操作系统实验2

1.进程的软中断通信

1.1实验目的

编程实现进程的创建和软中断通信，通过观察、分析实验现象，深入理解进程及进程在调度执行和内存空间等方面的特点，掌握在**POSIX**规范中系统调用的功能和使用。

1.2 实验内容

编制实现软中断通信的程序。

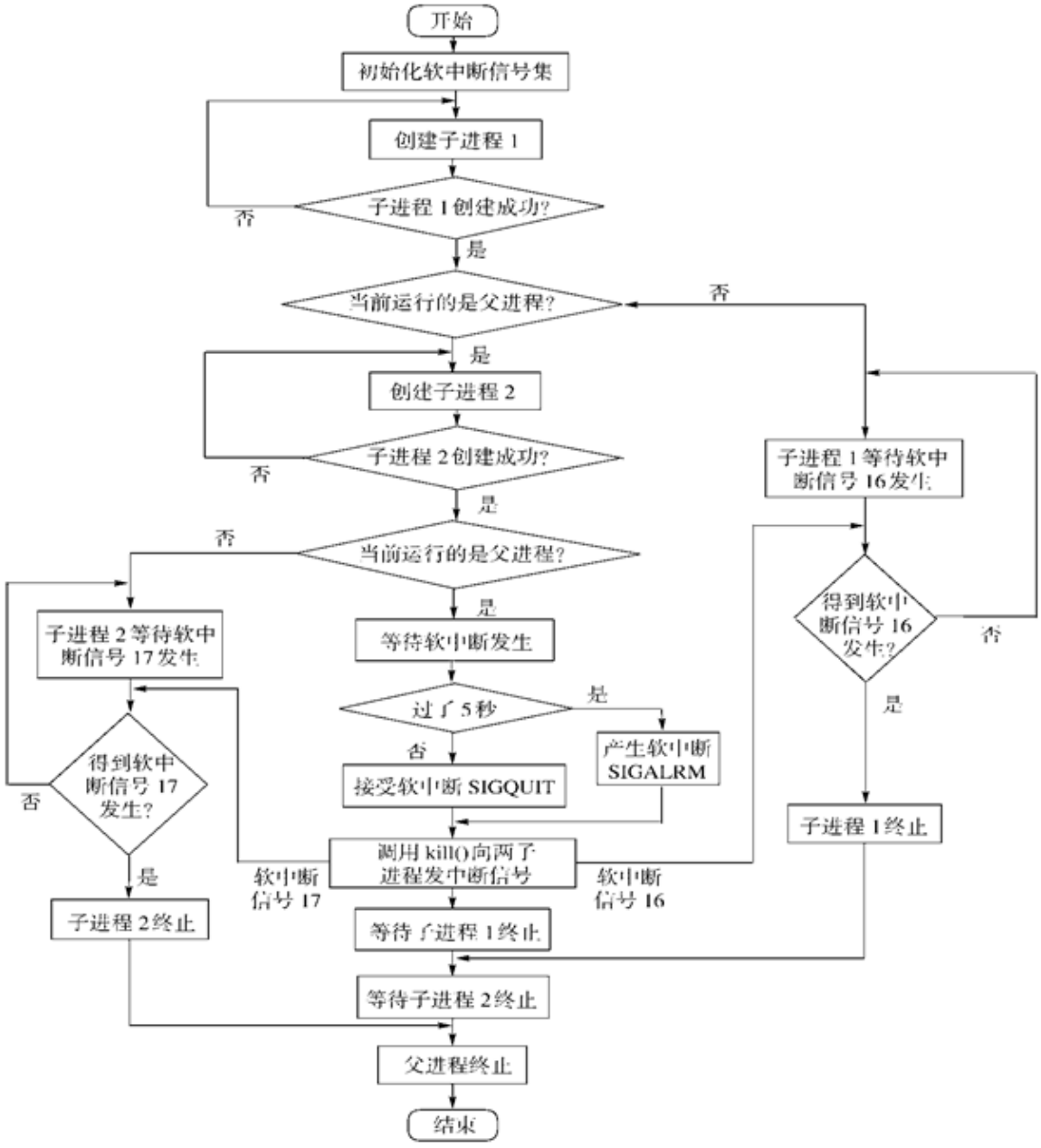
使用系统调用fork()创建两个子进程，
用系统调用signal()让父进程捕捉键盘上发出的中断信号（即按delete键），当父进程接收到这两个软中断的某一个后，父进程用系统调用kill()向两个子进程分别发出整数值为16和17软中断信号。
子进程获得对应软中断信号，然后分别输出下列信息后终止：
Child process 1 is killed by parent !!
Child process 2 is killed by parent !!
父进程调用wait()函数等待两个子进程终止后，输入以下信息，结束进程执行：
Parent process is killed!!

多运行几次编写的程序，简略分析出现不同结果的原因。

值	名字	说明
01	SIGHUP	挂起
02	SIGINT	中断，当用户从键盘键入“del”键时（ctrl+C）
03	SIGQUIT	退出，当用户从键盘键入“quit”键时（ctrl+\）
04	SIGILL	非法指令
05	SIGTRAP	断点或跟踪指令
06	SIGIOT	IOT指令
07	SIGEMT	EMT指令
08	SIGFPE	浮点运算溢出
09	SIGKILL	要求终止进程
10	SIGBUS	总线错误
11	SIGSEGV	段违例，即进程试图去访问其地址空间以外的地址
12	SIGSYS	系统调用错
13	SIGPIPE	向无读者的管道中写数据

值	名字	说明
14	SIGALARM	闹钟
15	SIGTERM	软件终止
16	SIGUSR1	用户自定义信号
17	SIGUSR2	用户自定义信号
18	SIGCLD	子进程死
19	SIGPWR	电源故障

程序流程图：



1.3实验过程记录

1.代码补充完整，运行程序：观察Delete/quit键前后，会出现什么结果？分析原因。

```
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$ gcc -o softirq softirq.c
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$ ./softirq

 Child process 1 is killed by parent !!

 Child process 2 is killed by parent !!

 Parent process is killed !!
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$
```

delete键后：

```
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$ ./softirq

 Child process 1 is killed by parent !!

 Child process 2 is killed by parent !!
^[[3~

 Parent process is killed !!
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$
```

程序仿佛没有受到影响，继续运行至正常结束。

quit后：

```
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$ ./softirq

 Child process 1 is killed by parent !!

 Child process 2 is killed by parent !!
^\
  3 stop test

 Parent process is killed !!
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$
```

