

实验三 Linux 进程管理

设计要求
实验步骤
结果演示
体会

实验三 Linux 进程管理

17042127

陶逸群

设计要求

- 实现一个模拟的 shell。
- 实现一个管道通信程序。
- 利用 Linux 消息队列通信机制实现两个线程之间的通信。
- 利用 Linux 的共享内存通信机制实现两个进程间的通信。
- 在模拟shell实验中，增加接收find和grep命令，并给结果显示。
- 在管道通信实验中，增加进程间有名管道通信。
- 在共享内存实验中，增加共享内存的双向通信。

实验步骤

1. 实现一个模拟的 shell

在第一个实验中，要实现一个模拟的 shell，共有 5 个命令，cmd1、cmd2、cmd3、find 以及 grep。

在具体实现中，前三个命令只打印命令的信息，后两个命令的功能和 Linux 系统中的命令一致。

我调用了 Linux 的 `execv` 系列系统原语实现模拟的 shell，每个命令单独写一个程序，然后在主函数中通过 `execv` 系列函数调用相应的命令程序并通过空格切分得到命令的参数，即可实现命令的相应功能。

与此同时，我还使用 `getpwuid` 等函数模拟了shell中对用户名主机名的输出。

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4  #include <sys/wait.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <ctype.h>
8  #include <pwd.h>
9  #define NUM 10
10 //获取登陆用户名
11 void GetLoginname(){
12     struct passwd* pass;
13     pass = getpwuid(getuid());
14     printf("[%s@ ",pass->pw_name);
15 }
16
17 //获取主机名
18 void GetHostname(){
19     char name[128];
20     gethostname(name,sizeof(name)-1);
21     printf("%s ",name);
22 }
23 void ChildProcess(int num,char * const myargv[],char ** environ){
```

```

24     pid_t pid=fork();
25     if(pid<0)
26         perror("fork error\n");
27     else if(pid==0){
28         switch(num){
29             case 1:{
30                 execve("cmd1",myargv,envron);
31                 break;
32             }
33             case 2:{
34                 execve("cmd2",myargv,envron);
35                 break;
36             }
37             case 3:{
38                 execve("cmd3",myargv,envron);
39                 break;
40             }
41             case 4:{
42                 execve("/bin/find",myargv,envron);
43                 break;
44             }
45             case 5:{
46                 execve("/bin/grep",myargv,envron);
47                 break;
48             }
49             default:{
50                 printf("process will never go here");
51                 break;
52             }
53         }
54         exit(1);
55     }else{
56         int status = 0;
57         pid_t ret = waitpid(pid,&status,0); //阻塞父进程
58         if(!(ret > 0 && WIFEXITED(status))){
59             perror("waitpid");
60         }
61     }
62 }
63 //获取当前文件路径
64 void GetDir(){
65     char pwd[128];
66     getcwd(pwd,sizeof(pwd)-1);
67     int len = strlen(pwd);
68     char *p = pwd+len-1;
69     while(*p != '/' && len--){
70         p--;
71     }
72     p++;
73     printf("%s] #",p);
74 }
75
76 int main(int argc,char * const argv[],char **envron)
77 {
78     while(1) {
79         GetLoginname();
80         GetHostname();
81         GetDir();
82         char cmd[1024];
83         char *myargv[NUM];
84         fflush(stdout); //清空输出缓冲区
85         int s = read(0,cmd,sizeof(cmd));
86         cmd[s-1] = '\0';
87         int i = 0;
88         myargv[0] = strtok(cmd, " ");

```

