

深圳大学实验报告

课程名称： 操作系统

实验项目名称： 综合实验 1

学 院： 计算机与软件学院

专 业： 计算机科学与技术

指导教师： 阮元

报告人： 刘睿辰 学号： 2018152051 班级： 数计班

实 验 时 间： 2021.4.22-2021.4.30

实验报告提交时间： 2021.4.30

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。

二、方法、步骤：

1. 学习实践：

- 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）：
 - 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/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）。
 - 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。

2. 设计简单的 shell 程序：

- 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）
- 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) 可以将 1/2 直接合并完成。
- 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：

1. 设计编写以下程序，着重考虑其同步问题：（代码复制在该部分后面）

- 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/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)。

- 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。

为了解决这个问题，我们首先应该构建公共缓冲区。为了让两个.c 文件都能使用公共缓冲区，我们可以写一个头文件来定义缓冲区。缓冲区中要规定缓冲区的大小，用 struct 结构体来进行规定，代码如图 1 所示。

```
1  #include <fcntl.h>
2  #include <sys/stat.h>
3  #include <semaphore.h>
4
5  #define LINE_SIZE 256
6  #define NUM_LINE 16
7
8  const char *queue_mutex = "queue_mutex";
9  const char *queue_empty = "queue_empty";
10 const char *queue_full = "queue_full";
11
12 /*生产者消费者公用的缓冲区*/
13 struct shared_mem_st
14 {
15     char buffer[NUM_LINE][LINE_SIZE]; //缓冲区
16     int line_write; //读指针
17     int line_read; //写指针
18 };
```

图 1. 公共缓冲区定义

接下来我们来编写生产者部分的代码。首先声明共享缓冲区、共享 id 以及访问缓冲区的指针。然后我们需要用 shmget 函数来分配一个 system V 共享内存段。我们要注意，生产者和消费者的第一个参数不能设置成 IPC_PRIVATE，因为这样的话二者都创建新的共享内容。所以我们将第一个参数设置为非 0 的相同值，这里都设置为 4。然后映射共享内存到进程空间，然后需要进行类型转换。代码如图 2 所示。

```
void *shared_memory = (void*)0; //缓冲区指针
struct shared_mem_st *shared_stuff;
char key_line[256]; //接收读入的字符，最大不超过LINE_SIZE
int shm_id; //共享内存id

// 创建key值
key_t key;
key = ftok("./ap", 4);
if (key < 0) // 创建key失败
{
    perror("fail ftok");
    exit(1);
}
shm_id = shmget(key, sizeof(struct shared_mem_st), 0666 | IPC_CREAT);
//创建共享内存
if (shm_id < 0)
{
    printf("shmget failed\n");
    exit(1);
}
if ((shared_memory = shmat(shm_id, 0, 0)) < (void*)0) //映射共享内存到进程空间
{
    printf("shmat failed\n");
    exit(1);
}
shared_stuff = (struct shared_mem_st*) shared_memory; //缓冲区指针类型转换
```

图 2. 设置共享内存参数

然后创建信号量。这里需要三个信号量，分别为互斥量、空缓冲区、满缓冲区信号量。当然这里面不存在互斥问题，互斥信号量是为了消费者进程使用的。然后初始化缓冲区写指针，这样我们才能按次序将信息写入缓冲区。然后开始输入字符串，用 reader 数组进行接收。reader 的值一旦等于 exit 那么就需要退出。

这部分的信号量主要是空缓冲区、满缓冲区信号量。由于在头文件中我们规定缓冲区为 16×256 大小的，所以我们将空缓冲区信号量初始值置为 16，满缓冲区信号量初始值置为 0，如图 3 所示。

```
//创建信号量,分别为访问共享内存的互斥量、空缓冲区、满缓冲区信号量
sem_t *sem_queue, *sem_queue_empty, *sem_queue_full;
sem_queue = sem_open("queue_mutex", O_CREAT, 0644, 1);
sem_queue_empty = sem_open("queue_empty", O_CREAT, 0644, 16);
sem_queue_full = sem_open("queue_full", O_CREAT, 0644, 0);
```

图 3. 设置信号量的初始值

接下来我们需要读入输入的信息。接收数组 reader 的大小设置为 256, 然后用 getchar() 方法进行读取。接收信息之后等待空缓冲区信号量，空缓冲区信号量减一代表空位少了一个，然后更新写指针。如果收到 exit 指令，程序需要退出。然后发送满缓冲区信号量，满缓冲区信号量加一代表当前被占用的缓冲池数量又多了一个。代码如下所示。

```
int flag = 1; //退出循环读取字符串的标志
//初始化读写指针
shared_stuff->line_write = 0;
shared_stuff->line_read = 0;
while (1)
{
    printf("Enter your text('quit' for exit):"); //输入提示

    i = -1;
    while((c = getchar()) != '\n')
    {
        key_line[++i] = c;
    }
    key_line[++i] = '\0';
    sem_wait(sem_queue_empty); //等待信号量输入
    strncpy(shared_stuff->buffer[shared_stuff->line_write], key_line, LINE_SIZE); //将reader的值写进
    //将行信息保存在缓冲区中
    if (strcmp(shared_stuff->buffer[shared_stuff->line_write], "quit\0") == 0) //输入quit则退出
    {
        flag = 0;
    }
    shared_stuff->line_write = (shared_stuff->line_write + 1) % NUM_LINE; //写指针增加，多了则从头开始
    sem_post(sem_queue_full); //发送信号量
    if (!flag)
    {
        break;
    }
}
```

图 4. 读取字符串并存入缓冲池

接下来我们要注意在最后依然要进行信号量接收和信号量释放的工作。由于消费者进程要产生子进程，所以如果生产者进程直接消失的话，消费者进程的子进程就不会收到消息，所以还要将最后一次信号，也就是将最后一次消息 exit 发送给消费者进程产生的子进程。代码如下所示。

```
sem_wait(sem_queue_empty);
strncpy(shared_stuff->buffer[shared_stuff->line_write], key_line, LINE_SIZE);
shared_stuff->line_write = (shared_stuff->line_write + 1) % NUM_LINE;
sem_post(sem_queue_full);
```

图 5. 最后一次消息 exit 发送给消费者进程产生的子进程

接下来我们来编写消费者部分的代码。这一部分的代码开始阶段与生产者部分代码是类似的，都要设置共享内存参数。然后就到了接收 producer 的信号量的部分。代码如图 6 所示。

```
//获取producer的三个信号量
sem_t *sem_queue, *sem_queue_empty, *sem_queue_full;
sem_queue = sem_open("queue_mutex", 0);
sem_queue_empty = sem_open("queue_empty", 0);
sem_queue_full = sem_open("queue_full", 0);
```

图 6. 获取 producer 信号量

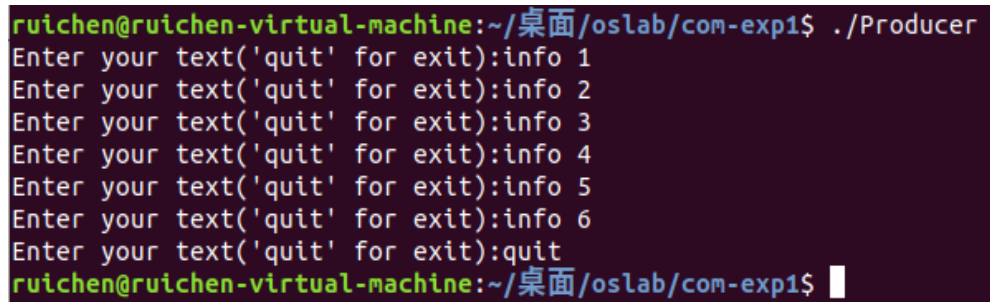
接下来通过 fork 方法创建子进程。然后对于父进程与子进程，我们有不同的操作：父进程先获取互斥信号量，此时互斥信号量为 0，所以子进程被阻塞，父进程先读取共享缓冲的内容，读取完毕之后释放互斥信号量；此时互斥信号量被释放，所以子进程可以占用互斥信号量，那么此时换成父进程被阻塞，子进程可以读取共享缓冲的内容。待子进程读取完毕之后，再释放互斥信号量，那么此时父进程又可以进行读取了。这样交替来进行，可以保证一次只有一个任务在操作缓冲区。而这样的设计可以导致奇数位置的信息被父进程读取，而偶数位置的信息被子进程读取。读取的进程应该更新读指针。以子进程为例，代码如图 7 所示。

```
while(1)
{
    sem_wait(sem_queue_full);
    sem_wait(sem_queue);

    printf("This is child process and its id is %d, it gets '%s'\n", getpid(), shared_stuff->buffer[shared_stuff->line_read]);
    //行信息打印
    if (strcmp(shared_stuff->buffer[shared_stuff->line_read], "quit\0") == 0) //exit退出
    {
        flag = 0;
    }
    strcpy(shared_stuff->buffer[shared_stuff->line_read], "", LINE_SIZE);
    shared_stuff->line_read = (shared_stuff->line_read + 1) % NUM_LINE;
    sem_post(sem_queue);
    sem_post(sem_queue_empty);
    if (flag == 0)
    {
        break; //退出循环
    }
}
sem_unlink(queue_mutex);
sem_unlink(queue_empty);
sem_unlink(queue_full);
```

图 7. 子进程与父进程通过互斥信号量来实现互斥

在 Linux 环境下运行 producer 之后运行 customer，可以看到奇数位置的信息被父进程捕获，偶数位置的信息被子进程捕获。最后的 exit 信息父子进程都将其打印了出来。结果如图 8 所示。



```
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./Producer
Enter your text('quit' for exit):info 1
Enter your text('quit' for exit):info 2
Enter your text('quit' for exit):info 3
Enter your text('quit' for exit):info 4
Enter your text('quit' for exit):info 5
Enter your text('quit' for exit):info 6
Enter your text('quit' for exit):quit
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$
```

