SOLUTIONS TO CHAPTER 6 PROBLEMS

1. In the U.S., consider a presidential election in which three or more candidates are trying for the nomination of some party. After all the primary elections are finished, when the delegates arrive at the party convention, it could happen that no candidate has a majority and that no delegate is willing to change his or her vote. This is a deadlock. Each candidate has some resources (votes) but needs more to get the job done. In countries with multiple political parties in the parliament, it could happen that each party supports a different version of the annual budget and that it is impossible to assemble a majority to pass the budget. This is also a deadlock.

2. Disk space on the spooling partition is a finite resource. Every block that comes in de facto claims a resource and every new one arriving wants more resources. If the spooling space is, say, 10 MB and the first half of ten 2-MB jobs arrive, the disk will be full and no more blocks can be stored so we have a deadlock. The deadlock can be avoided by allowing a job to start printing before it is fully spooled and reserving the space thus released for the rest of that job. In this way, one job will actually print to completion, then the next one can do the same thing. If jobs cannot start printing until they are fully spooled, deadlock is possible.

3. The printer is nonpreemptable; the system cannot start printing another job until the previous one is complete. The spool disk is preemptable; you can delete an incomplete file that is growing too large and have the user send it later, assuming the protocol allows that.

4. Yes. It does not make any difference whatsoever.

5. Suppose that there are three processes, *A*, *B* and *C*, and two resource types, *R* and *S*. Further assume that there are one instance of *R* and two instance of *S*. Consider the following execution scenario: *A* requests *R* and gets it; *B* requests *S* and gets; *C* requests *S* and gets it (there are two instances of *S*); *B* requests *R* and is blocked; *A* requests *S* and is blocked. At this stage all four conditions hold. However, there is no deadlock. When *C* finishes, one instance of *S* is released that is allocated to *A*. Now *A* can complete its execution and release *R* that can be allocated to *B*, which can then complete its execution. These four conditions are enough if there is one resource of each type.

6. ‘‘Don’t block the box’’ is a pre-allocation strategy, negating the hold and wait deadlock precondition, since we assume that cars can enter the street space following the intersection, thus freeing the intersection. Another strategy might allow cars to temporarily pull into garages and release enough space to clear the gridlock. Some cities have a traffic control policy to shape traffic; as city streets become more congested, traffic supervisors adjust the settings for red lights in order to throttle traffic entering heavily congested areas. Lighter traffic ensures less competition over resources and thus lowers the probability of gridlock occurring.

7. The above anomaly is not a communication deadlock since these cars are independent of each other and would drive through the intersection with a minimal delay if no competition occurred. It is not a resource deadlock, since no car is holding a resource that is requested by another car. Nor would the mechanisms of resource pre-allocation or of resource preemption assist in controlling this anomaly. This anomaly is one of competition synchronization, however, in which cars are waiting for resources in a circular chain and traffic throttling may be an effective strategy for control. To distinguish from resource deadlock, this anomaly might be termed a "scheduling deadlock." A similar deadlock could occur following a law that required two trains merging onto a shared railroad track to wait for the other to proceed. Note that a policeman signaling one of the competing cars or trains to proceed (and not the others) can break this dead state without rollback or any other overhead.

8. It is possible that one process holds some or all of the units of one resource type and requests another resource type, while another process holds the second resource while requesting the available units of the first resource type. If no other process can release units of the first resource type and the resource cannot be preempted or used concurrently, the system is deadlocked. For example, two processes are both allocated memory cells in a real memory system. (We assume that swapping of pages or processes is not supported, while dynamic requests for memory are supported.) The first process locks another resource - perhaps a data cell. The second process requests the locked data and is blocked. The first process needs more memory in order to execute the code to release the data. Assuming that no other processes in the system can complete and release memory cells, a deadlock exists in the system.

9. Yes, illegal graphs exist. We stated that a resource may only be held by a single process. An arc from a resource square to a process circle indicates that the process owns the resource. Thus, a square with arcs going from it to two or more processes means that all those processes hold the resource, which violates the rules. Consequently, any graph in which multiple arcs leave a square and end in different circles violates the rules unless there are multiple copies of the resources. Arcs from squares to squares or from circles to circles also violate the rules.

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

11. Consider three processes, *A*, *B* and *C* and two resources *R* and *S*. Suppose *A* is waiting for *I* that is held by *B*, *B* is waiting for *S* held by *A*, and *C* is waiting for *R* held by *A*. All three processes, *A*, *B* and *C* are deadlocked. However, only *A* and *B* belong to the circular chain.