```

89     while(myargv[i] !=NULL){
90         i++;
91         myargv[i] = strtok(NULL, " ");
92     }
93     myargv[i] = NULL;
94     if (strcmp(myargv[0],"exit")==0) break;
95     else if (strcmp(myargv[0],"cmd1")==0){
96         ChildProcess(1,myargv,environ);
97     }
98     else if (strcmp(myargv[0],"cmd2")==0){
99         ChildProcess(2,myargv,environ);
100    }
101    else if (strcmp(myargv[0],"cmd3")==0){
102        ChildProcess(3,myargv,environ);
103    }else if (strcmp(myargv[0],"find")==0){
104        ChildProcess(4,myargv,environ);
105    }else if (strcmp(myargv[0],"grep")==0){
106        ChildProcess(5,myargv,environ);
107    }
108    else printf("Command not found\n");
109 }
110 return 0;
111 }
112

```

2. 实现管道通信

○ 无名管道通信

首先由父进程通过 `int pipe(int fd[2])` 函数创建一个无名管道，然后再 创建四个子进程：

`pid0`、`pid1`、`pid2`和`pid3`，`pid0`设置成非阻塞写，不断写入字符，测试无名管道大小，父进程通过 `wait(0)` 等待0号进程写入完毕后非阻塞读，清空管道，`pid1`、`pid2`、`pid3`这三个子进程利用管道与父 进程进行通信。父进程通过等待

`read_mutex1`、`read_mutex2`、`read_mutex3` 三个信号量，等待三个子进程全部发送完消息（子进程结束）后接受子进程发送的消息，`write_mutex` 信号量控制`pid1`、`pid2`、`pid3`三个进程间写互斥。

```

1  #include <errno.h>
2  #include <fcntl.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <sys/ipc.h>
7  #include <sys/sem.h>
8  #include <sys/types.h>
9  #include <sys/wait.h>
10 #include <unistd.h>
11 #include <semaphore.h>
12
13 #define STR_MAX_SIZE 1024
14
15
16 int main(int argc, char **argv) {
17     int pipefd[2], i = 0;
18     int pid0,pid1,pid2,pid3;
19     ssize_t n;
20     char buf[1];
21     char str[STR_MAX_SIZE];
22     int count = 0;
23
24     // 创建有名信号量，若不存在则创建，若存在则直接打开，默认值为0
25     sem_t *write_mutex;
26     sem_t *read_mutex1;
27     sem_t *read_mutex2;

```

```

28     sem_t *read_mutex3;
29     write_mutex = sem_open("pipe_w", O_CREAT | O_RDWR, 0666, 0);
30     read_mutex1 = sem_open("pipe_r_1", O_CREAT | O_RDWR, 0666, 0);
31     read_mutex2 = sem_open("pipe_r_2", O_CREAT | O_RDWR, 0666, 0);
32     read_mutex3 = sem_open("pipe_r_3", O_CREAT | O_RDWR, 0666, 0);
33     memset(buf, 0, 1);
34     memset(str, 0, STR_MAX_SIZE);
35
36     // 创建管道并检查操作是否成功
37     if(pipe(pipefd) < 0){
38         printf("创建无名管道失败");
39         exit(-1);
40     }
41
42
43     // 创建0号子进程并检查操作是否成功
44     // 利用非阻塞写测试管道大小
45     pid0 = fork();
46     if(pid0 < 0){
47         printf("第一个子进程创建失败");
48         exit(-1);
49     }
50
51     if (pid0 == 0) {
52         count = 0;
53         close(pipefd[0]);
54         // 管道默认是阻塞写, 通过`fcntl` 设置成非阻塞写, 在管道满无法继续写入时
        返回-EAGAIN, 作为循环终止条件
55         int flags = fcntl(pipefd[1], F_GETFL);
56         fcntl(pipefd[1], F_SETFL, flags | O_NONBLOCK);
57         n = 0;
58         // 写入管道
59         while (n!--1) {
60             n = write(pipefd[1], buf, 1);
61             count++;
62         }
63         printf("space = %dKB\n", (count * 1) / 1024);
64         exit(0);
65     }
66
67
68
69     // 创建第一个子进程并检查操作是否成功
70     pid1 = fork();
71     if(pid1 < 0){
72         printf("第一个子进程创建失败");
73         exit(-1);
74     }
75
76     if (pid1 == 0) {
77         sem_wait(write_mutex);
78         close(pipefd[0]);
79         n = write(pipefd[1], "这是一号进程\n", 20);
80         printf("进程一写入了 %ld字节\n", n);
81         sem_post(write_mutex);
82         sem_post(read_mutex1);
83         exit(0);
84     }
85
86     // 创建第二个子进程并检查操作是否成功
87     pid2 = fork();
88     if(pid2 < 0){
89         printf("第二个子进程创建失败");
90         exit(-1);
91     }

```