图 8(a). 代码运行结果-生产者部分

```
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./Customer
This is parent process and its id is 8122, it gets 'info 1'
This is child process and its id is 8123, it gets 'info 2'
This is parent process and its id is 8122, it gets 'info 3'
This is child process and its id is 8123, it gets 'info 4'
This is parent process and its id is 8122, it gets 'info 5'
This is child process and its id is 8123, it gets 'info 6'
This is parent process and its id is 8122, it gets 'quit'
This is child process and its id is 8123, it gets 'quit'
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$
```

图 8(b). 代码运行结果-消费者部分

本部分的源代码:

shm_com_sem.h

```
1. #include <fcntl.h>
2. #include <sys/stat.h>
3. #include <semaphore.h>
4.
5. #define LINE_SIZE 256
6. #define NUM_LINE 16
7.
8. const char *queue_mutex = "queue_mutex";
9. const char *queue_empty = "queue_empty";
10. const char *queue_full = "queue_full";
11.
12. /*生产者消费者公用的缓冲区*/
13. struct shared_mem_st
14. {
15.     char buffer[NUM_LINE][LINE_SIZE]; //缓冲区
16.     int line_write; //读指针
17.     int line_read; //写指针
18. };
```

Producer.c

```
1. #include <unistd.h>
2. #include <stdlib.h>
3. #include <stdio.h>
4. #include <string.h>
5. #include <sys/shm.h>
6. #include <sys/types.h>
7. #include <sys/ipc.h>
8. #include <semaphore.h>
9. #include "shm_com_sem.h"
10.
```

```
11. int main()
12. {
13.     void *shared_memory = (void*)0; //缓冲区指针
14.     struct shared_mem_st *shared_stuff;
15.     char key_line[256]; //接收读入的字符, 最大不超过 LINE_SIZE
16.     int shm_id; //共享内存 id
17.
18.     // 创建 key 值
19.     key_t key;
20.     key = ftok("./ap", 4);
21.     if(key < 0) // 创建 key 失败
22.     {
23.         perror("fail ftok");
24.         exit(1);
25.     }
26.     shm_id = shmget(key, sizeof(struct shared_mem_st), 0666 | IPC_CREAT);
27.     //创建共享内存
28.     if (shm_id < 0)
29.     {
30.         printf("shmget failed\n");
31.         exit(1);
32.     }
33.     //映射共享内存到进程空间
34.     if ((shared_memory = shmat(shm_id, 0, 0)) < (void*)0)
35.     {
36.         printf("shmat failed\n");
37.         exit(1);
38.     }
39.     //缓冲区指针类型转换
40.     shared_stuff = (struct shared_mem_st*) shared_memory;
41.     //创建信号量, 分别为访问共享内存的互斥量、空缓冲区、满缓冲区信号量
42.     sem_t *sem_queue, *sem_queue_empty, *sem_queue_full;
43.     sem_queue = sem_open("queue_mutex", 0_CREAT, 0644, 1);
44.     sem_queue_empty = sem_open("queue_empty", 0_CREAT, 0644, 16);
45.     sem_queue_full = sem_open("queue_full", 0_CREAT, 0644, 0);
46.
47.     char c;
48.     int i;
49.     int flag = 1; //退出循环读取字符串的标志
50.     //初始化读写指针
51.     shared_stuff->line_write = 0;
52.     shared_stuff->line_read = 0;
53.     while (1)
54.     {
```

```

55.     printf("Enter your text('quit' for exit):");//输入提示
56.
57.     i = -1;
58.     while((c = getchar())!='\n')
59.     {
60.         key_line[++i] = c;
61.     }
62.     key_line[++i] = '\0';
63.     sem_wait(sem_queue_empty);        //等待信号量输入
64.     strncpy(shared_stuff->buffer[shared_stuff->line_write], key_line,
LINE_SIZE);    //将 reader 的值写进
65.     //将行信息保存在缓冲区中
66.     if (strcmp(shared_stuff->buffer[shared_stuff->line_write], "quit\0
") == 0)    //输入 quit 则退出
67.     {
68.         flag = 0;
69.     }
70.     shared_stuff->line_write = (shared_stuff->line_write + 1) % NUM_LI
NE;    //写指针增加, 多了则从头开始
71.     sem_post(sem_queue_full);    //发送信号量
72.     if (!flag)
73.     {
74.         break;
75.     }
76. }
77.
78.     sem_wait(sem_queue_empty);
79.     strncpy(shared_stuff->buffer[shared_stuff->line_write], key_line, LINE
_SIZE);
80.     shared_stuff->line_write = (shared_stuff->line_write + 1) % NUM_LINE;
81.     sem_post(sem_queue_full);
82.
83.     //释放信号量, 删除内存区域
84.     if (shmdt(shared_memory) < 0)
85.     {
86.         perror ( "shmdt");
87.         exit(1);
88.     }
89. }

```

Customer.c

```

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.
11. int main()
12. {
13.     void* shared_memory = (void*)0; //缓冲区指针
14.     struct shared_mem_st *shared_stuff;
15.     int shm_id;//共享内存 id
16.     pid_t fork_result;
17.     int flag = 1;
18.     // 创建 key 值
19.     key_t key;
20.     key = ftok("./ap",4);
21.     if(key < 0) // 创建 key 失败
22.     {
23.         perror("fail ftok");
24.         exit(1);
25.     }
26.     shm_id = shmget(key, sizeof(struct shared_mem_st), 0666 | IPC_CREAT);
27.     //创建共享内存
28.     if (shm_id < 0)
29.     {
30.         printf("shmget failed\n");
31.         exit(1);
32.     }
33.     if ((shared_memory = shmat(shm_id, 0, 0)) < (void*)0) //映射共享内存
        到进程空间
34.     {
35.         printf("shmat failed\n");
36.         exit(1);
37.     }
38.     shared_stuff = (struct shared_mem_st*) shared_memory; //缓冲区指针类型
        转换
39.
40.     //获取 producer 的三个信号量
41.     sem_t *sem_queue, *sem_queue_empty, *sem_queue_full;
42.     sem_queue = sem_open("queue_mutex", 0);
43.     sem_queue_empty = sem_open("queue_empty", 0);
```

```
44.     sem_queue_full = sem_open("queue_full", 0);
45.
46.     fork_result = fork();
47.
48.     if(fork_result < 0)
49.     {
50.         fprintf(stderr, "Fork failure\n");
51.     }
52.     else if (fork_result == 0) //子进程
53.     {
54.         while(1)
55.         {
56.             sem_wait(sem_queue_full);
57.             sem_wait(sem_queue);
58.
59.             printf("This is child process and its id is %d, it gets '%s'\n",
60.                 getpid(), shared_stuff->buffer[shared_stuff->line_read]);
61.             //行信息打印
62.             if (strcmp(shared_stuff->buffer[shared_stuff->line_read], "quit\0") == 0) //exit 退出
63.             {
64.                 flag = 0;
65.             }
66.             strncpy(shared_stuff->buffer[shared_stuff->line_read], "", LINE_SIZE);
67.             shared_stuff->line_read = (shared_stuff->line_read + 1) % NUM_LINE;
68.             sem_post(sem_queue);
69.             sem_post(sem_queue_empty);
70.             if (flag == 0)
71.             {
72.                 break; //退出循环
73.             }
74.             sem_unlink(queue_mutex);
75.             sem_unlink(queue_empty);
76.             sem_unlink(queue_full);
77.         }
78.     else //父进程
79.     {
80.         while (1)
81.         {
82.             sem_wait(sem_queue_full);
83.             sem_wait(sem_queue);
```

```

84.         printf("This is parent process and its id is %d, it gets '%s'\n", getpid(), shared_stuff->buffer[shared_stuff->line_read]);
85.         if (strcmp(shared_stuff->buffer[shared_stuff->line_read], "quit\0") == 0)
86.         {
87.             flag = 0;
88.         }
89.         strncpy(shared_stuff->buffer[shared_stuff->line_read], "", LINE_SIZE);
90.         shared_stuff->line_read = (shared_stuff->line_read + 1) % NUM_LINE;
91.         sem_post(sem_queue);
92.         sem_post(sem_queue_empty);
93.         if (flag == 0)
94.         {
95.             break;
96.         }
97.     }
98.     sem_unlink(queue_mutex);
99.     sem_unlink(queue_empty);
100.    sem_unlink(queue_full);
101. }
102. sleep(0.5);
103. exit(EXIT_SUCCESS);
104. }

