ELEC-H410
REAL-TIME SYSTEMS

Chapter 5

Priority-driven scheduling

Priority-driven scheduling

CONTENTS

- **foreground/background systems**
- RTOS
- conclusions

Foreground/background

asynchronous *event-driven* systems

- purpose : schedule jobs associated to asynchronous events and their deadlines D_i
- method
  - clock-driven (synchronous) scheduling with $T_s < \min(D_i)$ (see previous chapter)
  - asynchronous scheduling
    - associate an IRQ to each event
    - define the priority (by hard and/or software) for each IRQ
- remark
  - periodic tasks can be handled by periodic IRQ even if the global systen is asynchronous and dynamic

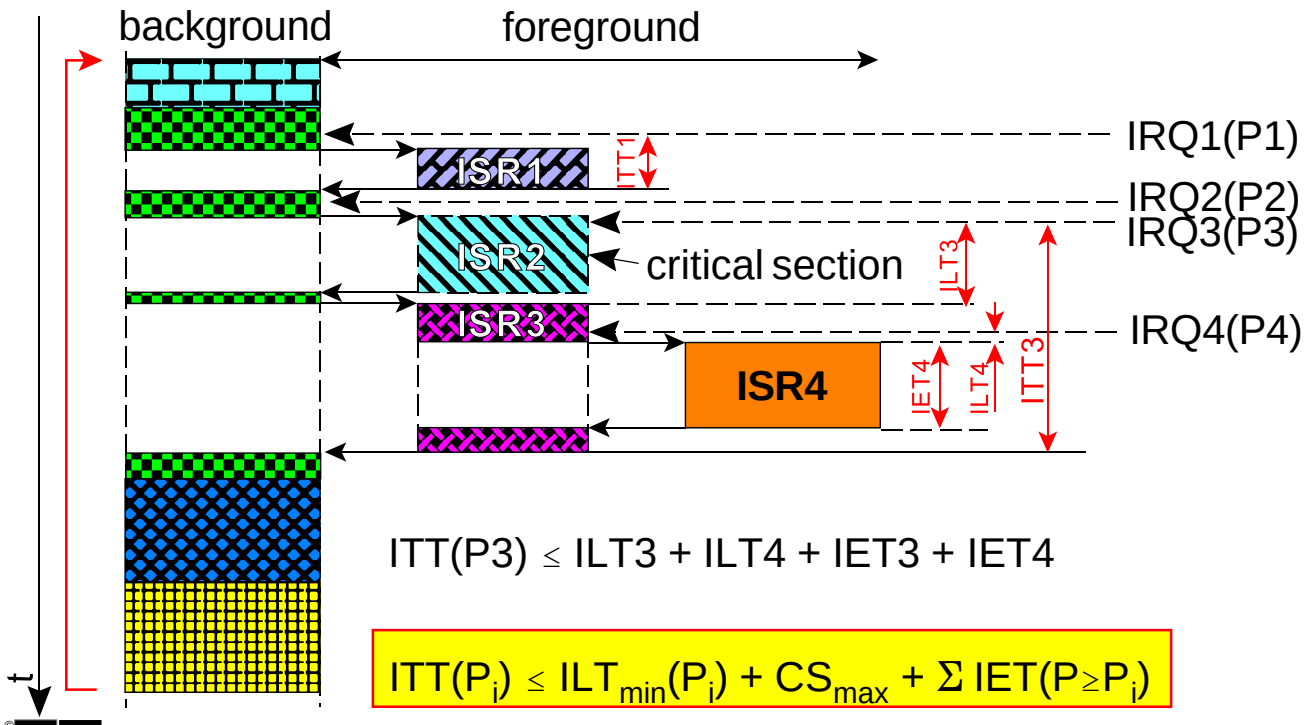
Schedulers that are not based on a clock are thus **asynchronous** and also known as **event-driven**. The task scheduling consists in:

- detecting these events, by interrupt requests in most cases
- defining a priority of each interrupt, either by the help of a special peripheral called PIC (Priority Interrupt Controller) or by software, or by a combination of hardware and software

It is also possible to manage **periodic tasks** in an asynchronous system, via **periodic interrupts** produced by a timer, while keeping an asynchronous philosophy based on the priorities.

Foreground/background

worst-case Interrupt Total Time



In the foreground/background system, the program executes a **background infinite loop** of tasks without any priority (or having all the least priority 0). This is basically the same concept as the superloop that we have seen at the beginning of the previous chapter. The background can also be empty; in this case the main() program contains only initialization instructions, then an empty loop while(true) {}. On this figure, the background consists in 4 jobs.

The **foreground** is a set of periodic and aperiodic events coming from external interrupt requests (I/O pins) or from internal peripherals such as timers, analog-to-digital converters and serial communication devices. Interrupts are here numbered in increasing order of priorities. Let us analyse this figure

- the interrupt request IRQ1 occurs; after the latency time, the execution of the service routine ISR1 starts; ISR1 unmasks IRQ2, IRQ3 and IRQ4, whose priority is higher. The total execution time is ITT1; it is the best case, because no higher priority events occurred.

- IRQ2, being a critical section, leaves the global interrupt mask set and executes normally. Let us notice that a *critical section does not imply the highest priority* , because IRQ2 cannot preempt IRQ3 and IRQ4.

- a worse case happens to IRQ3, indeed
  - IRQ3 occurs just after the beginning of ISR2, in critical section; the interrupt flag of IRQ3 is set, but ISR3 is delayed; the execution time of ISR2 increases the latency time of IRQ3
  - ISR2 terminates by Return from Interrupt, the processor restores the context and returns to the background (rem:most processors are designed to guarantee the execution of the first instruction of the code which was interrupted, in particular to enable a modification of the interrupt masks).
  - since ISR3 is pending, only 1 instruction of the background is executed, then ISR3 is called; one of its first instructions unmasks ISR4. ISR3 executes, but is preempted by IRQ4 whose execution time delays ISR3

In the worst-case, the total time of an interruption (normal latency time + execution time) can be increased by
- the execution time of the longest critical section
- the execution time of all interrupts which have got a higher priority

Foreground/background

put the minimum of code in an ISR !

- too long ISR => latency ↗ for other IRQ
  - just do what is essential
    - if low priority: enable higher priority IRQs
    - execute most urgent actions
    - prepare further background actions
      - set a boolean to indicate that the IRQ occurred
      - prepare data
  - delayed actions are performed later in background
- shorter deadline
  - set higher priority
  - critical section (/!\ iif required)
  - if deadline cannot be respected => subcontract to hardware (other µC with adapted peripheral, FPGA,...)
- potential safety problem => redundancy

After the conclusion of the previous slide, it is clear that we have to **put the minimum of code in ISRs** to avoid slowing down other events.

Let us take the example of a task responsible for filling a tank; when the level is reached, a float closes a switch which causes an interrupt request. The ISR will
- stop the pump which fills the tank
- set a flag (global variable) to warn a background task that the tank is full
- the background task will change the level and color of the tank on the display

The following rules should be applied
- the total response time to this event must be shorter than the deadline (i.e. the delay between the closing of the micro-switch and the overflow of the tank); if it cannot be guaranteed, it is necessary to resort to a hardware solution to cut off the supply of the tank; the interruption serves only as level indicator
- if the overflow is critical for safety, redundancy should be introduced

Foreground/background

conclusions

- difficulties of fore/background systems
  - manage priorities/critical sections
    - a lot of IRQ sources
    - not enough priority levels in the system
    - priorities partially imposed by the platform
  - ensure operation in all the cases
  - the problem has to be solved for each new application
  - if an IRQ is mostly treated in background, the response time to an event is the worst-case length of the loop
- easier solution: **RTOS** (RTK)
  Real Time Operating System (Kernel)

The main difficulty is thus the choice of the priorities of interrupt requests and of the critical sections. It is a hard work for the programmer, in particular because
- the number of events can be larger than the number of interrupts request inputs (fortunately, modern micro-controllers offer a lot of interrupt input pins)
- the number of priority levels of the processor is not always sufficient (e.g. 2 for 8051 and PIC18 families) most of the priority management has to be done in software
- the priorities can partially be fixed by construction of the processor (see 8051 family)
- some interrupts are reserved by the platform itself for its own peripherals.

In the worst case, the tasks treated in background (including those started by an interruption) have a processing time equal to that of the whole loop, and this time is not constant.

Finally the experience acquired during a design cannot necessarily be transposed to future cases. It is thus necessary to remake a similar effort and to seek for worst cases with each new application.

To answer this problem, special Operating Systems (OS) have been designed to perform dynamic scheduling and try to meet all the temporal constraints of real-time tasks. Such OSes are known as

RTOS ***Real Time Operating System*** or
RTK ***Real Time Kernel*** when they are reduced to scheduling and drivers of the peripherals (no file management is required in systems without mass memory)

Priority-driven scheduling

CONTENTS

- foreground/background systems
- **RTOS**
  - **principles**
  - inter-task communication
  - time *ticks*
  - managing interrupts
  - main scheduling algorithms
- conclusions

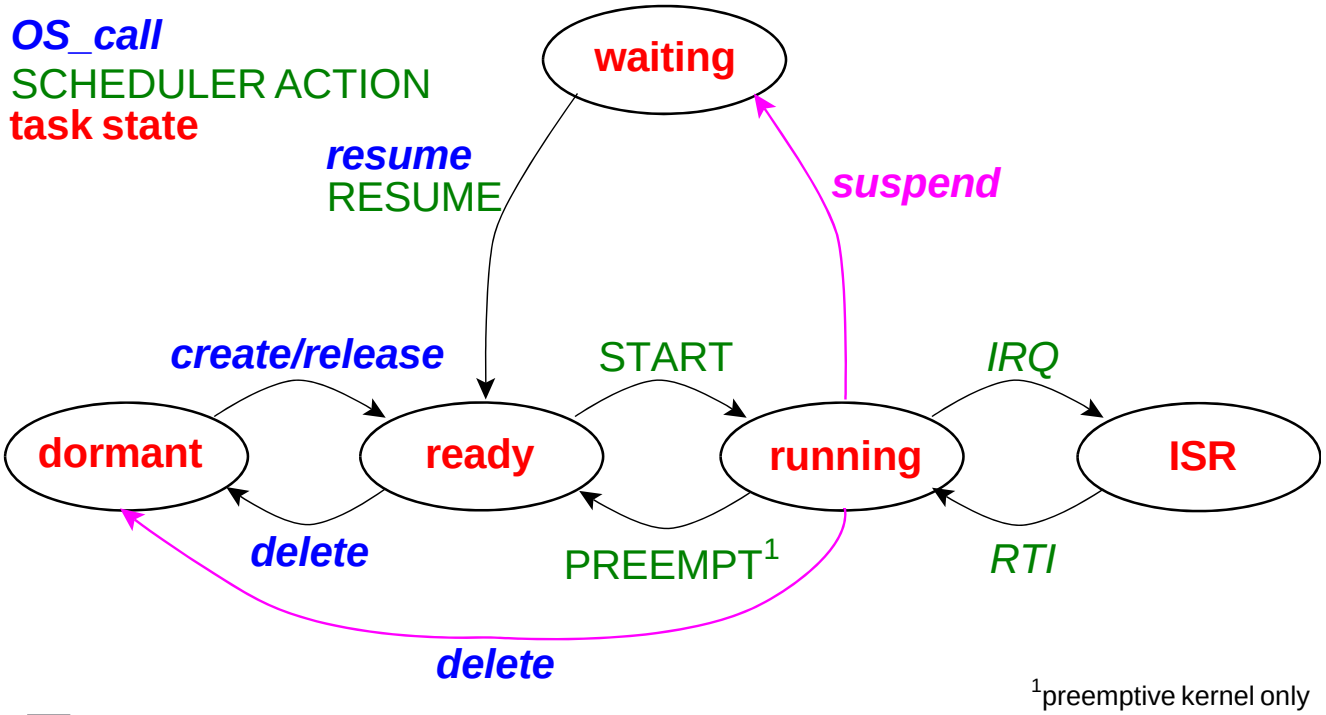
RTOS

principles

- each task
  - is written as if it were alone to execute
  - has got a priority level dictated by the application
  - is given resources when it is created
    - registers
    - dynamic memory: stack, heap
- at run-time, the scheduler
  - tries to respect priorities and deadlines
  - runs 1 task at a time by giving it
    - some registers
    - the CPU
- how to return to the scheduler : 2 variants
  - non-preemptive: the task gives the control back
  - preemptive: the task gives the control back or the scheduler is able to preempt the task

RTOS

state of tasks and transitions



We have already seen this diagram and the different definitions.

In **non-preemtive** scheduling a **running task** has to **give the control back to the scheduler** (which is also known as **cooperative multi-tasking**)

Two cases appear on this figure
- the running task is finished and makes a call to the OS to be **deleted** to return to the "dormant" state; this is only true for tasks that are rarely started and run only once (e.g. maintenance of the code, diagnosis)
- a recurrently running task **suspends** itself to a new state called the **"waiting"** state, because it wants to wait for
  - a certain **amount of time** (it is a means for a task to become periodical) or until a certain **date**
  - **data** coming from another task (see further: mailboxes)
  - a **flag** set by another task or an ISR to synchronize (see further: semaphores)
  - a **resource** which is momentarily not available (communication port, buffer,... see further: semaphore)s

In this diagram, a preemptive scheduler adds the transition PREEMPT from "running" to "ready", which happens when a higher priority task becomes "ready".
This is quite a fundamental difference: **the preemptive scheduler tries to have always the highest priority task running.**

The management of the interrupts will be detailed later in this chapter.