12. This is clearly a communication deadlock, and can be controlled by having *A* time out and retransmit its enabling message (the one that increases the window size) after some period of time (a heuristic). It is possible, however, that *B* has received both the original and the duplicate message. No harm will occur if the update on the window size is given as an absolute value and not as a differential. Sequence numbers on such messages are also effective to detect duplicates.

13. A portion of all such resources could be reserved for use only by processes owned by the administrator, so he or she could always run a shell and programs needed to evaluate a deadlock and make decisions about which processes to kill to make the system usable again.

14. First, the set of unmarked processes, P = (*P1 P2 P3 P4*)
R1 is not less than or equal to *A*
R2 is less than *A*; Mark *P2*; *A* = (0 2 0 3 1); *P* = (*P1 P3 P4*)
R1 is not less than or equal to *A*
R3 is equal to *A*; Mark *P3*; *A* = (0 2 0 3 2); *P* = (*P1 P4*)
R1 is not less than or equal to *A*
R4 is not less than or equal to *A*

So, processes *P1* and *P4* remain unmarked. They are deadlocked.

15. Recovery through preemption: After processes *P2* and *P3* complete, process *P1* can be forced to preempt 1 unit of *RS3*. This will make A = (0 2 1 3 2), and allow process *P4* to complete. Once *P4* completes and release its resources *P1* may complete. Recovery through rollback: Rollback *P1* to the state checkpointed before it acquired *RS3*. Recovery through killing processes: Kill *P1*.

16. The process is asking for more resources than the system has. There is no conceivable way it can get these resources, so it can never finish, even if no other processes want any resources at all.

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

18. Yes. Do the whole thing in three dimensions. The *z*-axis measures the number of instructions executed by the third process.

19. The method can only be used to guide the scheduling if the exact instant at which a resource is going to be claimed is known in advance. In practice, this is rarely the case.

20. There are states that are neither safe nor deadlocked, but which lead to deadlocked states. As an example, suppose we have four resources: tapes, plotters, scanners, and CD-ROMs, as in the text, and three processes competing for them. We could have the following situation:

	Has	Needs	Available
A:	2 0 0 0	1 0 2 0	0 1 2 1
B:	1 0 0 0	0 1 3 1	
C:	0 1 2 1	1 0 1 0	

This state is not deadlocked because many actions can still occur, for example, *A* can still get two printers. However, if each process asks for its remaining requirements, we have a deadlock.

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

22. The system is deadlock free. Suppose that each process has one resource. There is one resource free. Either process can ask for it and get it, in which case it can finish and release both resources. Consequently, deadlock is impossible.

23. If a process has m resources it can finish and cannot be involved in a deadlock. Therefore, the worst case is where every process has $m - 1$ resources and needs another one. If there is one resource left over, one process can finish and release all its resources, letting the rest finish too. Therefore the condition for avoiding deadlock is $r \geq p(m - 1) + 1$.

24. No. *D* can still finish. When it finishes, it returns enough resources to allow *E* (or *A*) to finish, and so on.

25. Comparing a row in the matrix to the vector of available resources takes m operations. This step must be repeated on the order of n times to find a process that can finish and be marked as done. Thus, marking a process as done takes on the order of mn steps. Repeating the algorithm for all n processes means that the number of steps is then mn^2. Thus, $a = 1$ and $b = 2$.

26. The needs matrix is as follows:

0 1 0 0 2
0 2 1 0 0
1 0 3 0 0
0 0 1 1 1

If x is 0, we have a deadlock immediately. If x is 1, process *D* can run to completion. When it is finished, the available vector is 1 1 2 2 1. Unfortunately we are now deadlocked. If x is 2, after *D* runs, the available vector is 1 1 3 2 1 and *C* can run. After it finishes and returns its resources the available vector is 2 2 3 3 1, which will allow *B* to run and complete, and then *A* to run and complete. Therefore, the smallest value of x that avoids a deadlock is 2.

27. Consider a process that needs to copy a huge file from a tape to a printer. Because the amount of memory is limited and the entire file cannot fit in this memory, the process will have to loop through the following statements until the entire file has been printed:

Acquire tape drive
Copy the next portion of the file in memory (limited memory size)
Release tape drive
Acquire printer
Print file from memory
Release printer

This will lengthen the execution time of the process. Furthermore, since the printer is released after every print step, there is no guarantee that all portions of the file will get printed on continuous pages.

28. Suppose that process *A* requests the records in the order *a*, *b*, *c*. If process *B* also asks for *a* first, one of them will get it and the other will block. This situation is always deadlock free since the winner can now run to completion without interference. Of the four other combinations, some may lead to deadlock and some are deadlock free. The six cases are as follows:

a b c	deadlock free
a c b	deadlock free
b a c	possible deadlock
b c a	possible deadlock
c a b	possible deadlock
c b a	possible deadlock

Since four of the six may lead to deadlock, there is a 1/3 chance of avoiding a deadlock and a 2/3 chance of getting one.

