Real-time Systems
Introduction

Tullio Facchinetti
<tullio.facchinetti@unipv.it>

12th May, 2026

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

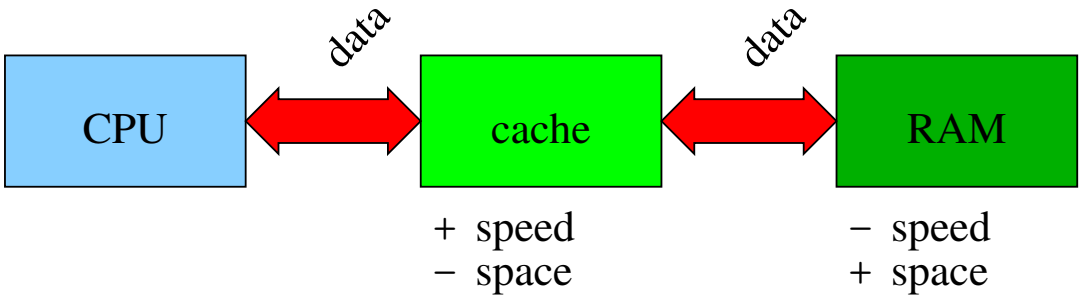
> the correct behavior of a real-time system depends not only on the logic correctness of the result produced by the calculation, but also from the time at which the result is made available

- 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

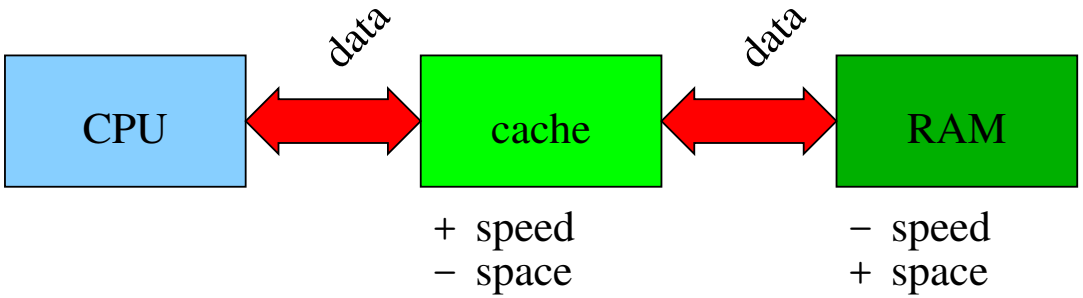
- the concept of "fast" is relative to the process
- the same system may be fast in some environments while being slow in others
- when (actually, always) more tasks must be managed by the same computing system, real-time systems aim at achieving the temporal constraints of each one
- fast systems usually 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



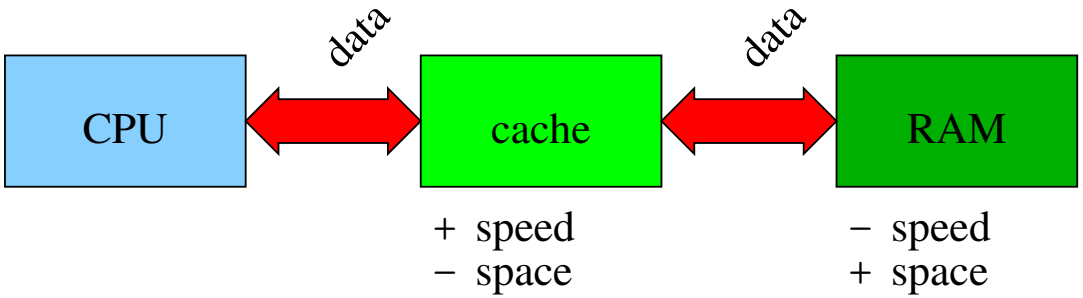
- The **cache memory** is a hardware technology implemented in most of computers
- It improves the **average** access time to the information stored in memory, to obatin a faster execution of programs
- It works by maintaining a copy of information (caching) in a memory located between the processor and the RAM

