# Embedded Systems

## Lecture 7

## Dynamic Scheduling - Real Time Operating System (RTOS)

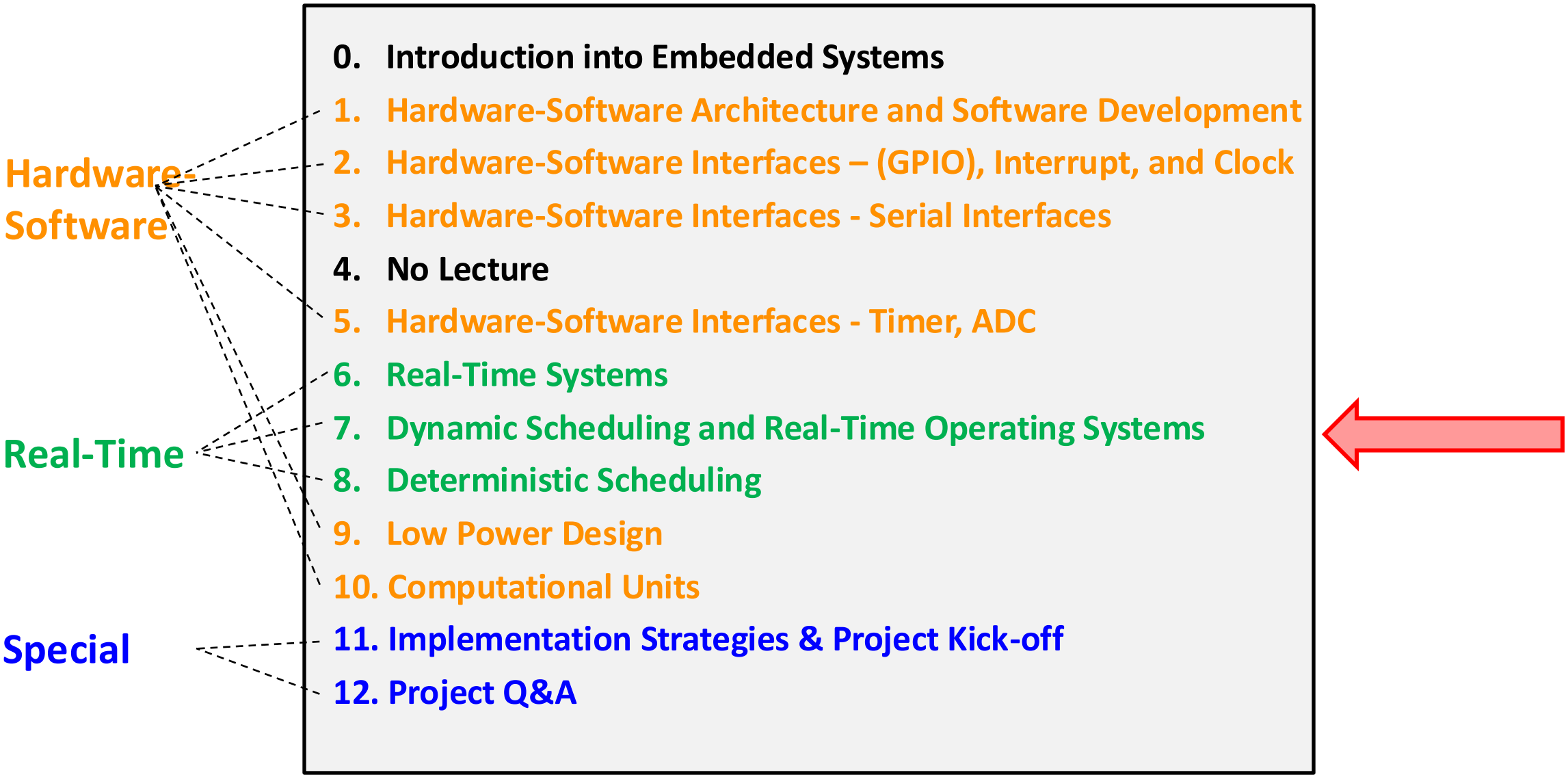
© Michele Magno

D-ITET Center for Project-Based Learning

Credits: Lothar Thiele

ETH zürich

# Where We are


Hardware-Software
Real-Time
Special
0. Introduction into Embedded Systems
1. Hardware-Software Architecture and Software Development
2. Hardware-Software Interfaces – (GPIO), Interrupt, and Clock
3. Hardware-Software Interfaces - Serial Interfaces
4. No Lecture
5. Hardware-Software Interfaces - Timer, ADC
6. Real-Time Systems
7. Dynamic Scheduling and Real-Time Operating Systems
8. Deterministic Scheduling
9. Low Power Design
10. Computational Units
11. Implementation Strategies & Project Kick-off
12. Project Q&A

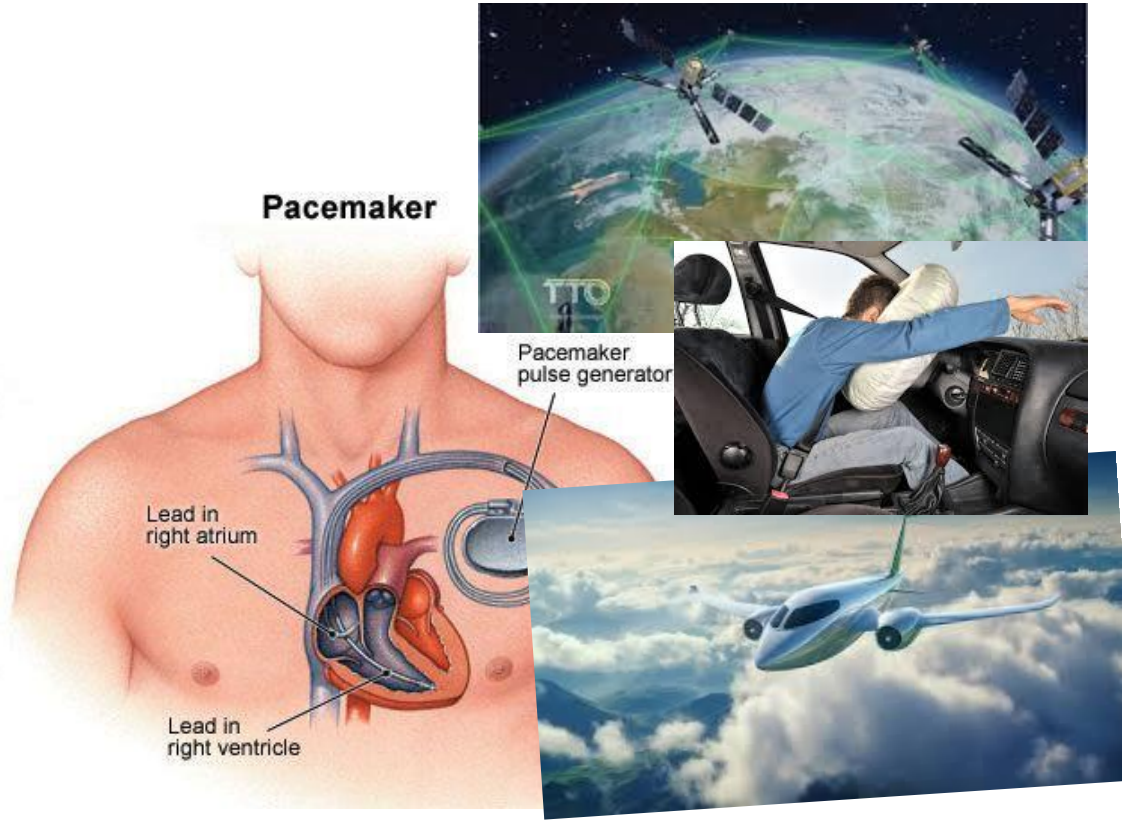
# Today's topics

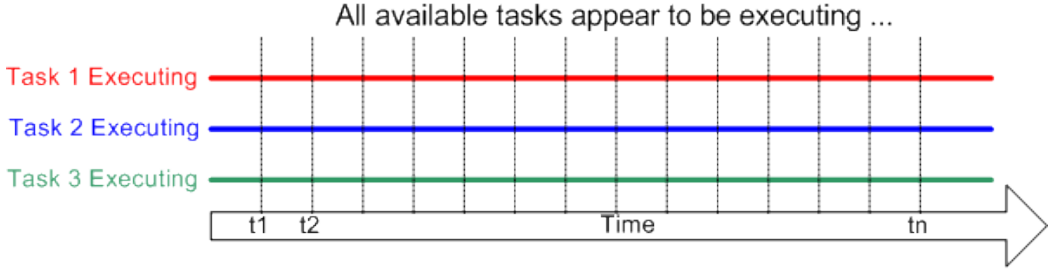
- Embedded Operating System (FreeRTOS)
- Task Scheduling
- Resource Sharing (Semaphore; Mutexes)
- Priority Inheritance Protocol (PIP)
- Timing Anomalies
  - Richard's Anomalies for multicores
- Communication and Synchronization (Queues)

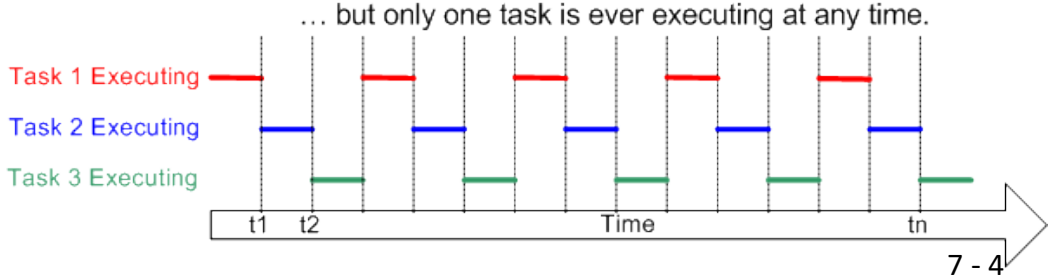
# RTOS Fundamentals:

## Overview of Real-Time Operating Systems

- **Goal:** Their primary objective is to ensure a timely and deterministic response to events. An event can be external, like a limit switch being hit, or internal like a character being received.
- **Multitasking**
  - an operating system can execute multiple threads in this manner it is said to be multitasking.
- **Multitasking Vs Concurrency**
  - A single core processor can only execute a single task at a time - but by rapidly switching between tasks a multitasking operating system appear as if each task is executing concurrently
- **Scheduling**
  - **In a RTOS** The scheduler is the part of the kernel responsible for deciding which task should be executing at any particular time.



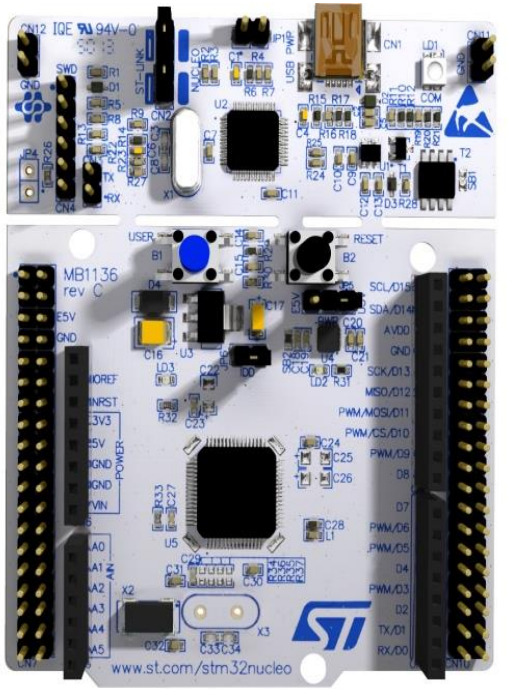



# Example: FreeRTOS(Exercises)

***FreeRTOS*** (https://www.freertos.org/) is a community-driven embedded operating system. It is available for most hardware platforms, open source, and widely used in industry. It will be used in the exercises.

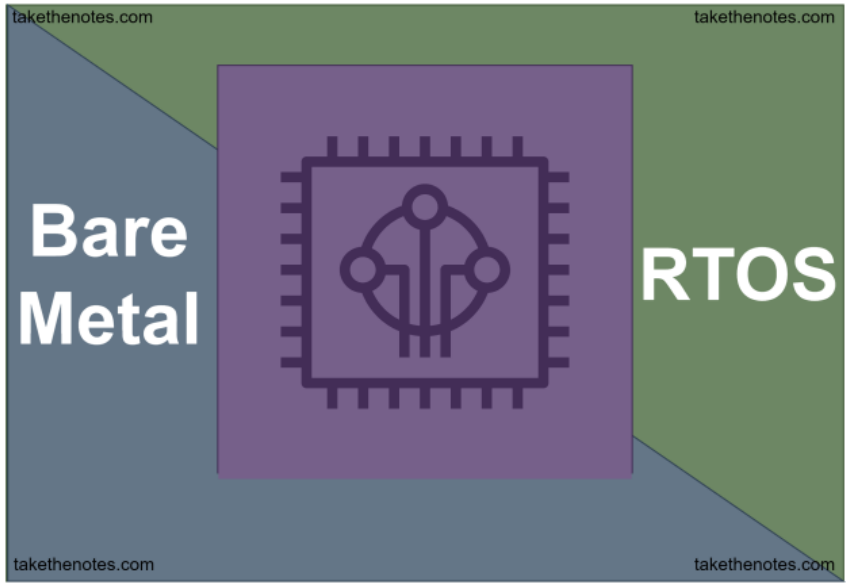
- Kernel  provides an "abstraction layer" that hides the hardware details of the processor from the application software
- FreeRTOS is a *high-performance real-time kernel*.




- Applications are organized as a *collection of independent threads* of execution.

- *Characteristics:* Preemptive and cooperative scheduling, flexible thread priority support, priority inheritance, automatic scaling, deterministic processing, run-time performance monitoring, ...

# Why do we need a RTOS?




- **Simplified Complex Task Management**
  - **Automated Scheduling and Prioritization**: RTOS handles task switching, scheduling, and priority management, which would be more **manual and error-prone on bare metal.**
- **Reliability in Time-Critical Applications**
  - Deterministic Response: RTOS ensures predictable response times for critical tasks, important in systems where timing is essential (e.g., medical devices, automotive systems).
- **Built-in Task Synchronization and Communication**
  - Semaphores, Queues, and Mutexes: RTOS provides ready-made tools for managing shared resources and inter-task communication, reducing complexity and potential bugs.
- **Modular and Scalable Code Structure**
  - Task-Based Design: RTOS encourages dividing code into independent tasks, making it easier to update, test, and scale compared to the often monolithic structure in bare metal.
- **Reduced Developer Workload and Errors**
  - Less Low-Level Code: RTOS handles many low-level tasks (e.g., context switching) automatically, allowing developers to focus on application logic rather than managing the hardware directly.

# Embedded Operating System

- *Why is a desktop OS not suited?*
  - Desktop OS offers too many features that take space in memory and consume time.
    - Requires too much memory space and is often too resource hungry in terms of computation time.
  - The OS are often not modular, fault-tolerant, configurable.
  - Not designed for mission-critical applications.
  - The timing uncertainty may be too large for some applications.

*One Essential characteristics of an Embedded OS:* **Configurability**

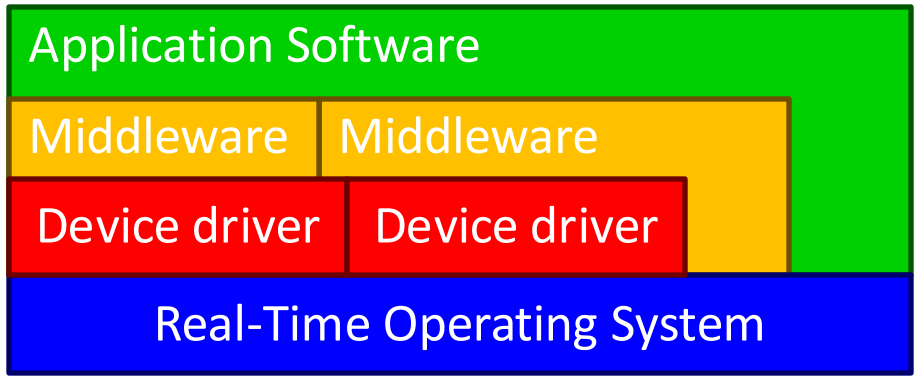
- *No single operating system will fit all needs -* unused functions/data is not tolerated.
  - For example, many embedded systems come  a keyboard, a screen or a mouse.
- *Remove unused functions*/libraries (for example by the linker).
- *Use conditional compilation* (using #if and #ifdef commands in C, for example).

