OPERATING SYSTEMS: DESIGN AND IMPLEMENTATION

THIRD EDITION

PROBLEM SOLUTIONS

ANDREW S. TANENBAUM

Vrije Universiteit
Amsterdam, The Netherlands

ALBERT S. WOODHULL

Amherst, Massachusetts

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. In kernel mode, every machine instruction is allowed, as is access to all the I/O devices. In user mode, many sensitive instructions are prohibited. Operating systems use these two modes to encapsulate user programs. Running user programs in user mode keeps them from doing I/O and prevents them from interfering with each other and with the kernel.

3. 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.

4. 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.

5. The prime reason for multiprogramming is to give the CPU something to do while waiting for I/O to complete. If there is no DMA, the CPU is fully occupied doing I/O, so there is nothing to be gained (at least in terms of CPU utilization) by multiprogramming. No matter how much I/O a program does, the CPU will be 100 percent busy. This of course assumes the major delay is the wait while data is copied. A CPU could do other work if the I/O were slow for other reasons (arriving on a serial line, for instance).

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

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

8. 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.

9. Arguments for closed source are that the company can vet the programmers, establish programming standards, and enforce a development and testing methodology. The main arguments for open source is that many more people look at the code, so there is a form of peer review and the odds of a bug slipping in are much smaller with so much more inspection.

10. The file will be executed.

11. 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.

12. Any file can easily be named using its absolute path. Thus getting rid of working directories and relative paths would only be a minor inconvenience. The other way around is also possible, but trickier. In principle if the working directory is, say, */home/ast/projects/research/proj1* one could refer to the password file as *../../../../etc/passwd*, but it is very clumsy. This would not be a practical way of working.

13. 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.

14. 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.

15. The read works normally. User 2's directory entry contains a pointer to the i-node of the file, and the reference count in the i-node was incremented when user 2 linked to it. So the reference count will be nonzero and the file itself will not be removed when user 1 removes his directory entry for it. Only when all directory entries for a file have been removed will its i-node and data actually vanish.

16. No, they are not so essential. In the absence of pipes, program 1 could write its output to a file and program 2 could read the file. While this is less efficient than using a pipe between them, and uses unnecessary disk space, in most circumstances it would work adequately.

17. The display command and reponse for a stereo or camera is similar to the shell. It is a graphical command interface to the device.

18. Windows has a call spawn that creates a new process and starts a specific program in it. It is effectively a combination of fork and exec.

19. If an ordinary user could set the root directory anywhere in the tree, he could create a file *etc/passwd* in his home directory, and then make that the root directory. He could then execute some command, such as *su* or *login* that reads the password file, and trick the system into using his password file, instead of the real one.

20. The getpid, getuid, getgid, and getpgrp, calls just extract a word from the process table and return it. They will execute very quickly. They are all equally fast.

21. The system calls can collectively use 500 million instructions/sec. If each call takes 1000 instructions, up to 500,000 system calls/sec are possible while consuming only half the CPU.

22. No, unlink removes any file, whether it be for a regular file or a special file.

23. When a user program writes on a file, the data does not really go to the disk. It goes to the buffer cache. The *update* program issues SYNC calls every 30 seconds to force the dirty blocks in the cache onto the disk, in order to limit the potential damage that a system crash could cause.

24. No. What is the point of asking for a signal after a certain number of seconds if you are going to tell the system not to deliver it to you?

25. Yes it can, especially if the system is a message passing system.

26. When a user program executes a kernel-mode instruction or does something else that is not allowed in user mode, the machine *must* trap to report the attempt. The early Pentiums often ignored such instructions. This made them impossible to fully virtualize and run an arbitrary unmodified operating system in user mode.

SOLUTIONS TO CHAPTER 2 PROBLEMS

1. It is central because there is so much parallel or pseudoparallel activity—multiple user processes and I/O devices running at once. The multiprogramming model allows this activity to be described and modeled better.

