Real-Time Systems

Lecture 2

Computational Models

Task models with explicit temporal constraints
Logic and temporal control (event -ET and time -TT)

Adapted from the slides developed by Prof. Luís Almeida for the course "Sistemas de Tempo-Real"

Previous lecture (1)

- Refreshing our memory ...
- Notion of **real-time** and **real-time system**
- Antagonism between **real-time** and ***best effort***
- Objectives of the study of RTS – how to **guarantee** the **adequate temporal behavior**
- Aspects to consider: **execution time**, **response-time** and **regularity** of periodic events
- Requirements of RTS: **functional,** **temporal** and **dependability**
- Notion of **real-time database**
- Constraints **soft**, **firm** and **hard**, and **hard real time** vs **soft real time**
- The importance of consider the **worst-case scenario**

Computational models

Transformational model

According to which a program begins and ends, turning data into results or output data.

Reactive model

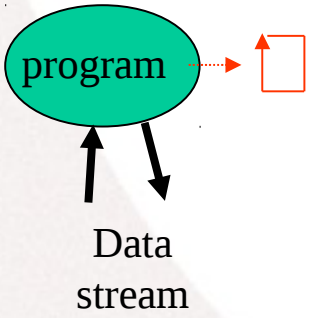
According to which a program may execute indefinitely a sequence of instructions, for example operating on a data stream.

Real-time model

Reactive model in which the program has to keep

synchronized with the input data stream, which thus

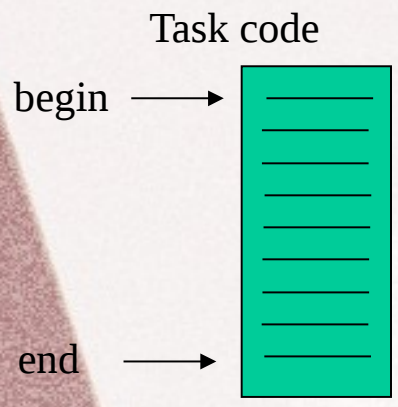
Imposes time constraints to the program.

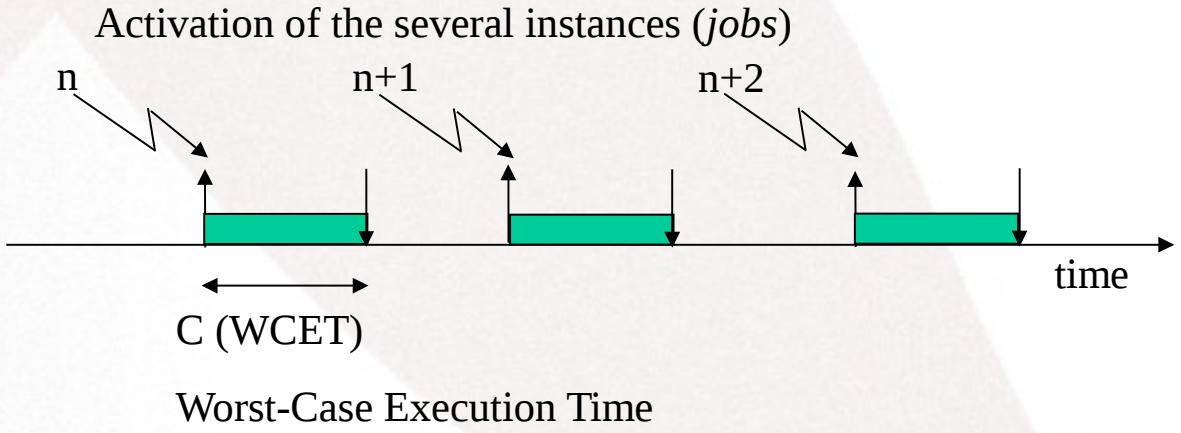


Real-time model

Definition of task (process, activity)

Sequence of activations (instances or jobs), each consisting of a set of instructions that, in the absence of other activities, is performed by the CPU without interruption.



