مدينة زويل للعلوم والتكنولوجيا
Zewail City of Science and Technology

COMMUNICATION AND INFORMATION ENGINEERING

CIE 314
Embedded Systems Fundamentals

Lecture #10
RTOS Events: Semaphores

Instructor:
Dr. Ahmad El-Banna

Agenda

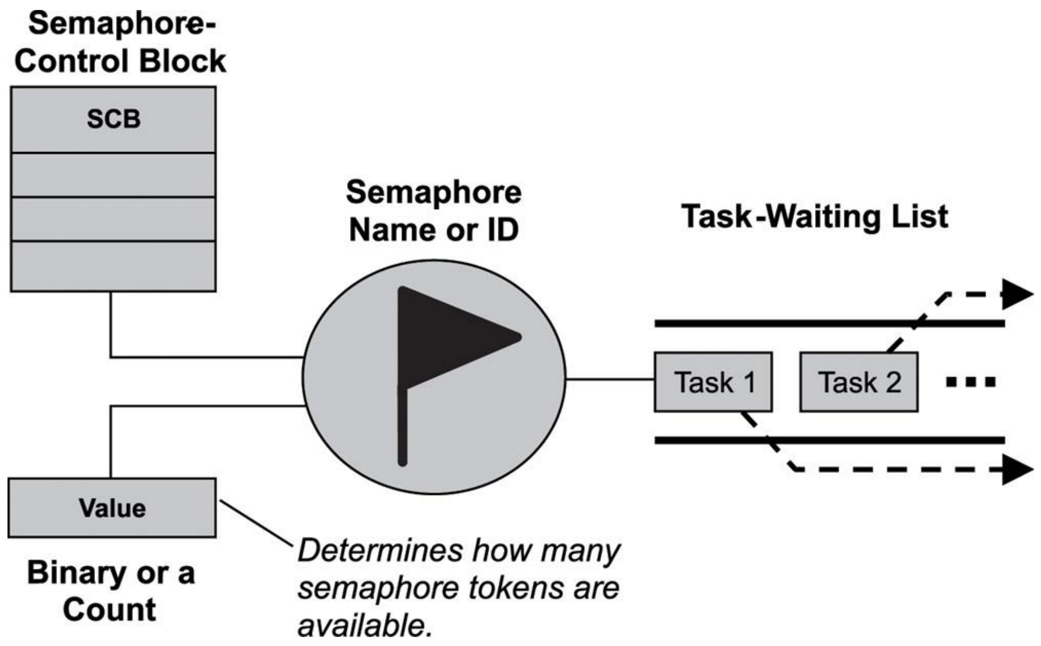
- Defining a semaphore
- Typical semaphore operations
- Common semaphore use
- Design Tips

Intro.

- Multiple concurrent threads of execution within an application must be able to synchronize their execution and coordinate mutually exclusive access to shared resources.
- To address these requirements, RTOS kernels provide a semaphore object and associated semaphore management services.

Defining Semaphores

- 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.
- 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..

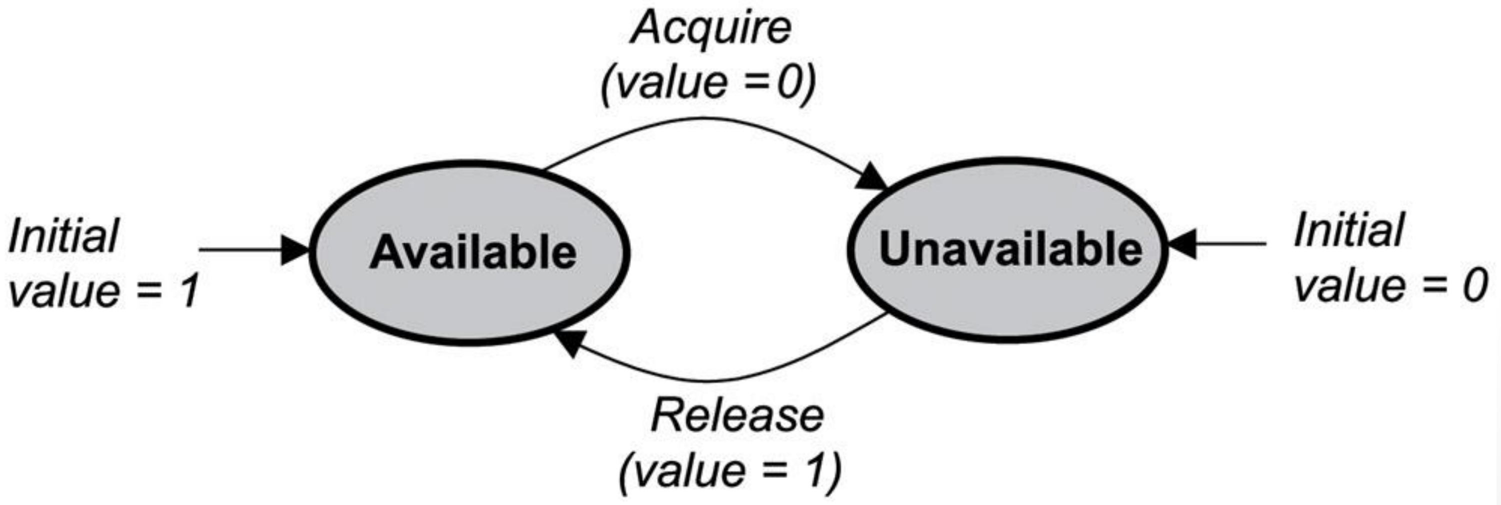
- A semaphore is like a key that allows a task to carry out some operation or to access a resource.
- The kernel tracks the number of times a semaphore has been acquired or released by maintaining a token count, which is initialized to a value when the semaphore is created.
- As a task acquires the semaphore, the token count is decremented; as a task releases the semaphore, the count is incremented.
- If the token count reaches 0, the semaphore has no tokens left.
- A requesting task, therefore, cannot acquire the semaphore, and the task blocks if it chooses to wait for the semaphore to become available.
- The task-waiting list tracks all tasks blocked while waiting on an unavailable semaphore.
- These blocked tasks are kept in the task-waiting list in either first in/first out (FIFO) order or highest priority first order.
- When an unavailable semaphore becomes available, the kernel allows the first task in the task-waiting list to acquire it.