Cache memory: speed vs size



- Cache memory uses a significantly more expensive technology (Static RAM) than RAM (Dynamic RAM)
- The cost depends on the density of circuits, and thus on the area of the chip
- More expensive technology implies that fetching information from the cache memory is much faster than from RAM
- The size of cache memory is kept much smaller than RAM to limit the cost of processors

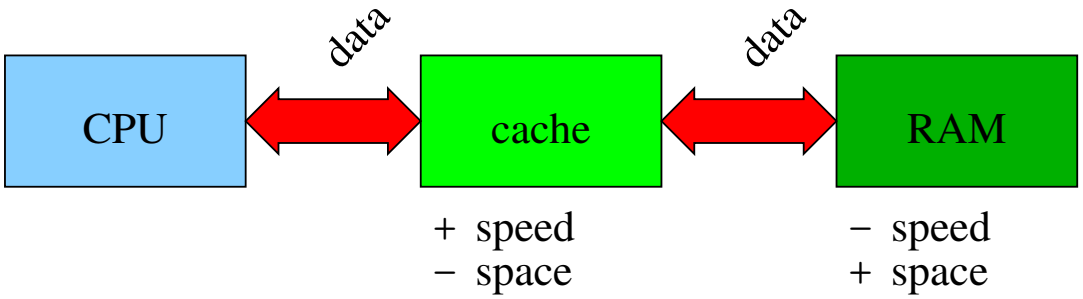
RAM vs cache



	RAM	cache
Size (Intel Core i7-5775C)	up to 16 GB	6 MB
Speed [ns]	$60 - 120$	$1 - 1.5$
Cost [\$ / mm^2]	$30 - 50$	$500 - 1,500$

- Several levels of cache are usually implemented to improve the trade-off among speed, size and cost
- E.g., Intel Core i7-5775C: L1: 4x64KB, L2: 4x256KB, L3: 6MB

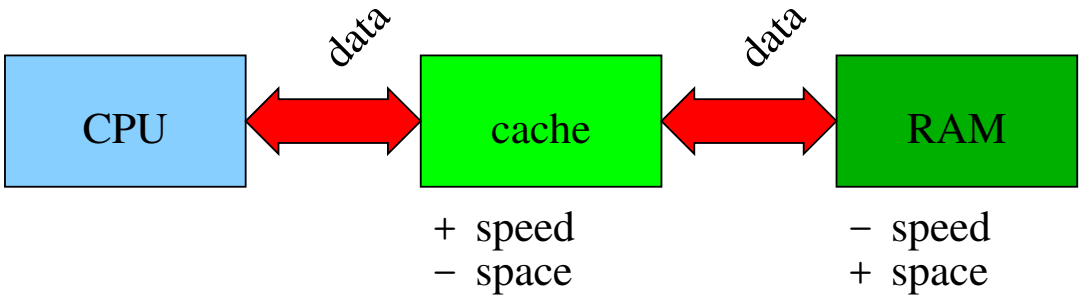
Cache memory: speed vs size



- The information stored in RAM can not fit within the cache memory
- The **cache controller** decides the information to keep within the cache (many sophisticated policies have been developed)

The data are exchanged when necessary between cache and RAM in order to keep data sync'ed.

Cache memory: hits, misses, and data locality



When the CPU looks for some data in memory:

- **cache hit**: the information can be fetched from the cache memory
- **cache miss**: the information is not cached, and must be retrieved from RAM

The efficiency of the cache memory technology is based on the **data locality principle**: many subsequent memory accesses address the same variables.

Data locality principle

The data locality principle says that programs should access data that is physically close in memory close together in time.

When this happens:

- fewer cache misses occur,
- less time is spent waiting for RAM,
- programs execute faster.

