Mälardalen University

Real-time Scheduling

Saad Mubeen

MÄLARDALEN UNIVERSITY

Real-time Systems

*"A real-time system is a system that reacts upon outside events and performs a function based on these and gives a response within a certain **time**. Correctness of the function does not only depend on correctness of the result, but also the timeliness of it".*

Non-real-time systems

- Correct function if produced result is correct

System does the right thing

Real-time systems

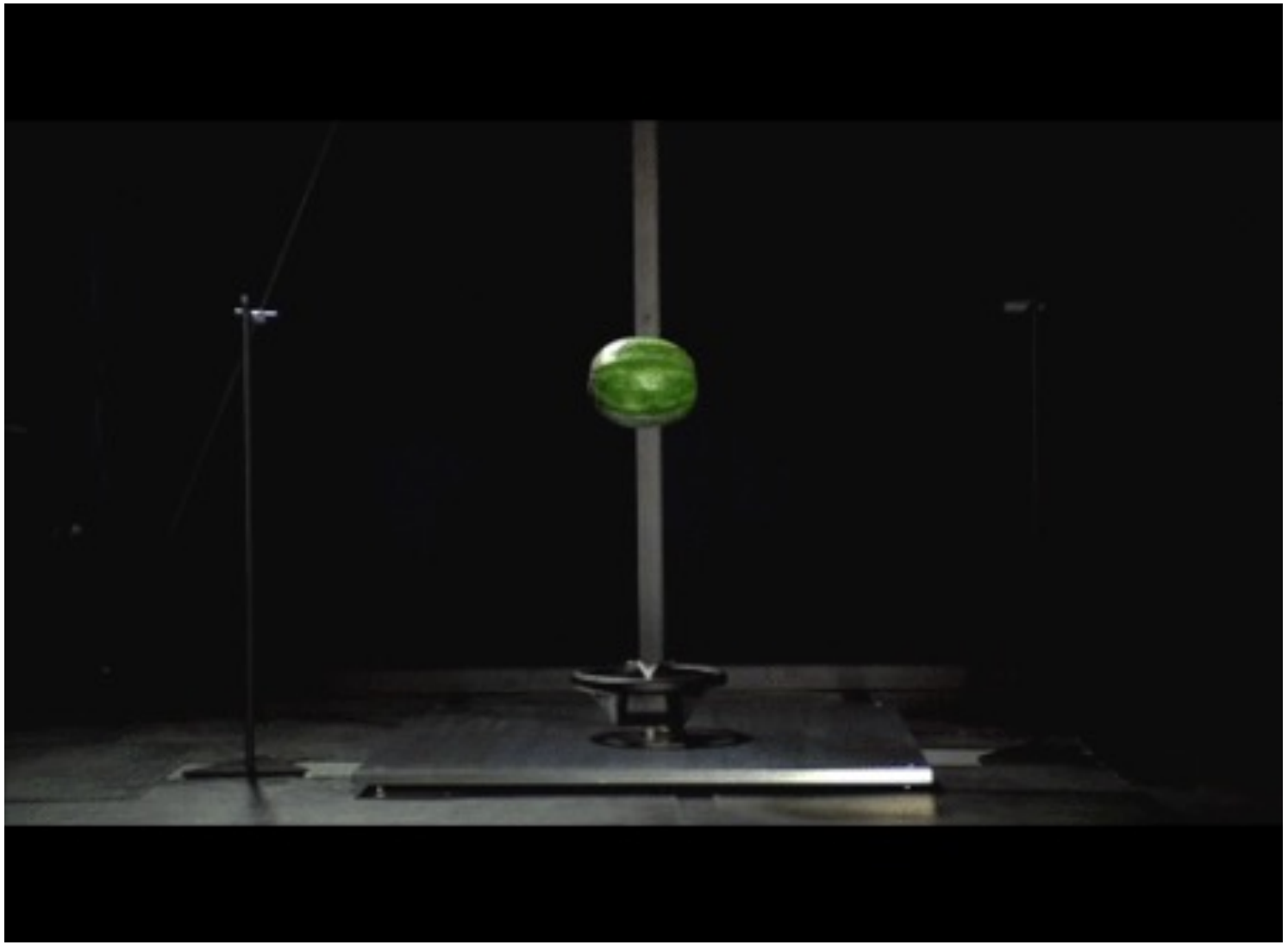
- Correct function if produced result is correct and **delivered on time**

System does the right thing...

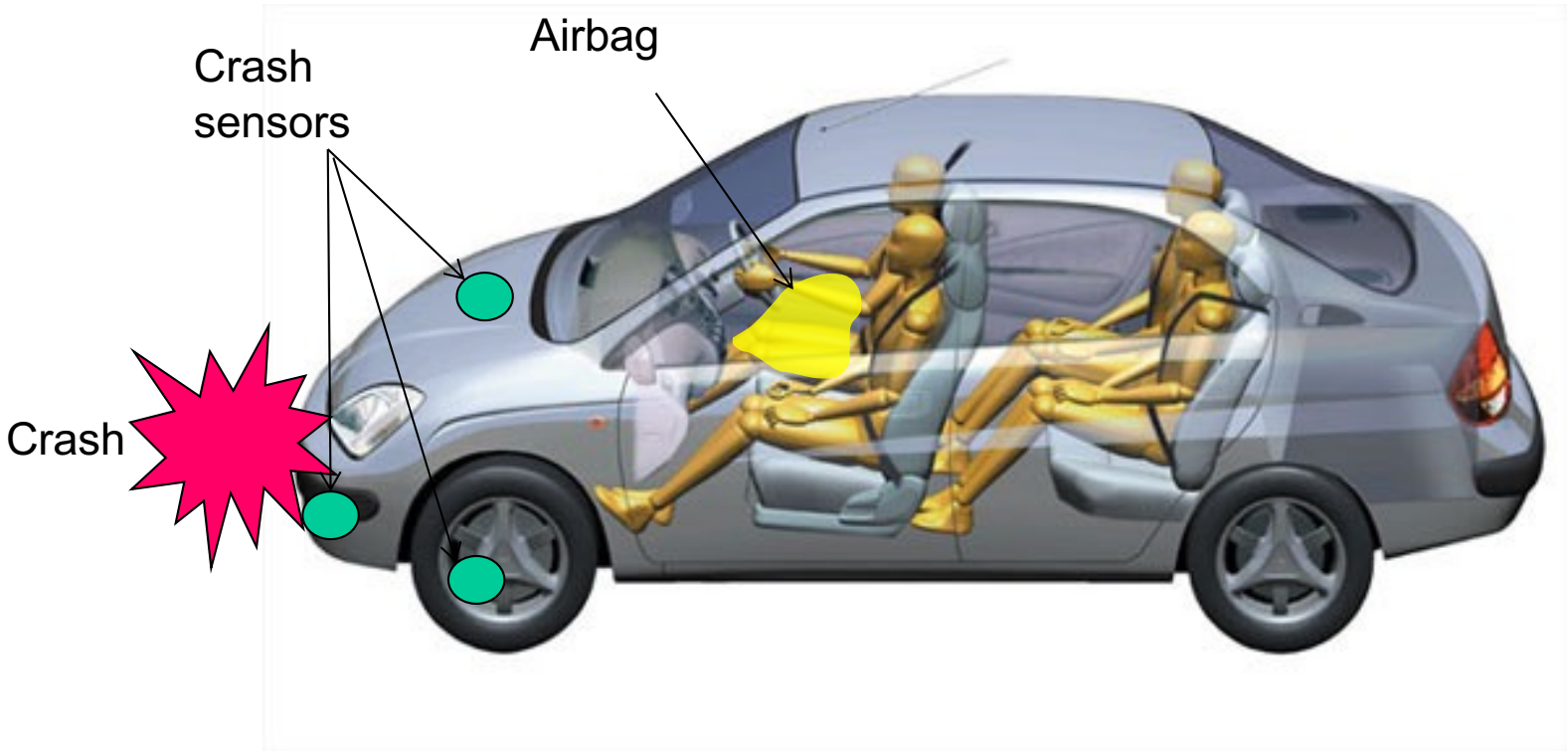
+

...and it does it on time

Real-time Systems

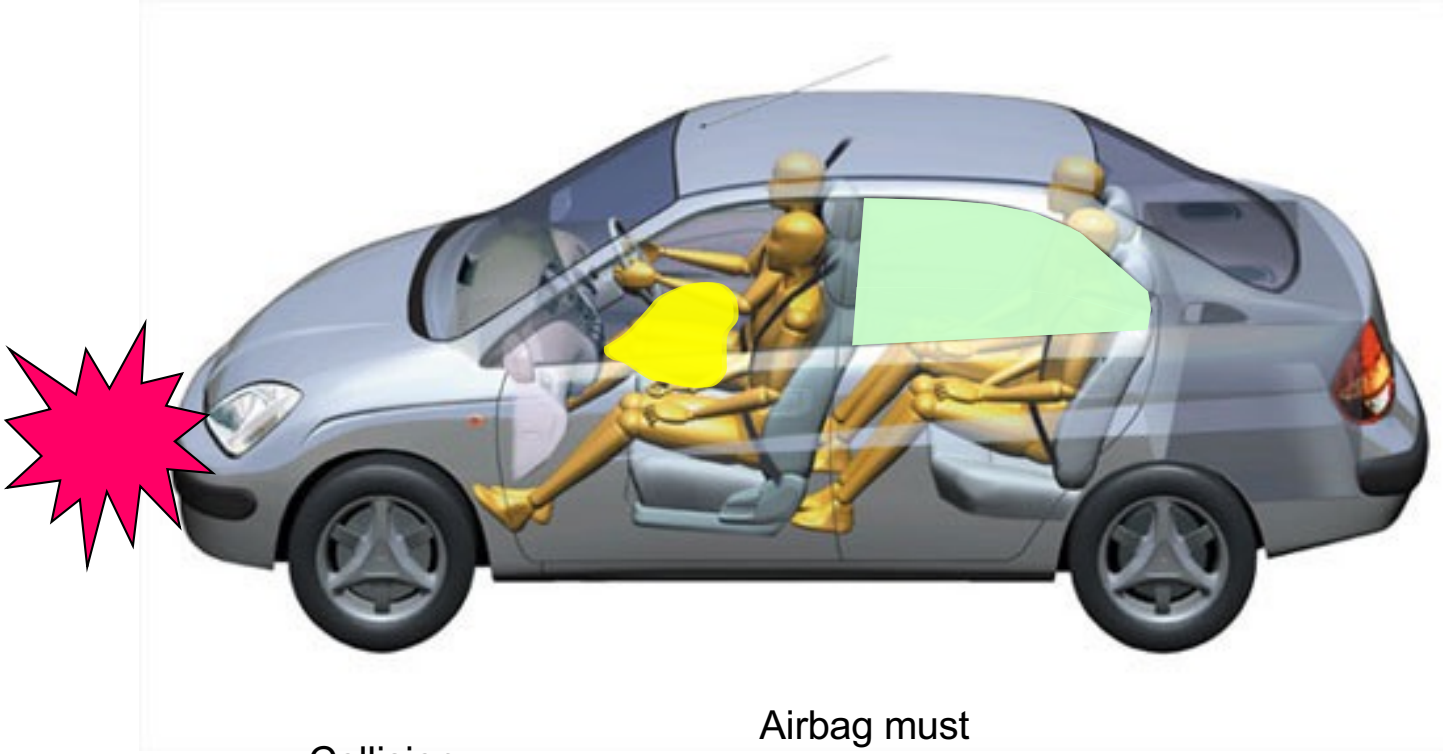


Real-time Systems



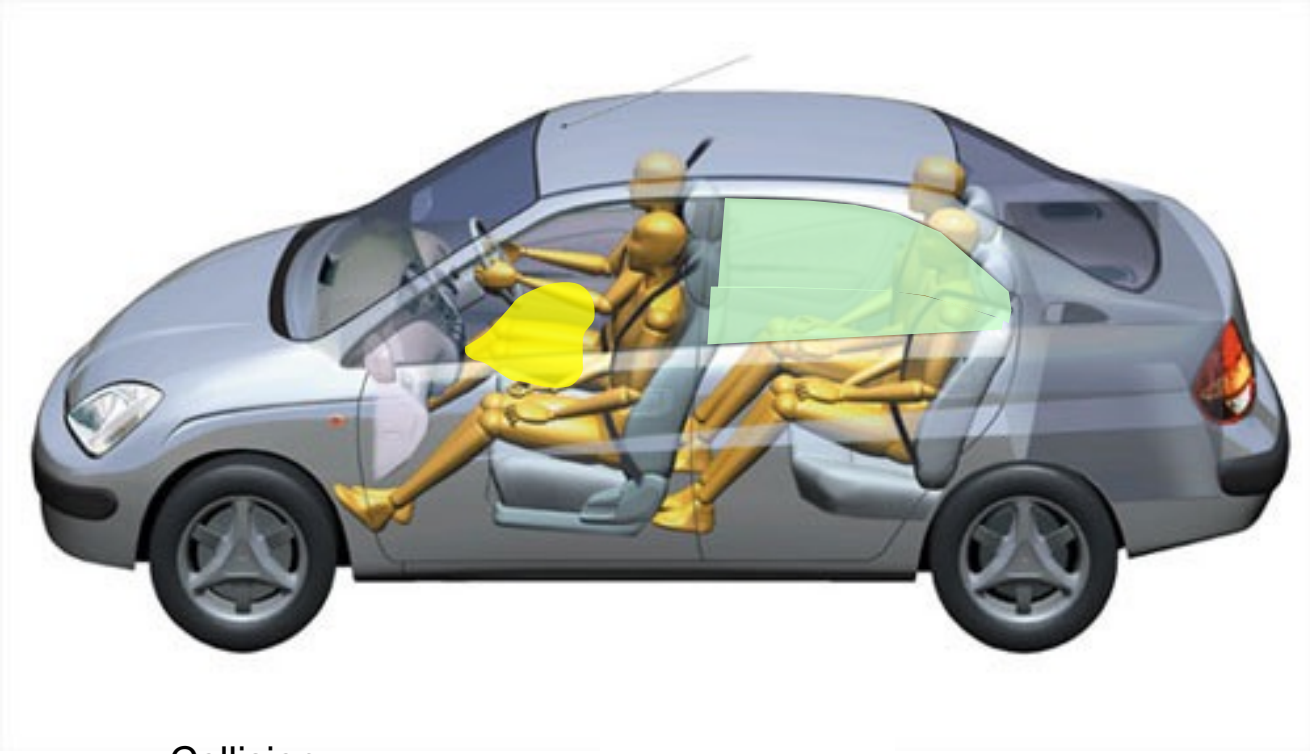
The decision to deploy an airbag is made within **15 to 30 milliseconds** after the onset of the crash, and the airbags are inflated within app. **50-80 milliseconds**.

Real-time Systems



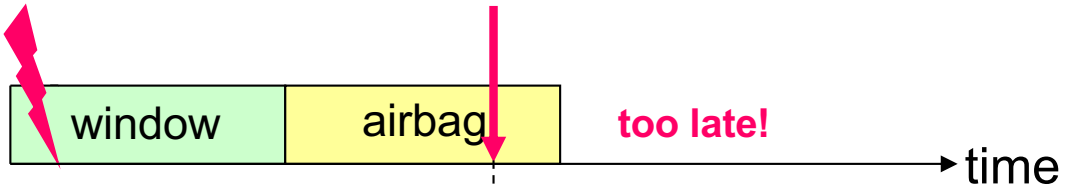
Non-real-time system:

Real-time Systems

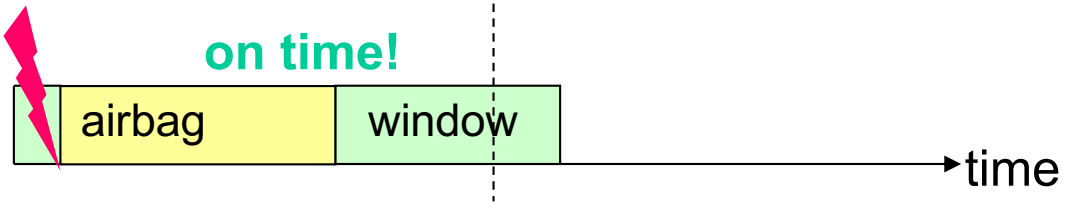


Collision
occurs here

Non-real-time system:

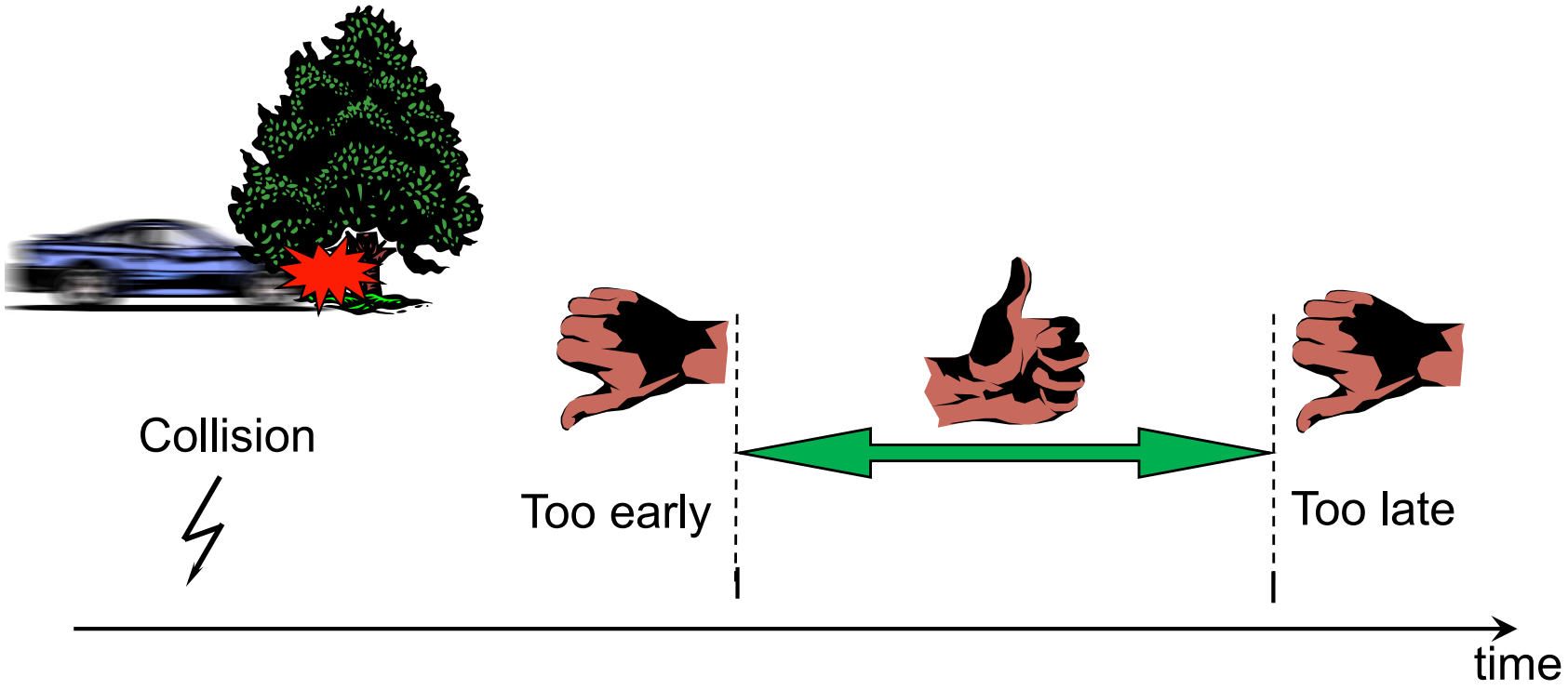


Real-time system:



Example: Airbag

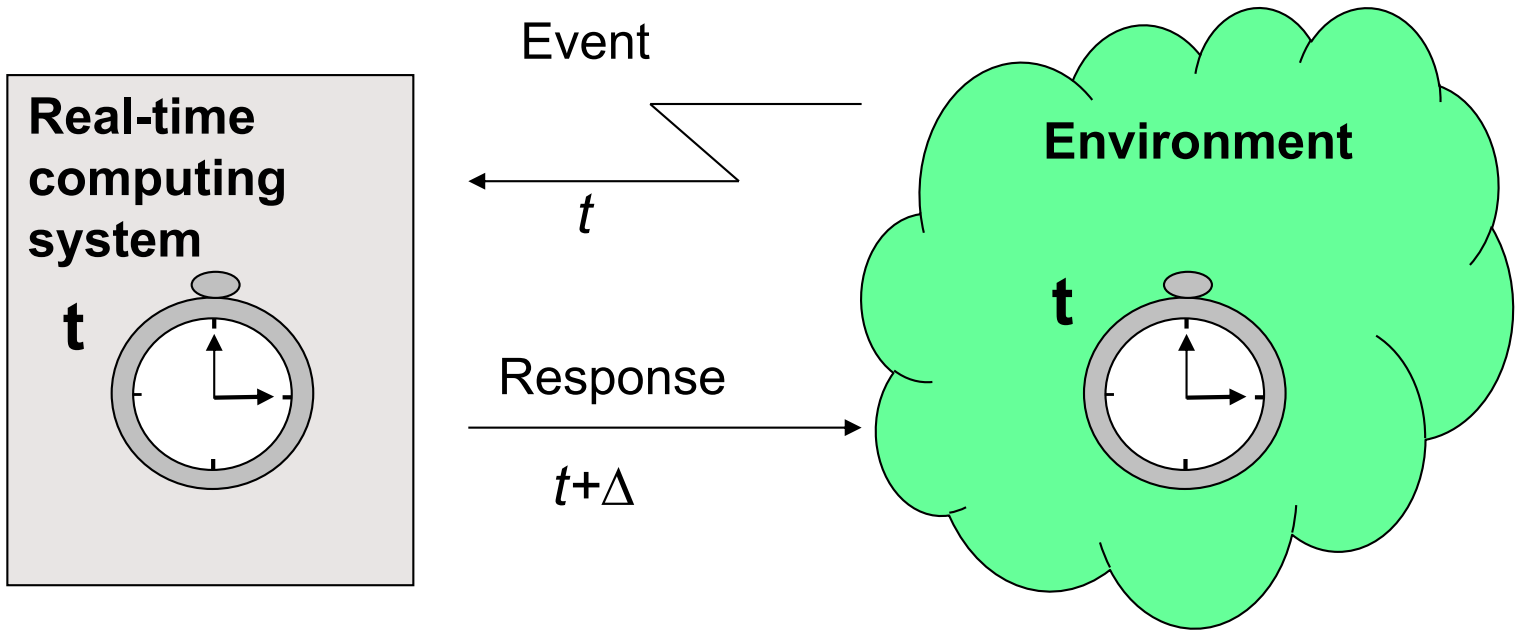
The airbag must not be inflated too late, nor too early!



Real-time ≠ fast !
Real-time = predictable!

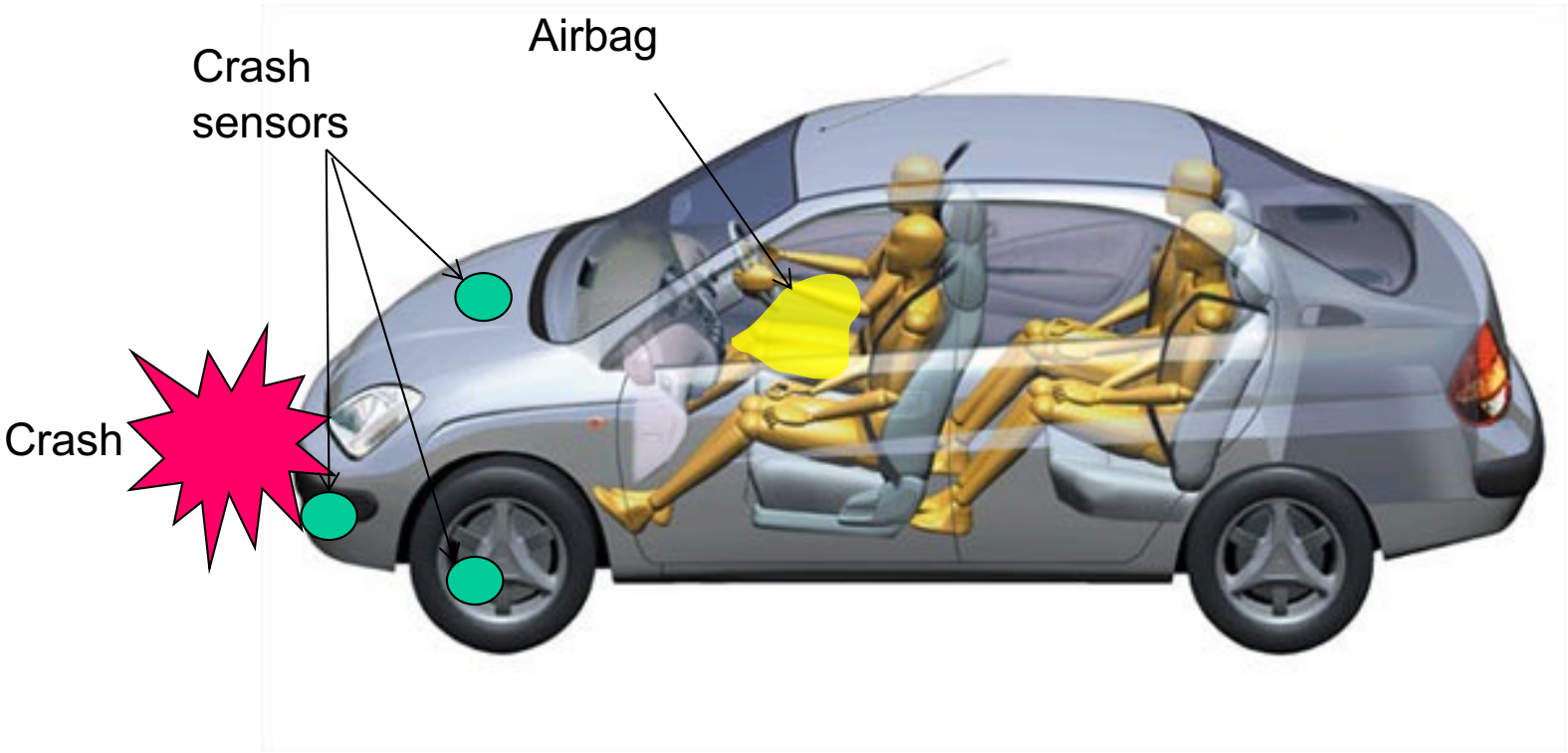
Real-Time Systems

"A real-time system is a system that **reacts upon outside events** and performs a function based on these and **gives a response within a certain time**. Correctness of the function does not only depend on correctness of the result, but also the **timeliness** of it".



Realtime means that the system must be synchronized with the environment. The **controlled process dictates the time scale** (some processes have demand on response at **second**-level, others at **milli-** or even **microsecond** level).

Example: Airbag

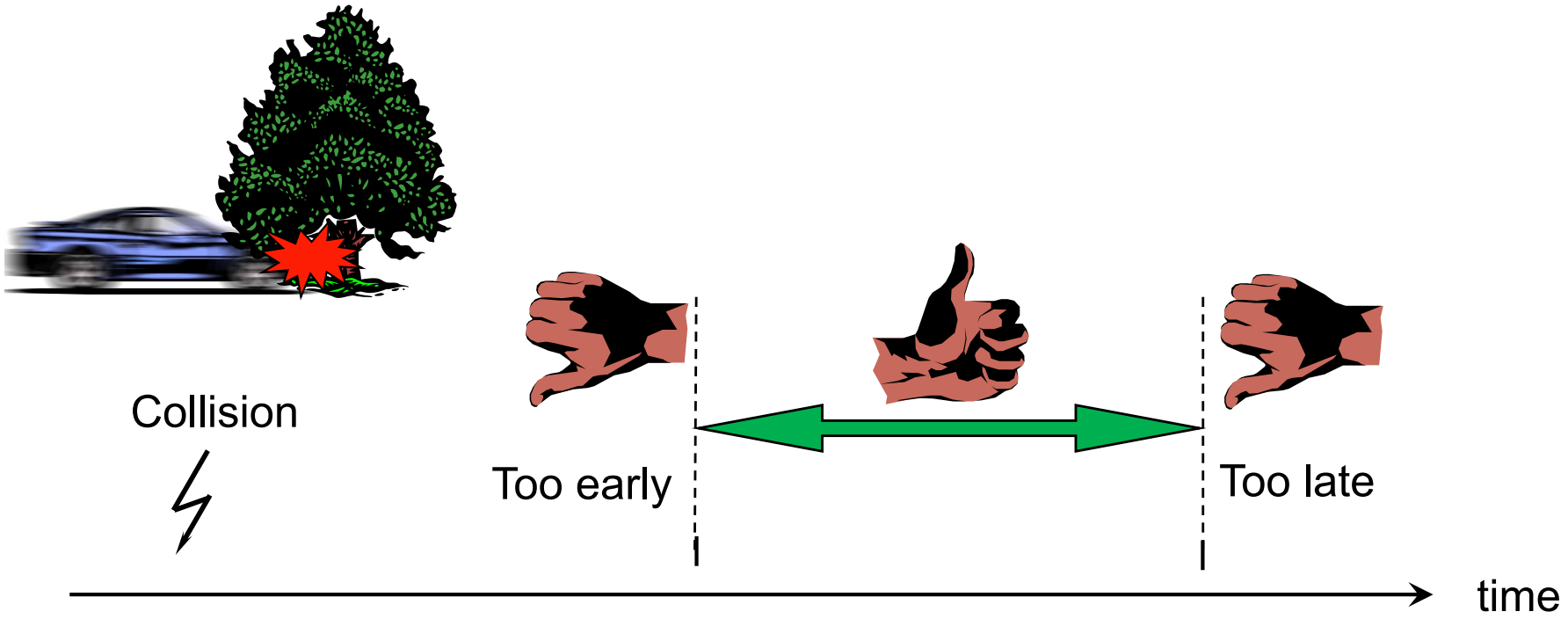


REAL-TIME:

The decision to deploy an airbag is made within 15 to 30 milliseconds after the onset of the crash, and the airbags are inflated within approximately 50-80 milliseconds.

