MODERN OPERATING SYSTEMS

FOURTH EDITION

PROBLEM SOLUTIONS

ANDREW S. TANENBAUM

HERBERT BOS

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 machine, and it must manage the I/O devices and other system resources. To some extent, these are different functions.

2. Obviously, there are a lot of possible answers. Here are some.
Mainframe operating system: Claims processing in an insurance company.
Server operating system: Speech-to-text conversion service for Siri.
Multiprocessor operating system: Video editing and rendering.
Personal computer operating system: Word processing application.
Handheld computer operating system: Context-aware recommendation system.
Embedded operating system: Programming a DVD recorder for recording TV.
Sensor-node operating system: Monitoring temperature in a wilderness area.
Real-time operating system: Air traffic control system.
Smart-card operating system: Electronic payment.

3. In a timesharing system, multiple users can access and perform computations on a computing system simultaneously using their own terminals. Multiprogramming systems allow a user to run multiple programs simultaneously. All timesharing systems are multiprogramming systems but not all multiprogramming systems are timesharing systems since a multiprogramming system may run on a PC with only one user.

4. Empirical evidence shows that memory access exhibits the principle of locality of reference, where if one location is read then the probability of accessing nearby locations next is very high, particularly the following memory locations. So, by caching an entire cache line, the probability of a cache hit next is increased. Also, modern hardware can do a block transfer of 32 or 64 bytes into a cache line much faster than reading the same data as individual words.

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

6. Access to I/O devices (e.g., a printer) is typically restricted for different users. Some users may be allowed to print as many pages as they like, some users may not be allowed to print at all, while some users may be limited to printing only a certain number of pages. These restrictions are set by system administrators based on some policies. Such policies need to be enforced so that user-level programs cannot interfere with them.

7. It is still alive. For example, Intel makes Core i3, i5, and i7 CPUs with a variety of different properties including speed and power consumption. All of these machines are architecturally compatible. They differ only in price and performance, which is the essence of the family idea.

8. A 25×80 character monochrome text screen requires a 2000-byte buffer. The 1200×900 pixel 24-bit color bitmap requires 3,240,000 bytes. In 1980 these two options would have cost $10 and $15,820, respectively. For current prices, check on how much RAM currently costs, probably pennies per MB.

9. Consider fairness and real time. Fairness requires that each process be allocated its resources in a fair way, with no process getting more than its fair share. On the other hand, real time requires that resources be allocated based on the times when different processes must complete their execution. A real-time process may get a disproportionate share of the resources.

10. Most modern CPUs provide two modes of execution: kernel mode and user mode. The CPU can execute every instruction in its instruction set and use every feature of the hardware when executing in kernel mode. However, it can execute only a subset of instructions and use only subset of features when executing in the user mode. Having two modes allows designers to run user programs in user mode and thus deny them access to critical instructions.

11. Number of heads = 255 GB / (65536*255*512) = 16
Number of platters = 16/2 = 8
The time for a read operation to complete is seek time + rotational latency + transfer time. The seek time is 11 ms, the rotational latency is 7 ms and the transfer time is 4 ms, so the average transfer takes 22 msec.

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

13. It may take 20, 25 or 30 msec to complete the execution of these programs depending on how the operating system schedules them. If *P0* and *P1* are scheduled on the same CPU and *P2* is scheduled on the other CPU, it will take 20 msec. If *P0* and *P2* are scheduled on the same CPU and *P1* is scheduled on the other CPU, it will take 25 msec. If *P1* and *P2* are scheduled on the same CPU and *P0* is scheduled on the other CPU, it will take 30 msec. If all three are on the same CPU, it will take 35 msec.

14. Every nanosecond one instruction emerges from the pipeline. This means the machine is executing 1 billion instructions per second. It does not matter at all how many stages the pipeline has. A 10-stage pipeline with 1 nsec per stage would also execute 1 billion instructions per second. All that matters is how often a finished instruction pops out the end of the pipeline.

15. Average access time =

0.95×1 nsec (word is in the cache)
$+ 0.05 \times 0.99 \times 10$ nsec (word is in RAM, but not in the cache)
$+ 0.05 \times 0.01 \times 10{,}000{,}000$ nsec (word on disk only)
$= 5001.445$ nsec
$= 5.001445$ μsec

16. Maybe. If the caller gets control back and immediately overwrites the data, when the write finally occurs, the wrong data will be written. However, if the driver first copies the data to a private buffer before returning, then the caller can be allowed to continue immediately. Another possibility is to allow the caller to continue and give it a signal when the buffer may be reused, but this is tricky and error prone.

17. A trap instruction switches the execution mode of a CPU from the user mode to the kernel mode. This instruction allows a user program to invoke functions in the operating system kernel.

18. The process table is needed to store the state of a process that is currently suspended, either ready or blocked. Modern personal computer systems have dozens of processes running even when the user is doing nothing and no programs are open. They are checking for updates, loading email, and many other things, On a UNIX system, use the *ps -a* command to see them. On a Windows system, use the task manager.

19. Mounting a file system makes any files already in the mount-point directory inaccessible, so mount points are normally empty. However, a system administrator might want to copy some of the most important files normally located in the mounted directory to the mount point so they could be found in their normal path in an emergency when the mounted device was being repaired.

20. Fork can fail if there are no free slots left in the process table (and possibly if there is no memory or swap space left). Exec can fail if the file name given does not exist or is not a valid executable file. Unlink can fail if the file to be unlinked does not exist or the calling process does not have the authority to unlink it.

21. Time multiplexing: CPU, network card, printer, keyboard.
Space multiplexing: memory, disk.
Both: display.

22. If the call fails, for example because *fd* is incorrect, it can return −1. It can also fail because the disk is full and it is not possible to write the number of bytes requested. On a correct termination, it always returns *nbytes*.

