Synchronization And Communication

A Training for Software Development Professionals

Training Goals

- Understanding Real-Time Concepts
- Gaining the Knowledge of Real-Time Operating Systems and Internals

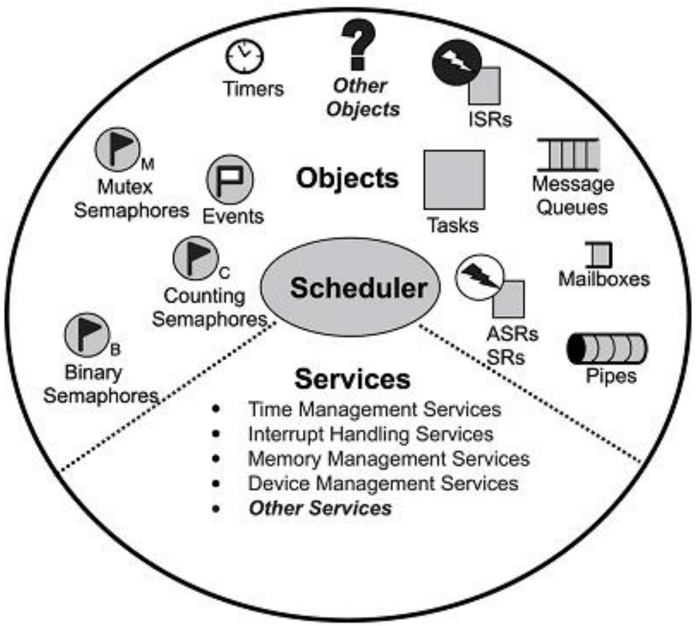
Contacts/Resources

- Özgür Aytekin
  - Embedded Software Engineer at Collins Aerospace
  - ozguraytekin@gmail.com

Chapter 1

Synchronization And Communication Kernel Objects

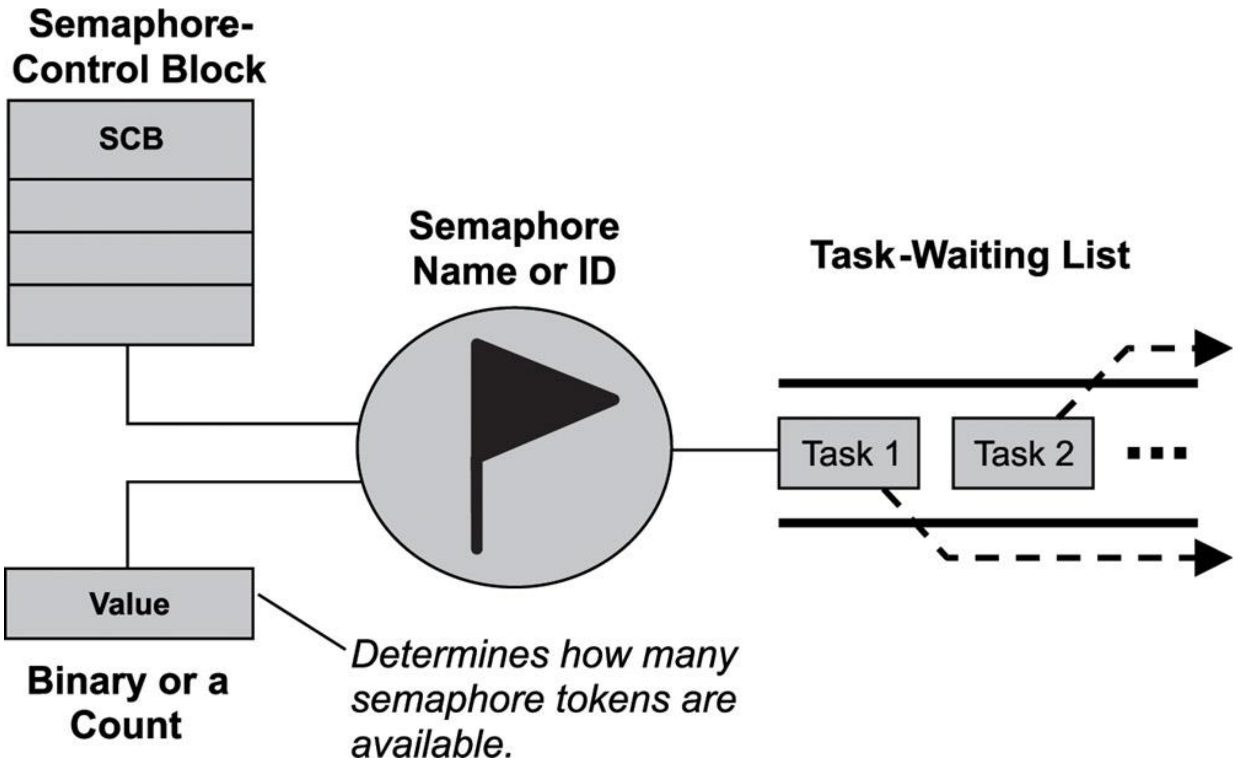
Agenda



- ➢ **Semaphore**
- ❑ Message Queues
- ❑ Pipes
- ❑ Connections
- ❑ Signals
- ❑ ARINC 653 Ports
- ❑ Buffer and Blackboards
- ❑ Event Registers

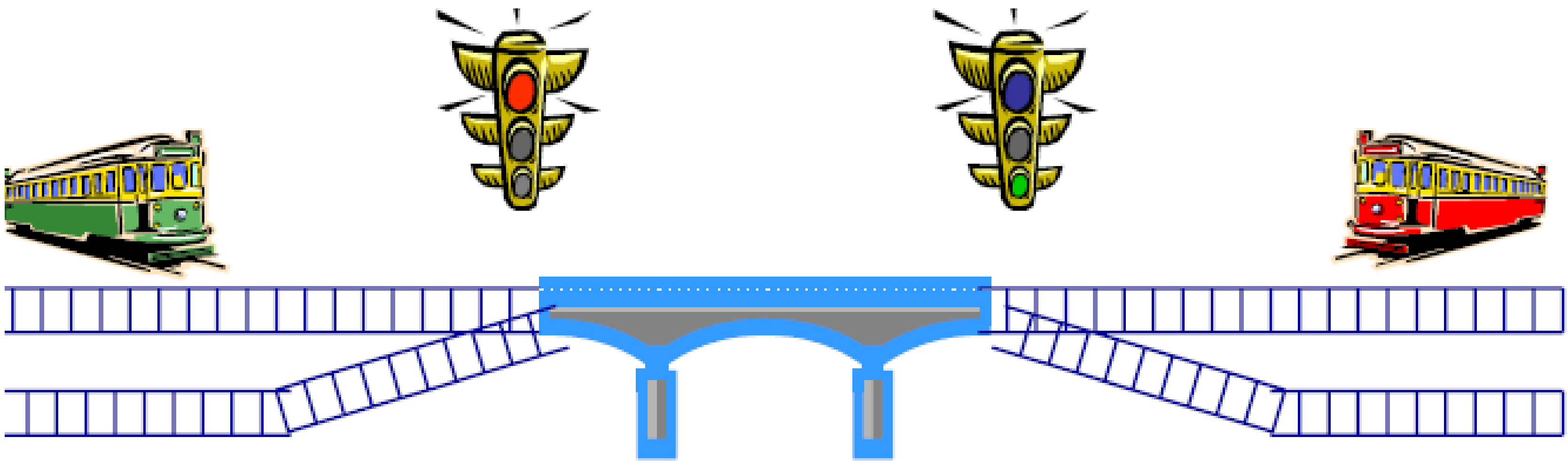
Semaphore

- 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.



Semaphore (cont)

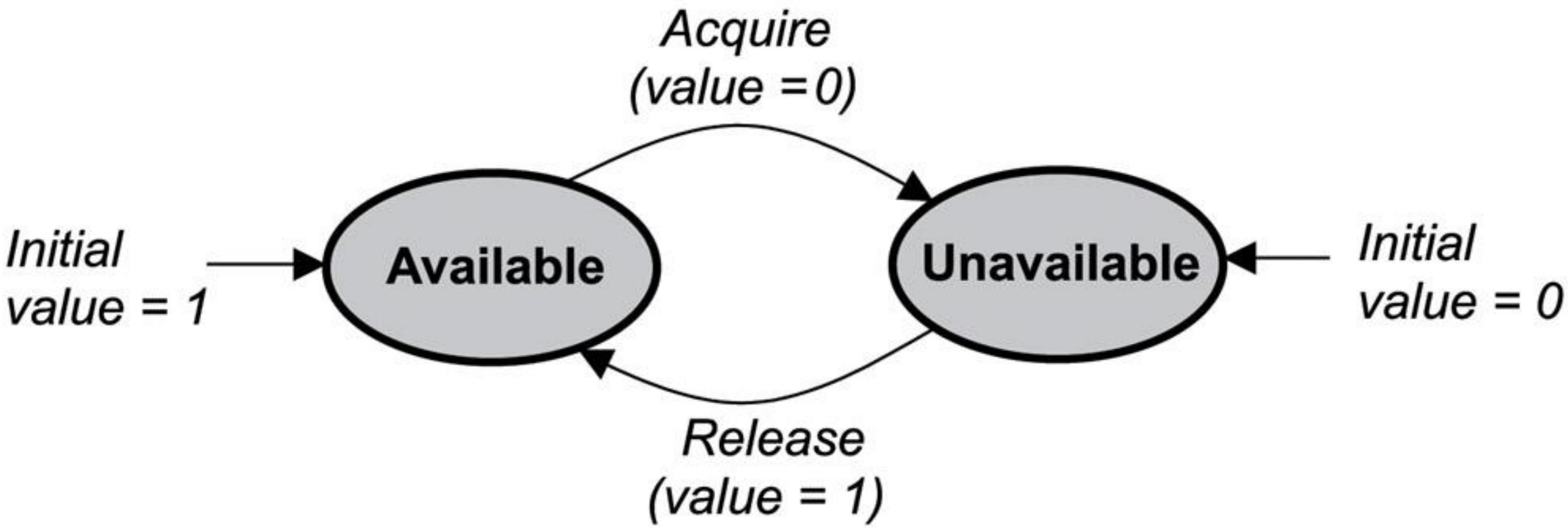
- A “Semaphore” is a traffic light



- Semaphores are a common solution in operating systems to deal with race conditions
- In real-time systems they lead to priority inversion

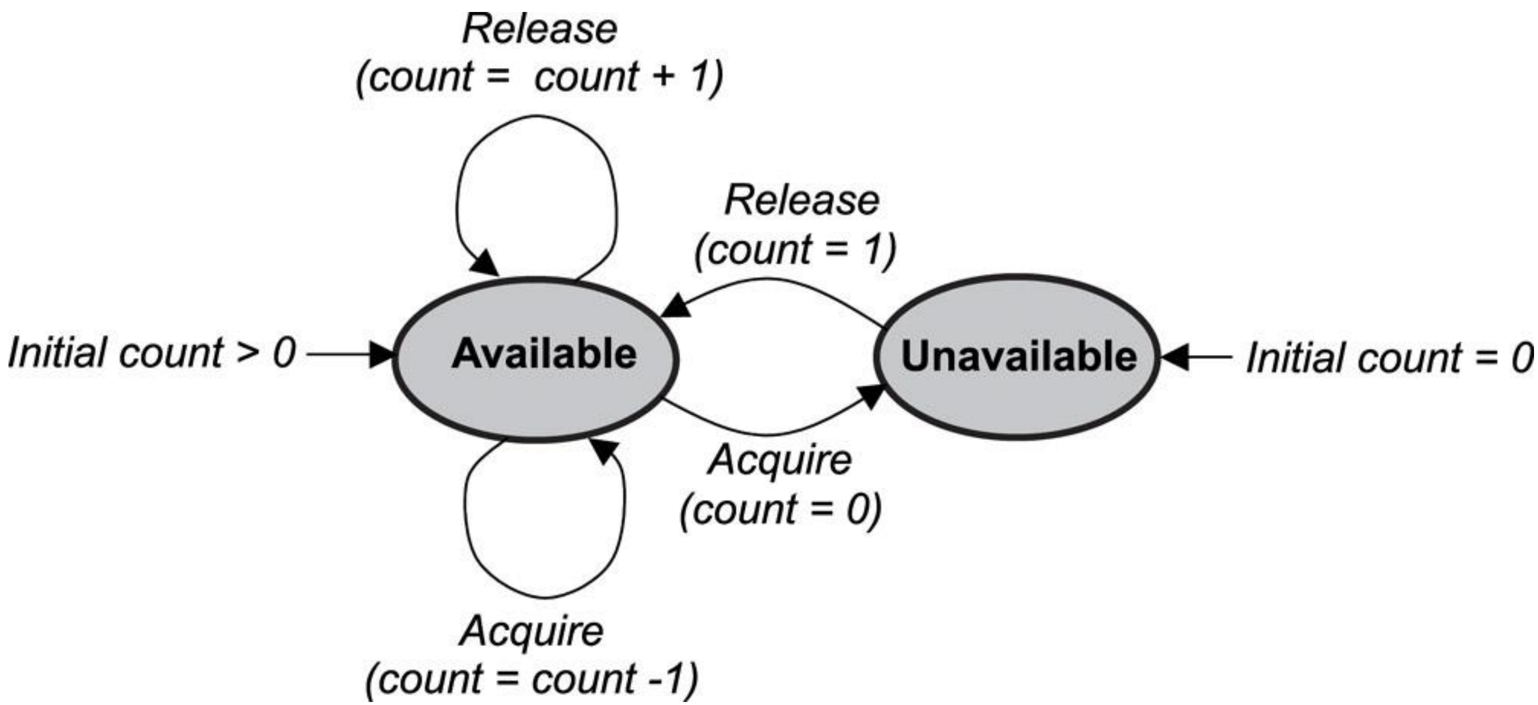
Binary Semaphores

- A binary semaphore can have a value of either 0 or 1



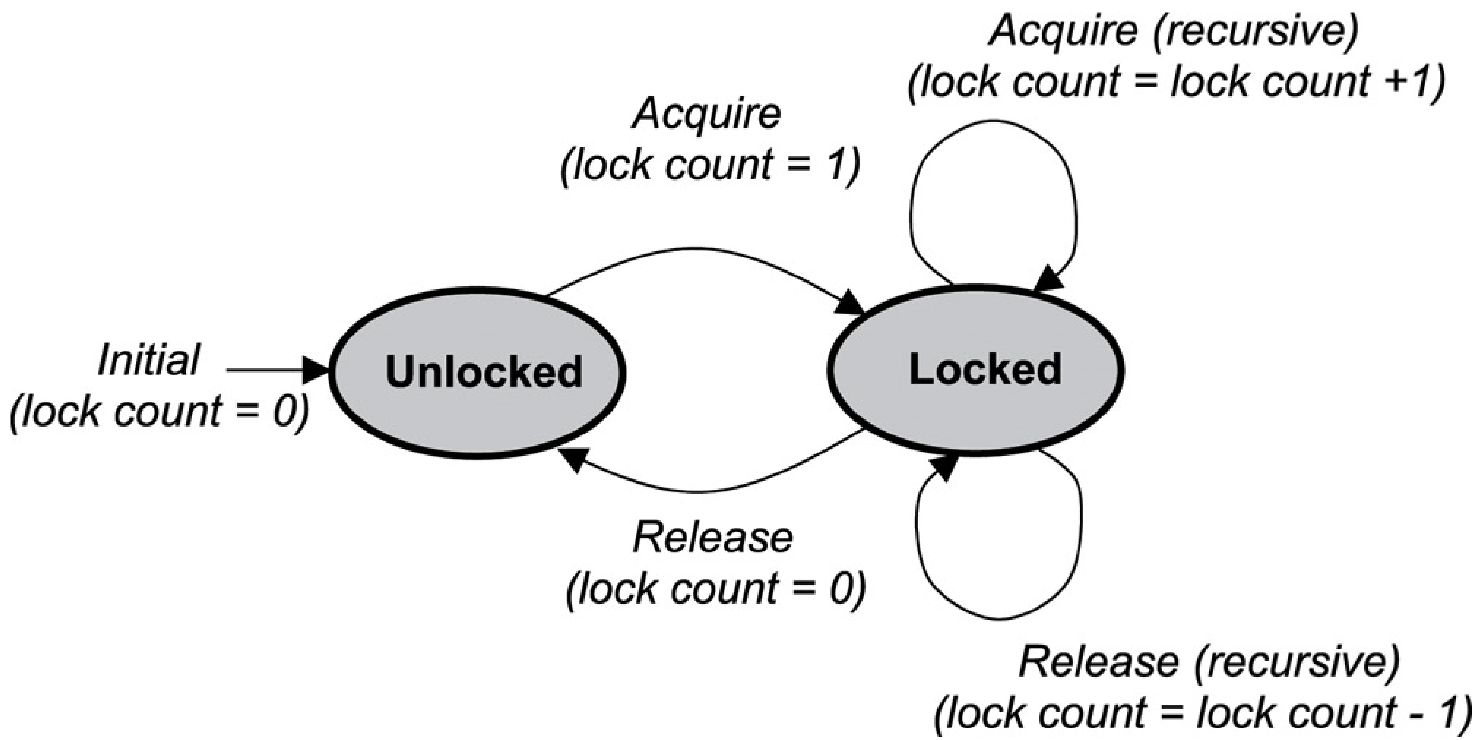
Counting Semaphores

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

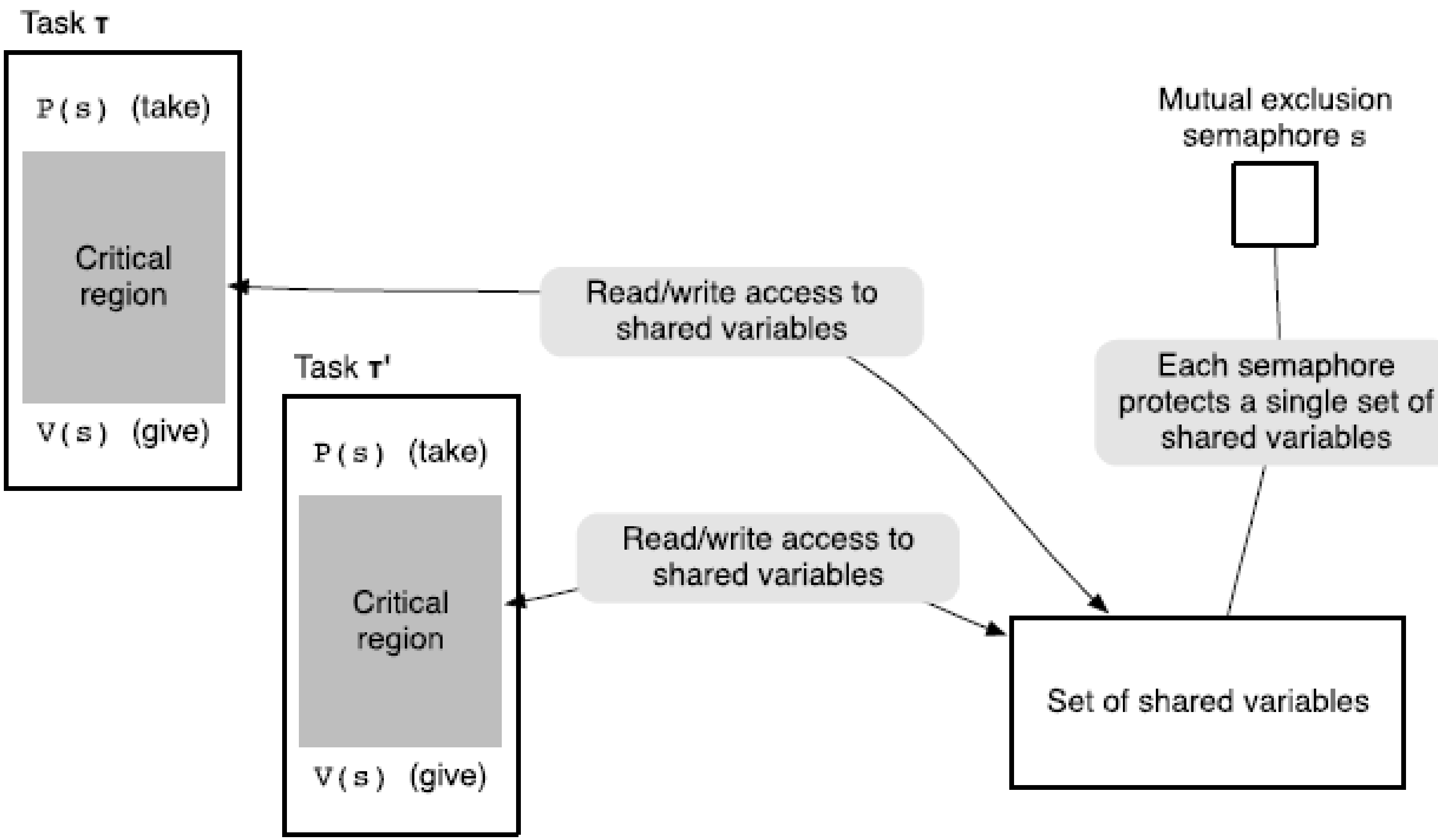


Mutual Exclusion (Mutex) Semaphores

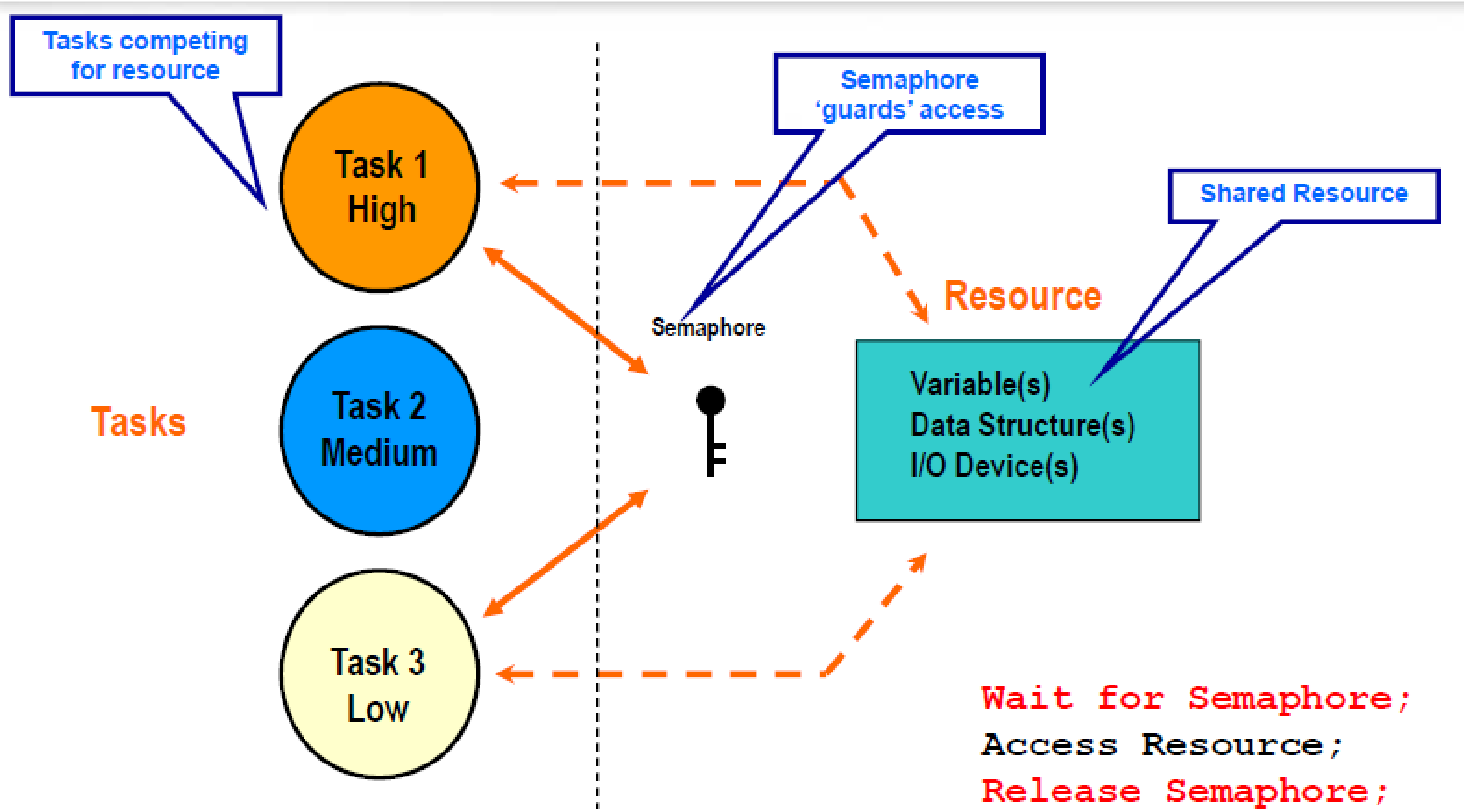
- A mutual exclusion (mutex) semaphore is a special binary semaphore that supports ownership, recursive access, task deletion safety



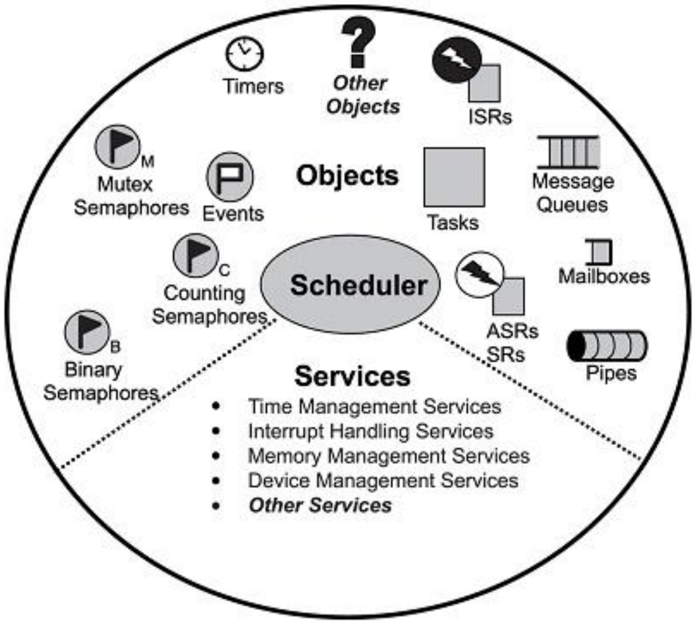
Usage of a semaphore for mutual exclusion



