# Semaphores in RTOS

# Introduction

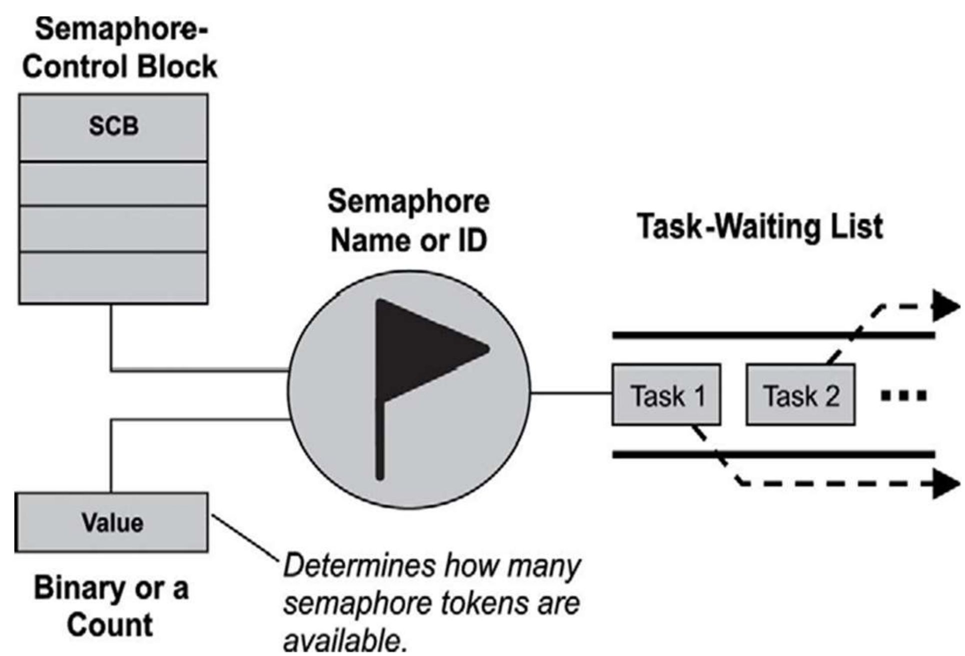
- Multiple concurrent threads ***(tasks)*** of execution within an application must be able to
  - ***synchronize their execution*** and
  - ***coordinate mutually exclusive access to shared resources***.
- This can be accomplished using RTOS kernels ***semaphore object*** and ***associated semaphore management*** services.

# Defining Semaphores

- Definition
  - A *semaphore* (sometimes called a *semaphore token*) is a kernel object that one or more threads of execution can acquire or release for the purposes of synchronization or mutual exclusion.

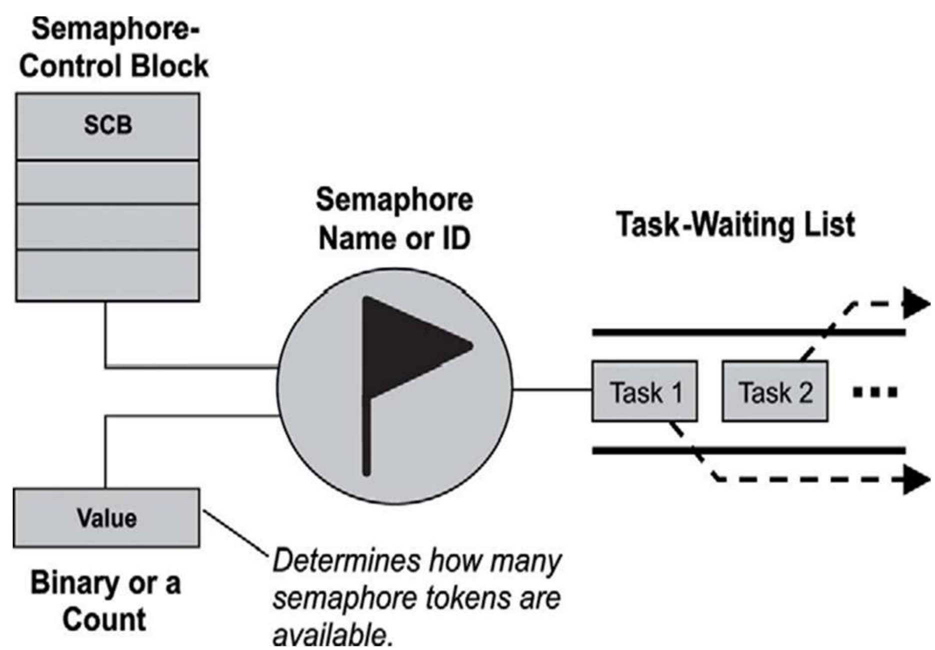
# Defining Semaphores

- When a semaphore is first created, the kernel assigns to it an associated
  - semaphore control block (SCB),
  - a unique ID,
  - a value (binary or a count), and
  - a task-waiting list



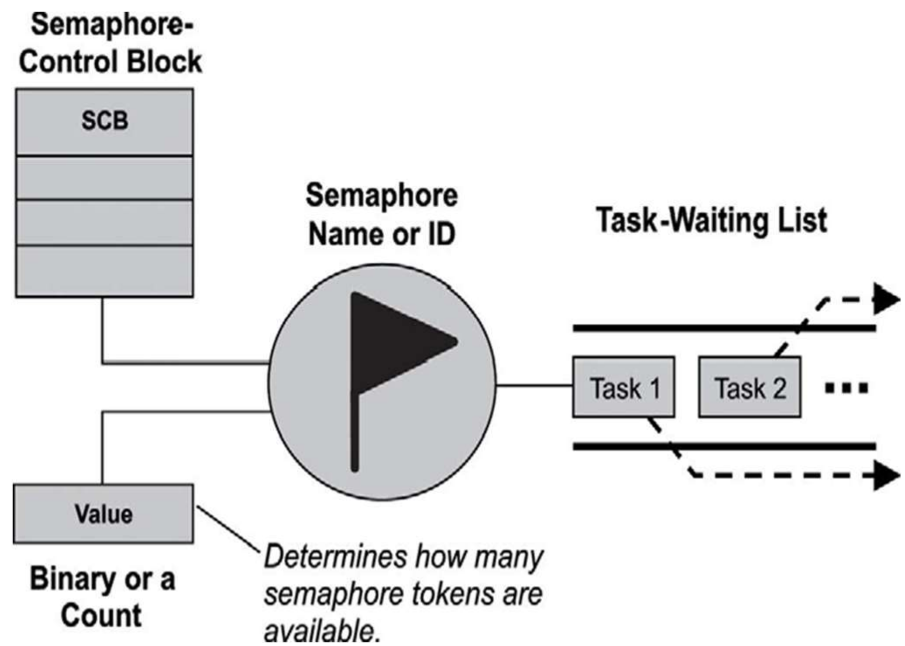
# Defining Semaphores

- Counting semaphore
  - A counter is maintained for token counting
  - When a task acquires the semaphore, the token count is decremented; as a task releases the semaphore, the count is incremented.



