一、实验目的与要求：

综合利用进程控制的相关知识，结合对 shell 功能的和进程间通信手段的认知，编写简易 shell 程序，加深操作系统的进程控制和 shell 接口的认识。可以使用 Linux 或其它 Unix 类操作系统；全面实践进程控制、进程间通信的手段、处理机调度；编写简易 shell 程序。

2.设计编写以下程序，着重考虑其同步问题：

2.1 分析题目与学习准备
2.1.1分析题目，画出进程关系
2.1.2实现共享内存的基本步骤：创建，关联，分离
2.2头文件shm_com_sem.h
2.3 producer.c
2.4 customer.c
2.5运行结果分析

3.设计简单的shell程序

3.1 shell的基本功能分析
3.2预计myshell实现的功能
3.3程序框架
3.4 安装 readline 库
3.5命令提示符：
3.6用户命令分析：实现内部命令
3.7命令处理：实现外部命令、可执行文件和无效命令
3.8管道支持
3.9重定向支持

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）

1. 学习使用 Linux 进程间通信：管道、消息队列、共享内存。
即学习 BlackBoard 中的“综合 1 预备-进程间通信与同步”，完成材料中的全部操作，并截屏记录（实验中的全部操作都需要截屏记录，并配有必要的文字说明或结果的解读）。（40 分）

2.设计编写以下程序，着重考虑其同步问题（30 分）：
2.1 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；
2.2 另一个程序（进程）生成两个进程/线程，用于显示缓冲区内的信息，这两个进程/线程并发读取缓冲区信息后将缓冲区清空（一个线程的两次显示操作之间可以加入适当的时延以便于观察）。

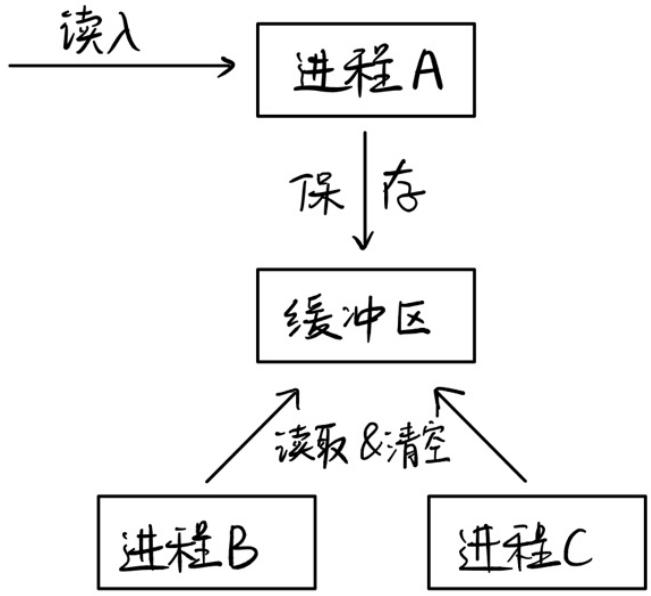
2.3 在两个独立的终端窗口上分别运行上述两个程序，展示其同步与通信功能，要求一次只有一个任务在操作缓冲区。
2.4 运行程序，记录操作过程的截屏并给出文字说明。（要求使用 POSIX 信号量来完成这里的生产者和消费者的同步关系。）

3 设计简单的 shell 程序

3.1 尝试自行设计一个 C 语言小程序，完成最基本的 shell 角色：给出命令行提示符、能够逐次接受命令；对于命令分成三种，内部命令（例如 help 命令、exit 命令等）、外部命令（常见的 ls、cp 等，以及其他磁盘上的可执行程序 HelloWrold 等）以及无效命令（不是上述三种命令）。（20 分）
3.2 参考“综合 1 预备-进程间通信与同步”中的 4.1.1 小节内容将上述 shell 进行扩展，使得你编写的 shell 程序具有支持管道的功能，也就是说你的 shell 中输入“dir || more”能够执行 dir 命令并将其输出通过管道将其输入传送给 more 作为标准输入。（10 分）
3.3 可以将 1/2 直接合并完成。
3.4 设计标准的参考。1)提示符最低标准是固定字符串。提升标准是使用含当前路径的信息为提示符。2）接受命令的最低标准是一次接受一个命令就退出 shell 程序，提升标准是在 shell 内部循环读取和执行命令。3）如果实现输出/输入重定向可以最多加 10 分，加分上限满分 100。

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）

2. 设计编写以下程序，着重考虑其同步问题：
2.1 分析题目与学习准备
2.1.1 分析题目，画出进程关系
分析题目可知，一个进程向缓冲区写数据，两个进程向缓冲区读取数据，缓冲区有多块数据，三个进程不能同时操作该缓冲区；这与进程同步问题中生产者——消费者模型非常相似，负责向缓冲区写数据的为“生产者”，向缓冲区读数据的为“消费者”，缓冲区里的数据为“产品”；



2.1.2 实现共享内存的基本步骤：创建，关联，分离

（1）创建

Shmget()函数的三个参数，key 为共享内存对象的键值，为 0 时会创建新的共享内存对象；size 表示要创建的共享内存的大小；shmflg 为共享内存标识符。该函数返回共享内存地址的引用标识符。

（2）关联

创建了共享内存后，还需要将共享内存区域映射到进程的虚拟地址空间，然后才能使用这块共享内存。和这个关联操作相关的函数为 shmat()。

其中：参数 shmid 是 shmget 返回的共享内存对象的引用标识符；参数 shmaddr 用来指定共享内存在进程虚拟地址空间中对应的虚拟地址； shmflg 是映射标志。该函数返回共享内存在进程中的虚拟地址。

（3）分离

当进程不再需要使用共享内存时，需要将它与共享内存分离，只有当最后一个使用该共享内存的进程与这块共享内存分离后，这块共享内存才会被释放。完成这个分离动作的函数是 shmdt()。

其中：传入的参数为共享内存的虚拟地址，然后找到进程的所有内存结构中和这个地址对应的一个，调用成功时返回 0

2.2 头文件 shm_com_sem.h

```
1. //头文件 shm_com_sem.h
2. #include <fcntl.h>
3. #include <sys/stat.h>
4. #include <semaphore.h>
5.
6. #define LINE_SIZE 256
7. #define NUM_LINE 16
```

```
8.
9. //用于创逮信号量时的识别 id
10. const char * queue_mutex="queue_mutex";
11. const char * queue_empty="queue_empty";
12. const char * queue_full="queue_full";
13.
14. //生产者消费者公用的缓冲区，含读写指针 line_write 和
    line_read
15. //以及缓冲数据区 buffer[NUM_LINE][LINE_SIZE], buffer [X]
    指向第 X 行信息，buffer[X][Y]指向 X 行信息的第 Y 个字符
16. struct shared_mem_st
17. {
18.     char buffer[NUM_LINE][LINE_SIZE];//缓冲数据区
19.     int line_write; //读写指针
20.     int line_read;
21. };
```

2.3 producer.c

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；

设计思路：

生产者创建共享内存并映射到本进程的进程空间；
生产者创建信号量；
生产者将输入的行写入缓冲区，要有信号量操作；
生产者释放信号量；
生产者解除共享内存映射；
生产者删除共享内存；

设计流程：

i. 从键盘获得数据（一行字符串）即一个产品；
ii. Sem_wait(sem_queue_empty)，判断缓冲区中是否有空行（没有添加过数据或者已被消费的数据）；如果没有，即信号量值为 0，则不能进入生产区；如果信号量值大于 0，信号量的值代表可用空行数目，则进入下一步，信号量减一；
iii. Sem_wait(sem_queue)，利用互斥信号量判断缓冲区是否上锁，即缓冲区是否正被使用，如果信号量值为 1，表示缓冲区可用，信号量值减一变为 0，上锁，进入临界区；
iv. 进入临界区（生产产品），将第一步获得的数据根据写指针（line_write）存到缓冲区，指针+1；
v. Sem_post(sem_queue);解锁，mutex 值+1，缓冲区恢复可用；
vi. Sem_post(sem_queue_full);信号量值变为 1，表示缓冲区有可消费产品，消费者可利用；

vii. 判断输入的数据是否为结束标志”quit”，若不是，重复 1~5；若是，退出循环，释放信号量，解除共享内存映射。

关键代码分析：

把这个问题看成一个生产-消费者问题，于是设置三个信号量，分别为：mutex,empty,full；当信号量不存在时需要使用信号量时，使用 sem_open 函数来创建一个信号量：第一个参数是这个有名信号量的名字，第二个参数选择创建或者打开一个信号量，第三个参数为信号量权限设置，最后一个参数为信号量的初始值。

这里将三个信号量的权限都设为 666，再将 mutex 和 empty 的初始值设置为 1，full 的初始值设置为 0。创建的共享内存的权限设置为 666（4（可读）+2（可写） = 6），否则读进程可能无法读取该内存区域，而在输出时出现错误。

```
1. //创建三个信号量
2. sem_queue=sem_open("queue_mutex",O_CREAT,0666,1);
3. sem_queue_empty=sem_open("queue_empty",O_CREAT,0666,1);
4. sem_queue_full=sem_open("queue_full",O_CREAT,0666,0);
```

设置访问共享内存的信号量指针，然后用 Shmget()函数创建共享内存

```
1.     int stmid;//共享内存 id
2.
3.     //访问共享内存的信号量指针
4.     sem_t *sem_queue,*sem_queue_empty,*sem_queue_full;
5.
6.     //创建一个新的共享内存
7.     stmid=shmget((key_t)1234,sizeof(struct shared_mem_s
   t),0666|IPC_CREAT);
8.      if(stmid==-1)
9.     {
10.       perror("shmget failed");
11.       exit(1);
12.     }
```

