

# Real-Time Task Model

Giuseppe Lipari

<http://www.univ-lille.fr/~lipari>

CRISTAL - Université de Lille

September 12, 2022



# Outline

- 1 Introduction to Real-Time Systems
- 2 Task Model
- 3 Computation times and workload
- 4 Scheduling
- 5 Other parameters

# Outline

- 1 Introduction to Real-Time Systems
- 2 Task Model
- 3 Computation times and workload
- 4 Scheduling
- 5 Other parameters

# Definition of embedded system

From “Wikipedia”:

*An embedded system is a special-purpose computer system designed to perform **one or a few dedicated functions**, sometimes with real-time computing constraints. It is usually embedded as part of a complete device including hardware and mechanical parts. In contrast, a general-purpose computer, such as a personal computer, can do many different tasks depending on programming.*

*Since embedded systems are dedicated to specific tasks, design engineers can optimize it, reducing the size and cost of the product, or increasing the reliability and performance.*

# Characteristics

- Embedded - **Dedicated**
- Interaction with physical processes
  - sensors, actuators ← timing constraints (latency, jitters)
- Reactivity
  - Need to operate at the same speed as the environment
- Functional and non functional properties
  - real-time, fault recovery, power, security, robustness,
- Heterogeneity
  - hardware/software tradeoffs
- Concurrency
  - Interaction with several processes at the same time
- Resource constraints
  - Cost, energy, space, etc.

# Characteristics

- There are many differences between traditional software development, and embedded system development
- Traditional software development:
  - It focuses on functionality
  - it hides the hardware completely
  - Example: Development of a Web application
  - It is done **best effort**, the timing is not so important
- Embedded system development:
  - It must consider the choice of the hardware
  - It must consider the interaction with the external environment (real-time constraints)
  - It must also consider other constraints (fault-tolerance, energy, etc.)

# Real-Time systems

*In real-time systems, the correct behaviour of a system depends, not only on the values of results that are produced, but also on the time at which they are produced.<sup>1</sup>*

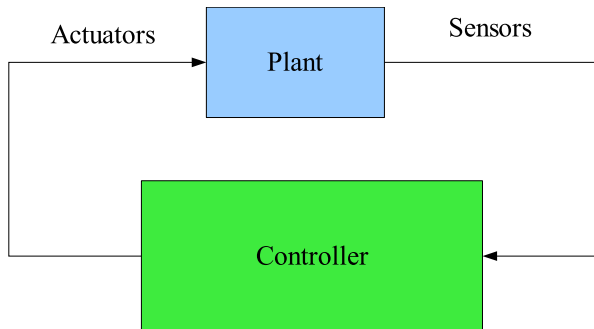
- **Predictable system**: we want to know a-priori if the system will satisfies the timing constraints.
- A real-time system is not a system that goes **fast**, rather it is a system that satisfies its timing constraints

---

<sup>1</sup>John Stankovic, Misconceptions about real-time computing, IEEE Computer, October 1988

# Control Systems

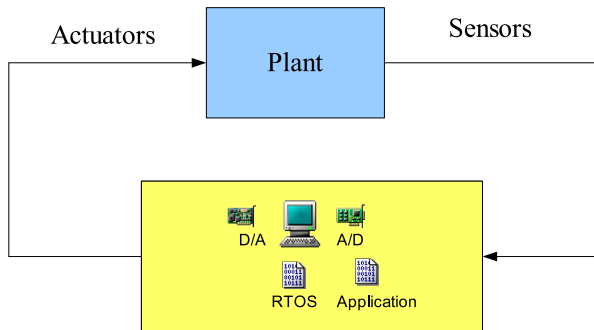
A generic control system has a well-know structure





# Embedded control system

In the case of an **Embedded Control System (ECS)**, the controller is implemented as a computer that executes control software



Advantages: portability, easy to extend, computation power

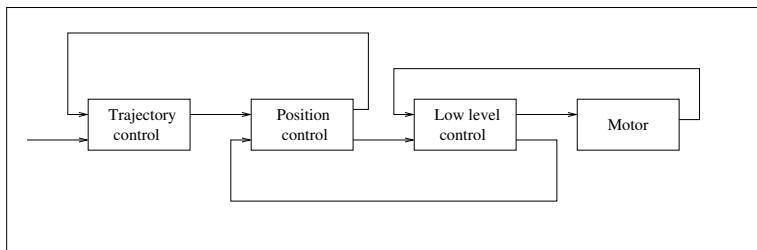
# ECS design and development

The system must respond to external events (**reactive system**) within a certain limit

- The control system is digital: sensors are sampled, the plant is controlled through actuators
- In a system with sampling time  $T$ , it is necessary to
  - sample sensors
  - compute the control function
  - send actuator commandsevery  $T$  units of time
- In this simple case, we have one single sampling period
- Cyclic structure

