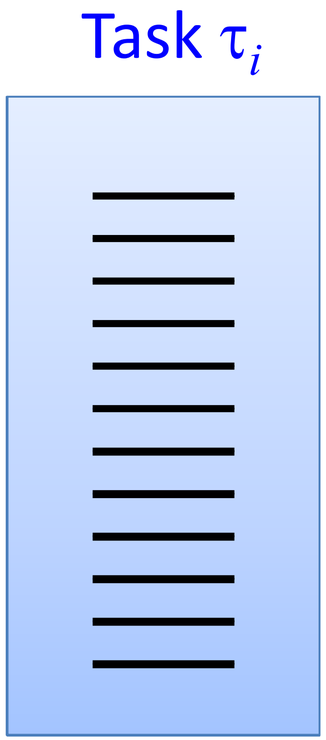
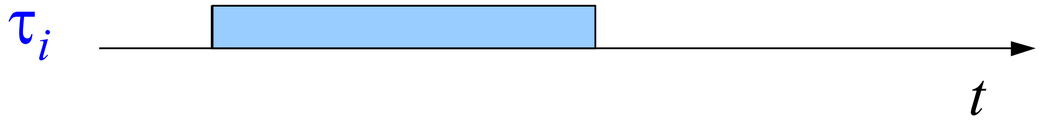
# Task Scheduling

# Definition: Task

A **task** (or **thread**) is a sequence of instructions that in the absence of other activities is continuously executed by the processor until completion.

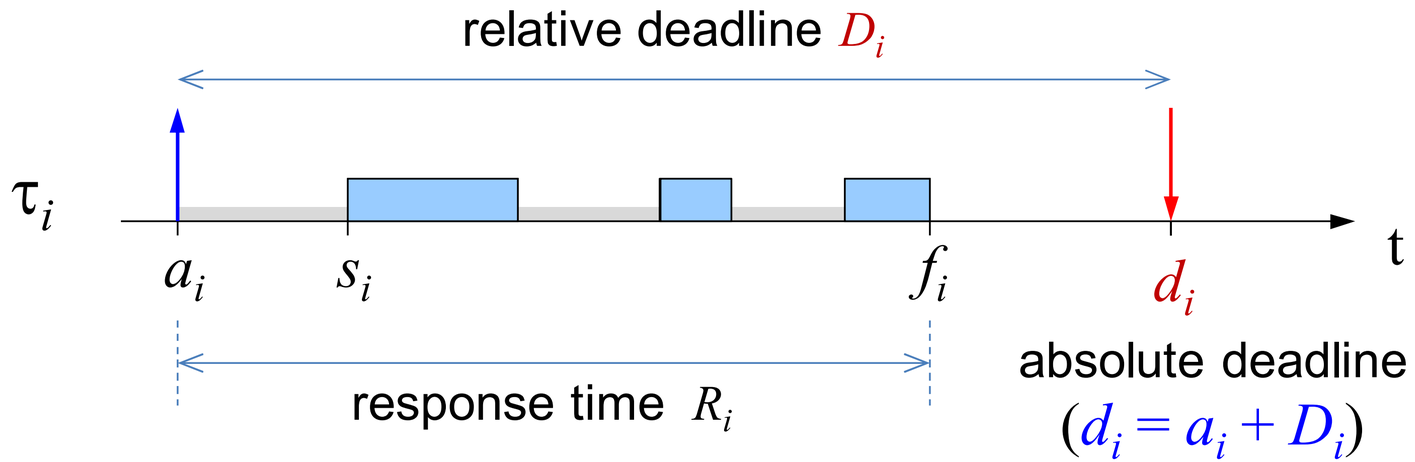



What are the important variables that characterize a computation if we want to perform a timing analysis?

# Definition: Real-Time Task

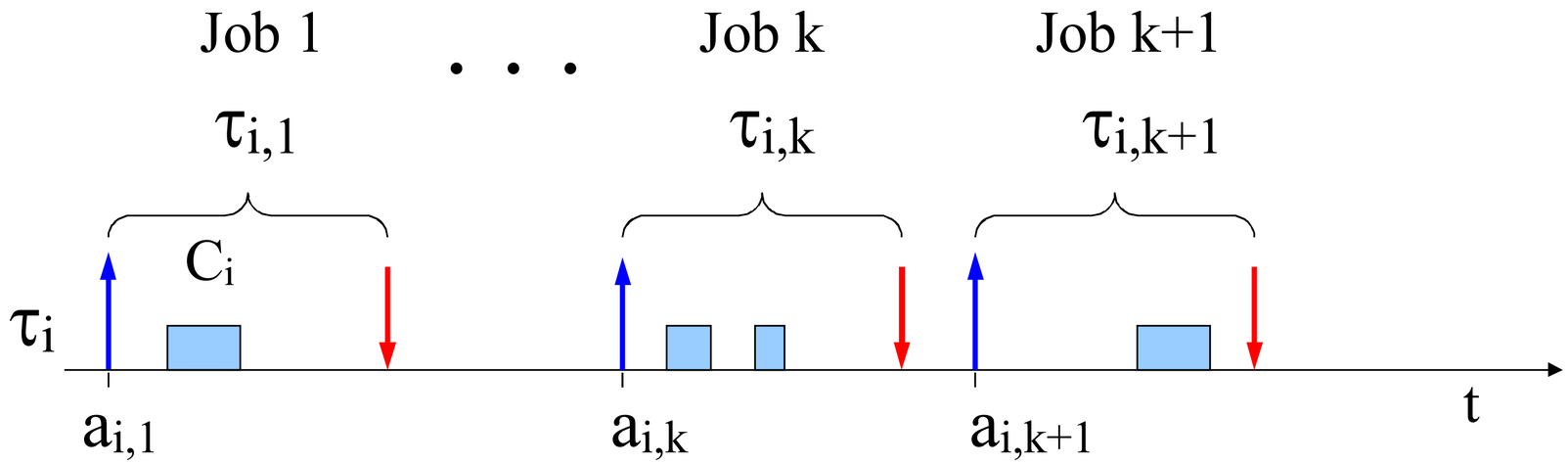
It is a task with a timing constraint on its response time, called deadline:



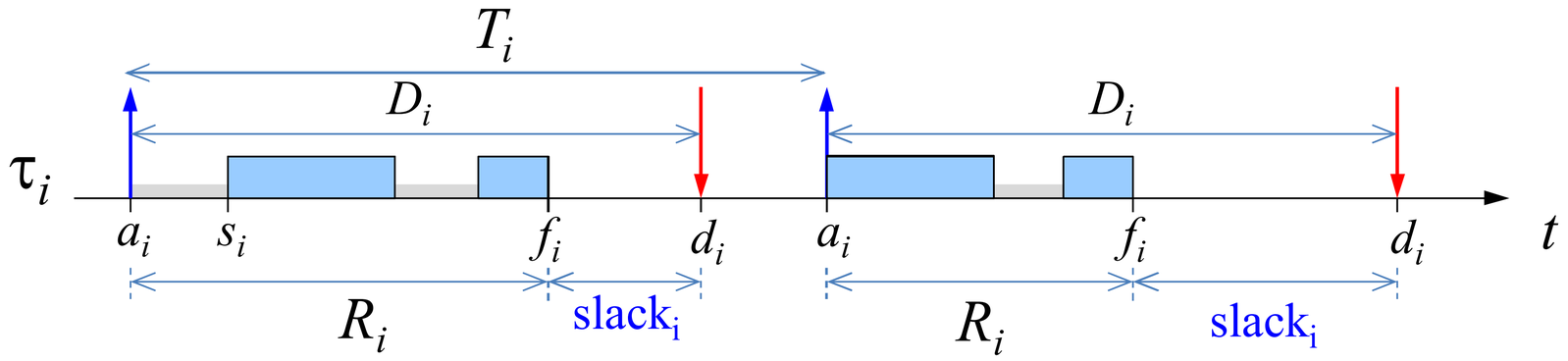

A real-time task $\tau_i$ is said to be feasible if it is guaranteed to complete within its deadline, that is, if $f_i \leq d_i$ (or $R_i \leq D_i$).

# Tasks and jobs

A task running several times on different input data generates a sequence of instances (or jobs):



# Parameters summary




- Computation time ($C_i$)
- Period ($T_i$)
- Relative deadline ($D_i$)

These parameters are specified by the programmer and are known off-line

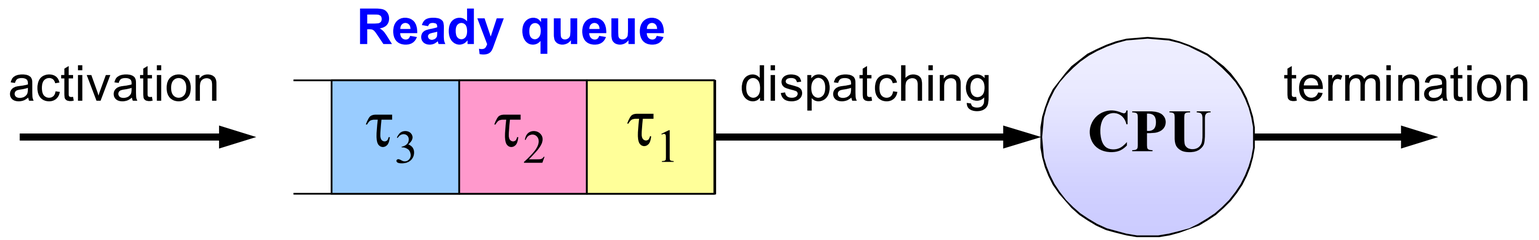
- Arrival time ($a_i$)
- Start time ($s_i$)
- Finishing time ($f_i$)
- Response time ($R_i$)
- Slack and Lateness
- Jitter

These parameters depend on the scheduler and on the actual execution, and are known at run time.

# Ready queue

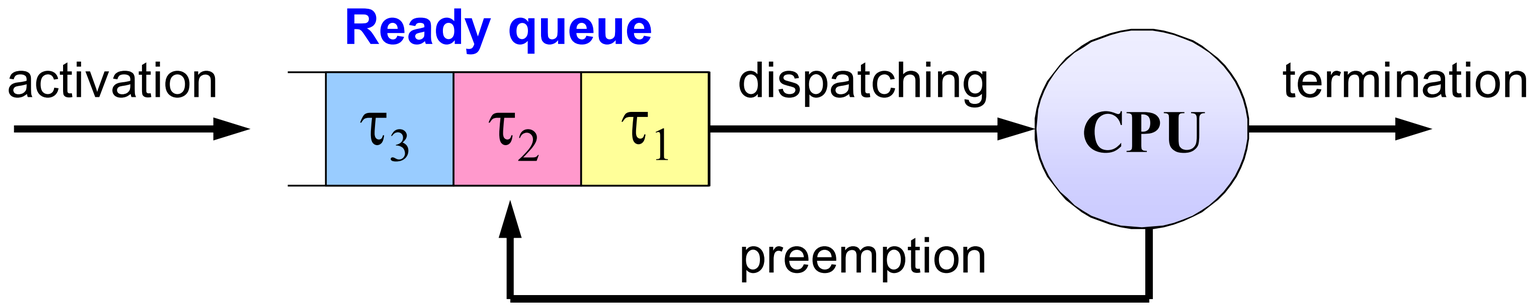
In a concurrent system, more tasks can be simultaneously active, but only one can be in execution (running).

- An active task that is not in execution is said to be ready.
- Ready tasks are kept in a ready queue, managed by a scheduling policy.
- The processor is assigned to the first task in the queue through a dispatching operation.



# Preemption

It is a kernel mechanism that allows to suspend the execution of the running task in favor of a more important task. The suspended task goes back in the ready queue.




- Preemption enhances concurrency and allows reducing the response times of high priority tasks.
- It can be disabled (completely or temporarily) to ensure the consistency of certain critical operations.

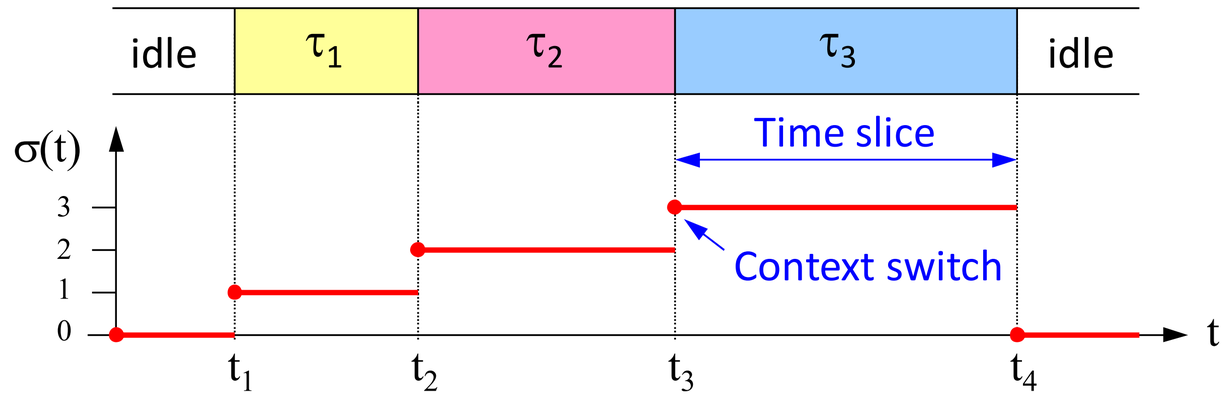
# Schedule

A schedule is a specific allocation of tasks to the processor, which determines the corresponding execution sequence.

Formally, given a task set $\Gamma = \{\tau_1, \ldots, \tau_n\}$, a schedule is a function $\sigma: R^+ \rightarrow N$ that associates an integer $k$ to each interval of time $[t, t+1)$ with the following meaning:

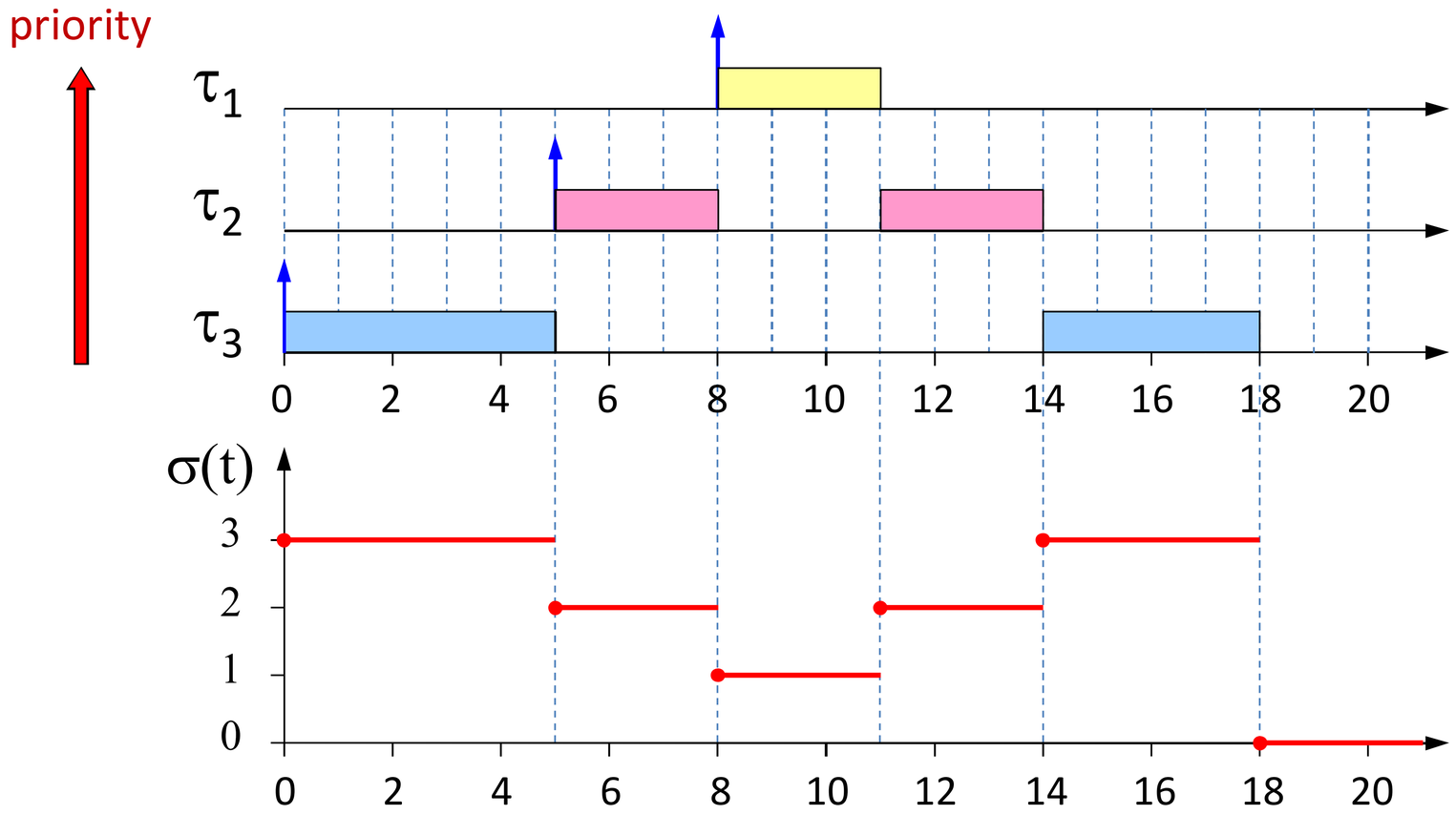
$k = 0$ ⟹ in $[t, t+1)$ the processor is IDLE

$k > 0$ ⟹ in $[t, t+1)$ the processor executes $\tau_k$

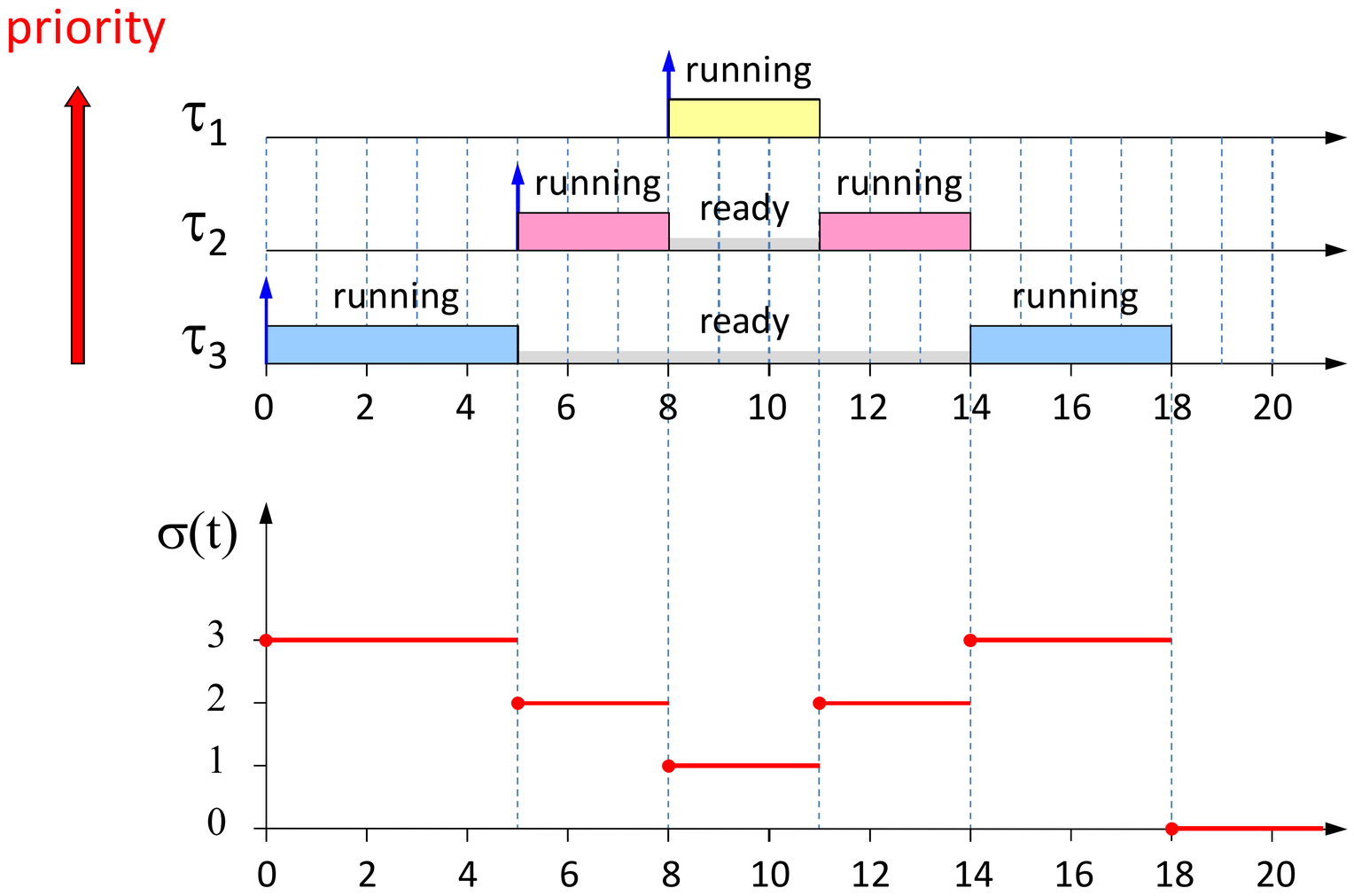


# Preemptive schedule

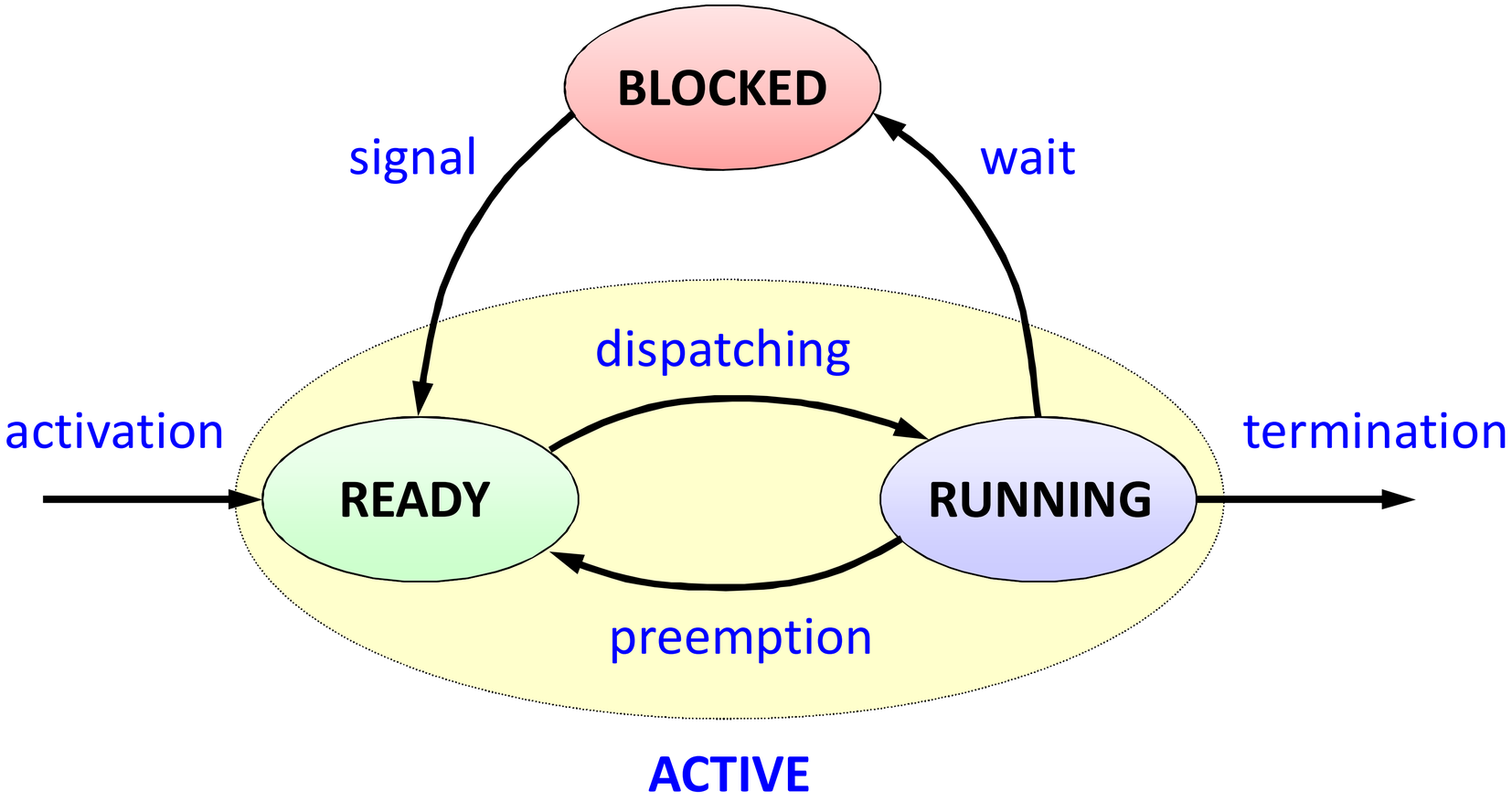
A schedule is said to be preemptive if a task can be interrupted at any time in favor of another task and then resumed later:



