实　验　报　告

学　　号	2103003 1009	姓　　名	惠欣宇	专业班级	2021 计算机 4 班
课程名称	**操作系统**			学期	**2023 年秋季学期**
任课教师	窦金凤	完成日期	2023.12.15	上机课时间	周五 3-6
实 验 名 称	**Linux 进程控制——fork**				

一、实验目的及要求

1.了解进程与程序的区别，加深对进程概念的理解；

2.进一步认识进程并发执行的原理，理解进程并发执行的特点，区别进程并发执行与顺序执行；

3.分析进程争用临界资源的现象，学习解决进程互斥的方法。

4.了解 fork()系统调用的返回值，掌握用 fork()创建进程的方法；

5.熟悉 wait、exit 等系统调用。

二、实验内容

内容一：

1．编写一 C 语言程序（程序名为 fork.c），使用系统调用 fork()创建两个子进程。当程序运行时，系统中有一个父进程和两个子进程在并发执行。父亲进程执行时屏幕显示“I am father”，儿子进程执行时屏幕显示“I am son”，女儿进程执行时屏幕显示“I am daughter”。

2．多次连续反复运行这个程序，观察屏幕显示结果的顺序，直至出现不一样的情况为止。记下这种情况，试简单分析其原因。

内容二：

1．编写一 C 语言程序（程序名为 fork.c），使用系统调用 fork( )创建一个子进程，然后在子进程中再创建子子进程。当程序运行时，系统中有一个父进程、一个子进程和一个子子进程在并发执行。父亲进程执行时屏幕显示“I am father”，儿子进程执行时屏幕显示“I am son”，孙子进程执行时屏幕显示“I am grandson”。

2．多次连续反复运行这个程序，观察屏幕显示结果的顺序，直至出现不一样的情况为止。记下这种情况，试简单分析其原因。

三、实验源码

内容一：

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    int pid1, pid2;  //记录fork出的两个子进程的PID
    if ((pid1 = fork()) < 0) {  //创建子进程son且创建失败
        printf("Son fail create!\n");  //打印创建失败信息
        return 0;
    } else if (pid1 == 0) {  //son创建成功
        printf("I am son. I was created by the father process.\n");  //打印创建成功信息
        return 0;
    } else {
        if ((pid2 = fork()) < 0) {  //创建子进程daughter且创建失败
            printf("Daughter fail create!\n");  //打印创建失败信息
            return 0;
        } else if (pid2 == 0) {  //daughter创建成功
            //打印创建成功信息
            printf("I am daughter. I was created by the father process.\n");
            return 0;
        }
        //父进程的输出，表明自己要创建两个子进程son和daughter
        printf("I am father. I will create two Child processes: son and daughter\n");
        usleep(1000);  //让父进程睡眠1毫秒，使得两个子进程能够执行完成
    }
    return 0;
}
```

内容二：

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    int pid1, pid2;  //pid1用来记录son的PID，pid2用来记录grandson的PID
    if ((pid1 = fork()) < 0) {
        printf("Son fail create!\n");
        return 0;
    } else if (pid1 == 0) {
        if ((pid2 = fork()) < 0) {  //这里由子进程（son）创建孙子进程（grandson）
            printf("Grandson fail create!\n");
            return 0;
        } else if (pid2 == 0) {
            printf("I am grandson. I was create by the son proccess.\n");
            return 0;
        }
        printf("I am son. I was create by the father proccess.\n");
    } else {
        printf("I am father. I will create son proccess.\n");
    }
    usleep(1000);  //与内容一的方法相同，防止子进程的输出结果在root之后
    return 0;
}