29. Yes. Suppose that all the mailboxes are empty. Now *A* sends to *B* and waits for a reply, *B* sends to *C* and waits for a reply, and *C* sends to *A* and waits for a reply. All the conditions for a communications deadlock are now fulfilled.

30. To avoid circular wait, number the resources (the accounts) with their account numbers. After reading an input line, a process locks the lower-numbered account first, then when it gets the lock (which may entail waiting), it locks the other one. Since no process ever waits for an account lower than what it already has, there is never a circular wait, hence never a deadlock.

31. Change the semantics of requesting a new resource as follows. If a process asks for a new resource and it is available, it gets the resource and keeps what it already has. If the new resource is not available, all existing resources are released. With this scenario, deadlock is impossible and there is no danger that the new resource is acquired but existing ones lost. Of course, the process only works if releasing a resource is possible (you can release a scanner between pages or a CD recorder between CDs).

32. I'd give it an F (failing) grade. What does the process do? Since it clearly needs the resource, it just asks again and blocks again. This is no better than staying blocked. In fact, it may be worse since the system may keep track of how long competing processes have been waiting and assign a newly freed resource to the process that has been waiting longest. By periodically timing out and trying again, a process loses its seniority.

33. Both virtual memory and time-sharing systems were developed mainly to assist system users. Virtualizing hardware shields users from the details of prestating needs, resource allocation, and overlays, in addition to preventing deadlock. The cost of context switching and interrupt handling, however, is considerable. Specialized registers, caches, and circuitry are required. Probably this cost would not have been incurred for the purpose of deadlock prevention alone.

34. A deadlock occurs when a set of processes are blocked waiting for an event that only some other process in the set can cause. On the other hand, processes in a livelock are not blocked. Instead, they continue to execute checking for a condition to become true that will never become true. Thus, in addition to the resources they are holding, processes in livelock continue to consume precious CPU time. Finally, starvation of a process occurs because of the presence of other processes as well as a stream of new incoming processes that end up with higher priority that the process being starved. Unlike deadlock or livelock, starvation can terminate on its own, e.g. when existing processes with higher priority terminate and no new processes with higher priority arrive.

35. This dead state is an anomaly of competition synchronization and can be controlled by resource pre-allocation. Processes, however, are not blocked from resources. In addition, resources are already requested in a linear order. This anomaly is not a resource deadlock; it is a livelock. Resource preallocation will prevent this anomaly. As a heuristic, processes may time-out and release their resources if they do not complete within some interval of time, then go to sleep for a random period and then try again

36. Here are the answers, albeit a bit complicated.

(a) This is a competition synchronization anomaly. It is also a livelock. We might term it a scheduling livelock. It is not a resource livelock or deadlock, since stations are not holding resources that are requested by others and thus a circular chain of stations holding resources while requesting others does not exist. It is not a communication deadlock, since stations are executing independently and would complete transmission were scheduled sequentially.

(b) Ethernet and slotted Aloha require that stations that detect a collision of their transmission must wait a random number of time slots before retransmitting. The interval within which the time slot is chosen is doubled after each successive collision, dynamically adjusting to heavy traffic loads. After sixteen successive retransmissions a frame is dropped.

(c) Because access to the channel is probabilistic, and because newly arriving stations can compete and be allocated the channel before stations that have retransmitted some number of times, starvation is enabled.

37. The anomaly is not a resource deadlock. Although processes are sharing a mutex, i.e., a competition mechanism, resource pre-allocation and deadlock avoidance methods are all ineffective for this dead state. Linearly ordering resources is also ineffective. Indeed one could argue that linear orders may be the problem; executing a mutex should be the last step before entering and the first after leaving a critical section. A circular dead state does exist in which both processes wait for an event that can only be caused by the other process. This is a communication deadlock. To make progress, a time-out will work to break this deadlock if it preempts the consumer's mutex. Writing careful code or using monitors for mutual exclusion are better solutions.

38. If both programs ask for Woofer first, the computers will starve with the endless sequence: request Woofer, cancel request, request Woofer, cancel request, and so on. If one of them asks for the doghouse and the other asks for the dog, we have a deadlock, which is detected by both parties and then broken, but it is just repeated on the next cycle. Either way, if both computers have been programmed to go after the dog or the doghouse first, either starvation or deadlock ensues. There is not really much difference between the two here. In most deadlock problems, starvation does not seem serious because introducing random delays will usually make it very unlikely. That approach does not work here.