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

24. Time to retrieve the file =
1 * 50 ms (Time to move the arm over track 50)
+ 5 ms (Time for the first sector to rotate under the head)
+ 10/200 * 1000 ms (Read 10 MB)
= 105 ms

25. Block special files consist of numbered blocks, each of which can be read or written independently of all the other ones. It is possible to seek to any block and start reading or writing. This is not possible with character special files.

26. System calls do not really have names, other than in a documentation sense. When the library procedure *read* traps to the kernel, it puts the number of the system call in a register or on the stack. This number is used to index into a table. There is really no name used anywhere. On the other hand, the name of the library procedure is very important, since that is what appears in the program.

27. This allows an executable program to be loaded in different parts of the machine's memory in different runs. Also, it enables program size to exceed the size of the machine's memory.

28. As far as program logic is concerned, it does not matter whether a call to a library procedure results in a system call. But if performance is an issue, if a task can be accomplished without a system call the program will run faster. Every system call involves overhead time in switching from the user context to the kernel context. Furthermore, on a multiuser system the operating system may schedule another process to run when a system call completes, further slowing the progress in real time of a calling process.

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

Link: a Win32 program cannot refer to a file by an alternative name or see it in more than one directory. Also, attempting to create a link is a convenient way to test for and create a lock on a file.

Mount and umount: a Windows program cannot make assumptions about standard path names because on systems with multiple disk drives the drive-name part of the path may be different.

Chmod: Windows uses access control lists.

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

30. Every system architecture has its own set of instructions that it can execute. Thus a Pentium cannot execute SPARC programs and a SPARC cannot execute Pentium programs. Also, different architectures differ in bus architecture used (such as VME, ISA, PCI, MCA, SBus, ...) as well as the word size of the CPU (usually 32 or 64 bit). Because of these differences in hardware, it is not feasible to build an operating system that is completely portable. A highly portable operating system will consist of two high-level layers—-a machine-dependent layer and a machine-independent layer. The machine-dependent layer addresses the specifics of the hardware and must be implemented separately for every architecture. This layer provides a uniform interface on which the machine-independent layer is built. The machine-independent layer has to be implemented only once. To be highly portable, the size of the machine-dependent layer must be kept as small as possible.

31. Separation of policy and mechanism allows OS designers to implement a small number of basic primitives in the kernel. These primitives are simplified, because they are not dependent of any specific policy. They can then be used to implement more complex mechanisms and policies at the user level.

32. The virtualization layer introduces increased memory usage and processor overhead as well as increased performance overhead.

33. The conversions are straightforward:

(a) A nanoyear is $10^{-9} \times 365 \times 24 \times 3600 = 31.536$ msec.
(b) 1 meter
(c) There are 2^{50} bytes, which is 1,099,511,627,776 bytes.
(d) It is 6×10^{24} kg or 6×10^{27} g.

SOLUTIONS TO CHAPTER 2 PROBLEMS

1. The transition from blocked to running is conceivable. Suppose that a process is blocked on I/O and the I/O finishes. If the CPU is otherwise idle, the process could go directly from blocked to running. The other missing transition, from ready to blocked, is impossible. A ready process cannot do I/O or anything else that might block it. Only a running process can block.

2. You could have a register containing a pointer to the current process-table entry. When I/O completed, the CPU would store the current machine state in the current process-table entry. Then it would go to the interrupt vector for the interrupting device and fetch a pointer to another process-table entry (the service procedure). This process would then be started up.

3. Generally, high-level languages do not allow the kind of access to CPU hardware that is required. For instance, an interrupt handler may be required to enable and disable the interrupt servicing a particular device, or to manipulate data within a process' stack area. Also, interrupt service routines must execute as rapidly as possible.

4. There are several reasons for using a separate stack for the kernel. Two of them are as follows. First, you do not want the operating system to crash because a poorly written user program does not allow for enough stack space. Second, if the kernel leaves stack data in a user program's memory space upon return from a system call, a malicious user might be able to use this data to find out information about other processes.

5. The chance that all five processes are idle is 1/32, so the CPU idle time is 1/32.

6. There is enough room for 14 processes in memory. If a process has an I/O of p, then the probability that they are all waiting for I/O is p^{14}. By equating this to 0.01, we get the equation $p^{14} = 0.01$. Solving this, we get $p = 0.72$, so we can tolerate processes with up to 72% I/O wait.

7. If each job has 50% I/O wait, then it will take 40 minutes to complete in the absence of competition. If run sequentially, the second one will finish 80 minutes after the first one starts. With two jobs, the approximate CPU utilization is $1 - 0.5^2$. Thus, each one gets 0.375 CPU minute per minute of real time. To accumulate 20 minutes of CPU time, a job must run for 20/0.375 minutes, or about 53.33 minutes. Thus running sequentially the jobs finish after 80 minutes, but running in parallel they finish after 53.33 minutes.

8. The probability that all processes are waiting for I/O is 0.4^6 which is 0.004096. Therefore, CPU utilization = $1 - 0.004096 = 0{:}\,995904$.

9. The client process can create separate threads; each thread can fetch a different part of the file from one of the mirror servers. This can help reduce downtime. Of course, there is a single network link being shared by all threads. This link can become a bottleneck as the number of threads becomes very large.

10. It would be difficult, if not impossible, to keep the file system consistent. Suppose that a client process sends a request to server process 1 to update a file. This process updates the cache entry in its memory. Shortly thereafter, another client process sends a request to server 2 to read that file. Unfortunately, if the file is also cached there, server 2, in its innocence, will return obsolete data. If the first process writes the file through to the disk after caching it, and server 2 checks the disk on every read to see if its cached copy is up-to-date, the system can be made to work, but it is precisely all these disk accesses that the caching system is trying to avoid.

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

12. A worker thread will block when it has to read a Web page from the disk. If user-level threads are being used, this action will block the entire process, destroying the value of multithreading. Thus it is essential that kernel threads are used to permit some threads to block without affecting the others.

