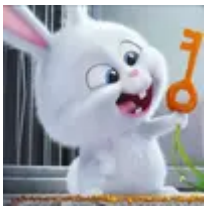
写文章



现代操作系统（第四版）课后习题解答

Modern operating systems (4e) Problem solutions，英文原版解答的翻译

第 1 章习题解答

1. 操作系统必须为用户提供扩展的机器，并管理 I/O 设备和其他系统资源。在某种程度上，这些是不同的功能。
2. 显然，有很多可能的答案。以下是一些示例。
   大型机操作系统：保险公司的索赔处理。
   服务器操作系统：Siri 的语音转文本服务。
   多处理器操作系统：视频编辑和渲染。
   个人电脑操作系统：文字处理应用程序。
   手持电脑操作系统：上下文感知推荐系统。
   嵌入式操作系统：编程 DVD 录像机记录电视。
   传感器节点操作系统：监测野外地区的温度。
   实时操作系统：空中交通管制系统。智能卡操作系统：电子支付。
3. 在分时系统中，多个用户可以使用自己的终端同时访问和执行计算系统上的计算。多道程序设计系统允许用户同时运行多个程序。所有分时系统都是多道程序设计系统，但并非所有多道程序设计系统都是分时系统，因为多道程序设计系统可以在只有一个用户的个人电脑上运行。
4. 实证证据显示，内存访问表现出引用局部性原理，即如果读取一个位置，则很可能接下来会访问附近的位置，尤其是下一个内存位置。因此，通过缓存整个缓存行，下一个缓存命中的概率增加。此外，现代硬件可以将 32 或 64 字节的数据块传送到缓存行中，比逐个字读取同样的数据要快得多。
5. 多道程序设计的主要原因是在等待 I/O 完成时给 CPU 一些工作做。如果没有 DMA，CPU 在执行 I/O 时将被充分占用，因此无法从多道程序设计中获益（至少从 CPU 利用率的角度来看）。无论程序进行多少 I/O，CPU 都将保持 100%忙碌。当然，这是在假设主要的延迟是等待数据复制的情况下。如果 I/O 由于其他原因变慢（例如，通过串行线到达），CPU 可以完成其他工作。
6. 对 I/O 设备（如打印机）的访问通常会对不同用户进行限制。某些用户可能被允许随意打印页面，某些用户可能不被允许打印，而某些用户可能只能打印一定数量的页面。这些限制是由系统管理员根据某些策略设置的。需要执行这些策略，以防止用户级程序对其进行干扰。
7. 例如，英特尔生产了具有各种不同属性（包括速度和功耗）的 Core i3、i5 和 i7 中央处理器。所有这些机器在体系结构上都是兼容的，它们的区别只在于价格和性能，这就是“家族”概念的本质。
8. 一个 25\times 80 字符的单色文本屏幕需要一个 2000 字节的缓冲区。一个 1200\times 900 像素的 24 位彩色位图需要 3240000 字节的空间。在 1980 年，这两种选择分别需要 10 美元和 15820 美元。如需了解当前价格，请查看当前 RAM 的价格，可能是每 MB 几美分。

9. 考虑公平性和实时性。公平性要求每个进程以公平的方式分配其资源，没有进程能获取超过其公平份额的资源。另一方面，实时性要求根据不同进程必须完成执行的时间来分配资源。实时进程可能会获取不成比例的资源份额。

10. 大多数现代中央处理器提供两种执行模式：内核模式和用户模式。当在内核模式下执行时，CPU 可以执行其指令集中的每个指令，并利用硬件的每个功能。然而，当在用户模式下执行时，它只能执行指令集的子集，并且只能使用硬件的子集。拥有两种模式允许设计者在用户模式下运行用户程序，从而限制它们对关键指令的访问。

11. 磁盘的磁头数目 = 255GB / (65536*255*512) = 16 磁盘的盘片数目 = 16/2 = 8 读操作完成所需的时间 = 寻道时间 + 旋转延迟 + 传输时间。寻道时间为 11 ms，旋转延迟为 7 ms，传输时间为 4 ms，因此平均传输时间为 22 毫秒。

12. 选项 (a), (c), 和 (d) 应限制在内核模式下。

13. 这些程序的执行时间可能为 20 毫秒、25 毫秒或 30 毫秒，这取决于操作系统如何进行调度。如果 *P0* 和 *P1* 被调度到同一个 CPU，而 *P2* 被调度到另一个 CPU，那么执行时间为 20 毫秒。如果 *P0* 和 *P2* 被调度到同一个 CPU，而 *P1* 被调度到另一个 CPU，那么执行时间为 25 毫秒。如果 *P1* 和 *P2* 被调度到同一个 CPU，而 *P0* 被调度到另一个 CPU，那么执行时间为 30 毫秒。如果所有三个进程都在同一个 CPU 上，那么执行时间为 35 毫秒。

14. 每纳秒，有一条指令从流水线中出来。这意味着机器每秒执行 10 亿条指令。流水线的级数并不重要。具有 10 级流水线且每级需要 1 纳秒的机器也能每秒执行 10 亿条指令。唯一重要的是一个完成的指令多久能从流水线末端弹出。

15. 平均访问时间 = 0.95\times 1 纳秒（字在缓存中） +0.05\times 0.99\times 10 纳秒（字在 RAM 中，但不在缓存中） +0.05\times 0.01\times 10,000,000 纳秒（字仅在磁盘中） =5001.445 纳秒 =5.001445 \mu 秒

16. 可能。如果调用者在返回之后立即覆盖数据，当写操作最终发生时，错误的数据将被写入。但是，如果驱动程序在返回之前先将数据复制到私有缓冲区中，那么调用者可以立即继续执行。另一种可能性是允许调用者继续执行，并在缓冲区可被重用时给予信号，但这样做很棘手且容易出错。

17. 陷阱指令将 CPU 的执行模式从用户模式切换到内核模式。该指令允许用户程序调用操作系统内核中的函数。

18. 进程表用于存储当前被挂起的进程（就绪或阻塞）的状态。即使用户什么都不做，也不打开任何程序，现代个人计算机系统也有数十个进程在运行。它们会检查更新、加载电子邮件等等。在 UNIX 系统上，可以使用 *ps -a* 命令查看它们。在 Windows 系统上，可以使用任务管理器。

19. 挂载文件系统会导致挂载点目录中的任何文件无法访问，因此挂载点通常为空。然而，系统管理员可能会希望将通常位于挂载目录中的一些最重要的文件复制到挂载点，以便在挂载设备进行修复时在紧急情况下能够按照正常路径找到它们。

20. 如果进程表中没有剩余的空闲槽位（或者可能没有内存或交换空间剩余），fork 可能会失败。如果给定的文件名不存在或不是有效的可执行文件，exec 可能会失败。如果要取消链接的文件不存在或调用进程没有取消链接的权限，则 unlink 可能会失败。

21. 时间复用：CPU，网络卡，打印机，键盘。空间复用：内存，磁盘。两者：显示器。

22. 如果调用失败，例如因为 *fd* 不正确，它可以返回 -1 。它也可能因为磁盘已满而无法写入所请求的字节数而失败。在正确终止时，它总是返回 *nbytes*。

23. 它包含的字节为：1、5、9、2。

24. 检索文件的时间 = 1 * 50 毫秒（将磁臂移动到第 50 道扇区所需的时间）+ 5 毫秒（第一个扇区在磁头下旋转的时间）+ 10 / 200 * 1000 毫秒（读取 10 MB 的时间）= 105 毫秒。

25. 块特殊文件由编号的块组成，每个块可以独立地进行读取或写入。可以定位到任何块并开始读取或写入。这对于字符特殊文件是不可能的。

26. 系统调用实际上没有名称，除了在文档上的意义上有一个名称。当库函数 *read* 陷入内核时，它将系统调用的编号放入寄存器或堆栈中。此编号用于索引到表中。实际上没有任何使用的名称。另一方面，库函数的名称非常重要，因为它出现在程序中。
27. 这允许可执行程序在机器的不同部分的内存中在不同的运行中加载。此外，它使得程序的大小能够超过机器的内存大小。
28. 就程序逻辑而言，调用库函数是否导致系统调用并不重要。但是如果性能是一个问题，如果一个任务可以在没有系统调用的情况下完成，程序将运行得更快。每个系统调用都会涉及在用户上下文与内核上下文之间切换的额外时间开销。此外，在多用户系统上，当系统调用完成时，操作系统可能会调度另一个进程运行，进一步减慢调用进程在实时中的进展速度。
29. 几个 UNIX 调用在 Win32 API 中没有对应项：Link：Win32 程序不能通过替代名称引用文件或在多个目录中看到该文件。而且，尝试创建链接是测试文件并创建文件锁的方便方法。Mount 和 umount：Windows 程序不能假设标准路径名，因为在具有多个磁盘驱动器的系统上，路径的驱动器名称部分可能不同。Chmod：Windows 使用访问控制列表。Kill：Windows 程序员无法终止不合作的恶意程序。
30. 每个系统架构都有其可以执行的指令集。因此，Pentium 无法执行 SPARC 程序，而 SPARC 无法执行 Pentium 程序。此外，不同的架构在使用的总线架构（如 VME、ISA、PCI、MCA、SBus 等）以及 CPU 的字大小（通常为 32 位或 64 位）上也有所不同。由于硬件上的这些差异，构建一个完全可移植的操作系统是不可行的。高度可移植的操作系统将包括两个高级层次——一个机器相关层和一个机器无关层。机器相关层处理硬件的具体细节，并且必须为每种架构单独实现。这一层提供了一个统一的接口，用于构建机器无关层。机器无关层只需实现一次。为了实现高度可移植，必须尽量减小机器相关层的大小。
31. 策略和机制的分离允许操作系统设计者在内核中实现一小组基本原语。这些原语简化了，因为它们不依赖于任何特定的策略。然后可以使用这些原语在用户级别实现更复杂的机制和策略。
32. 虚拟化层引入了增加的内存使用量和处理器开销，以及增加的性能开销。
33. 转换很简单：

(a) 一个纳秒年是 10^{-9}\times 365\times 24\times 3600=31.536 毫秒。

(b) 1 米

(c) 有 2^{50} 字节，即 1,099,511,627,776 字节。

(d) 它是 6\times 10^{24} 千克或 6\times 10^{27} 克。

第 2 章习题解答

1. 从被阻塞到运行的过渡是可以想象的。假设一个进程在 I/O 上被阻塞，然后 I/O 完成。如果 CPU 是空闲的，该进程可以直接从被阻塞状态转到运行状态。另外一个缺少的转换，从就绪到阻塞是不可能的。就绪状态的进程不能进行 I/O 或其他可能阻塞它的操作。只有运行状态的进程才能被阻塞。
2. 可以有一个寄存器包含指向当前进程表项的指针。当 I/O 完成时，CPU 会将当前的机器状态存储在当前进程表项中。然后它会到中断向量中取得中断设备的指针，指向另一个进程表项（服务程序）。然后启动这个进程。
3. 一般来说，高级语言不允许对 CPU 硬件进行所需的访问。例如，中断处理程序可能需要启用和禁用特定设备的中断，或在进程的堆栈区域内操作数据。而且，中断服务程序必须尽快执行完毕。
4. 使用一个单独的内核栈有几个原因。其中两个原因如下。首先，由于用户程序编写不良导致的栈空间不足，你不希望操作系统崩溃。其次，如果内核在系统调用返回时将栈数据保留在用户程序内存空间中，恶意用户可能利用这些数据来了解其他进程的信息。
5. 所有五个进程都处于空闲状态的机会是 1/32，所以 CPU 的空闲时间是 1/32。

6. 内存中有足够的空间容纳 14 个进程。如果一个进程的 I/O 概率为 p，那么所有进程都在等待 I/O 的概率是 p^{14} 。将此等式设为 0.01，我们得到了方程 p^{14}=0.01 。解得 p=0.72 ，所以我们可以容忍 I/O 等待时间最长为 72% 的进程。
7. 如果每个作业有 50% 的 I/O 等待时间，在没有竞争的情况下完成一个作业需要 40 分钟。按顺序运行，第二个作业将在第一个作业开始后 80 分钟完成。有两个作业时，近似的 CPU 利用率为 1-0.5^{2} 。因此，每个作业在每分钟的真实时间内可以获得 0.375 分钟的 CPU 时间。要累积 20 分钟的 CPU 时间，一个作业必须运行 20/0.375 分钟，约为 53.33 分钟。因此，按顺序运行，作业在 80 分钟后完成，但并行运行，它们在 53.33 分钟后完成。
8. 所有进程都等待 I/O 的概率是 0.4^{6} ，即 0.004096 。因此，CPU 利用率 = 1-0.004096 = 0 : 995904。
9. 客户端进程可以创建独立的线程，每个线程可以从镜像服务器中获取文件的不同部分。这可以帮助减少停机时间。当然，所有线程共享一个网络链接。随着线程数量的增加，这个链接可能成为瓶颈。
10. 保持文件系统一致将是困难的，甚至是不可能的。假设一个客户端进程发送一个请求到服务器进程 1 来更新一个文件。这个进程更新了它内存中的缓存条目。不久之后，另一个客户端进程发送一个请求到服务器 2 来读取那个文件。不幸的是，如果文件也被缓存在那里，服务器 2 在它的无辜中会返回过时的数据。如果第一个进程在将文件缓存之后将其写入磁盘，并且服务器 2 在每次读取时检查磁盘以查看它的缓存副本是否是最新的，那么系统可以正常工作，但恰恰是所有这些磁盘访问是缓存系统试图避免的。
11. 不可以。如果一个单线程进程在键盘上被阻塞，它不能进行复制。
12. 当工作线程需要从磁盘读取网页时，会被阻塞。如果使用的是用户级线程，这个操作将会阻塞整个进程，破坏了多线程的价值。因此，使用内核级线程非常重要，可以允许一些线程阻塞而不影响其他线程。
13. 是的。如果服务器完全受限于 CPU，就没有必要使用多个线程。这只会增加不必要的复杂性。举个例子，考虑一个地区有 100 万人口的电话号码目录辅助服务（如 555-1212）。如果每条（姓名，电话号码）记录大小为 64 个字符，整个数据库大小为 64 兆字节，可以轻松地保持在服务器的内存中以提供快速查找。
14. 当一个线程被停止时，寄存器中会有值。就像进程停止时需要保存寄存器一样，线程也需要保存。多线程和多进程没有什么不同，所以每个线程都需要有自己的寄存器保存区域。
15. 进程中的线程是相互合作的，它们并不敌对。如果需要让某个线程让出执行，为了应用程序的良好运行，它会让步。毕竟，通常是同一个程序员编写了所有线程的代码。
16. 用户级线程在没有使用完整个进程的时间片之前无法被时钟抢占（尽管可能发生透明的时钟中断）。内核级线程可以被单独时钟抢占。在后一种情况下，如果一个线程运行时间过长，时钟将中断当前进程，从而中断当前线程。内核可以自由选择从同一进程中选择不同的线程来运行，如果它愿意的话。
17. 在单线程情况下，缓存命中需要 12 毫秒，缓存未命中需要 87 毫秒。加权平均值为 2/3\times 12+1/3\times 87 。因此，平均请求时间为 37 毫秒，服务器每秒可以处理约 27 个请求。对于多线程服务器，等待磁盘的所有时间是重叠的，所以每个请求只需要 12 毫秒，服务器每秒可以处理 83 1/3 个请求。
18. 最大的优点是效率高。不需要进行内核陷阱来切换线程。最大的缺点是如果一个线程阻塞，整个进程都会阻塞。
19. 是的，可以做到。在每次调用 *pthread_create* 后，主程序可以使用 *pthread_join* 等待刚创建的线程退出，然后再创建下一个线程。
20. 指针是必要的，因为全局变量的大小是未知的。它可以是从字符到浮点数数组的任何类型。如果将值存储起来，就必须给 *create_global* 指定大小，这没问题，但是 *set_global* 的第二个参数应该是什么类型呢？*read_global* 的值应该是什么类型呢？

21. 可能会发生运行时系统正好在阻塞或解除阻塞线程时，忙于操作调度队列。这将是时钟中断处理程序开始检查这些队列是否是执行线程切换的时机非常不恰当的时刻，因为它们可能处于不一致的状态。一种解决方法是在进入运行时系统时设置一个标志。时钟处理程序会察觉到这一点并设置自己的标志，然后返回。当运行时系统完成后，它将检查时钟标志，看到时钟中断发生，然后运行时钟处理程序。

22. 是可能的，但效率低下。想要进行系统调用的线程首先设置一个定时器，然后执行调用。如果调用阻塞，定时器会将控制权返回给线程包。当然，大多数情况下，调用不会阻塞，需要清除定时器。因此，每个可能阻塞的系统调用都必须执行三次系统调用。如果定时器过早触发，就会出现各种问题。这不是一个理想的构建线程包的方式。

23. 是的，它仍然有效，但它仍然在忙等待。

24. 在使用抢占式调度时它肯定有效。实际上，它就是为这种情况设计的。当调度是非抢占式时，可能会失败。考虑这样一种情况，即 *turn* 最初为 0，但是进程 1 先运行。它将无限循环并且永远不会释放 CPU。

25. 优先级倒转问题发生在低优先级进程位于其临界区时，突然有一个高优先级进程就绪并进行调度。如果使用忙等待，它将永远运行。在用户级线程中，不会发生低优先级线程突然被抢占以允许高优先级线程运行的情况。没有抢占。但在内核级线程中可能会出现这个问题。

26. 在轮转调度中它能工作。迟早会运行到 L ，最终它会离开临界区。问题在于，在优先级调度中， L 根本没有机会运行；而在轮转调度中，它会定期获得一个正常的时间片，因此有机会离开临界区。

27. 每个线程都调用自己的过程，因此它必须有自己的栈来存储局部变量、返回地址等。这对于用户级线程和内核级线程来说都是一样的。

28. 是的。模拟的计算机可以进行多程序设计。例如，当进程 A 运行时，它读出一些共享变量。然后发生一个模拟时钟滴答，进程 B 运行。它也读取相同的变量。然后它对该变量加 1。当进程 A 运行时，如果它也对该变量加 1，就会出现竞争条件。

29. 是的，它将按原样工作。在给定的时刻，只能有一个生产者（消费者）向缓冲区添加（删除）项目。

30. 这个解决方案满足互斥条件，因为两个进程不可能同时处于临界区。也就是说，当 turn 为 0 时，P0 可以执行其临界区，但 P1 不能。同样地，当 turn 为 1 时。然而，这假设 P0 必须先运行。如果 P1 生成了某个东西并将其放入缓冲区，那么虽然 P0 可以进入其临界区，但它会发现缓冲区为空并阻塞。此外，该解决方案要求两个进程的严格交替执行，这是不可取的。

31. 执行信号量操作时，操作系统首先禁用中断。然后它读取信号量的值。如果进行的是 down 操作并且信号量等于零，它会将调用进程放入与信号量相关的阻塞进程列表中。如果进行的是 up 操作，它必须检查是否有任何进程被阻塞在该信号量上。如果有一个或多个进程被阻塞，就会从阻塞进程列表中移除一个进程并使其可运行。当所有这些操作完成后，再次启用中断。

32. 与每个计数信号量相关联的是两个二进制信号量， M 用于互斥， B 用于阻塞。每个计数信号量还有一个计数器，用于存储 up 操作次数与 down 操作次数之差，以及一个被阻塞在该信号量上的进程列表。为了执行 down 操作，进程首先通过对 M 进行 down 操作来获得对信号量、计数器和列表的独占访问。然后它递减计数器。如果计数器为零或更多，它只需对 M 进行 up 操作并退出。如果计数器为负数，将进程放入阻塞进程列表。然后对 M 进行 up 操作并对 B 进行 down 操作以阻塞该进程。为了执行 up 操作，首先 down M 以获取互斥访问，然后递增计数器。如果计数器大于零，没有任何进程被阻塞，所以只需要对 M 进行 up 操作。然而，如果计数器现在为负数或零，需要从列表中删除某个进程。最后，按照这个顺序对 B 和 M 进行 up 操作。

33. 如果程序按阶段运行，并且在当前阶段完成之前，两个进程都不能进入下一个阶段，那么使用栅栏是有意义的。

34. 带有内核线程时，线程可以在信号量上阻塞，内核可以在同一进程中运行其他线程。因此，使用信号量没有问题。而对于用户级线程，在一个线程在信号量上阻塞时，内核认为整个进程都被阻塞了，不再运行它，因此进程失败。

35. 这是非常昂贵的实现。每当出现在某个进程正在等待的谓词中的任何变量发生变化时，运行时系统必须重新评估谓词，以确定是否可以解除进程的阻塞。使用 Hoare 和 Brinch Hansen 监视器，进程只能在信号原语上被唤醒。

36. 员工通过传递消息来进行通信：订单、食物和袋子。用 UNIX 术语来说，这四个进程通过管道相连。

37. 它不会导致竞态条件（什么也不会丢失），但它实际上是忙等待。

38. 需要 nT 秒。

39. 创建了三个进程。在初始进程分叉后，有两个正在运行的进程，一个父进程和一个子进程。它们分别进行分叉，创建了两个额外的进程。然后所有的进程都退出。

40. 如果一个进程在列表中多次出现，它将在每个周期中获得多个时间片。这种方法可以用来给更重要的进程分配更大的 CPU 份额。但是当进程阻塞时，所有的可运行进程条目都应该从列表中删除。

41. 在简单的情况下，可以通过查看源代码来确定 I/O 是否会成为限制因素。例如，一个将所有输入文件一次性读入缓冲区的程序可能不会受到 I/O 限制，但是一个逐步从多个不同文件读取和写入的问题（例如编译器）可能会受到 I/O 限制。如果操作系统提供了像 UNIX *ps* 命令这样的工具，可以告诉你程序使用的 CPU 时间量，可以将其与完成程序的总时间进行比较。当然，这在你是唯一用户的系统上最有意义。

42. 如果上下文切换时间很长，那么时间片的值必须成比例地增加。否则，上下文切换的开销可能非常高。选择较大的时间片值可能会导致系统效率低下，如果典型的 CPU 突发时间小于时间片。如果上下文切换非常小或可忽略不计，则可以更自由地选择时间片的值。

43. CPU 效率是指有效 CPU 时间与总 CPU 时间的比值。当 Q\geq T 时，基本循环是进程运行 T 时间并经历进程切换 S 时间。因此，(a) 和 (b) 的效率为 T/(S+T)。当时间片小于 T 时，每次运行 T 将需要 T/Q 次进程切换，浪费时间 ST / Q 。这里的效率为
    $$
    \frac{T}{T+ST/Q}
    $$

    化简后得到 Q/(Q+S)，这是问题 (c) 的答案。对于问题 (d)，我们只需将 Q 替换为 S ，得到的效率为 50%。最后，对于问题 (e)，当 Q 趋近于 0 时，效率趋近于 0。

44. 最短作业优先是最小化平均响应时间的方法。
    $0<X\leq 3 : X , 3, 5, 6, 9$.

