Introduction to Real-Time Operating Systems

GPOS vs RTOS

- General purpose operating systems
- Real-time operating systems

GPOS vs RTOS: Similarities

- Multitasking
- Resource management
- OS services to applications
- Abstracting the hardware

Characteristics of RTOS

- Reliability in embedded application
- Scale up or down ability
- Faster performance
- Reduced memory requirement
- Scheduling policies for real-time
- Diskless
- portability

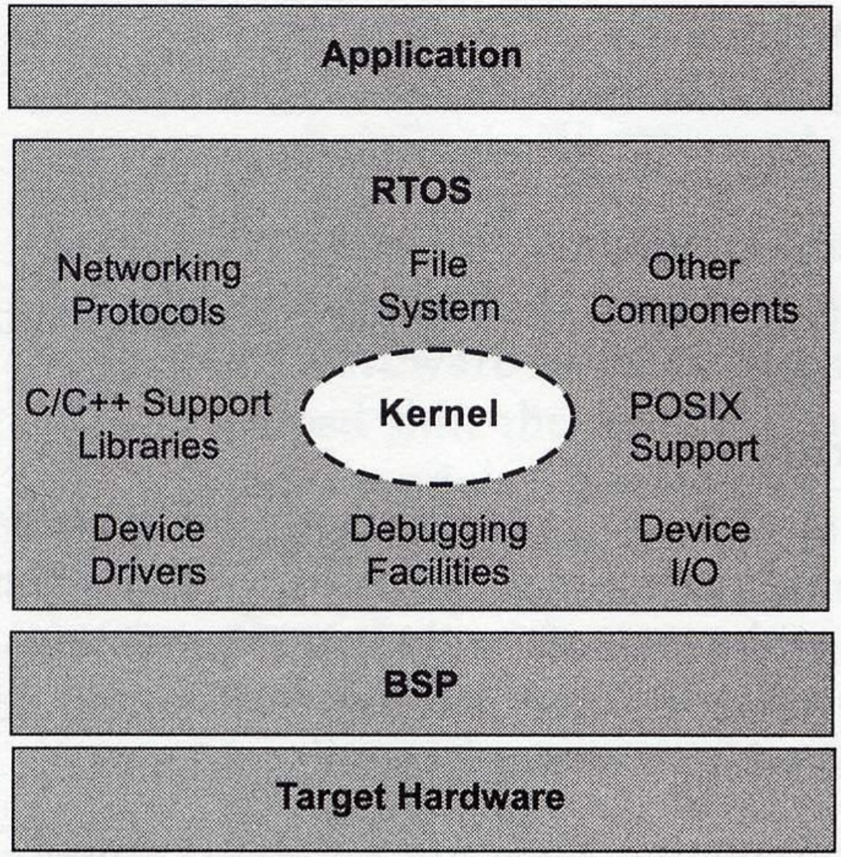


Figure 4.1 High-level view of an RTOS, its kernel, and other components found in embedded systems.

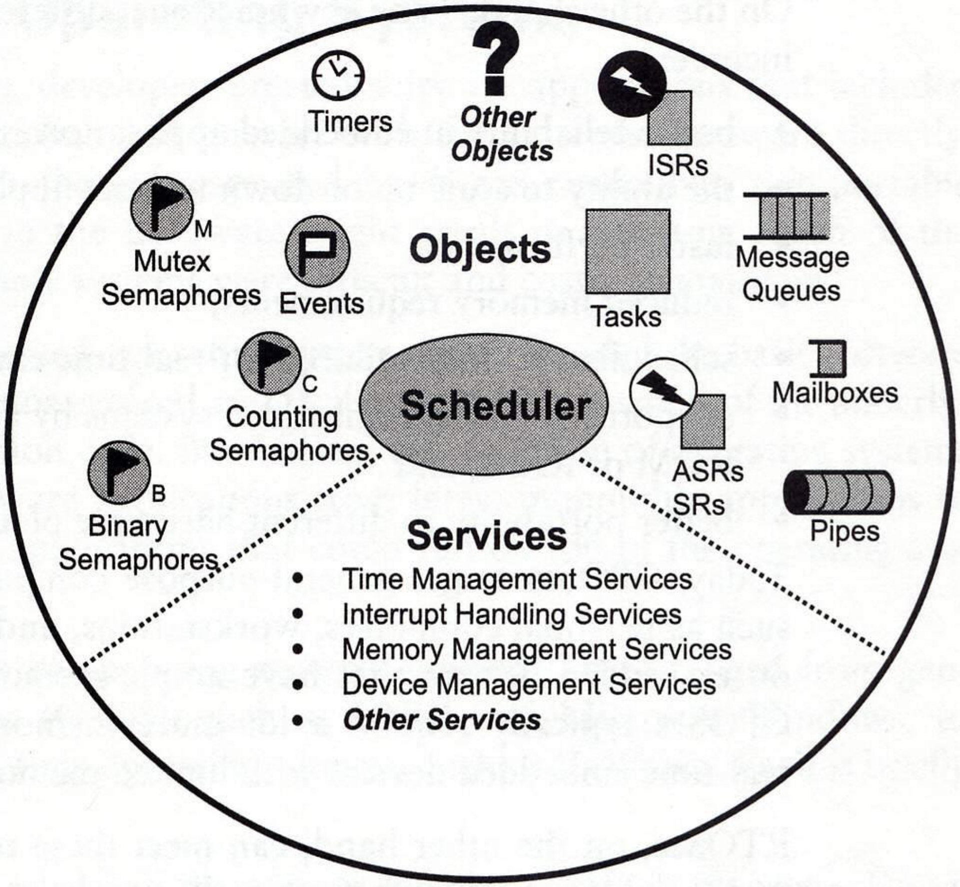


Figure 4.2 Common components in an RTOS kernel that including objects, the scheduler, and some services.

Kernel Objects

- Help developers creates applications for real-time embedded systems

Scheduler

- Determine which task executes when
- Schedulable entities-a kernel object that can compete for execution on a system-> process, task
- Multitasking: many thread of execution appear to be running concurrently

Scheduler

- Context: the state of CPU registers
- Context switch
- When a new task is created, TCB(task control block) is also created
- TCB: system data structure

