之后，事情开始变化。这个磁盘占据4平方米空间，可以存储500万7位长的字符，这足够存储一张中等分辨率的数字照片。但是其年租金高达35 000美元，比存储占据同样空间数量的胶卷还要贵。不过这个磁盘的价格终于还是下降了，并开始出现了原始的文件系统。

拥有这些新技术的典型机器是CDC 6600，该机器于1964年发布，在多年之内始终是世界上最快的计算机。用户可以通过指定名称的方式创建所谓“永久文件”，希望这个名称还没有被别人使用，比如“data”就是一个适合于文件的名称。这个系统使用单层目录。后来在大型机上开发出了复杂的多层文件系统，MULTICS文件系统可以算是多层文件系统的顶峰。

接着小型计算机投入使用，该机型最后也有了硬盘。1970年在PDP-11上引入了标准硬盘——RK05磁盘，容量为2.5MB，只有IBM RAMAC一半的容量，但是这个磁盘的直径只有40厘米，5厘米厚。不过，其原型也只有单层目录。随着微型计算机的出现，CP/M开始成为操作系统的主流，但是它也只是在（软）盘上支持单目录。

4. 虚拟内存

虚拟内存（安排在第3章中讨论）通过在RAM和磁盘中反复移动信息块的方式，提供了运行比机器物理内存大的程序的能力。虚拟内存也经历了类似的历程，首先出现在大型机上，然后是小型机和微型机。虚拟内存还使得程序可以在运行时动态地链接库，而不是必须在编译时链接。MULTICS是第一个可以做到这点的系统。最终，这个思想传播到所有的机型上，现在广泛用于多数UNIX和Windows系统中。

在所有这些发展过程中，我们看到，在一种环境中出现的思想，随着环境的变化被抛弃（汇编语言设计、单道程序处理、单层目录等），通常在十年之后，该思想在另一种环境下又重现了。由于这个原因，本书中，我们将不时回顾那些在今日的吉字节PC中过时的思想和算法，因为这些思想和算法可能会在嵌入式计算机和智能卡中再现。

1.6 系统调用

我们已经看到操作系统具有两种功能：为用户程序提供抽象和管理计算机资源。在多数情形下，用户程序和操作系统之间的交互处理的是前者，例如，创建、写入、读出和删除文件。对用户而言，资源管理部分主要是透明和自动完成的。这样，用户程序和操作系统之间的交互主要就是处理抽象。为了真正理解操作系统的行为，我们必须仔细地分析这个接口。接口中所提供的调用随着操作系统的不同而变化（尽管基于的概念是类似的）。

这样我们不得不在如下的可能方式中进行选择：（1）含混不清的一般性叙述（“操作系统提供读取文件的系统调用”）；（2）某个特定的系统（“UNIX提供一个有三个参数的read系统调用：一个参数指定文件，一个说明数据应存放的位置，另一个说明应读出多少字节”）。

我们选择后一种方式。这种方式需要更多的努力，但是它能更多地洞察操作系统具体在做什么。尽管这样的讨论会涉及专门的POSIX（International Standard 9945-1），以及UNIX、System V、BSD、Linux、MINIX 3等，但是多数现代操作系统都有实现相同功能的系统调用，尽管它们在细节上差别很大。由于引发系统调用的实际机制是非常依赖于机器的，而且必须用汇编代码表达，所以，通过提供过程库使C程序中能够使用系统调用，当然也包括其他语言。

记住下列事项是有益的。任何单CPU计算机一次只能执行一条指令。如果一个进程正在用户态运行一个用户程序，并且需要一个系统服务，比如从一个文件读数据，那么它就必须执行一个陷阱或系统调用指令，将控制转移到操作系统。操作系统接着通过参数检查找出所需要的调用进程。然后，它执行系统调用，并把控制返回给在系统调用后面跟随着的指令。在某种意义上，进行系统调用就像进行一个特殊的过程调用，但是只有系统调用可以进入内核，而过程调用则不能。

为了使系统调用机制更清晰，我们简要地考察read系统调用。如上所述，它有三个参数：第一个参数指定文件，第二个指向缓冲区，第三个说明要读出的字节数。几乎与所有的系统调用一样，它的调用由C程序完成，方法是调用一个与该系统调用名称相同的库过程：read。由C程序进行的调用形式如下：

```
count = read(fd, buffer, nbytes);
```

