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

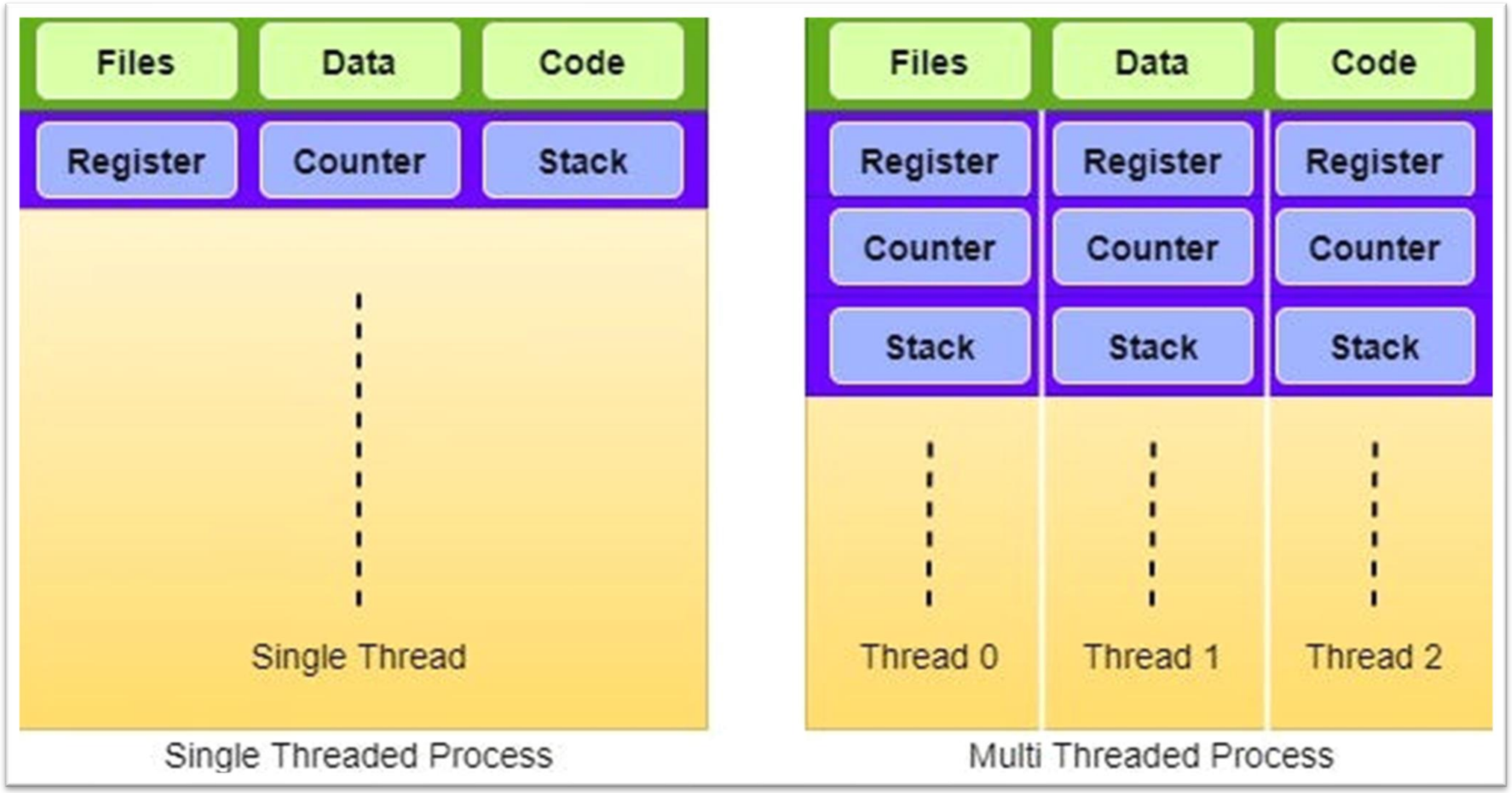
19ECT312 – EMBEDDED SYSTEM DESIGN

III YEAR/ VI SEMESTER

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

TOPIC :POSIX Thread Programming , POSIX Semaphores

POSIX Thread Programming



POSIX Thread Programming

❖POSIX threads, commonly known as Pthreads, are a threading standard that allows multiple threads to coexist within the same process, sharing resources but executing independently. In embedded systems, Pthreads facilitate concurrent task execution, which is essential for optimizing performance and responsiveness.

