

实验五：UNIX V6++ 中新进程创建与父子进程同步

1. 实验目的

结合课程所学知识，通过在 UNIX V6++ 实验环境中编写使用了父进程创建子进程的系统调用 `fork`，进程终止及父子进程同步的系统调用 `exit` 和 `wait` 的应用程序，并观察他们的运行结果，进一步熟悉 UNIX V6++ 中关于进程创建、调度、终止和撤销的全过程，实践 UNIX 中最基本的多进程编程技巧。

2. 实验设备及工具

已配置好 UNIX V6++ 运行和调试环境的 PC 机一台。

3. 预备知识

- (1) 熟悉如何在 UNIX V6++ 中编译、调试和运行一个用户编写的应用程序。
- (2) 熟练掌握 UNIX V6++ 进程管理的相关算法与实施细节。
- (3) 熟悉 `fork`，`exit`，`wait` 和 `sleep` 四个系统调用的执行过程。

4. 实验内容

4.1. 父进程等待所有子进程

利用前序实验中已经掌握的方法，在 UNIX V6++ 中添加一个名为 `procTest.exe` 的可执行程序。该程序通过 `fork` 系统调用，创建出如右图所示的进程树。从图中可以看出，0#和1#进程是系统启动后就创建好的，运行 `procTest.exe` 后，创建 2#进程从 `main1` 的入口开始执行。也就是说，`main1` 中要相继 `fork` 出 3#，4#和 5#三个子进程，且要求 3 个进程的父子关系如右图所示，即：4#和 5#是 3#的子进程。

本次实验将针对 3#，4#和 5#三个进程之前的父子关系，通过在三个进程代码的不同位置添加 `exit`，`wait` 和 `sleep` 系统调用，实现 3#，4#和 5#进程间不同的同步要求。

代码 1 是我们给出的参考代码，其中标注了每个进程在其中负责执行的部分（这里需要说明的是，代码中 `getppid` 函数为我们在实验四中添加的库函数，获取指定进程的父进程，对存在的进程，输出其父进程 ID 号，不存在的进程，输出 -1）。图 1 给出了代码 1 的执行时序，而图二给出了代码 1 的执行结果。结合图 1 和图 2，我们对代码 1 给出如下说明：

```
#include <stdio.h>
#include <sys.h>
main1()
{
    int ws=2;
    int i,j,k,pid,ppid;

    if(fork())
    {
        //2#
        sleep(2);
        for(k=1;k<6;k++)
        {
            printf("%d,%d; ",k,getppid(k));
        }
        printf("\n");
    }
    else
    {
        //3#
        if(fork())
        {
            if(fork())
            {
                //3#
                pid=getpid();
                ppid=getppid(pid);
                for(k=0;k<ws;k++)
                {
                    i=wait(&j);
                    printf("Process %d#:My child %d is finished with exit status %d\n",pid,i,j);
                }
                printf("Process %d# finished: My father is %d\n",pid,ppid);
                exit(ppid);
            }
            else
            {
                //5#
                pid=getpid();
                ppid=getppid(pid);
                printf("Process %d# finished: My father is %d\n",pid,ppid);
                exit(ppid);
            }
        }
        else
        {
            //4#
            pid=getpid();
            ppid=getppid(pid);
            printf("Process %d# finished: My father is %d\n",pid,ppid);
            exit(ppid);
        }
    }
}
```

