« Real Time Embedded systems » MicroC/OS-II

rene.beuchat@epfl.ch

LAP/ISIM/IC/EPFL

Chargé de cours

LSN/EIG

Prof. HES

Introduction

- MicroC/OS-II is a Real-Time Kernel developed by Jean-J. Labrosse since 1992.
- It's certified for avionics equipment requirements.
- Book : Microc/OS-II The Real-Time Kernel, ISBN-13: 978-1-57820-103-7, CMPBooks
- Source available, NOT a license free software
- Version Microc/OS-III available

Main features

- Very well **documented** source code
- **Portable** (ARM, NIOSII, 8/16/32/64 bits processor, DSP, …)
- **Robust and Reliable**
- **ROMable**
- **Scalable**, only needed services include with *#define* constant by user
- **Multitasking (64, 256 v>2.8)**
  - **Task priority scheduling**
    - 1 task → 1 priority
- **Preemptive**

Main features (2)

- **Deterministic** for most of functions and services, except for OSTimeTick() and some event flags services, execution time do NOT depend on the number of task running
- **Task stacks** with different size capability and stack size check
- **Interrupt Management** with nested interrupt (255 levels deep possible)

Main features (3)

- **Services** as :
  - Task management functions
  - Semaphores
  - Mutual exclusion semaphores
  - Event flags
  - Message mailboxes
  - Message queues
  - Fixed-size memory partitions
  - …

Real-Time Systems Concepts

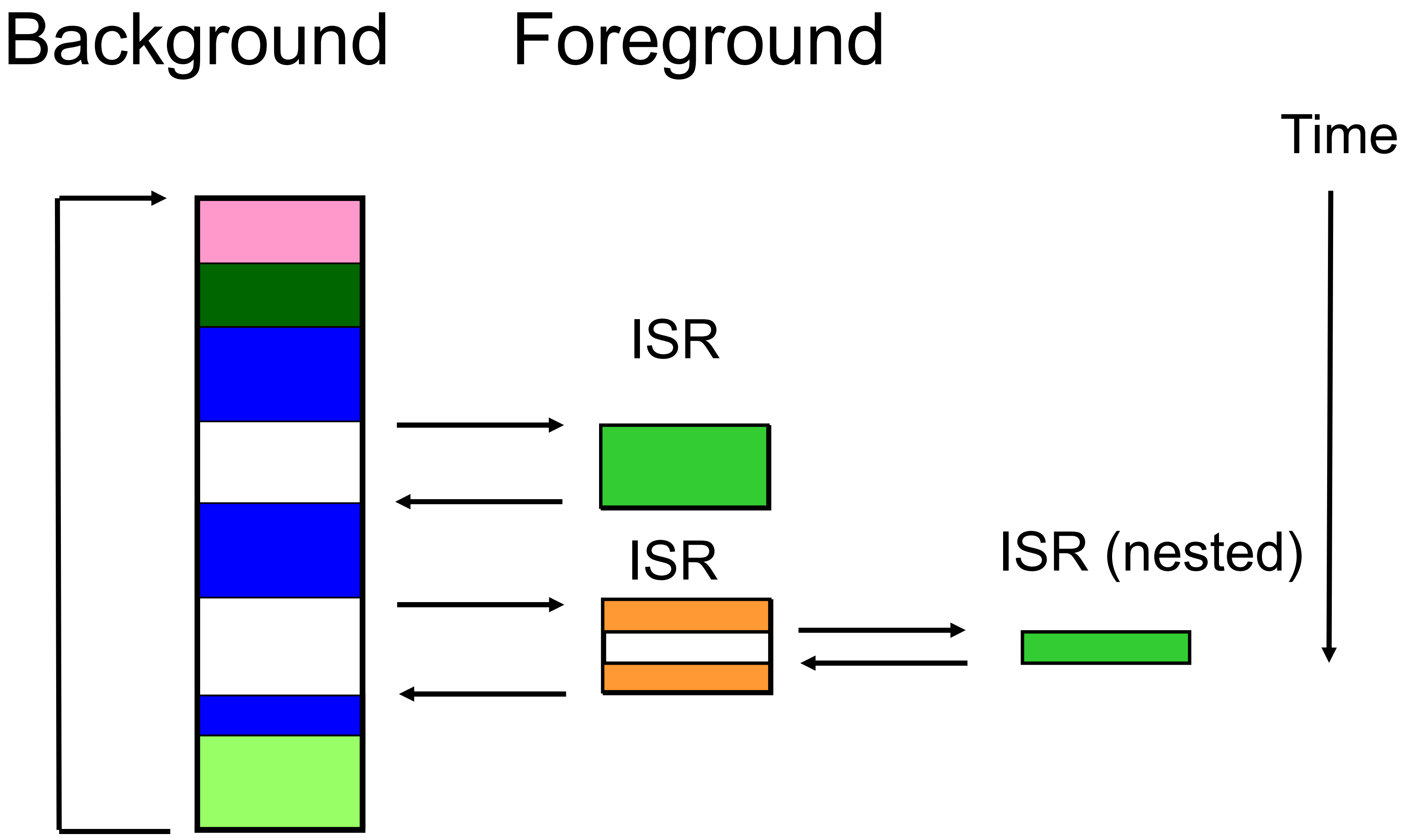
Soft/Hard Real Time

- **Soft Real Time** systems:
  - Tasks are executed as fast as possible but tasks don't have to finish by specific time
- **Hard Real Time** systems:
  - Task have to be finish on time
- Real systems :
  - both Hard and Soft Real Time requirement

Foreground/Background Systems

- Very often used model for small embedded systems microcontroller based
- NO kernel
  - ➢**Background**: A main loop program with sequentially executed tasks (functions call)
  - ➢**Foreground**: Some interrupt driven tasks, hardware event triggered

Foreground/Background Systems



Foreground/Background Systems

- Worst case *task-level response* depend on the length to execute the full main loop + ISR

Critical region

- A critical region or critical sections of code, is an indivisibly part of code. Thus it CAN NOT be interruptible.
- Thus the processor has to have it's **interrupts disabled** during the critical region
- Or it needs a way to verify that it has not been interrupted. In this case it has to run again the critical part.

Resources, shared resources

- A resource is any entity used by a task, ie:
  - I/O device
  - Variable
  - Data structure, array of data
- A **shared resource** is a resource than can be used by more than one task.
- Each task has to have exclusive access to the shared resource (it can be long: ex. Printer)
- It's the **mutual exclusion**

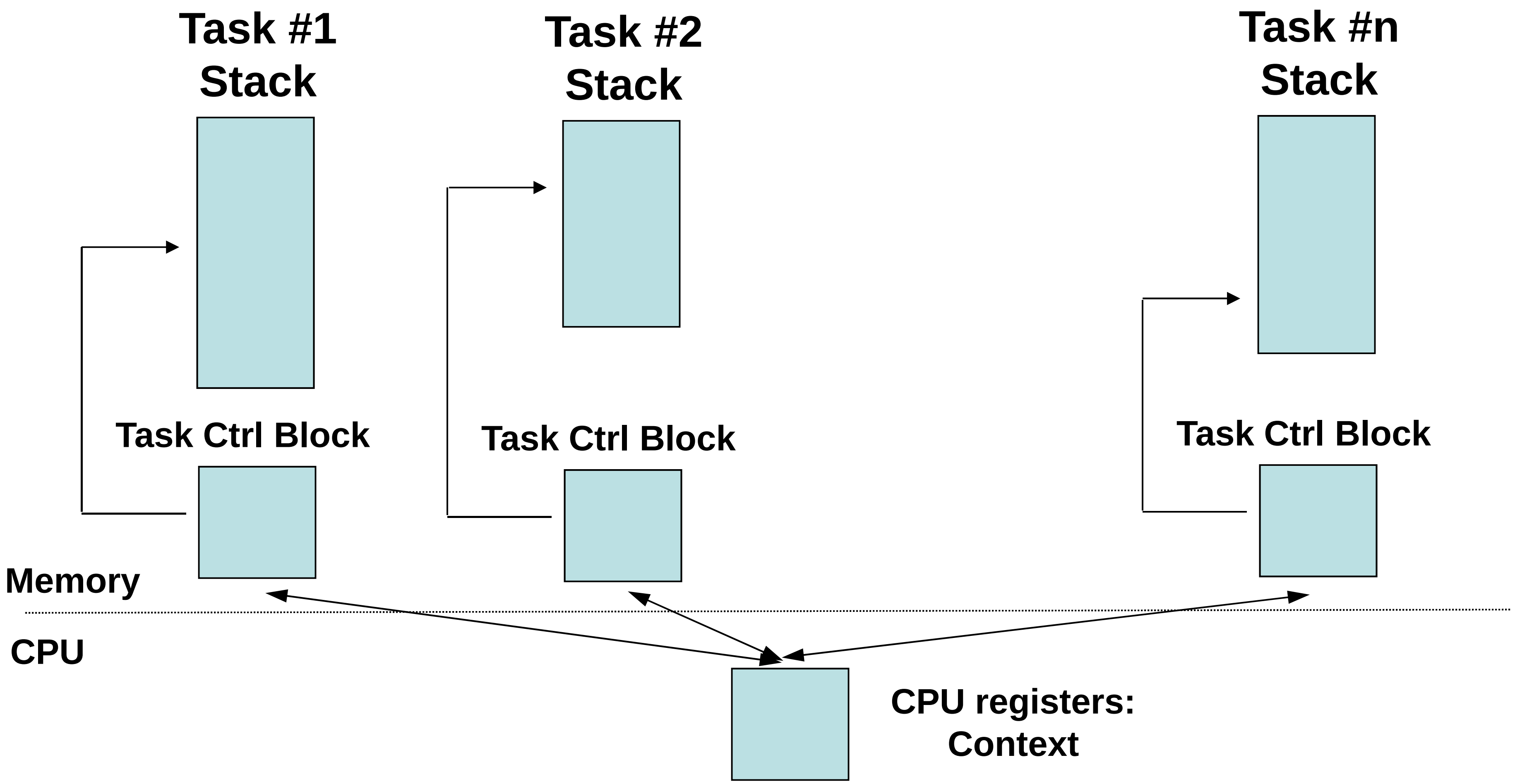
Multitasking

- The CPU has to be switched between **several tasks.**
- Multitasking is the process of scheduling and switching the CPU execution time between the tasks.
- A task is often call a ***thread***, is a simple program that "thinks" it has the CPU all to itself.

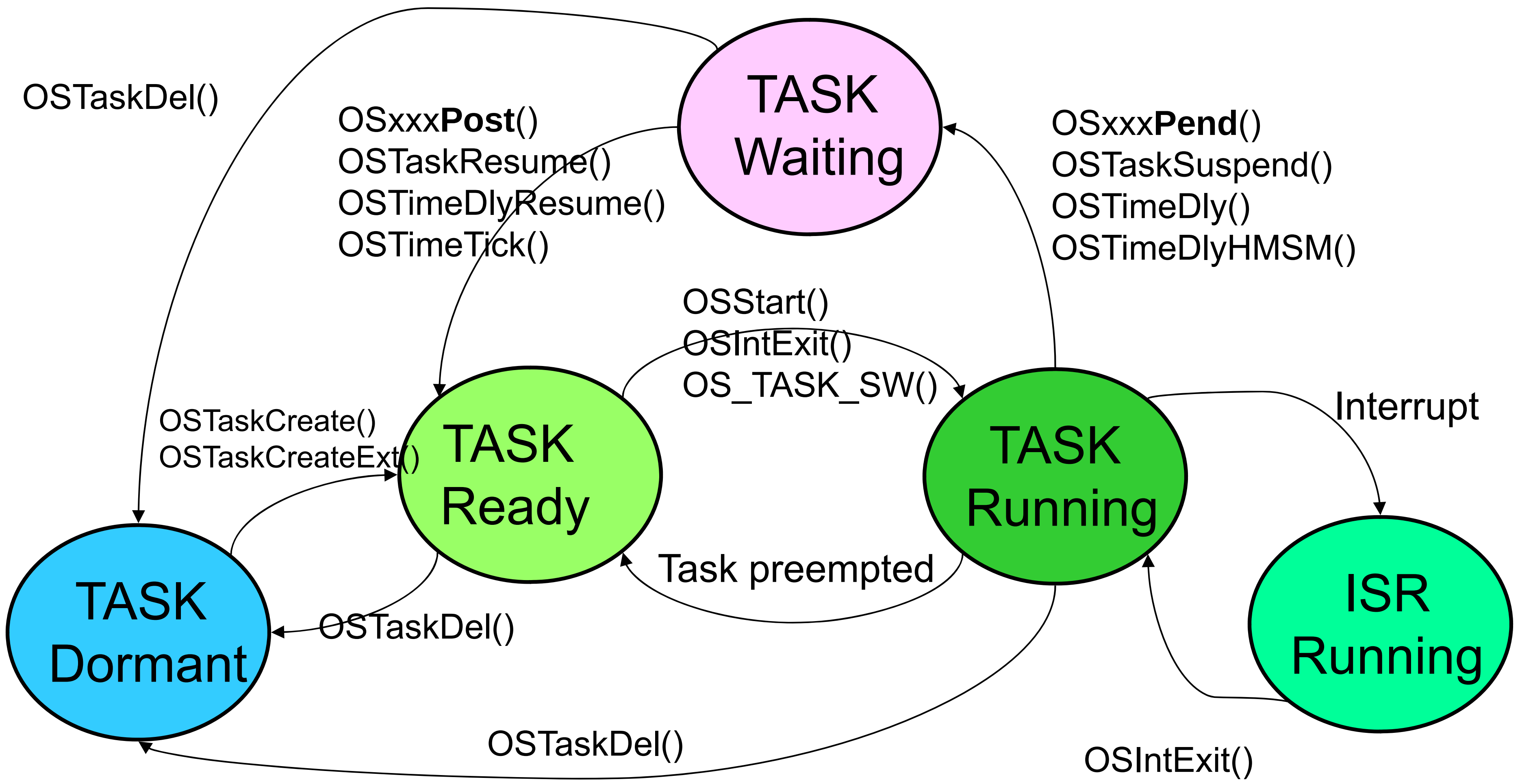
Tasks

- A task has :
  - ➢ it's own registers set
  - ➢ a priority
  - ➢ it's own stack area
- A task is typically an **infinite loop** that can be in any of 5 states :
  - ➢ **Dormant**
  - ➢ **Ready**
  - ➢ **Running**
  - ➢ **Waiting (for an event)**
  - ➢ **In an ISR (Interrupt Service Routine)**

Tasks


Task #1
Stack
Task #2
Stack
Task #n
Stack
Task Ctrl Block
Task Ctrl Block
Task Ctrl Block
Memory
CPU
CPU registers:
Context

Tasks states (MicroC/OS-II)



Tasks states (MicroC/OS-II)

- **Signal : Post**

  OS**xxx**Post():

