

一、实验目的与要求：

掌握进程同步与通信实验

加深对进程同步与通信操作的直观认识；

掌握 Linux 操作系统的进程、线程机制和编程接口；

掌握 Linux 操作系统的进程和线程间的同步和通信机制；

掌握经典同步问题的编程方法；

可以使用 Linux 或其它 Unix 类操作系统；

学习该操作系统提供的进程、线程创建的函数使用方法；

学习该操作系统提供的共享内存、管道、消息队列的通信机制；

利用该操作系统提供的进程间同步的信号量。

硬件：桌面 PC

软件：Linux 或其他操作系统

二、方法、步骤：(说明程序相关的算法原理或知识内容，程序设计的思路和方法，可以用流程图表述，程序主要数据结构的设计、主要函数之间的调用关系等)

操作部分（参考）：

- 1) 借助 google 工具查找资料，学习使用 Linux 进程间通信：管道、消息队列、共享内存；
- 2) 设计编写以下程序，着重考虑其同步问题：
 - a) 一个程序（进程）从客户端读入按键信息，一次将“一整行”按键信息保存到一个共享存储的缓冲区内并等待读取进程将数据读走，不断重复上面的操作；
 - b) 另一个程序（进程）生成两个进程，用于显示缓冲区内的信息，这两个进程并发读取缓冲区信息后将缓冲区清空（一个进程的两次显示操作之间可以加入适当的时延以便于观察）。
 - c) 在两个独立的终端窗口上分别运行上述两个程序，展示其同步与通信功能，要求一次只有一个任务在操作缓冲区。
 - d) 运行程序，记录操作过程的截屏并给出文字说明。要求使用 posix 信号量来完成这里的生产者和消费者的同步关系。

实验报告要求：

- 1) 按学校统一格式
- 2) 需要给出具体命令和自行编写的程序的源代码
- 3) 程序的设计需要给出设计思路或流程框图
- 4) 实验操作的截图需要有必要的说明文字

三. 实验过程及内容: (对程序代码进行说明和分析, 越详细越好, 代码排版要整齐, 可读性要高)

操作部分:

1. 借助 google 工具查找资料, 学习使用 Linux 进程间通信: 管道、消息队列、共享内存;

通过 google 搜索得到:

linux 使用的进程间通信方式有如下六种方法:

1. 管道 (pipe) ,流管道(s pipe)和有名管道 (FIFO)
2. 信号 (signal)
3. 消息队列
4. 共享内存
5. 信号量
6. 套接字 (socket)

在 Linux 中有查看资源使用情况的方法: (刚启动 ubuntu 所以没使用到信号量等资源)

```
ljt@ljt:~$ ipcs
----- 消息队列 -----
键          msqid      拥有者  权限      已用字节数  消息
----- 共享内存段 -----
键          shmid      拥有者  权限      字节        连接数  状态
0x00000000  753664    ljt     600       524288      2       目标
0x00000000  851969    ljt     600       16777216    2       目标
0x00000000  655362    ljt     600       524288      2       目标
0x00000000  1572867   ljt     600       524288      2       目标
0x00000000  1310724   ljt     600       524288      2       目标
0x00000000  983045    ljt     600       524288      2       目标
0x00000000  1146886   ljt     600       524288      2       目标
0x00000000  1179655   ljt     600       67108864    2       目标
0x00000000  1409032   ljt     600       524288      2       目标
----- 信号量数组 -----
键          semid      拥有者  权限      nsems
```

其中:

第一列就是共享内存的 key; 第二列是共享内存的编号 shmid;

第三列就是创建的用户 owner; 第四列就是权限 perms;

第五列为创建的大小 bytes; 第六列为连接到共享内存的进程数 nattach;

具体的用法总结如下:

- 1、显示所有的 IPC 设施# ipcs -a
- 2、显示所有的消息队列 Message Queue# ipcs -q
- 3、显示所有的信号量# ipcs -s
- 4、显示所有的共享内存# ipcs -m
- 5、显示 IPC 设施的详细信息# ipcs -q -i id
- 6、显示 IPC 设施的限制大小# ipcs -m -l
- 7、显示 IPC 设施的权限关系# ipcs -c
- 8、显示最近访问过 IPC 设施的进程 ID。# ipcs -p
- 9、显示 IPC 设施的最后操作时间# ipcs -t
- 10、显示 IPC 设施的当前状态# ipcs -u

在这里我们详细介绍管道, 消息队列, 信号量和共享内存四种。

管道:

管道这种通讯方式有两种限制，一是半双工的通信，数据只能单向流动，二是只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

流管道 `s pipe`: 去除了第一种限制,可以双向传输。

管道可用于具有亲缘关系进程间的通信，命名管道: `name pipe` 克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信；

消息队列:

消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。消息队列是消息的链接表，包括 Posix 消息队列 system V 消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。

信号量:

信号量用于进程间传递信号的一个整数值。在信号量上只有三种操作可以进行，初始化，递减和增加，这三种操作都是原子操作。递减的操作可以用于阻塞一个进程，增加操作可以用于接触阻塞一个进程。因此信号量也可以称之为一个计数器。

