Real Time Scheduling

Maryline CHETTO

Master Temps Réel – Systèmes Embarqués (CORO ERTS)

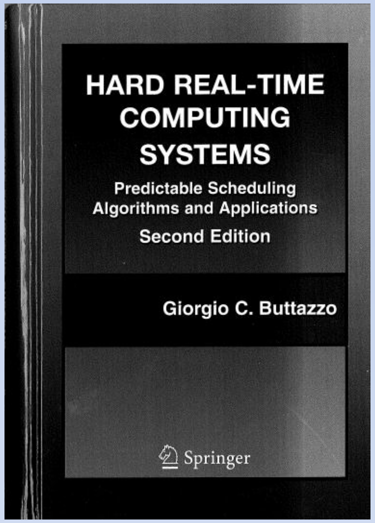
Topics

****

After this course you should be able to explain and apply the fundamental concepts and terminology of real-time systems.

Readings

:

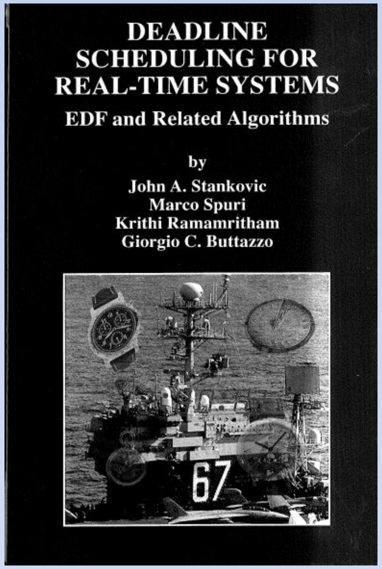

HARD REAL-TIME COMPUTING SYSTEMS

Predictable Scheduling Algorithms and Applications

Second Edition

Giorgio C. Buttazzo

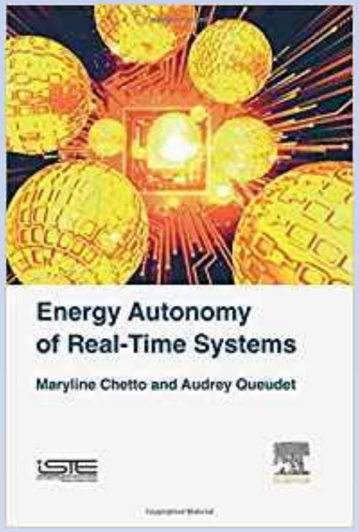
Springer


DEADLINE SCHEDULING FOR REAL-TIME SYSTEMS

EDF and Related Algorithms

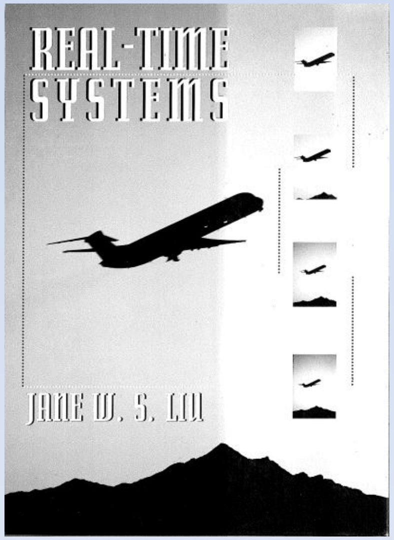
by

John A. Stankovic
Marco Spuri
Krithi Ramamritham
Giorgio C. Buttazzo


Energy Autonomy of Real-Time Systems

Maryline Chetto and Audrey Queudet

ISTE


REAL-TIME SYSTEMS

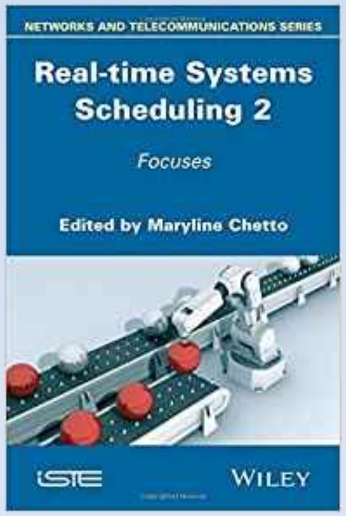
JANE W. S. LIU

COLLECTION RÉSEAUX ET TÉLÉCOMMUNICATIONS

Ordonnancement dans les systèmes temps réel

sous la direction de
Maryline Chetto

ISTE editions


NETWORKS AND TELECOMMUNICATIONS SERIES

Real-time Systems Scheduling 2

Focuses

Edited by Maryline Chetto

ISTE WILEY

Thanks to Giorgio Buttazzo and Mike Holenderski !

:

ABOUT ME

Business Address :

- Academic Institute of Technology of Nantes (IUT)
- LS2N (Laboratoire des Sciences du Numérique de Nantes)

Business tel. :+33 2 28 09 21 09   +33 6 10 20 35 94

Email: maryline.chetto@univ-nantes.fr

Position: Professor at University of Nantes (France)

REAL TIME COMPUTING

:

INTRODUCTION TO REAL TIME SYSTEMS

•Real-time systems have been defined as:

"those systems in which the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced";

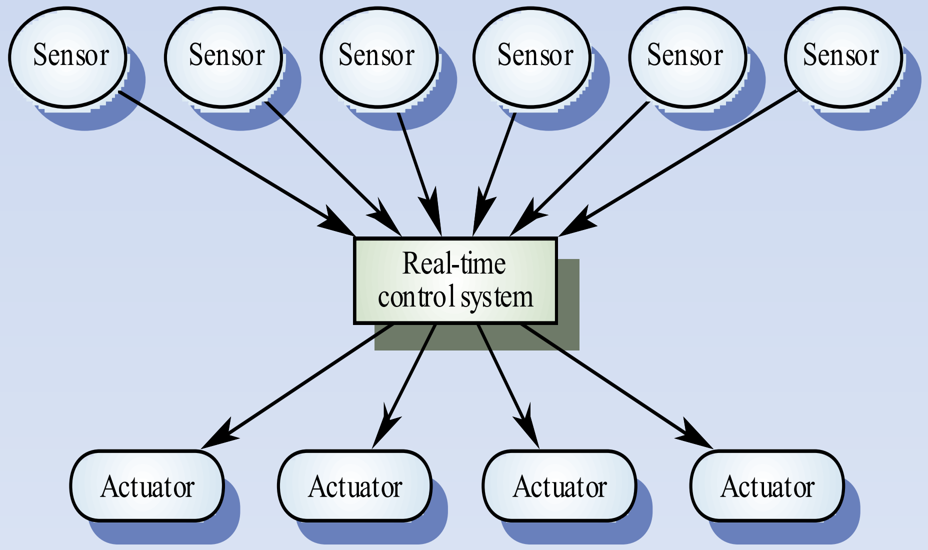
J. Stankovic, "Misconceptions About Real-Time Computing," ***IEEE Computer,*** 21(10), October 1988.

Examples of temporal constraints:

- Few milliseconds for radar systems.
- One second for machine-man interfaces (in an aircraft for example).
- Hours for some chemical reactions.
- 24 hours for weather forecast.

Monitoring and control systems = Important class of real-time systems

- Continuously check sensors and take actions depending on sensor values
- Monitoring systems examine sensors and report their results



EXAMPLE: CAR DRIVER

Mission: Reaching the destination safely.

Controlled System: Car.
Operating environment: Road conditions.
Controlling System
- *Human driver:* Sensors - Eyes and Ears of the driver.
- *Computer:* Sensors - Cameras, Infrared receiver, and Laser telemeter.

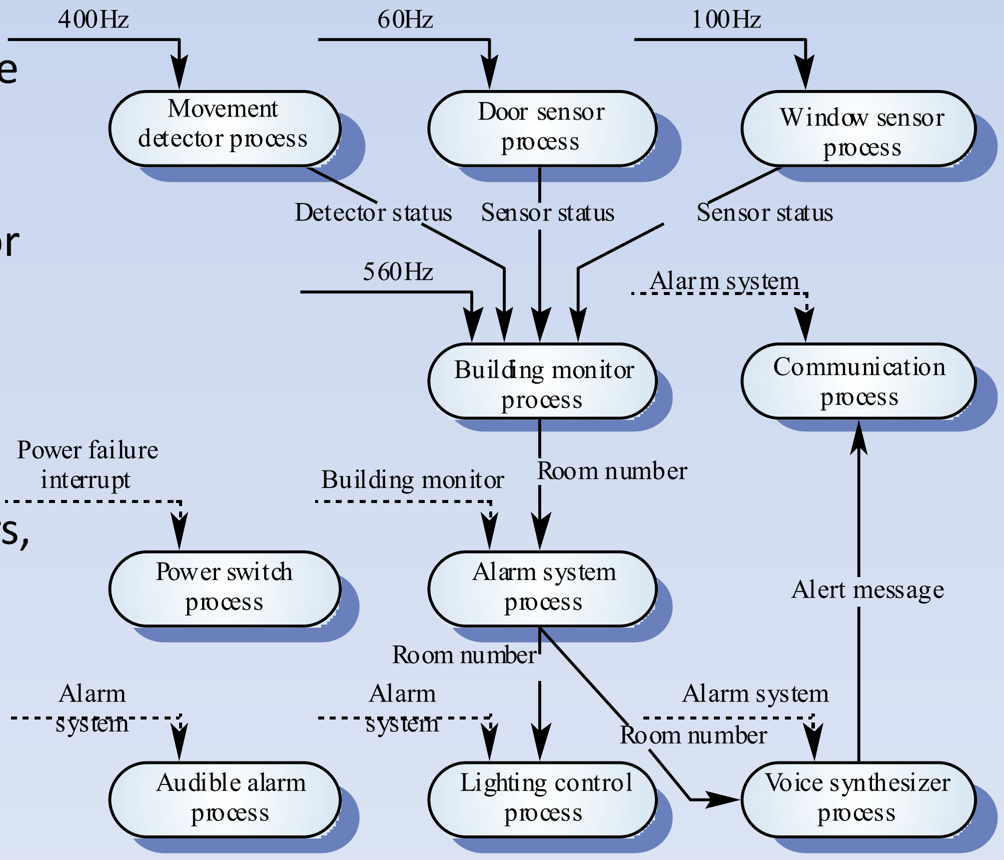
Controls: Accelerator, Steering wheel, Break-pedal.
Actuators: Wheels, Engines, and Brakes.

Critical tasks: Steering and breaking.
Non-critical tasks: Turning on radio.

:

EXAMPLE : A BURGLAR ALARM SYSTEM

A system is required to monitor sensors on doors and windows to detect the presence of intruders in a building. When a sensor indicates a break-in, the system switches on lights around the area and calls police automatically. The system should include provision for operation without a mains power supply.



Sensors

•Movement detectors, window sensors, door sensors.

•50 window sensors, 30 door sensors and 200 movement detectors

Actions

•When an intruder is detected, police are called automatically.

•Lights are switched on in rooms with active sensors.

•An audible alarm is switched on.

:

Other Examples

Combat aircraft flight control system

If a pilot gives a command (e.g. lower landing gear) he wants it to be done now, 10 seconds from now can be hazardous.
→ **Hard real-time**

Multimedia presentation system
It doesn't really matter if all the deadlines are met on time as long as most of the time the deadlines are met.
→ **Soft real-time**

:

TIMING CONSTRAINTS

Time is critical : Real-time systems **MUST** respond within specified times (usually short)

Real-time does not mean "*fast computing*" !!!!!

Main goal: **predictability**
Fastness remains a desired feature

Systems can be distinguished by the result of a delayed response:

- **A 'soft' real-time system** is a system whose operation is degraded if results are not produced according to the specified timing requirements

Failure to respond in time can be **accepted**.
Response time is **important but not crucial**.

- **A 'hard' real-time system** is a system whose operation is incorrect if results are not produced according to the timing specification

Failure to respond can be considered **as bad as wrong response**.

STIMULUS / RESPONSE SYSTEMS

Given a stimulus, the system must produce a response within a specified time

Periodic stimuli. Stimuli which occur at predictable time intervals
•For example, a temperature sensor may be polled 10 times per second

Aperiodic stimuli. Stimuli which occur at unpredictable times
•For example, a system power failure may trigger an interrupt which must be processed by the system

:

RT DESIGN PROCESS

- Identify the stimuli to be processed and the required responses to these stimuli For each stimulus and response, identify the timing constraints
- Aggregate the stimulus and response processing into concurrent processes (as in figure with sensor/actuator process).
- Design algorithms to process each class of stimulus and response. These must meet the given timing requirements
- Design a scheduling system which will ensure that processes (tasks) are started in time to meet their deadlines
- Integrate using a real-time executive or Real-Time Operating System

:

CHARACTERISTICS OF REAL-TIME SYSTEMS

Real-time systems tend to be:

•**Embedded system**: a component of a larger hardware/software system. (Many people use the term embedded systems means real-time systems).

•**Concurrent system**: the system simultaneously controls and/or reacts to different aspects of the environment, many events that need to be processed in parallel.

•**Safety critical system**: not only reliable but also safe, if it fails then without causing injury or loss of life. The development of safe system involves architectural redundancy.

•**Reactive system**: there is a continuous interaction with the environment, event-driven and must respond to external stimuli, system response is typically state dependent. Behavior of event-driven system primarily caused by reactions to external events.

:

Constraints on the design of an embedded system

Limited number of resources

Trade-off between cost, size, power consumption and real-time constraints

Energy-constrained

Battery technology is improving rather slowly

Battery influences autonomy, size, weight, and cost

Goal: minimize energy consumption guaranteeing real-time constraints

:

REAL-TIME OPERATING SYSTEMS

:

INTRODUCTION TO TASKS

Real time systems are implemented as a set of concurrent **tasks (processes) on a RTOS**

RTOS: Real-Time Operating System

A task is an executing program, including the current values of the program counter, registers, and variables. The subtle difference between a process and a program is that the program is a group of instructions whereas the task is the activity.

Multitasking is required to develop a good Real-time application

"Pseudo parallelism" - to handle multiple external events occurring simultaneously

:

O.S BASIS

An OS is a system program that provides **an interface**
between **application programs (tasks)**
and **the computer system (hardware)**

Primary Functions

- Provide a system that is convenient to use
- Organize efficient and correct us of system resources

A *Good* RTOS is one that has a bounded (predictable) behavior under all system load scenarios

:

RTOS COMPONENTS

Real-time clock
•Provides information for task scheduling.

Interrupt handler
•Manages aperiodic requests for service.

Scheduler
•Chooses the next task to be run.

Dispatcher
•Starts task execution.

Resource manager
•Allocates memory and processor resources (Services for file creation, deletion, reposition and protection)

:

Input/Output Management

•Handles requests and release subroutines for a variety of peripherals and read, write and reposition programs

Inter-task Communication

•Synchronization and coordination

•Deadlock and Livelock detection

•task Protection

•Data Exchange Mechanisms

Configuration manager

- Responsible for the dynamic reconfiguration of the system software and hardware.
- Hardware modules may be replaced and software upgraded without stopping the systems

Fault manager

- Responsible for detecting software and hardware faults
- and taking appropriate actions (e.g. switching to backup disks) to ensure that the system continues in operation
-

:

PRINCIPLE OF MULTIPROGRAMMING

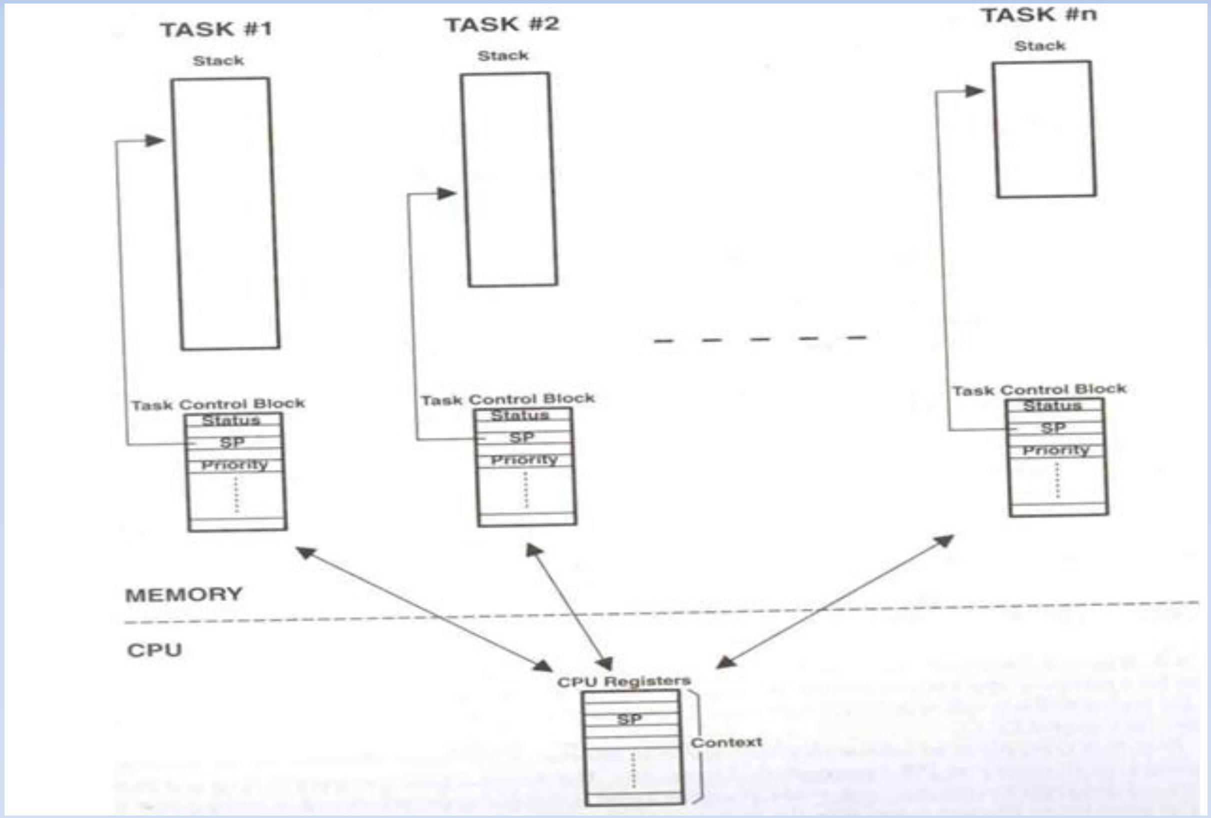
In multiprogramming systems, tasks are performed in a pseudo-parallelism as if each task has its own processor. In fact, there is only one processor but it switches back and forth from task to task.