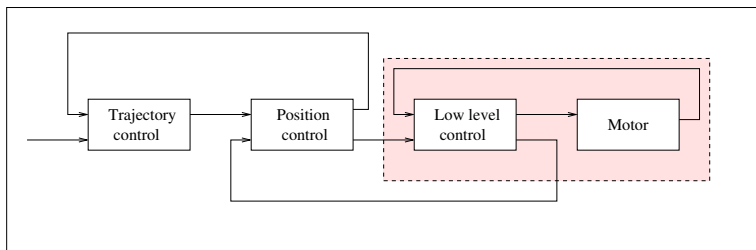
# Multi-level control system

- **Example** Consider a 2 level control system for a mobile robot
  - First level (motor control): we need high sampling frequencies
  - Second level (path following): we need to control the direction of the robot in order to reach a certain place. Lower frequencies are acceptable



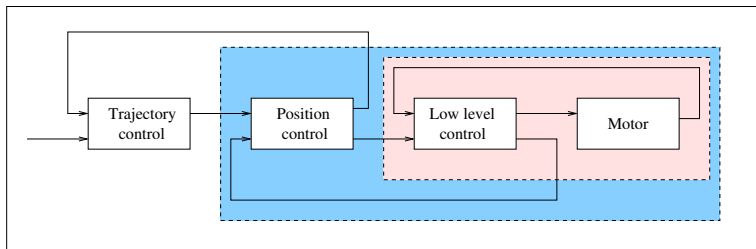
# Multi-level control system

- **Example** Consider a 2 level control system for a mobile robot
  - First level (motor control): we need high sampling frequencies
  - Second level (path following): we need to control the direction of the robot in order to reach a certain place. Lower frequencies are acceptable



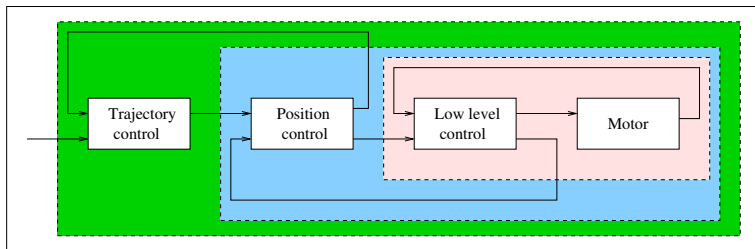
# Multi-level control system

- **Example** Consider a 2 level control system for a mobile robot
  - First level (motor control): we need high sampling frequencies
  - Second level (path following): we need to control the direction of the robot in order to reach a certain place. Lower frequencies are acceptable



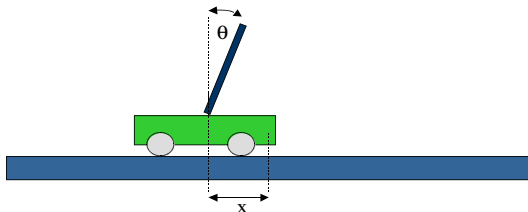
# Multi-level control system

- **Example** Consider a 2 level control system for a mobile robot
  - First level (motor control): we need high sampling frequencies
  - Second level (path following): we need to control the direction of the robot in order to reach a certain place. Lower frequencies are acceptable



# Multi-rate control systems

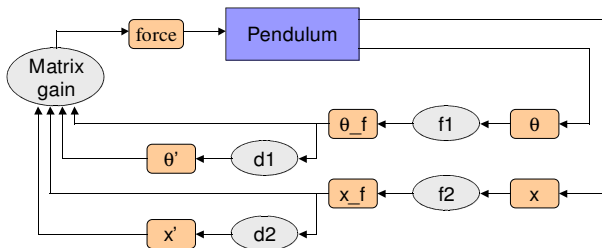
- Example: Consider an inverted pendulum



- Sensors:
  - angle  $\theta$ : with a potentiometer
  - distance  $x$  from the reference point: with a video camera
- actuator: with a motor, we impose an impulse  $F$  to the left or to the right

# Structure of the application “Inverted pendulum”

- Here is the structure of the application



- We have two cycles

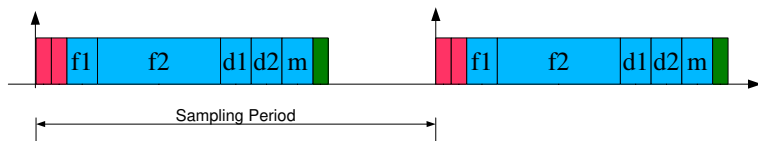


- The cycle for reading the position is at 40ms (the camera works at 25 fps), cannot go faster than that
- The angle position is read through a simple A/D converter, and can be very fast (e.g. 1 msec)



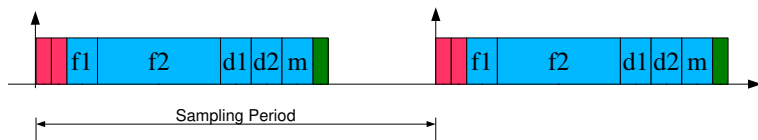
# Development

- Let see how this can be realised in practice
- First try:
  - A single periodic cycle that reads the sensors, compute the commands, updates the output



# Development

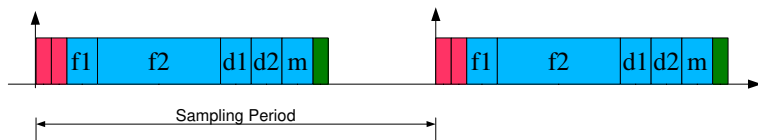
- Let see how this can be realised in practice
- First try:
  - A single periodic cycle that reads the sensors, compute the commands, updates the output



