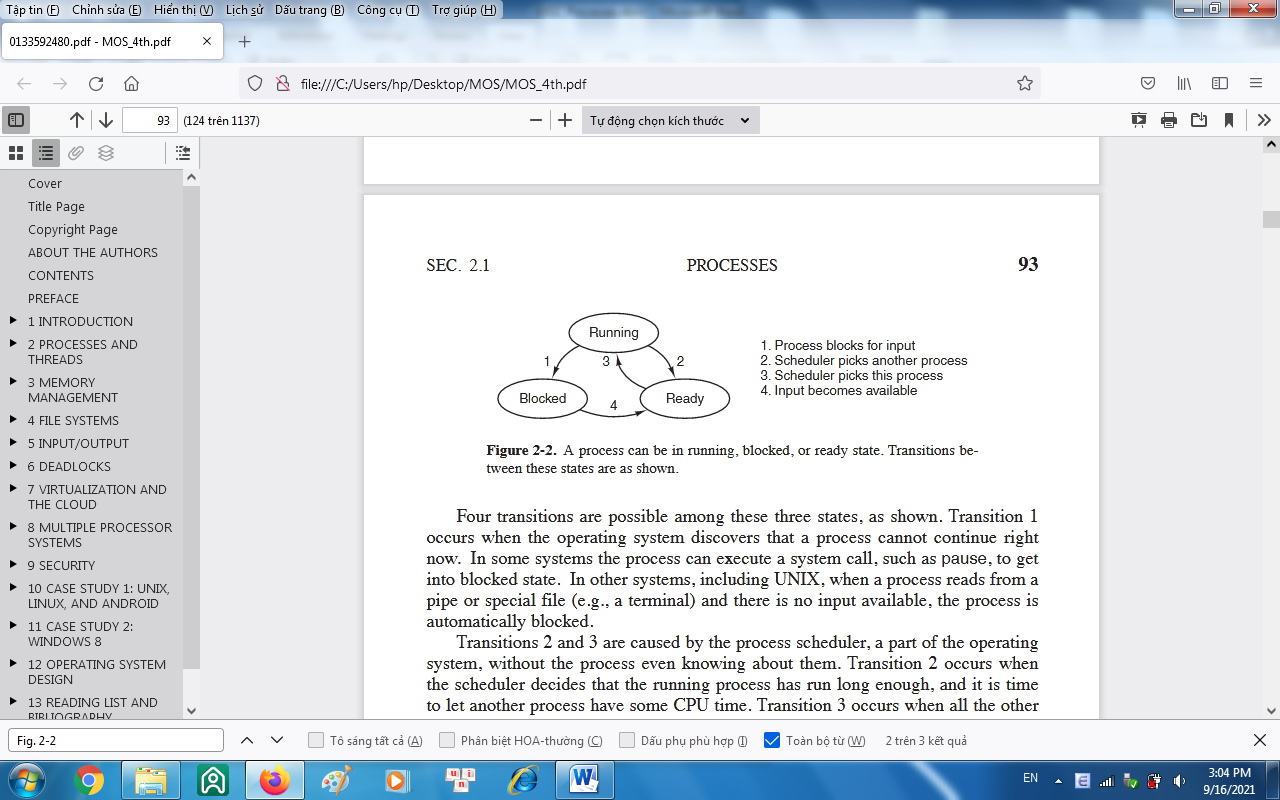
