Linux下的多进程编程 PDF转换可能丢失图片或格式，建议阅读原文

https://www.100test.com/kao_ti2020/181/2021_2022_Linux_E4_B8_8B_E7_9A_c103_181687.htm 什么是一个进程？进程这个概念是针对系统而不是针对用户的，对用户来说，他面对的概念是程序。当用户敲入命令执行一个程序的时候，对系统而言，它将启动一个进程。但和程序不同的是，在这个进程中，系统可能需要再启动一个或多个进程来完成独立的多个任务。多进程编程的主要内容包括进程控制和进程间通信，在了解这些之前，我们先要简单知道进程的结构。 2.1 Linux下进程的结构 Linux下一个进程在内存里有三部分的数据，就是"代码段"、"堆栈段"和"数据段".其实学过汇编语言的人一定知道，一般的CPU都有上述三种段寄存器，以方便操作系统的运行。这三个部分也是构成一个完整的执行序列的必要的部分。 "代码段"，顾名思义，就是存放了程序代码的数据，假如机器中有数个进程运行相同的一个程序，那么它们就可以使用相同的代码段。"堆栈段"存放的就是子程序的返回地址、子程序的参数以及程序的局部变量。而数据段则存放程序的全局变量，常数以及动态数据分配的数据空间（比如用malloc之类的函数取得的空间）。这其中有许多细节问题，这里限于篇幅就不多介绍了。系统如果同时运行数个相同的程序，它们之间就不能使用同一个堆栈段和数据段。 2.2 Linux下的进程控制 在传统的Unix环境下，有两个基本的操作用于创建和修改进程：函数fork（）用来创建一个新的进程，该进程几乎是当前进程的一个完全拷贝；函数族exec（）用来启动另外的进程以取代当前运行的进程。Linux的进程控

制和传统的Unix进程控制基本一致，只在一些细节的地方有些区别，例如在Linux系统中调用vfork和fork完全相同，而在有些版本的Unix系统中，vfork调用有不同的功能。由于这些差别几乎不影响我们大多数的编程，在这里我们不予考虑。2.2.1 fork（） fork在英文中是"分叉"的意思。为什么取这个名字呢？因为一个进程在运行中，如果使用了fork，就产生了另一个进程，于是进程就"分叉"了，所以这个名字取得很形象。下面就看看如何具体使用fork，这段程序演示了使用fork的基本框架：void main(){int i.if (fork() == 0) {/* 子进程程序*/for (i = 1. i }else {/* 父进程程序*/for (i = 1. i }} 程序运行后，你就能看到屏幕上交替出现子进程与父进程各打印出的一千条信息了。如果程序还在运行中，你用ps命令就能看到系统中有两个它在运行了。 那么调用这个fork函数时发生了什么呢？fork函数启动一个新的进程，前面我们说过，这个进程几乎是当前进程的一个拷贝：子进程和父进程使用相同的代码段；子进程复制父进程的堆栈段和数据段。这样，父进程的所有数据都可以留给子进程，但是，子进程一旦开始运行，虽然它继承了父进程的一切数据，但实际上数据却已经分开，相互之间不再有影响了，也就是说，它们之间不再共享任何数据了。它们再要交互信息时，只有通过进程间通信来实现，这将是我们下面的内容。既然它们如此相象，系统如何来区分它们呢？这是由函数的返回值来决定的。对于父进程，fork函数返回了子程序的进程号，而对于子程序，fork函数则返回零。在操作系统中，我们用ps函数就可以看到不同的进程号，对父进程而言，它的进程号是由比它更低层的系统调用赋予的，而对于子进程而言，它的进程号即是fork函数对