In the OS, each task is represented by its **TCB (Task Control Block)** also called **Task Descriptor**.

The TCB, generally contains the following information:

- Task State,
- Task ID,
- Program Counter (PC) value,
- Register values
- Memory Management Information (page tables, base/bound registers etc.)
- Processor Scheduling Information (priority, deadline, period, etc.)
- I/O Status Info (outstanding I/O requests, I/O devices held, etc.)
- List of Open Files
- …

:



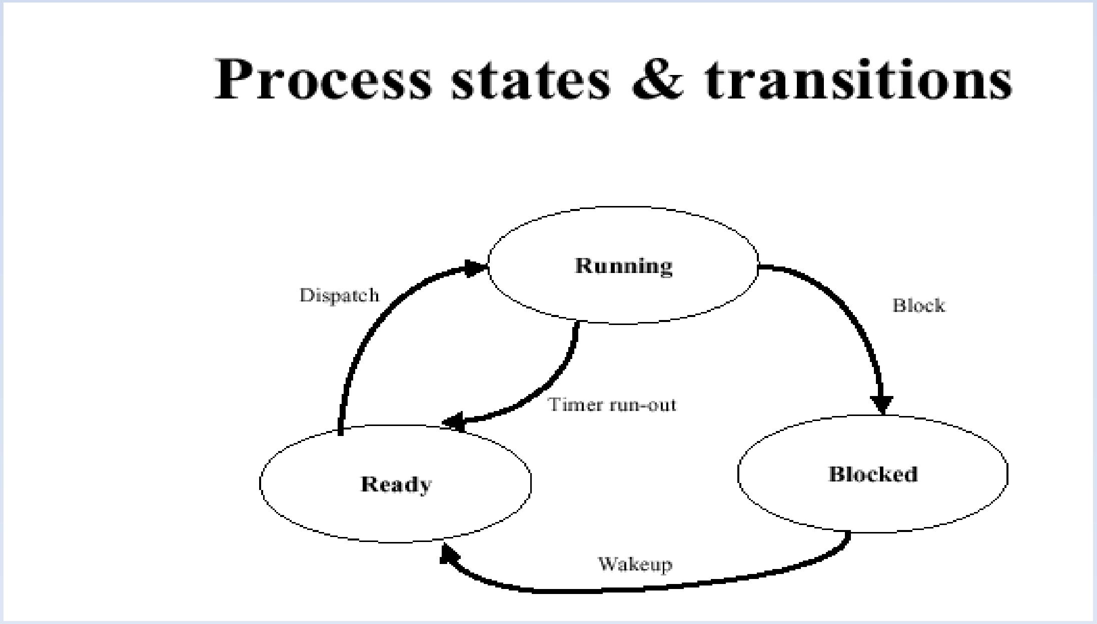
The **dispatcher** takes the task from ready list, loads it onto a processor and starts execution

Algorithms (**scheduling mechanism**) are needed to decide which task will run at a given time

At least 3 states are needed to allow the CPU to schedule.

Ready – waiting to run (in ready list)
Running – process (thread or task) is utilizing the processor to execute instructions
Blocked – waiting for resources (I/O, memory, critical section, etc.)

The **scheduler** chooses the next task to be executed by the processor. Function is to switch from one task to another (***Context Switching***)



:

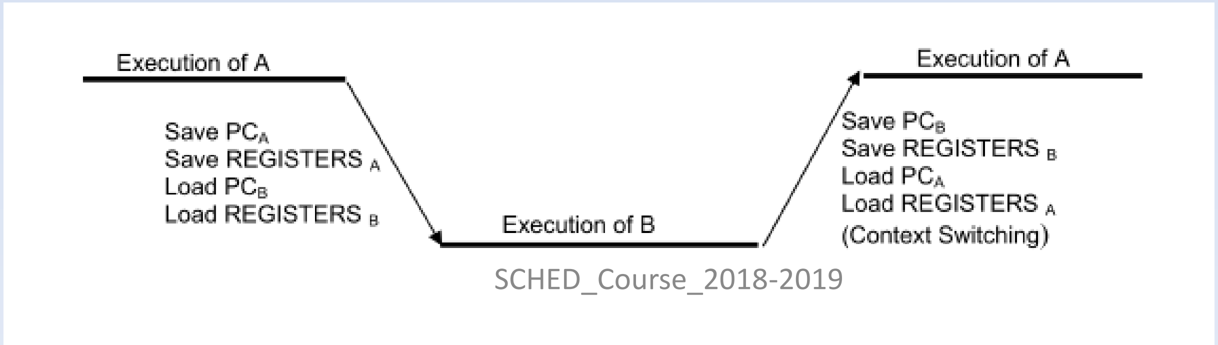
Overhead – should be completed as quickly as possible
Dispatch time should be independent of the number of tasks in the ready list
Ready list should be organized each time an element is added to the list

If we have a single processor in our system, there is only one running task at a time. Other ready tasks wait for the processor. The system keeps these ready tasks (their TCBs) on a list called **Ready Queue** which is generally implemented as a linked-list structure.

When a task is allocated by the processor, it executes and after some time it either finishes or passes to waiting state (for I/O).

In multiprogramming systems, the processor can be switched from one task to another.

Note that when an interrupt occurs, PC and register contents for the running task (which is being interrupted) must be saved so that the task can be continued correctly afterwards. Switching between tasks occurs as depicted below.



:

PERFORMANCE CRITERIA

In order to achieve an efficient processor management, OS tries to select the most appropriate task from the ready queue.

Processor Utilization: The ratio of busy time of the processor to the total time passes for taskss to finish. We would like to keep the processor as busy as possible.
Processor Utilization = (Processor buy time) / (Processor busy time + Processor idle time)

Throughput: The measure of work done in a unit time interval.
Throughput = (Number of tasks completed) / (Time Unit)

Waiting Time (wt): Time spent in ready queue.

Response Time (rt): The amount of time it takes to start responding to a request. This criterion is important for interactive systems.
rt = t(first response) – t(submission of request)

We, normally, want to maximize the processor utilization and throughput, and minimize response time and waiting time..

:

Synchronization and exclusion objects

Required so that tasks can execute critical code and to guarantee access in a particular order

Semaphores – synchronization and exclusion

Signals – asynchronous event

Mutual exlusion

The easiest way for tasks to communicate is through shared data structures

Global variables, pointers, buffers, linked lists, and ring buffers

Must ensure that each task has exclusive access to the data to avoid data corruption

The most common method is using **semaphores**.

- Invented by Dijkstra in the mid-1960s
- Offered by most multitasking OS
- Used for:
  - ✓ Mutual exclusion
  - ✓ Signaling the occurrence of an event
  - ✓ Synchronizing activities among tasks

:

A semaphore is a key that your code acquires in order to continue execution
If the key is already in use, the requesting task is suspended until the key is released
There are two types

- **Binary semaphores** : 0 or 1
- **Counting semaphores** : >= 0

Semaphore Operations

Initialize (or create)

- Value must be provided
- Waiting list is initially empty

Wait (or pend)

- Used for acquiring the semaphore
- If the semaphore is available (the semaphore value is positive), the value is decremented, and the task is not blocked
- Otherwise, the task is blocked and placed in the waiting list

:

Signal (or post)

- Used for releasing the semaphore
- If no task is waiting, the semaphore value is incremented
- Otherwise, make one of the waiting tasks ready to run but the value is not incremented
- Which waiting task to receive the key?
  - Highest-priority waiting task
  - First waiting task

Example:

```
Wait(s);
Access shared data;
Signal(s);
```

:

Applications of Binary Semaphores

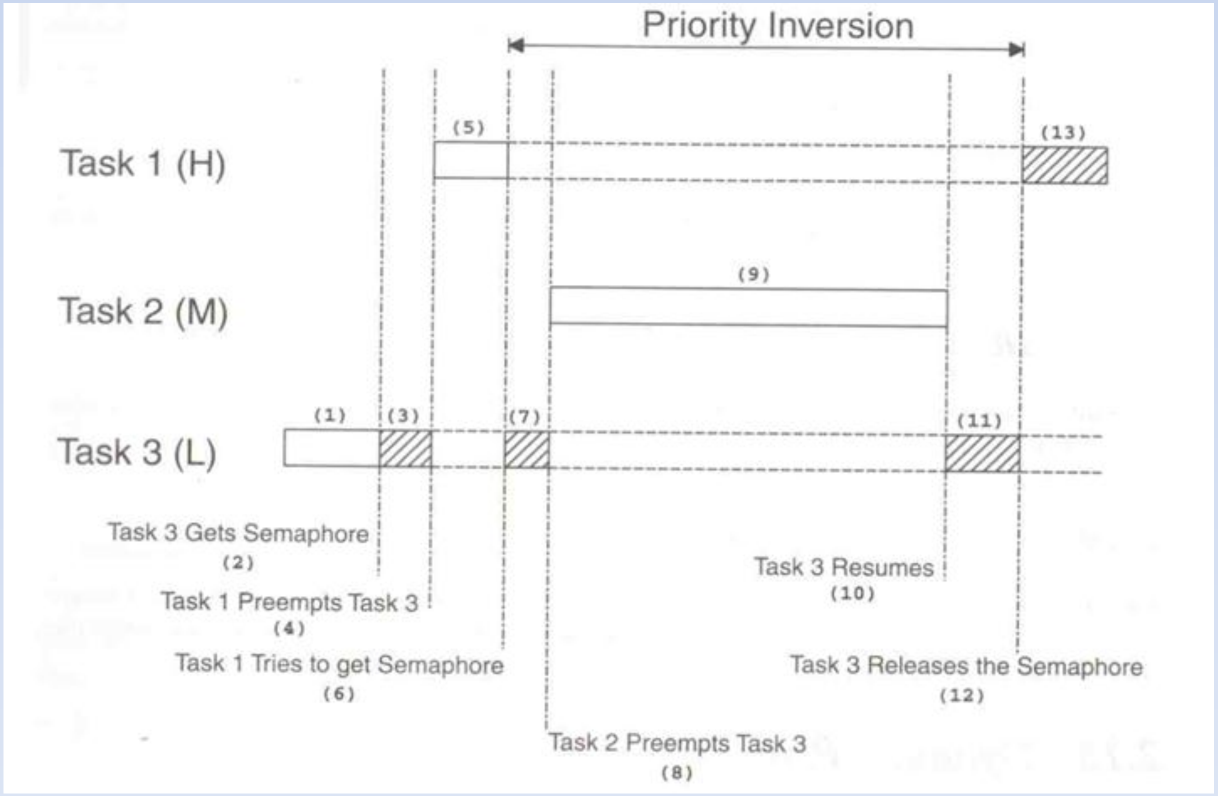
Suppose task 1 prints **"I am Task 1!"**

Task 2 prints **"I am Task 2!"**

If they were allowed to print at the same time, it could result in:

I Ia amm T Tasask k1!2!

The priority inversion problem



:

Interrupts

RT systems respond to external events
External events are translated by the hardware and interrupts are introduced to the system

Interrupt Service Routines (ISRs) handle system interrupts

May be stand alone or part of a device driver
RTOSs should allow lower level ISRs to be preempted by higher lever ISRs
ISRs must be completed as quickly as possible

:

EXISTING REAL-TIME OPERATING SYSTEMS

- Proprietary kernels
- Real-time extensions to general-purpose OS

Proprietary Kernels

- Homegrown kernels
- Highly specialized for specific applications
- e.g., nuclear power plant
- Less common now
- Commercial RTOS

Features for Efficiency

- Small
- Minimal set of functionality
- Fast context switch
- Fast and time bounded response to interrupts

:

OSEK : STANDARD FOR O.S. IN VEHICLES

OSEK (***O****ffene* ***S****ysteme und deren Schnittstellen für die* ***E****lektronik in* ***K****raftfahrzeugen*; English: "Open Systems and their Interfaces for the Electronics in Motor Vehicles") is a standards body that has produced specifications for an embedded operating system, a communications stack, and a network management protocol for automotive embedded systems.

OSEK was designed to provide a standard software architecture for the various **electronic control units** (ECUs) throughout a car.

OSEK was founded in 1993 by a German automotive company consortium (BMW, Siemens…) and the University of Karlsruhe.

In 1994, the French cars manufacturers Renault and PSA Peugeot Citroën, which had a similar project called VDX (Vehicle Distributed eXecutive), joined the consortium. Therefore, the official name is **OSEK/VDX**.

:

Code Size of several RTOS

Name	Code Size	Target CPU
pOSEK	2K	Microcontrollers
pSOSystem		PII->ARM Thumb
VxWorks	286K	Pentium -> Strong ARM
QNX Nutrino	>100K	Pentium II -> NEC
QNX RealTime	100K	Pentium II -> SH4
OS-9		Pentium -> SH4
Chorus OS	10K	Pentium -> Strong ARM
ARIEL	19K	SH2, ARM Thumb
Creem	560 bytes	ATMEL 8051

:

Features for Real-Time

- Preemptive priority scheduling
- At least 32 priority levels, commonly 128-256 priority levels
- Priority inheritance/ceiling protocol
- System calls
- Bounded execution times
- Short non-preemptable code
- High-resolution system clock
- Resolution down to nanoseconds
- But it takes about a microsecond to process a timer interrupt

Real-Time Extensions to General-Purpose OS

- Generally slower and less predictable than RTOS
- Much greater functionality and development support
- Standard interfaces
- Useful for soft real-time and complex applications

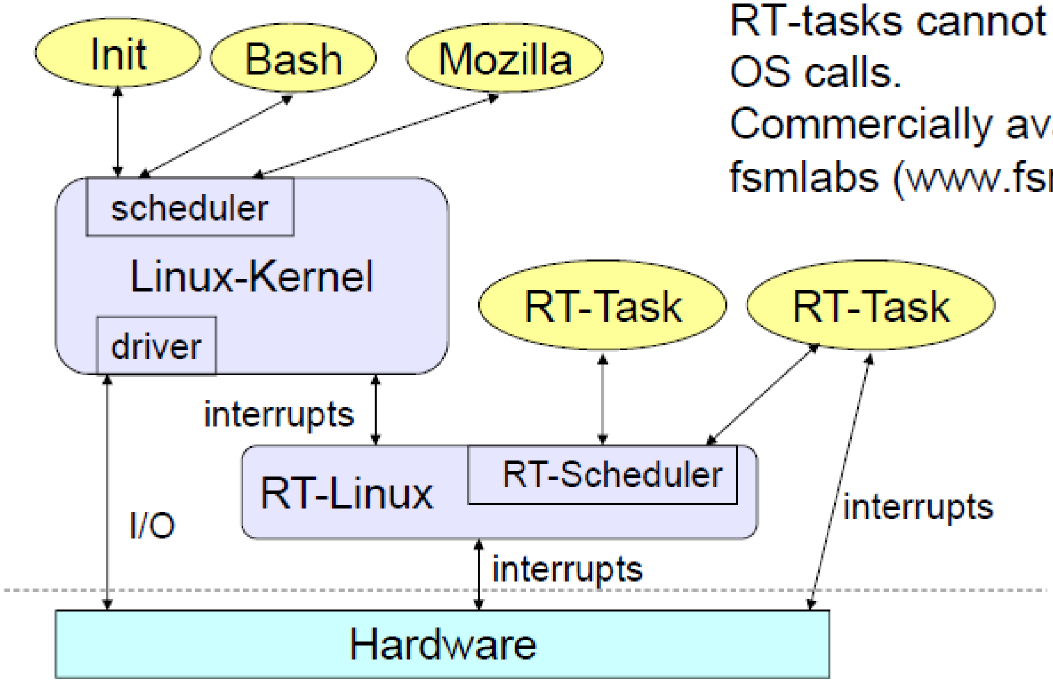
TYPE OF OPERATING SYSTEM USED (2006)

	%
Commercial OS	51
Internally developed or in-house OS	21
Open-source OS without commercial support	16
Commercial distribution of an open-source OS	12

Wind River (VxWorks) 26 %

:

EXAMPLE : RT LINUX



RT-tasks cannot use standard OS calls.
Commercially available from fsmlabs (www.fsmlabs.com)

:

BASIS OF SCHEDULING

:

BACKGROUND ON REAL TIME SYSTEMS

The environment changes in real time and imposes timing constraints on the system.

The input corresponds to some movement in the physical world, and the output has to relate to the same movement.

The lag from input time must be small enough to acceptable timeliness

:

NOTION OF DEADLINE

Definition: A point in time when some work must be finished is called a deadline
A real time system has performance **deadlines** on its computations and actions.

A deadline is either :

- a point in time (time-driven) → **absolute deadline**
- or a delta-time interval (event-driven) by which a system action must occur.