  - OS**Flag**Post()
  - OS**Mbox**Post()
  - OS**Mbox**Post**Opt**()
  - OS**Mutex**Post()
  - OS**Q**Post()
  - OS**Q**Post**Front**()
  - OS**Q**Post**Opy**()
  - OS**Sem**Post()

- **Wait : Pending**

  OS**xxx**Pend():

  - OS**Flag**Pend()
  - OS**Mbox**Pend ()
  -
  - OS**Mutex**Pend ()
  - OS**Q**Pend ()
  -
  -
  - OS**Sem**Pend ()

Tasks Switches

- When the multitasking kernel decides to run on a different task :
  - ➢It saves the current task's context (CPU registers) → current task stack area
  - ➢New task's context restored from new task stack area
  - ➢Resume execution of the new task's code
  - ➢MORE registers → more overhead to save ALL the registers on the task's stack

Kernel

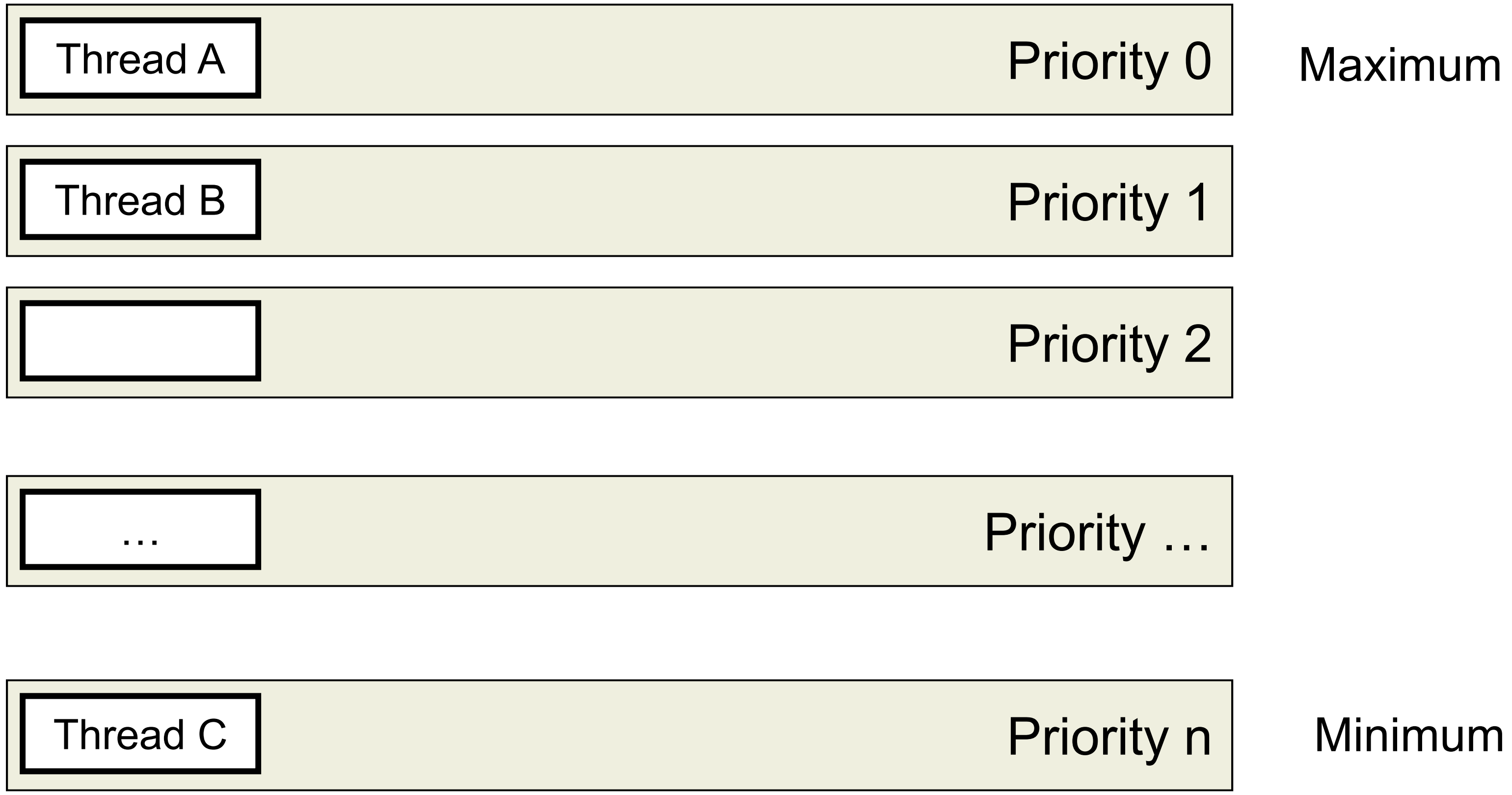
- The kernel is the part of the multitasking operating system for the management of tasks and communication between tasks
- Fundamental service is **context switching**
- Provide overhead of 2-5% (vs Background / Foreground systems) CPU time. Depend of the amount of invocation of these services

Scheduler

- The scheduler is the part of the kernel responsible for determining the next task to run.
- Most real-time systems are priority based
- The priority of each task is application dependant, in priority-based system, the CPU time is always given to the highest priority task

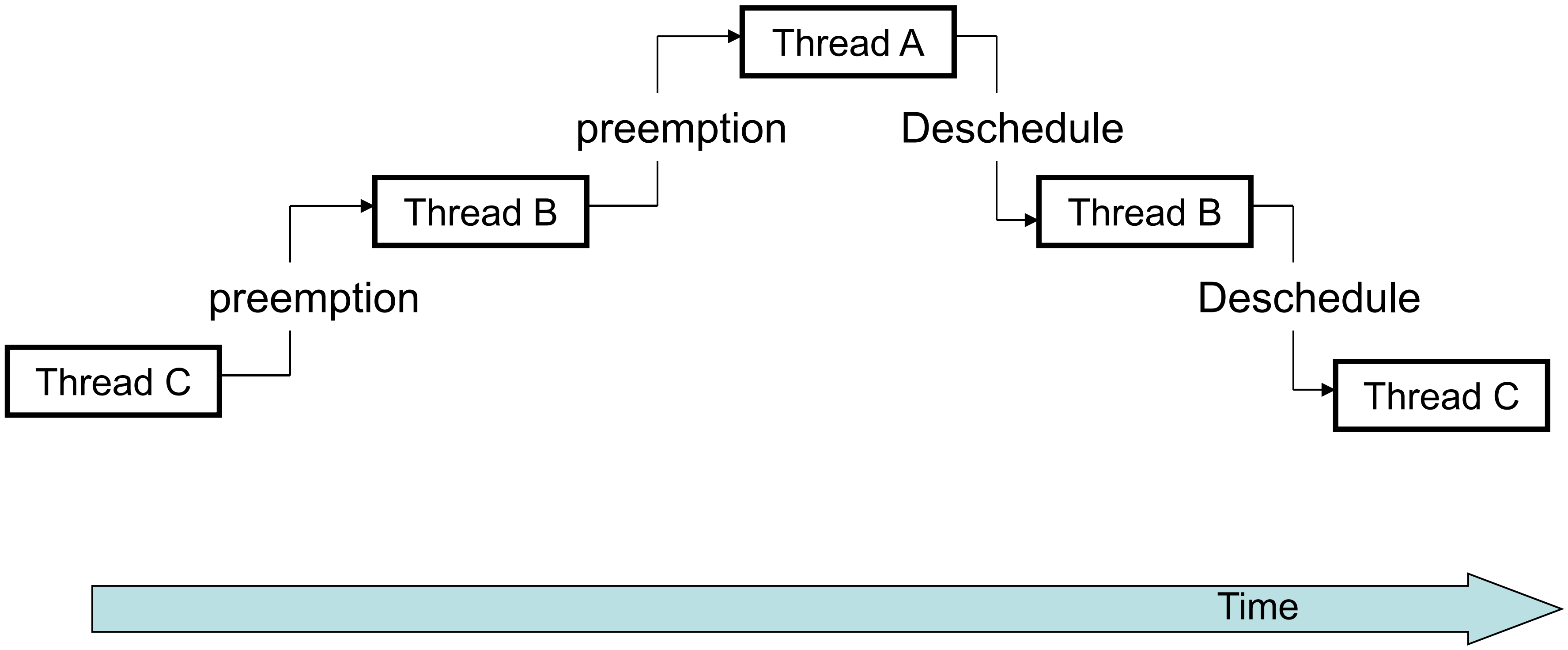
eCOS Kernel scheduler, Bit Map Scheduler
MicroC/OS-II

Only ONE thread on a priority level



eCOS Kernel scheduler, Bit Map Scheduler
MicroC/OS-II

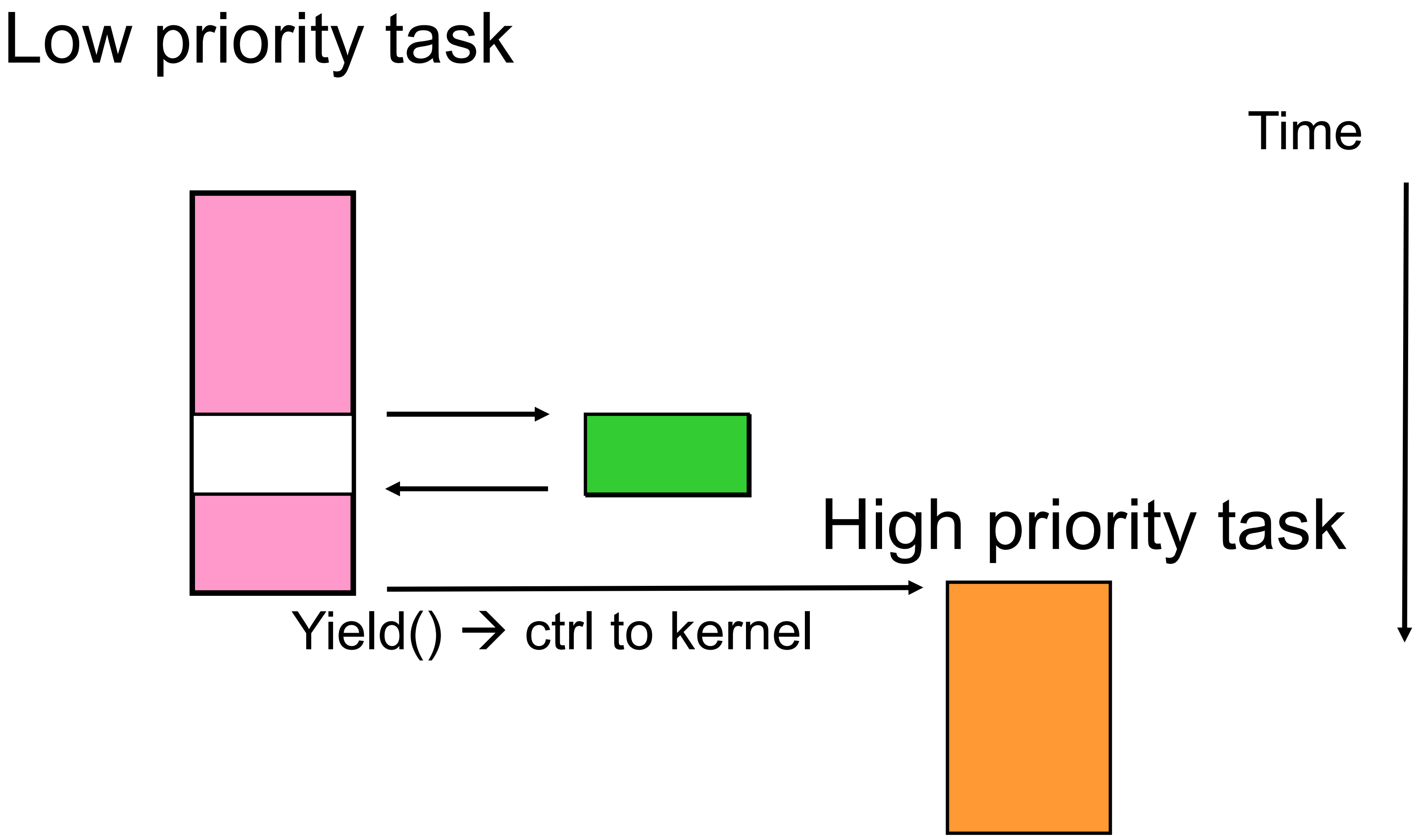
Only ONE thread on a priority level



Non-preemptive kernel

- In a NON preemptive kernel, the task's switching is always done when the task **explicitly** give up control of the CPU (ex. Yield() call)
- It's called *cooperative multitasking*
- ISR are available but do not provide context switching
- No-reentrant function can be used easily

Non-preemptive kernel



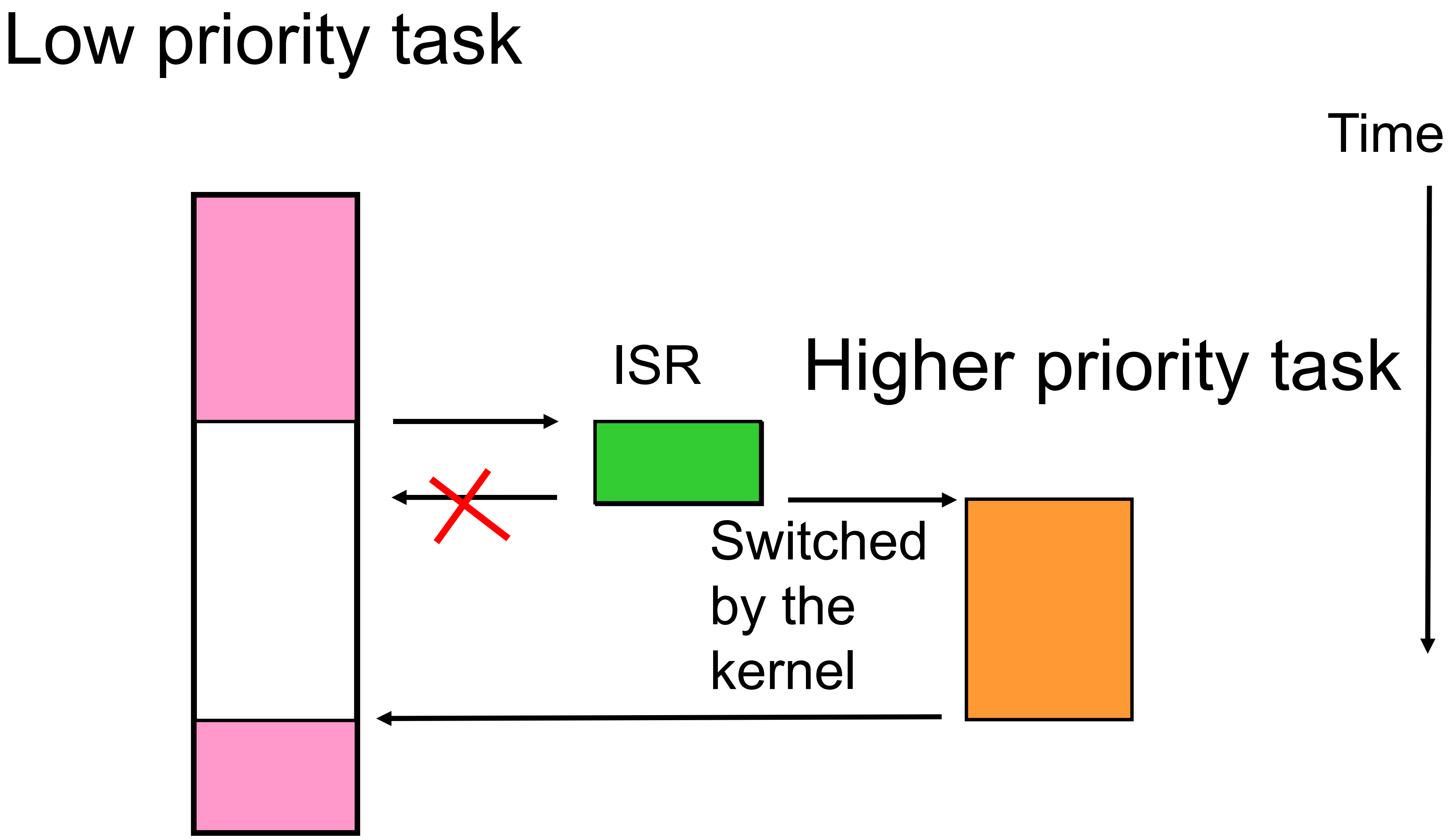
Non-preemptive kernel

- Drawback: responsiveness for high priority task can be very high if a lowest priority task do not relinquish the CPU for a long time
- Response time is not deterministic