# Task states


priority
τ1
τ2
τ3
running
running
ready
running
running
ready
running
0
2
4
6
8
10
12
14
16
18
20
σ(t)
3
2
1
0
0
2
4
6
8
10
12
14
16
18
20

# Task states


BLOCKED
signal
wait
dispatching
activation
termination
READY
RUNNING
preemption
ACTIVE

# Activation modes

- **Periodic** (time-driven activation)

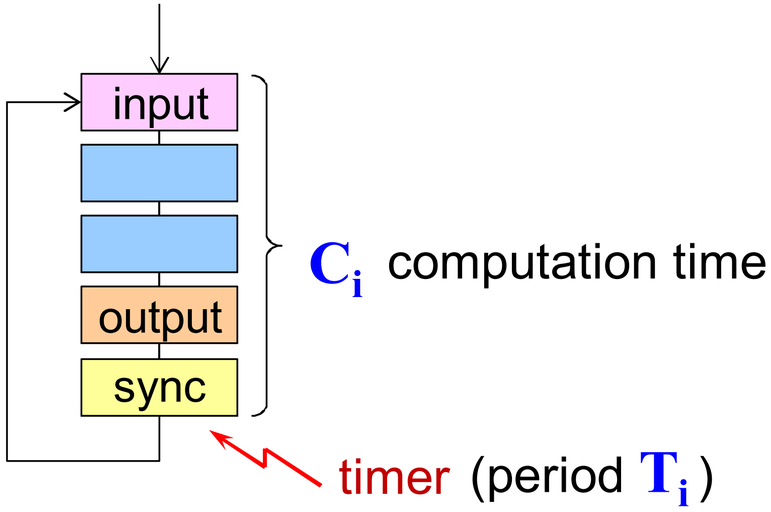
  A task is said to be periodic if its jobs are automatically activated by the operating system at predefined time instants.

- **Aperiodic** (event-driven activation)

  A task is said to be aperiodic if its jobs are activated at the arrival of an event (by interrupt or by another task through an explicit system call).

  If the activation interval between consecutive jobs cannot be smaller than a given quantity, the task is said to be **sporadic**.

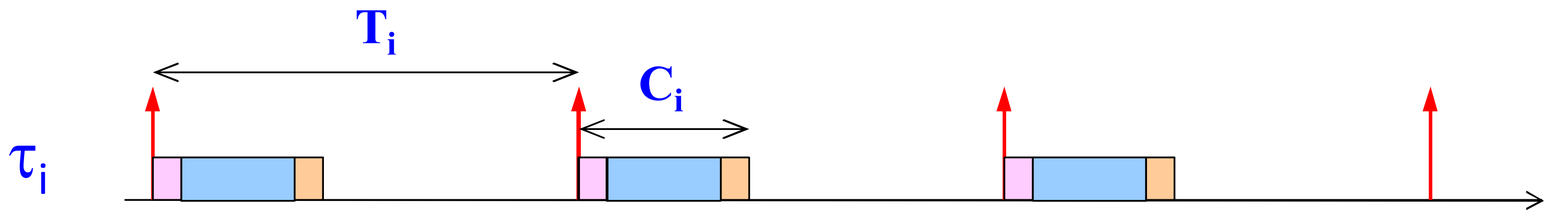
# Periodic task




$$U_i = \frac{C_i}{T_i}$$

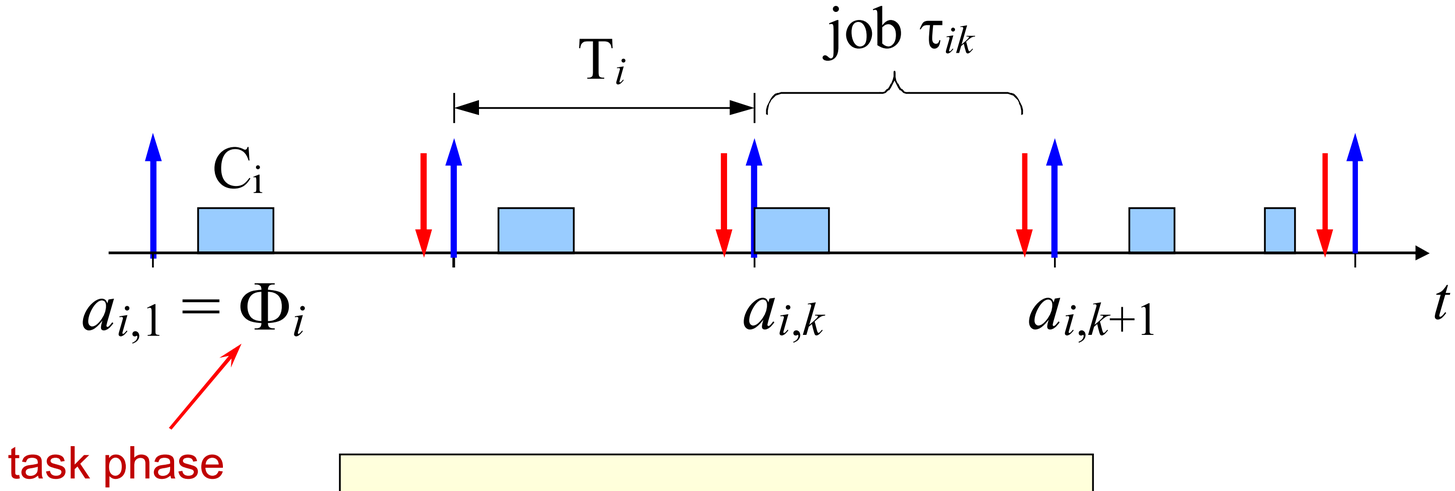
utilization factor

- A periodic task $\tau_i$ generates an infinite sequence of jobs: $\tau_{i1}$, $\tau_{i2}$, …, $\tau_{ik}$ (same code on different data):



# Periodic task

A periodic task can be fully described by four parameters only: phase ($\Phi_i$), worst-case computation time ($C_i$), period ($T_i$), and relative deadline ($D_i$).



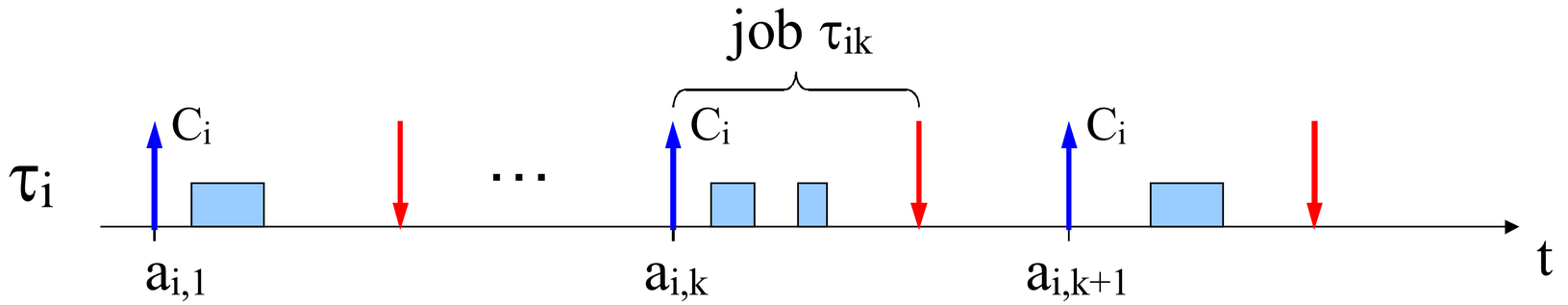

$$a_{i,k} = \Phi_i + (k-1)\,T_i$$

$$d_{i,k} = a_{i,k} + D_i$$

# Aperiodic task

- **Aperiodic:** $a_{i,k+1} > a_{i,k}$
- **Sporadic:** $a_{i,k+1} \geq a_{i,k} + T_i$

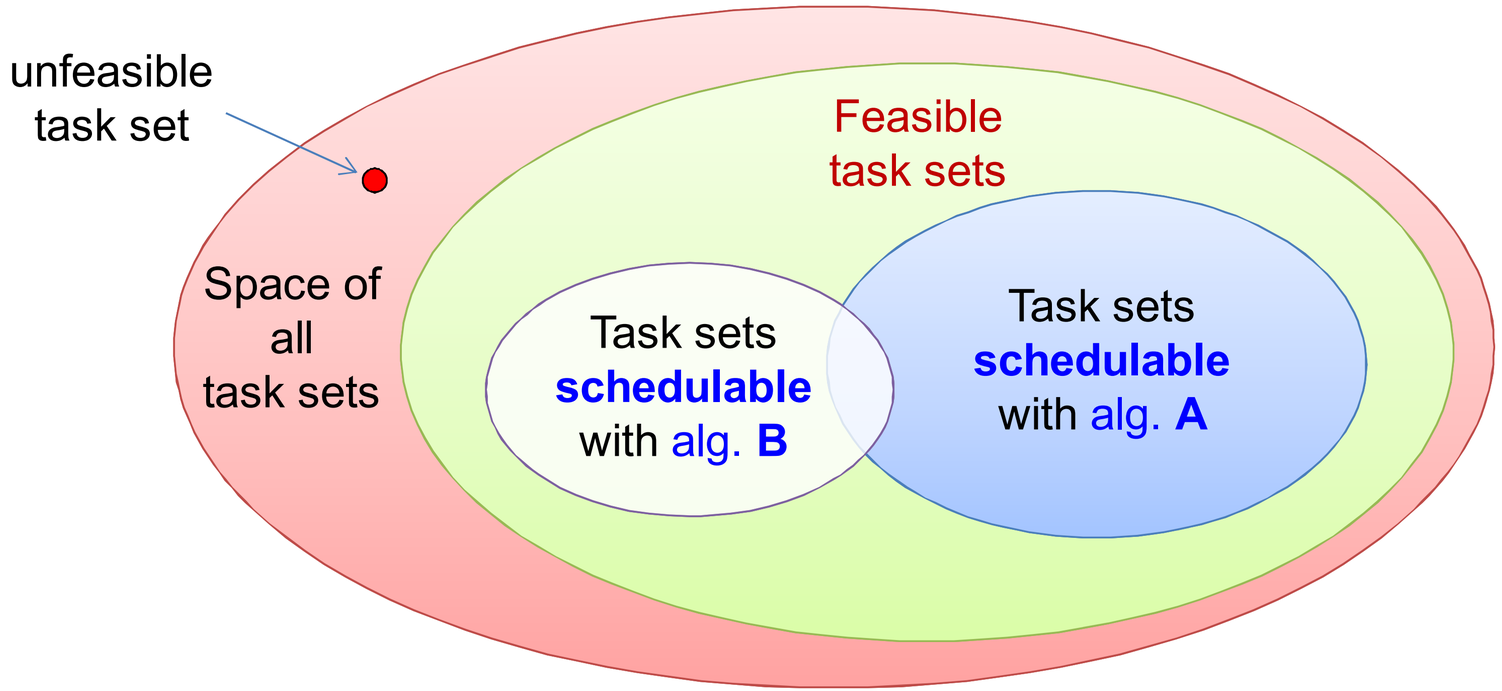
minimum interarrival time



# Definitions

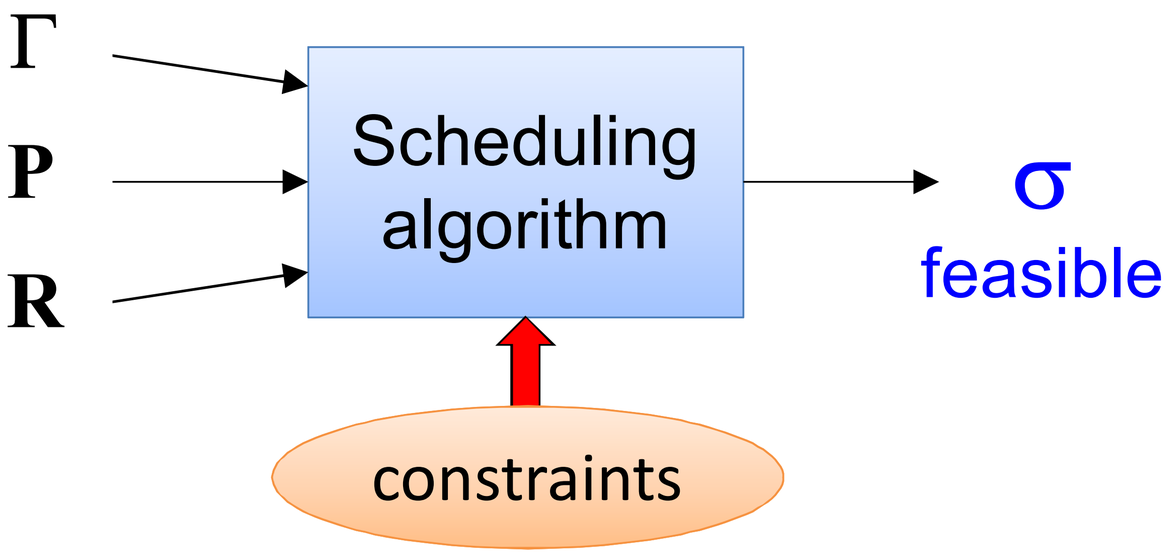
A schedule $\sigma$ is said to be **feasible** if it satisfies a set of constraints.

A task set $\Gamma$ is said to be **schedulable** with an algorithm A, if A generates a feasible schedule.



# The scheduling problem

Given a set $\Gamma$ of $n$ tasks, a set **P** of $p$ processors, and a set **R** of $r$ resources, find an assignment of **P** and **R** to $\Gamma$ that produces a feasible schedule under a set of constraints.



# Complexity

- In 1975, Garey and Johnson showed that the general scheduling problem is **NP hard**.

  In practice, it means that the time for finding a feasible schedule grows exponentially with the number of tasks.

Fortunately, polynomial time algorithms can be found under particular conditions.

# Why do we care about complexity?

- Let's consider an application with ***n* = 30** tasks on a processor in which the elementary step takes **1 μs**
- Consider 3 algorithms with the following complexity:

| $A_1$: ***O(n)*** | $A_2$: ***O($n^8$)*** | $A_3$: ***O($8^n$)*** |
|---|---|---|
| ⬇ | ⬇ | ⬇ |
| **30 μs** | **182 hours** | **40.000 billion years** |

# Simplifying assumptions

- Single processor
- Homogeneous task sets
- Fully preemptive tasks
- Simultaneous activations
- No precedence constraints
- No resource constraints

# Algorithm taxonomy

- Preemptive vs. Non Preemptive
- Static vs. dynamic
- On line vs. Off line
- Optimal vs. Heuristic

# Static vs. Dynamic

**Static**

scheduling decisions are taken based on fixed parameters, statically assigned to tasks before activation.

**Dynamic**

scheduling decisions are taken based on parameters that can change with time.

# Off-line vs. On-line

**Off-line**

all scheduling decisions are taken before task activation: the schedule is stored in a table (**table-driven scheduling**).

**On-line**

scheduling decisions are taken at run time on the set of active tasks.

# Optimal vs. Heuristic

**Optimal**

They generate a schedule that minimizes a cost function, defined based on an optimality criterion.

**Heuristic**

They generate a schedule according to a heuristic function that tries to satisfy an optimality criterion, but there is no guarantee of success.

# Optimality criteria

- **Feasibility**: Find a feasible schedule if there exists one.
- Minimize the **maximum lateness**
- Minimize the **number of deadline miss**
- Assign a value to each task, then maximize the **cumulative value** of the feasible tasks

# Task set assumptions

We consider algorithms for different types of tasks:

- Single-job tasks (one shot)

  tasks with a single activation (not recurrent)

- Periodic tasks

  recurrent tasks regularly activated by a timer (each task potentially generates infinite jobs)

- Aperiodic/Sporadic tasks

  recurrent tasks irregularly activated by events (each task potentially generates infinite jobs)

- Mixed task sets

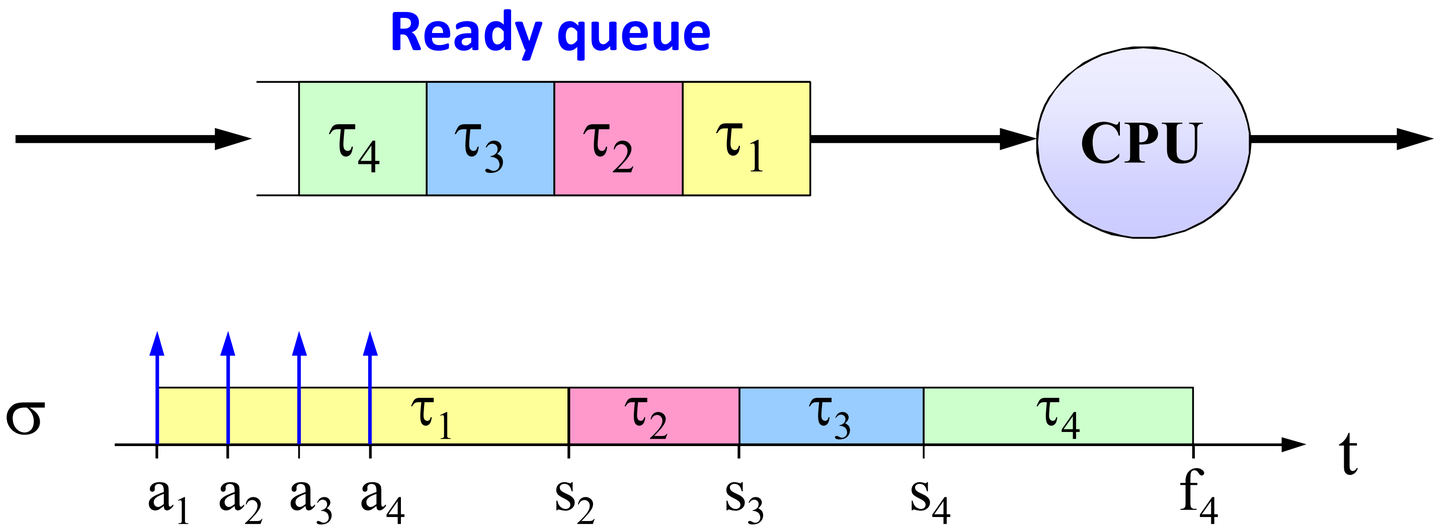
# Classical scheduling policies

- First Come First Served
- Shortest Job First
- Priority Scheduling
- Round Robin