这里的临界资源为共享内存，所以只需要将对共享内存的操作，写入（读出）的代码放在临界区即可。sem_wait 相当于 wait，而 sem_post 相当于 signal，这里需要按照“先私后公”的原则，以免出现死锁。

```
1.  char b[BUFSIZ];
2.     int running=1;
3.     while(running)
4.     {
5.         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);
     }
```

这里为了防止结束后进程阻塞，所以再发送一次信号量和内容

```
    //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
```

最后，为了避免不必要的麻烦，在使用后应该释放共享内存区

```
    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);
     }
}
```

2.4 customer.c

另一个程序（进程）生成两个进程/线程，用于显示缓冲区内的信息，这两个进程/线程并发读取缓冲区信息后将缓冲区清空（一个线程的两次显示操作之

间可以加入适当的时延以便于观察）。

设计思路：

消费者获取共享内存并映射到本进程的进程空间；
消费者获取信号量；
消费者打印消费内容及进程号，要有信号量操作，发现 quit 退出；
消费者释放信号量；
消费者解除共享内存映射；
消费者删除共享内存；

设计流程：

i. sem_wait(sem_queue_full)；判断缓冲区是否有可消费的产品；若信号量值为 1，进入下一步；

ii. Sem_wait(sem_queue)，利用互斥信号量判断缓冲区是否上锁，即缓冲区是否正被使用，如果信号量值为 1，表示缓冲区可用，信号量值减一变为 0，上锁，进入临界区；

iii. 进入临界区（消费），从共享内存读数据，判断是否为结束标志”quit”；如果是，表示没有产品可消费了，又因为不止一个消费者，所以为了让其他消费者也得知，我利用 sem_post(sem_queue_full)告诉其他进程可消费，这样每一个进程都会或者 quit 的通知，结束该消费者进程；如果产品不是”quit”，将数据输出，并且读指针(line_write)增加 1，然后解锁缓冲区，sem_post(sem_queue_empty)，计数信号量+1；

关键代码分析：

前面已经知道在使用共享内存前，要先创建，关联，而创建在前面 producer.c 进程中已经做了，这里只需要关联上已经创建好的共享内存即可，可以通过从程序的参数中读入已经创建好的共享内存编号。

```
1.     //获取已创建共享内存区，识别号为 1234
2.     stmid=shmget((key_t)1234,sizeof(struct shared_mem_s
   t),0666|IPC_CREAT);
3.      if(stmid==-1)
4.     {
5.       perror("shmget failed");
6.       exit(1);
7.     }
```

同样的，因为这几个有名信号量已经在 producer.c 进程中创建了，此时再打开信号量时只需要按照如下的方式即可：

```
1.  //获取已创建的三个信号量
```

```
    sem_queue=sem_open("queue_mutex",1);
    sem_queue_empty=sem_open("queue_empty",1);
    sem_queue_full=sem_open("queue_full",0);
```

接下来是创建两个进程，这里父子进程的操作都是相似的，这里以分析子进程为例。首先对于临界区区操作，如同上面一样，应该将读取共享内存和清除共享内存内容的操作放在临界区。然后打印所消费的消息内容和进程号，如果发现消息为“end” 则退出

```
    if(fork_result==0)//子进程
    {
        while(running)
        {
            sem_wait(sem_queue_full);
            sem_wait(sem_queue);
            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;
             }
```

同样的，这里为了防止结束后进程阻塞，所以再发送一次信号量和内容

```
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);
```

最后要记得释放信号量

```
   sem_unlink(queue_mutex);
        sem_unlink(queue_empty);
        sem_unlink(queue_full);
```

父进程与子进程是相似的，代码如下

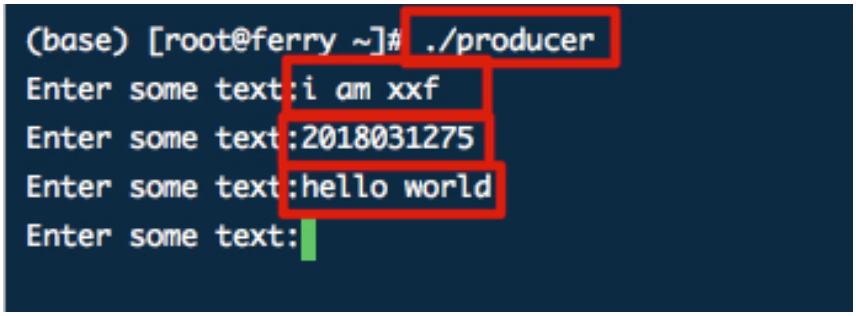
```
 else//父进程，与子进程相似
    {   while(running)
        {
```

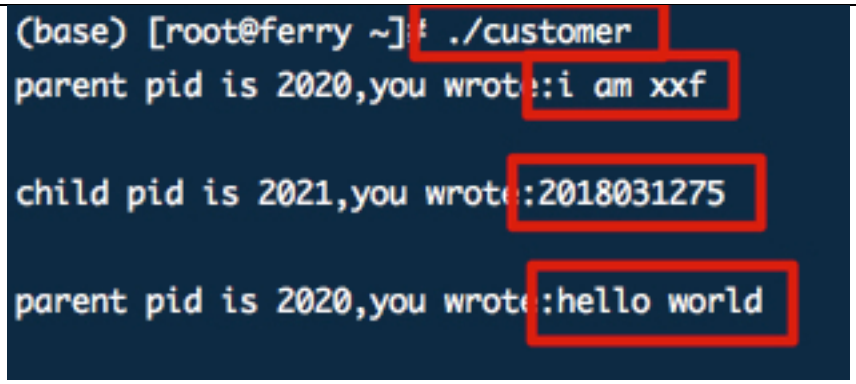
```
4.              sem_wait(sem_queue_full);
5.              sem_wait(sem_queue);//等待信号量
6.              sleep(2);
7.              printf("parent pid is %d,you wrote:%s\n",ge
   tpid(),shared_stuff->buffer[shared_stuff->line_read]);/
   /输出进程号和消息内容
8.
9.              if(strncmp(shared_stuff->buffer[shared_stuf
   f->line_read],"end",3)==0)//如果为 end 则退出
10.              {
11.               running=0;
12.              }
13.              shared_stuff->line_read=(shared_stuff->lin
   e_read+1)%NUM_LINE;//读指针改变
14.              sem_post(sem_queue);//发送信号量
15.              sem_post(sem_queue_empty);
16.
17.         }
18.
19.         sem_unlink(queue_mutex);
20.         sem_unlink(queue_empty);
21.         sem_unlink(queue_full);
22.     }
23.     waitpid(fork_result,NULL,0);
24.     exit(EXIT_SUCCESS);
25. }
```

2.5 运行结果分析

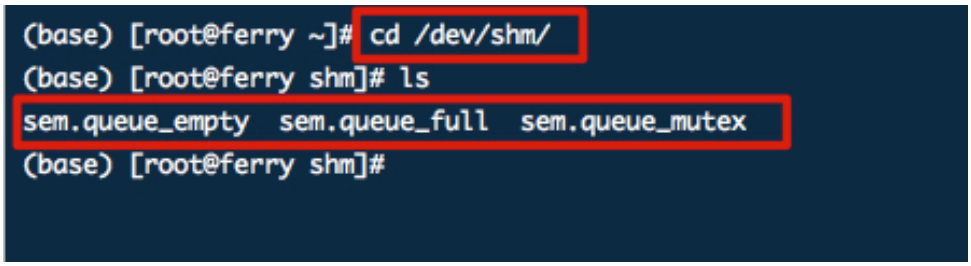
在两个独立的终端窗口上分别运行上述两个程序，展示其同步与通信功能，要求一次只有一个任务在操作缓冲区。

编译执行上面的两个程序，使用 gcc 编译时，需要加上-lpthread 选项。需要先通过./producer 运行 producer 进程创建共享内存，再通过./customer 运行 customer 进程。下图为测试结果（谢晓锋 2018031275）：





可以看到，进程之间正常通信，父子进程可以互斥地读取共享内存的内容。
还可以看一下所创建的有名信号量：位置在/dev/shm/ 可以看到，所创建的三个信号量，mutex，empty 和 full。



3 设计简单的 shell 程序

3.1 shell 的基本功能分析

Shell 是用户和系统内核之间的一个接口程序，shell 可以较好地保护系统内核免受非法操作的破坏，也能让用户能方便地完成自己的任务。它是一个命令解释 command-language interpreter，我们提交的每条命令都会通过 shell 的解释，然后再提交给内核执行。

shell 包含内部命令和外部命令，比如打印目录的 pwd 命令就是一个内部命令，而压缩文件时用的 tar 命令则是一个外部命令。对于用户来讲，内部命令和外部命令使用起来并没有差别，当然我们也并不关心这条命令究竟是外部命令还是内部命令。
对于 Shell 来说，当用户需要执行一条命令时，它会先看看这条命令是否为内部命令，如果不是，再去环境变量$PATH 中去找一下这是不是一个可执行的程序，如果都不是，shell 会返回错误，提示没找到这条命令。

