Semaphores

Semaphore (Token)

- A kernel object
- One or more threads of execution can acquire or release for the purpose of synchronization or mutual exclusion

Creation of Semaphore

- Semaphore control block (SCB)
- Unique ID
- Value (binary or count)
- Task-waiting list

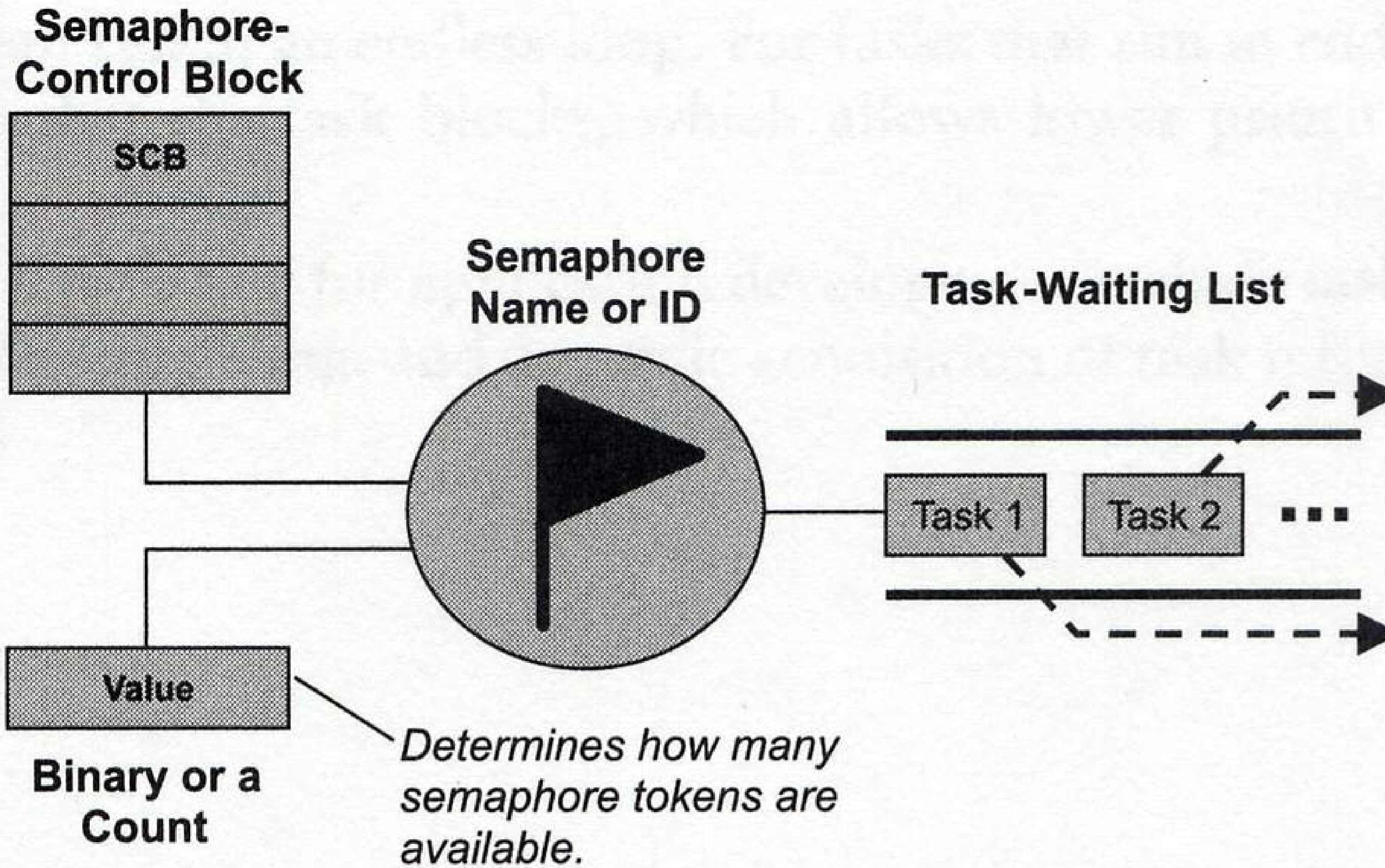


Figure 6.1 A semaphore, its associated parameters, and supporting data structures.

