# 实验一 进程和进程通信

**陈志扬 15331046**

注：由于在csdn上写报告，所以图片带有水印，请TA谅解。

---

## 一、 实验目的

1. 加深对进程概念的理解，明确进程和程序的区别。进一步认识并发执行的实质。
2. 了解信号处理。
3. 认识进程间通信（IPC）：进程间共享内存。
4. 实现shell：了解程序运行。

---

## 二、 实验运行环境

虚拟机VMware下的Ubuntu16.04系统

---

## 三、 实验内容

1. 进程的创建实验
2. 信号处理实验
3. 进程间共享内存实验
4. 实现shell的要求

---

## 四、 实验过程

①进程的创建实验

(1) 将下面的程序编译运行，并解释现象

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    int pid1 = fork();
    printf("**1**\n");
    int pid2 = fork();
    printf("**2**\n");

    if (pid1 == 0) {
        int pid3 = fork();
        printf("**3**\n");
    }
    else {
        printf("**4**\n");
    }
    return 0;
}
```

编译运行得到的结果如下图：

```
linjiafengyang@ubuntu:~/Desktop/操作系统实验一$ ./createProcess1
**1**
**2**
**4**
**2**
**4**
**1**
**2**
**3**
**3**
**2**
linjiafengyang@ubuntu:~/Desktop/操作系统实验一$ **3**
**3**
```



注意可能得到的结果不止一个，这里选取上图的结果来作分析。
可能的原因如下：

1. 首先执行父进程main，它创建子进程m-p1，进而输出1，再创建子进程m-p2，再输出2，然后执行if判断，此时pid1不为0，所以再输出4。main执行完毕，此过程输出1、2、4；

2. 两个子进程相互竞争，先执行子进程m-p2，它执行后面的代码，先输出2，然后执行if判断，此时该子进程继承了父进程main的pid1值，pid1不为0，所以输出4。m-p2执行完毕，此过程输出2、4；
3. 接着执行子进程m-p1，先输出1，再创建一个子进程m-p1-p2，继续输出2，然后执行if判断，此时pid1为0，创建子进程m-p1-p3，然后输出3。m-p1执行完毕，此过程输出1、2、3；
4. 然后执行子进程m-p1-p3，输出3。m-p1-p3执行完毕；
5. 接着执行m-p1-p2，先输出2，然后执行if判断，此时pid1为0，创建子进程m-p1-p2-p3，输出3。m-p1-p2执行完毕，此过程输出2、3；
6. 最后执行子进程m-p1-p2-p3，直接输出3。m-p1-p2-p3执行完毕。

所以：该结果为1、2、4、2、4、1、2、3、3、2、3、3.

---

(2) 编写一段程序，使用系统调用fork()创建两个子进程。当此程序运行时，在系统中有一个父进程和两个子进程活动。让每一个进程在屏幕上显示一个字符：父进程显示字符a；子进程分别显示字符b和字符c。试观察记录屏幕上的显示结果并分析原因。

所编写的程序如下：

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    int pid1, pid2;

    pid1 = fork();
    if (pid1 == 0) {
        printf("b\n");
    }
    else {
        pid2 = fork();
        if (pid2 == 0) {
            printf("c\n");
        }
        else {
            printf("a\n");
        }
    }
    return 0;
}
```



编译运行得到的结果如下：

```
linjiafengyang@ubuntu:~/Desktop/操作系统实验一$ ./createProcess2
a
c
b
```



注意：可能的结果有多种，这里选取上图的结果来分析。
原因：

1. 首先创建父进程main，执行相应代码，先创建一个子进程m-p1，然后执行if判断，此时pid1不为0，创建子进程m-p2，继续执行if判断，此时pid2不为0，所以输出a，父进程main执行完毕。
2. 接着子进程m-p1和m-p2竞争，先执行m-p2，此时pid2为0，所以输出c，子进程m-p2执行完毕。
3. 最后执行子进程m-p1，此时pid1为0，输出b，m-p1执行完毕。