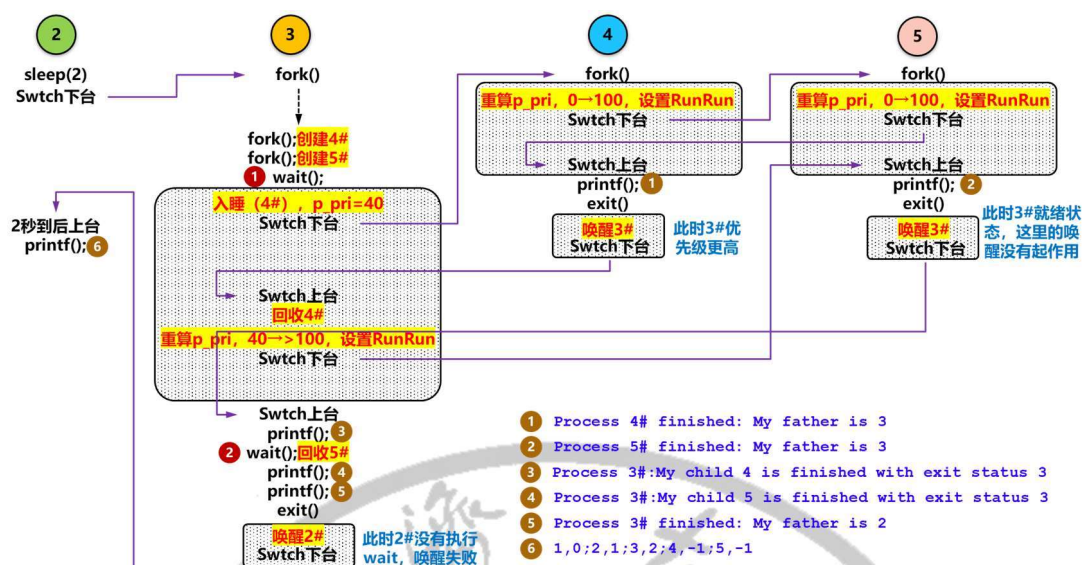


图 1: 代码 1 程序执行时序说明

```
Bochs for Windows - Display
[ ]#cd bin
[/bin]#procTest.exe
Process 4# finished: My father is 3
Process 5# finished: My father is 3
Process 3#:My child 4 is finished with exit status 3
Process 3#:My child 5 is finished with exit status 3
Process 3# finished: My father is 2
1,0; 2,1; 3,2; 4,-1; 5,-1;
[/bin]#

Process 4 (Status:5) end wait
Process 5 is exiting
Process 3 finding dead son. They are Process 5 (Status:5) end wait
Process 3 is exiting
Process 2 is exiting
My:2 's child 3 passed to 1#processend sleep
Process 2 (Status:5) end wait

CTRL + 3rd button enables mouse  IPS: 36.800M  NUM  CAPS  SCRL  HD:0-N
```

图 2: 父进程执行两次 wait 等待两个子进程

(1) 2#进程执行 main1 主程序, fork 出 3#进程, 通过 sleep 等待 2 秒钟后, 打印输出当前系统中所有进程及其父进程 ID 号 (见图 1 中 printf⑥)。此时, 4#和 5#已经终止并由 3#回收, 因而进程号不存在, getppid 返回为-1; 而 3#虽然终止, 但由于 2#没有执行 wait 回收, 因而可以看到该进程还存在, 且其父进程仍然为 2#;

(2) 3#进程作为 2#进程的子进程, 先后 fork 出 4#和 5#两个子进程 (由于这两个 fork 中 3#进程作为父进程, 优先数变化不大, 因而重算时不会设置 RunRun, 使得 3#进程始终

没有被抢占)，并通过循环执行两个 `wait` 操作，分别处理两个子进程的终止。其中，在执行第一次 `wait` 时，因等待子进程终止而入睡，4#进程执行 `exit` 时将其唤醒，重新上台后回收 4#，而在返回用户态前，由于优先数的变化被 5#进程抢占。再次上台后，依次执行图 1 中 `printf`③，第二个 `wait`（回收 5#）和 `printf`④。最后，执行 `printf`⑤后，通过 `exit` 结束自己；

（3）在 3#进程入睡后，对于此时同在就绪状态的 4#和 5#，作为刚刚创建好的子进程，具有相同的优先数 0，由于 UNIX V6++的 `Swch` 程序优先选择 ID 号小的#进程，所以 4#先上台，在 `fork` 返回用户态前由于重算优先数被 5#抢占。而 5#在 `fork` 返回用户态前，同样由于重算优先数被 4#抢占。

（4）4#再次上台后，执行 `printf`①打印出自己的 ID 号和父进程的 ID 号后，结束自己，唤醒 3#，并向 3#发生终止码，终止码为父进程 ID 号。而 5#在 3#执行第一个 `wait` 返回用户态前抢占 3#，执行 `printf`②打印自己的 ID 号和父进程的 ID 号后，结束自己，同样唤醒 3#，并向 3#发生终止码，终止码为父进程 ID 号。但是这里需要注意的是，5#进程执行唤醒操作时，3#在就绪状态，因而唤醒操作实际上没有作用。

接下来，我们将通过一组实验，尝试在不同的位置设置 `wait`，`exit` 和 `sleep` 来改变 3#，4#和 5#的同步关系。

4.2. 父进程先于所有子进程结束

这里，如果我们不希望 3#进程等待 4#进程和 5#进程结束后再结束，而是希望 3#进程先结束，可以怎么做呢？很简单，只要 3#进程不执行 `wait` 操作即可。因此代码中只需将 `ws` 的值改为 0。图 3 示出了修改后的代码的执行时序，而图 4 给出了程序执行的结果。结合图 3 和图 4，我们有以下几点需要说明：