```

四、实验结果

内容一：

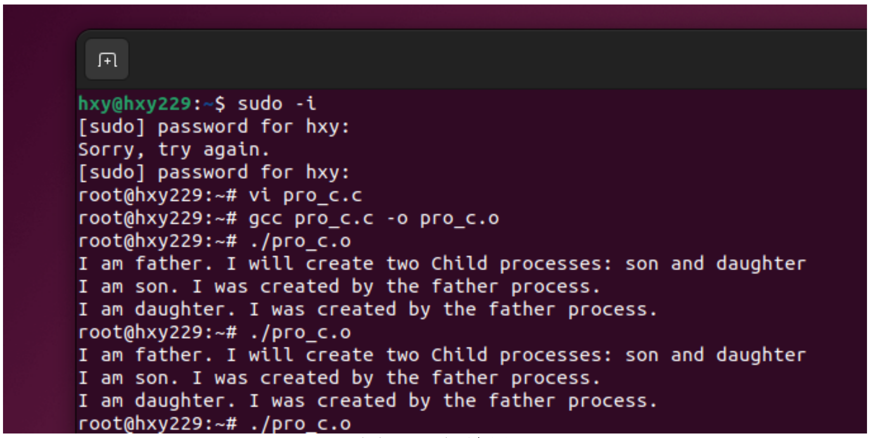

图 1 运行结果

图 2 多次运行发现乱序现象

内容二：
运行结果：

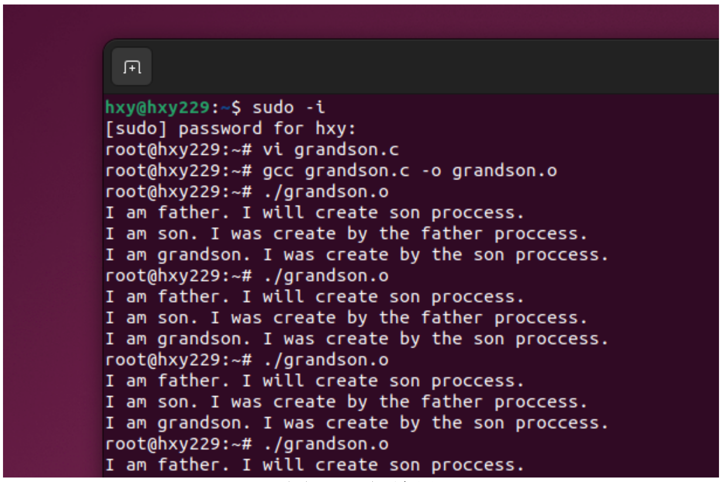

图 3 运行结果

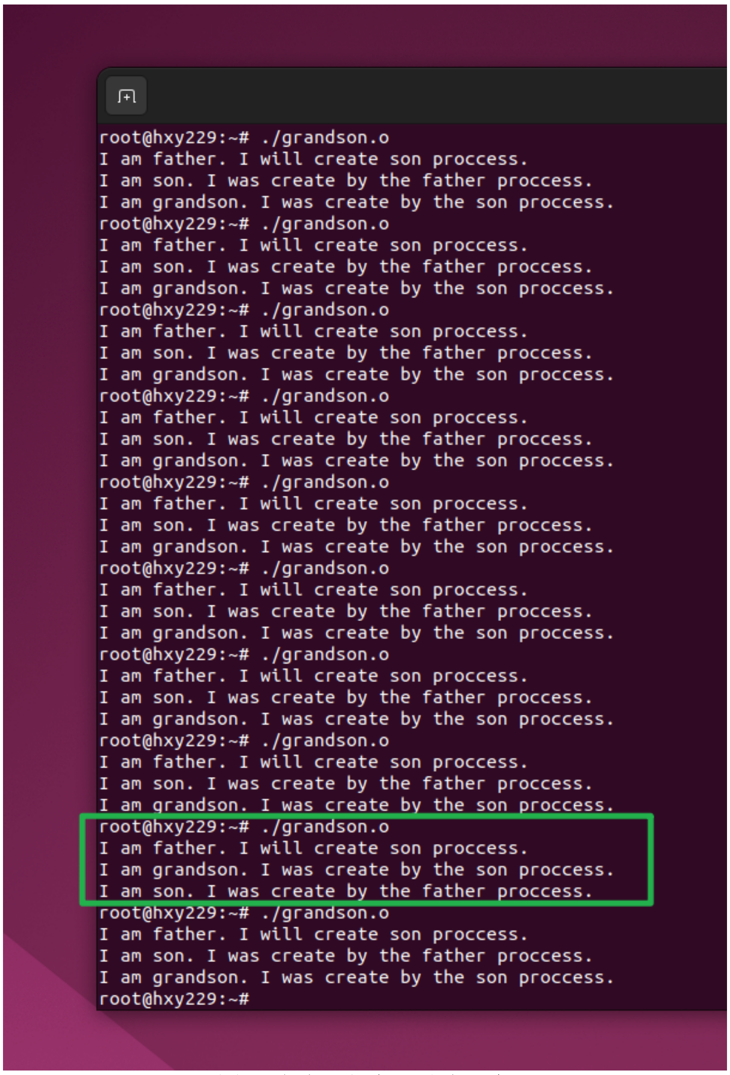

图 4 多次运行发现乱序现象

五、结果分析

这两个小实验中，使用系统调用 fork()创建子进程，并体会进程并发带来的不可再现性。首先我们要清楚：在每一个 fork 语句之前，只有一个进程（父进程）在执行这段代码，但在这条语句之后，就变成两个进程（父与子）在执行了，这两个进程的代码部分完全相同，将要执行的下一条语句都是相同的。

内容一：

在内容一中，我们由父进程执行系统调用 fork()，创建两个子进程。这里我们先讨论创建子进程 son，父子进程的区别除了进程标志符 PID 不同外，变量 pid1 的值也不相同，pid1 存放的是 fork 的返回值。fork 调用的一个奇特的地方在于就是它仅仅被调用一次，却能够返回两次。在父进程中，fork 返回新创建子进程的进程 PID；在子进程中，fork 返回 0；如果出现错误，fork 返回一个负值。这个值在父子进程中都由 pid1 来接收。

因此，在父进程中，pid1 为子进程 son 的 PID，一定大于 0，则父进程转头去执行 pid1 大于 0 的部分（即继续创建子进程 daughter）。而在子进程 son 中，由于创建成功，则子进程 son 的 pid1 为 0，执行判断条件 pid1==0 部分的代码，输出“I am son. I was created by the father process.”，然后执行 return 退出子进程。

父进程在子进程执行判断条件 pid1==0 部分的代码的同时继续执行创建下一个子进程 daughter 的工作，创建子进程 daughter 的过程与上述子进程 son 的过程相同。

