

Politecnico  
di Torino

Department of Control and  
Computer Engineering



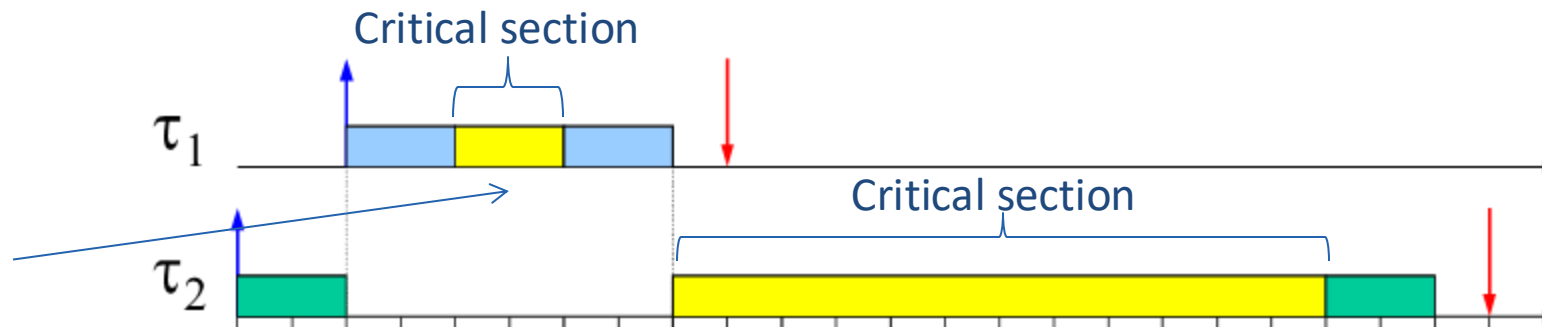
# SHARED RESOURCES

STEFANO DI CARLO

# CRITICAL SECTION

- ▶ **Resource**: any software structure used by processes to advance its execution
  - ▶ Data structure, set of variables, main memory area, file, set of registers of a peripheral device, ...
- ▶ **Private resource**: a resource dedicated to a particular process
- ▶ **Shared resource**: a resource that can be used by more tasks
- ▶ **Shared exclusive resource**: a shared resource protected against concurrent accesses (e.g., only one process at a time can use the resource)
- ▶ **Critical section**: it is a piece of software that accesses a shared exclusive resource
  - ▶ When a process runs its critical section, other processes willing to use the shared exclusive resource have to wait

$t_1$  and  $t_2$  share an exclusive resource  
 $t_1$  is in critical section, thus  $t_2$  has to wait



# CRITICAL SECTION

- ▶ An application can be structured through multiple concurrent tasks that can be
  - ▶ Independent tasks that cannot affect or be affected by the execution of another task
  - ▶ Cooperating tasks can affect or be affected by the execution of another task
- ▶ Advantages of task cooperation
  - ▶ Information sharing
  - ▶ Computation speed-up
  - ▶ Modularity
- ▶ Problems
  - ▶ Contention of exclusive shared resources: race problem

# RACE PROBLEM

## ► Producer

```
while (true) {  
    /* produce an item */  
    while (count == BUFFER_SIZE)  
        ; // do nothing  
    buffer [in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    count++;  
}
```

## ► Consumer

```
while (true) {  
    while (count == 0)  
        ; // do nothing  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    count--;  
    /* consume the item */  
}
```

Shared resource

Two arrows originate from the text 'Shared resource' at the bottom. One arrow points to the 'count++;' line in the Producer code block. The other arrow points to the 'count--;' line in the Consumer code block. This visualizes that both threads are accessing and modifying the same 'count' variable, which is a shared resource.

# RACE PROBLEM

- ▶ `count++` could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- ▶ `count--` could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

- ▶ Race condition is originated by tasks whose critical sections are executed concurrently

- ▶ Execution interleaving with “count = 5”:

```
S0: producer register1 = count
S1: producer register1 = register1 + 1
S2: consumer register2 = count
S3: consumer register2 = register2 - 1
S4: producer count = register1 → count = 6
S5: consumer count = register2 → count = 4
```

# SOLVING THE RACE CONDITION

- ▶ Stops preemption mechanisms by disabling the interrupts

```
do {  
    disable_interrupt() ;  
    //critical section  
    enable_interrupt() ;  
    //remainder section  
} while (TRUE);
```

- ▶ Advantages
  - ▶ Simple
- ▶ Disadvantages
  - ▶ Interrupt latency not predictable → bad for real time
  - ▶ Disable preemption may prevent scheduling of higher priority independent tasks.

# SOLVING THE RACE CONDITION

- ▶ Use atomic (=not interruptible) instructions provided by the CPU, such as

- ▶ Test and set

```
boolean TestAndSet(boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

- ▶ Swap

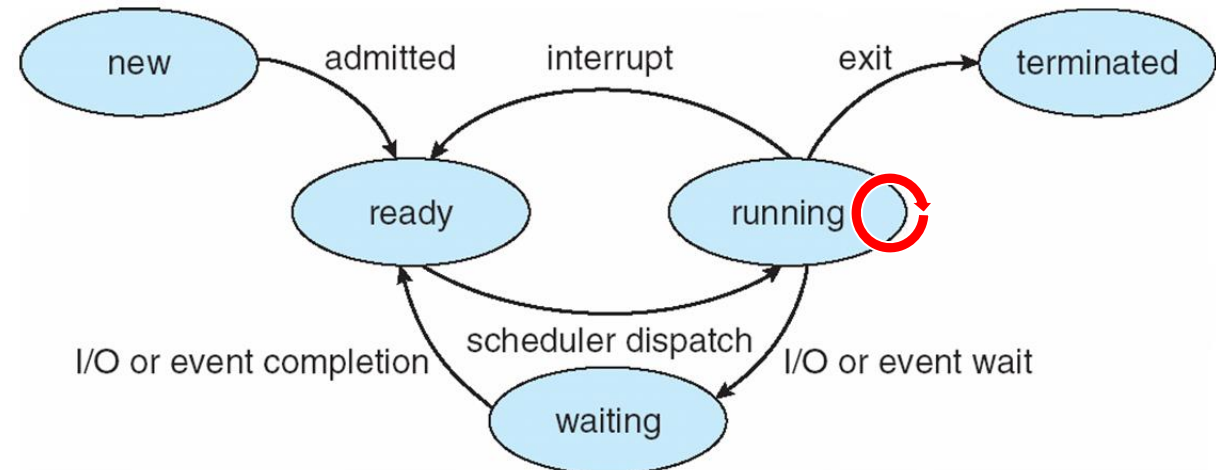
```
void Swap(boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

