TI RTOS - Introduction

RTOS Concepts & TI RTOS Introduction

Agenda

- Bare metal example
- TI RTOS Concepts
- Sharing resources
- Common challenges

Prerequisites

- Basic embedded firmware experience
  - Interrupts
  - Memory
    - Data
    - Stack
    - Heap

Bare metal vs. RTOS

Example – Smart Thermostat

Example – Smart Thermostat

Super loops, interrupts and how we would really want it to work

Example – Smart Thermostat

- Requirements – Version 1
  - User interface
    - LCD
    - Buttons
  - Remote control to change temperature
    - RF Transceiver
  - Sensor inputs
    - Temperature
  - Schedule operation based on time

Bare Metal – Smart Thermostat

- Bare metal system design – No OS
- Super loop:

```
void main() {
  /* Initialize hardware */

  while(1) {
    checkButtons();
    updateLcd();
    checkTemperature();
    checkTimeSchedule();
    checkRadioReceive();
    doRadioTransmit();
  }
}
```

Example – Smart Thermostat

- **Updated** Requirements – Version 2
  - User interface
    Smooth user experience, < X ms from button press to LCD updated
    - LCD
    - Buttons
  - Remote control to change temperature
    Needs to send an ACK within Y ms from receiving a packet
    - RF Transceiver
  - Sensor inputs
    Sensor readings every second
    - Temperature
    - Humidity
  - Schedule operation based on time

Bare Metal – Smart Thermostat

- Bare metal system design – No OS
- **Interrupt driven**

```c
void main() {
  /* Initialize hardware and setup interrupts */
  while(1);
}
```

Interrupt service routines

```c
void buttonIsr() {
  /* Read buttons */
  /* Update LCD */
  /* Do logic */
}
```

```c
void scheduleTimerIsr() {
  /* Check schedule */
  /* Update target temp. */
}
```

```c
void sensorTimerIsr() {
  /* Save current temp. */
}
```

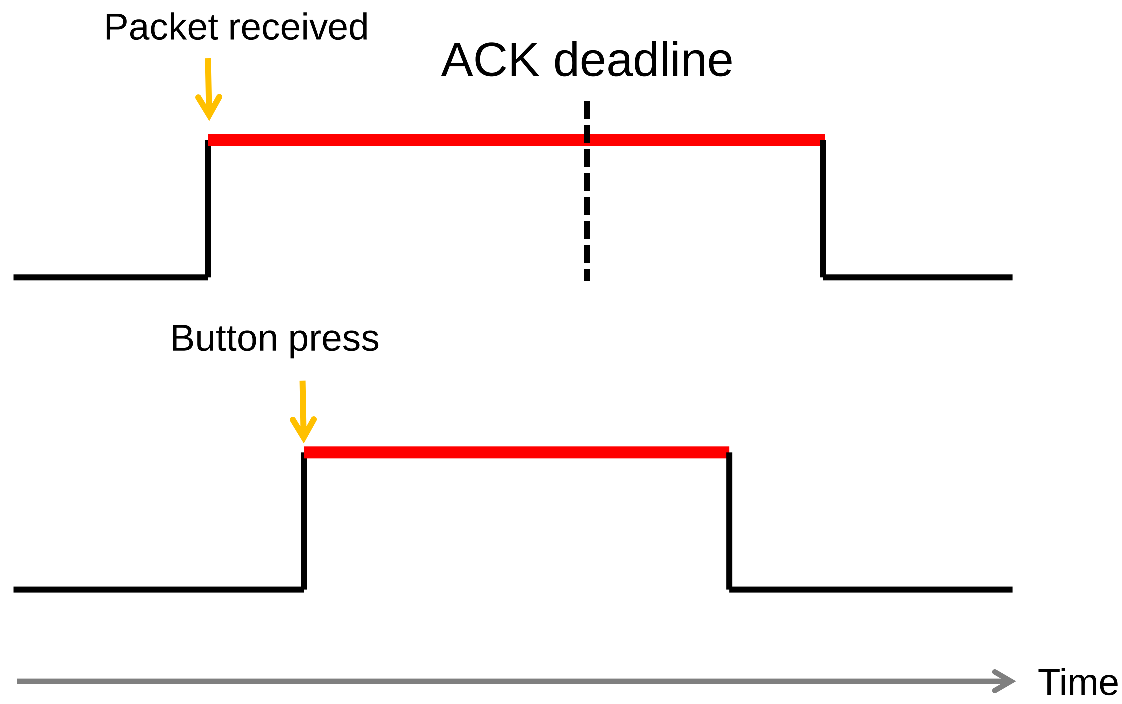
```c
void radioRxIsr() {
  /* Check packet*/
  /* Transmit ACK*/
}
```

Bare Metal – Smart Thermostat

Interrupt service routines – No nesting

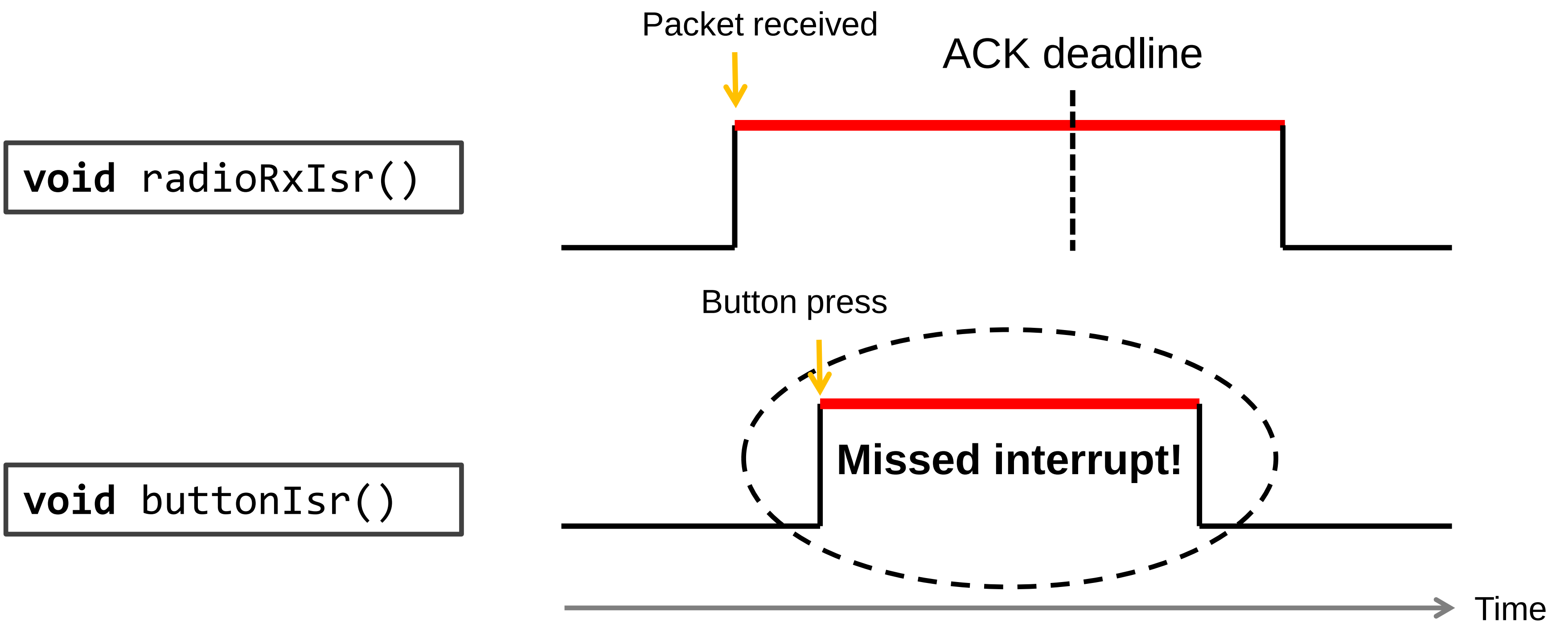
`void radioRxIsr()`

`void buttonIsr()`



Bare Metal – Smart Thermostat

Interrupt service routines – No nesting



Texas Instruments

Bare Metal – Smart Thermostat

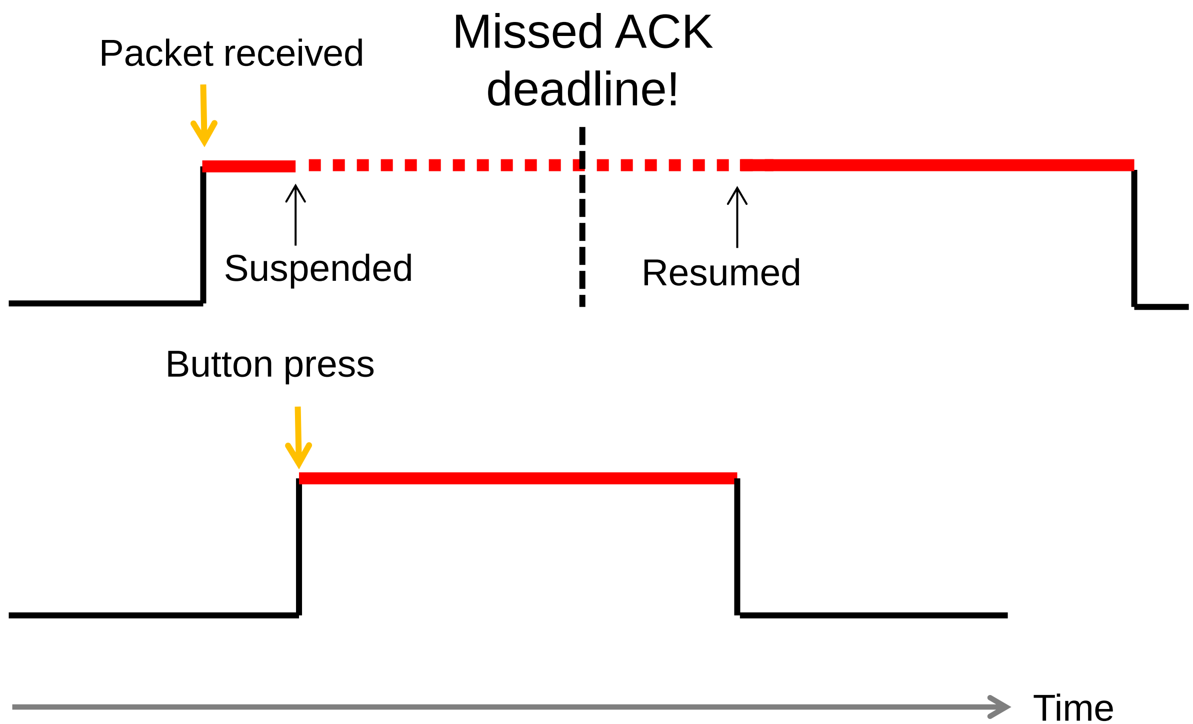
Solution: <u>Nested</u> Interrupt service routines

`void radioRxIsr()`

Lower priority

Higher priority

`void buttonIsr()`



Bare Metal – Smart Thermostat

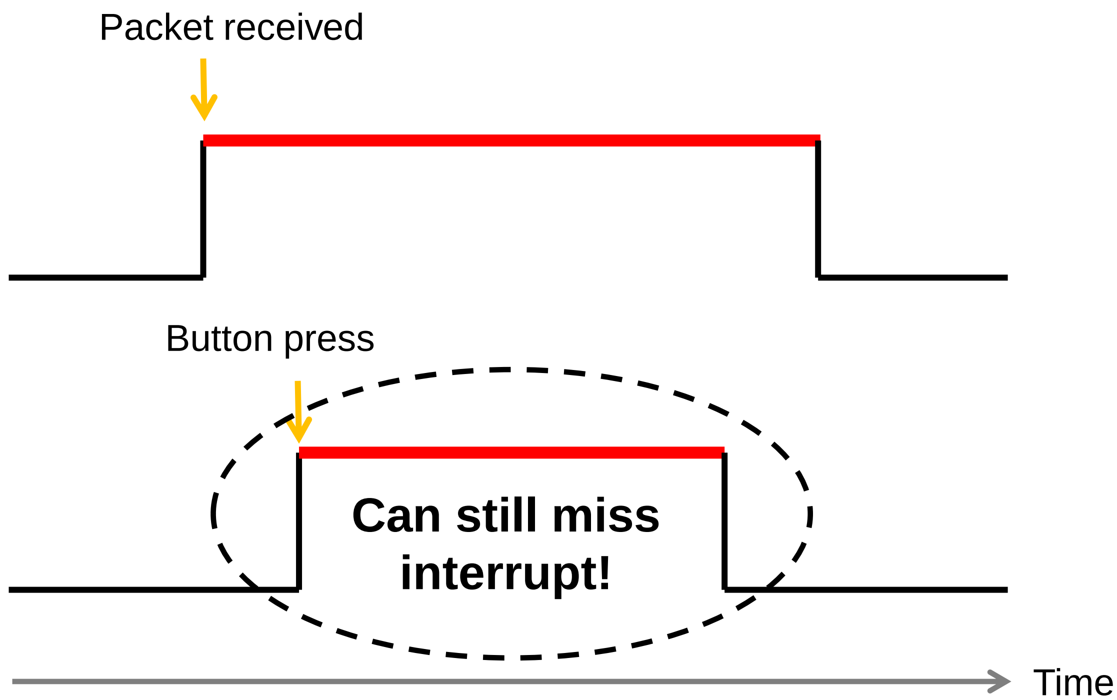
Solution: <u>Nested</u> Interrupt service routines

`void radioRxIsr()`

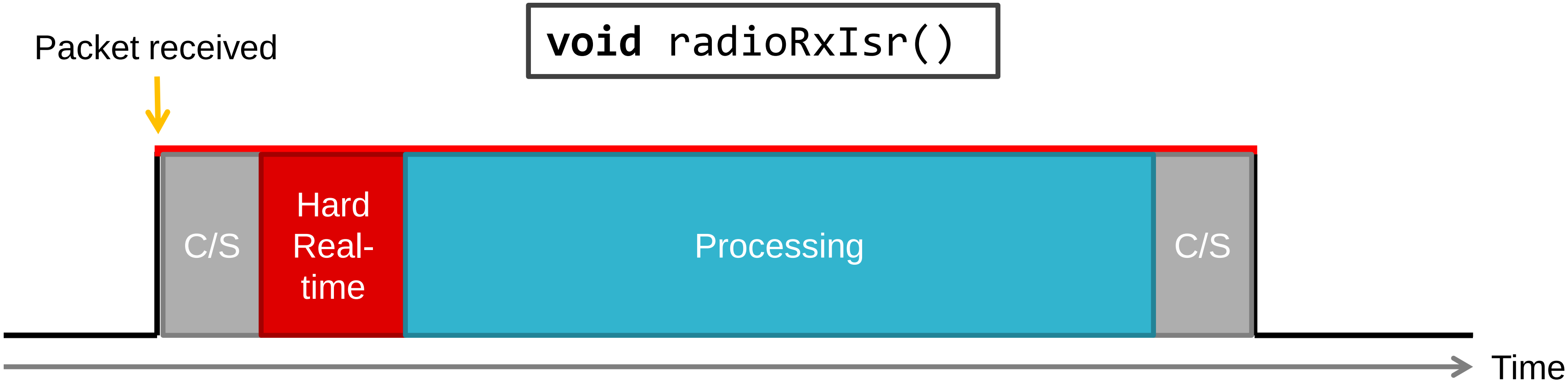
Higher priority

Lower priority

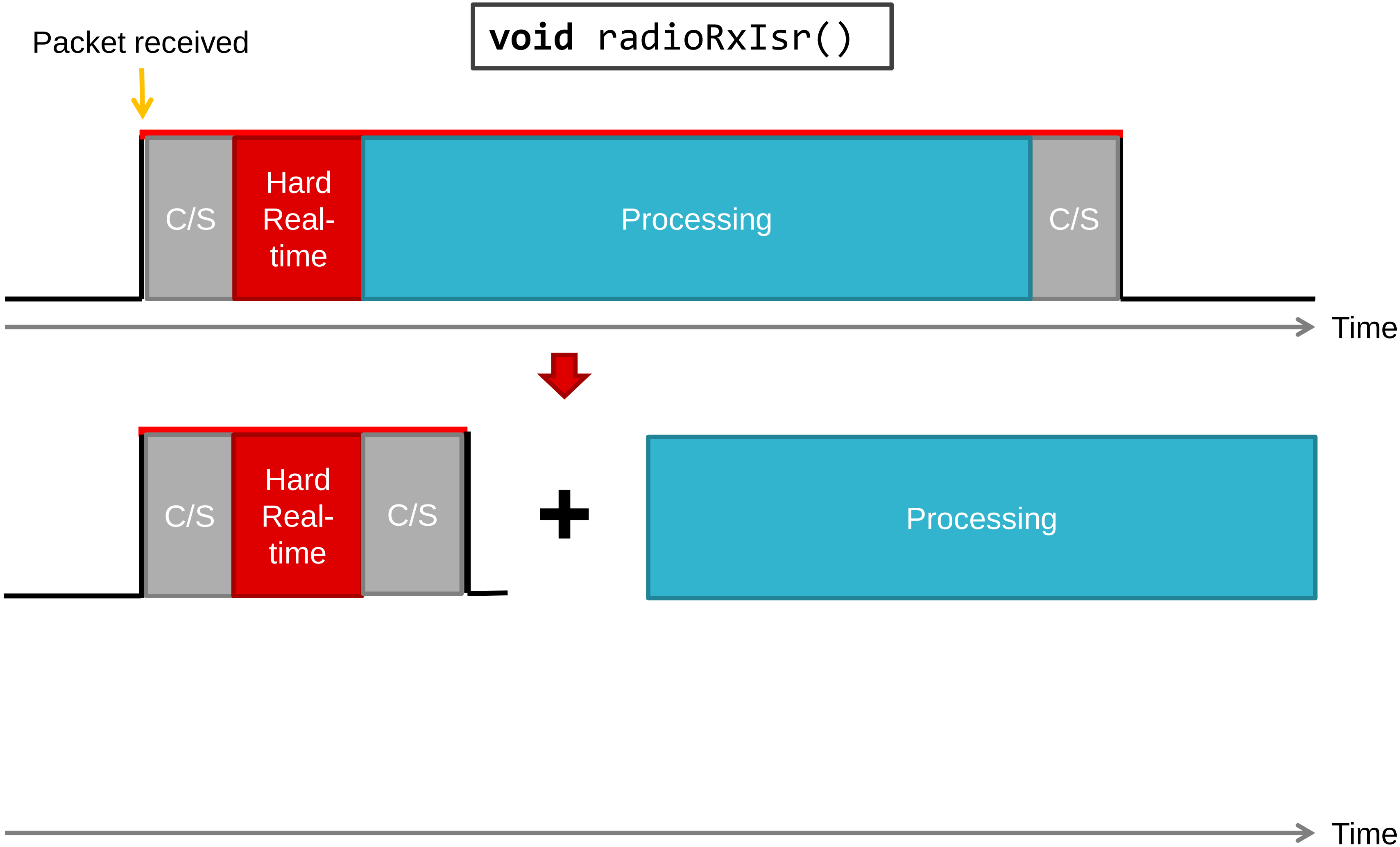
`void buttonIsr()`



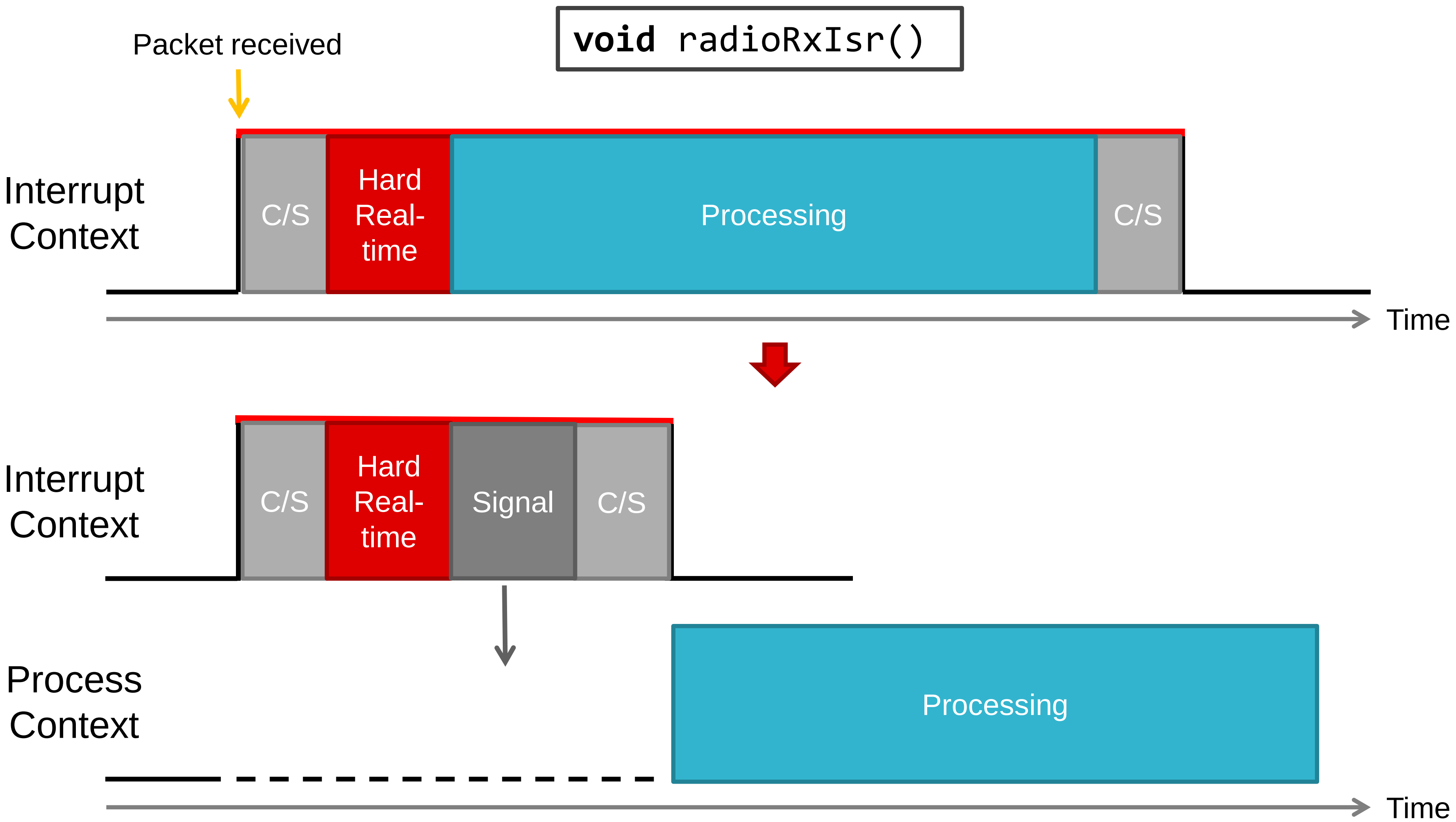
Bare Metal – Smart Thermostat



Bare Metal – Smart Thermostat



Bare Metal – Smart Thermostat



Bare Metal – Smart Thermostat

- **Interrupt driven with processing in main**

```c
void main() {
  /* Initialize hardware and setup interrupts */
  while(1) {
    if(buttonSemaphore)
      processButtonPress();     // Takes time
    if(radioRxSemaphore)
      processPacketReceived(); // Takes time
  }
}
```

Interrupt service routines

```c
void buttonIsr() {
  /* Read buttons */
  buttonSemaphore = 1;
}
```

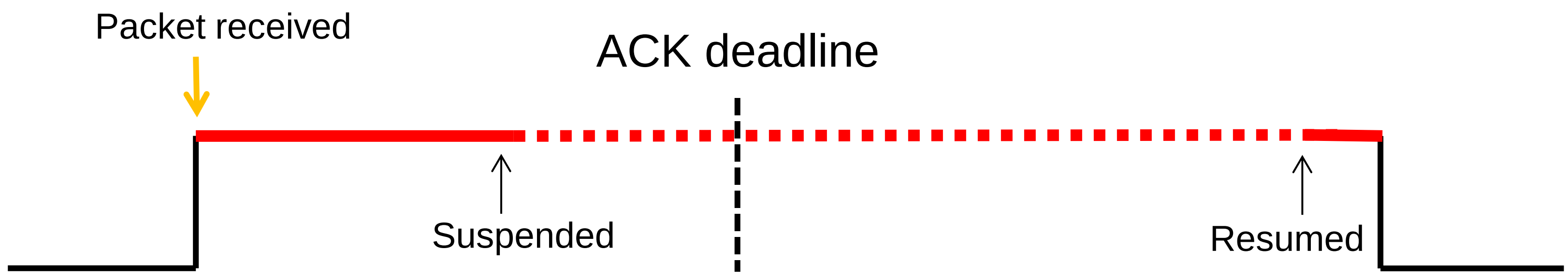
```c
void radioRxIsr() {
  /* Read packet*/
  radioRxSemaphore = 1;
}
```

