**同济大学计算机系**

**操作系统课程实验报告**

****

**学 号 2251557**

**姓 名 代文波**

**专 业 计算机科学与技术**

**授课老师 方钰**

**实验五：UNIX V6++中新进程创建与父子进程同步**

**一、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。

**二、实验设备及工具**

已配置好的UNIX V6++运行和调试环境。

**三、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。

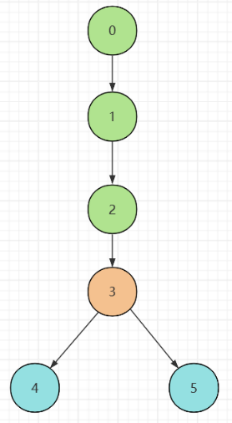
（2）熟练掌握 UNIX V6++进程管理的相关算法与实施细节。

（3）熟悉fork，exit，wait 和sleep 四个系统调用的执行过程。

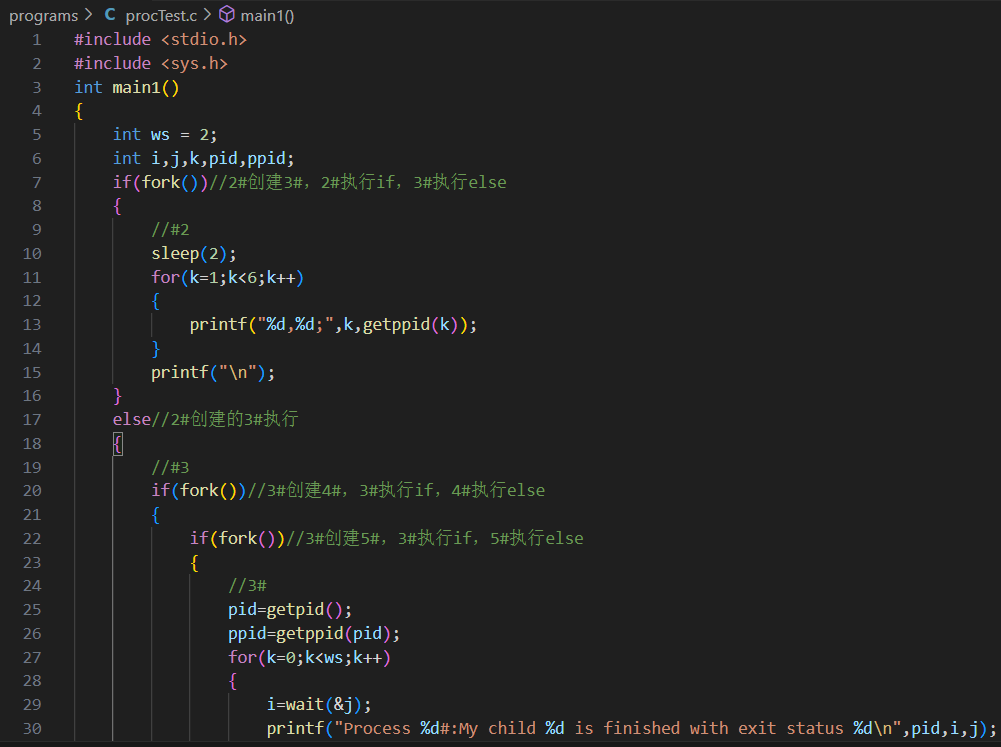
**四、实验内容**

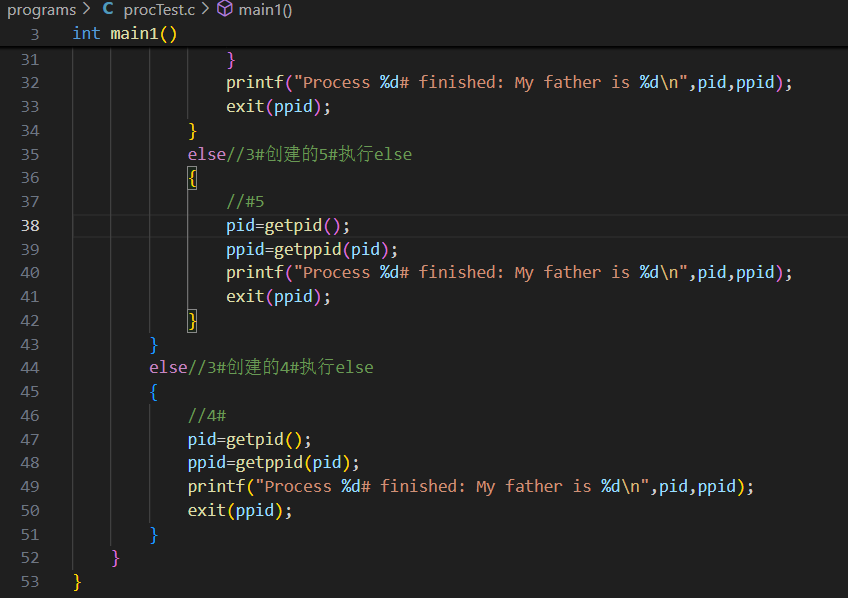
**第一部分：**完成实验 4.1 ~ 4.3，建立符合要求的进程树，并通过父进程是否执行 wait，执行几个 wait 来调整父子进程之前的同步顺序，实现父进程等待所有子进程、父进程先于所有子进程和父进程先于部分子进程等场景，截图展示程序运行结果；