❖Pthreads offer a range of functionalities in embedded systems, such as thread synchronization with mutexes and condition variables, thread management, and real-time scheduling. These capabilities are crucial for embedded applications where timing and resource constraints are critical.

POSIX Thread Programming

Thread Synchronization:

❖Thread synchronization is a programming concept that ensures the orderly execution of multiple threads within a concurrent processing environment. It involves coordinating thread access to shared resources to prevent conflicts and ensure data integrity.

❖**Mutexes**: Mutexes (short for mutual exclusion) are synchronization primitives used to protect shared resources from simultaneous access by multiple threads. In embedded systems, mutexes are commonly employed to prevent data corruption when multiple threads attempt to access critical sections of code or shared variables concurrently.

❖**Semaphores**: Semaphores are another synchronization mechanism used in embedded systems. They provide a way to control access to a shared resource by allowing a fixed number of threads to access it simultaneously. Semaphores are often used to manage access to finite resources, such as hardware peripherals or memory buffers.

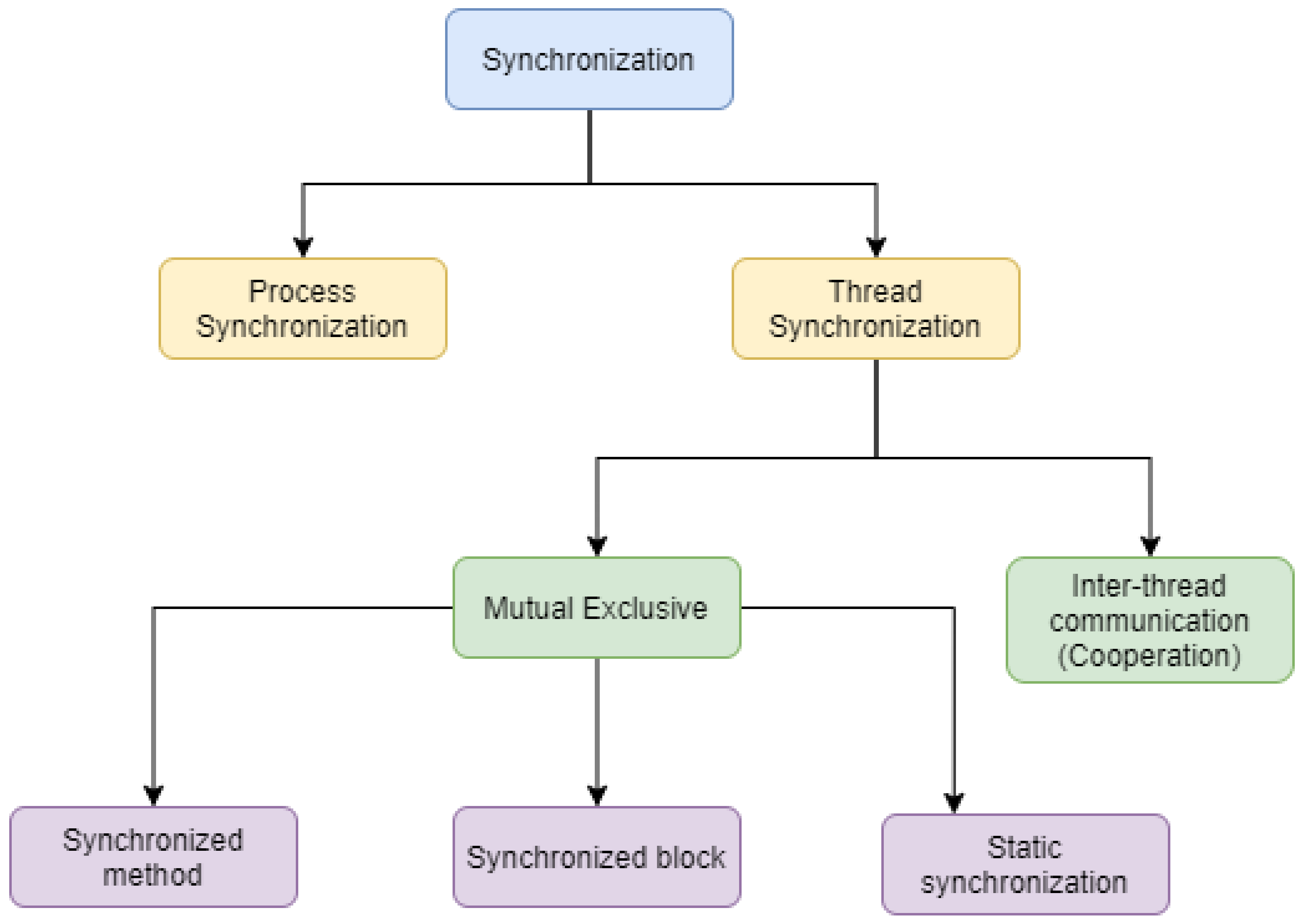
POSIX Thread Programming

❖**Critical Sections**: Critical sections are parts of code that must be executed atomically, without interruption from other threads. In embedded systems, critical sections are typically protected by mutexes or other synchronization primitives to prevent race conditions and ensure data consistency.

❖**Interrupt Handling**: Embedded systems often rely on interrupts to handle time-critical events and asynchronous I/O operations. Proper synchronization techniques, such as disabling interrupts or using atomic operations, are essential to ensure data integrity when accessing shared resources from interrupt service routines (ISRs) and regular threads.

❖**Event Flags**: Event flags are used to signal and synchronize between threads in embedded systems. Threads can wait for specific events to occur by blocking on event flags, and other threads can set or clear these flags to notify waiting threads of significant events or conditions.

