MODERN OPERATING SYSTEMS

SECOND EDITION

PROBLEM SOLUTIONS

ANDREW S. TANENBAUM

Vrije Universiteit
Amsterdam, The Netherlands

PRENTICE HALL

UPPER SADDLE RIVER, NJ 07458

SOLUTIONS TO CHAPTER 1 PROBLEMS

1. An operating system must provide the users with an extended (i.e., virtual) machine, and it must manage the I/O devices and other system resources.

2. Multiprogramming is the rapid switching of the CPU between multiple processes in memory. It is commonly used to keep the CPU busy while one or more processes are doing I/O.

3. Input spooling is the technique of reading in jobs, for example, from cards, onto the disk, so that when the currently executing processes are finished, there will be work waiting for the CPU. Output spooling consists of first copying printable files to disk before printing them, rather than printing directly as the output is generated. Input spooling on a personal computer is not very likely, but output spooling is.

4. 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 percent 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).

5. Second generation computers did not have the necessary hardware to protect the operating system from malicious user programs.

6. It is still alive. For example, Intel makes Pentium I, II, and III, and 4 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.

7. A 25 × 80 character monochrome text screen requires a 2000-byte buffer. The 1024 × 768 pixel 24-bit color bitmap requires 2,359,296 bytes. In 1980 these two options would have cost $10 and $11,520, respectively. For current prices, check on how much RAM currently costs, probably less than $1/MB.

8. Choices (a), (c), and (d) should be restricted to kernel mode.

9. Personal computer systems are always interactive, often with only a single user. Mainframe systems nearly always emphasize batch or timesharing with many users. Protection is much more of an issue on mainframe systems, as is efficient use of all resources.

10. 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 instructions pops out the end of the pipeline.

11. The manuscript contains 80 × 50 × 700 = 2.8 million characters. This is, of course, impossible to fit into the registers of any currently available CPU and is too big for a 1-MB cache, but if such hardware were available, the manuscript could be scanned in 2.8 msec from the registers or 5.8 msec from the cache. There are approximately 2700 1024-byte blocks of data, so scanning from the disk would require about 27 seconds, and from tape 2 minutes 7 seconds. Of course, these times are just to read the data. Processing and rewriting the data would increase the time.

12. Logically, it does not matter if the limit register uses a virtual address or a physical address. However, the performance of the former is better. If virtual addresses are used, the addition of the virtual address and the base register can start simultaneously with the comparison and then can run in parallel. If physical addresses are used, the comparison cannot start until the addition is complete, increasing the access time.

13. 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.

14. A trap is caused by the program and is synchronous with it. If the program is run again and again, the trap will always occur at exactly the same position in the instruction stream. An interrupt is caused by an external event and its timing is not reproducible.

15. Base = 40,000 and limit = 10,000. An answer of limit = 50,000 is incorrect for the way the system was described in this book. It could have been implemented that way, but doing so would have required waiting until the address + base calculation was completed before starting the limit check, thus slowing down the computer.

16. The process table is needed to store the state of a process that is currently suspended, either ready or blocked. It is not needed in a single process system because the single process is never suspended.

17. 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 checked or repaired.

18. 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.

19. 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*.

20. It contains the bytes: 1, 5, 9, 2.

21. 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.

22. 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.

23. Yes it can, especially if the kernel is a message-passing system.

24. 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.

25. Several UNIX calls have no counterpart in the Win32 API:

Link: a Win32 program cannot refer to a file by an alternate 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 programmers have to assume that every user can access every file.

Kill: Windows programmers cannot kill a misbehaving program that is not cooperating.

26. The conversions are straightforward:

(a) A micro year is $10^{-6} \times 365 \times 24 \times 3600 = 31.536$ sec.
(b) 1000 meters or 1 km.
(c) There are 2^{40} bytes, which is 1,099,511,627,776 bytes.
(d) It is 6×10^{24} kg.

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 one 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. 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.

6. 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. Timesharing threads is no different than timesharing processes, so each thread needs its own register save area.

7. No. If a single-threaded process is blocked on the keyboard, it cannot fork.

8. 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.

9. 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.

10. User-level threads cannot be preempted by the clock uless the whole process' quantum has been used up. 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.

11. In the single-threaded case, the cache hits take 15 msec and cache misses take 90 msec. The weighted average is $2/3 \times 15 + 1/3 \times 90$. Thus the mean request takes 40 msec and the server can do 25 per second. For a multithreaded server, all the waiting for the disk is overlapped, so every request takes 15 msec, and the server can handle 66 2/3 requests per second.

12. 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.

13. 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?

14. 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.

15. 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 can develop. This is not an attractive way to build a threads package.

16. 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.

17. 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.

18. A race condition is a situation in which two (or more) processes are about to perform some action. Depending on the exact timing, one or the other goes first. If one of the processes goes first, everything works, but if another one goes first, a fatal error occurs.

19. 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 one to the variable, we have a race condition.

20. Yes, it still works, but it still is busy waiting, of course.

21. 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.

22. Yes it can. The memory word is used as a flag, with 0 meaning that no one is using the critical variables and 1 meaning that someone is using them. Put a 1 in the register, and swap the memory word and the register. If the register contains a 0 after the swap, access has been granted. If it contains a 1, access has been denied. When a process is done, it stores a 0 in the flag in memory.