内容二：

在内容二中，我们由父进程执行系统调用 fork()，创建一个子进程，再由子进程执行系统调用 fork()创建孙子进程。我们由当然从题目要求中我们可以看出这里我们需要将孙子进程的创建放在子进程中。在父进程执行了 fork 之后，此时存在两个进程（父进程与子进程），父进程中的 pid1 接收第一个子进程 son 的 PID，父进程中的 pid1 显然大于 0，因此转头去执行自己的打印信息。

子进程如果创建成功，则子进程的 pid1 便为 0，因此执行判断条件“pid1==0”部分的代码。这部分代码中首先执行了子进程对孙子进程的创建，子进程的 pid2 为孙子进程的 PID，子进程中的 pid2 显然大于 0，因此转头去执行自己的打印信息，然后执行 return 退出子进程。

孙子进程如果创建成功，则孙子进程的 pid2 便为 0，因此执行判断条件“pid2==0”这部分的代码，打印完自己的信息后，然后执行 return 退出孙子进程。

体会进程并发的不可再现性：

由于使用系统调用 fork()创建子进程，因此不论是内容一的由父进程创建子进程 son 和女儿进程 daughter；还是内容二的由父进程创建子进程 son，再由子进程 son 创建孙子进程 grandson，它们最终都是有 3 个进程需要执行，而这 3 个进程是并发执行的，它们都需要去执行打印操作（即抢占显示器资源），但是显示器资源只有 1 个，这三个进程都需要去抢占显示器资源，哪个进程抢占到了显示器资源，就执行哪个进程的打印操作。

在内容一中，多次执行的抢占顺序大多都为 father、son、daughter，都是父进程先抢占到，然后是儿子进程抢占到，最后是女儿进程抢占到。但是我们在图 2 中可以发现，会有执行结果出现女儿进程比儿子进程先抢占到显示器资源，甚至会有儿子进程比父进程还早地抢占到显示器资源。多次执行的结果并不会有相同顺序的输出结果。这种多次执行结果的不可再现性是并发执行的一个特点。