- Does not work!
  - Computing the position (image processing) takes 20 msec
  - Reading the angle every 20 msec, the system become unstable

# Development

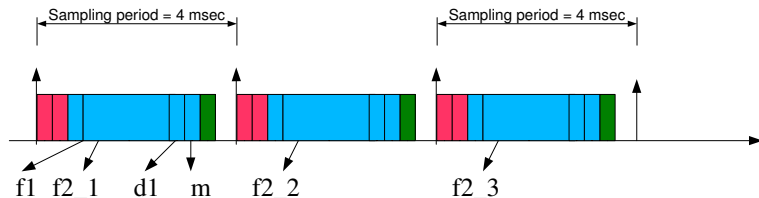
- Let see how this can be realised in practice
- First try:
  - A single periodic cycle that reads the sensors, compute the commands, updates the output



- Does not work!
  - Computing the position (image processing) takes 20 msec
  - Reading the angle every 20 msec, the system become unstable
- So we will sample the position at 4 msec, and the camera every 40msec

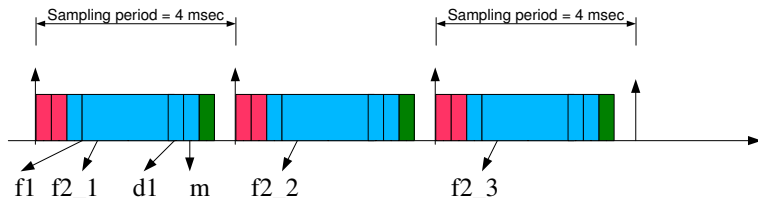
# Table based scheduling

- We split the function  $f_2$  in smaller functions  $f_{21}, f_{22}, \dots$
- Each one of the sub-functions takes not more that 2 msec.



# Table based scheduling

- We split the function  $f_2$  in smaller functions  $f_{21}, f_{22}, \dots$
- Each one of the sub-functions takes not more that 2 msec.




- Problem: not easy to split functions like this
- The solution is not easily portable or extensible
- Until today, in the embedded system market, many companies develop in this way

# Using a RTOS

We can take advantage of a real-time operating system that can execute multiple processes in parallel (or concurrently)

```
void * mytask1(void *) {  
    while (1) {  
        f1();  
        d1();  
        matrix();  
        actuate();  
        task_endcycle();  
    }  
}
```

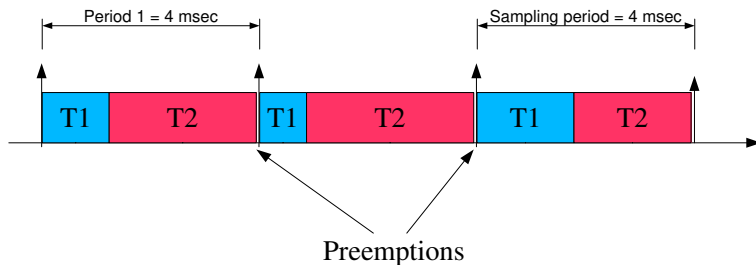


```
void * myTask2(void *) {  
    while (1) {  
        f2();  
        d2();  
        task_endcycle();  
    }  
}
```

- We associate myTask1() to a high priority
- We associate myTask2() to a low priority

# Schedule

- The order of execution is the following:



# Does it work?

- How do we know that everything works?
  - We need a **scheduling analysis**
  - We must analyse the system (tasks, computation times, etc.) to understand if every task will always complete before its deadline.
- What is the best way to assign priorities?
- What if we have more than one processor?
- What if I want to add a new functionality?



# Outline

- 1 Introduction to Real-Time Systems
- 2 Task Model**
- 3 Computation times and workload
- 4 Scheduling
- 5 Other parameters

# Task model

A task can be:

- *periodic*: has a regular structure, consisting of an infinite cycle, in which it executes a computation and then suspends itself waiting for the next periodic activation. An example of pthread library code for a periodic task is the following:

```
void * PeriodicTask(void *arg)
{
    <initialization>;
    <start periodic timer, period = T>;
    while (cond) {
        <read sensors>;
        <update outputs>;
        <update state variables>;
        <wait next activation>;
    }
}
```

# Model of a periodic task

From a mathematical point of view, a periodic task  $\tau_i = (C_i, D_i, T_i)$  consists of a (infinite) sequence of jobs  $J_{i,k} = (a_{i,k}, c_{i,k}, d_{i,k})$ ,  $k = 0, 1, 2, \dots$ , with

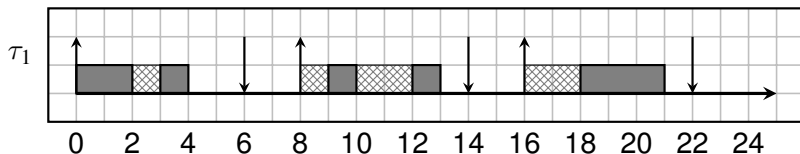
$$\begin{aligned}a_{i,0} &= 0 \\ \forall k > 0 \quad a_{i,k} &= a_{i,k-1} + T_i \\ \forall k \geq 0 \quad d_{i,k} &= a_{i,k} + D_i \\ C_i &= \max\{k \geq 0 \mid c_{i,k}\}\end{aligned}$$