POSIX Thread Programming



POSIX Thread Programming

Thread Safety:

- ❖ Thread safety refers to the property of a program or system where it can handle multiple threads executing concurrently without encountering data races, deadlocks, or other synchronization issues. Ensuring thread safety is crucial in multi-threaded environments to prevent unpredictable behavior and maintain data integrity. Here's a concise overview:
- ❖ **Atomicity**: Operations that involve multiple steps should appear as a single, indivisible operation to other threads. Atomic operations ensure that threads cannot interrupt each other midway through an operation, preventing inconsistent state.
- ❖ **Mutual Exclusion**: Concurrent access to shared resources should be controlled to avoid race conditions. Techniques such as mutexes, semaphores, or critical sections ensure that only one thread can access a resource at a time, preventing data corruption.

POSIX Thread Programming

❖**Synchronization**: Threads need to synchronize their actions to avoid conflicts and maintain consistency. Synchronization primitives like mutexes, condition variables, and barriers facilitate coordination between threads, ensuring that they execute in a synchronized manner.

❖**Memory Visibility**: Changes made by one thread to shared variables should be visible to other threads. Memory barriers, locks, and atomic operations ensure proper memory visibility, preventing inconsistencies due to caching and compiler optimizations.

❖**Reentrancy**: Functions and code segments should be designed to be reentrant, meaning they can be safely called by multiple threads simultaneously without interfering with each other's execution. Reentrant code avoids issues related to shared data and maintains thread safety.

POSIX Thread Programming

Thread Pool:

❖A thread pool is a collection of pre-initialized threads that are ready to perform tasks. Instead of creating a new thread for each task, threads from the pool are assigned tasks as needed. This approach reduces overhead associated with thread creation and destruction.