Priority-driven scheduling

CONTENTS

- foreground/background systems
- **RTOS**
  - principles
  - **inter task communication**
    - **shared global variables**
    - **semaphores**
    - **mailboxes**
    - dependancy hazards
  - time *ticks*
  - managing interrupts
  - main scheduling algorithms
- conclusions

RTOS: task communications

shared global variables: efficient, but dangerous

- definition
  - global variables are declared outside any function
- advantage
  - visible anytime, anywhere => observability in debug
  - efficient and fast
    - inherently shared
    - no time required to make a copy
    - no need to pass parameters to a function using it
- **/!\ danger**
  - anarchic sharing and corruption of data
  - static mechanism => **non-reentrant**

When variables are defined outside any function, they become **global**.

This seems an enormous **advantage** if you want to share their value between several functions, since they are **usable anywhere in the code**. Because of this observability, the value of a global variable can easily be monitored in a "watch window" during debug.

The **potential danger** is that they can be changed by several functions, which can lead to a concurrence and to **anarchic situations** where the behaviour on the program depends on the competition between several functions, or from the order of their execution.

The most typical example is a function F whose behaviour depends on the value of a global variable G. F begins its execution then and is interrupted or preempted by another function which changes G. When F resumes its execution later, G has been changed (i.e. the context of F is not restored properly) and F won't behave in the same way, which could be harmful.

Using a **global variable** to share data is said to be a **static mechanism**, because G is assigned a fixed address that won't change during the whole execution. Therefore F can be entered, but not interrupted and re-entered; this is called **non-reentrant code**.

During debugging, if the value of the global variable becomes incorrect, the fact that it can be changed from anywhere in the code complicates the identification of the error.

Therefore, **global variables are prohibited in most "good programming practice".** For some programmer it is real paranoia, which is unjustified because **there are safe ways to use global variables**, which are very efficient in embedded systems, which we shall see in the next slides.

RTOS: task communications

shared global variables CAN be safe

- pure read-only globals
  - process state variables are globals
    - changed by the process (T^o, P, switch,....)
    - stored in input ports and analog-to-digital converters
    - "read-only"
- almost read-only globals
  - single "producer" function can write to the global, other ones are read-only "consumers"
- non-reentrant => mutual exclusion required
  - read: prohibit interrupts (critical section)
  - write: use uninterruptible instructions (read-modify-write)
  - RTOS=>semaphores

- the **safest globals are read-only**. The most common example is the set of **physical state variables** (temperature, pressure, speed, position) of you process; they are inherently global and **changed by the process and not by your code**. They are acquired by the input ports (analog or digital) of your system which are global by definition: their name (and hence their address) is known by the compiler. Using them in a function does not require to pass the value as a parameter, which spares time and memory space for a copy of the variable or for its address). Just use the variable by its name, it's simple and it's completely safe.

- if only **one function** (called the"producer") **can change** the variable, it becomes read-only for all other functions (the "consumers"); in that case, the debugging is easier because there is only one place in the code where the variable is assigned its value.

- anyway, the sharing mechanism is still non-reentrant. To avoid reentrance problems, one solution is to avoid being interrupted or preempted is to access the global variable in a **critical section** (i.e. mask the interrupts). In most microcontrollers, the instructions that set or reset bits of variables (called read-modify-write) though they require 3 operations (**read-modify-write**) are single, **uninterruptible** instructions, which simplifies the problem.

- if we use an RTOS, a protection for shared resources is provided: the **semaphores**

RTOS: task communications

semaphores

- a semaphore is
  - a protocol for multitasking resource sharing or synchronisation
  - an object (also called event)
    - belonging to the kernel
    - created by a call OSSemCreate() in the initialization code
- a semaphore contains
  - a counter which is the "value" of the semaphore and initialized when semaphore is created
    0 means semaphore not available
  - a list of tasks waiting for the availability of the semaphore
- 3 types
  - counting: initialized to any integer value
  - binary: takes only values 0 or 1
    - key : initialized to 1
    - flag : initialized to 0

RTOS: task communications

how a task asks and gets a semaphore

- task calls **OSSemPEND**(SemName,timeout,&err)
- *if* semaphore value>0 the RTOS
  - decrements the semaphore
  - returns to the running task
- *else* the RTOS
  - changes task state to "waiting"
  - *if* timeout==0 task is ready to wait indefinitely
  - *else*
    - a software timer is initialized in the TCB of the task
    - scheduler makes a context switch and starts highest priority task
- special case **an ISR cannot wait for a semaphore**
  - no "waiting" state
  - use OSSemAccept() just to get the value (ISR not suspended)
  - test if it is 0 (or not) within the ISR

For the running task, **asking for a semaphore consists in calling a function of the RTOS** with the name of the semaphore as a parameter. We will use here the grammar of uCOSII, which you will use in the labs.

By calling the RTOS, the running task gives back the processor to the RTOS. The fact that the RTOS will give back the processor to the task depends on the state of the semaphore.

- if the semaphore is available, the RTOS will decrement its counter and the task will resume its execution

- if the semaphore is not available, the task will go to the waiting state. How long the task will wait depends on the second parameter, which is a **time-out**

    - if the time-out is infinite, the task is ready to wait until the semaphore becomes available. At that moment, the task becomes "ready" and finally "running" when its priority is the highest

    - if the time-out is finite, then the RTOS will create a **software timer** in the data structure associated to the task and called TCB (Task Control Block). The system is regularly interrupted by a hardware timer, those interrupts are called the ticks of the RTOS. At each tick, the ISR belonging to the RTOS decrements the timers in all TCBs. If the timer associated to the semaphore arrives at 0 before the semaphore is available, then the RTOS places an **error message** at the address passed as third parameter and puts the task "ready"

An important remark applies to Interrupt Service Routines. **ISRs** are functions that are called by an event (the interrupt) and hence are no scheduled by the same mechanism. Consequently they **cannot wait for a semaphore**, because they are not tasks that can be put in "waiting" state.
Therefore, a special call to the RTOS exists just to test the value of the semaphore and the OS always returns the processor to the ISR.

RTOS: task communications

how a task asks releases a semaphore

- task calls **OSSemPOST**(SemName)
- RTOS
  - browses the list of tasks waiting for this semaphore
    - *if* (list is empty)
      - increment the semaphore
      - returns to running task
    - *else*
      - make "ready"
        - the 1st task in the list (FIFO policy)
        - the highest-priority task in the list (priority policy) [µC/OSII]
      - calls the scheduler
        - if ∃ higher priority task => task switch
        - else return to running task

The release of a semaphore is a similar call to the RTOS.

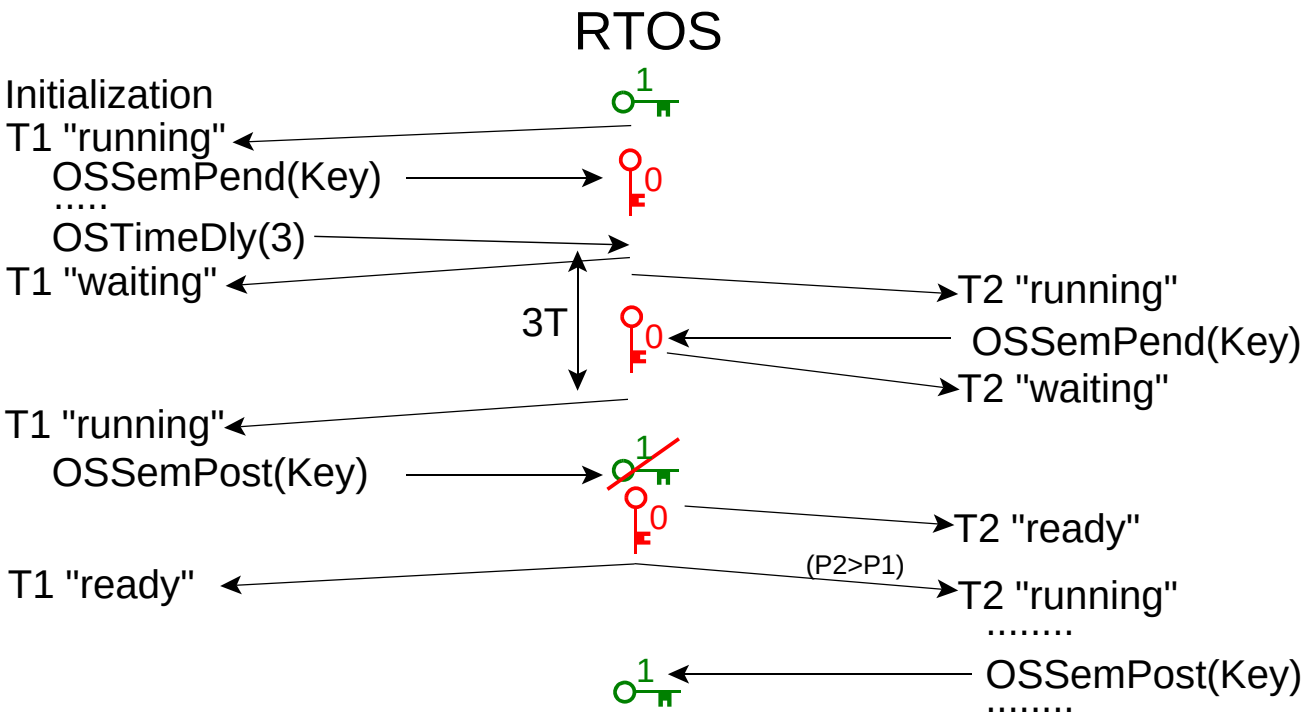
We could think that the RTOS just increments the semaphore to make it available and gives the processor back to the running task.

This is only true if no other task is currently pending for the semaphore.The RTOS can be verified it in the data structure associated to the semaphore which contains the lists of all such tasks. If the list is not empty and contains several tasks, there are to major ways to elect the winner
- the highest priority one; this is the simplest choice and also the one that corresponds to the philosophy of the RTOS
- the order of the calls; in that case, the list is organised as a FIFO stack (First In First Out)

The scheduler will be called to run the highest priority task.

RTOS: task communications

protection of a shared resource by a "key" semaphore



Suppose that two tasks T1 and T1 have to share a unique resource, e.g. a printer. We don't want parts of text coming from two sources to be interleaved on the paper, hence we need **mutual exclusion** (MUTEX).

In the initialization part of the program, after checking that the printer is ready, we place the instruction

Key=OSSemCreate(1)

"Key" is here a generic name to recall that this creates a "key semaphore" initialized to 1. In clean readable code, you should call it "PrinterReady" or any other meaningful name.

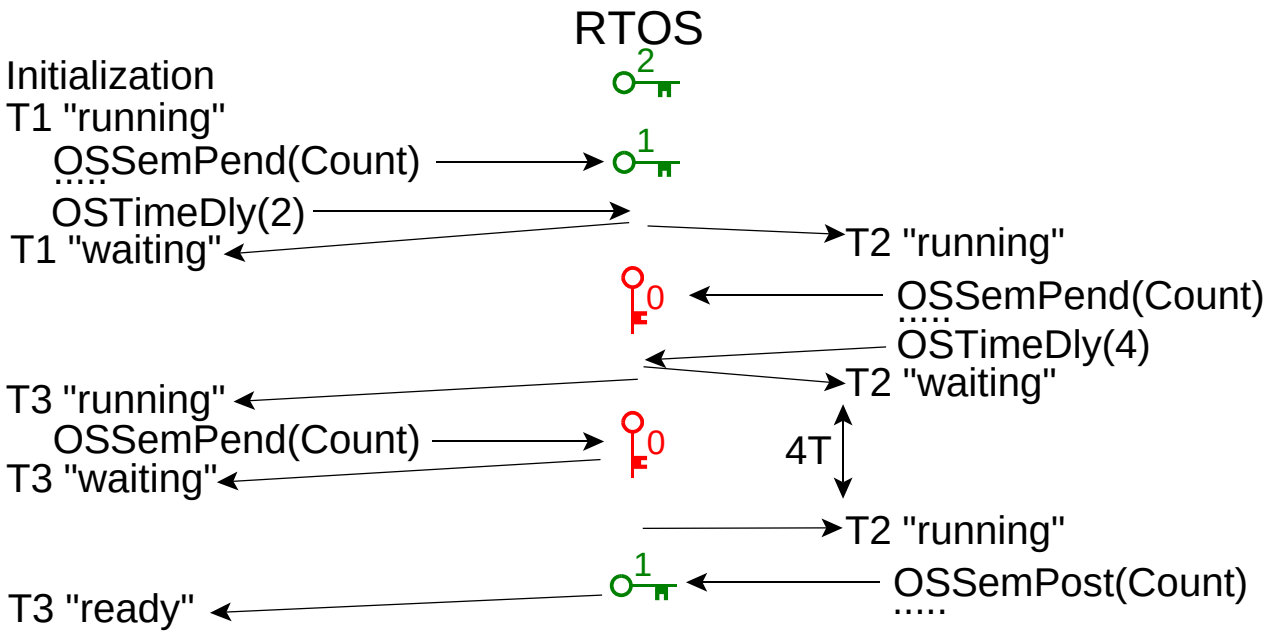
- during its execution, T1 asks for the resource by a call. Since the key is available, the RTOS "gives the Key to T1", simply by decrementing the Key to 0 and returning "no error" to T1. The resource now belongs exclusively to T1
- let us suppose that T1 returns to "waiting" by suspending itself of because it has been preempted

- when T2 becomes "running", it makes the same call OSSemPEND(Key). Since the key has been given to T1, T2 is preempted, goes to "waiting", and is placed in the waiting list of the semaphore Key

- later, when T1 is "running" and releases the semaphore by a call OSSemPost(Key),
  - the semaphore Key is not incremented since T2 is pending in its waiting list
  - T2 becomes "ready"
  - the scheduler is called which
    - sees that T2 has a higher priority than T1
    - preempts T1
    - changes the context and restarts T2
- T2 is resumed at the instruction just after the OSSemPEND(Key) which had suspended it
- T2 has got exclusive access to the resource protected by Key until it releases it
- T2 releases the semaphore by a call OSSemPost(Key), since no more tasks are waiting for the semaphore, it is incremented and becomes available.

