Scheduling of Real-Time Tasks

Real-Time Industrial Systems

Marcello Cinque

Roadmap

- Operating Systems, Real-Time Operating Systems, standardization
- Scheduling of periodic tasks

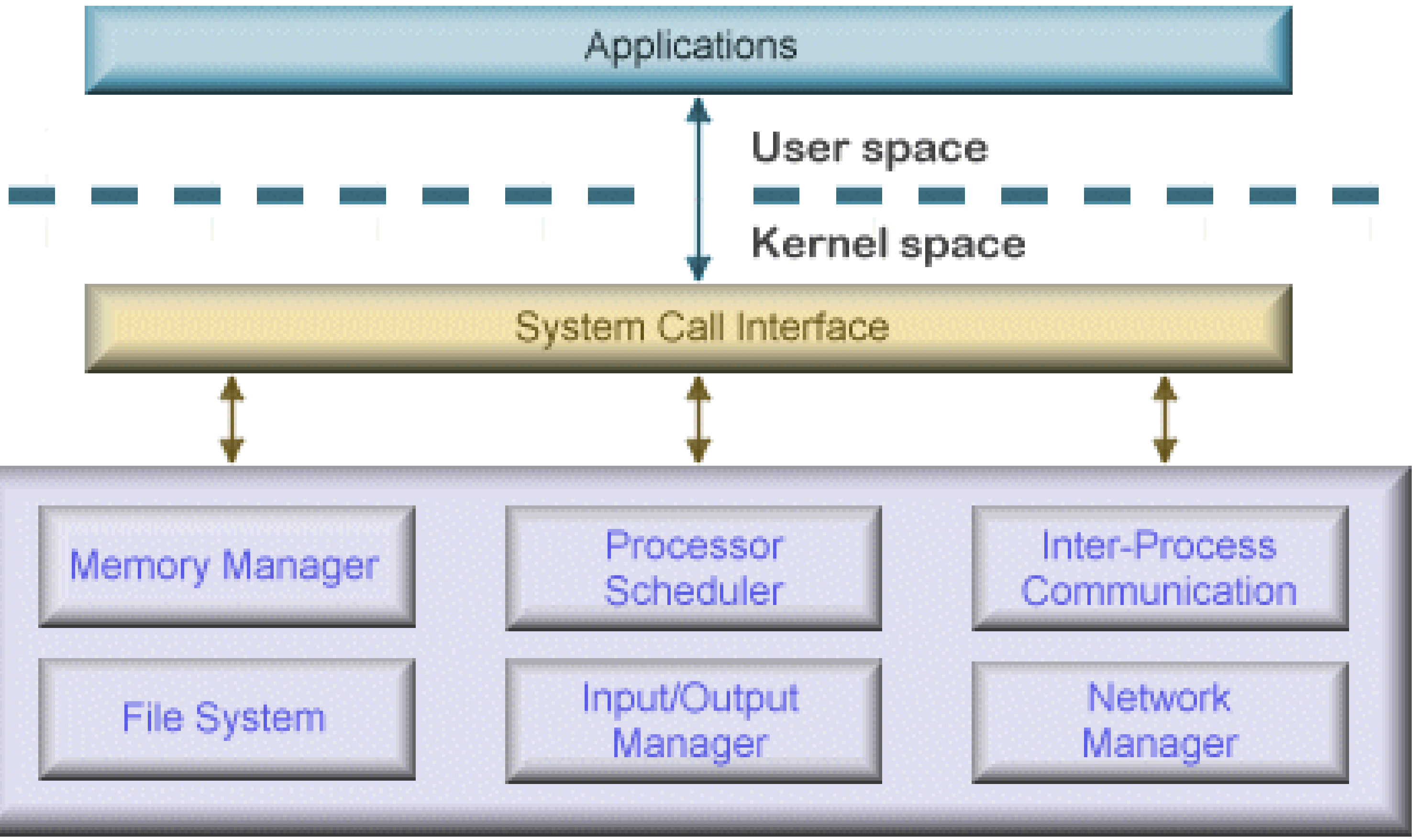
- Reference:
  - Giorgio Buttazzo: “Hard real-time computing systems: Predictable Scheduling Algorithms and Applications”, Third Edition, Springer, 2011

Operating systems and Real-time

- An OS can be defined as a set of programs that manage the hardware of a computer system
- It is the interface between hardware and software, providing an abstraction that is used to
  - Simplify the development of programs
  - Realize the policies to be used to manage the hardware resources

- Operating systems need to be ***re-designed*** for real-time systems, as some policies or design decision may introduce unbounded delays on real-time task operations
  - For this reason, Real-Time Operating Systems (**RTOS**) are used in the industry

OS Architecture

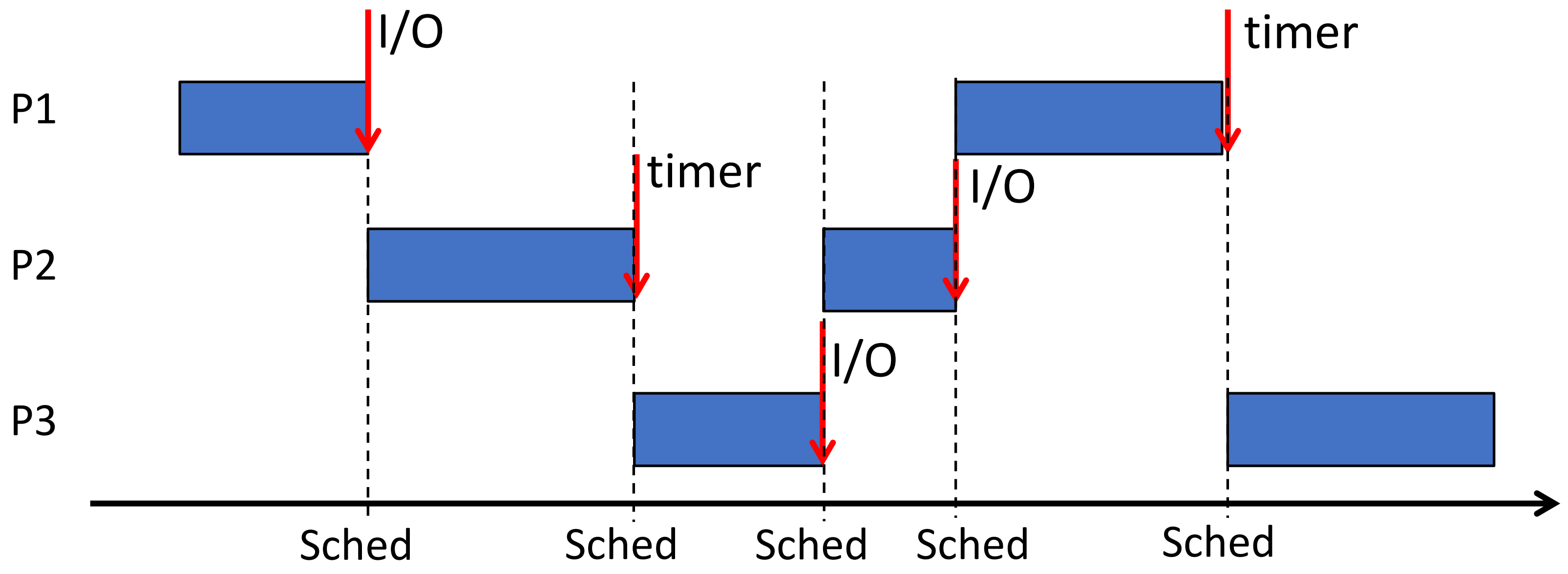


Kernel

- Portion of the OS that is resident in central memory
- Containing the most used functions
- It implements process **scheduling**:
  - Scheduling algorithms decide what process must be executed in multitasking enviroments: the problem is that the CPU must be shared (and thus assigned) to processes that are ready to execute
  - Traditional scheduling must be revisited to assure real-time performance

Multitasking

- When a process does an input/output operation (I/O) or when a time quantum expires (*timeslice*, in time sharing systems), the OS can assign the CPU to another process



Context Switch

- The set of operations exeucuted to preempt a process in favour of another process
- The OS needs to change the context, that is, the set of information stored in processor’s registers related to the execution of the process
  - Program Counter, Stack Pointer, Status Register, …
- The operation must be transparent to the interrupted process

- In an RTOS the context-switch must have a small, bounded duration

System Calls

- Request of a service to the OS
  - Start of a process, read of data from peripherials, communication between processes, …
- Usually implemented trough software interrupts, which are synchronous with the execution of the process
- They require the processor to change state:
  - From user mode (unprivledged) to kernel mode (priviledged)
- In order to run in isolation, system call code is often executed by temporary disabling interrupts → a serius threat to real time operations!