共享内存:

共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量，配合使用，来实现进程间的同步和通信。

使得多个进程可以访问同一块内存空间，是最快的可用 IPC 形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。

这三种通讯方式的 **优缺点**如下:

管道: 速度慢，容量有限，只有父子进程能通讯

消息队列: 容量受到系统限制，且要注意第一次读的时候，要考虑上一次没有读完数据的问题

信号量: 不能传递复杂消息，只能用来同步

共享内存区: 能够很容易控制容量，速度快，但要保持同步，比如一个进程在写的时候，另一个进程要注意读写的问题，相当于线程中的线程安全，当然，共享内存区同样可以用作线程间通讯，不过没这个必要，线程间本来就已经共享了同一进程内的一块内存

管道:

管道分为匿名管道和 FIFO 两种。

1) FIFO

创建 “`mkfifo -m XXX fifoname`” 其中 `xxx` 是访问权限 (8 进制) `fifoname` 是管道名。创建后可以用 `ls -l` 查看这个管道文件。

```

ljt@ljt: ~
ljt@ljt:~$ mkfifo -m 600 my_frist_fifo
ljt@ljt:~$ ls -l
总用量 60
drwxr-xr-x 23 root root 4096 12月 2 22:55 anaconda3
-rw-r--r-- 1 ljt ljt 8980 12月 2 22:13 examples.desktop
prw----- 1 ljt ljt 0 4月 8 10:52 my_frist_fifo
drwxrwxr-x 3 ljt ljt 12288 3月 26 10:14 xv6-public
drwxr-xr-x 2 ljt ljt 4096 12月 2 22:18 公共的
drwxr-xr-x 2 ljt ljt 4096 12月 2 22:18 模板
drwxr-xr-x 2 ljt ljt 4096 12月 2 22:18 视频
drwxr-xr-x 2 ljt ljt 4096 12月 2 22:18 图片
drwxr-xr-x 3 ljt ljt 4096 12月 21 15:26 文档
drwxr-xr-x 7 ljt ljt 4096 3月 25 20:43 下载
drwxr-xr-x 2 ljt ljt 4096 12月 2 22:18 音乐
drwxr-xr-x 5 ljt ljt 4096 12月 21 14:56 桌面
ljt@ljt:~$

```

可见管道文件在 `ls -l` 命令的输出中，第一列为“p”表示管道文件，以区别普通文件、目录和设备文件等。

2) 匿名管道

匿名管道仅存在于父子进程间，无法在任意两个进程间使用。通常先创建管道，然后再创建父子进程。

创建管道前线需要声明两个文件描述符，然后再用 `pipe()` 创建。

`int fds[2];` 声明两个文件描述符（整数）

`pipe(fds);` 由 `pipe()` 创建管道

创建后，第一个文件描述符（本例中的 `fds[0]` 对应读端，`fds[1]` 对应写端）。

管道的读写和普通文件的读写相同，使用 `read()` 和 `write()` 完成。

参考 google 上的代码：（为了减少报告行数，在将 `include` 放到一行中）

```

#include<sys/types.h> #include<sys/stat.h> #include<unistd.h> #include<fcntl.h>
#include<stdio.h> #include<stdlib.h> #include<errno.h> #include<string.h> #include<signal.h>
#define ERR_EXIT(m) \
do { \
    perror(m); \
    exit(EXIT_FAILURE); \
} while(0)
int main(int argc, char *argv[]){
    int pipefd[2];
    if(pipe(pipefd) == -1) ERR_EXIT("pipe error");
    pid_t pid; pid = fork();
    if(pid == -1) ERR_EXIT("fork error");
    if(pid == 0) {
        close(pipefd[0]); write(pipefd[1], "hello", 5);
        close(pipefd[1]); exit(EXIT_SUCCESS);
    }
    close(pipefd[1]); char buf[10] = {0};
    read(pipefd[0], buf, 10); printf("buf=%s\n", buf);
    return 0;
}

```

父进程调用 `pipe` 开辟管道，调用 `fork` 创建子进程，那么子进程也有两个文件描述符指向同一管道。子进程可以往管道里写，父进程可以从管道里读，数据从写端流入从读端流出，这样就实现了进程间通信。

结果如下：

```

ljt@ljt:~$ gedit pip.cpp
ljt@ljt:~$ g++ pip.cpp -o pip
ljt@ljt:~$ ./pip
buf=hello
ljt@ljt:~$

```

另外在 shell 中，使用在两个命令之间用 “|” 表示匿名管道，例如 ls|more，就是将 ls 的输出通过管道作为 more 的输入。

消息队列：消息队列使用邮箱来存储消息，ipcs -q 可以查看到不同的有效以及上面的消息数量。

创建 邮箱的方法为：

```
msgqueue_id = msgget(key, IPC_CREAT|0660)
```

其中 key 可以从 ftok() 函数来生成，ftok() 将一个文件路径名转换成一个 key 值。

在邮箱创建之后，想要用这个邮箱的进程需要执行类似下面的代码：

```
msgqueue_id = msgget(key, IPC_CREAT|0660)
```