    $3<X\leq 5 : 3, X , 5, 6, 9$.

    $5<X\leq 6 : 3, 5, X , 6, 9$.

    $6<X\leq 9 : 3, 5, 6, X , 9$.

    $X>9 : 3, 5, 6, 9, X$.

45. 对于循环调度算法，在前 10 分钟中，每个作业获得 1/5 的 CPU 时间。在 10 分钟结束时，作业 C 完成。在接下来的 8 分钟内，每个作业获得 1/4 的 CPU 时间，之后作业 D 完成。然后，剩下的三个作业每个在接下来的 6 分钟内获得 1/3 的 CPU 时间，直到作业 B 完成，以此类推。五个作业的完成时间分别为 10、18、24、28 和 30 分钟，平均为 22 分钟。对于优先级调度算法，首先运行作业 B。6 分钟后，它完成。其他作业分别在 14、24、26 和 30 分钟完成，平均为 18.8 分钟。如果作业按照 A 到 E 的顺序运行，它们分别在 10、16、18、22 和 30 分钟完成，平均为 19.2 分钟。最后，最短作业优先算法的完成时间为 2、6、12、20 和 30 分钟，平均为 14 分钟。

46. 第一次运行时，它获得 1 个时间量。在接下来的运行中，它分别获得 2、4、8 和 15 个时间量，因此它必须被交换入内存 5 次。

47. 每个语音通话需要 200 个 1 毫秒的样本，共计 200 毫秒的 CPU 时间。视频每秒需要 11 毫秒的时间，约为 33 1/3 次，总共约 367 毫秒。两者之和是每秒的 767 毫秒的真实时间，所以系统是可调度的。

48. 另一个视频流每秒消耗 367 毫秒的时间，总计 1134 毫秒的真实时间，所以系统是不可调度的。

49. 预测的序列是 40、30、35，现在是 25。

50. 使用 CPU 的分数是 35/50 + 20/100 + 10/200 + x/250。为了可调度，这个分数必须小于 1。因此，x 必须小于 12.5 毫秒。

51. 是的。总是至少有一个叉放免费，至少有一个哲学家可以同时拿到两个叉子。因此，不会发生死锁。可以尝试 N=2，N=3 和 N=4，然后推广。

52. 每个语音通话每秒运行 166.67 次，每次使用 1 毫秒的时间，所以每个语音通话每秒需要 166.67 毫秒，两个通话共需要 333.33 毫秒。视频每秒运行 25 次，每次使用 20 毫秒的时间，总共 500 毫秒。两者加起来消耗了 833.33 毫秒，所以还有剩下的时间，系统是可调度的。

53. 内核可以使用任何方式调度进程，但在每个进程内，它严格按优先级顺序运行线程。通过让用户进程设置自己线程的优先级，用户可以控制策略，但内核处理机制。

54. 如果一个哲学家阻塞，邻居可以通过检查他的状态在 *test* 函数中判断到他饿了，因此当叉子可用时可以唤醒他。

55. 这个改变意味着在一个哲学家停止进食之后，他的邻居都不会被选择。事实上，他们永远不会被选择。假设哲学家 2 完成进食。他会为哲学家 1 和哲学家 3 运行 *test* 函数，虽然两者都饿了并且两个叉子都可用，但他们都不会被唤醒。同样地，如果哲学家 4 完成进食，哲学家 3 不会被启动。没有任何东西可以启动他。

56. 变种 1：读者优先。当有读者活动时，没有写者可以开始。当出现新的读者时，它可以立即开始，除非当前有写者活动。当一位写者完成时，如果有读者在等待，所有读者将被同时启动，不考虑是否有写者在等待。
    变种 2：写者优先。当有写者等待时，没有读者可以开始。当最后一个活动进程完成时，如果有写者在等待，则启动一个写者；否则，启动所有读者（如果有）。
    变种 3：对称版本。当有读者活动时，新的读者可以立即开始。当一位写者完成时，如果有写者在等待，新的写者具有优先权。换言之，一旦我们开始阅读，我们会一直阅读，直到没有剩余的读者。同样地，一旦我们开始写作，所有待定的写者都可以运行。

57. 一种可能的 shell 脚本是

```
if [ ! –f numbers ]; then echo 0 > numbers; fi
count = 0
while (test $count != 200 )
do
count='expr $count + 1'
n='tail –1 numbers'
expr $n + 1 >>numbers
done
```

    在同时由后台（使用 &）和前台启动脚本的情况下，运行脚本两次。然后检查文件 *numbers*。它可能一开始看起来像是一个有序的数字列表，但在某个时刻由于并发条件而失去了顺序。可以通过让每个脚本副本在进入临界区之前测试并设置文件的锁，并在离开临界区时解锁来避免并发条件的发生。可以像这样实现：

```
if ln numbers numbers.lock
then
n='tail -1 numbers'
expr $n + 1 > numbers
rm numbers.lock
fi
```

这个版本在文件不可访问时会跳过一个回合。变种解决方案可以将进程置于休眠状态，进行忙等待，或仅计算成功操作的循环次数。

第 3 章习题解答

1. 首先，需要特殊的硬件来进行比较，并且它必须要快，因为它会在每次内存引用时使用。其次，使用 4 位密钥，只能同时存储 16 个程序在内存中（其中一个是操作系统）。
2. 这只是一个巧合。基地址寄存器为 16,384，是因为程序恰好加载在地址 16,384 处。它可以加载在任何位置。限制寄存器为 16,384，是因为程序包含 16,384 字节。它可以有任意长度。加载地址恰好与程序长度完全匹配纯属巧合。
3. 几乎需要复制整个内存，这需要读取每个字并将其重写到不同的位置。读取 4 字节需要 4 纳秒，因此读取 1 字节需要 1 纳秒，并且将其写入需要额外 2 纳秒，总共每个字节压缩需要 2 纳秒。这是一个每秒 500,000,000 字节的速率。要复制 4GB（ 2^{232} 字节，大约 4.295 \times 10^9 字节），计算机需要 2^{32}/500,000,000 秒，大约为 859 毫秒。这个数字稍微悲观，因为如果内存底部有一个初始空洞为 k 字节，这 k 字节则不需要复制。然而，如果存在许多空洞和许多数据段，空洞将很小，因此 k 将很小，计算中的误差也将很小。
4. First fit 需要 20MB、10MB 和 18MB。Best fit 需要 12MB、10MB 和 9MB。Worst fit 需要 20MB、18MB 和 15MB。Next fit 需要 20MB、18MB 和 9MB。
5. 实际内存使用物理地址。这些是内存芯片对总线上的反应数字。虚拟地址是指一个进程地址空间的逻辑地址。因此，一个具有 32 位字长的机器可以生成最大为 4GB 的虚拟地址，无论机器的内存大小是高于还是低于 4GB。
6. 对于 4KB 的页面大小，（页号，偏移量）对是（4，3616）、（8，0）和（14，2656）。对于 8KB 的页面大小，它们是（2，3616）、（4， 0）和（7， 2656）。
7. (a) 8212. (b) 4100. (c) 24684.
8. 他们建立了一个 MMU，并将其插入在 8086 和总线之间。因此，所有 8086 的物理地址作为虚拟地址输入到 MMU 中。然后 MMU 将其映射到实际物理地址，然后传递给总线。
9. 需要一个能够将虚拟页面重新映射到物理页面的 MMU。此外，当引用当前未被映射的页面时，需要向操作系统发出陷阱，以便它可以获取该页面。
10. 如果智能手机支持多程序运行，像 iPhone、Android 和 Windows 手机一样，那么就支持多个进程。如果一个进程分叉并且页面在父进程和子进程之间共享，写时复制是有意义的。智能手机比服务器小，但在逻辑上并没有太大差异。
11. 对于这些大小，为了确保每次访问 X 的元素都会发生 TLB（转换后备缓冲）缺失，M 至少要为 4096。由于 N 只影响 X 被访问的次数，任何 N 的值都可以。2. M 仍然应至少为 4096，以确保每次访问 X 的元素都会发生 TLB 缺失。但现在 N 应大于 64K，以引发 TLB 抖动，即 X 应超过 256 KB。
12. 所有进程的总虚拟地址空间为 nv，因此需要这么多存储空间来存放页面。然而，存储器中可能存在 r 量，因此所需的磁盘存储空间仅为 nv-r。实际上很少需要这么多空间，因为很少有 n 个进程实际运行，且更少有所有进程都需要达到最大允许的虚拟内存。
13. 每 k 条指令发生一次页错误会给平均值增加额外的开销 n/kμs，因此平均指令执行时间为 1+n/k ns。
14. 页表包含 2^32/2^13 个条目，即 524,288 个。加载页表需要 52 毫秒。如果一个进程获得 100 毫秒，其中 52 毫秒用于加载页表，48 毫秒用于运行。因此，52% 的时间用于加载页表。
15. 在这些情况下：1. 我们需要每个页面一个条目，即 2^24=16×1024×1024 个条目，因为页面号字段有 48-12=36 位。 2. 指令地址在 TLB 中命中率达到 100%。直到程序移动到下一个数据页面之前，数据页面的命中率将保持 100%。由于一个 4 KB 的页面包含 1024 个长整数，所以每 1024 次数据引用将有一个 TLB 缺失和一个额外的内存访问。

16. TLB 的命中概率为 0.99，页表的命中概率为 0.0099，页面错误的概率为 0.0001（即只有 10,000 次引用中的 1 次会导致页面错误）。因此，每次有效地址翻译的时间为：
    $$
    0.99\times 1+0.0099\times 100+0.0001\times 6\times 10^{6}\approx 602\text{个时钟周期}。
    $$
    注意，有效地址翻译时间相当高，因为它主要由页面替换时间主导，即使页面错误只发生在 10,000 次引用中的一次。

17. 考虑以下情况：
    (a) 多级页表通过其层次结构减少了需要存储在内存中的实际页表的数量。事实上，在一个具有大量指令和数据局部性的程序中，我们只需要顶层页表（一个页），一个指令页面和一个数据页面。
    (b) 为每个三个页面字段分配 12 位。偏移字段需要 14 位来寻址 16 KB。这样，页面字段还剩下 24 位。由于每个条目为 4 字节，一个页面可以容纳 2^12 个页表条目，因此需要 12 位来索引一个页面。因此，为每个页面字段分配 12 位将能够寻址所有 2^38 个字节。

18. 虚拟地址由（PT1，PT2，偏移量）更改为（PT1，PT2，PT3，偏移量）。但是虚拟地址仍然只使用 32 位。虚拟地址的位配置从（10，10，12）更改为（2，9，9，12）。

19. 20 位被用于虚拟页号，剩余 12 位用于偏移量。这产生了一个 4KB 的页面。20 位的虚拟页意味着有 2^{20} 个页面。

20. 对于一级页表，需要 2^{32}/2^{12} 或者 1M 个页面。因此，页表必须有 1M 个条目。对于二级分页，主页表有 1K 个条目，每个条目指向一个二级页表。其中只有两个被使用到。因此，总共只需要三个页表条目，一个在顶级表中，一个在每个低级表中。

21. 代码和引用字符串如下
    LOAD 6144,R0 1(I), 12(D)
    PUSH R0 2(I), 15(D)
    CALL 5120 2(I), 15(D)
    JEQ 5152 10(I)
    代码（I）表示指令引用，而（D）表示数据引用。

22. 有效指令时间为 1h+5(1-h)，其中 h 是命中率。如果我们将这个公式等于 2 并解出 h，我们可以发现 h 必须至少为 0.75。

23. 关联存储器从本质上将一个密钥与多个寄存器的内容同时进行比较。对于每个寄存器，必须有一组比较器，用于将寄存器内容的每一位与待搜索的密钥进行比较。实现这样一个设备所需的门（或晶体管）数量是寄存器数量的线性函数，因此扩展设计会线性增加成本。

24. 对于 8KB 的页面和 48 位的虚拟地址空间，虚拟页面的数量为 2^{48} / 2^{13}，即 2^{35}（约 340 亿）。

25. 主存储器有 2^{28} / 2^{13} = 32,768 个页面。一个 32K 的哈希表的平均链长度为 1。为了使链长度小于 1，我们必须增加到下一个大小，即 65,536 个条目。将 32,768 个条目分散到 65,536 个表槽位，将会得到平均链长度为 0.5，这样可以确保快速查找。

26. 除非是在编译时完全确定的程序，否则这可能是不可能的，除非是非常特殊且不太有用的情况。如果编译器收集有关调用过程代码中位置的信息，这些信息可以在链接时用于重新排列目标代码，使得过程位于调用代码附近。当然，对于从程序中的许多位置调用的过程，这并没有太大帮助。

27. 在这种情况下，1. 每次引用都会出现页面错误，除非页面帧的数量是 512，即整个序列的长度。2. 如果有 500 个帧，则将页面 0-498 映射到固定帧，并只变化一个帧。

28. FIFO 算法的页面帧如下：
    $$
    \begin{array}
    {l}\text{x0172333300}\\text{xx017222233}\\text{xxx01777722}\\text{xxxx0111177}\end{array

}
$$

LRU 算法的页面帧如下：
$$
\begin{array}
{l}\text{x0172327103}\\text{xx017232710}\\text{xxx01773271}\\text{xxxx0111327}\end{array
}
$$

FIFO 算法产生 6 个页面错误；LRU 算法产生 7 个页面错误。

29. 选择的第一个带有 0 位的页面是 D 。

30. 计数器的值为
页面 0: 0110110
页面 1: 01001001
页面 2: 00110111
页面 3: 10001011

31. 序列：0, 1, 2, 1, 2, 0, 3。在 LRU 算法中，页面 1 将被页面 3 替换。在 Clock 算法中，页面 1 将被替换，因为所有页面都将被标记，且指针指向页面 0。

32. 页面的年龄为 2204-1213=991 。如果 \tau=400 ，页面肯定不在工作集中且最近没有被引用，因此会被驱逐。当 \tau=1000 时情况不同。现在页面仍在工作集中（勉强），所以不会被移除。

33. a) 对于每个设置了 R 位的页面，将时间戳设为 10 并清除所有 R 位。也可以将 (0,1) 的 R - M 条目改为 (0,0*)。因此，页面 1 和页面 2 的条目将改为：*
$$
\begin{array}{|l|l|l|l|l|}\hline\text{页面}&\text{时间戳}&\text{V}& \text{R}&\text{M}\ \hline 0&6&1&0&0^{}\ \hline 1&10&1&0&0\ \hline 2&10&1&0&1\ \hline\end{array}
$$

b) 退出页面 3 （ R=0 且 M=0 ）并加载页面 4：
$$
\begin{array}{|l|l|l|l|l|}\hline\text{页面}&\text{时间戳}&\text{V}&
\text{R}&\text{M}&\text{注释}\ \hline 0&6&1&0&1&\ \hline 1&9&1&1&0&\ \hline
2&9&1&1&1&\ \hline 3&7&0&0&0&\text{原先是 7(1,0,0)}\ \hline 4&10&1&1&0&\text{原先是
4(0,0,0)}\ \hline\end{array}
$$

34. a) 属性为：(FIFO) 加载时间；(LRU) 最近的引用时间；(Optimal) 未来最近的引用时间。 b) 有标记算法和替换算法。标记算法为每个页面添加 a 部分给定的属性标签。替换算法将剔除标签值最小的页面。

35. 寻道加旋转延迟为 10 毫秒。对于 2KB 的页面，传输时间约为 0.009766 毫秒，总共约为 10.009766 毫秒。加载 32 个这样的页面将需要约 320.21 毫秒。对于 4KB 的页面，传输时间增加一倍，约为 0.01953 毫秒，所以每个页面的总时间为 10.01953 毫秒。加载 16 个这样的页面将需要约 160.3125 毫秒。对于如此快速的硬盘来说，唯一重要的是减少传输次数（或将页面连续放置在硬盘上）。

36. NRU 移除页面 2，FIFO 移除页面 3，LRU 移除页面 1，Second chance 移除页面 2。

37. 共享页面会引起各种复杂问题和选项：
(a) 如果进程 B 永远不会访问共享页面，或者当页面再次被交换出时访问它，应延迟进行进程 B 的页表更新。不幸的是，在一般情况下，我们不知道进程 B 将来会做什么。
(b) 代价是这种懒惰的页面故障处理可能会导致更多的页面故障。每个页面故障的开销在确定这种策

略是否更高效方面起着重要作用。（另外：这种代价与支持某些 UNIX fork 系统调用实现的写时复制策略类似。）

38. 碎片 B，因为代码具有比碎片 A 更高的空间局部性。内循环每隔一个外循环迭代才会引起一个页面故障（总共只有 32 个页面故障）。【另外（碎片 A）：由于一个帧是 128 个字，X 数组的一行占用半个页面（即 64 个字）。整个数组可以容纳 64×32/128=16 个帧。代码的内循环按列顺序遍历 X 的连续行。因此，对 X[i][j] 的每隔一个引用将引起一个页面故障。总的页面故障数将为 64×64/2=2,048】。

39. 这肯定是可以做到的。
    (a) 这种方法与在智能手机中使用闪存作为页面设备类似，不同之处是虚拟交换区是位于远程服务器上的 RAM。虚拟交换区的所有软件基础设施都必须被开发。
    (b) 这种方法可能是值得的，需要注意的是，磁盘驱动器的访问时间在毫秒级别，而通过网络连接访问 RAM 的访问时间在微秒级别，前提是软件开销不会太高。但是这种方法可能只在服务器群中存在大量空闲 RAM 的情况下才有意义。还有一个问题是可靠性。由于 RAM 是易失性的，如果远程服务器崩溃，虚拟交换区将丢失。

40. PDP-1 分页鼓的优点是没有旋转延迟。这样每次写入内存时可以节省一半的旋转时间。

41. 此程序有 8 页文本，5 页数据和 4 页堆栈。该程序不适合，因为它需要 17 个 4096 字节页面。使用 512 字节页面的情况不同。在这种情况下，文本为 64 页，数据为 33 页，堆栈为 31 页，总共为 128 个 512 字节页面，适合存放。小页面大小是可以的，但大页面大小就不行。

42. 程序发生了 15000 次页面故障，每次页面故障使用了额外的 2 毫秒处理时间。因此，页面故障的开销总共为 30 秒。这意味着在使用的 60 秒中，有一半的时间用于页面故障开销，另一半用于运行程序。如果我们使用两倍的内存来运行程序，那么页面故障的数量减半，只有 15 秒的页面故障开销，因此总运行时间将为 45 秒。

43. 如果程序无法修改，则该解决方案适用于程序。如果数据无法修改，则该解决方案适用于数据。然而，程序通常无法被修改，而数据几乎不可能无法被修改。如果二进制文件上的数据区被覆盖为更新的页面，下次启动程序时将无法获得原始数据。

44. 指令可能跨越页面边界，导致需要两次缺页错误来获取指令。要获取的单词也可能跨越页面边界，导致另外两次缺页错误，总共四次。如果内存中的单词必须对齐，数据单词只会导致一次缺页错误，但在具有 4KB 页面的机器上，加载地址为 4094 处的 32 位单词的指令在某些机器上（包括 x86）是合法的。

45. 当最后一个分配单元未被充分利用时，发生内部碎片。当两个分配单元之间有空间被浪费时，发生外部碎片。在分页系统中，在最后一页的浪费空间将导致内部碎片。在纯分段系统中，分段之间总会有一些空间浪费。这是由于外部碎片。