# Embedded Operating System vs. Standard OS

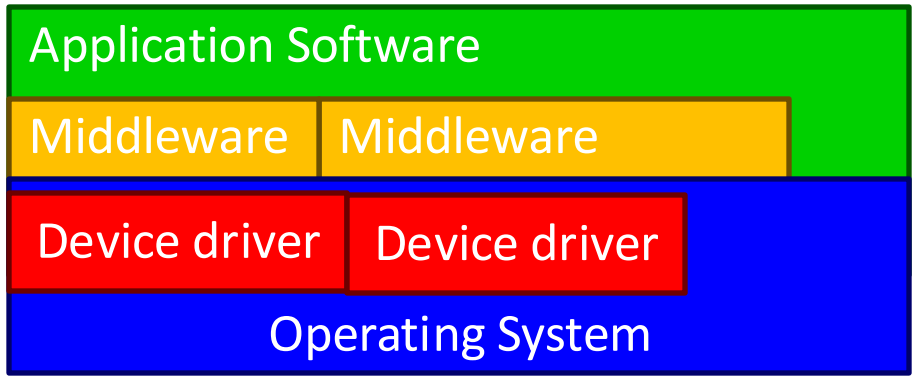
*Device drivers are typically handled directly by tasks* instead of drivers that are managed by the operating system:

- This architecture *improves timing predictability* as access to devices is also handled by the scheduler
- If several tasks use the same external device and the associated driver, then the access must be carefully managed (shared critical resource, ensure fairness of access)

Embedded OS

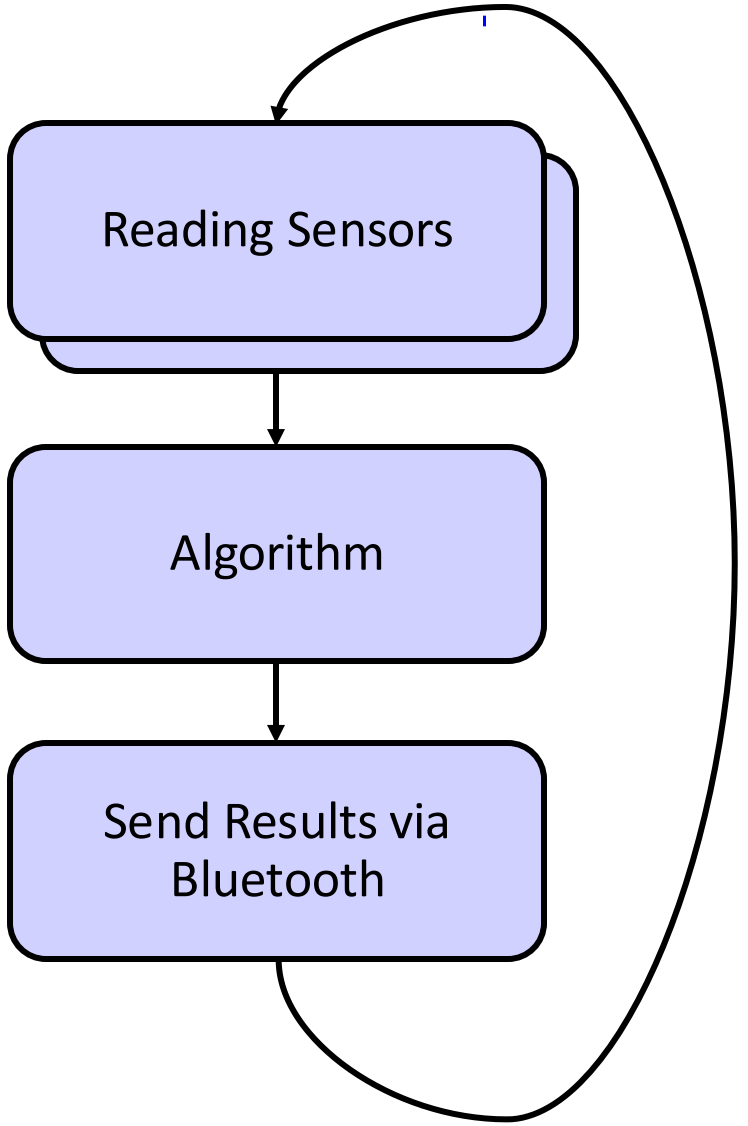



Standard OS



# Embedded Operating Systems

## FreeRTOS Task Management



# FreeRTOS – Task Management

***Tasks are implemented as threads.***

- The *functionality of a thread* is implemented in form of a *function:*
  - Prototype: **`void task1_entry(void *args);`**

some name of task function — pointer to task arguments

  - Task functions can instantiate other tasks. Each created task is a separate execution instance, with its own stack.
  - *Example*

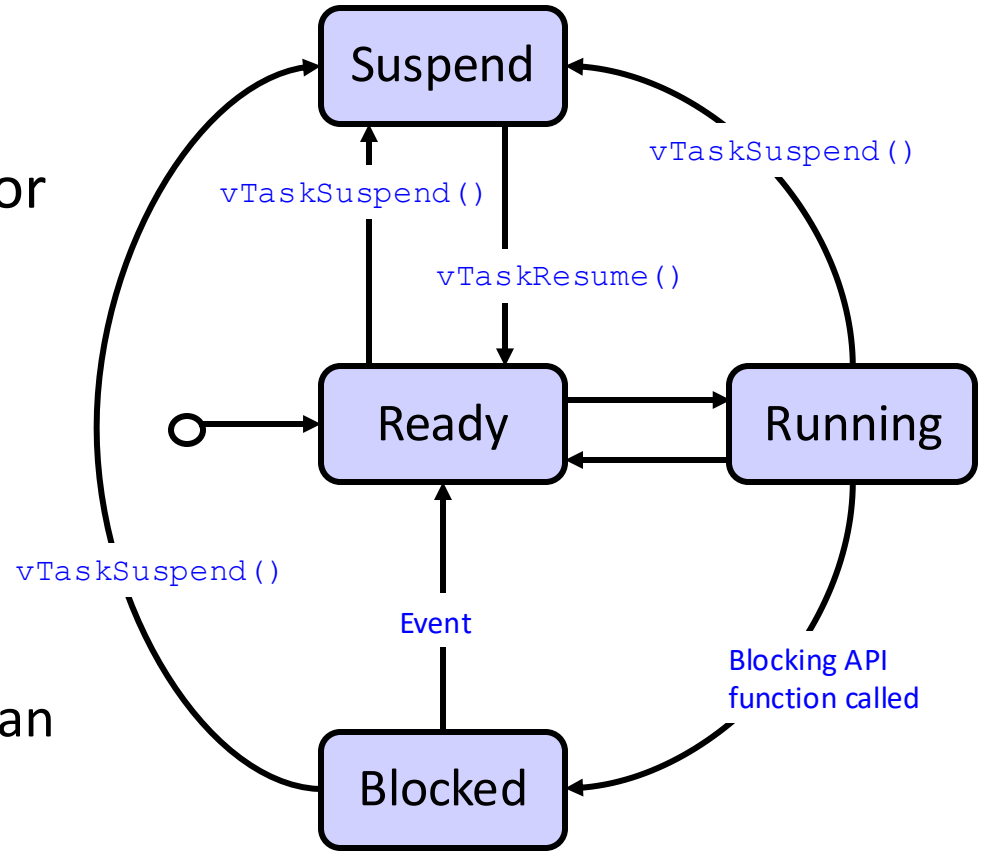
```
void task1_entry(void *args) {
   volatile uint32_t ul; /* volatile to ensure ul is implemented. */
   for( ;; ) {
     ... /* do something repeatedly */
     for( ul = 0; ul < 10000; ul++ ) {  /* delay by busy loop */ }
   }
}
```

# FreeRTOS – Task States

*What are the task states in FreeRTOS and the corresponding transitions?*

- A task that is waiting for an event is said to be in the *"suspended" state*.

- Tasks can enter the "suspended" state to wait for mainly two different types of event:
  - *Temporal (time-related) events*—the event being either a delay period expiring, or an absolute time being reached.
  - *Synchronization events*—where the events originate from another task or interrupt. For example, queues, semaphores, and mutexes, can be used to create synchronization events.




*Taken from FreeRTOS documentation

# FreeRTOS – Task Priorities

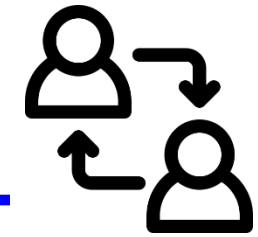
*Each Task has a Priority assigned, from P = 0 (lowest priority) to P = max (highest priority).*

- A Task with a higher Priority will run first.

- The maximum priority value for an application can be set in the configuration file.

# FreeRTOS – Task Priorities

Discuss 1 min



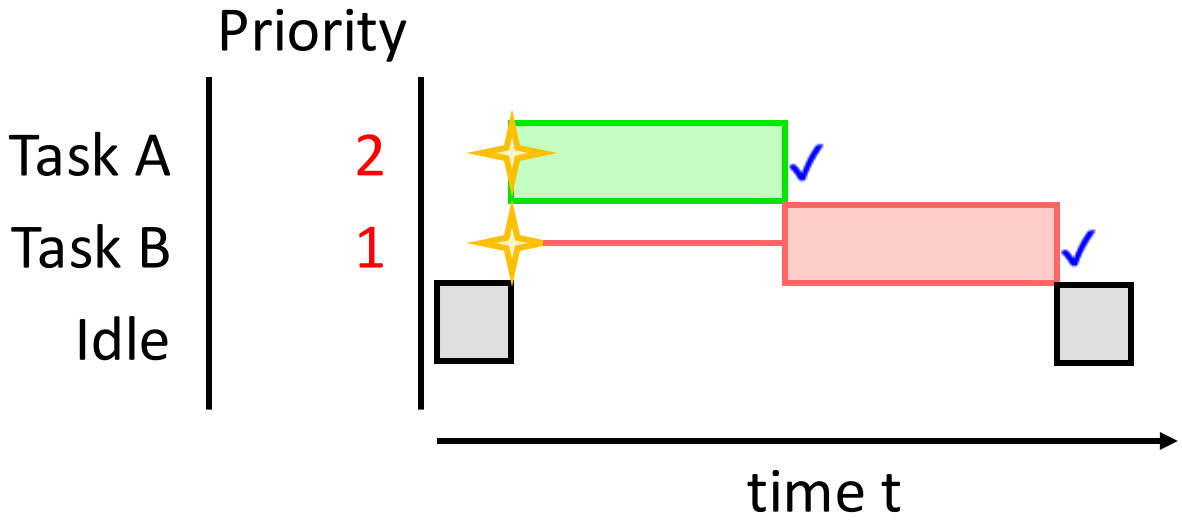
*Each Task has a Priority assigned, from P = 0 (lowest priority) to P = max (highest priority).*

- A Task with a higher Priority will run first.

- The maximum priority value for an application can be set in the configuration file.


Task Ready

✓ Suspended/Blocked/Deleted

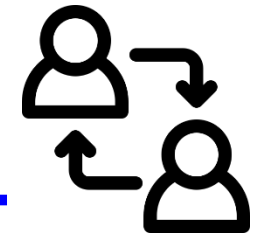



**Which Task finished first?**

**Task A**

# FreeRTOS – Task Priorities

Discuss 1 min



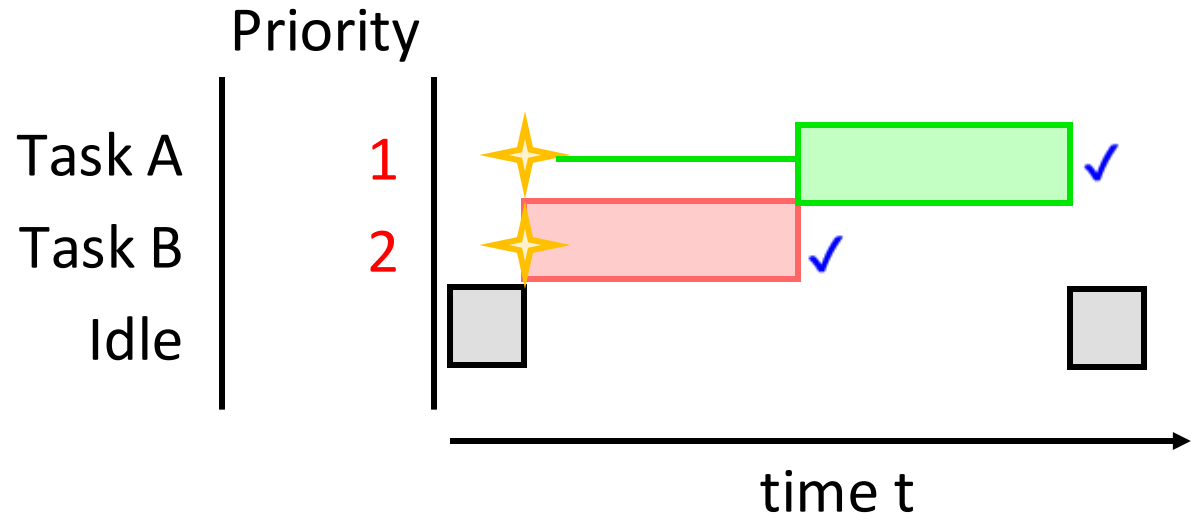
*Each Task has a Priority assigned, from P = 0 (lowest priority) to P = max (highest priority).*

- A Task with a higher Priority will run first.

- The maximum priority value for an application can be set in the configuration file.

Task Ready

✓ Suspended/Blocked/Deleted




**Which Task finished first?**

**Task B**

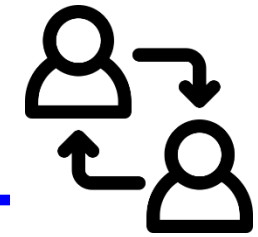
# FreeRTOS – Task Preemption

*Interrupting a running task, such that a task with higher priority becomes ready.*

- A Task with a higher Priority can interrupt a running task.
- This feature can be turned on/off in the configuration files.

# FreeRTOS – Task Preemption

Discuss 1 min



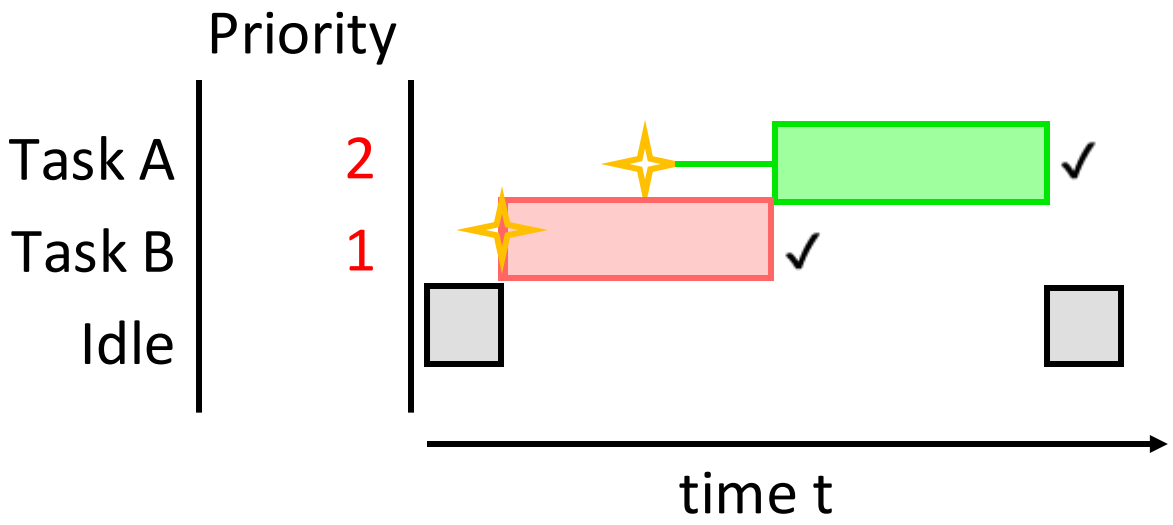
*Interrupting a running task, such that a task with higher priority becomes ready.*

- A Task with a higher Priority can interrupt a running task.
- This feature can be turned on/off in the configuration files.

 Task Ready

✓ Suspended/Blocked/Deleted

**Without preemption**

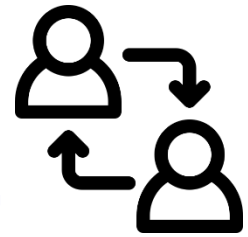



**Which Task finished first?**

**Task B**

# FreeRTOS – Task Preemption

Discuss 1 min



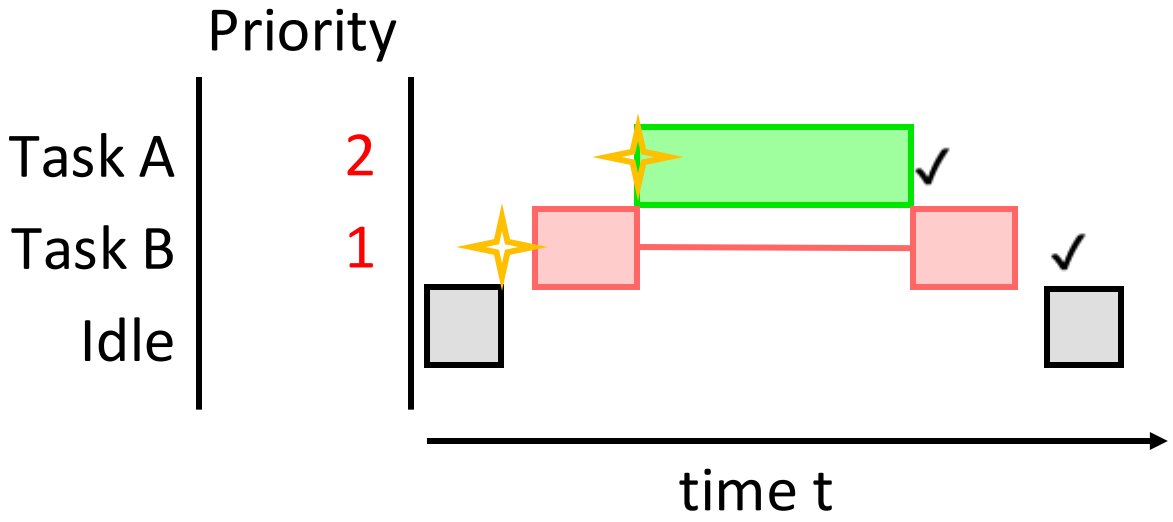
*Interrupting a running task, such that a task with higher priority becomes ready.*

- A Task with a higher Priority can interrupt a running task.
- This feature can be turned on/off in the configuration files.