Bare Metal – Smart Thermostat

Nested Interrupt service routines

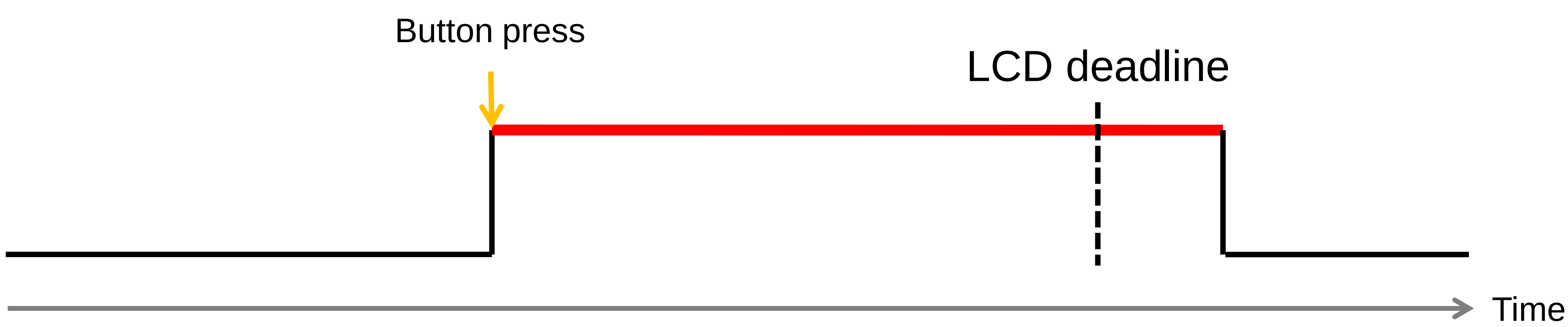
```
void radioRxIsr()
```

Lower priority

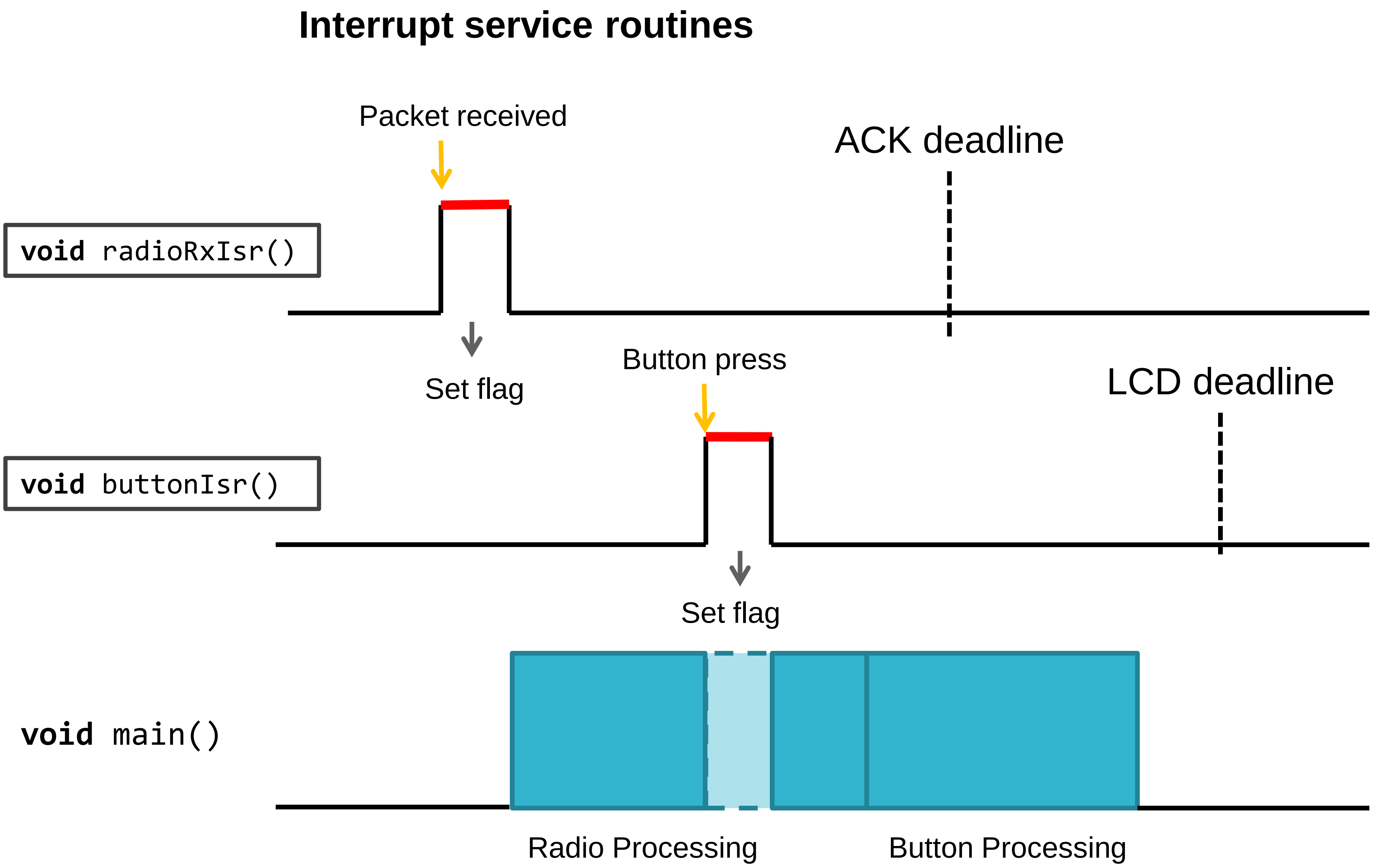



```
void buttonIsr()
```

Higher priority

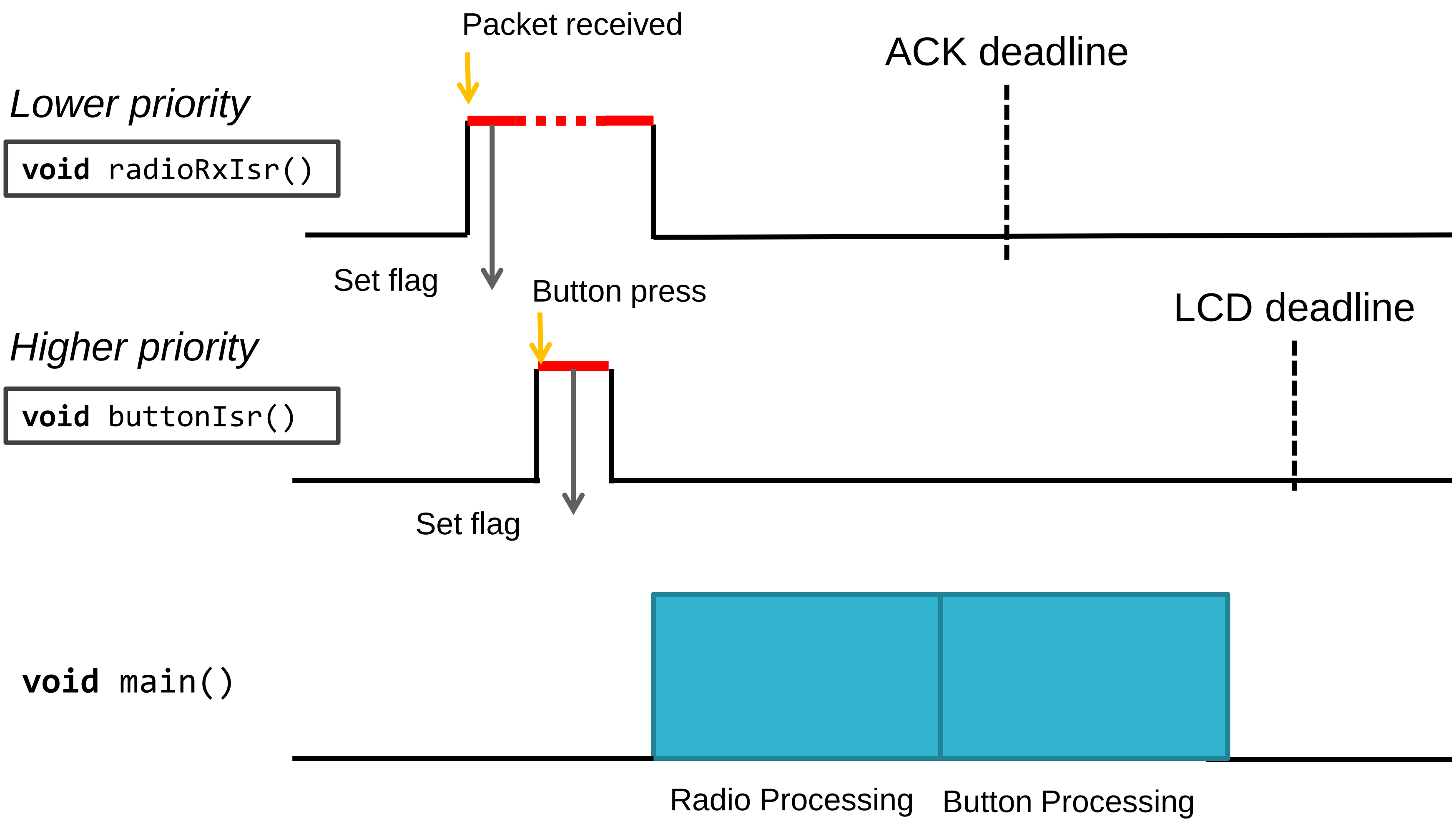


Bare Metal – Smart Thermostat



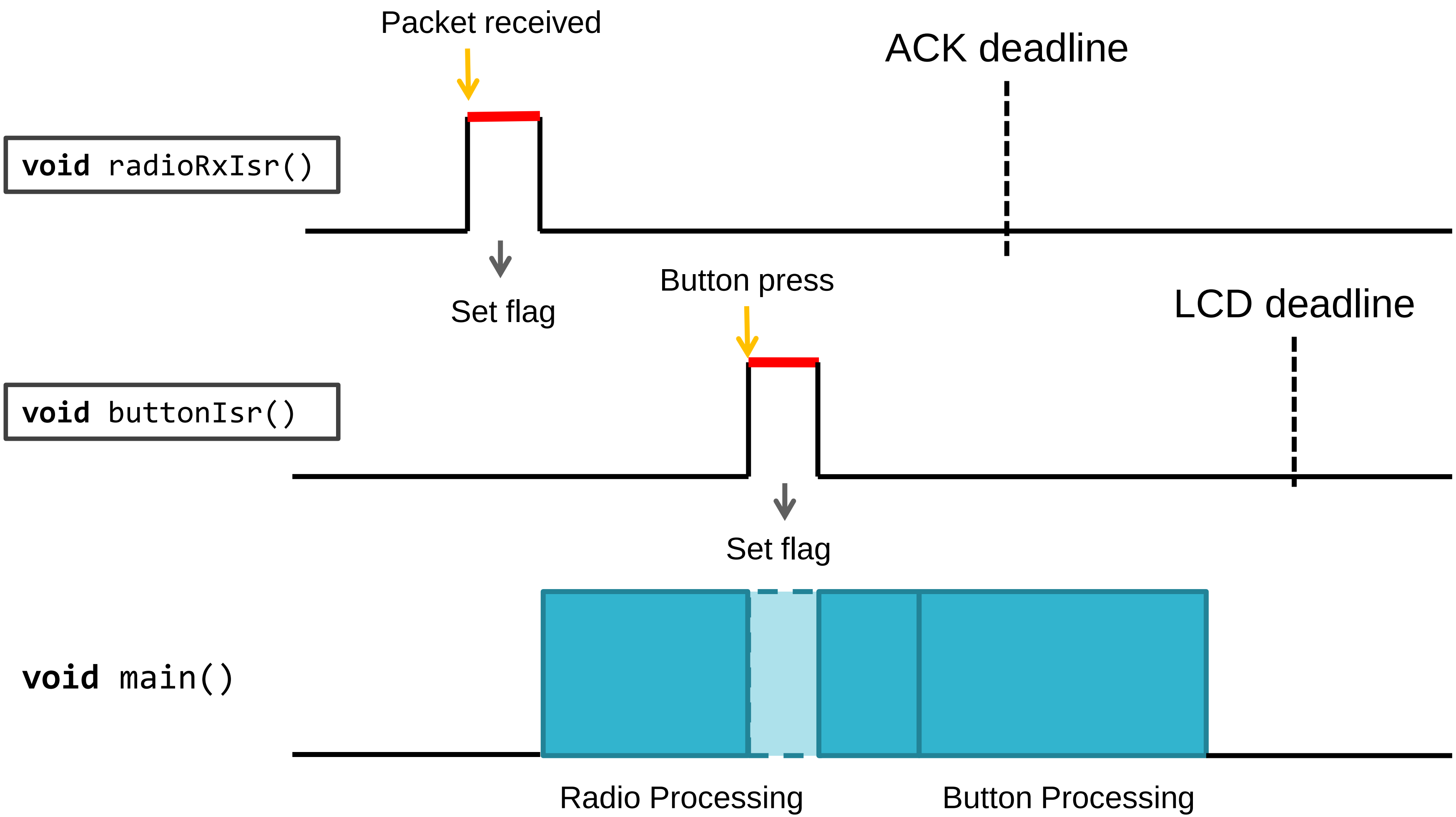
Bare Metal – Smart Thermostat

Nested Interrupt service routines



Bare Metal – Smart Thermostat

What is one of the major problems with this approach?



Example – Smart Thermostat

- **Updated** Requirements – Version 3
  A smooth user experience is most important. If necessary, it's okay to miss the ACK deadline to improve the user experience.

  – User interface
    Smooth user experience, <200ms from button press to LCD updated
    - LCD
    - Buttons

  – Remote control to change temperature
    Needs to send an ACK within 1ms from receiving a packet
    - RF Transceiver

  – Sensor inputs
    Sensor readings every second
    - Temperature
    - Humidity

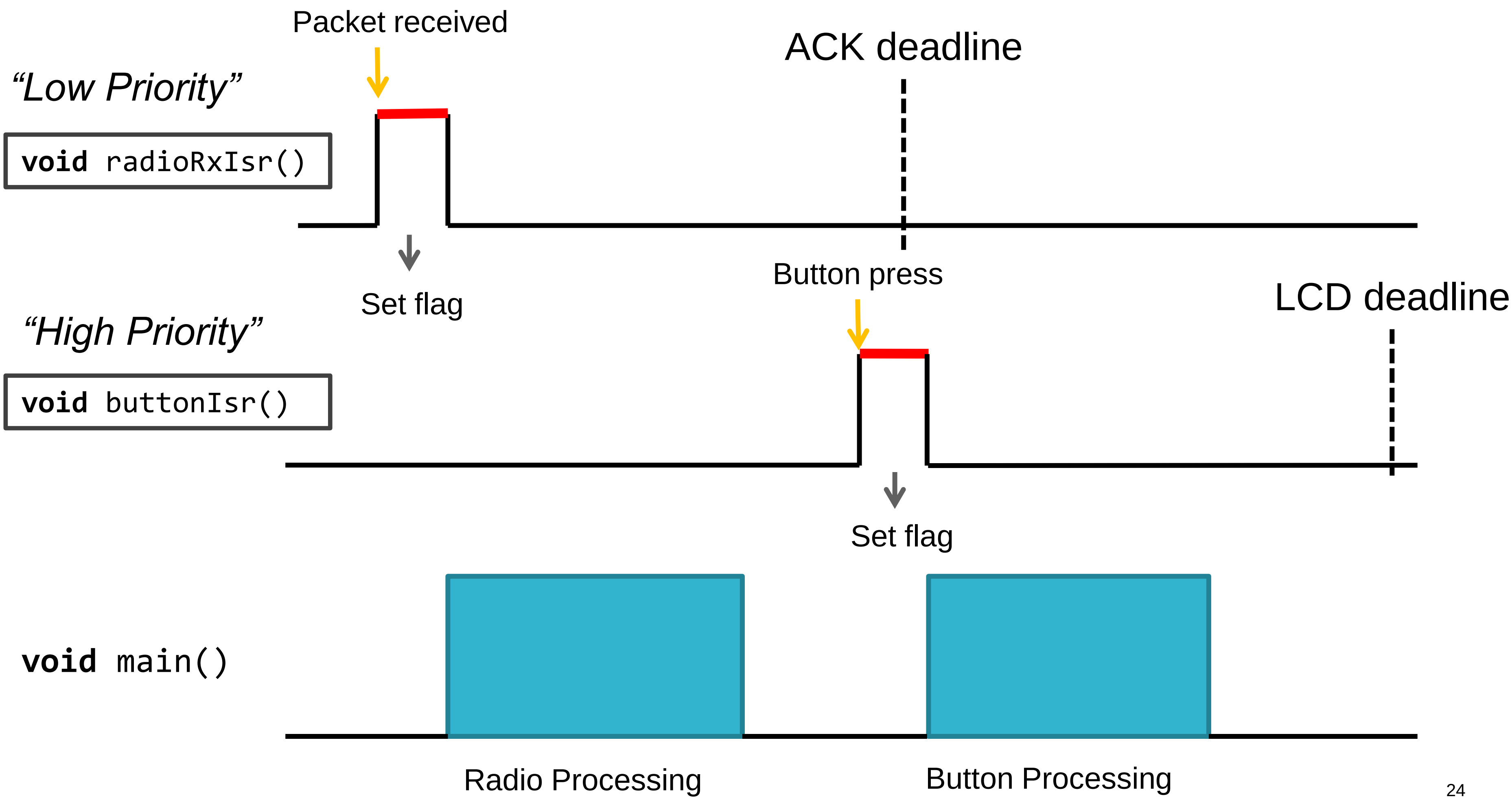
  – Schedule operation based on time

Example – Smart Thermostat

- **Updated** Requirements – Version 3
  A smooth user experience is most important. If necessary, it's okay to miss the ACK deadline to improve the user experience.
  - User interface – **"High Priority"**
    Smooth user experience, <200ms from button press to LCD updated
    - LCD
    - Buttons
  - Remote control to change temperature – **"Low Priority"**
    Needs to send an ACK within 1ms from receiving a packet
    - RF Transceiver
  - Sensor inputs
    Sensor readings every second
    - Temperature
    - Humidity
  - Schedule operation based on time

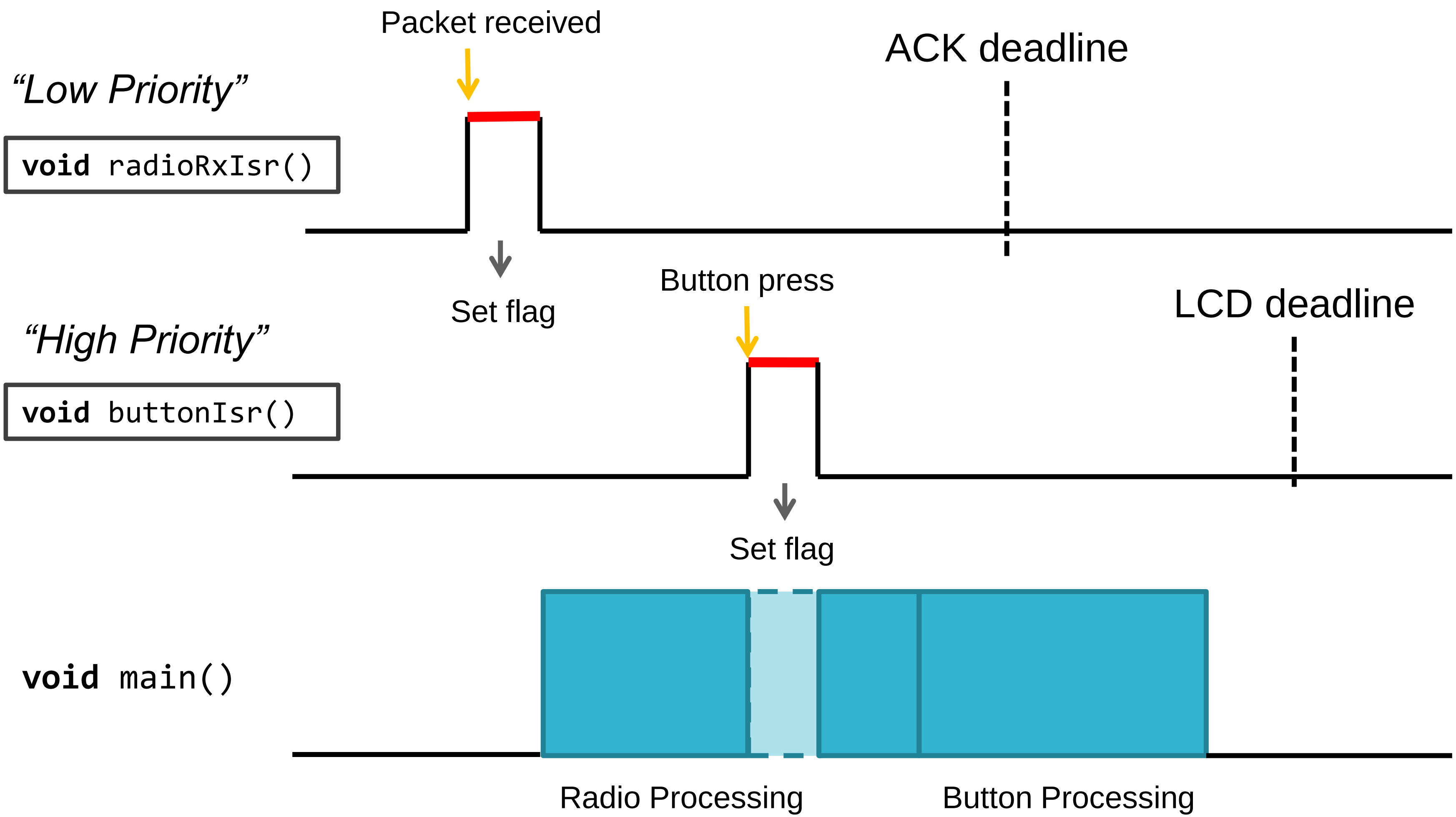
Bare Metal – Smart Thermostat

Case one: Interrupts so far apart that all process is done before next interrupt



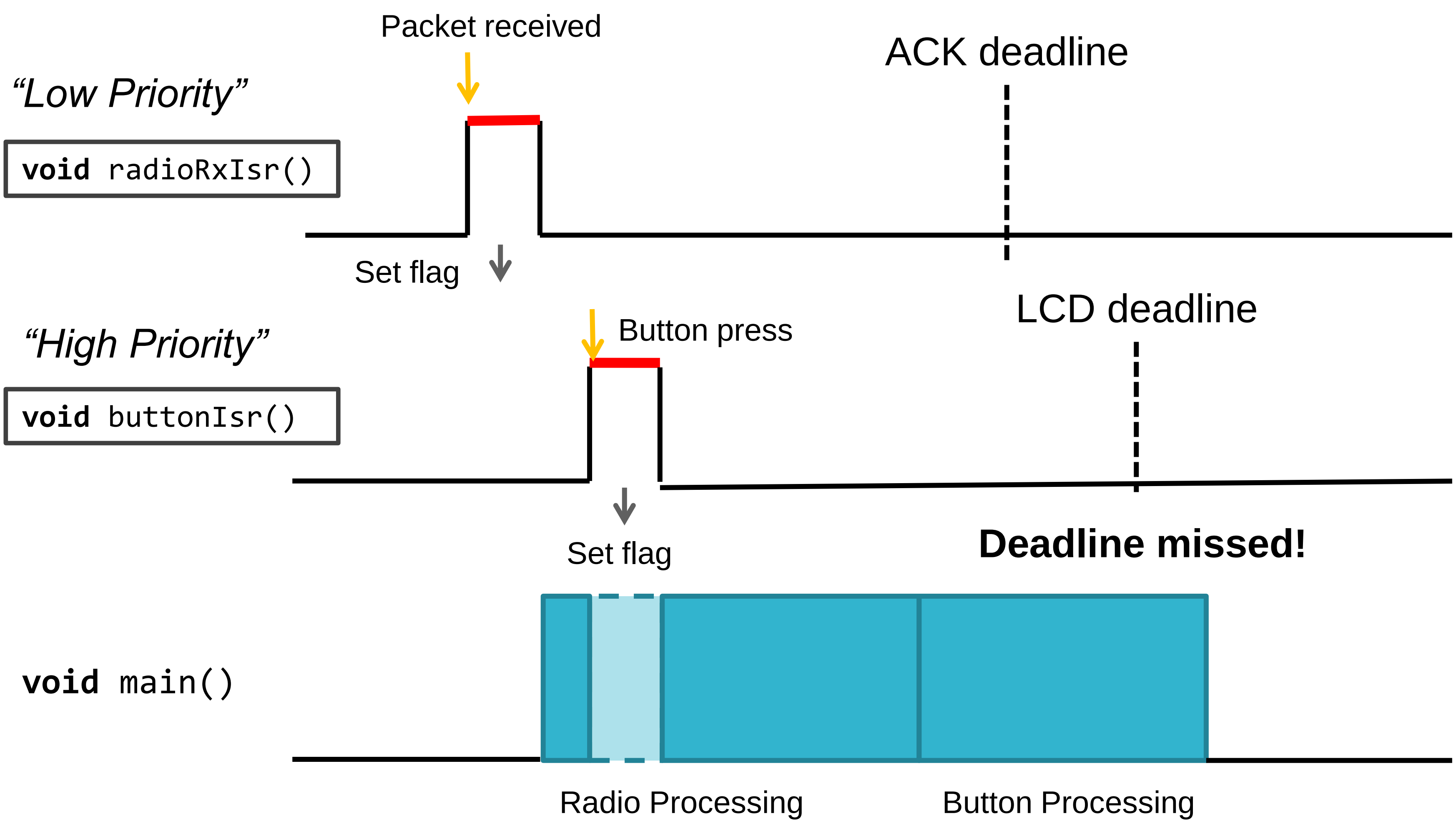
Bare Metal – Smart Thermostat

Case two: Process interrupted, but still time for all processing to finish



Bare Metal – Smart Thermostat

Case three: Low priority task makes higher priority task miss deadline!



Bare Metal – Smart Thermostat

Why did this happen?

```
void main() {
  /* Initialize hardware and setup interrupts */
  while(1) {
    if(buttonSemaphore)
      processButtonPress();       // Takes time
    if(radioRxSemaphore)
      processPacketReceived();    // Takes time
  }
}
```

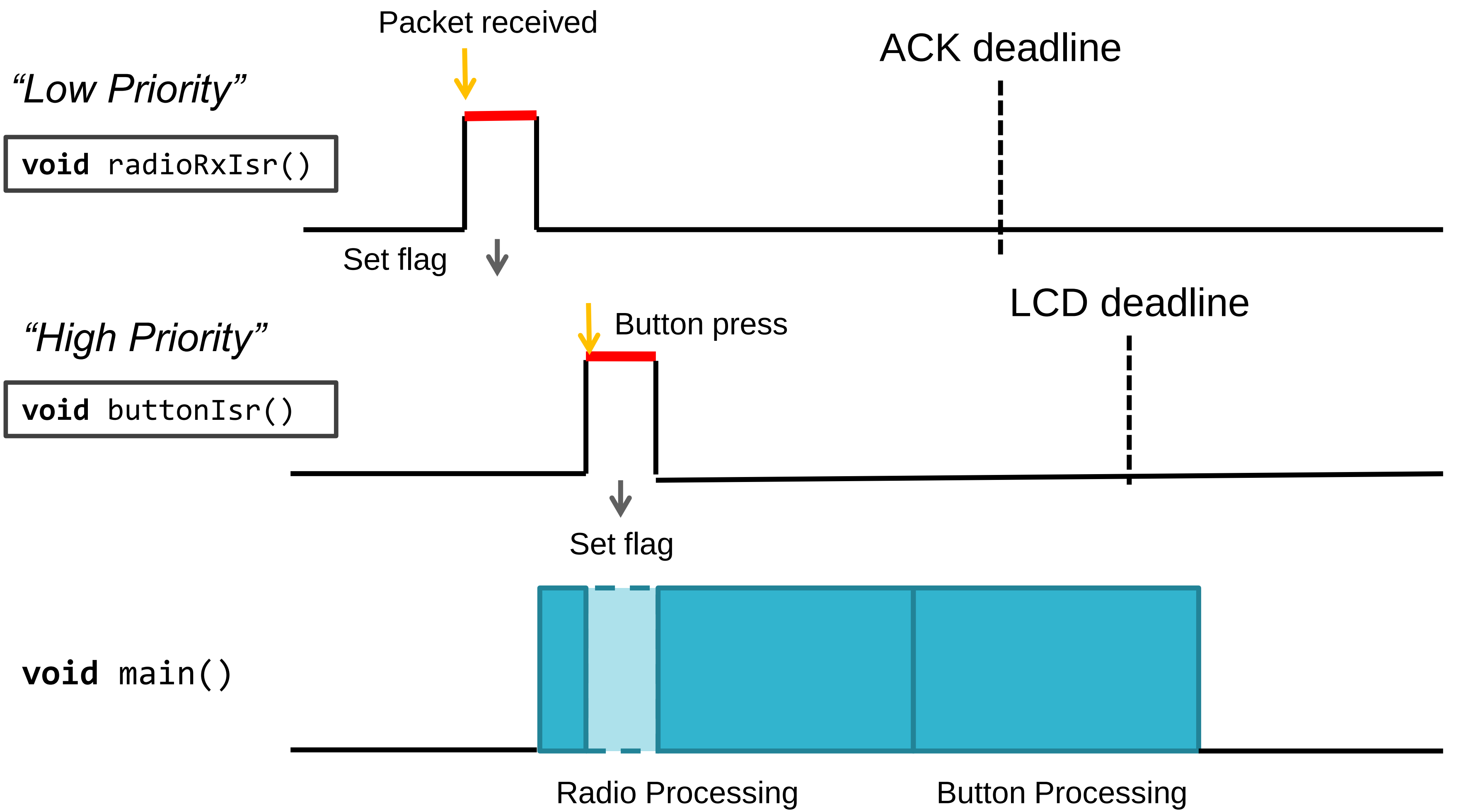
Interrupt service routines

```
void buttonIsr() {
  /* Read buttons */
  buttonSemaphore = 1;
}
```

```
void radioRxIsr() {
  /* Read packet*/
  radioRxSemaphore = 1;
}
```

Bare Metal – Smart Thermostat

How would we have wanted this to work?



Bare Metal – Smart Thermostat

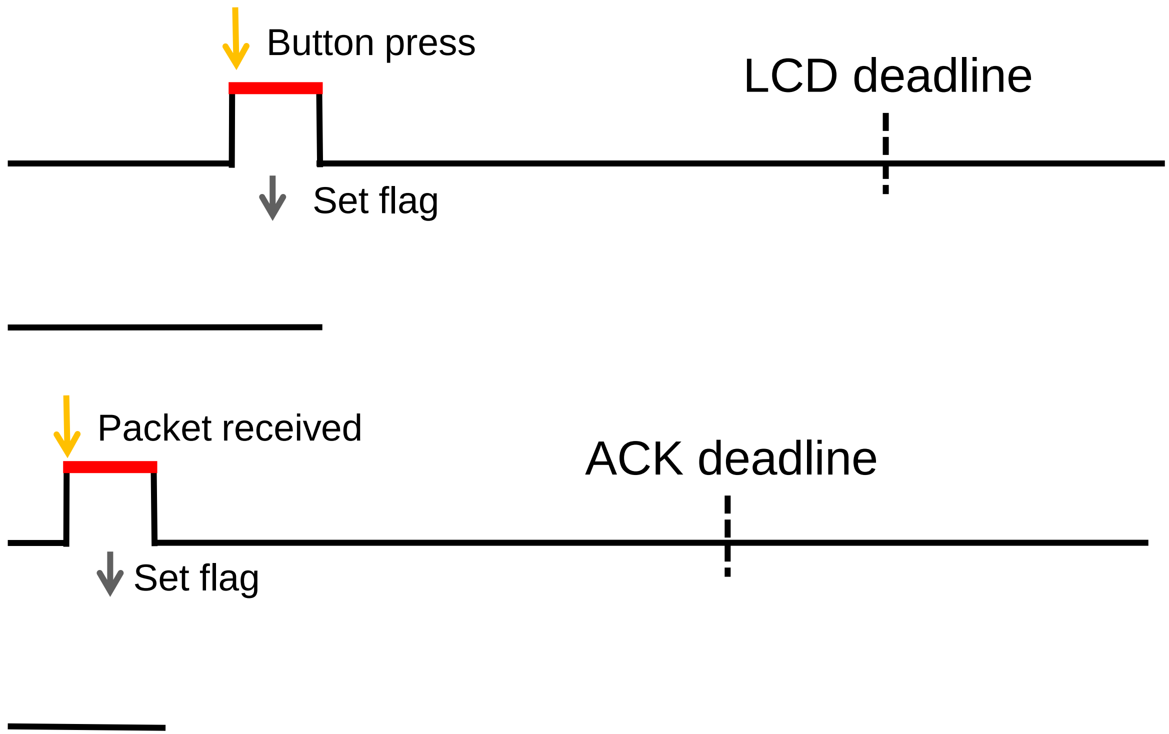
How would we have wanted this to work?