比如：执行 xiexiaofeng（谢晓锋）命令，就会返回错误

下图展示了几个常用的 shell 内部命令，echo 用来指定的字符串；cd 用来切

换目录；help 用来输出帮助信息。

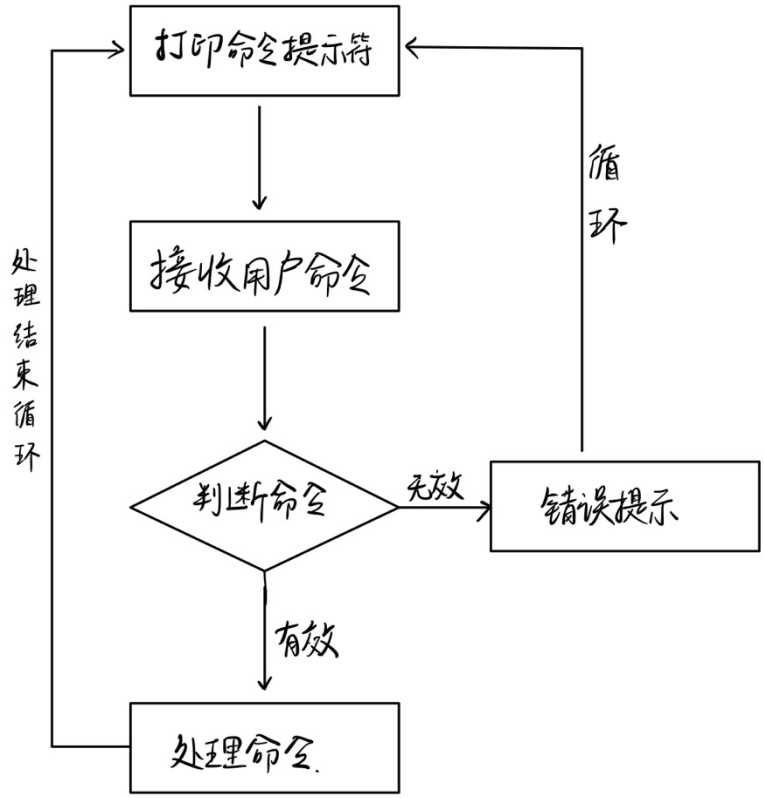
shell 外部命令包括 linux 提供的一些使用的命令，比如 ls，grep 等，和我们自行安装的一些软件。

3.2 预计 myshell 实现的功能

（1）命令提示符：输出命令提示符
（使用含当前路径的信息为提示符并设置对应的颜色）
（2）常用内部命令：help,echo,cd,exit 等
（3）常用外部命令：ls，cp 等
（4）可执行程序运行支持：通过./hellotest 这样的方式执行可执行程序
（5）无效命令提示：如上图执行 xiexiaofeng 时一样，提示未找到命令
（6）管道功能：能够执行 dir | more 等命令
（7）输出/输入重定向功能：以输出重定向为例
（8） myshel 支持在内部循环读取和执行命令，并可以查看历史命令

3.3 程序框架

根据以上分析，很容易想到 shell 的运行机制流程图如下所示：



根据 shell 的运行机制流程图，就可以设计出 shell 程序的大致框架：

```
while(TRUE){
print_prompt();
get_command();
if command valid
deal_command();
else
    print_error();
}
```

通过查阅资料，知道 shell 的基本框架可以用下面的代码概括，这部分代码出自于《现代操作系统（英文第三版）》原书 P54 图 1-19，可以看到和刚开始自己想的框架没有太大差别，说明大体思路是没有问题的。只是自己的框架太“抽象”了些，没有多少指导意义，而这里则比较清楚地展示了 shell 执行中需要 fork 命令子进程，以及调用 exec 族函数来执行命令等细节。

```
#define TRUE 1

while(TRUE) {
    type_prompt();
    read_command(command,parameters);
    if(fork()!=0) {
        /* Parent code */
```

```
        waitpid(-1,&status,0);
    } else {
        /* Child code */
        execve(command,parameters,0);
    }

}
```

通过命令 echo $SHELL 查看默认 shell 版本，可以看到正在使用的 shell 版本，所以本实验中模仿的 shell 以 bash 为基础。

```
(base) [root@ferry ~]# echo $SHELL
/bin/bash
```

3.4 安装 readline 库

根据搜索的资料，为了达到较好的命令交互效果，需要使用 realine 库。
首先通过镜像命令下载源码，并且解压

```
wget -c ftp://ftp.gnu.org/gnu/readline/readline-6.2.tar.gz
tar -zxvf readline-6.2.tar.gz
```

然后执行 cd readline-6.2 进入 readline 所在目录，执行./configure 命令进行必要的配置，然后执行 make，再执行 make install，完成对 readline 库的配置工作。

```
(base) [root@ferry ~]# wget -c ftp://ftp.gnu.org/gnu/readline/readline-6.2.tar.gz
--2020-04-14 18:29:51--  ftp://ftp.gnu.org/gnu/readline/readline-6.2.tar.gz
           => 'readline-6.2.tar.gz'
Resolving ftp.gnu.org (ftp.gnu.org)... 209.51.188.20, 2001:470:142:3::b
Connecting to ftp.gnu.org (ftp.gnu.org)|209.51.188.20|:21... connected.
```

使用 readline 库应该包含下面两个头文件，第一个头文件提供的 readline()可以获取用户输入，第二个头文件是实现历史命令时需要的。

```
#include <readline/readline.h>
#include <readline/history.h>
```

使用时用以下命令，其中 prompt 为命令提示符字符串，如果在这个 prompt 中使用了转义的话，应该用/001***/002 这样的方式把这部分括起来，否则可能出错。

```
if(!(line = readline(prompt)))
```

编译时使用 gcc myshell.c -o myshell -lreadline 这样的方式编译

3.5 命令提示符：

此部分关键的函数为 void set_prompt(char *prompt)，它用来输出命令提示符（并设置对应的颜色）。

为了达到打印不同颜色的字体的要求，首先定义一些基本的颜色。如下，要打印不同颜色字体时，只需要在要打印的字符串前面加上已经定义的颜色的宏即可，比如：printf(L_GREEN "hello world\n");

```
1. //define the printf color
2. #define L_GREEN "\e[1;32m"
3. #define L_BLUE  "\e[1;34m"
4. #define L_RED   "\e[1;31m"
5. #define WHITE   "\e[0m"
```

myshell 设置命令提示符输出形式为："用户名@主机名:路径$"（root 权限为#提示符），对应地应该包含 6 个元素：
（1）用户名；
（2）"@"符号；
（3）主机名；
（4）"："符号
（5）当前目录
（6）用户权限符号(用"#"表示 root)

其中，此部分比较关键的几个函数为:
（1）用户名：使用 getpwuid(getuid())获得，同时可以获得该用户 home 目录的路径；
（3）主机名：使用 gethostname()获得；
（4）当前目录：使用 getcwd()获得
（5）用户权限符号：模仿 bash 的风格，对于普通用户使用"$"，root 用户使用"#"，需要检测执行 myshell 的用户权限，利用 geteuid()是否为 0 来判断。

```
1.      if(gethostname(hostname,sizeof(hostname)) == -1){
2.          //get hostname failed
3.          strcpy(hostname,"unknown");
4.      }//if
5.      //getuid() get user id ,then getpwuid get the user information by user
    id
6.      pwp = getpwuid(getuid());
7.      if(!(getcwd(cwd,sizeof(cwd)))){
8.          //get cwd failed
9.          strcpy(cwd,"unknown");
10.     }//if
```

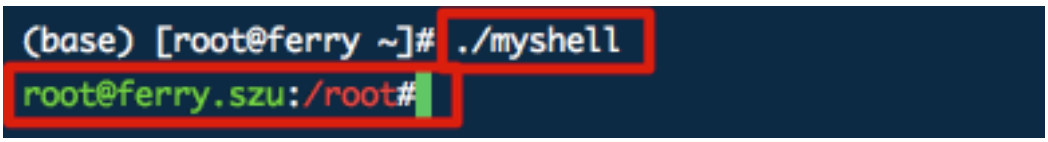
```
11.     char cwdcopy[100];
12.     strcpy(cwdcopy,cwd);
13.     char *first = strtok(cwdcopy,delims);
14.     char *second = strtok(NULL,delims);
15.     //if at home
16.     if(!(strcmp(first,"home")) && !(strcmp(second,pwp->pw_name))){
17.         int offset = strlen(first) + strlen(second)+2;
18.         char newcwd[100];
19.         char *p = cwd;
20.         char *q = newcwd;
21.
22.         p += offset;
23.         while(*(q++) = *(p++));
24.         char tmp[100];
25.         strcpy(tmp,"~");
26.         strcat(tmp,newcwd);
27.         strcpy(cwd,tmp);
28.     }
29.
30.     if(getuid() == 0)//if super
31.         super = '#';
32.     else
33.         super = '$';
```

为了和真正的命令提示符一样，这里使用字符串格式化函数 sprintf 来将命令提示符中的几部分（目录，主机名等）合到一个字符串中。

由于这里使用到 readline 库，readline.h 头文件中 readline 函数表示读入一行字符串，它包含一个 char *prompt 参数，传入的 prompt 将会作为命令提示符来处理。但需要对 prompt 再进行一些处理，根据 readline.h 的宏定义，prompt 中的转义应该用\001***\002 这样的方式来括起来。所以把颜色部分全部设置这样的方式：\001\e[1;32m\002，其中\e[1;32m 为颜色的控制信息，于是有如下的代码：

```
sprintf(prompt, "\001\e[1;32m\002%s@%s\001\e[0m\002:\001\e[1;31m\002%s\001\e[0m\002%c",pwp->pw_name,hostname,cwd,super);
```