父进程的返回值。在程序设计中，父进程和子进程都要调用函数fork（）下面的代码，而我们就是利用fork（）函数对父子进程的不同返回值用if……else……语句来实现让父子进程完成不同的功能，正如我们上面举的例子一样。我们看到，上面例子执行时两条信息是交互无规则的打印出来的，这是父子进程独立执行的结果，虽然我们的代码似乎和串行的代码没有什么区别。 读者也许会问，如果一个大程序在运行中，它的数据段和堆栈都很大，一次fork就要复制一次，那么fork的系统开销不是很大吗？其实UNIX自有其解决的办法，大家知道，一般CPU都是以"页"为单位来分配内存空间的，每一个页都是实际物理内存的一个映像，象INTEL的CPU，其一页在通常情况下是4086字节大小，而无论是数据段还是堆栈段都是由许多"页"构成的，fork函数复制这两个段，只是"逻辑"上的，并非"物理"上的，也就是说，实际执行fork时，物理空间上两个进程的数据段和堆栈段都还是共享着的，当有一个进程写了某个数据时，这时两个进程之间的数据才有了区别，系统就将有区别的"页"从物理上也分开。系统在空间上的开销就可以达到最小。 下面演示一个足以"搞死"Linux的小程序，其源代码非常简单： void main() { for(. .) fork(). } 这个程序什么也不做，就是死循环地fork，其结果是程序不断产生进程，而这些进程又不断产生新的进程，很快，系统的进程就满了，系统就被这么多不断产生的进程"撑死了"。当然只要系统管理员预先给每个用户设置可运行的最大进程数，这个恶意的程序就完成不了企图了。 2.2.2 exec( )函数族 下面我们来看看一个进程如何来启动另一个程序的执行。在Linux中要使用exec函数族。系统调用execve（）对当前进

程进行替换，替换者为一个指定的程序，其参数包括文件名（filename）、参数列表（argv）以及环境变量（envp）。exec函数族当然不止一个，但它们大致相同，在Linux中，它们分别是：execl，execlp，execle，execv，execve和execvp，下面我只以execlp为例，其它函数究竟与execlp有何区别，请通过manexec命令来了解它们的具体情况。 一个进程一旦调用exec类函数，它本身就"死亡"了，系统把代码段替换成新的程序的代码，废弃原有的数据段和堆栈段，并为新程序分配新的数据段与堆栈段，唯一留下的，就是进程号，也就是说，对系统而言，还是同一个进程，不过已经是另一个程序了。（不过exec类函数中有的还允许继承环境变量之类的信息。） 那么如果我的程序想启动另一程序的执行但自己仍想继续运行的话，怎么办呢？那就是结合fork与exec的使用。下面一段代码显示如何启动运行其它程序：char command[256].void main(){int rtn. /*子进程的返回数值*/while(1) {/* 从终端读取要执行的命令 */printf(">").fgets(command, 256, stdin).command[strlen(command)-1] = 0.if (fork() == 0) {/* 子进程执行此命令 */execlp(command, command)./* 如果exec函数返回，表明没有正常执行命令，打印错误信息*/perror(command).exit(errorno).}else {/* 父进程， 等待子进程结束，并打印子进程的返回值 */wait (&amp.rtn).printf(" child process return %d\n",. rtn).}}} 此程序从终端读入命令并执行之，执行完成后，父进程继续等待从终端读入命令。熟悉DOS和WINDOWS系统调用的朋友一定知道DOS/WINDOWS也有exec类函数，其使用方法是类似的，但DOS/WINDOWS还有spawn类函数，因为DOS是单任务的

系统，它只能将"父进程"驻留在机器内再执行"子进程"，这就是spawn类的函数。WIN32已经是多任务的系统了，但还保留了spawn类函数，WIN32中实现spawn函数的方法同前述UNIX中的方法差不多，开设子进程后父进程等待子进程结束后才继续运行。UNIX在其一开始就是多任务的系统，所以从核心角度上讲不需要spawn类函数。 在这一节里，我们还要讲讲system（）和popen（）函数。system（）函数先调用fork（），然后再调用exec（）来执行用户的登录shell，通过它来查找可执行文件的命令并分析参数，最后它么使用wait（）函数族之一来等待子进程的结束。函数popen（）和函数system（）相似，不同的是它调用pipe（）函数创建一个管道，通过它来完成程序的标准输入和标准输出。这两个函数是为那些不太勤快的程序员设计的，在效率和安全方面都有相当的缺陷，在可能的情况下，应该尽量避免。