`void buttonIsr()`

“High Priority”
`void processButton()`

`void radioRxIsr()`

“Low Priority”
`void processPacketRx()`

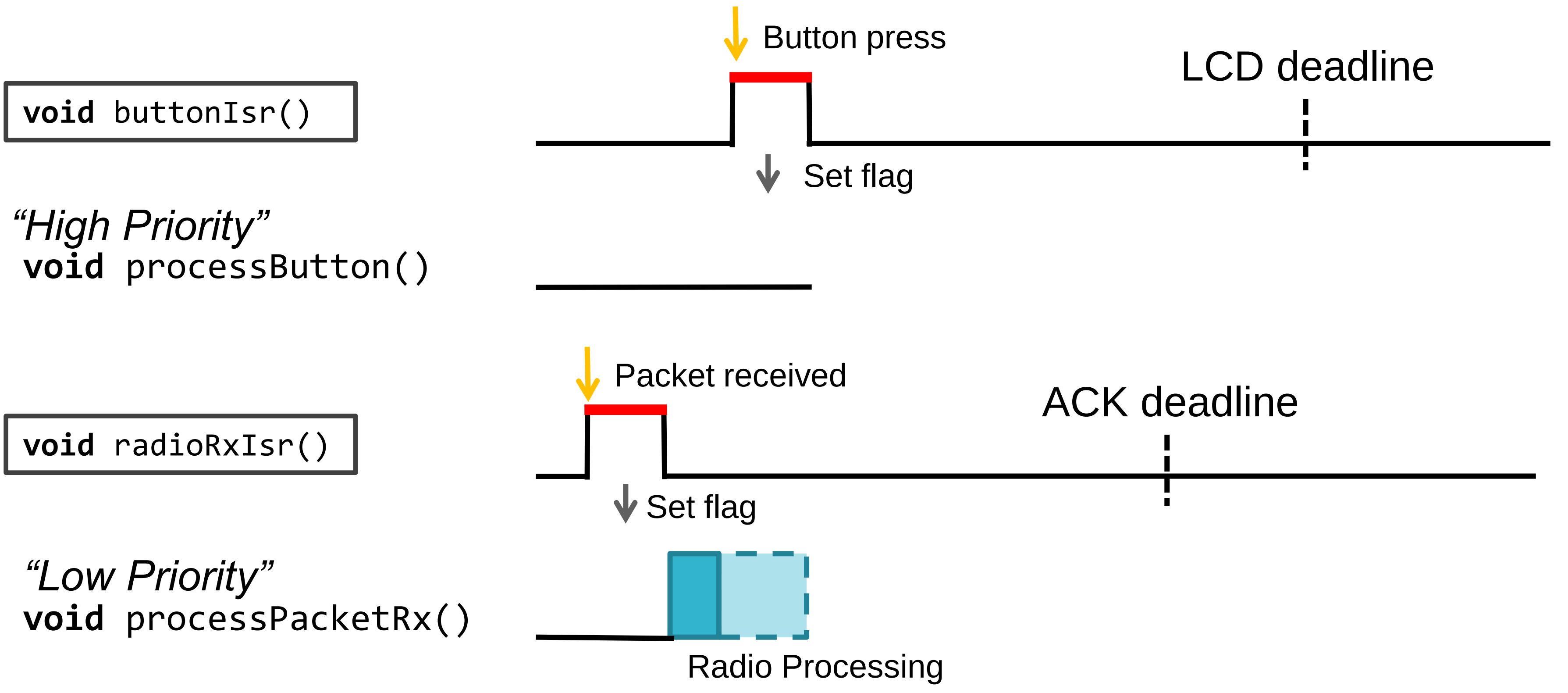


Deadline missed!

Texas Instruments

Bare Metal – Smart Thermostat

How would we have wanted this to work?



Bare Metal – Smart Thermostat

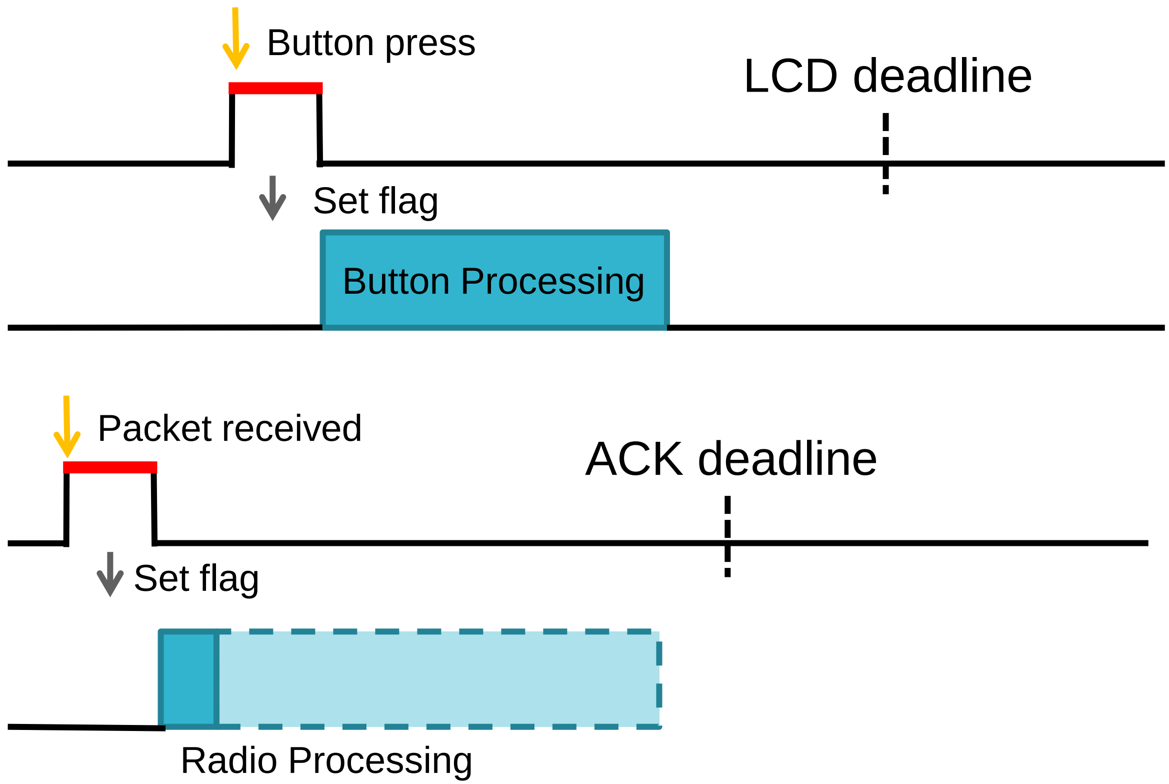
How would we have wanted this to work?

`void buttonIsr()`

“High Priority”
`void processButton()`

`void radioRxIsr()`

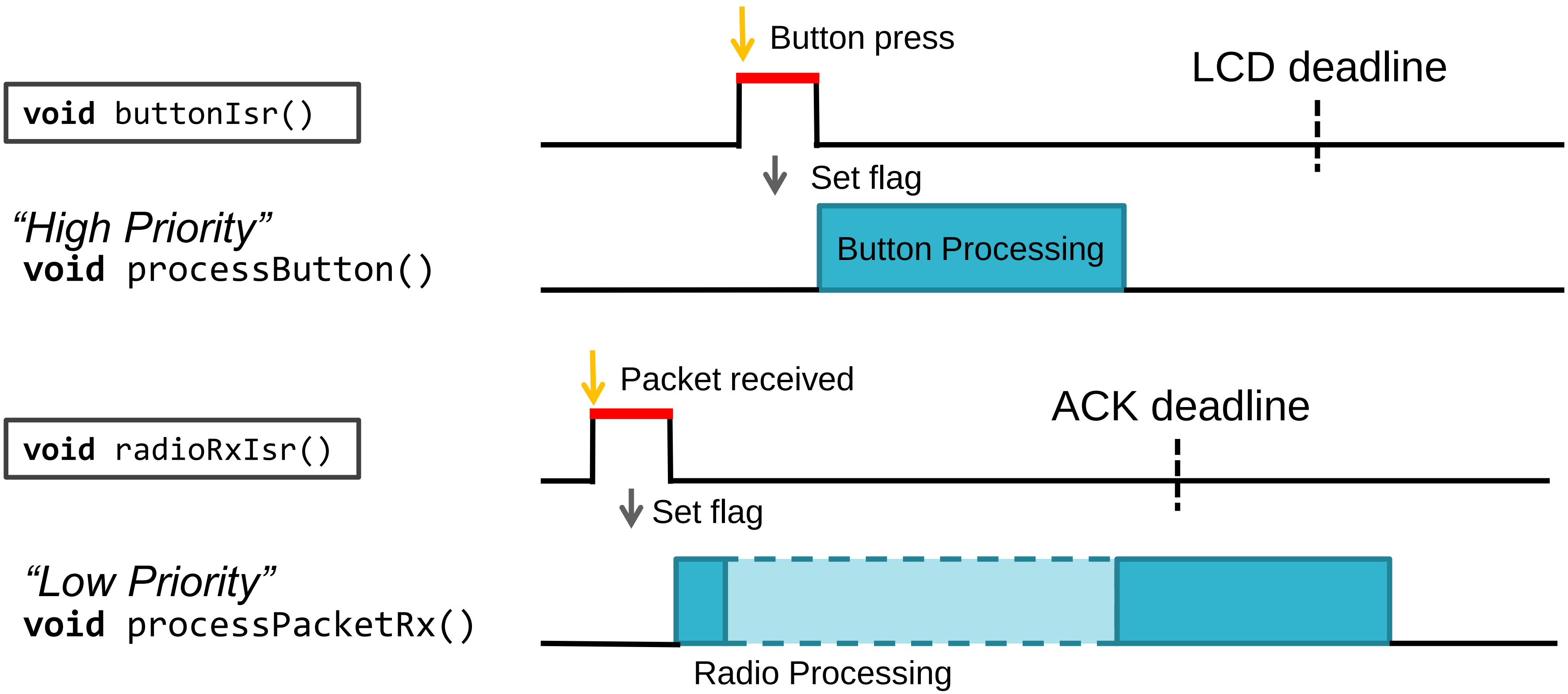
“Low Priority”
`void processPacketRx()`



Bare Metal – Smart Thermostat

How would we have wanted this to work?

Finished in time!



Deadline missed, but that's ok

Bare Metal – Smart Thermostat

Processing running in
- different contexts
- at different priorities

“High Priority”
`void processButton()`

```
void processButton() {
  /* Which button */
  /* Do logic */
  /* Update LCD */
}
```

“Low Priority”
`void processPacketRx()`

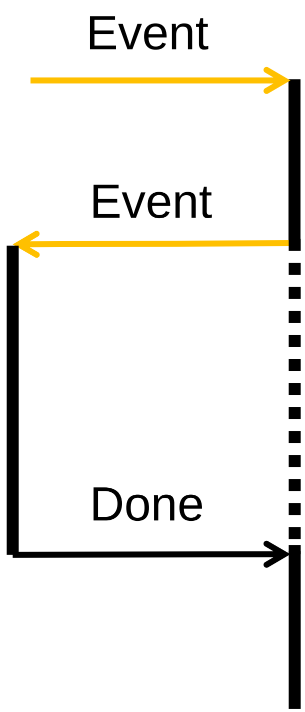
```
void processPacketRx() {
  /* Validate packet */
  /* Do logic */
  /* Send ACK */
}
```

Bare Metal – Smart Thermostat

Processing running in
- different contexts
- at different priorities

"High Priority"
`void processButton()`

```
void processButton() {
  /* Which button */
  /* Do logic */
  /* Update LCD */
}
```



"Low Priority"
`void processPacketRx()`

```
void processPacketRx() {
  /* Validate packet */
  /* Start doing logic */




  /* Continue logic */
  /* Send ACK */
}
```

Bare Metal – Smart Thermostat

This is what an RTOS is all about

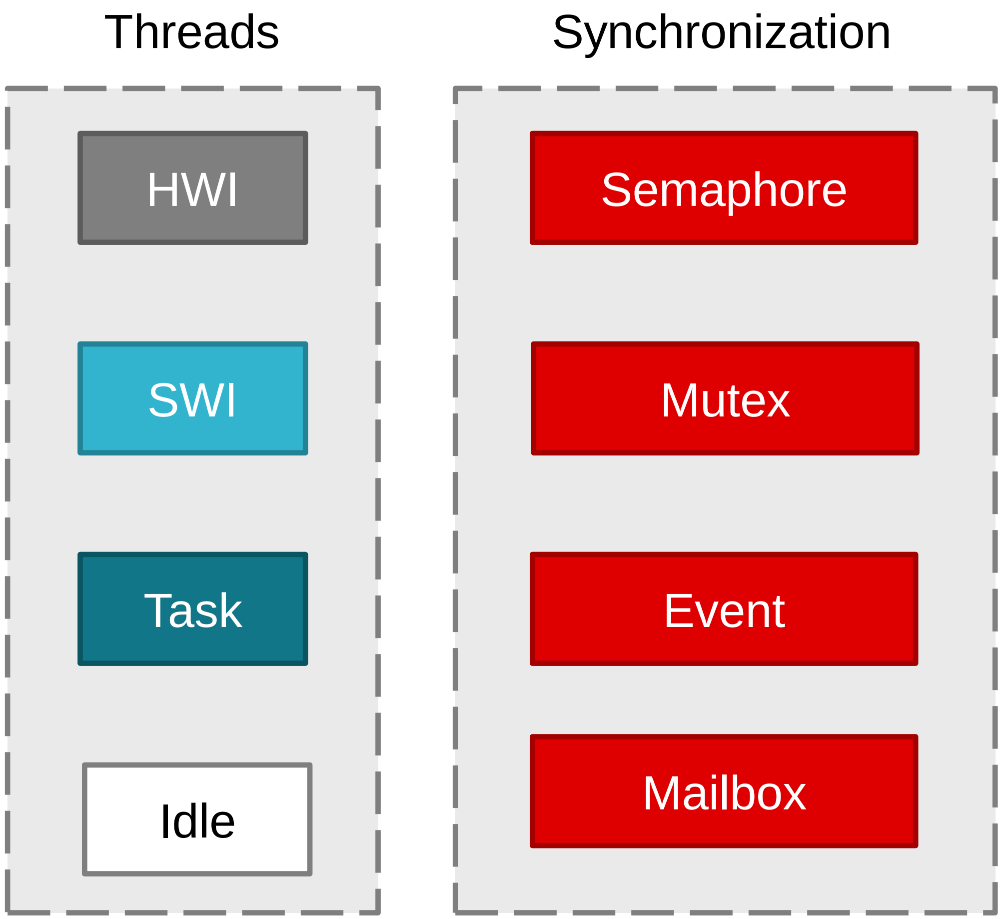
- Each process does its processing in its own context
- An event can trigger both
  - Start of a new process
    - If currently idle
  - Switching the running process
    - If the new process has higher priority than the running one

An RTOS does not do anything that cannot be without an OS, but

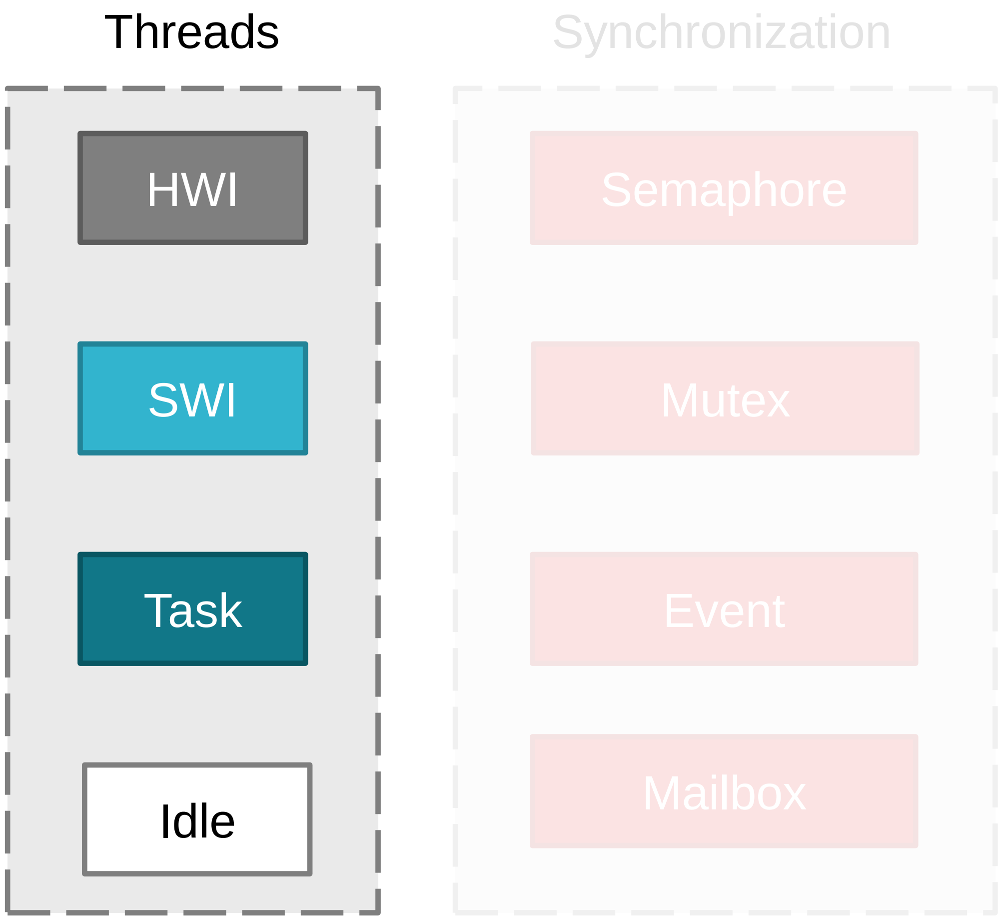
- It makes it simpler to write complex applications
  - Simpler to write event-driven code
  - Simpler to partition code
  - Simpler to guarantee deadlines

TI RTOS Concepts

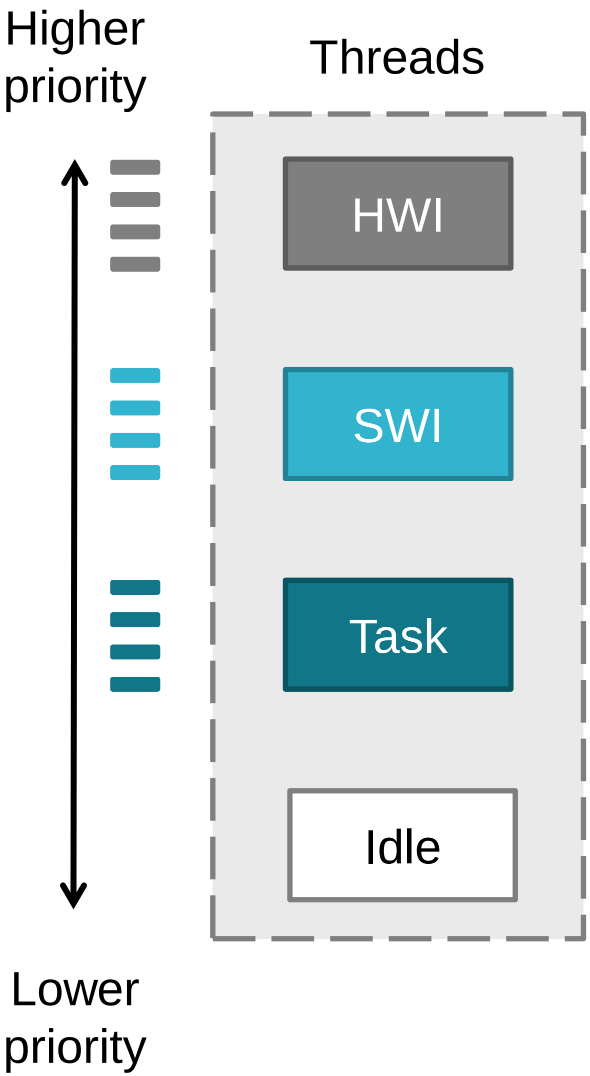
TI RTOS – Kernel Objects Overview



TI RTOS – Kernel Objects Overview



Threads overview



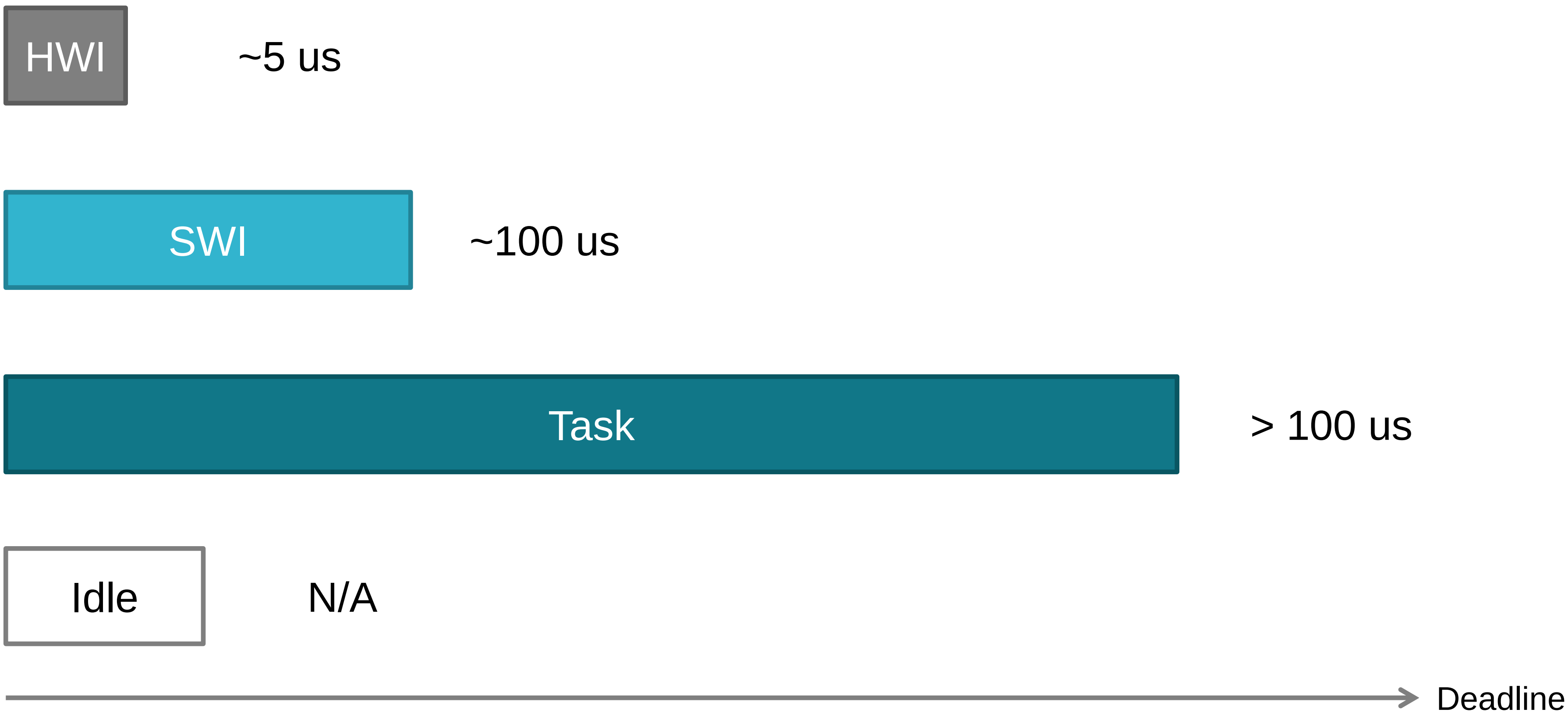
- Handles hardware interrupts
- Priorities set in hardware

- Usually used for HWI follow-up processing
- Up to 32 user configurable priorities

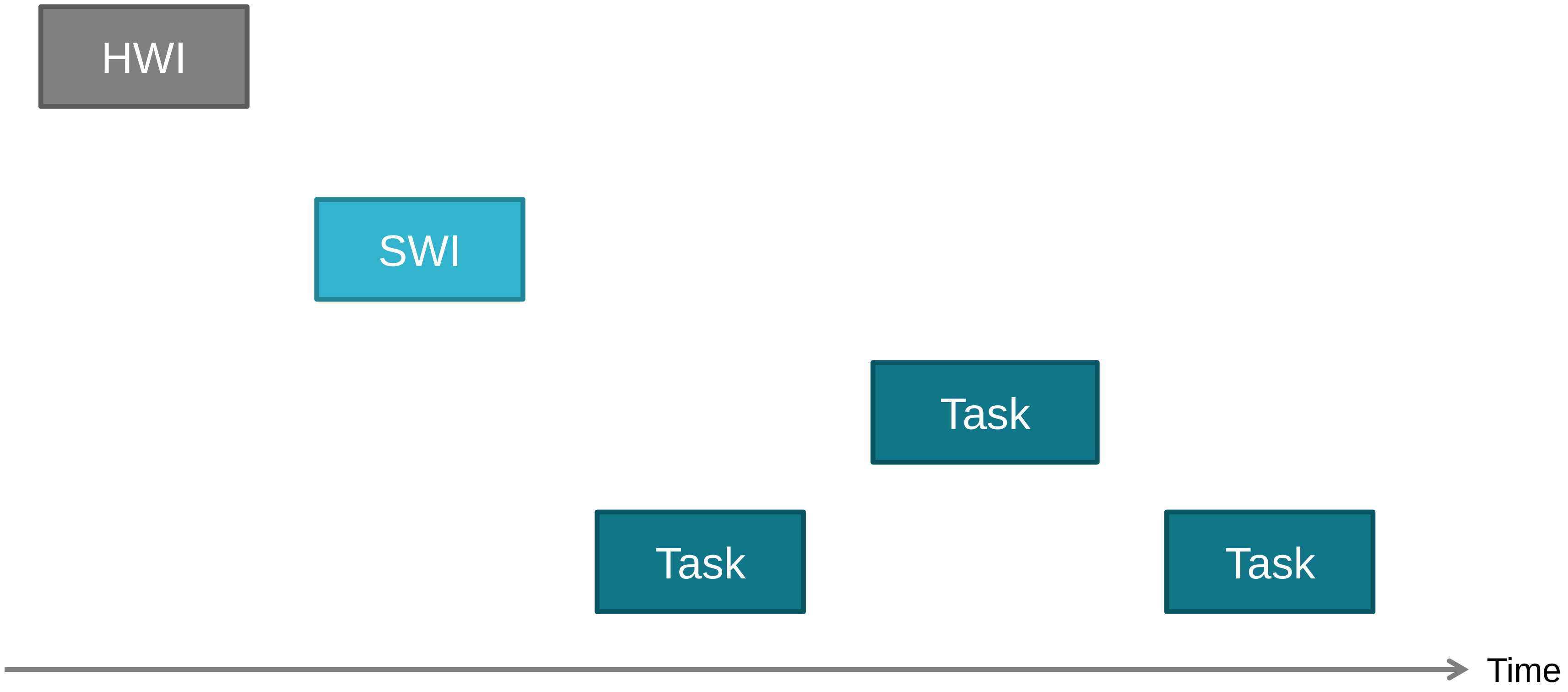
- Used for all complex and/or continuous processing
- Up to 32 user configurable priorities