- In an RTOS, all system calls must be *interruptible* and have a known bounded duration

Memory Management

- The OS transparently manage memory pages and maps them to processes
- Memory area extensions are managed by proper system calls
- The image of a process can be partially resident in memory and partially stored in secondary storage (disk) → virtual memory

- In an RTOS, critical real-time tasks must be resident in memory
- Static memory partionining can be adopted to achieve better detereminism

Real Time Operating Systems

ERIKA ENTERPRISE

QNX Neutrino 2

Mentor Graphics VRTX

Embedded With RTEMS www.rtems.com

VxWorks® WIND RIVER

freeRTOS

ChibiOS EmbeddedWare

LynxOS FROM LYNUXWORKS

RTLinux

RTAI

Examples of RTOS

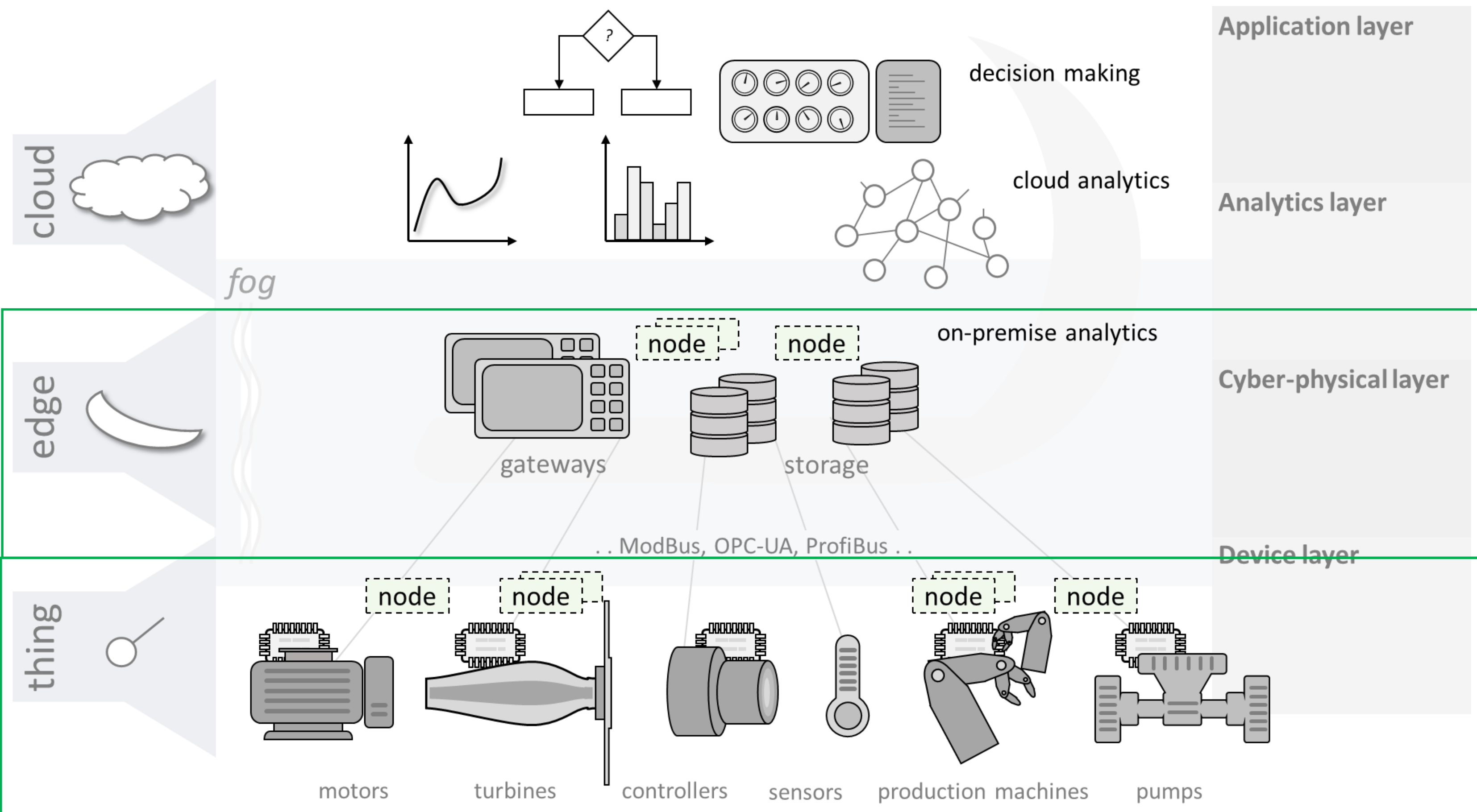
Commercial:

- VxWorks
- QNX
- OSE
- Erika enterprise
- PikeOS
- …

Open Source:

- Linux-derived:
  - Preempt-RT
  - Co-kernels
- FreeRTOS
- ChibiOS
- RTEMS
- …

RTOS mapping on the IIoT Architecture



Examples

Basic Solutions	Advanced Solutions
- Real-time threads in Linux	- RT-Linux on top of RT-XEN
- Bare metal SW - Rt-tasks on FreeRTOS	- Use of MPSOCs and multiple OSes

Standardization

- Standardization of a real-time operating system consists in the unambiguous definition of the APIs and/or kernel design.
- The standard is generally promoted by operating system manufacturers, in this case they’re called *de facto standards.*
    - For example, AUTOSAR; a recent standard promoted mainly by German vehicle manufacturers, is spreading in the automotive application domain

Standardization: benefits

- Why standardize an operating system?
  - One of the reasons is to help port applications on different operating systems. In practice the source code of the application is written only once because it invokes standardized primitives and in the ideal case it is only needed to recompile it to port it to another operating system.

POSIX

- The UNIX-based Portable Operating System Interface (POSIX) standard defined the programming interface for applications running on UNIX operating systems.
- Portability is therefore at the source code level.
- Real Time POSIX (RT-POSIX) is its extension for real-time systems.