- $T_i$  is the task's period;
- $D_i$  is the task's relative deadline;
- $C_i$  is the task's worst-case execution time (WCET);



# Graphical representation

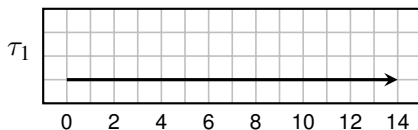
In this course, the tasks will be graphically represented with a GANTT chart. In the following example, we graphically show periodic task  $\tau_1 = (3, 6, 8)$ .



Notice that, while job  $J_{i,0}$  and  $J_{i,3}$  execute for 3 units of time (WCET), job  $J_{i,2}$  executes for only 2 units of time.

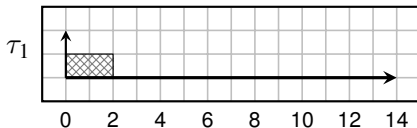
# Task states

- From the point of view of the operating system, a task can be in one of the following states:
  - WAITING: the task is waiting for the activation of the next periodic instance (job)
  - READY: one job has been activated, and it is ready to execute
  - EXECUTING: when the task is actually executing on one processor
  - BLOCKED: waiting for some condition, different from the periodic activation
- READY
- EXECUTING
- WAITING



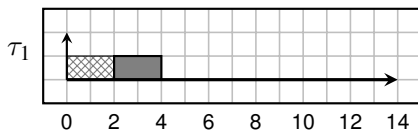
# Task states

- From the point of view of the operating system, a task can be in one of the following states:
  - WAITING: the task is waiting for the activation of the next periodic instance (job)
  - READY: one job has been activated, and it is ready to execute
  - EXECUTING: when the task is actually executing on one processor
  - BLOCKED: waiting for some condition, different from the periodic activation
- **READY**
- EXECUTING
- WAITING



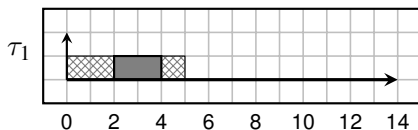
# Task states

- From the point of view of the operating system, a task can be in one of the following states:
  - WAITING: the task is waiting for the activation of the next periodic instance (job)
  - READY: one job has been activated, and it is ready to execute
  - EXECUTING: when the task is actually executing on one processor
  - BLOCKED: waiting for some condition, different from the periodic activation
- READY
- **EXECUTING**
- WAITING



# Task states

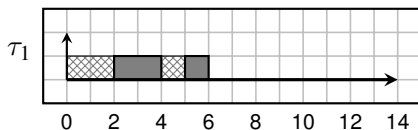
- From the point of view of the operating system, a task can be in one of the following states:
  - WAITING: the task is waiting for the activation of the next periodic instance (job)
  - READY: one job has been activated, and it is ready to execute
  - EXECUTING: when the task is actually executing on one processor
  - BLOCKED: waiting for some condition, different from the periodic activation
- **READY**
- EXECUTING
- WAITING





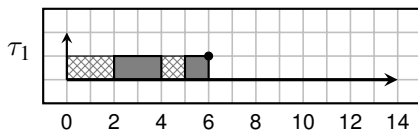
# Task states

- From the point of view of the operating system, a task can be in one of the following states:
  - WAITING: the task is waiting for the activation of the next periodic instance (job)
  - READY: one job has been activated, and it is ready to execute
  - EXECUTING: when the task is actually executing on one processor
  - BLOCKED: waiting for some condition, different from the periodic activation
- READY
- **EXECUTING**
- WAITING



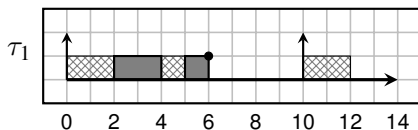
# Task states

- From the point of view of the operating system, a task can be in one of the following states:
  - WAITING: the task is waiting for the activation of the next periodic instance (job)
  - READY: one job has been activated, and it is ready to execute
  - EXECUTING: when the task is actually executing on one processor
  - BLOCKED: waiting for some condition, different from the periodic activation
- READY
- EXECUTING
- WAITING



# Task states

- From the point of view of the operating system, a task can be in one of the following states:
  - WAITING: the task is waiting for the activation of the next periodic instance (job)
  - READY: one job has been activated, and it is ready to execute
  - EXECUTING: when the task is actually executing on one processor
  - BLOCKED: waiting for some condition, different from the periodic activation
- **READY**
- EXECUTING
- WAITING



# Sporadic tasks

- *sporadic* tasks are very similar to periodic tasks. But, instead of suspending themselves on timer events, they wait for other events, which are not periodic in general (for example, the arrival of a packet from the network).
- however, for sporadic tasks, it is possible to define a *minimum interarrival time* between two occurrences of the event.

```
void * SporadicTask(void *)  
{  
    <initialization>;  
    while (cond) {  
        <computation>;  
        <wait event>;  
    }  
}
```