❖**Task Queue**: Thread pools often utilize a task queue, also known as a work queue or job queue, to store tasks that need to be executed. When a task is submitted to the thread pool, it is added to the task queue.

❖**Task Submission**: Applications submit tasks to the thread pool instead of directly creating threads. Tasks can be functions, methods, or any unit of work that needs to be executed concurrently.

❖**Task Execution**: Idle threads in the thread pool continuously monitor the task queue for new tasks. When a thread becomes available, it retrieves a task from the queue and executes it. This process continues until the thread pool is shut down.

POSIX Thread Programming

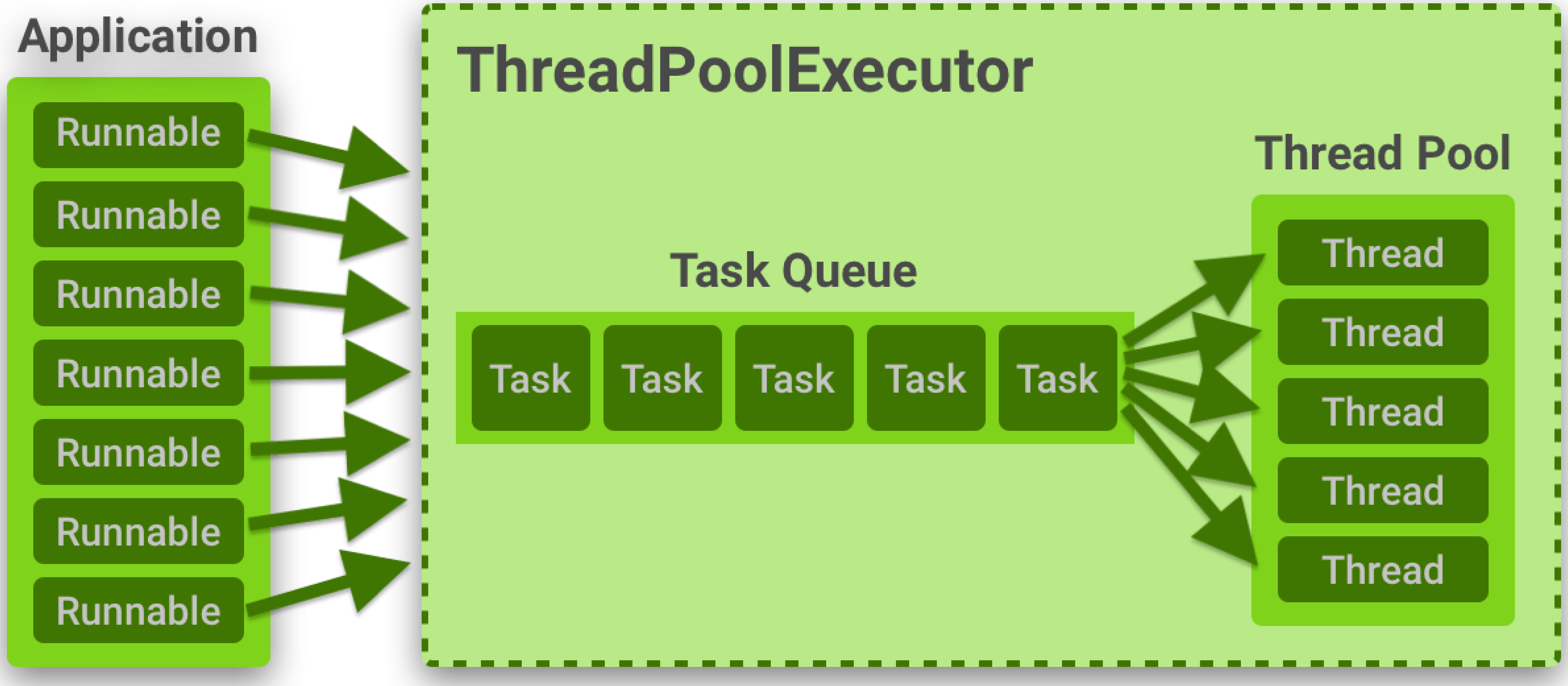
❖**Thread Lifespan**: Threads in the pool are long-lived and remain active throughout the lifespan of the application. After executing a task, a thread returns to the idle state, ready to accept and execute another task.