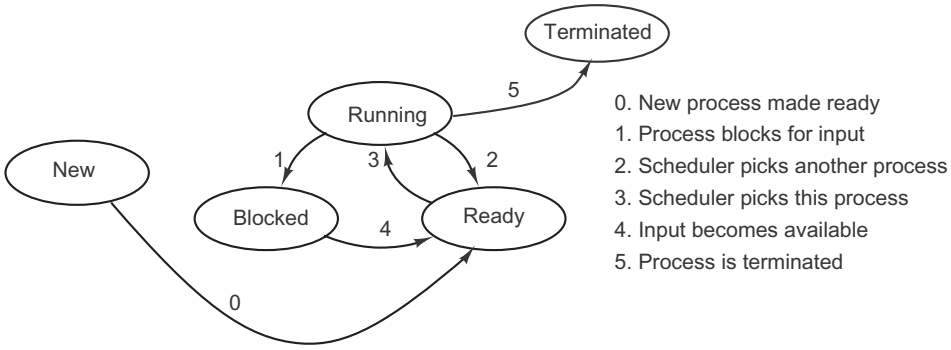
2. The states are running, blocked and ready. The running state means the process has the CPU and is executing. The blocked state means that the process cannot run because it is waiting for an external event to occur, such as a message or completion of I/O. The ready state means that the process wants to run and is just waiting until the CPU is available.

3. 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.

4. 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.

5. The figure looks like this



6. It would be difficult, if not impossible, to keep the file system consistent using the model in part (a) of the figure. 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.

7. A process is a grouping of resources: an address space, open files, signal handlers, and one or more threads. A thread is just an execution unit.

8. Each thread calls procedures on its own, so it must have its own stack for the local variables, return addresses, and so on.

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

10. One person calls up a travel agent to find about price and availability. Then he calls the other person for approval. When he calls back, the seats are gone.

