

SYSTÈMES EMBARQUÉS TEMPS RÉEL

Valentin Gies

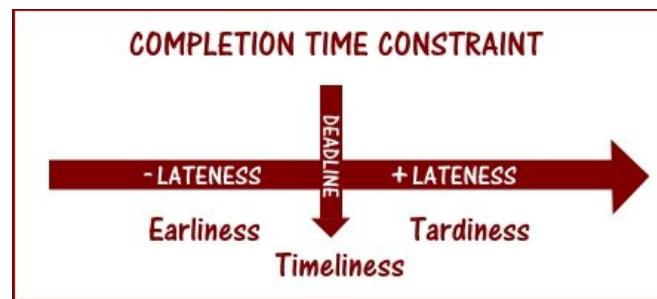
M2 ROC
Université de Toulon

Système temps réel : objectifs

- Déterminisme logique :
 - Des données identiques en entrée produisent des résultats identiques (classique dans les systèmes)
- Déterminisme temporel :
 - Des contraintes temporelles doivent être respectées.
 - Les contraintes temporelles dépendent beaucoup des constantes de temps du système.
- Fiabilité :
 - La disponibilité du système doit être assurée.

Système temps réel : catégories

- Un système temps réel doit répondre dans une contrainte de temps imparti.



- Attention, la contrainte de temps n'est pas seulement la deadline.
- Les effets d'un retard ou d'une avance de phase par rapport à la deadline sont variables selon la tâche à effectuer.
 - Ex : déposer ses enfants à l'école...
- Le respect de cette contrainte peut être strict, important ou annexe.

Système temps réel : catégories

- Temps réel « dur » (Hard Real Time)
 - Les deadlines temporelles ne doivent pas être ratées.
 - Ex : airbag, pacemaker, contrôle d'un réacteur nucléaire
- Temps réel « ferme » (Firm Real Time):
 - Les deadlines temporelles peuvent être occasionnellement ratées :
 - Pas de remise en cause de tout le système
 - Les output sont perdues dans le cas de deadlines ratées.
 - Ex : tâche d'un robot sur une chaîne de fabrication de pièces
 - En cas de raté, la pièce continue son circuit, mais sera rejetée.
- Temps réel « mou » (Soft Real Time):
 - Les deadlines peuvent être ratées régulièrement sans remettre en cause le système, les données en retard restent exploitables.
 - Exemple : données météo en provenance d'une station.

Système temps réel : contraintes

- Dans un même système temps réel, des tâches peuvent être de type différent :
 - Arrêt d'Urgence ou synchronisation : Hard
 - Données à débit élevé et nécessitant un ordonnancement temporel : Firm
 - Exemple : données d'un accéléromètre avec post-processing : la validité des algos repose sur le fait que l'échantillonnage soit régulier.
 - Données peu changeantes ou peu importantes : Soft
 - Exemple : donnée issues d'un capteur de température en IoT environnemental.

Système temps réel : notions importantes

- Prédictibilité :
 - Degré de confiance dans la réalisation de quelque-chose (par ex : le temps de complétion d'une action)
 - Mal défini et discuté dans la communauté scientifique...
- Concurrence
 - Exécution de tâches en parallèle
 - Peuvent être plus ou moins critiques.
 - Ex : acquisition capteurs + stockage data + IHM.
- Préemption :
 - Capacité de l'ordonnanceur d'un système à interrompre une tâche en cours au profit d'une tâche de priorité supérieure.

Système embarqués : contraintes additionnelles

- **Contraintes de consommation :**
 - Des données identiques en entrée produisent des résultats identiques (classique dans les systèmes)
- **Déterminisme temporel :**
 - Des contraintes temporelles doivent être respectées.
 - Les contraintes temporelles dépendent beaucoup des constantes de temps du système.
- **Fiabilité :**
 - La disponibilité du système doit être assurée.

Real Time Operating System (RTOS)

- Qu'est-ce qu'un RTOS ?
 - Couche logicielle permettant d'assurer un fonctionnement respectant des contraintes du système temps réel.
 - Intègre un noyau logiciel installé en bas niveau sur le hardware



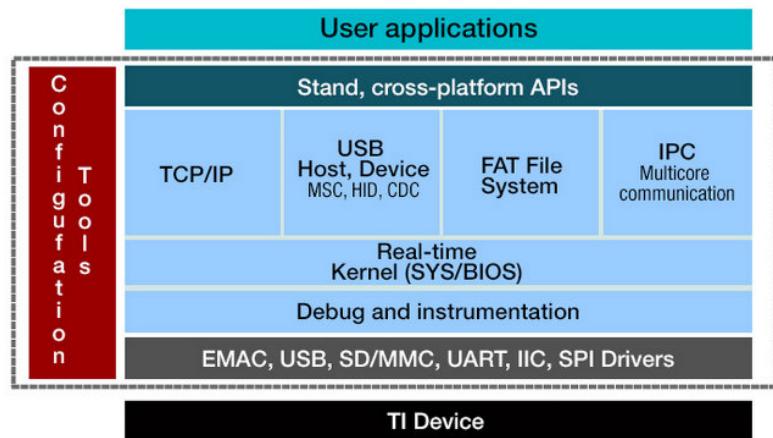
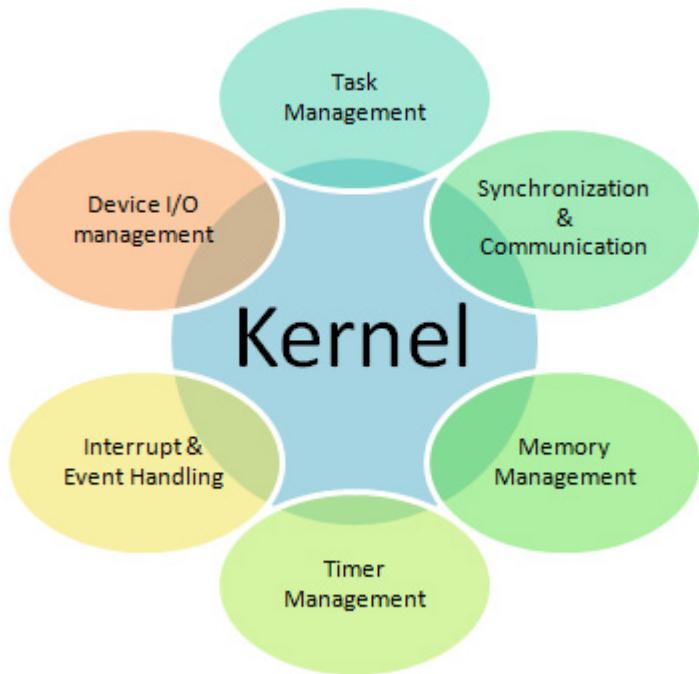
Fig. Real time embedded system with RTOS

- L'application est développée en surcouche du RTOS.
 - Programmation orientée RTOS
 - Besoin de connaître le RTOS

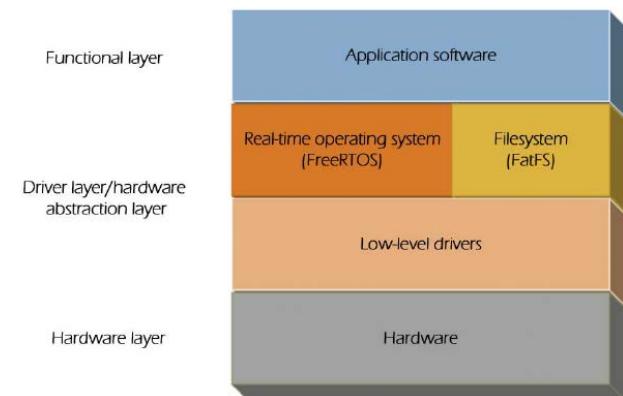
Système temps réel : architecture

□ Le noyau intègre des services de bas niveau

Ex : Ti RTOS



Ex : Free RTOS

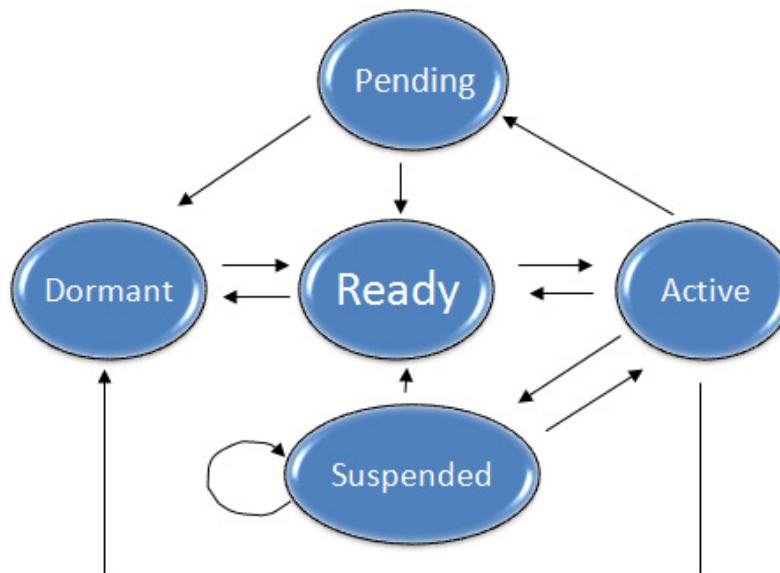


Système temps réel : les tâches

- L'application est divisée en petit morceaux séquentiels appelés tâches.
- Une tâche possède :
 - Un Task Control Block : permet la sauvegarde de contexte de la tâche en RAM
 - ID
 - Priorité
 - Etat
 - Pointeur de stack
 - Program counter
 - Une stack : zone RAM dédiée
 - Une routine : zone programme dédiée

Système temps réel : les tâches

- Les tâches sont créées et détruites par le RTOS
 - ▣ ThreadCreate / ThreadTerminate
- Elles sont appelée par le scheduler.
- Elles ont des états parmi les suivants :



Système temps réel : les tâches

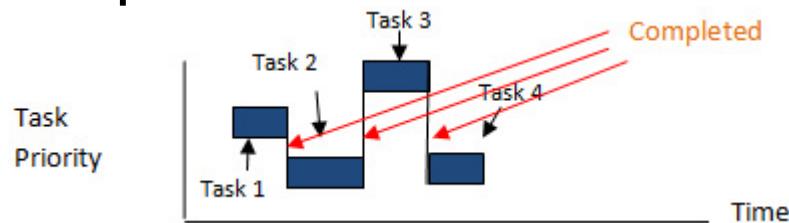
- Le noyau exécute l'exécution parallèle de tâches grâce au scheduler :
 - Polled loop : appel successif des tâches
 - Polled loop with interrupts : polling avec gestion des interruptions critiques
 - Round Robin : time slicing



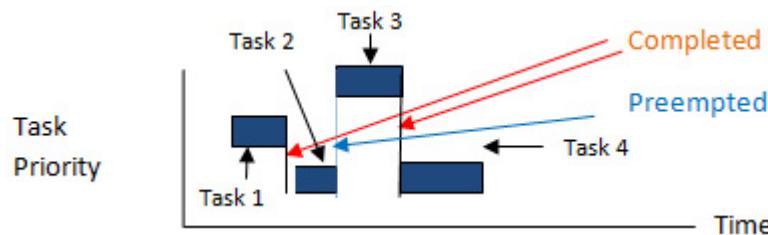
- Hybrid : round robin + interrupts

Système temps réel : les tâches

- Le scheduler peut être préemptif ou pas :
 - Non Préemptif :



- Préemptif : les tâches sont interrompues par celles de priorité supérieure



Système temps réel : synchronisation

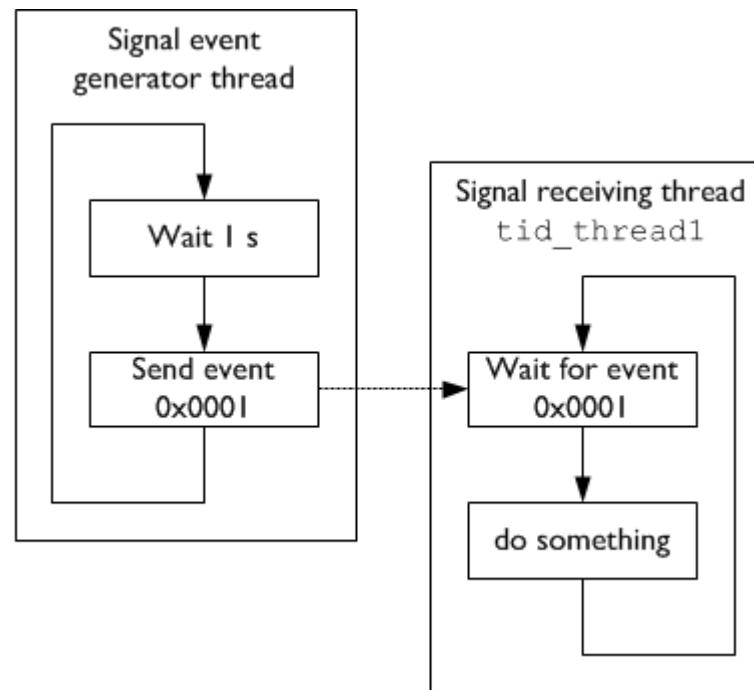
□ Synchronisation entre tâches :

- Permet l'ordonnancement de tâches successives indépendantes.
 - Basé sur les évènements
- Permet le partage de ressources mutualisées
 - Basé sur des sémaphores :
 - Contient un compteur et une queue
 - Trois types :
 - Sémaphores binaires
 - Sémaphores à compteur
 - Mutex (Mutual Exclusion)
- Permet le passage d'infos d'un thread à l'autre
 - Message Queue
 - Memory Pool

Système temps réel : synchronisation

□ Evènement

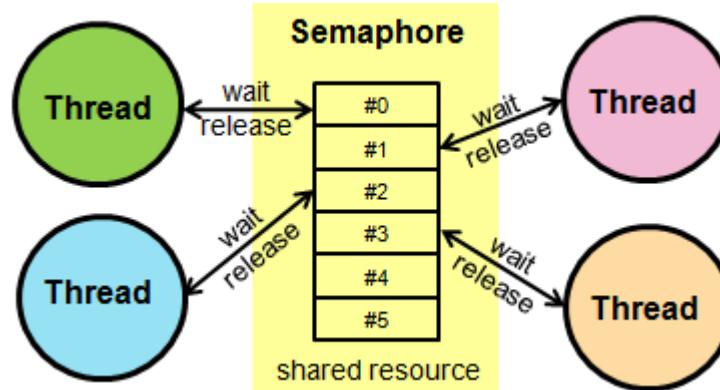
- Une tache attend qu'un évènement soit réalisé pour démarrer



Système temps réel : synchronisation

□ Séaphore

- Permet un nombre limité d'accès à une ressource depuis différentes tâches.
- La ressource reste partagée et donc indépendante

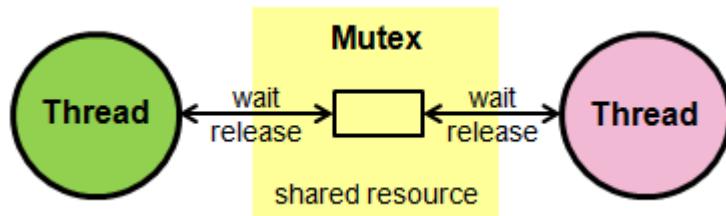


- Un séaphore binaire n'a qu'une ressource partagée.
- Prise de contrôle : Wait ou Acquire
- Rendu de la ressource : Release

Système temps réel : synchronisation

□ Mutex

- Sémaphore limité à un accès.
- Le contrôle de la ressource est attribué au thread appelant.



- Ne peut être utilisé depuis une interruption

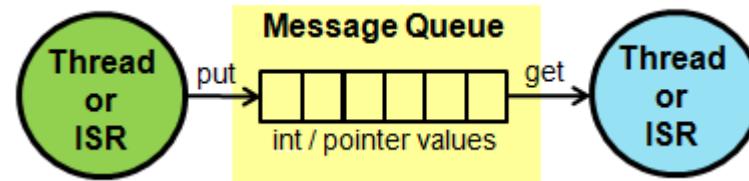
Système temps réel : synchronisation

□ Memory pool

- Permet de passer du contenu d'une tâche à une autre via un espace mémoire défini thread safe.
- Moins simple à utiliser qu'une message queue

□ Message queue

- Permet de passer du contenu d'une tâche à une autre



- Fonctionne à l'aide d'un FIFO type I/O
- Moins efficace qu'une memory pool.

Système temps réel : gestion mémoire

- La stack (pile) : utilisée par le TCB
 - La pile permet de stocker le contexte des tâches.
- La heap (tas) :
 - Mémoire autre que le programme, ses variables et la stack.
 - Permet de stocker des objets dynamiques en mémoire
- Stack et Heap sont géré par le RTOS

Système temps réel : gestion timer et interruptions

- La gestion des timers :
 - Géré par le RTOS.
- Gestion des interruptions :
 - Géré par le RTOS
 - Attention à éviter une gestion native si un RTOS est utilisé.
 - Voir le cas de TI RTOS.
- Gestion du hardware :
 - Le plus souvent géré par le RTOS au niveau des interruptions.

Un exemple de système temps réel : Ti RTOS

- Focus sur la gestion des HWI – SWI et des Tasks
- Implantation sur CC2650 ou CC2652
 - SoC basé sur un ARM M3 (CC2650) ou M4 (CC2652)
 - Permet de gérer des capteurs et IHM.
 - Permet de gérer la connectivité BLE ou IEEE 802.15.4

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**

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

Interrupt service routines

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

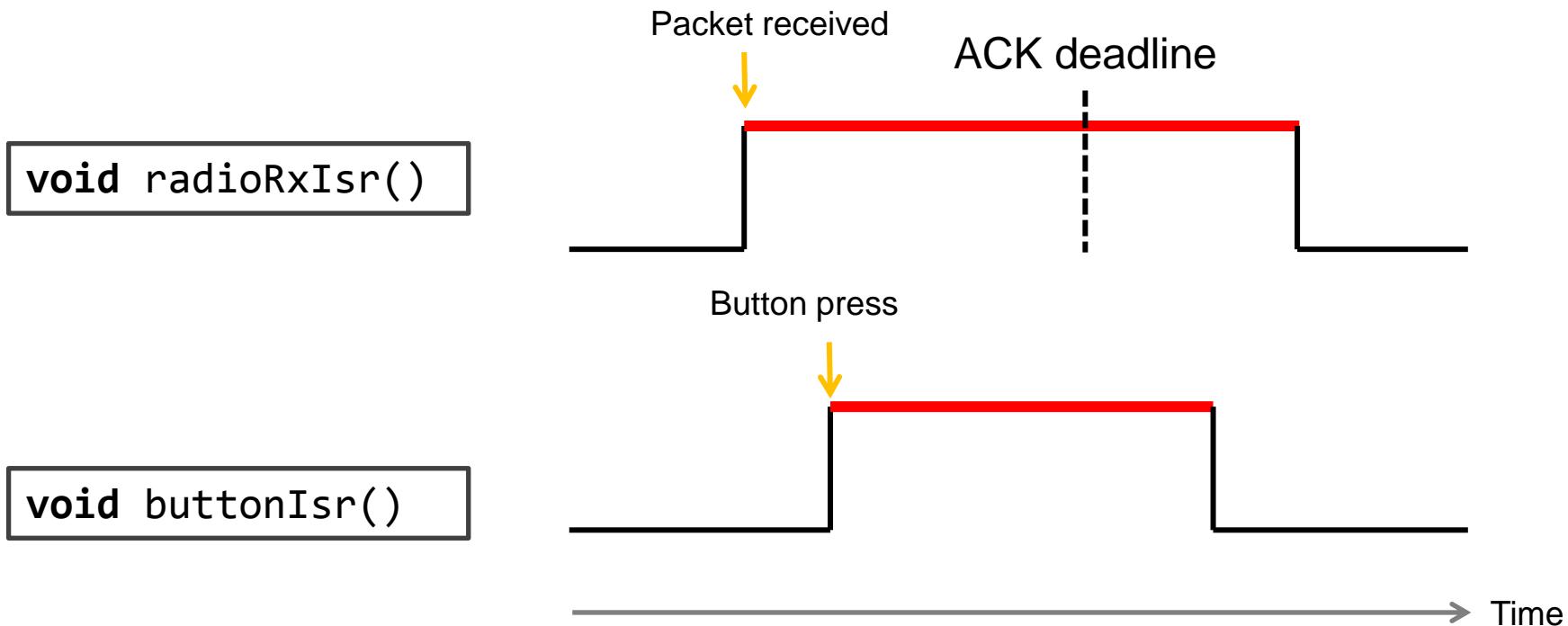
```
void scheduleTimerIsr() {  
    /* Check schedule */  
    /* Update target temp. */  
}
```

