Previous lecture

Timing constraints: origin and characterization

- Basic concepts of *real-time*
- Real-Time Systems requirements

Computing models

- Models of tasks with explicit time constraints
- Logic and temporal control (by events -ET and by time -TT)

Embedded Systems - Real-Time Scheduling

part 3

Basic scheduling concepts

Task scheduling, basic taxonomy
Preliminary task scheduling methods
Cyclic static scheduling

Temporal complexity

- Measure of how the **execution time** of an algorithm **grows** when we **increase the problem size** (i.e. width of input data)
- Expressed with operator **O( )**
- Algebra of the operator O( ), n=problem size, k=const.
  - O(k) = O(1)
  - O(kn) = O(n)
  - $O(k_1 n^m + k_2 n^{m-1} + ... + k_{m+1}) = O(n^m)$

```
for (k=0;k<N;k++)
  a[k]=0;
```

Compl. = O(N)

```
for (k=0;k<N-1;k++)
  for (m=k;m<N;m++)
    if a[k]<a[m]
      swap(a[k],a[m]);
```

Compl. = $O(N^2)$

Determine the permutations of a set $A=\{a_i, i=1..N\}$

Compl. = $O(N^N)$

Temporal complexity

- **P** and **NP classes** in decision problems
- P – problem that can be solved in polynomial time, O(p(N))
- NP – problem that cannot be solved in polynomial time but in which each solution can be verified in polynomial time

- Notion of **NP-complete** and **NP-hard**

- The temporal complexity is a measure of the performance of an algorithm (e.g. scheduling), the lower the better!

Schedule definition

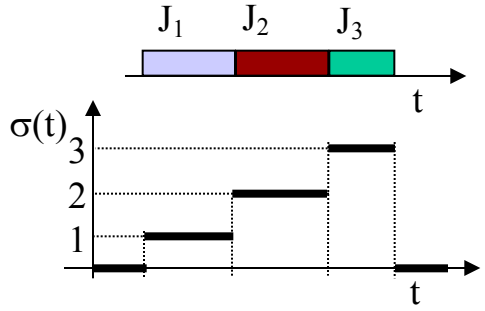
Task scheduling

- Sequence of tasks execution in one or more processors
- Function of R^+ (time) onto N_0^+ (task set), creating a correspondence between each instant of time *t* and the task *i* that is executing at that instant.

$$\sigma: R^+ \rightarrow N_0^+$$
$$i = \sigma(t),\ t \in R^+ \qquad (i=0 \Rightarrow \text{idle processor})$$

- $\sigma(t)$ is a step function and its graph is a *Gantt chart*

$\mathbf{J=\{J_1, J_2, J_3\}}$
(task set - jobs) →



Schedule definition

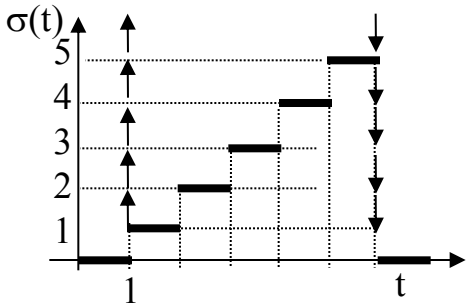
•A **schedule** is said to be **feasible** if it meets all the constraints associated to the task set
(**temporal**, non-preemption, shared resources, precedences…)

•A **task set** is said to be **schedulable** if there is at least one **feasible schedule** for that task set.

Scheduling problem

- Input:
  - A **task set**
  - Associated **constraints** (or cost function)
- Find an **Assignment** of processor time to **tasks** that allows:
  - **Executing tasks completely**
  - **Meeting** their **constraints** (or minimizing a cost function)

e.g. $J = \{J_i\ (C_i=1, a_i=1, D_i=5), i=1..5\}$ →



Scheduling algorithm

•A **scheduling algorithm** is a method to solve a scheduling problem.

Note: do not confuse the scheduling *algorithm* with the *schedule* itself

Classification of scheduling algorithms:

- **Preemptive** *versus* **non-preemptive**
- **Static** *versus* **dynamic**
- ***Off-line*** *versus* ***on-line***
- **Optimal** *versus* **sub-optimal** (heuristic)
- With worst-case **guarantees** *versus* **best effort**
- **Work-conserving** *versus* **non-work-conserving**

Preliminary algorithms

EDD – Earliest Due Date (Jackson, 1955)

Single job tasks and triggered synchronously:

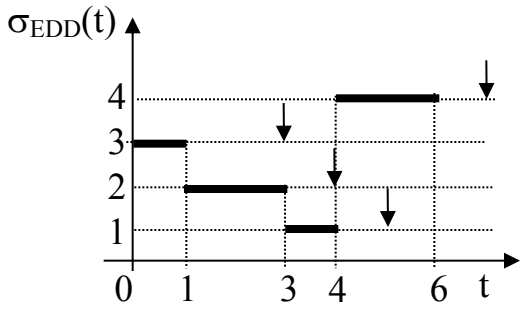
$J = \{ J_i (C_i, a_i=0, D_i) \; i=1..n\}$

Executing the tasks by **non-decreasing deadline order minimizes the maximum lateness** $L_{max}(J) = \max_i (f_i - d_i)$

O(n.log(n))

e.g. $J = \{J_1(1,5), J_2(2,4), J_3(1,3), J_4(2,7)\}$

$L_{max,EDD}(J) = -1$



Preliminary algorithms

EDF – Earliest Deadline First (Liu and Layland, 1973; Horn, 1974)

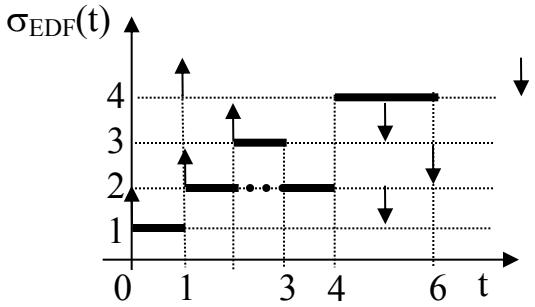
Periodic tasks, asynchronous, preemptive:
$J = \{ J_i (C_i, a_i, D_i)\ i=1..n)\}$

Executing at each instant the task with the **earliest deadline minimizes the maximum lateness** $L_{max}(J) = \max_i (f_i - d_i)$

O(n.log(n)), Optimal among all in its class

$J = \{J_1(1,0,5), J_2(2,1,5), J_3(1,2,3), J_4(2,1,7)\}$

$L_{max,EDF}(J) = -2$



Preliminary algorithms

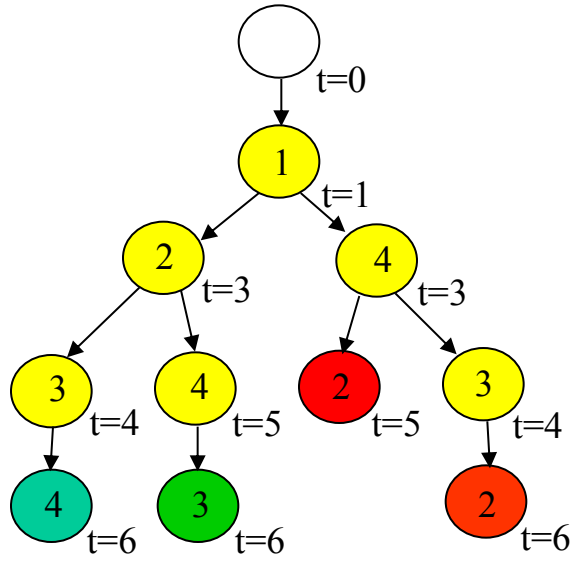
BB – Branch and Bound (Bratley, 1971)

Single job tasks, with offsets, non-preemptive:
$J = \{ J_i (C_i, a_i, D_i)\ i=1..n\}$

Schedule construction by **exhaustive search** in the space of possible permutations (schedule tree)

O(n!)

$J = \{J_1(1,0,5), J_2(2,1,3), J_3(1,2,4), J_4(2,1,7)\}$



$L_{max}(J)= -2$ $L_{max}(J)= 0$

Scheduling periodic tasks

Activation instants are known a priori

$\Gamma = \{ \tau_i (C_i, \Phi_i, T_i, D_i, \ i=1..n)\} \rightarrow \ a_{i,k} = \Phi_i + (k-1)T_i$

Thus, the schedule can be determined either

- **With the system executing (*on-line*)**
  the next task is chosen as the system executes.
- **Before system execution (*off-line*)**
  the order of execution is determined before the system is deployed and it is stored in a table that is read at run-time to trigger the tasks execution (**static cyclic scheduling**).

Static cyclic scheduling

The table is organized in **micro-cycles (uC)** with fixed duration so that going through all cycles generates the periodic pattern of tasks activation.