Semaphore

- Semaphore is like a key that allows a task to carry out some operation or to access a resource. (e.g. a key or keys to the lab)

Semaphore Count

- Semaphore (Token) count is initialized when created
- A task acquire the semaphore: count is decremented
- A task releases the semaphore: count is incremented

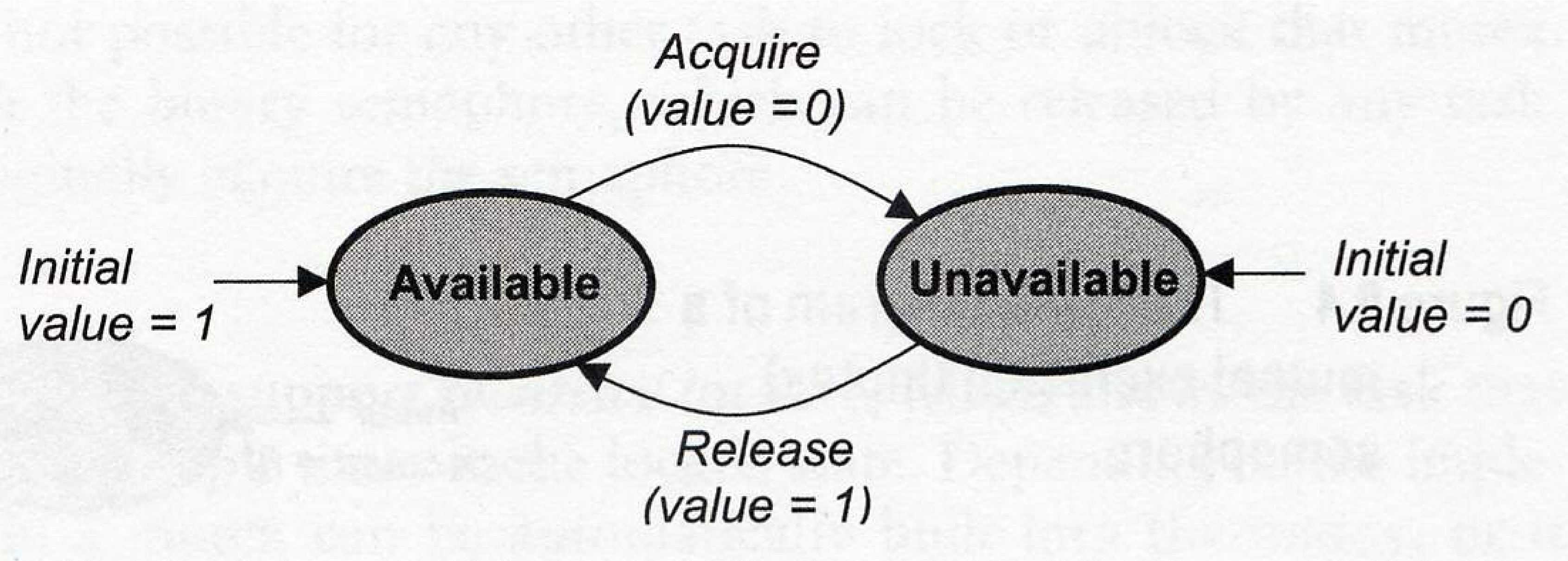
- Token count = 0 : a requesting task blocks