Usage of a semaphore for resource synchronization



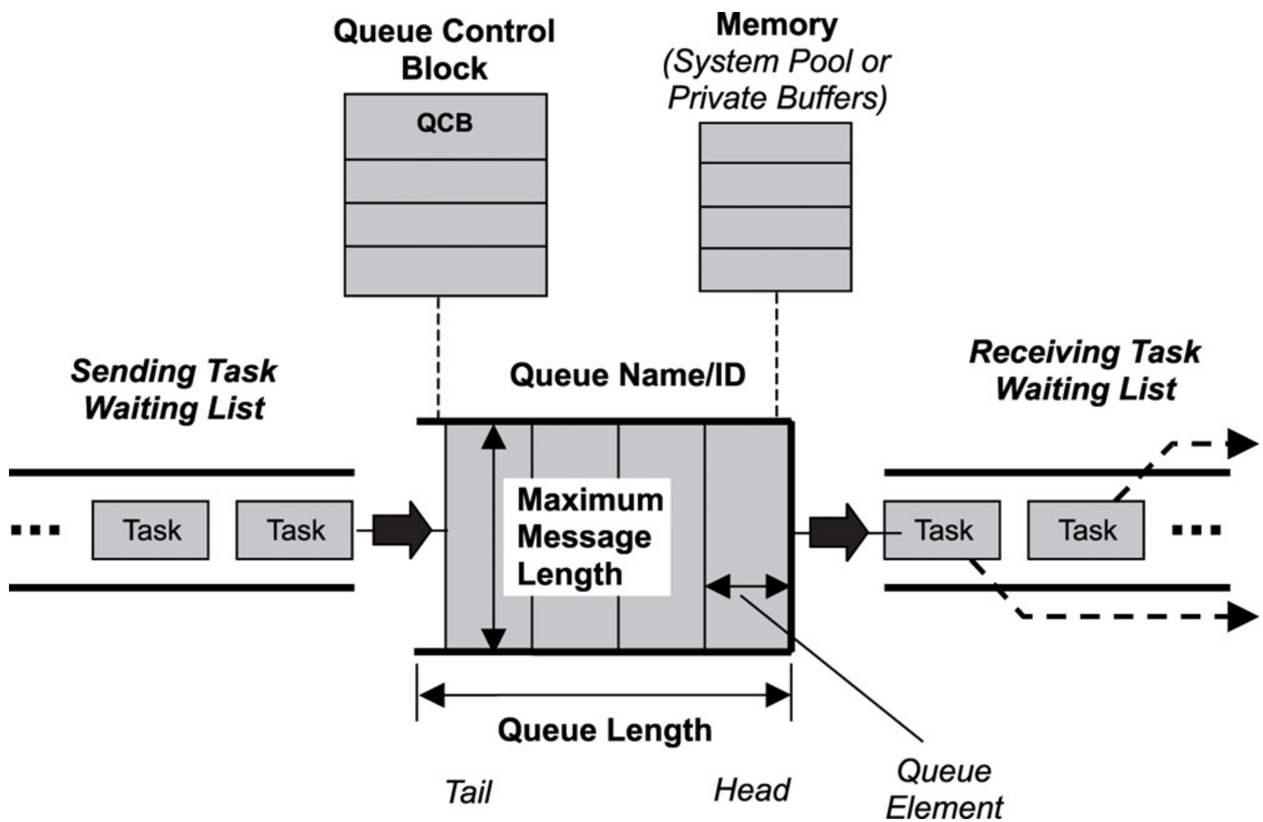
Agenda



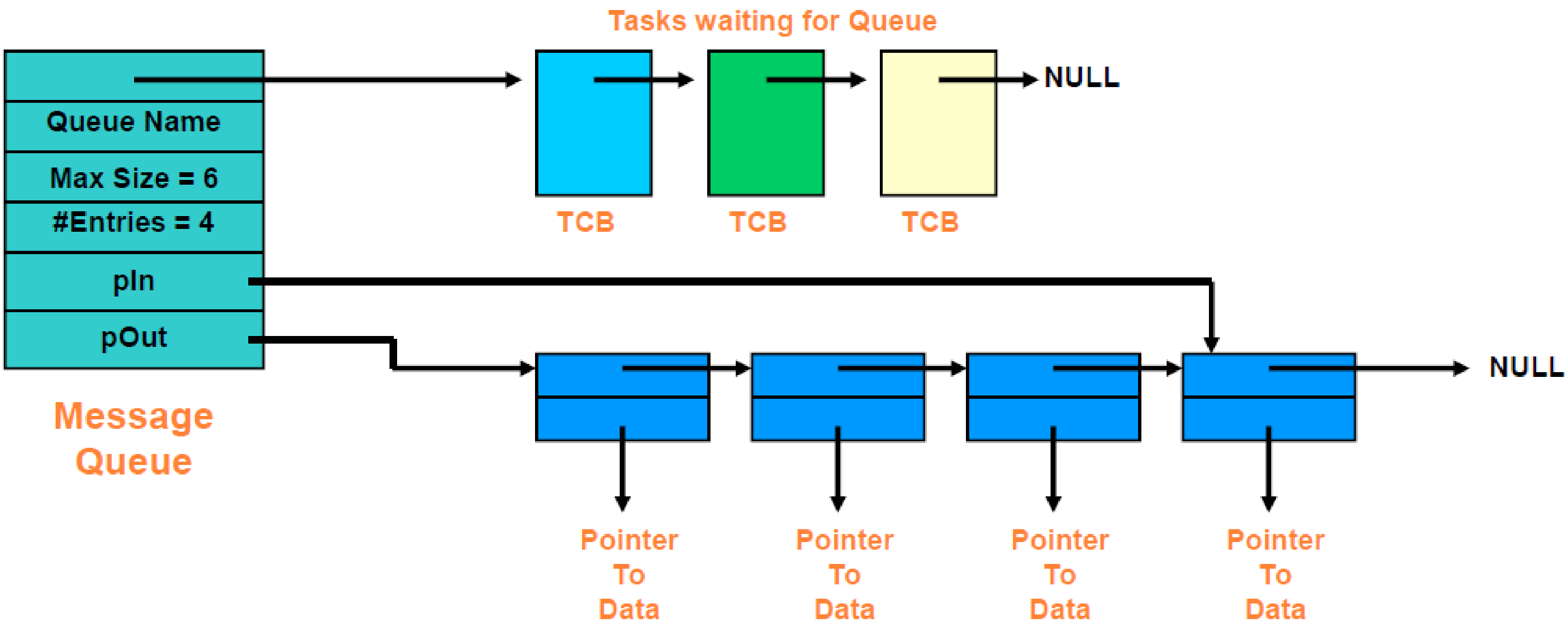
- ❑ Semaphore
- ➢ **Message Queues**
- ❑ Pipes
- ❑ Connections
- ❑ Signals
- ❑ ARINC 653 Ports
- ❑ Buffer and Blackboards
- ❑ Event Registers

Message Queues

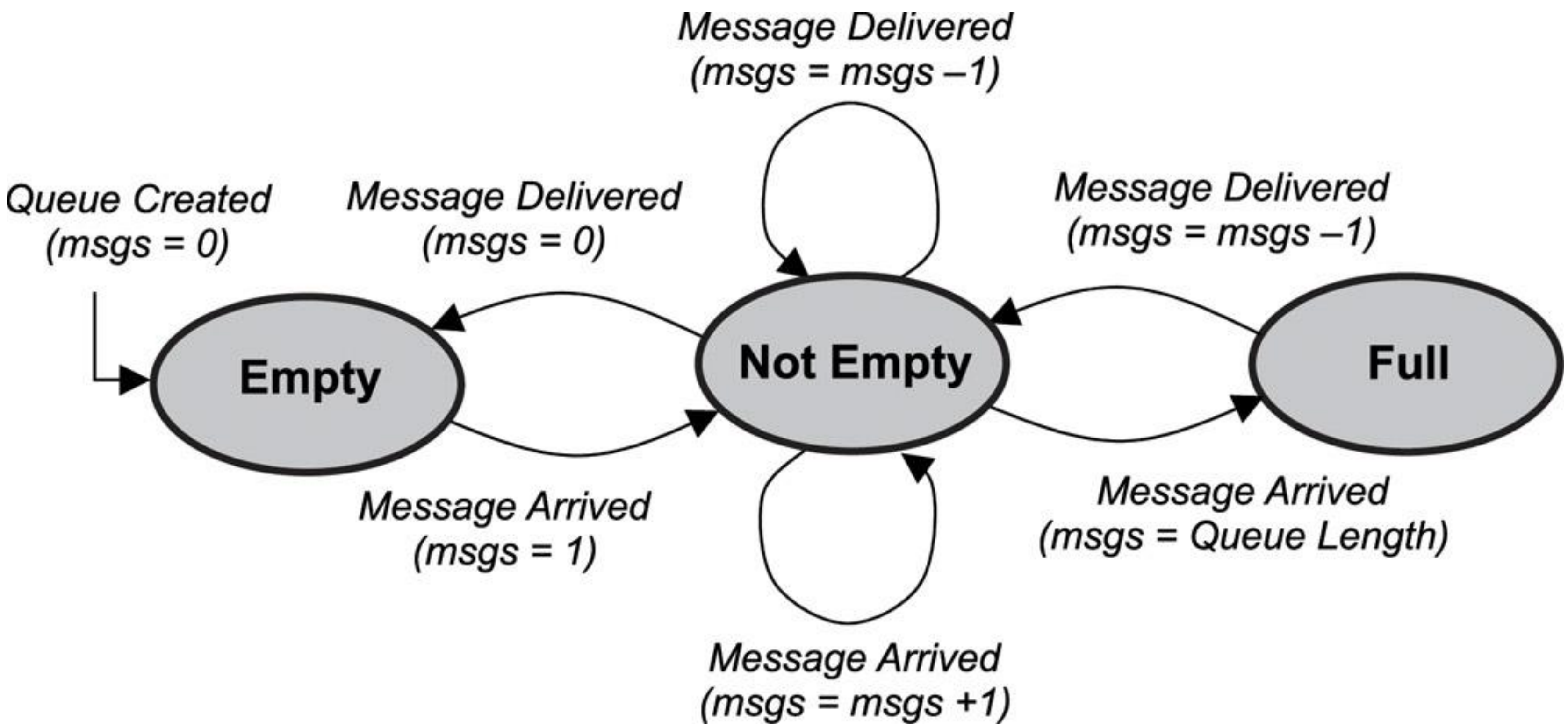
- A message queue is a buffer-like object through which tasks and ISRs send and receive messages to communicate and synchronize with data



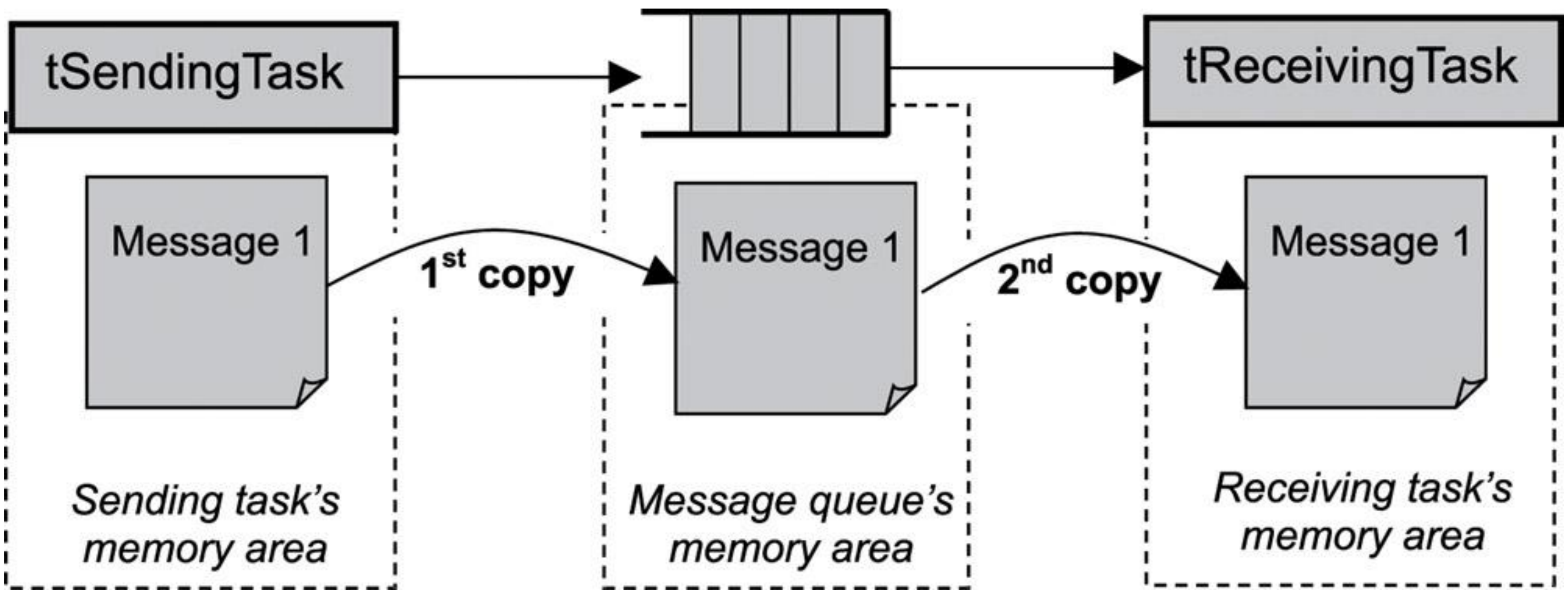
Message Queues - Implementation



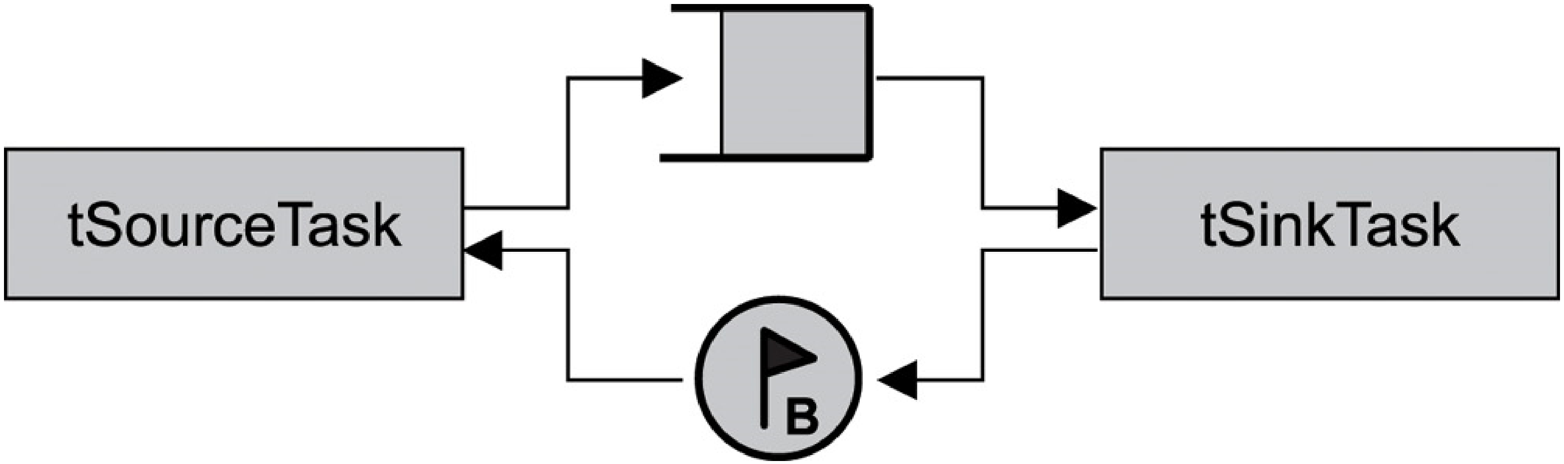
Message Queue States



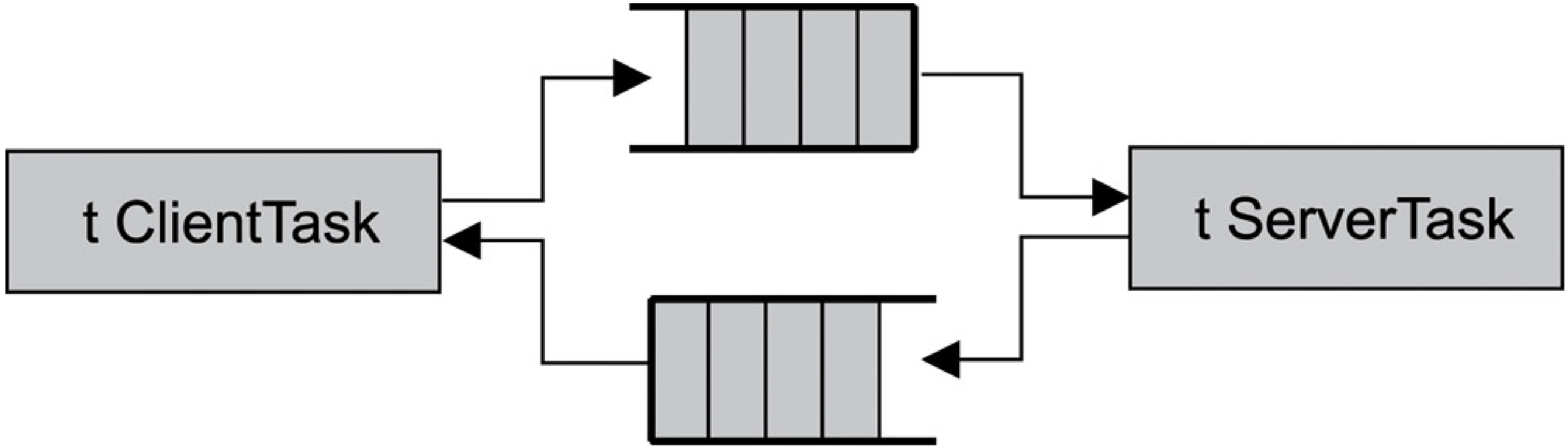
Message copying for sending and receiving messages



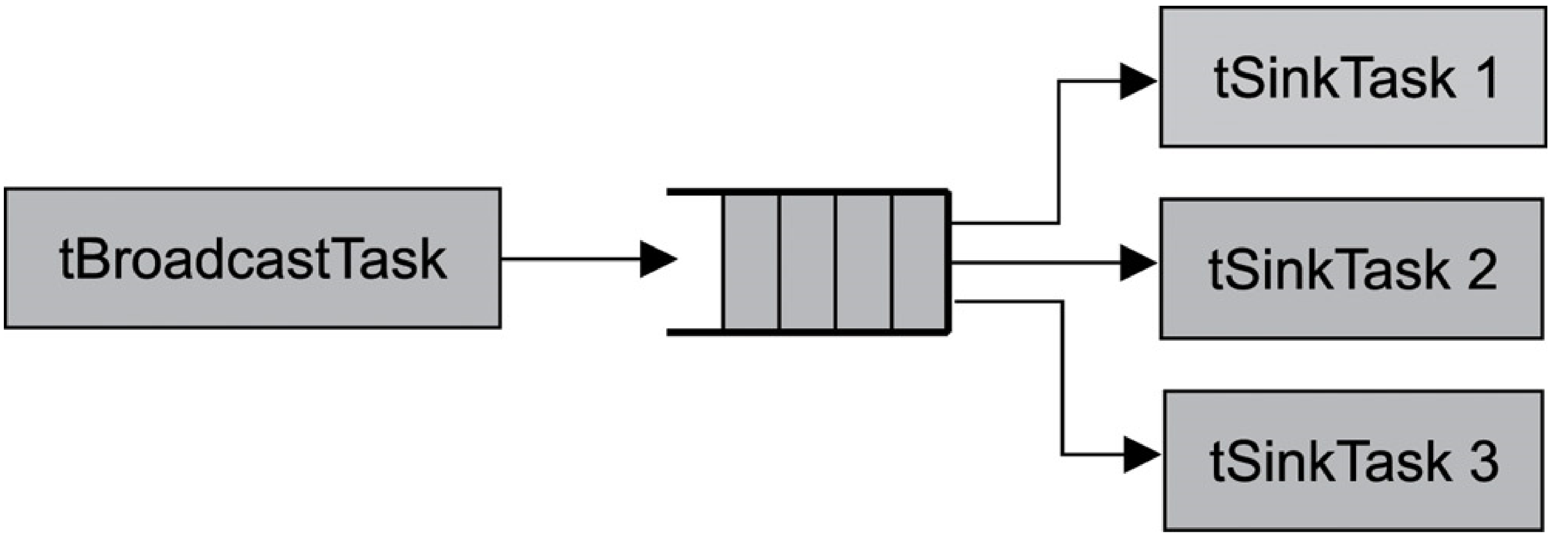
Non-Interlocked, One-Way Data Communication



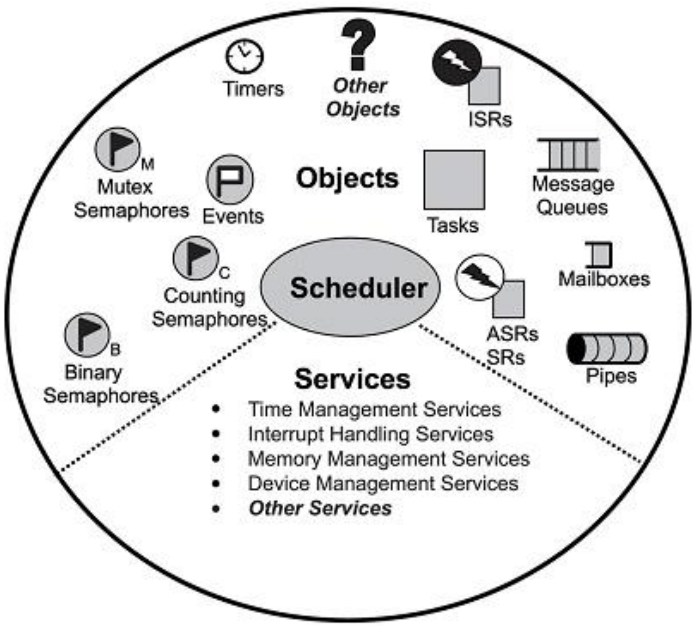
Interlocked, Two-Way Data Communication



Broadcast Communication



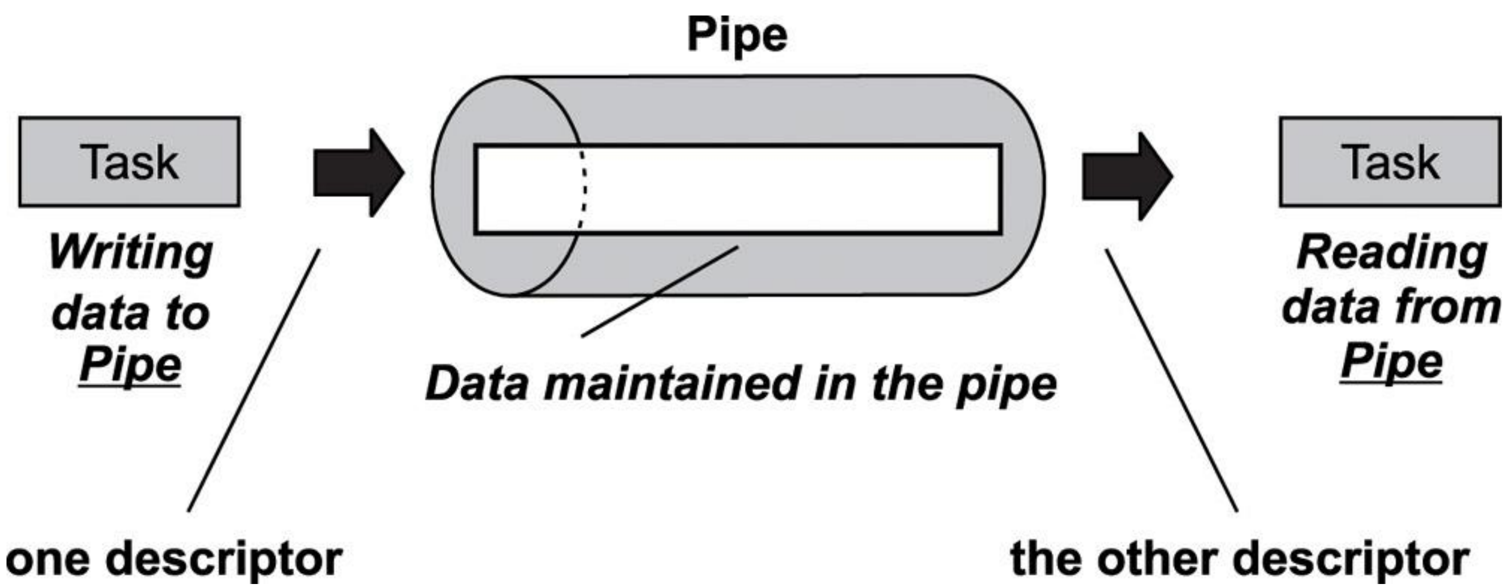
Agenda



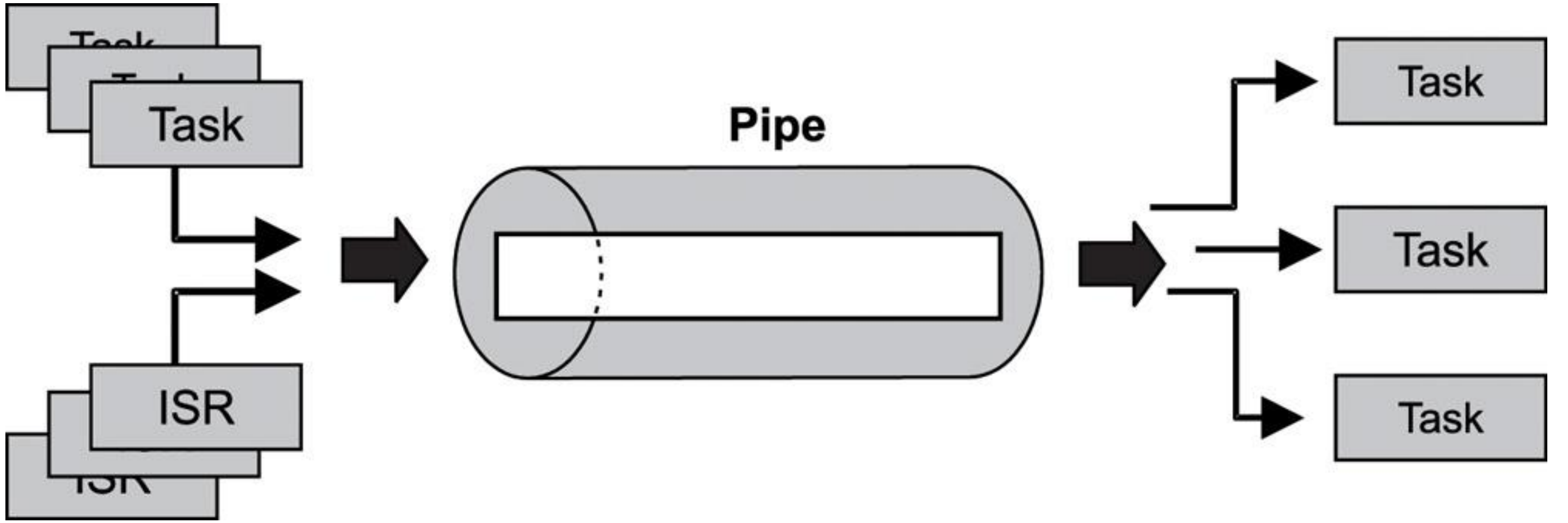
- ❑ Semaphore
- ❑ Message Queues
- ➢ **Pipes**
- ❑ Connections
- ❑ Signals
- ❑ ARINC 653 Ports
- ❑ Buffer and Blackboards
- ❑ Event Registers