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

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

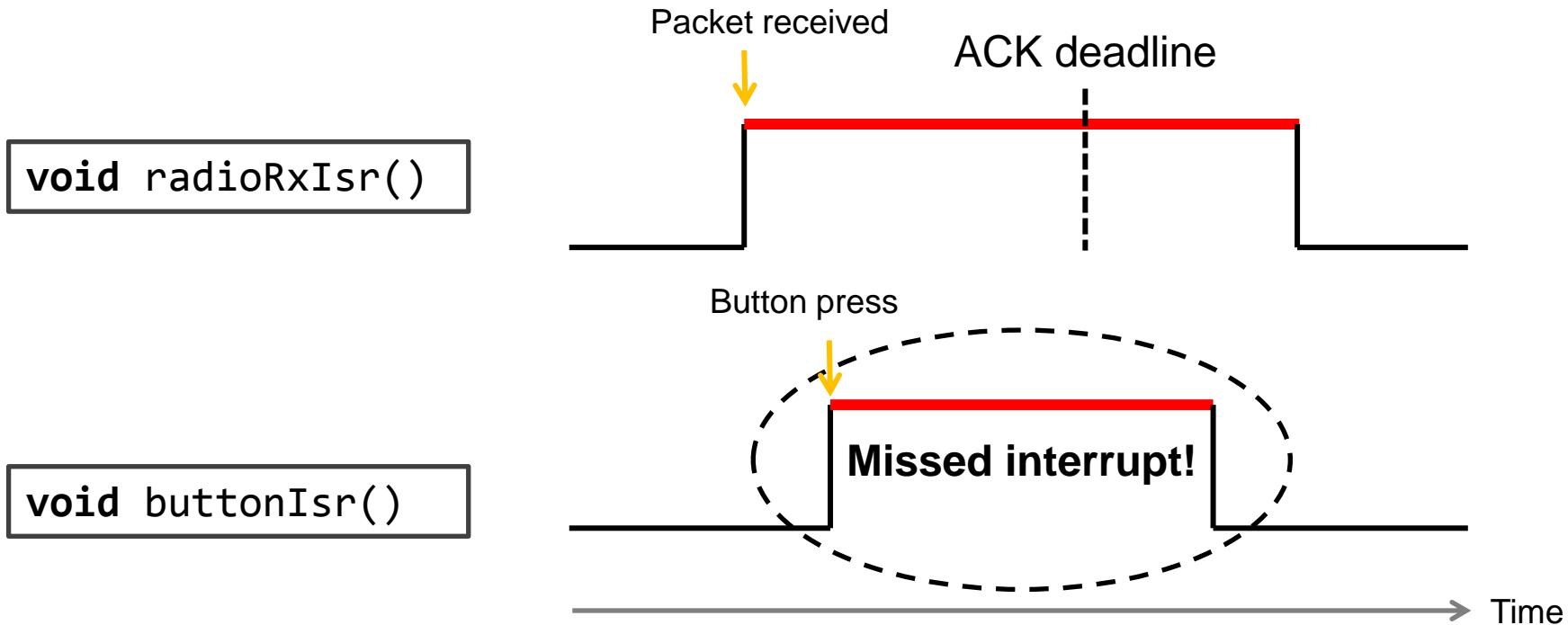
Bare Metal – Smart Thermostat

Interrupt service routines – No nesting



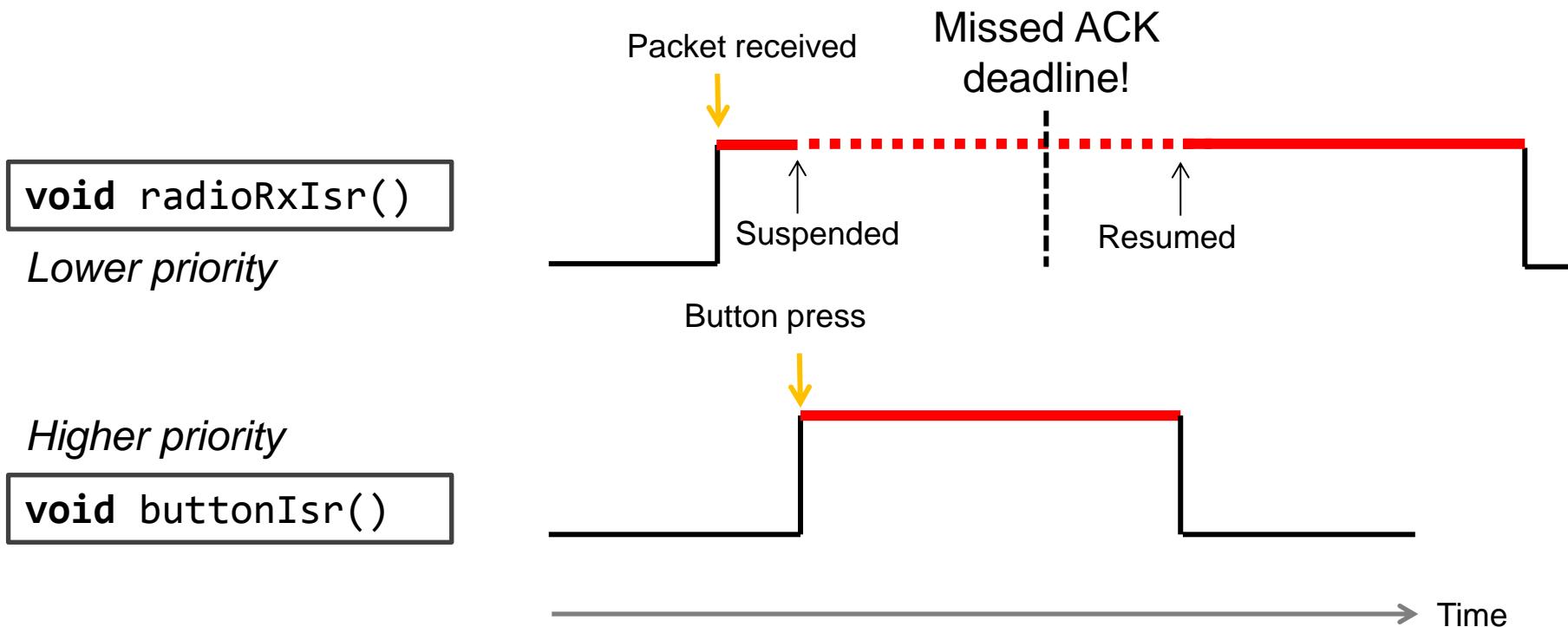
Bare Metal – Smart Thermostat

Interrupt service routines – No nesting



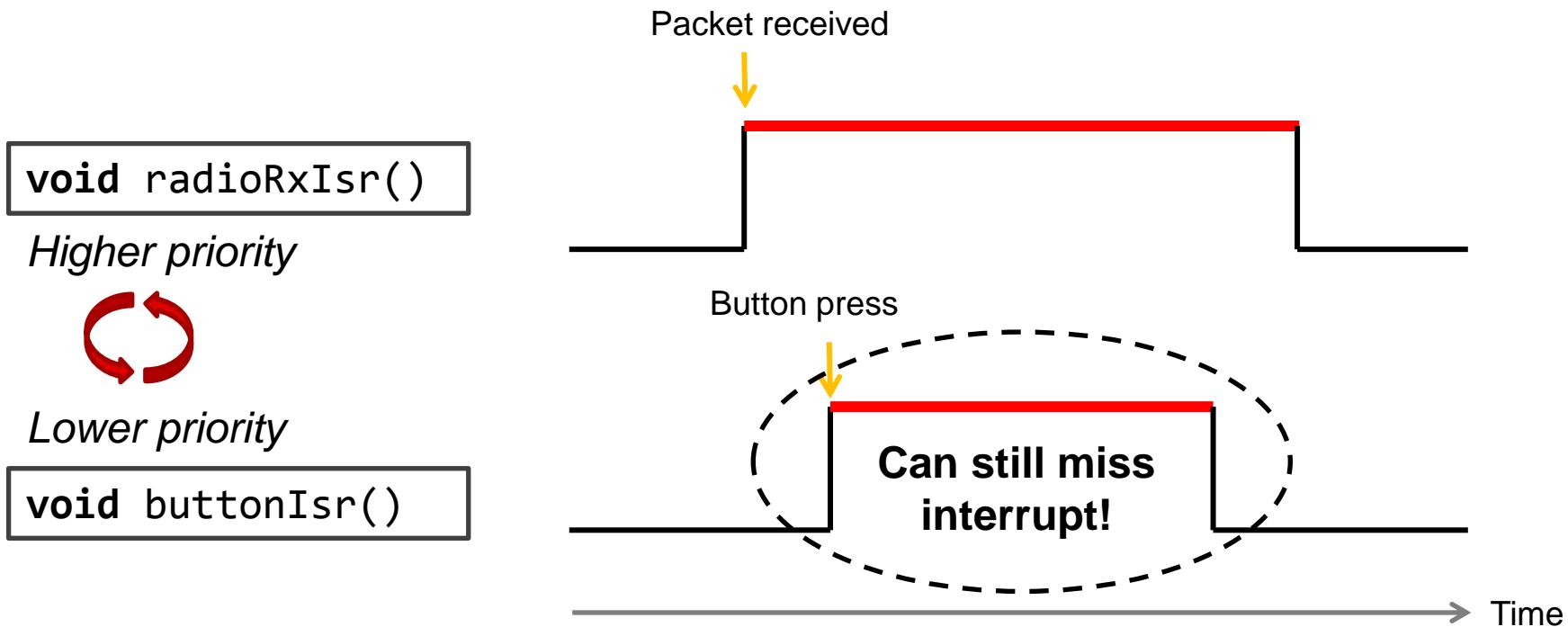
Bare Metal – Smart Thermostat

Solution: Nested Interrupt service routines

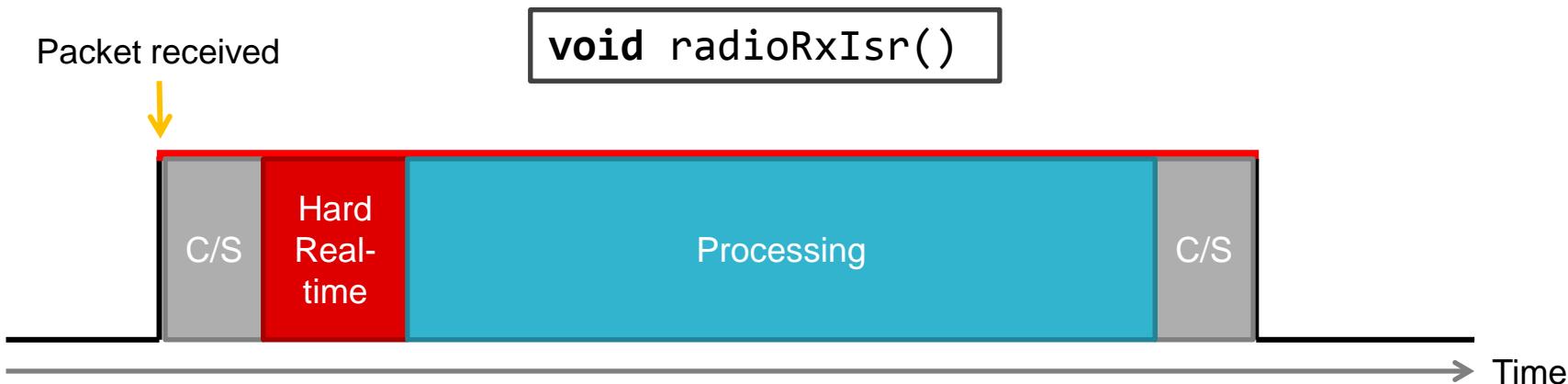


Bare Metal – Smart Thermostat

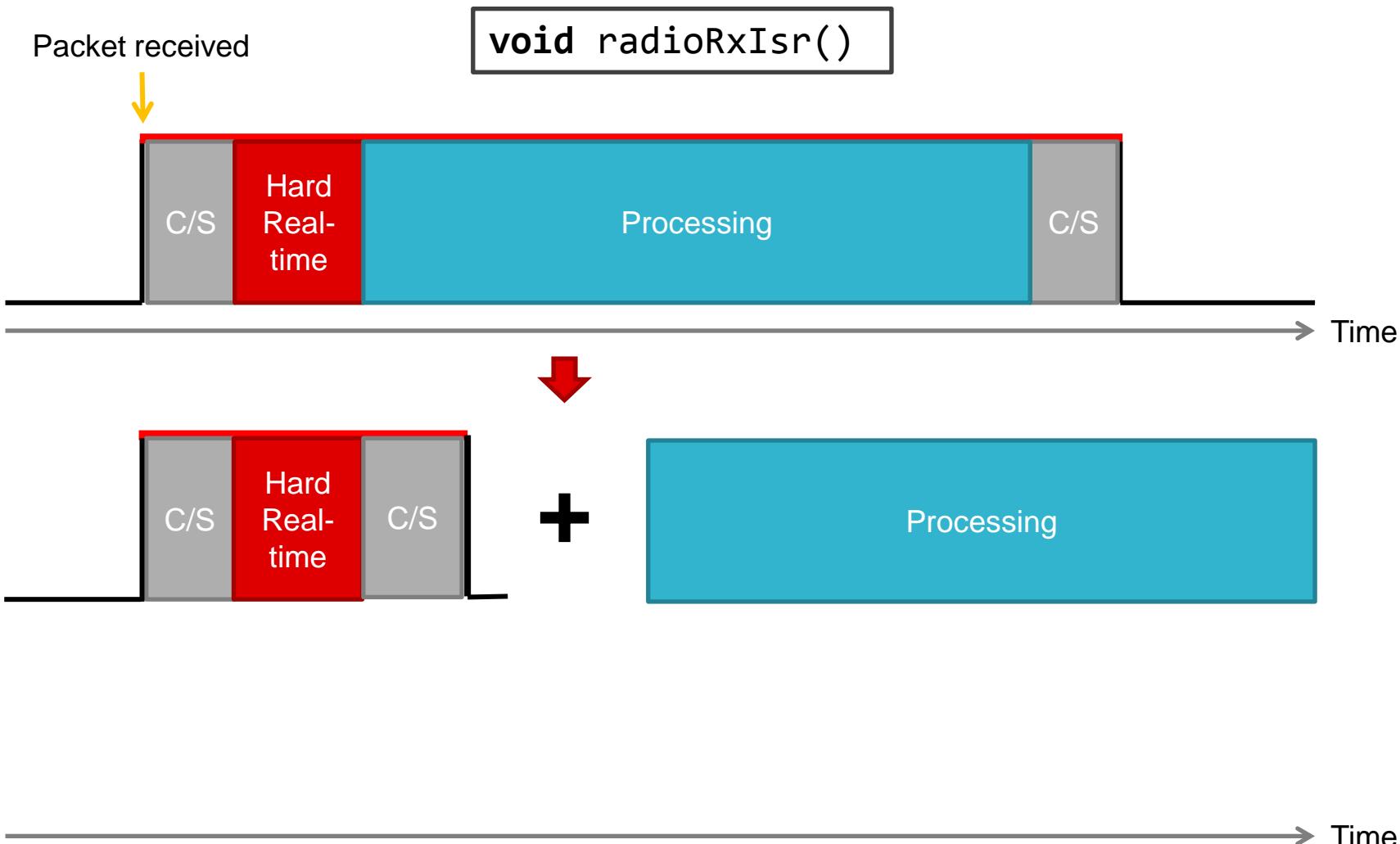
Solution: Nested Interrupt service routines



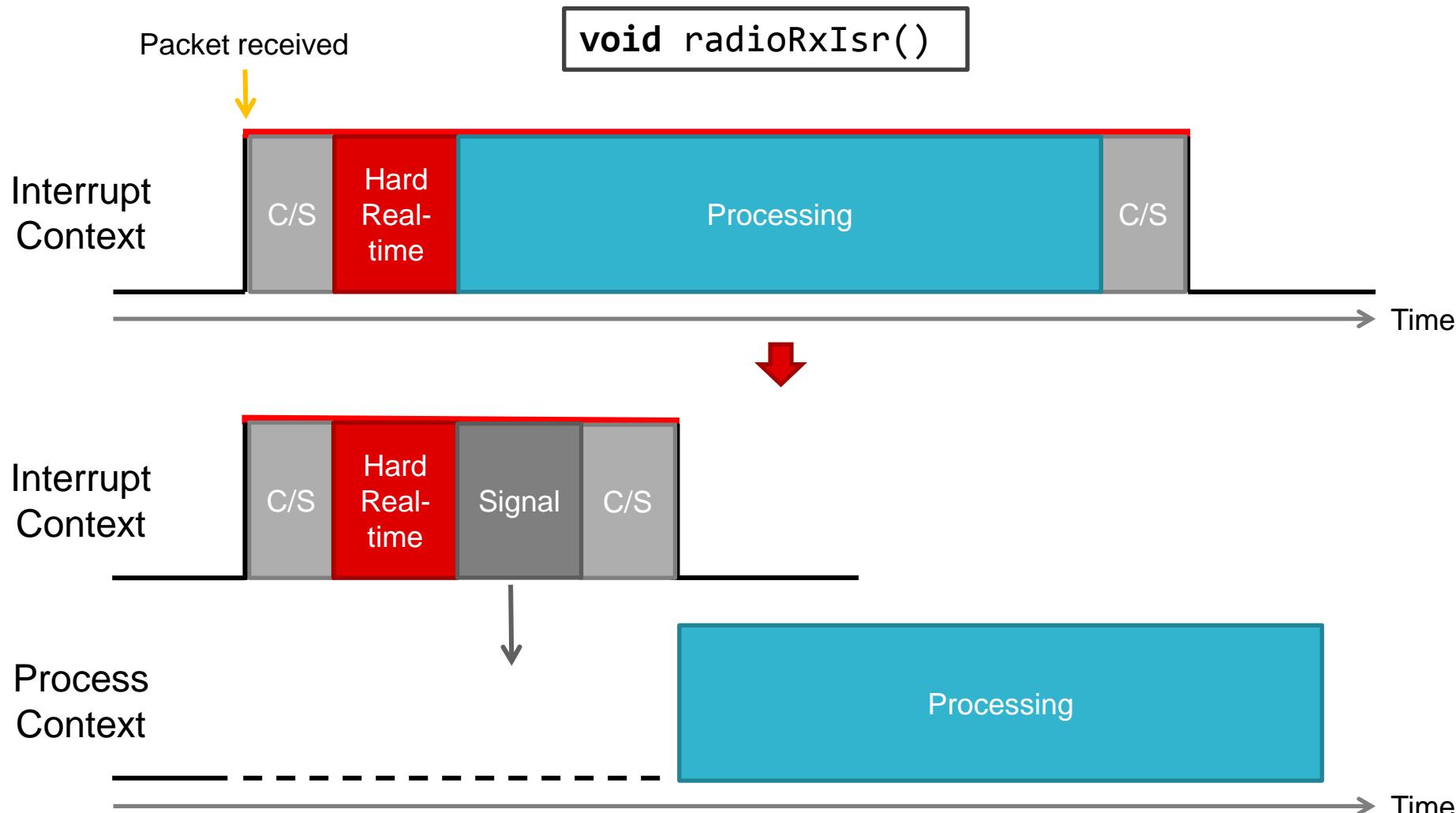
Bare Metal – Smart Thermostat



Bare Metal – Smart Thermostat



Bare Metal – Smart Thermostat



Bare Metal – Smart Thermostat

- Interrupt driven with processing in main

```
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

Nested Interrupt service routines

`void radioRxIsr()`

Lower priority

Packet received

ACK deadline

Suspended

Resumed

`void buttonIsr()`

Higher priority

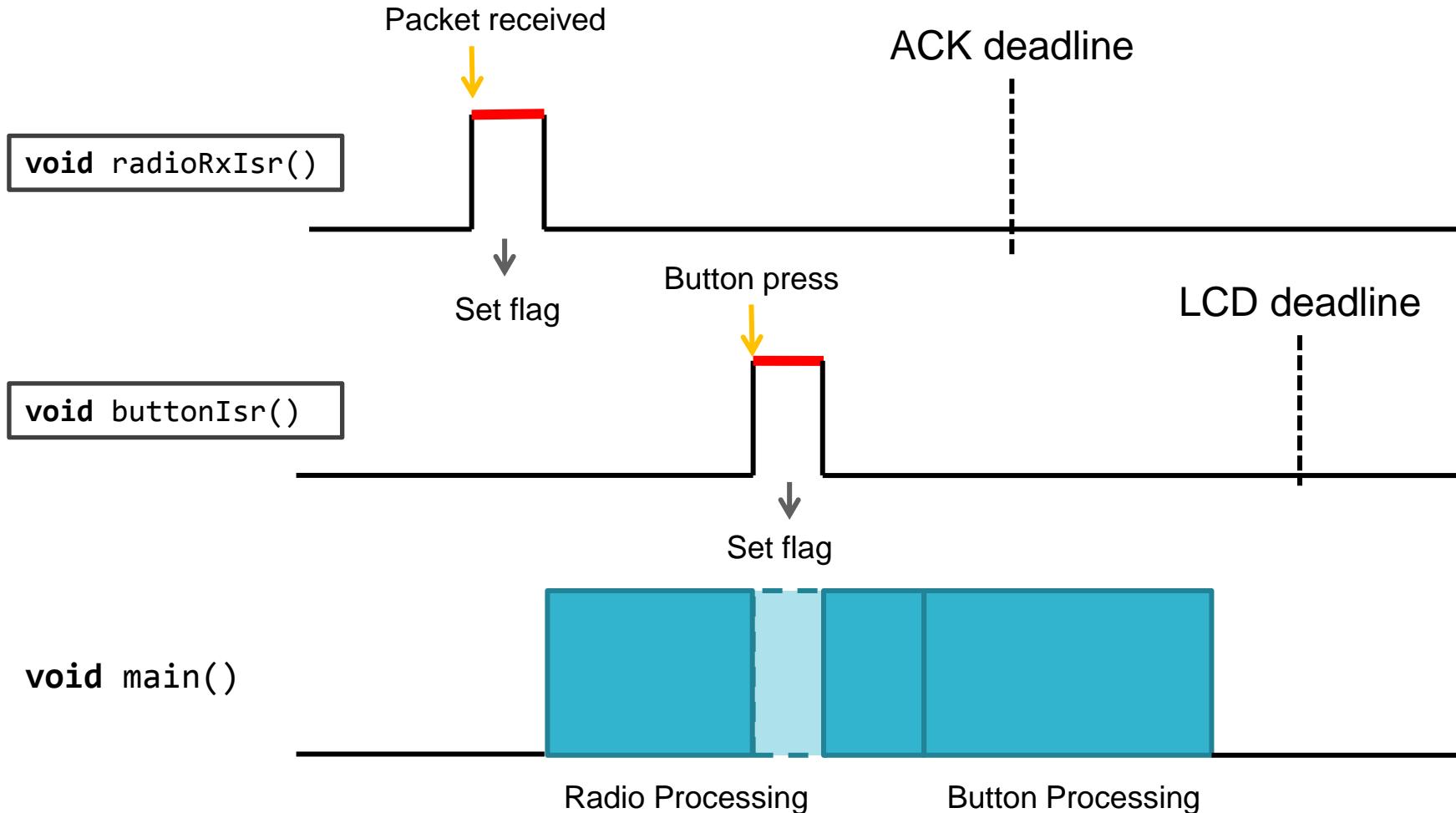
Button press

LCD deadline

Time

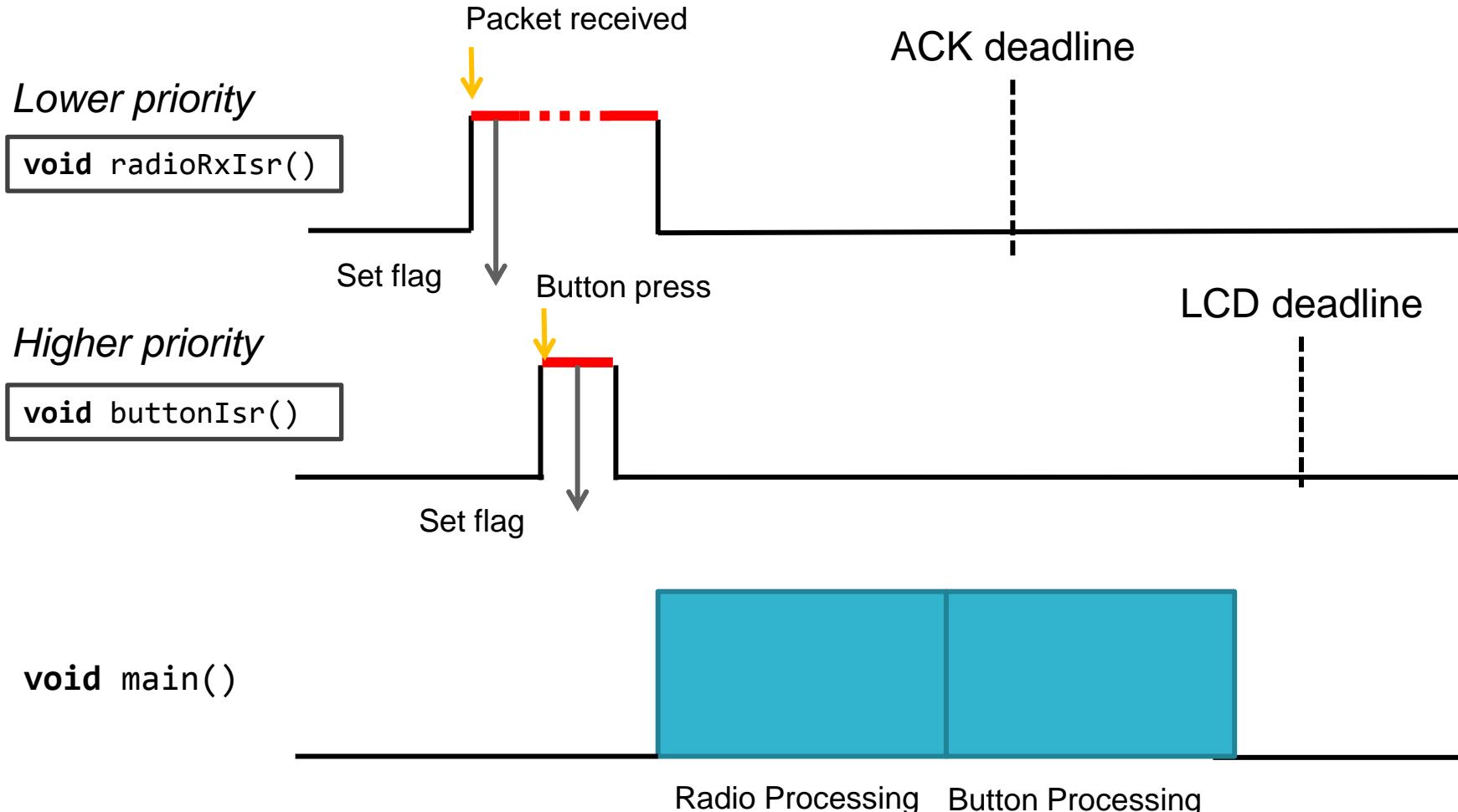
Bare Metal – Smart Thermostat

Interrupt service routines



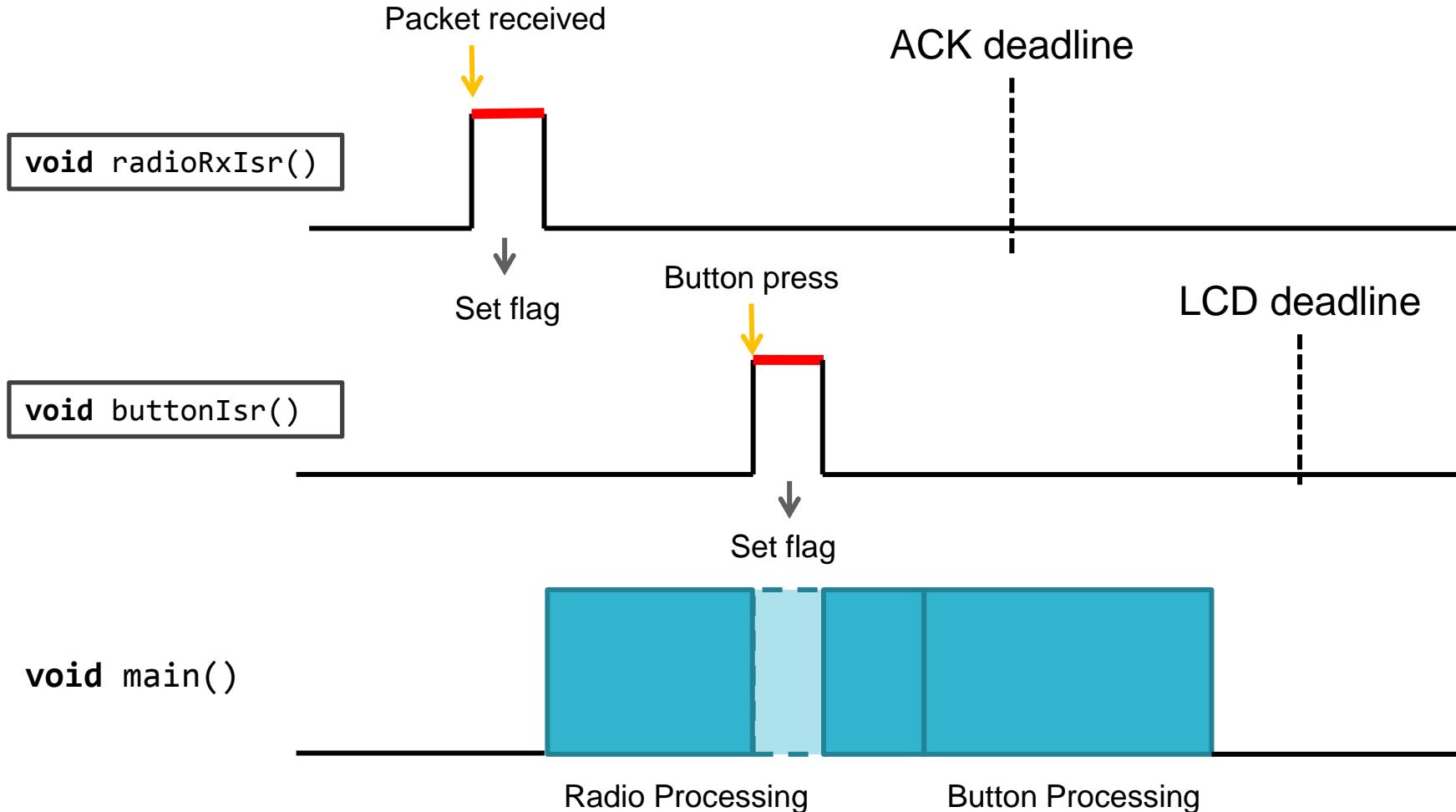
Bare Metal – Smart Thermostat

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

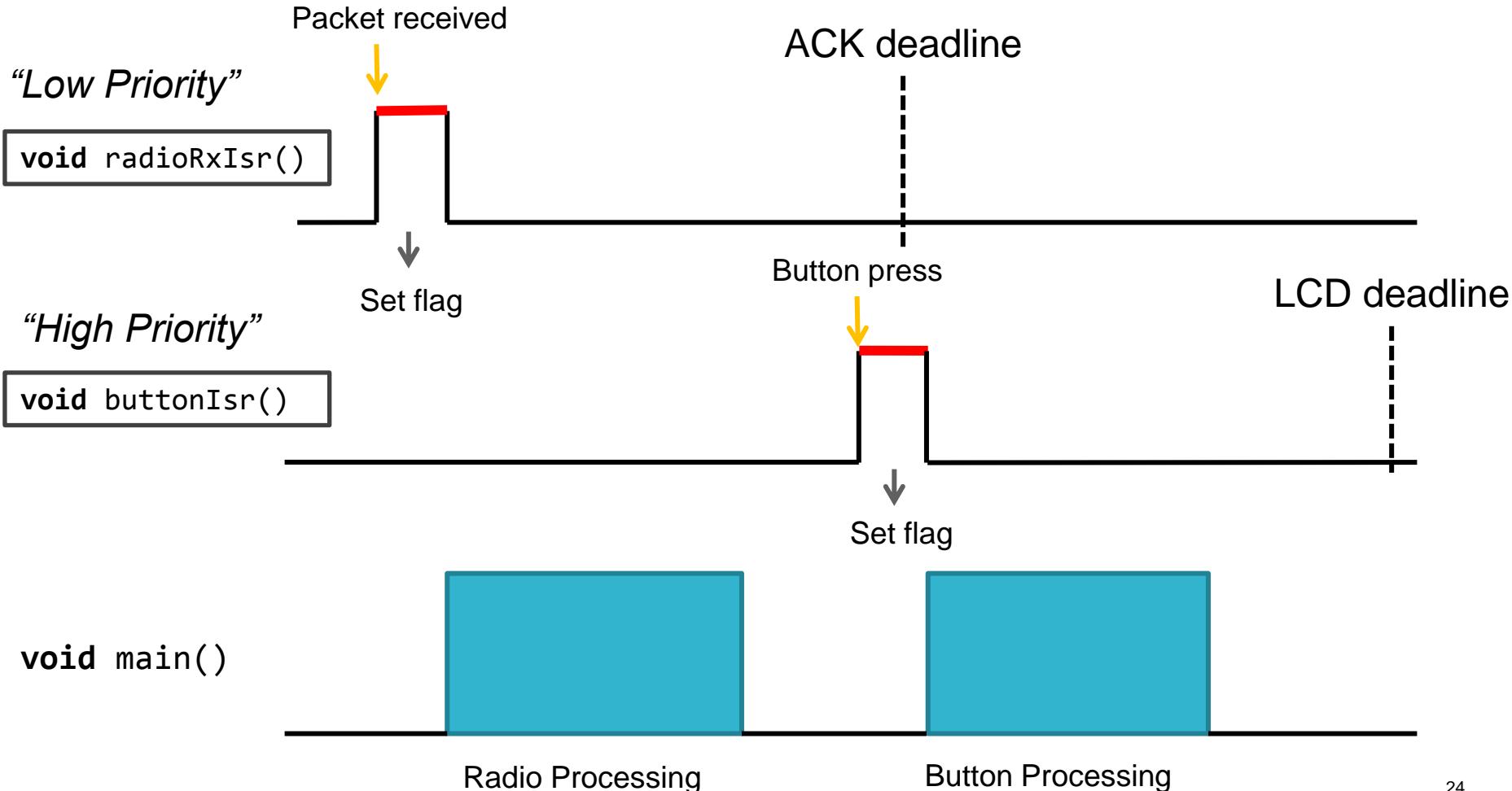
Sensor readings every second

- Temperature
- Humidity

- Schedule operation based on time

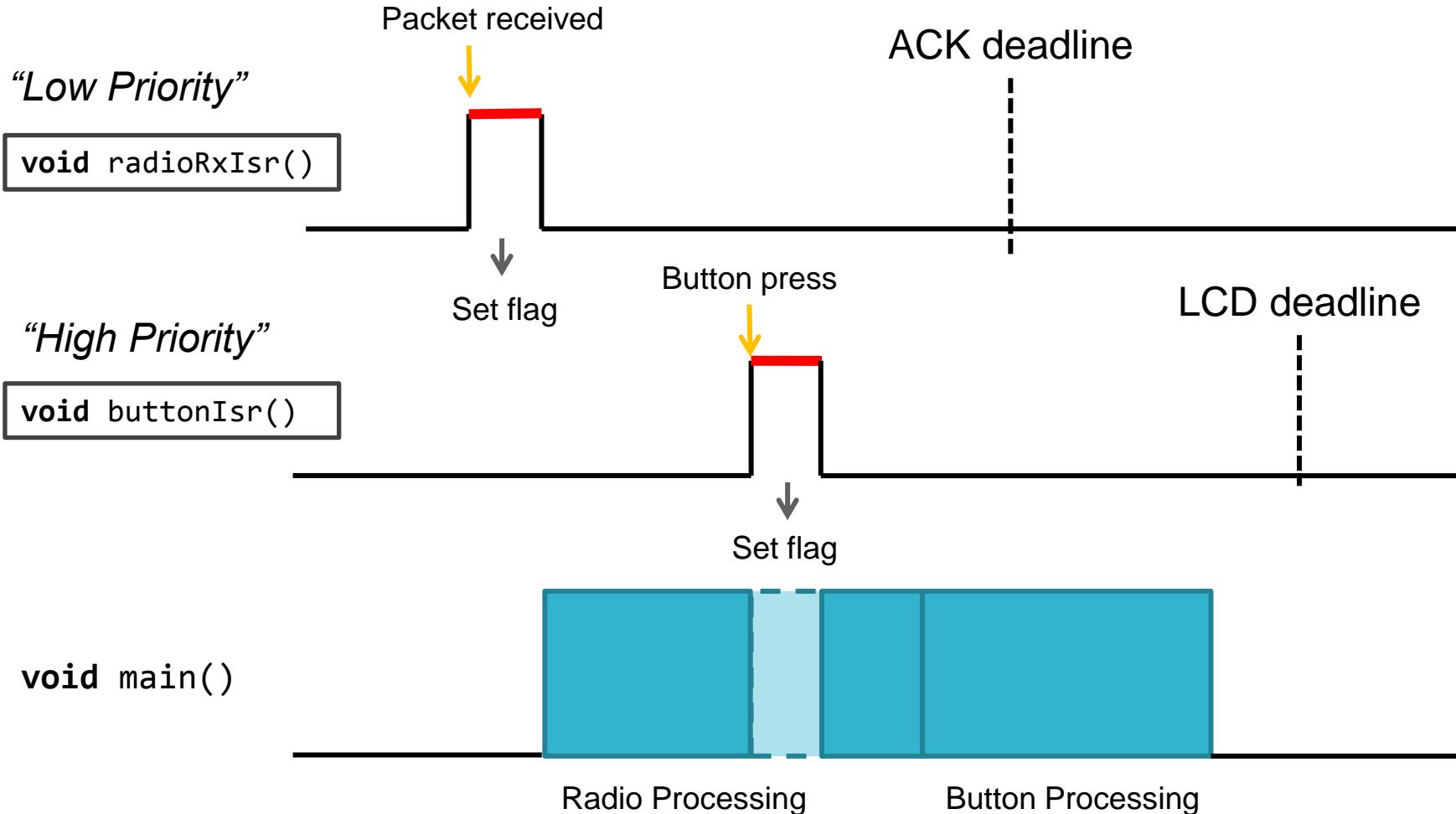
Bare Metal – Smart Thermostat

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



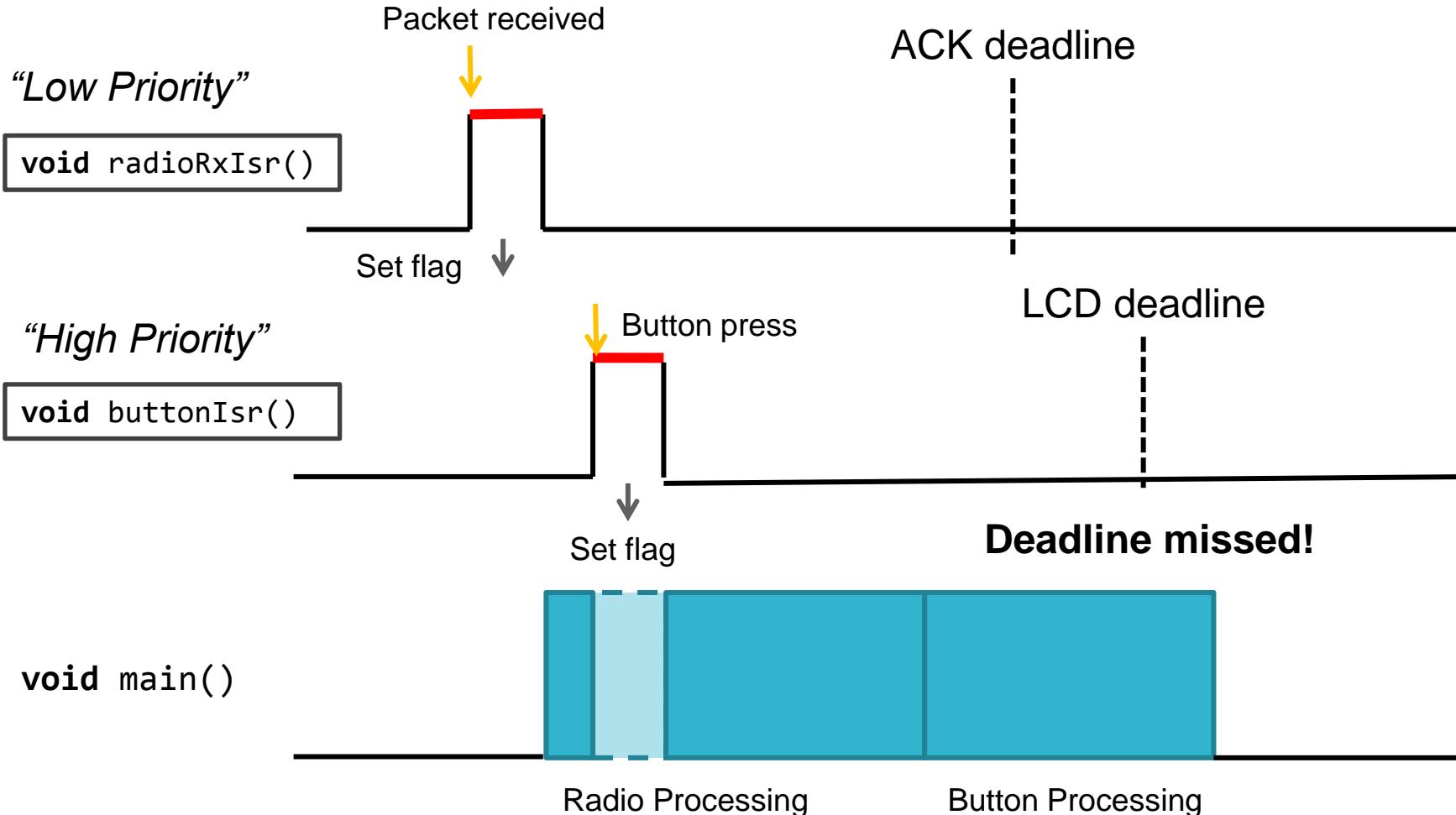
Bare Metal – Smart Thermostat

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
    }
}
```