编译运行 myshell.c，如下图所示，得到命令提示符：



3.6 用户命令分析：实现内部命令

根据前面对 Shell 的基本框架的分析，接下来要做的就是对用户输入的字符串进行分析。此部分的关键函数为 analysis_command()，它用来分析用户输入的命令，并将其进行拆分。

前面通过对 echo、help 等 shell 命令的执行，已经知道用户通常输入命令的一般格式为：命令 参数选项。在 myShell 中如果只是为了简单的达到执行命令的效果，可以使用 system(“command”)函数，但是这个函数不太安全，很容易出错，所以用户命令分析还是有必要的，通过分析输入的命令，拿到命令和参数，然后调用 exec 函数来执行它。

这里通过 readline()函数可以获得用户输入的字符串，再通过 strtok()函数完成字符串的工作，strtok()函数具有两个参数，第一个为要处理的字符串，第二个为分割字符，第一次调用这个函数时，需要指定要分割的字符，之后调用时，将此参数设置为“NULL”以继续完成分割。

```
strcpy(argv[0],strtok(command,delims));
    while(p = strtok(NULL,delims)){
        strcpy(argv[i++],p);
        argc++;
    }//while
```

这里用的分割字符为空格符，将分割后的字符串保存到 argv 中，用 argv[0]保存命令名称，argv[1]开始为命令的参数。分割完成字符后，可以进行一些基本命令的处理，比如 exit（退出 shell）,help（打印帮助信息）等。

```
//exit when command is exit
    if(strcmp(argv[0],"exit") == 0){
        exit(EXIT_SUCCESS);
    }
    else if(strcmp(argv[0],"help") == 0){
        help();
    }
```

由于 cd 命令比较容易出错，下面代码展示了如何处理 cd 命令。需要注意的是，cd 命令只能为 myShell 的内部命令，也就是说，不能 fork 子进程去执行“cd”命令。因为 fork 子进程执行完 cd 后，再返回父进程，当前目录会变为父进程的目录，达不到切换路径的效果。切换路径需要使用函数 chdir()。

```
    else if(strcmp(argv[0],"cd") == 0){
        char cd_path[100];
        if((strlen(argv[1])) == 0 ){
            pwp = getpwuid(getuid());
            sprintf(cd_path,"/home/%s",pwp->pw_name);
            strcpy(argv[1],cd_path);
            argc++;
```

```
        }
        else if((strcmp(argv[1],"~") == 0) ){
            pwp = getpwuid(getuid());
            sprintf(cd_path,"/home/%s",pwp->pw_name);

            strcpy(argv[1],cd_path);
        }

        //do cd
#ifdef DEBUG
        printf("cdpath = %s \n",argv[1]);
#endif
        if((chdir(argv[1]))< 0){
            printf("cd failed in builtin_command()\n")
  ;
        }
    }//else if cd
}
```

编译运行 myshell.c 从图中结果可以看出 ，内部命令：cd 命令，help 命令，echo 命令，exit 命令都能正常地工作。

```
(base) [root@ferry myshell]# gcc myshell.c -o myshell -lreadline
(base) [root@ferry myshell]# ./myshell
root@ferry.szu:/root/op/myshel#cd /root/op
        this is a builtin command: cd
cdpath = /root/op
root@ferry.szu:/root/op#help
        this is a builtin command: help
Hello,welcome to myShell!I am your Help Manual
root@ferry.szu:/root/op#echo hello

        the command is:echo with 2 parameter(s):
0(command): echo
1: hello
hello
root@ferry.szu:/root/op#exit
        this is a builtin command: exit
(base) [root@ferry myshell]#
```

3.7 命令处理：实现外部命令、可执行文件和无效命令

此部分的关键是“exec”函数的使用，但如果要实现管道重定向等扩展功能，还需要应用到进程同步和通信等知识。

事实上，Linux 中并没有一个名为“exec”的函数，而是六个以 exec 开头的

函数族，头文件都是#include<unistd.h>，调用成功则没有返回值，调用失败则返回-1，他们的函数原型分别为：

```
1. int execl(const char *path, const char *arg, …)
2. int execv(const char *path, char *const argv[])
3. int execle(const char *path, const char *arg, …, char *const envp[])
4. int execve(const char *path, char *const argv[], char *const envp[])
5. int execlp(const char *file, const char *arg, …)
6. int execvp(const char *file, char *const argv[])
```

前四个函数以完整的文件路径进行文件查找，后两个以 p 结尾的函数，可以直接给出文件名，由系统从$PATH 中指定的路径进行查找。

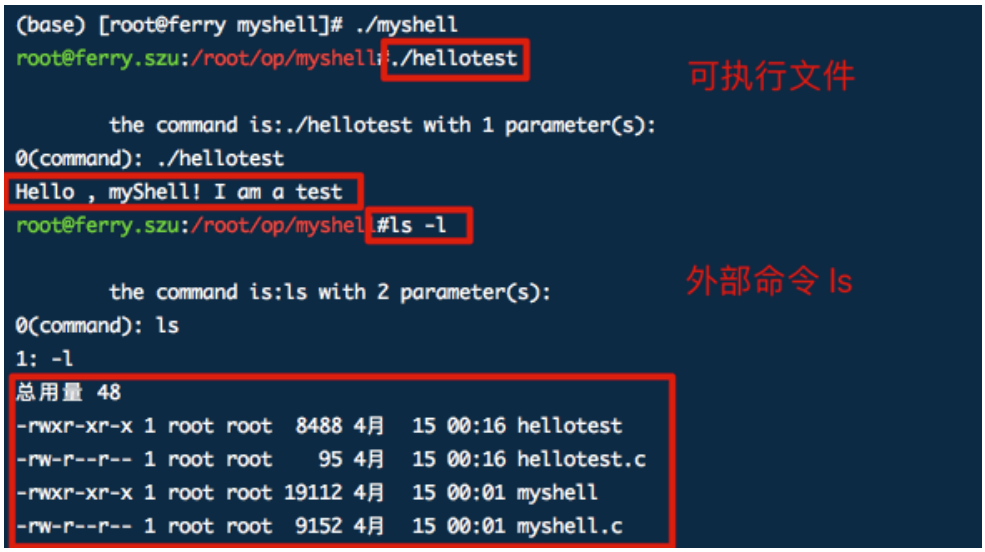
这里不同的函数后缀，代表不同的含义是：
l：接收以逗号分隔的参数列表，列表以 NULL 指针作为结束标志
v：接收到一个以 NULL 结尾的字符串数组的指针
p：是一个以 NULL 结尾的字符串数组指针，函数可以通过$PATH 变量查找文件
e：函数传递指定参数 envp，允许改变子进程的环境，无后缀 e 时，子进程使用当前程序的环境

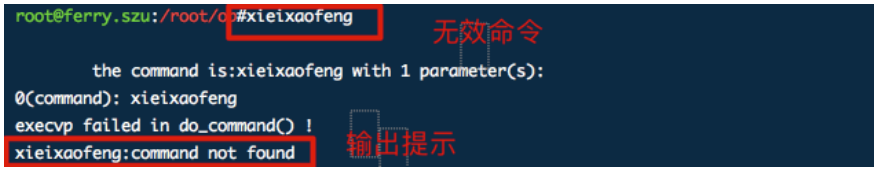
这六个函数中真正的系统调用只有 execve()，其他的都是库函数，它们最终都会调用到 execve()；由于 exec 函数常常会因为找不到文件，或者没有对应文件的运行权限等原因而执行失败，所以，在使用是最好加上错误判断语句。

这里直接使用 execvp(),然后传入待执行命令，待执行命令参数作为该函数的参数。根据 execvp()函数的要求，这里的待执行命令的参数必须以 NULL 结尾，而且这个参数的第 0 个字符串为待执行命令，第一个字符串为待执行命令的第一个参数

```
1.  else if(pid1 == 0){
2.                  dup2(fd[1],1);//dup the stdout
3.                  close(fd[0]);//close the read edge
4.                  if(execvp(argvtmp1[0],argvtmp1) < 0){
5.                      #ifdef DEBUG
6.                      printf("execvp failed in do_command
   () !\n");
7.                      #endif
8.                      printf("%s:command not found\n",arg
   vtmp1[0]);
9.                  }//if
10.         }//else if child pid1
```

编译运行 myshell.c 如下图可以看到，外部命令：ls、cp，可执行程序 hellotest 以及无效命令都正常执行，





3.8 管道支持

下图是 shell 中管道命令的使用，“|”前的表示第一条执行的命令，它的输出结果将通过管道传送给第二条命令，作为第二条命令的参数。

```
(base) [root@ferry op]# dir | more
hellotest.c  myshell
(base) [root@ferry op]#
```

这里使用无名管道（仅能在具有亲缘关系的进程间使用）在 myshell 中实现对管道命令的支持。这部分的设计伪代码如下

```
1.  BEGIN:
2.  pipe(fd);
3.  pid1 = fork();
4.  if(pid1 < 0)
5.   print(“ERROR”);
6.  else if(pid1 == 0)
7.   close(fd[0]);//close read
8.   dup2(fd[1],stdout);//duplicate the file handle
9.   exec(command1,parameters);
10. else if(pid1 > 0)
11.  waitpid(pid1);
12.         pid2 = fork();
13.         if(pid2 < 0)
14.            printf(“ERROR”);
15.         else if(pid2 == 0)
16.           close(fd[1]);//close write
17.           dup2(fd[0],stdin);
18.           exec(command2.parameters);
19. else
20. close(fd);
21.             waitpid(pid2);
22. END
```