- Special thread – Runs when nothing else is running

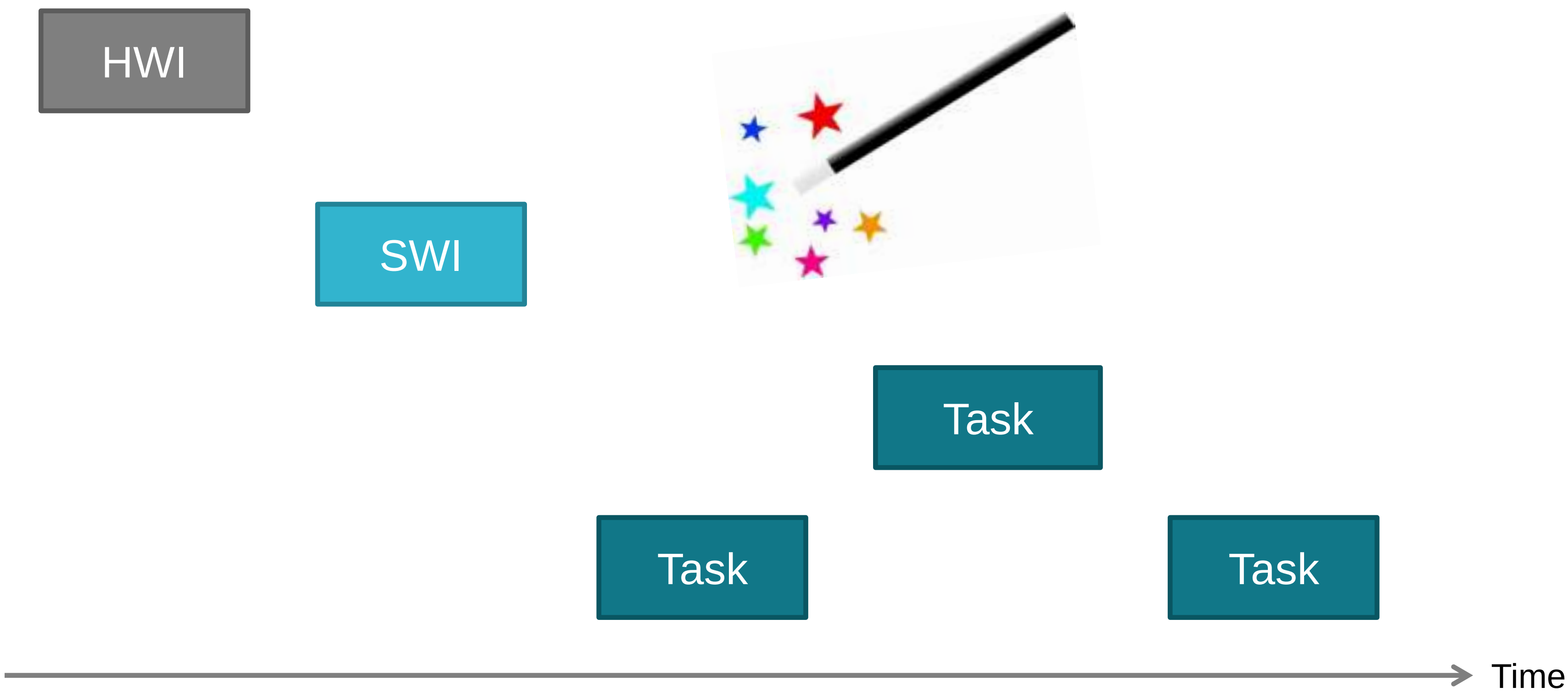
Threads and deadlines



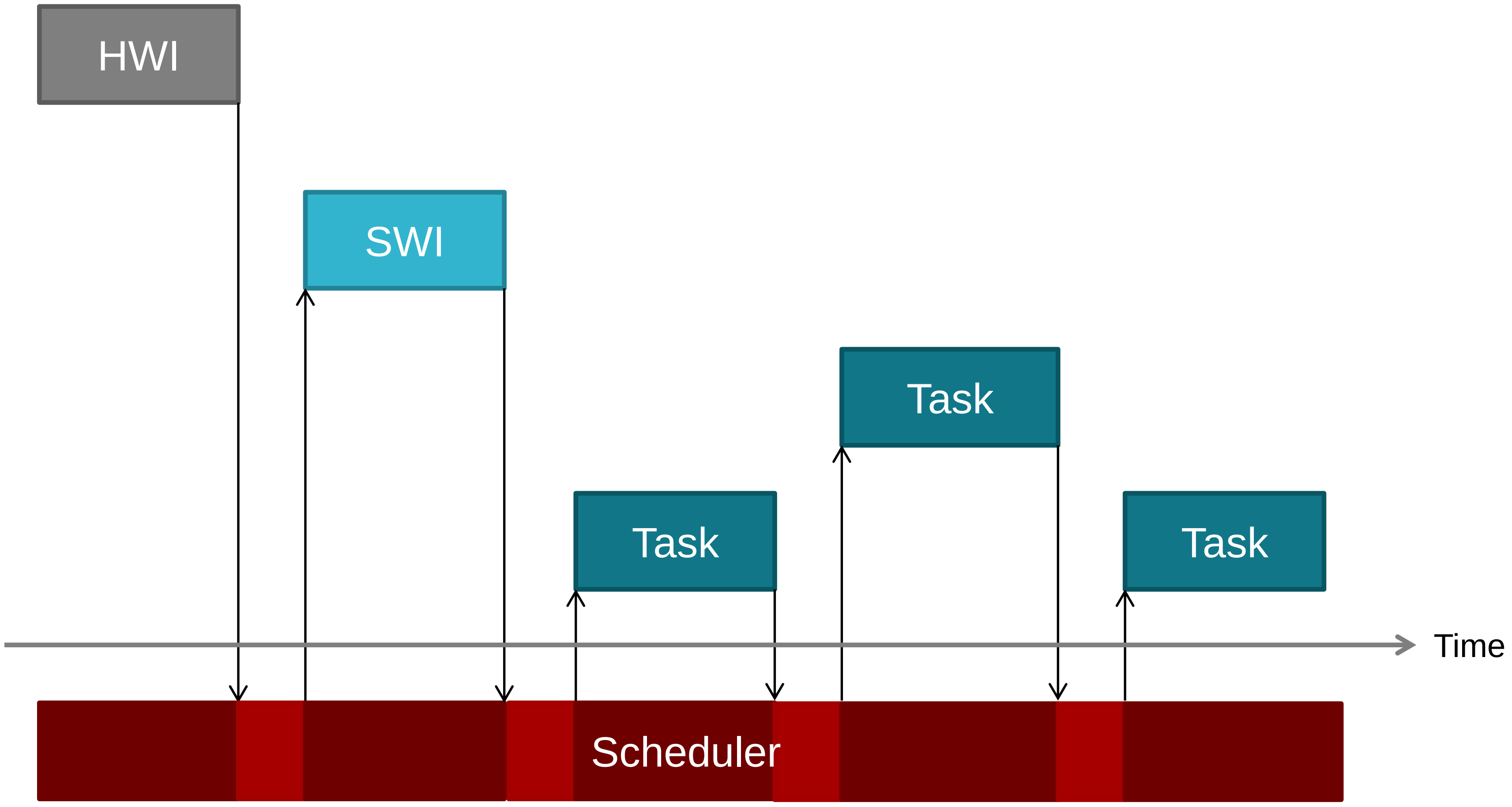
The RTOS scheduler



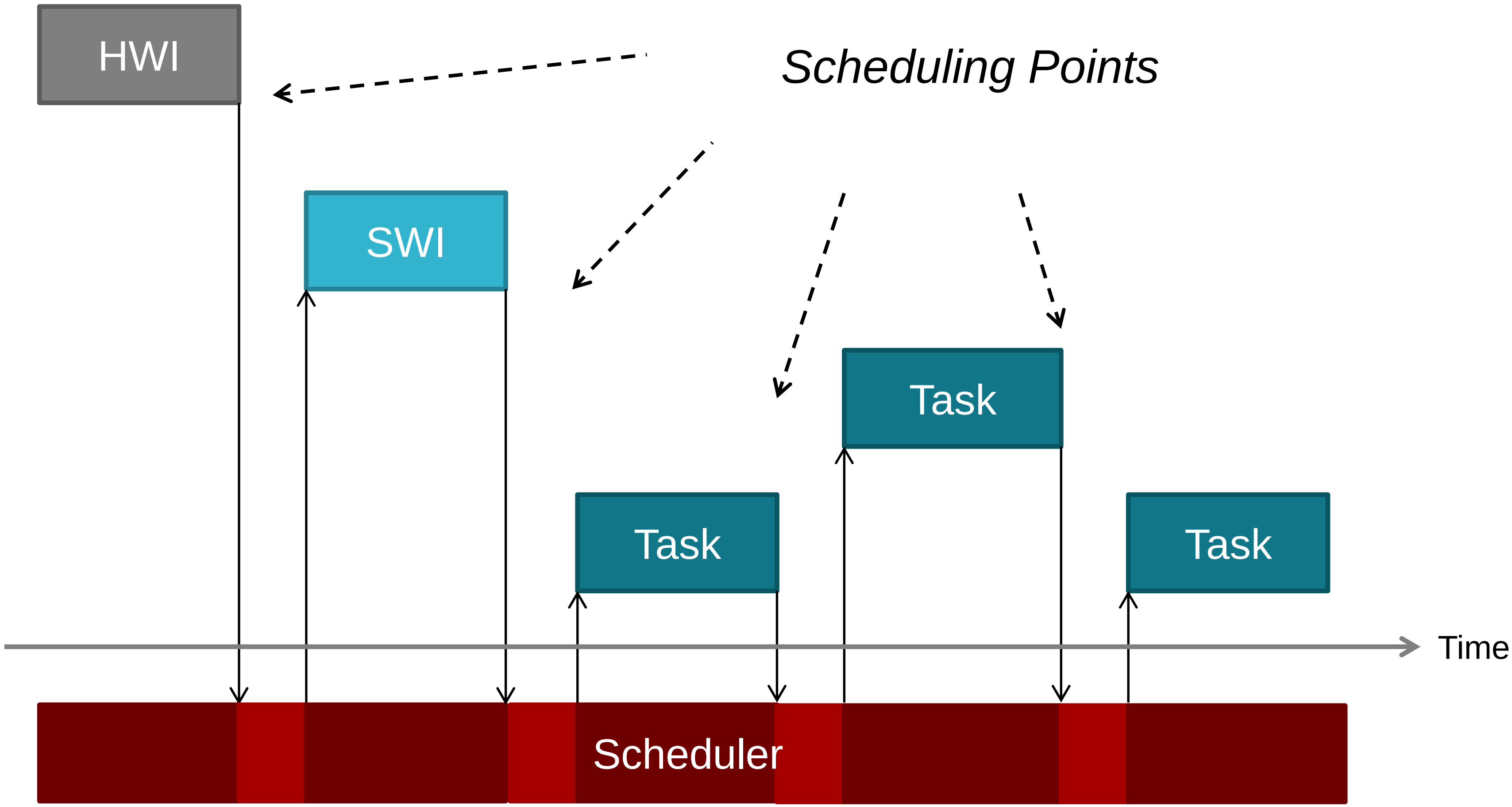
The RTOS scheduler



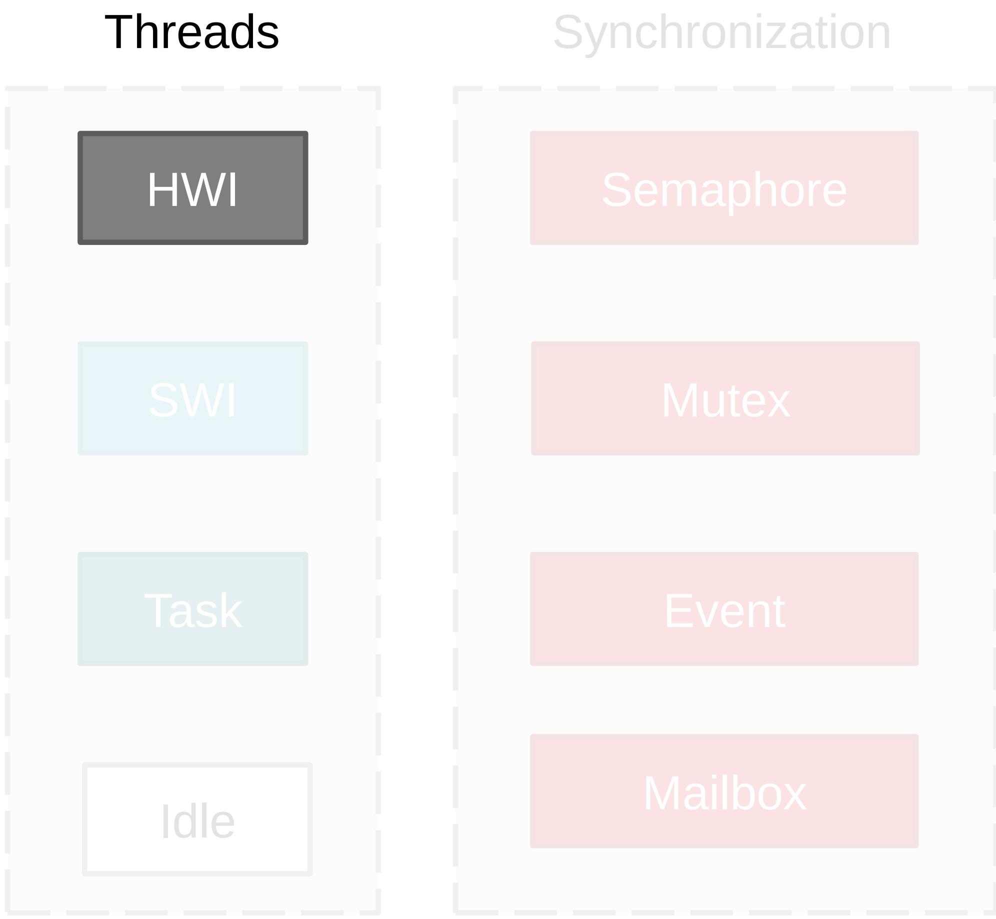
The RTOS scheduler



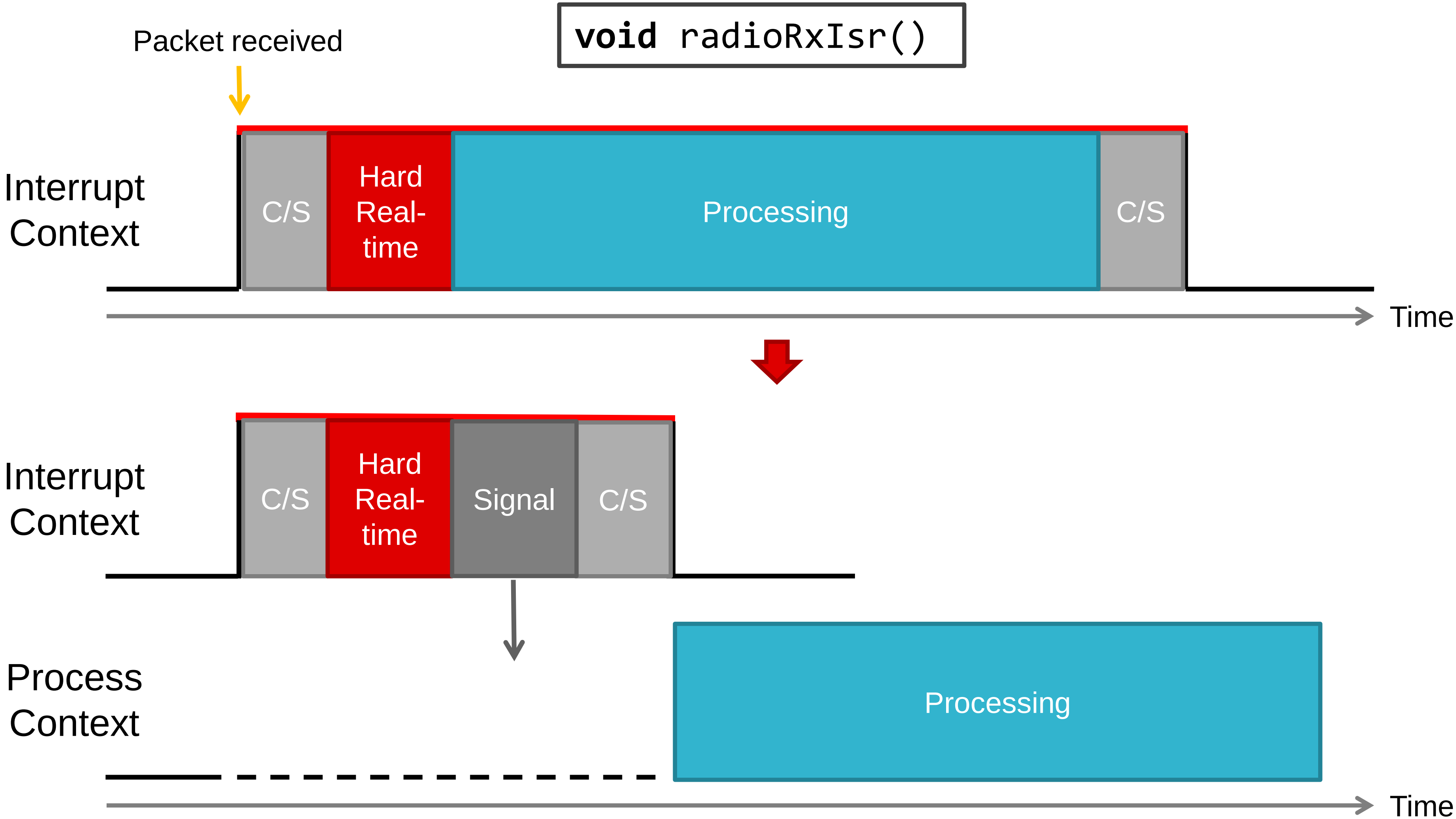
The RTOS scheduler



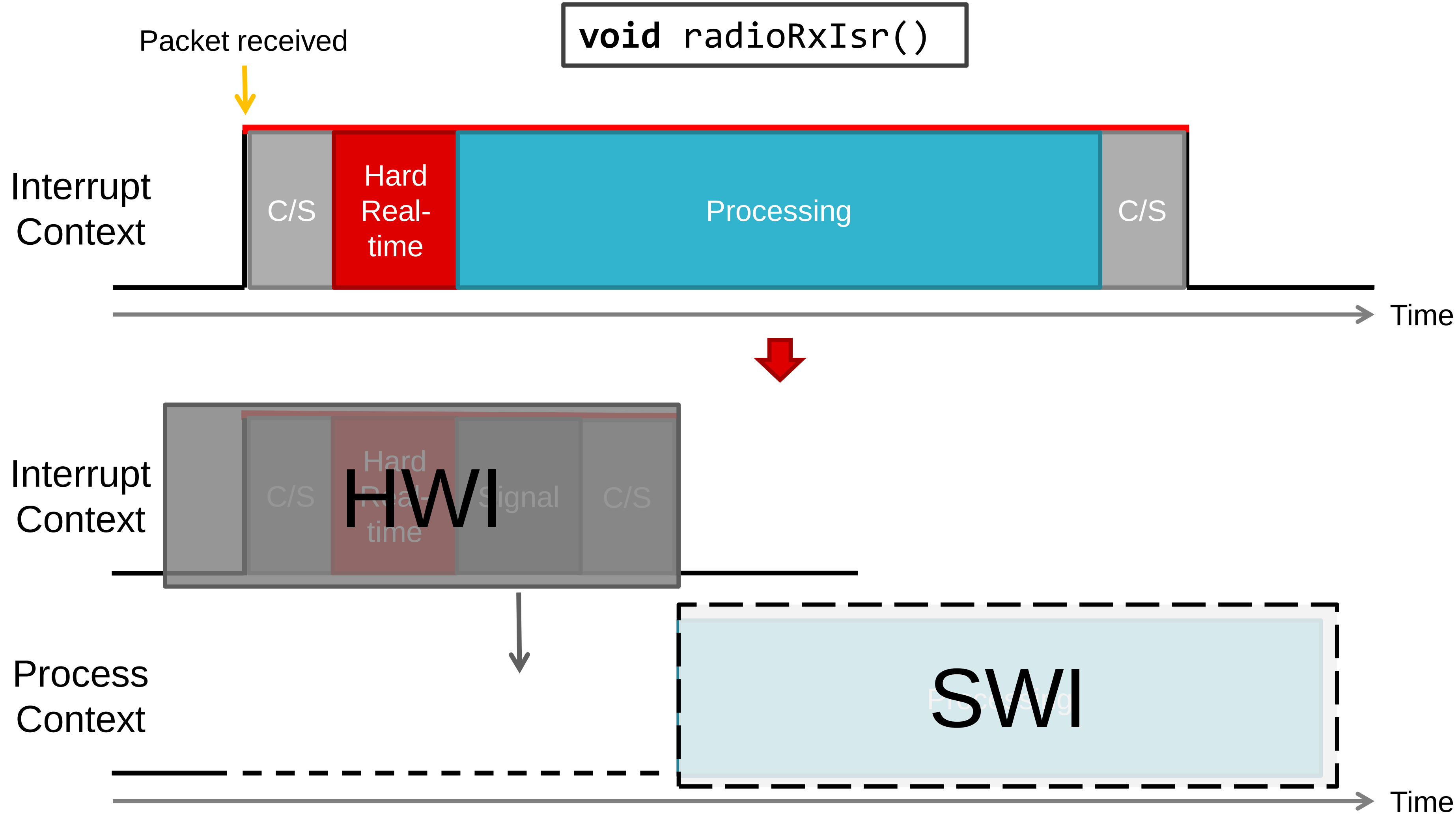
TI RTOS – HWI


Threads
Synchronization
HWI
SWI
Task
Idle
Semaphore
Mutex
Event
Mailbox

Bare Metal – Smart Thermostat



HWI – Hardware Interrupt



HWI – Hardware Interrupt

HWI's handles all hardware interrupts

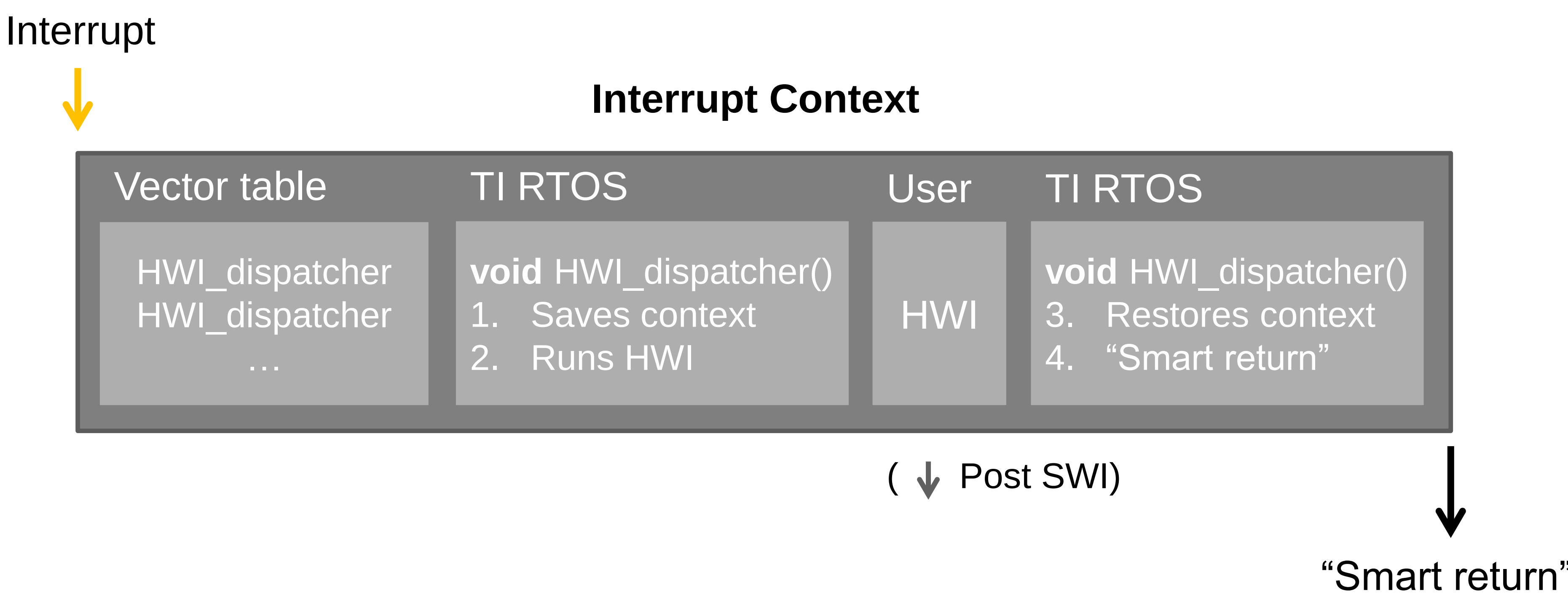
- Highest priority of all
- Always runs to completion
  - Even if interrupt happens again while currently being handled
- All HWI's shares the System Stack

Handle hardware interrupts

1. Using the RTOS HWI dispatcher (recommended)
2. Using regular, pure bare metal, interrupts

HWI – HWI dispatcher

- **Recommended** and the default
- All interrupts are mapped to the same handler
- Handles interrupt nesting
  - Default is that any interrupt preempts any other, except itself
- **Must** use this to use TI RTOS system call in ISR



TI RTOS – Smart Thermostat

Looking back

“Smart HWI return”
Returns to highest priority process

HWI

`void buttonIsr()`

SWI

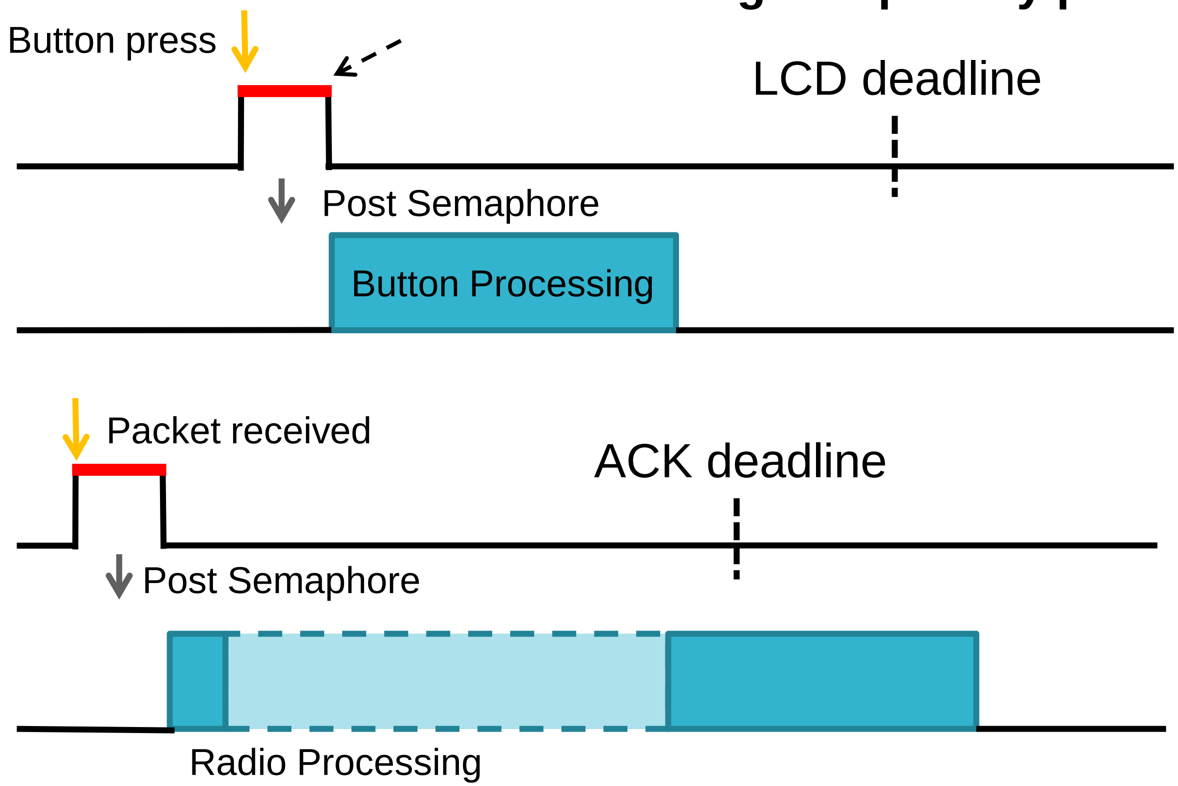
“High Priority”
`void processButton()`

HWI

`void radioRxIsr()`

SWI

“Low Priority”
`void processPacketRx()`

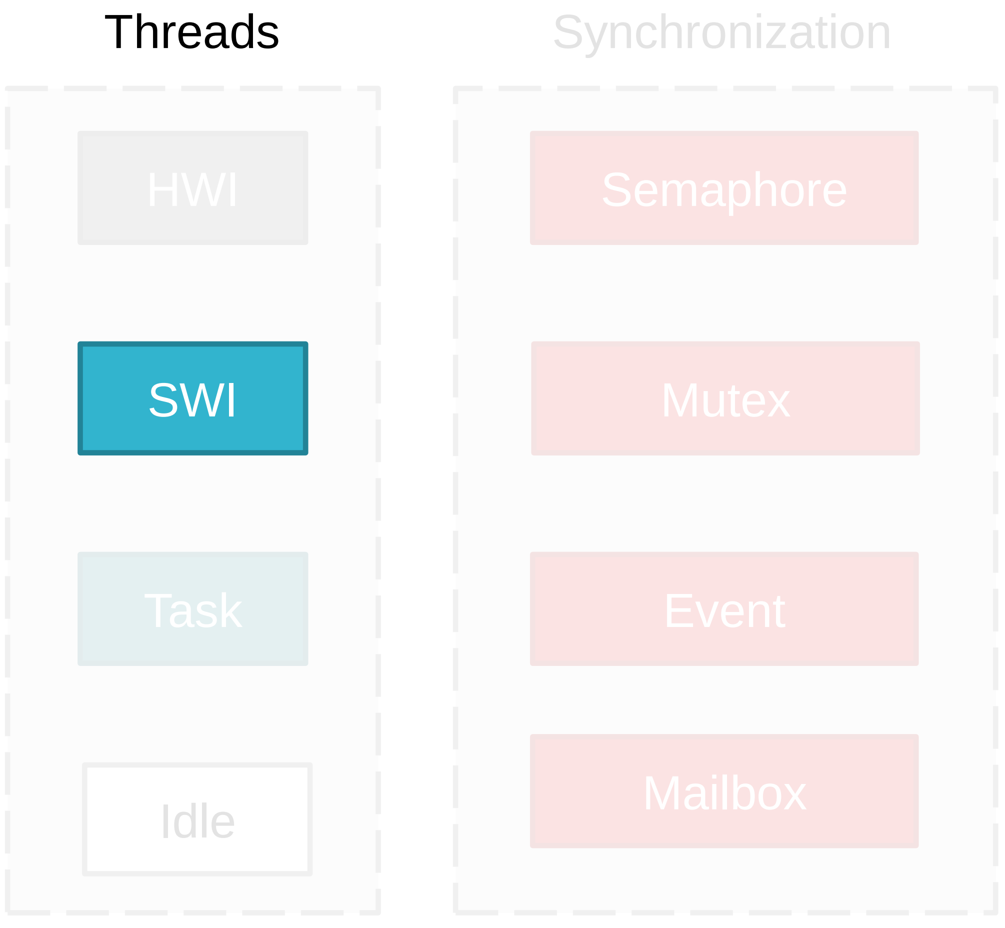


HWI – Hardware Interrupt

- Using regular interrupt handlers
- **Not recommended**
  - Cannot use any RTOS calls
- Must **explicitly** save and restore the RTOS context
- Must manually handle nesting
- Should only be used for extremely low latency ISR:s