RTOS: task communications

multiple resources: counting semaphore



The same mechanism can by extended when the resource is not unique. Here we take the example of a **double buffer**.
The only difference is that the semaphore is created as a **counter** initialized to the number of available resources (here 2).

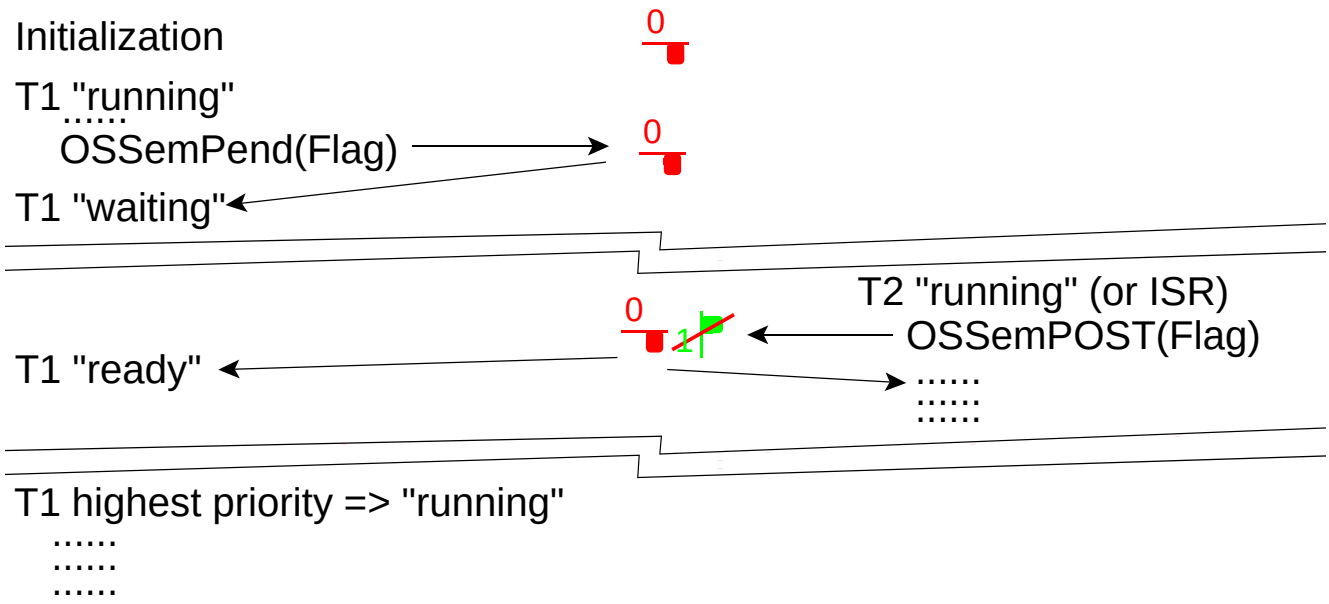
Each OSSemPend(Counter) decrements it, OSSemPost(Counter) increments it.

A task pending for a Counter=0 goes to "waiting" and will go to the "ready" when Counter >0.

The scheduler decides which task will be "running" each time a task become "ready".

RTOS: task communications

synchronization by a "flag" semaphore: *rendez-vous*



Suppose that T1 has to be synchronized to T2:
- we need a sequential execution, e.g. a state machine; T1 must occur after T2
- part of T1 cannot be executed until T2 has produced some results

This is called "**unilateral synchronisation**" or "**unilateral rendez-vous**"

In the initialization part of the program, before starting the tasks, we place the instruction

Flag=OSSemCreate(0)

"Flag" is here a generic name to recall that this creates a "flag semaphore" initialized to 0. In clean readable code, you should call it "T2_sync" or any other meaningful name.

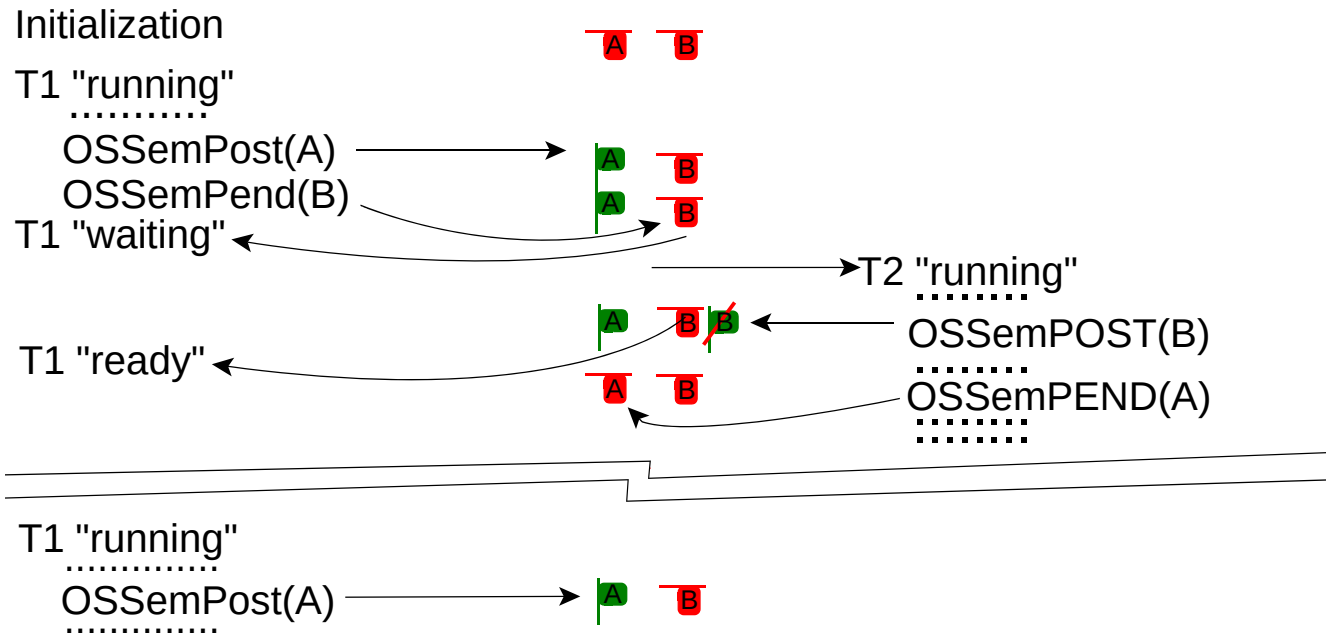
- during its execution, T1 suspends itself until T2 occurs by a call OSSemPend(Flag); since the Flag is false, the RTOS puts T1 in the "waiting" state, and places T1 in the waiting list of the semaphore Flag

- during its execution, T2 releases the semaphore by a call OSSemPost(Flag), the RTOS
  - sees that T1 is waiting for Flag and marks T1 as "ready"; semaphore is not incremented, since a task is waiting for it
  - T2 is resumed at the instruction just after the OSSemPOST(Flag)
- when T1 has got the highest priority, it resumes execution just after the OSSemPend(Flag)

REM
- T2 can be an Interrupt service Routine and T1 a task waiting for this event (see further §: management of interrupts, IISR and DISR)
- the reverse is not true: T1 cannot be an ISR, since an ISR cannot wait for a software event. An ISR it is only triggered by a hardware event, the interrupt request. An ISR can only test a semaphore by a call OSSemAccept(Flag).

RTOS: task communications

bilateral *rendez-vous*



Both task can synchronise to the other one, which is known as " **bilateral synchronization**" or "**bilateral rendez-vous**".
Its simply the extension of the previous mechanism, with the resort to a second flag.
Both flags A and B are initialized to 0

REM
- T1 and T2 have to be ordinary task; they cannot be ISRs
- beware of the order of the PEND and POST instructions, what happens if you exchange them ?

RTOS: task communications

mailboxes: exchanging data between tasks

- a mailbox is
  - a protocol for multitasking data exchange
  - an object
    - belonging to the kernel
    - created by a call OSMboxCreate() in the initialization
- a mailbox contains
  - a pointer-sized variable initialized to point to a message
    - **NULL pointer means empty mailbox**
  - a list of tasks waiting for a message in the mailbox
- very similar to the semaphore
  - PEND, POST, Timeout
  - can replace the semaphore but less efficient (code, latency)
- ISR can POST to a mailbox, but not PEND

The mechanism of the semaphore can be transposed to mailboxes for to the exchange of messages between tasks. Instead of a counter, the mailbox is based on a pointer to a message (or to null if no message is available).
You will have the opportunity to use semaphores and mailboxes during the labs.

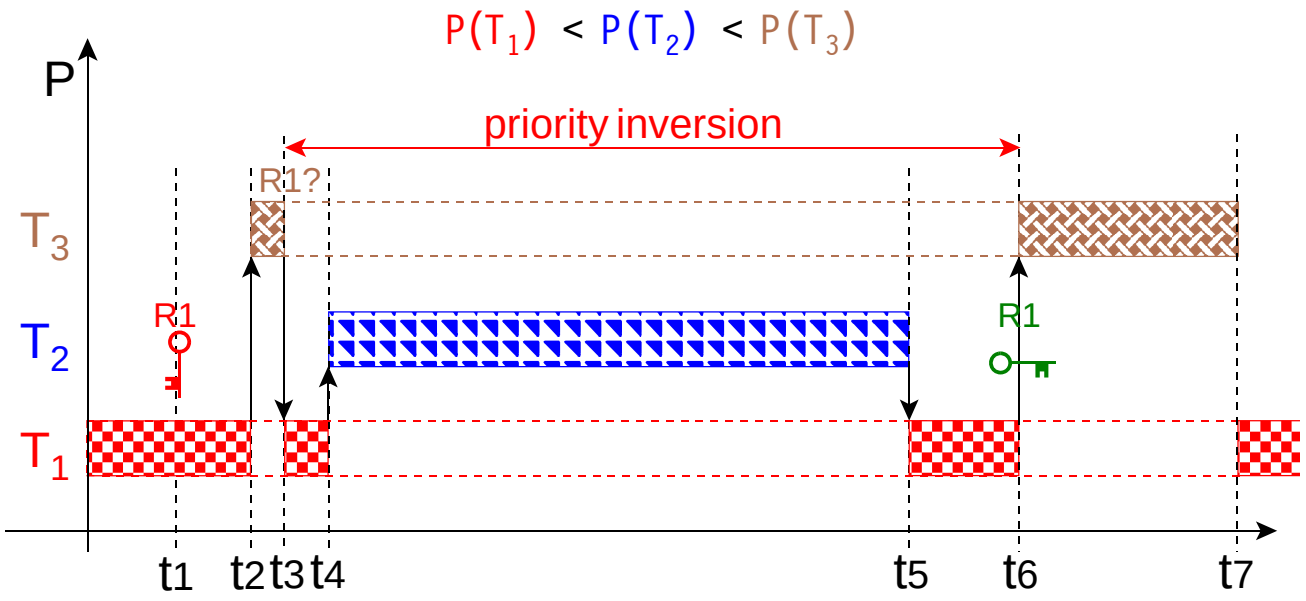
Priority-driven scheduling

CONTENTS

- foreground/background systems
- **RTOS**
  - principles
  - **inter task communication**
    - shared global variables
    - semaphores
    - mailboxes
    - **dependancy hazards**
  - time *ticks*
  - managing interrupts
  - main scheduling algorithms
- conclusions

RTOS: dependancy hazards

Priority Inversion: a high priority tasks doesn't run



A **high-priority task can be delayed by the execution of tasks of lower priority** ; this is known as **priority inversion**. This phenomenon occurs chiefly with non-preemptive schedulers (see managing interrupts) but it can also happen with a preemptive scheduler when tasks are in competition for a shared resource.

On this figure, we see three tasks whose index indicates the priority, T3 is thus the highest priority task.

Let us suppose that T1 and T3 share the same resource (for example an I/O port), protected by a semaphore R_1 initalized to 1, which means that the resource is free.