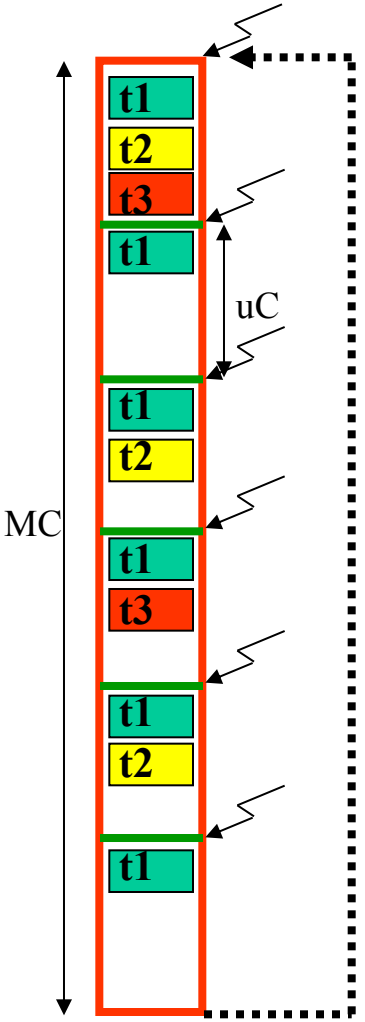
The micro-cycles are triggered by a timer.

Going through the whole table continuously results in a cyclic pattern called **macro-cycle (MC)**

$\Gamma = \{ \tau_i (C_i, \Phi_i, T_i, D_i, \ i=1..n)\}$

$uC = GCD(T_i)$
$MC = LCM(T_i)$

$\Phi_i = 0$,$C_i = 1ms$,
$T_1 = 5ms$
$T_2 = 10ms$
$T_3 = 15ms$



Static cyclic scheduling

Building the schedule table

- Determine the micro-cycle uC and the macro-cycle MC
- Express periods and offsets in micro-cycles
- Determine the cycles in which tasks are activated
- Using an adequate scheduling criterion, determine the order of execution of the active tasks
- Verify whether all tasks active in a micro-cycle can be completely executed within that cycle. Otherwise, other tasks will need to be delayed for the following cycles
- It maybe be necessary to break a task in several parts to make it fit the micro-cycles duration.

Static cyclic scheduling

Pros

- Simple implementation (timer+table)
- Low execution overhead (*dispatcher*)
- Allows optimizing the schedule
  (e.g. jitter control, precedences,…)

Cons

- Not scalable (small changes in the task set may cause significant changes in the schedule table, possibly large tables!)
- Not robust to transient overloads (sensitive to domino effect)

Wrapping up

- The concept of **temporal complexity**
- Definition of **schedule** and **scheduling algorithm**
- Some **preliminary scheduling techniques** (EDD, EDF, BB)
- **Static cyclic scheduling**

Embedded Systems - Real-Time Scheduling

part 4

Fixed priorities scheduling

On-line scheduling with fixed priorities
The Rate-Monotonic criterion – CPU utilization upper bound
The Deadline-Monotonic and arbitrary fixed priorities criteria
– worst-case response time analysis

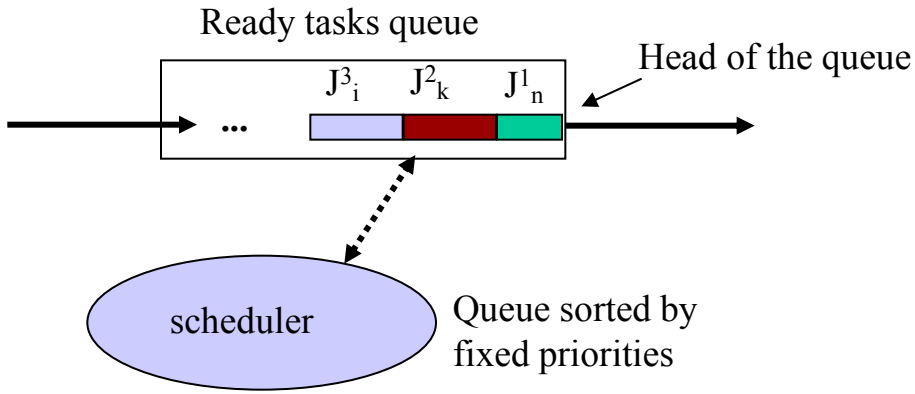
On-line scheduling with fixed priorities

The schedule is actually **built** with the system **working** (*on-line*) and is based on a **static parameter** (priority).

The ready tasks queue is sorted by descending priorities. The **first** to execute is the one with **higher priority** (head of the queue).

When preemption is allowed - Whenever at the head of the queue arises a task that has priority higher than the one that is currently executing, this one is suspended (returns to the queue) and the new one is executed.

Complexity **O(n)**



On-line scheduling with fixed priorities

Pros

- Easily scalable
  - the scheduler considers any tasks that become ready at any time
- For the same reason, easily accommodates sporadic tasks
- Deterministic behavior under overloads
  - just affects tasks with priority lower than that of the overload

Cons

- More complex implementation than the cyclic executive
  - needs a fixed priorities multi-tasking kernel
- Higher run-time overhead: *scheduler* + *dispatcher*
- Overloads at high priority levels can block the whole system

On-line scheduling with fixed priorities

Assigning priorities

- Inversely proportional to the period (***RM – Rate Monotonic***)
  (optimal with respect to all fixed priority criteria)
- Inversely proportional to the *deadline* (***DM – Deadline Monotonic***)
  (optimal when D<=T)
- Proportional to the tasks **importance**
  (may cause a reduction in efficiency – non optimal)

On-line scheduling with fixed priorities

Verifying schedulability

Since the schedule is built *on-line*, only, it is important to determine *a priori* whether a given task set will meet its associated timing constraints.

There are two main kinds of schedulability tests:

- Based on **CPU utilization**
- Based on **response time**

Rate-Monotonic Scheduling

Utilization-based tests for RM scheduling

(consider preemption, *n* independent tasks and D=T)

- **Least upper bound** of Liu&Layland (1973)

$$U(n) = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right) \Rightarrow$$ **one instance guaranteed per period**

- **Hyperbolic Bound** of Bini&Buttazzo&Buttazzo (2001)

$$\prod_{i=1}^{n} \left(\frac{C_i}{T_i} + 1\right) \leq 2 \Rightarrow$$ **one instance guaranteed per period**

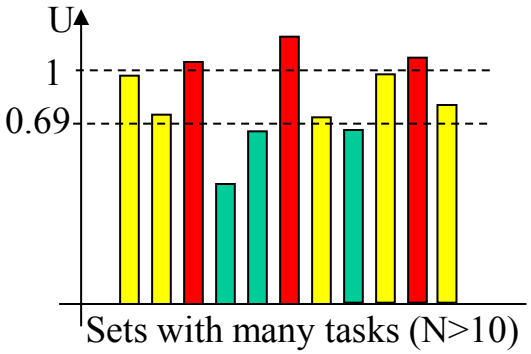
Rate-Monotonic Scheduling

Meaning of the Liu&Layland least upper bound (LUB)