# SPINLOCK – BUSY WAITING

- ▶ A variable `lock` is created and initialized to `FALSE`

```
do {  
    while( TestAndSet( &lock ) == TRUE)  
        ;    // do nothing, just wait  
  
    //critical section  
  
    lock = FALSE;  
  
    //remainder section  
} while (TRUE);
```

- ▶ As long as `lock == TRUE` the process cannot proceed
- ▶ The process remains **RUNNING** for its time slice, but it does nothing → CPU may be wasted





# SPINLOCK – BUSY WAITING

- ▶ Allows a thread to "spin" (actively wait) until it acquires a lock.
- ▶ Typically used in multiprocessor systems where waiting is expected to be brief.
- ▶ Advantages:
  - ▶ Simple and fast for short waits.
  - ▶ No context switching overhead.
- ▶ Disadvantages:
  - ▶ Wastes CPU cycles while waiting (busy-waiting).
  - ▶ Not ideal for single-processor systems.
- ▶ Example Usage:
  - ▶ Atomic increment of a shared variable in a multiprocessor system.

# SEMAPHORES

- ▶ Semaphore  $S$  contains
  - ▶ Integer value  $S \rightarrow \text{value}$
  - ▶ Waiting queue  $S \rightarrow \text{list}$ 
    - ▶ List of tasks in WAITING state, waiting for the semaphore to become available
- ▶ Two atomic operations are used to modify the value of  $S$ 
  - ▶ wait, to decrement the semaphore value, and block a task when the semaphore is 0
  - ▶ signal, to increase the semaphore value, and set to ready a previously blocked task

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this task to S->list;  
        set this process WAITING;  
        schedule();  
    }  
}
```

```
signal(semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        remove a task T from S->list;  
        set T READY;  
        schedule();  
    }  
}
```

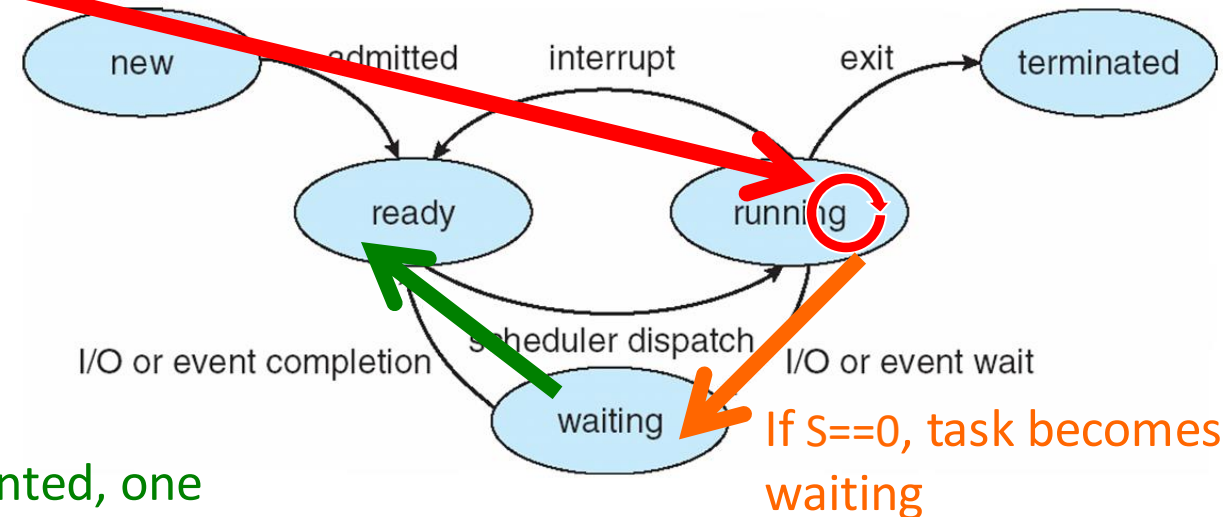
# SEMAPHORES



- ▶ A semaphore  $S$  initialized to 1

```
do {  
    wait( S );  
  
    //critical section  
  
    signal( S );  
  