```

92
93     if (pid2 == 0) {
94         sem_wait(write_mutex);
95         close(pipefd[0]);
96         n = write(pipefd[1], "这是二号进程\n", 20);
97         printf("进程二写入了 %ld字节\n", n);
98         sem_post(write_mutex);
99         sem_post(read_mutex2);
100         exit(0);
101     }
102
103     // 创建第三个子进程并检查操作是否成功
104     pid3 = fork();
105     if(pid3 < 0){
106         printf("第三个子进程创建失败");
107         exit(-1);
108     }
109
110     if (pid3 == 0) {
111         sem_wait(write_mutex);
112         close(pipefd[0]);
113         n = write(pipefd[1], "这是三号进程\n", 20);
114         printf("进程三写入了 %ld字节\n", n);
115         sem_post(write_mutex);
116         sem_post(read_mutex3);
117         exit(0);
118     }
119     // 等待0号子进程运行完成，父进程继续运行
120     // 用非阻塞读，清空管道
121     wait(0);
122     close(pipefd[1]);
123     int flags = fcntl(pipefd[0], F_GETFL);
124     n = 0 ;
125     count = 0;
126     // 设置非阻塞性读，作为循环结束标志
127     fcntl(pipefd[0], F_SETFL, flags | O_NONBLOCK);
128     while (n!=-1) {
129         n = read(pipefd[0], buf, 1);
130         count++;
131     }
132     printf("空间大小为 %dKB \n", (count * 1) / 1024);
133     sem_post(write_mutex);
134
135     // 等待子进程一、二、三写入完毕
136     sem_wait(read_mutex1);
137     sem_wait(read_mutex2);
138     sem_wait(read_mutex3);
139     n = read(pipefd[0], str, STR_MAX_SIZE);
140     printf("读取了%ldB \n", n);
141     for (i = 0; i < n; i++) {
142         printf("%c", str[i]);
143     }
144
145     sem_close(write_mutex);
146     sem_close(read_mutex1);
147     sem_close(read_mutex2);
148     sem_close(read_mutex3);
149     sem_unlink("pipe_w");
150     sem_unlink("pipe_r_1");
151     sem_unlink("pipe_r_2");
152     sem_unlink("pipe_r_3");
153     return 0;
154 }
155

```

- 有名管道

首先由通过mkfifo()函数传入参数管道名以及权限创建管道，然后通过 open()函数传入参数管道名和读写方式打开管道，即可实现有名管道的读写操作。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5  #include <sys/stat.h>
6  #include <fcntl.h>
7  #include <errno.h>
8  #include <pthread.h>
9  void *func(void * fd)
10 {
11
12     int wri = write(*(int*)fd, "this is Thread_write", 30);
13     if(wri < 0)
14     {
15         printf("wirte fifo failed!\n");
16     }
17     close(*(int*)fd);
18 }
19 int main()
20 {
21     if(mkfifo("fifo", 0666) < 0 ){
22         if(errno != EEXIST){           //建立管道失败，并且最后管道不存在则退出
23             printf("create FIFO falied!\n");
24             return 0;
25         }
26     }
27     int fd = open("fifo", O_WRONLY);
28     if(fd < 0)
29     {
30         printf("open fifo failed!\n");
31         return 0;
32     }
33     pthread_t tid = 1;
34     pthread_create(&tid, NULL, func, &fd);
35     pthread_join(tid, NULL);
36
37     return 0;
38 }
39
```

```
1
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <stdlib.h>
6  #include <sys/stat.h>
7  #include <fcntl.h>
8  #include <errno.h>
9  #include <pthread.h>
10 void *func(void *fd)
11 {
12
13     char readbuf[1024];
14     read( *(int*)fd, readbuf, 30);
15     printf("this is Thread_read!\n");
16     printf("Receive message: %s\n", readbuf);
17     close(*(int*)fd);
18 }
19 int main()
```

```

20 {
21     int fd;
22     char buff[2048];
23     if(mkfifo("fifo", 0666) < 0 ){
24         if(errno != EEXIST){ //建立管道失败，并且最后管道不存在则退出
25             printf("create FIFO failed!\n");
26             return 0;
27         }
28     }
29     fd = open("fifo", O_RDONLY);
30     if(fd < 0)
31     {
32         printf("open FIFO failed!\n");
33     }
34     pthread_t tid = 0;
35     pthread_create(&tid, NULL, func, &fd);
36     pthread_join(tid, NULL);
37
38
39     return 0;
40 }
41

