61guy89phiy9yZ ~/study/05/16281047_OperatingSystemExperiment/lab3 [master]
```

4.2.3 消息队列

1. 实验代码

本实验代码文件分为Server.c和Client.c两个

Server.c:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <sys/types.h>
```

```

6 #include <sys/msg.h>
7 #include <sys/ipc.h>
8 #include <signal.h>
9
10#define BUF_SIZE 128
11
12//Rebuild the strcut (must be)
13struct msgbuf
14{
15    long mtype;
16    char mtext[BUF_SIZE];
17};
18int main(int argc, char *argv[])
19{
20    //1. creat a mseg queue
21    key_t key;
22    int msgId;
23
24    key = ftok(".", 0xFF);
25    msgId = msgget(key, IPC_CREAT|0644);
26    if(-1 == msgId)
27    {
28        perror("msgget");
29        exit(EXIT_FAILURE);
30    }
31
32    printf("Process (%s) is started, pid=%d\n", argv[0], getpid());
33
34    while(1)
35    {
36        alarm(0);
37        alarm(600);      //if doesn't receive messge in 600s, timeout &
38        exit
39        struct msgbuf rcvBuf;
40        memset(&rcvBuf, '\0', sizeof(struct msgbuf));
41        msgrcv(msgId, &rcvBuf, BUF_SIZE, 1, 0);
42        printf("Receive msg: %s\n", rcvBuf.mtext);
43
44        struct msgbuf sndBuf;
45        memset(&sndBuf, '\0', sizeof(sndBuf));
46
47        strncpy((sndBuf.mtext), (rcvBuf.mtext),
48        strlen(rcvBuf.mtext)+1);
49        sndBuf.mtype = 2;
50
51        if(-1 == msgsnd(msgId, &sndBuf, strlen(rcvBuf.mtext)+1, 0))
52        {
53            perror("msgsnd");
54            exit(EXIT_FAILURE);

```

```

53     }
54
55     //if scanf "end~", exit
56     if(!strcmp("end~", rcvBuf.mtext))
57         break;
58 }
59 printf("THe process(%s),pid=%d exit~\n", argv[0], getpid());
60 return 0;
61 }
```

Client.c:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <sys/types.h>
6 #include <sys/msg.h>
7 #include <sys/ipc.h>
8 #include <signal.h>
9
10#define BUF_SIZE 128
11
12//Rebuild the strcut (must be)
13struct msgbuf
14{
15    long mtype;
16    char mtext[BUF_SIZE];
17};
18
19
20int main(int argc, char *argv[])
21{
22    //1. creat a msg queue
23    key_t key;
24    int msgId;
25
26    printf("THe process(%s),pid=%d started~\n", argv[0], getpid());
27
28    key = ftok(".", 0xFF);
29    msgId = msgget(key, IPC_CREAT|0644);
30    if(-1 == msgId)
31    {
32        perror("msgget");
33        exit(EXIT_FAILURE);
34    }
35
36    //2. creat a sub process, wait the server message
37    pid_t pid;
```

```

38     if(-1 == (pid = fork()))
39     {
40         perror("vfork");
41         exit(EXIT_FAILURE);
42     }
43
44     //In child process
45     if(0 == pid)
46     {
47         while(1)
48         {
49             alarm(0);
50             alarm(100);      //if doesn't receive messge in 100s,
51             timeout & exit
52             struct msgbuf recvBuf;
53             memset(&recvBuf, '\0', sizeof(struct msgbuf));
54             msgrcv(msgId, &recvBuf, BUF_SIZE, 2, 0);
55             printf("Server said: %s\n", recvBuf.mtext);
56         }
57
58         exit(EXIT_SUCCESS);
59     }
60
61     else    //parent process
62     {
63         while(1)
64         {
65             usleep(100);
66             struct msgbuf sndBuf;
67             memset(&sndBuf, '\0', sizeof(sndBuf));
68             char buf[BUF_SIZE] ;
69             memset(buf, '\0', sizeof(buf));
70
71             printf("\nInput snd mesg: ");
72             scanf("%s", buf);
73
74             strncpy(sndBuf.mtext, buf, strlen(buf)+1);
75             sndBuf.mtype = 1;
76
77             if(-1 == msgsnd(msgId, &sndBuf, strlen(buf)+1, 0))
78             {
79                 perror("msgsnd");
80                 exit(EXIT_FAILURE);
81             }
82             //if scanf "end~", exit
83             if(!strcmp("end~", buf))
84                 break;
85         }

```