程序立刻终止运行，因为程序中添加了signal变量3（sigquit），键入的sigquit能够被捕获，从而调用新定义的stop函数直接终止程序的运行。

2.如果程序运行，界面上显示“Child process 1 is killed by parent !! Child process 2 is killed by parent !!”，五秒之后显示“Parent process is killed !!”，怎样修改程序使得只有接收到相应的中断信号后再发生跳转，执行输出？

```
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$ gcc -o softirq softirq.c
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$ ./softirq

 Child process 1 is killed by parent !!

 Child process 2 is killed by parent !!

 Parent process is killed !!
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$
```

答：说明在子进程1和2中程序仅仅设置了软中断的相应方法，并没有使用软中断，而是直接继续运行下去了。如果想要实际测试新设置的软中断，可以让子程序1和2不断运行下去，如使用“ while(wait_flag); ”将其阻塞。在主进程发送软中断给子进程1和2后，在软中断设计的执行函数中将wait_flag设置为0，使得子进程停止被阻塞，继续运行。

```
28          } else {
29              wait_flag = 1;
30
31              signal(17,stop);        // 等待进程2被杀死的中断号17
32
33              while(wait_flag);       // 阻塞在这里，阻止程序直接继续运行，直到主进程发送软中断后再继续运行
34
35              printf("\n Child process 2 is killed by parent !!\n");
36              exit(0);
37          }
38      } else {
39          wait_flag = 1;
40          signal(16, stop);           // 等待进程1被杀死的中断号16
41
42          while(wait_flag);           // 阻塞在这里，阻止程序直接继续运行，直到主进程发送软中断后再继续运行
43
44          printf("\n Child process 1 is killed by parent !!\n");
45          exit(0);
46      }
47  }
48  void stop(int signum) {
49      wait_flag = 0;
50      printf("\n % d stop test \n", signum);
51  }
52
```

```
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$ gcc -o softirq softirq.c
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$ ./softirq


  16 stop test
  17 stop test

 Child process 1 is killed by parent !!

 Child process 2 is killed by parent !!

 Parent process is killed !!
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$ ./softirq
^\


  3 stop test
  3 stop test
  3 stop test


 Child process 1 is killed by parent !!
 Child process 2 is killed by parent !!


  17 stop test
  16 stop test

 Parent process is killed !!
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$
```

3.将本实验中通信产生的中断通过14 号信号值进行闹钟中断，将signal(3,stop)当中数字信号变为2，体会不同中断的执行样式，从而对软中断机制有一个更好的理解：

将signal(3,stop)当中数字信号变为2，通信产生的中断使用14号SIGALRM：

```c
signal(2, stop);        // 或者 signal(14,stop);
while ((pid1 = fork()) == -1)
    ;                   // 若创建子进程1不成功,则空循环
if (pid1 > 0) {         // 子进程创建成功,pid1为进程号
    while ((pid2 = fork()) == -1)
        ; // 创建子进程2
    if (pid2 > 0) {
        wait_flag = 1;
        sleep(5);               // 父进程等待5秒
        kill(pid1, 14);         // 杀死进程1发中断号16
        kill(pid2, 14);         // 杀死进程2
        wait(0);                // 等待第1个子进程1结束的信号
        wait(0);                // 等待第2个子进程2结束的信号
        printf("\n Parent process is killed !!\n");
        exit(0);                // 父进程结束
    } else {
        wait_flag = 1;

        signal(14,stop);        // 等待进程2被杀死的中断号17
```

```
xjtuhpc@ubuntu:~/Code/OS_exp/source_2$ ./softirq
^C
  2 stop test

  14 stop test

  2 stop test

 Child process 1 is killed by parent !!

  14 stop test

  2 stop test

 Child process 2 is killed by parent !!

 Parent process is killed !!
xjtuhpc@ubuntu:~/Code/OS_exp/source_2$
```

看到按下ctrl+C键，程序没有使用默认的终止程序的处理方式，而是转入我们自定义的处理方式了。

回答下列问题，写入实验报告。

1. 你最初认为运行结果会怎么样?

   以为若不按键，五秒后子程序被中断，退出；若按键，子程序直接被中断，退出

2. 实际的结果什么样？有什么特点？在接收不同中断前后有什么差别？请将5秒内中断和5秒后中断的运行结果截图，试对产生该现象的原因进行分析。

   实际中，按下没有在程序中做处理的按键，接收中断后并没有特殊的（和默认不同的）反应；按下添加了signal变量的按键，发出程序中处理的中断，能按照程序中的设计正常结束程序，

（此时仅处理SIGQUIT）

delete键后：

```
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$ ./softirq

 Child process 1 is killed by parent !!

 Child process 2 is killed by parent !!
^[[3~

 Parent process is killed !!
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$
```

程序仿佛没有受到影响，继续运行至正常结束。

quit后：

```
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$ ./softirq

 Child process 1 is killed by parent !!

 Child process 2 is killed by parent !!
^\
  3 stop test

 Parent process is killed !!
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$
```

程序立刻终止运行，因为程序中添加了signal变量3（sigquit），键入的sigquit能够被捕获，从而调用新定义的stop函数直接终止程序的运行。

3. 针对实验过程2，怎样修改的程序？修改前后程序的运行结果是什么？请截图说明。

在子进程1和2中程序仅仅设置了软中断的相应方法，并没有使用软中断，而是直接继续运行下去了。如果想要实际测试新设置的软中断，可以让子程序1和2不断运行下去，如使用“while(wait_flag); ”将其阻塞。在主进程发送软中断给子进程1和2后，在软中断设计的执行函数中将wait_flag设置为0，使得子进程停止被阻塞，继续运行。

修改程序：

```
28          } else {
29              wait_flag = 1;
30
31              signal(17,stop);        // 等待进程2被杀死的中断号17
32
33              while(wait_flag);       // 阻塞在这里，阻止程序直接继续运行，直到主进程发送软中断后再继续运行
34
35              printf("\n Child process 2 is killed by parent !!\n");
36              exit(0);
37          }
38      } else {
39          wait_flag = 1;
40          signal(16, stop);           // 等待进程1被杀死的中断号16
41
42          while(wait_flag);           // 阻塞在这里，阻止程序直接继续运行，直到主进程发送软中断后再继续运行
43
44          printf("\n Child process 1 is killed by parent !!\n");
45          exit(0);
46      }
47  }
48  void stop(int signum) {
49      wait_flag = 0;
50      printf("\n % d stop test \n", signum);
51  }
52
```

修改前：

```
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$ gcc -o softirq softirq.c
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$ ./softirq

 Child process 1 is killed by parent !!

 Child process 2 is killed by parent !!

 Parent process is killed !!
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$
```