创建消息队列代码如下：

```

#include <stdio.h> #include <string.h> #include <sys/ipc.h> #include <sys/msg.h>
int main(){
    int msgid;
    msgid = msgget(IPC_PRIVATE, 0600);
    if(msgid < 0){ perror("msgget"); return 1; }
    printf("%d\n", msgid);
    return 0;
}

```

实验结果如下：

```

ljt@ljt:~$ ipcs -q
----- 消息队列 -----
键          msqid      拥有者   权限    已用字节数 消息
ljt@ljt:~$ ./queue
32768
ljt@ljt:~$ ipcs -q
----- 消息队列 -----
键          msqid      拥有者   权限    已用字节数 消息
0x00000000 32768      ljt      600      0          0

```

在 **发送** 消息之前，需要构建消息。消息队列中的消息是有格式的：

```

struct mymsgbuf {
    long mtype;
    char mtext[MAX_SEND_SIZE];
};

```

其中 mtype 成员指出消息类型，使用整数数字来表示类型编码。

发送消息到指定邮箱：

```
msgsnd(qid, (struct msgbuf *)qbuf, strlen(qbuf->mtext)+1, 0)
```

代码如下所示：

```

#include <stdio.h> #include <string.h> #include <sys/ipc.h> #include <sys/msg.h> #include
<stdlib.h> #define MTEXTSIZE 10
int main(int argc, char* argv[]){
    int msgid;
    struct msgbuf{ long mtype; char mtext[MTEXTSIZE]; }mbuf;

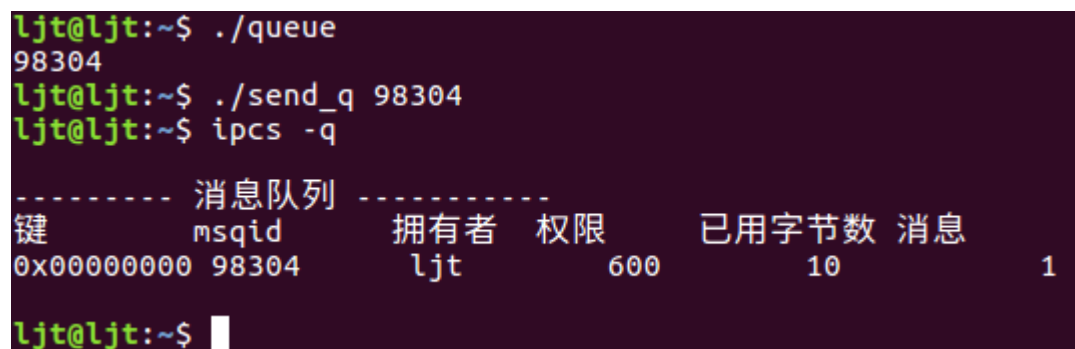
```

```

if(argc != 2){
printf("wrong argc"); return 1;
}
msgid = atoi(argv[1]);
mbuf.mtype = 777;
memset(mbuf.mtext, 0, sizeof(mbuf.mtext));
mbuf.mtext[0] = 'A';
if(msgsnd(msgid, &mbuf, MTEXTSIZE, 0) != 0){
perror("msgsnd"); return 1;
}
return 0;
}

```

实验结果如下：



```

ljt@ljt:~$ ./queue
98304
ljt@ljt:~$ ./send_q 98304
ljt@ljt:~$ ipcs -q
----- 消息队列 -----
键          msqid      拥有者   权限      已用字节数  消息
0x00000000  98304      ljt      600        10         1
ljt@ljt:~$

```

此时消息队列中的消息有一条了。

从指定邮箱 **接收** 消息：

```
msggrcv(qid, (struct msgbuf *)qbuf, MAX_SEND_SIZE, type, 0);
```

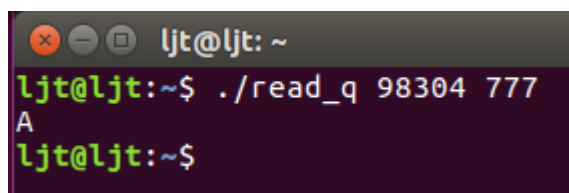
实例代码如下：

```

#include <stdio.h> #include <string.h> #include <sys/ipc.h> #include <sys/msg.h> #include
<stdlib.h> #define MTEXTSIZE 10
int main(int argc, char* argv[]){
int msgid, msgtype;
struct msgbuf{ long mtype; char mtext[MTEXTSIZE]; }mbuf;
if(argc != 3){ printf("wrong argc"); return 1; }
msgid = atoi(argv[1]);
msgtype = atoi(argv[2]);
if(msggrcv(msgid, &mbuf, MTEXTSIZE, msgtype, 0) <= 0){
perror("msggrcv"); return 1; }
printf("%c\n", mbuf.mtext[0]);
return 0;
}

```

实验结果如下：

A terminal window with a dark background. The prompt is 'ljt@ljt: ~'. The command './read_q 98304 777' is entered and executed. The output 'A' is displayed on the next line. The prompt returns to 'ljt@ljt: ~\$'.