```

86         printf("The process(%s), pid=%d exit~\n", argv[0], getpid());
87     }
88     return 0;
89 }

```

2. 程序解释

- 程序分为服务器端和客户端，客户端向服务器发起通信，服务器端收到数据后将一模一样的数据返回
- 通过msgsrcv函数读取客户端传过来的消息，msgsrcv的参数列表见下面。

```
int msgsrcv(int msqid, void *ptr, size_t length, long type, int flag);
```

参数	msgid	ptr	length	type	flag
含义	消息队列标识符	消息缓冲区指针	消息数据长度	决定从队列中返回那一条消息	阻塞与否
备注				=0 返回消息队列中第一条消息 >0 返回消息队列中等于mtype 类型的第一条消息。<0 返回mtype<=type 绝对值最小值的第一条消息。	msgflg 为 0 表示阻塞方式，设置IPC_NOWAIT 表示非阻塞方式

- 通过msgsnd函数向消息队列中加入消息，msgsnd的参数列表见下面。

```
int msgsnd(int msqid, const void *ptr, size_t length, int flag);
```

参数	msgid	ptr	length	flag
含义	消息队列标识符	消息缓冲区指针	消息数据长度	阻塞与否
备注				msgflg 为 0 表示阻塞方式，设置IPC_NOWAIT 表示非阻塞方式

- 客户端的子进程主要负责消息的接受，父进程主要负责消息的发送；
- 通过分析上面的代码可以知道，客户端和服务器端都是以阻塞的方式读取和写入消息

3. 程序运行结果

```
sun@iZwz9evpg61guy89phiy9yZ ~:/study/OS/16281047_OperatingSystemExperiment/lab3 ➜ 2 master ➜ ./Server
Process (. /Server) is started, pid=10921
Receive msg: helloworld
Receive msg: 孙汉武
Receive msg: end
Receive msg: end~
THe process(. /Server),pid=10921 exit~
sun@iZwz9evpg61guy89phiy9yZ ~:/study/OS/16281047_OperatingSystemExperiment/lab3 ➜ 2 master ➜
sun@iZwz9evpg61guy89phiy9yZ ~:/study/OS/16281047_OperatingSystemExperiment/lab3 ➜ 3 master ➜ ./Client
THe process(. /Client),pid=10922 started~
Input snd mesg: helloworld
Server said: helloworld

Input snd mesg: 孙汉武
Server said: 孙汉武

Input snd mesg: end
Server said: end

Input snd mesg: end~
THe process(. /Client),pid=10922 exit~
Server said: end~
sun@iZwz9evpg61guy89phiy9yZ ~:/study/OS/16281047_OperatingSystemExperiment/lab3 ➜ 3 master ➜
```

4. 探究消息队列的同步和阻塞机制

通过上面的程序解释中可以看出，消息队列通过msgrcv和msgsnd两个函数的flag参数控制是否阻塞，将其设置为IPC_NOWAIT表示不阻塞；如果客户端和服务器端都设置阻塞话，就可以达到同步的目的

现在做出如下探究：

- 客服端不阻塞(代码为Client_1.c),服务器端阻塞, 得到结果如下。

可以看到当客户端不阻塞的话在客户端接受服务器端消息的时候会无限制的打印消息队列中的空消息，哪怕消息队列中没有任何消息

- 客户端阻塞，服务器端不阻塞(代码为Server_1.c)

