Homework 3 Solutions:

5.5 Describe the actions taken by a thread library to context switch between user-level threads.

Answer: Context switching between user threads is quite similar to switching between kernel threads, although it is dependent on the threads library and how it maps user threads to kernel threads. In general, context switching between user threads involves taking a user thread of its LWP and replacing it with another thread. This act typically involves saving and restoring the state of the registers.

5.6 What resources are used when a thread is created? How do they differ from those used when a process is created?

Answer: Because a thread is smaller than a process, thread creation typically uses fewer resources than process creation. Creating a process requires allocating a process control block (PCB), a rather large data structure. The PCB includes a memory map, list of open files, and environment variables. Allocating and managing the memory map is typically the most time-consuming activity. Creating either a user or kernel thread involves allocating a small data structure to hold a register set, stack, and priority.

6.3 Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

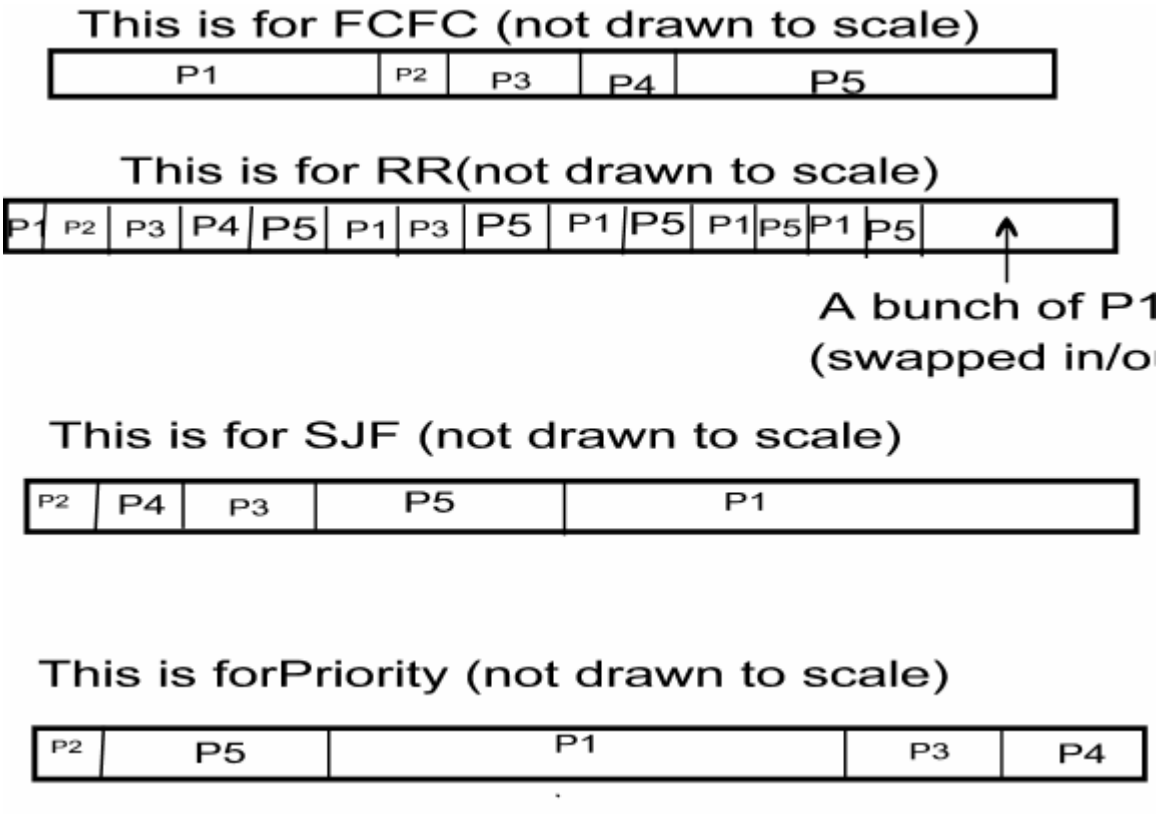
The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5, all at time 0.
a. Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a nonpreemptive priority (a smaller priority number implies a higher priority), and
RR (quantum =1) scheduling.

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

c. What is the waiting time of each process for each of the scheduling algorithms in part a?

d. Which of the schedules in part a results in the minimal average waiting time (over all processes)?

Answer:
a. The four Gantt charts are



b. Turnaround time

Process	FCFS	RR	SJF	Priority
P1	10	19	19	16
P2	11	2	1	1
P3	13	7	4	18
P4	14	4	2	19
P5	19	14	9	6

c. Waiting time (turnaround time minus burst time)

Process	FCFS	RR	SJF	PRIORITY
P1	0	9	9	6
P2	10	1	0	0
P3	11	5	2	16
P4	13	3	1	18
P5	14	9	4	1

d. Shortest Job First

6.6 What advantage is there in having different time-quantum sizes on different levels of a multilevel queueing system?

Answer: Processes that need more frequent servicing, for instance, interactive processes such as editors, can be in a queue with a small time quantum. Processes with no need for frequent servicing can be in a queue with a larger quantum, requiring fewer context switches to complete the processing, making more efficient use of the computer.

7.5 Servers may be designed so that they limit the number of open connections. For example, a server may only wish to have *N* socket connections at any point in time. As soon as
N connections are made; the server will not accept another incoming connection until an existing connection is released. Explain how semaphores can be used by a server to limit the number of concurrent connections.

Answer: A semaphore is initialized to the number of allowable open socket connections.
When a connection is accepted, the **acquire**() method is called, when a connection is released, the release() method is called. If the system reaches the number of allowable socket connections, subsequent calls to **acquire**() will block until an existing connection is terminated and the release method is invoked.

7.7 The **wait**() statement in all Java program examples was part of a while loop. Explain why you would always need to use a while statement when using **wait**() and why you would never use an if statement.

Answer: This is an important issue to emphasize! Java only provides anonymous notification—you cannot notify a certain thread that a certain condition is true. When a thread is notified, it is its responsibility to re-check the condition that it is waiting for. If a thread did not re-check the condition, it may have received the notification without the condition having been met. As an example, consider the **doWork**() method in Figure 7.38. If it wasn't the turn of the thread receiving the notification, without a while statement, the thread would proceed upon returning from the call to **wait**(). The while statement causes the thread to re-check the condition that it was waiting for.