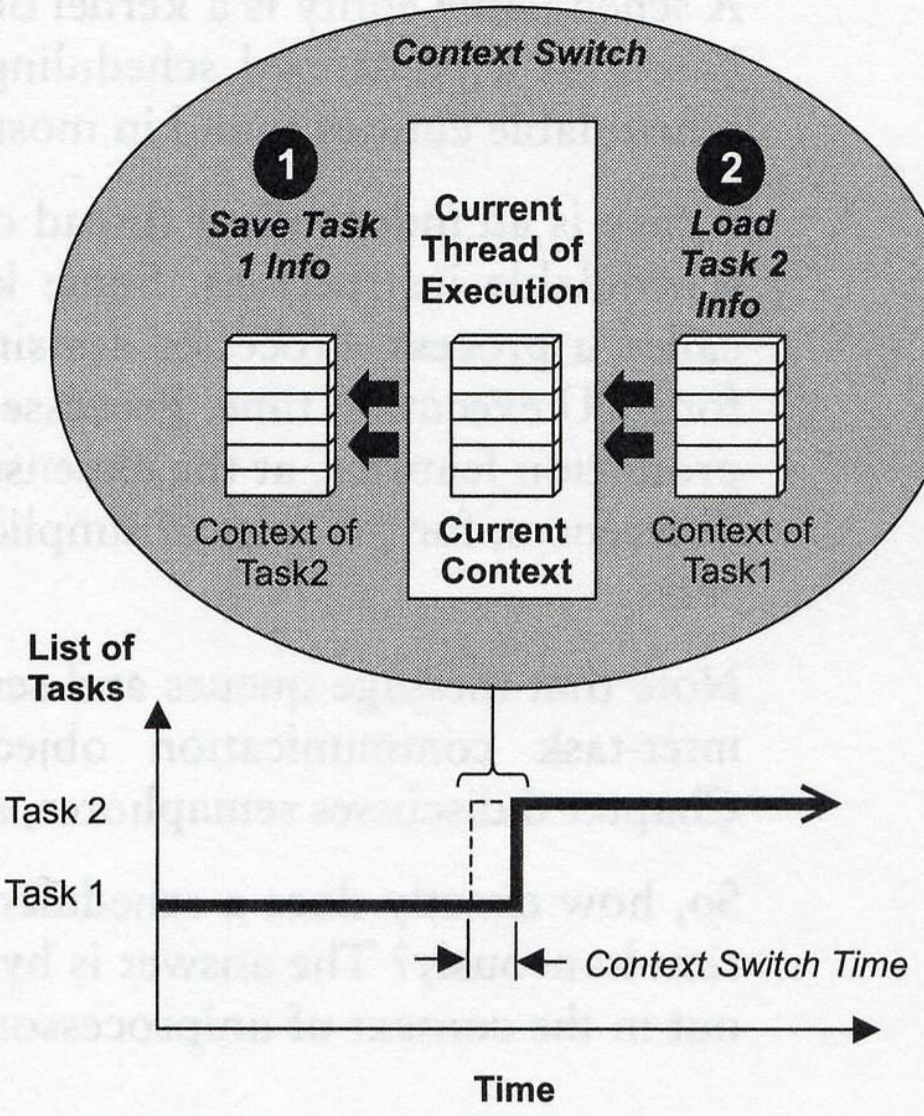


Figure 4.3 Multitasking using a context switch.

Scheduling Algorithms

- Preemptive priority-based scheduling
- Round-robin scheduling

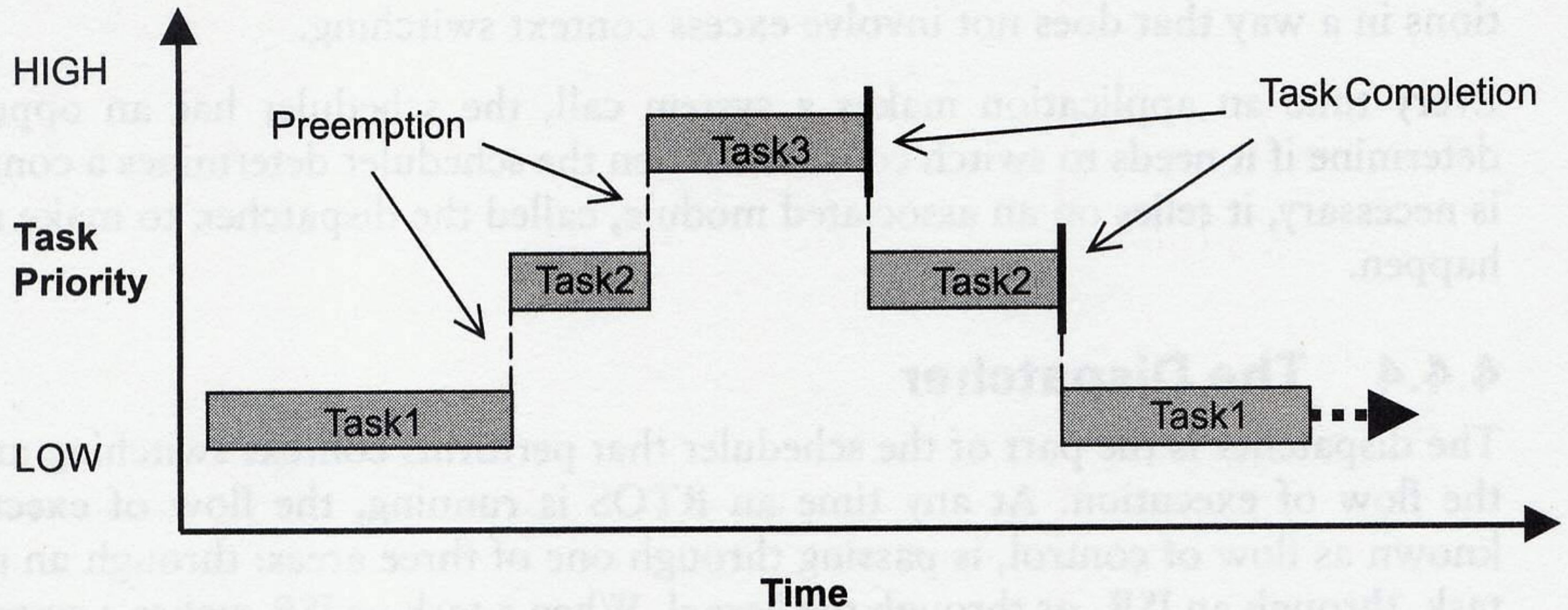


Figure 4.4 Preemptive priority-based scheduling.

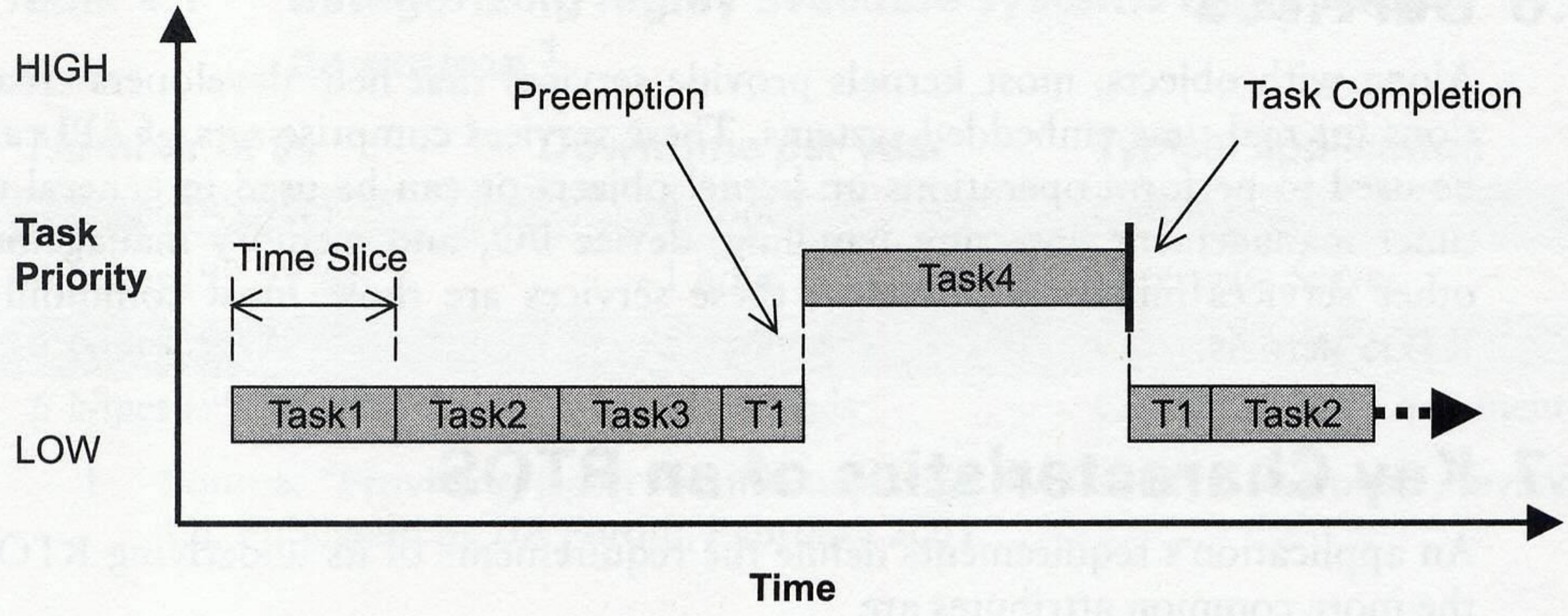


Figure 4.5 Round-robin and preemptive scheduling.