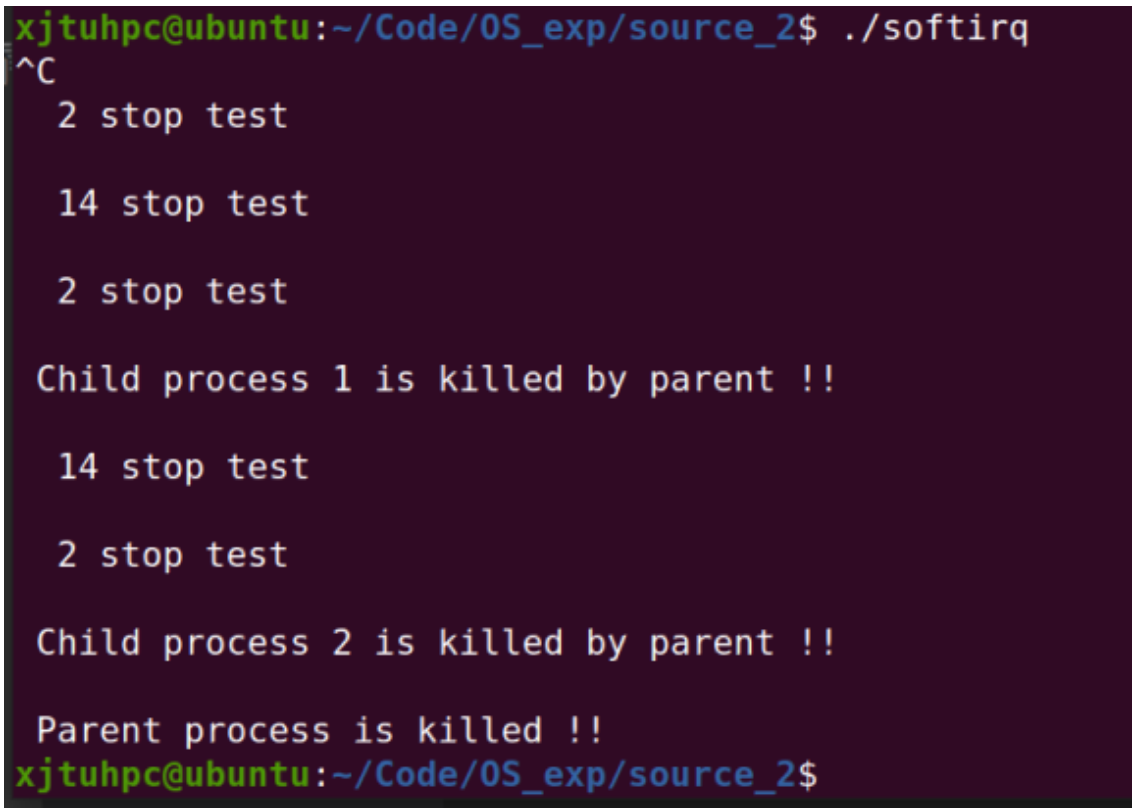
修改后：

```
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$ gcc -o softirq softirq.c
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$ ./softirq


  16 stop test
  17 stop test

 Child process 1 is killed by parent !!

 Child process 2 is killed by parent !!

 Parent process is killed !!
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$ ./softirq
^\


  3 stop test
  3 stop test
  3 stop test


 Child process 1 is killed by parent !!
 Child process 2 is killed by parent !!


  17 stop test
  16 stop test

 Parent process is killed !!
xjtuhpc@ubuntu:~/Code/use/exp2/my_OS_exp_02$
```

4. 针对实验过程3，程序运行的结果是什么样子？时钟中断有什么不同？

   将signal(3,stop)当中数字信号变为2，通信产生的中断使用14号SIGALRM：

```c
signal(2, stop);        // 或者 signal(14,stop);
while ((pid1 = fork()) == -1)
    ;                   // 若创建子进程1不成功,则空循环
if (pid1 > 0) {         // 子进程创建成功,pid1为进程号
    while ((pid2 = fork()) == -1)
        ; // 创建子进程2
    if (pid2 > 0) {
        wait_flag = 1;
        sleep(5);               // 父进程等待5秒
        kill(pid1, 14);         // 杀死进程1发中断号16
        kill(pid2, 14);         // 杀死进程2
        wait(0);                // 等待第1个子进程1结束的信号
        wait(0);                // 等待第2个子进程2结束的信号
        printf("\n Parent process is killed !!\n");
        exit(0);                // 父进程结束
    } else {
        wait_flag = 1;

        signal(14,stop);        // 等待进程2被杀死的中断号17
```



看到按下ctrl+C键，程序没有使用默认的终止程序的处理方式，而是转入我们自定义的处理方式了。时钟中断的使用并没有明显的不同。

5. kill 命令在程序中使用了几次？每次的作用是什么？执行后的现象是什么？

   kill命令共执行了两次,分别是向子进程1发送16号软中断和向子进程2发送17号软中断。执行的结果是触发两个子进程执行设计好的对应的软中断的执行函数。

6. 使用kill 命令可以在进程的外部杀死进程。进程怎样能主动退出？这两种退出方式哪种更好一些？

   进程可以调用exit() 来主动退出。但是exit返回的应当是 SIGCHLD信号，如果父进程没有处理这个信号，也没有wait子进程，子进程虽然终止，但是还会在内核进程表中占有表项，变成僵尸进程；kill命令带上不同的信号执行的方式不同。有的是直接杀死进程，不回收资源；有的是杀死进程并回收资源。最好是在外部使用适当的kill，使得进程能够返回应有的数据并释放资源。