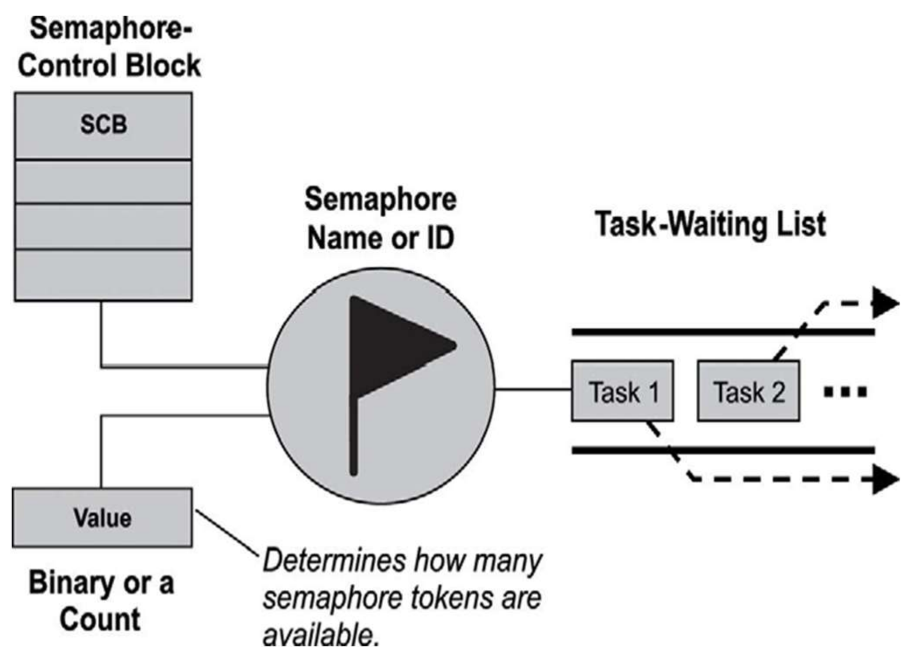
# Defining Semaphores

- If the token count reaches 0 that no more resources is available and  the requesting task is ***blocked.***



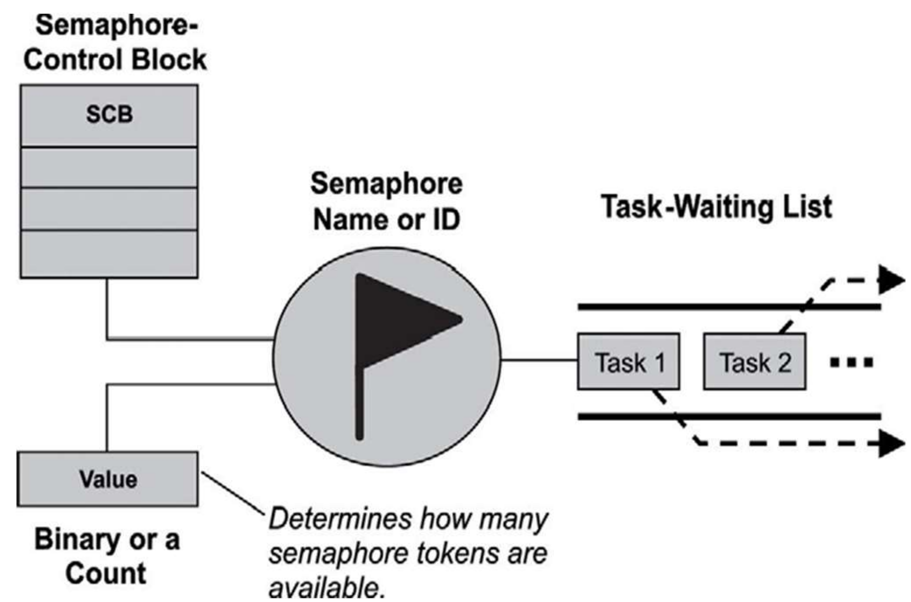
# Defining Semaphores

- The ***task-waiting list*** tracks all blocked tasks of the semaphore.
- The list is implemented either in FIFO order or highest priority first order.



# Defining Semaphores

- When an semaphore/resource becomes available, the kernel allows the first task in the task- waiting list to acquire it.
  - if the task has the highest priority, the task moves to the running state, otherwise it moves
  - to the ready state wait for its turn
  - Note that the exact implementation of a task-waiting list can vary from one kernel to another.

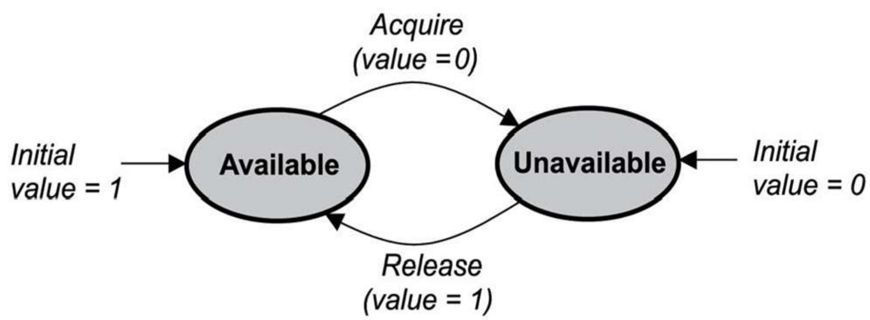


# Defining Semaphores

- A kernel can support many different types of semaphores, including
  - binary,
  - counting, and
  - mutual-exclusion (mutex) semaphores.

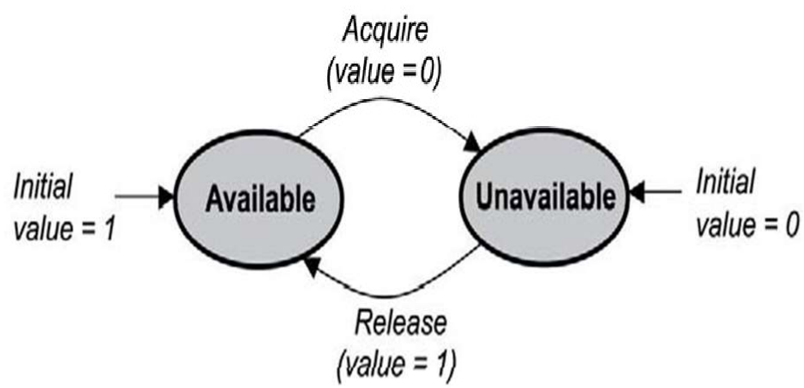
# Binary Semaphores

- A *binary semaphore* can have a value of either 0 or 1
  - 0, semaphore is *unavailable* (or *empty)*
  - 1, the semaphore is *available* (or *full* ).