```
Receive msg:  
Rec^C  
x sun@iZwz9evpg61guy89phiy9yZ ➤ ~/study/OS/16281047_OperatingSystemExperiment/lab3 ↵ master ➤  
x sun@iZwz9evpg61guy89phiy9yZ ➤ ~/study/OS/16281047_OperatingSystemExperiment/lab3 ↵ master ➤
```

可以看到当服务器端没有设置阻塞的时候，服务器端会一直接受消息队列中的空消息并向客户端转发。

Task 5

本实验分析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。

我们首先从 `devices/timer.c` 文件中的 `timer_sleep` 函数开始分析，下面是该函数的具体代码：

```

1  /* Sleeps for approximately TICKS timer ticks.  Interrupts must
2   be turned on. */
3  void timer_sleep (int64_t ticks)
4  {
5      int64_t start = timer_ticks ();
6      ASSERT (intr_get_level () == INTR_ON);
7      while (timer_elapsed (start) < ticks)
8          thread_yield ();
9  }

```

下面开始逐行分析这个函数，第5行的 `timer_ticks` 函数也在 `timer.c` 文件中，跳转到该函数中：

```

1  /* Returns the number of timer ticks since the OS booted. */
2  int64_t timer_ticks (void)
3  {
4      enum intr_level old_level = intr_disable ();
5      int64_t t = ticks;
6      intr_set_level (old_level);
7      return t;
8  }

```

`timer_ticks` 函数中第4行涉及一个名为 `intr_disable()` 的函数，该函数的具体定义在 `devices/interrupt.c` 文件中。

```

1  /* Disables interrupts and returns the previous interrupt status. */
2  enum intr_level intr_disable (void)
3  {
4      enum intr_level old_level = intr_get_level ();
5
6      /* Disable interrupts by clearing the interrupt flag.
7       See [IA32-v2b] "CLI" and [IA32-v3a] 5.8.1 "Masking Maskable
8       Hardware Interrupts". */
9      asm volatile ("cli" : : : "memory");
10     return old_level;
11 }

```

再看看返回值 `intr_level` 是个什么结构，代码在 `devices/interrupt.h` 中：

```

1  /* Interrupts on or off? */
2  enum intr_level
3  {
4      INTR_OFF,           /* Interrupts disabled. */
5      INTR_ON            /* Interrupts enabled. */
6  };

```

可以发现，`intr_level`这个枚举类型表示的是是否允许中断。于是分析得到`intr_disable`函数做了两件事。
1. 调用`intr_old_level`函数
2. 直接执行汇编代码保证这个线程不能被中断。之后返回调用`intr_old_level`函数的返回值。

再看看`intr_get_level`函数的实现细节，该函数的定义也在`devices/interrupt.c`文件中

```
1  /* Returns the current interrupt status. */
2  enum intr_level intr_get_level (void)
3  {
4      uint32_t flags;
5      /* Push the flags register on the processor stack, then pop the
6         value off the stack into `flags'. See [IA32-v2b] "PUSHF"
7         and "POP" and [IA32-v3a] 5.8.1 "Masking Maskable Hardware
8         Interrupts". */
9      asm volatile ("pushfl; popl %0" : "=g" (flags));
10     return flags & FLAG_IF ? INTR_ON : INTR_OFF;
11 }
```

通过注释信息和分析汇编代码可以知道，`intr_get_level`这个函数的作用是返回当前的中断状态。`intr_get_level`函数弄清楚了之后，返回上一层函数中，到了`intr_disable`函数中，这样就可以清楚的知道`intr_disable`函数的作用：

- 获取当前中断状态
- 将当前中断状态更改为不可中断
- 返回先前的中断状态

弄清楚了`intr_disable`函数，接着看`timer_ticks`函数的5、6、7行

- 第5行通过一个`int64_t`类型的变量`t`获取全局变量`ticks`的值；
- 第6行`intr_set_level(old_level)`表示将当前中断状态设置为之前的中断状态。
- 第7行返回`t`

这样，函数`timer_ticks`的含义也就弄清楚了。其实`timer_ticks`函数的作用很简单，就是想获取当前系统的`ticks`值而已，而上面通过这么大篇幅的介绍`timer_ticks`函数的4、6两行的作用，原因是第4行和第6行通过先关闭中断，待`t`获取到`ticks`值之后再恢复之前的中断状态，来保证操作的原子性，简单的说就是在`t`获取全局变量`ticks`的值的时候，不能被打断。

然后接着分析`timer_sleep`函数的第6行`ASSERT (intr_get_level () == INTR_ON);`这里是一个断言，当`intr_get_level`函数获取的当前中断状态不是`INTR_ON`的时候发生警告且退出。

`timer_sleep`函数剩下的就是一个循环了：