U(n) > 1 ⇒ non schedulable set (*overload*) – necessary condition

U(n) ≤ LUB ⇒ schedulable set – sufficient condition

1 ≥ U(n) ≥ LUB ⇒ indetermined case

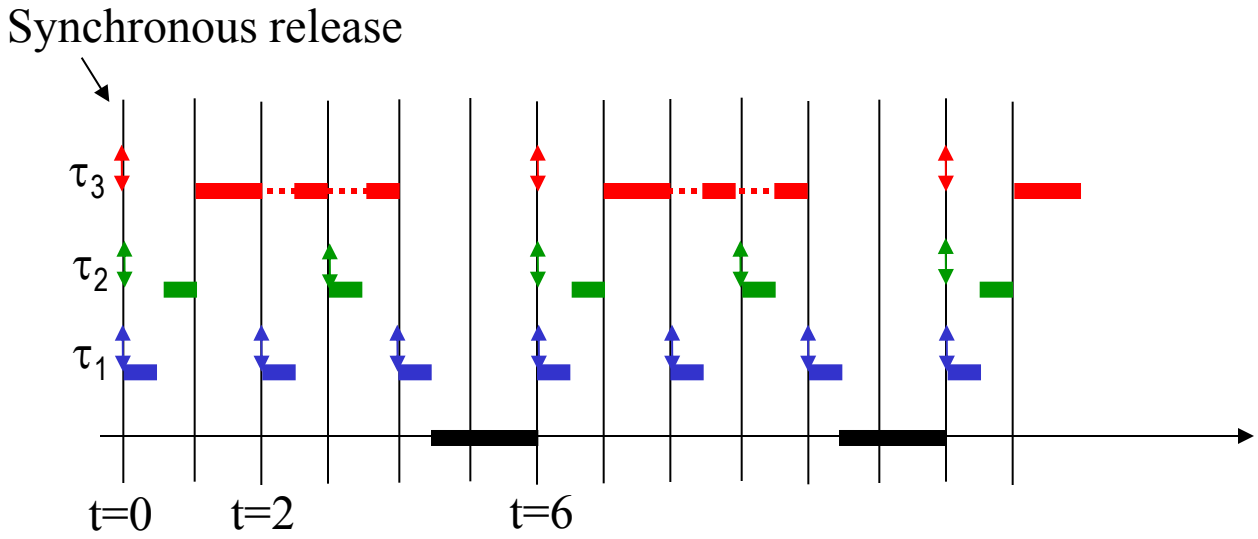


U(1) ≤ 1
U(2) ≤ 0.83
U(3) ≤ 0.78
...
U(10) ≤ 0.72
...
U(∞) → ln(2) ~=0.69

Rate-Monotonic Scheduling – example 1

Task set

τ_i	T_i	C_i
1	2	0.5
2	3	0.5
3	6	2



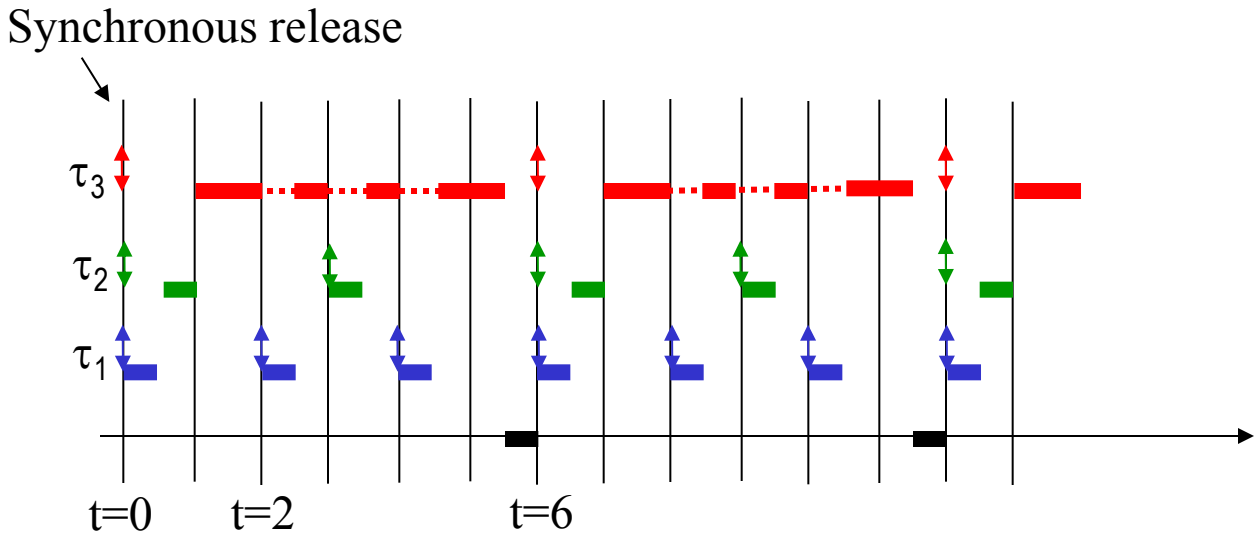
$U = 0.5/2 + 0.5/3 + 2/6$

$U = 0.75 < 0.78$ => 1 instance per period assured

Rate-Monotonic Scheduling – example 2

Task set

τ_i	T_i	C_i
1	2	0.5
2	3	0.5
3	6	**3**

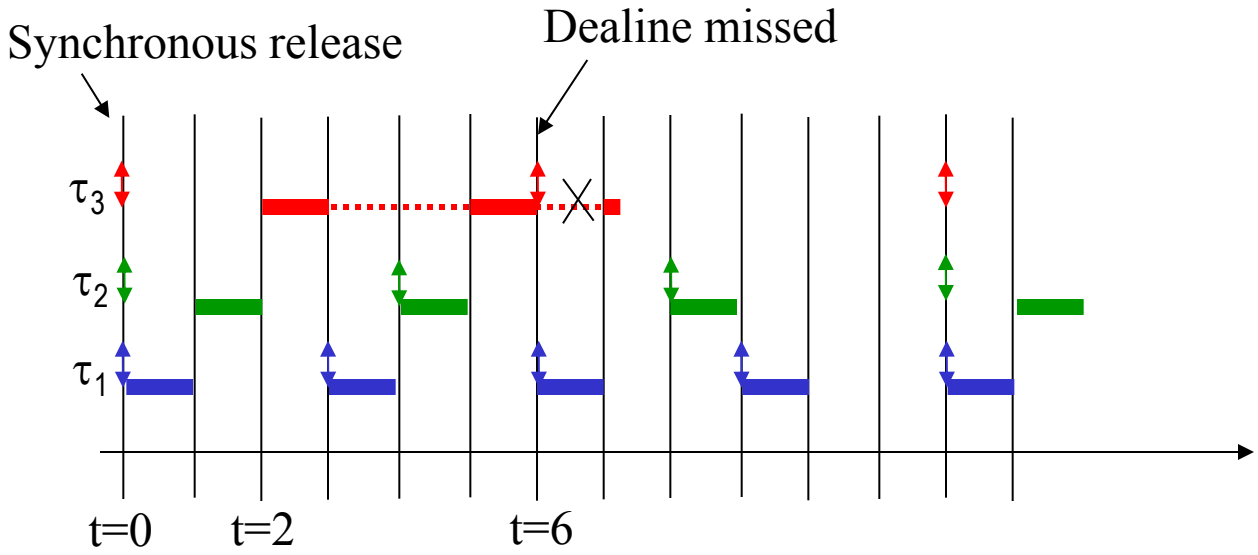


U = 0.5/2 + 0.5/3 + 3/6 = **0.92 > 0.78** => **1 instance per period NOT assured but the schedule is feasible**

Rate-Monotonic Scheduling – example 3

Task set

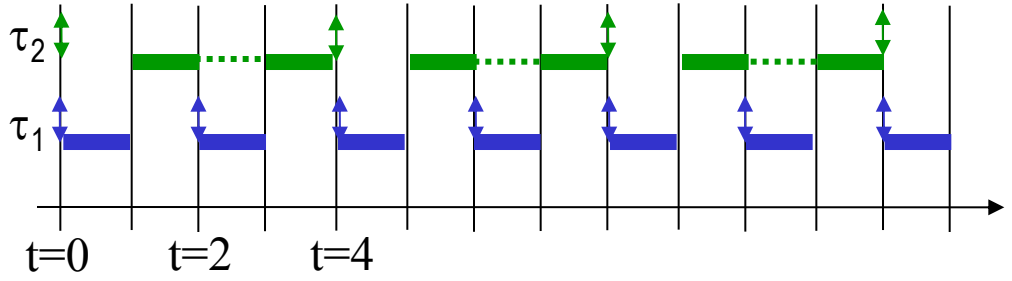
τ_i	T_i	C_i
1	3	1
2	4	1
3	6	2.1



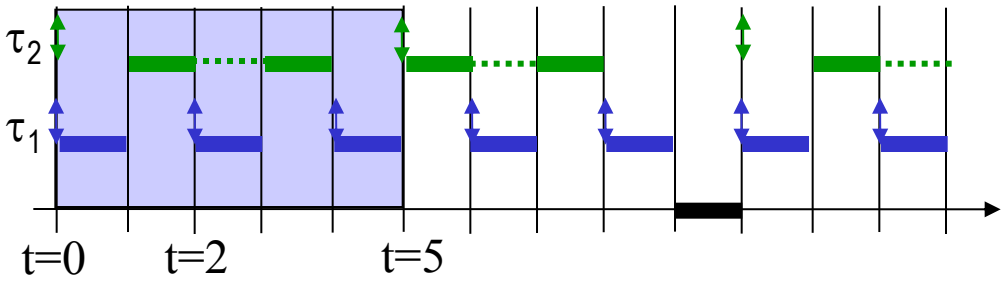
U = 1/3 + 1/4 + 2.1/6 = 0.93 > 0.78 => 1 instance per period NOT assured and the schedule is infeasible with a deadline miss by τ_3

Rate-Monotonic Scheduling – special cases

$\mathbf{U = 1/2 + 2/4 = 1}$



$\mathbf{U = 1/2 + 2/5 = 0.9}$



Deadline-Monotonic Scheduling

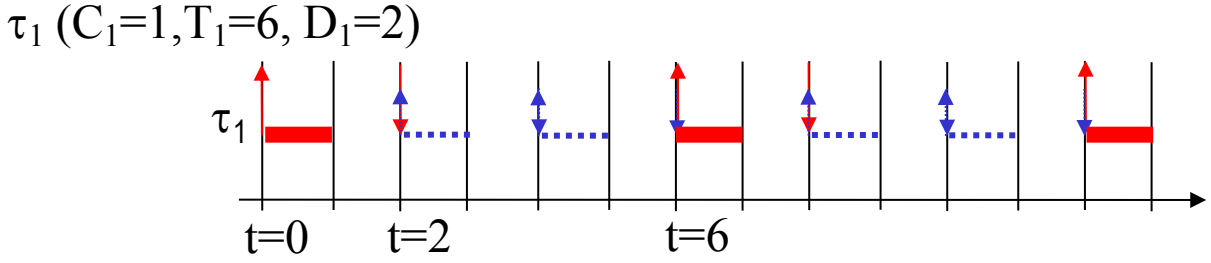
Schedulability tests for Deadline-Monotonic

When a task has a long period but needs to be served quickly after release then the RM criterion is non-optimal.

Better scheduling tasks according to their *deadline* instead of period.