Example: Airbag

The airbag must not be inflated too late, nor too early!



Real-time ≠ fast !
Real-Time = predictable!

Hard and soft real-time systems

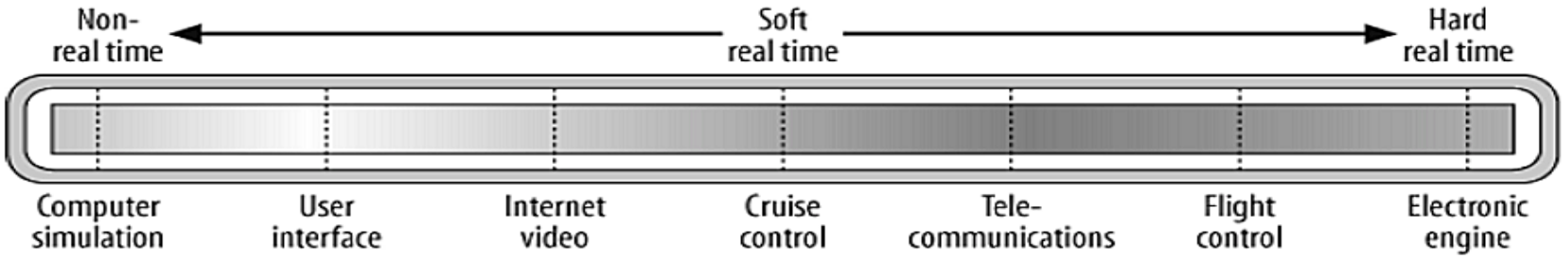
A real-time timing constraint is called *hard*, if not meeting that constraint could result in a catastrophe. All other constraints are called *soft*.

Hard real-time

- Hard deadlines – catastrophe if deadline is missed
- Worst-case system behavior considered
- E.g. automotive, airplanes, industrial control, …

Soft real-time

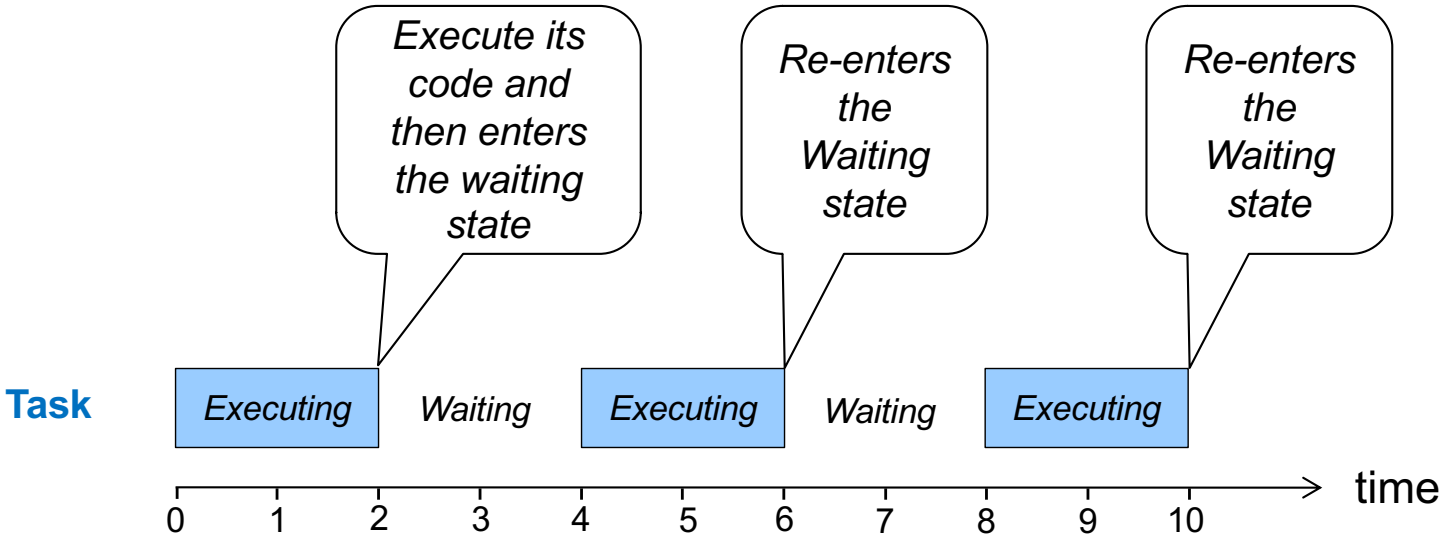
- Fuzzier deadlines – can miss some deadlines
- Average-case considered
- E.g., consumer electronics




Real-time
Tasks

Real-time tasks

Periodic tasks

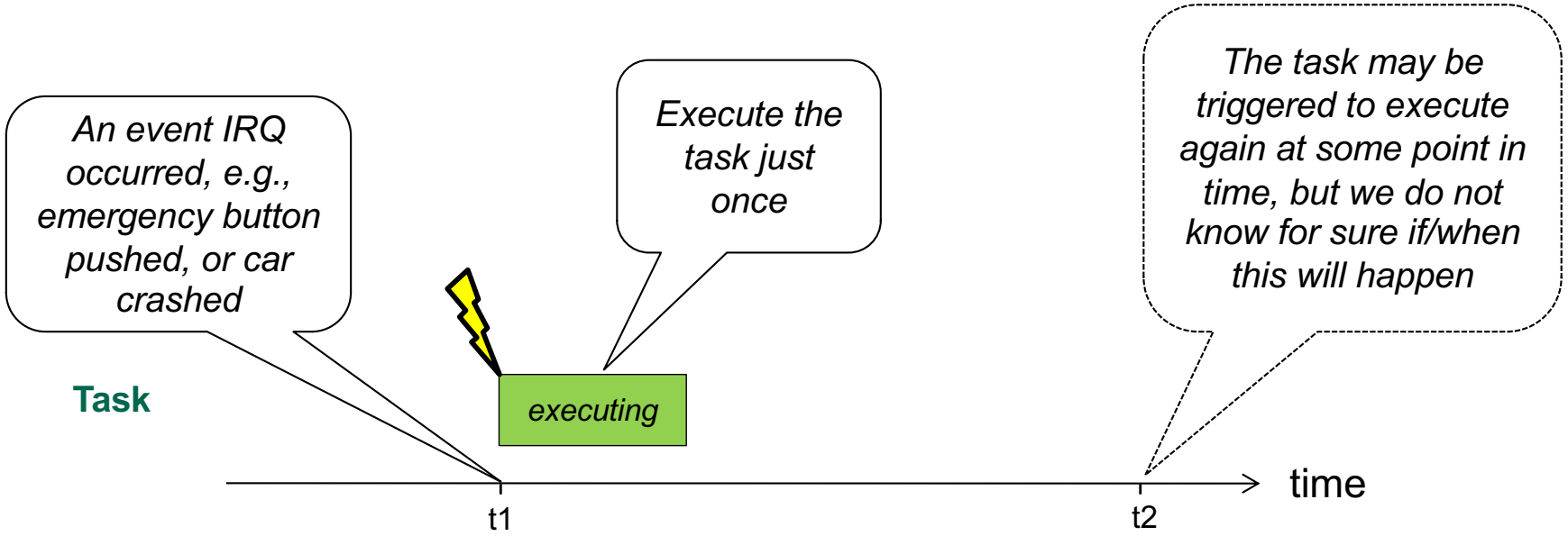


Examples

- sensory data acquisition (e.g., audio and video sampling)
- control loops (e.g., robot arm control, ABS control)
- system monitoring (e.g., temperature and pressure monitoring)

Real-time tasks

Aperiodic tasks

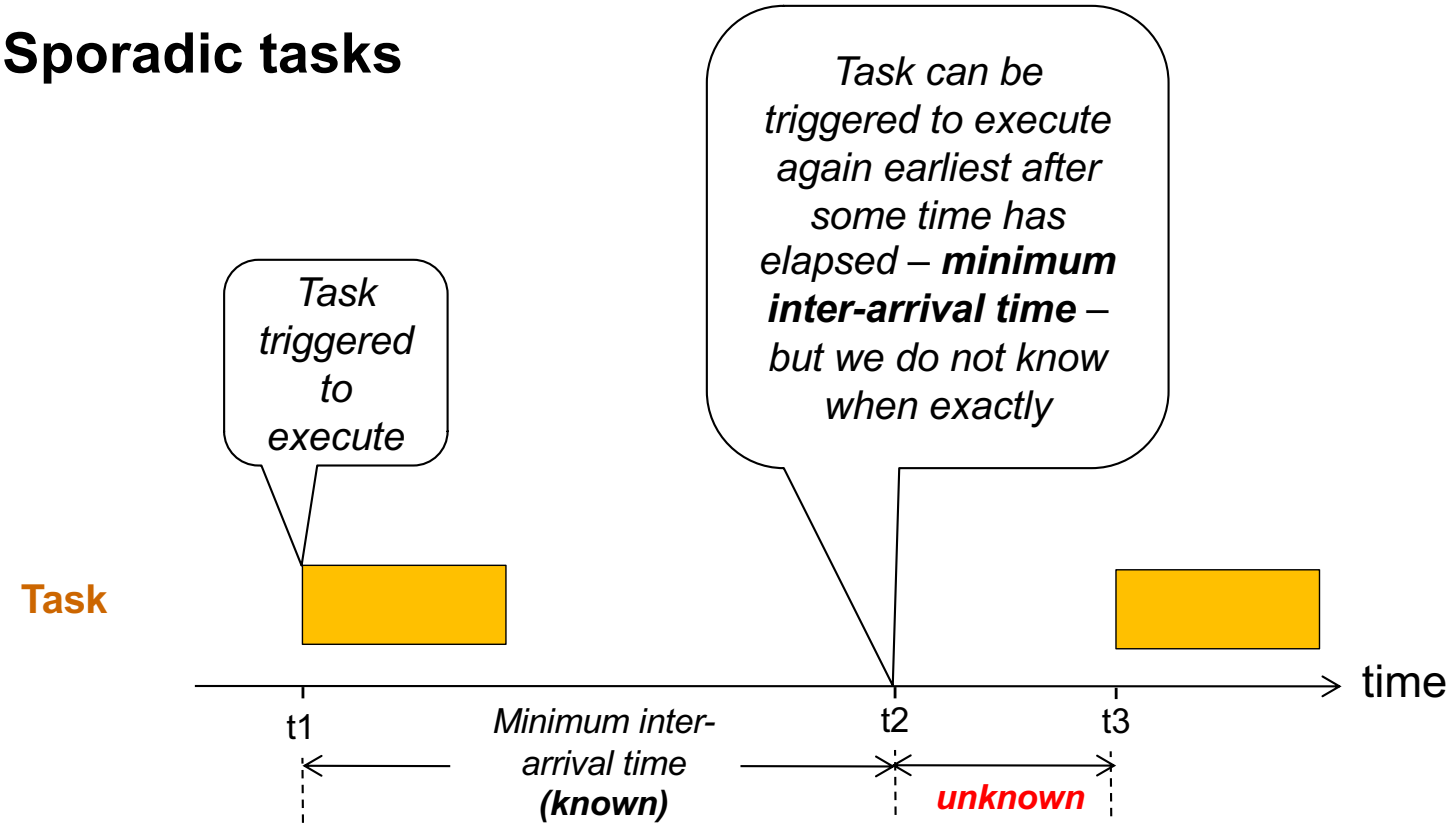


Examples

- Alarm task
- Airbag handling task
- Emergency button handling task

Real-time tasks

Sporadic tasks

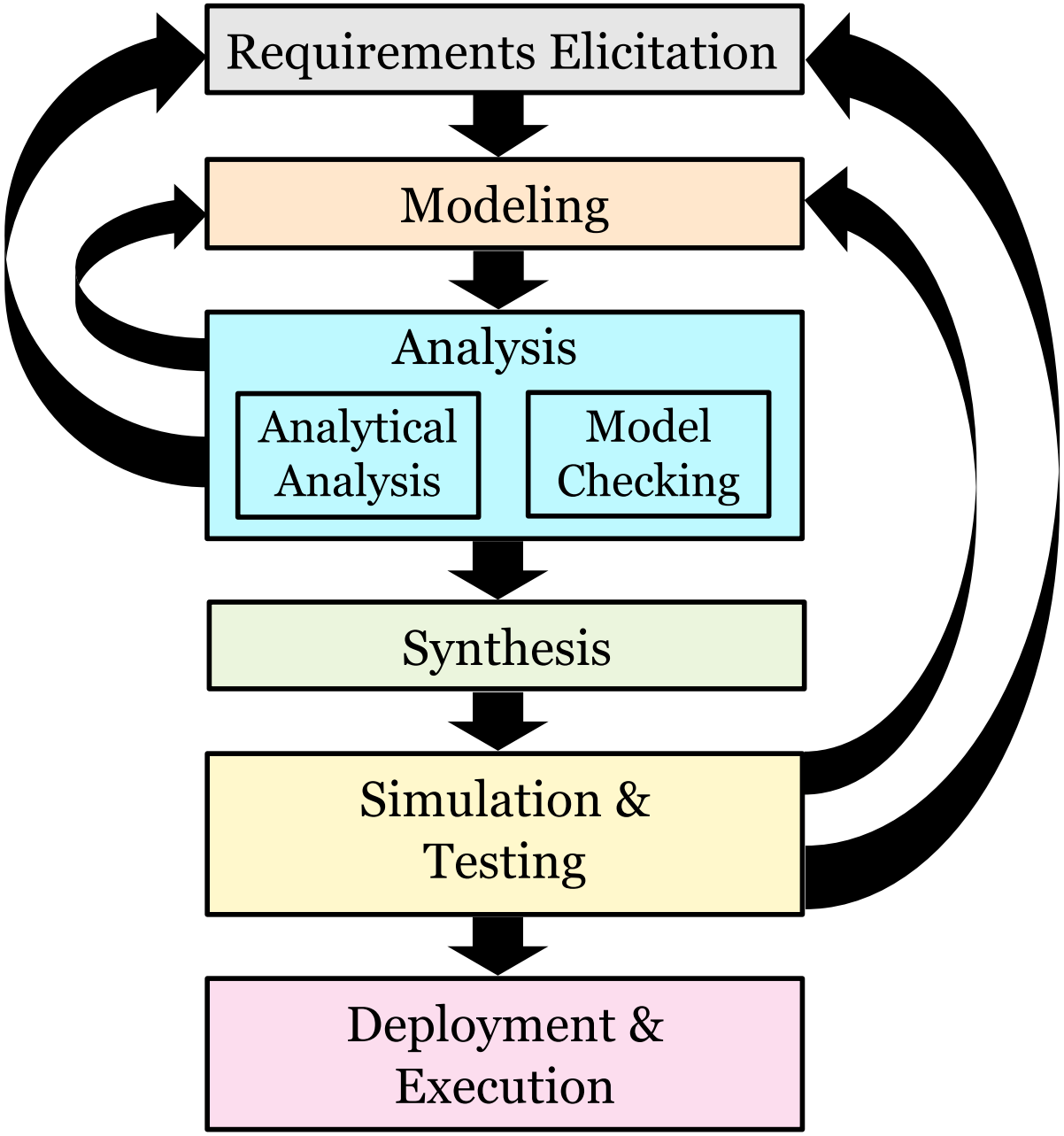


Examples

- Water level regulation task – minimum time to fill the water tank known
- Task handling keyboard input – minimum time between pressing two consecutive keys known


Real-time
Scheduling

Model-based Software Development Process



Real-time scheduling

At any time during execution of an RTOS there is a set of READY tasks.

Only one task can be executing at a given time.

The scheduler must decide which of the READY tasks should run.

Real-time *scheduling* – The process of deciding the execution order of real-time tasks.

Real-time *schedule* – an assignment of tasks to the processor such that all tasks meet their timing constraints.

Why scheduling?

We must have some task execution ordering mechanism!

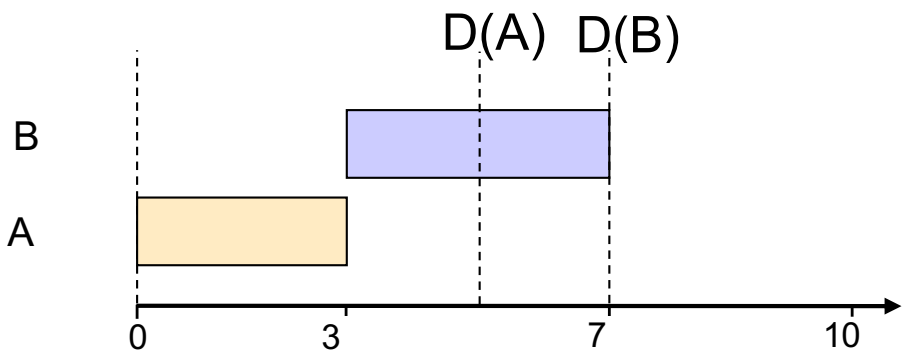
Example:

Assume two tasks are released at the same time:

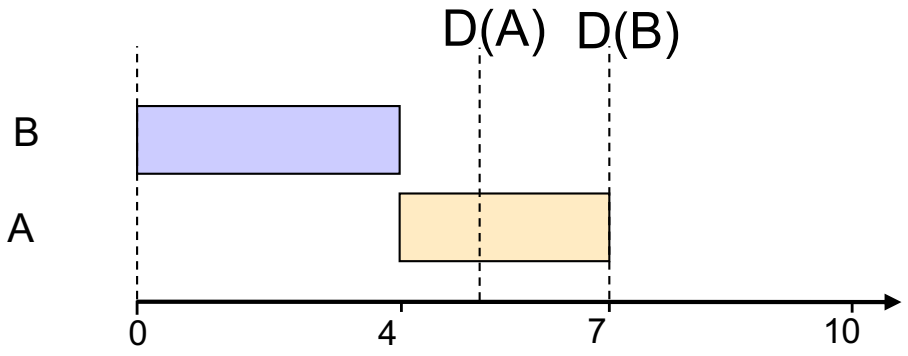
Task A: deadline = 5
execution time = 3

Task B: deadline = 7
execution time = 4

Does it matter in which order we execute the tasks above?



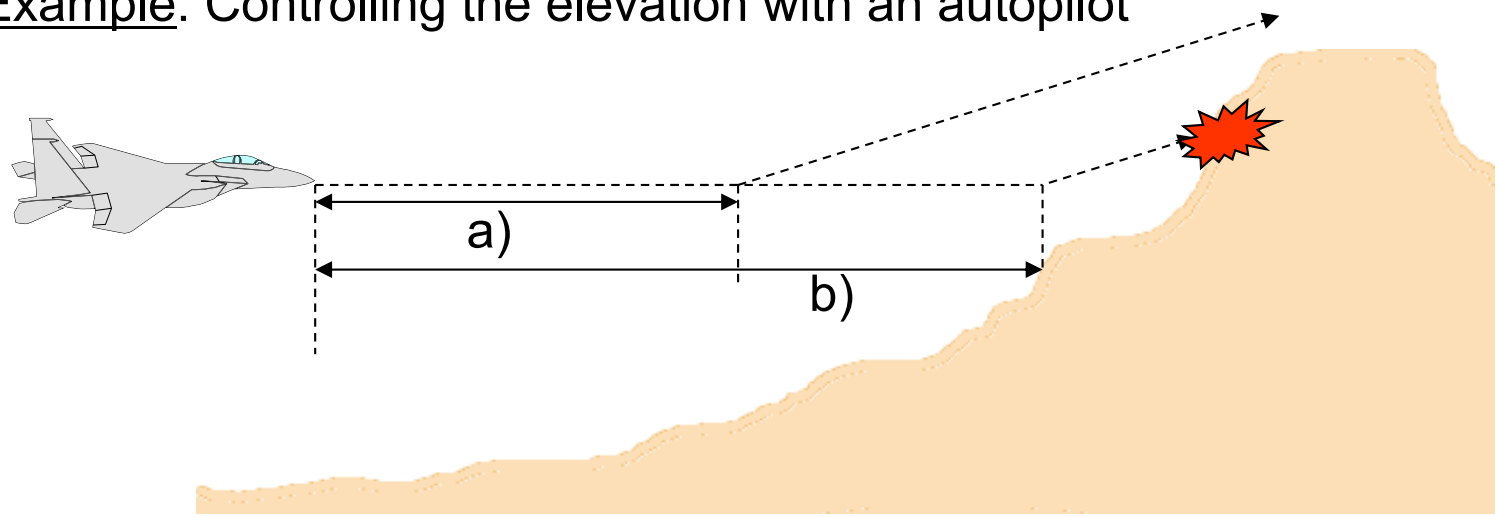
Both deadlines are met



Deadline miss for task A!

Why scheduling?

Example: Controlling the elevation with an autopilot



a) If we check the elevation often enough, we will discover changes in the terrain and have enough time to make corrections.
b) Sampling done too far apart → catastrophic consequences

Sampling too often means waste of resources (CPU). If too much time is spent in controlling the elevation, we might miss hostile missiles heading our way.

Thus it is important to distribute the time and resources in a good way.

Three parts of real-time scheduling

Configuration

- Before start of the system decide **which information will be provided during run-time**, e.g., task priorities.

Run-time dispatching

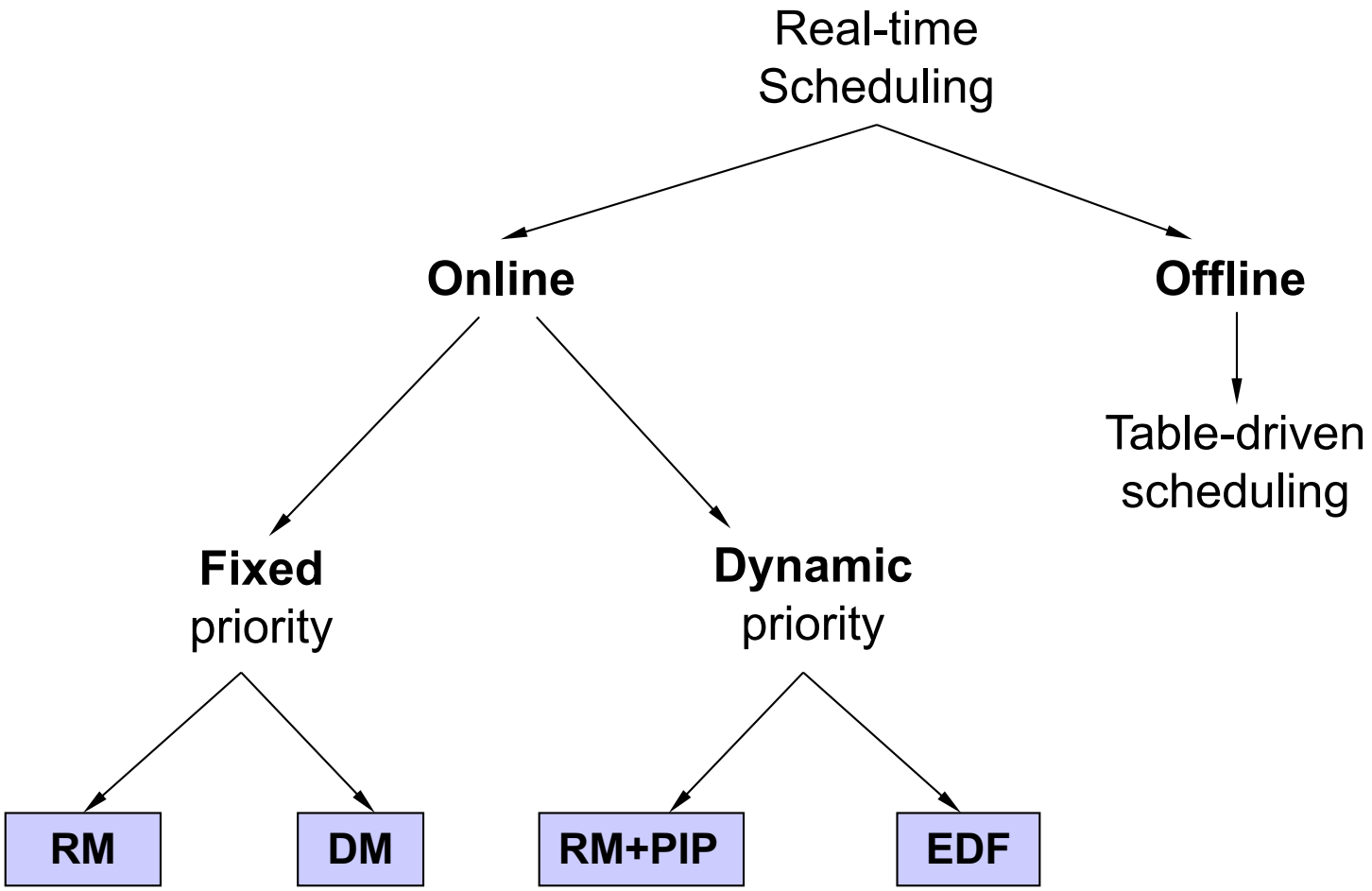
- Decide **how task switching will be performed at runtime**, e.g., according to some specific algorithm.

Analysis

- Provide guarantees *before runtime* that all timing constraints will be fulfilled.
- To be able to perform the analysis, we must know which **configuration** and which **run-time dispatching algorithm** are used.

The main difference between scheduling algorithms is how much of each part above is used

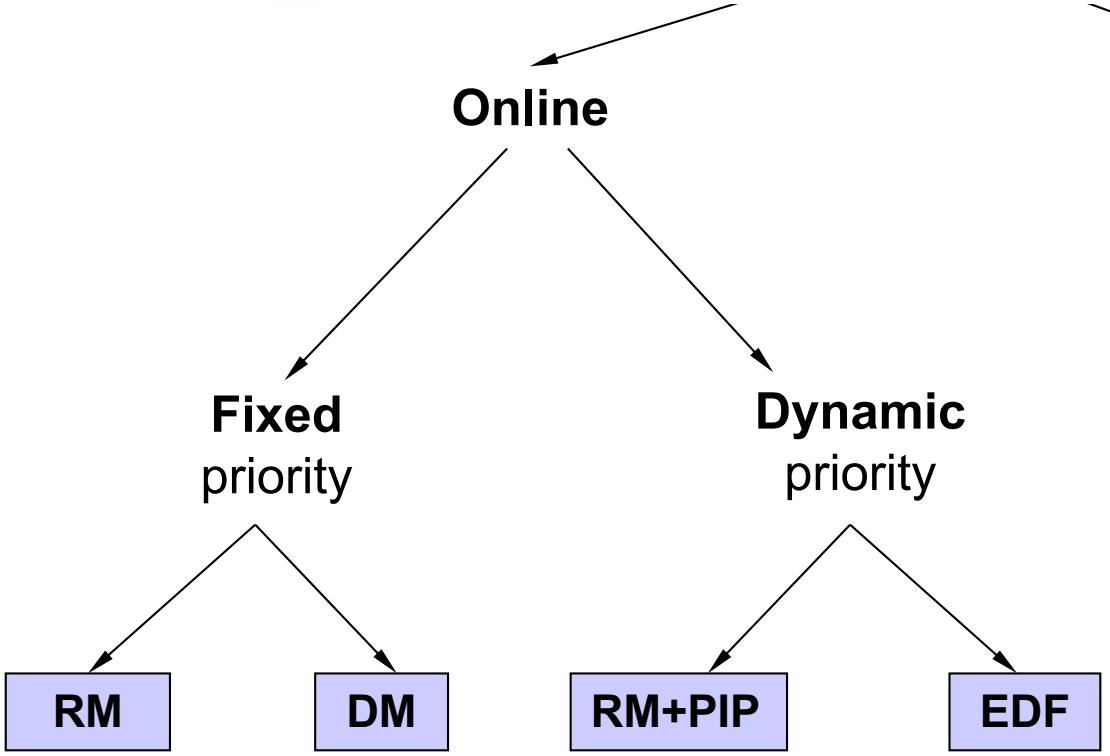
A simple classification of real-time scheduling



RM	Rate Monotonic
DM	Deadline Monotonic
EDF	Earliest Deadline First
PIP	Priority Inheritance Protocol

Go to www.menti.com and use the code: 67368867

Which type of scheduling is more suited to safety-critical systems?

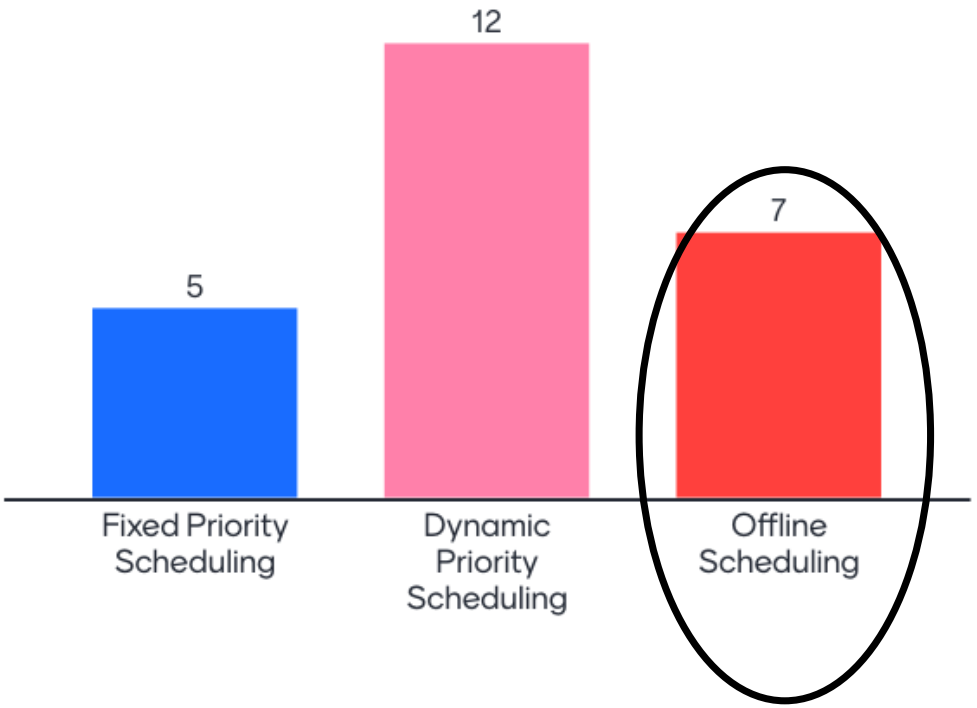


RM	Rate Monotonic
DM	Deadline Monotonic
EDF	Earliest Deadline First
PIP	Priority Inheritance Protocol

Go to www.menti.com and use the code: 67368867

Which type of scheduling is more suited to safety-critical systems?