→ **relative deadline**

A deadline is often measured relative to the occurrence of some event
• When the bill arrives, pay it within 10 days
• At 9am, complete the exam in 5 hours

Meeting a deadline usually amounts to generating some specific response before the specified time
• Signal level must reach 10mV before...

:

SOFT VS. HARD REAL-TIME SYSTEM

A hard real-time system is a system where it is absolutely imperative that the responses occur within the required deadline.

- safety-critical applications
- e.g.,aerospace, automotive,....

A soft real-time system is a system where deadlines are important but where the system still functions if the deadlines are occasionally missed.

- e.g.multimedia, user interfaces,...

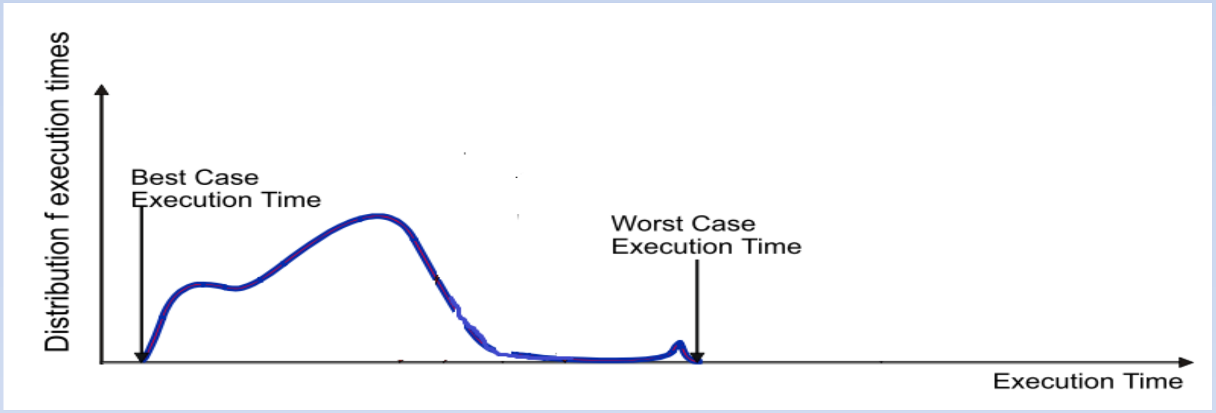
Common misconception:

•real-time = high-speed computations **not true!!!**

•execute at a speed that makes it possible to fulill the timing requirements **true!!!**

:

NOTION OF EXECUTION TIME

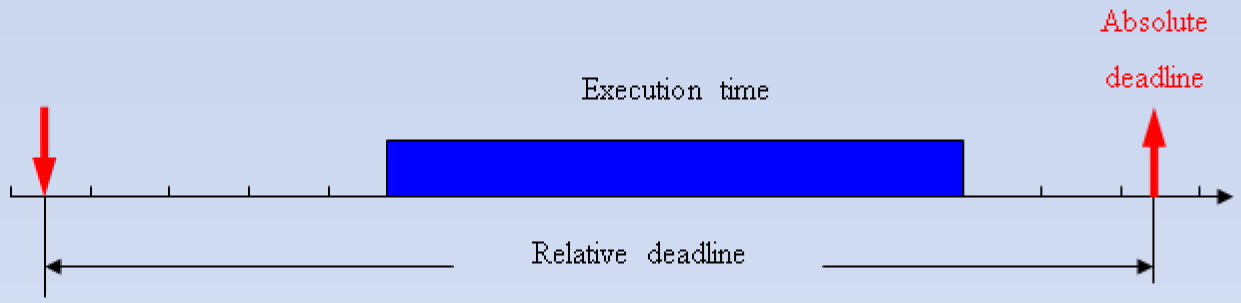


Modern processors increase performance by using: Caches, Pipelines,

These features make WCET computation difficult: Execution times of instructions vary widely.

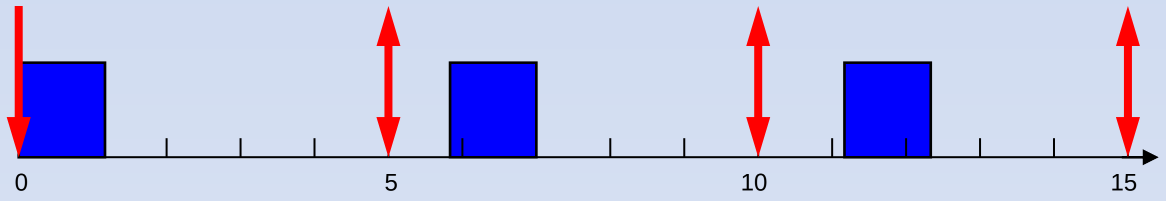
 Best case - everything goes smoothely: no cache miss, operands ready, needed resources free, branch correctly predicted.

 Worst case - everything goes wrong: all loads miss the cache, resources needed are occupied, operands are not ready.


Absolute deadline
Execution time
Relative deadline

Periodic task (*p*,*e*) : repeat regularly

•Period *p* = inter-release time (0 < *p*)
•Execution time *e* = maximum execution time (0 < *e* < *p*)
•Utilization U = e/p



:

DEADLINES: HARD VS. SOFT

- **Hard deadline**
  - –Disastrous or very serious consequences may occur if the deadline is missed
  - –Validation is essential : can all the deadlines be met, even under worst-case scenario?
  - –Deterministic guarantees
- **Soft deadline**
  - –Ideally, the deadline should be met for maximum performance. The performance degrades in case of deadline misses.
  - –Best effort approaches / statistical guarantees

In summary:

- Hard deadline: late data may be a bad data.
- Soft deadline: late data may be a good data.

:

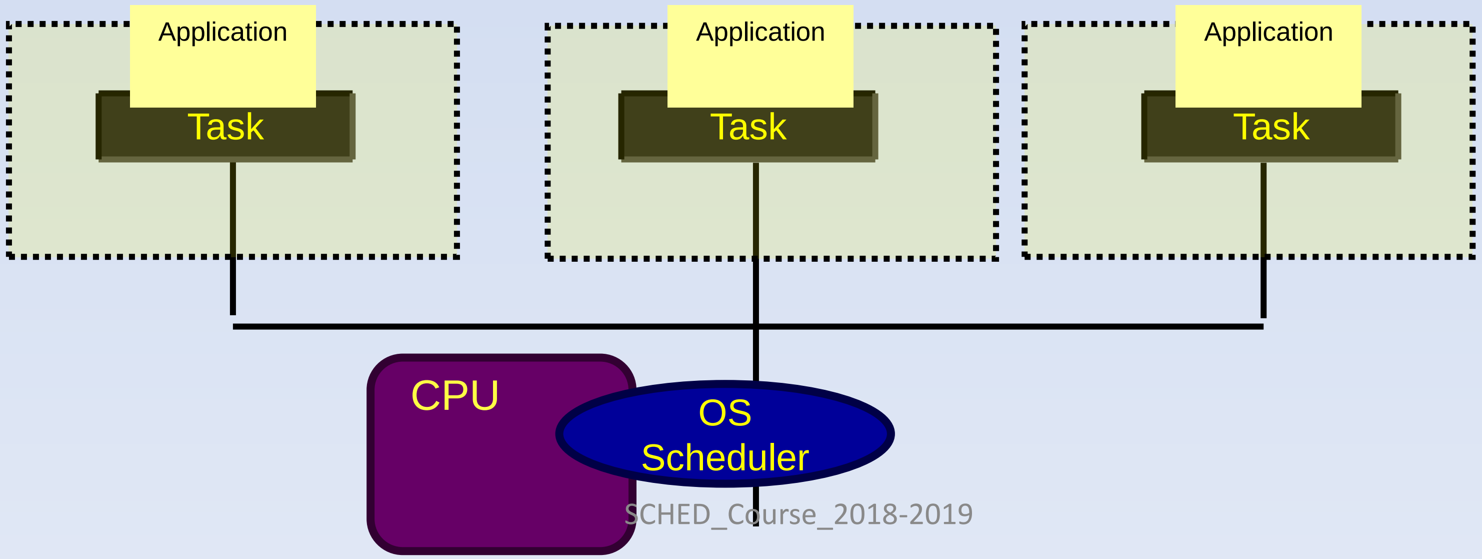
APPLICATIVE EXAMPLE : ENGINE CONTROL

Tasks:

 spark control
 crankshaft sensing
 fuel/air mixture
 oxygen sensor
 control algorithm

WHAT IS SCHEDULING ?

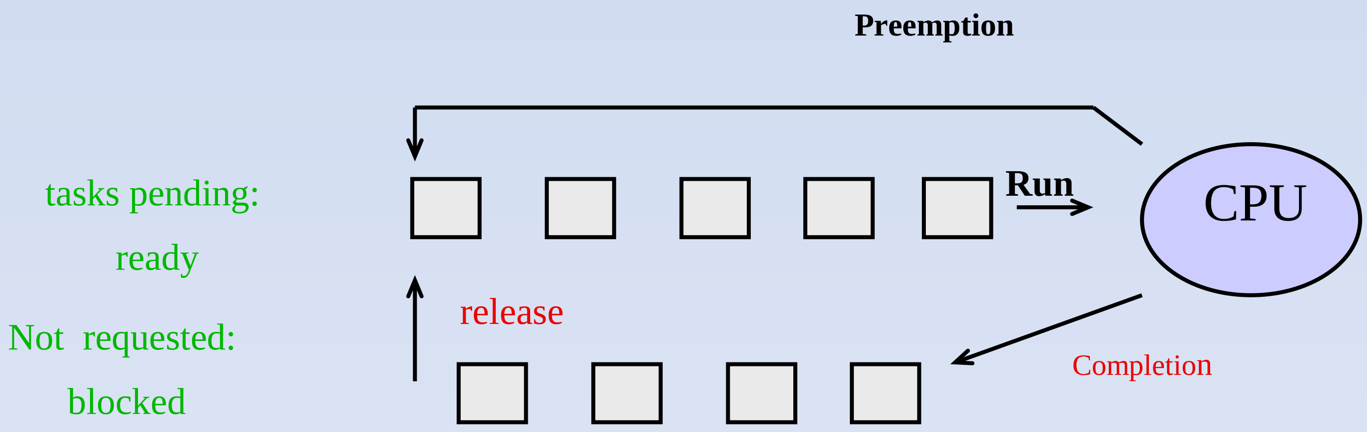
Scheduling is the assignment of tasks to resources (e.g., CPUs) over time, under a set of constraints, to achieve a given goal.



:

Remark:
If 2 tasks share a single processor, there are 2 ways of running one before the other
If 3 tasks share a single processor, there are 3*2 ways of running them in series
If n tasks share a single processor, there are n! ways of running them

Moreover, if tasks can be split into arbitrarily small fragments, there are infinitely many ways of running the fragments of even just 2 tasks!
The scheduler manages two queues for tasks in system: ready task list and blocked task list



:

GOALS OF SCHEDULING :

in General-purpose systems

- Fairness to all tasks (no starvation)
- Optimize throughput
- Optimize average performance

in real-time Embedded systems

- Meet all hard deadlines.
- Fairness or throughput is not important
- Hard real-time: worry about worst case performance

:

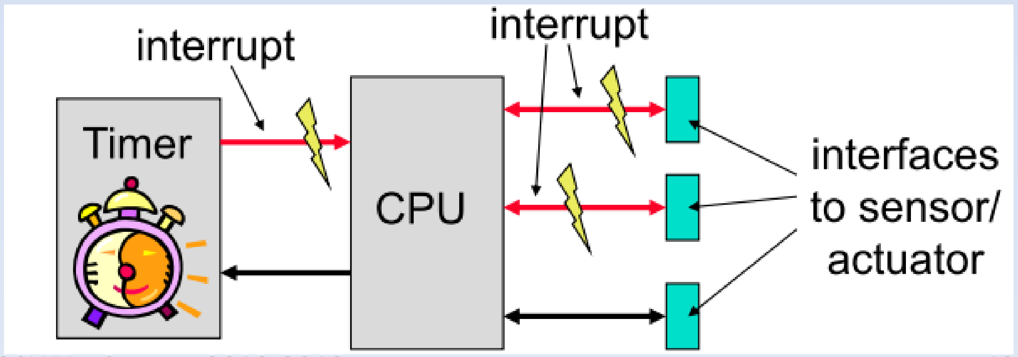
EVENT TRIGGERED SYSTEMS

The schedule of tasks is determined by the occurrence of events.

•**dynamic and adaptive**: there are possible problems with respect to timing, the use of shared resources and buffer over- or underflow

•guarantees can be given either off-line (if bounds on the behavior of the environment are known) or during run-time.

Principle:
To each event, there is associated a corresponding task that will be executed.
Events are emitted by (a) external interrupts and (b) by tasks themselves.
Events are collected in a queue; depending on the queuing discipline, an event is chosen for running.

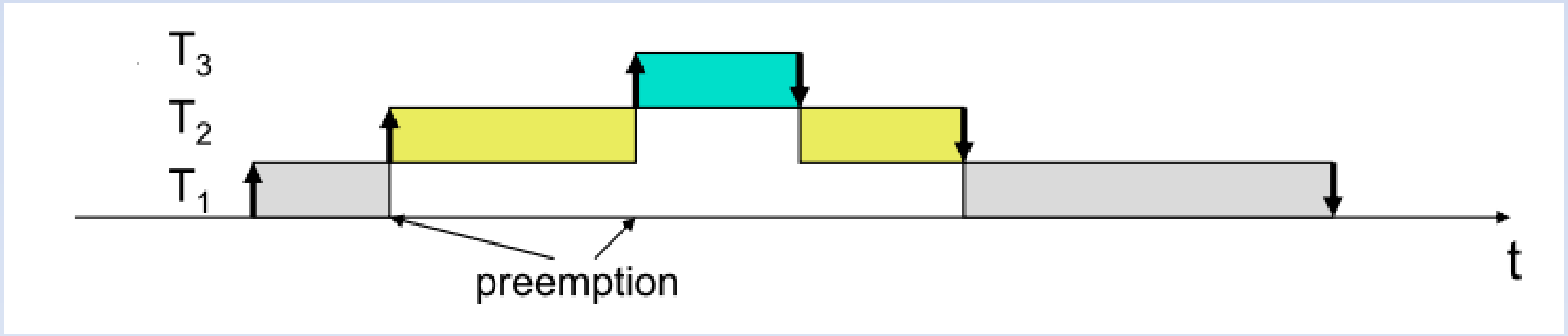


:

Preemptiveness

- A system where the scheduler is run only when a task calls the kernel (or terminates) is non-preemptive

- A system where it also runs as the result of interrupts is called preemptive



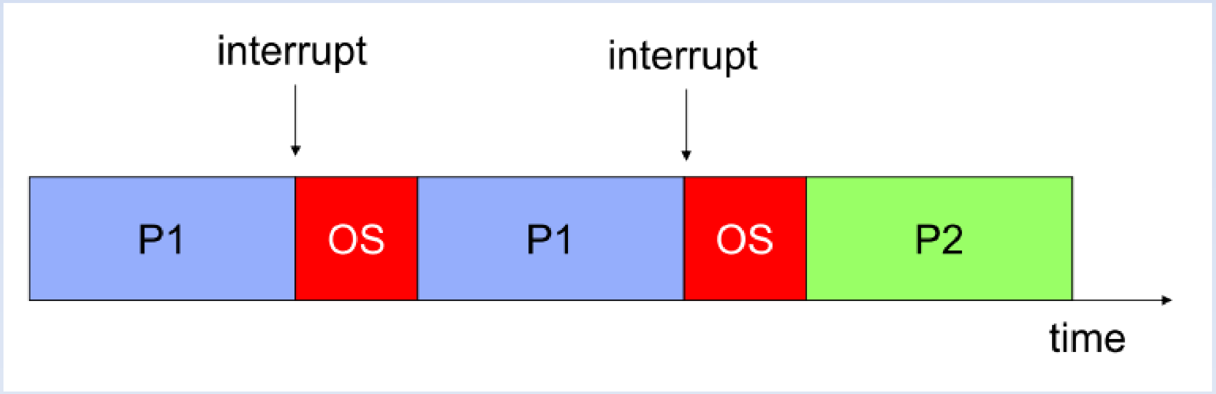
Remark:

Preemptive scheduling on basis of static priorities totally dominates the field of real-time programming

Supported by real-time operating systems like QNX, VxWorks, RTLinux, Lynx, and standards like POSIX

Also the basis of real-time languages like Ada and Real-time Java

Preemptive permits one task to preempt another one of lower priority



:

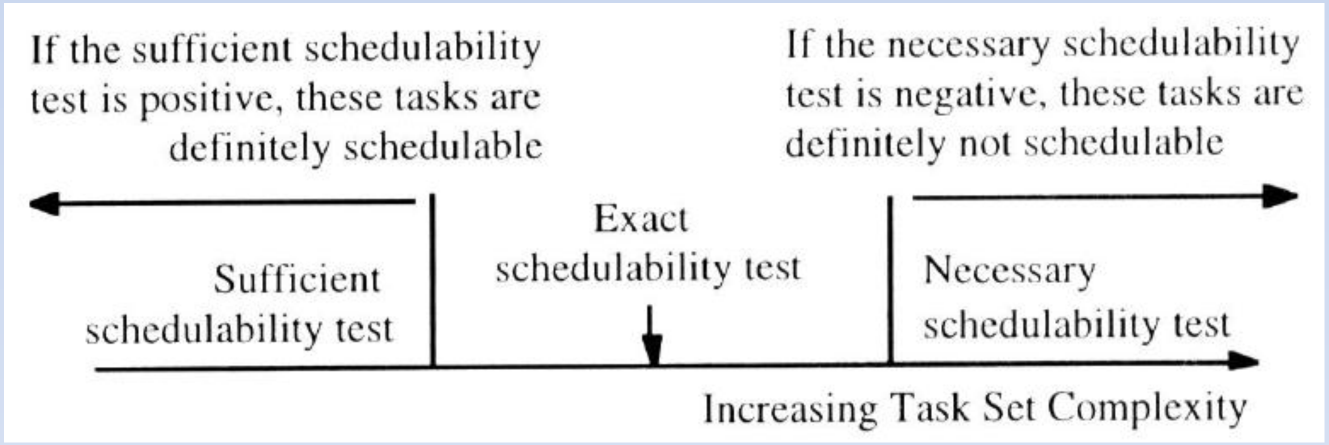
FIXED VS. DYNAMIC PRIORITIES

• A system where the programmer assigns the priorities of each task is said to use **static (or fixed) priorities**
• A system where priorities are automatically derived from some other run-time value is using **dynamic priorities**

Static priorities offer a way of assigning a relative importance to each task/thread/message
The highest priority task is offered the whole processor
Any cycles not used by this task are offered to the second but highest priority task
Any cycles not used by this task are offered to the third but highest priority task.
Etc...

