Real-Time Systems

Lecture 9

Other real-time scheduling issues

Non-preemptive scheduling
Practical aspects related with the implementation of real-time systems

Last lecture (8)

- Joint execution of **periodic** and **aperiodic** tasks
- **Background execution** of aperiodic tasks
- Notion and characteristics of aperiodic task servers
- **Fixed priority** servers
  - Polling Server - PS
  - Deferrable Server - DS
  - Sporadic Server - SS
- **Dynamic priority** servers
  - Total Bandwidth Server – TBS
  - Constant Bandwidth Server - CBS

Non-preemptive scheduling

Non preemptive scheduling consists in executing the jobs until completion, **without allowing its suspension** for the execution of higher priority jobs

Main characteristics/advantages:

- **Very simple to implement,** as it is not necessary to save the intermediate job's state.
- ***Stack size much lower*** (equal to the stack size of the task with higher requirements)
- No need for any **synchronization protocol** to access **shared resources**, since tasks execute inherently with mutual exclusion

Non-preemptive scheduling

<u>**Main characteristics/disadvantages:**</u>

- **Penalizes** the system **schedulability**, mainly when there are tasks with **long execution times**.
- This penalization may be **excessive** when, simultaneously, the system has tasks with **high activation rates** (short periods).

The penalization can be seen as a **blocking** on the access of a shared resource, **in the case the CPU**. This allows using the schedulability tests previously developed for access to shared resources on preemptive systems.

In this case,

$$B_i = \max_{k \in lp(i)}(C_k)$$

Non-preemptive scheduling

In addition to considering the corresponding blocking time, there are a few **adaptations** that must be made on the response time analysis.

Computation of the Rwc_i with fixed priorities:

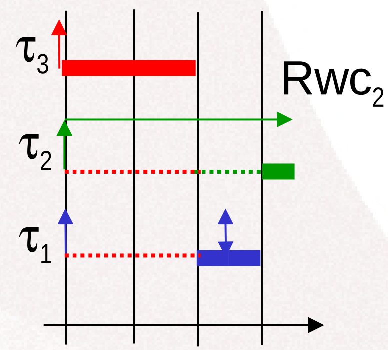
$$\forall i, R_{wc_i} = I_i + C_i$$

The iterative process is carried out only over $\mathbf{I_i}$, since once the task starts executing it will complete without interruption.

$$I_i = B_i + \sum_{k \in hp_i} \left(\left\lfloor \frac{I_i}{T_k} \right\rfloor + 1 \right) * C_k$$