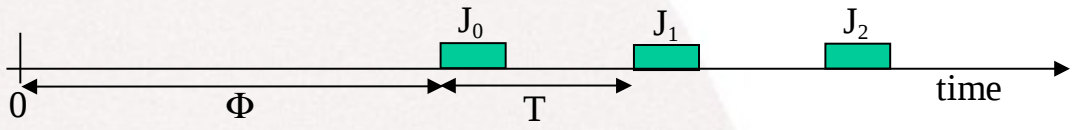


Real-time model

Concerning the periodicity, tasks can be classified as:

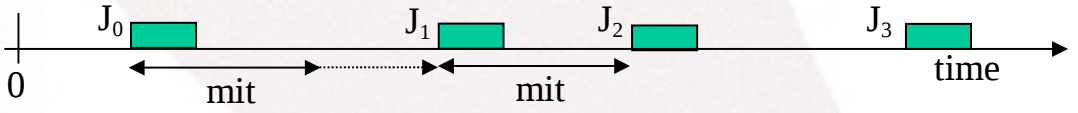
- **periodic**

instance *n* activated at $a_n = n*T + \Phi$



- **sporadic**

Minimum inter-arrival time (*mit*)

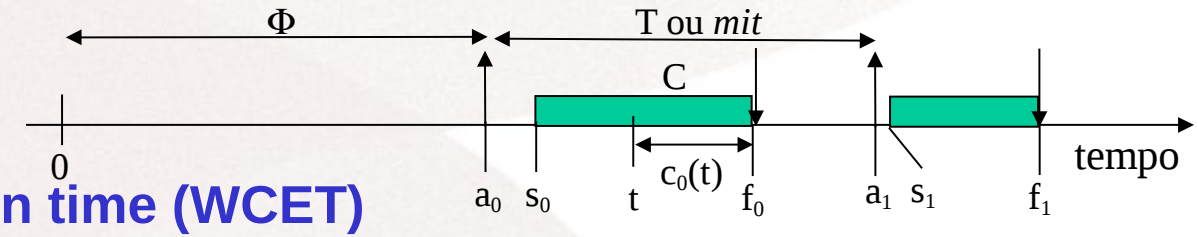


- **aperiodic**

Only characterizable by means of probabilistic arguments

Real-time model

Task characterization



- C – worst-case execution time (WCET)
- T – period (if periodic)
- Φ – relative phase = offset of 1st activation (if periodic)
- mit – *minimum inter-arrival time* (if sporadic)
- a_n – activation instant of the n^{th} instance
- s_n – start time of the n^{th} instance
- f_n – finish time of the n^{th} instance
- $c_n(t)$ – maximum residual execution time of the n^{th} instance at time t

Real-time model

The task constraints can be:

- **Temporal** – limits the temporal moments of termination or generation of certain output events.
- **Precedence** - establish a certain order of execution between tasks.
- **Use of resources** - need to use shared resources (e.g. communication ports, a buffer in shared memory, global variables, system peripherals). May involve the use of atomic operations (whose sequence can not be interrupted)