Types of Semaphores

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

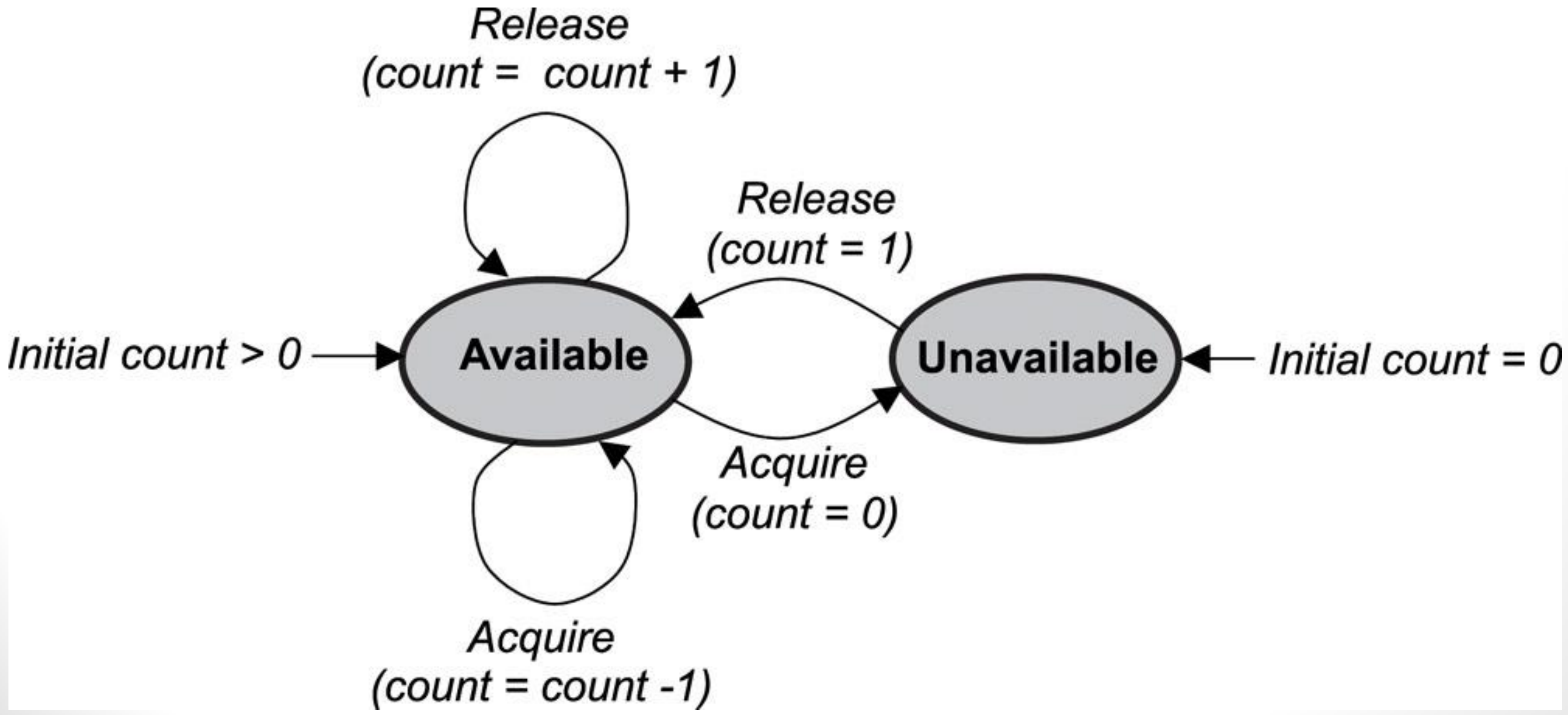
Binary Semaphores

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



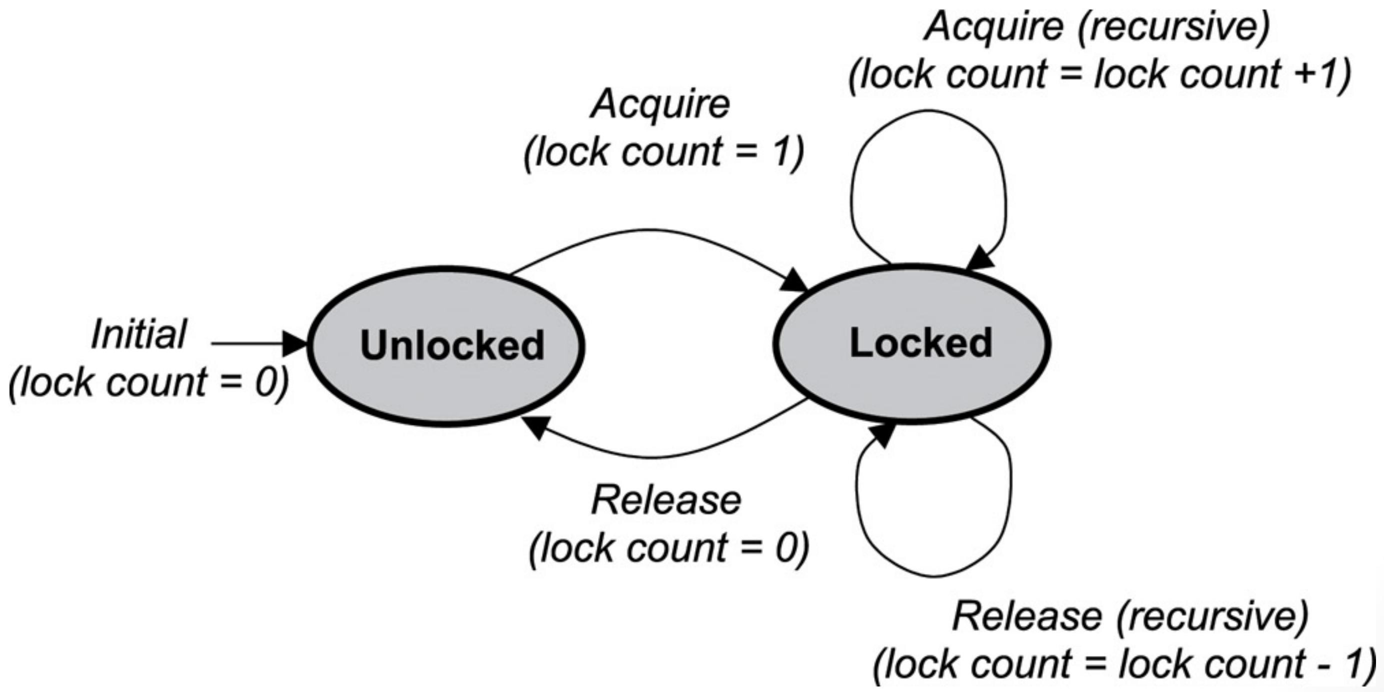
Counting Semaphores

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



Mutex Semaphores

- As opposed to the available and unavailable states in binary and counting semaphores, the states of a mutex are *unlocked* or *locked* (0 or 1, respectively).
- 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.
- Conversely, when the task releases the mutex, the mutex returns to the unlocked state. Some kernels might use the terms *lock* and *unlock* for a mutex instead of *acquire* and *release*.
- Depending on the implementation, a mutex can support additional features not found in binary or counting semaphores.
- These key differentiating features include ownership, recursive locking, task deletion safety, and priority inversion avoidance protocols.

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.
- When a task owns the mutex, it is not possible for any other task to lock or unlock that mutex.
- Contrast this concept with the binary semaphore, which can be released by any task, even a task that did not originally acquire the semaphore.

Recursive Locking

- Many mutex implementations also support *recursive locking*, which allows the task that owns the mutex to acquire it multiple times in the locked state.
- Depending on the implementation, recursion within a mutex can be automatically built into the mutex, or it might need to be enabled explicitly when the mutex is first created.
- The mutex with recursive locking is called a *recursive mutex* .
- This type of mutex is most useful when a task requiring exclusive access to a shared resource calls one or more routines that also require access to the same resource.
- A recursive mutex allows nested attempts to lock the mutex to succeed, rather than cause *deadlock* , which is a condition in which two or more tasks are blocked and are waiting on mutually locked resources.

Task Deletion Safety

- Some mutex implementations also have built-in *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.
- Typically protection from premature deletion is enabled by setting the appropriate initialization options when creating the mutex.

Priority Inversion Avoidance

- Priority inversion commonly happens in poorly designed real-time embedded applications.
- 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.