SCHEDULABILITY TEST

Schedulability = Property indicating whether a real-time system (a set of real-time tasks) can meet their deadlines


If the sufficient schedulability test is positive, these tasks are definitely schedulable
If the necessary schedulability test is negative, these tasks are definitely not schedulable
Exact schedulability test
Sufficient schedulability test
Necessary schedulability test
Increasing Task Set Complexity

:

METRICS OF A SCHEDULING STRATEGY

- **Schedulability:** All jobs can meet their deadlines ?

- **Optimality:**

A Scheduling algorithm S is optimal if a task set is not schedulable under S → it is not schedulable under any other algorithms

- **Overhead:** Time required for scheduling.

REAL-TIME SCHEDULING CHALLENGES

- What are the optimal scheduling algorithms?

  (How to assign priorities to tasks?)

- For a given scheduling algorithm, what is the schedulability test ?

  (Can a system meet all deadlines?)

:

SCHEDULING PERIODIC TASKS

:

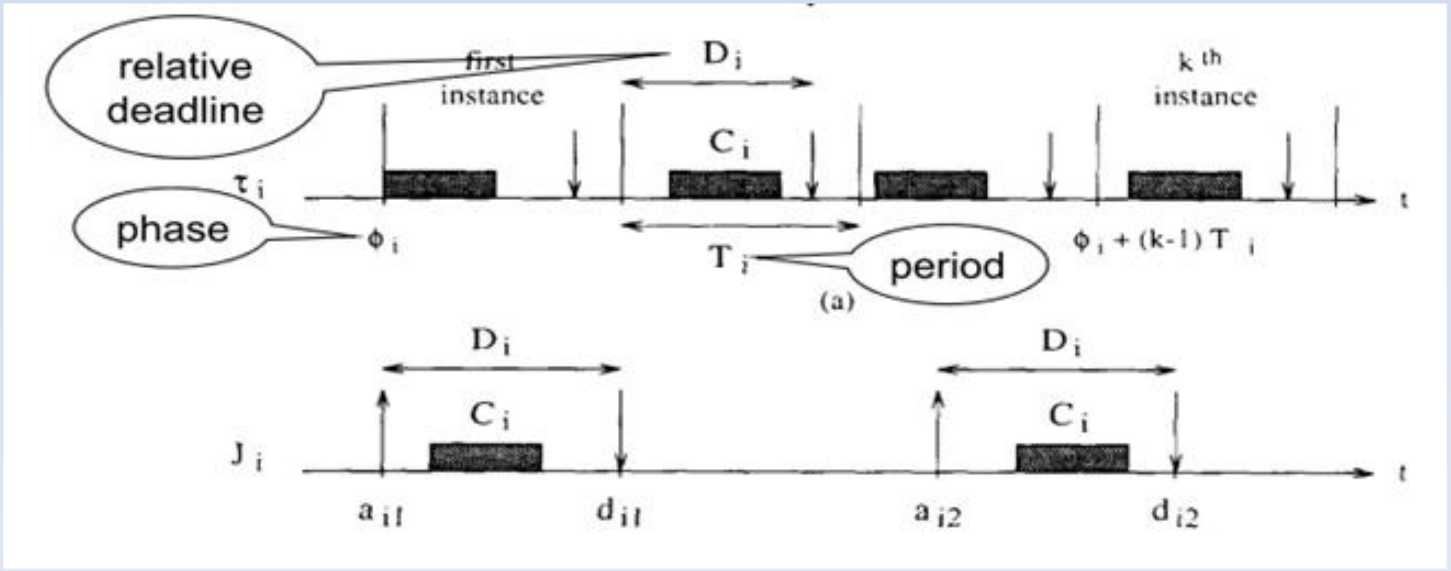
TERMINOLOGY

- **Job (*J_{ij}*)**: Unit of work, scheduled and executed by system. Jobs repeated at regular or semi-regular intervals modeled as periodic
- **Task (*T_i*)**: Set of related jobs.
- Jobs scheduled and allocated resources based on a set of scheduling algorithms and access control protocols**.**
- ***Scheduler***: Module implementing scheduling algorithms
- ***Schedule***: assignment of all jobs to available processors, produced by ***scheduler***.
- ***Valid schedule (or feasible schedule)*** : All jobs meet their deadline
- ***schedulable set of tasks*** : if there exists at least one scheduling algorithm that can produce feasible schedule

:

SCHEDULING PERIODIC TASKS

- In hard real-time systems, periodic tasks are known *a priori*
- **Task τ_i** is a series of **Jobs J_{ij}**. Each task has the following parameters
  - T_i period, minimum inter-release interval between jobs in Task T_i.
  - C_i maximum execution time for jobs in task T_i.
  - D_i :relative deadline
  - ϕ_i initial release time of Task τ_i (also called the phase).



Other characteristics of a task τ_i

laxity (slack time) $\mathbf{X_i = R_i - C_i}$ maximum time a task can be delayed on its activation to complete within deadline

utilization $\mathbf{u_i}$ factor of Task τ_i $\mathbf{u_i = C_i / T_i}$

Start time $\mathbf{s_i}$ is the time at which a task starts its execution.

Finishing time $\mathbf{f_i}$ is the time at which a task finishes its execution.

In addition the following parameters apply to a set of tasks:

H **Hyperperiod** = Least Common Multiple of T_i for all i:
$H = \text{LCM}(T_i)$, for all i.

U **Total utilization** = Sum over all u_i i.e. $U = \sum C_i / T_i$.
It is the fraction of processor time spent in the execution of the task set

Ch **Load factor**. Given by $Ch = \sum C_i / D_i$.

:

A necessary condition for a set of periodic tasks to be feasibly scheduled on one processor is that the total utilization be less than or equal to 1.

This means that if the total utilization of the task set is greater than one , the task set cannot be scheduled by any algorithm. We say that it is **overloaded**.

Other terminology:

•**Schedulable utilization of an algorithm** (Us) :
If U < Us the set of tasks can be guaranteed to be scheduled by the algorithm

:

FUNDAMENTAL RESULT : PERIODICITY OF THE SCHEDULE

This means that:

•if the processor does nothing at time t, it will do nothing at times t+kP with k=1,2,3,....

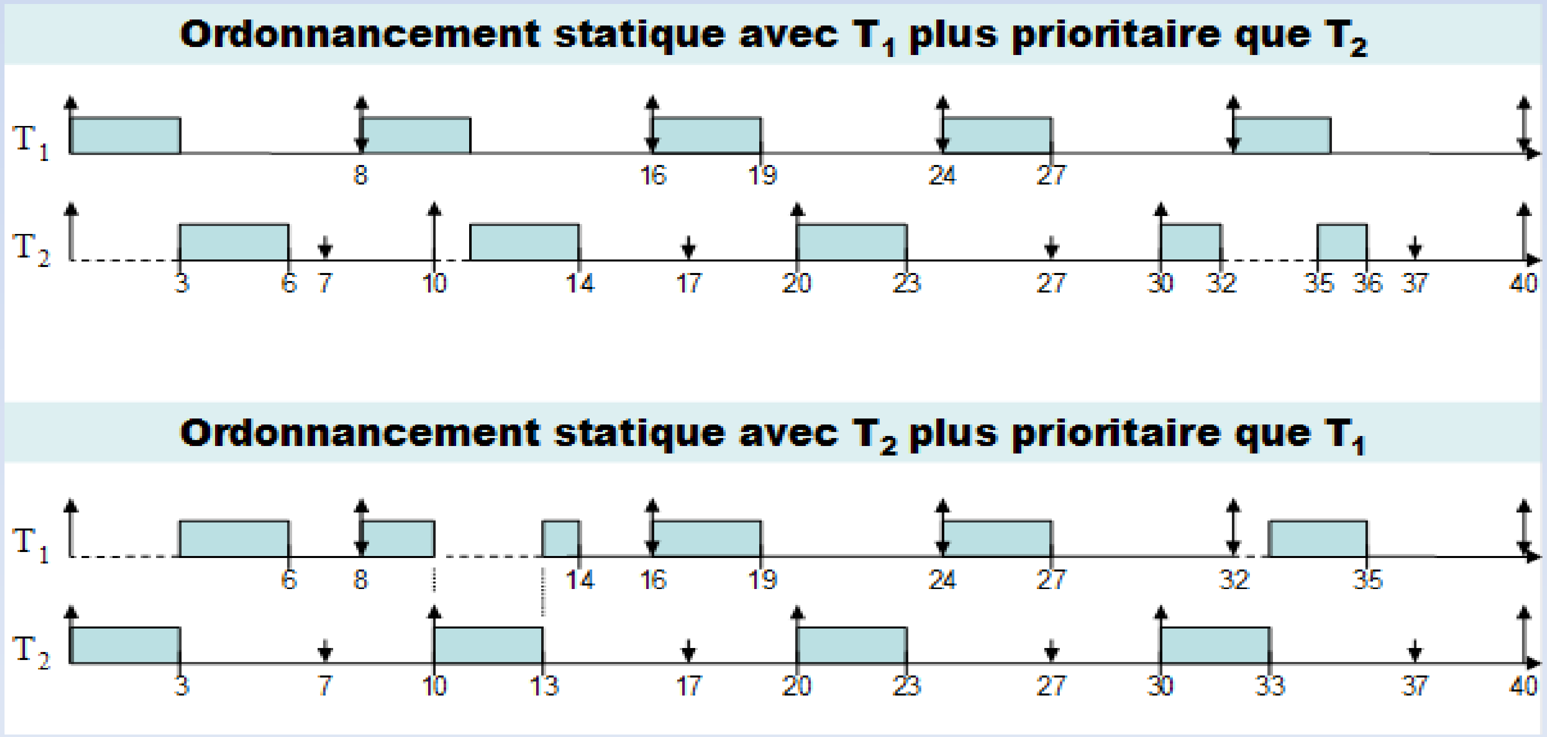
•if the processor executes a task τ_i at time t, it will execute the same task at times t+kH with k=1,2,3,....

The schedule that is produced on a periodic task set by a preemptive scheduling algorithm is

- **periodic**
- **with period equal to the least common multiple of the task periods.**

:

A periodic task set is feasibly scheduled by a preemptive scheduling algorithm **if and only if** it is feasibly scheduled over [0, H] where H is the least common multiple of task periods .


Ordonnancement statique avec T1 plus prioritaire que T2
T1
8
16
19
24
27
T2
3
6
7
10
14
17
20
23
27
30
32
35
36
37
40
Ordonnancement statique avec T2 plus prioritaire que T1
T1
6
8
14
16
19
24
27
32
35
T2
3
7
10
13
17
20
23
27
30
33
37
40

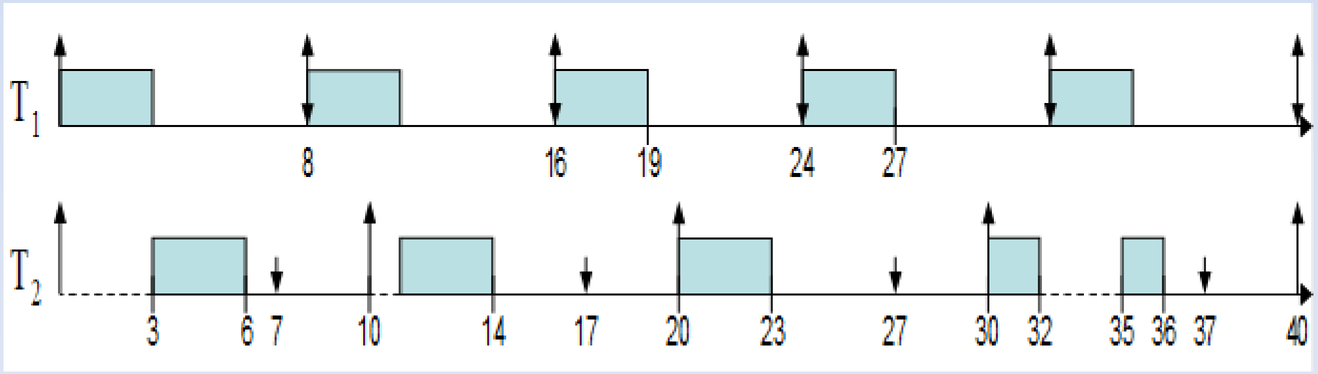
:

FIXED PRIORITY ALGORITHMS

Fixed-priority assignment: priorities assigned before the execution (all jobs of one task have the same priority)

Rate Monotonic scheduling

- Priority assignment based on periods of tasks
- Higher rate task assigned higher priority
- The most popular scheduler



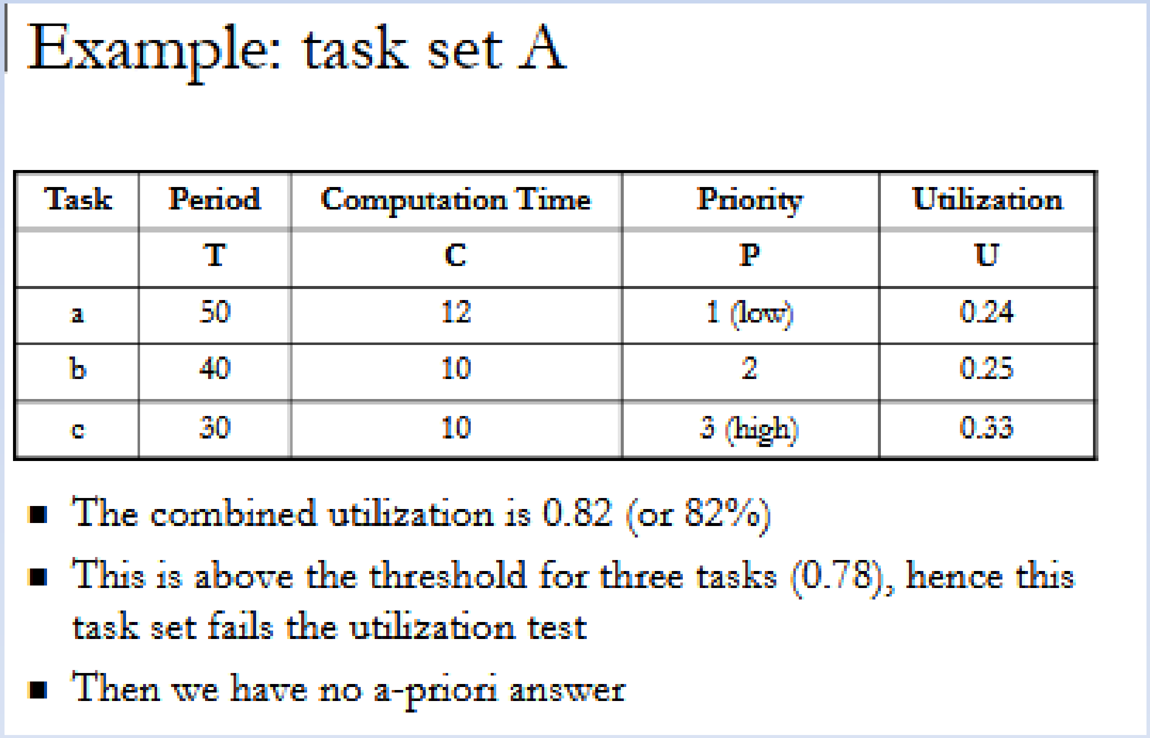
Utilization-based analysis

A set of n periodic tasks with deadline equal to period is feasibly scheduled by RM if $U \leq n(2^{1/n} - 1)$

(Liu and Layland 1973)

- Schedulable utilization = $n(2^{1/n} - 1)$
- For high values of n, the schedulable utilization converges to ln 2 ≈ 0.693
- RM cannot utilize 100% (1.0) of CPU, but for 1, 2, 3, 4, ... tasks : 1.00 , 0.83 , 0.78 , 0.76 , ... 0.69