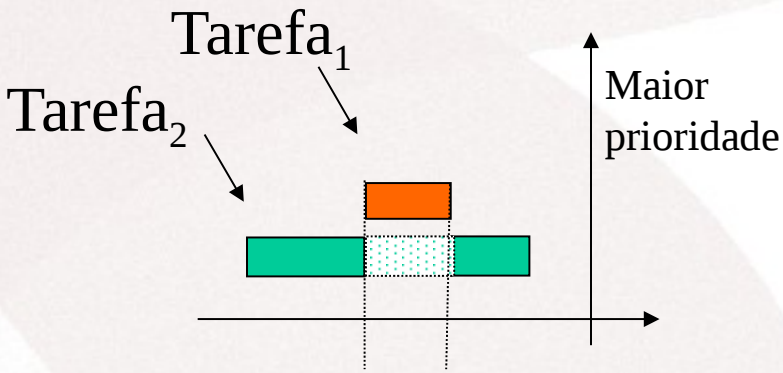
Real-time model

Preemption

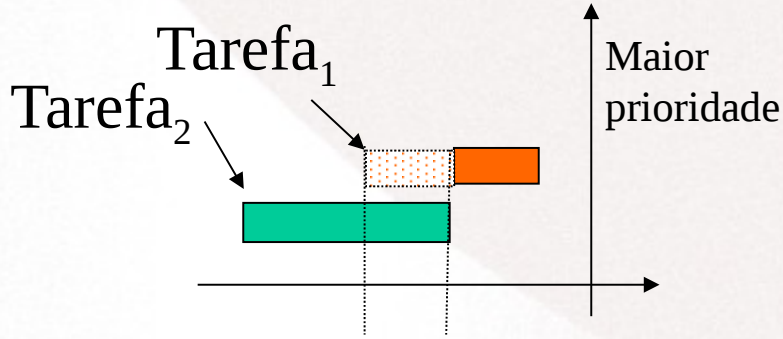
- When a **task** can be interrupted temporarily for execution of other higher priority task, it says that it **admits preemption**.
- When a **system** uses the preemption property of the tasks that it executes, it is designated **preemptive**.
- A set of tasks is said to admit **full preemption** when preemption is admitted by **all tasks at any point** in its execution (independent tasks)
- Note: access to **shared resources** (tasks with dependencies) may **restrict** the the preemptiveness of tasks.

Real-time model

With preemption



Without preemption



Real-time model

The temporal constraints can be of several types:

- **Deadline** – Upper bound on the maximum finish time of a task
- **Window** – Upper and lower bounds on the finish time of a task
- **Synchronization** – Bound on the temporal difference between the generation of two input/output events
- **Distance** – Limit on the delay (distance) between the the termination and the activation of two consecutive instances of a task
  (e.g., change the engine lubricant of a car every 30000Km)

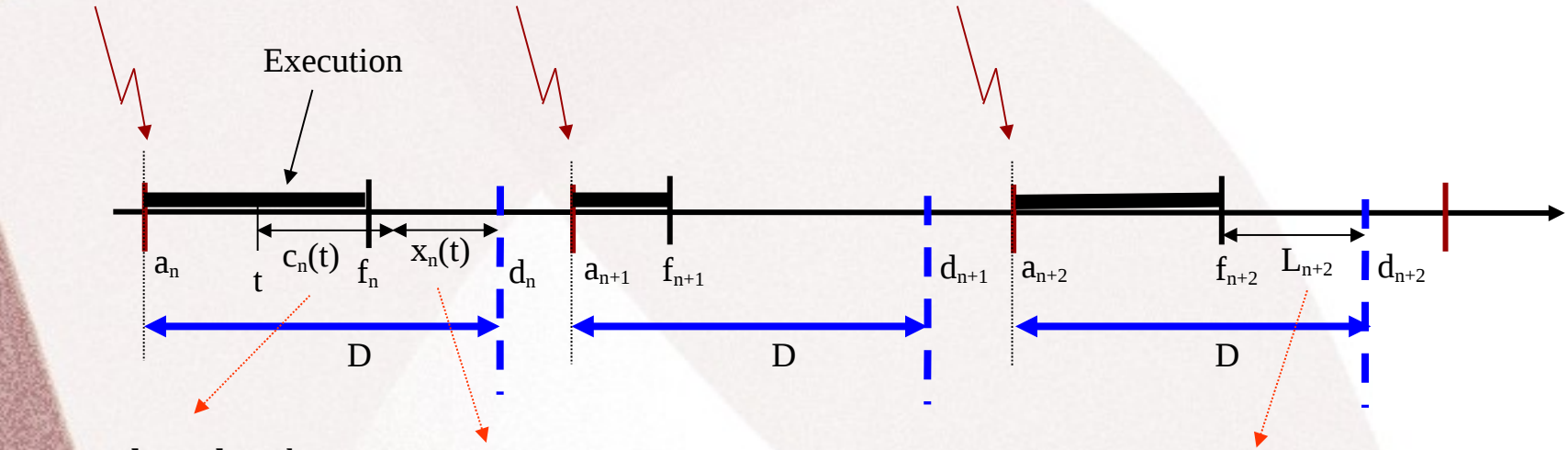
  - ***Deadline*** *is by far the most common temporal constraint*

Real-time model

Deadline

$\forall n, f_n - a_n < D$ (relative)

$\forall n, f_n < d_n$ (absolute)