实验中的问题：

1. 对signal的理解和使用不到位，老师的讲解和ppt讲到signal就听不懂

   解决：在网上阅读了大量的讲解文章，突然有一刻就开窍了，突然就明白了signal函数原来就是软中断，或者说中断的模拟。后面再回看ppt就如鱼得水了

2. 不知道什么是quit

   解决：查阅资料得知QUIT字符（SIGQUIT 信号）通常指的是是Ctrl-\，这些应当都属于终端I/O特殊输入字符。

   https://www.cnblogs.com/nufangrensheng/p/3576423.html

3. 虚拟机按键ctrl+C失效

   解决：可能跟虚拟机的设置有关，比如VMware有一个默认设置会将 Ctrl+Shift+C/V 和 Ctrl+C/V 交换按键映射。找到对应选项进行设置即可。

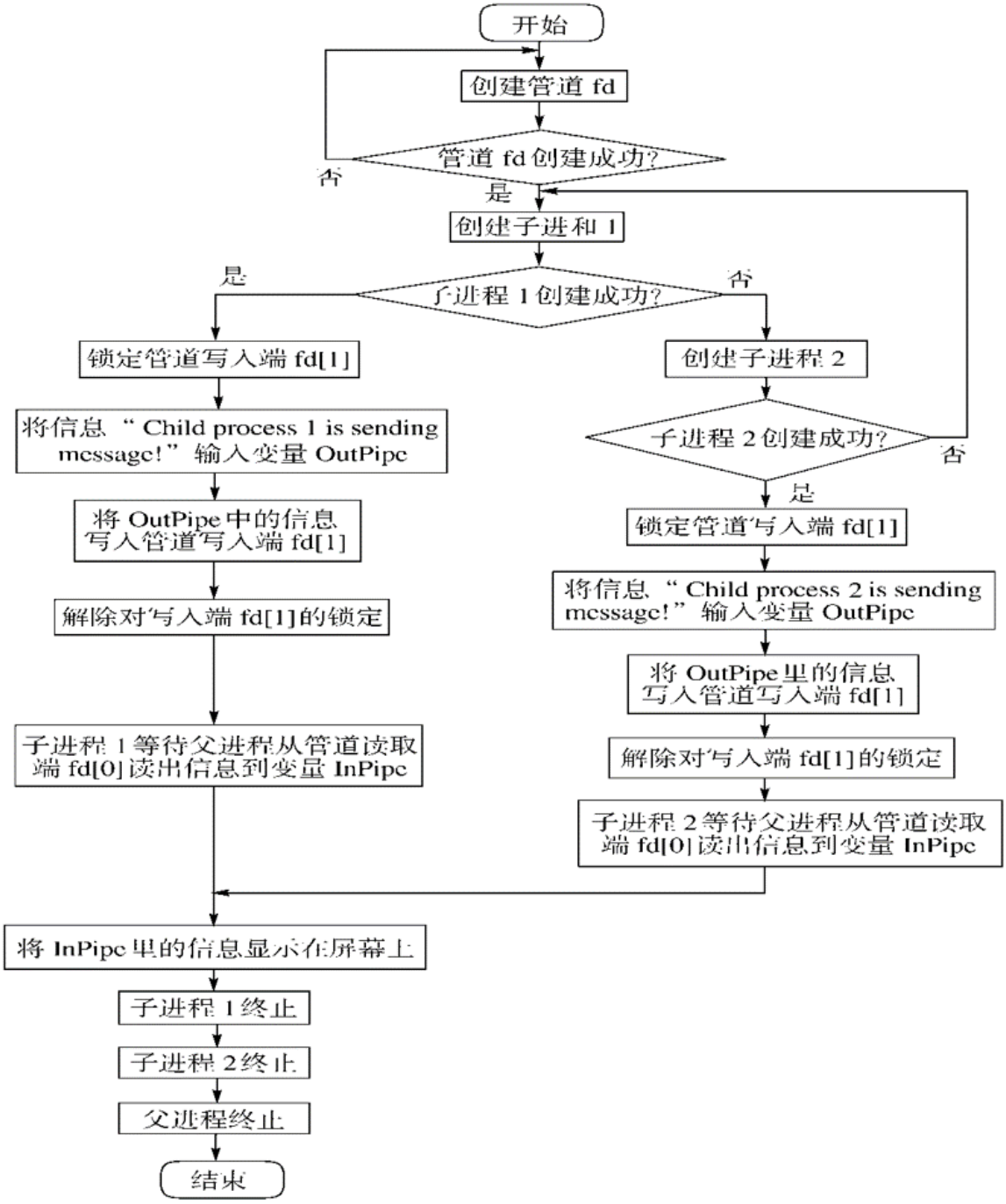
2.进程的管道通信

2.1实验目的

编程实现进程的管道通信，通过观察、分析实验现象，深入理解进程管道通信的特点，掌握管道通信的同步和互斥机制。

“管道”，是指用于连接一个读进程和一个写进程以实现他们之间通信的一个共享文件，又名pipe文件。向管道(共享文件)提供输入的发送进程(即写进程)，以字符流形式将大量的数据送入管道；而接受管道输出的接收进程(即读进程)，则从管道中接收(读)数据。由于发送进程和接收进程是利用管道进行通信的，故又称为管道通信。

管道通信程序流程图



猜想一下这个程序的运行结果。分析管道通信是怎样实现同步与互斥的，然后按照注释里的要求把代码补充完整，运行程序。 修改程序并运行，体会互斥锁的作用，比较有锁和无锁程序的运行结果，并解释之。

运行结果：

有锁程序：

```
if (pid1 == 0) {                    // 如果子进程1创建成功,pid1为进程号
    lockf(fd[1],1,0);               // 锁定管道

    for(int i =0;i<200;i++){     //  分200次每次向管道写入字符'1'
        OutPipe[i]=c1;

    write(fd[1],OutPipe,1);
    }
    // write(fd[1],OutPipe,200);

    sleep(2);                       // 等待读进程读出数据
    lockf(fd[1],0,0);               // 解除管道的锁定
    exit(0);                        // 结束进程1
}
```

```
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft-MIEngine-In-adenbpav.ejb" 1>"/tmp/Microsoft-MIEngine-Out-t0
slo5lp.1cq"
xjtuhpc@ubuntu:~/Code/OS_exp/source_2$
```

无锁程序：

```
if (pid1 == 0) {                // 如果子进程1创建成功,pid1为进程号
    // lockf(fd[1],1,0);        // 锁定管道

    for(int i =0;i<200;i++){    //  分200次每次向管道写入字符'1'
        OutPipe[i]=c1;

        write(fd[1],OutPipe,1);
    }
    // write(fd[1],OutPipe,200);

    sleep(2);                   // 等待读进程读出数据
    // lockf(fd[1],0,0);           // 解除管道的锁定
    exit(0);                    // 结束进程1
}
```

```
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft-MIEngine-In-xzfxpejn.gfz" 1>"/tmp/Microsoft-MIEngine-Out-to
hgnu4r.loy"
xjtuhpc@ubuntu:~/Code/OS_exp/source_2$ 
```

发现顺序并没有乱，可能是系统调度策略的问题。

遇到的问题：

不知道为什么在父进程和子进程中没有关闭管道的某一端，后来思考发现影响应当不大，同时这样做可以保证一直有着读和写的端口的开启，防止读写出现问题。

回答下列问题，写入实验报告。

1. 你最初认为运行结果会怎么样？

   最初以为程序运行后父进程读出应当只有一个111...1或222...2

2. 实际的结果什么样？有什么特点？试对产生该现象的原因进行分析。

   实际在父进程中成功的读出了200个“1”和200个“2”。注意到管道fd是在两个子进程都没有创建的时候就建立了，在最终的效果相当于一个管道由两个子进程共享，两个子进程都可以向管道中写入数据，在写入数据的时候对管道的写端进行了锁定以实现互斥，最后由父进程一次性全部读出。