# Binary Semaphores

- Binary semaphores are treated as ***global resources***,
  - they are shared among ***all tasks*** that need them.
  - Making the semaphore a global resource allows any task to release it, even if the task did not initially acquire it???.

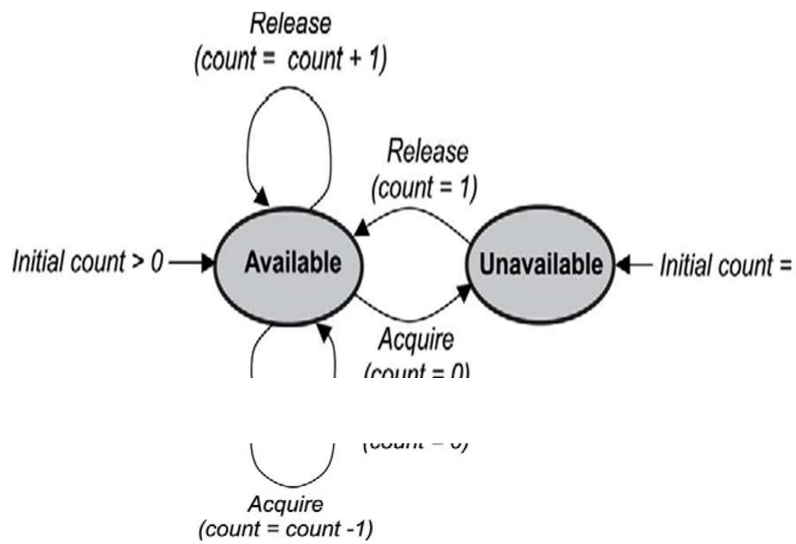


## Counting Semaphores

- A *counting semaphore* uses a count to allow it to be acquired or released multiple times.

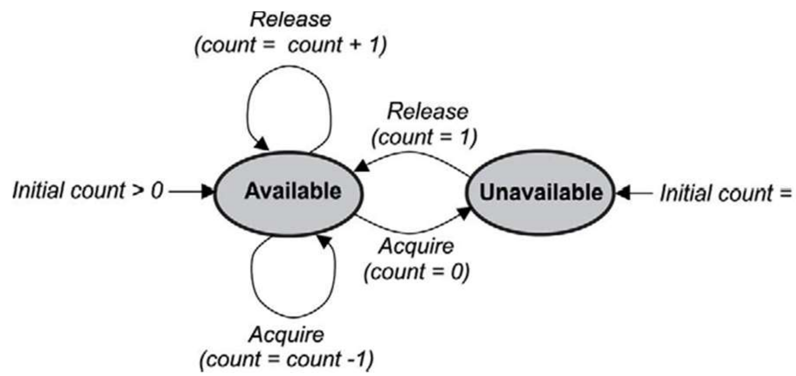
# Counting Semaphores

- If the initial count is
  - 0, the counting semaphore is created in the unavailable state.
  - greater than 0, the semaphore is created in the available state, and the number of tokens it has equals its count



# Counting Semaphores

- counting semaphores are **global resources** that can be shared by all tasks that need them

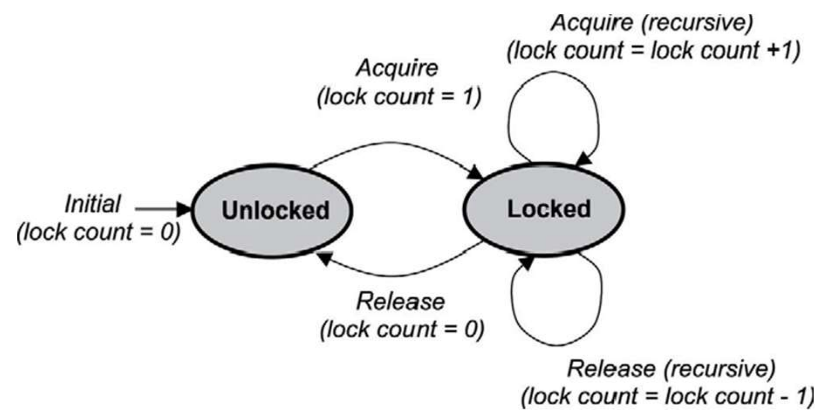


# Mutual Exclusion (Mutex) Semaphores

- *A mutual exclusion (mutex) semaphore* is a special binary semaphore that supports
  - **ownership**,
  - recursive access,
  - task deletion safety, and
  - one or more protocols for avoiding problems inherent to mutual exclusion.

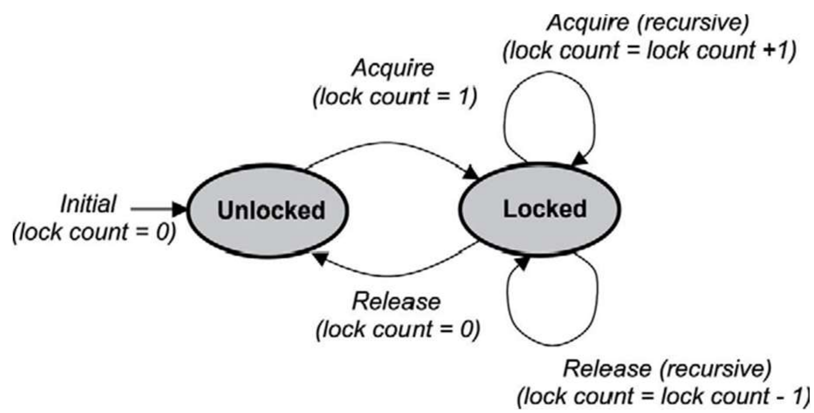
# Mutual Exclusion (Mutex) Semaphores

- the states of a mutex are
  - 0 *unlocked* or
  - 1 *locked*



# Mutual Exclusion (Mutex) Semaphores

- A mutex is initially created in the ***unlocked state***, in which it can be acquired by a task.
- After being acquired, the mutex moves to the ***locked state***

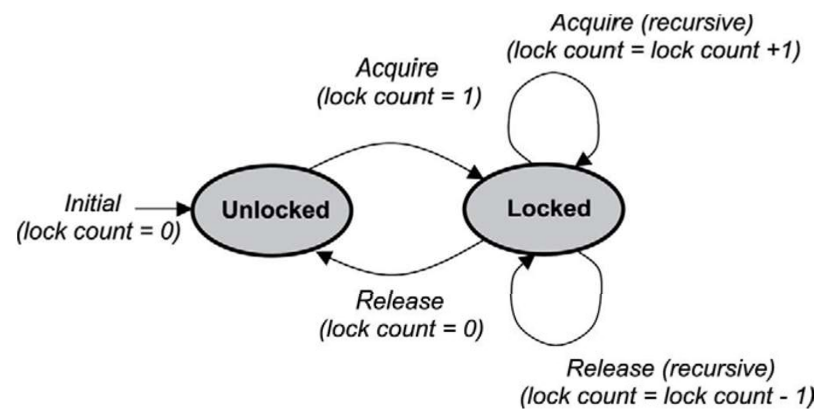


# Mutual Exclusion (Mutex) Semaphores

- Mutex Ownership
  - *Ownership* of a mutex is gained when a **task** first locks the mutex by acquiring it.
  - Conversely, a task loses ownership of the mutex when it unlocks it by releasing it.

