# Real-time Systems
# Introduction

Tullio Facchinetti
<tullio.facchinetti@unipv.it>

24 novembre 2023

http://robot.unipv.it/toolleeo

## Tasks having temporal constraints

which, among the following tasks, can be defined as a real-time task?

- the periodic sampling of a sensor every 30 minutes
- sending/receiving a TCP/IP packet every $10\mu s$ (in average)
- the framerate of a videogame, which is 60 FPS
  - what would you say if it is 12 FPS?
- the video stream of a video-conference is 25 FPS, but sometimes some frames are skipping/freezing
- the update of financial information have frequency of 20ms
  - what would you say if it is 20 seconds?
  - and if it is 20 minutes?
- opening a program (software) takes 10 seconds from the mouse click, on average

## Definition of real-time system

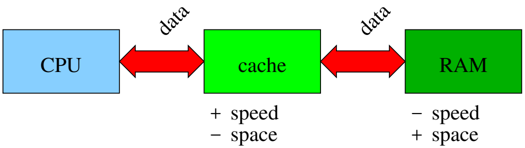
the correct behavior of a real-time system depends not only on the logic correctness of the result produced by the calculation, but also from the time at which the result is made available

- in general, there are temporal constraints that must be satisfied
- temporal constraints depend on the interaction between the digital system and the process
- a result that is numerically correct but that is provided too late may cause problems as big as a wrong result

## Real-time and fast response time

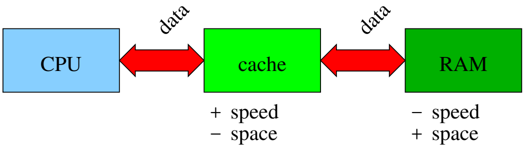
- the concept of “fast” is relative to the process
- the same system may be fast in some environments while being slow in others
- when (actually, always) more tasks must be managed by the same computing system, real-time systems aim at achieving the temporal constraints of each one
- fast systems usually aim to provide a low average response time on the set of tasks (all computer benchmarks are based on average values)
- guarantees on average values can not be trusted when individual temporal constraints must be achieved

## Worst vs average cases: the example of the cache memory



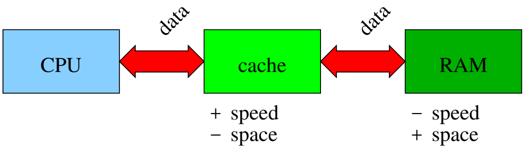
- the **cache memory** is a hardware technology implemented in most of computers
- it improves the access speed to the information stored in memory (on average)
- it works by maintaining a copy of information (caching) in a memory located between the processor and the RAM

## Worst vs average cases: the example of the cache memory



- fetching information from the cache memory is much faster than from RAM
- such faster memory technology is much more expensive than RAM
- therefore, cache memory is much smaller than RAM (e.g., Intel Core i7-5775C: 6MB – RAM can be 16GB)
- moreover, several levels of cache are usually implemented (Intel Core i7-5775C: L1: 4x64KB, L2: 4x256KB, L3: 6MB)

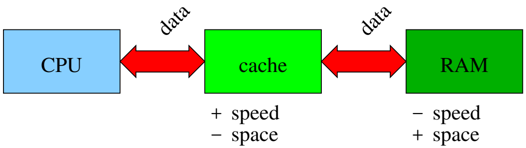
## Worst vs average cases: the example of the cache memory



- the information stored in RAM can not fit within the cache memory
- the cache controller must decide which is the information to keep within the cache (many sophisticated policies have been developed)

there is a continuous exchange of information between cache and RAM in order to keep data sync'ed

## Worst vs average cases: the example of the cache memory



- if the information can be fetched from the cache memory there is a **cache hit**
- if not, there is a **cache miss**
- the efficiency of the cache memory technology is based on the **data locality principle**

however, the performance may decrease in the worst case w.r.t. the absence of cache

## The example of the cache memory: notation

- $t_m$ : time to access data stored in RAM
- $t_c$ : time to access data stored in cache memory
- for $n >> 1$ memory accesses there are
  - $n_h$ cache hit
  - $n_m$ cache miss
- total memory accesses: $\text{n} = n_h + n_m$

cache is faster than RAM:

$$t_c << t_m$$

principle of locality of references:

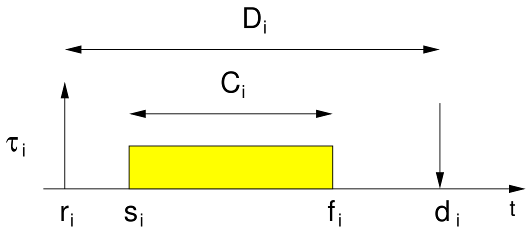
$$n_h >> n_m$$

## The example of the cache memory: evaluation

| | $t_{avg}$ | $t_{max}$ |
|---|---|---|
| cache | $(1/n)(t_c n_h + (t_m + t_c) n_m)$ | $t_m + t_c$ |
| no cache | $t_m$ | $t_m$ |

