

**MODERN  
OPERATING  
SYSTEMS**

**FOURTH EDITION**

**PROBLEM SOLUTIONS**

**ANDREW S. TANENBAUM**

**HERBERT BOS**

*Vrije Universiteit  
Amsterdam, The Netherlands*

**PRENTICE HALL**

**UPPER SADDLE RIVER, NJ 07458**

**Copyright Pearson Education, Inc. 2014**

## SOLUTIONS TO CHAPTER 1 PROBLEMS

1. An operating system must provide the users with an extended machine, and it must manage the I/O devices and other system resources. To some extent, these are different functions.
2. Obviously, there are a lot of possible answers. Here are some.  
Mainframe operating system: Claims processing in an insurance company.  
Server operating system: Speech-to-text conversion service for Siri.  
Multiprocessor operating system: Video editing and rendering.  
Personal computer operating system: Word processing application.  
Handheld computer operating system: Context-aware recommendation system.  
Embedded operating system: Programming a DVD recorder for recording TV.  
Sensor-node operating system: Monitoring temperature in a wilderness area.  
Real-time operating system: Air traffic control system.  
Smart-card operating system: Electronic payment.
3. In a timesharing system, multiple users can access and perform computations on a computing system simultaneously using their own terminals. Multiprogramming systems allow a user to run multiple programs simultaneously. All timesharing systems are multiprogramming systems but not all multiprogramming systems are timesharing systems since a multiprogramming system may run on a PC with only one user.
4. Empirical evidence shows that memory access exhibits the principle of locality of reference, where if one location is read then the probability of accessing nearby locations next is very high, particularly the following memory locations. So, by caching an entire cache line, the probability of a cache hit next is increased. Also, modern hardware can do a block transfer of 32 or 64 bytes into a cache line much faster than reading the same data as individual words.
5. The prime reason for multiprogramming is to give the CPU something to do while waiting for I/O to complete. If there is no DMA, the CPU is fully occupied doing I/O, so there is nothing to be gained (at least in terms of CPU utilization) by multiprogramming. No matter how much I/O a program does, the CPU will be 100% busy. This of course assumes the major delay is the wait while data are copied. A CPU could do other work if the I/O were slow for other reasons (arriving on a serial line, for instance).
6. Access to I/O devices (e.g., a printer) is typically restricted for different users. Some users may be allowed to print as many pages as they like, some users may not be allowed to print at all, while some users may be limited to printing only a certain number of pages. These restrictions are set by system administrators based on some policies. Such policies need to be enforced so that user-level programs cannot interfere with them.

7. It is still alive. For example, Intel makes Core i3, i5, and i7 CPUs with a variety of different properties including speed and power consumption. All of these machines are architecturally compatible. They differ only in price and performance, which is the essence of the family idea.
8. A  $25 \times 80$  character monochrome text screen requires a 2000-byte buffer. The  $1200 \times 900$  pixel 24-bit color bitmap requires 3,240,000 bytes. In 1980 these two options would have cost \$10 and \$15,820, respectively. For current prices, check on how much RAM currently costs, probably pennies per MB.
9. Consider fairness and real time. Fairness requires that each process be allocated its resources in a fair way, with no process getting more than its fair share. On the other hand, real time requires that resources be allocated based on the times when different processes must complete their execution. A real-time process may get a disproportionate share of the resources.
10. Most modern CPUs provide two modes of execution: kernel mode and user mode. The CPU can execute every instruction in its instruction set and use every feature of the hardware when executing in kernel mode. However, it can execute only a subset of instructions and use only subset of features when executing in the user mode. Having two modes allows designers to run user programs in user mode and thus deny them access to critical instructions.
11. Number of heads =  $255 \text{ GB} / (65536 * 255 * 512) = 16$   
 Number of platters =  $16/2 = 8$   
 The time for a read operation to complete is seek time + rotational latency + transfer time. The seek time is 11 ms, the rotational latency is 7 ms and the transfer time is 4 ms, so the average transfer takes 22 msec.
12. Choices (a), (c), and (d) should be restricted to kernel mode.
13. It may take 20, 25 or 30 msec to complete the execution of these programs depending on how the operating system schedules them. If  $P0$  and  $P1$  are scheduled on the same CPU and  $P2$  is scheduled on the other CPU, it will take 20 msec. If  $P0$  and  $P2$  are scheduled on the same CPU and  $P1$  is scheduled on the other CPU, it will take 25 msec. If  $P1$  and  $P2$  are scheduled on the same CPU and  $P0$  is scheduled on the other CPU, it will take 30 msec. If all three are on the same CPU, it will take 35 msec.
14. Every nanosecond one instruction emerges from the pipeline. This means the machine is executing 1 billion instructions per second. It does not matter at all how many stages the pipeline has. A 10-stage pipeline with 1 nsec per stage would also execute 1 billion instructions per second. All that matters is how often a finished instruction pops out the end of the pipeline.

15. Average access time =  
     $0.95 \times 1 \text{ nsec}$  (word is in the cache)  
     $+ 0.05 \times 0.99 \times 10 \text{ nsec}$  (word is in RAM, but not in the cache)  
     $+ 0.05 \times 0.01 \times 10,000,000 \text{ nsec}$  (word on disk only)  
     $= 5001.445 \text{ nsec}$   
     $= 5.001445 \mu\text{sec}$
16. Maybe. If the caller gets control back and immediately overwrites the data, when the write finally occurs, the wrong data will be written. However, if the driver first copies the data to a private buffer before returning, then the caller can be allowed to continue immediately. Another possibility is to allow the caller to continue and give it a signal when the buffer may be reused, but this is tricky and error prone.
17. A trap instruction switches the execution mode of a CPU from the user mode to the kernel mode. This instruction allows a user program to invoke functions in the operating system kernel.
18. The process table is needed to store the state of a process that is currently suspended, either ready or blocked. Modern personal computer systems have dozens of processes running even when the user is doing nothing and no programs are open. They are checking for updates, loading email, and many other things. On a UNIX system, use the *ps -a* command to see them. On a Windows system, use the task manager.
19. Mounting a file system makes any files already in the mount-point directory inaccessible, so mount points are normally empty. However, a system administrator might want to copy some of the most important files normally located in the mounted directory to the mount point so they could be found in their normal path in an emergency when the mounted device was being repaired.
20. Fork can fail if there are no free slots left in the process table (and possibly if there is no memory or swap space left). Exec can fail if the file name given does not exist or is not a valid executable file. Unlink can fail if the file to be unlinked does not exist or the calling process does not have the authority to unlink it.
21. Time multiplexing: CPU, network card, printer, keyboard.  
Space multiplexing: memory, disk.  
Both: display.
22. If the call fails, for example because *fd* is incorrect, it can return  $-1$ . It can also fail because the disk is full and it is not possible to write the number of bytes requested. On a correct termination, it always returns *nbytes*.

23. It contains the bytes: 1, 5, 9, 2.
24. Time to retrieve the file =  
    1 \* 50 ms (Time to move the arm over track 50)  
    + 5 ms (Time for the first sector to rotate under the head)  
    + 10/200 \* 1000 ms (Read 10 MB)  
    = 105 ms
25. Block special files consist of numbered blocks, each of which can be read or written independently of all the other ones. It is possible to seek to any block and start reading or writing. This is not possible with character special files.
26. System calls do not really have names, other than in a documentation sense. When the library procedure *read* traps to the kernel, it puts the number of the system call in a register or on the stack. This number is used to index into a table. There is really no name used anywhere. On the other hand, the name of the library procedure is very important, since that is what appears in the program.
27. This allows an executable program to be loaded in different parts of the machine's memory in different runs. Also, it enables program size to exceed the size of the machine's memory.
28. As far as program logic is concerned, it does not matter whether a call to a library procedure results in a system call. But if performance is an issue, if a task can be accomplished without a system call the program will run faster. Every system call involves overhead time in switching from the user context to the kernel context. Furthermore, on a multiuser system the operating system may schedule another process to run when a system call completes, further slowing the progress in real time of a calling process.
29. Several UNIX calls have no counterpart in the Win32 API:
- Link: a Win32 program cannot refer to a file by an alternative name or see it in more than one directory. Also, attempting to create a link is a convenient way to test for and create a lock on a file.
- Mount and umount: a Windows program cannot make assumptions about standard path names because on systems with multiple disk drives the drive-name part of the path may be different.
- Chmod: Windows uses access control lists.
- Kill: Windows programmers cannot kill a misbehaving program that is not co-operating.

- 30.** Every system architecture has its own set of instructions that it can execute. Thus a Pentium cannot execute SPARC programs and a SPARC cannot execute Pentium programs. Also, different architectures differ in bus architecture used (such as VME, ISA, PCI, MCA, SBus, ...) as well as the word size of the CPU (usually 32 or 64 bit). Because of these differences in hardware, it is not feasible to build an operating system that is completely portable. A highly portable operating system will consist of two high-level layers—a machine-dependent layer and a machine-independent layer. The machine-dependent layer addresses the specifics of the hardware and must be implemented separately for every architecture. This layer provides a uniform interface on which the machine-independent layer is built. The machine-independent layer has to be implemented only once. To be highly portable, the size of the machine-dependent layer must be kept as small as possible.
- 31.** Separation of policy and mechanism allows OS designers to implement a small number of basic primitives in the kernel. These primitives are simplified, because they are not dependent of any specific policy. They can then be used to implement more complex mechanisms and policies at the user level.
- 32.** The virtualization layer introduces increased memory usage and processor overhead as well as increased performance overhead.
- 33.** The conversions are straightforward:
- (a) A nanoyear is  $10^{-9} \times 365 \times 24 \times 3600 = 31.536$  msec.
  - (b) 1 meter
  - (c) There are  $2^{50}$  bytes, which is 1,099,511,627,776 bytes.
  - (d) It is  $6 \times 10^{24}$  kg or  $6 \times 10^{27}$  g.

**SOLUTIONS TO CHAPTER 2 PROBLEMS**

1. The transition from blocked to running is conceivable. Suppose that a process is blocked on I/O and the I/O finishes. If the CPU is otherwise idle, the process could go directly from blocked to running. The other missing transition, from ready to blocked, is impossible. A ready process cannot do I/O or anything else that might block it. Only a running process can block.
2. You could have a register containing a pointer to the current process-table entry. When I/O completed, the CPU would store the current machine state in the current process-table entry. Then it would go to the interrupt vector for the interrupting device and fetch a pointer to another process-table entry (the service procedure). This process would then be started up.
3. Generally, high-level languages do not allow the kind of access to CPU hardware that is required. For instance, an interrupt handler may be required to enable and disable the interrupt servicing a particular device, or to manipulate data within a process' stack area. Also, interrupt service routines must execute as rapidly as possible.
4. There are several reasons for using a separate stack for the kernel. Two of them are as follows. First, you do not want the operating system to crash because a poorly written user program does not allow for enough stack space. Second, if the kernel leaves stack data in a user program's memory space upon return from a system call, a malicious user might be able to use this data to find out information about other processes.
5. The chance that all five processes are idle is  $1/32$ , so the CPU idle time is  $1/32$ .
6. There is enough room for 14 processes in memory. If a process has an I/O of  $p$ , then the probability that they are all waiting for I/O is  $p^{14}$ . By equating this to 0.01, we get the equation  $p^{14} = 0.01$ . Solving this, we get  $p = 0.72$ , so we can tolerate processes with up to 72% I/O wait.
7. If each job has 50% I/O wait, then it will take 40 minutes to complete in the absence of competition. If run sequentially, the second one will finish 80 minutes after the first one starts. With two jobs, the approximate CPU utilization is  $1 - 0.5^2$ . Thus, each one gets 0.375 CPU minute per minute of real time. To accumulate 20 minutes of CPU time, a job must run for  $20/0.375$  minutes, or about 53.33 minutes. Thus running sequentially the jobs finish after 80 minutes, but running in parallel they finish after 53.33 minutes.
8. The probability that all processes are waiting for I/O is  $0.4^6$  which is 0.004096. Therefore, CPU utilization =  $1 - 0.004096 = 0.995904$ .



9. The client process can create separate threads; each thread can fetch a different part of the file from one of the mirror servers. This can help reduce downtime. Of course, there is a single network link being shared by all threads. This link can become a bottleneck as the number of threads becomes very large.
10. It would be difficult, if not impossible, to keep the file system consistent. Suppose that a client process sends a request to server process 1 to update a file. This process updates the cache entry in its memory. Shortly thereafter, another client process sends a request to server 2 to read that file. Unfortunately, if the file is also cached there, server 2, in its innocence, will return obsolete data. If the first process writes the file through to the disk after caching it, and server 2 checks the disk on every read to see if its cached copy is up-to-date, the system can be made to work, but it is precisely all these disk accesses that the caching system is trying to avoid.
11. No. If a single-threaded process is blocked on the keyboard, it cannot fork.
12. A worker thread will block when it has to read a Web page from the disk. If user-level threads are being used, this action will block the entire process, destroying the value of multithreading. Thus it is essential that kernel threads are used to permit some threads to block without affecting the others.
13. Yes. If the server is entirely CPU bound, there is no need to have multiple threads. It just adds unnecessary complexity. As an example, consider a telephone directory assistance number (like 555-1212) for an area with 1 million people. If each (name, telephone number) record is, say, 64 characters, the entire database takes 64 megabytes and can easily be kept in the server's memory to provide fast lookup.
14. When a thread is stopped, it has values in the registers. They must be saved, just as when the process is stopped. the registers must be saved. Multiprogramming threads is no different than multiprogramming processes, so each thread needs its own register save area.
15. Threads in a process cooperate. They are not hostile to one another. If yielding is needed for the good of the application, then a thread will yield. After all, it is usually the same programmer who writes the code for all of them.
16. User-level threads cannot be preempted by the clock unless the whole process' quantum has been used up (although transparent clock interrupts can happen). Kernel-level threads can be preempted individually. In the latter case, if a thread runs too long, the clock will interrupt the current process and thus the current thread. The kernel is free to pick a different thread from the same process to run next if it so desires.

17. In the single-threaded case, the cache hits take 12 msec and cache misses take 87 msec. The weighted average is  $2/3 \times 12 + 1/3 \times 87$ . Thus, the mean request takes 37 msec and the server can do about 27 per second. For a multi-threaded server, all the waiting for the disk is overlapped, so every request takes 12 msec, and the server can handle  $83 \frac{1}{3}$  requests per second.
18. The biggest advantage is the efficiency. No traps to the kernel are needed to switch threads. The biggest disadvantage is that if one thread blocks, the entire process blocks.
19. Yes, it can be done. After each call to *pthread\_create*, the main program could do a *pthread\_join* to wait until the thread just created has exited before creating the next thread.
20. The pointers are really necessary because the size of the global variable is unknown. It could be anything from a character to an array of floating-point numbers. If the value were stored, one would have to give the size to *create\_global*, which is all right, but what type should the second parameter of *set\_global* be, and what type should the value of *read\_global* be?
21. It could happen that the runtime system is precisely at the point of blocking or unblocking a thread, and is busy manipulating the scheduling queues. This would be a very inopportune moment for the clock interrupt handler to begin inspecting those queues to see if it was time to do thread switching, since they might be in an inconsistent state. One solution is to set a flag when the runtime system is entered. The clock handler would see this and set its own flag, then return. When the runtime system finished, it would check the clock flag, see that a clock interrupt occurred, and now run the clock handler.
22. Yes it is possible, but inefficient. A thread wanting to do a system call first sets an alarm timer, then does the call. If the call blocks, the timer returns control to the threads package. Of course, most of the time the call will not block, and the timer has to be cleared. Thus each system call that might block has to be executed as three system calls. If timers go off prematurely, all kinds of problems develop. This is not an attractive way to build a threads package.
23. Yes, it still works, but it still is busy waiting, of course.
24. It certainly works with preemptive scheduling. In fact, it was designed for that case. When scheduling is nonpreemptive, it might fail. Consider the case in which *turn* is initially 0 but process 1 runs first. It will just loop forever and never release the CPU.
25. The priority inversion problem occurs when a low-priority process is in its critical region and suddenly a high-priority process becomes ready and is scheduled. If it uses busy waiting, it will run forever. With user-level threads, it cannot happen that a low-priority thread is suddenly preempted to allow a

high-priority thread run. There is no preemption. With kernel-level threads this problem can arise.

26. With round-robin scheduling it works. Sooner or later  $L$  will run, and eventually it will leave its critical region. The point is, with priority scheduling,  $L$  never gets to run at all; with round robin, it gets a normal time slice periodically, so it has the chance to leave its critical region.
27. Each thread calls procedures on its own, so it must have its own stack for the local variables, return addresses, and so on. This is equally true for user-level threads as for kernel-level threads.
28. Yes. The simulated computer could be multiprogrammed. For example, while process  $A$  is running, it reads out some shared variable. Then a simulated clock tick happens and process  $B$  runs. It also reads out the same variable. Then it adds 1 to the variable. When process  $A$  runs, if it also adds 1 to the variable, we have a race condition.
29. Yes, it will work as is. At a given time instant, only one producer (consumer) can add (remove) an item to (from) the buffer.
30. The solution satisfies mutual exclusion since it is not possible for both processes to be in their critical section. That is, when turn is 0,  $P0$  can execute its critical section, but not  $P1$ . Likewise, when turn is 1. However, this assumes  $P0$  must run first. If  $P1$  produces something and it puts it in a buffer, then while  $P0$  can get into its critical section, it will find the buffer empty and block. Also, this solution requires strict alternation of the two processes, which is undesirable.
31. To do a semaphore operation, the operating system first disables interrupts. Then it reads the value of the semaphore. If it is doing a down and the semaphore is equal to zero, it puts the calling process on a list of blocked processes associated with the semaphore. If it is doing an up, it must check to see if any processes are blocked on the semaphore. If one or more processes are blocked, one of them is removed from the list of blocked processes and made runnable. When all these operations have been completed, interrupts can be enabled again.
32. Associated with each counting semaphore are two binary semaphores,  $M$ , used for mutual exclusion, and  $B$ , used for blocking. Also associated with each counting semaphore is a counter that holds the number of ups minus the number of downs, and a list of processes blocked on that semaphore. To implement down, a process first gains exclusive access to the semaphores, counter, and list by doing a down on  $M$ . It then decrements the counter. If it is zero or more, it just does an up on  $M$  and exits. If  $M$  is negative, the process is put on the list of blocked processes. Then an up is done on  $M$  and a down is done on  $B$  to block the process. To implement up, first  $M$  is downed to get mutual

exclusion, and then the counter is incremented. If it is more than zero, no one was blocked, so all that needs to be done is to up  $M$ . If, however, the counter is now negative or zero, some process must be removed from the list. Finally, an up is done on  $B$  and  $M$  in that order.

33. If the program operates in phases and neither process may enter the next phase until both are finished with the current phase, it makes perfect sense to use a barrier.
34. With kernel threads, a thread can block on a semaphore and the kernel can run some other thread in the same process. Consequently, there is no problem using semaphores. With user-level threads, when one thread blocks on a semaphore, the kernel thinks the entire process is blocked and does not run it ever again. Consequently, the process fails.
35. It is very expensive to implement. Each time any variable that appears in a predicate on which some process is waiting changes, the run-time system must re-evaluate the predicate to see if the process can be unblocked. With the Hoare and Brinch Hansen monitors, processes can only be awakened on a signal primitive.
36. The employees communicate by passing messages: orders, food, and bags in this case. In UNIX terms, the four processes are connected by pipes.
37. It does not lead to race conditions (nothing is ever lost), but it is effectively busy waiting.
38. It will take  $nT$  sec.
39. Three processes are created. After the initial process forks, there are two processes running, a parent and a child. Each of them then forks, creating two additional processes. Then all the processes exit.
40. If a process occurs multiple times in the list, it will get multiple quanta per cycle. This approach could be used to give more important processes a larger share of the CPU. But when the process blocks, all entries had better be removed from the list of runnable processes.
41. In simple cases it may be possible to see if I/O will be limiting by looking at source code. For instance a program that reads all its input files into buffers at the start will probably not be I/O bound, but a program that reads and writes incrementally to a number of different files (such as a compiler) is likely to be I/O bound. If the operating system provides a facility such as the UNIX *ps* command that can tell you the amount of CPU time used by a program, you can compare this with the total time to complete execution of the program. This is, of course, most meaningful on a system where you are the only user.

42. If the context switching time is large, then the time quantum value has to be proportionally large. Otherwise, the overhead of context switching can be quite high. Choosing large time quantum values can lead to an inefficient system if the typical CPU burst times are less than the time quantum. If context switching is very small or negligible, then the time quantum value can be chosen with more freedom.
43. The CPU efficiency is the useful CPU time divided by the total CPU time. When  $Q \geq T$ , the basic cycle is for the process to run for  $T$  and undergo a process switch for  $S$ . Thus, (a) and (b) have an efficiency of  $T/(S + T)$ . When the quantum is shorter than  $T$ , each run of  $T$  will require  $T/Q$  process switches, wasting a time  $ST/Q$ . The efficiency here is then

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

which reduces to  $Q/(Q + S)$ , which is the answer to (c). For (d), we just substitute  $Q$  for  $S$  and find that the efficiency is 50%. Finally, for (e), as  $Q \rightarrow 0$  the efficiency goes to 0.