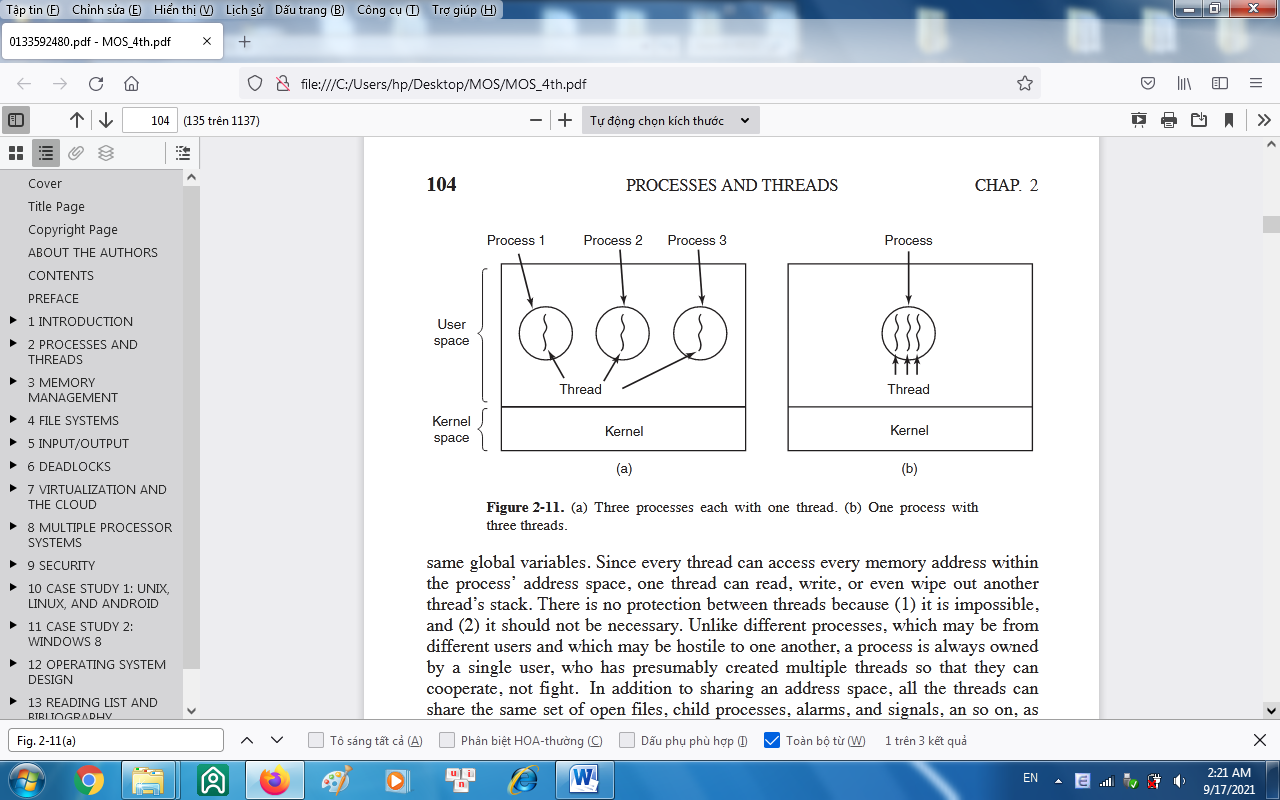
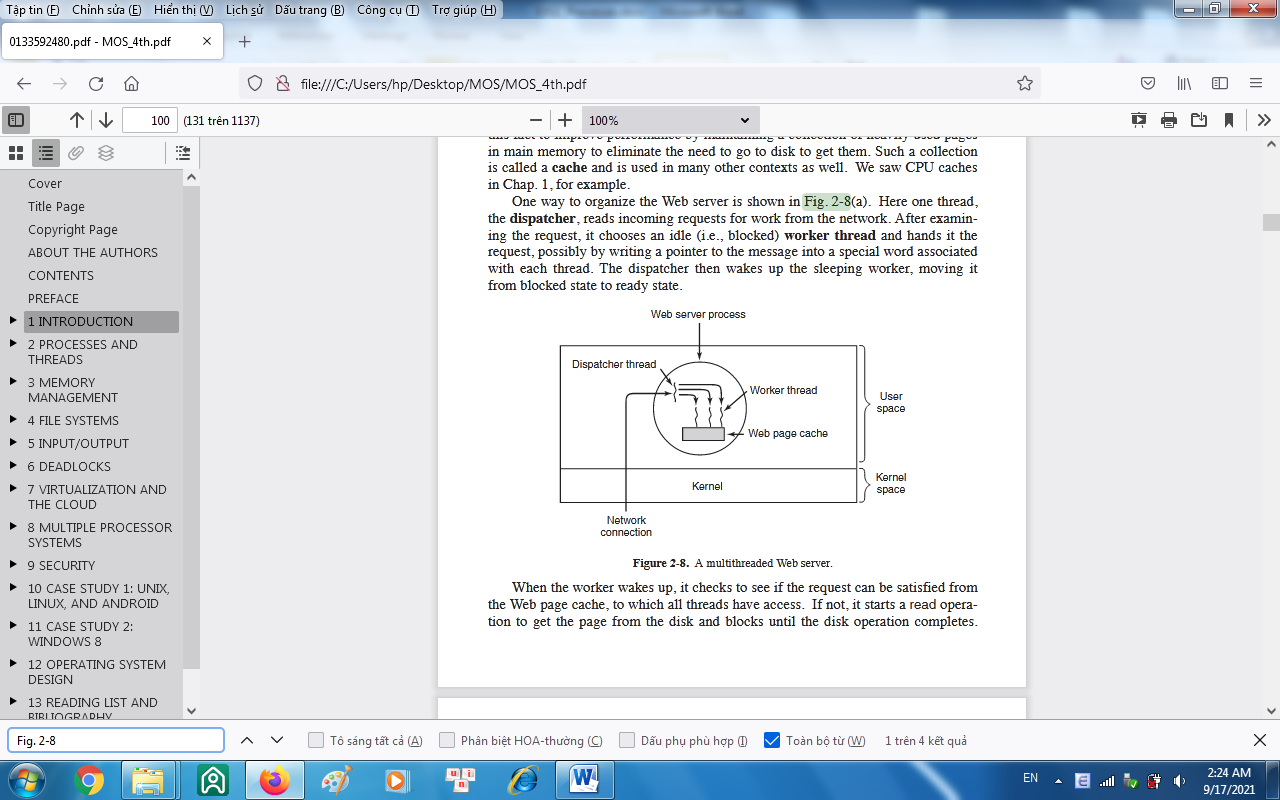
