但是，父进程和子进程可以共享已经打开的文件。也就是说，如果某一个文件在父进程调用fork函数之前就已经打开了，那么在父进程调用fork函数之后，对于父进程和子进程来说，这个文件也是打开的。如果父、子进程中任何一个进程对这个文件进行了修改，那么对于另一个进程而言，这些修改都是可见的。由于这些修改对于那些打开了这个文件的其他任何无关进程来说也是可见的，所以，在父、子进程间共享已经打开的文件以及对文件的修改彼此可见的做法也是很正常的。

事实上，父、子进程的内存映像、变量、寄存器以及其他所有的东西都是相同的，这就产生了一个问题：该如何区别这两个进程，即哪一个进程该去执行父进程的代码，哪一个进程该去执行子进程的代码呢？秘密在于fork系统调用给子进程返回一个零值，而给父进程返回一个非零值。这个非零值是子进程的进程标识符（Process Identifier，PID）。两个进程检验fork函数的返回值，并且根据返回值继续执行，如图10-4所示。

```
pid = fork( );                  /*如果创建成功，则父进程pid>0*/
if (pid < 0) {
    handle_error( );            /*创建失败（比如内存或某些表溢出）*/
} else if (pid > 0) {
                                /*这里是父进程的代码*/
} else {
                                /*这里是子进程的代码*/
}
```

图10-4 Linux中的进程创建

进程以其PID来命名。如前所述，当一个进程被创建的时候，它的父进程会得到它的PID。如果子进程希望知道它自己的PID，可以调用系统调用getpid。PID有很多用处，举个例子来说，当一个子进程结束的时候，它的父进程会得到该子进程的PID。这一点非常重要，因为一个父进程可能会有多个子进程。由于子进程还可以生成子进程，那么一个原始进程可以生成一个进程树，其中包含着子进程、孙子进程以及关系更疏远的后裔进程。

Linux系统中的进程可以通过一种消息传递的方式进行通信。在两个进程之间，可以建立一个通道，一个进程向这个通道里写入字节流，另一个进程从这个通道中读取字节流。这些通道称为管道（pipe）。使用管道也可以实现同步，因为如果一个进程试图从一个空的管道中读取数据，这个进程就会被挂起直到管道中有可用的数据为止。

shell中的管线就是用管道技术实现的。当shell看到类似下面的一行输入时：

```
sort <f | head
```

它会创建两个进程，分别是sort和head，同时在两个进程间建立一个管道使得sort进程的标准输出作为head进程的标准输入。这样一来，sort进程产生的输出可以直接作为head进程的输入而不必写入到一个文件当中去。如果管道满了，系统会停止运行sort进程直到head进程从管道中删除一些数据。

进程还可以通过另一种方式通信：软件中断。一个进程可以给另一个进程发送信号（signal）。进程可以告诉操作系统当信号到来时它们希望发生什么事件。相关的选择有忽略这个信号、抓取这个信号或者利用这个信号杀死某个进程（大部分情况下，这是处理信号的默认方式）。如果一个进程希望获取所有发送给它的信号，它就必须指定一个信号处理函数。当信号到达时，控制立即切换到信号处理函数。当信号处理函数结束并返回之后，控制像硬件I/O中断一样返回到陷入点处。一个进程只可以给它所在进程组中的其他进程发送信号，这个进程组包括它的父进程（以及远祖进程）、兄弟进程和子进程（以及后裔进程）。同时，一个进程可以利用系统调用给它所在的进程组中所有的成员发送信号。

信号还可以用于其他用途。比如说，如果一个进程正在进行浮点运算，但是不慎除数为0，它就会得到一个SIGFPE信号（浮点运算异常信号）。POSIX系统定义的信号详见图10-5所示。很多Linux

信　号	原　因
SIGABRT	进程中止且强迫核心转储
SIGALRM	警报时钟超时
SIGFPE	出现浮点错误（比如，除0）
SIGHUP	进程所使用的电话线被挂断
SIGILL	用户按了DEL键中断了进程
SIGQUIT	用户按键要求核心转储
SIGKILL	杀死进程（不能被捕捉或忽略）
SIGPIPE	进程写入了无读者的管道
SIGSEGV	进程引用了非法的内存地址
SIGTERM	用于要求进程正常终止
SIGUSR1	用于应用程序定义的目的
SIGUSR2	用于应用程序定义的目的

图10-5 POSIX定义的信号