44. Shortest job first is the way to minimize average response time.
- $0 < X \leq 3$ :  $X, 3, 5, 6, 9$ .
  - $3 < X \leq 5$ :  $3, X, 5, 6, 9$ .
  - $5 < X \leq 6$ :  $3, 5, X, 6, 9$ .
  - $6 < X \leq 9$ :  $3, 5, 6, X, 9$ .
  - $X > 9$ :  $3, 5, 6, 9, X$ .
45. For round robin, during the first 10 minutes each job gets 1/5 of the CPU. At the end of 10 minutes,  $C$  finishes. During the next 8 minutes, each job gets 1/4 of the CPU, after which time  $D$  finishes. Then each of the three remaining jobs gets 1/3 of the CPU for 6 minutes, until  $B$  finishes, and so on. The finishing times for the five jobs are 10, 18, 24, 28, and 30, for an average of 22 minutes. For priority scheduling,  $B$  is run first. After 6 minutes it is finished. The other jobs finish at 14, 24, 26, and 30, for an average of 18.8 minutes. If the jobs run in the order  $A$  through  $E$ , they finish at 10, 16, 18, 22, and 30, for an average of 19.2 minutes. Finally, shortest job first yields finishing times of 2, 6, 12, 20, and 30, for an average of 14 minutes.
46. The first time it gets 1 quantum. On succeeding runs it gets 2, 4, 8, and 15, so it must be swapped in 5 times.
47. Each voice call needs 200 samples of 1 msec or 200 msec. Together they use 400 msec of CPU time. The video needs 11 msec 33 1/3 times a second for a total of about 367 msec. The sum is 767 msec per second of real time so the system is schedulable.

48. Another video stream consumes 367 msec of time per second for a total of 1134 msec per second of real time so the system is not schedulable.
49. The sequence of predictions is 40, 30, 35, and now 25.
50. The fraction of the CPU used is  $35/50 + 20/100 + 10/200 + x/250$ . To be schedulable, this must be less than 1. Thus  $x$  must be less than 12.5 msec.
51. Yes. There will be always at least one fork free and at least one philosopher that can obtain both forks simultaneously. Hence, there will be no deadlock. You can try this for  $N = 2$ ,  $N = 3$  and  $N = 4$  and then generalize.
52. Each voice call runs 166.67 times/second and uses up 1 msec per burst, so each voice call needs 166.67 msec per second or 333.33 msec for the two of them. The video runs 25 times a second and uses up 20 msec each time, for a total of 500 msec per second. Together they consume 833.33 msec per second, so there is time left over and the system is schedulable.
53. The kernel could schedule processes by any means it wishes, but within each process it runs threads strictly in priority order. By letting the user process set the priority of its own threads, the user controls the policy but the kernel handles the mechanism.
54. If a philosopher blocks, neighbors can later see that she is hungry by checking his state, in *test*, so he can be awakened when the forks are available.
55. The change would mean that after a philosopher stopped eating, neither of his neighbors could be chosen next. In fact, they would never be chosen. Suppose that philosopher 2 finished eating. He would run *test* for philosophers 1 and 3, and neither would be started, even though both were hungry and both forks were available. Similarly, if philosopher 4 finished eating, philosopher 3 would not be started. Nothing would start him.
56. Variation 1: readers have priority. No writer may start when a reader is active. When a new reader appears, it may start immediately unless a writer is currently active. When a writer finishes, if readers are waiting, they are all started, regardless of the presence of waiting writers. Variation 2: Writers have priority. No reader may start when a writer is waiting. When the last active process finishes, a writer is started, if there is one; otherwise, all the readers (if any) are started. Variation 3: symmetric version. When a reader is active, new readers may start immediately. When a writer finishes, a new writer has priority, if one is waiting. In other words, once we have started reading, we keep reading until there are no readers left. Similarly, once we have started writing, all pending writers are allowed to run.

57. A possible shell script might be

```
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
```

Run the script twice simultaneously, by starting it once in the background (using `&`) and again in the foreground. Then examine the file *numbers*. It will probably start out looking like an orderly list of numbers, but at some point it will lose its orderliness, due to the race condition created by running two copies of the script. The race can be avoided by having each copy of the script test for and set a lock on the file before entering the critical area, and unlocking it upon leaving the critical area. This can be done like this:

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

This version will just skip a turn when the file is inaccessible. Variant solutions could put the process to sleep, do busy waiting, or count only loops in which the operation is successful.

**SOLUTIONS TO CHAPTER 3 PROBLEMS**

1. First, special hardware is needed to do the comparisons, and it must be fast, since it is used on every memory reference. Second, with 4-bit keys, only 16 programs can be in memory at once (one of which is the operating system).
2. It is an accident. The base register is 16,384 because the program happened to be loaded at address 16,384. It could have been loaded anywhere. The limit register is 16,384 because the program contains 16,384 bytes. It could have been any length. That the load address happens to exactly match the program length is pure coincidence.
3. Almost the entire memory has to be copied, which requires each word to be read and then rewritten at a different location. Reading 4 bytes takes 4 nsec, so reading 1 byte takes 1 nsec and writing it takes another 2 nsec, for a total of 2 nsec per byte compacted. This is a rate of 500,000,000 bytes/sec. To copy 4 GB ( $2^{232}$  bytes, which is about  $4.295 \times 10^9$  bytes), the computer needs  $2^{32}/500,000,000$  sec, which is about 859 msec. This number is slightly pessimistic because if the initial hole at the bottom of memory is  $k$  bytes, those  $k$  bytes do not need to be copied. However, if there are many holes and many data segments, the holes will be small, so  $k$  will be small and the error in the calculation will also be small.
4. First fit takes 20 MB, 10 MB, 18 MB. Best fit takes 12 MB, 10 MB, and 9 MB. Worst fit takes 20 MB, 18 MB, and 15 MB. Next fit takes 20 MB, 18 MB, and 9 MB.
5. Real memory uses physical addresses. These are the numbers that the memory chips react to on the bus. Virtual addresses are the logical addresses that refer to a process' address space. Thus a machine with a 32-bit word can generate virtual addresses up to 4 GB regardless of whether the machine has more or less memory than 4 GB.
6. For a 4-KB page size the (page, offset) pairs are (4, 3616), (8, 0), and (14, 2656). For an 8-KB page size they are (2, 3616), (4, 0), and (7, 2656).
7. (a) 8212. (b) 4100. (c) 24684.
8. They built an MMU and inserted it between the 8086 and the bus. Thus all 8086 physical addresses went into the MMU as virtual addresses. The MMU then mapped them onto physical addresses, which went to the bus.
9. There needs to be an MMU that can remap virtual pages to physical pages. Also, when a page not currently mapped is referenced, there needs to be a trap to the operating system so it can fetch the page.



- 10.** If the smartphone supports multiprogramming, which the iPhone, Android, and Windows phones all do, then multiple processes are supported. If a process forks and pages are shared between parent and child, copy on write definitely makes sense. A smartphone is smaller than a server, but logically it is not so different.
- 11.** For these sizes
- (a)  $M$  has to be at least 4096 to ensure a TLB miss for every access to an element of  $X$ . Since  $N$  affects only how many times  $X$  is accessed, any value of  $N$  will do.
  - (b)  $M$  should still be at least 4,096 to ensure a TLB miss for every access to an element of  $X$ . But now  $N$  should be greater than 64K to thrash the TLB, that is,  $X$  should exceed 256 KB.
- 12.** The total virtual address space for all the processes combined is  $nv$ , so this much storage is needed for pages. However, an amount  $r$  can be in RAM, so the amount of disk storage required is only  $nv - r$ . This amount is far more than is ever needed in practice because rarely will there be  $n$  processes actually running and even more rarely will all of them need the maximum allowed virtual memory.
- 13.** A page fault every  $k$  instructions adds an extra overhead of  $n/k$   $\mu$ sec to the average, so the average instruction takes  $1 + n/k$  nsec.
- 14.** The page table contains  $2^{32}/2^{13}$  entries, which is 524,288. Loading the page table takes 52 msec. If a process gets 100 msec, this consists of 52 msec for loading the page table and 48 msec for running. Thus 52% of the time is spent loading page tables.
- 15.** Under these circumstances:
- (a) We need one entry for each page, or  $2^{24} = 16 \times 1024 \times 1024$  entries, since there are  $36 = 48 - 12$  bits in the page number field.
  - (b) Instruction addresses will hit 100% in the TLB. The data pages will have a 100 hit rate until the program has moved onto the next data page. Since a 4-KB page contains 1,024 long integers, there will be one TLB miss and one extra memory access for every 1,024 data references.
- 16.** The chance of a hit is 0.99 for the TLB, 0.0099 for the page table, and 0.0001 for a page fault (i.e., only 1 in 10,000 references will cause a page fault). The effective address translation time in nsec is then:
- $$0.99 \times 1 + 0.0099 \times 100 + 0.0001 \times 6 \times 10^6 \approx 602 \text{ clock cycles.}$$
- Note that the effective address translation time is quite high because it is dominated by the page replacement time even when page faults only occur once in 10,000 references.

17. Consider,
- A multilevel page table reduces the number of actual pages of the page table that need to be in memory because of its hierarchic structure. In fact, in a program with lots of instruction and data locality, we only need the top-level page table (one page), one instruction page, and one data page.
  - Allocate 12 bits for each of the three page fields. The offset field requires 14 bits to address 16 KB. That leaves 24 bits for the page fields. Since each entry is 4 bytes, one page can hold  $2^{12}$  page table entries and therefore requires 12 bits to index one page. So allocating 12 bits for each of the page fields will address all  $2^{38}$  bytes.
18. The virtual address was changed from (PT1, PT2, Offset) to (PT1, PT2, PT3, Offset). But the virtual address still used only 32 bits. The bit configuration of a virtual address changed from (10, 10, 12) to (2, 9, 9, 12)
19. Twenty bits are used for the virtual page numbers, leaving 12 over for the offset. This yields a 4-KB page. Twenty bits for the virtual page implies  $2^{20}$  pages.
20. For a one-level page table, there are  $2^{32}/2^{12}$  or 1M pages needed. Thus the page table must have 1M entries. For two-level paging, the main page table has 1K entries, each of which points to a second page table. Only two of these are used. Thus, in total only three page table entries are needed, one in the top-level table and one in each of the lower-level tables.
21. The code and reference string are as follows
- |              |             |
|--------------|-------------|
| LOAD 6144,R0 | 1(I), 12(D) |
| PUSH R0      | 2(I), 15(D) |
| CALL 5120    | 2(I), 15(D) |
| JEQ 5152     | 10(I)       |
- The code (I) indicates an instruction reference, whereas (D) indicates a data reference.
22. The effective instruction time is  $1h + 5(1 - h)$ , where  $h$  is the hit rate. If we equate this formula with 2 and solve for  $h$ , we find that  $h$  must be at least 0.75.
23. An associative memory essentially compares a key to the contents of multiple registers simultaneously. For each register there must be a set of comparators that compare each bit in the register contents to the key being searched for. The number of gates (or transistors) needed to implement such a device is a linear function of the number of registers, so expanding the design gets expensive linearly.

24. With 8-KB pages and a 48-bit virtual address space, the number of virtual pages is  $2^{48}/2^{13}$ , which is  $2^{35}$  (about 34 billion).
25. The main memory has  $2^{28}/2^{13} = 32,768$  pages. A 32K hash table will have a mean chain length of 1. To get under 1, we have to go to the next size, 65,536 entries. Spreading 32,768 entries over 65,536 table slots will give a mean chain length of 0.5, which ensures fast lookup.
26. This is probably not possible except for the unusual and not very useful case of a program whose course of execution is completely predictable at compilation time. If a compiler collects information about the locations in the code of calls to procedures, this information might be used at link time to rearrange the object code so that procedures were located close to the code that calls them. This would make it more likely that a procedure would be on the same page as the calling code. Of course this would not help much for procedures called from many places in the program.
27. Under these circumstances
- Every reference will page fault unless the number of page frames is 512, the length of the entire sequence.
  - If there are 500 frames, map pages 0–498 to fixed frames and vary only one frame.

28. The page frames for FIFO are as follows:

```
x0172333300
x x017222233
x xx01777722
x xxx0111177
```

The page frames for LRU are as follows:

```
x0172327103
x x017232710
x xx01773271
x xxx0111327
```

FIFO yields six page faults; LRU yields seven.

29. The first page with a 0 bit will be chosen, in this case *D*.
30. The counters are
- Page 0: 0110110
  - Page 1: 01001001
  - Page 2: 00110111
  - Page 3: 10001011

31. The sequence: 0, 1, 2, 1, 2, 0, 3. In LRU, page 1 will be replaced by page 3. In clock, page 1 will be replaced, since all pages will be marked and the cursor is at page 0.
32. The age of the page is  $2204 - 1213 = 991$ . If  $\tau = 400$ , it is definitely out of the working set and was not recently referenced so it will be evicted. The  $\tau = 1000$  situation is different. Now the page falls within the working set (barely), so it is not removed.
33. Consider,
- (a) For every  $R$  bit that is set, set the time-stamp value to 10 and clear all  $R$  bits. You could also change the (0,1)  $R$ - $M$  entries to (0,0\*). So the entries for pages 1 and 2 will change to:

| Page | Time stamp | V | R | M  |
|------|------------|---|---|----|
| 0    | 6          | 1 | 0 | 0* |
| 1    | 10         | 1 | 0 | 0  |
| 2    | 10         | 1 | 0 | 1  |

- (b) Evict page 3 ( $R = 0$  and  $M = 0$ ) and load page 4:

| Page | Time stamp | V | R | M | Notes                  |
|------|------------|---|---|---|------------------------|
| 0    | 6          | 1 | 0 | 1 |                        |
| 1    | 9          | 1 | 1 | 0 |                        |
| 2    | 9          | 1 | 1 | 1 |                        |
| 3    | 7          | 0 | 0 | 0 | Changed from 7 (1,0,0) |
| 4    | 10         | 1 | 1 | 0 | Changed from 4 (0,0,0) |

34. Consider,
- (a) The attributes are: (FIFO) load time; (LRU) latest reference time; and (Optimal) nearest reference time in the future.
- (b) There is the labeling algorithm and the replacement algorithm. The labeling algorithm labels each page with the attribute given in part a. The replacement algorithm evicts the page with the smallest label.
35. The seek plus rotational latency is 10 msec. For 2-KB pages, the transfer time is about 0.009766 msec, for a total of about 10.009766 msec. Loading 32 of these pages will take about 320.21 msec. For 4-KB pages, the transfer time is doubled to about 0.01953 msec, so the total time per page is 10.01953 msec. Loading 16 of these pages takes about 160.3125 msec. With such fast disks, all that matters is reducing the number of transfers (or putting the pages consecutively on the disk).

36. NRU removes page 2. FIFO removes page 3. LRU removes page 1. Second chance removes page 2.
37. Sharing pages brings up all kinds of complications and options:
- (a) The page table update should be delayed for process *B* if it will never access the shared page or if it accesses it when the page has been swapped out again. Unfortunately, in the general case, we do not know what process *B* will do in the future.
  - (b) The cost is that this lazy page fault handling can incur more page faults. The overhead of each page fault plays an important role in determining if this strategy is more efficient. (*Aside:* This cost is similar to that faced by the copy-on-write strategy for supporting some UNIX fork system call implementations.)
38. Fragment *B* since the code has more spatial locality than Fragment *A*. The inner loop causes only one page fault for every other iteration of the outer loop. (There will be only 32 page faults.) [*Aside* (Fragment *A*): Since a frame is 128 words, one row of the *X* array occupies half of a page (i.e., 64 words). The entire array fits into  $64 \times 32 / 128 = 16$  frames. The inner loop of the code steps through consecutive rows of *X* for a given column. Thus, every other reference to  $X[i][j]$  will cause a page fault. The total number of page faults will be  $64 \times 64 / 2 = 2,048$ ].
39. It can certainly be done.
- (a) The approach has similarities to using flash memory as a paging device in smartphones except now the virtual swap area is a RAM located on a remote server. All of the software infrastructure for the virtual swap area would have to be developed.
  - (b) The approach might be worthwhile by noting that the access time of disk drives is in the millisecond range while the access time of RAM via a network connection is in the microsecond range if the software overhead is not too high. But the approach might make sense only if there is lots of idle RAM in the server farm. And then, there is also the issue of reliability. Since RAM is volatile, the virtual swap area would be lost if the remote server went down.
40. The PDP-1 paging drum had the advantage of no rotational latency. This saved half a rotation each time memory was written to the drum.
41. The text is eight pages, the data are five pages, and the stack is four pages. The program does not fit because it needs 17 4096-byte pages. With a 512-byte page, the situation is different. Here the text is 64 pages, the data are 33 pages, and the stack is 31 pages, for a total of 128 512-byte pages, which fits. With the small page size it is OK, but not with the large one.

42. The program is getting 15,000 page faults, each of which uses 2 msec of extra processing time. Together, the page fault overhead is 30 sec. This means that of the 60 sec used, half was spent on page fault overhead, and half on running the program. If we run the program with twice as much memory, we get half as many memory page faults, and only 15 sec of page fault overhead, so the total run time will be 45 sec.
43. It works for the program if the program cannot be modified. It works for the data if the data cannot be modified. However, it is common that the program cannot be modified and extremely rare that the data cannot be modified. If the data area on the binary file were overwritten with updated pages, the next time the program was started, it would not have the original data.
44. The instruction could lie astride a page boundary, causing two page faults just to fetch the instruction. The word fetched could also span a page boundary, generating two more faults, for a total of four. If words must be aligned in memory, the data word can cause only one fault, but an instruction to load a 32-bit word at address 4094 on a machine with a 4-KB page is legal on some machines (including the x86).
45. Internal fragmentation occurs when the last allocation unit is not full. External fragmentation occurs when space is wasted between two allocation units. In a paging system, the wasted space in the last page is lost to internal fragmentation. In a pure segmentation system, some space is invariably lost between the segments. This is due to external fragmentation.
46. No. The search key uses both the segment number and the virtual page number, so the exact page can be found in a single match.
47. Here are the results:

|     | Address | Fault?  |
|-----|---------|---|
| (a) | (14, 3) | No (or 0xD3 or 1110 0011)                       |
| (b) | NA      | Protection fault: Write to read/execute segment |
| (c) | NA      | Page fault                                      |
| (d) | NA      | Protection fault: Jump to read/write segment    |

48. General virtual memory support is not needed when the memory requirements of all applications are well known and controlled. Some examples are smart cards, special-purpose processors (e.g., network processors), and embedded processors. In these situations, we should always consider the possibility of using more real memory. If the operating system did not have to support virtual memory, the code would be much simpler and smaller. On the other hand, some ideas from virtual memory may still be profitably exploited, although with different design requirements. For example, program/thread isolation might be paging to flash memory.

- 49.** This question addresses one aspect of virtual machine support. Recent attempts include Denali, Xen, and VMware. The fundamental hurdle is how to achieve near-native performance, that is, as if the executing operating system had memory to itself. The problem is how to quickly switch to another operating system and therefore how to deal with the TLB. Typically, you want to give some number of TLB entries to each kernel and ensure that each kernel operates within its proper virtual memory context. But sometimes the hardware (e.g., some Intel architectures) wants to handle TLB misses without knowledge of what you are trying to do. So, you need to either handle the TLB miss in software or provide hardware support for tagging TLB entries with a context ID.