- the cache memory achieves $t_{avg} \to t_c$ for $n_m \to 0$ (best average performance)
- the access time of the cache in the worst conditions ($t_{max}$) is higher than without cache

## Real-time task model



**absolute parameters**

$r_i$ – release time (or arrival time $a_i$)

$s_i$ – start time

$d_i$ – absolute deadline

$f_i$ – finishing time

**relative parameters**

$C_i$ – worst-case execution time (WCET)

$D_i$ – relative deadline

## Criticality

tasks can be classified as:

- hard: the violation of a temporal constraint may have "catastrophic" consequences on the system
- soft: a limited number of violations of temporal constraints bring to a tolerable degradation of the performance

examples of (typical) hard/soft tasks

- hard: sensor sampling, actuator operation, control loops, communication
- soft: user I/O (keyboard input, messaging, ecc.), multimedia streaming

a computing system able to deal with hard tasks
is said hard real-time

## Type of constraints

tasks can be associated to different kind of constraints

### temporal constraints

- activation instant, finishing time, jitter, ...

### precedence constraints

- refers to the temporal ordering between task

### resource constraints

- mutual exclusion, synchronization

## Temporal constraints

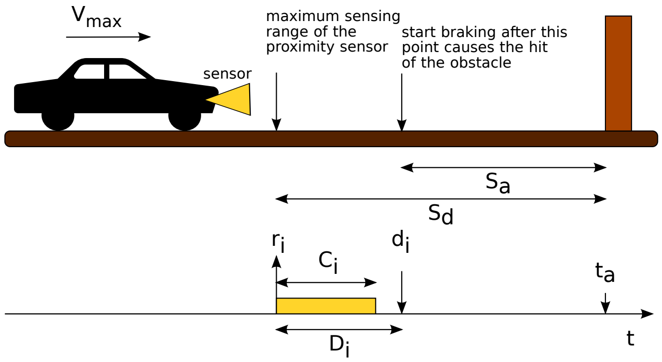
### explicit constraints

- they are explicitly stated by the problem formulation
- e.g.: "the sensor must be sampled every 20ms"

### implicit constraints

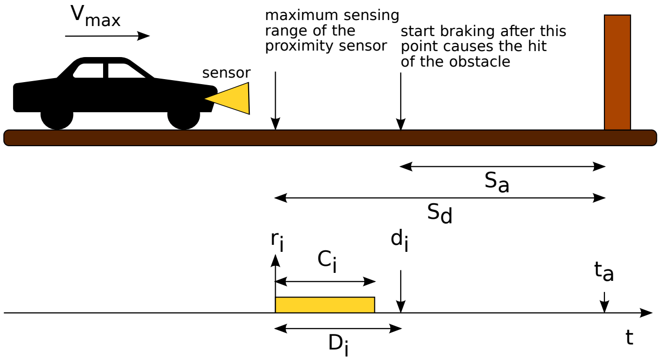
- they must be inferred from the problem parameters, where they do not appear explicitly
- e.g.: "the sensor shall detect an object having length $L = 10\text{cm}$ travelling at max speed of $V_{max} = 1\text{m/s}$"

## From physical to temporal parameters


$V_{max}$
sensor
maximum sensing range of the proximity sensor
start braking after this point causes the hit of the obstacle
$S_a$
$S_d$
$r_i$
$C_i$
$d_i$
$t_a$
$D_i$
t

- $S_a$ : minimum braking distance to stop safely
- $S_d$ : maximum sensing range of the sensor

## From physical to temporal parameters


$V_{max}$
sensor
maximum sensing range of the proximity sensor
start braking after this point causes the hit of the obstacle
$S_a$
$S_d$
$r_i$
$C_i$
$d_i$
$t_a$
$D_i$
t

- $r_i$ : time when the sensor detects the obstacle
- $d_i$ : latest instant to start braking to stop without hitting the obstacle $(D_i = d_i - r_i)$
- $C_i$ : duration of the task controlling the braking system