Mentimeter



24

Correct Answer

DM	Deadline Monotonic
EDF	Earliest Deadline First
PIP	Priority Inheritance Protocol


Real-time
Task Model

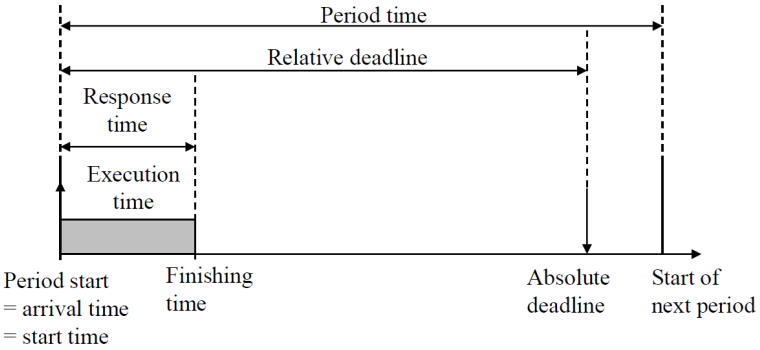
Task Model: Definitions and notation

We use the following notation:

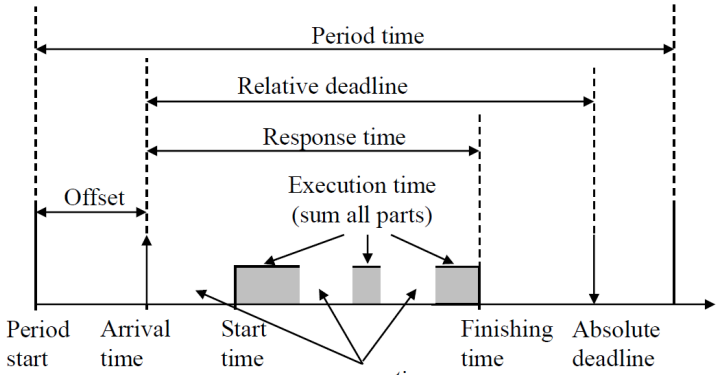
- τ_i – a task *i* (Greek symbol τ is pronounced as "tau")
- τ_i^j – the j^{th} instance (job) of τ_i

Parameters that characterize a task τ_i:

- T_i – ***period time* of the task** : The time between activation of two consecutive task instances
- D_i – ***relative deadline* of the task** : The latest time at which the task has to finish after its release
- C_i – **worst-case *execution time* of the task** : The time necessary for the processor to execute the task without interruptions
- R_i – **worst-case *response time* of the task** : The longest possible time needed to finish the task execution, when all preemptions from higher priority tasks are included
- O_i – ***offset* of the task** : The time interval that specifies how long the release of the task is delayed from the start of its period



a) without offset and without preemptions



b) with offset and with preemptions

Task Model: Definitions and notation

Parameters that characterize a task instance τ_i^j :

- a_i^j – *arrival time* of the instance : The time when a task instance gets activated (becomes ready to execute)
- st_i^j – *start time* of the instance : The time when a task instance starts to run, i.e., the task enters the executing state.
- ft_i^j – *finishing time* of the instance : The time when a task instance has completed its execution.
- c_i^j – actual *execution time* of the instance : The time it takes for a task instance to finish its execution without any interruptions by other task.
- r_i^j – *response time* of the instance : The time interval between the arrival time and the finishing time of a task instance.
- d_i^j – *absolute deadline* of the instance : The latest point in time the instance has to finish its computation and deliver an answer.

Task Model: Definitions and notation

Parameters that characterize a task instance τ_i^j :

a_i^j – ***arrival time* of the instance**: The time when a task instance gets activated (becomes ready to execute)

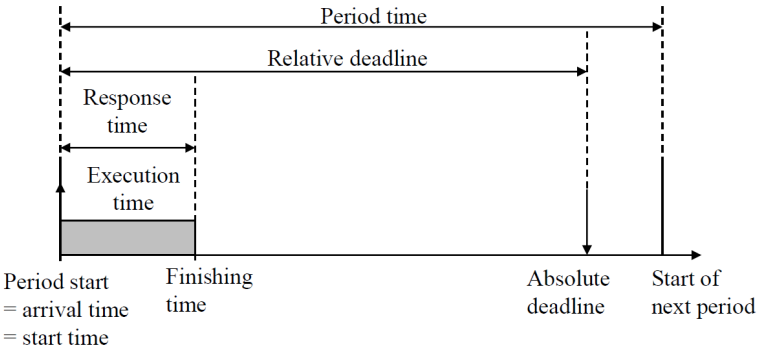
st_i^j – ***start time* of the instance**: The time when a task instance starts to run, i.e., the task enters the executing state.

ft_i^j – ***finishing time* of the instance**: The time when a task instance has completed its execution.

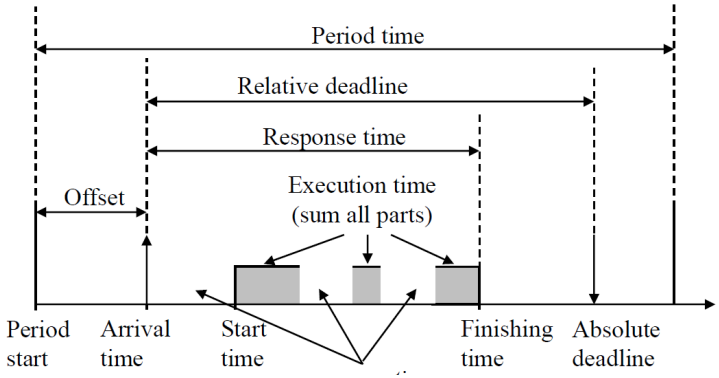
c_i^j – **actual *execution time* of the instance**: The time it takes for a task instance to finish its execution without any interruptions by other task.

r_i^j – ***response time* of the instance**: The time interval between the arrival time and the finishing time of a task instance.

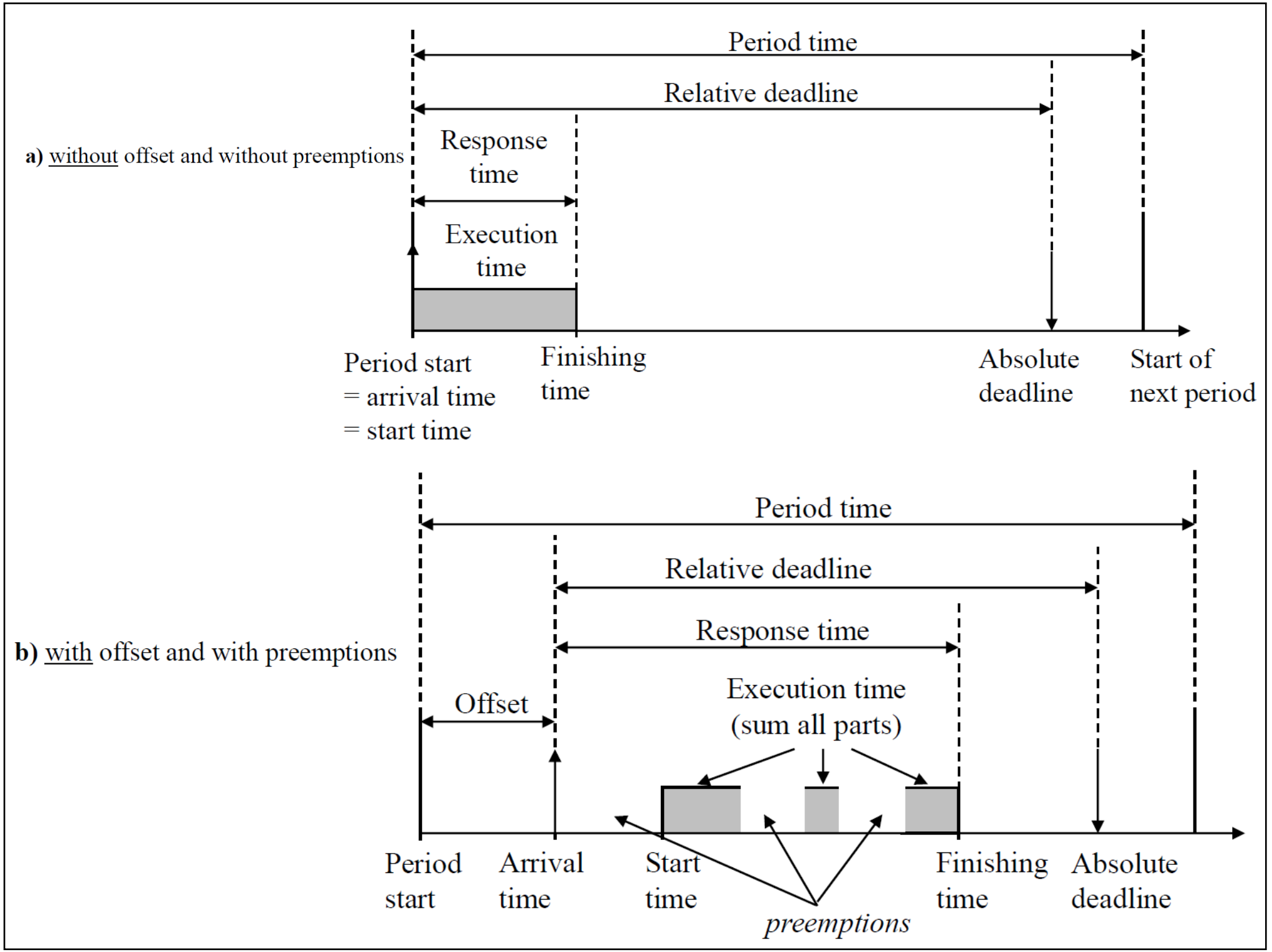
d_i^j – ***absolute deadline* of the instance**: The latest point in time the instance has to finish its computation and deliver an answer.

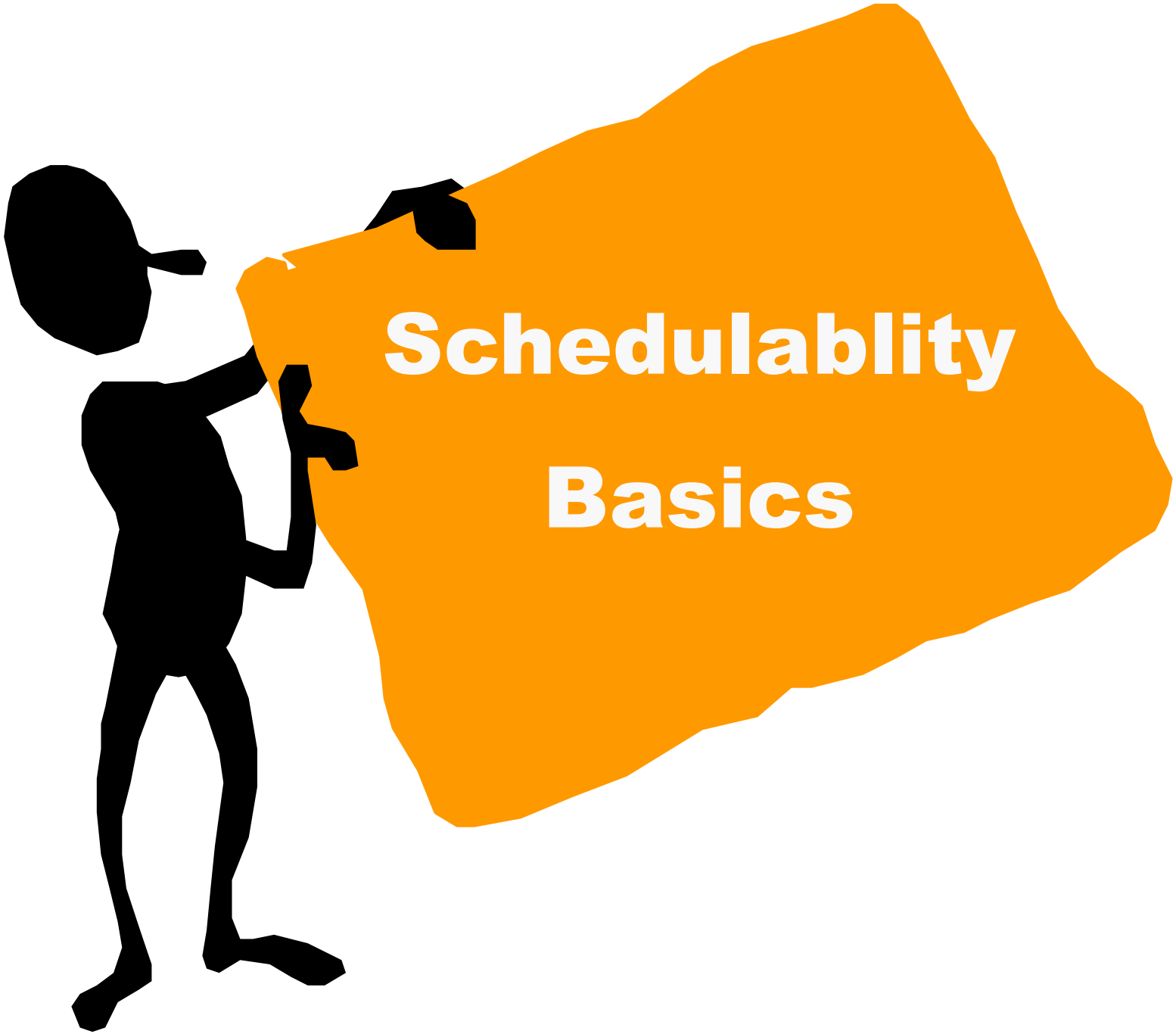


a) without offset and without preemptions



b) with offset and with preemptions


a) without offset and without preemptions
Period time
Relative deadline
Response time
Execution time
Period start = arrival time = start time
Finishing time
Absolute deadline
Start of next period
b) with offset and with preemptions
Period time
Relative deadline
Response time
Offset
Execution time (sum all parts)
Period start
Arrival time
Start time
Finishing time
Absolute deadline
preemptions


Schedulablity
Basics

Feasibility and schedulability

Feasibility

- A *feasible* schedule for a task set is schedule in which all tasks in the set are executed within their **deadlines** and all the other **constraints**, if any, are met.

Schedulability

- A task set is said to be *schedulable* if there exists a feasible schedule for the set.

Schedulability test

A *schedulability test* is used to determine if the task set is schedulable or not with a certain scheduling algorithm.

***Necessary* schedulability test**

- if the test is not passed, the task set is not schedulable

***Sufficient* schedulability test**

- If passed, it will guarantee that the task set is schedulable

***Exact* schedulability test**

- The test is both sufficient and necessary

Schedulability test

	Test Result	
	True	False
Necessary Schedulability Test		
Sufficient Schedulability Test		
Exact Schedulability Test		

Schedulability test

	Test Result	
	True	False
Test		

Schedulability test

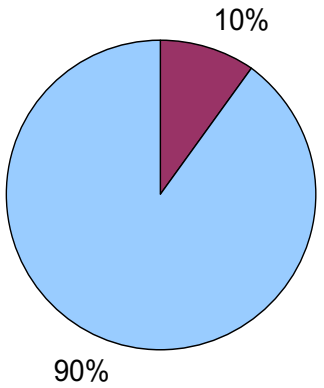
	Test Result	
	True	False
Necessary Schedulability Test	**Not a reliable answer**	**Reliable Answer**
Sufficient Schedulability Test	**Reliable Answer**	**Not a reliable answer**
Exact Schedulability Test	**Reliable Answer**	**Reliable Answer**

CPU utilization

A portion of CPU time needed to execute a task set

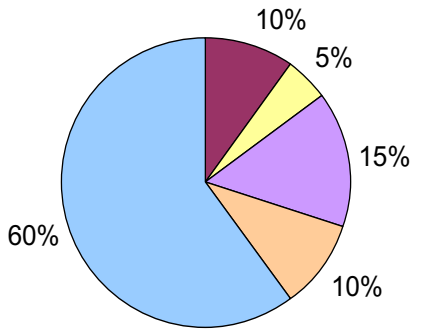
Utilization for one task τ_i :

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



The total utilization for all the tasks in the system (n tasks):

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} = \sum_{i=1}^{n} \frac{C_i}{T_i}$$



CPU utilization

Example:

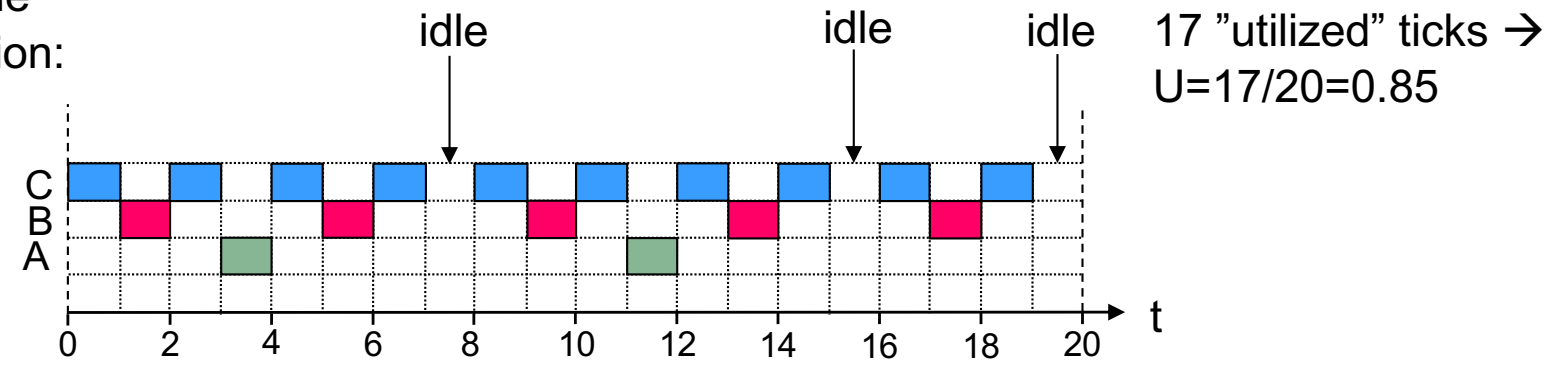
Calculate the utilization for the following task set:

Task	T	C	P
A	10	1	Low
B	4	1	Med
C	2	1	High

$$U = \frac{C_A}{T_A} + \frac{C_B}{T_B} + \frac{C_C}{T_C} = \frac{1}{10} + \frac{1}{4} + \frac{1}{2} = \frac{17}{20} = 0.85$$

Let us make an execution trace of this task set. **How long the execution trace should be?**

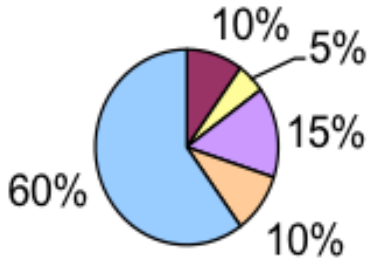
Possible execution:



17 ”utilized” ticks →
U=17/20=0.85

CPU utilization

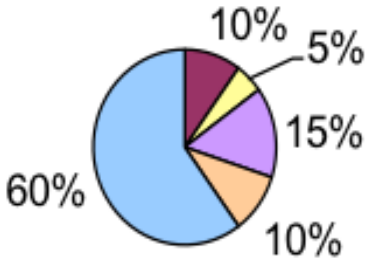
Does low utilization mean that the system is schedulable?



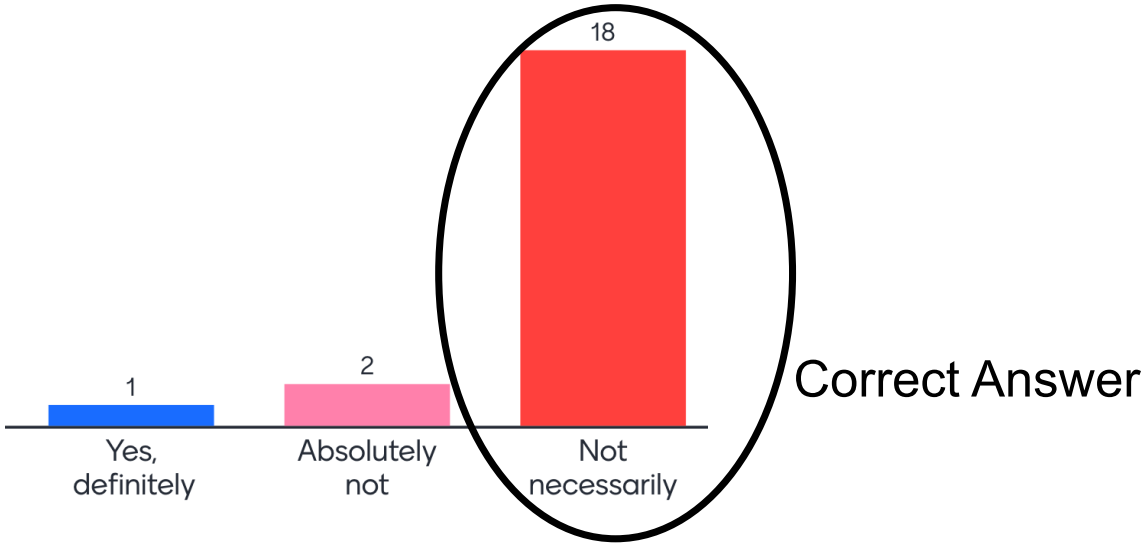
Go to www.menti.com and use the code: 67368867

CPU utilization

Does low utilization mean that the system is schedulable?

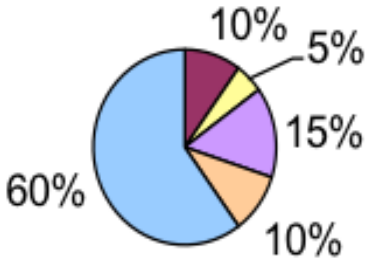


Go to www.menti.com and use the code: 67368867



CPU utilization

Does low utilization mean that the system is schedulable?

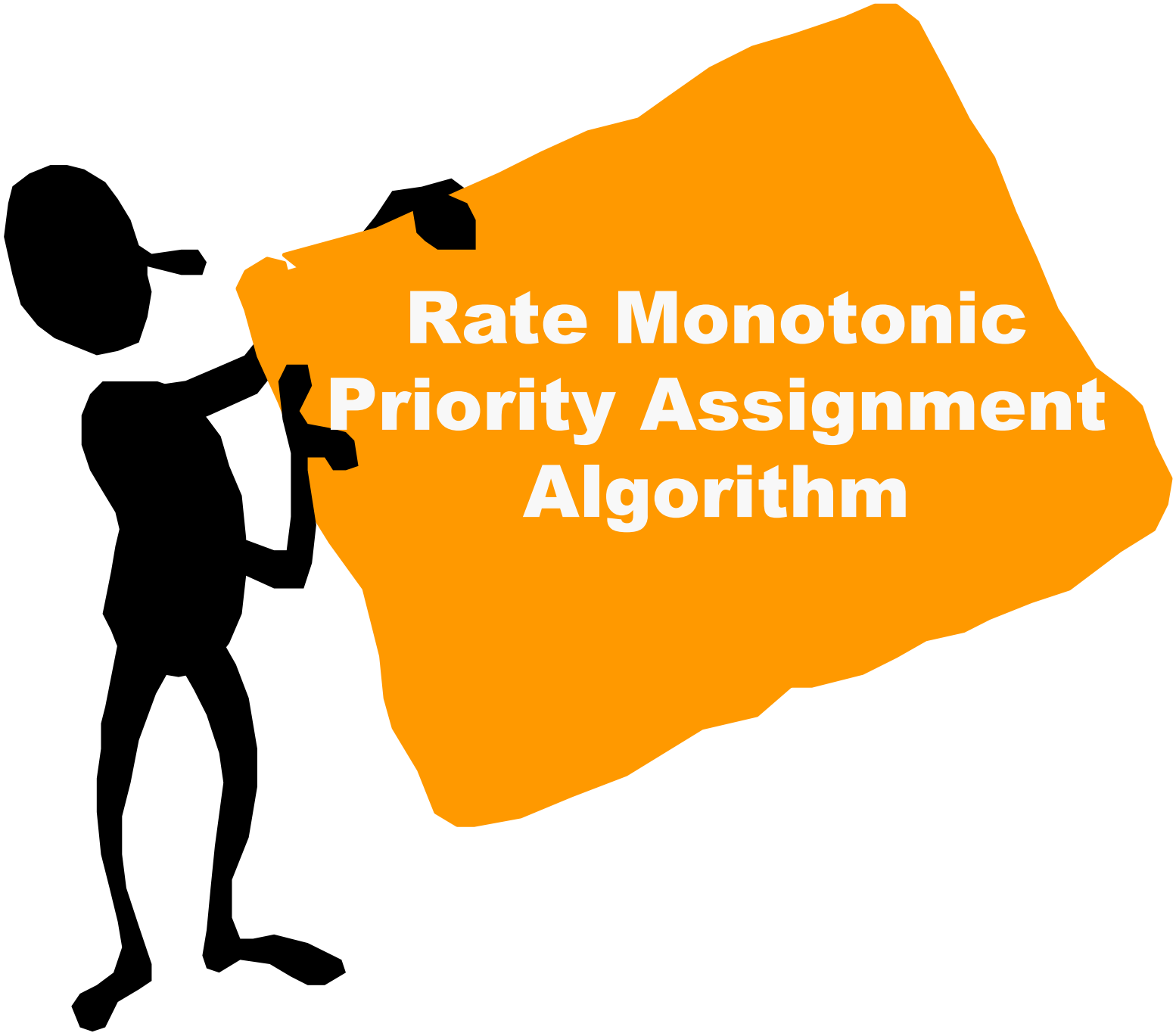


Not necessarily!

Example:

Is the following task set schedulable?

Task	T	C	D
A	1000	6	10
B	1000	6	6


Rate Monotonic
Priority Assignment
Algorithm

Rate Monotonic (RM)

Assumptions

- All tasks are periodic
- deadline = period
- Independent tasks (no communication)

Configuration

- Static priorities – assigned to tasks before run-time, and do not change at run-time (unless some priority inheritance protocol is used)
- Rule: *the shorter the period, the higher the priority*

Run-time

- Online scheduling decisions
- The task with the highest priority among all ready tasks will execute

Rate Monotonic - Analysis

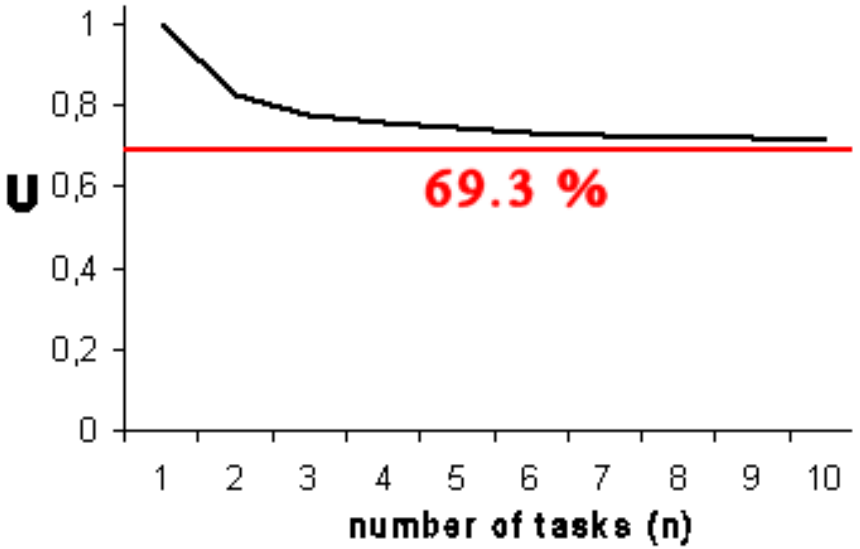
Schedulability condition: $U \leq n(2^{1/n} - 1)$

If the condition is fulfilled → then the set is schedulable!

If not → we do not know! (maybe schedulable, maybe not, we need other analysis methods)

Utilization upper bounds:

Nr of tasks (n)	Upper bound
1	100 %
2	82.8 %
3	78.0 %
4	75.7 %
5	74.3 %
10	71.8 %
11	71.5 %
12	71.3 %
13	71.2 %
14	71.1 %
...	...
∞	69.3 %



Rate Monotonic – Example 1

Task	T	C
A	3	1
B	6	1
C	5	1
D	10	2

a) Calculate CPU utilization for the task set.

b) Is system schedulable according to Rate Monotonic?

c) Assign task priorities according to Rate-Monotonic.

d) Show the execution trace for the tasks and draw the produced schedule (up to LCM)

Rate Monotonic – Example 1

Is the system schedulable with Rate Monotonic?