Objects

- Tasks
- Semaphore: token-like objects for synchronization & mutual exclusion
- Message queue: buffer-like data structures

Common Real-Time Design Problems

- Concurrency
- Activity synchronization
- Data communication

- Developers combine basic kernel objects

Tasks

Defining a Task

- A task is an independent thread of execution that can compete with each other concurrent tasks for processor execution time
- Developer decompose applications into multiple concurrent tasks to optimize the handling of inputs and outputs within set time constraints

Task States

- Ready
- Running
- Blocked

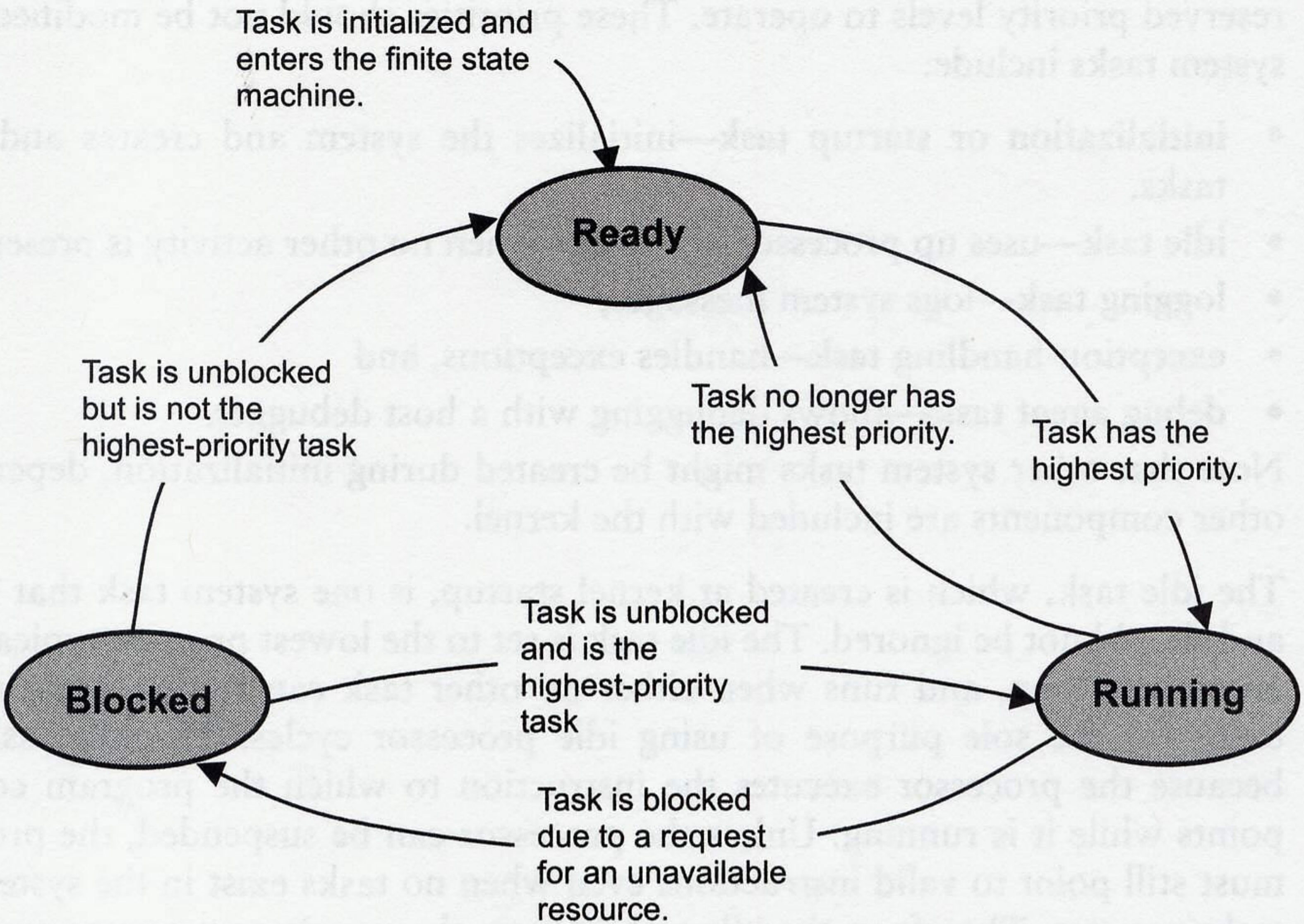


Figure 5.2 A typical finite state machine for task execution states.

Ready State

- Most kernels support more than one task per priority level
- Task-ready-list

Running State

- Can move to the blocked state

1. By making a call requesting an un available resource
2. By making a call requesting to wait for an event to occur
3. By making a call to delay

Blocked State

- Without blocked state, lower priority tasks could not run!
- CPU starvation occurs when higher priority tasks use all of the CPU execution time and lower priority tasks do not get to run.
- A task can only move to the blocked state by making a blocking call, requesting that some blocking condition be met.

Typical Task Structures

- Run-to-completion tasks
- Endless-loop tasks

Run-to-Completion Tasks

- Application-level initialization task
- The application initialization task typically has a higher priority than the application tasks that it creates so that its initialization work is not preempted.

```
RunToCompletionTask()
{
    initialize application
    create ‘endless loop tasks’-lower priority
    creates kernel objects
    delete or suspend this task
}
```

Endless-Loop Tasks

- One or more blocking calls within the body of the loop

```
EndlessLoopTask()
{
    initialization code
    Loop Forever
    {
        body of loop
        make one or more blocking calls
    }
}
```

Synchronization,Communication and Concurrency

- Tasks synchronize and communicate by using intertask primitives (semaphores, message queues, signals, pipes)

Semaphores

Semaphore (Token)

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

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)

