提供的一套过程，但是并没有规定它们是系统调用，是库调用还是其他的形式。如果不通过系统调用就可以执行一个过程（即无须陷入内核），那么从性能方面考虑，它通常会在用户空间中完成。不过，多数POSIX过程确实进行系统调用，通常是一个过程直接映射到一个系统调用上。在有一些情形下，特别是所需要的过程仅仅是某个调用的变体时，此时一个系统调用会对应若干个库调用。

1.6.1 用于进程管理的系统调用

在图1-18中的第一组调用处理进程管理。将有关fork（派生）的讨论作为本节的开始是较为合适的。在UNIX中，fork是惟一可以在POSIX创建进程的途径。它创建一个原有进程的精确副本，包括所有的文件描述符，寄存器等全部内容。在fork之后，原有的进程及其副本（父与子）就分开了。在fork时，所有的变量具有一样的值，虽然父进程的数据被复制用以创建子进程，但是其中一个的后续变化并不会影响到另一个。（由父进程和子进程共享的程序正文，是不可改变的。）fork调用返回一个值，在子进程中该值为零，并且等于子进程的进程标识符，或等于父进程中的PID。使用被返回的PID，就可以在两个进程中看出哪一个是父进程，哪一个是子进程。

多数情形下，在fork之后，子进程需要执行与父进程不同的代码。这里考虑shell的情形。它从终端读取命令，创建一个子进程，等待该子进程执行命令，在该子进程终止时，读入下一条命令。为了等待子进程结束，父进程执行一个waitpid系统调用，它只是等待，直至子进程终止（若有多个子进程存在的话，则直至任何一个子进程终止）。waitpid可以等待一个特定的子进程，或者通过将第一个参数设为-1的方式，从而等待任何一个老的子进程。在waitpid完成之后，将把第二个参数statloc所指向的地址设置为子进程的退出状态（正常或异常终止以及退出值）。有各种可使用的选项，它们由第三个参数确定。

现在考虑shell如何使用fork。在键入一条命令后，shell创建一个新的进程。这个子进程必须执行用户的命令。通过使用execve系统调用可以实现这一点，这个系统调用会引起其整个核心映像被一个文件所替代，该文件由第一个参数给定。（实际上，该系统调用自身是exec系统调用，但是若干个不同的库过程使用不同的参数和稍有差别的名称调用该系统调用。在这里，我们都把它们视为系统调用。）在图1-19中，用一个高度简化的shell说明fork、waitpid以及execve的使用。

```
#define TRUE 1

while (TRUE) {                                          /* 一直循环下去 */
    type_prompt( );                                     /* 在屏幕上显示提示符 */
    read_command(command, parameters);                  /* 从终端读取输入 */

    if (fork( ) != 0) {                                 /* 派生子进程 */
        /* 父代码 */
        waitpid(-1, &status, 0);                        /* 等待子进程退出 */
    } else {
        /* 子代码 */
        execve(command, parameters, 0);                 /* 执行命令 */
    }
}
```

图1-19 一条shell（在本书中，TRUE都被定义为1）

在最一般情形下，execve有三个参数：将要执行的文件名称，一个指向变量数组的指针，以及一个指向环境数组的指针。这里对这些参数做一个简要的说明。各种库例程，包括execl、execv、execle以及execve，可以允许略掉参数或以各种不同的方式给定。在本书中，我们在所有涉及的地方使用exec描述系统调用。

下面考虑诸如

```
cp file1 file2
```

的命令，该命令将file1 复制到 file2。在shell 创建进程之后，该子进程定位和执行文件cp，并将源文件名和目标文件名传递给它。

cp主程序（以及多数其他C程序的主程序）都有声明