



# Real-Time Scheduling for Mono-Processor Systems

---

Laurent Pautet

strec.wp.imt.fr

pautet.wp.imt.fr/jobs

[Laurent.Pautet@enst.fr](mailto:Laurent.Pautet@enst.fr)

Version 3.2

# Real-Time Embedded Systems

## Examples

Airplane



Cellphone



Automated Subway



Nuclear Plant



?

What do they have in common ?



# Embedded Systems

## Definition and Requirements

---

An embedded system is a special-purpose system which software, hardware, mechanical, ... components are encapsulated in the device it controls

As opposed to general-purpose systems, they have specific properties such as low consumption, small size and weight, limited resources ...

A cruise control, a washing machine, factory robot, ...



# Real-Time Systems

## Definition and Requirements

---

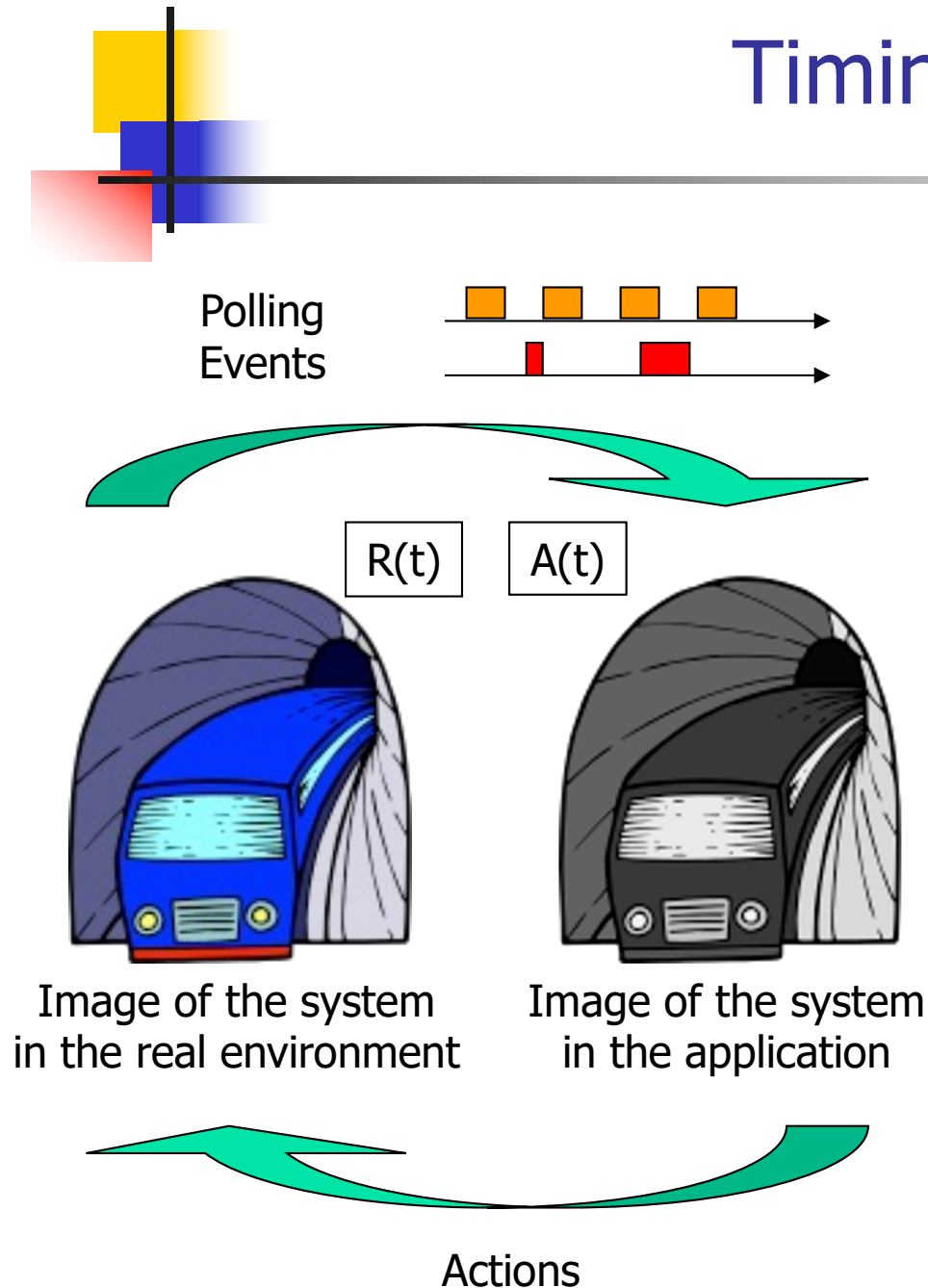
A real-time system consists in one or more sub-systems that have to react under specified time requirements to stimuli produced by the environment

A response after a deadline is invalid  
Even if the response is logically correct

A cruise control, a washing machine, factory robot, a nuclear plant, an air traffic control, trading center, ...

Most real-time systems are embedded systems

# Timing constraints



- The application must have a precise and consistent image of the system in its environment at anytime
- The goal of real-time systems is to minimize the difference between the images of the system in reality and in its application ( $|R(t)-A(t)| < \epsilon$ )
- To update the image in the application, it reads in particular sensors periodically. The period being a temporal granularity during which the measures evolve significantly)



# Non Fonctional Properties

---

## **These systems have to be predictable**

- **Reactivity and temporal consistency**
  - Define temporal interval during which data is valid
  - Define time granularity (ship sec, rail msec, airplane usec)
  - Guarantee response time boundaries (known in advance)
- **Reliability and Availability**
  - Guarantee the correctness of the computed data values
  - Enforce system availability in presence of hostile conditions (fault tolerance, malicious behavior ...)

Non-Fonctional Properties -> temporal & structural



# From requirements to technical solutions

---

- Requirements
  - Reactivity & temporal consistency
  - Reliability & Availability
- Solutions
  - Architectures and frameworks to help the design (Kernels, RT buses and networks, ...)
  - Models and methods to enforce predictability (RT scheduling, Fault tolerance, ...)
  - Suitable programming languages (C-Misra, Java-RT, Standard POSIX 1003.1c, Ada, ...)
  - Tools to integrate modeling, analysis and synthesis (AADL, Marte, verification, simulation, generation, testing)



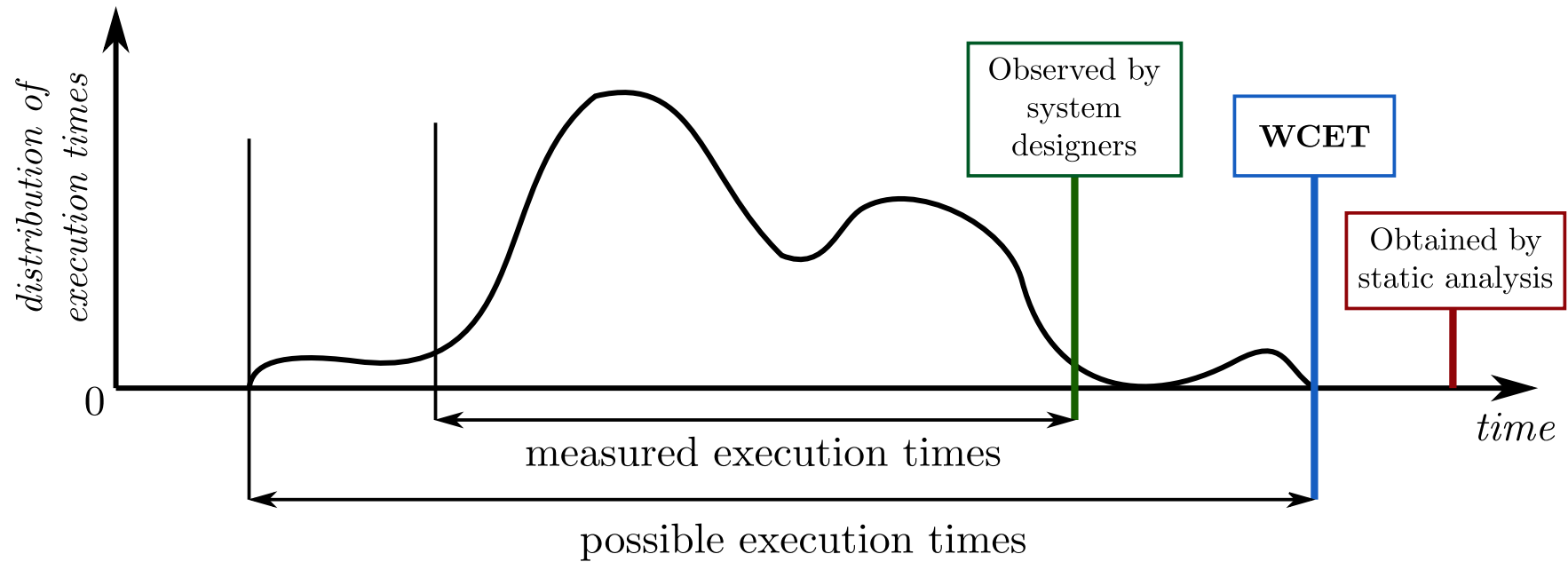
# Notations

---

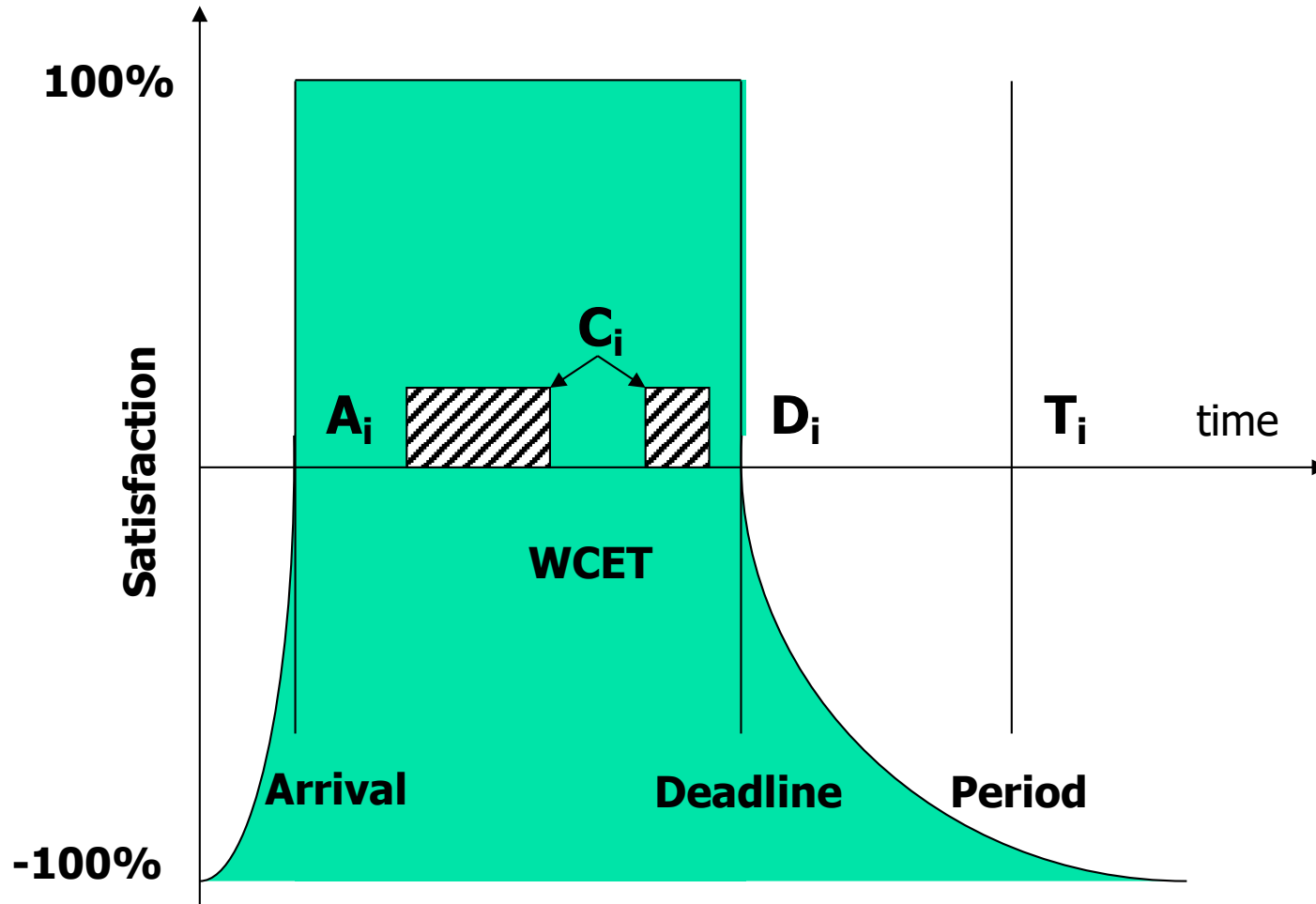
- Parameters of task  $t_i$ 
  - $C_i$  : Worst Case Execution Time (WCET) of task  $t_i$
  - $A_i$  : Arrival time of task  $t_i$ 
    - Task must not arrive before  $A_i$
    - $A_i$  may be different from 0 (dependency)
  - $T_i$  : Period of task  $t_i$
  - $D_i$  : Deadline of task  $t_i$ 
    - Task must not complete after  $D_i$
    - $A_i + C_i < D_i$  however ...
    - $D_i \leq T_i$  is not mandatory (constrained deadline)
  - $U_i = C_i / T_i =$  processor utilisation of task  $t_i$
- Operators
  - Ceiling  $\lceil x \rceil$  (least integer greater than or equal to  $x$ )
  - Floor  $\lfloor x \rfloor$  (greatest integer less than or equal to  $x$ )



