现代操作系统

第一章 答案

1. 操作系统必须向用户提供一台扩展(即，实际上)的机器，和它必须管理 I/O 设备和其它系统资源。

2. 多道程序就是 CPU 在内存中多个进程之间迅速切换。它一般被用来使 CPU 保持忙碌，当有一个或多个进程进行 I/O 时。

3. 输入 spooling 是作业中的读入技术，例如，从卡片在磁盘，这样当当前执行的进程完成时，将等候 CPU。输出 spooling 在打印之前首先复制打印文件，而非直接打印。在个人计算机上的输入 spooling 很少，但是输出 spooling 非常普遍。

4. 多道程序的主要原因是当等候 I/O 完成时 CPU 有事可做。如果没有 DMA，I/O 操作时 CPU 被完全占有，因此，多道程序无利可图 (至少在 CPU 利用方面)。无论程序作多少 I/O 操作，CPU 都是 100%的忙碌。当然，这里假定主要的延迟是数据复制时的等待。如果 I/O 很慢的话，CPU 可以做其它工作。

5. 第二代计算机没有必要的硬件保护操作系统免受恶意的用户程序的侵害。

6. 它依然存在。例如，Intel 以各种各样的不同的属性包括速度和能力消耗来生产 Pentium I, II, III 和 4。所有这些机器的体系结构都是兼容的，仅仅是价格上的不同，这些都是家族思想的本质。

7. 25 X 80 字符的单色文本屏幕需要 2000 字节的缓冲器。1024 X 768 象素 24 位颜色的位图需要 2359296 字节。1980 年代这两种选择将分别地耗费$10 和$11520。而对于当前的价格，将少于$1/MB。

8. 选择(a)，(c)，(d)应该被限制在内核模式。

9. 个人的计算机系统总是交互式的，而且经常只有一个用户。而大型机系统几乎总有许多用户强调批处理或者分时。除了对所有资源的有效使用，大型机系统上的保护更加重要。

10. 从管道中每纳秒出现一条指令。意味着该机器每秒执行十亿条指令。它对于管道有多少个阶段全然不予理睬。即使是 10-阶段管道，每阶段 1 nsec，也将执行对每秒十亿条指令。因为无论那种情况，管道末端输出的指令数都是一样的。

11. 原稿包含 80 X 50 X 700 = 2800000 字符。当然，这不可能放入任何目前的 CPU 中，而且对于 1MB 的 cache 来说也太大了，但是如果可能的话，在寄存器中只需 2.8msec，在 Cache 中需要 5.8msec。整本书大约有 2700 个 1024 字节的数据块，因此从磁盘扫描大约为 27 秒，从磁带扫描则需 2 分钟 7 秒。当然，这些时间仅为读取数据的时间。处理和重写数据将增加时间。

12. 从逻辑上说，边界寄存器使用虚拟地址或者物理地址没有任何关系。然而，前者的性能更好。如果使用虚拟地址，虚拟地址和基址寄存器的相加，与比较可以同时开始，而且可以

并行。如果使用物理地址，相加完成之前是不能进行比较的，这就增加了存取时间。

13. 也许。如果调用者取回控制，并且在最终发生写操作时立即重写数据，将会写入错误的数据。然而，如果驱动程序在返回之前首先复制将数据复制到一个专用的缓冲器，那么调用者可以立即继续执行。另一个可能性是允许调用者继续，并且在缓冲器可以再用时给它一个信号，但是这需要很高的技巧，而且容易出错。

14. 陷井由程序造成的，并且与它同步。如果程序一而再地被运行，陷井将总在指令流中相同位置的精确发生。而中断则是由外部事件和其时钟造成的，不具有重复性。

15. *Base* = 40000，*Limit* = 10000。按照本书中描述的方法 *Limit* = 50000 是不正确的。这种方法也是可以执行的，不过这样做将等待 *addree* + *Base* 完成后才能开始边界检查，将会使计算机速度减慢。

16. 进程表是为了存储当前被挂起、甚或是被延迟和阻塞的进程状态。在单一进程的系统中是不需要，因为单一进程从不挂起。

17. 装配文件系统将使得装配目录中已有的任何文件都不可访问，因此装配点通常都是空的。然而，系统管理人员可能需要将某些位于被装配目录中的非常重要的文件复制到装配点，使得他们在进行设备检查或修理时，可以在紧急事件中的普通路径上找到这些文件。

18. 如果进程表中没有空闲的槽(或者没有内存和交换空间)，Fork 将失败。如果所给的文件名不存在，或者不是一个有效的可执行文件，Exec 将失败。如果将要解除链接的文件不存在，或者调用 Unlink 的进程没有权限，则 unlink 将失败。

19. 如果 *fd* 不正确，调用失败，将返回.1.。同样，如果磁盘满，调用也失败，要求写入的字节数和实际写入的字节数可能不等。在正确终止时，总是返回 *nbytes*。

20. 包含字节： 1，5，9，2。

21. 块特殊文件包含被编号的块，每一块都可以独立地读取或者写入。而且可以定位于任何块，并且开始读出或写入。这些对于字符特殊文件是不可能的。

22. 系统调用实际上并没有名称，除了在文件中这样描述之外。当库例程 *read* 陷入内核时，它将系统调用号码放入寄存器或者堆栈中。该号码通常用于一张表的索引。这里确实没有使用任何名称。而另一方面，库例程的名称是十分重要的，因为它将用于程序中。

23. 是的，尤其当系统内核是消息传递系统时。

24. 就程序逻辑而言，库例程调用哪个系统调用是没有关系的。但是，如果需要考虑性能问题，无需系统调用就可以完成的任务将使程序运行更快。所有的系统调用都会导致用户环境和内核环境的切换开销。更进一步，在多用户系统中，在系统调用完成之前，操作系统可能调度到其他的进程，这将使得调用过程的处理更加迟缓。