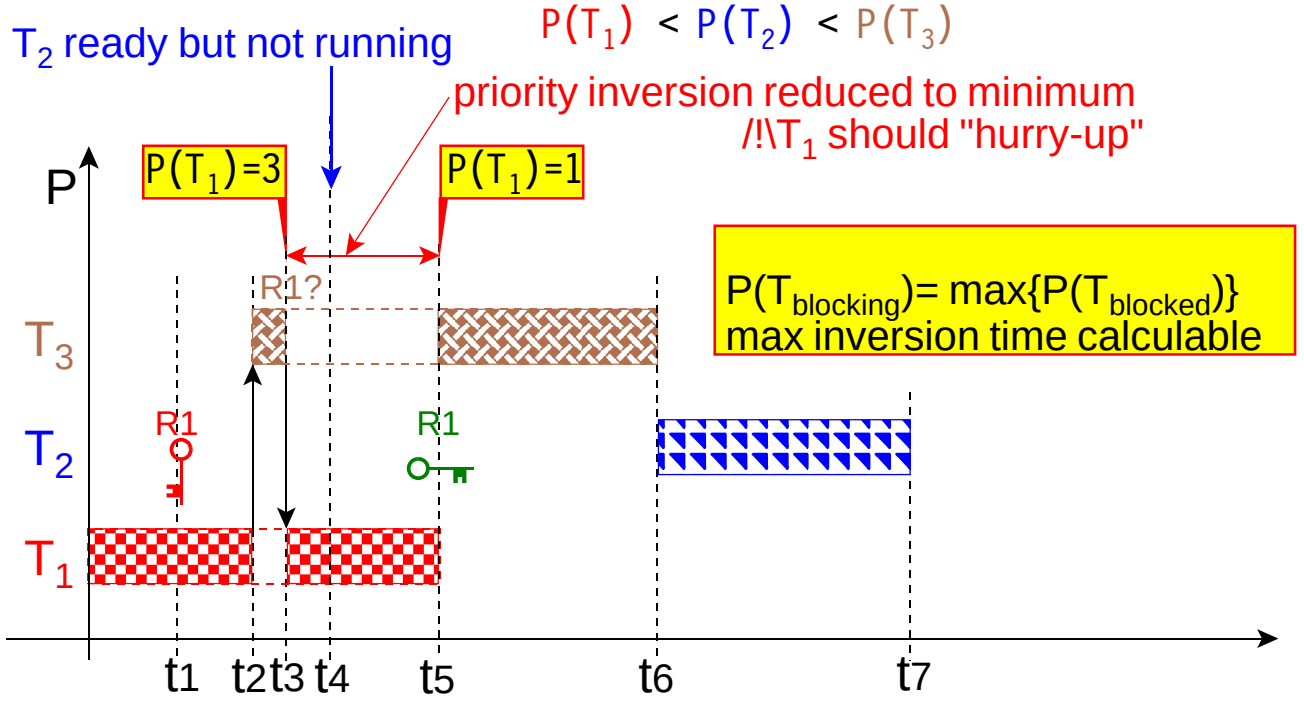
- at t=0, T_1 is ready, has got no concurrent task and starts immediately.
- at $t=t_1$, T_1 executes a *call OSSemPend(R_1)*; since the resource is free, **T_1 gets the semaphore (R_1 is decremented to 0 by the RTOS) and T_1 gets exclusive access to the shared resource**
- at $t=t_2$, T_3 becomes ready, T_1 is preempted and T_3 starts
- at $t=t_3$, T_3 requires the shared resource by a *call OSSemPend(R_1)* and is suspended to the "waiting" state, because the semaphore is not free. The scheduler then restarts T_1, which is the only "ready" task
- at $t=t_4$, T_2 becomes ready, T_1 is preempted and T_2 starts, which delays the execution of T_1
- at $t=t_6$, T_1 finally releases the semaphore by a call *OSSemPost(R1)*; this gives control back to the scheduler, which preempts T_1 and restarts T_3.
- at $t=t_7$, T_3 is finished, goes back to "waiting" and T_1 can be restarted.

Priority inversion occured during the interval (t_6 - t_1): T3 should have been running, being the highest priority task, while lower priority tasks T1 and T2 were excuted instead.

A solution, is to resort to **priority inheritance**, which we shall see now.

RTOS: dependancy hazards

Priority Inheritance minimizes the inversion duration



To reduce the duration of priority inversion, we can resort to the **priority inheritance.** It consists in temporarily allotting to a task causing the inversion (here T_1) the highest priority of the tasks that it blocks (i.e. the priority of T_3).

The goal is obviously that the blocking task releases as soon as possible the resource to which it has got exclusive access.

On the figure, we see that as soon as T1 blocks T3, T1 gets the priority level 3, which prevents the readiness of T2 to cause the preemption of T_1 at t=t4; task T_3 will get the resource at t=t5, when T1 releases the semaphore.

A task like T1 should always "hurry-up" when it owns a semaphore, **calls like OSSuspend() or OSTimeDly() are forbidden before the semaphore is released**, otherwise T1 freezes all tasks below its current inherited priority.

µC/OSII

MUTEX: a special semaphore

- µC/OSII Semaphores: no priority inheritance
- µC/OSII MUTEX
  - MUTEX ="key" semaphore + **priority "raising"**
  - same kind of calls
    - OSMutexPend(MutexName,timeout,&err)
    - OSMutexPost(MutexName)
  - creation is different
    - PrinterReadyMutex=**OSMutexCreate(Prio,&err)**
- protocol
  - a low-priority task T_1 gains the MUTEX by OSMutexPend()
  - a higher-priority task T_k calls OSMutexPend()
  - the priority of T_1 is raised to **Prio** until T_1 releases the MUTEX
  - if Prio>P(T_k) T_1 keeps running

The RTOS used in the labs, uCOSII uses a simplified version of those mechanisms.

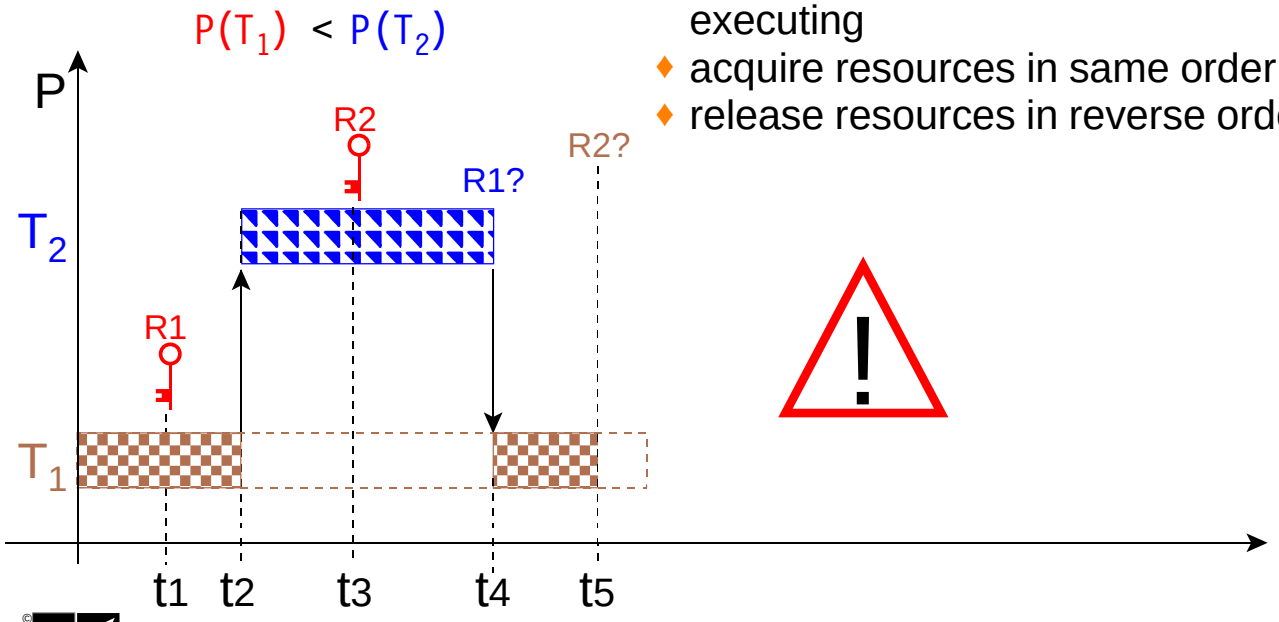
- semaphore have no priority inheritance

- another type of semaphore, the **MUTEX** implements a slightly different method to reduce priority inversion: **static priority raising**. As a matter of fact, this simple and small RTOS kernel is not able to detect the inversion and, a fortiori, to attribute automatically the priority of the blocked task to the blocking task. It is the responsability of the programmer to indicate, during the creation of the MUTEX, to which priority level any task getting the MUTEX will be raised.

RTOS: dependancy hazards

Deadlock: no tasks is running any longer

- solutions: all tasks
  - acquire all resources before executing
  - acquire resources in same order
  - release resources in reverse order



Another phenomenon can occur in preemptive systems where exclusive access is given to at least two shared resources: the **deadlock**.

- at $t=t_1$, T_1 gets the exclusive access to R_1 by a *call OSSemPend(R1)*
- at $t=t_2$, T_2 becomes "ready" and is started
- at $t=t_3$, T_2 gets the exclusive access to R2 by a *call OSSemPend(R2)*
- at $t=t_4$, T_2 wants to use R_1 by a *call OSSemPend(R1)* resource, and returns to "waiting" because R1 is reserved; T_1 is restarted
- at $t=t_5$, T_1 wants to use R2 by a *call OSSemPend(R2)* resource, and returns to "waiting" because R2 is reserved

The **deadlock** occurred: neither T1, nor T2 can be restarted anymore. They are both "waiting" for an event (the release of a semaphore) which will never occur.

The priority inheritance does not solve this problem.

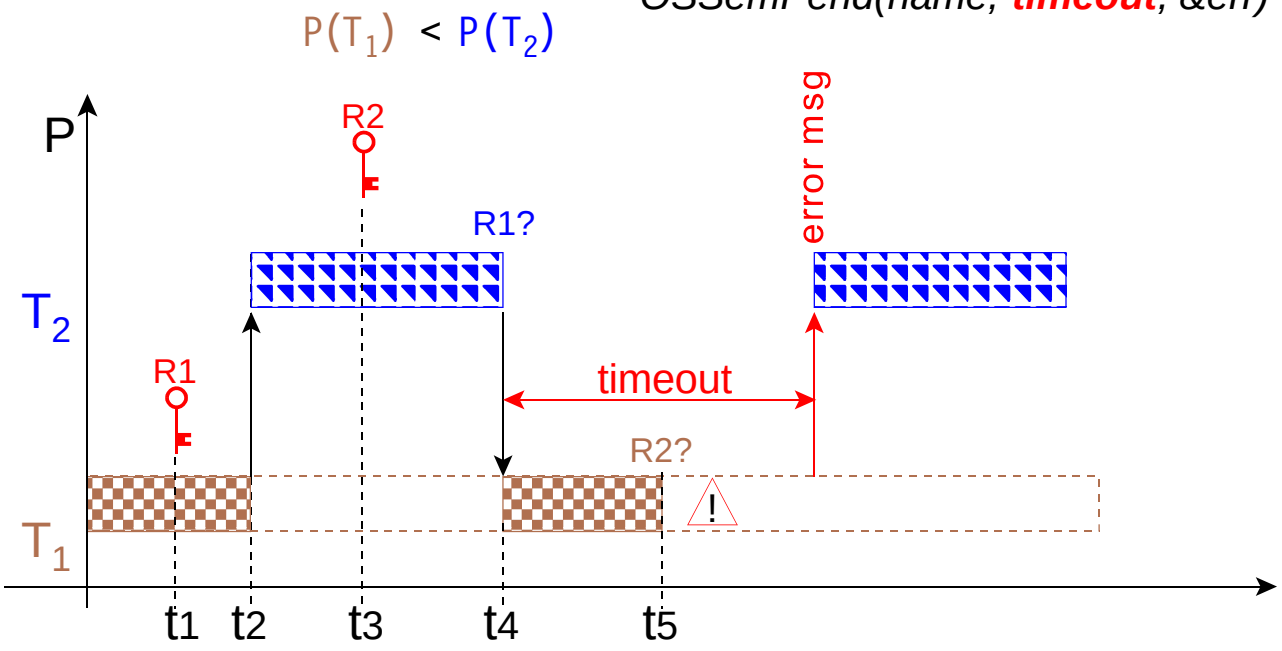
Two simple solutions can be found by putting **constraints on the way tasks acquire resources**
- either all tasks have to **reserve all the resources** they need **before proceeding**
- or all tasks **acquire resources in the same order and release them in the reverse order** .

Exercise: draw those scenarios

RTOS: dependancy hazards

Deadlock: *timeout*

*OSSemPend(name, **timeout**, &err)*



Deadlock can be broken thanks the two last parameters in the call *OSSemPend()*
- a **timeout** parameter
- an **error handler** passed by reference so that the RTOS can modify it

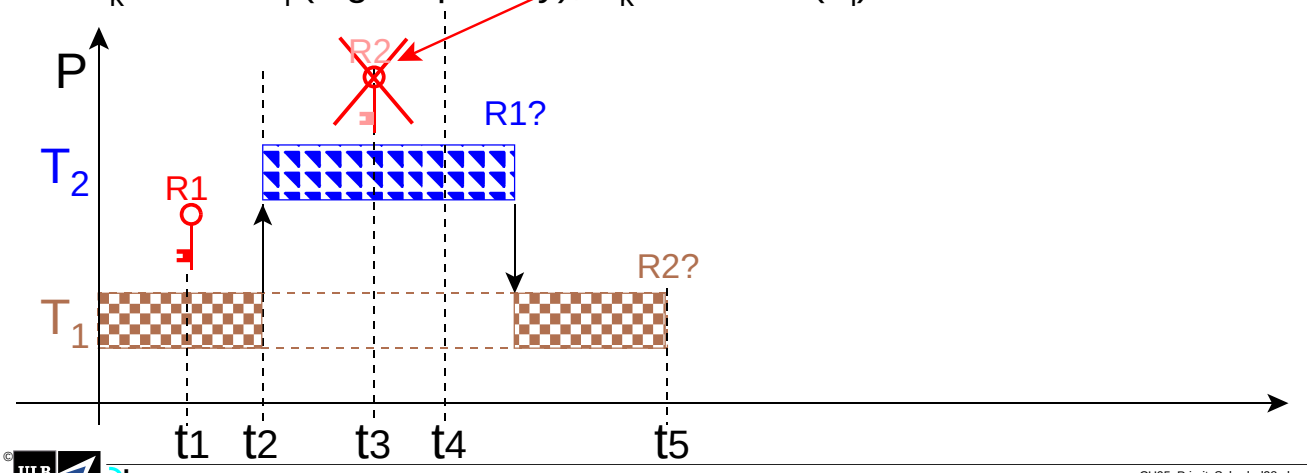
When it receives the call, the RTOS initializes a counter in the semaphore, which is decremented at each clock tick. When the counter reaches 0, the scheduler puts an error code (predefined constant TIME_OUT) in the variable "err" and changes the state of the task T_2 from "waiting for the semaphore" to "ready". When the task T_2 is restarted, it looks to the value of "err", is aware that the timeout is expired and can take actions based on this information.

RTOS: dependancy hazards

Priority Ceiling

- for each resource R_j
  - P_ceiling = max{T_i asking for resource}
- T_i can have access to the resource only if

  P_i > P_ceiling of already taken R_j
- if T_k blocks T_i (higher priority), T_k inherits $P(T_i)$



Deadlock can be solved by the **Priority Ceiling Protocol**.