Pipes

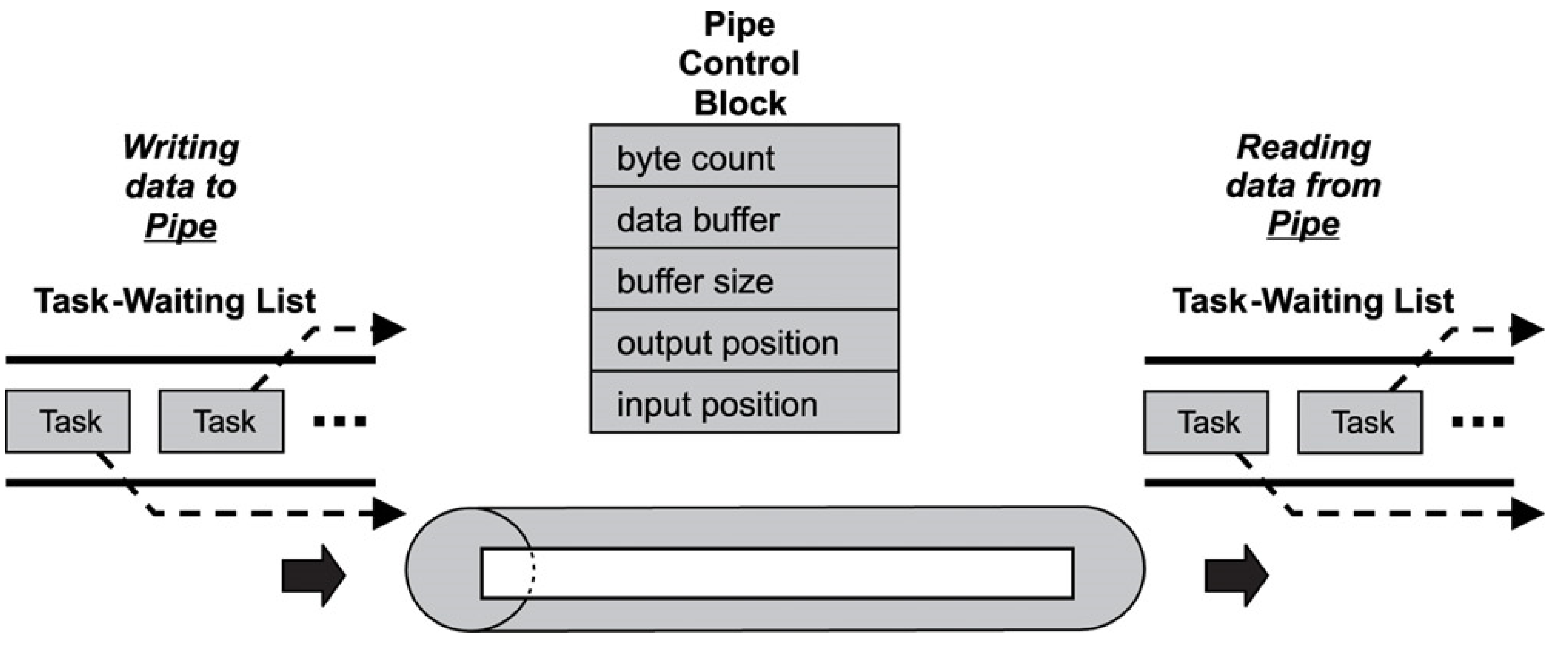
- Pipes are kernel objects that provide unstructured data exchange and facilitate synchronization among tasks
- Data consists of stream of bytes
- Data in a pipe cannot be prioritized



Common pipe operation



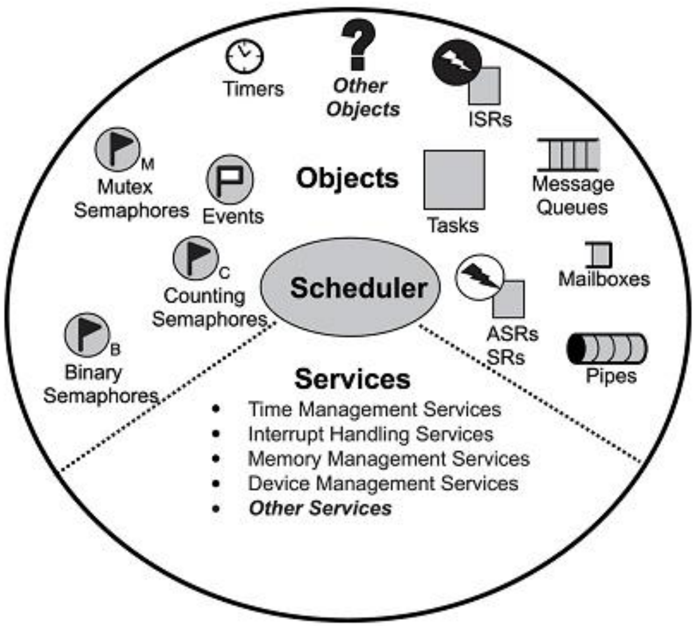
Pipe Control Blocks



Named and Unnamed Pipes

- A named pipe has a name similar to a file name and appears in the file system as if it were a file or a device
- Any task or ISR that needs to use the named pipe can reference it by name
- The unnamed pipe does not have a name and does not appear in the file system

Agenda



- ❑ Semaphore
- ❑ Message Queues
- ❑ Pipes
- ➢ **Connections**
- ❑ Signals
- ❑ ARINC 653 Ports
- ❑ Buffer and Blackboards
- ❑ Event Registers

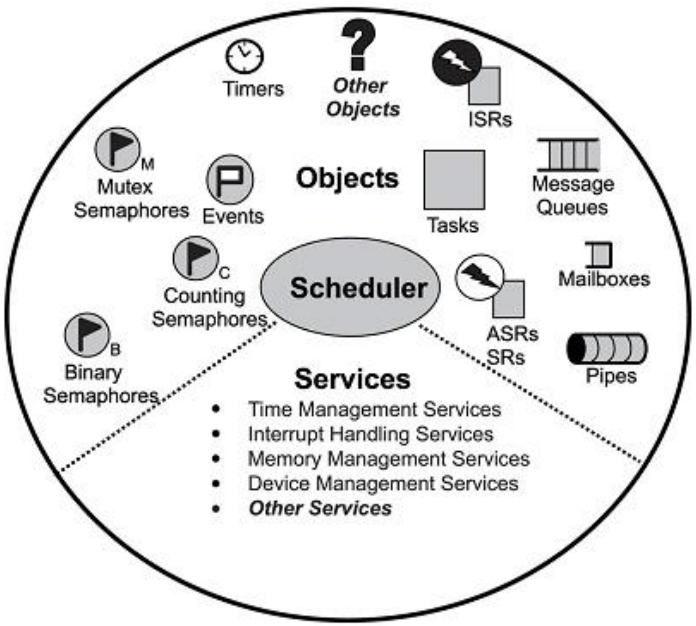
What is a Connection?

- A Connection Object represents one end of a bidirectional communication channel
- A Connection Object represents a permission to communicate with whomever has the other Connection in the pair
- The other Connection in the pair can be in the same or a different Address Space

What is a Connection? (cont)

- A Connection is NOT a pipe
  - A pipe has a buffer –a Connection does not
  - A pipe is unidirectional –a Connection is bi-directional
- Data is copied directly from the sender to the receiver
- The Connection itself has no data storage

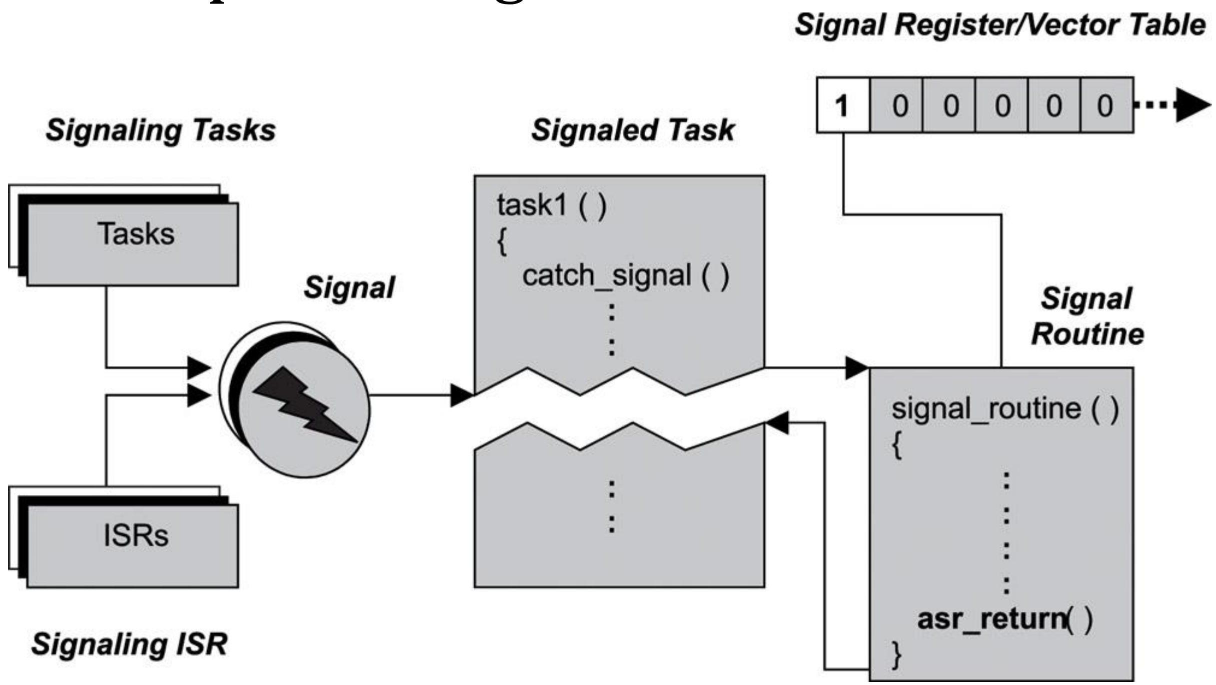
Agenda



- ❑ Semaphore
- ❑ Message Queues
- ❑ Pipes
- ❑ Connections
- ➢ **Signals**
- ❑ ARINC 653 Ports
- ❑ Buffer and Blackboards
- ❑ Event Registers

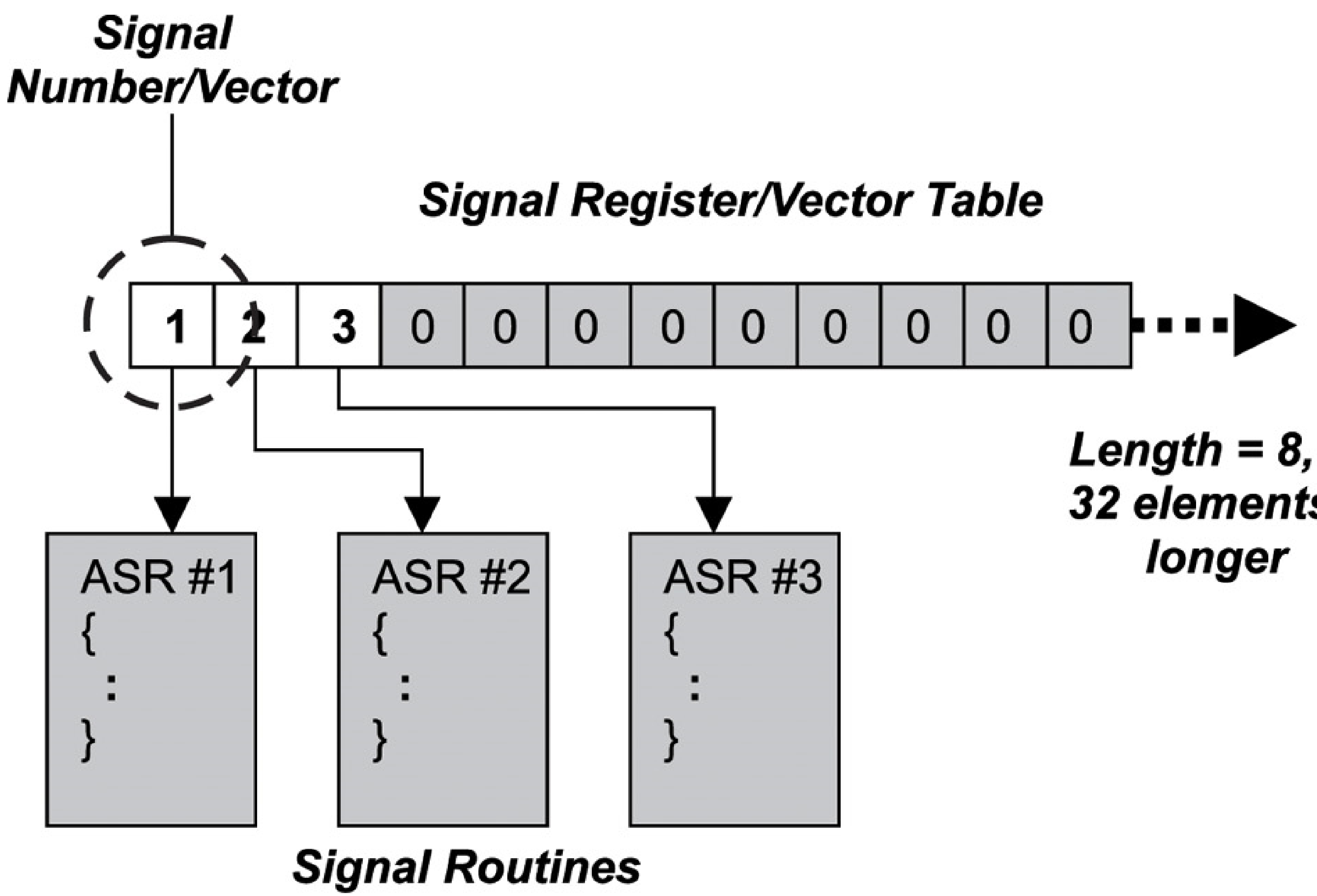
Signals

- A signal is a software interrupt that is generated when an event has occurred
- It diverts the signal receiver from its normal execution path and triggers the associated asynchronous processing.

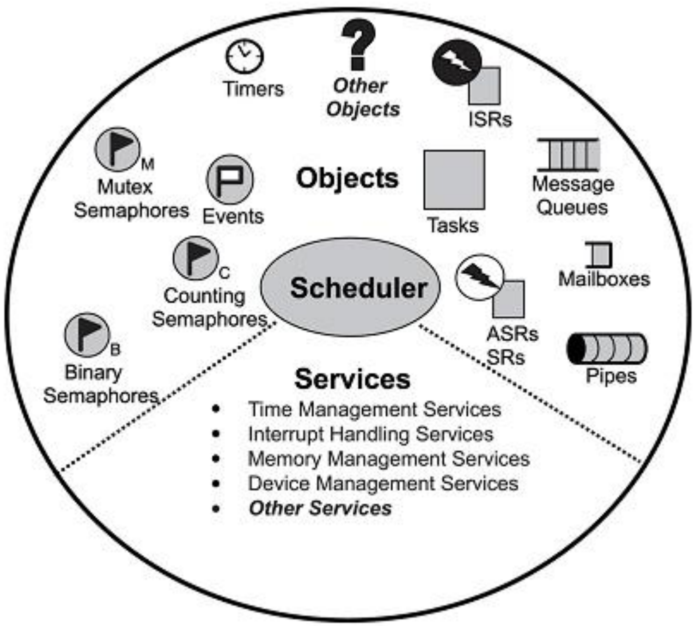


Signal Vector Table

- Each element in the vector table is a pointer or offset to an ASR



Agenda



- ❑ Semaphore
- ❑ Message Queues
- ❑ Pipes
- ❑ Connections
- ❑ Signals
- ➢ **ARINC 653 Ports**
- ❑ Buffer and Blackboards
- ❑ Event Registers

APEX Channels

- A channel defines the following:
  - Logical link between one source port and one or more destination ports
  - Mode of transfer of the messages from the source to the destination
  - Characteristics of the messages to be sent

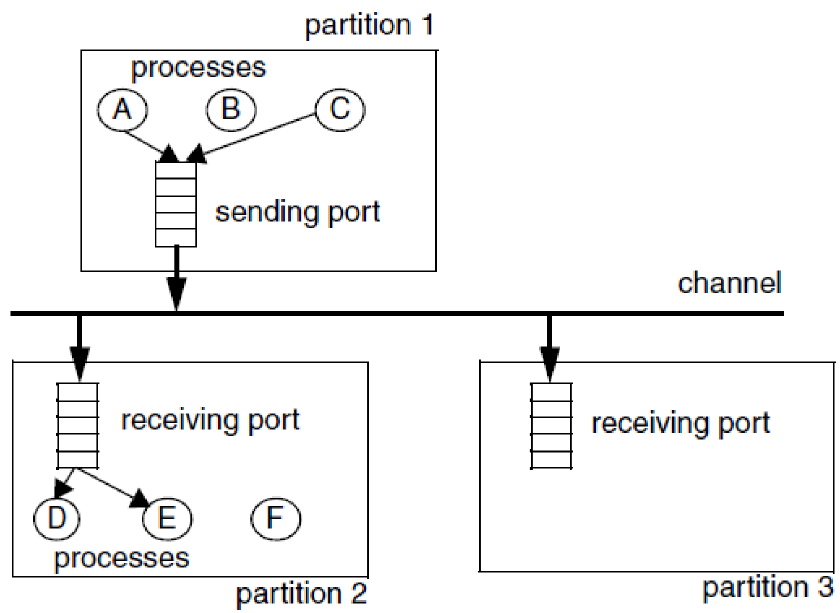
- Each channel can be configured to operate in a specific mode
  - sampling mode
  - queuing mode

Sampling Mode

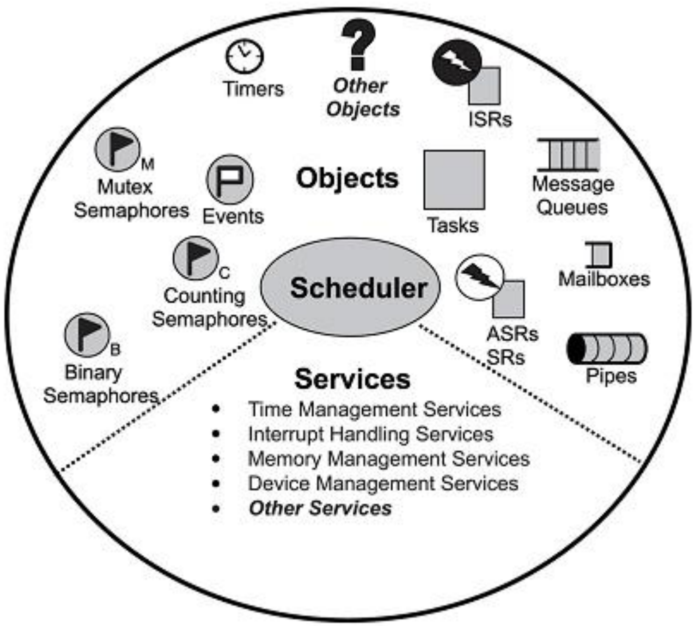
- Messages typically carry similar, but updated, data
- No queuing is performed
- A message remains in the source port until it is sent or overwritten.
- The refresh rate indicates the maximum acceptable age of a valid message, from the time it was received at the port.

Queueing Mode

- Each new instance of a message may contain uniquely different data.
- Messages are queued in the source port until they are sent
- Messages are stored in the receiver port until a process reads them



Agenda

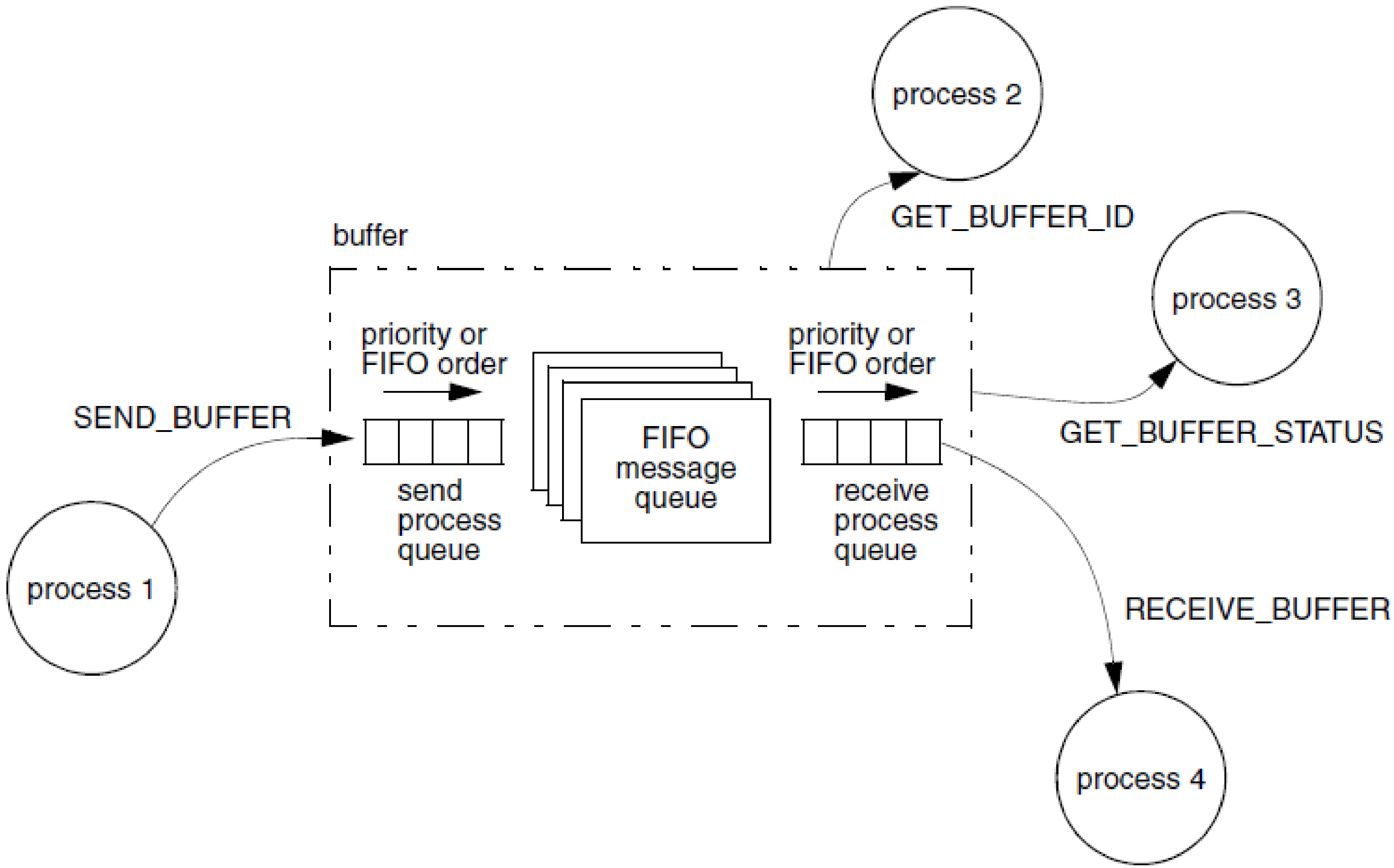


- ❑ Semaphore
- ❑ Message Queues
- ❑ Pipes
- ❑ Connections
- ❑ Signals
- ❑ ARINC 653 Ports
- ➢ **Buffer and Blackboards**
- ❑ Event Registers

Buffers

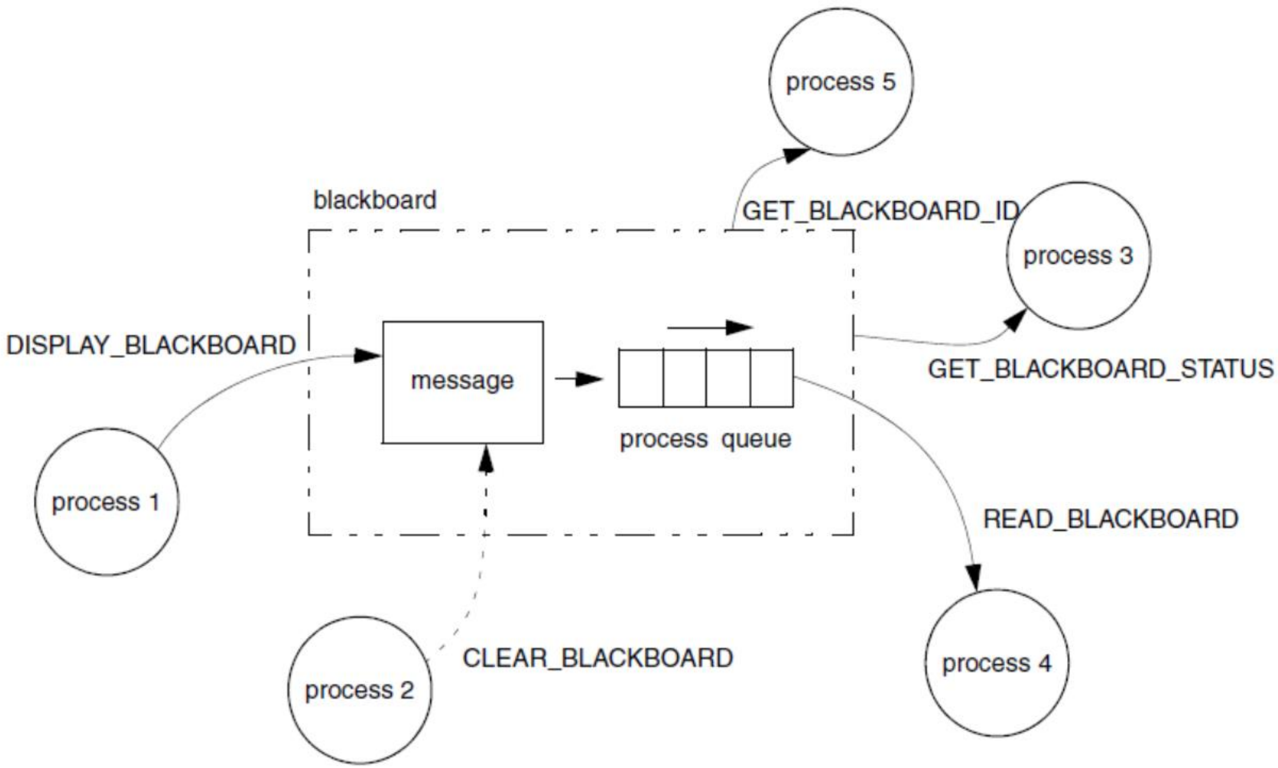
- Buffers store multiple messages in message queues and no messages are lost
- APEX buffers let processes communicate with each other within a partition
- A kind of message queue
- Asynchronous communication