❖**Resource Management**: Thread pools allow for efficient management of system resources by limiting the total number of concurrent threads. This prevents resource exhaustion and improves overall system stability.

❖**Performance Optimization**: Thread pools help improve application performance by reducing the overhead associated with thread creation and destruction. Reusing threads from the pool eliminates the need for frequent context switching and thread setup overhead.

❖**Dynamic Sizing**: Some thread pool implementations support dynamic resizing, allowing the pool size to adjust based on workload or system conditions. This flexibility ensures optimal resource utilization without compromising performance.

POSIX Thread Programming



POSIX Thread Programming

Parallelism:

❖Parallelism refers to the simultaneous execution of multiple tasks or processes to improve performance and efficiency.

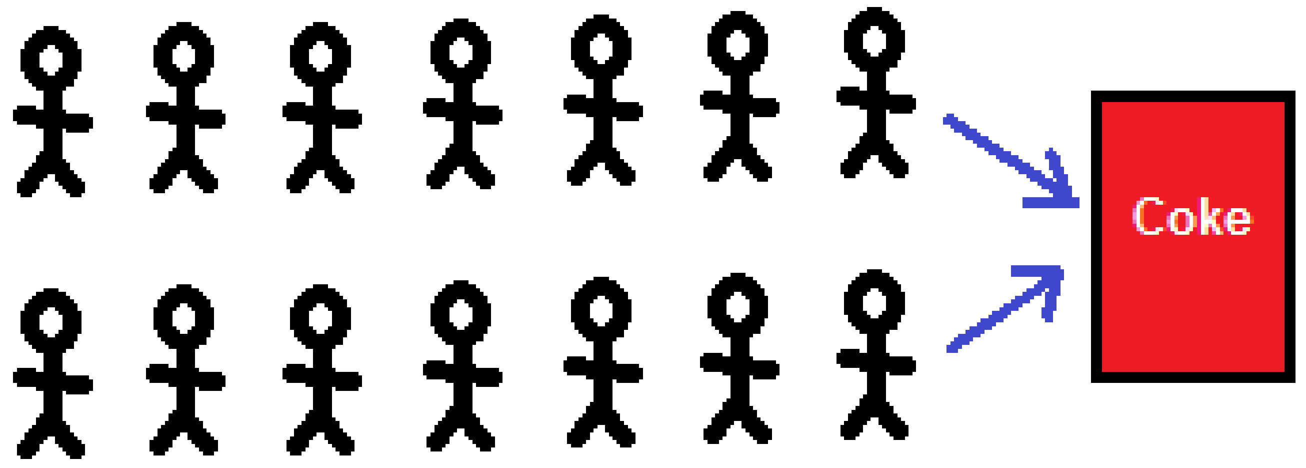
❖In parallel computing, tasks are divided into smaller subtasks that can be executed concurrently on multiple processing units, such as CPU cores or distributed computing nodes.