Data locality principle: example in C

A CPU cache loads memory in small blocks called **cache lines**.
For example, a cache line is made by 64 bytes, i.e., 16 integers.

```
/* Example of GOOD locality */
#define SIZE 1000000

int data[SIZE];

for (int i = 0; i < SIZE; i++) {
    data[i] = data[i] * 2;
}
```

- after loading `data[i]`, nearby values (`data[i+1]`, `data[i+2]`, ...) are already in cache
- therefore, most accesses are very fast

```
/* Example of BAD locality */
#define SIZE 1000000

int data[SIZE];

for (int i = 0; i < SIZE; i += 64) {
    data[i] = data[i] * 2;
}
```

- each access jumps to a different cache line
- previously loaded data is mostly unused
- this causes many expensive memory fetches from RAM

The example of the cache memory: notation

- t_{ram} : 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 hits
  - n_m cache misses
- total memory accesses: $n = n_h + n_m$

Cache is faster than RAM:

$$t_c << t_{ram}$$

Principle of locality:

$$n_h >> n_m$$

Cache memory: worst vs average cases

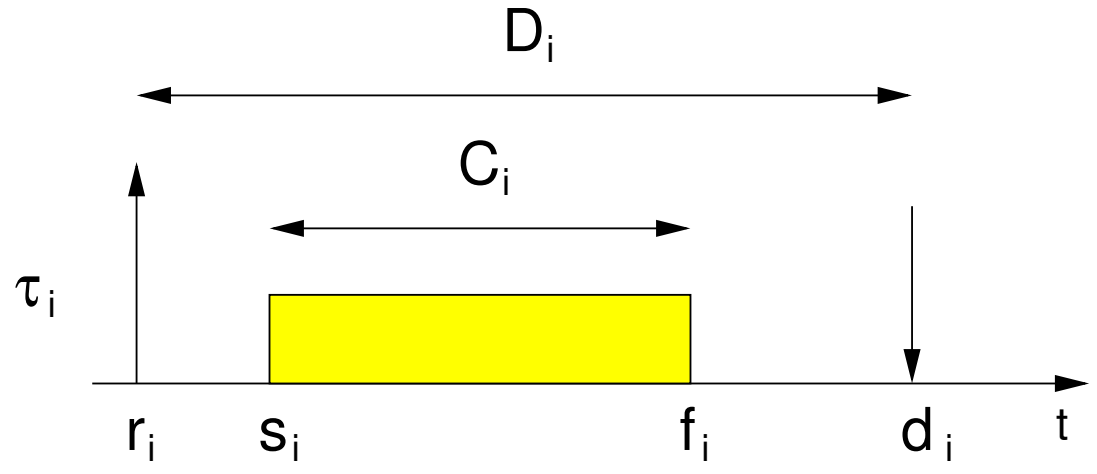
- n_h, n_m and n: number of hits, misses, and total accesses
- t_{ram} and t_c: time required to access RAM and cache

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

- the cache memory achieves $t_{avg} \to t_c$ ($<< t_{ram}$) 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