25. 某些 UNIX 调用没有相应的 Win32 API：

Link：Win32 程序不能给文件另外一个名称，或者使某个文件出现在多个目录中。同时，试图创建链接可以便于测试，并且在文件上加锁。

Mount 和 umount：Windows 程序不能创建关于标准的路径的假定命名，因为具有多个磁盘驱动器的系统上路径名，其驱动器部分是不同的。

Chmod：Windows 程序员不得不假定所有的用户都能访问每个文件。

Kill：Windows 程序员不能 kill 行为失常的程序。

26. 这些都可以直接转换：

(a) micro year = 10^6 X 365 X 24 X 3600 = 31.536 sec。

(b) 1km。

(c)有 2^{40}字节，也就是 1,099,511,627,776 字节。

(d)它是 6 X 10^{24}公斤。

第二章 答案

1. 从阻塞到运行的转换是可以想象的。假设某个进程在 I/O 上阻塞，而且 I/O 结束，如果此时 CPU 空闲，该进程就可以从阻塞态直接转到运行态。而另外一种转换(从就绪态到阻塞态)是不可能的。一个就绪进程是不可能做任何会产生阻塞的 I/O 或者别的什么事情。只有运行的进程才能被阻塞。

2. 应该有一个寄存器包含当前进程表项的指针。当 I/O 结束时，CPU 将把当前的机器状态存入到当前进程表项中。然后，将转到中断设备的中断向量，读取另一个过程表项的指针(服务例程)。然后，就可以启动这个进程了。

3. 通常，高级语言不允许访问 CPU 硬件，而这种访问是必需的。例如，中断处理程序可能需要禁用和启用某个特定设备的中断服务，或者处理进程堆栈区的数据。另外，中断服务例程需要尽快地执行。

4. 内核使用单独的堆栈有若干的原因。其中两个原因如下： 首先，不希望操作系统崩溃，由于某些用户程序不允许足够的堆栈空间。 第二，如果内核将数据保留在用户空间，然后从系统调用返回，那么恶意的用户可能使用这些数据找出某些关于其它进程的信息。

5. 即使是有可能实现，也是很难保持文件系统的一致性。假设某个客户进程给服务器进程 1 发送请求要更新文件。该进程更新其内存的 cache 项。然后，另一个客户进程给服务器进程 2 发送请求读取该文件。不幸的是，如果该文件还在 cache 中，服务器进程 2 对此毫不知情，将返回过时的数据。如果第一个进程在缓冲后将文件写到磁盘中，而服务器进程 2 每次读取时检查磁盘其缓存的备份是否是最新的，系统还可以工作，但是需要避免磁盘访问的所有缓存系统。

6. 当线程停止时，其值保留在寄存器中。当进程停止时寄存器必须被保存。分时线程与分时进程没有区别，因此每个线出都需要其自己的寄存器保存区。

7. 不会。如果单线程进程在键盘上阻塞，就不能创建子进程。

8. 当工作者线程从磁盘读取 Web 页时，它就会被阻塞。如果使用用户级线程，该动作将阻塞整个进程，而破坏多线程的价值。这就是使用内核线程的原因：某些线程的阻塞不会影响到其他线程。

9. 进程中的线程是相互协作的，而不是相互对立的。如果放弃是为了应用程序，那么线程将放弃 CPU。毕竟，通常是同一个程序员写的代码。

10. 用户级线程不能按时钟剥夺，除非整个进程的时间片用完。内核级线程可以单独地被剥夺。在后一种情况下，如果线程运行过久，时钟将中断该当前进程，因而当前线程也被中断。内核可以自由地从同一个进程中选取其他线程运行。

11. 在单线程情况下，cache 命中需 15 msec，cache 未命中需要 90 msec。其加权平均为 2/3 * 15 + 1/3 * 90。因此，平均请求为 40 msec，而服务器每秒可处理 25 个。对于多线程服务器，所有磁盘等待都是重叠的，因此每个请求都耗时 15 msec，而服务器每秒可处理 66.6666 个请求。

12. 是的。如果服务器是完全 CPU 绑定的，则不需要多线程。这只会增加不必要的复杂性。假设某个百万人口区域的电话查号系统(类似于 114)，如果每个(姓名，电话号码)记录为 64 个字符，整个的数据库则为 64MB，这就很容易全部读入服务器内存中以提供快速的查询。

13. 指针是确实必要的，因为全局变量的大小是未知的。它可能是从字符到浮点数数组的任何类型。如果保存其值，就不得不把其大小传递给 *create_global*，这都没有问题，但是必须将其类型作为 *set_global* 的第二个参数，那么 *read_global* 返回值的类型是什么呢?

14. runtime 系统可以正好在这一时刻阻塞或者解除阻塞某个线程，并且忙于处理调度队列。此时并不适合于时钟中断处理程序开始检查该队列是否应该进行线程切换，因为它们可能处于不一致的状态。解决方法可以是：当进入 runtime 系统后，设置一个标志。时钟处理程序将看到该标志，并且设置其自己的标志，然后返回。当 runtime 系统完成时，它将检测时钟标志，看是否有时钟中断发生，并且现在运行时钟处理程序。

15. 这是可能的，不过效率很低。线程想要做一个系统调用，首先设定警报定时器，然后才执行调用。如果线程阻塞，定时器将控制归还给线程包。当然，大多数调用是不阻塞的，而定时器必须被清除。每个可能被阻塞的系统调用都必须作为 3 个系统调用来执行。如果定时器过早时效，各种问题都可能发生。用这种方法建立线程包并不好。

16. 当低优先级进程位于其临界区，而高优先级进程就绪并且被调度时，将发生优先级倒置问题。如果使用忙等待，它将一直运行。对于用户级线程，不可能发生低优先级线程突然被剥夺而允许高优先级线程运行，因为是不可剥夺的。而内核级线程，就会出现这个问题。

17. 每个线程都是自己调用例程，因此它必须有其自己的堆栈以保存局部变量、返回地址等等。这一点用户级线程和内核级线程是一样的。

18. 竞争条件是指如下的情形：两个(或多个)进程将要执行某些动作，其执行依赖于准确的定时。如果某个进程首先执行，所有事件顺利完成，但是如果另一个先执行，则会产生致命的错误。

19. 是。模拟的计算机可能是多道程序的。例如，当进程 *A* 运行时，它读出某些共享变量。然后，发生一个模拟时钟计时，而进程 *B* 运行。它也读出相同的变量。接着，它把该变量加 1。当进程 *A* 运行时，如果它也是给变量加 1，就会产生竞争条件。

20. 是，它还是有用的。当然，它依然是忙等待。

21. 该方法对可剥夺调度完全没问题。事实上，它就是为这种情况设计的。当调度为不可剥夺的，该方法将会失败。假设 *turn* 初值为 0，而进程 1 首先运行。它将一直循环，永不释放 CPU。

22. 当然可以。将内存字作为标志，0 表示没有进程使用临界变量，1 表示有某个进程正在使用。将 1 放入寄存器中，并且交换内存字和寄存器。如果寄存器在交换后为 0，则准许访问。如果它包含 1，则拒绝访问。当进程完成时，将 0 存储在内存标志中。