# Worst Case Execution Time evaluation

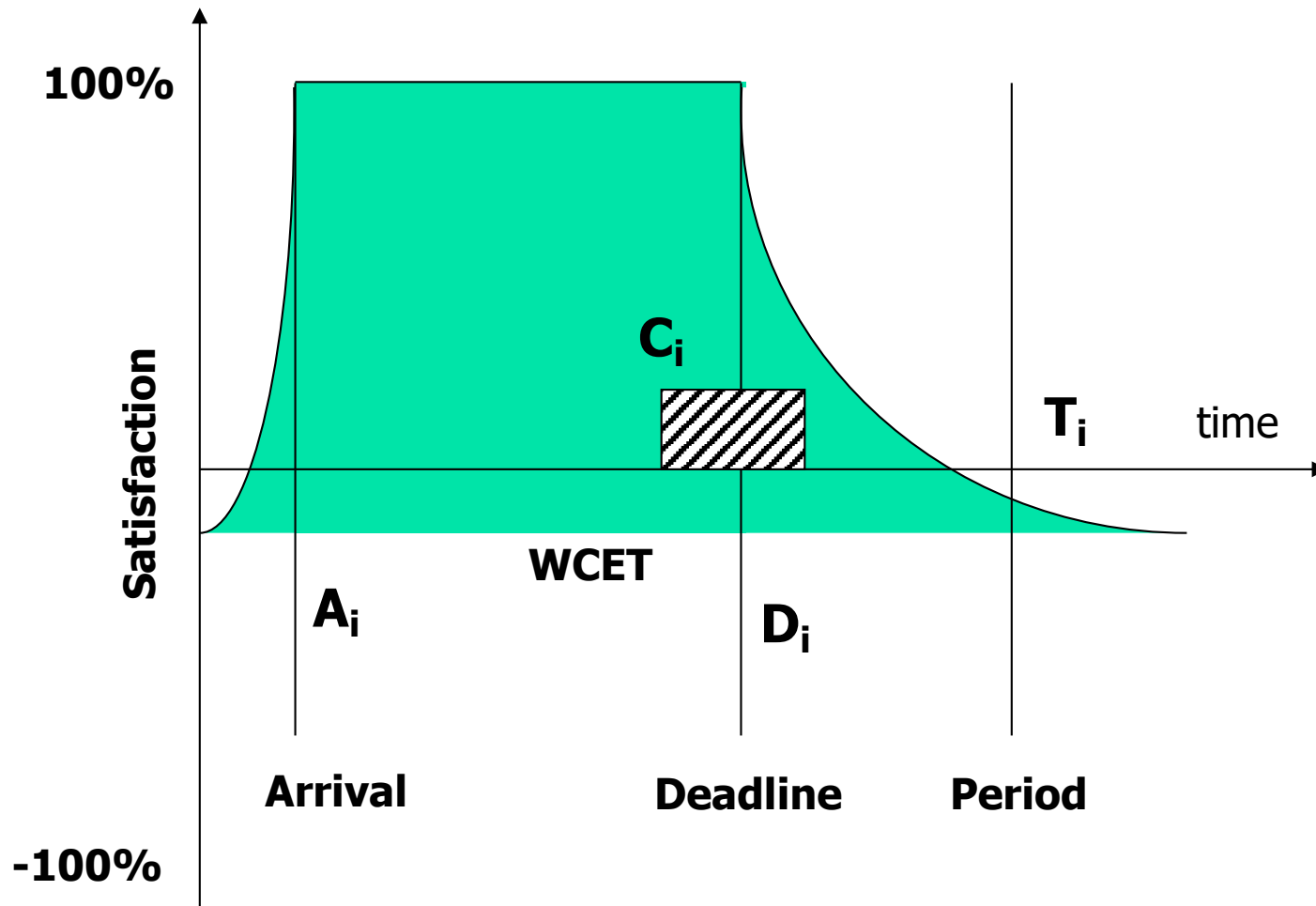


# Hard Real-Time Task



# Soft Real-Time Task

(Different from Best Effort Task that has No Deadline)





# Hard real-time vs soft real-time tasks

---

- For a hard real-time task
  - Deadlines must always be fulfilled
  - WCET must never be overrun (over-dimension)
  - Reduce inaccuracy from non-determinism
    - Pre-allocation (no dynamic allocation), ...
    - Non-predictible cache behavior, ...
- For a soft real-time task
  - Missing deadlines can be tolerated:
    - For a given percentage of times
    - For a given number of times
    - For a given frequency
  - And result in a degraded execution mode
    - Reuse previous job results for instance



# Sub-systems of real-time systems

---

A real-time system is composed of several sub-systems with different real-time properties

Some of these sub-systems may be non real-time, soft real-time or hard real-time sub-systems

- Hard real-time tasks must fulfill their deadlines.
- Soft real-time tasks may fail to fulfill their deadlines. If so, they may execute in a degraded mode.
- Other tasks execute in best-effort mode.



# Real-Time Schedulers

---

- Allocate (CPU) resources to guarantee safety (timing) properties
- In normal mode, respect the time constraints of all tasks
- Otherwise, limit the effects of time overflows and ensure compliance with the constraints of the most critical tasks

In the following, it will be ensured that the time dedicated to the scheduling algorithm and that of context switch are negligible which implies a low complexity and an effective implementation



# Definitions

## Execution Model

---

- Dependent or independent tasks
  - Independent tasks sharing only the processor
  - Dependent tasks with shared resources or linked by precedence constraints
- Synchronous tasks have the same activation time (0)
- Periodic task with implicit deadline means periodic task with deadline equals to period
- Task job : instantiation of a task during period
- Worst Case Execution Time : worst computation time
- Response time : time to complete a job while other jobs are also running on the same processor



# Definitions

---

- Preemptive and non-preemptive scheduling
  - A preemptive scheduler can interrupt a task for a higher priority task when a non-preemptive scheduler executes the task until it completes
- Offline or online scheduling
  - A scheduler decides offline or online when and which task to execute
- Optimal scheduling
  - Algorithm that produces a schedule for any set of schedulable tasks (if an algorithm does, it does too)
- Scheduling test
  - A necessary and / or sufficient condition for an algorithm to satisfy the temporal constraints of a set of tasks





# Overview of algorithms

---

- Scheduling periodic tasks
  - Non-preemptive table-based scheduling
  - Preemptive scheduling with static priorities
    - Rate and Deadline Monotonic Scheduling
  - Preemptive scheduling with dynamic priorities
    - Earliest Deadline First and Least Laxity First
- Scheduling aperiodic tasks
  - Background, polling, deferred & sporadic servers
- Sharing resources
  - Priority Inheritance, Priority Ceiling & Highest Locker



# Proving schedulability using a scheduling algorithm

---

- To prove schedulability/feasibility of a task set
  - Execute the algorithm over a feasibility interval
  - Compute a (necessary - sufficient) scheduling test
  - Compute response time & check against deadlines
- Feasibility interval : minimum interval needed to verify the schedulability of a system
  - For independent synchronous periodic tasks :  
 $\forall i: D_i \leq T_i$  with a fixed priority scheduling  
[0, LCM ( $\forall i: T_i$ )] (LCM or hyper-period)
  - For independent asynchronous periodic tasks  
 $\forall i: D_i \leq T_i$  with a fixed priority scheduling  
[0, 2 \* LCM ( $\forall i: T_i$ ) + max ( $\forall i: A_i$ )]



# Table Driven Scheduling

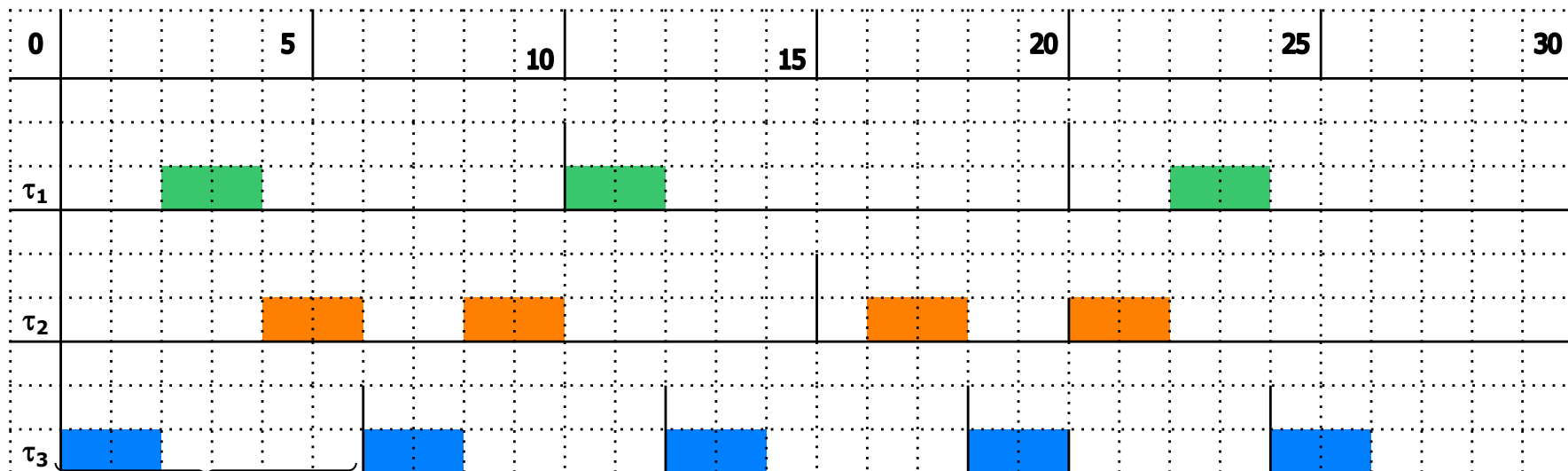
## Principles

---

- Hypotheses
  - Periodic tasks
- Principles
  - Major cycle = LCM of the task periods
  - Minor cycle = non-preemptible block
  - The minor cycle divides the major cycle
  - A cyclic scheduler loops on the major cycle by executing the sequence of minor cycles
  - The minor cycle provides a control point to check the respect of the timing constraints