系统调用（以及库过程）在count中返回实际读出的字节数。这个值通常和nbytes相同，但也可能更小，

例如，如果在读过程中遇到了文件尾的情形就是如此。

如果系统调用不能执行，不论是因为无效的参数还是磁盘错误，count都会被置为−1，而在全局变量errno中放入错误号。程序应该经常检查系统调用的结果，以了解是否出错。

系统调用是通过一系列的步骤实现的。为了更清楚地说明这个概念，考察上面的read调用。在准备调用这个实际用来进行read系统调用的read库过程时，调用程序首先把参数压进堆栈，如图1-17中步骤1～步骤3所示。

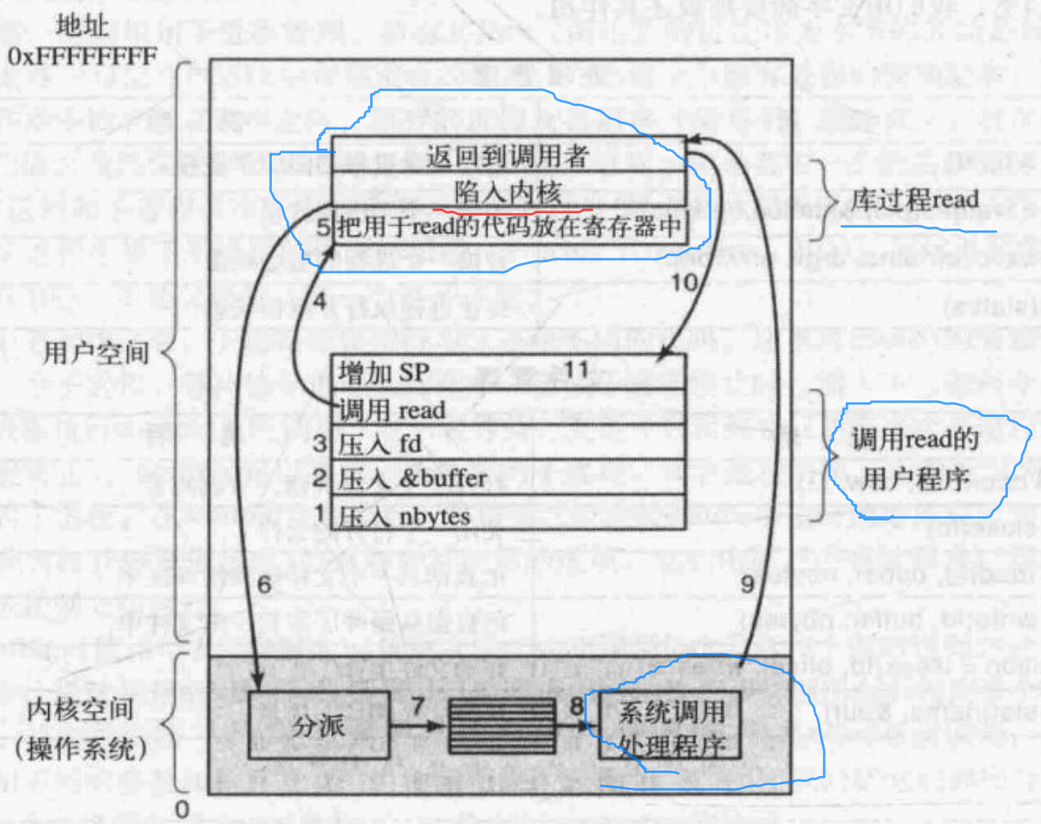

图1-17 完成系统调用read(fd, buffer, nbytes)的11个步骤

由于历史的原因，C以及C++编译器使用逆序（必须把第一个参数赋给printf（格式字符串），放在堆栈的顶部）。第一个和第三个参数是值调用，但是第二个参数通过引用传递，即传递的是缓冲区的地址（由&指示），而不是缓冲区的内容。接着是对库过程的实际调用（第4步）。这个指令是用来调用所有过程的正常过程调用指令。