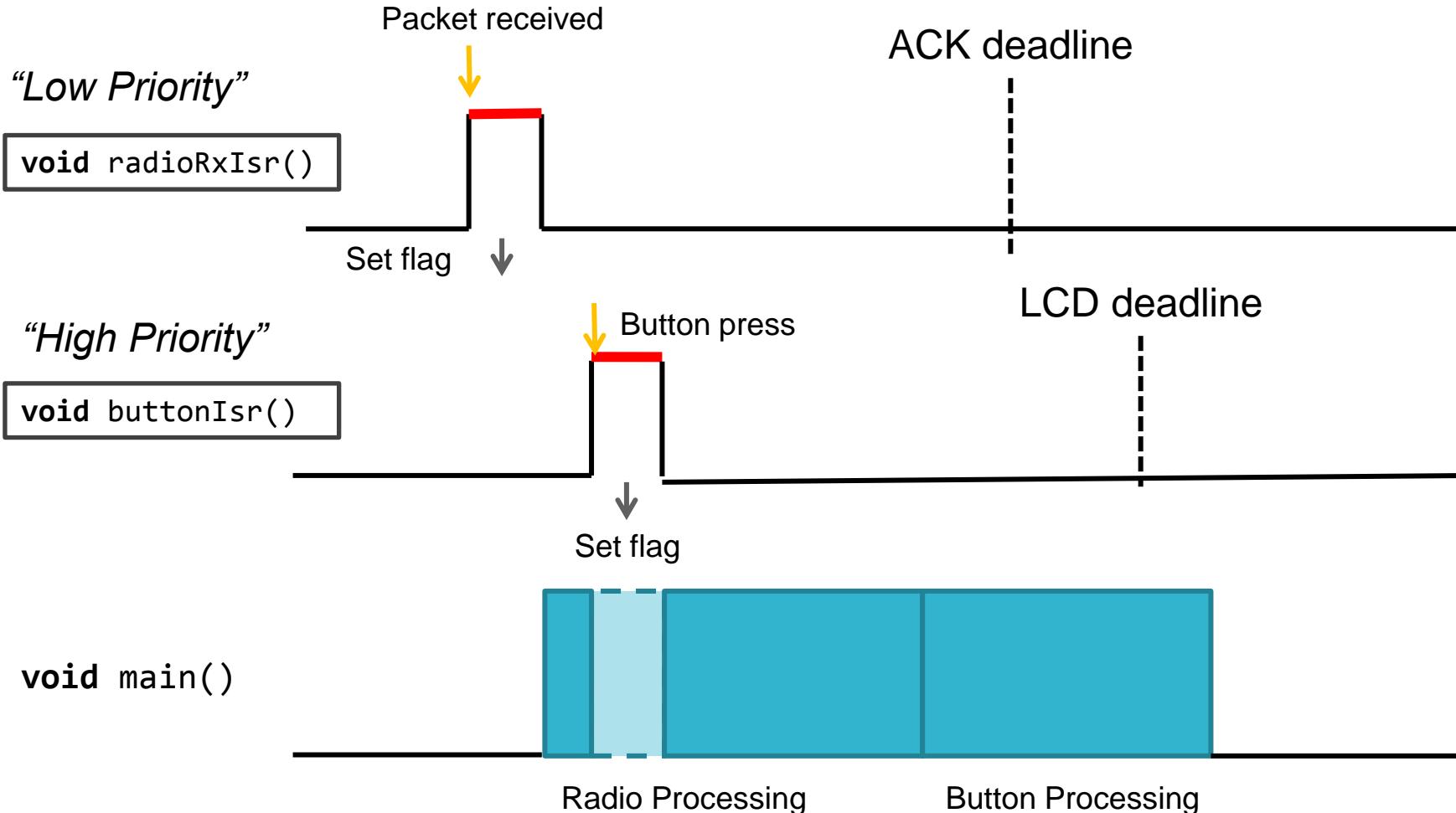
Interrupt service routines

```
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

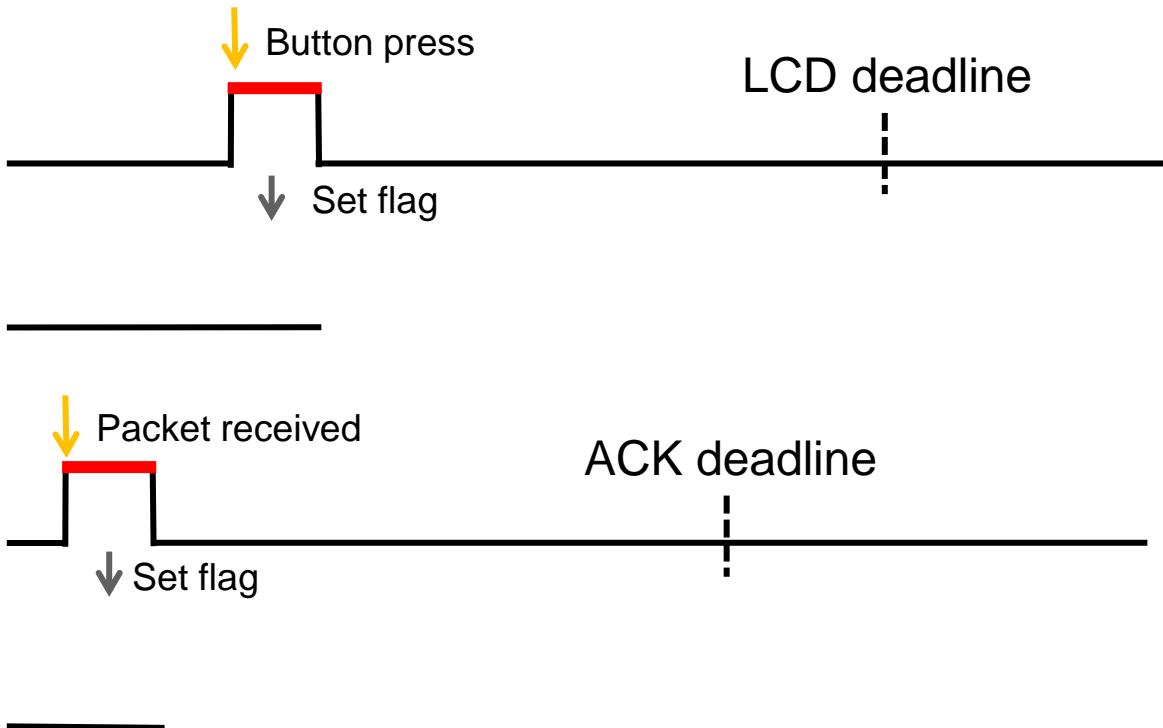
How would we have wanted this to work?

```
void buttonIsr()
```

“High Priority”
`void processButton()`

```
void radioRxIsr()
```

“Low Priority”
`void processPacketRx()`



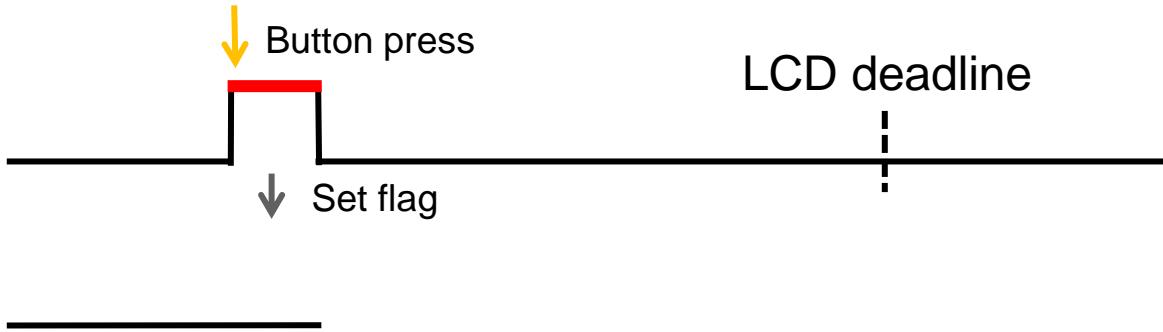
Deadline missed!

Bare Metal – Smart Thermostat

How would we have wanted this to work?

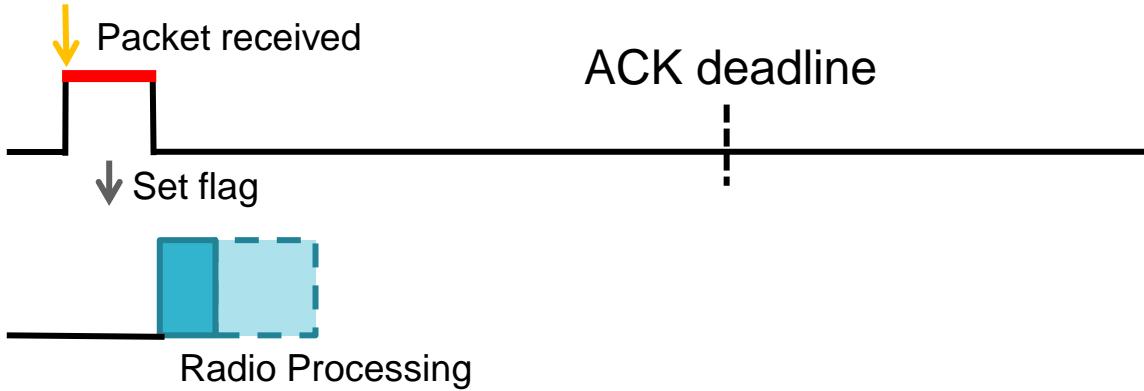
```
void buttonIsr()
```

"High Priority"
`void processButton()`



```
void radioRxIsr()
```

"Low Priority"
`void processPacketRx()`

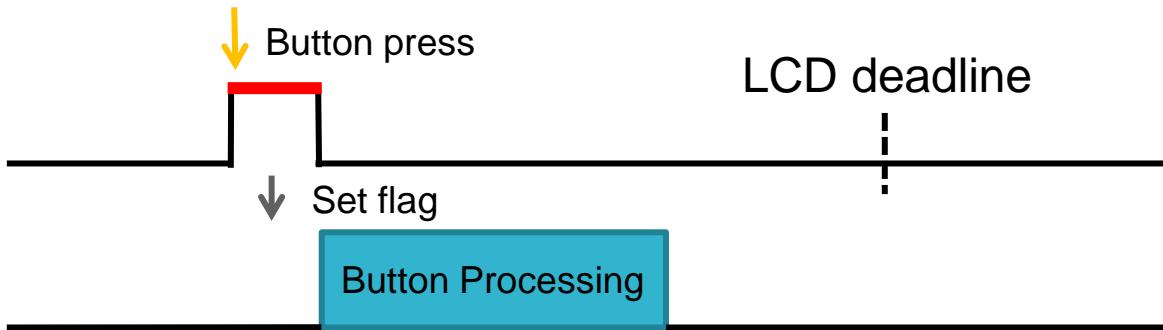


Bare Metal – Smart Thermostat

How would we have wanted this to work?

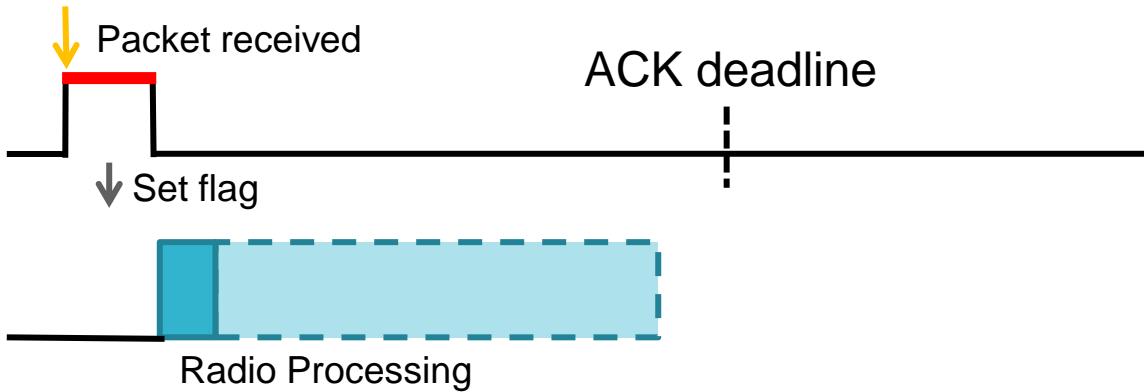
```
void buttonIsr()
```

"High Priority"
`void processButton()`



```
void radioRxIsr()
```

"Low Priority"
`void processPacketRx()`

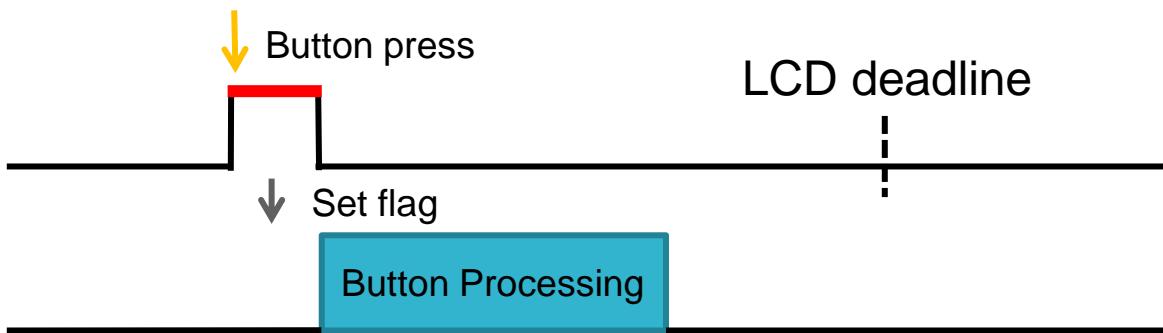


Bare Metal – Smart Thermostat

How would we have wanted this to work?

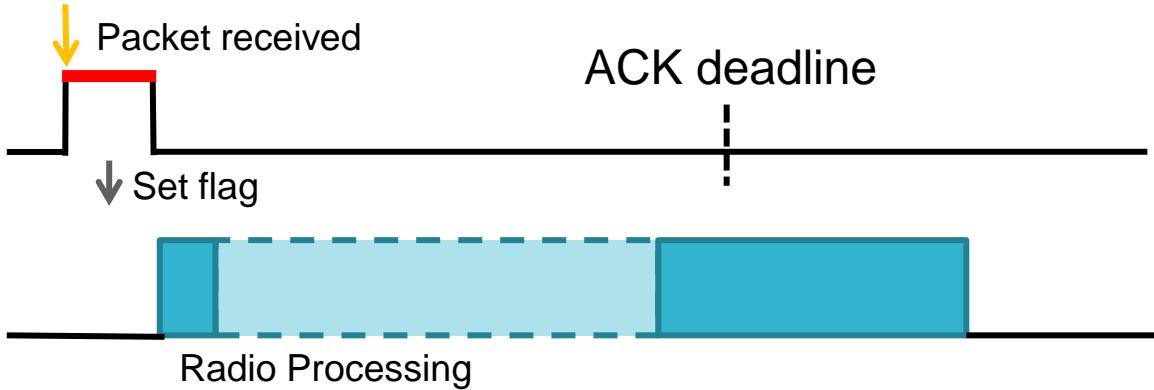
```
void buttonIsr()
```

"High Priority"
`void processButton()`



```
void radioRxIsr()
```

"Low Priority"
`void processPacketRx()`



**Deadline missed,
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”

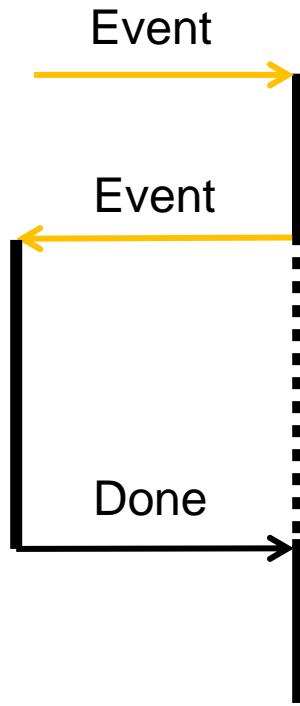
```
void processButton()
```

```
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

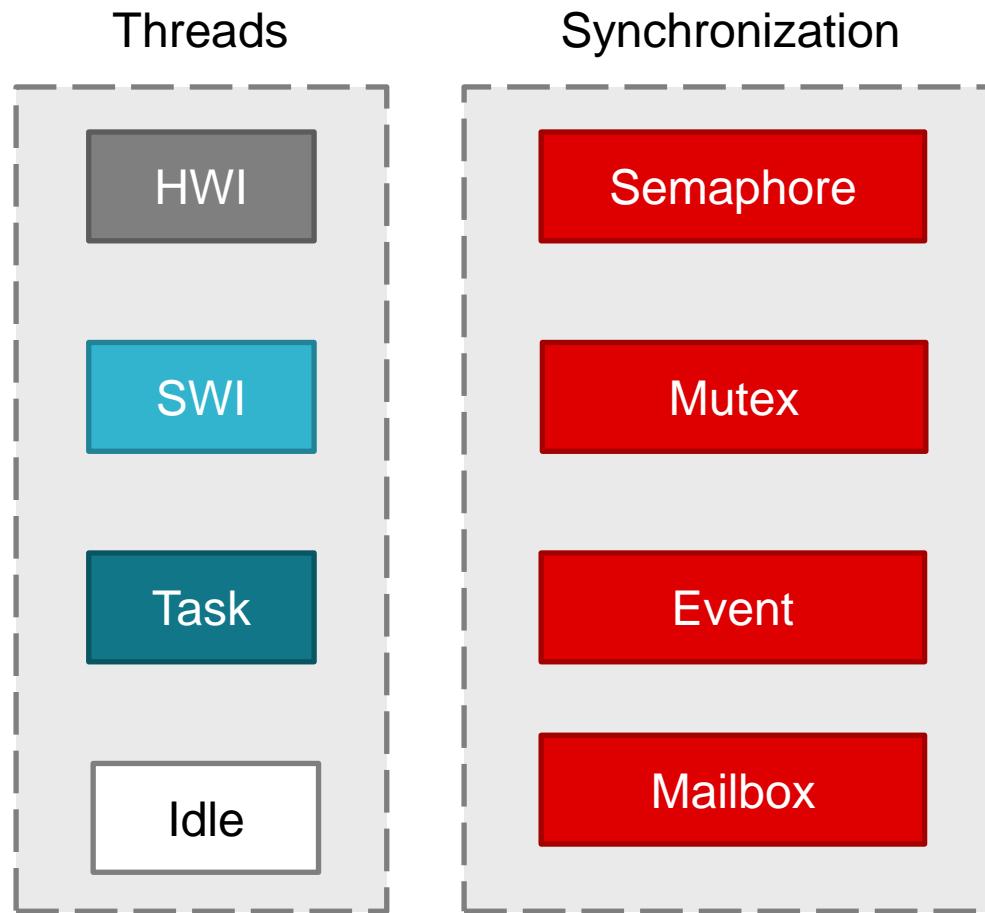
- Each process does its processing in its own context
- An event can trigger both
 - Start of a new process
 - If currently idle
 - Switching the running process
 - If the new process has higher priority than the running one

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

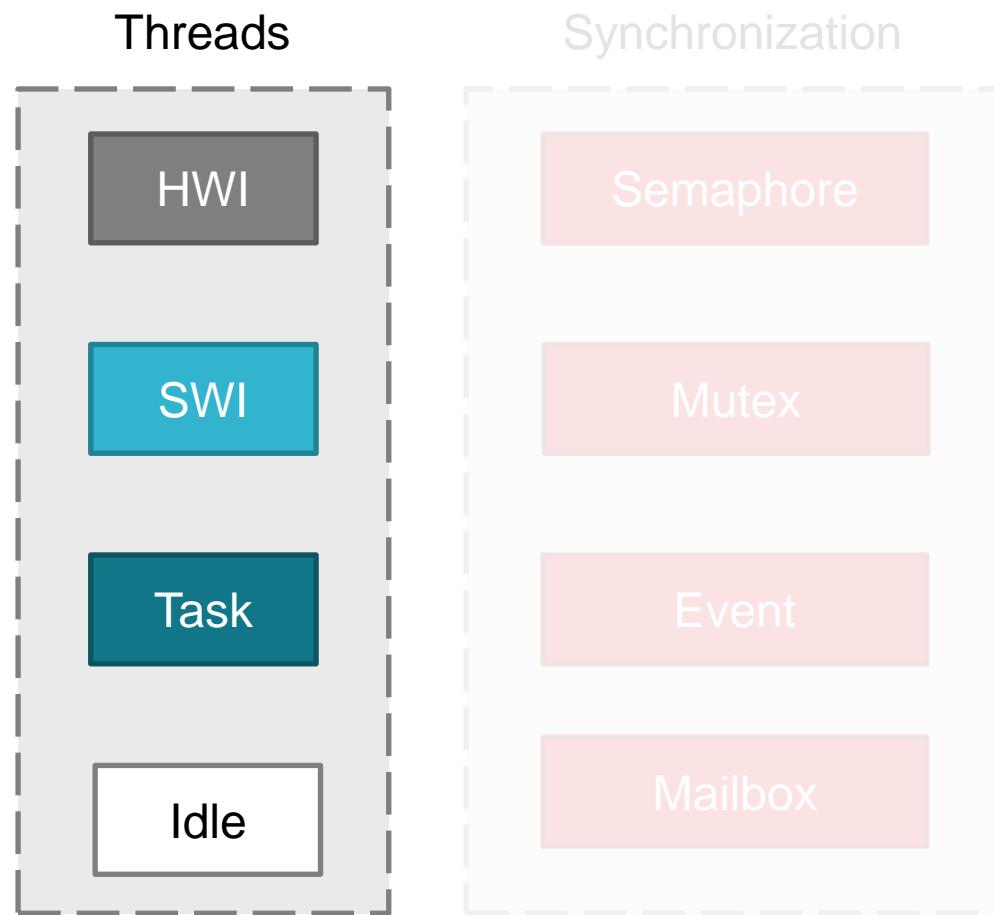
- It makes it simpler to write complex applications
 - Simpler to write event-driven code
 - Simpler to partition code
 - Simpler to guarantee deadlines