- Enabling certain protocols that are typically built into mutexes can help avoid priority inversion.

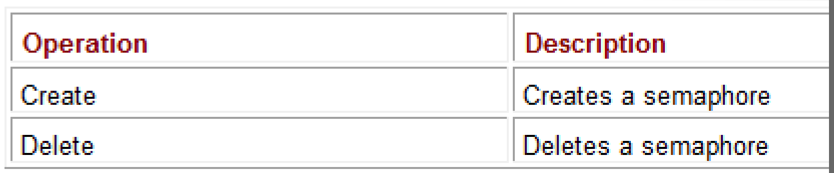
Priority Inversion Avoidance..

- Two common protocols used for avoiding priority inversion include:
- **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.
- **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.
- When the mutex is released, the priority of the task is lowered to its original value.
- Chapter 16 discusses priority inversion and both the priority inheritance and ceiling priority protocols in more detail.

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
  - getting semaphore information.

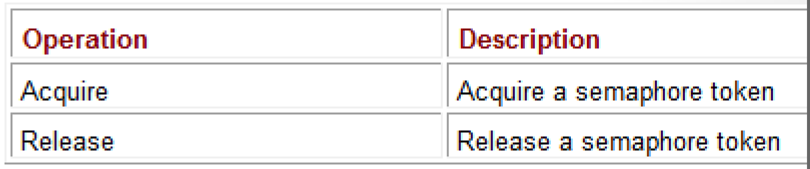
Creating and Deleting Semaphores



Operation	Description
Create	Creates a semaphore
Delete	Deletes a semaphore

- Several things must be considered when creating and deleting semaphores.
- If a kernel supports different types of 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.
  - **Mutex**: specify the task-waiting order and enable task deletion safety, recursion, and priority-inversion avoidance protocols, if supported.

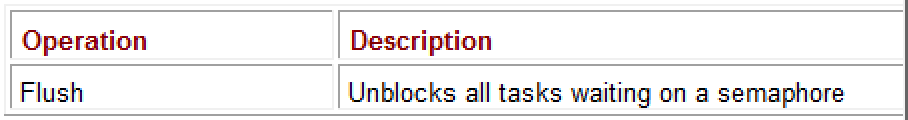
Acquiring and Releasing Semaphores



Operation	Description
Acquire	Acquire a semaphore token
Release	Release a semaphore token

- The operations for acquiring and releasing a semaphore might have different names, depending on the kernel: for example, *take* and *give* , *sm_p* and *sm_v* , *pend* and *post* , and *lock* and *unlock* . Regardless of the name, they all effectively acquire and release 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. At this point, the task is removed from the semaphore's task-waiting list and put in either the ready state or the running state.
    - **Do not wait**—task makes a request to acquire a semaphore token, but, if one is not available, the task does not block.

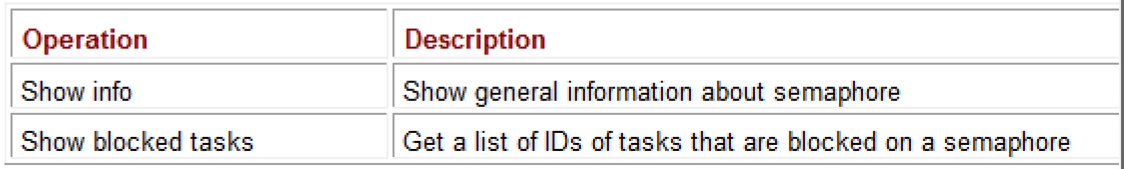
Clearing Semaphore Task-Waiting Lists



Operation	Description
Flush	Unblocks all tasks waiting on a semaphore

- To clear all tasks waiting on a semaphore task-waiting list, some kernels support a *flush* operation.
- The flush operation is useful for broadcast signaling to a group of tasks.
- For example, a developer might design 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.
- The synchronization scenario just described is also called *thread rendezvous*, when multiple tasks' executions need to meet at some point in time to synchronize execution control.