```

3. 利用 Linux 消息队列通信机制实现两个线程之间的通信

首先创建信号并对信号量初始化，在该实验中我使用了四个信号量，如下表所示：

| 信号量 | 作用 | 初始值 |
|-------------|---------------------------------|-----|
| sem_send | sender1和sender2发送消息互斥 | 1 |
| sem_receive | sender1, sender2和receiver接受消息同步 | 0 |
| sem_over1 | sender1结束 | 0 |
| sem_over2 | sender2结束 | 0 |

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <sys/stat.h>
6 #include <unistd.h>
7 #include <errno.h>
8 #include <sys/msg.h>
9 #include <pthread.h>
10 #include <semaphore.h>
11
12 #define snd_1 1
13 #define snd_2 2
14 #define rcv_1 3
15 #define rcv_2 4
16 #define MAX_SIZE 1024
17 #define QUEUE_ID 22222
18
19 typedef struct msg {
20     long message_type;
21     char message[MAX_SIZE];
22 }msg;
23
24 sem_t sem_send,sem_receive,sem_over1,sem_over2;

```

```

25
26 void *sender1() {
27     msg buf;
28     int mq;
29     mq = msgget((key_t) QUEUE_ID, IPC_CREAT|0666 );
30     while (1) {
31         sem_wait(&sem_send);
32         printf("sender:\n");
33         buf.message_type = snd_1;
34         fflush(stdout);
35         fgets(buf.message, BUFSIZ, stdin);
36         if(strcmp(buf.message,"exit\n")==0){
37             strcpy(buf.message,"end1\n");
38             msgsnd(mq, (void *) &buf, MAX_SIZE, 0);
39             sem_post(&sem_receive);
40             break;
41         }else{
42             msgsnd(mq, (void *) &buf, MAX_SIZE, 0);
43             sem_post(&sem_receive);
44         }
45     }
46     sem_wait(&sem_over1);
47     msgrcv(mq, (void *) &buf, MAX_SIZE, rcv_1, 0);
48     printf("%s", "over1\n");
49     printf("-----\n");
50     sem_post(&sem_send);
51     pthread_exit(NULL);
52 }
53
54
55 void *sender2() {
56     msg buf;
57     int mq;
58     mq = msgget((key_t) QUEUE_ID, IPC_CREAT|0666 );
59     while (1) {
60         sem_wait(&sem_send);
61         printf("sender2:\n");
62         buf.message_type = snd_2;
63         fflush(stdout);
64         fgets(buf.message, BUFSIZ, stdin);
65         if(strcmp(buf.message,"exit\n")==0){
66             strcpy(buf.message,"end2\n");
67             msgsnd(mq, (void *) &buf, MAX_SIZE, 0);
68             sem_post(&sem_receive);
69             break;
70         }else{
71             msgsnd(mq, (void *) &buf, MAX_SIZE, 0);
72             sem_post(&sem_receive);
73         }
74     }
75     sem_wait(&sem_over2);
76     msgrcv(mq, (void *) &buf, MAX_SIZE, rcv_2, 0);
77     printf("%s", "over2\n");
78     printf("-----\n");
79     sem_post(&sem_send);
80     pthread_exit(NULL);
81 }
82
83
84 void *receiver() {
85     msg buf, over1, over2;
86     struct msqid_ds t;
87     int stop = 2;
88     int mq;
89     over1.message_type = rcv_1;

```