**Not suited for real-time systems**

# First Come First Served

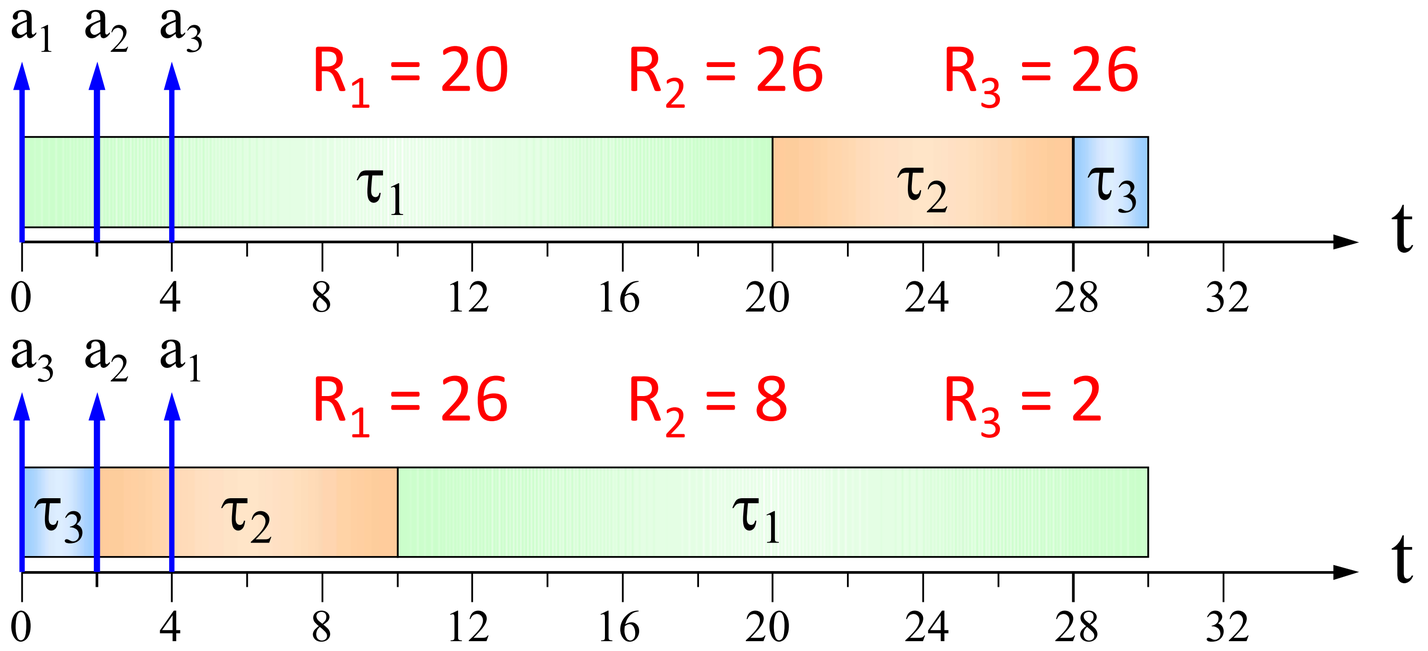
It assigns the CPU to tasks based on their arrival times (intrinsically non preemptive):



# First Come First Served

**- Very unpredictable**

response times strongly depend on task arrivals:



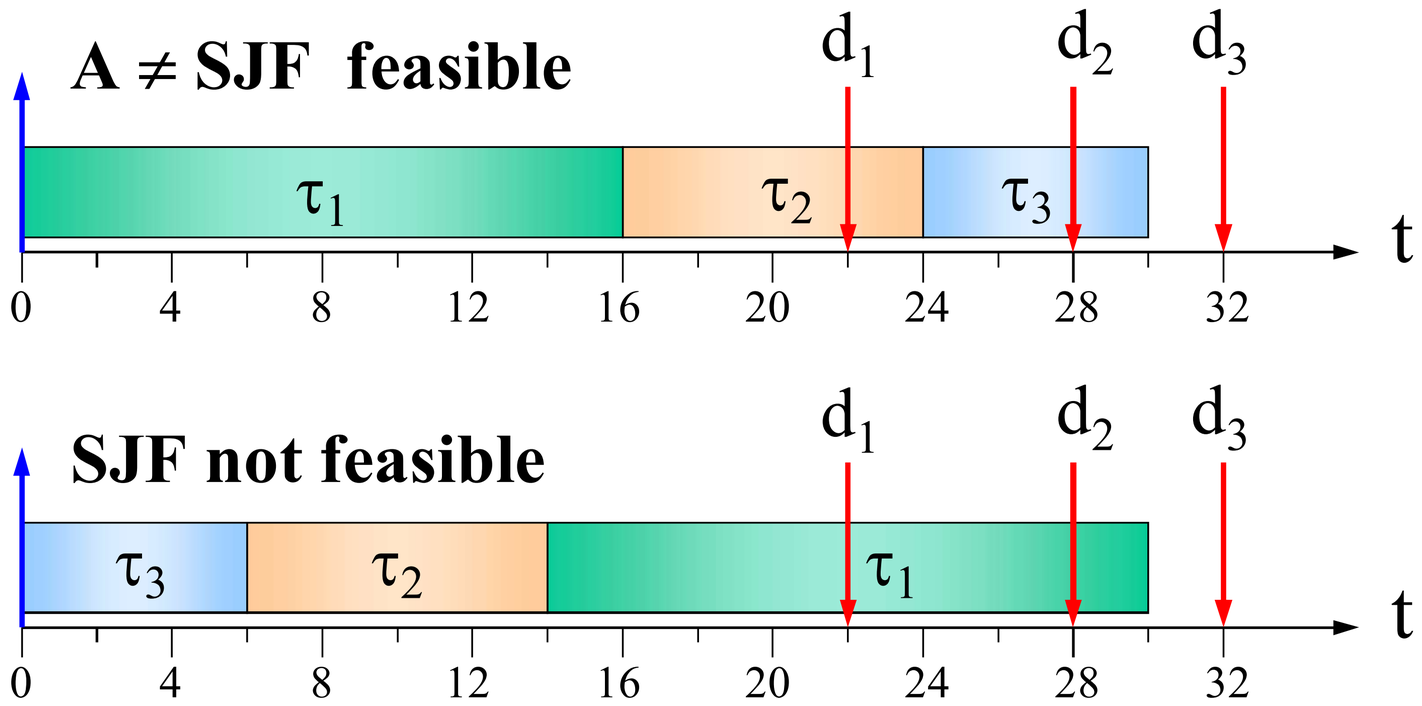
# Shortest Job First (SJF)

It selects the ready task with the shortest computation time.

- Static ($C_i$ is a constant parameter)
- It can be used on line or off-line
- Can be preemptive or non preemptive
- It minimizes the average response time

# Is SUF suited for Real-Time?

- It is not optimal in the sense of feasibility

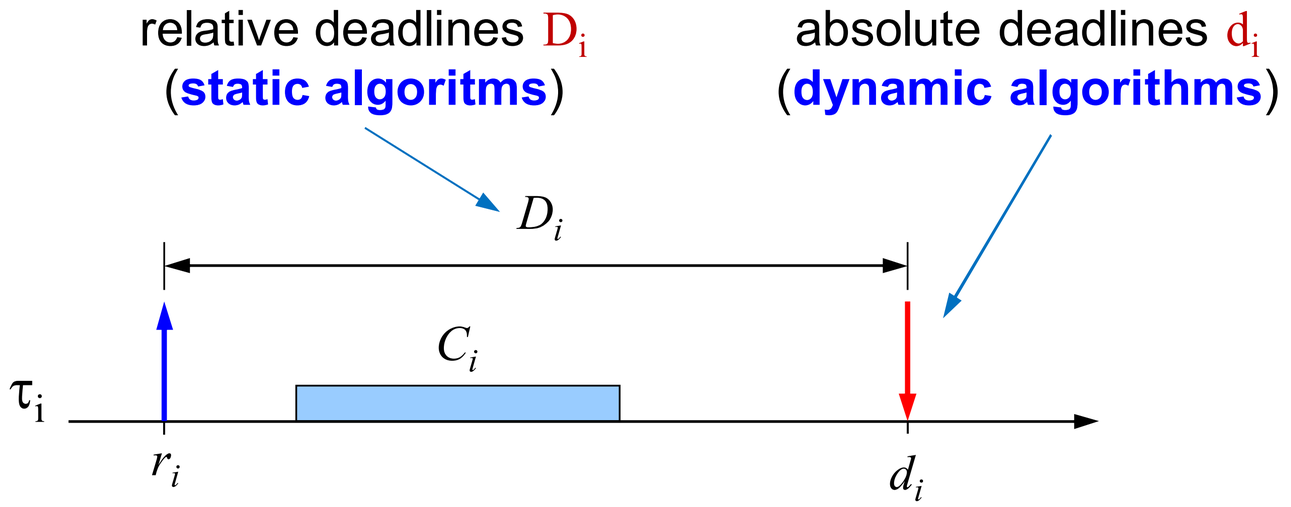


# Real-Time Scheduling Algorithms

Slides source: Prof. Giorgio Buttazzo

# Real-Time Algorithms

Real-time scheduling algorithm can take scheduling decisions base on:




Some consider synchronous arrivals: $\forall i \ r_i = 0$ (**off-line algorithms**)

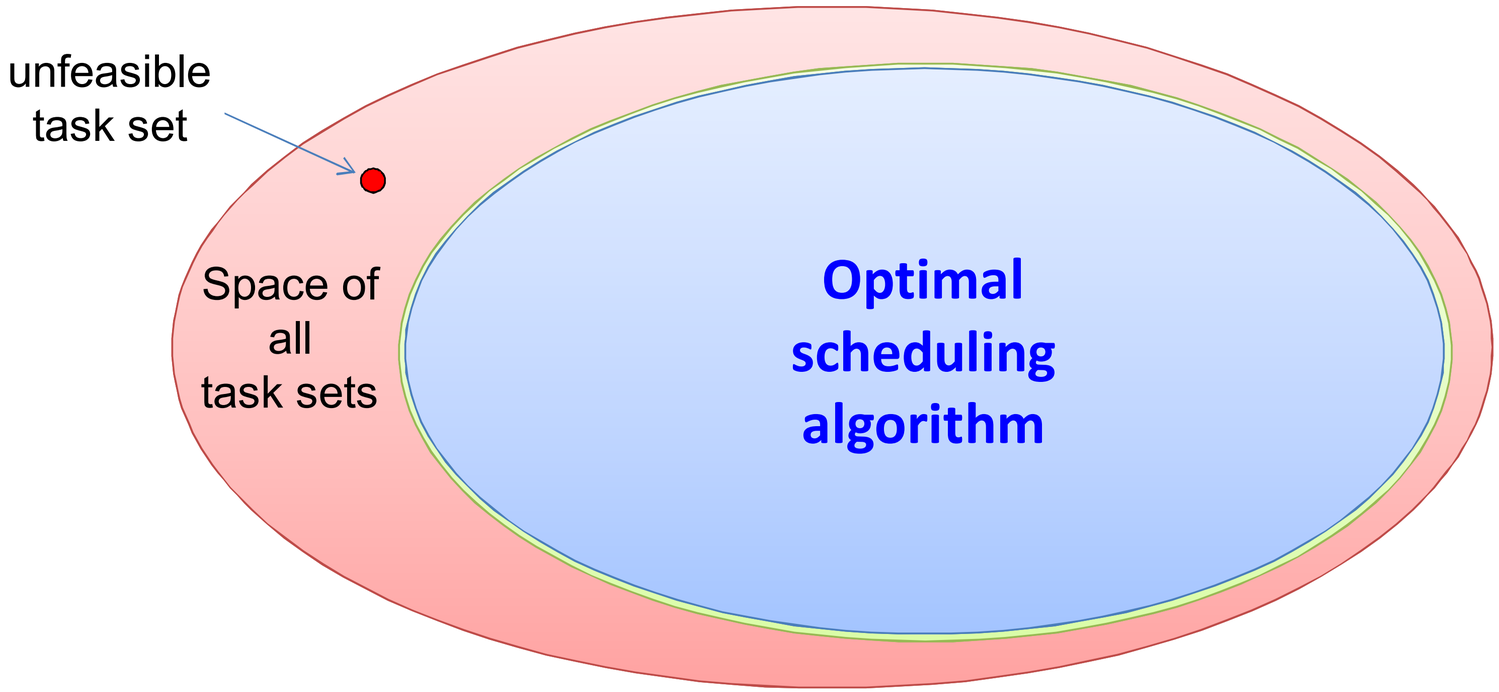
Some consider asynchronous arrivals: $\forall i \ r_i \geq 0$ (**on-line algorithms**)

**On-line algorithms** can be **preemptive** or **non preemptive**

(Preemption is not an issue if all tasks are ready at time $t = 0$).

# Definitions

An optimal algorithm is able to generate a feasibile schedule for all feasible task sets.



# A property of optimal algorithms

If a task set $\Gamma$ is not schedulable by an optimal algorithm, then $\Gamma$ cannot be scheduled by any other algorithm.

If an algorithm A minimizes $L_{max}$ then A is also optimal in the sense of feasibility. The opposite is not true.

# Periodic Task Scheduling

# Problem formulation

We consider a software system consisting of a set $\Gamma$ of $n$ periodic real-time tasks:

$$\Gamma = \{ \tau_1, \tau_2 , \ldots \tau_n \}$$

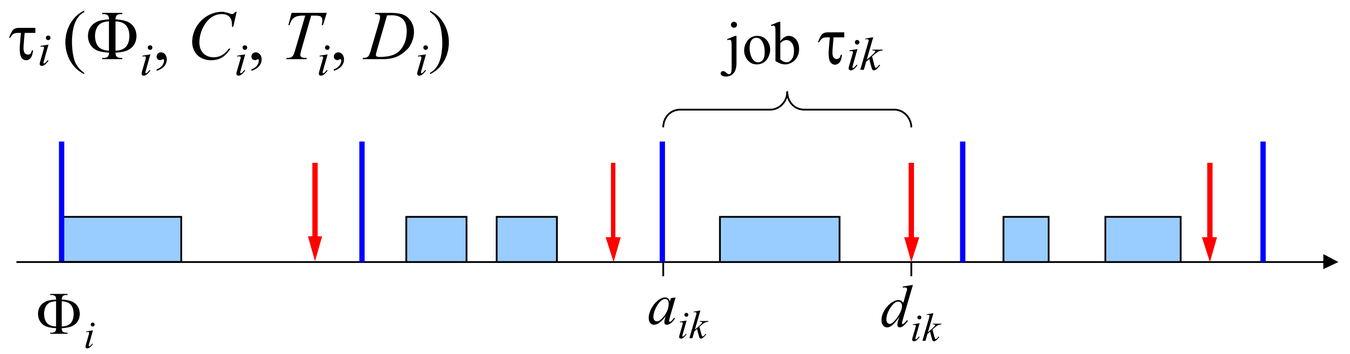
where each task $\tau_i$ is characterized by a set of **parameters**:

- $\Phi_i$ initial arrival time (phase)
- $C_i$ worst-case computation time
- $T_i$ activation period
- $D_i$ relative deadline

We initially consider the following **assumptions**:

- No precedence relations between tasks
- No mutually exclusive resources
- Deadlines equal to periods ($\forall i \ D_i = T_i$)
- Synchronous activations ($\forall i \ \Phi_i = 0$)

# Problem formulation



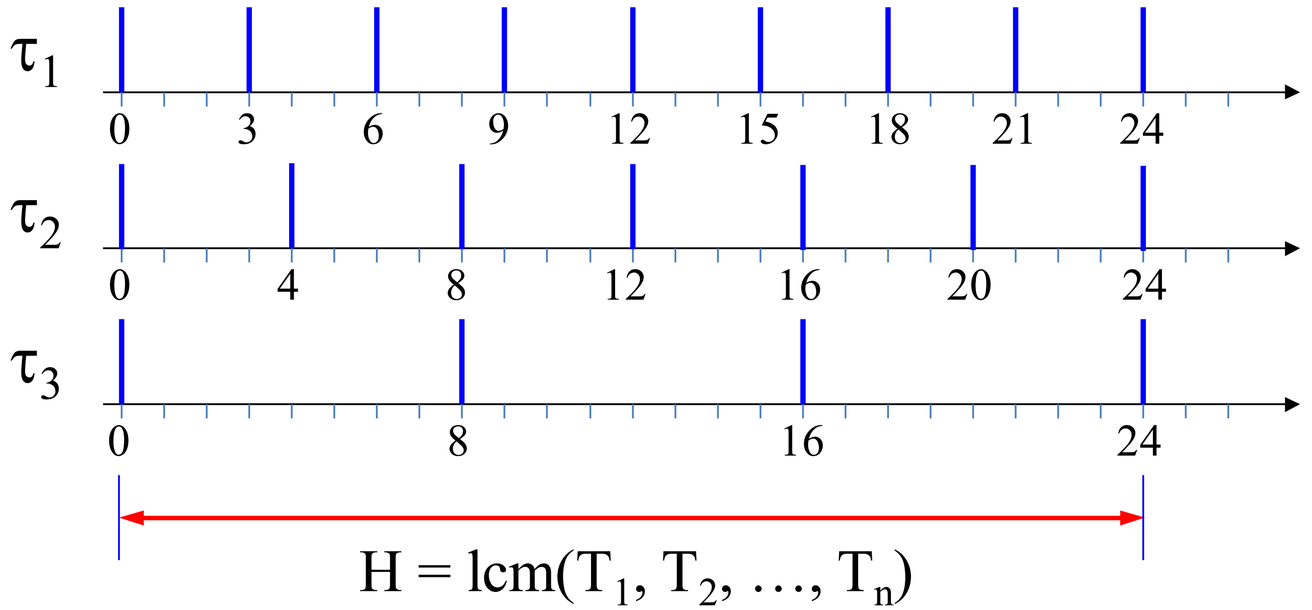

For each periodic task $\tau_i$ we must guarantee that:

- each job $\tau_{ik}$ is activated at $a_{ik} = \Phi_i + (k\text{-}1)T_i$
- each job $\tau_{ik}$ completes within $d_{ik} = a_{ik} + D_i$

There are several wrong ways to achieve this goal.

# Hyperperiod

The **hyperperiod** is the shortest interval of time after which a set of periodic tasks, activated at time $t = 0$, will be activated again at the same time.



# Timeline scheduling

Also known as **cyclic scheduling**, it has been used for 30 years in military systems, navigation, and monitoring systems.

- Air traffic control systems
- Space Shuttle
- Boeing 777
- Airbus navigation system

- This method is very simple and highly predictable. Therefore, it is still used in avionics to manage safety-critical components.
- But it has low flexibility, hence it should be avoided in systems that need to be adapted or frequently upgraded.

# Timeline scheduling

## Method

- The time axis is divided in intervals (*time slots*) of length equal to $\mathrm{GCD}(T_1, \ldots, T_n)$.
- Each task is statically allocated into a slot in order to meet the desired request rate.
- The execution in each slot is activated by a timer.

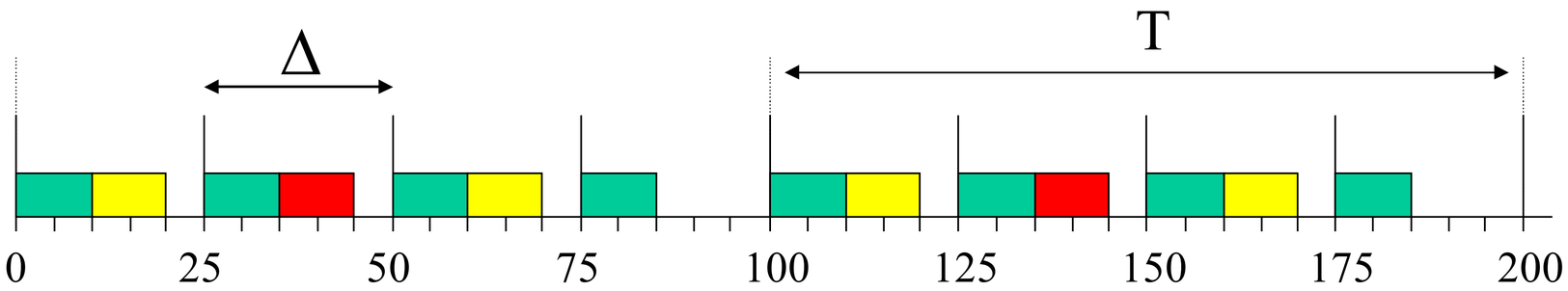
# Timeline scheduling

**Example:**

| task | $C_i$ | $T_i$ |
|---|---|---|
| A | 10 ms | 25 ms |
| B | 10 ms | 50 ms |
| C | 10 ms | 100 ms |

$\Delta$ = GCD (minor cycle)

T = lcm (major cycle)

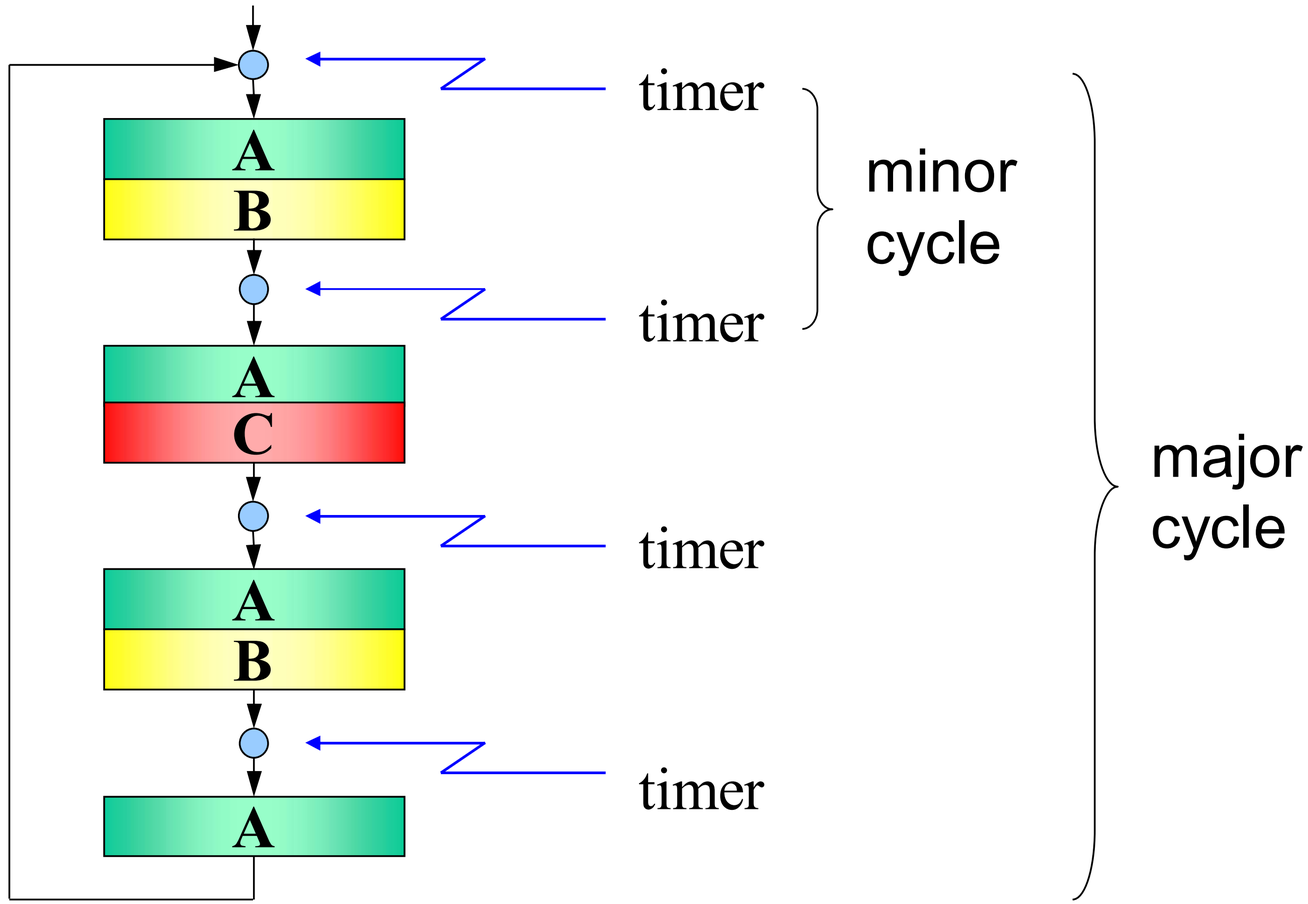



Schedulability test:

$$\begin{cases} C_A + C_B \le \Delta \\ C_A + C_C \le \Delta \end{cases}$$

# Timeline scheduling

**Implementation:**



# Timeline scheduling

## Advantages

- Simple implementation (no RTOS is required).
- Low run-time overhead.
- All tasks run with very low jitter.