3. 实验中管道通信是怎样实现同步与互斥的？如果不控制同步与互斥会发生什么后果？

   通过lockf函数对管道进行锁定。如果不控制同步和互斥可能会发生冲突，比如在一个子进程正在写的时候另一个子进程也在写，产生冲突等。

4. 把程序源代码附到实验报告后

3.内存分配与回收

实验目的

通过深入理解内存分配管理的三种算法，定义相应的数据结构，编写具体代码。 充分模拟三种算法的实现过程，并通过对比，分析三种算法的优劣。

1）掌握内存分配FF，BF，WF策略及实现的思路

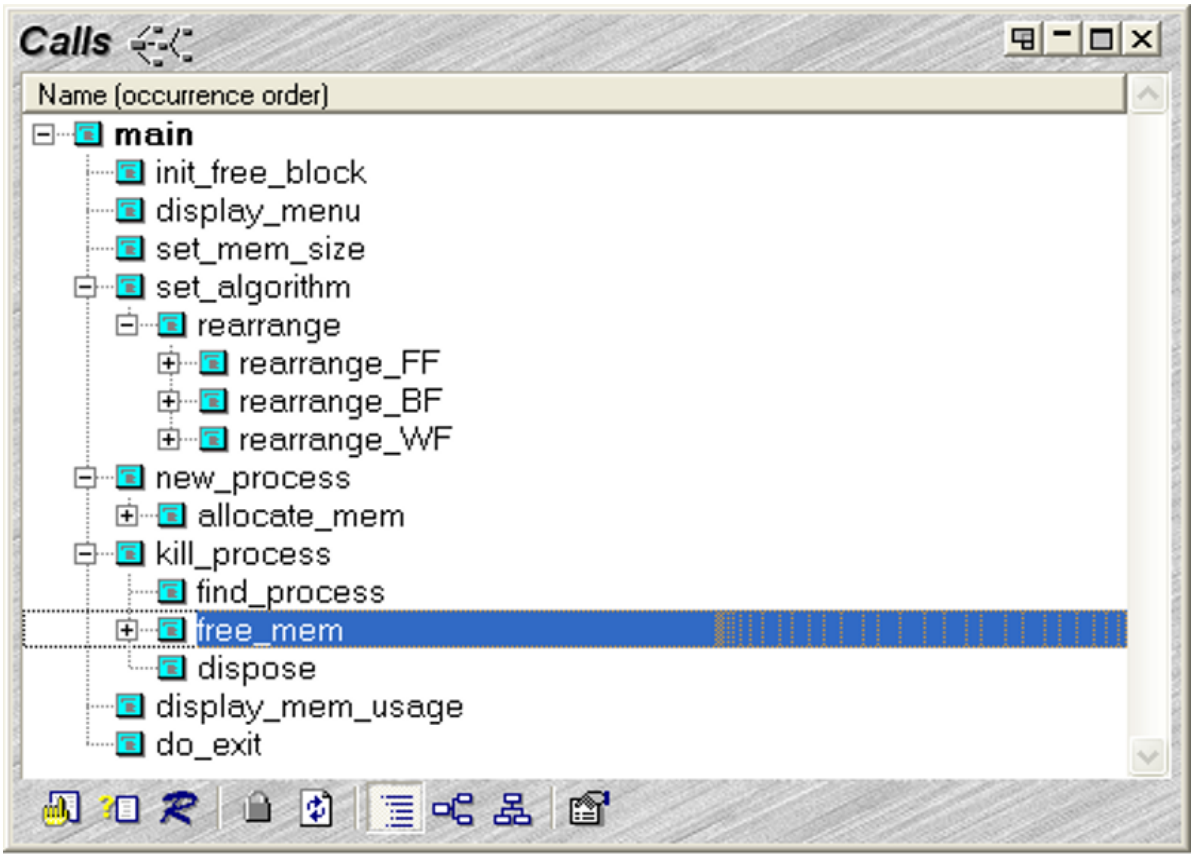
2）掌握内存回收过程及实现思路

3）参考给出的代码思路，实现内存的申请、释放的管理程序，调试运行，总结程序设计中出现的问题并找出原因，写出实验报告。

3.1主要功能

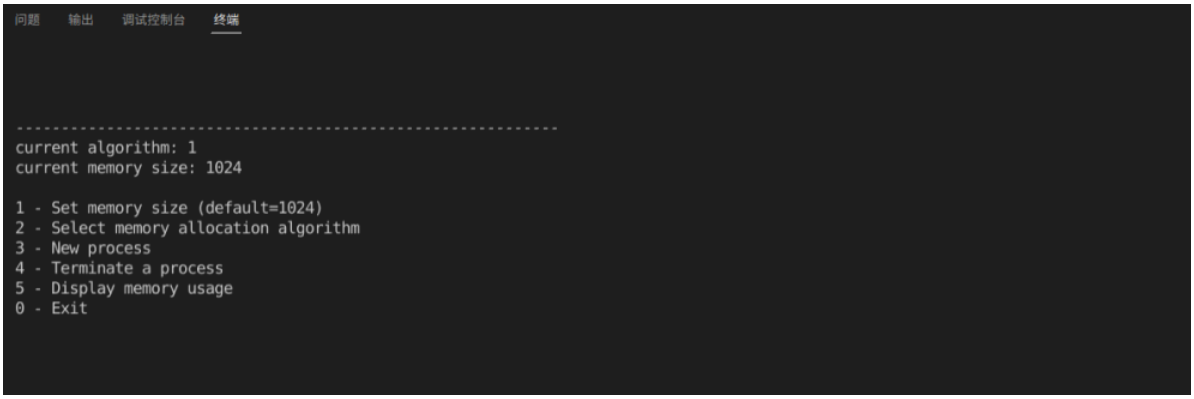
1 - Set memory size (default=1024) /*设置内存的大小*/
2 - Select memory allocation algorithm /* 设置当前的分配算法 */
*3 - New process /*创建新的进程，主要是获取内存的申请数量*/*
*4 - Terminate a process /*删除进程，归还分配的存储空间，并删除描述该进程内存分配的节点*/*
5 - Display memory usage / 显示当前内存的使用情况，包括空闲区的情况和已经分配的情况 */
0 - Exit