TI RTOS Concepts

TI RTOS – Kernel Objects Overview

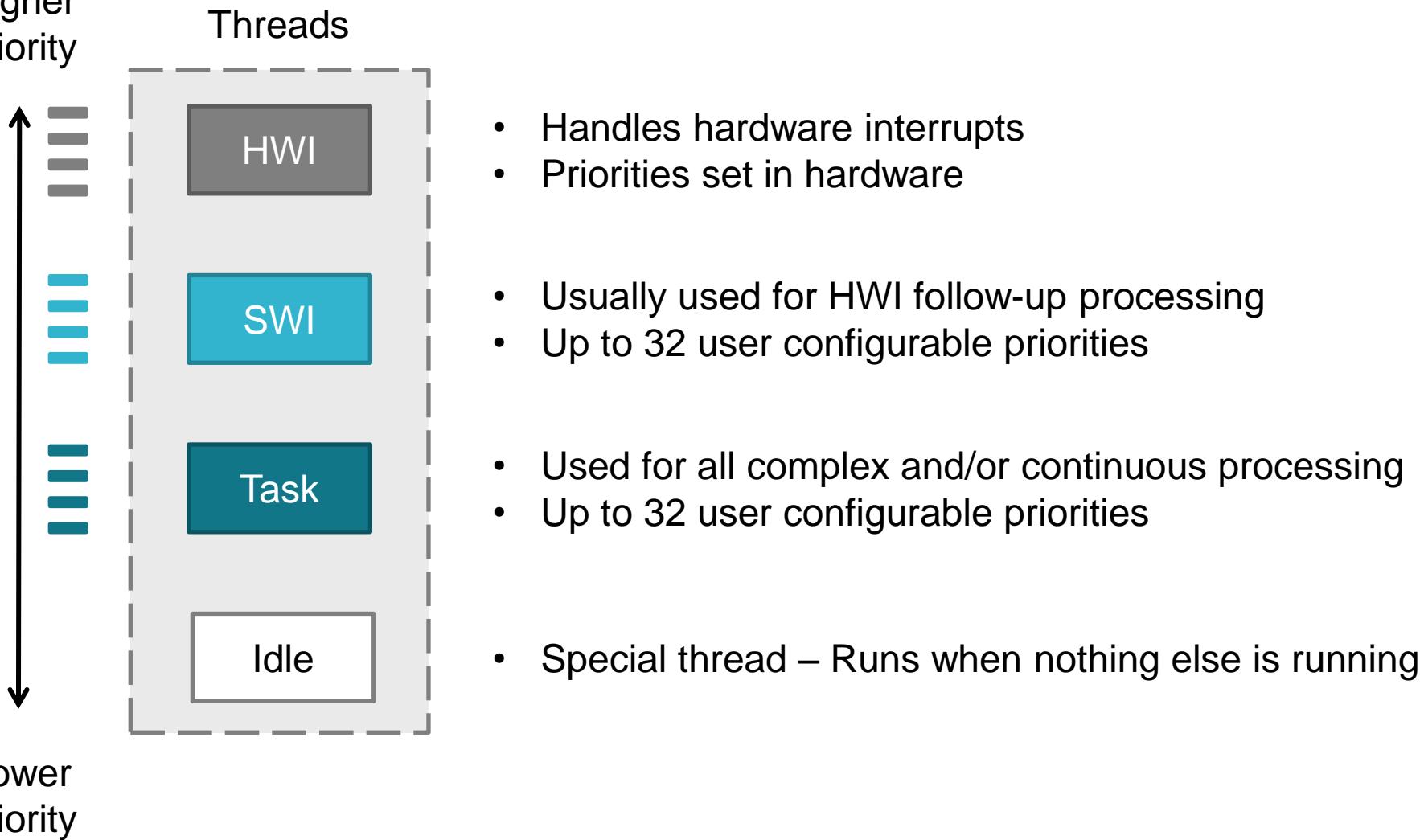


TI RTOS – Kernel Objects Overview



Threads overview

Higher priority



Lower priority

Threads and deadlines

HWI

~5 us

SWI

~100 us

Task

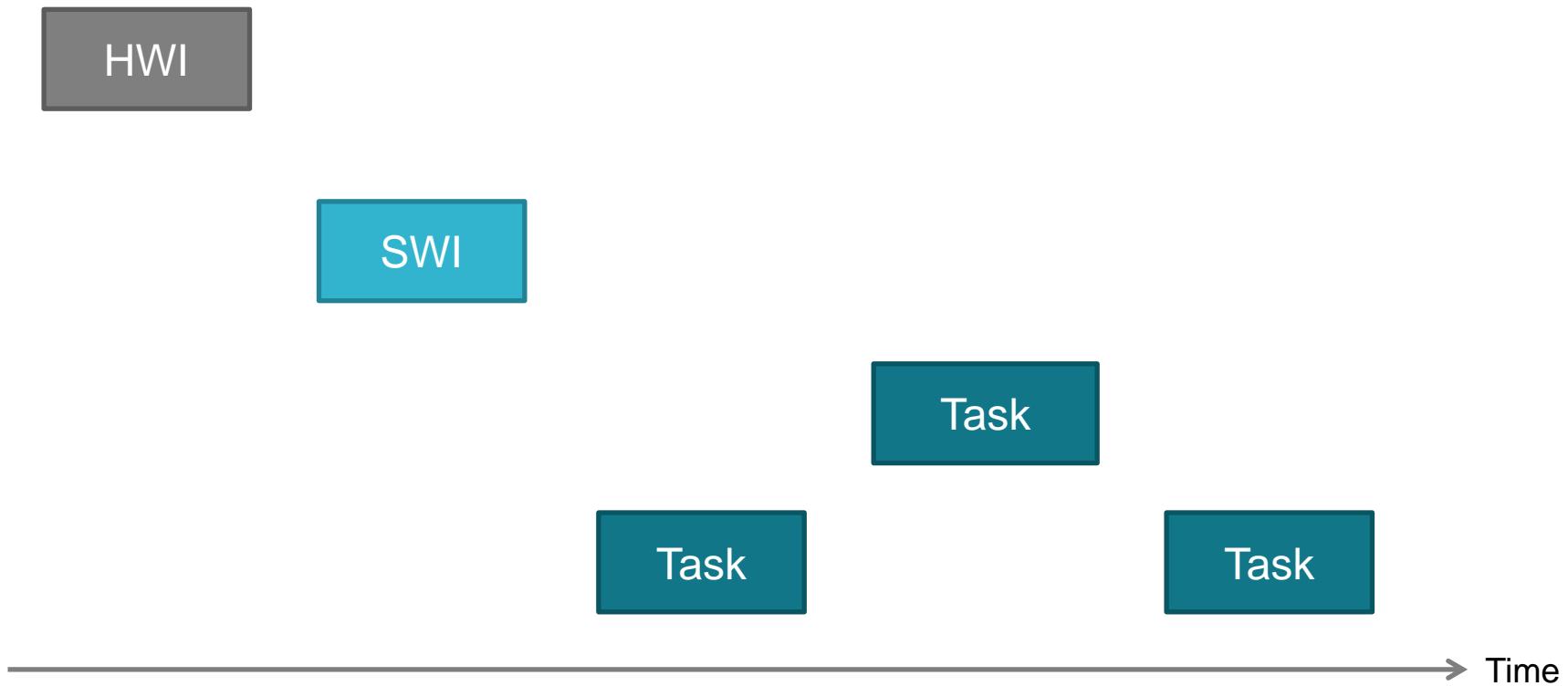
> 100 us

Idle

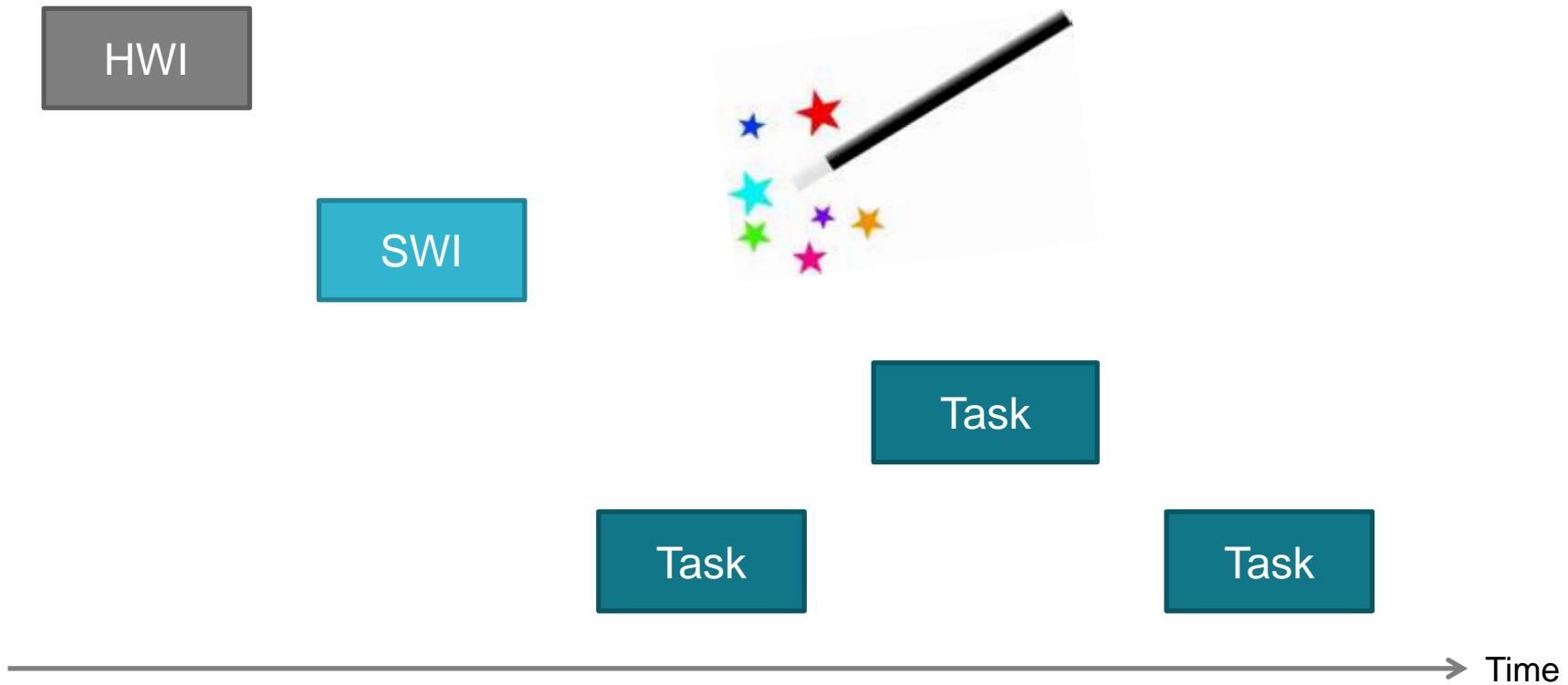
N/A

→ Deadline

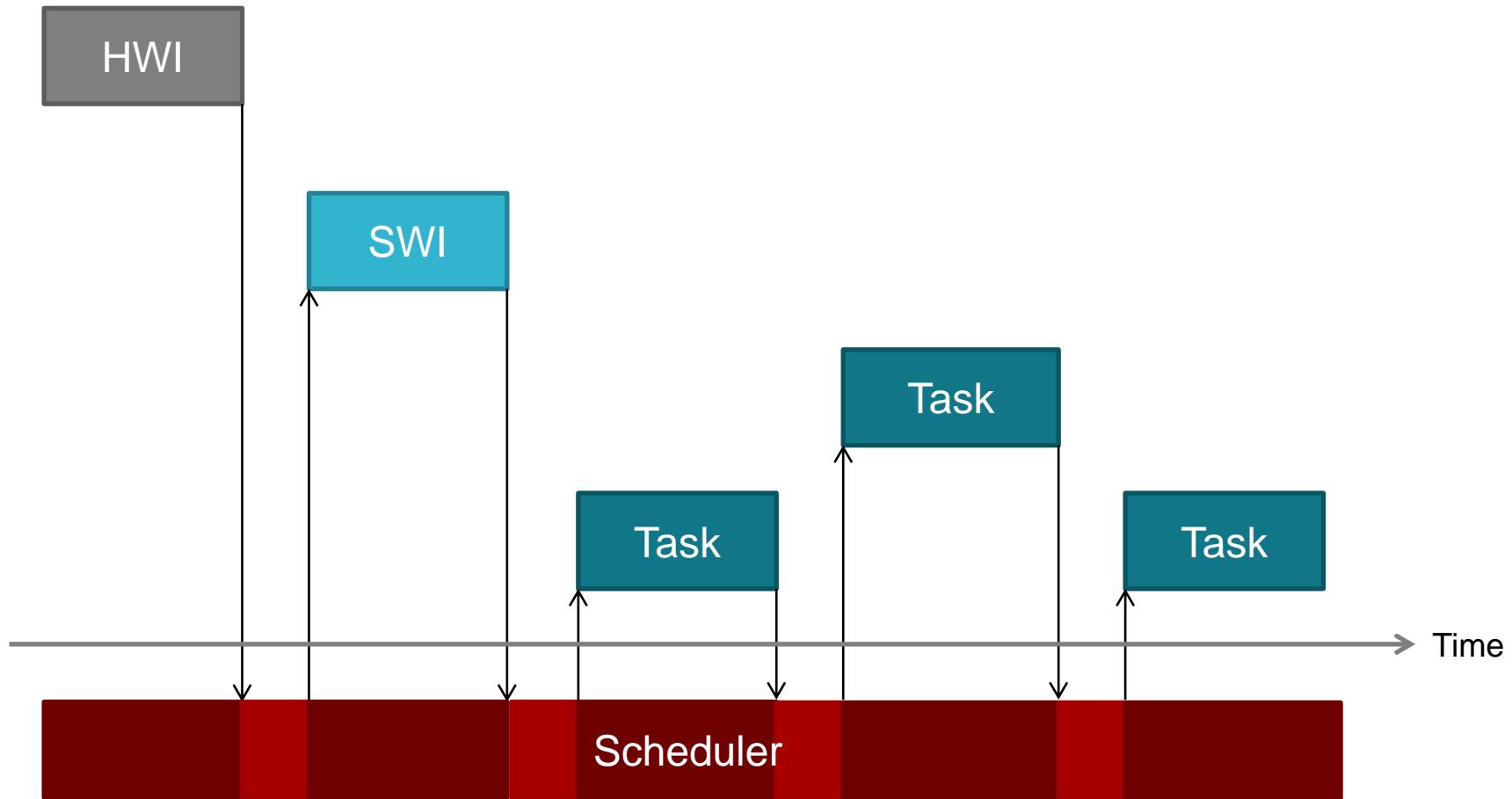
The RTOS scheduler



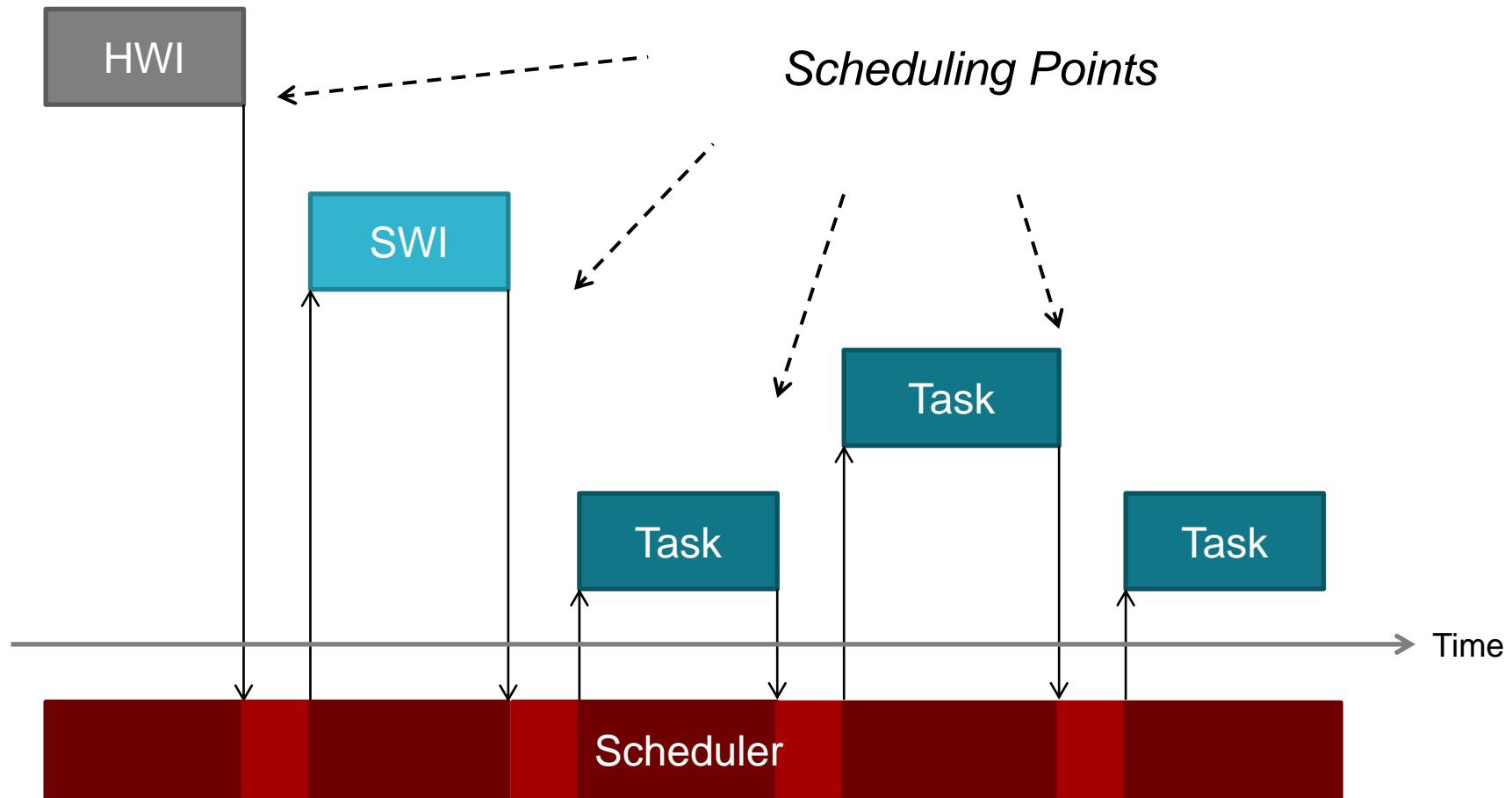
The RTOS scheduler



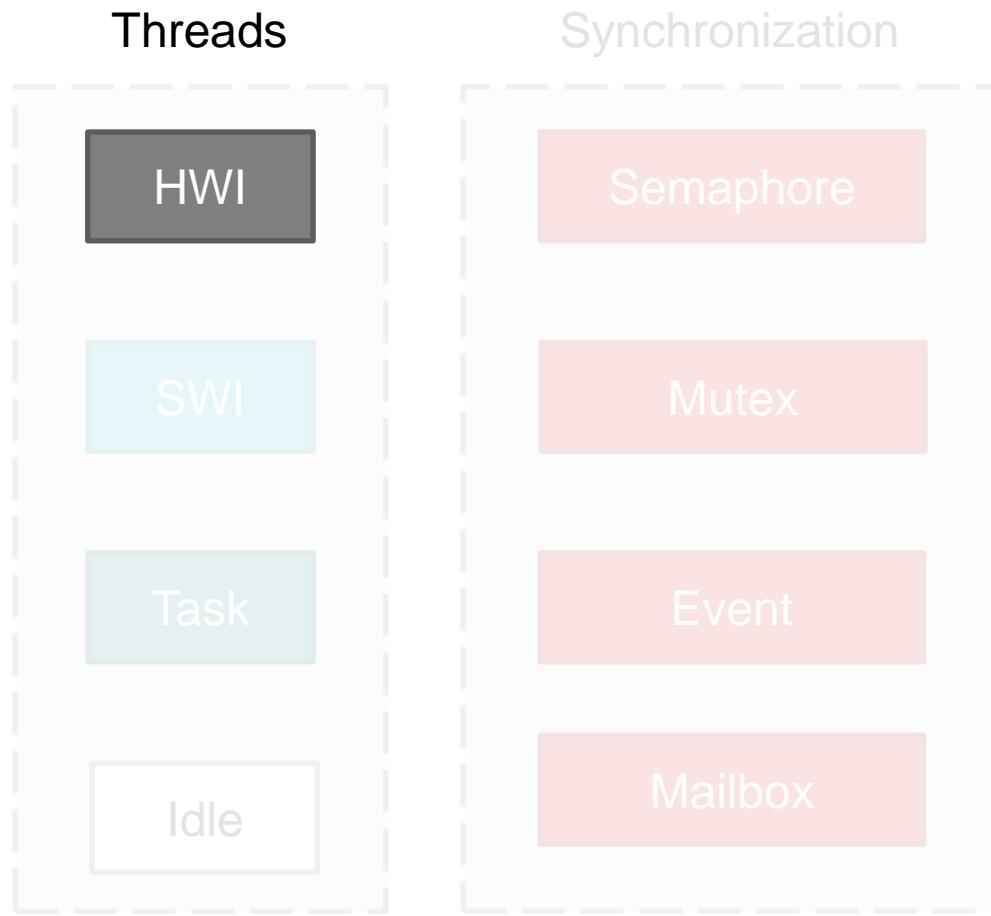
The RTOS scheduler



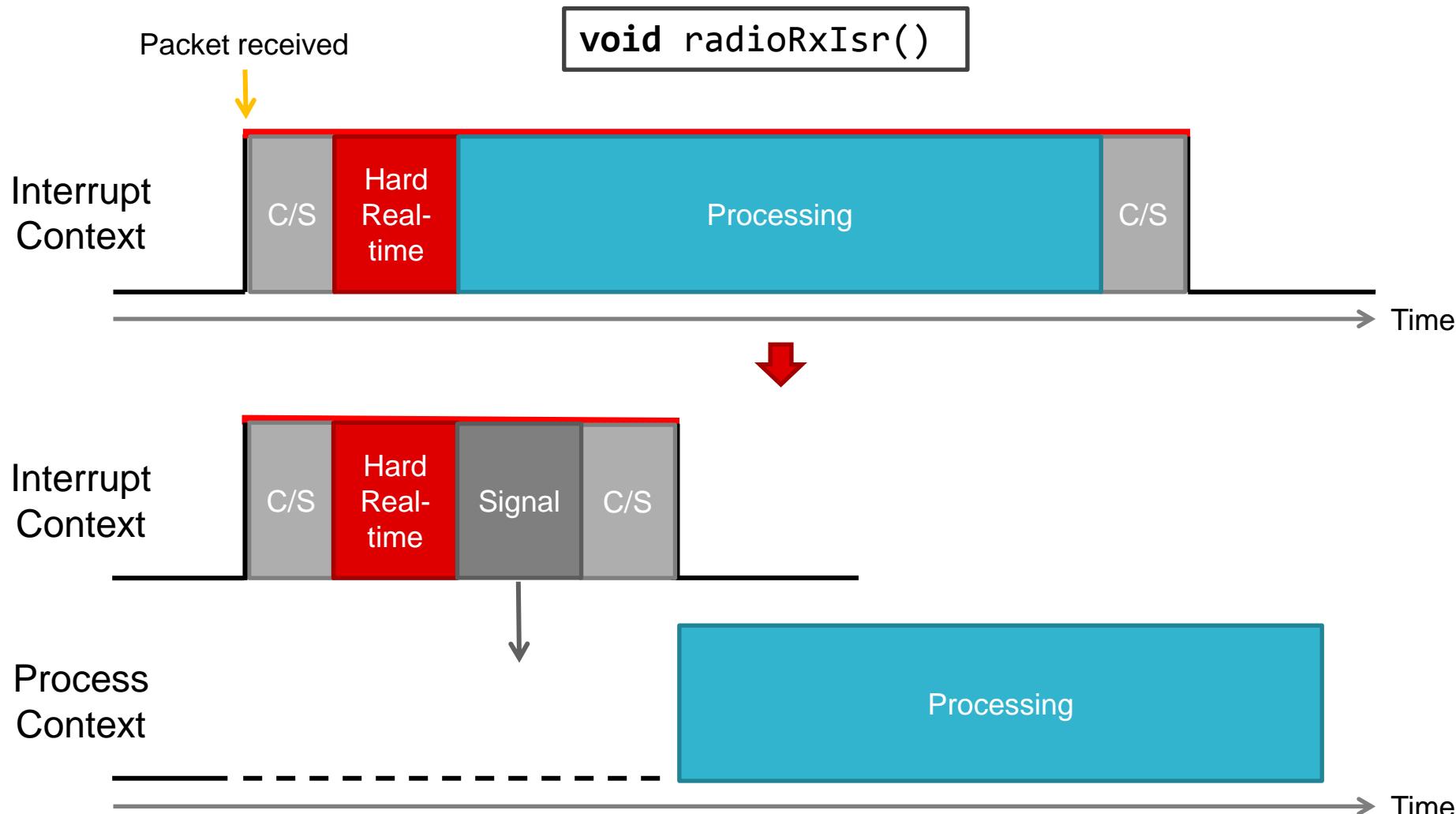
The RTOS scheduler



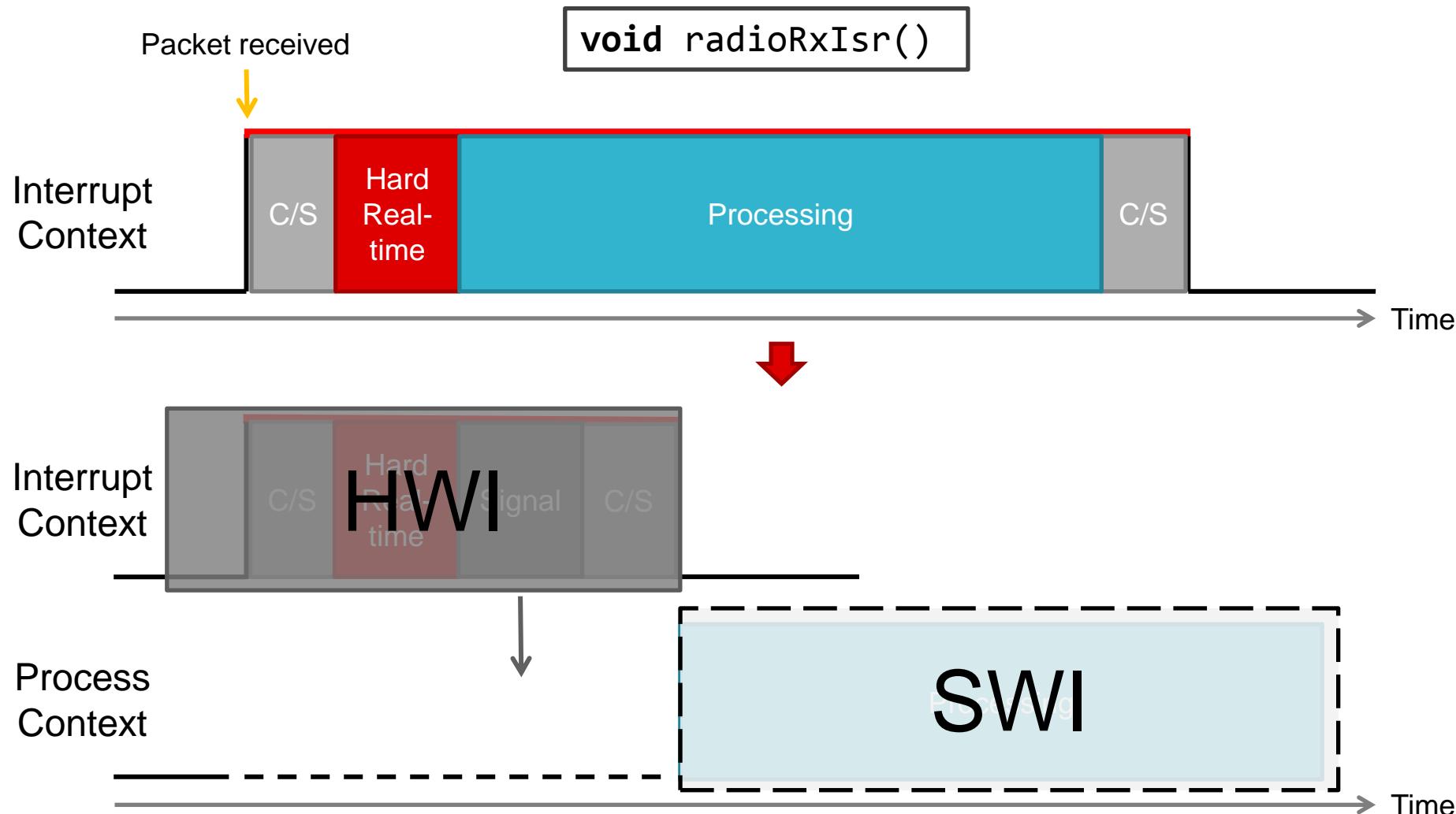
TI RTOS – HWI



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

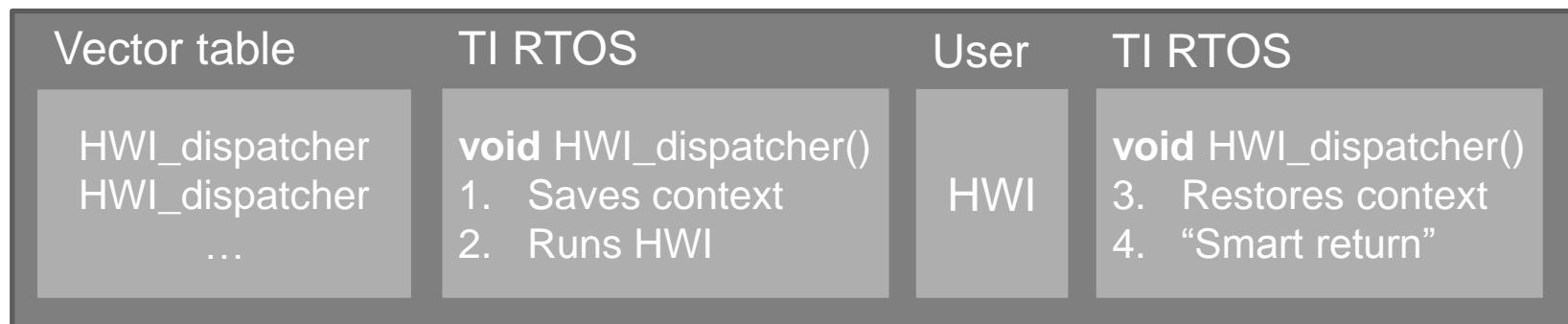
HWI – HWI dispatcher

- **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

Interrupt



Interrupt Context



(↓ Post SWI)



“Smart return”