23. 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 then is removed from the list of blocked processes and made runnable. When all these operations have been completed, interrupts can be enabled again.

24. 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.

25. 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.

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. 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.

28. It is very expensive to implement. Each time any variable that appears in a predicate on which some process is waiting changes, the runtime 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.

29. The employees communicate by passing messages: orders, food, and bags in this case. In UNIX terms, the four processes are connected by pipes.

30. It does not lead to race conditions (nothing is ever lost), but it is effectively busy waiting.

31. If a philosopher blocks, neighbors can later see that he is hungry by checking his state, in *test*, so he can be awakened when the forks are available.

32. 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. Similary, if philosopher 4 finished eating, philosopher 3 would not be started. Nothing would start him.

33. 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.

34. It will need *nT* sec.

35. 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.

36. In simple cases it may be possible to determine whether 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 problem 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 total time to complete execution of the program. This is, of course, most meaningful on a system where you are the only user.

37. For multiple processes in a pipeline, the common parent could pass to the operating system information about the flow of data. With this information the OS could, for instance, determine which process could supply output to a process blocking on a call for input.

38. 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 percent. Finally, for (e), as $Q \rightarrow 0$ the efficiency goes to 0.

39. 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.

40. 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.

41. 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.

42. A check could be made to see if the program was expecting input and did anything with it. A program that was not expecting input and did not process it would not get any special priority boost.

43. The sequence of predictions is 40, 30, 35, and now 25.

44. 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.

45. Two-level scheduling is needed when memory is too small to hold all the ready processes. Some set of them is put into memory, and a choice is made from that set. From time to time, the set of in-core processes is adjusted. This algorithm is easy to implement and reasonably efficient, certainly a lot better than say, round robin without regard to whether a process was in memory or not.

46. 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.

47. 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. 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. If the printer starts to print a file before the entire file has been received (this is often allowed to speed response), the disk may fill with other requests that can't be printed until the first file is done, but which use up disk space needed to receive the file currently being printed. If the spooler does not start to print a file until the entire file has been spooled it can reject a request that is too big. Starting to print a file is equivalent to reserving the printer; if the reservation is deferred until it is known that the entire file can be received, a deadlock of the entire system can be avoided. The user with the file that won't fit is still deadlocked of course, and must go to another facility that permits printing bigger files.

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. 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. Arcs from squares to squares or from circles to circles also violate the rules.

6. 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.

7. Neither change leads to deadlock. There is no circular wait in either case.

8. Voluntary relinquishment of a resource is most similar to recovery through preemption. The essential difference is that computer processes are not expected to solve such problems on their own. Preemption is analogous to the operator or the operating system acting as a policeman, overriding the normal rules individual processes obey.

9. 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.

10. If the system had two or more CPUs, two or more processes could run in parallel, leading to diagonal trajectories.

11. Yes. Do the whole thing in three dimensions. The *z*-axis measures the number of instructions executed by the third process.

12. 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.

13. A request from *D* is unsafe, but one from *C* is safe.

14. 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.

15. 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.

16. 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 $r \geq p(m - 1) + 1$.

17. No. *D* can still finish. When it finishes, it returns enough resources to allow *E* (or *A*) to finish, and so on.

18. With three processes, each one can have two drives. With four processes, the distribution of drives will be (2, 2, 1, 1), allowing the first two processes to finish. With five processes, the distribution will be (2, 1, 1, 1, 1), which still allows the first one to finish. With six, each holding one tape drive and wanting another, we have a deadlock. Thus for $n < 6$ the system is deadlock-free.

19. 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 or *mn* steps. Repeating the algorithm for all *n* processes means that the number of steps is then mn^2.

20. The needs matix 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.

21. 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 deadlock are now fulfilled.

22. 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:

a b c	deadlock free
a c b	deadlock free
b a c	possible deadlock
b c a	possible deadlock
c a b	possible deadlock
c b a	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.

23. Two-phase locking eliminates deadlocks, but introduces potential starvation. A process has to keep trying and failing to acquire all of its records. There is no upper bound on how long it may take.

24. 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.

25. 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).

26. 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.

27. If both programs ask for Woofer first, the computers will starve with the endless sequence: request Woofer, cancel request, request Woofer, cancel request, etc. 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 4 PROBLEMS

1. The chance that all four processes are idle is 1/16, so the CPU idle time is 1/16.

2. If each job has 50% I/O wait, then it will take 20 minutes to complete in the absence of competition. If run sequentially, the second one will finish 40 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 10 minutes of CPU time, a job must run for 10/0.375 minutes or about 26.67 minutes. Thus running sequentially the jobs finish after 40 minutes, but running in parallel they finish after 26.67 minutes.

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 10 nsec, so reading 1 byte takes 2.5 nsec and writing it takes another 2.5 nsec, for a total of 5 nsec per byte compacted. This is a rate of 200,000,000 bytes/sec. To copy 128 MB (2^{27} bytes, which is about 1.34×10^8 bytes), the computer needs $2^{27}/200,000,000$ sec, which is about 671 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. The bitmap needs 1 bit per allocation unit. With $2^{27}/n$ allocation units, this is $2^{24}/n$ bytes. The linked list has $2^{27}/2^{16}$ or 2^{11} nodes, each of 8 bytes for a total of 2^{14} bytes. For small *n*, the linked list is better. For large *n*, the bitmap is better. The crossover point can be calculated by equating these two formulas and solving for *n*. The result is 1 KB. For *n* smaller than 1 KB, a linked list is better. For *n* larger than 1 KB, a bitmap is better. Of course, the assumption of segments and holes alternating every 64 KB is very unrealistic. Also, we need $n <= 64$ KB if the segments and holes are 64 KB.