（1）与代码 1 一样，这里的 3#进程依然先于两个子进程从 `fork` 返回，但是没有执行 `wait` 入睡等待两个子进程，而是直接打印输出自己的 ID 和父进程的 ID 号（图 3 中的 `printf` ①）；之后进程执行 `exit` 终止，将 4#和 5#的父进程改为了 1#进程，这点可以从 4#和 5#终止前输出的父进程 ID 中得到验证；

（2）3#进程终止后，4#，5#依次上台执行，分别打印输出自己的 ID 和父进程的 ID 号后终止。由于父进程已经被改为 1#进程，这里两个进程都将唤醒 1#进程回收进程图象。这也就是为什么 `printf`②和 `printf`③显示两个进程的父进程是 1#，而 `printf`④中看到两个进程的父亲已经是不存在的进程了。

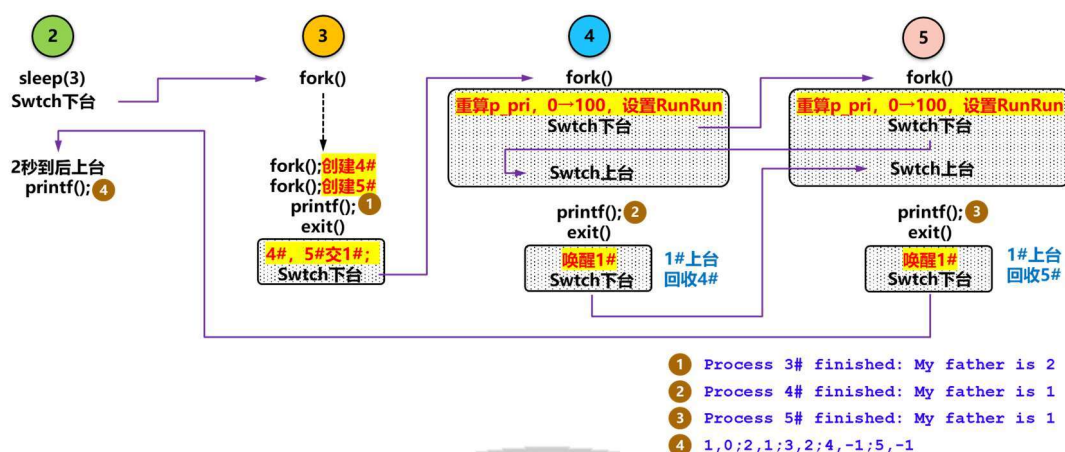


图 3: #进程先于两个子进程结束的程序执行时序

```

[~/bin]#cd bin
[~/bin]#procTest.exe
Process 3# finished: My father is 2
Process 4# finished: My father is 1
Process 5# finished: My father is 1
1,0; 2,1; 3,2; 4,-1; 5,-1;
[~/bin]#

end sleep
Process 2 (Status:2) Process 4 (Status:5) end wait
Process 5 is exiting
Process 1 finding dead son. They are Process 2 (Status:2) Process 5 (Status:5)
end wait
Process 1 finding dead son. They are Process 2 (Status:2) wait until child proc
ess Exit! Process 2 is exiting
My:2 's child 3 passed to 1#processend sleep
Process 2 (Status:5) end wait
  