遇到的问题：

问题 1：

在进行多次执行时，会发现总有几次的执行结果会像图 5 框中的现象一样，输出会出现在 root 之后，经过分析发现有可能是因为父进程执行太快，导致女儿进程还没来得及执行，父进程就结束了，父进程结束后，处理机又去执行女儿进程，就造成了下面这种情况。

我的解决方法便是在父进程即将执行完时，使用 usleep(1000)，让父进程休眠 1 毫秒，使得子进程有时间能够执行完毕，图 5 的问题解决。

```
root@hxy229:~# ./pro_c.o
I am father. I will create two Child processes: son and daughter
I am son. I was created by the father process.
I am daughter. I was created by the father process.
root@hxy229:~# ./pro_c.o
I am father. I will create two Child processes: son and daughter
I am son. I was created by the father process.
root@hxy229:~# I am daughter. I was created by the father process.
./pro_c.o
I am father. I will create two Child processes: son and daughter
I am son. I was created by the father process.
I am daughter. I was created by the father process.
root@hxy229:~#
```

图 5 女儿进程的输出在 root 之后

问题 2：

后续我自行编写了一段代码，代码如图 6。这段代码需要做的是让父进程创建一个子进程，然后两个进程都反复间隔一秒循环打印自己的信息。同样的，我们也可以在这两个进程的并发执行结果中体会到进程并发的不可再现性，如图 7。

