# 操作系统实验2

## 个人PC机上openEuler或Linux操作系统

徐中伟

计算机001

2206515211

实验1: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。

分析：我们在shell下运行起来一个程序，可以在这个进程正在运行的时候键盘输入一个Ctrl+C，就会看到这个进程被终止掉了，其实当我们键入Ctrl+C的时候是向进程发送了一个SIGINT信号，这时候产生了硬件中断则系统会从执行代码的用户态切入到内核态去处理这个信号，而一般这个信号的默认处理动作是终止进程，因此正在运行的进程就会被终止了。所以这也是本次实验为什么2和3信号的实验结果看上去一模一样的原因。

源代码&&实验结果

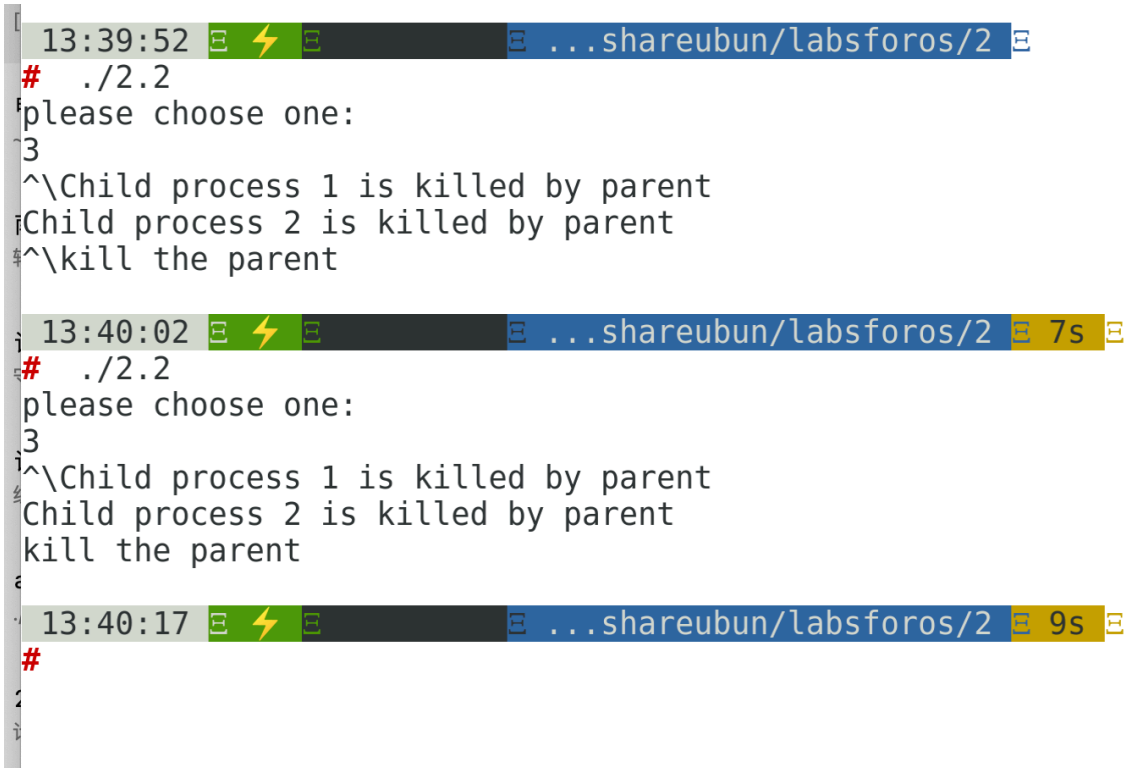
```
#include<stdio.h>
#include<unistd.h>
#include<signal.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<stdlib.h>

int wait_flag=1;

void stop(){
   wait_flag = 0;
}


int main(int argc, char** argv){
   pid_t pid1,pid2;
   printf("please choose one:\n");
   int n;
```

```
scanf("%d",&n);
signal(n,stop);
while(wait_flag);
while((pid1 = fork())<0);
if(pid1>0)//for parent
{
   while((pid2 = fork())<0);
   if(pid2>0)
   {
      wait_flag = 1;

     // int a = alarm(3);
      sleep(5);
     kill(pid1,16);
      kill(pid2,17);
      wait(0);
      wait(0);
      printf("kill the parent\n");
      exit(0);
   }else{
      wait_flag =1;
      signal(17,stop);
      printf("Child process 2 is killed by parent\n");
      exit(0);
   }
}
else{
   wait_flag = 1;
```