Preemptive kernel

- The highest priority task always receive the CPU time when it's ready
- An ISR preempt the task (could be a timer)
- Execution is deterministic, task response time is minimized
- Don't use non-reentrant functions without mutual exclusive access :
  - ➢Mutual semaphore (mutex)

Preemptive kernel



Reentrant functions

- A reentrant function can be used by more than one task without corrupting data or devices
- A reentrant function can be interrupted at any time and continue later without loss of data
- Use local variables on CPU registers or on stack
- Protect global variables by mutex or interrupt disabled (not for long time!)

Reentrant functions (ex.)

```
void strcpy(char *dest, char *src){
   while(*dest++ = *src++){
   }
   *dest = NUL;
}
```

- Parameters are passed by the stack
- Multiple tasks can call **strcpy** without problems

Non-Reentrant functions (ex.)

```
Int Temp;
void swap(int *x, int *y){
   Temp = *x;
   *x = *y;
   *y = Temp;
}
```

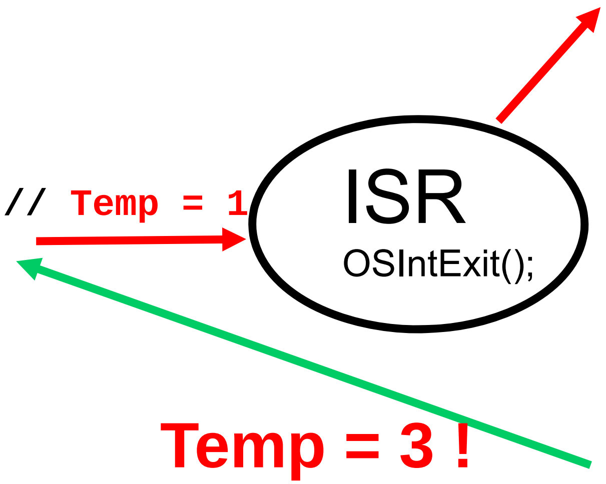
Problem if task switching is here!

- Temp : global variable

Non-Reentrant functions (ex.)

Low priority task

```
x=1;
y=2;
Swap(&x, &y);
    {
        Temp = *x; // Temp = 1

        *x = *y;
        *y = Temp;
}

OSTimeDly(1);
→ x = 2
→ y = 3
```



High priority task

```
z=3;
t=4;
Swap(&z, &t);
    {
        Temp = *z; // Temp = 3
        *z = *t;
        *t = Temp;
}
OSTimeDly(1);

→ z = 4
→ t = 3
```

Temp : global variable

Non-Reentrant functions (ex.)

Correction to make the swap function reentrant:

- Temp : global variable NO
- Temp → **local variable** to swap()

or

- Protect access to Temp use by **exclusion access semaphore**

or

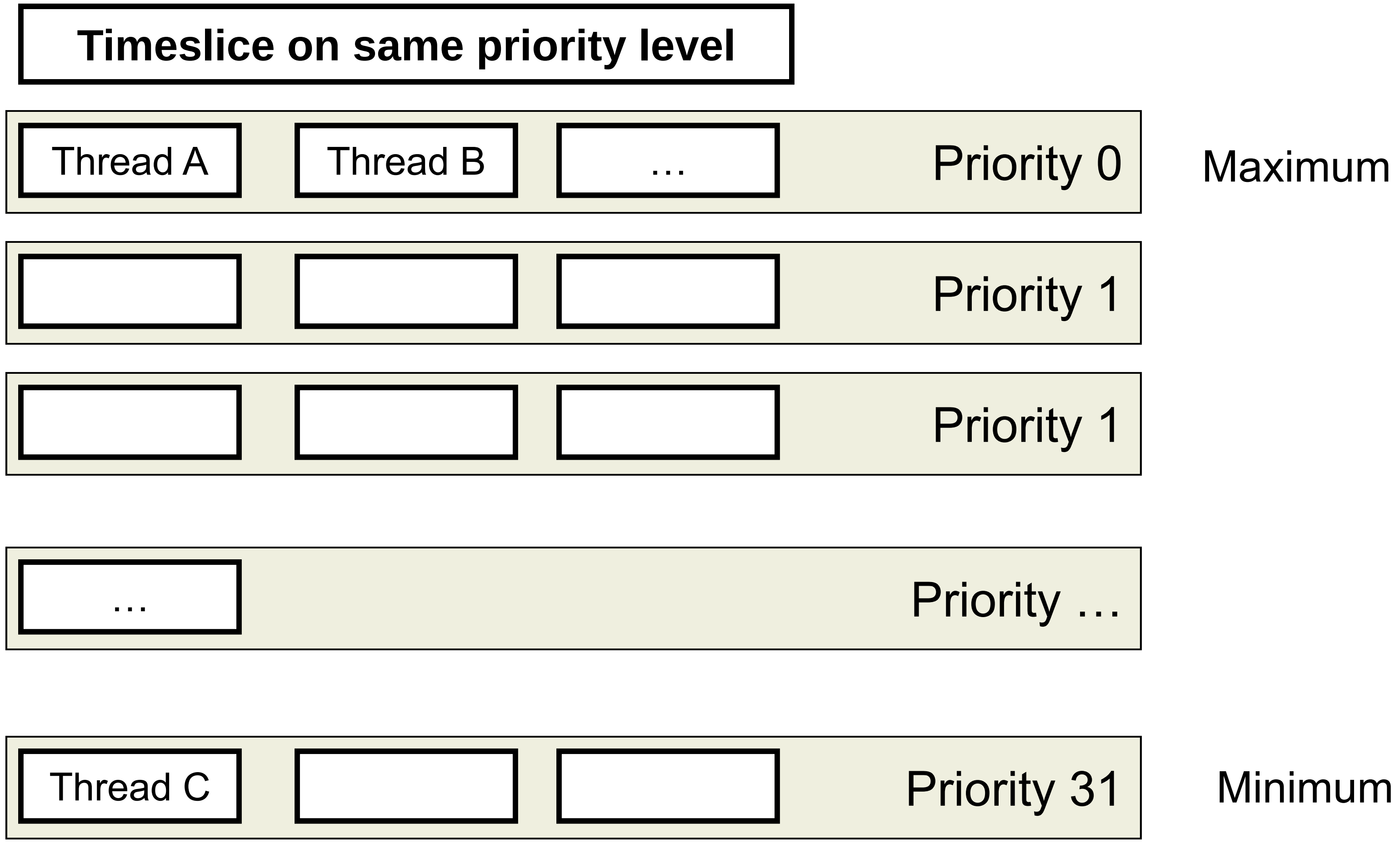
- **Disable interrupt** during Temp use

Round-robin scheduling

Capability for the scheduler to support more than 1 task to the same priority

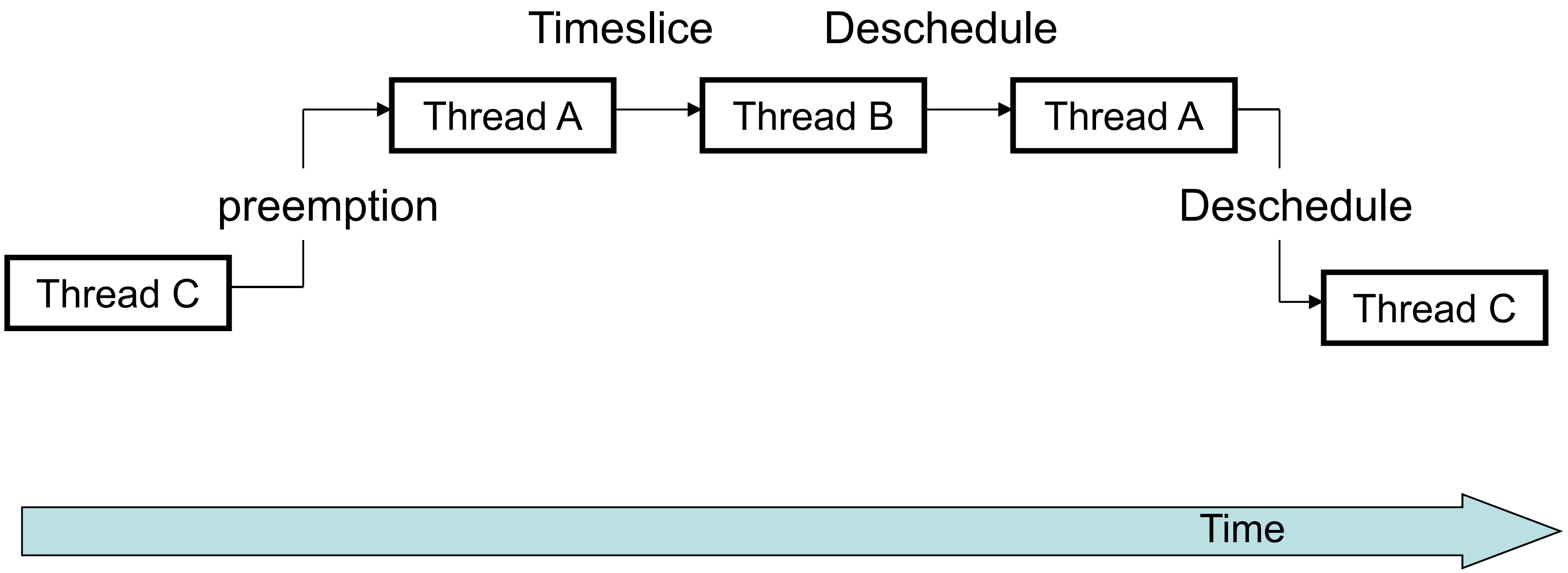
Allow task execution by quantum of time and task scheduling is run

eCOS Kernel scheduler
Multilevel Queue Scheduler



eCOS Kernel scheduler
Multilevel Queue Scheduler

Timeslice on same priority level



Task priorities

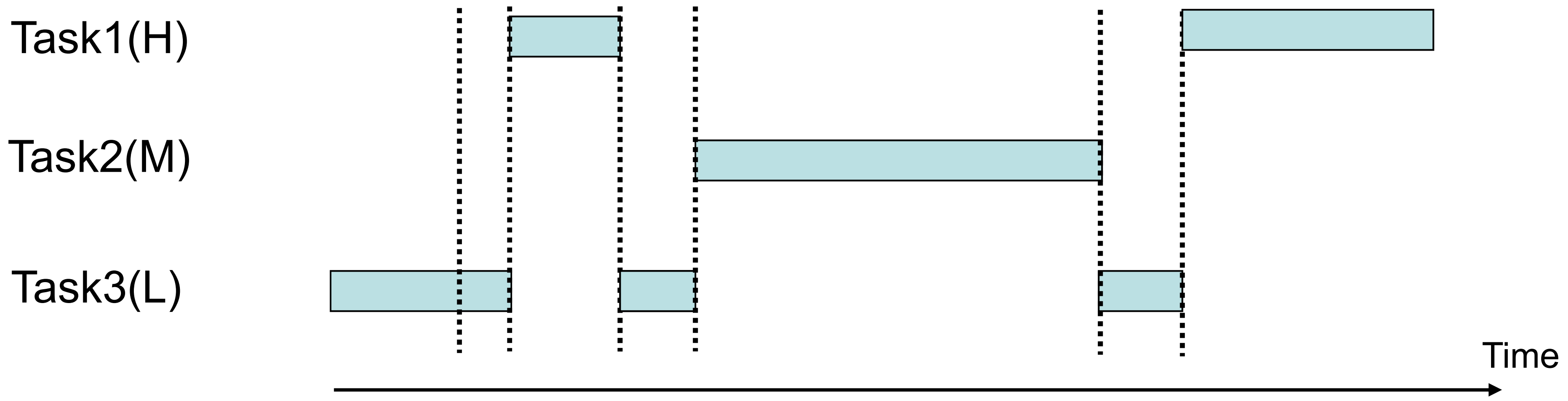
- Each task has a priority :
- The most important → the highest priority

- **Static priority**
  - ➢ The priority does not change during the application
- **Dynamic priority**
  - ➢ The priority can change during the application's execution at run time