46. 不可以。搜索键使用了段号和虚拟页号，因此可以在一次匹配中找到精确的页面。

47. 这是结果：

	地址	**错误?**
(a)	(14, 3)	无 (或 0xD3 或 1110 0011)
(b)	无	保护错误：写入读取/执行段
(c)	无	页面错误
(d)	无	保护错误：跳转到读/写段

1. 当所有应用程序的内存需求可预知且可控制时，不需要普通的虚拟内存支持。一些例子包括智能卡、专用处理器（例如网络处理器）和嵌入式处理器。在这些情况下，我们应当始终考虑使用更多的真实内存。如果操作系统不需要支持虚拟内存，代码会更简单、更小。另一方面，虚拟内存的一些思想可能仍然能够有利地利用，尽管需要有不同的设计要求。例如，程序 / 线程的隔离可以使用闪存进行页面交换。
2. 这个问题涉及虚拟机支持的一个方面。近期的尝试包括 Denali、Xen 和 VMware。关键问题是如何实现接近本机性能，即仿佛执行的操作系统独占了内存。问题在于如何快速切换到另一个操作系统，因此如何处理 TLB。通常情况下，您希望为每个内核分配一定数量的 TLB 条目，并确保每个内核在其适当的虚拟内存上下文中运行。但是，有时硬件（例如某些 Intel 架构）希望在不知道您的操作意图的情况下处理 TLB 缺失。因此，您需要在软件中处理 TLB 缺失，或者提供用于为 TLB 条目附加上下文 ID 的硬件支持。

第 4 章习题解答

1. 可以使用 "..." 无限制地在树上上下移动。其中一些路径为
   /etc/passwd
   /./etc/passwd
   /./../etc/passwd
   /./../../etc/passwd
   /etc/../etc/passwd
   /etc/../etc/../etc/passwd
   /etc/../etc/../etc/../etc/passwd
   /etc/../etc/../etc/../etc/../etc/passwd
2. Windows 的方式是使用文件扩展名。每个扩展名对应一个文件类型和一个处理该类型的程序。另一种方式是记住创建文件的程序，并运行该程序。Macintosh 也是这样工作的。
3. 这些系统直接将程序加载到内存中，并从字 0 开始执行，而字 0 是魔术数字。为了避免尝试将头部作为代码执行，魔术数字是一个带有目标地址的 BRANCH 指令，该目标地址正好在头部之上。通过这种方式，可以直接将二进制文件读入新进程的地址空间，并在字 0 处运行，甚至不知道头部的大小。
4. 首先，如果没有打开，每次读取都需要指定要打开的文件名。系统然后必须获取该文件的 i-node，虽然可以将其缓存。一个快速出现的问题是何时将 i-node 刷新回磁盘。然而，它可能会超时。这可能有点笨拙，但可能会起作用。
5. 不，如果要再次读取文件，只需随机访问字节 0。
6. 是的。重命名调用不会更改创建时间或最后修改时间，但创建一个新文件会将当前时间作为创建时间和最后修改时间。此外，如果磁盘几乎满了，复制可能失败。
7. 映射部分的文件必须从页面边界开始，并且长度必须是页面数的整数倍。每个映射的页面使用文件本身作为支持存储。未映射的内存使用一个临时文件或分区作为支持存储。
8. 使用像 */usr/ast/file* 这样的文件名。虽然看起来像是一个层次路径名，但实际上它只是一个包含嵌入斜杠的单个名称。
9. 一种方法是给读取系统调用添加一个额外的参数，用于指示从哪个地址读取。实际上，每次读取都有可能在文件内进行查找。这种方案的缺点是：(1) 每次读取调用都需要额外的参数，以及 (2) 需要用户跟踪文件指针的位置。
10. 点点组件将搜索移到 "/usr" 目录，所以 "./ast" 将它放在 "/usr/ast"。因此，"./ast/x" 等同于 "/usr/ast/x"。
11. 由于浪费的存储空间是在分配单元（文件）之间而不是其中，这是外部碎片。这与使用交换系统或纯分段的主存器发生的外部碎片化现象非常相似。

12. 在连续分配系统中，如果数据块损坏，只有这个块受影响；可以读取文件的其他块。在链式分配中，无法读取损坏的块；同时，从该损坏块开始的所有块的位置数据都丢失。在索引分配的情况下，只有损坏的数据块受影响。

13. 启动传输需要 9 毫秒。以每秒 80 MB 的传输速率读取 2^{13} 字节需要 0.0977 毫秒，共计 9.0977 毫秒。写回文件需要额外的 9.0977 毫秒。因此，复制一个文件需要 18.1954 毫秒。将 16 GB 硬盘的一半进行整理将涉及复制 8 GB 存储空间，即 2^{20} 个文件。每个文件需要 18.1954 毫秒，这将耗时 19,079.25 秒，即 5.3 小时。显然，在每次删除文件后整理磁盘不是一个好主意。

14. 如果正确执行，是的。在整理过程中，每个文件应该被组织成所有块连续，以便快速访问。Windows 有一个程序可以对磁盘进行碎片整理和重新组织。鼓励用户定期运行它以提高系统性能。但考虑到需要的时间，每月运行一次可能是一个好的频率。

15. 数码相机在非易失性存储介质（例如闪存存储器）上按顺序记录若干张照片。当相机被重置时，介质被清空。此后，照片按顺序逐一记录，直到介质已满，然后它们被上传到硬盘上。对于这种应用，相机内部的连续文件系统（例如在图片存储介质上）是理想的。

16. 间接块可以容纳 128 个磁盘地址。加上 10 个直接磁盘地址，最大的文件有 138 个块。由于每个块是 1 KB，所以最大的文件大小是 138 KB。

17. 对于随机访问，表 / 索引和连续分配都是合适的，而链式分配则不太适合，因为它通常需要对给定记录进行多次磁盘读取。

18. 由于文件大小变化很大，连续分配将是低效的，需要在文件增大时重新分配磁盘空间，在文件缩小时进行碎片整理。链式和表 / 索引分配都将是高效的；在这两种方法间，表 / 索引分配对于随机访问场景更为高效。

19. 必须有一种方式来表示地址块指针存储的是数据，而不是指针。如果在属性中还剩下一个比特，可以使用它。这样就可以为数据保留所有九个指针。如果指针每个占据 k 字节，存储的文件最长可以达到 9k 字节。如果属性中没有剩下的比特，第一个磁盘地址可以存储一个无效地址，以标记后面的字节是数据而不是指针。在这种情况下，最大的文件长度为 8k 字节。

20. Elinor 是正确的。在表中同时存在两个 i-node 副本是灾难性的，除非两者都是只读的。最糟糕的情况是两者同时被更新。当 i-node 被写回磁盘时，最后被写入的一个将抹去另一个所做的更改，并且磁盘块将会丢失。

21. 硬链接不需要额外的磁盘空间，只需要在 i-node 中添加一个计数器来跟踪链接的数量。符号链接需要空间来存储所指向的文件的名称。符号链接可以指向其他机器上甚至是通过互联网的文件。而硬链接只能指向自己所在分区内的文件。

22. 单个 i-node 被给定文件的所有硬链接的目录条目指向。对于软链接，将为软链接创建一个新的 i-node，并且该 i-node 实际上指向原始的文件。

23. 磁盘上的块数 = 4 TB / 4 KB = 2[30]。因此，每个块地址可以是 32 位（4 字节），最接近的 2 的幂次。因此，每个块可以存储 4 KB / 4 = 1024 个地址。

24. 位图需要 B 位。自由列表需要 DF 位。如果 DF < B ，则自由列表需要更少的位数。或者，如果 F/B < 1/ D ，则自由列表较短，其中 F/B 是空闲块比例。对于 16 位磁盘地址，如果磁盘上空闲块占比 6% 或更少，自由列表将较短。

25. 位图的开头如下：
    (a) 写入文件 B 后：1111 1111 1111 0000
    (b) 删除文件 A 后：1000 0001 1111 0000
    (c) 写入文件 C 后：1111 1111 1111 1100
    (d) 删除文件 B 后：1111 1110 0000 1100

26. 这并不是一个严重的问题。修复方法很简单，只需要时间。恢复算法是将所有文件中的所有块列出，并将其补集作为新的空闲列表。在 UNIX 中，可以通过扫描所有 i 节点来完成这个操作。在 FAT 文件系统中，这个问题不会发生，因为没有空闲列表。但即使有，恢复它所需的步骤也很简单，只需要扫描 FAT 寻找空闲条目。

27. 奥利的论文可能没有得到他希望的可靠的备份。备份程序可能会跳过当前正在写入的文件，因为这样的文件中的数据状态可能不确定。

28. 他们必须在硬盘上的一个文件中跟踪上次备份的时间。每次备份时，在该文件中追加一个条目。备份时，读取该文件并记录上次条目的时间。任何自此时间以来发生了更改的文件都会被备份。

29. 在 (a) 和 (b) 中，21 不会被标记。在 (c) 中，不会有任何变化。在 (d) 中，21 不会被标记。

30. 许多 UNIX 文件都很短。如果整个文件适合与 i 节点相同的块中，只需要一次磁盘访问来读取文件，而不是目前的两次。即使对于较长的文件，也会有所收益，因为需要更少的磁盘访问。

31. 这不应该发生，但由于某个地方存在错误，这种情况可能发生。这意味着某个块在两个文件中出现，并且还在空闲列表中出现两次。修复错误的第一步是从空闲列表中删除两个副本。接下来，需要获取一个空闲块，并将有问题的块的内容复制到那里。最后，将块在其中一个文件中的出现更改为引用这个新获取的块的副本。此时，系统再次恢复一致。

32. 所需的时间为 h+40\times(1-h) 。绘图是一条直线。

33. 在这种情况下，最好使用写通写 (cache write-through) 缓存，因为它在更新缓存的同时也将数据写入硬盘。这将确保即使用户在磁盘同步完成之前意外移除了硬盘，更新后的文件仍然在外部硬盘上。

34. 块预读技术是预先按顺序读取块，以提高性能。在这种应用中，由于用户可以在给定时刻输入任何学生 ID，所以很可能不会按顺序访问记录。因此，在这种情况下，块预读技术不会很有用。

35. 分配给 f1 的块是：22、19、15、17、21。
    分配给 f2 的块是：16、23、14、18、20。

36. 在 15,000 转 / 分钟的情况下，磁盘需要 4 毫秒才能完成一次旋转。读取 k 字节的平均访问时间（以毫秒为单位）为 6+2+(k/1,048,576)\times 4 。对于 1 KB、2 KB 和 4 KB 的块，访问时间分别约为 6.0039 毫秒、6.0078 毫秒和 6.0156 毫秒（几乎没有太大差异）。这相应地提供了大约 170.556 KB/sec、340.890 KB/sec 和 680.896 KB/sec 的速率。

37. 如果所有文件都是 1 KB 大小，那么每个 4 KB 的块将包含一个文件和 3 KB 的浪费空间。将两个文件放入一个块中是不允许的，因为用于跟踪数据的单位是块，而不是半块。这导致了 75% 的浪费空间。实际上，每个文件系统都有大文件以及许多小文件，这些文件能更有效地使用磁盘空间。例如，一个 32,769 字节的文件将使用 9 个磁盘块进行存储，给出了 32,769 / 36,864 的空间利用率，约为 89%。

38. 间接块可以容纳 1024 个地址。加上 10 个直接地址，总共有 1034 个地址。由于每个地址都指向一个 4 KB 的磁盘块，所以最大文件大小为 4,235,264 字节。

39. 它限制了所有文件长度之和不能超过磁盘大小。这不是一个非常严重的限制。如果文件总大小超过了磁盘大小，那么就没有地方将它们全部存储在磁盘上。

40. i-node 包含 10 个指针。单个间接块可容纳 1024 个指针。双重间接块可容纳 1024 {}^{2} 个指针。三重间接块可容纳 1024 {}^{3} 个指针。将它们加在一起，我们得到最大文件大小为 1,074,791,434 个块，约为 16.06 GB。

41. 需要进行以下磁盘读取：
    “/” 目录
    “/usr” 的 i-node
    “/usr” 目录
    “/usr/ast” 的 i-node
    “/usr/ast” 目录
    “/usr/last/courses” 的 i-node
    “/usr/last/courses” 目录
    “/usr/last/courses/os” 的 i-node
    “/usr/last/courses/os” 目录

"/usr/last/courses/os/handout.t" 的 i-node
总共需要进行 10 次磁盘读取。

42. 一些优点如下：首先，不会浪费磁盘空间来存放未使用的 i-node。其次，不可能用尽 i-node。第三，由于 i-node 和初始数据可以一次性读取，所以需要的磁盘移动较少。一些缺点如下：首先，目录条目现在需要一个 32 位的磁盘地址，而不是 16 位的 i-node 号码。其次，即使对于不包含任何数据的文件（空文件、设备文件），也将使用整个磁盘。第三，文件系统完整性检查将会变慢，因为需要为每个 i-node 读取整个块，并且 i-node 将在整个磁盘上散布。第四，由于 i-node 的存在，那些已经精心设计以适配块大小的文件将不再适配块大小，从而影响性能。

第 5 章习题解答

1. 在这个图中，我们将控制器和设备作为独立的单元进行展示。原因是为了使一个控制器能够处理多个设备，从而消除了每个设备都需要一个控制器的需求。如果控制器几乎免费，那么将控制器内置到设备中会更简单。这种设计也可以实现并行多次传输，从而提供更好的性能。
2. 很简单。扫描仪的最大输出速度为 400 KB/ 秒。无线网络的运行速度为 6.75 MB/ 秒，所以完全没有问题。
3. 这不是个好主意。内存总线肯定比 I/O 总线快，否则为什么要花这个功夫呢？考虑一下正常的内存请求会发生什么。内存总线先完成，但是 I/O 总线仍在忙碌。如果 CPU 等待直到 I/O 总线完成，那么它降低了内存性能，变成了 I/O 总线的性能。如果它试图通过内存总线进行第二次引用，如果这次引用是对 I/O 设备的引用，那么就会失败。如果有一种方式可以即时中止前一个 I/O 总线引用来尝试第二个引用，这种改进可能会有效，但实际上没有这种选择。总的来说，这个主意不好。
4. 精确中断的一个优点是操作系统中的代码简单，因为机器状态是明确定义的。另一方面，在不精确中断中，操作系统编写者必须找出哪些指令已经部分执行，以及执行到什么程度。然而，精确中断会增加芯片设计和芯片面积的复杂性，这可能导致 CPU 变慢。
5. 每个总线事务都有一个请求和一个响应，每个事务需要 50 纳秒，或者每个总线事务需要 100 纳秒。这意味着每秒可以处理 1000 万个总线事务。如果每个事务有 4 个字节，那么总线必须处理 40 MB/ 秒。这些事务可能以轮流的方式分布在五个 I/O 设备上是无关紧要的。总线事务需要 100 纳秒，无论连续的请求是针对同一设备还是不同设备，所以 DMA 控制器的通道数量并不重要。总线并不知道也不关心这些。
6. (a) 按字方式： $1000\times[(t_{1}+t_{2})+(t_{1}+t_{2})+(t_{1}+t_{2})]$ 其中第一项是用于获取总线并将命令发送到磁盘控制器，第二项是用于传输字，第三项是用于确认。总共需要 $3000\times(t_{1}+t_{2})$ 纳秒。
(b) 突发方式： $(t_{1}+t_{2})+t_{1+1000\ 次}\ t_{2}+(t_{1}+t_{2})$ ，其中第一项是获取总线并将命令发送到磁盘控制器，第二项是磁盘控制器获取总线，第三项是突发传输，第四项是获取总线并进行确认。总共需要 $3t_{1}+1002t_{2}$ 。
7. 内存到内存的复制可以通过首先发出读命令将字从内存传输到 DMA 控制器，然后发出写命令将字从 DMA 控制器传输到内存中的不同地址来实现。这种方法的优点是 CPU 可以并行进行其他有用的工作。缺点是这种内存到内存的复制可能会很慢，因为 DMA 控制器比 CPU 要慢得多，并且数据传输是通过系统总线而不是专用的 CPU-内存总线进行的。
8. 一个中断需要将 34 个字推送到堆栈上。从中断返回需要从堆栈中提取 34 个字。仅仅这个开销就需要 340 纳秒。因此，每秒最多处理的中断数量不超过大约 294 万个，假设每个中断都不需要执行任何工作。
9. 现代 CPU 的执行速率取决于每秒完成的指令数量，与指令的执行时间关系不大。如果一个 CPU 每秒可以完成 10 亿条指令，即使一条指令需要 30 纳秒，那么它也是一个 1000 MIPS 的机器。因此，通常没有太多的尝试使指令快速完成。直到当前执行的最后一条指令完成之前都保持中断可能会显著增加中断的延迟。此外，为了正确实现这一点，需要一些管理工作。

10. 在初始阶段就可以完成这个任务。选择在最后执行的原因是中断服务过程的代码非常简短。通过先输出另一个字符，然后确认中断，在另一个中断发生时，打印机将在中断期间工作，使得打印速度稍微提高。这种方法的劣势是在其他中断可能被禁用的时候，稍微存在较长的空闲时间。

11. 是的，堆栈中的程序计数器指向尚未获取的第一条指令。在此之前的所有指令都已执行，指向的指令及其后续指令尚未执行。这符合准确中断的条件。在拥有单个流水线的机器上实现准确中断并不难。问题出在指令的乱序执行，而这里并非如此。

12. 打印机每分钟可以打印 50 \times 80 \times 6 = 24,000 个字符，即每秒 400 个字符。每个字符在中断处理中需要 50 微秒的 CPU 时间，因此在每秒钟中，中断开销为 20 毫秒。使用中断驱动的输入 / 输出，剩余的 980 毫秒可以用于其他工作。换句话说，中断开销仅占 CPU 的 2%，几乎不会对运行中的程序产生影响。

13. UNIX 的做法如下。有一个由设备号索引的表格，其中每个表项都是一个 C 结构，包含指向设备的打开、关闭、读取和写入函数以及其他一些内容的指针。要安装一个新设备，需要在此表格中添加一个新表项，并填充指针，通常是指向新加载的设备驱动程序。

14. (a) 设备驱动程序。
    (b) 设备驱动程序。
    (c) 与设备无关的软件。
    (d) 用户级软件。

15. 在此过程中，一个数据包必须被复制四次，耗时 4.1 毫秒。还有两个中断，共计 2 毫秒。最后，传输时间为 0.83 毫秒，总共为每 1024 字节 6.93 毫秒。因此，最大数据传输速率为 147,763 字节 / 秒，约占名义的 10 兆位 / 秒网络容量的 12%。（如果考虑协议开销，这些数字会更糟。）

16. 如果在输出出现时立即分配打印机，一个进程可能通过打印几个字符然后睡眠一周来占用打印机。

17. 磁盘每秒旋转 120 次，因此旋转一周需要 1000/120 毫秒。每圈有 200 个扇区，每个扇区的时间为该数字的 1/200 或 5/120 = 1/24 毫秒。在 1 毫秒的寻道时间内，磁头下方通过了 24 个扇区。因此，柱面错位应为 24。

18. 在 7200 RPM 时，每秒有 120 次旋转，因此一次旋转大约需要 8.33 毫秒。将此除以 500，得到约 16.67 微秒的扇区时间。

19. 每秒有 120 次旋转。在其中一次旋转中， 500 \times 512 字节通过磁头。因此，磁盘每圈可以读取 256,000 字节，即每秒读取 30,720,000 字节。

20. RAID 2 级别不仅可以从故障驱动器中恢复，还可以从未被检测到的瞬态错误中恢复。如果一个驱动器传递了一个错误的位，RAID 2 级别可以纠正，但 RAID 3 级别不能纠正。

21. 发生 0 次故障的概率 P*{0} 是 (1-p)^{k} 。发生 1 次故障的概率 P*{1} 是 kp(1-p)^{k-1} 。RAID 故障的概率是 1-P*{0}-P*{1} 。即 1-(1-p)^{k}-kp(1-p)^{k-1} 。

22. 读取性能：RAID 0、2、3、4 和 5 级别允许并行读取以服务一个读取请求。然而，RAID 1 级别进一步允许两个读取请求同时进行。写入性能：所有的 RAID 级别提供类似的写入性能。空间开销：级别 0 没有空间开销，级别 1 的空间开销为 100%。对于 32 位数据字和六个奇偶校验盘，级别 2 的空间开销约为 18.75%。对于 32 位数据字，级别 3 的空间开销约为 3.13%。最后，假设级别 4 和 5 中有 33 个驱动器，它们的空间开销为 3.13%。可靠性：级别 0 没有可靠性支持。所有其他的 RAID 级别可以经受住一次硬盘崩溃。此外，在级别 3、4 和 5 中，一个字中的单个随机位错误可以被检测到，而在级别 2 中，一个字中的单个随机位错误可以被检测和纠正。

23. 1 个斯比字节等于 2^{70} 字节；1 个皮比字节等于 2^{50} 字节。1 个斯比字节可以容纳 2^{20} 个皮比字节。

24. 在两极之间会产生一个磁场。不仅制造出一个磁场的源很难变小，而且磁场的扩散速度很快，这导致了保持磁介质表面靠近磁源或传感器的机械问题。半导体激光器可以在一个非常小的地方产生光，并且可以通过光学手段将光聚焦到距离源较远的非常小的点上。