```
1  while (timer_elapsed (start) < ticks)
2      thread_yield ();
```

通过分析不难得出`timer_elapsed()`函数的作用是计算当前的系统`ticks`减去之前得到的`start`的差值，如果这个差值小于函数参数`ticks`的话一直执行`thread_yield()`函数。

再看看`thread_yield`函数的具体定义（在`thread/thread.c`文件中），分析一下该函数的作用：

```

1  /* Yields the CPU.  The current thread is not put to sleep and
2   may be scheduled again immediately at the scheduler's whim. */
3  void thread_yield (void)
4  {
5      struct thread *cur = thread_current ();
6      enum intr_level old_level;
7      ASSERT (!intr_context ());
8      old_level = intr_disable ();
9      if (cur != idle_thread)
10         list_push_back (&ready_list, &cur->elem);
11      cur->status = THREAD_READY;
12      schedule ();
13      intr_set_level (old_level);
14 }

```

`thread_yield` 函数第5行顾名思义，作用就是返回当前正在运行的线程，通过一个`thread`类型的结构体指针接受该函数返回值。

`thread_yield` 函数的第7行通过断言的方式判断中断类型，如果是由于I/O等引起的硬中断则退出，如果是软中断的话正常运行。

再看第8行和第13行的之前也分析过，这是保证9-12行操作的原子性。

再分析9-12行：

- 9-10行：如何当前线程不是空闲的线程就调用`list_push_back`把当前线程的元素扔到就绪队列里面，
- 11行：把线程改成`THREAD_READY`状态
- 12行：调用`schedule`函数

再深入 `schedule` 函数(`thread/thread.c`文件)看看

```

1  /* Schedules a new process.  At entry, interrupts must be off and
2   the running process's state must have been changed from
3   running to some other state.  This function finds another
4   thread to run and switches to it.
5   It's not safe to call printf() until thread_schedule_tail()
6   has completed. */
7  static void schedule (void)
8  {
9      struct thread *cur = running_thread ();
10     struct thread *next = next_thread_to_run ();
11     struct thread *prev = NULL;
12
13     ASSERT (intr_get_level () == INTR_OFF);
14     ASSERT (cur->status != THREAD_RUNNING);
15     ASSERT (is_thread (next));
16
17     if (cur != next)
18         prev = switch_threads (cur, next);

```

```
19     thread_schedule_tail (prev);  
20 }
```

schedule 函数首先获取当前正在运行的线程指针cur和下一个运行的线程next，之后是三个断言。

- `ASSERT (intr_get_level () == INTR_OFF)`：保证中断状态是开启的
- `ASSERT (cur->status != THREAD_RUNNING)`：保证当前运行的线程是RUNNING_THREAD的
- `ASSERT (is_thread (next))`：保证下一个线程有效

17-18行的作用是：如果当前线程和下一个要跑的线程不是同一个的话调用switch_threads返回给prev

下面再看看 `switch_threads` 函数(在 `threads/switch.s` 中)这是一个完全由汇编语言编写的函数