```

2. 设计简单的 shell 程序

- 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）。

- 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。

我们编写 shell 指令应该遵循以下的原则，如图 9 所示。

接下来我们开始逐步说明程序的工作原理。

- 1) 首先是提示信息的输入。我们直到，Linux 命令行会有一部分提示信息，以我的虚拟机为例，提示信息为

ruichen@ruichen – virtual – machine:~/桌面/oslab/com – exp1\$

所以为了和原来的提示信息做区分，我们先要打印这部分信息。我们首先要了

解我们的文件路径是什么。标准 C 中的 `getcwd` 函数可以帮助我们获得当前的文件路径。

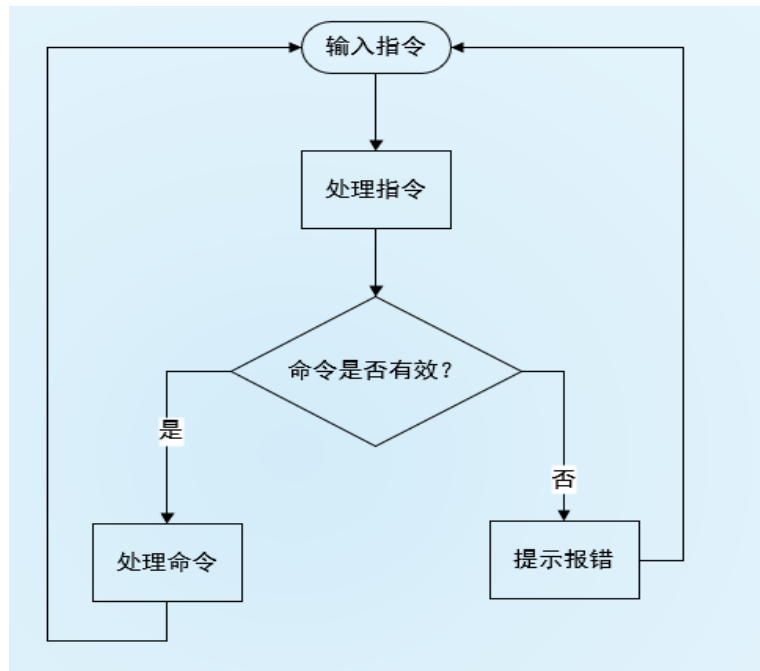


图 9. shell 命令执行流程

此外，`gethostname` 函数可以获得主机名，`getpwuid` 函数可以用来获得用户识别码，这样可以确定用户名称。有了用户名称、主机名以及绝对路径，我们就可以打印提示信息了。代码如下。

```
1. void prompt_info()
2. {
3.     char prompt[256];
4.     struct passwd *pwd;
5.
6.     //获取用户识别码,用来获得 username
7.     pwd = getpwuid(getuid());
8.
9.     char username[256], hostname[256], cwd[256];
10.
11.    //获得 username
12.    strcpy(username, pwd->pw_name);
13.
14.    //获得主机名
15.    gethostname(hostname, sizeof(hostname));
16.
17.    int i = 0, j = 0;
18.
19.    //ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$
```

```

20.  //[--myshell--][ruichen@ruichen-virtual-machine com-exp1]$
21.  //获得文件绝对路径 (eg. /home/ruichen/桌面/oslab/com-exp1)
22.  getcwd(cwd, sizeof(cwd));
23.  char *p[256];
24.  p[0]= strtok(cwd, "/");
25.  i = 0;
26.  while (p[i] != NULL)
27.  {
28.      i++;
29.      p[i]= strtok(NULL, "/");
30.  }
31.
32.  //拼接字符串
33.  strcpy(prompt, "[MyShell]");
34.  strcat(prompt, username);
35.  strcat(prompt, "@");
36.  strcat(prompt, hostname);
37.  strcat(prompt, " ");
38.  for (int j = 0; j < i; j++)
39.  {
40.      strcat(prompt, p[j]);
41.      if (j != i - 1)
42.      {
43.          strcat(prompt, "/");
44.      }
45.  }
46.  strcat(prompt, "]#");
47.  /*
48.  字体色
49.  30    Black
50.  31    Red
51.  32    Green
52.  33    Yellow
53.  34    Blue
54.  35    Magenta
55.  36    Cyan
56.  37    White*/
57.  printf("\e[1;31;40m %s \e[0m", prompt);
58. }

```

特别提到一点，C 标准库中 `strtok` 函数用于分割字符串。这里面我们根据文件路径进行分割，可以将每一层文件夹名称都提取出来。

此外，第 57 行的输出是为了使得我们的提示信心有别于 Ubuntu 自带的信息的

格式，我们这里用的是红色字体以及黑色底色，输出效果和自带的信息格式是不同的，效果如图 10 所示。

```
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]#  
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$
```

图 10. 输出信息格式

- 2) 接下来，我们需要为内部指令做准备。首先我们知道，Linux 内部命令像 `exit`，`help` 这些都属于内置命令，所以我们要想在我们自己的 shell 里面调用的话，必须重写这些方法。像 `exit` 比较容易，但是 `help` 指令需要打印出很多提示信息。方便起见，我们在 Linux 先运行 `help` 指令，复制下所有的内容，然后保存进一个 `txt` 文件中。然后当用户输入 `help`(这里指的是内置命令 `help`，不是外部命令 `help`)寻求帮助的时候，对这个文件进行读取即可。代码如下所示。

```
1. void printHelp()  
2. {  
3.     FILE *fp;  
4.     fp = 0;  
5.     //切换模式，改成只读  
6.     if((fp = fopen("./help.txt", "r")) == 0)  
7.     {  
8.         printf("文件打开失败!\n");  
9.         return -1;  
10.    }  
11.    char strBuf[1000];  
12.    memset(strBuf, 0, sizeof(strBuf));  
13.    while(1)  
14.    {  
15.        if(fgets(strBuf, sizeof(strBuf), fp) == 0)  
16.        {  
17.            break;  
18.        }  
19.        printf("%s", strBuf);  
20.    }  
21.    fclose(fp);  
22.    return;  
23. }
```

我们使用 C 语言的 `fopen` 函数并设置参数为 `r`(只读)来进行读取。将读取的内容全部输出，就可以起到 `help` 的作用。

- 3) 接下来我们来处理外部命令。外部命令的话，我们需要用到一个函数 `execvp`，其函数原型为：

`int execvp(const char* file, const char* argv[])`, 第一个参数是要运行的文件, 会在环境变量 `PATH` 中查找 `file`, 并执行; 第二个参数, 是一个参数列表, 也就是要执行的部分, 参数列表最后要以 `NULL` 结尾。执行失败的话会返回-1。

这部分我们以 `argv[0]`为指令内容进行执行, 如果执行失败就返回错误提示, 代表这是一个非法的指令。代码如下。

```
1. int execcmd(char* argv[])
2. {
3.     pid_t pid = fork();
4.     if(pid == 0)
5.     {
6.         if(argv[0])
7.         {
8.             if(execvp(argv[0], argv) == -1)
9.             {
10.                printf("Invalid command!\n");
11.            }
12.        }
13.        exit(-1);
14.    }
15.    else if(pid>0)
16.    {
17.        wait(NULL);
18.    }
19.    else
20.    {
21.        perror("error");
22.        return 0;
23.    }
24.    return 1;
25. }
```

- 4) 接下来我们读取指令。这里我们分三种情况进行讨论, 也就是正常字符、重定向符号以及空格。如果是正常字符, 那么就正常读入即可; 如果是空格, 我们需要将空格左右两端的指令分别记录, 保存在两个数组之中, 这样方便后续的处理; 如果是重定向符号, 我们需要更改标志值 `p`, 处理方法在后面会叙述。代码如图 11 所示。

```

//分割字符串
while((c=getchar())!='\n')
{
    if(c == '|') //读到重定向符号
    {
        p = i;
        i++;
    }
    else if(c != ' ') //正常非空格字符，正常读取
    {
        reader[j] = c;
        j++;
    }
    else //空格字符
    {
        reader[j] = '\0';
        strcpy(reader1[i], reader);
        strcpy(reader2[i], reader1[i]);
        reader2[i][j] = 0;
        i++;
        j = 0;
    }
}

```

图 11. 对输入指令的处理

- 5) 指令的处理：这里分为有管道和无管道的命令的分别处理。对于无管道的命令，我们可以直接来通过重写内置函数或者调用系统指令的方法来执行。例如，像 exit、help 和 cd 这种命令，我们可以进行重写，代码如图 12 所示。

```

if(!strcmp("exit", reader1[0]))
{
    printf("Bye from myshell.\n");
    exit(EXIT_SUCCESS);
}
else if(!strcmp("echo", reader1[0]))
{
    printf("%s\n", reader1[1]);
}
else if(!strcmp("cd", reader1[0]))
{
    if(chdir(reader1[1]) < 0)
    {
        printf("No such path or directory! Please check your spelling.\n");
    }
}
else if(!strcmp("help", reader1[0]))
{
    printHelp();
}

```

图 12. 重写内置函数方法

但是如果涉及到外部命令，我们可以通过之前叙述过的 `execvp` 命令来进行调用。所以这部分就调用 `execmd` 函数即可。注意，这里面我们将指令放置进了 `reader1` 和 `reader2` 中，所以调用该函数的话，参数自然就是 `reader1` 或者 `reader2`。

那么如果是有管道的程序，执行方法可以用图 13 来形象地表示。

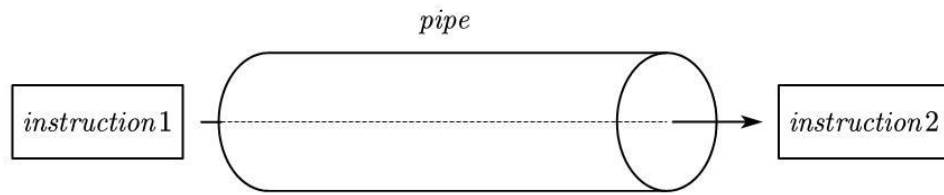


图 13. 管道程序执行

所以首先将重定向符号两端的指令 `instruction1` 和 `instruction2` 进行分隔，代码如下所示。

```
pid_t pid1 = 0;
int fds[2];
char *des1[256], *des2[256];
pipe(fds); //创建管道, fds存储文件描述符

for(j=0; j<p; j++)
{
    des1[j] = reader2[j]; // cmd1
}
des1[p] = 0;

for(j=p; j<i; j++)
{
    des2[j] = reader2[j]; // cmd2
}
```

图 14. 将重定向符号两端的指令进行分隔

然后通过 `fork` 生成子进程。这里面如果是子进程，那么应该关闭读端，通过写端 `fds[1]` 写入数据；如果是父进程，那么再生成一个子进程，通过读端 `fds[0]` 读出数据。读完数据之后随即关闭读端。