root@hxy229: ~

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    printf("I am father. My pid is %d\n",getpid());
    pid_t id = fork();
    if(id == 0) {
        while(1) {
            //printf("My father pid is %d", getppid());
            printf("$$$$$$I am son. My pid is %d. Downloading......\n",getpid());
            sleep(1);
        }
    } else if(id > 0) {
        while(1) {
            printf("@@@@@@I am father. My pid is %d. Printing a file......\n",getpid());
            sleep(1);
        }
    }
    return 0;
}
~
```

图 6 自编代码

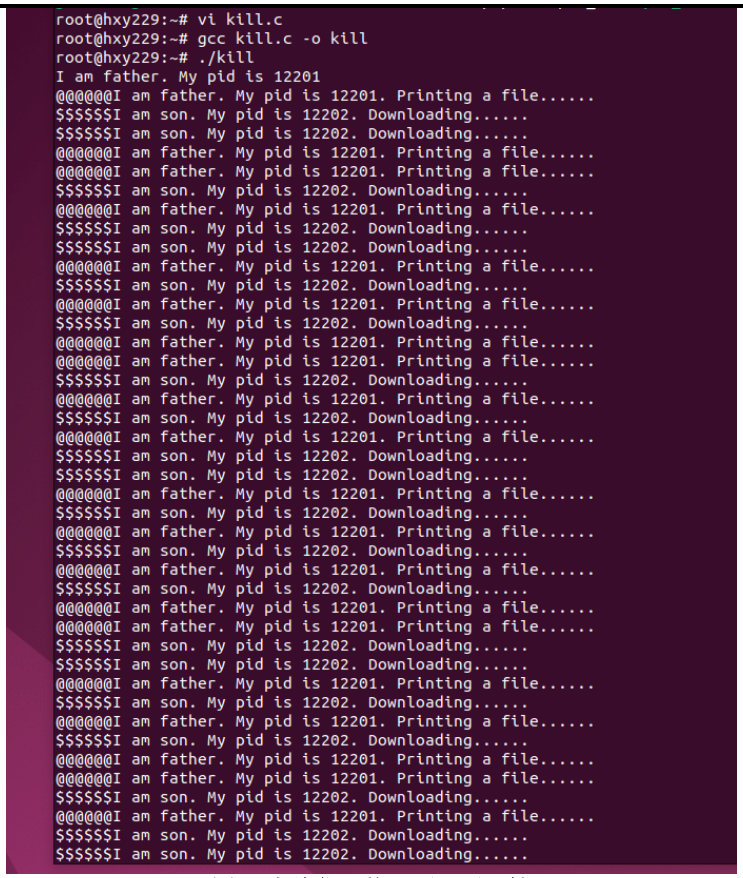

图 7 自编代码体现不可再现性

代码中给出了父进程的 PID(12201)子进程的 PID(12202)，在这两个进程的执行过程中，我使用命令“kill -9 12201”主动杀死了父进程（图 8），但是父进程被杀死之后子进程还在执行，一般来说父母被杀死，也会杀死孩子，但是孩子在父亲被杀死之后依然存在。

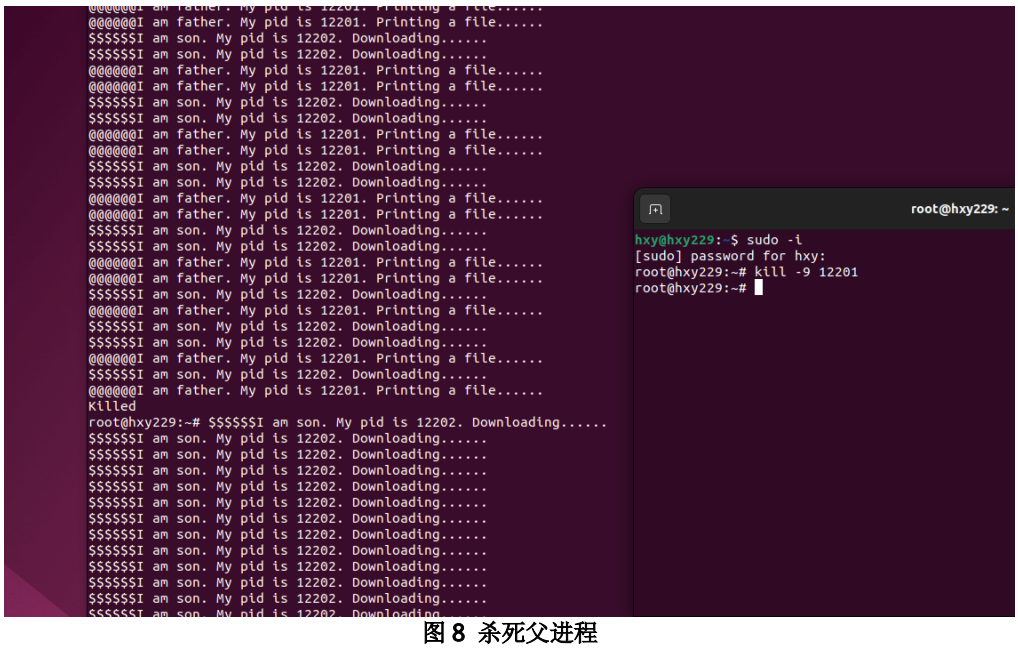

图 8 杀死父进程

通过分析，得到的结论是：在 kill 掉父进程后，父进程被移除了进程表，但是子进程还存在于进程表中，子进程的 PID 是由原先的父进程在 fork 时为其创建的，且是独立的，因此后续子进程打印自己的 PID 还是 12202。但是现在的子进程是以何种身份存在的呢？

我们观察图 6 中被注释掉的那一行代码，这行代码是在子进程中打印父进程的 PID，我们使用的是 getppid()函数，这个函数能够返回该子进程的父进程的 PID。我们将注释的输出信息加入子进程的输出中，如图 9。

root@hxy229: ~

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

int main() {
    printf("I am father. My pid is %d\n",getpid());
    pid_t id = fork();
    if(id == 0) {
        while(1) {
            printf("$$$$$$I am son. My pid is %d. My father's pid is %d.Downloading......\n",getpid(), getppid());
            sleep(1);
        }
    } else if(id > 0) {
        while(1) {
            printf("@@@@@@I am father. My pid is %d. Printing a file......\n",getpid());
            sleep(1);
        }
    }
    return 0;
}
```