25. 光盘的主要优势是其比磁盘具有更高的记录密度。磁盘的主要优势是其比光盘快一个数量级。

26. 可能。如果大多数文件存储在逻辑连续的扇区中，对扇区进行交错以使程序有时间处理刚刚接收到的数据，这样当下一个请求发出时，硬盘就处于正确位置。是否值得这样做取决于运行的程序类型以及它们的行为是否一致。

27. 可能是，也可能不是。双重交错实际上是两个扇区的柱面错位。如果磁头在两个扇区时间内可以进行一个柱面到另一个柱面的寻道，那么不需要额外的柱面错位。如果不能，那么需要额外的柱面错位以避免在寻道后错过一个扇区。

28. (a) 一个区域的容量是 轨道数 \times 柱面数 \times 每柱面扇区数 \times 每扇区字节数。
区域 1 的容量为 16 \times 100 \times 160 \times 512 = 131072000 字节。
区域 2 的容量为 16 \times 100 \times 200 \times 512 = 163840000 字节。
区域 3 的容量为 16 \times 100 \times 240 \times 512 = 196608000 字节。
区域 4 的容量为 16 \times 100 \times 280 \times 512 = 229376000 字节。
总和 = 131072000 + 163840000 + 196608000 + 229376000 = 720896000。
(b) 7200 转 / 分钟意味着每秒有 120 转。在 1 毫秒的寻道时间内，有 0.120 的扇区被覆盖。在区域 1 内，磁头将在 1 毫秒内经过 0.120\times 160 个扇区，因此，区域 1 的最佳柱面错位为 19.2 个扇区。在区域 2 内，磁头将在 1 毫秒内经过 0.120\times 200 个扇区，因此，区域 2 的最佳柱面错位为 24 个扇区。在区域 3 内，磁头将在 1 毫秒内经过 0.120\times 240 个扇区，因此，区域 3 的最佳柱面错位为 28.8 个扇区。在区域 4 内，磁头将在 1 毫秒内经过 0.120\times 280 个扇区，因此，区域 4 的最佳柱面错位为 33.6 个扇区。
(c) 最大数据传输速率将在读取 / 写入最外圈（区域 4）的柱面时实现。在该区域，每秒读取 120 次，每次读取 280 个扇区。因此，数据传输速率为 280 \times 120 \times 512 = 17,203,200 字节 / 秒。

29. 驱动器容量和传输速率加倍。搜索时间和平均旋转延迟保持不变。没有属性变差。

30. 一个相当明显的结果是，没有现有操作系统可以工作，因为它们都会查看这里以确定磁盘分区的位置。更改分区表的格式将导致所有操作系统失效。更改分区表的唯一方法是同时更改所有操作系统以使用新的格式。

31. (a) 10 + 12 + 2 + 18 + 38 + 34 + 32 = 146 柱面 = 876 毫秒。
(b) 0 + 2 + 12 + 4 + 4 + 36 + 2 = 60 柱面 = 360 毫秒。
(c) 0 + 2 + 16 + 2 + 30 + 4 + 4 = 58 柱面 = 348 毫秒。

32. 在电梯算法中，最坏情况下，读 / 写请求的响应时间将几乎是两个完整的磁盘扫描，而在修改后的算法中，响应时间最多为一个完整的磁盘扫描。

33. 单次模式的缺点是中断处理程序所消耗的时间未计算在内，因为在此期间减小计数器的过程会暂停。方波模式的缺点是高时钟频率可能导致多个中断在前一个中断完成之前引发。

34. 不一定。一个读取 10,000 个块的 UNIX 程序会逐个发出请求，在每个请求发出后阻塞，直到完成。因此，磁盘驱动器只能看到一次请求；它无法做任何事情，只能按照到达的顺序处理它们。Harry 应该同时启动多个进程，看看电梯算法是否有效。

35. 是有竞争，但这并不重要。由于稳定写本身已经完成，非易失性 RAM 未被更新仅意味着恢复程序将知道在写哪个块。它将读取两个副本。找到它们相同，它将不会更改任何一个，这是正确的操作。在非易失性 RAM 更新之前发生崩溃的影响只意味着恢复程序将比它应该的多进行两次磁盘读取。

36. 是的，即使 CPU 在恢复过程中崩溃，磁盘仍然保持一致。考虑图 5-31。图 (a) 和图 (e) 都不涉及恢复。假设在图 (b) 中的恢复过程中 CPU 崩溃。如果 CPU 在将来自驱动器 2 的块完全复制到驱动器 1 之前崩溃，情况与以前相同。随后的恢复过程将在驱动器 1 中检测到 ECC 错误，并再次将块从驱动器 2 复制到驱动器 1。如果 CPU 在将来自驱动器 2 的块复制到驱动器 1 后崩溃，情况与图 (e) 中的情况相同。假设在图 (c) 中的恢复过程中 CPU 崩溃。如果 CPU 在将来自驱动器 1 的块完全复制到驱动器 2 之前崩溃，情况与图 (d) 中的情况相同。随后的恢复过程将在驱动器 2 中检测到 ECC 错误，并将块从驱动器 1 复制到驱动器 2。如果 CPU 在将来自驱动器 1 的块复制到驱动器 2 后崩溃，情况与图 (e) 中的情况相同。最后，假设在图 (d) 中的恢复过程中 CPU 崩溃。如果 CPU 在将来自驱

动器 1 的块完全复制到驱动器 2 之前崩溃，情况与以前相同。随后的恢复过程将在驱动器 2 中检测到 ECC 错误，并再次将块从驱动器 1 复制到驱动器 2。如果 CPU 在将来自驱动器 1 的块复制到驱动器 2 后崩溃，情况与图 (e) 中的情况相同。

37. 在图 5-27(b) 和 5-27(d) 所示的场景中会出现问题，因为如果损坏块的 ECC 正确，则可能与情景 5-27(c) 看起来相同。在这种情况下，无法检测到哪个磁盘包含有效的（旧的或新的）块，因此无法进行恢复。

38. 每秒两毫秒，每秒 60 次等于 120 毫秒 / 秒，或者占 CPU 的 12%。

39. 在这些参数下，
    (a) 使用一个 500 MHz 的晶体，计数器可以每 2 纳秒递减一次。所以，为了每毫秒有一个计时信号，寄存器的值应为 1000000/2 = 500,000。
    (b) 为了每 100 微秒有一个时钟信号，保持寄存器的值应该是 50,000。

40. 在 5000 时刻：当前时间 = 5000；下一个信号 = 8；报头 → 8 → 4 → 3 → 14 → 8。
    在 5005 时刻：当前时间 = 5005；下一个信号 = 3；报头 → 3 → 4 → 3 → 14 → 8。
    在 5013 时刻：当前时间 = 5013；下一个信号 = 2；报头 2 → 14 → 8。
    在 5023 时刻：当前时间 = 5023；下一个信号 = 6；报头 → 6 → 4 → 5。

41. 一年的秒数是 365.25\times 24\times 3600 。这个数字是 31,557,600。从 1970 年 1 月 1 日开始，计数器在经过 2^{32} 秒后会循环。 2^{32} /31,557,600 的值是 136.1 年，所以循环将发生在 2106.1 年初的二月。当然，到那时，所有的计算机都将至少是 64 位的，所以根本不会发生循环。

42. 滚动窗口需要复制 79 行 80 个字符或 6320 个字符。复制 1 个字符（16 字节）需要 800 纳秒，所以整个窗口需要 5.056 毫秒。将 80 个字符写到屏幕上需要 400 纳秒，所以滚动并显示一个新行需要 5.456 毫秒。这样每秒可以达到大约 183.2 行。

43. 假设用户不小心要求编辑器打印成千上万行的文本。然后他按下 DEL 键来停止打印。如果驱动程序不丢弃输出，输出可能会在按下 DEL 键后继续几秒钟，这会使用户一次又一次地按下 DEL 键，并因为没有发生任何动作而感到沮丧。

44. 它应该将光标移动到第 5 行的第 7 个位置，然后删除 6 个字符。序列是 ESC [5 ; 7 H ESC [6 P。

45. 鼠标最大移动速率为 200 毫米 / 秒，相当于 2000 个鼠标事件 / 秒。如果每个报告是 3 个字节，则输出速率为 6000 个字节 / 秒。

46. 在 24 位颜色系统中，只能表示 {2}^{24} 种颜色。这还不是全部。例如，假设一位摄影师拍摄了 300 罐纯蓝色油漆，每罐油漆含有稍微不同的颜料量。第一罐可能由 (R, G, B) 值 (0, 0, 1) 表示。下一罐可能由 (0, 0, 2) 表示，依此类推。由于 B 坐标只有 8 位，无法表示 300 种纯蓝色的不同值。一些照片将以错误的颜色显示。另一个例子是颜色 (120.24, 150.47, 135.89)。它无法被表示，只能近似表示为 (120, 150, 136)。

47. (a) 每个像素在 RGB 中占据 3 个字节，所以表格空间为 16\times 24\times 3 字节，即 1152 字节。
    (b) 每个字符需要 100 纳秒，所以每个字符需要 115.2 微秒。这样输出速率约为 8681 个字符 / 秒。

48. 重写文本屏幕需要复制 2000 字节，可以在 4 微秒内完成。重写图形屏幕需要复制 1024\times 768\times 3 = 2,359,296 字节，大约需要 4.72 毫秒。

49. 在 Windows 中，操作系统自己调用处理程序。在 X Windows 中，没有类似这样的情况发生。X 只接收消息并在内部处理它。

50. 第一个参数是必要的。首先，坐标是相对于某个窗口的，所以需要 *hdc* 指定窗口，从而指定原点。其次，如果矩形落在窗口外部，它将被剪裁，所以需要窗口坐标。第三，矩形的颜色和其他属性取自由 *hdc* 指定的上下文，这是非常重要的。

51. 显示尺寸为 400\times 160\times 3 字节，即 192,000 字节。以每秒 10 帧的速度计算，每秒传输 1,920,000 字节或 15,360,000 位。这占据了快速以太网的 15%。

52. 网络段上的带宽是共享的，所以在 1Mbps 网络上同时请求不同数据的 100 个用户将会看到每个人的有效速度为 10Kbps。在共享网络上，可以进行多播电视节目，因此不管有多少用户，视频数据包仅广播一次，这在很大程度上是有效的。当有 100 个用户在浏览网页时，每个用户将获得 1/100 的带宽，因此性能可能会迅速下降。

53. 薄客户机的优点包括低成本和不需要为客户端进行复杂管理。缺点包括（潜在的）由于网络延迟而导致的性能降低以及（潜在的）隐私丧失，因为客户端的数据 / 信息与服务器共享。

54. 如果 n=10 ，CPU 仍然可以按时完成工作，但能耗显著减少。如果在全速运行 1 秒钟期间的能耗为 E ，那么全速运行 100 毫秒然后闲置 900 毫秒的能耗为 E /10。以 1/10 速度运行 1 秒钟的能耗为 E /100，节约了 9E /100。通过降低电压来节省的百分比为 90%。

55. 窗口系统在显示上使用的内存比文本模式多，并且更多地使用虚拟内存。这使得硬盘不太可能因为一段时间的不活动而自动关闭电源。

第 6 章习题解答

1. 在美国，考虑一个总统选举，有三个或更多候选人争取某个党派的提名。在完成所有初选后，当代表们抵达党派大会时，可能出现没有候选人获得多数票的情况，也没有代表愿意改变自己的投票。这是一种僵局。每个候选人都有一些资源（选票），但需要更多资源来完成任务。在议会中有多个政党的国家中，每个政党支持不同版本的年度预算，可能无法组成多数通过预算。这也是一种僵局。

2. 储存分区上的磁盘空间是有限资源。每个数据块的到来实际上占用了一定的资源，而每个新到达的数据块都需要更多的资源。例如，如果储存空间为 10MB，并且前五个 2MB 的作业的前半部分到达，磁盘将变满，无法存储更多数据块，这就产生了僵局。可以通过允许作业在完全储存之前开始打印，并保留释放出来的空间给该作业的其余部分来避免僵局。这样，一个作业将完全打印，然后下一个作业可以做同样的事情。如果作业必须等到完全储存后才能开始打印，就可能发生僵局。

3. 打印机是不可抢占的；系统在完成前一个作业之前无法开始打印另一个作业。储存磁盘是可抢占的；可以删除一个不完整且过大的文件，并要求用户稍后发送，前提是协议允许。

4. 是的。这完全没有任何区别。

5. 假设有三个进程 A、B 和 C，以及两种资源类型 R 和 S。进一步假设资源 R 有一个实例，资源 S 有两个实例。考虑以下执行场景：A 请求 R 并得到，B 请求 S 并得到，C 请求 S 并得到（有两个 S 实例），B 请求 R 并被阻塞，A 请求 S 并被阻塞。在这个阶段，满足四个条件。然而，没有发生死锁。当 C 完成时，释放了一个 S 实例，分配给了 A。现在 A 可以完成执行并释放 R，然后 R 可以被分配给 B，B 可以完成执行。如果每种类型的资源有一个实例，这四个条件就足够了。

6. "Don't block the box" 是一个预分配策略，它否定了持有和等待死锁的先决条件，因为我们假设车辆可以进入交叉口后的街道空间，从而释放交叉口。另一种策略可能是让车辆暂时进入停车场并释放足够的空间来消除交通阻塞。一些城市有交通控制策略来调控交通；随着城市街道的拥堵程度加剧，交通监管人员会调整红灯的设置，以限制进入高度拥堵区域的交通。较轻的交通流量确保资源竞争较少，从而降低交通阻塞发生的可能性。

7. 上述异常不是一种通信死锁，因为这些车辆彼此独立，并且如果没有竞争，它们将以最小的延迟通过交叉口。这也不是一种资源死锁，因为没有一辆车持有其他车辆请求的资源。资源预分配或资源抢占机制也不能帮助控制这种异常。然而，这种异常是一种竞争同步的情况，在这种情况下，车辆在一个循环链中等待资源，交通控制可能是一种有效的控制策略。为了与资源死锁区分开，这种异常可以被称为“调度死锁”。类似的死锁可能发生在规定要求两列列车在共享铁路轨道上合并时等待另一列列车先行的情况。需要注意的是，一个警察示意其中一辆竞争的车辆或列车前行（而不是其他车辆或列车）可以打破这种死锁状态，而无需回滚或其他开销。

8. 当一个进程持有某一种资源的一些或全部单位，并请求另一种资源类型时，同时另一个进程持有第二种资源并请求第一种资源的可用单位时，如果其他进程无法释放第一种资源的单位且资源不能被抢占或并发使用，系统将发生死锁。例如，在实际内存系统中，两个进程都被分配了内存单元（我们假设不支持页面或进程的交换，但支持对内存的动态请求）。第一个进程锁定另一个资源，例如，一个数据单元。第二个进程请求被锁定的数据，并被阻塞。第一个进程需要更多内存以执行释放数据的代码。假设系统中没有其他进程可以完成并释放内存单元，系统中存在死锁。
9. 是的，存在非法的图。我们声明一个资源只能被单个进程持有。从一个资源方块到一个进程圆圈的弧表示该进程拥有该资源。因此，一个方块上有从它指向两个或多个进程的弧表示所有这些进程都持有该资源，这违反了规则。因此，除非有资源的多个副本，否则任何一个方块上有多个弧离开并指向不同圆圈的图都违反了规则。方块到方块或圆圈到圆圈的弧也违反了规则。
10. 这两个变化都不会导致死锁。两种情况下都不存在环形等待。
11. 考虑三个进程 A、B 和 C，两个资源 R 和 S。假设 A 正在等待 B 持有的 I，B 正在等待 A 持有的 S，C 正在等待 A 持有的 R。三个进程 A、B 和 C 都处于死锁状态。然而，只有 A 和 B 属于循环链。
12. 这显然是一种通信死锁，可以通过使 A 在一段时间后（启发式）超时并重新传输其使能消息（增加窗口大小的消息）来进行控制。然而，B 可能已经收到了原始消息和重复消息。如果窗口大小的更新是绝对值而不是差值给出的，不会发生任何损害。在这种消息上使用序列号也可以有效地检测到重复消息。
13. 所有这些资源中的一部分可以被保留，仅供管理员拥有的进程使用，以便他或她可以始终运行一个 shell 和用于评估死锁并决定要终止哪些进程以使系统再次可用的程序。
14. 首先，未标记的进程集合 P = (P1 P2 P3 P4)
    R1 不小于或等于 A
    R2 小于 A；标记 P2；A = (02031)；P = (P1 P3 P4)
    R1 不小于或等于 A
    R3 等于 A；标记 P3；A = (02032)；P = (P1 P4)
    R1 不小于或等于 A
    R4 不小于或等于 A
    因此，进程 P1 和 P4 仍未标记。它们处于死锁状态。
15. 通过抢占恢复：在进程 P2 和 P3 完成之后，可以强制进程 P1 抢占 1 个 RS3。这将使 A = (02132)，并允许进程 P4 完成。一旦 P4 完成并释放其资源，P1 可能会完成。通过回滚恢复：将 P1 回滚到在其获取 RS3 之前备份的状态。通过终止进程进行恢复：终止 P1。
16. 该进程正在请求比系统拥有的资源更多的资源。无论其他进程是否需要任何资源，它都无法获得这些资源，因此永远无法完成。
17. 如果系统有两个或更多个 CPU，两个或更多个进程可以并行运行，导致对角轨迹。
18. 是的。用三维空间进行整个过程。z 轴表示第三个进程执行的指令数量。
19. 使用该方法来指导调度，只有在预先知道资源将在何时被索取的确切时刻时才能使用。实际上，这种情况很少发生。
20. 存在一些既不安全也不死锁的状态，但这些状态会导致死锁的发生。例如，假设我们有四个资源：磁带、绘图仪、扫描仪和 CD-ROM，还有三个竞争这些资源的进程。我们可以有以下情况：

	Has	**Needs**	**Available**
A:	2 0 0 0	1 0 2 0	0 1 2 1
B:	1 0 0 0	0 1 3 1	
C:	0 1 2 1	1 0 1 0	

这个状态并不是死锁，因为仍然可以发生许多操作，例如，进程 A 仍可以获取两个打印机。然而，如果每个进程请求其剩余的资源需求，就会发生死锁。

1. 进程 D 的请求是不安全的，而进程 C 的请求是安全的。
2. 系统是没有死锁的。假设每个进程都有一个资源。还有一个资源是空闲的。任何一个进程都可以请求并获取该资源，这样它就可以完成并释放两个资源。因此，死锁是不可能的。
3. 如果一个进程拥有 m 个资源，则它可以完成并且不会参与死锁。因此，最坏情况是每个进程都有 m-1 个资源，并且需要另一个资源。如果还剩下一个资源，一个进程可以完成并释放所有资源，从而让其余进程也可以完成。因此，避免死锁的条件是 r >= p(m-1) + 1。
4. 不，进程 D 仍然可以完成。当它完成时，它会释放足够的资源来让进程 E（或 A）完成，依此类推。
5. 将矩阵中的一行与可用资源向量进行比较需要 m 次操作。这个步骤必须重复 n 次，以找到一个可以完成并标记为完成的进程。因此，将一个进程标记为完成需要大约 mn 步。对所有 n 个进程重复算法意味着步骤的数量是 mn2。因此，a=1，b=2。
6. 需求矩阵如下：
   $$
   \begin{array} \ 0 & 1 & 0 & 0 & 2 \ 0 & 2 & 1 & 0 & 0 \ 1 & 0 & 3 & 0 & 0 \ 0 & 0 & 1 & 1 & 1
   \end{array}
   $$

   如果 x 等于 0，我们立即发生死锁。如果 x 等于 1，进程 D 可以执行到完成。当它完成时，可用向量变为 1 1 2 2 1。不幸的是，我们现在已经死锁。如果 x 等于 2，在 D 运行后，可用向量为 1 1 3 2 1，进程 C 可以运行。在它完成并释放其资源后，可用向量为 2 2 3 3 1，这将允许进程 B 运行并完成，然后进程 A 运行并完成。因此，避免死锁的最小值 x 为 2。
7. 考虑一个需要将大型文件从磁带复制到打印机的进程。由于内存的大小有限，并且整个文件无法完全放入内存中，进程将不得不循环执行以下步骤，直到整个文件被打印完毕：
   获取磁带驱动器；
   将文件的下一部分复制到内存中（内存大小有限）；
   释放磁带驱动器；
   获取打印机；
   从内存中打印文件；
   释放打印机。
   这会延长进程的执行时间。此外，由于在每个打印步骤之后都释放打印机，无法保证文件的所有部分都会连续打印在一页上。
8. 假设进程 A 按照顺序请求记录 a、b、c。如果进程 B 也首先请求 a，其中一个进程会得到该记录，而另一个会被阻塞。这种情况始终不会发生死锁，因为获胜者现在可以完整地运行而不受干扰。对于其他四种组合，有些可能导致死锁，而有些可能不会。六种情况如下：
   a b c 不会发生死锁；
   a c b 不会发生死锁；
   b a c 可能会发生死锁；
   b c a 可能会发生死锁；
   c a b 可能会发生死锁；
   c b a 可能会发生死锁。
   由于六种情况中有四种可能导致死锁，因此避免死锁的机会为 1/3，发生死锁的机会为 2/3。
9. 是的。假设所有邮箱都是空的。现在 A 发送给 B 并等待回复，B 发送给 C 并等待回复，C 发送给 A 并等待回复。现在满足了通信死锁的所有条件。