所以：输出结果为acb

(3) 下面程序将在屏幕上输出的字符X、数字1和0各多少个？为什么？

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    int i, a = 0;
    pid_t pid;
    if ((pid = fork())) {
        a = 1;
    }
    for (i = 0; i < 2; i++) {
        printf("X");
    }
    if (pid == 0) {
        printf("%d\n", a);
    }
    return 0;
}
```



编译运行得到的结果如下：

```
linjiafengyang@ubuntu:~/Desktop/操作系统实验一$ ./createProcess3
XXXX0
```



分析出现该结果的原因：

1. 首先创建父进程main，执行相应代码，先创建子进程m-p1，此时a=1，进入for循环输出两个X，执行if判断，此时pid不为0，所以不输出a的值，main执行完毕，此过程输出XX。
2. 接下来执行子进程m-p1，此时直接进入for循环，不会改变a的值，即之前父进程main对a赋值为1的语句不会在子进程中执行，a还是为0，进入for循环后输出两个X，然后执行if判断，此时pid为0所以输出a的值为0，m-p1执行完毕，此过程输出XX0。

所以：输出结果为XXXX0

(4) 如果将上面(3)的main函数修改如下，则屏幕上输出的字符X、数字1和0各多少个？为什么？

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    int i, a = 0;
    pid_t pid[2];
    for (i = 0; i < 2; i++) {
        if ((pid[i] = fork())) {
            a = 1;
        }
        printf("X\n");
    }
    if (pid[0] == 0) {
        printf("%d\n", a);
    }
    if (pid[1] == 0) {
        printf("%d\n", a);
    }
    return 0;
}
```



编译运行得到的结果如下：

```
linjiafengyang@ubuntu:~/Desktop/操作系统实验一$ ./createProcess4
XXXX1
XX1
XX0
0
```



注意得到的结果可能不止一个，这里选取上图所得的结果来分析。
可能原因：由于输出X的时候是没有换行的，输出X的时候是将X保存在一个缓冲区里等程序结束后才输出的，在fork()调用时会复制缓冲区，因此父进程在第一次循环创建一个子进程时，这个子进程的缓冲区已经有了一个X，之后再输出一个X，也就是这个子进程也会输出两个X。
由程序代码分析可知父子进程共有4个，所以共有8个X被输出，父进程的pid[0]和pid[1]都不为0，所以不会输出a的值，而三个子进程中一个pid[0]==0，a=1；一个子进程pid[1]==0，a=1；另一个子进程pid[0]==pid[1]==0，但是a=0，所以输出结果中有两个1和两个0

所以：输出结果为XXXX1 XX1 XX0 0

---

②信号处理实验
(a)编写一段程序，使用系统调用fork()创建两个子进程，再用系统调用signal()让父进程捕捉键盘上来的中断信号（即按Ctrl C键），当捕捉到中断信号后，父进程调用kill()向两个子进程发出信号，子进程捕捉到信号后，分别输出下面信息后终止：
child process 1 is killed by parent!
child process 2 is killed by parent!
父进程等待两个子进程终止后，输出以下信息后终止：
parent process is killed!

所编写的程序如下图：

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

void waiting();
void stop(int signum);

int wait_mark;

int main() {
    int pid1, pid2;
    pid1 = fork();
    if (pid1 > 0) {
        pid2 = fork();
        if (pid2 > 0) {
            wait_mark = 1;
            signal(SIGINT, stop);//设置收到信号ctrl c时执行stop函数
            waiting();
            kill(pid1, SIGINT);//向进程p1发出信号SIGINT
            kill(pid2, SIGINT);//向进程p2发出信号SIGINT
            waitpid(pid1, NULL, 0);
            waitpid(pid2, NULL, 0);
```

```
            printf("parent process is killed!\n");
            exit(0);
        }
        else {
            wait_mark = 1;
            signal(SIGINT, stop);
            waiting();
            printf("child process 2 is killed by parent!\n");
            exit(0);
        }
    }
    else {
        wait_mark = 1;
        signal(SIGINT, stop);//设置收到信号SIGINT时执行stop函数
        waiting();
        printf("child process 1 is killed by parent!\n");
        exit(0);
    }
}
void waiting() {
    while (wait_mark != 0);
}
void stop(int signum) {
    wait_mark = 0;
}
```



编译运行所得结果如图：满足实验要求。（不止这种结果）