$$I_i(0) = B_i + \sum_{k \in hp_i} C_k$$

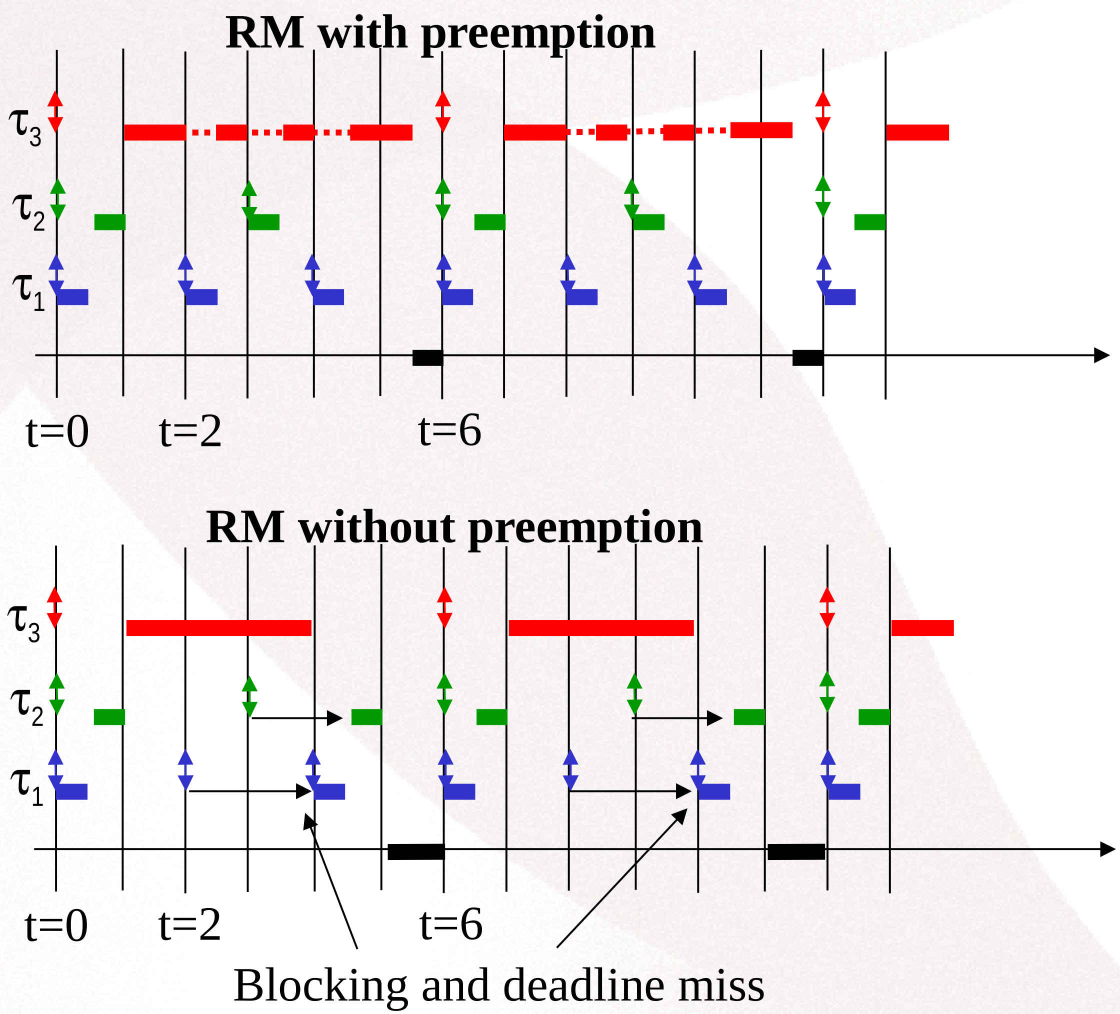
$$I_i(m+1) = B_i + \sum_{k \in hp_i} \left(\left\lfloor \frac{I_i(m)}{T_k} \right\rfloor + 1 \right) * C_k$$



Non-preemptive scheduling

Task properties

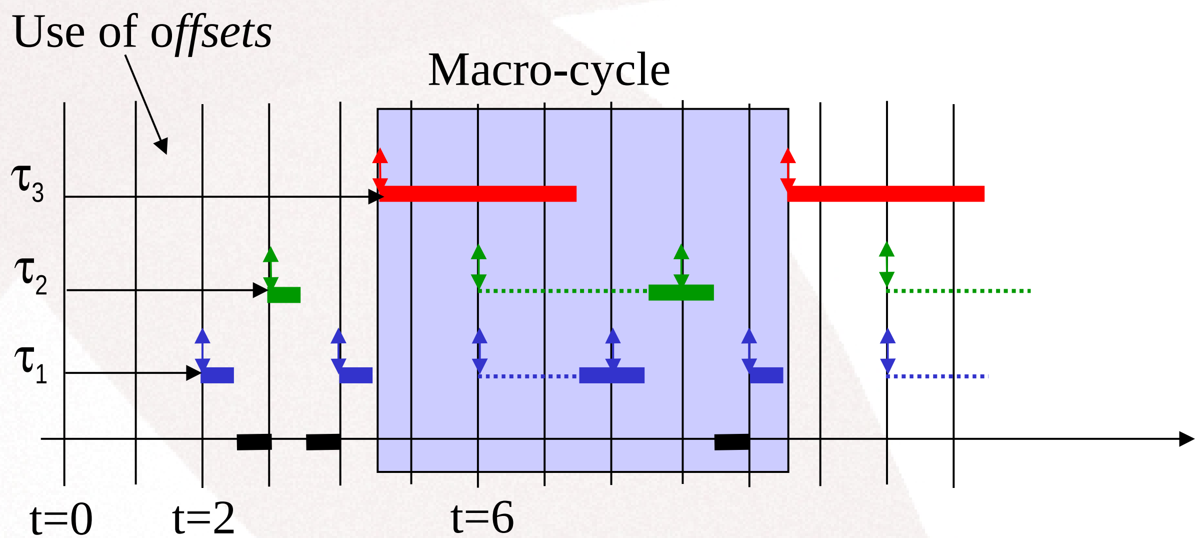
τ_i	T_i	C_i
1	2	0.5
2	3	0.5
3	6	3



Non-preemptive scheduling

Task properties

τ_i	T_i	C_i	O_i
1	2	0.5	**2**
2	3	0.5	**3**
3	6	3	**4.5**



The **use of offsets** may be particularly effective on the non-preemptive scheduling, allowing sometimes turning a system schedulable.