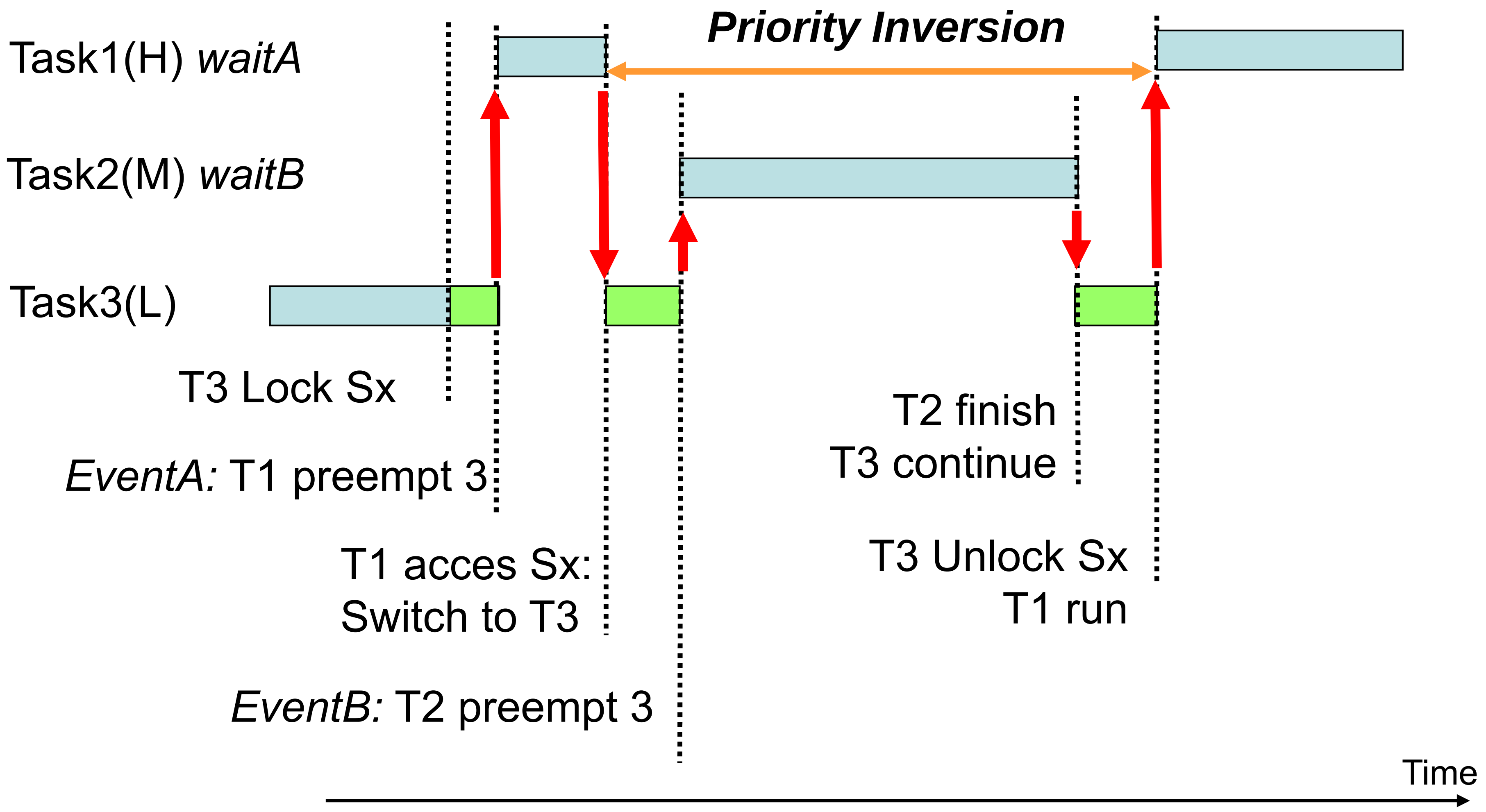
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Priority inversion

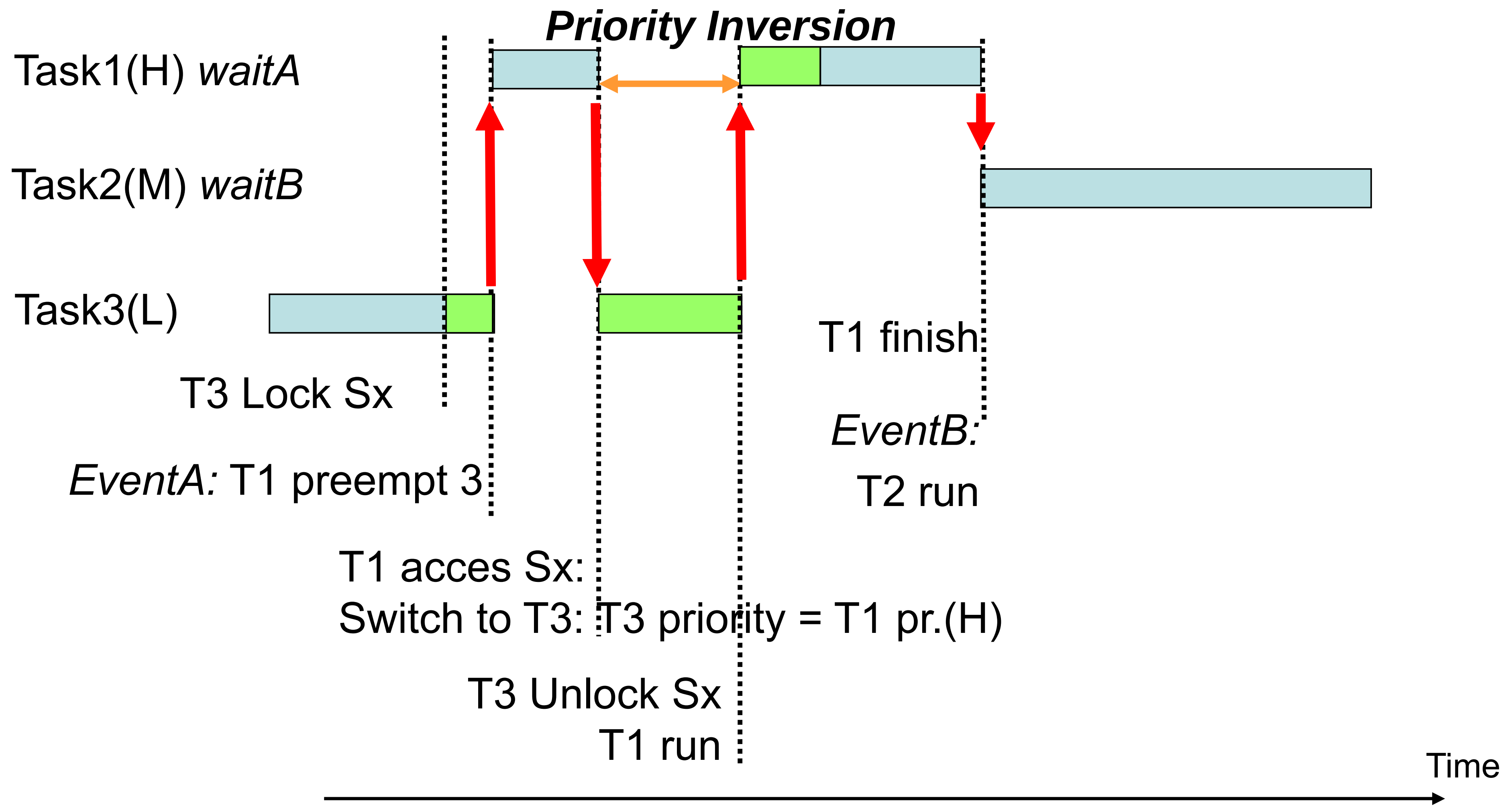
- Problem in real time systems, example:
  - Task1 highest priority
  - Task 2 middle priority
  - Task 3 lowest priority
  - Semaphore X: Sx



Priority inversion



Priority inversion → inheritance



Priority inversion → inheritance

- Utilization of mutex for mutual exclusive access to a resource with priority change to the highest of the tasks waiting for it.

Assigning task priority

- Very difficult task, but some rules:
- Non critical tasks → lowest priority
- *Rate Monotonic Scheduling* is a technique to assign priority with the simple rule :
  - ➢Put the highest rate of execution the highest priority

RMS (*Rate Monotonic Scheduling*)

- Some assumptions:
  - *All tasks are periodic*
  - *Tasks do not synchronize with one another, share resources or exchange data*
  - *Preemptive scheduling on highest priority task*
  - *Hard real time deadlines are always met if:*
- $\sum Ei / Ti \leq n * (2^{1/n} - 1)$
  - *Ei : max execution time task i*
  - *Ti : execution period of task I*
  - *Ei / Ti : fraction of execution time of task I*
  - *n: number of task*

RMS (*Rate Monotonic Scheduling*)

- *$n \to \infty$, $\sum Ei / Ti \leq ln(2) \approx 0.693$*
- *The sum of all critica task need to be less than 70% CPU time!*
- *Stay some times for non critical task !*
- *It's a starting point for priority choice !*

Mutual exclusion

- **Disabling interrupts** (MicroC/OS-II):
  - OS_ENTER_CRITICAL();
    - Interrupt disabled
  - OS_EXIT_CRITICAL();

- **Disabling Scheduler**
  - If not accessed by ISR
  - OSSchedLock();
    - Scheduler disabled, interrupt enabled !
    - DO NOT use OSxxxPend() or TimeDlyxx() functions !
  - OSSchedUnLock();

Mutual exclusion

- **Semaphores** (MicroC/OS-II):
  - ➢Mutual exclusion to access shared resources
  - ➢Signal the occurrence of an event
  - ➢Allow two task to synchronize

  - ➢Binary semaphore (0, 1)
  - ➢Counter semaphore

Semaphore

- Initialize() or Create() with n > 0
- WAIT() or PEND(),
  - ➢if n > 0 → n= n-1 → access allowed
  - ➢If n == 0 → calling task go on waiting list, control to another ready task
- SIGNAL() or POST()
  - ➢Task waiting → execute
  - ➢No task waiting → n = n+1

Semaphore

- Execution :
  - Highest priority task (MicroC/OS-II)
  - First task waiting for the semaphore (FIFO)

```
OS_EVENT *SharedDataSem;
SharedDataSem  OSSemCreate( 1 );  // Create the semaphore


Void Function(void){
   INT8U err;
   OSSemPend(SharedDataSem, 0, &er);
       //Exclusion access
   OSSemPost(SharedDataSem);
}
```

Mutual exclusion semaphores

- Binary semaphore for mutual exclusion
- A parameter PIP (Priority inheritance priority) is pass at Creation time.
- It's a priority reserved with the value of the highest priority of all task that can wait on the mutex
- It's a way to resolve the priority inversion without having two task at the same priority !

Mutual exclusion semaphores

OS_EVENT ***OSMutexCreate**(INT8U prio, INT8U *err);
OSMutexCreate() is used to create and initialize a mutex. A mutex is used to gain exclusive access to a resource.

void **OSMutexPend**(OS_EVENT *pevent, INT16U timeout, INT8U *err);
OSMutexPend() is used when a task desires to get exclusive access to a resource. If a task calls OSMutexPend() and the mutex is available, then OSMutexPend() gives the mutex to the caller and returns to its caller. Note that nothing is actually given to the caller except for the fact that if err is set to OS_NO_ERR, the caller can assume that it owns the mutex. However, if the mutex is already owned by another task, OSMutexPend() places the calling task in the wait list for the mutex. The task thus waits until the task that owns the mutex releases the mutex and thus the resource or until the specified timeout expires. If the mutex is signaled before the timeout expires, _C/OS-II resumes the highest priority task that is waiting for the mutex. Note that if the mutex is owned by a lower priority task, then OSMutexPend() raises the priority of the task that owns the mutex to the PIP, as specified when you created the mutex [see OSMutexCreate()].

INT8U **OSMutexPost**(OS_EVENT *pevent);
A mutex is signaled (i.e., released) by calling OSMutexPost(). You call this function only if you acquire the mutex by first calling either OSMutexAccept() or OSMutexPend(). If the priority of the task that owns the mutex has been raised when a higher priority task attempts to acquire the mutex, the original task priority of the task is restored. If one or more tasks are waiting for the mutex, the mutex is given to the highest priority task waiting on the mutex. The scheduler is then called to determine if the awakened task is now the highest priority task ready to run, and if so, a context switch is done to run the readied task. If no task is waiting for the mutex, the mutex value is simply set to available (0xFF).

Mutual exclusion semaphores

INT8U **OSMutexQuery**(OS_EVENT *pevent, OS_MUTEX_DATA *pdata);

OSMutexQuery() is used to obtain run-time information about a mutex. Your application must allocate an OS_MUTEX_DATA data structure that is used to receive data from the event control block of the mutex. OSMutexQuery() allows you to determine whether any task is waiting on the mutex, how many tasks are waiting (by counting the number of 1s) in the .OSEventTbl[] field, obtain the PIP, and determine whether the mutex is available (1) or not (0). Note that the size of .OSEventTbl[] is established by the #define constant OS_EVENT_TBL_SIZE (see uCOS_II.H).

OS_EVENT ***OSMutexDel**(OS_EVENT *pevent, INT8U opt, INT8U *err);

OSMutexDel() is used to delete a mutex. This function is dangerous to use because multiple tasks could attempt to access a deleted mutex. You should always use this function with great care. Generally speaking, before you delete a mutex, you must first delete all the tasks that can access the mutex.

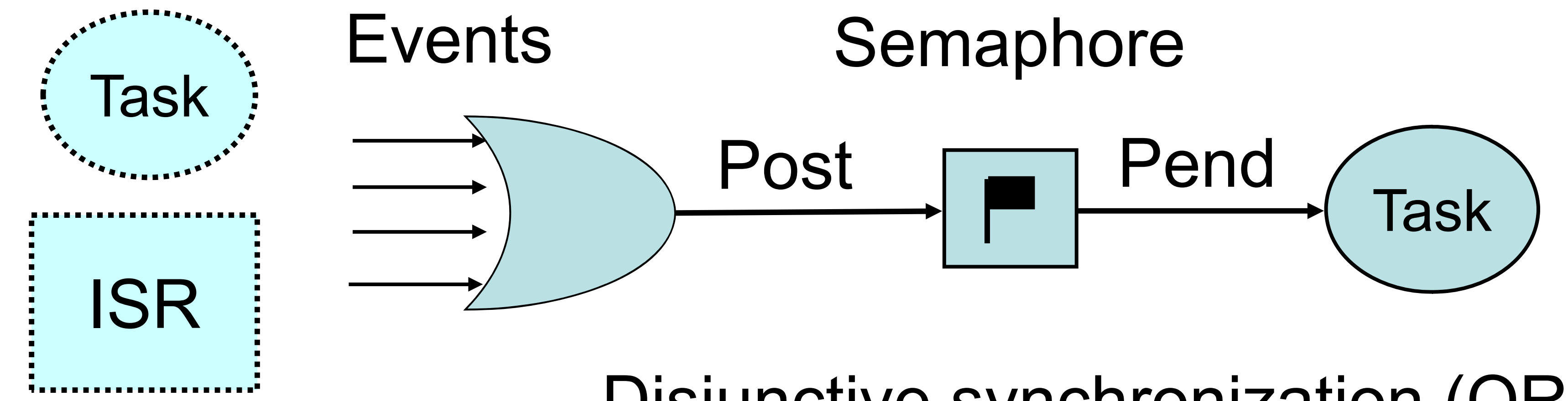
INT8U **OSMutexAccept**(OS_EVENT *pevent, INT8U *err);

OSMutexAccept() allows to check to see if a resource is available. Unlike OSMutexPend(), OSMutexAccept() does not suspend the calling task if the resource is not available. In other words, OSMutexAccept() is non-blocking.