撤销 邮箱:
`msgctl(qid, IPC_RMID, 0);`
实例代码如下:

```
#include <stdio.h> #include <string.h> #include <sys/ipc.h> #include <sys/msg.h>
#include <stdlib.h>
int main(int argc, char* argv[]){
    int msgid;
    msgid_ds mds;
    if(argc != 2){
        printf("wrong argv\n");
        return 1;
    }
    msgid = atoi(argv[1]);
    if(msgctl(msgid, IPC_RMID, &mds) != 0){
        perror("msgctl");
        return 1;
    }
    return 0;
}
```

信号量: 信号量的工作原理:
由于信号量只能进行两种操作等待和发送信号, 即 P(sv)和 V(sv),他们的行为是这样的:
P(sv): 如果 sv 的值大于零, 就给它减 1; 如果它的值为零, 就挂起该进程的
V(sv): 如果有其他进程因等待 sv 而被挂起, 就让它恢复运行, 如果没有进程因等待 sv 而挂起, 就给它加 1.

代码演示:

```
#include <stdio.h> #include <string.h> #include <sys/ipc.h> #include <sys/sem.h>
#define NSEMS 16
int main(){
    int semid;
    unsigned short semun_array[NSEMS];
    int i;
    semid = semget(IPC_PRIVATE, NSEMS, 0600);
    if(semid < 0){ perror("semget"); return 1; }
    for(i = 0; i < NSEMS; ++i){ semun_array[i] = 1; }
    if(semctl(semid, 1000, SETALL, &semun_array) != 0){
        perror("semctl"); return 1;
    }
    printf("semid:%d\n", semid);
    return 0;
}
```

```
}
```

结果如下：

```
ljt@ljt:~$ gedit sig.cpp
ljt@ljt:~$ g++ sig.cpp -o sig
ljt@ljt:~$ ./sig
semid:0
ljt@ljt:~$ ipcs -s

----- 信号量数组 -----
键          semid      拥有者    权限      nsems
0x00000000  0          ljt       600       16

ljt@ljt:~$
```

共享内存区：与下题雷同便不重复叙述。

2. 设计编写以下程序，着重考虑其同步问题：

- 一个程序（进程）从客户端读入按键信息，一次将“一整行”按键信息保存到一个共享存储的缓冲区内并等待读取进程将数据读走，不断重复上面的操作；
- 另一个程序（进程）生成两个进程，用于显示缓冲区内的信息，这两个进程并发读取缓冲区信息后将缓冲区清空（一个进程的两次显示操作之间可以加入适当的时延以便于观察）。
- 在两个独立的终端窗口上分别运行上述两个程序，展示其同步与通信功能，要求一次只有一个任务在操作缓冲区。
- 运行程序，记录操作过程的截屏并给出文字说名。
 - 要求使用 `posix` 信号量来完成这里的生产者和消费者的同步关系。

首先按照要求一步一步实现：

1. 客户端（productor）读取按键信息写入到共享内存上：

首先需要设置共享内存块的信息。

```
1  #include <fcntl.h>
2  #include <sys/stat.h>
3  #include <semaphore.h>
4  #define LINE_SIZE 256
5  #define NUM_LINE 16
6  //用于创建信号量时识别的ID
7  const char * queue_mutex="queue_mutex";
8  const char * queue_empty="queue_empty";
9  const char * queue_full="queue_full";
10 struct shared_mem_st{//公用缓冲区
11     char buffer[NUM_LINE][LINE_SIZE]; //缓冲数据组
12     int line_write;//读写指针
13     int line_read;//读写指针
14 };
15
```

在 `productor` 中，需要创建一个新的共享内存，上面的头文件只表明声明并没有创建。创建新的共享内存并映射到本进程的进程空间。并对创建失败进行输出报错处理。设置好共享内存区域的大小。


```

1  #include <stdio.h>
2  #include <sys/ipc.h>
3  #include <sys/shm.h>
4  #include <semaphore.h>
5  #include <fcntl.h>
6  #include <stdlib.h>
7  #include <unistd.h>
8  #include <string.h>
9  #include "shm_com_sem.h"
10 int main(void){
11     void *shared_memory=(void *)0; //共享内存 缓冲区指针
12     struct shared_mem_st *shared_stuff;
13     char key_line[256];
14     int stmid; //共享内存id
15     sem_t *sem_queue,*sem_queue_empty,*sem_queue_full; //访问共享内存的信号量指针
16     stmid=shmget((key_t)1234,sizeof(struct shared_mem_st),0666|IPC_CREAT);
17     //创建一个新的共享内存
18     if(stmid==-1) {
19         perror("shmget failed");
20         exit(1);
21     }
22     if((shared_memory = shmat(stmid,0,0))<(void *)0){
23         //若共享内存区映射到本进程的进程空间失败
24         perror("shmat failed"); exit(1);
25     }
26 }

```

致此都只是共享内存的申请，申请成功后对获得 shared_memory 的变量，对应的是共享内存的结构，我们将它强制转化成我们自定义的 struct 结构，以方便分配数组，并初始化读写指针。

```

24     shared_stuff=(struct shared_mem_st *)shared_memory;
25     //将缓冲区指针转换为shared_mem_st格式
26     shared_stuff->line_write=0; //读写指针初始化
27     shared_stuff->line_read=0;