# Table Driven Scheduling Example

	Period	Deadline	WCET	Usage
$\tau_1$	10	10	2	0,200
$\tau_2$	15	15	4	0.267
$\tau_3$	6	6	2	0.333



Minor Cycle

Major Cycle



# Table Driven Scheduling

## Advantages and Disadvantages

---

- Advantages
  - Effective implementation
  - No need for mutual exclusion between tasks
- Disadvantages
  - Not work conserving :
    - the processor may be idle while jobs are not completed
  - Impact of an additional task
  - Execution of aperiodic tasks
  - Difficult construction of the table
    - Allocating slots is a complex problem (NP-hard)

# Static Priority Scheduling

## Highest Priority First

- Offline, each task is assigned a priority (integer number) before runtime
- Online, the scheduler always executes the task of the ready tasks list with the highest priority
- The scheduler can preempt the current task to execute a new task that has just been activated
- There are many algorithms to assign offline priorities to tasks (mostly based on their timing parameters)
- The objective is to find a mapping that makes the task set schedulable

# Static Priority Scheduling

## Response Time

- The *critical instant* for a set of synchronous periodic tasks is when all jobs start at the same time
- For each task, compute time  $t$  at which its first activation completes by integrating the execution of highest priority tasks activated in the mean time
  - Start with a first response time  $R^i_0 = C_i$
  - Compute  $R^i_{n+1} = \sum_{j \leq i} C_j * \lceil R^i_n / T_j \rceil$  to integrate the execution of the tasks of highest priority
  - Iterate until a fixed point is reached
  - In other word :  $\forall i, 1 \leq i \leq n, \exists t \leq D_i W_i(t) = \sum_{j \leq i} C_j * \lceil t / T_j \rceil \leq t$
- The task is schedulable if the response time is a fixed value less than or equal to the deadline
- Valid for any static priority scheduling

# Static Priority Scheduling

## Response Time (RMS)

	T	C	P
$\tau_1$	3	1	3
$\tau_2$	5	2	2
$\tau_3$	15	4	1

1. Check for  $\tau_1$ 
  1.  $R_0 = C_1 = 1$
  2.  $R_1 = \text{Response}(R_0) = 1*1 = 1$



2. Check for  $\tau_2$  and take into account  $\tau_1$ 
  1.  $R_0 = C_2 = 2$



1.  $R_1 = \text{Response}(R_0) = 1*1 + 1*2 = 3$
2.  $R_2 = \text{Response}(R_1) = 1*1 + 1*2 = 3$





# Static Priority Scheduling

## Response Time (RMS)

	T	C	P
$\tau_1$	3	1	3
$\tau_2$	5	2	2
$\tau_3$	15	4	1

1. Check for  $\tau_3$  and take into account  $\tau_1$  and  $\tau_2$

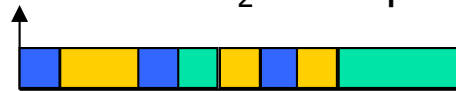
1.  $R_0 = C_3 = 4$



2.  $R_1 = \text{Response}(R_0) = 2*1 + 1*2 + 1*4 = 8$



3.  $R_2 = \text{Response}(R_1) = 3*1 + 2*2 + 1*4 = 11$



4.  $R_3 = \text{Response}(R_2) = 4*1 + 3*2 + 1*4 = 14$



5.  $R_4 = \text{Response}(R_3) = 5*1 + 3*2 + 1*4 = 15$

6.  $R_5 = \text{Response}(R_4) = 5*1 + 3*2 + 1*4 = 15$





# Static Priority Scheduling

## OPA – Optimal Priority Assignment

---

- Let have  $N$  fixed priority tasks
- Among these tasks, find a task that can have the lowest priority ...
  - Its response time should be less than its deadline when all the others have a higher priority
  - If there is such a task, give it the lowest priority
  - Otherwise, the system is not schedulable
- Repeat process with the  $N-1$  remaining tasks



# Static Priority Scheduling

## Rate Monotonic Scheduling

---

- Hypotheses

- Synchronous, deadline implicit & independent tasks
- Synchronous ( $A_i = 0$ )
- Deadline implicit ( $D_i = T_i$ )

- Principle

- Task activation or completion wake up the scheduler
- Select the ready task with the shortest period

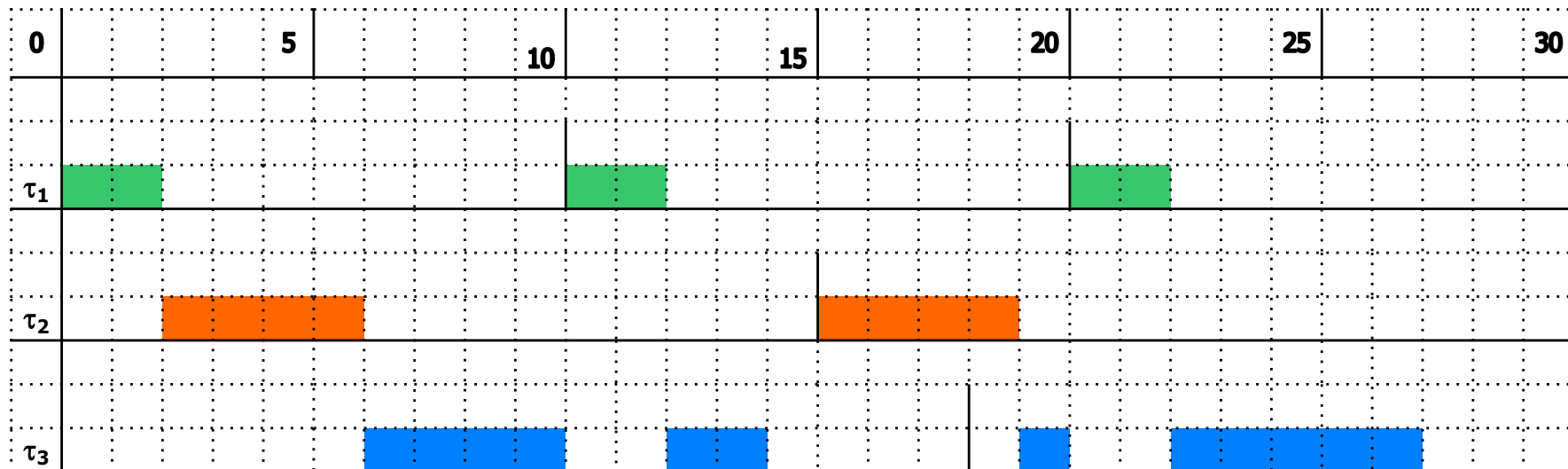
- Scheduling test

- Necessary condition:  $\sum C_i/T_i \leq 1$
- Sufficient condition:  $\sum C_i/T_i \leq n (2^{1/n} - 1)$   
→  $\log(2) = 69\%$

# Static Priority Scheduling

## Rate Monotonic Scheduling

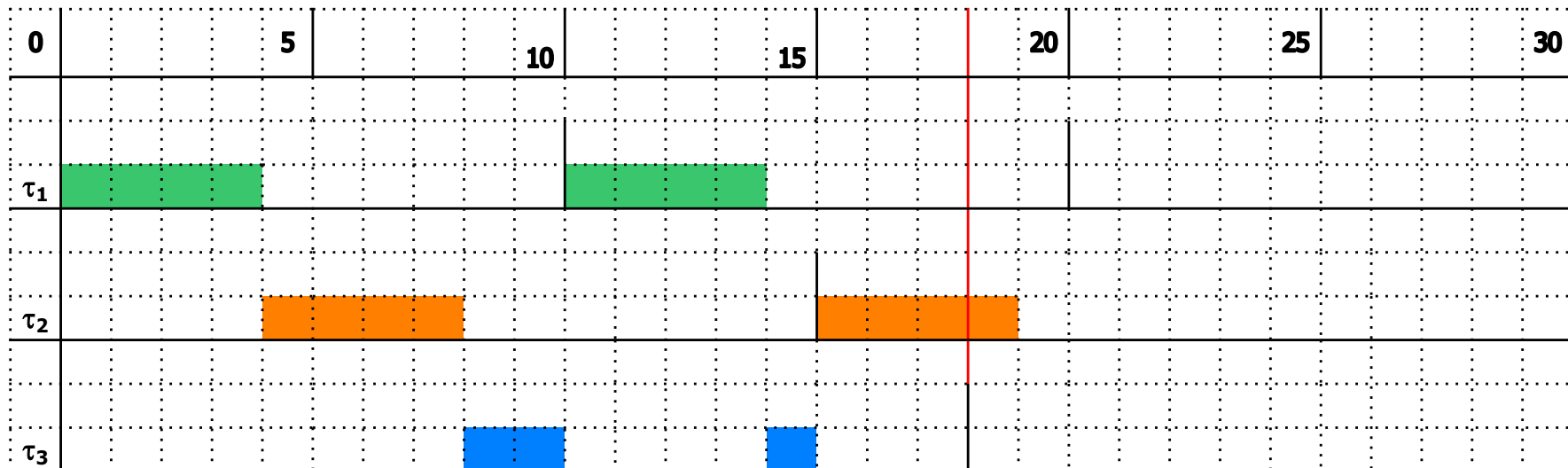
$3(2^{1/3}-1)=78\%$	Period	WCET	Usage
$\tau_1$	10	2	0.200
$\tau_2$	15	4	0.267
$\tau_3$	18	6	0.333



# Static Priority Scheduling

## Rate Monotonic Scheduling

$3(2^{1/3}-1)=78\%$	Period	WCET	Usage
$\tau_1$	10	4	0.400
$\tau_2$	15	4	0.267
$\tau_3$	18	6	0.333

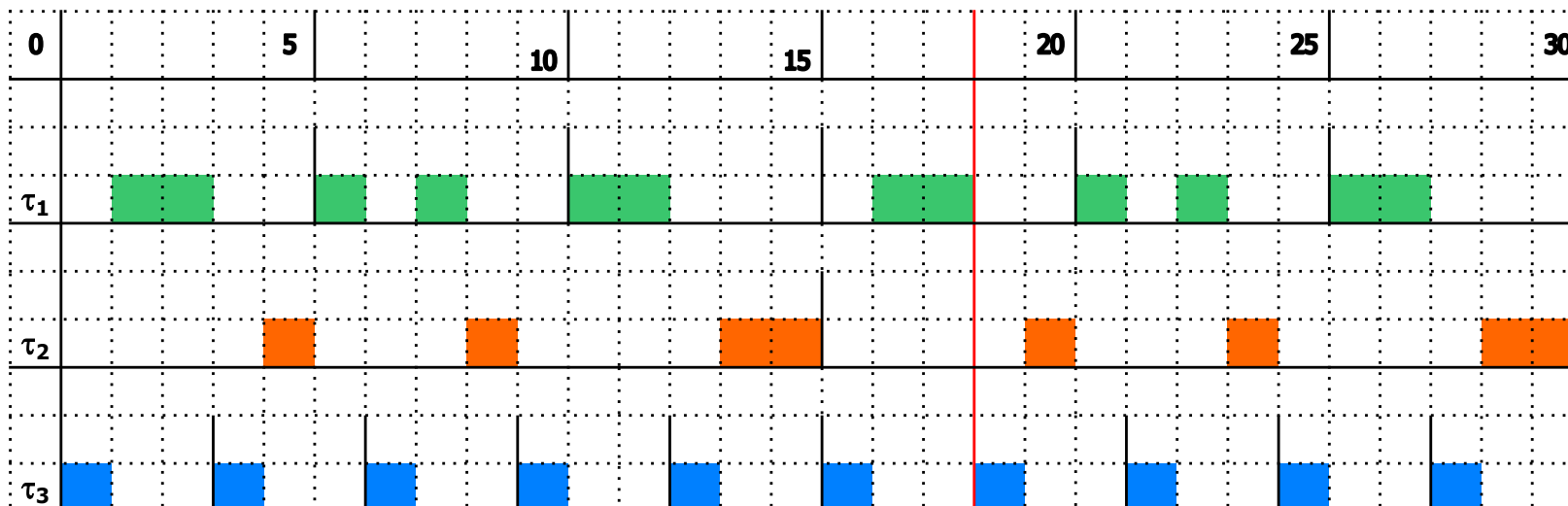


# Static Priority Scheduling

## Rate Monotonic Scheduling

$3x(2^{1/3}-1)=0.78$	Period	WCET	Usage
$\tau_1$	10	4	0.400
$\tau_2$	15	4	0.267
$\tau_3$	18	6	0.333