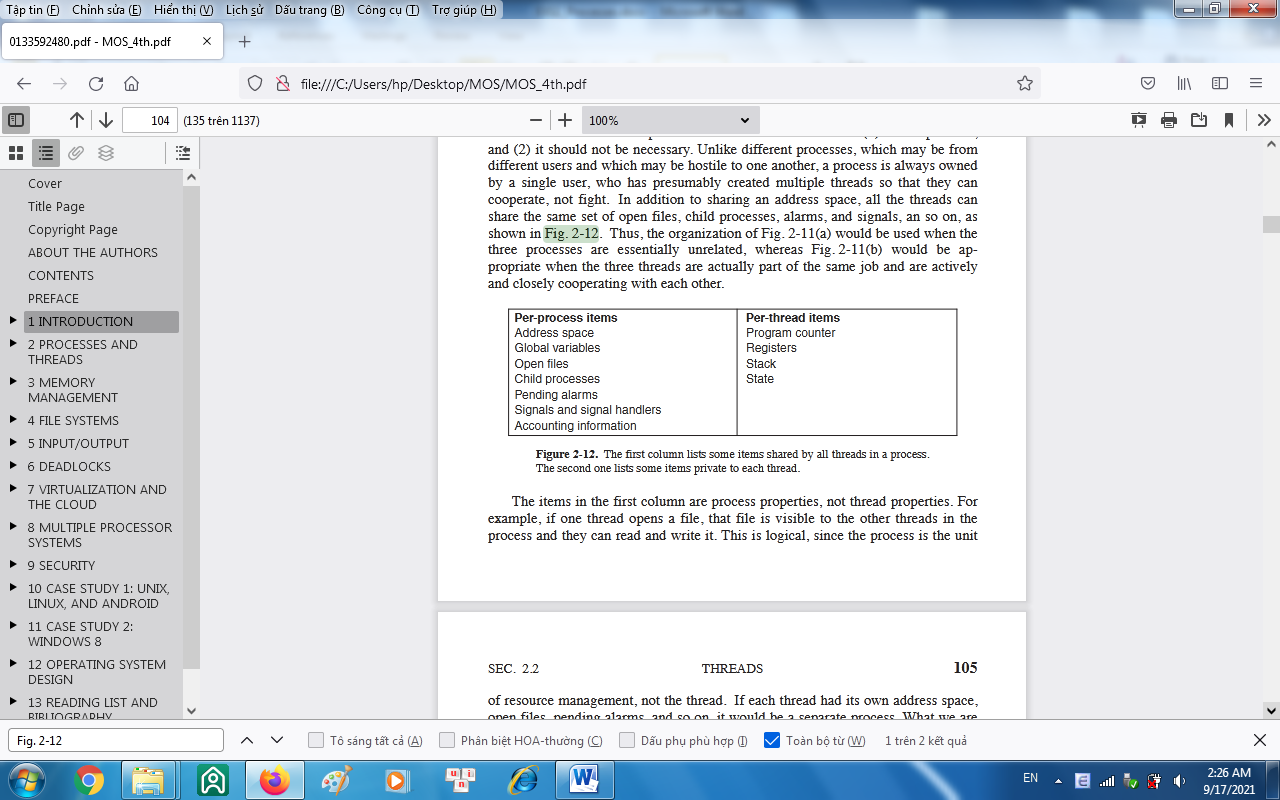
**PROBLEMS**