```

然后是创建三个信号量，分别在其中起到控制共享内存的输入输出的作用，加入共享内存满了以后，producer 不在允许输入，亦或是但 customer 在读取的时候，producer 不予许输入。

```

27     sem_queue=sem_open("queue_mutex",O_CREAT,0644,1);
28     sem_queue_empty=sem_open("queue_empty",O_CREAT,0644,1);
29     sem_queue_full=sem_open("queue_full",O_CREAT,0644,1); //创建三个信号量

```

到这里所有的准备工作都做好了（product 里面的共享内存创建映射到进程空间中，共享内存的转化等操作）。

接下来是设置循环输入字符串，等待信号量输入后，如果信号量正常。则获取键盘输入，并将其放置到共享内存空间中。

```

32     while(running) {
33         sleep(1);
34         sem_wait(sem_queue_empty);
35         sem_wait(sem_queue); //等待信号量输入
36         printf("Enter some text:");
37         fgets(b,BUFSIZ,stdin);
38         strncpy(shared_stuff->buffer[shared_stuff->line_write],b,LINE_SIZE);
39         shared_stuff->line_write=(shared_stuff->line_write+1)%NUM_LINE; //写指针增加
40         if(strncmp(b,"end",3)==0) //如果输入为end则退出 {
41             running=0;
42         }
43         sem_post(sem_queue); //发送信号量
44         sem_post(sem_queue_full);
45     } //为了防止结束后进程阻塞，所以再发送一次信号量和内容

```

其中为了防止结束后进程阻塞，所以再发送一次信号量和内容

```

46     sem_post(sem_queue); //发送信号量
47     sem_post(sem_queue_full);

```

之后便是解除映射和删除共享内存空间：

```

50     if(shmdt(shared_memory)==-1){//解除映射
51         fprintf(stderr,"shmdt failed\n");
52         exit(EXIT_FAILURE);
53     } //删除共享内存区
54     if(shmctl(stmid,IPC_RMID,0)==-1){
55         fprintf(stderr,"shmctl failed\n");
56         exit(EXIT_FAILURE);
57     }

```

2. 另一个 (customer) 产生两个进程并发读取缓冲区信息并清空:

就如同网络连接中的 client 和 server 一样, producer 如同 server 一般, 发送数据, 而 customer 就如同 client 一样接受数据并将其显示出来, 所以前面的是获取到 producer 创建的共享内存地址, 将其映射到自身进程空间。

```

1  #include <stdio.h>
2  #include <sys/ipc.h>
3  #include <sys/shm.h>
4  #include <semaphore.h>
5  #include <fcntl.h>
6  #include <stdlib.h>
7  #include <unistd.h>
8  #include <string.h>
9  #include "shm_com_sem.h"
10 int main(void){
11     void * shared_memory=(void *)0;
12     struct shared_mem_st *shared_stuff;
13     int stmid;
14     int num_read;
15     pid_t fork_result;
16     sem_t *sem_queue,*sem_queue_empty,*sem_queue_full;
17     stmid=shmget((key_t)1234,sizeof(struct shared_mem_st),0666|IPC_CREAT);
18     //获得已创建共享内存
19     if(stmid==-1){
20         perror("shmget failed"); exit(1);
21     }
22     if((shared_memory = shmat(stmid,0,0))<(void *)0){
23         //若共享内存区映射到本进程的进程空间失败
24         perror("shmat failed");
25         exit(1);
26     }

```

再者如同 producer 一般, 需要将 share space 进行结构转化, 转化成相同的 struct。并且获取相同的三个信号量来达成 **同步** 的效果。

```

25     shared_stuff=(struct shared_mem_st *)shared_memory;
26     //获取三个信号量
27     sem_queue=sem_open("queue_mutex",0);
28     sem_queue_empty=sem_open("queue_empty",0);
29     sem_queue_full=sem_open("queue_full",0);

```

在这里与 producer 不同的是, customer 需要创建一个子进程来并发的读取数据, 并且我们可以查看其中的共享内存的内容。

首先父进程创建一个子进程。

```

30     //创建两个进程
31     fork_result=fork();
32     if(fork_result==-1)
33         fprintf(stderr,"fork failed\n");
34     int running=1;

```

其次是一个跳转语句将父子进程分别放在两个 while 循环中。

```

35     if(fork_result==0)//子进程 {
36         while(running) {
46             sem_unlink(queue_mutex);
47             sem_unlink(queue_empty);
48             sem_unlink(queue_full);
49         }
50     else//父进程
51     { while(running) {
65         waitpid(fork_result,NULL,0);