Other issues of practical importance

When developing real applications, there are several aspects that must be taken into account, as they have impact on system schedulability.

Examples are:

- The processing cost of **internal mechanisms** (e.g. *tick handler*)
- The overhead due to **context switching**
- The **task execution** times
- **Interrupt Service Routines**
- Deviations on the **tasks activation instants**

Other issues of practical importance

<u>**Evaluating the computational cost of the system *tick***</u>

- The service to the system tick **uses CPU** time (overhead), which is taken from the tasks' execution.
- It is the highest priority activity on the system and can be **modeled by a periodic task**.
- The respective ***overhead*** (σ) may have a substantial impact on the system, as it is a part of the CPU availability that is not available to the application tasks.

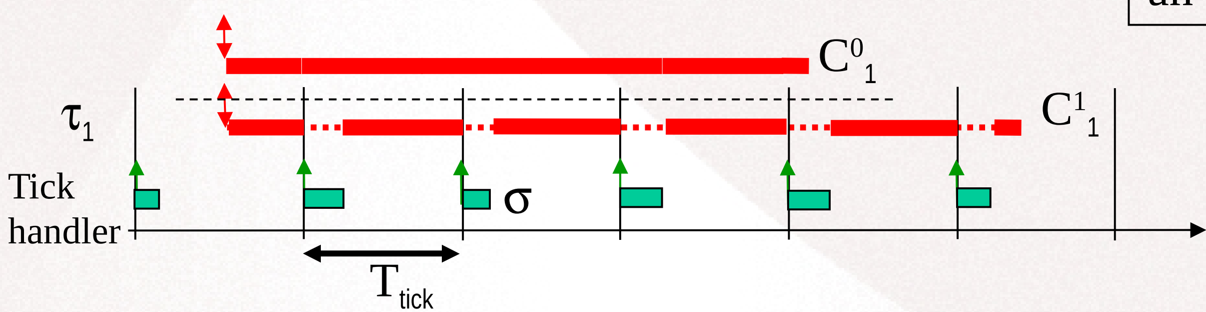
Other issues of practical importance

<u>Evaluating the computational cost of the system *tick*</u>

- Can be measured either directly or via the timed execution of a long function, executed with and without tick interrupts (period T_{tick}) and measuring the difference on the execution times (C^0_1 e C^1_1 respectively).
  In this case,

$$\sigma = \frac{C_1^1 - C_1^0}{\left\lceil \frac{C_1^1}{T_{Tick}} \right\rceil}$$

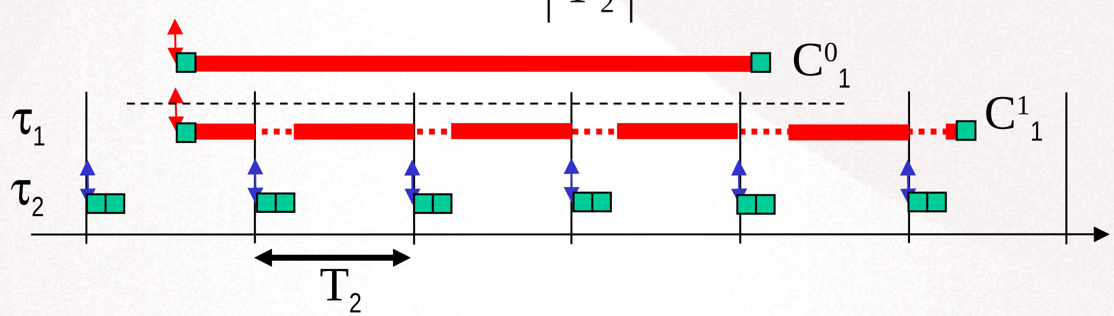
This technique gives an average value for σ !



