# 实验1 进程和进程通信实验报告

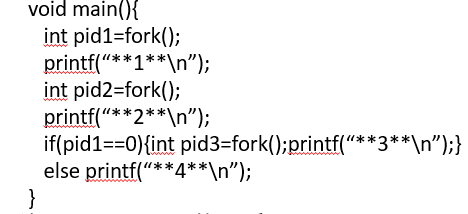
## 实验目的

1. 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
2. 了解信号处理
3. 认识进程间通信（IPC）：进程间共享内存
4. 实现shell：了解程序运行

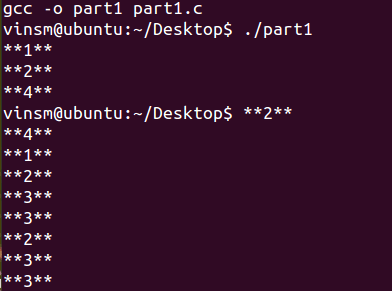
## 实验内容

1. 进程的创建实验

1-1.将下面的程序编译运行，并解释现象。



运行结果：



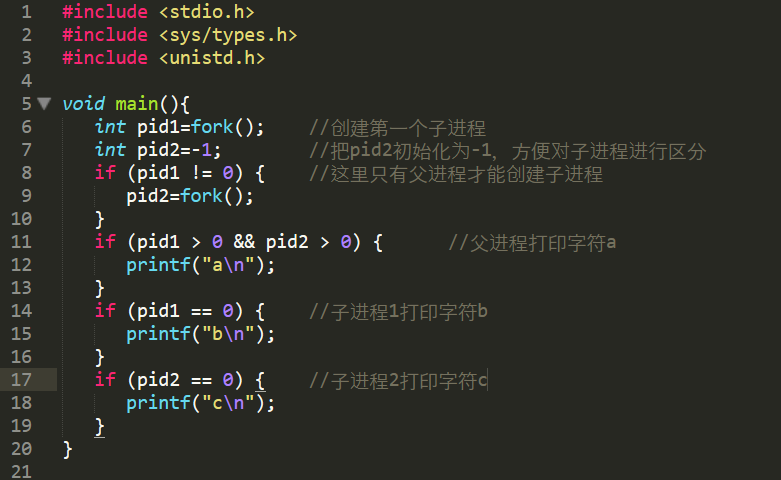
解释：

该程序创建的所有进程可用进程树表示：

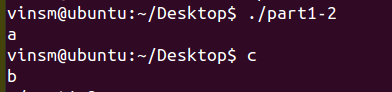
1是父进程，2、3是1的子进程，4、5是2的子进程，6是4的子进程，进程创建的先后顺序与序号对应。由代码可知，进程1输出124，进程2输出123，进程3输出24，进程4输出23，进程5输出3，进程6输出3。子进程被产生时，是复制父进程fork()语句前的变量值，并从fork()语句后开始执行的，这就解释了为什么有的进程输出3个数字而有的输出2个或1个，以及为什么子进程会进入由pid1的值控制的不同分支语句。因此由该运行结果可推测，进程执行顺序为1,3,2,5,4,6。

1-2.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。

程序代码为：



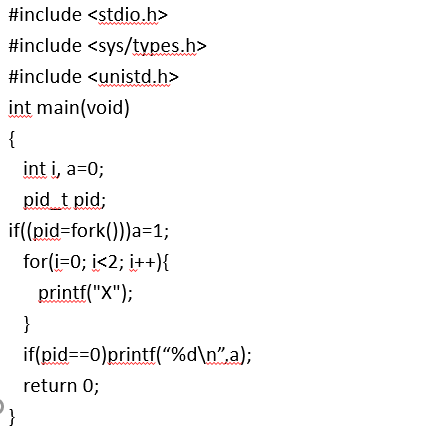
运行结果：



分析原因：

父进程调用fork()后创建子进程1，父进程的pid1 > 0，子进程1的pid1 = 0；往下执行，子进程1的pid2 = -1，父进程调用fork()后创建子进程2，父进程的pid2 > 0,子进程2的pid1 > 0， pid2 = 0。再往下执行，通过pid1、pid2值的不同区分出三个进程，从而打印不同的字符。由运行结果可推测，进程执行顺序为父进程、子进程2、子进程1。