```

首先在子进程中，进行着信号量的等待和读取，在其中加入 sleep，让其有一个显示的效果。并做出判断语句，如果接收到的字符串含有 quit，则退出循环：

```

36     while(running) {
37         sem_wait(sem_queue_full);
38         sem_wait(sem_queue);//等待信号量
39         sleep(2);
40         printf("child pid is %d,you wrote:%s\n",getpid(),shared_stuff->buffer[
shared_stuff->line_read]);//输出进程号和消息内容
41         if(strncmp(shared_stuff->buffer[shared_stuff->line_read],"quit",4)==0)
//如果为end则退出 {running=0;}
42         shared_stuff->line_read=(shared_stuff->line_read+1)%NUM_LINE;//读指针改变
43         sem_post(sem_queue);//发送信号量
44         sem_post(sem_queue_empty);
45     }
46     sem_unlink(queue_mutex);
47     sem_unlink(queue_empty);
48     sem_unlink(queue_full);

```

在父进程中也是一样的语句，由于是要实现并发的条件，所以代码段是相同的。

```

51     { while(running) {
52         sem_wait(sem_queue_full);
53         sem_wait(sem_queue);//等待信号量
54         sleep(2);
55         printf("parent pid is %d,you wrote:%s\n",getpid(),shared_stuff->buffer[
shared_stuff->line_read]);//输出进程号和消息内容
56         if(strncmp(shared_stuff->buffer[shared_stuff->line_read],"quit",4)==0)
//如果为end则退出 { running=0;}
57         shared_stuff->line_read=(shared_stuff->line_read+1)%NUM_LINE;//读指针改变
58         sem_post(sem_queue);//发送信号量
59         sem_post(sem_queue_empty);
60     }
61     sem_unlink(queue_mutex);
62     sem_unlink(queue_empty);
63     sem_unlink(queue_full);

```

之后是退出了：

```

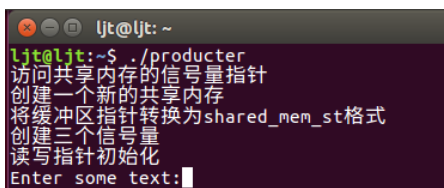
65     waitpid(fork_result,NULL,0);
66     exit(EXIT_SUCCESS);
67 }

```

3. 两个独立终端运行：

在 Linux 中运行两个程序：

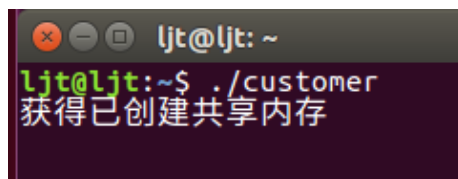
开始时：



```

ljt@ljt: ~
ljt@ljt:~$ ./producer
访问共享内存的信号量指针
创建一个新的共享内存
将缓冲区指针转换为shared_mem_st格式
创建三个信号量
读写指针初始化
Enter some text:

```



```

ljt@ljt: ~
ljt@ljt:~$ ./customer
获得已创建共享内存

```

Productor 初始化完所有的步骤后，等待键盘输入。而在 customer 中，获取到了 productor 建立的共享内存。

我在 productor 中输入相应的字符结果如下：

```

ljt@ljt: ~
ljt@ljt:~$ ./producter
访问共享内存的信号量指针
创建一个新的共享内存
将缓冲区指针转换为shared_mem_st格式
创建三个信号量
读写指针初始化
Enter some text:ljt
Enter some text:

```

输入后，在 customer 中显示的是：

```

ljt@ljt: ~
ljt@ljt:~$ ./customer
获得已创建共享内存
parent pid is 2351,you wrote:ljt

child pid is 2352,you wrote:

```

由于最开始的时候是父进程刚开始创建子进程，获取到了共享内存中的数据。再次输入数据有：

```

ljt@ljt:~$ ./producter
访问共享内存的信号量指针
创建一个新的共享内存
将缓冲区指针转换为shared_mem_st格式
创建三个信号量
读写指针初始化
Enter some text:lin
Enter some text:jia
Enter some text:tao
nEnter some textnihao
Enter some text:

```

```

ljt@ljt: ~
ljt@ljt:~$ ./customer
获得已创建共享内存
parent pid is 2826,you wrote:lin

child pid is 2827,you wrote:jia

parent pid is 2826,you wrote:tao

```

```

ljt@ljt:~$ ./producter
访问共享内存的信号量指针
创建一个新的共享内存
将缓冲区指针转换为shared_mem_st格式
创建三个信号量
读写指针初始化
Enter some text:lin
Enter some text:jia
Enter some text:tao
Enter some text:quit
ljt@ljt:~$

```

```

ljt@ljt: ~
ljt@ljt:~$ ./customer
获得已创建共享内存
child pid is 2513,you wrote:lin

child pid is 2513,you wrote:jia

child pid is 2513,you wrote:tao

child pid is 2513,you wrote:quit
parent pid is 2512,you wrote:quit

ljt@ljt:~$

```

当我输入 quit 的时候，两方都退出，两个线程都退出程序。

代码如下：

头文件 `shm_com_sem.h`

```

#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
#define LINE_SIZE 256
#define NUM_LINE 16
const char * queue_mutex="queue_mutex";
const char * queue_empty="queue_empty";
const char * queue_full="queue_full";
struct shared_mem_st{///公用缓冲区

```

```

char buffer[NUM_LINE][LINE_SIZE];//缓冲数据组
int line_write;//读写指针
int line_read;
};