Utilization-based schedulability tests can still be used considering the deadline instead of period, i.e. $\boldsymbol{U(n) = \sum_{i=1}^{n} \frac{C_i}{D_i} \leq n\left(2^{\frac{1}{n}} - 1\right)}$

However, this test easily becomes very pessimistic!



Response time analysis

For **arbitrary fixed priorities**, including RM, DM and any other, the **response time analysis** provides a schedulability test that, in the specified conditions (preemption and synchronous release plus independence) is **necessary and sufficient** (i.e. exact!).

Worst-case response time = largest time interval since a task is released until is finishes.

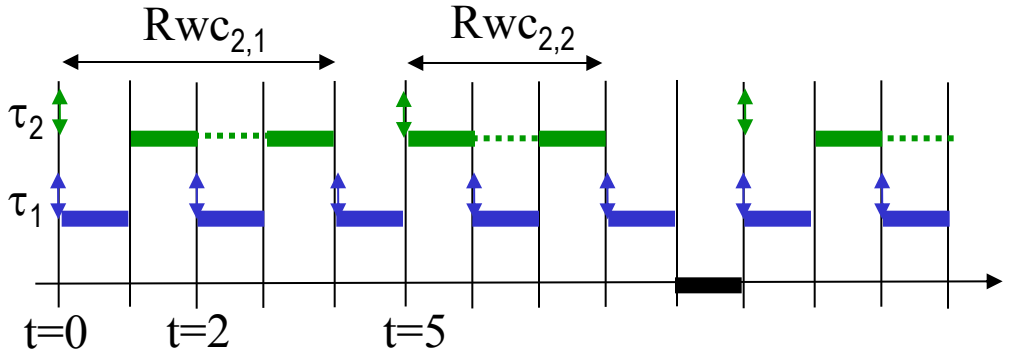
$$Rwc_i = \max_k \left(f_{i,k} - a_{i,k} \right)$$

Response time-based schedulability test

$$\forall_i \; Rwc_i \leq D_i \quad \Leftrightarrow \quad \textbf{schedulable set}$$

Response time analysis

The worst-case response time of a task occurs when that task is released together with all higher priority tasks (**critical instant** that in this case is the **synchronous release**)



Computing Rwc_i

$$\forall_i \; \boldsymbol{Rwc_i = I_i + C_i}$$

I_i – interference caused by the execution of tasks with higher priority

$$\boldsymbol{I_i = \sum_{k \in hp(i)} \left\lceil \frac{Rwc_i}{T_k} \right\rceil * C_k}$$

Number of times that higher priority task k is released in the interval Rwc_i

Response time analysis

Solving the non-linear response time equation can be carried out with a **fixed point iterative technique** that either diverges ($Rwc_i > D_i$) or **converges in a finite number of steps** ($Rwc_i(m+1) = Rwc_i(m) = Rwc_i$)

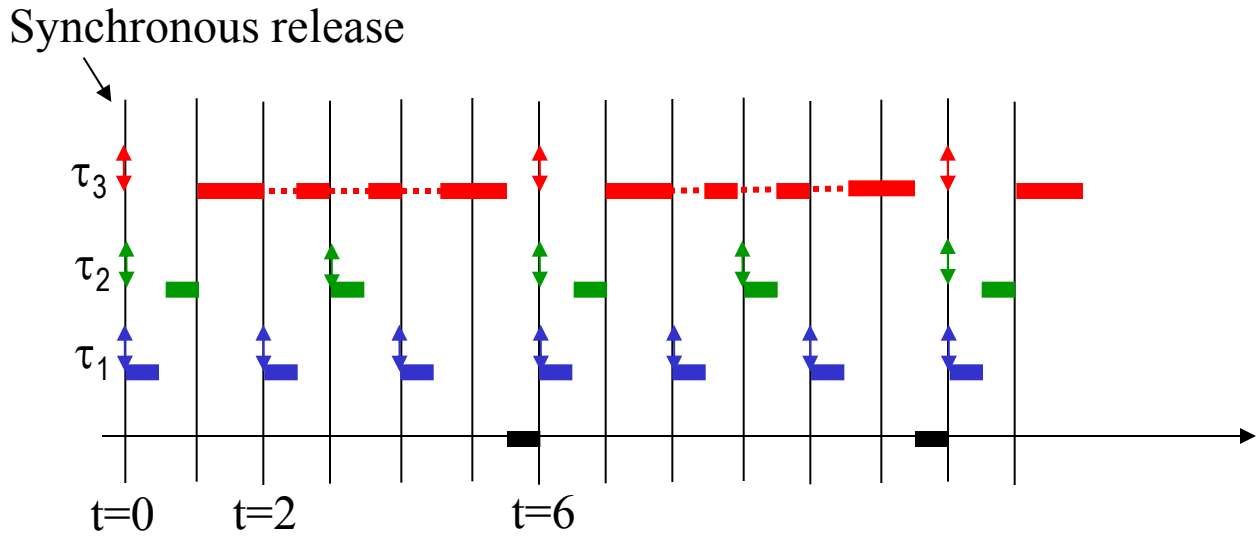
$$\boldsymbol{Rwc_i(0) = \sum_{k \in hp(i)} C_k + C_i}$$

$$\boldsymbol{Rwc_i(m+1) = \sum_{k \in hp(i)} \left\lceil \frac{Rwc_i(m)}{T_k} \right\rceil * C_k + C_i}$$

Response time analysis

Task set

τ_i	T_i	C_i
1	2	0.5
2	3	0.5
3	6	3



Rwc_3: $Rwc_3(0) = C_1+C_2+C_3 = 4$

$Rwc_3(1) = \lceil Rwc_3(0)/T_1 \rceil * C_1 + \lceil Rwc_3(0)/T_2 \rceil * C_2 + C_3 = 5$

$Rwc_3(2) = \lceil Rwc_3(1)/T_1 \rceil * C_1 + \lceil Rwc_3(1)/T_2 \rceil * C_2 + C_3 = 5.5$

$Rwc_3(3) = \lceil Rwc_3(2)/T_1 \rceil * C_1 + \lceil Rwc_3(2)/T_2 \rceil * C_2 + C_3 = 5.5$

$Rwc_3 = 5.5$

Restrictions to previous analysis

The previous analysis need to be adapted in the presence of:

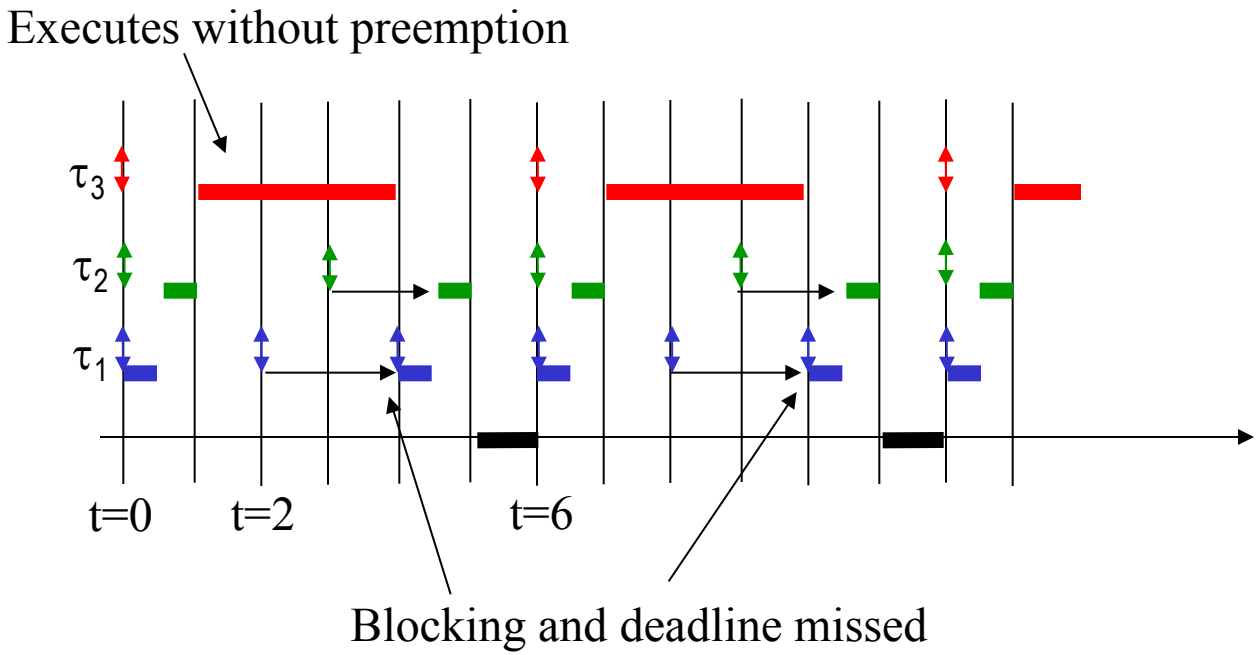
- **Non-preemption**
- **Non independence:**
  - **Shared resources**
  - **Precedences**

- And for a more accurate result, it is also necessary to consider
  - the **operating system** or **kernel overhead** (e.g., context switches, time management associated to the clock tick…)
  - the time taken in **interrupt handling** in general

Impact of non-preemption

Task set

τ_i	T_i	C_i
1	2	0.5
2	3	0.5
3	6	**3**



Wrapping up

- **On-line** scheduling using **fixed priorities**
- The ***Rate Monotonic*** scheduling criterion
  - schedulability analysis based on **utilization**
- The **Deadline Monotonic** scheduling criterion and the scheduling with **arbitrary fixed priorities**
  - Schedulability analysis based on the **worst-case reponse time**.