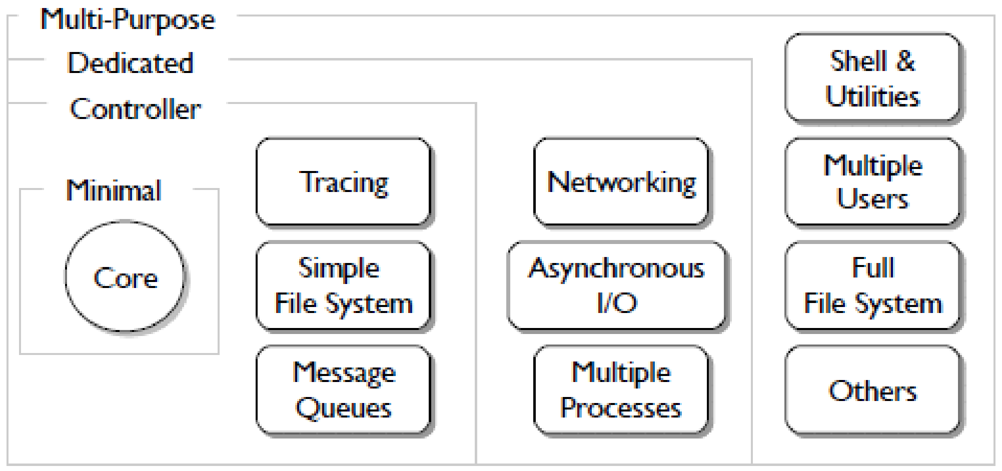
RT-POSIX

- RT-POSIX is the most popular standard for RT operating systems.
- The standard specifies the primitives for:
  - Concurrent programming
  - Mutual exclusion with priority inheritance
  - Synchronizing with condition variables
  - Priorized message queues for inter-task communication

- RT-POSIX also specifies services to ensure predictable time behavior of the operating system

RT-POSIX: Profiles (1/3)

- 4 "profiles" of the RT-POSIX standard are available:
  - Minimal
  - Controller
  - Dedicated
  - Multi-Purpose



RT-POSIX: profiles (2/3)

- **Minimal Real Time System (PSE51)**: this profile is dedicated to small embedded systems without a real file system, where I/O operations are possible through dedicated file devices. The minimum scheduled unit is the thread.
- **Real Time Controller profile (PSE52):** Compared to PSE51, there is a file system (simplified) for file I/O operations. It is a profile used for robots where a file system is needed.

RT-POSIX: profiles (3/3)

- **Dedicated Real Time System Profile (PSE53):** is used for larger embedded systems such as those used in avionics with support for scheduling multiple processes and memory protection.
- **Multi-purpose Real Time System Profile (PSE54):** is developed for general purpose operating systems that have both real-time and non-real time requirements.

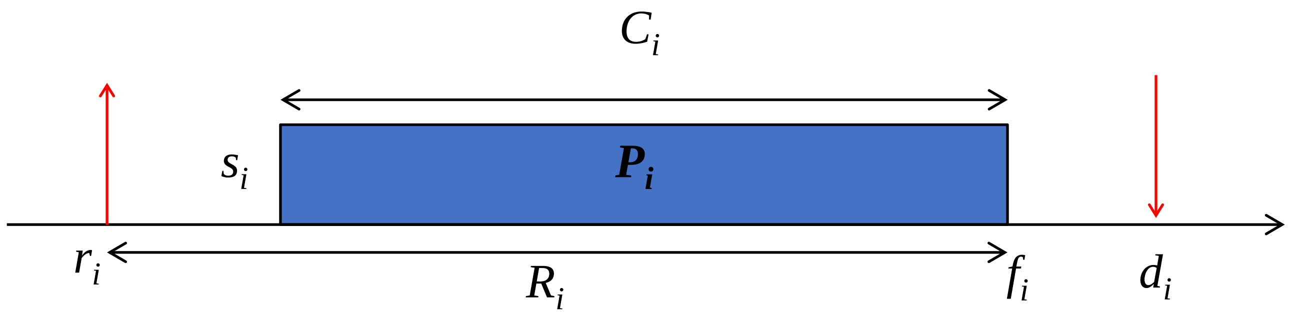
RT-POSIX: requirements

- An RTOS to be considered RT-POSIX compliant must meet requirements defined within the standard. For example,:
  - Execution scheduling: Provide support for real-time (static) priorities.
  - Performance requirements on system calls: Satisfy the worst case execution times required for many real-time operating services.
  - Priority levels: The number of priority levels must be at least 32.
  - Timers: Periodic and one shot timers must be supported.
  - Real-time files: A real-time file system can allocate space for files and should be able to store block files contiguously, allowing for a predictable delay in accessing files in virtual memory systems.
  - Memory locking: Predicting memory locking services is required to support deterministic in-memory access.
  - Multithreading support: Support real-time threading. Real-time threads are scheduleable entities of a real-time application and are subject to time constraints.

Tasks: basic definitions

Process or Task

- Sequence of instructions that, in the absence of other tasks, is executed by the processor continuously until it is completed



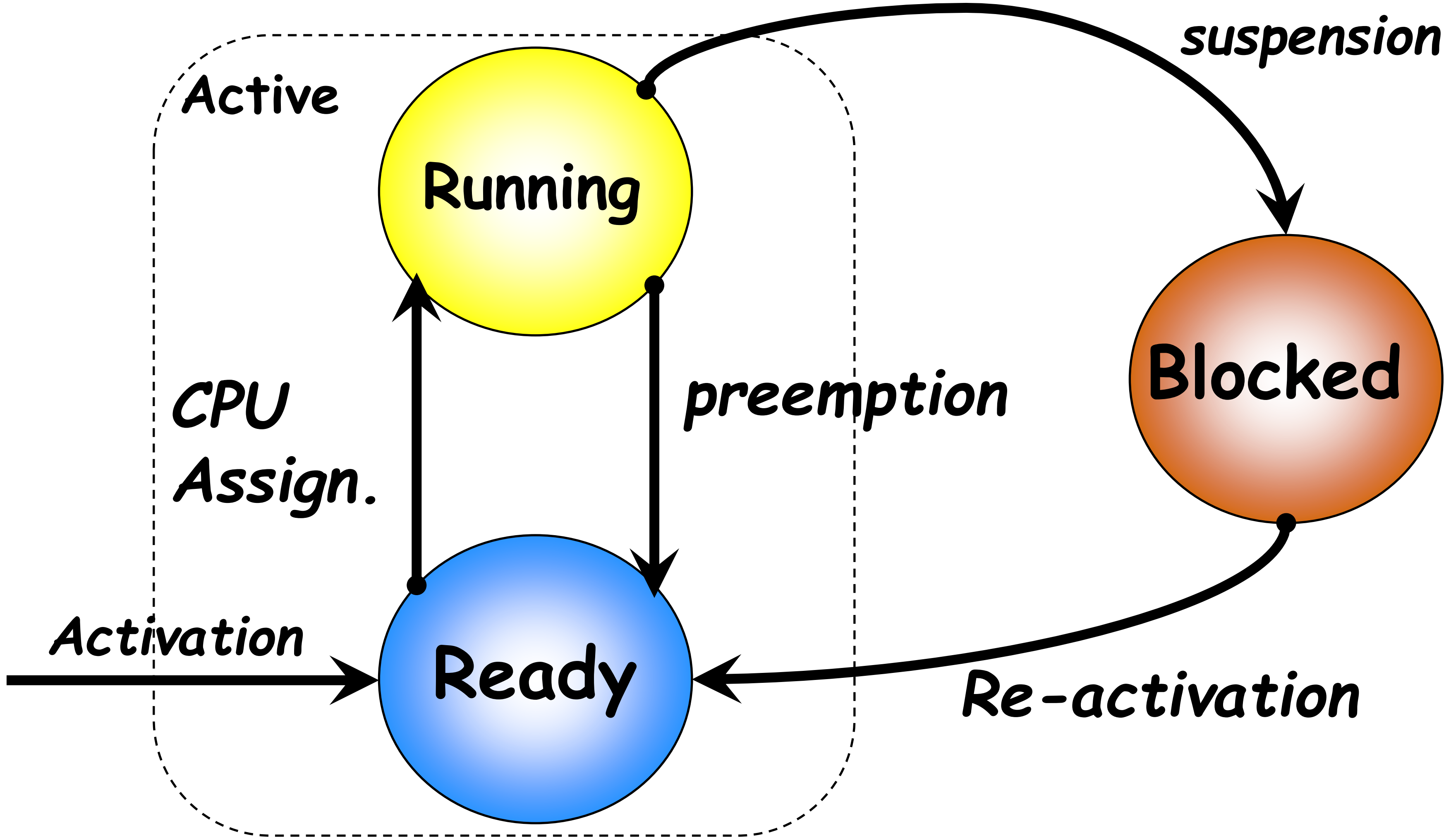
- r_i : release time (o arrival time a_i)
- s_i : start time
- C_i : computation time (or WCET - Worst Computation Execucution Time)
- f_i : finisihing time (or completion time)
- $R_i = f_i - r_i$: response time
- d_i : deadline