5. First fit takes 20 KB, 10 KB, 18 KB. Best fit takes 12 KB, 10 KB, and 9 KB. Worst fit takes 20 KB, 18 KB, and 15 KB. Next fit takes 20 KB, 18 KB, and 9 KB.

6. 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 16-bit word can generate virtual addresses up to 64K, regardless of whether the machine has more or less memory than 64 KB.

7. 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), (7, 2656).

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

9. 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.

10. 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.

11. A page fault every *k* instructions adds an extra overhead of n/k μsec to the average, so the average instruction takes $10 + n/k$ nsec.

12. 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 percent of the time is spent loading page tables.

13. 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.

14. The number of pages depends on the total number of bits in *a*, *b*, and *c* combined. How they are split among the fields does not matter.

15. 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.

16. The code and reference string is 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.

17. 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.

18. The *R* bit is never needed in the TLB. The mere presence of a page there means the page has been referenced; otherwise it would not be there. Thus the bit is completely redundant. When the entry is written back to memory, however, the *R* bit in the memory page table is set.

19. 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.

20. 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).

21. 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 a 65,536 table slots will give a mean chain length of 0.5, which ensures fast lookup.

22. 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 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.

23. The page frames for FIFO are as follows:

x0172333300 xx017222233 xxx01777722 xxxx0111177

The page frames for LRU are as follows:

x0172327103 xx017232710 xxx01773271 xxxx0111327

FIFO yields 6 page faults; LRU yields 7.

24. The first page with a 0 bit will be chosen, in this case *D*.

25. The counters are
Page 0: 0110110
Page 1: 01001001
Page 2: 00110111
Page 3: 10001011

26. The first page with $R = 0$ and age $> \tau$ will be chosen. Since the scan starts at the bottom, the very first page (1620) gets evicted.

27. The age of the page is $2204 - 1213 = 991$. If $\tau = 400$, it is definitely out of the working set and it was not recently referenced so it will be evicted. The $\tau = 1000$ the situation is different. Now the page falls within the working set (barely), so it is not removed.

28. The seek plus rotational latency is 20 msec. For 2-KB pages, the transfer time is 1.25 msec, for a total of 21.25 msec. Loading 32 of these pages will take 680 msec. For 4-KB pages, the transfer time is doubled to 2.5 msec, so the total time per page is 22.50 msec. Loading 16 of these pages takes 360 msec.

29. NRU removes page 2. FIFO removes page 3. LRU removes page 1. Second chance removes page 2.

30. 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.

31. 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.

32. If pages can be shared, yes. For example, if two users of a timesharing system are running the same editor at the same time and the program text is shared rather than copied, some of those pages may be in each user's working set at the same time.

33. It is possible. Assuming that segmentation is not present, the protection information must be in the page table. If each process has its own page table, each one also has its own protection bits. They could be different.

34. 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 memory page faults, and only 15 sec of page fault overhead, so the total run time will be 45 sec.

35. 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.

36. 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 Pentium).

37. 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.

38. 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.

SOLUTIONS TO CHAPTER 5 PROBLEMS

1. In the figure, we see a controller with two devices. The reason that a single controller is expected to handle multiple devices is to 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 bus and disk both run at 16.7 MB/sec, so neither the disk nor the bus comes anywhere near saturation.

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. Each bus transaction has a request and a response, each taking 100 nsec, or 200 nsec per bus transaction. This gives 5 million bus transactions/sec. If each one is good for 4 bytes, the bus has to handle 20 MB/sec. The fact that these transactions may be sprayed over four I/O devices in round-robin fashion is irrelevant. A bus transaction takes 200 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.

5. An interrupt requires pushing 34 words onto the stack. Returning from the interrupt requires fetching 34 words from the stack. This overhead alone is 680 nsec. Thus the maximum number of interrupts per second is no more than about 1.47 million, assuming no work for each interrupt.

6. 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.

7. 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.

8. The printer prints $50 \times 80 \times 6 = 24,000$ characters/min, which is 400 characters/sec. Each character uses 50 μsec 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.

9. Device independence means that files and devices are accessed the same way, independent of their physical nature. Systems that have one set of calls for writing on a file, but a different set of calls for writing on the console (terminal) do not exhibit device independence.

10. (a) Device driver.
(b) Device driver.
(c) Device-independent software.
(d) User-level software.

11. Based on the data in Fig. 5-17, for the floppy disk example, there are $9 \times 512 \times 8 = 36864$ bits per track. At 200 msec per rotation the bit rate is 184,320 bits/sec. The hard disk has an average of 281 sectors per track, so there are $281 \times 512 \times 8 = 1,150,976$ bits/track on the average. A rotation time of 8.33 msec corresponds to 120 rotation/sec (7200 rpm), so in one second the disk can transfer $120 \times 1,150,976$ bits. This is about 138 million bits/sec. The data rate of the floppy disk is roughly three times that of a 56-Kbps modem. The data rate of the hard disk is about 38% faster than Fast Ethernet. However, these calculations underestimate the actual maximum data rates, because for every 512 bytes of data on a disk there are also a number of bytes of formatting information, to identify the track and sector, as well as a gap between sectors, necessary to prevent sectors from overlapping if there are slight speed variations.

