# 操作系统课程设计报告

**实验三：Linux 进程管理**

**专业： 理工类实验班**

**班级： 19185312**

**姓名： 郑凯心**

**学号： 19063140**

**指导老师： 任彧**

**上机时间：周五下午6、7、8**

**杭州电子科技大学 卓越学院**

**2021.10**

## 一、实验目的

（1）通过对 linux 进程控制的相关系统调用的编程应用，进一步加深对进程概念的理解，明确进程和程序的联系和区别，理解进程并发执行的具体含义。

（2）通过 Linux 管道通信机制、消息队列通信机制、共享内存通信机制的应用，加深对不同类型的进程通信方式的理解。

（3）通过对 linux 的 Posix 信号量及 IPC 信号量的应用，加深对信号量同步机制的理解。

（4）请根据自身情况，进一步阅读分析相关系统调用的内核源码实现。

### 二、实验内容

（1） 实现一个模拟的 shell：

编写三个不同的程序 cmd1.c，cmd2.c，cmd3.c，每个程序的功能自定，分别编译成可执行文件 cmd1，cmd2，cmd3。然后再编写一个程序，模拟 shell 程序的功能：能根据用户输入的字符串（表示相应的命令名），为相应的命令创建子进程并让它去执行相应的程序，而父进程则等待子进程结束，然后再等待接收下一条命令。如果接收到的命令为 exit，则父进程结束，退出模拟 shell；如果接收到的命令是无效命令，则显示“Command not found”，继续等待输入下一条命令。

（2） 实现一个管道通信程序：

由父进程创建一个管道，然后再创建 3 个子进程，并由这三个子进程利用管道与父进程之间进行通信：子进程发送信息，父进程等三个子进程全部发完消息后再接收信息。通信的具体内容可根据自己的需要随意设计，要求能试验阻塞型读写过程中的各种情况，测试管道的默认大小，并且要求利用 Posix 信号量机制实现进程间对管道的互斥访问。运行程序，观察各种情况下，进程实际读写的字节数以及进程阻塞唤醒的情况。

（3）利用 linux 的消息队列通信机制实现两个线程间的通信：

编写程序创建三个线程：sender1 线程、sender2 线程和 receive 线程，三个线程的功能描述如下：

①sender1 线程：运行函数 sender1()，它创建一个消息队列，然后等待用户通过终端输入一串字符，并将这串字符通过消息队列发送给 receiver 线程；可循环发送多个消息，直到用户输入“exit”为止，表示它不再发送消息，最后向 receiver 线程发送消息“end1”，并且等待 receiver 的应答，等到应答消息后，将接收到的应答信息显示在终端屏幕上，结束线程的运行。

②sender2 线程：运行函数 sender2()，共享 sender1 创建的消息队列，等待用户通过终端输入一串字符，并将这串字符通过消息队列发送给 receiver 线程；可循环发送多个消息，直到用户输入“exit”为止，表示它不再发送消息，最后向 receiver 线程发送消息“end2”，并且等待 receiver 的应答，等到应答消息后，将接收到的应答信息显示在终端屏幕上，结束线程的运行。

③Receiver 线程：运行函数 receive()，它通过消息队列接收来自 sender1 和 sender2 两个线程的消息，将消息显示在终端屏幕上，当收到内容为“end1”的消息时，就向 sender1发送一个应答消息“over1”； 当收到内容为“end2”的消息时，就向 sender2 发送一个应答消息“over2”；消息接收完成后删除消息队列，结束线程的运行。选择合适的信号量机制实现三个线程之间的同步与互斥。

（4）利用 linux 的共享内存通信机制实现两个进程间的通信：

编写程序 sender，它创建一个共享内存，然后等待用户通过终端输入一串字符，并将这串字符通过共享内存发送给 receiver；最后，它等待 receiver 的应答，收到应答消息后，将接收到的应答信息显示在终端屏幕上，删除共享内存，结束程序的运行。编写 receiver 程序，它通过共享内存接收来自 sender 的消息，将消息显示在终端屏幕上，然后再通过该共享内存向 sender 发送一个应答消息“over”，结束程序的运行。选择合适的信号量机制实现两个进程对共享内存的互斥及同步使用。

