操作系统实验

实验 1 Linux进程控制

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。

二、实验内容

内容一

> 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”。
>
> 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

> 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”。
>
> 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

内容一源码：

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

int main()
```

```
{
  int pid1,pid2;
  printf("I am father!\n");
  if ((pid1 = fork())<0)
  {
     printf("Child1 fail create!\n");
     return 1;
  }
  else if (pid1 == 0)
  {
     printf("I am son!\n");
     return;
  }
  if ((pid2 = fork())<0)
  {
     printf("Child2 fail create!\n");
     return;
  }
  else if (pid2 == 0)
  {
     printf("I am daughter!\n");
     return;
  }
}

```

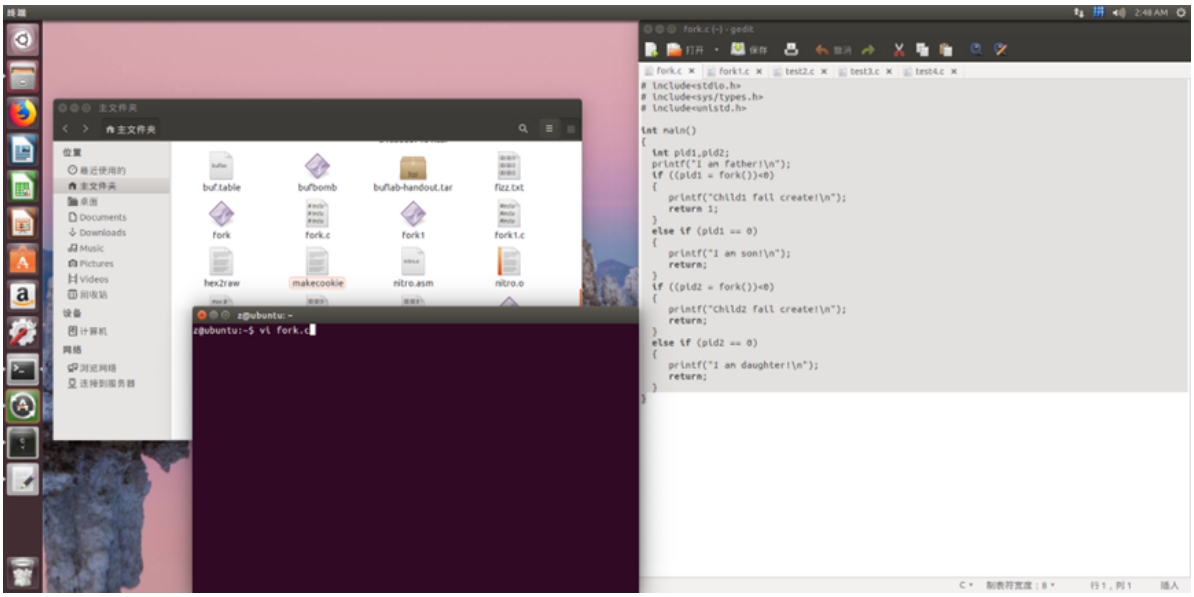
内容二源码：

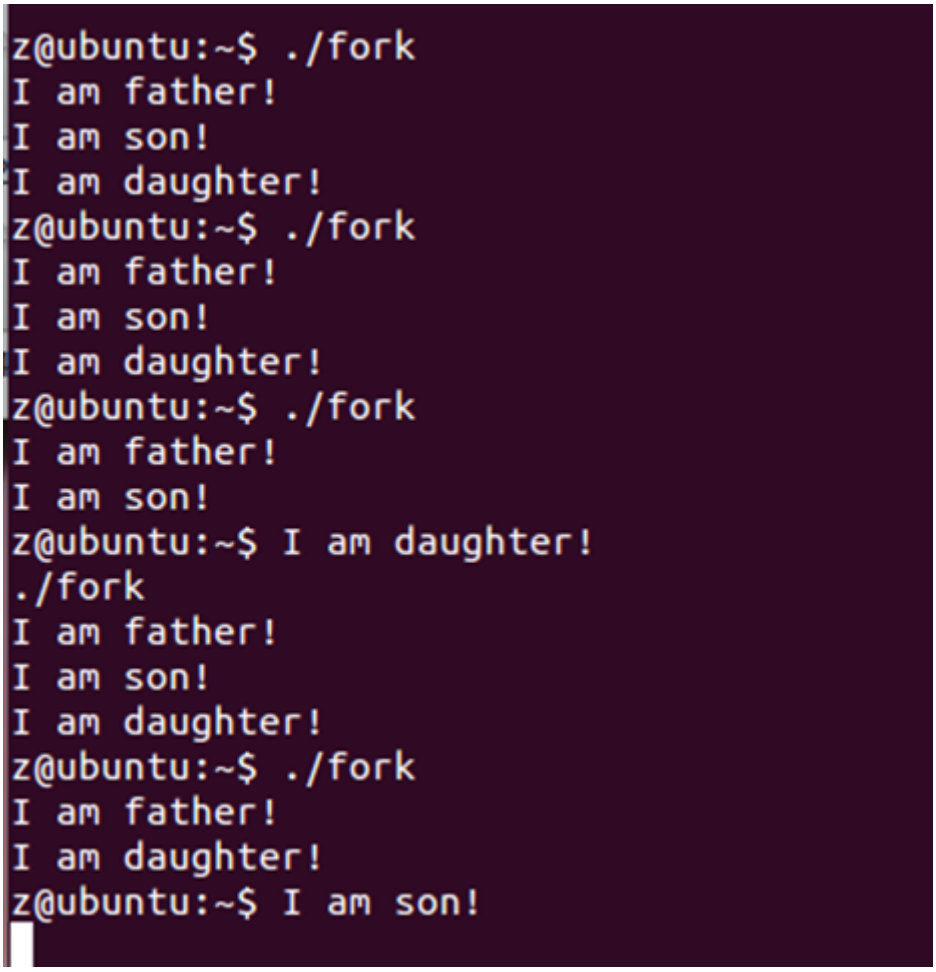
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    int pid1, pid2;
    printf("I am father!\n");

    // 创建子进程
    if ((pid1 = fork()) < 0) {
        printf("Child1 failed to create!\n");
        return 1;
    } else if (pid1 == 0) {
        // 子进程代码

        // 创建孙子进程
        if ((pid2 = fork()) < 0) {
            printf("Grandchild failed to create!\n");
            return 1;
        } else if (pid2 == 0) {
            printf("I am grandson!\n");
            return;
        } else {
            printf("I am son!\n");
            return;
        }
    }
```

```
28 }
```