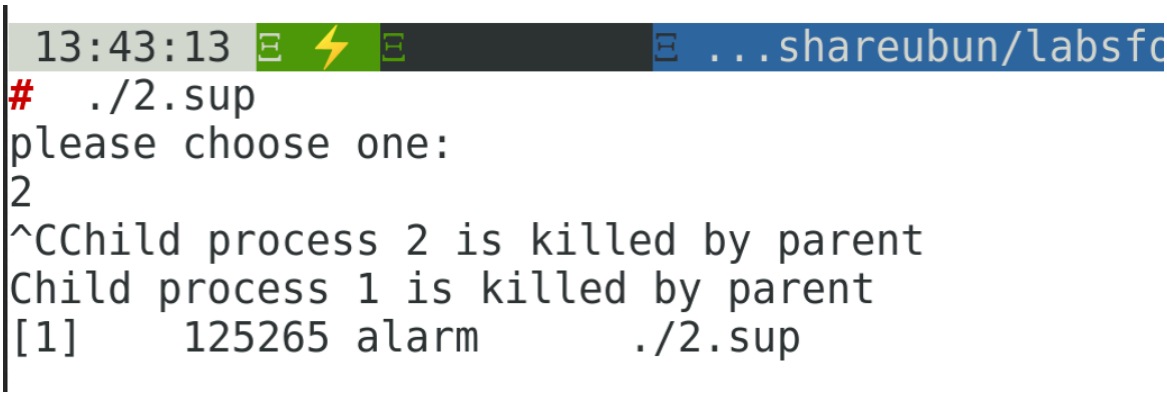
```
        signal(16,stop);
        printf("Child process 1 is killed by parent\n");
        exit(0);
    }

}
```

```
 13:32:36 [redacted] ...shareubun/labsforos/2
#  ./2.2
please choose one:
2
^CChild process 1 is killed by parent
Child process 2 is killed by parent
^Ckill the parent

 13:37:44 [redacted] ...shareubun/labsforos/2 11s
#  ./2.2
please choose one:
2
^CChild process 1 is killed by parent
Child process 2 is killed by parent
kill the parent

 13:37:56 [redacted] ...shareubun/labsforos/2 10s
```

