SNS COLLEGE OF TECHNOLOGY

Coimbatore-35
An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A++' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

19ECT312 – EMBEDDED SYSTEM DESIGN

III YEAR/ VI SEMESTER

UNIT 4 :**Embedded Operating System and Modelling**

TOPIC 4.8 : POSIX Semaphores

POSIX Semaphores

❖POSIX semaphores are synchronization primitives used in multi-threaded programming to control access to shared resources among concurrent threads. Unlike mutexes, which allow only one thread to access a resource at a time, semaphores can permit multiple threads to access a resource simultaneously, up to a specified limit. Semaphores maintain an internal counter that represents the number of available resources or permits, which threads acquire or release using the sem_wait() and sem_post() functions, respectively.

❖This flexibility makes semaphores suitable for scenarios where multiple threads need controlled access to shared resources or where synchronization needs to be more granular than what mutexes offer. However, improper usage of semaphores can lead to deadlocks or race conditions, so careful programming and understanding of concurrency principles are essential when working with POSIX semaphores.

POSIX Semaphores

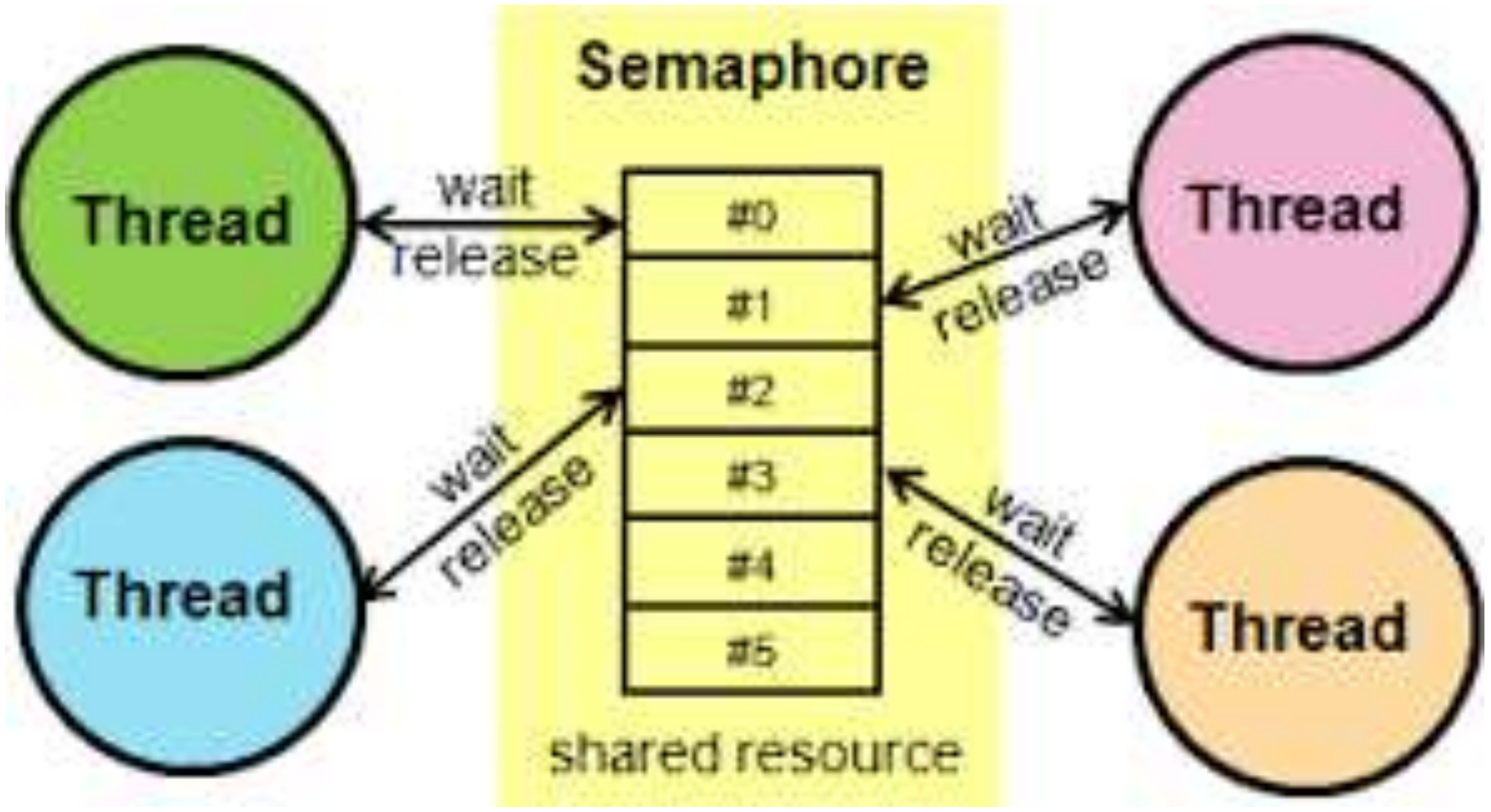
POSIX Semaphors API:

❖POSIX (Portable Operating System Interface) semaphores API provides a standardized interface for controlling semaphores in Unix-like operating systems. Semaphores are synchronization primitives used for inter-process communication and coordination.

❖In POSIX, semaphores are typically used to coordinate access to shared resources among multiple processes or threads. They can be thought of as counters with associated atomic operations for incrementing, decrementing, and testing their values.

1. **sem_init:** Initializes a semaphore with a specified initial value.
2. **sem_destroy**: Destroys a semaphore, releasing any associated resources.
3. **sem_wait:** Decrements the value of a semaphore. If the value is zero, the function blocks until the semaphore becomes non-zero.
4. **sem_post**: Increments the value of a semaphore.
5. **sem_getvalue**: Retrieves the current value of a semaphore without modifying it.

POSIX Semaphores



POSIX Semaphores

Advanced Semaphore Techniques:

Advanced semaphore techniques involve more sophisticated usage patterns and scenarios beyond basic synchronization. Here are a few advanced techniques:

1. **Multiple Semaphores for Resource Allocation**: Instead of using a single semaphore to control access to a shared resource, you can use multiple semaphores to manage different aspects of resource allocation. For example, one semaphore can control read access, another semaphore can control write access, and additional semaphores can manage other types of access or resource states.
2. **Counting Semaphores**: While binary semaphores have only two states (0 and 1), counting semaphores can have an initial count greater than 1. They are useful for scenarios where multiple instances of a resource can be allocated simultaneously. Threads or processes decrement the semaphore count when they acquire the resource and increment it when they release it.

POSIX Semaphores

❖**Semaphore Hierarchies**: In complex systems, you may need to manage multiple resources with different dependencies. Semaphore hierarchies involve organizing semaphores into a hierarchical structure, where acquiring a higher-level semaphore automatically acquires all lower-level semaphores. This technique helps prevent deadlocks and ensures consistent resource allocation.