```

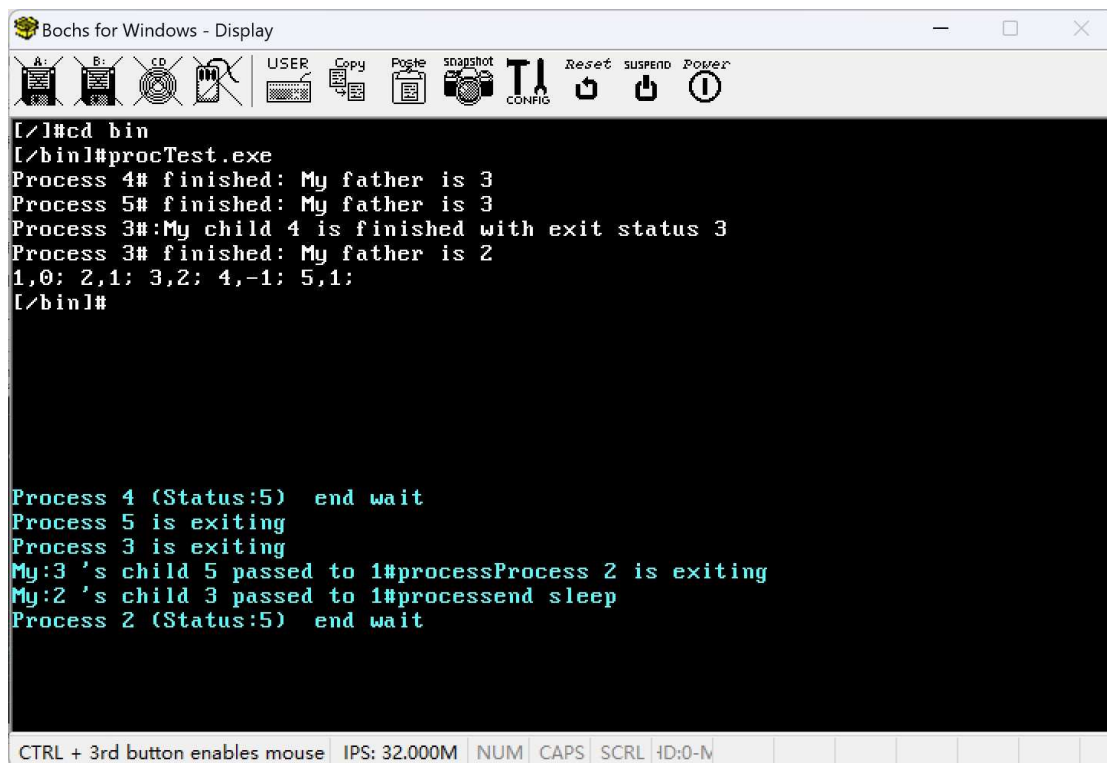
图 4: 3#进程先于两个子进程结束的程序执行结果

总结来说，如果父进程先于子进程终止，会将子进程交给 1#进程。那么子进程的完整的撤销工作，将在子进程终止时，通过唤醒 1#进程，由 1#进程来完成。

4.3.父进程先于部分子进程结束

通过前述两份代码，我们可以发现，执行几次 `wait`，父进程就可以接收并处理几个终止的子进程。这里我们可以尝试只删除代码 1 中的一个 `wait`，而不是两个都删除。这样父进程将只等待其中的一个子进程，于是我们可以得到如图 5 所示的输出。可以看到父进程只

接收到了 4#子进程的终止码。请读者参照上一个实验，尝试绘图或文字解释这个输出。特别是回答以下两个问题：



```

[/]#cd bin
[/bin]#procTest.exe
Process 4# finished: My father is 3
Process 5# finished: My father is 3
Process 3#:My child 4 is finished with exit status 3
Process 3# finished: My father is 2
1,0; 2,1; 3,2; 4,-1; 5,1;
[/bin]#

Process 4 (Status:5) end wait
Process 5 is exiting
Process 3 is exiting
My:3 's child 5 passed to 1#processProcess 2 is exiting
My:2 's child 3 passed to 1#processend sleep
Process 2 (Status:5) end wait
  
```

图 5：父进程只等待 4#进程的执行结果

- (1) 为什么 4#进程和 5#进程终止时，都能找到自己的父进程是 3#进程？
- (2) 最后的打印输出中，5#进程的父进程是 1#，说明 5#进程此时还是存在的，为什么？5#进程的图象将由谁在什么时间回收？

通过上面的几组实验，我们可以总结出父子进程同步过程中的几种情况，如表 1 所示。请读者结合几个实验的结果，认真阅读体会。

表 1：子进程终止时的不同情况

子进程终止时：	完成的操作：
(1) 父进程已终止	由于父进程终止时已经将子进程的父进程改为 1#，所以子进程终止时唤醒 1#，1#完整撤销子进程。 * 如图 4 中的 4#和 5#。
(2) 父进程未终止，且执行 wait 等待子进程	唤醒父进程，父进程完整撤销子进程。 * 如图 2 中的 4#和 5#，图 5 中的 4#。
(3) 父进程未终止，但没有执行 wait 等待子进程	由于唤醒父进程无效，子进程没有完全撤销。父进程终止时，将子进程交给 1#，等待 1#进程处理。 * 如图 5 中的 5#