$3x(2^{1/3}-1)=0.78$	Period	WCET	Usage
<b><math>\tau'_1 = \tau_1 / 2</math></b>	<b>5</b>	<b>2</b>	<b>0.400</b>
$\tau'_2$	15	4	0.267
<b><math>\tau'_3 = \tau_3 / 6</math></b>	<b>3</b>	<b>1</b>	<b>0.333</b>





# Static Priority Scheduling

## Rate Monotonic Scheduling

---

- Advantages
  - Easy to implement
  - Optimal for static priority scheduling
  - Frequent in the classic executives
  - Good behavior in case of overload
- Disadvantages
  - Possible oversizing of the system
- RMS is always a possible result of OPA
  - Both RMS and OPA are optimal

# OPA vs RMS

$3x(2^{1/3}-1)=0.78$	Period	WCET	Usage
$\tau_1$	5	2	0.400
$\tau_2$	15	4	0.267
$\tau_3$	3	1	0.333

- $\tau_1$  lowest priority:  $R_0 = 2$ ;  $R_1 = 7$ ; or  $R_1 > T_1$
- $\tau_2$  lowest priority:  $R_0 = 4$ ;  $R_1 = 8$ ;  $R_2 = 14$ ;  $R_3 = 15$ ;
  - $\tau_1$ :  $R_0 = 2$ ;  $R_1 = 3$ ;  $\tau_2 < \tau_1 < \tau_3$ : same as RMS
  - $\tau_3$ :  $R_0 = 1$ ;  $R_1 = 3$ ;  $\tau_2 < \tau_3 < \tau_1$ : different from RMS
- $\tau_3$  lowest priority:  $R_0 = 1$ ;  $R_1 = 7$ ; or  $R_1 > T_3$
- OPA always finds an assignment if it exists (optimal), in particular the assignment of RMS (also optimal)





# Static Priority Scheduling

## Deadline Monotonic Scheduling

---

- Hypotheses
  - Synchronous and independent tasks
  - The deadline is less than the period ( $D_i \leq T_i$ )
- Principle
  - Select the ready task with the shortest deadline
  - When for all tasks  $T_i = D_i$ , DMS becomes RMS
- Scheduling test
  - The necessary and sufficient condition exists
  - Sufficient condition:  $\sum C_i/D_i \leq n (2^{1/n} - 1)$   
(we oversize : the task period is its deadline)



# Static Priority Scheduling

## Deadline Monotonic Scheduling

---

- Advantages
  - See RMS
  - RMS penalizes long period but short deadline tasks
  - DMS is better in this case.
- Disadvantages
  - See RMS
  - Do not to be confused with EDF

# Dynamic Priority Scheduling

## Earliest Deadline First

- Hypotheses

- Periodic, independent tasks
- Deadline implicit ( $D_i = T_i$ ) or not ( $D_i \leq T_i$ )

- Principle

- Task activation or completion wake up the scheduler
- Select the ready task with the earliest deadline

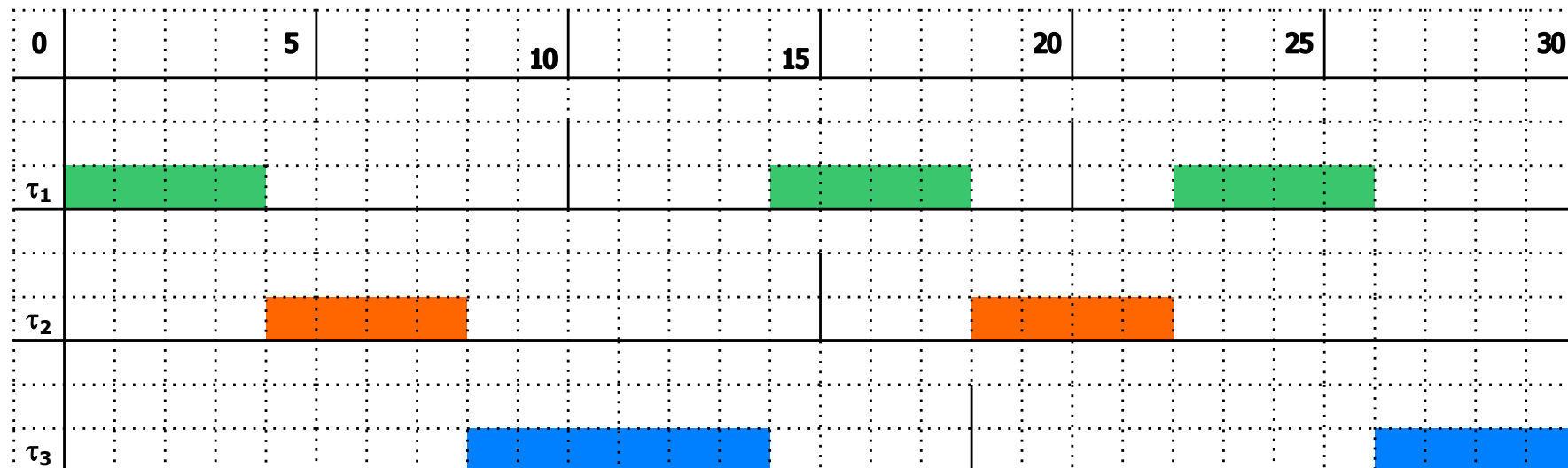
- Scheduling test

- Necessary and sufficient condition :  $\sum C_i / T_i \leq 1$
- Sufficient when not implicit ( $D_i \leq T_i$ ) :  $\sum C_i / D_i \leq 1$

# Dynamic Priority Scheduling

## Earliest Deadline First

	Period	WCET	Usage
$\tau_1$	10	4	0.400
$\tau_2$	15	4	0.267
$\tau_3$	18	6	0.333





# Dynamic Priority Scheduling

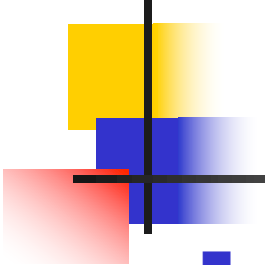
## Earliest Deadline First

---

- Advantages
  - Possible use of 100% of the processor
  - Optimal for dynamic priority scheduling if the deadlines are lower than the periods
- Disadvantages
  - Slight complexity of implementation
  - Less common in executives than RMS
  - Bad behavior in case of overload
- Remarks
  - If  $D_i$  is arbitrary compared to  $T_i$ , the necessary and sufficient condition is no longer sufficient.

# Dynamic Priority Scheduling

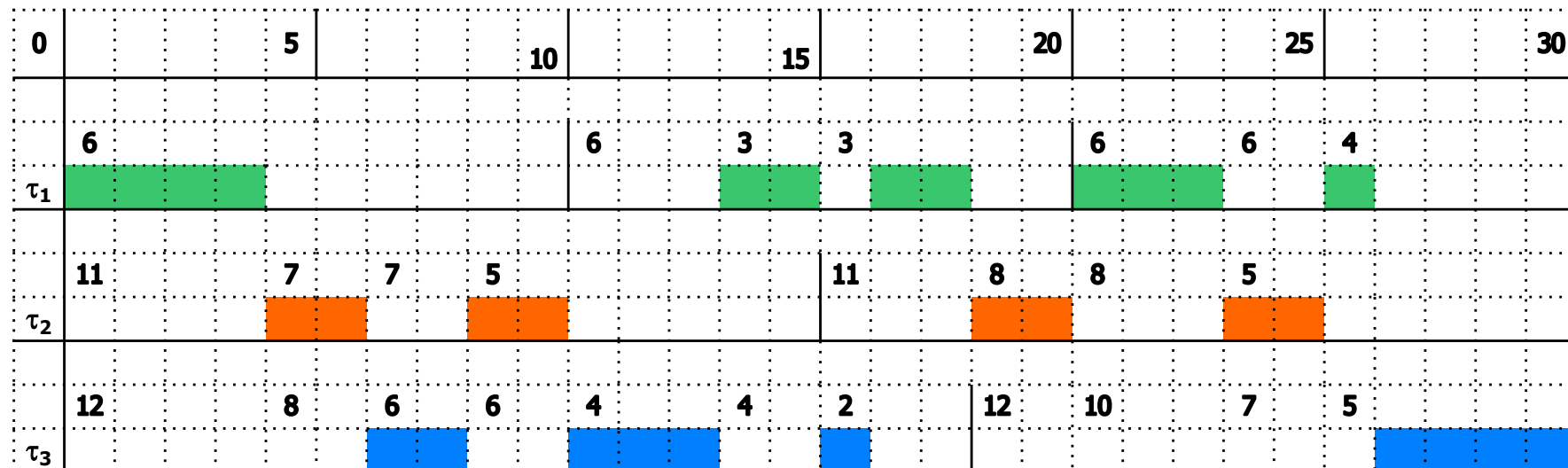
## Least Laxity First

- 
- Hypotheses
    - Similar to those of EDF
  - Principle
    - Task activation or completion wake the scheduler
    - Select the ready task with the lowest margin
    - margin = deadline – remaining comp. time – current time
  - Scheduling test
    - Necessary and sufficient condition:  $\sum C_i/T_i \leq 1$

# Dynamic Priority Scheduling

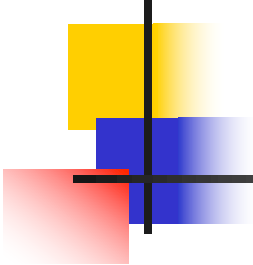
## Least Laxity First

	Period	WCET	Usage
$\tau_1$	10	4	0.400
$\tau_2$	15	4	0.267
$\tau_3$	18	6	0.333



# Dynamic Priority Scheduling

## Least Laxity First

- 
- Advantages
    - Better than EDF in the case of multi-processor
  - Disadvantages
    - High complexity of implementation
    - Complex to compute remaining execution time
    - Bad behavior in case of overload
    - High number of preemptions
    - LLF oscillates in case of tied-laxities tasks





# Aperiodic Task Scheduling

---

## ■ Definitions

- Aperiodic tasks are activated at arbitrary instants
- Sporadic tasks are aperiodic tasks activated with a minimum delay between two activations
- Sporadic tasks are almost periodic as they are activated with a variable but minimal period
- Aperiodic tasks must respect their deadlines

## ■ Principles

- Aperiodic tasks must be integrated into the scheduling of periodic tasks



# Aperiodic tasks with periodic tasks

---

- First solution for sporadic tasks
  - Handle sporadic tasks as periodic tasks when scheduling algorithm **supports** it
  - Ie the scheduling algorithm accepts tasks that are not activated at fixed time
- Second solution (more general)
  - Handle aperiodic tasks with a periodic server
  - The periodic server when it is active handles the aperiodic tasks as long as it is allowed
- Reuse schedulability tests for periodic tasks