    //remainder section  
} while (TRUE);
```

If  $S==1$ , task keeps running



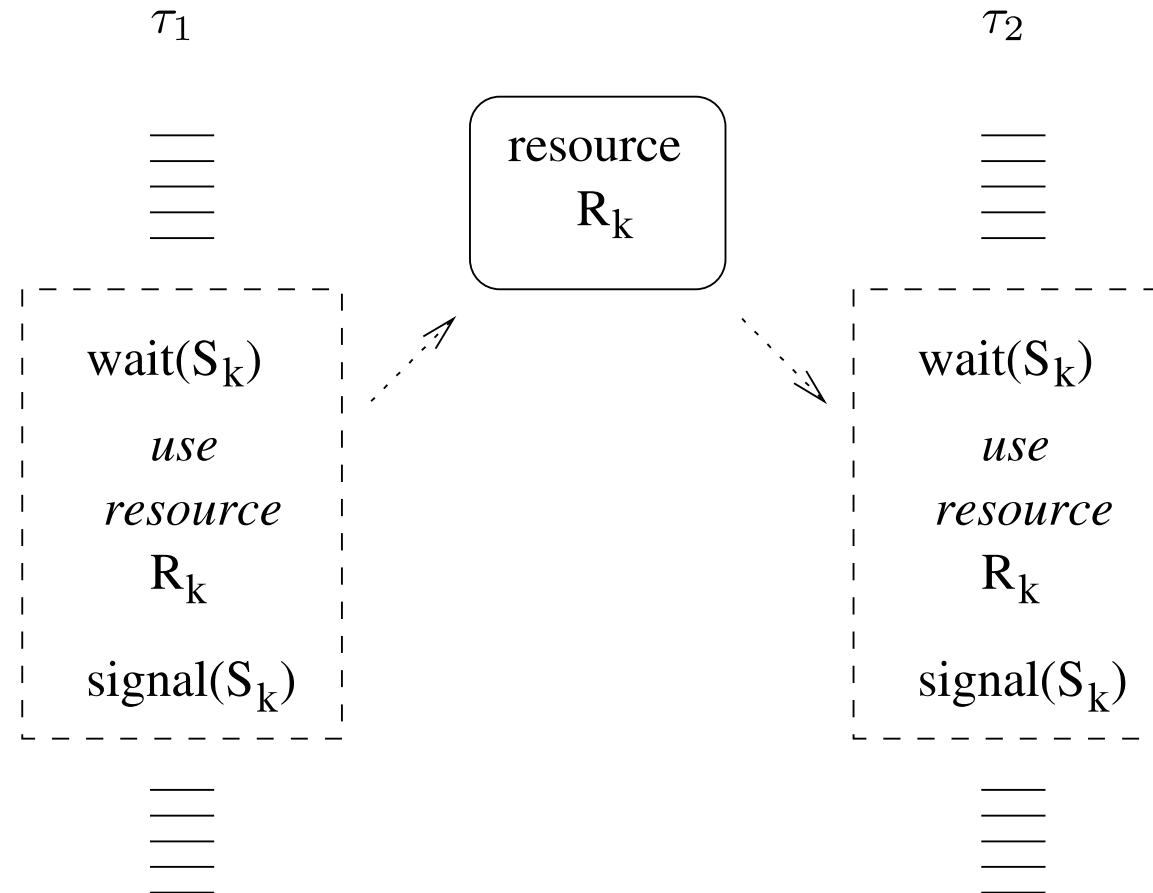
When  $S$  is incremented, one waiting task moves to ready

# BUSY WAITING VS SEMAPHORE

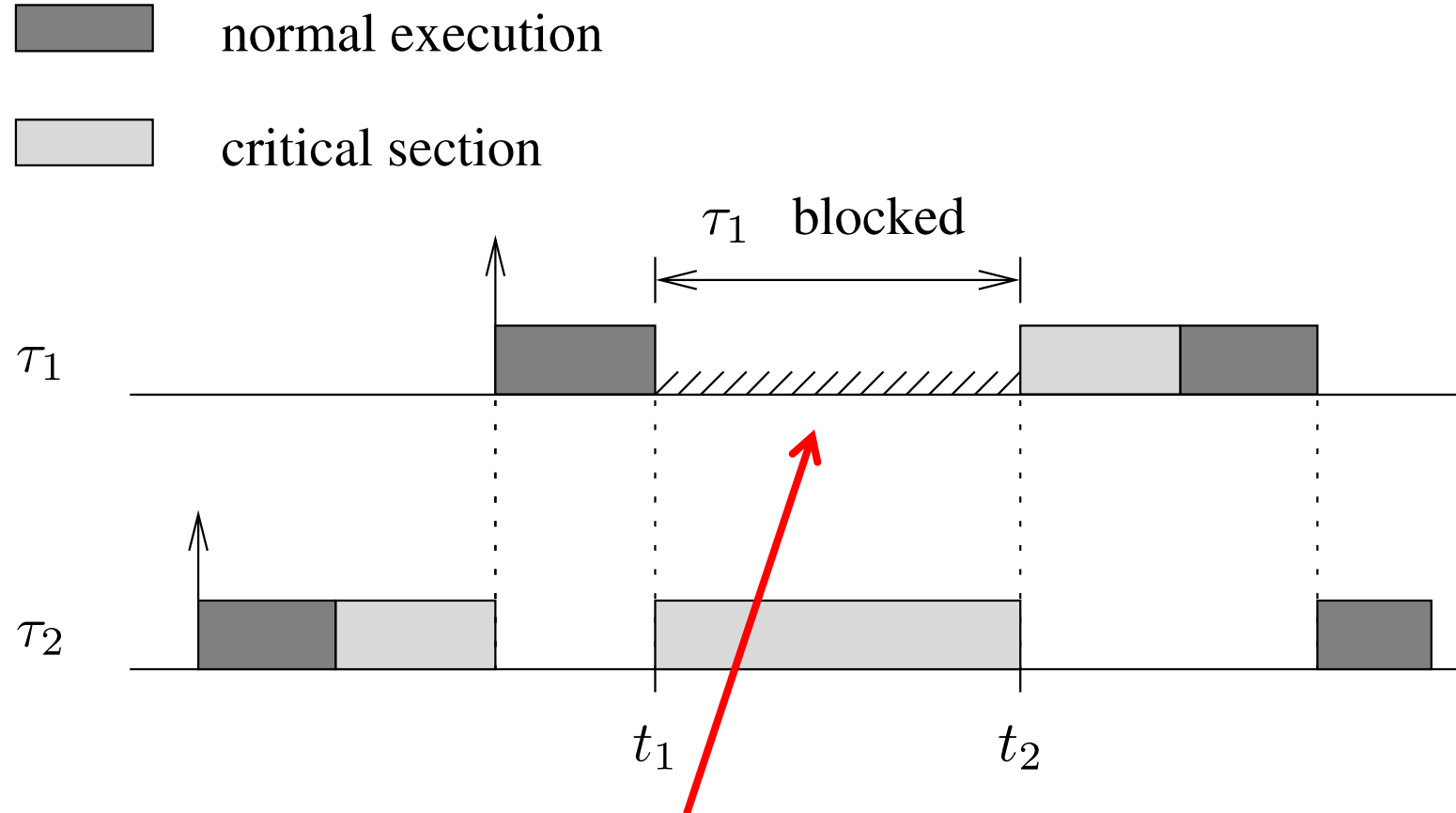
- ▶ Busy waiting
  - ▶ The process keeps running while waiting
- ▶ Semaphore
  - ▶ The process is moved to wait state if it cannot enter its critical section, and context switch takes place
- ▶ Performance issue
  - ▶ CPU clock cycles for 1 context switch = 1000x CPU clock cycles for 1 spin of busy waiting
- ▶ For small critical section (few CPU cycles) busy waiting is better than semaphore
  - ▶ Few CPU cycles wasted waiting, but no context switch
- ▶ For big critical section (many CPU cycles) semaphore is better than busy waiting
  - ▶ The entire time slice is not wasted doing busy waiting

# THE PRIORITY INVERSION PROBLEM

- Let us assume  $\text{Priority}(\tau_1) > \text{Priority}(\tau_2)$



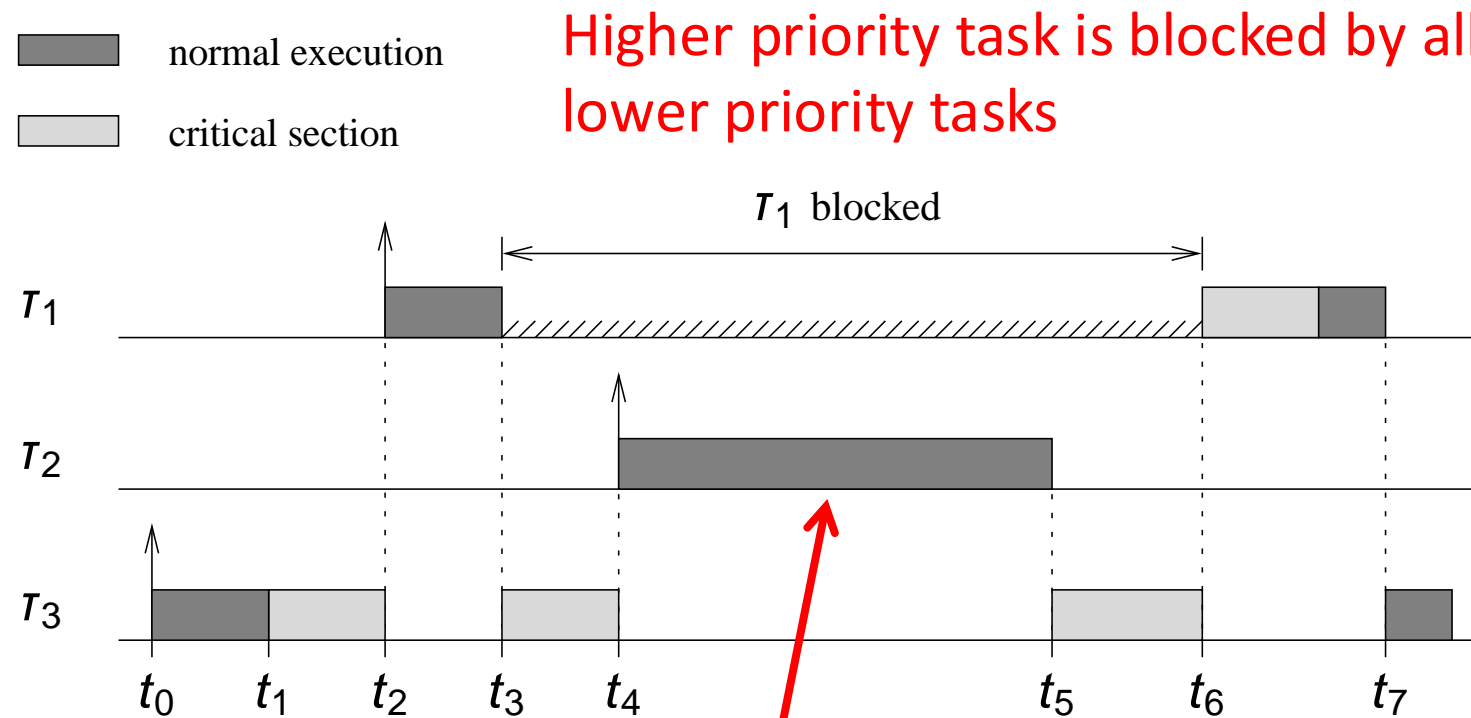
# THE PRIORITY INVERSION PROBLEM



Problem: higher priority task is blocked by lower priority task critical section