Utilization-based analysis



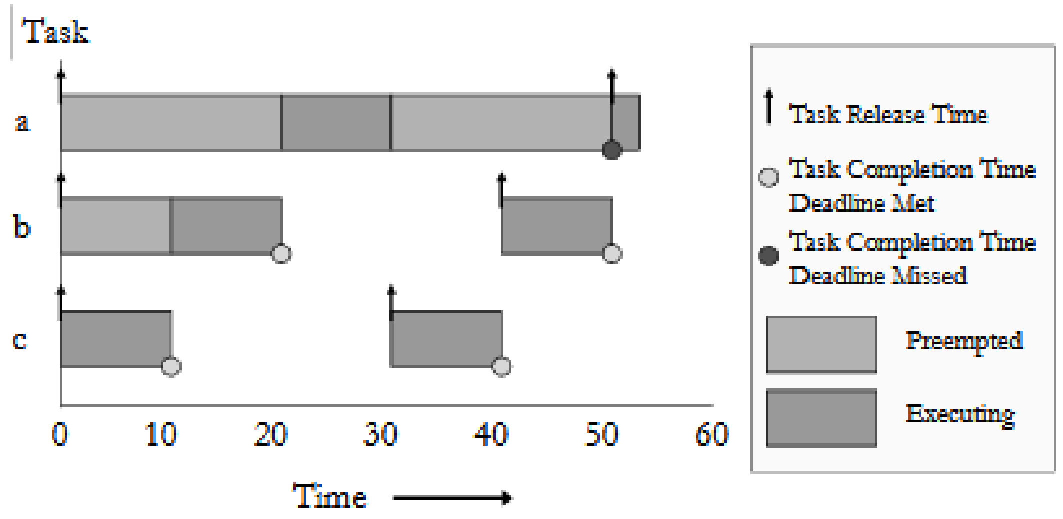
Example: task set A

Task	Period	Computation Time	Priority	Utilization
	T	C	P	U
a	50	12	1 (low)	0.24
b	40	10	2	0.25
c	30	10	3 (high)	0.33

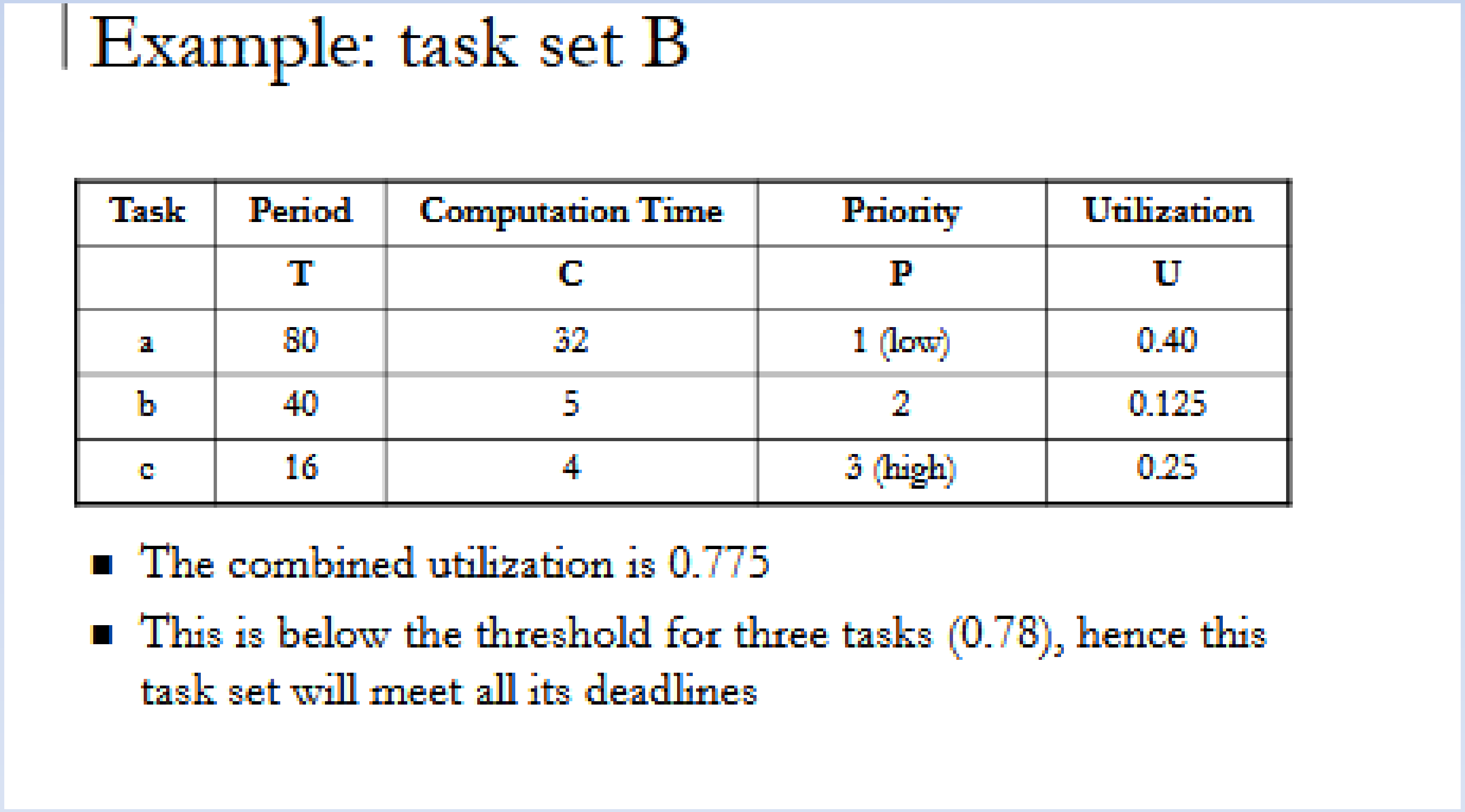
- The combined utilization is 0.82 (or 82%)
- This is above the threshold for three tasks (0.78), hence this task set fails the utilization test
- Then we have no a-priori answer

:

Timeline for task set A



:



Example: task set B

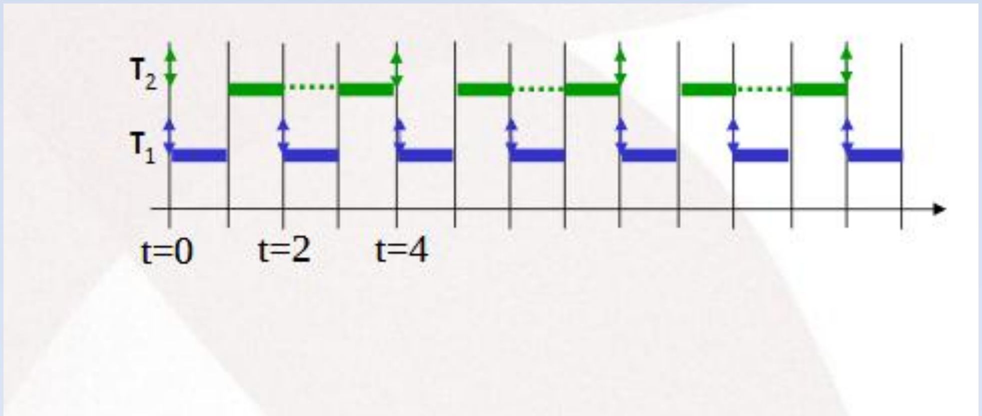
Task	Period	Computation Time	Priority	Utilization
	T	**C**	**P**	**U**
a	80	32	1 (low)	0.40
b	40	5	2	0.125
c	16	4	3 (high)	0.25

- The combined utilization is 0.775
- This is below the threshold for three tasks (0.78), hence this task set will meet all its deadlines

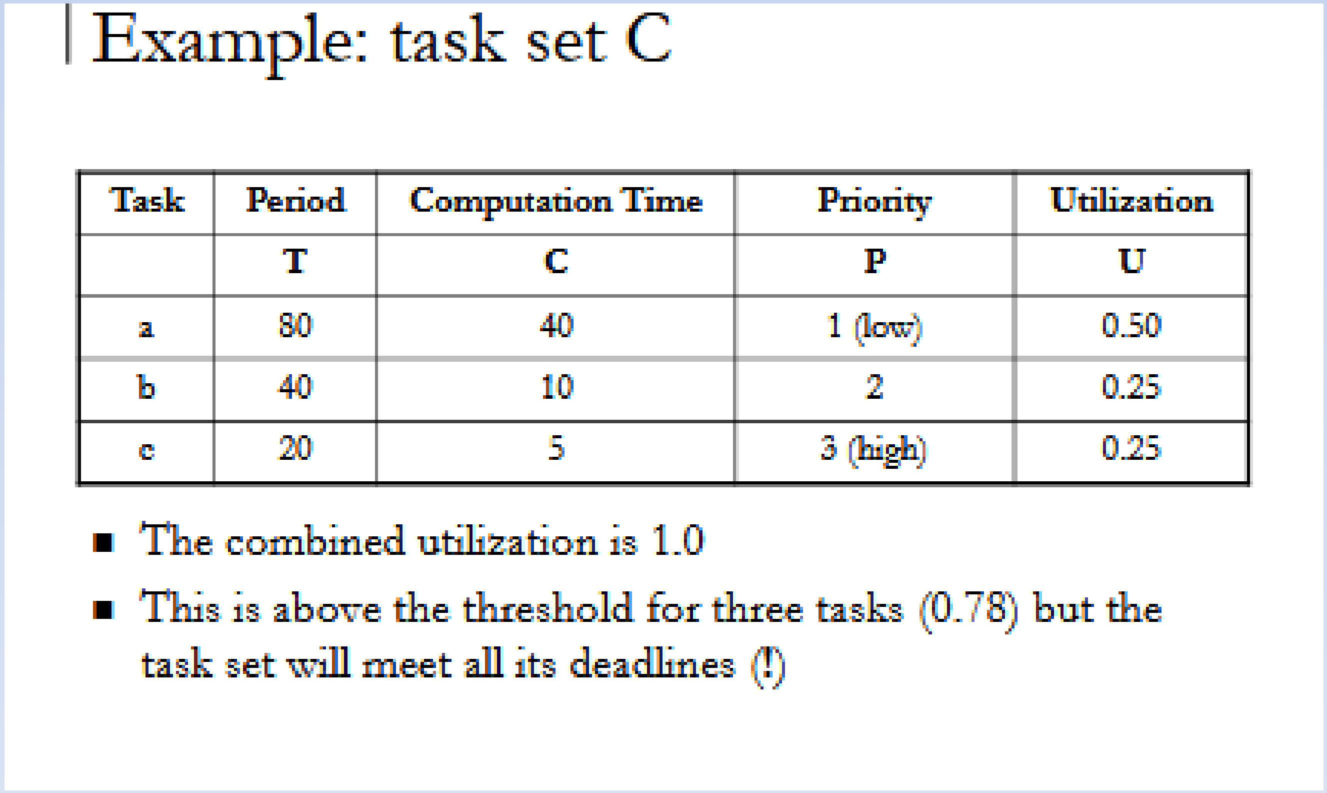
:

A periodic task set is simply periodic if for every pair of tasks τ_i and τ_k such that $T_i < T_k$, T_k is an integer multiple of T_i.

A set of simply periodic tasks with deadline equal to period is feasibly scheduled according to RM algorithm if and only if $U \leq 1$.



Harmonic periods

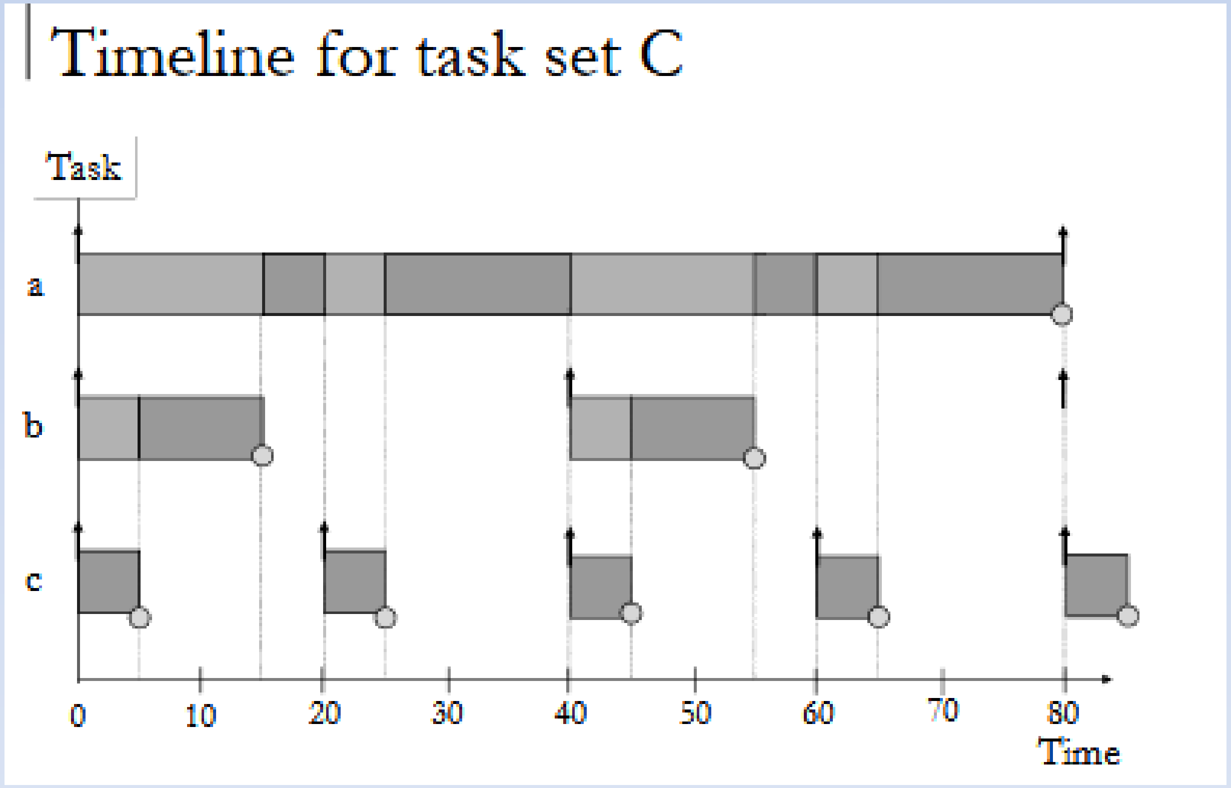


Example: task set C

Task	Period	Computation Time	Priority	Utilization
	T	C	P	U
a	80	40	1 (low)	0.50
b	40	10	2	0.25
c	20	5	3 (high)	0.25

- The combined utilization is 1.0
- This is above the threshold for three tasks (0.78) but the task set will meet all its deadlines (!)

:

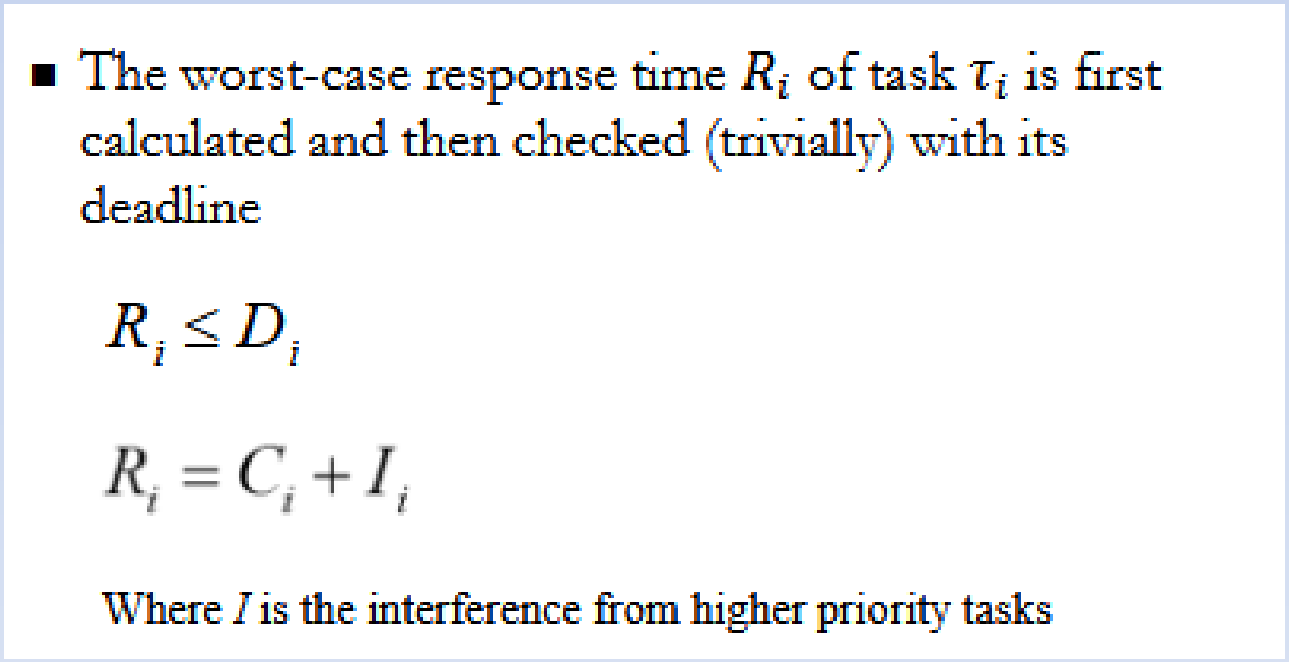

Timeline for task set C
Task
a
b
c
0
10
20
30
40
50
60
70
80
Time

Critique of utilization-based tests

- They are not exact
- They are not general
- But they are in O (N) Which makes them interesting for a large class of users

:

Exact schedulability tests based on the response time analysis (RTA)



