SOLUTIONS TO CHAPTER 2 PROBLEMS

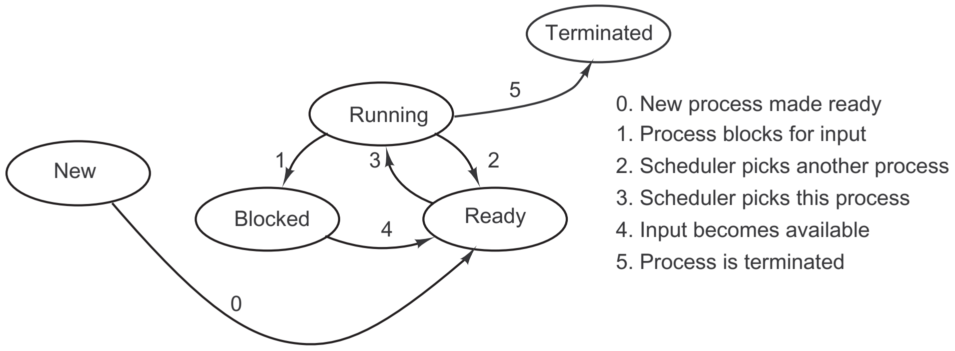
1. It is central because there is so much parallel or pseudoparallel activity—multiple user processes and I/O devices running at once. The multiprogramming model allows this activity to be described and modeled better.

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

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

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

5. The figure looks like this



Terminated
Running
New
Blocked
Ready
0
1
2
3
4
5
0. New process made ready
1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available
5. Process is terminated

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

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

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

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

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

11. A possible shell script might be:

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

```
   n=‘tail –1 numbers‘
   expr $n + 1 >>numbers
  done
```

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

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

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

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

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

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

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

16. Associated with each counting semaphore are two binary semaphores, *M*, used for mutual exclusion, and *B*, used for blocking. Also associated with each counting semaphore is a counter that holds the number of UPs minus the number of DOWNs, and a list of processes blocked on that semaphore. To implement DOWN, a process first gains exclusive access to the semaphores, counter, and list by doing a DOWN on *M*. It then decrements the counter. If it is zero or more, it just does an UP on *M* and exits. If *M* is negative, the process is put on the list of blocked processes. Then an UP is done on *M* and a DOWN is done on *B* to block the process. To implement UP, first *M* is DOWNed to get mutual exclusion, and then the counter is incremented. If it is more than zero, no one was blocked, so all that needs to be done is to UP *M*. If, however, the counter is now negative or zero, some process must be removed from the list. Finally, an UP is done on *B* and *M* in that order.

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

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

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

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

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

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

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

24. It will need nT sec.

25. If a process occurs multiple times in the list, it will get multiple quanta per cycle. This approach could be used to give more important processes a larger share of the CPU.

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

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

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

27. Shortest job first is the way to minimize average response time.
$0 < X \leq 3$: X, 3, 5, 6, 9.
$3 < X \leq 5$: 3, X, 5, 6, 9.
$5 < X \leq 6$: 3, 5, X, 6, 9.
$6 < X \leq 9$: 3, 5, 6, X, 9.
$X > 9$: 3, 5, 6, 9, X.

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

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

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

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

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

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

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

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

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

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

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

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