fork,source,exec

1. exec和shell都属于内部命令，外部命令是通过系统调用或独立的程序实现的，如sed，awk等；内部命令是由特殊的文件格式(.def)所实现的，如cd/history/exec等
2. fork是linux系统调用，用来创建子进程(child process)，子进程是父进程(parent process)的一个副本，从父进程那里获得一定的资源分配以及继承父进程的环境。子进程和父进程唯一不同的地方在于pid(process id)。环境变量(传给子进程的变量)只能单向从父进程传给子进程。不管子进程的环境变量如何变化，都不会影响父进程的环境变量。
3. 有两种方法执行shell scripts，一种是新产生一个shell，然后执行在其下执行命令，执行完child，会返回parent；另一种是通过source命令，不再产生新的shell(sub-shell)，而在当前shell下执行一切命令。产生新的shell然后再执行的方法是在scrits文件开头加入以下语句：#!/bin/sh；source命令即点(.)命令。
4. exec命令在执行时会把当前的shell process关闭，然后换到后面的命令继续执行，系统调用exec是以新的进程去代替原来的进程，但进程的PID保持不变，运行完毕之后不会回到原先的程序中去。因此，可以这样认为，exec系统调用并没有创建新的进程，只是替换了原来进程上下文的内容。原进程的代码段，数据段，堆栈段被新的进程所代替。
5. I/O重定向通常与FD(file discriptor)有关，shell的FD通常位10个，即0～9，常用的FD有3个，为0(stdin, 标准输入), 1(stdout, 标准输出), 2(stderr, 标准错误输出)；&-关闭标准输出，n&-表示将n号输出关闭
6. 其他，date日期函数，date <+事件日期格式>; date +"%Y-%m-%d", 2019-03-18。wait，wait [n]，表示当前shell中某个执行的后台命令的pid，wait命令会等待该后台进程执行完毕才允许下一个shell语句执行；如果没指定则代表当前shell后台执行的语句，wait会等待到所有的后台程序执行完毕为止。

---

shell实现多线程

1. 实例一：全前台进程：

```
#!/bin/bash
#filename:simple.sh
starttime=$(date +%s)
for ((i=0;i<5;i++));do
        {
                sleep 3;echo 1>>aa && endtime=$(date +%s) && echo "我是$i,
运行了3秒,整个脚本执行了$(expr $endtime - $starttime)秒"
        }
done
cat aa|wc -l
rm aa
```

2. 实例二：使用'&'+wait 实现“多进程”实现

```
#!/bin/bash
#filename:multithreading.sh
starttime=$(date +%s)
for ((i=0;i<5;i++));do
        {
                sleep 3;echo 1>>aa && endtime=$(date +%s) && echo "我是$i,
运行了3秒,整个脚本执行了$(expr $endtime - $starttime)秒"
        }&
done
wait
cat aa|wc -l
rm aa
```

3. 实例三：可控制后台进程数量的“多进程”shell

   尚不能理解，😢😢😢😢

   https://blog.csdn.net/dubendi/article/details/78919322

要搞清楚fork的执行过程, 就必须先清楚操作系统中的“进程(process)“概念。一个进程包含三个元素:

- 一个可以执行的程序
- 和该进程相关联的全部数据(包括变量，内存空间，缓冲区等等)
- 程序的执行上下文(execution context)

不妨简单理解为，一个进程表示，就是一个可执行程序的一次执行过程中的一个状态。操作系统对进程的管理，典型的情况，是通过进程表完成的。进程表中的每一个表项，记录的是当前操作系统中一个进程的情况。对于单CPU的情况而言，每一特定时刻只有一个进程占用CPU，但是系统中可能同时存在多个活动的(等待执行或继续执行的)进程。

一个称为"程序计数器(program counter, pc)“的寄存器，指出当前占用CPU的进程要执行的下一条指令的位置。

当分给某个进程的CPU时间已经用完，操作系统将该进程相关的寄存器的值，保存到该进程在进程表中对应的表项里面；把将要接替这个进程占用CPU的那个进程的上下文，从进程表中读出，并更新相应的寄存器(这个过程称为"上下文交换(process context switch)“，实际的上下文交换需要涉及到更多的数据，那和fork无关，不再多说，主要记住程序寄存器pc指出程序当前已经执行到哪里，是进程上下文的重要内容，换出CPU的进程要保存这个寄存器的值，换入CPU的进程，也要根据进程表中保存的本进程执行上下文信息，更新这个寄存器)。

fork, 当你的程序执行到下面的语句:

pid=fork();

操作系统创建一个新的进程(子进程)，并且在进程表中相应为它建立一个新的表项。新进程和原有进程的可执行程序是同一个程序；上下文和数据，绝大部分就是原进程(父进程)的拷贝，但它们是两个相互独立的进程！此时程序寄存器pc，在父，子进程的上下文中都声称，这个进程目前执行到fork调用即将返回(此时子进程不占有CPU，子进程的pc不是真正保存在寄存器中，而是作为进程上下文保存在进程表中的对应表项内）。问题是怎么返回，在父子进程中就分道扬镳。

父进程继续执行，操作系统对fork的实现，使这个调用在父进程中返回刚刚创建的子进程的pid(一个正整数)，所以下面的if语句中的pid<0，pid=0的两个分支都不会执行。所以输出：I am the parent process...

子进程在之后的某个时候得到调度，它的上下文被换入，占据CPU，操作系统对fork的实现，使得子进程中fork调用返回0。所以在这个进程(注意这不是父进程了哦，虽然是同一程序，但是这是同一个程序的另一次执行，在操作系统中这次执行是由另一个进程表示的，从执行的角度说和父进程相互独立)中pid=0。这个进程继续执行的过程中，if语句中pid<0不满足，但是pid==0时true。所以输出I am the child process...

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main () {
pid_t childpid= 0;
int i, n=4;

for (i= 1; i< n; i++)
if (childpid= fork()) break;
wait();
printf("i:%d process ID:%ld parent ID:%ld child ID:%ld\n", i, (long)getpid(),
(long)getppid(), (long)childpid); return 0;
}
```