$c_n(t)$ = **upper bound** on the remaining executon time at time t

$x_n(t)$ = slack

$x_n(t) = d_n - t - c_n(t)$

$L_n = f_n - d_n$ = delay of the n^{th} activation

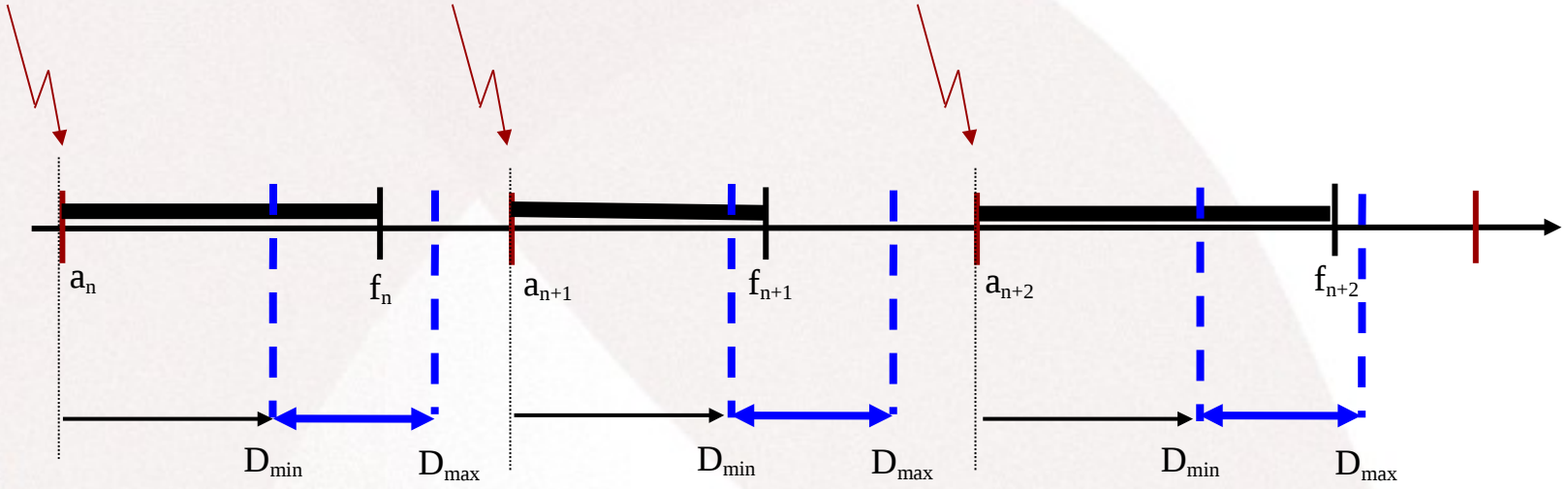
$L_n <= 0$ → on time termination

$L_n > 0$ → late termination

Real-time model

Activation

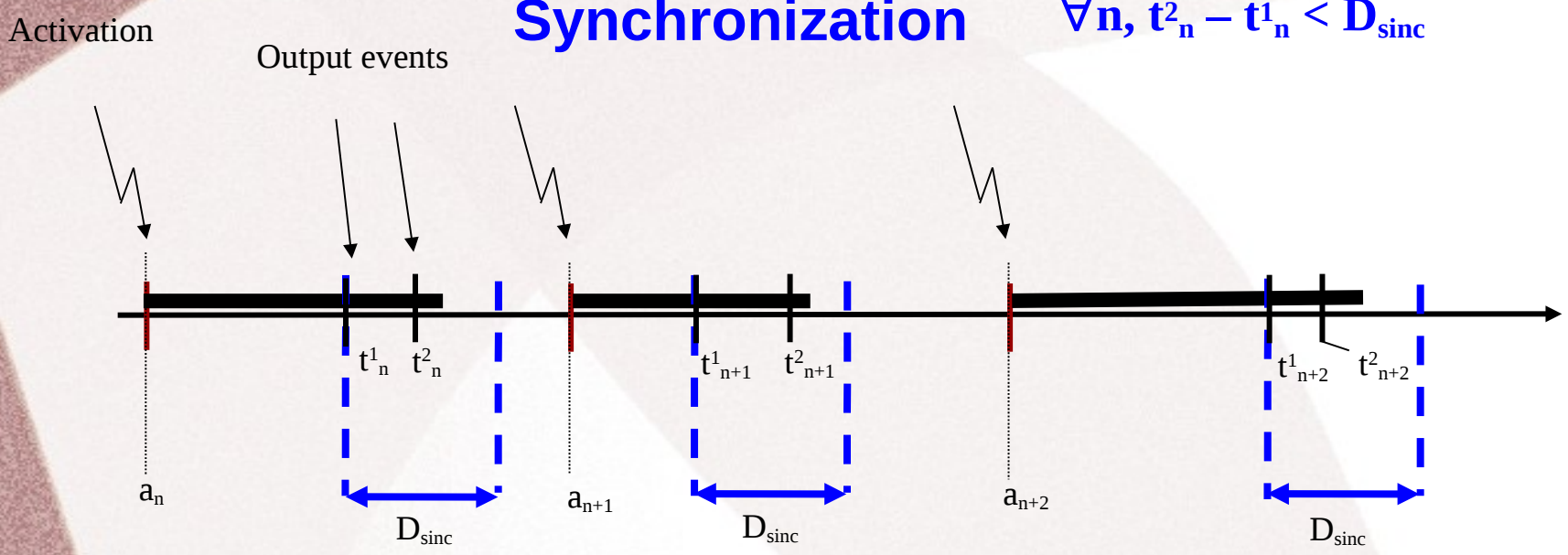
Window $\forall n, D_{min} < f_n - a_n < D_{max}$