Embedded Systems - Real-Time Scheduling

part 5

Dynamic priorities scheduling

On-line scheduling with dynamic priorities
The Earliest Deadline First criterion – CPU utilization upper bound
Optimality compared to Rate-Monotonic with respect to
schedulability level, number of preemptions, release jitter and response time
Other dynamic priorities based criteria
– *Least Slack First, First Come First Served*

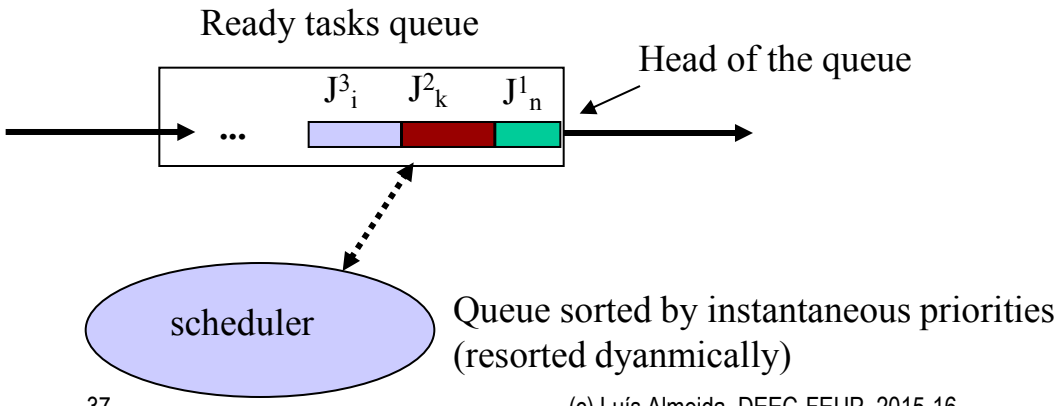
On-line scheduling with dynamic priorities

The actual schedule is **built** with the system **working** (*on-line*) and is based on a **dynamic parameter** (just known at scheduler run-time).

Such **dynamic parameter** used to guide the scheduling can be seen as a **dynamic priority**

The ready tasks queue is sorted by decreasing priorities whenever there is a change in relative priorities. The **first to run** is the tasks with **highest instantaneous priority**.

Complexity **O(n.log(n))**



On-line scheduling with dynamic priorities

Pros

- Easily scalable
  - the scheduler considers any tasks that become ready at any time
- For the same reason, easily accommodates sporadic tasks

Cons

- More complex implementation than a cyclic executive
  - needs a multi-tasking kernel with dynamic priorities
- Higher run-time overhead:
  - dynamic resorting of the ready tasks queue (depends on the algorithm)
- Instability with overloads
  - impossible to know a priori which tasks will meet their deadlines

On-line scheduling with dynamic priorities

Assigning priorities

- Inversely proportional to the time to the *deadline*
(***EDF – Earliest Deadline First***)
(optimal with respect to dynamic priorities criteria)

- Inversely proportional to the *slack or laxity*
(***LSF (LLF) – Least Slack First***)
(optimal with respect to dynamic priorities criteria)

- Inversely proportional to the *waiting time*
(***FCFS – First Come First Served***)
(may impose long blockings – non optimal)

On-line scheduling with dynamic priorities

Verifying schedulability

Since the schedule is built *on-line*, only, it is important to determine *a priori* whether a given task set will meet its associated timing constraints.

There are three main kinds of schedulability tests:

- Based on **CPU utilization**
- Based on **CPU demand**
- Based on **response time**

Earliest Deadline First Scheduling

Utilization-based schedulability tests for EDF

(with preemption and *n* independent tasks)

- D=T

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \Leftrightarrow \text{schedulable set}$$

Allows using **100% of the CPU guaranteeing the time constraints**

- D<T

$$U^* = \sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1 \Rightarrow \text{schedulable set}$$

U=density*

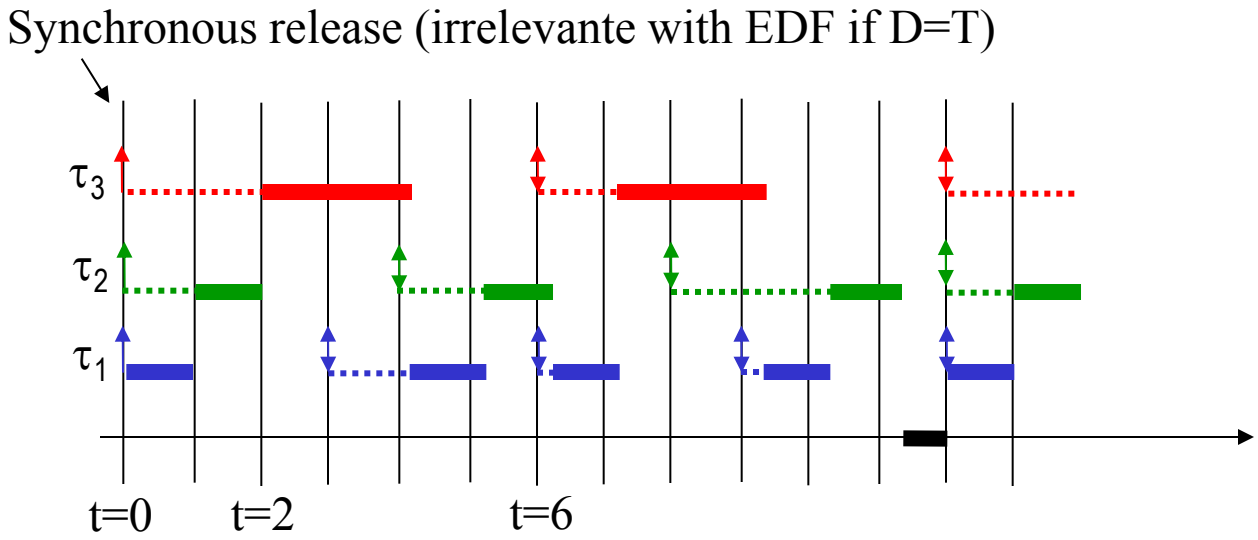
- Arbitrary D

$$\sum_{i=1}^{n} \frac{C_i}{min(D_i,T_i)} \leq 1 \ (\Leftrightarrow) \ \Rightarrow \text{schedulable set}$$