13. Yes. If the server is entirely CPU bound, there is no need to have multiple threads. It just adds unnecessary complexity. As an example, consider a telephone directory assistance number (like 555-1212) for an area with 1 million people. If each (name, telephone number) record is, say, 64 characters, the entire database takes 64 megabytes and can easily be kept in the server's memory to provide fast lookup.

14. When a thread is stopped, it has values in the registers. They must be saved, just as when the process is stopped. the registers must be saved. Multiprogramming threads is no different than multiprogramming processes, so each thread needs its own register save area.

15. Threads in a process cooperate. They are not hostile to one another. If yielding is needed for the good of the application, then a thread will yield. After all, it is usually the same programmer who writes the code for all of them.

16. User-level threads cannot be preempted by the clock unless the whole process' quantum has been used up (although transparent clock interrupts can happen). Kernel-level threads can be preempted individually. In the latter case, if a thread runs too long, the clock will interrupt the current process and thus the current thread. The kernel is free to pick a different thread from the same process to run next if it so desires.

17. In the single-threaded case, the cache hits take 12 msec and cache misses take 87 msec. The weighted average is $2/3 \times 12 + 1/3 \times 87$. Thus, the mean request takes 37 msec and the server can do about 27 per second. For a multithreaded server, all the waiting for the disk is overlapped, so every request takes 12 msec, and the server can handle 83 1/3 requests per second.

18. The biggest advantage is the efficiency. No traps to the kernel are needed to switch threads. The biggest disadvantage is that if one thread blocks, the entire process blocks.

19. Yes, it can be done. After each call to *pthread_create*, the main program could do a *pthread_join* to wait until the thread just created has exited before creating the next thread.

20. The pointers are really necessary because the size of the global variable is unknown. It could be anything from a character to an array of floating-point numbers. If the value were stored, one would have to give the size to *create_global*, which is all right, but what type should the second parameter of *set_global* be, and what type should the value of *read_global* be?

21. It could happen that the runtime system is precisely at the point of blocking or unblocking a thread, and is busy manipulating the scheduling queues. This would be a very inopportune moment for the clock interrupt handler to begin inspecting those queues to see if it was time to do thread switching, since they might be in an inconsistent state. One solution is to set a flag when the runtime system is entered. The clock handler would see this and set its own flag, then return. When the runtime system finished, it would check the clock flag, see that a clock interrupt occurred, and now run the clock handler.

22. Yes it is possible, but inefficient. A thread wanting to do a system call first sets an alarm timer, then does the call. If the call blocks, the timer returns control to the threads package. Of course, most of the time the call will not block, and the timer has to be cleared. Thus each system call that might block has to be executed as three system calls. If timers go off prematurely, all kinds of problems develop. This is not an attractive way to build a threads package.

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

24. It certainly works with preemptive scheduling. In fact, it was designed for that case. When scheduling is nonpreemptive, it might fail. Consider the case in which *turn* is initially 0 but process 1 runs first. It will just loop forever and never release the CPU.

25. The priority inversion problem occurs when a low-priority process is in its critical region and suddenly a high-priority process becomes ready and is scheduled. If it uses busy waiting, it will run forever. With user-level threads, it cannot happen that a low-priority thread is suddenly preempted to allow a

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

26. With round-robin scheduling it works. Sooner or later *L* will run, and eventually it will leave its critical region. The point is, with priority scheduling, *L* never gets to run at all; with round robin, it gets a normal time slice periodically, so it has the chance to leave its critical region.

27. Each thread calls procedures on its own, so it must have its own stack for the local variables, return addresses, and so on. This is equally true for user-level threads as for kernel-level threads.

28. Yes. The simulated computer could be multiprogrammed. For example, while process *A* is running, it reads out some shared variable. Then a simulated clock tick happens and process *B* runs. It also reads out the same variable. Then it adds 1 to the variable. When process *A* runs, if it also adds 1 to the variable, we have a race condition.

29. Yes, it will work as is. At a given time instant, only one producer (consumer) can add (remove) an item to (from) the buffer.

30. The solution satisfies mutual exclusion since it is not possible for both processes to be in their critical section. That is, when turn is 0, *P0* can execute its critical section, but not *P1*. Likewise, when turn is 1. However, this assumes *P0* must run first. If *P1* produces something and it puts it in a buffer, then while *P0* can get into its critical section, it will find the buffer empty and block. Also, this solution requires strict alternation of the two processes, which is undesirable.

31. To do a semaphore operation, the operating system first disables interrupts. Then it reads the value of the semaphore. If it is doing a down and the semaphore is equal to zero, it puts the calling process on a list of blocked processes associated with the semaphore. If it is doing an up, it must check to see if any processes are blocked on the semaphore. If one or more processes are blocked, one of them is removed from the list of blocked processes and made runnable. When all these operations have been completed, interrupts can be enabled again.

32. Associated with each counting semaphore are two binary semaphores, *M*, used for mutual exclusion, and *B*, used for blocking. Also associated with each counting semaphore is a counter that holds the number of ups minus the number of downs, and a list of processes blocked on that semaphore. To implement down, a process first gains exclusive access to the semaphores, counter, and list by doing a down on *M*. It then decrements the counter. If it is zero or more, it just does an up on *M* and exits. If *M* is negative, the process is put on the list of blocked processes. Then an up is done on *M* and a down is done on *B* to block the process. To implement up, first *M* is downed to get mutual

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

33. If the program operates in phases and neither process may enter the next phase until both are finished with the current phase, it makes perfect sense to use a barrier.

34. With kernel threads, a thread can block on a semaphore and the kernel can run some other thread in the same process. Consequently, there is no problem using semaphores. With user-level threads, when one thread blocks on a semaphore, the kernel thinks the entire process is blocked and does not run it ever again. Consequently, the process fails.