```

90     strcpy(over1.message, "over1\n");
91     over2.message_type = rcv_2;
92     strcpy(over2.message, "over2\n");
93
94
95     mq = msgget((key_t) QUEUE_ID,  IPC_CREAT|0666 );
96
97     do {
98         sem_wait(&sem_receive);
99         msgrcv(mq, (void *) &buf, MAX_SIZE, 0, 0);
100        printf("Received%ld: %s", buf.message_type, buf.message);
101        printf("-----\n");
102
103        if ((strncmp(buf.message, "end1",
104strlen("end1"))==0)&&buf.message_type == snd_1) {
105            msgsnd(mq, (void *) &over1, MAX_SIZE, 0);
106            sem_post(&sem_over1);
107            stop--;
108        }else if((strncmp(buf.message, "end2",
109strlen("end2"))==0)&&buf.message_type == snd_2) {
110            msgsnd(mq, (void *) &over2, MAX_SIZE, 0);
111            sem_post(&sem_over2);
112            stop--;
113        }else{
114            sem_post(&sem_send);
115        }
116    } while (stop);
117    msgctl(mq, IPC_RMID, &t);
118    pthread_exit(NULL);
119 }
120
121 int main(int argc, char **argv) {
122     pthread_t t1, t2,t3;
123
124     sem_init(&sem_send, 0, 1);
125     sem_init(&sem_receive, 0, 0);
126     sem_init(&sem_over1, 0, 0);
127     sem_init(&sem_over2, 0, 0);
128
129     pthread_create(&t3, NULL, receiver, NULL);
130     pthread_create(&t1, NULL, sender1, NULL);
131     pthread_create(&t2, NULL, sender2, NULL);
132
133     pthread_join(t3, NULL);
134     pthread_join(t1, NULL);
135     pthread_join(t2, NULL);
136     return 0;
137 }

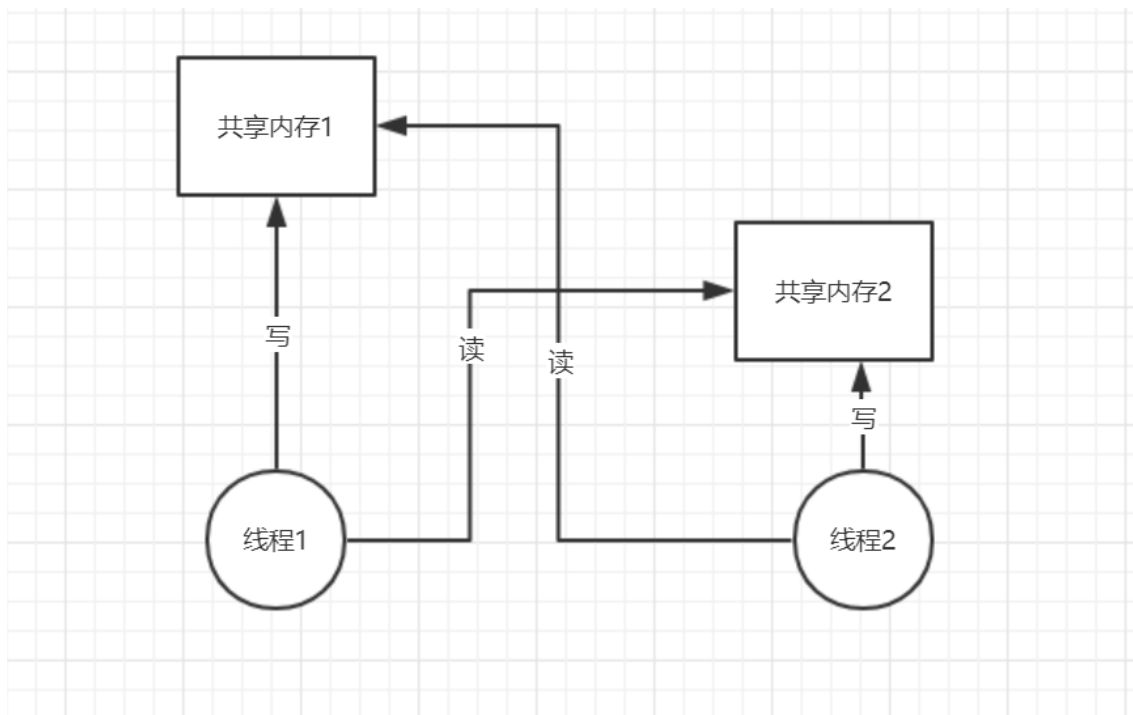
```

编写程序创建三个线程：sender1 线程、sender2 线程线程二和 receiver 线程。

receiver 线程先等待 `sem_receive` 信号量，然后读出队列中的消息。并释放 `sem_receive` 信号量。如果 sender 线程发送的消息为 exit，则 receiver 线程根据发送线程不同释放 `sem_over1` 或 `sem_over2` 信号量。sender 线程首先等待 `sem_send` 信号量，然后向消息队列写入消息，并释放 `sem_receive` 信号量。如果 sender 线程发送消息 exit，则退出循环，等待对应的 `sem_over1` 或者 `sem_over2` 信号量。

4. 利用 Linux 的共享内存通信机制实现两个进程间的双向通信

首先创建共享内存，创建信号并对信号量初始化，两个进程分别创建一块共享内存，他们间的关系如下图所示：



这样不需要任何信号量即可实现两个线程间的双向通信。