11. A possible shell script might be:

```
if [ ! –f numbers ]; 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.

12. Yes, at least in MINIX 3. Since LINK is a system call, it will activate server and task level processes, which, because of the multi-level scheduling of MINIX 3, will receive priority over user processes. So one would expect that from the point of view of a user process, linking would be equivalent to an atomic act, and another user process could not interfere. Also, even if another user process gets a chance to run before the LINK call is complete, perhaps because the disk task blocks looking for the inode and directory, the servers and tasks complete what they are doing before accepting more work. So, even if two processes try to make a LINK call at the same time, whichever one causes a software interrupt first should have its LINK call completed first.

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

14. 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.

15. 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.

16. 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.

17. 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.

18. It is very expensive to implement. Each time any variable that appears in a predicate on which some process is waiting changes, the run-time system must re-evaluate the predicate to see if the process can be unblocked. With the Hoare and Brinch Hansen monitors, processes can only be awakened on a SIGNAL primitive.

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

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

21. 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.

22. The change would mean that after a philosopher stopped eating, neither of his neighbors could be chosen next. With only two other philosophers, both of them neighbors, the system would deadlock. With 100 philosophers, all that would happen would be a slight loss of parallelism.

23. 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.

24. It will need nT sec.

25. 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.

26. 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.

27. 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.

28. 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.

29. 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.

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

31. Yes. Two-level scheduling could be used if 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.

32. There are three ways to pick the first one, four ways to pick the second, three ways to pick the third and four ways to pick the fourth, for a total of $3 \times 4 \times 3 \times 4 = 144$. Note that a thread can be chosen a second time.

33. 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.

34. This pointer makes it easy to find the place to save the registers when a process switch is needed, either due to a system call or an interrupt.

35. When a clock or keyboard interrupt occurs, and the task that should get the message is not blocked, the system has to do something strange to avoid losing the interrupt. With buffered messages this problem would not occur. Notification bitmaps provide provide a simple alternative to buffering.

36. While the system is adjusting the scheduling queues, they can be in an inconsistent state for a few instructions. It is essential that no interrupts occur during this short interval, to avoid having the queues accessed by the interrupt handler while they are inconsistent. Disabling interrupts prevents this problem by preventing recursive entries into the scheduler.

37. When a RECEIVE is done, a source process is specified, telling who the receiving process is interested in hearing from. The loop checks to see if that process is among the process that are currently blocked trying to send to the receiving process. Each iteration of the loop examines another blocked process to see who it is.

38. Tasks, drivers and servers get large quanta, but even they can be preempted if they run too long. Also if a driver or server is not allowing other processes to run it can be demoted to a lower-priority queue. Even though they are given large quanta, all system processes are expected to block eventually. They only run to carry out work requested by user processes, and eventually they will complete their work and allow user processes to run.

39. MINIX 3 could probably be used for data logging with long sampling periods, for instance weather monitoring, but there is no way to guarantee immediate availability in response to an external event. However, faster data acquisition would be possible if the data to be collected were received by means of an existing interface supported by an interrupt (i.e., a serial port), or if a new interrupt-driven driver for an interface to the data source were added. Also, the priorities of drivers and servers are not engraved in stone—a new driver could be configured to run at higher priority than existing drivers or existing drivers could be configured for lower priorities in order to provide better service for a time critical interface.

SOLUTIONS TO CHAPTER 3 PROBLEMS

1. With 1x at 1.32 MB/sec and USB 2.0 at 60 MB/sec, a 45x DVD drive would just barely make it. In practice, 30x or 40x might be safer though.

2. One possibility is for the disk controller to notice and reread the block. If it is successful the second time, the software is not even told about the error (except possibly for logging purposes). An alternative is to report all errors to the device driver and let it worry about the problem. With a sophisticated drive with advanced integrated electronics repeated errors might lead to automatic substitution of a spare sector for the faulty one.

3. Memory-mapped I/O puts the I/O registers in the normal memory space so they can be accessed just as any other memory locations. This allows them to be accessed by any machine instruction. It also allows them to be protected by whatever memory-management scheme is used (e.g., paging).

4. Most I/O consists of repeatedly transferring bytes between an I/O device and consecutive locations in memory. DMA allows this transfer to be performed by a special chip rather than the main CPU, thus freeing up the CPU for other work in parallel with the transfer.

5. The limiting factor could be the speed the device can produce data, the speed of the bus, or the speed of the memory.

6. At 1 GHz, the total time required per second for interrupt handling is 44,100 microsec or 44.1 msec. If this were 22.67 times slower, it would take up the entire CPU, so we can reduce the clock by this factor to get a 44.1 MHz clock.

