南昌大学实验报告

姓名：田宇琛

学号：5601115010

邮箱地址：tyc896@qq.com

专业班级：计科153

实验日期：2018.04.23

课程名称：Linux程序设计实验

实验项目名称

Multi-processing in Linux

实验目的

- Understanding the mechanism of multi-processing
- Understanding the idea of process scheduling
- Understanding multi-thread programming

实验基础

Linux系统的熟练使用，Linux下几个相关的C语言系统相关库

实验步骤

The Fork Question

1. fork_problem

```
forkprob: fork_problem.o
    gcc fork_problem.o -o forkprob

  fork_problem.o: fork_problem.c
    gcc -c fork_problem.c -o fork_problem.o
```

调用了`fork`函数后，就像产生了两个平行的宇宙，他们拥有同样的资源，但是程序走向却可能朝着不同的方向，在子进程中，它修改了`value`这个变量（此时，该变量已经不是两个线程共享了，父进程和子进程各有一个）。于是，父进程中的`value`没有发生变化，而子进程中的`value`被修改了。

```
control@ubuntu:~/Desktop/lab4$ make
gcc -c fork_problem.c -o fork_problem.o
fork_problem.c: In function ‘main’:
fork_problem.c:36:3: warning: implicit declaration of function ‘wait’ [-Wimplici
t-function-declaration]
   wait(NULL);
   ^
gcc fork_problem.o -o forkprob
control@ubuntu:~/Desktop/lab4$ ./forkprob
I am the child - value = 20
Child Completed ....
I am the parent - value = 5
```

这里报了一个`warning`，加一头文件的包含即可：`#include<sys/wait.h>`

```
control@ubuntu:~/Desktop/lab4$ vim fork_problem.c
control@ubuntu:~/Desktop/lab4$ make
gcc -c fork_problem.c -o fork_problem.o
gcc fork_problem.o -o forkprob
control@ubuntu:~/Desktop/lab4$ ./forkprob
I am the child - value = 20
Child Completed ....
I am the parent - value = 5
```

2. multi-fork

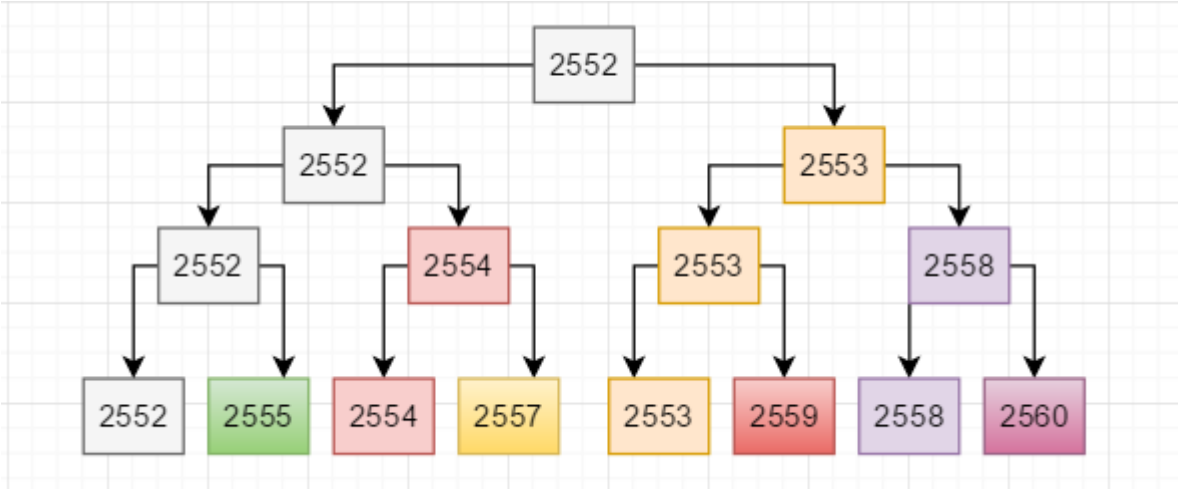
```
muti: muti.o
    gcc muti.o -o muti

  muti.o: multi-fork.c
    gcc -c multi-fork.c -o muti.o
```

```
control@ubuntu:~/Desktop/lab4$ vim multi-fork.c
control@ubuntu:~/Desktop/lab4$ make
gcc -c multi-fork.c -o muti.o
gcc muti.o -o muti
control@ubuntu:~/Desktop/lab4$ ./muti;ps
2552
2552
2552
2552
2555
2554
2554
2553
2553
2553
2557
2559
2558
2558
2560
  PID TTY          TIME CMD
 1957 pts/17   00:00:00 bash
 2556 pts/17   00:00:00 ps
```