Earliest Deadline First Scheduling – example

Task set

τ_i	T_i	C_i
1	3	1
2	4	1
3	6	2.1

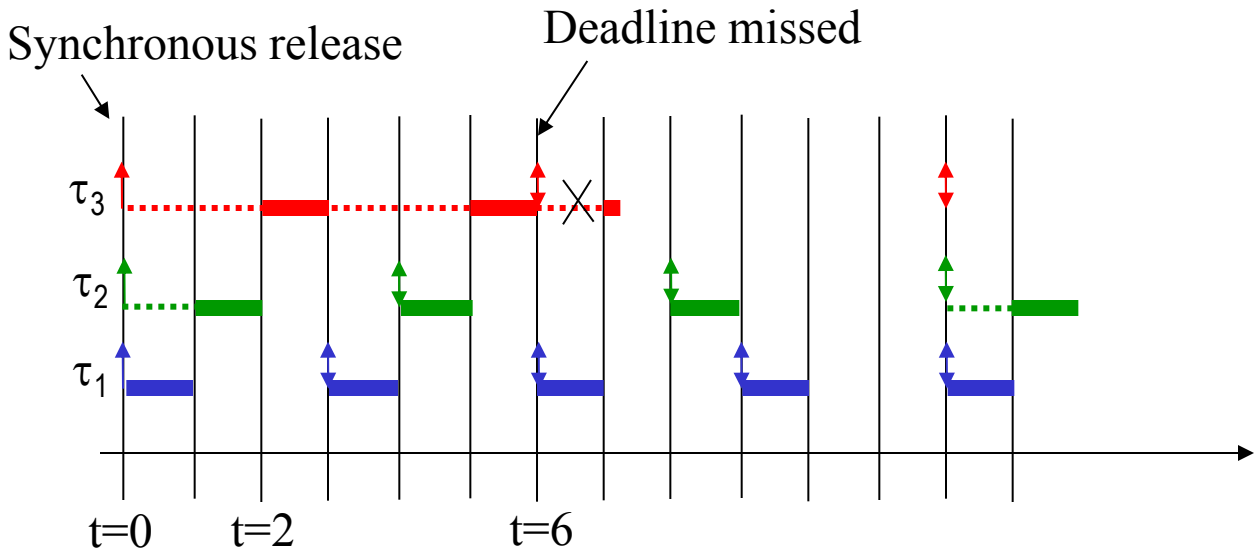


U = 1/3 + 1/4 + 2.1/6 = 0.93 ≤ 1 ⇔ 1 instance per period assured

Rate-Monotonic Scheduling – same example

Task set

τ_i	T_i	C_i
1	3	1
2	4	1
3	6	2.1

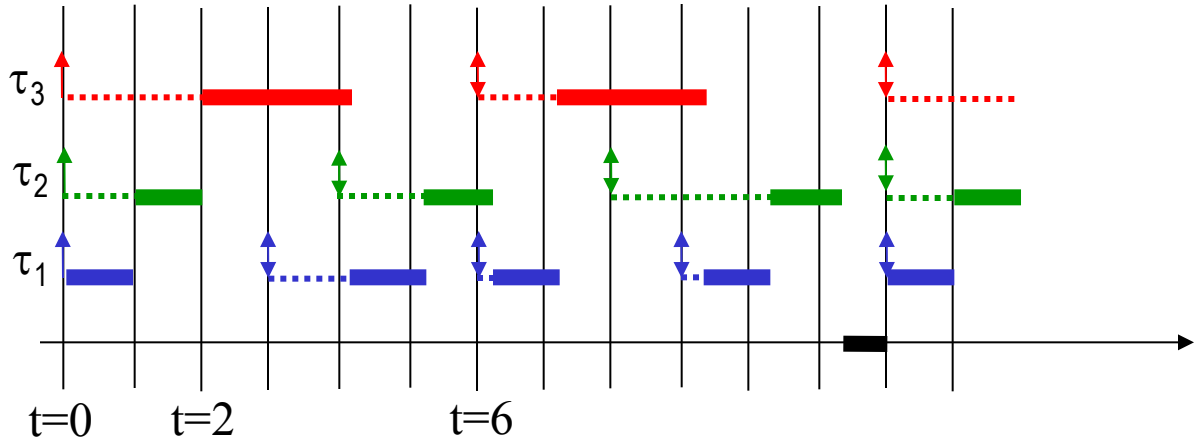


U = 1/3 + 1/4 + 2.1/6 = 0.93 > 0.78 ⇒ 1 instance per period NOT assured and the schedule is infeasible with a deadline miss by τ_3

Earliest Deadline First Scheduling – example

Task set

τ_i	T_i	C_i
1	3	1
2	4	1
3	6	2.1



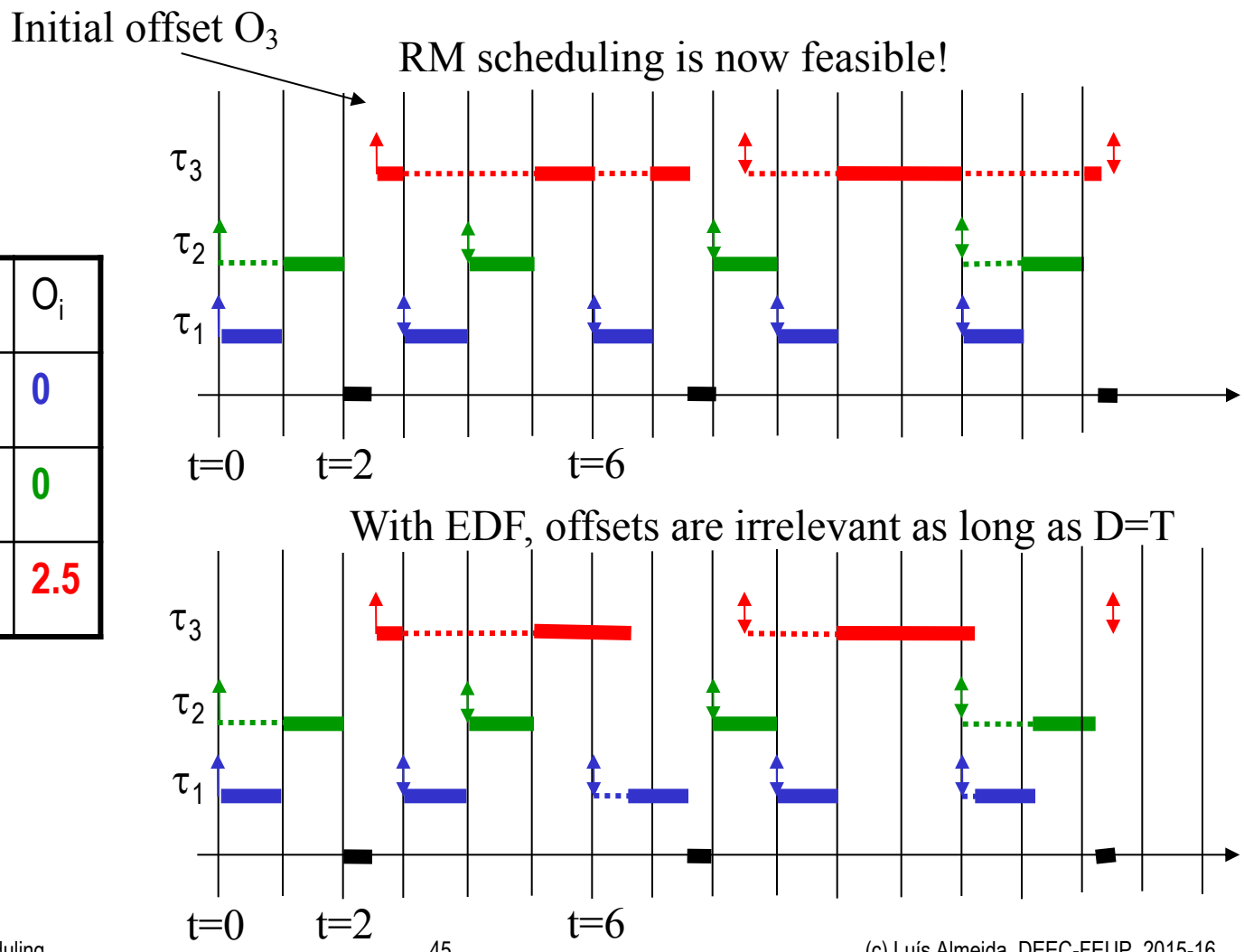
Note:

- No deadlines missed
- Less preemptions
- *Jitter* in the fast tasks
- The worst-case response time does not necessarily occur in the synchronous release

RM vs EDF scheduling – initial offsets

Task set

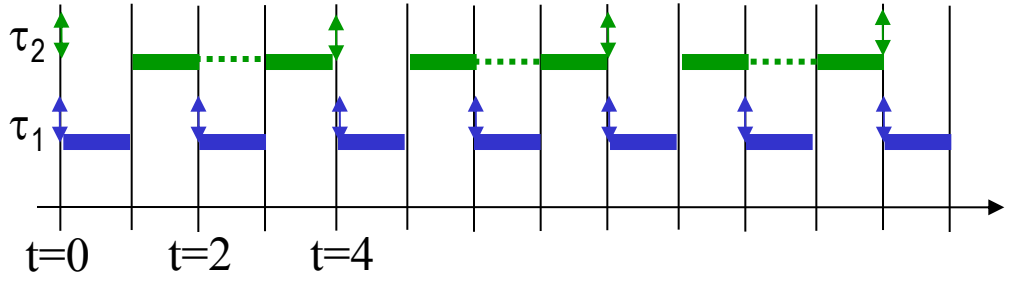
τ_i	T_i	C_i	O_i
1	3	1	0
2	4	1	0
3	6	2.1	2.5