Task Ready

✓ Suspended/Blocked/Deleted

**With preemption**




**Which Task finished first?**

**Task A**

# FreeRTOS – Task Time Slicing

*Handling tasks of equal priority which are ready at the same time.*

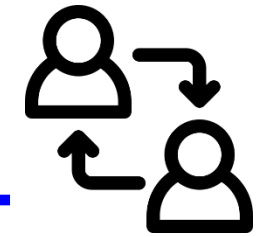
- The scheduler jumps between tasks.
- This feature can be turned on/off
  in the configuration files.

time t

time t

# FreeRTOS – Task Time Slicing

Discuss 1 min



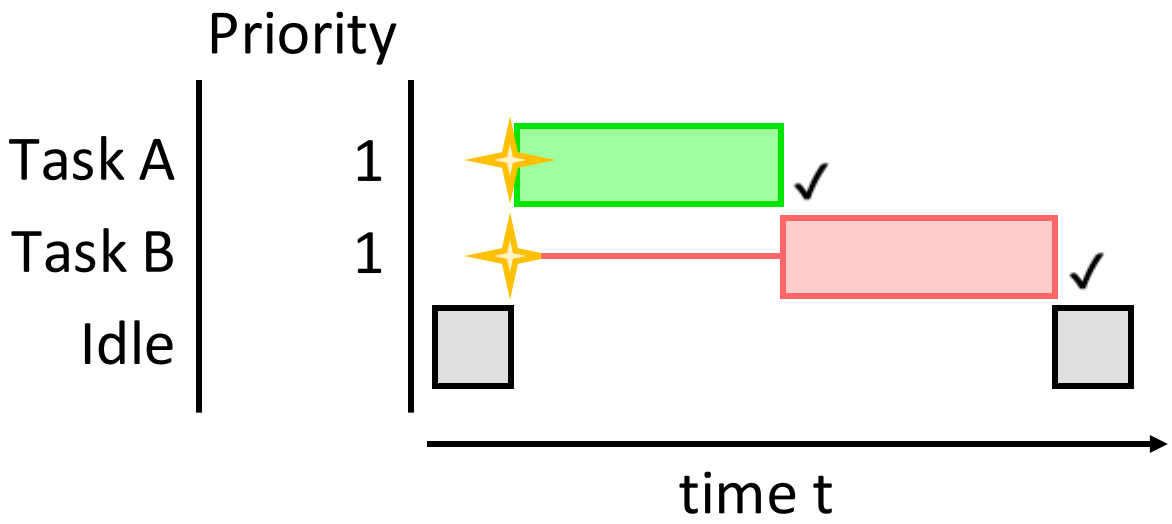
*Handling tasks of equal priority which are ready at the same time.*

- The scheduler jumps between tasks.
- This feature can be turned on/off
  in the configuration files.


Task Ready

✓ Suspended/Blocked/Deleted

**Without time slicing**



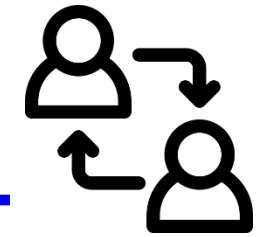

**Which Task finished first?**

**Task A**

time t

# FreeRTOS – Task Time Slicing

Discuss 1 min



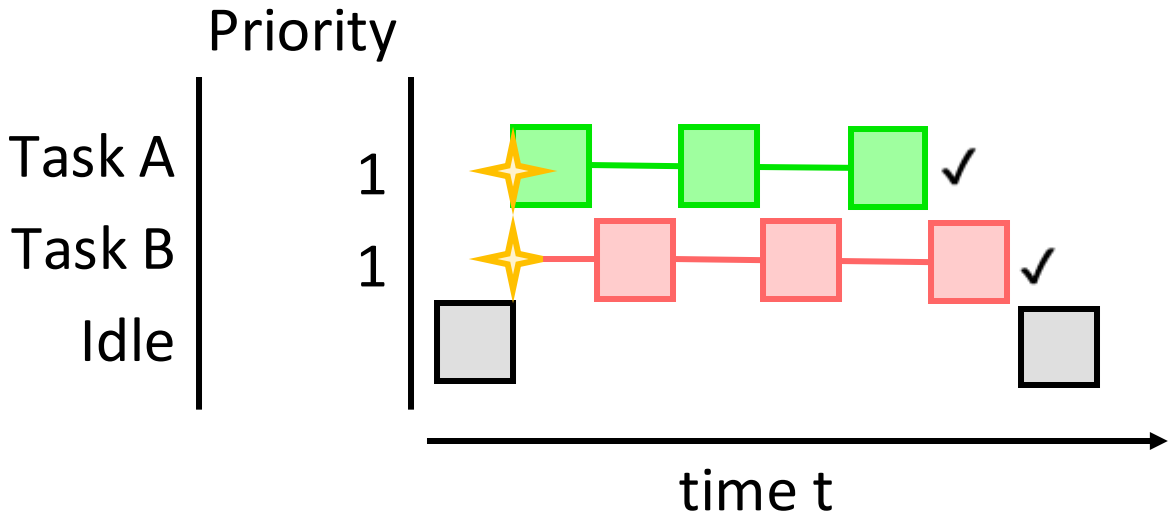
*Handling tasks of equal priority which are ready at the same time.*

- The scheduler jumps between tasks.
- This feature can be turned on/off
  in the configuration files.

✦ Task Ready

✓ Suspended/Blocked/Deleted

**With time slicing**




**Which Task finished first?**

**Task A/B - fair!**

time t

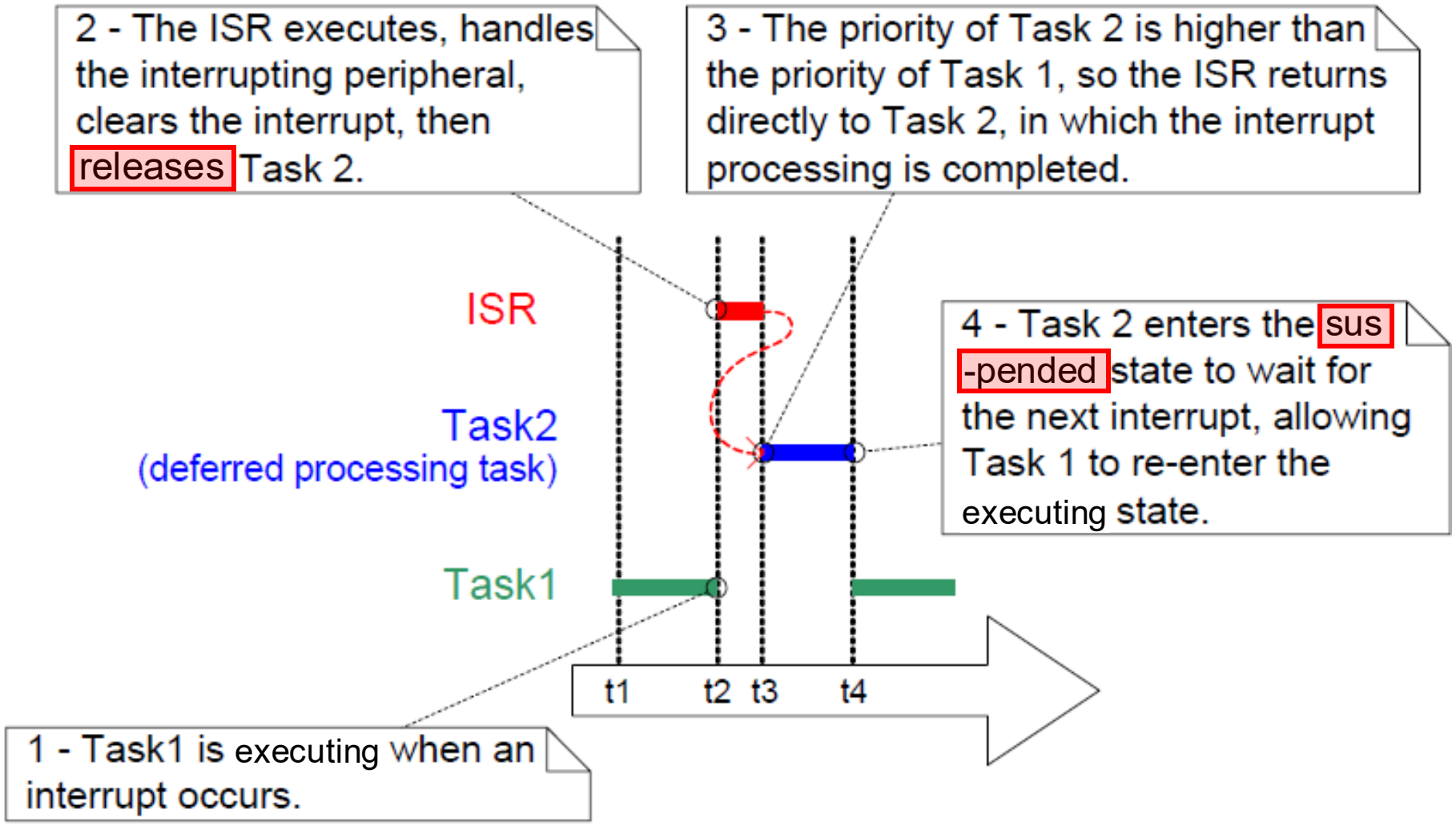
# FreeRTOS – Interrupts

*How are tasks (threads) and hardware interrupts scheduled jointly?*

- Although written in software, an *interrupt service routine (ISR)* is a hardware/software feature because the hardware controls which interrupt service routine will run, and when it will run.
- *Tasks will only run when there are no ISRs running*, so the lowest priority interrupt will interrupt the highest priority task, and there is no way for a task to pre-empt an ISR. In other words, ISRs have always a higher priority than any other task.

- *Usual pattern:*
  - ISRs are usually very short. They find out the reason for the interrupt, clear the interrupt flag and determine what to do in order to handle the interrupt.
  - Then, they unblock a regular task (thread) that performs the necessary processing related to the interrupt.
  - For blocking and unblocking, usually semaphores are used.

# FreeRTOS – Interrupts



# Resource Sharing

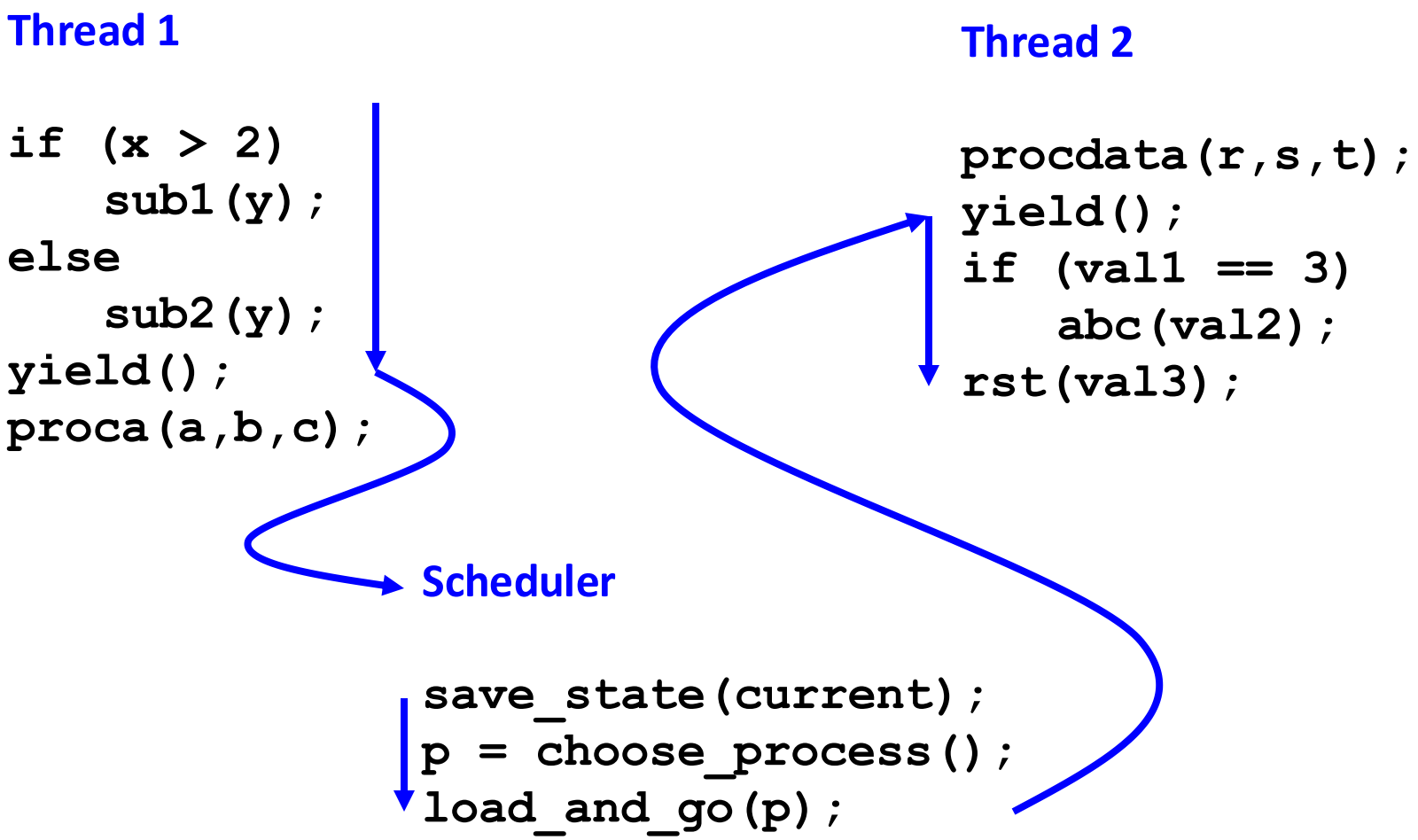
# Co-operative Multitasking

- *Each thread allows a context switch to another thread* by voluntarily yielding the control to another process.
  - This function is part of the underlying runtime system (operating system).
  - A *scheduler* within this runtime system chooses which thread will run next.

- ***Advantages***:
  - predictable, where context switches can occur
  - less errors with use of shared resources if the switch locations are chosen carefully

- ***Problems***:
  - programming errors can keep other threads out as a thread may never give up CPU
  - real-time behavior may be at risk if a thread runs too long before the next context switch is allowed

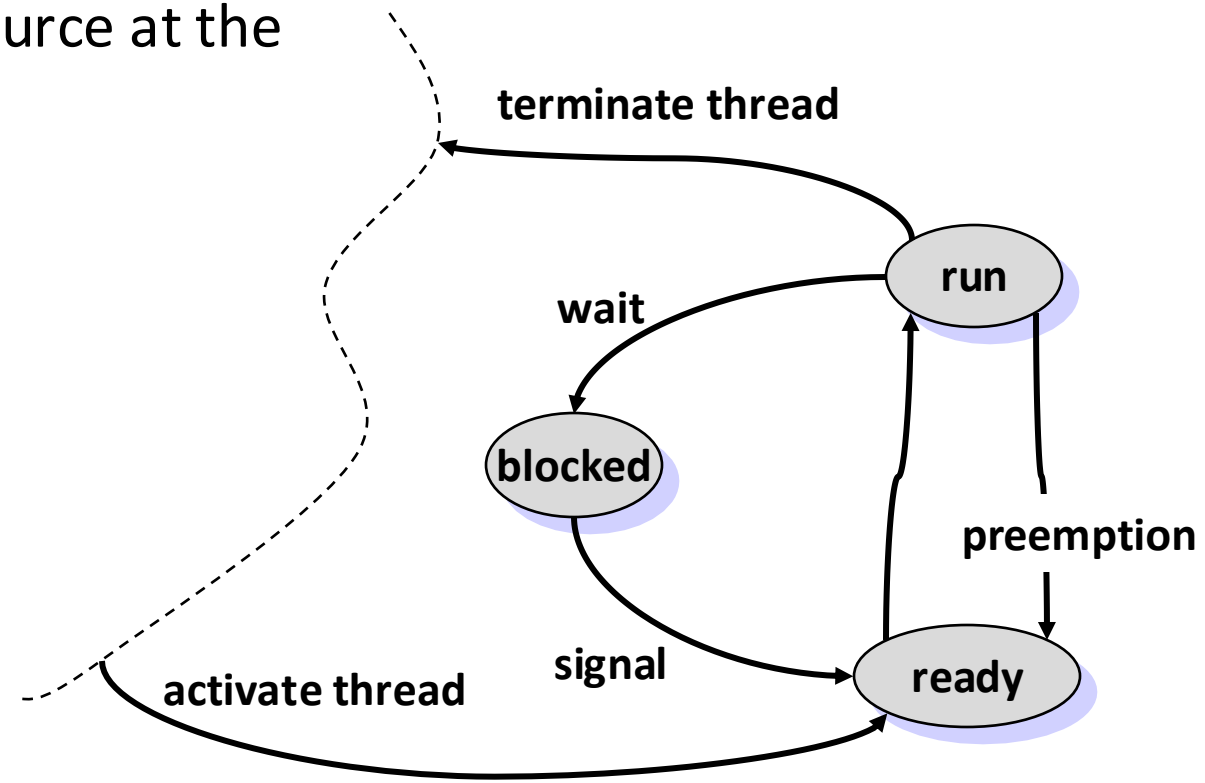
# Example: Co-operative Multitasking



# Resource Sharing

- Examples of *shared resources*: data structures, variables, main memory area, file, set of registers, I/O unit, … .
- Many shared resources do not allow simultaneous accesses but require *mutual exclusion*. These resources are called *exclusive resources*. In this case, no two threads are allowed to operate on the resource at the same time.