**CHAPTER 2. PROCESSES AND THREADS**

1. 1. In Fig. 2-2, three process states are shown. In theory, with three states, there could be six transitions, two out of each state. However, only four transitions are shown. Are there any circumstances in which either or both of the missing transitions might occur?
2. PROCESS
3. 
4. 2. Suppose that you were to design an advanced computer architecture that did process switching in hardware, instead of having interrupts. What information would the CPU need? Describe how the hardware process switching might work.
5. 3. On all current computers, at least part of the interrupt handlers are written in assembly language. Why?
6. 4. When an interrupt or a system call transfers control to the operating system, a kernel stack area separate from the stack of the interrupted process is generally used. Why?
7. 5. A computer system has enough room to hold five programs in its main memory. These programs are idle waiting for I/O half the time. What fraction of the CPU time is wasted?
8. 6. A computer has 4 GB of RAM of which the operating system occupies 512 MB. The processes are all 256 MB (for simplicity) and have the same characteristics. If the goal is 99% CPU utilization, what is the maximum I/O wait that can be tolerated?
9. 7. Multiple jobs can run in parallel and finish faster than if they had run sequentially. Suppose that two jobs, each needing 20 minutes of CPU time, start simultaneously. How long will the last one take to complete if they run sequentially? How long if they run in parallel? Assume 50% I/O wait.
10. 8. Consider a multiprogrammed system with degree of 6 (i.e., six programs in memory at the same time). Assume that each process spends 40% of its time waiting for I/O. What will be the CPU utilization?
11. 9. Assume that you are trying to download a large 2-GB file from the Internet. The file is available from a set of mirror servers, each of which can deliver a subset of the file’s bytes; assume that a given request specifies the starting and ending bytes of the file. Explain how you might use threads to improve the download time.
12. 10. In the text it was stated that the model of Fig. 2-11(a) was not suited to a file server using a cache in memory. Why not? Could each process have its own cache?
    * 1. 
13. 11. If a multithreaded process forks, a problem occurs if the child gets copies of all the parent’s threads. Suppose that one of the original threads was waiting for keyboard input. Now two threads are waiting for keyboard input, one in each process. Does this problem ever occur in single-threaded processes?
14. 12. In Fig. 2-8, a multithreaded Web server is shown. If the only way to read from a file is the normal blocking read system call, do you think user-level threads or kernel-level threads are being used for the Web server? Why?
    * + - 1. 
15. 13. In the text, we described a multithreaded Web server, showing why it is better than a single-threaded server and a finite-state machine server. Are there any circumstances in which a single-threaded server might be better? Give an example.
16. 14. In Fig. 2-12 the register set is listed as a per-thread rather than a per-process item. Why? After all, the machine has only one set of registers.
    * + - 1. 
17. 15. Why would a thread ever voluntarily give up the CPU by calling *thread yield*? After all, since there is no periodic clock interrupt, it may never get the CPU back.
18. 16. Can a thread ever be preempted by a clock interrupt? If so, under what circumstances? If not, why not?
19. 17. In this problem you are to compare reading a file using a single-threaded file server and a multithreaded server. It takes 12 msec to get a request for work, dispatch it, and do the rest of the necessary processing, assuming that the data needed are in the block cache. If a disk operation is needed, as is the case one-third of the time, an additional 75 msec is required, during which time the thread sleeps. How many requests/sec can the server handle if it is single threaded? If it is multithreaded?
20. 18. What is the biggest advantage of implementing threads in user space? What is the biggest disadvantage?
21. 19. In Fig. 2-15 the thread creations and messages printed by the threads are interleaved at random. Is there a way to force the order to be strictly thread 1 created, thread 1 prints message, thread 1 exits, thread 2 created, thread 2 prints message, thread 2 exists, and so on? If so, how? If not, why not?
22. 20. In the discussion on global variables in threads, we used a procedure *create global* to allocate storage for a pointer to the variable, rather than the variable itself. Is this es-sential, or could the procedures work with the values themselves just as well?
23. 21. Consider a system in which threads are implemented entirely in user space, with the run-time system getting a clock interrupt once a second. Suppose that a clock interrupt occurs while some thread is executing in the run-time system. What problem might oc-cur? Can you suggest a way to solve it?
24. 22. Suppose that an operating system does not have anything like the select system call to see in advance if it is safe to read from a file, pipe, or device, but it does allow alarm clocks to be set that interrupt blocked system calls. Is it possible to implement a threads package in user space under these conditions? Discuss.
25. 23. Does the busy waiting solution using the *turn* variable (Fig. 2-23) work when the two processes are running on a shared-memory multiprocessor, that is, two CPUs sharing a common memory?
26. 24. Does Peterson’s solution to the mutual-exclusion problem shown in Fig. 2-24 work when process scheduling is preemptive? How about when it is nonpreemptive?
27. 25. Can the priority inversion problem discussed in Sec. 2.3.4 happen with user-level threads? Why or why not?
28. 26. In Sec. 2.3.4, a situation with a high-priority process, *H*, and a low-priority process, *L*, was described, which led to *H* looping forever. Does the same problem occur if round-robin scheduling is used instead of priority scheduling? Discuss.
29. 27. In a system with threads, is there one stack per thread or one stack per process when user-level threads are used? What about when kernel-level threads are used? Explain.
30. 28. When a computer is being developed, it is usually first simulated by a program that runs one instruction at a time. Even multiprocessors are simulated strictly sequentially like this. Is it possible for a race condition to occur when there are no simultaneous events like this?
31. 29. The producer-consumer problem can be extended to a system with multiple producers and consumers that write (or read) to (from) one shared buffer. Assume that each pro-ducer and consumer runs in its own thread. Will the solution presented in Fig. 2-28, using semaphores, work for this system?
32. 30. Consider the following solution to the mutual-exclusion problem involving two proc-esses *P0* and *P1*. Assume that the variable turn is initialized to 0. Process *P0*’s code is presented below.