```
linjiafengyang@ubuntu:~/Desktop/操作系统实验一$ ./signal1
^Cchild process 2 is killed by parent!
child process 1 is killed by parent!
parent process is killed!
linjiafengyang@ubuntu:~/Desktop/操作系统实验一$
```



(b)在上述(a)中的程序中增加语句
signal(SIGINT, SIG_IGN) 和
signal(SIGQUIT, SIG_IGN)，观察执行结果并分析原因。这里
signal(SIGINT, SIG_IGN) 和
signal(SIGQUIT, SIG_IGN)分别为忽略“Ctrl Z”键信号以及忽略“Ctrl C”中断信号。

加入两个signal的代码如下：

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

void waiting();
void stop(int signum);

int wait_mark;

int main() {
    int pid1, pid2;
    signal(SIGINT, SIG_IGN);
    signal(SIGQUIT, SIG_IGN);
    pid1 = fork();
    if (pid1 > 0) {
        pid2 = fork();
        if (pid2 > 0) {
            wait_mark = 1;
            signal(SIGINT, stop);//设置收到信号ctrl c时执行stop函数
            waiting();
            kill(pid1, SIGINT);//向进程p1发出信号SIGINT
            kill(pid2, SIGINT);//向进程p2发出信号SIGINT
            waitpid(pid1, NULL, 0);
            waitpid(pid2, NULL, 0);
            printf("parent process is killed!\n");
            exit(0);
        }
        else {
            wait_mark = 1;
            signal(SIGINT, stop);
            waiting();
            printf("child process 2 is killed by parent!\n");
            exit(0);
        }
```

```
    }
    else {
        wait_mark = 1;
        signal(SIGINT, stop);//设置收到信号SIGINT时执行stop函数
        waiting();
        printf("child process 1 is killed by parent!\n");
        exit(0);
    }
```



编译运行的结果如下图：出现该结果的原因是按ctrl c键会被忽略，由于子进程m-p1先被创建，此时会先执行m-p1输出第一条child1，然后再执行m-p2输出第二条child2，最后执行main输出parent

```
linjiafengyang@ubuntu:~/Desktop/操作系统实验一$ ./signal2
^Cchild process 1 is killed by parent!
child process 2 is killed by parent!
parent process is killed!
linjiafengyang@ubuntu:~/Desktop/操作系统实验一$ ./signal2
^Z
[1]+  已停止               ./signal2
```



---

③进程间共享内存实验
完成第三章练习3.10的程序，利用共享内存的方法调用子进程来输出斐波那契数列。

根据要求编写的程序代码如下图：

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/shm.h>
#include <sys/stat.h>

// 斐波那契数列最大个数
#define MAX_SIZE 30

// 定义结构体share_data
// 包含最大长度为MAX_SIZE的数组fib_sequence(int)和斐波那契数列的个数(int)
typedef struct
{
    int fib_sequence[MAX_SIZE];
    int sequence_size;
} share_data;

int main(int argc, char const *argv[])
{
    // 程序运行时需要输入argv的大小
    if (argc != 2) {
        printf("Error:only one arguement is input.\n");
        exit(0);
    }
    // 利用atoi函数将字符串转化为整型数
    else if (atoi(argv[1]) <= 0 || atoi(argv[1]) > 30) {
        printf("Error:arguement should be between 0 and 30.\n");
        exit(0);
    }
    // 共享存储区id
    int segment_id;
    const int size = sizeof(share_data);
    share_data *shared_memory;
    // 创建或打开共享存储区shmget
    segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
    // 连接共享存储区shmat
    shared_memory = (share_data *)shmat(segment_id, NULL, 0);
    shared_memory -> sequence_size = atoi(argv[1]);
    int pid;
```

