云南大学数学与统计学院
上机实践报告

课程名称：操作系统实验	**年级**：2015 级	**上机实践成绩**：
指导教师：李源	**姓名**：刘鹏	
上机实践名称：Linux 下的进程管理	**学号**：20151910042	**上机实践日期**：2017-10-25
上机实践编号：No.02	**创建时间**：	**最近一次修改时间**：00:11

一、实验目的

1. 掌握进程的概念，明确进程的含义；
2. 认识并了解并发执行的实质。

二、实验内容

1. 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`'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。

2. 修改上述程序，每一个进程循环显示一句话。子进程显示`'daughter …'`及`'son ……'`，父进程显示 `'parent ……'`，观察结果，分析原因。

三、实验平台

1. 硬件平台

CPU： Intel Core i3 550
RAM： Kinston DDR3 1333MHz 6GB
主硬盘： Toshiba SATA SSD 128GB

2. 软件平台

CentOS 7 GNOME
Windows 10 Enterprise 64bit 1709
X shell 5 (Build 1332)
Microsoft Word 2016

四、实验指导

1. 进程

Linux 中，进程既是一个独立拥有资源的基本单位，又是一个独立调度的基本单位。一个进程实体由若干个区（段）组成，包括**程序区**、**数据区**、**栈区**、**共享存储区**等。每个区又分为若干页，每个进程配置有唯一的进程控制块 ***PCB***，用于控制和管理进程。

PCB 的数据结构如下：

(1) 进程表项（Process Table Entry）

包括一些最常用的核心数据：**进程标识符 PID**、**用户标识符 UID**、**进程状态**、**事件描述符**、**进程和 U 区在内存或外存的地址**、**软中断信号**、**计时域**、**进程的大小**、**偏置值 nice**、**指向就绪队列中下一个 PCB 的指针 P_Link**、指向 **U 区进程正文**、**数据及栈在内存区域的指针**。

(2) U 区（U Area）

用于存放进程表项的一些扩充信息。

每一个进程都有一个私用的 U 区，其中含有：**进程表项指针**、**真正用户标识符 u-ruid** (read user ID)、**有效用户标识符 u-euid** (effective user ID)、**用户文件描述符表**、**计时器**、**内部 I/O 参数**、**限制字段**、**差错字段**、**返回值**、**信号处理数组**。

由于 **LINUX** 系统采用**段页式存储管理**，为了把段的起始虚地址变换为段在系统中的物理地址，便于实现区的共享，所以还有：

(3) 系统区表项

以存放各个段在物理存储器中的位置等信息。

系统把一个进程的虚地址空间划分为若干个连续的逻辑区，有正文区、数据区、栈区等。这些区是可被共享和保护的独立实体，多个进程可共享一个区。为了对区进行管理，核心中设置一个系统区表，各表项中记录了以下有关描述活动区的信息：

区的类型和大小、区的状态、区在物理存储器中的位置、引用计数、指向文件索引结点的指针。

(4) 进程区表

系统为每个进程配置了一张进程区表。表中，每一项记录一个区的起始虚地址及指向系统区表中对应的区表项。核心通过查找进程区表和系统区表，便可将区的逻辑地址变换为物理地址。

2. 进程映像

LINUX 系统中，进程是进程映像的执行过程，也就是正在执行的进程实体。它由三部分组成：

1、用户级上、下文。主要成分是用户程序；
2、寄存器上、下文。由 CPU 中的一些寄存器的内容组成，如 PC，PSW，SP 及通用寄存器等；
3、系统级上、下文。包括 OS 为管理进程所用的信息，有静态和动态之分。

3. 所涉及的系统调用

1、fork()

创建一个新进程。

系统调用格式：

```
pid = fork()
```

参数定义：

```
int fork()
```

fork()返回值意义如下：

0：在子进程中，pid 变量保存的 fork()返回值为 0，表示当前进程是子进程。
> 0：在父进程中，pid 变量保存的 fork()返回值为子进程的 id 值（进程唯一标识符）。
-1：创建失败。

如果 fork()调用成功，它向父进程返回子进程的 PID，并向子进程返回 0，即 fork()被调用了一次，但返回了两次。此时 OS 在内存中建立一个新进程，所建的新进程是调用 fork()父进程（*parent process*）的副本，称为子进程（*child process*）。子进程继承了父进程的许多特性，并具有与父进程完全相同的用户级上下文。**父进程与子进程并发执行**。