```
1  #include<stdio.h>
2  #include<sys/types.h>
3  #include<sys/ipc.h>
4  #include<sys/shm.h>
5
6  #define PATHNAME "."
7  #define PROJ_ID_R 0x6638
8  #define PROJ_ID_W 0x6639
9
10
11 int CreateShm(int size);
12 int DestroyShm(int shmid);
13 int GetShm(int size);
14
15 static int CommShm(int size,int flags,int app)
16 {
17     key_t key = ftok(PATHNAME,PROJ_ID_W);
18     if(app==0){
19         key = ftok(PATHNAME,PROJ_ID_R);
20     }
21     if(key < 0)
22     {
23         perror("ftok");
24         return -1;
25     }
26     int shmid = 0;
27     if((shmid = shmget(key,size,flags)) < 0)
28     {
29         perror("shmget");
30         return -2;
31     }
32     return shmid;
33 }
34 int DestroyShm(int shmid)
35 {
36     if(shmctl(shmid,IPC_RMID,NULL) < 0)
37     {
38         perror("shmctl");
39         return -1;
40     }
41 }
```

```

41     return 0;
42 }
43 int CreateShm(int size)
44 {
45     return CommShm(size,IPC_CREAT,1);
46 }
47 int GetShm(int size)
48 {
49     return CommShm(size,IPC_CREAT,0);
50 }
51
52 int main()
53 {
54     int shmid_r = GetShm(4096);
55     int shmid_w = CreateShm(4096);
56     char *addr_r = shmat(shmid_r,NULL,0);
57     char *addr_w = shmat(shmid_w,NULL,0);
58     int app;
59     while(1)
60     {
61         printf("1--read 2--write 3--exit");
62         scanf("%d",&app);
63         if(app == 3){
64             break;
65         }else if(app == 1){
66             printf("%s\n",addr_r);
67         }else{
68             printf("input:\n");
69             scanf("%s",addr_w);
70         }
71     }
72     shmdt(addr_r);
73     shmdt(addr_w);
74     DestroyShm(shmid_w);
75     return 0;
76 }
77

```

```

1  #include<stdio.h>
2  #include<sys/types.h>
3  #include<sys/ipc.h>
4  #include<sys/shm.h>
5
6  #define PATHNAME "."
7  #define PROJ_ID_R 0x6639
8  #define PROJ_ID_W 0x6638
9
10
11 int CreateShm(int size);
12 int DestroyShm(int shmid);
13 int GetShm(int size);
14
15 static int CommShm(int size,int flags,int app)
16 {
17     key_t key = ftok(PATHNAME,PROJ_ID_W);
18     if(app==0){
19         key = ftok(PATHNAME,PROJ_ID_R);
20     }
21     if(key < 0)
22     {
23         perror("ftok");
24         return -1;
25     }
26     int shmid = 0;

```