# Scheduling aperiodic tasks

## Background server

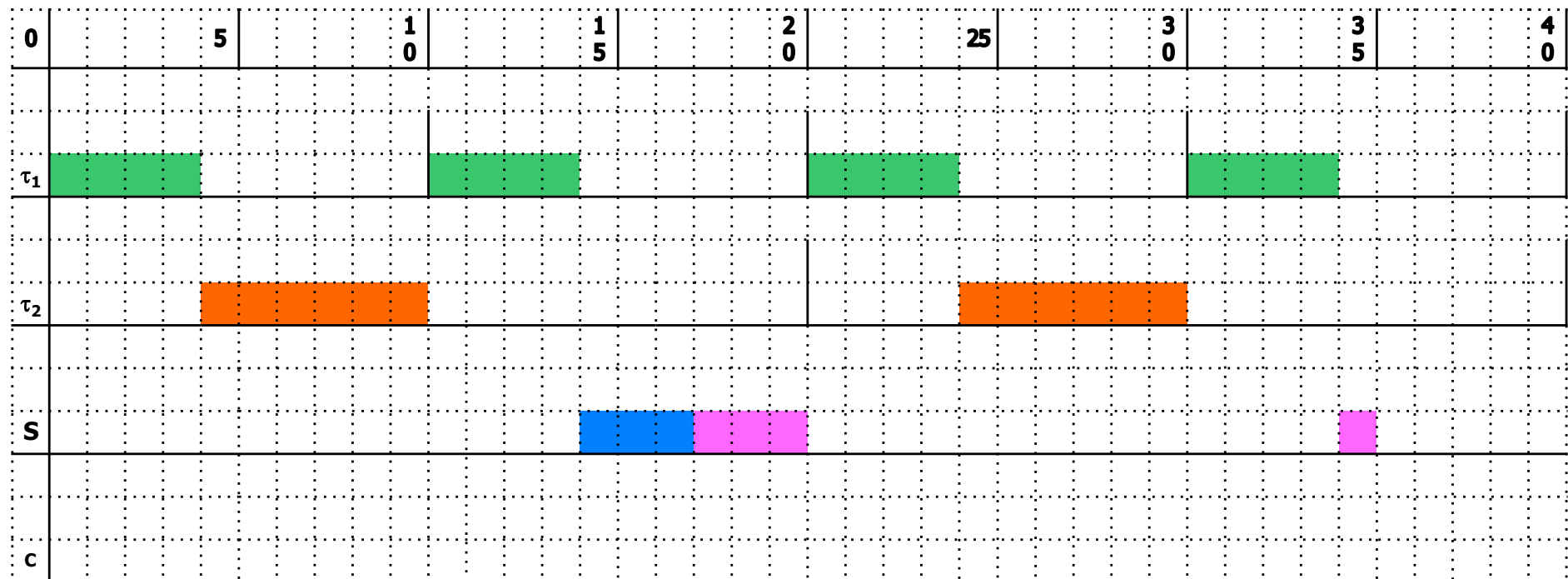
---

- The aperiodic tasks are processed sequentially by a low priority server
- The server has no associated budget (since it has the lowest priority)
- The lack of budget comes from the fact that the server fills the holes in the scheduling

# Scheduling aperiodic tasks

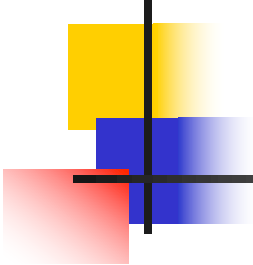
## Background server

	Period/Release	Budget	Priority	Utilization/Response
Aperiodic task $\varepsilon_1$	7	3		17-7=10
Aperiodic task $\varepsilon_2$	11	4		35-11=24
Periodic task $\tau_1$	10	4	3	0,400
Periodic task $\tau_2$	20	6	2	0,300
Background Server	*	*	1	*



# Scheduling aperiodic tasks

## Background server

- 
- Advantages
    - Simplicity of implementation
  - Disadvantages
    - Difficult to predict response time of aperiodic tasks
    - ... although aperiodic tasks can be critical
    - Bad response time under heavy workload



# Scheduling aperiodic tasks

## Polling server

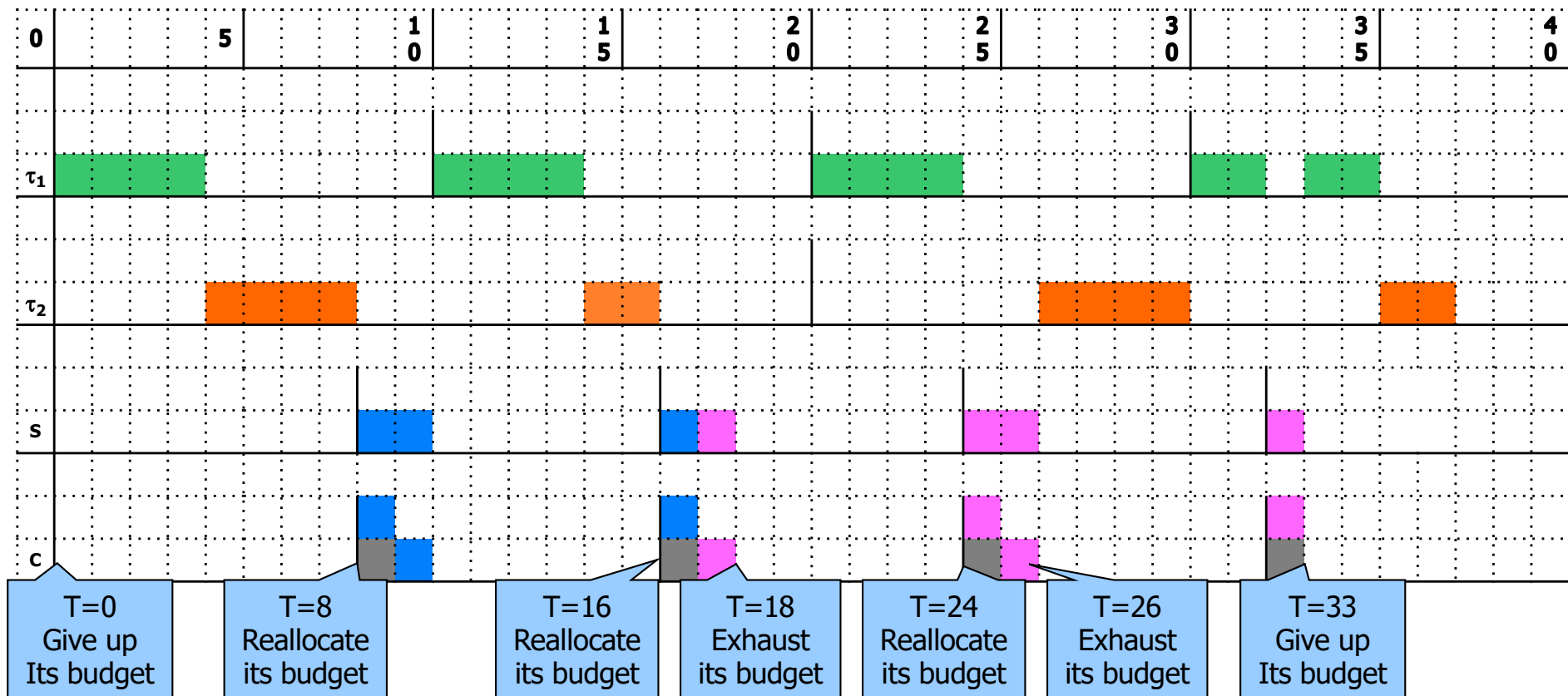
---

- Aperiodic tasks are processed sequentially by a high priority server
- The server has a budget and a period
- The budget is reallocated every period
- The time consumed to process an aperiodic task is debited on its budget
- The server executes aperiodic tasks within its budget
- The server becomes inactive when there is no task to execute and gives up its budget until next period

# Scheduling aperiodic tasks

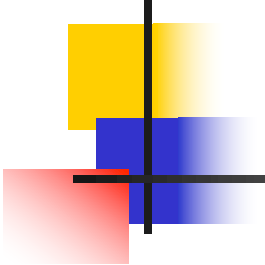
## Polling server

	Period/Release	Budget	Priority	Utilisation/Response
Aperiodic task $\varepsilon_1$	7	3		10
Aperiodic task $\varepsilon_2$	11	4		22
Periodic task $\tau_1$	10	4	2	0,400 6
Periodic task $\tau_2$	20	6	1	0,300 20
Polling Server	8	2	3	0,250 2



# Scheduling aperiodic tasks

## Polling server

- 
- Advantages
    - Simplicity of implementation
  - Disadvantages
    - By giving up its budget, the server exhausts the processing time allocated to future tasks
    - Bad response time even when aperiodic tasks are released just after server activations



# Scheduling aperiodic tasks

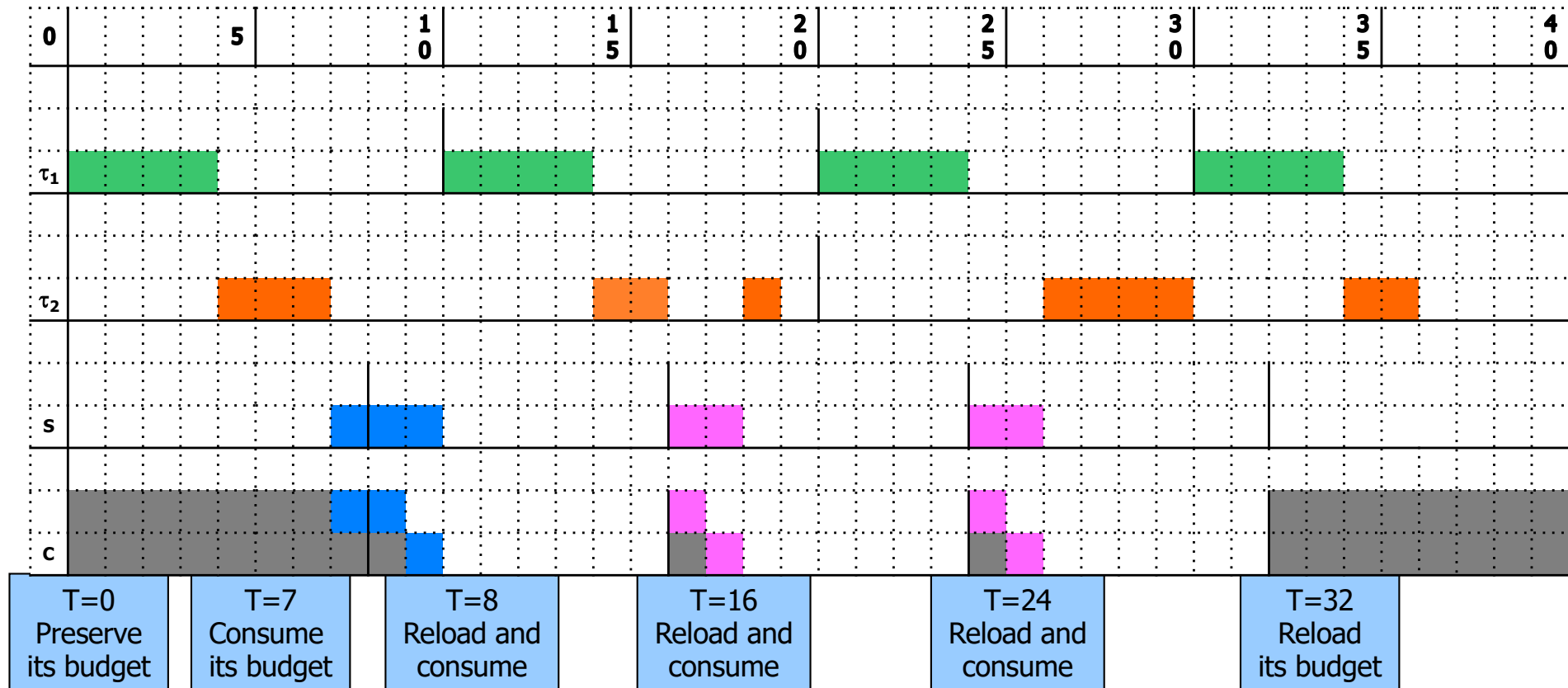
## Deferred server

- The aperiodic tasks are processed sequentially by a high priority server
- The server has a budget and a period
- The budget is reallocated every period
- The time consumed to process an aperiodic task is debited on its budget
- The server becomes active only when an aperiodic task is to be processed and its budget is not yet exhausted