少了两个头文件：`sys/types.h和unistd.h`，加上之后，不会有**warning**报错。另外，程序打出了一系列进程的`pid`的值，我在后面用`ps`命令打断了一下，否则输出格式有点bug。这里可以统计一下进程的`pid`个数，

图中模拟了进程的**fork**过程以及相关**pid**的打印次数



3. fork and fork more 将`fork1.c`和`fork2.c`添加头文件，将函数加入`main`函数经行调用。结果如下：

```
control@ubuntu:~/Desktop/lab4/sourc$ ./fk1;ps
L0L1
Bye
Bye
L0L1
Bye
Bye
control@ubuntu:~/Desktop/lab4/sourc$ ./fk2;ps
L0
L1
Bye
Bye
L1
Bye
Bye
```

在源代码中，两个函数不同的地方就是其中一个输出内容的结尾没有加上`\n`符号，`\n`的作用就像是`C++`的`std::endl`（这里说像`C++`可能不太合适，但是我当时是通过`C++`才了解到**刷新缓冲区**这么个概念的，这里就按照我的认知过程来解释好了）。`\n`会将输出到标准输出流中的数据强制输出，并添加一个换行。`fork1`中并没有强制地将数据输出，于是紧接着就执行了`fork()`函数，将缓冲区的内容一并带入了子进程所在的平行宇宙，所以后面会打印出两个**L0**。

Processes

1. collatz.c

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

int main(int argc, char *argv[])
{
    int n = atoi(argv[1]);

    while(n != 1){
        printf("%d,", n);
        n = (n % 2) ? (3 * n + 1) : (n / 2);
    }
    printf("1\n");
    return 0;
}
```

```
control@ubuntu:~/Desktop/lab4/code$ ./co 13
13,40,20,10,5,16,8,4,2,1
control@ubuntu:~/Desktop/lab4/code$ ./co 100
100,50,25,76,38,19,58,29,88,44,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1
control@ubuntu:~/Desktop/lab4/code$
```

2. collatz.c

   利用**execlp()**

```
control@ubuntu:~/Desktop/lab4/code$ gcc collatz2.c -o co2
collatz2.c: In function ‘main’:
collatz2.c:23:3: warning: implicit declaration of function ‘wait’ [-Wimplicit-fu
nction-declaration]
   wait(NULL);
   ^
control@ubuntu:~/Desktop/lab4/code$ ./co2 55
55,166,83,250,125,376,188,94,47,142,71,214,107,322,161,484,242,121,364,182,91,27
4,137,412,206,103,310,155,466,233,700,350,175,526,263,790,395,1186,593,1780,890,
445,1336,668,334,167,502,251,754,377,1132,566,283,850,425,1276,638,319,958,479,1
438,719,2158,1079,3238,1619,4858,2429,7288,3644,1822,911,2734,1367,4102,2051,615
4,3077,9232,4616,2308,1154,577,1732,866,433,1300,650,325,976,488,244,122,61,184,
92,46,23,70,35,106,53,160,80,40,20,10,5,16,8,4,2,1
```

Thread

- collatz-thr.c

```
control@ubuntu:~/Desktop/lab4/sourc$ gcc collatz-thrd.c -lpthread -o x
control@ubuntu:~/Desktop/lab4/sourc$ ./x 13
13
40
20
10
5
16
8
4
2
1
```

> 详细代码见压缩包

实验数据或结果

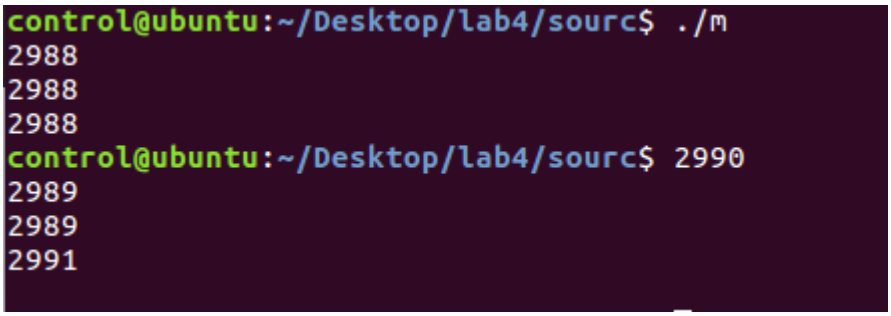
结果已经在**实验步骤**中展示

实验思考

关于使用**fork()**函数后，终端提示符显示异常的思考

问题描述

在使用`fork()`函数创建出多个进程时，让每个进程打印出自己的**pid**，会出现终端的命令提示符提前出现的现象（一般都是在程序结束时，命令提示符才会自动出现）。