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

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

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

38. It will take nT sec.

39. Three processes are created. After the initial process forks, there are two processes running, a parent and a child. Each of them then forks, creating two additional processes. Then all the processes exit.

40. If a process occurs multiple times in the list, it will get multiple quanta per cycle. This approach could be used to give more important processes a larger share of the CPU. But when the process blocks, all entries had better be removed from the list of runnable processes.

41. In simple cases it may be possible to see if I/O will be limiting by looking at source code. For instance a program that reads all its input files into buffers at the start will probably not be I/O bound, but a 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 the total time to complete execution of the program. This is, of course, most meaningful on a system where you are the only user.

42. If the context switching time is large, then the time quantum value has to be proportionally large. Otherwise, the overhead of context switching can be quite high. Choosing large time quantum values can lead to an inefficient system if the typical CPU burst times are less than the time quantum. If context switching is very small or negligible, then the time quantum value can be chosen with more freedom.

43. The CPU efficiency is the useful CPU time divided by the total CPU time. When $Q \geq T$, the basic cycle is for the process to run for T and undergo a process switch for S. Thus, (a) and (b) have an efficiency of $T/(S + T)$. When the quantum is shorter than T, each run of T will require T/Q process switches, wasting a time ST/Q. The efficiency here is then

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

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

44. Shortest job first is the way to minimize average response time.

$0 < X \leq 3$: X, 3, 5, 6, 9.
$3 < X \leq 5$: 3, X, 5, 6, 9.
$5 < X \leq 6$: 3, 5, X, 6, 9.
$6 < X \leq 9$: 3, 5, 6, X, 9.
$X > 9$: 3, 5, 6, 9, X.

45. For round robin, during the first 10 minutes each job gets 1/5 of the CPU. At the end of 10 minutes, C finishes. During the next 8 minutes, each job gets 1/4 of the CPU, after which time D finishes. Then each of the three remaining jobs gets 1/3 of the CPU for 6 minutes, until B finishes, and so on. The finishing times for the five jobs are 10, 18, 24, 28, and 30, for an average of 22 minutes. For priority scheduling, B is run first. After 6 minutes it is finished. The other jobs finish at 14, 24, 26, and 30, for an average of 18.8 minutes. If the jobs run in the order A through E, they finish at 10, 16, 18, 22, and 30, for an average of 19.2 minutes. Finally, shortest job first yields finishing times of 2, 6, 12, 20, and 30, for an average of 14 minutes.

46. The first time it gets 1 quantum. On succeeding runs it gets 2, 4, 8, and 15, so it must be swapped in 5 times.

47. Each voice call needs 200 samples of 1 msec or 200 msec. Together they use 400 msec of CPU time. The video needs 11 msec 33 1/3 times a second for a total of about 367 msec. The sum is 767 msec per second of real time so the system is schedulable.

48. Another video stream consumes 367 msec of time per second for a total of 1134 msec per second of real time so the system is not schedulable.

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

50. The fraction of the CPU used is 35/50 + 20/100 + 10/200 + x/250. To be schedulable, this must be less than 1. Thus x must be less than 12.5 msec.

51. Yes. There will be always at least one fork free and at least one philosopher that can obtain both forks simultaneously. Hence, there will be no deadlock. You can try this for $N = 2$, $N = 3$ and $N = 4$ and then generalize.

52. Each voice call runs 166.67 times/second and uses up 1 msec per burst, so each voice call needs 166.67 msec per second or 333.33 msec for the two of them. The video runs 25 times a second and uses up 20 msec each time, for a total of 500 msec per second. Together they consume 833.33 msec per second, so there is time left over and the system is schedulable.

53. The kernel could schedule processes by any means it wishes, but within each process it runs threads strictly in priority order. By letting the user process set the priority of its own threads, the user controls the policy but the kernel handles the mechanism.

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

55. The change would mean that after a philosopher stopped eating, neither of his neighbors could be chosen next. In fact, they would never be chosen. Suppose that philosopher 2 finished eating. He would run *test* for philosophers 1 and 3, and neither would be started, even though both were hungry and both forks were available. Similarly, if philosopher 4 finished eating, philosopher 3 would not be started. Nothing would start him.

56. Variation 1: readers have priority. No writer may start when a reader is active. When a new reader appears, it may start immediately unless a writer is currently active. When a writer finishes, if readers are waiting, they are all started, regardless of the presence of waiting writers. Variation 2: Writers have priority. No reader may start when a writer is waiting. When the last active process finishes, a writer is started, if there is one; otherwise, all the readers (if any) are started. Variation 3: symmetric version. When a reader is active, new readers may start immediately. When a writer finishes, a new writer has priority, if one is waiting. In other words, once we have started reading, we keep reading until there are no readers left. Similarly, once we have started writing, all pending writers are allowed to run.

57. A possible shell script might be

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

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

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

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

SOLUTIONS TO CHAPTER 3 PROBLEMS

1. First, special hardware is needed to do the comparisons, and it must be fast, since it is used on every memory reference. Second, with 4-bit keys, only 16 programs can be in memory at once (one of which is the operating system).

2. It is an accident. The base register is 16,384 because the program happened to be loaded at address 16,384. It could have been loaded anywhere. The limit register is 16,384 because the program contains 16,384 bytes. It could have been any length. That the load address happens to exactly match the program length is pure coincidence.

3. Almost the entire memory has to be copied, which requires each word to be read and then rewritten at a different location. Reading 4 bytes takes 4 nsec, so reading 1 byte takes 1 nsec and writing it takes another 2 nsec, for a total of 2 nsec per byte compacted. This is a rate of 500,000,000 bytes/sec. To copy 4 GB (2^{232} bytes, which is about 4.295×10^9 bytes), the computer needs $2^{32}/500,000,000$ sec, which is about 859 msec. This number is slightly pessimistic because if the initial hole at the bottom of memory is k bytes, those k bytes do not need to be copied. However, if there are many holes and many data segments, the holes will be small, so k will be small and the error in the calculation will also be small.