Relationship between task and program

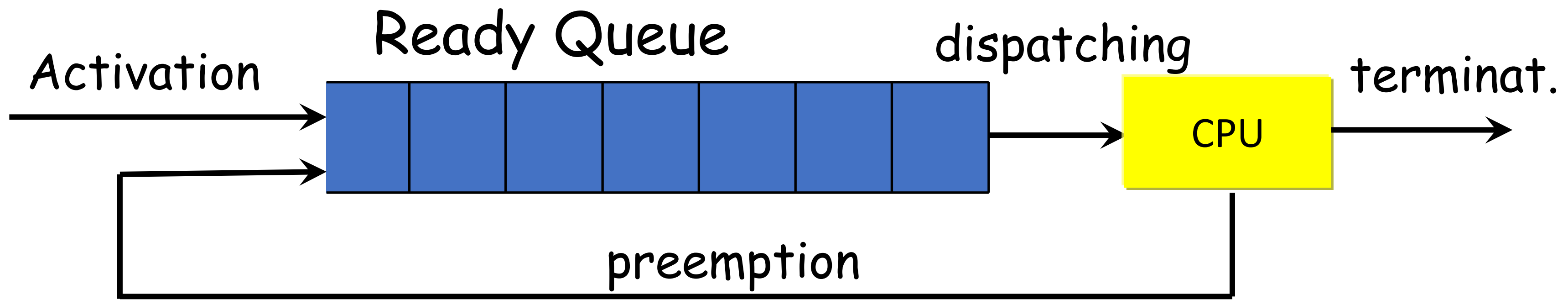
- **<u>Program</u>**
  - .. is the encoding of an algorithm in an appropriate programming language that makes it possible for a computer to run it;
  - is a static description of the processing to be performed.
- **<u>Task</u>**
  - ... is the base unit of execution of the OS that identifies the computer tasks related to a specific execution of a program;
  - A task is a running program
  - It's a dynamic entity (program + execution state).

- Different tasks can run multiple instances of the same program

Task states

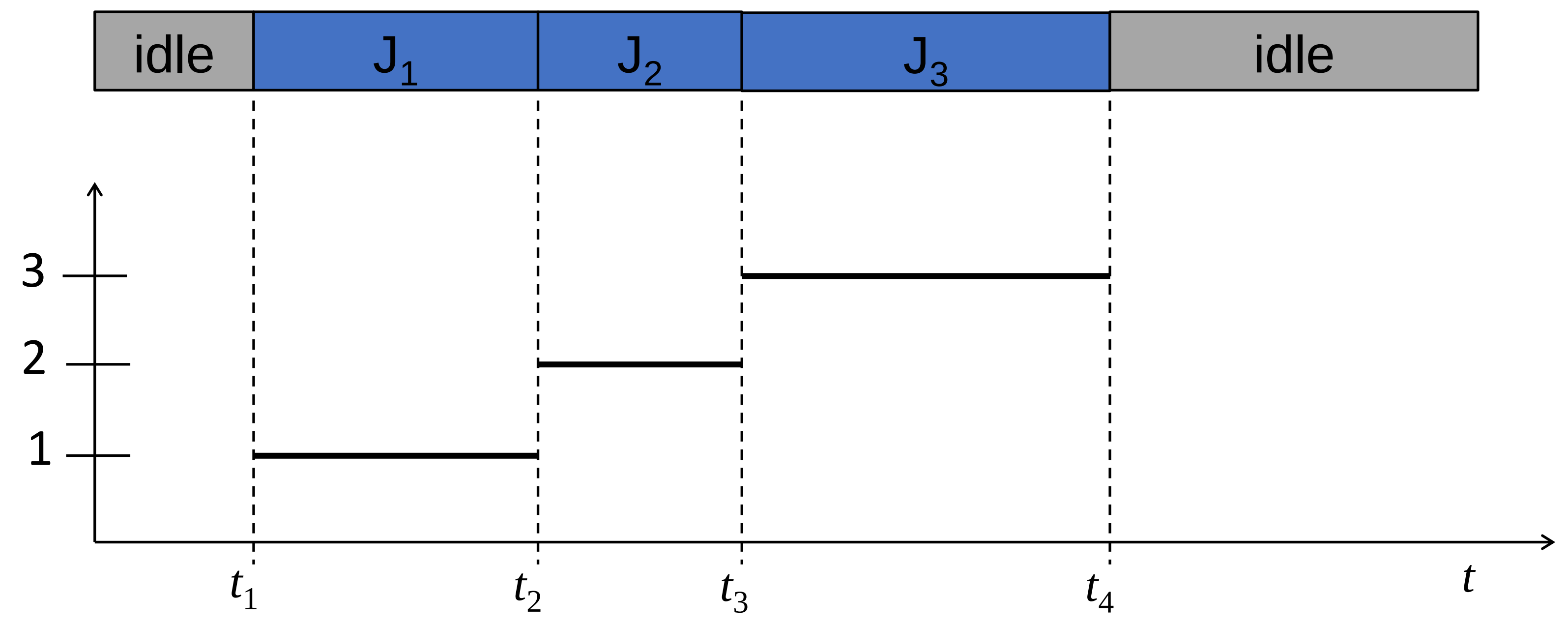

suspension
Active
Running
Blocked
CPU Assign.
preemption
Activation
Ready
Re-activation

Ready queue



- Ready jobs are kept in a queue called ready queue
- **scheduling algorithm:** a set of rules that determine the choice of the process to run
- **dispatching**: CPU assignment to the scheduled ready process, carried out by the OS after scheduling

Example of a schedule

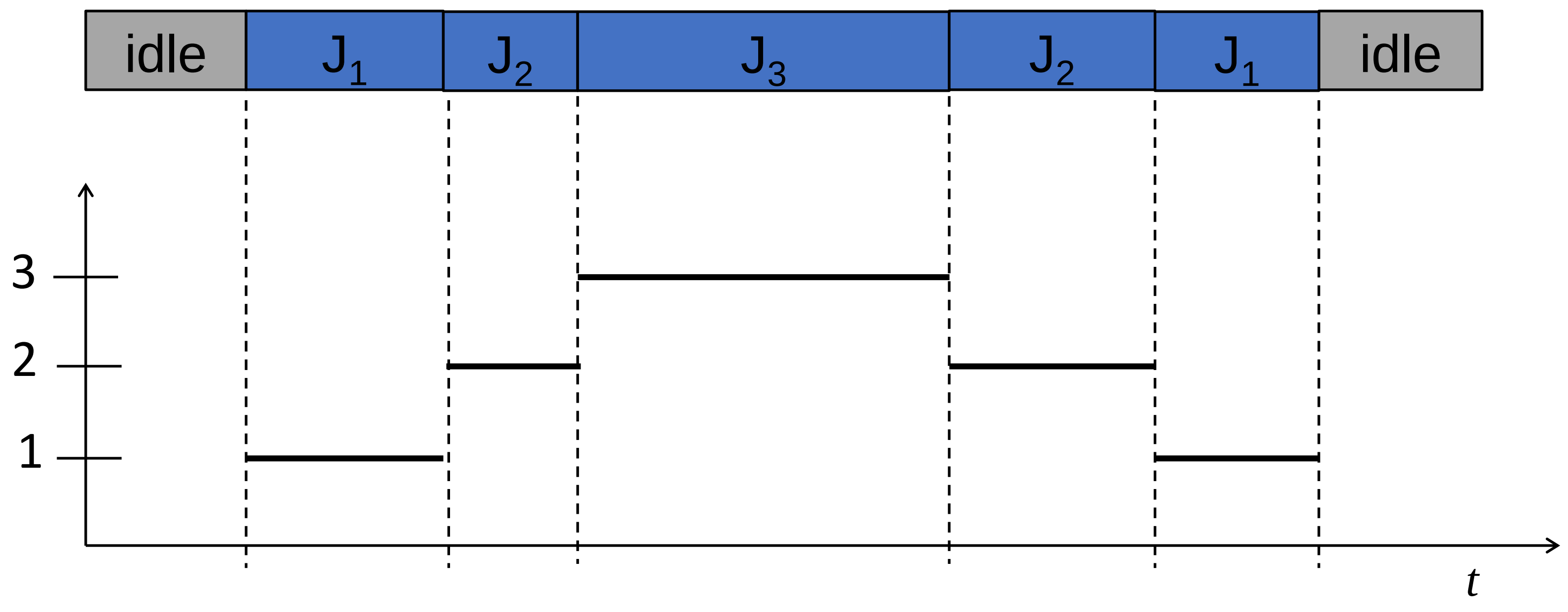


- At the times t_1, t_2, t_3 e t_4 a **context switch** takes place
- **time slice**: intervals $[t_i\, , t_{i+1})$ where $\sigma(t)$ it's constant

