**Question 3 Threading related Problems**

**3.1 What is a semaphore? How is it different from a mutex?**

Ans - A semaphore is a synchronization primitive used in concurrent programming to control access to shared resources. It is an integer variable that is used to manage the access and coordination of multiple threads or processes. The value of a semaphore represents the number of available resources.

Semaphores have two main operations:

1. Wait (P): Decrements the semaphore value. If the value becomes negative, the thread or process is blocked, and it waits until the semaphore value becomes positive again.
2. Signal (V): Increments the semaphore value. If there are blocked threads or processes waiting on the semaphore, one of them is woken up and allowed to proceed.

A mutex (short for mutual exclusion) is also a synchronization primitive used to protect shared resources. It is typically implemented as a binary semaphore, meaning it can have only two states: locked or unlocked. Mutexes provide exclusive access to a shared resource, allowing only one thread or process to enter the critical section at a time.

The main difference between a semaphore and a mutex is their usage and behavior:

1. Counting: A semaphore can have a value greater than 1, representing the number of available resources. Multiple threads can acquire and release the semaphore independently. In contrast, a mutex is binary and allows only one thread to acquire it at a time.
2. Ownership: Mutexes have an owner concept, meaning the thread that acquires the mutex is responsible for releasing it. In contrast, semaphores do not have an ownership concept. Any thread can increment or decrement the semaphore value.
3. Synchronization: Semaphores are used to synchronize access to shared resources among multiple threads or processes. They can be used for more complex synchronization patterns, such as allowing a certain number of threads to access a resource simultaneously. Mutexes are primarily used for mutual exclusion, ensuring that only one thread can access a critical section at a time.

**3.2 What is the difference between a task and a thread in a RTOS?**

1. Ans- > Scheduling: Tasks in an RTOS typically have a fixed priority and are scheduled based on priority levels. The scheduler assigns time slots to tasks based on their priority, and the execution order is determined by the priority scheduling algorithm. On the other hand, threads in an RTOS can have dynamic priorities and are typically scheduled using a preemptive scheduling algorithm, such as round-robin or priority-based preemptive scheduling.
2. Resource Usage: Tasks in an RTOS are generally designed to encapsulate a specific set of functionalities and may have their own dedicated resources, such as stack space, memory, and context. Each task has its own task control block (TCB) to store task-specific information. Threads, on the other hand, share the same resources, such as memory and stack space, and have a shared thread control block (TCB) to store thread-specific information. Threads are lightweight compared to tasks since they share resources.
3. Context Switching: Context switching between tasks in an RTOS involves saving and restoring the entire task context, including the program counter, stack pointer, and other CPU registers. Context switches between tasks are relatively expensive in terms of time and overhead. In contrast, context switching between threads is faster because threads share the same memory space and can switch their execution context more efficiently.
4. Synchronization and Communication: Tasks in an RTOS often use synchronization mechanisms like semaphores, mutexes, and message queues to communicate and coordinate with each other. These mechanisms allow tasks to synchronize their execution and share data safely. Threads in an RTOS can also use similar synchronization mechanisms for coordination and communication.