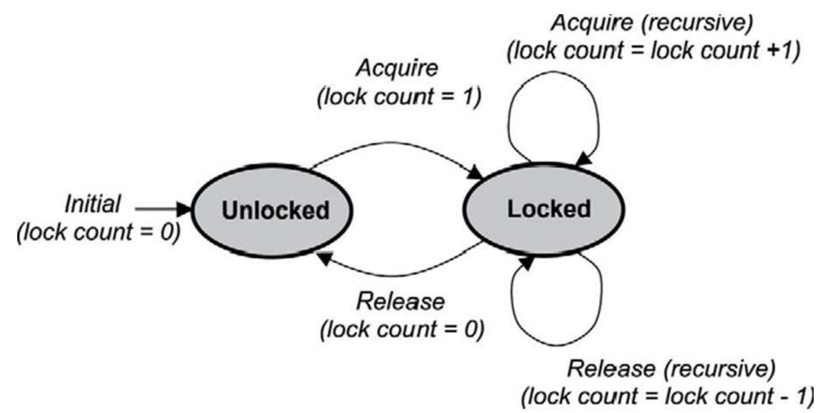
# Mutual Exclusion (Mutex) Semaphores

- Recursive Locking
  - allows the task that owns the mutex to acquire it ***multiple times*** in the locked state
  - The mutex with recursive locking is called a ***recursive mutex***



# Mutual Exclusion (Mutex) Semaphores

- Recursive Locking
  - A recursive mutex allows ***nested attempts to lock the mutex to succeed***, rather than cause *deadlock* ,




***See 6.4.5 Recursive Shared-Resource-Access Synchronization***

# Mutual Exclusion (Mutex) Semaphores

- Difference between Counting Sepaphore and Mutex
  - The count used for the mutex tracks the number of times that ***the task owning the mutex*** has locked or unlocked the mutex.
  - The count used for the counting semaphore tracks the number of tokens that have been ***acquired or released by any task***.

# Mutual Exclusion (Mutex) Semaphores

- Task Deletion Safety
  - Premature task deletion is avoided by using ***task deletion locks*** when a task locks and unlocks a mutex
  - Enabling this capability within a mutex ensures that ***while a task owns the mutex, the task cannot be deleted***

# Mutual Exclusion (Mutex) Semaphores

- Priority Inversion Avoidance
  - *Priority inversion* occurs when a higher priority task is blocked and is waiting for a resource being used by a lower priority task, which has itself been preempted by an unrelated medium-priority task. In this situation, the higher priority task's priority level has effectively been inverted to the lower priority task's level.

# Mutual Exclusion (Mutex) Semaphores

- Priority Inversion Avoidance
- Two common protocols used for avoiding priority inversion include:
  - **priority inheritance protocol**
  - **ceiling priority protocol**

apply to the task that owns the mutex.

# Mutual Exclusion (Mutex) Semaphores

- **priority inheritance protocol**—
  - ensures that the priority level of the lower priority task that has acquired the mutex is ***raised to that of the higher priority task that has requested the mutex*** when inversion happens.
  - The priority of the raised task is lowered to its original value after the task releases the mutex that the higher priority task requires.

# Mutual Exclusion (Mutex) Semaphores

- **ceiling priority protocol**—
  - ensures that the priority level of the task that acquires the mutex is automatically ***set to the highest priority of all possible tasks*** that might request that mutex when it is first acquired until it is released.

# Typical Semaphore Operations

- Typical operations that developers might want to perform with the semaphores in an application include:
  - creating and deleting semaphores,
  - acquiring and releasing semaphores,
  - clearing a semaphore’s task-waiting list, and
  - getting semaphore information.

# Creating and Deleting Semaphores

- Create and Delteting

Table 6.1: Semaphore creation and deletion operations.

| Operation | Description |
|---|---|
| Create | Creates a semaphore |
| Delete | Deletes a semaphore |

```
create_binary_semaphore(initial_state, task_waiting_order);
create_counting_semaphore(initial_count, task_waiting_order);
create_mutex_semaphore(task_waiting_order);
```

# Creating and Deleting Semaphores

- different calls might be used for creating binary, counting, and mutex semaphores, as follows:
  - **binary**—specify the initial semaphore state and the task-waiting order.
  - **counting**—specify the initial semaphore count and the task-waiting order.

```
delete_semaphore(semaphore_id);
```

# Creating and Deleting Semaphores

- different calls might be used for creating binary, counting, and mutex semaphores, as follows:
  - **mutex**—specify the task-waiting order and enable task deletion safety, recursion, and priority-inversion avoidance protocols, if supported.

```
create_mutex_semaphore(task_waiting_order, enable_task_deletion_safety,
enable_recursion, enable_priority_inversion_avoidance);
```

# Creating and Deleting Semaphores

- Deleting (implementation dependent)

  - Semaphores can be deleted from within any task by specifying their IDs and making semaphore-deletion calls.
  - Deleting a semaphore is not the same as releasing it. When a semaphore is deleted, blocked tasks in its task-waiting list are unblocked and moved either to the ready state or to the running state (if the unblocked task has the highest priority).
  - Any tasks (not in the task waiting list), however, that try to acquire the deleted semaphore return with an error because the semaphore no longer exists.

# Creating and Deleting Semaphores

- Deleting
  - Additionally, do not delete a semaphore while it is in use (e.g., acquired). This action might result in data corruption or other serious problems if the semaphore is protecting a shared resource or a critical section of code.

# Acquiring and Releasing Semaphores

Table 6.2: Semaphore acquire and release operations.

| **Operation** | **Description** |
|---|---|
| Acquire *( take/ sm_p/ pend/ lock )* | Acquire a semaphore token |
| Release *( give/ sm_v/ post/ unlock )* | Release a semaphore token |

# Acquiring and Releasing Semaphores

- Tasks typically make a request to ***acquire*** a semaphore in one of the following ways:
  - **Wait forever**—task remains blocked until it is able to acquire a semaphore.
  - **Wait with a timeout**—task remains blocked until it is able to acquire a semaphore or until a set interval of time, called the *timeout interval* , passes.
  - **Do not wait**—task makes a request to acquire a semaphore token, but, if one is not available, the task does not block.