从实验结果不难看出，当输入信号为2时，进程中断，同时，如果继续输入，那么进程便会提前终止，如果不输入，根据代码会五分钟内终止。




将信号换成3后也是同一个效果，此时表现相同的原因则是因为中断的默认处理函数为终止当前进程。




设置alarm信号则会提前终止，因为alarm到了指定时间会发送一个终止信号。

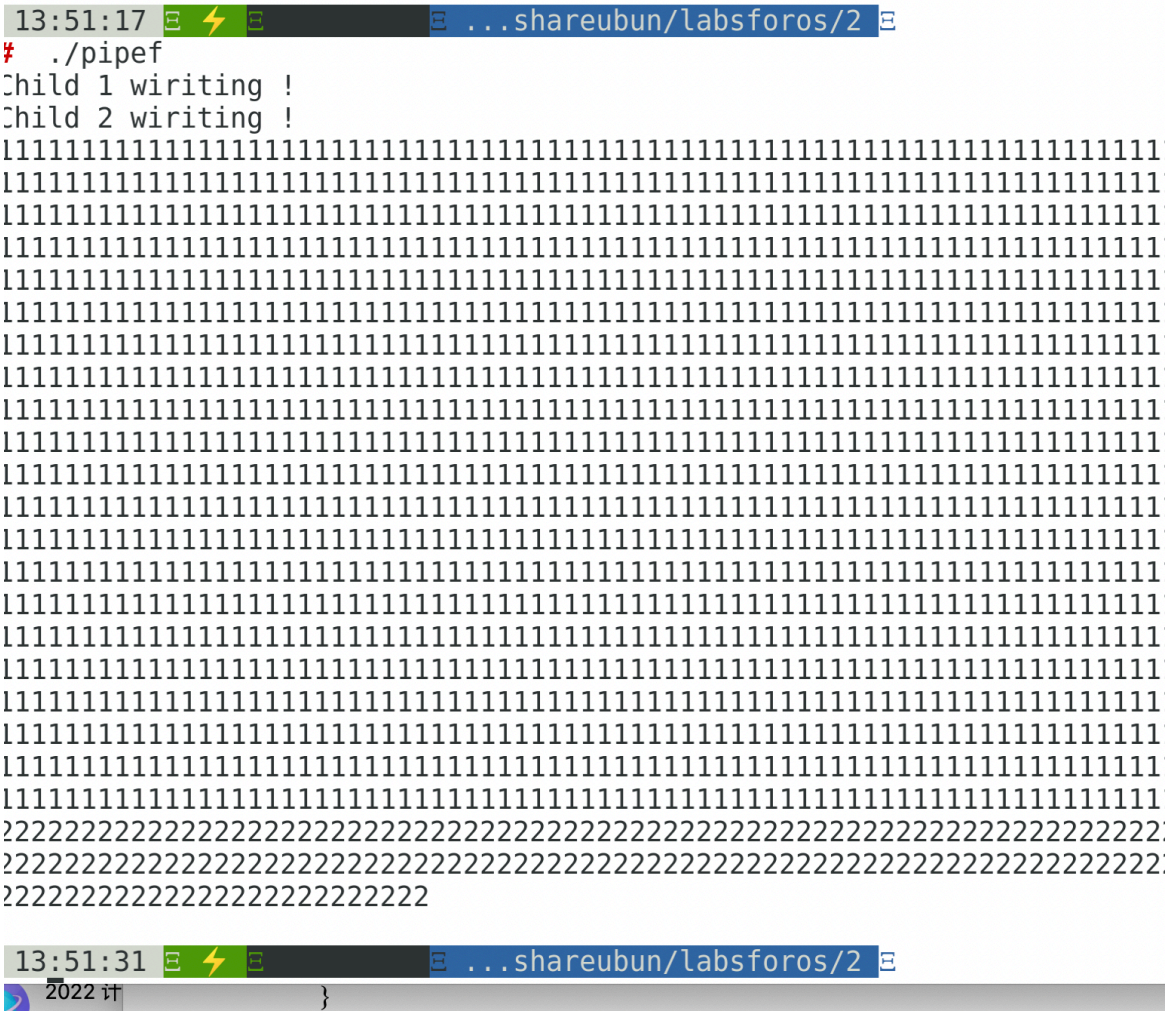
实验2:编程实现进程的管道通信，通过观察、分析实验现象，深入理解进程管道通信的特点，掌握管道通信的同步和互斥机制。

实验源码&&结果截图:

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <unistd.h>
#include  <signal.h>
#include  <sys/wait.h>
#include  <sys/types.h>

int main(void) {
    int pipefd[2];
    int pid1, pid2;
    char OutPipe[100], InPipe[4096];          // 定义两个字符数组

    // create pipe
    pipe(pipefd);

    // create child 1
    while((pid1 = fork( )) == -1);
    // child 1
    if(pid1 == 0) {
        // load data
        //sprintf(OutPipe,"\n Child process 1 is sending message!\n");
        // lock out pipe
        lockf(pipefd[1], 1, 0);
        // write data
```