/\* Other code \*/

while (turn != 0) { } /\* Do nothing and wait. \*/

Cr itical Section /\* . . . \*/

tur n = 0;

/\* Other code \*/

For process *P1*, replace 0 by 1 in above code. Determine if the solution meets *all* the required conditions for a correct mutual-exclusion solution.

1. 31. How could an operating system that can disable interrupts implement semaphores?
2. 32. Show how counting semaphores (i.e., semaphores that can hold an arbitrary value) can be implemented using only binary semaphores and ordinary machine instructions.
3. 33. If a system has only two processes, does it make sense to use a barrier to synchronize them? Why or why not?
4. 34. Can two threads in the same process synchronize using a kernel semaphore if the threads are implemented by the kernel? What if they are implemented in user space? Assume that no threads in any other processes have access to the semaphore. Discuss your answers.
5. 35. Synchronization within monitors uses condition variables and two special operations, wait and signal. A more general form of synchronization would be to have a single primitive, waituntil, that had an arbitrary Boolean predicate as parameter. Thus, one could say, for example,

wait until *x* < 0 or *y* + *z* < *n*

The signal primitive would no longer be needed. This scheme is clearly more general than that of Hoare or Brinch Hansen, but it is not used. Why not? (*Hint*: Think about the implementation.)

1. 36. A fast-food restaurant has four kinds of employees: (1) order takers, who take custom-ers’ orders; (2) cooks, who prepare the food; (3) packaging specialists, who stuff the food into bags; and (4) cashiers, who give the bags to customers and take their money. Each employee can be regarded as a communicating sequential process. What form of interprocess communication do they use? Relate this model to processes in UNIX.
2. 37. Suppose that we have a message-passing system using mailboxes. When sending to a full mailbox or trying to receive from an empty one, a process does not block. Instead, it gets an error code back. The process responds to the error code by just trying again, over and over, until it succeeds. Does this scheme lead to race conditions?
3. 38. The CDC 6600 computers could handle up to 10 I/O processes simultaneously using an interesting form of round-robin scheduling called processor sharing. A process switch occurred after each instruction, so instruction 1 came from process 1, instruc-tion 2 came from process 2, etc. The process switching was done by special hardware, and the overhead was zero. If a process needed *T* sec to complete in the absence of competition, how much time would it need if processor sharing was used with *n* proc-esses?
4. 39. Consider the following piece of C code:

void main( ) {

fork( );

fork( );

exit( );

}

How many child processes are created upon execution of this program?

1. 40. Round-robin schedulers normally maintain a list of all runnable processes, with each process occurring exactly once in the list. What would happen if a process occurred twice in the list? Can you think of any reason for allowing this?
2. 41. Can a measure of whether a process is likely to be CPU bound or I/O bound be deter-mined by analyzing source code? How can this be determined at run time?
3. 42. Explain how time quantum value and context switching time affect each other, in a round-robin scheduling algorithm.
4. 43. Measurements of a certain system have shown that the average process runs for a time *T* before blocking on I/O. A process switch requires a time *S*, which is effectivelywasted (overhead). For round-robin scheduling with quantum *Q*, give a formula for the CPU efficiency for each of the following:
   1. *Q* =∞
   2. *Q* > *T*
   3. *S*<*Q*<*T*
   4. *Q* = *S*
   5. *Q* nearly 0