```
1 #include "threads/switch.h"  
2  
3 ##### struct thread *switch_threads (struct thread *cur, struct thread  
4 *next);  
4 #####  
5 ##### Switches from CUR, which must be the running thread, to NEXT,  
6 ##### which must also be running switch_threads(), returning CUR in  
7 ##### NEXT's context.  
8 #####  
9 ##### This function works by assuming that the thread we're switching  
10 ##### into is also running switch_threads(). Thus, all it has to do is  
11 ##### preserve a few registers on the stack, then switch stacks and  
12 ##### restore the registers. As part of switching stacks we record the  
13 ##### current stack pointer in CUR's thread structure.  
14  
15 .globl switch_threads  
16 .func switch_threads  
17 switch_threads:  
18     # Save caller's register state.  
19     #  
20     # Note that the SVR4 ABI allows us to destroy %eax, %ecx, %edx,  
21     # but requires us to preserve %ebx, %ebp, %esi, %edi. See  
22     # [SysV-ABI-386] pages 3-11 and 3-12 for details.  
23     #  
24     # This stack frame must match the one set up by thread_create()  
25     # in size.  
26     pushl %ebx  
27     pushl %ebp  
28     pushl %esi  
29     pushl %edi  
30  
31     # Get offsetof (struct thread, stack).  
32 .globl thread_stack_ofs  
33     mov thread_stack_ofs, %edx  
34  
35     # Save current stack pointer to old thread's stack, if any.  
36     movl SWITCH_CUR(%esp), %eax
```

```

37    movl %esp, (%eax,%edx,1)
38
39    # Restore stack pointer from new thread's stack.
40    movl SWITCH_NEXT(%esp), %ecx
41    movl (%ecx,%edx,1), %esp
42
43    # Restore caller's register state.
44    popl %edi
45    popl %esi
46    popl %ebp
47    popl %ebx
48        ret
49 .endfunc
50
51 .globl switch_entry
52 .func switch_entry
53 switch_entry:
54     # Discard switch_threads() arguments.
55     addl $8, %esp
56
57     # Call thread_schedule_tail(prev).
58     pushl %eax
59 .globl thread_schedule_tail
60     call thread_schedule_tail
61     addl $4, %esp
62
63     # Start thread proper.
64     ret
65 .endfunc

```

分析这段汇编代码，首先将4个寄存器的值压栈保护寄存器状态，这四个寄存器的值是 `switch_threads_frame` 的成员，`switch_threads_frame` 结构的具体定义如下 (`thread/thread.h` 中定义)：

```

1 /* switch_thread()'s stack frame. */
2 struct switch_threads_frame
3 {
4     uint32_t edi;           /* 0: Saved %edi. */
5     uint32_t esi;           /* 4: Saved %esi. */
6     uint32_t ebp;           /* 8: Saved %ebp. */
7     uint32_t ebx;           /* 12: Saved %ebx. */
8     void (*eip) (void);    /* 16: Return address. */
9     struct thread *cur;    /* 20: switch_threads()'s CUR argument. */
10    struct thread *next;   /* 24: switch_threads()'s NEXT argument.
11 */
12 };

```

全局变量 `thread_stack_ofs` 记录线程和栈之间的间隙，下面我们来看线程切换中保存现场的过程。

- 35-36行：先把当前的线程指针放到eax中，并把线程指针保存在相对基址偏移量为edx的地址中
- 40-41：切换到下一个线程的线程栈指针，保存在ecx中，再把这个线程相对基址偏移量edx地址（上一次保存现场的时候存放的）放到esp当中继续执行。

这里ecx, eax起容器的作用，edx指向当前现场保存的地址偏移量。简单来说就是保存当前线程状态，恢复新线程之前保存的线程状态。

由此我们可以看出 `schedule` 函数是先将当前线程放入就绪队列，如果下一个线程和当前线程不一样的话切换到下一个线程。

再看看 `shcedule` 函数最后一行执行的操作，最后一行调用 `thread_schedule_tail` 函数，下面详细分析一下这个函数（`thread/thread.c` 文件中）。