# Acquiring and Releasing Semaphores

- Release
  - Any task can ***release*** a binary or counting semaphore;
  - however, a mutex can only be released (unlocked) by the task that first acquired (locked) it.

# Acquiring and Releasing Semaphores

- Releaase
  - Note that incorrectly releasing a binary or counting semaphore can result in losing mutually exclusive access to a shared resource or in an I/O device malfunction.

# Acquiring and Releasing Semaphores

- ISR and Semaphores
  - Note that ISRs can also release binary and counting semaphores.
  - Note that most kernels ***do not*** support ISRs locking and unlocking mutexes, as it is not meaningful to do so from an ISR.
  - It is also not meaningful to acquire either binary or counting semaphores inside an ISR.

# Clearing Semaphore Task-Waiting Lists

- To ***clear*** all tasks waiting on a semaphore task-waiting list

Table 6.3: Semaphore unblock operations.

| **Operation** | **Description** |
|---|---|
| Flush | Unblocks all tasks waiting on a semaphore |

# Clearing Semaphore Task-Waiting Lists

- ***thread rendezvous*** *that implement flush in* synchronization tasks
  - multiple tasks to complete certain activities first and then
  - block while trying to acquire a common semaphore that is made unavailable, after
  - the last task finishes doing what it needs to, the task can execute a semaphore flush operation on the common semaphore.
  - This operation frees all tasks waiting in the semaphore's task waiting list.

# Getting Semaphore Information

- obtain semaphore information to perform monitoring or debugging

Table 6.4: Semaphore information operations.

| Operation | Description |
|---|---|
| Show info | Show general information about semaphore |
| Show blocked tasks | Get a list of IDs of tasks that are blocked on a semaphore |

# Typical Semaphore Use

- Semaphores are useful either for synchronizing execution of multiple tasks or for coordinating access to a shared resource.

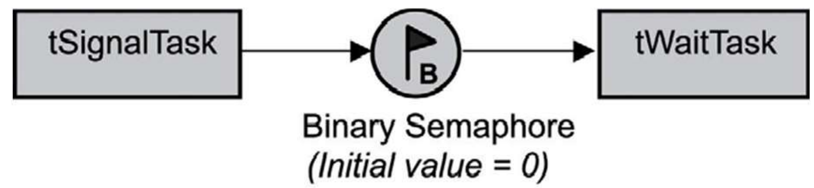
# Typical Semaphore Use

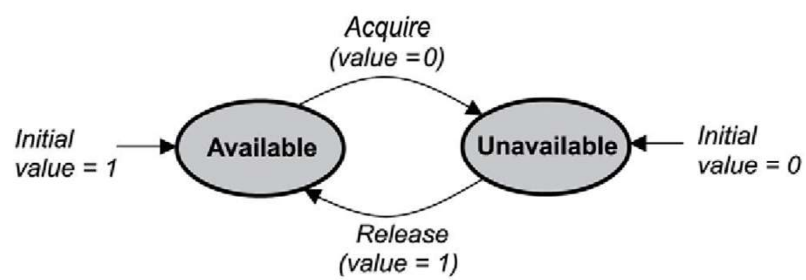
- Semaphores are useful
  - wait-and-signal synchronization,
  - multiple-task wait-and-signal synchronization,
  - credit-tracking synchronization,
  - single shared-resource-access synchronization,
  - recursive shared-resource-access synchronization, and
  - multiple shared-resource-access synchronization.

  and etc.

# Wait-and-Signal Synchronization

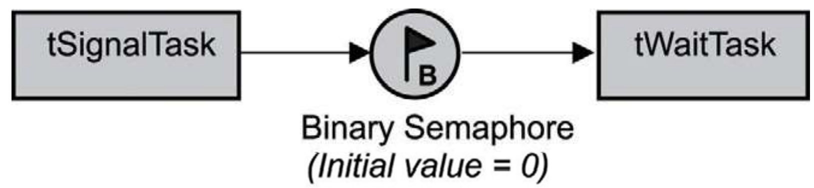
- Two tasks can communicate for the purpose of synchronization without exchanging data.



# Wait-and-Signal Synchronization

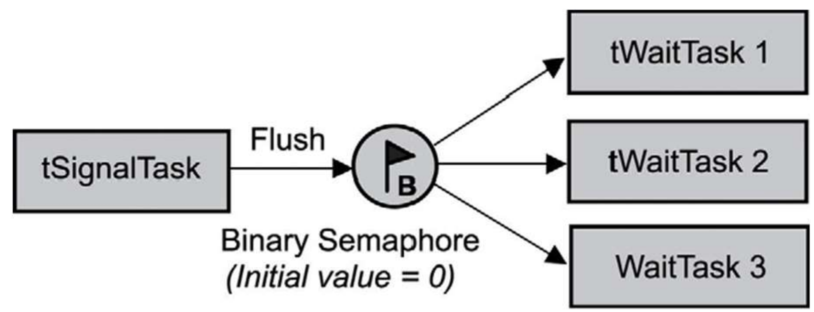
1. binary semaphore is initially unavailable (value of 0).
2. tWaitTask has higher priority and runs first and blocked by the semaphore
3. lower priority tSignalTask a chance to run and release semaphore
4. tWaitTask unblocked and preempts tSignalTask and starts to execute




```
tWaitTask ( ) {
        :
    Acquire binary semaphore token
        :
}
tSignalTask ( ) {
        .
    Release binary semaphore token
        :
}
```

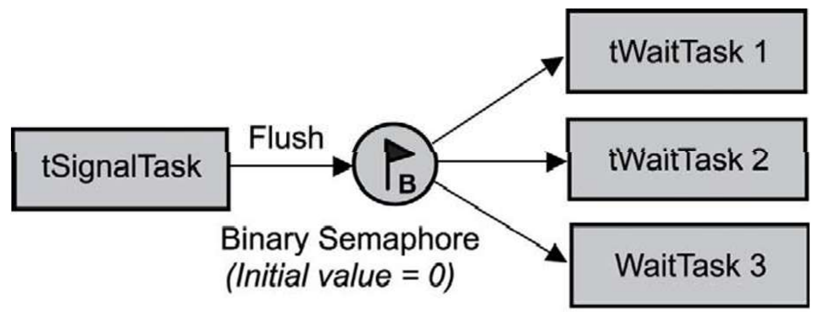
# Multiple-Task Wait-and-Signal Synchronization

- To coordinate the synchronization of more than two tasks, use the flush operation on the task-waiting list of a binary semaphore,