我们不能直接用父进程来读取数据，因为父进程需要不断执行来接收指令。代码如下所示。

```
1. else if(pid1 == 0) //子进程, 通过 stdout 输出流写入 fds[1]
2. {
3.     close(fds[0]);
4.     dup2(fds[1], stdout);
5.     execvp(des1[0], des1); //执行 execve()系统调用来执行 dir 命令
6.     close(fds[1]);
7. }
8. else //父进程
9. {
10.    waitpid(pid1, NULL, 0);
11.    pid_t pid2 = fork();
12.    if(pid2 < 0)
```

```
13.     {
14.         printf("Fork error!\n");
15.     }
16.     else if(pid2 == 0) // 子进程 2
17.     {
18.         close(fds[1]);
19.         dup2(fds[0], stdin); //通过 stdin 输入流读取管道的读端提供的数据
20.         execvp(des2[0], des2); //执行 execve() 系统调用来执行 dir 命令
21.         close(fds[0]);
22.     }
23.     else
24.     {
25.         close(fds);
26.         wait(pid2);
27.     }
28. }
```

6) 代码运行结果展示部分

内部命令:

help 指令运行结果, 如图 15 所示。

```
[MyShell]#truten@truten-virtual-machine home/ubuntu/桌面/ostlab/com-exp1# help
GNU bash, 版本 4.4.20(1)-release (x86_64-pc-linux-gnu)
这些 shell 命令是内部定义的。请输入 'help' 以获取一个列表。
输入 'help 名称' 以得到有关函数 '名称' 的更多信息。
使用 'info bash' 来获得关于 shell 的更多一般性信息。
使用 'man -k' 或 'info' 来获取不在列表中的命令的更多信息。

名称旁边的星号(*)表示该命令被禁用。

job_spec [&]
(( 表达式 ))
: 文件名 [参数]
:
[ 参数... ]
[[ 表达式 ]]
alias [-p] [名称[=值] ... ]
bg [任务声明 ...]
bind [-lpvsPSUX] [-m 键映射] [-f 文件名] [-q]
break [n]
builtin [shell 内建 [参数 ...]]
caller [表达式]
case 词 in [模式 [...]] 命令 ;;... esac
cd [-L][-P [-e]] [-@] [目录]
command [-pvf] 命令 [参数 ...]
cmpgen [-abdcfgjksuv] [-o 选项] [-A 动作] >
complete [-abdcfgjksuv] [-pr] [-DE] [-o 选项]
comptop [-o]o 选项 [-DE] [名称 ...]
continue [n]
coproc [名称] 命令 [重定向]
declare [-aAfGgIlNrtux] [-p] [名称[=值] ...]
dirs [-clpv] [+N] [-N]
dtsown [-h] [-ar] [jobspec ... | pid ...]
echo [-neE] [参数 ...]
enable [-a] [-dnps] [-f 文件名] [名称 ...]
eval [参数 ...]
exec [-cl] [-a 名称] [命令 [参数 ...]] [重定]
exit [n]
export [-fn] [名称[=值] ...] 或 export -p
false
fc [-e 编辑器名] [-lnr] [起始] [终结] 或 fc ->
fg [任务声明]
for 名称 [in 词语 ... ] ; do 命令; done
for (( 表达式1; 表达式2; 表达式3 )); do 命令; >
function 名称 { 命令 ; } 或 name () { 命令 ; >
getopts 选项字符串 名称 [参数]
hash [-lr] [-p 路径名] [-dt] [名称 ...]
help [-dms] [模式 ...]

history [-c] [-d 偏移量] [n] 或 history -anr>
if 命令; then 命令; [ elif 命令; then 命令; >
jobs [-lnprs] [任务声明 ...] 或 jobs -x 命令>
kill [-s 信号声明 | -n 信号编号 | -信号声明>
let 参数 [参数 ...]
local [option] 名称[=值] ...
logout [n]
mapfile [-d 分隔符] [-n 计数] [-O 起始序号] >
popd [-n] [+N] [-N]
printf [-v var] 格式 [参数]
pushd [-n] [+N] [-N | 目录]
pwd [-LP]
read [-ers] [-a 数组] [-d 分隔符] [-i 缓冲>
readarray [-n 计数] [-O 起始序号] [-s 计数] >
readonly [-aaf] [名称[=值] ...] 或 readonly >
return [n]
select NAME [in 词语 ... ] do 命令; done
set [-abefghkmnptuvxBCHP] [-o 选项名] [--] >
shift [n]
shopt [-pgsu] [-o] [选项名 ...]
source 文件名 [参数]
suspend [-f]
test [表达式]
time [-p] 管道
times
trap [-lp] [[参数] 信号声明 ...]
true
type [-afptP] 名称 [名称 ...]
typeset [-aAfGgIlNrtux] [-p] 名称[=值] ...
ulimit [-SHabcdefiklmnpqrstuvXT] [限制]
umask [-p] [-S] [模式]
unalias [-a] 名称 [名称 ...]
unset [-f] [-v] [-N] [名称 ...]
until 命令; do 命令; done
variables - 一些 shell 变量的名称和含义
wait [-n] [编号 ...]
while 命令; do 命令; done
{ 命令 ; }
```

图 15. 内部命令-help 指令

exit 指令运行结果，如图 16 所示，程序已经退出。

```
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]# exit
Bye from myshell.
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$
```

图 16. 内部命令-exit 指令

外部命令：

cd 指令和 ls 指令综合运用，可以看到我们可以前往不同的路径，提示信息也会发生变化，同时也会展示出不同的文件，如图 17 所示。

```
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]# cd ..
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab]# ls
com-exp1  exp1
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab]# cd com-exp1
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]# ls
ap          msgtool      no-mutex-demo  Producer.c    psem-named-wait-demo.c  shmatt-write-demo.c
Customer    msgtool.c    no-mutex-demo.c  psem-named-open  shared_buffer.h.gch     shm_com_sem.h
Customer.c  mutex-demo   os-exp-fifo     psem-named-open.c  shared_buffer_src.h     shmget-demo
help.txt    mutex-demo.c pipe-demo       psem-named-post-demo  shmatt-read-demo       shmget-demo.c
line_read   myshell      pipe-demo.c     psem-named-post-demo.c  shmatt-read-demo.c
line_write  myshell.c    Producer        psem-named-wait-demo  shmatt-write-demo
```

图 17. 内部命令-cd 指令和 ls 指令

再来看 cp(copy)命令，首先我们在当前文件夹下执行我们之前的 Producer.c 文件的复制，然后编译运行复制版本的文件，可以看到结果是一样的，cp 指令工作正常，如图 18 所示。

```
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]# cp Producer.c Producer_dup.c
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]# gcc Producer_dup.c -o Producer_dup -lpthread
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]# ./Producer_dup
Enter your text('quit' for exit):1
Enter your text('quit' for exit):2
Enter your text('quit' for exit):quit
```

图 18. 内部命令-cp 指令（1）

再来试一下带参数的 cp 操作。以参数-r（复制整个文件夹）为例，我们先返回上一层文件夹，然后复制两份文件，如图 19 所示。再进入新的文件夹，用 ls 指令，可以看到与原文件中文件内容是一样的，如图 20 所示。

```
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]# cd ..
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab]# ls
com-exp1  exp1
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab]# cp -r com-exp1 com-exp1-dup
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab]# ls
com-exp1  com-exp1-dup  exp1
```

图 19. 内部命令-cp 指令（2）

```
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab]# cd com-exp1-dup
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1-dup]# ls
ap          msgtool      no-mutex-demo  Producer.c    psem-named-post-demo.c  shmatt-read-demo.c
Customer    msgtool.c    no-mutex-demo.c  Producer_dup  psem-named-wait-demo.c  shmatt-write-demo
Customer.c  mutex-demo   os-exp-fifo     Producer_dup.c  psem-named-wait-demo.c  shmatt-write-demo.c
help.txt    mutex-demo.c pipe-demo       psem-named-open  shared_buffer.h.gch     shm_com_sem.h
line_read   myshell      pipe-demo.c     psem-named-open.c  shared_buffer_src.h     shmget-demo
line_write  myshell.c    Producer        psem-named-post-demo  shmatt-read-demo       shmget-demo.c
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1-dup]# cd ..
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab]# cd com-exp1
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]# ls
ap          msgtool      no-mutex-demo  Producer.c    psem-named-post-demo.c  shmatt-read-demo.c
Customer    msgtool.c    no-mutex-demo.c  Producer_dup  psem-named-wait-demo.c  shmatt-write-demo
Customer.c  mutex-demo   os-exp-fifo     Producer_dup.c  psem-named-wait-demo.c  shmatt-write-demo.c
help.txt    mutex-demo.c pipe-demo       psem-named-open  shared_buffer.h.gch     shm_com_sem.h
line_read   myshell      pipe-demo.c     psem-named-open.c  shared_buffer_src.h     shmget-demo
line_write  myshell.c    Producer        psem-named-post-demo  shmatt-read-demo       shmget-demo.c
```

图 20. 内部命令-cp 指令（3）

echo 操作：如图 21 所示。

```
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]# echo 13
13
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]# echo 'hey!'
'hey!'
```

图 21. echo 命令的执行

可执行程序：

我们以 HelloWorld 程序为例，结果如图 22 所示。

```
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]# gedit HelloWorld.c
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]# gcc HelloWorld.c -o HelloWorld
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]# ./HelloWorld
Hello world!
```

图 22. 可执行文件的执行

无效命令：以拼写错误、违法字符以及前往不存在的路径为例，结果如图 23 所示。

```
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]# invalidcommand
invalid command!
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]# ??
invalid command!
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]# cd noexistdict
No such path or directory! Please check your spelling.
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]# gedit HelloWorld.c
invalid command!
```