TI RTOS – Smart Thermostat

Looking back

HWI

```
void buttonIsr()
```

SWI

"High Priority"

```
void processButton()
```

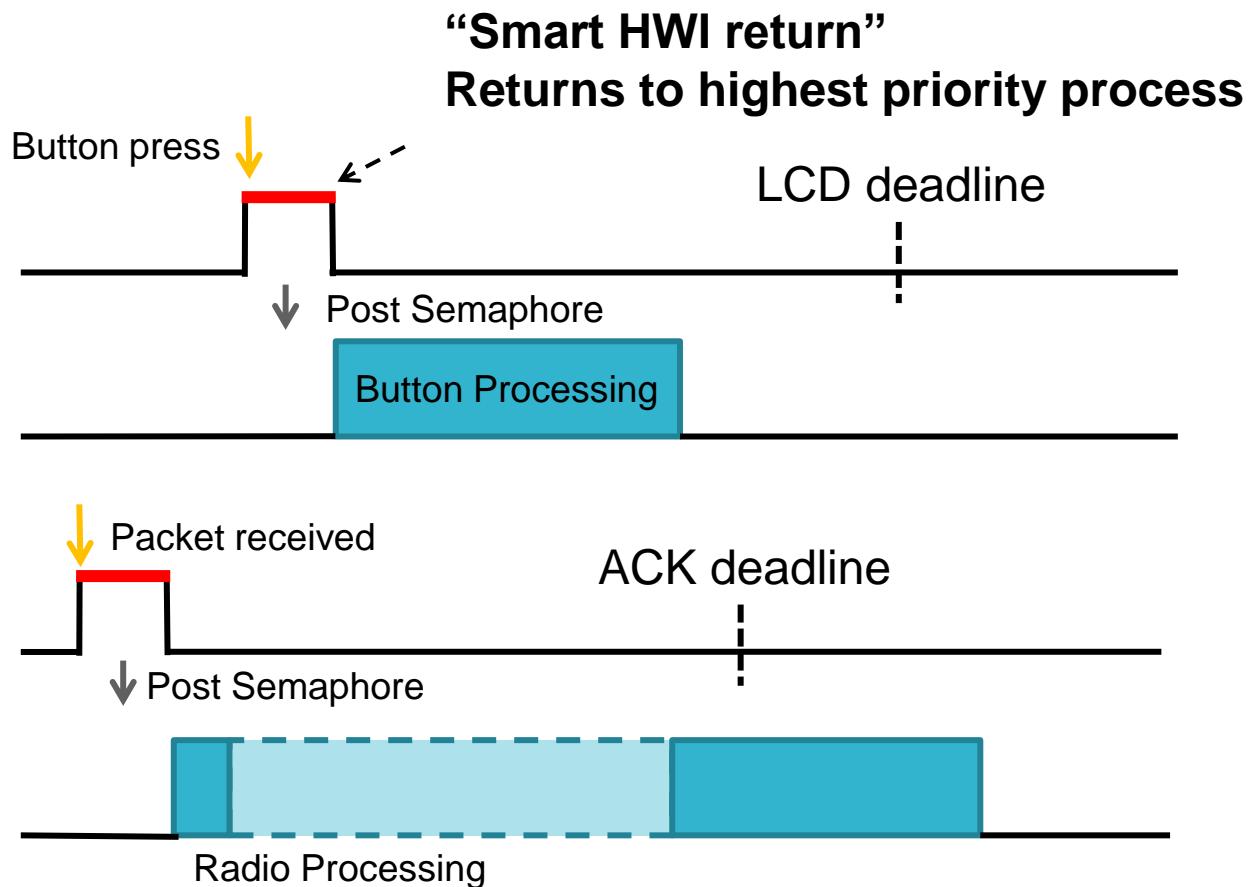
HWI

```
void radioRxIsr()
```

SWI

"Low Priority"

```
void processPacketRx()
```

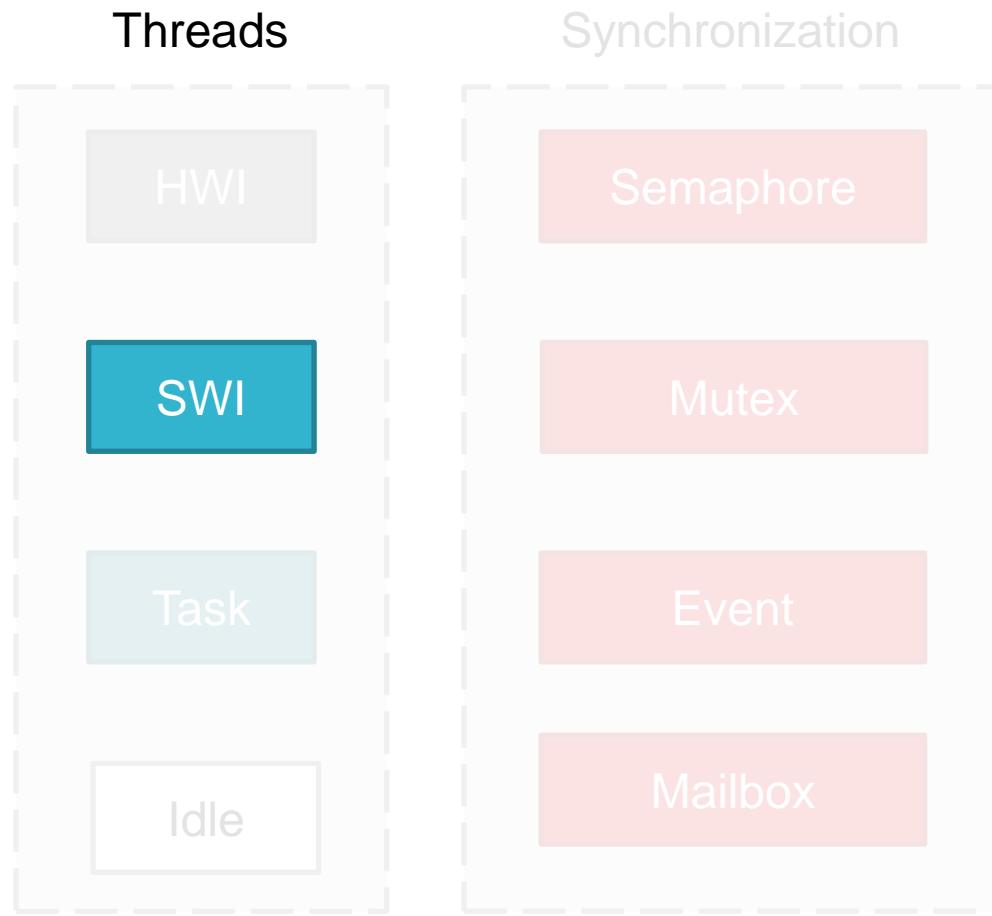


HWI – Hardware Interrupt

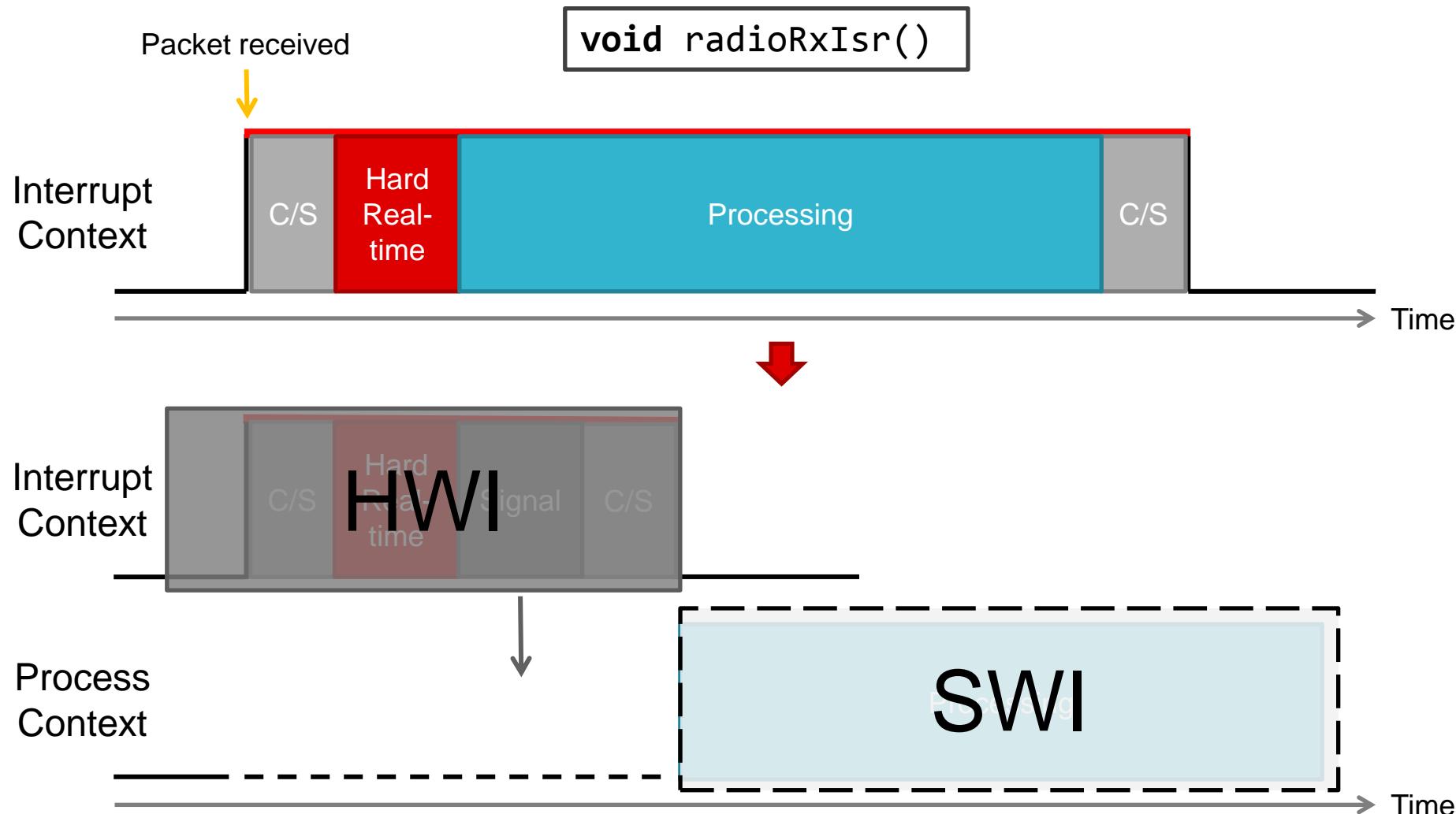
- Using regular interrupt handlers
- **Not recommended**
 - Cannot use any RTOS calls
- Must **explicitly** save and restore the RTOS context
- Must manually handle nesting
- Should only be used for extremely low latency ISR:s

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

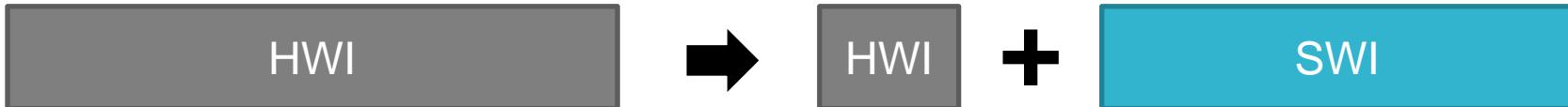
TI RTOS – SWI



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

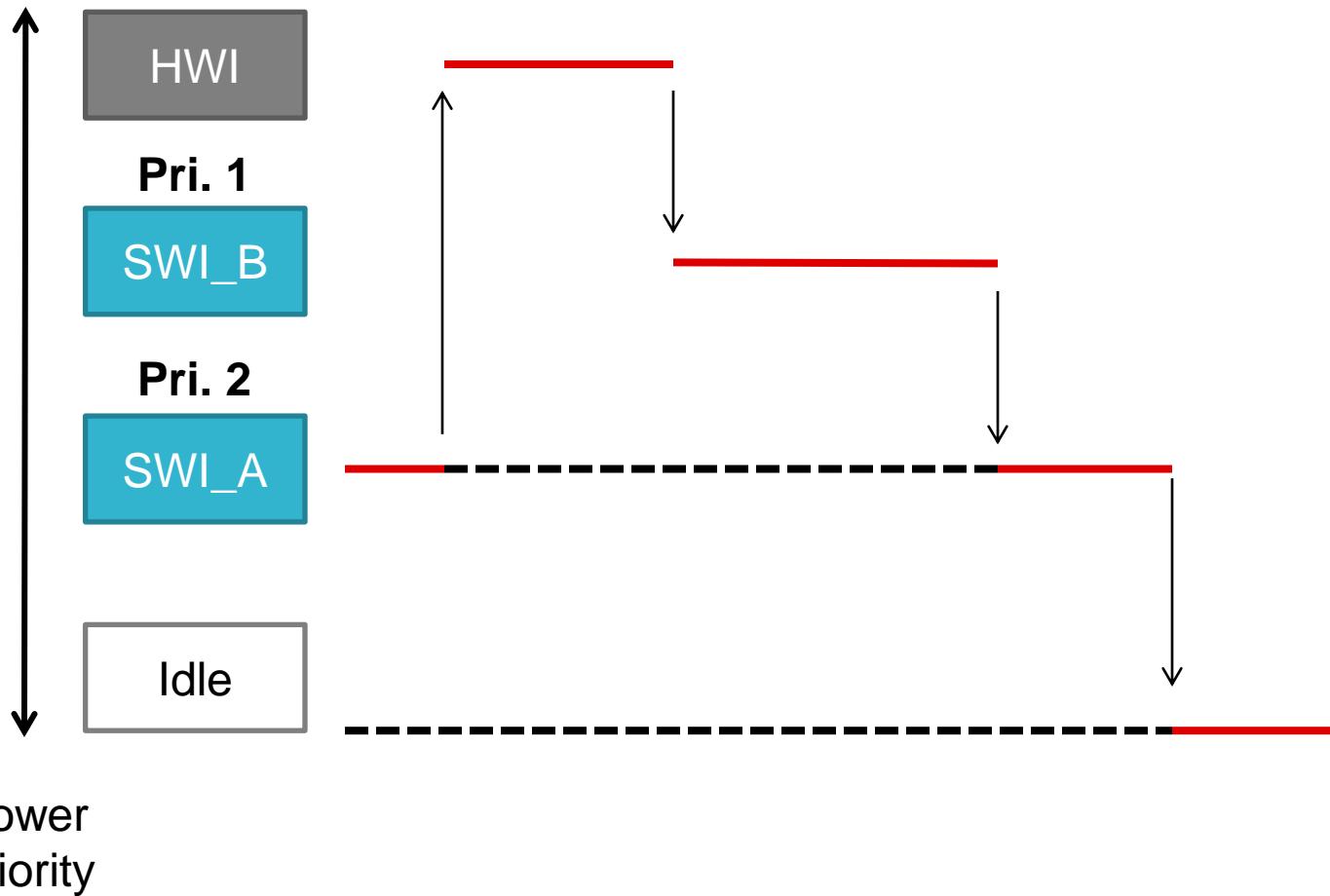
SWI states

- **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

Higher priority

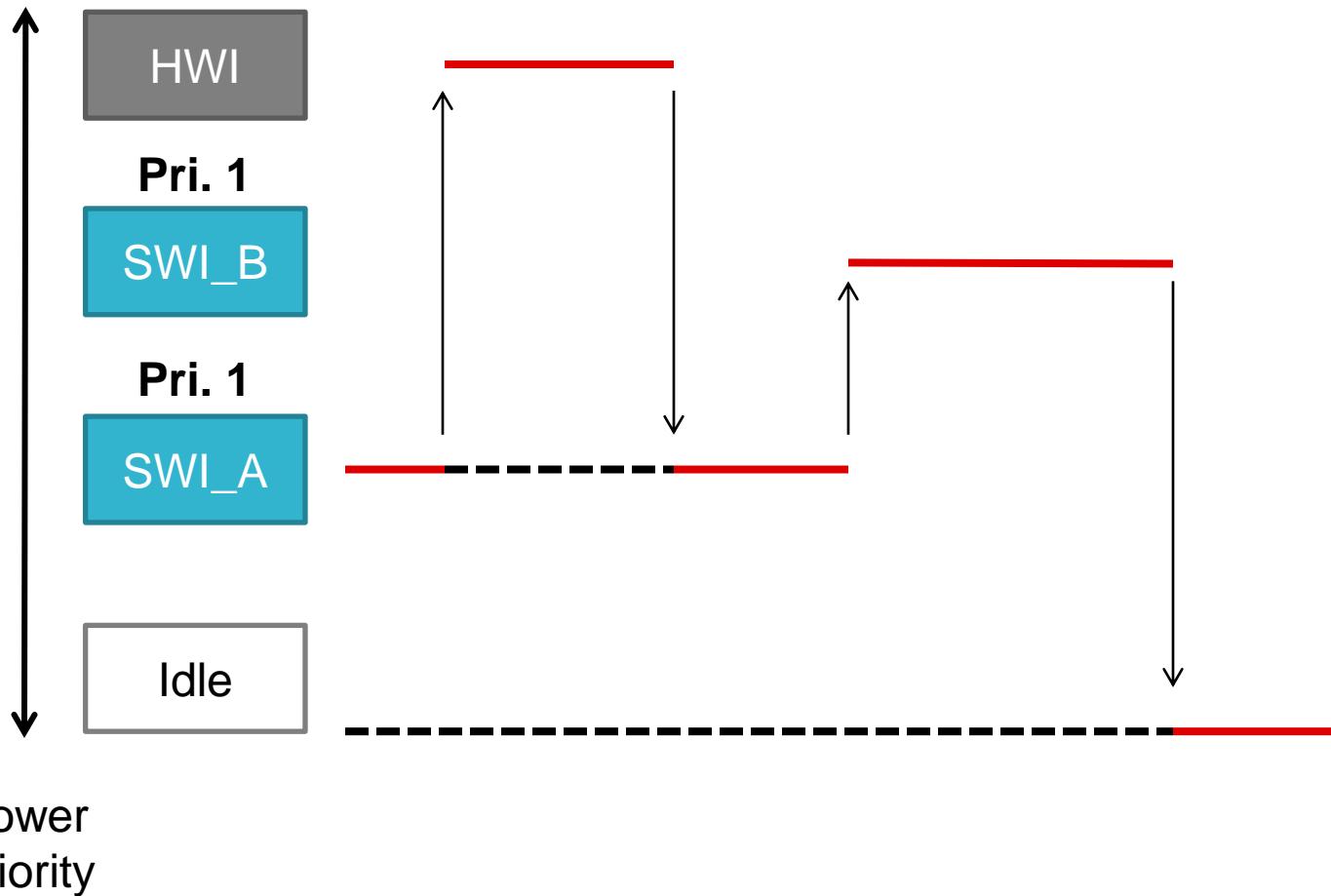
Different priorities – Executed in order of priority



SWI – Software Interrupt

Higher priority

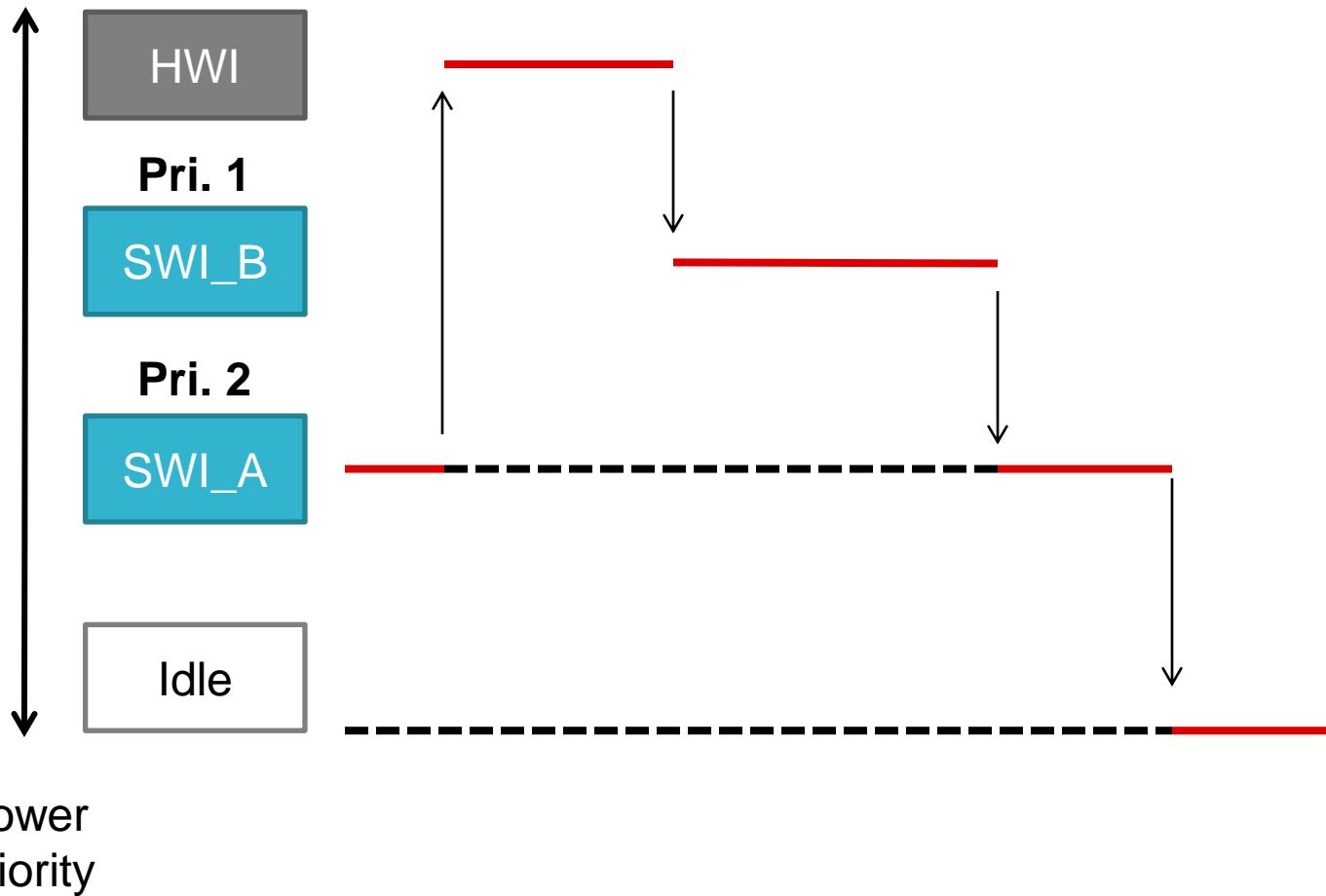
Same priorities – Executed First In, First Out, FIFO



SWI – Software Interrupt

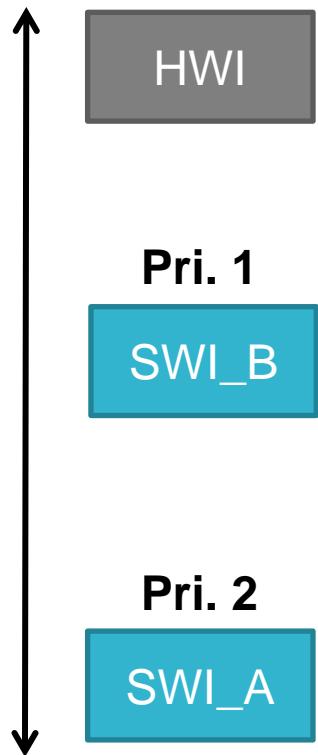
Higher
priority

How does this work?