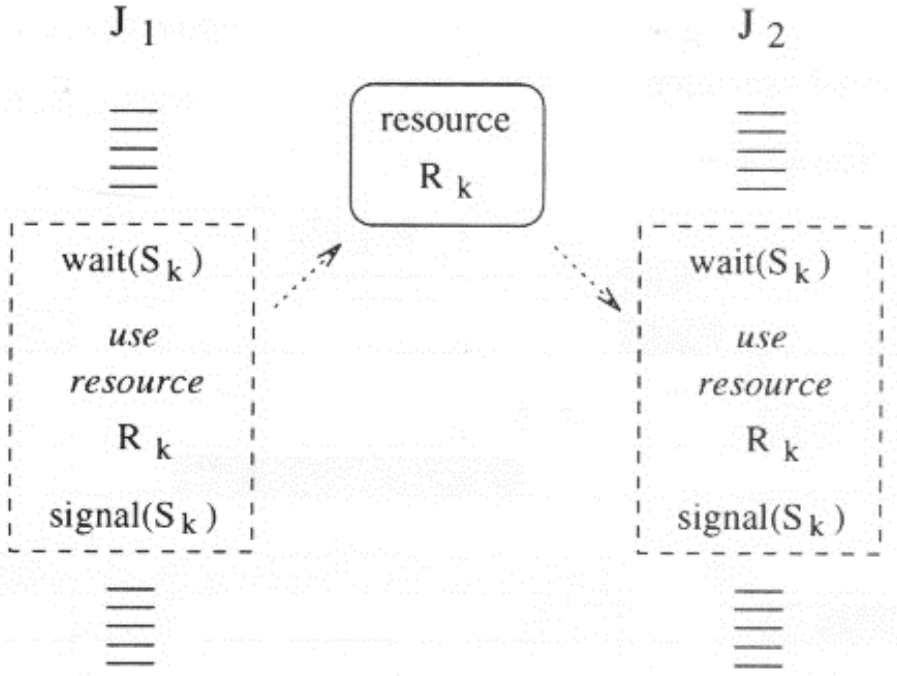
- There are several methods available to *protect exclusive resources*, for example
  - *disabling interrupts* and preemption or
  - using concepts like *semaphores and mutex* that put threads into the blocked state if necessary.



# Protecting Exclusive Resources using Semaphores

- Each *exclusive resource* $R_i$ must be protected by a different *semaphore* $S_i$ . Each critical section operating on a resource must begin with a `wait(`$S_i$`)` primitive and end with a `signal(`$S_i$`)` primitive.
  - A **job** (like **J1** or **J2**) is a specific instance or activation of a task.

    in this lecture we assume Task = Job.




- All tasks blocked on the same resource are kept in a **queue associated with the semaphore.** When a running task executes a *wait* on a *locked semaphore*, it enters a *blocked state*, until another tasks executes a *signal* primitive that *unlocks the semaphore*.

# FreeRTOS - Mutex

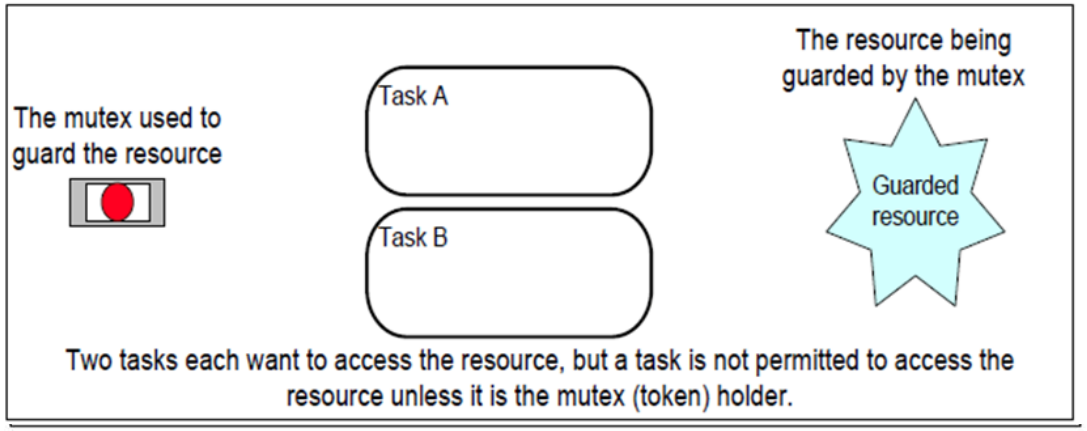
*Another possibility is to use mutual exclusion:* In FreeRTOS, a *mutex* is a special type of *semaphore* that is used to *control access* to a resource that is shared between two or more tasks. *A semaphore that is used for mutual exclusion must always be returned:*

- When used in a mutual exclusion scenario, the mutex can be thought of as a token that is associated with the resource being shared.
- For a task to access the resource legitimately, it must first successfully 'get' the token (be the token holder). When the token holder has finished with the resource, it must 'put' the token back.
- Only when the token has been returned can another task successfully take the token, and then safely access the same shared resource.

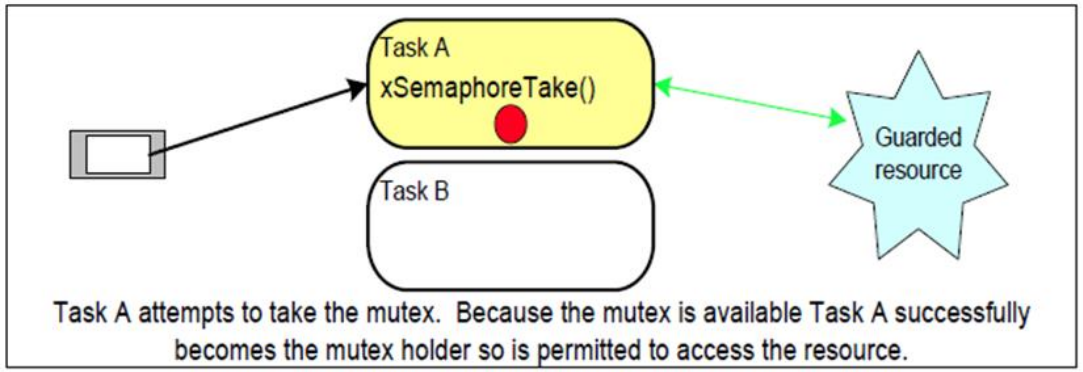
# FreeRTOS – Semaphore vs. Mutex

| | Semaphore | Mutex |
|---|---|---|
| **Purpose** | Used for **signaling** between tasks or tasks and interrupts | **Protect shared resources** (mutual-exclusion) |
| **Ownership** | **No ownership**. Any task can release a semaphore, regardless of which task obtained it. | Has **ownership**. Only task that owns the mutex can release it. |
| **Priority Inheritance** | Does not support priority inheritance. Can can lead to issues if not managed carefully. | Supports priority inheritance. temporarily boosts the priority of the task holding the mutex to match that of the highest-priority and waiting task. |
| **Use-case** | More suitable for event signaling and synchronization between tasks or between tasks and interrupts, such as indicating that an item has been produced or consumed. | Should be used when tasks need exclusive access to resources (like global variables, data structures, or hardware peripherals). |

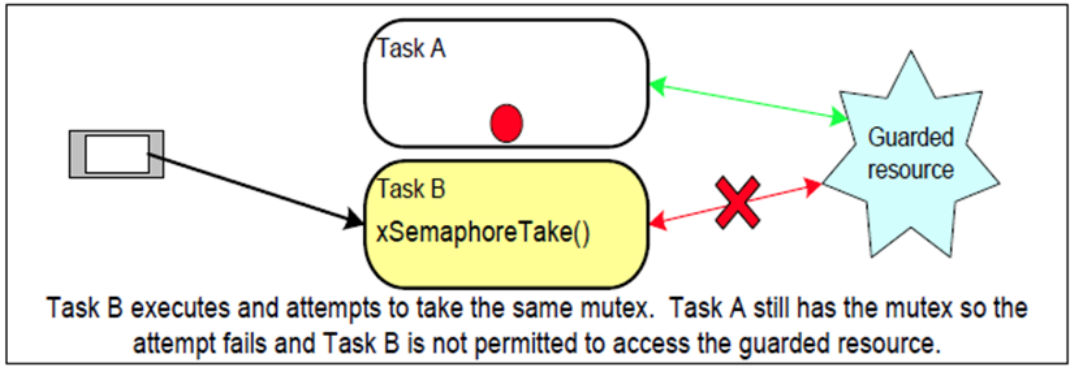
# FreeRTOS - Example Mutex



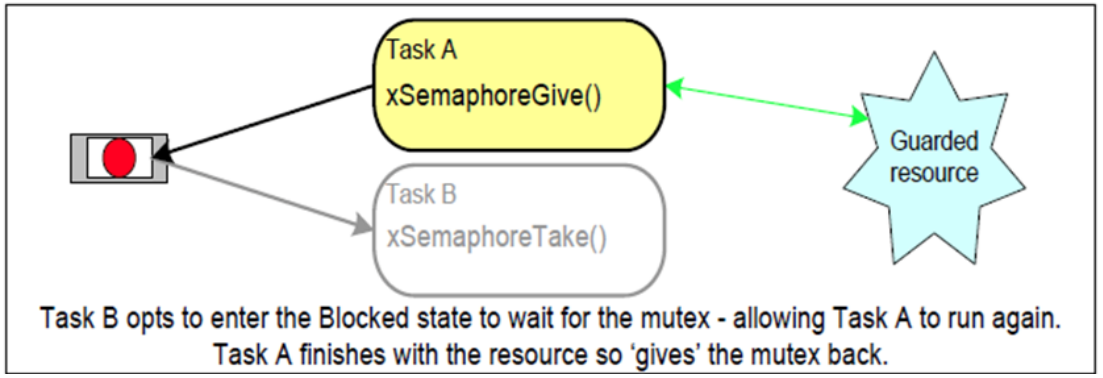

Two tasks each want to access the resource, but a task is not permitted to access the resource unless it is the mutex (token) holder.



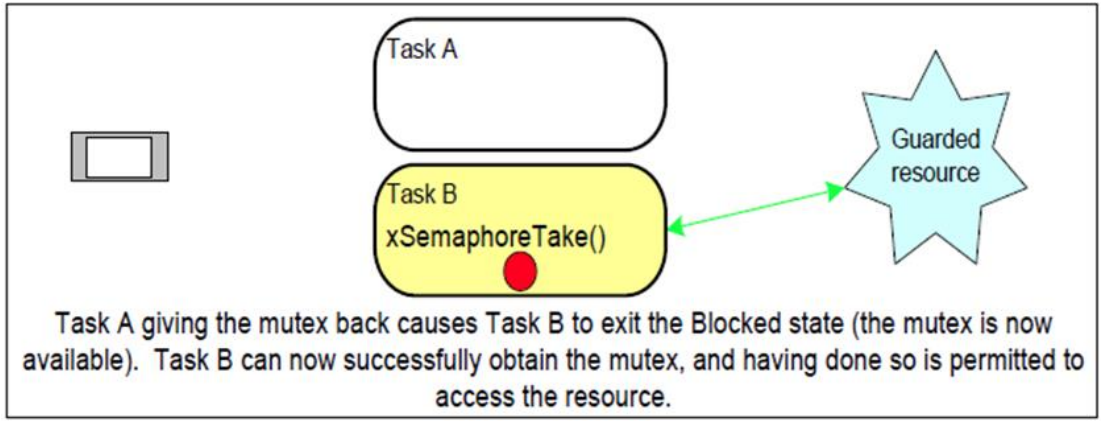

Task A attempts to take the mutex. Because the mutex is available Task A successfully becomes the mutex holder so is permitted to access the resource.



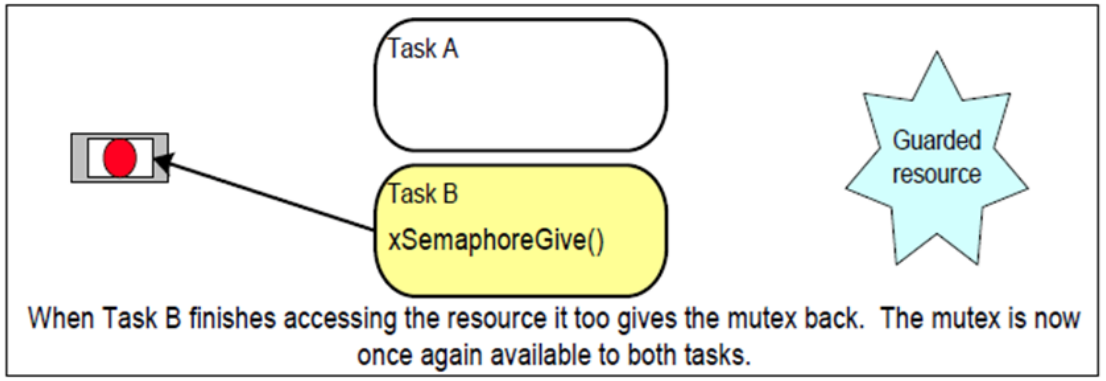

Task B executes and attempts to take the same mutex. Task A still has the mutex so the attempt fails and Task B is not permitted to access the guarded resource.




Task B opts to enter the Blocked state to wait for the mutex - allowing Task A to run again. Task A finishes with the resource so 'gives' the mutex back.




Task A giving the mutex back causes Task B to exit the Blocked state (the mutex is now available). Task B can now successfully obtain the mutex, and having done so is permitted to access the resource.

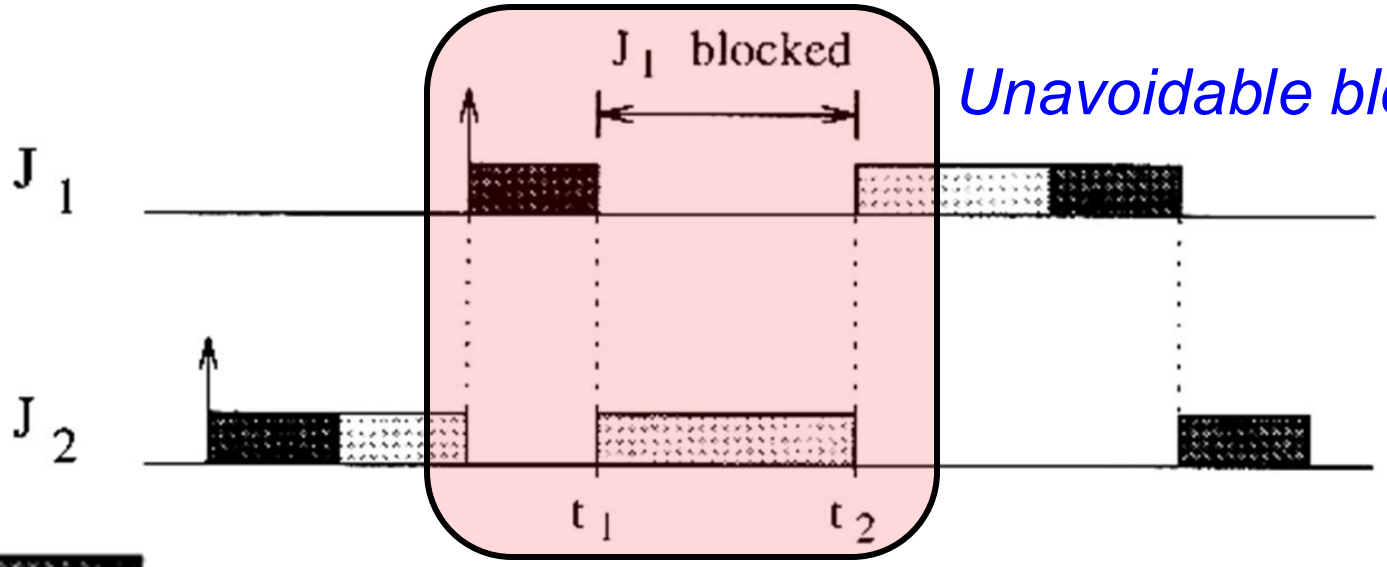



When Task B finishes accessing the resource it too gives the mutex back. The mutex is now once again available to both tasks.