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

Real-time task model



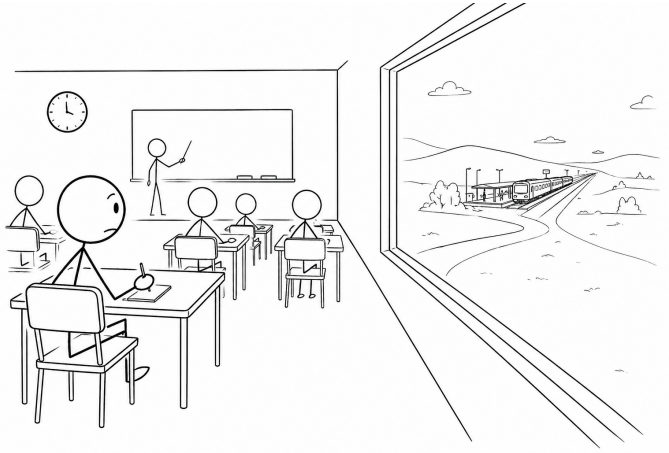
absolute parameters

- 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

Real-time in the real world



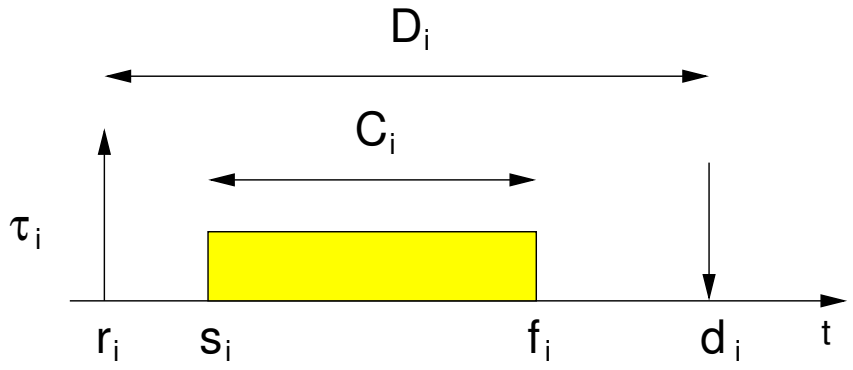
Context

- You are in the classroom attending the course
- You have to take the train at 17:00

How do you model the task of "going to take the train" in terms of real-time parameters?

Real-time in the real world

Model of the task of "going to take the train"
in terms of real-time parameters.



- What does the yellow rectangle represent?
- What is C_i ? And d_i ? And s_i ? And f_i ? And r_i ? And D_i ?
- What could be the reasons of having $s_i \neq r_i$?
- What is the timing constraint that you definitely want to achieve?