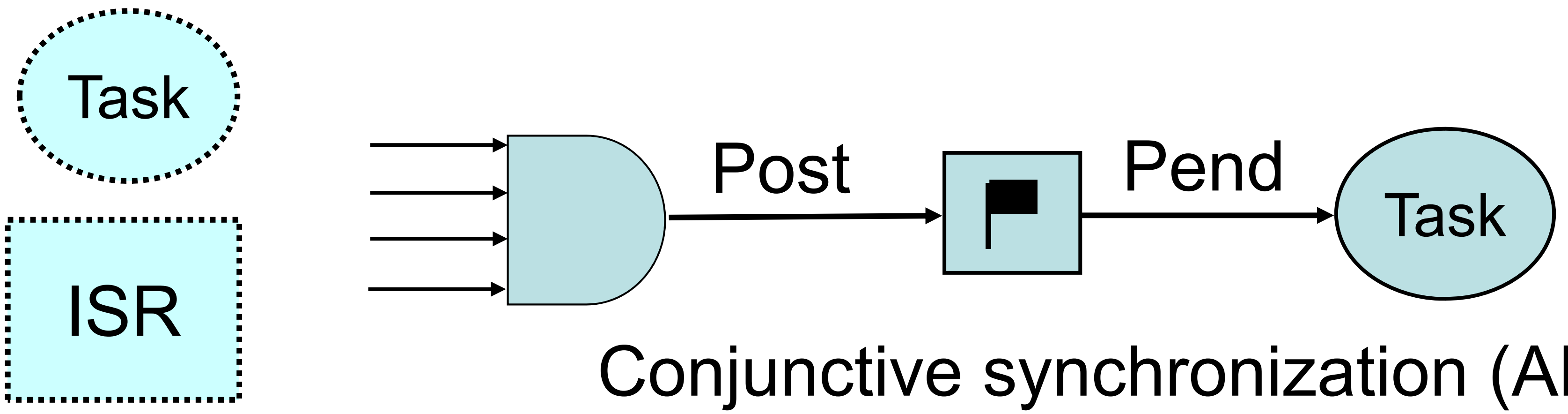
Event flags

- An events flag is used when a task needs to synchronize with the occurrence of multiple events.
- The task can be synchronized when ANY of the events have occurred→ OR function, it's call *disjunctive synchronization*
- The task can be synchronized when ALL of the events have occurred→ AND function, it's call *conjunctive synchronization*

Event flags



Disjunctive synchronization (OR)



Conjunctive synchronization (AND)

Event flags

- An Event is a bit activated in a Task or ISR on an event flag
- Evaluation is done when SET function is done

Event flags

OS_FLAG_GRP ***OSFlagCreate** (OS_FLAGS flags, INT8U *err);

OSFlagCreate() is used to create and initialize an event flag group.

Event flags

- **OSFlagPost()**
- OS_FLAGS **OSFlagPost**(OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U opt, INT8U err);

- You set or clear event flag bits by calling OSFlagPost(). The bits set or cleared are specified in a bit mask. OSFlagPost() readies each task that has its desired bits satisfied by this call. You can set or clear bits that are already set or cleared.

Event flags

OS_FLAGS **OSFlagPend** (OS_FLAG_GRP *pgrp,
    OS_FLAGS flags,
    INT8U wait_type,
    INT16U timeout,
    INT8U *err);

OSFlagPend() is used to have a task wait for a combination of conditions (i.e., events or bits) to be set (or cleared) in an event flag group. Your application can wait for **any** condition to be set or cleared or for **all** conditions to be set or cleared. If the events that the calling task desires are not available, then the calling task is blocked until the desired conditions are satisfied or the specified timeout expires.

Event flags Arguments

- **pgrp** is a pointer to the event flag group. This pointer is returned to your application when the event flag group is created [see OSFlagCreate()].
- **flags** is a bit pattern indicating which bit(s) (i.e., flags) you wish to check. The bits you want are specified by setting the corresponding bits in flags.
- **wait_type** specifies whether you want **all** bits to be set/cleared or **any** of the bits to be set/cleared. You can specify the following arguments:
  - OS_FLAG_WAIT_CLR_ALL You check **all** bits in flags to be clear (0)
  - OS_FLAG_WAIT_CLR_ANY You check **any** bit in flags to be clear (0)
  - OS_FLAG_WAIT_SET_ALL You check **all** bits in flags to be set (1)
  - OS_FLAG_WAIT_SET_ANY You check **any** bit in flags to be set (1)
- You can also specify whether the flags are consumed by adding **OS_FLAG_CONSUME** to the wait_type. For example, to wait for **any** flag in a group and then **clear** the flags that satisfy the condition, set wait_type to
  - OS_FLAG_WAIT_SET_ANY + OS_FLAG_CONSUME
- **err** is a pointer to an error code and can be:
  - OS_NO_ERR No error.
  - OS_ERR_PEND_ISR You try to call OSFlagPend from an ISR, which is not allowed.
  - OS_FLAG_INVALID_PGRP You pass a NULL pointer instead of the event flag handle.
  - OS_ERR_EVENT_TYPE You are not pointing to an event flag group.
  - OS_TIMEOUT The flags are not available within the specified amount of time.
  - OS_FLAG_ERR_WAIT_TYPE You don’t specify a proper wait_type argument.

Event flags

OS_FLAGS **OSFlagAccept** (OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, NT8U *err);

OSFlagAccept() allows to check the status of a combination of bits to be either set or cleared in an event flag group. Your application can check for **any** bit to be set/cleared or **all** bits to be set/cleared. This function behaves exactly as OSFlagPend() does, except that the caller does NOT block if the desired event flags are not present.

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Intertask Communication

- For intertask communication, global variables can be used. The access has to be protected by semaphore.
- In case of ISR (Interrupt Service Routine), **disabling interrupt** is necessary, as waiting on a semaphore is NOT allowed in an ISR.

Intertask Communication

- If an ISR need to signal a variable modification, synchronizing semaphore can be used. The Post (Signal) can be done in the ISR, but never the Pend (Wait)
- Active wait on a variable modification can sometimes be used (polling on a variable), during this time only task with higher priority can take the processor.

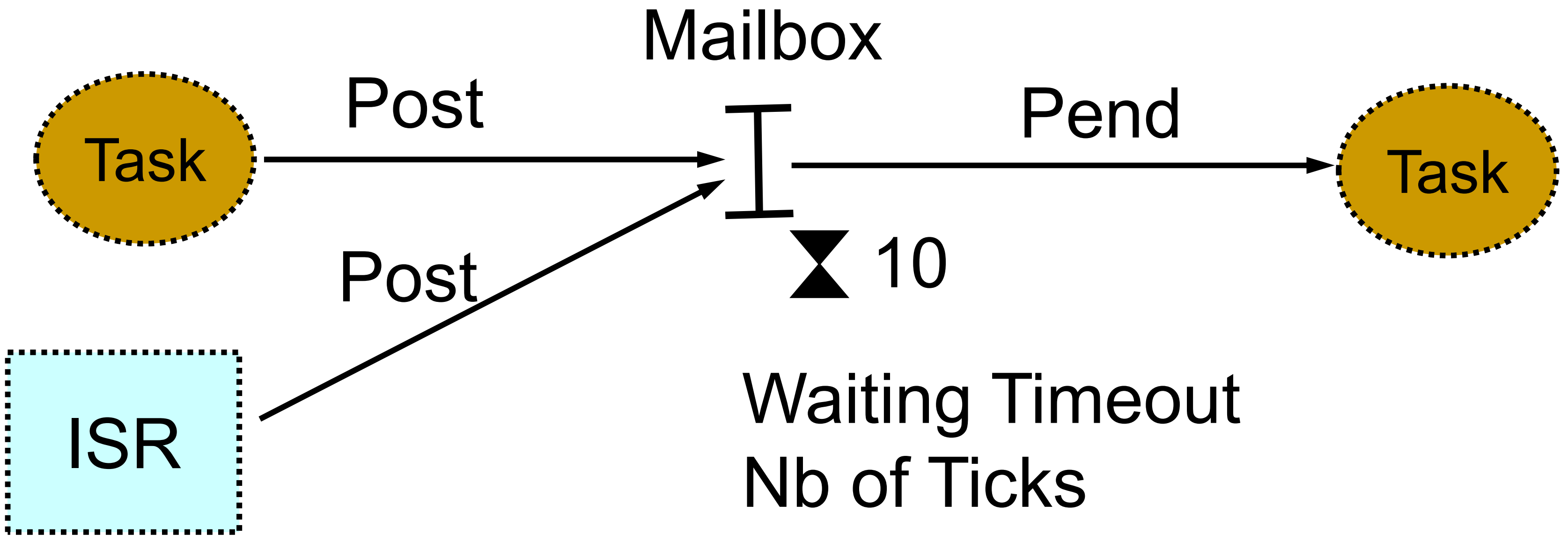
Message mailboxes

- Message can be used to communicate between Tasks and ISR to Task.
- The kernel can provide *message mailbox* services. In general the message is a **pointer** to an known structure by both the sender and the receiver

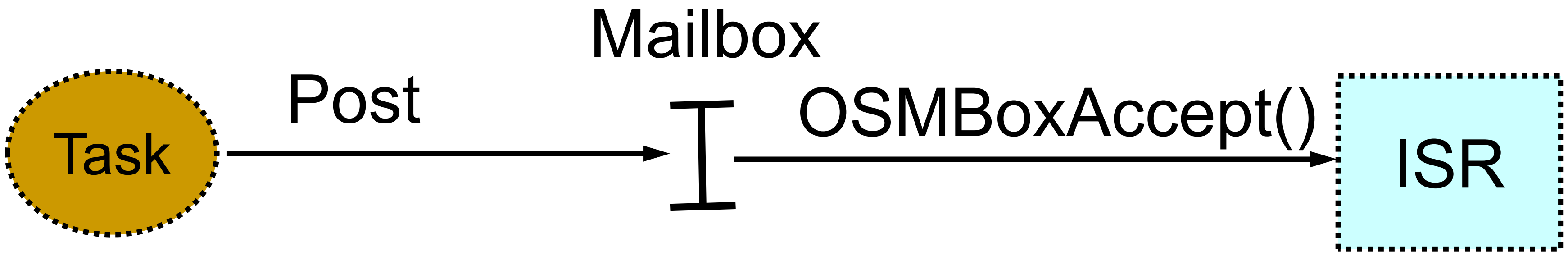
Message mailboxes

- Only ONE message can be put in the mailbox at a time.
- If a new message is Posted, and the previous one not consumed (Pend) an error is provided and the new message is not accepted

Message mailboxes



Message mailboxes



In an ISR, only the Accept() access is allowed to read a Mailbox

Message mailboxes

- OS_EVENT ***OSMboxCreate** (void *msg);

- OSMboxCreate() creates and initializes a mailbox. A mailbox allows tasks or ISRs to send a pointer-variable (message) to one or more tasks.
- **msg** is used to initialize the contents of the mailbox. The mailbox is empty when msg is a NULL pointer. The mailbox initially contains a message when msg is non-NULL.

Message mailboxes

- void ***OSMboxPend** (OS_EVENT *pevent, INT16U timeout, INT8U *err);

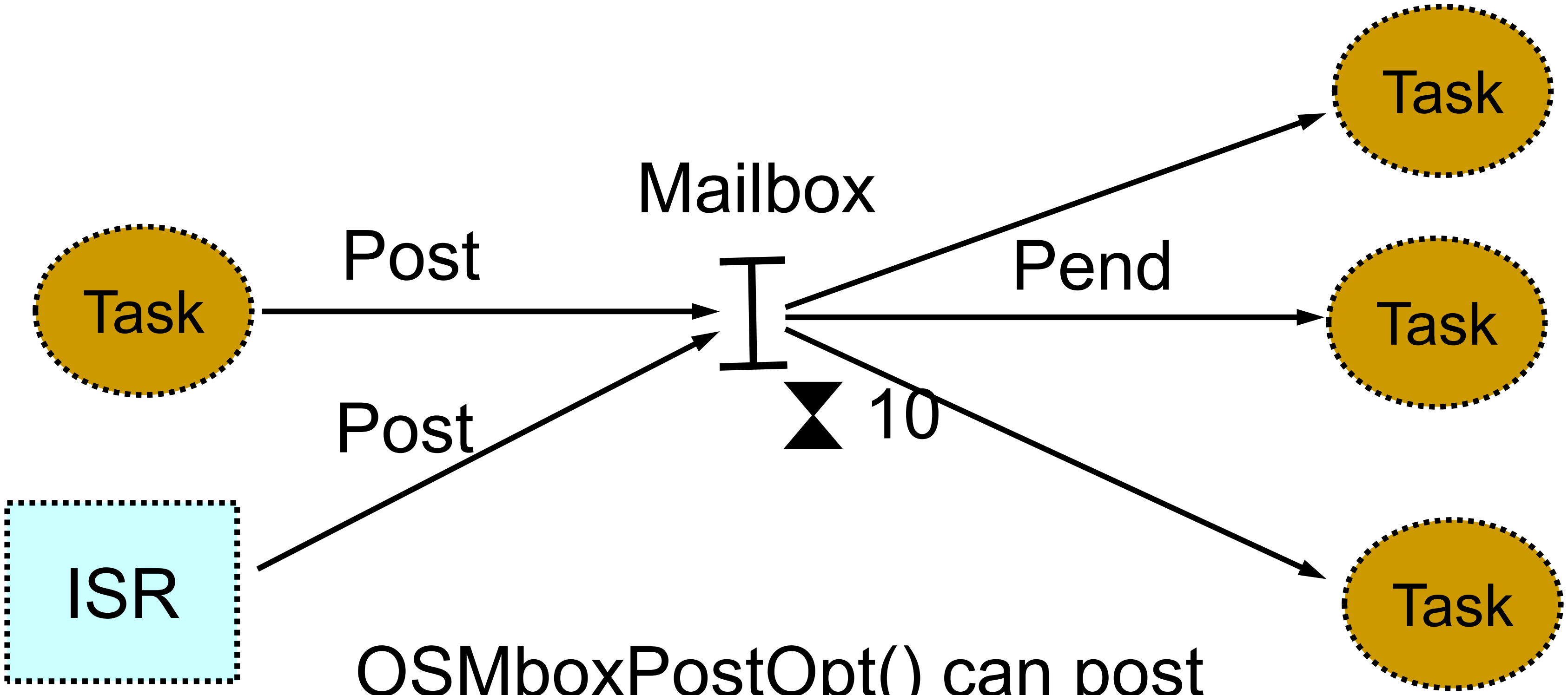
- OSMboxPend() is used when a task expects to receive a message. The message is sent to the task either by an ISR or by another task.
- The message received is a pointer-sized variable, and its use is application specific. If a message is present in the mailbox when OSMboxPend() is called, the message is retrieved, the mailbox is emptied, and the retrieved message is returned to the caller.
- If no message is present in the mailbox, OSMboxPend() suspends the current task until either a message is received or a user-specified timeout expires.
- If a message is sent to the mailbox and multiple tasks are waiting for the message, µC/OS-II resumes the highest priority task waiting to run. A pended task that has been suspended with OSTaskSuspend() can receive a message. However, the task remains suspended until it is resumed by calling OSTaskResume().