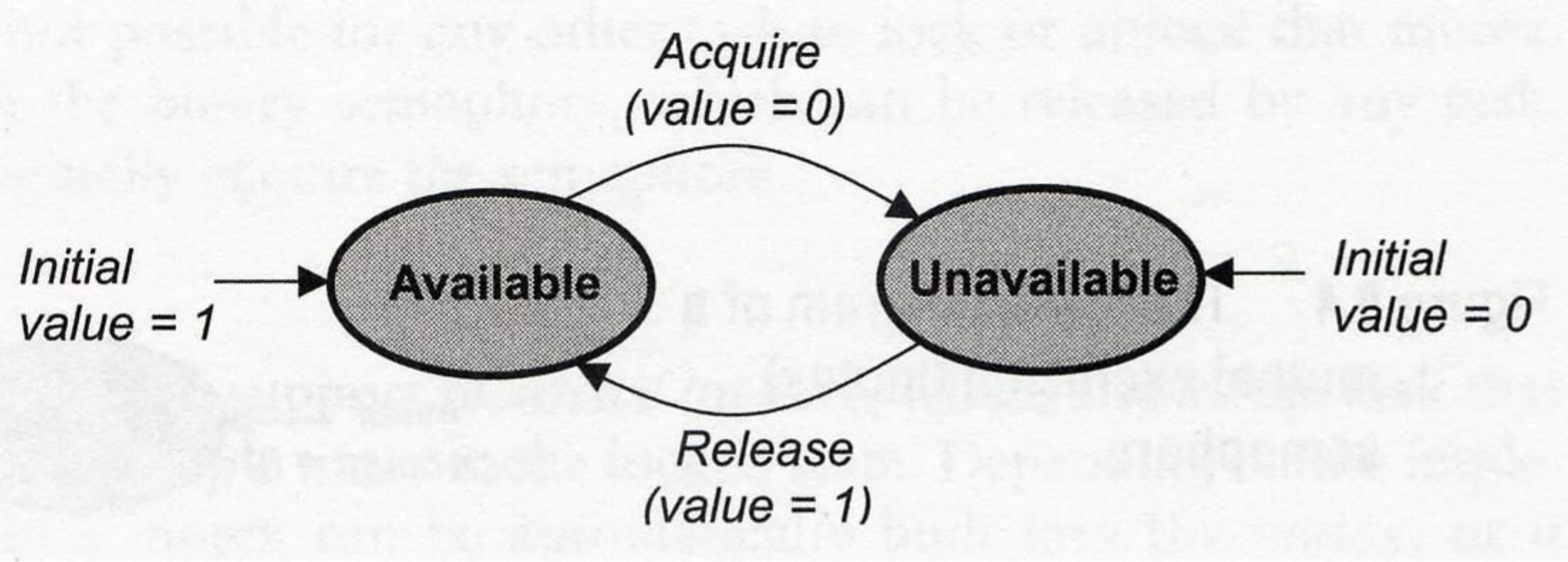
Semaphore Count

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

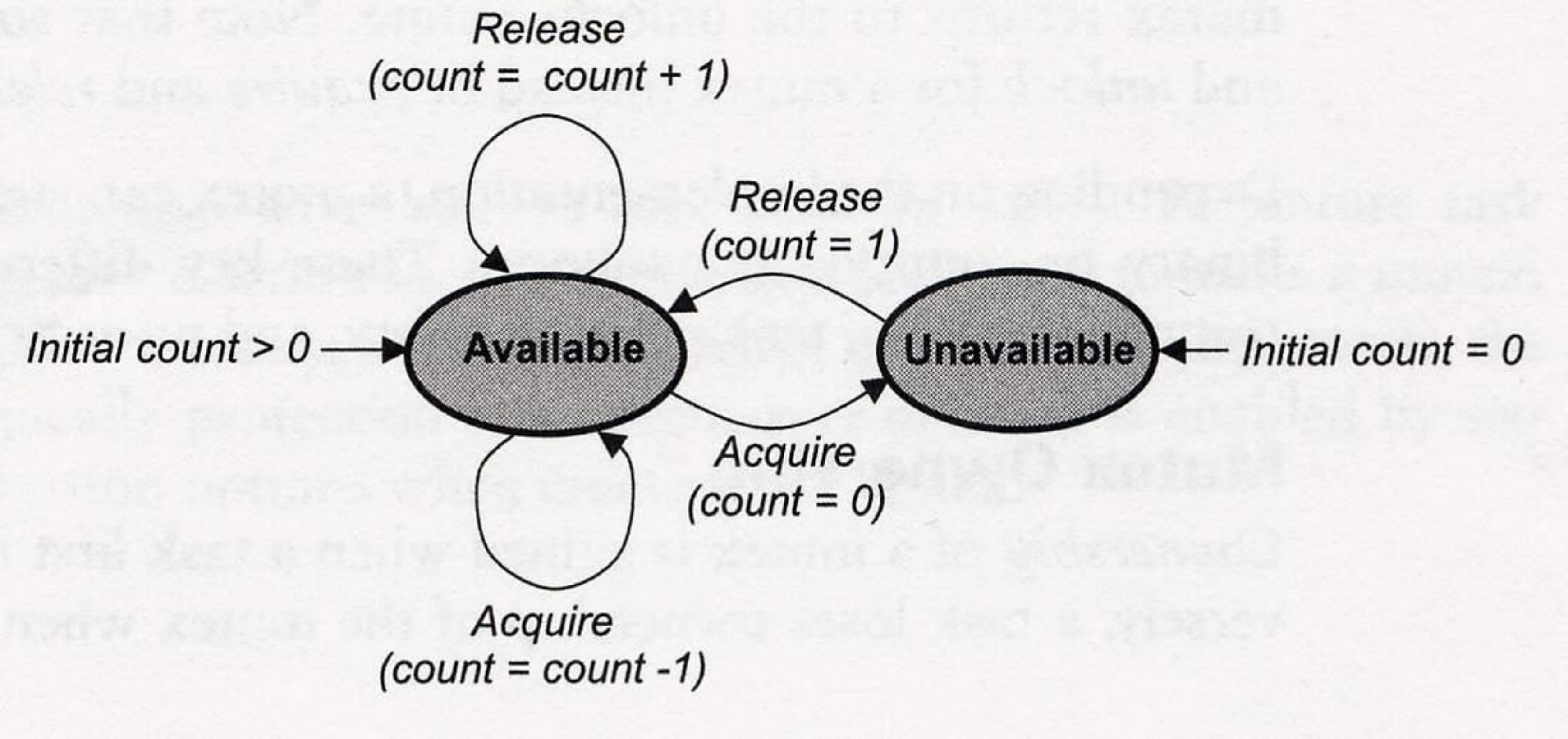
- Token count = 0 : a requesting task blocks

Binary Semaphore

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

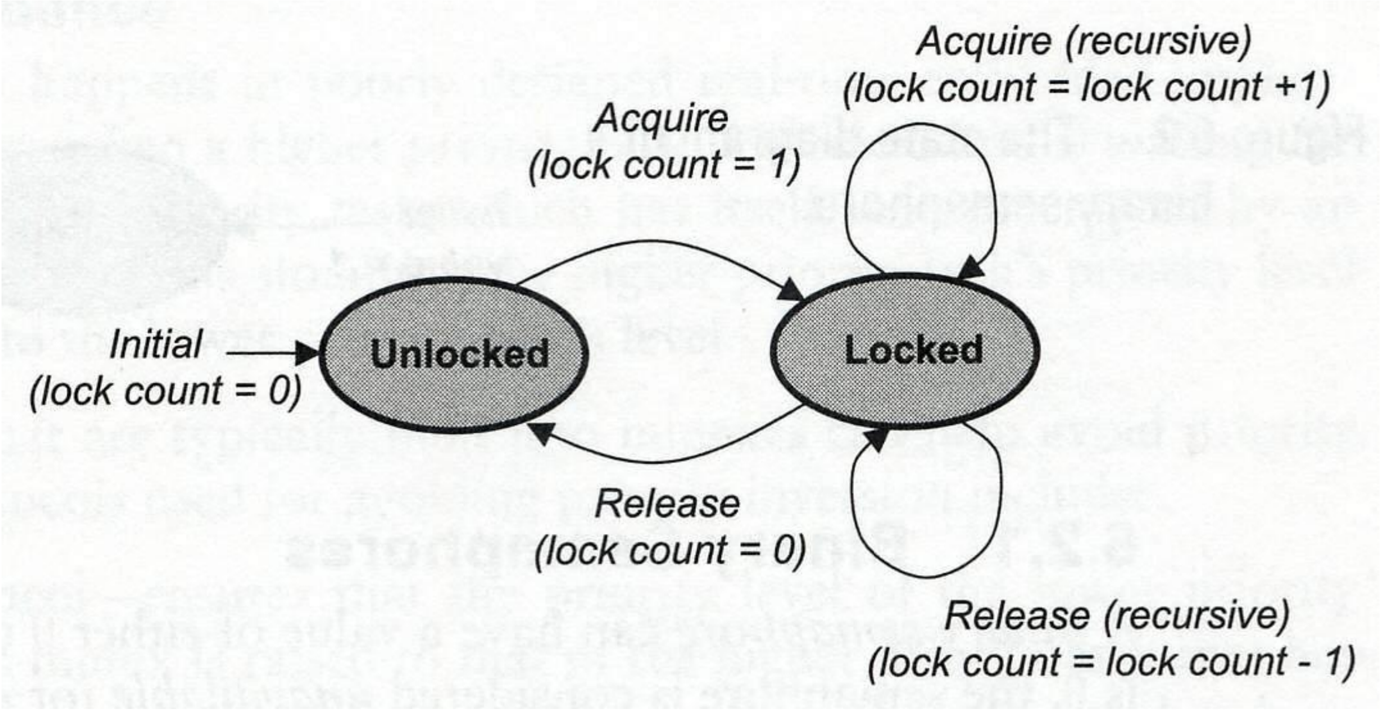


Counting Semaphore



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

Priority Inversion

Priority Inversion

- Priority inversion is a situation in which a low-priority task executes while a higher priority task wait on it due to resource contentions
- Task interdependency