Getting Semaphore Information



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

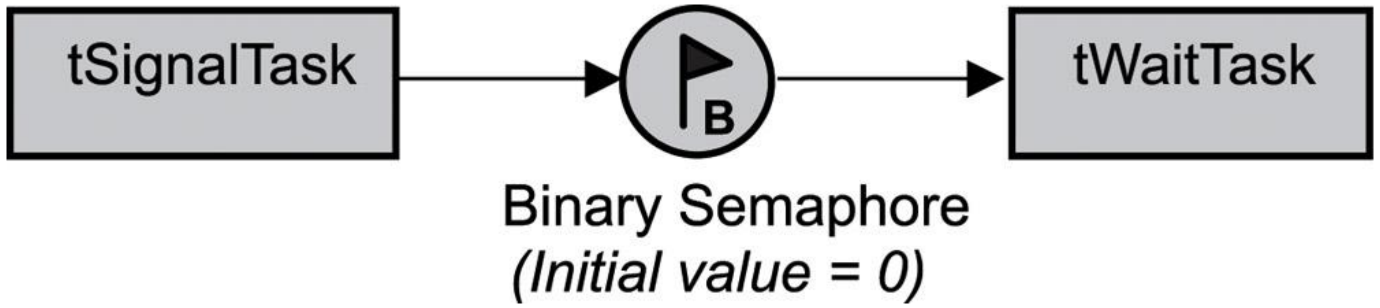
- At some point in the application design, developers need to obtain semaphore information to perform monitoring or debugging.
- These operations are relatively straightforward but should be used judiciously, as the semaphore information might be dynamic at the time it is requested.

Typical Semaphore Use

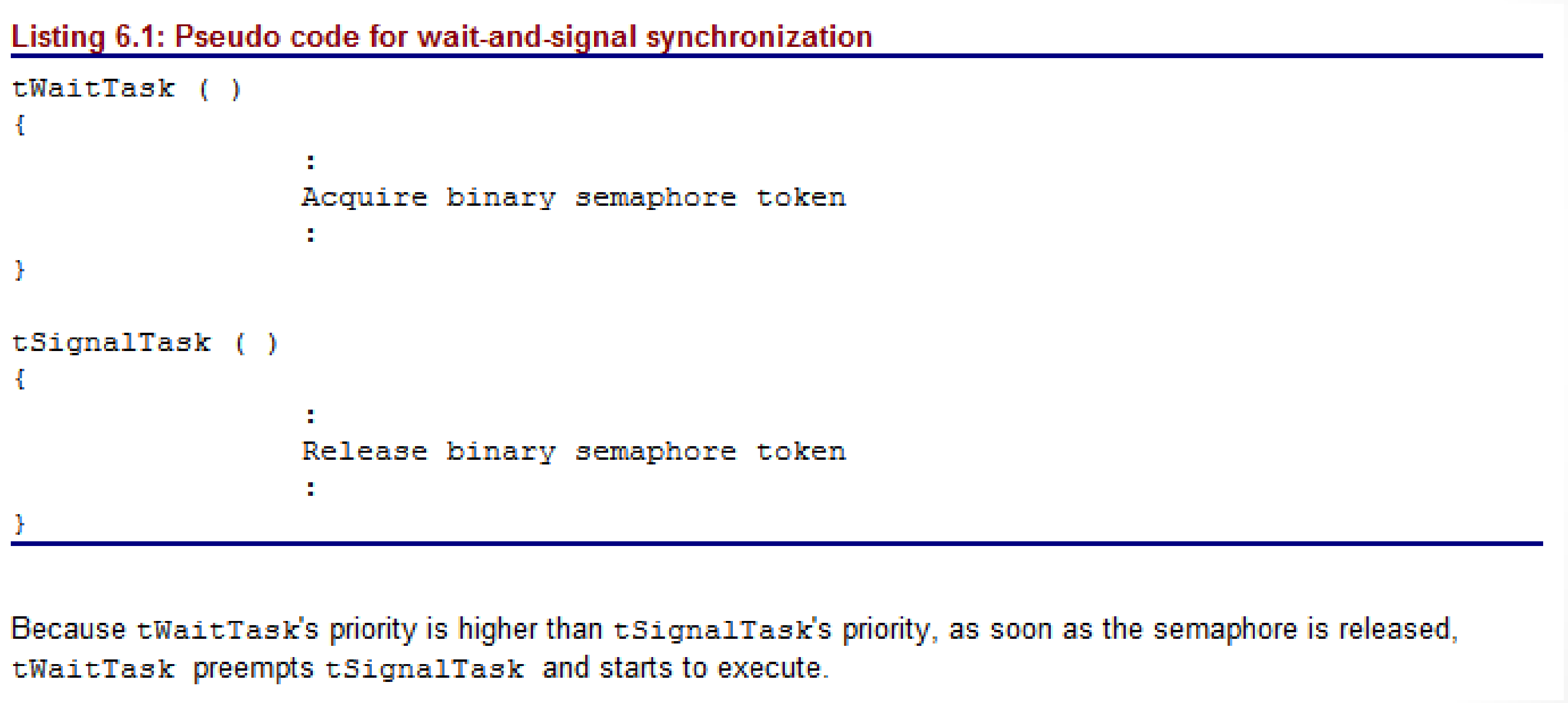
- Semaphores are useful either for synchronizing execution of multiple tasks or for coordinating access to a shared resource.
- The following examples illustrate using different types of semaphores to address common synchronization design requirements effectively, as listed:
  - wait-and-signal synchronization,
  - 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.

Wait-and-Signal Synchronization

- Two tasks can communicate for the purpose of synchronization without exchanging data.
- For example, a binary semaphore can be used between two tasks to coordinate the transfer of execution control.



Wait-and-Signal Synchronization ..