Schedule

- A schedule is said:
  - **Feasible**: If there is a task assignment to the process such that all tasks are completed in accordance with a set of constraints
    - A set of tasks is said to be schedulable if there is a feasible schedule for it
  - **Preemptive**: if the running process can be suspended at any time in order to assign the processor to another task in a set

Preemtive schedule example



Task Constraints

- The constraints that can be specified on tasks, and which a schedule must comply with in order to be feasible, are of three types:
  - Time constraints
    - activation, completion, …
  - Precedence constraints
    - Enforce an ordering between tasks in an execution
  - Resource constraints
    - Enforce synchronization in access to mutually exclusive resources

Activation mode

- **Time driven**: **<u>Periodic tasks</u>**

  infinite sequence of identical activities, performed on different data, each of which is named **instance** or **job**, activated on a regular basis with a constant frequency

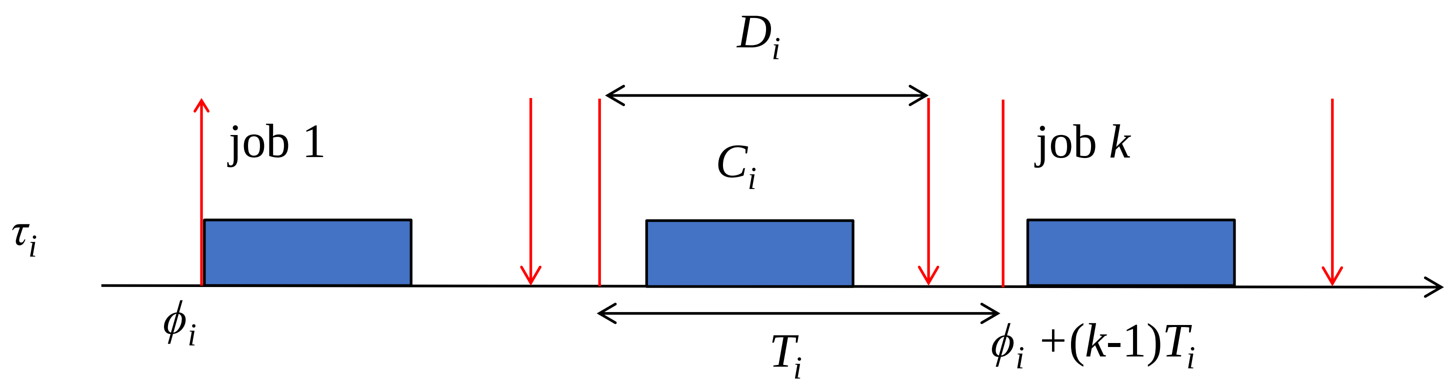
  - Control loops
  - Data acquisition (sampling and filtering)
  - …

- **Event driven**: **<u>Aperiodic tasks</u>**

  infinite sequence of identical activities, performed on different data, each of which characterized an arrival time, a calculation time, and a deadline

  - management of alarms
  - handling of events or input commands
  - …

Periodic tasks

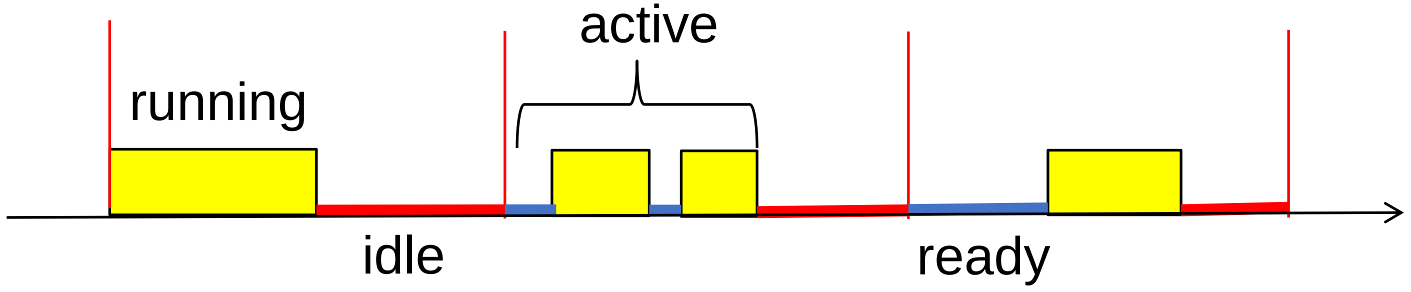


- ϕ_i : phase (activation time of the first instance)
- T_i : Period
- The parameters T_i, C_i e D_i are constant (often $T_i = D_i$)
- Jitter: time variation of periodic events
  - E.g. start time jitter: $max(s_{i,k} - r_{i,k}) - min(s_{i,k} - r_{i,k})$