Message mailboxes

- INT8U **OSMboxPost** (OS_EVENT *pevent, void *msg);

- OSMboxPost() sends a message to a task through a mailbox. A message is a pointer-sized variable and, its use is application specific.
- If a message is already in the mailbox, an error code is returned indicating that the mailbox is full. OSMboxPost() then immediately returns to its caller, and the message is not placed in the mailbox.
- If any task is waiting for a message at the mailbox, the highest priority task waiting receives the message. If the task waiting for the message has a higher priority than the task sending the message, the higher priority task is resumed, and the task sending the message is suspended.
- In other words, a context switch occurs.

Message mailboxes



OSMboxPostOpt() can post
Mailbox to the highest priority waiting task
OR to ALL the waiting tasks

Message mailboxes

- INT8U **OSMboxPostOpt** (OS_EVENT *pevent, void *msg, INT8U opt);

- OSMboxPostOpt() works just like OSMboxPost() except that it allows you to post a message to **multiple** tasks. In other words, OSMboxPostOpt() allows the message posted to be broadcast to **all** tasks waiting on the mailbox. OSMboxPostOpt() can actually replace OSMboxPost() because it can emulate OSMboxPost().
- OSMboxPostOpt() is used to send a message to a task through a mailbox. A message is a pointer-sized variable, and its use is application specific.
- If a message is already in the mailbox, an error code is returned indicating that the mailbox is full. OSMboxPostOpt() then immediately returns to its caller, and the message is not placed in the mailbox.
- If any task is waiting for a message at the mailbox, OSMboxPostOpt() allows you either to post the message to the highest priority task waiting at the mailbox (opt set to OS_POST_OPT_NONE) or to all tasks waiting at the mailbox (opt is set to OS_POST_OPT_BROADCAST).
- In either case, scheduling occurs and, if any of the tasks that receives the message have a higher priority than the task that is posting the message, then the higher priority task is resumed, and the sending task is suspended. In other words, a context switch occurs.

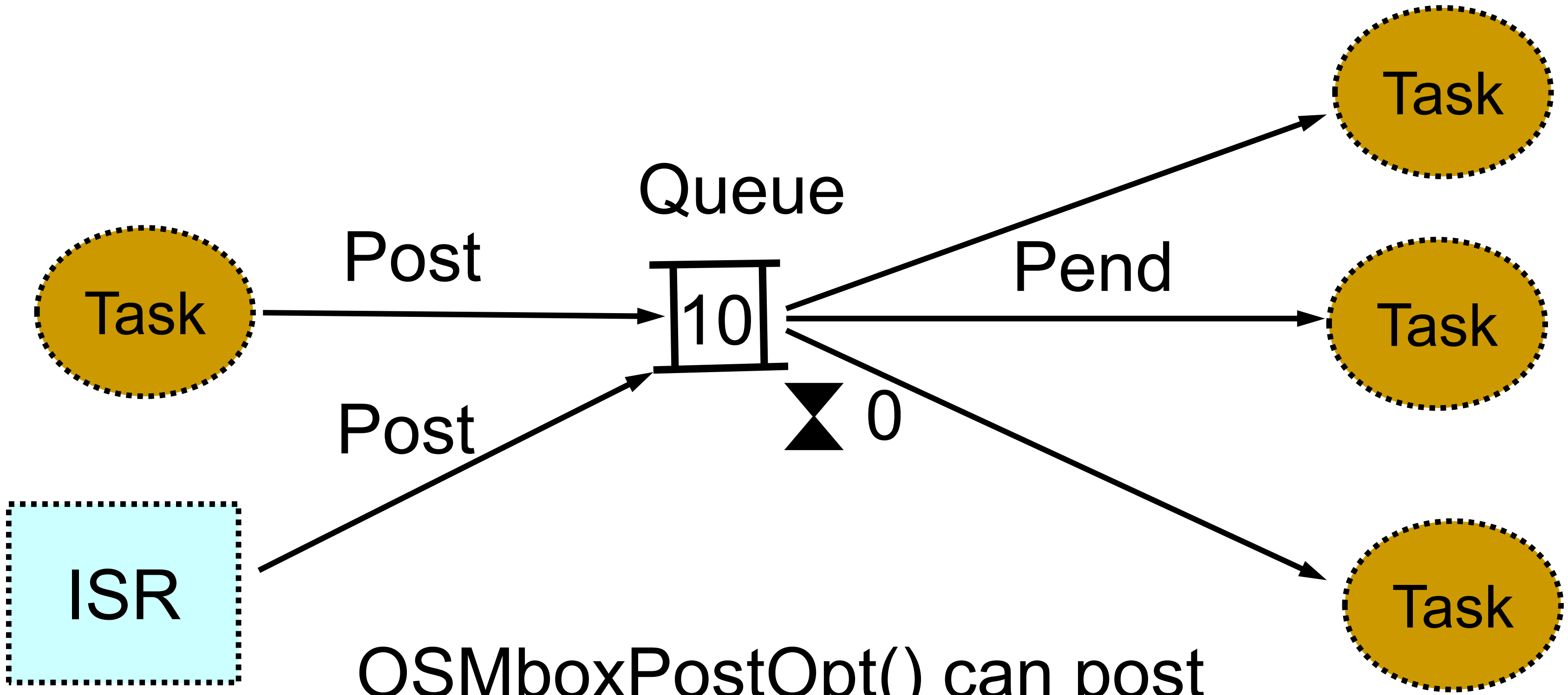
Message mailboxes

- Others OSMBox functions:
  - ➢OS_EVENT ***OSMboxCreate** (void *msg);
  - ➢OS_EVENT ***OSMboxDel** (OS_EVENT *pevent, INT8U opt, INT8U *err);
  - ➢ void ***OSMboxAccept** (OS_EVENT *pevent);
  - ➢ INT8U **OSMboxQuery** (OS_EVENT *pevent, OS_MBOX_DATA *pdata);
-

Message queues

- A message queue is similar to the mailbox, but it can accept more than 1 message. As many as the queue can accept them.
- The queue has to be provide when creating the message queue with OSQCreate()
- Void *MyArrayofMsg[SIZE];
- The array is seen as a circular buffer

Message queues



OSMboxPostOpt() can post
Mailbox to the highest priority waiting task
OR to ALL the waiting tasks

Message queues

- OS_EVENT ***OSQCreate** (void **start,
  INT8U size);
- OSQCreate() creates a message queue. A message queue allows tasks or ISRs to send pointer-sized variables (messages) to one or more tasks. The meaning of the messages sent are application specific.
  - **start** is the base address of the message storage area. A message storage area is declared as an array of pointers to voids.
  - **size** is the size (in number of entries) of the message storage area.

Message queues

- void ***OSQPend** (OS_EVENT *pevent, INT16U timeout, INT8U *err);
- OSQPend() is used when a task wants to receive messages from a queue. The messages are sent to the task either by an ISR or by another task.
- The messages received are pointer-sized variables, and their use is application specific.
- If at least one message is present at the queue when OSQPend() is called, the message is retrieved and returned to the caller.
- If no message is present at the queue, OSQPend() suspends the current task until either a message is received or a user-specified timeout expires.
- If a message is sent to the queue and multiple tasks are waiting for such a message, then µC/OS-II resumes the highest priority task that is waiting.
- A pended task that has been suspended with OSTaskSuspend() can receive a message. However, the task remains suspended until it is resumed by calling OSTaskResume().

Message queues

- INT8U **OSQPost** (OS_EVENT *pevent, void *msg);

- OSQPost() sends a message to a task through a queue.
- A message is a pointer-sized variable, and its use is application specific.
- If the message queue is full, an error code is returned to the caller. In this case, OSQPost() immediately returns to its caller, and the message is not placed in the queue.
- If any task is waiting for a message at the queue, the highest priority task receives the message.
- If the task waiting for the message has a higher priority than the task sending the message, the higher priority task resumes, and the task sending the message is suspended; that is, a context switch occurs.
- Message queues are **first-in first-out (FIFO),** which means that the first message sent is the first message received.

Message queues

- INT8U **OSQPostFront** (OS_EVENT *pevent, void *msg);

- OSQPostFront() sends a message to a task through a queue. OSQPostFront() behaves very much like OSQPost(), except that the message is inserted at the **front of the queue**.
- This means that OSQPostFront() makes the message queue behave like a **last-in first-out (LIFO)** queue instead of a first-in first-out (FIFO) queue.
- The message is a pointer-sized variable, and its use is application specific.
- If the message queue is full, an error code is returned to the caller. OSQPostFront() immediately returns to its caller, and the message is not placed in the queue.
- If any tasks are waiting for a message at the queue, the highest priority task receives the message.
- If the task waiting for the message has a higher priority than the task sending the message, the higher priority task is resumed, and the task sending the message is suspended; that is, a context switch occurs.

Message queues

- INT8U **OSQPostOpt** (OS_EVENT *pevent, void *msg, INT8U opt);

- OSQPostOpt() is used to send a message to a task through a queue. A message is a pointer-sized variable, and its use is application specific.
- If the message queue is full, an error code is returned indicating that the queue is full. OSQPostOpt() then immediately returns to its caller, and the message is not placed in the queue.
- If any task is waiting for a message at the queue, OSQPostOpt() allows you to either post the message to the highest priority task waiting at the queue (opt set to OS_POST_OPT_NONE) or to **all tasks waiting at the queue** (opt is set to OS_POST_OPT_BROADCAST).
- In either case, scheduling occurs, and, if any of the tasks that receive the message have a higher priority than the task that is posting the message, then the higher priority task is resumed, and the sending task is suspended. In other words, a context switch occurs.

Message queues

- OSQPostOpt() emulates both OSQPost() and OSQPostFront() and also allows to post a message to **multiple** tasks. In other words, it allows the message posted to be broadcast to **all** tasks waiting on the queue. OSQPostOpt() can actually replace OSQPost() and OSQPostFront() because the mode of operation is specified via an option argument, *opt*. Doing this allows you to reduce the amount of code space needed by µC/OS-II.

Message queues

- **pevent** is a pointer to the queue. This pointer is returned to your application when the queue is created [see OSQCreate()].
- **msg** is the actual message sent to the task(s). msg is a pointer-sized variable, and what msg points to is application specific. As of V2.60, you are now allowed to post a NULL pointer.
- opt determines the type of POST performed:
  - ➢ **OS_POST_OPT_NONE** POST to a single waiting task [identical to OSQPost()].
  - ➢ **OS_POST_OPT_BROADCAST** POST to **all** tasks waiting on the queue.
  - ➢ **OS_POST_OPT_FRONT** POST as **LIFO** [simulates OSQPostFront()].
  - ➢ Below is a list of **all** the possible combination of these flags:
  - ➢ OS_POST_OPT_NONE is identical to OSQPost()
  - ➢ OS_POST_OPT_FRONT is identical to OSQPostFront()
  - ➢ OS_POST_OPT_BROADCAST is identical to OSQPost() but broadcasts msg to **all** waiting tasks
  - ➢ OS_POST_OPT_FRONT + OS_POST_OPT_BROADCAST
  - ➢ is identical to OSQPostFront() except that broadcasts msg to **all** waiting tasks.

Message queues

- INT8U ***OSQFlush** (OS_EVENT *pevent);
- INT8U **OSQQuery** (OS_EVENT *pevent, OS_Q_DATA *pdata);
- OS_EVENT ***OSQDel** (OS_EVENT *pevent, INT8U opt, INT8U *err);

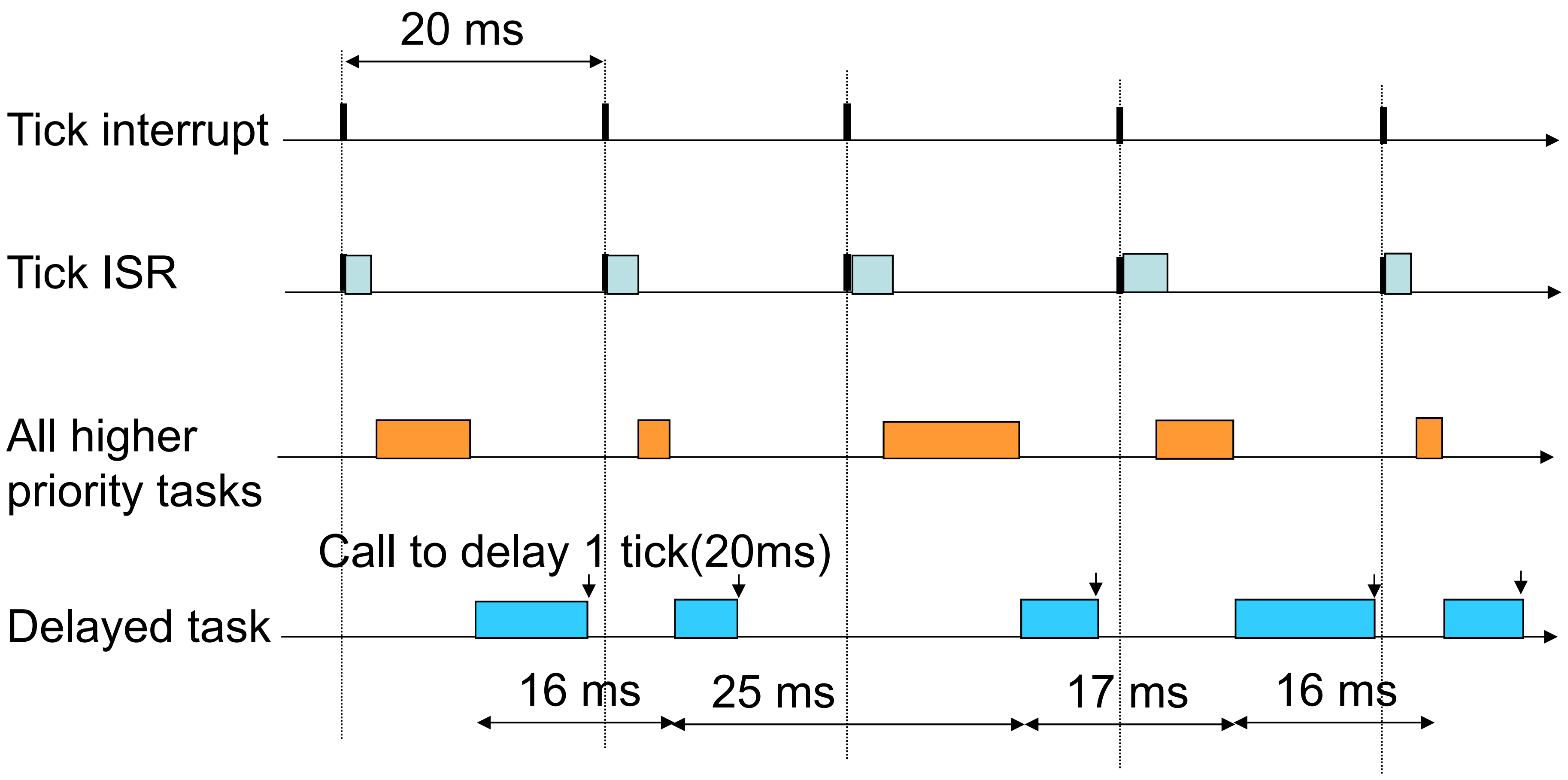
Clock Tick

- A *Clock Tick* is a special interrupt that occurs periodically.
- A Timer provide this interrupt
- It can be used for delay, for time-out
- Faster the clock tick higher the overhead imposed to the system in general → 10..200ms
- The resolution is one clock tick
- The accuracy is NOT one clock tick