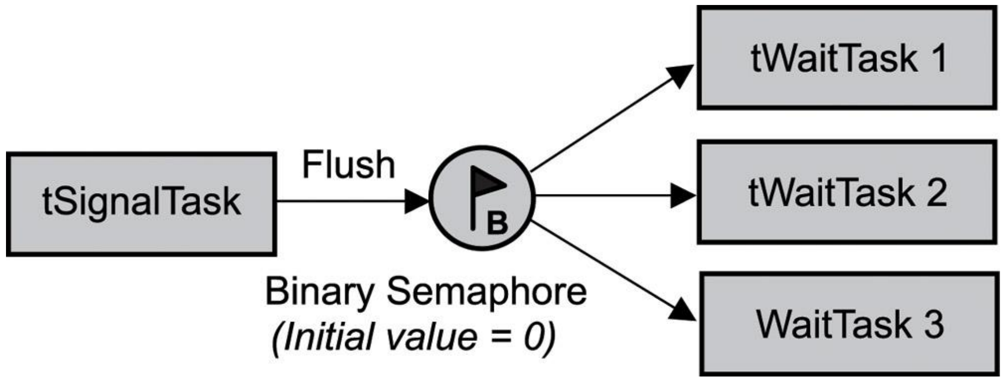


Listing 6.1: Pseudo code for wait-and-signal synchronization

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

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

Because `tWaitTask`'s priority is higher than `tSignalTask`'s priority, as soon as the semaphore is released, `tWaitTask` preempts `tSignalTask` and starts to execute.

Multiple-Task Wait-and-Signal Synchronization



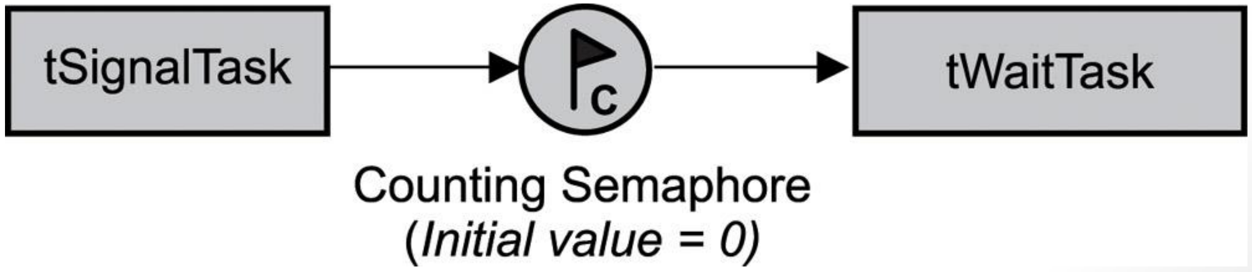
Listing 6.2: Pseudo code for wait-and-signal synchronization.

```
tWaitTask ()
{
                :
                Do some processing specific to task Acquire binary semaphore token
                :
}

tSignalTask ()
{
                :
                Do some processing Flush binary semaphore's task-waiting list
                :
}
```

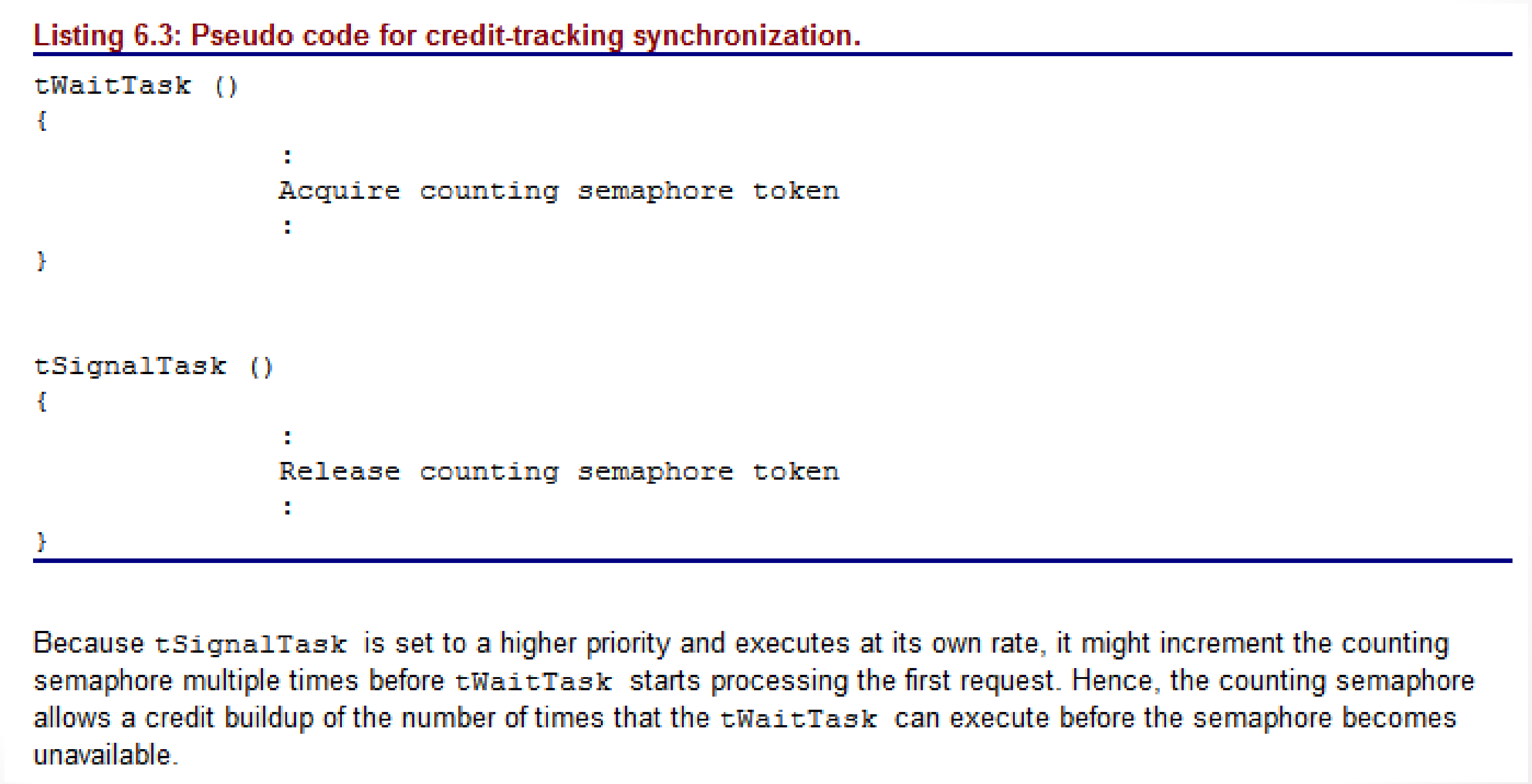
Because the `tWaitTasks'` priorities are higher than `tSignalTask`'s priority, as soon as the semaphore is released, one of the higher priority `tWaitTasks` preempts `tSignalTask` and starts to execute.