Processes Using a Buffer to Communicate

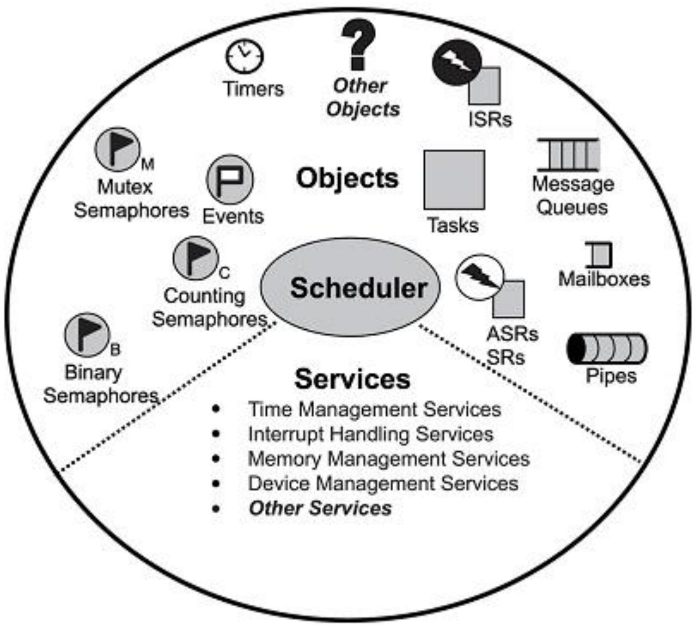


Blackboards

- Blackboards support a single message type between multiple source and destination processes



Agenda

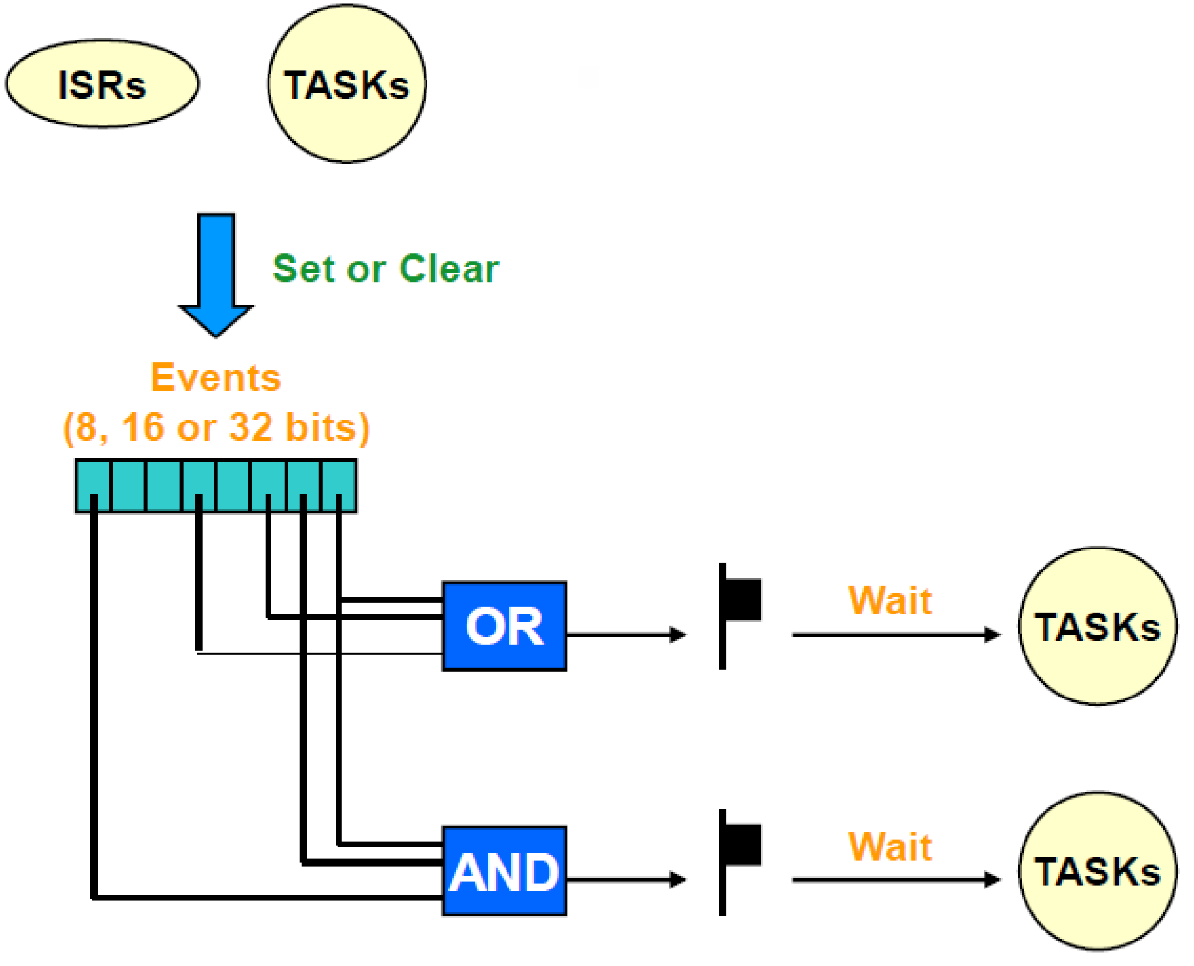


- ❑ Semaphore
- ❑ Message Queues
- ❑ Pipes
- ❑ Connections
- ❑ Signals
- ❑ ARINC 653 Ports
- ❑ Buffer and Blackboards
- ➢ **Event Registers**

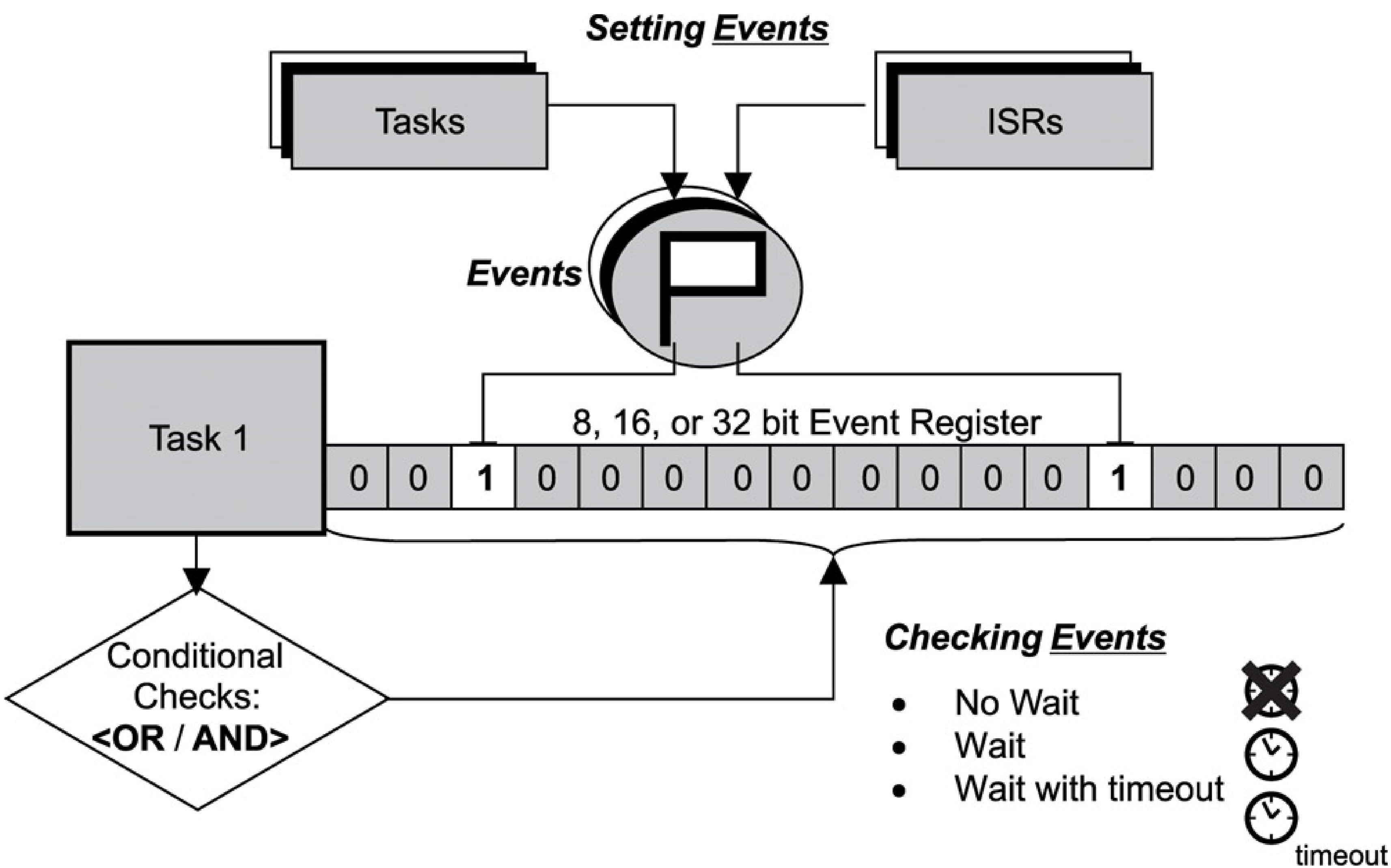
Event Registers

- Synchronization of tasks with the occurrence of multiple events
- Events are grouped
  - 8, 16 or 32 bits per group
- Types of synchronization:
  - Disjunctive (OR) : Any event occurred
  - Conjunctive (AND): All events occurred
- Task(s) or ISR(s) can either Set or Clear event flags
- Only tasks can Wait for events

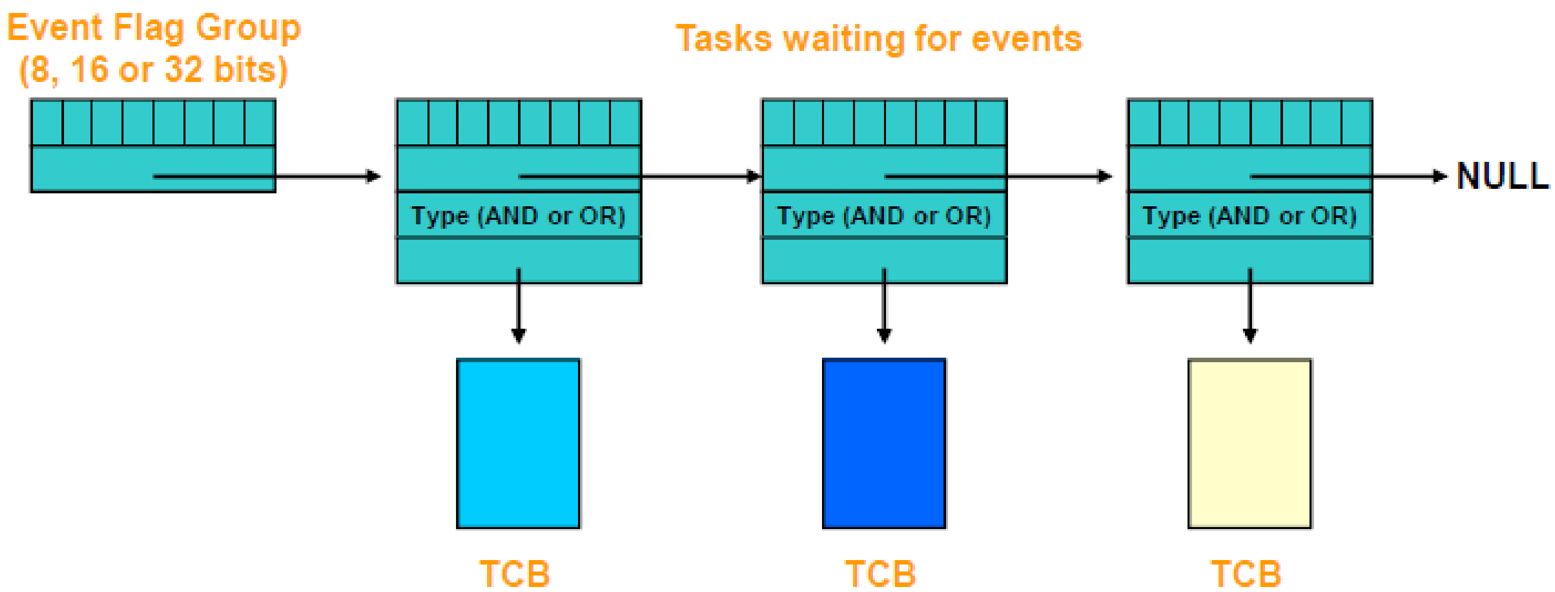
Event Registers



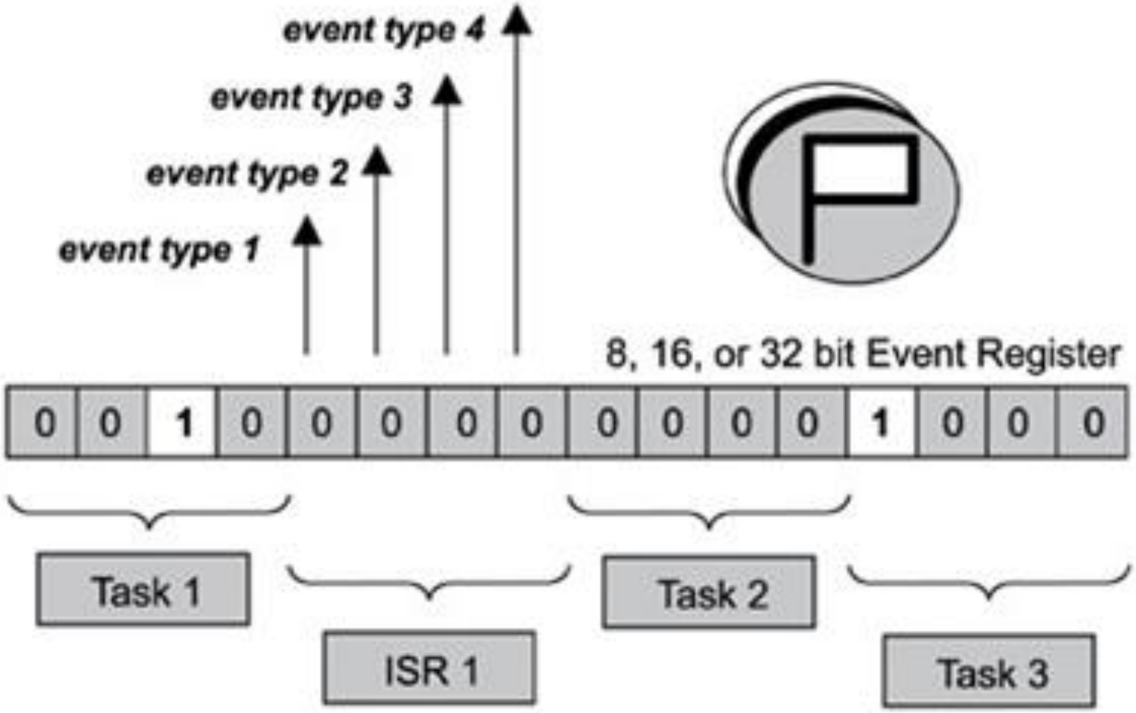
Event Registers



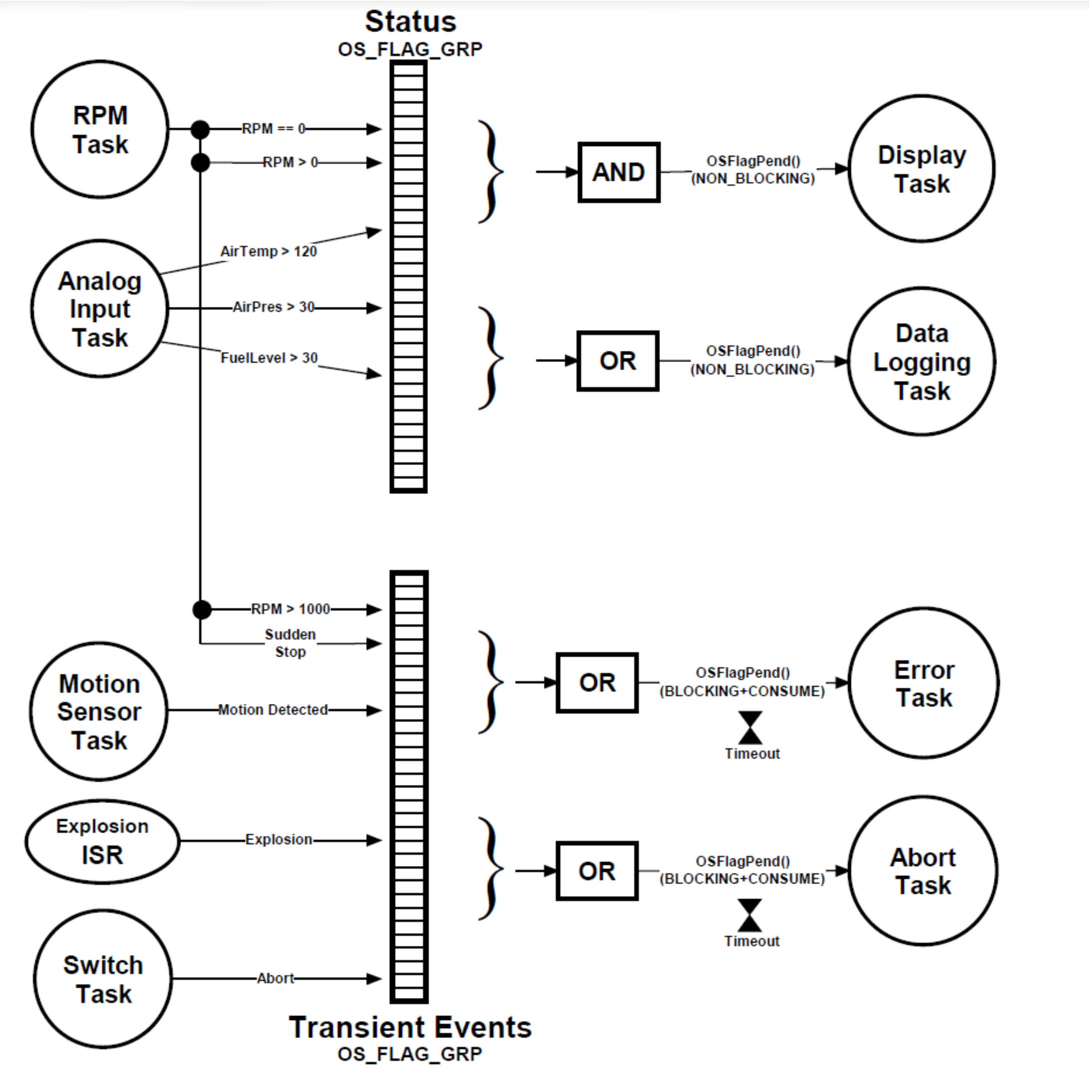
Event Registers - Implementation



Identifying an event source



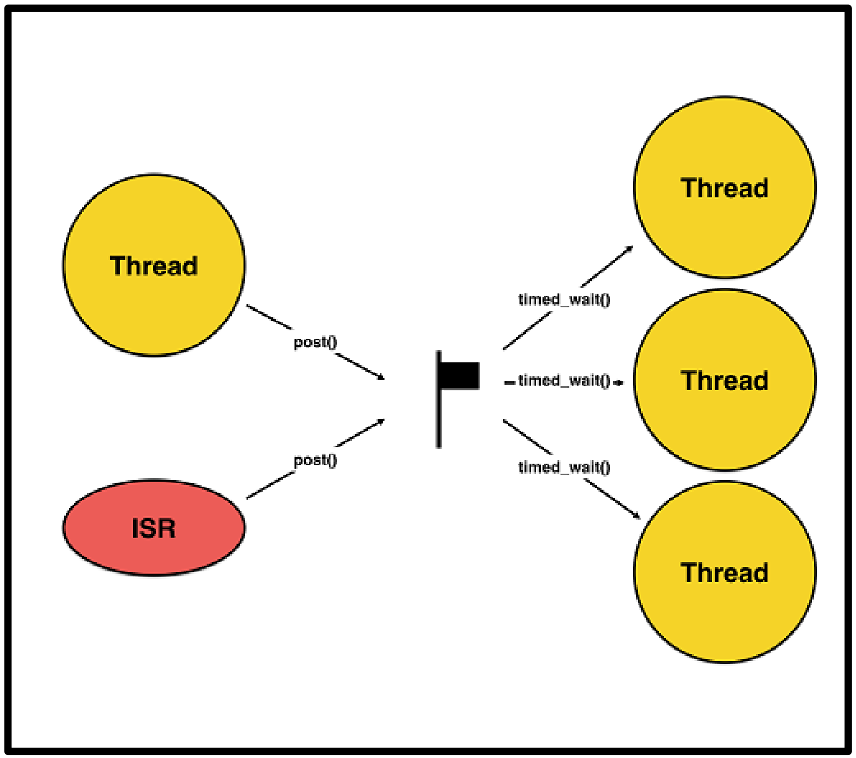
Event Registers - Example



Chapter 2

Synchronization And Communication Methods

Agenda



- ➢ **Synchronization**
- ❑ Communication
- ❑ Resource Synchronization
- ❑ Practical Design Patterns
- ❑ Deadlocks
- ❑ Priority Inversion

Synchronization

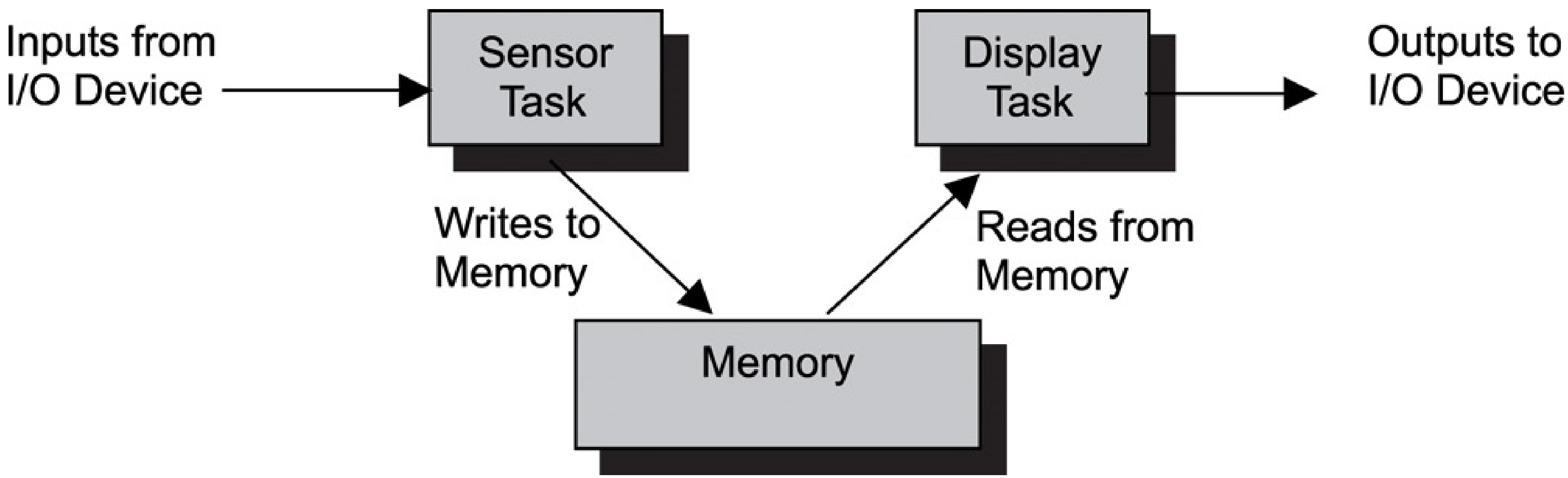
- Synchronization is classified into two categories:
  - Resource synchronization determines whether access to a shared resource is safe
  - Activity synchronization determines whether the execution of a multithreaded program has reached a certain state

Resource Synchronization

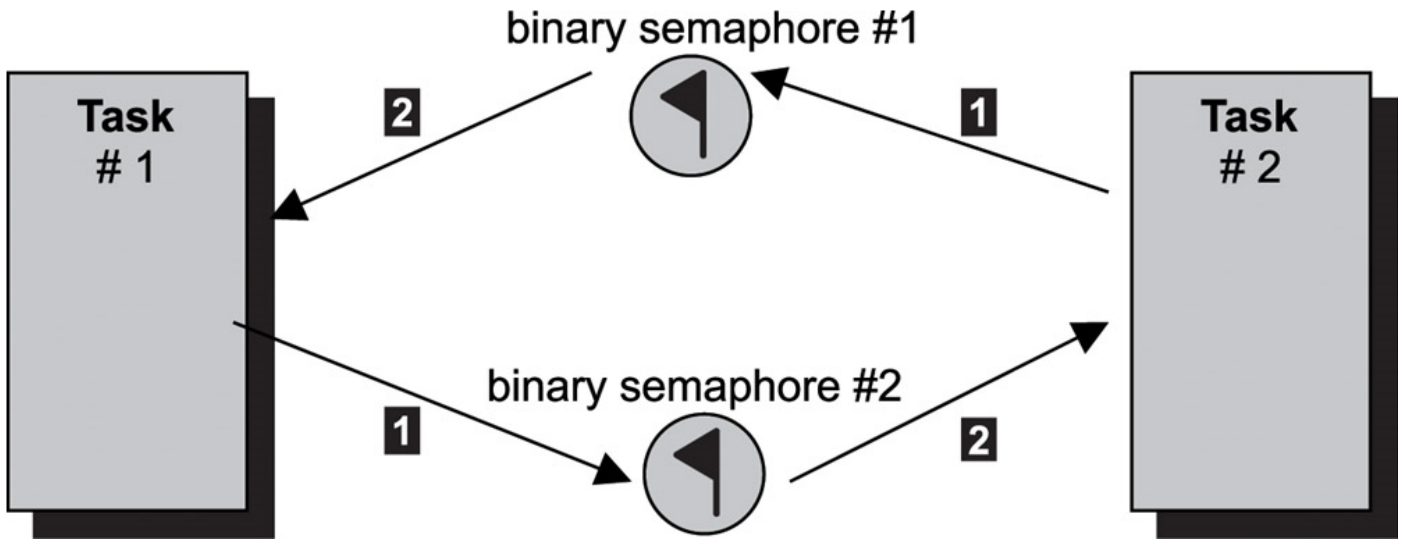
- Access by multiple tasks must be synchronized to maintain the integrity of a shared resource
- Mutual exclusion is a provision by which only one task at a time can access a shared resource
- A critical section is the section of code from which the shared resource is accessed.