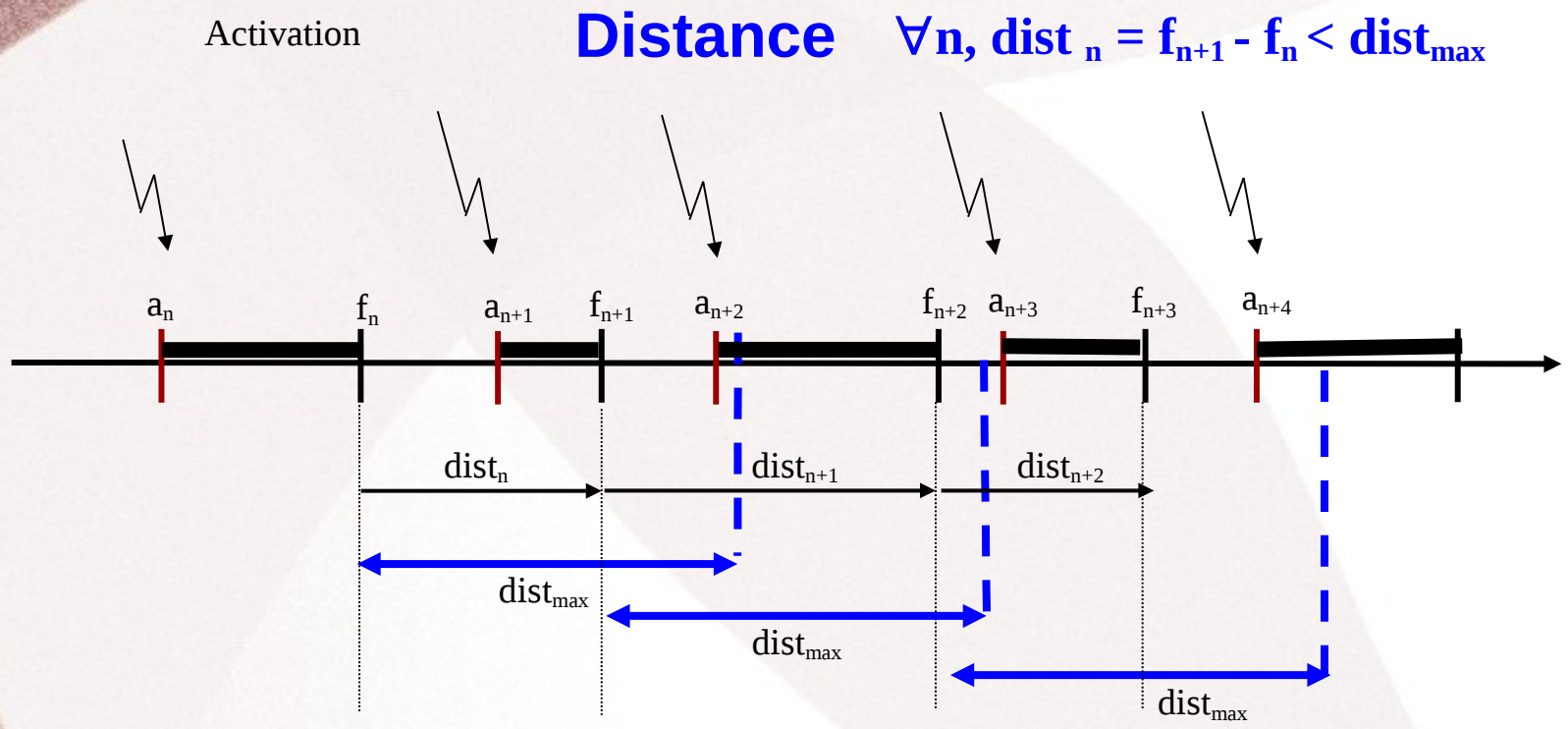
Real-time model

Synchronization

$\forall n,\ t^2_n - t^1_n < D_{sinc}$



Real-time model



Real-time model

Example of task characterization:

- **Periodic:** $\tau_i = \tau_i\ (C_i, \Phi_i, T_i, D_i)$

  $\tau_1 = \tau_1\ (2,5,10,10)$ $\tau_2 = \tau_2\ (3,10,20,20)$

- **Sporadic:** Similar to periodic, but mit_i replaces T_i and Φ_i usually is not used (tough it could be used to specify a minimum delay until the first activation).

  $\tau_i = \tau_i\ (C_i, mit_i, D_i)$

  $\tau_1 = \tau_1\ (2,5,5)$ $\tau_2 = \tau_2\ (3,10,7)$

Implementation of real-time applications

When **software structure** of a real-time application involves only:

- **a main cycle** and, eventually,
- **a reduced number of asynchronous activities**
  (which may be encapsulated in interrupt service routines)

The software is often implemented **directly on the CPU**, without resorting to any intermediate SW structures, like operative systems and kernels.

Implementation of real-time applications

In the case of direct programming on the CPU, the triggering of activities is usually done by interruptions

- **Periodic interruptions** (via timers) for periodic activities. These interrupts are used to count time.

- **Asynchronous interrupts** (communications, external, etc.) for activities triggered by events (changes in system status, e.g., triggering an alarm, reception of data over a communication interface, operator action)

Implementation of real-time applications

However, the use of interrupts:

- **Imposes an additional computational cost**, required to save the state of the CPU whenever an interrupt occurs (i.e., save the CPU context so the application can be resumed latter)

