操 作 系 统
实 验 报 告

实验名称：实验一 进程的创建实验

姓名： 王迎旭

学号： 16340226

实验名称：**进程的创建实验**

一、实验目的：

加深对进程概念的理解，明确进程和程序的区别。进一步认识并发执行的实质。

二、实验要求：

认识进程生成的过程，学会使用 fork 生成子进程，并知道如何使子进程完成与父进程不同的工作。

三、实验过程：

(1) 编译并运行 PPT 中所给 C 代码并解释现象

这周的作业主要是尝试使用 fork 创建简单的进程并且通过在 linux 环境下编译运行，进而去了解进程之间的相互关系。

所以需要先了解 fork 这个函数的基本特性，如下：

1. 调用 fork()可以创建进程。不过，fork 是把进程当前的情况拷贝一份，执行 fork 时，进程只拷贝下一个要执行的代码到新的进程。

2. fork 调用的一个奇妙之处就是它仅仅被调用一次，却能够返回两次，它可能有三种不同的返回值：

① 在父进程中，fork 返回新创建子进程的进程 ID;
② 在子进程中，fork 返回 0;
③ 如果出现错误，fork 返回一个负值;

3.fork 出错可能有两种原因：

① 当前的进程数已经达到了系统规定的上限，这时 errno 的值被设置为 EAGAIN。
② 系统内存不足，这时 errno 的值被设置为 ENOMEM。

在完成实验时候为了更好的追踪进程之间的关系，我使用了 **getppid()查询父进程函数** 与 **getpid()查询当前进程函数**，同时也**输出相应的子进程的编号**，所以对程序也进行了相应的修改。

修改后如下：

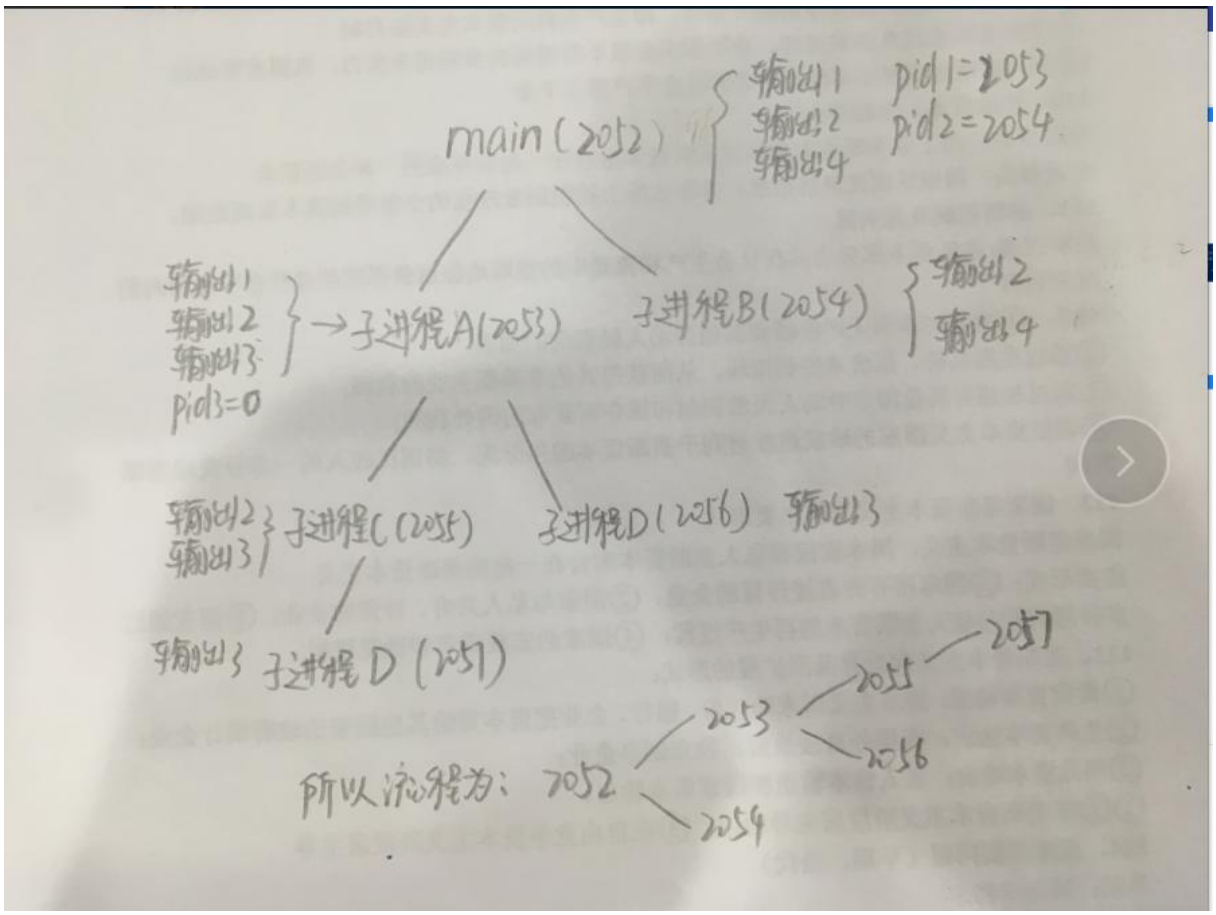
```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main() {
    int pid1 = fork();
    printf("**1**%d  %d  %d  \n",getppid(),getpid(),pid1);
    int pid2 = fork();
    printf("**2**%d  %d  %d\n",getppid(),getpid(),pid2);
    if (pid1 == 0) {
        int pid3 = fork();
        printf("**3**%d  %d  %d\n",getppid(),getpid(),pid3);
    }
    else
    {
        printf("**4**\n");
    }
    return 0;
}