10. 为了避免循环等待，给资源（账户）编号，按照账户号码的顺序进行加锁。读取输入行后，进程先锁定较小编号的账户，然后在获得锁定（可能需要等待）之后，锁定较大编号的账户。由于没有进程会等待比自己拥有的账户编号更低的账户，因此永远不会发生循环等待，因此也就永远不会发生死锁。

11. 修改请求新资源的语义如下：如果进程请求新资源并且该资源可用，则获得该资源并保留已有资源。如果新资源不可用，则释放所有已有资源。在这种情况下，死锁是不可能的，并且没有危险获得新资源但失去现有资源。当然，该进程只有在释放资源是可能的情况下（例如可以在页面之间释放扫描仪或在光盘之间释放记录器）才能正常工作。

12. 我给它一个 F（不及格）的评分。这个进程是做什么的？由于明显需要该资源，它只会再次请求并再次被阻塞。这比保持阻塞状态还要糟糕，在这种情况下，系统可能会记录竞争进程等待的时间有多长，并将新释放的资源分配给等待时间最长的进程。通过定期超时并重新尝试，进程会失去其优先级。

13. 虚拟存储器和时间分片系统主要是为了辅助系统用户而开发的。虚拟化硬件将用户从预先设置需求、资源分配和覆盖等细节中隔离出来，同时防止死锁。然而，上下文切换和中断处理的成本相当大。需要专门的寄存器、缓存和电路。可能不会仅仅为了防止死锁而承担这种成本。

14. 当一组进程被阻塞等待只有该组中其他某个进程可以引发的事件时，就会发生死锁。另一方面，处于活锁状态的进程并不会被阻塞。相反，它们继续执行，检查一个条件是否成为真，但该条件永远不会成为真。因此，除了它们所持有的资源之外，活锁状态的进程仍然会消耗宝贵的 CPU 时间。最后，进程被饥饿是因为存在其他进程以及大量新到来的进程，这些新进程的优先级比被饥饿的进程高。与死锁或活锁不同，饥饿可以自行终止，例如当具有更高优先级的现有进程终止并且没有新的高优先级进程到达时。

15. 这种死锁状态是竞争同步的异常情况，可以通过资源预分配来控制。然而，进程不会被阻塞于资源。此外，资源已经按照线性顺序请求。这个异常状态不是资源死锁，而是活锁。资源预分配将防止这种异常情况发生。作为一种启发式方法，如果进程在一定的时间间隔内无法完成，则可能会超时并释放其资源，然后休眠一段随机时间，然后重试。

16. 这里给出了答案，虽然有点复杂。
    (a) 这是一个竞争同步异常。它也是一个活锁。我们可以称之为调度活锁。它不是资源活锁或死锁，因为站点并没有持有其他站点所请求的资源，因此不存在一个循环的站点持有资源并请求其他资源的链条。它也不是通信死锁，因为站点在独立执行，并且如果按顺序进行调度，则会完成传输。
    (b) 以太网和时隙 Aloha 要求，在检测到自己的传输冲突后，站点必须等待一定数量的时间槽后重新发送。选择时间槽的间隔在每次冲突后加倍，根据流量负载动态调整。连续重传 16 次后，帧会被丢弃。
    (c) 由于对信道的访问是概率性的，并且新到达的站点可以在已经多次重传的站点之前竞争并分配信道，导致了饥饿现象的发生。

17. 这个异常不是资源死锁。虽然进程在共享一个互斥锁（即竞争机制），但资源预分配和死锁避免方法对于这种死锁状态都是无效的。对资源进行线性排序也无效。实际上，可以认为线性排序可能是问题所在；执行互斥锁应该是在进入临界区之前的最后一步，以及在离开临界区之后的第一步。存在一个循环的死锁状态，其中两个进程都在等待只能由另一个进程引起的事件。这是一个通信死锁。为了取得进展，如果抢占消费者的互斥锁，设置一个超时将打破这个死锁。编写仔细的代码或使用监视器进行互斥访问是更好的解决方案。

18. 如果两个程序都首先请求 Woofer，计算机将陷入无限循环：请求 Woofer，取消请求，请求 Woofer，取消请求，依此类推。如果一个程序请求狗窝，另一个程序请求狗，我们将遇到死锁，它会被双方检测到并被打破，但在下一个循环中又会出现。无论哪种情况，如果两台计算机都被编程为首先获取狗或狗窝，都将导致饥饿或死锁。在大多数死锁问题中，饥饿似乎不严重，因为引入随机延迟通常会使其变得不太可能。但在这种情况下，这种方法不起作用。

第 7 章习题解答

1. 其中有许多原因，包括为了节省硬件投资成本、机架空间和电力，并且使管理成千上万台服务器变得更容易而进行服务器整合。
2. 如果硬件配置得到升级，虚拟化可以隐藏这一点，并允许旧软件继续运行。
3. 有各种原因。一个关键原因是在单个桌面机器上有许多平台，如 Windows 7、Windows 8、Linux、FreeBSD、OS X 等，用于测试正在开发的软件。此外，虚拟机在由软件错误引起的崩溃后重新启动速度更快。
4. 在升级到新的计算机和操作系统后，个人可能希望运行一些在旧计算机上有的软件。虚拟化使得在同一台计算机上运行旧系统和新系统成为可能，从而保存旧软件。
5. 很少有程序员能够访问 IBM 大型机。从 20 世纪 80 年代开始，Intel x86 系列主导了计算机领域，而这种架构不支持虚拟化。虽然二进制翻译可以解决这个问题，但直到 20 世纪 90 年代后期才提出了这个想法。
6. 任何改变页表或内存映射的指令肯定是敏感的，还有涉及输入 / 输出的任何指令。任何能够读取计算机真实状态的指令也是敏感的。
7. 有很多，包括移动、算术指令、跳转和调用指令、位移等。
8. 完全虚拟化意味着完全模拟硬件，使得在虚拟机上运行的每个操作系统的行为与在裸机上完全一样。半虚拟化包括更改操作系统，使其不执行任何难以虚拟化的操作。在缺乏硬件支持的情况下，复杂的架构（如 x86）上的完全虚拟化是困难的，而在 RISC 机器上则相对容易。如果存在虚拟化硬件支持，则完全虚拟化不那么困难。因此，哪种更难可能取决于是否有硬件支持。如果有硬件支持，那么半虚拟化操作系统可能更需要工作。如果没有硬件支持，将操作系统更改为更友好可能更容易。如果有许多操作系统需要进行半虚拟化，那可能需要更多工作。
9. 是的，当然。Linux 已经进行了半虚拟化，正是因为其源代码是开放的。Windows 已经被微软进行了半虚拟化处理（因为微软拥有源代码），但尚未发布任何半虚拟化版本。
10. 虚拟机与磁盘分区没有直接关系。虚拟机监视器可以将磁盘分区划分为子分区，并将每个虚拟机赋予其中一部分。原则上，可以有成百上千个子分区。它可以静态地将磁盘划分为 n 个部分，也可以按需划分。在托管的虚拟机中，常常使用主机上的文件来存储虚拟机的磁盘镜像。
11. 在运行时，通过在应用程序和操作系统之间使用虚拟化层，来实现应用程序或进程的虚拟化。这个层在执行应用程序的指令之前，根据需要修改这些指令。应用程序对底层层存在的感知是透明的。Windows 仿真器（WINE）就是一个例子，它可以在其他操作系统（如 Linux）上执行 Microsoft Windows 的二进制可执行文件。这是通过将 Windows API 的调用实时映射到 POSIX 调用来实现的。
12. 类型 1 的虚拟机监视器通常需要改变计算机的引导过程，先加载虚拟机监视器，然后创建虚拟机，在其中安装操作系统。对于由专家系统管理员运营的数据中心来说，这不是个问题，但对于大多数普通用户来说，这太复杂了。类型 2 的虚拟机监视器的发明就是为了使安装虚拟机监视器不比安装应用程序更加复杂，而用户经常进行应用程序的安装。此外，通过使用主机操作系统来服务本地外设，虚拟机监视器不需要为所有外设提供驱动程序，因为它可以使用主机操作系统中的驱动程序。
13. 是的。当客户操作系统进行 I/O 操作时，虚拟化硬件会捕获它，并将控制权传递给类型 2 虚拟机监视器，然后虚拟机监视器决定如何处理。通常，这会涉及向主机操作系统发出 I/O 请求，但虚拟机监视器无需担心捕获 I/O 指令，确实简化了处理过程。
14. 在虚拟化硬件不存在的早期，这项技术得以发明。它是必要的，以防止在用户模式下运行的客户操作系统执行未受特权保护的敏感指令。随着硬件的进步，现代硬件在用户模式程序执行敏感指令时会出现陷阱。然而，在某些情况下，二进制翻译比陷阱更快。然而，随着硬件的改进，对二进制翻译的需求将减少。

15. 通常，ring 0（具有最高特权级别）用于在内核模式下运行；ring 3 用于用户模式。有些虚拟机监视器使用 ring 0 来在内核模式下运行虚拟机监视器；客户操作系统在 ring 1 上运行。当客户操作系统调用特权指令时，它可能会陷入虚拟机监视器，虚拟机监视器在验证访问权限、权限等后执行这些指令。还有其他的方式也是可能的。

16. 已经证明了基于 VT-enabled CPU 的方法导致了很多陷阱，这是由于陷阱-模拟方法的使用。由于陷阱处理开销大，存在一些情况下，基于翻译的方法优于基于硬件的方法。

17. 当客户操作系统发出“清除中断”指令（例如 CLI）时，在一些 CPU（如具有深度流水线的 CPU）中以硬件方式执行可能非常耗时。另一方面，在虚拟化系统中，虚拟机监视器不需要实际在硬件上禁用中断，只需使用一个变量来表示指定的客户操作系统的“中断标志”被设置为零，从而使执行 CLI 指令更快。

18. 它可以预先翻译整个程序。不这样做的原因是许多程序有大段的代码从不被执行。通过按需翻译基本块，没有未使用的代码被翻译。按需翻译的一个潜在劣势是可能稍微不那么高效地启动和停止翻译器，但这种影响可能很小。此外，由于 x86 上的间接分支（目标在运行时计算的分支）和可变大小指令，对 x86 代码进行静态分析和翻译很复杂。因此，您可能不确定要翻译哪些指令。最后，还有自修改代码的问题。

19. 纯虚拟机监视器只是模拟真实硬件，没有其他功能。纯微内核是一个小型操作系统，为使用它的程序提供基本服务。在纯虚拟机监视器上运行的虚拟机运行传统操作系统，如 Windows 和 Linux。在微内核的顶层通常是实现操作系统服务的进程，但以分散方式实现。

20. 如果多个客户操作系统都将它们认为是物理页 k 分配给他们的某个进程，就会出现问题。需要一种方法来执行第二次页面映射，因为虽然客户可能会认为自己控制物理页面，但实际上并不是如此。这就是为什么需要嵌套页表。

21. 机器不仅需要内存来容纳正常（客户）操作系统和所有应用程序，还需要内存来容纳虚拟机监视器的功能和执行客户操作系统敏感指令所需的数据结构。类型 2 虚拟机监视器还具有主机操作系统的额外成本。此外，每个虚拟机将拥有自己的操作系统，因此在内存中将存储 N 个操作系统副本。减少内存使用的一种方法是识别“共享代码”，并仅在内存中保留一份该代码的副本。例如，一个 Web 托管公司可以运行多个虚拟机，每个虚拟机都运行相同版本的 Linux 和 Apache web 服务器代码。在这种情况下，代码段可以在虚拟机之间共享，尽管数据区域必须是私有的。

22. 每个客户操作系统将维护一个页表，将其虚拟页号映射到（在其所共享的虚拟内存中的）物理帧号。为了防止不同的客户操作系统错误地引用相同的物理页号，虚拟机监视器创建了一个影子页表，将虚拟机的虚拟页号映射到虚拟机监视器提供的物理帧号。

23. 页表只能由客户操作系统修改，而不能由客户中的应用程序修改。当客户操作系统完成修改表后，必须通过发出 RETURN FROM TRAP 等敏感指令切换回用户模式。这个指令将产生一个陷阱，并给予虚拟机监视器控制权。然后，虚拟机监视器可以检查客户操作系统的页表是否已被修改。虽然这种方法可能奏效，但必须在每个由客户应用程序执行的系统中检查所有的页表，也就是每次客户操作系统返回到用户模式时都要进行检查。每秒可能会发生数千次这样的转换，因此不太可能像使用只读页面的页表那样高效。

24. 当虚拟机监视器（hypervisor）的页面用尽时，它无法确定客户操作系统真正重视哪些页面。解决办法是在客户操作系统中包含 " 气球驱动 "（balloon drivers）。然后，虚拟机监视器向气球驱动发出信号，扩展其内存使用量，迫使客户操作系统决定哪些页面被淘汰。这确实是作弊，因为虚拟机监视器不应与客户操作系统的特定部分进行交互。它实际上不应该知道虚拟机中的情况。但这种技术以一种简单的方式解决了一个问题，因此所有人都假装不知道其中有什么问题。

25. 如果虚拟机监视器不知道运行在其虚拟机上的客户操作系统的任何信息，则气球驱动无法正常工作。如果无法在客户操作系统中包含气球驱动，例如，如果它们不支持可加载驱动程序并且源代码不可用，因此无法重新编译以包含气球驱动，则气球驱动也无法正常工作。

26. 考虑一个情况，同一个客户操作系统的多个虚拟机副本存在于系统中。在这种情况下，没有必要在内存中维护只读部分（如代码段）的多个副本。只需要维护一个副本，从而减少内存需求并允许在系统上运行更多的虚拟机。这种技术被称为去重。VMware 将其称为“透明页共享”。

27. 是的。早期的直接内存访问（DMA）硬件使用绝对内存地址。如果一个客户操作系统开始对其认为是物理地址 k 的 DMA 操作，它很可能不会传输到应该传输的缓冲区，可能会覆盖一些重要内容。早期的虚拟机监视器必须重写使用 DMA 的代码以使用不会引起问题的地址。

28. 使用云服务意味着您无需设置和维护计算基础架构。您还可以外包备份的任务。此外，如果您的计算需求快速变化，您可以轻松添加或删除计算机。但是，云服务提供商可能轻易窃取您的机密数据，并且承诺的可扩展性可能只是一种幻觉，如果您需要额外的容量，而沃尔玛或其他一些大客户恰好决定使用 10,000 台机器。此外，您与云之间的带宽可能是一个问题。其带宽很可能远远低于本地带宽，因此如果需要在您和云之间传输大量数据，这可能会成为一个问题。此外，如果您正在进行实时工作，您与云之间的带宽可能会随时产生大幅波动，引发问题。

29. 显然存在许多提供不同服务的提供商，但提供虚拟的 x86 机器的提供商将提供基础设施即服务 (IAAS)。提供 Windows 8 或 Linux 机器的提供商将提供平台即服务 (PAAS)。提供在云中运行的字处理程序（例如 Microsoft Word）的提供商将提供软件即服务 (SaaS)。

30. 假设在一台服务器上启动了许多虚拟机。最初，它们都完成相同数量的工作并且需要相同的资源，情况很好。突然之间，其中一个虚拟机开始使用大量资源（CPU、内存等），干扰了其他所有虚拟机。这可能是将其迁移到专用服务器的好时机。

31. 物理 I/O 设备仍然存在问题，因为它们无法随虚拟机迁移，但它们的寄存器可能保存着对系统正常运行至关重要的状态。想象一下对已发出但尚未完成的设备（例如磁盘）的读取或写入操作。网络 I/O 尤其困难，因为其他机器将继续向虚拟机发送数据包，而不知道虚拟机已经移动。即使数据包可以重定向到新的虚拟机管理程序，虚拟机在迁移期间将无响应，这可能需要很长时间，因为整个虚拟机，包括它上面运行的客户操作系统和所有进程，必须迁移到新的机器。结果，如果设备 / 虚拟机管理程序缓冲区溢出，数据包可能会遇到很大的延迟甚至丢包。

32. 为了迁移特定的进程，必须存储并传递进程状态信息，包括打开的文件、警报、信号处理器等。在状态捕获任务期间可能会出现错误，导致可能不正确、不完整或不一致的状态信息。在虚拟机迁移的情况下，整个内存和磁盘映像被移动到新系统，这更容易实现。

33. 标准（停止）迁移包括停止虚拟机并将其内存映像保存为文件。然后将文件传输到目标位置，在虚拟机中安装并重新启动。这样做会导致应用程序在传输过程中停止一小段时间。在许多情况下，应用程序停止是不可取的。通过动态迁移，在虚拟机运行时移动虚拟机的页面。在所有页面到达目标位置后，检查是否有任何页面在迁移后发生变化。如果有变化，它们将被再次复制。这个过程将重复，直到目标位置的所有页面都是最新的。通过这种方式工作（动态迁移），应用程序可以在没有停机时间的情况下被迁移。

34. 三个主要要求是：兼容性（能够以虚拟机的形式运行现有的客户操作系统，无需任何修改）；性能（在虚拟机执行期间的开销最小；否则，用户不会选择在虚拟机中运行他们的应用程序）和隔离性（保护硬件资源免受恶意或未经授权的访问）。

35. VMware 不可能为数以千计的不同 I/O 设备拥有驱动程序。通过使 VMware Workstation 成为类型 2 虚拟机管理程序，它可以间接使用已在主机操作系统中安装的驱动程序来解决这个问题。

36. VMware ESXi 被设计为小型，可以放入服务器的固件中。当服务器开启时，BIOS 可以将其复制到 RAM 并开始创建虚拟机。这极大简化了启动和启动过程。

37. 可以在 *http://virtualboximages.com* 上找到几个示例。这些示例包括各种预安装的开源操作系统分发版。例如，不必获取新 Linux 版本的 ISO 文件，经历安装过程，然后启动虚拟机，而是更容易地下载预安装的 VDI。还有类似的用于 VMWare 的虚拟机。其他示例可以在 *http://www.turnkeylinux.org* 上找到。

第 8 章习题解答

1. USENET 和 SETI@home 都可以被描述为广域分布式系统。然而，USENET 实际上比图 8-1(c) 中的方案更为原始，因为它不需要除了两台机器之间的点对点连接之外的任何网络基础设施。另外，由于它除了确保正确传播新闻文章所需的处理工作之外没有进行任何处理工作，可以争论它是否真正是我们在本章关注的分布式系统的一种。SETI@home 是广域分布式系统的一个更典型的例子；数据被分发到远程节点，然后返回计算结果到协调节点。
2. 根据 CPU 如何连接到内存，其中一个首先通过，例如，先占用总线。它完成内存操作，然后另一个操作发生，依此类推。无法预测哪个首先执行，但如果系统被设计为顺序一致性，那么执行顺序就不应该有影响。
3. 一台运行速度为 200 MIPS 的机器将会发出 2 亿次内存引用每秒，消耗 2 亿个总线周期或占据总线容量的一半。只需两个 CPU 就能占据整个总线。缓存将内存请求数降低到 2000 万次每秒，允许 20 个 CPU 共享总线。要在总线上使用 32 个 CPU，每个 CPU 的请求量不能超过 1250 万次每秒。如果只有来自 2 亿个内存引用的 1250 万个出现在总线上，缓存未命中率必须为 12.5/200，即 6.25%。这意味着命中率为 93.75%。
4. CPU 000、010、100 和 110 无法访问内存 010 和 011。
5. 每个 CPU 完全管理自己的信号。如果键盘生成一个信号并且键盘未分配给任何特定的 CPU（通常情况），则需要将信号发送给正确的 CPU 处理。
6. 要执行系统调用，进程会生成一个陷阱。该陷阱中断自己的 CPU。需要以某种方式将一个从属 CPU 触发陷阱的信息传输给主 CPU。这在第一个模型中没有发生。如果存在处理器间陷阱指令，可以用它来向主处理器发送信号。如果不存在此类指令，则当从属处理器处于空闲状态时，它可以收集系统调用的参数，并将其放入内存中的数据结构中，主处理器可以持续轮询。
7. 这是一个可能的解决方案。
   enter region:
   TST LOCK | Test the value of lock
   JNE ENTER REGION | If it is nonzero, go try again
   TSL REGISTER,LOCK | Copy lock to register and set lock to 1
   CMP REGISTER,#0 | Was lock zero?
   JNE ENTER REGION | If it was nonzero, lock was set, so loop
   RET | Retur n to caller; critical region entered
8. 正如文中所指出的，我们对于编写高并行桌面应用程序的经验（和工具）很有限。虽然桌面应用程序有时是多线程的，但线程通常用于简化 I/O 编程，因此它们不是计算密集型线程。在桌面应用程序领域，有一些可能大规模并行化的可能性是视频游戏，因为游戏的许多方面都需要大量（并行）计算。一种更有前景的方法是将操作系统和库服务并行化。我们已经在当前硬件和操作系统设计中看到了这方面的例子。例如，网络卡现在具有用于加速数据处理并以线速（例如加密，入侵检测等）提供更高级网络服务的片上并行处理器（网络处理器）。另一个例子是视频卡上的强大处理器，用于将视频渲染从主 CPU 卸载，并为应用程序提供更高级的图形 API（例如 Open GL）。人们可以设想用单芯片多核处理器取代这些特殊用途卡。而且，随着核心数量的增加，可以使用相同的基本方法来并行化其他操作系统和常见库服务。
9. 可能只需要在数据结构上加锁就足够了。很难想象一段代码可能做的任何关键操作而不涉及某些内核数据结构。例如，所有的资源获取和释放都使用数据结构。虽然无法证明，但很可能只需要在数据结构上加锁就足够了。
10. 移动该块需要 16 个总线周期，每个 TSL 都进行双向移动。因此，每经过 50 个总线周期，就有 32 个用于移动缓存块的周期被浪费掉了。因此，总线带宽的 64% 被浪费在缓存传输上。
11. 是的，它会，但中断轮询的时间可能会变得非常长，降低性能。但即使没有最大值，也是正确的。

12. 它与 TSL 的效果一样好。通过将 1 预加载到要使用的寄存器中，然后原子地交换该寄存器和内存单元。指令执行后，内存单元被锁定（即值为 1），其先前的值现在包含在寄存器中。如果之前已锁定，则单元未发生更改，调用者必须进行循环。如果之前未锁定，则现在已锁定。

13. 循环由 TSL 指令（5 纳秒）、总线周期（10 纳秒）和向后跳转到 TSL 指令（5 纳秒）组成。因此，在 20 纳秒内，请求 1 个总线周期，占用 10 纳秒。循环占用总线的 50%。

14. 亲和性调度涉及将正确的线程放在正确的 CPU 上。这样做可能会减少 TLB 缺失，因为 TLB 保存在每个 CPU 内部。另一方面，它对页面错误没有影响，因为如果一个页面对一个 CPU 而言在内存中，对所有 CPU 而言也在内存中。

15. (a) 2 (b) 4 (c) 8 (d) 5 (e) 3 (f) 4.