# Mathematical model of a sporadic task

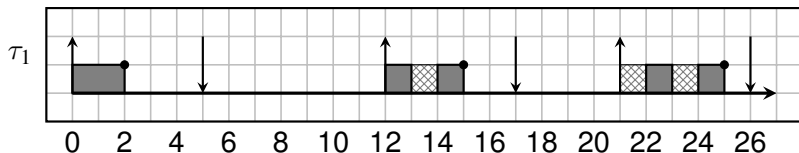
Similar to a periodic task: sporadic task  $\tau_i = (C_i, D_i, T_i)$  consists of a (infinite) sequence of jobs  $J_{i,k}$ ,  $k = 0, 1, 2, \dots$ , with

$$\begin{aligned}\forall k > 0 \quad a_{i,k} &\geq a_{i,k-1} + T_i \\ \forall k \geq 0 \quad d_{i,k} &= a_{i,k} + D_i \\ C_i &= \max\{k \geq 0 \mid c_{i,k}\}\end{aligned}$$

- $T_i$  is the task's minimum interarrival time (MIT);
- $D_i$  is the task's relative deadline;
- $C_i$  is the task's worst-case execution time (WCET).

# Graphical representation

In the following example, we show sporadic task  $\tau_1 = (2, 5, 9)$ .



Notice that

$$a_{1,1} = 12 > a_{1,0} + T_1 = 9.$$

$$a_{1,2} = 21 = a_{1,1} + T_1 = 9$$



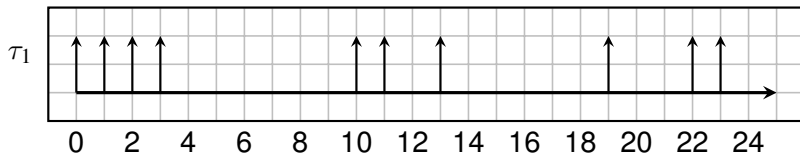
# Aperiodic Tasks



For them, it is not possible to set a minimum separation time between two consecutive jobs. Also, they do not have a particular structure.

With this kind of task we can model:

- Tasks that respond to events that occur rarely. Example: a mode change.
- Tasks that respond to events that happen with an irregular structure. Example: bursts of packets arriving from the network.



# Hyperperiod

- A task set  $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$  is periodic if it consists of periodic tasks only.
- The *hyperperiod* of a periodic task set is the least common multiple (lcm) of the task's periods;

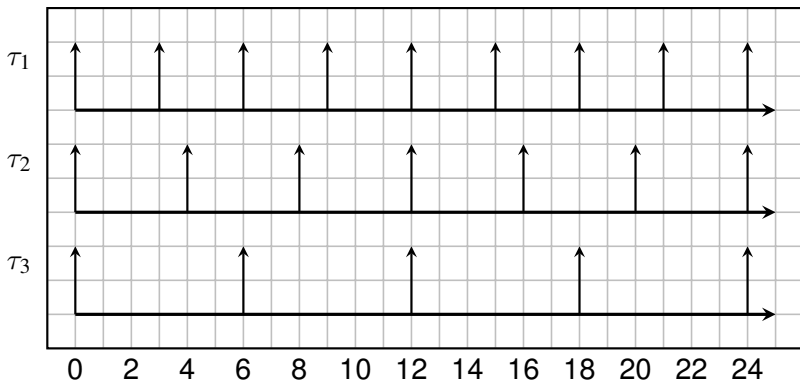
$$H(\mathcal{T}) = \text{lcm}_{\tau_i \in \mathcal{T}}(T_i)$$

- The patterns of arrival repeats every hyperperiod. In practice, if two tasks arrive at the same time  $t$ , they will arrive at the same time  $t + kH$ , for every integer number  $k \geq 0$ ;
- Sometimes, the hyperperiod is defined also for task sets including sporadic tasks.



# Hyperperiod: example

- Consider a task set with 3 tasks, with the following periods:  
 $T_1 = 3$ ,  $T_2 = 4$ ,  $T_3 = 6$ . The hyperperiod is  $H = 12$ . We show below the pattern of arrivals.



# Offsets

- A periodic task can have an *initial offset*  $\phi_i$
- The offset is the arrival time of the first instance of a periodic task;
- Hence:

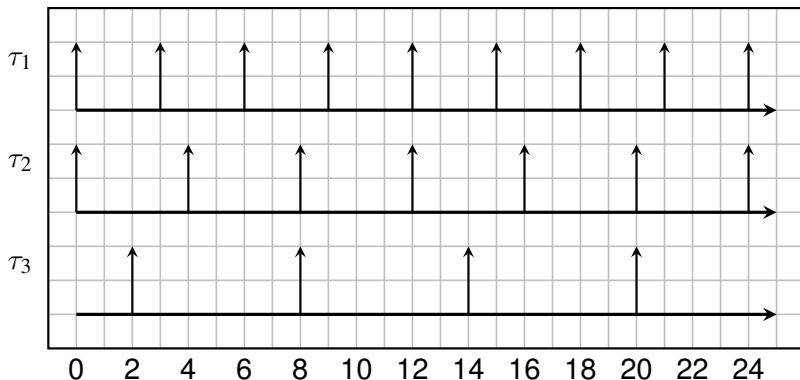
$$a_{i,0} = \phi_i$$

$$a_{i,k} = \phi_i + kT_j$$

- In some case, offsets may be set to a value different from 0 to avoid all tasks starting at the same time