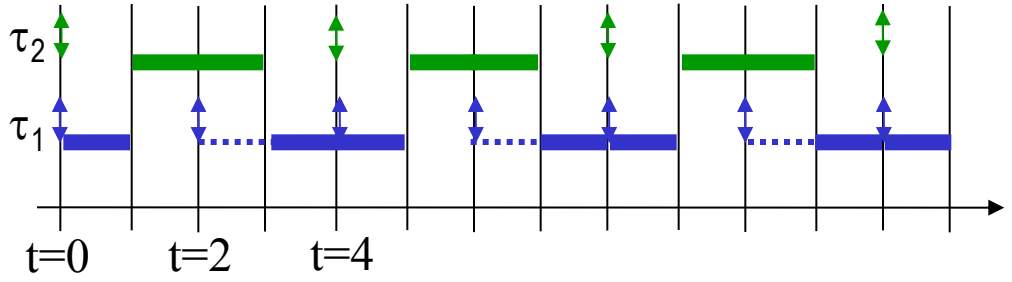
RM vs EDF scheduling – particular cases

$\mathbf{U = 1/2 + 2/4 = 1}$

RM

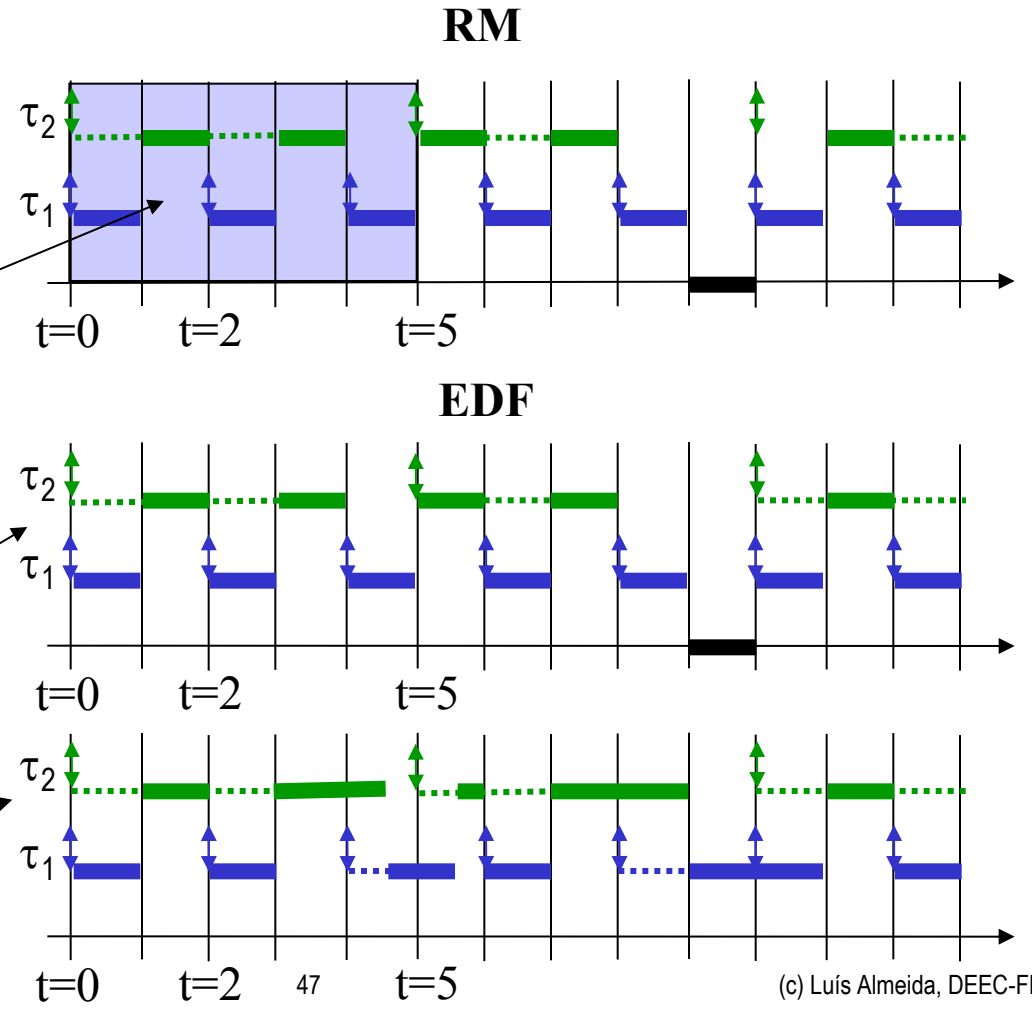


EDF



The actual schedule depends on the criterion to break ties but the deadline are met anyway

RM vs EDF scheduling – particular cases



$U = 1/2 + 2/5 = \mathbf{0.9}$

C_1 or C_2 **cannot** be increased without missing a deadline

C_1 or C_2 **can** be increased without missing deadlines until U=1

$C_2 = \mathbf{2.5} \Rightarrow U=1$

Earliest Deadline First Scheduling

Notion of *fairness*

Fairness in the access to a resource (e.g. CPU)

EDF is intrinsically more fair than RM in the sense that all tasks see their priority raised as they are delayed close to their deadline, independently of their period or other static parameter.

Consequences:

- Facilitates meeting deadlines
- Avoids preemptions when tasks come closer to their deadlines
- Uses the *slack* of the tasks of faster activation but which deadline is later (larger *jitter* in the faster tasks)

CPU demand analysis

For D ≤ T, the longest interval of consecutive CPU load (i.e. without interruption) still correspondes to the situation in which all tasks are released synchronsouly. That interval is called *synchronous busy period* and lasts for *L* time units.

L is computed with the following fixed point iterative method, which returns the first instant, after the synchronous release, in which the CPU becomes free after having executed all tasks intances that were released in the meanwhile

$$\boldsymbol{L(0) = \sum_i C_i}$$

$$\boldsymbol{L(m+1) = \sum_i \left\lceil \frac{L(m)}{T_i} \right\rceil * C_i}$$

CPU demand analysis

Knowing L, we then need to guarantee that the following load condition is met, i.e.

$$h(t) \leq t \quad \forall_{t \in [0\ L[} \Leftrightarrow \textbf{schedulable set (synchronous release)}$$

in which *h(t)* is the load function $h(t) = \sum_{D_i \leq t} \left(1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor\right) * C_i$

Computing *h(t)* for $\forall_{t \in [0,L)}$ is unfeasible... However, it is enough to check the load condition in the points in which the function varies, i.e.

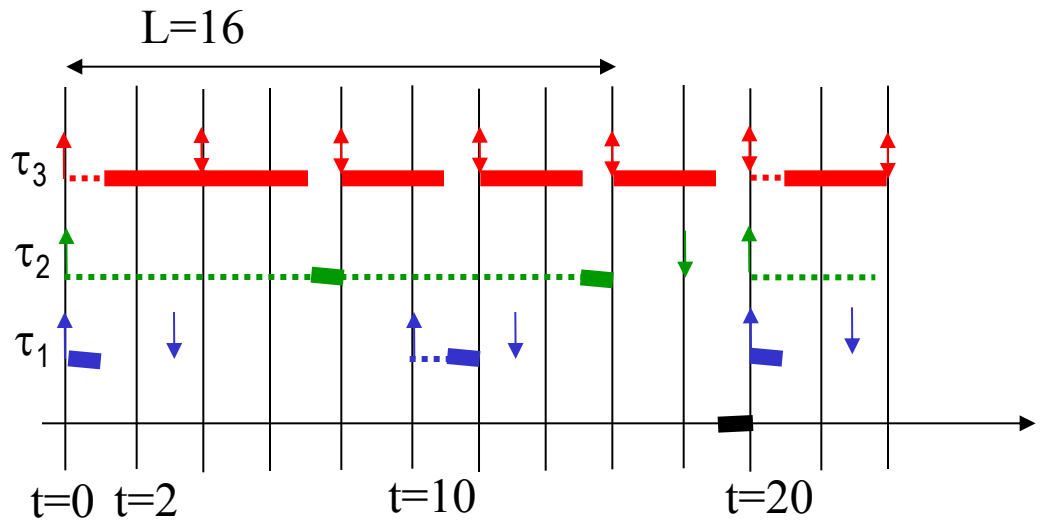
$$S \equiv \bigcup_i S_i, \qquad S_i \equiv \{m * T_i + D_i, m = 0, 1, \ldots\}$$

Note: there are other intervals possibly shorter than L

Earliest Deadline First Scheduling

Task set

τ_i	C_i	D_i	T_i
1	1	3	10
2	2	18	20
3	3	4	4



$\Sigma^{n}_{i=1}(C_i/\min(D_i,T_i)) = 1/3 + 2/18 + 3/4 =$ **1.194 > 1 ⇒ 1 instance per period NOT assured but the schedule is feasible**

However, the CPU demand analysis returns a positive schedulablity results!

Response time analysis

With EDF, the **response time analysis** is more complex than with fixed priorities since we do not know *a priori* which is the task instance that suffers the maximal interference.

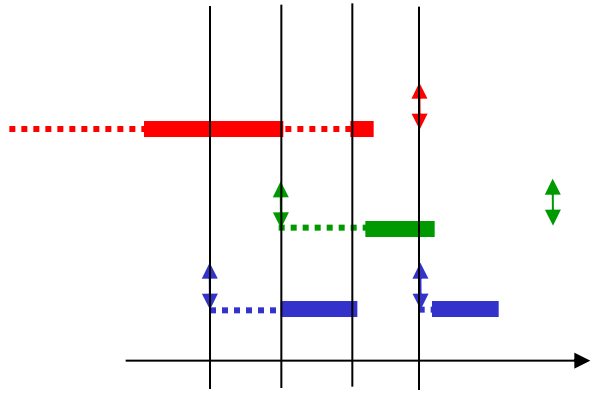
However, it is known that the worst-case response time occurs within the synchronous busy period, but even so it is necessary to check all instances within such period.

A rather simple response-time upper bound can be obtained with the following expression, as long as U ≤ 1

$$\forall_i \; \boldsymbol{Rwc_i \leq T_i * U}$$

Note that this upper bound is rather pessimistic.