Priority Inversion Example

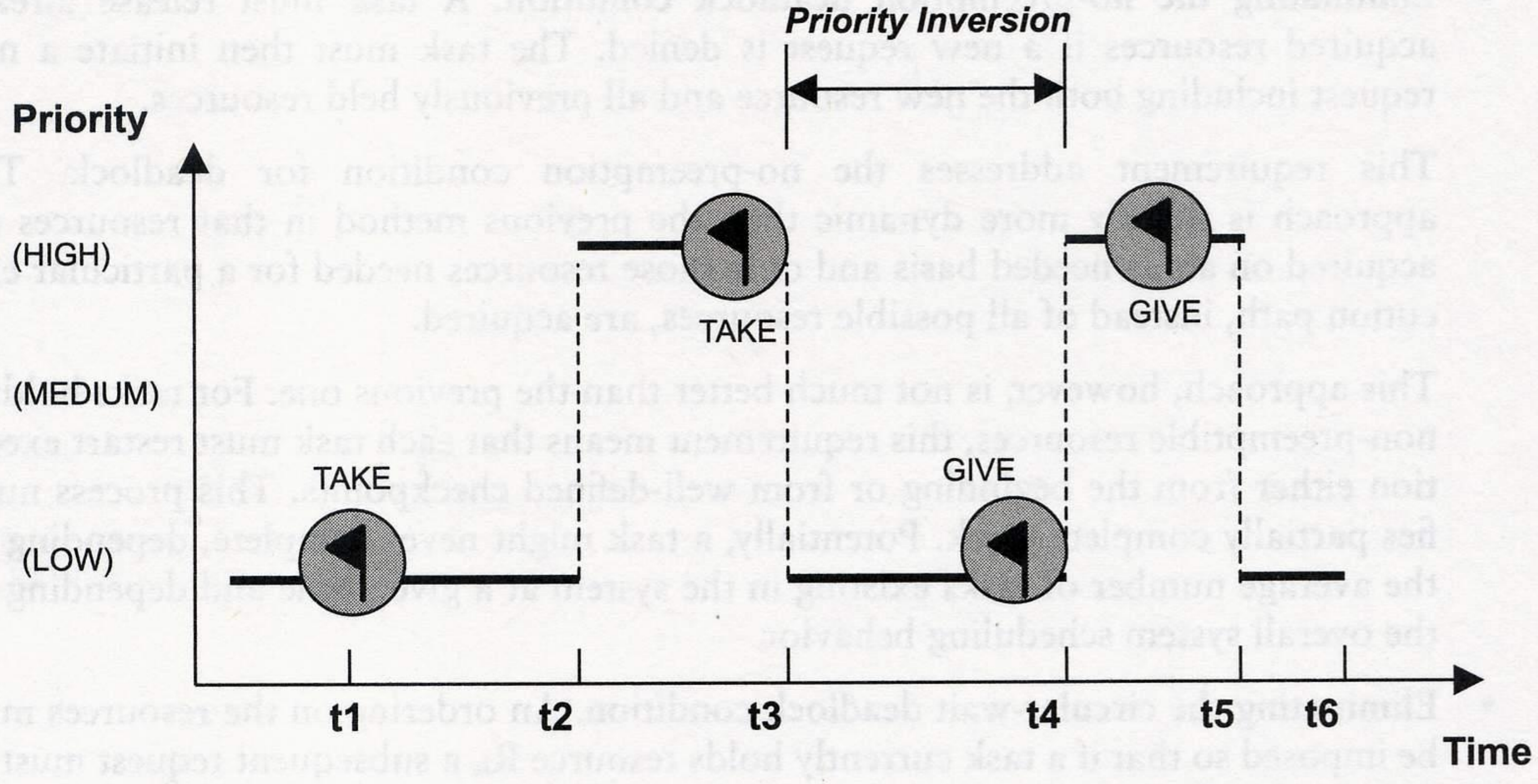


Figure 16.6 Priority inversion example.

Unbounded Priority Inversion Example

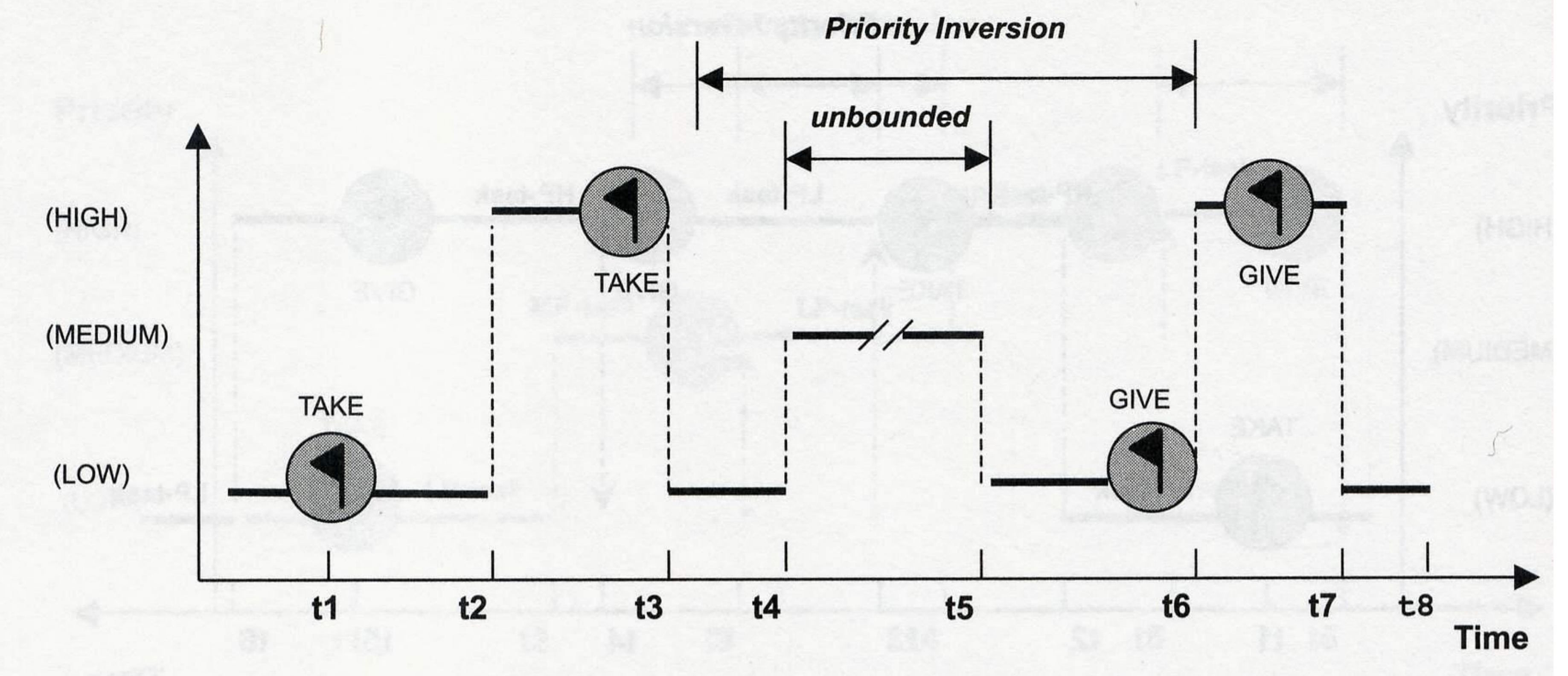


Figure 16.7 Unbounded priority inversion example.