12. 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 percent of the nominal 10 megabit/sec network capacity. (If we include protocol overhead, the figures get even worse.)

13. 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.

14. The disk rotates at 120 rpm, 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.

15. As we saw in the previous problem, the sector time is 1/24 msec. This means that the disk can read 24,000 sectors/sec. Since each sector contains 512 bytes, the data rate is 12,288,000 bytes/sec. This rate is 11.7 MB/sec.

16. 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.

17. 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 $1 - (1 - p)^k - kp(1 - p)^{k-1}$.

18. 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.

19. 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.

20. The rotation time is 200 msec. Reading all the sectors in order requires 1/2 rotation to get sector 0 and 2.75 rotations to get the data (after sector 7 is read, the transfer is finished). Thus 3.25 rotations are needed for 650 msec. Reading 4K in 650 msec is 6302 bytes/sec. For a noninterleaved disk, it takes 300 msec to read 4K, which is 13,653 bytes/sec. Interleaving reduces the capacity to 6302/13653 or 0.46 of its former capacity.

21. 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, than no additional cylinder skew is needed. If it cannot, then additional cylinder skew is needed to avoid missing a sector after a seek.

22. The drive capacity and transfer rates are doubled. The seek time and average rotational delay are the same.

23. 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.

24. (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.

25. 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.

26. 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.

27. Two msec 60 times a second is 120 msec/sec, or 12 percent of the CPU

28. 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.

29. Each line requires $3200 \times 8 = 25{,}600$ samples/sec. At 1 μsec per sample, each line takes up 25.6 msec of the processor's time each second. With 39 lines, the processor is busy for $39 \times 25.6 = 998.4$ msec each second, which gives the capacity of the card as 39 lines.

30. After a character is written to an RS232 terminal, it takes a (relatively) long time before it is printed. Waiting would be wasteful, so interrupts are used. With memory-mapped terminals, the character is accepted instantly, so interrupts make no sense.

31. At 56 Kbps, we have 5600 interrupts/sec, which is 560 msec. This is 56% of the CPU.

32. Scrolling the window requires copying 59 lines of 80 characters or 4720 characters. Copying 1 character (16 bytes) takes 800 nsec, so the whole window takes 3.776 msec. Writing 80 characters to the screen takes 400 nsec, so scrolling and displaying a new line take 4.176 msec. This gives about 239.5 lines/sec.

33. 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.

34. 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

35. The embedded processor inside the terminal has to move all the characters up one line by just copying them. Viewed from the inside, the terminal is memory mapped. There is no easy way to avoid this organization unless special hardware is available.

36. The 25 lines of characters, each 8 pixels high, requires 200 scans to draw. There are 60 screens a second, or 12,000 scans/sec. At 63.6 μsec/scan, the beam is moving horizontally 763 msec per second, leaving 237 msec for writing in the video RAM. Thus the video RAM is available 23.7% of the time.

37. The maximum rate the mouse can move is 200 mm/sec, which is 2000 mickeys/sec. If each report is 3 byte, the output rate is 6000 bytes/sec.

38. 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 can might be represented by the (R, G, B) value (0, 0, 1). The next one might be represented by (0, 0, 2), etc. 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).

39. (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 μsec. This gives an output rate of about 8681 chars/sec.

40. Rewriting the text screen requires copying 2000 bytes, which can be done in 20 μseconds. Rewriting the graphics screen requires copying $1024 \times 768 \times 3 = 2{,}359{,}296$ bytes, or about 23.6 msec.

41. 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.

42. 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.

43. 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.

44. 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.

45. 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%.

46. 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. 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. The operating system cares about record length when files can be structured as records with keys at a specific position within each record and it is possible to ask for a record with a given key. In that case, the system has to know how big the records are so it can search each one for the key.

5. 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.

6. No. If you want to read the file again, just randomly access byte 0.

7. 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 full, the copy might fail.

8. 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.

9. 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.

10. 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.

11. 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*.

12. 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.

13. It takes 9 msec to start the transfer. To read 2^{13} bytes at a transfer rate of 2^{23} bytes/sec requires 2^{-10} sec (977 msec), for a total of 9.977 msec. Writing it back takes another 9.977 msec. Thus copying a file takes 19.954 msec. To compact half of a 16-GB disk would involve copying 8 GB of storage, which is 2^{20} files. At 19.954 msec per file, this takes 20,923 sec, which is 5.8 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. It finds the address of the first block in the directory entry. It then follows the chain of block pointers in the FAT until it has located the block it needs. It then remembers this block number for the next read system call.

17. The indirect block can hold 256 disk addresses. Together with the 10 direct disk addresses, the maximum file has 266 blocks. Since each block is 1 KB, the largest file is 266 KB.

18. 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.

19. 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.

20. 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.

21. 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 percent or less of the disk is free.

22. 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

23. 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.

24. 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.

25. 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.

26. In (a) and (b), 21 would not be marked. In (c), there would be no change. In (d), 21 would not be marked.

27. 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.

28. 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.

29. The time needed is $h + 40 \times (1 - h)$. The plot is just a straight line.

30. The time per block is built up of three components: seek time, rotational latency, and transfer time. In all cases the rotational latency plus transfer time is the same, 125 msec. Only the seek time differs. For 13 cylinders it is 78 msec; for 2 cylinders it is 12 msec. Thus for randomly placed files the total is 203 msec, and for clustered files it is 137 msec.