在可能是由汇编语言写成的库过程中，一般把系统调用的编号放在操作系统所期望的地方，如寄存器中（第5步）。然后执行一个TRAP指令，将用户态切换到内核态，并在内核中的一个固定地址开始执行（第6步）。TRAP指令实际上与过程调用指令非常类似，它们后面都跟随一个来自远处位置的指令，以及供以后使用的一个保存在栈中的返回地址。

然而，TRAP指令与过程指令存在两个方面的差别。首先，它的副作用是，切换到内核态。而过程调用指令并不改变模式。其次，不像给定过程所在的相对或绝对地址那样，TRAP指令不能跳转到任意地址上。根据机器的体系结构，或者跳转到一个单固定地址上，或者指令中有一8位长的字段，它给定了内存中一张表格的索引，这张表格中含有跳转地址。

跟随在TRAP指令后的内核代码开始检查系统调用编号，然后分派给正确的系统调用处理器，这通常是通过一张由系统调用编号所引用的、指向系统调用处理器的指针表来完成（第7步）。此时，系统调用处理器运行（第8步）。一旦系统调用处理器完成其工作，控制可能会在跟随TRAP指令后面的指令中返回给用户空间库过程（第9步）。这个过程接着以通常的过程调用返回的方式，返回到用户程序（第10步）。

为了完成整个工作，用户程序还必须清除堆栈，如同它在进行任何过程调用之后一样（第11步）。假设堆栈向下增长，如经常所做的那样，编译后的代码准确地增加堆栈指针值，以便清除调用read之前压入的参数。在这之后，原来的程序就可以随意执行了。

在前面第9步中，我们提到“控制可能会在跟随TRAP指令后面的指令中返回给用户空间库过程”，这是有原因的。系统调用可能堵塞调用者，避免它继续执行。例如，如果试图读键盘，但是并没有任何键入，那么调用者就必须被阻塞。在这种情形下，操作系统会查看是否有其他可以运行的进程。稍后，当需要的输入出现时，进程会提醒系统注意，然后步骤9～步骤11会接着进行。

下面几小节中，我们将考察一些常用的POSIX系统调用，或者用更专业的说法，考察进行这些系统调用的库过程。POSIX大约有100个过程调用，它们中最重要的过程调用列在图1-18中。为方便起见，它们被分成4类。我们用文字简要地叙述其作用。

进 程 管 理

调　用	说　明
pid = fork()	创建与父进程相同的子进程
pid = waitpid(pid, &statloc,options)	等待一个子进程终止
s = execve(name, argv, environp)	替换一个进程的核心映像
exit(status)	终止进程执行并返回状态

文 件 管 理

调　用	说　明
fd = open(file, how, ...)	打开一个文件供读、写或两者
s = close(fd)	关闭一个打开的文件
n = read(fd, buffer, nbytes)	把数据从一个文件读到缓冲区中
n = write(fd, buffer, nbytes)	把数据从缓冲区写到一个文件中
position = lseek(fd, offset, whence)	移动文件指针
s = stat(name, &buf)	取得文件的状态信息

目录和文件系统管理

调　用	说　明
s = mkdir(name, mode)	创建一个新目录
s = rmdir(name)	删去一个空目录
s = link(name1, name2)	创建一个新目录项name2，并指向name1
s = unlink(name)	删去一个目录项
s = mount(special, name, flag)	安装一个文件系统
s = umount(special)	卸载一个文件系统

杂　项

调　用	说　明
s = chdir(dirname)	改变工作目录
s = chmod(name, mode)	修改一个文件的保护位
s = kill(pid, signal)	发送信号给一个进程
seconds =time(&seconds)	自1970年1月1日起的流逝时间

图1-18　一些重要的POSIX系统调用。若出错则返回代码s为−1。返回代码如下：pid是进程的id，fd是文件描述符，*n*是字节数，position是在文件中的偏移量，而seconds是流逝时间。参数在正文中解释

从广义上看，由这些调用所提供的服务确定了多数操作系统应该具有的功能，而在个人计算机上，资源管理功能是较弱的（至少与多用户的大型机相比较是这样）。所包含的服务有创建与终止进程，创建、删除、读出和写入文件，目录管理以及完成输入/输出。