1-3.下面程序将在屏幕上输出的字符‘X’、数字“1” 和“0”各多少个？为什么？

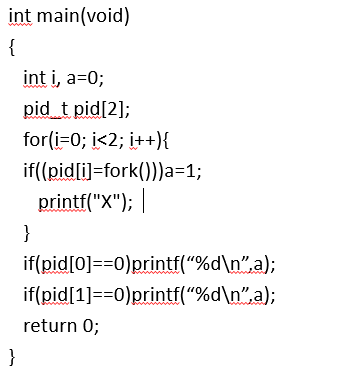


运行结果：

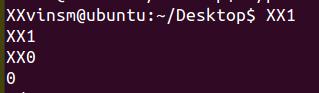


打印字符“X”4个，“1”0个，“0”1个。因为父进程进入第一个分支而子进程不能进入，所以对于父进程，a = 1，对于子进程，a = 0。父子进程均打印两个“X”，总共打印了4个“X”。父进程不进入第二个分支而子进程进入，所以打印a的值为0。

1-4.如果将上面main函数修改如下，则屏幕上输出的字符‘X’、数字“1”和“0”各多少个？为什么？

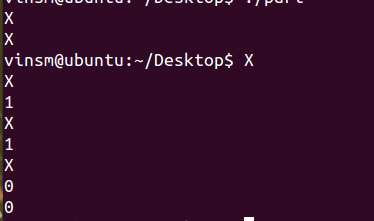


运行结果：



打印字符“X”8个，“1”2个，“0”2个。父进程创建了子进程1和2，子进程1创建了子进程1.1，在这个过程中，父进程的pid[0] > 0，pid[1] > 0, a = 1,打印“X”2次，不打印a；子进程1的pid[0] = 0， pid[1] > 0，a = 1, 打印“X”2次，打印“1”1次;子进程2的pid[0] > 0（与父进程相同）， pid[1] = 0，a = 1（与父进程相同）, 打印“X”1次，打印“1”1次;子进程1.1的pid[0] = 0（与父进程相同）， pid[1] = 0，a = 0, 打印“X”1次，打印“0”2次。

这样“X”理应打印6次，但这里打印了8次，查询后得知是打印时没有加“\n”，是与缓冲区相关的问题。加入“\n”后再进行打印，结果与预期相符，如下图。

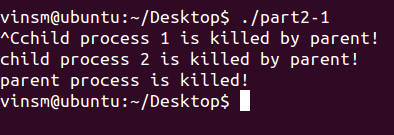


1. 信号处理实验

2-1.编制一段程序，使用系统调用fork()创建两个子程序，再用系统调用signal()

让父进程捕捉键盘上来的中断信号(即按Ctrl C键)，当捕捉到中断信号后，父进程调用kill()向两个子进程发出信号，子进程捕捉到信号后，分别输出下面信息后终止：  
child process 1 is killed by parent!  
child process 2 is killed by parent!  
父进程等待两个子进程终止后，输出以下信息后终止：  
parent process is killed!

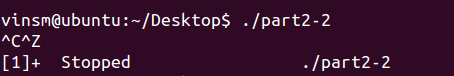
运行结果：



2-2.在上述程序中增加语句signal(SIGINT, SIG\_IGN)和signal(SIGQUIT,

SIG\_IGN)，观察执行结果并分析原因。这里signal(SIGINT, SIG\_IGN)和signal(SIGQUIT, SIG\_IGN)分别为忽略“Ctrl C”信号以及忽略“Ctrl Z”信号。

运行结果：

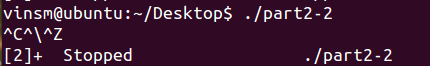


分析：

其中，“Ctrl C”信号被signal(SIGINT, SIG\_IGN)忽略，而“Ctrl Z”信号却没有被忽

略。把signal(SIGQUIT, SIG\_IGN)放在其他位置尝试多次也是同样结果，后来查阅资料后发现，“Ctrl Z”信号是将任务中止（暂停的意思）,但是此任务并没有结束,它仍然在进程中，只是维持挂起的状态,用户可以使用fg/bg操作继续前台或后台的任务,fg命令重新启动前台被中断的任务,bg命令把被中断的任务放在后台执行。

与“Ctrl Z”信号对应的为SIGSTOP，而该信号不能被阻塞，处理或忽略。signal(SIGQUIT, SIG\_IGN)真正忽略的是“Ctrl \”，它和SIGINT类似，但由QUIT字符(通常是Ctrl \)来控制。进程在因收到SIGQUIT退出时会产生core文件，在这个意义上类似于一个程序错误信号。因此，正确的运行结果应该为“Ctrl C”和“Ctrl \”被忽略：