```c
/*
对实验提供的源代码经行了简化
*/
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>

int main()
{
    printf("%d\n", getpid());
    fork();
    printf("%d\n", getpid());
    fork();
    return 0;
}
```

问题分析

1. 当执行的程序结束后，命令提示符才会出现，那么，执行多次`fork()`函数后，此时应该有多个进程先后终止，命令提示符会重现多次才对，但是，**实际上只出现了一次**。
2. 命令提示符出现的地点不是随机的，当父进程的**pid**被全部打印完后，才出现了命令提示符。
3. 在所执行程序后面添加其他命令顺序执行时，有些命令会导致命令提示符正常出现，有些不会（意外发现）。如图：

```
control@ubuntu:~/Desktop/lab4/sourc$ ./m;ps
3050
3050
3050
3052
3051
3051
3054
  PID TTY          TIME CMD
 2448 pts/4    00:00:00 bash
 3053 pts/4    00:00:00 ps
```

```
control@ubuntu:~/Desktop/lab4/sourc$ ./m;echo "xxxx"
3264
3264
3264
xxxx
control@ubuntu:~/Desktop/lab4/sourc$ 3266
3265
3265
3267

control@ubuntu:~/Desktop/lab4/sourc$
```

上网查了一下`shell`的工作原理，提取了以下几个关键点：

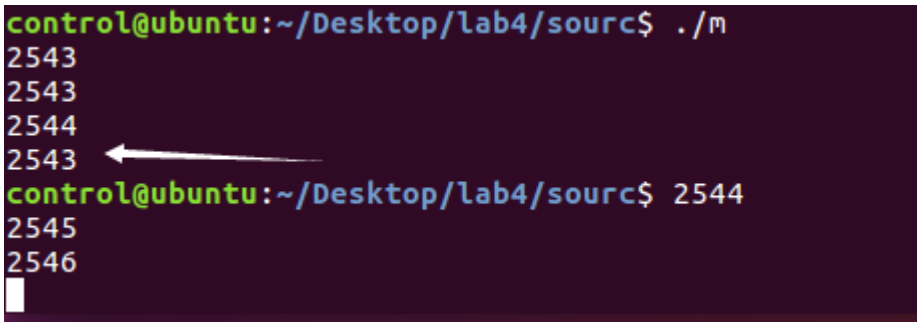
> **shell**也是一个进程 **shell**会分析用户的输入，将输入的内容改造成使用**execve**（）函数启动的程序 **shell**调用`fork()`函数，来创建子进程，子进程再调用**execve**（），完成启动 **shell**进程还要通过`wait()`函数来等待子进程结束

有了上面四点，基本上答案就出来了。`shell`调用`wait()`函数是一个关键，等子进程结束后，才返回`shell`。**这里的子进程，是直接子进程，也就是终端打印出的pid中最小的那一个**。当这个进程结束后，`shell`就会认为用户输入的那个程序结束了，然后就会重现命令提示符，但是用户输入的那个程序自己又`fork`出一堆子进程留在内存里，他们还要向标准输出流中输出东西！所以终端上又会出现一些pid数值。