```

1 void thread_schedule_tail (struct thread *prev)
2 {
3     struct thread *cur = running_thread ();
4
5     ASSERT (intr_get_level () == INTR_OFF);
6
7     /* Mark us as running. */
8     cur->status = THREAD_RUNNING;
9
10    /* Start new time slice. */
11    thread_ticks = 0;
12
13 #ifdef USERPROG
14     /* Activate the new address space. */
15     process_activate ();
16 #endif
17
18     /* If the thread we switched from is dying, destroy its struct
19      thread. This must happen late so that thread_exit() doesn't
20      pull out the rug under itself. (We don't free
21      initial_thread because its memory was not obtained via
22      palloc().) */
23     if (prev != NULL && prev->status == THREAD_DYING && prev != initial_thread)
24     {
25         ASSERT (prev != cur);
26         palloc_free_page (prev);
27     }
28 }
```

首先是获得当前线程的的cur(切换之后的线程)，然后将cur的状态改为 `THREAD_RUNNING`，然后 `thread_ticks` 清零开始新的线程切换时间片。然后调用 `diaoyong process_activate` 函数申请新的地址空间，再分析 `process_active` 函数(在 `useruserprog/process.c` 文件中定义)

```

1  /* Sets up the CPU for running user code in the current
2   thread.
3   This function is called on every context switch. */
4  void process_activate (void)
5  {
6      struct thread *t = thread_current ();
7      /* Activate thread's page tables. */
8      pagedir_activate (t->pagedir);
9      /* Set thread's kernel stack for use in processing
10     interrupts. */
11     tss_update ();
12 }

```

关键的就是 `pagedir_activate()` 函数和 `tss_update` 函数，这两个函数分别位于 `userprog/pagedir.c` 和 `userprog/tss.c` 文件中

下面再进入 `pagedir_activate()` 函数中查看。

```

1  /* Loads page directory PD into the CPU's page directory base
2   register. */
3  void pagedir_activate (uint32_t *pd)
4  {
5      if (pd == NULL)
6          pd = init_page_dir;
7
8      /* Store the physical address of the page directory into CR3
9       aka PDBR (page directory base register). This activates our
10      new page tables immediately. See [IA32-v2a] "MOV--Move
11      to/from Control Registers" and [IA32-v3a] 3.7.5 "Base
12      Address of the Page Directory". */
13      asm volatile ("movl %0, %%cr3" : : "r" (vtop (pd)) : "memory");
14 }

```

这个汇编指令将当前线程的页目录指针存储到CR3（页目录表物理内存基地址寄存器）中，也就是说这个函数更新了现在的页目录表

再进入 `tss_update` 函数中：

```

1  /* Sets the ring 0 stack pointer in the TSS to point to the end
2   of the thread stack. */
3  void tss_update (void)
4  {
5      ASSERT (tss != NULL);
6      tss->esp0 = (uint8_t *) thread_current () + PGSIZE;
7 }

```

`tss` 指的是 task state segment，叫任务状态段，任务（进程）切换时的任务现场信息。这里其实是把 TSS 的一个栈指针指向了当前线程栈的尾部，也就是更新了任务现场的信息和状态。

到此 `process_activate` 函数的分析完毕，它做了两件事：

- 更新页目录表
- 更新任务现场信息 (tss)

在继续看 `thread_schedule_tail` 函数的最后4行：

```
1 if (prev != NULL && prev->status == THREAD_DYING && prev != initial_thread)
2 {
3     ASSERT (prev != cur);
4     palloc_free_page (prev);
5 }
```

这里是说如果我们切换的线程状态是 `THREAD_DYING` (代表欲要销毁的线程) 的话，调用 `palloc_free_page` (`thread/palloc.c` 文件中定义)：

```
1 /* Frees the page at PAGE. */
2 void palloc_free_page (void *page)
3 {
4     palloc_free_multiple (page, 1);
5 }
```

简单而言作用就是释放 `PAGE` 参数中的页面

到此，`thread_schedule_tail` 函数分析完毕，其作用就是分配恢复之前执行的状态和现场，如果当前线程死了就清空资源。

`schedule` 函数的作用就是拿下一个线程切换过来继续运行。`thread_yield` 函数的作用是把当前进程放在就绪队列里，调用 `schedule` 切换到下一个进程。

最后返回到最顶层的 `timer_sleep` 函数，他的作用就是在 `ticks` 的时间内 `nei`，如果线程处于 `running` 状态就不断的把它放在就绪队列不让它执行。