四、实验结果

内容一结果：





内容二结果：

z@ubuntu: ~

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

int main() {
    int pid1, pid2;
    printf("I am father!\n");

    // 创建子进程
    if ((pid1 = fork()) < 0) {
        printf("Child1 failed to create!\n");
        return 1;
    } else if (pid1 == 0) {
        // 子进程代码

        // 创建孙子进程
        if ((pid2 = fork()) < 0) {
            printf("Grandchild failed to create!\n");
            return 1;
        } else if (pid2 == 0) {
            printf("I am grandson!\n");
            return;
        } else {
            printf("I am son!\n");
            return;
        }
    }
}
~
~
"fork1.c" 28L, 624C                                          1,1
```

```
z@ubuntu:~$ vi fork1.c
z@ubuntu:~$ gcc -o fork1 fork1.c
z@ubuntu:~$
```

```
./fork1
I am father!
z@ubuntu:~$ I am son!
I am grandson!
./fork1
I am father!
I am grandson!
z@ubuntu:~$ I am son!
```

五、结果分析

内容一分析：

发现它会有两种不同的输出结果

一种情况是因为父进程首先创建了Child1进程，然后继续创建Child2进程，所以两个子进程会按顺序依次输出。

另一种情况是因为父进程首先创建了Child1进程，然后创建Child2进程，但操作系统可能选择在Child2进程先执行，所以两个子进程的输出顺序出现了变化。

无论是哪种输出结果，父进程和两个进程之间的执行是并行的，它们互相独立地执行各自的代码段。

内容二分析：

发现它也会有两种不同的输出结果

一种情况：父进程首先创建了Child1进程，然后创建Child2进程。但由于操作系统在调度进程时具有随机性，无法确定哪个子进程会先执行。因此，两个子进程的输出顺序可能会发生变化。

另一种情况：父进程首先创建了Child1进程，然后创建Child2进程。根据操作系统的调度策略，可能会选择先执行Child1进程，然后再执行Child2进程，所以输出结果的顺序与第一个示例不同。