The principles are:
- each **resource** is allotted a **ceiling_priority** equal to the maximum priority of the tasks which can access to it
- **a task can access to this resource if and only if the priority of this task is higher than the ceiling_priority of all the resources currently taken**
- if a task T_k blocks a higher priority task T_i, T_k inherits the priority of T_i

If we apply these principles to the previous example:
- the priority of T_1 is 1
- the priority of T_2 is 2
- the two shared resources R_1 and R_2 have got a ceiling_priority CP=priority of T_2=2
- at $t=t_1$, T_1 gets exclusive access to R_1 with a semaphore. From that moment, only resources whose CP=3 are still available
- at $t=t_2$, T_2 becomes "ready" and is started
- at $t=t_3$, T_2 tries to get the resource R_2, but this request is refused because its priority (=2) is less than the ceiling priority (=3)
- no deadlock occurs

RTOS: task communications

global variable, semaphore, mailbox: which is the best?

- global variable
  - advantages
    - lots of globals inherent to the system
    - good way to send data to an ISR
  - drawbacks
    - critical section to guarantee exclusivity => latency
      - harmless if latency < scheduler latency
    - globals changed in an ISR have to be *polled* by other tasks
- semaphore & mailboxes
  - advantages
    - easier and safer for the programmer (RTOS manages everything)
    - ISR can test a semaphore or a mailbox
    - ISR can send signals or messages to tasks which don't have to polling
  - drawbacks
    - heavier (calls to the OS)
    - an over-kill to manipulate single variables

All mechanisms can be used for communication.

Do not throw away the usage of global variables:

- they are **inherent** to your system (interrupt flags, state variables of the process.....). For example, if a periodic task computes a temperature by making two successive read operations in an analog-to-digital converter, compensate the offset, and convert to Celsius scale, why not "publish" the result in a global variable so that every task gets access to it immediately?
- even if you have to protect the exclusive access to the variable by disabling/enabling interrupts (i.e. using critical sections), it is not necessarily harmful for the system. Critical sections are part of the latency time, but if you use an RTOS, the scheduler itself is generally a critical section, hence **if you create critical sections that are not longer than the scheduler, you do not increase the latency**
- global variables are the most efficient mechanism to manipulate single boolean or integer variables
- are convenient to send data to an ISR
- receiving data from an ISR implies polling this data in every "consumer" task

On the other hand, **semaphores and mailboxes**

- are **easier and safer** because the programmer subcontracts responsibilities to the RTOS
- can be tested by ISRs to get signals or messages
- are the **best way for an ISR to signal events or send data** to tasks because you do not have to code loops to poll for the changes in a global and manage timeouts, you just have to PEND for the event and let the RTOS do the rest.
- are rather **slow**, so do not overuse them
- are too powerful to exchange single variables

Priority-driven scheduling

CONTENTS

- foreground/background systems
- **RTOS**
  - principles
  - inter task communication
  - **time *ticks***
  - managing interrupts
  - main scheduling algorithms
- conclusions

RTOS: time management

an RTOS needs a timebase

- RTOS consumes a hardware timer
  - generates high-priority periodic interrupts called **ticks**
- software timers
  - int variable
  - initialized on an event
  - decremented by the Tick_ISR belonging to the RTOS
  - lots of software timers
    - OSTaskSuspend(delay) from tasks
    - timeouts while waiting for events (semaphore, mailbox)
    - time-out to preempt a task (see further "round-robin")
- "time"variable
  - counter incremented at each tick
  - generally 32 bits (i.e. 50 days at 1kHz)

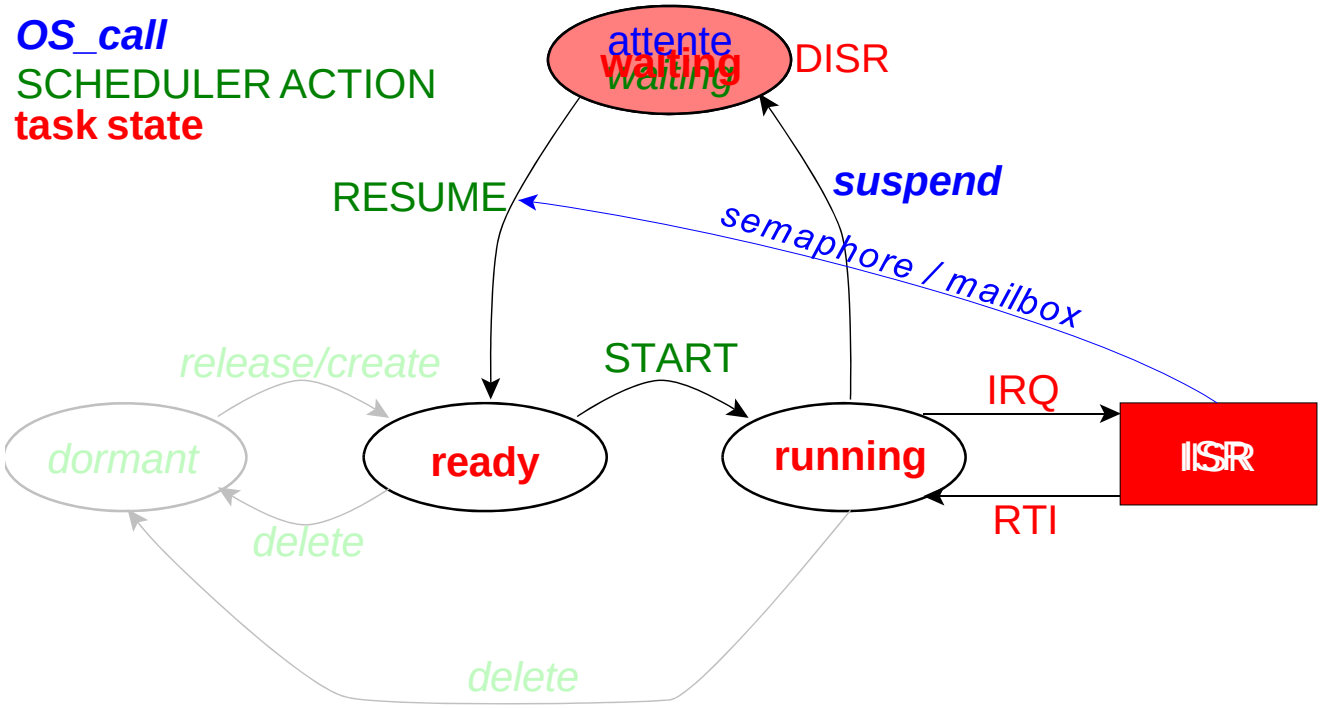
Priority-driven scheduling

CONTENTS

- foreground/background systems
- **RTOS**
  - principles
  - inter task communication
  - time *ticks*
  - **managing interrupt**
  - main scheduling algorithms
- conclusions

RTOS: managing interrupts

non-premptive RTOS: IISR, DISR



The running task can be interrupted by the arrival of a non-masked interrupt request.

A non-preemptive RTOS does not manage interrupts. The Interrupt Service Routine (ISR) is executed and the Return from Interrupt gives the control back to the interrupted running task.
We can consider in this case the **interrupt request as a super-priority event** and the **ISR as a super-task which escapes scheduling** (but the **scheduler is in most of the cases in critical section** and hence cannot be interrupted). The latency introduced by the RTOS should be documented. The latency of the ISR is minimal. Priority hierarchy among the IRQs is managed by the hardware (Priority Interrupt Controller) and/or by the software (a low-priority ISR has to unmask higher priority ones).
The ISR must thus be sufficiently short not to disturb scheduling significantly. We can find several alternatives:

- the ISR entirely manages all the tasks relating to the event; it is thus likely to be long; this problem is still worsened if, during its execution, the ISR unmasks other higher-priority interruptions, thereby authorizing *nested interrupts*

- the ISR, very short, executes only urgent instructions, hence the name IISR (Immediate Interrupt Service Routine) and leaves the remainder to an ordinary task called DISR (Deferred Interrupt Service Routine), which is scheduled like any other task.

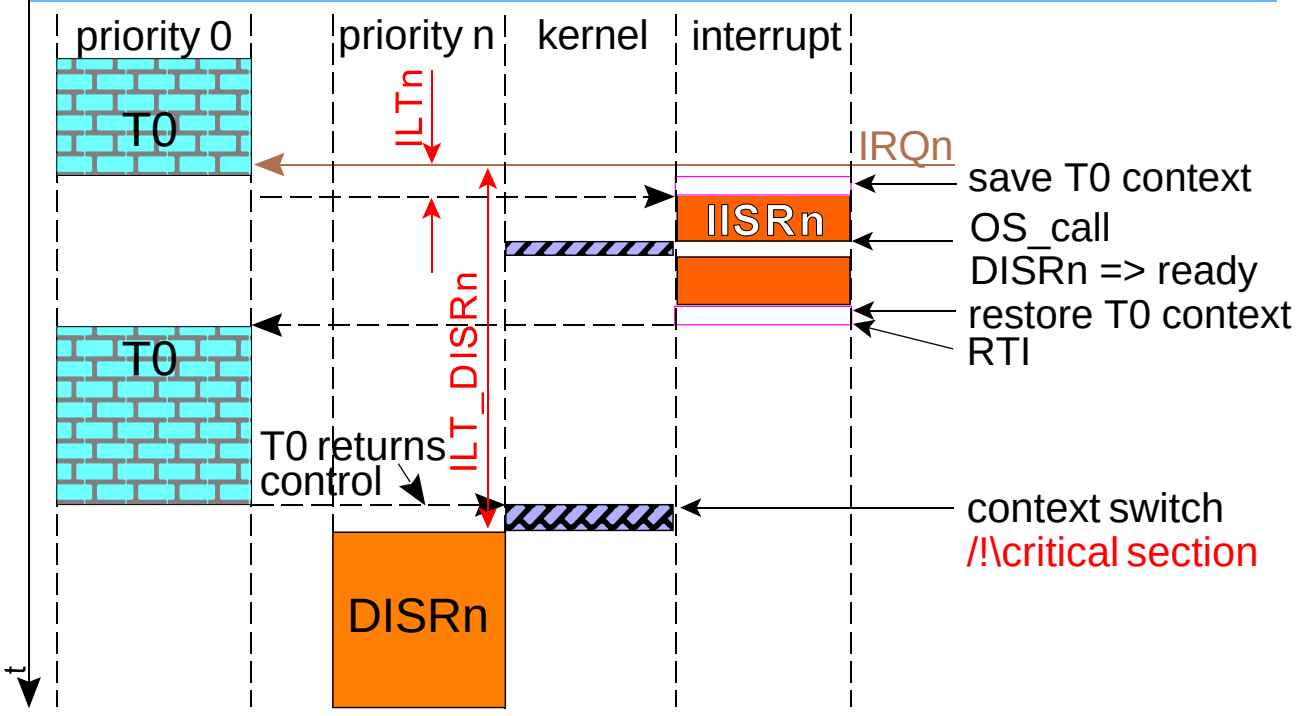
Let us go back to the example of a float which monitors the filling of a tank; when the level reaches 90%, a switch activates the IRQ, the IISR resets the output bit driving the pump, and the DISR is charged to modify the appearance of the tank on the synoptic panel of the installation.

The DISR is a task that, when running for the first time suspends immediately itself to the waiting state, because it has to wait for the occurrence of the IISR. This is done via a semaphore (if the DISR only waits for a trigger) or via a mail box (if the DISR waits for a message containing data). The IISR will either signal the semaphore or fill the mailbox and call the RTOS, which shall put the DISR in the "ready "state, so that it can be scheduled normally. It is in fact a *unilateral rendez-vous*.

Even if the DISR has got a high priority, a non-preemptive RTOS cannot start it until the interrupted running task has suspended or deleted itself. The timing diagram of the interrupt is presented at the next slide.

RTOS: managing interrupts

non-preemptive RTOS: interrupt timing



Let us illustrate a non-preemptive kernel managing priority levels from 0 to n.
The kernel itself runs at the highest priority. The interrupt level is a kind of *super-priority* created by the CPU hardware.