Resource Synchronization

- Multiple tasks accessing shared memory



Rendezvous synchronization

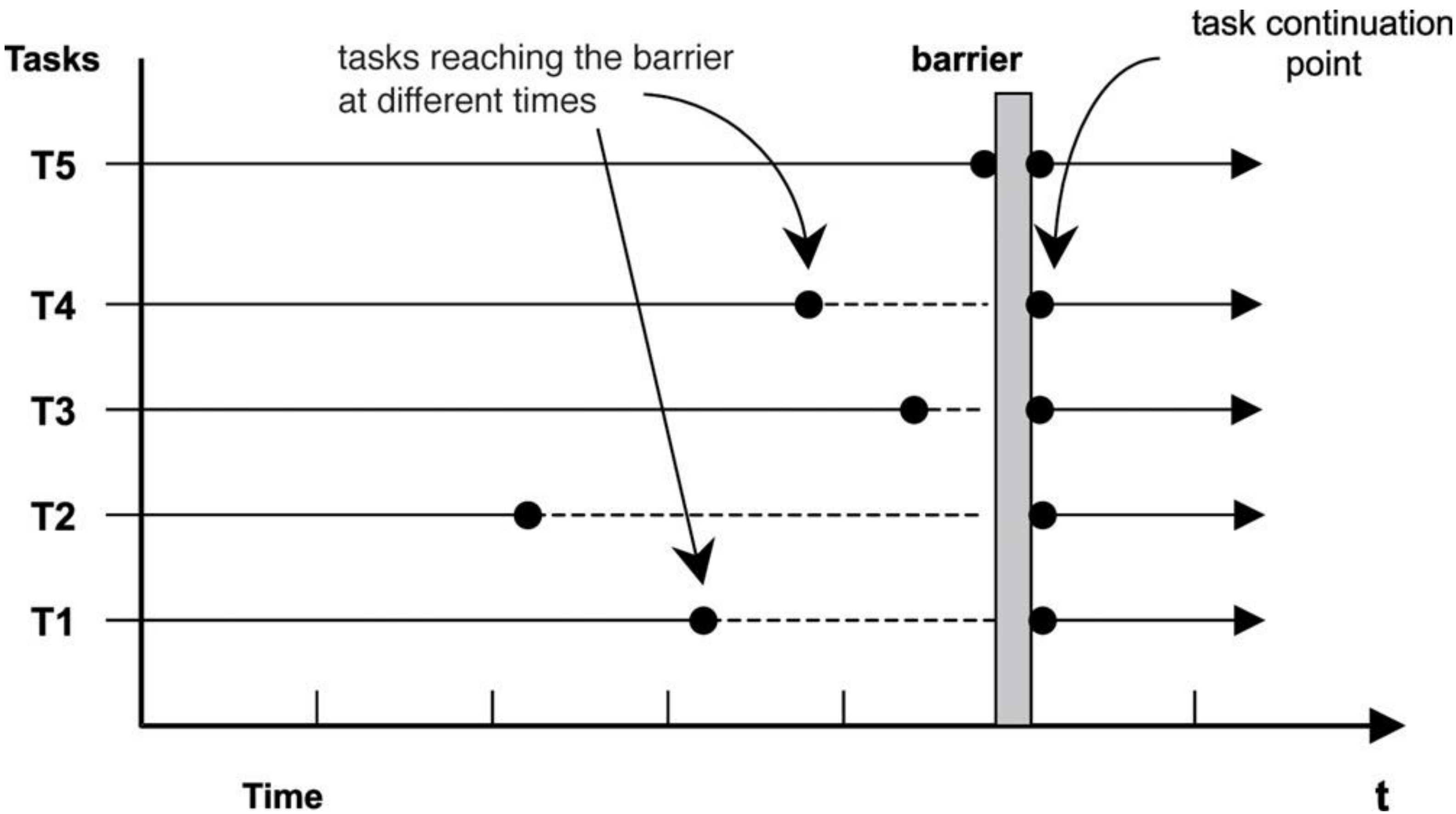


- Both binary semaphores are initialized to 0
- When task #1 reaches the rendezvous, it gives semaphore #2, and then it gets on semaphore #1
- When task #2 reaches the rendezvous, it gives semaphore #1, and then it gets on semaphore #2

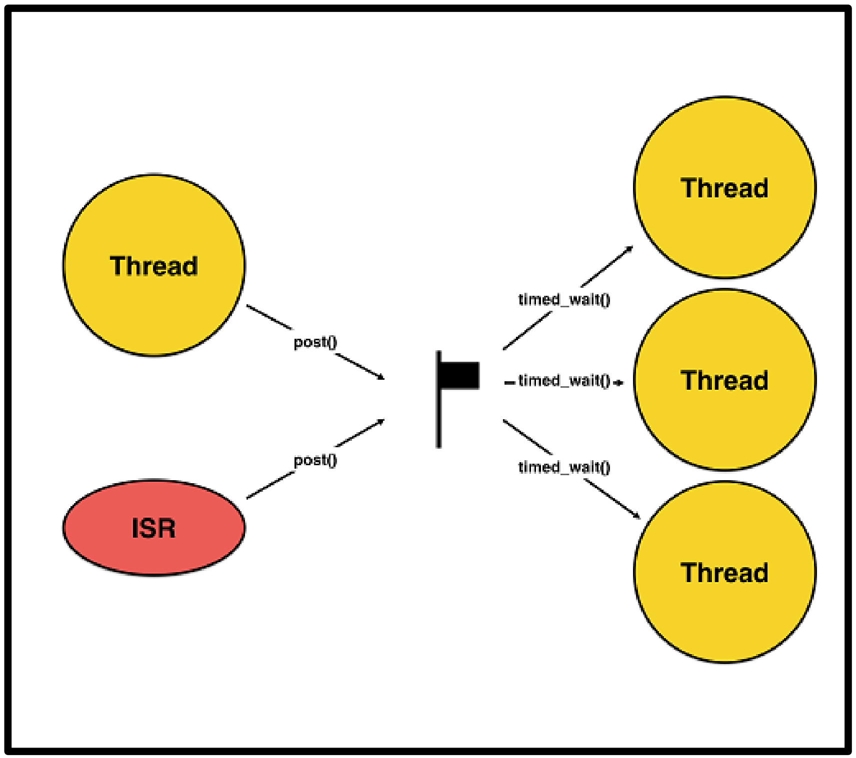
Activity Synchronization

- A task must synchronize its activity with other tasks to execute a multithreaded program properly
- Activity synchronization ensures that the correct execution order among cooperating tasks is used
- Activity synchronization can be either synchronous or asynchronous.

Barrier synchronization



Agenda



- ❑ Synchronization
- ➢ **Communication**
- ❑ Resource Synchronization
- ❑ Practical Design Patterns
- ❑ Deadlocks
- ❑ Priority Inversion

Communication

- Tasks communicate with one another
  - To pass information
  - To coordinate their activities in a multithreaded embedded application

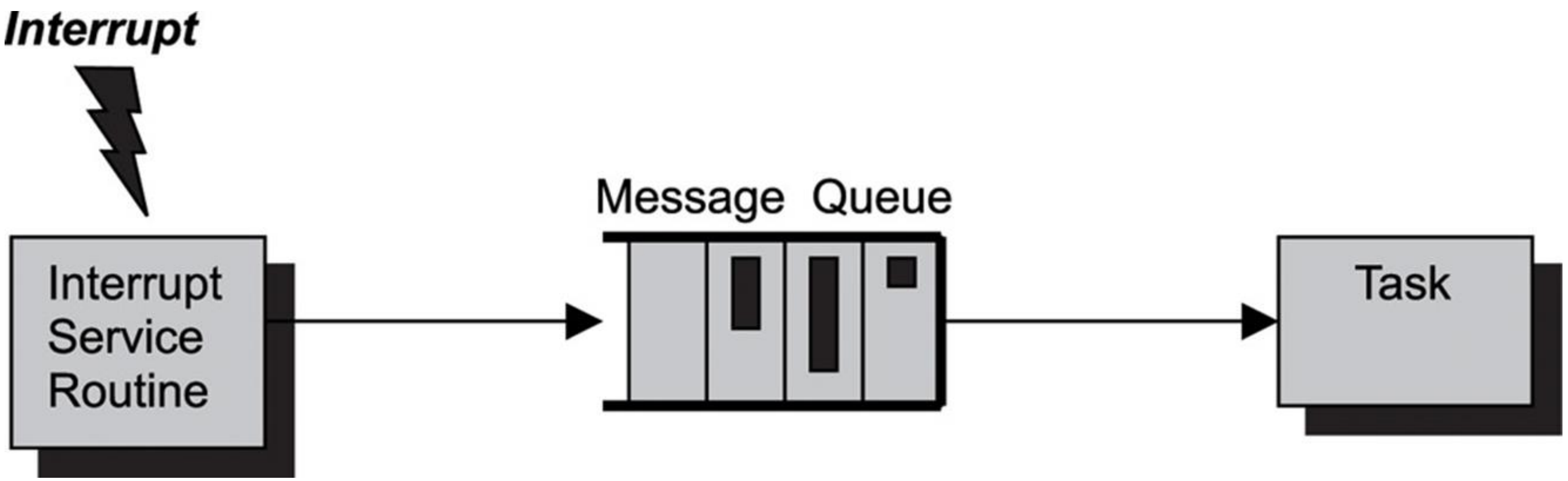
- Communication can be signal-centric, data-centric, or both

Communication purposes

- Communication has several purposes, including the following:
  - Transferring data from one task to another,
  - Signaling the occurrences of events between tasks,
  - Allowing one task to control the execution of other tasks,
  - Synchronizing activities
  - Implementing custom synchronization protocols for resource sharing

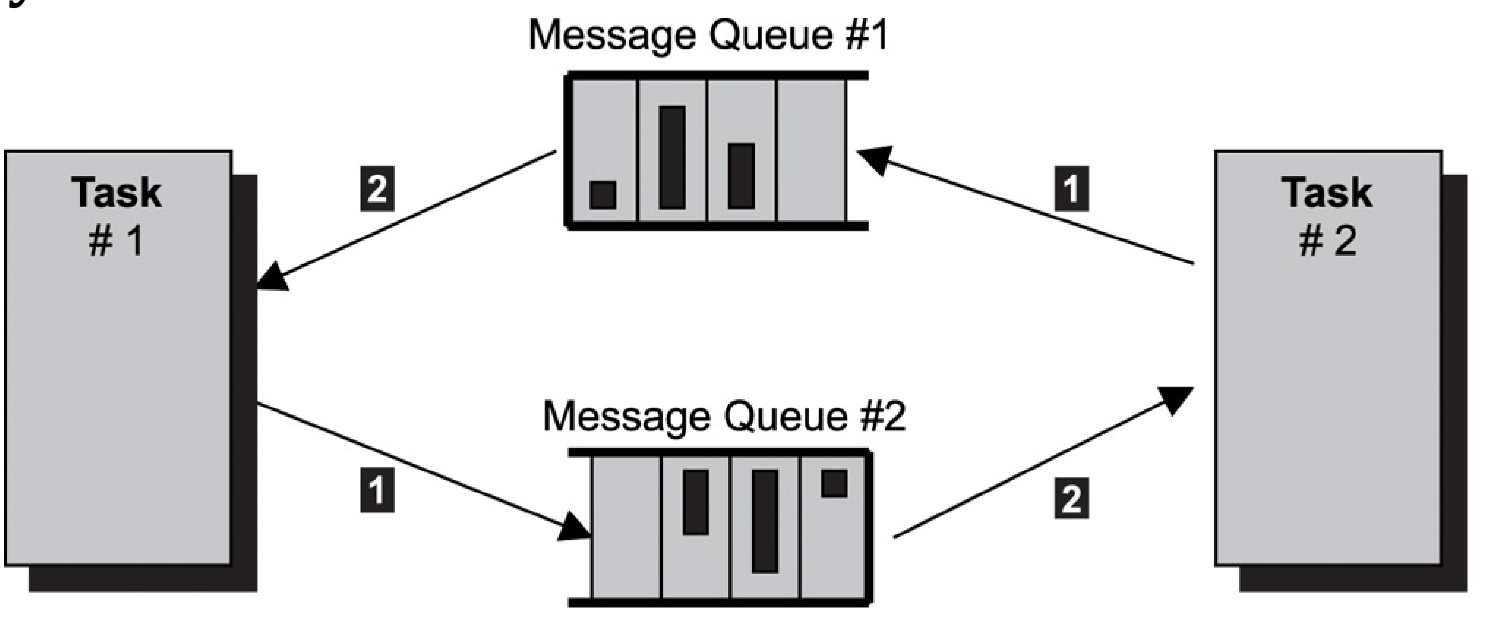
Loosely coupled communication

- When communication involves data flow and is unidirectional, this communication model is called loosely coupled communication
- Asynchronous

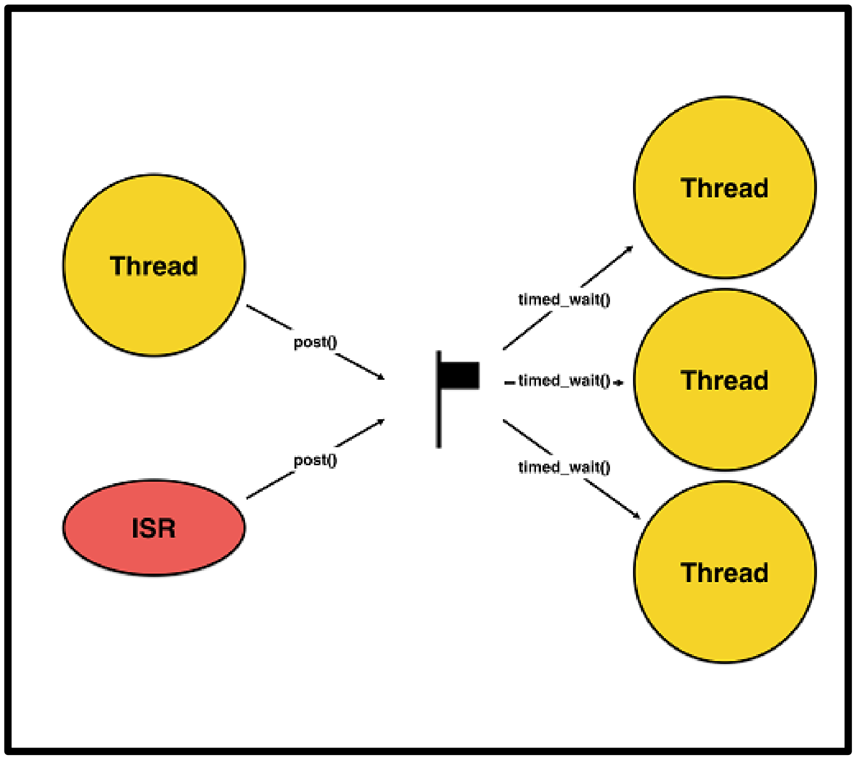


Tightly coupled communication

- Data movement is bidirectional
- The data producer synchronously waits for a response to its data transfer before resuming execution
- Synchronous



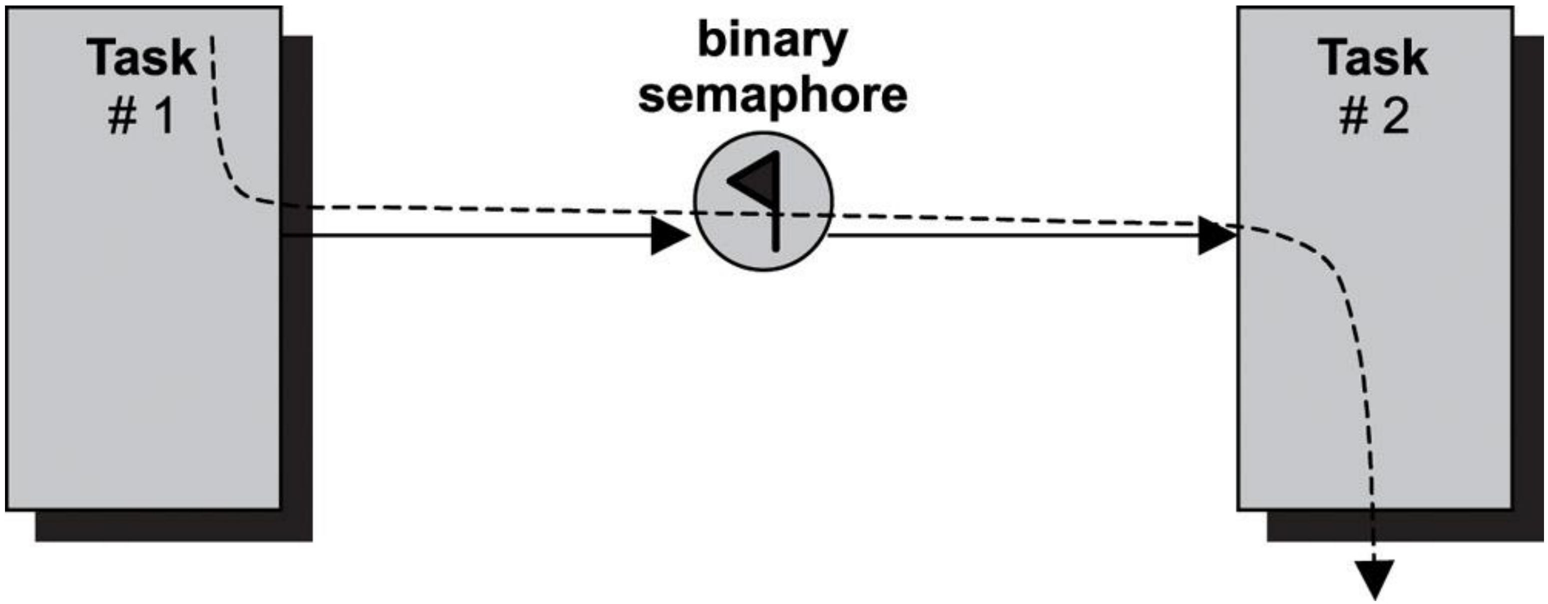
Agenda



- ❑ Synchronization
- ❑ Communication
- ❑ Resource Synchronization
- ➢ **Practical Design Patterns**
- ❑ Deadlocks
- ❑ Priority Inversion