有必要指出，将POSIX 过程映射到系统调用并不是一对一的。POSIX标准定义了构造系统所必须提供的一套过程，但是并没有规定它们是系统调用、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，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

图1-18中的第一组调用用于进程管理。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是唯一可以在POSIX中创建进程的途径。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，并且在父进程中等于子进程的**进程标识符**（Process IDentifier，PID）。使用返回的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waitpid系统调用，它只是等待，直至子进程终止（若有多个子进程的话，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，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。例如，如果没有已经退出的子进程则立即返回。

现在考虑shell如何使用fork。在键入一条命令后，shell调用fork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，我们把它们都视为系统调用。）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 一个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，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

```
main(argc, argv, envp)
```

其中argc是该命令行内有关参数数目的计数器，包括程序名称。例如，上面的例子中，argc为3。

第二个参数argv是一个指向数组的指针。该数组的元素*i*是指向该命令行第*i*个字符串的指针。在本例中，argv[0]指向字符串“cp”，argv[1]指向字符串“file1”，argv[2]指向字符串“file2”。

main的第三个参数envp是一个指向环境的指针，该环境是一个数组，含有name = value的赋值形式，用以将诸如终端类型以及根目录等信息传送给程序。还有供程序调用的库过程，用来取得环境变量，这些变量通常用来确定用户希望如何完成特定的任务（例如，使用默认打印机）。在图1-19中，没有环境参数传递给子进程，所以execve的第三个参数为0。

如果读者认为exec过于复杂，那么也不要失望。这是在POSIX的全部系统调用中最复杂的一个（语义上），其他的都非常简单。作为一个简单例子，考虑exit，这是在进程完成执行后应执行的系统调用。这个系统调用有一个参数——退出状态（0至255），该参数通过waitpid系统调用中的statloc返回父进程。

图1-20 进程有三段：正文段、数据段和堆栈段

在UNIX中的进程将其存储空间划分为三段：**正文段**（如程序代码）、**数据段**（如变量）以及**堆栈段**。数据向上增长而堆栈向下增长，如图1-20所示。夹在中间的是未使用的地址空间。堆栈在需要时自动地向中间增长，不过数据段的扩展是显式地通过系统调用brk进行的，在数据段扩充后，该系统调用指定一个新地址。但是，这个调用不是POSIX标准中定义的，对于存储器的动态分配，鼓励程序员使用malloc库过程，而malloc的内部实现则不是一个适合标准化的主题，因为几乎没有程序员直接使用它，我们有理由怀疑是否会有人注意到brk实际不是属于POSIX的。

1.6.2 用于文件管理的系统调用

许多系统调用与文件系统有关。本小节讨论在单个文件上的操作，1.6.3节将讨论与目录和整个文件系统有关的内容。

要读写一个文件，先要使用open打开该文件。这个系统调用通过绝对路径名或指向工作目录的相对路径名指定要打开文件的名称，而代码O_RDONLY、O_WRONLY或O_RDWR的含义分别是只读、只写或两者都可以。为了创建一个新文件，使用O_CREAT参数。然后可使用返回的文件描述符进行读写操作。接着，可以用close关闭文件，这个调用使得该文件描述符在后续的open中被再次使用。

毫无疑问，最常用的调用是read和write。我们在前面已经讨论过read。write具有与read相同的参数。

尽管多数程序频繁地读写文件，但是仍有一些应用程序需要能够随机访问一个文件的任意部分。与每个文件相关的是一个指向文件当前位置的指针。在顺序读（写）时，该指针通常指向要读出（写入）的下一个字节。lseek调用可以改变该位置指针的值，这样后续的read或write调用就可以在文件的任何地方开始。

lseek有三个参数：第一个是文件的描述符，第二个是文件位置，第三个说明该文件位置是相对于文件起始位置、当前位置还是文件的结尾。在修改了指针之后，lseek所返回的值是文件中的绝对位置。

UNIX为每个文件保存了该文件的类型（普通文件、特殊文件、目录等）、大小、最后修改时间以及其他信息。程序可以通过stat系统调用查看这些信息。第一个参数指定了要被检查的文件；第二个参数是一个指针，该指针指向存放这些信息的结构。对于一个打开的文件而言，fstat调用完成同样的工作。