- **Reduces the computational capacity** available to execute the interrupted program. The more interruptions arise the slower the program runs. Ultimately, the program execution can be completely blocked.

Implementation of real-time applications

The use of interrupts may be made **with** or **without** ***nesting***

- **With nesting** – interrupt service routines (ISR) can be interrupted by higher priority ISRs
  - Harder to bound the stack size
  - Better response time to high priority ISRs
- **Without nesting** – once started, each ISR executes until its end, without being interrupted. All other pending interrupts are delayed.
  - Opposed characteristics with respect to the previous case
  - Note that high-priority ISRs are blocked by lower-priority ones

Implementation of real-time applications

On the other hand, when the application involves **multiple activities**, asynchronous or not:

- the respective programming is **facilitated** by the use of **Operating Systems** or **multi-task executives** which directly support multiple tasks that can run independently or sharing of system resources.

Each activity is encapsulated in a task.

Multi-task executives

The **application programming** using the support of SW structures like **Executives** or **Operating Systems** enables:

- **Higher level of abstraction**
- **Reduced dependence on the HW**
- **Easier maintenance of SW**

Notes:

- Even in these cases, the **triggering of tasks is done by interrupts,** via a periodic interrupt that provides a periodic measurement of time for the OS or Executive (tick).
- It is also possible to use **asynchronous interrupts**, although they are usually **encapsulated in device drivers**.