SWI – Software Interrupt

Higher priority



`Swi_post(swi_b);`

“Smart HWI return”
Returns to highest priority process

“Smart SWI return”
Returns to highest priority process

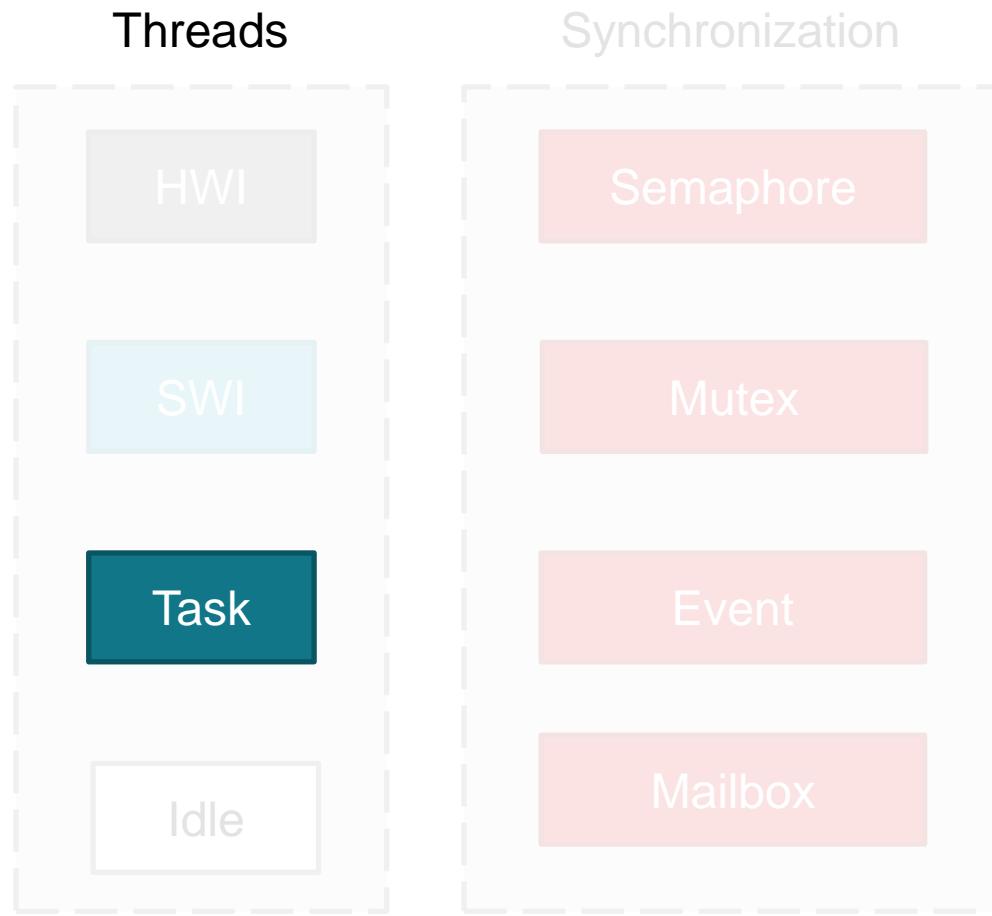
Lower priority

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 overview

Higher priority

Threads



HWI

- Handles hardware interrupts
- Priorities set in hardware

SWI

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

Task

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

Idle

- Special thread – Runs when nothing else is running

Lower priority

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

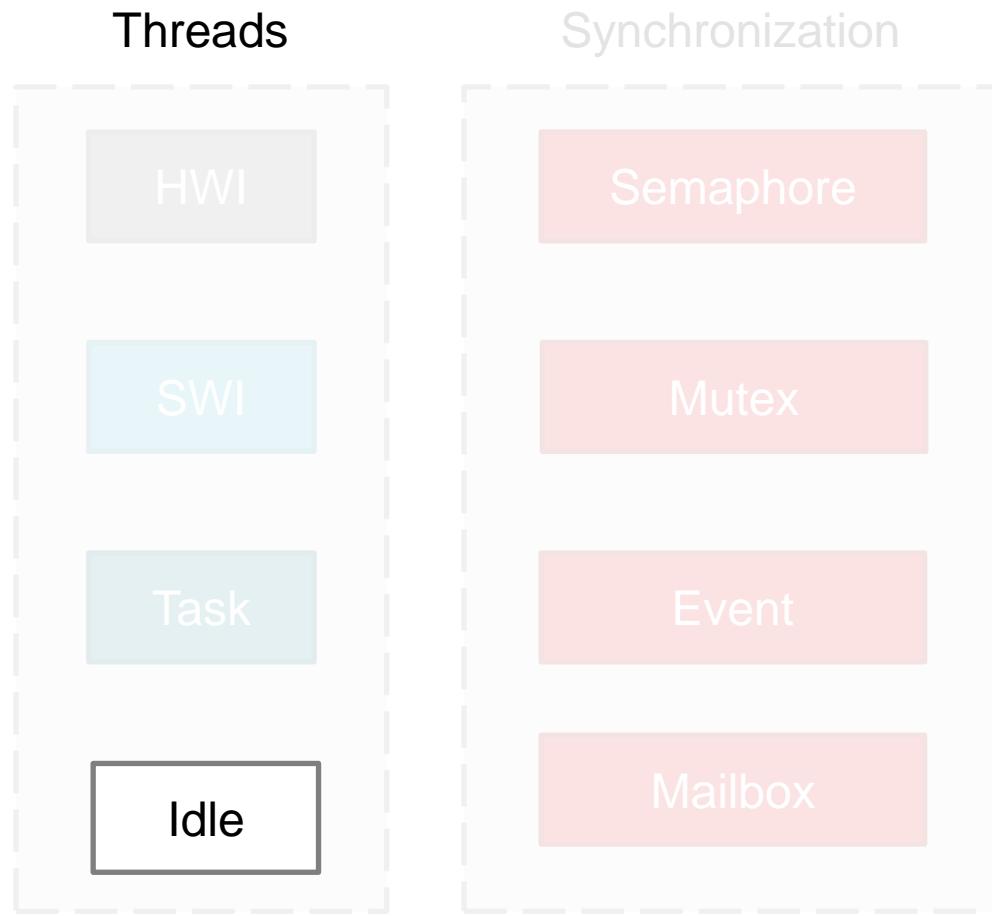
- Runs as soon as the system is started, or the task is created.
- Generally never exists, although it can

```
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

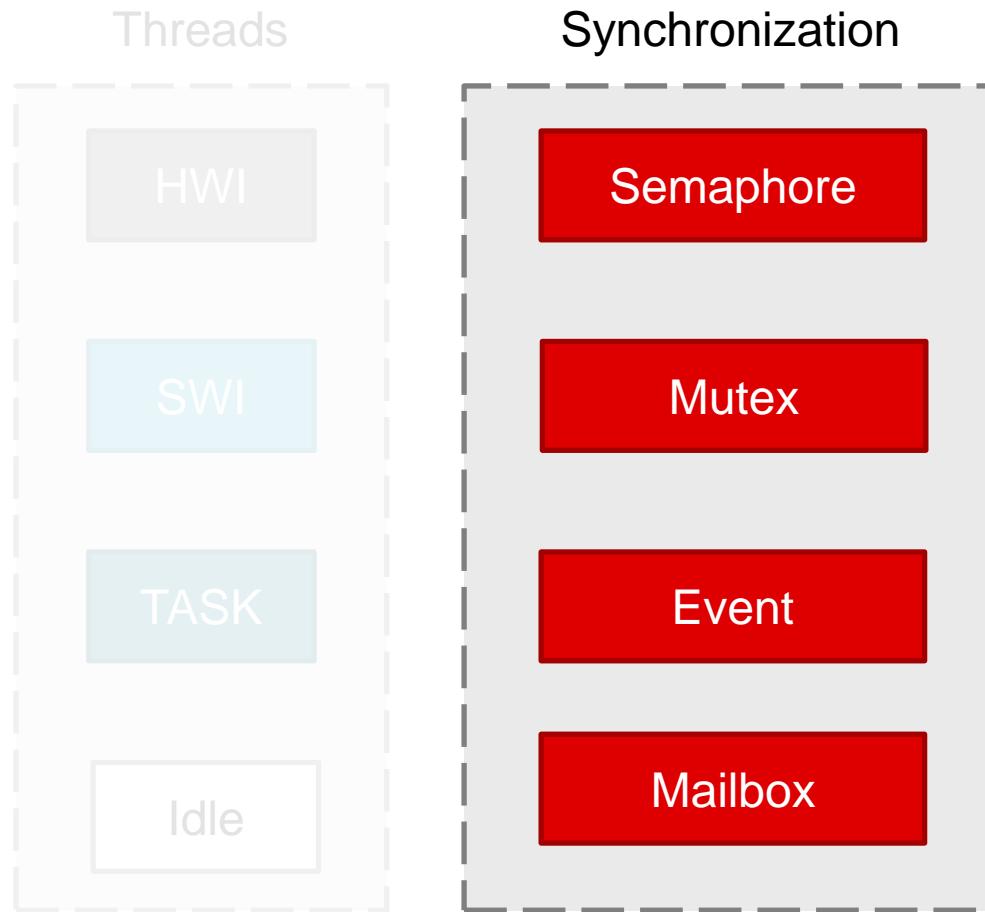


Idle thread

- 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



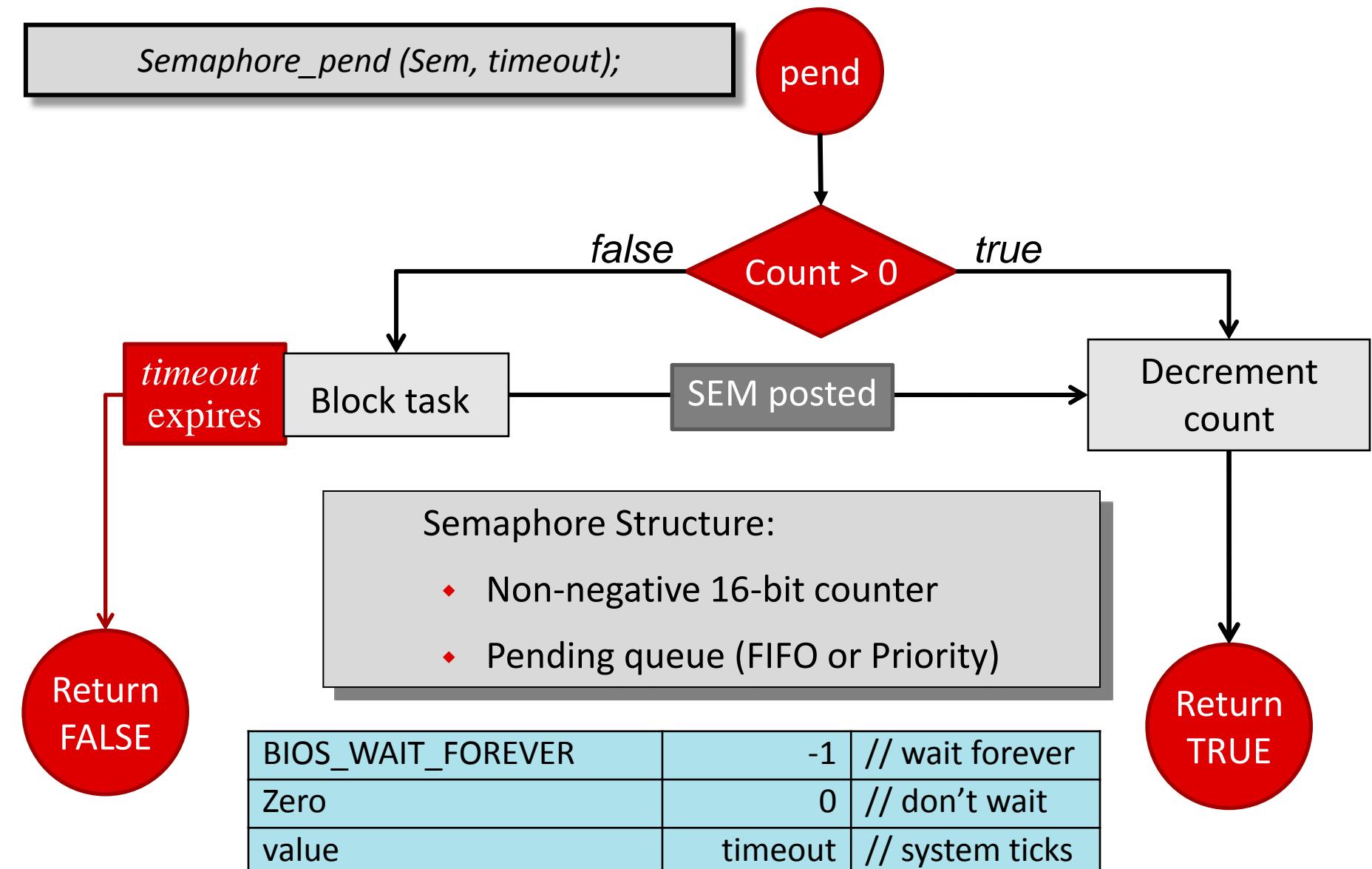
Semaphore

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

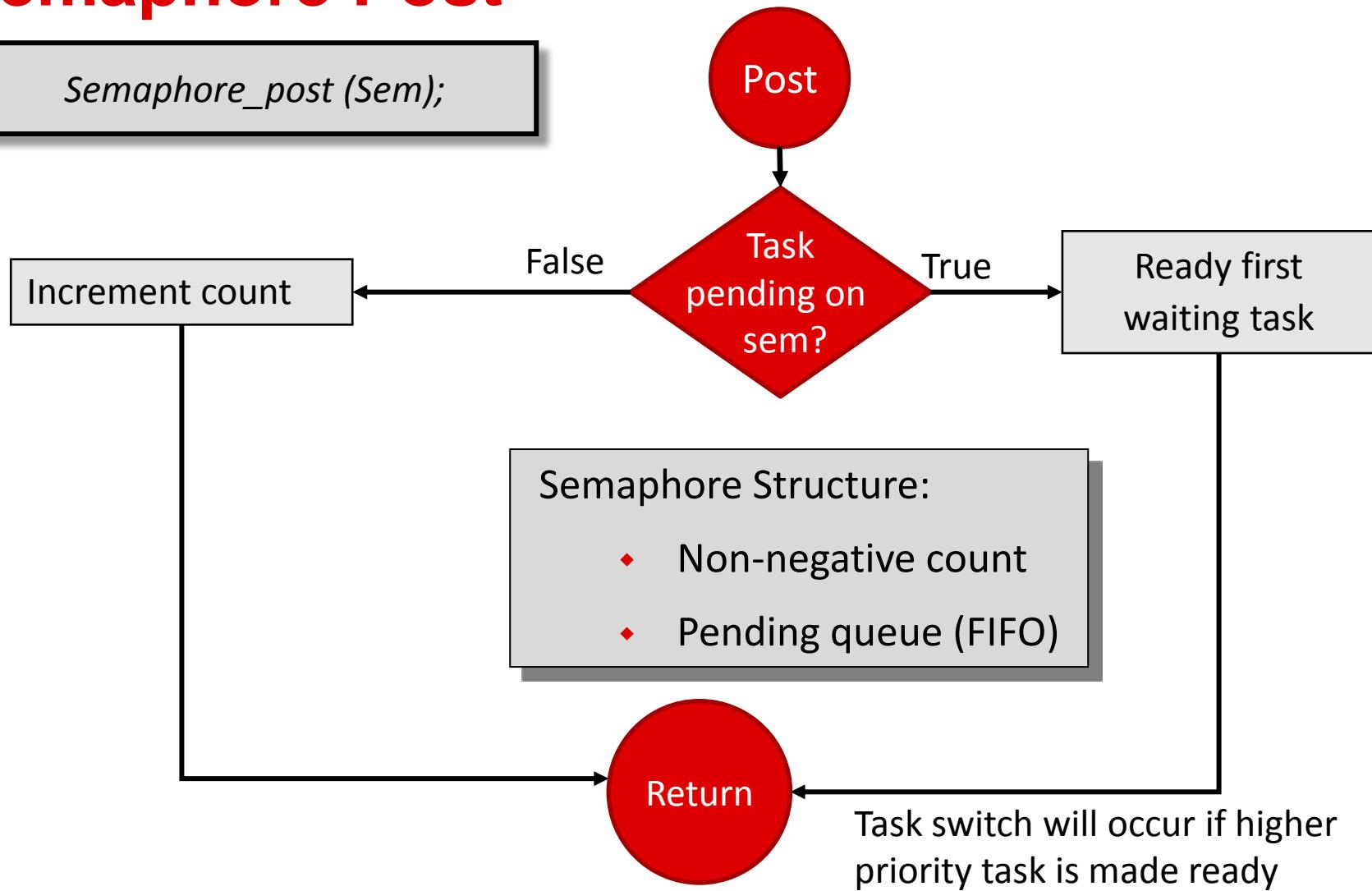
Semaphore Pend

