## 实验三报告：Linux进程管理

### 一、实验目的

* 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。
* 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。
* 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。
* 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

* **实现一个模拟的shell**
* 编写三个不同的程序cmd1.c、cmd2.c及cmd3.c，每个程序的功能自定，分别编译成可执行文件cmd1、cmd2及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（表示响应的命令名），为响应的命令创建子进程并让它去执行响应的程序，而父进程则等到子进程结束，然后再等待接受下一条命令。如果接受到的命令为exit，则父进程结束，退出模拟shell；如果接受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。
* **实现一个管道通信机制**
* 由父进程创建一个管道，然后再创建三个子进程，并由这三个子进程利用管道与父进程之间进行通信：子进程发送消息，父进程等三个子进程全部发完消息后再接受信息。通信的具体内容可根据自己的需要随意设计，要求能实验阻塞型读写工程中的各种情况，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的情况。
* **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，是三个线程的功能描述如下：
* （1）sender1线程：运行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。
* （2）sender2 线程：运行函数sender2(), 共享sender1创建的消息队列，等待用户通过线端输入一串字符，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。
* （3） Receiver 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- 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。
* **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串字符，并将这串字符通过共享内存发送给reierer最后，它等待rceiver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。

### 设计方案与实现

#### 模拟shell

* cmd1, cmd2, cmd3: 都是简单输出，使用fork函数创建子进程，复制当前进程的状态，以执行cmd的三个子程序。
* fork函数会将父进程的相关数据结构继承到子进程中，这样就使子进程中的文件描述符表中的fd[0]和fd[1]指向父进程所指向的管道文件,就能实现两个进程之间的通信
* 父进程使用wait函数等待子进程结束
* fpid=fork();  
  if(fpid<0){  
   printf("创建进程失败");  
   return -1;  
  }  
  //子进程  
  else if(fpid==0){  
   ...  
  }  
  //父进程  
  else{  
   wait(&status);  
   printf("愉快的对话结束了！（父进程）\n");  
  }
* 主程序用了枚举与字符串数组完成指令识别功能

#### 管道通信

* 创建管道文件描述符 ，子进程写前关闭读端（close(fd[0]），父进程读前关闭写端close(fd[1])，读写函数分别为read, write
* if(pipe(fd)!=0){  
   printf("创建管道失败\n");  
   exit(1);  
   }  
    
  //写入管道  
  write(fd[1], msg1, MSG1\_SIZE);  
    
  //读管道  
  read(fd[0], result, MSG1\_SIZE);
* 用fork函数创建子进程
* 通过fcntl获取当前描述符fd的文件状态标记，然后将之与非阻塞标志O\_NONBLOCK进行或操作再进行设置；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
* fcntl(fd[1], F\_SETFL, flags | O\_NONBLOCK);
* 利用信号量机制完成父进程等待子进程写完再读的顺序：父进程必须接收到子进程结束之后返回的 0，才能继续运行，否则阻塞。读取子进程一写入的数据，否则子进程二、三无法继续写入. 读空管道后结束循环，释放信号量，子进程二、三继续运行
* sem\_t \*write\_mutex;  
  sem\_t \*read\_mutex1;  
  sem\_t \*read\_mutex2;  
  write\_mutex = sem\_open("write", O\_CREAT | O\_RDWR, 0666, 0);  
  read\_mutex1 = sem\_open("read1", O\_CREAT | O\_RDWR, 0666, 0);  
  read\_mutex2 = sem\_open("read2", O\_CREAT | O\_RDWR, 0666, 0);

#### 消息队列

* 消息队列结构体msg\_s: long mtype;char mtext[MAX\_MSG\_LEN];
* 信号量设计
  + full: 消息队列的已有消息长度，当大于0的时候receiver可以成功申请到资源，小于0被阻塞。初始值0。
* empty：消息队列剩余的长度，当大于0的时候sender可以成功申请到资源，小于0被阻塞。初始值就是想定义的消息队列长度。
  + mutex: 消息队列控制权，初始值1。
* resp1, resp2: 回复信号量，因为这个实验的sender最后也需要收到receiver的回复，也需要申请full，所以用这两个信号量来阻塞sender的收(receiver一运行就申请了reply,等到over发出后再释放)，初始值1。
  + 具体控制：sender在发送前先申请mutex, empty，发完后释放mutex, full，resp在接收前申请full, 收完后释放empty

### 实验总结

* 一开始写了三个if else流程块，发现会先执行第一个之后，直接执行了第三个的else

需要等待子进程写完才能读

pid1 = waitpid(pid1, NULL, WUNTRACED);  
pid2 = waitpid(pid2, NULL, WUNTRACED);  
pid3 = waitpid(pid3, NULL, WUNTRACED);

* Segmentation fault (core dumped)
* 字符串输入写错了，char \*input,没有非配内存大小，改成char input[1024]
* 一开始实验消息队列的时候给的value值为0，导致后来再重新运行的时候一直都是阻塞状态
* 今天写了个消息队列的小测试程序结果send端程序总是出现：msgsnd: Invalid argument，搞了半个小时也没搞明白，后来查资料发现我将（st\_msg\_buf.msg\_type = 0; //设置消息类型）设置为0了，原来0表示是任意类型的消息，只有recv端才可以设置为0表示：可以接受任意类型消息。
* Segmentation fault (core dumped) 设置最大值太大了