```

编译运行之后命令框显示如下：

```
[dell@localhost Desktop]$ gcc test.c
[dell@localhost Desktop]$ ./a.out
**1**1848  2052  2053
**2**1848  2052  2054
**4**
**1**2052  2053  0
**2**1  2053  2055
**3**1  2053  2056
[dell@localhost Desktop]$ **2**1  2054  0
**4**
**3**1  2056  0
**2**1  2055  0
**3**1  2055  2057
**3**2055  2057  0
```

所以这样就很容易可以画出进程之间的相互联系关系：



相关解释:

①

Ⅰ、程序开始 main 函数执行:

创建进程 A,输出 1，pid1 = 2053，创建进程 B ，pid2 = 2054 ，输出 2 ，此时 pid1 ！= 0 ，所以输出 4 ，main 函数结束。

Ⅱ、进程 A 开始执行，此时 pid1 = 0

输出 1，创建进程 AB，pid2 = 2055，输出 2，由于 pid1 = 0 ，所以创建进程 ABC1，输出 3，进程 A 结束

Ⅲ、进程 B 开始执行，此时 pid1 = 2053，pid2 = 0

输出 2，输出 4 ，进程 B 结束

Ⅳ、进程 ABC1 开始执行

输出 3，进程 ABC1 结束

Ⅴ、进程 AB 开始执行，此时 pid1 = 0 ，pid2 = 0

输出 2，创建进程 ABC2，输出 3，进程 AB 结束

Ⅵ、进程 ABC2 开始执行

输出 3，进程 ABC2 结束

Ⅶ、所以输出顺序为 124123243233

② 观察截图会发现，进程 2053、2055、2056 的父进程是 1，并不是 2053 对应的 2052 以及 2055 与 2056 对应的 2053，这是因为 2053 执行完之后，这个 main 已经结束了，所以整个进程相当于已经结束也就是父进程已经死亡，但是子进程一定要有对应的父进程，所以系统默认给这三个子进程的父进程设为 1.

③ 观察截图还会发现，有些进程的子进程为 0，这是因为这个进程并不会生成子进程，所以就置为 0.

④ 同时我发现了一个问题，就是多次执行可执行文件，输出 123 的总个数不变，但是输出 123 的顺序却不同，这说明所有的进程都是可以被执行的，但是进程被执行的顺序是有差异的；刚开始做到这个地方时候不太懂为什么同样的可执行程序输出结果不同，后来查询资料之后发现，进程之间存在竞争，并且这个存在于进程被创建之后的运行过程中，竞争体系的存在也就导致了执行可执行文件时候输出的结果不相同。

(2) 通过实验完成教材上的习题 3.4

代码：

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

int value = 5;

int main(){
    pid_t pid ;
    pid = fork() ;
    if(pid == 0)
    {
        value += 15 ;
    }
    else if(pid > 0)
    {
        wait(NULL);
        printf("PARENT:value=%d",value);/*LINE A*/
        exit(0);
    }
}
```

运行结果：