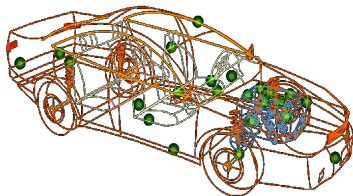
# Example of offsets

- Consider the previous example, 3 tasks with the following periods and offsets:  $T_1 = 3$   $\phi_1 = 0$ ,  $T_2 = 4$   $\phi_2 = 0$ ,  $T_3 = 6$   $\phi_3 = 2$ .



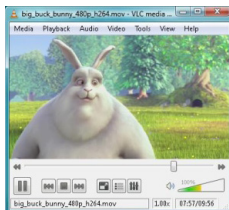
# Hard real-time tasks

- a task is said *hard real-time* if all the its deadlines must be respected, otherwise a critical failure occurs in the system;
  - therefore, we have to guarantee *a-priori*, before the system runs, that all deadlines will be respected under all possible conditions;
- An example of hard real-time task:
  - from the detection of an enemy missile, a maximum of 2 seconds must pass before the defense missile is launched and directed on target, otherwise it will be too late!



# Soft real-time tasks

- In a *soft real-time* task, nothing “catastrophic” happens if a deadline is missed;
- Some deadline can be missed with little or no consequences on the correctness of the system;
- However, the number of missed deadline must be kept under control, because the “quality” of the results depend upon on the number of deadline missed;



- A video player is a typical example of soft real-time application.

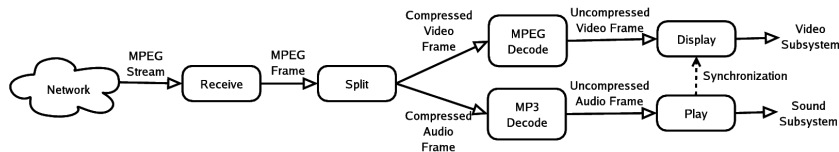
# Requirements on soft real-time tasks

- Example of requirements on soft tasks;
  - no more than  $X$  consecutive deadlines can be missed;
  - no more than  $X$  missed deadline in an interval of time  $T$ ;
  - the *deadline miss ratio* (i.e. percentage of total missed deadlines over the total number of deadlines) must not exceed a certain threshold;
  - the maximum *tardiness* (i.e. the ratio of the worst-case response time and the relative deadline) must not exceed a certain threshold;
  - etc.

# Example of soft real-time task

The classical example of soft real-time task is a MPEG player.

- suppose that the frame rate of a video movie is 25 fps;
- this means that one video frame has to be loaded from disk, decoded and displayed every 40 msec;



- the goal is to *minimise* the number of missed deadlines.



# Hard and soft

- In a system, often there is a mixture of hard and soft real-time tasks. For example:
  - In a critical system, in addition to critical control tasks, we have logging tasks that collect information on the functionality of the system for monitoring;
  - Also, communication over the network of this information is not a critical task, and if some packet is lost nothing catastrophic happens;
  - Some control task can also be non critical; for example, in a robotic system, some actuation may be delayed a little more with little consequences (degradation of the quality of the control).



# Outline

- 1 Introduction to Real-Time Systems
- 2 Task Model
- 3 Computation times and workload**
- 4 Scheduling
- 5 Other parameters

# Worst case and average case

- Tasks can have variable execution times between different jobs
- The execution time depends on different things:
  - input data;
  - the hardware architecture (presence of cache, pipeline, etc.);
  - internal conditions;

# Worst-case computation: example

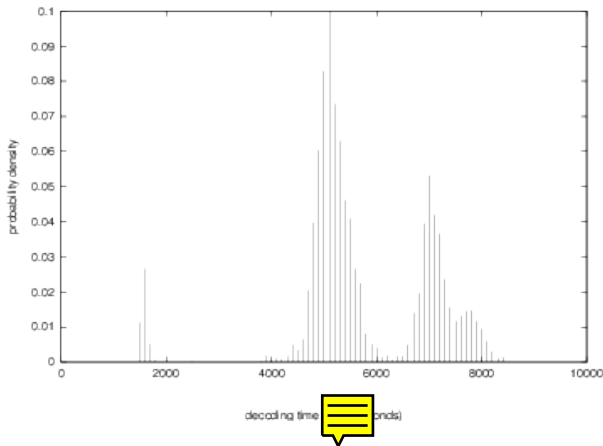
- Tasks can have variable execution times between different jobs

```
while (cond) {  
    if (a > 10) {  
        // long computation  
    } else {  
        // short computation  
    }  
    a = // external input;  
}
```

We have two sources of variability: the value of variable `a` influences which computation is performed (long or short); the value of `cond` influences the number of times the cycle is executed.