```
control@ubuntu:~/Desktop/lab4/sourc$ ./m
2988
2988
2988
control@ubuntu:~/Desktop/lab4/sourc$ 2990
2989
2989
2991
```

所以这里终端上显示了3个2988后（父进程结束），命令提示符出现了。**shell**只会注意直接调用自己的那个进程，而不在意随后又衍生出的一堆子进程，所以命令提示符只会重现一次。随后，那些子进程打印出自己的pid后也结束了。

我修改了一下代码，让父进程的pid出现的稍微晚一点。结果也是在三次2543出现后出现了提示符。



到这里，问题1，2已经解决。那么我意外发现的问题3呢？

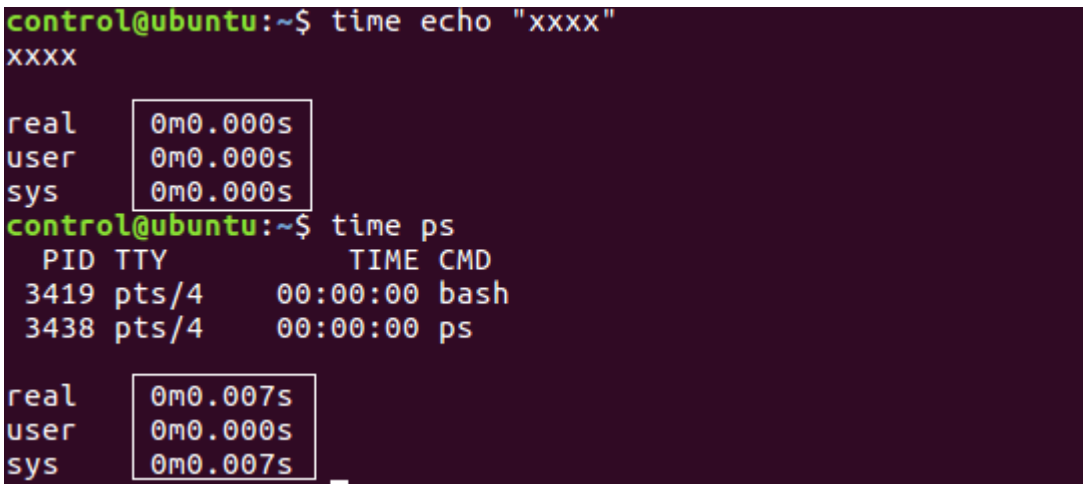
使用`;`将命令隔开，意味着当这一串命令执行完后，才重现提示符。但为何使用不同的命令有不同的结果呢？使用`ps`命令，先打印了所有pid，然后展示`ps`的结果；使用`echo`命令，pid打印到一半就把`echo`的结果输出了。

其实，要等待第一个进程结束才会执行第二个，所以`echo`也好，`ps`也好,都是在**父进程的pid**打印完后再出现的。那为什么使用`ps`命令，所有子进程pid都提前打印了，而使用`echo`命令则子进程pid都推迟打印了？

我猜想是和**时间片**有关系：**ps命令所需的时间大于了一个时间片，还没等它执行完，处理机就被分给了那些子进程，然后子进程在一个时间片内打印pid，结束，当处理机再分配给ps时，它才把工作做完，将信息显示在终端上，于是就有了 所有的pid打印完成的现象。同理，echo所需的时间比较少，一个时间片内就能完成，所以它的输出在子进程之前。**

```
如果我的猜想正确，那么执行顺序就是这样：
父进程-->ps(1)-->子进程-->ps(done)
父进程-->echo-->子进程
```

正好，系统有自带的工具可以查看命令运行时间，执行时间快慢已经很明显了：

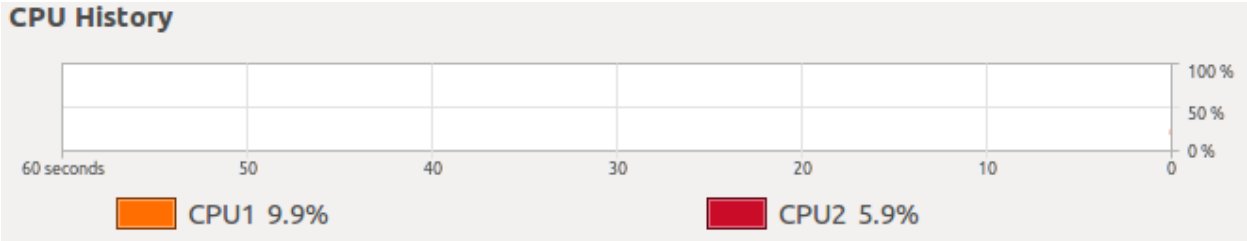


到这里，问题3应该也算是解决了

新的问题

> 灾难总是接踵而至，这也是不能避免的。——罗罗诺亚·索隆

1. 以上分析都是针对单核处理机，那多核处理机会怎样呢？




```
control@ubuntu:~$ ./dd
1996
1996
1997
control@ubuntu:~$ ./dd
1998
1998
1999
control@ubuntu:~$ ./dd
2000
2000
2001
control@ubuntu:~$ ./dd
2002
2002
2003
```

按照结果来看，应该是父进程先结束了，但是没有提示符乱入.....如果程序的规模增大一点，提示符又乱入了.....

```
control@ubuntu:~/Desktop/lab4/sourc$ ./m
2045
2045
2046
2045
2046
2048
2047
```

```
control@ubuntu:~/Desktop/lab4/sourc$ ./m
2066
2066
2067
2066
2067
2066
2067
2071
2068
control@ubuntu:~/Desktop/lab4/sourc$ 2068
2070
2072
2069
2069
2073
```

2. 将打印结果重定向到一个文件中，根据结果，应该是所有的输出都被重定向了，高兴地打开文件一看.....

```
control@ubuntu:~/Desktop/lab4/sourc$ ./mm
2078
2078
2079
control@ubuntu:~/Desktop/lab4/sourc$ ./mm > 1.txt
control@ubuntu:~/Desktop/lab4/sourc$ cat 1.txt
2082
2082
2082
2083
```

   为啥多了一个**pid**号？？？？

两个新的问题暂时没有解决。

参考资料

《Linux 程序设计（第二版）》金国庆主编