```
    pid = fork();
    if (pid < 0) {
        printf("Error:fail to create a child process.\n");
        exit(0);
    }
    // 子进程执行中，共享内存(数据)
    else if (pid == 0) {
        shared_memory -> fib_sequence[0] = 0;
        shared_memory -> fib_sequence[1] = 1;
        if (atoi(argv[1]) > 2) {
            int i = 2;
            for (; i < shared_memory -> sequence_size; i++) {
                shared_memory -> fib_sequence[i] = shared_memory -> fib_sequence[i - 1]
                + shared_memory -> fib_sequence[i - 2];
            }
        }
    }
    // 子进程执行完毕，main执行，从共享内存中读取并输出结果
    else {
        wait(0);
        printf("The child process is finished.\nThe result is:\n");
        for (int j = 0; j < atoi(argv[1]); j++) {
            printf("%d\n", shared_memory -> fib_sequence[j]);
        }
        printf("\n");
    }
    // 拆除共享存储区连接shmdt
    shmdt(shared_memory);
    // 共享存储区控制shmctl
    shmctl(segment_id, IPC_RMID, NULL);
    return 0;
}
```



编译运行可得到下面的结果：注意这里需要加上参数，可以得出程序编写正确，实现了进程间共享内存。


---

④实现shell的要求
实验要求：
完成课本上第三章的项目：实现shell。除此之外满足下面要求：

- 在shell下，按ctrl c时不会终止shell；
- 实现程序的后台运行。

根据老师的指导课件，可编写下面的程序实现简单shell

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

#define MAXLINE 80
#define BUFFER_SIZE 50
char buffer[BUFFER_SIZE];
char *history[10][10];// 最多存十条指令，第一个下标表示命令的数量，第二个下标表示这些命令的第n个命令
int pos = 0;// 下一条即将被输入的命令的位置
int com_l[11] = {0};

 /*
  * setup()用于读入下一行输入的命令，并将它分成没有空格的命令和参数存于数组args[]中，
  * 用NULL作为数组结束的标志
  */
void setup(char inputBuffer[], char *args[], int *background) {
    /*
     * length:命令的字符数目
     * i:循环变量
     * start:命令的第一个字符位置
     * ct:下一个参数存入args[]的位置
     */
    int length, i, start, ct;
    ct = 0;
    // 读入命令行字符，存入inputBuffer
    length = read(STDIN_FILENO, inputBuffer, MAXLINE);
    start = -1;
    // 输入ctrl d，结束shell程序
    if (length == 0) {
        exit(0);
    }
    // 出错时用错误码-1结束shell
    else if (length < 0) {
        printf("Error in reading command.\n");
        exit(-1);
    }

     else {
        // 检查inputBuffer中的每一个字符
        for (i = 0; i < length; i++) {
            switch(inputBuffer[i]) {
                // 字符为分割参数的空格或制表符
                case ' ':
                case '\t':
```

```
                    if (start != -1) {
                        args[ct] = &inputBuffer[start];
                        ct++;
                    }
                    inputBuffer[i] = '\0';// 设置C string的结束符
                    start = -1;
                    break;
                // 命令行结束
                case '\n':
                    if (start != -1) {
                        args[ct] = &inputBuffer[start];
                        ct++;
                    }
                    inputBuffer[i] = '\0';
                    args[ct] = NULL;// 命令及参数结束
                    break;
                // 置命令在后台运行
                case '&':
                    *background = 1;
                    inputBuffer[i] = '\0';
                    break;
                // 其他字符
                default:
                    if (start == -1) {
                        start = i;
                    }
            }
        }
    }
    if (ct != 0) {
        args[ct] = NULL;
    }
}
```

```
// 按ctrl c输出存放在history中的命令记录
void handle_SIGINT(int signum) {
    write(STDOUT_FILENO, buffer, strlen(buffer));
    printf("The command history is:\n");
    int i = pos;
    for (int count = 10; count > 0; count--) {
        for (int j = 0; j < com_l[i]; j++) {
            printf("%s ", history[i][j]);
        }
        printf("\n");
        i = (i + 1) % 10;
    }
    printf("\nCOMMAND->");
    fflush(stdout);
    return;
}
```

```c
int main() {
    char inputBuffer[MAXLINE];// 这个缓存用来存放输入的命令
    int background;// ==1时，表示在后台运行命令，即在命令后加上'&'
    char *args[MAXLINE / 2 + 1];// 命令最多40个参数
    int i, j;
    for (i = 0; i < 10; i++) {
        for (j = 0; j < 10; j++) {