4. First fit takes 20 MB, 10 MB, 18 MB. Best fit takes 12 MB, 10 MB, and 9 MB. Worst fit takes 20 MB, 18 MB, and 15 MB. Next fit takes 20 MB, 18 MB, and 9 MB.

5. Real memory uses physical addresses. These are the numbers that the memory chips react to on the bus. Virtual addresses are the logical addresses that refer to a process' address space. Thus a machine with a 32-bit word can generate virtual addresses up to 4 GB regardless of whether the machine has more or less memory than 4 GB.

6. For a 4-KB page size the (page, offset) pairs are (4, 3616), (8, 0), and (14, 2656). For an 8-KB page size they are (2, 3616), (4, 0), and (7, 2656).

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

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

9. There needs to be an MMU that can remap virtual pages to physical pages. Also, when a page not currently mapped is referenced, there needs to be a trap to the operating system so it can fetch the page.

10. If the smartphone supports multiprogramming, which the iPhone, Android, and Windows phones all do, then multiple processes are supported. If a process forks and pages are shared between parent and child, copy on write definitely makes sense. A smartphone is smaller than a server, but logically it is not so different.

11. For these sizes

(a) M has to be at least 4096 to ensure a TLB miss for every access to an element of X. Since N affects only how many times X is accessed, any value of N will do.

(b) M should still be at least 4,096 to ensure a TLB miss for every access to an element of X. But now N should be greater than 64K to thrash the TLB, that is, X should exceed 256 KB.

12. The total virtual address space for all the processes combined is nv, so this much storage is needed for pages. However, an amount r can be in RAM, so the amount of disk storage required is only $nv - r$. This amount is far more than is ever needed in practice because rarely will there be n processes actually running and even more rarely will all of them need the maximum allowed virtual memory.

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

14. The page table contains $2^{32}/2^{13}$ entries, which is 524,288. Loading the page table takes 52 msec. If a process gets 100 msec, this consists of 52 msec for loading the page table and 48 msec for running. Thus 52% of the time is spent loading page tables.

15. Under these circumstances:

(a) We need one entry for each page, or $2^{24} = 16 \times 1024 \times 1024$ entries, since there are $36 = 48 - 12$ bits in the page number field.

(b) Instruction addresses will hit 100% in the TLB. The data pages will have a 100 hit rate until the program has moved onto the next data page. Since a 4-KB page contains 1,024 long integers, there will be one TLB miss and one extra memory access for every 1,024 data references.

16. The chance of a hit is 0.99 for the TLB, 0.0099 for the page table, and 0.0001 for a page fault (i.e., only 1 in 10,000 references will cause a page fault). The effective address translation time in nsec is then:

$$0.99 \times 1 + 0.0099 \times 100 + 0.0001 \times 6 \times 10^6 \approx 602 \text{ clock cycles.}$$

Note that the effective address translation time is quite high because it is dominated by the page replacement time even when page faults only occur once in 10,000 references.

17. Consider,

(a) A multilevel page table reduces the number of actual pages of the page table that need to be in memory because of its hierarchic structure. In fact, in a program with lots of instruction and data locality, we only need the top-level page table (one page), one instruction page, and one data page.

(b) Allocate 12 bits for each of the three page fields. The offset field requires 14 bits to address 16 KB. That leaves 24 bits for the page fields. Since each entry is 4 bytes, one page can hold 2^{12} page table entries and therefore requires 12 bits to index one page. So allocating 12 bits for each of the page fields will address all 2^{38} bytes.

18. The virtual address was changed from (PT1, PT2, Offset) to (PT1, PT2, PT3, Offset). But the virtual address still used only 32 bits. The bit configuration of a virtual address changed from (10, 10, 12) to (2, 9, 9, 12)

19. Twenty bits are used for the virtual page numbers, leaving 12 over for the offset. This yields a 4-KB page. Twenty bits for the virtual page implies 2^{20} pages.

20. For a one-level page table, there are $2^{32}/2^{12}$ or 1M pages needed. Thus the page table must have 1M entries. For two-level paging, the main page table has 1K entries, each of which points to a second page table. Only two of these are used. Thus, in total only three page table entries are needed, one in the top-level table and one in each of the lower-level tables.

21. The code and reference string are as follows

LOAD 6144,R0	1(I), 12(D)
PUSH R0	2(I), 15(D)
CALL 5120	2(I), 15(D)
JEQ 5152	10(I)

The code (I) indicates an instruction reference, whereas (D) indicates a data reference.

22. The effective instruction time is $1h + 5(1 - h)$, where h is the hit rate. If we equate this formula with 2 and solve for h, we find that h must be at least 0.75.

23. An associative memory essentially compares a key to the contents of multiple registers simultaneously. For each register there must be a set of comparators that compare each bit in the register contents to the key being searched for. The number of gates (or transistors) needed to implement such a device is a linear function of the number of registers, so expanding the design gets expensive linearly.

24. With 8-KB pages and a 48-bit virtual address space, the number of virtual pages is $2^{48}/2^{13}$, which is 2^{35} (about 34 billion).

25. The main memory has $2^{28}/2^{13}$ = 32,768 pages. A 32K hash table will have a mean chain length of 1. To get under 1, we have to go to the next size, 65,536 entries. Spreading 32,768 entries over 65,536 table slots will give a mean chain length of 0.5, which ensures fast lookup.

26. This is probably not possible except for the unusual and not very useful case of a program whose course of execution is completely predictable at compilation time. If a compiler collects information about the locations in the code of calls to procedures, this information might be used at link time to rearrange the object code so that procedures were located close to the code that calls them. This would make it more likely that a procedure would be on the same page as the calling code. Of course this would not help much for procedures called from many places in the program.