Concurrency:

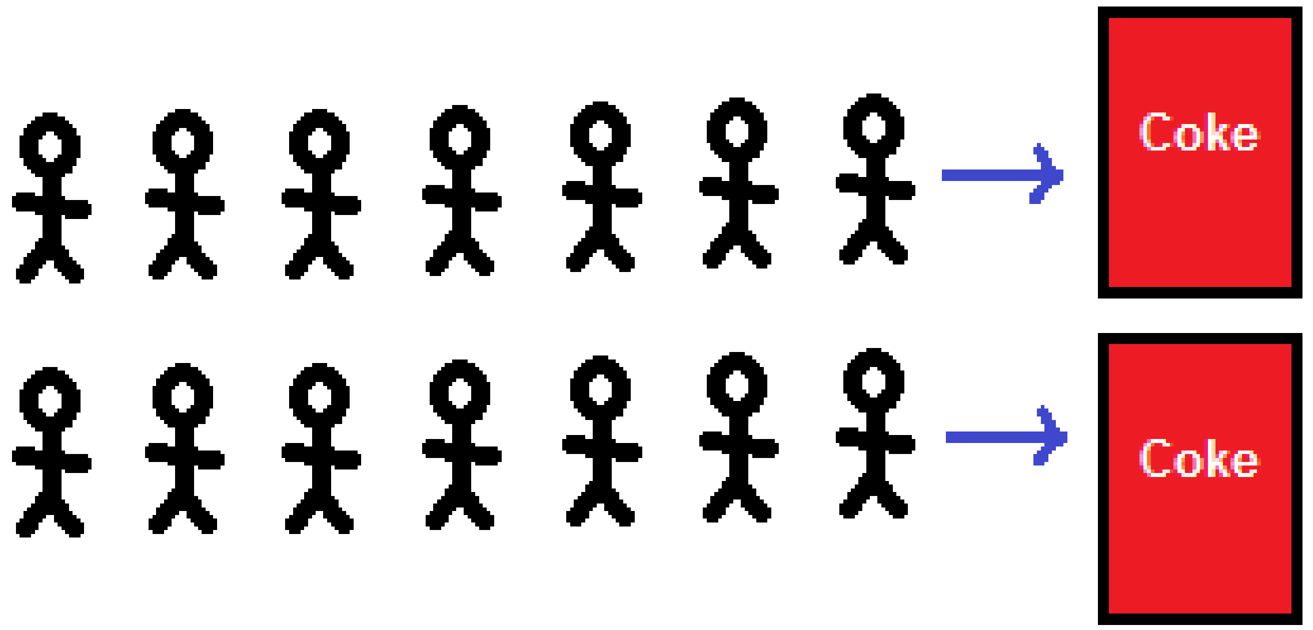
❖Concurrency, on the other hand, involves the execution of multiple tasks or processes seemingly simultaneously, but not necessarily concurrently.

❖Concurrent programming focuses on managing the execution flow of multiple tasks, allowing them to progress independently and make progress concurrently.

POSIX Thread Programming



Concurrent: 2 queues, 1 vending machine



Parallel: 2 queues, 2 vending machines

POSIX Thread Programming

Real World Application:

❖**Web Servers**: Web servers handle multiple client requests concurrently. POSIX threads can be used to create a pool of worker threads that handle incoming requests, allowing the server to serve multiple clients simultaneously without blocking.

❖**Multimedia Processing**: Applications that deal with multimedia processing, such as video editing software or audio processing tools, often benefit from parallelism. POSIX threads can be used to parallelize tasks like video encoding, decoding, and rendering to improve performance.

❖**Database Systems**: Database management systems (DBMS) need to handle multiple concurrent queries and transactions efficiently. POSIX threads can be employed to handle query processing, transaction management, and concurrency control mechanisms like locking and transactions.

POSIX Thread Programming

❖**Embedded Systems**: Embedded systems with multitasking requirements, such as real-time control systems or IoT devices, can benefit from POSIX thread programming. Threads can be used to handle various tasks concurrently, such as sensor data processing, communication protocols, and user interface updates.

❖**Parallel Algorithms**: Parallel algorithms, such as sorting, searching, and graph processing, can leverage POSIX threads to divide the workload across multiple threads and exploit parallelism in modern multi-core processors.

❖**Parallel File Processing**: Applications that involve processing large volumes of data stored in files can benefit from POSIX thread programming. Multiple threads can be used to read, process, and write data concurrently, improving overall throughput and reducing processing time.