**4.1 添加一个名为procTest.exe 的可执行程序。**

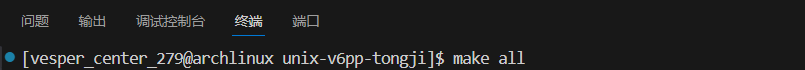
要求procTest.exe程序通过fork系统调用，创建出如下图所示的进程树。

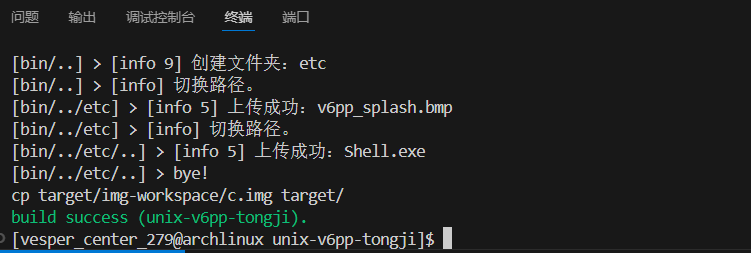
**4.1.1在program文件中加入一个名为procTest.c的文件**



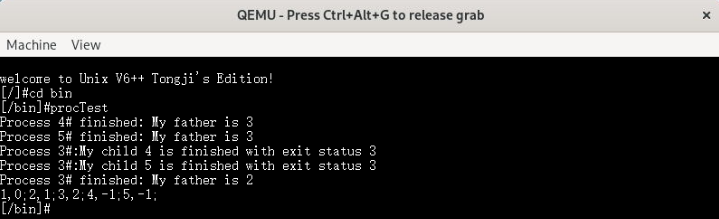


**4.1.2重新编译运行UNIX V6++代码**

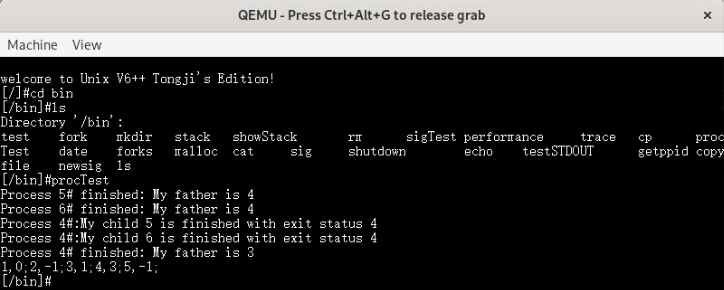




**4.1.3程序运行结果**



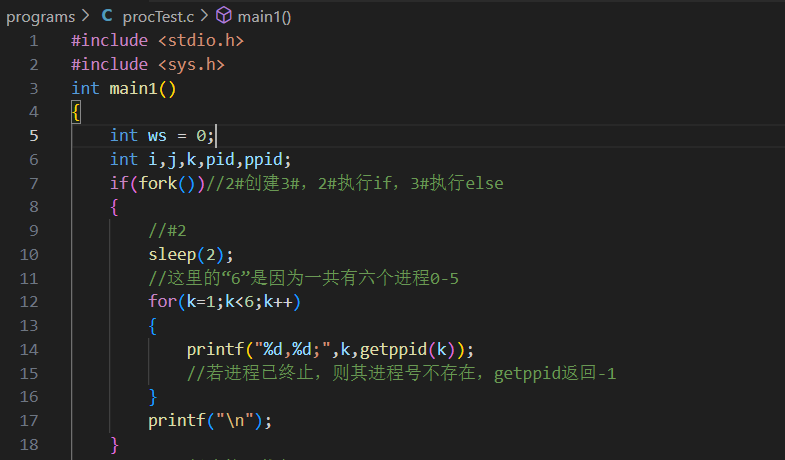
**【注意】**这里一旦先调用ls展示文件夹目录，则ls自己会作为一个进程，编号为2，导致后面的procTest的进程号会变成3号，进而导致程序预测结果都向后移动一位，进而造成有差异的程序运行结果如下：