16. 在网格上，最坏的情况是处于相对角落的节点尝试通信。然而，对于一个环形结构，相对角落只相隔两跳。最坏情况是一个角落与中间的节点进行通信。对于奇数 k ，从角落到中间水平跳跃需要 (k-1)/2 跳，再从垂直方向跳 (k-1)/2 次，总共 k-1 跳。对于偶数 k ，中间是一个由四个点组成的方形，所以最坏情况是从角落到这个四点方块中最远的点。水平上需要 k/2 跳，垂直上也需要 k/2 跳，因此直径为 k 。

17. 通过中间切割平面，网络可以分为两个系统，每个系统的结构为 8\times 8\times 4 。两个部分之间有 128 条链接，双向切割带宽为 128 Gbps。

18. 如果仅考虑网络传输时间，每个位需要 1 纳秒，或每个数据包需要 512 纳秒延迟。每边需要 320 纳秒来复制 64 字节（每次复制 4 字节），总共为 640 纳秒。加上 512 纳秒的传输时间，总共为 1132 纳秒。如果需要两次额外的复制，总共为 1792 纳秒。

19. 如果仅考虑传输时间，1 Gbps 的网络传输速度为每秒 125 MB。在 1152 纳秒内传输 64 字节相当于 55.6 MB/ 秒。在 1792 纳秒内传输 64 字节相当于 35.7 MB/ 秒。

20. 使用程序控制的 I/O 移动 k 字节的时间为 20k 纳秒。使用 DMA 的时间为 2000 + 5k 纳秒。将它们相等并解出 k ，得到断点为 133 字节。

21. 显然，如果在远程执行系统调用，将发生错误。如果文件不存在，则无法在远程机器上读取文件。同时，在远程机器上设置闹钟不会向调用机器发送信号。处理远程系统调用的一种方法是将其捕获并发送回原始站点进行执行。

22. 首先，在广播网络上可以发送广播请求。其次，可以维护一个中央数据库，记录每个页面属于谁。第三，每个页面可以有一个主页，由其虚拟地址的高 k 位指示；主页可以跟踪其每个页面的位置。

23. 在这个划分中，节点 1 有 A、E 和 G，节点 2 有 B 和 F，节点 3 有 C、D、H 和 I。节点 1 和节点 2 之间的划分现在包含 AB 和 EB，权重为 5。节点 2 和节点 3 之间的划分现在包含 CD、CI、FI 和 FH，权重为 14。节点 1 和节点 3 之间的划分现在包含 EH 和 GH，权重为 8。总和为 27。

24. 打开文件的表格保存在内核中，因此如果一个进程有打开的文件，在解冻并尝试使用其中一个文件时，新的内核并不知道这些文件。第二个问题是信号屏蔽，这也存储在原始内核中。第三个问题是如果有一个未决的闹钟，它将在错误的机器上触发。总的来说，内核中充满了关于进程的各种信息碎片，它们也必须成功迁移。

25. 以太网节点必须能够检测到数据包之间的碰撞，因此最远分离的两个节点之间的传播延迟必须小于要发送的最短数据包的持续时间。否则，发送方可能完全传输一个数据包并且不会检测到碰撞，即使数据包在电缆的另一端附近发生碰撞。

26. 中间件在不同的操作系统上运行，所以代码显然是不同的，因为嵌入式系统调用是不同的。它们的共同之处在于为上面的应用层提供一个共同的接口。如果应用层只调用中间件层而不是系统调用，那么所有版本都可以有相同的源代码。如果它们也进行真正的系统调用，这些调用将不同。

27. 最合适的服务是 1. 不可靠连接。2. 可靠的字节流。

28. 它是层次结构维护的。全球服务器用于管理 .edu 域名，了解所有大学的信息，而 .com 服务器用于管理以 .com 结尾的域名的信息。因此，要查找 http://cs.uni.edu，计算机首先会在 .edu 服务器上查找 uni，然后前往那里询问 cs，以此类推。

29. 一台计算机可能有许多等待传入连接的进程。这些进程可以是 Web 服务器、邮件服务器、新闻服务器等。必须找到一种方法，使得可以将传入连接定向到特定的进程。这是通过使每个进程侦听特定的端口来实现的。已经约定 Web 服务器将侦听端口 80，因此定向到 Web 服务器的传入连接将发送到端口 80。这个数字本身是任意选择的，但必须选择一个数值。

30. 物理输入 / 输出设备仍然存在问题，因为它们不会随虚拟机一起迁移，但它们的寄存器可能包含对系统正常运行至关重要的状态。想象一下已经发出但尚未完成的对设备（如磁盘）的读取或写入操作。网络输入 / 输出尤其困难，因为其他机器将继续向虚拟机管理程序发送数据包，而不知道虚拟机已经移动。即使可以将数据包重定向到新的虚拟机管理程序，虚拟机在迁移期间将无响应，而迁移时间可能很长，因为整个虚拟机，包括它上面运行的宿主操作系统和所有进程，都必须迁移到新的机器上。结果，数据包可能会出现较大的延迟，甚至丢包，如果设备 / 虚拟机管理程序缓冲区溢出。

31. 一种方法是让 Web 服务器将整个页面（包括所有图像）打包成一个大的 zip 文件，然后第一次发送整个文件，这样只需要一个连接。第二种方法是使用无连接的协议，如 UDP。这将消除连接开销，但会要求服务器和浏览器自己进行错误控制。

32. 使读取的值取决于进程是否恰好在与上一次写入者相同的机器上，这一点根本不透明。这就要求只有进行更改的进程能看到这些更改。另一方面，每台机器只有一个缓存管理器更容易和更便宜实现。如果它必须维护每个修改文件的多个副本，并且返回的值取决于读取的对象，这样的管理器会变得非常复杂。

33. 共享内存作用于整个页面。这可能导致虚假共享，在同一个页面上访问不相关的变量会引发频繁的切换。将每个变量放在独立的页面上是浪费的。基于对象的访问消除了这些问题，并允许更精细的共享粒度。

34. 在将元组插入元组空间时，基于元组中的任何字段进行哈希并不起作用，因为 *in* 可能大部分是形式参数。一种始终有效的优化方法是注意到 *out* 和 *in* 的所有字段都有类型。因此，已知元组空间中所有元组的类型签名，并且需要的 *in* 的元组类型也是已知的。这建议为每个类型签名创建一个元组子空间。例如，所有的 (int, int, int) 元组放入一个空间，而所有的 (string, int, float) 元组放入另一个空间。当执行 *in* 时，只需要搜索匹配的子空间。

第 9 章习题解答

1. 流行的新闻服务需要保证完整性和可用性，但不需要保密性。备份存储系统需要保密性和完整性，但不一定需要 24/7 的可用性。最后，银行服务需要保密性、完整性和可用性。

2. (a) 和 (c) 必须是 TCB 的一部分，而 (b)、(d) 和 (e) 可以在 TCB 之外实现。

3. 隐蔽信道是指通过观察和操纵系统的可测量性能特征在系统中创建的未经授权的通信通道。隐蔽信道存在的关键要求是有一些共享的系统资源，例如 CPU、磁盘或网络，可以用于发送秘密信号。

4. 它只被输入到矩阵中两次。在文本中给出的例子中，*print-er1* 同时在两个域中。这里没有问题。

5. 完全保护矩阵 : 5000\times 100 = 500,000 个单位的空间。
   ACL: 50\times 100 (1% 的对象在所有域中可访问；每个 ACL 有 100 个条目) + 500\times 2 (10% 的对象在两个域中可访问；每个 ACL 有两个条目) + 4450\times 1 (89% 的对象在一个域中可访问；每个 ACL 有一个条目) = 10,450 个单位的空间。用于存储 capability list 所需的空间与 ACL 相同。

6. (a) Capability list
   (b) Capability list
   (c) ACL
   (d) ACL

7. 要使一个文件对除某个人以外的所有人可读，只能使用访问控制列表。为了共享私人文件，可以使用访问控制列表或 capabilities。要使文件公开，最简单的方法是使用访问控制列表，但也可能将文件的 capability 放在一个可识别的位置，以供 capability 系统使用。

8. 这是保护矩阵 :

[MISSING_PAGE_FAIL:57]

到处)。因此，尽可能多的空间被覆盖，但颜色分辨率只有原来的一半。总共只能表示 1/8 的颜色。那些被禁止的颜色被映射到相邻颜色，其所有值都是偶数，例如颜色 (201, 43, 97)、(201, 42, 97)、(200, 43, 96) 和 (200, 42, 97) 现在都映射到点 (200, 42, 96)，不能再区分它们了。

1. “时机已至”，海象对镜中花纹鱼说，“该谈论很多事情，船只、鞋子和蜡烛，甘蓝和君王，为何海水沸腾，猪是否有翅膀，但等一等”，蚝说，“我们先稍作休息，有些人无法喘息，我们所有人都太胖了”，“不急”，木匠说，他们非常感谢他的这一点。
   引自《镜中世界》(Tweedledum 和 Tweedledee)。
2. 约束条件是没有两个单元格含有相同的两个字母，否则解密将是模糊的。因此，这 676 个矩阵元素中每个元素都包含不同的 676 个二连体之一。不同组合的数量因此是 676！这是一个非常大的数字。
3. 排列的数量是 n!，因此这就是密钥空间的大小。一个优点是基于自然语言属性的统计攻击不起作用，因为 E 确实表示 E，等等。
4. 发送者选择一个随机密钥，并用他们共享的秘密密钥对其进行加密，然后将其发送给可信第三方。然后，可信第三方解密随机密钥，并用它与接收者共享的秘密密钥加密它。然后将此消息发送给接收者。
5. 像 y=x^{k} 这样的函数很容易计算，但是取 y 的 k 次方根要困难得多。
6. A 和 B 选择随机密钥 Ka 和 Kb ，并使用 C 的公钥对其进行加密，然后将它们发送给 C 。 C 选择一个随机密钥 K ，并使用 Ka 对其进行加密发送给 A ，使用 Kb 对其进行加密发送给 B 。
7. 一种签署文档的方法是智能卡读取文档后，对其进行哈希，并使用储存在卡中的用户私钥对哈希进行加密。加密哈希将输出到互联网咖啡店的计算机，但是秘密密钥永远不会离开智能卡，因此该方案是安全的。
8. 图像包含 1,920,000 个像素。每个像素有 3 位可用，给出了原始容量为 720,000 字节。如果在存储之前压缩文本，则其有效地增加了一倍，图像可以容纳大约 1,440,000 字节的 ASCII 文本。由于隐写术没有扩展，图像与隐藏数据的大小相同。效率为 25%。这可以轻易地看出，每个 8 位色样本的 1 位包含有效载荷，并且压缩每个载荷位的 ASCII 文本为 2 位。因此，对于每个 24 位像素，实际上编码了 6 位 ASCII 文本。
9. 反抗派可以使用私钥签署消息，然后试图广泛宣传他们的公钥。这可能通过让某人将其从该国家走私出来，然后从一个自由国家发布到互联网上而实现。
10. (a) 两个文件都是 2.25 MB。
    (b) *Hamlet, Julius Caesar, King Lear, Macbeth* 和 *Merchant of Venice*。
    (c) 秘密存储了六个文本文件，总计约为 722 KB。
11. 这取决于密码的长度。密码构建的字母表有 62 个符号。总的搜索空间是 62^{5}+62^{6}+62^{7}+62^{8} ，约为 2\times 10^{14} 。如果已知密码为 k 个字符，则搜索空间仅减少到 62^{k} 。因此，比值为 2\times 10^{14} / 62^{k} 。对于从 5 到 8 的值，这些值分别为 242,235、3907、63 和 1。换句话说，了解密码只有五个字符意味着搜索空间减少了约 242,235 倍，因为不需要尝试所有长密码。这是一个巨大的优势。然而，了解密码有八个字符帮助并不大，因为这意味着可以跳过所有简短（容易）的密码。
12. 试图安抚助手。密码加密算法是公开的。密码在被输入时就被 *login* 程序加密，并将加密后的密码与密码文件中的条目进行比较。
13. 不，他不能。学生可以很容易地找到他的超级用户的随机数。这个信息在密码文件中是未加密的。举个例子，如果它是 0003，那么他只需要尝试将潜在密码加密为 *Susan0003*、*Boston0003*、*IBMPC0003* 等等。但如果另一个用户的密码是 *Boston0004*，他就无法发现它。

14. 假设系统中有 m 个用户。黑客可以收集 m 个盐值，假定这些值都是不同的。黑客需要尝试对每个猜测的密码进行 m 次加密，每次都使用系统使用的 m 个盐中的一个。因此，黑客破解所有密码所需的时间增加了 m 倍。

15. 这里有许多评判标准。以下是其中一些：
    它应该容易和无痛地进行测量（不需要血样）。
    应该有许多可用的数值（不只是眼睛颜色）。
    这个特征在时间上不应该发生变化（不是发色）。
    这个特征应该很难伪造（不是体重）。

16. 不同认证机制的组合将提供更强大的认证。然而，有两个缺点。首先，实施这个系统的成本很高。系统需要使用三种不同的认证机制。其次，这种认证机制给用户增加了额外的负担。用户必须记住他的登录 / 密码，携带他的塑料卡并记住其 PIN 码，并且必须经过指纹匹配的过程。关键问题在于，所有这些都会增加验证用户所需要的时间，从而增加用户的不满。

17. 如果所有机器都是可信的，它可以正常工作。但如果有些机器是不可信的，这个方案就会崩溃，因为一个不可信任的机器可以向一个可信任的机器发送一条消息，让它代表超级用户执行某个命令。接收消息的机器无法判断命令是真的来自超级用户还是学生。

18. 把它们往前使用是行不通的。如果入侵者捕获了一个，他会知道下一次要使用哪个。将它们往后使用可以防止这种危险。

19. 不，这是不可行的。问题在于数组边界没有进行检查。数组不与页面边界对齐，所以 MMU 对此无助。此外，在每次过程调用时，通过内核调用改变 MMU 的成本会非常昂贵。

20. 攻击者通过在访问权限被检查和文件打开之间执行类似符号链接的操作来利用竞争条件。如果文件系统访问是一个事务，访问权限检查和文件打开将成为一个单一事务的一部分，而可序列化属性将确保在其中不能创建符号链接。这种方法的主要缺点是文件系统的性能会受到影响，因为事务会带来额外的开销。

21. 编译器可以在所有数组引用上插入代码来进行边界检查。这个特性将防止缓冲区溢出攻击。但这样做会显著减慢所有程序的运行速度。此外，在 C 语言中，如果将大小为 1 的数组声明为过程参数，然后引用第 20 个元素并不违法，但显然，实际传递地址的数组必须至少有 20 个元素。此外，C 语言中经常使用的函数，如 *memset* 和 *memcpy*，即使它们包含单独的数组，也可以一次性复制整个结构。换句话说，它们被设计成可以导致缓冲区溢出。

22. 如果能够使用能力来实现小保护域，那么不会有问题；否则会有问题。例如，如果一个编辑器仅通过能力访问要编辑的文件和临时文件，则无论编辑器中隐藏了什么样的技巧，它只能读取这两个文件。然而，如果编辑器可以访问用户的所有对象，那么无论有没有能力，特洛伊木马都可以执行其恶意行为。

23. 从安全的角度来看，这是理想的。已使用的块有时会暴露出有价值的信息。从性能的角度来看，清零块会浪费 CPU 时间，从而降低性能。

24. 对于任何操作系统，所有程序必须要么在已知地址开始执行，要么在程序文件头部存储了一个起始地址。 (a) 病毒首先将普通起始地址或头部地址处的指令复制到一个安全的位置，然后在代码中插入一个跳转到自身的指令，或者在头部中插入自己的起始地址。 (b) 在完成自身工作后，病毒执行借用的指令，然后跳转到原本将要执行的下一条指令，或者将控制权转移到原始头部中找到的地址。

25. 主引导记录只需要一个扇区，如果第一个磁道的其余部分是空闲的，它可以提供一个病毒可以隐藏原始引导扇区以及自身代码相当部分的空间。现代磁盘控制器一次读取并缓冲整个磁道，因此读取额外数据时不会产生可察觉的延迟或额外的寻道声音。

26. C 程序的文件扩展名为 .c。不需要使用访问系统调用来测试执行权限，只需要检查文件名是否以 .c 结尾。以下代码可以实现：
    ```
    char *file_name;
    int len;
    ```

```
file_name = dp>d_name;
len = strlen(file_name);
if (strcmp(&file_name[len - 2], *. $c$ *) == 0) infect(s);
```

27. 他们可能无法确定，但他们可以猜测对病毒中的一个字与其他字进行异或操作会生成有效的机器码。他们的计算机可以逐个尝试每个病毒字，并查看它们中有哪些产生有效的机器码。为了减慢这个过程，Virgil 可以使用更好的加密算法，例如对奇数和偶数字使用不同的密钥，然后通过密钥的哈希函数确定位数左移第一个字，以及位数加一左移第二个字，以此类推。
28. 压缩器用于在感染其他可执行程序的过程中进行压缩。
29. 大多数病毒不希望对一个文件进行重复感染，可能甚至不会起作用。因此，能够检测文件中的病毒以查看它是否已经感染是很重要的。用于使病毒难以被防病毒软件检测到的所有技术也使得病毒本身很难告诉哪些文件已被感染。
30. 首先，从硬盘运行 Ifdisk 程序是一个错误。它可能已被感染，可能会感染引导扇区。必须要从原始光盘或只读的软盘中运行它。其次，恢复的文件可能已被感染。不清理它们就将其放回可能会重新安装病毒。
31. 是的，但机制与 Windows 略有不同。在 UNIX 中，伴生病毒可以安装在搜索路径中在真实程序所在的目录之前的目录中。最常见的例子是在用户目录中插入一个名为 *ls* 的程序，它会有效地覆盖掉 /bin/ls，因为它被先找到。
32. 显然，从不明来源执行任何程序都是危险的。自解压缩的存档尤其危险，因为它们可能会将多个文件释放到多个目录中，而解压缩程序本身可能是特洛伊木马。如果有选择的话，最好以普通存档的形式获取文件，并使用你信任的工具进行解压缩。
33. 由于 rootkit 的设计是为了隐藏自己的存在，它会感染操作系统、库和应用程序。因此，任何依赖于系统功能的检测软件都不能被信任。从本质上讲，rootkit 的存在会破坏用于发现它自身的软件。因此，rootkit 检测器必须依赖于外部组件，例如外部 TCB 的扫描。
34. 由于 rootkit 可以破坏恢复软件，例如重设系统还原点，这种系统恢复的方法是不可行的。
35. 不可能编写这样的程序，因为如果这样的程序存在，黑客可以使用该程序来绕过带有病毒的程序中的病毒检查。
36. 可以检查所有进入数据包的源 IP 地址。第二套规则将丢弃所有源 IP 地址属于已知垃圾邮件发送者的进入 IP 数据包。
37. 这没关系。如果使用了零填充，那么 S2 必须在低位的 \text{k} 位中包含真前缀作为无符号整数。如果使用了符号扩展，那么 S2 也必须进行符号扩展。只要 S2 包含了对真实地址进行移位的正确结果，那么 S2 的未使用上位位中的内容就不重要了。
38. 现有的浏览器预装了几个受信任的第三方机构的公钥，例如 Verisign 公司。他们的业务包括验证其他公司的公钥并为其创建证书。这些证书是由 Verisign 的私钥签名的。由于 Verisign 的公钥内置在浏览器中，使用其私钥签名的证书可以被验证。
39. 首先，Java 不提供指针变量。这限制了进程覆盖任意内存位置的能力。其次，Java 不允许用户控制的存储分配（*malloc/free*）。这简化了内存管理。第三，Java 是一种类型安全的语言，确保变量根据其类型以预期的方式使用。
40. 规则如下。