4.4. 抢占父进程

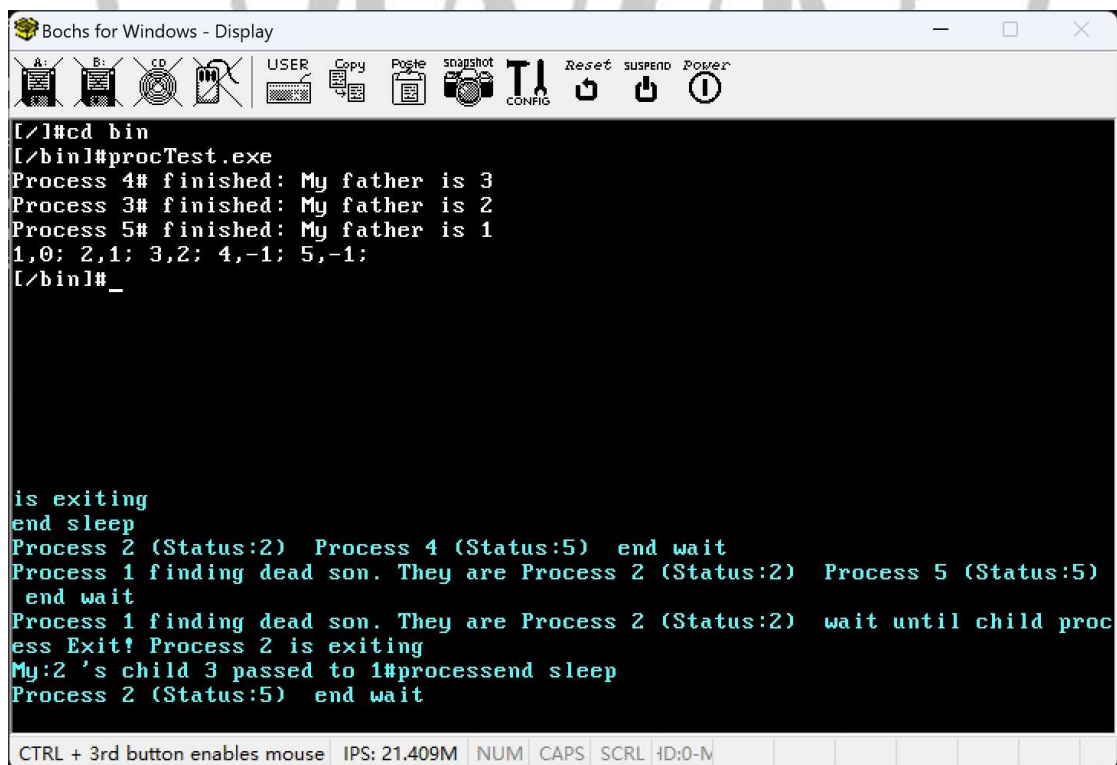
在学习 `fork` 系统调用过程中，我们曾经提及：父进程在内存成功创建子进程后，在 `fork` 系统调用的末尾，由于重算优先数，有可能导致 `RunRun` 标志位被设置，进而在返回用户态前的例行调度中，由于子进程的 `p_pri = 0`，优先级较高，从而被子进程抢占。

但是在我们目前的代码中，大家可以多次运行，会发现，除非 3# 执行 `wait`，否则总是其先从 `fork` 返回。也就是说，除非 3# 自己主动放弃处理器，否则子进程是没有机会抢占父进程的。

请读者根据课程所需知识，认真分析并回答下列问题：

(1) 为什么 4.1~4.3 的实验中，父进程 3# 进程始终没有被抢占？在本实验的代码中，如果父进程 3# 进程不执行 `wait`，可以被子进程抢占的时机和条件是什么？

(2) 先将 3# 进程执行代码部分的两次 `wait` 操作删除，再尝试两种修改代码的方案，创造 3# 进程可以被抢占的机会，进而得到如图 6 和图 7 所示的执行结果。观察图 6 和图 7 不难发现，图 6 中 3# 仅被 4# 进程抢占，而图 7 中，4# 和 5# 均抢占了 3#。因为被抢占的位置不同，导致程序最后输出时，图 6 中 4# 和 5# 已被撤销，而图 7 中，4# 和 5# 显示父进程为 1#，为什么？