Priority Inheritance Protocol

- R: resource, T: the Task requesting R

1. If R is in use, T is blocked
2. If R is free, R is allocated to T
3. When a task of a higher priority requests the same resource, T's executing priority is raised to the requesting task's priority
4. The task returns to its previous priority when it releases R

Priority Inheritance Protocol

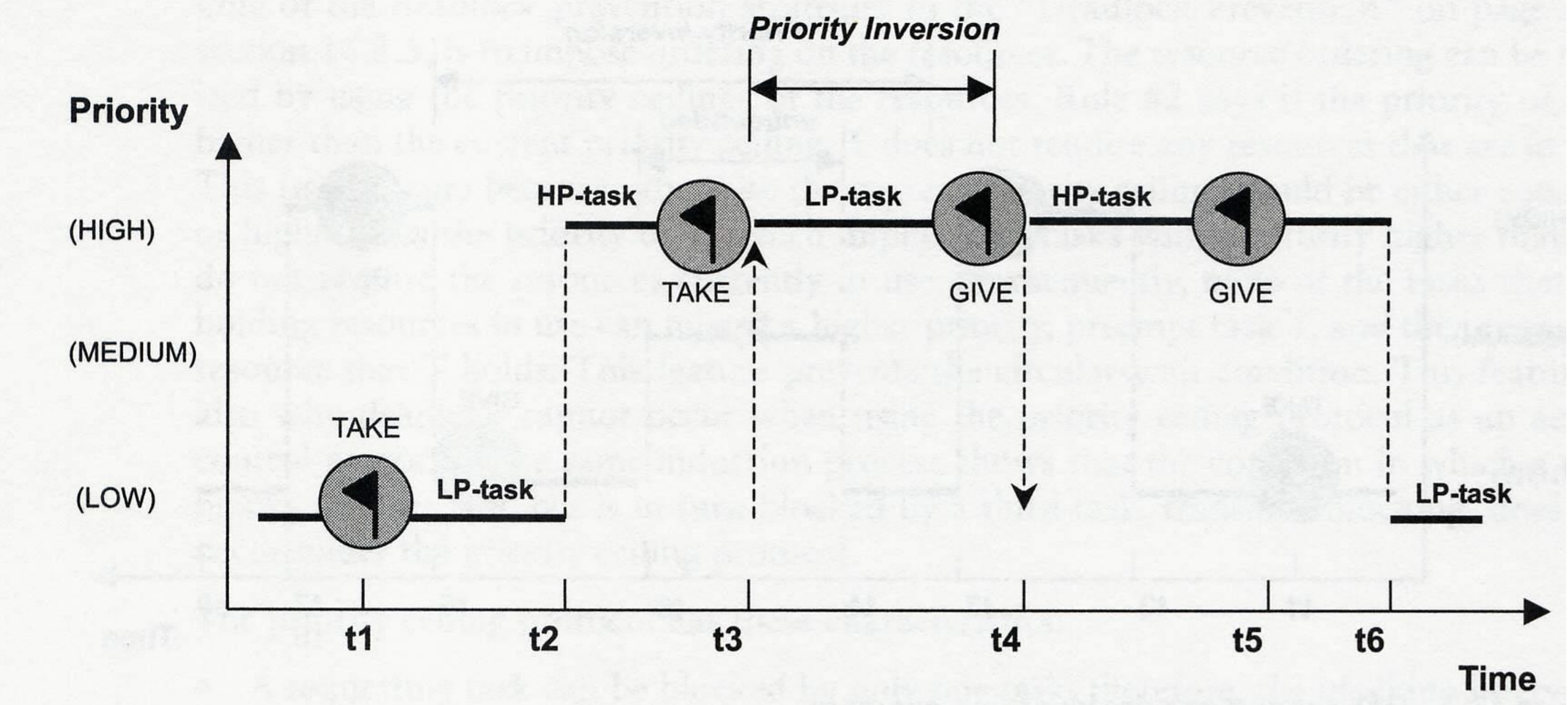


Figure 16.8 Priority inheritance protocol example.

Priority Inheritance Protocol

- Priority inheritance is dynamic

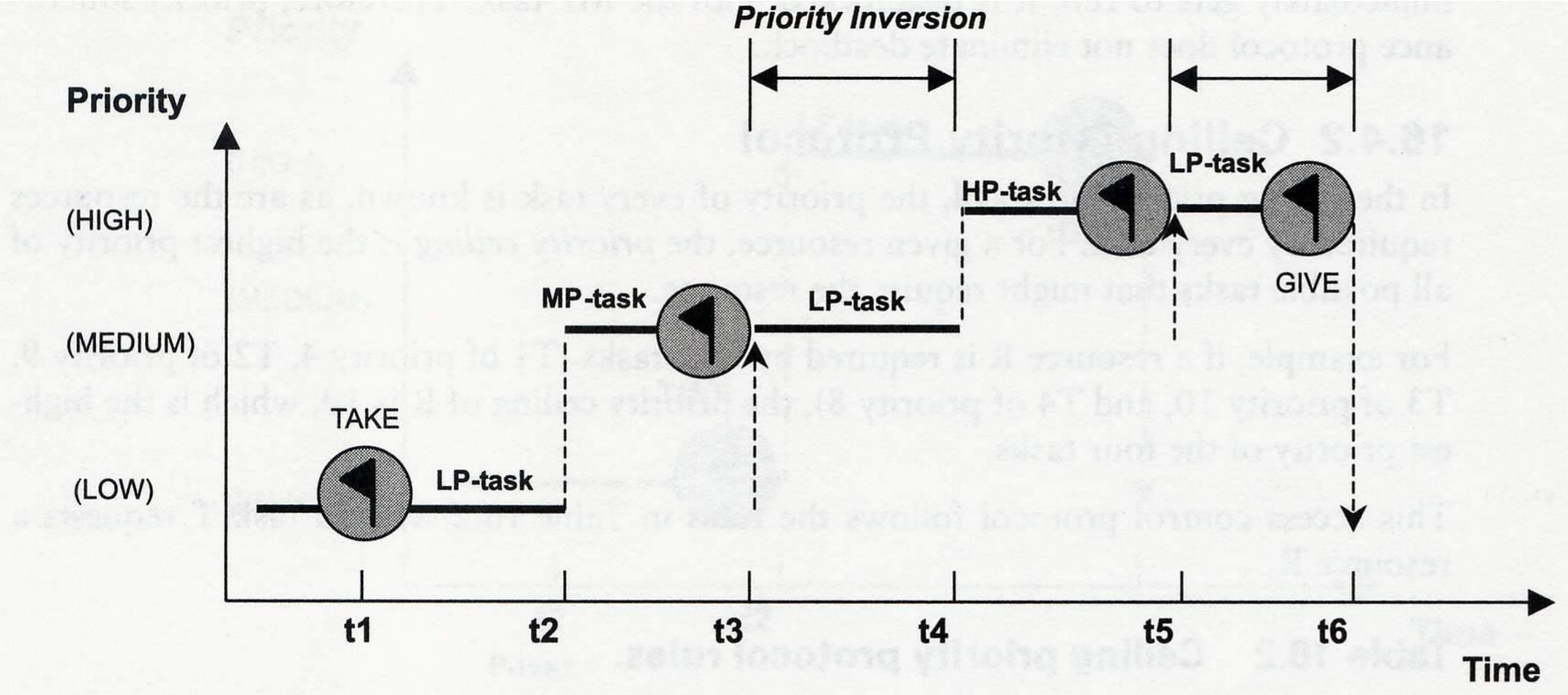


Figure 16.9 Transitive priority promotion example.