1. 进程间共享内存实验

完成课本第三章的练习3.10的程序。

程序代码：

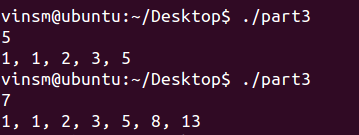


分析：

按照书本要求，为程序定义了斐波那契数列最大长度MAX\_SEQUENCE为8，分配共享内存

空间大小为share\_data，从命令行接受参数num为请求的斐波那契数列的长度，该长度需在2~7之间。接下来分配共享内存空间，指定错误处理，并把内存指针附加到p\_addr和c\_addr中。创建子进程，把对应的斐波那契数列写入共享内存段，父进程等待子进程结束后输出共享内存段中的数据，最后共享内存段detach，结束程序。

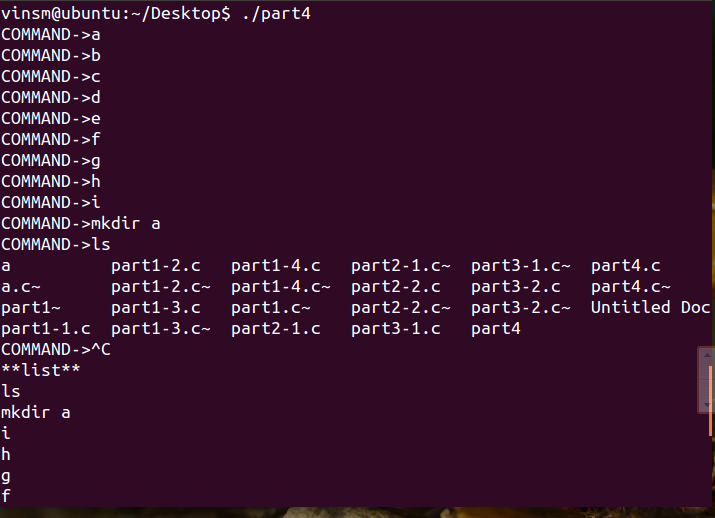
运行结果：

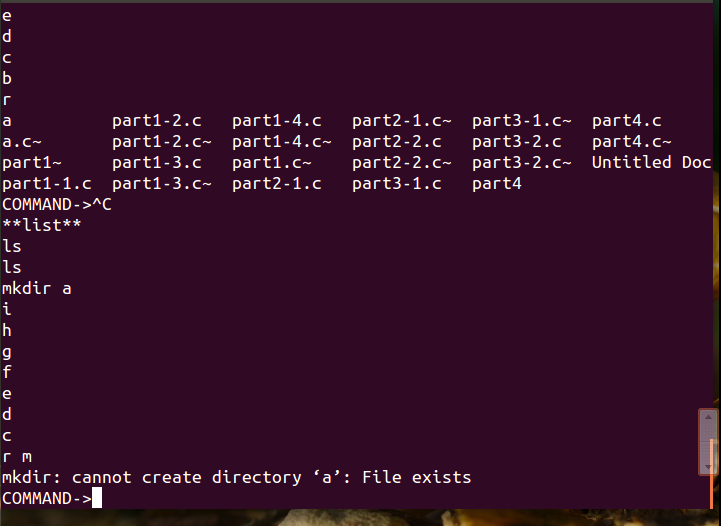


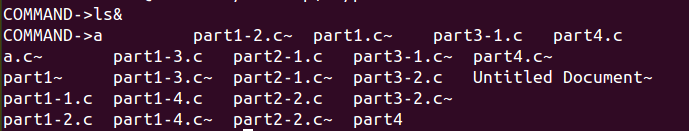
1. 实现shell的要求

完成课本上第三章的项目：实现shell。除此之外满足下面要求：在shell下，按ctrl+C时不会终止shell。

运行结果：









分析：

图1、2实现了创建历史特性，通过维护一个大小为10的队列，把用过的命令保存在其中。用sigaction和SIGINT\_HANDLER函数捕获Ctrl+C信号后，打印队列中的命令。其后，用户输入“r”或“r x”后，分别执行最近执行的一条命令或最近执行的以“x”开头的命令。在这里把“r”和“r x”作为命令进行特殊处理（不会保存在队列中），相当于先进行字符串匹配再执行一次setup函数的功能。

图3体现了创建子进程并放在后台执行，父进程需要等待子进程退出。图4体现了Ctrl+D退出简单Shell的功能。

程序代码：