23. 执行信号量操作，操作系统首先要禁用中断。然后，它读取信号量的值。如果执行 **down** 操作，而信号量等于 0，就将调用进程放入与信号量有关的阻塞进程列表中。如果执行 **up** 操作，必须检测看是否有任何进程在信号量上被阻塞。如果有一个或多个进程被阻塞，从阻塞进程的列表中移出一个，使之就绪。当所有这些操作都完成后，就可以开启中断了。

24. 将每个计数信号量与 2 个二值信号量联合：*M* 用于互斥；*B* 用于阻塞。另外，每个计数

信号量都组合一个用于保存 **up** 次数减去 **down** 次数的计数器，以及在该信号量上阻塞的进程列表。为了实现 **down** 操作，进程首先通过对 *M* 执行 **down** 操作，以获得对信号量、计数器以及列表的独占访问权。然后，将计数器减 1。如果大于等于 0，只需对 *M* 执行 **up** 操作并退出即可。如果 *M* 为负数，就将该进程放入阻塞进程列表中。接着，对 *M* 执行 **up** 操作，对 *B* 执行 **down** 操作来阻塞该进程。为了实现 **up** 操作，首先对 *M* 执行 **down** 操作以获得互斥，然后将计数器加 1。如果计数器大于 0，则没有进程阻塞，就只需对 *M* 执行 **up** 操作。不过，如果计数器小于等于 0，则必须从列表中移出某些进程。最后，按次序对 *B* 和 *M* 执行 **up** 操作。

25. 如果程序操作按阶段执行，直到两个进程都完成当前阶段才能进入下一阶段，这时就应该使用屏障。

26. 对于时间片轮转调度，该方法不会出现问题。*L* 迟早会运行，而且最终将离开其临界区。对于优先级调度，*L* 永远得不到运行；而对于时间片轮转，它将周期性地得到一时间片，因此就有机会离开其临界区。

27. 对于内核线程，线程可以在信号量上阻塞，而内核可以运行该进程中的其它线程。因而，使用信号量没有问题。而对于用户级线程，当某个线程在信号量上阻塞时，内核将认为整个进程都被阻塞，而且不再执行它。因此，进程失败。

28. 其实现的代价很高。每次在某些等待变化的进程的谓词中出现的任何变量，runtime 系统都必须重新计算该谓词，以判断该进程是否能够被解锁。而对于 Hoare 和 Brinch Hansen 管程，则只需 **signal** 原语即可唤醒进程。

29. 雇员之间通过消息传递进行通信：在该例中，消息为订单、食物和袋子。在 UNIX 中，该 4 个进程通过管道连接。

30. 它不会导致竞争条件(不会丢失任何东西)，不过它是完全的忙等待。

31. 如果某个哲学家阻塞，其邻居稍后能够在 *test* 中检测其状态，发现他已经饥饿，当叉子可用时，就可以唤醒他了。

32. 该变化将意味着在哲学家停止进餐后，他的邻居都不能接着被选择。事实上，他们永远不会被选择。假设哲学家 2 完成了进餐，他将为哲学家 1 和 3 运行 *test*，而两者都不会被启动，即使他们两个都饿了而且两个叉子都是可用的。类似的，如果哲学家 4 完成进餐，哲学家 3 也不会被启动。他将无法启动。

33. 变种 1：读者优先。当读着活跃时，写者都无法启动。当一个新的读者出现时，它可以立即开始除非当前有写者是活跃的。当写者完成时，如果有读者在等待，他们全都启动，无论是否有写者存在。

变种 2：写者优先。当有写者等待时，读者都不会开始。当最后活跃的进程结束，如果有作者，就启动它；否则，所有读者(如果有)全部开始。

变种 3：平衡的版本。当有读者是活跃的，新的读者可以立即开始。当写者完成时，新的写者优先，如果有写者等待的话。也就是说，一旦开始读，就一直读到没有读者为止。同样地，一旦开始作，所有挂起的写者都被允许运行。

34. 它将需要 *nT* sec。

35. 如果进程在列表中出现多次，它在每个周期内得到多个时间片。这种方法可以用来给重

要的进程更多的 CPU 时间。但是当该进程阻塞时，最好从可运行进程的列表中将其所有项都删除。

36. 在简单的情况下是有可能通过看源代码来判断是否为 I/O 绑定的。例如，程序开始时，将其所有输入文件读入到缓冲器中，这种程序通常不是 I/O 绑定的；但是，对不同文件进行增量地读写(诸如编译程序)的问题很有可能是 I/O 绑定的。如果操作系统提供诸如 UNIX *ps* 的命令，就可以得知被程序使用的 CPU 时间的量，你能把这个时间量与整个的时间比较以判断。当然，最有意义就是你是系统中唯一的用户。

37. 对于管道中的多个进程，普通的父进程可以将数据流的信息传递给操作系统。有了这个信息，OS 就可以确定哪个进程可以向需要输入的阻塞进程提供输出。

38. CPU 的效率就是有用的 CPU 时间除以整个的 CPU 时间。

当 $Q \geq T$ 时，基本的周期就是进程运行 T，然后进程切换 S。因此，(a)和(b)的效率都是 $T/(S + T)$。当时间片比 T 短时，每运行一次 T 就要求 T/Q 次进程切换，浪费时间为 ST/Q。因此，其效率为

$$\frac{T}{T + ST/Q}$$

也就是下降到 $Q/(Q + S)$，这就是(c)的答案。至于(d)，只需以 S 替代 Q，就可以计算出其效率为 50%。最后，(e)的效率趋近于 0。

39. 最短作业优先可以使得平均响应时间最短。

$0 < X \leq 3$: 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.