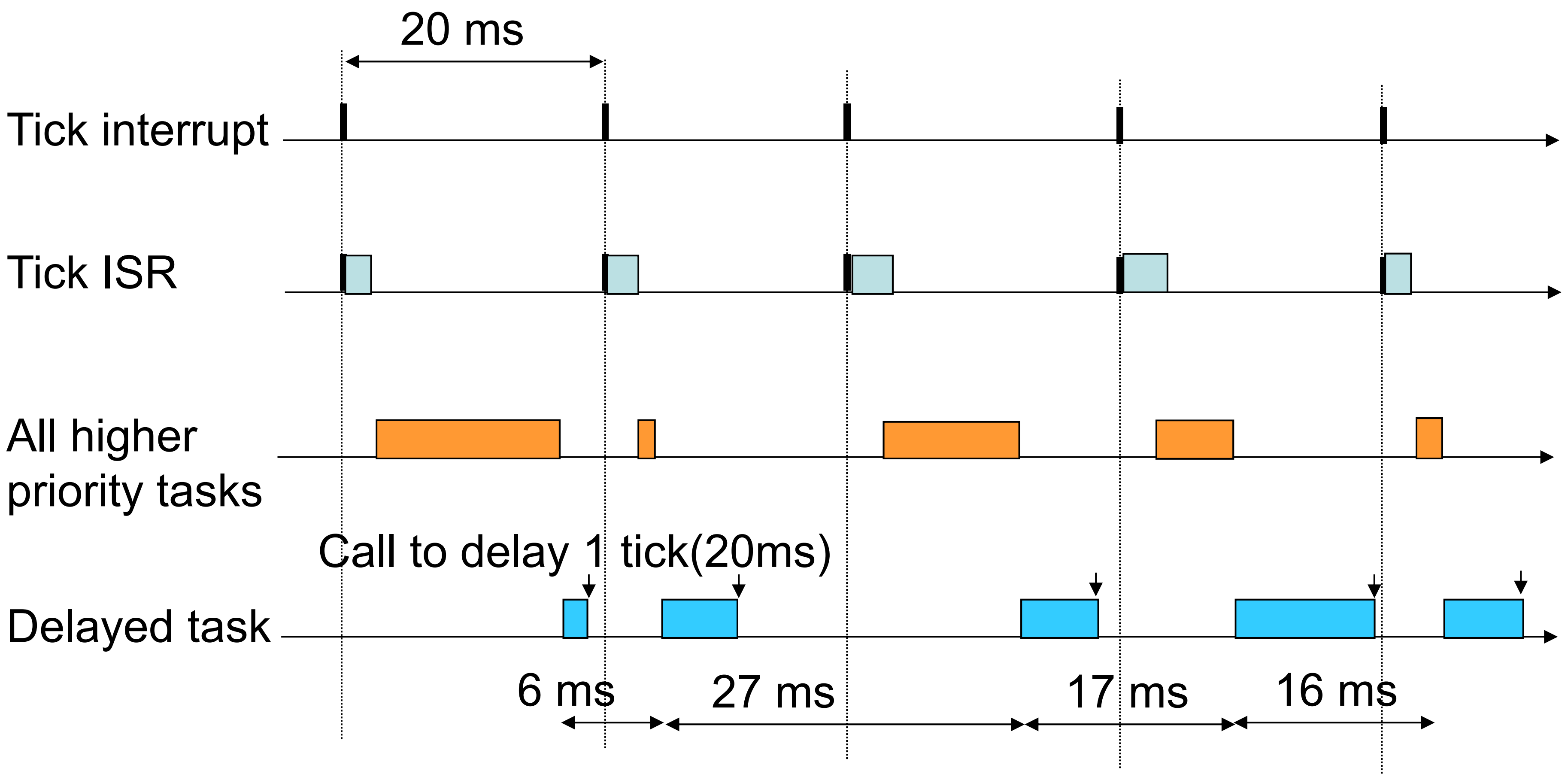
Clock Tick

- An example of a low priority task delayed by 1 tick in different timing configuration
- In all case the real waiting time is NOT exactly a multiple of 1 tick,
- There is a **jitter** in the execution starting time !
- It could be less than 1 tick or more than 1 tick !

Clock Tick


20 ms
Tick interrupt
Tick ISR
All higher priority tasks
Call to delay 1 tick(20ms)
Delayed task
16 ms
25 ms
17 ms
16 ms

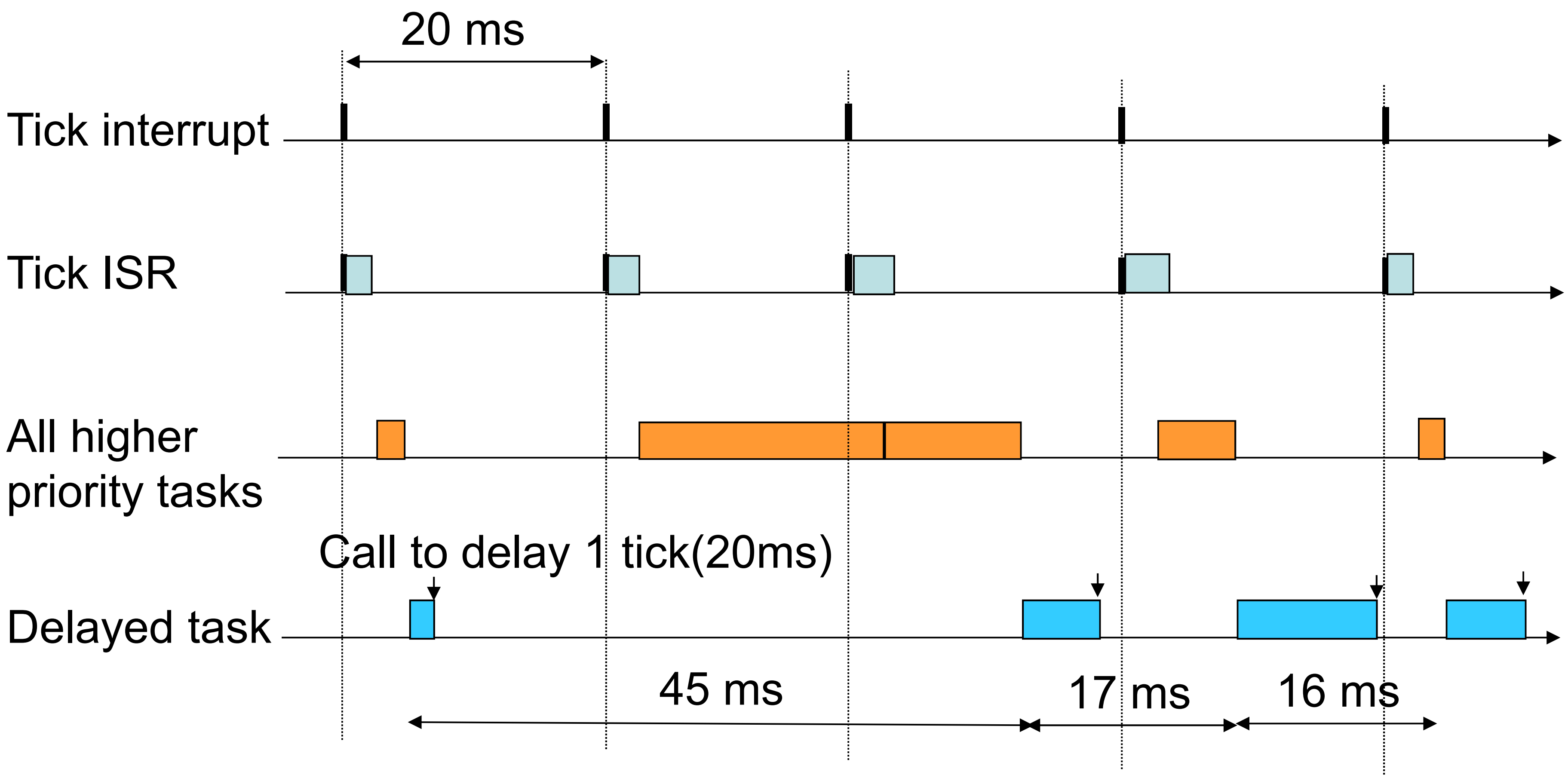
Clock Tick


20 ms
Tick interrupt
Tick ISR
All higher priority tasks
Call to delay 1 tick(20ms)
Delayed task
6 ms
27 ms
17 ms
16 ms

Clock Tick

- If there is not enough time in a tick slot to finish all the higher priority task, the waiting time can be far more than 1 tick !
- The deadline can be missed !
- Acceptable in certain applications, not in others !

Clock Tick


20 ms
Tick interrupt
Tick ISR
All higher priority tasks
Call to delay 1 tick(20ms)
Delayed task
45 ms
17 ms
16 ms

Kernel Structure

Example from MicroC/OS-II

Task Control Bloc

Event Control Block

Task Control Blocks

- For each task created, a task bloc is take from a list of free TCB
- A TCB is a data structure used to maintain the state of a task when it's preempted
- A TCB contain all the information about a task for a resume and to continue again it's execution

Task Control Blocks

- A list of Free TCB is available at starting
- Each time a task is created, a TCB is take from the free list
- Each task has a unique priority level (0..60) or (0..252)
- Idle task has the lowest priority
- Static task allows the calculation of statistics on the tasks timings

Task Control Blocks

```
*********************************************************************************************************
*                                           TASK CONTROL BLOCK
*********************************************************************************************************
*/

typedef struct os_tcb {
    OS_STK          *OSTCBStkPtr;      /* Pointer to current top of stack                              */

#if OS_TASK_CREATE_EXT_EN > 0
    void            *OSTCBExtPtr;      /* Pointer to user definable data for TCB extension             */
    OS_STK          *OSTCBStkBottom;   /* Pointer to bottom of stack                                   */
    INT32U           OSTCBStkSize;     /* Size of task stack (in number of stack elements)             */
    INT16U           OSTCBOpt;         /* Task options as passed by OSTaskCreateExt()                  */
    INT16U           OSTCBId;          /* Task ID (0..65535)                                           */
#endif

    struct os_tcb   *OSTCBNext;        /* Pointer to next     TCB in the TCB list                      */
    struct os_tcb   *OSTCBPrev;        /* Pointer to previous TCB in the TCB list                      */

#if OS_EVENT_EN
    OS_EVENT        *OSTCBEventPtr;    /* Pointer to event control block                               */
#endif

#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0)
    void            *OSTCBMsg;         /* Message received from OSMboxPost() or OSQPost()              */
#endif

#if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
#if OS_TASK_DEL_EN > 0
    OS_FLAG_NODE    *OSTCBFlagNode;    /* Pointer to event flag node                                   */
#endif
    OS_FLAGS         OSTCBFlagsRdy;    /* Event flags that made task ready to run                      */
#endif
```

Task Control Blocks

```
INT16U          OSTCBDly;        /* Nbr ticks to delay task or, timeout waiting for event            */
    INT8U           OSTCBStat;       /* Task status                                                      */
    BOOLEAN         OSTCBPendTO;     /* Flag indicating PEND timed out (OS_TRUE == timed out)            */
    INT8U           OSTCBPrio;       /* Task priority (0 == highest)                                     */

    INT8U           OSTCBX;          /* Bit position in group  corresponding to task priority            */
    INT8U           OSTCBY;          /* Index into ready table corresponding to task priority            */
#if OS_LOWEST_PRIO <= 63
    INT8U           OSTCBBitX;       /* Bit mask to access bit position in ready table                   */
    INT8U           OSTCBBitY;       /* Bit mask to access bit position in ready group                   */
#else
    INT16U          OSTCBBitX;       /* Bit mask to access bit position in ready table                   */
    INT16U          OSTCBBitY;       /* Bit mask to access bit position in ready group                   */
#endif

#if OS_TASK_DEL_EN > 0
    INT8U           OSTCBDelReq;     /* Indicates whether a task needs to delete itself                  */
#endif

#if OS_TASK_PROFILE_EN > 0
    INT32U          OSTCBCtxSwCtr;    /* Number of time the task was switched in                         */
    INT32U          OSTCBCyclesTot;   /* Total number of clock cycles the task has been running          */
    INT32U          OSTCBCyclesStart; /* Snapshot of cycle counter at start of task resumption           */
    OS_STK         *OSTCBStkBase;     /* Pointer to the beginning of the task stack                      */
    INT32U          OSTCBStkUsed;     /* Number of bytes used from the stack                             */
#endif

#if OS_TASK_NAME_SIZE > 1
    INT8U           OSTCBTaskName[OS_TASK_NAME_SIZE];
#endif
} OS_TCB;
```

ECB: Event Control Blocks

- a semaphore, a mutex, a flag, a mailbox, a queue are considered as Event
- Signal can be done on a Event and control transferred to a waiting task on this event
- The signal can be done from an ISR or from an other task
- The ECB is a data structure to handle the events: OS_EVENT from ucos_ii.h