27. Under these circumstances

(a) Every reference will page fault unless the number of page frames is 512, the length of the entire sequence.

(b) If there are 500 frames, map pages 0–498 to fixed frames and vary only one frame.

28. The page frames for FIFO are as follows:

x0172333300
xx017222233
xxx01777722
xxxx0111177

The page frames for LRU are as follows:

x0172327103
xx017232710
xxx01773271
xxxx0111327

FIFO yields six page faults; LRU yields seven.

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

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

31. The sequence: 0, 1, 2, 1, 2, 0, 3. In LRU, page 1 will be replaced by page 3. In clock, page 1 will be replaced, since all pages will be marked and the cursor is at page 0.

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

33. Consider,

(a) For every *R* bit that is set, set the time-stamp value to 10 and clear all *R* bits. You could also change the (0,1) *R*-*M* entries to (0,0*). So the entries for pages 1 and 2 will change to:

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

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

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

34. Consider,

(a) The attributes are: (FIFO) load time; (LRU) latest reference time; and (Optimal) nearest reference time in the future.

(b) There is the labeling algorithm and the replacement algorithm. The labeling algorithm labels each page with the attribute given in part a. The replacement algorithm evicts the page with the smallest label.

35. The seek plus rotational latency is 10 msec. For 2-KB pages, the transfer time is about 0.009766 msec, for a total of about 10.009766 msec. Loading 32 of these pages will take about 320.21 msec. For 4-KB pages, the transfer time is doubled to about 0.01953 msec, so the total time per page is 10.01953 msec. Loading 16 of these pages takes about 160.3125 msec. With such fast disks, all that matters is reducing the number of transfers (or putting the pages consecutively on the disk).

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

37. Sharing pages brings up all kinds of complications and options:

(a) The page table update should be delayed for process *B* if it will never access the shared page or if it accesses it when the page has been swapped out again. Unfortunately, in the general case, we do not know what process *B* will do in the future.

(b) The cost is that this lazy page fault handling can incur more page faults. The overhead of each page fault plays an important role in determining if this strategy is more efficient. (*Aside*: This cost is similar to that faced by the copy-on-write strategy for supporting some UNIX fork system call implementations.)

38. Fragment *B* since the code has more spatial locality than Fragment *A*. The inner loop causes only one page fault for every other iteration of the outer loop. (There will be only 32 page faults.) [*Aside* (Fragment *A*): Since a frame is 128 words, one row of the *X* array occupies half of a page (i.e., 64 words). The entire array fits into $64 \times 32/128 = 16$ frames. The inner loop of the code steps through consecutive rows of *X* for a given column. Thus, every other reference to $X[i][j]$ will cause a page fault. The total number of page faults will be $64 \times 64/2 = 2,048$].

39. It can certainly be done.

(a) The approach has similarities to using flash memory as a paging device in smartphones except now the virtual swap area is a RAM located on a remote server. All of the software infrastructure for the virtual swap area would have to be developed.

(b) The approach might be worthwhile by noting that the access time of disk drives is in the millisecond range while the access time of RAM via a network connection is in the microsecond range if the software overhead is not too high. But the approach might make sense only if there is lots of idle RAM in the server farm. And then, there is also the issue of reliability. Since RAM is volatile, the virtual swap area would be lost if the remote server went down.

40. The PDP-1 paging drum had the advantage of no rotational latency. This saved half a rotation each time memory was written to the drum.

41. The text is eight pages, the data are five pages, and the stack is four pages. The program does not fit because it needs 17 4096-byte pages. With a 512-byte page, the situation is different. Here the text is 64 pages, the data are 33 pages, and the stack is 31 pages, for a total of 128 512-byte pages, which fits. With the small page size it is OK, but not with the large one.

42. The program is getting 15,000 page faults, each of which uses 2 msec of extra processing time. Together, the page fault overhead is 30 sec. This means that of the 60 sec used, half was spent on page fault overhead, and half on running the program. If we run the program with twice as much memory, we get half as many memory page faults, and only 15 sec of page fault overhead, so the total run time will be 45 sec.

43. It works for the program if the program cannot be modified. It works for the data if the data cannot be modified. However, it is common that the program cannot be modified and extremely rare that the data cannot be modified. If the data area on the binary file were overwritten with updated pages, the next time the program was started, it would not have the original data.

44. The instruction could lie astride a page boundary, causing two page faults just to fetch the instruction. The word fetched could also span a page boundary, generating two more faults, for a total of four. If words must be aligned in memory, the data word can cause only one fault, but an instruction to load a 32-bit word at address 4094 on a machine with a 4-KB page is legal on some machines (including the x86).

45. Internal fragmentation occurs when the last allocation unit is not full. External fragmentation occurs when space is wasted between two allocation units. In a paging system, the wasted space in the last page is lost to internal fragmentation. In a pure segmentation system, some space is invariably lost between the segments. This is due to external fragmentation.

46. No. The search key uses both the segment number and the virtual page number, so the exact page can be found in a single match.

47. Here are the results:

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

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

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

SOLUTIONS TO CHAPTER 4 PROBLEMS

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

```
/etc/passwd
/./etc/passwd
/./../etc/passwd
/./../../etc/passwd
/etc/../etc/passwd
/etc/../etc/../etc/passwd
/etc/../etc/../etc/../etc/passwd
/etc/../etc/../etc/../etc/../etc/passwd
```

2. The Windows way is to use the file extension. Each extension corresponds to a file type and to some program that handles that type. Another way is to remember which program created the file and run that program. The Macintosh works this way.

3. These systems loaded the program directly in memory and began executing at word 0, which was the magic number. To avoid trying to execute the header as code, the magic number was a BRANCH instruction with a target address just above the header. In this way it was possible to read the binary file directly into the new process' address space and run it at 0, without even knowing how big the header was.

