Problem 1

The load-linked / store-conditional is a pair of instructions that provide a race-condition free way to write to a memory address. The load-linked instruction retrieves a memory address and the store-conditional writes to that memory address, but only if there have been no updates to that location since the load-link.

Without this check on the second instruction, a race condition can occur, such as in the following example:

Instructions:

1) Value in memory: 5
2) Process 1 reads value: 5
3) Process 2 reads value: 5
4) Process 1 adds 2 to value 5: 7
5) Process 2 adds 3 to value 5: 8

6) Process 1 writes value: 7
7) Process 2 writes value: 8
8) Value in memory: 8

OR

6) Process 2 writes value: 8
7) Process 1 writes value: 7
8) Value in memory: 7

The end result is that the value in memory could be either 7 or 8, even though the value should be 10. The problem is that both processes read the value in memory before either modified it.

With the conditional check on the write operation, this issue is solved.

Instructions:

1) Value in memory: 5
2) Process 1 reads value: 5
3) Process 2 reads value: 5
4) Process 1 adds 2 to value 5: 7
5) Process 2 adds 3 to value 5: 8

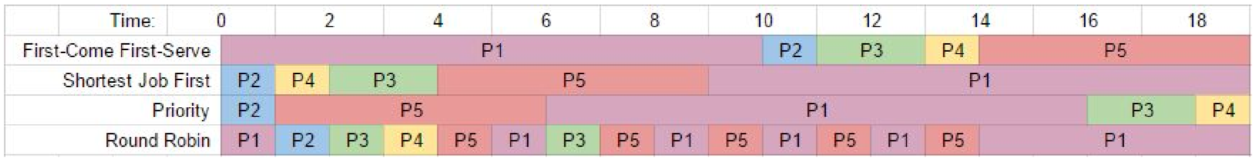
6) Process 1 writes value: 7
7) Process 2 attempts but fails to write value: 8
8) Value in memory: 7
9) Process 2 reads value: 7
10) Process 2 adds 3 to value 7: 10
11) Process 2 writes value: 10
12) Value in memory: 10

OR

6) Process 2 writes value: 8
7) Process 1 attempts but fails to write value: 7
8) Value in memory: 8
9) Process 1 reads value: 8
10) Process 1 adds 2 to value 8: 10
11) Process 1 writes value: 10
12) Value in memory: 10

In this case with the conditional check, the second process to write will fail, so that process knows it must re-read the value in memory to complete its operation. Thus the race condition is avoided.

Problem 2



Turnaround Time				
	First-Come First-Serve	Shortest Job First	Priority	Round Robin
P1	10	19	16	19
P2	11	1	1	2
P3	13	4	18	7
P4	14	2	19	4
P5	19	9	6	14

Waiting Time				
	First-Come First-Serve	Shortest Job First	Priority	Round Robin
P1	0	9	6	9
P2	10	0	0	1
P3	11	2	16	5
P4	13	1	18	3
P5	14	4	1	9

Average Waiting Time				
	First-Come First-Serve	Shortest Job First	Priority	Round Robin
t	9.6	3.2	8.2	5.4

Shortest job first yields the lowest average waiting time.

Problem 3

A scheduling algorithm that prioritizes processes with larger I/O bursts than CPU bursts will run those processes first. As a result, CPU-heavy processes will have to wait a little while to get processor time. However, since the processes with large I/O bursts will spend a lot of time waiting for an I/O response, the CPU-heavy processes can get CPU time while the I/O-heavy processes are waiting. Because of this, the CPU-heavy processes will not be permanently starved.

Problem 4

Assume two threads, T1 and T2, are running on two different CPUs on a multi-CPU system. T1 currently has a mutex M, but the critical section in this mutex is quite short so T1 will be able to give up the mutex shortly.

While T1 has the mutex, T2 reaches an instruction which tries to acquire the mutex. It will fail since T1 still has the mutex. However, if we know that T1 will give up the mutex shortly, it doesn't make sense to block T2 and go through all the overhead of context-switching to another process. Thus, we should just spin T2 a few cycles until T1 gives up the mutex.

On a single processor system, this situation can not occur. Say T1 acquires a mutex, but during the critical section it is preempted and a context switch to T2 occurs. Now T2 reaches the instruction where it needs the mutex that T1 has. Since T1 cannot be running concurrently (as this is a single processor system), T1 will not give up the mutex no matter how long we spin T2. Thus, we must context-switch away from T2 and should not spin.