40. 对于时间片轮转，在头 10 分钟里，每个作业获得 1/5 的 CPU 时间。在第 10 分钟时，*C* 结束。在接下来的 8 分钟里，每个作业获得 1/4 的 CPU 时间，然后 *D* 完成。然后，在接下来的 6 分钟内，余下的 3 个作业各获得 1/3 的 CPU 时间，直到 *B* 结束，以此类推。因此，5 个作业的完成时间分别为是 10，18，24，28 和 30，平均为 22 分钟。

对于优先级调度，*B* 最先运行，6 分钟完成。其它作业分别在第 14，24，26 和 30 分钟完成，平均为 18.8 分钟。

如果作业按 *A*→*E* 的次序执行，则分别在第 10，16，18，22 和 30 分钟完成，因此，平均为 19.2 分钟。

最后，最短作业优先调度的完成时间分别为第 2，6，12，20 和 30 分钟，平均为 14 分钟。

41. 第一次得到 1 个时间片。随后获得 2，4，8 和 15 个时间片，因此必须经过 5 次交换。

42. 可以检查程序是否期待输入，并且对输入进行处理。不期待输入也不对输入进行处理的程序将不得到任何特殊的优先级提升。

43. 预测值为 40/8 + 20/8 + 40/4 + 15/2 = 25。

44. 所使用的 CPU 的片断为 35/50 + 20/100 + 10/200 + *x*/250。为了使得进程可调度，必须是总和小于 1。因此，*x* 必须小于 12.5 msec。

45. 当内存太小不能载入所有就绪进程时，就需要使用两级调度。某些进程被载入内存，并且从中选择一个运行。内存中进程会随着时间调整。这种算法容易实现也非常有效，另外，时间片轮转调度并不管进程是否在内存中。

第三章 答案

1. 在美国的，假设有三个或更多候选人竞选总统。每个人都不足多数，又都不愿放弃。这就是死锁。

2. 如果打印机在收到整个文件之前开始打印(加速响应)，磁盘可能被其他请求文件装满，而其他文件又必须等到第一个文件完成后才能开始，但是磁盘空间又无法接受当前文件。如果后台打印直到整个的文件接受后才开始打印文件，就可以拒绝大的请求。开始打印文件相当于保留打印机；如果将打印机保留到整个文件接受后，整个系统就可以避免死锁。当然无法容纳的文件仍然是死锁的，必须另选其他设备打印较大的文件。

3. 打印机是不可剥夺的；系统在完成前一个作业之前，不能开始打印另一个作业。Spool 磁盘是可剥夺的；假设协议允许的话，可以删除未发送完毕的大文件，让用户稍后再发送。

4. 是的，没有任何区别。

5. 是的，还有其他形式的图表。我们声明资源只能由单个的进程占用。由资源方块到进程圆圈的弧线表明该进程拥有该资源。因此，具有从两个或更多弧线的方块意味着所有这些进程都占用该资源，这是与规则不符的。因而，有多重的出弧的方块，而结束于不同的圆圈的任何图表都是不合规则的。从方块到方块的弧，以及从圆圈到圆圈的弧也是违反规则的。

6. 可以将部分资源预留，只能由管理员自己的进程使用，这样就可以运行外壳和程序，对死锁做出评估，以杀死某些进程是系统可以继续运行。

7. 这些变化都不会导致死锁。无论哪种情况，都没有出现循环等待。

8. 资源的自愿放弃非常类似于通过剥夺进行恢复。其主要区别在于计算机进程不能自己解决这种问题。把操作系统作为警察，以覆盖独立进程所服从的一般规则。

9. 该进程请求的资源比系统所有的资源要多。没有什么方法可以获得这些资源，因此无法完成，即使其他进程不再请求资源。

10. 如果系统有两个或多个 CPU，两个或多个进程可以平行，就可导致斜向的轨迹。

11. 是的。可以在三维中实现。Z 轴用于表示第三个进程执行的指令。

12. 该方法只能在预先确知什么时候使用资源的情况下才能进行调度。在实践中，这是很少见的。

13. *D* 的请求是不安全的，而 *C* 的请求是安全的。

14. 某些状态是既非死锁也不安全的，但是这些状态会导致死锁状态。例如，假设有 4 种资源：磁带，绘图仪，扫描仪和 CD-ROM，以及竞争这些资源的 3 个进程。有下列的情形：

已分配	请求矩阵	可用资源
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	

该状态并未死锁，因为还可以执行很多行动，例如，*A* 仍然可以得到 2 台打印机。然而，如果每个进程都请求其剩余需求，就会导致死锁。

15. 该系统是不会死锁的。假设每个进程都已占用 1 个资源，还有 1 个资源是空闲的。任一进程都可以请求该资源并且得到它，然后该进程能够完成并且释放 2 个资源。因此，死锁是不可能的。

16. 如果某个进程有 m 个资源，该进程就能完成，而不会造成死锁。因此，最坏的情况是所有进程都占又 $m-1$ 个资源，而且都再申请另一个。如果系统还剩余 1 个资源，其中一个进程就能完成，并且释放其所有资源，同时也可以让其余进程也完成。因此，避免死锁的条件就是 $r \geq p(m-1)+1$。

17. 不会。D 仍然能完成。当它完成后，它将归还足够的资源使得 E(或者 A)完成，以此类推。

18. 如果是 3 个进程，每个都能有 2 个驱动器。如果有 4 个进程，驱动器分配为(2, 2, 1, 1)，头两个进程可以完成。如果为 5 个进程，分配为(2, 1, 1, 1, 1)，第一个进程仍然可以完成。如果有 6 个进程，每个都占用 1 个磁带驱动器，而且再申请另一个，就会死锁。因此，只要 $n<6$，系统就不会死锁。

19. 可用资源的向量与矩阵中一行的比较需要m次操作。该步骤必须重复n次以查找某个可以完成的进程，并且标记该进程。因此，标记进程需要mn级操作。对所有n个进程重复该算法意味着需要mn^2次操作。

20. 需求矩阵如下：

0 1 0 0 2
0 2 1 0 0
1 0 3 0 0
0 0 1 1 1

如果 $x=0$，立即死锁。如果 $x=1$，进程 D 能完成。当它完成时，可用向量为 11221。不幸的是，还是死锁。如果 $x=2$，在 D 运行之后，可用向量为 11321，而 C 就能运行。C 完成后，归还其资源，可用向量是 22331，这样 B 也能运行并且完成，然后 A 运行且完成。因此，为了避免死锁，x 的最小值为 2。

21. 是。假设所有邮箱都是空的。此时，A 发送到 B 并且等待应答，B 发送到 C 并且等待应答，而 C 发送到 A 并且等待应答。这样，所有死锁的条件都满足了。