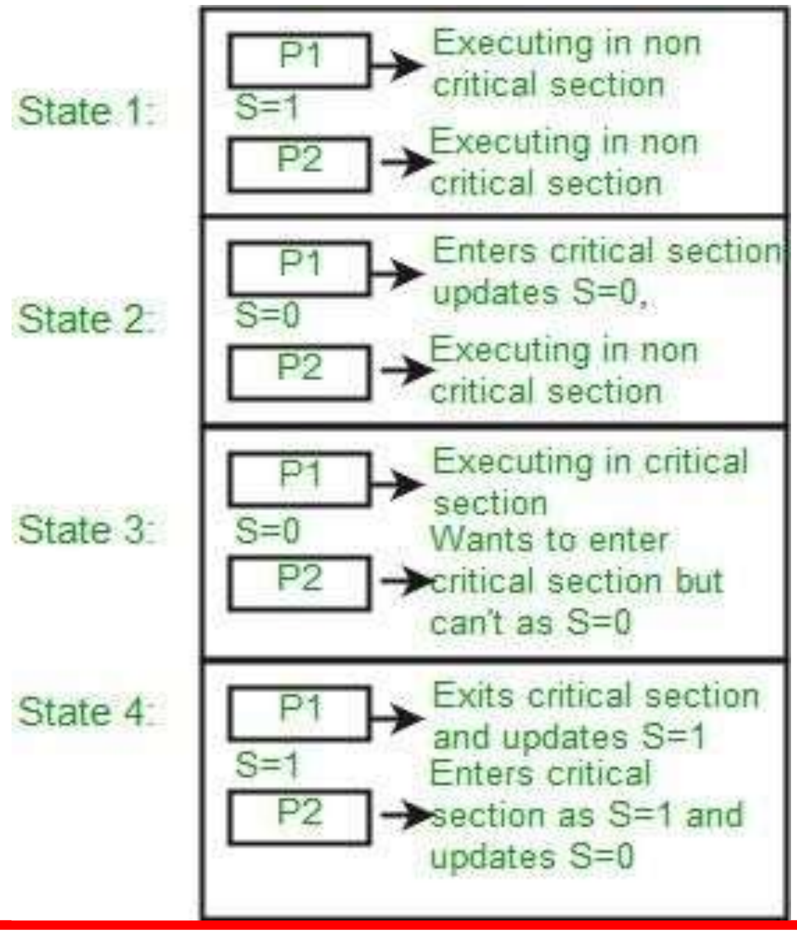
❖**Priority Inheritance**: Priority inversion can occur when a low-priority task holds a semaphore needed by a high-priority task, causing the high-priority task to wait longer than necessary. Priority inheritance is a technique where the priority of the low-priority task is temporarily raised to that of the high-priority task while it holds the semaphore. This ensures that the high-priority task can proceed without unnecessary delay.

❖**Readers-Writers Problem**: In scenarios where multiple threads need simultaneous read access to a shared resource but write access must be exclusive, specialized semaphore techniques like readers-writers locks can be employed. These locks allow multiple readers to access the resource concurrently while ensuring exclusive access for writers.

POSIX Semaphores

❖**Dynamic Semaphores**: Instead of statically defining semaphores at compile time, dynamic semaphores are created and destroyed at runtime as needed. This flexibility is useful in scenarios where the number of resources or threads is not known in advance.

❖**Semaphore Timeouts**: Some semaphore implementations support timeouts, allowing threads to wait for a semaphore for a specified period before giving up. Timeout mechanisms are essential for preventing indefinite waits and handling exceptional conditions.


State 1:
P1 → Executing in non critical section
S=1
P2 → Executing in non critical section
State 2:
P1 → Enters critical section updates S=0,
S=0
P2 → Executing in non critical section
State 3:
P1 → Executing in critical section
S=0
P2 → Wants to enter critical section but can't as S=0
State 4:
P1 → Exits critical section and updates S=1
S=1
P2 → Enters critical section as S=1 and updates S=0

POSIX Semaphores

Advantage:

❖**Portability**: Standardized interface across Unix-like operating systems ensures compatibility and easy migration of code.

❖**Inter-Process Communication (IPC)**: Facilitates synchronization and communication between multiple processes.

❖**Scalability**: Adaptable for simple to complex synchronization needs in applications with multiple processes or threads.

❖**Flexibility**: Offers binary and counting semaphore types for diverse synchronization requirements.

❖**Efficiency**: Implemented with efficient algorithms and system calls, minimizing overhead in memory and processing time.

❖**Ease of Use**: Simple API with intuitive functions for semaphore management simplifies development and maintenance.

POSIX Semaphores

Limitations:

❖**Limited Functionality**: Lack advanced features like deadlock detection and priority inheritance found in other synchronization primitives.

❖**Complex Error Handling**: Error handling can be intricate, requiring careful attention to return values and error codes.

❖**Kernel Dependency**: Performance and behavior may vary based on the underlying operating system and kernel version.

❖**Resource Overhead**: Each semaphore consumes system resources, potentially becoming problematic in applications requiring many semaphores.

❖**Portability Challenges**: While aiming for portability, differences in behavior and implementation across platforms may arise.

❖**Risk of Deadlocks and Races**: Improper use can lead to deadlocks or race conditions, demanding careful programming to avoid.

Thank you