其中，dup2 函数用来复制文件描述符。在子进程（执行第一条命令）中，复制 stdout 描述符到管道写端，表示子进程将输出它的结果，类似地在另外一个子进程中（执行第二条命令）读端的描述符现在变为 stdin，它用来接收第一条命令的结果。如此达到了通过管道在第一条命令和第二条命令间通信的目的。

这里由于在命令的保存问题中，管道命令不能当成一条命令来执行，所以，在命令分析阶段通过判断“|”的存在来将命令分割成两部分，然后再分别在子进程中调用 execvp 来执行。

3.9 重定向支持

重定向包含输入和输出重定向。

输入重定向：是指不使用系统提供的标准输入端口，而进行重新的指定。换言

之，输入重定向就是不使用标准输入端口输入文件，而是使用指定的文件作为标准输入设备。

输出重定向：是指不使用 linux 默认的标准输出设备显示信息，而是指定某个文件做为标准输出设备来存储文件信息。（以覆盖的方式把指定文件的信息输出到指定文件）

这里以输出重定向为例进行分析。输出重定向和管道命令类似，扫描一下整个命令字符串，看看有没有输出重定向符号“>”，如果有，则把它当做一个重定向命令，重定向符号前面部分为命令，后面部分为重定向的目标（保存在另一个字符串 redirect_target 中）。

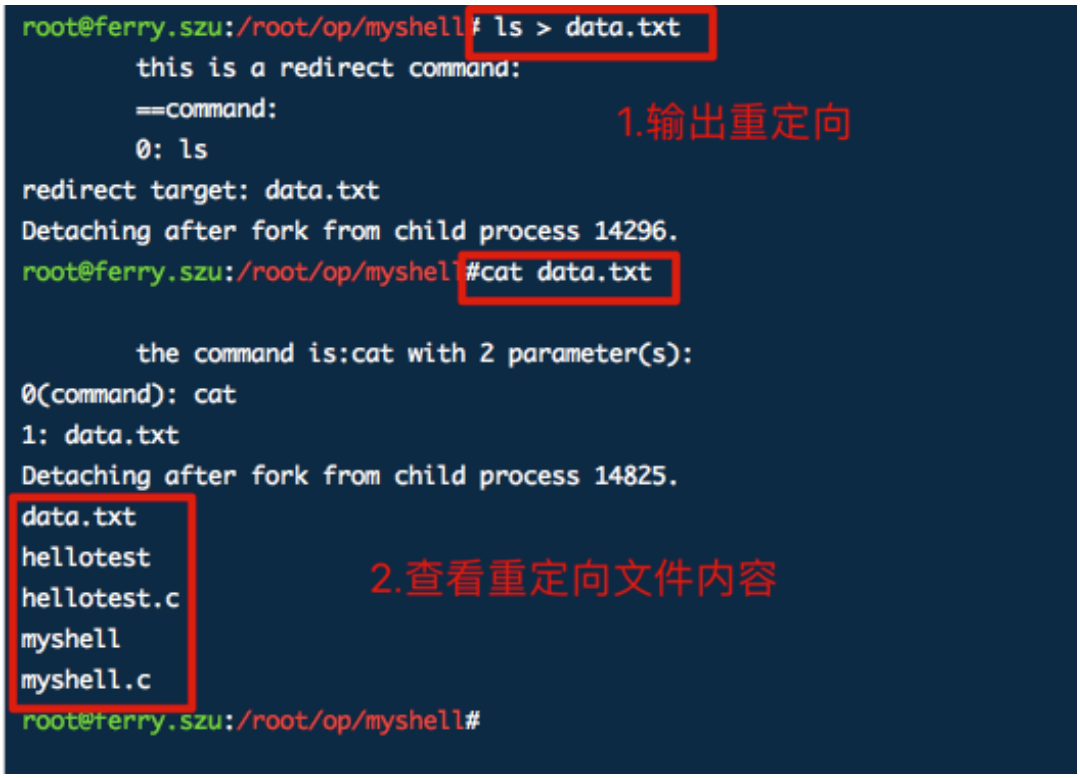
重定向的关键函数为 freopen()，它的原型为:

```
FILE *freopen( const char *path, const char *mode, FILE *stream )
```

这部分设计的伪代码为：

```
1. BEGIN:
2. freopen(redirect_target,”w”,stdout);
3. execvp(redirect_command,parameters);
4. fclose(stdout);
5. END
```

编译运行 myshell.c,执行 ls 命令并将结果重定向到 data.txt 文件中，然后使用 cat 命令查看 data.txt 的内容，可以看出，重定向命令执行成功：



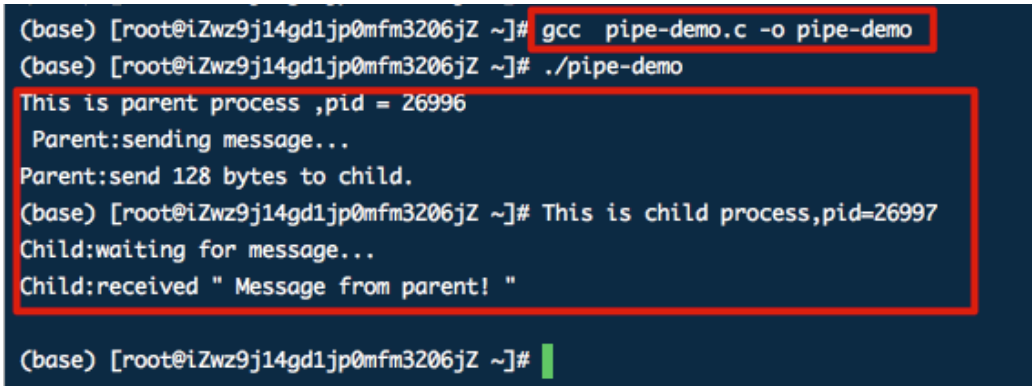
1. 学习使用 Linux 进程间通信：管道、消息队列、共享内存。

1.1 管道

1.1.1 无名管道

pipe()将通过两个文件描述符（整数）来指代管道缓冲区的读端和写端（代码中用 fds[] 变量记录）。其中父进程关闭管道的读端 fds[0]并往管道的写端 fds[1]写出信息，子进程关闭了管道的写端 fds[1]并从管道的读端 fds[0]读回信息。

下图为 pipe-demo.c 运行的输出，其表明父进程发送了消息到管道，自进程成功接受到 了“Message from parent”。

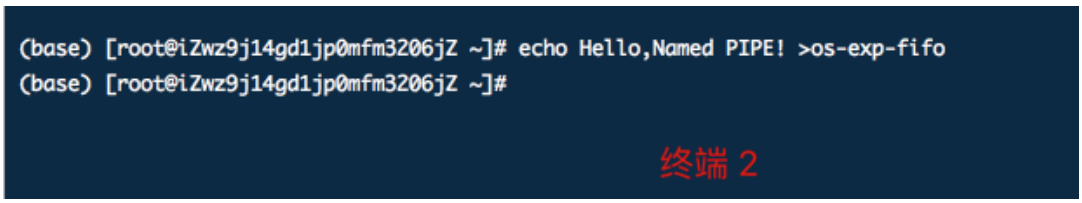


1.1.2 命名管道

用 mkfilo 命令来创建命名管道 os-exp-fifo，ls 命令查看时，可以看出其类型是管道“p”；

用 cat os-exp-fifo 命令尝试从管道中读入数据，但是此时管道中还没有写入任何数据，因此 cat 将进入阻塞状态。

如果此时在另一个终端 2 上，用“echo Hello, Named PIPE! > os-exp-fifo”相关到写入数据，



再回到终端 1，则 cat 会被唤醒并读入管道数据回显字符串“Hello, Named PIPE!”

1.2 System V IPC

1.2.1 System V IPC 介绍

在 Linux 中执行 ipcs 命令可以查看到当前系统中所有的 System V IPC 对象，如屏显所示。此时系统中还没有创建消息队列和信号量数组（或称信号量集），有 3 段共享内存区。

```
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# 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     nattch     状态
0x00000000 65536      lightdm    600        33554432   2          目标

--------- 信号量数组 -----------
键         semid      拥有者  权限     nsems

(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]#
```

查看这些 IPC 对象时还可以带上参数，ipcs -a 是默认的输出全部信息、ipcs -m 显示共享内 存的信息、ipcs -q 显示消息队列的信息、ipcs -s 显示信号量集的信息。另外用还有一些格式控 制的参数， -t 将会输出带时间信息、-p 将输出进程 PID 信息、-c 将输出创建者/拥有者的 PID、 -l 输出相关的限制条件。

例如用 ipcs -ql 将显示消息队列的限制条件，如图所示。

```
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ipcs -ql

---------- 消息限制 -----------
系统最大队列数量 = 7573
最大消息尺寸（字节）= 8192
默认的队列最大尺寸（字节）= 16384
```

1.2.2 消息队列：

执行 msgtool s 1 Hello,my_msg_queue!以便发送类型为 1 的消息，然后用

ipcs -q 查看到新创建了一个消息队列（ID 为 0x6d00e3e3），里面有 20 个字节（Hello,my_msg_queue!） 的 1 条消息。

```
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ./msgtool s 1 Hello,my_msq_queue!
Sending a message
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ipcs -q

--------- 消息队列 -----------
键         msqid      拥有者   权限     已用字节数 消息
0x6d010001 0          root       666        20         1
```