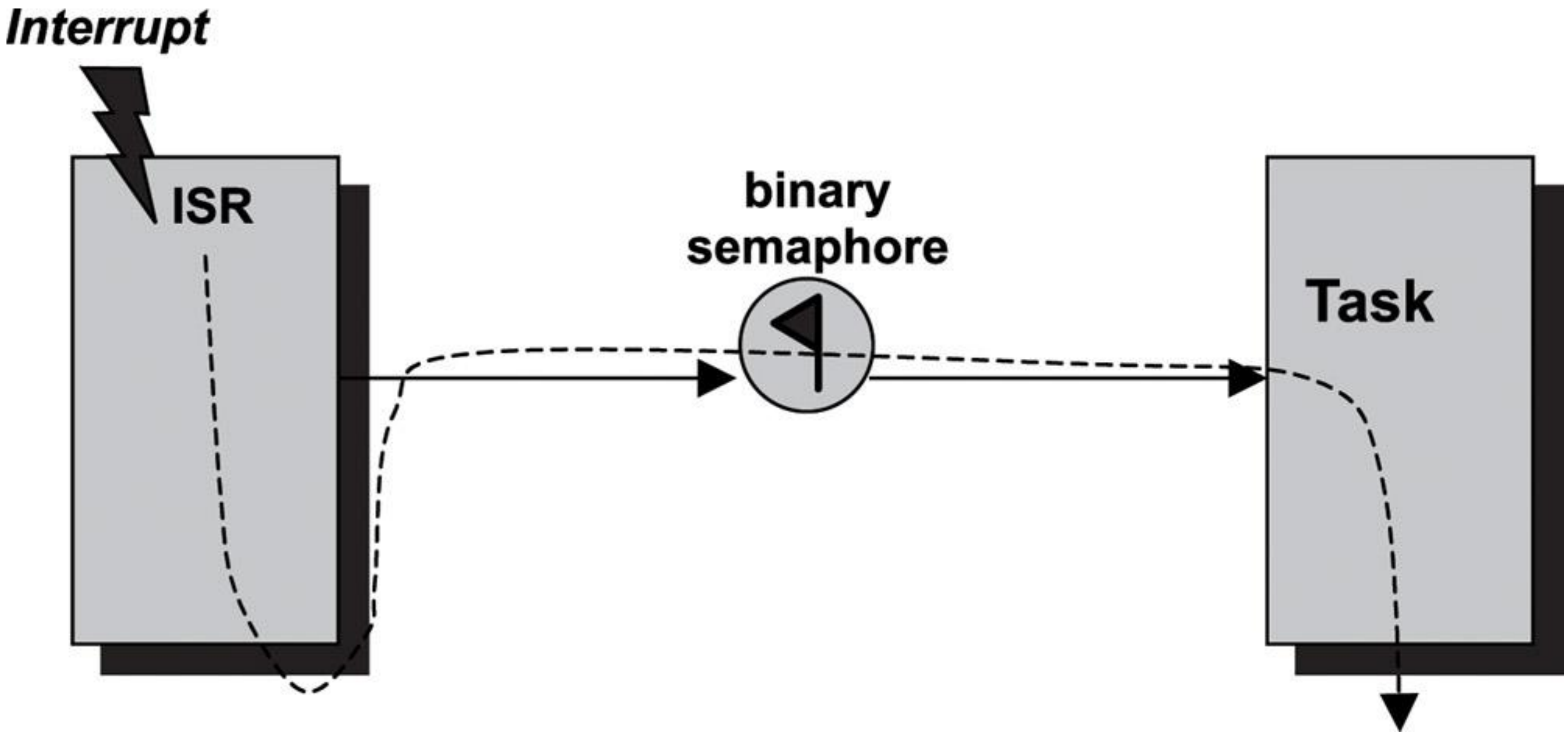
Synchronous Activity Synchronization

- Task-to-Task Synchronization Using Binary Semaphores



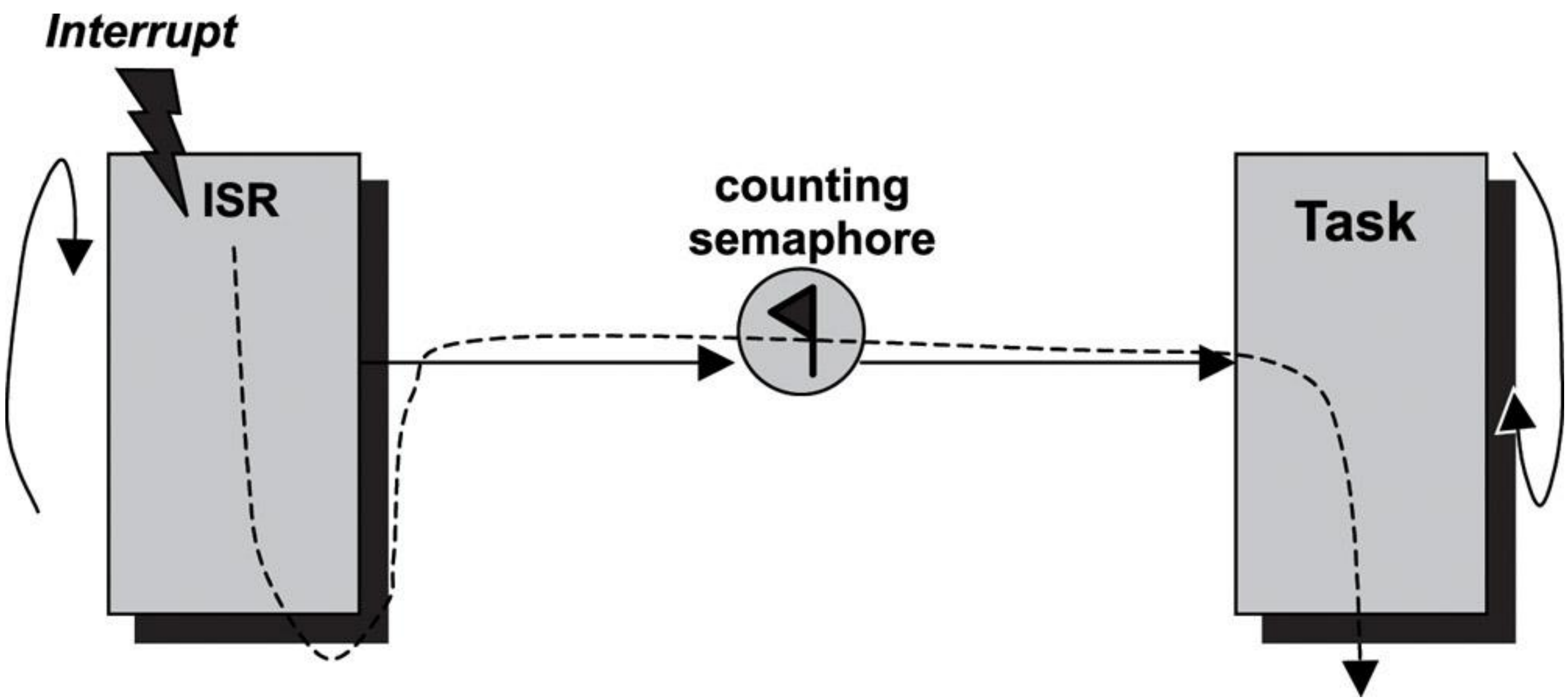
Synchronous Activity Synchronization

- ISR-to-Task Synchronization Using Binary Semaphores



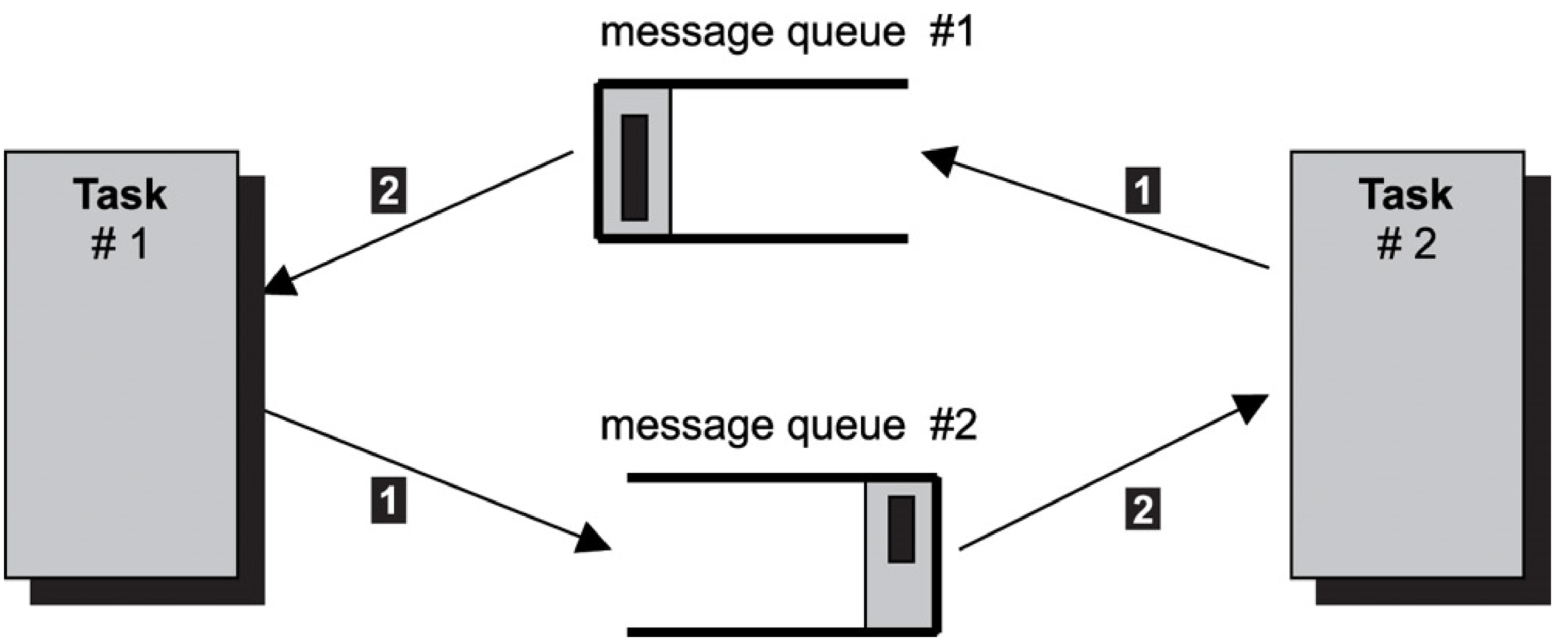
Synchronous Activity Synchronization

- ISR-to-Task Synchronization Using Counting Semaphores



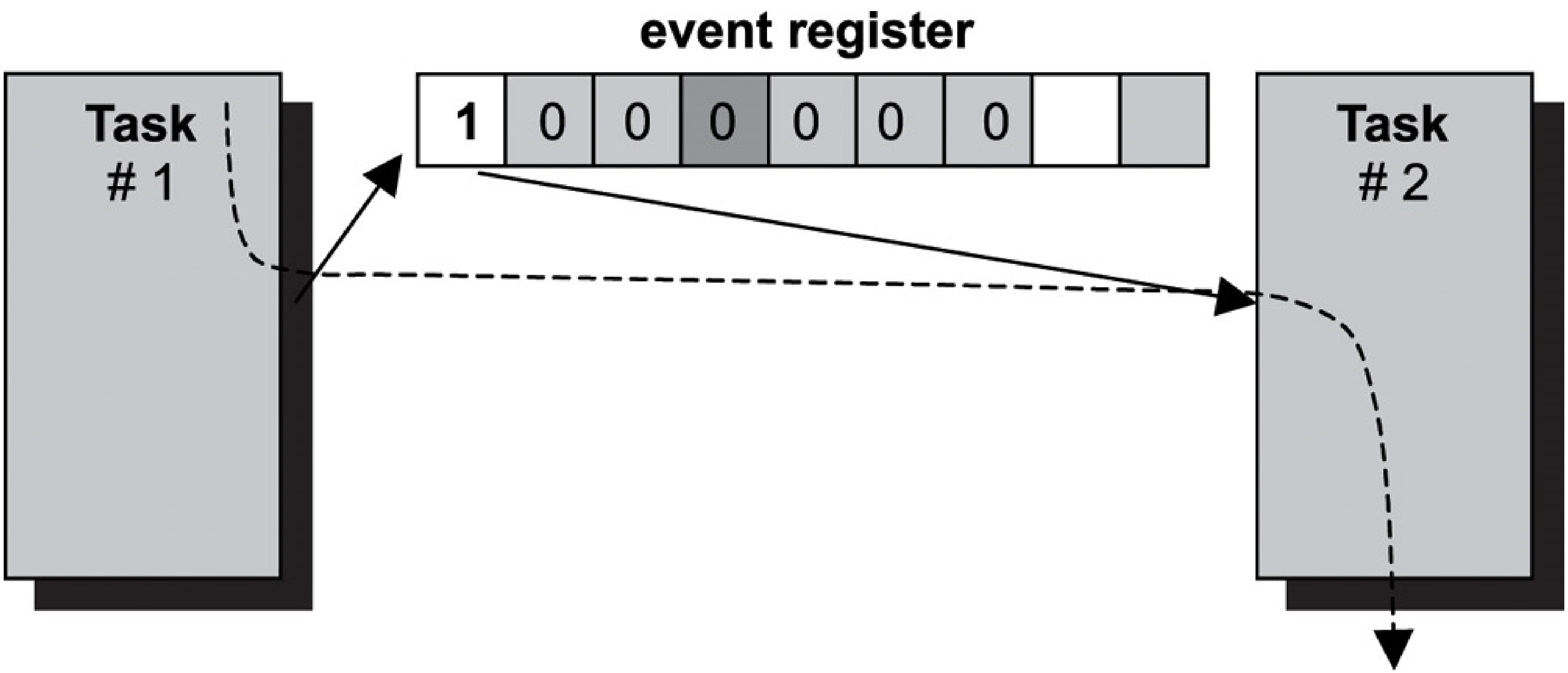
Synchronous Activity Synchronization

- Simple Rendezvous with Data Passing



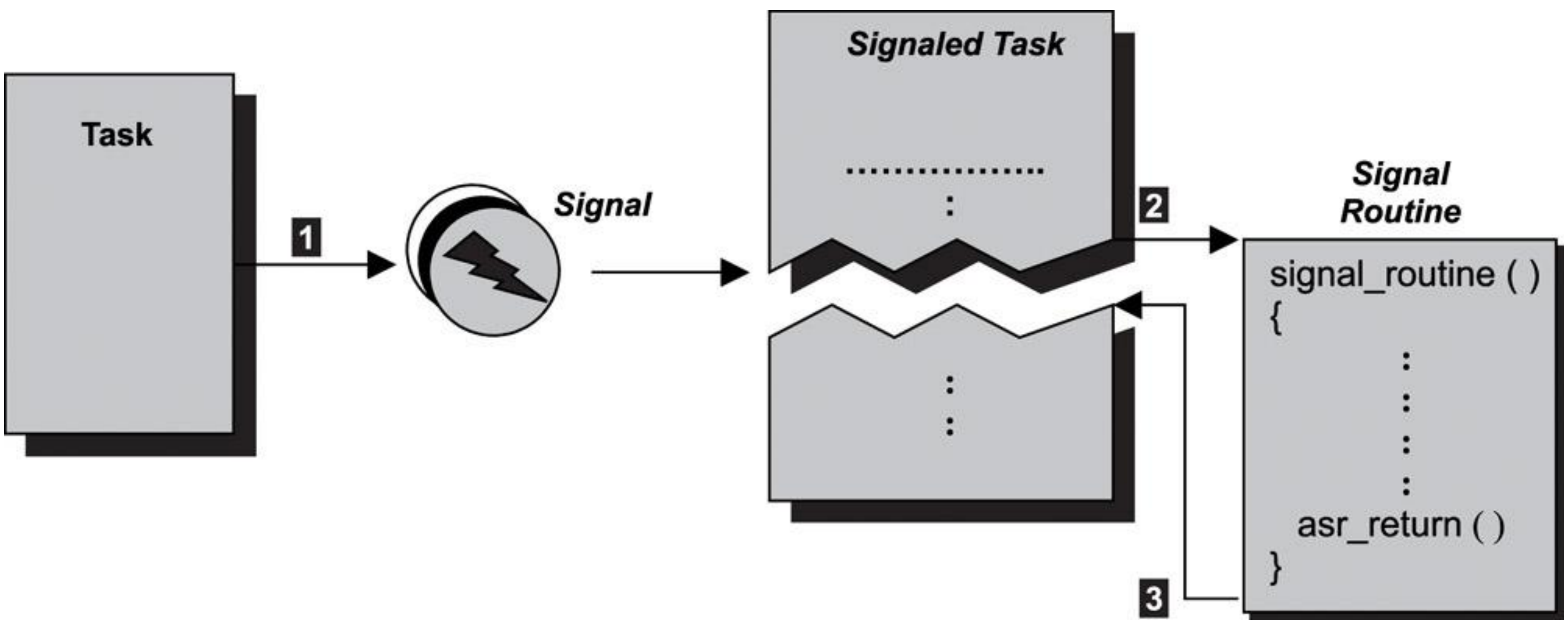
Synchronous Activity Synchronization

- Task-to-Task Synchronization Using Event Registers



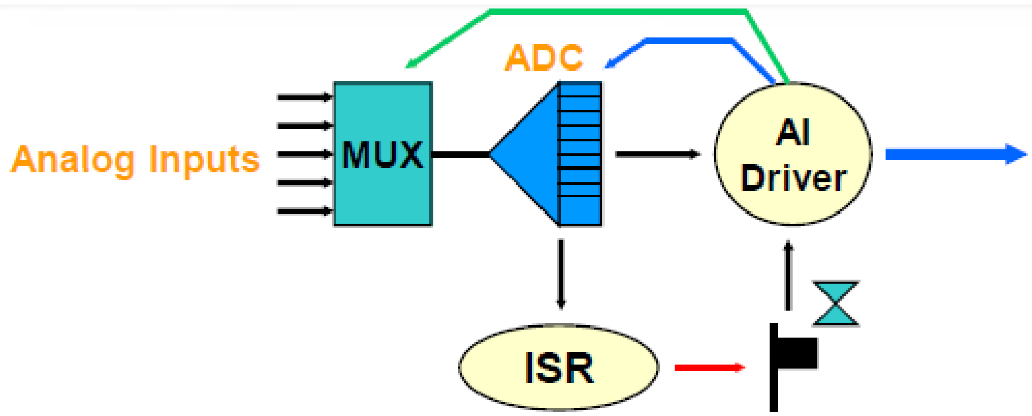
Asynchronous Event Notification Using Signals

- Using signals for urgent data communication



Asynchronous Event Notification Using Signals

- Using semaphores for data communication

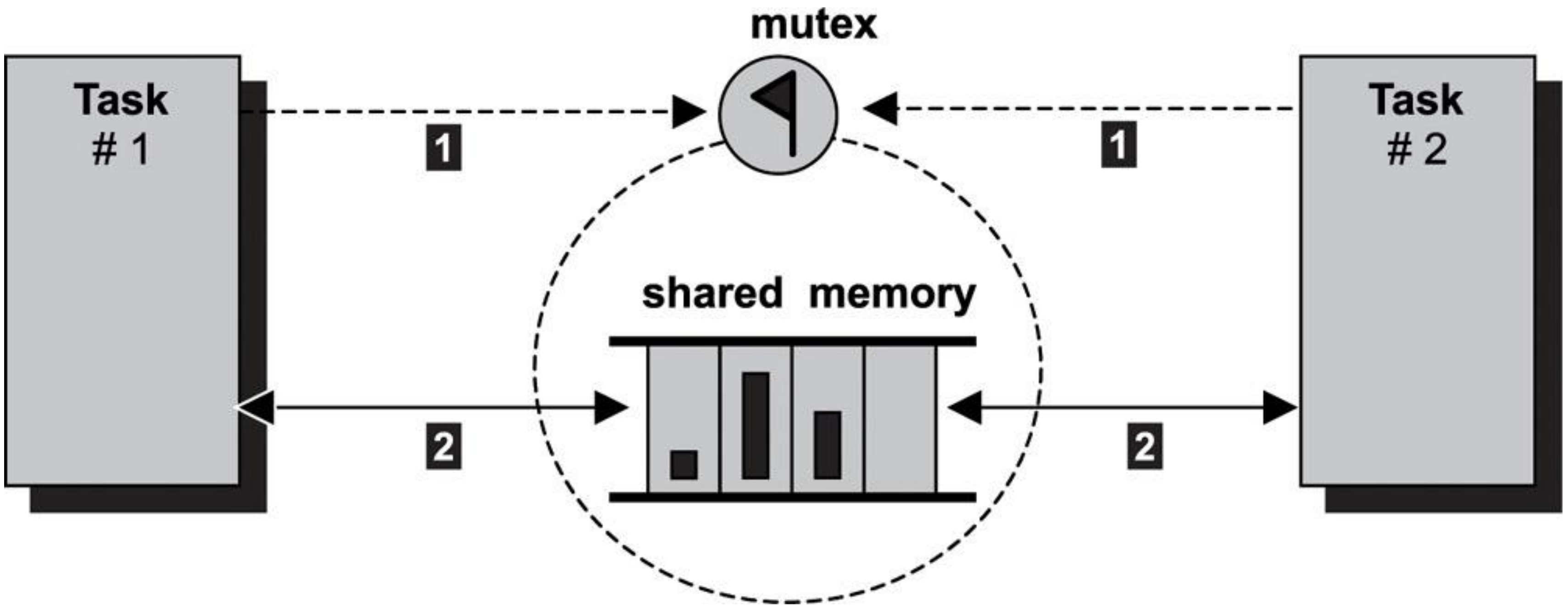



```
Read_Analog_Input_Channel_Cnts(channel#, *adc_counts)
{
    Select the desired analog input channel
    Wait for MUX output to stabilize
    Start the ADC Conversion
    Wait for signal from ADC ISR (with timeout)
    if (timed out)
        Return error code to caller
    else
        Read ADC counts
        Return ADC counts to caller
}
```

```
ADC_ISR(void)
{
    Signal Event
    Clear EOC interrupt
}
```

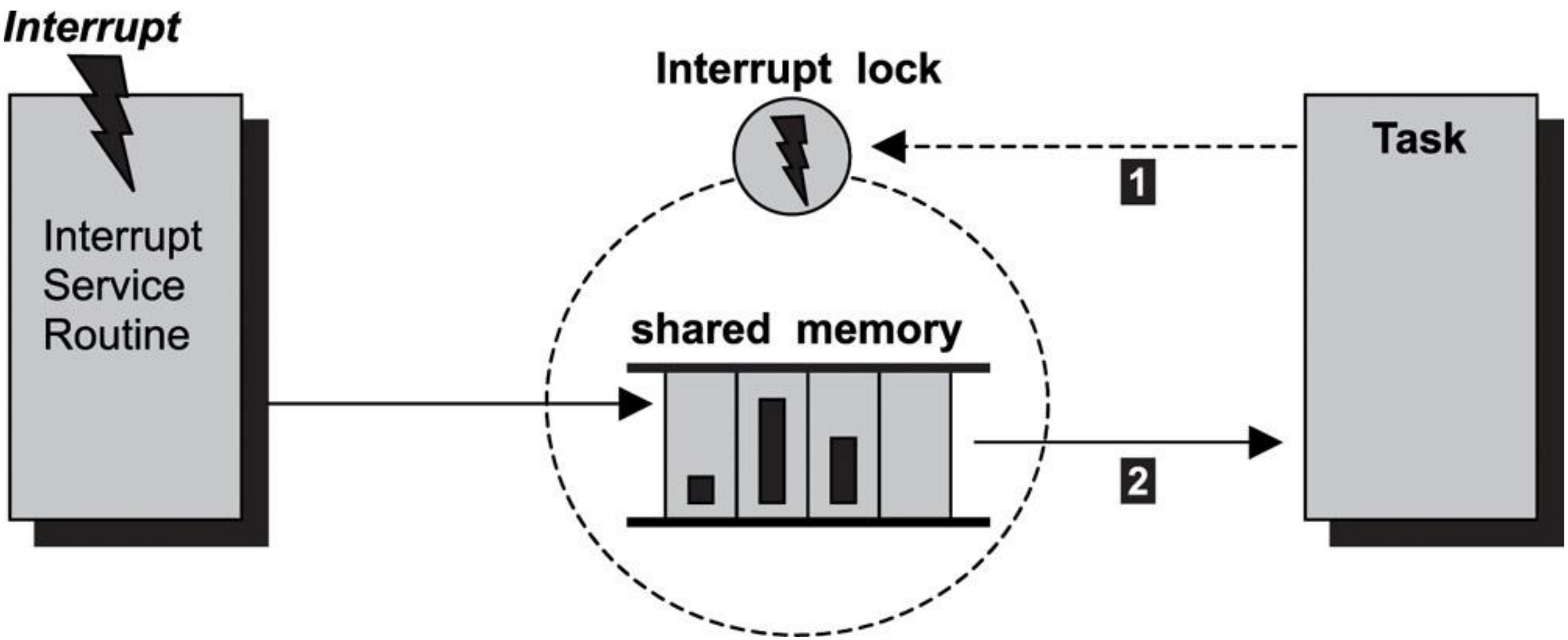
Resource Synchronization

❑Shared Memory with Mutexes



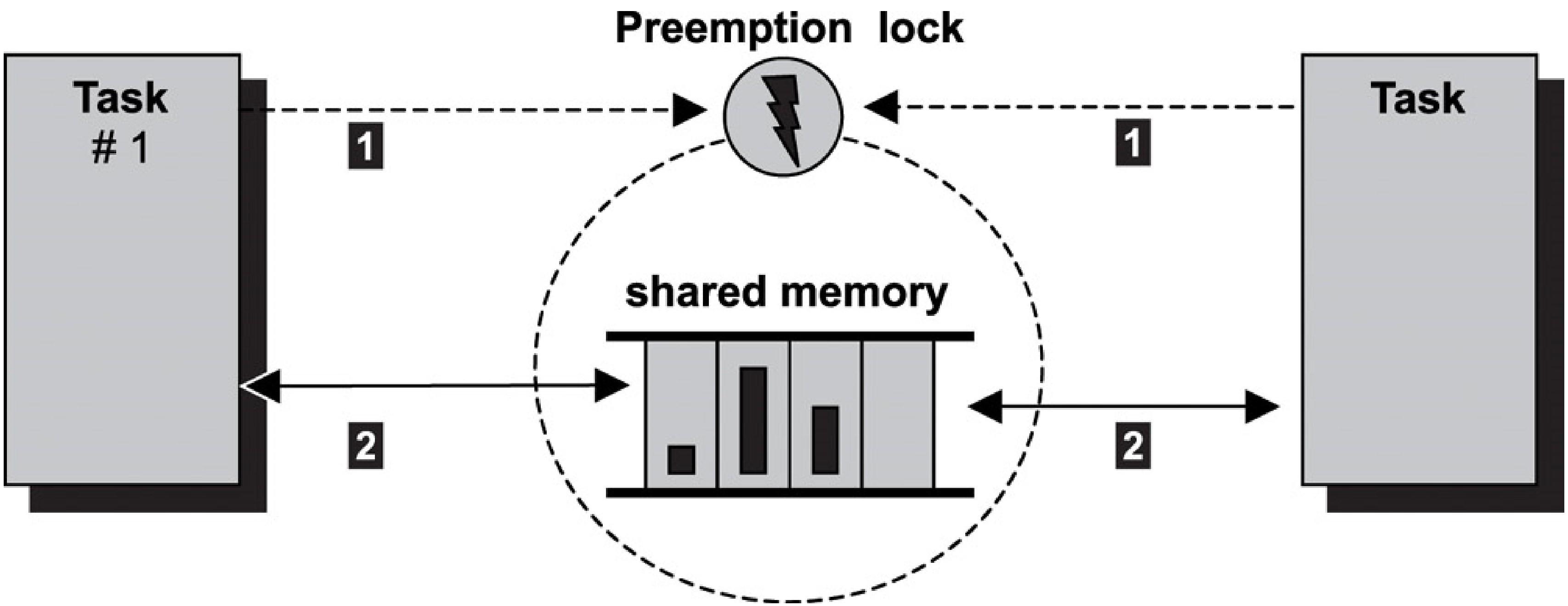
Resource Synchronization

- Shared Memory with Interrupt Locks



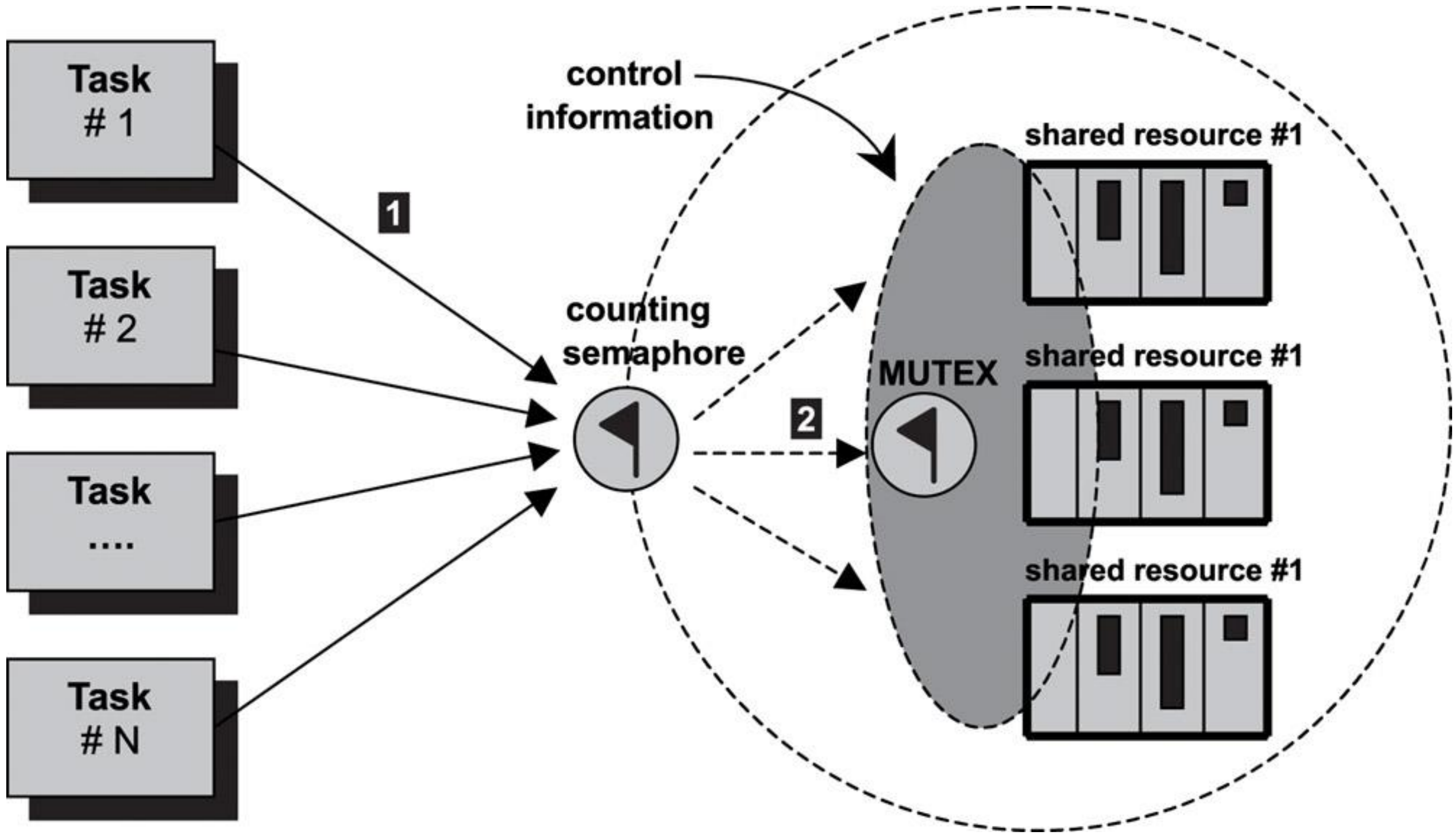
Resource Synchronization

- Shared Memory with Preemption Locks



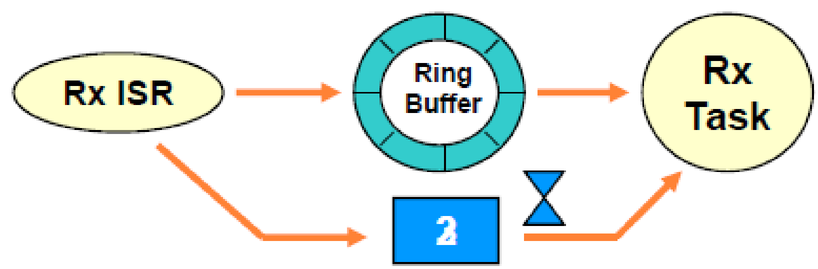
Resource Synchronization

❑ Sharing Multiple Instances of Resources Using Counting Semaphores and Mutexes



Resource Synchronization

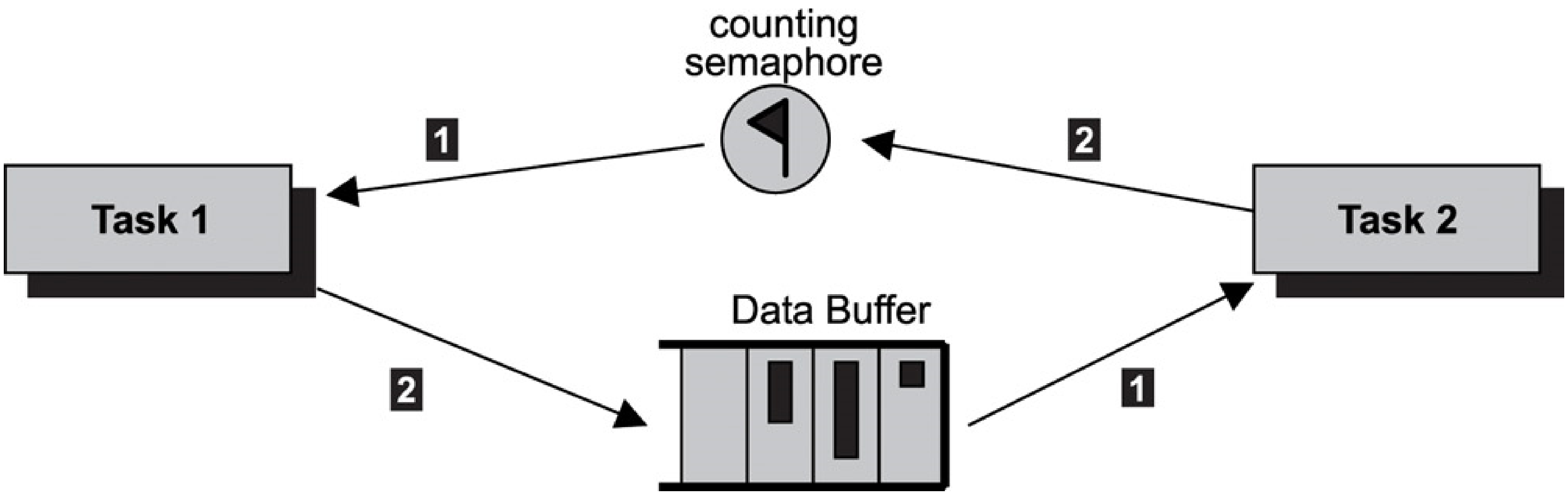
- Serial Communications (Rx)
- Rx ISR
  - Reads and buffers character
  - Clears interrupt source
  - Signals semaphore (every character)