1.6.3 用于目录管理的系统调用

本小节我们讨论与目录或整个文件系统有关的某些系统调用，而不是1.6.2节中与一个特定文件有关的系统调用。mkdir和rmdir分别用于创建和删除空目录。下一个调用是link。它的作用是允许同一个文

件以两个或多个名称出现，多数情形下是在不同的目录中这样做。它的典型应用是，在同一个开发团队中允许若干个成员共享一个共同的文件，他们每个人都在自己的目录中有该文件，但可能采用的是不同的名称。共享一个文件，与每个团队成员都有一个私用副本并不是同一件事，因为共享文件意味着任何成员所做的修改都立即为其他成员所见——只有一个文件存在。而在复制了一个文件的多个副本之后，对其中一个副本所进行的修改并不会影响到其他的副本。

为了考察link是如何工作的，考虑图1-21a中的情形。有两个用户ast和jim，每个用户都有一些文件的目录。若ast现在执行一个含有系统调用的程序

```
link("/usr/jim/memo","usr/ast/note");
```

jim目录中的文件memo以文件名note进入ast的目录。之后，/usr/jim/memo和/usr/ast/note都引用相同的文件。顺便提及，用户是将目录保存在/usr、/user、/home还是其他地方，完全取决于本地系统管理员。

理解link是如何工作的也许有助于读者看清其作用。在UNIX中，每个文件都有唯一的编号，即i-编号，用以标识文件。该i-编号是对i-**节点**表格的一个引用，它们一一对应，说明该文件的拥有者、磁盘块的位置等。目录就是一个包含了（i-编号，ASCII名称）对集合的文件。在UNIX的第一个版本中，每个目录项有16字节——2字节用于i-编号，14字节用于名称。现在为了支持长文件名，采用了更复杂的结构，但是，在概念上，目录仍然是（i-编号，ASCII名称）对的一个集合。在图1-21中，mail为i-编号16，等等。link所做的只是利用某个已有文件的i-编号，创建一个新目录项（也许用一个新名称）。在图1-21b中两个目录项有相同的i-编号（70），从而指向同一个文件。如果使用unlink系统调用将其中一个文件移走了，可以保留另一个。如果两个都被移走了，UNIX 00看到尚且存在的文件没有目录项（i-节点中的一个域记录着指向该文件的目录项），就会把该文件从磁盘中移去。

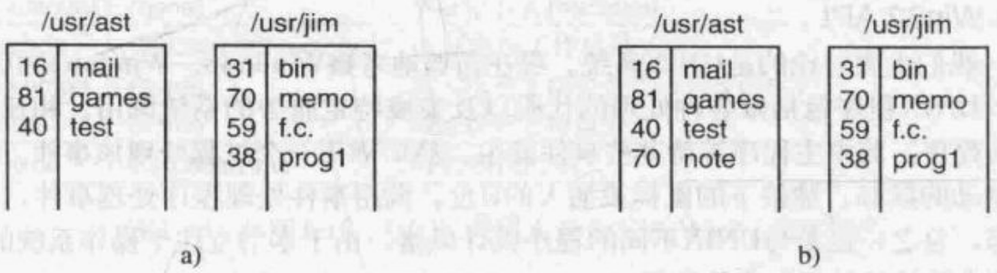

图1-21 a) 将/usr/jim/memo链接到ast目录之前的两个目录；b) 链接之后的两个目录

正如我们已经叙述过的，mount系统调用允许将两个文件系统合并成为一个。通常的情形是，在硬盘某个分区中的根文件系统含有常用命令的二进制（可执行）版和其他常用的文件，用户文件在另一个分区。并且，用户可插入包含需要读入的文件的U盘。

通过执行mount系统调用，可以将一个USB文件系统添加到根文件系统中，如图1-22所示。完成安装操作的典型C语句为

```
mount("/dev/sdb0","/mnt",0);
```