## Disadvantages

- It is not robust during overloads.
- It is difficult to expand the schedule.
- It is not easy to handle aperiodic activities.

# Problems during overloads

What do we do during task overruns?

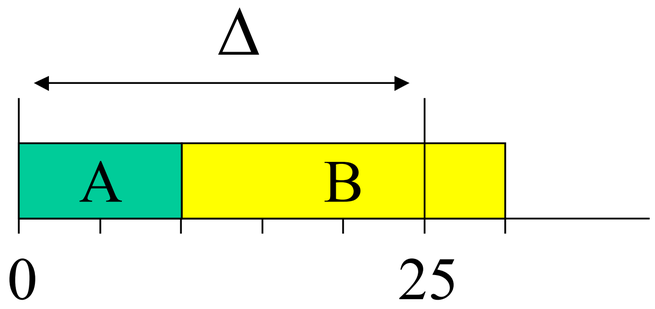
- **Let the task continue**
  - we can have a ***domino effect*** on all the other tasks (timeline break)

- **Abort the task**
  - the system can remain in inconsistent states.

# Expansibility

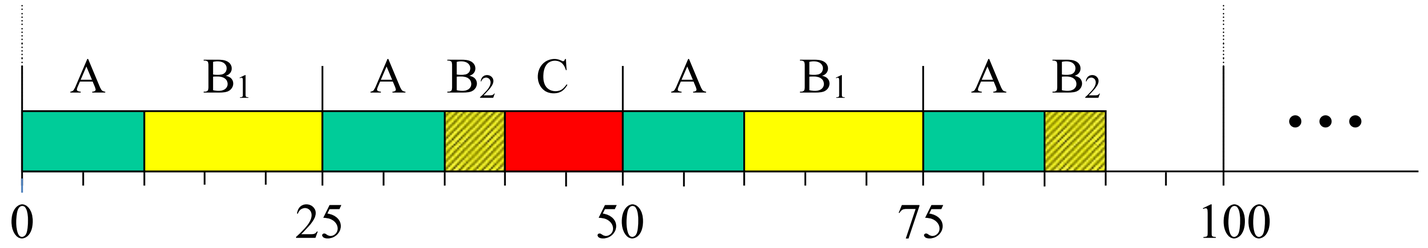
If one or more tasks need to be upgraded, we may have to re-design the whole schedule again.

**Example:** B is updated so that $C_B = 20$ ms
now $C_A + C_B > \Delta$



# Expansibility

- We have to split task B in two subtasks ($B_1$, $B_2$) and re-build the schedule:




**Guarantee:** $\begin{cases} C_A + C_{B1} \leq \Delta \\ C_A + C_{B2} + C_C \leq \Delta \end{cases}$

# Expansibility

If the frequency of some task is changed, the impact can be even more significant:

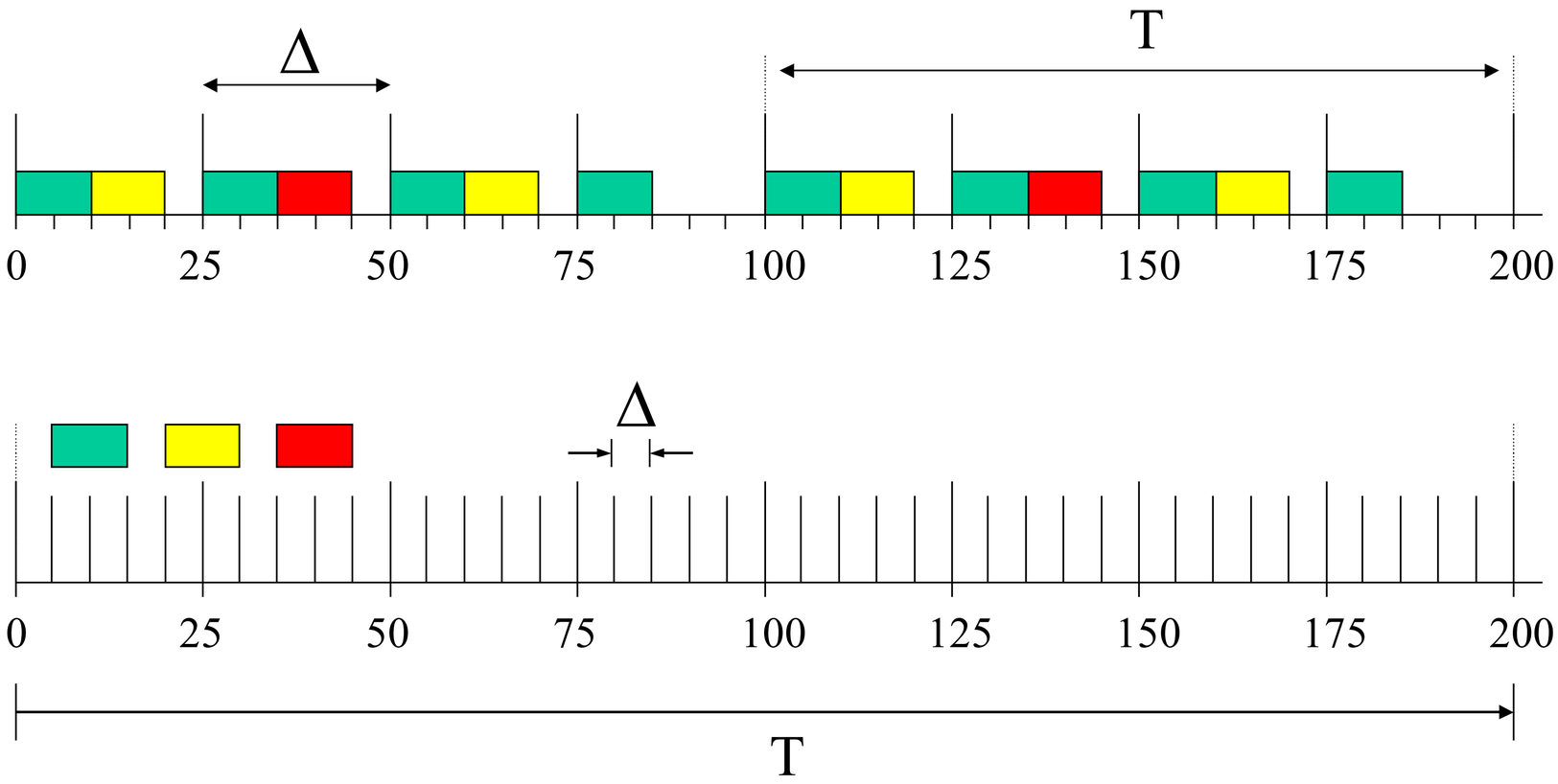
| task | $T_{old}$ | $T_{new}$ |
|---|---|---|
| A | 25 ms | 25 ms |
| B | 50 ms | **40 ms** |
| C | 100 ms | 100 ms |

minor cycle: $\Delta = 25$ $\Delta = 5$

major cycle: $T = 100$ $T = 200$

(40 sync. per cycle!)

# Example



# Priority Scheduling

## Method

1. Assign priorities to each task based on its timing constraints.

2. Verify the feasibility of the schedule using analytical techniques.

3. Execute tasks on a priority-based kernel.

# How to assign priorities?

- Typically, task priorities are assigned based on the their relative importance.
- However, different priority assignments can lead to different processor utilization bounds.

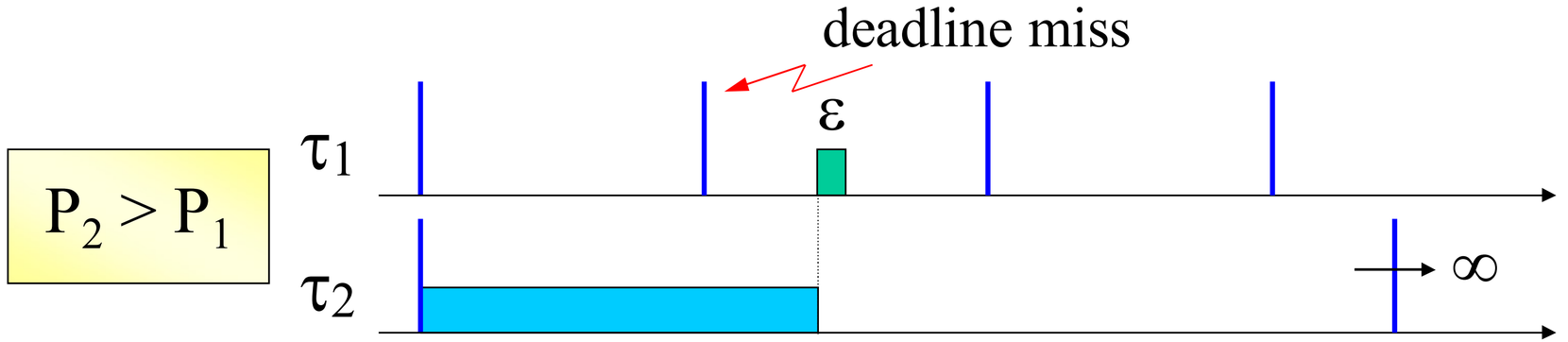
# Priority vs. importance

If $\tau_2$ is more important than $\tau_1$ and is assigned higher priority, the schedule may not be feasible:

# Priority vs. importance

If priority are not properly assigned, the utilization bound can be arbitrarily small:

An application can be unfeasible even when the processor is almost empty!




$$U = \frac{\varepsilon}{T_1} + \frac{C_2}{\infty} \rightarrow 0$$

# Optimal priority assignments

An algorithm A is **optimal** if it generates a feasible schedule, if there exists one.

- **Rate Monotonic (RM):**

  $P_i \propto 1/T_i$ (static)

  optimal among FP algs for T = D

- **Deadline Monotonic (DM):**

  $P_i \propto 1/D_i$ (static)

  optimal among FP algs for D ≤ T

- **Earliest Deadline First (EDF):**

  $P_i \propto 1/d_{ik}$ (dynamic)

  $d_{i,k} = r_{i,k} + D_i$

  optimal among all algs for D ≤ T

# Rate Monotonic is optimal

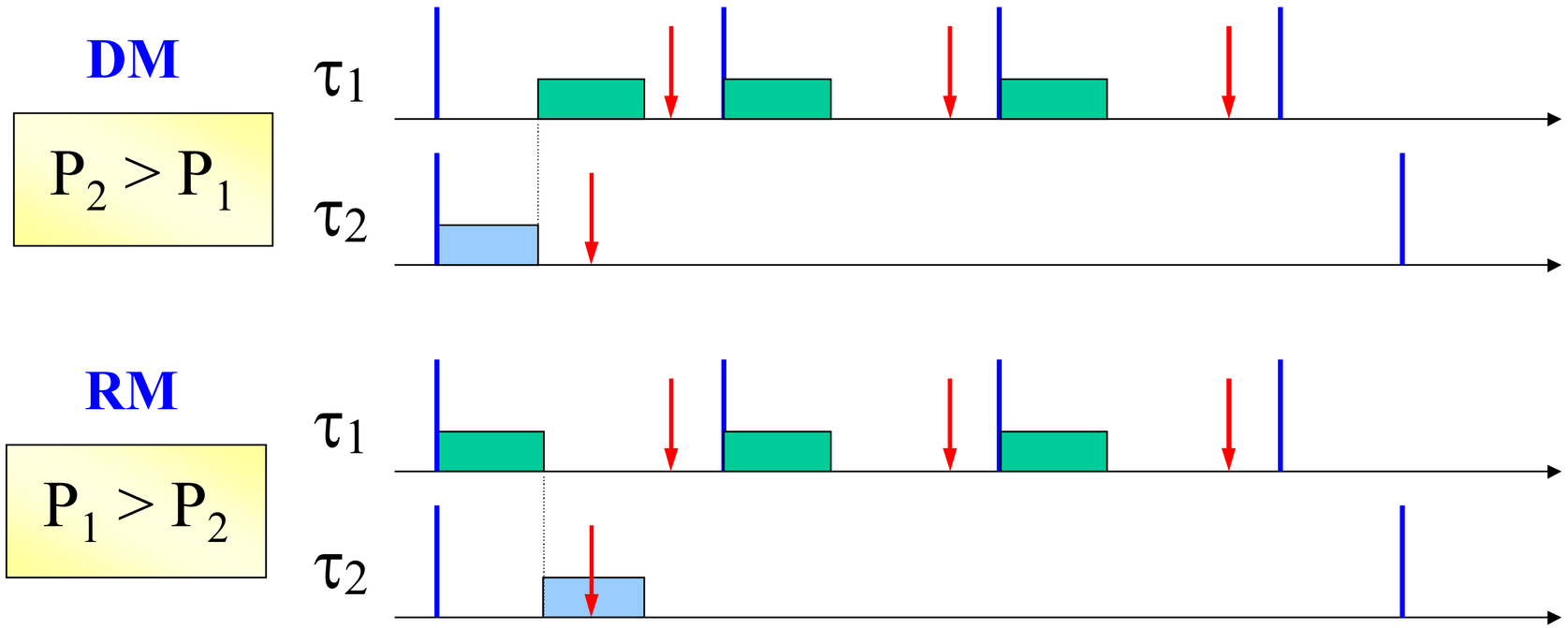
**RM** is **optimal** among all fixed priority algorithms (if $D_i = T_i$):

If there exists a fixed priority assignment which leads to a feasible schedule, then the RM schedule is feasible.

If a task set is not schedulable by RM, then it cannot be scheduled by any fixed priority assignment.

# Deadline Monotonic is optimal

If $D_i \leq T_i$ then the **optimal** priority assignment is given by **Deadline Monotonic** (**DM**):



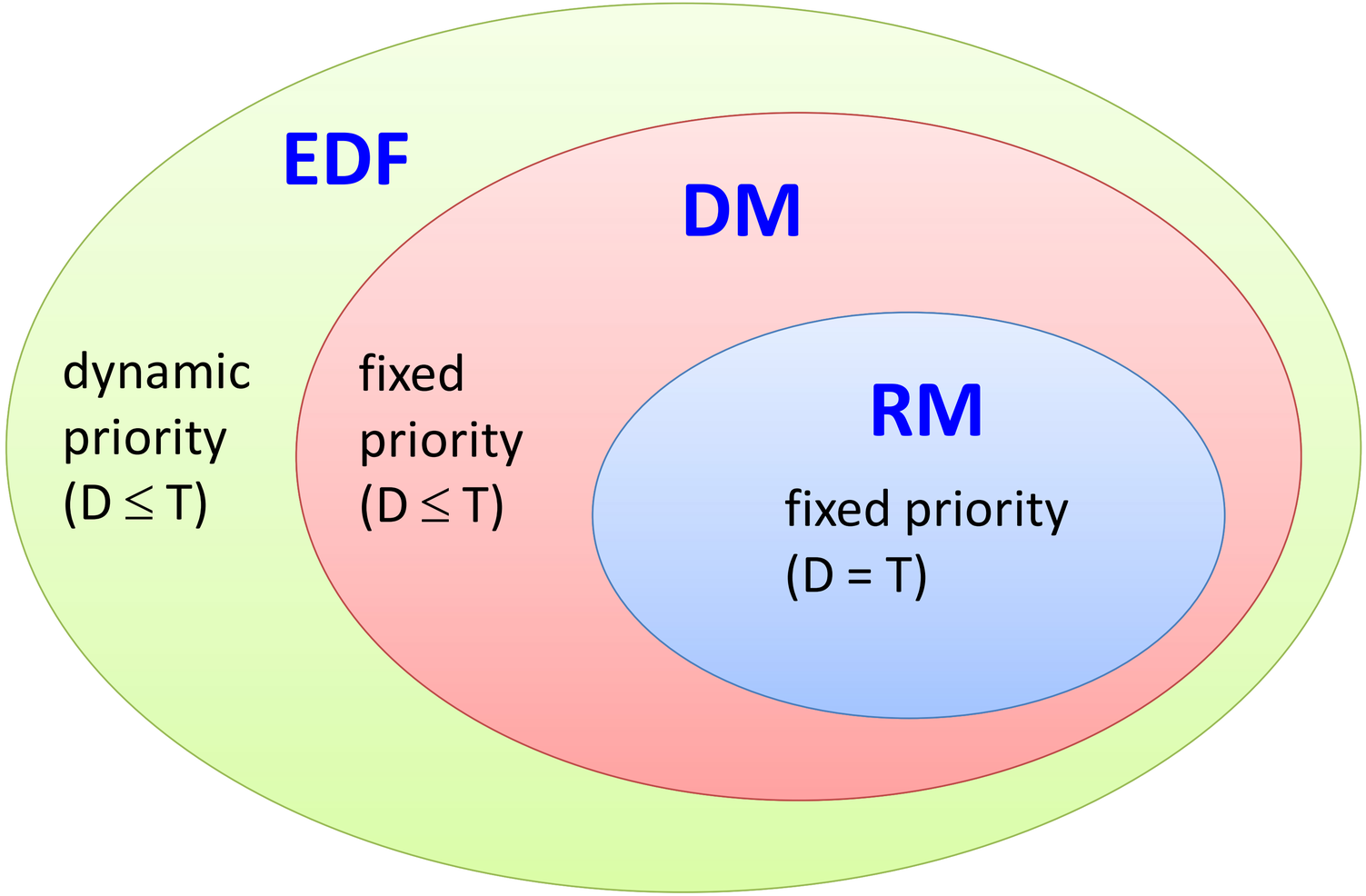
# EDF optimality

**EDF** is **optimal** among all algorithms:

If there exists a feasible schedule for a task set, then EDF will generate a feasible schedule.

If a task set is not schedulable by EDF, then it cannot be scheduled by any algorithm.

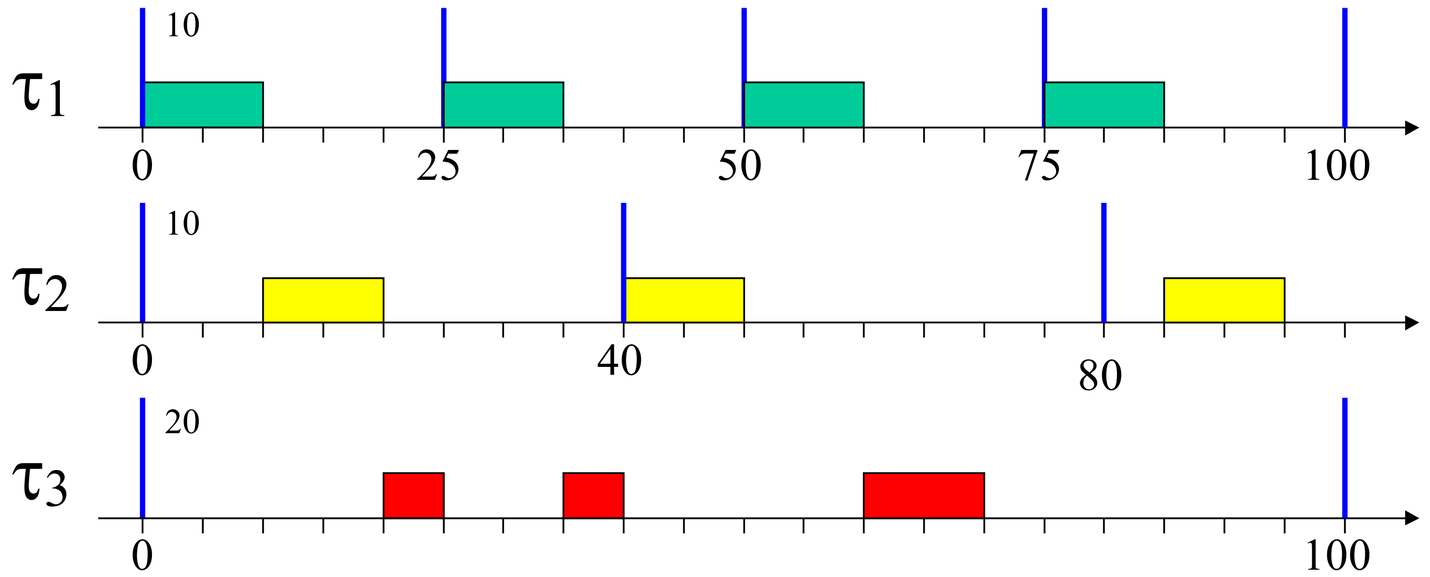
# Optimality


EDF
DM
RM
dynamic
priority
(D ≤ T)
fixed
priority
(D ≤ T)
fixed priority
(D = T)

# Rate Monotonic (RM)

- Each task is assigned a fixed priority proportional to its rate.

$$U_p = \frac{10}{25} + \frac{10}{40} + \frac{20}{100} = 0.85$$

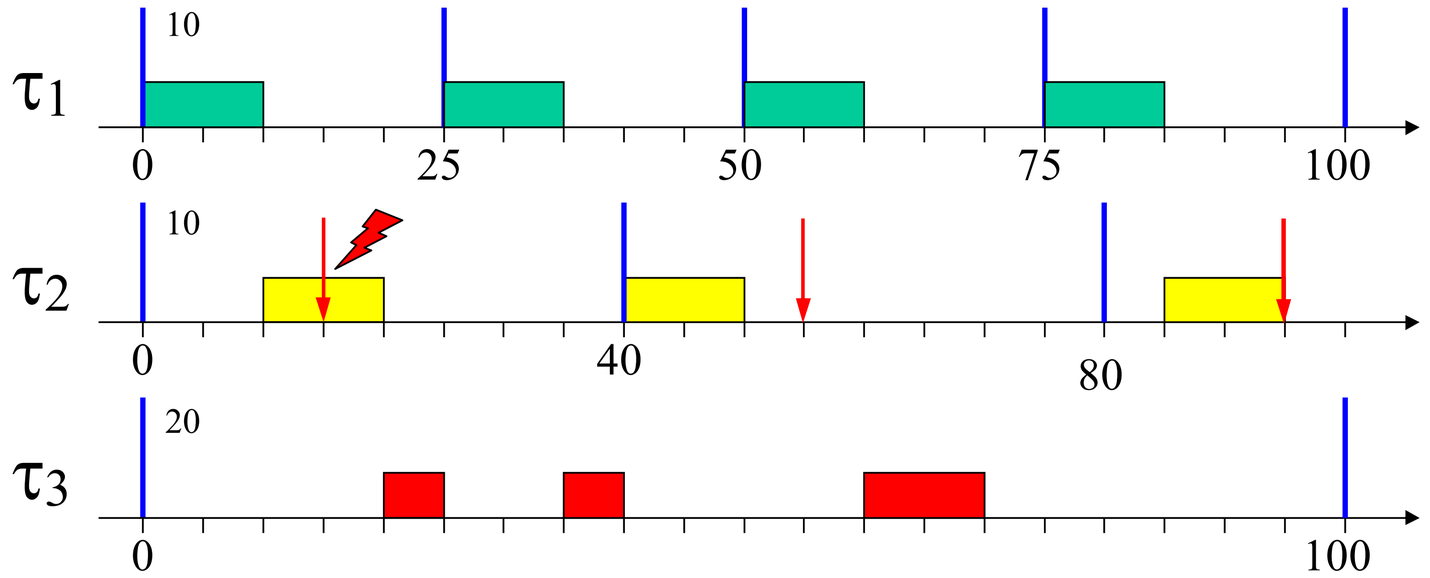



Note that small parameter variations are automatically handled by the scheduler without any intervention.