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

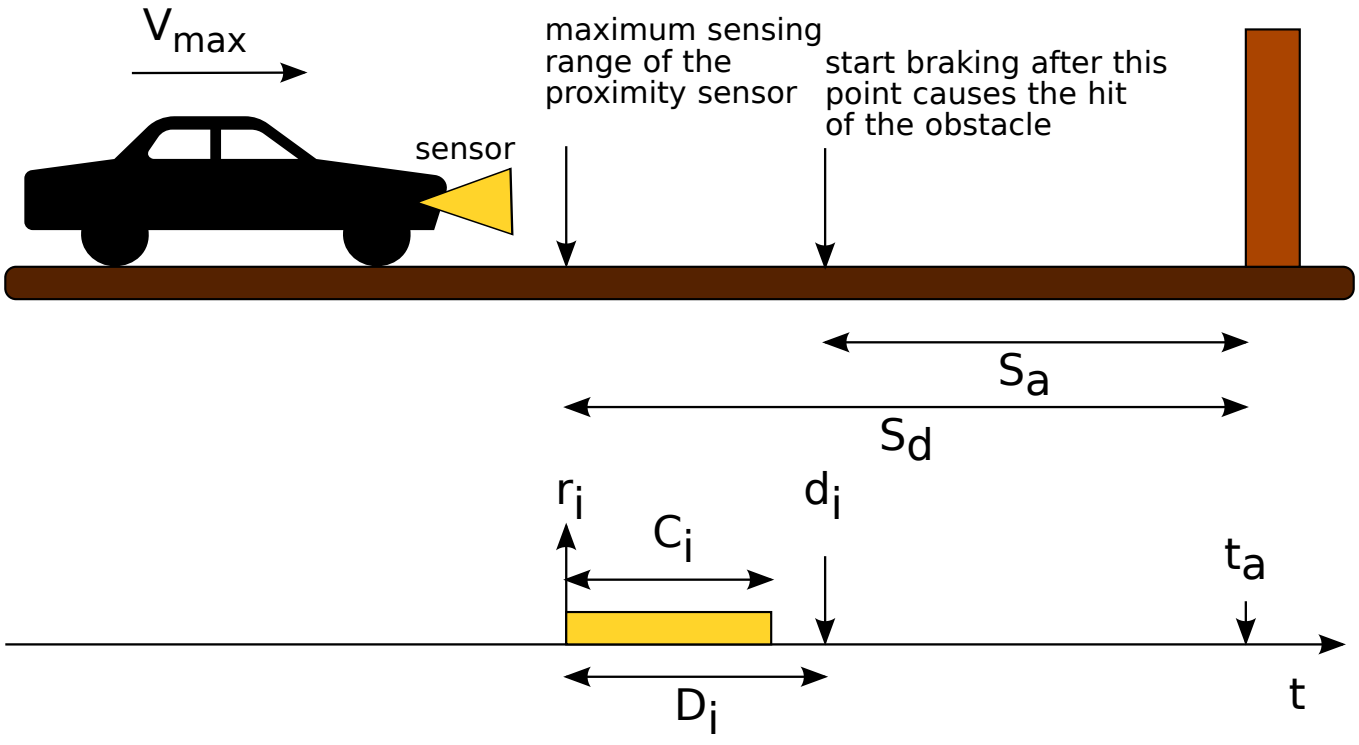
explicit constraints

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

implicit constraints

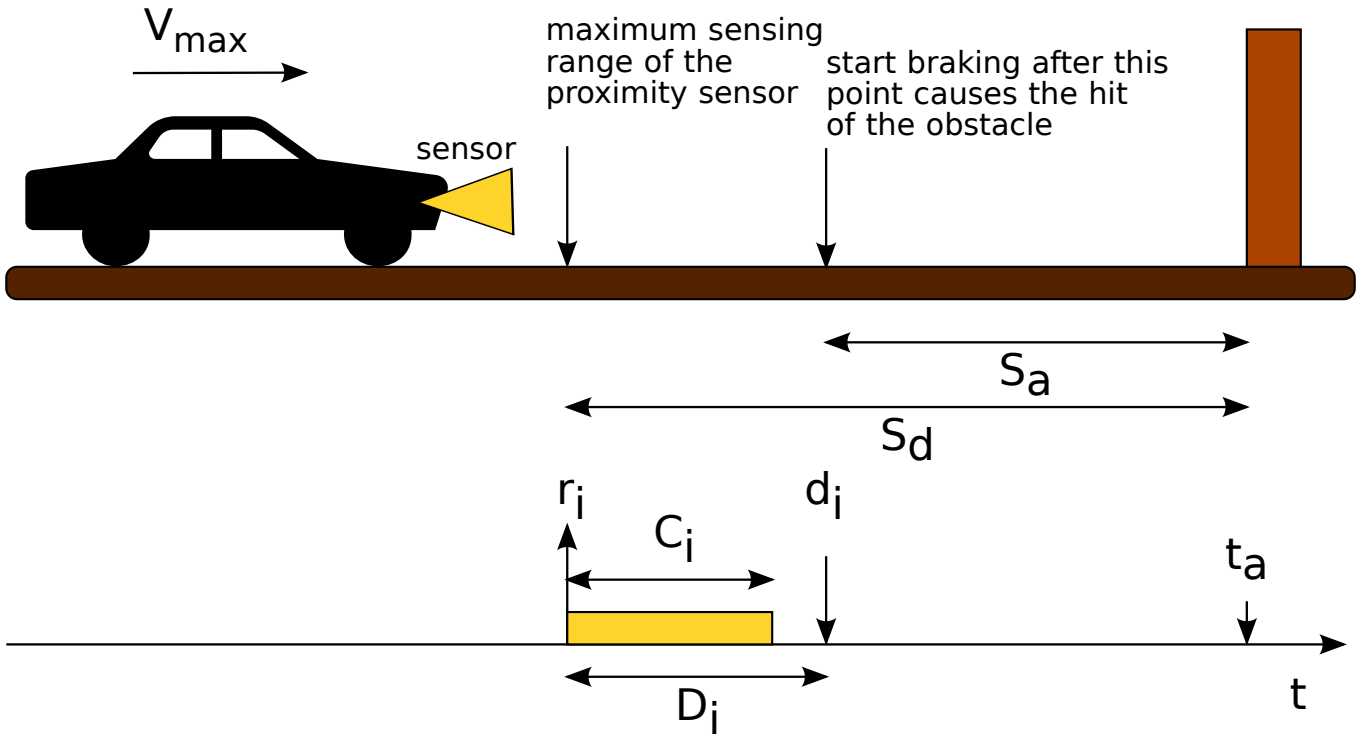
- they must be inferred from the problem parameters, where they do not appear explicitly
- e.g.: "the sensor shall detect an object having length $L = 10\text{cm}$ travelling at max speed of $V_{max} = 1\text{m/s}$"