- The worst-case response time R_i of task τ_i is first calculated and then checked (trivially) with its deadline

$$R_i \le D_i$$

$$R_i = C_i + I_i$$

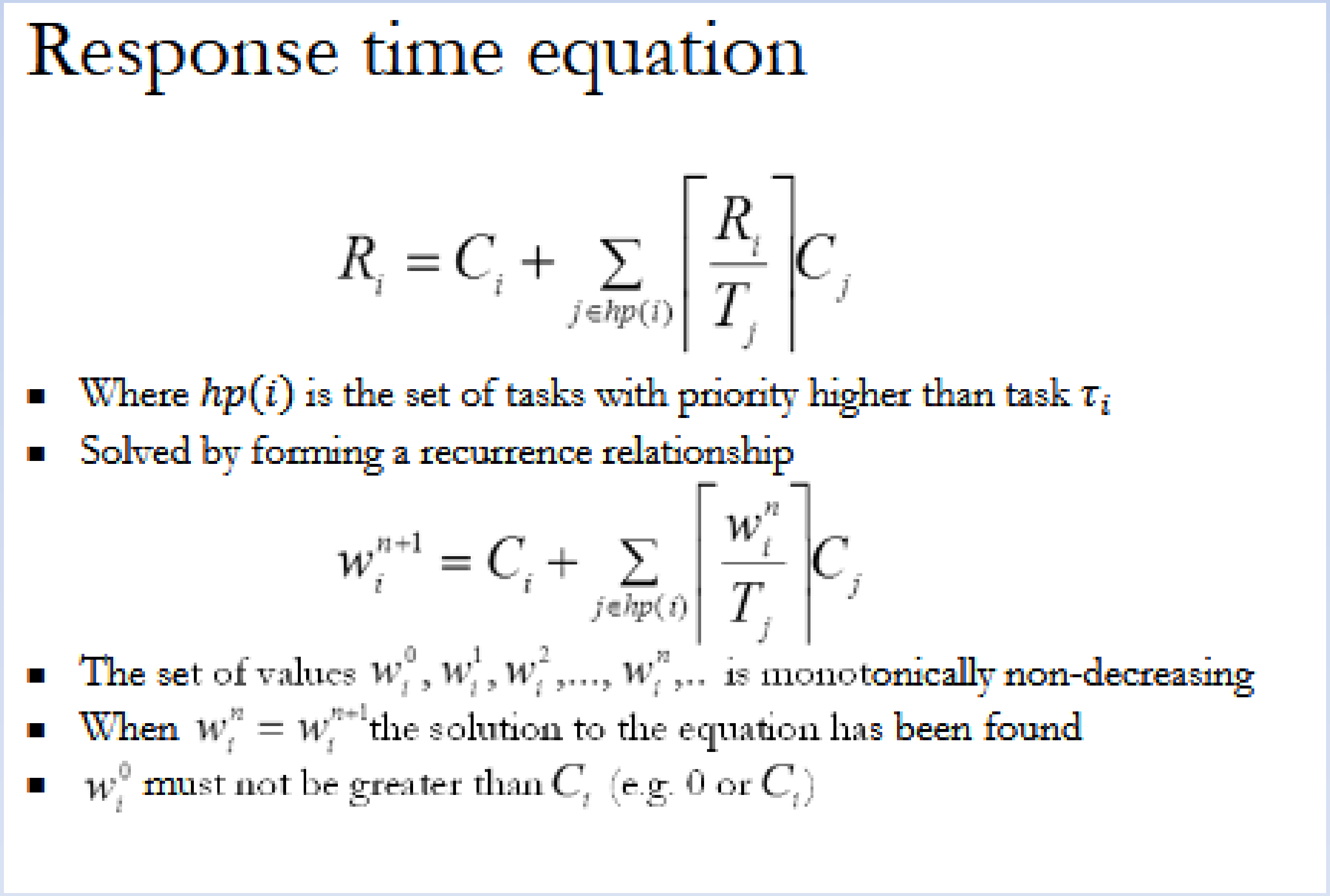
Where I is the interference from higher priority tasks

Critical instant : all higher priority tasks start at the same instant

:

- Within R_i, each higher priority task τ_j will execute a $\left\lceil \frac{R_i}{T_j} \right\rceil$ times
  - The ceiling function $\lceil f \rceil$ gives the smallest integer greater than the fractional number f on which it acts
    - E.g., the ceiling of $1/3$ is 1, of $6/5$ is 2, and of $6/3$ is 2
- The total interference suffered by τ_i from τ_j in R_i is given by $\left\lceil \frac{R_i}{T_j} \right\rceil C_j$

:



Response time equation

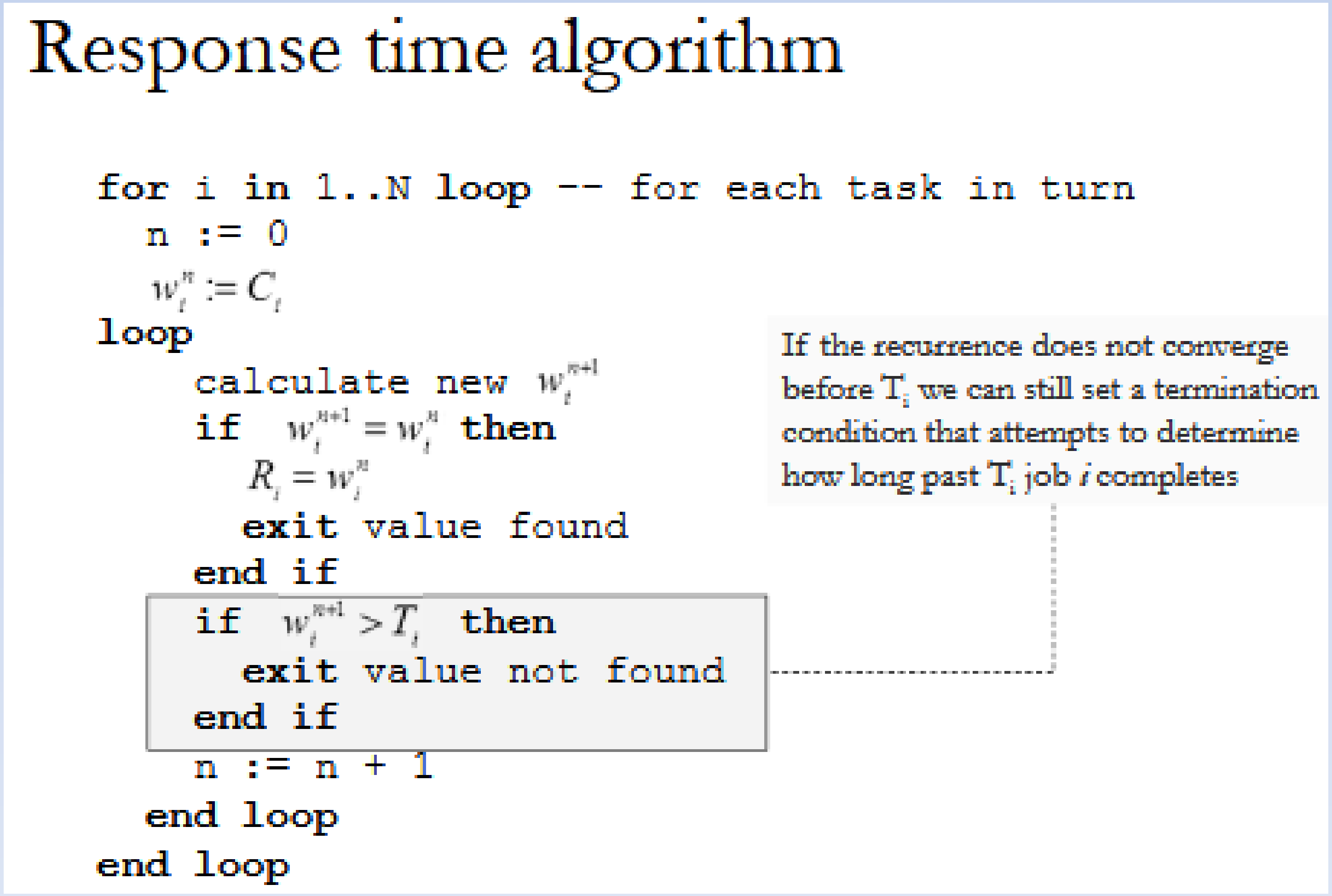
$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

- Where $hp(i)$ is the set of tasks with priority higher than task τ_i
- Solved by forming a recurrence relationship

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

- The set of values $w_i^0, w_i^1, w_i^2, \ldots, w_i^n, \ldots$ is monotonically non-decreasing
- When $w_i^n = w_i^{n+1}$ the solution to the equation has been found
- w_i^0 must not be greater than C_i (e.g. 0 or C_i)

:

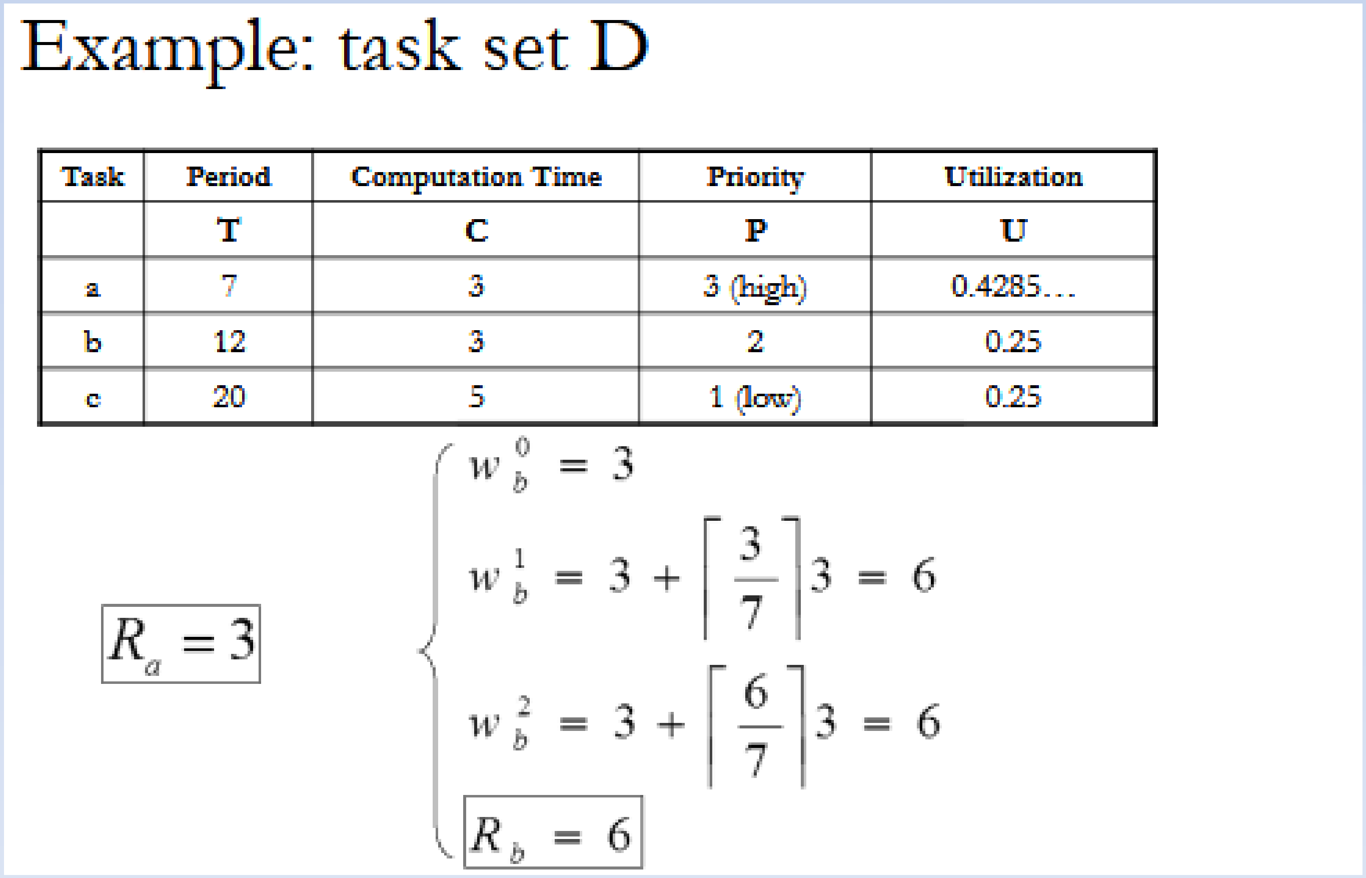


Response time algorithm

```
for i in 1..N loop -- for each task in turn
  n := 0
  $w_i^n := C_i$
loop
    calculate new $w_i^{n+1}$
    if $w_i^{n+1} = w_i^n$ then
      $R_i = w_i^n$
      exit value found
    end if
    if $w_i^{n+1} > T_i$ then
      exit value not found
    end if
    n := n + 1
  end loop
end loop
```

If the recurrence does not converge before T_i we can still set a termination condition that attempts to determine how long past T_i job *i* completes

:



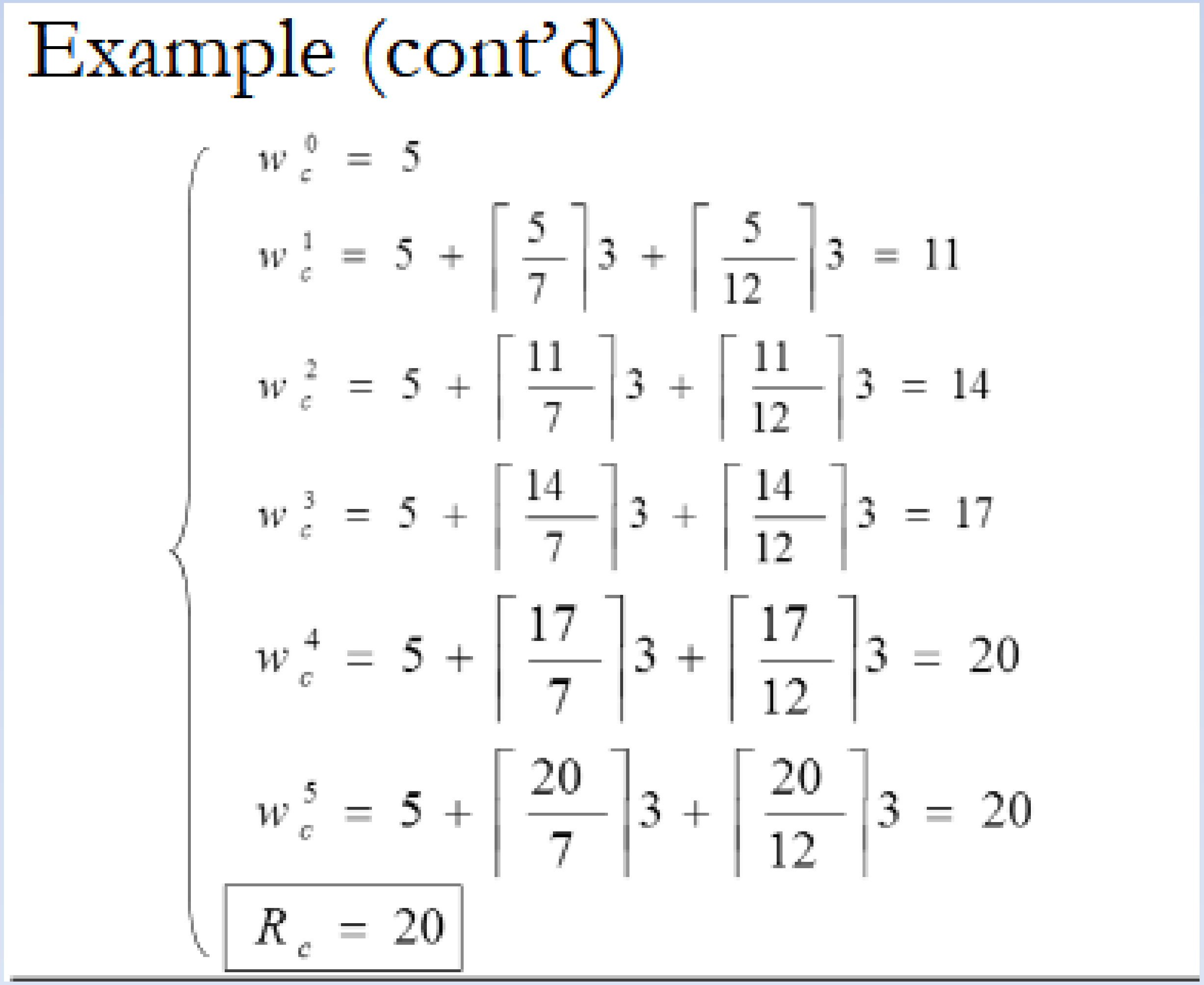
Example: task set D

Task	Period	Computation Time	Priority	Utilization
	T	**C**	**P**	**U**
a	7	3	3 (high)	0.4285…
b	12	3	2	0.25
c	20	5	1 (low)	0.25

$$\boxed{R_a = 3}$$

$$\begin{cases} w_b^0 = 3 \\ w_b^1 = 3 + \left\lceil \frac{3}{7} \right\rceil 3 = 6 \\ w_b^2 = 3 + \left\lceil \frac{6}{7} \right\rceil 3 = 6 \\ \boxed{R_b = 6} \end{cases}$$

:

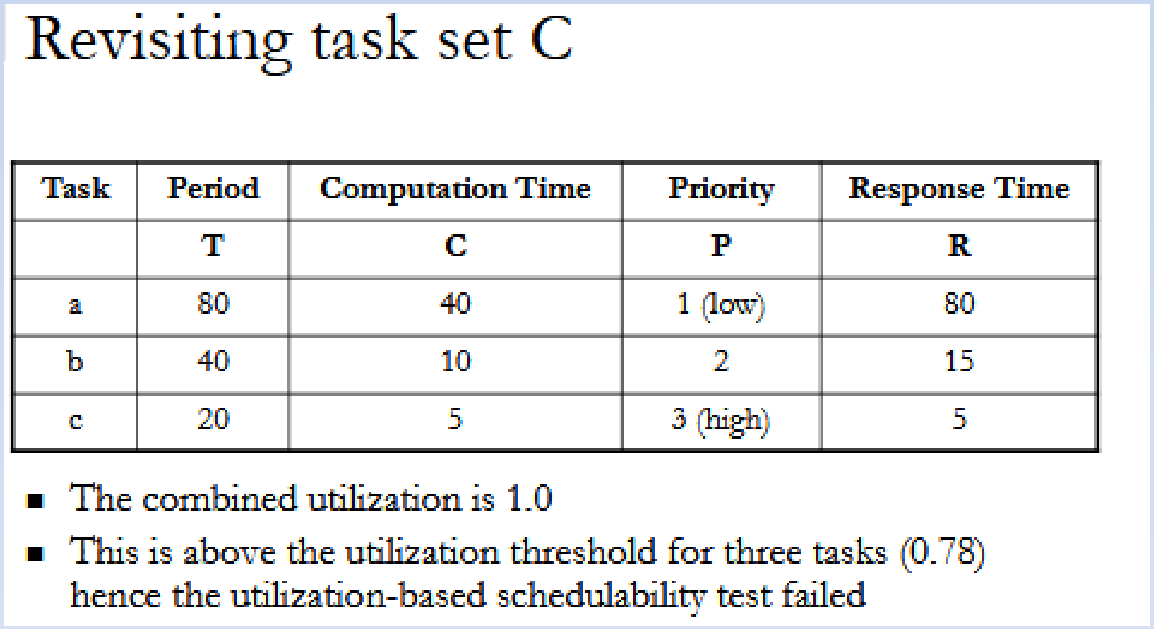


Example (cont'd)

$$\begin{cases} w_c^0 = 5 \\ w_c^1 = 5 + \left\lceil \frac{5}{7} \right\rceil 3 + \left\lceil \frac{5}{12} \right\rceil 3 = 11 \\ w_c^2 = 5 + \left\lceil \frac{11}{7} \right\rceil 3 + \left\lceil \frac{11}{12} \right\rceil 3 = 14 \\ w_c^3 = 5 + \left\lceil \frac{14}{7} \right\rceil 3 + \left\lceil \frac{14}{12} \right\rceil 3 = 17 \\ w_c^4 = 5 + \left\lceil \frac{17}{7} \right\rceil 3 + \left\lceil \frac{17}{12} \right\rceil 3 = 20 \\ w_c^5 = 5 + \left\lceil \frac{20}{7} \right\rceil 3 + \left\lceil \frac{20}{12} \right\rceil 3 = 20 \\ \boxed{R_c = 20} \end{cases}$$

:

Exercise: Apply RTA (Response Time Analysis)



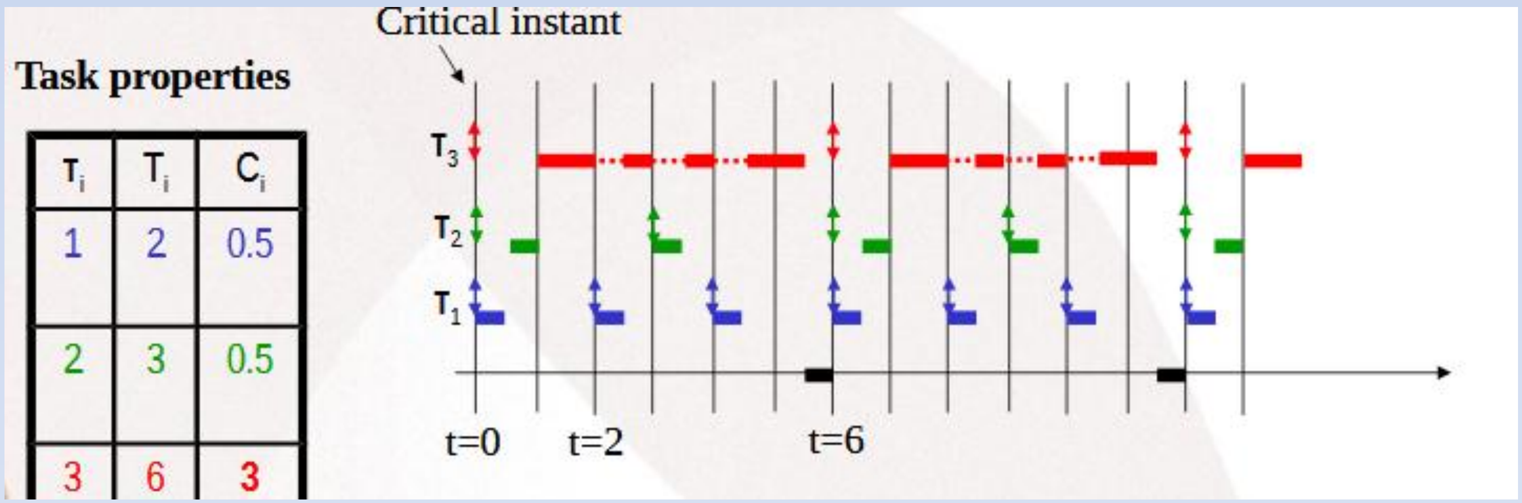
Revisiting task set C

Task	Period	Computation Time	Priority	Response Time
	T	C	P	R
a	80	40	1 (low)	80
b	40	10	2	15
c	20	5	3 (high)	5

- The combined utilization is 1.0
- This is above the utilization threshold for three tasks (0.78) hence the utilization-based schedulability test failed

:

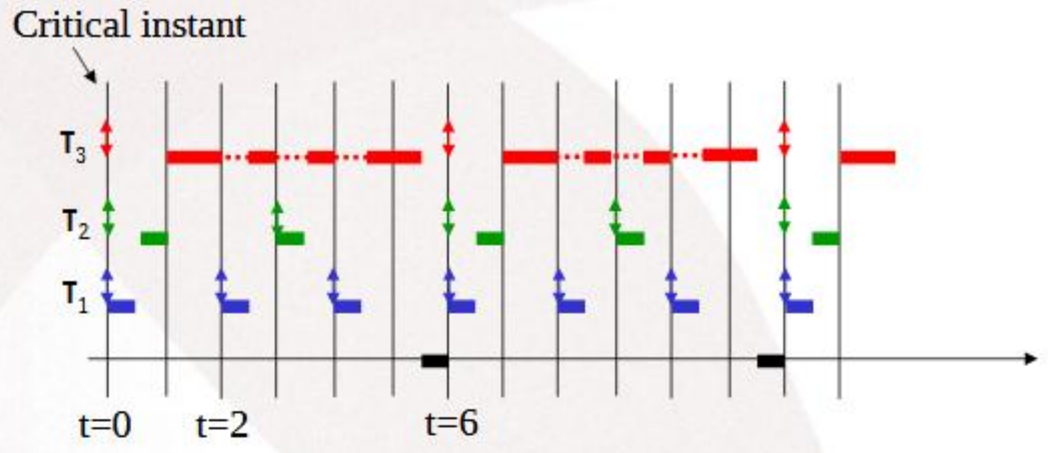
Exercise: Apply RTA (Response Time Analysis)



Task properties

τ_i	T_i	C_i
1	2	0.5
2	3	0.5
3	6	3

Exercise: Apply RTA (Response Time Analysis)



Task properties

τ_i	T_i	C_i
1	2	0.5
2	3	0.5
3	6	**3**

Rwc_1: $Rwc_1(0) = C_1 = 0.5$

Rwc_2: $Rwc_2(0) = C_1 + C_2 = 1$

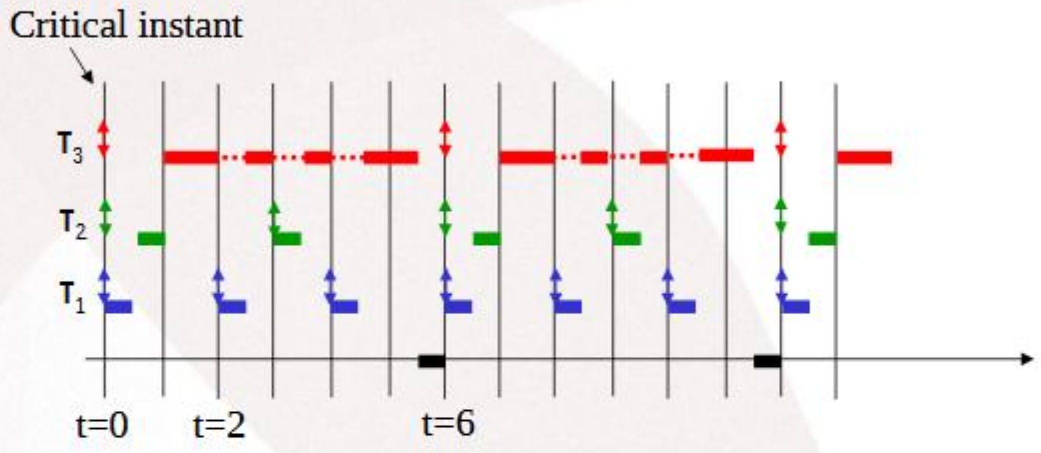
$Rwc_2(1) = \lceil Rwc_2(0)/T_1 \rceil * C_1 + C_2 = 1$

$Rwc_2 = 1$

Exercise: Apply RTA (Response Time Analysis)

Task properties

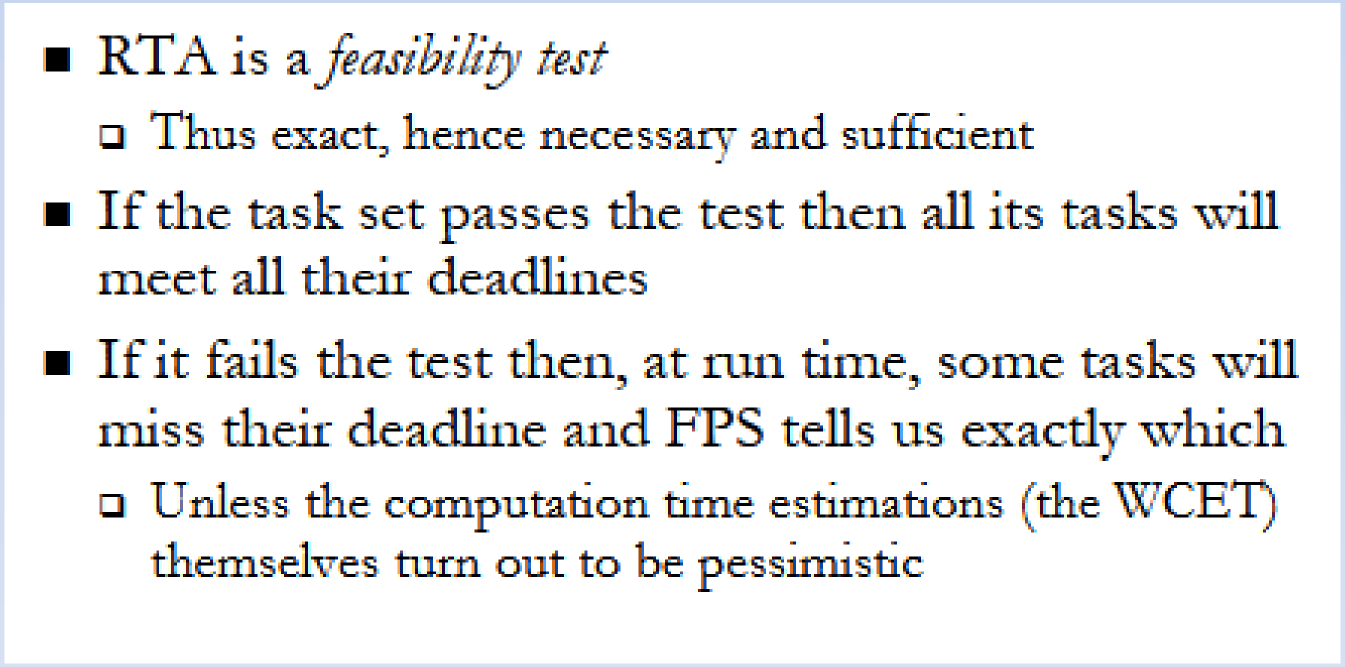
τ_i	T_i	C_i
1	2	0.5
2	3	0.5
3	6	**3**



Rwc_3:

$$Rwc_3(0) = C_1+C_2+C_3 = 4$$
$$Rwc_3(1) = \lceil Rwc_3(0)/T_1 \rceil * C_1 + \lceil Rwc_3(0)/T_2 \rceil * C_2 + C_3 = 5$$
$$Rwc_3(2) = \lceil Rwc_3(1)/T_1 \rceil * C_1 + \lceil Rwc_3(1)/T_2 \rceil * C_2 + C_3 = 5.5$$
$$Rwc_3(3) = \lceil Rwc_3(2)/T_1 \rceil * C_1 + \lceil Rwc_3(2)/T_2 \rceil * C_2 + C_3 = 5.5$$
$$Rwc_3 = 5.5$$

:



- RTA is a *feasibility test*
  - Thus exact, hence necessary and sufficient
- If the task set passes the test then all its tasks will meet all their deadlines
- If it fails the test then, at run time, some tasks will miss their deadline and FPS tells us exactly which
  - Unless the computation time estimations (the WCET) themselves turn out to be pessimistic

:

The previous schedulability tests must be modified in the following cases:

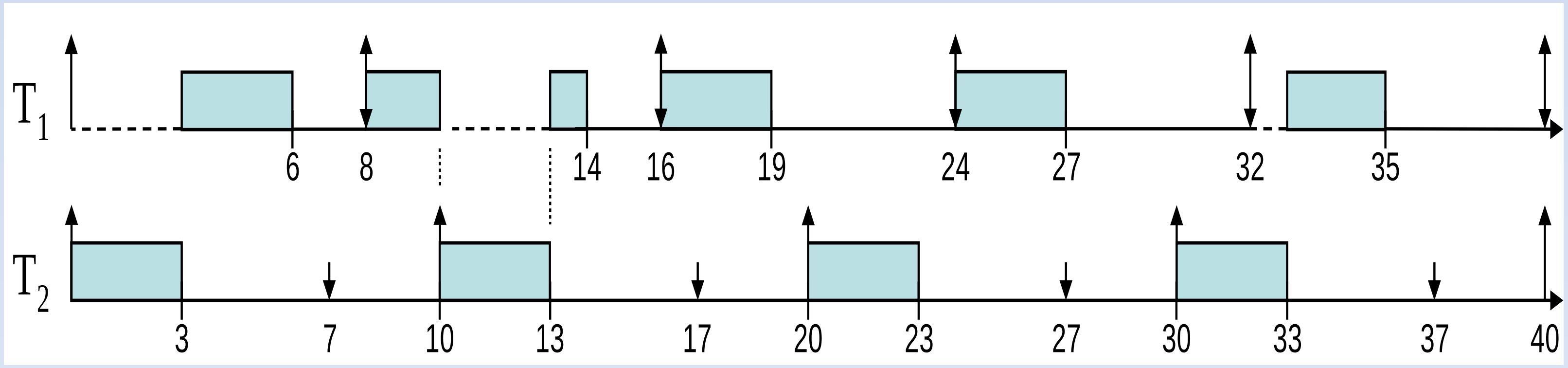
- Non-preemption
- Tasks not independent
- Share mutually exclusive resources
- Have precedence constraints

It is also necessary to take into account the overhead of the kernel,
because the scheduler, dispatcher and interrupts consume CPU time

Deadline Monotonic scheduling

- Priority assignment based on relative deadlines of tasks
- Shorter the relative deadline, higher the priority
- Useful when relative deadline ≠ period

- Both of the above usually done off-line since fixed priority assigned at task level
- On line dispatcher enforces the schedule by dispatching higher priority jobs



Note: The worst case response time of a task occurs when it is released simultaneously with all higher-priority tasks.

DM is optimal among fixed-priority algorithms i.e

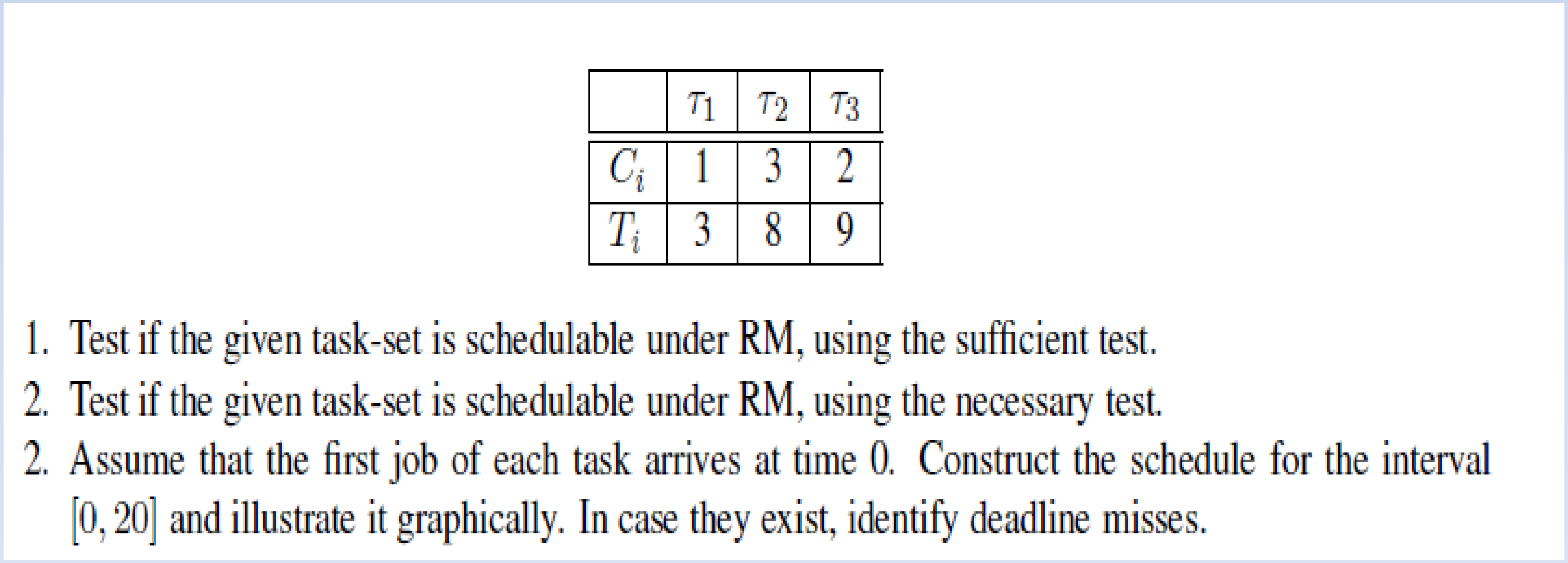
If a task set can be feasibly scheduled by a fixed-priority algorithm then it can be feasibly scheduled by Deadline Monotonic algorithm (DM).