A task T0 is executing at the lowest priority. The interrupt request IRQn occurs
- the context of T0 (i.e. all registers it uses) is saved on its own stack
- after a latency time ILTn, the Immediate Interrupt Service Routine IISRn starts
  - immediate actions are executed
  - IISRn asks the OS to pass the Deferred Interrupt Service Routine DISRn from "waiting" to "ready" (e.g. via a semaphore or a mailbox). DISRn has got the highest priority level n.
  - IISRn executes its RTI
- the context of the interrupted task T0 is restored by the processor and T0 resumes its execution

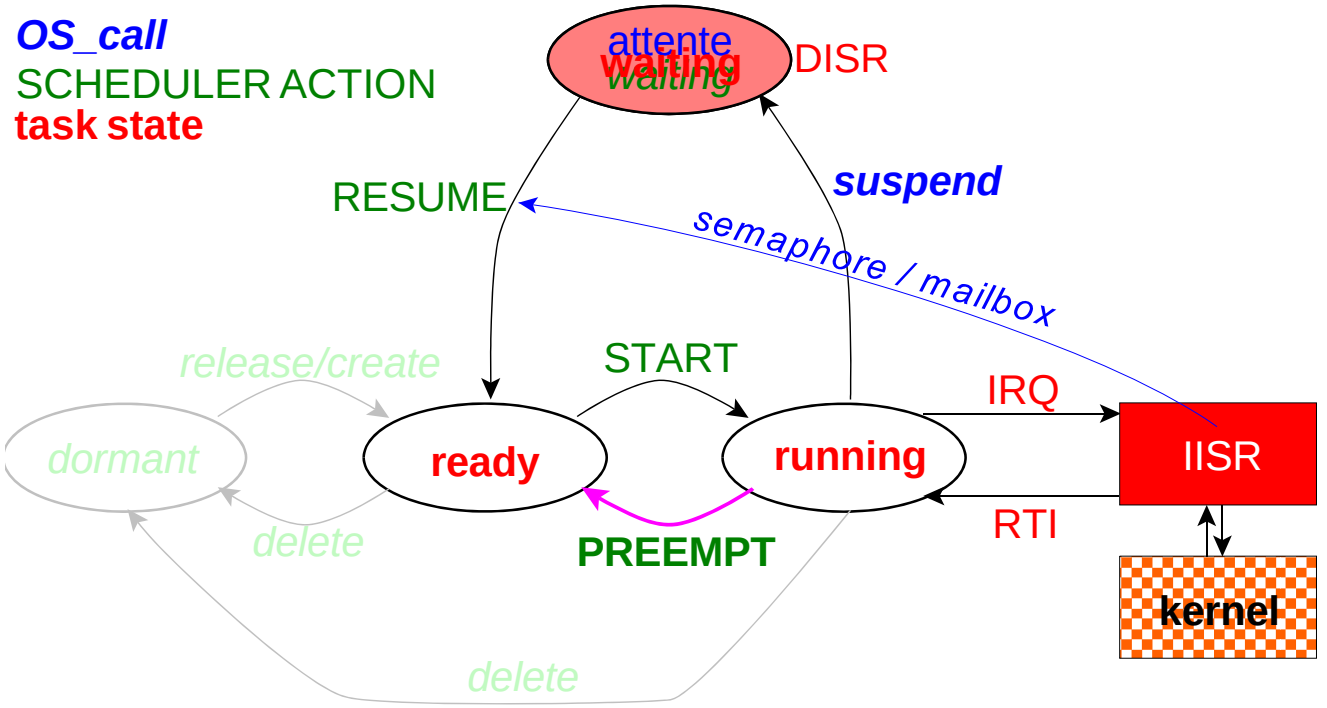
T0 is running whereas DISRn should run *because of its higher priority: this is a case of* ***priority inversion*** *due to the fact that* ***the scheduler cannot preempt T0****.*

- T0 asks the OS to return to "waiting"; this gives the control back to the scheduler, which will
  - examine the list of ready tasks, and see that DISRn has got the highest priority
  - make a context switch, i.e. save all registers and data belonging to T0 to its own stack and restore all registers and data of DISRn from DISRn stack
  - finally start DISRn which becomes "running";

In conclusion, with a non-preemptive scheduler
- the latency time of the IISR is not modified by the RTOS compared to a foreground/background system (there are some critical sections in RTOS, but it is the same for the background)
- the latency time of the DISR is in the worst case equal to the sum of
  - the execution time of the longest task (ISRn can occur just after the beginning of this task)
  - the execution time of the IISR
  - the delay to switch the context

RTOS: managing interrupts

premptive RTOS: IISR, DISR, preemption



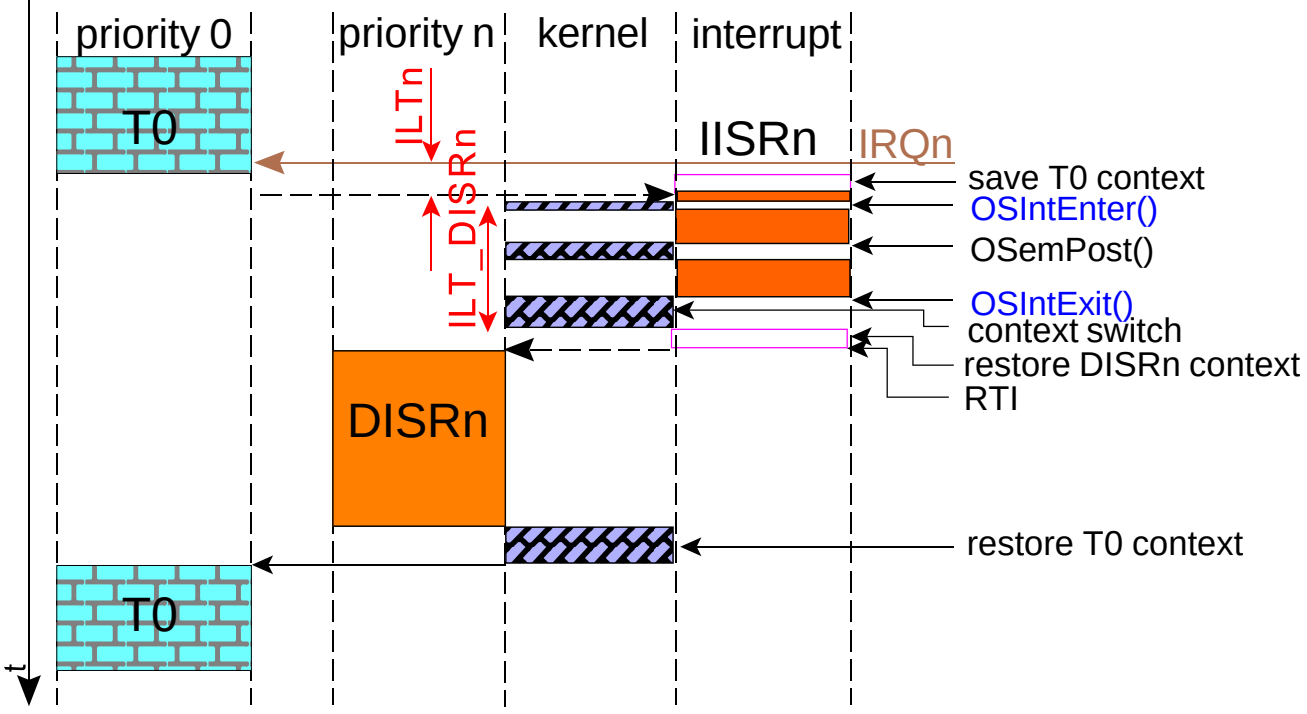
The fact that the kernel can **preempt** a task allows a **better respect of the priorities**. In particular, the problem of priority inversion and long latency time of a DISR is reduced.

The **IISR has to make special calls to the kernel**. Moreover, the fact that DISR becomes ready invokes the scheduler. If DISR has got a higher priority than the interrupted task, this one is preempted and DISR is started.

Details of timing and calls are given in the next slide.

RTOS: managing interrupts

preemptive RTOS: interrupt timing

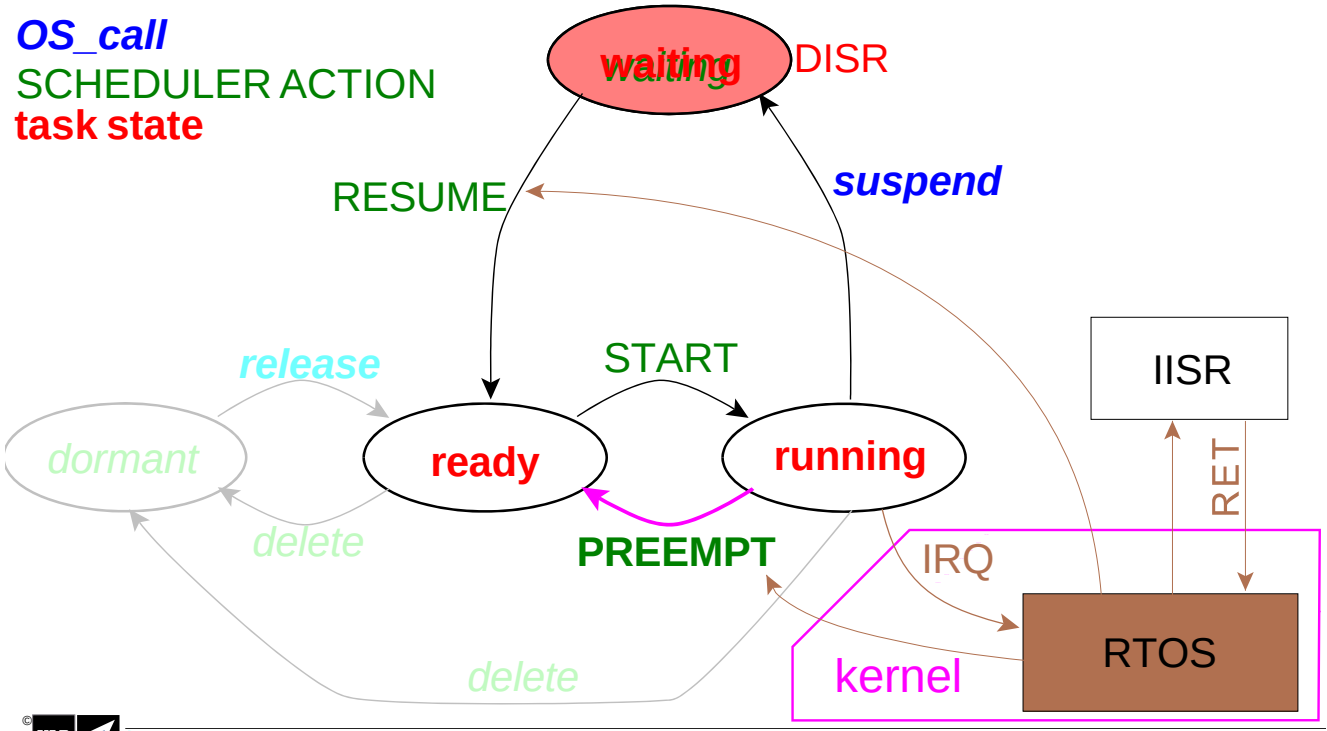


The IISR must include 3 special calls to the RTOS (we shall take µC/OSII as an example). The sequence in the IISR is:

1. save T0 context on its stack, as usual
2. **call OSIntEnter()** to warn the scheduler that an interrupt is currently serviced
3. execute IISR instructions
4. **signal a semaphore OSSemPost()** or **fill a mailbox OSMboxPost()** for which the DISR is waiting. This call gives the control to the scheduler, which will change the DISR state from "waiting" to "ready"
5. **call OSIntExit()** to signal the scheduler that the IISR is finished. The scheduler will scan the list of "ready" task; if we suppose that the DISR has got the highest current priority, **the scheduler will preempt the interrupted task and change the context** which means that the stack pointer now points to the stack of the DISR; the scheduler returns from this call to the IISR
6. the IISR executes a return from interrupt (RTI) and, because of the context switch which happened at step5, the **DISR is started** just as if it had been interrupted by the IISR.

RTOS: managing interrupts

interrupts: high-end processors and RTOS



In much of modern processors, an interruption is a major event that switches in hardware the processor in a **special mode/context called "kernel" or "supervisor"**
- high priority/privilege
- protected memory space
- private set of registers (General Purpose Registers, Stack Pointer)

This mode is normally reserved to the OS.

Normal tasks run in the "user" context.

In that case, the IISR must :
- either belong to the RTOS
- or have the same privileges of execution than the RTOS (effective, but the protection of the RTOS is weak if the IISR is badly written)
- or be called by the RTOS, which switches the processor back to "user" mode (safer, but slower, because of the context switch)

The documentation of the RTOS has to specify how to write the code of an IISR and how to "install" it in the system.

One of the most important difference between a "normal" and a "real-time" OS in the capacity to manage interrupts in an efficient way
- low latency
- determinism
- respect of priorities

RTOS: context switch

similarities between IRQ and task-switching

- IRQ context switch
  - CPU hardware PUSHes PC (+some registers A_i) on the current stack
  - ISR preamble PUSHes other registers B_i used by the ISR
  - ....
  - ISR postamble POPs registers B_i (in the reverse order !)
  - RTI => CPU hardware POPs registers A_i then PC
- Task context switch
  - PUSH registers A_i+B_i on the stack of the current task and mark it as "ready"
  - change Stack Pointer to point at the top_of_stack of next task
  - POP registers Bi from stack of next task
  - mark next task as "running"
  - RTI as if an interrupt has occured to POP PC and registers A_i

simplicity, determinism, systematicity => most RTOS save all GPR for IRQ and context switch => "slow"

Changing the context is a mechanism which is implemented in every processor as part of the management of the interrupts. The current state of the processor (i.e. the **context**) must be saved so that the execution can resume exactly in the same condition when the interrupt has been fully serviced.

The fundamental part of this context saving is done in **hardware** and uses the **stack** as temporary storage space for the context:
- the PC is always PUSHed on the stack
- another set of registers A_i can also be PUSHed automatically in hardware. A_i depends
  - on the processor itself
  - on the kind of interrupt (e.g. "fast" IRQ saving few registers and "slow" IRQ saving a lot of registers)

It is the responsibility of the programmer (and/or of the compiler) to save, in the preamble of the ISR, all registers B_i that have not been saved automatically and are susceptible to be modified in the ISR.
At the end of the ISR
- the registers B_i are POPped from the stack (if you write in assembler, you must do it exactly in the reverse order of the PUSHes)
- the last instruction of the ISR is RTI (ReTurn from Interrupt); the execution of this instruction by the CPU will then automatically restore the registers A_i, and finally the PC

When the **RTOS** wants to make a **task switch**, a similar mechanism can be used except that
- all the PUSHes related to saving the context of the current "running" task have to be done in software on the stack of this task, which is then preempted, i.e. marked as "ready"
- each task has got its own stack; the context switching consists in changing the content of the Stack Pointer, so that it points to the top of the stack of the new task (this value of the stack pointer is contained in a structure belonging to the OS called Task Control Block (TCB))
- then B_i registers are POPped from the stack of the future task, which is marked as "running"
- finally a fake RTI is executed to restore A_i and the PC of this task, which starts it

All registers (or at least all registers considered by the compiler as General Purpose Registers) are generally saved when the context is switched, to perform systematic operations. For modern processor it means saving and restoring 32 registers, which is rather slow.