31. 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 $8 + 2 + (k/262144) \times 4$. For blocks of 1 KB, 2 KB, and 4 KB, the access times are 10.015625 msec, 10.03125 msec, and 10.0625 msec, respectively (hardly any different). These give rates of about 102,240 KB/sec, 204,162 KB/sec, and 407,056 KB/sec, respectively.

32. If all files were 1 KB, then each 2-KB block would contain one file and 1 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 50 percent 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 17 disk blocks for storage, given a space efficiency of 32768/34816, which is about 94 percent.

33. The directory format allows for up to 255 blocks on the disk (due to the 8-bit number in the directory entry). This is clearly not enough so the first change has to be to go to more disk blocks. Instead of 16 8-bit addresses, we should use 8 16-bit addresses. This allows 65,536 disk blocks per disk. If we make blocks 32,768 bytes, the same size MS-DOS uses on large disks, the maximum disk size (actually, the maximum partition size) is now 2 GB. However, individual files are limited to eight disk blocks per extent and 255 extents for a maximum of 2040 blocks or 63.75 MB. A design with a maximum disk of 2 GB and a maximum file of 63.75 MB is plausible. If this is not enough, one could use one of the unused bytes in the directory entry to raise the extent field to 16 bits. This raises the number of blocks per file to 524,280 and the file size to almost 16 GB. Larger blocks sizes go even further.

34. The largest block is 32,768. With 32,768 of these blocks, the biggest file would be 1 GB.

35. 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.

36. The i-node holds 10 pointers. The single indirect block holds 256 pointers. The double indirect block is good for 256^2 pointers. The triple indirect block is good for 256^3 pointers. Adding these up, we get a maximum file size of 16,843,018 blocks, which is about 16.06 GB.

37. 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.

38. 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 7 PROBLEMS

1. XGA is 1024 × 768. With 24 bits/pixel and 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 × 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 × 720 versus 640 × 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 (Δx, Δy) 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. 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.

14. 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.

15. 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 buffered in an extremely large memory. Hard disks are simply better.

16. 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.

17. 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.

18. 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.

19. 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.

20. 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.

21. The buffer is big enough if the number of I-frames is 4 or less. The probability of getting exactly k I-frames is $C(24,k)I^kB^{24-k}$, 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 far too large to accept.

22. To get the play point in the middle of the buffer, we need to be able to read and store three streams at once. When the movie resumes at 12 min, we start storing the streams that are currently at 15 min and 20 min. After 3 minutes, we have stored 15–18 min and 20–23 min. At that point we drop the private stream and start displaying from the buffer. After an additional 2 min, we have 15–25 min stored and the play point is 17 min. At this point we only load the buffer from the stream now at 25 min. In 3 min, we have 15–28 min in the buffer and the play point is 20 min. We have achieved our goal. Because we are out of sync with the near video-on-demand streams, this is the best we can do.

23. 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.

24. 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.

25. A 14-GB disk holds 14×2^{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.

26. 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. 7-0(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. 7-0(b) would initially hit the first disk hardest, then the next disk 15 minutes later, etc. The arrangements of Fig. 7-0(c) or (d) would not be affected this way.

27. 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%.

28. 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 μsec. 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.

29. 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.

30. 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 μsec or 13 μsec does not make much difference.

31. 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.

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. 8-1c, 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 is distributed to remote nodes which then return results of calculations to the coordinating node.

2. Depending on the details of how CPUs are connected to memory, one of them gets through first, e.g., seizes the bus first. It completes its memory operation, then the other one happens. 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 12,500,000 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 non zero, lock was set, so loop
        RET                     | Return to caller; critical region entered
```

8. Probably locks on data structures are enough. 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, it is very likely that locks on data structures are sufficient.

9. 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.

10. Yes it would, but the interpoll time might end up being very long, degrading performance. But it would be correct, even without a maximum.

11. 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 be changed and the caller must loop. If it was previously unlocked, it is now locked.

12. 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.

13. *A* is the process just selected. There may well be others on the same CPU.

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 worse 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 64 links running between the two halves, for bisection bandwidth of 64 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 1132 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. On a shared memory machine it suffices to pass the pointer to the message from the CPU executing the send to the CPU executing the receive, with possible translations between virtual and physical memory addresses. On a multicomputer without shared memory an address in one CPU's address space has no meaning to another CPU, so the actual contents of the send buffer have to be transmitted as packets and then reassembled in the buffer of the receiving process. To the programmer the processes look identical, but the time required will be much longer on the multicomputer.

21. The time to move *k* bytes by programmed I/O is 20*k* nsec. The time for DMA is 2000 + 5*k* nsec. Equating these and solving for *k* we get the breakeven point at 133 bytes.

22. 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.

23. 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.

24. 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.

25. 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.

26. 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.

27. 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 only makes calls 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.

28. The most appropriate services are

(a) Unreliable connection.
(b) Reliable byte stream.

29. 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.

30. 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.

31. They can. For example, *www.intel.com* says nothing about where the server is.

32. 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 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.

33. 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.

34. Some cached file should be sent back to the server. All of the standard paging algorithms can be used, such as LRU or second chance. In contrast to virtual memory, however, using exact LRU is possible because file references are infrequent (millisecond time scale, not nanosecond time scale).

35. 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 cause thrashing. Putting each variable on a separate page is wasteful. Object-based access eliminates these problems and allows a finer grain of sharing.

36. 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 in 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. 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.

2. 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).

3. 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, etc.

4. 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 recrypts it with the secret key it shares with the receiver. This message is then sent to the receiver.

5. A function like $y = x^k$ is easy to compute but taking the k-th root of y is far more difficult.

6. 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 2×10^{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 5 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.

7. 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.

8. 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,* etc. If another user has password *Boston0004,* he will not discover it, however.

9. 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)

10. 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.

11. 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.

12. Both of them make use of one-way encryption functions. UNIX stores all passwords in the password file encrypted and Lamport’s scheme uses one-way functions to generate a sequence of passwords.

13. 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.

14. 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.

15. 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.

16. 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.

17. 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.

18. It should read the entire password all the time, even if it sees early on that the password is wrong. That way, a page fault will always occur when the password is partly on an out-of-memory page.

19. 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.

20. 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 is read.

21. 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);
```