Credit-Tracking Synchronization

- Sometimes the rate at which the signaling task executes is higher than that of the signaled task.
- In this case, a mechanism is needed to count each signaling occurrence.
- The counting semaphore provides just this facility.
- With a counting semaphore, the signaling task can continue to execute and increment a count at its own pace, while the wait task, when unblocked, executes at its own pace.



Credit-Tracking Synchronization..



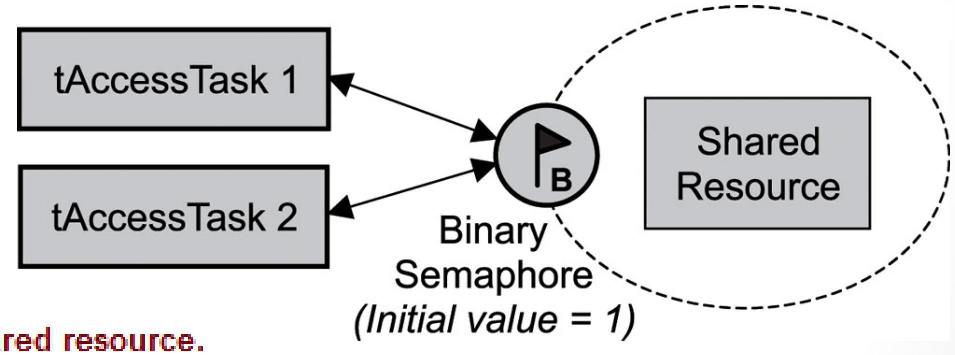
Listing 6.3: Pseudo code for credit-tracking synchronization.

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


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

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

Single Shared-Resource-Access Synchronization

- One of the more common uses of semaphores is to provide for mutually exclusive access to a shared resource.
- A shared resource might be 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

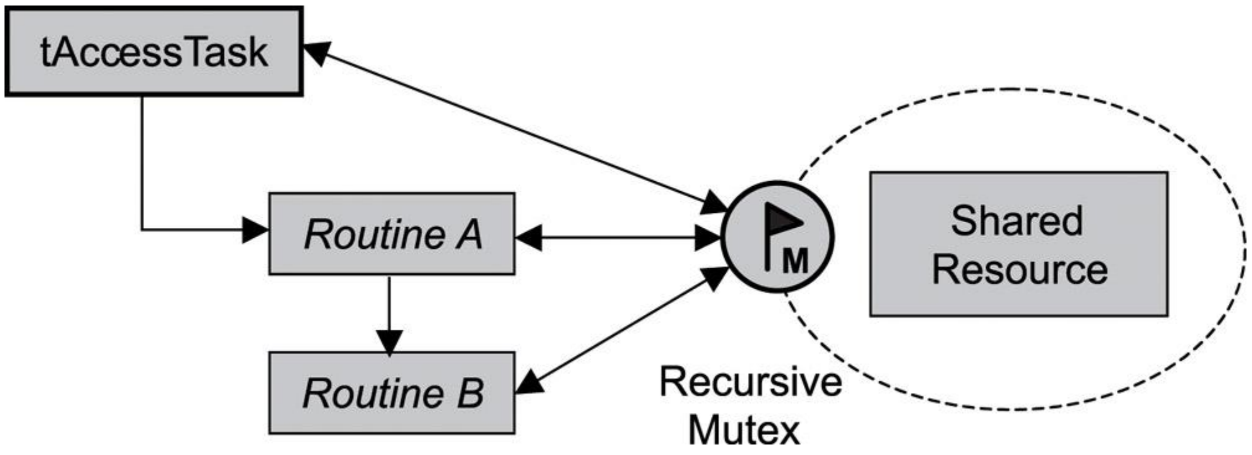


Listing 6.4: Pseudo code for tasks accessing a shared resource.