# Rate Monotonic (RM)

- RM is not optimal if $D_i < T_i$:

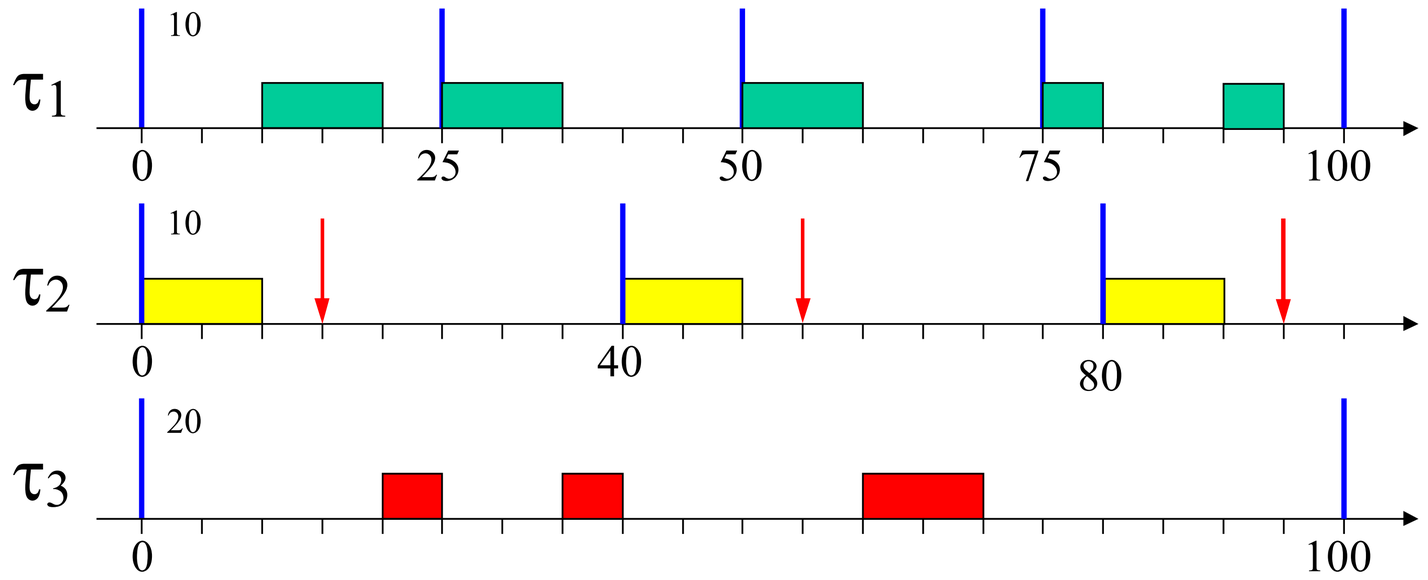
$$U_p = \frac{10}{25} + \frac{10}{40} + \frac{20}{100} = 0.85$$



# Deadline Monotonic (DM)

- DM is able to schedule this task set:

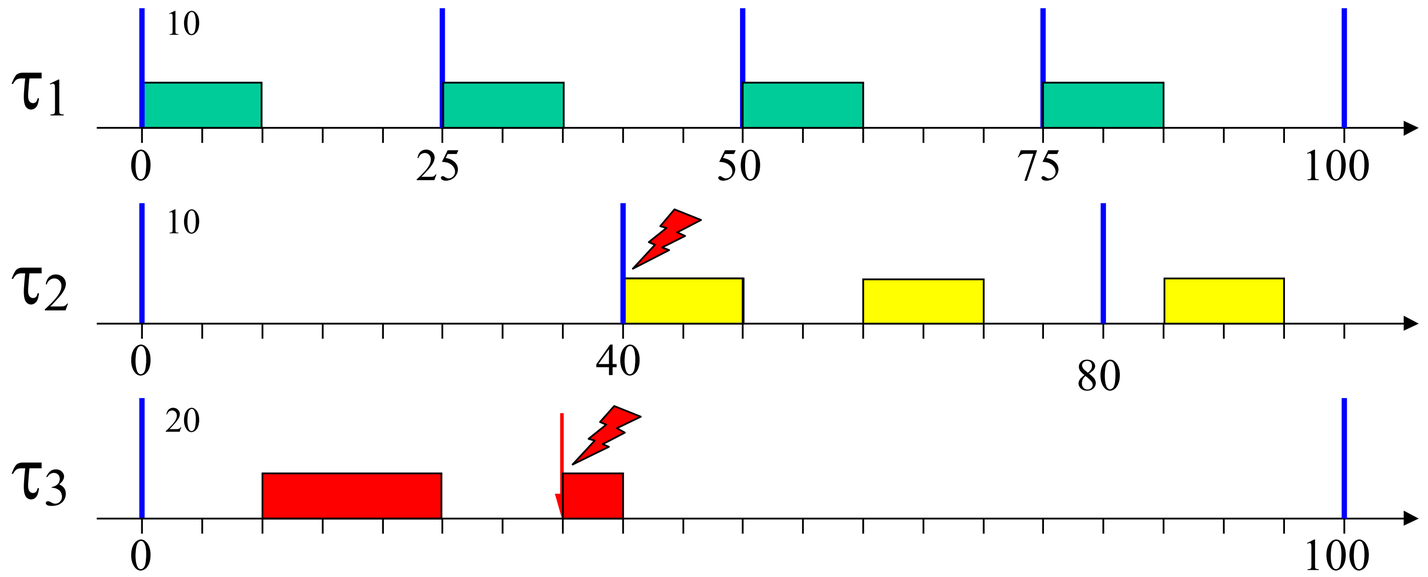
$$U_p = \frac{10}{25} + \frac{10}{40} + \frac{20}{100} = 0.85$$



# Deadline Monotonic (DM)

- However, DM is not able to schedule the tasks if $D_3 = 35$:

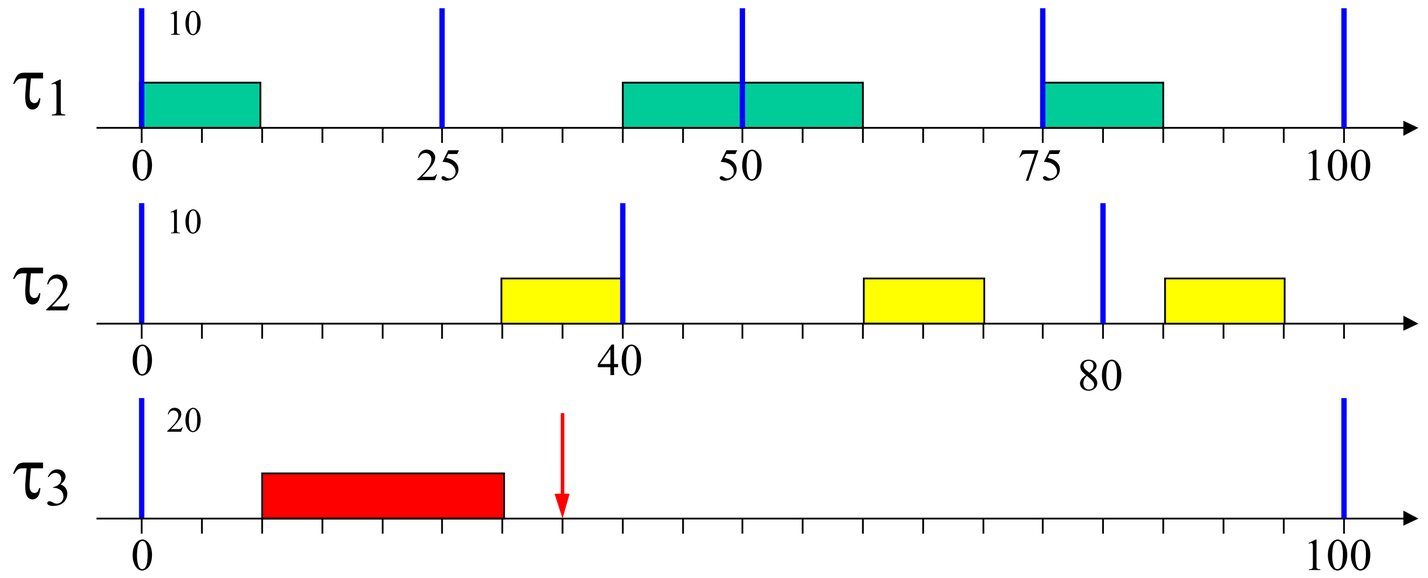
$$U_p = \frac{10}{25} + \frac{10}{40} + \frac{20}{100} = 0.85$$



# EDF

- While EDF generates a feasible schedule:

$$U_p = \frac{10}{25} + \frac{10}{40} + \frac{20}{100} = 0.85$$



# How can we verify feasibility?

- Each task uses the processor for a fraction of time:

$$U_i = \frac{C_i}{T_i}$$

- Hence the total **processor utilization** is:

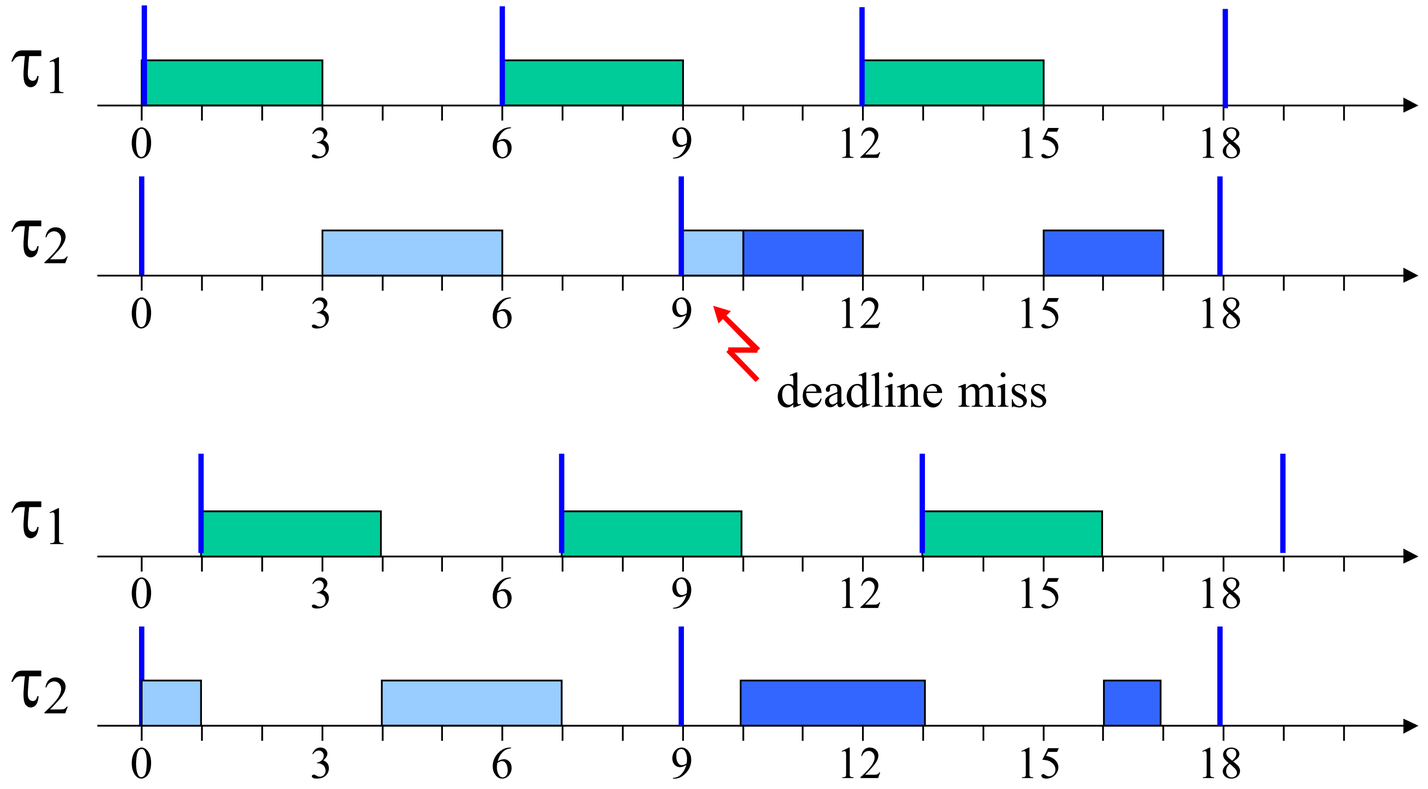
$$U_p = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

- $U_p$ is a measure of the **processor load**.

# Identifying the worst case

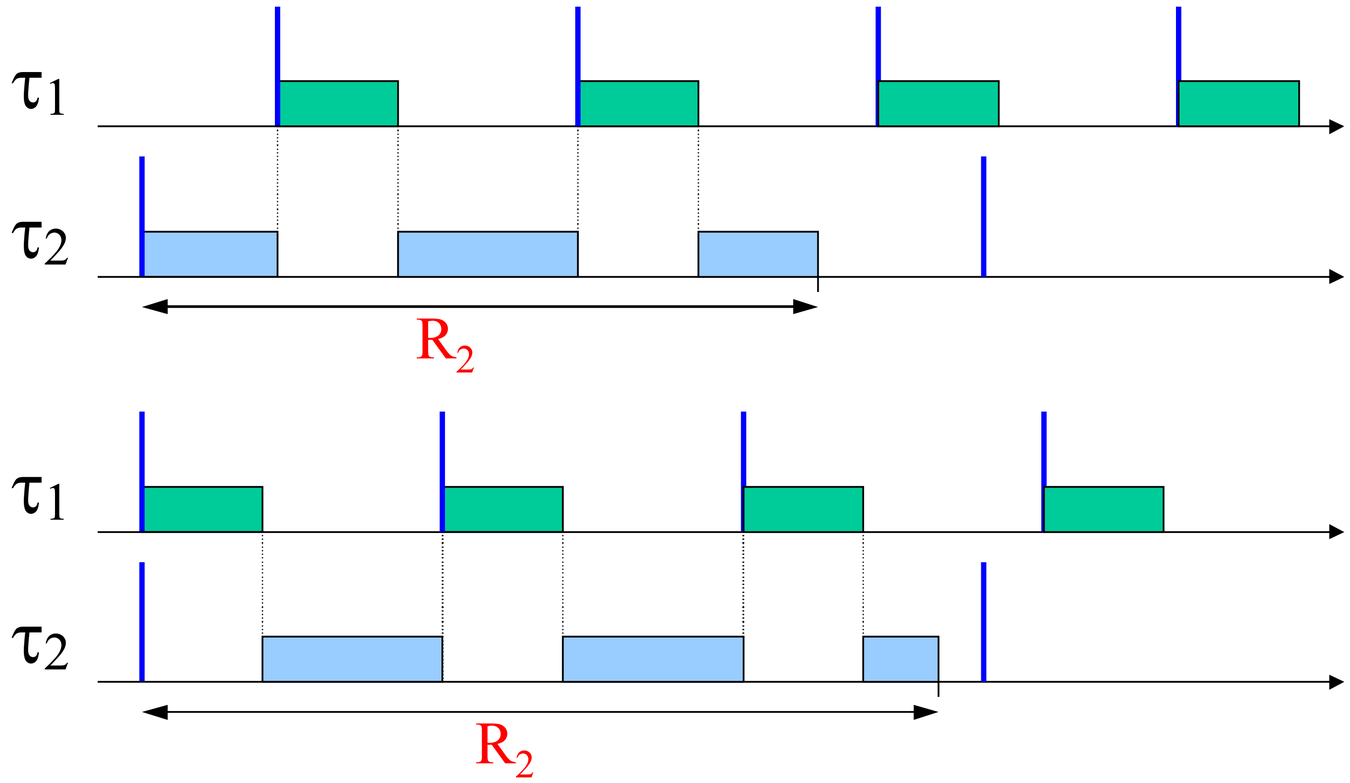
Feasibility may depend on the initial activations (phases):

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.994$$



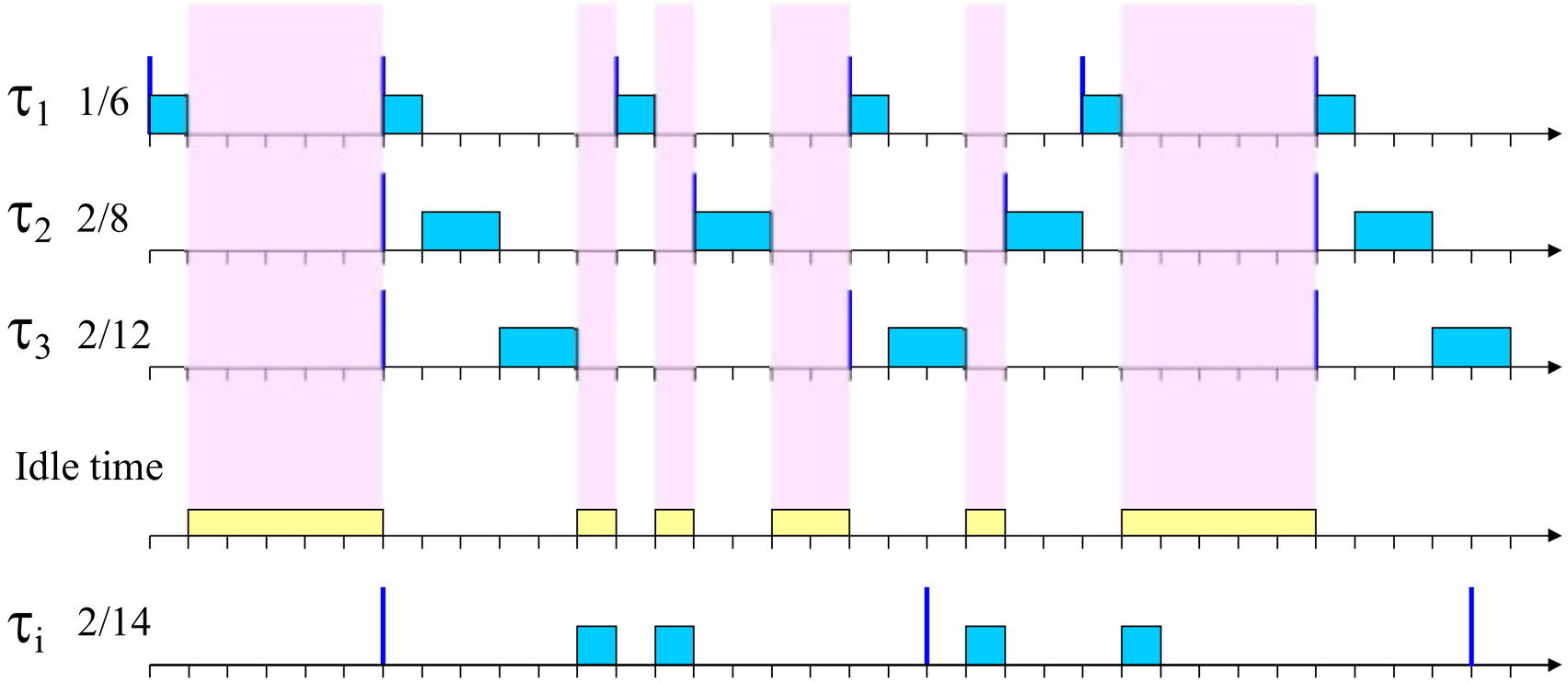
# Critical Instant

For any task $\tau_i$, the longest response time occurs when it arrives together with all higher priority tasks.



# Critical Instant

For independent preemptive tasks under fixed priorities, the critical instant of $\tau_i$, occurs when it arrives together with all higher priority tasks.