Typical Semaphore Use

- Wait-and-Signal Synchronization

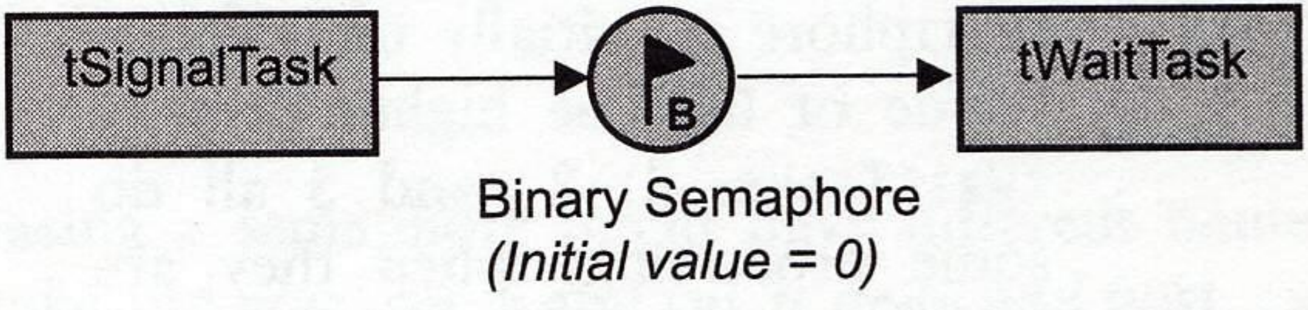


Figure 6.5 Wait-and-signal synchronization between two tasks.

Wait-and-Signal Synchronization

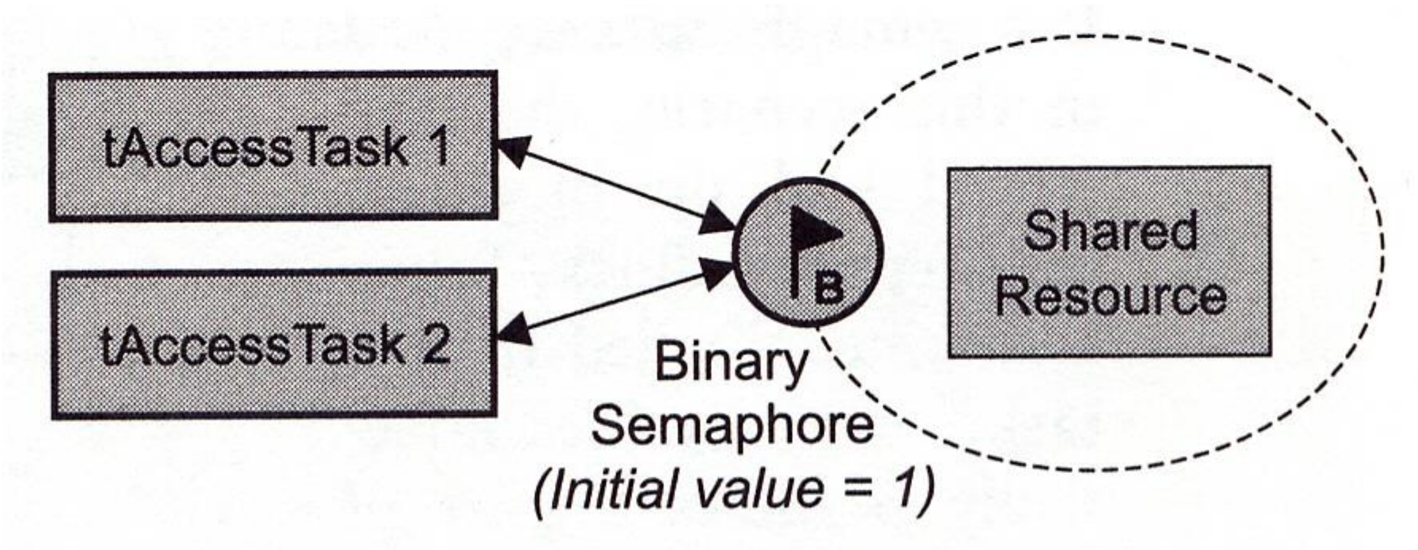
- tWaitTask runs first
- tWaitTask makes a request to acquire the semaphore but blocked
- tSignalTask has a chance to run
- tSignalTask releases the semaphore
- tWaitTask unblocked and running

Wait-and-Signal Synchronization

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

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

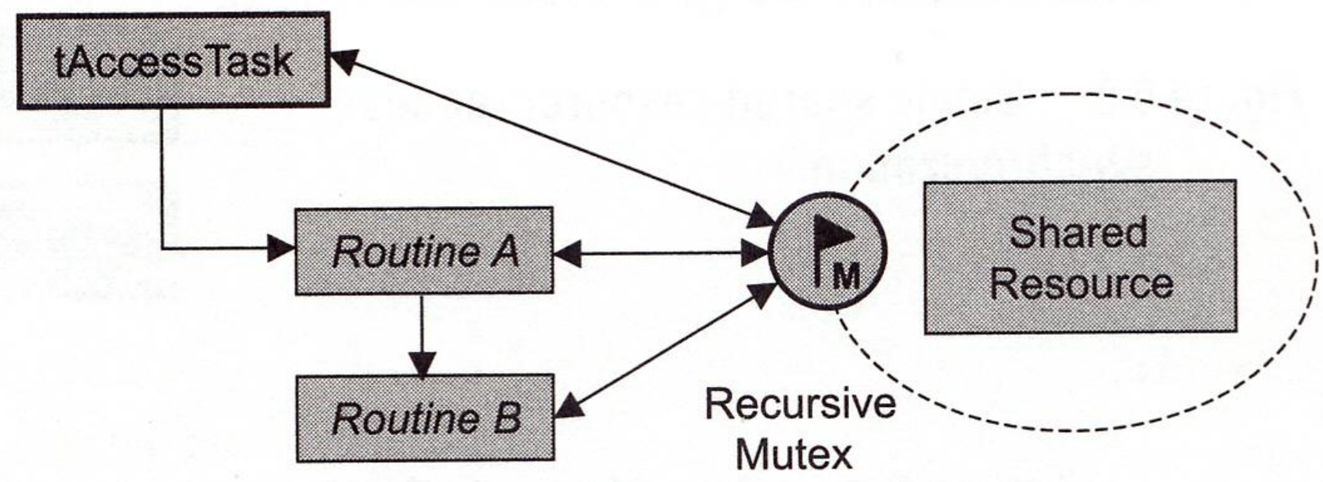


Figure 6.9 Recursive shared- resource-access synchronization.