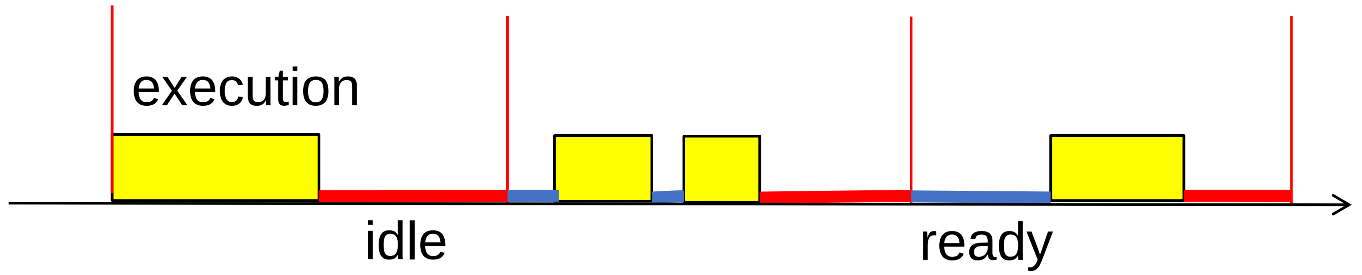
Periodic tasks

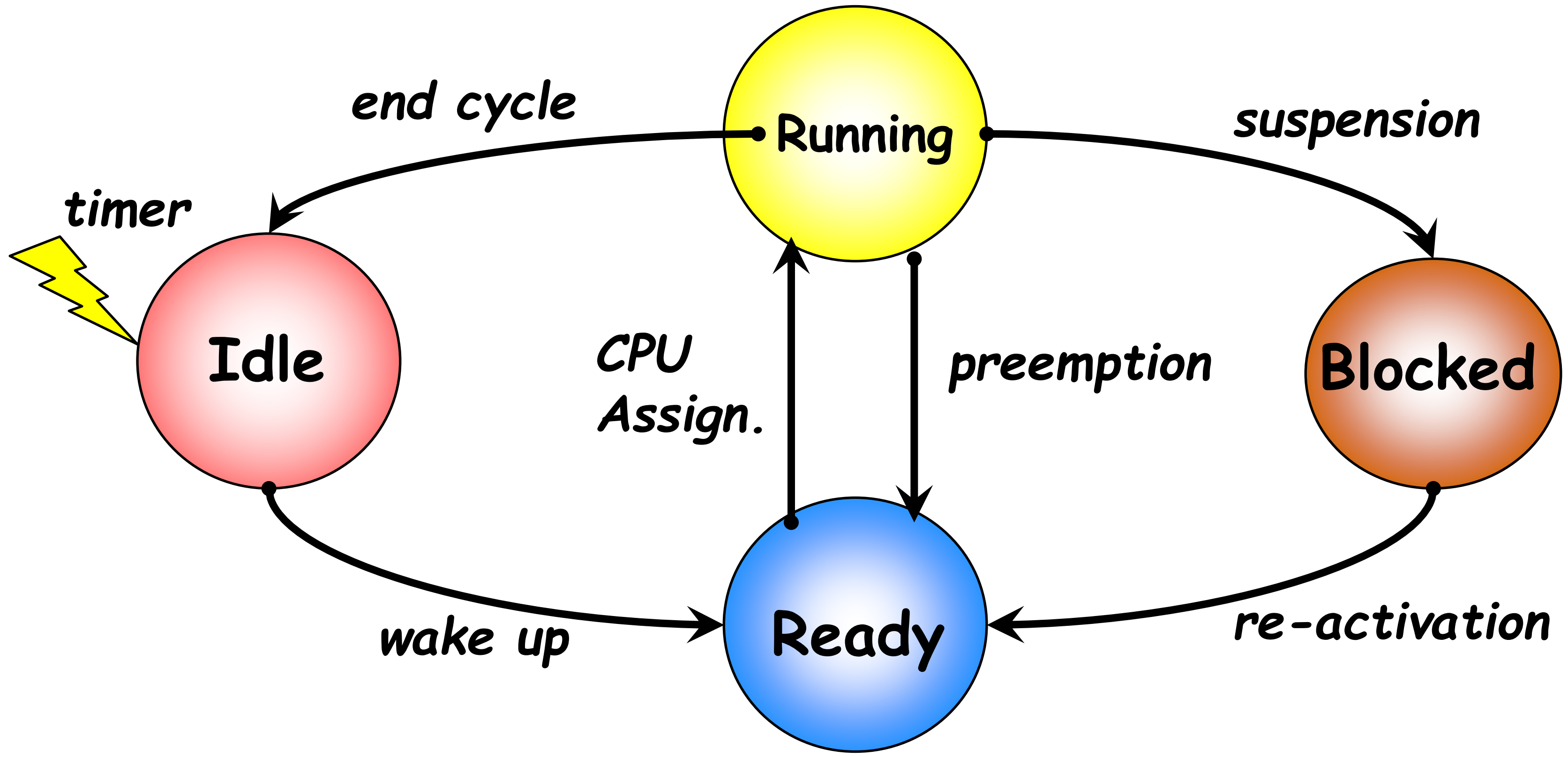
- They require specific primitives and ad-hoc mechanisms of the operating system to manage them, for instance:

```
while (condition) {
          ...
          ...
          end_cycle();
}
```



The idle state





Scheduling of periodic real-time tasks

Classification of scheduling algorithms

Uniprocessor	$\longleftrightarrow$ Multiprocessor
Preemptive	$\longleftrightarrow$ Non Preemptive
Static	$\longleftrightarrow$ Dynamic
Off-line	$\longleftrightarrow$ On-line
Optimal	$\longleftrightarrow$ Heuristic

Optimality

- A scheduling algorithm is said to be optimal if it can minimize a cost function defined on tasks

- In the sense of schedulability, an algorithm $A*$ it’s optimal if it's able to generate a feasible schedule

- If for a set of tasks J there is a feasible schedule, then definitely J is schedulable with $A*$
- If J is not schedulable with $A*$, then it’s schedulable with no other algorithm (of the same kind of $A*$)

Timeline Scheduling (Cyclic executive)

- Used for years in critical systems, such as military defense systems, automatic navigation, monitoring systems, …
- Examples:
  - Space Shuttle
  - Boeing 777
  - Airbus
  - Air Traffic Control (ATC)

Timeline Scheduling: method

- The time axis is divided into equal time intervals
    - ***time slots*** or ***slices***
- In each slot, one or more tasks are allocated, in order to respect their activation frequencies
- A timer synchronizes the activation of procedures at the beginning of each slot

Timeline Scheduling: example

	τ_A	τ_B	τ_C
fr (Hz)	40	20	10
$T_i(ms)$	25	50	100

$\longrightarrow$

Optimal length time slots : 25 ms (MCD between periods)

Massimo comune divisore

- τ_A will have to run in each slot
- τ_B must run every 2 slots
- τ_C must run every 4 slots

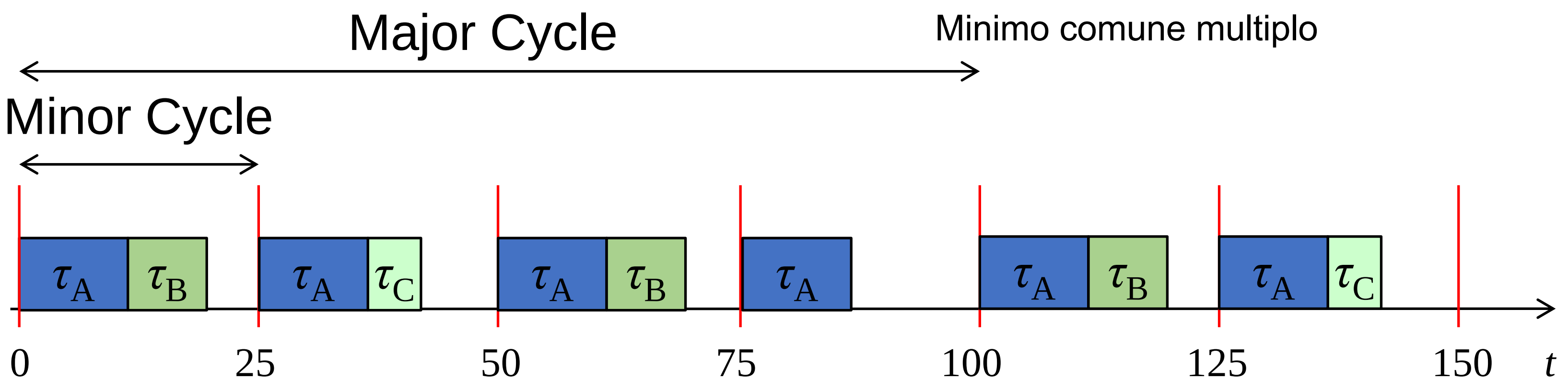
Timeline Scheduling: example

	τ_A	τ_B	τ_C
fr (Hz)	40	20	10
$T_i(ms)$	25	50	100

→ Time slots = Minor Cycle = 25 ms (MCD between periods)

Major Cycle = 100 ms (mcm among periods)

Minimo comune multiplo



Timeline Scheduling: implementation

```
#define MAJ_C 4
int minor_cycle = 0;
void Timer_ISR()
{
      minor_cycle = (minor_cycle++) % MAJ_C;
      switch(minor_cycle) {
            case 1: taskA(); taskB(); break;
            case 2: taskA(); taskC(); break;
            case 3: taskA(); taskB(); break;
            case 0: taskA(); break;
      }
}
```