5. 44. Five jobs are waiting to be run. Their expected run times are 9, 6, 3, 5, and *X*. In what order should they be run to minimize average response time? (Your answer will depend on *X*.)
6. 45. Five batch jobs. *A* through *E*, arrive at a computer center at almost the same time. They have estimated running times of 10, 6, 2, 4, and 8 minutes. Their (externally de-termined) priorities are 3, 5, 2, 1, and 4, respectively, with 5 being the highest priority. For each of the following scheduling algorithms, determine the mean process turnaround time. Ignore process switching overhead.
7. (a) Round robin.
8. (b) Priority scheduling.
9. (c) First-come, first-served (run in order 10, 6, 2, 4, 8).
10. (d) Shortest job first.

For (a), assume that the system is multiprogrammed, and that each job gets its fair share of the CPU. For (b) through (d), assume that only one job at a time runs, until it finishes. All jobs are completely CPU bound.

1. 46. A process running on CTSS needs 30 quanta to complete. How many times must it be swapped in, including the very first time (before it has run at all)?
2. 47. Consider a real-time system with two voice calls of periodicity 5 msec each with CPU time per call of 1 msec, and one video stream of periodicity 33 ms with CPU time per call of 11 msec. Is this system schedulable?
3. 48. For the above problem, can another video stream be added and have the system still be schedulable?
4. 49. The aging algorithm with *a* = 1/2 is being used to predict run times. The previous four runs, from oldest to most recent, are 40, 20, 40, and 15 msec. What is the prediction of the next time?
5. 50. A soft real-time system has four periodic events with periods of 50, 100, 200, and 250 msec each. Suppose that the four events require 35, 20, 10, and *x* msec of CPU time, respectively. What is the largest value of *x* for which the system is schedulable?
6. 51. In the dining philosophers problem, let the following protocol be used: An even-num-bered philosopher always picks up his left fork before picking up his right fork; an odd-numbered philosopher always picks up his right fork before picking up his left fork. Will this protocol guarantee deadlock-free operation?
7. 52. A real-time system needs to handle two voice calls that each run every 6 msec and con-sume 1 msec of CPU time per burst, plus one video at 25 frames/sec, with each frame requiring 20 msec of CPU time. Is this system schedulable?
8. 53. Consider a system in which it is desired to separate policy and mechanism for the scheduling of kernel threads. Propose a means of achieving this goal.
9. 54. In the solution to the dining philosophers problem (Fig. 2-47), why is the state variable set to *HUNGRY* in the procedure *take forks*?
10. 55. Consider the procedure *put forks* in Fig. 2-47. Suppose that the variable *state*[*i*] was set to *THINKING after* the two calls to *test*, rather than *before*. How would this change affect the solution?
11. 56. The readers and writers problem can be formulated in several ways with regard to which category of processes can be started when. Carefully describe three different variations of the problem, each one favoring (or not favoring) some category of proc-esses. For each variation, specify what happens when a reader or a writer becomes ready to access the database, and what happens when a process is finished.
12. 57. Write a shell script that produces a file of sequential numbers by reading the last num-ber in the file, adding 1 to it, and then appending it to the file. Run one instance of the script in the background and one in the foreground, each accessing the same file. How long does it take before a race condition manifests itself? What is the critical region? Modify the script to prevent the race. (*Hint*: use