Task Waiting List

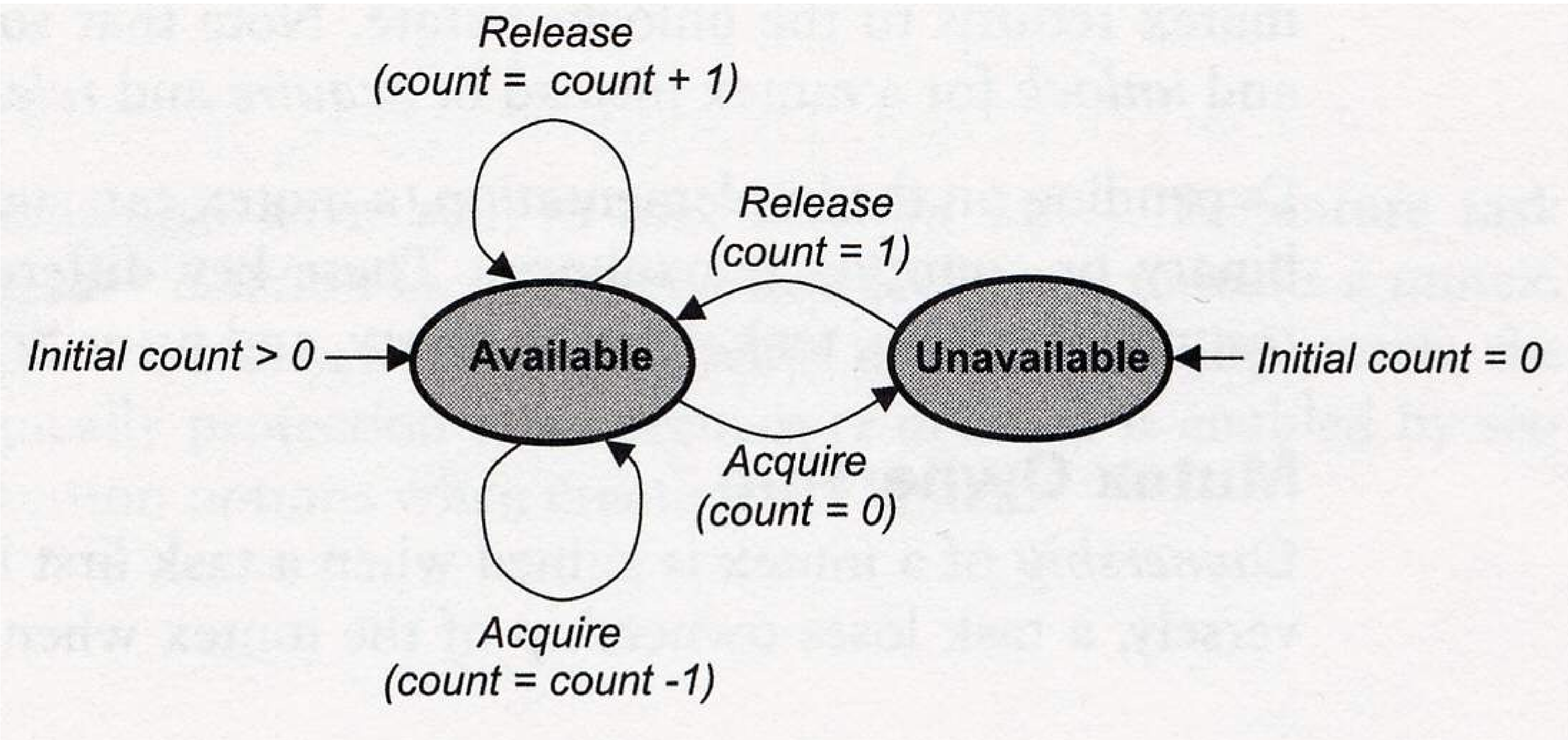
- FIFO or priority
- When an unavailable semaphore becomes available : first task in the list to acquire, blocked task-> running state (highest priority) or ready state