# THE PRIORITY INVERSION PROBLEM

- The blocking time of a task on a busy resource cannot be bounded by the duration of the critical section executed by the lower-priority task



$t_2$  must finish before  $t_3$  can exit critical section

# SOLUTION TO PRIORITY INVERSION

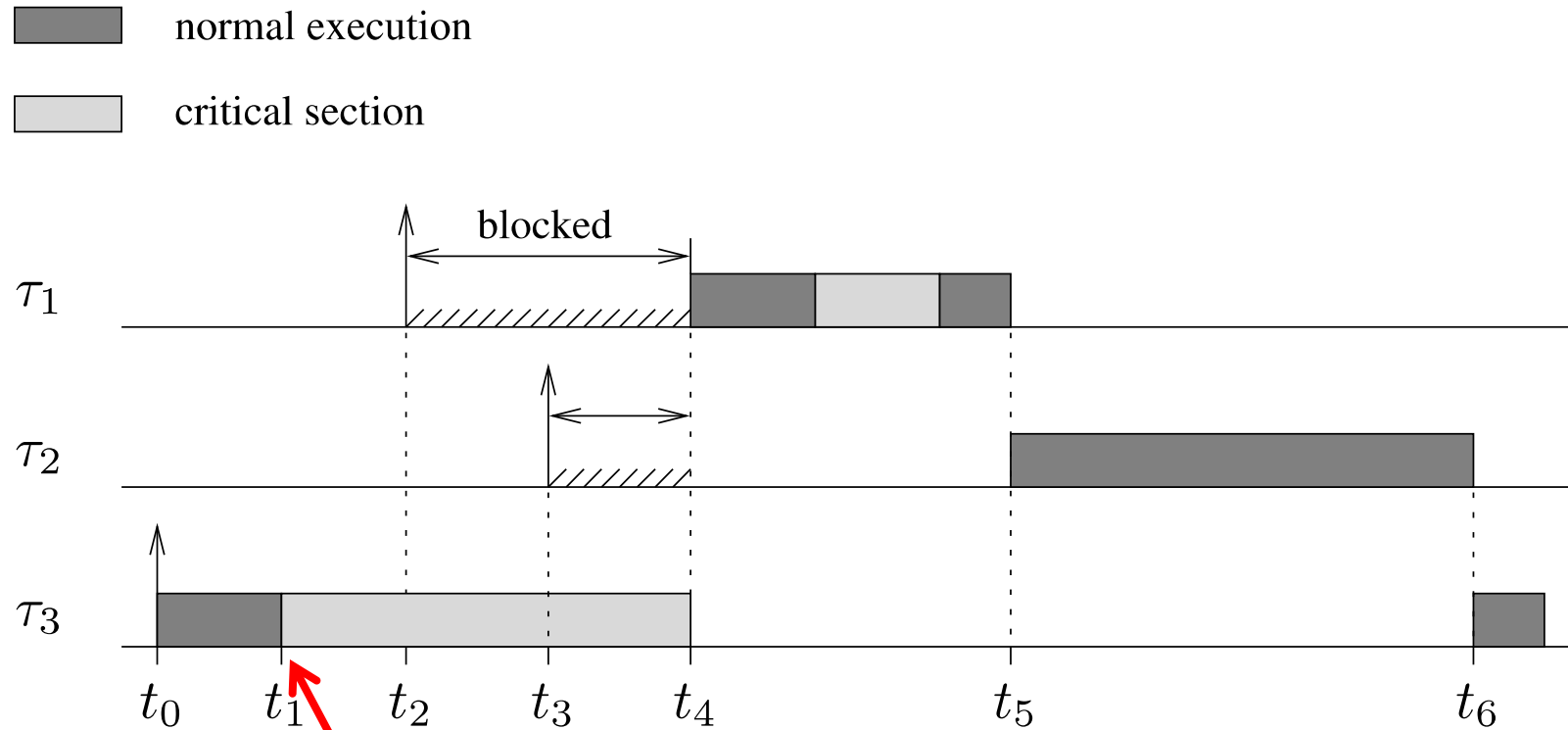
- ▶ Three solutions
  - ▶ Non-Preemptive Protocol (NPP)
  - ▶ Immediate Priority Ceiling (IPC)
  - ▶ Priority Inheritance Protocol (PIP)
- ▶ Assumption:
  - ▶ Rate monotonic scheduling (RM)



# NON-PREEMPTIVE PROTOCOL (NPP)

- ▶ Preemption is not allowed during the execution of any critical section
- ▶ The approach:
  - ▶ The priority of a task is raised to the highest priority level whenever it enters a critical section
  - ▶ The priority is lowered to the original value when the task exits the critical section

# EXAMPLE



The priority of  $t_3$  is set to that of  $t_1$  so it is not preempted while in its critical section

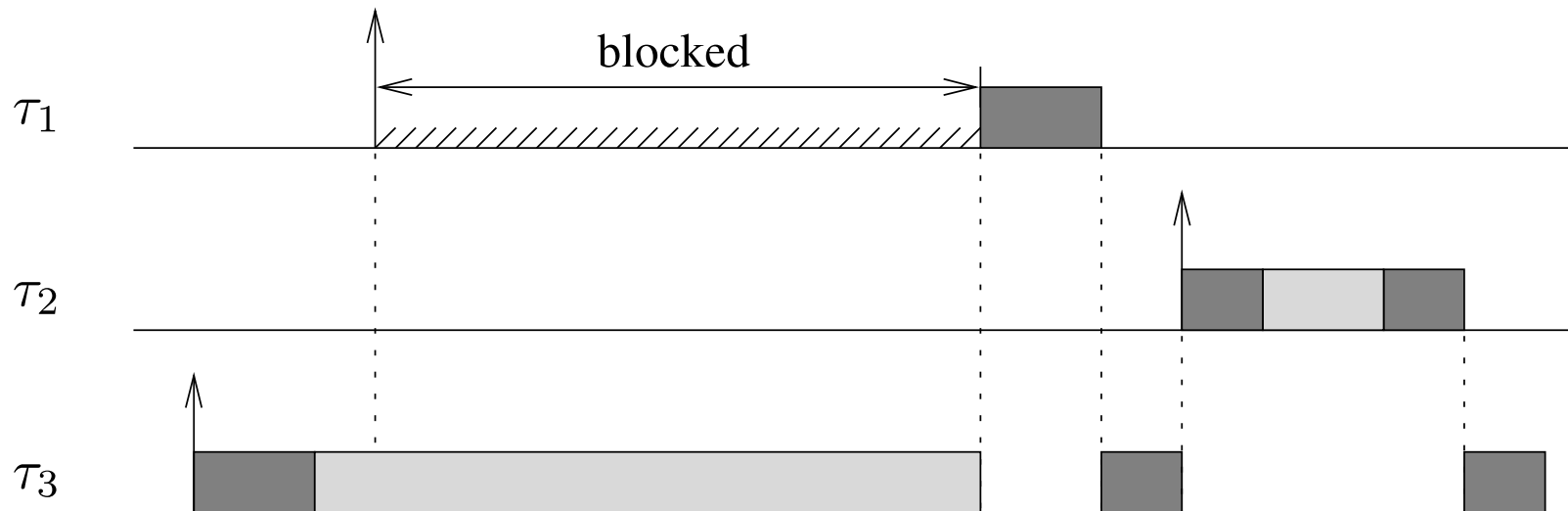
# EXAMPLE

- ▶ A high priority task may be blocked for long time although it does not need the exclusive resource