Prio(1) Highest ←
Prio(3) ←
Prio(2) ←
Prio(4) ← Lowest

Task	T	C
A	3	1
B	6	1
C	5	1
D	10	2

$$U = \frac{1}{3} + \frac{1}{6} + \frac{1}{5} + \frac{2}{10} = \frac{27}{30} = 0.9$$

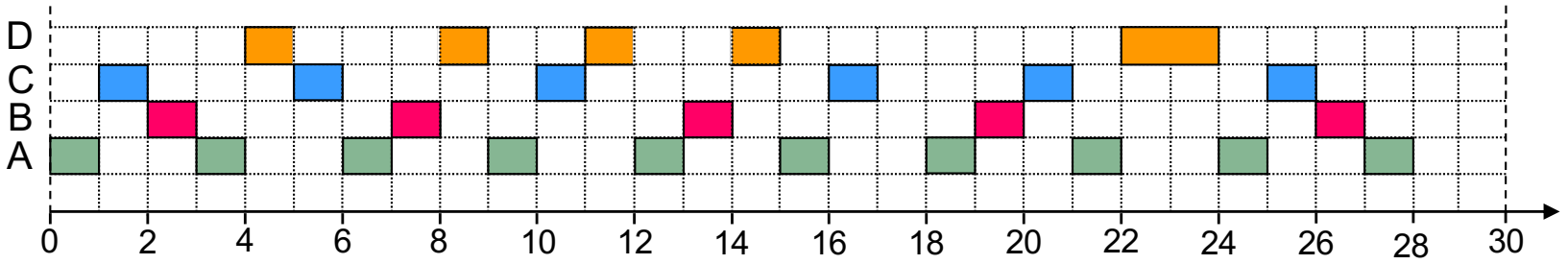
$$n(2^{1/n} - 1) = 4(2^{1/4} - 1) = 0.75$$

$$U \leq n(2^{1/n} - 1)$$ **false!**

The condition 0.9 < 0.75 is false but the tasks still meet their deadlies(!?) (see below)

⇒ The condition is **sufficient** but **not neccessary**!

Schedule:



Go to www.menti.com and use the code: 67368867

Assume that a task set having utilization (U <= 69%) is scheduled with Rate Monotonic scheduling. Is the schedulability test $U \leq n(2^{1/n} - 1)$

Sufficient

Necessary

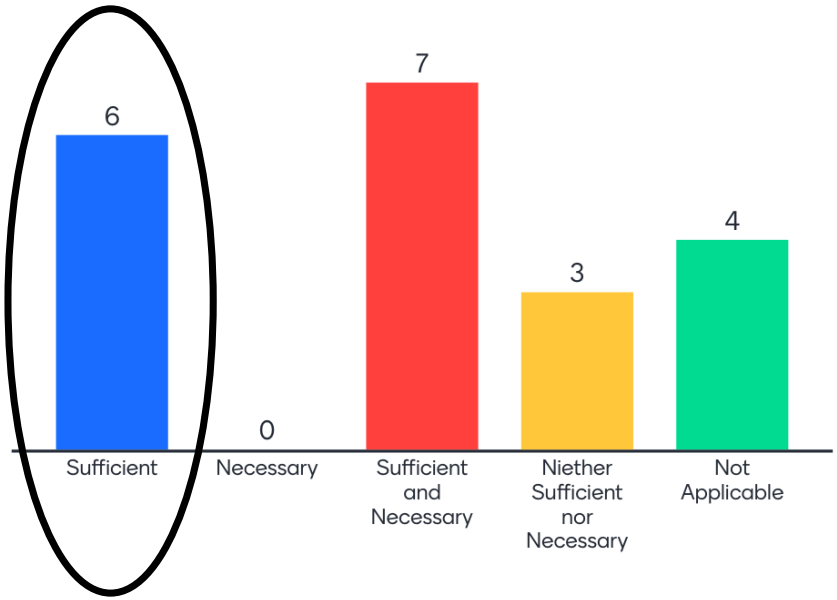
Sufficient and Necessary

Niether Sufficient nor Necessary

Not Applicable

Go to www.menti.com and use the code: 67368867

Assume that a task set having utilization (U <= 69%) is scheduled with Rate Monotonic scheduling. Is the schedulability test $U \leq n(2^{1/n} - 1)$



Correct Answer

Go to www.menti.com and use the code: 67368867

Assume that a task set having utilization (U <= 50%) is scheduled with Rate Monotonic scheduling. Is the schedulability test $U \leq n(2^{1/n} - 1)$

Sufficient

Necessary

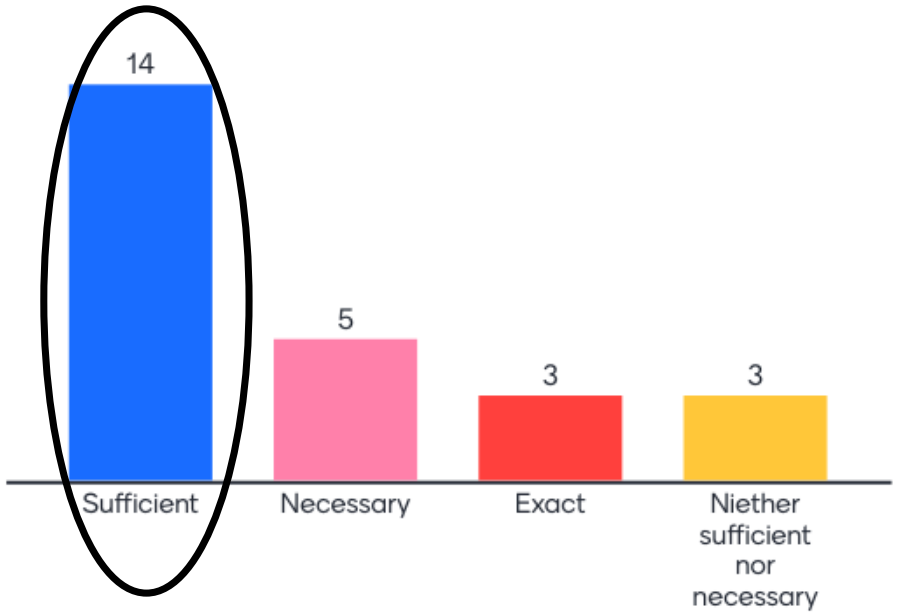
Sufficient and Necessary

Niether Sufficient nor Necessary

Not Applicable

Go to www.menti.com and use the code: 67368867

Assume that a task set having utilization (U <= 50%) is scheduled with Rate Monotonic scheduling. Is the schedulability test $U \leq n(2^{1/n} - 1)$



Correct Answer

25

Rate Monotonic – Example 1

Why do we need schedulability tests if we can check the schedulability of a task set by drawing its execution trace?

Prio(2) ←	C	5	1
Prio(4) ←	D	10	2

Lowest

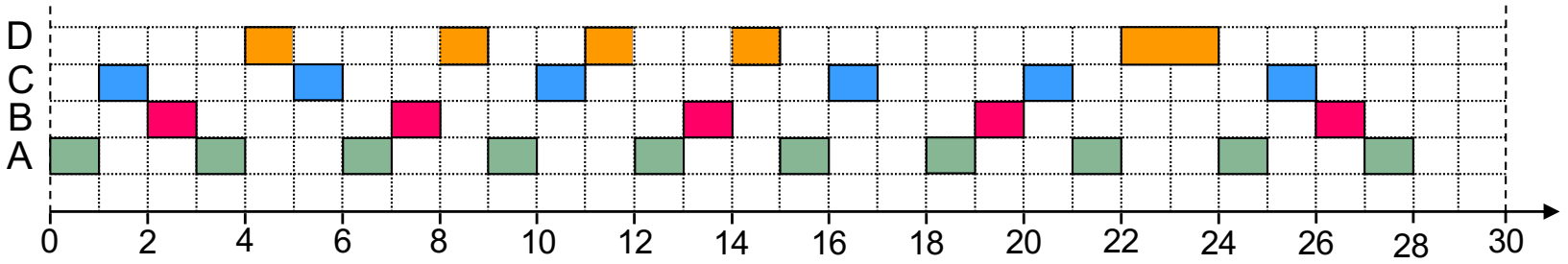
$$n(2^{1/n}-1) = 4(2^{1/4}-1) = 0.75$$

$$U < n(2^{1/n}-1)\,?$$ **false!**

The condition 0.9 < 0.75 is false but the tasks still meet their deadlies(!?) (see below)

$\Rightarrow$ The condition is **sufficient** but **not neccessary**!

Schedule:



Go to www.menti.com and use the code: 67368867

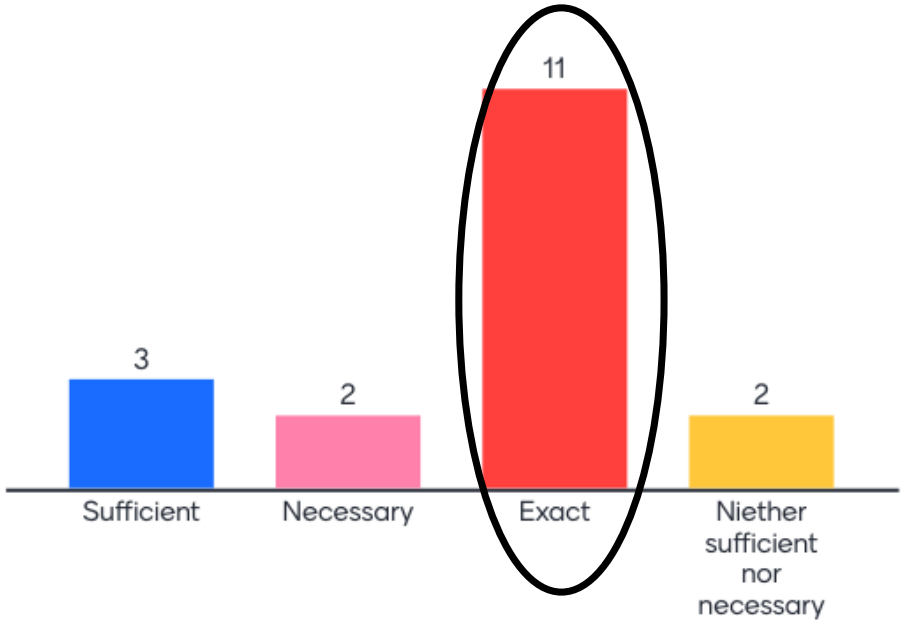
Checking the schedulability of a taskset by drawing an exeuction trace until hyperperiod (LCM) can be considered a schedulability test. Is this test

Sufficient	Necessary	Exact	Niether sufficient nor necessary

Go to www.menti.com and use the code: 67368867

Mentimeter

Checking the schedulability of a taskset by drawing an exeuction trace until hyperperiod (LCM) can be considered a schedulability test. Is this test



Correct Answer

18

Rate Monotonic – Example 2

Task	T	C
A	3	1
B	6	1
C	5	1

Home practice

a) Assign task priorities according to Rate-Monotonic.

b) Calculate CPU utilization for the task set.

c) Is system schedulable according to Rate Monotonic?

d) Show the execution trace for the tasks and draw the produced schedule (up to LCM)

Critical Instant and Critical Time Zone

Definition: **Critical instant** of a task is the time at which the release of a task will produce the largest response time

Definition: **Critical time zone** of a task is the interval between the critical instant and the response time of the corresponding task instance

Critical Instant and Critical Time Zone

Theorem1: A critical instant for an arbitrary task is the time at which all higher priority tasks are released at the same time.

Consider a periodic task set

$\Gamma = \{T_1, T_2, T_3, \ldots, T_n\}$;

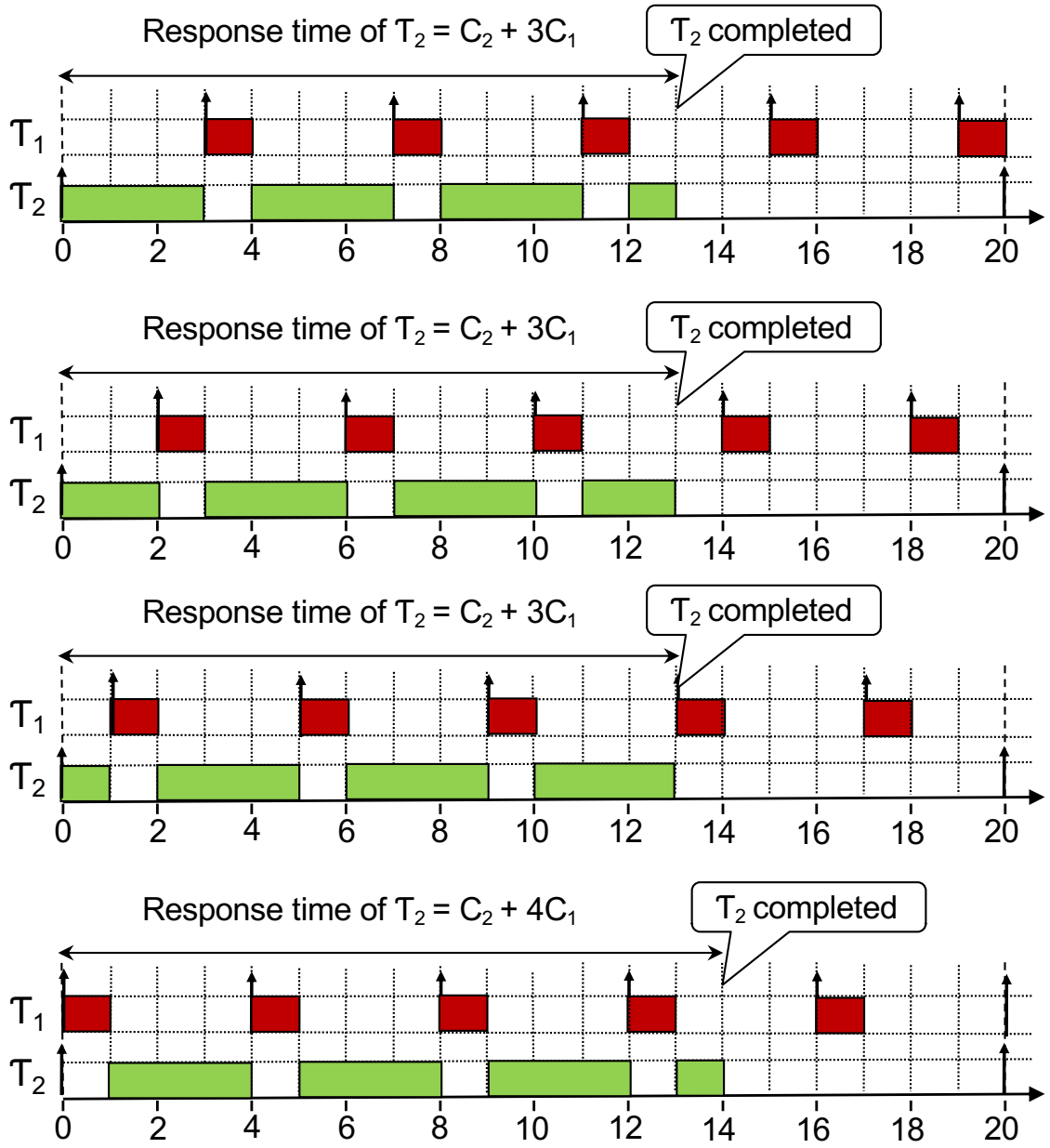
such that $T_2 > T_1$; $T_n > T_{n-1}$

Task	T	C	
T_1	4	C_1=1	→ High priority
T_2	20	C_2=10	→ Low priority

Hyper period: LCM(4,20)=20

Response time of T_2 is largest when T_1 is realesed at the same time.

Similarly, response time of T_n is largest when T_{n-1}, T_{n-2}, …, T_2, T_1 are released at the same time.




Earliest
Deadline
First
Scheduling

Earliest Deadline First – EDF

Each task instance receives an absolute deadline:

$$d_i^j = a_i^j + D_i$$

At any time, the processor is assigned to the instance with the earliest absolute deadline.

Conditions

- Independent tasks
- Deadline = period
- Release time = start of period

Analysis

- All tasks will meet their deadlines if:

$$U \leq 1$$

Exact (necessary and sufficient) schedulability test for task sets scheduled with EDF

Earliest Deadline First – Example

Task	T=D	C
A	5	2
B	7	4

Schedulable with RM? And EDF?

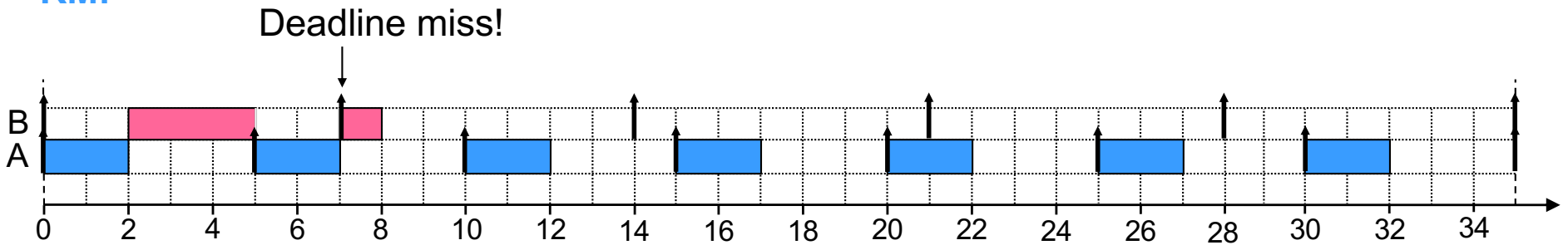
$$U = \frac{2}{5} + \frac{4}{7} = \frac{34}{35} = 0.97$$

$$U \leq n(2^{1/n} - 1) \quad \textbf{False!}$$

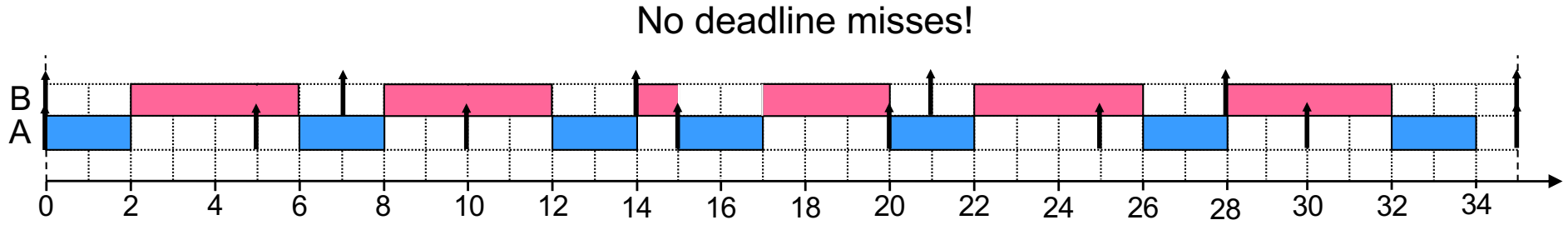
$2(2^{1/2}-1) = 0.828$

$$U \leq 1 \quad \textbf{True!}$$

RM:



EDF:



Go to www.menti.com and use the code: 67368867

Assume that the schedulability test U <= 1 is applied to a task set that is scheduled with Rate Monotonic. Is this test

Sufficient

Necessary

Sufficient and Necessary

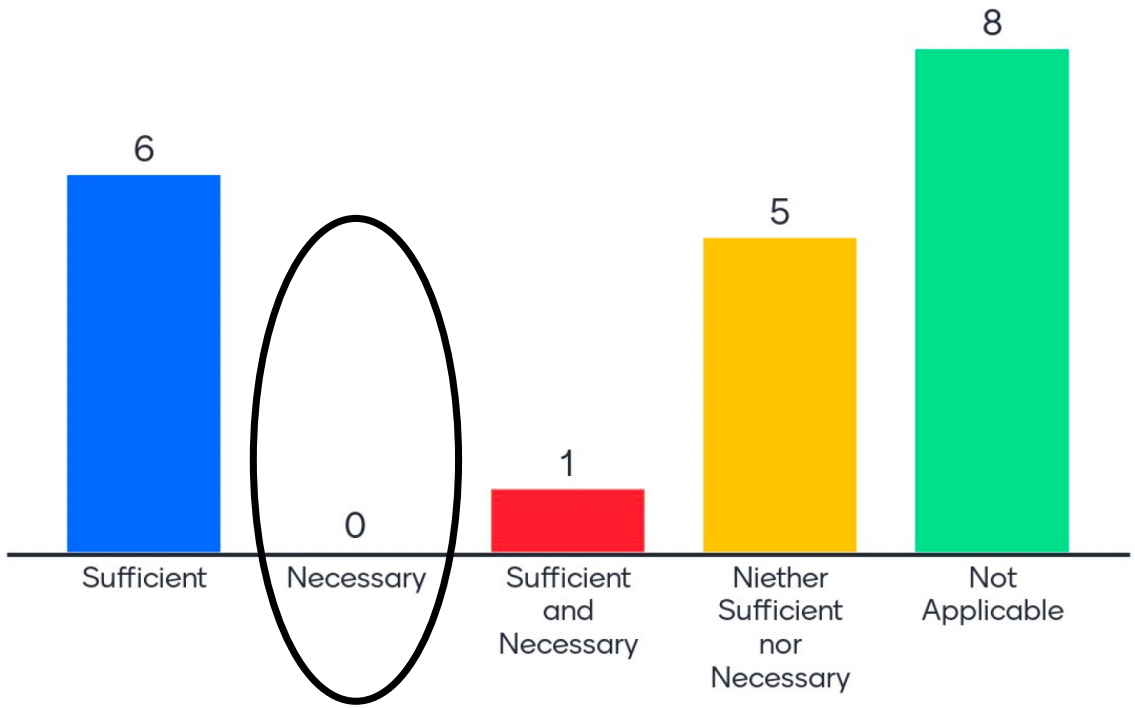
Niether Sufficient nor Necessary

Not Applicable

Go to www.menti.com and use the code: 67368867

Assume that the schedulability test U <= 1 is applied to a task set that is scheduled with Rate Monotonic. Is this test

Mentimeter



Correct Answer

Processor Demand Analysis (PDA)

Schedulability analysis of periodic tasks with *deadlines less than periods* under EDF can be performed using the *Processor Demand Analysis* (PDA).

Generally, the processor demand for a task τ_i in any given time interval *[t, t+L]* is the amount of processing time required by τ_i in *[t, t+L]* that has to be completed at ,or before, *t+L.*

With deadlines, this is the processing time required *in [t, t+L]* that has to be executed with deadlines ≤ *t+L*

PDA – Deadlines less than periods

Sufficient and necessary condition for EDF scheduling where D<T:

$$\forall L \in D : L \geq \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i$$

where

$$D = \{ d_i^j \mid d_i^j = jT_i + D_i, d_i^j < LCM, 1 \leq i \leq n, j \geq 0 \}$$

i.e., we check intervals from 0 to each of the deadlines, until we either have checked intervals up to the LCM or the condition fails for some deadline

$$\forall L \in D : L \geq \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i$$

PDA – Example

Check the EDF schedulability of the following task set:

Task	T	D	C
A	6	4	3
B	8	7	4

$D = \{d_i^j \mid d_i^j = jT_i + D_i, d_i^j < LCM, 1 \leq i \leq n, j \geq 0\}$

LCM (6,8) = 24

Deadlines up to the LCM: D =

$C^T(0,4) =$

$$C^T(0,7) = \left(\left\lfloor \frac{7-4}{6} \right\rfloor + 1 \right) 3 + \left(\left\lfloor \frac{7-7}{8} \right\rfloor + 1 \right) 4 = 3 + 4 = 7 \leq 7$$

$$C^T(0,10) = \left(\left\lfloor \frac{10-4}{6} \right\rfloor + 1 \right) 3 + \left(\left\lfloor \frac{10-7}{8} \right\rfloor + 1 \right) 4 = 6 + 4 = 10 \leq 10$$

$$C^T(0,15) = \left(\left\lfloor \frac{15-4}{6} \right\rfloor + 1 \right) 3 + \left(\left\lfloor \frac{15-7}{8} \right\rfloor + 1 \right) 4 = 6 + 8 = 14 < 15$$

$$C^T(0,16) = \left(\left\lfloor \frac{16-4}{6} \right\rfloor + 1 \right) 3 + \left(\left\lfloor \frac{16-7}{8} \right\rfloor + 1 \right) 4 = 9 + 8 = 17 > 16$$

Deadline miss!

Not schedulable!

$$\forall L \in D : L \geq \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i$$

PDA – Example

Check the EDF schedulability of the following task set:

Task	T	D	C
A	6	4	3
B	8	7	4

$D = \{ d_i^j \mid d_i^j = jT_i + D_i, d_i^j < LCM, 1 \leq i \leq n, j \geq 0 \}$

LCM (6,8) = 24

Deadlines up to the LCM: D = *{ 4,7,10,15,16,22,23 }*

$$C^T(0,4) = \left(\left\lfloor \frac{4-4}{6} \right\rfloor + 1 \right) 3 + \left(\left\lfloor \frac{4-7}{8} \right\rfloor + 1 \right) 4 = (0+1)3 + \left(\left\lfloor \frac{-3}{8} \right\rfloor + 1 \right) 4 = 3 + (-1+1)4 = 3 \leq 4$$

$$C^T(0,7) = \left(\left\lfloor \frac{7-4}{6} \right\rfloor + 1 \right) 3 + \left(\left\lfloor \frac{7-7}{8} \right\rfloor + 1 \right) 4 = 3 + 4 = 7 \leq 7$$

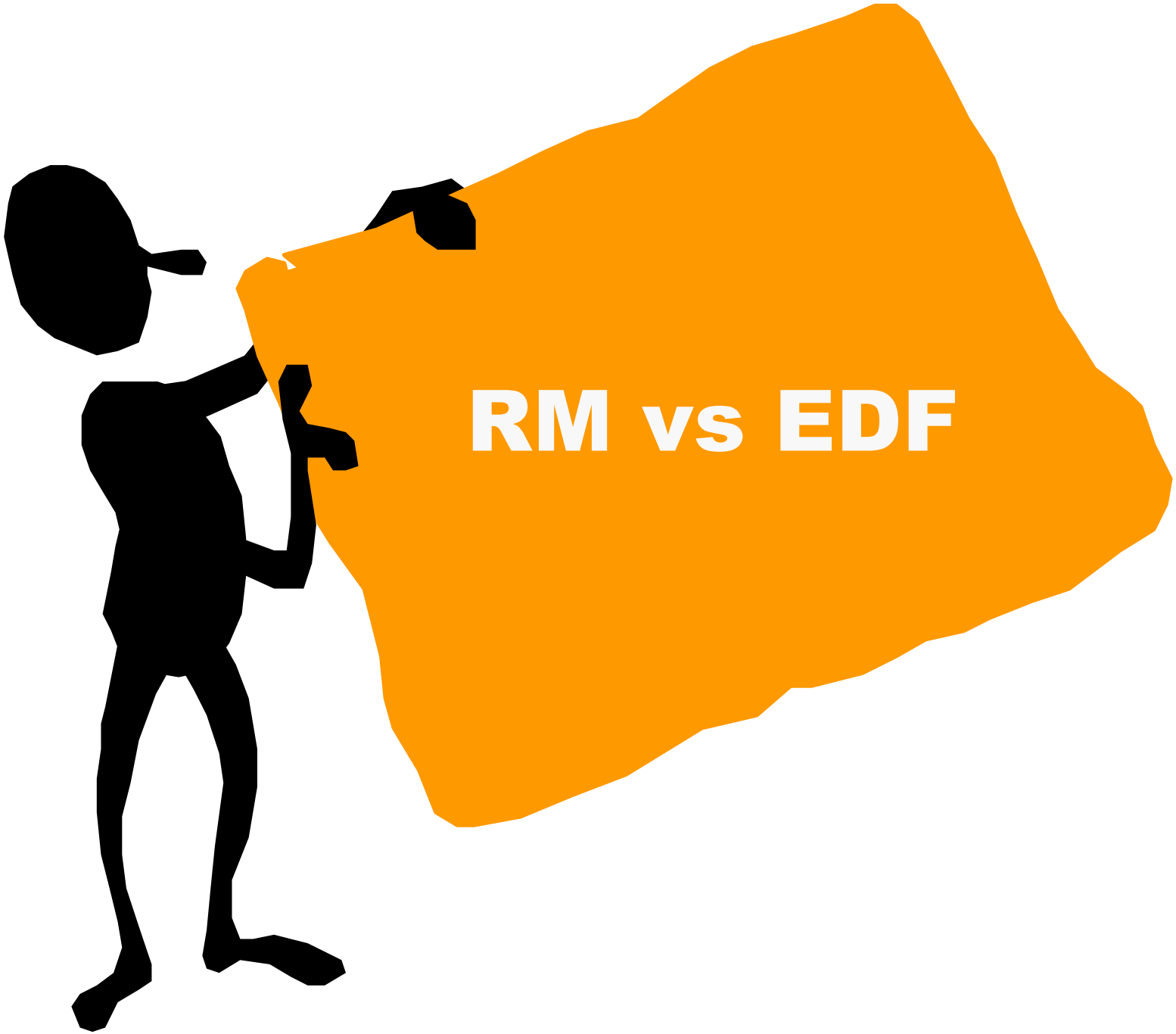
$$C^T(0,10) = \left(\left\lfloor \frac{10-4}{6} \right\rfloor + 1 \right) 3 + \left(\left\lfloor \frac{10-7}{8} \right\rfloor + 1 \right) 4 = 6 + 4 = 10 \leq 10$$

$$C^T(0,15) = \left(\left\lfloor \frac{15-4}{6} \right\rfloor + 1 \right) 3 + \left(\left\lfloor \frac{15-7}{8} \right\rfloor + 1 \right) 4 = 6 + 8 = 14 < 15$$

$$C^T(0,16) = \left(\left\lfloor \frac{16-4}{6} \right\rfloor + 1 \right) 3 + \left(\left\lfloor \frac{16-7}{8} \right\rfloor + 1 \right) 4 = 9 + 8 = 17 > 16$$

Deadline miss!