Configured to be invoked every 25 ms

Timeline Scheduling

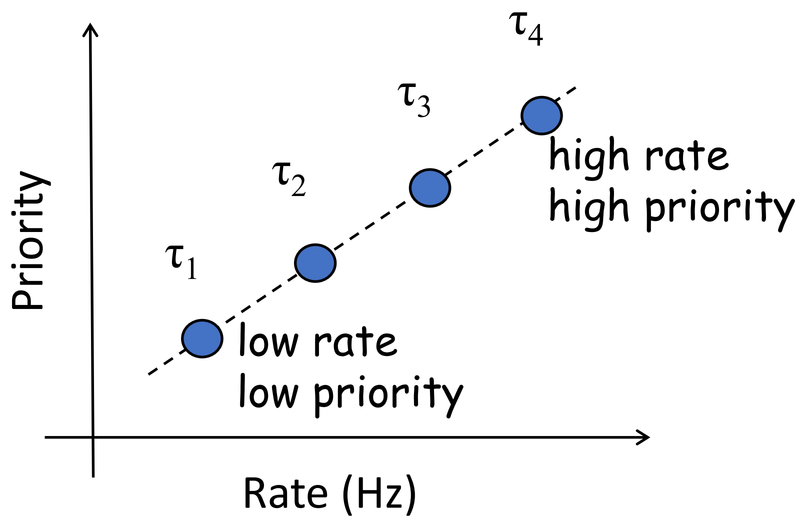
- ✔ Simplicity (no RTOS needed)
- ✔ Extremely low overhead (no context switch)
- ✔ Deterministic and non-jitter activation sequence

- ✗ Fragility in overloads
- ✗ Poor flexibility
- ✗ Difficulty to manage aperiodic tasks efficiently

Rate Monotonic (RM)

- It's a priority-driven algorithm:
  - Gives to each task a priority directly proportional to its request frequency
  - Tasks with shorter periods get a higher priority

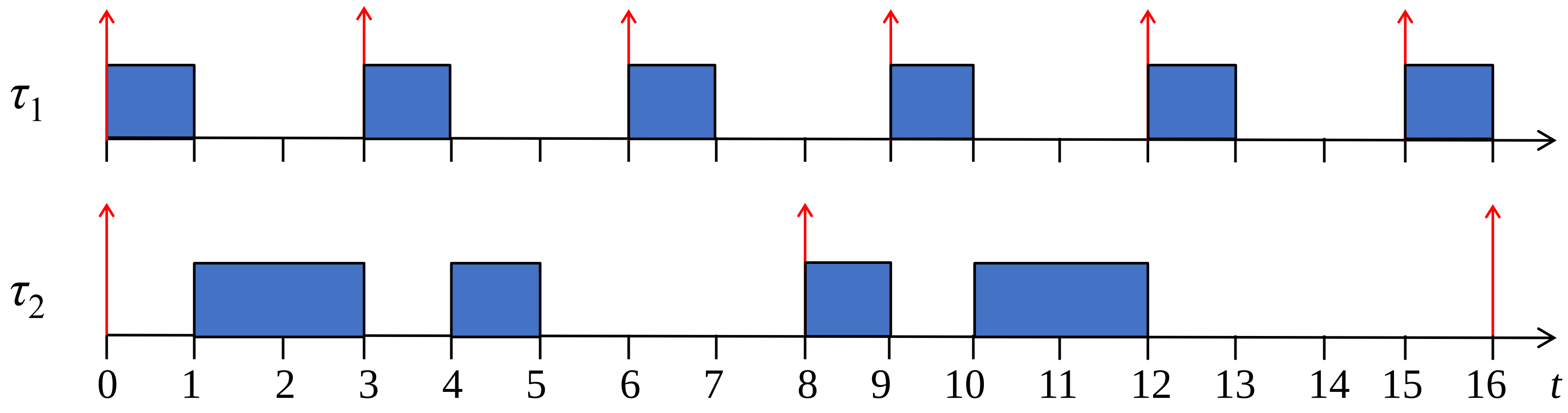
- It's online, preemptive, static (fixed priority)



RM: example

	τ_1	τ_2
C_i	1	3
T_i	3	8

1/3 + 1/8 = 0,7 < 2(rad(2)-1)=0,83



RM: optimality

- Liu and Layland have shown that RM is optimal
  - if a task set is not feasible with RM, then the set is not feasible with any other fixed-priority algorithm

Utilization factor

- Given a set Γ of n periodic tasks, the utilization factor U of the processor is the fraction of time used by the CPU to execute Γ

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

Ci caso peggiore durata esecuzione / Ti periodo

Schedulability theorem for RM

- A set of periodic tasks can be scheduled with Rate Monotonic if:

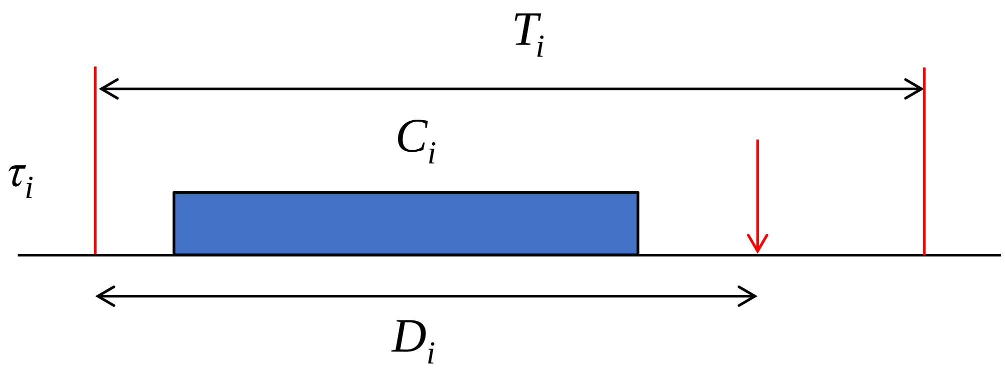
$$\boxed{\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)}$$

this is called Least Upper Bound

- This (sufficient) condition, tested by Liu and Layland, constitutes a feasibility test for RM

Deadline Monotonic

- A variation of RM where relative deadlines D_i can be shorter than periods T_i



- The classical Liu and Layland test can be applied replacing periods with deadlines in the utilization factor U

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{1/n} - 1)$$

- However, this test is pessimistic as it overestimates the real CPU load

Response Time Analysis

- Necessary and sufficient method for schedulability analysis of fixed-priority algorithms introduced by Joseph and Pandya:

per eccesso

Questo lo dobbiamo calcolare per ogni task e controllare che sia minore uguale al suo Deadline

$$R_i = C_i + \underbrace{\sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k}$$

k=1 task a priorità maggiore

Response Time of task *i*

WCET of task *i*

Interference due to tasks with higher priority than task *i*

tutte le volte che il task viene preemptivato da altri task a priotià maggiore

Dato che Ri si trova si a destra che a sinistra la possiamo risolvere con un processo iterativo come l'esempio

Response Time Analysis

- The computation of R_i can be done iteratively

```
DM_guarantee ($\Gamma$) {
    for each $\tau_i$ in $\Gamma$ {
        $R = \Sigma_{k=1} C_k$;
        do {
            $R\_old = R$;
            $R = C_i + \Sigma_{k=1}\text{ceil}(R\_old/T_k)C_k$;
            if ($R > D_i$) return (UNSCHEDULABLE);
        } while ($R > R\_old$);
    }
    return (SCHEDULABLE);
}
```