22. 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, etc.

23. The compressor is needed to compress other executable programs as part of the process of infecting them.

24. 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.

25. First, running the *fdisk* 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.

26. 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.

27. A worm is a freestanding program that works by itself. A virus is a code fragment that attaches to another program. The worm reproduces by making more copies of the worm program. The virus reproduces by infecting other programs.

28. 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.

29. 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.

30. Here is the protection matrix:

Object				
Domain	**PPP-Notes**	**prog1**	**project.t**	**splash.gif**
asw	Read	Read Exec	Read Write	Read Write
gmw	Read Write		Read Write	
users	Read		Read Write	
devel		Read Exec		Read

31. 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.

32. If *asw* wants to allow *gmw* but no other member of users to look at splash.gif he could modify the ACL to asw:RW; devel:R; gmw:R.

33. 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.

34. 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.

35. 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.

36. 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.

37. No. That would be writing down, which violates the * property.

38. No. That would be reading up, which violates the simple security property.

39. 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*.

40. 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.

41. 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. Thus a single image can hold an entire floppy disk's worth of ASCII data. 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 two bits of ASCII text per payload bit. Thus per 24-bit pixel, effectively 6 bits of ASCII text are being encoded.

42. 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.

43. Following are two C programs that do the job. Run in in the same directory but in different windows. Start decode first.

```
/* covert encoder - Written by Albert S. Woodhull 3 Dec 2000
   C version, using file permissions

   This program expects to read a string of ASCII '0' and '1' characters
   from its standard input. It generates a covert output by alternately
   making a file owner readable or not readable, controlling the time the
   file is in each state.

   Each "bit time" consists of three time intervals. The middle interval
   determines the value of the bit.  To signal a ’0’ the file is readable
   during the first two intervals and not readable during the last
   interval.  To signal a ’1’ the file is readable during the first
   interval and not readable during the second and third intervals.

   A collaborator can determine the readability of a file for its owner even
   though the collaborator himself has no access, as long as the file is in
   a directory that can be read.

   The sleep system call is used to control the timing. This makes
   the whole process pretty slow, since you can’t sleep less than 1 second.
*/

#define MAX 80

#include <stdio.h>
#include <fcntl.h>

int main(void)
{
  int c;
  int i = 0;
  int n = 0;
  int fd;
  char s[MAX];

  /* get the input string, save only ’0’s and ’1’s, count chars */
  while (((c = getchar()) != EOF) && (n < MAX))
    if (( c == ’0’) || (c == ’1’)) s[n++] = c;
  s[n] = ’  ’;

  /* create the signal file */
  fd = creat("/tmp/tmp000", 0600);

  /* for each ’0’ or ’1’ execute the corresponding sequence */
  while (i != n)
  {
    c = s[i++];
    chmod("/tmp/tmp000", 0);
    switch(c)
    {
      case ’0’:
        sleep(2);
        chmod("/tmp/tmp000", 0400);
        sleep(1);
        break;
      case ’1’:
        sleep(1);
        chmod("/tmp/tmp000", 0400);
        sleep(2);
        break;
    }
  }

  /* get rid of the evidence */
  unlink("/tmp/tmp000");
}

----------------------------------------------------------------------------------------

/* covert decoder - Written by Albert S. Woodhull 3 Dec 2000
   C version, using file permissions.

   This program repeatedly checks the permissions of a file.  The
   companion encoder alternately changes the owner readable bit from 0 to
   1 on a timed basis. A transition from readable to unreadable signals
   the beginning of a bit, the timing of the transition back to readable
   signals the value of the bit. If the unreadable time is longer than
   the readable time the bit is a zero, if it is shorter the bit is a 1.
*/

#include <stdio.h>
#include <sys/stat.h>

#define TRUE      1
#define FALSE     0

int main(void)
{
  struct stat statbuf;
  int decoding, c0, c1, mode, p0, p1;
```

```
/* Start the decoder before starting the encoder. This loop waits for
   the file to be created. */
while (stat("/tmp/tmp000", &statbuf) < 0) /* do nothing */ ;

mode = statbuf.st_mode;

/* This loop detects the beginning of the first bit. */
while ((mode & S_IRUSR) != 0 )
{
  stat("/tmp/tmp000", &statbuf);
  mode = statbuf.st_mode;
}

decoding = TRUE;
while (decoding == TRUE)
{
  c0 = c1 = 0;

  /* phase 0, use c0 to count */
  p0 = TRUE;
  while (p0 == TRUE)
  {
    /* refresh statbuf and check for end of signal */
    if (stat("/tmp/tmp000", &statbuf) < 0)
      p0 = FALSE;
    mode = statbuf.st_mode;
    if ((mode & S_IRUSR) == 0) c0++; else p0 = FALSE;
  }

  /* phase 1, use c1 to count */
  p1 = TRUE;
  while (p1 == TRUE)
  {
    /* refresh statbuf and check for end of signal */
    if (stat("/tmp/tmp000", &statbuf) <0) p1 = FALSE;
    mode = statbuf.st_mode;
    if ((mode & S_IRUSR) != 0) c1++; else p1 = FALSE;
  }

  /* decide upon the bit value and output it */
  if (c0 > c1) printf("0"); else printf("1");

  /* make output visible now */
```

```
    fflush(NULL);

    /* see if the signal is still there */
    if (stat("/tmp/tmp000", &statbuf) < 0)
    {
      decoding = FALSE;
      putchar(’0);
    }
  }
}
```