# Multiple-Task Wait-and-Signal Synchronization

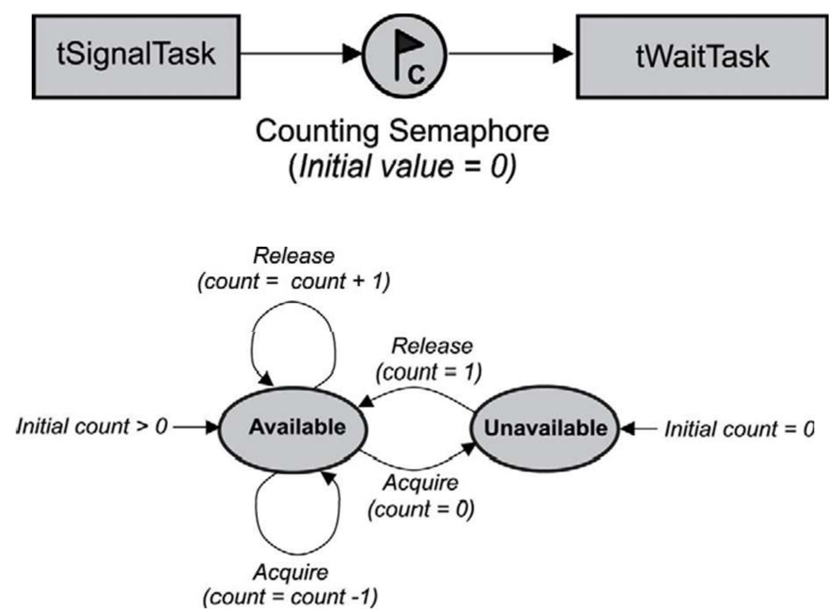
1. the binary semaphore is initially unavailable (value of 0).
2. The higher priority ***tWaitTasks 1***, ***2***, and ***3*** execute, acquire the unavailable semaphore, and blocked.
3. ***tSignalTask*** execute and flush the semaphore
4. one of the higher priority tWaitTasks preempts tSignalTask and starts to execute




```
tWaitTask () {
        Acquire binary semaphore token
                :
        }
tSignalTask () {
                :
        Flush binary semaphore's task-waiting list
                :
```

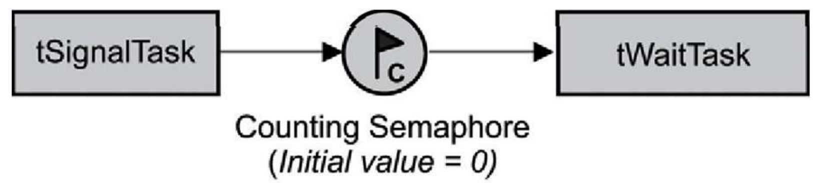
# Credit-Tracking Synchronization

1. counting semaphore's count is initially 0, making it unavailable
2. lower priority ***tWaitTask*** tries to ***acquire*** this semaphore but blocks until ***tSignalTask*** makes the semaphore available by performing a ***release***



# Credit-Tracking Synchronization

3. Even then, **tWaitTask** will waits in the ready state until the higher priority **tSignalTask** eventually relinquishes the CPU by making a blocking call or delaying itself,
4. Because **tSignalTask** is set to a higher priority and executes at its own rate, it might increment the counting semaphore multiple times before **tWaitTask** starts processing the first request.

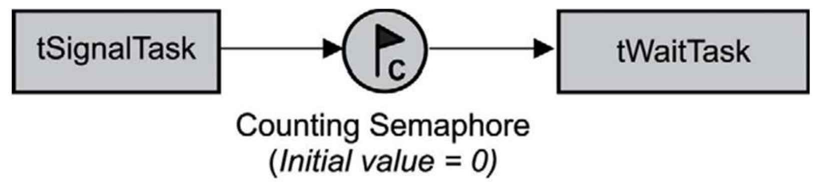



```
tWaitTask () {
        :
        Acquire counting semaphore token
        :
}

tSignalTask () {
        :
        Release counting semaphore token
        :
}
```

# Credit-Tracking Synchronization

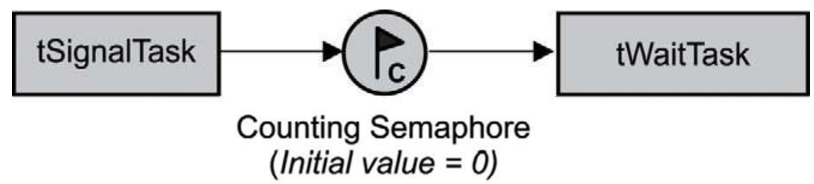
5. Hence, the counting semaphore allows a credit buildup of the number of times that the **tWaitTask** can execute before the semaphore becomes unavailable




tSignelTask is the source,
releasing token
tWaitTask is the sink,
Acquire (consume) the token

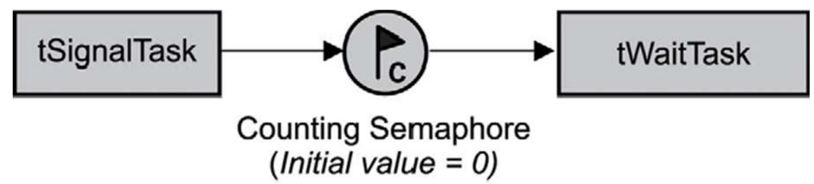
# Credit-Tracking Synchronization

6. when **tSignalTask's** rate of releasing the semaphore tokens slows, tWaitTask can catch up and eventually deplete the count until the counting semaphore is empty



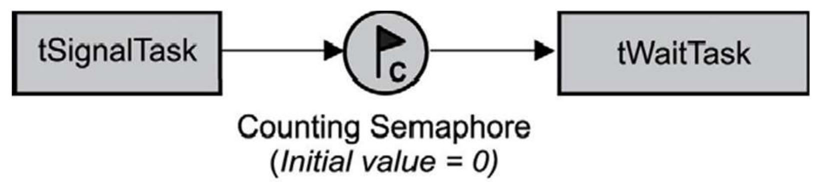
# Credit-Tracking Synchronization

7. At this point, **tWaitTask** blocks again at the counting semaphore, waiting for **tSignalTask** to release the semaphore again



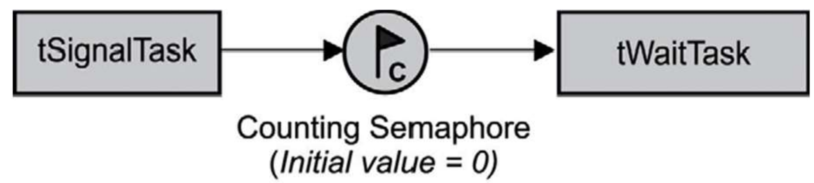
# Credit-Tracking Synchronization

8. useful if **tSignalTask** releases semaphores in ***bursts***, giving **tWaitTask** the chance to catch up every once in a while