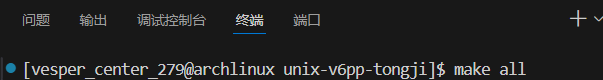
**4.2 父进程先于所有子进程结束**

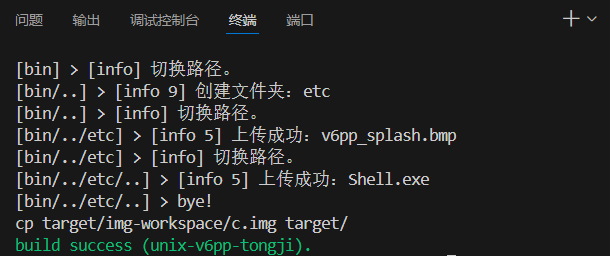
如果我们不希望3#进程等待4#进程和5#进程结束后再结束，而是希望3#进程先于两个子进程结束，只要3#进程不执行wait操作即可，即需将代码中ws的值改为0。

**4.2.1代码修改**

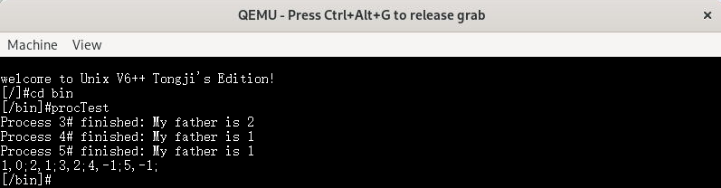


**4.2.2重新编译运行UNIX V6++代码**





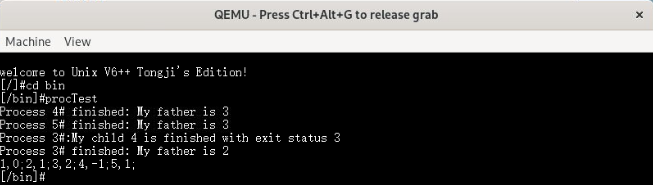
**4.2.3程序运行结果**



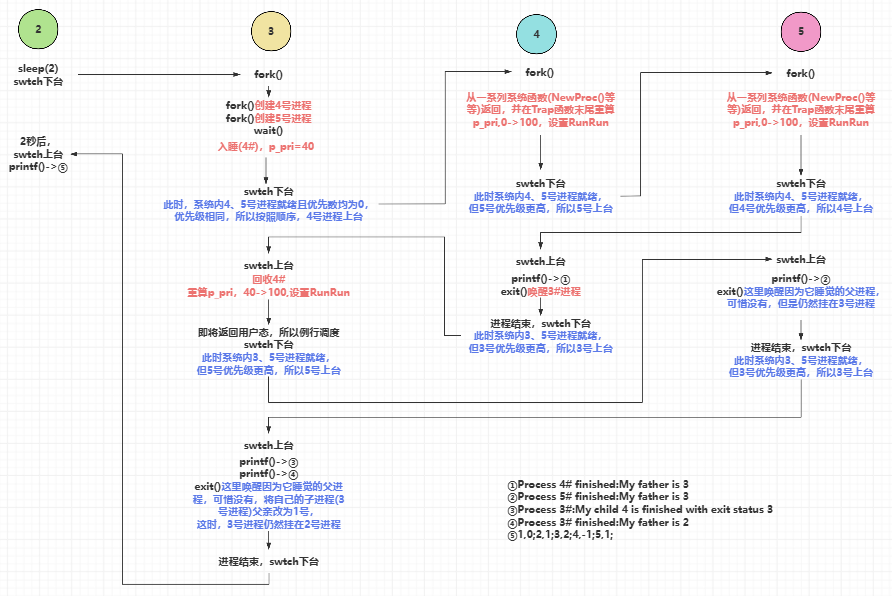
**第二部分：**

**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下图所示的输出。

****

**4.3.1绘图解释**

****

**4.3.2 问题解答**

**(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？**

答：①4#进程结束时，3#进程因wait()一直在睡觉，所以4#进程终止时可以找到父进程是3#；②对于5#进程，3#此前因4#唤醒而回收4#，并重算了3#自己的优先数并设置RunRun>0。在3#进程在wait()返回用户态时的例行调度中，5#进程抢占上台，所以尽管3#进程没有用wait()等待它，但是此时3#进程仍然存在，尚未消失，所以5#进程终止时可以找到父进程是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？