图 23. 违法命令的报错信息

shell 程序支持管道功能：如图 24 所示。

```
[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]# dir | more
ap          line_read  myshell    pipe-demo.c  psem-named-open.c  shared_buffer_src.h  shmget-demo
Customer    line_write myshell.c   Producer      psem-named-post-demo  shmatt-read-demo     shmget-demo.c
Customer.c  msgtool   no-mutex-demo  Producer.c    psem-named-post-demo.c  shmatt-read-demo.c
HelloWorld  msgtool.c no-mutex-demo.c  Producer_dup  psem-named-wait-demo    shmatt-write-demo
HelloWorld.c mutex-demo os-exp-fifo    Producer_dup.c  psem-named-wait-demo.c  shmatt-write-demo.c
help.txt    mutex-demo.c pipe-demo      psem-named-open  shared_buffer.h.gch     shm_com_sem.h

[MyShell][ruichen@ruichen-virtual-machine home/ruichen/桌面/oslab/com-exp1]# ls | more
ap          line_read  myshell    pipe-demo.c  psem-named-open.c  shared_buffer_src.h  shmget-demo
Customer    line_write myshell.c   Producer      psem-named-post-demo  shmatt-read-demo     shmget-demo.c
Customer.c  msgtool   no-mutex-demo  Producer.c    psem-named-post-demo.c  shmatt-read-demo.c
HelloWorld  msgtool.c no-mutex-demo.c  Producer_dup  psem-named-wait-demo    shmatt-write-demo
HelloWorld.c mutex-demo os-exp-fifo    Producer_dup.c  psem-named-wait-demo.c  shmatt-write-demo.c
help.txt    mutex-demo.c pipe-demo      psem-named-open  shared_buffer.h.gch     shm_com_sem.h
```

图 24. shell 程序支持管道功能

Linux 进程间通信内容部分的学习：

1. 管道

进程间的管道通信有两种形式，无名管道用于父子进程间，命名管道可以用于任意进程间，因为命名管道在文件系统中具有可访问的路径名。管道通信方式主要用于单向通信，如果需要双向通信则建立两条相反方向的管道。管道实质是由内核管理的一个缓冲区（一边由进程写入，另一边由进程读出），因此要注意，如果缓冲区满了则写管道的进程将会阻塞。另外管道内部没有显式的格式和边界，需要自行处理消息边界，如果多进程间共享还需要处理传送目标等工作。

1) 无名管道：管道是单向的信道，进程从管道的写端口写入数据，需要数据的进程从读端口中获取数据，数据在管道中按到达顺序流动。Unix 命令中使用“|”来连接两个命令时使用的就是管道，例如“ls | more”将 ls 命令的标准输出内容写入到管道中，管道的输出内容作为 more 命令的标准输入。

管道比较灵活，能看到管道所对应的文件描述符的进程之间都可以使用，但是务必注意同步关系。另外用户要关闭不使用的管道端口，否则可能会出现异常情况。C 库中的用户态函数 popen() 和 pclose() 对 pipe() 系统调用进行了封装，更方便和安全。如下代码所示，代码 pipe-demo.c 以 pipe() 为例展示父子进程间使用管道进行通信的方法，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。

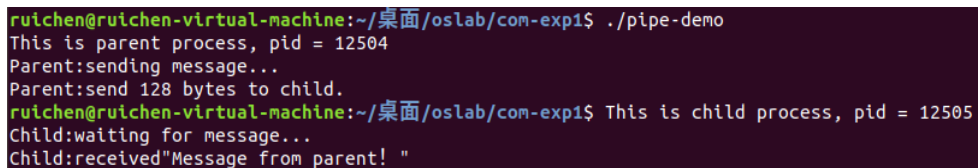
```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <sys/types.h>
5. int main()
6. {
7.     pid_t pid = 0;
8.     int fds[2];
9.     char buf[128];
10.    int nwr = 0;
11.    pipe(fds); //在 fork()前执行
12.    pid = fork();
13.    if(pid < 0)
14.    {
15.        printf("Fork error!\n");
16.        return -1;
17.    }
18.    else if(pid == 0)
```

```

19.  {
20.      printf("This is child process, pid = %d\n", getpid());
21.      printf("Child:waiting for message...\n");
22.      close(fds[1]); //关闭写端 fds[1]
23.      nwr = read(fds[0], buf, sizeof(buf)); //从读端 fds[0]读入数据
24.      printf("Child:received\"%s\"\n", buf);
25.  }
26.  else
27.  {
28.      printf("This is parent process, pid = %d\n", getpid());
29.      printf("Parent:sending message...\n");
30.      close(fds[0]); //关闭读端 fds[0]
31.      strcpy(buf, "Message from parent! ");
32.      nwr = write(fds[1], buf, sizeof(buf)); //往写端 fds[1]写出数据
33.      printf("Parent:send %d bytes to child.\n", nwr);
34.  }
35.  return 0;
36. }

```

我们在 Linux 系统下运行该代码，结果如图 25 所示。



```

ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./pipe-demo
This is parent process, pid = 12504
Parent:sending message...
Parent:send 128 bytes to child.
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ This is child process, pid = 12505
Child:waiting for message...
Child:received"Message from parent! "

```

图 25. pipe-demo.c 运行结果

根据 pipe-demo.c 的结果，我们看到父进程发送了消息到管道，子进程成功接受到了 *Message from parent* 这一信息。

2) 命名管道

无名管道有一个主要缺点，那就是只能通过父子进程之间（及其后代）使用文件描述符的继承来访问，无法在任意的进程之间使用。命名管道（named pipe）或者叫 FIFO 则突破了这个限制。可以说 FIFO 就是无名管道的升级版，有可访问的磁盘索引节点，即 FIFO 文件将出现在目录树中（不像无名管道那样只存在于 pipefs 特殊文件系统中）。

下面我们用 mkfifo 命令来创建命名管道 os-exp-fifo，如图 26 所示，其中 ls 命令查看其类型是管道“p”。

```
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ mkfifo os-exp-fifo
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ls -l os-exp-fifo
prw-r--r-- 1 ruichen ruichen 0 4月 22 19:59 os-exp-fifo
```

图 26. mkfifo 创建命名管道

我们可以尝试从管道读入数据。我们使用的是 `cat` 指令，如图 27 所示。此时管道中没有信息，`cat` 进入阻塞状态。

```
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ cat os-exp-fifo
```

图 27. `cat` 读取空的管道文件形成阻塞

为了使管道中有数据，我们开启另一个终端，然后用 `echo Hello, Named PIPE! >os-exp-fifo` 写入数据，此时 `cat` 会被唤醒，回显字符串 *Hello, Named PIPE!*，如图 28 所示。

```
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ echo Hello,Named PIPE! >os-exp-fifo
```

图 28(a). 开启另一个终端向管道写入数据

```
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ cat os-exp-fifo
Hello,Named PIPE!
```

图 28(b). `cat` 被唤醒

在程序中使用命名管道的方法和普通文件差不多，只是创建时不能像普通文件那样直接用 `open()` 创建，而是需要使用 `mkfifo()` 函数。函数 `mkfifo()` 的函数原型如图 29 所示。

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char * pathname, mode_t mode);
```

图 29. `mkfifo` 函数原型

函数 `mkfifo()` 会依参数 `pathname` 创建特殊的 FIFO 文件（如果已经存在则创建失败），而参数 `mode` 为该文件的权限（`umask` 值也会影响到 FIFO 文件的权限）。`mkfifo()` 建立的 FIFO 文件后，本进程或其他进程都可以用读写一般文件的方式存取。

2. System V IPC

Linux 的进程通信继承了 System V IPC。System V IPC 指的是 AT&T 在 System V.2 发行版中引入的三种进程间通信工具：

- (1) 信号量，用来管理对共享资源的访问；
- (2) 共享内存，用来高效地实现进程间的数据共享；

(3) 消息队列，用来实现进程间数据的传递。

把这三种工具统称为 System V IPC 的对象，每个对象都具有一个唯一的 IPC 标识符 ID。为了使不同的进程能够获取同一个 IPC 对象，必须提供一个 IPC 关键字（IPC key），内核负责把 IPC 关键字转换成 IPC 标识符 ID。下面我们观察这三种 IPC 工具。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，如图 30 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。

```
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ 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      目标
0x00000000  557056      ruichen  600        524288        2        2        目标
0x00000000  294913      ruichen  600       33554432        2        2        目标
0x00000000  393218      ruichen  600        524288        2        2        目标
0x00000000  2162691     ruichen  600        524288        2        2        目标
0x00000000  524292      ruichen  600        524288        2        2        目标
0x00000000  1802245     ruichen  600        524288        2        2        目标
0x00000000  1900550     ruichen  600       16777216        2        2        目标
-----
----- 信号量数组 -----
键          semid      拥有者  权限      nsems
-----
```

图 30. `ipcs` 命令的输出

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 31 所示。