Not schedulable!


RM vs EDF

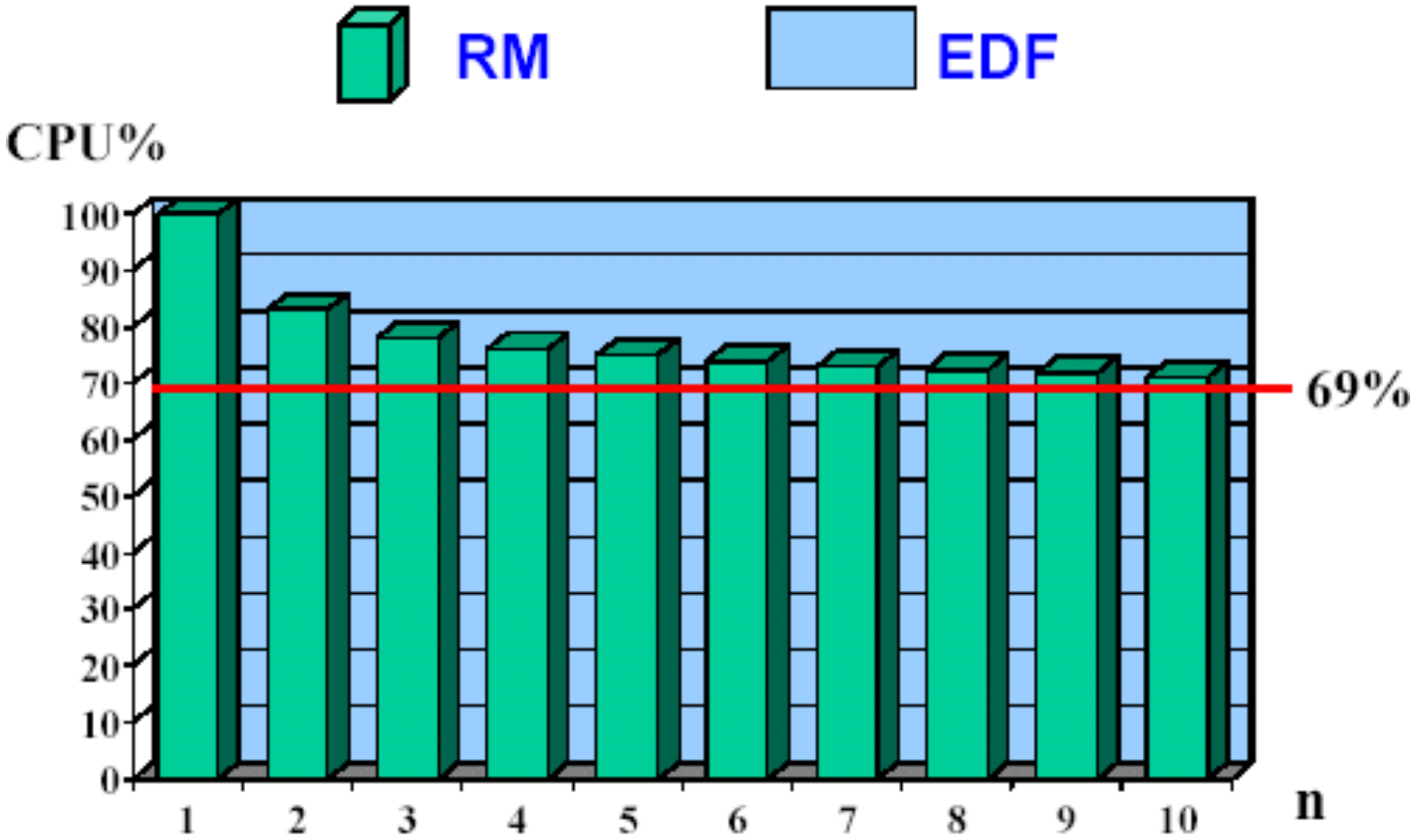
RM vs EDF

We compare EDF and RM with respect to:

- Processor utilization
- Implementation complexity
- Runtime overhead
- Jitter

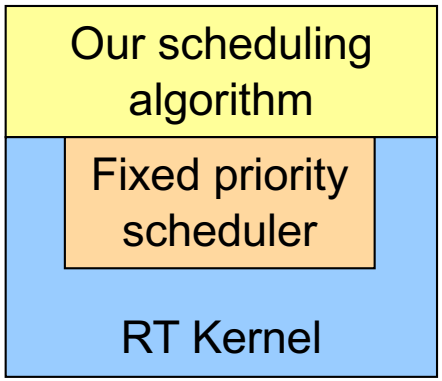
RM vs EDF: Processor utilization

EDF utilizes the CPU better than RM



RM vs EDF: Implementation complexity

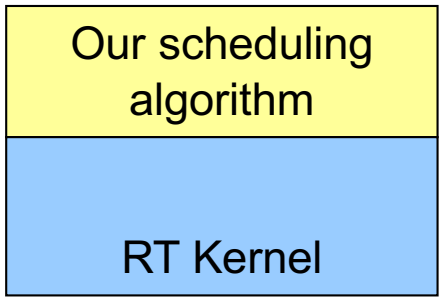
Case 1: on top of existing fixed-priority scheduler



RM: straight-forward

EDF: needs re-mapping of priorities under run-time

Case 2: implementation from scratch

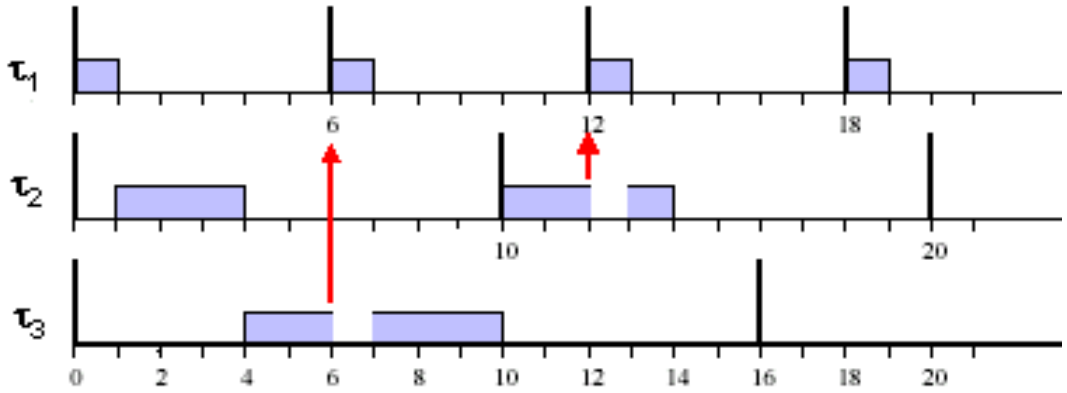


Same complexity

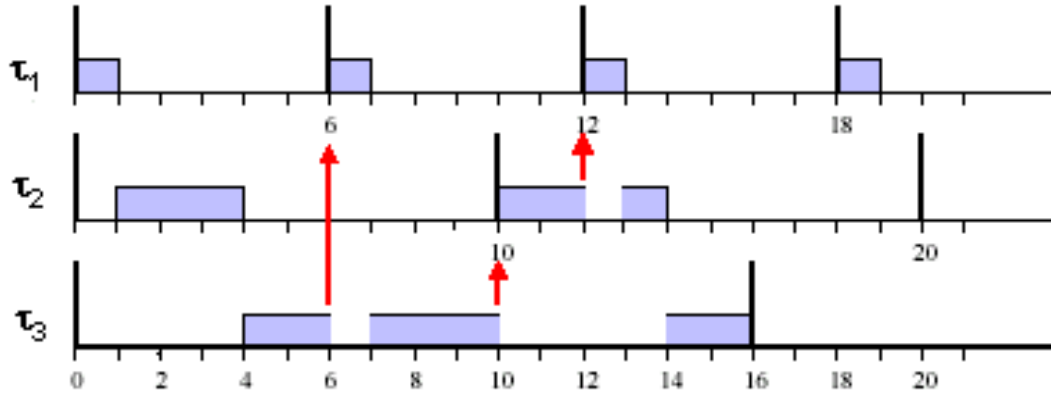
RM vs EDF: Run-time overhead

EDF has higher overhead for **task release** since absolute deadline must be updated for each instance. RM has higher **context-switch** overhead due to more preemptions.

Example RM:

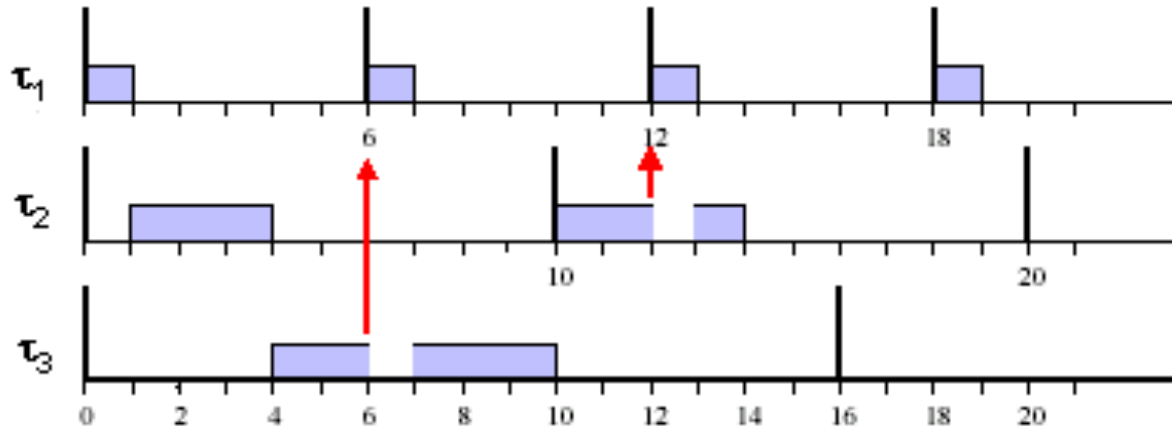


If we increase the execution time of τ_3 we get 3 preemtions instead of 2:

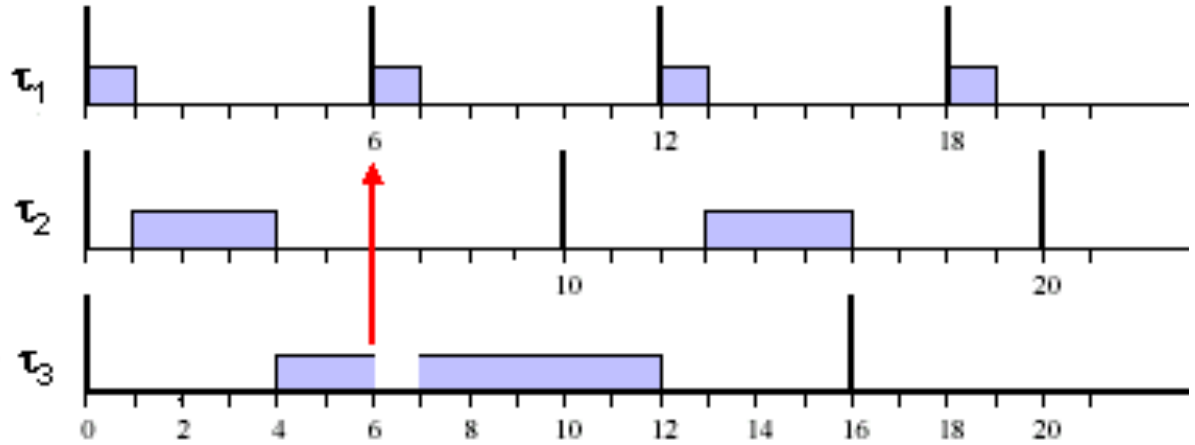


RM vs EDF: Run-time overhead

Example EDF:

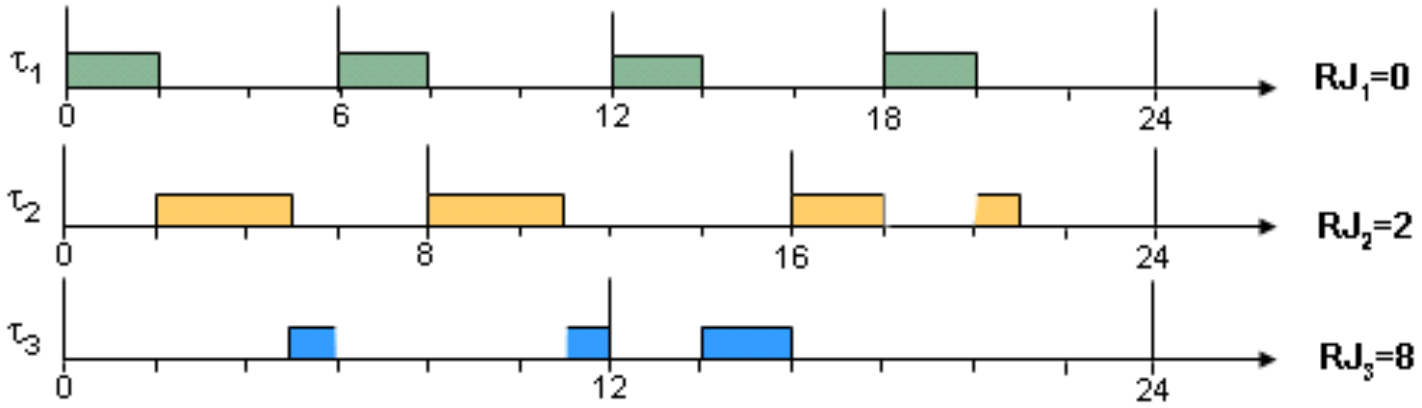


If we increase the execution time of τ_3, we get only 1 preemption!

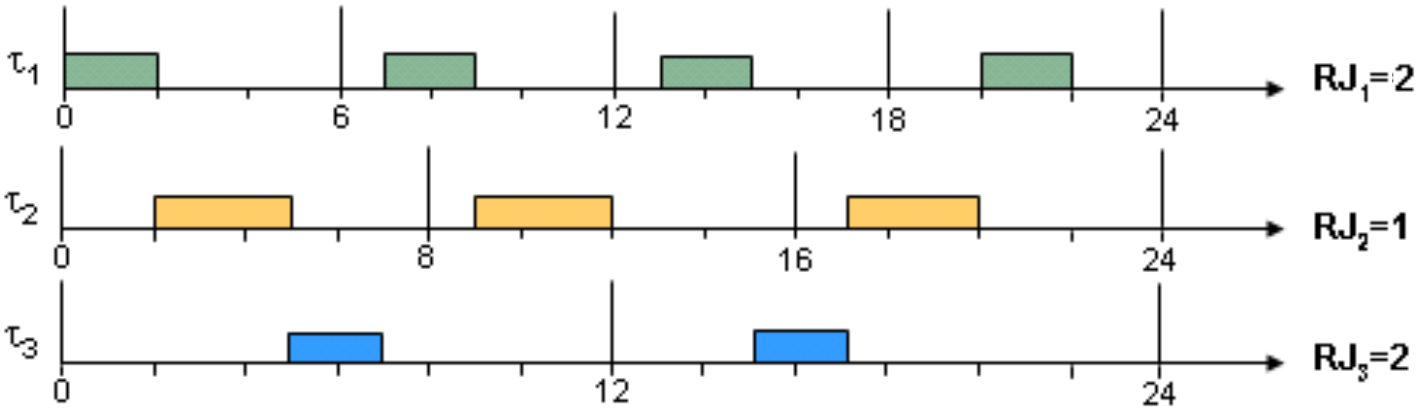


RM vs EDF: Effects of jitter

Response time jitter under RM: No jitter for τ_1 but τ_3 experiences very high jitter.



Jitter under EDF: For a little increase of RJ_1 we get large decrease for RJ_3.



RM vs EDF: conclusions

RM and EDF have ***same implementation complexity*** – a small additional overhead is needed in EDF to update the absolute deadlines of instances.

RM *is supported by commercial RTOSs* – a big advantage of RM is that it can be **easily implemented on top of fixed priority kernels**.

Runtime overhead is smaller in EDF – smaller number of context switches.

EDF utilizes the processor better than RM – EDF achieves **full processor utilization**, 100%, whereas RM only guarantees 69%

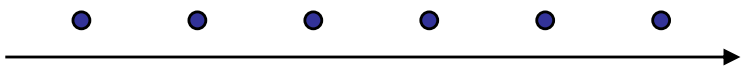
EDF is simpler to analyze if D = T, RM is simpler for D<T

EDF is *fair* in reducing jitter, whereas RM only reduces the jitter of the *highest* priority tasks

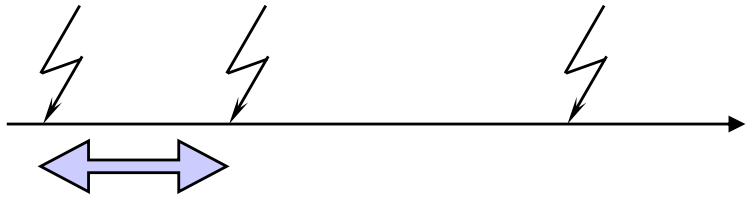
EDF is more efficient than RM for handling *aperiodic* tasks

Non-periodic task scheduling

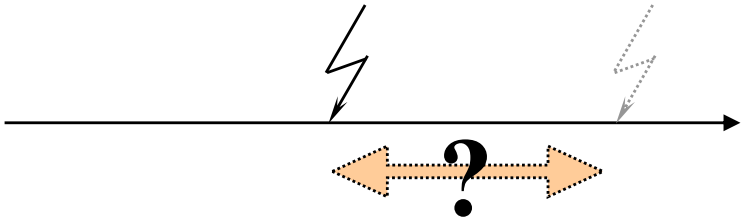
***Periodic* tasks**



***Sporadic* tasks**



***Aperiodic* tasks**



So far we have talked about periodic events and tasks.

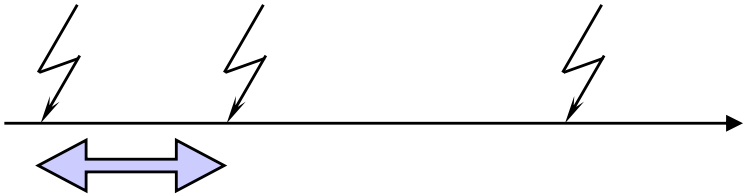
What about the others?


Aperiodic
Task
Scheduling

Non-periodic task scheduling

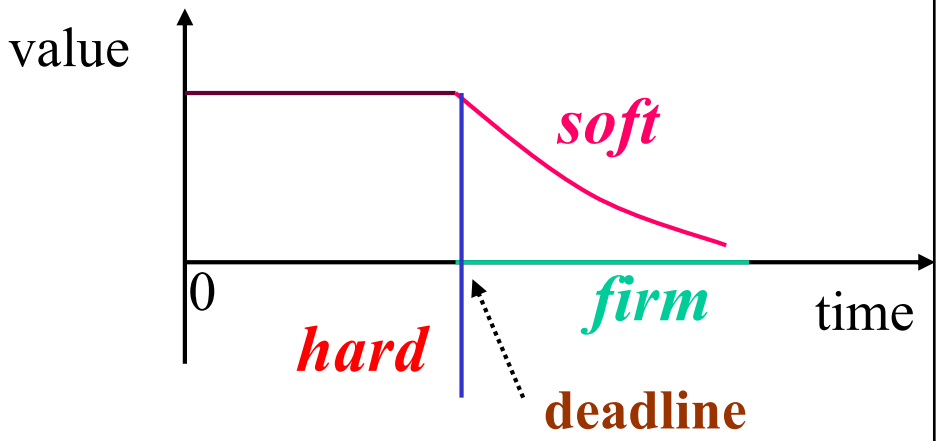
Sporadic

- Aperiodic with *minimum inter-arrival* time known
- Worst case: all sporadic tasks have highest frequency
- If we can schedule worst case, we can schedule all other



Aperiodic

- *Soft* aperiodic
  - The result has (some) value even after deadline
- ***Firm*** aperiodic:
  - The result has zero value after deadline
- *Hard aperiodic:*
  - missing a deadline may be catastrophic



Non-periodic task scheduling

Non-periodic tasks are typically activated by interrupts

- We want to reduce the response times of non-periodic tasks.
- We do not want to jeopardize schedulability of periodic tasks.

Handling criticality

- Soft aperiodic tasks
  - tasks should be executed as soon as possible, but without jeopardizing hard tasks.
- Firm aperiodic tasks
  - Online guarantee upon arrival
  - guarantee deadline or don't start
- Hard aperiodic tasks
  - Offline guarantee
- Sporadic tasks
  - possible if we can bound inter-arrival times
  - can be guaranteed as periodic tasks with period equal to minimum inter-arrival time

Background scheduling

What is the minimum we can do for aperiodic tasks in a periodically scheduled system, e.g., in a RM schedule?

Background service

- execute aperiodic tasks when no periodic ones are executing
- no disturbance of periodic tasks (and their feasibility)
- simplest method to handle soft aperiodic tasks

Can be used only if

- Aperiodic tasks do not have stringent timing constraints (soft aperiodic)
- Periodic load is not too high – poor response time if high periodic load

Background scheduling

Example:

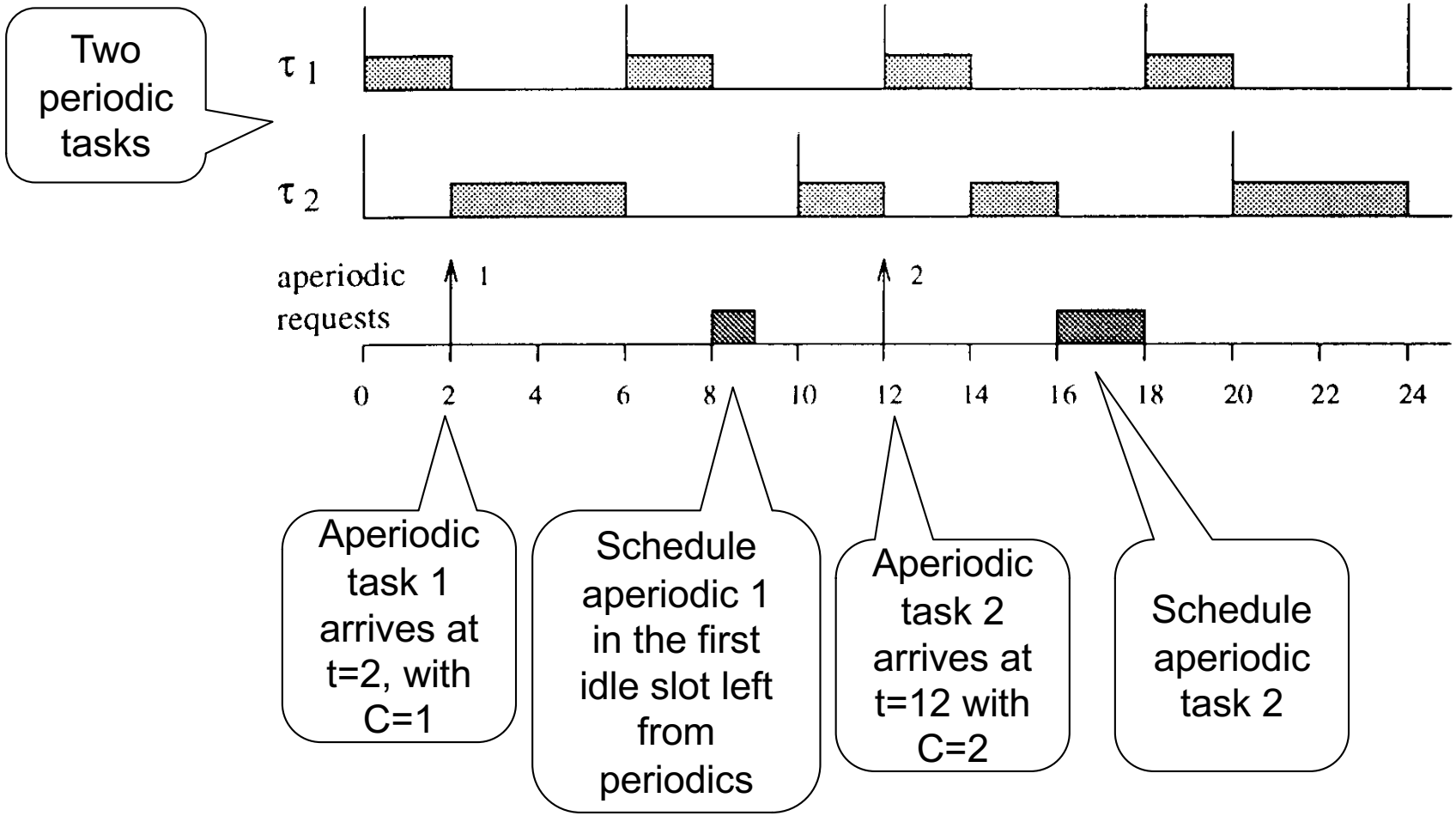


*Figure from Giorgio Buttazzo, Hard Real-time computing systems, Kluwer Academic Publishers

Background scheduling

Easy to implement, requires two queues:

- High priority for periodic tasks (priority-based, e.g., Rate Monotonic)
- Low priority for aperiodic (e.g., First Come First Serve)
- Periodic tasks pre-empt aperiodic ones

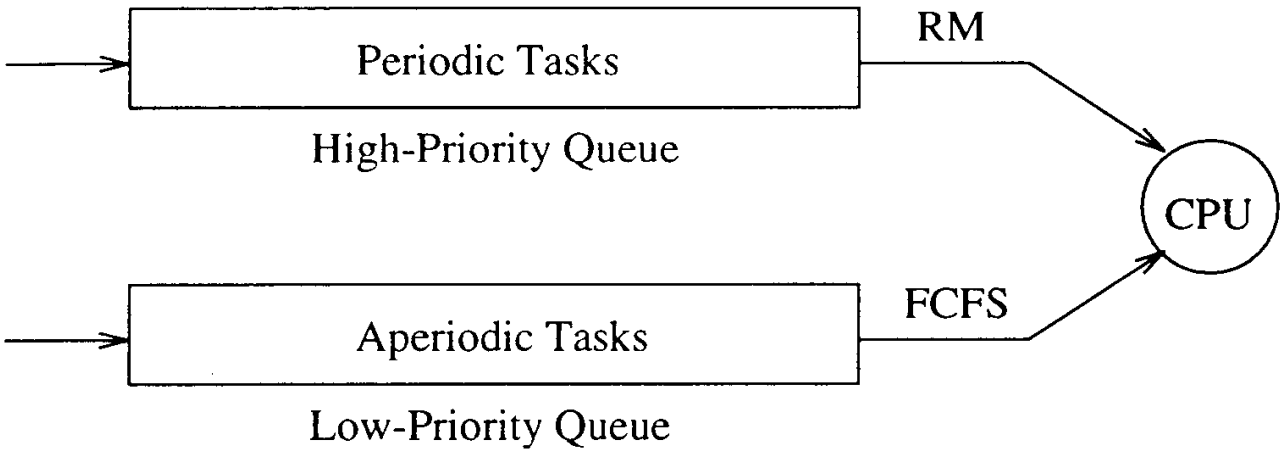


**Figure from Giorgio Buttazzo, Hard Real-time computing systems, Kluwer Academic Publishers*

Aperiodic servers

Background scheduling lives from "left overs" of periodic tasks, without guarantees

- How can we guarantee that at least a certain amount of processing goes to aperiodic tasks?

Answer: Aperiodic servers

- A server is a kernel activity aimed at controlling the execution of aperiodic tasks
- A sever task serves aperiodic requests as soon as possible
- Normally, a server task is implemented as a periodic task having two parameters:

  C_s – execution time, also called *capacity* (or budget) of the server

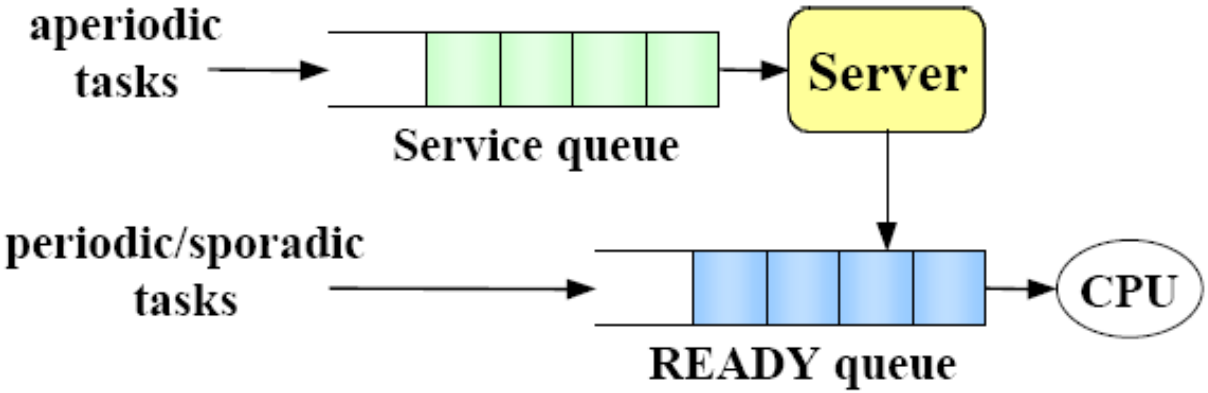
  T_s – server period

- To preserve periodic tasks, *no more* than C_s units must be executed every period T_s

Aperiodic servers

The server is scheduled as any other periodic task.

Aperiodic tasks can be selected using an arbitrary queuing policy.



Polling Server – PS

Improves response times over background scheduling.

Can provide an on-line guarantee.