ln file file.lock

to lock the data file.)

1. 58. Assume that you have an operating system that provides semaphores. Implement a message system. Write the procedures for sending and receiving messages.
2. 59. Solve the dining philosophers problem using monitors instead of semaphores.
3. 60. Suppose that a university wants to show off how politically correct it is by applying the U.S. Supreme Court’s ‘‘Separate but equal is inherently unequal’’ doctrine to gender as well as race, ending its long-standing practice of gender-segregated bathrooms on cam-pus. However, as a concession to tradition, it decrees that when a woman is in a bath-room, other women may enter, but no men, and vice versa. A sign with a sliding marker on the door of each bathroom indicates which of three possible states it is cur-rently in:
   1. Empty
   2. Women present
   3. Men present

In some programming language you like, write the following procedures: *woman wants to enter*, *man wants to enter*, *woman leaves*, *man leaves*. Youmay use whatever counters and synchronization techniques you like.

1. 61. Rewrite the program of Fig. 2-23 to handle more than two processes.
2. 62. Write a producer-consumer problem that uses threads and shares a common buffer. However, do not use semaphores or any other synchronization primitives to guard the shared data structures. Just let each thread access them when it wants to. Use sleep and wakeup to handle the full and empty conditions. See how long it takes for a fatal race condition to occur. For example, you might have the producer print a number once in a while. Do not print more than one number every minute because the I/O could affect the race conditions.
3. 63. A process can be put into a round-robin queue more than once to give it a higher prior-ity. Running multiple instances of a program each working on a different part of a data pool can have the same effect. First write a program that tests a list of numbers for pri-mality. Then devise a method to allow multiple instances of the program to run at once in such a way that no two instances of the program will work on the same number. Can you in fact get through the list faster by running multiple copies of the program? Note that your results will depend upon what else your computer is doing; on a personal computer running only instances of this program you would not expect an im-provement, but on a system with other processes, you should be able to grab a bigger share of the CPU this way.
4. 64. The objective of this exercise is to implement a multithreaded solution to find if a given number is a perfect number. *N* is a perfect number if the sum of all its factors, excluding itself, is *N* ; examples are 6 and 28. The input is an integer, *N* . The output is true if the number is a perfect number and false otherwise. The main program will read the numbers *N* and *P* from the command line. The main process will spawn a set of *P* threads. The numbers from 1 to *N* will be partitioned among these threads so that two threads do not work on the name number. For each number in this set, the thread will determine if the number is a factor of *N* . If it is, it adds the number to a shared buffer that stores factors of *N* . The parent process waits till all the threads complete. Use the appropriate synchronization primitive here. The parent will then determine if the input number is perfect, that is, if *N* is a sum of all its factors and then report accordingly. (**Note**: You can make the computation faster by restricting the numbers searched from 1 to the square root of *N* .)
5. 65. Implement a program to count the frequency of words in a text file. The text file is partitioned into *N* segments. Each segment is processed by a separate thread that out-puts the intermediate frequency count for its segment. The main process waits until all the threads complete; then it computes the consolidated word-frequency data based on the individual threads’ output.