 normal execution

 critical section



$t_1$  remains blocked although it does not need the exclusive resource



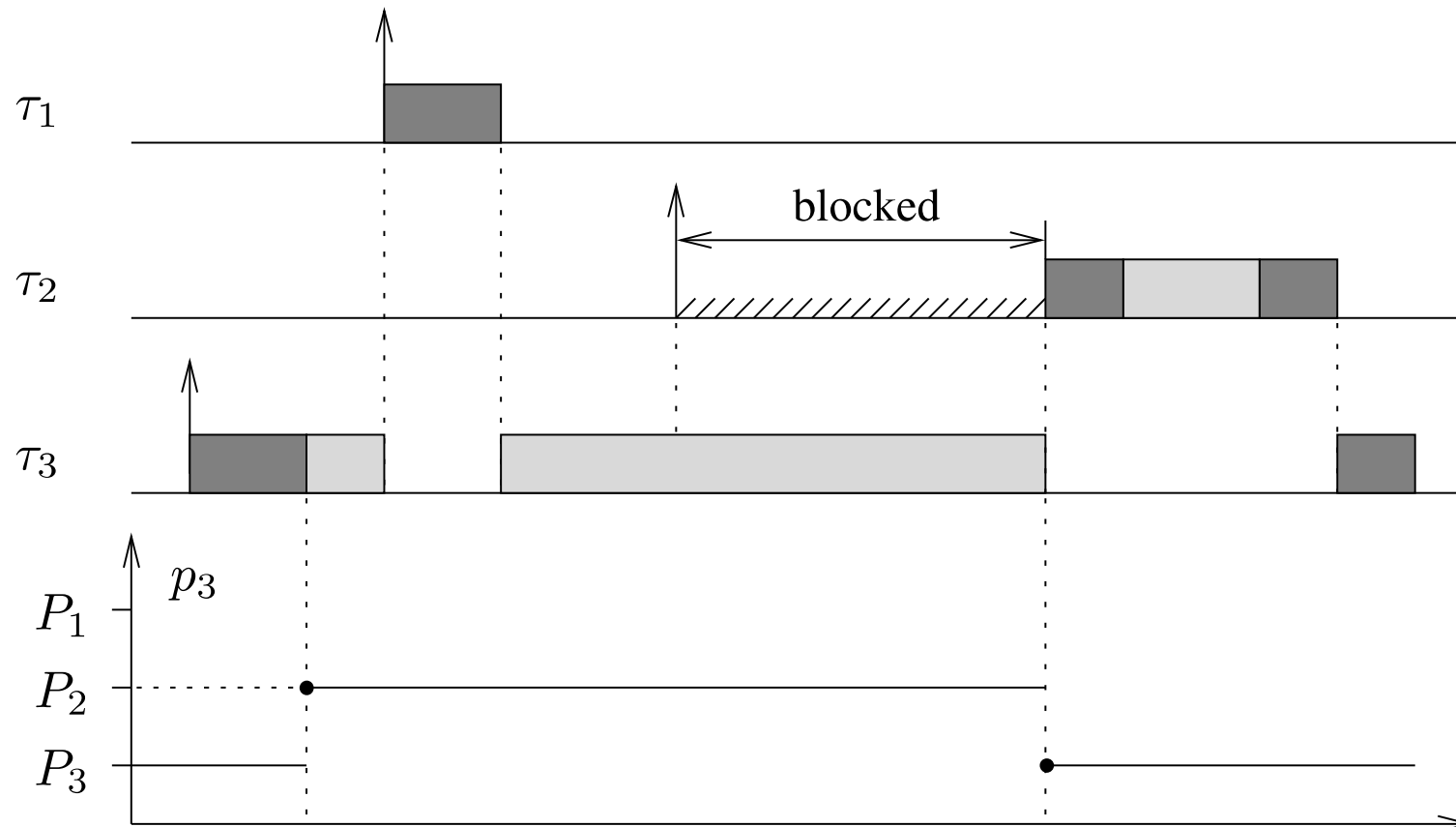
# IMMEDIATE PRIORITY CEILING (IPC)

- ▶ The approach
  - ▶ The priority of a task that enters the critical section for an exclusive resource R is set to the **highest priority among the tasks sharing that resource**
  - ▶ The priority is lowered to the initial value when the task exists the critical section

# EXAMPLE

 normal execution  
 critical section

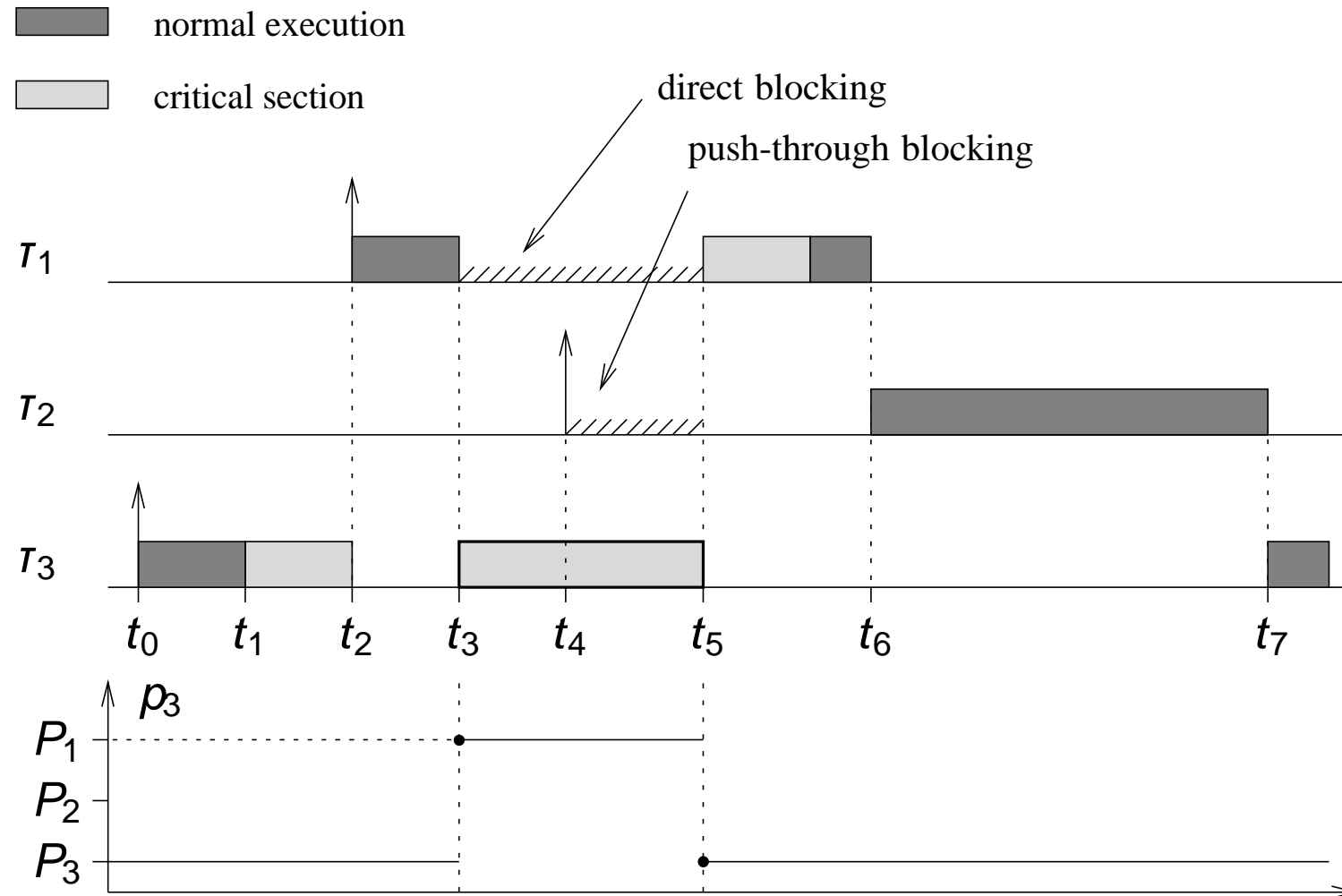
$t_1$  is not blocked as it does not need the exclusive resource.  $t_3$  get the priority of  $t_2$  when in critical section



# PRIORITY INHERITANCE PROTOCOL (PIP)

- ▶ The approach
  - ▶ When a task  $\tau_i$  blocks one or more higher-priority tasks, it temporarily assumes **(inherits) the highest priority of the blocked tasks**
  - ▶ This prevents medium-priority tasks from preempting  $\tau_i$  and prolonging the blocking duration experienced by the higher-priority tasks