# Scheduling aperiodic tasks

## Deferred server

	Period / Release	Budget	Priority	Usage / Response
Aperiodic task $\varepsilon_1$	7	3		3
Aperiodic task $\varepsilon_2$	11	4		15
Periodic task $\tau_1$	10	4	2	0,400
Periodic task $\tau_2$	20	6	1	0,300
Deferred server S	8	2	3	0,250



# Scheduling aperiodic tasks

## Deferred server

- Advantages

- It preserves its budget for future tasks

- Disadvantages

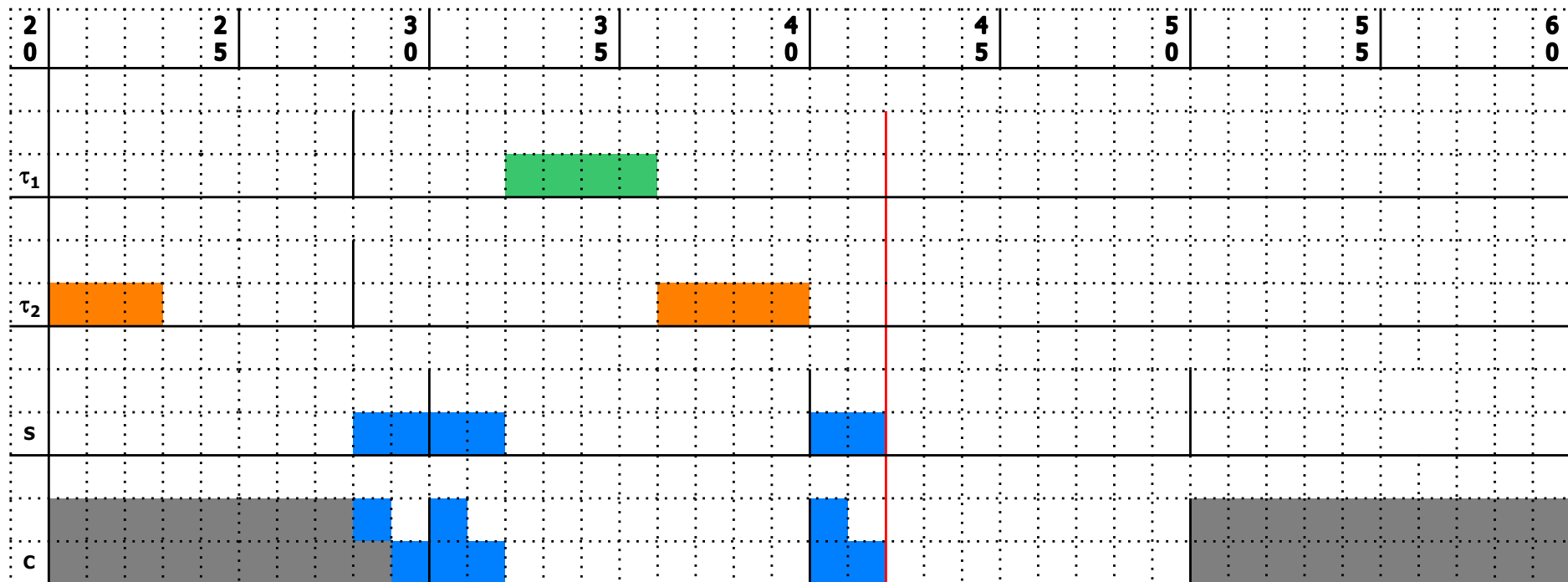
- By not immediately consuming its budget, a deferred server violates the scheduling hypotheses of a periodic task because it does not execute while it can.
- A scheduling analysis can claim that the scheduling is correct while the server causes a deadline miss of a low priority task by delaying its execution

# Scheduling aperiodic tasks

## Issue with deferred server

	Period / Release	Budget	Priority	Utilisation/Response
Aperiodic task $\varepsilon_1$	28	6		12
Periodic task $\tau_1$	14	4	1	0,285 13
Periodic task $\tau_2$	14	5	2	0,357 9
Deferred Server S	10	2	3	0,200 2

Tasks S,  $\tau_1$  and  $\tau_2$  are schedulable with RMS:  $R_S=2<8$ ,  $R_{\tau_1}=8<14$ ,  $R_{\tau_2}=13<14$



# Scheduling aperiodic tasks

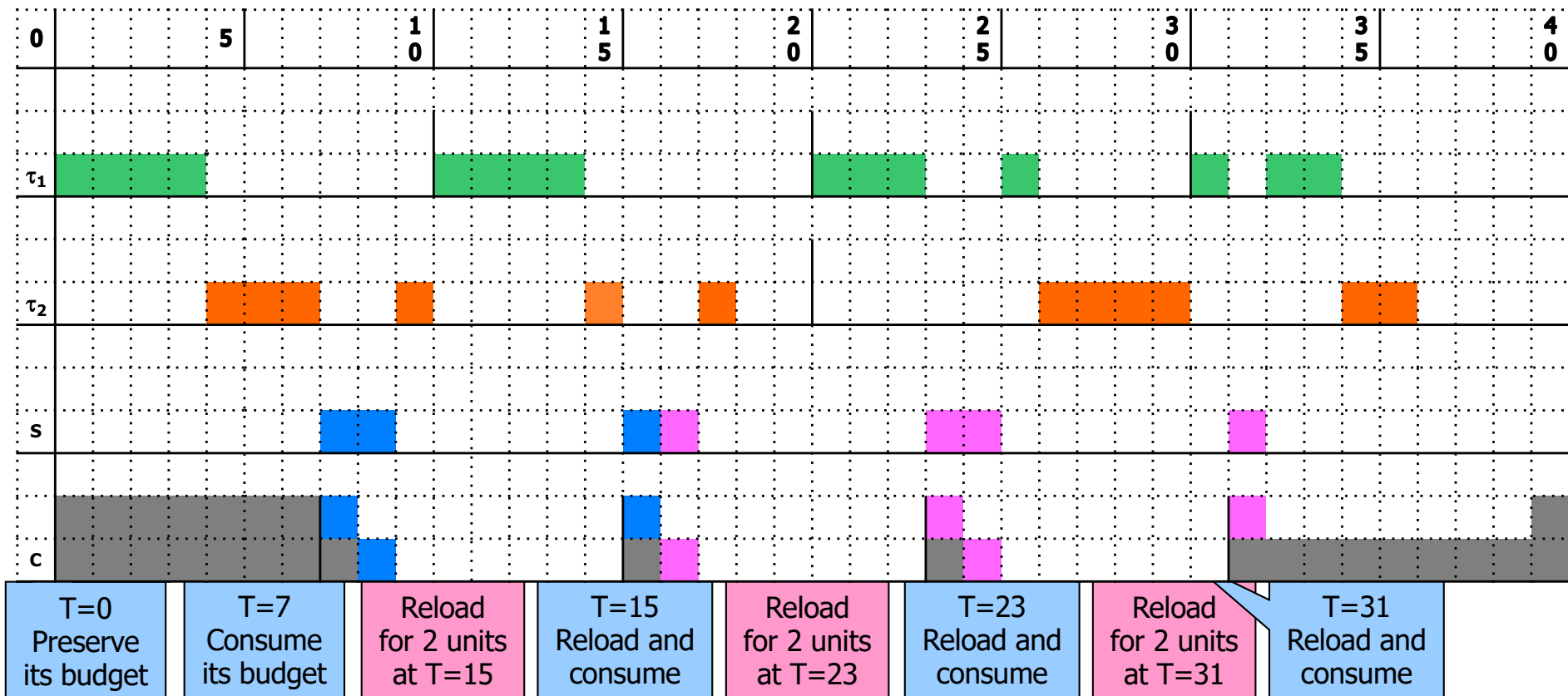
## Sporadic server

- The aperiodic tasks are processed sequentially by a high priority server
- The server has a budget and a period
- The time consumed to process an aperiodic task is debited on its budget
- The time consumed is credited back after a task period from the time it starts consuming
- The server becomes active when an aperiodic task is to be processed and its budget is not exhausted

# Scheduling aperiodic tasks

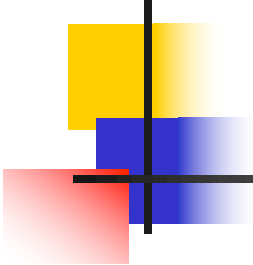
## Sporadic server

	Period/Release	Budget	Priority	Utilisation/Response
Aperiodic Task $\varepsilon_1$	7	3		9
Aperiodic Task $\varepsilon_2$	11	4		21
Aperiodic Task $\tau_1$	10	4	2	0,400
Aperiodic Task $\tau_2$	20	6	1	0,300
Sporadic Server S	8	2	3	0,250



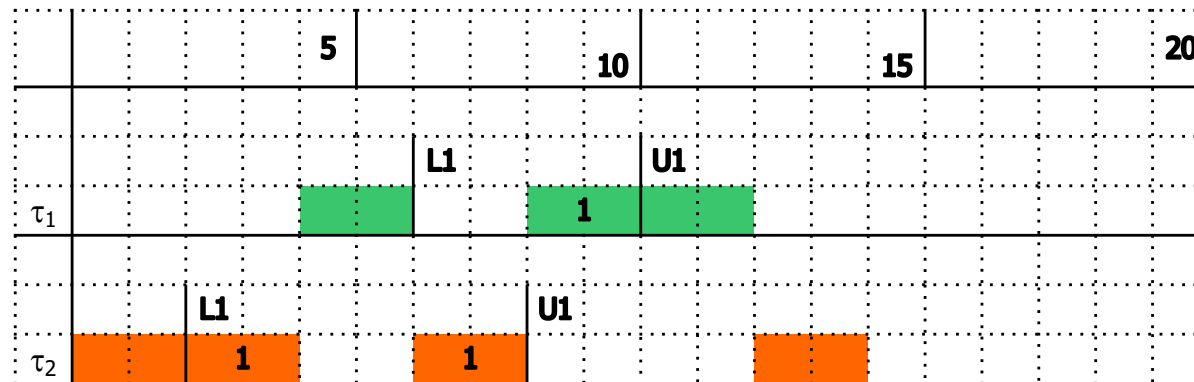
# Scheduling aperiodic tasks

## Sporadic server

- 
- Advantages
    - Better properties than previous servers
  - Disadvantages
    - High complexity compared to the deferred server
  - Variant
    - An alternative is to transform the sporadic server into a background server (low priority) when its budget is exhausted in order to take advantage the unused processing time

# Sharing resources

## Blocking Time (BT) and Scheduling



BT = 2u  
Max BT = 4u

- Analogy with the previous model (independent tasks)
  - Let  $B_i$  be the longest duration of potential blocking of task  $t_i$  by a task of lower priority
  - Analogy with a scenario in which task  $t_i$  would have a WCET of  $C_i + B_i$  instead of  $C_i$
- The goal is to reduce  $B_i$  by introducing adequate resource sharing policies



# Sharing resources

## Including blocking time in scheduling test

- A high priority task can be blocked **directly** by a low priority task because they share a common resource
- A middle priority task can be blocked **indirectly** when a high priority task is blocked by a low priority task, those tasks having no resource shared with the middle priority task

- Sufficient scheduling condition with RMS

$$\forall i, 1 \leq i \leq n, \sum_{j \leq i} C_j / T_j + B_i / T_i \leq n (2^{1/n} - 1)$$

- Response time with static priorities and blocking times

$$\forall i, 1 \leq i \leq n, \exists t \leq D_i W_i(t) = \sum_{j \leq i} C_j * \lceil t / T_j \rceil + B_i \leq t$$