```
        printf("Child 1 wiriting !\n");
        for(int i=0;i<2000;i++){
            OutPipe[0] = '1';
            write(pipefd[1], OutPipe, 1);
        }
        // wait reading
        sleep(3);
        // free out pipe
        lockf(pipefd[1] ,0 ,0);
        exit(0);
    }
    else {
        // create child 2
        while((pid2 = fork()) == -1);                    // 若进程2创建不成功,
则空循环
    // child 2
        if(pid2 == 0) {
            // load data
            //sprintf(OutPipe,"\n Child process 2 is sending message!\n");
            // lock out pipe
            lockf(pipefd[1], 1, 0);
            // write data
            printf("Child 2 wiriting !\n");
            for(int i=0;i<2000;i++){
            OutPipe[0] = '2';
            write(pipefd[1], OutPipe, 1);
        }
            // wait reading
            sleep(3);
```

```
            // free out pipe
            lockf(pipefd[1], 0, 0);
            exit(0);
        }
    // parent
        else {
            wait(0);
            // lock in pipe
            close(pipefd[1]);
            lockf(pipefd[0], 1, 0);
            // read data
            char t = '\0';
            write(pipefd[0],&t,1);
            read(pipefd[0], InPipe, 4000);
            // free in pipe
            lockf(pipefd[0], 0, 0);
            // print data
            printf("%s\n",InPipe);
        }
    }

    return 0;
}
```




分析：当两个进程需要相互之间通信时可以选择诸如管道，以及共享内存，管道通信的一般流程是：在一个进程中创建管道，通过fork()创建子进程，关闭管道多余端口，使父子进程与管道形成单向通道，进行数据传输。Linux标准I/O库中封装了两个函数——popen()和pclose()，使用这两个函数即可完成管道通信的流程。通俗的说，管道就是创建了一个连接两个进程的文件描述符号。

实验3:

- **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。**

注：因为实验3实现相对内容较多，所以提供额外说明：

**memory_manager.h:**

定义了一些基本的常量（内存大小，空间大小，地址起始位置等）（1～9）

定义了基本的分配算法（11～14）

首先定义了block类：（23～79）

数据：内存大小，地址所在位置，下一个内存块的位置

函数：（注：序号顺序即位实现顺序）

1.生成一个空闲内存块

2.在当前的内存块链表后生成一个新的内存块

3.根据输入的参数确定比较地址或大小，方便后续的函数调用

4，交换两个块

定义已经分配了的内存块：（80～94）

数据：进程号，快大小，起始地址，下一个已经分配的块，其他进程

定义管理者：（96～）

数据：当前占用内存的进程的数量，剩余的内存的空间，内存分配算法，标志某一个内存块是否被占用

函数：

1.生成函数

2.展示当前内存分配情况的函数mem

3.设置内存分配大小

4.调整内存分配

5.选择内存分配算法

6.创建新的进程

7.分配内存

8.回收内存

9.杀死进程

10.查询某一进程

**11.**显示内存使用情况

**12.**释放内存

**memory_manager.cpp(**用于实现**.h**中所定义的函数）

对应函数**manager**的实现（序号为准）：

**1.**初始化（**9**～**15**），设置基本的内存块的大小（**21**～**25**），选择相应的操作（**27**～**36**）

**2.**显示相对应的操作（**42**～**51**）检查内存块是否被设置过

**3.**设置内存块的大小（**54**～**74**）

**4.**根据所选择的算法调整内存链（**76**～**99**）

**5.**首先调用判断**args**，来确定选择的算法，然后根据选择的算法来确定**target**内存

块的位置（即下一个要被分配到内存块），其中调用了**compare**函数来决定该怎么指（比如按照**ff**算法就是找第一个空闲的。。。。。）

**6.**选择算法

**7.**调用函数**8**去分配给进程相对应的空间

**8.**相对应的**process**的内存分配，先根据所选择的算法对剩余的空间进行分配，如果剩余的空间不够，就将现在的所占用的内存释放一部分。

**9.**利用内存紧缩技术，从地址起始处开始，将内存填满，同时下一块内存的容量加上相对应的容量，然后再重新创建进程块

**10.**通过保存的**map**来寻找相对应的**pid**

**11.**创建一个新的内存块，然后将该内存块指向**head**，然后删除想要删除的内存块

**12.**展示设置