c'é convergenza quando la R nuova è uguale a quella vecchia, sarà schedulabile

Earliest Deadline First – EDF

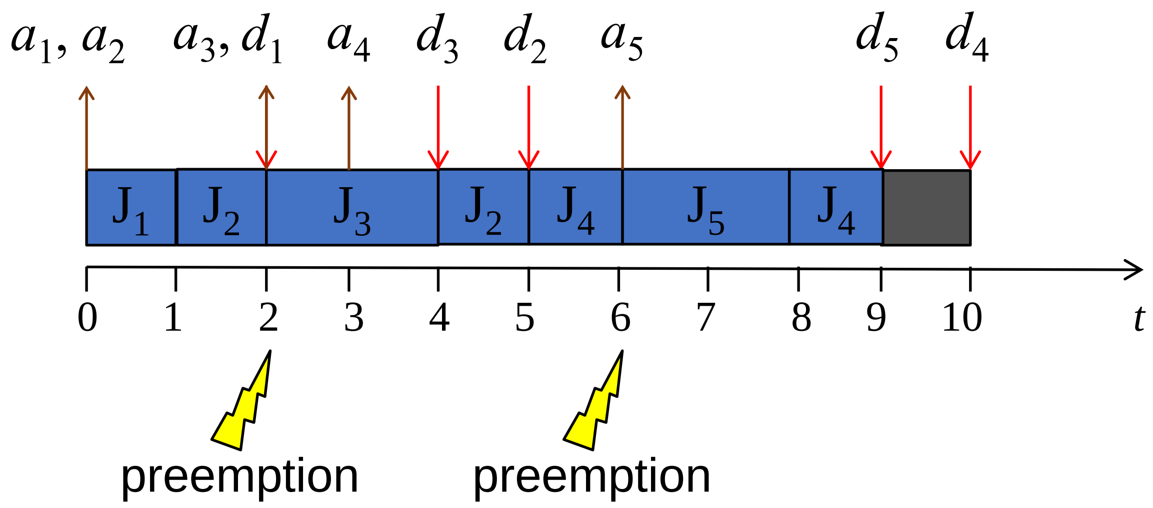
- Dynamic algorithm

- Each time a task enters the system, the ready queue is reordered for increasing deadline, and the CPU is assigned to the task with the earliest deadline

- Dynamic algorithm, online
- Complexity is $O(n^2)$

EDF: example

	J_1	J_2	J_3	J_4	J_5
a_i	0	0	2	3	6
C_i	1	2	2	2	2
d_i	2	5	4	10	9



Optimality and schedulability

- EDF is optimal among all scheduling algorithms

A differenza di Rate Monotonic che era ottimale solo per gli algoritmi di tipo fixed

Con questo algoritmo miglioro la lateness

- **Schedulability theorem for EDF**: A set of periodic tasks can be scheduled with EDF if and only if:

$$\boxed{\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1}$$

Non devo saturare la CPU per essere schedulabili

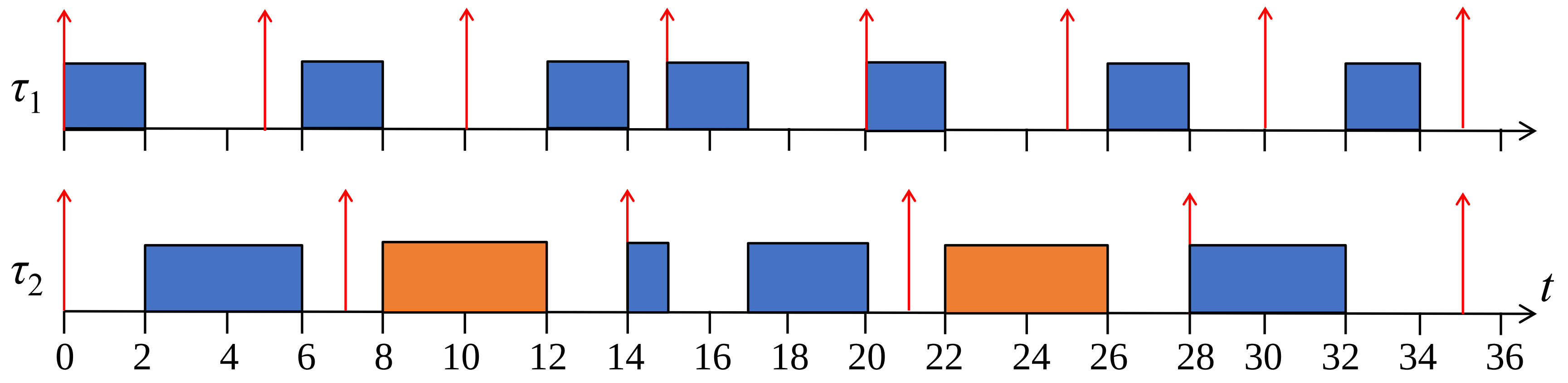
EDF: example

	τ_1	τ_2
C_i	2	4
T_i	5	7

$U = 2/5 + 4/7 = 34/35 \cong 0.97$

- τ_2 suffers a single preemption in favour of τ_1

EDF scheduling



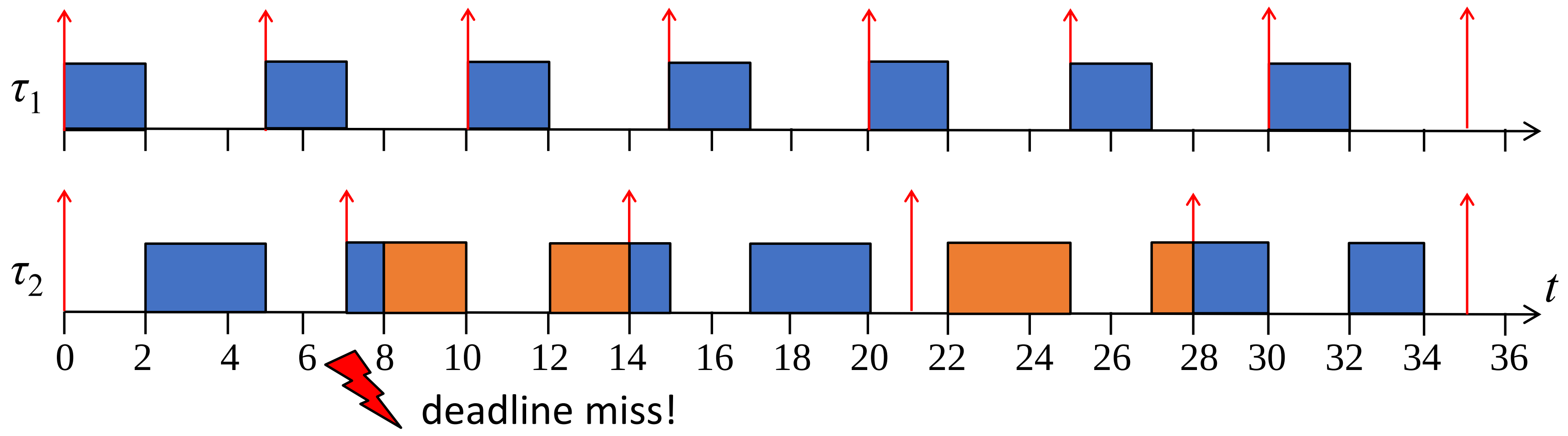
EDF: example (use of RM)

	τ_1	τ_2
C_i	2	4
T_i	5	7

$U = 2/5 + 4/7 = 34/35 \cong 0.97$

- Schedulability is not guaranteed
- τ_2 suffers 5 preemptions in favor of τ_1!!

RM scheduling



EDF vs. RM

- Using a dynamic priority algorithm has several advantages over a fixed-priority algorithm
  - EDF reduces the number of context switches
  - EDF is able to take advantage of the full computing power of the processor
  - EDF enables more efficient management of aperiodic activities
  - In case of overload, RM tends to penalize long-term tasks, while EDF is fairer

EDF vs. RM

- Both algorithms can be easily implemented above a priority-based kernel
  - For RM, it is sufficient to prioritize tasks based on their period
  - Once the priorities are set, the schedule is made by the operating system scheduler
  - Instead, EDF requires explicit time support for managing absolute deadlines
    - When an instance becomes active, it must be queued ordered according to the deadlines of all active instances
    - In a priority kernel, this is equivalent to changing the priorities of active tasks
    - This operation must be repeated when each instance is activated → More overhead

EDF non è predicibile a differenza di RTM, che comunque resta lo standard: il suo codice è più semplice, esecuzione predicibile, testabile e verificabile