- Sufficient scheduling condition with EDF

$$\forall i, 1 \leq i \leq n, \sum_{j \leq i} C_j / T_j + B_i / T_i \leq 1$$



# Sharing resources

## Priority Inheritance Protocol

---

### ■ Problems

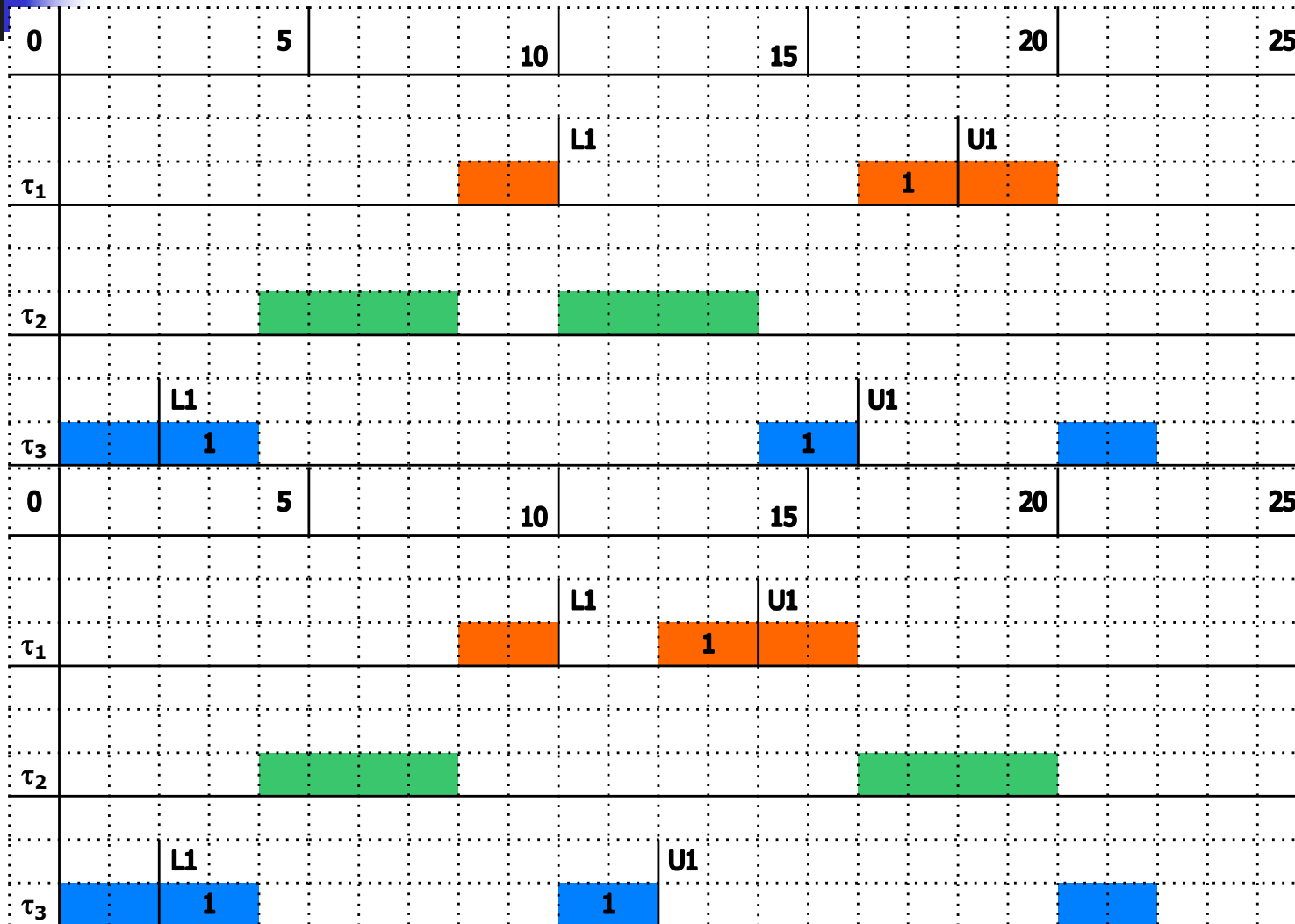
- Preemptive scheduling with fixed priority can lead to a situation of priority inversion
- A low priority task blocks a high priority task for a time longer than that of its mutual exclusion
- Difficult to estimate upper bound of blocking time

### ■ Solution

- Priority inheritance raises the priority of the blocking task to the blocked one
- Once the semaphore is released, the blocking task returns to its initial priority

# Priority Inheritance Protocol

Blocking time longer than the expected one



Without PIP  
 BT = 6u  
 Max BT = 12

With PIP  
 BT = 2u  
 Max BT = 4u

Compute the maximum blocking time (BT) on this example



# Priority Inheritance Protocol

## Benefits and Drawbacks

---

- Benefits
  - Reduce blocking time
- Remaining drawbacks
  - N resources : N blocking times & priority elevations
  - Deadlocks are possible
- Properties
  - A low priority task blocks a priority task **only once**
  - A high priority task blocks on a resource **only once**
  - A low task can **indirectly** block a task **without sharing** a resource because of priority inheritance

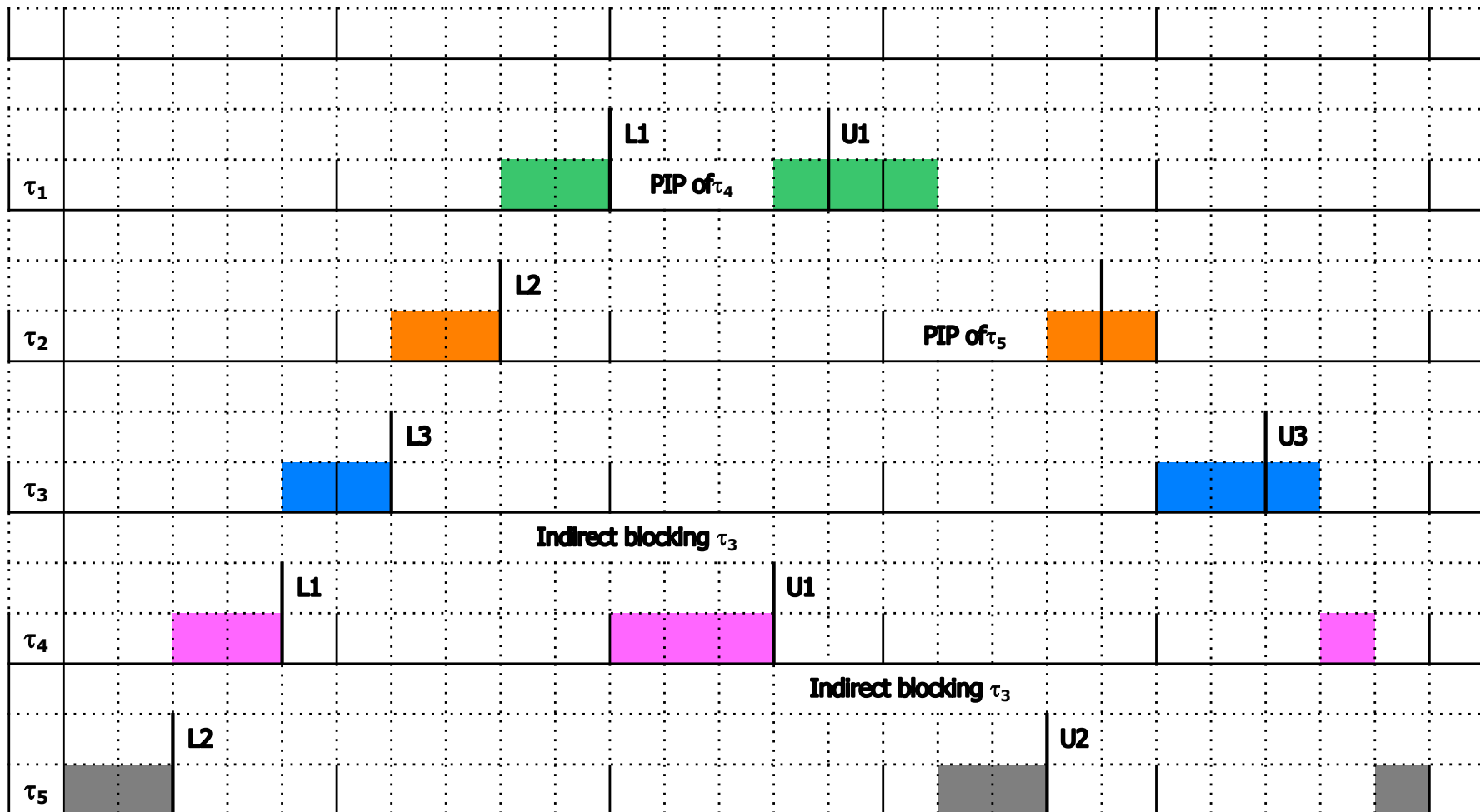
# Priority Inheritance Protocol

## Blocking Time Analysis

- $\tau_3$  can be actually blocked by the 3 resources
- Although  $\tau_3$  uses only R3, it can be indirectly blocked by  $\tau_4$  or  $\tau_5$  when they block  $\tau_1$  or  $\tau_2$  using R1 et R2 because of priority inheritance ( $\tau_4$  inherits  $\tau_1$  priority)
- A low priority task blocks a priority task **only once**
- A high priority task blocks on a resource **only once**
- The worst case for  $\tau_3$  occurs when
  - $\tau_4$  locks R1, inherits from  $\tau_1$
  - $\tau_5$  locks R2, inherits from  $\tau_2$
- $B_3 = \max(\mathbf{3+2}, 3+1, 1+2, 1+1) = 5$

	R1	R2	R3	B
$\tau_1$	2	.	.	3
$\tau_2$	.	1	.	5
$\tau_3$	.	.	2	5
$\tau_4$	3	3	1	2
$\tau_5$	1	2	1	0