# A necessary condition

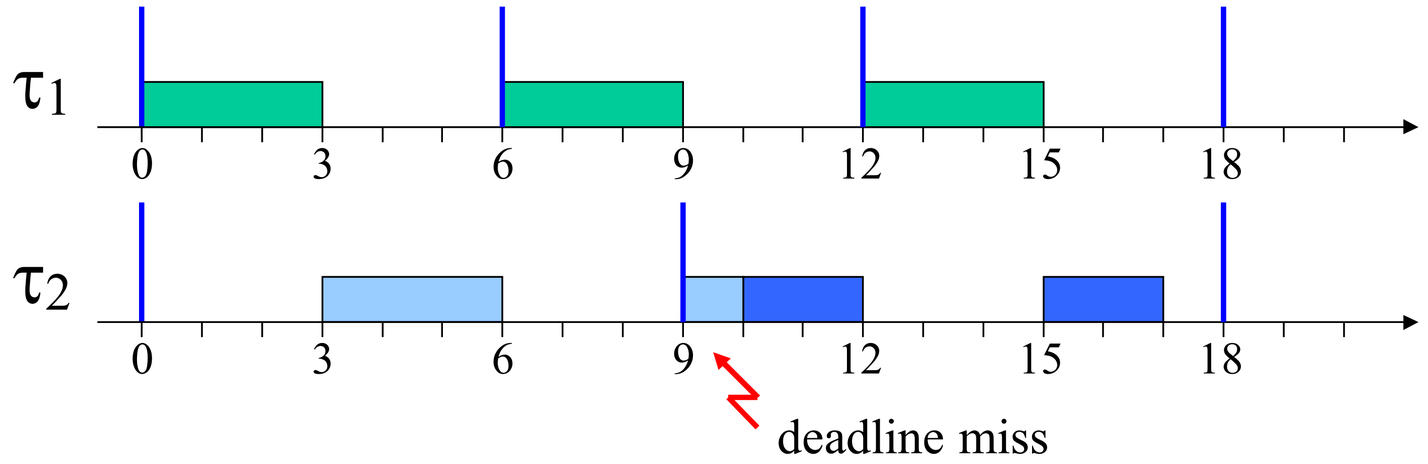
A necessary condition for having a feasible schedule is that $U_p \leq 1$.

In fact, if $U_p > 1$ the processor is overloaded hence the task set cannot be schedulable.

However, there are cases in which $U_p \leq 1$ but the task set is not schedulable by RM.

# An unfeasible RM schedule

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.994$$

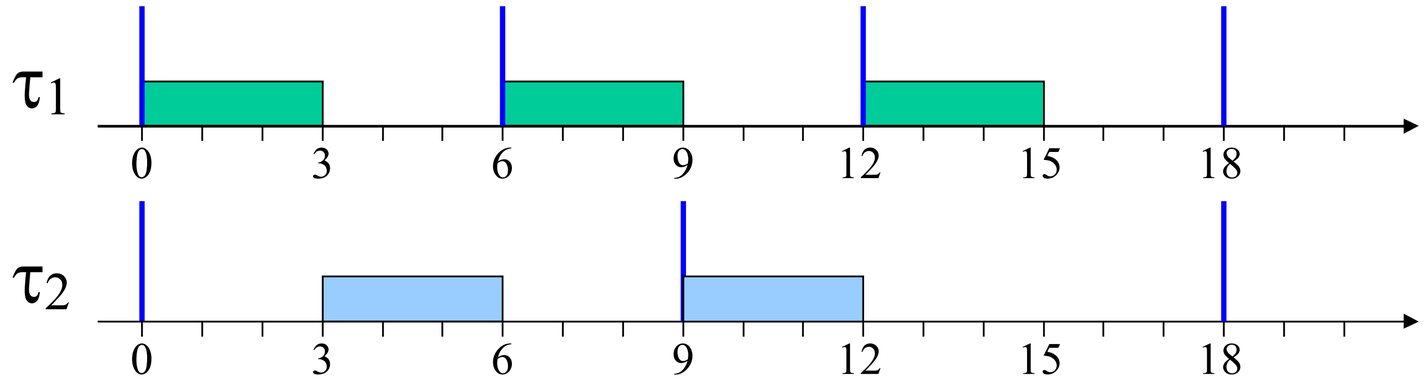



Given this task set (period configuration), what is the higher utilization that guarantees feasibility?

# Utilization upper bound

Utilization upper bound: $U_{ub}(\Gamma) = \max\{U_p(\Gamma) : \Gamma \text{ is schedulable}\}$

$$U_{ub} = \frac{3}{6} + \frac{3}{9} = 0.833$$

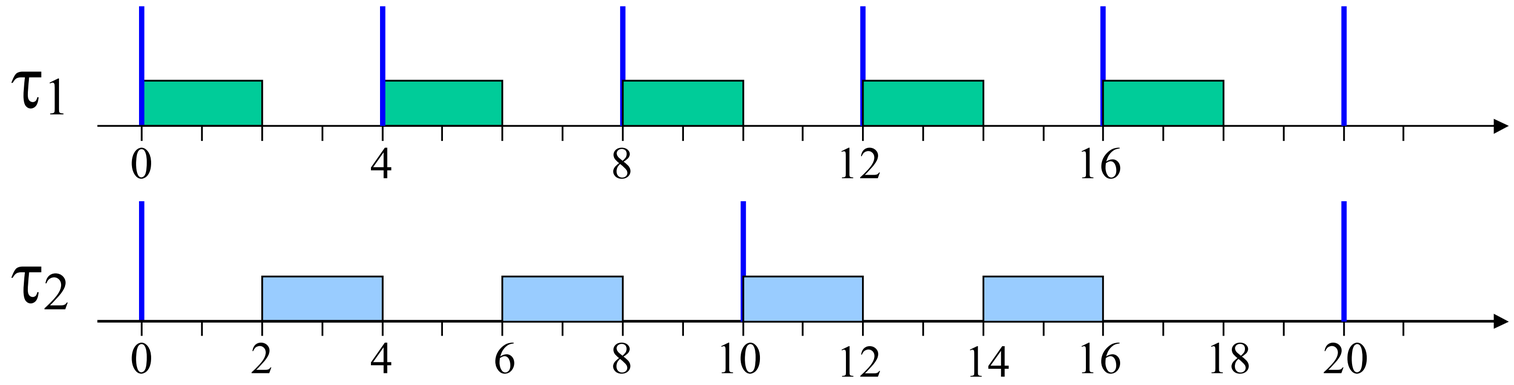



**NOTE**: If $C_1$ or $C_2$ is increased, $\tau_2$ will miss its deadline!

# A different upper bound

Utilization upper bound: $U_{ub}(\Gamma) = \max\{U_p(\Gamma) : \Gamma \text{ is schedulable}\}$

$$U_{ub} = \frac{2}{4} + \frac{4}{10} = 0.9$$

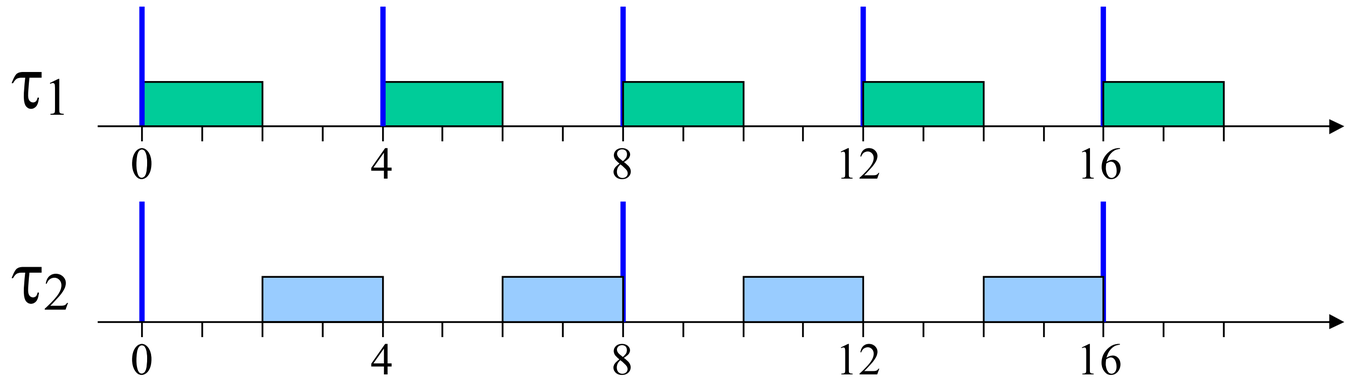



**NOTE**: The upper bound $U_{ub}$ depends on the specific task set.

# A different upper bound

Utilization upper bound: $U_{ub}(\Gamma) = \max\{U_p(\Gamma) : \Gamma \text{ is schedulable}\}$

$$U_{ub} = \frac{2}{4} + \frac{4}{8} = 1$$



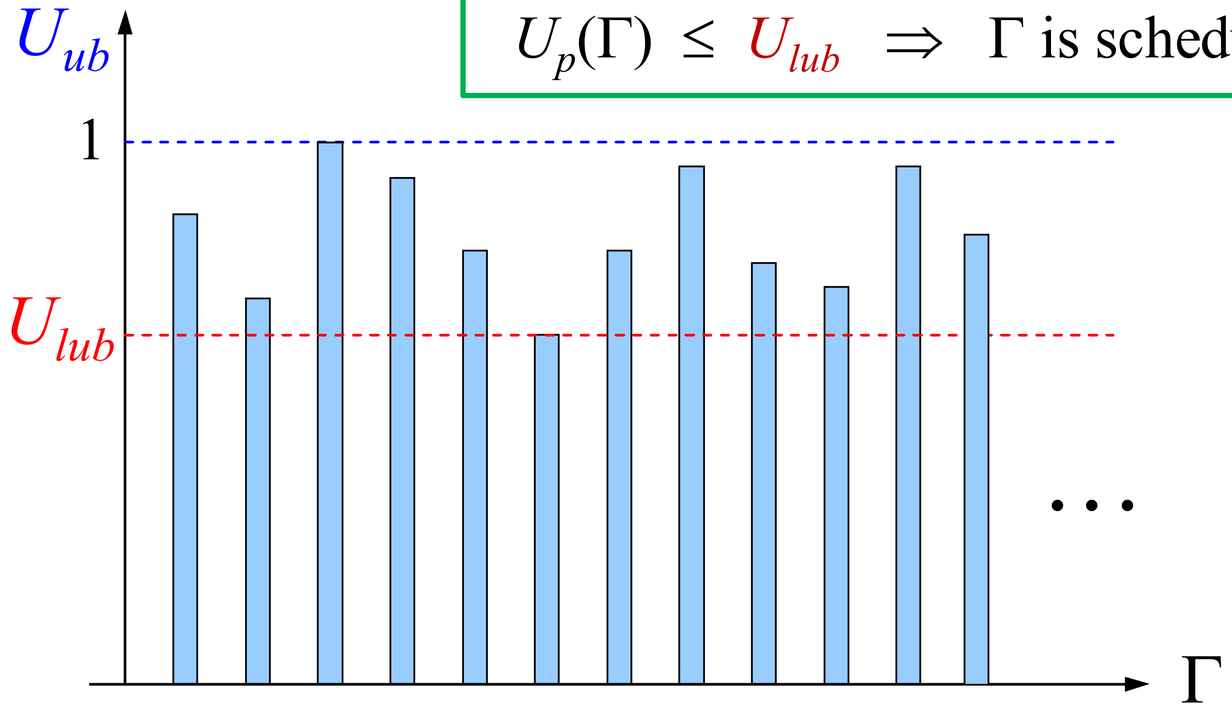

**NOTE**: The upper bound $U_{ub}$ depends on the specific task set.

# The least upper bound

Utilization upper bound: $U_{ub}(\Gamma) = \max\{U_p(\Gamma) : \Gamma \text{ is schedulable}\}$

Util. least upper bound: $U_{lub} = \min_{\Gamma}\{U_{ub}(\Gamma)\}$

$$U_p(\Gamma) \le U_{ub} \Leftrightarrow \Gamma \text{ is schedulable}$$
$$U_p(\Gamma) \le U_{lub} \Rightarrow \Gamma \text{ is schedulable}$$



# A sufficient condition

If $U_p \leq U_{lub}$ the task set is certainly schedulable with the RM algorithm.

**NOTE**

If $U_{lub} < U_p \leq 1$ we cannot say anything about the feasibility of that task set.

# $U_{lub}$ for RM

In 1973, **Liu and Layland** proved that for a set of $n$ periodic tasks:

$$U_{lub}^{\mathrm{RM}} = n\left(2^{1/n} - 1\right)$$

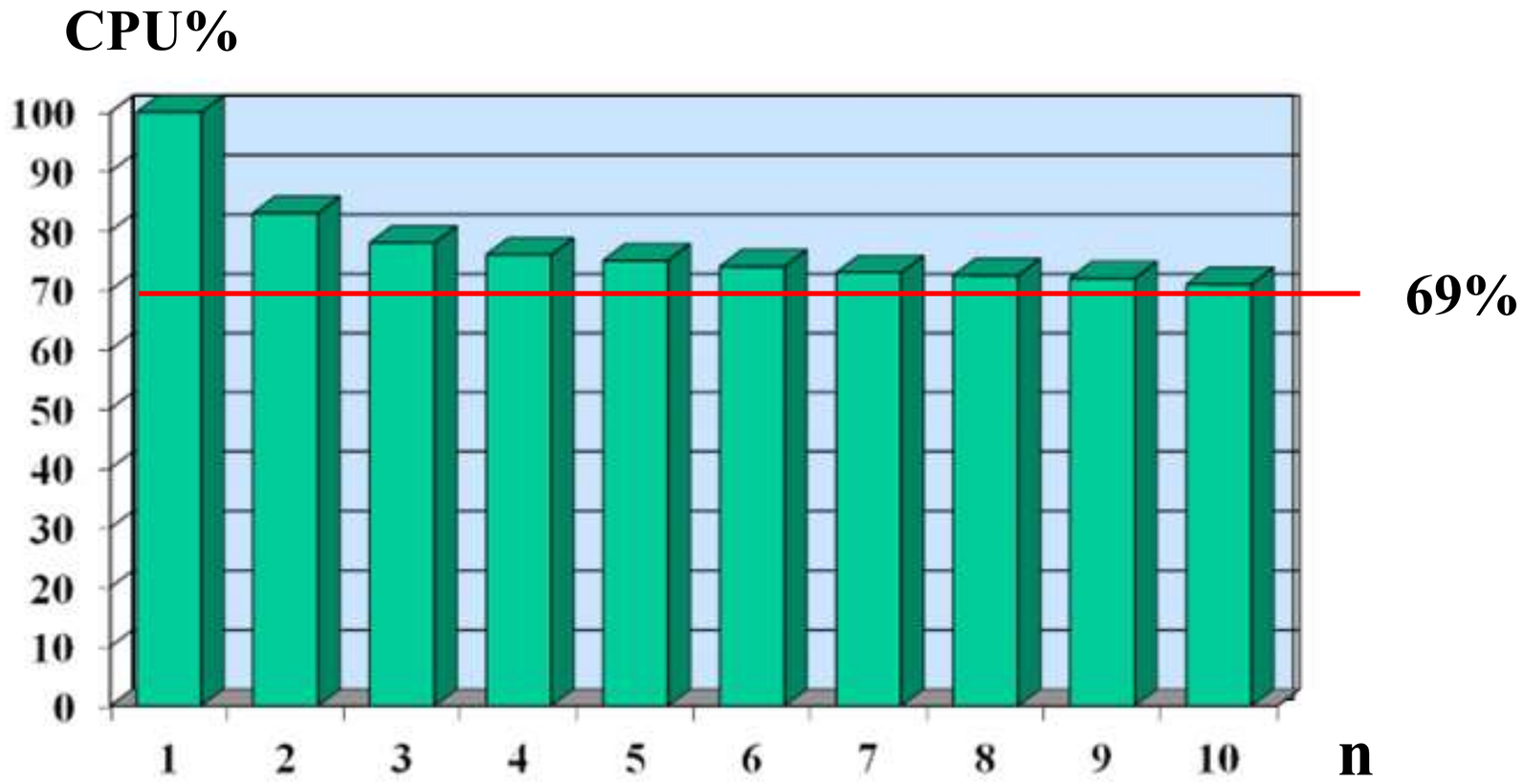
**Assumptions:**

1. For each task, $D_i = T_i$
2. For each task, $\Phi_i = 0$
3. Tasks are independent

**Reference**

C.L. Liu and J.W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment", *Journal of the Association for Computing Machinery*, 20(1), 1973.

# RM Least Upper Bound

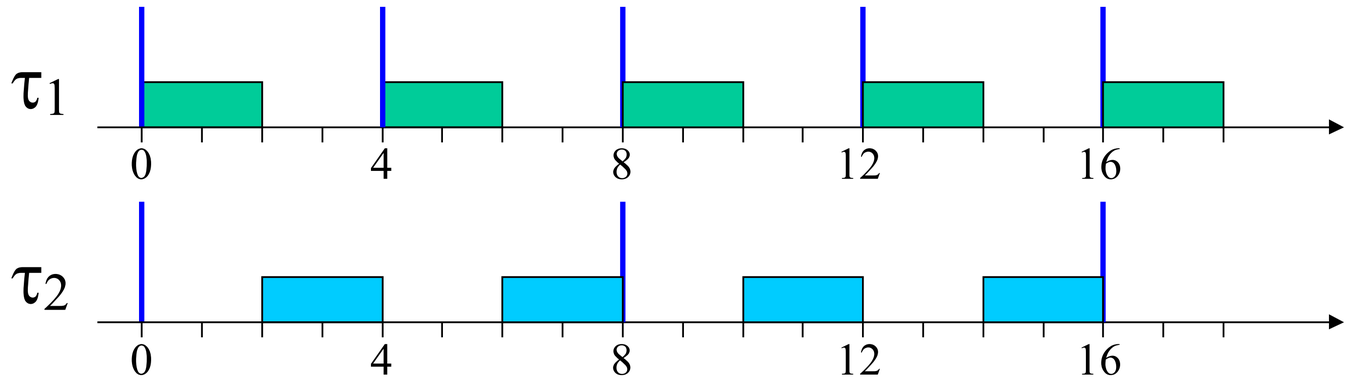



for $n \to \infty \qquad U_{lub} \to \ln 2$

# A special case

If tasks have harmonic periods $U_{lub} = 1$.

$$U_p = \frac{2}{4} + \frac{4}{8} = 1$$



# RM Guarantee Test

- We compute the processor utilization as:

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

- Guarantee Test (only sufficient):

$$U_{lub}^{\mathrm{RM}} = n\left(2^{1/n} - 1\right)$$

# The Hyperbolic Bound

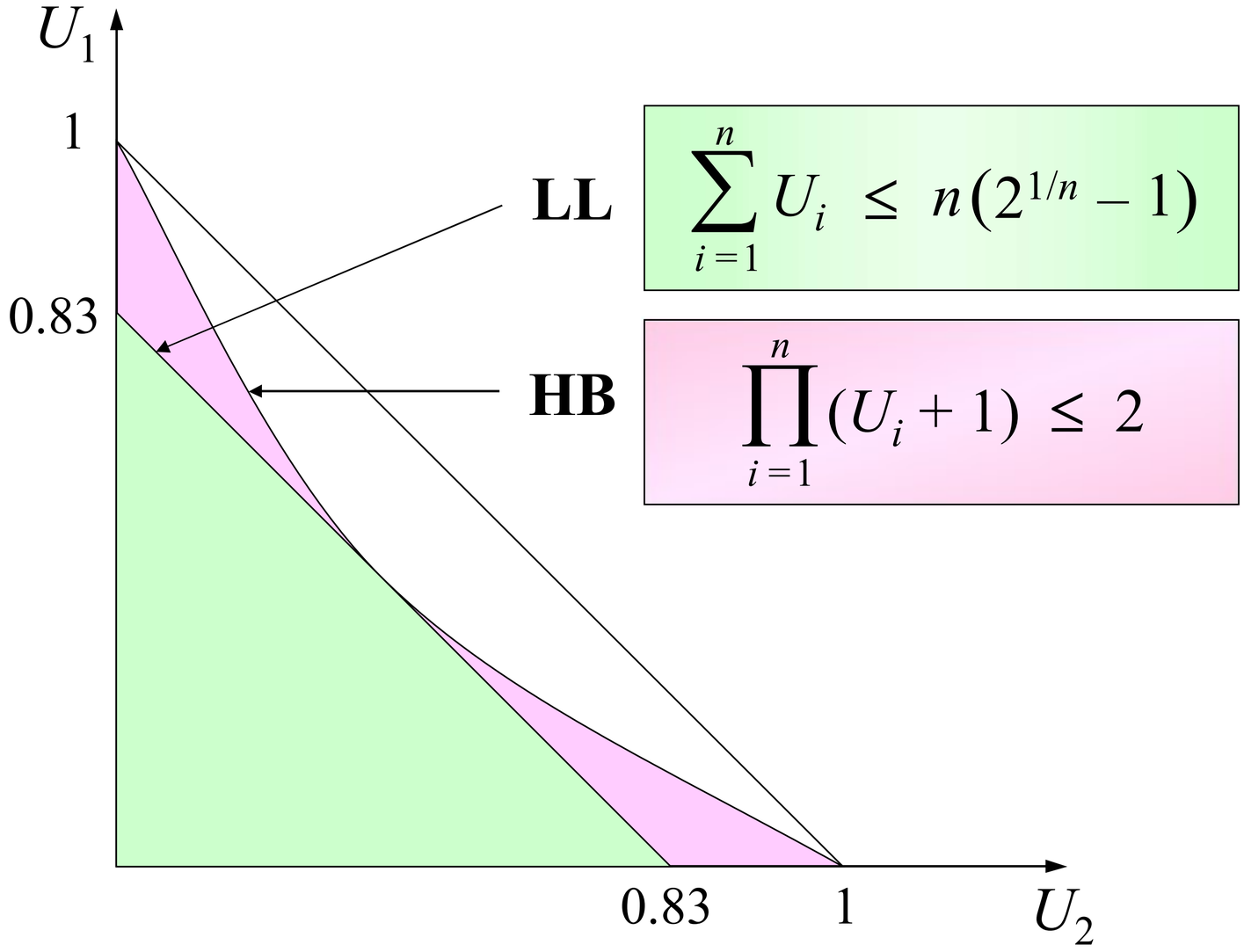
A more efficient schedulability test for RM based on task utilizations (assuming $D_i = T_i$) is the **Hyberbolic Bound**:

$$\prod_{i=1}^{n} (U_i + 1) \leq 2$$

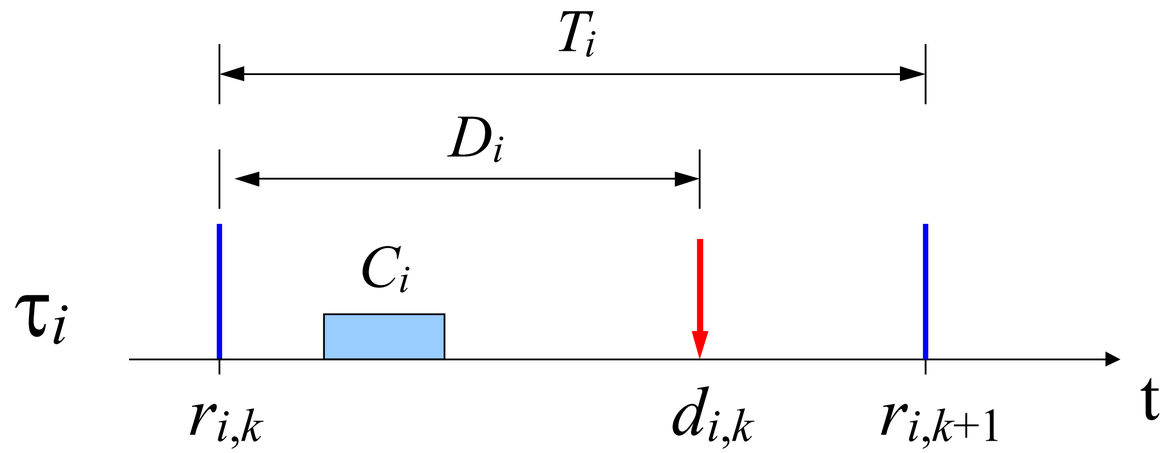
**Reference**

E. Bini, G. C. Buttazzo, and G. M. Buttazzo, "Rate Monotonic Analysis: The Hyperbolic Bound", *IEEE Transactions on Computers*, Vol. 52, No. 7, pp. 933-942, July 2003.