22. 假设进程 A 要求按 a, b, c 申请记录。如果进程 B 同样先请求 a，其中某个进程将得到它，而另一个将被阻塞。这种情形总是不会造成死锁的，因为，获得记录 a 的进程现在没有干扰而运行结束。在其余 4 种组合中，某些可能导致死锁而某些不会死锁。6 种情况如下：

a b c　不会死锁
a c b　不会死锁
b a c　可能死锁
b c a　可能死锁
c a b　可能死锁
c b a　可能死锁

因为 6 种组合中有 4 个可能导致死锁，因此，有 1/3 机会避免死锁，2/3 机会可能死锁。

23. 两阶段加锁可以消除死锁，但是引入了潜在的饥饿。进程必须不断测试，而无法获取其所有记录。应该设置其尝试的上限。

24. 为了避免循环等待，按照账号号码对资源(也就是账号)进行编号。在读入输入数据行后，

该进程锁首先对低编号的账号加锁，锁定后，再锁另一个。由于没有进程会等候比锁定账号更低的账号，因此不会循环等待，因而也不会死锁。

25. 按照如下方法改变请求新资源的方案。如果某进程请求某个新资源而且该资源是可用的，它将得到该资源，并且保留其已有的。如果新资源不可用，释放其所有已占用资源。此时，死锁是不可能的，并且不会出现获得新资源而失去已有资源的危险。当然，进程只有在可能释放资源的情况下才行。

26. 我给的是 F(不及格)。该进程做什么？由于它明显需要资源，它只不过再次申请并且再次阻塞。并不比一直阻塞更好。事实上，它可能更糟，因为系统需要记录各竞争进程等待的时间，而把新近释放的资源分配给等待最久的进程。而周期性地计时和不断尝试，使得进程失去其资历。

27. 如果两个程序都是先申请得到 Woofer，计算机将得到无尽序列的饥饿：请求 Woofer，取消请求，要求 Woofer，取消要求等等。如果其一要求得到狗屋，而另一个要求得到狗，就是死锁，这样双方发现和然后重来，但是在下一个周期再次重复。无论那种情况，如果两台计算机被编程首先请求狗或者狗屋，不是饥饿就是死锁。这里两者之间没什么区别。在大多数死锁问题中，饥饿似乎并不是严重问题，只需引入随机延迟就使得饥饿几乎不可能出现。不过，该方法在此没有作用。

第四章 答案

1. 4 个进程空闲的总量为 1/16，因此，CPU 的空闲时间也是 1/16。

2. 如果每个作业都有 50%的I/O等待，那么没有竞争的情况下，需要花费 20 分钟完成。如果顺序运行，第二个将在第 40 分钟完成。对于 2 个作业，CPU近似利用率为 1 - 0.5^2。因此，每一分钟实际时间，每个作业获得 0.375 分钟CPU时间。为了获得 10 分钟CPU时间，每个作业必须运行 10/0.375 分钟，大约为 26.67 分钟。所以，顺序运行的作业需要 40 分钟完成，但是并行在 26.67 分钟之后完成。

3. 几乎整个内存都必须复制，也就是要求读出每个内存字，然后重写到不同的位置。读 4 个字节需 10 nsec，因此读 1 个字节需 2.5 nsec，而还要花费 2.5 nsec用于重写，因此，对于每个字节的压缩需要 5 nsec。其速率为 200,000,000 字节/sec。为了复制 128 MB(2^{27}字节，也就是大约 1.34×10^8字节)，该计算机需要 2^{27}/200,000,000 sec，也就是大约 671 msec。该数字是稍微悲观的，因为如果内存底部的最初的空洞为*k*字节，那么该*k*字节无需复制。不过，如果有许多空洞和许多数据碎片，因此空洞将非常小，*k*也就非常小，而误差也非常小。

4. 位映像每个分配单元需要 1 位。对于 $2^{27}/n$个分配单元，需要 $2^{24}/n$字节。链表总共有 $2^{27}/2^{16} = 2^{11}$个节点，每个节点 8 字节，也就是总共 2^{14}字节。对于小的*n*，链表较好。对于大的*n*，位映像较好。*n*为 1 KB时，两者相等。

5. 首次适配分别为 20 KB，10 KB，18 KB。最佳适配为 12 KB，10 KB，和 9 KB。最差适配为 20 KB，18 KB 和 15 KB。下次适配为 20 KB，18 KB 和 9 KB。

6. 实际内存使用物理地址。也就是内存芯片对之作出反应的总线数字。虚拟地址为对应于进程地址空间的逻辑地址。因此，16 位字的机器可以生成的虚拟地址最高达 64K，无论该机器的内存是多于还是少于 64 KB。

7. 对于 4-KB 页尺寸(页号，偏移)对分别为(4, 3616)，(8, 0)和(14, 2656)。对于 8-KB 页尺寸分别为(2, 3616)，(4, 0)和(7, 2656)。

8. (a) 8212
   (b) 4100
   (c) 24684

9. 他们创建一个 MMU 并将其插入到 8086 和总线之间。因此，所有进入 8086 的物理地址都作为虚拟地址。然后，MMU 将其映射到物理地址上，也就是发送到总线上的地址。

10. 所有进程的整个虚拟地址空间为 *nv*，这就是页面存储所需的。不过，可以在 RAM 中存储量为 *r*，因此需要的磁盘存储量仅为 *nv* - *r*。该量比实际所需的要大得多，因为极少有 *n* 个进程实际运行，而且这些进程也极少需要其最大允许的虚拟内存。

11. 每 *k* 条指令一次缺页，需另加 *n*/*k* nsec，因此平均的指令时间为 10 + *n*/*k* nsec。

12. 该页表包含 $2^{32}/2^{13}$项，也就是 524,288 项。装载该页表需 52 msec。如果某进程得到 100 msec，那么载入页表为 52 msec，运行时间为 48 msec。因此，52%的时间用于载入页表。

13. 20 位被用于虚页号，因此偏移还剩 12 位。页面大小为 2^{12} = 4-KB。20 位的虚页号也就意味着 2^{20}页。

14. 页面数只依赖于 *a*，*b* 和 *c* 之和。至于它们是如何划分是无关的。

15. 对于一级页表，需要 $2^{32}/2^{12}$ = 1M页。因此，该页表必需 1M项。对于二级页表，主页表有 1K项，每项都指向一个二级页格。仅使用了这两个。总共只需 3 个页格项，一个在顶级页表中，以及每个低级页表中一个。