```
[dell@localhost Desktop]$ gcc test.c
test.c: 在函数‘main’中：
test.c:16:9: 警告： 隐式声明函数‘wait’ [-Wimplicit-function-declaration]
         wait(NULL);
         ^~~~
test.c:18:9: 警告： 隐式声明函数‘exit’ [-Wimplicit-function-declaration]
         exit(0);
         ^~~~
test.c:18:9: 警告： 隐式声明与内建函数‘exit’不兼容
test.c:18:9: 附注： include ‘<stdlib.h>’ or provide a declaration of ‘exit’
[dell@localhost Desktop]$ ./a.out
PARENT:value=5[dell@localhost Desktop]$
```

相关解释：

① 由于题中给的头文件与所要使用的函数搭配出了点冲突，所以会出现警告信息，但是并不影响程序的正常运行。

② 至于输出 value = 5，是因为当程序的控制权返回到父进程的时候，这个 value 的值仍会保持为进入子进程之前的那个初始值。

③ 这里我把 value 放在了 main 函数中再次进行了编译并运行（如图），发现输出的结果仍然是 5，这就证明不管是全局变量还是局部变量，子进程对参数更改之后，父进程使用参数时候仍然是使用的参数的初始值。

```
1  #include <sys/types.h>
2  #include <stdio.h>
3  #include <unistd.h>
4
5
6  int main(){
7      pid_t pid ;
8      pid = fork() ;
9      int value = 5;
10     if(pid == 0)
11     {
12         value += 15 ;
13     }
14     else if(pid > 0)
15     {
16         wait(NULL);
17         printf("PARENT:value=%d",value);/*LINE A*/
18         exit(0);
19     }
20 }
```

```
test.c:18:9: 警告： 隐式声明与内建函数‘exit’不兼容
test.c:18:9: 附注： include ‘<stdlib.h>’ or provide a
[dell@localhost Desktop]$ ./a.out
PARENT:value=5[dell@localhost Desktop]$ gcc test.c
test.c: 在函数‘main’中：
test.c:16:9: 警告： 隐式声明函数‘wait’ [-Wimplicit-fu
         wait(NULL);
         ^~~~
test.c:18:9: 警告： 隐式声明函数‘exit’ [-Wimplicit-fu
         exit(0);
         ^~~~
test.c:18:9: 警告： 隐式声明与内建函数‘exit’不兼容
test.c:18:9: 附注： include ‘<stdlib.h>’ or provide a
[dell@localhost Desktop]$ ./a.out
PARENT:value=5[dell@localhost Desktop]$
```

(3) 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；父进程显示字符“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。

代码：

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

int main(){
    printf("a\n");
    pid_t pid1 = fork();
    if(pid1 == 0 )
    {
        printf("b%d  %d  %d\n",getppid(),getpid(),pid1);
        pid_t pid2= fork();
        if(pid2 == 0)
        {
            printf("c%d  %d  %d\n",getppid(),getpid(),pid2);
        }
    }
    return 0;
}
```

运行结果：

```
[dell@localhost Desktop]$ gcc test.c
[dell@localhost Desktop]$ ./a.out
a
b1  2215  0
c1  2216  0
```

相关解释：

① main 函数执行，输出 a ，创建进程 A，pid1 = 2214 ，main 结束

② 进程 A 执行 ，输出 b，创建进程 AB，此时 pid1 = 0 ，pid2 = 2015，进程结束

③ 进程 AB 执行，pid1 = 0 ， pid2 = 0 ， 输出 c ，进程 AB 结束

四、实验感想：

① 第一次研究进程之间的相互联系，如果没有借助两个辅助函数帮助 debug，真的是十分难以寻找对应的父子进程关系。

② 即使是完成了此次实验作业，但是仍然还是要对进程之间的联系做进一步研究，力求达到不使用辅助函数就可以搞清彼此之间的关系。

③ 作业布置的那天就抽了时间做完了三个题写下了前两个感想,但是感觉收获并不是很大,隔了大概 3 天又重新做了一遍，这次没有借助辅助函数，但是这次靠着自己的理解弄清楚程序执行的顺序以及机理，同时也对进程之间的关系有了更深层次的理解。**不过进程之间的竞争问题，还是没有搞清楚**