Polling server algorithm

- At periods T_s server task becomes active
- At the beginning of each period, the server capacity C_s is recharged to its maximum value.
- When the server executes, C_s is consumed:
  - If there are pending aperiodic requests, they are served until C_s is greater than 0
  - If there are no pending aperiodic requests, C_s is discharged to zero, i.e., the server does not execute

- Note: If aperiodic request arrives just **after** server suspends it has to wait for next server period

Polling Server

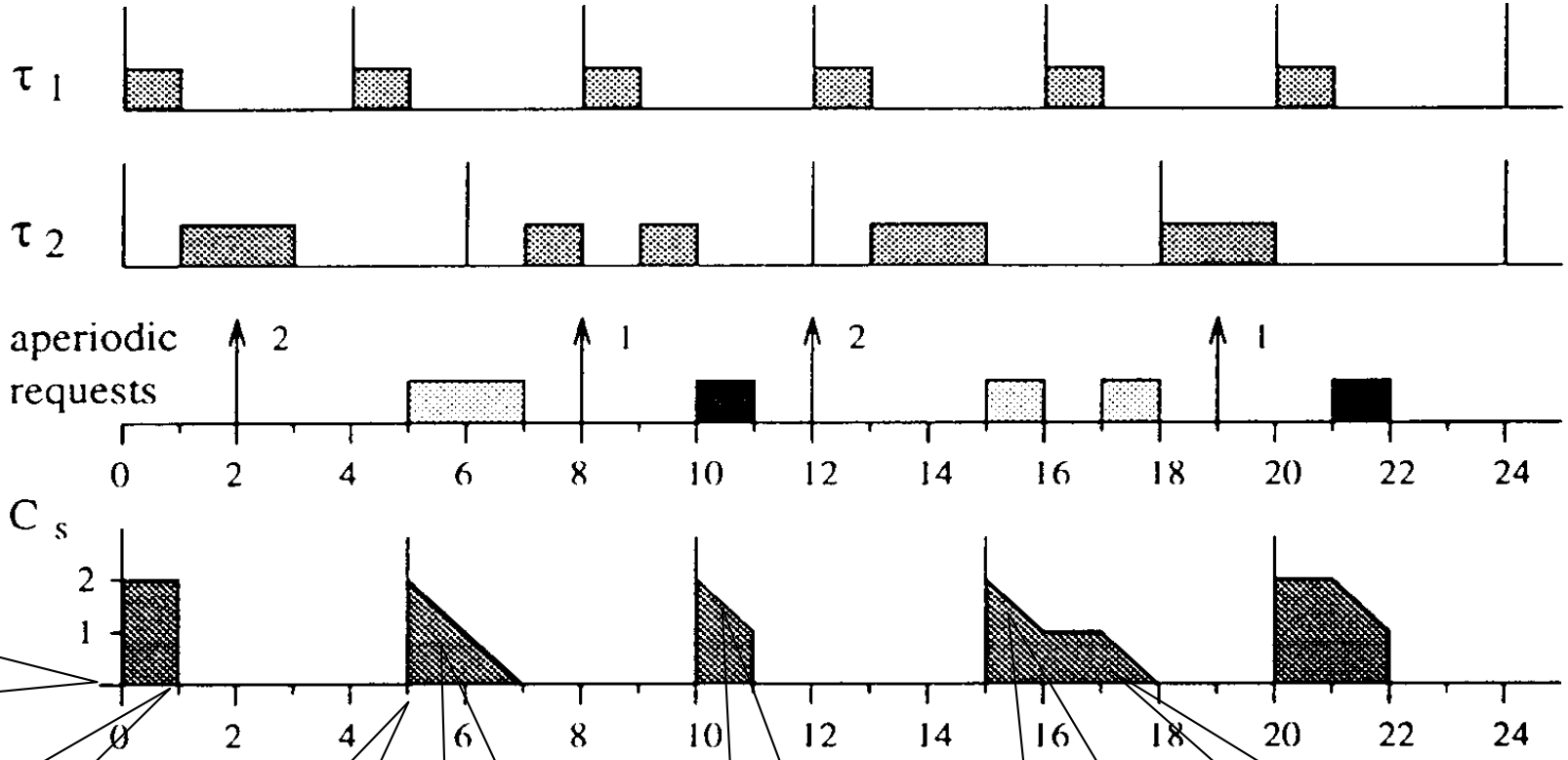
Example:

	C_i	T_i
τ_1	1	4
τ_2	2	6

Server

$C_s = 2$

$T_s = 5$



Server capacity charged to 2

Time for server to run, but no pending aperiodic.Capacity lost

Capacity recharged to 2

Ap1 pending, C_s>0, ap1 consumes both server ticks

Ap2 pending, it consumes only one server tick

Ap3 consumes one server tick, and then gets preempted by τ_1

Ap3 continues and consumes one more tick

Polling Server

Properties

- When used together with Rate Monotonic, aperiodic tasks execute at the highest priority if $\mathbf{T_s = min(T_1, \ldots, T_n)}$

- In the worst-case, the PS uses all assigned capacity, hence it behaves as a periodic task with utilization:

$$U_s = C_s/T_s$$

- Under RM, a periodic task set extended by PS is schedulable if:

$$U_P + U_s \leq (n+1)(2^{1/(n+1)} - 1)$$

(that is, regular RM test extended with PS as a periodic task)

- Note: **several servers** can be created, e.g., one for urgent aperiodic tasks, and one for low priority aperiodic tasks

Polling server

***Firm* aperiodic tasks can be online guaranteed:**

If $C_a \leq C_s$, it will in worst case be completed in two server periods

$$2T_s \leq D_a$$

For arbitrary computation time C_a:

$$T_s + \left\lceil \frac{C_a}{C_s} \right\rceil T_s \leq D_a$$

Deferrable Server – DS

Under the Polling Server, aperiodic requests have to wait

- before server start: until server becomes active
- after server start: until next server instance becomes active

A server does not execute if no aperiodics are present at its start

- Consequently, an aperiodic request that occur *immediately* after server start, has to wait until next server instance, even if it could be served right away
- Server capacity is lost in this case
- **Simple, but not smart...**
- Better to preserve capacity if not needed for later...

Deferrable Server

- ***Preserves* its capacity** if no pending aperiodic requests
- Capacity is **maintained** until the end of server period

Deferrable Server

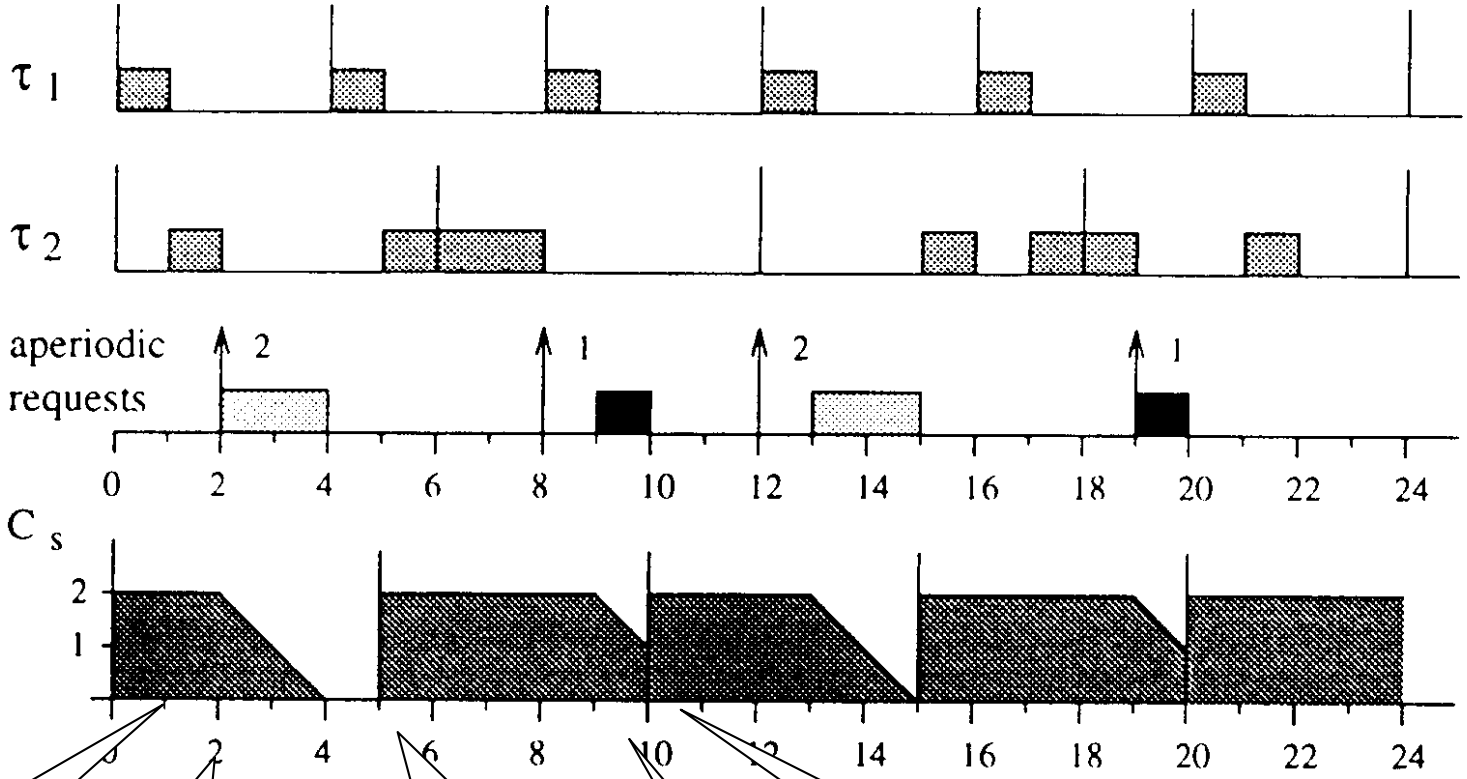
Example (same tasks as in PS)

	C_i	T_i
τ_1	1	4
τ_2	2	6

Server

$C_s = 2$

$T_s = 5$



No pending aperiodics. The capacity is preserved for later

Ap1 arrival. It can be serviced right away (from preserved capacity)

No pending aperiodics. The capacity is preserved for later

Cap. kept until needed by ap2

Capacity replenished at the beginning of each new period

Deferrable Server

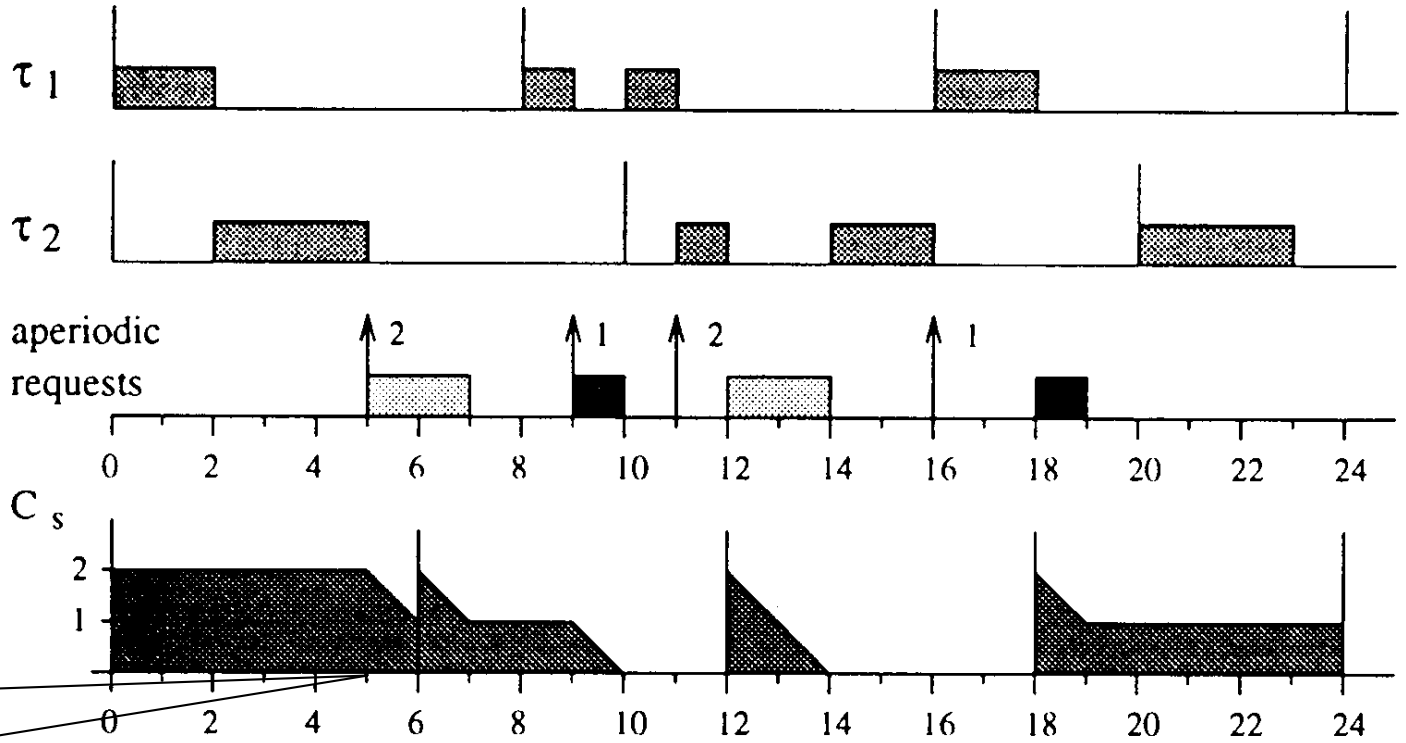
Example DS as highest priority

	C_i	T_i
τ_1	2	8
τ_2	3	10

Server

$C_s = 2$

$T_s = 6$

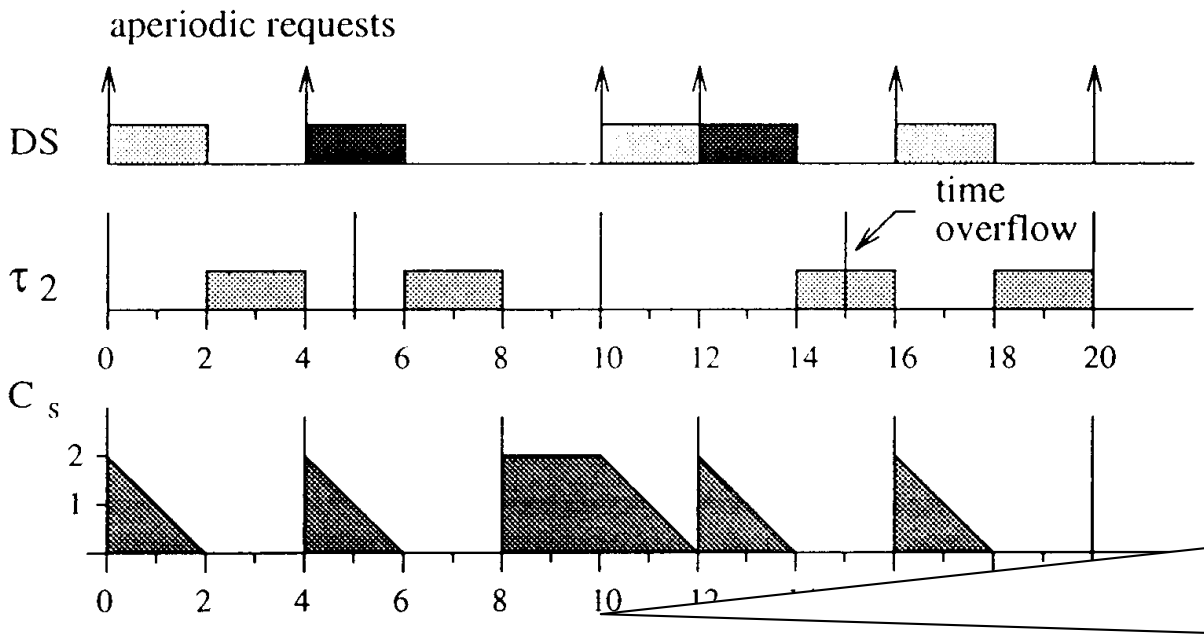


Server execution deferred until t=5

Deferrable Server

Schedulability analysis

- RM schedulability analysis assumes that the highest priority task must execute when it is ready to run
- DS violates this assumption!
- On the previous slide, at time t=0, the server is the highest priority task, but it defers its execution until t=5
- This can lead to possible deadline violation for lower priority tasks:



At t=10, τ_2 must wait for DS to serve ap3. If PS is used, this would not happen since in PS the capacity would get lost at t=9

Deferrable Server

Deferred execution improves responsiveness of aperiodic requests, but decreases the utilization bound.

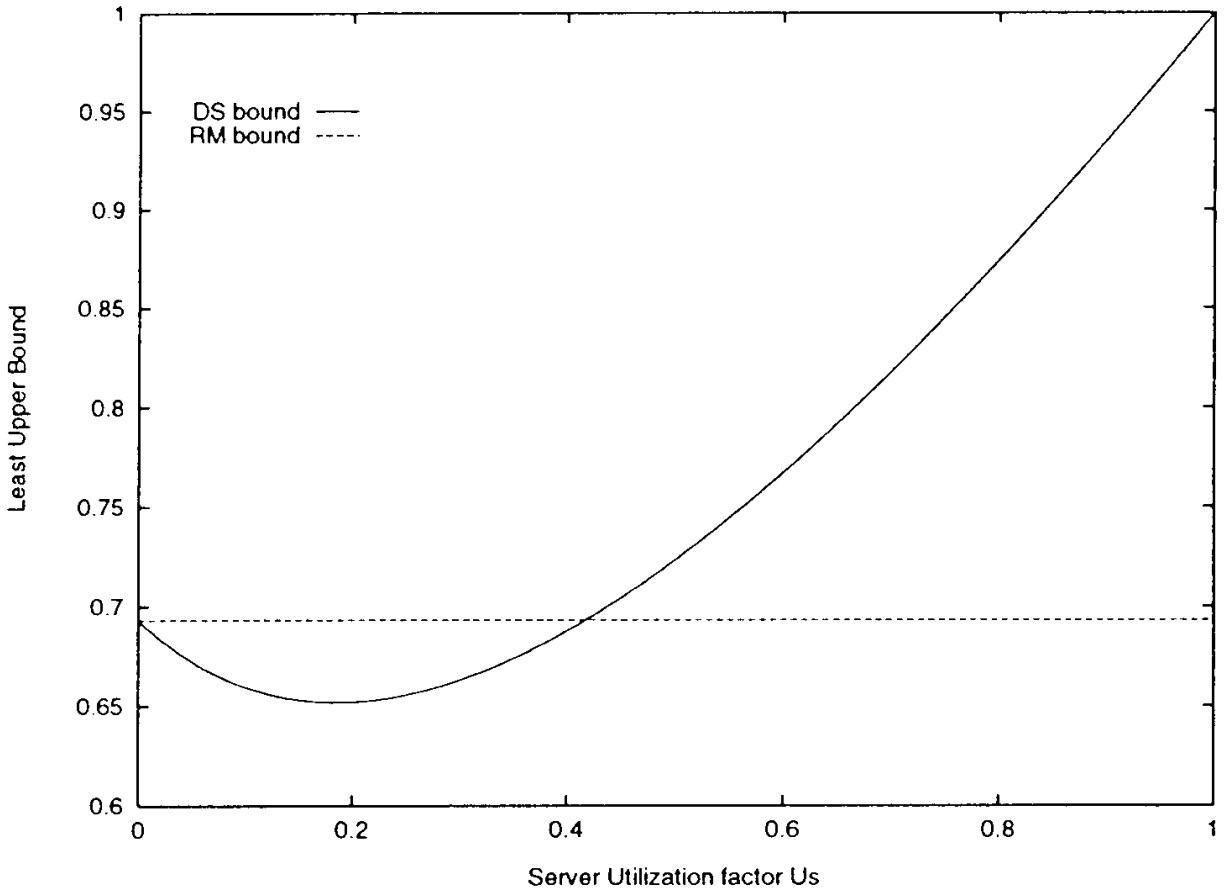
A task set is schedulable by DS if:

$$U_p \leq n\left[\left(\frac{U_s+2}{2U_s+1}\right)^{1/n}-1\right]$$

When n→ ∞

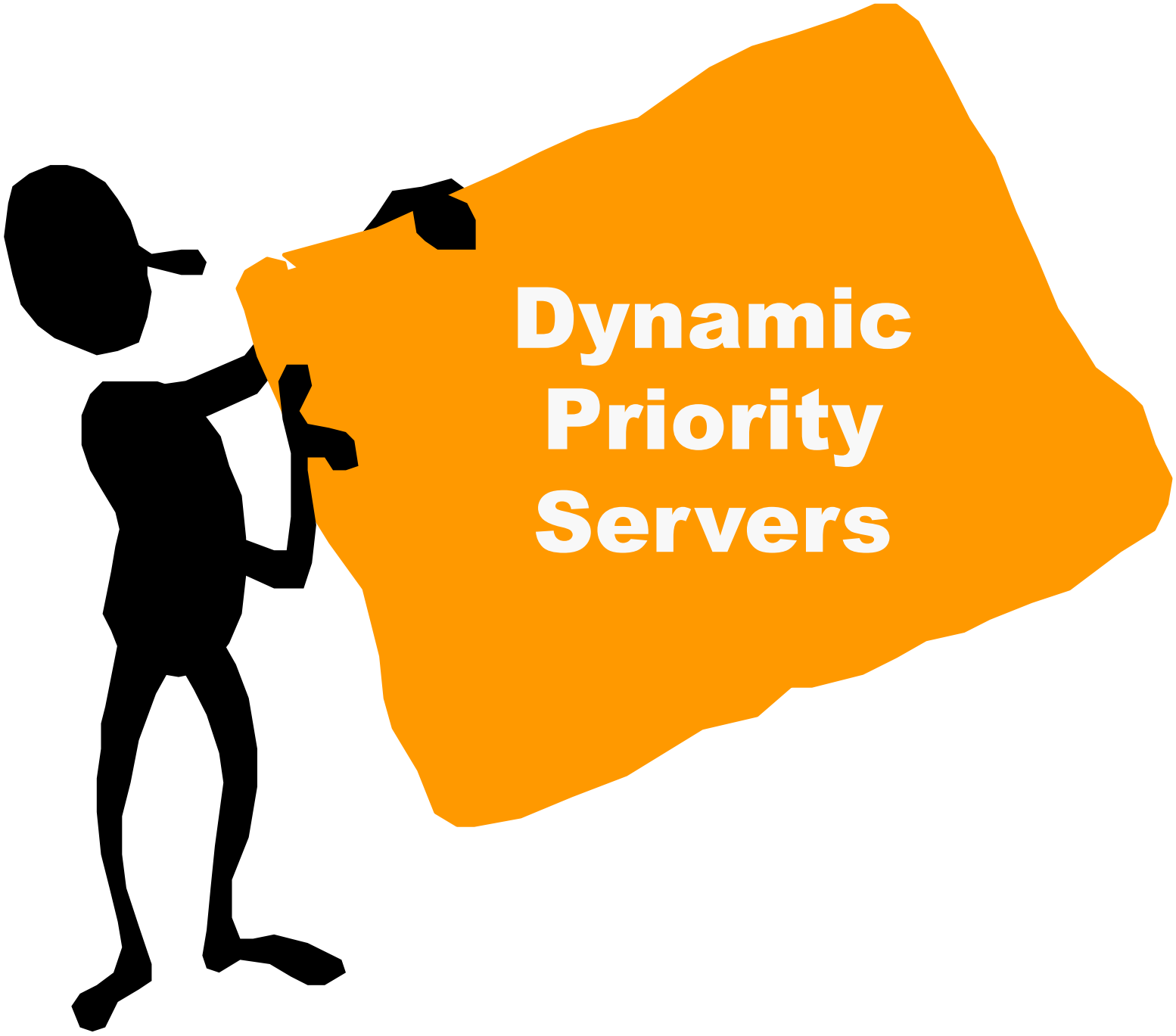
$$U_p \leq \ln\left(\frac{U_s+2}{2U_s+1}\right)$$

Deferrable Server



$U_s < 0.4$ → the presence of DS worsens the RM bound

$U_s > 0.4$ → the RM bound is improved


Dynamic
Priority
Servers

Dynamic priority servers

So far, we have looked into static priorities servers based on RM...

There are also *dynamic* priority servers, used along with EDF

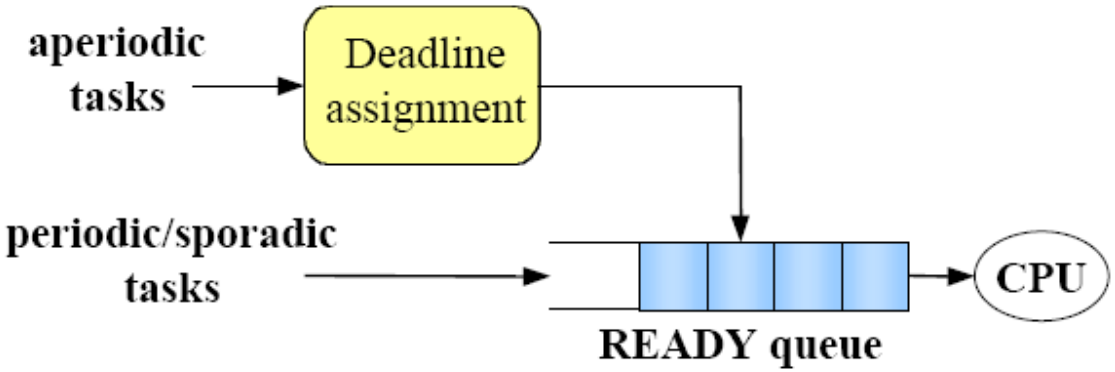
Compared to static priority servers

- Higher schedulability bounds ⇒better processor utilisation
- Increased size of aperiodic servers
- Better aperiodic responsiveness.

Total Bandwidth Server – TBS

TBS mechanism

- Each aperiodic request is assigned *an internal deadline* so that the server demand does not exceed a given bandwidth U_s
- Aperiodic tasks are inserted in the ready queue and scheduled together with periodic tasks.



- Necessary and sufficient schedulability test:

$$U_P + U_S \leq 1$$

Total Bandwidth Server

Deadline assignment rule

- Each aperiodic request that enters the system gets a unique deadline
- If the aperiodic task A_k arrives at time a_k it receives a deadline

$$d_k = a_k + \frac{C_k}{U_s}$$

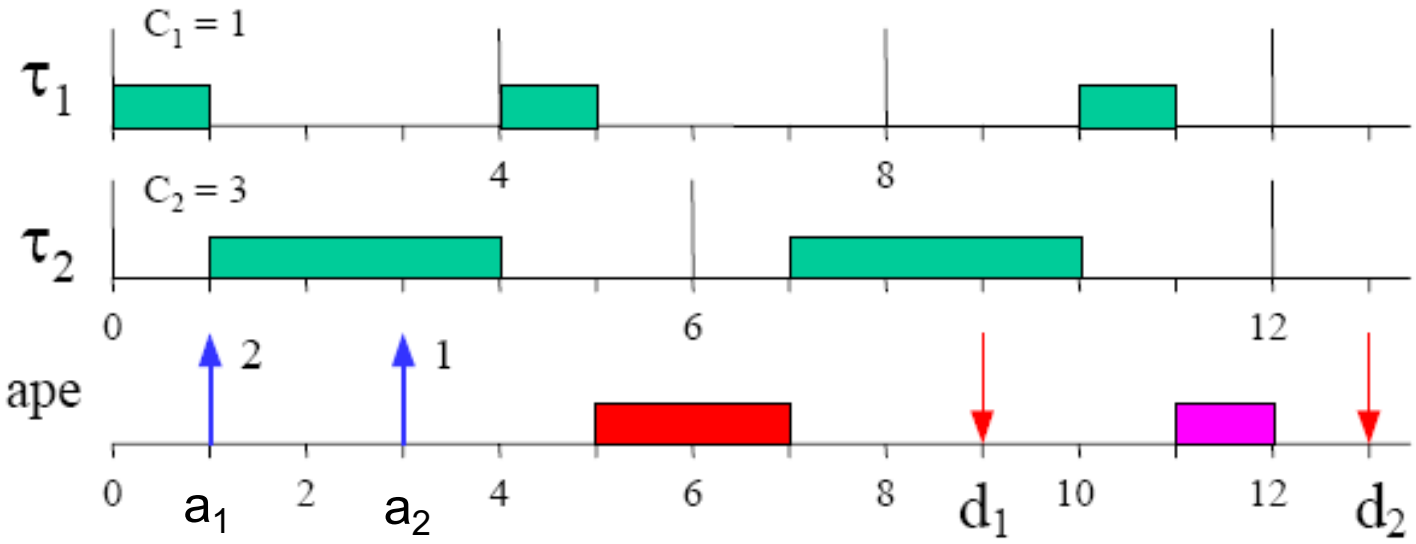
- To keep track of the capacity assigned to previous aperiodic tasks, d_k must be computed as:

$$d_k = \max(a_k, d_{k-1}) + \frac{C_k}{U_s}$$

- By definition $d_0=0$

Total Bandwidth Server

Example:



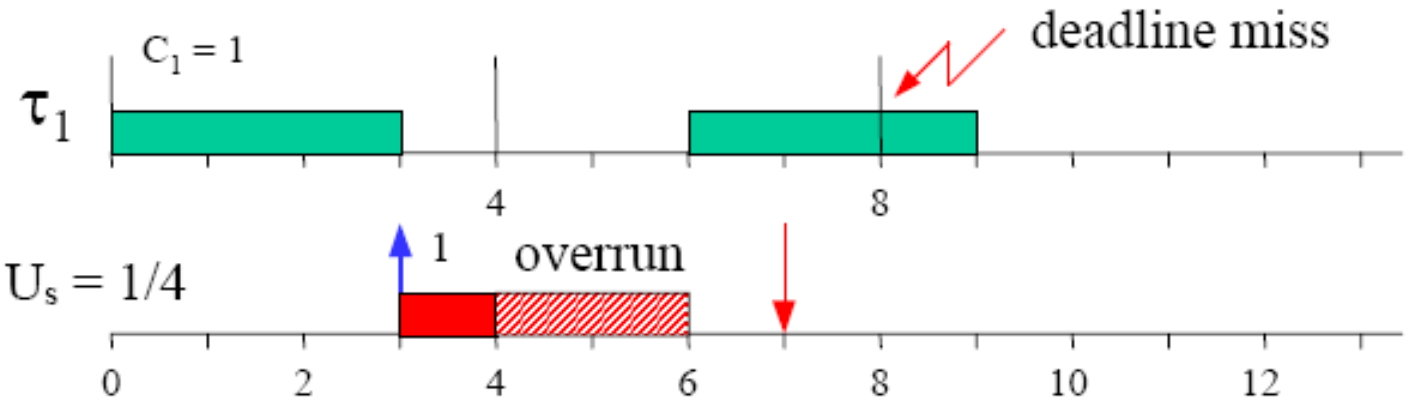
$$U_s = 1 - U_p = 1/4$$

$$\begin{cases} d_1 = a_1 + C_1 / U_s = 1 + 2 \cdot 4 = 9 \\ d_2 = \max(a_2, d_1) + C_2 / U_s = 9 + 1 \cdot 4 = 13 \end{cases}$$

Total Bandwidth Server

Problems with TBS

- Without a budget management, there is no protection against execution overruns
- If a task executes more than expected, hard tasks could miss their deadlines



Solution: task isolation

- In the presence of overruns, only the faulty task should be delayed.
- Each task should not demand more than its declared utilization
- If a task executes more than expected, its priority should be decreased (i.e., its deadline postponed).