**SOLUTIONS TO CHAPTER 4 PROBLEMS**

1. You can go up and down the tree as often as you want using “..”. Some of the many paths are

```
/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. The Windows way is to use the file extension. Each extension corresponds to a file type and to some program that handles that type. Another way is to remember which program created the file and run that program. The Macintosh works this way.
3. These systems loaded the program directly in memory and began executing at word 0, which was the magic number. To avoid trying to execute the header as code, the magic number was a BRANCH instruction with a target address just above the header. In this way it was possible to read the binary file directly into the new process’ address space and run it at 0, without even knowing how big the header was.
4. To start with, if there were no open, on every read it would be necessary to specify the name of the file to be opened. The system would then have to fetch the i-node for it, although that could be cached. One issue that quickly arises is when to flush the i-node back to disk. It could time out, however. It would be a bit clumsy, but it might work.
5. No. If you want to read the file again, just randomly access byte 0.
6. Yes. The rename call does not change the creation time or the time of last modification, but creating a new file causes it to get the current time as both the creation time and the time of last modification. Also, if the disk is nearly full, the copy might fail.
7. The mapped portion of the file must start at a page boundary and be an integral number of pages in length. Each mapped page uses the file itself as backing store. Unmapped memory uses a scratch file or partition as backing store.
8. Use file names such as */usr/ast/file*. While it looks like a hierarchical path name, it is really just a single name containing embedded slashes.



9. One way is to add an extra parameter to the read system call that tells what address to read from. In effect, every read then has a potential for doing a seek within the file. The disadvantages of this scheme are (1) an extra parameter in every read call, and (2) requiring the user to keep track of where the file pointer is.
10. The dotdot component moves the search to */usr*, so *../ast* puts it in */usr/ast*. Thus *../ast/x* is the same as */usr/ast/x*.
11. Since the wasted storage is *between* the allocation units (files), not inside them, this is external fragmentation. It is precisely analogous to the external fragmentation of main memory that occurs with a swapping system or a system using pure segmentation.
12. If a data block gets corrupted in a contiguous allocation system, then only this block is affected; the remainder of the file's blocks can be read. In the case of linked allocation, the corrupted block cannot be read; also, location data about all blocks starting from this corrupted block is lost. In case of indexed allocation, only the corrupted data block is affected.
13. It takes 9 msec to start the transfer. To read  $2^{13}$  bytes at a transfer rate of 80 MB/sec requires 0.0977 msec, for a total of 9.0977 msec. Writing it back takes another 9.0977 msec. Thus, copying a file takes 18.1954 msec. To compact half of a 16-GB disk would involve copying 8 GB of storage, which is  $2^{20}$  files. At 18.1954 msec per file, this takes 19,079.25 sec, which is 5.3 hours. Clearly, compacting the disk after every file removal is not a great idea.
14. If done right, yes. While compacting, each file should be organized so that all of its blocks are consecutive, for fast access. Windows has a program that defragments and reorganizes the disk. Users are encouraged to run it periodically to improve system performance. But given how long it takes, running once a month might be a good frequency.
15. A digital still camera records some number of photographs in sequence on a nonvolatile storage medium (e.g., flash memory). When the camera is reset, the medium is emptied. Thereafter, pictures are recorded one at a time in sequence until the medium is full, at which time they are uploaded to a hard disk. For this application, a contiguous file system inside the camera (e.g., on the picture storage medium) is ideal.
16. The indirect block can hold 128 disk addresses. Together with the 10 direct disk addresses, the maximum file has 138 blocks. Since each block is 1 KB, the largest file is 138 KB.
17. For random access, table/indexed and contiguous will be both appropriate, while linked allocation is not as it typically requires multiple disk reads for a given record.

18. Since the file size changes a lot, contiguous allocation will be inefficient requiring reallocation of disk space as the file grows in size and compaction of free blocks as the file shrinks in size. Both linked and table/indexed allocation will be efficient; between the two, table/indexed allocation will be more efficient for random-access scenarios.
19. There must be a way to signal that the address-block pointers hold data, rather than pointers. If there is a bit left over somewhere among the attributes, it can be used. This leaves all nine pointers for data. If the pointers are  $k$  bytes each, the stored file could be up to  $9k$  bytes long. If no bit is left over among the attributes, the first disk address can hold an invalid address to mark the following bytes as data rather than pointers. In that case, the maximum file is  $8k$  bytes.
20. Elinor is right. Having two copies of the i-node in the table at the same time is a disaster, unless both are read only. The worst case is when both are being updated simultaneously. When the i-nodes are written back to the disk, whichever one gets written last will erase the changes made by the other one, and disk blocks will be lost.
21. Hard links do not require any extra disk space, just a counter in the i-node to keep track of how many there are. Symbolic links need space to store the name of the file pointed to. Symbolic links can point to files on other machines, even over the Internet. Hard links are restricted to pointing to files within their own partition.
22. A single i-node is pointed to by all directory entries of hard links for a given file. In the case of soft-links, a new i-node is created for the soft link and this inode essentially points to the original file being linked.
23. The number of blocks on the disk =  $4 \text{ TB} / 4 \text{ KB} = 2^{30}$ . Thus, each block address can be 32 bits (4 bytes), the nearest power of 2. Thus, each block can store  $4 \text{ KB} / 4 = 1024$  addresses.
24. The bitmap requires  $B$  bits. The free list requires  $DF$  bits. The free list requires fewer bits if  $DF < B$ . Alternatively, the free list is shorter if  $F/B < 1/D$ , where  $F/B$  is the fraction of blocks free. For 16-bit disk addresses, the free list is shorter if 6% or less of the disk is free.
25. The beginning of the bitmap looks like:
  - (a) After writing file  $B$ : 1111 1111 1111 0000
  - (b) After deleting file  $A$ : 1000 0001 1111 0000
  - (c) After writing file  $C$ : 1111 1111 1111 1100
  - (d) After deleting file  $B$ : 1111 1110 0000 1100

26. It is not a serious problem at all. Repair is straightforward; it just takes time. The recovery algorithm is to make a list of all the blocks in all the files and take the complement as the new free list. In UNIX this can be done by scanning all the i-nodes. In the FAT file system, the problem cannot occur because there is no free list. But even if there were, all that would have to be done to recover it is to scan the FAT looking for free entries.
27. Ollie's thesis may not be backed up as reliably as he might wish. A backup program may pass over a file that is currently open for writing, as the state of the data in such a file may be indeterminate.
28. They must keep track of the time of the last dump in a file on disk. At every dump, an entry is appended to this file. At dump time, the file is read and the time of the last entry noted. Any file changed since that time is dumped.
29. In (a) and (b), 21 would not be marked. In (c), there would be no change. In (d), 21 would not be marked.
30. Many UNIX files are short. If the entire file fits in the same block as the i-node, only one disk access would be needed to read the file, instead of two, as is presently the case. Even for longer files there would be a gain, since one fewer disk accesses would be needed.
31. It should not happen, but due to a bug somewhere it could happen. It means that some block occurs in two files and also twice in the free list. The first step in repairing the error is to remove both copies from the free list. Next a free block has to be acquired and the contents of the sick block copied there. Finally, the occurrence of the block in one of the files should be changed to refer to the newly acquired copy of the block. At this point the system is once again consistent.
32. The time needed is  $h + 40 \times (1 - h)$ . The plot is just a straight line.
33. In this case, it is better to use a write-through cache since it writes data to the hard drive while also updating the cache. This will ensure that the updated file is always on the external hard drive even if the user accidentally removes the hard drive before disk sync is completed.
34. The block read-ahead technique reads blocks sequentially, ahead of their use, in order to improve performance. In this application, the records will likely not be accessed sequentially since the user can input any student ID at a given instant. Thus, the read-ahead technique will not be very useful in this scenario.
35. The blocks allotted to f1 are: 22, 19, 15, 17, 21.  
The blocks allotted to f2 are: 16, 23, 14, 18, 20.

36. At 15,000 rpm, the disk takes 4 msec to go around once. The average access time (in msec) to read  $k$  bytes is then  $6 + 2 + (k/1,048,576) \times 4$ . For blocks of 1 KB, 2 KB, and 4 KB, the access times are about 6.0039 msec, 6.0078 msec, and 6.0156 msec, respectively (hardly any different). These give rates of about 170.556 KB/sec, 340.890 KB/sec, and 680.896 KB/sec, respectively.
37. If all files were 1 KB, then each 4-KB block would contain one file and 3 KB of wasted space. Trying to put two files in a block is not allowed because the unit used to keep track of data is the block, not the semiblock. This leads to 75% wasted space. In practice, every file system has large files as well as many small ones, and these files use the disk much more efficiently. For example, a 32,769-byte file would use 9 disk blocks for storage, given a space efficiency of  $32,769/36,864$ , which is about 89%.
38. The indirect block can hold 1024 addresses. Added to the 10 direct addresses, there are 1034 addresses in all. Since each one points to a 4-KB disk block, the largest file is 4,235,264 bytes.
39. It constrains the sum of all the file lengths to being no larger than the disk. This is not a very serious constraint. If the files were collectively larger than the disk, there would be no place to store all of them on the disk.
40. The i-node holds 10 pointers. The single indirect block holds 1024 pointers. The double indirect block is good for  $1024^2$  pointers. The triple indirect block is good for  $1024^3$  pointers. Adding these up, we get a maximum file size of 1,074,791,434 blocks, which is about 16.06 GB.
41. The following disk reads are needed:
- directory for /
  - i-node for */usr*
  - directory for */usr*
  - i-node for */usr/ast*
  - directory for */usr/ast*
  - i-node for */usr/ast/courses*
  - directory for */usr/ast/courses*
  - i-node for */usr/ast/courses/os*
  - directory for */usr/ast/courses/os*
  - i-node for */usr/ast/courses/os/handout.t*
- In total, 10 disk reads are required.
42. Some pros are as follows. First, no disk space is wasted on unused i-nodes. Second, it is not possible to run out of i-nodes. Third, less disk movement is needed since the i-node and the initial data can be read in one operation. Some cons are as follows. First, directory entries will now need a 32-bit disk address instead of a 16-bit i-node number. Second, an entire disk will be used even for

files which contain no data (empty files, device files). Third, file-system integrity checks will be slower because of the need to read an entire block for each i-node and because i-nodes will be scattered all over the disk. Fourth, files whose size has been carefully designed to fit the block size will no longer fit the block size due to the i-node, messing up performance.

**SOLUTIONS TO CHAPTER 5 PROBLEMS**

1. In the figure, we see controllers and devices as separate units. The reason is to allow a controller to handle multiple devices, and thus eliminate the need for having a controller per device. If controllers become almost free, then it will be simpler just to build the controller into the device itself. This design will also allow multiple transfers in parallel and thus give better performance.
2. Easy. The scanner puts out 400 KB/sec maximum. The wireless network runs at 6.75 MB/sec, so there is no problem at all.
3. It is not a good idea. The memory bus is surely faster than the I/O bus, otherwise why bother with it? Consider what happens with a normal memory request. The memory bus finishes first, but the I/O bus is still busy. If the CPU waits until the I/O bus finishes, it has reduced memory performance to that of the I/O bus. If it just tries the memory bus for the second reference, it will fail if this one is an I/O device reference. If there were some way to instantaneously abort the previous I/O bus reference to try the second one, the improvement might work, but there is never such an option. All in all, it is a bad idea.
4. An advantage of precise interrupts is simplicity of code in the operating system since the machine state is well defined. On the other hand, in imprecise interrupts, OS writers have to figure out what instructions have been partially executed and up to what point. However, precise interrupts increase complexity of chip design and chip area, which may result in slower CPU.
5. Each bus transaction has a request and a response, each taking 50 nsec, or 100 nsec per bus transaction. This gives 10 million bus transactions/sec. If each one is good for 4 bytes, the bus has to handle 40 MB/sec. The fact that these transactions may be sprayed over five I/O devices in round-robin fashion is irrelevant. A bus transaction takes 100 nsec, regardless of whether consecutive requests are to the same device or different devices, so the number of channels the DMA controller has does not matter. The bus does not know or care.
6. (a) Word-at-a-time mode:  $1000 \times [(t_1 + t_2) + (t_1 + t_2) + (t_1 + t_2)]$   
Where the first term is for acquiring the bus and sending the command to the disk controller, the second term is for transferring the word, and the third term is for the acknowledgement. All in all, a total of  $3000 \times (t_1 + t_2)$  nsec.  
  
(b) Burst mode:  $(t_1 + t_2) + t_1 + 1000 \text{ times } t_2 + (t_1 + t_2)$   
where the first term is for acquiring the bus and sending the command to the disk controller, the second term is for the disk controller to acquire the bus, the third term is for the burst transfer, and the fourth term is for acquiring the bus and doing the acknowledgement. All in all, a total of  $3t_1 + 1002t_2$ .

7. Memory to memory copy can be performed by first issuing a read command that will transfer the word from memory to DMA controller and then issuing a write to memory to transfer the word from the DMA controller to a different address in memory. This method has the advantage that the CPU can do other useful work in parallel. The disadvantage is that this memory to memory copy is likely to be slow since DMA controller is much slower than CPU and the data transfer takes place over system bus as opposed to the dedicated CPU-memory bus.
8. An interrupt requires pushing 34 words onto the stack. Returning from the interrupt requires fetching 34 words from the stack. This overhead alone is 340 nsec. Thus the maximum number of interrupts per second is no more than about 2.94 million, assuming no work for each interrupt.
9. The execution rate of a modern CPU is determined by the number of instructions that finish per second and has little to do with how long an instruction takes. If a CPU can finish 1 billion instructions/sec it is a 1000 MIPS machine, even if an instruction takes 30 nsec. Thus there is generally little attempt to make instructions finish quickly. Holding the interrupt until the last instruction currently executing finishes may increase the latency of interrupts appreciably. Furthermore, some administration is required to get this right.
10. It could have been done at the start. A reason for doing it at the end is that the code of the interrupt service procedure is very short. By first outputting another character and then acknowledging the interrupt, if another interrupt happens immediately, the printer will be working during the interrupt, making it print slightly faster. A disadvantage of this approach is slightly longer dead time when other interrupts may be disabled.
11. Yes. The stacked PC points to the first instruction not fetched. All instructions before that have been executed and the instruction pointed to and its successors have not been executed. This is the condition for precise interrupts. Precise interrupts are not hard to achieve on machine with a single pipeline. The trouble comes in when instructions are executed out of order, which is not the case here.
12. The printer prints  $50 \times 80 \times 6 = 24,000$  characters/min, which is 400 characters/sec. Each character uses 50  $\mu$ sec of CPU time for the interrupt, so collectively in each second the interrupt overhead is 20 msec. Using interrupt-driven I/O, the remaining 980 msec of time is available for other work. In other words, the interrupt overhead costs only 2% of the CPU, which will hardly affect the running program at all.

13. UNIX does it as follows. There is a table indexed by device number, with each table entry being a C struct containing pointers to the functions for opening, closing, reading, and writing and a few other things from the device. To install a new device, a new entry has to be made in this table and the pointers filled in, often to the newly loaded device driver.
14. (a) Device driver.  
(b) Device driver.  
(c) Device-independent software.  
(d) User-level software.
15. A packet must be copied four times during this process, which takes 4.1 msec. There are also two interrupts, which account for 2 msec. Finally, the transmission time is 0.83 msec, for a total of 6.93 msec per 1024 bytes. The maximum data rate is thus 147,763 bytes/sec, or about 12% of the nominal 10 megabit/sec network capacity. (If we include protocol overhead, the figures get even worse.)
16. If the printer were assigned as soon as the output appeared, a process could tie up the printer by printing a few characters and then going to sleep for a week.
17. The disk rotates at 120 RPS, so 1 rotation takes  $1000/120$  msec. With 200 sectors per rotation, the sector time is  $1/200$  of this number or  $5/120 = 1/24$  msec. During the 1-msec seek, 24 sectors pass under the head. Thus the cylinder skew should be 24.
18. At 7200 RPM, there are 120 rotations per second, so 1 rotation takes about 8.33 msec. Dividing this by 500 we get a sector time of about  $16.67 \mu\text{sec}$ .
19. There are 120 rotations in a second. During one of them,  $500 \times 512$  bytes pass under the head. So the disk can read 256,000 bytes per rotation or 30,720,000 bytes/sec.
20. RAID level 2 can not only recover from crashed drives, but also from undetected transient errors. If one drive delivers a single bad bit, RAID level 2 will correct this, but RAID level 3 will not.
21. The probability of 0 failures,  $P_0$ , is  $(1 - p)^k$ . The probability of 1 failure,  $P_1$ , is  $kp(1 - p)^{k-1}$ . The probability of a RAID failure is then  $1 - P_0 - P_1$ . This is  $1 - (1 - p)^k - kp(1 - p)^{k-1}$ .
22. Read performance: RAID levels 0, 2, 3, 4, and 5 allow for parallel reads to service one read request. However, RAID level 1 further allows two read requests to simultaneously proceed. Write performance: All RAID levels provide similar write performance. Space overhead: There is no space overhead in level 0 and 100% overhead in level 1. With 32-bit data word and six parity drives, the space overhead is about 18.75% in level 2. For a 32-bit data word, the space overhead in level 3 is about 3.13%. Finally, assuming 33 drives in levels 4 and



5, the space overhead is 3.13% in them. Reliability: There is no reliability support in level 0. All other RAID levels can survive one disk crash. In addition, in levels 3, 4 and 5, a single random bit error in a word can be detected, while in level 2, a single random bit error in a word can be detected and corrected.

23. A zebibyte is  $2^{70}$  bytes; a pebibyte is  $2^{50}$ .  $2^{20}$  pebibytes fit in a zebibyte.
24. A magnetic field is generated between two poles. Not only is it difficult to make the source of a magnetic field small, but also the field spreads rapidly, which leads to mechanical problems trying to keep the surface of a magnetic medium close to a magnetic source or sensor. A semiconductor laser generates light in a very small place, and the light can be optically manipulated to illuminate a very small spot at a relatively great distance from the source.