Priority-driven scheduling

CONTENTS

- foreground/background systems
- **RTOS**
  - principles
  - inter task communication
  - managing interrupts
  - **main scheduling algorithms**
    - **RM, EDF, DM, LL**
    - round robin
    - multiprocessor
- conclusions

RTOS

algorithms for periodic tasks : hypothesis

- n tasks: worst execution e_i and min period p_i
- independant tasks
  - do not have to wait for synchronisation
  - do not have to wait for data
  - no shared resources
- preemptive kernel
- $U_{kernel} << \Sigma e_i/p_i$: the kernel is a negligible CPU load (<5% in most cases)
- single processor

RTOS

RM (*Rate Monotonic*): principle

- implicit deadine : $D_i = p_i$
- priorities : **static, sorted on the frequency $1/p_i$**
  - $p_i \searrow$ => $P_i \nearrow$
  - intuitive: shortest p=>shortest D=>most urgent
- properties
  - optimal in mono-processor
- schedulability
  - SC [LIU73] $U = \sum e_i/p_i \leq n(\sqrt[n]{2} - 1)$
  - NSC [JP86] $W_i = e_i + \sum_{j=1}^{i-1} \left\lceil \frac{W_i}{p_j} \right\rceil \times e_j \quad et \quad W_i < p_i \quad \forall i$

n	Umax
1	1.00
2	0.83
3	0.78
4	0.76
5	0.74
∞	0.69

One of the most famous algorithms is **Rate Monotonic**.
- *Rate* represents in fact the frequency of execution of the task, i.e. the inverse of their period.
- *Monotonic* means that the priority of the tasks results from a monotonous classification on the frequency.

In other words, **the task which has the smaller period will have the highest priority,** and so on. Priorities are thus **static**.

In the classical version, this algorithm is valid for implicit deadlines (relative deadline=period). Hence, it becomes **quite intuitive: shortest period=shortest deadline=most urgent task=highest priority.**

It has been demonstrated that **RM is optimal** on a single processor with the following conditions of schedulability:
A **sufficient condition** is based on a limit of the utilisation ratio of the processor, as a function of the number of tasks to be executed. The formula gives a quite interesting result, illustrated by the table: even for a high number of tasks, the problem is schedulable for a **processor load which does not exceed approximately 70%.**

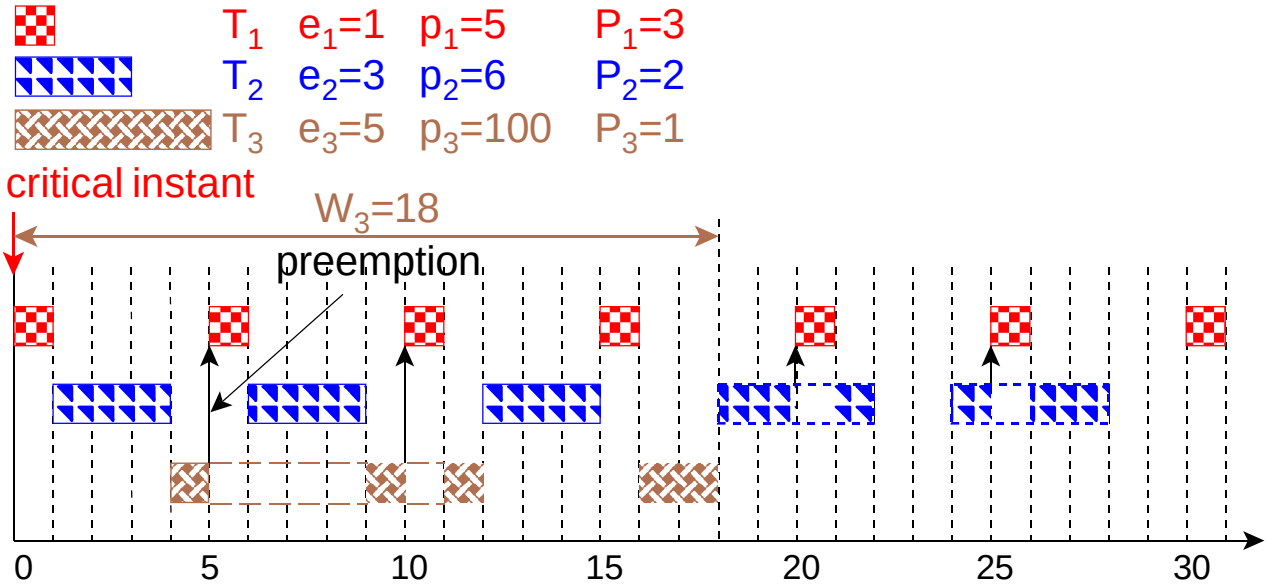
A necessary and sufficient condition was demonstrated later and utilizes the concept of **response time W**.
The formula expresses that the response time W_i between the start time and the end of the execution is equal, for each task:
- to its own execution time e_i
- PLUS a lapse of time due the preemption to execute higher priority tasks (see next slide)

For each task, it is necessary to find the smallest W_i value which is solution of this equation, then to check that it is indeed lower than the deadline, i.e., the period.

The notation $\lceil .. \rceil$ indicate that it is necessary to round to the next integer value.

RM is the base of several RTOS available on the market; it can be extended to less restrictive assumptions (synchronization, asynchronous tasks)

RM (*Rate Monotonic*): example



SC: $U = \frac{1}{5} + \frac{3}{6} + \frac{5}{100} = 0.75 < 3(\sqrt[3]{2} - 1) = 0.78$ OK

NSC: $W_3 = 5 + \left\lceil \frac{W_3}{5} \times 1 \right\rceil + \left\lceil \frac{W_3}{6} \times 3 \right\rceil \Rightarrow W_3 = 18 \Rightarrow p_3 > 18$

On this figure, we see an example of scheduling for 3 periodic tasks.
Task T_1 has got the highest frequency and thus priority. It will be executed with a strict periodicity, which will require preempting T_2 or T_3.
Task T_3 has got the smallest priority and is thus systematically preempted by T_1 and T_2.
All tasks have a phase equal to 0; t=0 thus constitutes for task T_3 a critical instant, because its execution will be delayed to the maximum by the execution of T_1 and T_2.

The sufficient condition of schedulability is respected

$$CS \Rightarrow U = \frac{1}{5} + \frac{3}{6} + \frac{5}{100} = 0.75 < 3(\sqrt[3]{2} - 1) = 0.78$$

This condition is not very severe here. Indeed, applying this formula gives a lower limit of 62,5 for period p_3 below by. However, looking at the figure, we can obviously reduce p_3 down to 30, case where the pattern of tasks of this figure will reproduce indefinitely.

The necessary and sufficient condition will enable us to calculate the most critical response time. Here is the calculation for W_3 which is the execution time of T_3 (=5) plus all the time "stolen" by the two other tasks

$$CNS \Rightarrow W_3 = 5 + \left\lceil \frac{W_3}{5} \times 1 \right\rceil + \left\lceil \frac{W_3}{6} \times 3 \right\rceil \Rightarrow W_3 = 18$$

Exercise: compute the minimal period for T2 and in this case, which becomes the period minimum for T3 ?

RTOS

DM (*Deadline Monotonic*)

- constrained deadline $D_i \leq p_i$
  - same as RM if $D_i = p_i$
- priorities**: static, sorted on $1/D_i$**
  - $D_i \searrow = P_i \nearrow$
- properties
  - optimal
- schedulability
  - CS [AB90]

$$e_i + \sum_{j=1}^{i-1} \left\lceil \frac{D_i}{p_j} \right\rceil \times e_j \leq D_i \quad \forall i$$

When constrained dealines are imposed, the priority is non longer sorted on the period, but naturally on the deadline and the algorithm is called deadline monotonic.

RTOS

EDF (*Earliest Deadline First*)

- **dynamic priorities** based on **absolute deadline**
  - $d_i \searrow \Rightarrow P_i \nearrow$
- properties
  - optimal, more flexible than RM
- schedulability
  - $\mathbf{D_i = p_i}$ NSC : $\Sigma e_i / p_i \leq 1$
  - $\mathbf{D_i \leq p_i}$ NC : $\Sigma e_i / p_i \leq 1$ SC : $\Sigma e_i / D_i \leq 1$

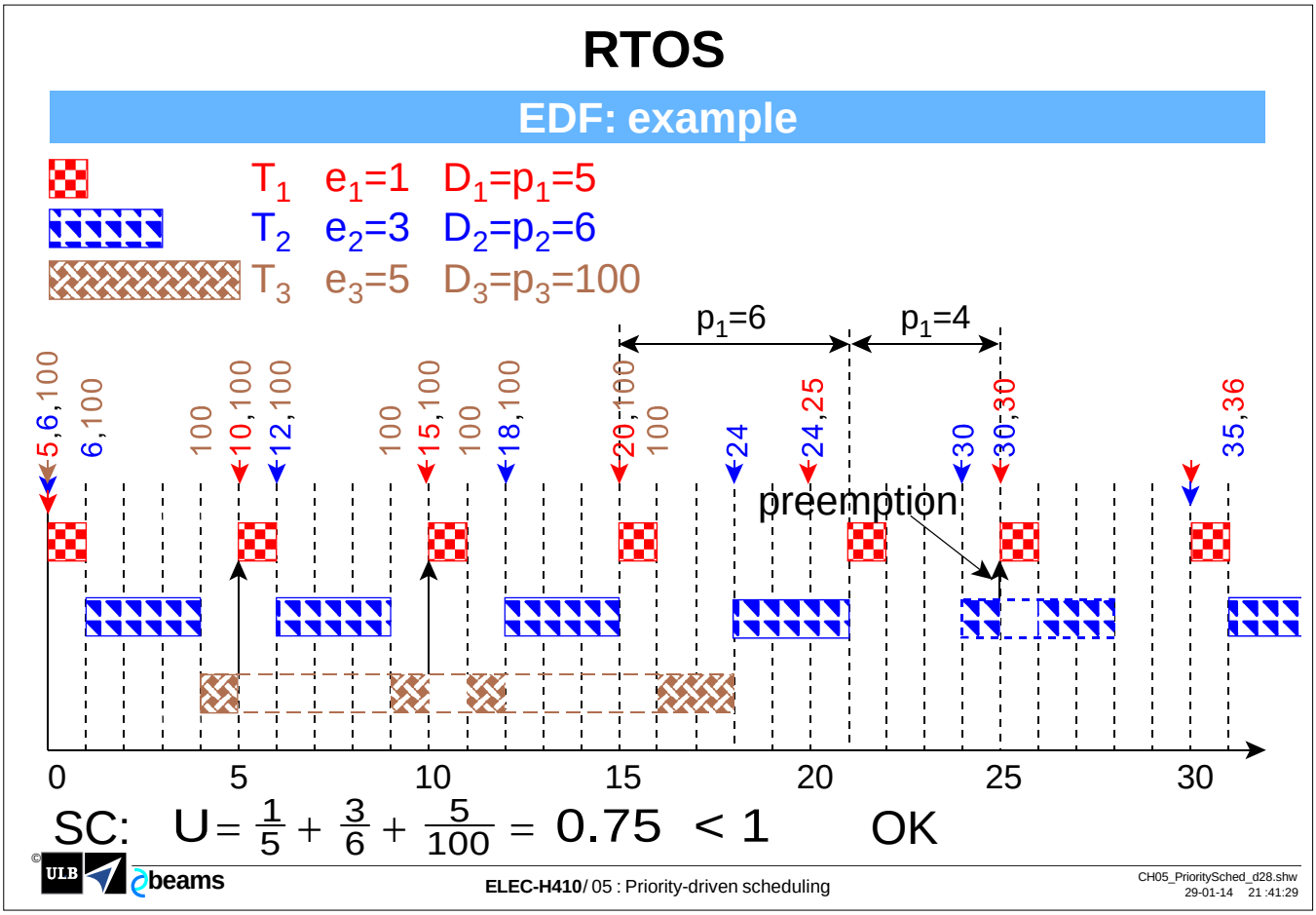
Another famous algorithm is the algorithm **EDF** (***Earliest Deadline First***)

Contrary to RM and DM, EDF is an algorithm based upon dynamic priorities. Each time the **scheduler** gets the control, it computes **which task** has got the **shortest absolute deadline** and starts it. It is the most intuitive algorithm, indeed.

It has been demonstrated that this algorithm is also **optimal**.

If we work with implicit deadlines ($D_i = p_i$) a necessary and sufficient condition of schedulability is simply the feasibility of the algorithm, i.e. a total utilisation ratio of the CPU lower than 1, which shows that EDF is more flexible than RM.

If the deadline is constrained, i.e. before the end of the period, U<1 becomes a necessary condition; a sufficient condition is based on a modified U, where the period p_i is replaced by the relative deadline D_i.



This figure illustrates the operation of algorithm EDF, with the same set of tasks used previously for RM. The arrows signal the instants of activation, the numbers give the instants of the deadlines:

- in t=0, the 3 tasks are activated: T_1 is started because it has got the shortest deadline
- in t=1, T_1 is finished; T_2 starts because its deadline is shorter than that of T_3
- in t=4, T_2 is finished; T3 remains the only "ready" task and starts.
- in t=5, T_1 is activated and has the shortest deadline, therefore T3 is preempted and T1 is started.

This mechanism continues in a way similar to the RM until t=18, when T_3 is finished. From this moment, RM and EDF take different decisions

- in t=20, T_1 is ready but does not start, because the deadline of T_2 is 24, whereas that of T_1 is 25
- in t=21, T_2 is finished and T_1 start; T1 has been delayed by 1 tick
- in t=24, T_2 is ready and started
- in t=25, T_1 is started and the deadlines of T_1 and T_2 are the same; we suppose here that the scheduler decides to preempt T_2, which will keep the correct average period for T_1, while meeting the deadline of T_2.

We see little difference between RM and EDF, because Di=pi, a priority based on the rate or on the deadline does not modify significantly the scheduling:

- RM results in a strictly periodic operation for T1 which gets always the highest (static) priority
- EDF distorts the period of T_1 (p_1=6, followed by p_1=4) because of the dynamic allocation of priority

Algorithms RM and EDF would give more different results with constrained deadlines (Di < pi); let us reduce for example D2=4, i.e. a deadline shorter than p_2 but also than p_1. The behaviour of RM will not be modified whereas EDF will give more often the priority to T_2.