URL	签名者	对象	动作
http://www.appletsRus.com	AppletsRus	/usr/me/appletdir/*	读取
http://www.appletsRus.com	AppletsRus	/usr/tmp/*	读取，写入
http://www.appletsRus.com	AppletsRus	www.appletsRus; port: 5004	连接，读取

1. 小程序是一种执行特定任务（例如填写表单）的小型程序。小程序与应用程序的主要区别在于，小程序在专用大型程序的范围内运行，例如渲染网页。小程序是典型的辅助应用程序示例，它们不会占据用户的注意力，并且旨在易于访问。由于小程序通常是从第三方下载的，它们实际上包含在用户的计算机上运行的外部代码。它们可能包含病毒、蠕虫或其他有害代码。

第 10 章习题解答

1. 由于汇编语言是特定于每台机器的，将 UNIX 移植到一台新机器需要将整个代码重写为新机器的汇编语言。然而，一旦 UNIX 使用 C 语言编写完成，只需要重新编写操作系统的一小部分（例如用于 I/O 设备的设备驱动程序）。
2. 系统调用接口与操作系统内核紧密耦合。标准化系统调用接口会对操作系统内核的设计施加严格限制（减少灵活性）。这也会使得 UNIX 的可移植性降低。
3. 这允许 Linux 使用 gcc 编译器的特殊功能（如语言扩展），这些功能包括提供快捷方式和简化，以及为优化提供编译器提示。主要的缺点是，还有其他流行的、功能丰富的 C 编译器，如 LLVM，无法用于编译 Linux。如果在将来的某个时候，LLVM 或其他编译器在所有方面都变得比 gcc 更好，那么 Linux 将无法使用它。这可能成为一个严重的问题。
4. 列出的文件有：*bonefish, quacker, seahorse*, 和 *weasel*。
5. 它打印文件 *xyz* 中包含字符串 "nd" 的行数。
6. 管道的步骤如下：
   head –8 z | tail –1
   第一部分选择出文件 z 的前八行，并将它们传递给 *tail*，*tail* 只会将最后一行显示在屏幕上。
7. 它们是分开的，因此可以在不影响标准错误的情况下重定向标准输出。在管道中，标准输出可能会传递给另一个进程，但标准错误仍然会写入终端。
8. 每个程序在自己的进程中运行，因此会启动六个新进程。
9. 是的。子进程的内存是父进程的精确副本，包括栈。因此，如果环境变量存在于父进程的栈中，它们也会存在于子进程的栈中。
10. 由于文本段是共享的，只需复制 36 KB。机器每微秒可以复制 80 字节，所以 36 KB 需要 0.46 毫秒。再加上 1 毫秒用于进入和退出内核，整个过程大约需要 1.46 毫秒。
11. Linux 依赖写时复制。它给子进程提供了指向父进程地址空间的指针，但将父进程的页面标记为只读。当子进程试图写入父进程的地址空间时，会发生一个故障，并创建并分配父进程页面的副本到子进程的地址空间中。
12. 负值允许该进程优先于所有普通进程。用户无法信任这种权力。只有超级用户才能使用它，并且只在关键情况下使用。
13. 文本中表示 nice 值的范围为 -20 到 +19，因此默认的静态优先级必须为 120，而事实上也是如此。通过选择一个正的 nice 值并友好地请求将进程置于较低的优先级下。
14. 是的。它不能再运行，所以它的内存越早回到空闲列表中，效果越好。
15. 信号类似于硬件中断。一个例子是闹钟信号，它在未来的指定秒数的时间内向进程发送信号。另一个是浮点数异常信号，它表示除零或其他错误。还有许多其他信号存在。
16. 恶意用户如果能够向任意的不相关进程发送信号，就会对系统造成严重破坏。没有任何东西能够阻止用户编写一个由循环组成的程序，该程序向所有 PID 从 1 到最大 PID 的进程发送信号。这些进程中的许多进程对于这个信号是毫无准备的，而且会被该信号杀死。如果你想杀死自己的进程，那是可以的，但是杀死邻居的进程是不可接受的。

17. 在 Linux 或 Windows 中是不可能的，但 Pentium 硬件确实可以实现这一点。所需要的是利用硬件的分段功能，而这种功能在 Linux 或 Windows 中不被支持。操作系统可以放在一个或多个全局段中，通过受保护的过程调用来进行系统调用，而不是使用陷阱。OS/2 就是这样工作的。

18. 通常，守护进程在后台运行，负责打印和发送电子邮件等任务。由于人们通常不会坐在椅子边等待它们完成，所以它们被赋予低优先级，利用交互式进程不需要的多余 CPU 时间。

19. PID 必须是唯一的。计数器迟早会循环回到 0，然后向上增长，例如增长到 15。如果恰好进程 15 是几个月前启动的，但仍在运行，那么新进程将无法被分配到 PID 15。因此，在使用计数器选择了一个提议的 PID 后，必须搜索进程表，以查看该 PID 是否已被使用。

20. 当进程退出时，父进程将获得其子进程的退出状态。需要 PID 来标识父进程，以便将退出状态传递到正确的进程。

21. 在这种情况下，一个页面现在可以被这三个进程共享。一般来说，写时复制机制可能导致多个进程共享一个页面。为了处理这种情况，操作系统必须为每个共享页面保持一个引用计数。如果 *p1* 在一个被三个进程共享的页面上进行写操作，它会得到一个私有副本，而其他两个进程继续共享它们的副本。

22. 如果所有的 *sharing_flags* 位都被设置，clone 调用会启动一个传统的线程。如果所有的位都被清除，该调用本质上是一个 fork。

23. 每个调度决策都需要查找活动数组的位图，并搜索数组中第一个设置的位，这可以在常数时间内完成；从选定的队列中取出一个任务，同样是常数时间的操作；或者如果位图值为零，则交换活动和过期列表的值，同样也是常数时间的操作。

24. 时钟中断，尤其是在高频率下，会占用相当多的 CPU 时间。它们的主要功能（除了用来跟踪时间，还可以用其他方式完成）是确定何时抢占长时间运行进程。然而，在正常情况下，一个进程每秒会进行数百次系统调用，因此内核可以在每次调用时检查正在运行的进程是否运行时间过长。为了处理完全 CPU 绑定的进程运行时间过长的情况，可以设置一个计时器，例如在 1 秒钟后触发，以防它没有进行任何系统调用。如果只有一个进程，不需要抢占，因此无需进行打断。

25. 从块 0 加载的程序最多只能有 512 个字节长，因此它不能太复杂。加载操作系统需要理解文件系统布局，以便找到并加载操作系统。不同的操作系统具有非常不同的文件系统；只期望一个 512 字节的程序来解决所有这些问题是不够的。相反，块 0 的加载程序只需从磁盘分区上的固定位置获取另一个加载程序。这个程序可以更长且与系统相关，以便它能找到并加载操作系统。

26. 如果使用了共享的文本，那么需要 100KB 的空间。每个进程需要 80KB 用于数据段和 10KB 用于栈，因此总共需要 370KB 的内存。如果没有使用共享文本，每个程序都需要 190KB，所以三个程序需要总共 570KB。

27. 共享文件的进程，包括当前文件指针位置，可以共享一个打开的文件描述符，而不需要更新彼此的私有文件描述符表中的任何内容。与此同时，另一个进程可以通过一个单独的打开文件描述符访问同样的文件，获取一个不同的文件指针，并且按照自己的意愿在文件中移动。

28. 文本段不会改变，因此它永远不需要分页出去。如果需要它的帧，则可以直接放弃它。页面总是可以从文件系统中检索到。数据段不能回到可执行文件，因为它很可能在带入之后发生了改变。将其分页回去会破坏可执行文件。栈段甚至在可执行文件中都不存在。

29. 两个进程可以同时将同一个文件映射到它们的地址空间中。这为它们提供了共享物理内存的一种方式。共享内存的一半可以用作从 A 到 B 的缓冲区，另一半可以用作从 B 到 A 的缓冲区。为了通信，一个进程将一个消息写入到其共享内存的部分，然后向另一个进程发出一个信号，指示有一个等待它的消息。回复可以使用另一个缓冲区。

30. 内存地址 65,536 是文件字节 0，所以内存地址 72,000 是字节 6464。

31. 最初，文件的四个页面被映射：0，1，2 和 3。调用成功后，只剩下页面 2 和 3 仍然被映射，即字节 16,384 至 32,767。

32. 这是可能的。例如，当栈增长超过底部页面时，会触发页错误，操作系统通常会将下一个最低的页面分配给它。然而，如果栈已经撞到了数据段，那么下一个页面无法分配给栈，因此进程必须被终止，因为它已经用完了虚拟地址空间。此外，即使虚拟内存中还有另一个可用的页面，磁盘的分页区域可能已满，无法为新页面分配支持存储器，这也将终止进程。

33. 如果两个块不是兄弟块，则可以出现这种情况。考虑图 10-17（e）中的情况。有两个新的请求，每个请求 8 页。在这一点上，内存的底部 32 页由四个不同的用户拥有，每个用户有 8 页。现在用户 1 和 2 释放他们的页面，但是用户 0 和 3 保留他们的页面。这会产生一个使用了 8 页，8 页空闲，8 页空闲和 8 页使用的情况。我们有两个大小相等的相邻块，但不能合并，因为它们不是兄弟块。

34. 对一个分区进行分页使得可以使用原始设备，而无需使用文件系统数据结构的开销。为了访问块 n，操作系统只需将其加到分区的起始块上，即可计算出其在磁盘上的位置。不需要经过所有本来需要的间接块。

35. 通过工作目录的相对路径打开文件通常对编程人员或用户更方便，因为只需要更短的路径名。而且通常更简单，需要更少的磁盘访问。

36. 结果如下：(a) 锁已被授予。(b) 锁已被授予。(c) C 被阻塞，因为字节 20 到 30 不可用。(d) A 被阻塞，因为字节 20 到 25 不可用。(e) B 被阻塞，因为字节 8 不可用于独占锁定。在这一点上，我们现在有一个死锁。这些进程中的任何一个都无法再次运行。

37. 现在问题是，当锁变得可用时，哪个进程获得锁。最简单的解决方案是将其定义为未确定的。这是 POSIX 所做的，因为它是最容易实现的。另一个解决方案是要求按请求顺序授予锁。这种方法对实现来说更复杂，但可以防止饥饿。还有另一种可能性是让进程在请求锁时提供优先级，并使用这些优先级进行选择。

38. 如果一个进程想要读取一些字节，它将请求共享锁；如果它想要更新一些字节，它将请求独占锁。如果有一系列请求共享锁的进程，请求独占锁的进程可能会被无限期地阻塞。换句话说，如果读者总是在写者之前，写者可能会遭受饥饿。

39. 所有者可以读取、写入和执行它，其他人（包括所有者的组）只能读取和执行它，但不能写入它。

40. 是的。任何能够读取和写入任意块的块设备都可以用于保存文件系统。即使无法定位到特定块，始终可以倒带磁带，然后向前计数到所请求的块。这样的文件系统不会是高性能的文件系统，但可以正常工作。作者实际上在使用 DECtapes 的 PDP-11 上做到了这一点，而且它可行。

41. 不。文件仍然只有一个所有者。例如，如果只有所有者能够对文件进行写操作，那么其他方就无法进行写操作。将文件链接到你的目录中并不会突然赋予你任何在此之前没有的权限。它只是为访问文件创建了一个新的路径。

42. 当使用 chdir 更改工作目录时，会获取新工作目录的 i 节点，并将其保留在内存中的 i 节点表中。根目录的 i 节点也在那里。在用户结构中，维护对这两个的指针。当需要解析路径名时，会检查第一个字符。如果是“/”，则使用指向根 i 节点的指针作为起始位置，否则使用指向工作目录的 i 节点的指针。

43. 访问根目录的 i-node 不需要磁盘访问，因此我们有以下步骤：

    1. 读取 / 目录以查找 "usr"。
    2. 读取 i-node 以获取 *usr* 的 i-node。
    3. 读取 *usr* 目录以查找 "ast"。
    4. 读取 i-node 以获取 */usr/ast* 的 i-node。
    5. 读取 *usr/ast* 目录以查找 "work"。
    6. 读取 i-node 以获取 *usr/ast/work* 的 i-node。
    7. 读取 *usr/ast/work* 目录以查找 "f"。
    8. 读取 i-node 以获取 *usr/ast/work/f* 的 i-node。

因此，总共需要八次磁盘访问来获取 i-node。

1. i-node 包含 12 个地址。单间接块包含 256 个地址。双间接块引导到 65,536 个地址，三重间接块引导到 16,777,216 个地址，总共 16,843,018 个块。这限制了最大文件大小为 12 + 256 + 65,536 + 16,777,218 个块，约为 16GB。
2. 当文件关闭时，内存中其 i-node 的计数器会递减。如果计数器大于零，则不能从表中删除 i-node，因为文件仍在某些进程中打开。只有当计数器归零时，才能删除 i-node。没有引用计数，系统将无法知道何时从表中删除 i-node。每次打开文件时单独复制 i-node 的做法行不通，因为在一个副本中所做的更改在其他副本中不可见。
3. 通过维护每个 CPU 的运行队列，可以在本地进行调度决策，无需执行昂贵的同步机制来始终访问和更新共享运行队列。此外，如果在已经执行的 CPU 上调度线程，则有可能所有相关的内存页仍在缓存中。
4. 一个缺点是安全性问题，因为可加载模块可能包含漏洞和攻击。另一个缺点是随着加载越来越多的模块，内核虚拟地址空间可能变得碎片化。
5. 每 30 秒将修改后的文件内容强制写入磁盘，意味着崩溃造成的损失仅限于 30 秒的数据。如果没有运行 *pdflush*，一个进程可能会写入一个文件，然后退出，而文件的完整内容仍然在缓存中。实际上，用户可能会注销并带着仍在缓存中的文件回家。 一个小时后，系统可能会崩溃，丢失仍然只存在于缓存而不在磁盘上的文件。 第二天我们便会没有一个满意的用户。
6. 它只需要将链接计数设置为 1，因为只有一个目录项引用该 i-node。
7. 通常是 getpid，getuid，getgid，或类似的函数。它们只是从已知位置获取一个整数并返回。其他每个调用都会做更多的操作。
8. 文件被简单地删除。这是删除文件的正常方式（实际上是唯一的方式）。
9. 1.44MB 的软盘可以容纳 1440 个原始数据块。 ext2 文件系统的引导块、超级块、组描述符块、块位图和 i-node 位图每个都使用一个块的空间。如果创建 8192 个 128 字节的 i-node，这些 i-node 将占用另外 1024 个块，只剩下 411 个块未使用。至少需要一个块来存储根目录，留下 410 个块的文件数据空间。实际上，Linux 的 *mkfs* 程序足够智能，不会创建比可能会被使用的 i-node 更多的 i-node，因此效率并不会这么差。默认情况下，将创建占用 23 个块的 184 个 i-node。然而，由于 ext2 文件系统的开销，Linux 通常在软盘和其他小设备上使用 MINIX 1 文件系统。
10. 通常情况下，有时候需要一个人能够执行通常禁止的操作。例如，一个用户启动了一个生成无限输出的作业。然后用户注销并去伦敦度假三周。迟早磁盘会填满，超级用户将不得不手动终止该进程并删除输出文件。还有其他类似的例子存在。
11. 可能有人在教授更改权限时仍然打开了文件。教授应该删除该文件，然后将另一个副本放入公共目录中。此外，他应该使用更好的文件分发方法，比如网页，但这超出了本练习的范围。
12. 例如，如果使用 fsuid 系统调用将超级用户权限授予另一个用户，该用户可以访问超级用户文件，但无法发送信号、终止进程或执行其他需要超级用户特权的操作。
13. (a) 大多数文件在大小上被列为零字节，并包含大量信息。这是因为这些文件在磁盘上不存在。系统会根据需要从实际进程中检索信息。(b) 大多数时间和日期设置反映了当前的时间和日期。这是因为信息只是被检索，并且在许多情况下，它是不断更新的。(c) 大多数文件是只读的。这是因为它们提供的信息与进程相关，无法被用户更改。
14. 这个活动需要保存用户当前的滚动位置和缩放级别，以便能够在恢复时显示相同部分的网页。它还需要保存用户在表单字段中输入的任何数据，以免丢失。它不需要保存它下载的用来显示网页的任何数据（HTML、图像、JavaScript 等）；当活动被恢复时，这些数据可以重新获取和解析，并且通常这些数据仍然在浏览器的本地磁盘缓存中。
15. 当您正在阅读和存储下载的数据时，您需要让设备保持运行状态，因此网络代码应在此期间持有一个唤醒锁（wake lock）。（额外积分：该应用程序还需要请求 "INTERNET" 权限，如图 10-63 所示。）

16. 所有线程都需要在 fork 之后启动。新进程中只会有进行 fork 的线程在运行；如果您已经启动了其他线程，它们将不会在新进程中，并且会使进程处于一个不完全定义的状态。
17. 您知道调用者以某种方式显式地获得了对您原始对象的引用。但是您无法确定调用者是否是您最初发送对象的同一个进程，该进程可能已经将对象传递给另一个进程。
18. 随着需要更多的 RAM，内存耗尽杀手将按照从不必要到最必要的顺序开始终止进程。根据我们已经给出的操作顺序，我们可以确定正在运行具有以下内存耗尽级别的进程：
    1. 浏览器：FOREGROUND
    2. 邮件：SERVICE
    3. 启动器：HOME
    4. 相机：CACHED

因此，内存耗尽杀手将从底部开始工作，首先终止相机进程，然后是启动器，最后是邮件。

第 11 章习题解答

1. 优点是所有内容都在一个地方，很容易找到。缺点是在一个蜂窝中顶层索引的一个坏磁盘块可能会对整个系统造成灾难。
2. 硬件抽象层（HAL）是简单明了的。如果将鼠标、磁盘和所有其他设备驱动程序都包含在其中，将使其变得笨重，并破坏其作为一个隐藏计算机自身某些基本硬件差异（而不是 I/O 设备差异）的薄层的功能。
3. 一个家谱数据库可以方便地使用标准的系统时间格式记录祖先的出生和死亡日期。实际上，任何历史数据库都可以使用这种格式。
4. 如果句柄包含一个序列号，那么当关闭句柄后继续使用时，可以通过比较每个句柄中的序列号与句柄表中的序列号来轻松检测到这一点。在句柄和表项中都必须找到序列号的空间。每次重新使用句柄表项时，序列号就会增加，并且序列号被嵌入到句柄中。如果序列号是 N 位，则要避免检测，一个特定的句柄必须关闭和重新打开 2^N 次。因此，即使对于一个很小的 N ，也可以检测到许多程序中的句柄竞争条件。
5. (a) 进程管理器使用对象管理器来创建线程。
   (b) 内存管理器使用安全管理器来判断是否可以映射文件。
   (c) 插拔管理器使用配置管理器来注册新设备。
6. 信号是在某个进程的上下文中由一个新线程处理的，例如，当按下退出键或甚至当线程出现错误时。在线程的上下文中捕获信号并没有真正意义。这实际上是每个进程的活动。
7. 在服务器上这样做可能更有意义。客户端机器的并发进程较少。只有在有多个进程共享它们时，共享库才有意义。否则，将库进行静态链接并接受重复是更高效的。静态链接的优点是只加载实际需要的过程。使用动态链接库可能导致内存中存在没有被任何人使用的过程。
8. 这个问题是关于内核线程是否可以使用与其对应的用户线程相同的堆栈的。如果这样做，是有意义的，因为大多数系统调用都以空堆栈开始并在返回用户模式时再次将堆栈置空。这些堆栈分开的原因之一是因为肩负的开销与维护单独的堆栈所需的成本相比。开销是因为内核必须保护即将运行的用户堆栈的部分，并确保它足够大，否则必须让另一个线程请求将其扩大。这可能可以通过让内核总是在用户模式堆栈顶部使用一组固定页面来避免，但这就像有一个单独的堆栈一样。分开的另一个原因是内核喜欢知道数据来自哪里以便决定需要多少验证。如果将缓冲区分配在一个单独的内核堆栈上，缓冲区将具有内核地址，通常不需要进行检查。如果内核堆栈使用用户堆栈的一部分，那么内核将更难判断堆栈缓冲区是来自内核线程还是用户线程。此外，内核通常依赖于内核堆栈不能被页面替换。因此，内核要使用的用户堆栈页必须被锁定在内存中。另一方面，分配一个单独的堆栈并不是非常昂贵-远比解决所有这些问题要便宜得多。使线程昂贵的不是它们使用的堆栈，而是管理它们所需的所有元数据。