7. In situations where response time after an external signal is asserted must be minimized, polling may be better, especially in embedded applications. Catching an interrupt, changing the page map, flushing all the caches, etc. takes some time so polling is generally faster. If the CPU has nothing else to do while waiting (as is often the case in embedded applications), polling is probably preferred.

8. When the controller does a read, it generally continues to read the rest of the track it just read from and store the data in case it needs it later. The more internal memory it has, the more tracks it can store like this and the better the performance. Faster processors demand faster data access, but the same advances in technology make bigger buffers more affordable.

9. The system calls the driver to get work done, such as read or write a block of data, turn lights on the device on or off, or go into a low-power state. The driver calls the system when unpredictable data arrives, such as a key press on a keyboard or a button push on a mouse, and to get system services such as allocating memory for buffering data.

10. Operating system designers want users to be able to do I/O without worrying about the characteristics of the device. Having to code differently to access an IDE disk vs. a SCSI disk would be a real nuisance. Characteristics such as device speed, block size, and geometry, among many others, should be completely hidden from users.

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

12. 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.

13. Imagine that four cars pull up to a four-way stop simultaneously. If the rule is first-come first-served, no one was first so no one can go. If the rule is car on the right goes first, they are equally deadlocked. There are obviously many other answers.

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

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

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

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 processes, each holding one tape drive and wanting another one, we have a deadlock. Thus for $n < 6$, the system is deadlock-free.

19. 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, printers, 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.

20. 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.

21. 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.

22. Comparing a row in the matrix to the vector of available resources takes m operations. This step must be repeated on the order of n times to find a process that can finish and be marked as done. Thus marking a process as done takes on the order of mn steps. Repeating the algorithm for all n processes means that the number of steps is then mn^2.

23. The new need matrix after these requests is:

A: 2 3 1 0
B: 0 1 1 2
C: 3 1 0 0
D: 3 3 2 0

At this point there is no process all of whose needs can be met, so the system is in an unsafe state.

24. If both programs ask for Woofer first, the computers will starve with the endless sequence: request Woofer, cancel request, request Woofer, cancel request, and so forth. 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.

25. There are 512,000 bytes around the circumference of the disk. These bytes can be read in 10 msec, for a data rate of 51,200,000 bytes/sec. Such track-to-track switching is electronic, such a burst can be maintained for an entire cylinder, which takes 80 msec to read. After that a seek is needed, during which time no data can be transferred.

26. 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.)

27. The *POSITION* field is not needed for character devices, since they are not randomly addressable.

28. (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.

29. Not necessarily. A 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.

30. It is like a subroutine (procedure). Each time the user part invokes the kernel part, the kernel starts out in the same place. It does not remember where it was last time.

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

32. Tabs and line feeds on hardcopy terminals may also need delays. Other examples are checking whether a possibly slow operation actually completed, as with a disk read or waiting for a network ACK.

33. 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.

34. At 110 baud, we have 10 interrupts/sec, which is 40 msec or 4 percent of the CPU. No problem. At 4800 baud, we have 480 interrupts/sec, which takes 1920 msec. In other words, it cannot be done. The system can handle at most 250 interrupts/sec.

35. Scrolling the window requires copying 65 lines of 80 characters or 5200 characters. Copying 1 character (12 bytes) takes 6 microsec, so the whole window takes 31.2 msec. Writing 80 characters to the screen takes 4 msec, so scrolling and displaying a new line take 35.2 msec for 80 characters. This is only 2273 characters/sec, or 22.7 kilobaud, barely faster than 19.2 kilobaud. For color, everything takes four times as long, so we only get 5683 baud.

36. Escape characters make it possible to output backspaces and other characters that have special meaning to the driver, such as cursor motion.

37. 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.

38. The microprocessor 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.

39. 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 microsec/scan, the beam is moving horizontally 763 msec per second, leaving 237 msec for writing in the video RAM. Thus the video RAM is only available 23.7 percent of the time.

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. 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.

3. Memory is 2^{30} bytes and the allocation unit is 2^{16} bytes, so the number of allocation units is 2^{14}. Each of these requires one bit, so 2 KB of memory are needed for the bit map.

4. Each list entry will be 12 bytes: 4 bytes for the base address of the hole, 4 bytes for the hole size, and 4 bytes for the pointer to the next entry. In the best case, memory above the operating system is one big hole, so only 12 bytes are needed. Worst case is memory in which 64-KB data segments and holes alternate. Excluding the operating system, there are 16,376 such units, half of them holes, so the list needs 8188 nodes at 12 bytes each for a total of 98,256 bytes or almost 96 KB.

5. Real memory uses physical addresses. These are the numbers that the memory chips react to on the bus. Virtual addresses are the logical addresses that refer to a process' address space. Thus a machine with a 16-bit word can generate virtual addresses up to 64K, regardless of whether the machine has more or less memory than 64 KB.

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

7. If the virtual address is smaller than the physical address, the entire address space of a given program can be in memory at once. If it is larger, the entire program cannot be in memory at once and paging will be needed. Both are possible. If the two are equal, in theory the program could fit, except that the operating system probably takes up some space.