这里，第一个参数是USB驱动器0的块特殊文件名称，第二个参数是要被安装在树中的位置，第三个参数说明将要安装的文件系统是可读写的还是只读的。

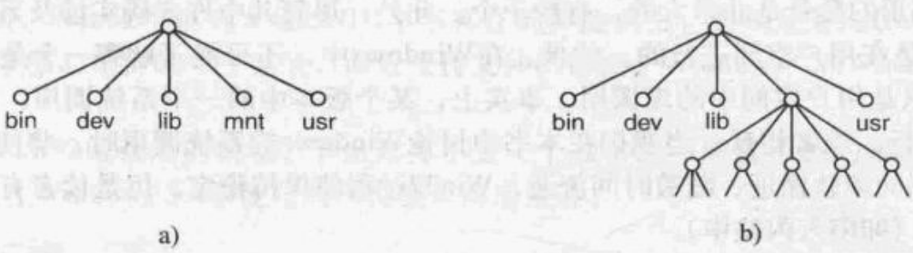

图1-22 a) 安装前的文件系统；b) 安装后的文件系统

在mount调用之后，驱动器0上的文件可以使用从根目录开始的路径或工作目录路径，而不用考虑文件在哪个驱动器上。事实上，第二个、第三个以及第四个驱动器也可安装在树上的任何地方。mount调用使得把可移动介质都集中到一个文件层次中成为可能，而不用考虑文件在哪个驱动器上。尽管这是个CD-ROM的例子，但是也可以用同样的方法安装硬盘或者硬盘的一部分（常称为**分区**或**次级设备**），外部硬盘和USB盘也一样。当不再需要一个文件系统时，可以用umount系统调用卸载之。

1.6.4 各种系统调用

有各种的系统调用。这里介绍系统调用中的一部分。chdir调用改变当前的工作目录。在调用

```
chdir("/usr/ast/test");
```

之后，打开xyz文件，会打开/usr/ast/test/xyz。工作目录的概念消除了总是键入（长）绝对路径名的需要。

在UNIX中，每个文件有一个保护模式。该模式包括针对所有者、组和其他用户的读-写-执行位。chmod系统调用可以改变文件的模式。例如，要使一个文件对除了所有者之外的用户只读，可以执行

```
chmod("file",0644);
```

kill系统调用供用户或用户进程发送信号用。若一个进程准备好捕捉一个特定的信号，那么，在信号到来时，运行一个信号处理程序。如果该进程没有准备好，那么信号的到来会杀掉该进程（此调用名称的由来）。

POSIX定义了若干处理时间的过程。例如，time以秒为单位返回当前时间，0对应着1970年1月1日午夜（从此日开始，没有结束）。在一台32位字的计算机中，time的最大值是$2^{32}-1$秒（假设是无符号整数）。这个数字对应136年多一点。所以在2106年，32位的UNIX系统会发狂，与2000年对世界计算机造成严重破坏的知名Y2K问题是类似的。如果读者现在有32位UNIX系统，建议在2106年之前的某时刻更换为64位的系统。

1.6.5 Windows Win32 API

到目前为止，我们主要讨论的是UNIX系统。现在简要地考察Windows。Windows和UNIX的主要差别在于编程方式。UNIX程序包括做各种处理的代码以及完成特定服务的系统调用。相反，Windows程序通常是事件驱动程序。其中主程序等待某些事件发生，然后调用一个过程处理该事件。典型的事件包括被敲击的键、移动的鼠标、被按下的鼠标或插入的U盘。调用事件处理程序处理事件，刷新屏幕，并更新内部程序状态。总之，这是与UNIX不同的程序设计风格，由于本书专注于操作系统的功能和结构，这些程序设计方式上的差异就不过多涉及了。

当然，在Windows中也有系统调用。在UNIX中，系统调用（如read）和系统调用所使用的库过程（如read）之间几乎是一一对应的关系。换句话说，对于每个系统调用，差不多就涉及一个被调用的库过程，如图1-17所示。此外，POSIX有约100个过程调用。

在Windows中，情况就大不相同了。首先，库调用和实际的系统调用几乎是不对应的。微软定义了一套过程，称为**Win32应用编程接口**（Application Program Interface，API），程序员用这套过程获得操作系统的服务。从Windows 95开始的所有Windows版本都（或部分）支持这个接口。由于接口与实际的系统调用不对应，微软保留了随着时间（甚至随着版本到版本）改变实际系统调用的能力，防止已有的程序失效。由于最新几版Windows中有许多过去没有的新调用，所以究竟Win32是由什么构成的，这个问题的答案仍然是含混不清的。在本小节中，Win32表示所有Windows版本都支持的接口。Win32提供各Windows版本的兼容性。