核心为 fork()完成以下操作：

（1）为新进程分配一进程表项和进程标识符

进入 fork()后，核心检查系统是否有足够的资源来建立一个新进程。若资源不足，则 fork()系统调用失败；否则，核心为新进程分配一进程表项和唯一的进程标识符。

（2）检查同时运行的进程数目

超过预先规定的最大数目时，fork()系统调用失败。

（3）拷贝进程表项中的数据

将父进程的当前目录和所有**已打开的数据**拷贝到子进程表项中，并置进程的状态为**“创建”**状态。

（4）子进程继承父进程的所有文件

对父进程当前目录和所有已打开的文件表项中的**引用计数加 1**。

（5）为子进程创建进程上、下文

进程创建结束，设子进程状态为“内存中**就绪**”并返回子进程的标识符。

（6）子进程执行

虽然父进程与子进程程序**完全相同**，但每个进程都有自己的程序计数器 **PC（注意子进程的 PC 开始位置）**，然后根据 pid 变量保存的 fork()返回值的不同，执行了不同的分支语句。

五、实验内容与步骤

题 1

程序代码：

```
1   // filename: a.c
```

```
2
3   #include<stdio.h>
4
5   int main()
6   {
7       printf("%d-----HEAD BEGIN?\n",getpid());
8       int p1, p2;
9       // the process ID we will need
10      p1 = fork();
11      if (p1 == -1)
12      {
13          printf("Failed Create New Process!");
14          return 0;
15      }
16
17      if (p1 == 0)
18      // if not failed, p1 == ProcessID(in father prcess) or 0(in child process)
19      {
20          printf("%d\tb\n",getpid());
21          // child process
22      }
23      else
24      {
25          // father process
26          p2 = fork();
27          if (p2 == -1)
28          {
29              printf("Failed Create New Process");
30              return 0;
31          }
32
33          if (p2 == 0)
34          {
35              printf("%d\tc\n",getpid());
36          }
37          else
38          {
39              printf("%d\ta\n",getpid());
40          }
41
42
43      printf("\t%d\n",getpid());
44
45      }
46      return 0;
47  }
```

Code Box 1

运行结果：

```
[root@Newton-PC-1 OS_experiment_Report_2]# ./one_Output.exe
7623-----HEAD BEGIN?
7624    b
7623    a
7625    c


[root@Newton-PC-1 OS_experiment_Report_2]# ./one_Output.exe
7627-----HEAD BEGIN?
7628    b
7627    a

7629    c

[root@Newton-PC-1 OS_experiment_Report_2]# ./one_Output.exe
7639-----HEAD BEGIN?
7640    b
7639    a

7641    c

[root@Newton-PC-1 OS_experiment_Report_2]#
```

图 1 程序运行结果

结果分析：

用这个程序来理解并行的 CPU 调用方式的特征。

这个程序的语句比较简单，简单说来就是通过调用 `fork()`函数，进行多进程开启。由教材知识可以知道，多线程并发执行在固有的算法调度下，是具有可再现性的。但是从表面上看，几乎每次的执行结果都不一样，但是输出的顺序却都是一致的。这与我的分析基本一致。看起来这与可再现性有很大的差异。

```
[root@neNewton-PC-1 OS_experiment_Report_2]# ./one_Output.exe
11417-----HEAD BEGIN?
11418   b
11417   a
11419   c


[root@neNewton-PC-1 OS_experiment_Report_2]# ./one_Output.exe
11420-----HEAD BEGIN?
11421   b
11420   a
11422   c


[root@neNewton-PC-1 OS_experiment_Report_2]# ./one_Output.exe
11423-----HEAD BEGIN?
11424   b
11423   a
11425   c


[root@neNewton-PC-1 OS_experiment_Report_2]# ./one_Output.exe
11426-----HEAD BEGIN?
11427   b
11426   a
11428   c


[root@neNewton-PC-1 OS_experiment_Report_2]# ./one_Output.exe
11429-----HEAD BEGIN?
11430   b
11429   a
11431   c


[root@neNewton-PC-1 OS_experiment_Report_2]# ./one_Output.exe
11432-----HEAD BEGIN?
11433   b
11432   a

11434   c

[root@neNewton-PC-1 OS_experiment_Report_2]# ./one_Output.exe
11443-----HEAD BEGIN?
11444   b
11443   a
11445   c


[root@neNewton-PC-1 OS_experiment_Report_2]#
```