❖**Parallel Computing**: High-performance computing (HPC) applications often use POSIX threads for parallel computing tasks like numerical simulations, scientific computing, and data analysis

POSIX Thread Programming

Challenges:

❖**Concurrency and Parallelism**: Efficiently leveraging multi-core and many-core processors for improved performance.

❖**Scalability**: Developing techniques to handle large-scale systems with increasing core counts.

❖**Performance Portability**: Ensuring performance across diverse hardware platforms.

❖**Fault Tolerance and Reliability**: Enhancing error-handling mechanisms and resilience in multi-threaded applications.

❖**Energy Efficiency**: Minimizing energy consumption while maintaining performance.

POSIX Thread Programming

Challenges:

❖**Concurrency Management**: Effectively managing concurrent execution of multiple threads to avoid race conditions and deadlocks.

❖**Synchronization**: Ensuring proper synchronization between threads to prevent data corruption and maintain consistency.

❖**Scalability**: Scaling thread-based applications to handle increasing core counts and workload diversity on modern multi-core and many-core processors.

❖**Performance Optimization**: Optimizing thread management, load balancing, and task scheduling to maximize performance and efficiency.

❖**Fault Tolerance**: Implementing robust error-handling mechanisms and fault-tolerant synchronization primitives to enhance application reliability.

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

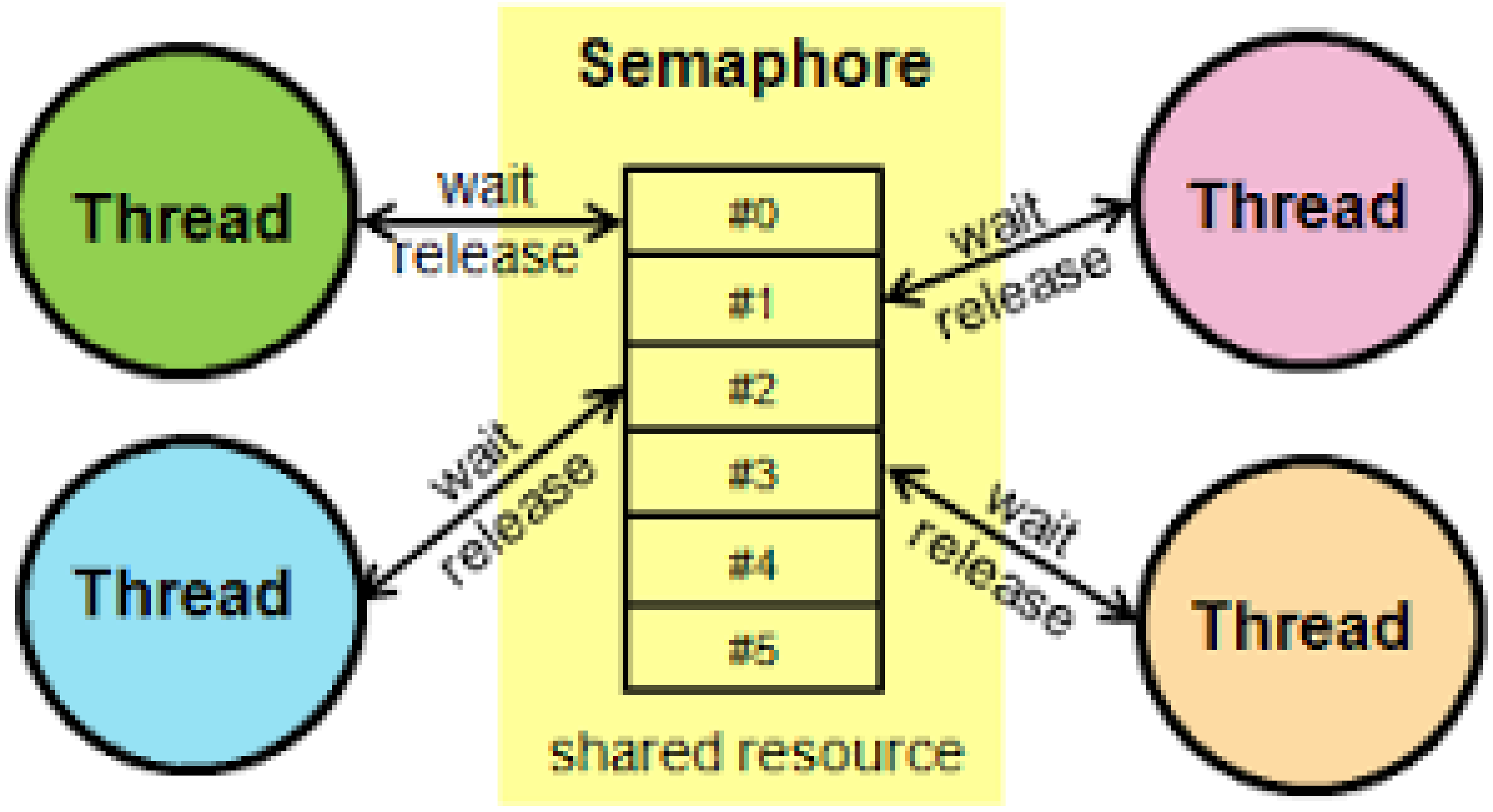
POSIX Semaphors API:

❖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.

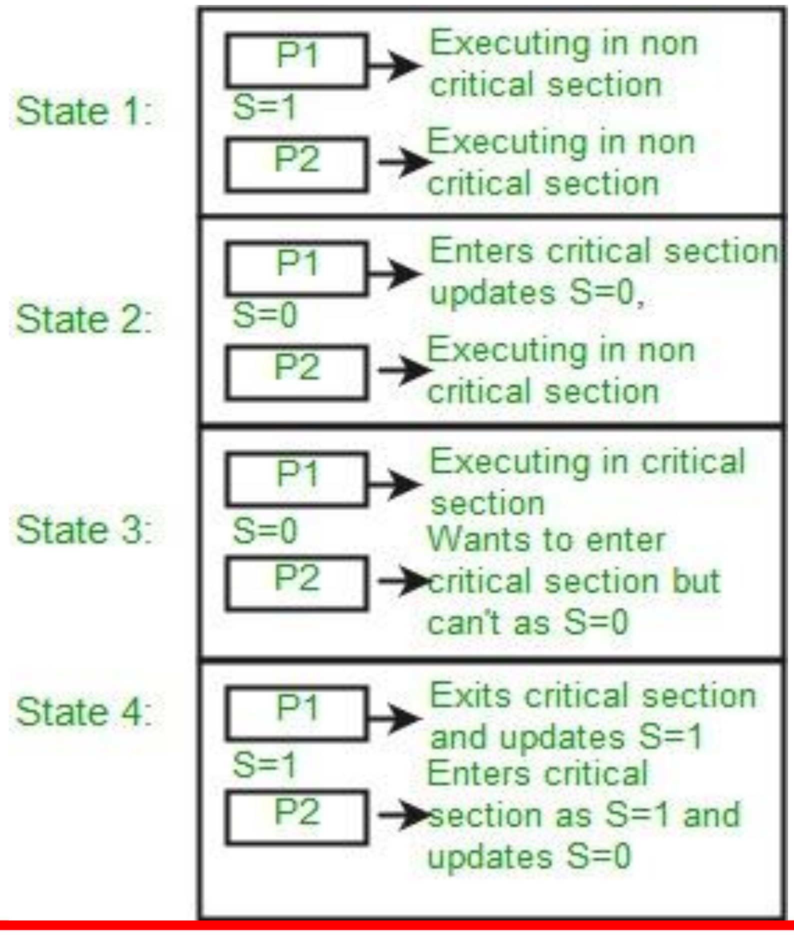
POSIX Semaphores

- ❖**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