Binary Semaphore

- Value: 0 unavailable/empty
- Value: 1 available/full

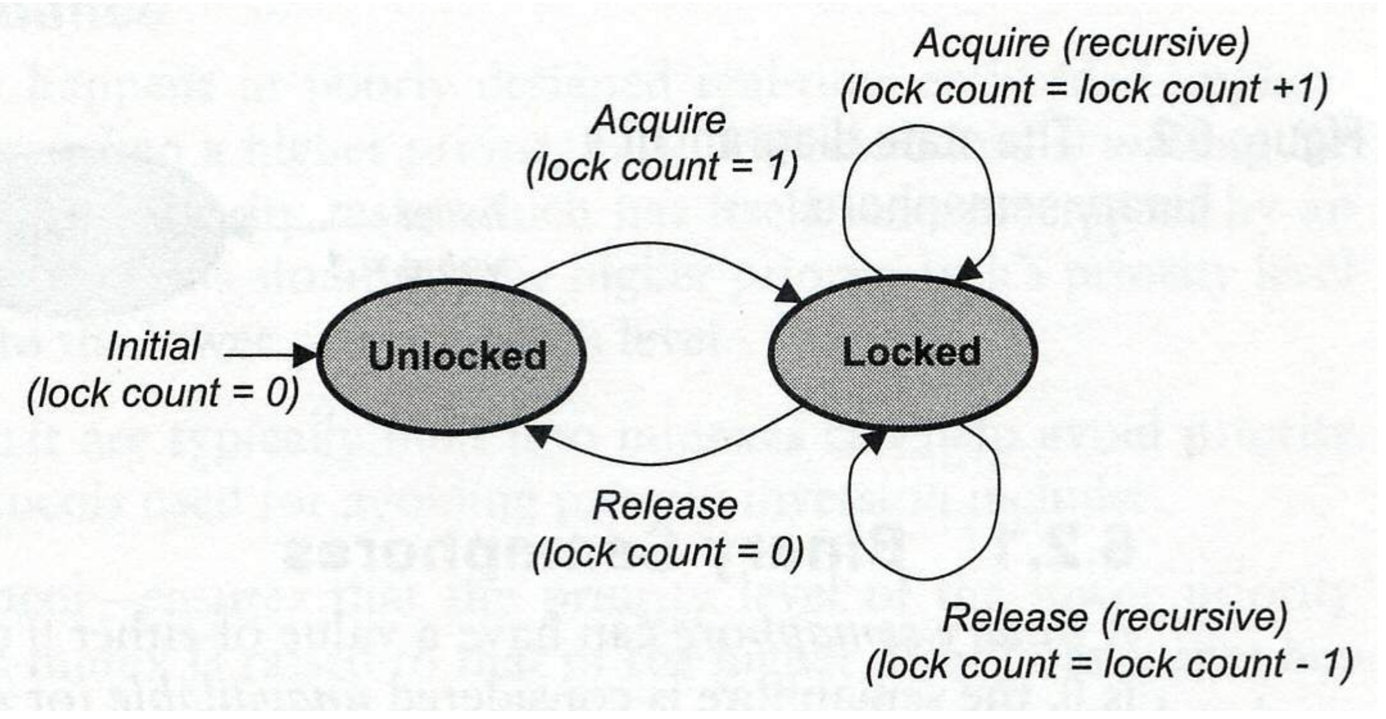


Counting Semaphore



Mutual Exclusion (Mutex) Semaphore

- A special binary semaphore that supports ownership, recursive access, task deletion safety, priority inversion avoidance protocol.



Mutex Ownership

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

- Recursive locking: when a task requiring exclusive access to a shared resource calls one or more routines that also require access to the same resource.

Mutex

- Task Deletion Safety: While a task owns a mutex, the task cannot be deleted

- Priority inversion avoidance

Typical Semaphore Operations

- Create
- Delete
- Acquire : wait forever, wait with a timeout, do not wait
- Release
- Flush: unlocks all tasks waiting on a semaphore

Typical Semaphore Use

- Wait-and-Signal Synchronization

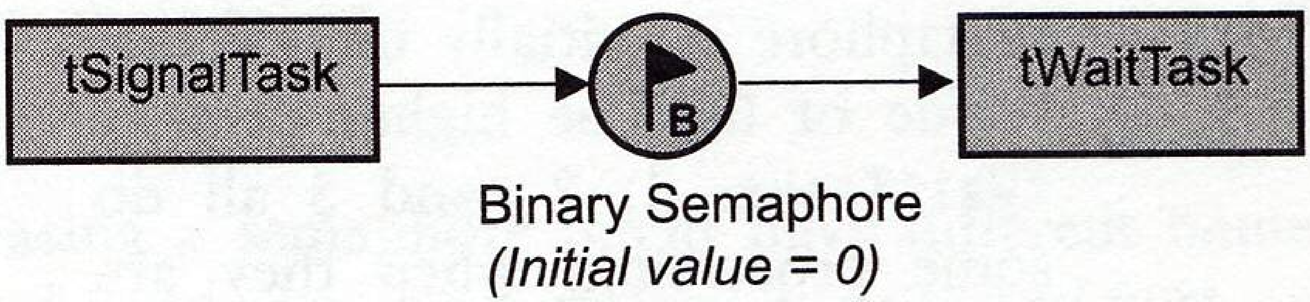


Figure 6.5 Wait-and-signal synchronization between two tasks.

