Module 4 - Deadlock & Synchronization

- 4.1 Learning Objectives
- 4.2 Introduction: The Problem of Shared Resource Access
- 4.3 Synchronization Mechanisms in FreeRTOS
- 4.4 Common Problems in Synchronization
- 4.5 Prevention and Handling Strategies

4.1 Learning Objectives

After completing this module, students are expected to be able to:

- Understand the importance of synchronization in multi-tasking systems and the risks of race conditions.
- Recognize and implement basic synchronization mechanisms: Mutexes and Semaphores in FreeRTOS.
- Differentiate the appropriate use cases for Mutexes versus Semaphores.
- Identify and understand common problems in concurrent systems: Deadlock, Starvation, and Priority Inversion.

4.2 Introduction: The Problem of Shared Resource Access

In multi-tasking systems, multiple tasks often need to access the same resource simultaneously, such as a global variable, a sensor interface, or a data structure. If this access is not managed properly, it can lead to data corruption, inconsistencies, or unexpected system behavior.

- **Race Condition**
  A race condition is a situation where the outcome of a process depends on the unpredictable sequence of execution of concurrent tasks. Imagine two tasks: one is responsible for incrementing a counter variable, and the other for decrementing it. If both tasks read the counter's value, modify it, and write it back at nearly the same time, one of the updates could be lost, resulting in an incorrect final value.

  To prevent race conditions, we need a mechanism that ensures only one task can access the shared resource at a time. This mechanism is known as Mutual Exclusion, or Mutex for short.

4.3 Synchronization Mechanisms in FreeRTOS

FreeRTOS provides several synchronization primitives, the most common of which are Mutexes and Semaphores. Both are built upon a basic data structure called a queue.

1. **Mutex (Mutual Exclusion)**
   A mutex can be thought of as a "key" to a resource. A task that wants to access the resource must first "take" the key. As long as that task holds the key, any other task that also needs the resource must wait. Once finished, the first task must "release" the key so another task can use it.

   Note: In FreeRTOS, the task that takes a mutex must be the same task that releases it.

2. **Semaphore**
   A semaphore is a more general synchronization mechanism than a mutex. It acts like a counter that controls access to a number of resources.

   Types of Semaphores:
   - Binary Semaphore : Has a maximum count of 1 (it can be either 0 or 1). It is often used for signaling between tasks (event synchronization) rather than for pure mutual exclusion, as it lacks a priority inheritance mechanism.
   - Counting Semaphore : Has a count value greater than 1. It is very useful for managing access to a pool of identical resources, such as connections to a server, memory buffers, or slots in a pool.

4.4 Common Problems in Synchronization

1. **Deadlock**
   A deadlock is a situation where two or more tasks are blocked forever, each waiting for a resource that is held by another task in the cycle.

   Example Deadlock Scenario:
   Task A successfully locks Mutex 1. Task B successfully locks Mutex 2. Task A now tries to lock Mutex 2, but it must wait because Mutex 2 is held by Task B. Task B now tries to lock Mutex 1, but it must wait because Mutex 1 is held by Task A. Both tasks are now waiting for each other indefinitely, and neither can proceed.

   The Four Conditions for Deadlock **(Coffman's Conditions)**:
   1. **Mutual Exclusion**: At least one resource must be non-sharable (can only be used by one task at a time).
   2. **Hold and Wait**: A task holds at least one resource while waiting for another resource held by a different task.
   3. **No Preemption**: A resource cannot be forcibly taken from the task holding it; it can only be released voluntarily.
   4. **Circular Wait**: A circular chain of two or more tasks exists, where each task is waiting for a resource held by the next task in the chain.

2. **Priority Inversion**
   Priority inversion is a scenario where a high-priority task is forced to wait for a much lower-priority task. This happens when the low-priority task is holding a lock (e.g., a mutex) that the high-priority task needs. The problem worsens if a medium-priority task starts running, as it will preempt the low-priority task, preventing it from releasing the lock. As a result, the high-priority task never gets a chance to run.

3. **Starvation**
   Starvation is a condition where a task is perpetually denied access to the resources it needs to complete its execution. This often happens to low-priority tasks that are constantly being preempted by higher-priority tasks, preventing them from ever getting their slice of CPU time.

4.5 Prevention and Handling Strategies

1. **Overcoming Deadlock**
   Since detecting and recovering from a deadlock in an embedded system is very difficult, the best approach is prevention. This is done by breaking one of the four Coffman conditions.
   - **Break Circular Wait**: Enforce a strict lock ordering for all resources. If all tasks are required to lock Mutex 1 before Mutex 2, the deadlock scenario described above could never happen. This is the most common and effective deadlock prevention strategy
   - **Break Hold and Wait**: Request all required resources at once.
   - **Detection and Recovery**: Mechanisms like a wait-for graph can be used to detect cycles. If a deadlock is detected, the system can recover by aborting one of the tasks (termination) or forcibly taking a resource (preemption). However, this is rarely implemented on microcontrollers.

2. **Overcoming Priority Inversion and Starvation**
   - **Priority Inheritance**: This is the solution to priority inversion. If a high-priority task is blocked waiting for a mutex held by a low-priority task, the low-priority task will temporarily "inherit" the priority of the high-priority task. This allows the low-priority task to run quickly, finish its work, and release the mutex as soon as possible.
     Note: Mutexes in FreeRTOS already implement this mechanism automatically.
   - **Priority Ceiling**: Each resource is assigned a priority ceiling, which is the highest priority level of any task that can access it. When a task locks the resource, its priority is immediately raised to that ceiling level until it releases it.
   - **Aging**: To prevent starvation, the priority of a task that has been waiting for a long time can be gradually increased. This way, a low-priority task will eventually have a high enough priority to be executed.
   - **Fair Scheduling**: Using a scheduling algorithm (like round-robin for tasks of the same priority) to ensure all tasks get a fair share of CPU time.