```
Semaphore_pend (Sem, timeout);
```



Semaphore Post

```
Semaphore_post (Sem);
```



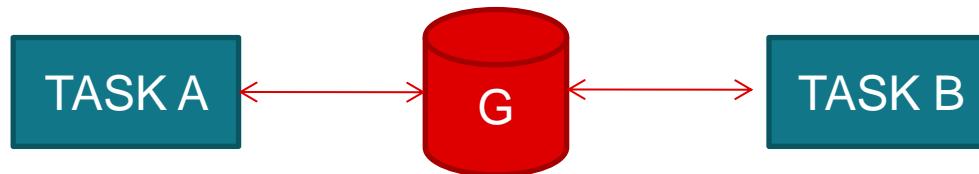
Semaphore

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

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

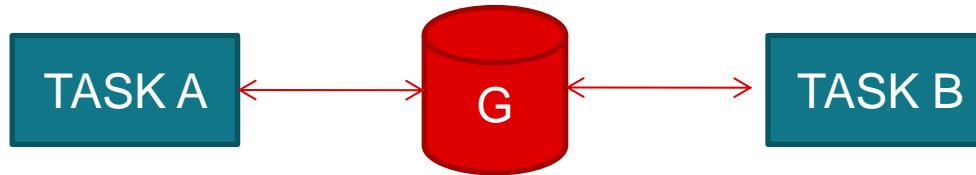
Shared resource - Global variables

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

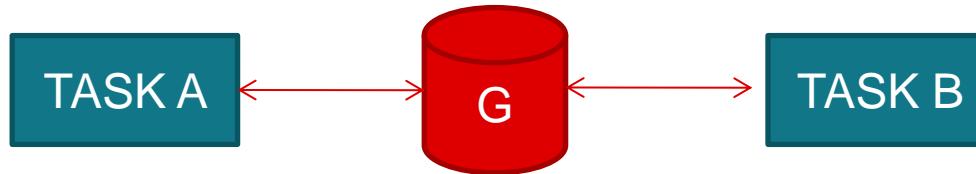
```
void hiPriTask() {  
    ...  
    cnt += 1;  
    ...  
}
```

TASK B

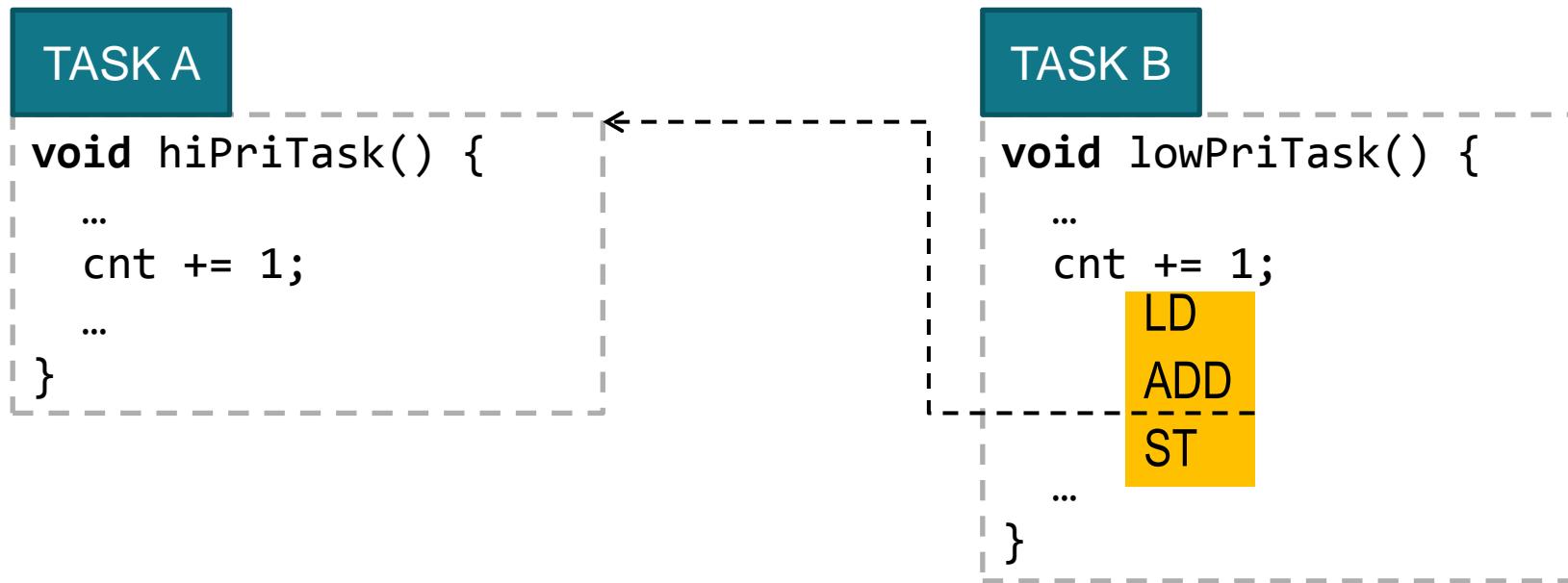
```
void lowPriTask() {  
    ...  
    cnt += 1;  
    ...  
}
```

Shared resource - Global variables

We have two Tasks and a shared global resource:



Worst case:



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!

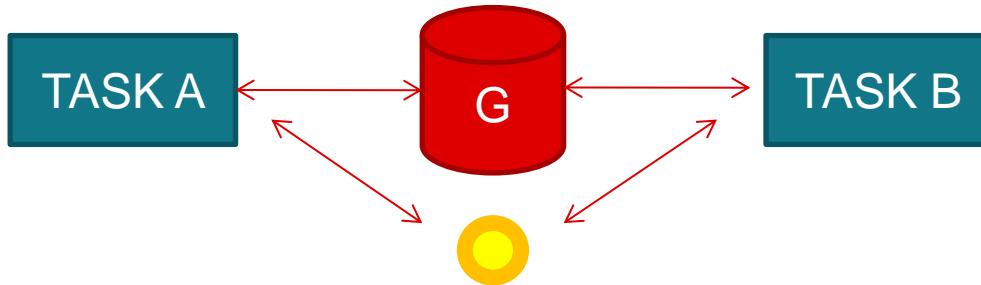
Mutex

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:

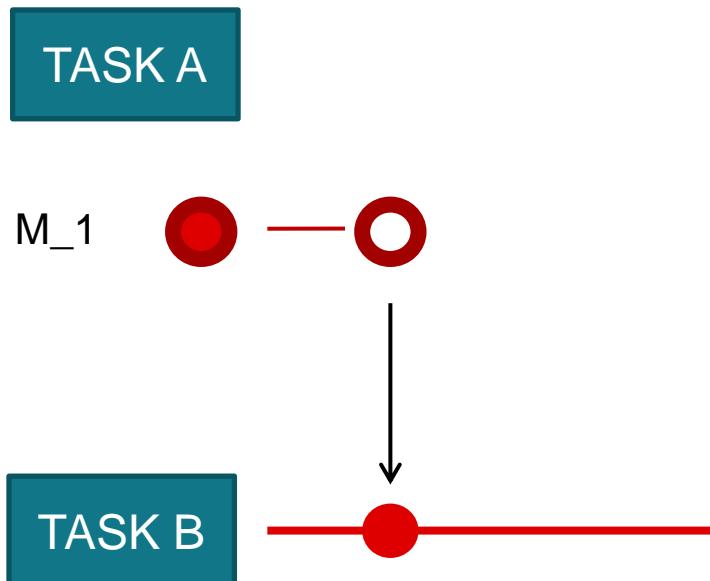
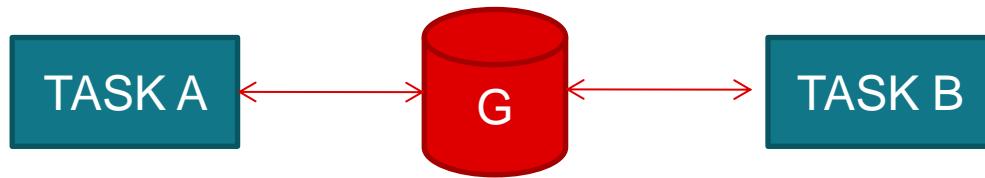


TASK B

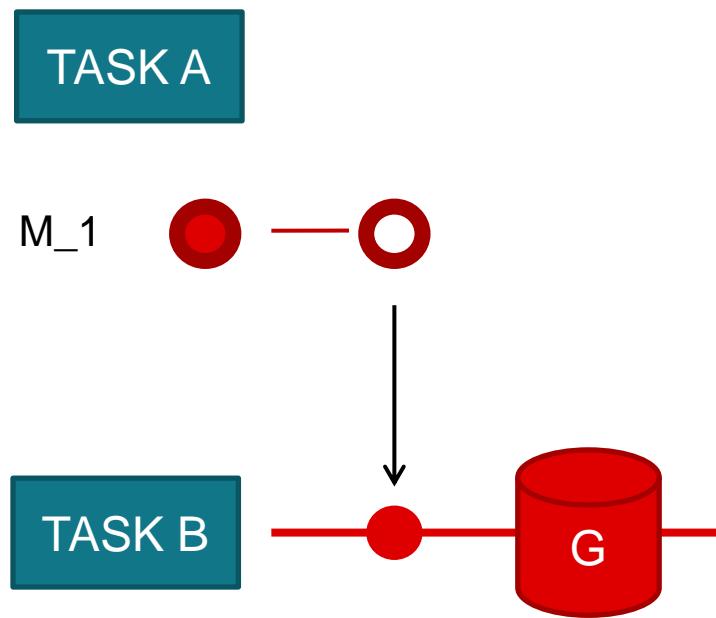
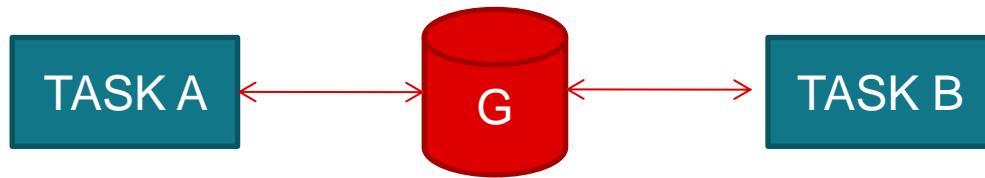
TASK A

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

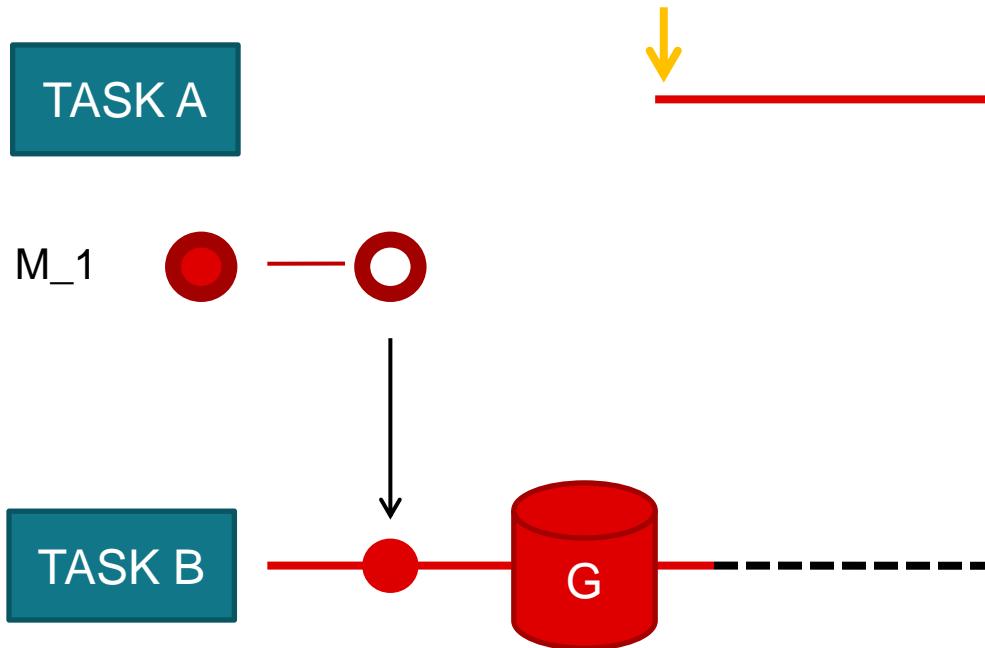
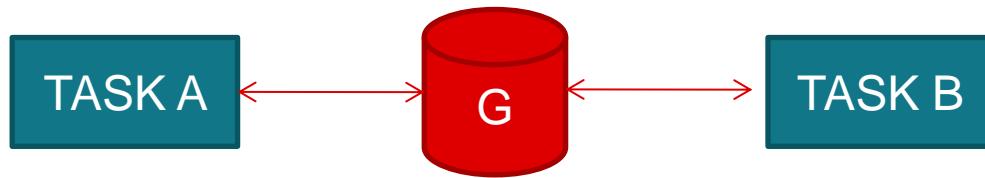
Global variables with Mutex



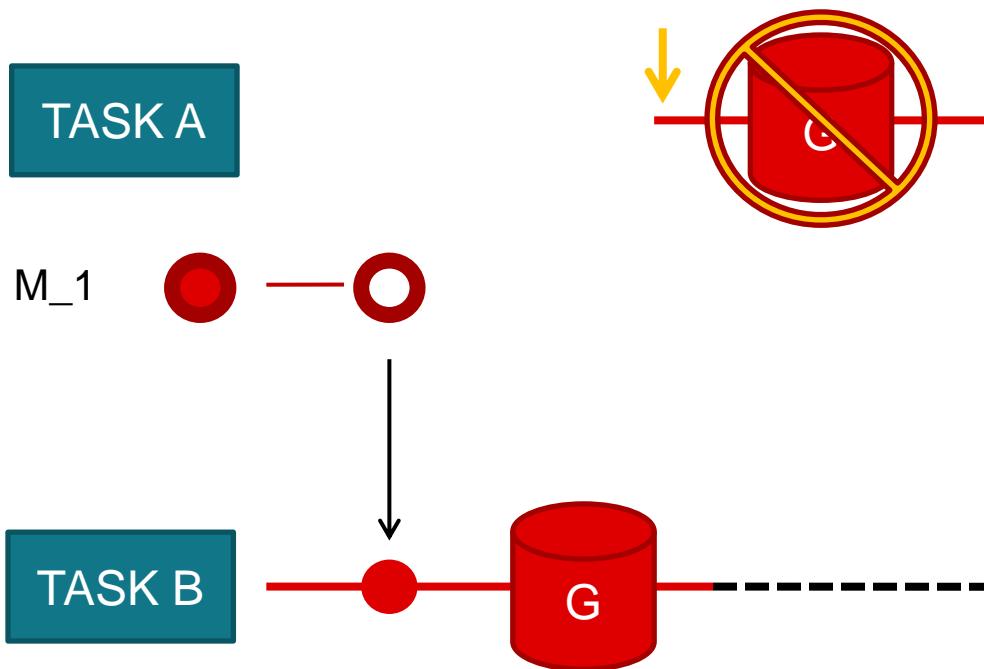
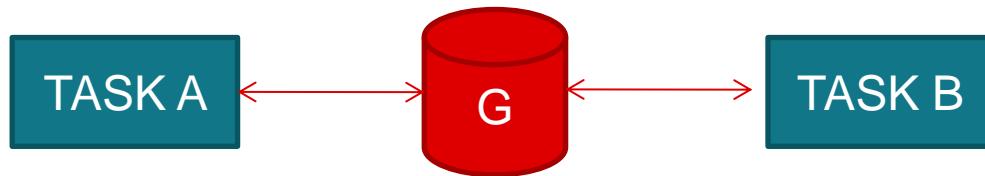
Global variables with Mutex



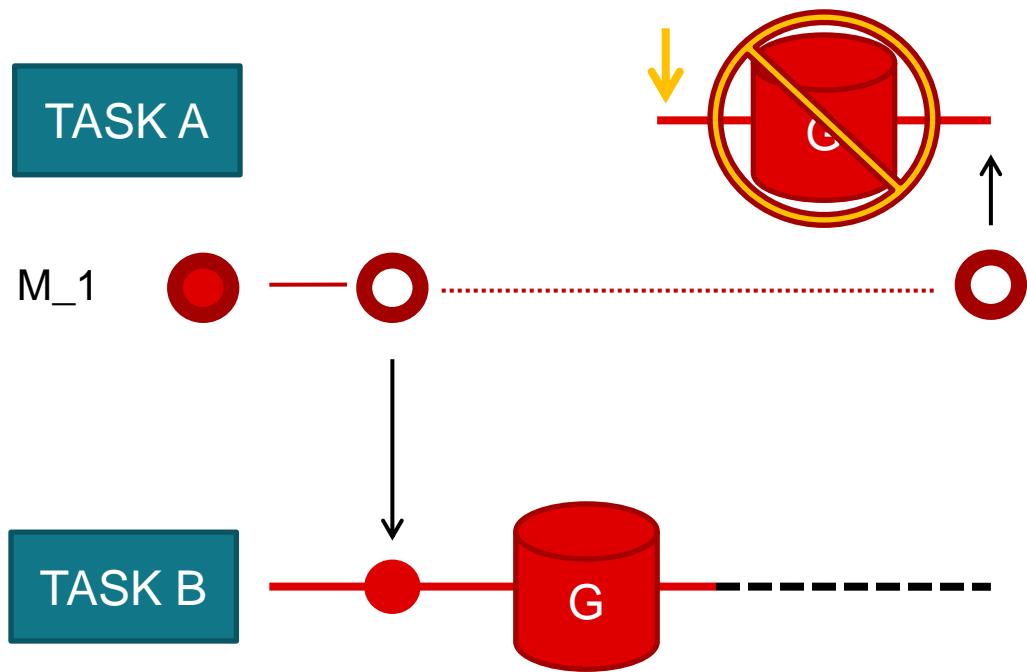
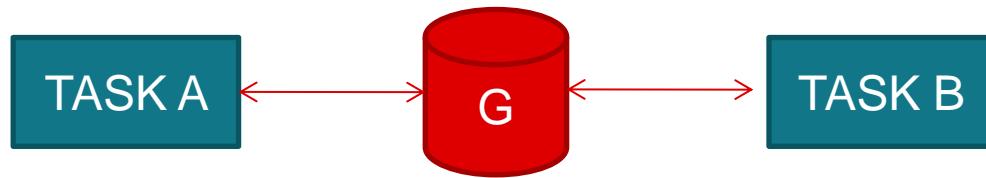
Global variables with Mutex



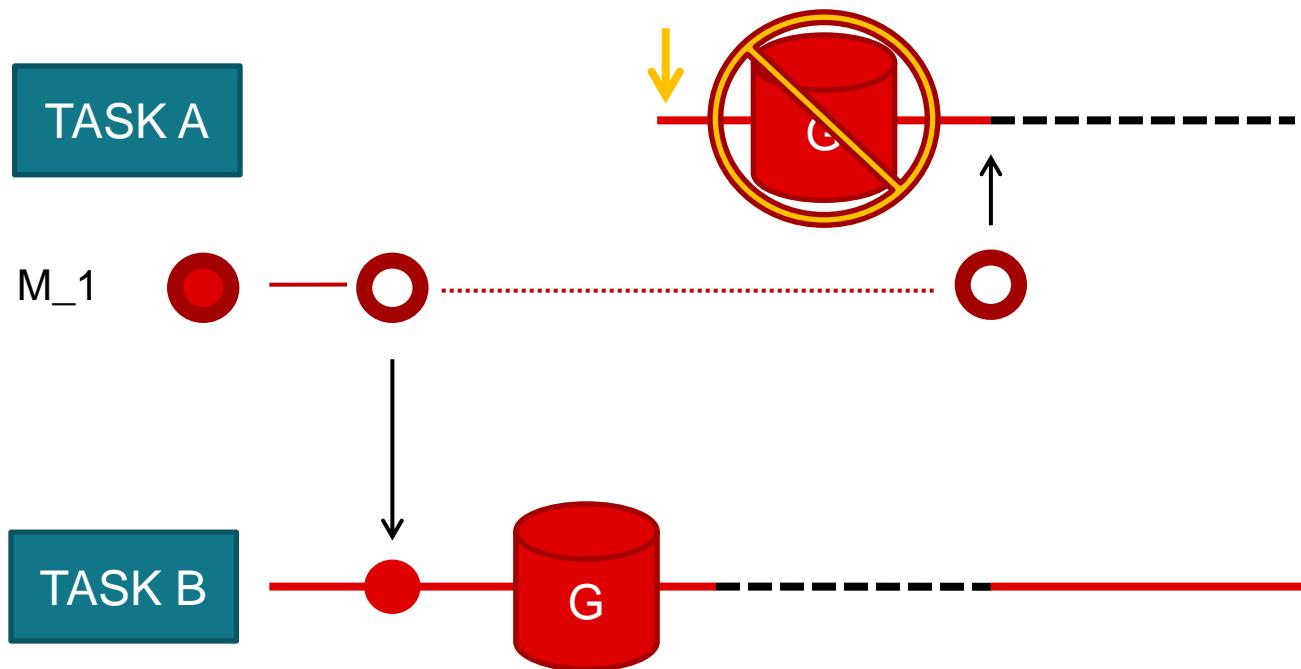
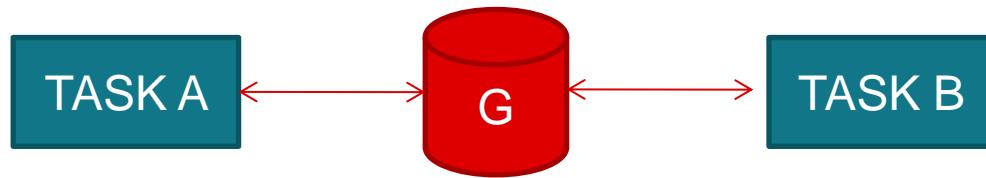
Global variables with Mutex



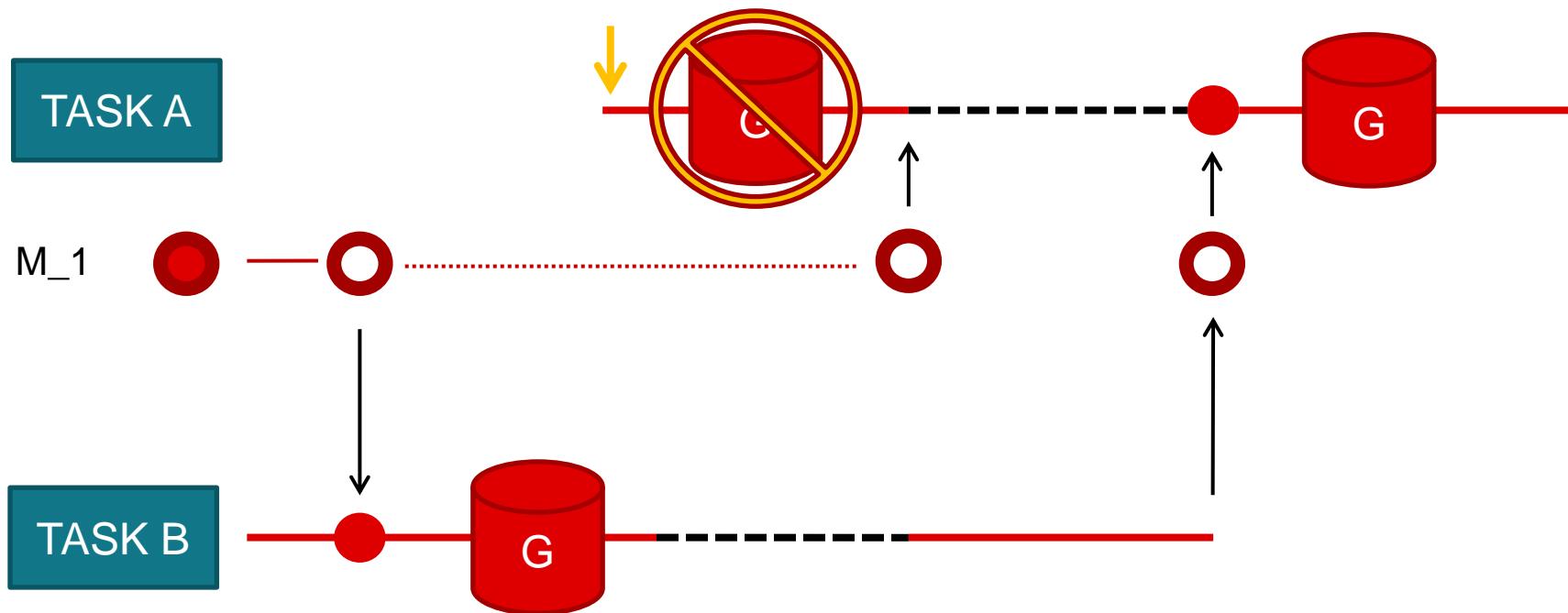
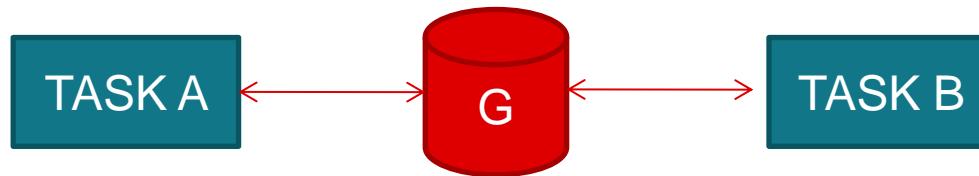
Global variables with Mutex



Global variables with Mutex

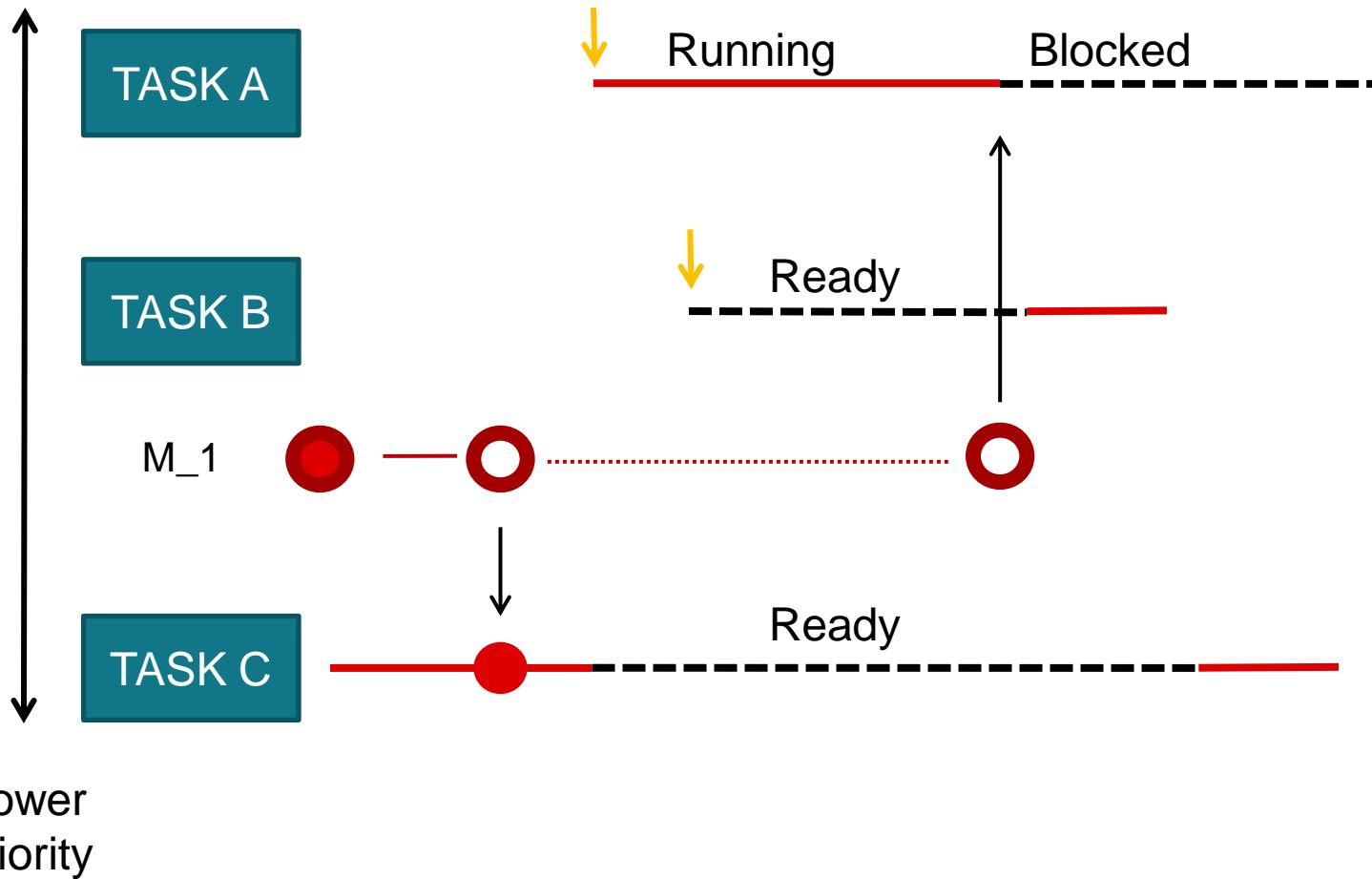


Global variables with Mutex



Task states

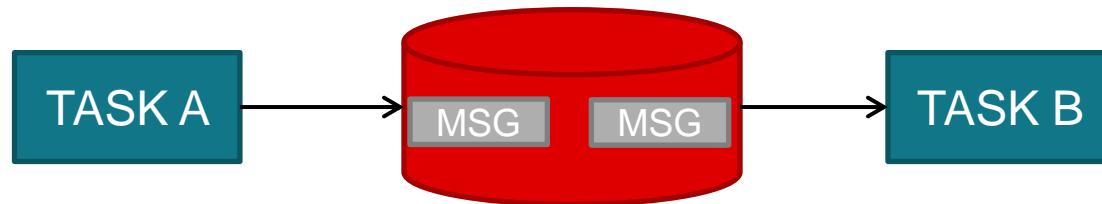
Higher priority



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

TASK A

M_1



M_2



TASK B

Deadlock

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

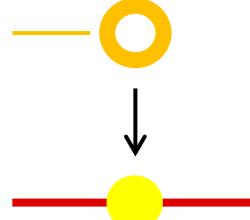
- At least two Mutexes
- Two threads of different priorities

TASK A

M_1



M_2

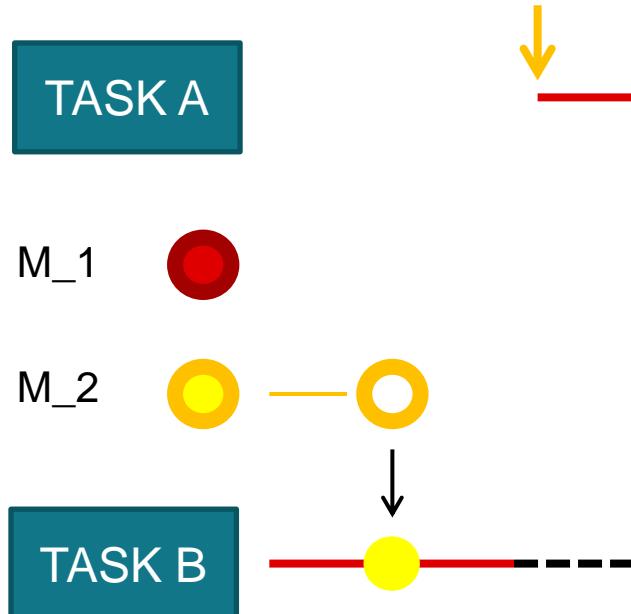


TASK B

Deadlock

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

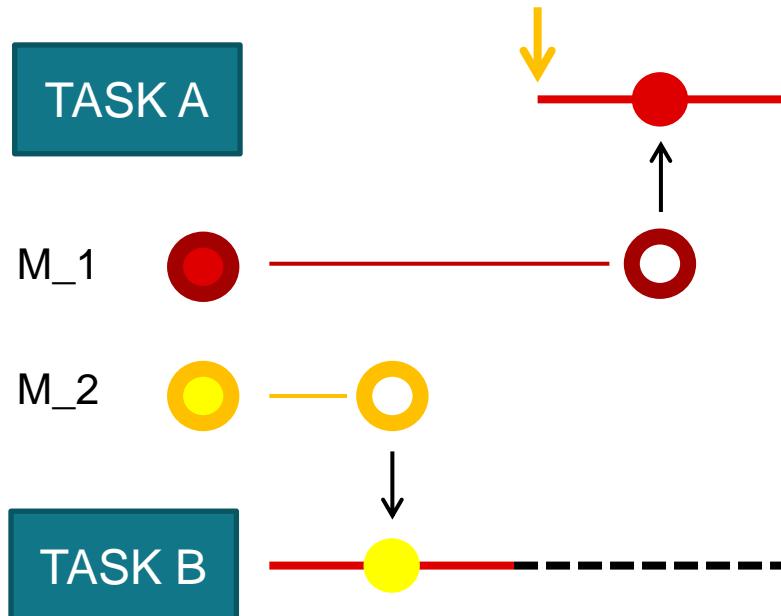
- At least two Mutexes
- Two threads of different priorities