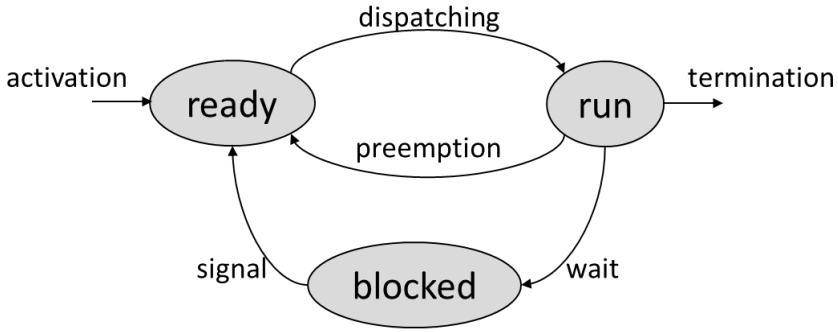
# Ressource Sharing

## Priority Inversion

# Priority Inversion (1): J1 with higher Priority of J2




*Unavoidable blocking*



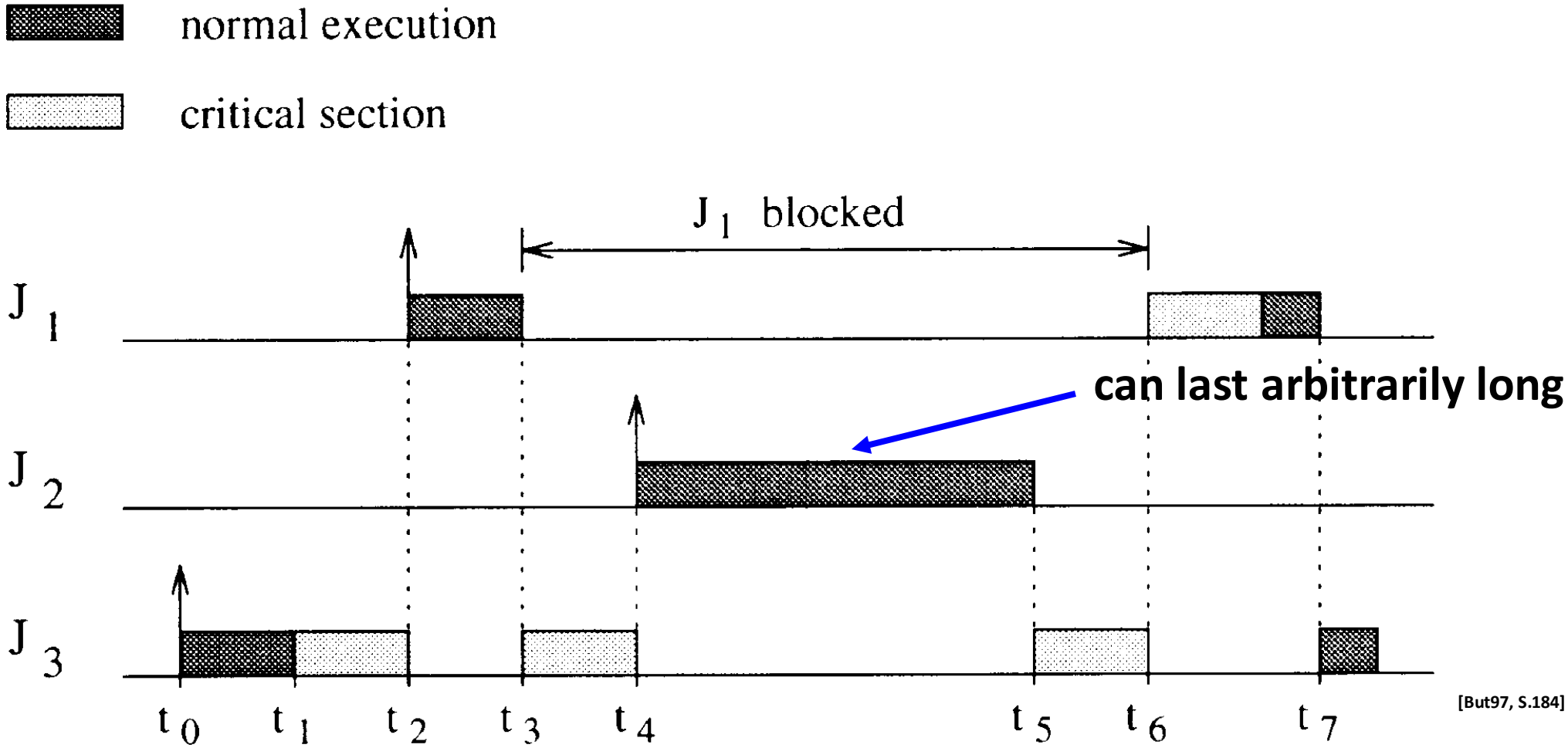

***Normal Execution:*** ***P****eriods where a job is executing regular tasks that do not involve the shared resource.*

***Critical Section****: Periods where a job is using a shared resource that requires mutual exclusion. Only one job can enter this section at a time.*

***Blocked State****: When a job tries to enter the critical section but finds it occupied, it becomes blocked and waits until the resource is released.*

# Priority Inversion (2): J2 does not need the shared resource

***Priority Inversion:***

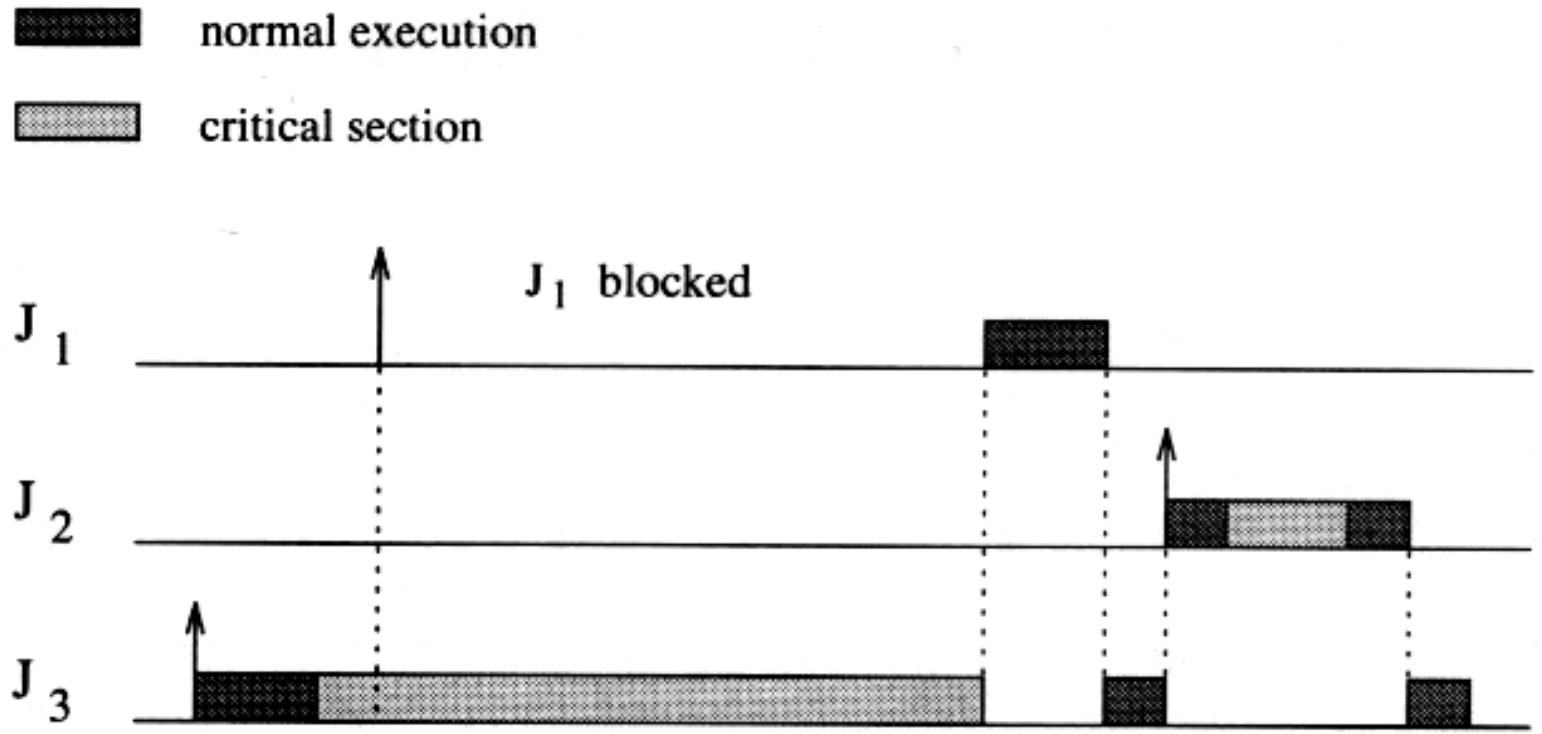



[But97, S.184]

*Hard Real-Time Computing Systems, page 208*

# Solutions to Priority Inversion

*Disallow preemption* during the execution of all critical sections. Simple approach, but it creates unnecessary blocking as unrelated tasks may be blocked.




*Hard Real-Time Computing Systems, page 211*

# Resource Access Protocols

***Basic idea:*** Modify the priority of those tasks that cause blocking. When a task $J_i$ blocks one or more higher priority tasks, **it temporarily assumes a higher priority.**

- If a higher priority task is blocked by a lower priority task that owns the mutex, the lower priority task can inherit the higher priority to prevent priority inversion. This means that the system temporarily **boosts the priority of the task holding the mutex** to match that of the highest-priority waiting task

*Specific Methods:*

- Priority Inheritance Protocol (PIP), for static priorities
- Priority Ceiling Protocol (PCP), for static priorities
- Stack Resource Policy (SRP),

  for static and dynamic priorities
- others …

# Priority Inheritance Protocol (PIP)

*Assumptions:*

*n* tasks which cooperate through *m* shared resources; fixed priorities, all critical sections on a resource begin with a `wait(`$S_i$`)` and end with a `signal(`$S_i$`)` operation.

*Basic idea:*

When a task $J_i$ blocks one or more higher priority tasks, **it temporarily assumes (inherits) the highest priority** of the blocked tasks.

*Terms:*

We distinguish a fixed *nominal priority* $P_i$ and an *active priority* $p_i$ larger or equal to $P_i$. Jobs $J_1$, ...$J_n$ are ordered with respect to nominal priority where $J_1$ has *highest priority*. Jobs do not suspend themselves.

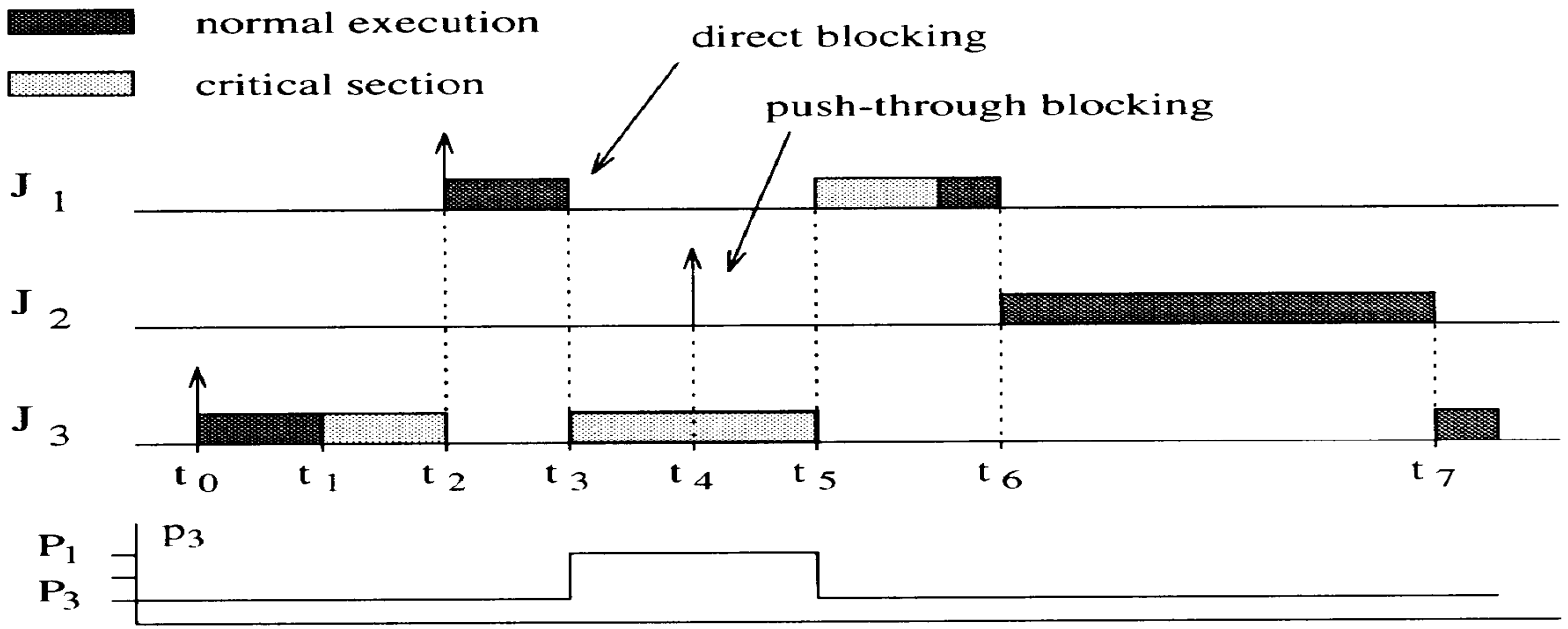
# Priority Inheritance Protocol (PIP)

*Algorithm:*

- Jobs are scheduled based on their *active priorities*. Jobs with the same priority are executed in a *First-Come, First-Served* (FCFS) discipline.
- When a job $J_i$ tries to *enter a critical section* and the resource is blocked by a lower priority job, the job $J_i$ is blocked. Otherwise it enters the critical section.
- When a job $J_i$ is *blocked*, it transmits its active priority to the job $J_k$ that holds the semaphore. $J_k$ resumes and executes the rest of its critical section with a priority $p_k=p_i$ (it *inherits* the priority of the highest priority of the jobs blocked by it).
- When $J_k$ exits a critical section, it *unlocks* the semaphore and the highest priority job blocked on that semaphore is awakened. If no other jobs are blocked by $J_k$, then $p_k$ is set to $P_k$, otherwise it is set to the highest priority of the jobs still blocked by $J_k$.
- Priority inheritance is *transitive*, i.e. if 1 is blocked by 2 and 2 is blocked by 3, then 3 inherits the priority of 1 via 2.

# Priority Inheritance Protocol (PIP)

***Example:***



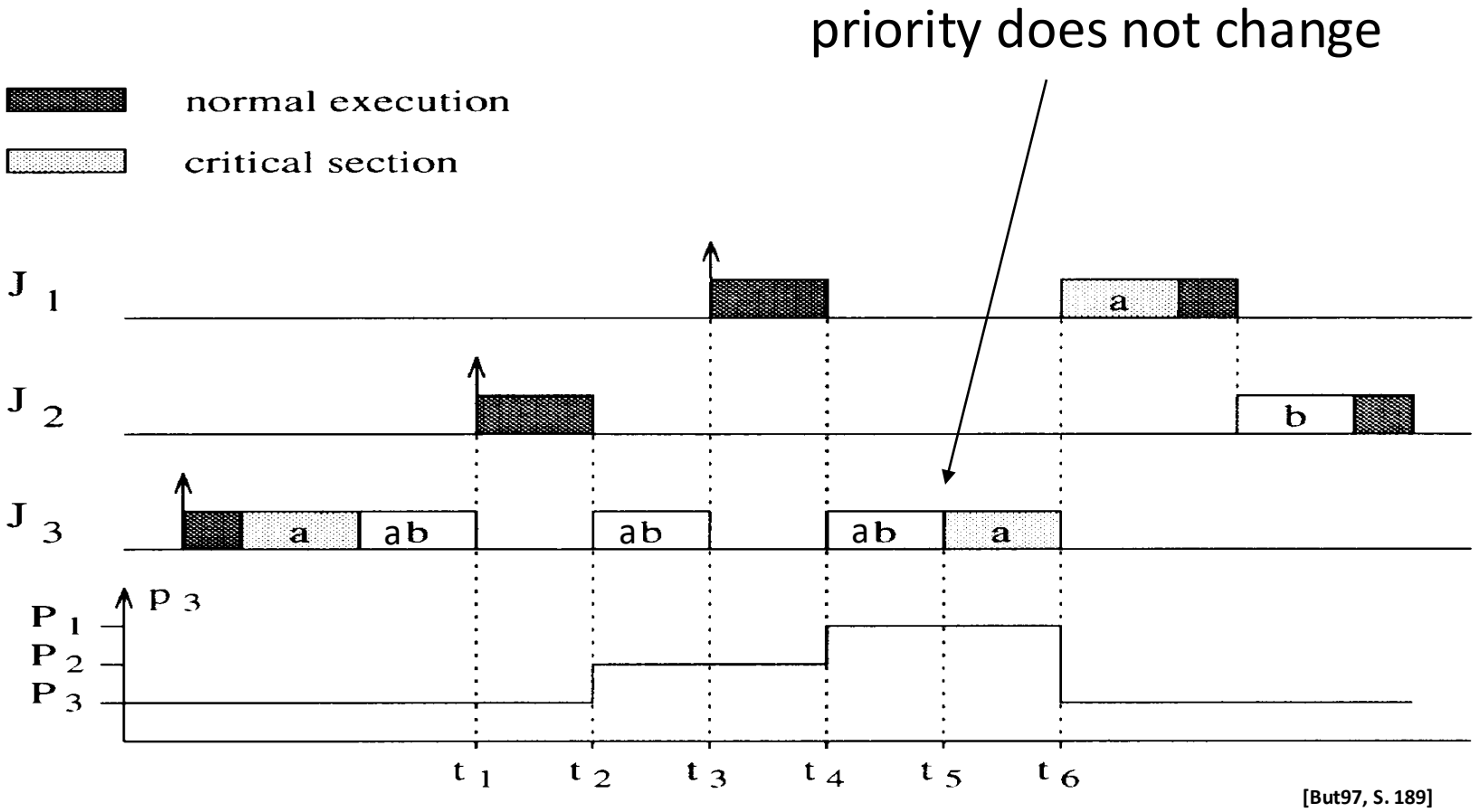

*Direct Blocking*: higher-priority job tries to acquire a resource held by a lower-priority job

*Push-through Blocking*: medium-priority job is blocked by a lower-priority job that has inherited a higher priority from a job it directly blocks