4. To start with, if there were no open, on every read it would be necessary to specify the name of the file to be opened. The system would then have to fetch the i-node for it, although that could be cached. One issue that quickly arises is when to flush the i-node back to disk. It could time out, however. It would be a bit clumsy, but it might work.

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

6. Yes. The rename call does not change the creation time or the time of last modification, but creating a new file causes it to get the current time as both the creation time and the time of last modification. Also, if the disk is nearly full, the copy might fail.

7. The mapped portion of the file must start at a page boundary and be an integral number of pages in length. Each mapped page uses the file itself as backing store. Unmapped memory uses a scratch file or partition as backing store.

8. Use file names such as */usr/ast/file*. While it looks like a hierarchical path name, it is really just a single name containing embedded slashes.

9. One way is to add an extra parameter to the read system call that tells what address to read from. In effect, every read then has a potential for doing a seek within the file. The disadvantages of this scheme are (1) an extra parameter in every read call, and (2) requiring the user to keep track of where the file pointer is.

10. The dotdot component moves the search to */usr*, so *../ast* puts it in */usr/ast*. Thus *../ast/x* is the same as */usr/ast/x*.

11. Since the wasted storage is *between* the allocation units (files), not inside them, this is external fragmentation. It is precisely analogous to the external fragmentation of main memory that occurs with a swapping system or a system using pure segmentation.

12. If a data block gets corrupted in a contiguous allocation system, then only this block is affected; the remainder of the file's blocks can be read. In the case of linked allocation, the corrupted block cannot be read; also, location data about all blocks starting from this corrupted block is lost. In case of indexed allocation, only the corrupted data block is affected.

13. It takes 9 msec to start the transfer. To read 2^{13} bytes at a transfer rate of 80 MB/sec requires 0.0977 msec, for a total of 9.0977 msec. Writing it back takes another 9.0977 msec. Thus, copying a file takes 18.1954 msec. To compact half of a 16-GB disk would involve copying 8 GB of storage, which is 2^{20} files. At 18.1954 msec per file, this takes 19,079.25 sec, which is 5.3 hours. Clearly, compacting the disk after every file removal is not a great idea.

14. If done right, yes. While compacting, each file should be organized so that all of its blocks are consecutive, for fast access. Windows has a program that defragments and reorganizes the disk. Users are encouraged to run it periodically to improve system performance. But given how long it takes, running once a month might be a good frequency.

15. A digital still camera records some number of photographs in sequence on a nonvolatile storage medium (e.g., flash memory). When the camera is reset, the medium is emptied. Thereafter, pictures are recorded one at a time in sequence until the medium is full, at which time they are uploaded to a hard disk. For this application, a contiguous file system inside the camera (e.g., on the picture storage medium) is ideal.

16. The indirect block can hold 128 disk addresses. Together with the 10 direct disk addresses, the maximum file has 138 blocks. Since each block is 1 KB, the largest file is 138 KB.

17. For random access, table/indexed and contiguous will be both appropriate, while linked allocation is not as it typically requires multiple disk reads for a given record.

18. Since the file size changes a lot, contiguous allocation will be inefficient requiring reallocation of disk space as the file grows in size and compaction of free blocks as the file shrinks in size. Both linked and table/indexed allocation will be efficient; between the two, table/indexed allocation will be more efficient for random-access scenarios.

19. There must be a way to signal that the address-block pointers hold data, rather than pointers. If there is a bit left over somewhere among the attributes, it can be used. This leaves all nine pointers for data. If the pointers are k bytes each, the stored file could be up to $9k$ bytes long. If no bit is left over among the attributes, the first disk address can hold an invalid address to mark the following bytes as data rather than pointers. In that case, the maximum file is $8k$ bytes.

20. Elinor is right. Having two copies of the i-node in the table at the same time is a disaster, unless both are read only. The worst case is when both are being updated simultaneously. When the i-nodes are written back to the disk, whichever one gets written last will erase the changes made by the other one, and disk blocks will be lost.

21. Hard links do not require any extra disk space, just a counter in the i-node to keep track of how many there are. Symbolic links need space to store the name of the file pointed to. Symbolic links can point to files on other machines, even over the Internet. Hard links are restricted to pointing to files within their own partition.

22. A single i-node is pointed to by all directory entries of hard links for a given file. In the case of soft-links, a new i-node is created for the soft link and this inode essentially points to the original file being linked.

23. The number of blocks on the disk = 4 TB / 4 KB = $2^{30.}$ Thus, each block address can be 32 bits (4 bytes), the nearest power of 2. Thus, each block can store 4 KB / 4 = 1024 addresses.

24. The bitmap requires B bits. The free list requires DF bits. The free list requires fewer bits if $DF < B$. Alternatively, the free list is shorter if $F/B < 1/D$, where F/B is the fraction of blocks free. For 16-bit disk addresses, the free list is shorter if 6% or less of the disk is free.

25. The beginning of the bitmap looks like:

(a) After writing file *B*: 1111 1111 1111 0000
(b) After deleting file *A*: 1000 0001 1111 0000
(c) After writing file *C*: 1111 1111 1111 1100
(d) After deleting file *B*: 1111 1110 0000 1100

26. It is not a serious problem at all. Repair is straightforward; it just takes time. The recovery algorithm is to make a list of all the blocks in all the files and take the complement as the new free list. In UNIX this can be done by scanning all the i-nodes. In the FAT file system, the problem cannot occur because there is no free list. But even if there were, all that would have to be done to recover it is to scan the FAT looking for free entries.

27. Ollie's thesis may not be backed up as reliably as he might wish. A backup program may pass over a file that is currently open for writing, as the state of the data in such a file may be indeterminate.

28. They must keep track of the time of the last dump in a file on disk. At every dump, an entry is appended to this file. At dump time, the file is read and the time of the last entry noted. Any file changed since that time is dumped.