# EXAMPLE



# FREE RTOS SYNCHRONIZATION PRIMITIVES

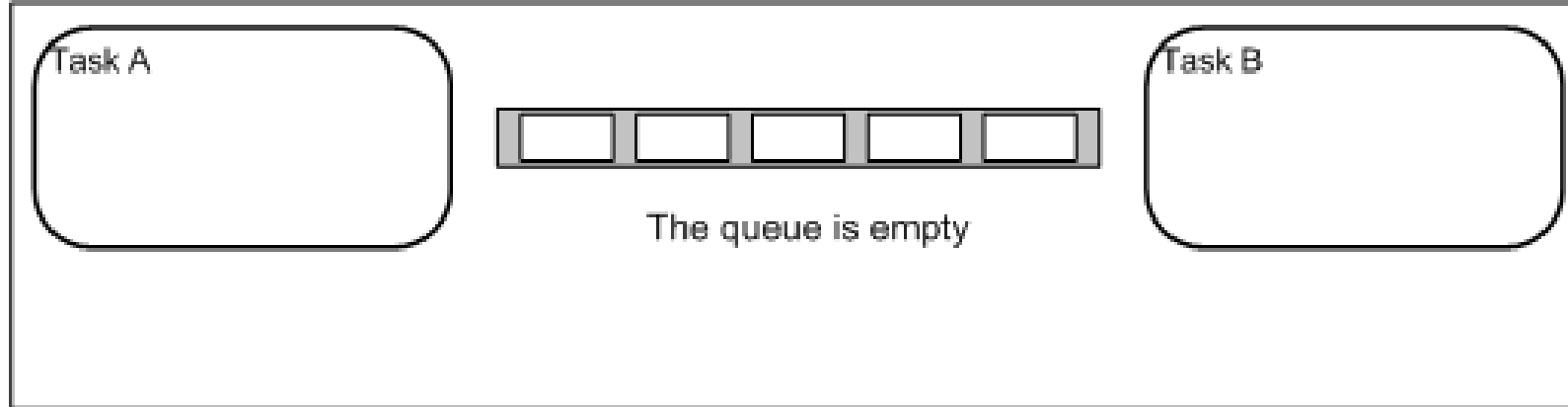
- ▶ Queue (<https://freertos.org/Documentation/02-Kernel/02-Kernel-features/02-Queues-mutexes-and-semaphores/01-Queues>)
- ▶ Binary Semaphores (<https://freertos.org/Documentation/02-Kernel/02-Kernel-features/02-Queues-mutexes-and-semaphores/02-Binary-semaphores>)
- ▶ Counting Semaphores (<https://freertos.org/Documentation/02-Kernel/02-Kernel-features/02-Queues-mutexes-and-semaphores/03-Counting-semaphores>)
- ▶ Mutexes (<https://freertos.org/Documentation/02-Kernel/02-Kernel-features/02-Queues-mutexes-and-semaphores/04-Mutexes>)
- ▶ Recursive Mutexes (<https://freertos.org/Documentation/02-Kernel/02-Kernel-features/02-Queues-mutexes-and-semaphores/05-Recursive-mutexes>)





# QUEUE

- ▶ Queues are the primary form of intertask communications. They can be used to send messages between tasks, and between interrupts and tasks. In most cases they are used as thread safe FIFO (First In First Out) buffers with new data being sent to the back of the queue, although data can also be sent to the front.