# Priority Inheritance Protocol (PIP)

*Example with nested critical sections:*

- *a task might need 2 resources, only a and-or a & b*

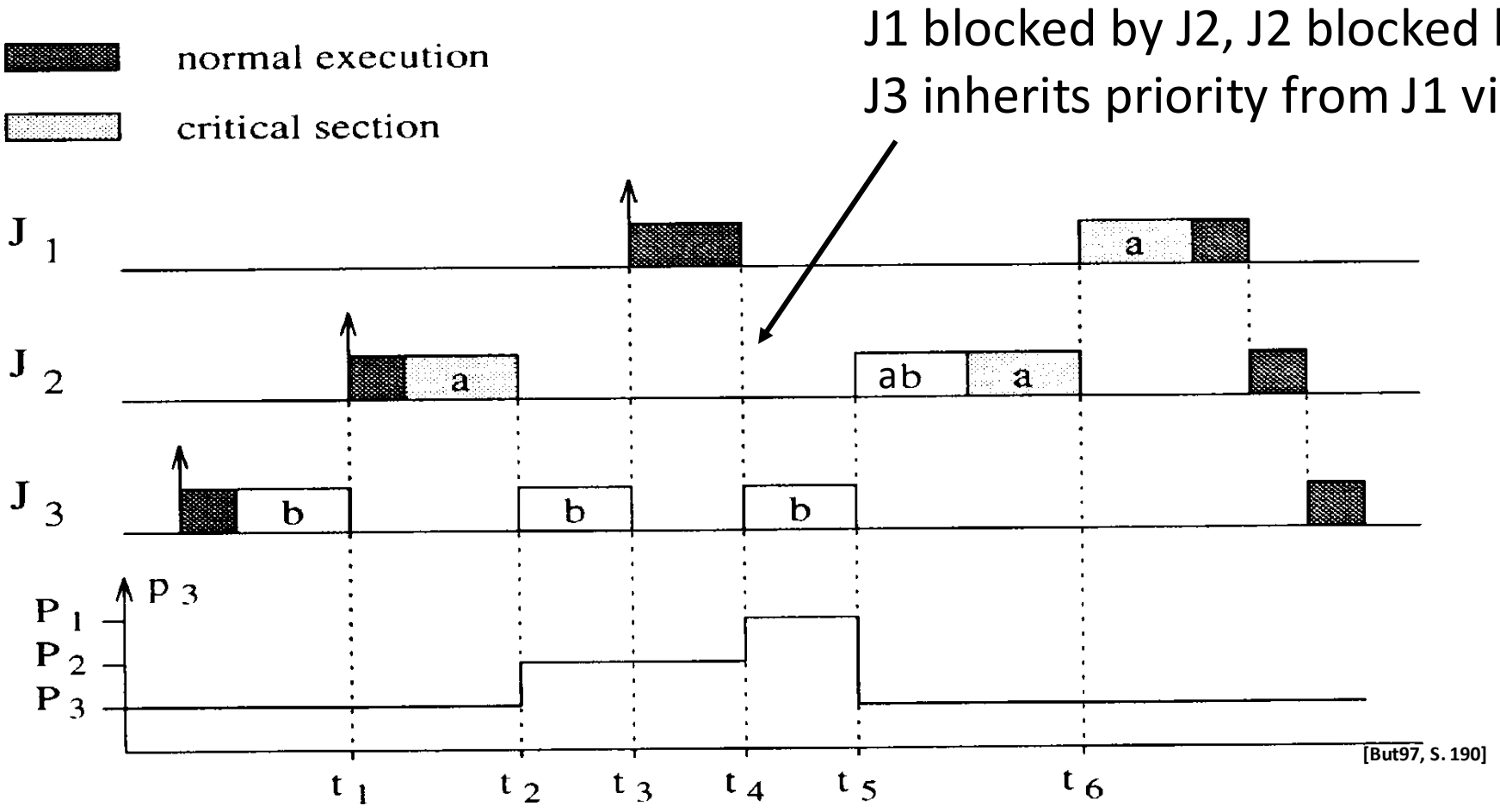



[But97, S. 189]

*Hard Real-Time Computing Systems, page 217*

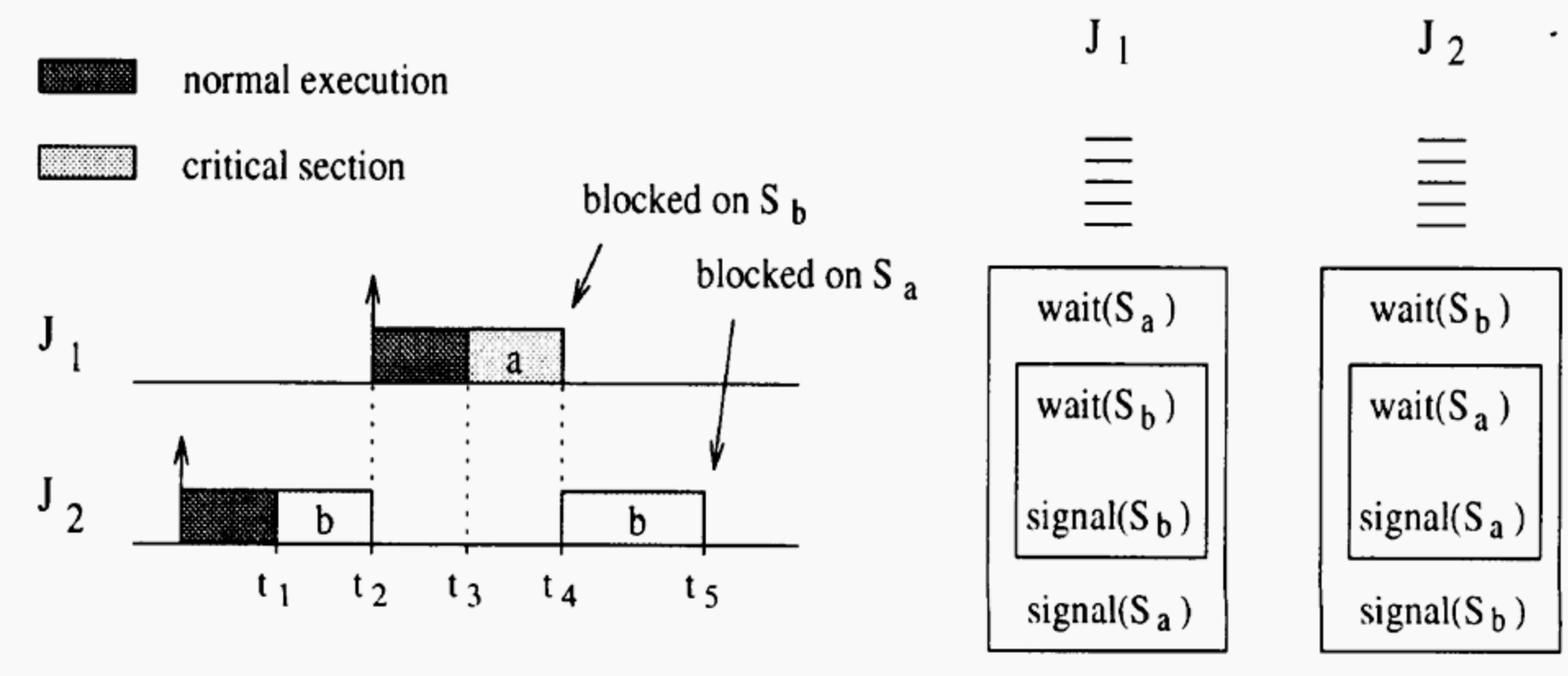
# Priority Inheritance Protocol (PIP)

*Example of transitive priority inheritance:*



# Priority Inheritance Protocol (PIP)

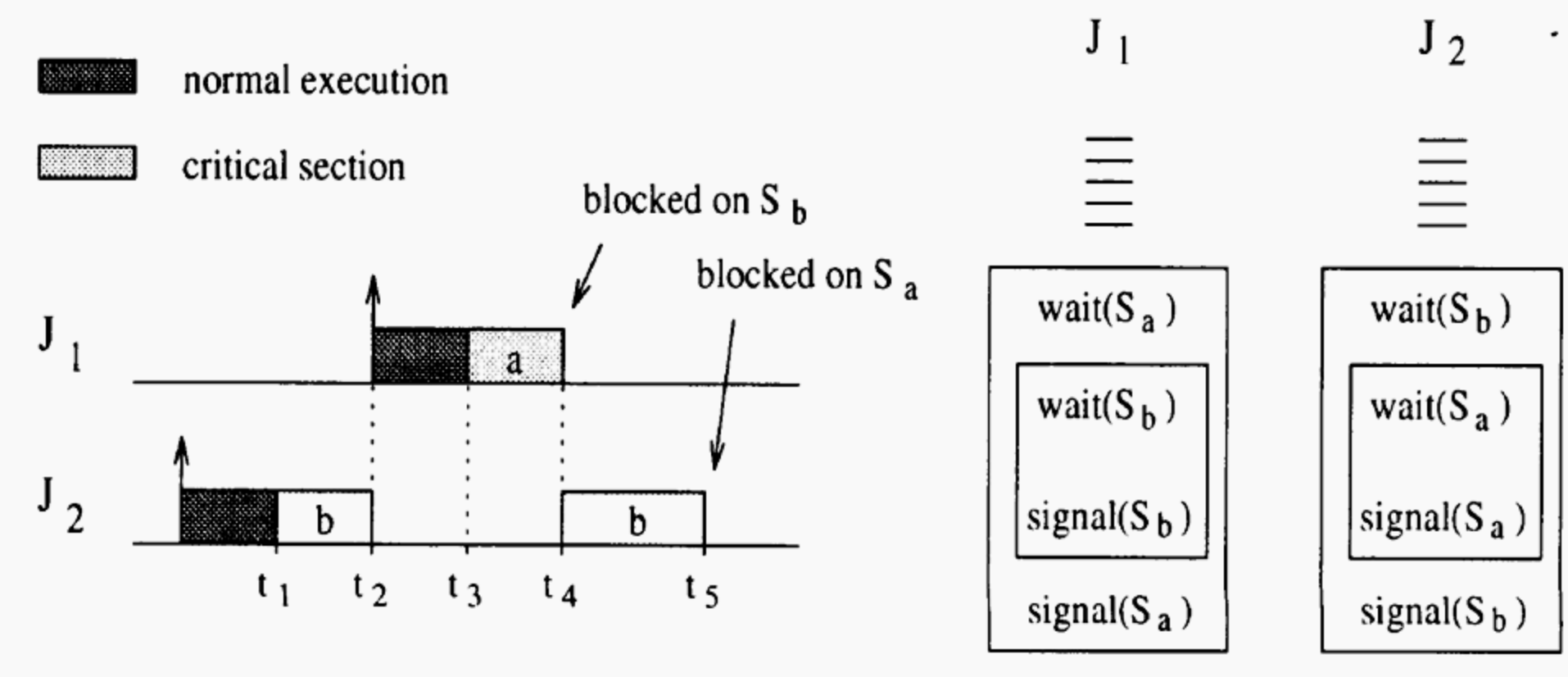
- Which Task will finish first?



# Priority Inheritance Protocol (PIP)

## *Still a Problem: Deadlock*

…. but there are other protocols like the Priority Ceiling Protocol …




[But97, S. 200]

# The MARS Pathfinder Problem (1)

"But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data.

# The MARS Pathfinder Problem (2)


“VxWorks provides preemptive priority scheduling of threads. Tasks on the Pathfinder spacecraft were executed as threads with priorities that were assigned in the usual manner reflecting the relative urgency of these tasks.”

“Pathfinder contained an "information bus", which you can think of as a shared memory area used for passing information between different components of the spacecraft.”

- A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes).”

# The MARS Pathfinder Problem (3)

- The meteorological data gathering task ran as an infrequent, low priority thread. When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex.
- The spacecraft also contained a communications task that ran with medium priority.

| | |
|---|---|
| High priority: | retrieval of data from shared memory |
| Medium priority: | communications task |
| Low priority: | thread collecting meteorological data |

# The MARS Pathfinder Problem (4)

"Most of the time this combination worked fine.

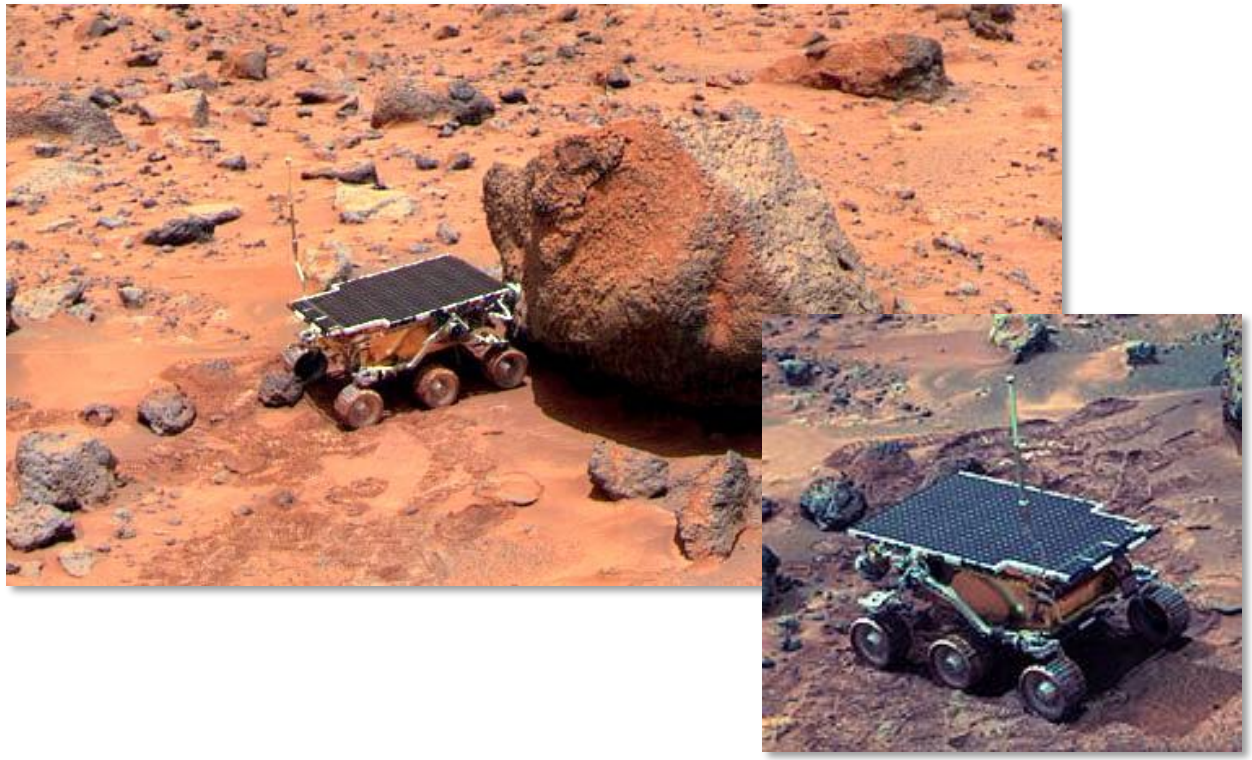
However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running.

After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset. This scenario is a classic case of priority inversion."

# Priority Inversion on Mars

Priority inheritance also solved the Mars Pathfinder problem: the VxWorks operating system used in the pathfinder implements a flag for the calls to mutex primitives. This flag allows priority inheritance to be set to “on”. When the software was shipped, it was set to “off”.

The problem on Mars was corrected by using the debugging facilities of VxWorks to change the flag to “on”, while the Pathfinder was already on the Mars [Jones, 1997].



# Timing Anomalies

# Timing Anomaly

Suppose, a real-time system works correctly with a given processor architecture. Now, **you replace the processor with a faster one.**

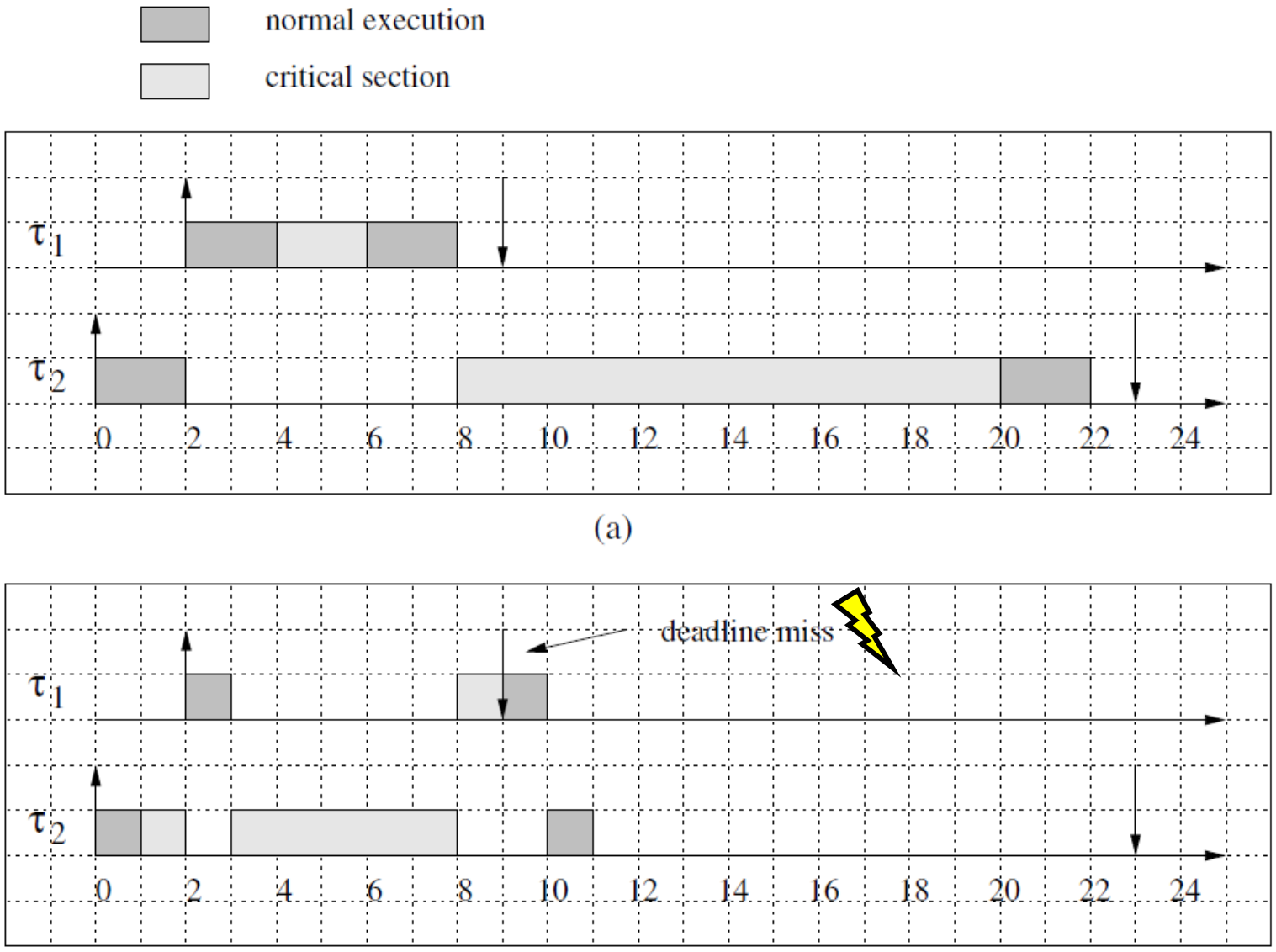
Are real-time constraints still satisfied?