Other issues of practical importance

Evaluating the cost of context switches

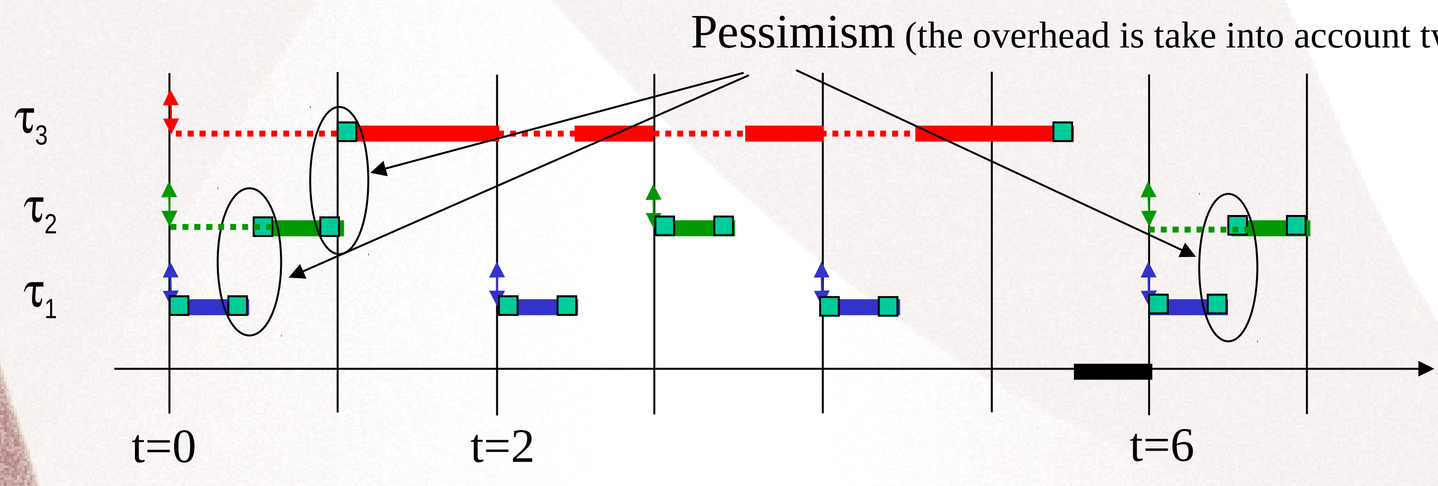
- **Context switches** also require CPU time to save and restore the tasks' context.
- A simple way of measuring this overhead (δ) consists in using two tasks, a long one (τ_1) and another one with higher priority (τ_2), quick (period T_2) and empty (no code). Then it is only required measuring the execution time of the first task alone (C^0_1) and together with the second one (C^1_1).
- In this case, $$\delta = \frac{C_1^1 - C_1^0}{\left\lceil \frac{C_1^1}{T_2} \right\rceil}$$



Other issues of practical importance

Evaluating the cost of context switches (cont.)

- A **simple** (but pessimistic) way of taking into account the overhead due to context switching (δ) consists in **adding** that time to the **execution time** of the tasks. This way it is taken into account not only the context switching overhead due to the task itself as well as the one relative to all context switches that may occur.



Other issues of practical importance

Evaluating the task's execution time

- Can be made via **source code analysis**, to determine the **longest execution path**, according with the input data.
- Then the corresponding object code is analyzed to determine the required number of CPU cycles

- Note that the **execution time** of a task may **vary** from instance to instance, according with the input data or internal state, due to presence of **conditionals and cycles**.

Other issues of practical importance

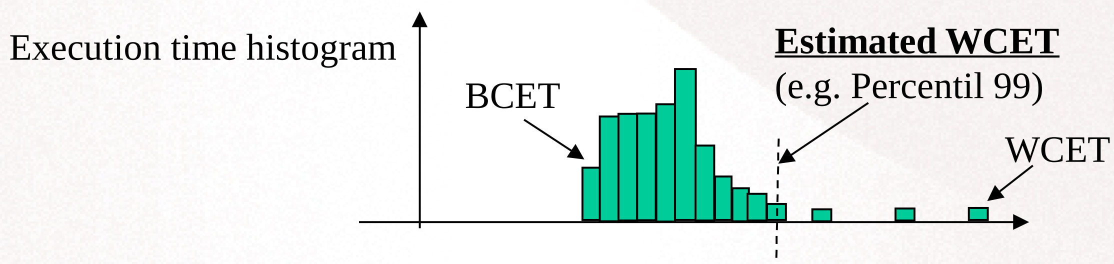
<u>Evaluating the task's execution time</u> (cont.)