Constant Bandwidth Server

It assigns deadlines to tasks as the TBS, but keeps track of aperiodic task executions through a budget mechanism.

When the budget is exhausted

- it is immediately replenished
- the deadline is postponed to keep the demand constant

CBS parameters

- Given by the user:

  Q_s = Maximum capacity (budget)

  T_s = Server period

  $U_s = Q_s / T_s$, Server bandwidth

- Maintained by the server:

  c_s – Current capacity, initialized to Q_s

  d_s – Server deadline, initialized to T_s

Constant Bandwidth Server

Basic CBS rules

- Arrival of aperiodic task A_k → assign d_s

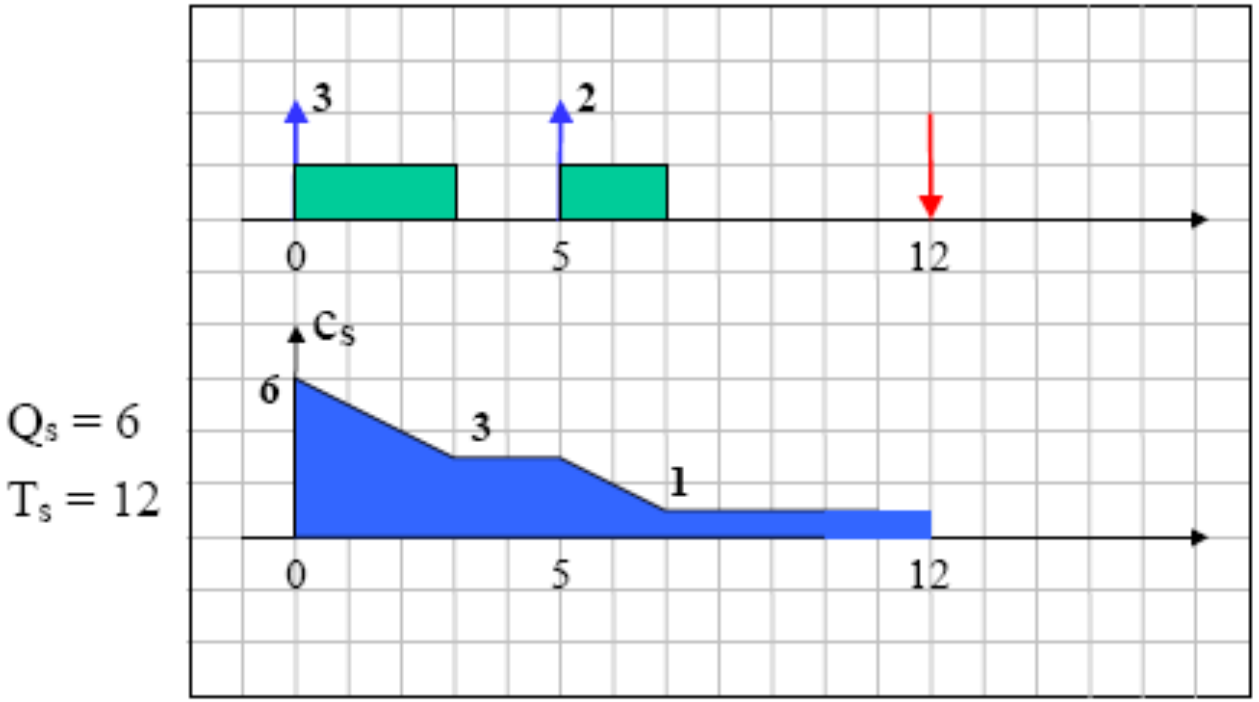
  if ($a_k + c_s/U_s \leq d_s$)
  recycle d_s
  else
  $d_s = a_k + T_s$
  $c_s = Q_s$

- Server capacity exhausted → postpone d_s

  $d_s = d_s + T_s$
  $c_s = Q_s$

Constant Bandwidth Server

Example deadline assignment



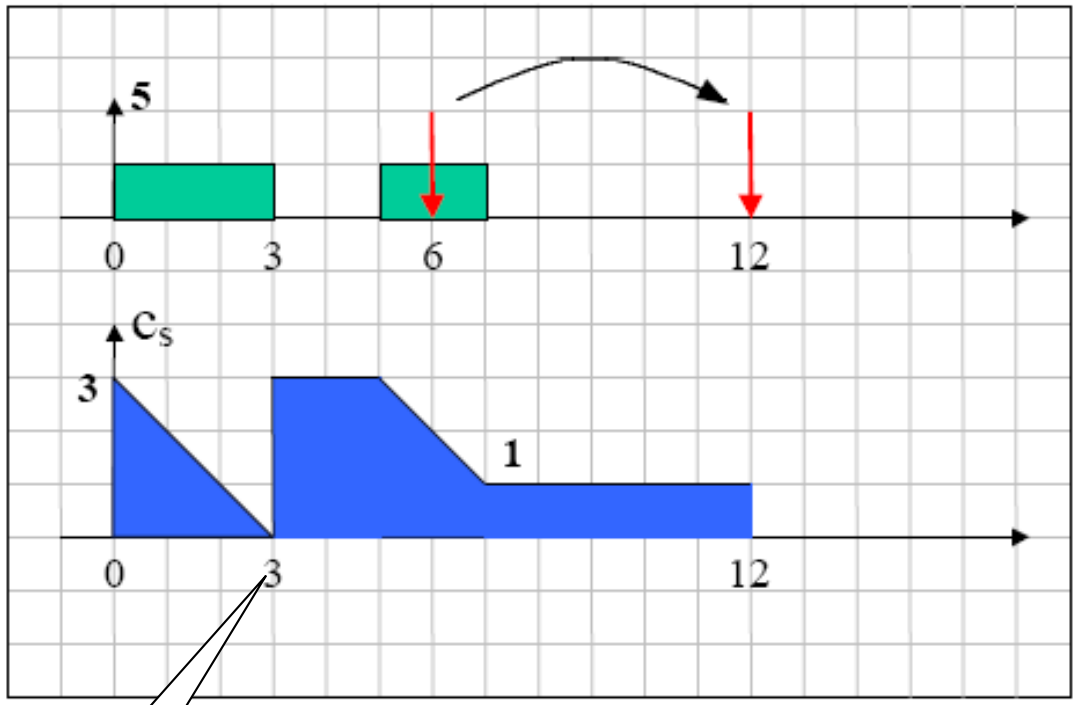
Aperiodic 1: $d_S = a_1 + T_S = 0 + 12 = 12$

Aperiodic 2: $a_2 + \frac{c_S}{U_S} = 5 + \frac{3}{6/12} = 5 + 6 = 11 \leq d_S \Rightarrow d_S = 12$

(recycled)

Constant Bandwidth Server

Example budget exhausted



$Q_s = 3$

$T_s = 6$

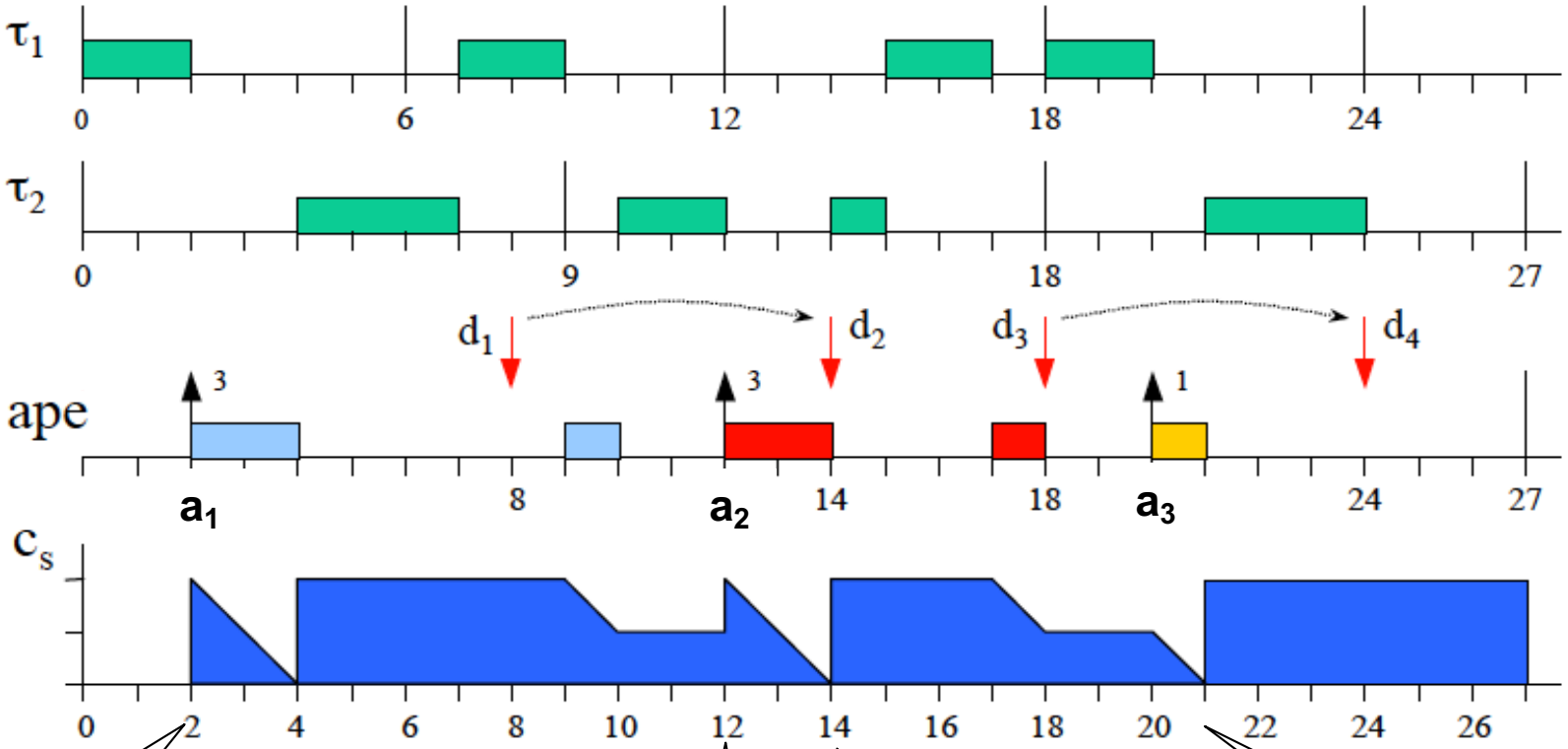
At t=3, c_S is all used. The deadline is postponed from 6 to 12, and the budget is replenished

$$t = 0 : d_S = a_1 + T_S = 0 + 6 = 6$$

$$t = 3 : d_S = d_S + T_S = 6 + 6 = 12$$

$$c_S = Q_S = 3$$

Constant Bandwidth Server – example schedule



$c_s=Q_s=2$
$d_1=a_1+T_s=8$

Capacity exhaused:
$c_s=Q_s=2$
$d_2=d_1+T_s=14$

$c_s=Q_s=2$
$d_3=a_2+T_s=18$

Capacity exhaused.
$c_s=Q_s=2$
$d_4=d_3+T_s=24$

20 + 1(2/6)
≤ 24
True
Reuse d_4

Capacity exhaused:
$c_s=Q_s=2$
$d_5=d_4+T_s=30$

Constant Bandwidth Server

Properties

- *Bandwidth Isolation* – If a task τ_i is served by a CBS with bandwidth U_s then, in any interval Δt, τ_i will never demand more than $U_s \Delta t$

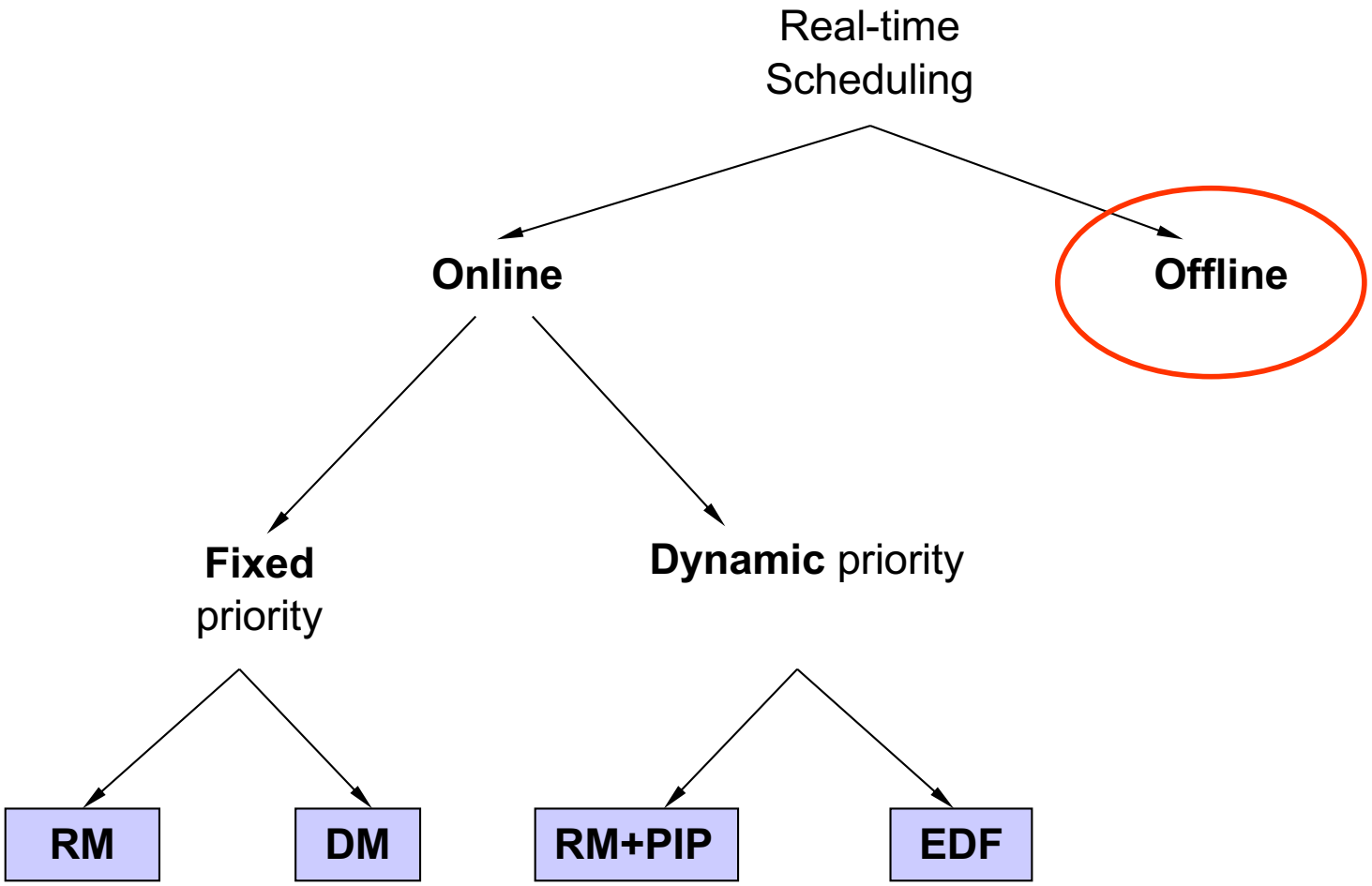
- *Hard schedulability* - a hard task τ_i (C_i, T_i) is schedulable by a CBS with $Q_s = C_i$ and $T_s = T_i$, if and only if τ_i is schedulable by EDF

Remarks

- CBS can be used as a **safe server** for handling aperiodic tasks under EDF
- It can be used as a **bandwidth reservation mechanism** to achieve task isolation
- It allows to **guarantee a minimum performance** to soft aperiodic tasks, based on the assigned bandwidth.

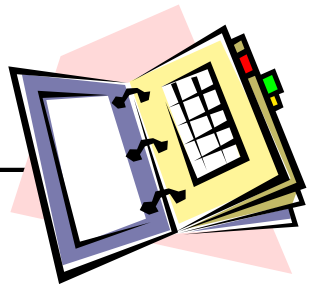

Offline
Scheduling

A simple classification of real-time scheduling



RM	Rate Monotonic
DM	Deadline Monotonic
EDF	Earliest Deadline First
PIP	Priority Inheritance Protocol

Offline scheduling



Also known as *static* or *pre-run-time* scheduling

- It has been used for 40 years in military systems, navigation, and monitoring systems…

Properties (compared to online scheduling)

(+) Allows more complex task models

(+) Can solve more difficult scheduling problems

(+) More predictable (proof by construction)

(+) Jitter can be minimized (even avoided if fully offline scheduling is used)

(−) Less flexible and expandable

(−) Difficult to handle aperiodic activities

(−) Not supported by operating systems

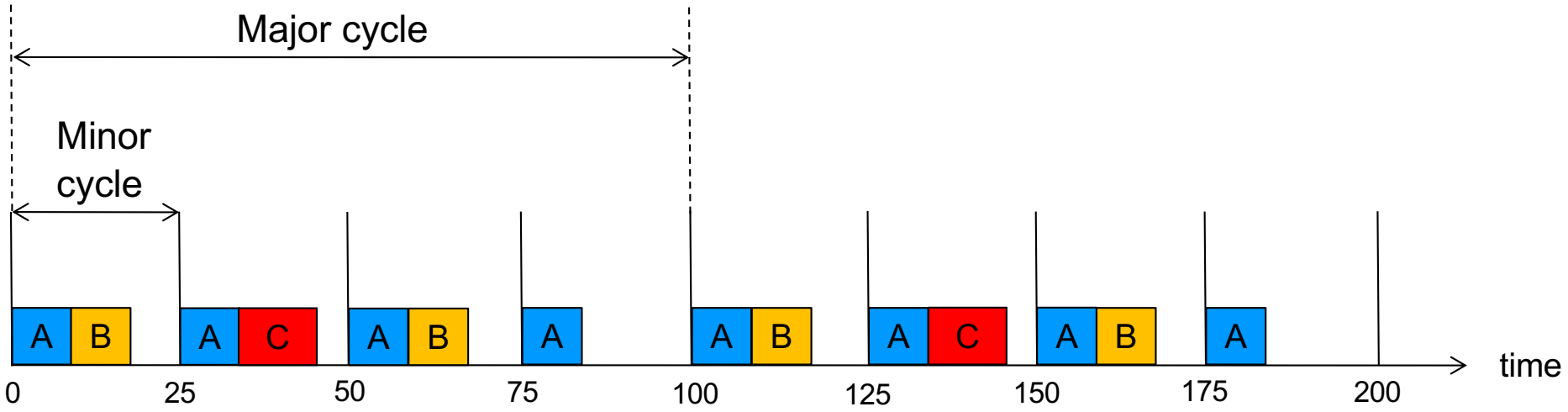
Scheduling approach

- The time axis is divided in intervals of equal length (time slots).
- Each task is statically allocated in a slot
- The execution in each slot is **activated by a timer**
- Repeating cycles identified

Offline scheduling

Example

Task	Period time
A	25
B	50
C	100



Offline schedule creation

Each task is usually described with three temporal parameters:

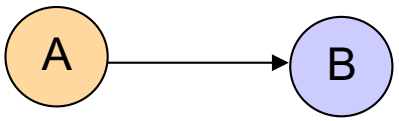
C– execution time
D – deadline
RT – release time

Some additional relations:

- Precedence order between task
- Period for precedence graphs

Precedence

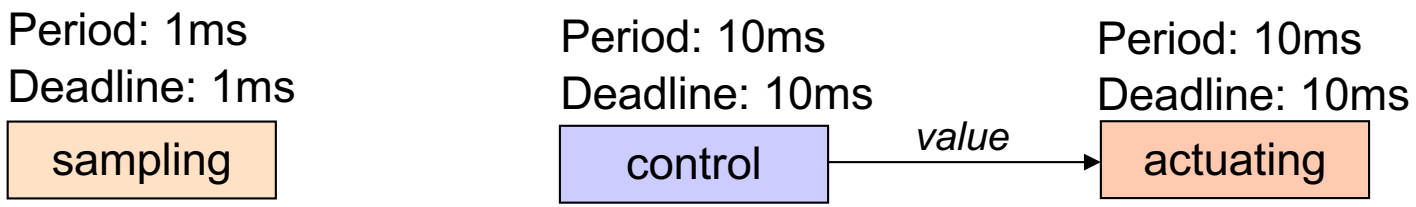
- If **A *precedes* B** then both tasks run with the same period, but A must complete before B starts to execute.



Offline schedule creation

Example precedence relation:

Assume a small control system which needs to sample the environment with the rate of 1000 Hz and control and actuate at 100 Hz

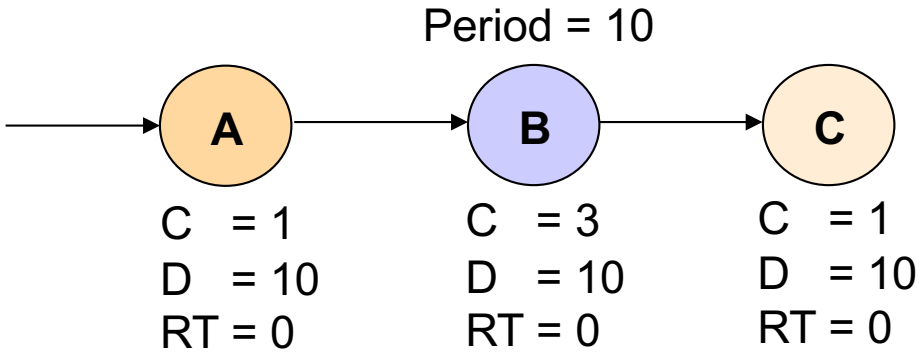


We specify a precedence relation between the control task and the actuator task.

Based on this, the offline scheduler will schedule the control and the actuator task **in the same minor cycle**, and it will always **execute the control task before the actuator task**, so that the control task always sends the latest calculated value to the actuator task.

Offline schedule creation

Example of a **precedence graph** with several tasks and their timing constraints



A real-time system will usually contain *several* precedence graphs

Scheduling goal:

- to find a *common* offline schedule that satisfies the requirements of *all* precedence graphs in the systems.

Offline schedule creation

Example scheduling approach

1. Construct a ***joint graph*** of all individual precedence graphs. The period of the joint graph is equal to the ***least common multiple*** of all individual precedence graph's periods.

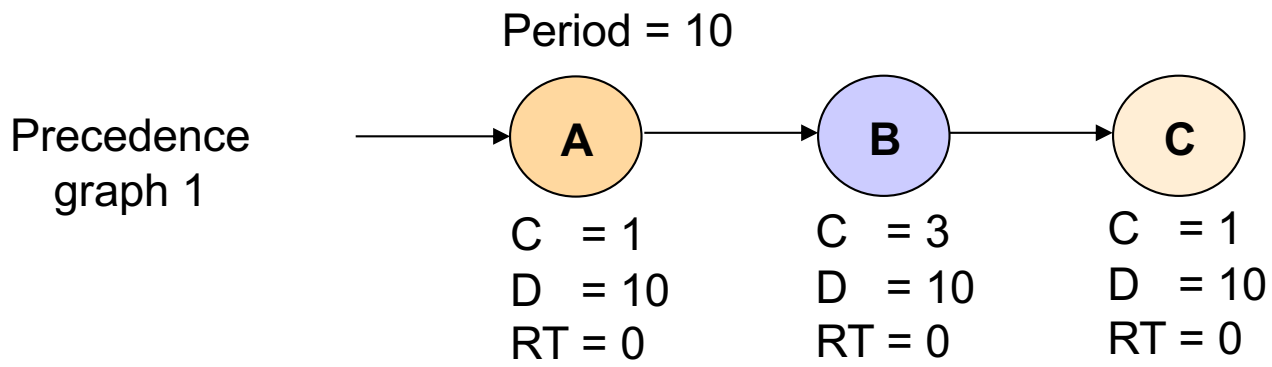
2. **Generate a search tree** that contains all possible paths through the joint graph (i.e., both valid and invalid schedules)

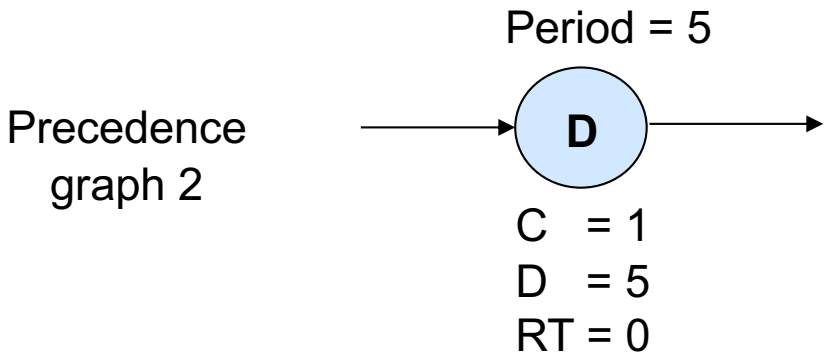
3. **Traverse the search tree** find all solutions (valid schedules). Use some heuristic function when selecting a schedule among all feasible solutions.