```

27     if((shmid = shmget(key,size,flags)) < 0)
28     {
29         perror("shmget");
30         return -2;
31     }
32     return shmid;
33 }
34 int DestroyShm(int shmid)
35 {
36     if(shmctl(shmid,IPC_RMID,NULL) < 0)
37     {
38         perror("shmctl");
39         return -1;
40     }
41     return 0;
42 }
43 int CreateShm(int size)
44 {
45     return CommShm(size,IPC_CREAT,1);
46 }
47 int GetShm(int size)
48 {
49     return CommShm(size,IPC_CREAT,0);
50 }
51
52 int main()
53 {
54     int shmid_r = GetShm(4096);
55     int shmid_w = CreateShm(4096);
56     char *addr_r = shmat(shmid_r,NULL,0);
57     char *addr_w = shmat(shmid_w,NULL,0);
58     int app;
59     while(1)
60     {
61         printf("1--read 2--write 3--exit");
62         scanf("%d",&app);
63         if(app == 3){
64             break;
65         }else if(app == 1){
66             printf("%s\n",addr_r);
67         }else{
68             printf("input:\n");
69             scanf("%s",addr_w);
70         }
71     }
72     shmdt(addr_r);
73     shmdt(addr_w);
74     DestroyShm(shmid_w);
75     return 0;
76 }
77

```

结果演示

- 模拟 shell

```
应用程序 位置 终端
root@localhost~/.exercise3

文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[root@localhost exercise3] # ./shell
[root@localhost exercise3] # cmd1
cmd1
[root@localhost exercise3] # cmd2
cmd2
[root@localhost exercise3] # find . -print
.
./_shell.c.swp
./_shell.c
./cmd1
./cmd1.c
./cmd2
./cmd2.c
./cmd3
./cmd3.c
./test.c
./test
./shelldtest
./pipe.c
./fifo_read.c
./fifo_read
./fifo_write.c
./fifo_write
./fifo
./queue.c
./queue
./shm1.c
./shm2.c
./shm2
./shm3
./shell
./pipe
[root@localhost exercise3] # grep int shell.c
printf("%s\n", pass->pw_name);
printf("%s", name);
void ChildProcess(int num, char * const myargv[], char ** environ){
    printf("process will never go here");
    int status = 0;
    int len = strlen(pwd);
    printf("%s\n", p);
    int main(int argc, char * const argv[], char **environ)
    int s = read(0, cmd, sizeof(cmd));
    int i = 0;
    else printf("Command not found\n");
[root@localhost exercise3] # exit
[root@localhost exercise3] #
```

- 管道通信实验

- 无名管道

```
[ root@localhost exercise3] # ./pipe
space = 64KB
空间大小为 64KB
进程一写入了 20字节
进程二写入了 20字节
进程三写入了 20字节
读取了60B
这是一号进程
这是二号进程
这是三号进程
[ root@localhost exercise3] #
```

- 有名管道