- Rx Task
  - Wait on semaphore
  - Gets character(s) from buffer

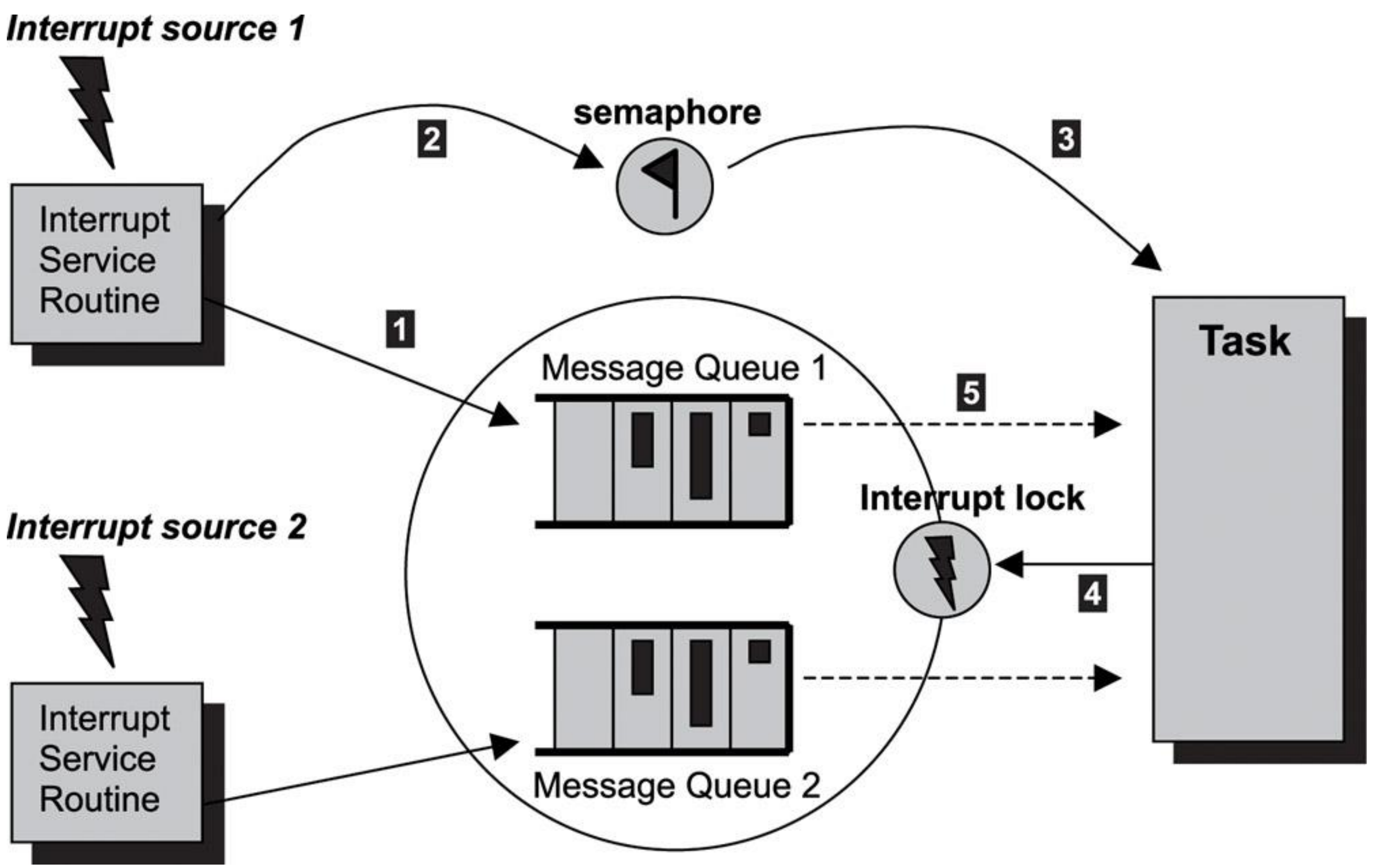
Specific Solution Design Patterns

- Data Transfer with Flow Control



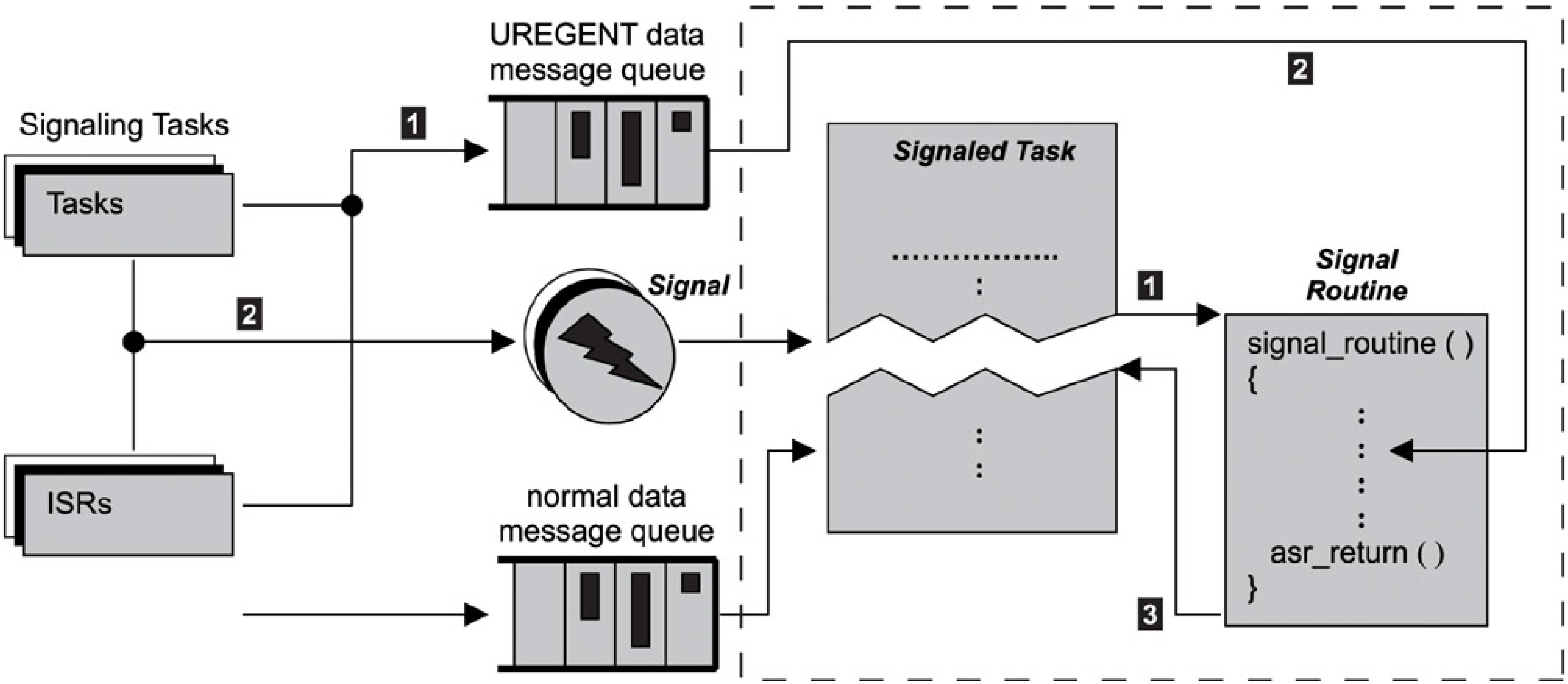
Specific Solution Design Patterns

- Asynchronous Data Reception from Multiple Data Communication Channels



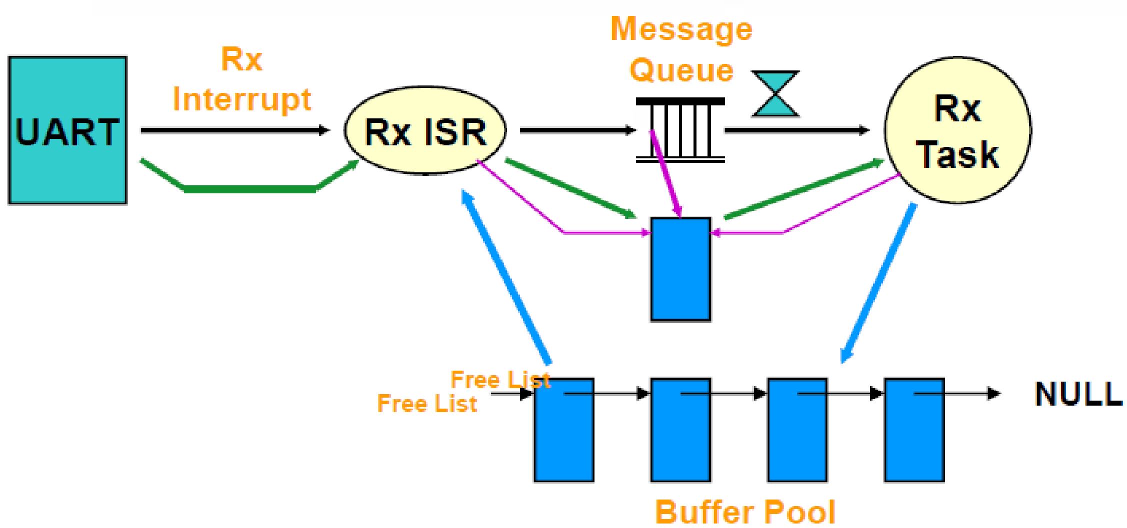
Specific Solution Design Patterns

- Sending High Priority Data between Tasks



Specific Solution Design Patterns

❑ Message Queues & Buffer Pools

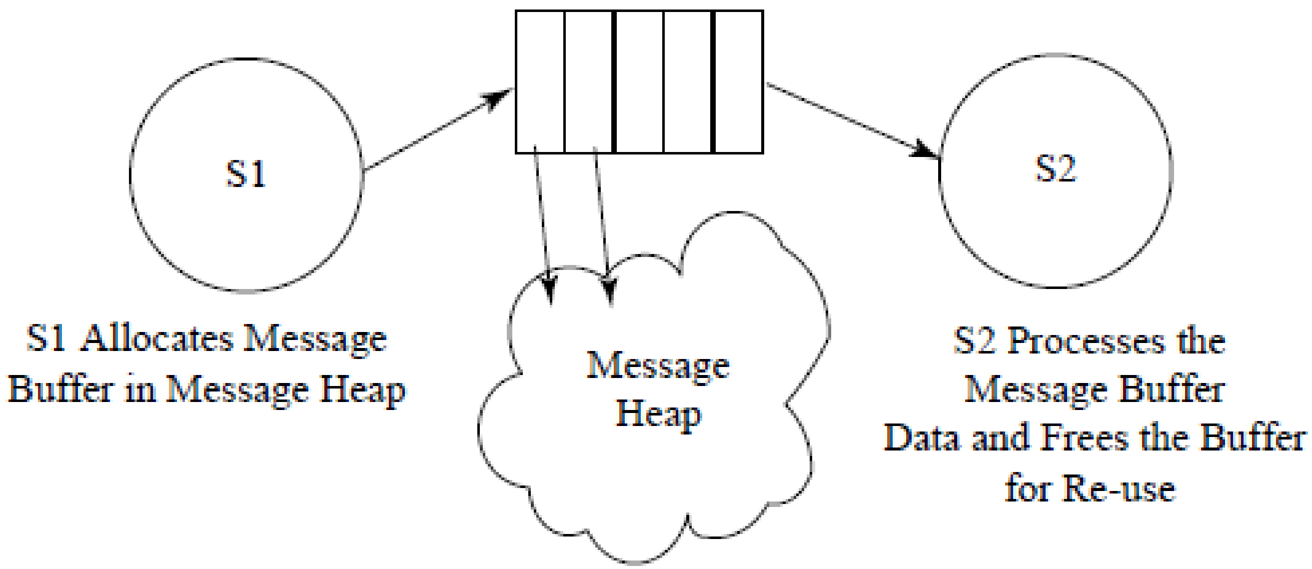



```
RxISR(void)
{
  Read character from UART
  if (1st character of message)
    Get a buffer from Buffer Manager
  Put character in buffer
  if (End of Packet character)
    Send packet to RxTask for processing
}
```

```
RxTask(void)
{
  PACKET *p_msg;

  while (1)
    p_msg = OSQPend(RxQ, timeout)
    Process packet received
    Return packet buffer to Buffer Manager
}
```

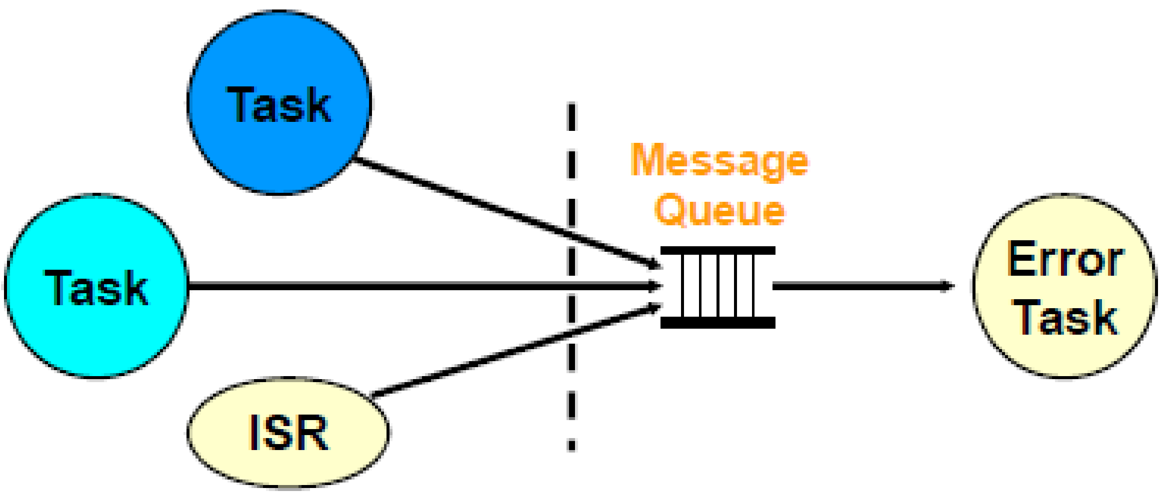
Specific Solution Design Patterns

- Heap-Based Message Queue Communication between Tasks



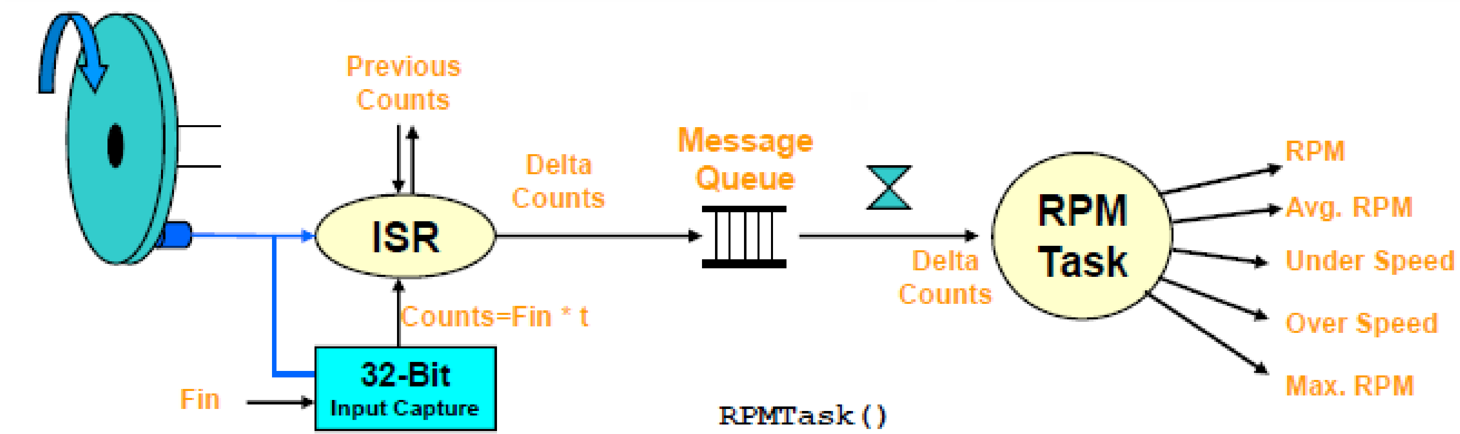
Specific Solution Design Patterns

- Error Handling with Message Queues
  - Error conditions are detected by tasks and ISRs then sent to an error handler
  - The error handler acts as a server



Specific Solution Design Patterns

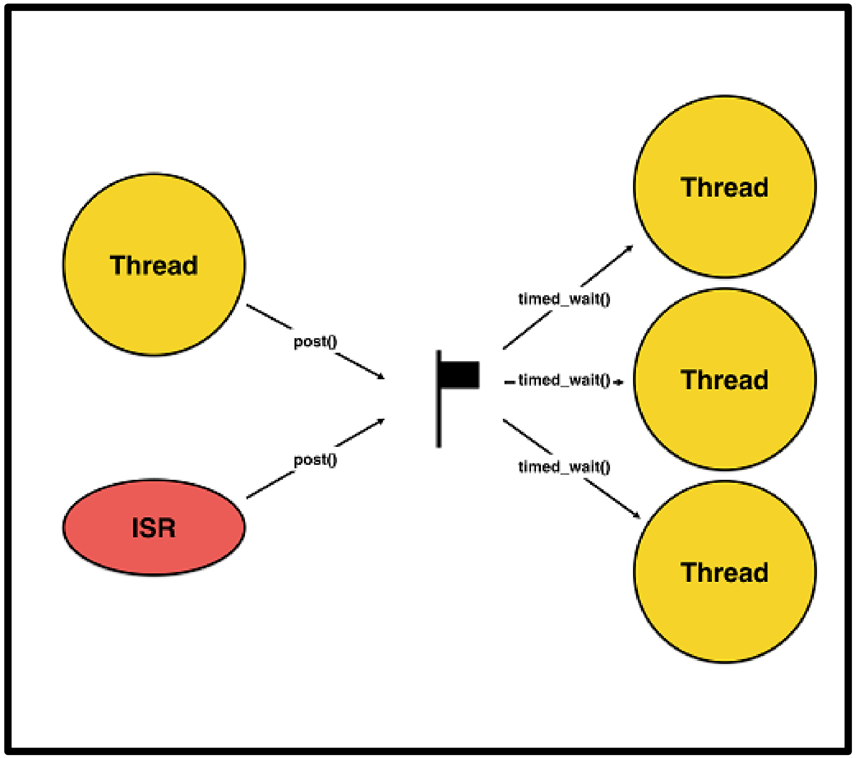
❑ I/O with Message Queue




```
RPM_ISR()
{
  Read Timer;
  DeltaCounts = Counts
              - PreviousCounts;
  PreviousCounts = Counts
  Post DeltaCounts;
}
```

```
RPMTask()
{
  while (1)
    Wait for message from ISR (with timeout);
    if (timed out)
       RPM = 0;
    else
       RPM = 60 * Fin / counts;
    Compute average RPM;
    Check for overspeed/underspeed;
    Keep track of peak RPM;
    etc.
}
```

Agenda



- ❑Synchronization
- ❑Communication
- ❑Resource Synchronization
- ❑Practical Design Patterns
- ➢**Deadlocks**
- ❑Priority Inversion

Deadlocks

Deadlock is the situation in which multiple concurrent threads of execution in a system are blocked permanently because of resource requirements that can never be satisfied.

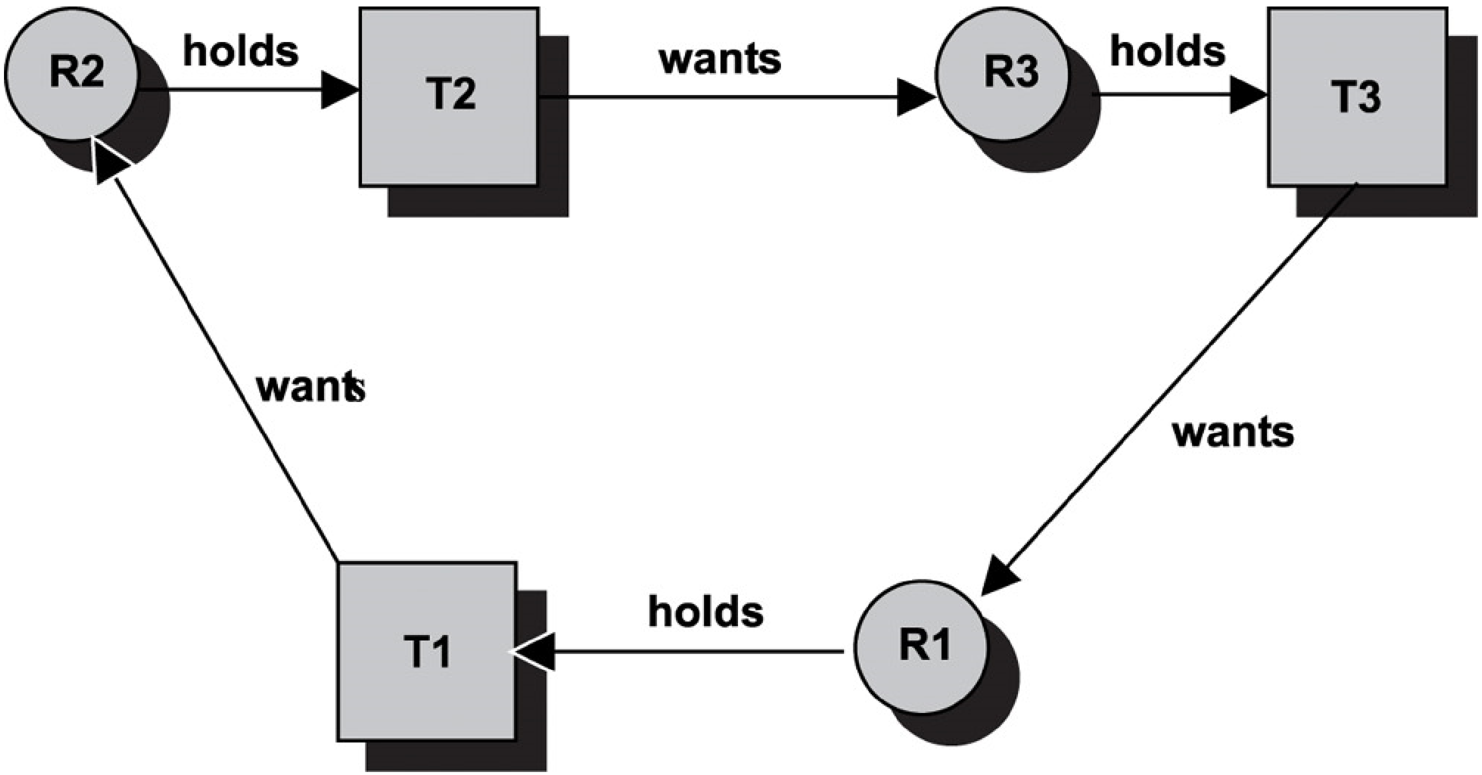
Deadlocks (cont)

- Potential for deadlocks exist in a system in which the underlying RTOS permits resource sharing among multiple threads of execution

- Deadlock occurs when the following four conditions are present:
  - Mutual exclusion
  - No preemption
  - Hold and wait
  - Circular wait

Deadlocks (cont)

- Deadlocks can involve more than two tasks.