Wait-and-Signal Synchronization

- tWaitTask runs first
- tWaitTask makes a request to acquire the semaphore but blocked
- tSignalTask has a chance to run
- tSignalTask releases the semaphore
- tWaitTask unblocked and running

Wait-and-Signal Synchronization

```
tWaitTask()
{
    …
    Acquire binary semaphore
    …
}
tSignalTask()
{
    …
    Release binary semaphore
    …
}
```

Multiple-Task Wait_and_Signal Synchronization

- tSignalTask: lower priority

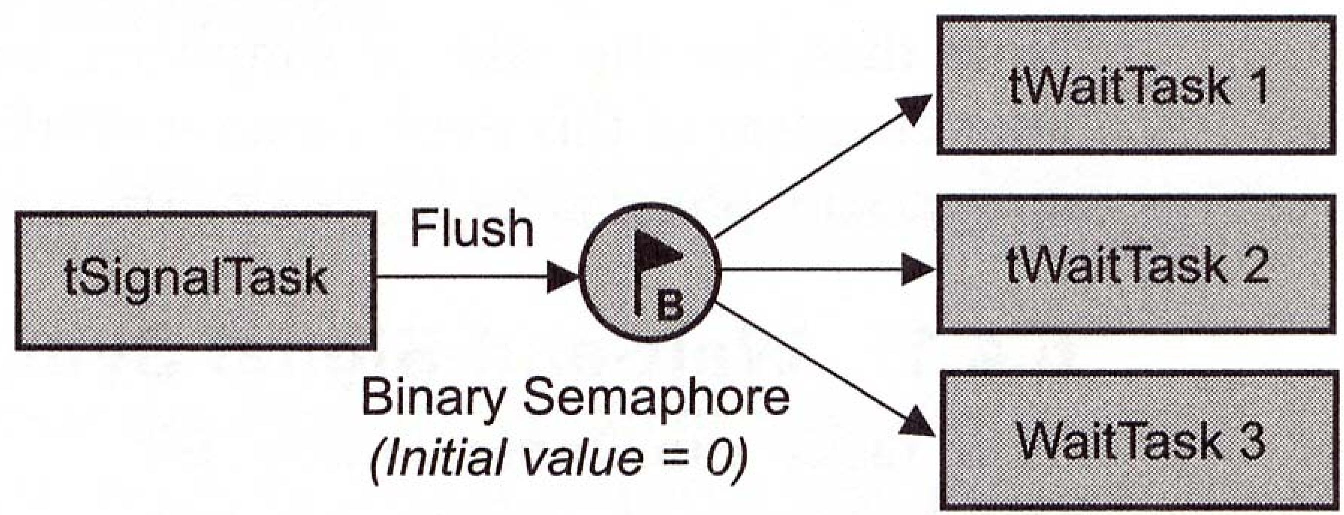


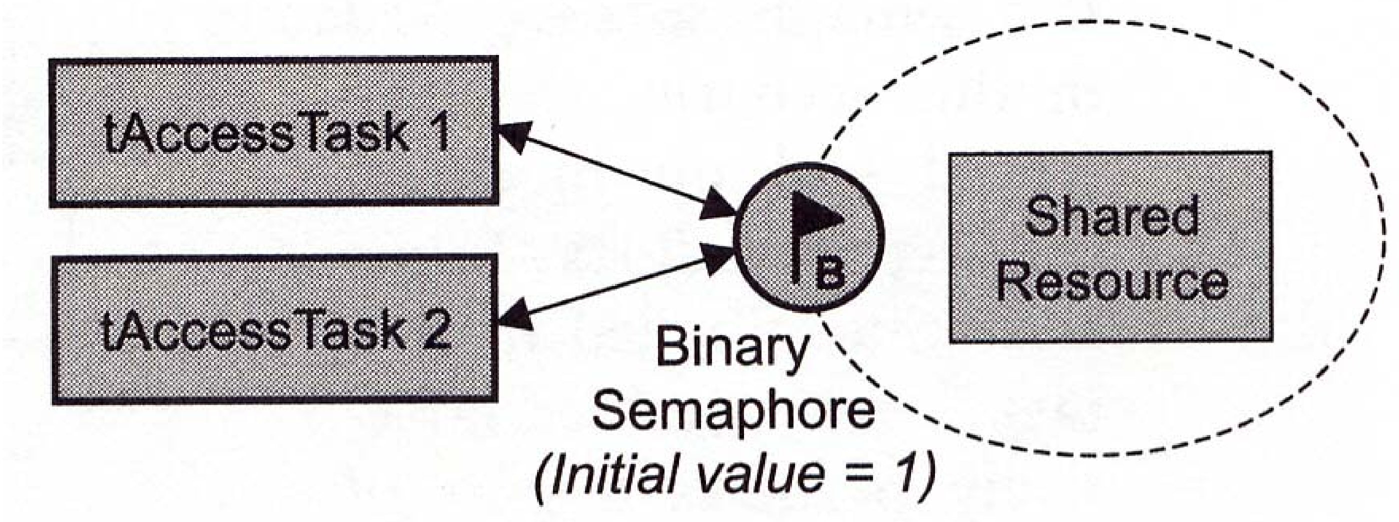
Figure 6.6 Wait-and-signal synchronization between multiple tasks.