# Examples of indirect blocking





# Sharing resources

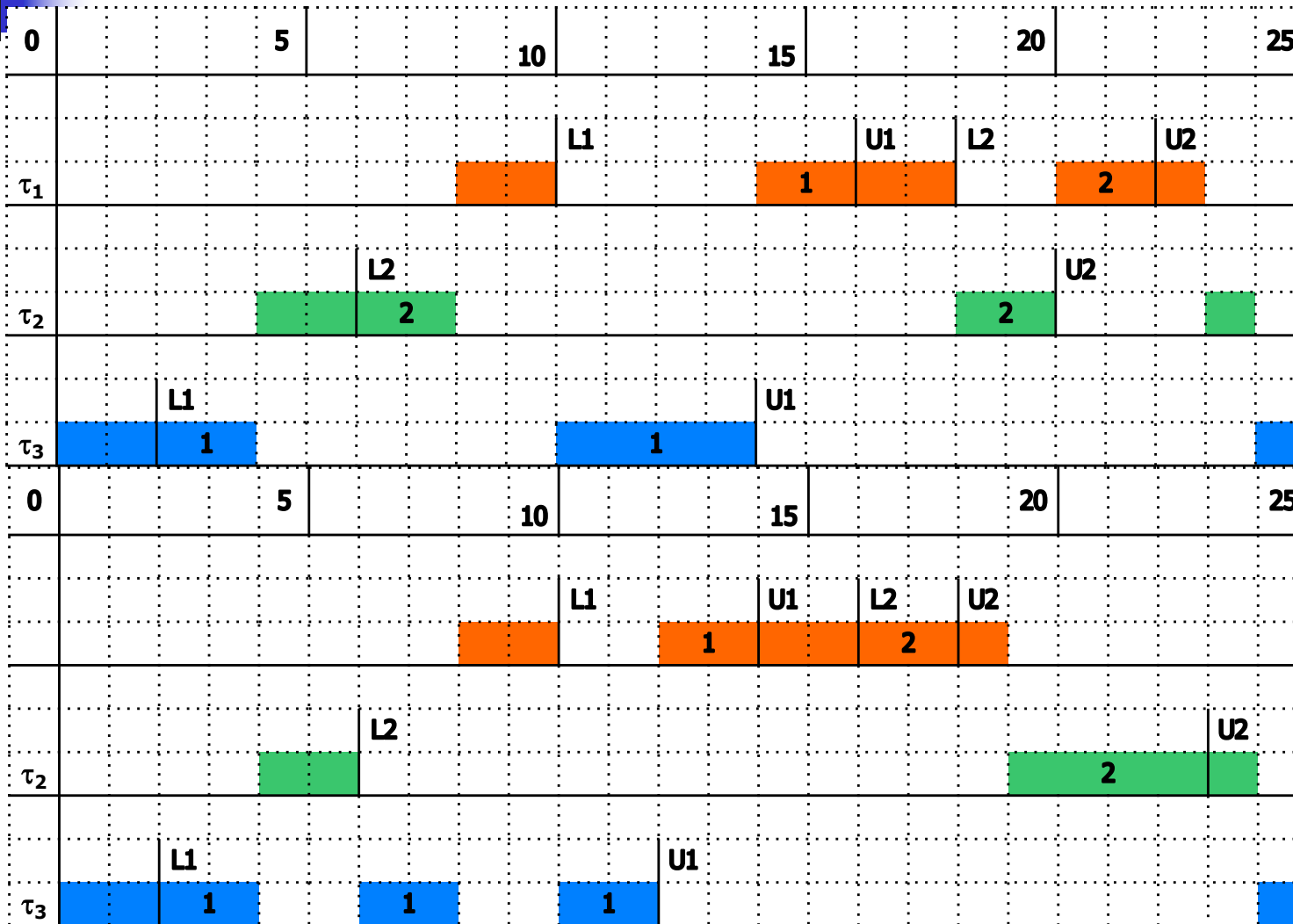
## Priority Ceiling Protocol

---

- Problems
  - N resources : N blocking times & priority elevations
  - Deadlocks are possible
- Solution (fixed priorities)
  - The (static) priority ceiling represents the maximum priority of tasks using the resource
  - A task gets access to a resource when its priority is strictly greater than all the priority ceiling of the resources in use
  - The blocking task inherits the priority of the highest priority blocked task

# Priority Ceiling Protocol

## Chained blocking



Without PCP  
 BT = 6u  
 Max BT = 10u

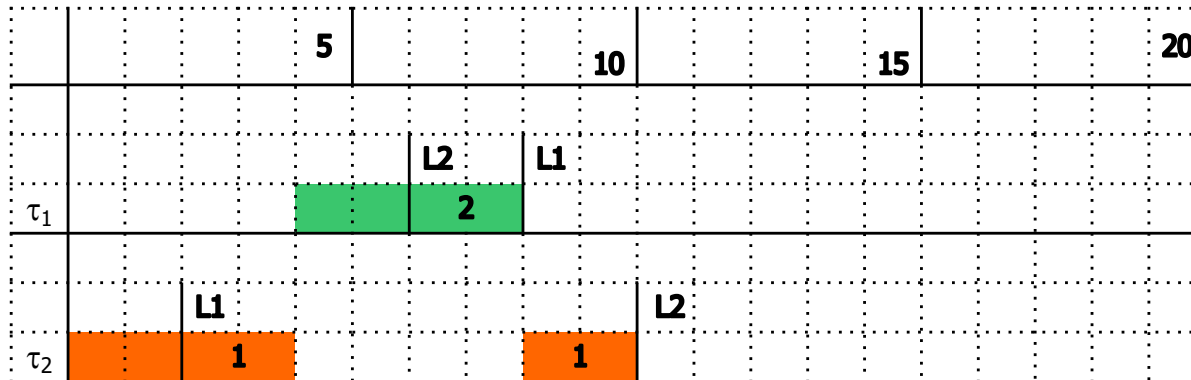
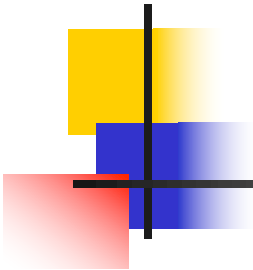
With PCP  
 BT = 2u  
 Max BT = 6u

Compute the maximum blocking time (BT) on this example

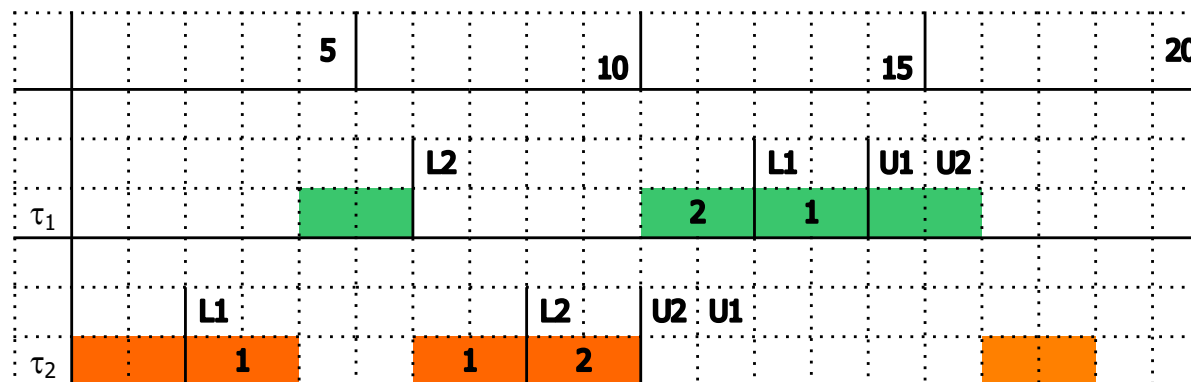


# Priority Ceiling Protocol

## Deadlock



Without PCP



With PCP



# Priority Ceiling Protocol

## Benefits and drawbacks

---

- Benefits
  - No chained blocking time
  - The task is blocked at most once whatever the number of shared resources
  - No deadlock
- Remaining drawbacks
  - Implementation complexity
  - Multiple priority elevations

# Priority Ceiling Protocol

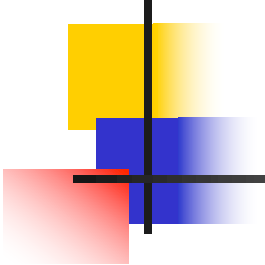
## Analysis of blocking time

- A low priority task blocks a priority task **only once**
- A high priority task blocks on a resource **only once**
- A low task can **indirectly** block a task **without sharing** a resource because of priority inheritance
- With PCP, a task can be blocked by a lower priority task only once and on a **single** resource

	R1	R2	R3	B
$\tau_1$	2	.	.	3
$\tau_2$	.	1	.	3
$\tau_3$	.	.	2	3
$\tau_4$	3	3	1	2
$\tau_5$	1	2	1	0

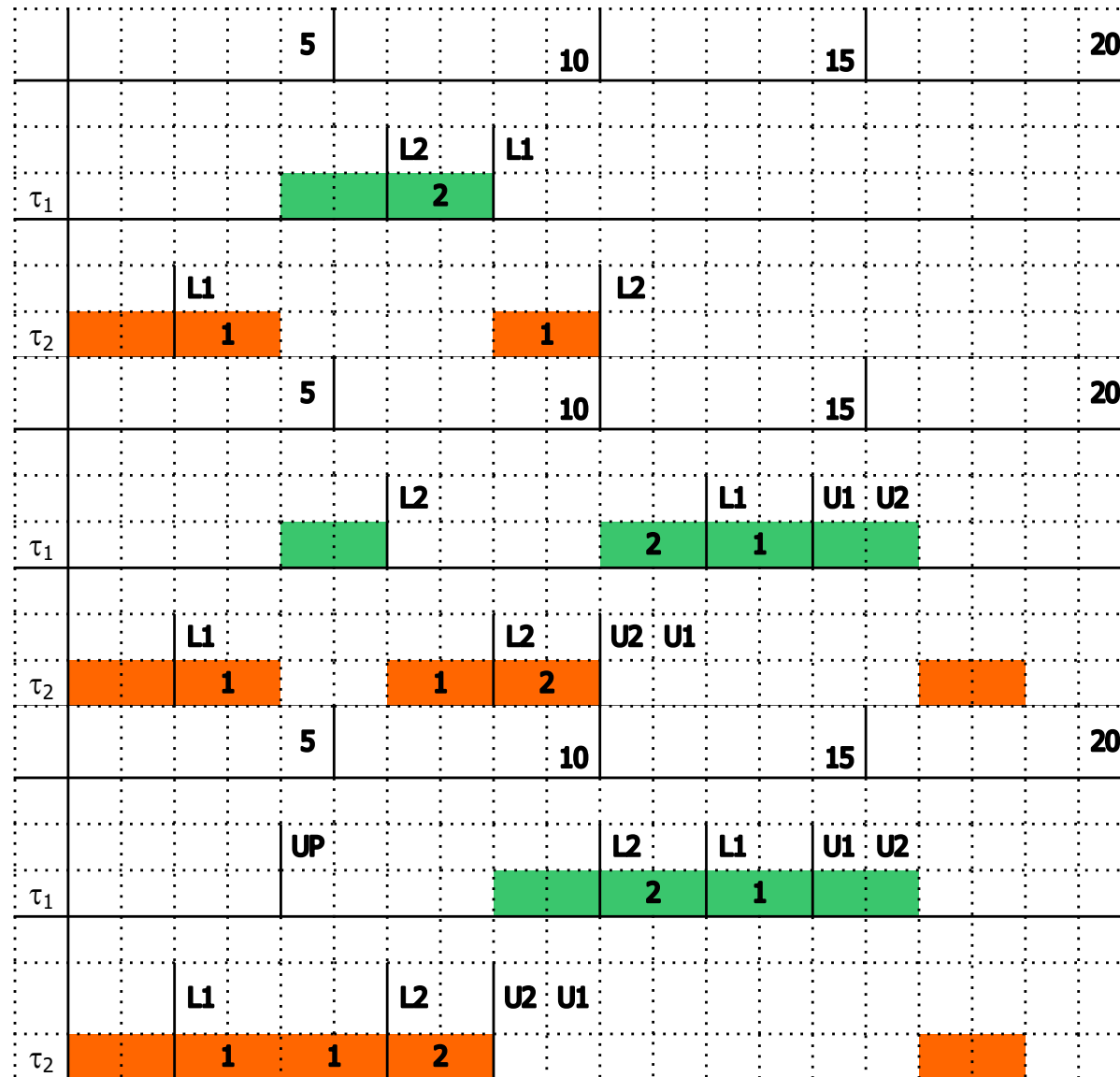
# Sharing resources

## Immediate Priority Ceiling Protocol

- 
- Problems
    - PCP implementation complexity
    - PCP multiple priority changes
  - Solution
    - The (static) priority ceiling represents the maximum priority of the tasks that use it
    - When a task gets access to a resource, it inherits (immediately) a priority strictly greater than the priority ceiling

# Immediate Priority Ceiling Protocol

## Deadlock



Laurent Pautet



# Immediate Priority Ceiling Protocol

## Benefits and drawbacks

---

- Benefits
  - Less complex than PCP
- Drawbacks
  - IPCP (and PCP) relies on fixed priority scheduling



# Conclusions

---

- To satisfy the time constraints in hard real time systems, the first concern must be the predetermination of the system behavior.
- Offline static scheduling is most often the only practical way to achieve predictable behavior in a complex system



# Lectures

---

- G. Buttazzo. Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications Kluwer academic Publishers, Boston, 1997.
- C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment. Journal of the ACM, 20(1):46--61, Jan. 1973.
- L. Sha, R. Rajkumar and J. Lehoczky, "*Priority Inheritance Protocol*": An Approach to real-time synchronisation," IEEE Transaction on Computers 39(9), pp.1175-1185, 1993.
- Min-Ih Chen and Kwei-Jay Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. Journal of Real-Time Systems, 2:325--346, 1990.
- T. Baker. Stack-based scheduling of real-time processes. Real-Time Systems, 3(1):67--99, March 1991.
- B. Sprunt, L. Sha, and J. P. Lehoczky, "Aperiodic scheduling for hard real-time system". The Journal of Real-Time Systems, 1, pp. 27-60, 1989.