SOLUTIONS TO CHAPTER 10 PROBLEMS

1. The calling process has to put the system call number in a register or on the stack.

2. The files that will be listed are: *bonefish*, *quacker*, *seahorse*, and *weasel*.

3. It prints the number of lines of the file *xyz* that contain the string “nd” in them.

4. 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.

5. 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.

6. Each program runs in its own process so six new processes are started.

7. 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.

8. 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.

9. The child can change a few variables and then exit. With fork, the parent is guaranteed that nothing the child can do will affect the parent’s address space. With vfork, this guarantee is no longer valid, introducing the possibility of hard-to-find bugs.

10. Every ΔT the CPU usage is divided in half, so after one interval it is 10, then 5, 2, 1, and 0. It takes $5\Delta T$ to hit 0.

11. Yes. It cannot run any more so the earlier its memory goes back on the free list, the better.

12. 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.

13. 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.

14. It would be impossible using UNIX or Windows 2000, 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 UNIX or Windows 2000. 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.

15. Generally, daemons run in the background doing things like printing and sending email. 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.

16. A PID must be unique. Sooner or later the counter will wrap around and go back to 0. Then it will so 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.

17. 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.

18. 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.

19. The 1000 is completely arbitrary. The only requirement is that every real-time thread gets a higher goodness than every timesharing thread.

20. Loading the operating system requires understanding the file system format, being able to search the root directory, and being able to interpret the executable binary format in which the operating system is stored on the disk. That is asking a lot from a 512-byte bootstrap program. It can just barely load the boot program from a fixed location, but this can be a long program, with detailed knowledge of the file system, directory, and binary formats.

21. 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.

22. Yes. With a 16-bit field, there can be a maximum of 64K core map entries, hence a maximum of 64K page frames. Thus there is no way to handle memories larger than 64 MB. When the original VAX came out, 2 MB was considered a very large memory, and 64 MB was effectively infinity. Now this 64-MB limit has become quite noticeable.

23. 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.

24. Two process 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.

25. Memory address 65,536 is file byte 0, so memory address 72,000 is file byte 6464.

26. 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 though 32,767

27. 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, it the stack has bumped into the data segment, the next page cannot be allocated to the stack, so the process must be terminated. 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.

28. It is possible if the two blocks are not buddies. Consider the situation of Fig. 10-0(e). Two new requests come in for 8 pages each. At this point the bottom 32 pages of memory are owned by 4 different users, each with 8 pages. Now users 1 and 2 release their pages, but users 0 and 3 hold theirs. This yields a situation with 8 pages used, 8 pages free, 8 pages free, and 8 pages used. We have two adjacent blocks of equal size that cannot be merged because they are not buddies.

29. 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.

30. 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.

31. 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.

32. 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.

33. One approach is give an error and refuse to carry out the lseek. Another is to make the offset become negative. As long as it is not used, there is no harm done. Only if an attempt is made to read or write the file should be error message be given. If the lseek is followed by another lseek that makes the offset positive, no error is given.

34. 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.

35. 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.

36. No. The file still has only one owner. If, for example, only the owner can write on the file, the other party cannot do so. 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.

37. When the working directory is changed, using the chdir system call, 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.

38. 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 before the needed i-node is in memory.

39. The i-node holds 10 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 10 + 256 + 65,536 + 16,777,216 blocks, which is about 16 gigabytes.

40. 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.

41. Buffer cache accesses are so infrequent compared to memory accesses, that the cost of managing the LRU queue in software is acceptable. With pages, there are typically one or two references per instruction, which makes maintaining an LRU queue impractical.

42. By forcing the contents of the buffer cache out onto the disk every 30 sec, damage done by a crash is limited to 30 sec. If *update* did not run, a process might write a file, then exit with the full contents of the file still in the buffer cache. In fact, the user might then log out and go home with the file still in the buffer cache. An hour later the system might crash and lose the file, still only in the buffer cache and not on disk. The next day we would not have a happy user.

43. All it has to do is set the link count to 1 since only one directory entry references the i-node.

44. 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.