- It is also possible **execute the tasks in isolation** and in a controlled fashion, feeding it with adequate **input data** and measuring its execution time on the target platform.
  - This experimental method requires extreme care to make sure that the longest execution paths are reached, a necessary condition to obtain an upper bound on the execution time!
- **Modern complex processors** use features like pipelines and caches (data and/or instructions) that improve dramatically the average execution time but that present an increased gap between the average and the worst-case scenarios.
  - For these cases are used specific analysis that try to reduce the pessimism, e.g. by bounding the maximum number of *cache misses and pipeline flushes,* according with the particular instruction sequences.

Other issues of practical importance

<u>Evaluating the task's execution time</u> (cont.)

- Nowadays there is an growing interest on **stochastic analysis** of the execution times and respective impact in terms of interference.
- The basic idea consists in determining the distribution of the probability of the execution times and use **an estimate** that covers a given target (e.g. 99% of the instances).
- In many cases (mainly when the worst case is infrequent and much worst than the average case) this technique allows reducing drastically the impact of the gap between the average execution time and the WCET (**higher efficiency**)



Other issues of practical importance

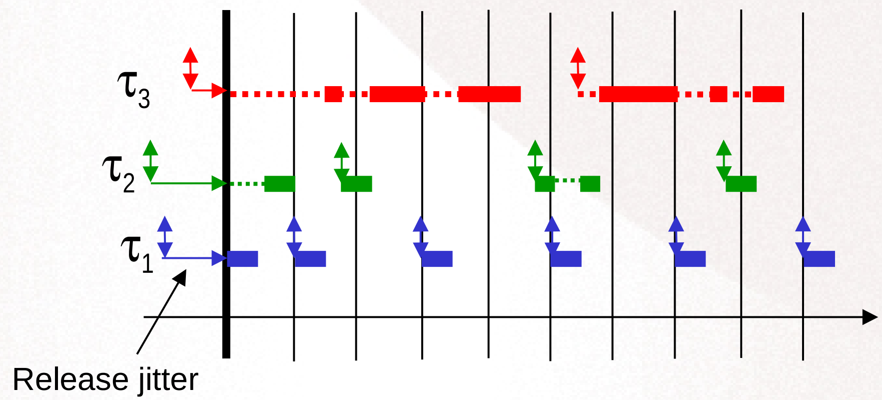
Impact of Interrupt Service Routines

- Generally, the **Interrupt Service Routines (ISR)** execute with an **higher priority level** than all other system tasks.
- Therefore, on a **fixed priority system**, the respective impact can be taken directly into account, by including these **ISR as tasks** in the schedulability analysis.
- In systems with **dynamic priorities** the situation is much more complex (e.g. how to assign deadlines?). In these cases it is usually considered that the time windows in which such ISR execute are not available for normal tasks execution. This can be taken into account in the CPU load analysis.

Other issues of practical importance

Impact of the variations on the tasks' activation instants

- Tasks may suffer **deviations** on the respective activation instants, e.g. when a task is activated by the completion of another one, by an external interrupt or by the reception of a message on a communication port. In such cases the real time lapse between consecutive activations may vary with respect to the predicted values – ***release jitter***
- The existence of release jitter must be taken into account in the schedulability analysis, as in such cases the tasks can **execute during time instants different from the predicted ones.**



Other issues of practical importance

Impact of the variations on the tasks' activation instants

- The presence of *release jitter* can be modeled by the anticipation of the activation instants of the following task instances.

Computing the Rwc_i with release jitter (J_k) for preemptive systems scheduled with fixed priorities

$$\forall i, Rwc_i = I_i + C_i, \text{ with } I_i = \sum_{k \in hp(i)} \left\lceil \frac{Rwc_i + J_k}{T_k} \right\rceil * C_k$$

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

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

Summary of lecture 9

Other real-time scheduling issues

- **Non-preemptive** scheduling
- **Practical aspects** related with the implementation of real-time systems