# BINARY SEMAPHORE

- ▶ Binary semaphores are used for both mutual exclusion and synchronization purposes.
- ▶ Binary semaphores and mutexes are very similar but have some subtle differences:
  - ▶ Mutexes include a priority inheritance mechanism, binary semaphores do not.
  - ▶ This makes binary semaphores the better choice for implementing synchronization (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.



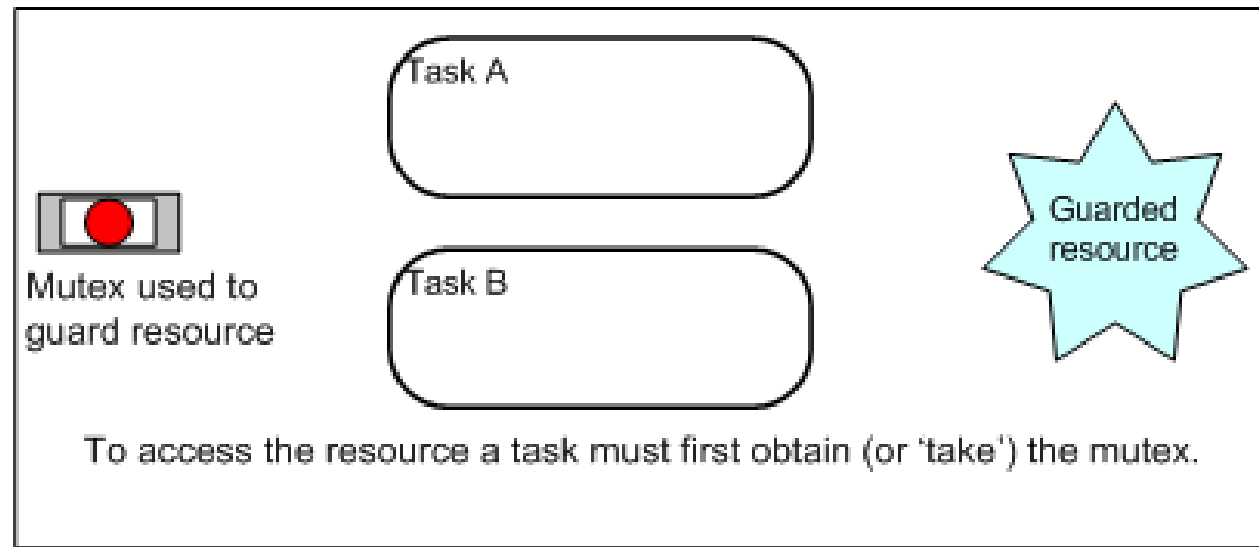
# COUNTING SEMAPHORE

- ▶ Just as binary semaphores can be thought of as queues of length one, counting semaphores can be thought of as queues of length greater than one.
- ▶ Users of the semaphore are not interested in the data that is stored in the queue - just whether the queue is empty or not.
- ▶ Counting semaphores are typically used for two things:
  - ▶ Counting events
  - ▶ Resource management.



# MUTEX

- ▶ Mutexes are binary semaphores that include a priority inheritance mechanism.
- ▶ Whereas binary semaphores are the better choice for implementing synchronization (between tasks or between tasks and an interrupt), mutexes are the better choice for implementing simple mutual exclusion (hence 'MUT'ual 'EX'clusion).



# SIMPLIFIED PRIORITY INHERITANCE

- ▶ FreeRTOS implements a basic priority inheritance mechanism which was designed to optimize both space and execution cycles.
- ▶ A full priority inheritance mechanism requires significantly more data and processor cycles to determine inherited priority at any moment, especially when a task holds more than one mutex at a time.
- ▶ Keep in mind these specific behaviors of the priority inheritance mechanism:
  - ▶ A task can have its inherited priority raised further if it takes a mutex without first releasing mutexes it already holds.
  - ▶ A task remains at its highest inherited priority until it has released all the mutexes it holds. This is regardless of the order the mutexes are released.
  - ▶ A task will remain at the highest inherited priority if multiple mutexes are held regardless of tasks waiting on any of the held mutexes completing their wait (timing out).



**QUESTIONS?**

**THANK YOU!**