```

producer.cpp

```

#include <stdio.h>  #include <sys/ipc.h>
#include <sys/shm.h>  #include <semaphore.h>
#include <fcntl.h>  #include <stdlib.h>
#include <unistd.h>  #include <string.h>
#include "shm_com_sem.h"
int main(void){
    void *shared_memory=(void *)0;//共享内存 缓冲区指针
    struct shared_mem_st *shared_stuff;
    char key_line[256];
    int stmid;//共享内存 id
    sem_t *sem_queue,*sem_queue_empty,*sem_queue_full;//访问共享内存的信号量指针
    stmid=shmget((key_t)1234,sizeof(struct shared_mem_st),0666|IPC_CREAT);//创建一个新的共享内存
    if(stmid==-1)    {
        perror("shmget failed");
        exit(1);
    }
    if((shared_memory = shmat(stmid,0,0))<(void *)0){ //若共享内存区映射到本进程的进程空间失败
        perror("shmat failed");  exit(1);
    }
    shared_stuff=(struct shared_mem_st *)shared_memory;//将缓冲区指针转换为shared_mem_st 格式
    sem_queue=sem_open("queue_mutex",O_CREAT,0644,1);
    sem_queue_empty=sem_open("queue_empty",O_CREAT,0644,1);
    sem_queue_full=sem_open("queue_full",O_CREAT,0644,1);//创建三个信号量
    shared_stuff->line_write=0; //读写指针初始化
    shared_stuff->line_read=0;
    char b[BUFSIZ];
    int running=1;
    while(running){
        sleep(1);
        sem_wait(sem_queue_empty);
        sem_wait(sem_queue);//等待信号量输入
        printf("Enter some text:");
        fgets(b,BUFSIZ,stdin);
        strncpy(shared_stuff->buffer[shared_stuff->line_write],b,LINE_SIZE);
        shared_stuff->line_write=(shared_stuff->line_write+1)%NUM_LINE;//写指针增
    }
}

```

加

```
    if(strncmp(b,"end",3)==0)//如果输入为 end 则退出  {
        running=0;
    }
    sem_post(sem_queue);//发送信号量
    sem_post(sem_queue_full);
} //为了防止结束后进程阻塞，所以再发送一次信号量和内容
sem_post(sem_queue);//发送信号量
sem_post(sem_queue_full);
strncpy(shared_stuff->buffer[shared_stuff->line_write],b,LINE_SIZE);
shared_stuff->line_write=(shared_stuff->line_write+1)%NUM_LINE;//写指针增加
if(shmdt(shared_memory)==-1)//解除映射 {
    fprintf(stderr,"shmdt failed\n");
    exit(EXIT_FAILURE);
} //删除共享内存区
if(shmctl(stmid,IPC_RMID,0)==-1) {
    fprintf(stderr,"shmctl failed\n");
    exit(EXIT_FAILURE);
}
}
```

customer.cpp

```
#include <stdio.h>  #include <sys/ipc.h>
#include <sys/shm.h>  #include <semaphore.h>
#include <fcntl.h>  #include <stdlib.h>
#include <unistd.h>  #include <string.h>
#include "shm_com_sem.h"
int main(void){
    void * shared_memory=(void *)0;
    struct shared_mem_st *shared_stuff;
    int stmid;
    int num_read;
    pid_t fork_result;
    sem_t *sem_queue,*sem_queue_empty,*sem_queue_full;
    stmid=shmget((key_t)1234,sizeof(struct shared_mem_st),0666|IPC_CREAT);//获得已
创建共享内存
    if(stmid==-1) {
        perror("shmget failed");  exit(1);
    }
    if((shared_memory = shmat(stmid,0,0))<(void *)0){ //若共享内存区映射到本进程的
进程空间失败
        perror("shmat failed");
        exit(1);
    }
    shared_stuff=(struct shared_mem_st *)shared_memory;
```

```

//获取三个信号量
sem_queue=sem_open("queue_mutex",0);
sem_queue_empty=sem_open("queue_empty",0);
sem_queue_full=sem_open("queue_full",0);
//创建两个进程
fork_result=fork();
if(fork_result==-1)
    fprintf(stderr,"fork failed\n");
int running=1;
if(fork_result==0)//子进程 {
    while(running) {
        sem_wait(sem_queue_full);
        sem_wait(sem_queue);//等待信号量
        sleep(2);
        printf("child pid is %d,you wrote:%s\n",
getpid(),shared_stuff->buffer[shared_stuff->line_read]);//输出进程号和消息内容
        if(strncmp(shared_stuff->buffer[shared_stuff->line_read],"end",3)==0)// 如果 为
end 则退出 {running=0;}
        shared_stuff->line_read=(shared_stuff->line_read+1)%NUM_LINE;//读指针
改变

        sem_post(sem_queue);//发送信号量
        sem_post(sem_queue_empty);
    }
    sem_unlink(queue_mutex);
    sem_unlink(queue_empty);
    sem_unlink(queue_full);
}
else//父进程
{ while(running){
    sem_wait(sem_queue_full);
    sem_wait(sem_queue);//等待信号量
    sleep(2);
    printf("parent pid is %d,you wrote:%s\n",getpid(),
shared_stuff->buffer[shared_stuff->line_read]);//输出进程号和消息内容
    if(strncmp(shared_stuff->buffer[shared_stuff->line_read],"end",3)==0)// 如果
为 end 则退出{ running=0;}
    shared_stuff->line_read=(shared_stuff->line_read+1)%NUM_LINE;//读指针
改变

    sem_post(sem_queue);//发送信号量
    sem_post(sem_queue_empty);
}
    sem_unlink(queue_mutex);
    sem_unlink(queue_empty);
    sem_unlink(queue_full);
}

