1.

a) Preemptive Priority Scheduling

P_1	P_2	P_3	P_2	P_3	P_4	P_1

0 2 6 10 13 16 21 29

Average Waiting time = ((21 – 2) + (10 – 6) + ((6 – 4) + (13 – 10)) + (16 – 6))/4 = 19+4+5+10/4= 9.5
Average Turnaround time = ((29 – 0) + (13 – 2) + (16 – 4) + (21 – 6))/4 =29+11+12+15/4= 16.75

b) Shortest remain time first

P_1	P_2	P_2	P_2	P_4	P_3	P_1

0 2 4 6 9 14 21 29

Average Waiting time = ((21 – 2) + 0 + (14 – 4) + (9 – 6)/4 =8
Average Turnaround time = ((29 – 0)+ (9 – 2) + (21 – 4) + (14 – 6))/4 =15.25

2.

- CPU utilization
- Throughput
- Turnaround time
- Average waiting time
- Response time

3.

lets assume Permit =0 at time T_0
P_0 tries to enter C.S. and can enter since Permit =0.
P_0 finish its job in C.S. and set Permit =1
P_1 is currently running outside C.S ,it is terminated with fatal error.
P_0 tries to enter C.S. again but P_0 never can.

4.

Sol) Lets assume a short-term scheduler use the priority to select a process from the ready queue. At time t_0 , there is only one process P_L with low priority in the ready queue. The short term scheduler select P_L and let it use CPU. Then P_L enter a critical region (section). At time t_1, a process P_H with higher priority becomes ready state. The short-term scheduler stop P_L to use CPU. Now P_H and P_L are in ready queue. The short-term scheduler select higher priority process P_H and let it use CPU. P_H try to get into the critical section. P_H must wait outside critical section since P_L is already in the critical section. Since P_L has lower priority, P_L never get change to use CPU. P_H never be able to enter critical session.

5.

1) No two processes may be simultaneously inside their critical regions – mutual exclusion
2) No process running outside its critical region may block other processes
3) No process should have to wait forever to enter critical region
4) No assumptions may be made about speeds or the number of CPUs.

6.

Let's assume at time T_0: empty = N, full = 0, mutex =1

- consumer is scheduled : down mutex (now mutex =0), try to down full. Since full =0, consumer cannot finish down operation and sleep on semaphore full.
- producer is scheduled: produce item and call down (&empty). Since empty =N, Since empty=N, producer can finish down(&empty), then call down(&mutex). Since mutex is already down by producer, consumer cannot finish down operation and producer sleep on semaphore mutex.
- Now producer and consumer sleep forever!

7.

- Data Parallelism - In data parallelism, the same task or operation is performed on multiple pieces of data simultaneously.

- Task Parallelism - In task parallelism, different threads or processes perform distinct, independent tasks concurrently.

8.

- Since kernel only involved in creation of a shared memory, to access shared memory does not need context switch between kernel and process.

9.

- **Many-to-One** - Multiple user-level threads are mapped to a single kernel-level thread. All thread management is handled by the user-level thread library. The operating system sees only one thread, which means that if one thread blocks for any reason (e.g., I/O operation), it blocks the entire process, including all other user-level threads.

- **One-to-One** - Each user-level thread corresponds to exactly one kernel-level thread. Thread management is handled by both the user-level thread library and the operating system kernel. It provides true parallelism but a large number of kernel threads may burden the performance of a system.

- **Many-to-Many** - It allows multiple user-level threads to be mapped to a smaller or equal number of kernel-level threads. The user-level thread library is responsible for managing user-level threads, and the kernel manages a pool of kernel-level threads. This model provides flexibility and can adapt to the number of available processor cores.