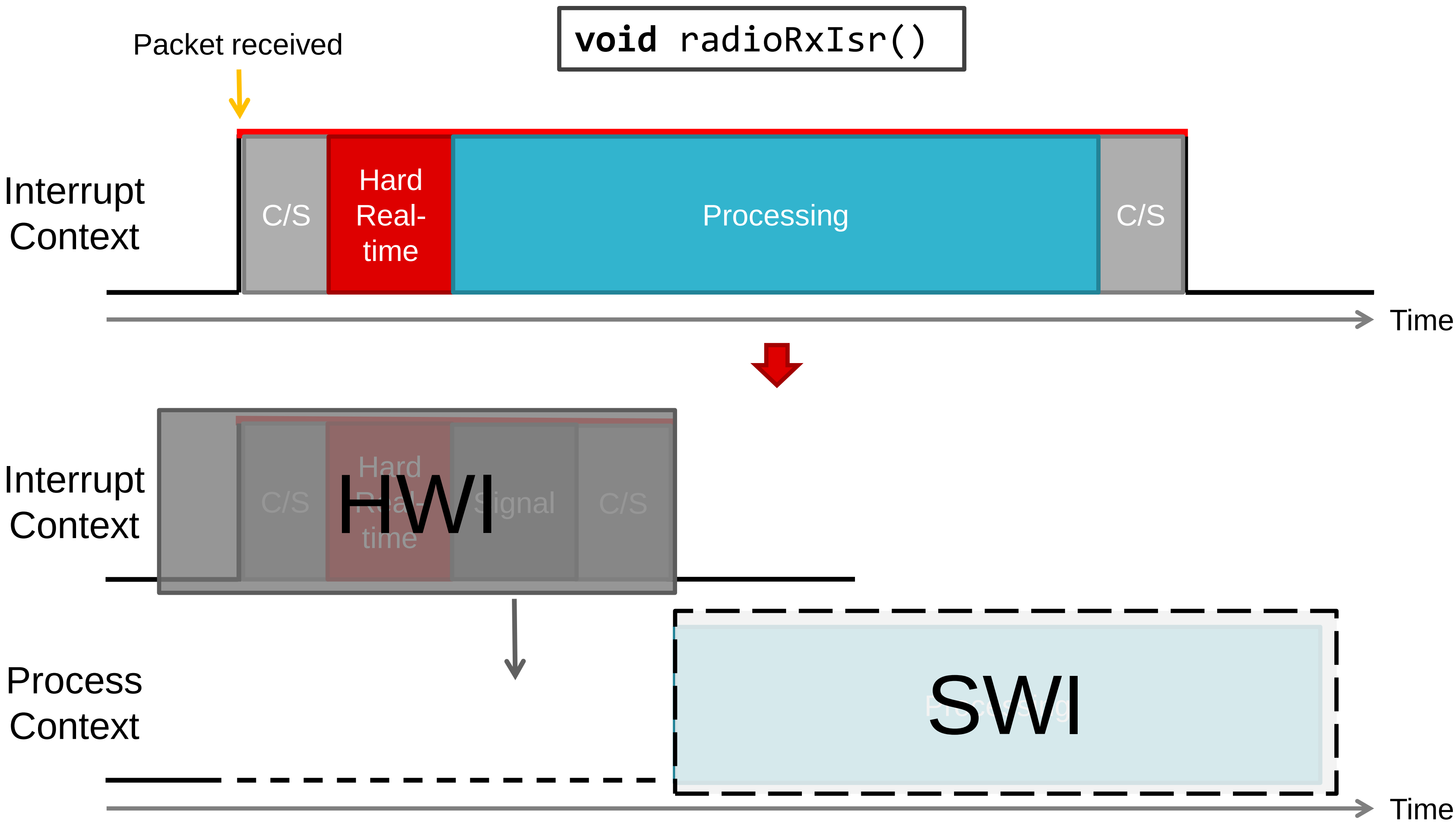
```
void timerISR() {
  /* Save RTOS context */
  /* Do HW close stuff */
  /* Do processing */
  /* Restore RTOS context */
}
```

TI RTOS – SWI

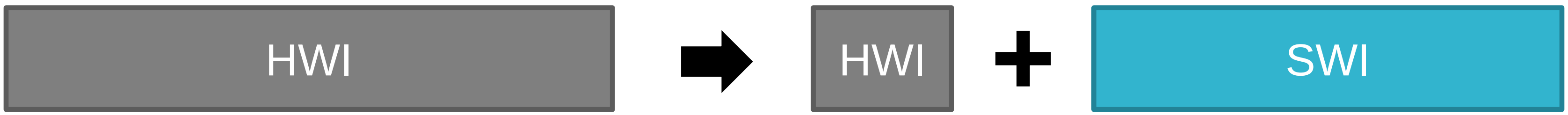


Threads
Synchronization
HWI
SWI
Task
Idle
Semaphore
Mutex
Event
Mailbox

HWI – Hardware Interrupt



SWI vs. HWI



Move the processing from the HWI to a SWI

- Reduces the time in HW interrupt context
- Reduces the worst case HW interrupt latency
- Reduces need for HWI nesting
- Reduces the effects a HW interrupt has on high priority processing

What do we keep in the HWI?

- Hard real-time code such as reading from a HW register as fast as possible
- Storing that information at a shared location
- Posting a semaphore to the SWI that the information is available

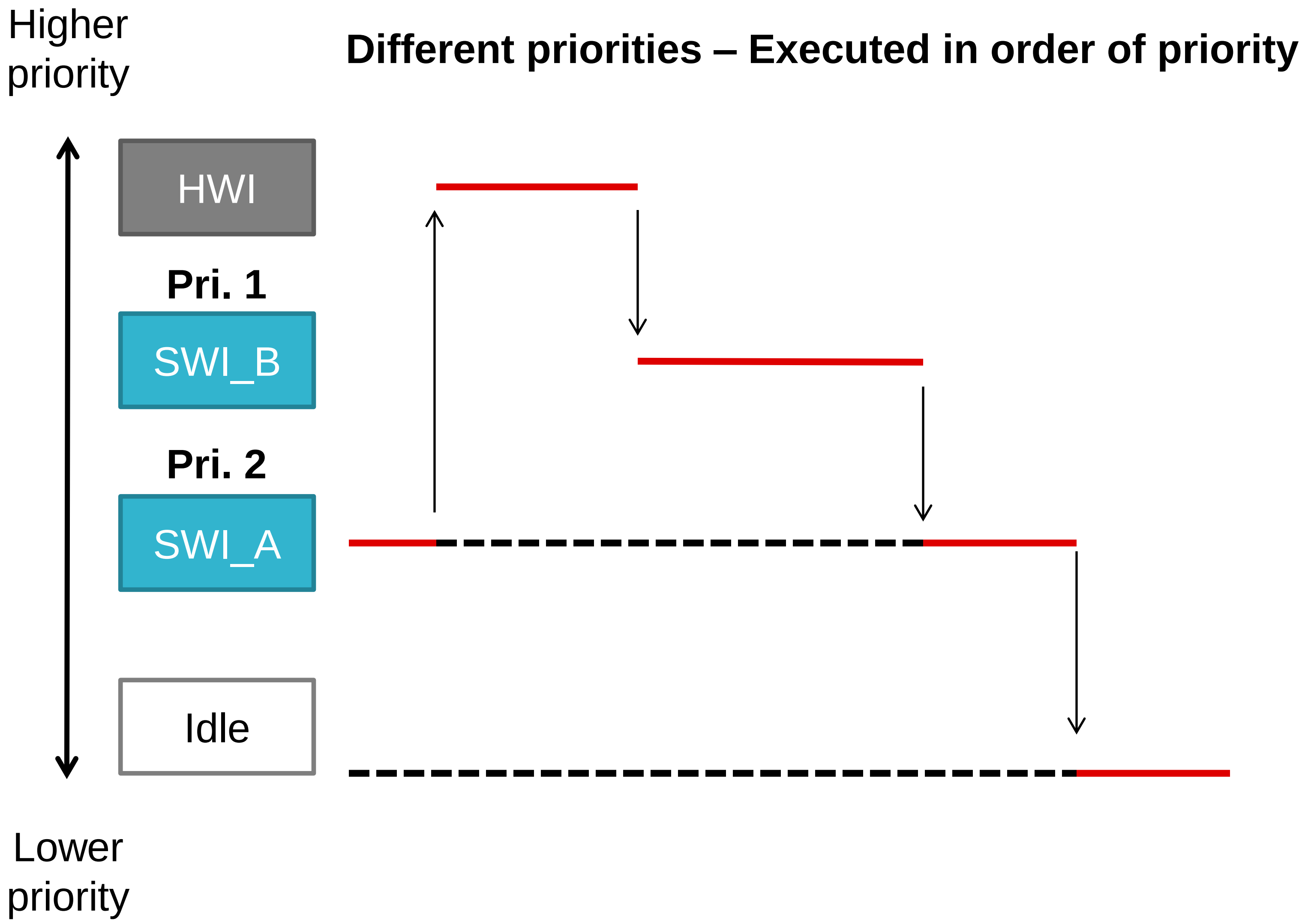
SWI properties

SWI states

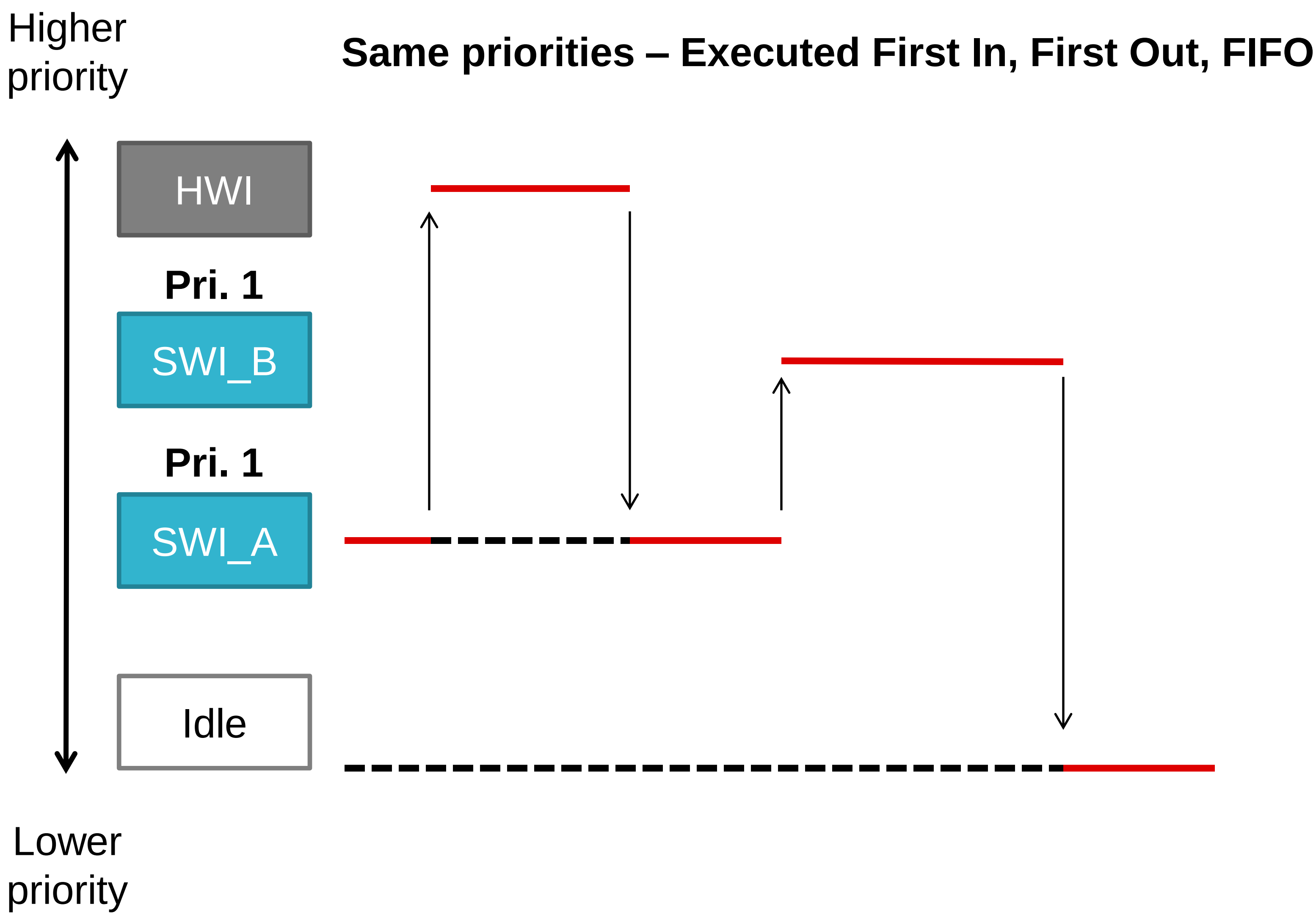
- **Running**
  - The SWI is currently executing
  - Only **one** Thread can be Running at any given time
- **Ready**
  - The SWI is ready to run, but is not since a higher priority SWI is Running
  - Any number of tasks can be Ready
- (Cannot be Blocked!)

- **Shares** the System Stack with all HWI and other SWI's
  - Important to take into account that several SWI and/or HWI may preempt each other, increasing the stack usage

SWI – Software Interrupt

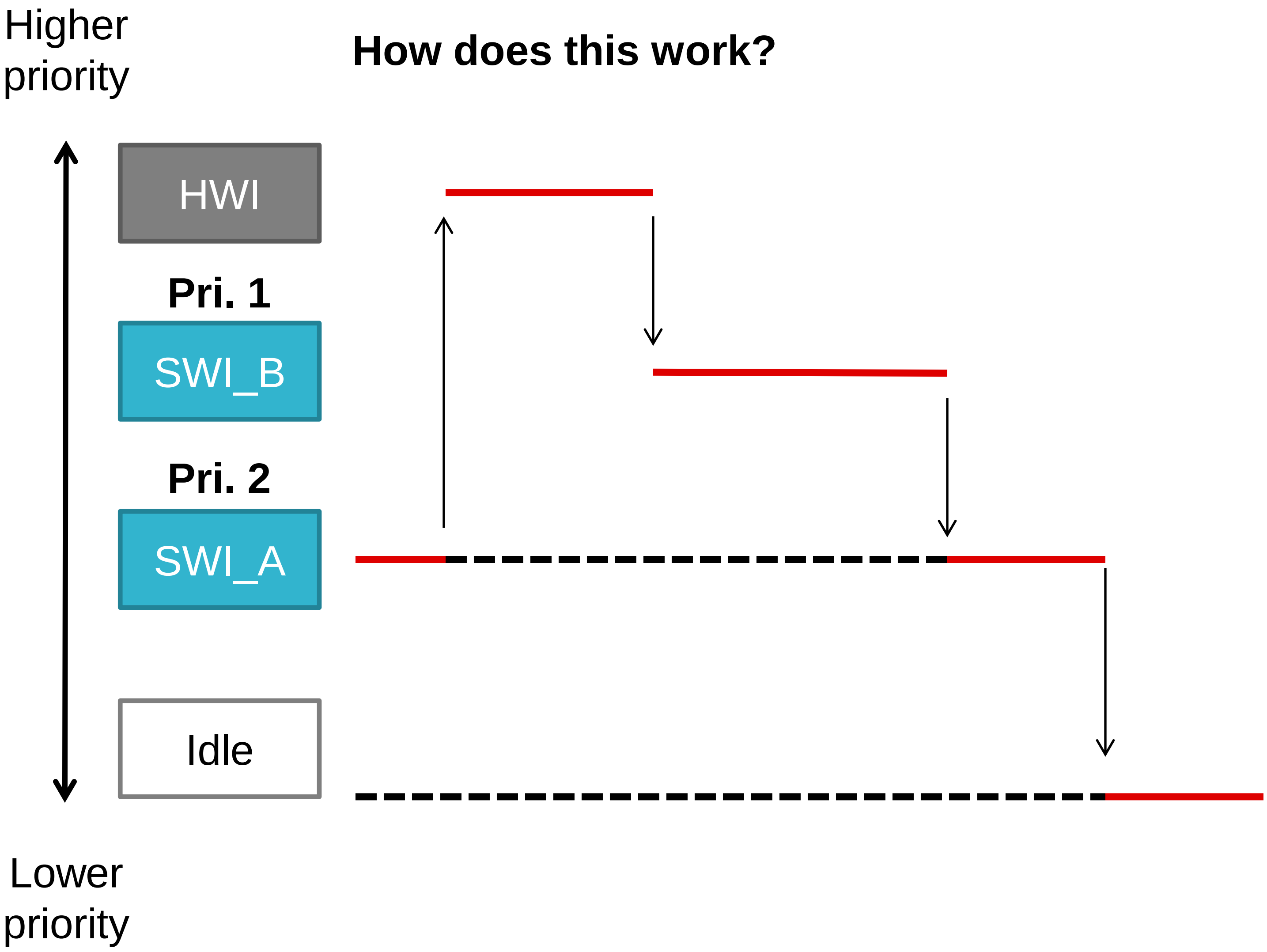


SWI – Software Interrupt



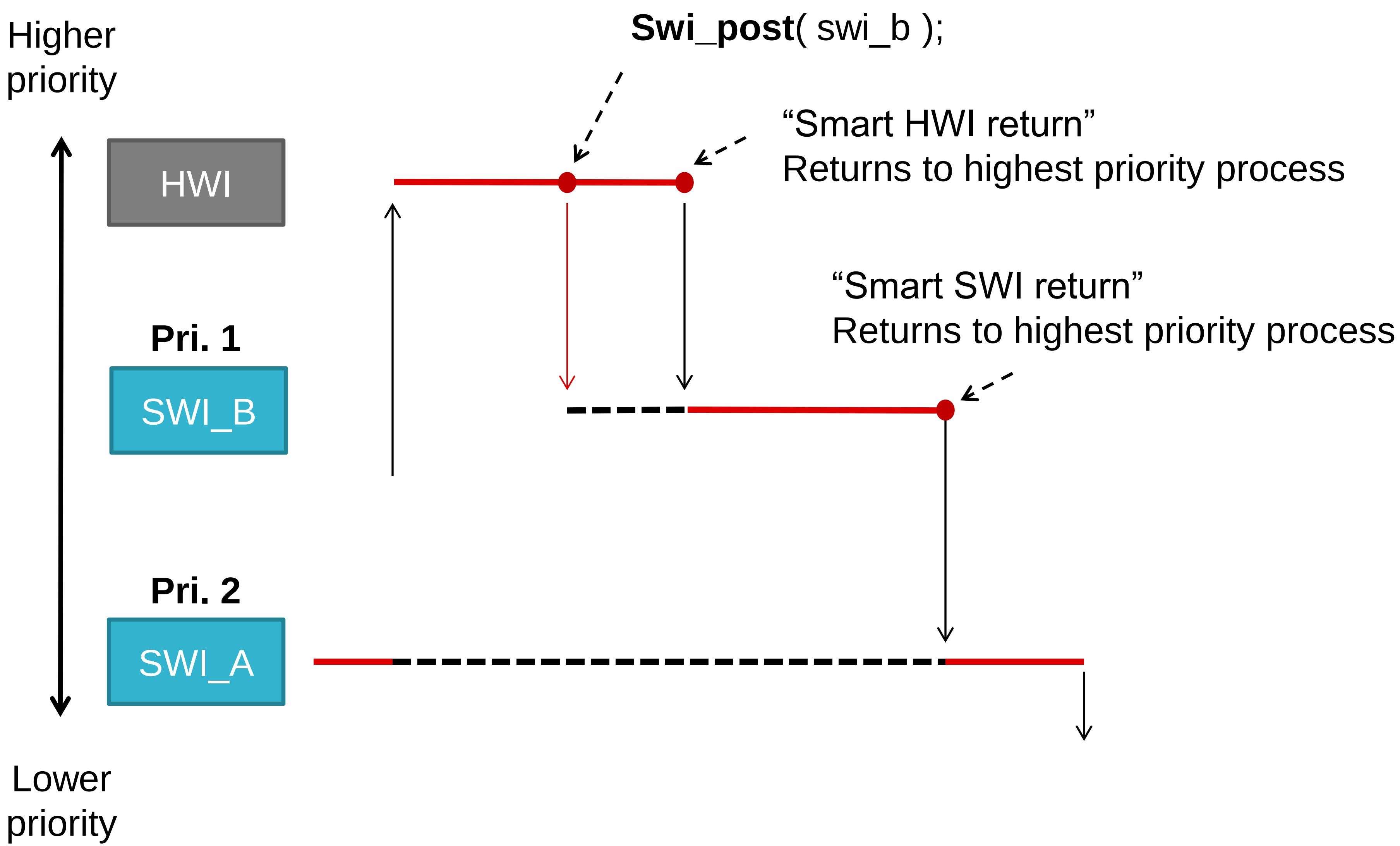
SWI – Software Interrupt

How does this work?



Texas Instruments

SWI – Software Interrupt



SWI API

API	Description
Swi_inc()	Post, increment count in trigger
Swi_dec()	Decrement count, post if trigger = 0
Swi_or()	Post, OR bit into trigger
Swi_andn()	Zero a bit in trigger, Post if trigger = 0
Swi_getPri()	Get any Swi Priority
Swi_raisePri()	Raise priority of any Swi
Swi_getTrigger()	Get any Swi's trigger value
Swi_enable()	Global Swi enable
Swi_disable()	Global Swi disable
Swi_restore()	Global Swi restore

What about continuous processing?

TI RTOS – Task

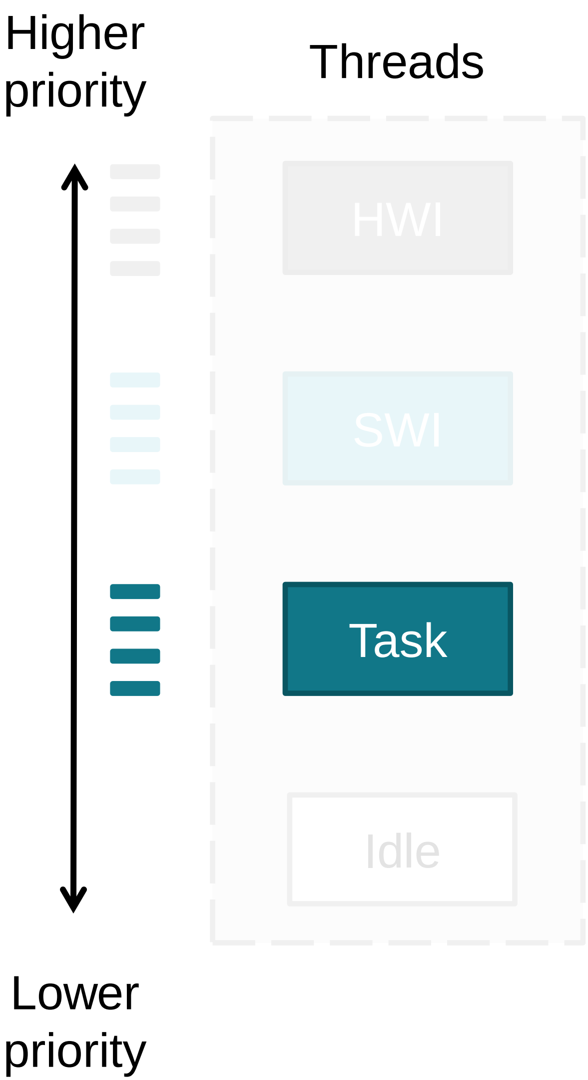
Threads

- HWI
- SWI
- Task
- Idle

Synchronization

- Semaphore
- Mutex
- Event
- Mailbox

Threads overview



- Handles hardware interrupts
- Priorities set in hardware

- Usually used for HWI follow-up processing
- Up to 32 user configurable priorities

- Used for all complex and/or continuous processing
- Up to 32 user configurable priorities

- Special thread – Runs when nothing else is running

SWI vs. Tasks

SWI

Intended for HWI follow-up processing

- Executed once per HWI
- Does not keep any local state between calls
- Uses the System Stack

```
void rxSwi() {
  /* Read FIFO bytes */
  /* Write to buffer */
}
```

Task

Intended to run **continuously** and concurrently

- Normally runs forever
    - Pending while waiting for data
    - Processes data
    - Then pends again
- Keeps local state
- Has its own stack

```
void packetEngineTask() {
  while(1) {
    /* Wait for bytes */
    if(bytesInBuffer == len)
      /* Decode packet */
  }
}
```

SWI vs. Tasks

Task

- Runs as soon as the system is started, or the task is created.
- Generally never exists, although it can

```
void task() {

  /* Initialization – Run once */

  while(1) {
    /* Wait for Semaphore */
    /* Processing */
  }
}
```

Task states

- Running
  - The Task is currently executing
  - Only **one** Task can be Running at any given time
- Ready
  - The Task is ready to run, but is not since a higher priority Task is Running
  - Any number of tasks can be Ready
- Blocked
  - A task is blocked because it's waiting for a shared resource that is not available
  - Any number of tasks can be Blocked
- (Inactive)
  - Priority set to -1, will not be scheduled
- (Terminated)
  - Task has exited

TI RTOS – Task

Threads

- HWI
- SWI
- Task
- Idle

Synchronization

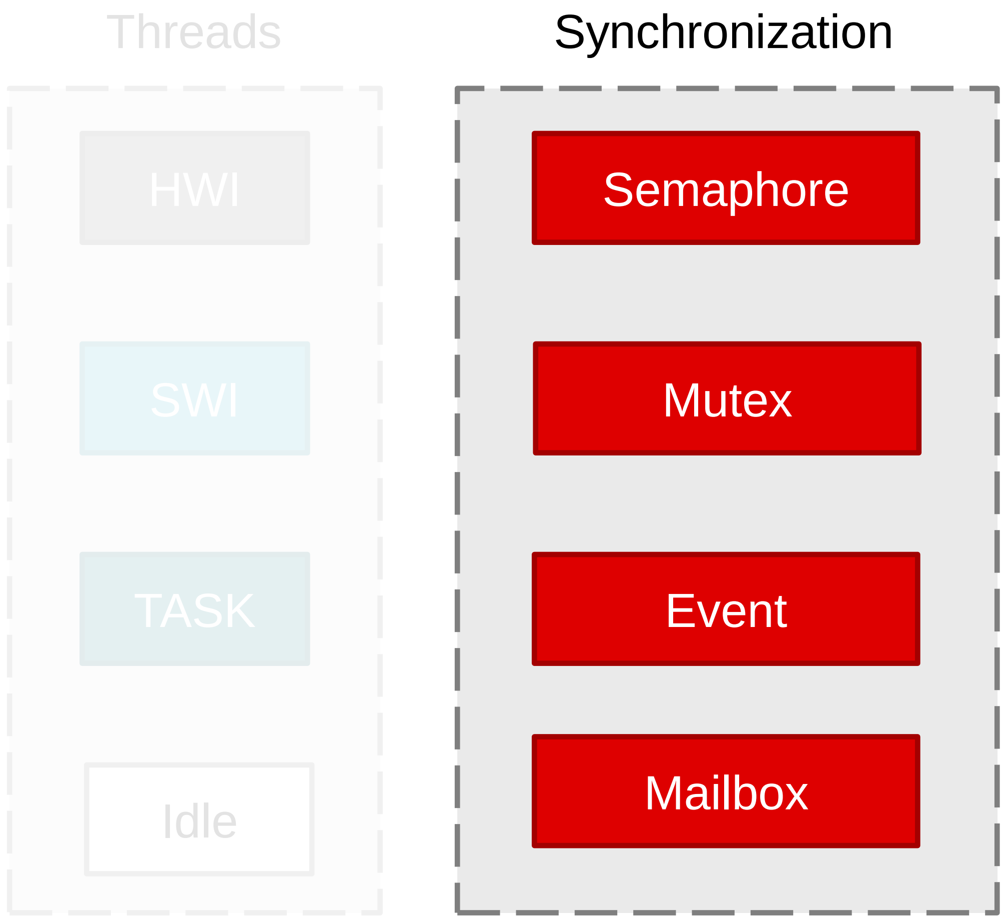
- Semaphore
- Mutex
- Event
- Mailbox

Idle thread

- The Idle thread is only run when no other thread is running
- Sets the device into the **lower possible** power mode

Signaling and resource sharing between SWI to Task and Task to Task

TI RTOS – Kernel Objects Overview



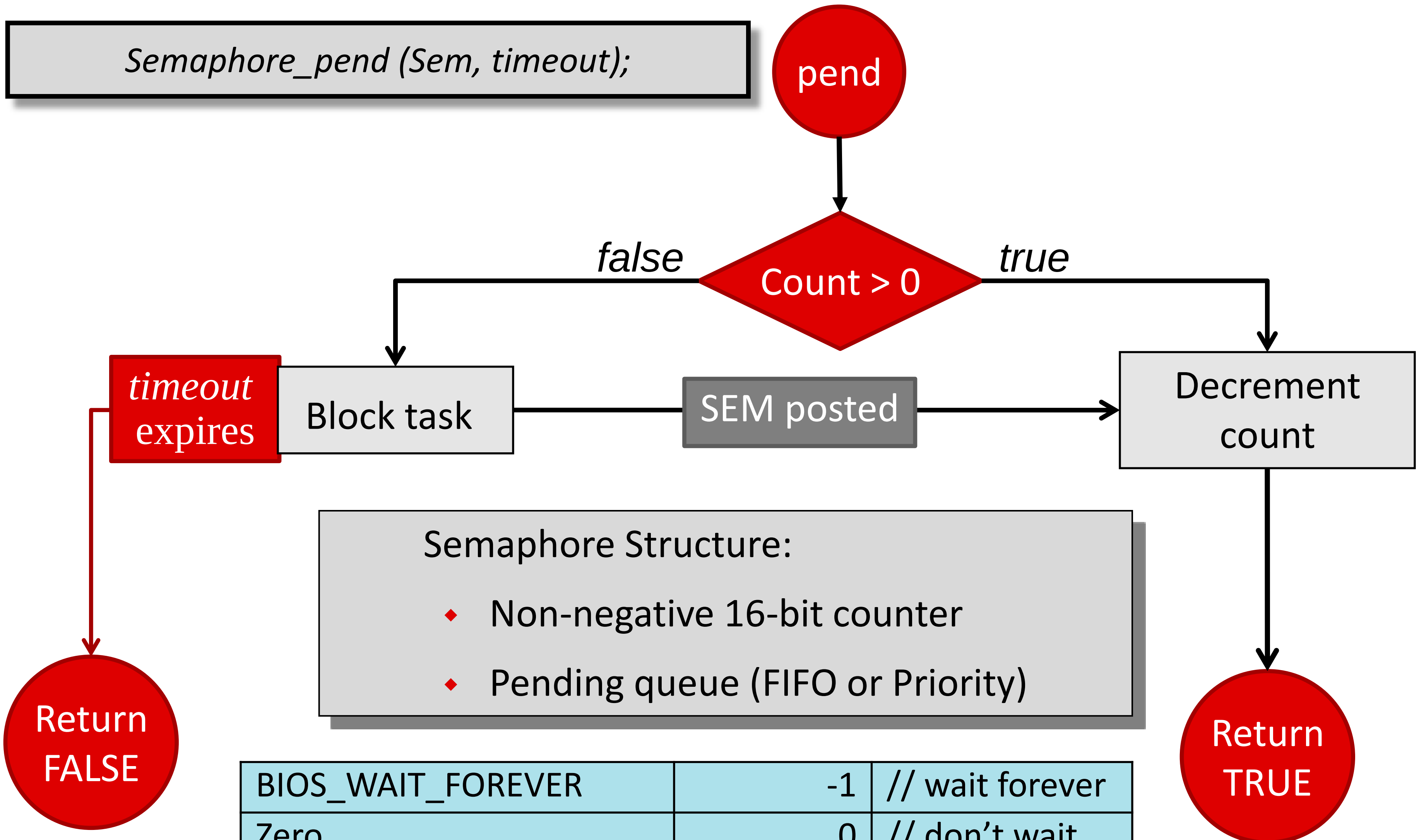
Semaphore

Semaphore is a concept used for **synchronizing** threads. Is implemented by using an internal counter which is incremented and decremented.