16. 代码和引用串如下：

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。

17. 有效的指令时间为 $1h + 5(1 - h)$，h 为命中率。因此，要使时间小于 2 nsec，*h* 必须至少为 0.75。

18. TLB 中不需要 *R* 位。这里出现的页表明其已被引用；否则，就不会再次出现。因此，*R* 位完全是多余的。不过，当该项被写回内存时，需要设置内存页表的 *R* 位。

19. 相联存储器的本质是同时把多个寄存器的内容与关键字进行比较。对于每个寄存器，必须有一组比较器，将其每一位与搜索的关键字逐位比较。实现这样的设备所需的门(或者晶体管)的数目是寄存器数目的线形函数，因此扩展该设计的费用也是线形增长的。

20. 8-KB页和 48-位虚拟地址空间，虚页数为 $2^{48}/2^{13} = 2^{35}$(大约 340 亿)个。

21. 内存有 $2^{28}/2^{13}$ = 32,768 页。32K的哈希表的平均链长为 1。为了使之小于 1，必须使用下一个尺寸 65,536 项。将 32,768 项放入 65,536 格中使得其平均链长为 0.5，以保证快速的查询。

22. 这似乎不太可能，除非其程序在编译时是完全可预测其执行的，这并常见也没什么用处。如果编译程序收集关于例程的调用代码的位置信息，这些信息可以链接时被用来重整目标代码，使例程被定位于接近其调用代码。这很有可能做例程与调用代码处于同一页。当然，这对该程序中从许多其它地方的调用没有什么帮助。

23. FIFO 的页框如下：

x0172333300
xx017222233
xxx01777722
xxxx0111177

LRU 的页框如下：

x0172327103
xx017232710
xxx01773271
xxxx0111327

FIFO 发生 6 次缺页；LRU 为 7 次。

24. 第一个 *R* 位为 0 页将被选择，也就是 *D*。

25. 计数器：

Page 0: 01101110
Page 1: 01001001
Page 2: 00110111
Page 3: 10001011

26. 第一个 $R = 0$ 且 age > τ将被选择。因此从头开始扫描，最前的一页(1620)被清除。

27. 该页的 age 为 2204 – 1213 = 991。如果τ = 400，它肯定地不在工作集中，最近没有被引用，因此，将被清除。τ = 1000 就不同了。该页位于工作集中，就不会被清除。

28. 寻道加上旋转延时为 20 msec。对于 2-KB 页，传输时间为 1.25 msec，总共为 21.25 msec。载入 32 页需 680 msec。对于 4-KB 页，传输时间为 2.5 msec，总计 22.50 msec。载入 16 页需 360 msec。

29. NRU 移去第 2 页。FIFO 移去第 3 页。LRU 移去第 1 页。第二次机会移去第 2 页。

30. PDP-1 分页磁鼓的优点是无需旋转延时。每次内存读写都可以节约一半的旋转时间。

31. 正文为 8 页，数据为 5 页，而堆栈为 4 页。该程序无法放入，因为它需要 17 个 4096-字节的页。而对于 512-字节的页，就不一样了。正文正好 64 页，数据 33 页，而堆栈 31 页，总共 128 个 512-字节的页，刚好。

32. 如果页面可以共享，就可以出现在两个工作集中。例如，如果某分时系统的 2 个用户运行同样的编辑程序，程序正文是共享的，而不是拷贝，其中某些页可以同时在每个用户的工作集中。

33. 这是可能的。假设段不在，其保护信息肯定在页表中。如果每个进程都有其自己的页表，那么每个都有其自己保护位。它们就可以是不同的。

34. 该程序有 15,000 次缺页，每次需额外花费 2 msec 的处理时间。缺页的总处理时间为 30 sec。也就是意味着所使用的 60 sec 中，一半用于处理缺页，一半用于运行程序。如果两倍的内存运行该程序，却也将减少一半，而处理缺页的时间只需 15 sec，因此整个运行时间为 45 sec。

35. 如果不能修改程序，则该方法是有效的。如果数据不能被修改，对数据也是有效的。不过，程序通常是不能被修改的，而数据很少不能被修改。如果二进制文件中的数据区被更新的页重写，下次运行程序时，将不包含原来的数据。

36. 该指令可以跨越一页的边界，仅是读指令时就可导致两次缺页。读取内存字也可能跨越一页的边界，又产生 2 次缺页，总共 4 次。如果内存字必须挨着排列，该数据字还可能导致一次缺页。

37. 当最后的分配单元不满时，就产生了内部碎片。而外部碎片是指两个分配单元之间被浪费的空间。在分页系统中，最后一页丢失的空间为内部碎片。在纯分段系统中，段之间的某些空间是不可用的。这就是外部碎片。

38. 不用。查找关键字为段号加上虚页号，因此，一次匹配就可以找到该页。

第五章 答案

1. 在此图中，一个控制器有两个设备。单个控制器可以有多个设备就无需每个设备都有一个控制器。如果控制器变得几乎是自由的，那么只需把控制器做入设备本身就行了。这种设计同样也可以并行多个传输，因而也获得较好的性能。

2. 太简单了。扫描仪最高速率为 400 KB/sec。而总线程和磁盘都为 16.7 MB/sec，因此磁盘和总线都无法饱和。

3. 这不是一个好主意。内存总线肯定比 I/O 总线快。一般的内存请求总是内存总线先完成，而 I/O 总线仍然忙碌。如果 CPU 要一直等待 I/O 总线完成，那就是将内存的性能降低为 I/O 总线的水平。

4. 每次总线处理都有请求和响应，各需 100 nsec，也就是每次 200 nsec。这就表明其处理能力为 500 万/sec。如果每次都是 4 个字节，总线速率必须到达 20 MB/sec。这些处理与 4 个 I/O 设备的事实是不相干的。总线处理为 200 nsec，不论是针对相同设备或者不同设备，因此 DMA 控制器的通道数有没有关系。总线根本不知道或者也不关心。

5. 一次中断需要入栈 34 个字。而从中断返回需要把 34 个字从栈中取出。总耗时为 680 nsec。因此，每秒最多处理 147 万次中断，假设每次中断什么也不做。

6. 在开始中断服务例程时就确认是可以的。而在最后才做的原因是因为中断服务例程的代码都非常短。通过先输出另一个字符和然后确认该中断，如果立即发生另一个中断，打印机将在此中断期间工作，将使得打印稍快。该方法的缺点是当其他中断禁用时，死机时间稍长。

7. 是的。入栈的 PC 指向第一条未读取的指令。之前的所有指令都已执行，而指向的指令及其后续指令均尚未执行。这就是精确中断的条件。精确中断在单管线的机器上不难实现。只有当指令不按序执行时才会有麻烦，该例不是。