答：①5#进程结束时调用了exit()，想唤醒因为它睡觉的父进程来回收自己，可惜其父进程3#进程没有因为它入睡，所以并未成功唤醒（顺便提一下，3#进程此时处于就绪状态），进而5#进程只能清空内存并将user结构搬到盘交换区后终止进程，但是5#进程仍然挂在3#进程上。后来，3#进程结束时，将自己的子进程的父进程都设置为1#进程，这时5#进程的父进程就变成了1#进程，所以在2#进程最后的打印输出中，5#进程还存在并且其父进程是1#。②5#进程的图像最后会由1号进程在2#进程执行完毕后通过swtch上台后回收。

**第三部分：**

**4.4. 抢占父进程**

**4.4.1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？**

答：第一问：对于4.1~4.3的实验，3#进程一方面执行时间太短，没有执行时钟中断的复杂部分，一方面没有因中断唤醒优先级更高的进程，最后一方面执行过程中没有因调用输入输出设备等原因睡眠而下台。

第二问：如果如果父进程3#进程不执行 wait，可以被子进程抢占的时机和条件如下：

1. 时机：一秒结束，3#重算自己的优先数，必然会设置RunRun，进而例行调度中可以抢占；

条件：3#进程上台有足够多的事情去做，待够较长时间并且时钟中断来临时Time::lbolt满足让该进程执行复杂任务的条件（课上讲的是Time::lbolt>60,UNIX V6++中代码写的是Time::lbolt>120）;

1. 时机：给3#进程安排中断，让3#唤醒拥有更高优先级的进程而设置RunRun后，最后中 断返回时的例行调度。

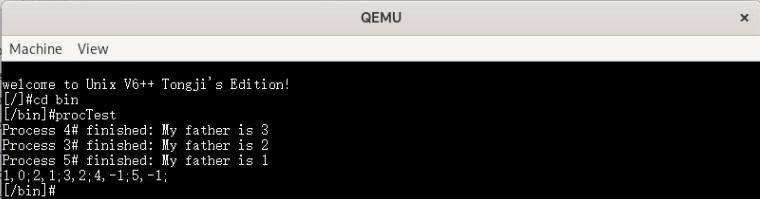
条件：3#进程响应中断时，必须唤醒比自己上台时优先级更高的进程进而设置RunRun