            history[i][j] = (char*)malloc(80 * sizeof(char));
        }
    }

    strcpy(buffer, "\nCaught Control C\n");
    signal(SIGINT, handle_SIGINT);

    while (1) {
        background = 0;
        printf("COMMAND->");
        fflush(stdout);// 输出输出缓存内容用fflush(stdout)
        // 获取下一个输入的命令
        setup(inputBuffer, args, &background);

        // 如果不是r型指令
        if ((args[0] != NULL) && (strcmp(args[0], "r") != 0)) {
            if (args[0] != "\n") {      // 存入二维数组history中
                for (i = 0; args[i] != NULL; i++) {
                    strcpy(history[pos][i], args[i]);
                }
                com_l[pos] = i;
                pos = (pos + 1) % 10;
            }
        }
```

```
        // 如果是r型指令
        if ((args[0] != NULL) && (strcmp(args[0], "r") == 0)) {
            // 只是r型指令
            // 存入二维数组history中
            if (args[1] == NULL) {
                i = (pos + 9) % 10;
                for (j = 0; j < com_l[i]; j++) {
                    strcpy(history[pos][j], history[i][j]);
                }
                com_l[pos] = j;
                pos = (pos + 1) % 10;
            }
            // 是r x指令
            else {
                i = pos;
                for (int count = 10; count > 0; count--) {
                    i = (i + 9) % 10;
                    if (strncmp(args[1], history[i][0], 1) == 0) {
                        for (j = 0; j < com_l[i]; j++) {
                            strcpy(history[pos][j], history[i][j]);
                        }
                        com_l[pos] = j;
                        pos = (pos + 1) % 10;
                    }
                }
            }
        }
```

```
154         // 用fork()创建一个子进程
155         pid_t pid = fork();
156         // 创建失败
157         if (pid < 0) {
158             printf("Fork failed!\n");
159         }
160         // 子进程将调用execvp()执行命令, 即execvp(args[0], args);
161         else if (pid == 0) {
162             if (strcmp(args[0], "r") != 0) {
163                 execvp(args[0], args);
164                 int k = (pos - 1) % 10;
165             }
166             else {
167                 char *newargs[MAXLINE / 2 + 1];
168                 for (i = 0; i < MAXLINE / 2 + 1; i++) {
169                     newargs[i] = (char*)malloc((MAXLINE / 2 + 1) * sizeof(char));
170                 }
171                 pos = (pos + 9) % 10;
172                 history[pos][0] = '\0';
```

```
172                 history[pos][0] = '\0';
173                 if (args[1] == NULL) {
174                     i = (pos + 9) % 10;
175                     for (j = 0; j < com_l[i]; j++) {
176                         strcpy(newargs[j], history[i][j]);
177                     }
178                     newargs[j] = NULL;
179                     execvp(newargs[0], newargs);
180                     exit(0);
181                 }
182                 else {
183                     i = pos;
184                     for (int count2 = 10; count2 > 0; count2--) {
185                         i = (i + 9) % 10;
186                         if (strncmp(args[1], history[i][0], 1) == 0) {
187                             for (j = 0; j < com_l[i]; j++) {
188                                 strcpy(newargs[j], history[i][j]);
189                             }
190                             newargs[j] = NULL;
191                             execvp(newargs[0], newargs);
192                         }
193                     }
194                 }
195                 exit(0);
196             }
197             exit(0);
198         }
198         }
199         else {
200             // 如果background==0, 父进程将等待子进程结束
201             // 否则将回到函数setup()中等待新命令输入
202             if (background == 0) {
203                 wait(NULL);
204             }
205             else {
206                 setup(inputBuffer, args, &background);
207             }
208         }
209     }
210 }
```