## Example: MPEG player

- An algorithm with highly variable computation time is the decoding of a MPEG frame: in the figure below, we show the distribution of the execution times of decoding “Star Wars”.



# Computing the WCET

- Computing the WCET of a task is possible in principle;
  - It consists in computing all possible paths that the program may follow;
  - Computing the number of processor cycles needed for every path;
  - Selecting the path with the maximum number of cycles;
- However, in practice the problem is almost intractable:
  - the presence of the **cache** and of the **pipeline** makes the problem of computing the number of cycles very difficult;
  - most algorithm can only give an upper bound (i.e. the selected path may not be possible in the program execution, the number of cycles is overestimated, etc.);
  - unfortunately such bounds are often too large (like 2 or 3 times the real WCET);



# Computing the WCET - II

- Practically, today the WCET is estimated by executing the program several times over a large number of different input data;
- However, since we cannot execute the program for *every* input data, the measured bound is not safe!
  - (i.e. the real WCET may be *larger* than the one that has been measured).

# Maximum Utilization

- The maximum *utilization* of a task  $\tau_i$ , and of the whole system, are defined respectively, as:

$$U_i = \frac{C_i}{T_i} \quad U = \sum_{i=1}^n U_i = \sum_{i=1}^n \frac{C_i}{T_i}$$

- The density of a task and of the system are defined respectively as:

$$\delta_i = \frac{C_i}{D_i} \quad \delta = \sum_{i=1}^n \delta_i$$

- Example:

Task	$C$	$D$	$T$	$U$	$\delta$
$\tau_1$	2	6	6	0.333	0.33
$\tau_2$	2	5	8	0.25	0.4
$\tau_3$	2	10	12	0.166	0.2
				0.75	0.933



# Outline

- 1 Introduction to Real-Time Systems
- 2 Task Model
- 3 Computation times and workload
- 4 Scheduling**
- 5 Other parameters



# Schedule

Let us consider a system consisting of one processor.

- a *schedule* is a function  $\sigma(t)$  that maps each instant of time into a task or to  $\emptyset$ .

$$\sigma : t \rightarrow \mathcal{T} \cup \emptyset$$

- If  $\sigma(t) = \emptyset$ , then the processor is idle.
- The definition can be easily extended to  $m$  processors, by considering a function  $\sigma(t)$  that maps instants of time into vectors of elements of  $\mathcal{T} \cup \emptyset$ .

# Scheduling, schedulability, feasibility

- Scheduling algorithm

- An on-line or off-line algorithm  $\mathcal{A}$  that, given a task set  $\mathcal{T}$  decides which tasks are executed at each instant on each processor (the *schedule*  $\sigma(\mathcal{A}, \mathcal{T}, t)$ )

# Scheduling, schedulability, feasibility

- Scheduling algorithm

- An on-line or off-line algorithm  $\mathcal{A}$  that, given a task set  $\mathcal{T}$  decides which tasks are executed at each instant on each processor (the *schedule*  $\sigma(\mathcal{A}, \mathcal{T}, t)$ )

- Schedulable task set

- A task set  $\mathcal{T}$  is schedulable by algorithm  $\mathcal{A}$  if all jobs complete before their deadlines in the schedule  $\sigma(\mathcal{A}, \mathcal{T}, t)$

# Scheduling, schedulability, feasibility

- Scheduling algorithm

- An on-line or off-line algorithm  $\mathcal{A}$  that, given a task set  $\mathcal{T}$  decides which tasks are executed at each instant on each processor (the *schedule*  $\sigma(\mathcal{A}, \mathcal{T}, t)$ )

- Schedulable task set

- A task set  $\mathcal{T}$  is schedulable by algorithm  $\mathcal{A}$  if all jobs complete before their deadlines in the schedule  $\sigma(\mathcal{A}, \mathcal{T}, t)$

- Schedulability test

- Given a scheduling algorithm  $\mathcal{A}$ , and a set of tasks  $\mathcal{T}$ , decide if  $\mathcal{A}$  will produce a feasible schedule (i.e. all jobs complete before their deadlines)

# Scheduling, schedulability, feasibility

- Scheduling algorithm

- An on-line or off-line algorithm  $\mathcal{A}$  that, given a task set  $\mathcal{T}$  decides which tasks are executed at each instant on each processor (the *schedule*  $\sigma(\mathcal{A}, \mathcal{T}, t)$ )

- Schedulable task set

- A task set  $\mathcal{T}$  is schedulable by algorithm  $\mathcal{A}$  if all jobs complete before their deadlines in the schedule  $\sigma(\mathcal{A}, \mathcal{T}, t)$

- Schedulability test

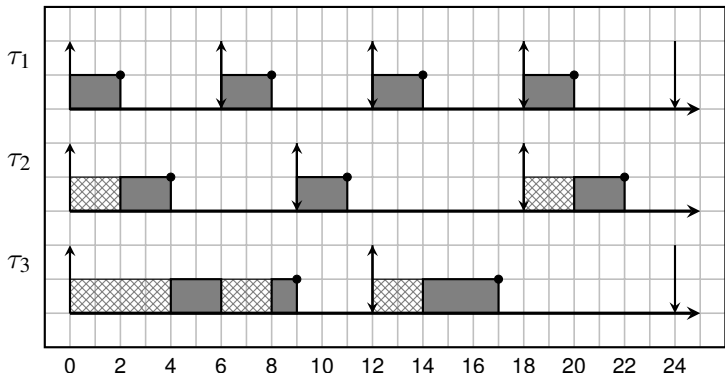
- Given a scheduling algorithm  $\mathcal{A}$ , and a set of tasks  $\mathcal{T}$ , decide if  $\mathcal{A}$  will produce a feasible schedule (i.e. all jobs complete before their deadlines)

