Short Answer Questions

1.
   a. Spinlocks are efficient if threads are likely to be blocked for only short periods, however; if held for long duration this may prevent other threads from running. In a uniprocessor system, it is unnecessary to use spinlocks because the thread that is currently "spinning" to acquire the lock is hogging the entire CPU. This means that the entire system will be held up. Only work in multi-core CPUs.

   b. Mutex locks are similar to spinlocks, the only difference is that the thread that is waiting for the lock to be released is put to sleep instead of being awake and constantly checking, in a loop, if the lock has been released. Mutex locks work in single and multi-core CPUs.

   c. A semaphore is a protected integer variable that is used to control access to shared resources by multiple processes. When a certain resource has been used up, the "thread" must wait until the semaphore variable is incremented which represents a resource being available. When the thread takes the resource, the variable is in turn decremented symbolizing that the system now has one less of that resource for use by other threads. Only works in multi-core CPUs.

   d. Condition variables allow threads to wait until a condition occurs, condition variables are usually combined with mutual exclusion which is what a monitor is. Works in multi-core CPUs, but not so well single core CPUs.

   e. An RW lock allows concurrent access for read-only operations, while write operations require exclusive access. This means that multiple threads can read the data in parallel but an exclusive lock is needed for writing or modifying data. Readers–writer locks are usually constructed on top of mutexes and condition variables, or on top of semaphores.

2.
   a. Disabling interrupts disallow the possibility of a thread currently in a critical area of code to be interrupted which would ultimately allow another thread to enter the holy sacred protected area (critical section).

   b. Non-system programs do not have the capability to know the effects of disabling interrupts; only the OS kernel does. This is why programs running in user mode must contact that OS kernel for services.

3.

a.

I. In the **else** block, it is possible for there to be **1+x** or more threads in there and **if** **resources_allocated = MAX_RESOURCES-1**, then you will have allocated an extra **x** resources which is a problem.

II. In the **else** block, it is possible for concurrent threads to increment the **resources_allocated** variable at the same time.

III. In the **else** block, it is possible for a thread to be interrupted, thus allowing another thread(s) to come in and increment the **resources_allocated** variable and once the original thread comes back from the interrupt it can potentially increment it past the value of **MAX_RESOURCES**.

IV. In the **release_resource** function, it is possible for concurrent threads to decrement the **resources_allocated** variable at the same time.

b.

```
_Bool allocate_resource(void) {
   acquire()
   if (resouces_allocated >= MAX_RESOURCES) {
         release();
         return false;
   }
   else {
         resources_allocated++;
         release();
         return true;
   }
}

void release_resource(void)
{
   acquire();
   resources_allocated--;
   release();
}
```

c. No, even if they are done atomically, it does not prevent the possibility of multiple threads entering the **else** block.

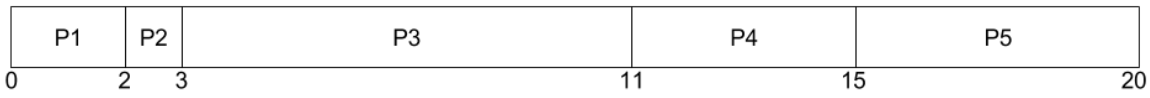
1.

a.

i. FCFS

0	P1
1	P1
2	P2
3	P3
4	P3
5	P3
6	P3
7	P3
8	P3
9	P3
10	P3
11	P4
12	P4
13	P4
14	P4
15	P5
16	P5
17	P5
18	P5
19	P5

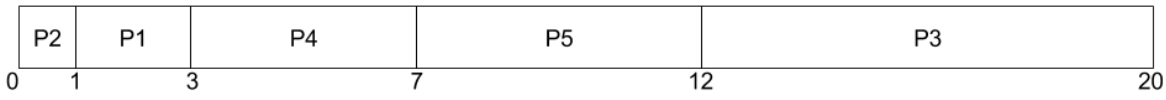
Gantt chart:



ii. SJF

0	P2
1	P1
2	P1
3	P4
4	P4
5	P4
6	P4
7	P5
8	P5
9	P5
10	P5
11	P5
12	P3
13	P3
14	P3
15	P3
16	P3
17	P3
18	P3
19	P3

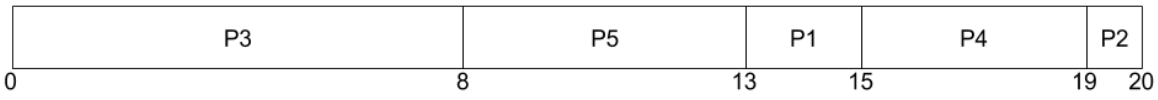
Gantt chart:



iii. Non-preemptive priority

0	P3
1	P3
2	P3
3	P3
4	P3
5	P3
6	P3
7	P3
8	P5
9	P5
10	P5
11	P5
12	P5
13	P1
14	P1
15	P4
16	P4
17	P4
18	P4
19	P2

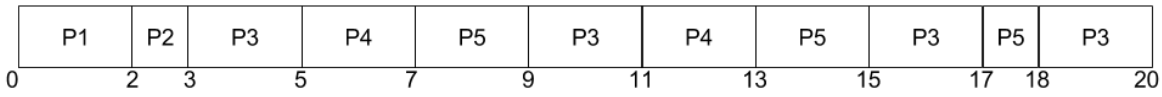
Gantt chart:



iv. RR (quantum = 2)

0	P1
1	P1
2	P2
3	P3
4	P3
5	P4
6	P4
7	P5
8	P5
9	P3
10	P3
11	P4
12	P4
13	P5
14	P5
15	P3
16	P3
17	P5
18	P3
19	P3

Gantt chart:



b. Turnaround times

	FCFS	SJF	Non-preemptive priority	RR
P1	2	3	15	2
P2	3	1	20	3
P3	11	20	8	20
P4	15	7	19	13
P5	20	12	13	18

c. Waiting times

	FCFS	SJF	Non-preemptive priority	RR
P1	0	1	13	0
P2	2	0	19	2
P3	3	12	0	12
P4	11	3	15	9
P5	15	7	8	13

d. SJF have the smallest average waiting time.

2.

a. Each task is given a quantum of 1 millisecond and a context-switch requires 0.1 milliseconds. This results in a CPU utilization of (1/1.1)*100% = 91%.
b. Each task is given a quantum of 10 milliseconds and for each CPU burst of 1 millisecond a task issues an I/O operation. The time required to cycle through all the processes is therefore 10*1.1 + 10.1. This results in a CPU utilization of (20/21.1)*100% = 94%.

3.

a. FCFS does not favour short processes if a longer process comes before it which means you will have a short job waiting on a potentially really long job. In fact, it is better if the short job completes first and then the longer job follows after because the average waiting time is greatly reduced in this case.
b. It favours short jobs to a certain extent, it allows the short jobs to be completed even if there are long jobs that get processed first in the RR algorithm.
c. Probably not, it depends on how you designed the multilevel queue. In the worst case, the short job can go through many levels before it is even finished which is not desired.