编译运行可得到下面的结果：

```
linjiafengyang@ubuntu:~/Desktop/操作系统实验一/实现shell$ ./myShell
COMMAND->
linjiafengyang@ubuntu:~/Desktop/操作系统实验一/实现shell$ ./myShell
COMMAND->ls
myShell  myShell.cpp
```

命令ls -l:七个字段分别为：
文件属性 文件inode数量 所有者 所属用户组 文件大小 修改时间 文件名

```
COMMAND->ls -l
总用量 32
-rwxrwxr-x 1 linjiafengyang linjiafengyang 22424 4月  14 14:21 myShell
-rwxrw-rw- 1 linjiafengyang linjiafengyang  5224 4月  14 14:29 myShell.cpp
```

命令ls -L -R:列出某文件夹下的所有文件和目录的详细信息

```
COMMAND->ls -L -R /home/linjiafengyang/Desktop/操作系统实验一
/home/linjiafengyang/Desktop/操作系统实验一:
进程的创建实验  进程间共享内存实验  实现shell  信号处理实验

/home/linjiafengyang/Desktop/操作系统实验一/进程的创建实验:
createProcess1      createProcess2      createProcess3      createP
createProcess1.cpp  createProcess2.cpp  createProcess3.cpp  createP

/home/linjiafengyang/Desktop/操作系统实验一/进程间共享内存实验:
shareMemory  shareMemory.cpp

/home/linjiafengyang/Desktop/操作系统实验一/实现shell:
myShell  myShell.cpp

/home/linjiafengyang/Desktop/操作系统实验一/信号处理实验:
signal1  signal1.cpp  signal2  signal2.cpp
```

创建一个新目录并删除(这里先前建了个helloworld.cpp):

```
COMMAND->mkdir newDirectory
COMMAND->ls
helloworld.cpp  myShell  myShell.cpp  newDirectory
COMMAND->rm newDirectory
rm: 无法删除'newDirectory': 是一个目录
COMMAND->rmdir newDirectory
COMMAND->ls
helloworld.cpp  myShell  myShell.cpp
```

编译运行helloworld.cpp然后运行可执行文件helloworld:

```
COMMAND->g++ helloworld.cpp -o helloworld
COMMAND->./helloworld
hello world
```



按下ctrl c输出命令的历史记录:

```
COMMAND->^C
Caught Control C
The command history is:
ls
ls -l
ls -L -R /home/linjiafengyang/Desktop/操作系统实验一
mkdir newDirectory
ls
rm newDirectory
rmdir newDirectory
ls
g++ helloworld.cpp -o helloworld
./helloworld
```



输入命令，后面紧接着&，可以使命令在后台运行(光标停止在下一行，不会出现COMMAND->):

```
COMMAND->ls&
helloworld  helloworld.cpp  myShell  myShell.cpp
```



最后按下ctrl d退出程序。

---

## 五、 实验总结

1. 第一个实验是关于创建进程的实验，刚开始接触进程，对进程之间的相互竞争非常陌生，尤其看到同一个程序出现了很多种结果感觉很不可思议，最后仔细分析出现每一种结果的原因初步认识了进程间的相互竞争。
2. 第二个实验是关于信号处理的实验。根据老师的指导代码，可编写出正确的程序，进一步理解了进程之间的竞争以及程序处理信号的机制。
3. 第三个实验是关于进程间共享内存的实验，初步认识进程间如何通过内存共享进行通信。本身斐波那契数列是常见的算法，但是在此题中我们需要思考数据的处理，怎么让一个进程的数据段分享给另一个进程，让另一个进程使用相同的数据并返回给原进程。通过老师的指导课件，我们知道shmget用来创建或

打开共享存储区，shmat用来连接共享存储区，shmdt用来拆除共享存储区连接，shmctl用来共享存储区控制，所以可通过这些函数实现共享内存。

4. 第四个是实现一个简单shell的实验。这个比较有难度，还好老师提供的指导代码很多，我们只要实现ctrl c输出历史记录中的命令以及其他一些基本东西。 主要的困难在于怎么存储这些记录，这里费了好多功夫理解，最后采用了一个二维数组来存储。具体代码可见上面第四部分的图或者代码文件夹中。

通过这次实验，我加深了对进程的理解，深刻理解进程之间的竞争与通信！