- Semaphore_post(Sem_1)
  - Increment semaphore counter
- Semaphore_pend(Sem_1, Timeout)
  - Decrement semaphore counter
  - Potentially blocks if count is zero

Semaphore Pend

Semaphore_pend (Sem, timeout);



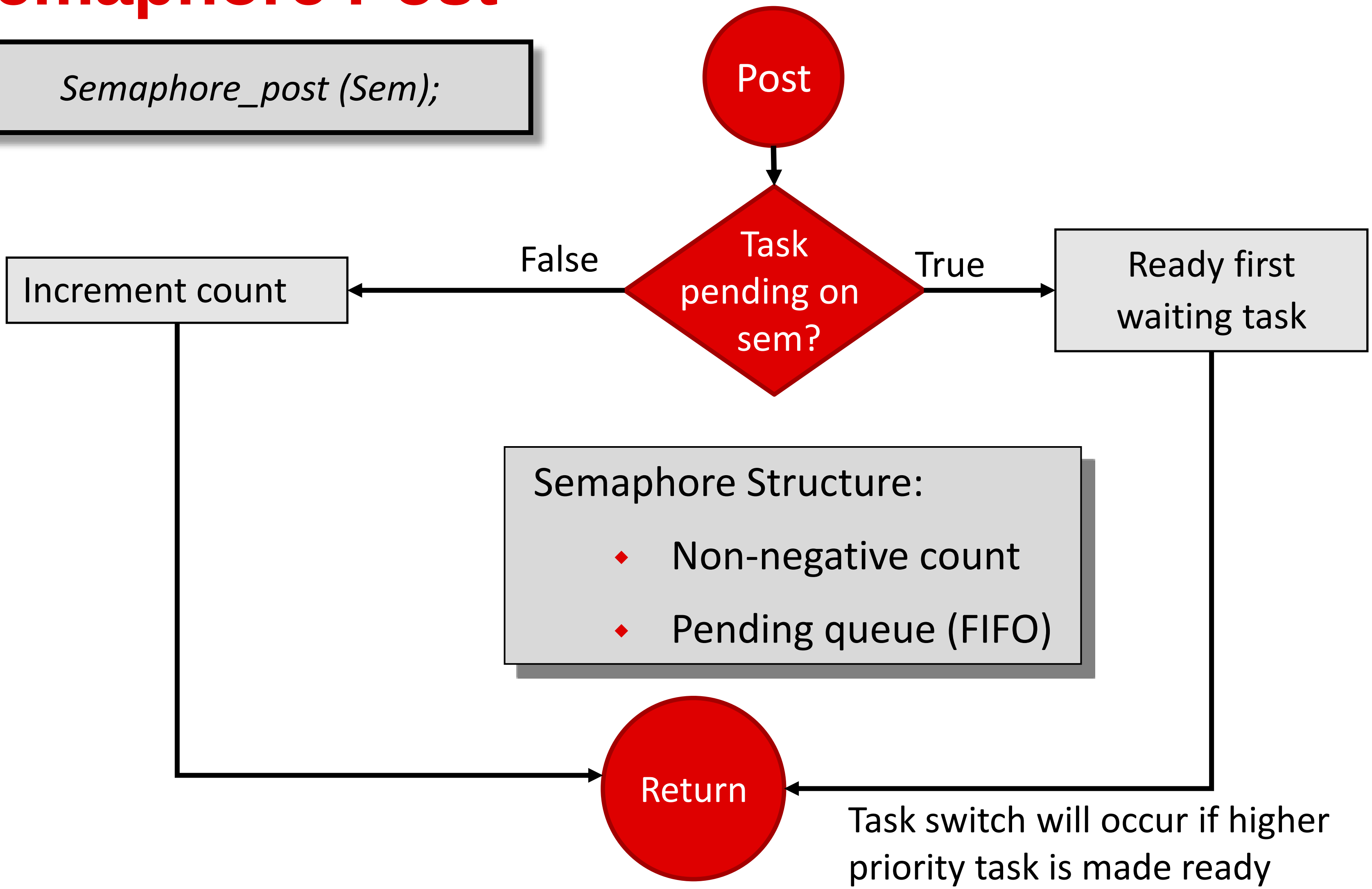
Semaphore Structure:

- Non-negative 16-bit counter
- Pending queue (FIFO or Priority)

BIOS_WAIT_FOREVER	-1	// wait forever
Zero	0	// don't wait
value	timeout	// system ticks

Semaphore Post

Semaphore_post (Sem);



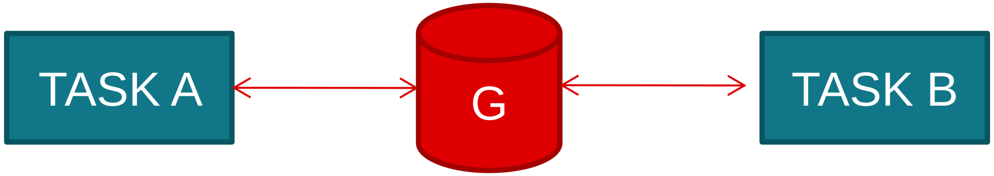
Semaphore

```
void timerSwi() {
  Semaphore_post(Sem_1);
}
```

```
void task() {
  while(1) {
    Semaphore_pend(Sem_1, Timeout);
    /* Do processing */
    /* Blink LED */
  }
}
```

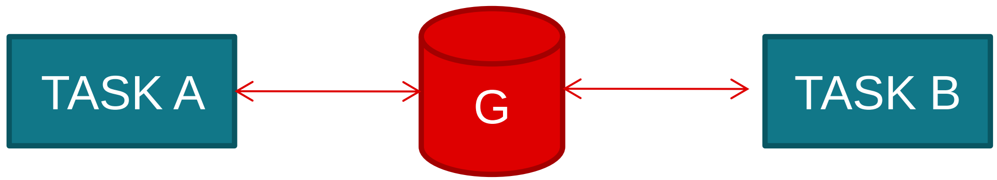
Shared resource - Global variables

We have two Tasks and a shared global resource:



Shared resource - Global variables

We have two Tasks and a shared global resource:



Worst case:

TASK A

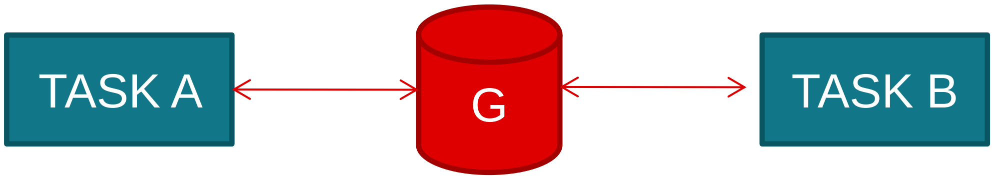
```
void hiPriTask() {
  …
  cnt += 1;
  …
}
```

TASK B

```
void lowPriTask() {
  …
  cnt += 1;
  …
}
```

Shared resource - Global variables

We have two Tasks and a shared global resource:



Worst case:

TASK A

```
void hiPriTask() {
  …
  cnt += 1;
  …
}
```

TASK B

```
void lowPriTask() {
  …
  cnt += 1;
     LD
     ADD
     ST
  …
}
```

Global variables with Critical Section

TASK B

TASK A

```
void Task() {
  …
  pGIE = Hwi_disable();
  cnt += 1;
  Hwi_restore(pGIE);

  …
}
```

Simple, but overkill in most cases. This will block **all** HWI's, including everyone that has nothing to do with the "cnt" resource!

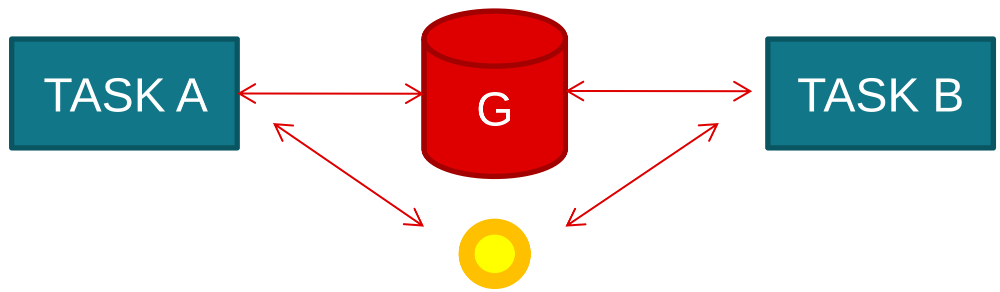
Mutex

A **Mutex** is used for **Mutual exclusion.** It provides a way to guarantee that only one thread at a time has access to a specific resource.

Similar to Semaphore with count = 1 but it has an **owner**

Global variables with Mutex

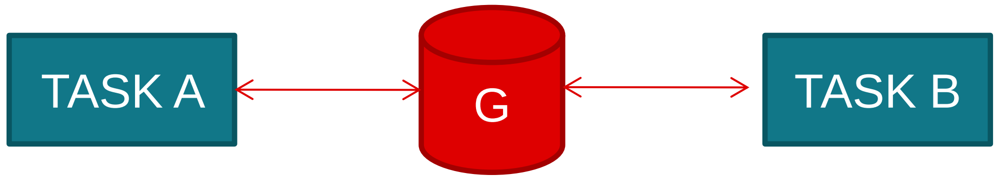
We have two Tasks and a shared global resource:

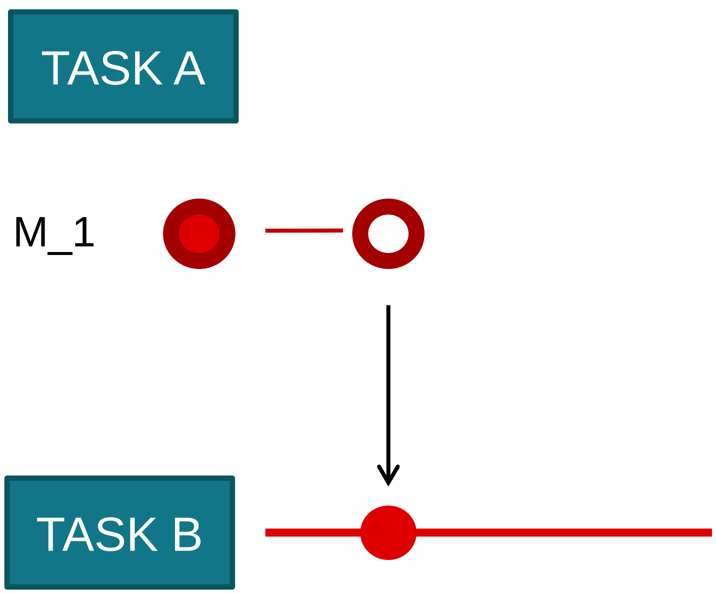


TASK B TASK A

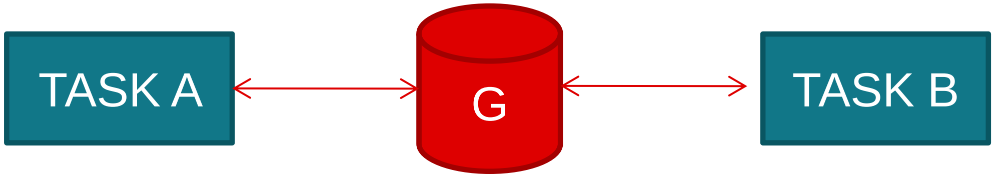
```
void Task() {
   …
   gateKey = GateMutex_enter(cntMutex);
   cnt += 1;
   GateMutex_leave(cntMutex, gateKey);
   …
}
```