### 三、实验方法

**信号量的使用**

POSIX无名信号量

1. sem\_t sem; //创建信号量
2. **int** sem\_init(sem\_t \*sem, **int** pshared, unsigned **int** val); //pshared为0则线程间共享，pshared为1则父子进程共享
3. **int** sem\_wait(sem\_t \*sem); //阻塞，信号量减一
4. **int** sem\_trywait(sem\_t \*sem); //非阻塞，信号量减一
5. **int** sem\_post(sem\_t \*sem); //信号量加一
6. **int** sem\_destroy(sem\_t \*sem); //删除信号量

POSIX有名信号量

1. sem\_t \*sem\_open(**const** **char** \*name, **int** oflag, mode\_t mode, **int** val); //创建/打开信号量
2. //name：信号量的名字
3. //oflag：使用O\_CREAT（如果信号量已经存在，则O\_CREAT什么都不做函数也不出错）。如果使用O\_CREAT|O\_EXCL（表示如果信号量已存在，则创建失败，sem\_open函数出错返回）
4. //mode：表示谁可以访问信号量
5. //value：指定信号量的初始值。取值范围是0~SEM\_VALUE\_MAX
6. **int** sem\_wait(sem\_t \*sem); //阻塞，信号量减一
7. **int** sem\_trywait(sem\_t \*sem); //非阻塞，信号量减一
8. **int** sem\_post(sem\_t \*sem); //信号量加一
9. **int** sem\_close(sem\_t \*sem); //关闭信号量
10. **int** sem\_unlink(**const** **char** \*name); //删除信号量

（1） 实现一个模拟的 shell：

循环，先输出“shell:”标识符号，然后读入指令command，整行读入

1. scanf("%[^\n]%\*c", command);
2. sscanf(command, "%s", token);

提取出指令中的第一个字符串，即为命令token，拿这个字符串和cmd1,cmd2,cmd3,find,grep去匹配，如果为前三者，创建子进程去调用对应程序，如果为后两者，系统调用。

（2） 实现一个管道通信程序：

**无名管道**：需要指定两个文件描述符，通过pipe()函数创建一个管道文件使其一端读文件一端写。对管道文件的读写和普通文件一样，直接用read()和write()函数。读写之前要调用close()关闭不需要的写读描述符。

1. **int** fd[2];
2. pipe(fd);

**有名管道**： 有名管道使用的是fifo()函数，一个进程写数据，一个进程读数据，两个进程不能结束。对管道文件的读写和普通文件一样，直接用open()、write()和read()函数。

1. **int** mkfifo(**const** **char** \* pathname, mode\_t mode); //pathname为管道文件所在地址，mode为管道文件权限和open()函数一致。

先创建管道，然后创建三个子进程，父进程等待三个子进程运行结束接受数据，子进程则读入数据放进管道。有名管道和无名管道除了管道部分实现不同，其余部分基本一致。

对于三子进程，是互斥关系，同时只能有一个运行，故设置一个信号量 Mutex,在进行放入数据的过程中要sem\_wait(Mutex)，执行完成后sem\_post(Mutex),另外对于每个子进程设计一个信号量receive[i]，因为父进程要等三个子进程都结束了，即三个receive信号量都为1时才能执行。

编程时发现三个子进程的功能几乎一致，如果将相同的代码复制三遍修改变量名则显得非常不明智不优雅，于是打算采用for循环来实现这一过程。信号量等采用数组形式，用下标实现一一对应。

信号量定义及初始化如下

1. sem\_t \*Mutex;
2. sem\_t \*receive[N];
3. **int** i;
4. sem\_unlink("Mutex");
5. **for** (i = 0; i < N; ++i) {
6. sprintf(buf, "receive%d", i);
7. sem\_unlink(buf);
8. }
9. Mutex = sem\_open("Mutex",O\_CREAT,0666,1);
10. **for** (i = 0; i < N; ++i) {
11. sprintf(buf, "receive%d", i);
12. receive[i] = sem\_open(buf, O\_CREAT,0666,0);
13. }