Win32 API 调用的数量是非常大的，有数千个。此外，尽管其中许多确实涉及系统调用，但有一大批Win32 API完全是在用户空间进行的。结果，在Windows中，不可能了解哪一个是系统调用（如由内核完成），哪一个只是用户空间中的库调用。事实上，某个版本中的一个系统调用，会在另一个不同版本的用户空间中执行，或者相反。当我们在本书中讨论Windows的系统调用时，将使用Win32过程（在合适之处），这是因为微软保证：随着时间流逝，Win32过程将保持稳定。但是读者有必要记住，它们并不全都是系统调用（即陷入内核中）。

Win32 API中有大量的调用，用来管理视窗、几何图形、文本、字体、滚动条、对话框、菜单以及GUI的其他功能。为了使图形子系统在内核中运行（某些Windows版本中确实是这样，但不是所有的版

本），需要系统调用，否则只有库调用。在本书中是否应该讨论这些调用呢？由于它们与操作系统的功能并不相关，我们还是决定不讨论它们，尽管它们会在内核中运行。对Win32 API有兴趣的读者应该参阅一些书籍中的有关内容（例如，Hart，1997；Rector和Newcomer，1997；Simon，1997）。

我们在这里介绍所有的Win32 API，不过这不是我们关心的主要问题，所以做了一些限制，只将那些与图1-18中UNIX系统调用大致对应的Windows调用列在图1-23中。

UNIX	Win32	说 明
fork	CreateProcess	创建一个新进程
waitpid	WaitForSingleObject	等待一个进程退出
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	终止执行
open	CreateFile	创建一个文件或打开一个已有的文件
close	CloseHandle	关闭一个文件
read	ReadFile	从一个文件读数据
write	WriteFile	把数据写入一个文件
lseek	SetFilePointer	移动文件指针
stat	GetFileAttributesEx	取得文件的属性
mkdir	CreateDirectory	创建一个新目录
rmdir	RemoveDirectory	删除一个空目录
link	(none)	Win32不支持link
unlink	DeleteFile	销毁一个已有的文件
mount	(none)	Win32不支持mount
umount	(none)	Win32不支持umount
chdir	SetCurrentDirectory	改变当前工作目录
chmod	(none)	Win32不支持安全性（但NT支持）
kill	(none)	Win32不支持信号
time	GetLocalTime	获得当前时间

图1-23 与图1-18中UNIX调用大致对应的Win32 API 调用

下面简要地说明一下图1-23中的内容。CreateProcess用于创建一个新进程，它把UNIX中的fork和execve结合起来。它有许多参数用来指定新创建进程的性质。Windows中没有类似UNIX中的进程层次，所以不存在父进程和子进程的概念。在进程创建之后，创建者和被创建者是平等的。WaitForSingleObject用于等待一个事件，等待的事件可以是多种可能的事件。如果有参数指定了某个进程，那么调用者等待所指定的进程退出，这通过使用ExitProcess完成。

接下来的6个调用进行文件操作，在功能上和UNIX的对应调用类似，而在参数和细节上是不同的。和UNIX中一样，文件可被打开、关闭和写入。SetFilePointer以及GetFileAttributesEx调用设置文件的位置并取得文件的属性。

Windows中有目录，目录分别用CreateDirectory以及RemoveDirectory API调用创建和删除。也有对当前目录的标记，这可以通过SetCurrentDirectory来设置。使用GetLocalTime可获得当前时间。

Win32接口中没有文件的链接、文件系统的安装、安全属性或信号，所以对应于UNIX中的这些调用就不存在了。当然，Win32中也有大量UNIX中不存在的其他调用，特别是管理GUI的各种调用。在Windows Vista中有了精心设计的安全系统，而且支持文件的链接。Windows 7和Windows 8也加入了更多特性和系统调用。

也许有必要对Win32做最后的说明。Win32并不是一个非常统一的或一致的接口。其主要原因是Win32需要与早期的在Windows 3.x中使用的16位接口向后兼容。

1.7 操作系统结构

我们已经分析了操作系统的外部（如程序员接口），现在是分析其内部的时候了。在下面的小节中，