1. 时机：让3#进程因调用输入输出设备等原因睡眠进而下台。

条件：3#进程需要因为一些原因睡眠而主动下台让出cpu。

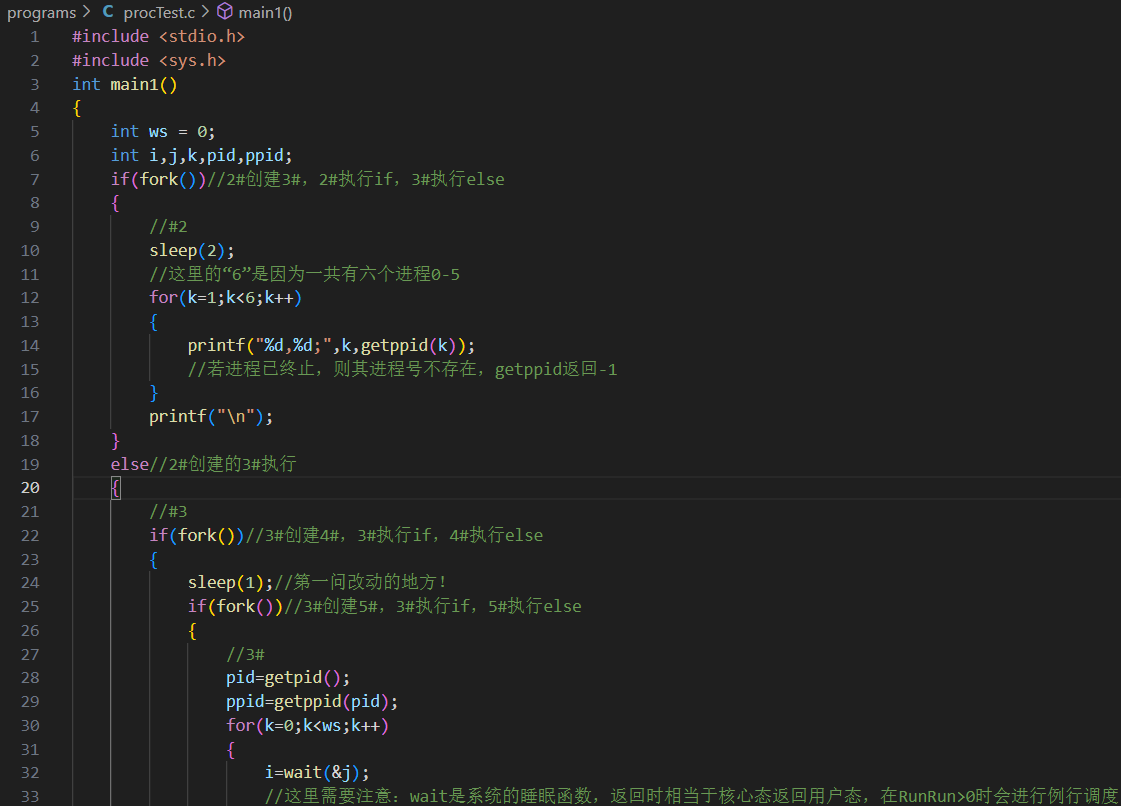
**4.4.2先将 3#进程执行代码部分的两次 wait 操作删除，再尝试两种修改代码的方案，创造 3#进程可以被抢占的机会，进而得到如图6和图7所示的执行结果。**