From physical to temporal parameters



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

From physical to temporal parameters



- 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:

$$F = -\mu mg$$

where

- μ : 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

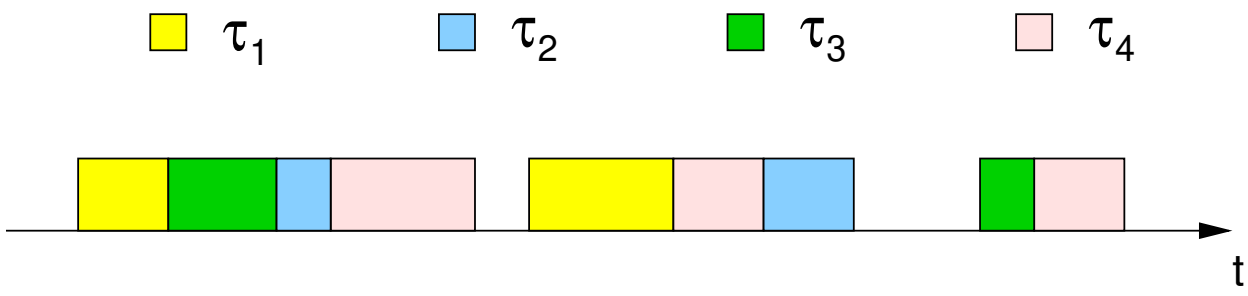
drawbacks:

- 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

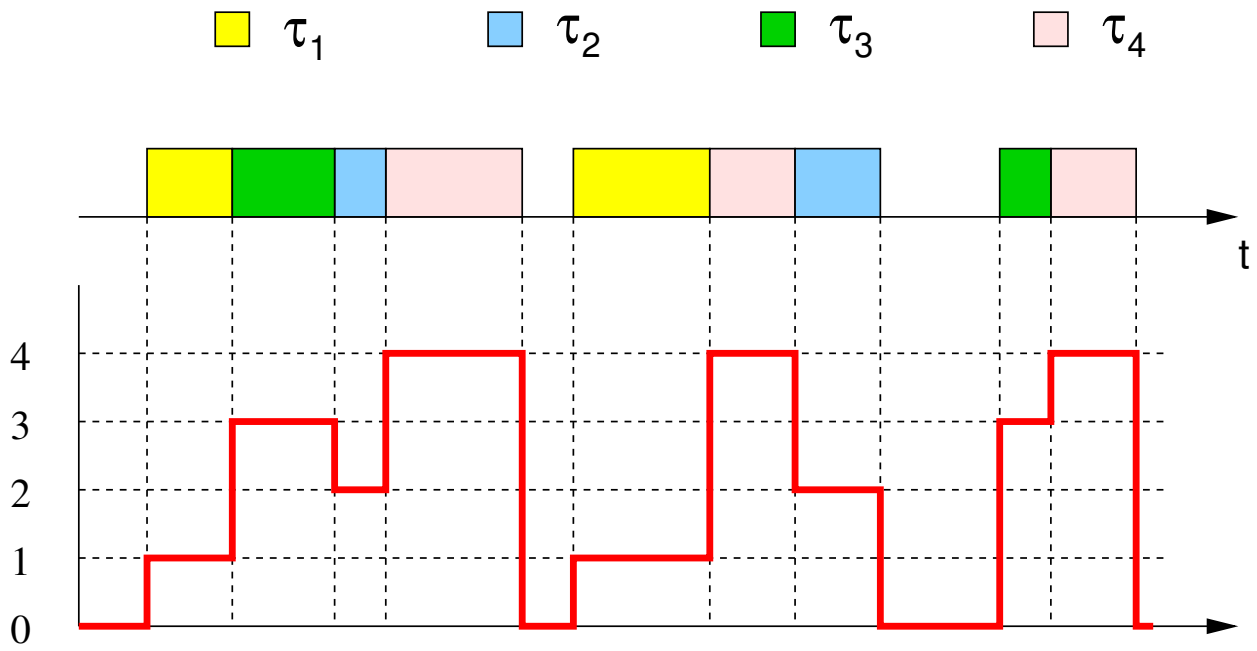


formally, given a task set $\Gamma = \{\tau_1, \ldots, \tau_n\}$, a schedule is a function $\sigma : \mathbb{R}^+ \to \mathbb{N}$ such that, for each time interval $[t_1, t_2)$, it holds

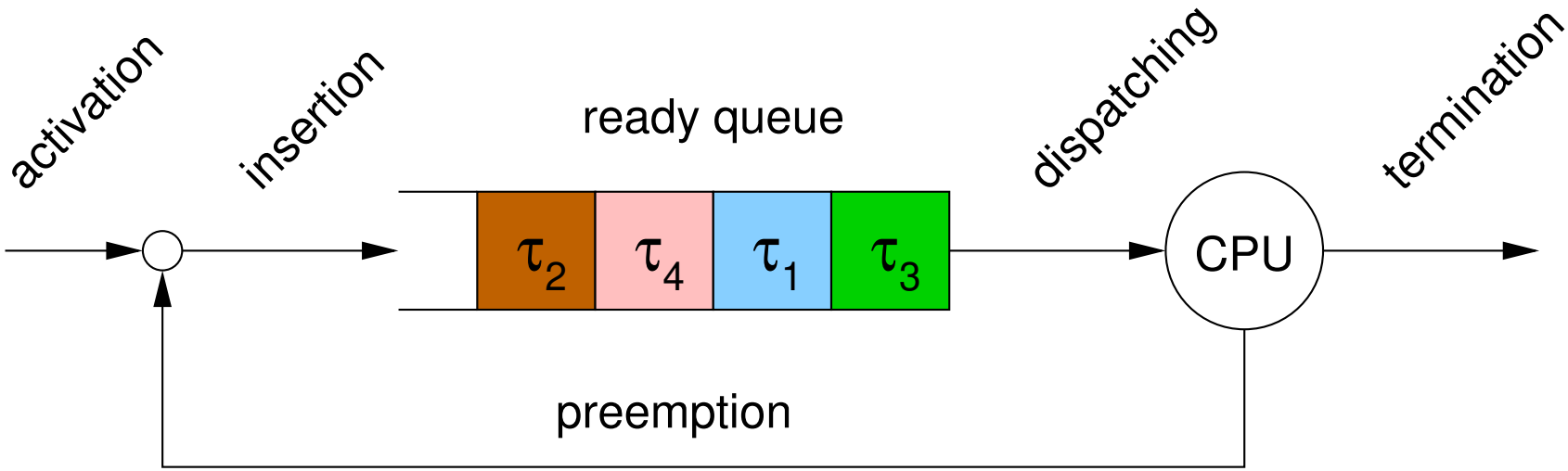
$$\sigma(t) = \begin{cases} k & \text{if } \tau_k \text{ is running at time } t \in [t_1, t_2) \\ 0 & \text{if the processor is idle} \end{cases}$$

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** Γ and a **set of resources** R (processors, variables, peripherals, etc.), the problem of finding a feasible schedule for Γ 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

