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 $Q \to 0$ 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.