**1、情况一**

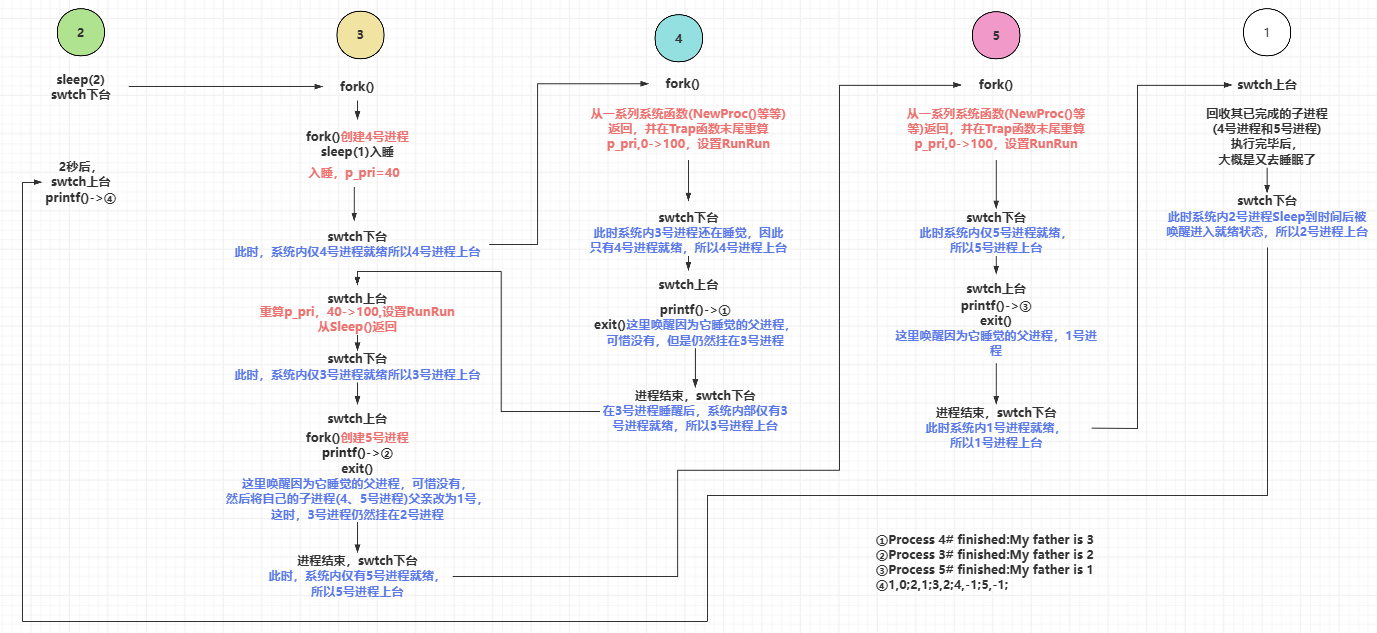


**（1）修改代码：**

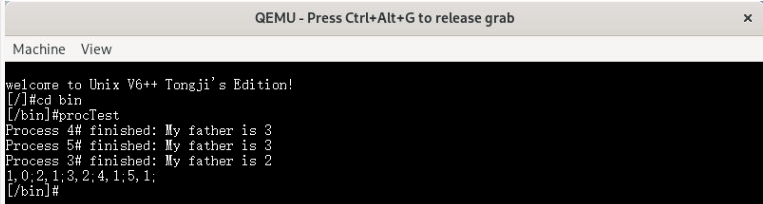
在3#创建4#、5#进程之间调用Sleep进而让其主动下台。



**（2）绘图解释**

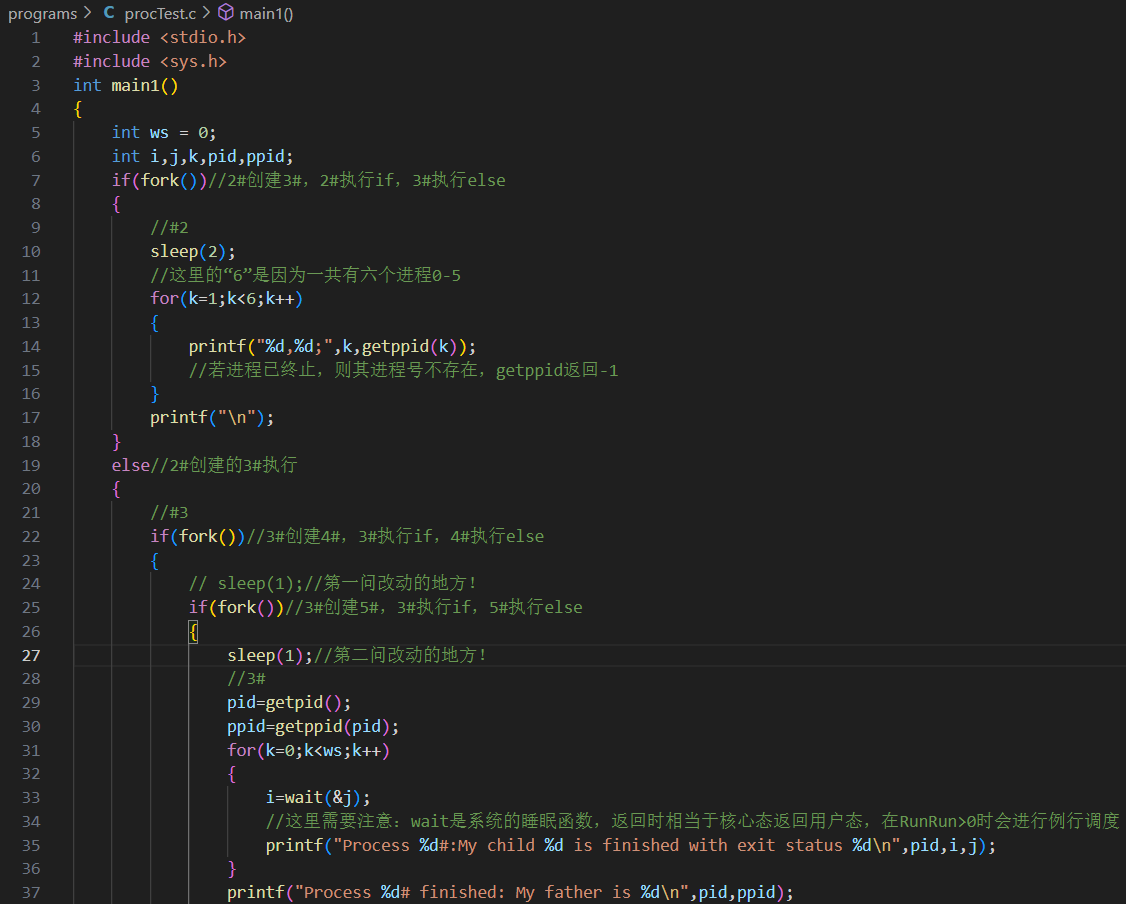


**情况二：**

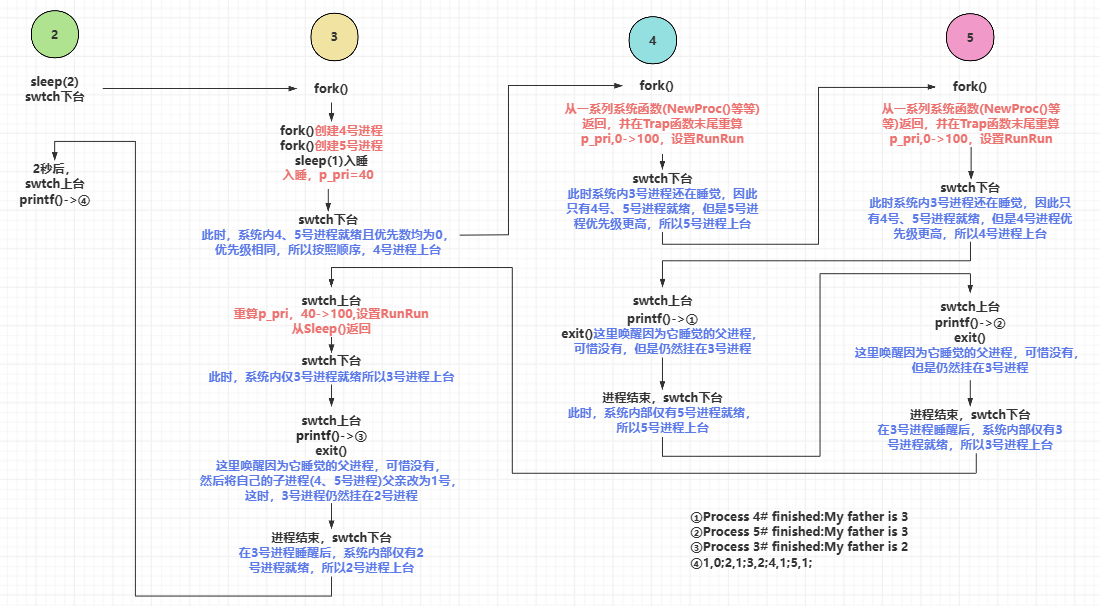


**（1）修改代码**

在3号进程创建完4、5号进程后调用Sleep进而让其主动下台。



**（2）绘图解释**



**（3）问题解答**

**观察图6（情况一）和图7（情况二）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？**

答：（1）图6（即情况1中），3#创建好4#后就入睡了，主动让出cpu。此时，系统内仅4#就绪，所以4#进程上台执行。4#在fork()的Trap末尾重算优先数，设置RunRun，在fork()返回用户态时例行调度，但此时系统内仍旧只有4#就绪，所以4#进程再次上台。4#再次上台后，打印输出，调用exit()删除内存、保存user到盘交换区。