```
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ipcs -ql

----- 消息限制 -----
系统最大队列数量 = 32000
最大消息尺寸（字节）= 8192
默认的队列最大尺寸（字节）= 16384
```

图 31. `ipcs -ql` 命令的输出

删除这些 IPC 对象的命令是 `ipcrm`，它会将与 IPC 对象及其相关联的数据也一起删除，管理员或者 IPC 对象的创建者才能执行删除操作。该命令可以使用 IPC 键或者 IPC 的 ID 来指定 IPC 对象：

- 1) `ipcrm -M shmkey` 删除用 `shmkey` 创建的共享内存段而 `ipcrm -m shmid` 删除用 `shmid` 标识的共享内存段；
- 2) `ipcrm -Q msgkey` 删除用 `msgkey` 创建的消息队列而 `ipcrm -q msqid` 删除用

msgid 标识的消息队列；

- 3) ipcrm -S semkey 删除用 semkey 创建的信号而 ipcrm -s semid 删除用 semid 标识的信号。

下面我们来看消息队列、共享内存以及信号量数组/信号量集。

1) 消息队列

消息队列有些像邮政中的邮箱，里面的消息有点像信件。由于各条消息可以通过类型（type）进行区分，因此可以用于多个进程间通信。比如一个任务分派进程，创建了若干个执行子进程，不管是父进程发送分派任务的消息，还是子进程发送任务执行的消息，都将 type 设置为目标进程的 PID，目标进程只接收消息类型为 type 的消息就实现了子进程只接收自己的任务，父进程只接收任务结果。

下面给出一个多功能的消息队列操作程序 msgtool.c。由于 msgtool 每次启动都是以新的进程形式运行，因此各次运行间是相互独立的，为了能访问到指定的消息队列，我们需一个外部的“键”转换为内部的消息队列的 ID，例如此处使用的是当前目录“.”并通过 ftok() 转换成内部 ID。msgtool.c 主函数中首先打开（或创建）一个消息队列 msgget(key, IPC_CREAT|0660)，第一个参数就是前面通过 ftok() 将键转换而来的消息队列 ID，第二个参数类似于文件打开的参数——IPC_CREAT 表示消息队列，若是还不存在则创建一个新的。0660 表示创建者及其同组用户可以读（收）也可以写（发）消息。然后根据命令行参数调用不同操作函数，

如果是“s”则调用 send_message() 发送一条消息；

如果是“r”则调用 read_message() 接受一条消息；

如果是“d”则调用 remove_queue() 删除指定的消息队列；

如果是“m”则调用 change_queue_mode() 改变消息队列的访问模式。

其中发送消息的核心函数是 msgsnd()，第一个参数是消息队列的 ID，第二个参数是被发送消息的起始地址（消息的第一个成员是一个整数用于指出消息类型），第三个参数是消息长度，第四个参数指定写消息时的一些行为（此例子用 0）；接受消息的函数是 msgrcv()，第一个参数用于指定消息队列的 ID，第二个参数是接受缓冲区地址，第三个参数指出希望接受的消息类型（0 表示接受任意类型的一条消息，>0 表示接受指定类型的消息，<0 则表示接受类型数值小于该数字绝对值的一条信息）；删除和修改访问模式都是使用 msgctl()（分别指出操作为 IPC_RMID 或 IPC_SET）。

代码如下：

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

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

```

#include <ctype.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX_SEND_SIZE 80
struct mymsgbuf //消息的结构体
{
    long mtype; //消息类型
    char mtext[MAX_SEND_SIZE]; //消息内容
};
void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text);
void read_message(int qid, struct mymsgbuf *qbuf, long type);
void remove_queue(int qid);
void change_queue_mode(int qid, char *mode);
void usage(void);
int main(int argc, char *argv[])
{
    key_t key;
    int msgqueue_id;
    struct mymsgbuf qbuf;
    if(argc == 1)
        usage();
    /* Create unique key via call to ftok() */
    key = ftok(".", 'm');
    /* Open the queue - create if necessary */
    if((msgqueue_id = msgget(key, IPC_CREAT|0660)) == -1)
    {
        perror("msgget");
        exit(1);
    }
    switch(tolower(argv[1][0]))
    {
        case 's':

```

```

        send_message(msgqueue_id, (struct mymsgbuf*)&qbuf, atol(argv[2]), argv[3]);
        break;
        case 'r':
            read_message(msgqueue_id, &qbuf, atol(argv[2]));
            break;
        case 'd':
            remove_queue(msgqueue_id);
            break;
        case 'm':
            change_queue_mode(msgqueue_id, argv[2]);
            break;
        default:
            usage();
    }
    return(0);
}

void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text)
{
    /* Send a message to the queue */
    printf("Sending a message \n");
    qbuf->mtype = type; //填写消息的类型
    strcpy(qbuf->mtext, text); //填写消息内容
    if((msgsnd(qid, (struct msgbuf *)qbuf, strlen(qbuf->mtext)+1, 0)) == -1)
    {
        perror("msgsnd");
        exit(1);
    }
    return;
}

void read_message(int qid, struct mymsgbuf *qbuf, long type)
{
    /* Read a message from the queue */
    printf("Reading a message \n");
    qbuf->mtype = type;

```

```

msgrcv(qid, (struct msgbuf *)qbuf, MAX_SEND_SIZE, type, 0);
printf("Type: %ld Text: %s\n", qbuf->mtype, qbuf->mtext);
return;
}

void remove_queue(int qid)
{
    /* Remove the queue */
    msgctl(qid, IPC_RMID, 0);
    return;
}

void change_queue_mode(int qid, char *mode)
{
    struct msqid_ds myqueue_ds;
    /* Get current info */
    msgctl(qid, IPC_STAT, &myqueue_ds);
    /* Convert and load the mode */
    sscanf(mode, "%ho", &myqueue_ds.msg_perm.mode);
    /* Update the mode */
    msgctl(qid, IPC_SET, &myqueue_ds);
    return;
}

void usage(void)
{
    fprintf(stderr, "msgtool - A utility for tinkering with msg queues\n");
    fprintf(stderr, "\nUSAGE: msgtool (s)end \n");
    fprintf(stderr, " msgtool (r)ecv \n");
    fprintf(stderr, " msgtool (d)elele\n");
    fprintf(stderr, " msgtool (m)ode \n");
    exit(1);
}

```

下面我们来执行 `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 0x6d0105d1），里面有 20 个字节（Hello,my_msg_queue!）的 1 条消息。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。上述操作的输出如图 32 所示。

```
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./msgtool s 1 Hello,my_msg_queue!
Sending a message
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ipcs -q

----- 消息队列 -----
键          msqid      所有者  权限  已用字节数  消息
0x6d0105d1 0          ruichen 660    20          1

ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./msgtool r 1
Reading a message
Type: 1 Text: Hello,my_msg_queue!
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ipcs -q

----- 消息队列 -----
键          msqid      所有者  权限  已用字节数  消息
0x6d0105d1 0          ruichen 660     0          0
```

图 32. msgtool 的执行结果

2) 共享内存

System V IPC 的共享内存是由内核提供的一段内存，可以映射到多个进程的续存空间上，从而通过内存上的读写操作而完成进程间的数据共享。我们首先来看看如何创建共享内存的，示例代码如下所示，它创建了一个 4096 字节的共享内存区。shmget() 的第一个参数 IPC_PRIVATE (为 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)。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>
#include <stdio.h>
#define BUFSZ 4096
int main ( void )
{
    int shm_id;
    shm_id=shmget(IPC_PRIVATE, BUFSZ, 0666 ); //创建共享内存
    if (shm_id < 0 )
    {
        perror( "shmget fail!\n" );
        exit ( 1 );
    }
}
```

```
printf ( "Successfully created segment : %d \n", shm_id );

system( "ipcs -m"); //执行 ipcs -m 命令，显示系统的共享内存信息

return 0;

}
```

执行这段代码，结果如图 33 所示。

```
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./shmget-demo
Successfully created segment : 2523143
```

----- 共享内存段 -----						
键	shmid	拥有者	权限	字节	连接数	状态
0x00000000	557056	ruichen	600	524288	2	目标
0x00000000	294913	ruichen	600	33554432	2	目标
0x00000000	393218	ruichen	600	524288	2	目标
0x00000000	2162691	ruichen	600	524288	2	目标
0x00000000	524292	ruichen	600	524288	2	目标
0x00000000	1802245	ruichen	600	524288	2	目标
0x00000000	1900550	ruichen	600	16777216	2	目标
0x00000000	2523143	ruichen	666	4096	0	

图 33. shmget-demo 的输出

如图 33 所示，执行 shmget-demo 程序，输出结果表明新创建的共享内存的 ID 为 2523143， 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（因为此时连接数为 0）。

下面展示另一个进程通过影射该共享内存而使用它的过程，具体如下所示。

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

int main ( int argc, char *argv[] )
{
    int shm_id ;
    char * shm_buf;
    if ( argc != 2 )
    {
        printf ( "USAGE: atshm <identifier>" );
        exit(1);
    }

    shm_id = atoi(argv[1]);
```

```

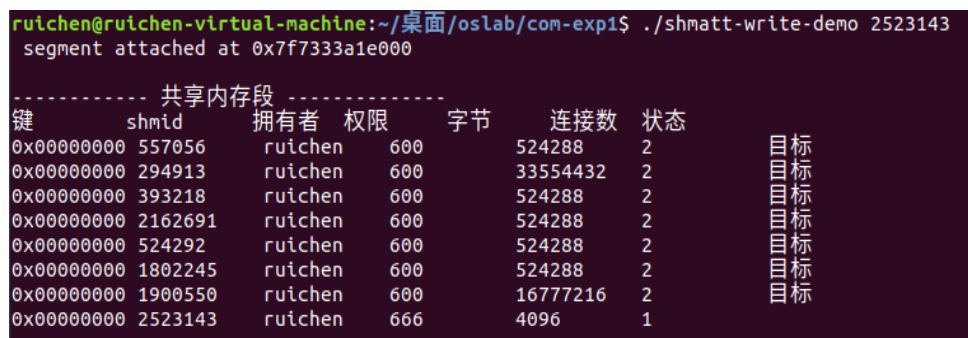
if ( (shm_buf = shmat( shm_id, 0, 0)) < (char *) 0 ) //映射共享内存到进程空间
{
    perror ( "shmat fail!\n" );
    exit (1);
}

printf ( " segment attached at %p\n", shm_buf );
system("ipcs -m"); //显示共享内存信息
strcpy(shm_buf,"Hello shared memory!\n");
getchar();
if ( (shmdt(shm_buf)) < 0 ) //解除共享内存的映射
{
    perror ( "shmdt");
    exit(1);
}

printf ( "segment detached \n" );
system ( "ipcs -m " );
getchar();
exit ( 0 );
}

```

我们运行 `./shmatt-write-demo 2523143`（命令行参数中指出共享内存的 ID 为 252314），其输出结果如图 34 所示。完成共享内存的映射后，`shmatt-write-demo` 往共享内存中写入一个字符串“Hello shared memory!”。`shmatt-write-demo` 还通过 `system()` 执行了“`ipcs -m`”，因此也输出了当前的共享内存信息，可以看到 ID 为 2523143 的共享内存已经有被映射了一次（连接数为 1）。