创建子进程的代码如下所示

1. **for** (i = 0; i < N; ++i) {
2. pid[i] = fork();
3. **if** (pid[i] > 0) **continue**;
4. **else** **break**;
5. }

发送过程代码如下

1. **for** (i = 0; i < N; ++i) **if** (pid[i] == 0) {
2. sem\_wait(Mutex);
3. printf("pid:%d 进程%d放入数据：",getpid(), i+1);
4. scanf("%[^\n]%\*c",buf);
5. 将buf放入管道
6. sleep(0.1);
7. sem\_post(Mutex);
8. sem\_post(receive[i]);
9. **break**;
10. }

父进程代码如下

1. **for** (i = 0; i < N; ++i) sem\_wait(receive[i]);
2. sem\_wait(Mutex);
3. 从管道中读取数据放入buf
4. printf("pid:%d 父进程接收数据:%s\n",getpid(),buf);
5. sleep(0.1);
6. sem\_post(Mutex);

测试管道的默认大小，创建一个子进程用for循环不断往管道写入数据，当管道满后会阻塞，循环变量i就会保持不变，若一段时间后i都不变则可以确定管道已经满了。

1. **int** i = 0;
3. **void** \*kwrite() {
4. **int** pipefds[2];
5. pipe(pipefds);
6. **for** (i = 0; ; ++i) write(pipefds[1], "1", 1);
7. close(pipefds[0]);
8. close(pipefds[1]);
9. }
11. **int** main() {
12. pthread\_t **thread**;
13. **int** lasti = 0, t = 0;
14. pthread\_create(&**thread**,NULL,kwrite,&i);
15. **while**(1) {
16. sleep(0.01);
17. **if**(i != lasti) {
18. lasti = i;
19. t = 0;
20. } **else** **if**(++t == 100) {
21. printf("管道大小为: %dB\n", i \* 1);
22. **break**;
23. }
24. }
25. **return** 0;
26. }

（3）利用 linux 的消息队列通信机制实现两个线程间的通信：

设置信号量sem\_send来控制发送过程互斥，设置信号量sem\_receive保证接受过程互斥，信号量sem\_over来通知发送进程的结束。

启动两个sender0，sender1即发送端，一个receiver接受端，一开始sem\_send=1，sem\_receive=0，同一时刻仅有一个发送端可以发送数据，发送数据后sem\_send=0，sem\_receive=1，接受端接受数据，然后重新恢复sem\_send=1，sem\_receive=0。

另一种情况时发送进程结束了，此时接受端还要向发送端发送一条信息，此时如果简单的继续使用sem\_send，sem\_receive信号量则可能造成另一个发送端开始发送，就会混乱，所以此时使用另外的信号量sem\_ove来控制接收端向发送端发送的结束消息，此时就由结束的那个发送端恢复sem\_send=1。

另外接受端要等两个发送端都结束了才可以结束，因此需要一个变量来记录结束的数量。

另外发现两个发送端代码几乎一致，如果复制两遍会显得非常不优雅，于是通过小小的手段来区分两个端，即启动程序时往main函数传入参数来区别不同端口的id

发送端主要代码

1. **while**(1) {
2. printf("发送：");
3. scanf("%[^\n]%\*c",str);
4. sem\_wait(sem\_send);
5. **if**(strcmp(str, "exit") == 0 || strcmp(str, "quit") == 0) {
6. sprintf(s\_msg.mtext, "end%d", id);
7. msgsnd(msgid, &s\_msg, **sizeof**(**struct** my\_msgbuf), 0);
8. sem\_post(sem\_receive);
9. **break**;
10. }
11. strcpy(s\_msg.mtext, str);
12. msgsnd(msgid, &s\_msg, **sizeof**(**struct** my\_msgbuf), 0);
13. sem\_post(sem\_receive);
14. }
15. sem\_wait(sem\_over[id]);
16. msgrcv(msgid, &r\_msg, **sizeof**(**struct** my\_msgbuf), 0, 0);
17. printf("收到线程%d的消息: %s\n", r\_msg.sendid, r\_msg.mtext);
18. sem\_post(sem\_send);