此时再执行 msgtool -r 1（是另一个进程了） 读走类型为 1 的消息，然后再用 ipcs -q 可以看到该消息队列为空（0 字节）了。如图所示。

```
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ./msgtool r 1
Reading a message
Type: 1 Text: Hello,my_msq_queue!
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ipcs -q

--------- 消息队列 -----------
键         msqid      拥有者   权限     已用字节数 消息
0x6d010001 0          root       666        0          0

(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]#
```

1.2.3 共享内存：

创建共享内存：shmget-demo.c 代码创建了一个 4096 字节的共享内存区。shmget()的第一个参数 IPC_PRIVATE（=0，表示创建新的共享内存），第二个参数是共享内存区的大小，第三个是访问模式。虽然也可以像前面的消息队列的例子那样通过 ftok()将键值转换成 ID，但这里没有指定 ID，而是创建共享内存后由系统返回一个 ID 值(后面的进程要使用该共享内存时需要指定该 ID）。

执行 shmget-demo 程序，其输出如图所示。输出结果表明新创建的共享内存的 ID 为 98305， 长度为 4096 字节，当前还没有进程将他映射到自己的进程空间（nattch 列为 0）

```
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# gcc shmget-demo.c -o shmget-demo
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ./shmget-demo
Successfully created segment: 98305

------------ 共享内存段 --------------
键         shmid      拥有者  权限     字节     nattch     状态
0x00000000 65536      lightdm    600        33554432   2          目标
0x00000000 98305      root       666        4096       0

(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]#
```

执行 shmatt-write-demo 98305（命令行参数中指出共享内存的 ID 为 98305），输出了当前的共享内存信息，可以看到 ID 为 98305 的共享内存已经有被映射了一次 （nattach 列为 1），该进程将共享内存映射到了进程空间 0x7fd344917000 位置的地方:

```
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# gcc shmatt-write-demo.c -o shm
att-write-demo
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ./shmatt-write-demo 98305
Segment attached at 0x7fd344917000

------------ 共享内存段 --------------
键         shmid      拥有者  权限     字节     nattch     状态
0x00000000 65536      lightdm    600        33554432   2          目标
0x00000000 98305      root       666        4096       1
```

在另外一个终端上观察该进程的进程空间，可以看到相应位置有一个新的区域（7fd344917000-7fd344918000）

```
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ps -a
  PID TTY          TIME CMD
21835 pts/2    00:00:00 shmatt-write-de
23524 pts/1    00:00:00 top
25917 pts/0    00:00:00 ps
32544 pts/3    00:00:02 top
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# cat /proc/21835/maps
00400000-00401000 r-xp 00000000 fd:01 932252                             /root/shmatt-write-demo
00600000-00601000 r--p 00000000 fd:01 932252                             /root/shmatt-write-demo
00601000-00602000 rw-p 00001000 fd:01 932252                             /root/shmatt-write-demo
7fd34432b000-7fd3444ed000 r-xp 00000000 fd:01 265443                     /usr/lib64/libc-2.17.so
7fd3444ed000-7fd3446ed000 ---p 001c2000 fd:01 265443                     /usr/lib64/libc-2.17.so
7fd3446ed000-7fd3446f1000 r--p 001c2000 fd:01 265443                     /usr/lib64/libc-2.17.so
7fd3446f1000-7fd3446f3000 rw-p 001c6000 fd:01 265443                     /usr/lib64/libc-2.17.so
7fd3446f3000-7fd3446f8000 rw-p 00000000 00:00 0
7fd3446f8000-7fd34471a000 r-xp 00000000 fd:01 265153                     /usr/lib64/ld-2.17.so
7fd344904000-7fd344907000 rw-p 00000000 00:00 0
7fd344915000-7fd344917000 rw-p 00000000 00:00 0
7fd344917000-7fd344918000 rw-s 00000000 00:04 98305                      /SYSV00000000 (deleted)
7fd344918000-7fd344919000 rw-p 00000000 00:00 0
7fd344919000-7fd34491a000 r--p 00021000 fd:01 265153                     /usr/lib64/ld-2.17.so
7fd34491a000-7fd34491b000 rw-p 00022000 fd:01 265153                     /usr/lib64/ld-2.17.so
7fd34491b000-7fd34491c000 rw-p 00000000 00:00 0
7fffb185f000-7fffb1880000 rw-p 00000000 00:00 0                          [stack]
7fffb1914000-7fffb1916000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]#
```

击键回车后 shmat-write-demo 将解除共享内存的映射，此时 ipcs -m 显示对应的共享内 存区没有人使用（nattch 列为 0）

```
------------ 共享内存段 --------------
键         shmid      拥有者   权限     字节     nattch     状态
0x00000000 65536      lightdm    600        33554432   2          目标
0x00000000 98305      root       666        4096       1

Segment detached

------------ 共享内存段 --------------
键         shmid      拥有者   权限     字节     nattch     状态
0x00000000 65536      lightdm    600        33554432   2          目标
0x00000000 98305      root       666        4096       0
```

此时再尝试用另一个程序去映射该共享内存并从中读取数据,虽然创建该共享内存的进程已经结束了，可是 shmatt-read-demo 映射 ID 为的共享内存后，仍读出了原来写入的字符串

```
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# gcc Shmatt-read-demo.c -o Shmatt-read-demo
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ./Shmatt-read-demo 98305
Segment attached at 0x7f2b8e248000

------------ 共享内存段 --------------
键         shmid      拥有者   权限     字节     nattch     状态
0x00000000 65536      lightdm    600        33554432   2          目标
0x00000000 98305      root       666        4096       1

The string in SHM is: Hello shared memory!
```

```
Segment detached

------------ 共享内存段 --------------
键         shmid      拥有者   权限     字节     nattch     状态
0x00000000 65536      lightdm    600        33554432   2          目标
0x00000000 98305      root       666        4096       0
```

1.3 POSXI 信号量

1.3.1 有名信号量：

psem-named-open.c 中先用 sem_open()创建了一个信号量，该信号量由一个字符串所标识（代码中是从命令行读入的一个文件名字符串），代码中使用了 O_CREAT 标志（如果信号量还不存在则创建它）并将信号量初值置为 1。

用 gcc psem-named-open.c -o psem-named-open -lthread（参数-lpthread

用于指出链 接时所用的线程库）完成编译，然后运行 psem-named-open()。如果没有输入作为标识的文件名字符串，则给出体系要求用户输入；如果输入一个文件名字符串，正常情况将完成创建过程；（此处输入 producer.c）

```
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# gcc psem-named-open.c -o psem-named-open -lpthread
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ./psem-named-open
Please input a file name to act as the ID of the sem!
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ./psem-named-open producer.c
```

然后，尝试执行 P/V 操作中的 V 操作（即对信号量进行减 1 操作，可能引发阻塞），程序 psem-named-wait-demo.c 通过 sem_wait()来执行 V 操作(减 1 操作），并且通过 sem_getvalue() 来查看信号量的值。
编译并执行 psem-named-wait-demo，输入前面创建信号量时使用的文件名标识（前面输入的 producer.c），此时打印出当前信号量值为 0（也就是说前面创建的时候初值是 1）。

```
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# gcc psem-named-wait-demo.c -o psem-named-wait-demo -lpthread
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ./psem-named-wait-demo producer.c
pid 19557 has semaphore, value = 0
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]#
```

如果在运行一遍，此时信号量的值已经为 0，在进行 V 操作（减 1 操作）将阻塞该进程。显示 psem-named-wait-demo 第二次运行后并没有返回到 shell 提示符，如果此时 用另一个中断执行 ps 命令可以看到该进程处于 S 状态。

```
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# gcc psem-named-wait-demo.c -o psem-named-wait-demo -lpthread
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ./psem-named-wait-demo producer.c
                                                                    阻塞
```

```
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ps aux |grep psem-named-wait-demo
root     29586  0.0  0.0   6504   384 pts/2    S+   00:17   0:00 ./psem-named-wait-demo producer.c
root     29950  0.0  0.0 112724  1004 pts/0    S+   00:17   0:00 grep --color=auto psem-named-wait-demo
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]#
```

再接着来看看对该信量进行 P 操作（增 1 操作），使得前面的 psem-named-wait-demo 进 程从原来的阻塞状态唤醒并执行结束。
编译并执行 psem-named-post-demo（与前面 psem-named-wait-demo 不在同一个终端 shell 上），可以看到此时信号量的值增加到 1，并使得原来阻塞的 psem-named-post-demo 被唤醒并执行完毕，如图所示。

```
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# gcc psem-named-post-demo.c -o psem-named-post-demo -lpthread
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ./psem-named-post-demo producer.c
value = 1
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]#
```

```
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ./psem-named-wait-demo producer.c

pid 9930 has semaphore, value = 0
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]#
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]#
```