```

ruichen@ruichen-virtual-machine:~/桌面/oslab/con-exp1$ ./shmatt-write-demo 2523143
segment attached at 0x7f7333a1e000
----- 共享内存段 -----
键          shmid    拥有者  权限    字节    连接数  状态    目标
0x00000000  557056    ruichen  600     524288    2      目标
0x00000000  294913    ruichen  600     33554432  2      目标
0x00000000  393218    ruichen  600     524288    2      目标
0x00000000  2162691   ruichen  600     524288    2      目标
0x00000000  524292    ruichen  600     524288    2      目标
0x00000000  1802245   ruichen  600     524288    2      目标
0x00000000  1900550   ruichen  600     16777216  2      目标
0x00000000  2523143   ruichen  666     4096      1

```

图 34. shmatt-write-demo 的输出

图 10 第二行信息是该进程将共享内存映射到了进程空间 `0x7f7333a1e000` 位置的地方。为了观察该进程的进程空间，我们先用 `ps -a` 指令查看当前进程的进程号

pid=13697，然后开启另一个终端，用 `cat /proc/<PID>/maps` 可以查看进程的虚拟地址空间是如何使用的（7fc6d8335000-7fc6d8336000），如图 35 所示。

```
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ cat /proc/13697/maps
55dbc542b000-55dbc542c000 r-xp 00000000 08:01 132579 /home/ruichen/桌面/oslab/com-exp1/shmatt-write-demo
55dbc562b000-55dbc562c000 r--p 00000000 08:01 132579 /home/ruichen/桌面/oslab/com-exp1/shmatt-write-demo
55dbc562c000-55dbc562d000 rw-p 00001000 08:01 132579 /home/ruichen/桌面/oslab/com-exp1/shmatt-write-demo
55dbc6547000-55dbc6568000 rw-p 00000000 00:00 0 [heap]
7fc6d7d1e000-7fc6d7f05000 r-xp 00000000 08:01 394867 /lib/x86_64-linux-gnu/libc-2.27.so
7fc6d7f05000-7fc6d8105000 ---p 001e7000 08:01 394867 /lib/x86_64-linux-gnu/libc-2.27.so
7fc6d8105000-7fc6d8109000 r--p 001e7000 08:01 394867 /lib/x86_64-linux-gnu/libc-2.27.so
7fc6d8109000-7fc6d810b000 rw-p 001eb000 08:01 394867 /lib/x86_64-linux-gnu/libc-2.27.so
7fc6d810b000-7fc6d810f000 rw-p 00000000 00:00 0
7fc6d810f000-7fc6d8136000 r-xp 00000000 08:01 394863 /lib/x86_64-linux-gnu/ld-2.27.so
7fc6d8320000-7fc6d8322000 rw-p 00000000 00:00 0
7fc6d8335000-7fc6d8336000 rw-s 00000000 00:05 2523143 /SYSV00000000 (deleted)
7fc6d8336000-7fc6d8337000 r--p 00027000 08:01 394863 /lib/x86_64-linux-gnu/ld-2.27.so
7fc6d8337000-7fc6d8338000 rw-p 00028000 08:01 394863 /lib/x86_64-linux-gnu/ld-2.27.so
7fc6d8338000-7fc6d8339000 rw-p 00000000 00:00 0
7ffc6b2e3000-7ffc6b304000 rw-p 00000000 00:00 0 [stack]
7ffc6b382000-7ffc6b385000 r--p 00000000 00:00 0 [vvar]
7ffc6b385000-7ffc6b387000 r-xp 00000000 00:00 0 [vdso]
fffffffff6000000-fffffffff6010000 r-xp 00000000 00:00 0 [vsyscall]
```

图 35. shmatt-write-demo 映射共享内存时的进程布局

击键回车后 `shmatt-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（连接数为 0），如图 36 所示。此时如果检查进程布局，将发现 `7f89e55fc000-7f89e55fd000` 区间的虚存已经没有了，如图 37 所示。

```
segment detached
----- 共享内存段 -----
键          shmid      拥有者  权限    字节      连接数  状态
0x00000000  557056      ruichen  600      524288    2
0x00000000  294913      ruichen  600      33554432  2
0x00000000  393218      ruichen  600      524288    2
0x00000000  2162691     ruichen  600      524288    2
0x00000000  524292      ruichen  600      524288    2
0x00000000  1802245     ruichen  600      524288    2
0x00000000  1900550     ruichen  600      16777216  2
0x00000000  2523143     ruichen  666      4096      0
```

图 36. 解除共享内存的映射

```
7fc6d8105000-7fc6d8109000 r--p 001e7000 08:01 394867 /lib/x86_64-linux-gnu/libc-2.27.so
7fc6d8109000-7fc6d810b000 rw-p 001eb000 08:01 394867 /lib/x86_64-linux-gnu/libc-2.27.so
7fc6d810b000-7fc6d810f000 rw-p 00000000 00:00 0
7fc6d810f000-7fc6d8136000 r-xp 00000000 08:01 394863 /lib/x86_64-linux-gnu/ld-2.27.so
7fc6d8320000-7fc6d8322000 rw-p 00000000 00:00 0
7fc6d8336000-7fc6d8337000 r--p 00027000 08:01 394863 /lib/x86_64-linux-gnu/ld-2.27.so
7fc6d8337000-7fc6d8338000 rw-p 00028000 08:01 394863 /lib/x86_64-linux-gnu/ld-2.27.so
7fc6d8338000-7fc6d8339000 rw-p 00000000 00:00 0
```

图 37. 虚存被释放

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，代码如下。

```
#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdlib.h>
```



```

#include <stdio.h>
#include <string.h>
int main ( int argc, char *argv[] )
{
    int shm_id ;
    char * shm_buf;
    if ( argc != 2 )
    {
        printf ( "USAGE: atshm <identifier>" );
        exit (1 );
    }
    shm_id = atoi(argv[1]);
    if ( (shm_buf = shmat( shm_id, 0, 0)) < (char *) 0 )
    {
        perror ( "shmat fail!\n" );
        exit (1);
    }
    printf ( " segment attached at %p\n", shm_buf );
    system("ipcs -m");
    printf("The string in SHM is :%s\n",shm_buf); //将共享内存区的内容打印出来
    getchar();
    if ( (shmdt(shm_buf)) < 0 )
    {
        perror ( "shmdt");
        exit(1);
    }
    printf ( "segment detached \n" );
    system ( "ipcs -m " );
    getchar();
    exit (0);
}

```

运行代码，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，如图 38 所示。

```

ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./shmatt-read-demo 2523143
segment attached at 0x7f39a2bbc000

----- 共享内存段 -----
键          shmid    拥有者  权限    字节    连接数  状态
0x00000000  557056    ruichen  600     524288  2
0x00000000  294913    ruichen  600     33554432 2
0x00000000  393218    ruichen  600     524288  2
0x00000000  2162691   ruichen  600     524288  2
0x00000000  524292    ruichen  600     524288  2
0x00000000  1802245   ruichen  600     524288  2
0x00000000  1900550   ruichen  600     16777216 2
0x00000000  2523143   ruichen  666     4096    1
目标
目标
目标
目标
目标
目标
目标

The string in SHM is :Hello shared memory!

```

图 38. shmatt-read-demo 的输出

从上面实验看出共享内存是比较灵活的通信方式，不需要像管道那要用文件接口 read()、write()等函数，也不需要像消息队列那样用 msgsend()/msgrcv()等函数来操作，直接用内存指针方式就可以操作。虽然实验中没有验证其容量，但是共享内存的容量远比管道和消息队列大。

3) 信号量数组/信号量集

Linux 支持的 System V IPC 中的信号量实际上是信号量数组（信号量集），一次可以创建多个信号量。创建或者获得信号量集之后，可以对各个信号量进行 P/V 操作（或者称 up/down 操作），进程进行 P/V 操作时遵循信号的同步约束关系——由操作系统完成进程的阻塞或唤醒。下面我们来感受 Linux 编程中信号量集上的同步操作。

Linux 进程间同步内容部分的学习：

Linux 同时支持 System V IPC 中的信号量集和 POSIX 信号量。前者常用于进程间通信，而后者是常用于线程间同步、方便使用且仅含一个信号量。

POSIX 信号量分成有名信号量和无名信号量，前者和一个文件的路径名相关联，创建后不随进程结束而消失（可用于进程间通信），反之无名信号量则只在进程生命周期内存在且只能在该进程创建的线程间使用

POSIX 信号量

有名信号量：可以通过标识来访问，因此可以同时用于进程间同步和线程间同步。如下代码 (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。

然后用这里 gcc 编译要加-lthread 参数（参数-lpthread 用于指出链接时所用的线程库）完成编译，然后运行。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，如图 39 所示。