8. They built an MMU and inserted it between the 8086 and the bus. Thus all 8086 physical addresses went into the MMU as virtual addresses. The MMU then mapped them onto physical addresses, which went to the bus.

9. A page fault every k instructions adds an extra overhead of n/k nsec to the average, so the average instruction takes $1 + n/k$ nsec.

10. 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.

11. 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.

12. The reference string is 1(I), 12(D); 2(I), 15(D); 2(I), 15(D); 10(I); 10(I); 15(D); 10(I). The code (I) indicates an instruction reference, whereas (D) indicates a data reference. Semicolons give the instruction boundaries. Note that for some architectures an instruction with immediate data might result in two (I) references, it would depend upon whether the data and the opcode both fit in a single word, or whether after decoding the instruction a second fetch is necessary to fetch the immediate data from the (I) space.

13. 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.

14. The effective instruction time is $100h + 500(1 - h)$, where h is the hit rate. If we equate this formula with 200 and solve for h, we find that h must be at least 0.75.

15. 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.

16. 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).

17. Technically, a TLB with two entries—one for the code page and one for a data page—would be enough to make it run, but performance would not be very good. In practice, there is no architectural reason for a particular size TLB. It is just an engineering tradeoff. A size of 64 or 128 entries might work well.

18. NRU removes page 0. FIFO removes page 2. LRU removes page 1. Second chance removes page 0.

19. 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.

20. The inspection order is from oldest to newest. As soon as one is found with the reference bit 0, it is chosen. The sequence is: *N*, *G*, *H*, *D*. Since *D* was not referenced in the last interval, it is selected for replacement.

21. No, the algorithm is the same. They differ only in the data structure used to keep track of the pages.

22. The algorithm will take memory away from each process as a punishment for not faulting enough. Eventually all processes will begin faulting, but there will be a large pool of unused pages. The algorithm does not specify what to do with them.

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

24. The seek plus rotational latency is 14 msec. The transfer rate is 1 MB in 8 msec or 1 KB in 8 μsec. For 2-KB pages, the transfer time is 16 μsec, for a total of 14.016 msec. Loading 32 of these pages will take 448.5 msec. For 4-KB pages, the transfer time is doubled to 32 μsec, so the total time per page is 14.032 msec. Loading 16 of these pages takes 224.5 msec. For a single 64-KB page, the transfer time is 512 μsec. Adding in seek and rotational latency we get 14.5 msec for loading the whole 64 KB. Conclusion: big pages are more efficient for transfer.

25. First, there is internal fragmentation on the disk, wasting disk space. Second, there is internal fragmentation in memory, wasting RAM. Furthermore, large pages reduce the number of pages in memory, and for a program whose working set is spread all over its address space, reducing the number of pages may reduce performance. In an extreme case, if the virtual address space is 4 GB, physical memory is 64 MB and pages are 4 MB, only 16 pages fit in memory. If the program is actively using 1024 regions of memory of size 2 KB each spread uniformly throughout the virtual address space, it will thrash wildly, even though it is actively using only 2 MB.

26. 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.

27. 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.

28. 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.

29. What happens if the executable binary file is modified while the program is running? Using a mixture of pages from the old and new binaries is sure to crash it. While this event is unlikely, it is a calculated risk. Alternatively, the binary could be locked against modification while it was being used as a backing store.

30. 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.

31. 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.

32. When a program is started up, it gets a fixed allocation that it can never change. The operating system has to know how much to give it. That information is in the header. If the programmer is not sure how much is needed, he needs something like *chmem* to be able to try different values.

33. The boot monitor display shows the cs and ds segment addresses as hexidecimal numbers, but the sizes are reported as decimal numbers. The data segment for rs starts at 0x70e000 and occupies 616 + 4696 + 131072 = 136384 bytes, and thus extends to 0x72f4c0. Round this up to the next click boundary, 0x72f800, to find the next address where another process can be loaded.

34. If the monitor were given exactly as much space as it needed, the upper limit of the available memory region would be at 655360 - 52256 = 603104 or 0x933e0. The next lower click boundary is at 0x93000. The kernel's data segment starts at 0x5800 and occupies 3140 + 30076 = 33216 bytes, or up to address 0xd9c0. The next higher click boundary is at 0xdc00, so a total of 0x93000 - 0xdc00 = 0x85400 or 545972 (decimal) bytes are available in this region for MINIX 3 programs.

35. Even if the boot monitor were given exactly the amount of space it needs, rather than being loaded on a click boundary, it would not matter to the rest of MINIX 3, which can only allocate memory in click units.

SOLUTIONS TO CHAPTER 5 PROBLEMS

1. The Unicode character set includes virtual every alphabet on earth as well as many Japanese kanji symbols, etc. Thus it is possible for Greeks, Russians, Israelis, Japanese, Chinese, and many other people to give files names in their own language.

2. When a program opens a file, it can check if the right magic number is there to avoid problems when it is inadvertently given an incorrect file. For example, because executable files begin with a magic number, the operating system can determine if a file is executable or not by checking the magic number.

3. It is plausible. If a file were of type ASCII, one bit is available in each character for parity. A file with the parity attribute could have a parity bit in every character. Such a bit would provide some error detection capability. In fact, if a run of 512 characters used all 512 parity bits for error correction, a fairly substantial error correcting code could be implemented in the parity bits.

4. 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
```