1.3.2 无名信号量：

POSIX 无名信号量适用于线程间通信，如果无名信号量要用于进程间同步，信号量要放在 共享内存中（只要该共享内存区存在，该信号灯就可用）。无名

信号量和有名信号量的区别主 要在创建上，无名信号量使用 sem_init()创建

1.3.3 互斥量：

互斥量是信号量的一个退化版本，仅用于并发任务间的互斥访问。下面先用一个代码来展 示多线程并发且没有用互斥量保护共享变量的情形，
no-mutex-demo.c 该程序对一个缓冲区（缓冲区内是数值为 3、4、3、4.....交织的整数）内的每个整数进 行检查，并对数值为 3 的整数进行计数统计，统计工作由 16 个线程并发完成（每个线程负责 缓冲区的 1/16 的数据）。编译后运行 no-mutex-demo(注意编译时要有-lpthread 参数指出所需的线程库)，结果如图所示，每次运行结果并不唯一（共享变量未能得到互斥访问）。

```
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# gcc no-mutex-demo.c -lpthread -o no-mutex-demo
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ./no-mutex-demo
Total count = 18078140
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ./no-mutex-demo
Total count = 11135139
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ./no-mutex-demo
Total count = 12371236
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ./no-mutex-demo
Total count = 10404212
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]#
```

如果对 count++这个临界区加以保护，就能避免出现这个错误。运行 mutex_demo，每次运行都获得相同的结果，如图所示。由于实现了共享变量的互斥访问，因此每次运行的结构都是确定的值。
运行 mutex_demo，每次运行都获得相同的结果，如图所示。由于实现了共享变 量的互斥访问，因此每次运行的结构都是确定的值。

```
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# gcc mutex-demo.c -lpthread -o mutex-demo
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ./mutex-demo
otal count = 33554432
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ./mutex-demo
^[[ Total count = 33554432
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ./mutex-demo
otal count = 33554432
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]# ./mutex-demo
Total count = 33554432
(base) [root@iZwz9j14gd1jp0mfm3206jZ ~]#
```

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）

2.设计编写以下程序，着重考虑其同步问题：
2.1 分析题目与学习准备
2.1.1分析题目，画出进程关系
2.1.2实现共享内存的基本步骤：创建，关联，分离
2.2头文件shm_com_sem.h
2.3 producer.c
2.4 customer.c
2.5运行结果分析

3.设计简单的shell程序
3.1 shell的基本功能分析
3.2预计myshell实现的功能
3.3程序框架
3.4 安装 readline 库
3.5命令提示符：
3.6用户命令分析：实现内部命令
3.7命令处理：实现外部命令、可执行文件和无效命令
3.8管道支持
3.9重定向支持

1. 学习使用Linux进程间通信
1.1管道
1.1.1无名管道
1.1.2命名管道
1.2 System V IPC
1.2.1 System V IPC介绍
1.2.2消息队列
1.2.3共享内存
1.3 POSXI信号量
1.3.1有名信号量
1.3.2无名信号量
1.3.3互斥量

五、实验体会：（根据自己情况填写）

（1）此实验报告在阿里云服务器的实验环境中完成，对于第一部分由于完成时间比较早，主机名为系统自带的一串不可识别的英文，收到要求后修改主机名为 ferry.szu（ferry 为自己的英文名），之后的实验也将在这个主机名下完成截图。

（2）实验报告的代码都是在此网站“Syntax Highlight Code In Word Documents”，进行高亮处理再复制到 Word 上，以保证实验报告代码排版整齐美观。

（3）在完成实验过程中，遇到了特别多的问题，最后还好通过查阅了很多资料和请教别人基本都解决了，在遇到问题和解决问题之间的重复到现在还挺有成就感的。

（4）通过第 2 题的生产者-消费者问题，自己对于信号量机制和进程间的同步与通信有了更好的认识。学习使用了与共享内存相关的几个函数以及和信号量相关的几个函数的使用，并且对课本中的经典进程同步问题——生产消费者问题有了更深的认识和理解。

（5）第 3 题，基本实现了预计的 8 个 myshell 功能，不再对 shell 感到神秘莫测：因为自己也可以实现一个嘛。

（6）事实上，myshell 可以被当做一个真的 shell（它实际上只是一个 C 程序），可以在 myshell 中执行“./myshell”这个可执行程序，这个可执行程序会再产生一个 myshell，就可以在这个 myshell 中再次执行“./myshell”…。当我们多次执行./myshell 的时候，有时候可能连自己都记不清到底处在哪一层 myshell 中了，因为看起来它们都像是“真的”。有点类似电影《盗梦空间》中的多层空间，这是比较有趣的。

附录:第一大题代码

1.1.1 无名管道 pipe-demo.c

```
(1)  #include<stdio.h>
(2)  #include<stdlib.h>
(3)  #include<unistd.h>
(4)  #include<string.h>
(5)  #include<sys/types.h>
(6)
(7)  int main(){
(8)      pid_t pid;
(9)      int fds[2];
(10)      char buf[128];
(11)      int nwr = 0;
(12)
(13)      pipe(fds);                                    //
     打开两个文件描述符
(14)      pid = fork();
(15)      if( pid < 0 ){
(16)          printf("Error with fork\n");
(17)          exit(1);
(18)      }else if( pid == 0 ){
(19)          printf("This is child process, pid = %d\n", ge
     tpid());
(20)          printf("Child: waiting for message...\n");
(21)          close(fds[1]);                              //
      子进程关闭管道的写端
(22)          nwr = read(fds[0], buf, sizeof(buf));
(23)          printf("Child: receive %d bytes from parent: \
     "%s\"\n", nwr, buf);
(24)          exit(0);
(25)      }else{
(26)          printf("This is parent process, pid = %d\n", g
     etpid());
(27)          printf("Parent: sending message...\n");
(28)          close(fds[0]);                              //
      父进程关闭管道的读端
(29)          strcpy(buf, "Message from parent!");
(30)          nwr = write(fds[1], buf, strlen(buf)+1);
(31)          printf("Parent: send %d bytes to child\n", nwr
     );
(32)          wait(pid);
(33)          exit(0);
```

```
(34)      }
(35)  }
```

1.2.2 消息队列： msgtool.c

```
(36)  #include<stdio.h>
(37)  #include<stdlib.h>
(38)  #include<ctype.h>
(39)  #include<string.h>
(40)  #include<sys/types.h>
(41)  #include<sys/ipc.h>
(42)  #include<sys/msg.h>
(43)
(44)  #define MAX_SEND_SIZE 80
(45)
(46)  struct MyMsgBuf{
(47)      long msgType;
(48)      char msgContent[MAX_SEND_SIZE];
(49)  };
(50)
(51)  extern void SendMessage(int qid, struct MyMsgBuf* qbuf
      , long type, char* text);
(52)  extern void ReadMessage(int qid, struct MyMsgBuf* qbuf
      , long type);
(53)  extern void RemoveQueue(int qid);
(54)  extern void ChangeMessageQueueMode(int qid, char* mode
      );
(55)  extern void usage();
(56)
(57)  int main(int argc, char** argv){
(58)      key_t key;
(59)      int qid;
(60)      struct MyMsgBuf qbuf;
(61)
(62)      if( argc == 1 ){
(63)          usage();
(64)      }
(65)
(66)      key = ftok(".",'m');                              //
      通过 ftok 创建唯一的 key
(67)      qid = msgget(key, IPC_CREAT|0666);                //
      通过 key 创建对应的 IPC 对象
(68)      if( qid < 0 ){
```

```
(69)            perror("msgget");
(70)            exit(1);
(71)        }
(72)
(73)        switch (tolower(argv[1][0]))
(74)        {
(75)            case 's':
(76)                SendMessage(qid, &qbuf, atol(argv[2]), arg
    v[3]);
(77)                break;
(78)            case 'r':
(79)                ReadMessage(qid, &qbuf, atol(argv[2]));
(80)                break;
(81)            case 'd':
(82)                RemoveQueue(qid);
(83)            break;
(84)            case 'm':
(85)                ChangeMessageQueueMode(qid, argv[2]);
(86)            break;
(87)
(88)            default:
(89)                usage();
(90)            break;
(91)        }
(92)        return 0;
(93)  }
(94)
(95)  void SendMessage(int qid, struct MyMsgBuf* qbuf, long
    type, char* text){
(96)        printf("Sending a message\n");
(97)        qbuf->msgType = type;
(98)        strcpy(qbuf->msgContent, text);
(99)
(100)           if( msgsnd(qid, qbuf, strlen(qbuf->msgContent)+
    1, 0) == -1 ){
(101)               perror("msgsnd");
(102)               exit(1);
(103)           }
(104)           return;
(105)     }
(106)     void ReadMessage(int qid, struct MyMsgBuf* qbuf, lo
    ng type ){
(107)           printf("Reading a message\n");
(108)           qbuf->msgType = type;
```

```
(109)         msgrcv(qid, qbuf, MAX_SEND_SIZE, type, 0);
(110)         printf("Type: %ld Text: %s\n", qbuf->msgType, q
      buf->msgContent);
(111)         return;
(112)     }
(113)     void RemoveQueue(int qid){
(114)         msgctl(qid, IPC_RMID, 0);
(115)         return;
(116)     }
(117)     void ChangeMessageQueueMode(int qid, char* mode){
(118)         struct msqid_ds ds;
(119)
(120)         msgctl(qid, IPC_STAT, &ds);
(121)         sscanf(mode, "%ho", &ds.msg_perm.mode);
(122)
(123)         msgctl(qid, IPC_SET, &ds);
(124)         return;
(125)     }
(126)     void usage(){
(127)         fprintf(stderr, "MsgTool -A utility for tinkeri
      ng with msg queue\n");
(128)         fprintf(stderr, "nUsage: MsgTool s(end)\n");
(129)         fprintf(stderr, "        MsgTool r(ecv)\n");
(130)         fprintf(stderr, "        MsgTool d(elete)\n");

(131)         fprintf(stderr, "        MsgTool m(ode)\n");
(132)         exit(1);
(133)     }
```