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

30. Many UNIX files are short. If the entire file fits in the same block as the i-node, only one disk access would be needed to read the file, instead of two, as is presently the case. Even for longer files there would be a gain, since one fewer disk accesses would be needed.

31. It should not happen, but due to a bug somewhere it could happen. It means that some block occurs in two files and also twice in the free list. The first step in repairing the error is to remove both copies from the free list. Next a free block has to be acquired and the contents of the sick block copied there. Finally, the occurrence of the block in one of the files should be changed to refer to the newly acquired copy of the block. At this point the system is once again consistent.

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

33. In this case, it is better to use a write-through cache since it writes data to the hard drive while also updating the cache. This will ensure that the updated file is always on the external hard drive even if the user accidentally removes the hard drive before disk sync is completed.

34. The block read-ahead technique reads blocks sequentially, ahead of their use, in order to improve performance. In this application, the records will likely not be accessed sequentially since the user can input any student ID at a given instant. Thus, the read-ahead technique will not be very useful in this scenario.

35. The blocks allotted to f1 are: 22, 19, 15, 17, 21.
The blocks allotted to f2 are: 16, 23, 14, 18, 20.

36. At 15,000 rpm, the disk takes 4 msec to go around once. The average access time (in msec) to read k bytes is then $6 + 2 + (k/1,048,576) \times 4$. For blocks of 1 KB, 2 KB, and 4 KB, the access times are about 6.0039 msec, 6.0078 msec, and 6.0156 msec, respectively (hardly any different). These give rates of about 170.556 KB/sec, 340.890 KB/sec, and 680.896 KB/sec, respectively.

37. If all files were 1 KB, then each 4-KB block would contain one file and 3 KB of wasted space. Trying to put two files in a block is not allowed because the unit used to keep track of data is the block, not the semiblock. This leads to 75% wasted space. In practice, every file system has large files as well as many small ones, and these files use the disk much more efficiently. For example, a 32,769-byte file would use 9 disk blocks for storage, given a space efficiency of 32,769/36,864, which is about 89%.

38. The indirect block can hold 1024 addresses. Added to the 10 direct addresses, there are 1034 addresses in all. Since each one points to a 4-KB disk block, the largest file is 4,235,264 bytes

39. It constrains the sum of all the file lengths to being no larger than the disk. This is not a very serious constraint. If the files were collectively larger than the disk, there would be no place to store all of them on the disk.

40. The i-node holds 10 pointers. The single indirect block holds 1024 pointers. The double indirect block is good for 1024^2 pointers. The triple indirect block is good for 1024^3 pointers. Adding these up, we get a maximum file size of 1,074,791,434 blocks, which is about 16.06 GB.

41. The following disk reads are needed:

directory for /
i-node for */usr*
directory for */usr*
i-node for */usr/ast*
directory for */usr/ast*
i-node for */usr/ast/courses*
directory for */usr/ast/courses*
i-node for */usr/ast/courses/os*
directory for */usr/ast/courses/os*
i-node for */usr/ast/courses/os/handout.t*

In total, 10 disk reads are required.

42. Some pros are as follows. First, no disk space is wasted on unused i-nodes. Second, it is not possible to run out of i-nodes. Third, less disk movement is needed since the i-node and the initial data can be read in one operation. Some cons are as follows. First, directory entries will now need a 32-bit disk address instead of a 16-bit i-node number. Second, an entire disk will be used even for

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

SOLUTIONS TO CHAPTER 5 PROBLEMS

1. In the figure, we see controllers and devices as separate units. The reason is to allow a controller to handle multiple devices, and thus eliminate the need for having a controller per device. If controllers become almost free, then it will be simpler just to build the controller into the device itself. This design will also allow multiple transfers in parallel and thus give better performance.

2. Easy. The scanner puts out 400 KB/sec maximum. The wireless network runs at 6.75 MB/sec, so there is no problem at all.

3. It is not a good idea. The memory bus is surely faster than the I/O bus, otherwise why bother with it? Consider what happens with a normal memory request. The memory bus finishes first, but the I/O bus is still busy. If the CPU waits until the I/O bus finishes, it has reduced memory performance to that of the I/O bus. If it just tries the memory bus for the second reference, it will fail if this one is an I/O device reference. If there were some way to instantaneously abort the previous I/O bus reference to try the second one, the improvement might work, but there is never such an option. All in all, it is a bad idea.

4. An advantage of precise interrupts is simplicity of code in the operating system since the machine state is well defined. On the other hand, in imprecise interrupts, OS writers have to figure out what instructions have been partially executed and up to what point. However, precise interrupts increase complexity of chip design and chip area, which may result in slower CPU.

5. Each bus transaction has a request and a response, each taking 50 nsec, or 100 nsec per bus transaction. This gives 10 million bus transactions/sec. If each one is good for 4 bytes, the bus has to handle 40 MB/sec. The fact that these transactions may be sprayed over five I/O devices in round-robin fashion is irrelevant. A bus transaction takes 100 nsec, regardless of whether consecutive requests are to the same device or different devices, so the number of channels the DMA controller has does not matter. The bus does not know or care.

6. (a) Word-at-a-time mode: $1000 \times [(t_1 + t_2) + (t_1 + t_2) + (t_1 + t_2)]$
Where the first term is for acquiring the bus and sending the command to the disk controller, the second term is for transferring the word, and the third term is for the acknowledgement. All in all, a total of $3000 \times (t_1 + t_2)$ nsec.

(b) Burst mode: $(t_1 + t_2) + t_1 + 1000$ times $t_2 + (t_1 + t_2)$
where the first term is for acquiring the bus and sending the command to the disk controller, the second term is for the disk controller to acquire the bus, the third term is for the burst transfer, and the fourth term is for acquiring the bus and doing the acknowledgement. All in all, a total of $3t_1 + 1002t_2$.