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

6. Yes. The rename call does not change the creation time or the time of last modification, but creating a new file causes it to get the current time as both the creation time and the time of last modification.

7. 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*.

8. It is fairly pointless. Given that arrangement, one could create a global root that listed all the personal roots. Then we are back to the standard hierarchical file system. Furthermore, with the chroot system call, it is already possible to have a personal root, at least if you have permission to call chroot.

9. It certainly does. If you can turn */usr/ast* into the root, then you can create a directory *etc* in the root and place a file *passwd* in there. If you then issue the *su* command, the system looks in your password file to see if you know the root password. Since you have created this file yourself, you probably know the password. The entire security system collapses if you can create your own root. This is why chroot is a superuser only call.

10. In theory yes they could, but then they would have to fully understand the layout of directories, which can be quite complicated. Having a special call to read a directory entry makes reading directories much simpler.

11. The limit of four comes from the number of table entries in the MBR. By changing this table to, say, 8 entries, up to 8 operating systems could be supported. The downside of doing this would be that your system would not be backward compatible with a great deal of existing software.

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. The FAT would need 2^{32} entries of 4 bytes each for a total size of 2^{34} bytes. Not very practical to keep in memory all at once.

14. 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.

15. The bit map 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.

16. Many UNIX files are short. If the entire file fit 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.

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

18. A hard link is a directory entry that points directly to a file's metadata (e.g., i-node in UNIX). A symbolic link is a small file that contains the name of a file. Hard links are more efficient than symbolic links. Symbolic links can point to files on different disks.

19. Watch out for backing up bad (i.e., damaged and unreadable) blocks, holes inside files, and shared files.

20. 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, 5.02 msec per block read. Only the seek time differs. For 100 random seeks it is 500 msec; for 2-cylinder seeks it is 10 msec. Thus for randomly placed files the total is 502 + 500 = 1002 msec, and for clustered files it is 502 + 2 = 504 msec.

21. If done right, yes. While compacting, each file should be organized so that all of its blocks are consecutive, for fast access.

22. 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.

23. Nothing (except maybe 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.

24. No, it does not. The student can easily find out what the random number for his super-user 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.

25. 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.

26. 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 super-user. The machine receiving the message has no way of telling if the command really did originate with the super-user, or with a student.

27. From a security point of view, it would be ideal. Used blocks sometimes are exposed, leaking valuable information. From a performance point of view, it would generate a large number of additional disk writes, thus degrading performance. On the large Burroughs mainframes, files designated as critical are erased when released, but ordinary files are not. This is a reasonable compromise.

28. 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 or the *rwx* bit mechanism are easy to use. It may also be possible to put a capability for the file or files in a well-known place in a capability system.

29. 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.

30. It might be better to define *NR_FILPS* as *4 * NR_PROCS*, so the same average number of *filp* entries will be available for each process in a system that accomodates more processes. There are other constants that might better be defined as functions of other configurable constants; another example is *NR_INODES*, in the same file as *NR_FILPS*.

31. Maintaining data integrity in the event of a power outage would be easier. There will be complications, however. With current memory technology the main worry is that failure during a write will corrupt the data on the disk. With non-volatile memory, failure during a read could corrupt data in memory, so recovering from a failure will require determining which copy of the data is correct when there is a difference.