1.2.3 共享内存

（1）shmget-demo.c

```
(134)     #include<sys/types.h>
(135)     #include<sys/ipc.h>
(136)     #include<sys/shm.h>
(137)     #include<stdlib.h>
(138)     #include<stdio.h>
(139)
(140)     #define BUFSIZE 4096
(141)
(142)     int main(int argc, char** argv){
(143)         int shm_id;
```

```
(144)
(145)         shm_id = shmget(IPC_PRIVATE, BUFSIZE, 0666);
(146)         if( shm_id < 0 ){
(147)             perror("shmget fail!\n");
(148)             exit(1);
(149)         }
(150)
(151)         printf("Successfully created segment: %d\n", sh
      m_id);
(152)         system("ipcs -m");
(153)
(154)         return 0;
(155)     }
```

(2) shmatt-write-demo.c

```
(156)     #include<sys/types.h>
(157)     #include<sys/ipc.h>
(158)     #include<sys/shm.h>
(159)     #include<stdlib.h>
(160)     #include<stdio.h>
(161)     #include<string.h>
(162)
(163)     int main(int argc, char** argv){
(164)         int shm_id;
(165)         char* shm_buf;
(166)
(167)         if( argc != 2 ){
(168)             fprintf(stderr, "USAGE: atshm <identifier>\
      n");
(169)             exit(1);
(170)         }
(171)         shm_id = atoi(argv[1]);
(172)
(173)         if( ( shm_buf = shmat(shm_id, 0, 0) ) < 0 ){
(174)             perror("Shmat fail!\n");
(175)             exit(1);
(176)         }
(177)
(178)         printf("Segment attached at %p\n", shm_buf);
(179)         system("ipcs -m");
(180)         strcpy(shm_buf, "Hello shared memory!\n");
(181)         getchar();
(182)
```

```
(183)         if( shmdt(shm_buf) < 0 ){
(184)             perror("Shmdt fail!\n");
(185)             exit(1);
(186)         }
(187)
(188)         printf("Segment detached\n");
(189)         system("ipcs -m");
(190)
(191)         getchar();
(192)         return 0;
(193)     }
```

(3)Shmatt-read-demo.c

```
(194)     #include<sys/types.h>
(195)     #include<sys/ipc.h>
(196)     #include<sys/shm.h>
(197)     #include<stdlib.h>
(198)     #include<stdio.h>
(199)     #include<string.h>
(200)
(201)     int main(int argc, char** argv){
(202)         int shm_id;
(203)         char* shm_buf;
(204)
(205)         if( argc != 2 ){
(206)             printf("USAGE: atshm <identifier>\n");
(207)             exit(1);
(208)         }
(209)
(210)         shm_id = atoi(argv[1]);
(211)         if( (shm_buf = shmat(shm_id, 0, 0)) < (char*)0
      ){
(212)             perror("shmat fail!\n");
(213)             exit(1);
(214)         }
(215)
(216)         printf("Segment attached at %p\n", shm_buf);
(217)         system("ipcs -m");
(218)         printf("The string in SHM is: %s\n", shm_buf);

(219)         getchar();
(220)
(221)         if( shmdt(shm_buf) < 0 ){
```

```
(222)            perror("shmdt fail!\n");
(223)            exit(1);
(224)        }
(225)        printf("Segment detached\n");
(226)        system("ipcs -m");
(227)
(228)        getchar();
(229)        return 0;
(230)    }
```

1.3.1 有名信号量：

(1)psem-named-open.c

```
(231)    #include<semaphore.h>
(232)    #include<unistd.h>
(233)    #include<stdio.h>
(234)    #include<stdlib.h>
(235)    #include<fcntl.h>
(236)
(237)    int main(int argc, char** argv){
(238)        sem_t* sem;
(239)
(240)        if( argc != 2 ){
(241)            printf("Please input a file name to act as
     the ID of the sem!\n");
(242)            exit(1);
(243)        }
(244)        sem = sem_open(argv[1], O_CREAT, 0644, 1);
(245)        exit(0);
(246)    }
```

(2) psem-named-wait-demo.c

```
(247)    #include<semaphore.h>
(248)    #include<unistd.h>
(249)    #include<stdio.h>
(250)    #include<stdlib.h>
(251)    #include<fcntl.h>
(252)
(253)    int main(int argc, char** argv){
(254)        sem_t* sem;
(255)        int val;
(256)
(257)        if( argc != 2 ){
```

```
(258)            printf("Please input a file name!\n");
(259)            exit(1);
(260)        }
(261)        sem = sem_open(argv[1], 0);
(262)        sem_wait(sem);
(263)        sem_getvalue(sem, &val);
(264)        printf("pid %d has semaphore, value = %d\n", ge
     tpid(), val);
(265)        return 0;
(266)    }
```

(3) psem-named-post-demo.c

```
(267)    #include<semaphore.h>
(268)    #include<unistd.h>
(269)    #include<stdio.h>
(270)    #include<stdlib.h>
(271)    #include<fcntl.h>
(272)
(273)    int main(int argc, char** argv){
(274)        sem_t *sem;
(275)        int val;
(276)
(277)        if( argc != 2 ){
(278)            printf("Please input a file name\n");
(279)            exit(1);
(280)        }
(281)        sem = sem_open(argv[1], 0);
(282)        sem_post(sem);
(283)        sem_getvalue(sem, &val);
(284)        printf("value = %d\n", val);
(285)        return 0;
(286)    }
```

(4)psem-named-unlike-demo.c

```
(287)    #include<sempahore.h>
(288)    #include<unistd.h>
(289)    #include<stdio.h>
(290)    #include<stdlib.h>
(291)    #include<fcntl.h>
(292)
(293)    int main(int argc, char** argv){
(294)        if( argc != 2 ){
(295)            printf("Please input the ID of sem!\n");
```

```
(296)            exit(1);
(297)        }
(298)        sem_unlink(argv[1]);
(299)        return 0;
(300)    }
```

1.3.3 互斥量：

(301) no-mutex-demo.c

```
(302)    #include<pthread.h>
(303)    #include<unistd.h>
(304)    #include<stdio.h>
(305)    #include<stdlib.h>
(306)    #include<malloc.h>
(307)
(308)    #define THREAD_NUM 16
(309)    #define MB 1024 * 1024
(310)
(311)    int *array;
(312)    int length;
(313)    int count;
(314)    int t;
(315)
(316)    int* count3s_thread(void* id){
(317)        int length_per_thread = length / THREAD_NUM;
(318)        int start = *(int*)id * length_per_thread;
(319)        int i;
(320)        for( i = start; i < start + length_per_thread ;
     i++ ){
(321)            if( array[i] == 3 ){
(322)                count = count + 1;
(323)            }
(324)        }
(325)    }
(326)    int main(int argc, char** argv){
(327)        int i;
(328)        int tid[THREAD_NUM];
(329)        pthread_t threads[THREAD_NUM];
(330)        length = 64 * MB;
(331)        array = malloc(length * sizeof(int));
(332)
(333)        for( i = 0; i < length; i++ ){
(334)            array[i] = i % 2 ? 4 : 3;
```

```
(335)         }
(336)         count = 0;
(337)         for( t = 0 ; t < THREAD_NUM ; t++ ){
(338)             tid[t] = t;
(339)             int err = pthread_create( &threads[t], NULL
    , (void*)count3s_thread, &tid[t] );
(340)             if( err ){
(341)                 printf("cread thread error!\n");
(342)                 exit(1);
(343)             }
(344)         }
(345)         for( t = 0 ; t < THREAD_NUM; t++ ){
(346)             pthread_join( threads[t], NULL );
(347)         }
(348)         printf("Total count = %d\n", count);
(349)         return 0;
(350)     }
```

(2) mutex-demo.c

```
1. #include<pthread.h>
2. #include<unistd.h>
3. #include<stdio.h>
4. #include<stdlib.h>
5. #include<malloc.h>
6.
7. #define THREAD_NUM 16
8. #define MB 1024 * 1024
9.
10. int *array;
11. int length;
12. int count;
13. int t;
14.
15. pthread_mutex_t m;
16.
17. int* count3s_thread(void* id){
18.     int length_per_thread = length / THREAD_NUM;
19.     int start = *(int*)id * length_per_thread;
20.     int i;
21.     for( i = start; i < start + length_per_thread ; i+
   + ){
22.         if( array[i] == 3 ){
23.             pthread_mutex_lock(&m);
```

```
            count++;
            pthread_mutex_unlock(&m);
        }
    }
}

int main(int argc, char** argv){
    int i;
    int tid[THREAD_NUM];
    pthread_t threads[THREAD_NUM];
    length = 64 * MB;
    array = malloc(length * sizeof(int));
    pthread_mutex_init(&m, NULL);

    for( i = 0; i < length; i++ ){
        array[i] = i % 2 ? 4 : 3;
    }
    count = 0;
    for( t = 0 ; t < THREAD_NUM ; t++ ){
        tid[t] = t;
        int err = pthread_create( &threads[t], NULL, (void*)count3s_thread, &tid[t] );
        if( err ){
            printf("cread thread error!\n");
            exit(1);
        }
    }
    for( t = 0 ; t < THREAD_NUM; t++ ){
        pthread_join( threads[t], NULL );
    }
    printf("Total count = %d\n", count);
    return 0;
}
```