1.8 fork系统调用

下一个我想查看的例子叫做fork。fork会创建一个新的进程，下面是使用fork的一个简单用例。

Terminal

```
1
2  // fork.c: create a new process
3
4  #include "kernel/types.h"
5  #include "user/user.h"
6
7  int
8  main()
9  {
10   int pid;
11
12   pid = fork();
13
14   printf("fork() returned %d\n", pid);
15
16   if(pid == 0){
17     printf("child\n");
18   } else {
19     printf("parent\n");
20   }
21
22   exit(0);
23 }
[crash]
```

在第12行，我们调用了fork。fork会拷贝当前进程的内存，并创建一个新的进程，这里的内存包含了进程的指令和数据。之后，我们就有了两个拥有完全一样内存的进程。fork系统调用在两个进程中都会返回，在原始的进程中，fork系统调用会返回大于0的整数，这个是新创建进程的ID。而在新创建的进程中，fork系统调用会返回0。所以即使两个进程的内存是完全一样的，我们还是可以通过fork的返回值区分旧进程和新进程。

在第16行，你可以看到代码检查pid。如果pid等于0，那么这必然是子进程。在我们的例子中，调用进程通常称为父进程，父进程看到的pid必然大于0。所以父进程会打印“parent”，子进程会打印“child”。之后两个进程都会退出。接下来我运行这个程序：

```
$ fork
ffoorrkk(())  rreettuurrnende d 0
1c9h
ilpda
rent
```

输出看起来像是垃圾数据。这里实际发生的是，fork系统调用之后，两个进程都在同时运行，QEMU实际上是在模拟多核处理器，所以这两个进程实际上就是同时在运行。所以当这两个进程在输出的时候，它们会同时一个字节一个字节的输出，两个进程的输出交织在一起，所以你可以看到两个f，两个o等等。在第一行最后，你可以看到0，这是子进程的输出。我猜父进程返回了19，作为子进程的进程ID。通常来说，这意味着这是操作系统启动之后的第19个进程。之后一个进程输出了child，一个进程输出了parent，这两个输出交织在一起。虽然这只是对于fork的一个简单应用，但是我们可以清晰的从输出看到这里创建了两个运行的进程，其中一个进程打印了child，另一个打印了parent。所以，fork（在子父进程中）返回不同的值是比较重要的。

> 学生提问：fork产生的子进程是不是总是与父进程是一样的？它们有可能不一样吗？
>
> Robert教授：在XV6中，除了fork的返回值，两个进程是一样的。两个进程的指令是一样的，数据是一样的，栈是一样的，同时，两个进程又有各自独立的地址空间，它们都认为自己的内存从0开始增长，但这里是不同的内存。 在一个更加复杂的操作系统，有一些细节我们现在并不关心，这些细节偶尔会导致父子进程不一致，但是在XV6中，父子进程除了fork的返回值，其他都是一样的。除了内存是一样的以外，文件描述符的表单也从父进程拷贝到子进程。所以如果父进程打开了一个文件，子进程可以看到同一个文件描述符，尽管子进程看到的是一个文件描述符的表单的拷贝。除了拷贝内存以外，fork还会拷贝文件描述符表单这一点还挺重要的，我们接下来会看到。

fork创建了一个新的进程。当我们在Shell中运行东西的时候，Shell实际上会创建一个新的进程来运行你输入的每一个指令。所以，当我输入ls时，我们需要Shell通过fork创建一个进程来运行ls，这里需要某种方式来让这个新的进程来运行ls程序中的指令，加载名为ls的文件中的指令（也就是后面的exec系统调用）。