8. 该打印机打印每分钟打印 50 X 80 X 6 = 24000 个字符，也就是 400 字符/sec。每个字符使用 50 nsec 的 CPU 时间用于中断，因此，每秒总共的中断时间是 20 msec。使用中断驱动 I/O，余下的 980 msec 可供其它使用。换句话说，中断耗时只占 CPU 时间的 2%，这几乎不会影响运行的程序。

9. 设备无关性是指以相同的方法访问文件和设备，而与其物理特性无关。如果系统采用一组调用写文件，而使用另一组调用写控制台(终端)，那么就不具有设备无关性。

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

11. 按照图 5-17 中的数据，对于软盘，每磁道为 9 X 512 X 8 = 36864 位。按照每旋转一周 200 msec 其位速率为 184,320 位/sec。硬盘平均每磁道 281 个扇区，因此平均每磁道有 281 X 512 X 8 = 1,150,976 位。对应于 8.33 msec 的旋转时间，每秒旋转 120 周(7200 rpm)，因此每秒传输 120 X 1,150,976 位。也就是大约 138M 位/sec。软盘的数据率大概是 56-Kbps 调制解调器的 3 倍。而硬盘的数据率比快速以太网要快大约 38%的。不过，这些计算高估了实际的最高数据速率，因为磁盘上每 512 字节的数据中包含一些字节的格式化信息，用于识别磁道和扇

区；以及扇区间隔使得速度稍有变化，以防扇区重叠。

13. 如果每次输出都立即分配打印机，某进程可以通过打印机 1 个字符来冻结打印机，然后休眠一个星期。

14. 该磁盘旋转速率为 120 rps，因此每旋转一周需 1000/120 msec。对于每次旋转 200 个扇区，则扇区时间为(1/200) X (1000/120) = 5/120 = 1/24 msec。在 1 msec 的寻道时间内，将越过 24 个扇区，因此柱面倾斜应该为 24。

15. 如上题所述，扇区时间为 1/24 msec。也就意味着该磁盘每秒读取 24,000 个扇区。而每个扇区为 512 个字节，其数据速率为 12,288,000 字节/sec。也就是 11.7 MB/sec。

16. RAID level 2 不仅可以从故障驱动器来恢复错误位，还可以从未被检测的的瞬时差错中恢复。如果某驱动器发送一个坏数据位，RAID level 2 可以纠正，而 RAID level 3 不能。

17. 0 次故障的概率P_0为$(1 - p)^k$。1 次故障的概率P_1为$kp(1 - p)^{k-1}$。而整个RAID发生故障的概率为 $1 - P_0 - P_1$。也就是 $1 - (1 - p)^k - kp(1 - p)^{k-1}$。

18. 在两个磁极之间会产生磁场。不仅难于使磁场源变小，而且磁场传播迅速，这将导致此行媒体的表面接近磁源或者传感器的机械问题。而半导体激光可以在非常小的地方产生激光，而且激光可以从较远的地方感知这些极小的点。

19. 有可能。如果大多数文件被存储在逻辑上连续的扇区内，那么就可能使得程序有时间以交叉扇区的形式处理刚刚接收的数据，这样当下一请求发出时，磁盘正好在正确的地方。

20. 旋转时间为 200 msec。顺序读取所有扇区需要旋转 1/2 周以到达扇区 0，以及旋转 2.75 周以读取数据(扇区 7 读取之后，传输完成)。因此，需要旋转 3.25 周，也就是 650 msec。650 msec 读取 4K，其速率为 6302 字节/sec。而对于非交叉方式磁盘，需要 300 msec 读取 4K，也就是 13,653 字节/sec。

21. 也许要，也许不要。如果跨道时磁头移动少于 2 个扇区，就不需要柱面倾斜。如果大于 2 个扇区，则需要柱面倾斜。

22. 驱动器容量和传输速率是原来的 2 倍。寻道时间和平均旋转延时是相同的。

23. 一个相当明显的后果是没有哪个操作系统可以生效，因为这些操作系统都会在原来的分区表位置查找分区。改变分区表格式将使所有操作系统都失败。分区表的唯一方法是同时改变所有操作系统以使用新的格式。

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

28. 一年的平均秒数为 365.25 X 24 X 3600 = 31,557,600。计数器大约在 2^{32}秒之后回绕。2^{32}/31,557,600 = 136.1 年，也就是大约在 2106 年 2 月。当然，到那时所有计算机至少是 64 位的，因此该情形将不会发生。

29. 每条线需要 3200 X 8 = 25,600 次采样/sec。对于每次采样 1 nsec，每秒钟内每条线都需耗时 25.6 msec。如果为 39 条线程，处理器忙碌时间为 39 X 25.6 = 998.4 msec/sec，也就是说该插卡容纳最多 39 条线。

30. 向 RS232 终端写入一个字符之后，在它被打印之前，需花费(比较)长的时间。如果只是

等待浪费时间，因此使用中断。而内存映射终端，字符即时被接受，因此中断没有意义。

31. 按照 56 Kbps 速率，每秒 5600 次中断，总共 5600 X 100 nsec = 560 msec。也就是占用 56% 的 CPU 时间。

32. 滚动窗口需要复制 59 行 X 80 字符 = 4720 字符。复制 1 个字符(16 个字节)需 800 nsec，因此整个窗口需要 3.776 msec。向屏幕写 80 个字符需 400 nsec，因此滚动和显示新的一行需 4.176 msec。也就是大约 239.5 行/sec。

34. 它应该移动光标到第 5 行，第 7 列，然后删除 6 个字符。该序列为 ESC [5; 7 H ESC [6 P。

35. 终端内部嵌入的处理器必须复制所有字符，使之上移。从内部看，终端是内存映射的。这是没办法的，除非使用特殊的硬件。

37. 鼠标的最大速率为 200mm/sec，也就是 2000 mickeys/sec。如果每次报告为 3 个字节，输出速率为 6000 字节/sec。

38. 对于 24-位彩色系统，仅能够表示 2^{24}种颜色。这不是颜色的全部。例如，假设某摄影师以 300 中纯蓝绘图，每种都有一点区别。那么，而(R, G, B)中的B只有 8 位，只能表示 256 种蓝色，而无法表示 300 种蓝色。

39. (a) 24 位真彩 RGB 中每像素需 3 个字节，因此该表中的字符为 16 X 24 X 3 = 1152 字节。

(b) 如果每字节 100 纳秒，那么每个字符需 1152 X 100 nsec = 115.2 微秒。也就是大约每秒输出 8681 字符。

40. 重写文本屏幕需要复制 2000 个字节，也就是 20 微秒。重写图形屏幕需要复制 1024 x 768 x 3 = 2,359,296 字节，大约为 23.6 毫秒。