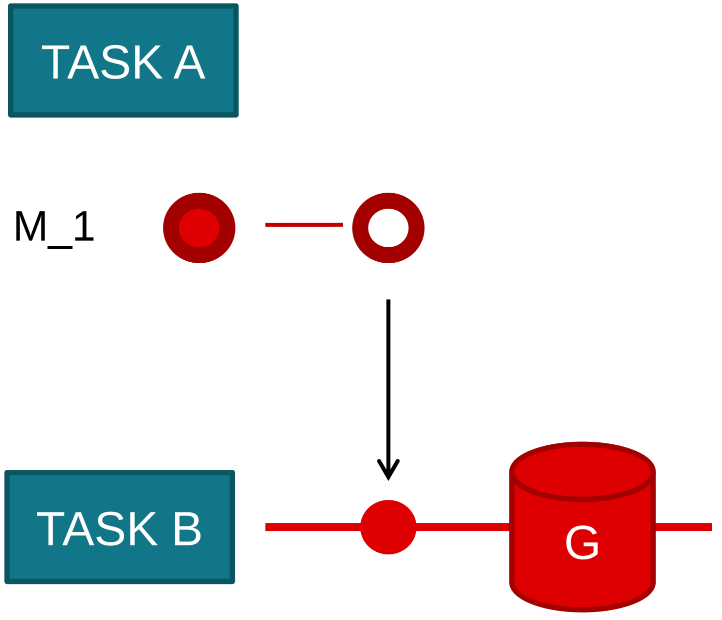
Global variables with Mutex



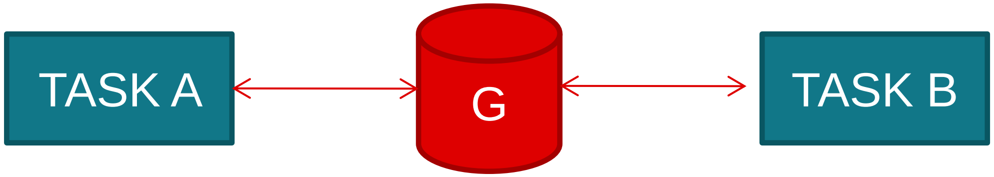


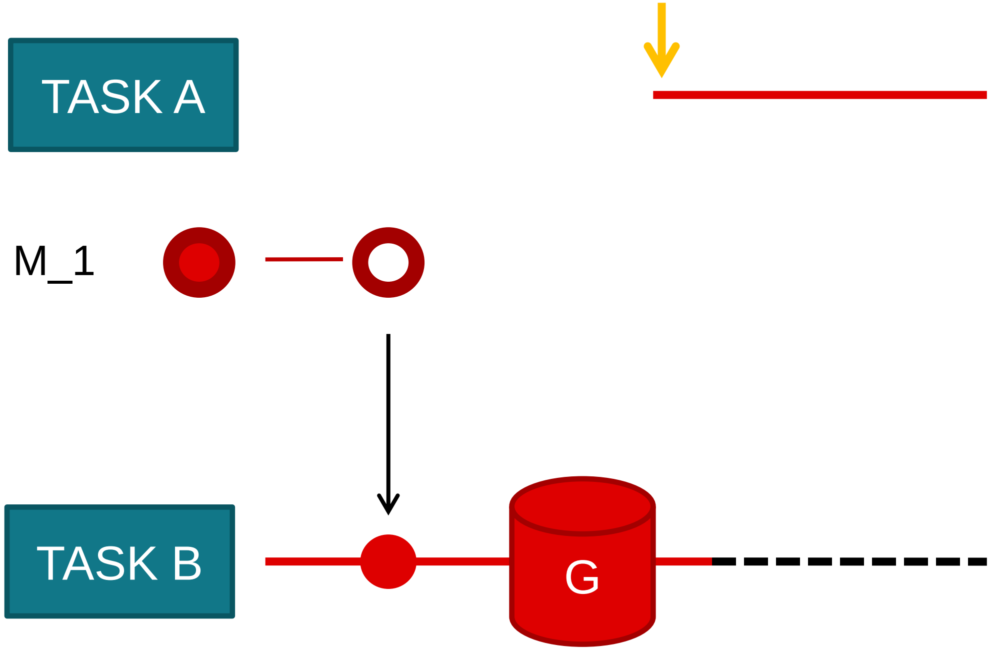
Global variables with Mutex



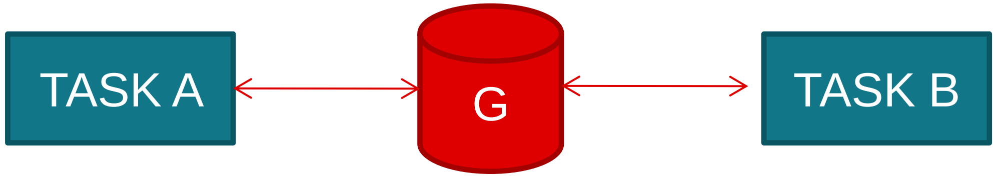


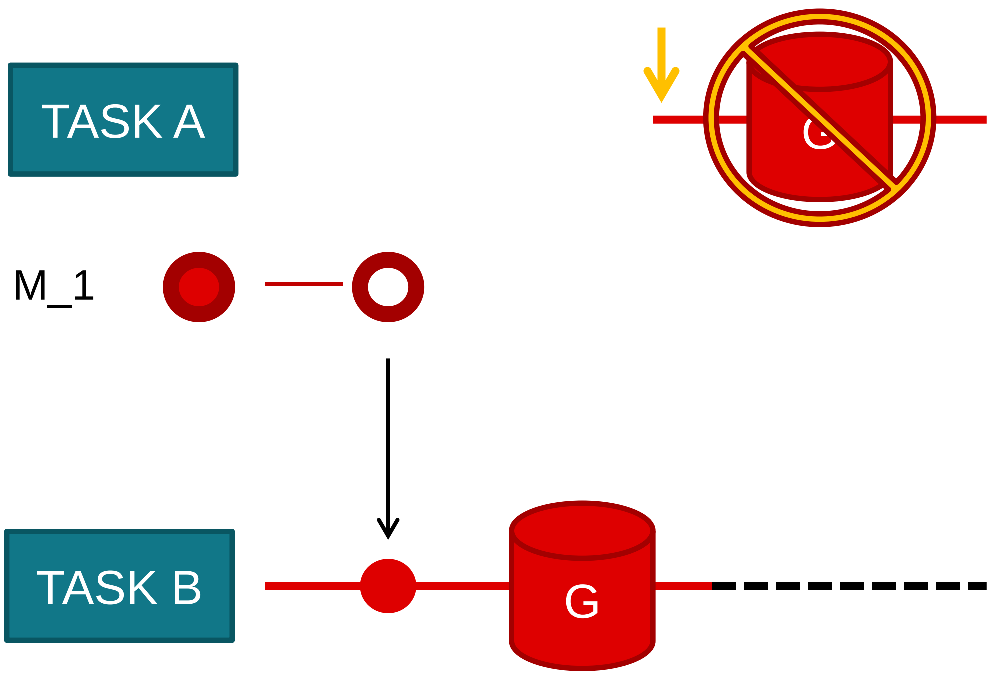
Global variables with Mutex



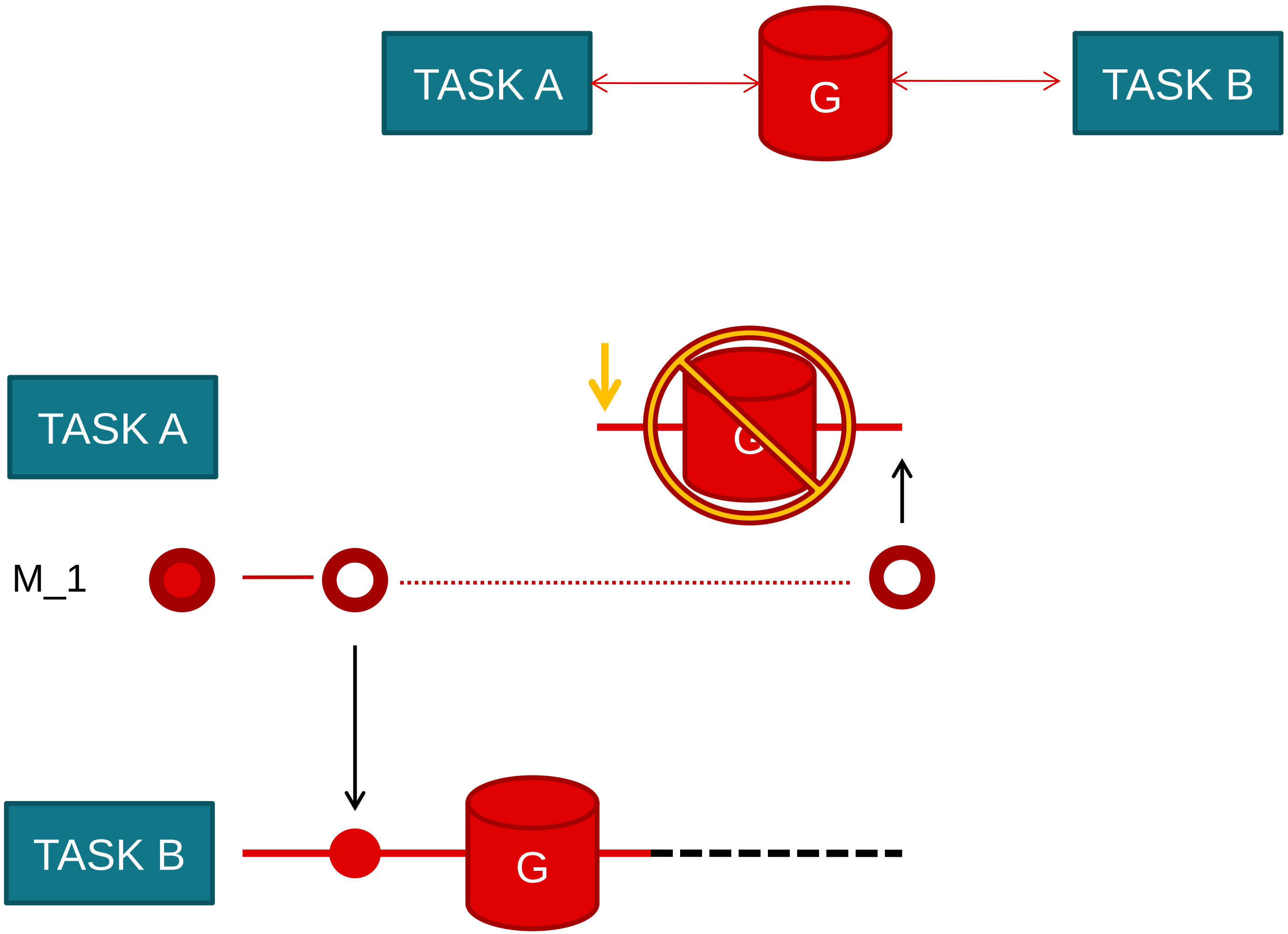


Global variables with Mutex

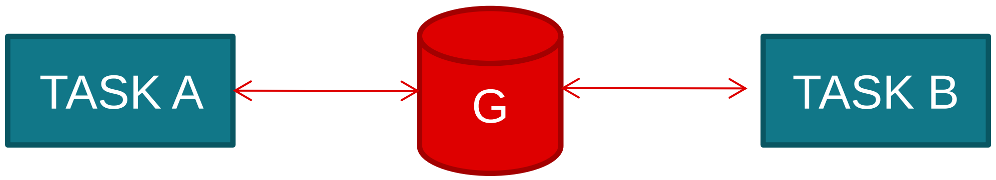


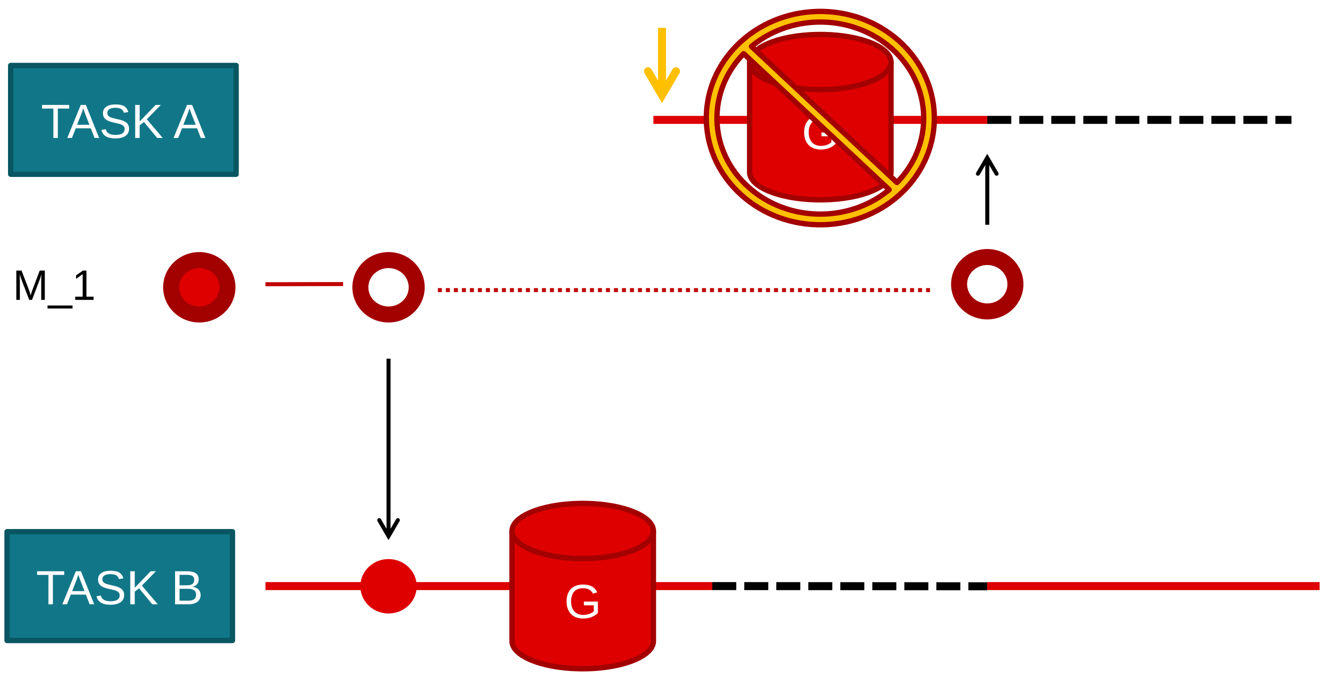


Global variables with Mutex

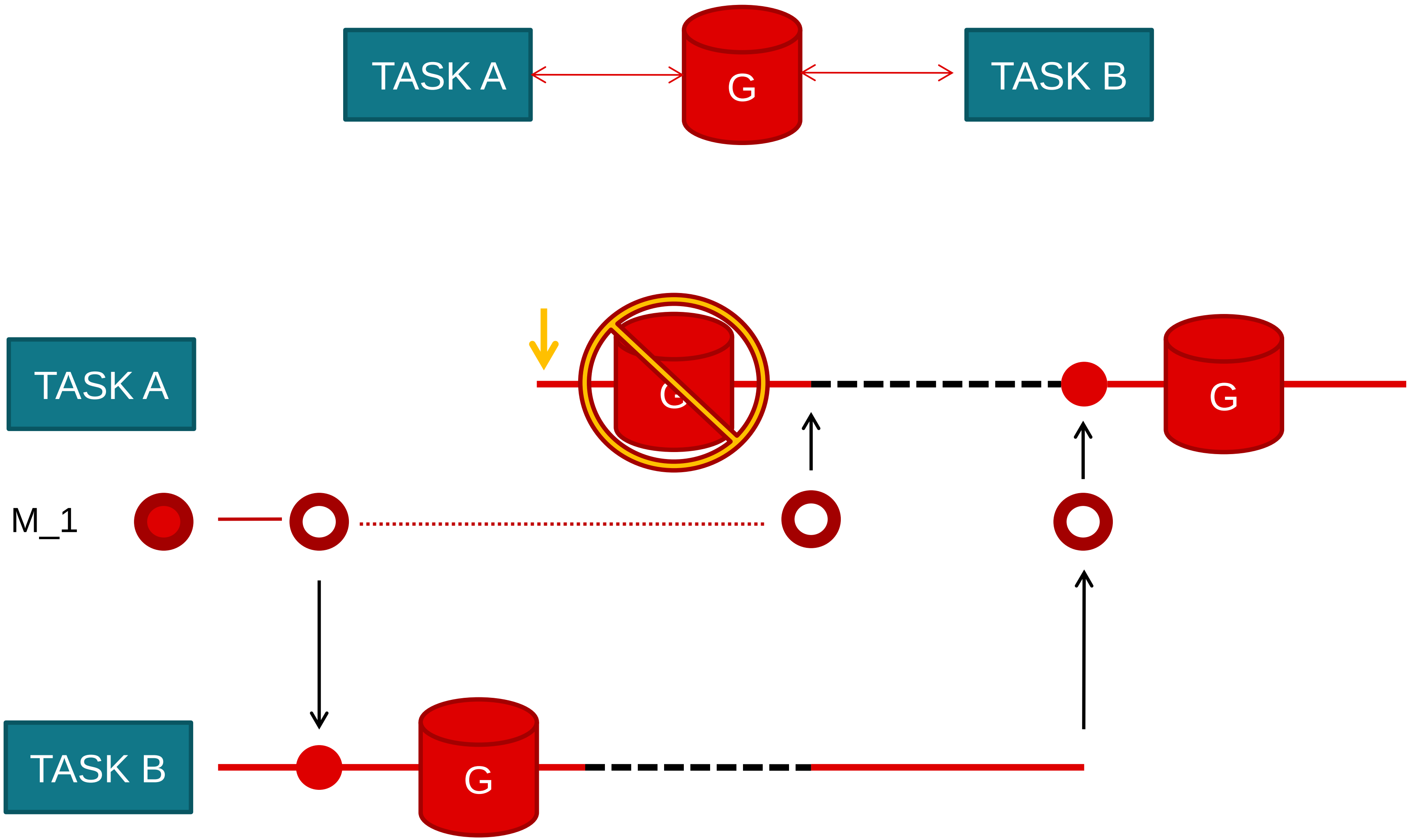


Global variables with Mutex

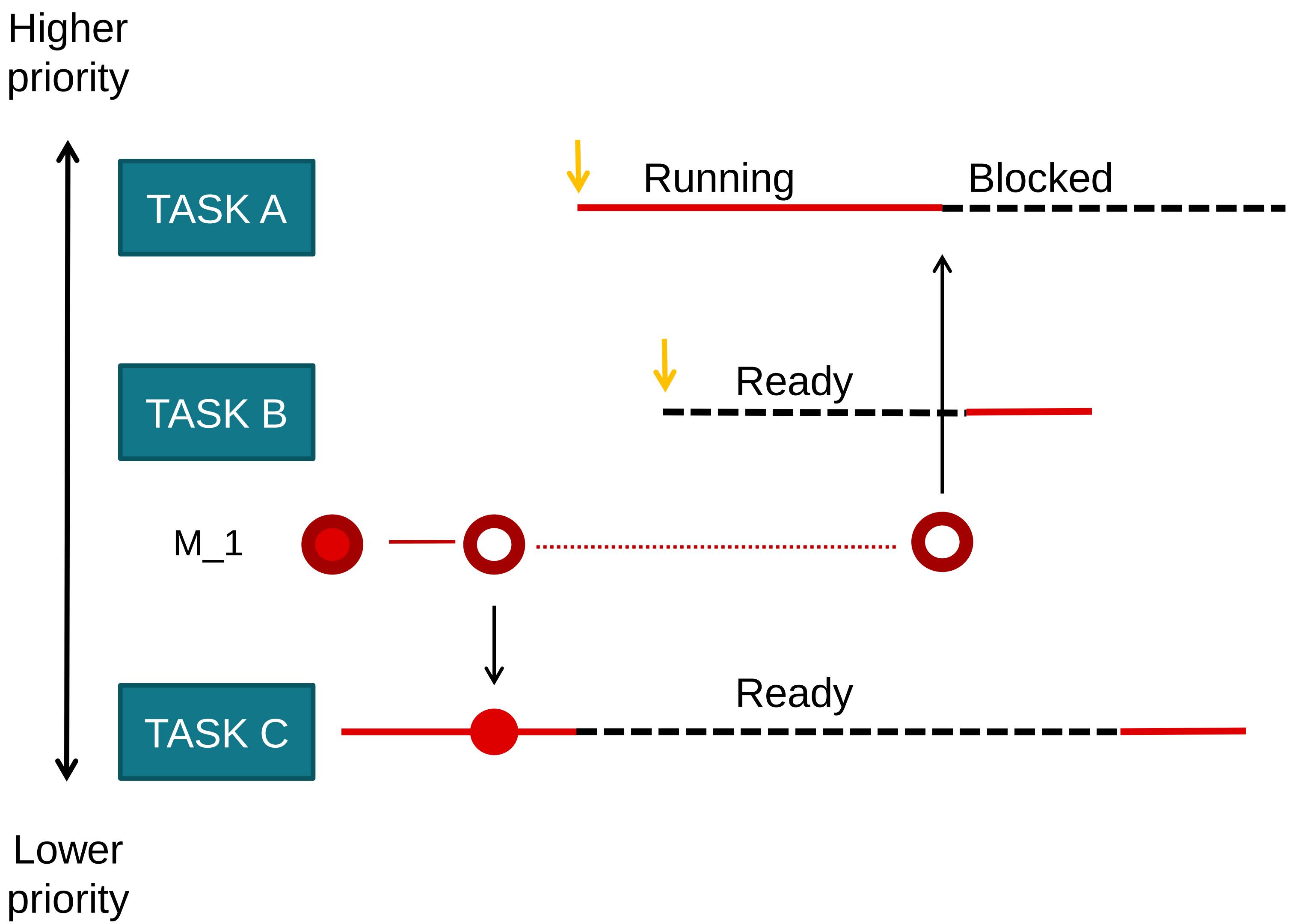




Global variables with Mutex



Task states

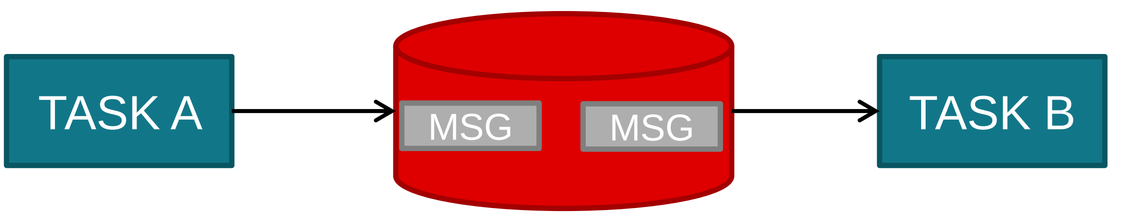

Higher
priority
TASK A
Running
Blocked
TASK B
Ready
M_1
TASK C
Ready
Lower
priority

Mailbox

- Publish – Subscribe
- Semaphore + FIFO
- Fixed size
  - Size of message
  - Number of messages

```
Mailbox_post(&msgBox, &msg, timeout)

Mailbox_pend(&msgBox, &msg, timeout)
```



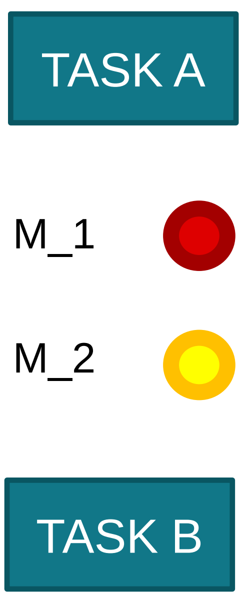
RTOS Challenges

Common challenges

Deadlock

When two threads both needs the resource the other thread has to continue their processing

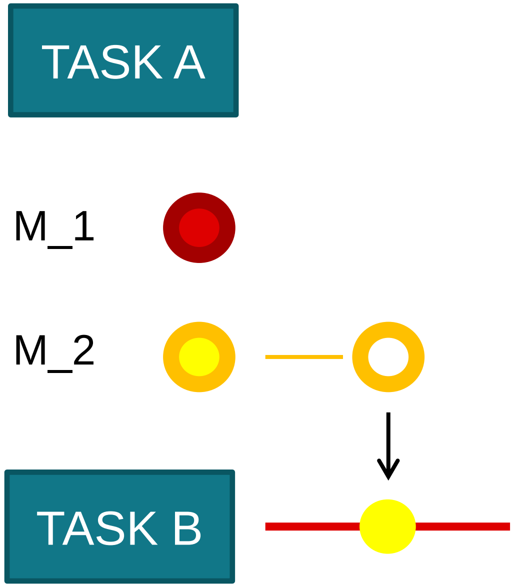
- At least two Mutexes
- Two threads of different priorities



Deadlock

When two threads both needs the resource the other thread has to continue their processing

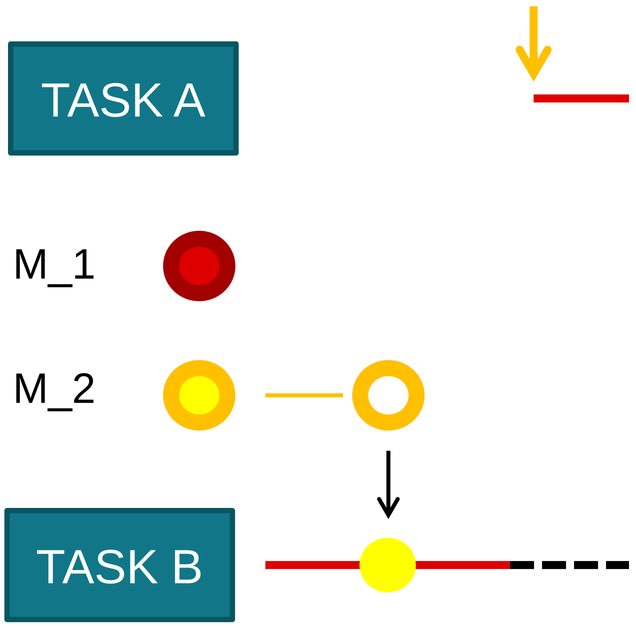
- At least two Mutexes
- Two threads of different priorities



Deadlock

When two threads both needs the resource the other thread has to continue their processing

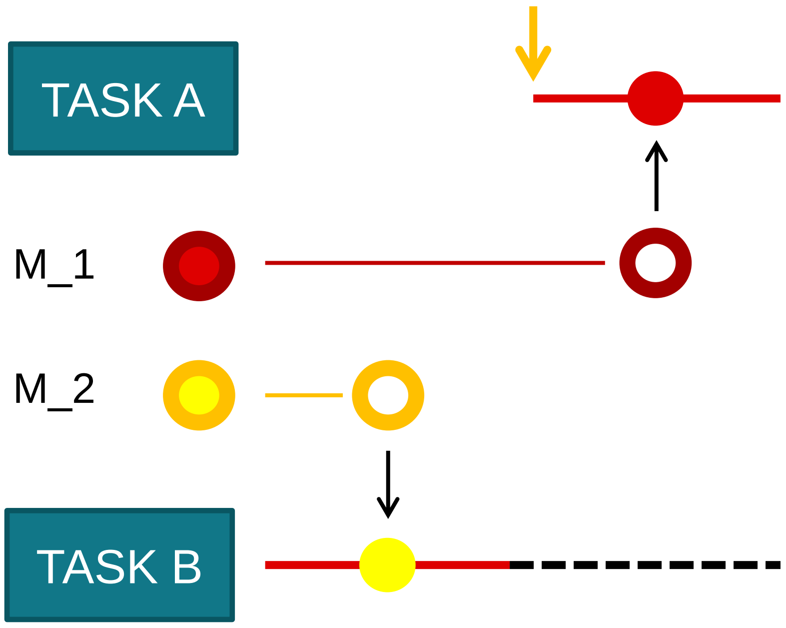
- At least two Mutexes
- Two threads of different priorities



Deadlock

When two threads both needs the resource the other thread has to continue their processing

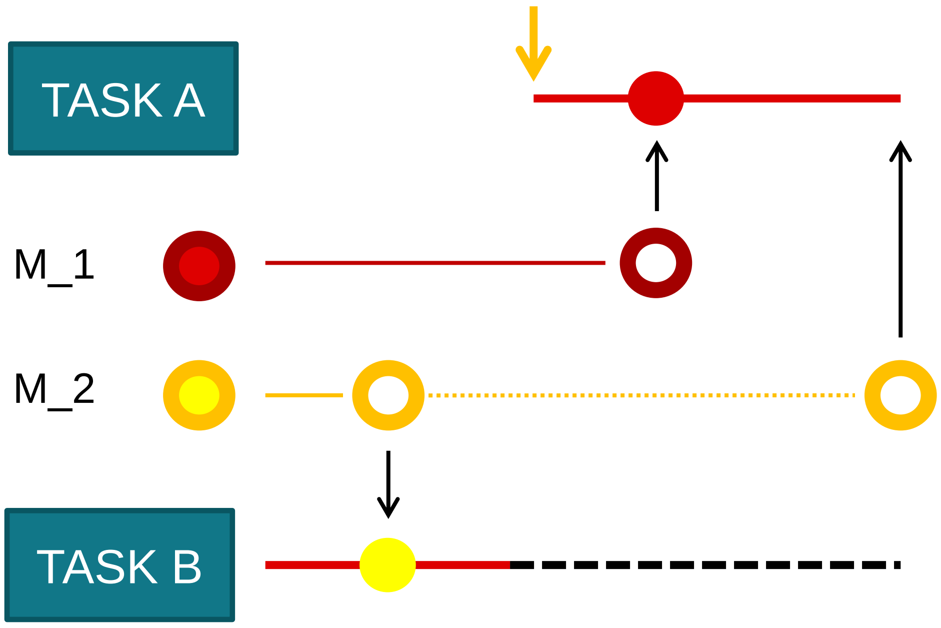
- At least two Mutexes
- Two threads of different priorities



Deadlock

When two threads both needs the resource the other thread has to continue their processing

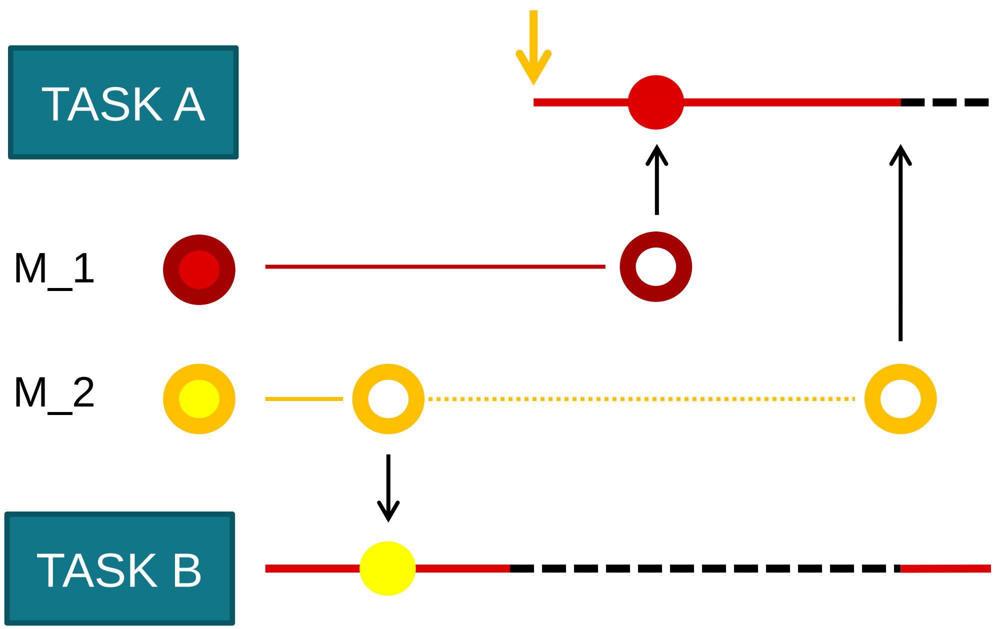
- At least two Mutexes
- Two threads of different priorities



Deadlock

When two threads both needs the resource the other thread has to continue their processing

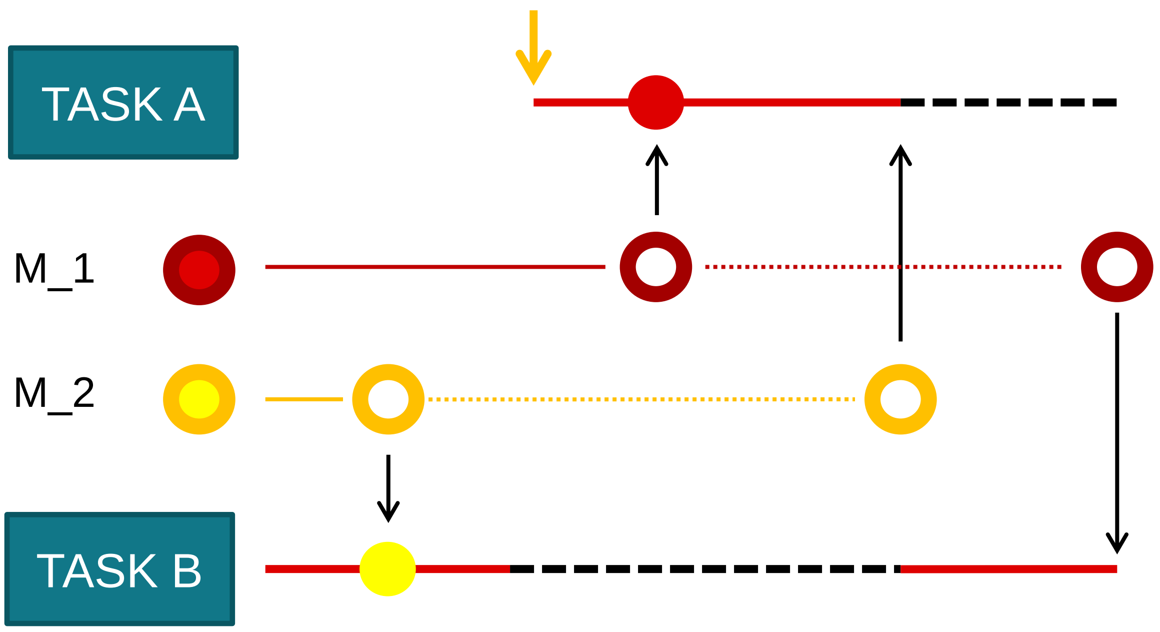
- At least two Mutexes
- Two threads of different priorities



Deadlock

When two threads both needs the resource the other thread has to continue their processing

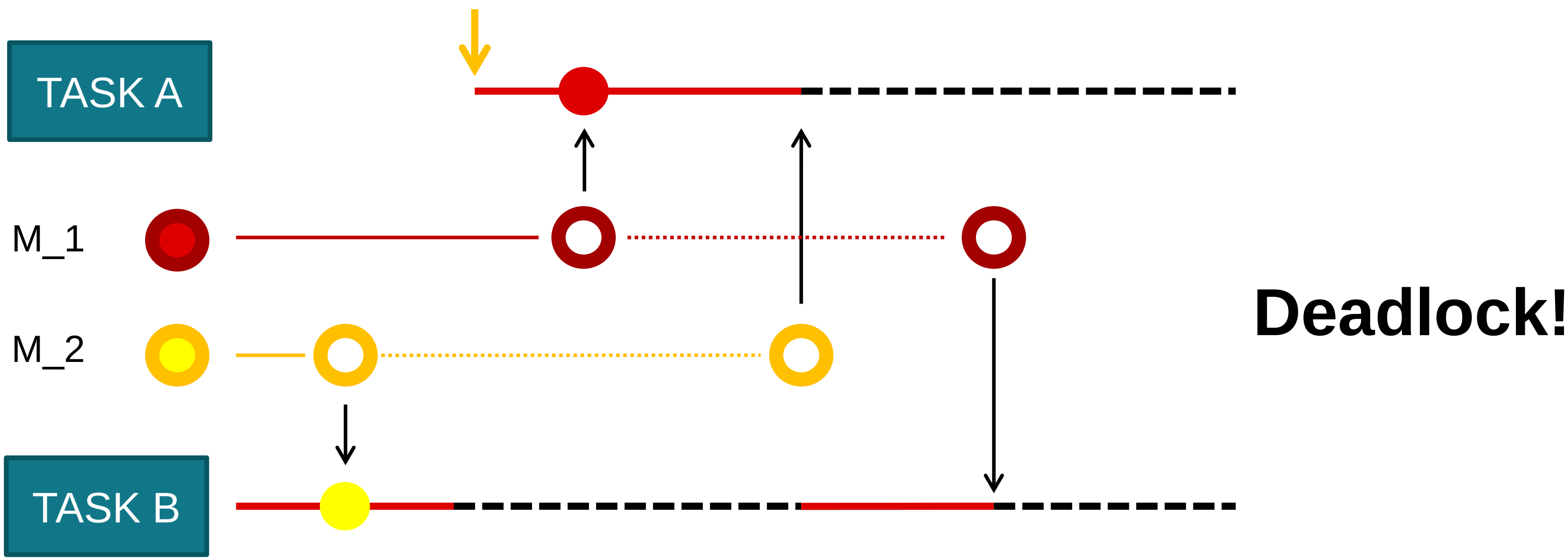
- At least two Mutexes
- Two threads of different priorities



Deadlock

When two threads both needs the resource the other thread has to continue their processing

- At least two Mutexes
- Two threads of different priorities



Deadlock

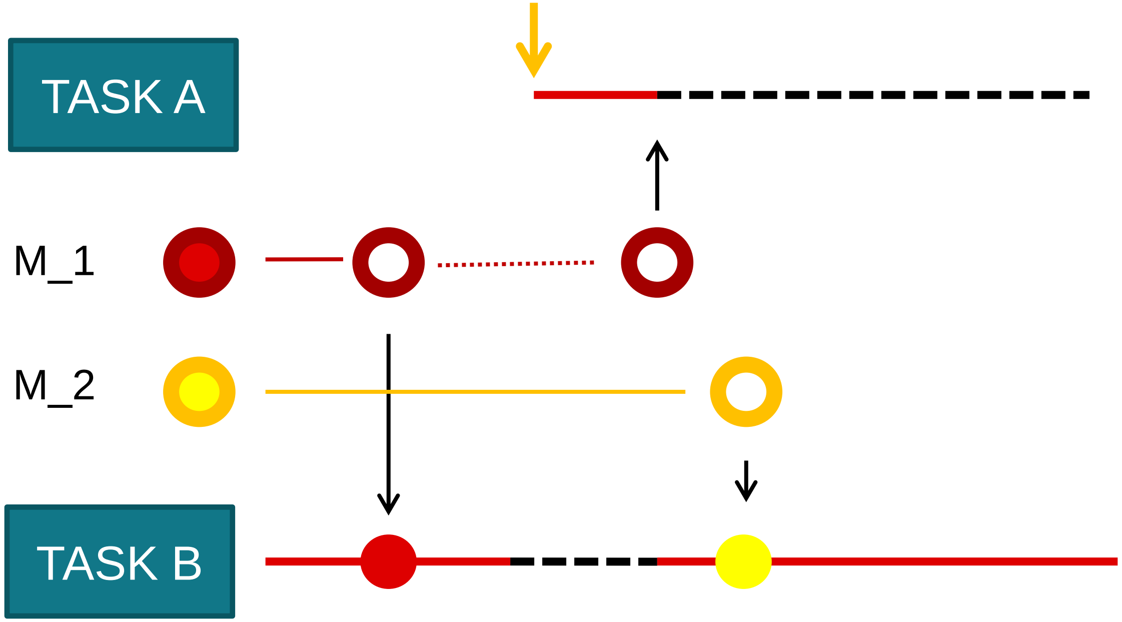
Ways to avoid deadlock:

- Use one big Mutex to cover both resources
- Always pend on the Mutex in the same order in all threads

Deadlock

Ways to avoid deadlock:

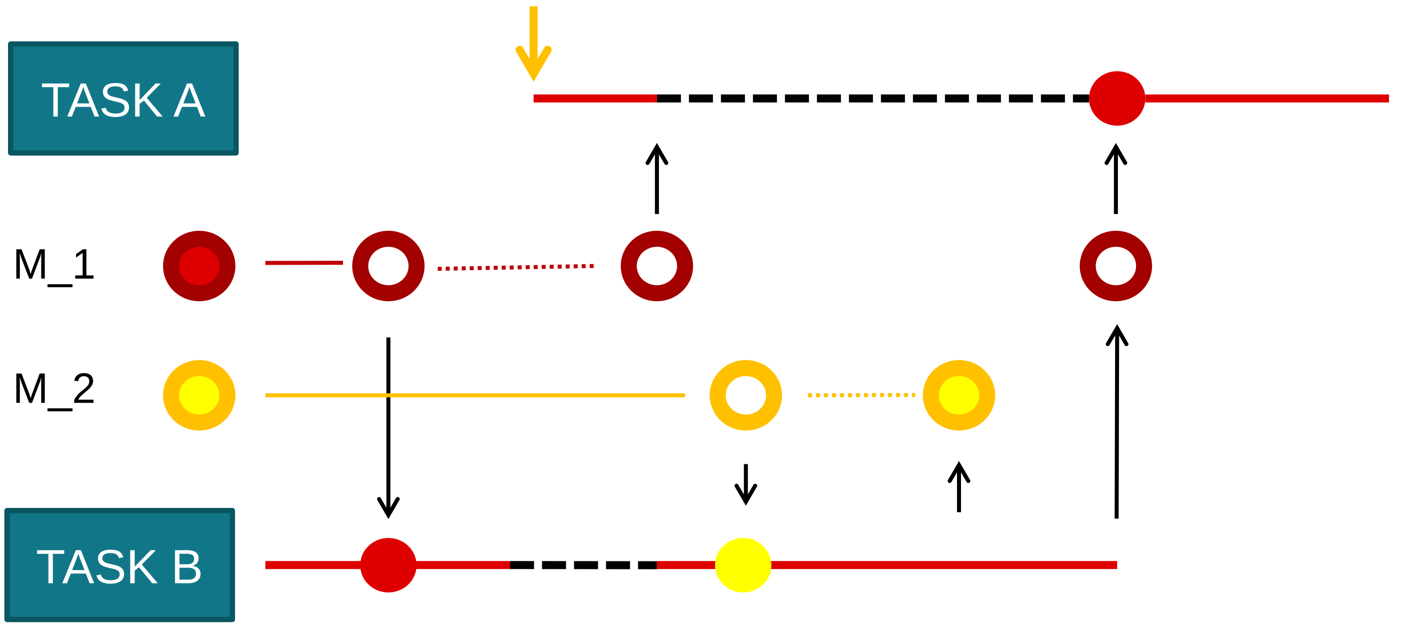
- Use one big Mutex to cover both resources
- Always pend on the Mutex in the same order in all threads



Deadlock

Ways to avoid deadlock:

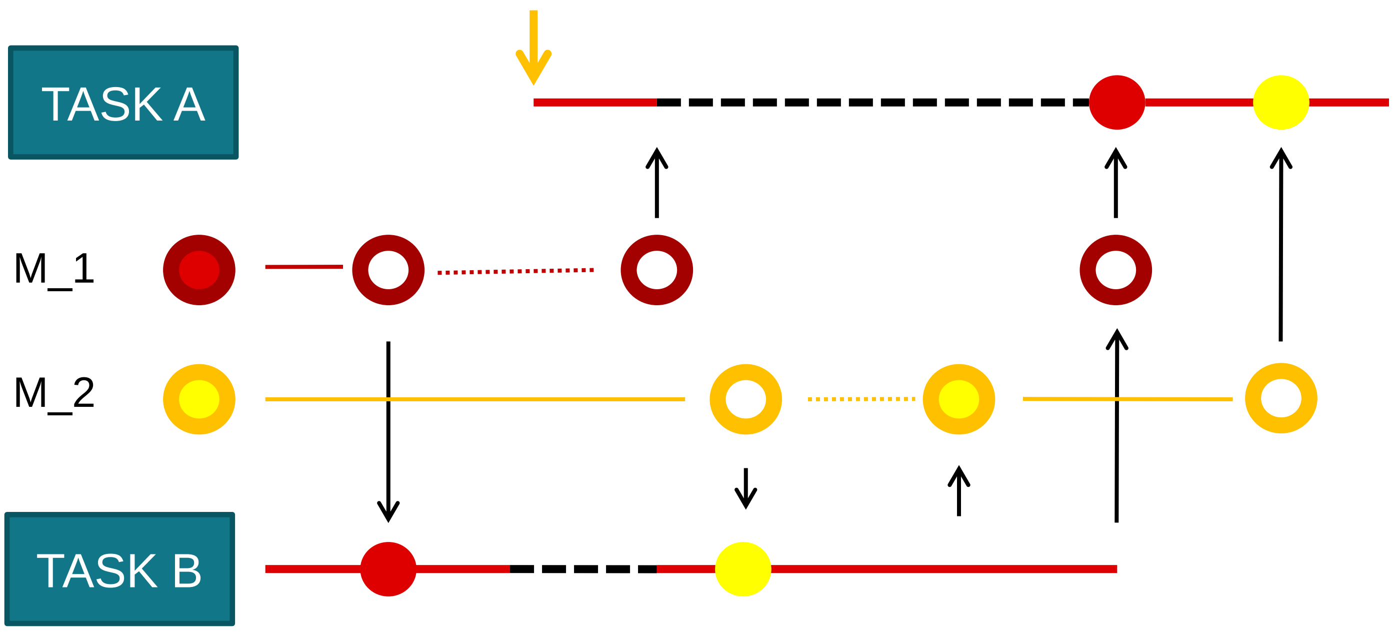
- Use one big Mutex to cover both resources
- Always pend on the Mutex in the same order in all threads



Deadlock

Ways to avoid deadlock:

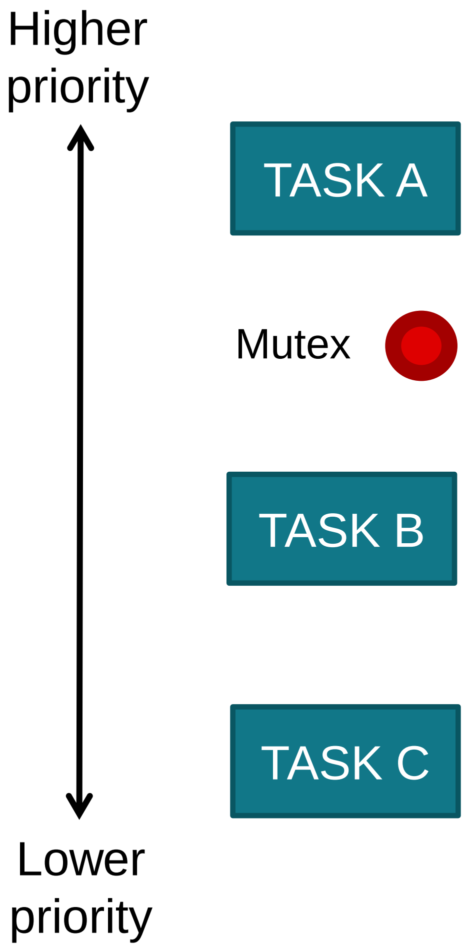
- Use one big Mutex to cover both resources
- Always pend on the Mutex in the same order in all threads



No deadlock

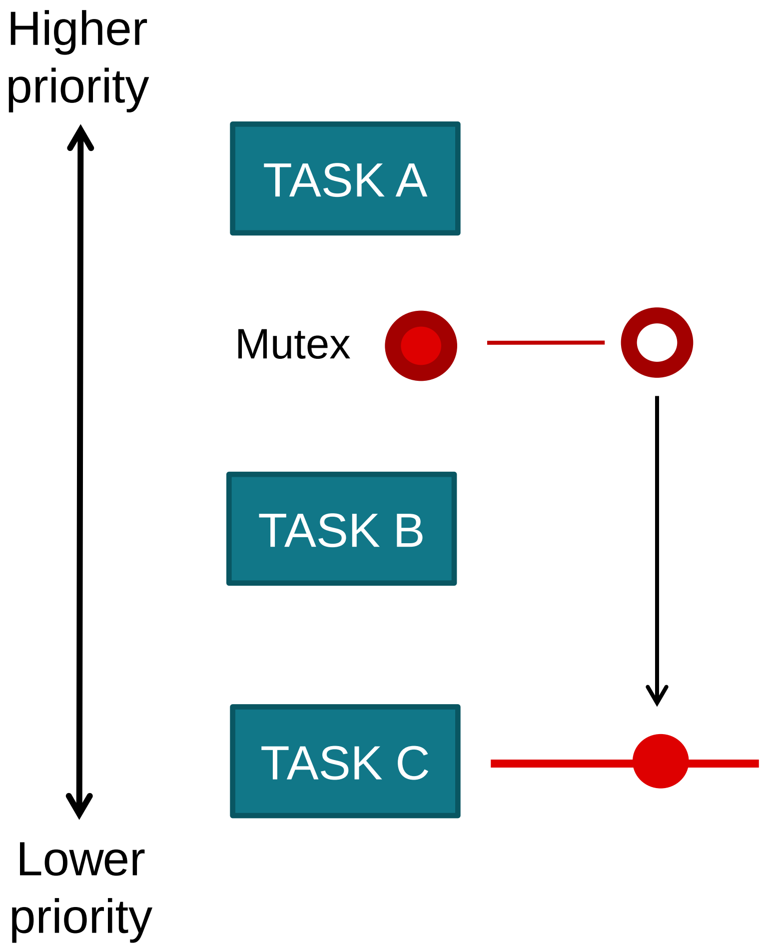
Priority Inversion

When a lower priority thread blocks a higher priority thread



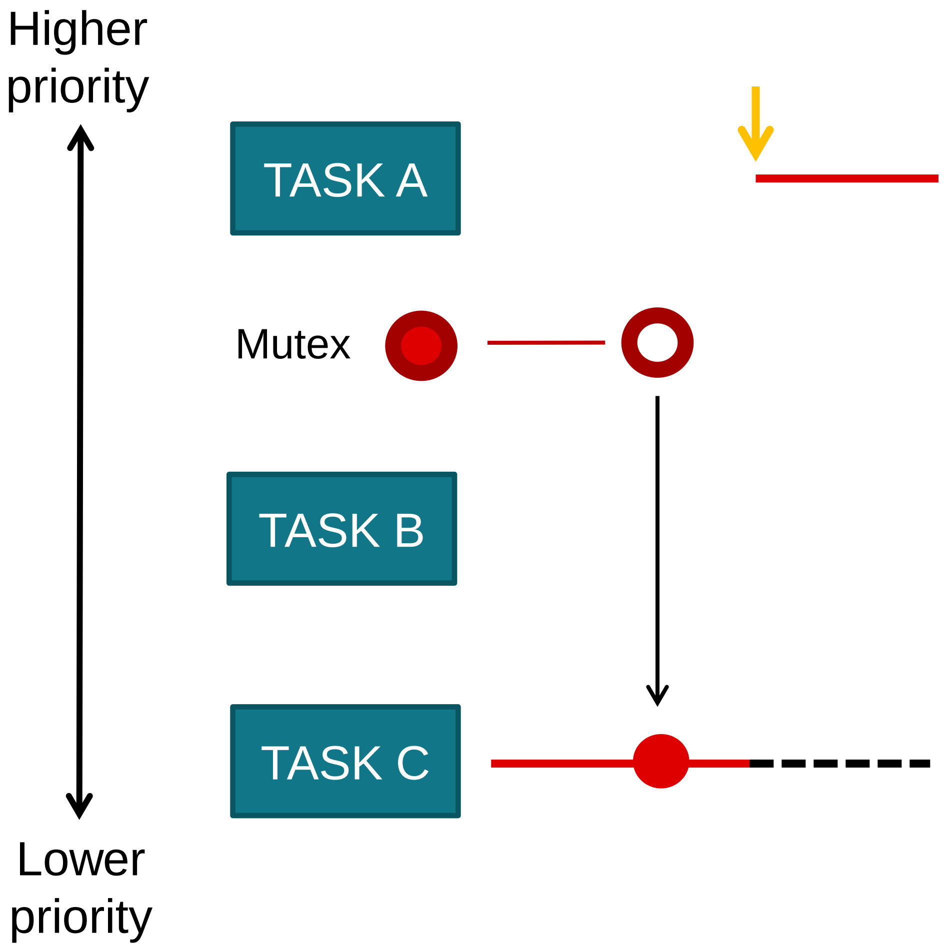
Priority Inversion

When a lower priority thread blocks a higher priority thread



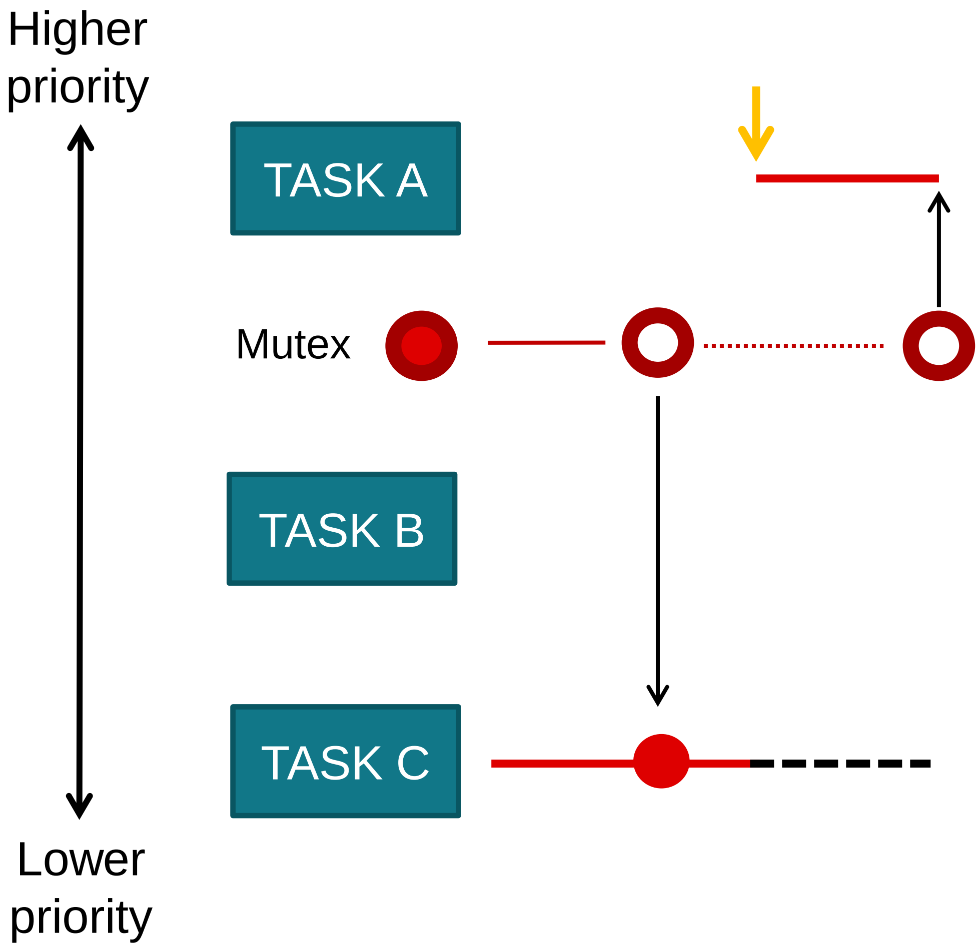
Priority Inversion

When a lower priority thread blocks a higher priority thread



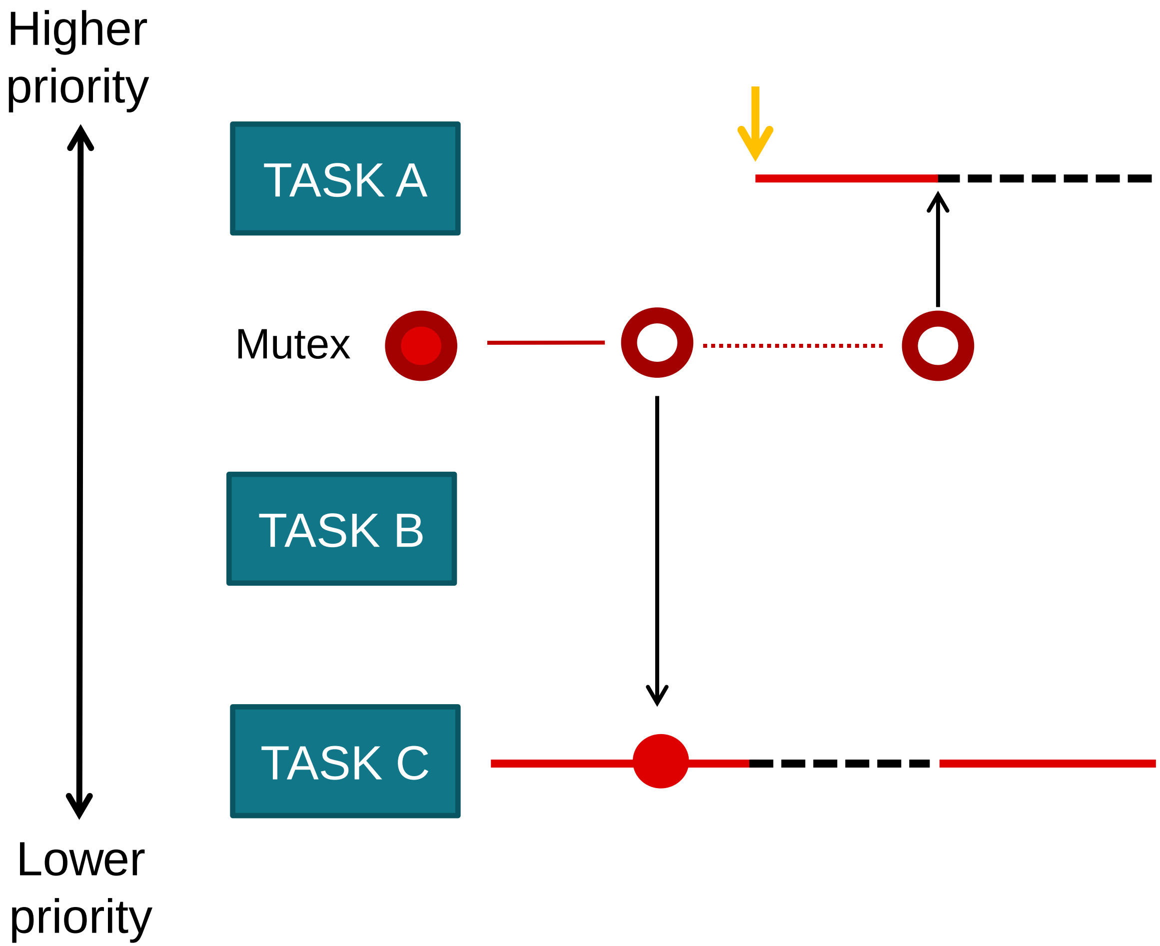
Priority Inversion

When a lower priority thread blocks a higher priority thread



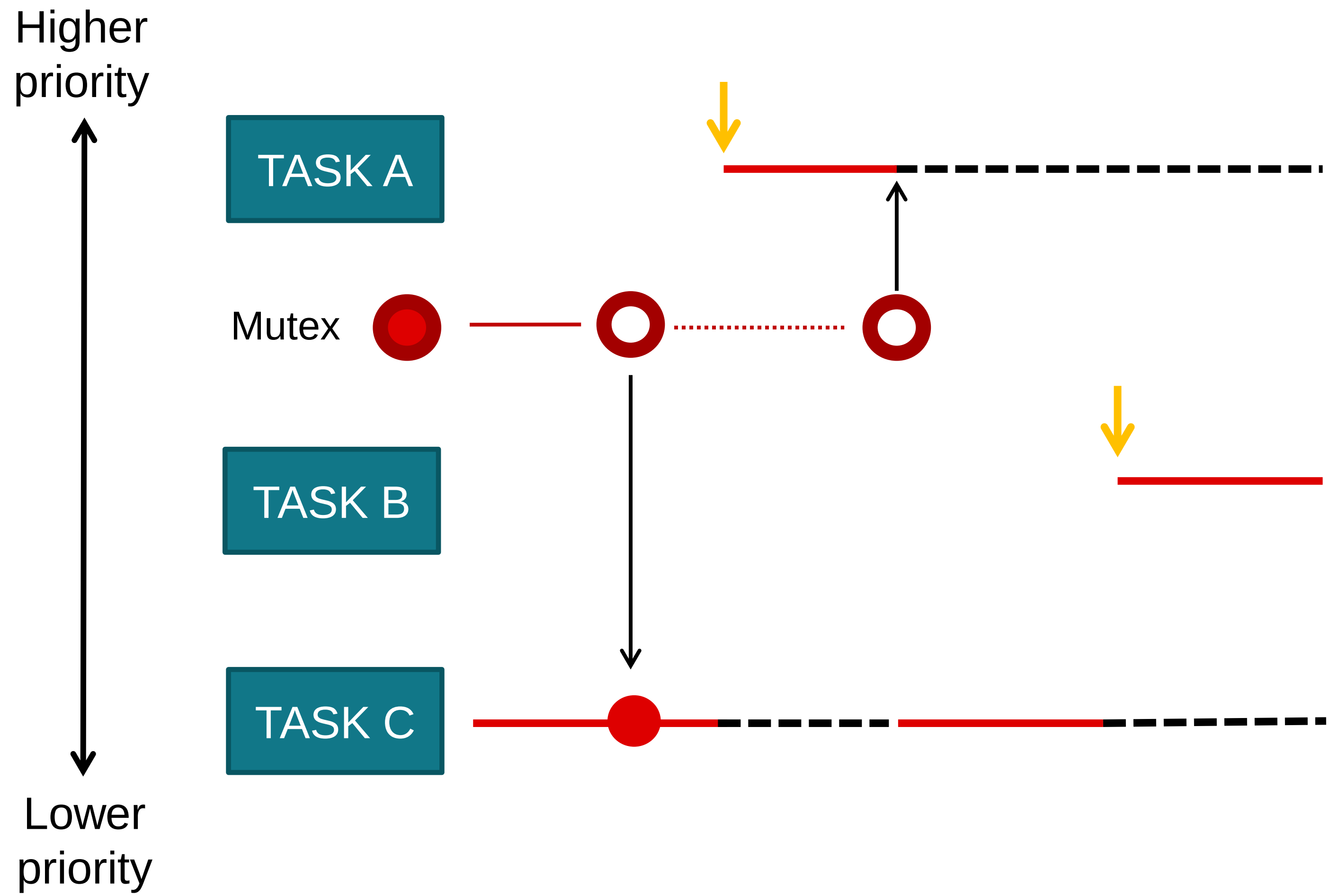
Priority Inversion

When a lower priority thread blocks a higher priority thread



Priority Inversion

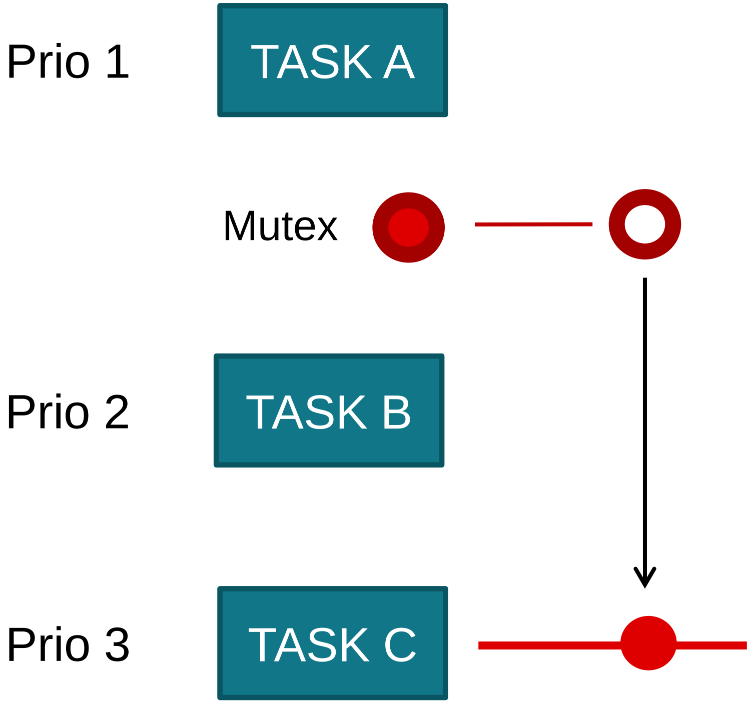
When a lower priority thread blocks a higher priority thread



Priority inversion in effect!

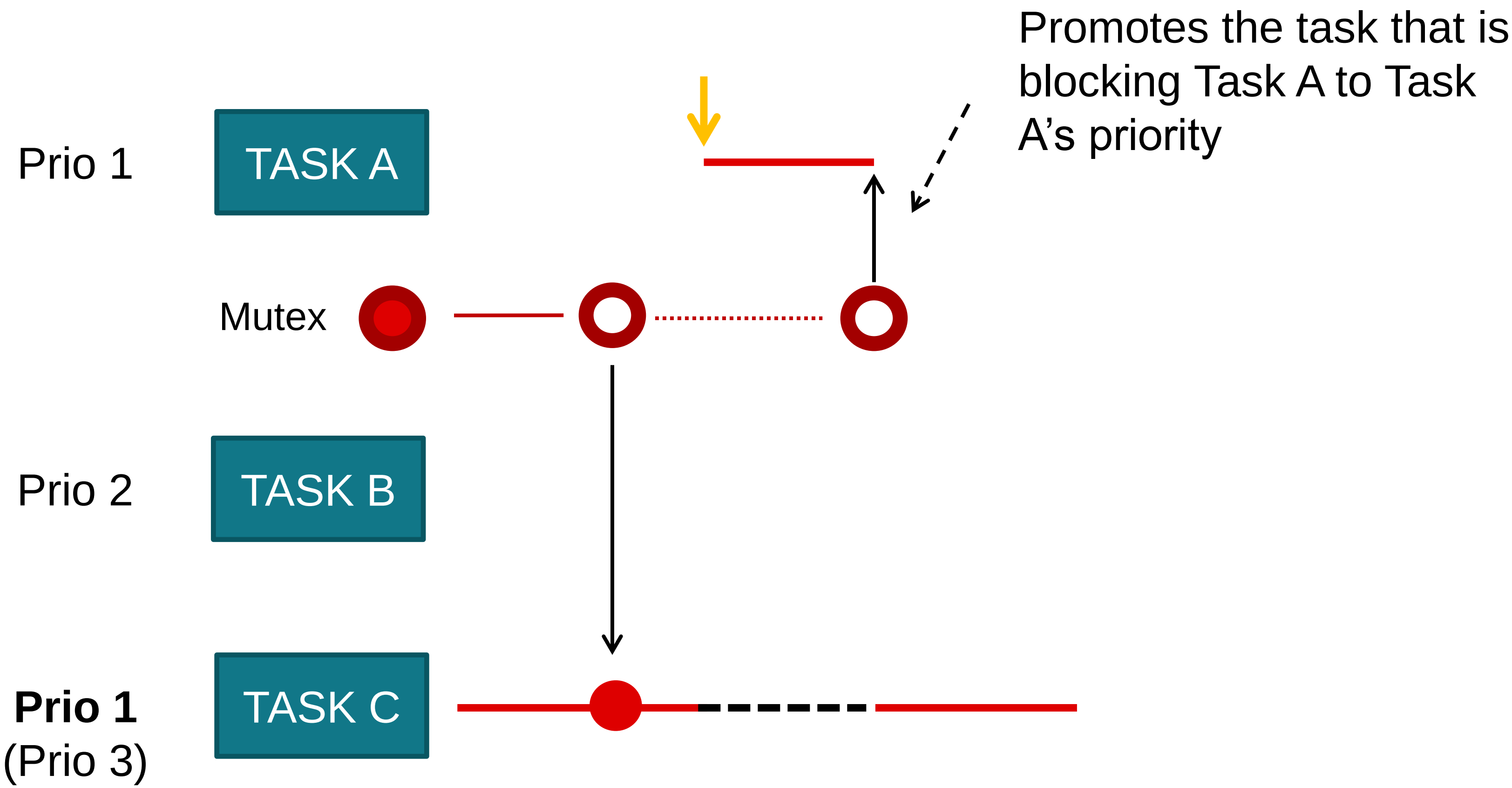
Priority Inversion

How to avoid it: Temporarily promote Task C to Task A's priority



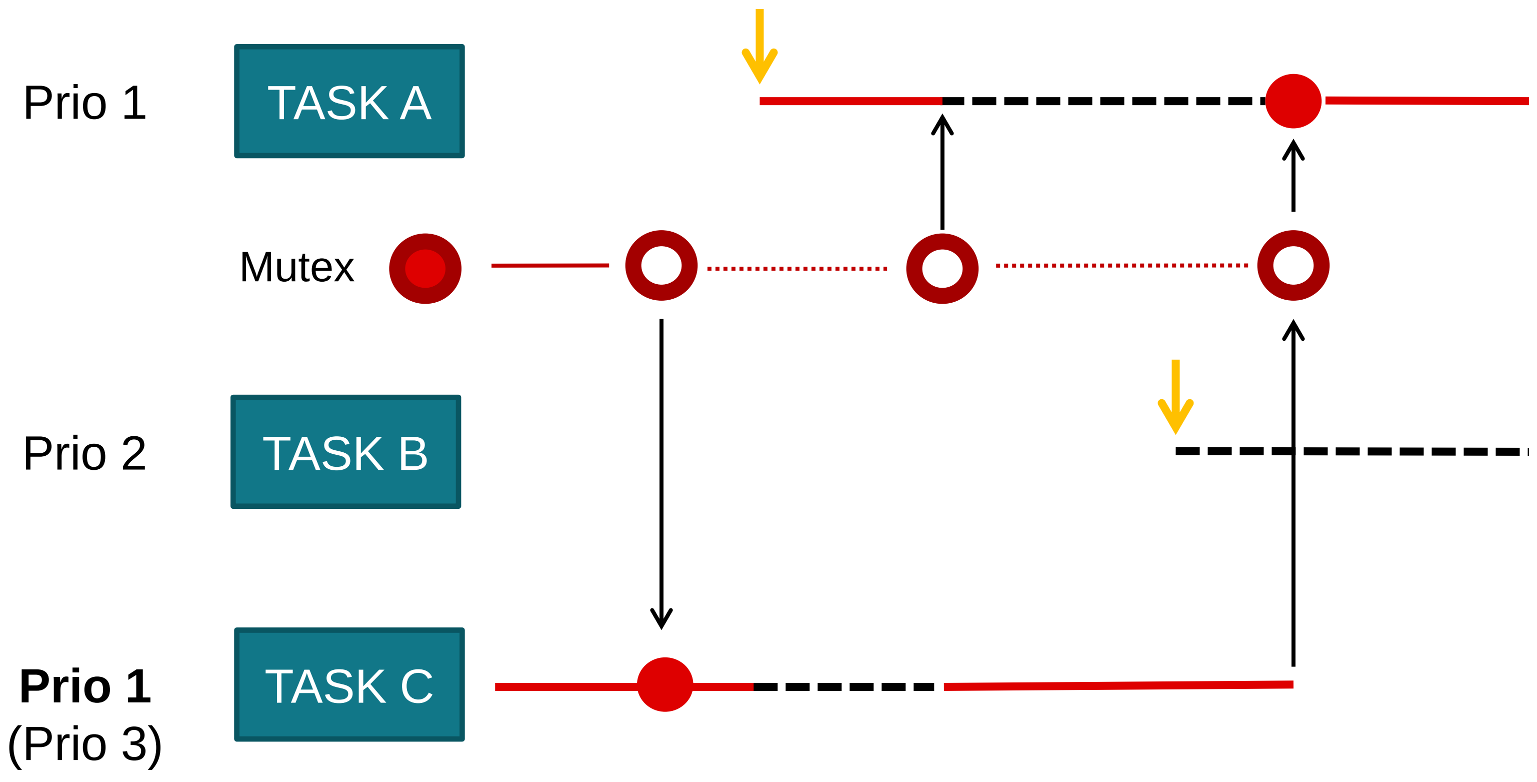
Priority Inversion

How to avoid it: Temporarily promote Task C to Task A's priority



Priority Inversion

How to avoid it: Temporarily promote Task C to Task A's priority



Priority Inversion

How to avoid it: Temporarily promote Task C to Task A's priority