```

ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ gcc psem-named-open.c -o psem-named-open -lpthread
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./psem-named-open
please input a file name to act as the ID of the sem!
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./psem-named-open HelloWorld.c
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$

```

图 39. psem-named-open-demo 的输出

然后，我们来尝试执行 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），代码如下所示。它通过 `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。同样出于代码简洁的考虑，这里的代码也是没有检查 `sem_open()` 是否成功获得了信号量。因此，如果输入错误的标识字符串，则无法成功获得所指定的信号量，`sem_wait()` 引用无效的信号量而引发段错误。

```

#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
    sem_t *sem;
    int val;
    if(argc!=2)
    {
        printf("please input a file name!\n");
        exit(1);
    }
    sem=sem_open(argv[1],0); //获取信号量对象
    sem_wait(sem); //执行 P 操作（-1 操作）
    sem_getvalue(sem,&val); //获得当前信号量的值
    printf("pid %ld has semaphore,value=%d\n", (long) getpid(), val);
    return 0;
}

```

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（图 39 中输入的 `HelloWorld.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）。如果再运行一遍，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。这两次运行的情况如图 40 所示。

```

ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./psem-named-wait-demo HelloWorld.c
pid 14817 has semaphore,value=0
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./psem-named-wait-demo HelloWorld.c

```

图 40. psem-named-wait-demo 的运行输出

图 41 显示 psem-named-wait-demo 第二次运行后并没有返回到 shell 提示符，如果此时用另一个终端执行 ps 命令可以看到该进程处于 S 状态，如图 41 所示。

```

ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ps -aux | grep psem-named-wait-demo
ruichen  14843  0.0  0.0  6696  824 pts/0    S+   01:41   0:00  ./psem-named-wait-demo HelloWorld.c
ruichen  14850  8.0  0.0 16180 1120 pts/1    S+   01:46   0:00  grep --color=auto psem-named-wait-demo

```

图 41. ps 查看 psem-named-wait-demo 的运行状态

再接着来看看对该信号量进行 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。代码如下所示，这里也要注意代码并没有对 sem_open() 的返回值进行判定，因此输入错误的文件标识时隐含出现段错误的可能。

```

#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
    sem_t *sem;
    int val;
    if(argc!=2)
    {
        printf("please input a file name!\n");
        exit(1);
    }
    sem=sem_open(argv[1],0);
    sem_post(sem); //对信号量执行 P 操作（增 1）
    sem_getvalue(sem,&val);
    printf("value=%d\n", val);
    exit(0);
}

```

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 42 所示。

```
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ gedit psem-named-post-demo.c
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ gcc psem-named-post-demo.c -o psem-named-post-demo -lpthread
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./psem-named-post-demo HelloWorld.c
value=0
```

图 42(a). psem-named-post-demo 的运行输出

```
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./psem-named-wait-demo HelloWorld.c
pid 14817 has semaphore,value=0
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./psem-named-wait-demo HelloWorld.c
pid 14843 has semaphore,value=0
```

图 42 (b). psem-named-post-demo 唤醒阻塞的 psem-named-wait-demo 进程

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, 该信号量就可用)。。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, 其函数原型如图 43 所示。

```
#include<semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

图 43. 无名信号量的函数原型

第二个参数 pshared 表示该信号量是否由于进程间通步。当 pshared = 0, 那么表示该信号量只能用于进程内部的线程间的同步; 当 pshared != 0, 表示该信号量存放在共享内存区中(例如使用 shmget()), 使得引用它的进程能够访问该共享内存区进行进程同步。

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, 代码如下所示, 此时结果可能会出现错误。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)。

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <malloc.h>

#define thread_num 16
#define MB 1024 * 1024

int *array;

int length; //array length

int count;
```

```

int t; //number of thread

void *count3s_thread(void* id);

int main()
{
    int i;
    int tid[thread_num];
    pthread_t threads[thread_num];
    length = 64 * MB;
    array = malloc(length*4); //256MB
    for (i = 0; i < length; i++) //initial array
    {
        array[i] = i % 2 ? 4 : 3; //偶数 i 对应数值 3
    }
    for (t = 0; t < thread_num; t++) //循环创建 16 个线程
    {
        count = 0;
        tid[t]=t;
        int err = pthread_create(&(threads[t]), NULL, count3s_thread,&(tid[t]) );
        if (err)
        {
            printf("create thread error!\n");
            return 0;
        }
    }
    for (t = 1; t < thread_num; t++)
    {
        pthread_join(threads[t], NULL); //等待前面创建的计数线程结束
    }
    printf("Total count= %d \n",count);
    return 0;
}

void *count3s_thread(void* id) //计数线程执行的函数
{
    /*compute portion of the array that this thread should work on*/

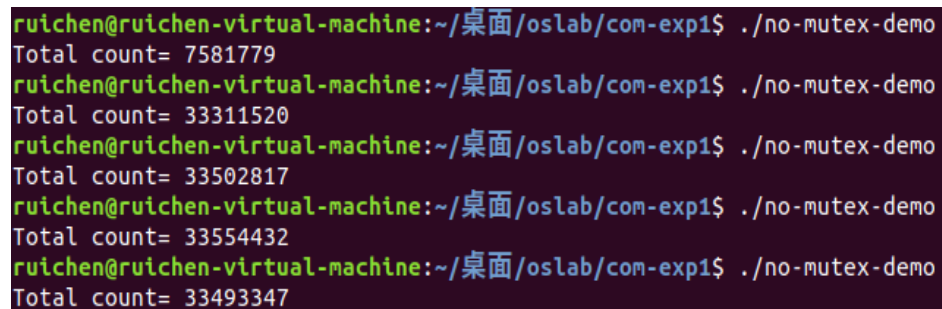
```

```

int length_per_thread = length / thread_num; //length of every thread
int start = *(int *)id * length_per_thread; //every thread start position
int i;
for (i = start; i < start + length_per_thread; i++)
{
    if (array[i] == 3)
    {
        count++; //计数，为加入互斥保护
    }
}
}

```

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），得到如图 44 所示的输出，每次运行结果并不唯一（共享变量未能得到互斥访问）。



```

ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./no-mutex-demo
Total count= 7581779
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./no-mutex-demo
Total count= 33311520
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./no-mutex-demo
Total count= 33502817
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./no-mutex-demo
Total count= 33554432
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./no-mutex-demo
Total count= 33493347

```

图 44. no-mutex-demo 的输出

如果对 count++这个临界区加以保护，就能避免出现这个错误。代码如下所示。

```

#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <malloc.h>

#define thread_num 16
#define MB 1024 * 1024

int *array;
int length; //array length
int count;
int t; //number of thread
void *count3s_thread(void* id);
pthread_mutex_t m; //增加一个互斥量
int main()

```

```

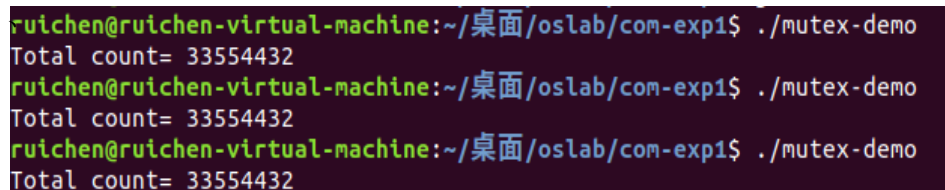
{
    pthread_mutex_init(&m,NULL); //初始化互斥量
    .....
}

void *count3s_thread(void* id)
{
    /*compute portion of the array that this thread should work on*/
    int length_per_thread = length / thread_num; //length of every thread
    int start = *(int *)id * length_per_thread; //every thread start position
    int i;
    for (i = start; i < start + length_per_thread; i++)
    {
        if (array[i] == 3)
        {
            pthread_mutex_lock(&m); //进入临界区
            count++;
            pthread_mutex_unlock(&m); //推出临界区
        }
    }
}

```

这里要注意红色标记位置类型应该这样声明，代表这是一个互斥量。

运行 `mutex_demo`，每次运行都获得相同的结果，如图 45 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。



```

ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./mutex-demo
Total count= 33554432
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./mutex-demo
Total count= 33554432
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp1$ ./mutex-demo
Total count= 33554432

```

图 45. mutex-demo 的输出

综上所述，这一部分讨论了 Linux 操作系统上的进程间的通信和同步手段，并进行了最基本的编程实践。在实际应用中，需要能有选择地使用这些通信和同步手段，并能将这些基本技能综合应用以解决多进程/多线程并发程序的具体问题。编写并发的服务器程序是进程间通信和同步的典型应用之一，必须综合多种通信和同步手段。同时要注意到，Linux 的进程间通信中实际上已经实现了必要的同步，例如在读空的管道或消息队列时进程会阻塞等等。

四、实验结论：

本次实验首先通过学习综合 1 预备-进程间通信与同步.pdf 的内容，学习了管道、System V IPC 作为进程间通信的方法、POSIX 信号量中有名信号量、无名信号量以及互斥量作为进程间同步的方法，再结合课上所学知识，实现了生产者和消费者之间的通信以及通过信号量实现互斥交替进行。

此外，本次实验实现了 shell 功能，包括接收内部命令、外部命令，可执行文件的执行以及错误提示。最后结合管道的知识实现了重定向功能。

五、实验体会：

本次实验内容比较多，需要自己去学习管道方面的知识，同时还要结合课上所学来真正实现生产者消费者的同步问题。此外，通过熟悉 Linux 内部命令、外部命令的具体操作来自己简单实现 shell，正常的命令比较容易，但是对于重定向的管道方面的问题相对复杂，需要仔细地思考，读取指令之后不要弄错指令之间的关系。

指导教师批阅意见：

成绩评定：

指导教师签字：

年 月 日

备注：