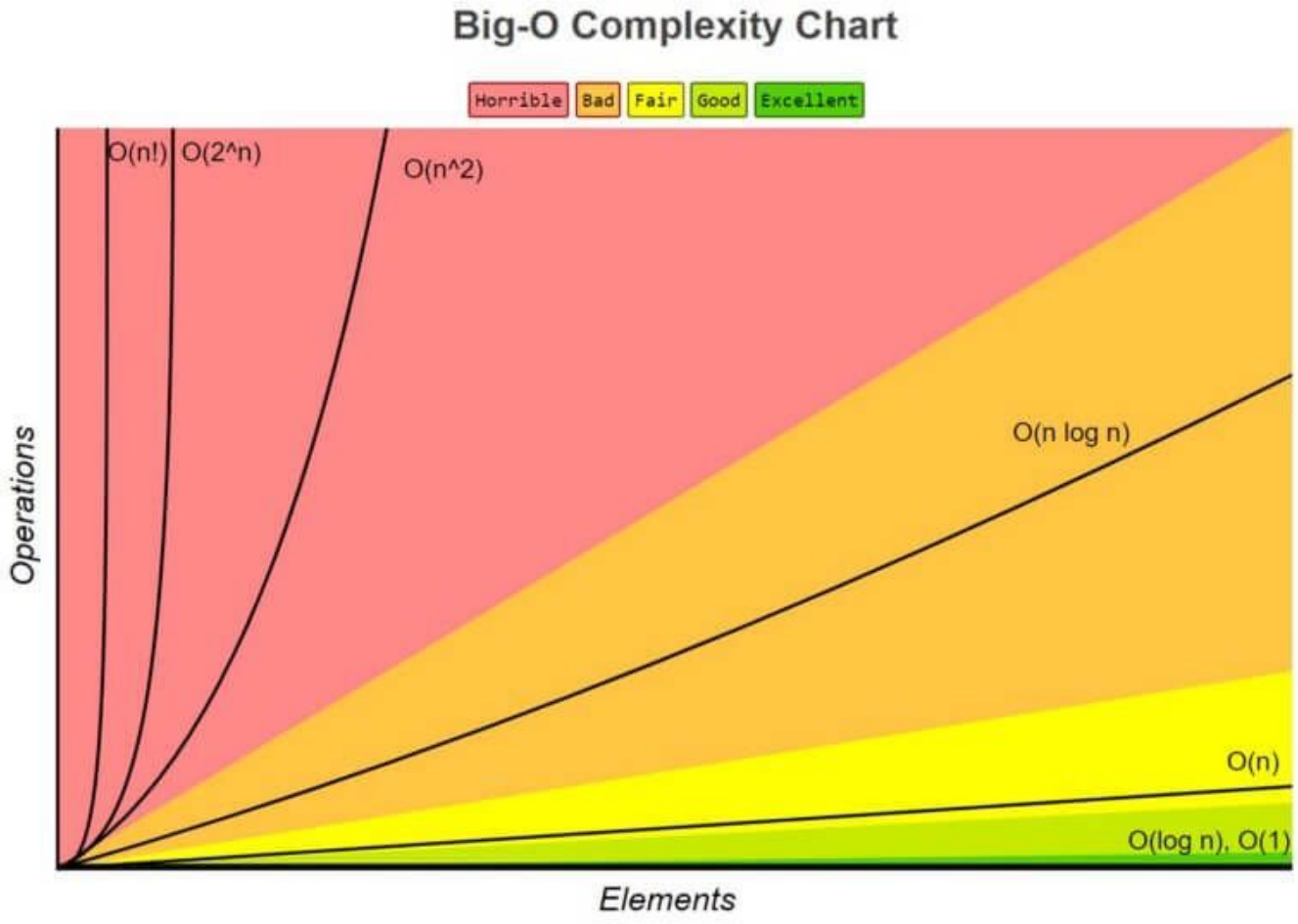
The complexity is typically expressed using the so-called Big O notation, indicated as $O(.)$.

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.)