Unfortunately, this is not true in general. *Monotonicity* does not hold in general, i.e., making a part of the system operate faster does not lead to a faster system execution. In other words, *many software and systems architectures are fragile.*

There are usually many timing anomalies in a system, starting from the microarchitecture (caches, pipelines, speculation) via single processor scheduling to multiprocessor scheduling.

# Single Processor with Critical Sections

*Example:* Replacing the processor with one that is twice as fast leads to a deadline miss.

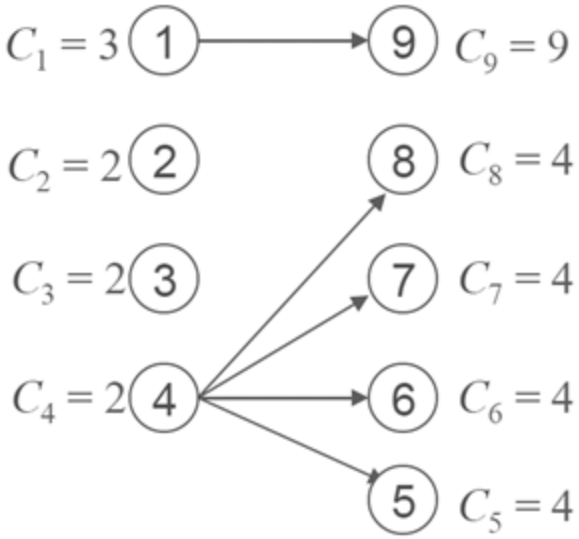


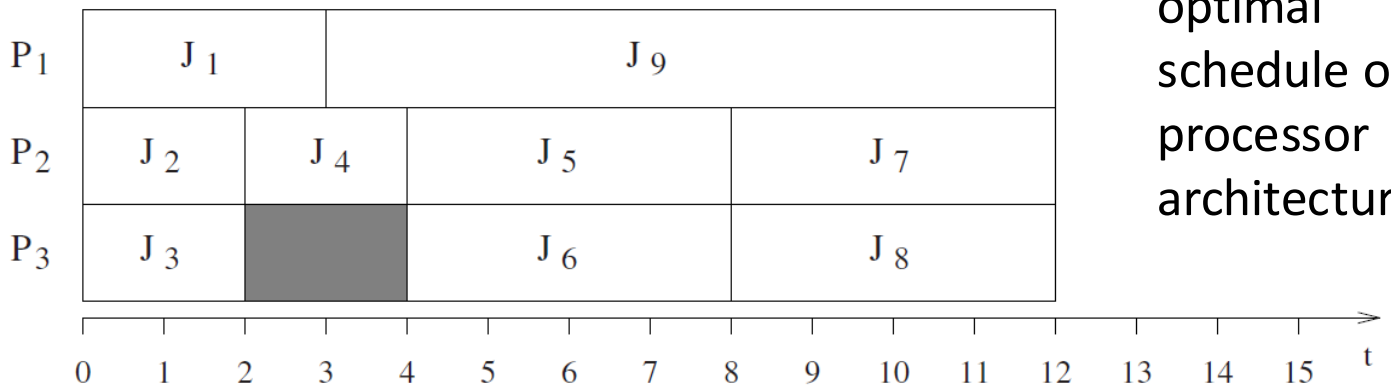

*Hard Real-Time Computing Systems, page 48*

# What About Multiprocessors?

# Multiprocessor Example (Richard's Anomalies)

*Example:* 9 tasks (Ji) with precedence constraints ( i.e. $J_9$ need to be executed after $J_1$)and the shown execution times. Scheduling is preemptive fixed priority, where lower numbered tasks have higher priority than higher numbers. Assignment of tasks to processors is greedy.



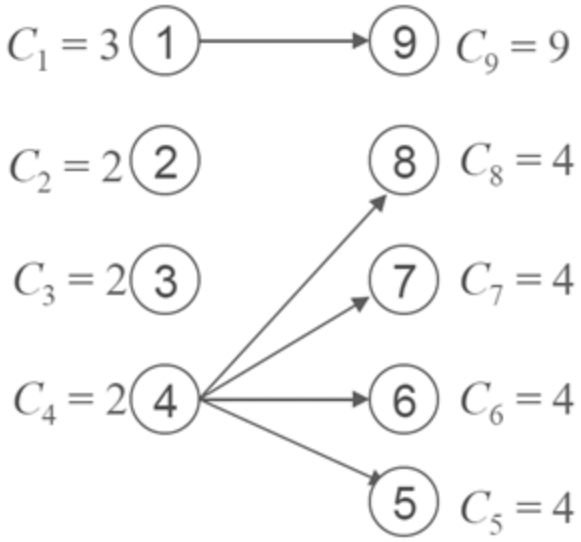



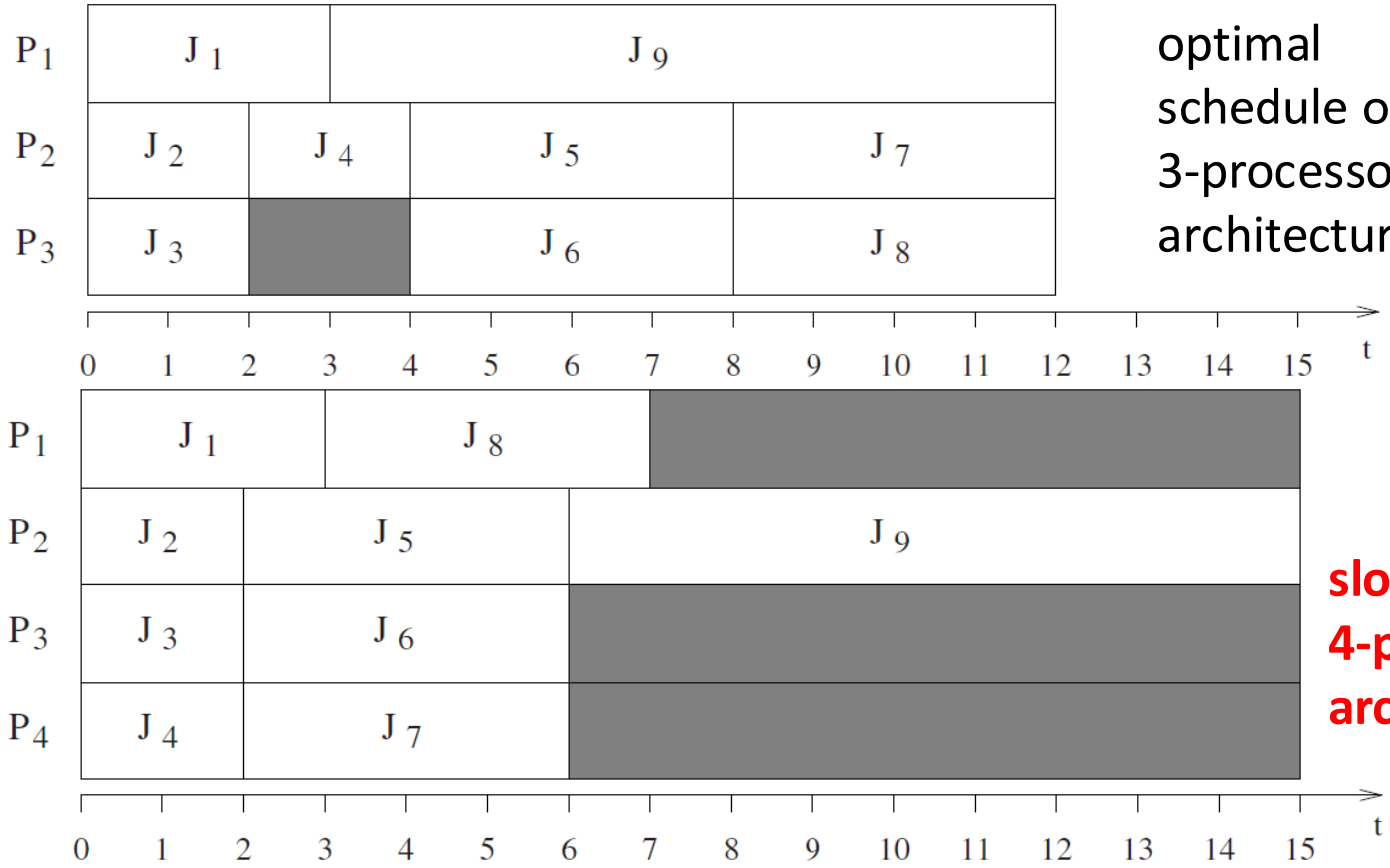

optimal schedule on a 3-processor architecture

Task $J_i$ with $C_i$ = WCET

# Multiprocessor Example (Richard's Anomalies)

*Example:* 9 tasks with precedence constraints and the shown execution times. Scheduling is preemptive fixed priority, where lower numbered tasks have higher priority than higher numbers. Assignment of **tasks to processors is greedy.**



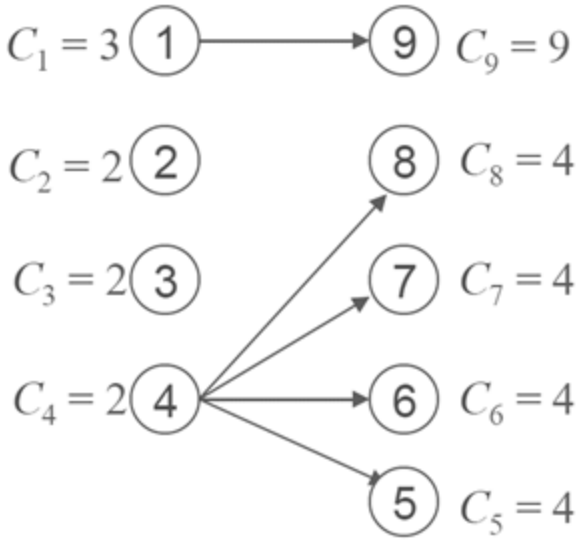



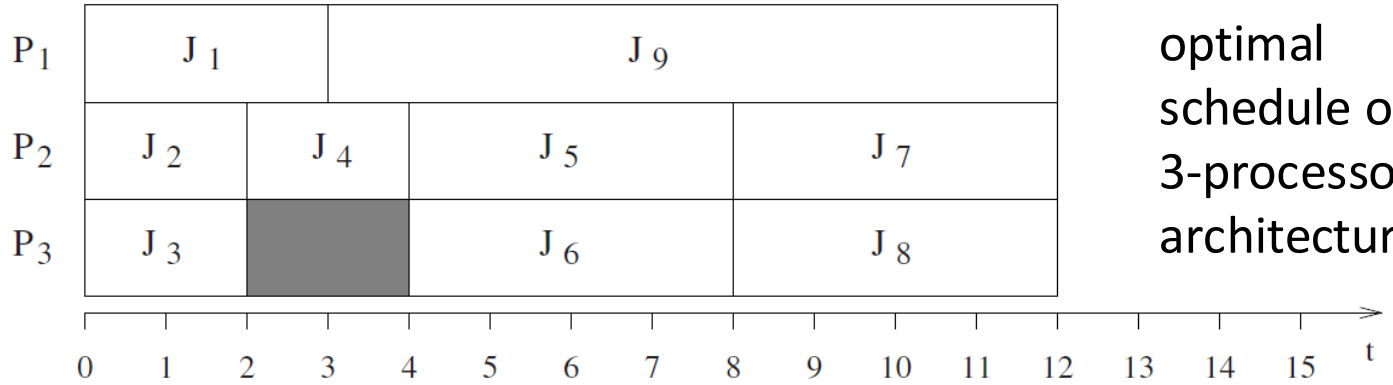

optimal schedule on a 3-processor architecture

**slower on a 4-processor architecture!**

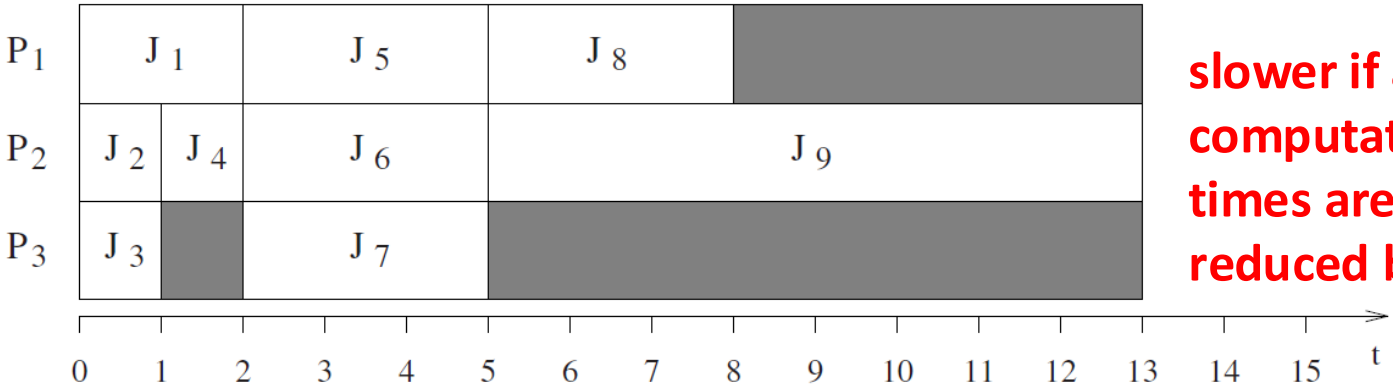
# Multiprocessor Example (Richard's Anomalies)

*Example:* 9 tasks with precedence constraints and the shown execution times. Scheduling is preemptive fixed priority, where lower numbered tasks have higher priority than higher numbers. Assignment of tasks to processors is greedy.