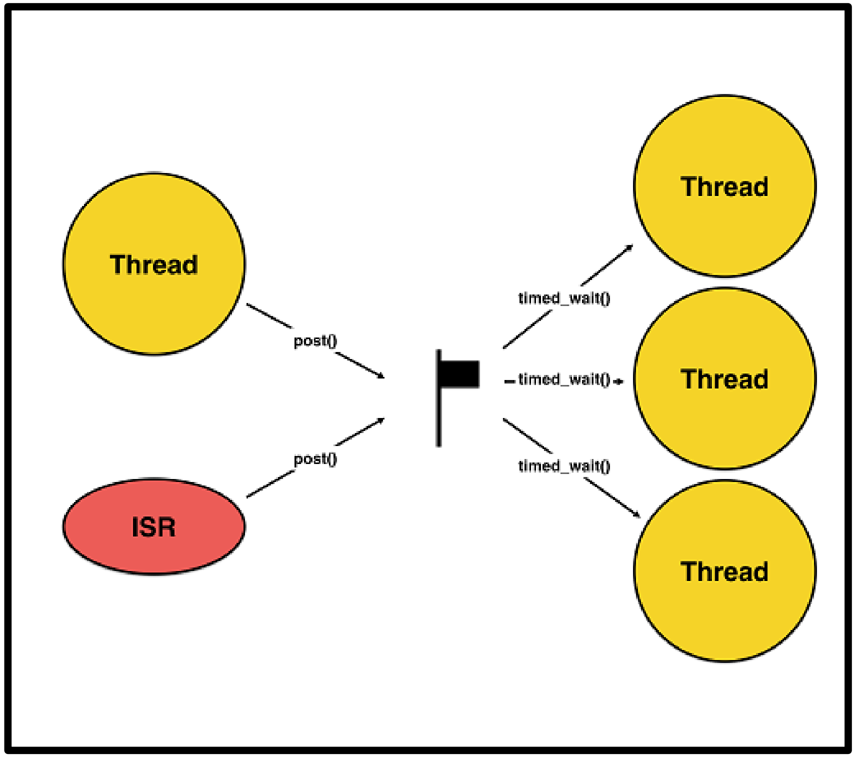
A Deadlock example

```
void  T1 (void)
{
    while (1) {
        Wait for event to occur;            (1)
        Acquire M1;                         (2)
        Access  R1;                         (3)
        :
        :
        \--------  Interrupt!               (4)
        :
        :                                   (8)
        Acquire M2;                         (9)
        Access  R2;
    }
}



void  T2 (void)
{
    while (1) {
        Wait for event to occur;            (5)
        Acquire M2;                         (6)
        Access  R2;
        :
        :
        Acquire M1;                         (7)
        Access  R1;
    }
}
```

Some techniques used to avoid deadlocks

- Acquire all resources before proceeding
- Always acquire resources in the same order
- Use timeouts on wait calls

Agenda



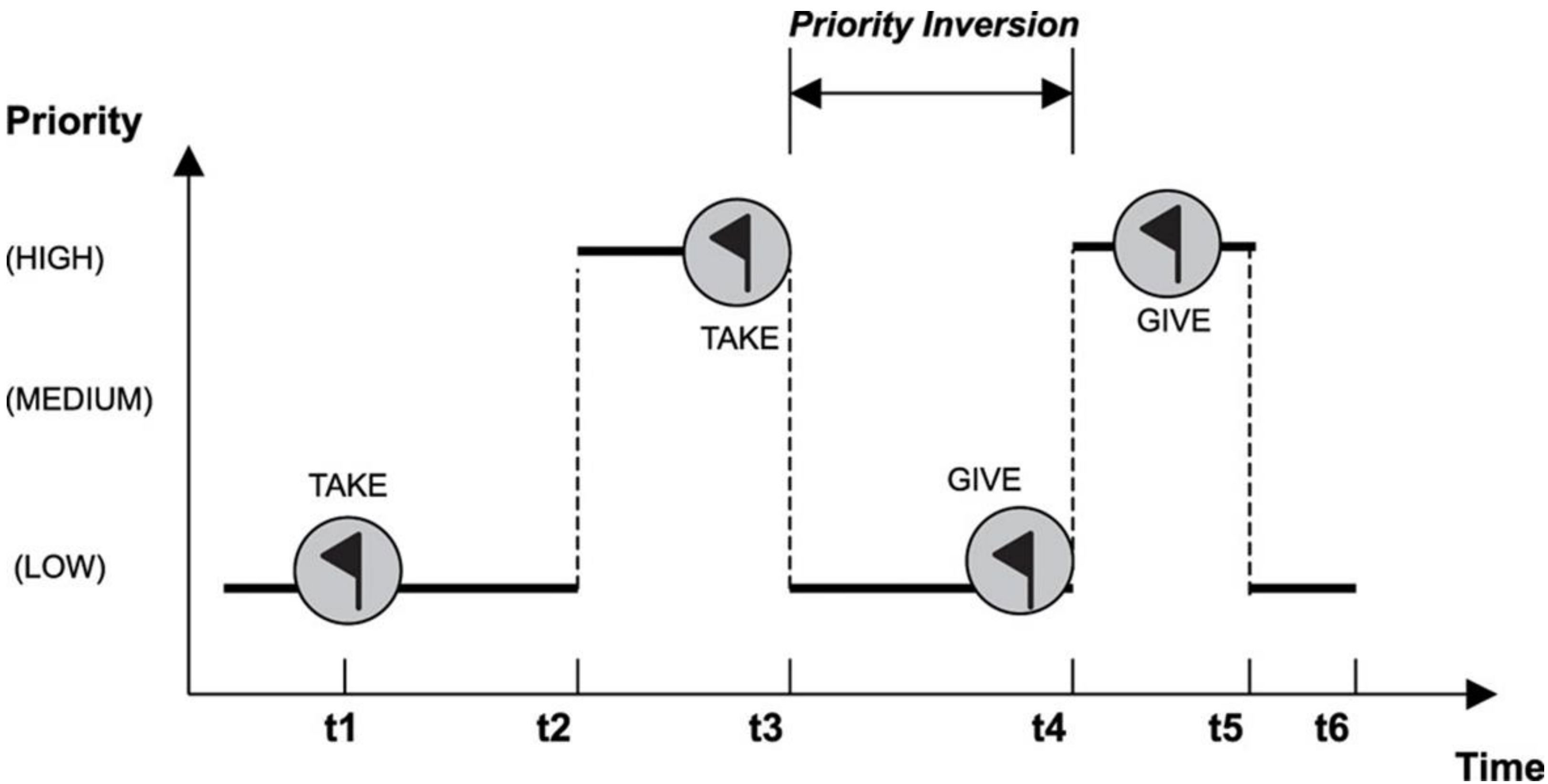
- ❑ Synchronization
- ❑ Communication
- ❑ Resource Synchronization
- ❑ Practical Design Patterns
- ❑ Deadlocks
- ➢ **Priority Inversion**

Priority Inversion

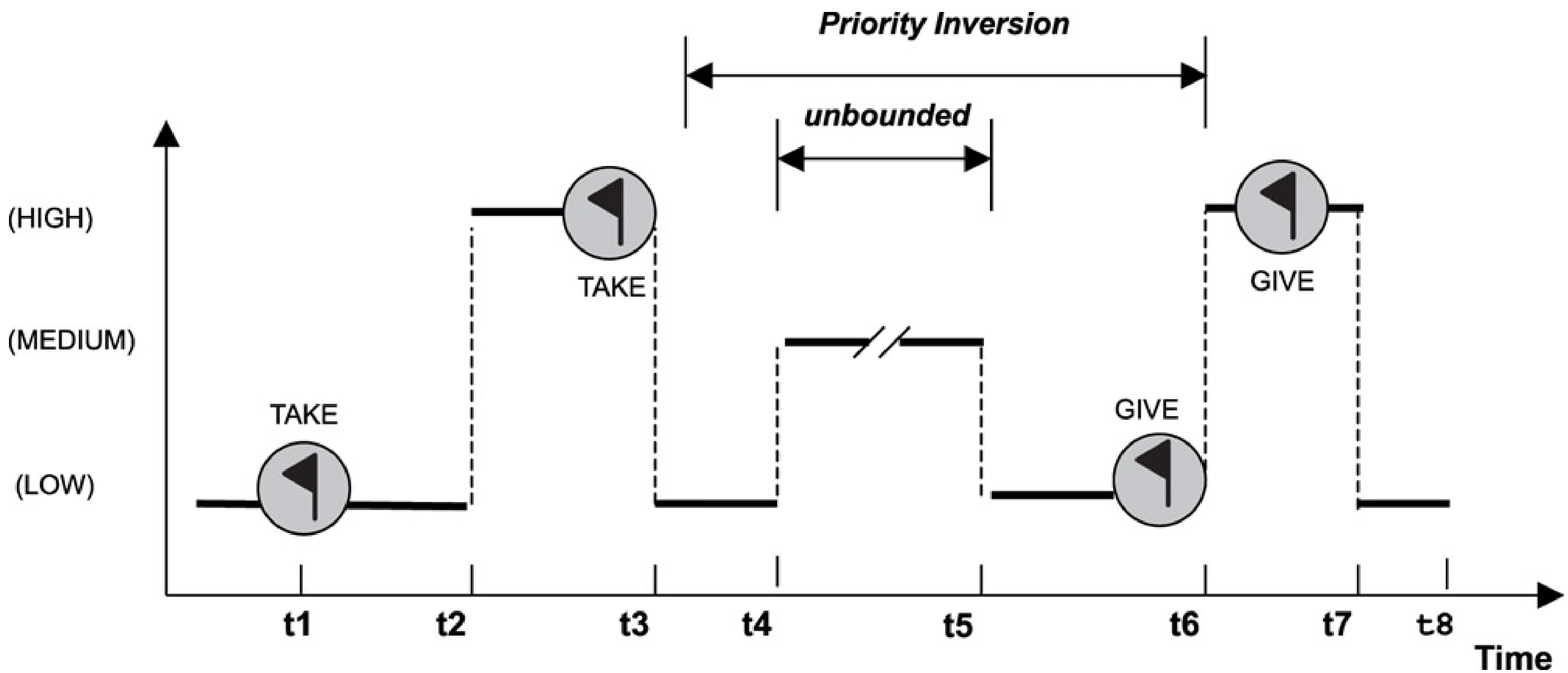
- Blocking of a high priority task due to a lower priority task locking a shared resource.

- Priority inversion results from resource synchronization among tasks of differing priorities

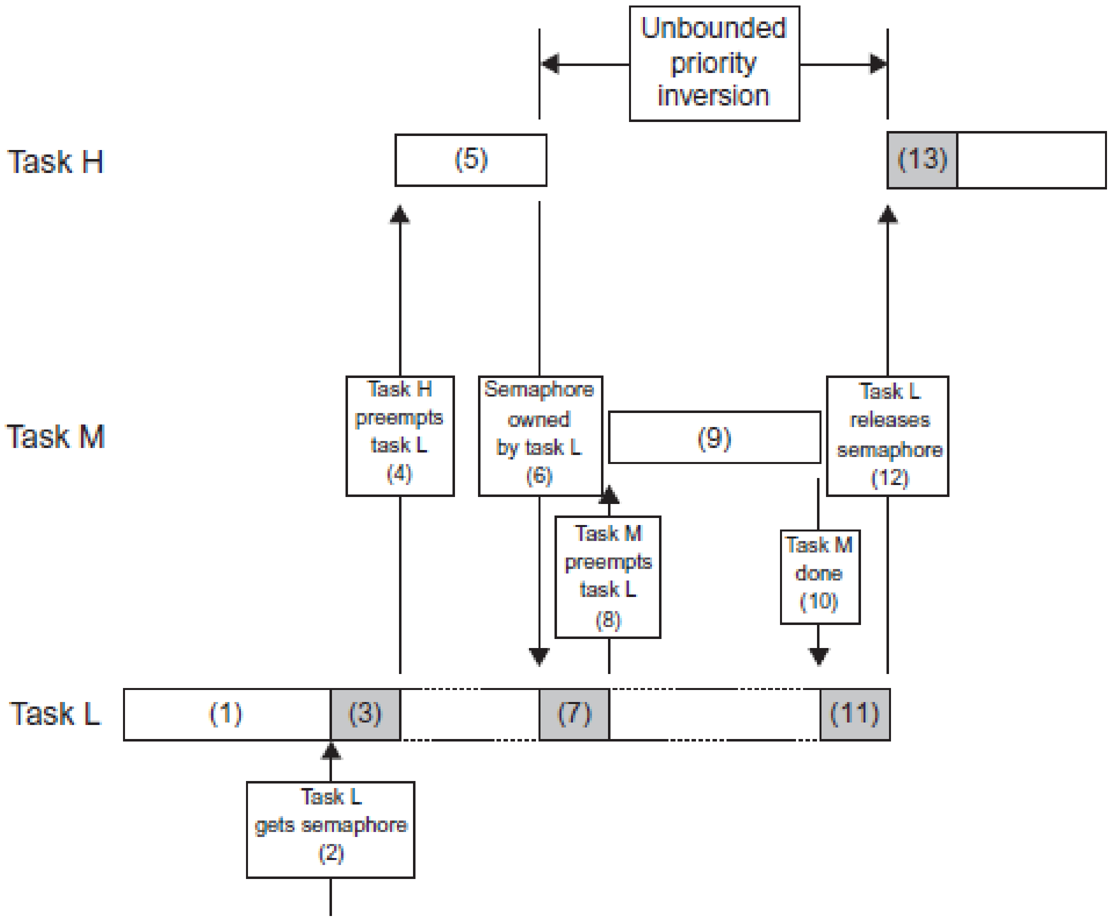
Bounded priority inversion



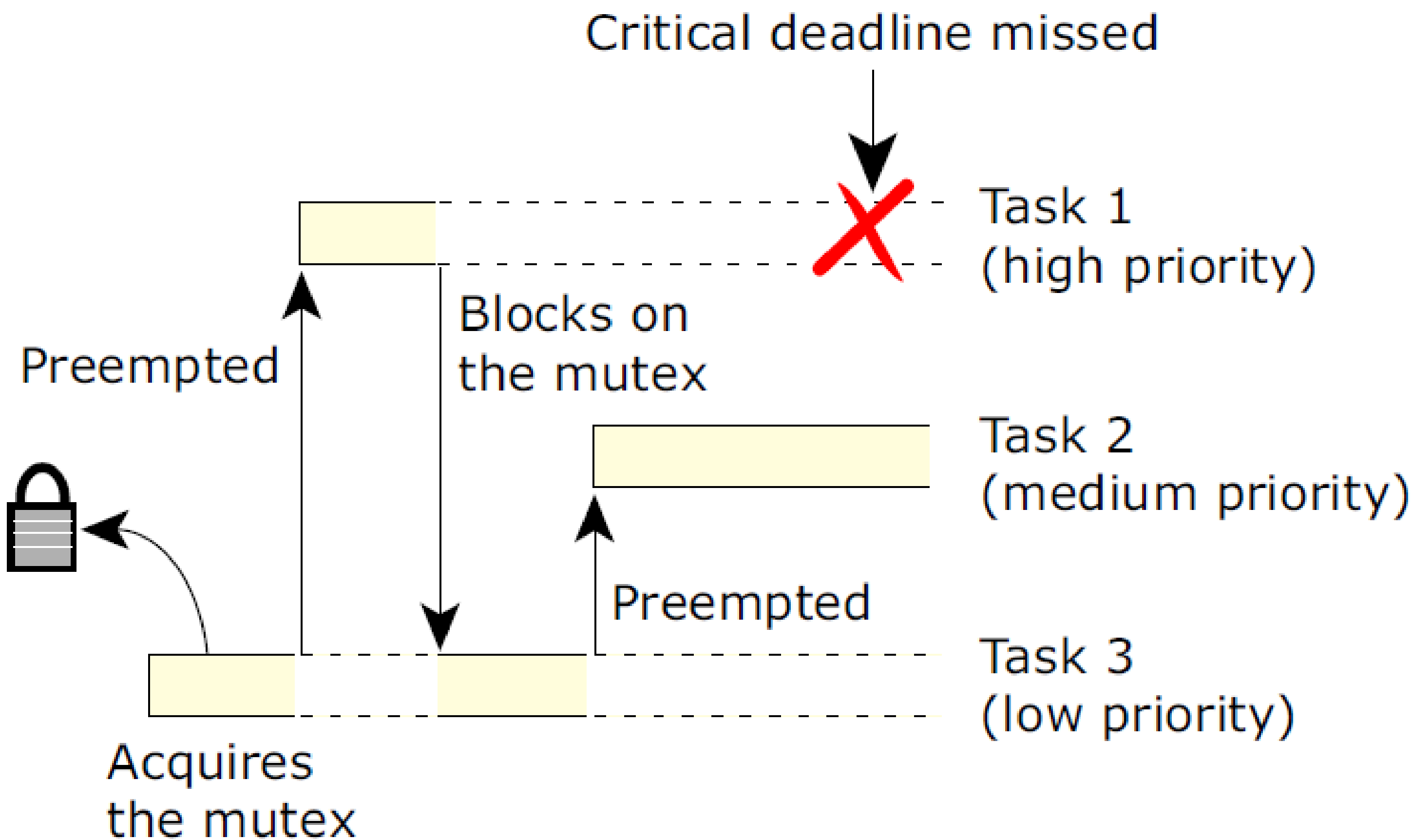
Unbounded priority inversion



Unbounded priority inversion - 2



Another example of priority inversion

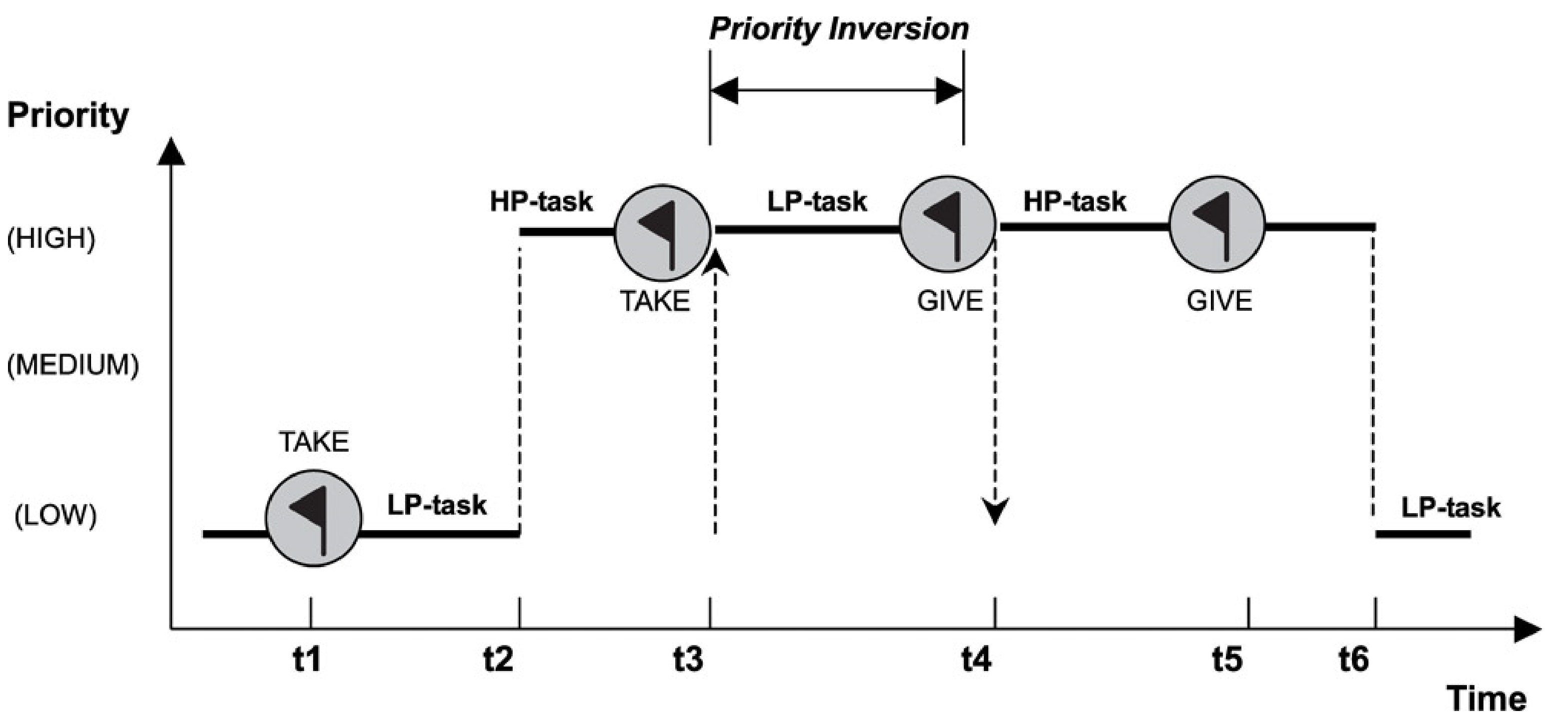


Solutions to Priority Inversion

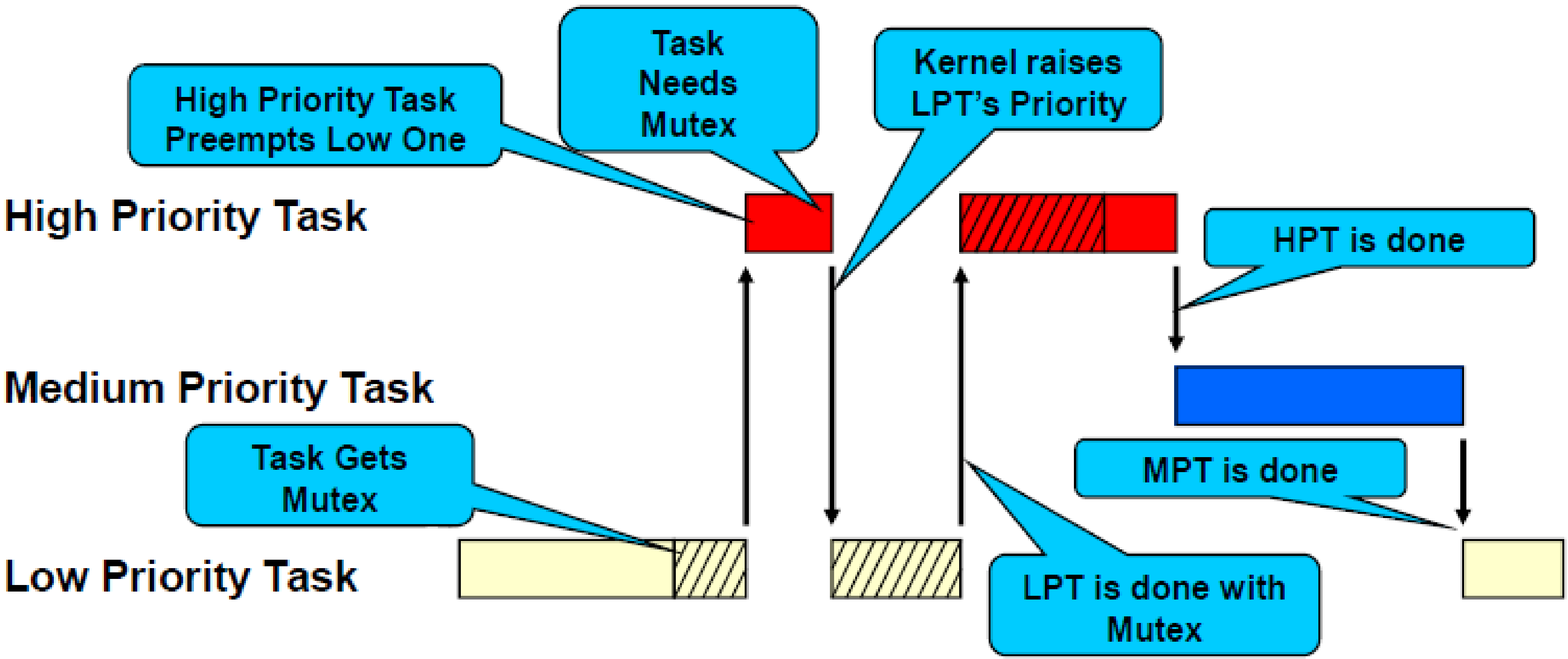
- There are two fundamental ways to deal with race conditions while avoiding priority inversion:
  - Use a mechanism that adjusts priorities of low priority tasks holding resources
  - Use a non-blocking mechanism

Priority Inheritance Protocol

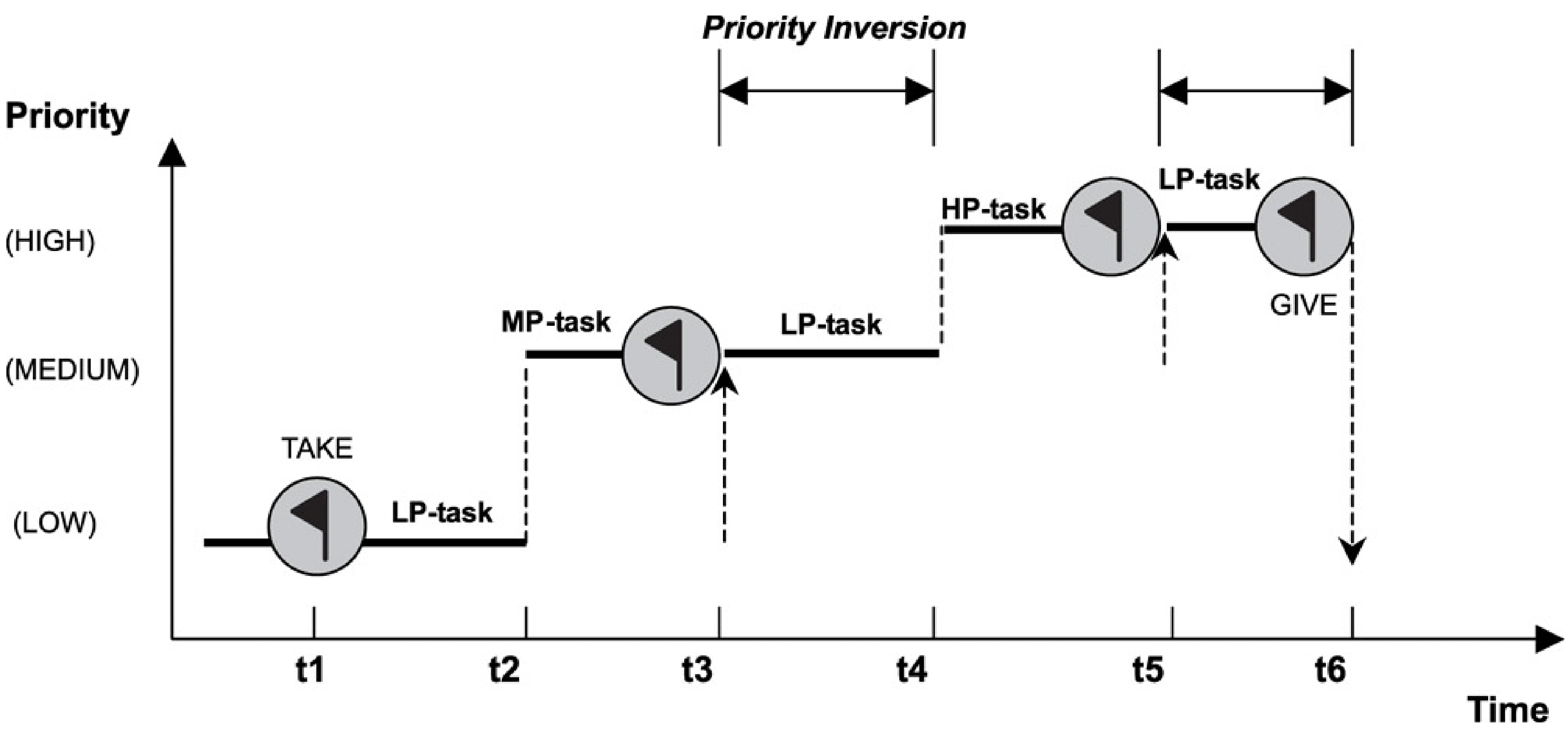
- A resource access control protocol that raises the priority of a task
- Not a good protocol



Mutex with Priority Inheritance

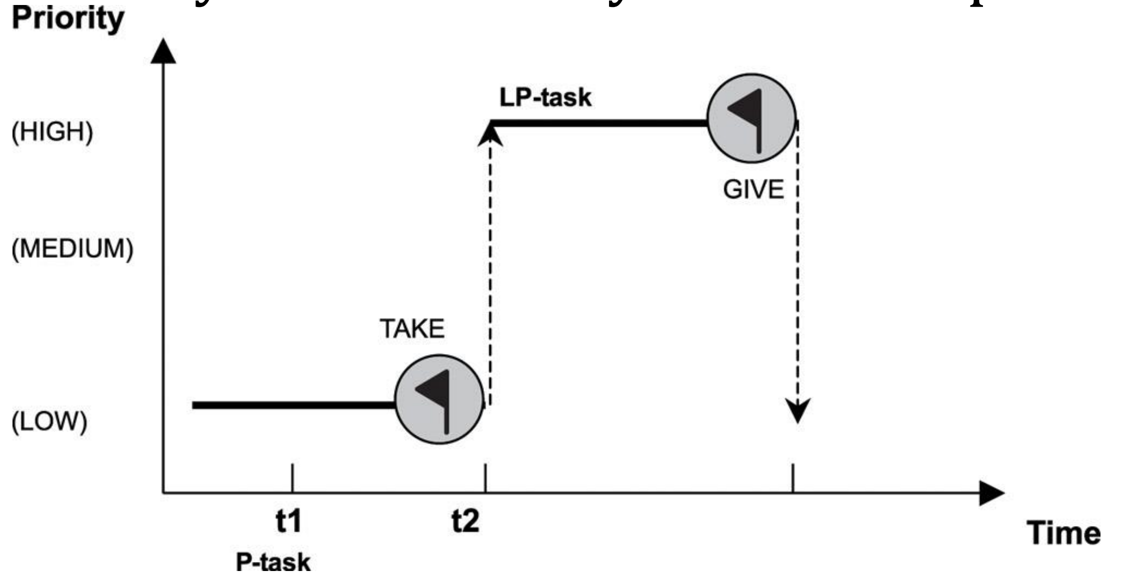


Transitive priority promotion



Highest Locking Protocol

- A fixed Priority, called priority ceiling is assigned to a resource
  - Priority is higher than any Task that contends for the resource
- When Task Obtains the resource
  - Task Priority is immediately elevated to priority ceiling



Priority Ceiling Protocol

- Highest Locking Protocol (HLP) is very similar
  - Similar to Priority Inheritance, except that each resource is given a priority ceiling, which is the priority of the highest-priority task that may lock the resource
  - When a resource is locked, task immediately inherits the priority ceiling
- Advantage:
  - It Works
- Disadvantages:
  - Significant RTOS overhead
  - Significant reduction in schedulable bound due to blocking priority ceiling