# HB vs. LL



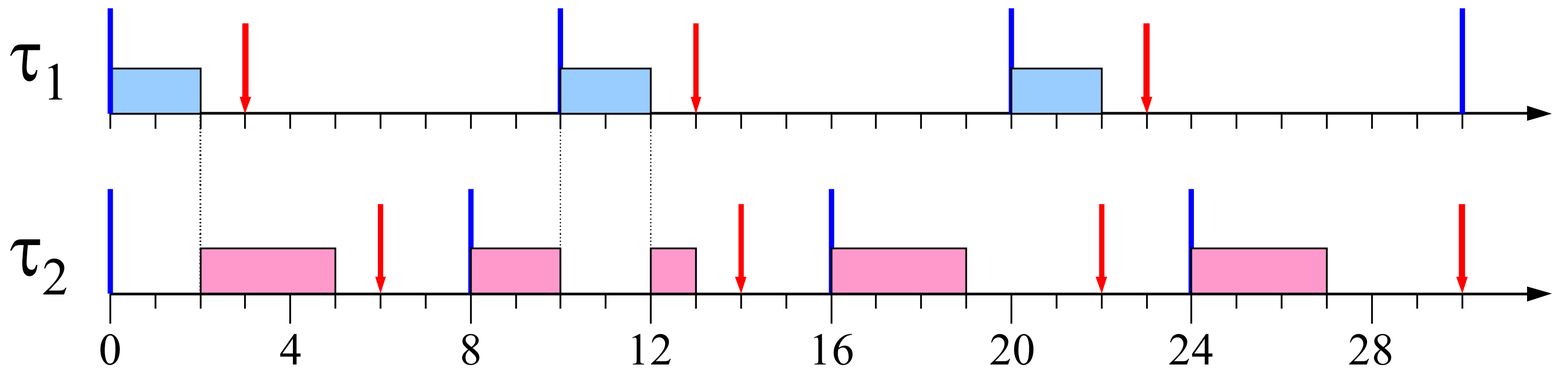
# Extension to tasks with D $\leq$ T




**Scheduling algorithms**

- Deadline Monotonic: $P_i \propto 1/D_i$ (static)
- Earliest Deadline First: $P_i \propto 1/d_i$ (dynamic)

# Deadline Monotonic




**Problem with the Utilization Bound**

$$U_p = \sum_{i=1}^{n} \frac{C_i}{D_i} = \frac{2}{3} + \frac{3}{6} = 1.16 > 1$$

but the task set is schedulable.

# Response Time Analysis

An exact schedulability test for fixed priority algorithms requires computing the worst-case interference from high-priority tasks:

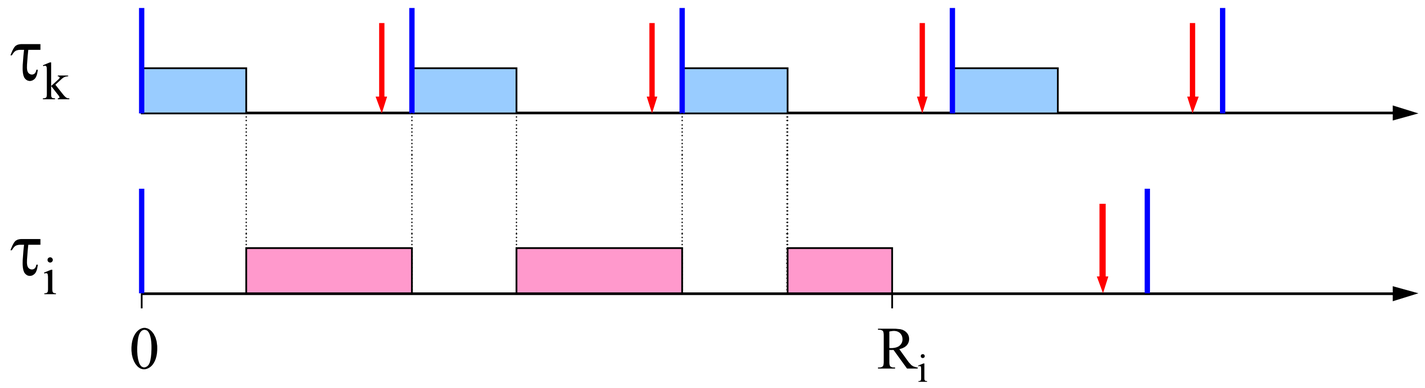
For each task $\tau_i$

- Compute the interference of higher priority jobs $\tau_{hj}$ on $\tau_i$ : $I_i = \sum_{\forall \tau_{hj} : D_{hj} < D_i} C_h$
- Compute its response time as $R_i = C_i + I_i$
- Verify that $R_i \leq D_i$

**Reference**

N. C. Audsley, A. Burns, M.F. Richardson, K. Tindell, and A.J.Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling", *Software Engineering Journal*, 8(5):284–292, September 1993.

# Computing Interference




Interference of $\tau_k$ on $\tau_i$ in the interval [0, $R_i$]:

$$I_{ih} = \left\lceil \frac{R_i}{T_h} \right\rceil C_h$$

Interference on $\tau_i$ by high-priority tasks:

$$I_i = \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil C_h$$

# Computing Response Times

$$R_i = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil C_h$$

**Iterative solution:**

$$\begin{cases} R_i^{(0)} = C_i \\ R_i^{(s)} = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i^{(s-1)}}{T_h} \right\rceil C_h \end{cases}$$

iterate while

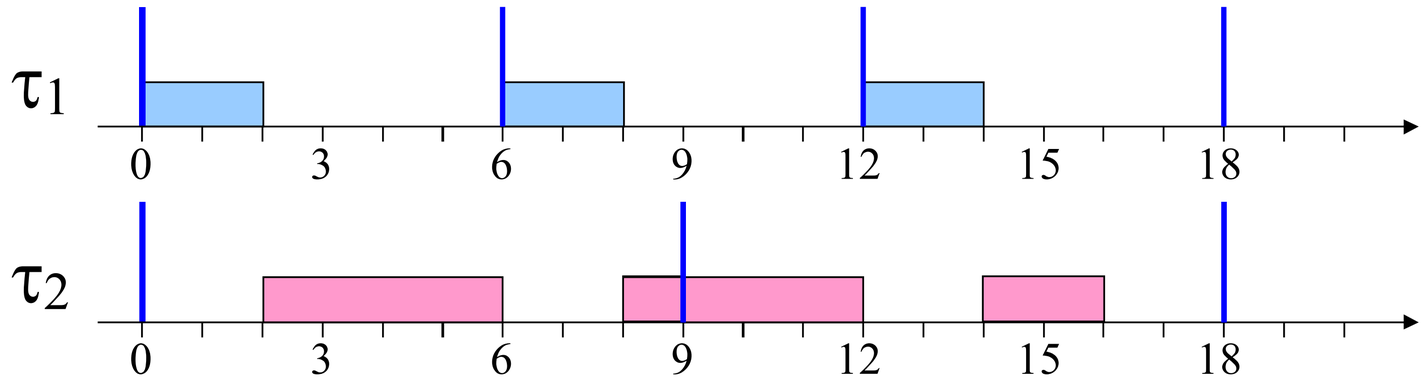
$$R_i^{(s)} > R_i^{(s-1)}$$

# Dynamic Priority Scheduling

# Example

$$U_p = \frac{2}{6} + \frac{5}{9} = 0.888$$

$D_i = T_i$

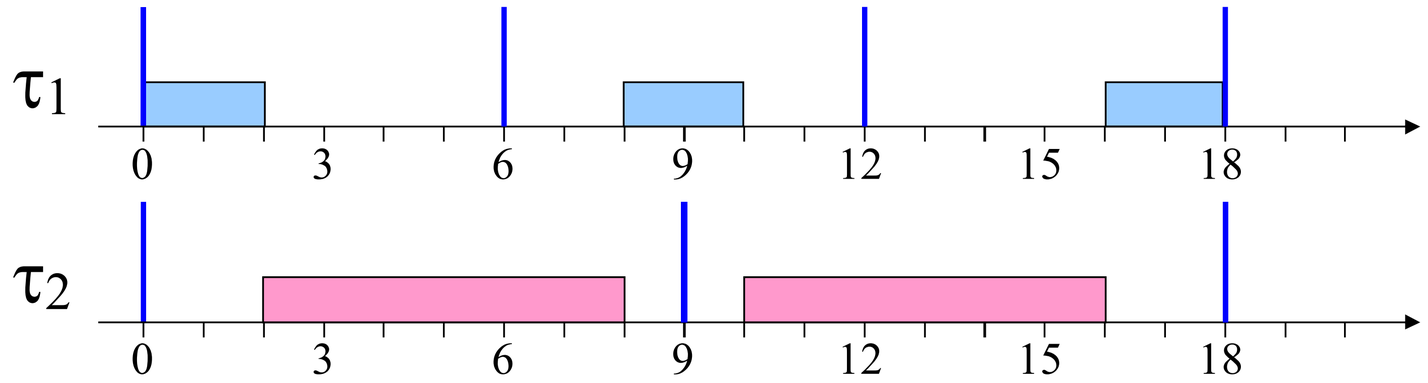



Under RM, this value of $U_p$ is an upper bound, since any increase in the computation times will cause a deadline miss.

# Example

$$U_p = \frac{2}{6} + \frac{6}{9} = 1$$

$D_i = T_i$




Under EDF, $U_p$ can reach 1 without causing a deadline miss.

# EDF Optimality

**EDF** is **optimal** among all algorithms:

If there exists a feasible schedule for a task set $\Gamma$, then EDF will generate a feasible schedule.

If $\Gamma$ is not schedulable by EDF, then it cannot be scheduled by any algorithm.

# EDF schedulability

In 1973, Liu and Layland proved that for a set of $n$ periodic tasks:

$$U_{lub}^{\text{EDF}} = 1$$

This means that a set $\Gamma$ of periodic tasks is schedulable by EDF if and only if $U_p \leq 1$.

**Reference**

C.L. Liu and J.W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment", *Journal of the Association for Computing Machinery*, 20(1), 1973.

# EDF with D<T

## Schedulability Analysis

**Processor Demand Criterion** [Baruah '90]

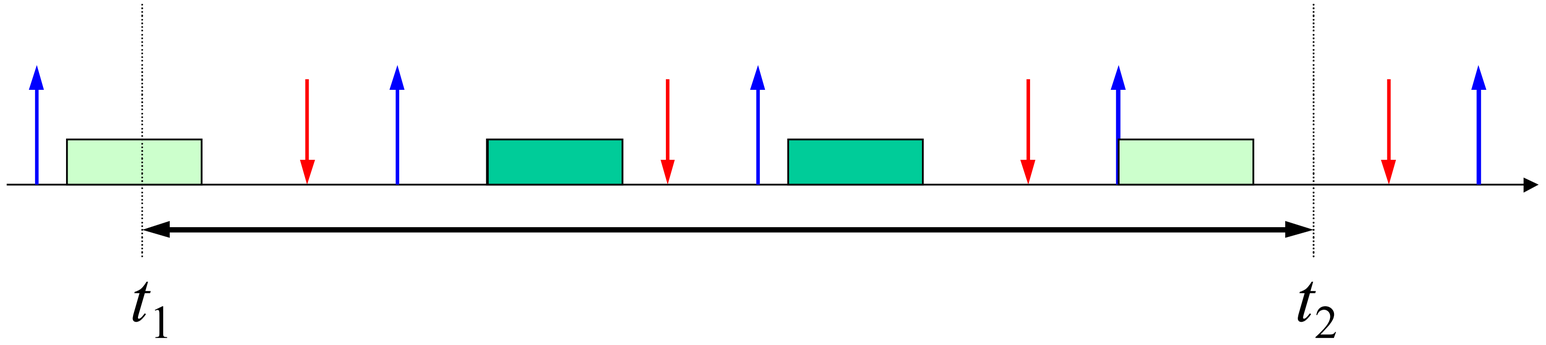
In any interval of length $L$, the computational demand $g(0, L)$ of the task set must be no greater than the available time in that interval.

$$\forall L > 0, \quad g(0, L) \leq L$$

**Reference**

S. K. Baruah, L. E. Rosier, and R. R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor", *Journal of Real-Time Systems*, 2, 1990.

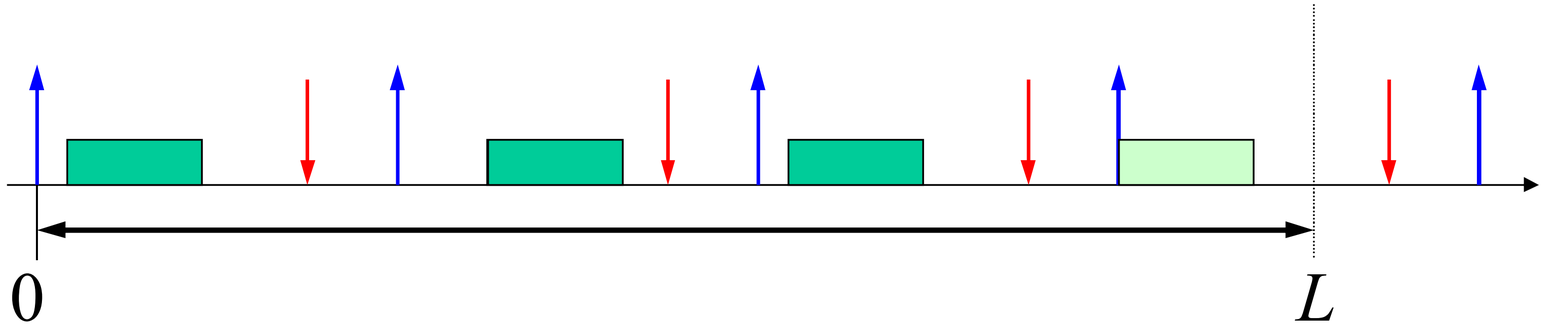
# Processor Demand




The computational demand in the interval $[t_1, t_2]$ is the computation time of those tasks started at or after $t_1$ with deadline less than or equal to $t_2$:

$$g(t_1, t_2) = \sum_{\substack{r_i \geq t_1 \\ d_i \leq t_2}} C_i$$

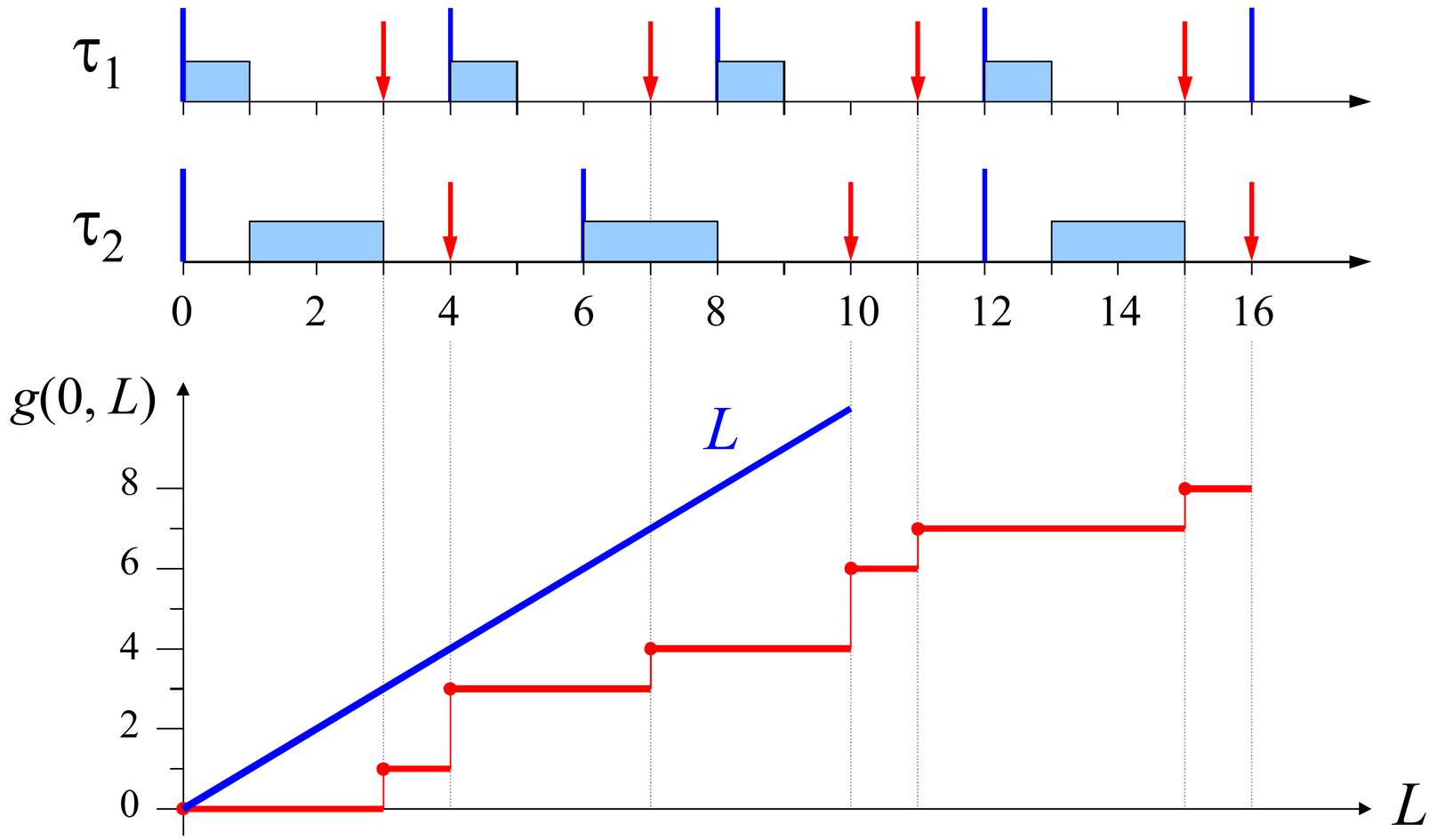
# Demand of a periodic task




$$g_i(0, L) = \left\lfloor \frac{L + T_i - D_i}{T_i} \right\rfloor C_i$$

$$g(0, L) = \sum_{i=1}^{n} \left\lfloor \frac{L + T_i - D_i}{T_i} \right\rfloor C_i$$

# Example



# Bounding complexity

- Since $g(0, L)$ is a step function, we can check feasibility only at deadline points.