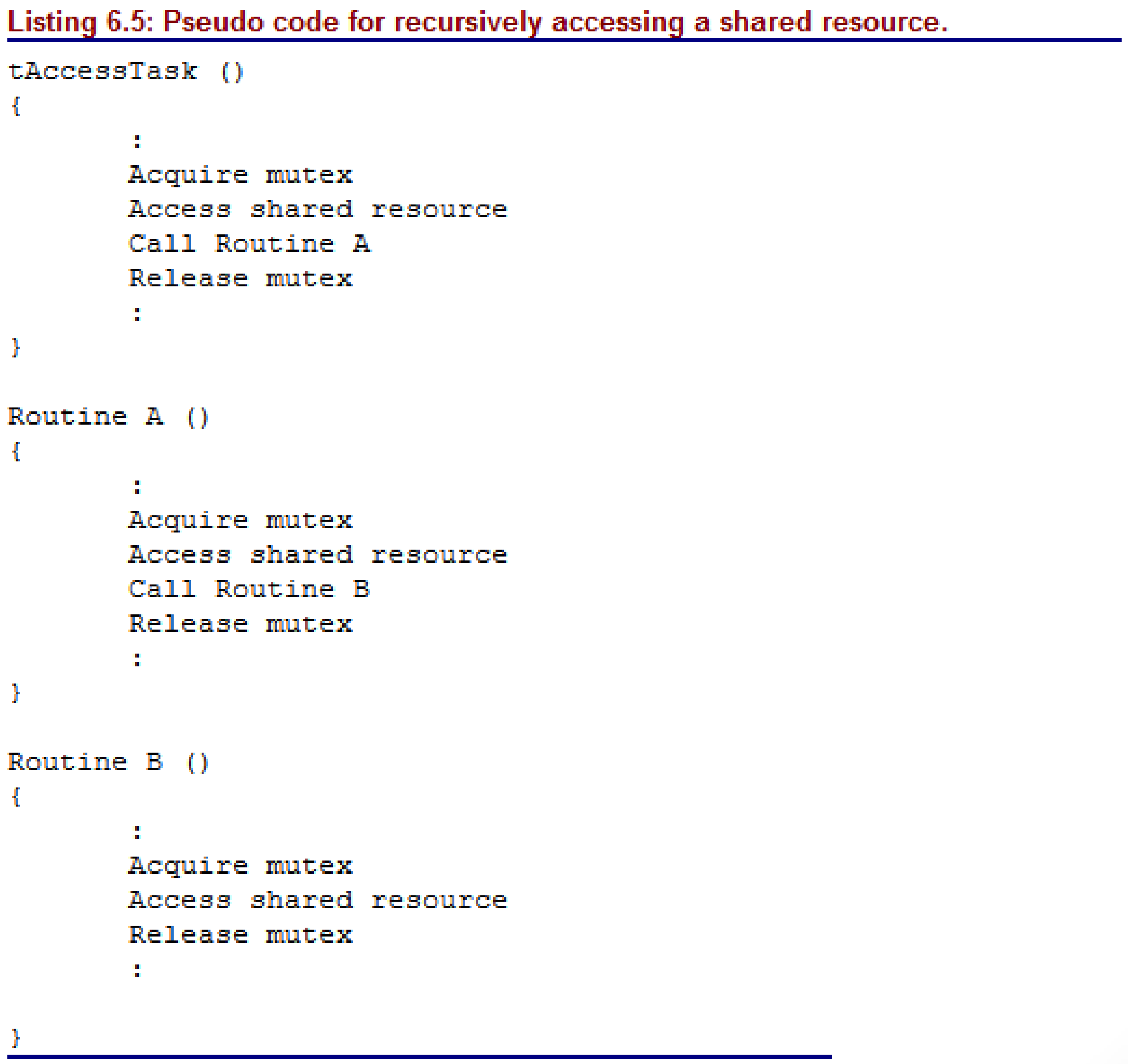
```
tAccessTask ()
{
              :
              Acquire binary semaphore token
              Read or write to shared resource
              Release binary semaphore token
              :
}
```

Recursive Shared-Resource-Access Synchronization

- Sometimes a developer might want a task to access a shared resource recursively.
- This situation might exist if tAccessTask calls Routine A that calls Routine B, and all three need access to the same shared resource.



Recursive Shared-Resource-Access Synchronization ..

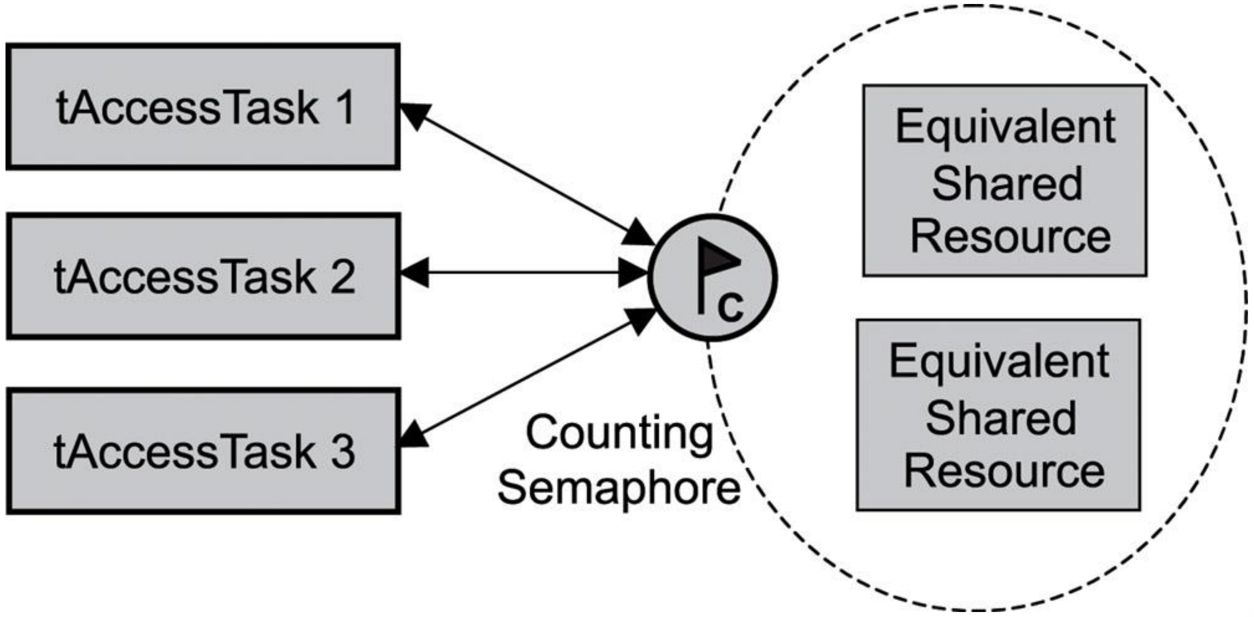


Listing 6.5: Pseudo code for recursively accessing a shared resource.