图 9 修改子进程输出获得父进程 PID

再次编译运行一下这段代码。发现此时的子进程的输出结果中父进程的 PID 与父进程自己的输出结果的 PID 是相同的都是 14524，子进程的 PID 为 14525，如图 10。

root@hxy229: ~

```
hxy@hxy229:~$ sudo -i
[sudo] password for hxy:
root@hxy229:~# vi kill.c
root@hxy229:~# gcc kill.c -o kill
root@hxy229:~# ./kill
I am father. My pid is 14524
@@@@@@I am father. My pid is 14524. Printing a file......
$$$$$$I am son. My pid is 14525.My father's pid is 14524. Downloading......
@@@@@@I am father. My pid is 14524. Printing a file......
$$$$$$I am son. My pid is 14525.My father's pid is 14524. Downloading......
@@@@@@I am father. My pid is 14524. Printing a file......
$$$$$$I am son. My pid is 14525.My father's pid is 14524. Downloading......
@@@@@@I am father. My pid is 14524. Printing a file......
$$$$$$I am son. My pid is 14525.My father's pid is 14524. Downloading......
$$$$$$I am son. My pid is 14525.My father's pid is 14524. Downloading......
@@@@@@I am father. My pid is 14524. Printing a file......
@@@@@@I am father. My pid is 14524. Printing a file......
$$$$$$I am son. My pid is 14525.My father's pid is 14524. Downloading......
@@@@@@I am father. My pid is 14524. Printing a file......
$$$$$$I am son. My pid is 14525.My father's pid is 14524. Downloading......
@@@@@@I am father. My pid is 14524. Printing a file......
$$$$$$I am son. My pid is 14525.My father's pid is 14524. Downloading......
$$$$$$I am son. My pid is 14525.My father's pid is 14524. Downloading......
```

图 10 kill.c 修改之后的输出结果

在此结果的基础上，我们再将父进程杀死，观察杀死父进程之后，子进程中父进程 PID 的输出结果，如图 11。

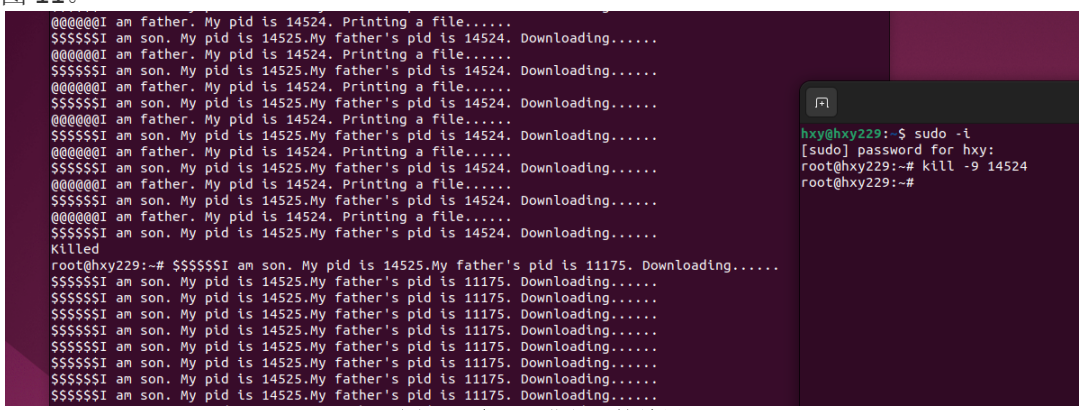

图 11 杀死父进程后的结果

通过观察图 11 我们发现在杀死父进程之后，子进程对父进程 PID 的输出不再是原先的 14524，而变为了 11175，这说明这个子进程所属的父进程已经不再是原先那个父进程了，那个父进程已经被杀死而从进程表中删除了，此时这个子进程所属的进程就变为了 init 进程，也就是说这个子进程被 init 进程接管了，init 进程是一个特殊的进程，是 Linux 系统中所有进程的起点。它是系统引导过程中由内核启动的第一个用户级进程。init 进程的 PID 始终为 1，是所有其他进程的祖先进程。因此这里的父进程输出应当为 1 才是正确的结果，但这里输出的父进程的 PID 为 11175。经过讨论后认为出现这种情况的可能是这个父进程的 pid 可能为虚拟机在本机中的 PID，因此不为 1。