25. The main advantage of optical disks is that they have much higher recording densities than magnetic disks. The main advantage of magnetic disks is that they are an order of magnitude faster than the optical disks.
26. Possibly. If most files are stored in logically consecutive sectors, it might be worthwhile interleaving the sectors to give programs time to process the data just received, so that when the next request is issued, the disk would be in the right place. Whether this is worth the trouble depends strongly on the kind of programs run and how uniform their behavior is.
27. Maybe yes and maybe no. Double interleaving is effectively a cylinder skew of two sectors. If the head can make a track-to-track seek in fewer than two sector times, then no additional cylinder skew is needed. If it cannot, then additional cylinder skew is needed to avoid missing a sector after a seek.
28. Consider,
  - (a) The capacity of a zone is  $\text{tracks} \times \text{cylinders} \times \text{sectors/cylinder} \times \text{bytes/sect}$ .  
 Capacity of zone 1:  $16 \times 100 \times 160 \times 512 = 131072000$  bytes  
 Capacity of zone 2:  $16 \times 100 \times 200 \times 512 = 163840000$  bytes  
 Capacity of zone 3:  $16 \times 100 \times 240 \times 512 = 196608000$  bytes  
 Capacity of zone 4:  $16 \times 100 \times 280 \times 512 = 229376000$  bytes  
 Sum =  $131072000 + 163840000 + 196608000 + 229376000 = 720896000$
  - (b) A rotation rate of 7200 means there are 120 rotations/sec. In the 1 msec track-to-track seek time, 0.120 of the sectors are covered. In zone 1, the disk head will pass over  $0.120 \times 160$  sectors in 1 msec, so, optimal track skew for zone 1 is 19.2 sectors. In zone 2, the disk head will pass over  $0.120 \times 200$  sectors in 1 msec, so, optimal track skew for zone 2 is 24 sectors. In zone 3, the disk head will pass over  $0.120 \times 240$  sectors in 1 msec, so, optimal track skew for zone 3 is 28.8 sectors. In zone 4, the disk head will pass over  $0.120 \times 280$  sectors in 1 msec, so, optimal track skew for zone 4 is 33.6 sectors.

- (c) The maximum data transfer rate will be when the cylinders in the outermost zone (zone 4) are being read/written. In that zone, in one second, 280 sectors are read 120 times. Thus the data rate is  $280 \times 120 \times 512 = 17,203,200$  bytes/sec.
29. The drive capacity and transfer rates are doubled. The seek time and average rotational delay are the same. No properties are worse.
30. One fairly obvious consequence is that no existing operating system will work because they all look there to see where the disk partitions are. Changing the format of the partition table will cause all the operating systems to fail. The only way to change the partition table is to simultaneously change all the operating systems to use the new format.
31. (a)  $10 + 12 + 2 + 18 + 38 + 34 + 32 = 146$  cylinders = 876 msec.  
(b)  $0 + 2 + 12 + 4 + 4 + 36 + 2 = 60$  cylinders = 360 msec.  
(c)  $0 + 2 + 16 + 2 + 30 + 4 + 4 = 58$  cylinders = 348 msec.
32. In the worst case, a read/write request is not serviced for almost two full disk scans in the elevator algorithm, while it is at most one full disk scan in the modified algorithm.
33. A disadvantage of one-shot mode is that the time consumed by interrupt handler is unaccounted for as the process of decrementing the counter is paused during this time. A Disadvantage of square-wave mode is that high clock frequencies may result in multiple interrupts being queued when new interrupts are raised before the previous ones have been serviced.
34. Not necessarily. A UNIX program that reads 10,000 blocks issues the requests one at a time, blocking after each one is issued until after it is completed. Thus the disk driver sees only one request at a time; it has no opportunity to do anything but process them in the order of arrival. Harry should have started up many processes at the same time to see if the elevator algorithm worked.
35. There is a race but it does not matter. Since the stable write itself has already completed, the fact that the nonvolatile RAM has not been updated just means that the recovery program will know which block was being written. It will read both copies. Finding them identical, it will change neither, which is the correct action. The effect of the crash just before the nonvolatile RAM was updated just means the recovery program will have to make two disk reads more than it should.
36. Yes the disk remains consistent even if the CPU crashes during a recovery procedure. Consider Fig. 5-31. There is no recovery involved in (a) or (e). Suppose that the CPU crashes during recovery in (b). If CPU crashes before the block from drive 2 has been completely copied to drive 1, the situation remains same as earlier. The subsequent recovery procedure will detect an

ECC error in drive 1 and again copy the block from drive 2 to drive 1. If CPU crashes after the block from drive 2 has been copied to drive 1, the situation is same as that in case (e). Suppose that the CPU crashes during recovery in (c). If CPU crashes before the block from drive 1 has been completely copied to drive 2, the situation is same as that in case (d). The subsequent recovery procedure will detect an ECC error in drive 2 and copy the block from drive 1 to drive 2. If CPU crashes after the block from drive 1 has been copied to drive 2, the situation is same as that in case (e). Finally, suppose the CPU crashes during recovery in (d). If CPU crashes before the block from drive 1 has been completely copied to drive 2, the situation remains same as earlier. The subsequent recovery procedure will detect an ECC error in drive 2 and again copy the block from drive 1 to drive 2. If CPU crashes after the block from drive 1 has been copied to drive 2, the situation is same as that in case (e).

- 37.** Problems arise in scenarios shown in Figure 5-27 (b) and 5-27 (d), because they may look like scenario 5-27 (c), if the ECC of the corrupted block is correct. In this case, it is not possible to detect which disk contains the valid (old or new) lock, and a recovery is not possible.
- 38.** Two msec 60 times a second is 120 msec/sec, or 12% of the CPU
- 39.** With these parameters,
- (a) Using a 500 MHz crystal, the counter can be decremented every 2 nsec. So, for a tick every millisecond, the register should be  $1000000/2 = 500,000$ .
  - (b) To get a clock tick every 100  $\mu$ sec, holding register value should be 50,000.
- 40.** At time 5000:  
Current time = 5000; Next Signal = 8; Header  $\rightarrow 8 \rightarrow 4 \rightarrow 3 \rightarrow 14 \rightarrow 8$ .
- At time 5005:  
Current time = 5005; Next Signal = 3; Header  $\rightarrow 3 \rightarrow 4 \rightarrow 3 \rightarrow 14 \rightarrow 8$ .
- At time 5013:  
Current time = 5013; Next Signal = 2; Header  $2 \rightarrow 14 \rightarrow 8$ .
- At time 5023:  
Current time = 5023; Next Signal = 6; Header  $\rightarrow 6 \rightarrow 4 \rightarrow 5$ .
- 41.** The number of seconds in a mean year is  $365.25 \times 24 \times 3600$ . This number is 31,557,600. The counter wraps around after  $2^{32}$  seconds from 1 January 1970. The value of  $2^{32}/31,557,600$  is 136.1 years, so wrapping will happen at 2106.1, which is early February 2106. Of course, by then, all computers will be at least 64 bits, so it will not happen at all.
- 42.** Scrolling the window requires copying 79 lines of 80 characters or 6320 characters. Copying 1 character (16 bytes) takes 800 nsec, so the whole window takes 5.056 msec. Writing 80 characters to the screen takes 400 nsec, so

scrolling and displaying a new line take 5.456 msec. This gives about 183.2 lines/sec.

43. Suppose that the user inadvertently asked the editor to print thousands of lines. Then he hits DEL to stop it. If the driver did not discard output, output might continue for several seconds after the DEL, which would make the user hit DEL again and again and get frustrated when nothing happened.
44. It should move the cursor to line 5 position 7 and then delete 6 characters. The sequence is ESC [ 5 ; 7 H ESC [ 6 P
45. The maximum rate the mouse can move is 200 mm/sec, which is 2000 microns/sec. If each report is 3 byte, the output rate is 6000 bytes/sec.
46. With a 24-bit color system, only  $2^{24}$  colors can be represented. This is not all of them. For example, suppose that a photographer takes pictures of 300 cans of pure blue paint, each with a slightly different amount of pigment. The first might be represented by the (R, G, B) value (0, 0, 1). The next one might be represented by (0, 0, 2), and so forth. Since the B coordinate is only 8 bits, there is no way to represent 300 different values of pure blue. Some of the photographs will have to be rendered as the wrong color. Another example is the color (120.24, 150.47, 135.89). It cannot be represented, only approximated by (120, 150, 136).
47.
  - (a) Each pixel takes 3 bytes in RGB, so the table space is  $16 \times 24 \times 3$  bytes, which is 1152 bytes.
  - (b) At 100 nsec per byte, each character takes 115.2  $\mu$ sec. This gives an output rate of about 8681 chars/sec.
48. Rewriting the text screen requires copying 2000 bytes, which can be done in 4  $\mu$ seconds. Rewriting the graphics screen requires copying  $1024 \times 768 \times 3 = 2,359,296$  bytes, or about 4.72 msec.
49. In Windows, the OS calls the handler procedures itself. In X Windows, nothing like this happens. X just gets a message and processes it internally.
50. The first parameter is essential. First of all, the coordinates are relative to some window, so *hdc* is needed to specify the window and thus the origin. Second, the rectangle will be clipped if it falls outside the window, so the window coordinates are needed. Third, the color and other properties of the rectangle are taken from the context specified by *hdc*. It is quite essential.
51. The display size is  $400 \times 160 \times 3$  bytes, which is 192,000 bytes. At 10 fps this is 1,920,000 bytes/sec or 15,360,000 bits/sec. This consumes 15% of the Fast Ethernet.

- 52. The bandwidth on a network segment is shared, so 100 users requesting different data simultaneously on a 1-Mbps network will each see a 10-Kbps effective speed. With a shared network, a TV program can be multicast, so the video packets are only broadcast once, no matter how many users there are and it should work well. With 100 users browsing the Web, each user will get 1/100 of the bandwidth, so performance may degrade very quickly.
- 53. Advantages of thin clients include low cost and no need for complex management for the clients. Disadvantages include (potentially) lower performance due to network latency and (potential) loss of privacy because the client's data/information is shared with the server.
- 54. If  $n = 10$ , the CPU can still get its work done on time, but the energy used drops appreciably. If the energy consumed in 1 sec at full speed is  $E$ , then running at full speed for 100 msec then going idle for 900 msec uses  $E/10$ . Running at 1/10 speed for a whole second uses  $E/100$ , a saving of  $9E/100$ . The percent savings by cutting the voltage is 90%.
- 55. The windowing system uses much more memory for its display and uses virtual memory more than the text mode. This makes it less likely that the hard disk will be inactive for a period long enough to cause it to be automatically powered down.

**SOLUTIONS TO CHAPTER 6 PROBLEMS**

1. In the U.S., consider a presidential election in which three or more candidates are trying for the nomination of some party. After all the primary elections are finished, when the delegates arrive at the party convention, it could happen that no candidate has a majority and that no delegate is willing to change his or her vote. This is a deadlock. Each candidate has some resources (votes) but needs more to get the job done. In countries with multiple political parties in the parliament, it could happen that each party supports a different version of the annual budget and that it is impossible to assemble a majority to pass the budget. This is also a deadlock.
2. Disk space on the spooling partition is a finite resource. Every block that comes in de facto claims a resource and every new one arriving wants more resources. If the spooling space is, say, 10 MB and the first half of ten 2-MB jobs arrive, the disk will be full and no more blocks can be stored so we have a deadlock. The deadlock can be avoided by allowing a job to start printing before it is fully spooled and reserving the space thus released for the rest of that job. In this way, one job will actually print to completion, then the next one can do the same thing. If jobs cannot start printing until they are fully spooled, deadlock is possible.
3. The printer is nonpreemptable; the system cannot start printing another job until the previous one is complete. The spool disk is preemptable; you can delete an incomplete file that is growing too large and have the user send it later, assuming the protocol allows that.
4. Yes. It does not make any difference whatsoever.
5. Suppose that there are three processes, *A*, *B* and *C*, and two resource types, *R* and *S*. Further assume that there are one instance of *R* and two instance of *S*. Consider the following execution scenario: *A* requests *R* and gets it; *B* requests *S* and gets; *C* requests *S* and gets it (there are two instances of *S*); *B* requests *R* and is blocked; *A* requests *S* and is blocked. At this stage all four conditions hold. However, there is no deadlock. When *C* finishes, one instance of *S* is released that is allocated to *A*. Now *A* can complete its execution and release *R* that can be allocated to *B*, which can then complete its execution. These four conditions are enough if there is one resource of each type.
6. “Don’t block the box” is a pre-allocation strategy, negating the hold and wait deadlock precondition, since we assume that cars can enter the street space following the intersection, thus freeing the intersection. Another strategy might allow cars to temporarily pull into garages and release enough space to clear the gridlock. Some cities have a traffic control policy to shape traffic; as city streets become more congested, traffic supervisors adjust the settings for red lights in order to throttle traffic entering heavily congested areas. Lighter traf-

fic ensures less competition over resources and thus lowers the probability of gridlock occurring.

7. The above anomaly is not a communication deadlock since these cars are independent of each other and would drive through the intersection with a minimal delay if no competition occurred. It is not a resource deadlock, since no car is holding a resource that is requested by another car. Nor would the mechanisms of resource pre-allocation or of resource preemption assist in controlling this anomaly. This anomaly is one of competition synchronization, however, in which cars are waiting for resources in a circular chain and traffic throttling may be an effective strategy for control. To distinguish from resource deadlock, this anomaly might be termed a “scheduling deadlock.” A similar deadlock could occur following a law that required two trains merging onto a shared railroad track to wait for the other to proceed. Note that a policeman signaling one of the competing cars or trains to proceed (and not the others) can break this dead state without rollback or any other overhead.
8. It is possible that one process holds some or all of the units of one resource type and requests another resource type, while another process holds the second resource while requesting the available units of the first resource type. If no other process can release units of the first resource type and the resource cannot be preempted or used concurrently, the system is deadlocked. For example, two processes are both allocated memory cells in a real memory system. (We assume that swapping of pages or processes is not supported, while dynamic requests for memory are supported.) The first process locks another resource - perhaps a data cell. The second process requests the locked data and is blocked. The first process needs more memory in order to execute the code to release the data. Assuming that no other processes in the system can complete and release memory cells, a deadlock exists in the system.
9. Yes, illegal graphs exist. We stated that a resource may only be held by a single process. An arc from a resource square to a process circle indicates that the process owns the resource. Thus, a square with arcs going from it to two or more processes means that all those processes hold the resource, which violates the rules. Consequently, any graph in which multiple arcs leave a square and end in different circles violates the rules unless there are multiple copies of the resources. Arcs from squares to squares or from circles to circles also violate the rules.
10. Neither change leads to deadlock. There is no circular wait in either case.
11. Consider three processes,  $A$ ,  $B$  and  $C$  and two resources  $R$  and  $S$ . Suppose  $A$  is waiting for  $I$  that is held by  $B$ ,  $B$  is waiting for  $S$  held by  $A$ , and  $C$  is waiting for  $R$  held by  $A$ . All three processes,  $A$ ,  $B$  and  $C$  are deadlocked. However, only  $A$  and  $B$  belong to the circular chain.

12. This is clearly a communication deadlock, and can be controlled by having *A* time out and retransmit its enabling message (the one that increases the window size) after some period of time (a heuristic). It is possible, however, that *B* has received both the original and the duplicate message. No harm will occur if the update on the window size is given as an absolute value and not as a differential. Sequence numbers on such messages are also effective to detect duplicates.
13. A portion of all such resources could be reserved for use only by processes owned by the administrator, so he or she could always run a shell and programs needed to evaluate a deadlock and make decisions about which processes to kill to make the system usable again.
14. First, the set of unmarked processes,  $P = (P1 \ P2 \ P3 \ P4)$   
 $R1$  is not less than or equal to  $A$   
 $R2$  is less than  $A$ ; Mark  $P2$ ;  $A = (0 \ 2 \ 0 \ 3 \ 1)$ ;  $P = (P1 \ P3 \ P4)$   
 $R1$  is not less than or equal to  $A$   
 $R3$  is equal to  $A$ ; Mark  $P3$ ;  $A = (0 \ 2 \ 0 \ 3 \ 2)$ ;  $P = (P1 \ P4)$   
 $R1$  is not less than or equal to  $A$   
 $R4$  is not less than or equal to  $A$   

So, processes  $P1$  and  $P4$  remain unmarked. They are deadlocked.
15. Recovery through preemption: After processes  $P2$  and  $P3$  complete, process  $P1$  can be forced to preempt 1 unit of  $RS3$ . This will make  $A = (0 \ 2 \ 1 \ 3 \ 2)$ , and allow process  $P4$  to complete. Once  $P4$  completes and release its resources  $P1$  may complete. Recovery through rollback: Rollback  $P1$  to the state checkpointed before it acquired  $RS3$ . Recovery through killing processes: Kill  $P1$ .
16. The process is asking for more resources than the system has. There is no conceivable way it can get these resources, so it can never finish, even if no other processes want any resources at all.
17. If the system had two or more CPUs, two or more processes could run in parallel, leading to diagonal trajectories.
18. Yes. Do the whole thing in three dimensions. The  $z$ -axis measures the number of instructions executed by the third process.
19. The method can only be used to guide the scheduling if the exact instant at which a resource is going to be claimed is known in advance. In practice, this is rarely the case.
20. There are states that are neither safe nor deadlocked, but which lead to deadlocked states. As an example, suppose we have four resources: tapes, plotters, scanners, and CD-ROMs, as in the text, and three processes competing for them. We could have the following situation:



|    | 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 |           |

This state is not deadlocked because many actions can still occur, for example, A can still get two printers. However, if each process asks for its remaining requirements, we have a deadlock.

21. A request from *D* is unsafe, but one from *C* is safe.
22. The system is deadlock free. Suppose that each process has one resource. There is one resource free. Either process can ask for it and get it, in which case it can finish and release both resources. Consequently, deadlock is impossible.
23. If a process has  $m$  resources it can finish and cannot be involved in a deadlock. Therefore, the worst case is where every process has  $m - 1$  resources and needs another one. If there is one resource left over, one process can finish and release all its resources, letting the rest finish too. Therefore the condition for avoiding deadlock is  $r \geq p(m - 1) + 1$ .
24. No. *D* can still finish. When it finishes, it returns enough resources to allow *E* (or *A*) to finish, and so on.
25. Comparing a row in the matrix to the vector of available resources takes  $m$  operations. This step must be repeated on the order of  $n$  times to find a process that can finish and be marked as done. Thus, marking a process as done takes on the order of  $mn$  steps. Repeating the algorithm for all  $n$  processes means that the number of steps is then  $mn^2$ . Thus,  $a = 1$  and  $b = 2$ .
26. The needs matrix is as follows:

```

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

```

If  $x$  is 0, we have a deadlock immediately. If  $x$  is 1, process *D* can run to completion. When it is finished, the available vector is 1 1 2 2 1. Unfortunately we are now deadlocked. If  $x$  is 2, after *D* runs, the available vector is 1 1 3 2 1 and *C* can run. After it finishes and returns its resources the available vector is 2 2 3 3 1, which will allow *B* to run and complete, and then *A* to run and complete. Therefore, the smallest value of  $x$  that avoids a deadlock is 2.

27. Consider a process that needs to copy a huge file from a tape to a printer. Because the amount of memory is limited and the entire file cannot fit in this memory, the process will have to loop through the following statements until the entire file has been printed:

Acquire tape drive  
Copy the next portion of the file in memory (limited memory size)  
Release tape drive  
Acquire printer  
Print file from memory  
Release printer

This will lengthen the execution time of the process. Furthermore, since the printer is released after every print step, there is no guarantee that all portions of the file will get printed on continuous pages.

28. Suppose that process *A* requests the records in the order *a, b, c*. If process *B* also asks for *a* first, one of them will get it and the other will block. This situation is always deadlock free since the winner can now run to completion without interference. Of the four other combinations, some may lead to deadlock and some are deadlock free. The six cases are as follows:

|              |                   |
|--------------|-------------------|
| <i>a b c</i> | deadlock free     |
| <i>a c b</i> | deadlock free     |
| <i>b a c</i> | possible deadlock |
| <i>b c a</i> | possible deadlock |
| <i>c a b</i> | possible deadlock |
| <i>c b a</i> | possible deadlock |

Since four of the six may lead to deadlock, there is a 1/3 chance of avoiding a deadlock and a 2/3 chance of getting one.

29. Yes. Suppose that all the mailboxes are empty. Now *A* sends to *B* and waits for a reply, *B* sends to *C* and waits for a reply, and *C* sends to *A* and waits for a reply. All the conditions for a communications deadlock are now fulfilled.
30. To avoid circular wait, number the resources (the accounts) with their account numbers. After reading an input line, a process locks the lower-numbered account first, then when it gets the lock (which may entail waiting), it locks the other one. Since no process ever waits for an account lower than what it already has, there is never a circular wait, hence never a deadlock.
31. Change the semantics of requesting a new resource as follows. If a process asks for a new resource and it is available, it gets the resource and keeps what it already has. If the new resource is not available, all existing resources are released. With this scenario, deadlock is impossible and there is no danger that the new resource is acquired but existing ones lost. Of course, the process

only works if releasing a resource is possible (you can release a scanner between pages or a CD recorder between CDs).

32. I'd give it an F (failing) grade. What does the process do? Since it clearly needs the resource, it just asks again and blocks again. This is no better than staying blocked. In fact, it may be worse since the system may keep track of how long competing processes have been waiting and assign a newly freed resource to the process that has been waiting longest. By periodically timing out and trying again, a process loses its seniority.
33. Both virtual memory and time-sharing systems were developed mainly to assist system users. Virtualizing hardware shields users from the details of prestatting needs, resource allocation, and overlays, in addition to preventing deadlock. The cost of context switching and interrupt handling, however, is considerable. Specialized registers, caches, and circuitry are required. Probably this cost would not have been incurred for the purpose of deadlock prevention alone.
34. A deadlock occurs when a set of processes are blocked waiting for an event that only some other process in the set can cause. On the other hand, processes in a livelock are not blocked. Instead, they continue to execute checking for a condition to become true that will never become true. Thus, in addition to the resources they are holding, processes in livelock continue to consume precious CPU time. Finally, starvation of a process occurs because of the presence of other processes as well as a stream of new incoming processes that end up with higher priority than the process being starved. Unlike deadlock or livelock, starvation can terminate on its own, e.g. when existing processes with higher priority terminate and no new processes with higher priority arrive.
35. This dead state is an anomaly of competition synchronization and can be controlled by resource pre-allocation. Processes, however, are not blocked from resources. In addition, resources are already requested in a linear order. This anomaly is not a resource deadlock; it is a livelock. Resource preallocation will prevent this anomaly. As a heuristic, processes may time-out and release their resources if they do not complete within some interval of time, then go to sleep for a random period and then try again.
36. Here are the answers, albeit a bit complicated.
  - (a) This is a competition synchronization anomaly. It is also a livelock. We might term it a scheduling livelock. It is not a resource livelock or deadlock, since stations are not holding resources that are requested by others and thus a circular chain of stations holding resources while requesting others does not exist. It is not a communication deadlock, since stations are executing independently and would complete transmission were scheduled sequentially.

(b) Ethernet and slotted Aloha require that stations that detect a collision of their transmission must wait a random number of time slots before retransmitting. The interval within which the time slot is chosen is doubled after each successive collision, dynamically adjusting to heavy traffic loads. After sixteen successive retransmissions a frame is dropped.

(c) Because access to the channel is probabilistic, and because newly arriving stations can compete and be allocated the channel before stations that have retransmitted some number of times, starvation is enabled.

37. The anomaly is not a resource deadlock. Although processes are sharing a mutex, i.e., a competition mechanism, resource pre-allocation and deadlock avoidance methods are all ineffective for this dead state. Linearly ordering resources is also ineffective. Indeed one could argue that linear orders may be the problem; executing a mutex should be the last step before entering and the first after leaving a critical section. A circular dead state does exist in which both processes wait for an event that can only be caused by the other process. This is a communication deadlock. To make progress, a time-out will work to break this deadlock if it preempts the consumer's mutex. Writing careful code or using monitors for mutual exclusion are better solutions.
38. If both programs ask for Woofers first, the computers will starve with the endless sequence: request Woofers, cancel request, request Woofers, cancel request, and so on. If one of them asks for the doghouse and the other asks for the dog, we have a deadlock, which is detected by both parties and then broken, but it is just repeated on the next cycle. Either way, if both computers have been programmed to go after the dog or the doghouse first, either starvation or deadlock ensues. There is not really much difference between the two here. In most deadlock problems, starvation does not seem serious because introducing random delays will usually make it very unlikely. That approach does not work here.

**SOLUTIONS TO CHAPTER 7 PROBLEMS**

1. There are numerous reasons, among them consolidating servers to save hardware investment cost, rack space, and electrical power, and make management of thousands of servers easier.
2. If the hardware configuration was upgraded, virtualization could hide this and allow old software to continue working.
3. There are various reasons. A key one is to have many platforms such as Windows 7, Windows 8, Linux, FreeBSD, OS X, etc. available on a single desktop machine to test the software being developed. Also, rebooting a virtual machine after a crash induced by a software bug is much faster.
4. After upgrading to a new computer and operating system, the person might want to run some software that he had on the old one. Virtualization makes it possible to run the old system and new one on the same computer, thus preserving the old software.
5. Very few programmers had access to an IBM mainframe. Starting on the 1980s, the Intel x86 series dominated computing and it was not virtualizable. While binary translation could solve that problem, that idea was not thought of until the late 1990s.
6. Any instruction changing the page tables or memory map is certainly sensitive, as well as anything involving I/O. Any instruction capable of reading the true state of the machine is also sensitive.
7. There are many, including moves, arithmetic instructions, jump and call instructions, shifts, etc.
8. Full virtualization means emulating the hardware exactly so every operating system running on the virtual machine behaves exactly as it would on the bare metal. Paravirtualization consists of changing the operating system so it does not do anything that is hard to virtualize. Full virtualization in the absence of hardware support is complicated on any architecture that is complex, like the x86. It is easier on RISC machines. If virtualization hardware is present, full virtualization is not so difficult. So, which is harder probably depends on whether hardware support is available. If it is, then paravirtualizing an operating system is probably more work. If there is no hardware support, it may be easier to change the operating system to be more friendly. If there are many operating systems that have to be paravirtualized, that could be more work.
9. Yes, of course. Linux has been paravirtualized precisely because the source code is available. Windows has been paravirtualized by Microsoft (which has the source code), but has not released any paravirtualized versions.

10. Virtual machines have nothing to do with disk partitions. The hypervisor can take a disk partition and divide it up into subpartitions and give each virtual machine one of them. In principle, there can be hundreds. It can either statically partition the disk into  $n$  pieces or do this on demand. In hosted virtual machines, it is common to use files on the host to store disk images of the guest.
11. An application or process is virtualized during runtime, by using a virtualization layer between the application and the OS. This layer executes the application's instructions, modifying them as required prior to execution. The application is transparent to the presence of the underlying layer. Windows Emulator (WINE) is an example, where Microsoft Windows binary executables can be executed on another operating system such as Linux. This is done using on-the-fly mapping of Windows API calls to POSIX calls.
12. Type 1 hypervisors generally require changing the boot procedure of the computer to load the hypervisor first, then create virtual machines, and then install operating systems in them. At data centers run by expert system administrators, this is not a problem, but for most ordinary users, doing this is far too complicated. Type 2 hypervisors were invented to make installing a hypervisor no more difficult than installing an application program, something that users frequently do. Also, by using a host operating system to service local peripherals, it was not necessary for the hypervisor to have drivers for all of them since it could use the ones inside the host OS.
13. Yes. When a guest OS does I/O, for example, the virtualization hardware catches it and gets control to the type 2 hypervisor, which then figures out what to do. Usually this will involve making a request to the host OS to perform the I/O, but not having to worry about trapping the I/O instruction definitely simplifies matters for the hypervisor.
14. It was invented in the early days, before virtualization hardware existed. It was necessary to prevent guest operating systems, which were running in user mode, from executing sensitive instructions that were not privileged. Going forward, this is less necessary since modern hardware traps when a user-mode program executes a sensitive instruction. However, in some circumstances, binary translation is faster than trapping. Nevertheless, as the hardware improves, the need for binary translation will decrease.
15. Typically, ring 0 (with the highest set of privileges) is used for running in kernel mode; and ring 3 for user mode. Some hypervisors used ring 0 for running the hypervisor in kernel mode; the guest OS was run in ring 1. When the guest OS invokes privileged instructions, it could trap to the hypervisor that runs these instructions after verifying the access rights, permissions, etc. Other ways are also possible.

16. It has been shown that the VT-enabled CPU approach results in a lot of traps due to the use of trap-and-emulate approach. Since traps are expensive to handle, there are instances where translation based approaches outperform the hardware based approach.
17. When the guest OS issues a “Clear Interrupt” instruction (such as CLI), doing it in hardware can be very time consuming in some CPUs such as those with deep pipelines. On the other hand, in a virtualized system, the hypervisor need not actually disable interrupts in hardware and simply use a variable to indicate that the specified guest OS’s “Interrupt Flag” is set to zero, making it faster to execute the CLI instruction.
18. It could translate the entire program in advance. The reason for not doing so is that many programs have large pieces of code that are never executed. By translating basic blocks on demand, no unused code is ever translated. A potential disadvantage of on-demand translation is that it might be slightly less efficient to keep starting and stopping the translator, but this effect is probably small. In addition, static analysis and translation of x86 code is complicated due to indirect branches (branches whose targets are computed at run time). This is made worse by the variable-size instructions on the x86. Thus you may not be sure which instructions to translate. Finally, there is the issue of self-modifying code.
19. A pure hypervisor just emulates the real hardware and nothing else. A pure microkernel is a small operating system that offers basic services to the programs using it. The virtual machines running on a pure hypervisor run traditional operating systems such as Windows and Linux. On top of a microkernel are generally processes that implement operating system services but in a decentralized way.
20. If multiple guest OSes all allocate what they think is physical page  $k$  to one of their processes, there is a problem. Some way is needed to perform a second mapping between pages because the guests do not really control the physical pages, despite what they may think. This is why nested page tables are needed.
21. Not only does the machine need memory for the normal (guest) operating system and all its applications, but it also needs memory for the hypervisor functions and data structures needed to execute sensitive instructions on behalf of the guest OS. Type 2 hypervisors have the added cost of the host operating system. Moreover, each virtual machine will have its own operating system, so there will be  $N$  operating system copies stored in memory. One way to reduce memory usage would be to identify “shared code” and keep only one copy of this code in memory. For example, a Web hosting company may run multiple VMs, each running an identical version of Linux and an identical copy of the Apache web server code. In this case, the code segment can be shared across VMs, even though the data regions must be private.

22. Each guest OS will maintain a page table that maps its virtual page numbers to physical frame numbers (on its share of the virtualized memory). In order to prevent different guest operating systems from incorrectly referring to the same physical page number, the hypervisor creates a shadow page table that maps the virtual machine's virtual page number to the physical frame number provided by the hypervisor.
23. Page tables can be modified only by the guest operating system, not the application programs in the guest. When the guest OS is finished modifying the tables, it must switch back to user mode by issuing a sensitive instruction like RETURN FROM TRAP. This instruction will trap and give the hypervisor control. It could then examine the page tables in the guest OS to see if they had been modified. While this could work, all the page tables would have to be checked on every system made by a guest application, that is, every time the guest OS returned to user mode. There could be thousands of these transitions per second, so it is not likely to be as efficient as using read-only pages for the page table.
24. When a hypervisor runs out of pages, it has no way of figuring out which pages the guest operating systems really value. The solution is to cheat and include balloon drivers in the guests. The hypervisor then signals the balloon drivers to expand their memory usage, forcing the guest operating systems to decide which pages to evict. This is definitely cheating because the hypervisor is not supposed to talk to specific pieces of the guest operating systems. It is not supposed to know what is going on in the virtual machines at all. But this technique solves a problem in a simple way, so everyone pretends there is nothing iffy going on.
25. Balloon drivers do not work if the hypervisor does not know anything about the guest operating systems running on its virtual machines. It also does not work if there is no way to include a balloon driver in them, for example, if they do not support loadable drivers and the source code is not available so they cannot be recompiled to include the balloon driver.
26. Consider a case where multiple virtual machines copies of the same guest OS reside in a system. In this case, it is not necessary to maintain multiple copies of the read-only portion of the OS (such as code segments) in memory. Only one copy needs to be maintained, thereby reducing memory requirements and allowing more virtual machines on a system. This technique is called deduplication. VMware calls this "transparent page sharing."
27. Yes. Early DMA hardware used absolute memory addresses. If a guest operating system started a DMA operating to what it thought was physical address  $k$ , this would probably not go to the buffer it was supposed to go to and might overwrite something important. Early hypervisors had to rewrite code that used DMA to use addresses that would not cause trouble.



28. Using cloud services means you do not have to set up and maintain a computing infrastructure. You may also be able to outsource making backups. Furthermore, if your computing needs change rapidly, you can add or remove machines easily. On the downside, the cloud provider could easily steal your confidential data, and the promised expandability might be illusory if you need extra capacity just at the moment Walmart or some other big customer decides to grab 10,000 machines. Also, the bandwidth between you and the cloud might be an issue. It is likely to be far less than the local bandwidth, so if a lot of data needs to move between you and the cloud, that could be an issue. Also, if you are doing real-time work, the bandwidth between you and the cloud could vary wildly from moment to moment, causing trouble.
29. Obviously there are many, but a provider offering empty virtual x86 machines would be offering IAAS. A provider offering Windows 8 or Linux machines would be offering PAAS. A provider offering a word-processing program, such as Microsoft Word, running in the cloud would be offering software as a service.
30. Suppose many virtual machines were started up on a single server. Initially, all of them did about the same amount of work and required the same resources and the situation was fine. Then all of a sudden, one of them began using massive resources (CPU, memory, etc.) disturbing all the other virtual machines. This might be a good time to migrate it to a dedicated server of its own.
31. Physical I/O devices still present problems because they do not migrate with the virtual machine, yet their registers may hold state that is critical to the proper functioning of the system. Think of read or write operations to devices (e.g., the disk) that have been issued but have not yet completed. Network I/O is particularly difficult because other machines will continue to send packets to the hypervisor, unaware that the virtual machine has moved. Even if packets can be redirected to the new hypervisor, the virtual machine will be unresponsive during the migration period, which can be long because the entire virtual machine, including the guest operating system and all processes executing on it, must be moved to the new machine. As a result packets can experience large delays or even packet loss if the device/hypervisor buffers overflow.
32. In order to migrate a specific process, process state information has to be stored and then transferred, including open files, alarms, signal handlers, etc. Errors may creep in during the state capture task leading to potentially incorrect, incomplete or inconsistent state information. In the case of VM migration, the entire memory and disk images are moved to the new system, which is easier.
33. Standard (dead) migration consists of stopping the virtual machine and saving its memory image as a file. The file is then transported to the destination, installed in a virtual machine, and restarted. Doing so causes the application to stop for a little while during transport. In many circumstances having the ap-

plication stop is undesirable. With live migration, the pages of the virtual machine are moved while it is running. After they all arrive at the destination, a check is made to see if any of them have changed since being migrated. If so, they are copied again. This process is repeated until all the pages at the destination are up to date. Working this way (live migration) means applications can be moved with no downtime.

34. The three main requirements were: Compability (ability to run an existing guest OS without any modifications as a virtual machine); Performance (minimal overhead during VM execution; else, users would not choose to run their applications inside a VM) and Isolation (protecting the hardware resources from malicious or otherwise unauthorized access).
35. There was no way that VMware could have drivers for the thousands of different I/O devices in existence. By having VMware Workstation be a type 2 hypervisor, it could solve the problem by indirectly using the drivers already installed in the host OS.
36. VMware ESXi has been made small so it can be put into the firmware of the servers. When the server is turned on, the BIOS can then copy itself to RAM and start creating virtual machines. This greatly simplifies the booting and startup process.
37. Several examples can be found at: *virtualboximages.com*. These include various distributions of preinstalled Open Source Operating Systems. For example, rather than get an ISO for a new Linux flavor go through the install process and then get the VM running, it is easier to download the preinstalled VDI. There are similar appliances that run on VMWare. Other examples can be found at: <http://www.turnkeylinux.org>

**SOLUTIONS TO CHAPTER 8 PROBLEMS**

1. Both USENET and SETI@home could be described as wide area distributed systems. However, USENET is actually more primitive than the scheme of Fig. 8-1(c), since it does not require any network infrastructure other than point-to-point connections between pairs of machines. Also, since it does no processing work beyond that necessary to ensure proper dissemination of news articles, it could be debated whether it is really a distributed system of the sort we are concerned with in this chapter. SETI@home is a more typical example of a wide area distributed system; data are distributed to remote nodes which then return results of calculations to the coordinating node.
2. Depending on how CPUs are connected to memory, one of them gets through first, for example, seizes the bus first. It completes its memory operation, then another one happens, etc. It is not predictable which one goes first, but if the system has been designed for sequential consistency, it should not matter.
3. A 200-MIPS machine will issue 200 million memory references/sec, consuming 200 million bus cycles or half of the bus' capacity. It takes only two CPUs to consume the entire bus. Caching drops the number of memory requests/sec to 20 million, allowing 20 CPUs to share the bus. To get 32 CPUs on the bus, each one could request no more than 12.5 million requests/sec. If only 12.5 million of the 200 million of the memory references go out on the bus, the cache miss rate must be  $12.5/200$ , or 6.25%. This means the hit rate is 93.75%.
4. CPUs 000, 010, 100, and 110 are cut off from memories 010 and 011.
5. Each CPU manages its own signals completely. If a signal is generated from the keyboard and the keyboard is not assigned to any particular CPU (the usual case), somehow the signal has to be given to the correct CPU to handle.
6. To issue a system call, a process generates a trap. The trap interrupts its own CPU. Somehow, the information that a slave CPU has had a trap has to be conveyed to the master CPU. This does not happen in the first model. If there are interprocessor trap instructions, this can be used to signal the master. If no such instructions exist, the slave can collect the parameters of the system call and put them in a data structure in memory that the master polls continuously when it is idle.
7. Here is a possible solution:

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              | Return to caller; critical region entered |

8. As noted in the text, we have little experience (and tools) for writing highly parallel desktop applications. Although desktop applications are sometimes multithreaded, the threads are often used to simplify I/O programming and thus they are not compute-intensive threads. The one desktop application area that has some chance at large-scale parallelization is video gaming, where many aspects of the game require significant (parallel) computation. A more promising approach is to parallelize operating system and library services. We have already seen examples of this in current hardware and OS designs. For example, network cards now have on-board parallel processors (network processors) that are used to speed up packet processing and offer higher-level network services at line speeds (e.g., encryption, intrusion detection, etc). As another example, consider the powerful processors found on video cards used to offload video rendering from the main CPU and offer higher-level graphics APIs to applications (e.g., Open GL). One can imagine replacing these special-purpose cards with single-chip multicore processors. Moreover, as the number of cores increases, the same basic approach can be used to parallelize other operating system and common library services.
9. Probably locks on data structures are sufficient. It is hard to imagine anything a piece of code could do that is critical and does not involve some kernel data structure. All resource acquisition and release uses data structures, for example. While it cannot be proven, probably locks on data structures are enough.
10. It takes 16 bus cycles to move the block and it goes both ways for each TSL. Thus every 50 bus cycles, 32 of them are wasted on moving the cache block. Consequently, 64% of the bus bandwidth is wasted on cache transfers.