图 2 多次实验

进行了多次的实验之后，得出的结果基本一致。所以根据实验数据的统计结果，可以认为输出顺序基本就是如此了。如图 2 所示。

但是如果对程序进行稍微修改，观察一下输出回车时的进程号。

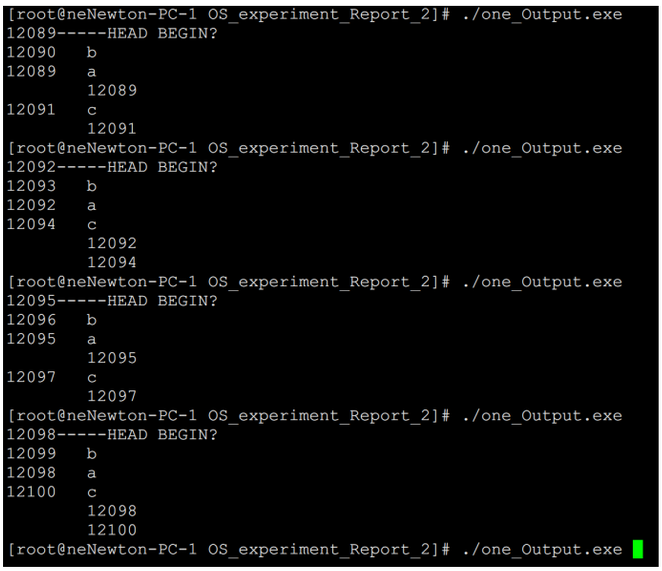

图 3 带进程号的输出

可以发现，同在一个块里面的两个语句，在并发条件下的执行也是有区别的。有时候是两个语句一起执行，有的时候两个语句会分开进行。我猜测，这是因为系统中有其他进程在使用 CPU 资源，所以所有进程在队列中的执行结果，与仅有三个进程是完全不一样的。

不过可以看到，虽然绝对顺序有一点差异，但是相对顺序毫无差错，这几个进程彼此之间毫无差别，所以无论是采取哪种分配策略，得到的最终队列顺序都一样，在排队过程中，无论中间有什么进程插入，彼此之间的相对顺序都是一样的，所以打印的结果也不足为怪，只是绝对不是随机产生！

题 2

程序代码

```c
// filename: b.c

#include<stdio.h>

int main()
{
    int p1, p2, i;
    p1 = fork();
    if(p1 == -1)
    {
        printf("Failed Create New Process!");
        return 0;
    }

    if (p1 == 0)
    {
        for (i = 0; i < 10; i++)
        {
            printf("%d\tdaughter-----%d\n",getpid(), i);
        }
    }
    else
    {
        p2 = fork();
        if (p2 == -1)
        {
            printf("Failed Create New Process!");
            return 0;
        }

        if (p2 == 0)
        {
            for (i = 0; i < 10; i++)
            {
                printf("%d\tson----------%d\n",getpid(), i);
            }
        }
        else
        {
            for (i = 0; i < 10; i++)
            {
                printf("%d\tparent-------%d\n",getpid(), i);
            }
        }
    }
```

```
46      return 0;
47  }
```