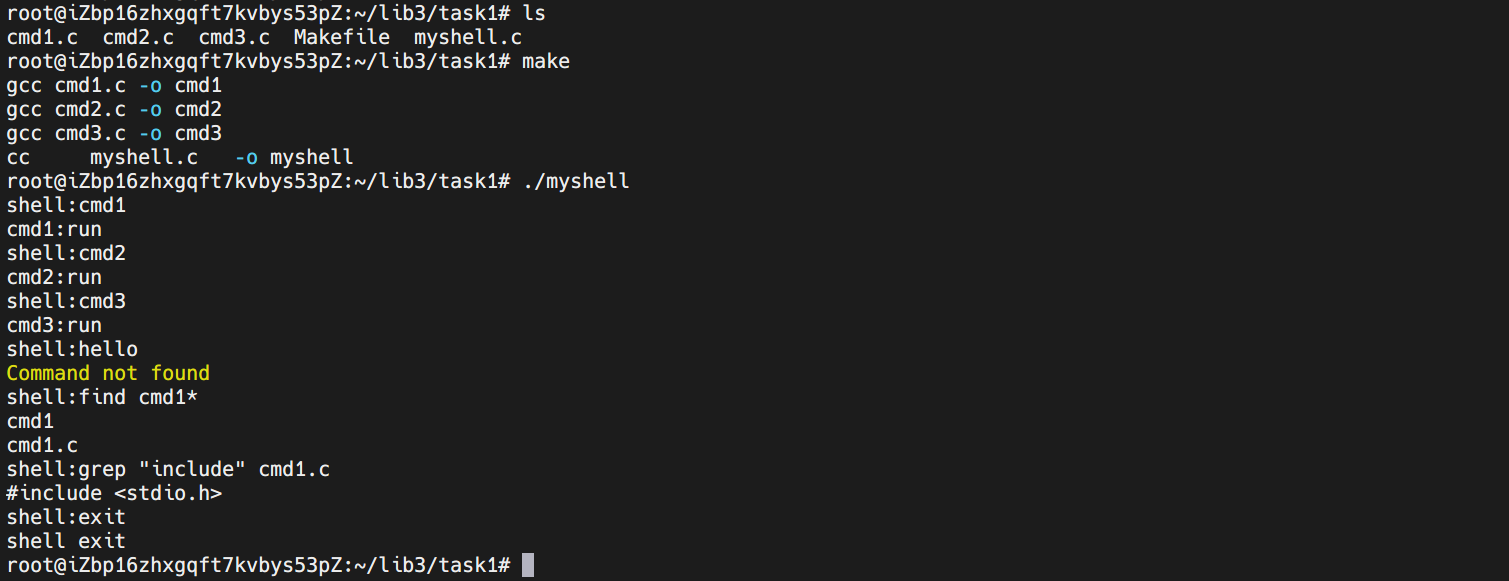
接收端主要代码

1. **while** (1) {
2. sem\_wait(sem\_receive);
3. msgrcv(msgid, &r\_msg, **sizeof**(**struct** my\_msgbuf), 0, 0);
4. printf("收到线程%u的消息: %s\n", r\_msg.sendid, r\_msg.mtext);
6. **if** (r\_msg.mtext[0] == 'e' && r\_msg.mtext[1] == 'n' && r\_msg.mtext[2] == 'd') {
7. printf("发送给线程%d：over%d\n", r\_msg.sendid, r\_msg.sendid);
8. sprintf(s\_msg.mtext, "over%d", r\_msg.sendid);
9. msgsnd(msgid, &s\_msg, **sizeof**(**struct** my\_msgbuf), 0);
10. sem\_post(sem\_over[r\_msg.sendid]);
11. ++over\_cnt;
12. } **else** {
13. sem\_post(sem\_send);
14. }
15. **if** (over\_cnt == N) **break**;
16. }