- If tasks are synchronous and $U_p < 1$, then we can check feasibility up to the hyperperiod $H$:

$$H = \text{lcm}(T_1, \ldots, T_n)$$

# Bounding complexity

- Moreover, removing the floor operator in the demand bound function $g(0, L)$, we can find an upper bound $G(0, L)$:

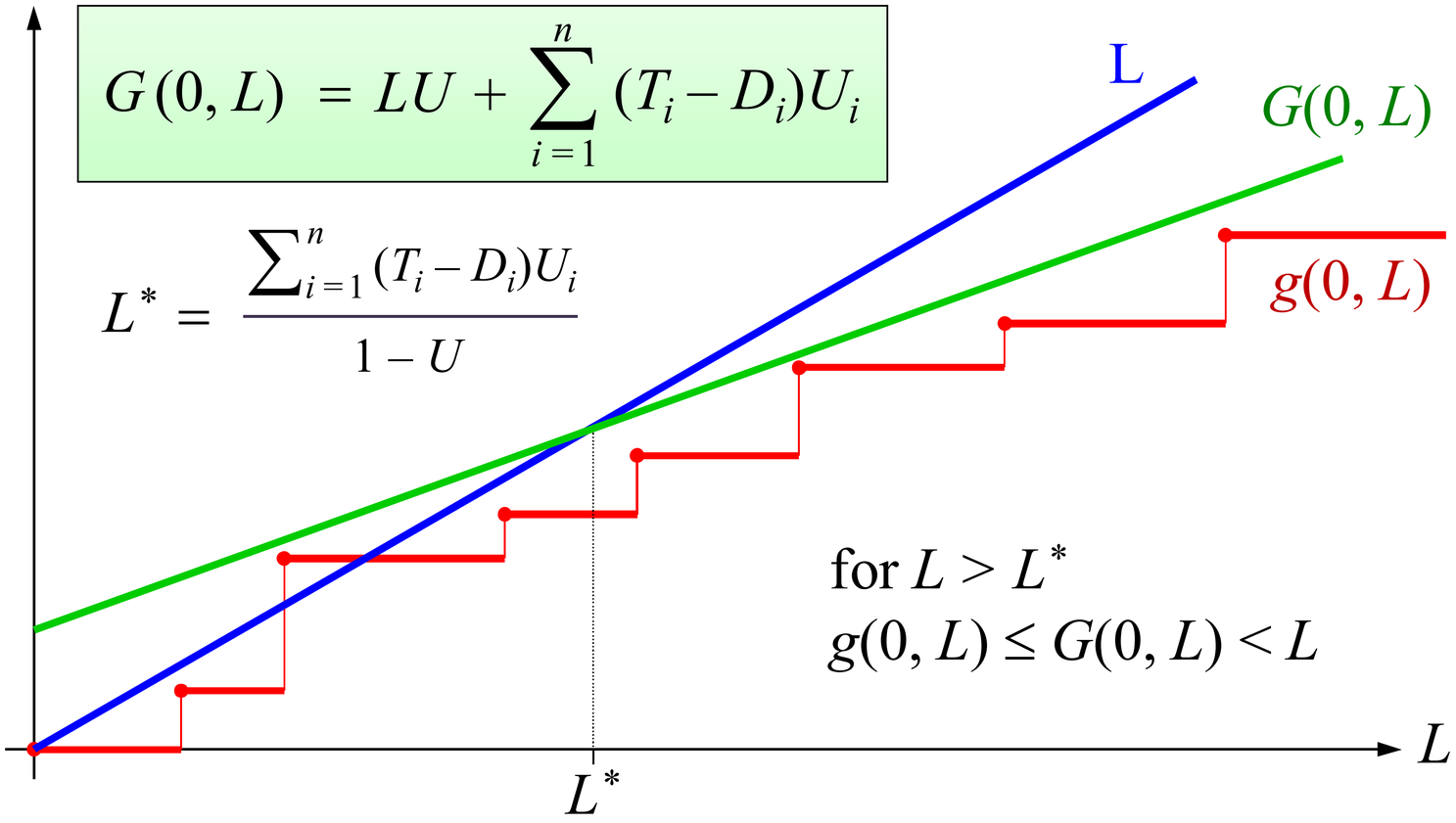
$$G(0, L) = \sum_{i=1}^{n} \left( \frac{L + T_i - D_i}{T_i} \right) C_i$$

$$= \sum_{i=1}^{n} L \frac{C_i}{T_i} + \sum_{i=1}^{n} (T_i - D_i) \frac{C_i}{T_i}$$

$$= LU + \sum_{i=1}^{n} (T_i - D_i) U_i$$

# Limiting L

Since $G(0, L)$ is a line with slope $U < 1$:



# Processor Demand Test

$$\forall L > 0, \qquad g(0, L) \leq L$$

$$\mathcal{D} = \{ d_k \mid d_k \leq \min(H, L^*) \}$$

$$\begin{cases} H = \text{lcm}(T_1, \ldots, T_n) \\ L^* = \dfrac{\sum_{i=1}^{n} (T_i - D_i) U_i}{1 - U} \end{cases}$$

# Summary

## Three scheduling approaches

- Off-line construction (Timeline)
- Fixed priority (RM, DM)
- Dynamic priority (EDF)

## Four analysis techniques

| | | Applicable to |
|---|---|---|
| Processor Utilization Bound: | $\sum U_i \leq U_{lub}$ | RM, EDF $(D_i = T_i)$ |
| Hyperbolic Bound: | $\prod(U_i + 1) \leq 2$ | RM $(D_i = T_i)$ |
| Response Time Analysis: | $\forall i \ R_i \leq D_i$ | RM, DM $(D_i \leq T_i)$ |
| Processor Demand Criterion: | $\forall L \ g(0,L) \leq L$ | EDF $(D_i \leq T_i)$ |

# Complexity Issues

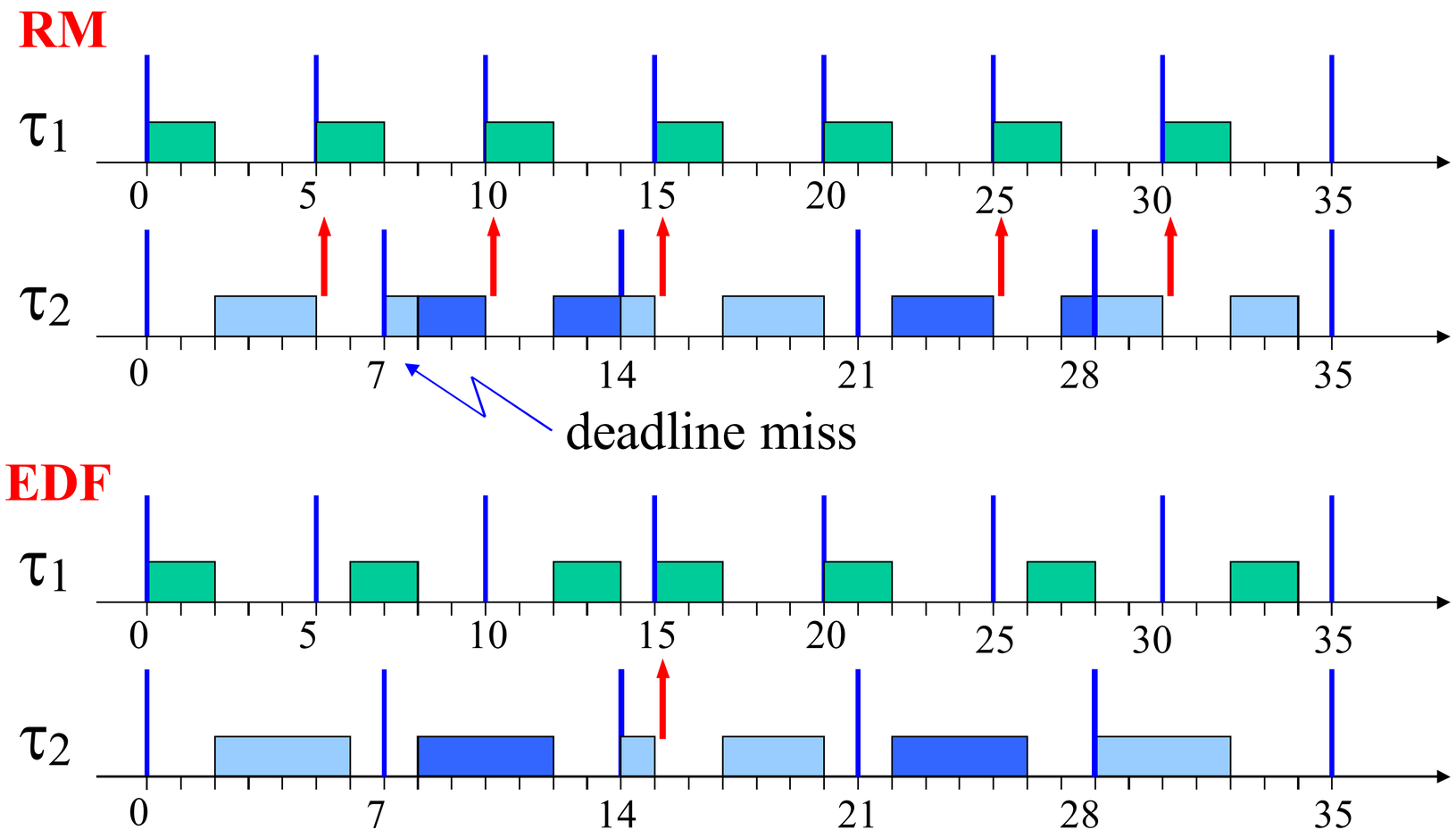
- **Utilization based analysis** $U \leq U_{lub}$
  - $O(n)$ complexity
- **Response time analysis** $\forall i \;\; R_i \leq D_i$
  - Pseudo-polynomial complexity
- **Processor demand analysis** $\forall L \;\; g(0,L) \leq L$
  - Pseudo-polynomial complexity

# RM vs. EDF

## Metrics

- Implementation complexity
- Efficiency
- Schedulability analysis
- Runtime overhead
- Overload conditions
- Jitter
- Aperiodic task handling

# Context switches



# Schedulability Analysis

| | $D_i = T_i$ | $D_i \leq T_i$ |
|---|---|---|
| **RM** | ***Suff.:*** ***polynomial*** $O(n)$<br>LL: $\sum U_i \leq n(2^{1/n}-1)$<br>HB: $\prod(U_i+1) \leq 2$<br>***Exact*** ***pseudo-polynomial***<br>RTA | ***pseudo-polynomial***<br>Response Time Analysis<br>$\forall i \quad R_i \leq D_i$<br>$R_i = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil C_h$ |
| **EDF** | ***polynomial:*** $O(n)$<br>$\sum U_i \leq 1$ | ***pseudo-polynomial***<br>Processor Demand Analysis<br>$\forall L > 0, \quad g(0, L) \leq L$ |

# Question

If EDF is more efficient than RM, why commercial RT operating systems are still based on RM?

**Main reason**

- RM is simpler to implement on top of commercial kernels, all based on fixed-priority scheduling.
- EDF requires explicit kernel support for deadline scheduling.

# RM: harmonic periods

Harmonic task sets are schedulable by RM

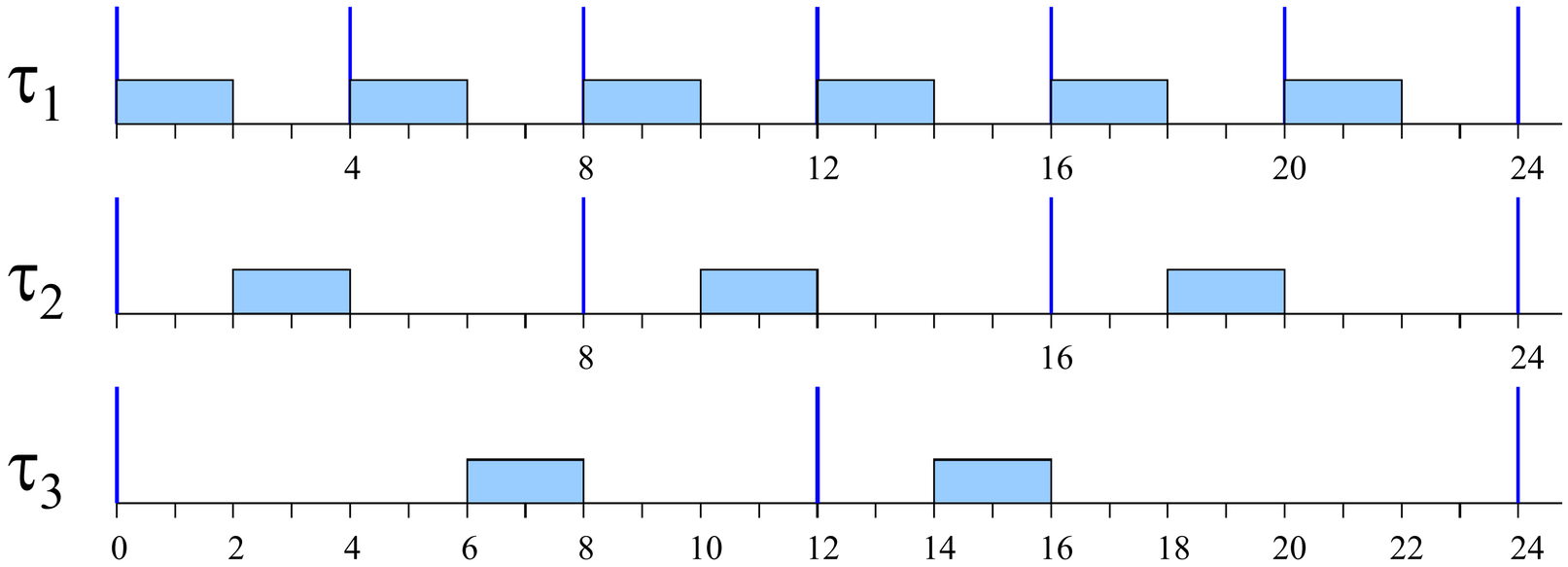
if and only if $U \leq 1$.

A set of tasks is harmonic if **every** pair of periods are in harmonic relation.

**A common misconception**

The RM schedulability bound is 1 if every period is multiple of the shortest period.

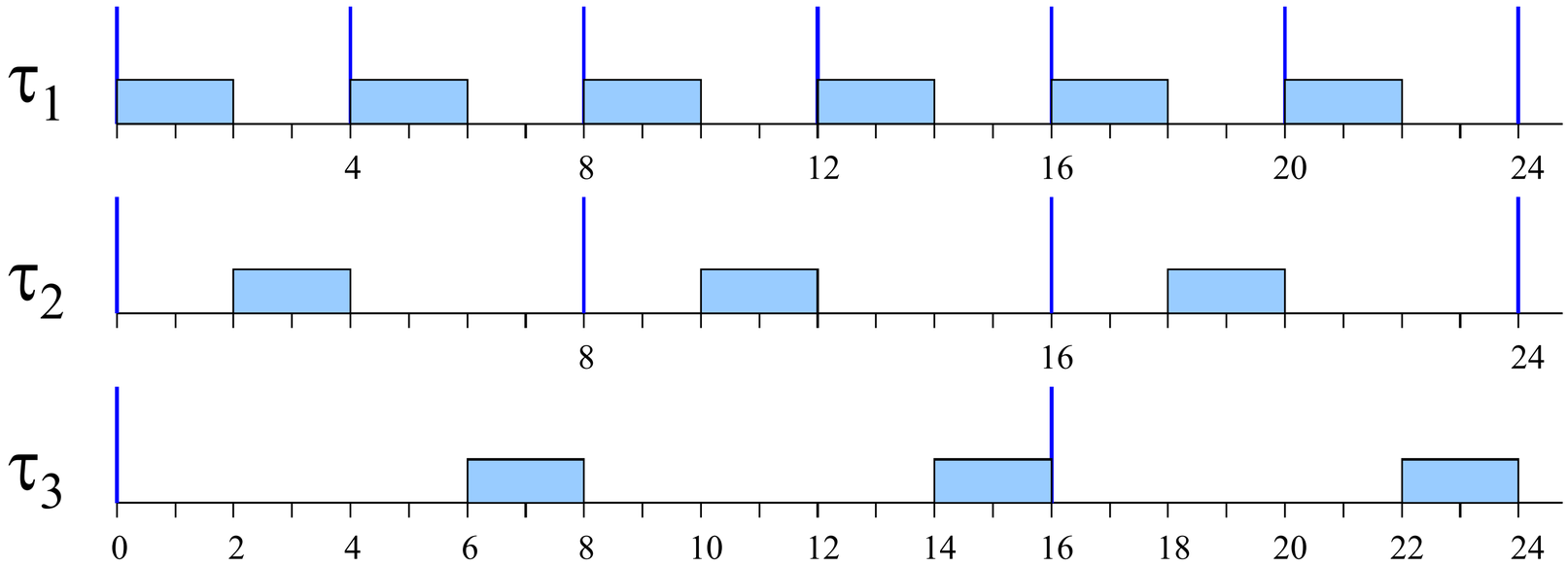
# Non harmonic periods




$$U = \frac{2}{4} + \frac{2}{8} + \frac{2}{12} \cong 0.917$$

Any increase in the $C_i$'s makes the system unschedulable

# Harmonic task set




$$U = \frac{2}{4} + \frac{2}{8} + \frac{4}{16} = 1$$

# Robustness under overloads

Two situations are considered:

**1. Permanent overload**

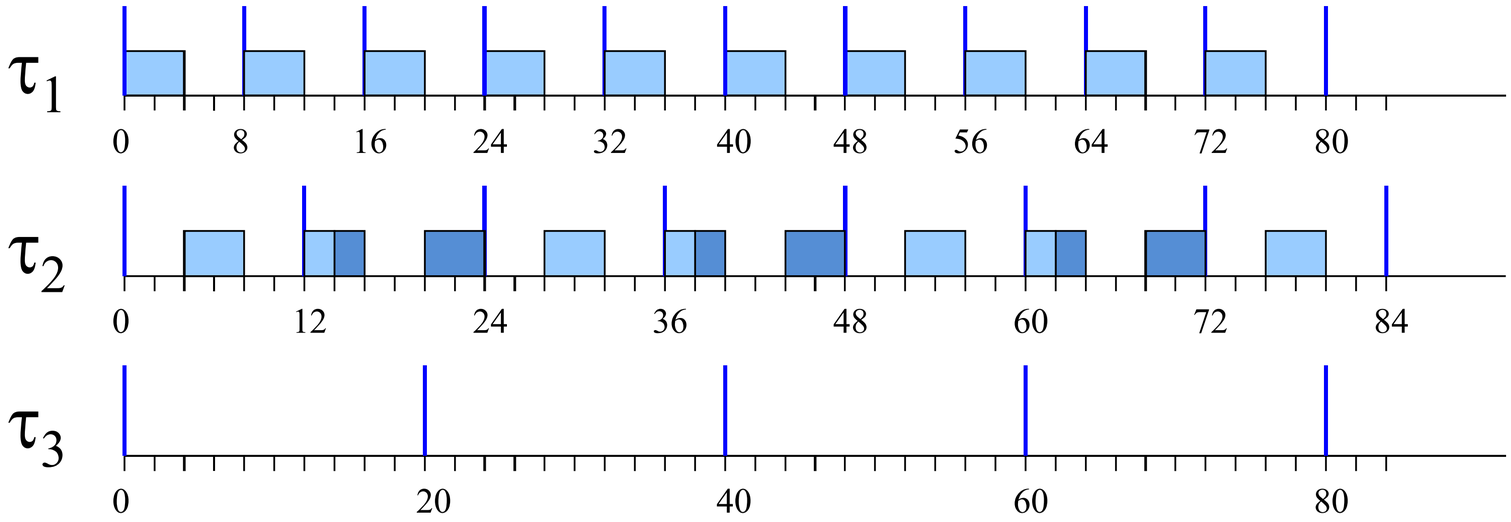
$\Rightarrow$ This occurs when $U > 1$

**2. Transient overload**

$\Rightarrow$ This occurs when some job executes more than expected

# RM under permanent overload

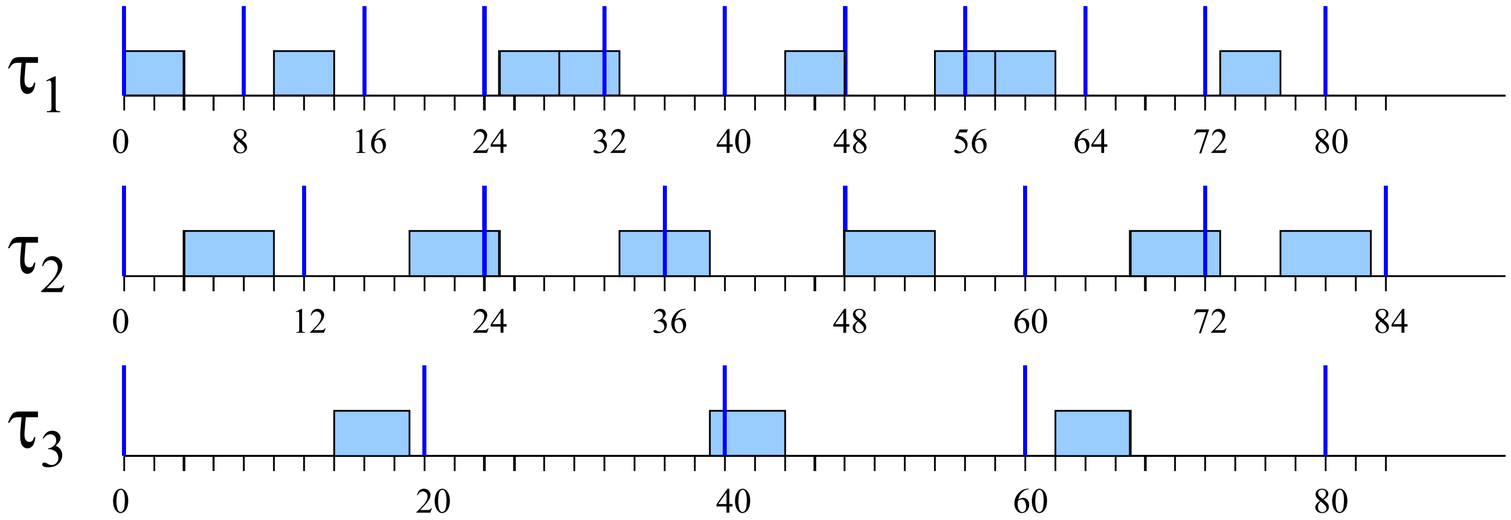
$$U = \frac{4}{8} + \frac{6}{12} + \frac{5}{20} = 1.25$$




- High priority tasks execute at the proper rate.
- Low priority tasks may miss deadlines (as $\tau_2$) or are completely blocked (as $\tau_3$).

# EDF under permanent overload

$$U = \frac{4}{8} + \frac{6}{12} + \frac{5}{20} = 1.25$$



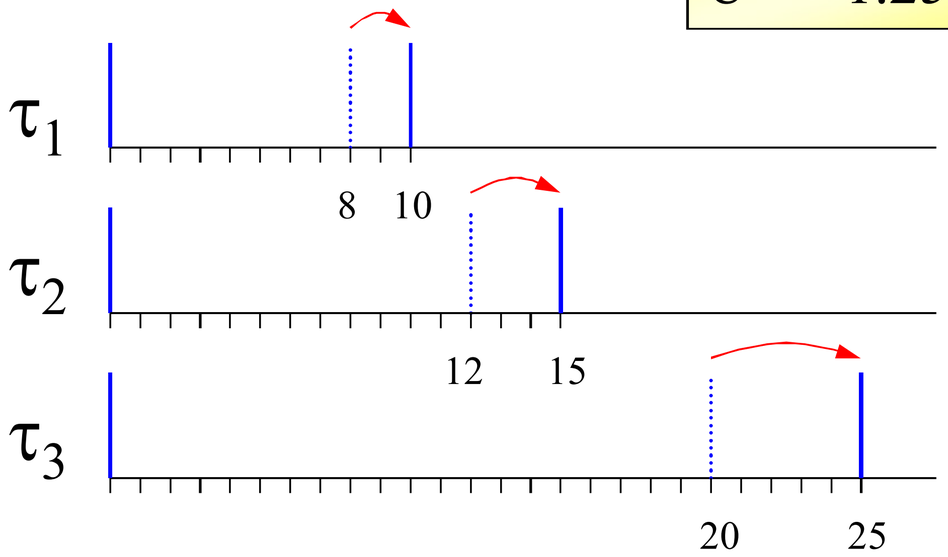

- All tasks execute at a slower rate
- No task is blocked

# EDF is predictable in overloads

**Theorem** (Cervin '03)

If $U > 1$, EDF executes each task $\tau_i$ with an average period $T_i' = T_i\, U$.

$U = 1.25$




| | $T_i$ | $T_i'$ |
|---|---|---|
| $\tau_1$ | 8 | 10 |
| $\tau_2$ | 12 | 15 |
| $\tau_3$ | 20 | 25 |

# EDF vs. RM

## Advantages of **EDF** vs. **RM**

- Processor utilization can always reach 100% (for $D_i = T_i$ ).
- Exact utilization test for $D_i = T_i$.
- Less overhead due to preemptions.
- Fair behavior during overloads.
- Better aperiodic responsiveness.

## Advantages of **RM** vs. **EDF**

- Processor utilization is 69% in the worst case, 88% in the average case (for $D_i = T_i$ ).
- Very easy implementation (just assign priorities to tasks proportional to their rates).
- During overloads, the highest priority task is guaranteed not to miss its deadline.