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

    Release mutex
    …
}
```

Message Queues

Message Queues

- A message queue is a buffer-like object through which tasks and ISRs send and receive messages to communicate and synchronize with data
- It temporarily holds message from a sender until the intended receiver is ready to read them.

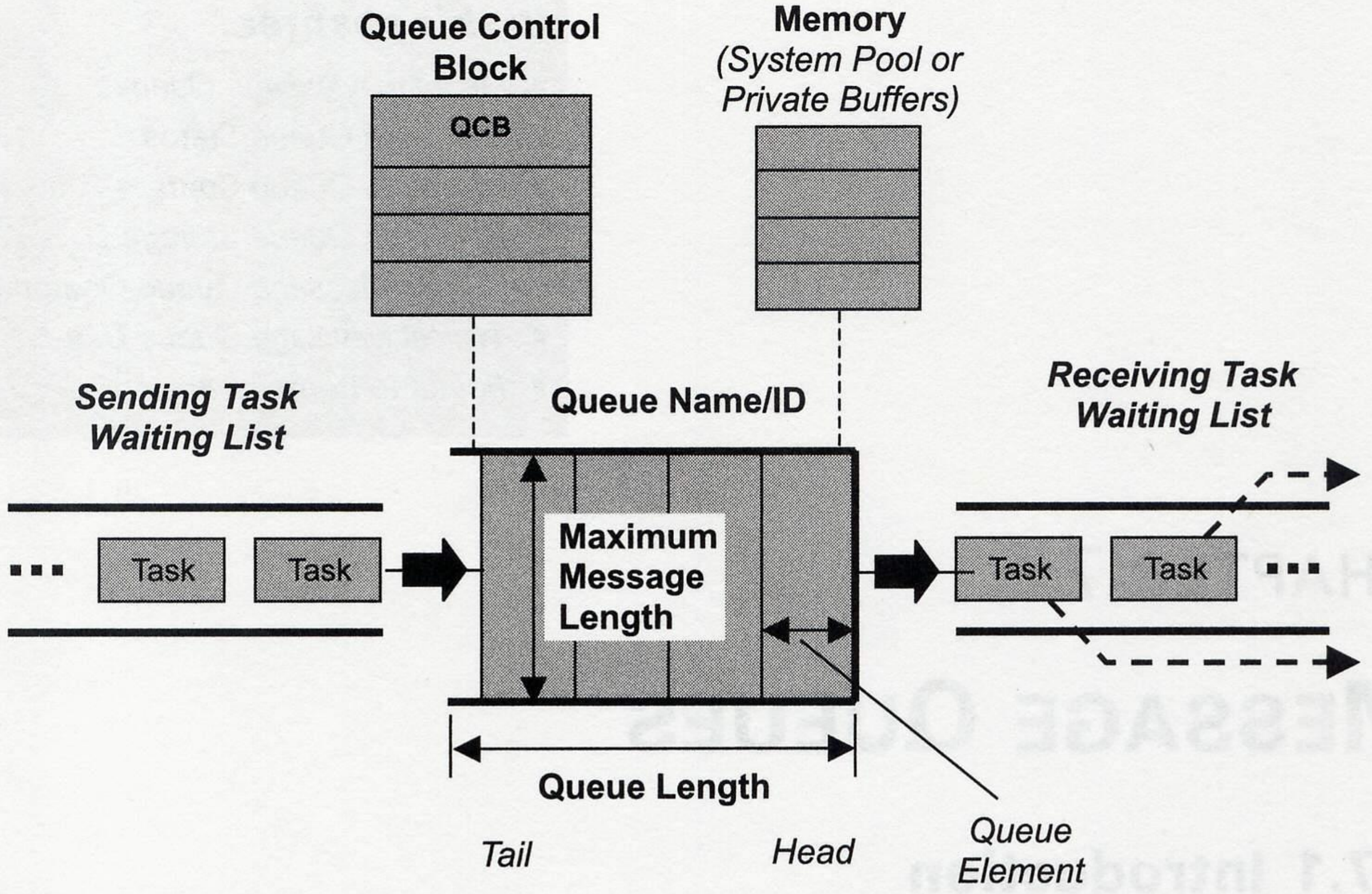


Figure 7.1 A message queue, its associated parameters, and supporting data structures.

Message Queue Content

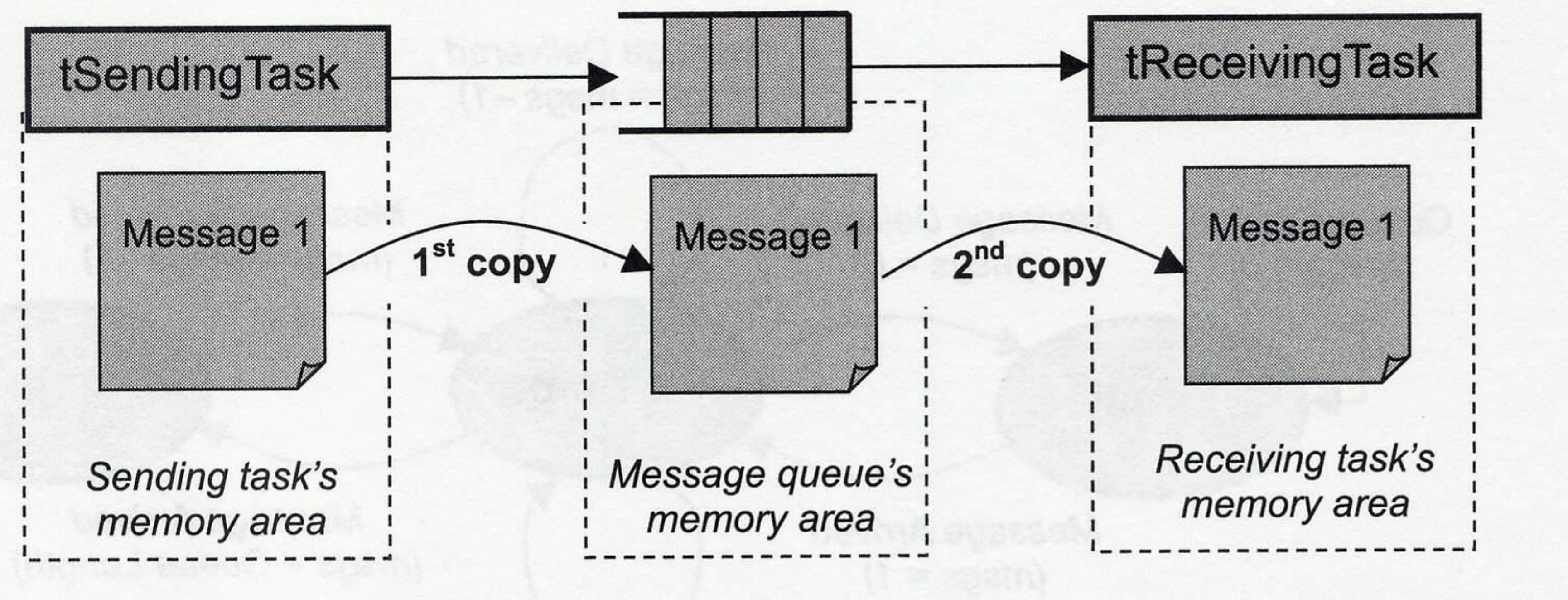


Figure 7.3 Message copying and memory use for sending and receiving messages.

- For long message, use pointer

Typical Message Queue Use

- Non-interlocked, one-way data communication (loosely coupled)
- Not synchronized
- Does not require ACK

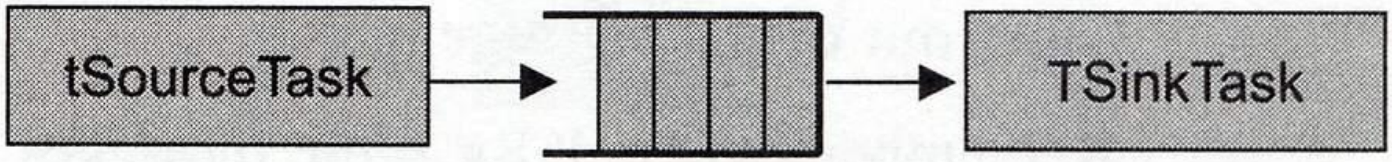


Figure 7.6 Non-interlocked, one-way data communication.