41. 在 Windows 中，OS 自己调用处理程序例程。在 X Window 中，不会发生类似的事情。X 只是获取消息，并在内部处理它。

42. 第一个参数是必需的。首先，坐标是对应于某个窗口的，因此需要 *hdc* 来指定窗口，也就是其坐标原点。第二，如果该矩形落在窗口外面，将被截除，因此还需要该窗口的坐标。第三，矩形颜色和其它属性必须由 *hdc* 指定。因此，*hdc* 是十分重要的。

43. 显示大小为 400 x 160 x 3 = 192,000 字节。10 fps 也就是 1,920,000 字节/sec = 15,360,000 位/sec。占用 100-Mbps 快速以太网带宽的 15%。

44. 网络带宽是共享的，因此 100 个用户同时请求不同的数据，1-Mbps 的网络将使得每个用户的有效带宽为 10-Kbps。对于一个共享网络，Tv 节目是多点传送的，因此视频包只需发送一次，无论多少用户在观看都不影响其效果。而如果是 100 个用户浏览 Web，每个用户的性能可能迅速降低。

第六章 答案

1. 可以使用如下的表示：
/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 的方法是使用文件扩展名。每种扩展名对应于一种文件类型，以及某些程序来处理这种类型的文件。另一个方法是记住由哪个程序创建了该文件，然后运行该程序。Macintosh 就是用这种方法。

3. 这些系统直接把程序载入内存，并且从 word 0(也就是幻数)开始执行。为了避免将 header 作为代码执行，幻数是一条 BRANCH 指令，其目标地址正好在 header 之上。按这种方法，就可能把二进制文件直接读取到新的进程地址空间，并且从 0 运行，甚至无需知道 header 有多大。

4. 如果文件按照记录构造的，而且每个记录的具体位置为关键词，当按照关键词请求记录时，操作系统就会关心记录长度。在这种情况下，系统必须知道记录有多大，才能按照关键词搜索记录。

5. 如果没有 open，每次 read 时都必须指定要打开的文件名。然后，系统必须读取该文件的 i-节点，即使该节点已经在 cache 中。可能出现的问题就是来回从磁盘读入 i-节点。这多少显得有点笨，不过还是可以运行的。

6. 不需要。如果想要再次读取文件，只需随机地访问字节 0 即可。

7. 是的。rename 调用不会改变文件的创建时间和最后的修改时间，但是创建一个新的文件，其创建时间和最后的修改时间都会改为当前的系统时间。另外，如果磁盘满，复制可能会失败。

8. 被映射的文件部分必须从页边界开始，并且其长度是页长的整数倍。每个映射的页把文件本身作为后备存储。

9. 使用诸如/usr/ast/file 之类的文件名。这样看起来就象多层目录的路径名，而只不过是文件名中包含斜线(/)而已。

10. 一种方法是在 read 系统调用中添加额外的参数，以告知从什么地址读取。实际上，每次 read 都会在文件中执行隐含的定位。不过该方法的缺点是：
(1) 每次 read 都需要额外的参数；
(2) 用户需要记录文件指针的位置。

11. “..”部分把搜索移动到/usr，因此../ast 即为/usr/ast。因此，../ast/x 就是/usr/ast/x。

12. 由于这些被浪费的空间在分配单元(文件)之间，而不是在它们内部，因此，这是外部碎片。这类似于交换系统或者纯分段系统中出现的外部碎片。

13. 需要 9 毫秒才开始传输。以 223 字节/秒的传输速率读取 213 字节需要大约 2-10 = 977 微秒，总共 9.977 毫秒。写回磁盘同样需要 9.977 毫秒。这样，复制一个文件需 19.954 毫秒。为了压缩 8-GB 的磁盘空间，也就是 220 个文件。每个文件 19.954 毫秒，总共需要 20,923 秒，也就是 5.8 个小时。显然，在每个文件删除后都压缩磁盘不是一个好办法。

15. 数码相机按顺序记录一系列照片，存储在某种非易失性存储媒介(例如，闪存)上。当照

相机复位时，该媒介被清空。然后，按顺序依次记录一张照片，直到存储满，然后上载到硬

盘上。对于这种应用，相机内部的连续文件系统时最理想。

16. 它在目录项中找到文件第一块的地址。然后，顺着 FAT 中的块指针链，直到定位于它所需要的块。然后记下其块号码，用于下一次 read 系统调用。

17. 间接块可以保存 256 个磁盘地址。与 10 个直接的磁盘地址一道，最大文件有 266 块。由于每块为 1 KB，最大的文件是 266 KB。

19. Elinor 是正确的。表格中同时有 i-节点的 2 个备份是灾难性的，除非都是只读的。最坏的情况就是当两个都同时被更新时。当把 i-节点写回磁盘时，后写入的会把先写入的删除，而磁盘块就丢失了。

20. 硬链接无需额外的磁盘空间，而只需在 i-节点中记录有多少个链接。符号链接需要空间存储所指的文件的名称。符号链接可以指向其他机器上的文件，甚至是 Internet 上的文件。而硬链接只能指向其自己分区中的文件。

21. 位映像需要 B 位。而空闲列表需要 DF 位。

22. 位映像为：

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

29. 平均时间为 h + 40(1 - h)。

31. 对于 15000rpm，每旋转一周需 4 毫秒。那么，读取 k 字节的平均存取时间为 8(寻道时间) + 2(旋转半周的时间) + (k/262144) x 4(读取 k 字节的时间)毫秒。

对于 1 KB，2 KB 和 4 KB 的块，访问时间分别为 10.015625 毫秒，10.03125 毫秒和 10.0625 毫秒(几乎没有什么不同)。其数据速率分别为 102240 KB/sec，204162 KB/sec 和 407,056 KB/sec。

37. 需要下列的磁盘操作：

读根目录
读/usr 的 i-节点 读
/usr 的目录 读
/usr/ast 的 i-节点
读/usr/ast 的目录
读/usr/ast/courses 的 i-节点 读
/usr/ast/courses 的目录 读
/usr/ast/courses/os 的 i-节点 读
/usr/ast/courses/os 的目录 读
/usr/ast/courses/os/handout.t 的 i-节点
总共 10 个磁盘读操作。