```

```
}  
waitpid(fork_result,NULL,0);  
exit(EXIT_SUCCESS);  
}
```

四、实验结论：（提供运行结果，对结果进行探讨、分析、评价，并提出结论性意见和改进想法）

父进程调用 pipe 开辟管道,结果如下：

```
ljt@ljt:~$ gedit pip.cpp  
ljt@ljt:~$ g++ pip.cpp -o pip  
ljt@ljt:~$ ./pip  
buf=hello  
ljt@ljt:~$
```

消息队列：

创建消息队列

```
ljt@ljt:~$ ipcs -q  
----- 消息队列 -----  
键          msqid      拥有者   权限      已用字节数 消息  
  
ljt@ljt:~$ ./queue  
32768  
ljt@ljt:~$ ipcs -q  
----- 消息队列 -----  
键          msqid      拥有者   权限      已用字节数 消息  
0x00000000 32768      ljt      600       0          0
```

发送 消息

```
ljt@ljt:~$ ./queue  
98304  
ljt@ljt:~$ ./send_q 98304  
ljt@ljt:~$ ipcs -q  
----- 消息队列 -----  
键          msqid      拥有者   权限      已用字节数 消息  
0x00000000 98304      ljt      600      10         1  
  
ljt@ljt:~$
```

此时消息队列中的消息有一条了。

从指定邮箱**接收** 消息：

实验结果如下：


```
ljt@ljt: ~  
ljt@ljt:~$ ./read_q 98304 777  
A  
ljt@ljt:~$
```

开始时:

```
ljt@ljt: ~  
ljt@ljt:~$ ./producer  
访问共享内存的信号量指针  
创建一个新的共享内存  
将缓冲区指针转换为shared_mem_st格式  
创建三个信号量  
读写指针初始化  
Enter some text:ljt
```

```
ljt@ljt: ~  
ljt@ljt:~$ ./customer  
获得已创建共享内存
```

Producer 初始化完所有的步骤后, 等待键盘输入。而在 customer 中, 获取到了 producer 建立的共享内存。

我在 producer 中输入相应的字符结果如下:

```
ljt@ljt: ~  
ljt@ljt:~$ ./producer  
访问共享内存的信号量指针  
创建一个新的共享内存  
将缓冲区指针转换为shared_mem_st格式  
创建三个信号量  
读写指针初始化  
Enter some text:ljt  
Enter some text:
```

输入后, 在 customer 中显示的是:

```
ljt@ljt: ~  
ljt@ljt:~$ ./customer  
获得已创建共享内存  
parent pid is 2351,you wrote:ljt  
child pid is 2352,you wrote:  
ljt
```

由于最开始的时候是父进程刚开始创建子进程, 获取到了共享内存中的数据。再次输入数据有:

```
ljt@ljt:~$ ./producer  
访问共享内存的信号量指针  
创建一个新的共享内存  
将缓冲区指针转换为shared_mem_st格式  
创建三个信号量  
读写指针初始化  
Enter some text:lin  
Enter some text:jia  
Enter some text:tao  
nEnter some textnihao  
Enter some text:
```

```
ljt@ljt: ~  
ljt@ljt:~$ ./customer  
获得已创建共享内存  
parent pid is 2826,you wrote:lin  
child pid is 2827,you wrote:jia  
parent pid is 2826,you wrote:tao
```

```
ljt@ljt:~$ ./producer  
访问共享内存的信号量指针  
创建一个新的共享内存  
将缓冲区指针转换为shared_mem_st格式  
创建三个信号量  
读写指针初始化  
Enter some text:lin  
Enter some text:jia  
Enter some text:tao  
Enter some text:quit  
ljt@ljt:~$
```

五、实验体会：（根据自己情况填写）

通过本次实验，学习到了共享进程和信号量的使用，通过实际的案例来加深对共享内存的理解。通过 Google 搜索得到了各种资料，学习到了各种进程之间的通信方法以及其中的 C++ 实现。尤其是信号量和管道的使用，虽然其优点并不多，但是在编程上和使用上相对于共享内存再说要方便许多，没有共享内存那么复杂的功能，所以使用起来比较快捷。使用 `ipcs` 查看共享内存的是 `linux` 的一个十分方便的命令，对系统资源的状态有直观的显示。

注：“指导教师批阅意见”栏请单独放置一页

指导教师批阅意见：

成绩评定：

指导教师签字：

年 月 日

备注：