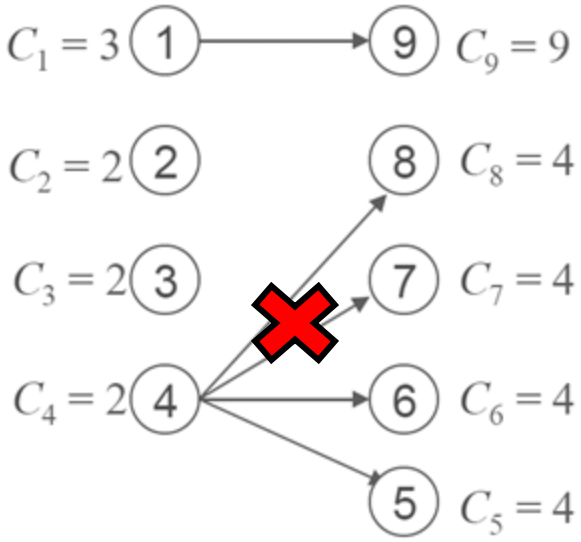

optimal schedule on a 3-processor architecture

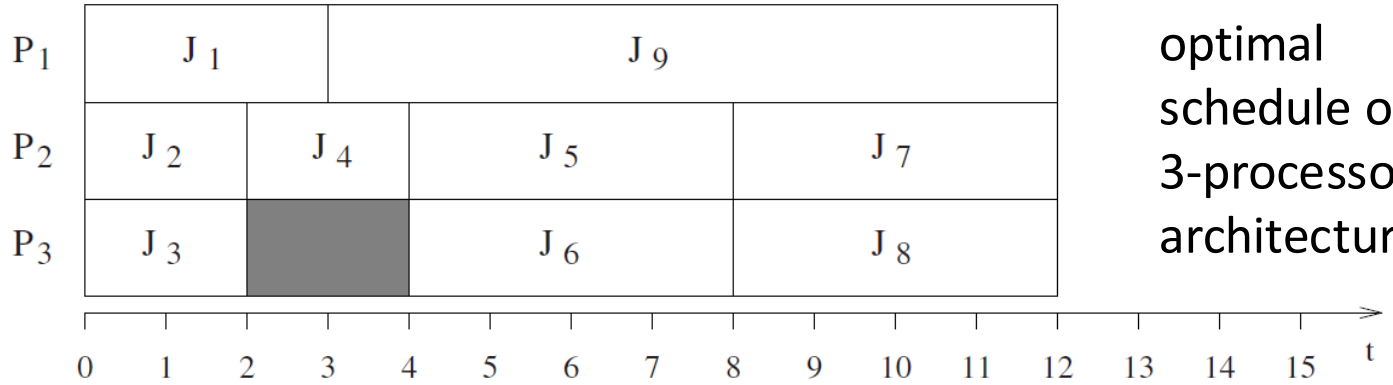



**slower if all computation times are reduced by 1!**

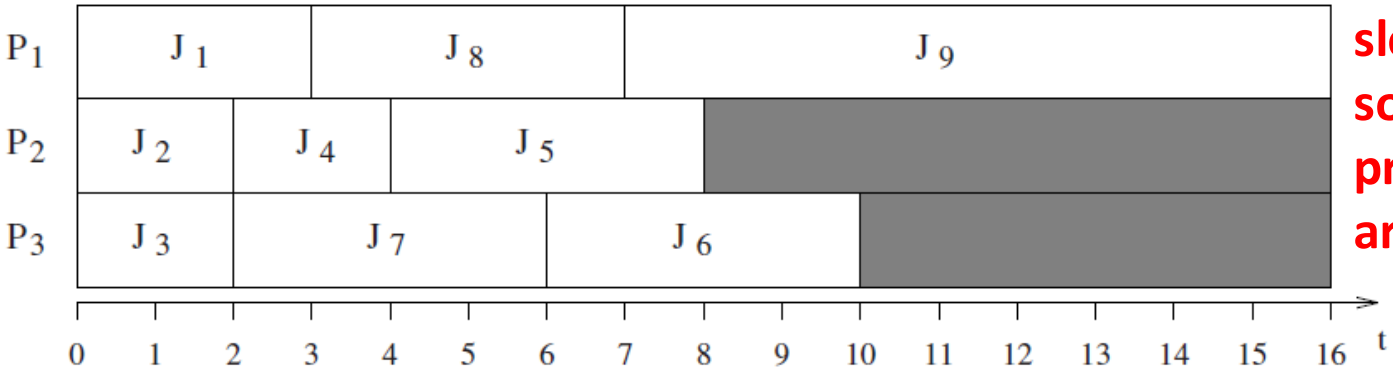
# Multiprocessor Example (Richard's Anomalies)

*Example:* 9 tasks with precedence constraints and the shown execution times. Scheduling is preemptive fixed priority, where lower numbered tasks have higher priority than higher numbers. Assignment of tasks to processors is greedy.




optimal schedule on a 3-processor architecture



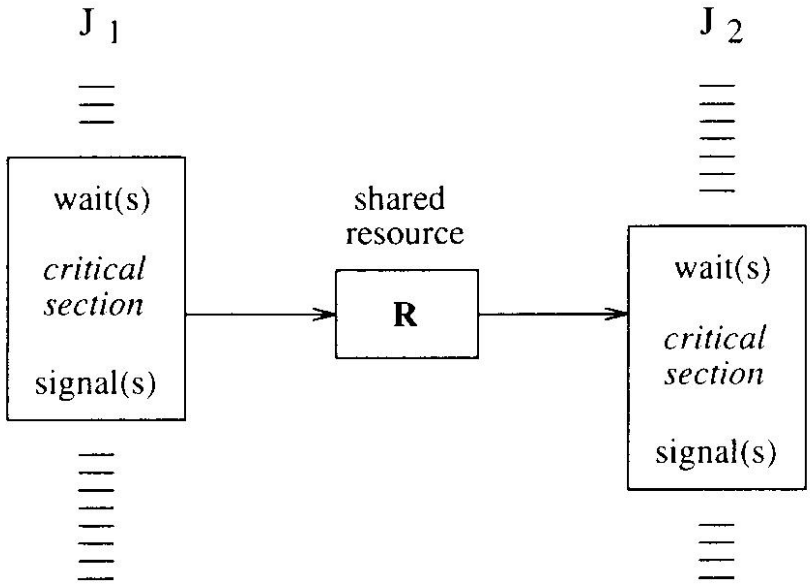

**slower if some precedences are removed!**

# Communication and Synchronization

# Communication Between Tasks

*Problem:* the use of shared memory for implementing communication between tasks may cause priority inversion and blocking.

Therefore, either the implementation of the shared medium is “thread safe” or the data exchange must be *protected by critical sections*.




**thread-safe**" means that multiple threads (or tasks) can access and interact with the shared medium (like memory, data buffers, or communication channels) **without causing data corruption, conflicts, or unpredictable behavior**

*Hard Real-Time Computing Systems, page 33*

# Communication Mechanisms

*Synchronous communication:*

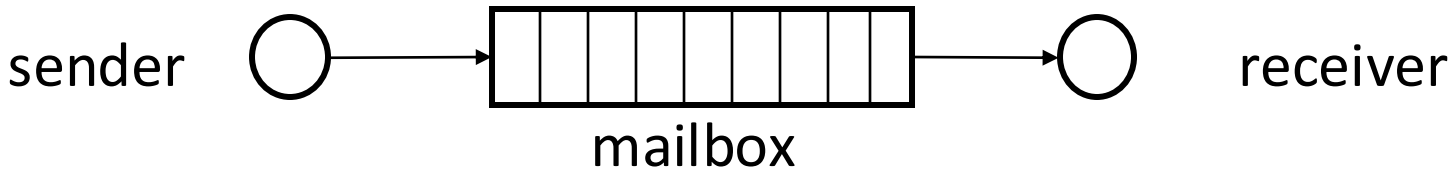
- Whenever two tasks want to communicate they must be synchronized for a message transfer to take place (rendez-vous).
- They have to wait for each other, i.e. both must be at the same time ready to do the data exchange.

- *Problem:*
  - In case of dynamic real-time systems, estimating the maximum blocking time for a process rendez-vous is difficult.
  - Communication always needs synchronization. Therefore, the timing of the communication partners is closely linked.

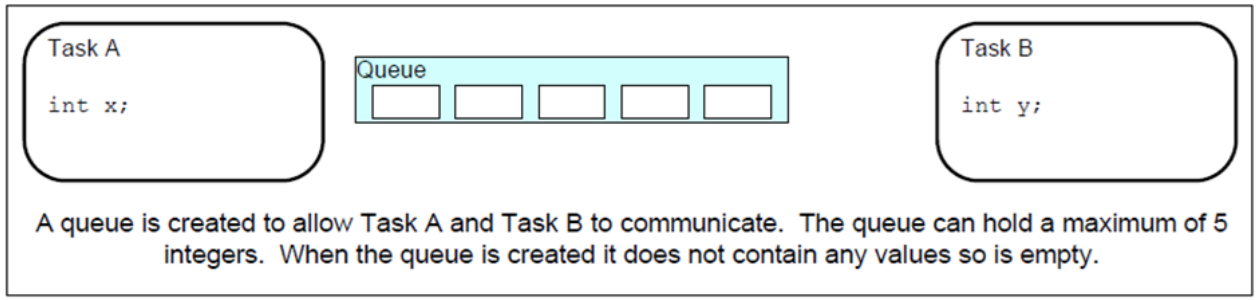
# Communication Mechanisms

*Asynchronous communication:*

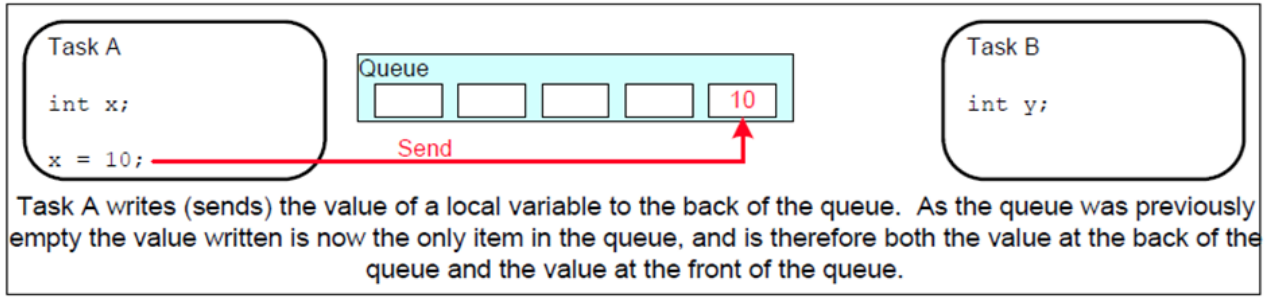
- Tasks do not necessarily have to wait for each other.
- The sender just deposits its message into a channel and continues its execution; similarly the receiver can directly access the message if at least a message has been deposited into the channel.
- More suited for real-time systems than synchronous communication.
- *Mailbox:* **Shared memory buffer**, **FIFO-queue**, basic operations are send and receive, usually has a fixed capacity.
- *Problem:* Blocking behavior if the channel is full or empty; alternative approach is provided by cyclical asynchronous buffers or double buffering.



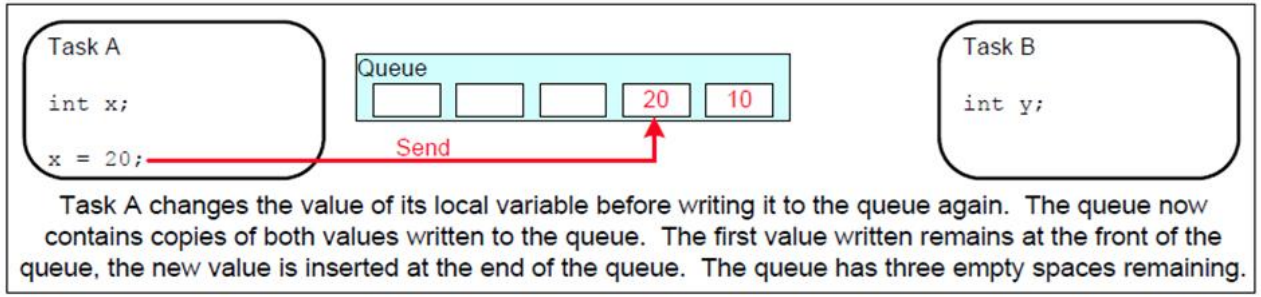
# FreeRTOS - Queues



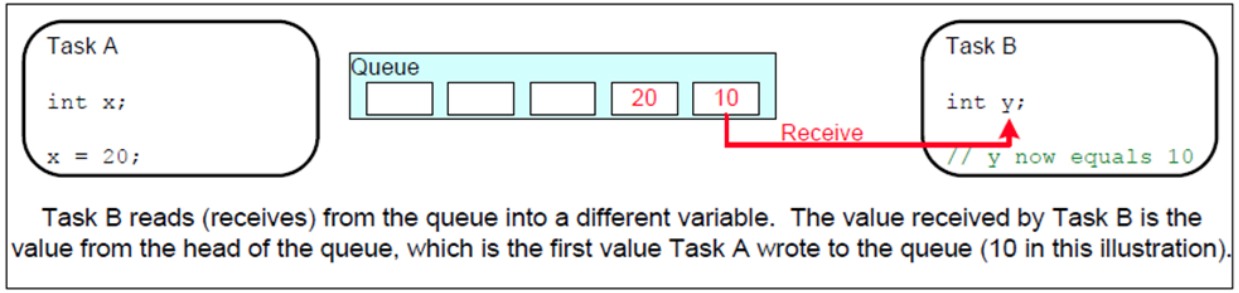

A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.



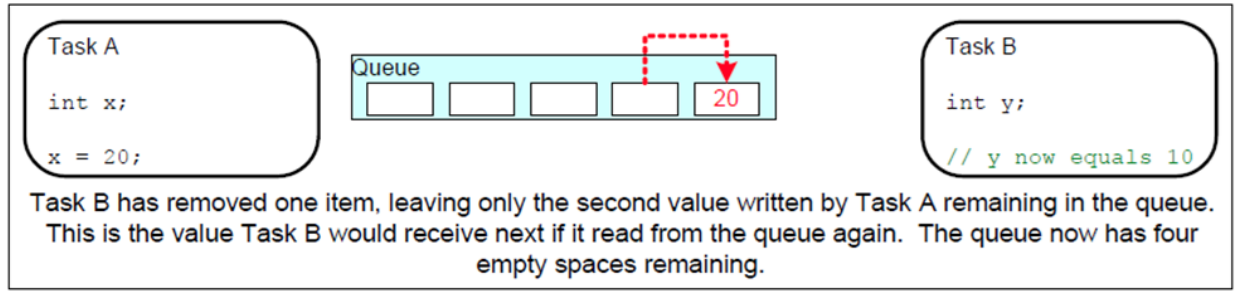

Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.




Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.




Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).




Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

# FreeRTOS – Create a Queue

*Allocating (static) memory for the queue:*

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength, UBaseType_t uxItemSize );
```

returns handle to create queue

The maximum number of items that the queue being created can hold at any time

the size in bytes of each data item

# FreeRTOS – Send to Queue

*Sending item to a queue:*

```
BaseType_t xQueueSend ( QueueHandle_t xQueue,
                        const void * pvItemToQueue,
                        TickType_t xTicksToWait );
```

Returns pdPass if item was successfully added to queue

The maximum amount of time the task should remain in the Blocked state to wait for space to become available on the queue

A pointer to the data to be copied into the queue

# FreeRTOS – Receive from Queue

*Receiving item from a queue:*

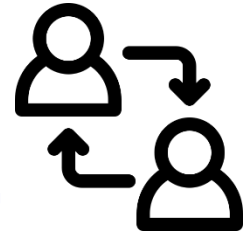
```
BaseType_t xQueueReceive ( QueueHandle_t xQueue,
                           void * const pvBuffer,
                           TickType_t xTicksToWait );
```

Returns pdPass if data was successfully read from the queue

The maximum amount of time the task should remain in the Blocked state to wait for data to become available on the queue

A pointer to the memory into which the received data will be copied

# FreeRTOS – Queues Priority



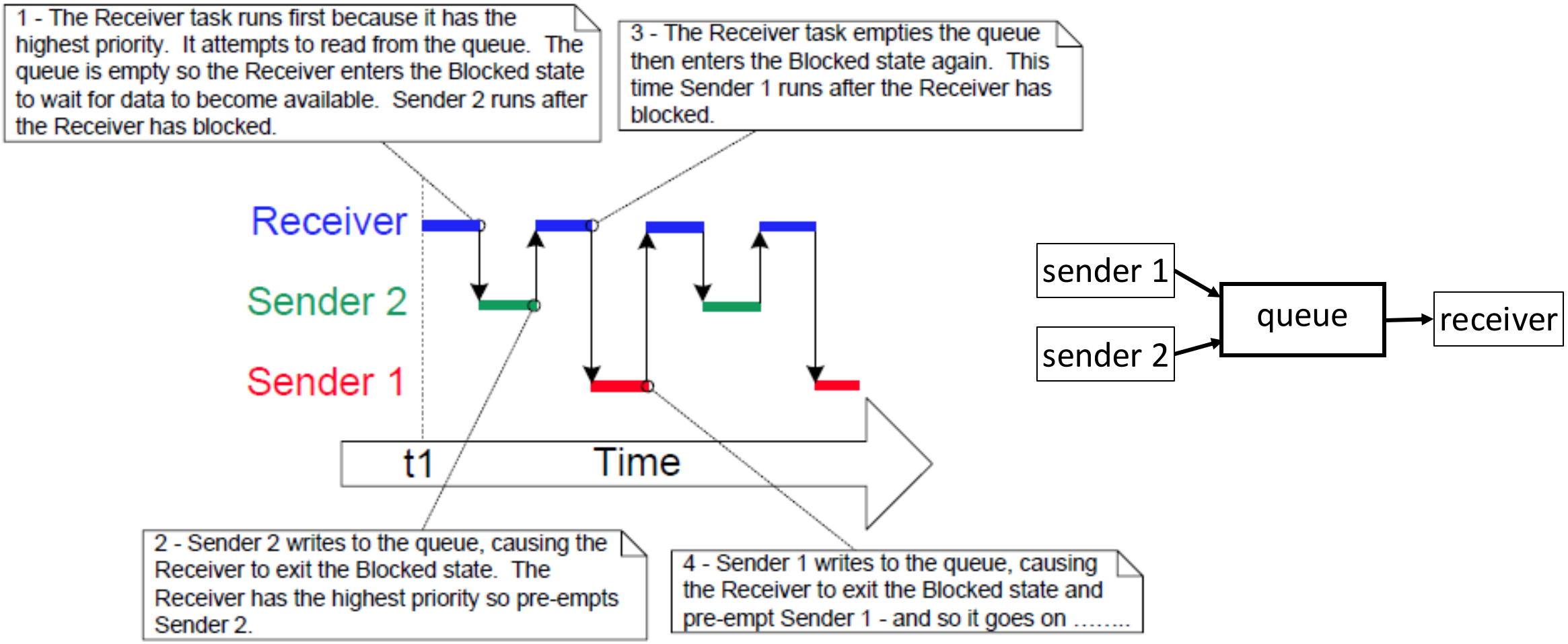
***Example:***

- Two sending tasks with equal priority 1 and one receiving task with priority 2.

- FreeRTOS schedules tasks with equal priority in **a round-robin manner: A blocked or preempted task is put to the end of the ready queue for its priority.** The same holds for the currently running task at the expiration of the time slice.

# FreeRTOS – Queues Priority

*Example cont.:*



# What Did You Learn?

- ✓ Embedded Operating System (FreeRTOS)
- ✓ Task Scheduling
  - ✓ Priorities
  - ✓ Preemption
  - ✓ Time Slicing
- ✓ Resource Sharing
  - ✓ Semaphore
  - ✓ Mutexes
- ✓ Priority Inversion
  - ✓ Priority Inheritance Protocol (PIP)
- ✓ Timing Anomalies
  - ✓ Richard's Anomalies
- ✓ Communication and Synchronization (Queues)