11. Yes, it would, but the interpoll time might end up being very long, degrading performance. But it would be correct, even without a maximum.
12. It is just as good as TSL. It is used by preloading a 1 into the register to be used. Then that register and the memory word are atomically swapped. After the instruction, the memory word is locked (i.e., has a value of 1). Its previous value is now contained in the register. If it was previously locked, the word has not been changed and the caller must loop. If it was previously unlocked, it is now locked.
13. The loop consists of a TSL instruction (5 nsec), a bus cycle (10 nsec), and a JMP back to the TSL instruction (5 nsec). Thus in 20 nsec, 1 bus cycle is requested occupying 10 nsec. The loop consumes 50% of the bus.

14. Affinity scheduling has to do with putting the right thread on the right CPU. Doing so might well reduce TLB misses since these are kept inside each CPU. On the other hand, it has no effect on page faults, since if a page is in memory for one CPU, it is in memory for all CPUs.
15. (a) 2 (b) 4 (c) 8 (d) 5 (e) 3 (f) 4.
16. On a grid, the worst case is nodes at opposite corners trying to communicate. However, with a torus, opposite corners are only two hops apart. The worst case is one corner trying to talk to a node in the middle. For odd  $k$ , it takes  $(k - 1)/2$  hops to go from a corner to the middle horizontally and another  $(k - 1)/2$  hops to go to the middle vertically, for a total of  $k - 1$ . For even  $k$ , the middle is a square of four dots in the middle, so the worst case is from a corner to the most distant dot in that four-dot square. It takes  $k/2$  hops to get there horizontally and also  $k/2$  vertically, so the diameter is  $k$ .
17. The network can be sliced in two by a plane through the middle, giving two systems, each with a geometry of  $8 \times 8 \times 4$ . There are 128 links running between the two halves, for bisection bandwidth of 128 Gbps.
18. If we just consider the network time, we get 1 nsec per bit or 512-nsec delay per packet. To copy 64 bytes 4 bytes at a time, 320 nsec are needed on each side, or 640 nsec total. Adding the 512-nsec wire time, we get 1152 nsec total. If two additional copies are needed, we get 1792 nsec.
19. If we consider only the wire time, a 1-Gbps network delivers 125 MB/sec. Moving 64 bytes in 1152 nsec is 55.6 MB/sec. Moving 64 bytes in 1792 nsec is 35.7 MB/sec.
20. The time to move  $k$  bytes by programmed I/O is  $20k$  nsec. The time for DMA is  $2000 + 5k$  nsec. Equating these and solving for  $k$  we get the breakeven point at 133 bytes.
21. Clearly, the wrong thing happens if a system call is executed remotely. Trying to read a file on the remote machine will not work if the file is not there. Also, setting an alarm on the remote machine will not send a signal back to the calling machine. One way to handle remote system calls is to trap them and send them back to the originating site for execution.
22. First, on a broadcast network, a broadcast request could be made. Second, a centralized database of who has which page could be maintained. Third, each page could have a home base, indicated by the upper  $k$  bits of its virtual address; the home base could keep track of the location of each of its pages.

23. In this split, node 1 has *A*, *E*, and *G*, node 2 has *B* and *F*, and node 3 has *C*, *D*, *H*, and *I*. The cut between nodes 1 and 2 now contains *AB* and *EB* for a weight of 5. The cut between nodes 2 and 3 now contains *CD*, *CI*, *FI*, and *FH* for a weight of 14. The cut between nodes 1 and 3 now contains *EH* and *GH* for a weight of 8. The sum is 27.
24. The table of open files is kept in the kernel, so if a process has open files, when it is unfrozen and tries to use one of its files, the new kernel does not know about them. A second problem is the signal mask, which is also stored on the original kernel. A third problem is that if an alarm is pending, it will go off on the wrong machine. In general, the kernel is full of bits and pieces of information about the process, and they have to be successfully migrated as well.
25. Ethernet nodes must be able to detect collisions between packets, so the propagation delay between the two most widely separated nodes must be less than the duration of the shortest packet to be sent. Otherwise the sender may fully transmit a packet and not detect a collision even though the packet suffers a collision close to the other end of the cable.
26. The middleware runs on different operating systems, so the code is clearly different because the embedded system calls are different. What they have in common is producing a common interface to the application layer above them. If the application layer makes calls only to the middleware layer and no system calls, then all the versions of it can have the same source code. If they also make true system calls, these will differ.
27. The most appropriate services are
- (a) Unreliable connection.
  - (b) Reliable byte stream.
28. It is maintained hierarchically. There is a worldwide server for *.edu* that knows about all the universities and a *.com* server that knows about all the names ending in *.com*. Thus to look up *cs.uni.edu*, a machine would first look up *uni* at the *.edu* server, then go there to ask about *cs*, and so on.
29. A computer may have many processes waiting for incoming connections. These could be the Web server, mail server, news server, and others. Some way is needed to make it possible to direct an incoming connection to some particular process. That is done by having each process listen to a specific port. It has been agreed upon that Web servers will listen to port 80, so incoming connections directed to the Web server are sent to port 80. The number itself was an arbitrary choice, but some number had to be chosen.
30. Physical I/O devices still present problems because they do not migrate with the virtual machine, yet their registers may hold state that is critical to the proper functioning of the system. Think of read or write operations to devices

(e.g., the disk) that have been issued but have not yet completed. Network I/O is particularly difficult because other machines will continue to send packets to the hypervisor, unaware that the virtual machine has moved. Even if packets can be redirected to the new hypervisor, the virtual machine will be unresponsive during the migration period, which can be long because the entire virtual machine, including the guest operating system and all processes executing on it, must be moved to the new machine. As a result packets can experience large delays or even packet loss if the device/hypervisor buffers overflow.

31. One way would be for the Web server to package the entire page, including all the images, in a big zip file and send the whole thing the first time so that only one connection is needed. A second way would be to use a connectionless protocol like UDP. This would eliminate the connection overhead, but would require servers and browsers to do their own error control.
32. Having the value of a read depend on whether a process happens to be on the same machine as the last writer is not at all transparent. This argues for making changes only visible to the process making the changes. On the other hand, having a single cache manager per machine is easier and cheaper to implement. Such a manager becomes a great deal more complicated if it has to maintain multiple copies of each modified file, with the value returned depending on who is doing the reading.
33. Shared memory works with whole pages. This can lead to false sharing, in which access to unrelated variables that happen to lie on the same page causes thrashing. Putting each variable on a separate page is wasteful. Object-based access eliminates these problems and allows a finer grain of sharing.
34. Hashing on any of the fields of the tuple when it is inserted into the tuple space does not help because the *in* may have mostly formal parameters. One optimization that always works is noting that all the fields of both *out* and *in* are typed. Thus the type signature of all tuples in the tuple space is known, and the tuple type needed on an *in* is also known. This suggests creating a tuple subspace for each type signature. For example, all the (int, int, int) tuples go into one space, and all the (string, int, float) tuples go into a different space. When an *in* is executed, only the matching subspace has to be searched.

**SOLUTIONS TO CHAPTER 9 PROBLEMS**

1. Popular news services require integrity and availability but not confidentiality. Backup storage systems require confidentiality and integrity but not necessarily 24/7 availability. Finally, banking services require confidentiality, integrity, and availability.
2. (a) and (c) have to be a part of the TCB and (b), (d) and (e) can be implemented outside the TCB.
3. A covert channel is an unauthorized communications channel that can be created in a system by observing and manipulating measurable performance characteristics of the system. The key requirement of a covert channel to exist is to have some shared system resources such as CPU, disk, or network that can be used for sending secret signals.
4. It is just entered into the matrix twice. In the example given in the text, *printer1* is in two domains simultaneously. There is no problem here.
5. Full protection matrix:  $5000 \times 100 = 500,000$  units of space.  
 ACL:  
 $50 \times 100$       (1% objects accessible in all domains; 100 entries/ACL)  
 $+ 500 \times 2$       (10% objects accessible in two domains; two entries/ACL)  
 $+ 4450 \times 1$       (89% objects accessible in one domain; one entry/ACL)  
 $= 10,450$  units of space  
 The space needed for storing a capability list will be same as that for ACL.
6. (a) Capability list  
 (b) Capability list  
 (c) ACL  
 (d) ACL
7. To make a file readable by everyone *except* one person, access-control lists are the only possibility. For sharing private files, access-control lists or capabilities can be used. To make files public, access-control lists are easiest, but it may also be possible to put a capability for the file or files in a well-known place in a capability system.
8. Here is the protection matrix:



| Object |               |              |               |               |
|--------|---------------|--------------|---------------|---------------|
| Domain | PPP-Notes     | prog1        | project.t     | splash.gif    |
| asw    | Read          | Read<br>Exec | Read<br>Write | Read<br>Write |
| gmw    | Read<br>Write |              | Read<br>Write |               |
| users  | Read          |              | Read<br>Write |               |
| devel  |               | Read<br>Exec |               | Read          |

9. The ACLs are as follows:

| File       | ACL                    |
|------------|------------------------|
| PPP-Notes  | gmw:RW; *:R            |
| prog1      | asw:RWX; devel:RX; *:R |
| project.t  | asw:RW; users:RW       |
| splash.gif | asw:RW; devel:R        |

Assume that \* means all.

10. If *asw* wants to allow *gmw* but no other members of users to look at *splash.gif*, he could modify the ACL to *asw:RW; devel:R; gmw:R*.
11. Bell-LaPadula model only: (a), (b), (h)  
Biba model only: (e), (i)  
Both: (c), (d), (f), (g)
12. The server will verify that the capability is valid and then generate a weaker capability. This is legal. After all, the friend can just give away the capability it already has. Giving it the power to give away something even weaker is not a security threat. If you have the ability to give away, say, read/write power, giving away read-only power is not a problem.
13. No. That would be writing down, which violates the \* property.
14. A process writing to another process is similar to a process writing to a file. Consequently, the \* property would have to hold. A process could write up but not write down. Process *B* could send to *C*, *D*, and *E*, but not to *A*.
15. In the original photo, the R, G, and B axes each allow discrete integral values from 0 to 255, inclusive. This means that there are  $2^{24}$  valid points in color space that a pixel can occupy. When 1 bit is taken away for the covert channel, only the even values are allowed (assuming the secret bit is replaced by a 0

everywhere). Thus as much of the space is covered, but the color resolution is only half as good. In total, only  $1/8$  of the colors can be represented. The disallowed colors are mapped onto the adjacent color all of whose values are even numbers, for example, the colors (201, 43, 97), (201, 42, 97), (200, 43, 96), and (200, 42, 97) now all map onto the point (200, 42, 96) and can no longer be distinguished.

16. the time has come the walrus said to talk of many things  
 of ships and shoes and sealing wax of cabbages and kings  
 of why the sea is boiling hot and whether pigs have wings  
 but wait a bit the oysters cried before we have our chat  
 for some of us are out of breath and all of us are fat  
 no hurry said the carpenter they thanked him much for that

From *Through the Looking Glass* (Tweedledum and Tweedledee).

17. The constraint is that no two cells contain the same two letters, otherwise decryption would be ambiguous. Thus each of the 676 matrix elements contains a different one of the 676 digrams. The number of different combinations is thus  $676!$  This is a very big number.
18. The number of permutations is  $n!$ , so this is the size of the key space. One advantage is that the statistical attack based on properties of natural languages does not work because an E really does represent an E, and so on.
19. The sender picks a random key and sends it to the trusted third party encrypted with the secret key that they share. The trusted third party then decrypts the random key and reencrypts it with the secret key it shares with the receiver. This message is then sent to the receiver.
20. A function like  $y = x^k$  is easy to compute but taking the  $k$ th root of  $y$  is far more difficult.
21. A and B pick random keys  $Ka$  and  $Kb$  and send them to C encrypted with C's public key. C picks a random key  $K$  and sends it to A encrypted using  $Ka$  and to B encrypted using  $Kb$ .
22. One way to sign a document would be for the smart card to read in the document, make a hash of it, and then encrypt the hash with the user's private key, stored in the card. The encrypted hash would be output to the Internet cafe computer, but the secret key would never leave the smart card, so the scheme is secure.
23. The image contains 1,920,000 pixels. Each pixel has 3 bits that can be used, given a raw capacity of 720,000 bytes. If this is effectively doubled due to compressing the text before storing it, the image can hold ASCII text occupying about 1,440,000 bytes before compression. There is no expansion due to

the steganography. The image with the hidden data is the same size as the original image. The efficiency is 25%. This can be easily seen from the fact that 1 bit of every 8-bit color sample contains payload, and the compression squeezes 2 bits of ASCII text per payload bit. Thus per 24-bit pixel, effectively 6 bits of ASCII text are being encoded.

24. The dissidents could sign the messages using a private key and then try to widely publicize their public key. This might be possible by having someone smuggle it out of the country and then post it to the Internet from a free country.
25. (a) Both files are 2.25 MB.  
 (b) *Hamlet*, *Julius Caesar*, *King Lear*, *Macbeth*, and *Merchant of Venice*.  
 (c) There are six text files secretly stored, totaling about 722 KB.
26. It depends on how long the password is. The alphabet from which passwords is built has 62 symbols. The total search space is  $62^5 + 62^6 + 62^7 + 62^8$ , which is about  $2 \times 10^{14}$ . If the password is known to be  $k$  characters, the search space is reduced to only  $62^k$ . The ratio of these is thus  $2 \times 10^{14}/62^k$ . For  $k$  from 5 to 8, these values are 242,235, 3907, 63, and 1. In other words, learning that the password is only five characters reduces the search space by a factor of 242,235 because all the long passwords do not have to be tried. This is a big win. However, learning that it is eight characters does not help much because it means that all the short (easy) passwords can be skipped.
27. Try to calm the assistant. The password encryption algorithm is public. Passwords are encrypted by the *login* program as soon as they are typed in, and the encrypted password is compared to the entry in the password file.
28. No, it does not. The student can easily find out what the random number for his superuser is. This information is in the password file unencrypted. If it is 0003, for example, then he just tries encrypting potential passwords as *Susan0003*, *Boston0003*, *IBMPC0003*, and so on. If another user has password *Boston0004*, he will not discover it, however.
29. Suppose there are  $m$  users in the systems. The cracker can then collect the  $m$  salt values, assumed all different here. The cracker will have to try encrypting each guessed password  $m$  times, once with each of the  $m$  salts used by the system. Thus the cracker's time to crack all passwords is increased  $m$ -fold.
30. There are many criteria. Here are a few of them:
  - It should be easy and painless to measure (not blood samples).
  - There should be many values available (not eye color).
  - The characteristic should not change over time (not hair color).
  - It should be difficult to forge the characteristic (not weight).

31. The combination of different authentication mechanisms will provide stronger authentication. However, there are two drawbacks. First, the cost involved in implementing this system is high. The system incurs the cost of three different authentication mechanisms. Second, this authentication mechanism puts extra burden on the user. The user has to remember his login/password, carry his plastic card and remember its PIN, and has to go through the process of fingerprint matching. The key issue is that all this will increase the time it takes to authenticate a user, resulting in increased user dissatisfaction.
32. If all the machines can be trusted, it works OK. If some cannot be trusted, the scheme breaks down, because an untrustworthy machine could send a message to a trustworthy machine asking it to carry out some command on behalf of the superuser. The machine receiving the message has no way of telling if the command really did originate with the superuser, or with a student.
33. It would not work to use them forward. If an intruder captured one, he would know which one to use next time. Using them backward prevents this danger.
34. No, it is not feasible. The problem is that array bounds are not checked. Arrays do not line up with page boundaries, so the MMU is not of any help. Furthermore, making a kernel call to change the MMU on every procedure call would be prohibitively expensive.
35. The attacker exploits the race condition by executing an operation like symbolic link after the access rights are checked and before the file is opened. If the file system access is a transaction, the access rights check and file open will be a part of a single transaction and the serializability property will ensure that symbolic link cannot be created in between. The main downside of this approach is that the performance of the file system will suffer since transactions incur extra overhead.
36. The compiler could insert code on all array references to do bounds checking. This feature would prevent buffer overflow attacks. It is not done because it would slow down all programs significantly. In addition, in C it is not illegal to declare an array of size 1 as a procedure parameter and then reference element 20, but clearly the actual array whose address has been passed had better have at least 20 elements. In addition, C functions like *memset* and *memcpy* are used all the time to copy entire structures at once even if they contain separate arrays. In other words, they are buffer overflows by design.
37. If the capabilities are used to make it possible to have small protection domains, no; otherwise yes. If an editor, for example, is started up with only the capabilities for the file to be edited and its scratch file, then no matter what tricks are lurking inside the editor, all it can do is read those two files. On the other hand, if the editor can access all of the user's objects, then Trojan horses can do their dirty work, capabilities or not.

38. From a security point of view, it would be ideal. Used blocks sometimes are exposed, leaking valuable information. From a performance point of view, zeroing blocks wastes CPU time, thus degrading performance.
39. For any operating system all programs must either start execution at a known address or have a starting address stored in a known position in the program file header. (a) The virus first copies the instructions at the normal start address or the address in the header to a safe place, and then inserts a jump to itself into the code or its own start address into the header. (b) When done with its own work, the virus executes the instructions it borrowed, followed by a jump to the next instruction that would have been executed, or transfers control to the address it found in the original header.
40. A master boot record requires only one sector, and if the rest of the first track is free, it provides space where a virus can hide the original boot sector as well as a substantial part of its own code. Modern disk controllers read and buffer entire tracks at a time, so there will be no perceivable delay or sounds of additional seeks as the extra data are read.
41. C programs have extension `.c`. Instead of using the `access` system call to test for execute permission, examine the file name to see if it ends in `.c`. This code will do it:

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

42. They probably cannot tell, but they can guess that XORing one word within the virus with the rest will produce valid machine code. Their computers can just try each virus word in turn and see if any of them produce valid machine code. To slow down this process, Virgil can use a better encryption algorithm, such as using different keys for the odd and even words, and then rotating the first word left by some number of bits determined by a hash function on the keys, rotating the second word that number of bits plus one, and so on.
43. The compressor is needed to compress other executable programs as part of the process of infecting them.
44. Most viruses do not want to infect a file twice. It might not even work. Therefore it is important to be able to detect the virus in a file to see if it is already infected. All the techniques used to make it hard for antivirus software to detect viruses also make it hard for the virus itself to tell which files have been infected.

45. First, running the Ifdisk program from the hard disk is a mistake. It may be infected and it may infect the boot sector. It has to be run from the original CD-ROM or a write-protected floppy disk. Second, the restored files may be infected. Putting them back without cleaning them may just reinstall the virus.
46. Yes, but the mechanism is slightly different from Windows. In UNIX a companion virus can be installed in a directory on the search path ahead of the one in which the real program lives. The most common example is to insert a program *ls* in a user directory, which effectively overrides */bin/ls* because it is found first.
47. Obviously, executing any program from an unknown source is dangerous. Self-extracting archives can be especially dangerous, because they can release multiple files into multiple directories, and the extraction program itself could be a Trojan horse. If a choice is available it is much better to obtain files in the form of an ordinary archive, which you can then extract with tools you trust.
48. Since a rootkit is designed to conceal its existence, it infects operating system, libraries and applications. So, any detection software that relies on any system functionality cannot be trusted. Essentially, a rootkit subverts the software that is intended to find it. As a result, rootkit detectors have to rely on external components such as scanning from an external TCB.
49. Since a rootkit can subvert the recovery software, for example, by resetting the system restore points, this approach to system recovery does not work.
50. It is not possible to write such a program, because if such a program is possible, a cracker can use that program to circumvent virus checking in the virus-laden program she writes.
51. The source IP address of all incoming packets can be inspected. The second set of rules will drop all incoming IP packets with source IP addresses belonging to known spammers.
52. It does not matter. If zero fill is used, then S2 must contain the true prefix as an unsigned integer in the low-order  $k$  bits. If sign extension is used, then S2 must also be sign extended. As long as S2 contains the correct results of shifting a true address, it does not matter what is in the unused upper bits of S2.
53. Existing browsers come preloaded with the public keys of several trusted third parties such as the Verisign Corporation. Their business consists of verifying other companies' public keys and making up certificates for them. These certificates are signed by, for example, Verisign's private key. Since Verisign's public key is built into the browser, certificates signed with its private key can be verified.

**54.** First, Java does not provide pointer variables. This limits a process' ability to overwrite an arbitrary memory location. Second, Java does not allow user-controlled storage allocation (*malloc/free*). This simplifies memory management. Third, Java is a type-safe language, ensuring that a variable is used in exactly way it is supposed to be, based on its type.

**55.** Here are the rules.

| URL                | Signer     | Object                     | Action        |
|--------------------|------------|----------------------------|---------------|
| www.appletsRus.com | AppletsRus | /usr/me/appletdir/*        | Read          |
| www.appletsRus.com | AppletsRUs | /usr/tmp/*                 | Read, Write   |
| www.appletsRus.com | AppletsRUs | www.appletsRus; port: 5004 | Connect, Read |

**56.** An applet is any small program that performs a specific task such as filling up a form. The main difference between an applet and an application is that an applet runs within the scope of a dedicated larger program such rendering a webpage. Applets are typical examples of auxiliary applications that don't monopolize the user's attention and are intended to be easily accessible. Since applets are typically downloaded from a third party, they essentially contain foreign code designed to run on a user's machine. They may contain viruses, worms, or other harmful code.

**SOLUTIONS TO CHAPTER 10 PROBLEMS**

1. Since assembly language is specific to each machine, a port of UNIX to a new machine required rewriting the entire code in the new machine's assembly language. On the other hand, once UNIX was written in C, only a small part of the OS (e.g. device drivers for I/O devices) had to be rewritten.
2. System call interface is tightly coupled to the OS kernel. Standardizing the system call interface would have put severe restrictions (less flexibility) on the design of the OS kernel. It would also make UNIX less portable.
3. This allows Linux to use special capabilities (like language extensions) of the gcc compiler that range from providing shortcuts and simplifications to providing the compiler with hints for optimization. The main disadvantage is that there are other popular, feature-rich C compilers like LLVM that cannot be used to compile Linux. If at some future time LLVM or some other compiler becomes better than gcc in all ways, Linux will not be able to use it. This could become a serious problem.
4. The files that will be listed are: *bonefish*, *quacker*, *seahorse*, and *weasel*.
5. It prints the number of lines of the file *xyz* that contain the string "nd" in them.
6. The pipeline is as follows:

```
head -8 z | tail -1
```

The first part selects out the first eight lines of *z* and passes them to *tail*, which just writes the last one on the screen.

7. They are separate, so standard output can be redirected without affecting standard error. In a pipeline, standard output may go to another process, but standard error still writes on the terminal.
8. Each program runs in its own process, so six new processes are started.
9. Yes. The child's memory is an exact copy of the parent's, including the stack. Thus if the environment variables were on the parent's stack, they will be on the child's stack, too.
10. Since text segments are shared, only 36 KB has to be copied. The machine can copy 80 bytes per microsec, so 36 KB takes 0.46 msec. Add another 1 msec for getting into and out of the kernel, and the whole thing takes roughly 1.46 msec.
11. Linux relies on copy on write. It gives the child pointers to the parent's address space, but marks the parent's pages write-protected. When the child attempts to write into the parent's address space, a fault occurs, and a copy of the parent's page is created and allocated into the child's address space.



12. Negative values allow the process to have priority over all normal processes. Users cannot be trusted with such power. It is only for the superuser and then used only in critical situations.
13. The text says that the nice value is in the range  $-20$  to  $+19$ , so the default static priority must be 120, which it is indeed. By being nice and selecting a positive nice value, a process can request to be put under lower priority.
14. Yes. It cannot run any more, so the earlier its memory goes back on the free list, the better.
15. Signals are like hardware interrupts. One example is the alarm signal, which signals the process at a specific number of seconds in the future. Another is the floating-point exception signal, which indicates division by zero or some other error. Many other signals also exist.
16. Malicious users could wreak havoc with the system if they could send signals to arbitrary unrelated processes. Nothing would stop a user from writing a program consisting of a loop that sent a signal to the process with PID  $i$  for all  $i$  from 1 to the maximum PID. Many of these processes would be unprepared for the signal and would be killed by it. If you want to kill off your own processes, that is all right, but killing off your neighbor's processes is not acceptable.
17. It would be impossible using Linux or Windows, but the Pentium hardware does make this possible. What is needed is to use the segmentation features of the hardware, which are not supported by either Linux or Windows. The operating system could be put in one or more global segments, with protected procedure calls to make system calls instead of traps. OS/2 works this way.
18. Generally, daemons run in the background doing things like printing and sending e-mail. Since people are not usually sitting on the edge of their chairs waiting for them to finish, they are given low priority, soaking up excess CPU time not needed by interactive processes.
19. A PID must be unique. Sooner or later the counter will wrap around and go back to 0. Then it will go upward to, for example, 15. If it just happens that process 15 was started months ago, but is still running, 15 cannot be assigned to a new process. Thus after a proposed PID is chosen using the counter, a search of the process table must be made to see if the PID is already in use.
20. When the process exits, the parent will be given the exit status of its child. The PID is needed to be able to identify the parent so the exit status can be transferred to the correct process.

21. A page may now be shared by all three processes in this case. In general, the copy-on-write mechanism may result in several processes sharing a page. To handle this situation, the OS has to keep a reference count for each shared page. If *p1* writes on a page shared three ways, it gets a private copy while the other two processes continue to share their copy.
22. If all of the *sharing\_flags* bits are set, the clone call starts a conventional thread. If all the bits are cleared, the call is essentially a fork.
23. Each scheduling decision requires looking up a bitmap for the active array and searching for the first set bit in the array, which can be done in constant time, dequeuing a single task from the selected queue, again a constant-time operation, or if the bitmap value is zero, swapping the values of the active and expired lists, again a constant-time operation.
24. Clock interrupts, especially at high frequency, can eat up a fair amount of CPU time. Their main function (other than keeping track of the time of day, which can be done other ways) is to determine when to preempt a long-running process. However, under normal conditions, a process makes hundreds of system calls per second, so the kernel can check on each call if the running process has run too long. To handle the case of a completely CPU-bound process running too long, a timer can be set up to go off in, say, 1 second, just in case it makes no system calls. If there is only one process, preemption is not needed, so ticklessness is just fine.
25. The program loaded from block 0 is a maximum of 512 bytes long, so it cannot be very complicated. Loading the operating system requires understanding the file system-layout in order to find and load the operating system. Different operating systems have very different file systems; it is asking too much to expect a 512-byte program to sort all this out. Instead, the block 0 loader just fetches another loader from a fixed location on the disk partition. This program can be much longer and system specific so that it can find and load the OS.
26. With shared text, 100 KB is needed for the text. Each of the three processes needs 80 KB for its data segment and 10 KB for its stack, so the total memory needed is 370 KB. Without shared text, each program needs 190 KB, so three of them need a total of 570 KB.
27. Processes sharing a file, including the current file-pointer position, can just share an open file descriptor, without having to update anything in each other's private file-descriptor tables. At the same time, another process can access the same file through a separate open-file descriptor, obtain a different file pointer, and move through the file at its own will.

28. The text segment cannot change, so it never has to be paged out. If its frames are needed, they can just be abandoned. The pages can always be retrieved from the file system. The data segment must not be paged back to the executable file, because it is likely that it has changed since being brought in. Paging it back would ruin the executable file. The stack segment is not even present in the executable file.
29. Two processes could map the same file into their address spaces at the same time. This gives them a way to share physical memory. Half of the shared memory could be used as a buffer from *A* to *B* and half as a buffer from *B* to *A*. To communicate, one process writes a message to its part of the shared memory, then a signal to the other one to indicate there is a message waiting for it. The reply could use the other buffer.
30. Memory address 65,536 is file byte 0, so memory address 72,000 is byte 6464.
31. Originally, four pages worth of the file were mapped: 0, 1, 2, and 3. The call succeeds and after it is done, only pages 2 and 3 are still mapped, that is, bytes 16,384 through 32,767.
32. It is possible. For example, when the stack grows beyond the bottom page, a page fault occurs and the operating system normally assigns the next-lowest page to it. However, if the stack has bumped into the data segment, the next page cannot be allocated to the stack, so the process must be terminated because it has run out of virtual address space. Also, even if there is another page available in virtual memory, the paging area of the disk might be full, making it impossible to allocate backing store for the new page, which would also terminate the process.
33. It is possible if the two blocks are not buddies. Consider the situation of Fig. 10-17(e). Two new requests come in for eight pages each. At this point the bottom 32 pages of memory are owned by four different users, each with eight pages. Now users 1 and 2 release their pages, but users 0 and 3 hold theirs. This yields a situation with eight pages used, eight pages free, eight pages free, and eight pages used. We have two adjacent blocks of equal size that cannot be merged because they are not buddies.
34. Paging to a partition allows the use of a raw device, without the overhead of using file-system data structures. To access block *n*, the operating system can calculate its disk position by just adding it to the starting block of the partition. There is no need to go through all the indirect blocks that would otherwise be needed.
35. Opening a file by a path relative to the working directory is usually more convenient for the programmer or user, since a shorter path name is needed. It is also usually much simpler and requires fewer disk accesses.

**36.** The results are as follows:

- (a) The lock is granted.
- (b) The lock is granted.
- (c) *C* is blocked, since bytes 20 through 30 are unavailable.
- (d) *A* is blocked, since bytes 20 through 25 are unavailable.
- (e) *B* is blocked, since byte 8 is unavailable for exclusive locking.

At this point we now have a deadlock. None of the processes will ever be able to run again.

- 37.** The issue arises of which process gets the lock when it becomes available. The simplest solution is to leave it undefined. This is what POSIX does because it is the easiest to implement. Another is to require the locks to be granted in the order they were requested. This approach is more work for the implementation, but prevents starvation. Still another possibility is to let processes provide a priority when asking for a lock, and use these priorities to make a choice.
- 38.** A process will request a shared lock if it wants to read some bytes, whereas it will request an exclusive lock if it wants update some bytes. A process requesting an exclusive lock may be blocked indefinitely if there is a sequence of processes requesting shared locks. In other words, if readers always go before writers, writers could suffer from starvation.
- 39.** The owner can read, write, and execute it, and everyone else (including the owner's group) can just read and execute it, but not write it.
- 40.** Yes. Any block device capable of reading and writing an arbitrary block can be used to hold a file system. Even if there were no way to seek to a specific block, it is always possible to rewind the tape and then count forward to the requested block. Such a file system would not be a high-performance file system, but it would work. The author has actually done this on a PDP-11 using DECtapes and it works.
- 41.** No. The file still has only one owner. If, for example, only the owner can write on the file, the other party cannot. Linking a file into your directory does not suddenly give you any rights you did not have before. It just creates a new path for accessing the file.
- 42.** When the working directory is changed, using `chdir`, the i-node for the new working directory is fetched and kept in memory, in the i-node table. The i-node for the root directory is also there. In the user structure, pointers to both of these are maintained. When a path name has to be parsed, the first character is inspected. If it is a `"/"`, the pointer to the root i-node is used as the starting place, otherwise the pointer to the working directory's i-node is used.

43. Access to the root directory's i-node does not require a disk access, so we have the following:
1. Reading the / directory to look up "usr".
  2. Reading in the i-node for /usr.
  3. Reading the /usr directory to look up "ast".
  4. Reading in the i-node for /usr/ast.
  5. Reading the /usr/ast directory to look up "work".
  6. Reading in the i-node for /usr/ast/work.
  7. Reading the /usr/ast/work directory to look up "f".
  8. Reading in the i-node for /usr/ast/work/f.

Thus, in total, eight disk accesses are needed to fetch the i-node.

44. The i-node holds 12 addresses. The single indirect block holds 256. The double indirect block leads to 65,536, and the triple indirect leads to 16,777,216, for a total of 16,843,018 blocks. This limits the maximum file size to  $12 + 256 + 65,536 + 16,777,218$  blocks, which is about 16 gigabytes.
45. When a file is closed, the counter of its i-node in memory is decremented. If it is greater than zero, the i-node cannot be removed from the table because the file is still open in some process. Only when the counter hits zero can the i-node be removed. Without the reference count, the system would not know when to remove the i-node from the table. Making a separate copy of the i-node each time the file was opened would not work because changes made in one copy would not be visible in the others.
46. By maintaining per-CPU runqueues, scheduling decisions can be made locally, without executing expensive synchronization mechanisms to always access, and update, a shared runqueue. Also, it is more likely that all relevant memory pages will still be in the cache if we schedule a thread on the same CPU where it already executed.
47. One disadvantage is security, as a loadable module may contain bugs and exploits. Another disadvantage is that the kernel virtual address space may become fragmented as more and more modules are loaded.
48. By forcing the contents of the modified file out onto the disk every 30 sec, damage done by a crash is limited to 30 sec. If *pdflush* did not run, a process might write a file, then exit with the full contents of the file still in the cache. In fact, the user might then log out and go home with the file still in the cache. An hour later the system might crash and lose the file, still only in the cache and not on disk. The next day we would not have a happy user.
49. All it has to do is set the link count to 1, since only one directory entry references the i-node.

- 50. It is generally `getpid`, `getuid`, `getgid`, or something like that. All they do is fetch one integer from a known place and return it. Every other call does more.
- 51. The file is simply removed. This is the normal way (actually, the only way) to remove a file.
- 52. A 1.44-MB floppy disk can hold 1440 blocks of raw data. The boot block, superblock, group descriptor block, block bitmap, and i-node bitmap of an ext2 file system each use one block. If 8192 128-byte i-nodes are created, these i-nodes will occupy another 1024 blocks, leaving only 411 blocks unused. At least one block is needed for the root directory, leaving space for 410 blocks of file data. Actually the Linux *mkfs* program is smart enough not to make more i-nodes than can possibly be used, so the inefficiency is not this bad. By default 184 i-nodes occupying 23 blocks will be created. However, because of the overhead of the ext2 file system, Linux normally uses the MINIX 1 file system on floppy disks and other small devices.
- 53. It is often essential to have someone who can do things that are normally forbidden. For example, a user starts up a job that generates an infinite amount of output. The user then logs out and goes on a three-week vacation to London. Sooner or later the disk will fill up, and the superuser will have to manually kill the process and remove the output file. Other such examples exist.
- 54. Probably someone had the file open when the professor changed the permissions. The professor should have deleted the file and then put another copy into the public directory. Also, he should use a better method for distributing files, such as a Web page, but that is beyond the scope of this exercise.
- 55. If, say, superuser rights are given to another user with the `fsuid` system call, that user can access superuser files, but will not be able to send signals, kill processes, or perform other operations which require superuser privileges.
- 56. (a) Most of the files are listed as zero bytes in size and contain a large amount of information. This is because these files do not exist on disk. The system retrieves the information from the actual process as needed.  
(b) Most of the time and date settings reflect the current time and date. This is due to the fact that the information is just retrieved and, in many cases, it is constantly updated.  
(c) Most of the files are read only. This is because the information they provide is related to the process and cannot be changed by users.
- 57. The activity will need to save the user's current scroll position and zoom level to be able to return showing the same part of the web page as it had been. It will also need to save any data the user had input in to form fields so they will not be lost. It does not need to save anything it had downloaded to display the web page (HTML, images, JavaScript, etc); this can always be re-fetched and

parsed when the activity is restored, and usually that data will still be in the browser's local on-disk cache.

58. You need to keep the device running while you are reading and storing the downloaded data, so the networking code should hold a wake lock during this time. (Extra credit: the application will also need to have requested the *internet* permission, as implied by Fig. 10-63.)
59. All of your threads need to be started after forking. The new process will only have the forking thread running in it; if you had already started other threads, they would not be in the new process, and would leave the process in a poorly defined state.
60. You know that the caller has in some way been explicitly handed a reference to your original object. However you do not know if the caller is the same process that you originally sent the object to—that process may have itself given the object to another process.
61. As more RAM is needed, the out-of-memory killer will start killing process in order from least necessary to most. From the order of operations we have been given, we can determine that we have processes running with these out-of-memory levels:
  1. browser: FOREGROUND
  2. email: SERVICE
  3. launcher: HOME
  4. camera: CACHED

The out-of-memory killer will thus start at the bottom and work up, first killing the camera process, then launcher, and finally e-mail.

**SOLUTIONS TO CHAPTER 11 PROBLEMS**

1. An advantage is that everything is in one place, which makes it easy to find. A disadvantage is that one bad disk block in the top-level index in a hive can wreak disaster on the whole system.
2. The HAL is simple and straightforward. Including the mouse, the disk, and all the other device drivers in it would make it unwieldy and destroy its function as a thin layer that hides certain basic hardware differences of the computer itself, but not the I/O devices.
3. A genealogical database might find it convenient to record the birth and death dates of one's ancestors using the standard-system-time format. In fact, any historical database might use this.
4. If handles contained a sequence number, then when a handle was closed and continued to be used, this could easily be detected by comparing the sequence in each handle to a sequence number in the handle table. Room for the sequence number would have to be found in both the handle and the table entry. Every time a handle-table entry is reused the sequence number is incremented, and the sequence number is embedded in the handle. If the sequence number is  $N$  bits, then a particular handle would have to be closed and reopened  $2^N$  times to avoid detection. So for even a small  $N$  a lot of handle race conditions could be detected in programs.
5. (a) The process manager uses the object manager to create threads.  
(b) The memory manager uses the security manager to see if a file can be mapped.  
(c) The plug-and-play manager uses the configuration manager to register a new device.
6. A signal is handled by a new thread in some process' context, for example, when the Quit key is hit or even when a thread faults. It does not really make any sense to catch a signal in a thread's context. It really has to be per process. Thus, signal handling is really a per-process activity.
7. It would make more sense on servers. Client machines have fewer concurrent processes. Shared libraries make sense only if there are multiple processes sharing them. Otherwise, it is more efficient to statically link the libraries and accept duplication. The advantage of static linking is that only those procedures that are actually needed are loaded. With DLLs there may be procedures in memory that no one is using.
8. The question is about whether a kernel thread could use the same stack as its corresponding user thread, which would make sense, as most system calls start with an empty stack and return to user mode with the stack again empty. One reason these stacks are separate is because of the overhead that would be re-



quired vs. the cost of maintaining separate stacks. The overhead is because the kernel would have to protect the part of the user stack it was going to run on and make sure it was large enough or else get another thread to request it be made larger. This could perhaps be avoided by having the kernel always use a fixed set of pages at the top of the user-mode stack, but then that is like having a separate stack. Another reason for the separation is that the kernel likes to know where data comes from so it can decide how much validation is needed. If a buffer is allocated on a separate kernel stack, the buffer will have a kernel address and checking is normally not needed. If the kernel stack uses part of the user stack, then the kernel will have more difficulty telling whether a stack buffer is from the kernel thread or the user thread. Additionally the kernel often relies on the fact that the kernel stacks cannot be paged out. So the pages of the user stack the kernel was going to use would have to be locked into memory. On the other hand, it is not very expensive to allocate a separate stack—far less expensive than solving all these problems. What makes threads expensive is not the stacks they use, it is all the metadata required to manage them.

9. The hit rate on the page table entries cached in the TLB has a big impact on system performance. The operation to walk the page tables and find a missing entry is very expensive. Since the TLB has only a limited number of entries, using 2-MB large pages greatly increases how many virtual addresses can be mapped by the TLB at a time. Large pages will waste large amounts of memory because of the unused space at the end of the final page within a region of the file. So they are effective only when used with very large regions. But even so, they increase memory pressure on memory because a large page is likely to have large amounts of data that is not currently being accessed, and would have been paged out to disk if using 4-KB pages.
10. There is a limit of 32 operations because there are only 32 rights bits in the object handle.
11. It is not possible because semaphores and mutexes are executive objects and critical sections are not. They are managed mostly in user space (but do have a backing semaphore when blocking is needed). The object manager does not know about them and they do not have handles, as was stated in the text. Since `WaitForMultipleObjects` is a system call, the system cannot perform a Boolean OR of several things, one of which it knows nothing about. The call must be a system call because semaphores and mutexes are kernel objects. In short, it is not possible to have any system call that mixes kernel objects and user objects like this. It has to be one or the other.
12. The variable might be a pointer to a dynamically allocated data structure, where the initialization and allocation of the structure was the responsibility of the first thread that wanted to use it. If two threads attempt to initialize the

variable at nearly the same time, two different structures might be allocated and one of threads might use the wrong instance. `InitOnceExecuteOnce` uses an extra variable, say `foo` to record the state of each separate initialization. Valid states are: *uninitialized*, *initializing*, *initialized*. The thread that will actually do the initialization records atomically sets `foo` to *initializing*. When initialization is complete it atomically sets `foo` to *initialized* and calls `WakeByAddressAll`. Atomically setting and examining a variable can be performed either by using a lock or a hardware instruction like *compare&swap*. If a thread finds that `foo` is set to *initialized*, then it skips the initialization. If it finds `foo` set to *initializing*, it calls `WaitOnAddress` to wait for initialization to complete.

13. (a) The last thread exits.  
(b) A thread executes `ExitProcess`.  
(c) Another process with a handle to this one kills it.  
(Modern apps) The OS decided to terminate it to reclaim room in the swap file or because the application was being serviced.
14. The operating system terminates modern applications mainly when the system is low on memory or is being rebooted. If applications were to attempt to run in the former case, there might not be enough resources for them to successfully save their state. And in the latter case they might indefinitely delay the shutdown of Windows, as often happens in desktop Windows. Though users might switch between applications often, the frequency of those switches is at the human scale of seconds or minutes. For a well-written application, the few milliseconds required to save state does not have much impact. Additionally, writing applications this way creates a better user experience were the application to crash, since the last state save is available as a checkpoint.
15. Because the IDs are reused right away, a program that identified a process that it wanted to operate on by ID might find that the process had died, and the ID had been reused between the time it finds the ID and when it uses it. UNIX does not reuse process IDs until all other (32,000) IDs have been used. So while the same problem could theoretically occur, the odds are very low. Windows avoids this problem by keeping the free list in FIFO order, so that IDs are not normally reused for a long time. Another solution might have been to add sequence numbers, as was suggested for object handles to solve a similar problem with ordinary handle reuse. In general, applications should not identify a particular process by just the ID, but also by the creation timestamp. To do an operation on a process the ID is used to get a handle. The creation timestamp can be verified once the handle is obtained.
16. At most a few microseconds. It preempts the current thread immediately. It is just a question of how long it takes to run the dispatcher code to do the thread switch.

17. Having your priority lowered below the base priority could be used as a punishment for using excessive CPU time or other resources.
18. AutoBoost would need to boost the priority of a thread A only when a higher-priority thread B was waiting for a resource held by thread A. To do this AutoBoost would need to keep track of all the resources in the system, and what thread held each resource. When B blocks, AutoBoost would find A and temporarily raise the its priority to that of B.
19. For kernel-mode threads, their stacks and most of the data structures they need to access remain accessible even after changing address spaces, because they are shared across all processes. If user-mode threads were to switch address spaces, they would lose access to their stacks and other data. Finding a solution for user mode, such as changing stacks during a cross-process call, would allow threads to create processes as isolation boundaries and pass the CPU between them much as the CPU is passed from user mode to kernel mode. Cross-process procedure calls could be made without involving the scheduler to suspend the caller and wake the callee in the server process (and vice versa when returning).
20. One way is to increase the priority of important processes. A second way is to give important processes longer quanta.
21. Though working sets are not being trimmed, processes are still modifying pages backed by files. These files will be periodically pushed to disk. Also applications do explicit flushes of pages to disk to preserve results and maintain consistency in files. Both the file system and registry are active all the time, and they try to ensure that the volumes and hives will not lose data if the system were to crash.
22. For rotating disks, the most significant variable affecting performance is the number of seek operations required to perform I/O. It is not usually the number of I/Os that affect performance, but when many different chunks on disk must be accessed. Reducing the working set causes Windows to deal with the memory pages for a process a few at a time. Because Windows allocates space in the page file when the pages are written out rather than when they are allocated in memory, the writes of pages can generally be written to the same region of disk, reducing the number of seeks needed to write pages out to disk. The foreground processes belonging to modern applications stop running shortly after the user switches away or turns off the screen. This means that the working set required by the process suddenly goes to zero. But it does not mean that the process does not need these pages. It just does not need them until the user switches back to the application. At that time it is likely that most of these pages will suddenly all be needed again. This is different than the usual situation where pages are trimmed because the application was doing something else. In the case of modern apps, it is that the user (and thus the

system) are doing something else. Swapping does not just allow the pages from a particular application to efficiently be moved to disk together. More importantly it efficiently brings all the pages the application will need back into memory at one time instead of using demand paging. Demand paging would bring in only pages as they were needed. Initially the application would run a little, page fault, run a little more, page fault again, until the working set had been built back up. Each separate page fault may require a seek, if any intervening I/O activity caused the disk to seek away from the swap area.

23. On the x86, the self-map is made by taking two of the entries in the page directory and, rather than having them point at a page table page, they point at the page directory that contains them. These entries are set up in the page directory for a process when it is first initialized. Since the same PDE entries is always used, the virtual address of the self-map will be the same in every process.
24. To map 4 GB of address space requires 1M PTEs. If PTEs are only 4 bytes each, then the entire page table is only 4 MB.
25. Yes. The VADs are the way the memory manager keeps track of which addresses are in use and which are free. A VAD is needed for a reserved region to prevent a subsequent attempt to reserve or commit it from succeeding.
26. (1) is a policy decision about when and how to trim a working set. (2) and (3) are required. (4) is a policy decision about how aggressively to write dirty pages to the disk. (5) and (6) are required. (7) is not really a policy question or required; the system never has to zero pages, but if the system is otherwise idle, zeroing pages is always better than just executing the idle loop.
27. It is not moved at all. A page goes onto one of the lists only when it is not present in any working set. If it is still in one working set, it does not go on any of the free lists.
28. It cannot go on the modified list, since that contains pages that are still mapped in and could be faulted back. An unmapped page is not in that category. It certainly cannot go directly to the free list because those pages can be abandoned at will. A dirty page cannot be abandoned at will. Consequently, it must first be written back to the disk, then it can go on the free list.
29. A notification event means that all the waiting threads will be made runnable when the event is signaled. This could result in a lot of extra context switches as threads attempt to acquire the event only to block again. On a single-processor, if the lock hold time is much shorter than the quantum, then it is likely that by the time the other threads get to run, the lock will have been released by the previous thread to acquire it. But with short hold times the likelihood of very many threads waiting for the lock is much lower anyway. With a multiprocessor, even short hold times can result in unnecessary context switches

because the threads that become unblocked may all be scheduled to run immediately on different processors but immediately block, as only one thread will succeed in acquiring the lock.

30. Examples of programs that host DLLs for third parties include Web servers, browsers, and document editors. The hosted DLLs may be of questionable quality and may destabilize the hosting program. By creating a copy of the hosted process to run the DLL, the original process can be insulated against bugs or (if using some form of process sandboxing) attacks.
31. There are two records. The fields are as follows. The values before the colon are the header fields:
- Record 1 = 0, 8: (3, 50), (1, 22), (3, 24), (2, 53)  
Record 2 = 10, 10: (1, 60)
32. The fact that block 66 is contiguous with an existing run does not help, as the blocks are not in logical file order. In other words, using block 66 as the new block is no better than using block 90. The entries in the MFT are:
- 0, 8: (4, 20), (2, 64), (3, 80), (1, 66)
33. It is an accident. The 16 blocks apparently compressed to 8 blocks. It could have been 9 or 11 just as easily.
34. All except the user SID could be removed without affecting the strength of the security.
35. An attacker generally has access to one computing environment and uses it to try and gain control of a different, protected computing environment. There are many examples: using one system to connect to a service on another system that does not have an adequate firewall; running Javascript inside a browser and getting control of the browser; running as an ordinary user and getting control of the kernel.

Starting with a vulnerability, such as an overflow that somehow allows the attacker to modify arbitrary locations in memory, what do they do with that? How do they find the memory to modify? If stacks, or virtual function tables in the heap, are at the same address on every machine, they can just modify return addresses and function pointers they find there to ultimately get the program to jump where they want. Where do they want the program to go? If it were not for no-execute protection, they could just jump into instructions masquerading as input data. Without that, they have to choose instructions that are already executable and jump to those. If they can find a series of useful instruction sequences, usually ending in a return (i.e. gadgets), and they can build up data on the stack that can be interpreted as a series of stackframes, they can execute fairly interesting programs using this technique.

The relevant point is that at two points in the above description the attacker

needs to have a good idea where data and code are located in the address space she is attacking. ASLR scrambles address spaces so it is not so likely the attacker can just guess with much likelihood of success. But information leaks mean he may not have to.

An information leak is a vulnerability where the system will inappropriately make information available that can be used to successfully guess where something useful like a stack or DLL lives in the address space of a particular system or process.

Source of information leaks can include remote service APIs, the implementation of languages hosted in other programs, and system calls—including the implementation of I/O controls in device drivers.

- 36.** No. When a DLL is loaded it gets to run code inside the process. This code can access all the memory of the process and can use any of the handles, and all of the credentials in the process' default token. The only thing impersonation does is allow the process to acquire new handles using the credentials of the client that it is impersonating. Allowing ANY untrusted code to execute within the process means that any of the credentials accessible to that process now or in the future can be misused. This is something to consider next time you use any browser configured to download code from untrusted sites.
- 37.** The scheduler will always try to put the thread onto its ideal processor, or failing that, back onto a processor in the same node. Thus, even if the thread is currently running on a processor in a different node, at the end of the page fault it may indeed be scheduled back onto its preferred node. If all processors on that node continue to be busy, it may run in a different node and pay the penalty for slower access to the page that just got faulted in. But it is more likely to end up back on its preferred node sooner or later, and will thus benefit from this page placement in the long run.
- 38.** Metadata operations on the file system, like switching recovery files, could create inopportune moments where crashes would result in no recovery file. Having the opportunity to flush internal data buffers allows the application to create a consistent state that is easier to recover from. Allowing the application to finish complex operations would simplify the amount of logic needed when recovering.

In general allowing an application to prepare for the snapshot of the volume allows it to reduce the amount of transient state that must be dealt with when recovering later. Crash recovery is generally difficult to test because it is so hard to evaluate all the possible combinations of race conditions that are possible. Theoretically it is possible to design software that can deal with all sorts of recovery and failure scenarios, but it is very hard. One example of software that faces these challenges is the NTFS file system itself. A lot of design, analysis, and testing efforts have been invested to make it highly likely that NTFS can recover from a crash without having to run *chkdsk* (the Windows' e-

quivalent of UNIX's *fsck*). On very large enterprise file systems a *chkdsk* to recover after a crash can take hours or days.

39. The last page in a memory-mapped file must contain zeros after the last valid data. Otherwise if an I/O driver for a disk overwrote only part of the page with data from the disk, the previous data in the rest of the page would be exposed. This is not a problem for earlier pages in the file because the system will read a complete page from the file before giving access to user-mode code. When stack pages are allocated as the stack grows, these must contain all zeros, not whatever random data might have been left there by the previous tenant.
40. When an individual server is patched, the system administrator has a lot of discretion about scheduling the updates for a convenient time. If the root operating system needs to be rebooted to install updates, the guest systems do not need to reboot but must have all the memory they are using saved to disk before the root operating system reboots and restored afterward. This involves a very large amount of I/O and can take minutes though the root operating system reboot takes only seconds. During these minutes the guest operating system will be unavailable.

One solution is to avoid doing the I/O by not modifying or clearing the memory used by the guest systems when the root operating system reboots. Rather than save and restore all of memory to disk, the root operating system can just leave it in RAM and write to disk only the information needed to find the guest's memory after the reboot.

**SOLUTIONS TO CHAPTER 12 PROBLEMS**

1. Improvements in computer hardware have been largely due to smaller transistors. Some factors that can limit this are: (a) the wave properties of light may limit conventional photolithographic techniques for producing integrated circuits, (b) the mobility of individual atoms in solids can lead to degradation of the properties of very thin layers of semiconductors, insulators, and conductors, and (c) background radioactivity can disrupt molecular bonds or affect very small stored charges. There are certainly others.
2. For highly interactive programs, the event model may be better. Of these, only (b) is interactive. Thus, (a) and (c) are algorithmic and (b) is event driven.
3. No. The difference relates more to the fact that DNS servers cache and are organized hierarchically. The paths could easily have been given in the top-down order, but the convention of doing it backward is now well established.
4. Possibly stat is redundant. It could be achieved by a combination of open, fstat, and close. It would be very difficult to simulate any of the others.
5. If drivers are put below the threads, then drivers cannot be independent threads in the style of MINIX 3. They have to run as part of some other thread, more in the UNIX style.
6. It is possible. What is needed is a user-level process, the semaphore server that manages semaphores. To create a semaphore, a user sends it a message asking for a new semaphore. To use it, the user process passes the identity of the semaphore to other processes. They can then send messages to the semaphore server asking for an operation. If the operation blocks, no reply is sent back, thus blocking the caller.
7. The pattern is 8 msec of user code, then 2 msec of system code. With the optimization, each cycle is now 8 msec of user code and 1 msec of system code. Thus the cycle is reduced from 10 msec to 9 msec. Multiplying by 1000 such cycles, a 10-sec program now takes 9 sec.
8. External names can be as long as needed and of variable length. Internal names are generally 32 or 64 bits and always fixed length. External names need not be unique. Two names can point to the same object, for example, links in the UNIX file system. Internal names must be unique. External names may be hierarchical. Internal names are generally indices into tables and thus form a flat namespace.
9. If the new table is 2× as big as the old one, it will not fill up quickly, reducing the number of times an upgraded table will be needed. On the other hand, so much space may not be needed, so it may waste memory. This is a classic time vs. space trade-off.



10. It would be risky to do that. Suppose that the PID was at the very last entry. In that case, exiting the loop would leave  $p$  pointing to the last entry. However, if the PID was not found,  $p$  might end up pointing to the last entry or to one beyond it, depending on the details of the compiled code, which optimizations were turned on, and so on. What might work with one compiler could fail with a different one. It is better to set a flag.
11. It could be done, but would not be a good idea. An IDE or SCSI driver is many pages long. Having conditional code so long makes the source code hard to follow. It would be better to put each one in a separate file and then use the *Makefile* to determine which one to include. Or at the very least, conditional compilation could be used to include one driver file or the other.
12. Yes. It makes the code slower. Also, more code means more bugs.
13. Not easily. Multiple invocations at the same time could interfere with one another. It might be possible if the static data were guarded by a mutex, but that would mean that a call to a simple procedure might unexpectedly block.
14. Yes. The code is replicated every time the macro is called. If it is called many times, the program will be much bigger. This is a typical of a time-space trade-off: a bigger, faster program instead of a smaller, slower program. However, in an extreme case, the larger program might not fit in the TLB, causing it to thrash and thus run slower.
15. Start by EXCLUSIVE-ORing the lower and upper 16 bits of the word together to form a 16-bit integer,  $s$ . For each bit, there are four cases: 00 (results in a 0), 01 (results in a 1), 10 (results in a 1), and 11 (results in a 0). Thus, if the number of 1s in  $s$  is odd, the parity is odd; otherwise it is even. Make a table with 65,536 entries, each containing one byte with the parity bit in it. The macro looks like this:  