# Credit-Tracking Synchronization

- ISR and DST
  - ISR that acts in a similar way to the signaling task
  - DSR/Wait task with lower priority offloads works for ISR

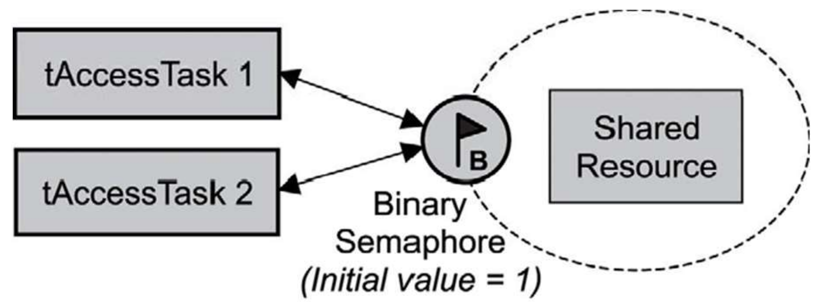


# Single Shared-Resource-Access Synchronization

- To provide for mutually exclusive access to a shared resource, that a
  - memory location,
  - a data structure, or an
  - I/O device
  - essentially anything that might have to be shared between two or more concurrent threads of execution.
- A semaphore can be used to serialize access to a shared resource,

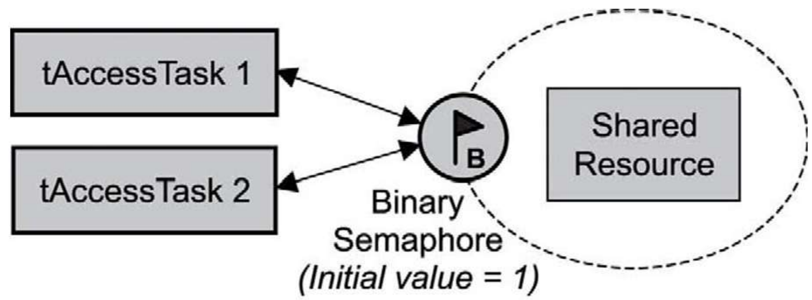
# Single Shared-Resource-Access Synchronization

1. a binary semaphore is initially created in the available state (value = 1) and is used to protect the shared resource



# Single Shared-Resource-Access Synchronization

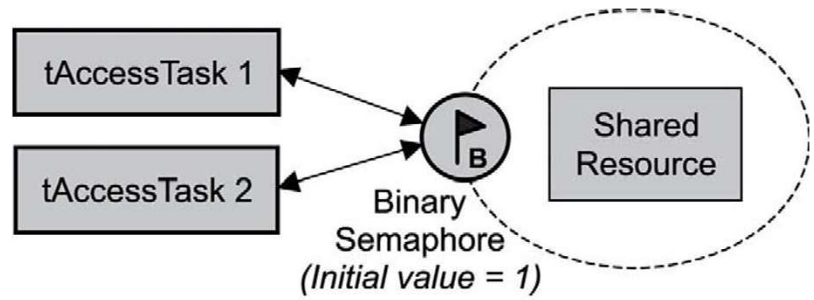
2. To access the shared resource, task 1 or 2 needs to first successfully acquire the binary semaphore before reading from or writing to the shared resource




```
tAccessTask () {
        :
        Acquire binary semaphore token
        Read or write to shared resource
        Release binary semaphore token
        :
} # task1 and task 2
```

# Single Shared-Resource-Access Synchronization

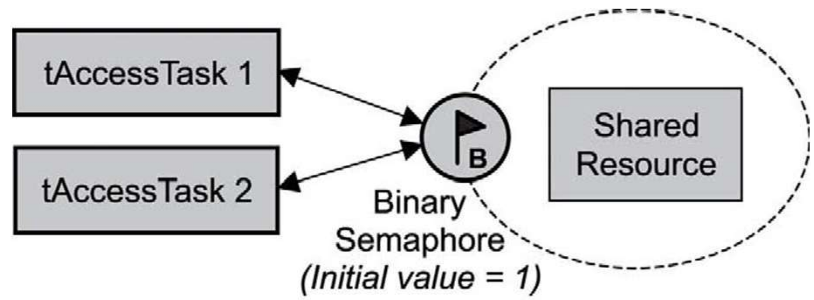
- **Pitfall**!!
- A third party task can accidentally release the binary semaphore, even one that never acquired the semaphore in the first place.
- If this is the case, both **tAccessTask 1** and **tAccessTask 2** could end up acquiring the semaphore and accessing the resource at the same time
- Using **mutex** to avoid




```
tAccessTask () {
        :
        Acquire binary semaphore token
        Read or write to shared resource
        Release binary semaphore token
        :
} # task1 and task 2
```

# Single Shared-Resource-Access Synchronization

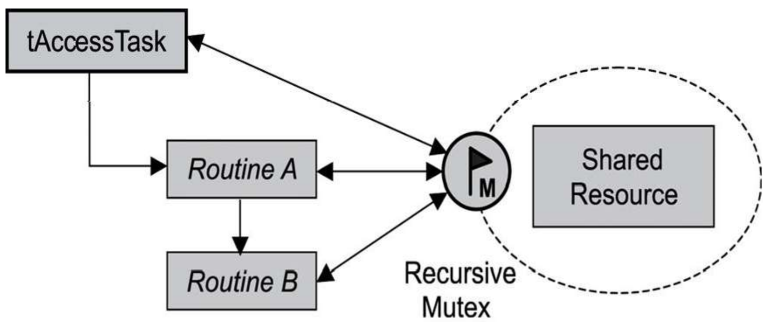
- **MUTEX**
- Mutex is a special semaphore that only the task acquired it can release it.




```
tAccessTask () {
        :
        Acquire binary semaphore token
        Read or write to shared resource
        Release binary semaphore token
        :
}  # task1 and task 2
```

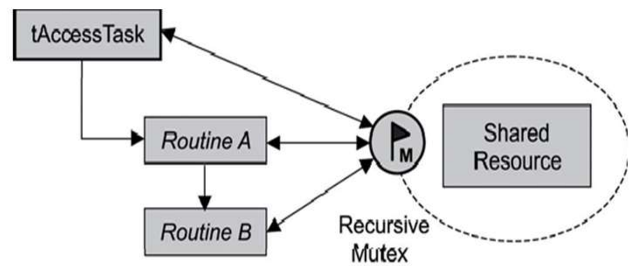
# Recursive Shared-Resource-Access Synchronization