Least Slack First Scheduling



Some properties of LSF vs EDF

Optimal (such as EDF)

Slack ↓ ⇒ Priority ↑

Priority of ready tasks increases as the time passes

Priority of the executing task remains constant

(in EDF, the priorities of both ready and executing tasks increase equally as time passes)

Scheduled task may change over and over in an oscillatory behavior

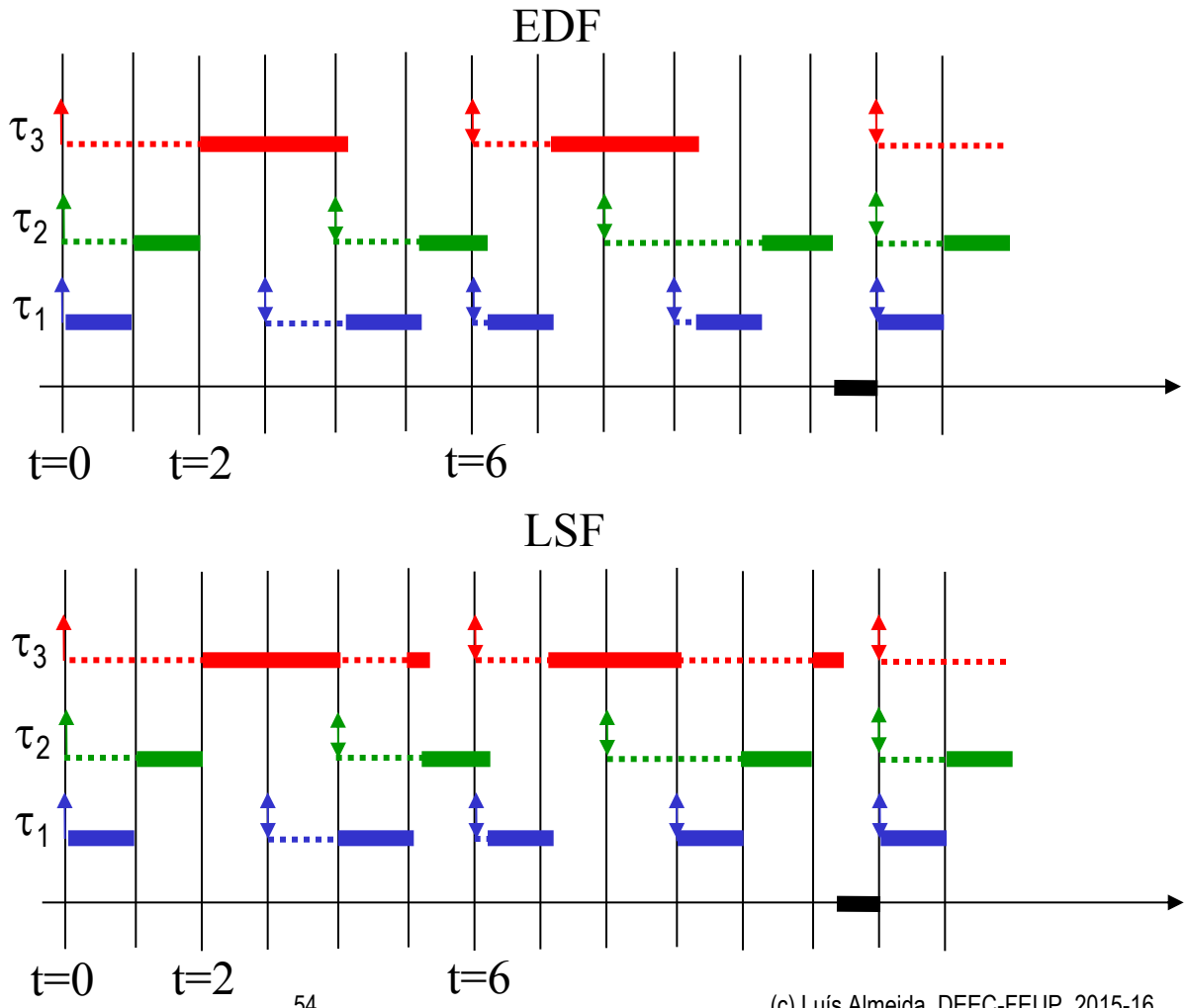
Causes higher number of preemptions than EDF (higher overhead)

No real advantages, in practice, with respect to EDF !

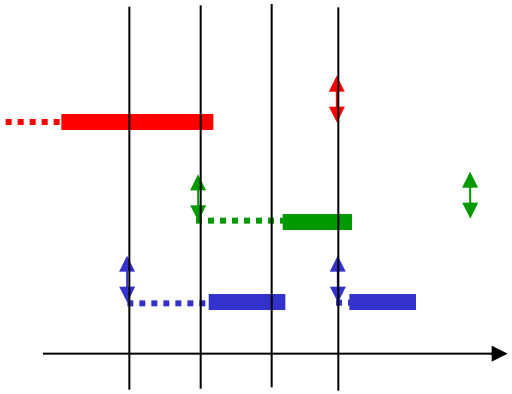
Least Slack First Scheduling – same example

Task set

τ_i	T_i	C_i
1	3	1
2	4	1
3	6	2.1



First Come First Served Scheduling



Some properties of FCFS vs EDF/LLF

Non-optimal (easily causes deadline misses)

"age of an instance" ↑ ⇒ Priority ↑

Priority of ready and executing tasks increases equally for all as time passes (such as with EDF)

When a new task instance arrives (is activated) it is always given a priority lower than that of all others

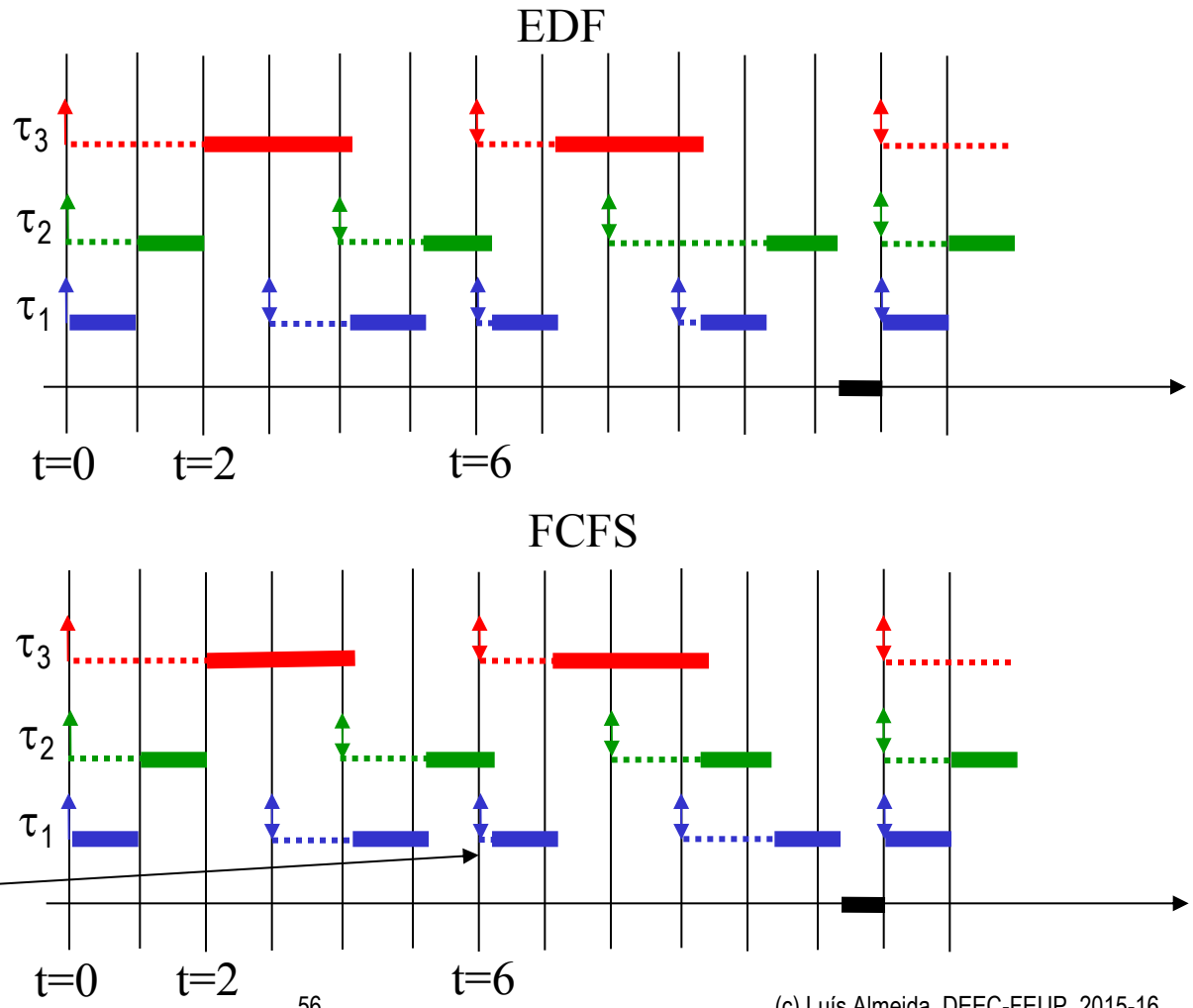
No preemptions (lower overhead – easy implementation)

Poor temporal behavior! But good fairness…

First Come First Served Scheduling – same example

Task set

τ_i	T_i	C_i
1	3	1
2	4	1
3	6	2.1



When age is the same, the criterion to break the tie is determinant!

Wrapping up

- Scheduling ***on-line*** with **dynamic priorities**

- ***EDF - Earliest Deadline First*** criterion: **CPU utilization** upper bound

- **Optimality** of EDF and **comparison with RM**:
  - **Schedulability level, number of preemptions, release *jitter and* response time**

- Other dynamic priorities scheduling criteria:
  - ***LLF (LST) - Least Laxity (Slack) First***
  - ***FCFS - First Come First Served***