## From physical to temporal parameters

### dynamic friction force:

$$F = -\mu m g$$

where

- $\mu$ : friction constant
- $m$ : the vehicle mass

### deceleration (Second Newton's Law):

$$a = F/m = -\mu g$$

### law of the uniform acceleration (kept it simple):

$$x(t) = vt - \frac{1}{2}\mu g t^2$$

$$v(t) = v - \mu g t$$

## From physical to temporal parameters

the system must be evaluated in worst conditions, i.e., when the speed equals the maximum speed $v_{max}$

at time $t_a$ the vehicle stops, i.e., $v(t_a) = 0$; therefore

$$t_a = \frac{v_{max}}{\mu g}$$

by substitution within the first equation:

$$x(t_a) = S_a = \frac{v_{max}^2}{2\mu g}$$

## From physical to temporal parameters

### known quantities:

- $S_a$ : just calculated
- $S_d$ : depends from the adopted sensor
- $C_i$ : known, since the implemented task has a given worst-case duration under the considered computing platform

the maximum time to travel the distance $S_d - S_a$ in the worst case (assuming $S_d > S_a$, otherwise.... CRASH!!):

$$D_i = \frac{S_d - S_a}{v_{max}}$$

it is possible to state whether the timing required for the computation is suitable for the dynamics (timing constants) of the system, or tailor the computing parameters accordingly

## Sources of temporal non-determinism

### computing architecture

- cache, pipelining, interrupts, DMA

### operating system

- scheduling, synchronization, communication

### programming languages

- in most languages there is no explicit support for the specification of temporal constraints

### design methodologies

- lack of methods and tools for analysis and verification

## Solutions and drawbacks

### common solutions:

- low level programming (assembler)
- timing imposed by explicit manipulation of hardware/software timers
- processing within device drivers (faster response time)
- explicit manipulation of task priorities

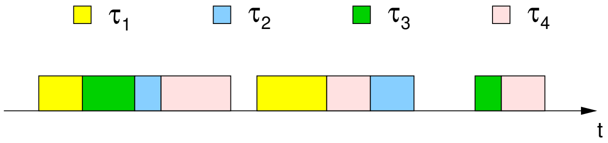
### drawbacks:

- time-consuming development, strongly dependent on programmer's skills
- poor clarity of code
- reduced portability
- prone to errors

## The schedule

**Schedule**

a sequence of assignments of tasks to the processor

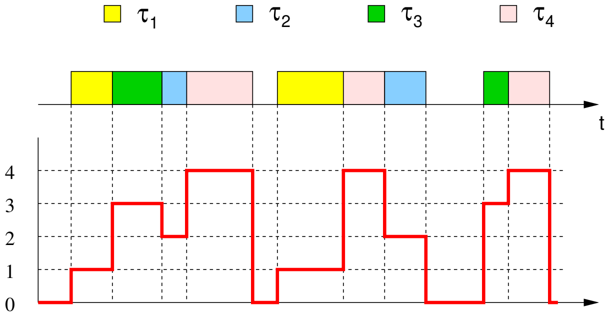


formally, given a task set $\Gamma = \{\tau_1, \ldots, \tau_n\}$, a schedule is a function $\sigma : \mathbb{R}^+ \to \mathbb{N}$ such that, for each time interval $[t_1, t_2)$, it holds

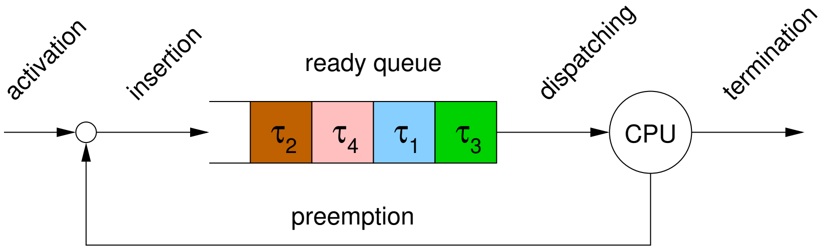
$$\sigma(t) = \begin{cases} k & \text{if } \tau_k \text{ is running at time } t \in [t_1, t_2) \\ 0 & \text{if the processor is idle} \end{cases}$$

## The schedule: an example

- the task set is composed by 4 tasks
- $\sigma = 0$ when no tasks are running



## The ready queue



- the ready queue contains **tasks** ready to execute
- ordered in descending priority order, such that the highest priority task will be the first to be fetched
- the **scheduler** is the OS component that inserts a task into the ready queue using some given criteria (called **scheduling algorithm**)

## Taxonomy of scheduling algorithms

| | | |
|---|---|---|
| preemptive | vs. | non-preemptive |
| static | vs. | dynamic |
| on-line | vs. | off-line |
| optimal | vs. | best effort |
| time-triggered | vs. | event-triggered |

for each class, an algorithm is classified in either one or the other category

example

- EDF is a preemptive, dynamic, on-line and optimal algorithm

## Preemptive vs. non-preemptive

preemptivity refers to the possibility (or not) to temporary stop the running task

### preemptive

- the kernel can stop the running task to execute one having higher priority

### non-preemptive

- once the execution of a task has started, it can not be stopped by the kernel until its execution has ended (or self-suspended)

## Static vs. dynamic

algorithms have different approaches
regarding the assignment of priorities

### static

- the priority of a task is based on static parameters
- the value of a static parameter does never change