3.2 主要模块介绍

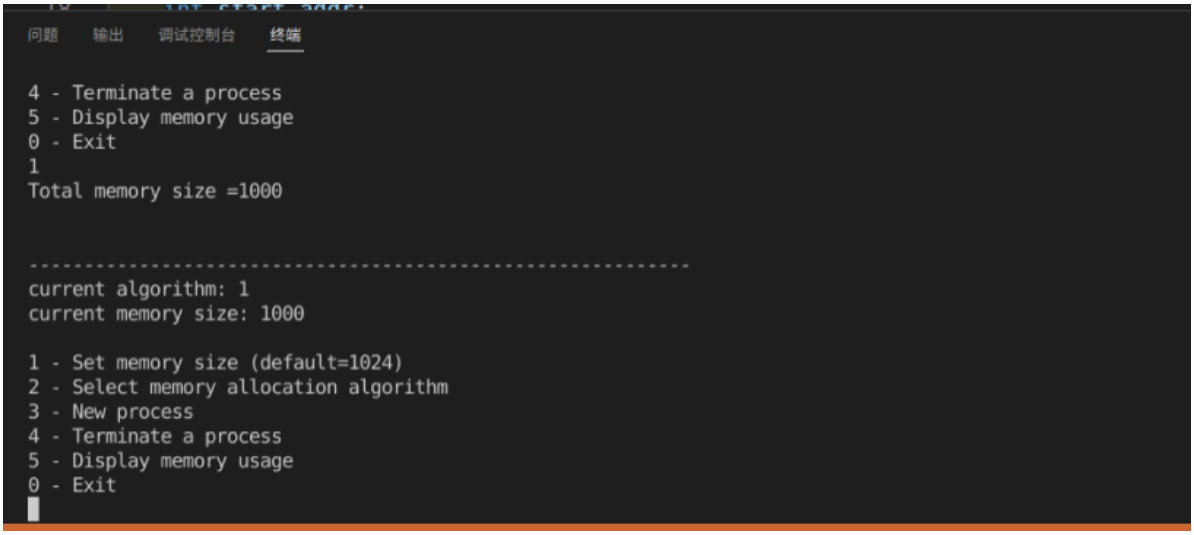


实验过程记录：

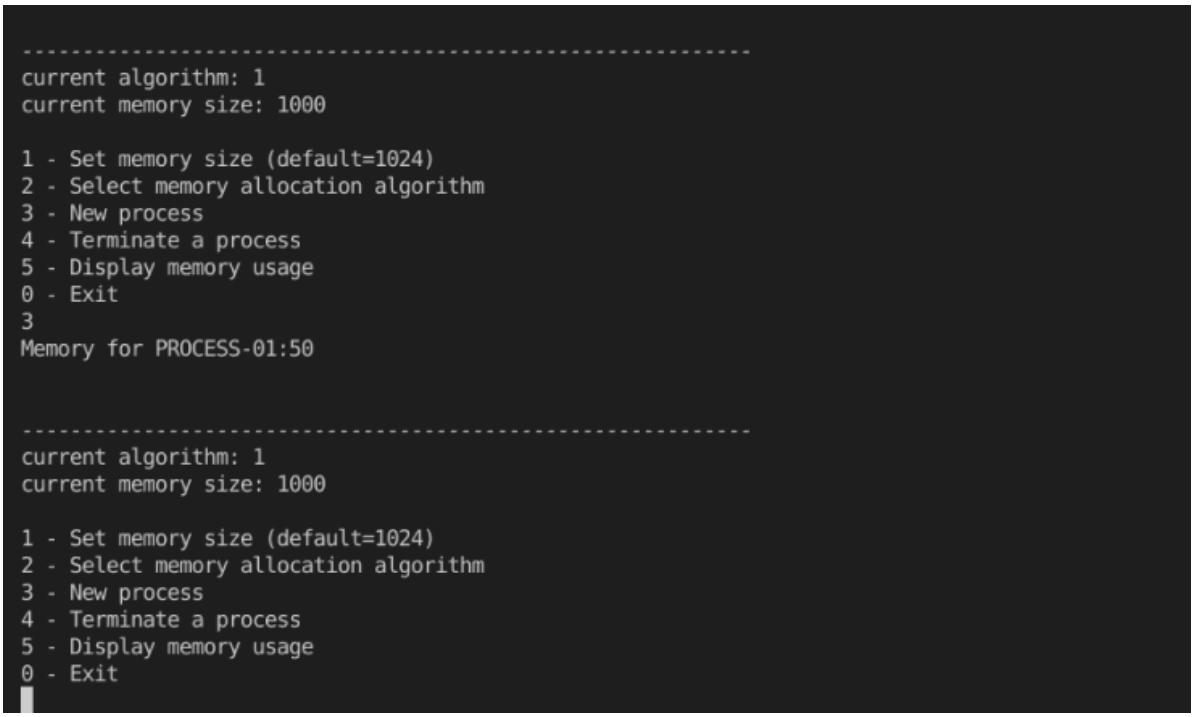
运行程序：



设置内存大小：

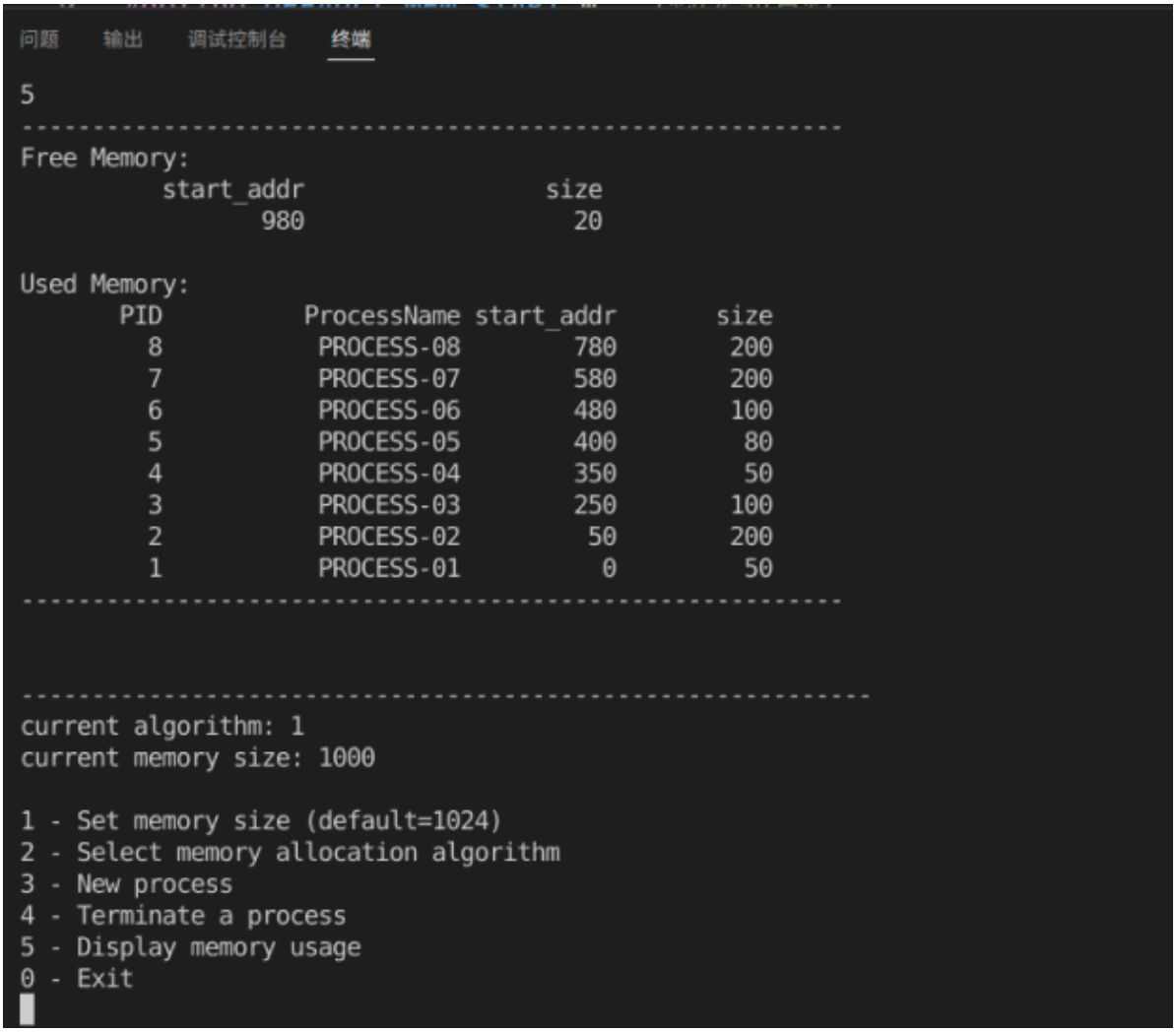


新建进程：

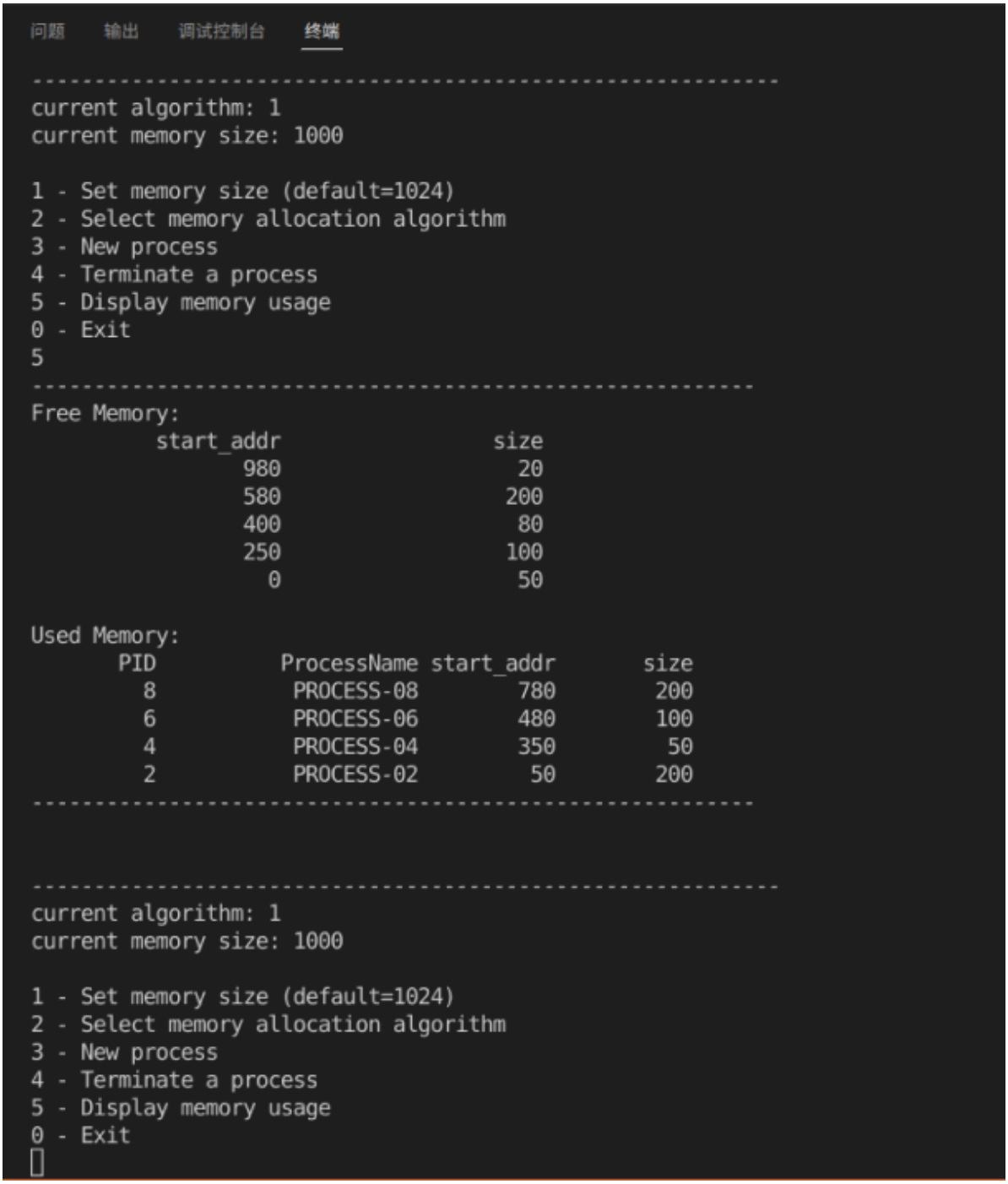


重复多次，直到内存几乎全部分配完毕。

查看内存分配状况：


问题 输出 调试控制台 终端
5
Free Memory:
start_addr size
980 20
Used Memory:
PID ProcessName start_addr size
8 PROCESS-08 780 200
7 PROCESS-07 580 200
6 PROCESS-06 480 100
5 PROCESS-05 400 80
4 PROCESS-04 350 50
3 PROCESS-03 250 100
2 PROCESS-02 50 200
1 PROCESS-01 0 50
current algorithm: 1
current memory size: 1000
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit

终止部分进程，空出位置：



选择内存分配算法：

选择First Fit：

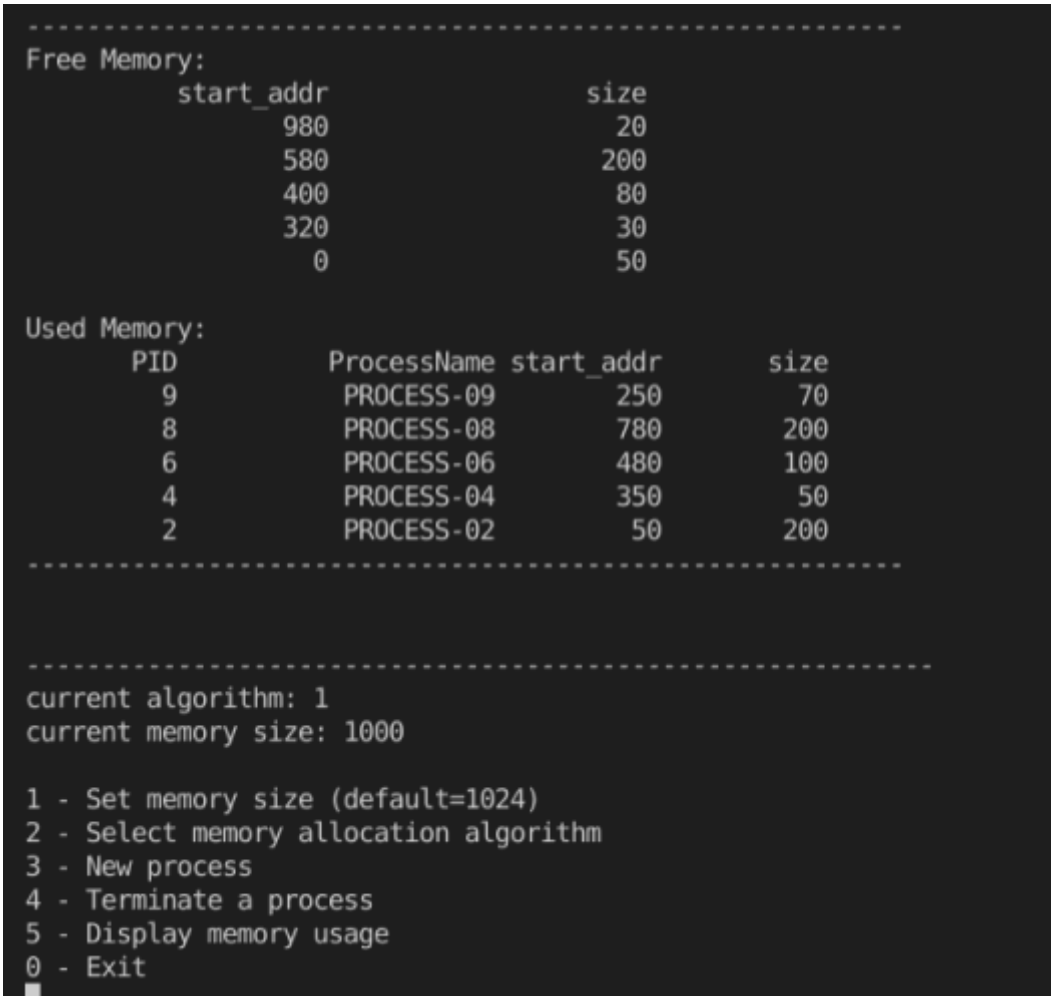
```
------------------------------------------------------------
current algorithm: 1
current memory size: 1000

1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
2
        1 - First Fit
        2 - Best Fit
        3 - Worst Fit
1


------------------------------------------------------------
current algorithm: 1
current memory size: 1000

1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
```

建立一个大小为70的进程，可以看到它正确完成了内存分配：

分配前：

```
------------------------------------------------------------
Free Memory:
          start_addr                size
                 980                  20
                 580                 200
                 400                  80
                 250                 100
                   0                  50

Used Memory:
       PID          ProcessName start_addr       size
         8           PROCESS-08        780        200
         6           PROCESS-06        480        100
         4           PROCESS-04        350         50
         2           PROCESS-02         50        200
------------------------------------------------------------
```

分配后：



选择Best fit，创建一个大小为20的进程：

```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
2
        1 - First Fit
        2 - Best Fit
        3 - Worst Fit
2


------------------------------------------------------------
current algorithm: 2
current memory size: 1000

1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
3
Memory for PROCESS-10:20


------------------------------------------------------------
current algorithm: 2
current memory size: 1000

1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
```

分配前：

```
------------------------------------------------------------
Free Memory:
         start_addr                size
                980                  20
                580                 200
                400                  80
                320                  30
                  0                  50

Used Memory:
       PID          ProcessName start_addr       size
         9           PROCESS-09        250         70
         8           PROCESS-08        780        200
         6           PROCESS-06        480        100
         4           PROCESS-04        350         50
         2           PROCESS-02         50        200
------------------------------------------------------------
```

分配后：

```
----------------------------------------------------------
Free Memory:
          start_addr                size
                 580                 200
                 400                  80
                   0                  50
                 320                  30

Used Memory:
       PID          ProcessName start_addr       size
        10           PROCESS-10        980         20
         9           PROCESS-09        250         70
         8           PROCESS-08        780        200
         6           PROCESS-06        480        100
         4           PROCESS-04        350         50
         2           PROCESS-02         50        200
----------------------------------------------------------
```

选择Worst fit，创建一个大小为20的进程：

```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
2
        1 - First Fit
        2 - Best Fit
        3 - Worst Fit
3


------------------------------------------------------------
current algorithm: 3
current memory size: 1000

1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
3
Memory for PROCESS-11:20


------------------------------------------------------------
current algorithm: 3
current memory size: 1000

1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
```

分配前：

```
5
-----------------------------------------------------------
Free Memory:
          start_addr                size
                 580                 200
                 400                  80
                   0                  50
                 320                  30

Used Memory:
       PID          ProcessName start_addr       size
        10           PROCESS-10        980         20
         9           PROCESS-09        250         70
         8           PROCESS-08        780        200
         6           PROCESS-06        480        100
         4           PROCESS-04        350         50
         2           PROCESS-02         50        200
-----------------------------------------------------------
```

分配后：

```
-----------------------------------------------------------
Free Memory:
          start_addr                size
                 320                  30
                   0                  50
                 400                  80
                 600                 180

Used Memory:
       PID          ProcessName start_addr       size
        11           PROCESS-11        580         20
        10           PROCESS-10        980         20
         9           PROCESS-09        250         70
         8           PROCESS-08        780        200
         6           PROCESS-06        480        100
         4           PROCESS-04        350         50
         2           PROCESS-02         50        200
-----------------------------------------------------------
```

退出程序：

```
-----------------------------------------------------------
current algorithm: 3
current memory size: 1000

1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
0
[1] + Done                    "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft-MIEngine-In-01h2xrhq.sbo" 1
>"/tmp/Microsoft-MIEngine-Out-fp20pme0.dqv"
xjtuhpc@ubuntu:~/Code/use/exp2$
```

遇到的问题：

getchar很难很好地处理在输入时键入的回车键，几次调试也没有很好地方法替代，最终改用scanf之后较好的解决了问题。

菜单的输出有些不清晰，美化了菜单的输出。