Code Box 2

运行结果：

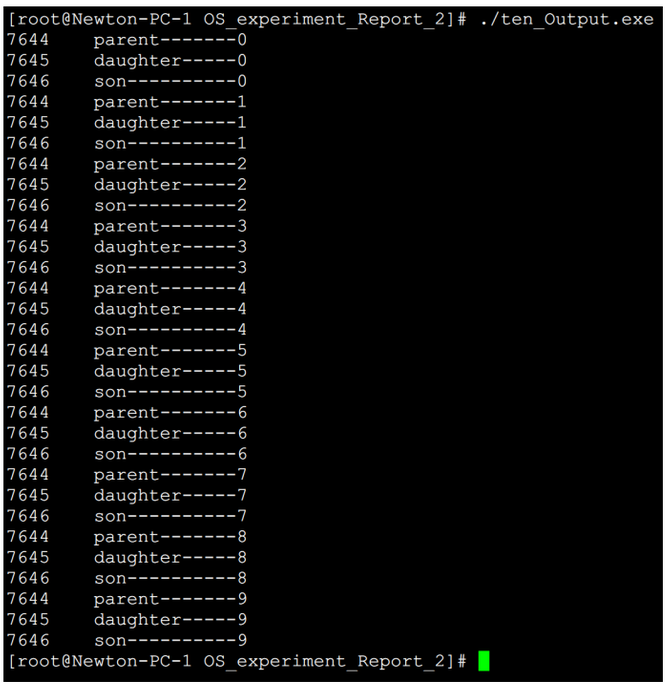

图 4 多进程循环输出

结果分析：

屏幕作为标准输出设备，在 LINUX 内核下，是被认为是文件资源。屏幕不能被认为是临界资源，当 C 语言的代码被翻译成二进制之后，这些二进制的语言应该也是有块的存在，不是毫无规则的一串 0-1 代码。二进制的执行，是按照原子操作进行的。所以一个进程在单独执行的时候，也是分成了好多原子操作，按部就班完成。

我猜测：一个运算符所带来的计算被认为是一个占用处理机的原子操作，其二进制不可再分。如 **`if(a < b = c)`**，被分为了 3 个操作，首先是赋值，其次是比较，最后是判断。可能确定的标准是 CPU 寄存器的使用。当然，一个不带计算的输出语句，被认为是一个原子操作。CPU 可以在多个含有多原子操作的进程之间来回抖动，模拟出一种并行的假象。

虽然图 4 的反应结果非常不错，但是经过多次实验之后发现，并不是每次的执行都很有顺序。如图 5，可以看出，结果非常乱。甚至 daughter 进程早早就结束了。这也充分说明，在一个复杂的系统下，不能简单地认识一些理所当然的东西，之前认为相对顺序不变，在这里也不成立了。可能在处理这种情况时，系统更加复杂了。这里也不好揣测，不过我们知道了，这里必然有某种策略。

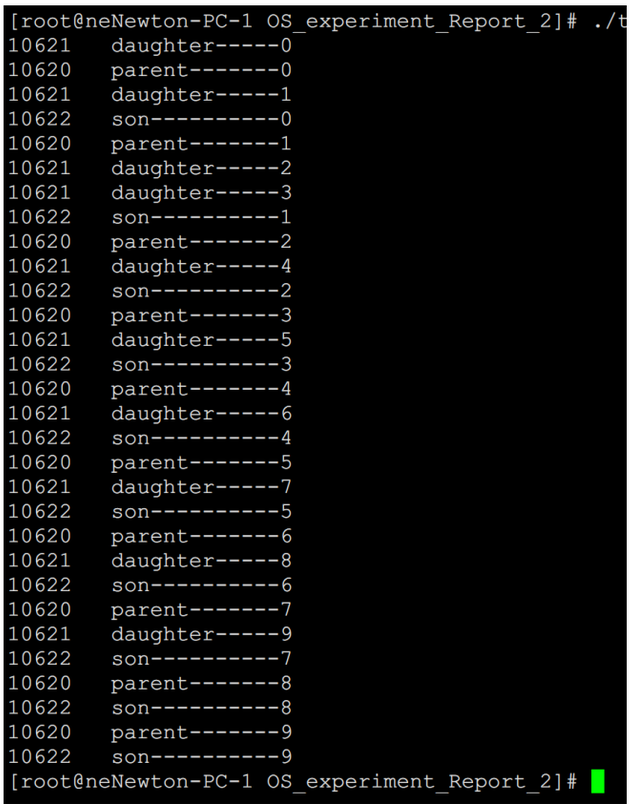

图 5 多次实验之后的结果输出

六、实验总结

对一个操作系统进行类似“黑盒调试”的工作比较累，更何况，LINUX 内核身经百战，早已十分复杂。这里仅是知道了可以调用函数开辟新的进程，可能对以后的多进程编程更加有好处。但是在理解 LINUX 下的进程管理角度看，还是比较失败的。

不过从里一个方面看，LINUX 的进程管理也不是一两句话可以阐述清楚的，等过几个星期的学习后，再来看或许就豁然开朗了。

七、参考文献