### dynamic

- the priority of tasks is based on dynamic parameters
- the value of a dynamic parameter continuously changes over the time

## On-line vs. off-line

they differ regarding
**when** scheduling decisions are taken

### on-line

- scheduling decisions are made at run-time
- they are based on changing parameters, such as task activation time and temporal constraints

### off-line

- scheduling decisions are taken before starting the system
- they are suitably stored and applied at run-time

## Optimal vs. best effort

they optimize (or not) some cost function

**optimal** (regarding schedulability)

- if a feasible schedule exists, the scheduling algorithm will find it

**best effort**

- a suitable heuristic method is used to obtain the best possible result

## Time-triggered vs. event-triggered

they have different policies
regarding the activation of tasks

### time-triggered

- a timer triggers the activation of task on a regular basis
- periodic tasks

### event-triggered

- a task is triggered by an external event
- aperiodic/sporadic tasks

examples of tasks are:

- time-triggered: periodic sampling of sensors
- event-triggered: interrupt management (e.g., keyboard input)

## Task scheduling

**feasible schedule**

the schedule achieves the temporal constraints of all tasks

examples:

- every task terminates within its deadline
- the precedence order of tasks is guaranteed

given a task set, if a feasible schedule exists, then the task set is said **feasible**

NOTE: feasibility is **a property of the schedule** only; it is independent from the adopted schedulig algorithm

## Schedulability test

Given a scheduling algorithm, the schedulability test is a formula or algorithm that allows to check the guarantee of timing constraints of each task.

- The schedulability test **depends from the algorithm**, i.e., different algorithms have different schedulability tests.
- It can be applied **before** running the scheduling algorithm to know whether the task set is schedulable or not.

## The general problem of scheduling

given a **task set** $\Gamma$ and a **set of resources** $R$ (processors, variables, peripherals, etc.), the problem of finding a feasible schedule for $\Gamma$ is said general problem of scheduling

- the general problem of scheduling has NP-hard complexity
- in presence of simplifying assumptions, it is possible to fomulate algorithms having polynomial complexity

## Notes on the complexity

The complexity of an algorithm is related to the number of elementary steps that are required to complete.

- The number of elementary steps determines the time required to complete.
- The complexity is expressed as a function of the problem parameters.

## The Big O notation

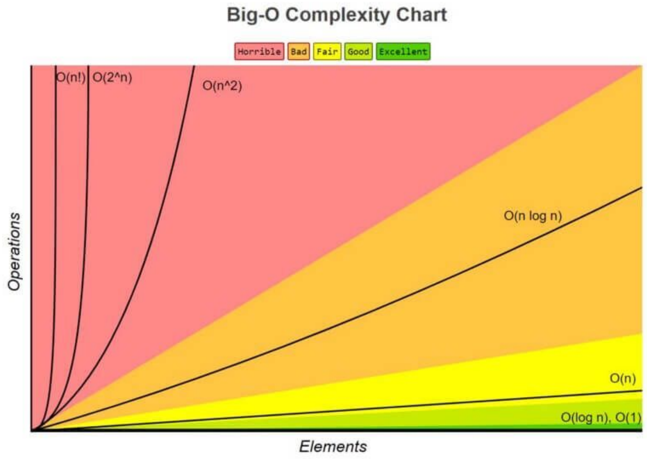
The complexity is typically expressed using the so-called Big O notation, indicated as $O(.)$.

Mathematically speaking, "$f(n)$ is $O(g(n))$" iff for some constants $c$ and $N_0$, it holds $f(N) \leq cg(N)$ for all $N > N_0$.

Informally:

- The Big O notation indicates **how fast a function increases**.
- For this reason, the Big O notation contains the term of the original function that makes it to increase faster (without constants).
- Example: the function $n^2 + 3n + 1$ is $O(n^2)$.

## Complexity comparison between common Big Os



From `https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/`

## Complexity comparison between common Big Os

in real-time systems, the typical parameter is the number of tasks $n$

- $O(1)$: does not depend on $n$ (i.e., it is constant)
- $O(n)$: linearly depends on $n$
- $O(n^4)$: polinomially depends on $n$
- $O(4^n)$: exponentially depends on $n$

let's assume $n = 50$ and a duration of the elementary step of 1ms

- $O(1)$: does not depend on $n$, e.g., it is equal to 20ms
- $O(n)$: milliseconds (e.g., 50ms)
- $O(n^4)$: hours (e.g., 1 hour, 44 min e 10 sec)
- $O(4^n)$: billions of billions of years (e.g., 40 b.b.y.)

## Common simplifying assumptions

1. only one processor
2. set of tasks having common features
3. full preemptive systems
4. simultaneous activations
5. no precedence constraints
6. no shared resources
7. no overhead (context switch, cache issues, etc.)