```
root@localhost:~/exercise3
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[root@localhost exercise3] # ./fifo_write
[root@localhost exercise3] #
```

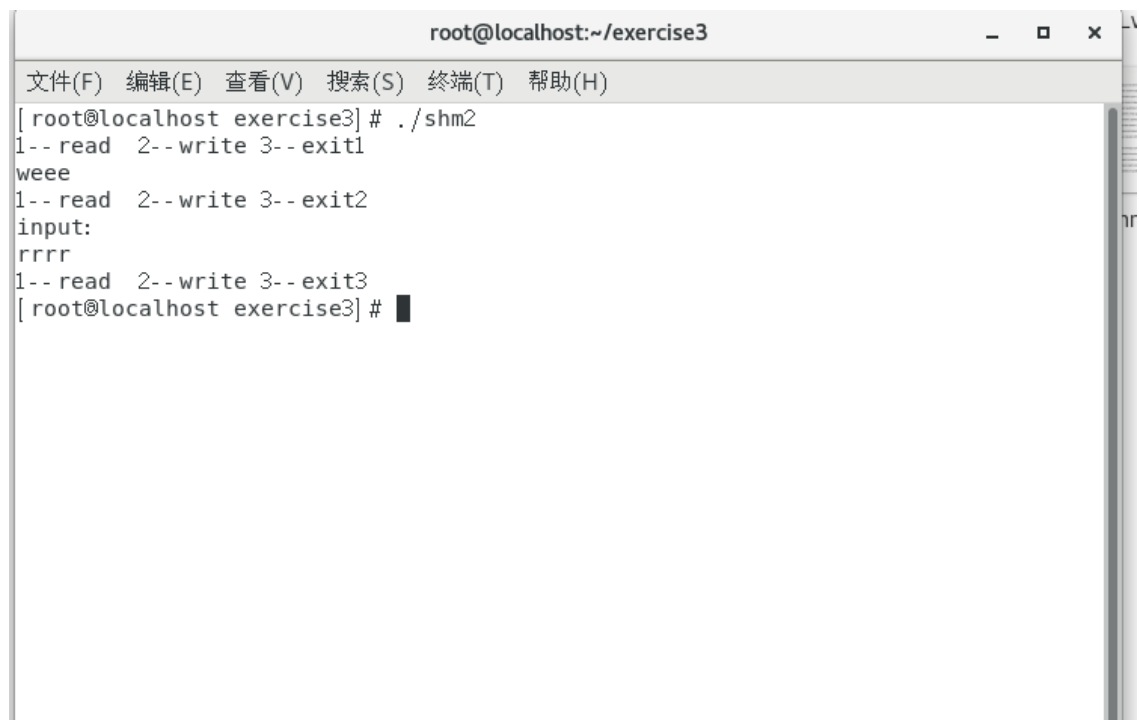
```
root@localhost:~/exercise3
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[root@localhost exercise3] # ./pipe
space = 64KB
空间大小为 64KB
进程一写入了 20字节
进程二写入了 20字节
进程三写入了 20字节
读取了60B
这是一号进程
这是二号进程
这是三号进程
[root@localhost exercise3] # ./fifo_read
this is Thread_read!
Receive message: this is Thread_write
[root@localhost exercise3] #
```

- 消息队列

```
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[ root@localhost exercise3] # ./queue
sender:
123
Received1: 123
-----
sender2:
1234
Received2: 1234
-----
sender:
12345
Received1: 12345
-----
sender2:
exit
Received2: end2
-----
over2
-----
sender:
exit
Received1: end1
-----
over1
-----
[ root@localhost exercise3] #
```

- 共享内存双向通信

```
root@localhost:~/exercise3
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[ root@localhost exercise3] # ./shm1
1-- read  2-- write 3-- exit2
input:
weee
1-- read  2-- write 3-- exit1
rrrr
1-- read  2-- write 3-- exit3
[ root@localhost exercise3] #
```

A terminal window titled 'root@localhost:~/exercise3' with standard window controls. The menu bar includes '文件(F)', '编辑(E)', '查看(V)', '搜索(S)', '终端(T)', and '帮助(H)'. The terminal content shows a shell prompt '[root@localhost exercise3] # ./shm2' followed by the execution of a program that uses shared memory. The program prints '1-- read 2-- write 3-- exit1', 'weee', '1-- read 2-- write 3-- exit2', 'input:', 'rrrr', and '1-- read 2-- write 3-- exit3' before returning to the shell prompt '[root@localhost exercise3] #'.

```
root@localhost:~/exercise3
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[root@localhost exercise3] # ./shm2
1-- read 2-- write 3-- exit1
weee
1-- read 2-- write 3-- exit2
input:
rrrr
1-- read 2-- write 3-- exit3
[root@localhost exercise3] #
```

体会

通过本次实验，我对 Linux 的通信机制有了深入的了解，通过实验我分别通过有名管道、无名管道、消息队列、共享内存，四种方式实现了进程或线程间的通信，并通过对比了解了不同机制之间的区别。同时我还对 Linux 信号量实现 进程互斥有了更加深入的体会。