9. TLB 中缓存的页表项的命中率对系统性能有很大影响。查找缺失项需要耗费大量资源。由于 TLB 只有有限数量的项，使用 2-MB 大页面可以映射更多虚拟地址到 TLB 中。大页面会浪费大量内存，因为在文件区域的最后一页会有未使用的空间。因此，它们只在大区域内使用时才有效。但即使如此，它们也会增加内存压力，因为大页面很有可能包含大量当前未访问的数据，如果使用 4-KB 页面则会被分页到磁盘上。

10. 由于对象句柄中只有 32 位用于权限位，所以操作数有限制为 32。

11. 不可能实现，因为信号量和互斥量是执行对象，而临界区不是。它们主要在用户空间中进行管理（但在需要阻塞时会有一个后备信号量）。对象管理器并不知道它们，而且它们也没有句柄，正如文本中所述。由于 WaitForMultipleObjects 是一个系统调用，系统无法对一些没有了解的事物进行布尔或操作。调用必须是一个系统调用，因为信号量和互斥量都是内核对象。简而言之，不可能有一个混合使用内核对象和用户对象的系统调用，只能选择其中之一。

12. 这个变量可能是指向动态分配的数据结构的指针，其初始化和分配结构体的责任由第一个要使用它的线程承担。如果两个线程几乎同时尝试初始化变量，可能会分配两个不同的结构体，其中一个线程可能使用了错误的实例。InitOnceExecuteOnce 使用了额外的变量来记录每个独立初始化的状态，例如：*uninitialized, initializing, initialized*。实际进行初始化的线程设置额外的变量为 *initializing*，并在初始化完成后将其设置为 *initialized*，并调用 WakeByAddressAll。原子地设置和检查变量可以通过使用锁或类似 *compare&swap* 的硬件指令来执行。如果一个线程发现 foo 被设置为 *initialized*，则跳过初始化。如果它发现 foo 被设置为 *initializing*，则调用 WaitOnAddress 等待初始化完成。

13. (a) 最后一个线程退出。
    (b) 一个线程执行 ExitProcess。
    (c) 拥有该进程句柄的另一个进程终止它。
    (现代应用程序) 操作系统决定终止它以便在交换文件中回收空间，或者因为应用程序正在被服务。

14. 操作系统主要在内存不足或重新启动时终止现代应用程序。如果应用程序在前一种情况下尝试运行，可能没有足够的资源来成功保存其状态。而在后一种情况下，它们可能会无限期地延迟 Windows 的关闭，就像在桌面版 Windows 中经常发生的情况。尽管用户经常在应用程序之间切换，但这些切换的频率是以秒或分钟为单位的人类尺度。对于编写良好的应用程序来说，保存状态所需的几毫秒时间影响很小。此外，以这种方式编写应用程序会在应用程序崩溃时创建更好的用户体验，因为上次保存的状态可作为检查点使用。

15. 由于 ID 立即被重用，一个程序如果通过 ID 识别出要操作的进程，可能会发现该进程已经终止，并且 ID 在它找到 ID 和使用 ID 之间已经被重用。UNIX 在所有其他（32,000）ID 被使用之前不会重用进程 ID。因此，虽然理论上可能发生同样的问题，但可能性非常低。Windows 通过按照 FIFO 顺序保持空闲列表来避免这个问题，因此 ID 通常长时间不被重用。另一种解决方案可能是添加序列号，就像建议使用普通句柄来解决类似问题的对象句柄一样。一般情况下，应用程序不应仅通过 ID 来标识特定进程，还应通过创建时间戳进行标识。要对进程执行操作，可以使用该 ID 获取一个句柄。一旦获取了句柄，可以验证创建时间戳。

16. 最多几微秒。它立即抢占当前线程。只是要运行调度器代码进行线程切换需要多长时间的问题。

17. 将优先级降低到基本优先级以下可以用作对使用过多 CPU 时间或其他资源的惩罚。

18. AutoBoost 需要在仅当一个优先级较高的线程 B 正在等待被线程 A 持有的资源时，才提高线程 A 的优先级。为此，AutoBoost 需要跟踪系统中的所有资源以及每个资源所持有的线程。当 B 阻塞时，AutoBoost 会找到 A 并临时将其优先级提高到与 B 相同的优先级。

19. 对于内核模式线程，它们的堆栈和大多数需要访问的数据结构在更改地址空间之后仍然可访问，因为它们在所有进程之间共享。如果用户模式线程要切换地址空间，它们将丢失对它们的堆栈和其他数据的访问。为用户模式找到一个解决方案，例如在跨进程调用期间更改堆栈，将允许线程创建进程作为隔离边界，并将 CPU 在它们之间传递，就像 CPU 在用户模式和内核模式之间传递一样。可以在未涉及调度器暂停调用方并唤醒服务器进程中（反之亦然）进行跨进程过程调用。

20. 一种方法是提高重要进程的优先级。第二种方法是为重要进程提供更长的时间片。

21. 尽管工作集没有被修剪，但进程仍然修改由文件支持的页面。这些文件将被定期推送到磁盘上。此外，应用程序会显式刷新页面到磁盘，以保存结果并保持文件的一致性。文件系统和注册表始终处于活动状态，并且它们试图确保如果系统崩溃，卷和注册表也不会丢失数据。

22. 对于旋转盘，影响性能最重要的变量是执行 I/O 所需的寻道次数。通常不是 I/O 次数影响性能，而是当需要访问磁盘上的许多不同块时。减少工作集会导致 Windows 一次仅处理进程的少量内存页面。因为 Windows 在页面写出时分配页面文件中的空间，而不是在内存中分配页面，因此页面的写入通常可以写入磁盘的相同区域，从而减少写出页面所需的寻道次数。现代应用程序的前台进程在用户切换或关闭屏幕后很快停止运行。这意味着进程所需的工作集突然变为零。但这并不意味着进程不需要这些页面。只是在用户切换回应用程序之前不需要它们。到那时，很可能会再次需要大部分这些页面。这与通常的情况不同，通常情况下是因为应用程序在执行其他操作而进行页面修剪。在现代应用程序的情况下，是用户（因此是系统）在执行其他操作。换页不仅允许将特定应用程序的页面一起高效地移动到磁盘上，更重要的是它可以高效地将应用程序将要使用的所有页面一次性带回内存，而不使用需求分页。需求分页只会在需要时带入页面。最初，应用程序会运行一小段时间，产生页面错误，再运行一小段时间，再次产生页面错误，直到工作集被重新建立。每次单独的页面错误可能会导致一次寻道，如果任何中间的 I/O 活动使磁盘从交换区域移开。

23. 在 x86 上，自映射是通过在页目录中获取两个条目，并且而不是让它们指向一个页表页，而是让它们指向包含它们的页目录来实现的。这些条目在进程的页目录中进行设置时首次初始化。由于始终使用相同的 PDE 条目，自映射的虚拟地址在每个进程中都是相同的。

24. 要映射 4 GB 的地址空间需要 1M 个 PTE。如果每个 PTE 只有 4 个字节，则整个页表只有 4 MB。

25. 是的。VAD 是内存管理器用来跟踪哪些地址正在使用和哪些是空闲的方式。为了防止后续的预留或提交尝试成功，预留区域需要一个 VAD。

26. (1) 是关于何时以及如何修剪工作集的策略决策。(2) 和 (3) 是必要的。(4) 是关于如何积极地将脏页面写入磁盘的策略决策。(5) 和 (6) 是必要的。(7) 实际上不是一个策略问题，也不是必需的；系统永远不需要清零页面，但如果系统空闲，清零页面总是比执行空闲循环要好。

27. 页面根本不会被移动。只有当页面不在任何工作集中时，才会将其放入列表之一。如果它仍然在一个工作集中，它不会进入任何空闲列表。

28. 它不能放在已修改列表中，因为其中包含仍然映射中的页面，可能会发生缺页错误。未映射的页面不属于该类别。它当然不能直接进入空闲列表，因为这些页面随时可被放弃。脏页面不能随意放弃。因此，必须先将其写回磁盘，然后才能放入空闲列表。

29. 通知事件意味着当事件被发出信号时，所有正在等待的线程都将变为可运行状态。这可能导致大量额外的上下文切换，因为线程尝试获取事件只会再次阻塞。在单处理器上，如果锁持有时间远远短于时间片，那么当其他线程获得运行机会时，前一个线程已经释放了锁。但是，对于持有时间较短的情况下，等待锁的线程数量的可能性也较低。在多处理器上，即使持有时间很短，也可能导致不必要的上下文切换，因为解除阻塞的线程可能会立即在不同的处理器上运行，但立即阻塞，因为只有一个线程能够成功获取锁。

30. 托管第三方 DLL 的程序包括 Web 服务器、浏览器和文档编辑器等。托管的 DLL 可能质量不高，可能会破坏托管程序。通过创建托管处理的副本来运行 DLL，可以使原始进程免受错误的影响，或者（如果使用某种形式的进程沙盒）免受攻击。

31. 这里有两个记录，字段如下。冒号前的值是头字段： 记录 1 = 0, 8: (3, 50), (1, 22), (3, 24), (2, 53) 记录 2 = 10, 10: (1, 60)

32. 块 66 与现有运行块是连续的事实并没有帮助，因为块不按照逻辑文件顺序排列。换句话说，将块 66 用作新的块与使用块 90 没有什么区别。MFT 中的条目为：0, 8: (4, 20), (2, 64), (3, 80), (1, 66)

33. 这只是个巧合。16 个块实际上压缩成了 8 个块。它完全有可能是 9 或 11 块。

34. 除了用户 SID 外，其他信息都可以删除而不会影响安全策略的强度。

35. 攻击者通常可以访问一个计算环境，并使用它来试图控制不同的受保护的计算环境。有很多例子：使用一个系统连接到没有足够防火墙的另一个系统上的服务；在浏览器中运行 Javascript 并控制浏览器；以普通用户身份运行并控制内核。
    从一个漏洞开始，例如某种方式允许攻击者修改内存中的任意位置，他们会如何利用它？他们如何找到要修改的内存位置？如果堆栈或堆中的虚函数表在每台机器上都位于相同的地址上，他们只需修改他们在那里找到的返回地址和函数指针，最终使程序跳转到他们想要的位置。他们希望程序去哪里？如果没有禁止执行保护，他们可以直接跳转到伪装成输入数据的指令中。如果没有禁止执行保护，他们必须选择已经可执行的指令并跳转到它们。如果他们能找到一系列有用的指令序列，通常以返回（即 gadgets）结束，他们可以在使用这种技术时执行相当有趣的程序。
    相关的一点是，在以上描述的两个点上，攻击者需要对他们攻击的地址空间中的数据和代码位置有一个良好的了解。地址空间布局随机化（ASLR）对地址空间进行了混淆，因此攻击者不太可能只是凭借猜测的可能性来进行攻击成功。
    但是，信息泄露可能意味着他们不必这样做。信息泄露是指系统不适当地提供可以被用于成功猜测特定系统或进程地址空间中堆栈或 DLL 等重要数据的信息的漏洞。
    信息泄露的信息源可以包括远程服务 API、托管在其他程序中的语言实现以及系统调用，包括设备驱动程序中 I/O 控制的实现。

36. 不可以。当一个 DLL 加载时，它可以在进程内运行代码。这段代码可以访问进程的所有内存，并使用进程默认令牌中的所有句柄和凭据。模仿只是允许进程使用模仿的客户端的凭据获取新的句柄。允许任何不受信任的代码在进程内执行意味着该进程现在或将来可以被滥用其可访问的任何凭据。下次您使用任何配置为从不受信任站点下载代码的浏览器时，请考虑这一点。

37. 调度程序总是会尝试将线程放置在其理想的处理器上，如果无法实现，则将其放置在同一节点上的处理器上。因此，即使线程当前正在不同节点的处理器上运行，在页错误结束时它可能会被调度回到首选节点。如果该节点上的所有处理器继续忙碌，它可能会在不同的节点上运行，并且会为刚刚出错的页面访问速度慢而支付代价。但是，它更有可能在早晚之间回到其首选节点，并从这个页面放置中受益。

38. 文件系统上的元数据操作，如切换恢复文件，可能会产生不适当的时机，导致崩溃时没有恢复文件。有机会刷新内部数据缓冲区允许应用程序创建一致的状态，这样恢复时更容易。允许应用程序完成复杂操作将简化恢复时所需的逻辑量。
    通常，允许应用程序为卷的快照做准备允许它减少恢复后必须处理的临时状态的数量。崩溃恢复通常很难测试，因为很难评估所有可能的竞争条件的组合。从理论上讲，可以设计软件来处理各种恢复和故障场景，但这非常困难。一个面临这些挑战的软件的示例是 NTFS 文件系统本身。已经投入了大量的设计、分析和测试工作，以确保 NTFS 能够在崩溃后恢复而不需要运行 *chkdsk*（Windows 等价于 UNIX 的 *fsck*）。对于非常庞大的企业文件系统，进行崩溃后的 *chkdsk* 恢复可能需要几个小时或几天。

39. 内存映射文件中的最后一页必须包含最后有效数据后的空数据。否则，如果磁盘的 I/O 驱动程序只覆盖了页面的一部分，并从磁盘中获取了数据，页面的其他部分中的先前数据将会被暴露。对于文件中的早期页面，这不是问题，因为系统会在向用户模式代码提供访问权限之前从文件中读取完整页面。当分配堆栈页面时，由于堆栈增长，这些页面必须包含全部零值，而不是由前一个用户留下的随机数据。

40. 当单个服务器被打补丁时，系统管理员可以自由决定在方便的时间安排更新。如果需要重新启动根操作系统以安装更新，那么虚拟机系统不需要重新启动，但必须将其使用的所有内存保存到磁盘，并在根操作系统重新启动前进行恢复。这涉及到非常大量的 I/O，并且可能需要几分钟，尽管根操作系统的重启只需要几秒钟。在这几分钟内，虚拟机操作系统将无法使用。
    一种解决方案是通过在根操作系统重新启动时不修改或清除虚拟机系统使用的内存来避免进行 I/O 操作。根操作系统只需将内存保留在 RAM 中，并仅将重新启动后找到虚拟机内存所需的信息写入磁盘。

第 12 章习题解答

1. 计算机硬件的改进主要归功于更小的晶体管。一些限制因素包括：(a) 光的波动性可能限制传统光刻技术制造集成电路的能力，(b) 固体中个别原子的迁移性可能导致非常薄的半导体、绝缘体和导体层的性能退化，(c) 背景辐射活性可能破坏分子键或影响非常小的储存电荷。当然还有其他因素。
2. 对于高度交互式的程序，事件模型可能更好。其中，只有 (b) 是交互式的。因此，(a) 和 (c) 是算法驱动的，(b) 是事件驱动的。
3. 不是。差异更多地与 DNS 服务器缓存并且以分级方式组织有关。路径可以很容易地按自顶向下的顺序给出，但倒序的惯例现在已被广泛接受。
4. 可能 stat 是多余的。可以通过 open、fstat 和 close 的组合来实现。对于其他操作，模拟将非常困难。
5. 如果将驱动程序放置在线程之下，那么驱动程序无法像 MINIX 3 那样独立运行。它们必须作为某个其他线程的一部分运行，更类似于 UNIX 风格。
6. 是可能的。所需的是一个管理信号量的用户级进程，也就是信号量服务器。要创建信号量，用户向其发送一条消息，请求一个新的信号量。要使用它，用户进程将信号量的标识传递给其他进程。然后，它们可以向信号量服务器发送请求操作的消息。如果操作阻塞，不会发送回复，从而阻塞调用者。
7. 模式是 8 毫秒的用户代码，然后 2 毫秒的系统代码。通过优化，每个周期现在是 8 毫秒的用户代码和 1 毫秒的系统代码。因此，一个周期从 10 毫秒减少到 9 毫秒。将这样的周期乘以 1000，一个 10 秒的程序现在只需要 9 秒。
8. 外部名称可以任意长度且可变长。内部名称通常是 32 位或 64 位长度，且总是固定长度。外部名称不需要唯一。例如，UNIX 文件系统中的链接可以指向同一对象。内部名称必须唯一。外部名称可以是分层的。内部名称通常是表的索引，因此形成一个扁平的命名空间。
9. 如果新表的大小是旧表的 2 倍，它不会很快填满，从而减少需要升级表的次数。另一方面，可能不需要这么多空间，因此可能会浪费内存。这是经典的时间与空间的权衡。
10. 这样做是有风险的。假设 PID 正好在最后一个条目上。在这种情况下，退出循环会使 p 指向最后一个条目。然而，如果未找到 PID，则 p 可能指向最后一个条目或超出最后一个条目，这取决于编译代码的细节，启用了哪种优化等等。在一个编译器上可能行得通，但在另一个编译器上可能失败。设置一个标志更好。
11. 可以这样做，但不是一个好主意。一个 IDE 或 SCSI 驱动程序可能很长。如此长的条件代码会使源代码难以跟踪。最好将每个驱动程序放在单独的文件中，然后使用 *Makefile* 来确定应该包含哪个文件。或者至少可以使用条件编译来包含一个驱动程序文件或另一个。
12. 是的。它会使代码变慢，而且更多的代码意味着更多的错误。
13. 不容易。同时进行的多个调用可能会干扰彼此。如果静态数据由互斥锁保护，可能是可能的，但这意味着对简单过程的调用可能会意外阻塞。
14. 是的。每次调用宏时，代码都会被复制。如果调用多次，程序会变得非常庞大。这是典型的时间和空间的权衡：一个更大，更快的程序而不是一个更小，更慢的程序。然而，在极端情况下，较大的程序可能无法适应 TLB，导致抖动，从而运行更慢。
15. 从将单词的低 16 位和高 16 位进行异或运算得到一个 16 位整数 s 。对于每一位，有四种情况：00（结果为 0），01（结果为 1），10（结果为 1）和 11（结果为 0）。因此，如果 s 中 1 的个数是奇数，则奇偶校验为奇数；否则为偶数。创建一个包含 65,536 个条目的表格，每个条目包含一个字节和奇偶校验位。宏的定义如下：
    #define parity(w) bits[(w & 0xFFFF) ^ ((w>>16) & 0xFFFF)]
16. 没有情况可以。" 压缩 " 后的颜色值大小与原始值一样大，而且可能需要一个庞大的调色板。这完全没有意义。

17. 8 位的调色板包含 256 个条目，每个条目占用 3 个字节，总共 768 字节。每个像素的节省是 2 个字节。因此，对于超过 384 个像素的图像，GIF 会获胜。16 位的调色板包含 65,536 个条目，每个条目占用 3 个字节，总共 196,608 字节。在这里，每个像素的节省是 1 个字节。因此，超过 196,608 个像素的图像，16 位压缩会获胜。假设宽高比为 4：3，平衡点是一个 512\times 384 像素的图像。对于 VGA (640\times 480)，16 位颜色所需的数据量比真实的 24 位颜色要少。
18. 对于在路径名缓存中的路径，它没有影响，因为绕过了 i-node。如果没有读取它，则它是否已经在内存中并不重要。对于不在名称缓存中但涉及固定的 i-node 的路径，固定 i-node 可以帮助，因为它消除了磁盘读取的需求。
19. 记录最后修改日期、大小，可能还有一个计算出的签名（如校验和或 CRC）可以帮助确定文件自上次引用以来是否发生了变化。警告：远程服务器可能提供有关文件的虚假信息，可能需要本地重新生成计算出的签名。
20. 可以给文件赋予版本号或校验和，并将这些信息与提示一起存储。在访问远程文件之前，将检查版本号或校验和是否与当前文件一致。
21. 文件系统通常会尝试将新数据写入紧随上次使用过的最近可用的磁盘块。如果两个文件同时被写入，这可能导致数据块在磁盘上交错排列，从而导致两个文件都被碎片化，从而更难以读取。可以通过在内存中缓冲数据以最大化写入大小，或将数据写入临时文件，然后在程序终止时将每个输出复制到永久文件来减轻这种效果。
22. Brooks 谈论的是在大型项目中程序员之间的沟通会减慢一切速度。这个问题在一个单人项目中不会出现，因此生产力可能会更高。
23. 如果一个程序员可以以 10 万美元的成本生成 1000 行代码，那么一行代码的成本为 100 美元。Windows 8 由 5000-1 亿行代码组成，这相当于 50-1000 亿美元。这似乎非常庞大。可能微软已经通过使用更好的工具提高了程序员的生产力，使得一个程序员每年可以生成数千行代码。此外，Windows 8 的许多部分未经修改地从 Windows 7 中取出，因此 Windows 8 中新代码的数量只是其总体规模的一小部分。另一方面，微软的年收入约为 700 亿美元，所以在 Windows 8 上花费数十亿美元是可能的。
24. 假设内存每 GB 成本 10 美元（与当前价格对比）。那么一个具有 100GB 硬盘的低端机器需要价值 1000 美元的 RAM。如果 PC 的其他部分成本为 500 美元，则总成本为 1500 美元。这对于低端市场来说太贵了。
25. 嵌入式系统可能只运行一个或少数几个程序。如果所有程序都可以一直保持加载到内存中，可能就不需要内存管理器或文件系统。此外，仅需要驱动程序用于少数几个 I/O 设备，编写 I/O 驱动程序作为库程序可能更有意义。库程序可能会更好地编译为独立程序，而不是共享库，消除了对共享库的需求。在特定情况下，可能可以消除许多其他功能。