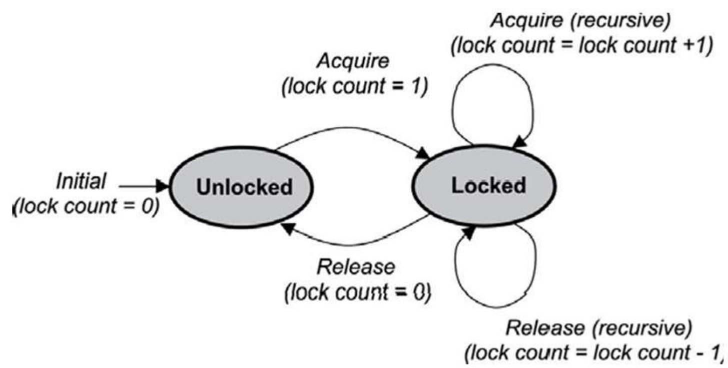
- Task to access a shared resource recursively.
  - For example, **tAccessTask** calls Routine A that calls Routine B, and all three need access to the same shared resource,
  - tasks would end up blocking, causing a deadlock



# Recursive Shared-Resource-Access Synchronization

```
tAccessTask () {
    :
    Acquire mutex
    Access shared resource
    Call Routine A
    Release mutex
    :
    }
Routine A () {
    :
    Acquire mutex
    Access shared resource
    Call Routine B
    Release mutex
    :
    }
Routine B () {
    :
    Acquire mutex
    Access shared resource
    Release mutex
    :
     }
```

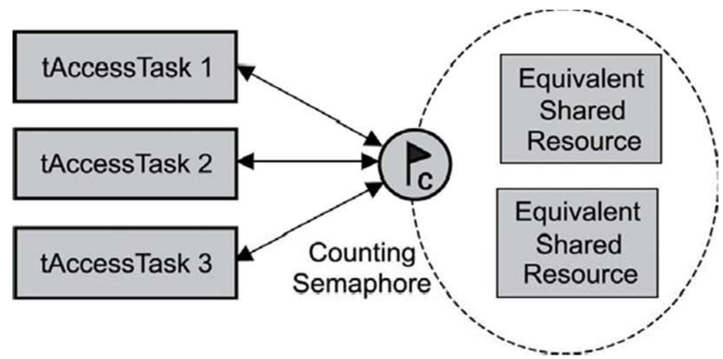



***recursive mutex***. After tAccessTask locks the mutex, the task owns it. Additional attempts from the task itself or from routines that it calls to lock the mutex succeed.

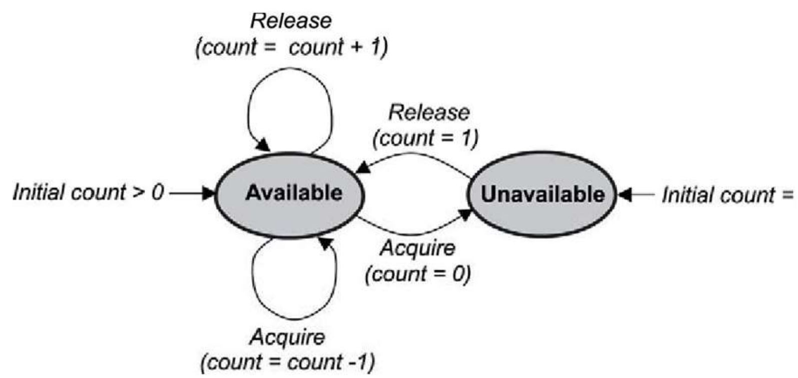
# Multiple Shared-Resource-Access Synchronization

- When multiple equivalent shared resources are used, a counting semaphore comes in handy
- The counting semaphore's count is initially set to the number of equivalent shared resources

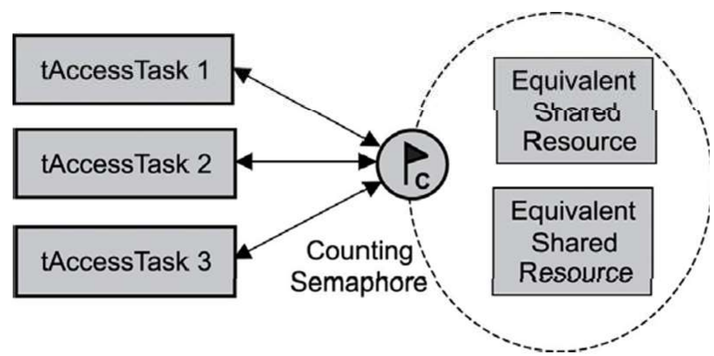


# Multiple Shared-Resource-Access Synchronization

```
tAccessTask () {
    :
    Acquire a counting semaphore token
    Read or Write to shared resource
    Release a counting semaphore token
    :
}
```




similar code is used for
  tAccessTask 1, 2, and 3



# Multiple Shared-Resource-Access Synchronization

```
tAccessTask () {
    :
    Acquire first mutex in non-blocking way
            If not successful then acquire 2nd mutex in a blocking way
    Read or Write to shared resource
    Release the acquired mutex
    :
}
```

- similar code is used for tAccessTask 1, 2, and 3
- Pseudo code for multiple tasks accessing equivalent shared resources using mutexes

# Đoạn mã giả định cho việc nhiều tác vụ (tAccessTask1, tAccessTask2, tAccessTask3, vv.)

```
tAccessTask1():
    while True:
        # Attempt to acquire the first mutex in a non-blocking way
        if mutex1.try_acquire():
            # First mutex acquired successfully
            break  # Exit the loop
        else:
            # First mutex not acquired, try to acquire the second mutex in a blocking way
            mutex2.acquire()

    # Access the shared resource (read or write)
    # [Code to read from or write to the shared resource]

    # Release the acquired mutex
    mutex1.release()  # Release the first mutex
    mutex2.release()  # Release the second mutex
```

# Đoạn mã giả định cho việc nhiều tác vụ (tAccessTask1, tAccessTask2, tAccessTask3, vv.)_

```
tAccessTask2():
    while True:
        # Attempt to acquire the first mutex in a non-blocking way
        if mutex1.try_acquire():
            # First mutex acquired successfully
            break  # Exit the loop
        else:
            # First mutex not acquired, try to acquire the second mutex in a blocking
way
            mutex2.acquire()

    # Access the shared resource (read or write)
    # [Code to read from or write to the shared resource]

    # Release the acquired mutex
    mutex1.release()  # Release the first mutex
    mutex2.release()  # Release the second mutex
```

# Đoạn mã giả định cho việc nhiều tác vụ (tAccessTask1, tAccessTask2, tAccessTask3, vv.)

```
tAccessTask3():
    while True:
        # Attempt to acquire the first mutex in a non-blocking way
        if mutex1.try_acquire():
            # First mutex acquired successfully
            break  # Exit the loop
        else:
            # First mutex not acquired, try to acquire the second mutex in a blocking way
            mutex2.acquire()

    # Access the shared resource (read or write)
    # [Code to read from or write to the shared resource]

    # Release the acquired mutex
    mutex1.release()  # Release the first mutex
    mutex2.release()  # Release the second mutex
```