因没有唤醒因为它睡的父进程回收自己，所以结束时仍挂在3#上。之后3#睡醒上台创建好5#，打印输出，调用exit()删除内存、保存user到盘交换区、将子进程4#、5#交给1#。因为没有唤醒因为它睡的父进程回收自己，所以结束时仍挂在2#上。之后5#swtch上台，在fork()的Trap末尾重算优先数，设置RunRun，在fork()返回用户态时例行调度，但此时系统内仍旧只有5#就绪，所以5#进程再次上台。5#再次上台后，打印输出，调用exit()删除内存、保存user到盘交换区。但是唤醒了因为自己睡觉的1#，之后1#上台将其已经结束的进程4#、5#回收。综上，程序最后输出时，图6中4#和 5#已被撤销。

（2）图7（情况2）中，3#在创建完4#、5#进程后入睡。4#、5#进程先依次上台进行fork()返回，然后再依次上台打印输出，调用exit()删除内存、保存user到盘交换区，因为没唤醒因自己睡觉的父进程而结束时仍挂在3#上。之后，3#睡醒后上台，打印输出，调用exit删除内存、保存user到盘交换区、将其子进程(4#、5#)的父进程都设置成为了1#，因为没唤醒因自己睡觉的父进程而结束时仍挂在2#上。最后，2#上台打印信息，这时因为1#尚未上台，所以4#、5#作为其已经完成的子进程还没有被收回，进而导致图7中，4#和5#显示父进程为1#。