- Feasibility problem

- Given a set of tasks  $\mathcal{T}$ , decide if it exists a scheduling algorithm  $\mathcal{A}$  that produces a feasible schedule on  $\mathcal{T}$ .

## Example of schedule

- Fixed priority: the active task with the highest priority is executed on the processor.
- Consider the following task set:  $\tau_1 = (2, 6, 6)$ ,  $\tau_2 = (2, 9, 9)$ ,  $\tau_3 = (3, 12, 12)$ . Task  $\tau_1$  has priority  $p_1 = 3$  (highest), task  $\tau_2$  has priority  $p_2 = 2$ , task  $\tau_3$  has priority  $p_3 = 1$  (lowest).



# Schedulability test

- One key objective of *real-time analysis* is to be able to know in advance if the task set is schedulable by a certain scheduling algorithm
- Generate and check the schedule (hint: it is a periodic function)
  - Pro: in this case, feasibility can be reduced to a classical MILP problem
  - Cons: NP-Hard

# Schedulability test

- One key objective of *real-time analysis* is to be able to know in advance if the task set is schedulable by a certain scheduling algorithm
- Generate and check the schedule (hint: it is a periodic function)
  - Pro: in this case, feasibility can be reduced to a classical MILP problem
  - Cons: NP-Hard
- Worst-case approach: try to identify *worst-case scenario*
  - Pro: feasibility in polynomial (or pseudo-polynomial) complexity
  - Cons: not quite easy to identify the worst-case
  - Cons: often, only sufficient conditions



# Classification of scheduling algorithms

- **Work conserving schedule**: A processor will never be idle if there are ready tasks
- Preemption
  - **Non-Preemptive**: once a task starts executing, cannot go back to the ready state
  - **Preemptive**: a task can be suspended at any time
- Priority
  - **Static** (fixed) priority: every task is assigned a fixed priority that will not change during its lifetime
  - **Job-level** static priority: every job is assigned a priority that does not change until the job is completed (e.g. EDF)
  - **Job-level** dynamic priority: priority can change at any moment (e.g. LLF)
- Multi-processors:
  - **Partitioned**: tasks are statically assigned to processors
  - **Global**: tasks can migrate from one processor to another one

# Difference with Classical Scheduling problems

- In classical scheduling problems (i.e. job-shop)
  - Tasks are non-recurrent (not periodic)
  - No timing constraints
  - Goal is to minimise completion time (the *make-span* problem), or some *cost function*
  - Resources can be complex (different machines, precedence constraints, etc.)
  - The general form is often only solvable by Mixed-Integer Linear Programming
- In real-time scheduling
  - tasks are periodic or sporadic
  - emphasis on time constraints
  - resources are simple (single processors, uniform multiprocessors)
  - many problems can be solved in polynomial (or pseudo-polynomial) time

# Outline

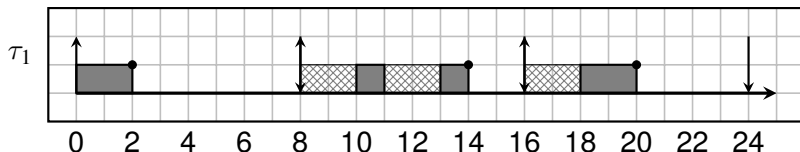
- 1 Introduction to Real-Time Systems
- 2 Task Model
- 3 Computation times and workload
- 4 Scheduling
- 5 Other parameters**

## Other important parameters

- in some case, even if the job has been activated, cannot start executing because it must wait an additional event;
  - the *start time jitter*  $stj_i$  is the maximum delay between the activation time  $a_{ij}$  and the time the job can actually start executing;
- the *output jitter*  $oj_i$  is the maximum of the absolute value of the difference between the distance of two consecutive finishing times and the period:

$$oj_i = \max_k \{|f_{i,k+1} - f_{i,k} - T_i|\}$$

Example:



$$oj_1 = \max\{|14 - 2 - 8|, |20 - 14 - 8|\} = 4.$$

# Considerations on the output jitter

- Usually, a job produces its outputs at the end of its execution;
- for example, just before suspending, a task sends the control command to the actuators;
- in general, it is a good idea to provide the actuation command at regular (periodic) instants;
- If we know the delay, and it is constant, it is possible to compensate for it at design time;
  - ideally, the output jitter should be equal to 0;
  - This means that the task finishes always at the same time with respect to the activation instant:  $\rho_{i,k} = \text{const}$
  - however, due to the presence of other tasks, and to scheduling, this may be impossible.
- the larger is the output jitter, the larger is the “noise” on the actuator.