Deadlock

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

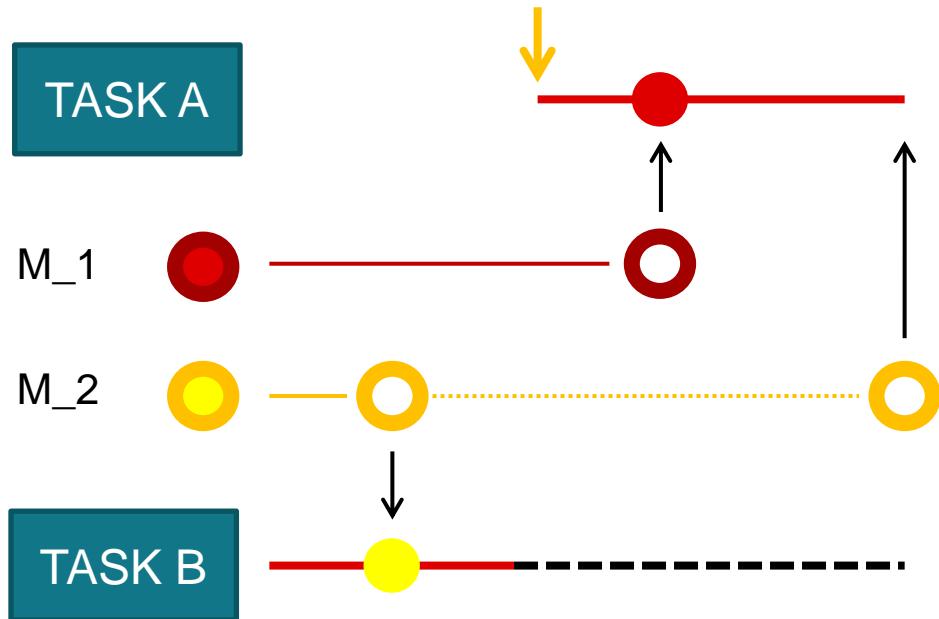
- At least two Mutexes
- Two threads of different priorities



Deadlock

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

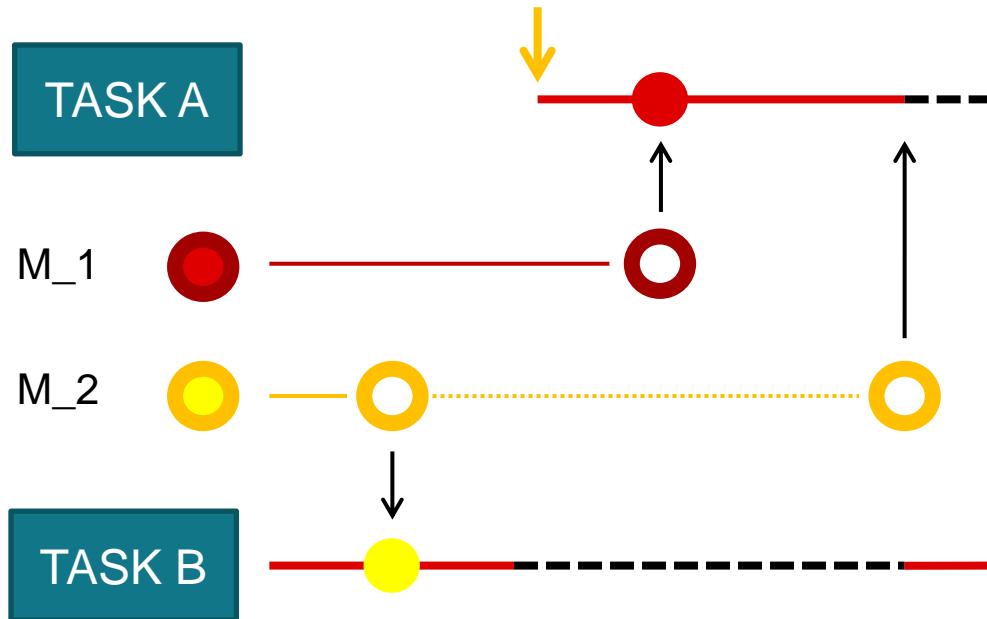
- At least two Mutexes
- Two threads of different priorities



Deadlock

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

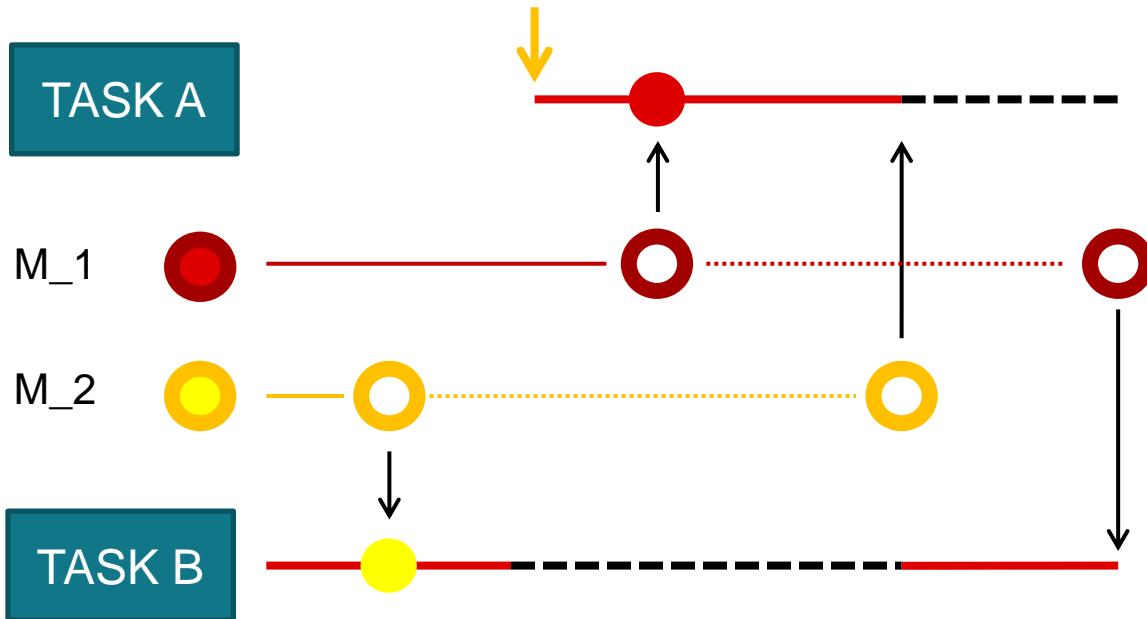
- At least two Mutexes
- Two threads of different priorities



Deadlock

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

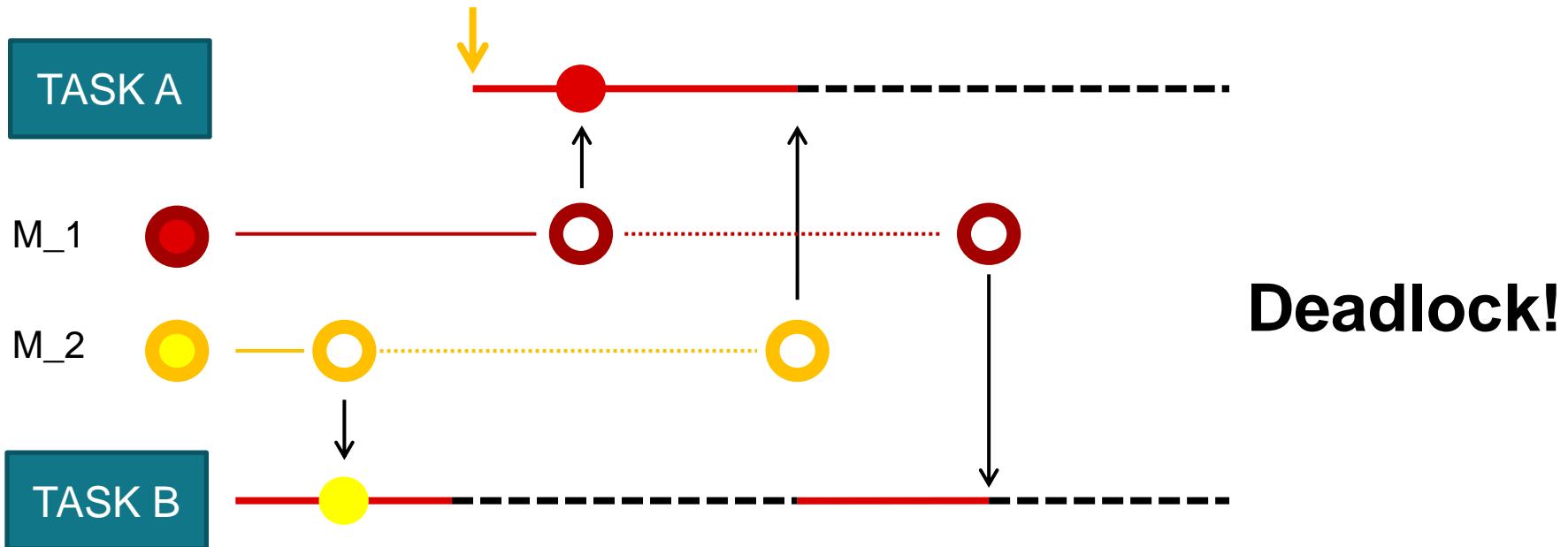
- At least two Mutexes
- Two threads of different priorities



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

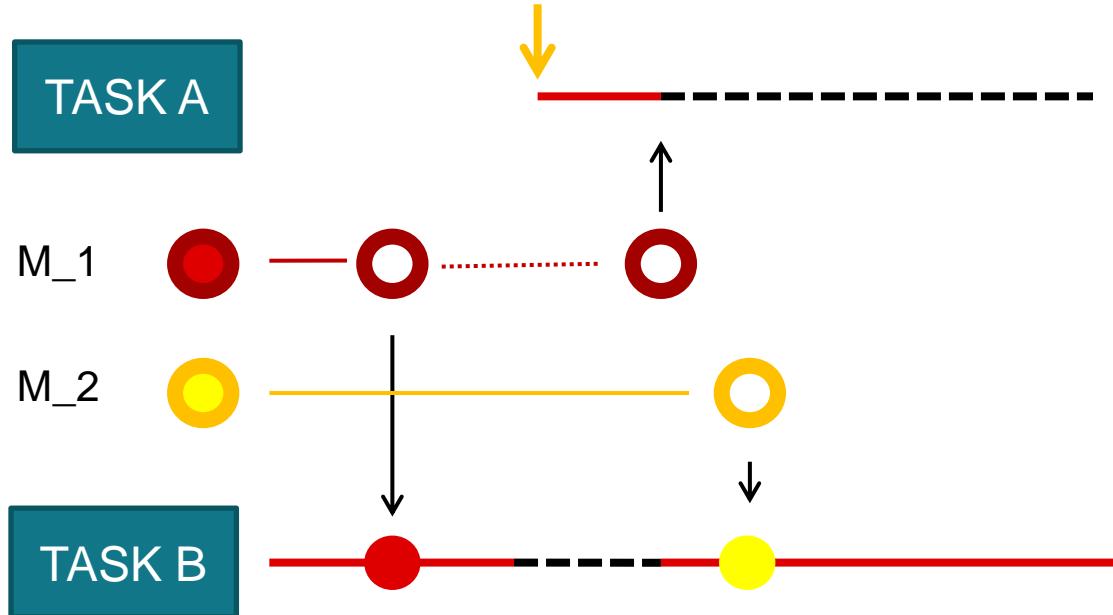
Ways to avoid deadlock:

- 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:

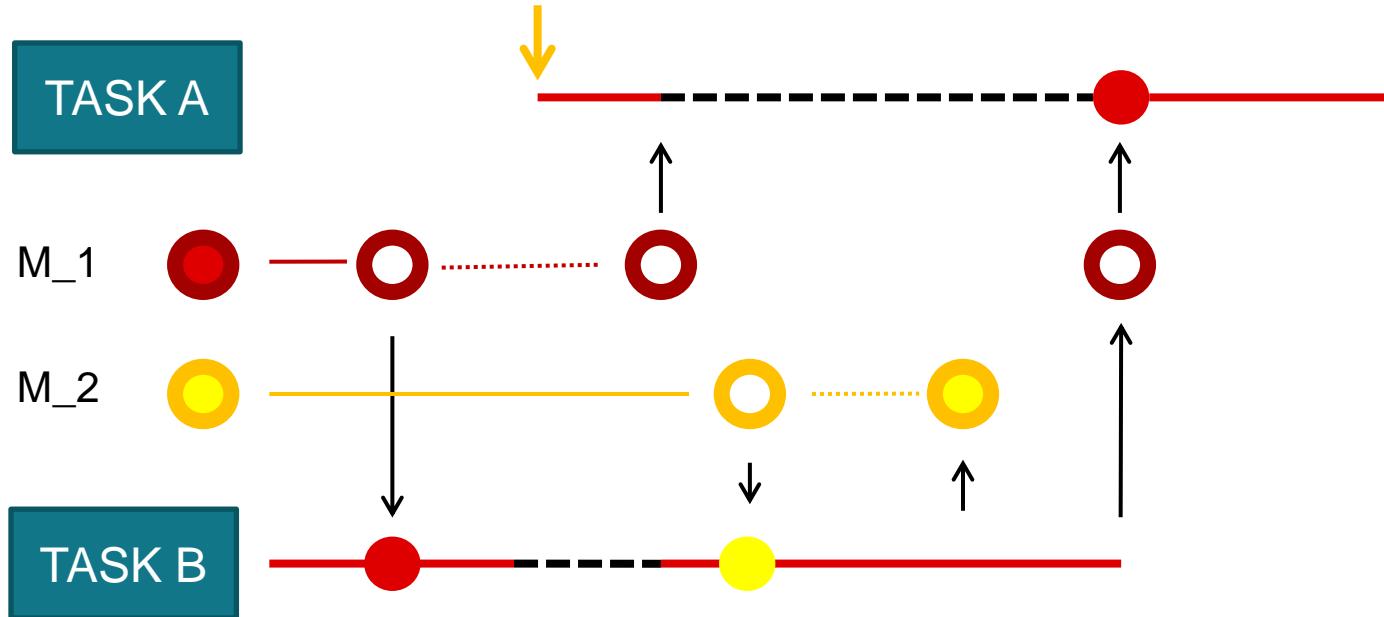
- Use one big Mutex to cover both resources
- Always pend on the Mutex in the same order in all threads



Deadlock

Ways to avoid deadlock:

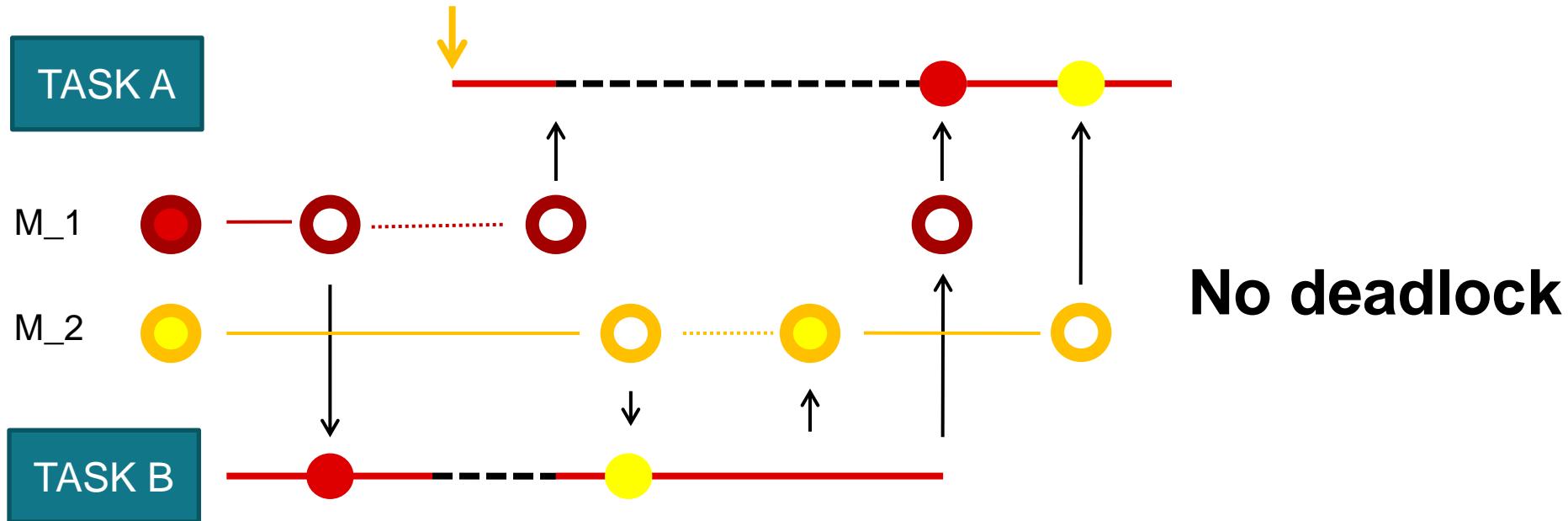
- Use one big Mutex to cover both resources
- Always pend on the Mutex in the same order in all threads



Deadlock

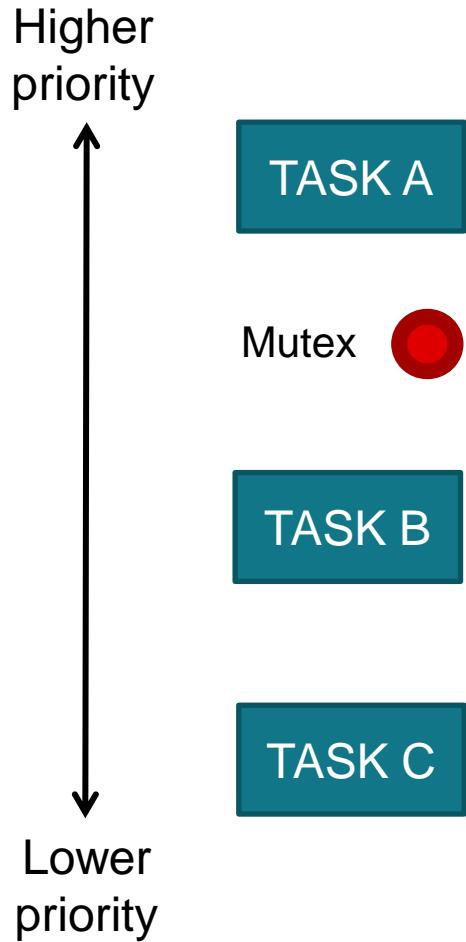
Ways to avoid deadlock:

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



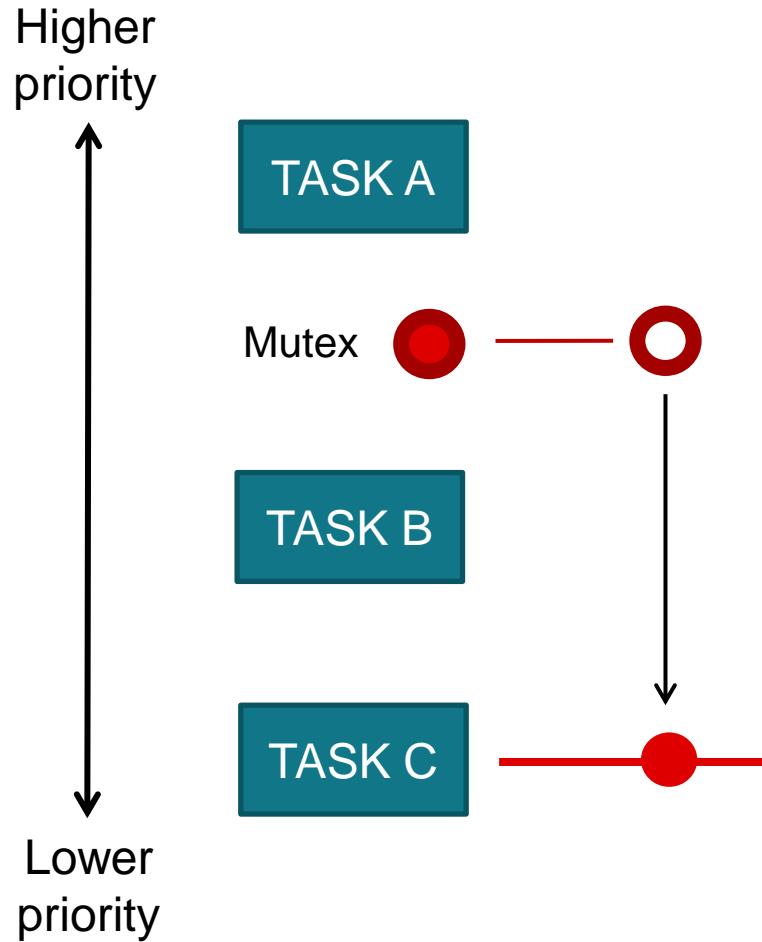
Priority Inversion

When a lower priority thread blocks a higher priority thread



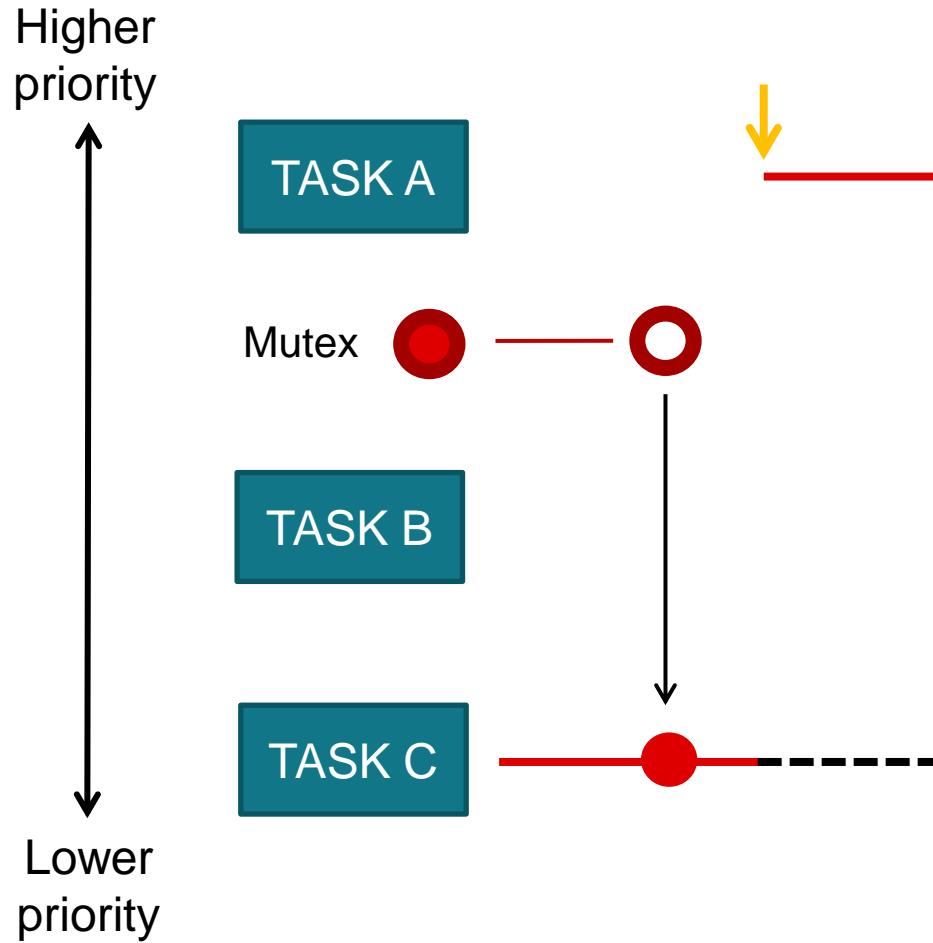
Priority Inversion

When a lower priority thread blocks a higher priority thread



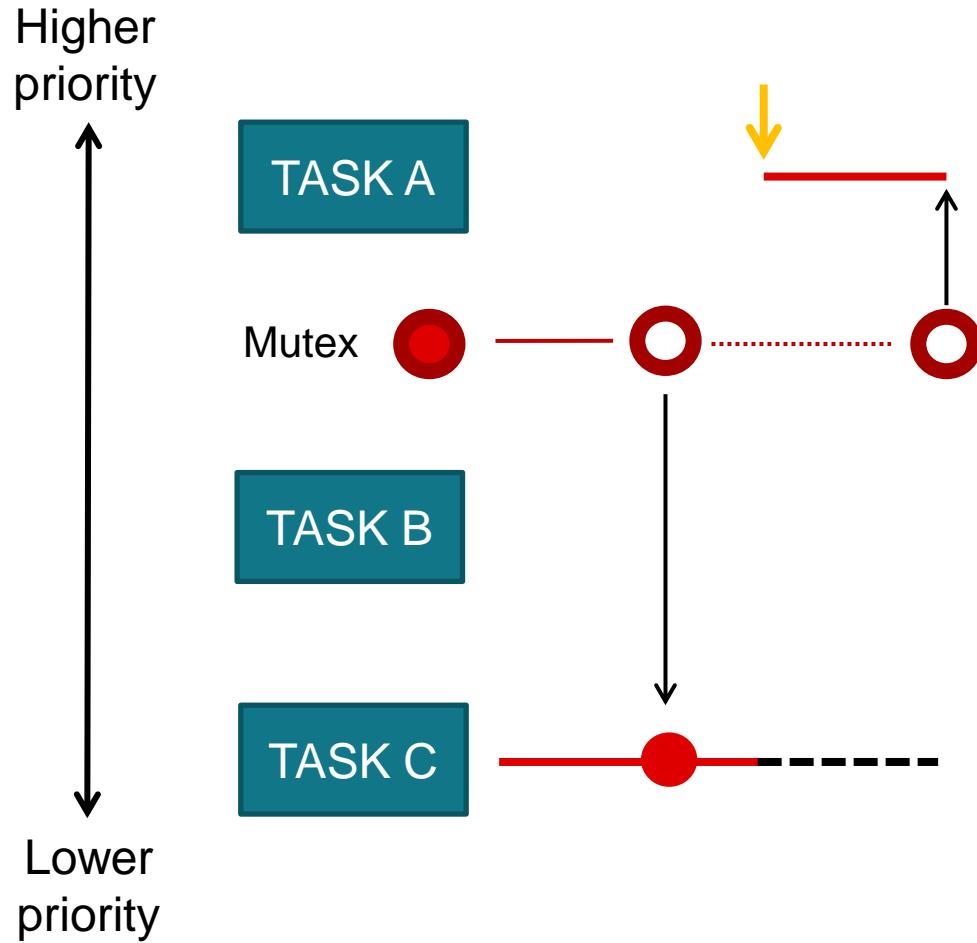
Priority Inversion

When a lower priority thread blocks a higher priority thread



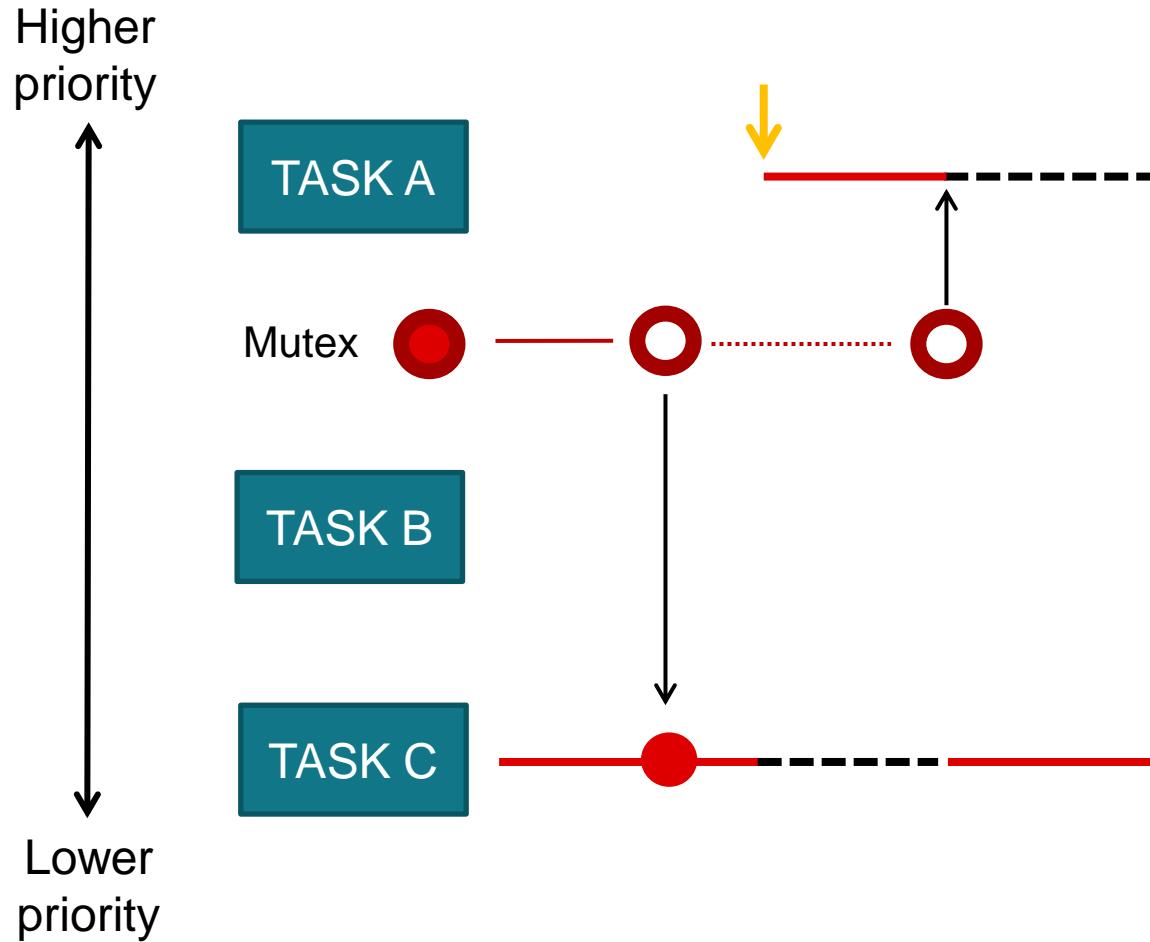
Priority Inversion

When a lower priority thread blocks a higher priority thread



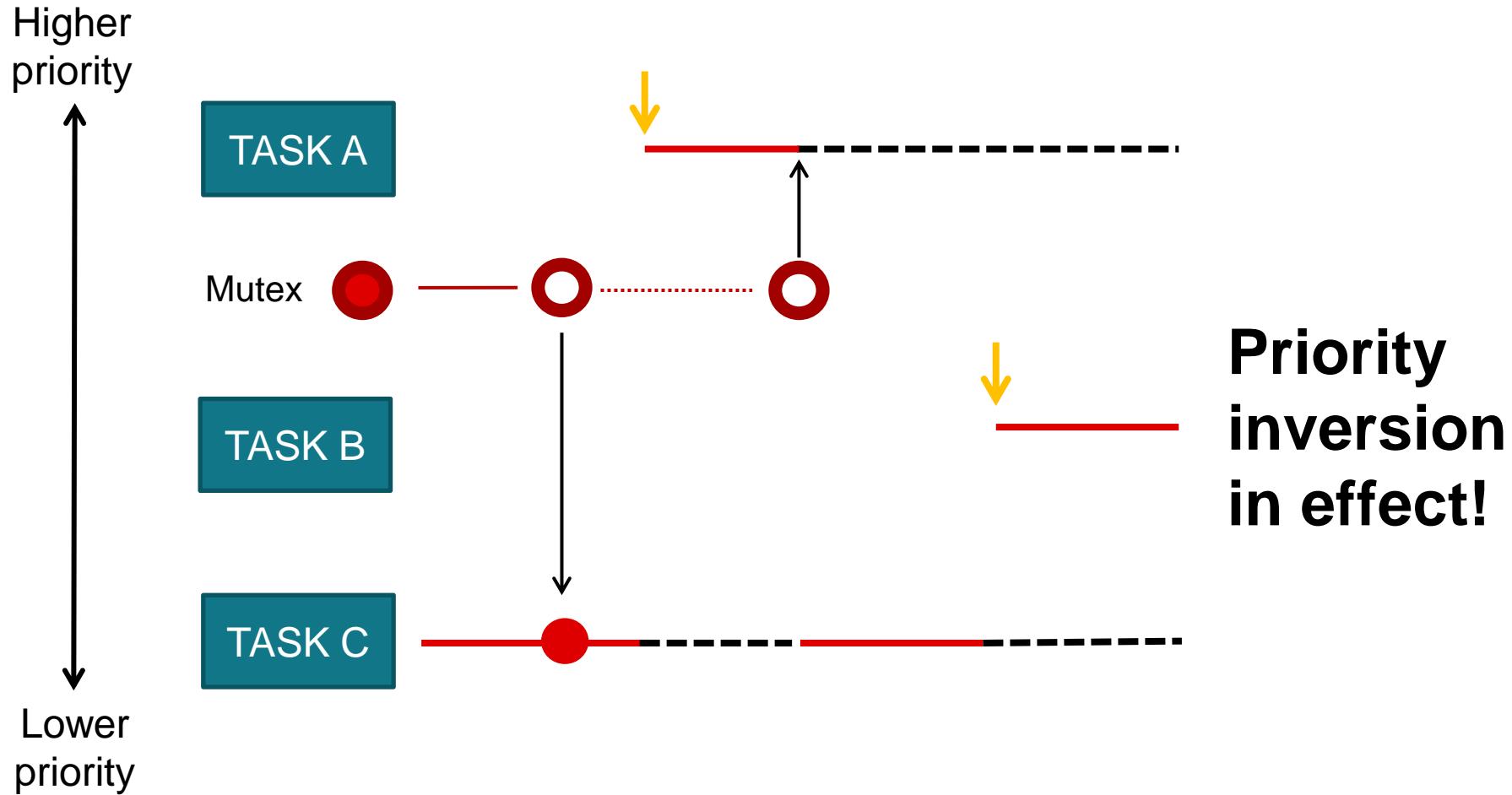
Priority Inversion

When a lower priority thread blocks a higher priority thread



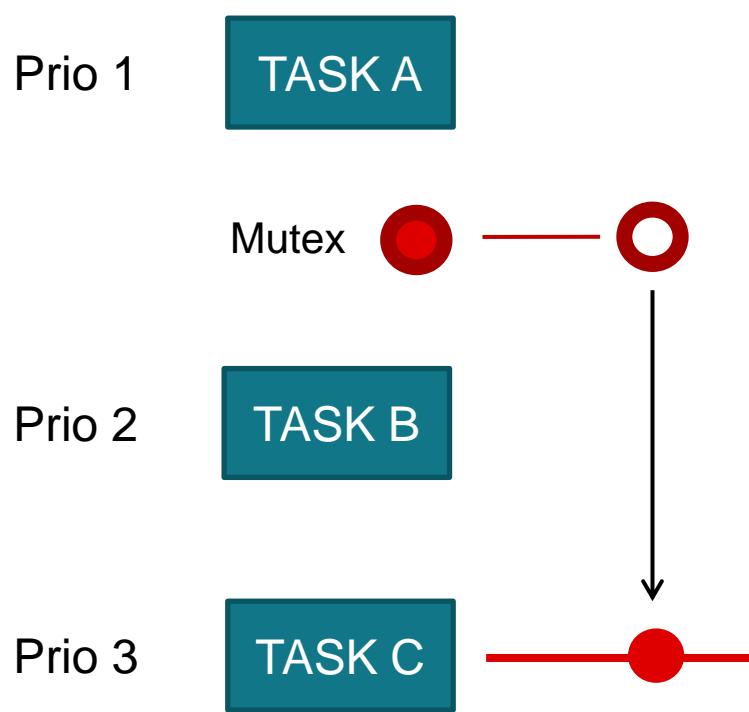
Priority Inversion

When a lower priority thread blocks a higher priority thread



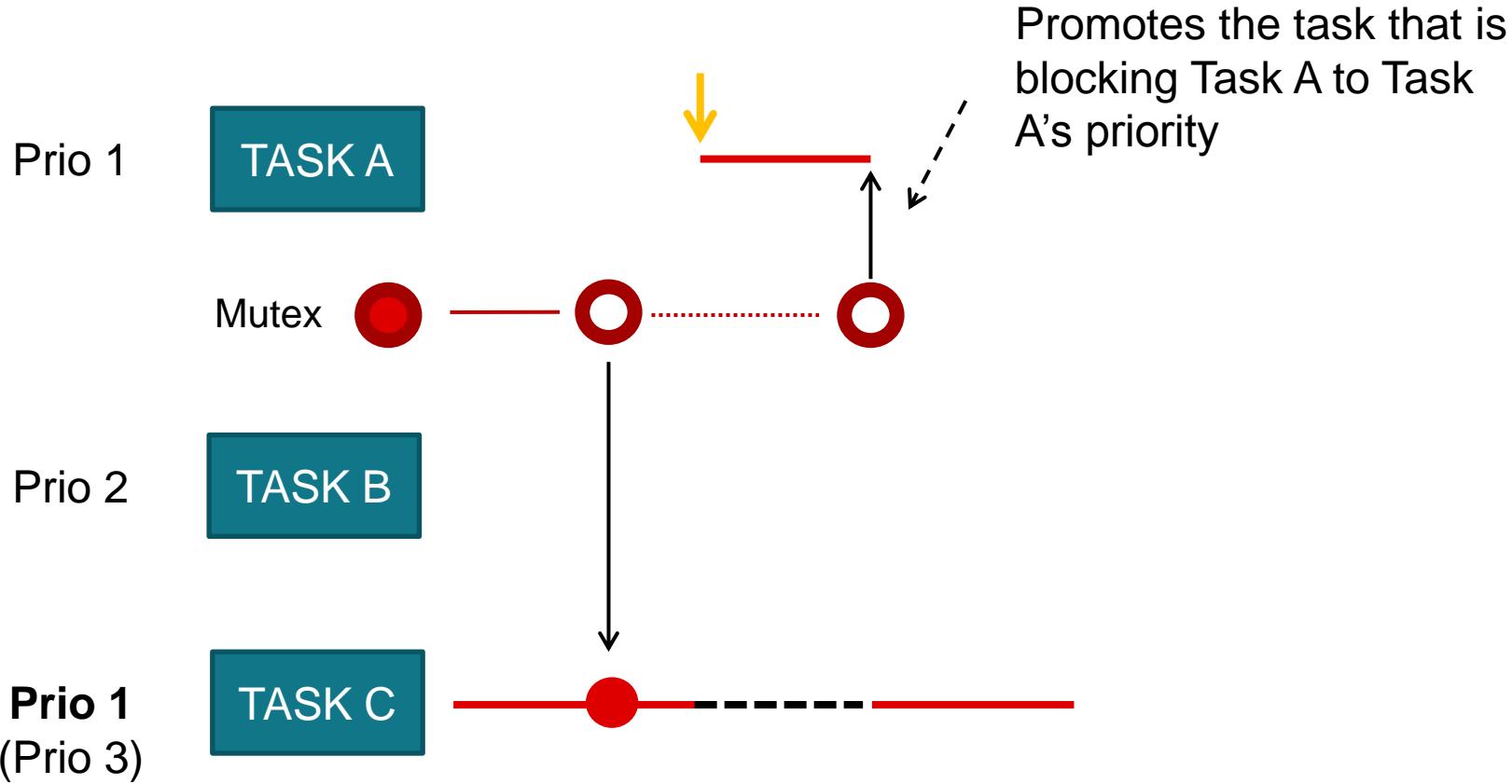
Priority Inversion

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



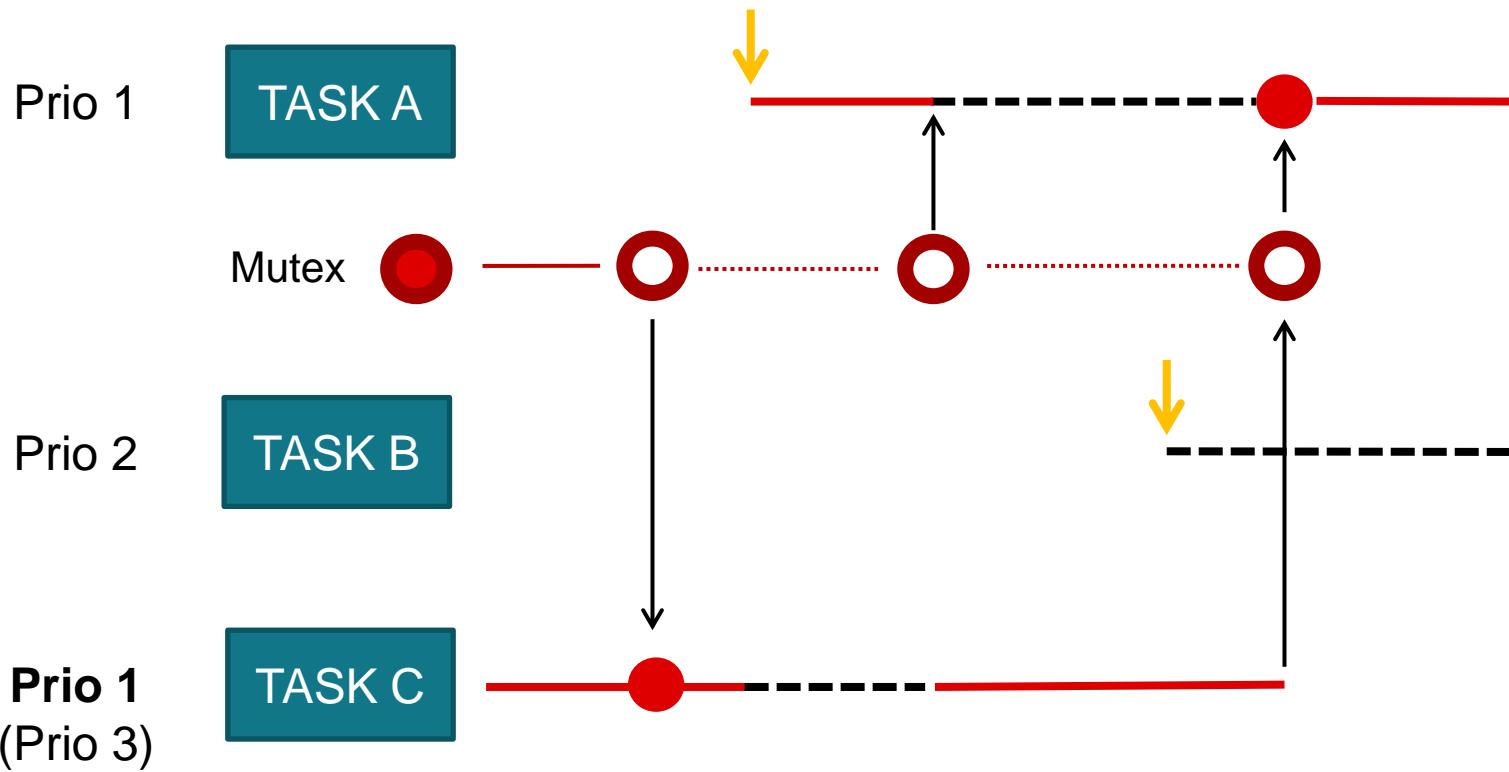
Priority Inversion

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