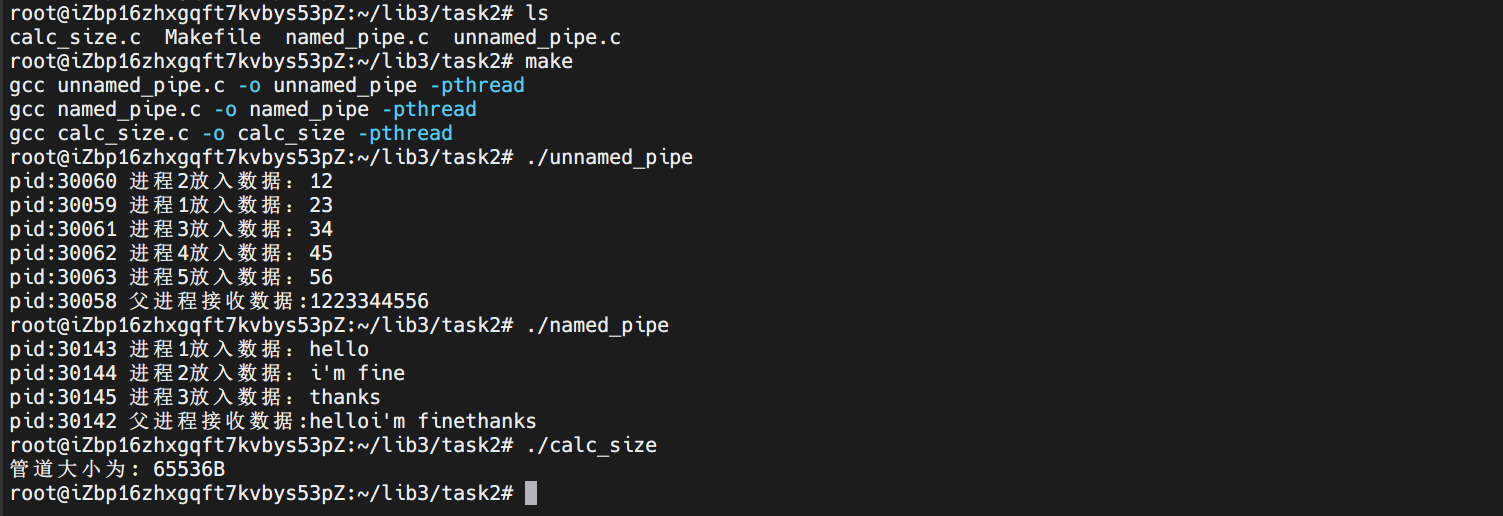
（4）利用 linux 的共享内存通信机制实现两个进程间的通信：

### 四、实验过程和结果

（1） 实现一个模拟的 shell：



（2） 实现一个管道通信程序：



（3）利用 linux 的消息队列通信机制实现两个线程间的通信：

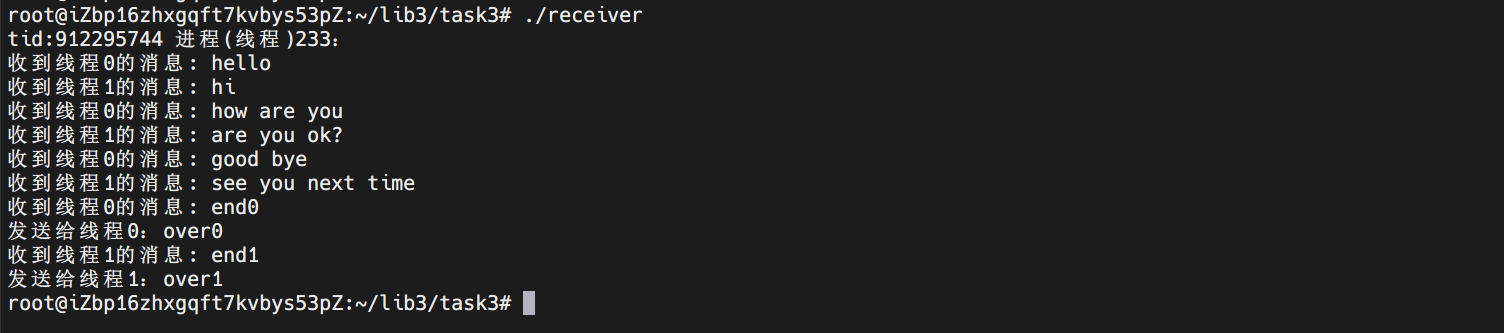
Sender 0



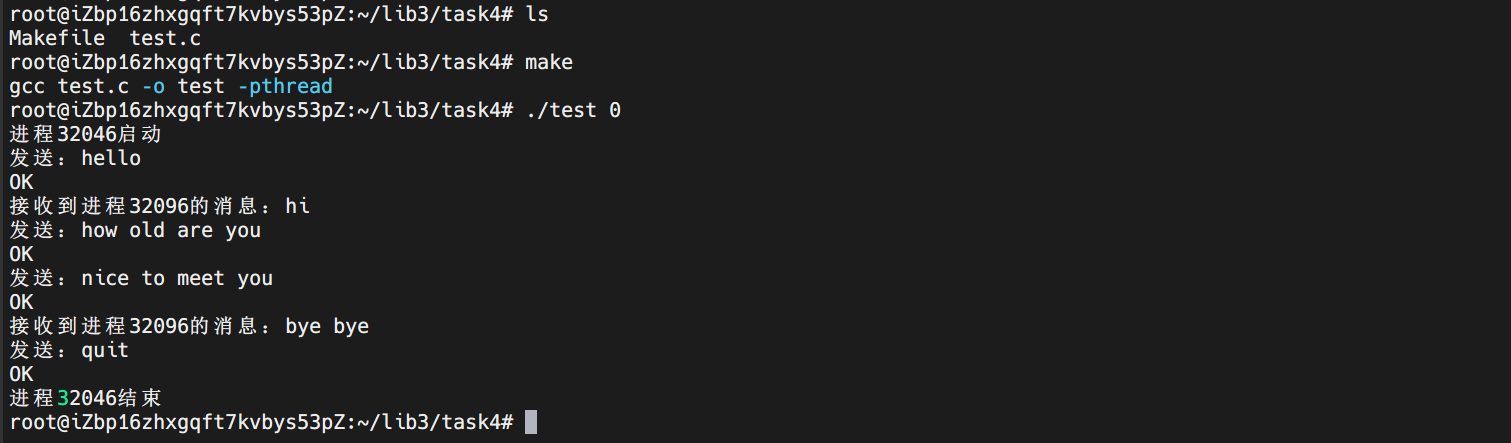
Sender 1



Receiver



（4）利用 linux 的共享内存通信机制实现两个进程间的通信：





### 五、实验体会

遇到的问题和分析：实验系统调用无法被成功调用，原因是系统调用表分配了两个一样的系统调用号，后面的系统调用函数覆盖了前面的系统调用函数，修改后成功实现系统调用。

实验总结：通过本次实验，我学会了如何编译内核，dmesg命令的用法和如何添加与调用系统调用函数。此外，我也对nice值和prio值有了深入的理解，对uname命令、hostname命令等的实现原理有一定了解，为更深入学习linux系统打下了基础。

### 六、参考文献

云服务器centOS编译升级Linux内核-Linux（1）, https://blog.csdn.net/babybabyup/article/details/79720082 2018,04,02

添加一个系统调用，实现对指定进程的nice值的修改或读取功能，并返回系统最新的nice值即优先级prio。-Linux(2), https://blog.csdn.net/babybabyup/article/details/79839734 2018,04,08