Offline scheduling creation

Example:

- Create an offline schedule for the following precedence graphs:

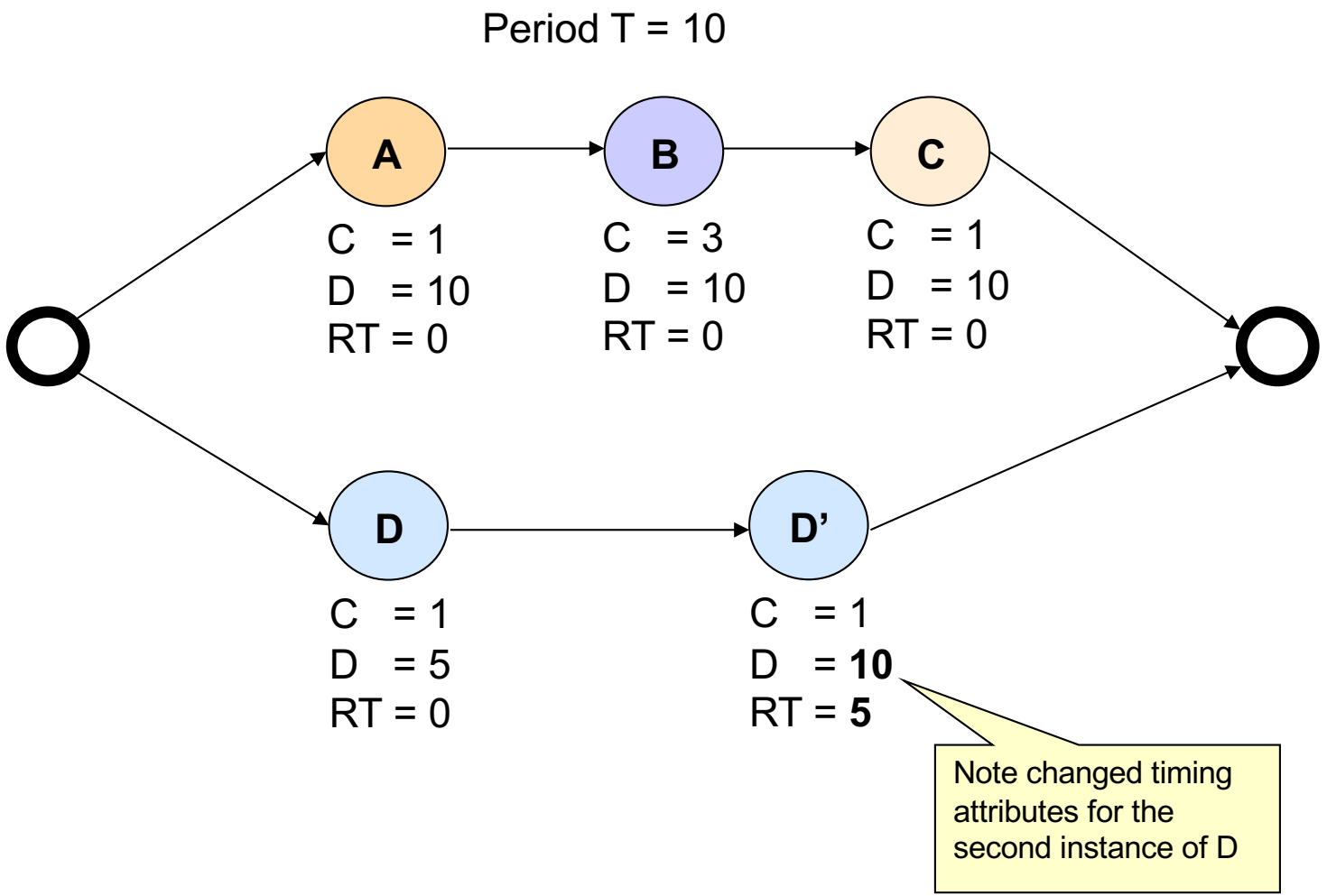




Example

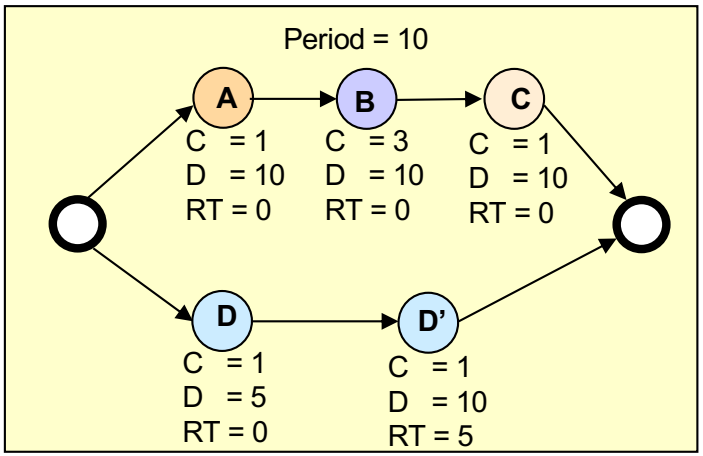
Step 1: joint graph

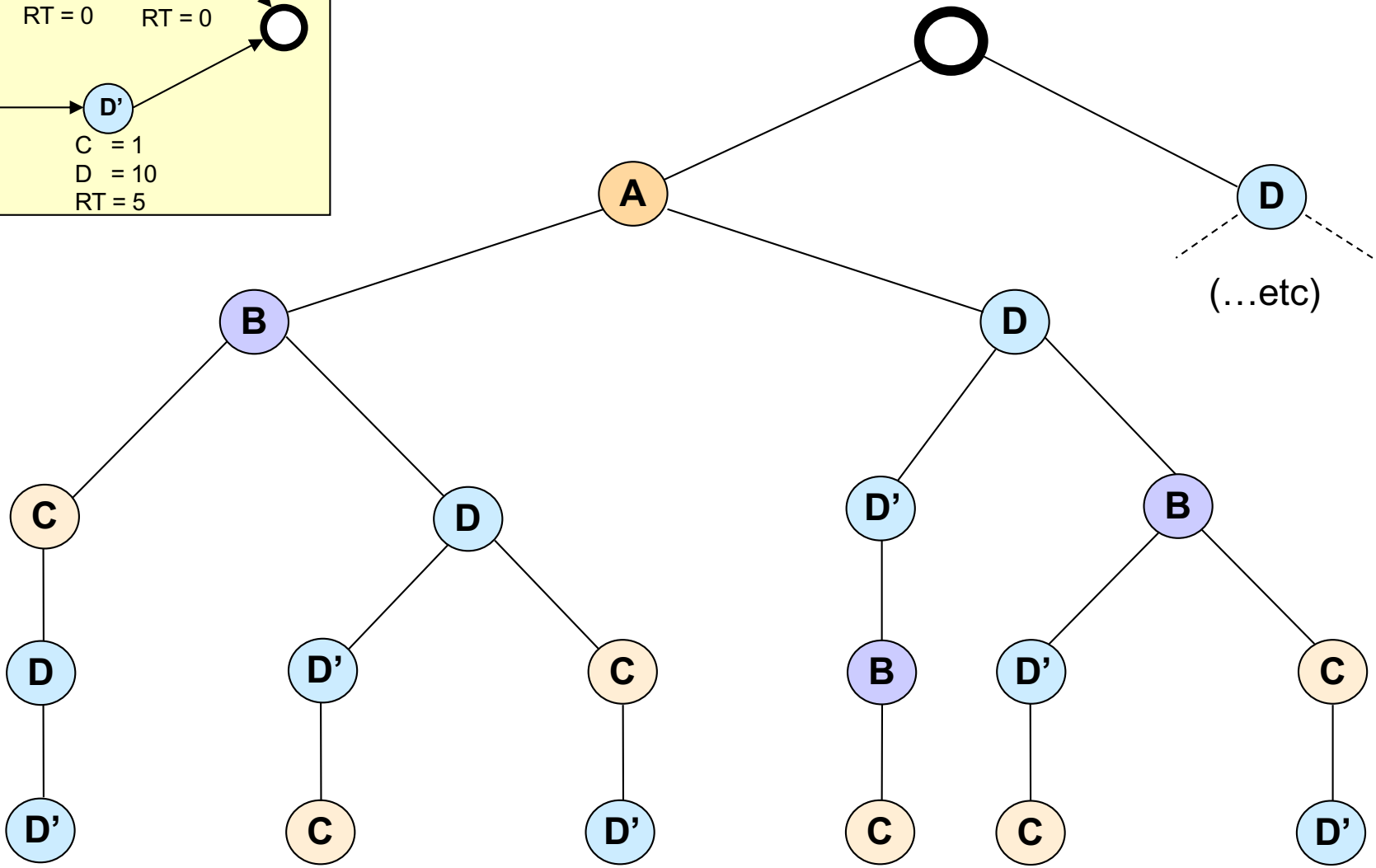
LCM (10, 5) = 10



Example

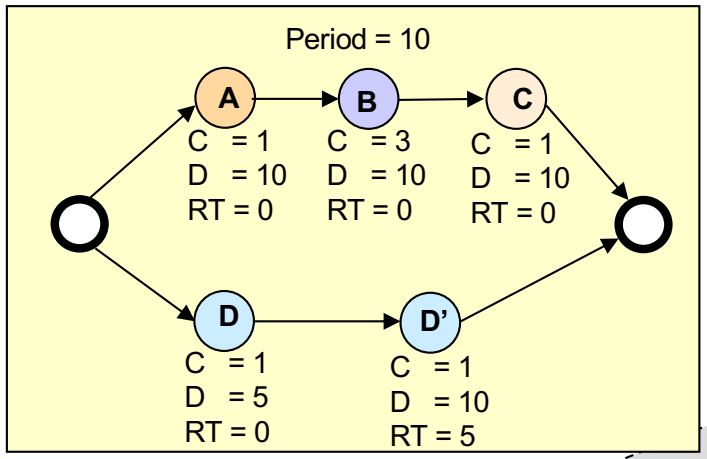
Step 2: generate search tree

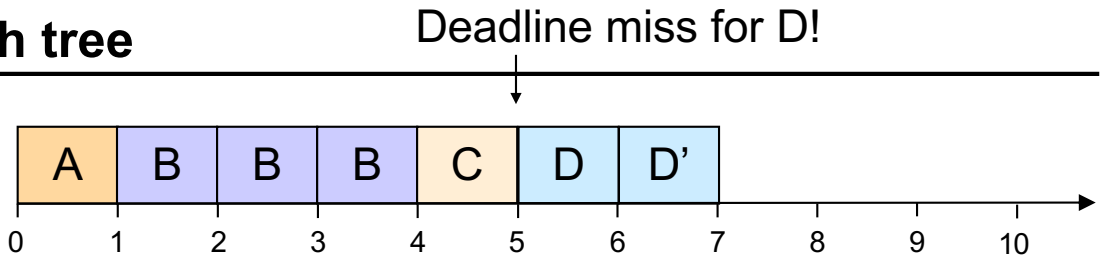




Example

Step 3: traverse the search tree

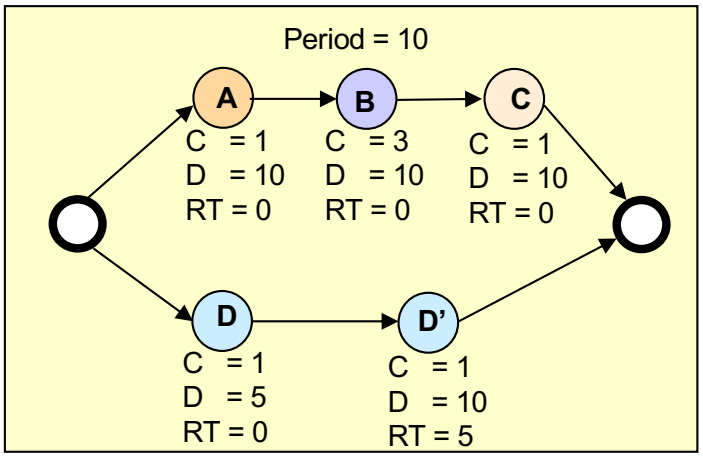


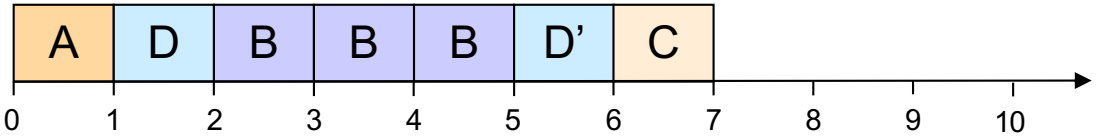


Not a Feasible schedule

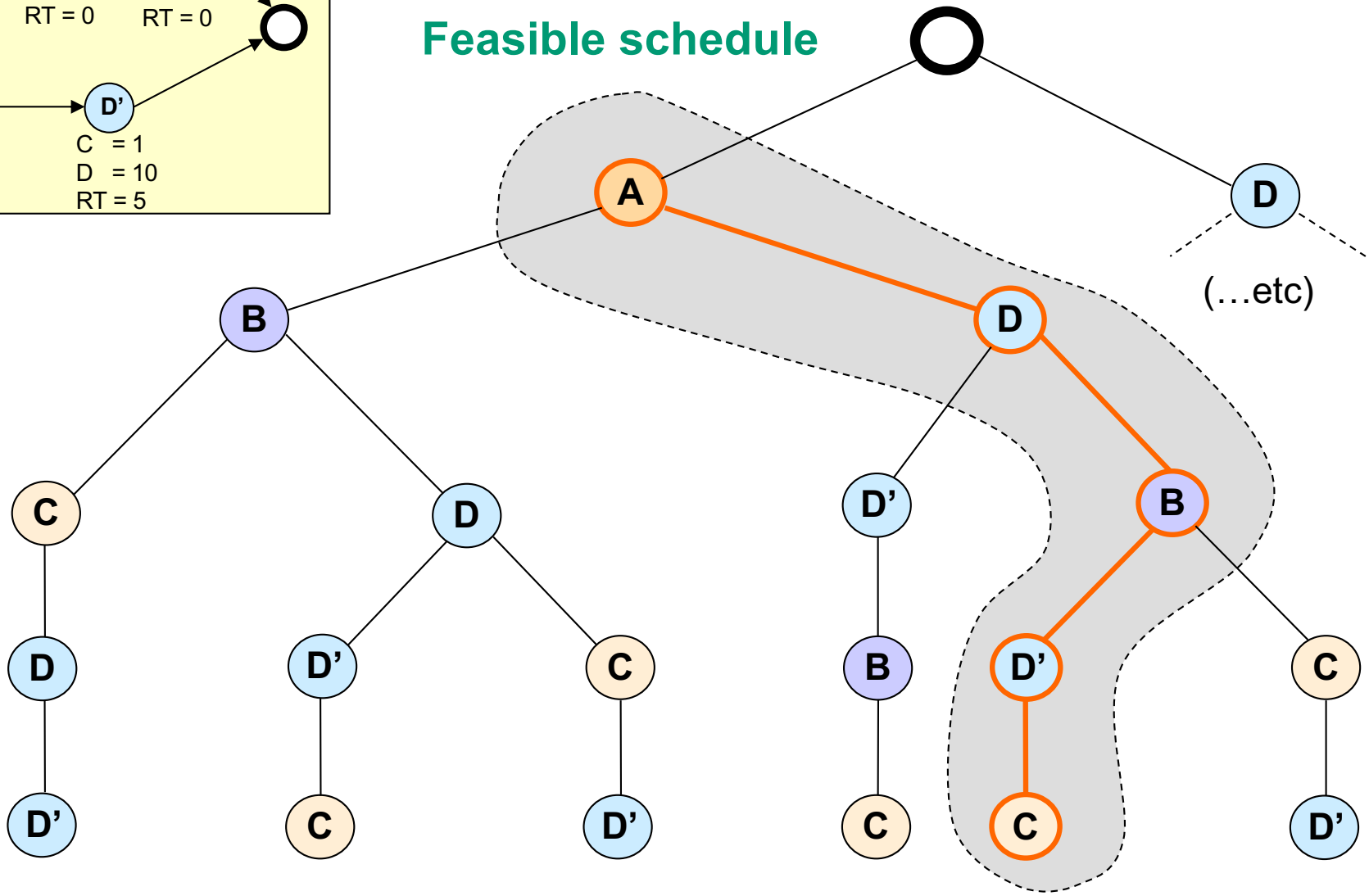
Example

Step 3: traverse the search tree





Feasible schedule



Heuristic functions

The size of the search tree increases exponentially for each new task we add to the system → it can take very long time to find a solution

If we allow preemption

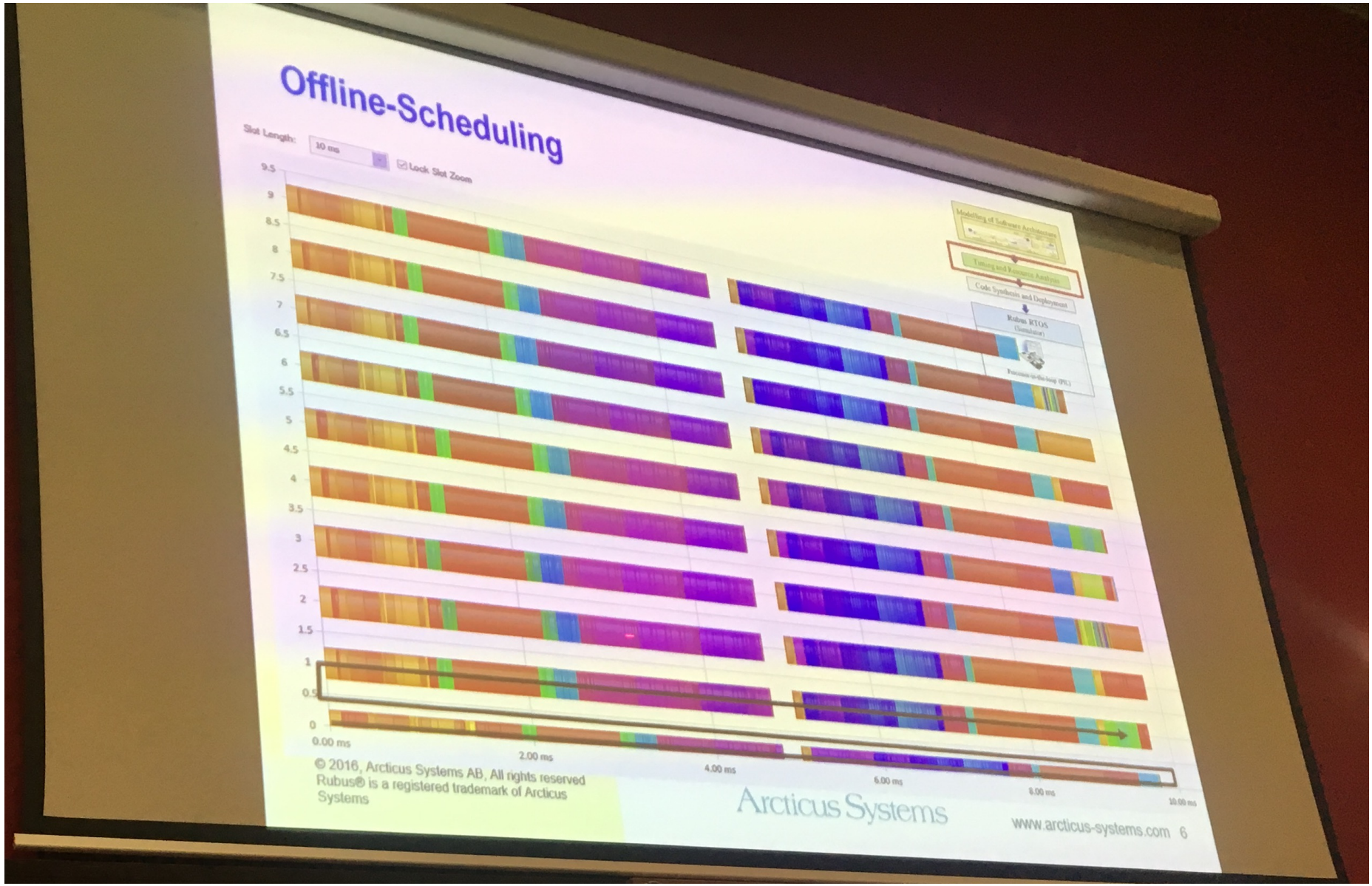
(+) higher probability that we'll find a solution

(-) bigger search tree

We must use some strategy – heuristic

- Ex 1: Chose the task with the shortest deadline
- Ex 2: Chose the task with the longest execution time

Offline Schedule: a practical example from Automotive Industry



Offline Schedule: a practical example from Automotive Industry

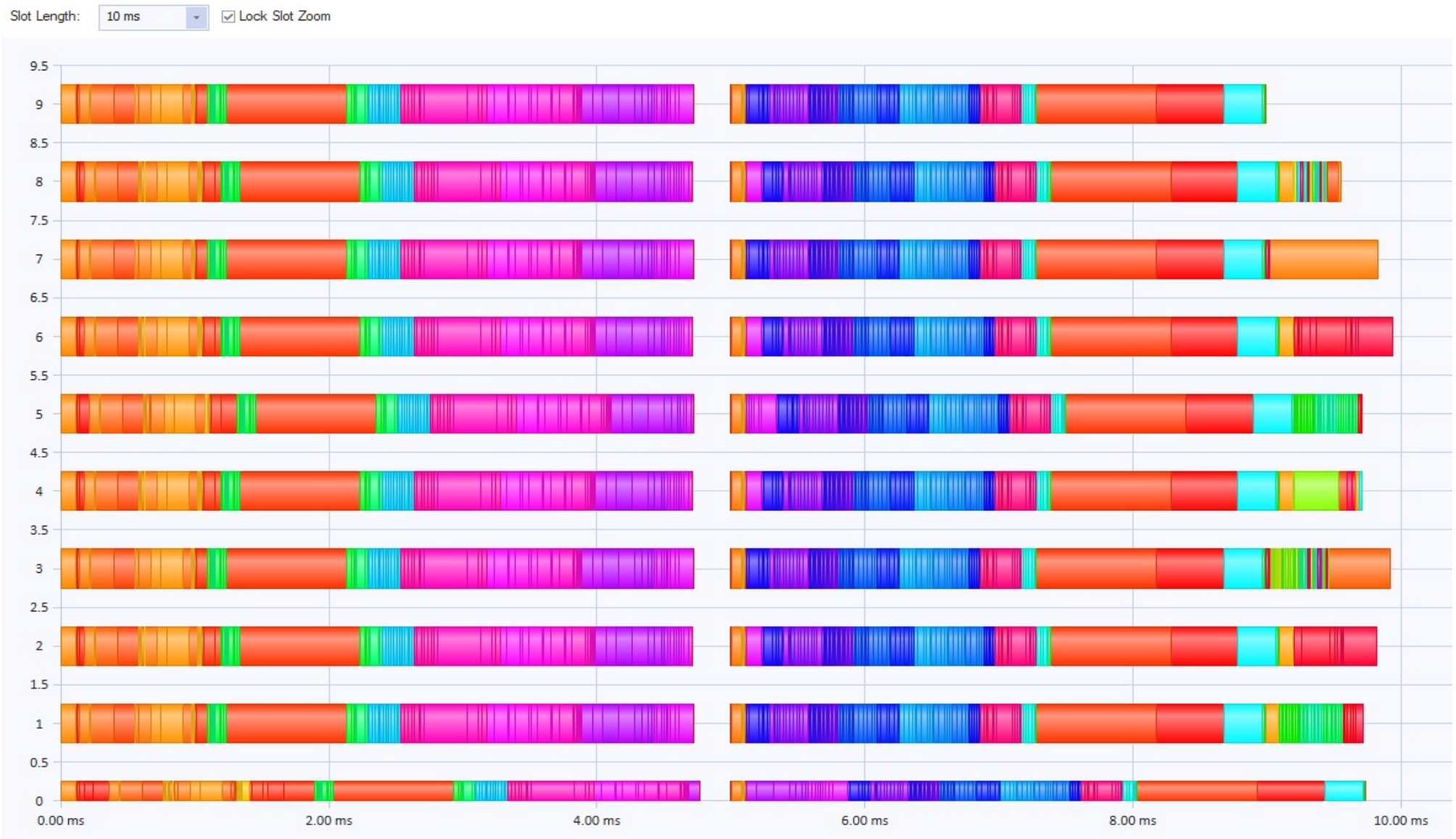


Figure courtesy of Arcticus Systems AB

Introducing flexibility to offline schedules

We mentioned before that offline scheduling is not appropriate for handling aperiodic tasks

We will now look into **slot shifting** method that allows for flexible execution of aperiodic and sporadic tasks **on top of offline schedules**.

It **combines offline and online scheduling**, while providing real-time guarantees for online tasks with hard deadlines.

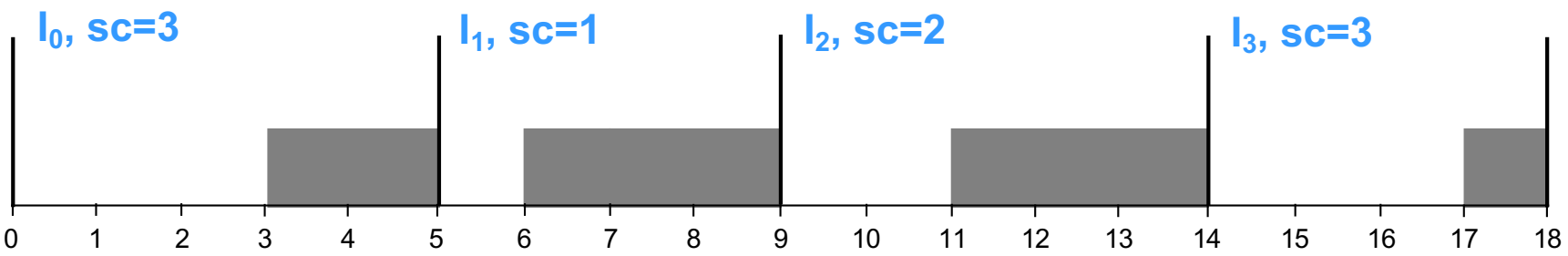
Slot shifting

Basic idea

- Combined offline and online scheduling
- Safety-critical tasks are scheduled **offline** as time-triggered tasks
- Dynamically arriving tasks (aperiodic and sporadic) are scheduled **online**, on top of the offline schedule

Offline schedules are not tight, there are resources left for online tasks

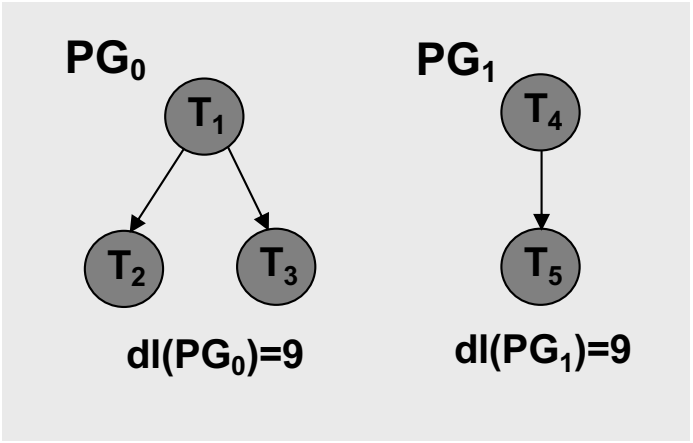
- Offline part: Amount and distribution of available resources is calculated – **spare capacity (sc)**
- Online part: **sc** is used to guarantee and execute online tasks



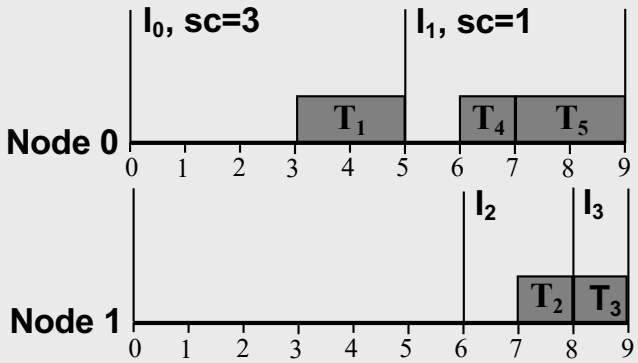
Slot Shifting

offline

Tasks with constraints

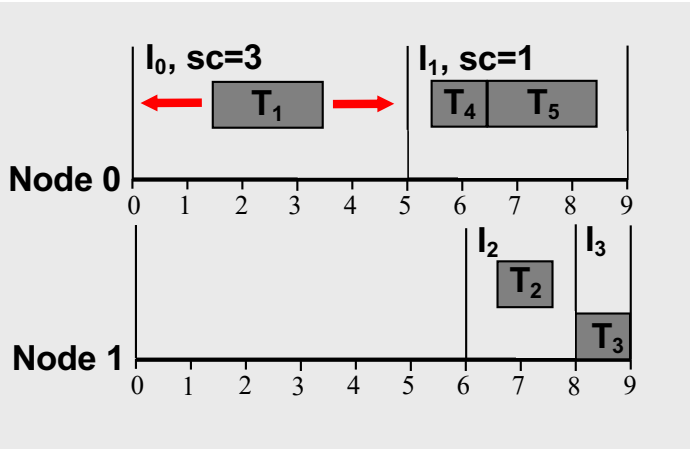


Offline schedule: intervals and spare capacities

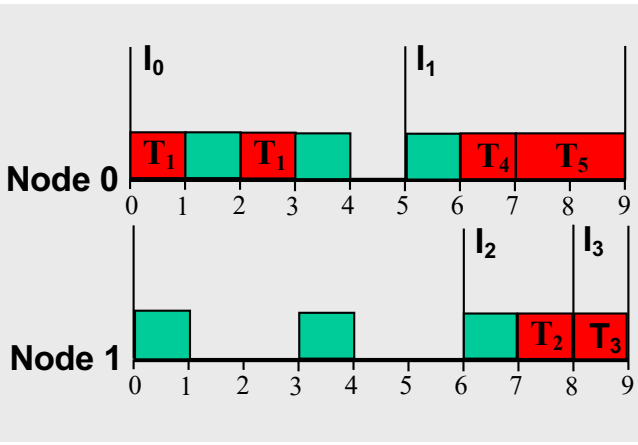


online

offline scheduled tasks (periodic) can flexibly execute within their intervals



Online tasks (aperiodic and sporadic) are scheduled on top of the **offline** schedule



Calculation of spare capacities

offline

Spare capacity of an interval is calculated as:

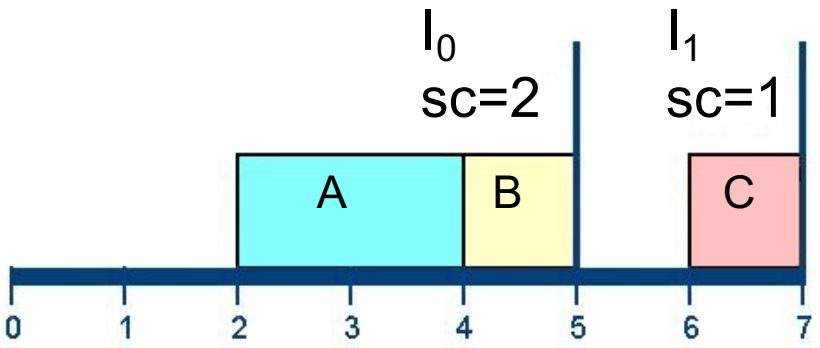
$$sc(I_i) = |I_i| - \sum_{\tau_j \in I_i} C_j + \min(sc(I_{i+1}),0)$$

Total execution time of all tasks that belong to the interval

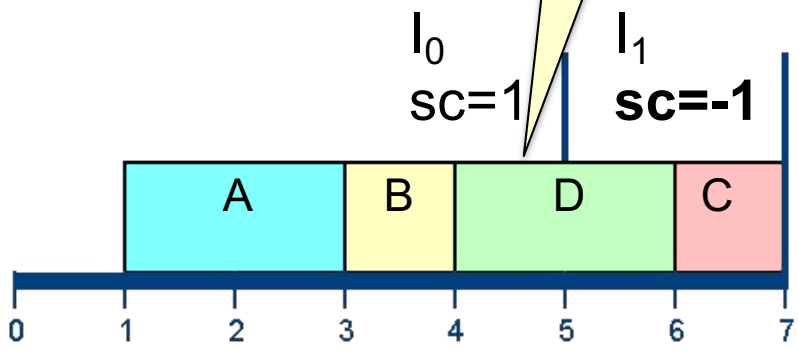
Tasks that belong to the next interval but execute within I_i

Task D belongs to I_1 but it executes in I_0, i.e., I_1 "borrows" the spare capacity from I_0

Ex1:



Ex2:



Acceptance test for aperiodic tasks

- Based on standard EDF
- Sets EDF to work on top of the offline schedule
- Finishing time of an aperiodic task τ_i:

$$ft_i = ft_{i-1} + C_i + \text{offline\_reserved}[ft_{i-1}, ft_i]$$