Multiple-Task Wait_and_Signal Synchronization

```
tWaitTask1()
{
    Acquire binary semaphore
}
tWaitTask2()
{
    …
}
tSignalTask()
{
    Flush binary semaphore’s task-waiting list
}
```

Single Shared-Resource-Access Synchronization

- Danger: problem when the 3rd task release -> use mutex



Single Shared-Resource-Access Synchronization

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

Recursive Shared-Resource-Access Synchronization

- tAccessTask calls -> Routine A -> Routine B : need to access to the same shared resource

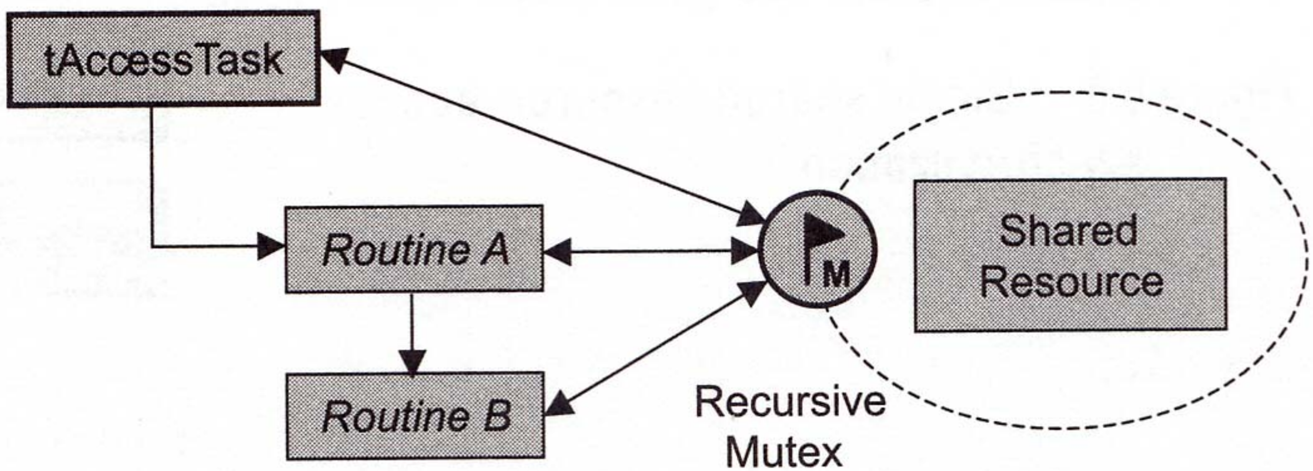


Figure 6.9 Recursive shared- resource-access synchronization.

Recursive Shared-Resource-Access Synchronization

```
tAccessTask()
{
    ...
    Acquire mutex
    Access shared resource
    Call RoutineA
    Release mutex
    ...
}
RoutineA()
{
    ...
    Acquire mutex
    Access shared resource
    Call RoutineB
    Release mutex
    ...
}
```

```
RoutineB()
{
    ...
    Acquire mutex
    Access shared resource
    Call RoutineB
    Release mutex
    ...
}
```