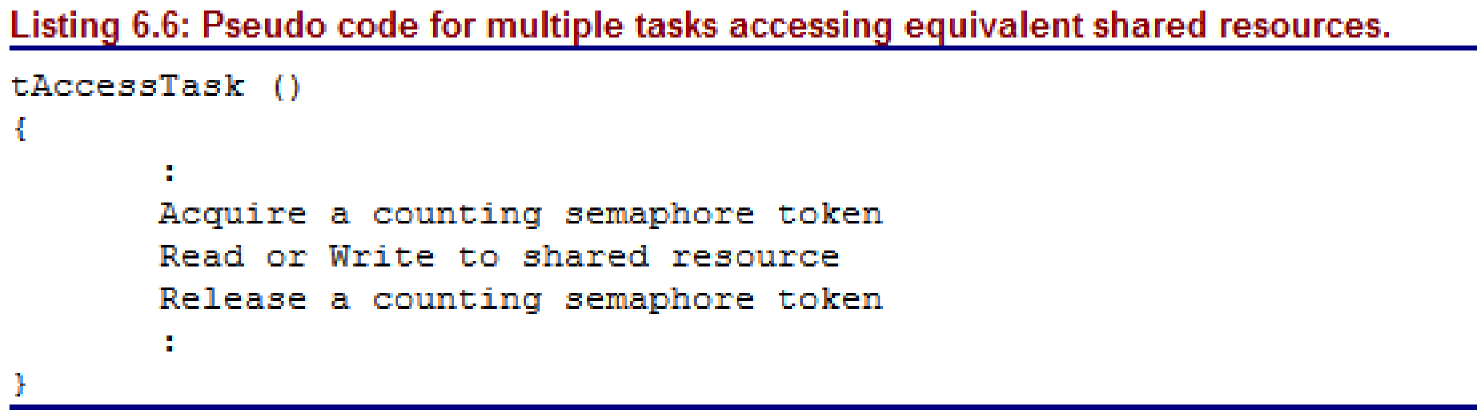
```
tAccessTask ()
{
        :
        Acquire mutex
        Access shared resource
        Call Routine A
        Release mutex
        :
}

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

Routine B ()
{
        :
        Acquire mutex
        Access shared resource
        Release mutex
        :

}
```

Multiple Shared-Resource-Access Synchronization

- For cases in which multiple equivalent shared resources are used, a counting semaphore comes in handy.

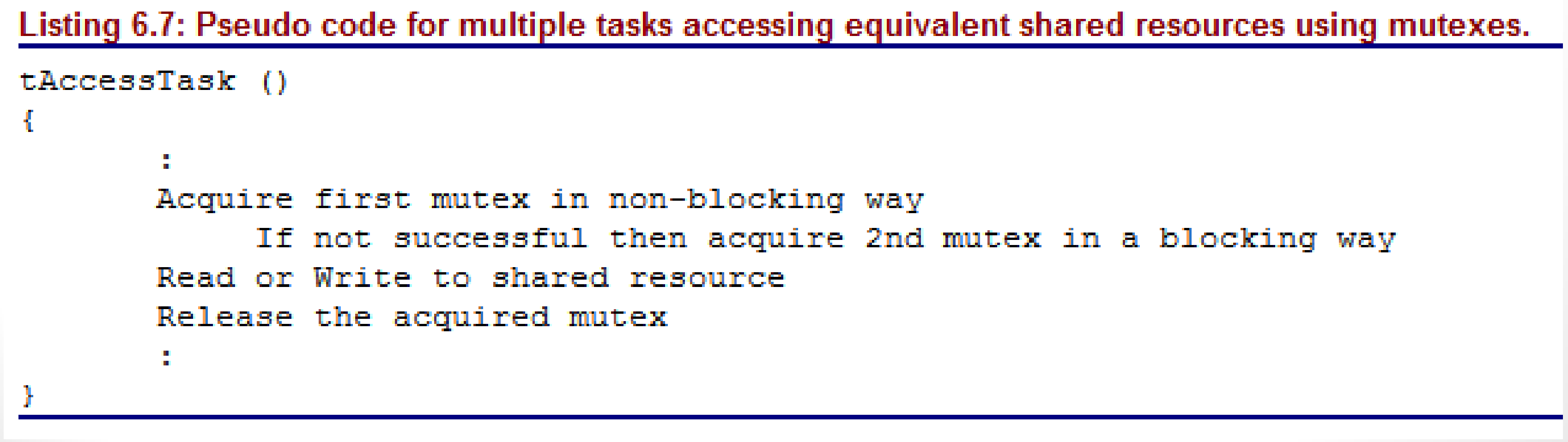


Multiple Shared-Resource-Access Synchronization..



Listing 6.6: Pseudo code for multiple tasks accessing equivalent shared resources.

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



Listing 6.7: Pseudo code for multiple tasks accessing equivalent shared resources using mutexes.

```
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
        :
}
```

Points to Remember

Some points to remember include the following:

- Using semaphores allows multiple tasks, or ISRs to tasks, to synchronize execution to synchronize execution or coordinate mutually exclusive access to a shared resource.
- Semaphores have an associated semaphore control block (SCB), a unique ID, a user-assigned value (binary or a count), and a task-waiting list.
- Three common types of semaphores are binary, counting, and mutual exclusion (mutex), each of which can be acquired or released.
- Binary semaphores are either available (1) or unavailable (0). Counting semaphores are also either available (count =1) or unavailable (0). Mutexes, however, are either unlocked (0) or locked (lock count =1).

Points to Remember..

- Acquiring a binary or counting semaphore results in decrementing its value or count, except when the semaphore's value is already 0. In this case, the requesting task blocks if it chooses to wait for the semaphore.
- Releasing a binary or counting semaphore results in incrementing the value or count, unless it is a binary semaphore with a value of 1 or a bounded semaphore at its maximum count. In this case, the release of additional semaphores is typically ignored.
- Recursive mutexes can be locked and unlocked multiple times by the task that owns them. Acquiring an unlocked recursive mutex increments its lock count, while releasing it decrements the lock count.
- Typical semaphore operations that kernels provide for application development include creating and deleting semaphores, acquiring and releasing semaphores, flushing semaphore's task-waiting list, and providing dynamic access to semaphore information.

DESIGN TIPS

Digital Watch Design

Assignment#5

- Design, implement and test a digital Thermestor, the watch should have the functions of:
    - Display the current temperature.

- For more details, refer to:
  - Chapter 6 at **Real-time concepts for embedded systems**, CMP Books, 2003 by Qing Li and Carolyn Yao (ISBN:1578201241).
  - Chapter 5 at **Embedded Software Development with C**, Springer 2009 by Kai Qian et al.
  - Chapter 8,9,10 at **Introduction to Embedded Systems,** Springer 2014 by Manuel Jiménez et al.
- The lecture is available online at:
  - *http://bu.edu.eg/staff/ahmad.elbanna-courses*
- For inquires, send to:
  - ahmad.elbanna@feng.bu.edu.eg