201311211914_赵帅帅_实验 3

1、 修改实验二的程序，将每个进程输出一个字符改为每个进程输出一句话，观察分析显示结果。

代码 lab31.c：

```
#include<stdio.h>
int main(void)
{
 int  p1, p2;
 p1 = fork();
 if(!p1)
 {
     printf("p1 is working\n");
 }
 else
 {
     p2 = fork();
     if(!p2)
     {
         printf("p2 is working\n");
     }
     else
     {
         printf("father is working\n");
     }
 }
 return 0;
}
```

运行结果：

```
p1 is working
p2 is working
father is working
```

结果分析：

当字符串的长度较短时，输出字符串的过程中不会出现进程切换，故而运行结果如上。我们可以推断，当输出字符串很长时会出现交替输出的现象。

2、 如果在父进程 fork 之前，输出一句话，这句话后面不加“\n”或加“\n”，结果有什么不同，为什么？

代码 lab32.c：

```
#include<stdio.h>
int main(void)
{
 int  p1, p2;
 printf("before p1 fork\t");
 p1 = fork();
 if(!p1)
 {

 }
 else
 {
     printf("before p2 fork\t");
     p2 = fork();
     if(!p2)
     {

     }
     else
     {

     }
 }
 return 0;
}
```

运行结果：

before p1 fork before p1 fork before p2 fork before p1 fork before p2 fork

代码 lab33.c：

```
#include<stdio.h>
int main(void)
{
 int  p1, p2;
 printf("before p1 fork\n");
 p1 = fork();
 if(!p1)
 {

 }
```

```
    else
    {
        printf("before p2 fork\n");
        p2 = fork();
        if(!p2)
        {

        }
        else
        {

        }
    }
    return 0;
}
```

运行结果：

```
before p1 fork
before p2 fork
```

结果分析：

不加\n 的情况下，使用 printf 时只是将要打印的字符串放在缓冲区里，除非缓冲区满或者有其他强制性命令时，缓冲区中的内容才被打印到屏幕上。因此，子进程会在其生成时继承父进程缓冲区中的内容，所以才会出现打印多次的现象。

加\n 的情况下,系统会强制把缓冲区的内容输出到屏幕上，就不会出现因继承而导致的多次输出了。

3、 如果在程序中使用系统调用 lockf 来给每一个进程加锁，可以实现进程之间的互斥。将 lockf 加在输出语句前后运行试试；将一条输出语句变成多条输出语句，将 lockf 语句放在循环语句外部或内部试试，观察分析显示结果。

代码 lab34.c:

```
#include<stdio.h>
int main(void)
{
int   i = 0, p1, p2;
p1 = fork();
if(!p1)
{
    for(i = 0; i < 10; i ++)
    {
        printf("p1 is working %d\n", i);
    }
```

```
    }
    else
    {
        printf("before p2 fork\n");
        p2 = fork();
        int j = 0;
        if(!p2)
        {
            for(j = 0.; j < 10; j ++)
            {
                printf("p2 is working %d\n", j );
            }
        }
        else
        {
            int k = 0;
            for(k = 0; k < 10; k ++)
            {
                printf("father is working %d", k);
            }
        }
    }
    return 0;
}
```

运行结果：

```
p1 is working 0
p1 is working 1
p1 is working 2
p1 is working 3
p1 is working 4
p1 is working 5
p1 is working 6
p1 is working 7
p1 is working 8
p1 is working 9
p2 is working 0
p2 is working 1
p2 is working 2
p2 is working 3
p2 is working 4
p2 is working 5
p2 is working 6
p2 is working 7
p2 is working 8
```

```
p2 is working 9
father is working 0
father is working 1
father is working 2
father is working 3
father is working 4
father is working 5
father is working 6
father is working 7
father is working 8
father is working 9
```

代码 lab35.c:

```
#include<stdio.h>
int main(void)
{
    int i = 0, j = 0, k = 0, p1, p2;
    p1 = fork();
    if(!p1)
    {
        for(i = 0; i < 10; i ++)
        {
            lockf(1, 1, 0);
            printf("p1 is working %d\n", i);
            lockf(1, 0, 0);
            sleep(1);
        }
    }
    else
    {
        p2 = fork();
        if(!p2)
        {
            for(j = 0.; j < 10; j ++)
            {
                lockf(1, 1, 0);
                printf("p2 is working %d\n", j );
                lockf(1, 0, 0);
                sleep(1);
            }
        }
        else
        {
            for(k = 0; k < 10; k ++)
            {
```

```
                lockf(1, 1, 0);
                printf("father is working %d\n", k);
                lockf(1, 0, 0);
                sleep(1);
            }
        }
    }
    return 0;
}
```

运行结果：

```
p1 is working 0
p2 is working 0
father is working 0
p1 is working 1
p2 is working 1
father is working 1
p1 is working 2
p2 is working 2
father is working 2
p1 is working 3
p2 is working 3
father is working 3
p1 is working 4
p2 is working 4
father is working 4
p1 is working 5
p2 is working 5
father is working 5
p1 is working 6
p2 is working 6
father is working 6
p1 is working 7
p2 is working 7
father is working 7
p1 is working 8
p2 is working 8
father is working 8
p1 is working 9
p2 is working 9
father is working 9
```

代码 lab36.c:

```
#include<stdio.h>
int main(void)
```

```
{
int i = 0, j = 0, k = 0, p1, p2;
p1 = fork();
if(!p1)
{
    lockf(1, 1, 0);
    for(i = 0; i < 10; i ++)
    {
        printf("p1 is working %d\n", i);
        sleep(1);
    }
    lockf(1, 0, 0);
}
else
{
    p2 = fork();
    if(!p2)
    {
        lockf(1, 1, 0);
        for(j = 0.; j < 10; j ++)
        {
            printf("p2 is working %d\n", j );
            sleep(1);
        }
        lockf(1, 0, 0);
    }
    else
    {
        lockf(1, 1, 0);
        for(k = 0; k < 10; k ++)
        {
            printf("father is working %d\n", k);
            sleep(1);
        }
        lockf(1, 0, 0);
    }
}
return 0;
}
```

运行结果：

```
p1 is working 0
p1 is working 1
p1 is working 2
p1 is working 3
```

```
p1 is working 4
p1 is working 5
p1 is working 6
p1 is working 7
p1 is working 8
p1 is working 9
p2 is working 0
p2 is working 1
p2 is working 2
p2 is working 3
p2 is working 4
p2 is working 5
p2 is working 6
p2 is working 7
p2 is working 8
p2 is working 9
father is working 0
father is working 1
father is working 2
father is working 3
father is working 4
father is working 5
father is working 6
father is working 7
father is working 8
father is working 9
```

结果解释：

lockf(1, 1, 0)表示对屏幕的锁定，只允许当前进程在屏幕上输出；lockf(1, 0, 0)表示解锁屏幕，其他进程也可以在屏幕上输出。

首先，我们让进程自然输出，由于输出字符串较短，故而一个进程的所有输出一次便可以完成；而我们在进程输出前后加锁，并让进程休眠 1 秒，使得进程有时间来回切换，便得到了输出交替进行的现象；而当我们把进程休眠 1 秒放在了锁的内部时，虽然切换到了其他进程，但是屏幕被当前进程锁定，其他程序无法输出到屏幕上，所以后面再次出现了顺序输出的现象。

4、 以上各种情况都多运行几次，观察每次运行结果是否都一致？为什么？

有时不一致。因为操作系统的进程切换顺序是不可预知的。