```
#define parity(w) bits[(w & 0xFFFF) ^ ((w>>16) & 0xFFFF)]
```
16. No circumstances. The “compressed” color value would be as big as the original, and in addition, a huge color palette could be needed. It makes no sense at all.
17. The 8-bit-wide color palette contains 256 entries of 3 bytes each for a total of 768 bytes. The saving per pixel is 2 bytes. Thus, with more than 384 pixels, GIF wins. A 16-bit-wide color palette contains 65,536 entries of 3 bytes each, for 196,608 bytes. The saving here is 1 byte per pixel. Thus, with more than 196,608 pixels, the 16-bit compression wins. Assuming a 4:3 ratio, the break-even point is an image of  $512 \times 384$  pixels. For VGA ( $640 \times 480$ ), 16-bit color requires less data than true 24-bit color.

18. For a path that is in the path-name cache, it has no effect because the i-node is bypassed anyway. If it is not read, it does not matter if it is already in memory. For a path that is not in the name cache but involves a pinned i-node, then pinning does help since it eliminates a disk read.
19. Recording the date of last modification, the size, and possibly a calculated signature such as a checksum or CRC can help determine if it has changed since last referenced. A caveat: a remote server could provide false information about a file, and local regeneration of a calculated signature might be necessary.
20. The file could be given a version number or a checksum and this information stored along with the hint. Before accessing a remote file, a check would be made to make sure the version number or checksum still agreed with the current file.
21. A file system will typically try to write new data to the nearest available disk block following the last one used. If two files are being written simultaneously this can result in interleaving the data blocks on the disk, resulting in both files being fragmented and thus more difficult to read. This effect can be ameliorated by buffering data in memory to maximize the size of writes, or writing to temporary files and then copying each output to a permanent file when the program terminates.
22. Brooks was talking about large projects in which communication between the programmers slows everything down. That problem does not occur with a one-person project and so that productivity can be higher.
23. If a programmer can produce 1000 lines of code for a cost of \$100,000, a line of code costs \$100. Windows 8 consists of 50–100 million lines of code, which comes to \$5–10 billion. That seems like an awful lot. Probably Microsoft has managed to improve programmer productivity using better tools so that a programmer can produce thousands of lines of code per year. Also, large pieces of Windows 8 were taken unmodified from Windows 7, so the amount of new code in Windows 8 is only a fraction of its total size. On the other hand, Microsoft's annual revenue is around \$70 billion, so spending billions of dollars on Windows 8 is possible.
24. Suppose that memory costs \$10 per GB (check against current prices). Then a low-end machine with a 100-GB disk needs \$1000 worth of RAM for its disk. If the rest of the PC is \$500, the total cost comes to \$1500. This is too expensive for the low-end market.
25. An embedded system may run only one or a small number of programs. If all programs can be kept loaded into memory at all times, there might be no need for either a memory manager or a file system. Additionally, drivers would be needed only for a few I/O devices, and it might make more sense to write

the I/O drivers as library routines. Library routines might also be better compiled into individual programs, rather than into shared libraries, eliminating the need for shared libraries. Probably many other features could be eliminated in specific cases.

**SOLUTIONS TO APPENDIX A PROBLEMS**

1. XGA is  $1024 \times 768$ . With 24 bits/pixel and a frame rate of 25 frames/sec we get 471,859,200 bits/sec. This rate is too high for UltraWide SCSI, which can only go up to 320 Mbps.
2. Standard NTSC television is about  $640 \times 480$  pixels. At 8 bits/pixel and 30 frames/sec we get a bandwidth of 73 Mbps. It just barely makes it with one channel. Two channels would be too much.
3. From the table, HDTV is  $1280 \times 720$  versus  $640 \times 480$  for regular TV. It has three times as many pixels and thus needs three times the bandwidth. The reason it does not need four times as much bandwidth is that the aspect ratio of HDTV is different from conventional TV to match that of 35-mm film better.
4. For slow motion going forward, it is sufficient for each frame to be displayed two or more times in a row. No additional file is needed. To go backward slowly is as bad as going backward quickly, so an additional file is needed.
5. Audio is sampled at 16 bits per sample, 44,100 times/sec with two channels. This gives an uncompressed audio rate of 1,411,200 bits/sec or 176,400 bytes/sec. In 74 minutes, this adds up to 747 MB. This is the full capacity of the CD. It is not compressed at all. The reason data is limited to 650 MB is that better error correction is used for data since an error is more serious than for music. If even a factor of two compression had been used on audio CDs, the data would have been less than 374 MB and more than 74 minutes could be stored on a CD.
6. There are 32,768 possible magnitudes. For example, suppose the signal ranges from  $-32.768$  volts to  $+32.767$  volts and the value stored for each sample is the signal rounded off to the nearest number of millivolts, as a signed 16-bit integer. A signal of 16.0005 volts would have to be recorded as either 16,000 or as 16,001. The percent error here is 1/320 percent. However, suppose the signal is 0.0005 volts. This is recorded at either 0 or 1. In the latter case, the error is 50%. Thus quantization noise affects low amplitudes more than high amplitudes. Flute concertos will be hit harder than rock and roll due to their lower amplitudes.
7. A volume compression/expansion scheme could be implemented as follows. One bit of the output is reserved to signal that the recorded signal is expanded. The remaining 15 bits are used for the signal. When the high-order 5 bits of the 20-bit signal are not 00000, the expansion bit is 0 and the other 15 bits contain the high-order 15 bits of the sampled data. When the high-order 5 bits of the signal are 00000, the expansion bit is turned on and the 20-bit amplitude signal is shifted left 5 bits. At the listener's end the reverse process takes place. This scheme increases quantization noise slightly for loud signals (due to a 15-bit signal instead of a 16-bit signal), but decreases it for quiet signals,

when the effect of quantization is most noticeable. A major disadvantage is that this is not a standard and would not work with existing CD players, but it could work for online music played with a special plugin that used this scheme on both ends. A more sophisticated version could use 2 bits to denote four different expansion regimes for different signal levels.

8. PAL has more scan lines and more spatial resolution than NTSC. It has 625 vertical lines versus 525 for NTSC. It also has more pixels per line. These result in a sharper image, and use the extra bandwidth. On the other hand, NTSC has more frames per second, so it is better for catching rapid action. Neither one is “better” than the other in this sense. Different trade-offs have been made: better resolution in time versus better resolution in space. All of this is completely independent of the color encoding schemes used.
9. The difference does not cause problems at all. The DCT algorithm is used to encode I-frames in a JPEG-like scheme. The macroblocks are used in P-frames to locate macroblocks that appeared in previous frames. The two things have nothing to do with each other and do not conflict.
10. No they do not. The motion compensation algorithm will find each macroblock in the previous frame at some offset from its current location. By encoding the fact that the current macroblock should be taken from the previous frame at a position  $(\Delta x, \Delta y)$  from the current one, it is not necessary to transmit the block itself again.
11. The processes supporting the three video streams already use 0.808 of the CPU time, so there are 192 msec per second left over for audio. Audio process *A* runs 33.333 times/sec, audio process *B* runs 25 times/sec, and audio process *C* runs 20 times/sec, for a total of 78.333 runs/sec. These 78.333 runs may use 192 msec, so each run can use  $192/78.333$  or 2.45 msec.
12. The first process uses 0.400 of the CPU. The second one uses 0.375 of the CPU. Together they use 0.775. The RMS limit for two processes is  $2 \times (2^{0.5} - 1)$ , which is 0.828, so RMS is guaranteed to work.
13. PAL generates a frame every 40 ms. Assume that all streams are activated at the same instant (time 0). The upper bound on utilization, as proved by Liu and Layland, need not apply here. It is possible to schedule the first frame of stream 0 at 0 ms, of stream 1 at 5 ms, and so on, up to a maximum of 8 streams. Then, at time 40 ms, the second frame of stream 0 is scheduled, and so on.
14. Since  $0.65 < \ln 2$ , RMS can always schedule the movies, no matter how many there are. Thus RMS does not limit the number of movies.

15. The sequence starting at  $t = 150$  is  $A6, B5, C4, A7, B6, A5$ , and  $C5$ . When  $C5$  ends at  $t = 235$  there is no work to do until  $t = 240$  when  $A$  and  $B$  become ready, so the system goes idle for 5 msec. The choice of running  $B5$  before  $C4$  is arbitrary. The other way is also allowed.
16. A DVD reader is OK for home viewing, but the high seek time of current optical recording systems limits their usefulness to providing a single stream of data. DVD drives cannot support multiple streams with different start times or VCR-like control functions such as pause, rewind, and fast forward for different users. With current technology the data would have to be buffered in an extremely large memory. Hard disks are simply better.
17. If the worst-case wait is 6 min, a new stream must start every 6 min. For a 180-min movie, 30 streams are needed.
18. The data rate is 0.5 MB/sec. One minute of video uses 30 MB. To go forward or backward 1 min each requires 60 MB.
19. We have  $2 \times \Delta T \times 2 \times 10^6 \leq 150 \times 2^{20} \times 8$ . So  $\Delta T \leq 105$  sec.
20. HDTV does not make any difference. There are still 216,000 frames in the movie. The wastage for each frame is about half a disk block, or 0.5 KB. For the whole movie, this loss is 108 KB.
21. There is some loss for each frame. The more frames you have, the more loss you have. NTSC has a higher frame rate, so it has slightly more loss. But given the numbers involved, this loss is not a significant fraction of the total disk space.
22. The main effect of HDTV is larger frames. Large frames tend to make the disadvantage of small blocks less serious because large frames can be read in efficiently. Thus the disk performance argument in favor of large blocks diminishes. In addition, if frames are not split over blocks (as they are not here), having I-frames that are a substantial fraction of a block is a serious problem. It may often occur that a block is partly full and a large I-frame appears next, wasting a large amount of space in the current block. On the whole, going to HDTV favors the small block model.
23. On an average, 1 KB can be wasted per block. A 2-hour PAL movie requires 180,000 frames. Thus, the average total disk space wastage for this movie is 180,000 KB.
24. Each frame index block can store up to  $2048/8 = 512$  entries. Since each frame can be as large as  $255 \times 2 \text{ KB} = 510 \text{ KB}$ , the largest file size can be:  $512 \times 510 \text{ KB} = 261,120 \text{ KB}$ .

25. From the Web, the length is 226 minutes. Hence, the total number of frames is  $25 \times 226 \times 60 = 339,000$  frames. Since each frame index block can refer to 512 frames, the number of needed index blocks is  $339000/512$  or 663.
26. The buffer is big enough if the number of I-frames is 4 or less. The probability of getting exactly  $k$  I-frames is  $C(24, k)I^k B^{24-k}$ , where  $I$  is 0.1 and  $B$  is 0.9. The probabilities of getting exactly 0, 1, 2, 3, and 4 I-frames are 0.0798, 0.213, 0.272, 0.221, and 0.129, respectively. The sum of these is 0.915. This means there is a 0.085 or 8.5% chance of failure. This is too large to accept.
27. The buffer should be large enough to hold five frames, since only 5% of the tracks have more than five I-frames.
28. Regardless of format, the number of concurrent streams is  $3 \times 60/15 = 12$ .
29. One alternative is to have a separate file for each language. This alternative minimizes RAM use but wastes large amounts of disk space. If disk space is cheap and the goal is to support as many possible streams at once, this approach is attractive. Another alternative is to store the audio track for each language separately and do an extra seek per frame to fetch the audio. This scheme makes efficient use of disk space, but introduces extra seeks and thus slows down performance.
30. The normalization constant,  $C$ , is 0.36794, so the probabilities are 0.368, 0.184, 0.123, 0.092, 0.074, 0.061, 0.053, and 0.046.
31. A 14-GB disk holds  $14 \times 2^{30}$  or 15,032,385,536 bytes. If these are uniformly split over 1000 cylinders, each cylinder holds 15,032,385, which is just enough for a 30-sec video clip. Thus each clip occupies one cylinder. The question is then what fraction of the total weight is represented by the top 10 clips out of 1000. Adding up 1,  $1/2$ , ...  $1/10$ , we get 2.92895. Multiplying this by 0.134 we get 0.392, so the arm spends nearly 40% of its time within the middle 10 cylinders.
32. For four items, Zipf's law yields probabilities of 0.48, 0.24, 0.16, and 0.12. Ratios of these probabilities also describe the relative utilization of the drives for Fig. 12-0(a). For the other three striping arrangements all drives will be used equally, assuming that everybody who pays for a movie watches it through to the end. The result at a particular time might be different, however. If everybody in the town wants to start watching a movie at 8 A.M. the arrangement of Fig. 12-0(b) would initially hit the first disk hardest, then the next disk 15 minutes later, etc. The arrangements of Fig. 12-0(c) or (d) would not be affected this way.
33. PAL runs at 25 frames/sec, so the two users are off by 150 frames. To merge them in 3 min means closing the gap by 50 frames/min. One goes 25 frames/min faster and one goes 25 frames/min slower. The normal frame rate

is 1500 frames/min, so the speed up or down is 25/1500 or 1/60, which is about 1.67%.

34. For NTSC, with 30 frames/sec, a round is 33.3 msec. The disk rotates 180 times/sec, so the average rotational latency is half a rotation or 2.8 msec. MPEG-2 runs at about 500,000 bytes/sec or about 16,667 bytes/frame. At 320 MB/sec, the transfer time for a frame is about 51  $\mu$ sec. Thus the seek, rotational latency, and transfer times add up to about 5.8 msec. Five streams thus eat up 29 msec of the 33.3 msec, which is the maximum.
35. The average seek time goes from 3.0 msec to 2.4 msec, so the time per operation is reduced to 5.2 msec. This adds one more stream, making six in all.
36. The requests are ordered, based on cylinder requested, as: (40, 210), (32, 300), (34, 310), (36, 500). The end time of each request is, in order: 21 msec, 27 msec, 33 msec, 39 msec. Hence, the system will miss the last deadline, if the above algorithm is used.
37. Six streams. Striping is useless. Each disk operation still takes 5.2 msec to get the arm over the data. Whether the transfer time is 51  $\mu$ sec or 13  $\mu$ sec does not make much difference.
38. For the first batch of five requests, the critical one is for cylinder 676, fourth in the list, but with a deadline of  $t = 712$  msec. So each request must be served in 3 msec or less in order for the fourth one to be done at  $t = 712$  msec.