45. The file is simply removed. This is the normal way (actually, the only way) to remove a file.

46. A 1.44-MB floppy disk can hold 1440 blocks of raw data. The boot block, super block, group descriptor block, block bitmap, and i-node bitmap of an ext2 file system each use 1 block. If 8192 128-byte i-nodes are created, these i-nodes would 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 inodes occupying 23 blocks will be created. However, because of the overhead of the ext2 file system, Linux normally uses the MINIX file system on floppy disks and other small devices.

47. 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. Many other such examples exist.

48. Probably someone had the file open when the professor changed the permissions. The professor should have deleted the file and then put another copy of his master file 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.

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 with 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. DPCs run in an arbitrary context. APCs run in the context of a specific thread. The act of signaling the process involves writing a frame on the user's stack so the signal can be returned from. This requires access to the user's address space. Consequently, an APC is needed.

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 mgr 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 only make sense 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 functions in *ntdll.dll* are those needed by the subsystems, that is, the system calls. The functions in *ntoskrnl.exe* are exported functions used by drivers and other parts of the operating system. There is no relation between them. The fact that they are even close is just a coincidence. They could easily have differed by a factor of five.

9. No. The low-order 3 bits of the object pointer in the handle are used for flags, as stated in the text. These must be masked off before the pointer can be followed. Consequently every pointer has 3 zero bits as the low-order bits. This means that object headers must start at addresses that are multiples of 8 bytes.

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. (a) The last thread exits.
(b) A thread executes ExitProcess.
(c) Another process with a handle to this one kills it.

13. The Windows 2000 Professional quantum is 20 msec and there are 12 threads at priorities higher than 3. Thus the first priority 3 thread must wait 240 msec until it gets a shot.

14. 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.

15. Having your priority lowered below the base priority could be used as a punishment for using excessive CPU time or other resources.

16. The processor will not permit such instructions to be executed in user mode and will trap them as errors. In principle, a filter that reads a binary program and replaces all IN and OUT instructions with calls to routines that use legal OS services could be written. Alternatively, the program could be interpreted, with these instructions handled by making Windows 2000 system calls to do the I/O.

17. One way is to increase the priority of important processes. A second way is to give important processes longer quanta.

18. The problem cannot be solved using page tables. Pages tables map between virtual and physical addresses. The problem here is that the shared procedure is being placed at different virtual addresses. If an instruction at address 65,536 reads JMP 300, it will jump to invalid memory (below 64 KB) and trap. If it is patched to read JMP 65836 that will go forward 300 bytes, which is correct for that process. But if a different process has this instruction at, say, address 131,072, jumping to 65836 is wrong. It does not matter where in physical memory the page is. For the second process, the CPU is generating an incorrect virtual address. No setting of the page tables can correctly map an incorrect virtual address.

19. The Pentium has multiple segments, each starting at virtual address 0. Each *.dll* file could be mapped into a separate segment starting at virtual address 0 in that segment. To do this, segmentation would have to be supported by Windows 2000.

20. 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.

21. (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.

22. It is not moved at all. A page only goes onto one of the lists 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.

23. 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.

24. Copying one word requires one read and one write, for a total of 20 nsec to move 4 bytes. This comes to 5 nsec per byte copied. An XGA screen has $1024 \times 768 \times 3$ bytes, which takes about 11.8 msec to copy in the best case (assuming the program runs entirely out of the L1 cache).

25. 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)

26. 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)

27. It is an accident. The 16 blocks apparently compressed to 8 blocks. It could have been 9 or 11 just as easily.

28. All except the user SID could be removed without affecting the strength of the security.

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. Putting it there saved some RAM and reduced loading time to 0, but most important, it made it easy for third-party software developers to use the GUI, thus ensuring a uniformity of look and feel across all software.

4. 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.

5. 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.

6. If drivers are put below the threads, then drivers cannot be independent threads in the style of MINIX. They have to run as part of some other thread, more in the UNIX style.

7. It is possible. What is needed is a user-level process, the semaphore server. 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.

8. 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.

9. The mechanism for selling to customers is a building with shelves, employees to stock the shelves, cashiers to handle payment, etc. The policy is what kind of products the store sells.

10. External names can be as long as needed and variable length. Internal names are generally 32 bits 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 name space.

11. 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 versus space trade-off.

12. 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 are turned on, and so on. What might work with one compiler could fail with a different one. It is better to set a flag.

13. 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.

14. Yes. It makes the code slower. Also, more code means more bugs.

15. 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 caller to a simple procedure might be unexpectedly blocked.

16. 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 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.

17. 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) ˆ ((w>>16) & 0xFFFF)]
```

18. 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.

19. 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 × 384 pixels. For VGA (640 × 480), 16-bit color requires less data than true 24-bit color.

20. 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.

21. 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.

22. 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.

23. 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.

24. Brooks was talking about large projects in which communication between the programmers slows everything down. That problem does not occur with a 1-person project and so productivity can be higher.

25. If a programmer can produce 1000 lines of code for a cost of $100,000, a line of code costs $100. In Chap. 11, we stated that Windows 2000 consisted of

29 million lines of code, which comes to $2.9 billion. That seems like an awful lot. Probably Microsoft has managed to improve programmer productivity using better tools so a programmer can produce several thousand lines of code per year.

26. Suppose memory costs $100 for 64 MB (check against current prices). Then a low-end machine needs $1600 worth of disk. If the rest of the PC is $500, the total cost comes to $2100. This is too expensive for the low-end market.

27. An embedded system may run one 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.