Multi-task executives

The processing associated with a given activity can be performed:

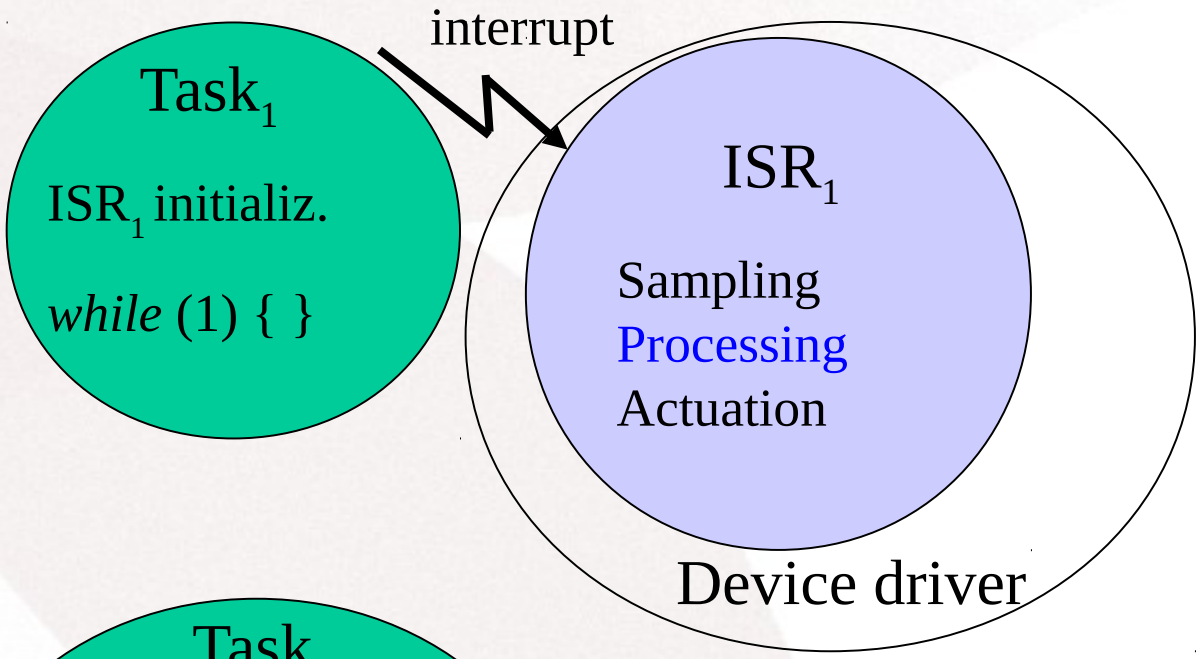
- **At the level of ISR**
  - Does not take advantage of certain benefits of OS or Executive (low-level programming - very dependent on HW)
  - High reactivity to external events (micro-seconds ...)
  - Large interference suffered by tasks
  - Limited # of ISRs
- **At the level of a task**
  - Takes advantage of the OS or Executive (high-level programming, less dependent on HW, better maintenance)
  - Lower reactivity to external events (higher overhead)
  - Lower disturbance on tasks

Multi-task executives

Processing:

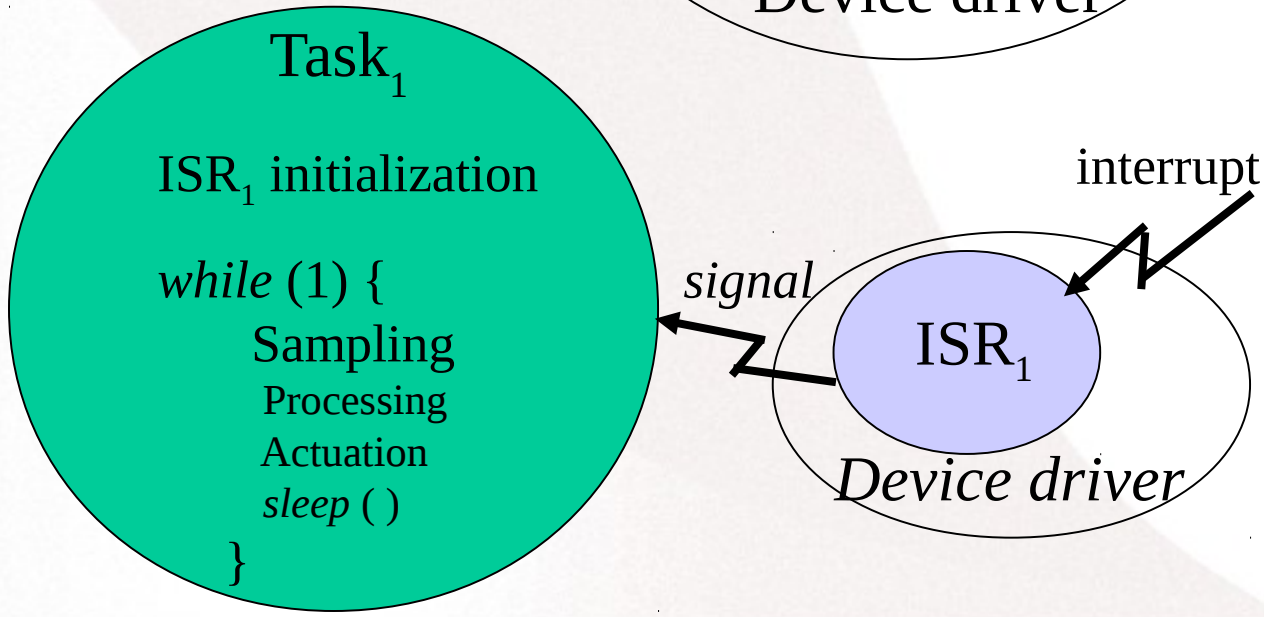
- **At ISR**

  (non standard)



- **At task level**

  (standard)



Multi-task executives

Classification of OSes and executives regarding **temporal guarantees**

- **Non-Real Time (time-sharing)**
  (eg, Unix, Linux, Windows NT, follow the transformational model)
  - Can not bound the response time to an event (e.g., due to swapping, blocking access to peripherals, scheduling that favors equitable distribution of CPU)
- **Soft Real-Time**
  (eg OS9, some services Linux / Windows)
  - Use techniques from real-time (excluding virtual memory, fast IPC mechanisms and reduced blockages, short system calls) but offer no temporal guarantees (best-effort type)
- **Hard Real-Time**
  (e.g. SHaRK, Xenomai/RTLinux, QNX, VxWorks, FreeRTOS, ...)
  - Provide temporal guarantees

Logic and temporal control

- **Logic control**
  - Control of the program flow, i.e., the effective sequence of operations to be performed (e.g. described by a flowchart) - **crucial to determine C (WCET)**

- **Temporal control**
  - Control of the **instants of execution of program operations** (e.g., triggering of activities, monitoring of the compliance with the time constraints, ...)

Temporal control

Triggering of activities

- **By time (*time-triggered*)**
  - The execution of an activity (function) is triggered via a control signal based on the progression of time (e.g., through a periodic interrupt).
  -
- **By events (*event-triggered*)**
  - The execution of activities (functions) is triggered through an asynchronous control signal based on the change of system state (e.g., through an external interrupt).

Temporal control

Systems triggered by the progression of time

time-triggered (TT) systems

- Typical in **control applications** (sampling of analog variables).
- There is a **common time reference** (allows establishing phase relations)
- CPU **utilization** is **constant**, even when there are no changes in the system state.

Worst case situation is well-defined

Temporal control

Systems controlled by the occurrence of events on the environment

event-triggered (ET) systems

- Typical of sporadic conditions monitoring on the system state (e.g., alarm verification or asynchronous requests).
- Utilization rate of the the computing system (eg CPU) is variable, depending on the frequency of occurrence of events.
  - Ill-defined worst case situation. Implies:
    - use of probability arguments
    - limitation on the maximum rate of events

Temporal control

- Example
  - For the following task sets compute the maximum delay that each task may suffer.:
    - TT $\{\tau_i = \tau_i\ (C_i=1, \Phi_i=i, T_i=5, D_i=T_i\ \ i=1..5)\}$
    - ET $\{\tau_i = \tau_i\ (C_i=1, (\Phi_i=0), mit_i=5, D_i=mit_i\ \ i=1..5)\}$
  - Compute the average and maximum CPU utilization rate for both cases. Admit that, on average, ET tasks are activated every 100 time units.

    Note: the CPU utilization is given by:

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i}$$

Summary of lecture 2

- Computational models (**real-time model**)
- Real-time tasks: periodic, sporadic and aperiodic
- Temporal constraints of types: **deadline**, window, synchronization and distance
- Implementation of tasks using ***multitasking kernels***
- **Logic** and **temporal** control
- **Event-triggered** and **time-triggered** tasks