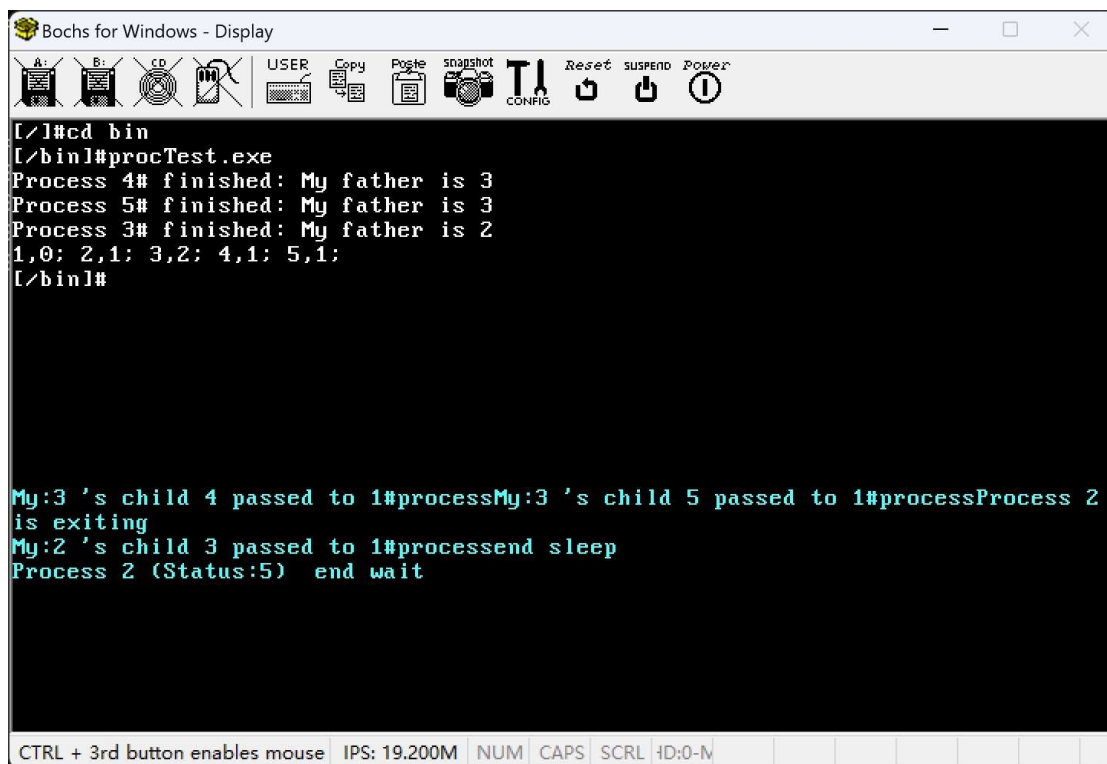
```

Bochs for Windows - Display
[ ]#cd bin
[/bin]#procTest.exe
Process 4# finished: My father is 3
Process 3# finished: My father is 2
Process 5# finished: My father is 1
1,0; 2,1; 3,2; 4,-1; 5,-1;
[/bin]#_

is exiting
end sleep
Process 2 (Status:2) Process 4 (Status:5) end wait
Process 1 finding dead son. They are Process 2 (Status:2) Process 5 (Status:5)
end wait
Process 1 finding dead son. They are Process 2 (Status:2) wait until child proc
ess Exit! Process 2 is exiting
My:2 's child 3 passed to 1#processend sleep
Process 2 (Status:5) end wait

```

图 6：父进程创建 5# 之前被抢占



```
[ / ]#cd bin
[ / bin ]#procTest.exe
Process 4# finished: My father is 3
Process 5# finished: My father is 3
Process 3# finished: My father is 2
1,0; 2,1; 3,2; 4,1; 5,1;
[ / bin ]#

My:3 's child 4 passed to 1#processMy:3 's child 5 passed to 1#processProcess 2
is exiting
My:2 's child 3 passed to 1#processend sleep
Process 2 (Status:5) end wait
```

图 7：父进程创建 5#之后被抢占

5. 实验报告要求

(1) (1 分) 完成实验 4.1 ~ 4.3，建立符合要求的进程树，并通过父进程是否执行 `wait`，执行几个 `wait` 来调整父子进程之前的同步顺序，实现父进程等待所有子进程、父进程先于所有子进程和父进程先于部分子进程等场景，截图展示程序运行结果；

(2) (1 分) 针对实验 4.3，采用绘图或者文字方式解释图 5 的输出，包括：进程的调度顺序及产生这样的调度顺序的原因，5#在最后的打印输出语句时，为什么显示进程还存在，且父进程为 1#，并回答 5#将由谁在何时回收；

(3) (2 分) 完成实验 4.4，首先回答 4.4 中的问题 (1) 和 (2)，然后设计合理的方案，修改代码实现图 6 和图 7 的输出结果，采用绘图或者文字方式解释输出，包括：进程的调度顺序及产生这样的调度顺序的原因。