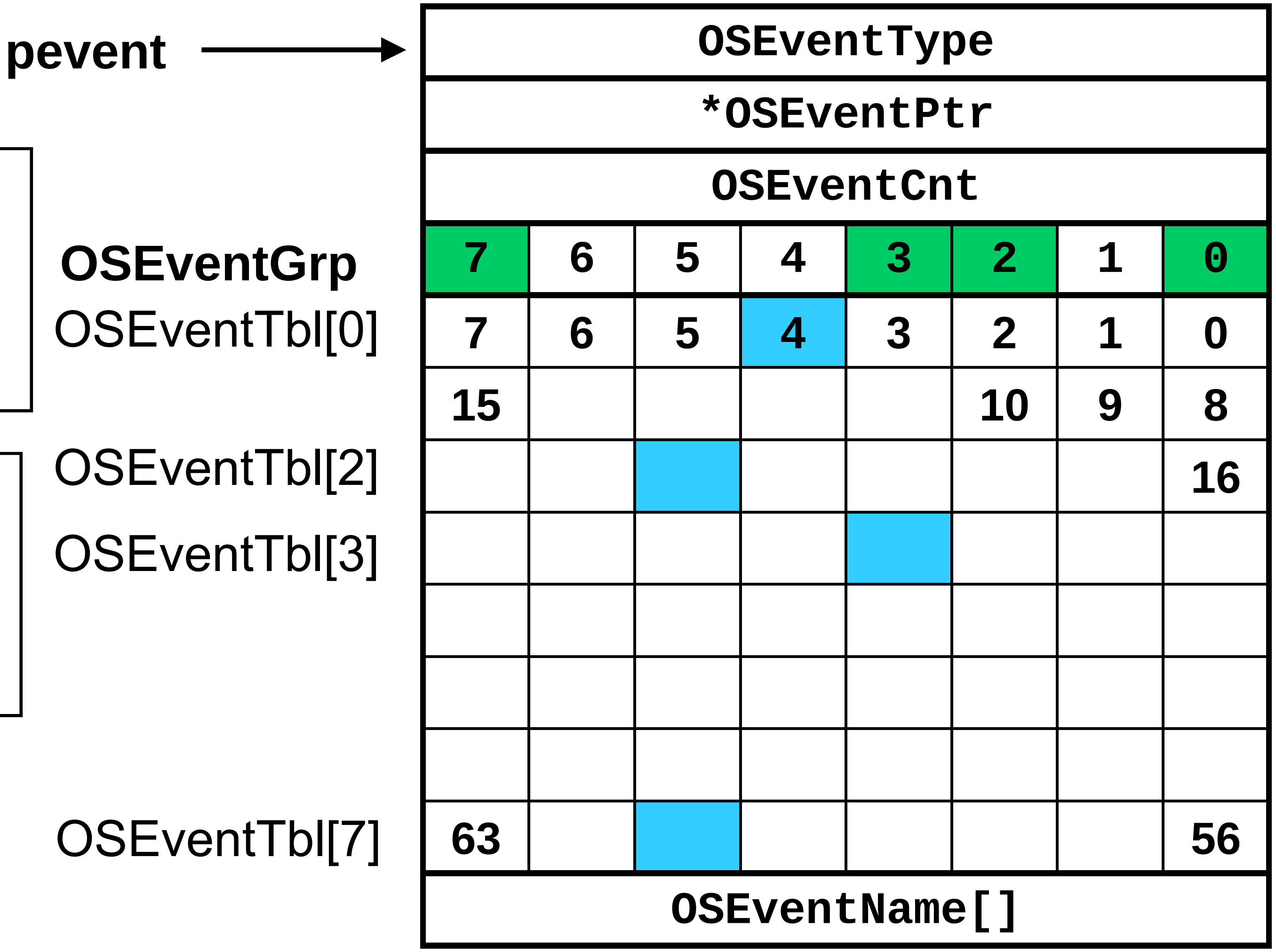
ECB: Event Control Blocks

- `OSEventType` specify the Type of Event as:
  - a semaphore,
  - a mutex,
  - a mailbox,
  - a queue
- `OSEventPtr` is a pointer for mailbox and message queue
- `OSEventCnt` is used for the semaphore counter
- `OSEventGrp` and `OSEventTbl[OS_EVENT_TBL_SIZE]` are used for priority waiting for the associated event
- `OSEventName[OS_EVENT_NAME_SIZE]` allow to specify a name to the event

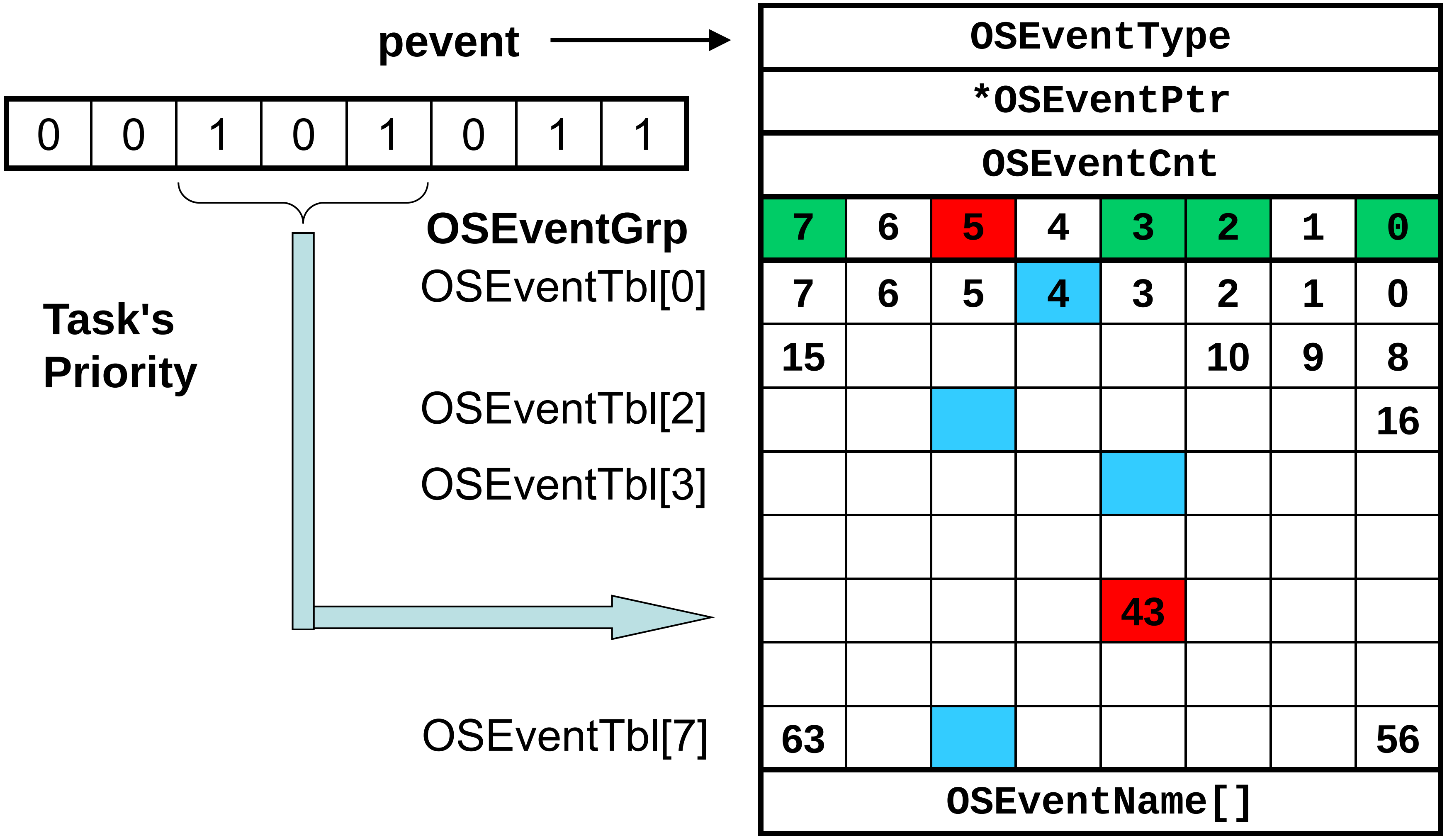
ECB: Event Control Blocks

Offset in OSEvenTbl waiting for the event

Priorities of all tasks waiting for the event, Activated by a '1'



ECB: Event Control Blocks



ECB: Event Control Blocks

```
*********************************************************************************************
*                                          EVENT CONTROL BLOCK
*********************************************************************************************

#if OS_EVENT_EN && (OS_MAX_EVENTS > 0)
typedef struct os_event {
    INT8U    OSEventType;                    /* Type of event control block (see OS_EVENT_TYPE_xxxx)    */
    void    *OSEventPtr;                     /* Pointer to message or queue structure                   */
    INT16U   OSEventCnt;                     /* Semaphore Count (not used if other EVENT type)          */
#if OS_LOWEST_PRIO <= 63
    INT8U    OSEventGrp;                     /* Group corresponding to tasks waiting for event to occur */
    INT8U    OSEventTbl[OS_EVENT_TBL_SIZE];  /* List of tasks waiting for event to occur                */
#else
    INT16U   OSEventGrp;                     /* Group corresponding to tasks waiting for event to occur */
    INT16U   OSEventTbl[OS_EVENT_TBL_SIZE];  /* List of tasks waiting for event to occur                */
#endif

#if OS_EVENT_NAME_SIZE > 1
    INT8U    OSEventName[OS_EVENT_NAME_SIZE];
#endif
} OS_EVENT;
#endif
```

ECB: Event Control Blocks

- An ECB is to be reserved for each event to create:
  - OS_EVENT MySemaphore;   //
  - MySemaphore = OSSemCreate(1);

  - OS_EVENT MyMailbox;
  - …

EFG: Event Flag Group

- For the Flags, the structure is OS_FLAG_GRP

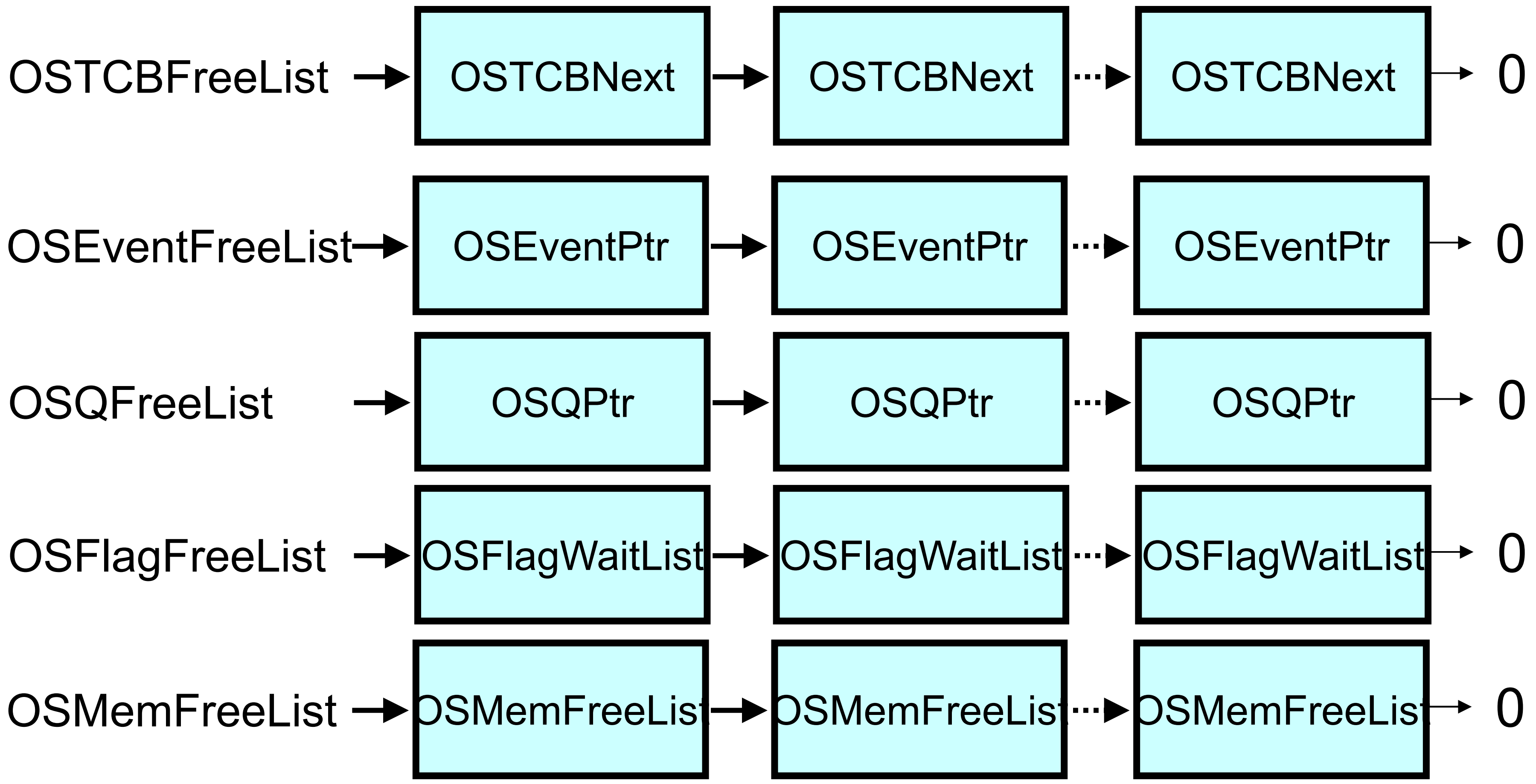
```
/*
*********************************************************************************************************
*                                      EVENT FLAGS CONTROL BLOCK
*********************************************************************************************************
*/

#if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)

#if OS_FLAGS_NBITS == 8                     /* Determine the size of OS_FLAGS (8, 16 or 32 bits)       */
typedef  INT8U    OS_FLAGS;
#endif

#if OS_FLAGS_NBITS == 16
typedef  INT16U   OS_FLAGS;
#endif

#if OS_FLAGS_NBITS == 32
typedef  INT32U   OS_FLAGS;
#endif


typedef struct os_flag_grp {                /* Event Flag Group                                        */
    INT8U         OSFlagType;               /* Should be set to OS_EVENT_TYPE_FLAG                     */
    void         *OSFlagWaitList;           /* Pointer to first NODE of task waiting on event flag     */
    OS_FLAGS      OSFlagFlags;              /* 8, 16 or 32 bit flags                                   */
#if OS_FLAG_NAME_SIZE > 1
    INT8U         OSFlagName[OS_FLAG_NAME_SIZE];
#endif
} OS_FLAG_GRP;
```

ECB: Event Control Blocks

- Associated with OS_FLAG_NODE

```
typedef struct os_flag_node {                  /* Event Flag Wait List Node                            */
    void          *OSFlagNodeNext;             /* Pointer to next     NODE in wait list                */
    void          *OSFlagNodePrev;             /* Pointer to previous NODE in wait list                */
    void          *OSFlagNodeTCB;              /* Pointer to TCB of waiting task                       */
    void          *OSFlagNodeFlagGrp;          /* Pointer to Event Flag Group                          */
    OS_FLAGS       OSFlagNodeFlags;            /* Event flag to wait on                                */
    INT8U          OSFlagNodeWaitType;         /* Type of wait:                                        */
                                               /*      OS_FLAG_WAIT_AND                                */
                                               /*      OS_FLAG_WAIT_ALL                                */
                                               /*      OS_FLAG_WAIT_OR                                 */
                                               /*      OS_FLAG_WAIT_ANY                                */
} OS_FLAG_NODE;
#endif
```

Free Pools after OSInit()



A Task that run indefinitely

- **A Task has the structure like this one**

```
1.  void Task1(void *pdata){
    {
        for(;;){
            //One of uC/OS-II's services :
            OSFlagPend();
            OSMboxPend();
            OSMutexPend();
            OSQPend();
            OSSemPend();
            OSTaskSuspend();
            OSTimeDly();
            OSTimeDlyHMSM();
        }
    }
```

A Task than run for a limited time

- **Or this one**

1. void Task2(void *pdata){

```
{
    // USER CODE
    OSTaskDel();
}
```