RTOS

LL (*Least Laxity*)

- **dynamic priorities** based on **laxity**
  - $L_i \searrow \Rightarrow P_i \nearrow$
- properties
  - optimal as EDF
- drawback
  - creates more premptions
  - computing priorities consumes more CPU

Priority-driven scheduling

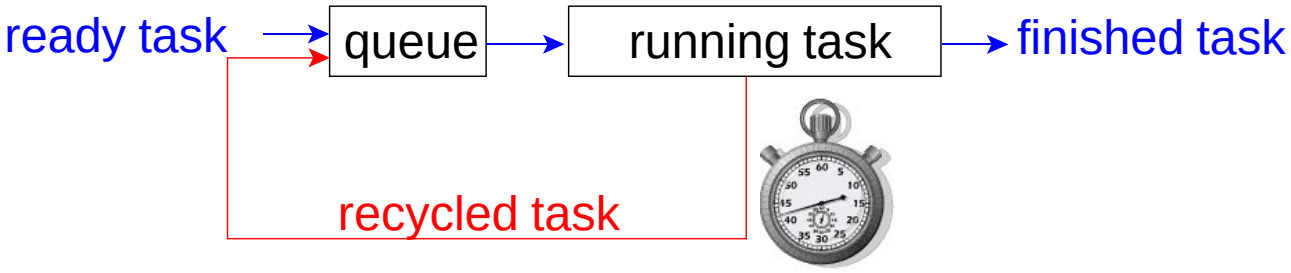
CONTENTS

- foreground/background systems
- **RTOS**
  - principles
  - inter task communication
  - managing interrupts
  - **main scheduling algorithms**
    - RM, EDF, DM, LL
    - **round robin**
    - multiprocessor
- conclusions

RTOS

time- slicing and round-robin

- well-known in normal OSes
- for tasks of same priority



If two or more tasks have the same priority, we can use the **round-robin** algorithm to give them an equitable access to the CPU.

The "ready" tasks are placed in a queue, the scheduler picks the first one and starts it for a given lapse of time.
- if the task exceeds the allocated duration, it is preempted and recycled at the end of the queue
- if the task is shorter, it gives the control back to the scheduler by an OSSuspend() call

The task which is at the head of the queue is started.

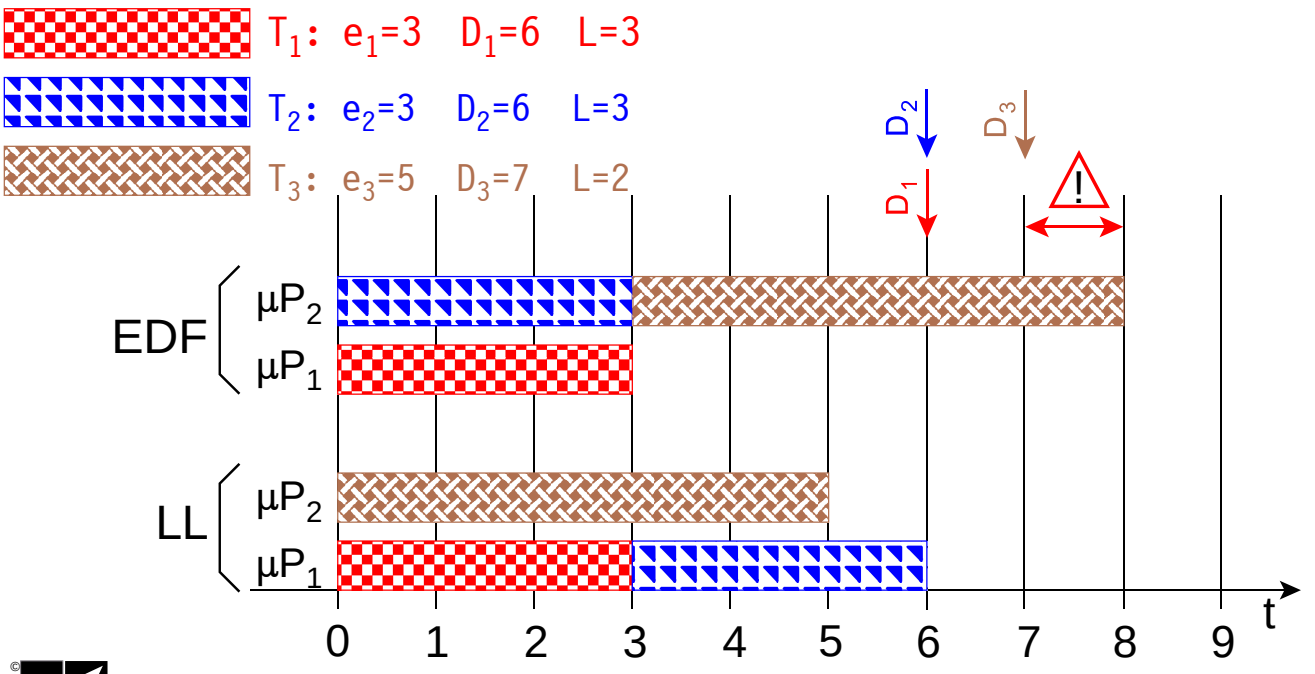
Priority-driven scheduling

CONTENTS

- foreground/background systems
- **RTOS**
  - principles
  - inter task communication
  - **main scheduling algorithms**
    - RM, EDF, DM, LL
    - round robin
    - **multiprocessor**
  - dependancies
- conclusions

RTOS

EDF is not optimal in multiprocessor



It is much more difficult to guarantee the optimality of a scheduling algorithm in a multi-processor environment.

We will not enter into details in this course, let us simply show by this example that EDF is no longer optimal in that case.

Let us suppose that three tasks T1, T2 and T3 are ready at the same time in t=0; the scheduler will give the priority to the tasks T1 and T2 and start them simultaneously on the two processors.
There is then no other solution but to start T3 in t=3, on any processor; unfortunately the deadline of T3 is not respected.

In this case, the Least Laxity can give better results.

In t=0, task T3 has got the highest priority because its laxity

$$L_3 = D_3 - e_3 = 7-5 = 2$$

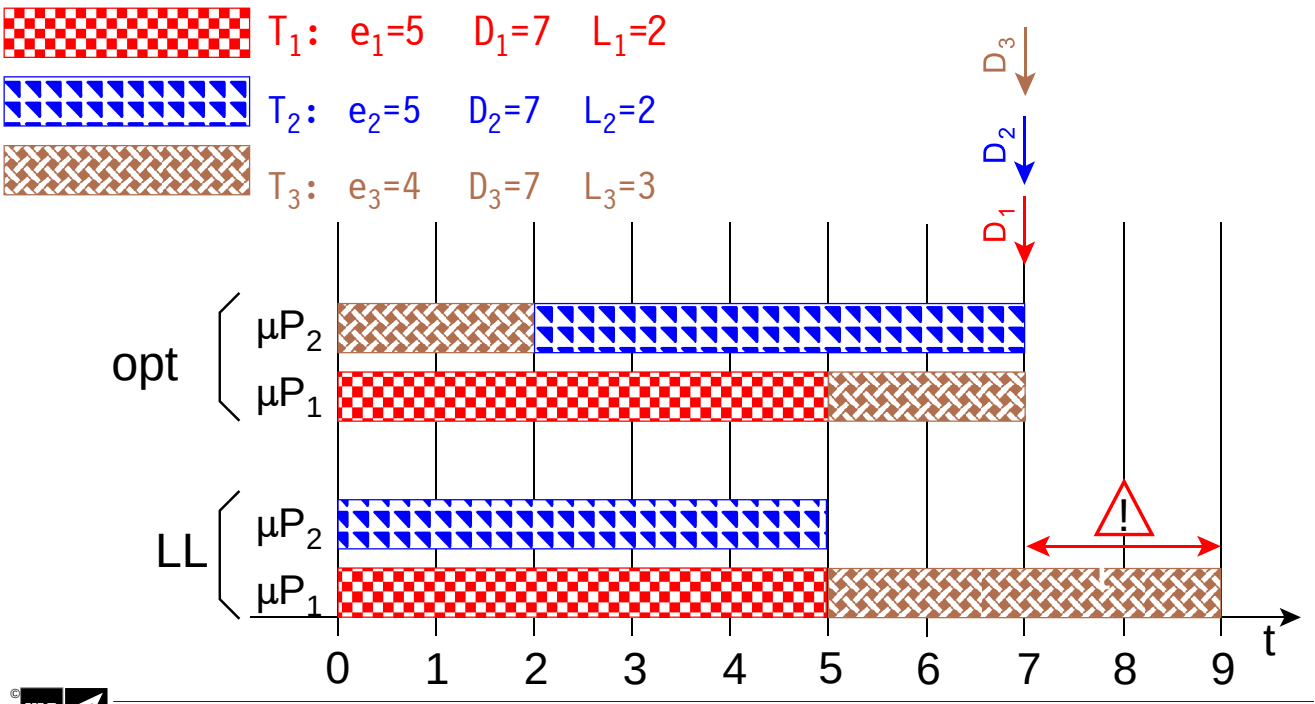
whereas the two other tasks have a laxity of

$$L_1 = L_2 = 6-3 = 3$$

LL will start T3 on a processor and T1 or T2 on the other one and all the deadlines are met.

RTOS

LL is not optimal in multi-processor



The previous example could make us believe that LL is optimal; this counterexample proves that it is not true.

In t=0, T_1 and T_2 have got the least laxity (L_1=L_2=2; L_3=3) and are started, which delays the start of T_3 in t=5, which is too late.

A better algorithm exists, provided T3 can be distributed on the two processors.

Priority-driven scheduling

CONTENTS

- foreground/background systems
- RTOS
- **conclusions**

RTOS

non-preemptive scheduler: conclusions

- advantages : a task is always finished before the next task is executed
  - few risks of corruption of shared data
  - critical sections or semaphores only requires for shared I/O
  - IRQs generally not masked => low latency for IISR
  - functions can be compiled in non-reentrant mode (more compact and efficient)
- drawbacks
  - task latency is ≈ worst-case execution of longest task (better than foreground/background)
  - non-deterministic, even for the highest priority task
  - risk of hanging if a task does not give back control (infinite loop without time-out)

The advantages of a non-preemptive kernel are related to the fact that each task finishes (or suspends itself) before the next one starts:
- there is **no conflict of access** to the data shared between several tasks
- **critical sections or semaphores are not necessary** , except for sharing some slow peripherals, like printers or serial communication links
- the interrupts are not often masked and the **latency time of IISR** is thus **short**
- the tasks can resort to functions written in **non-reentrant code,** which is more effective

The principal disadvantages are:
- the **latency time** in the execution of the **DISR** can be long (chiefly determined by the longest task), which is anyway it is better than in a foreground/background system (duration of the whole background)
- the **absence of determinism** in the latency of a task with a given priority
- the risk of **total locking** if a task does not return the control to the scheduler; the trivial case is a task waiting (by polling) an external event which never comes (e.g. the tank does not fill because the fuse of the pump is blown), whereas the programmer did not foresee a **time-out**

Never let any task wait for an event without time-out.

RTOS

preemptive scheduler: conclusions

- advantages
  - priorities: better respect (not perfect though)
  - response time: optimum and deterministic (at least for the task of highest priority)
- drawbacks
  - a preempted task is susceptible to share resources with the running task
    - reentrant code required for any function which can be called by 2 tasks
    - semaphore/critical sections requires for shared resources (buffer, I/O)
  - kernel consumes more resources (CPU, registers, ROM and RAM)

The premptive sheduling gives a better respect of priorities.

Do not forget that most mathematical proofs of scheduling algorithms make the assuption that the kernel is preemptive.

This better quality of service has a price to pay
- the kernel takes more resources (CPU time, memory)
- the tasks have to be compiled in reentrant mode, because the RTOS must work in a dynamic environment and assume that all tasks save their data on a stack. Its is not impossible to write functions in non-reentrant mode, but you must be sure that they will never be interrupted and called again.
- the risk of corruption of shared data means a rather extensive use of critical sections, semaphores and mailboxes.

Conclusions

- clock-driven systems
  - sure and simple
  - if asynchronous events, works fine if total loop time < required response time
- RTOS
  - tasks written individually
  - management of priorities always crucial
  - more difficult to guarantee determinism in all cases
- do not forget
  - hardware tasks
    - internal or external peripherals (e.g. PWM)
  - multiprocessor + real-time network

Only the synchronous systems are entirely deterministic and they are made above all to manage periodic and sporadic tasks which are perfectly known at design time.

Aperiodic tasks with a "soft" deadline can be taken into account of if the slack is sufficient and if the sampling period (or micro-cycle) is shorter than the relative deadline.

Aperiodic tasks with a "hard", but bounded, deadlines can be managed at the price of a wasting of CPU power, which is not necessarily awkward nor expensive.

Many safe/secure systems are clock-driven.

Using an **RTOS is much easier for the programmer**, because each task can be coded individually. Care must be taken to define the priorities of the tasks. RTOS are characterized by **low interrupt latency** and **low context-switching latency** (a few µs). Lots of researches have led to robust RTOS based on **proven scheduling algorithms** but under some hypothesis. Nevertheless, no RTOS can offer an absolute guarantee of scheduling any pattern of tasks, including aperiodic ones.

Lastly, considering the remarkable performance/price ratio of current microcontrollers, real-time problems can be solved by decentralization over a set of small processors managing a few tasks. Those processors have to communicate, hence the development of real-time networks.

Cabled logic, or better programmed logic (PLD, FPGA) remains an invaluable solution for fast critical systems because they can bring massive parallelism and reaction times as low as a few tens of ns.