A set of n periodic tasks is feasibly scheduled by DM if

$Ch \leq n\,(2^{1/n} - 1)$

:

Question : Schedulability test with RM

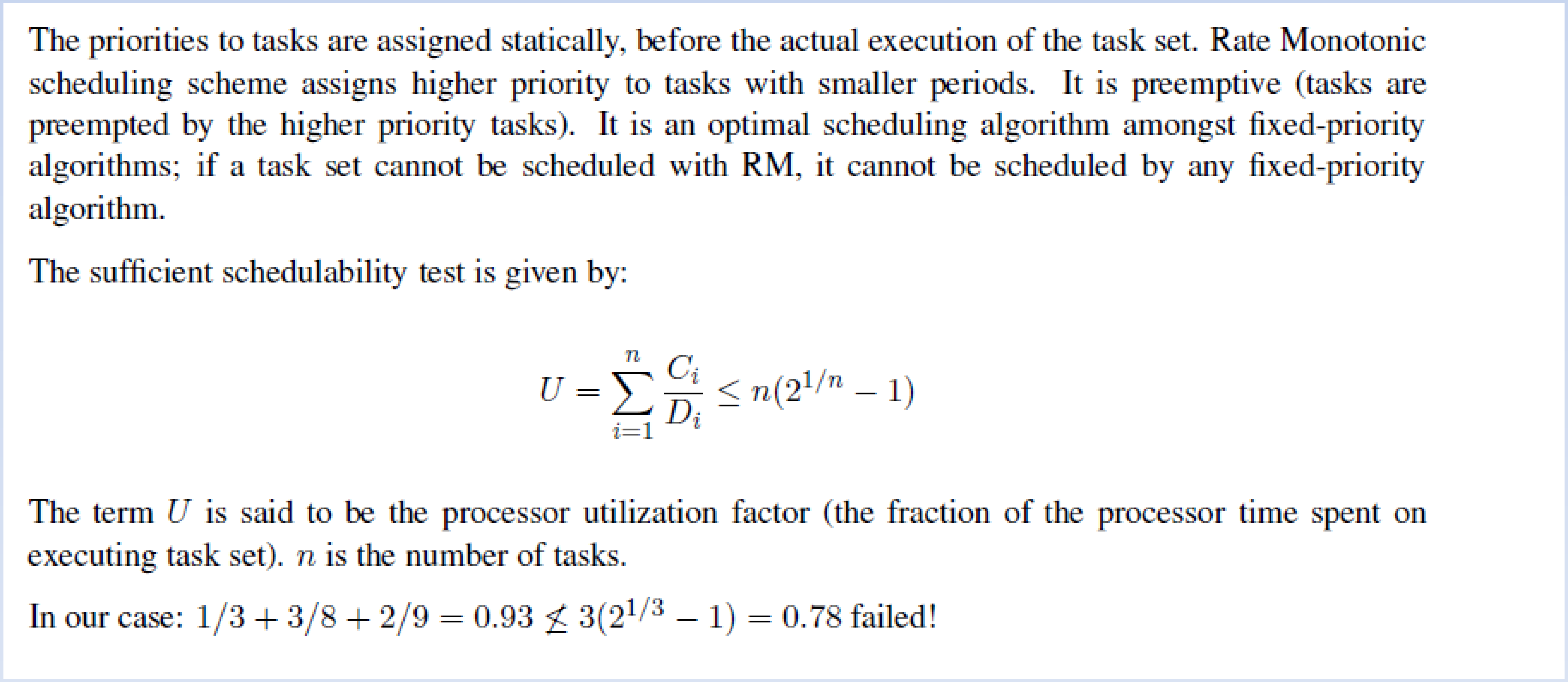


	τ_1	τ_2	τ_3
C_i	1	3	2
T_i	3	8	9

1. Test if the given task-set is schedulable under RM, using the sufficient test.
2. Test if the given task-set is schedulable under RM, using the necessary test.
2. Assume that the first job of each task arrives at time 0. Construct the schedule for the interval $[0, 20]$ and illustrate it graphically. In case they exist, identify deadline misses.

:

Response : Schedulability test with RM



The priorities to tasks are assigned statically, before the actual execution of the task set. Rate Monotonic scheduling scheme assigns higher priority to tasks with smaller periods. It is preemptive (tasks are preempted by the higher priority tasks). It is an optimal scheduling algorithm amongst fixed-priority algorithms; if a task set cannot be scheduled with RM, it cannot be scheduled by any fixed-priority algorithm.

The sufficient schedulability test is given by:

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

The term U is said to be the processor utilization factor (the fraction of the processor time spent on executing task set). n is the number of tasks.

In our case: $1/3 + 3/8 + 2/9 = 0.93 \nleq 3(2^{1/3} - 1) = 0.78$ failed!

Response : Schedulability test with RM

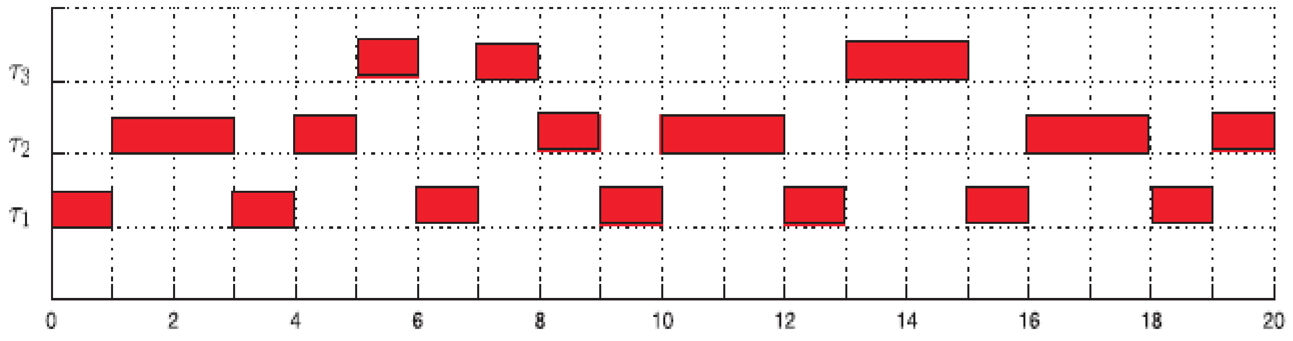
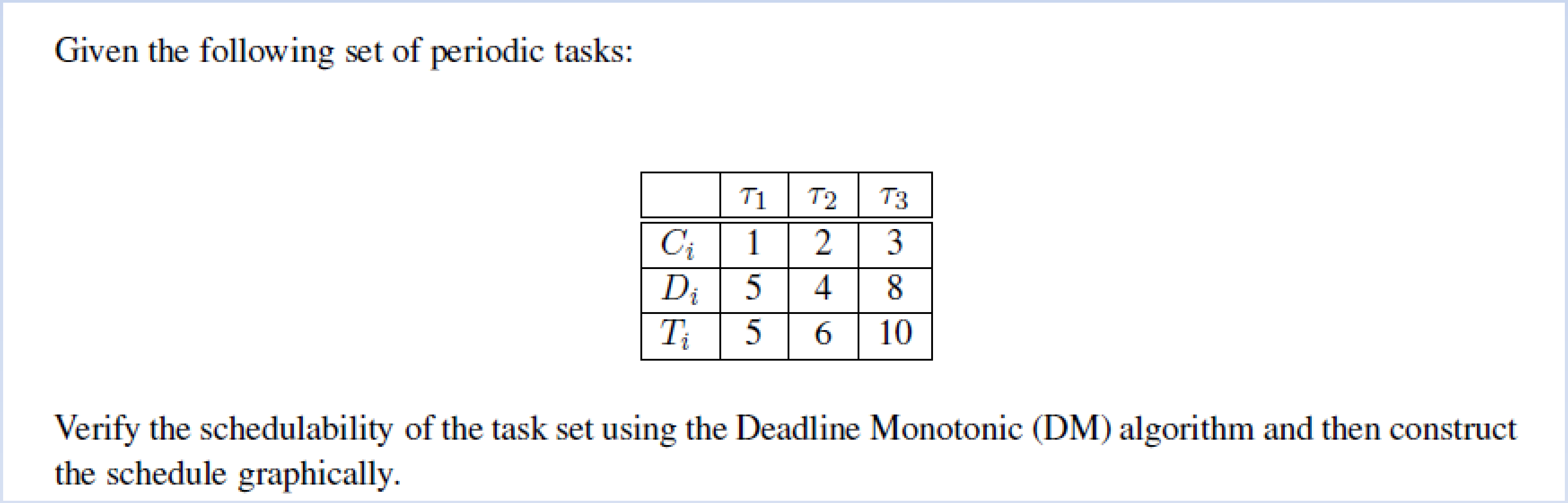


Figure 2: RM schedule.

:

Question : Schedulability test with DM



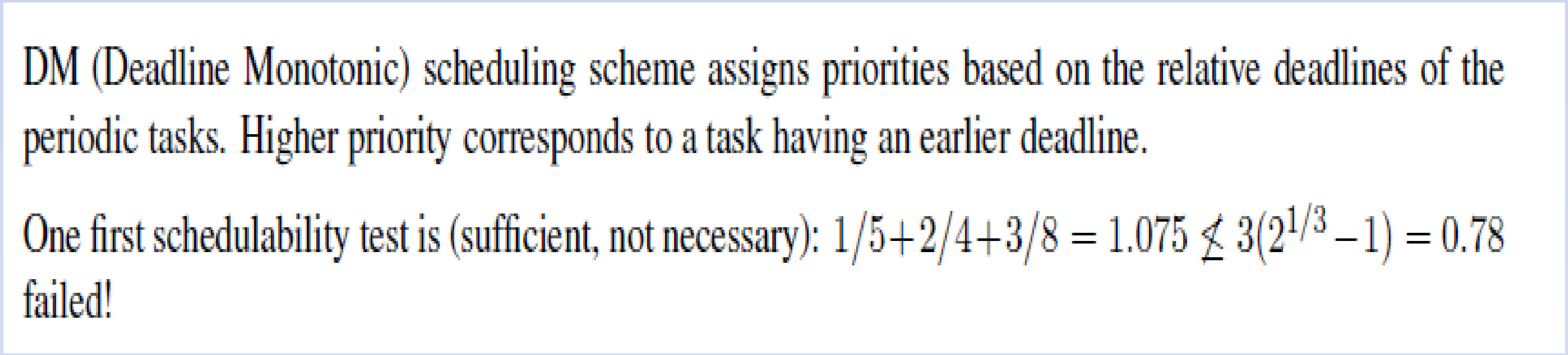
Given the following set of periodic tasks:

	τ_1	τ_2	τ_3
C_i	1	2	3
D_i	5	4	8
T_i	5	6	10

Verify the schedulability of the task set using the Deadline Monotonic (DM) algorithm and then construct the schedule graphically.

:

Response : Schedulability test with DM

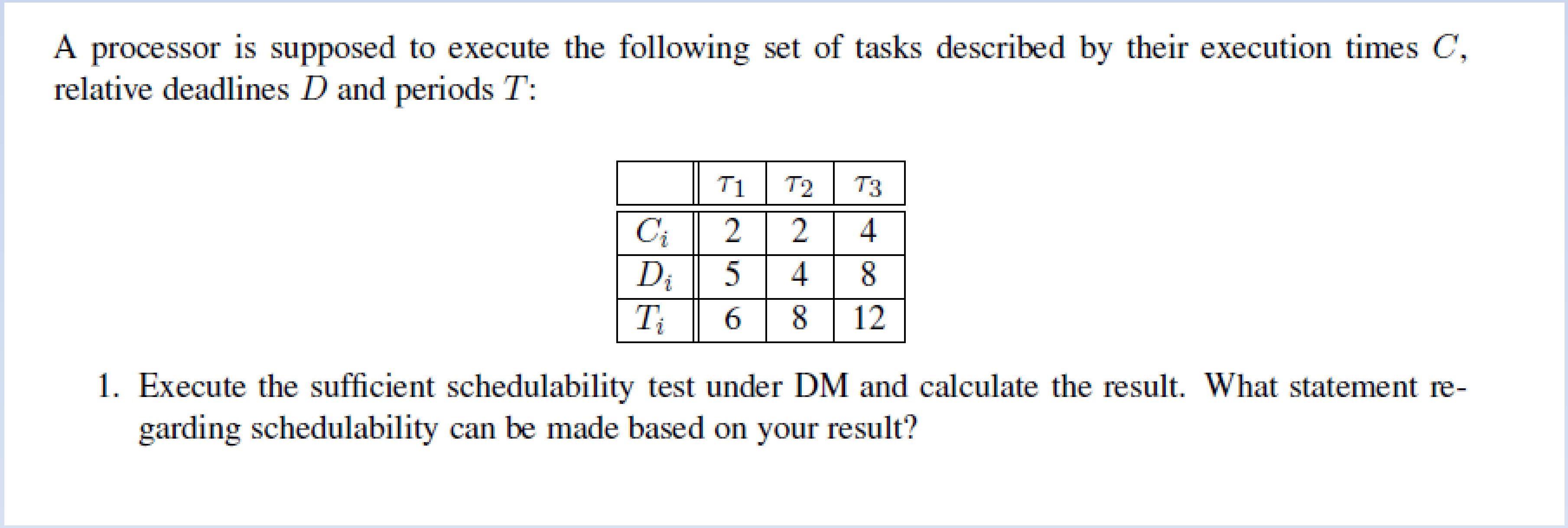


DM (Deadline Monotonic) scheduling scheme assigns priorities based on the relative deadlines of the periodic tasks. Higher priority corresponds to a task having an earlier deadline.

One first schedulability test is (sufficient, not necessary): $1/5+2/4+3/8 = 1.075 \not\leq 3(2^{1/3}-1) = 0.78$ failed!

:

Question : Schedulability test with DM



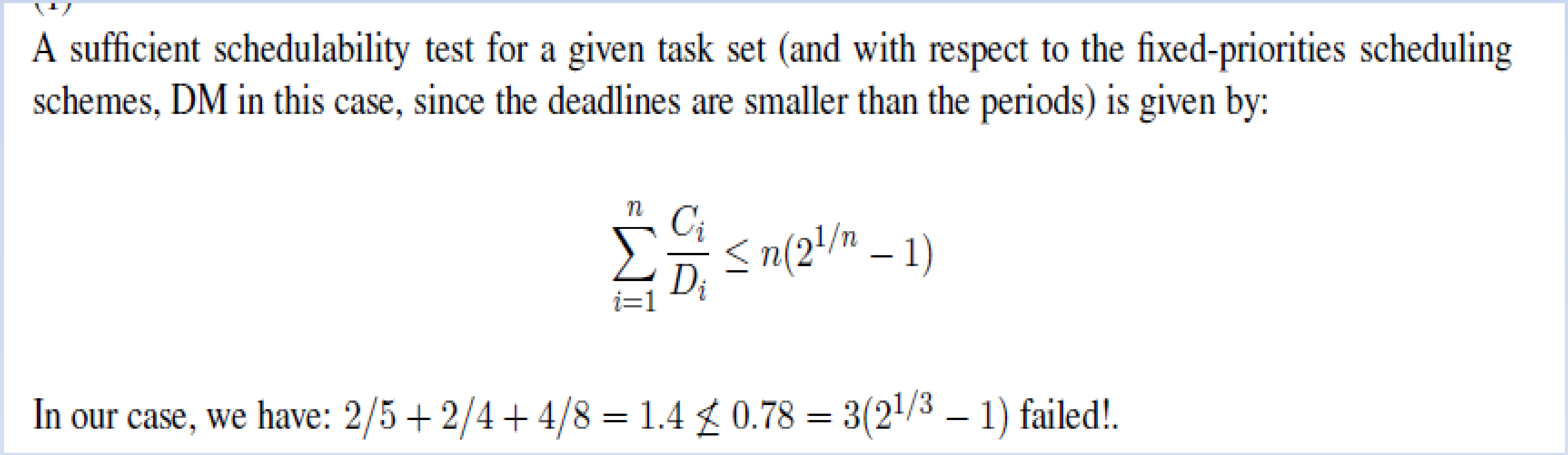
A processor is supposed to execute the following set of tasks described by their execution times C, relative deadlines D and periods T:

	τ_1	τ_2	τ_3
C_i	2	2	4
D_i	5	4	8
T_i	6	8	12

1. Execute the sufficient schedulability test under DM and calculate the result. What statement regarding schedulability can be made based on your result?

:

Response : Schedulability test with DM



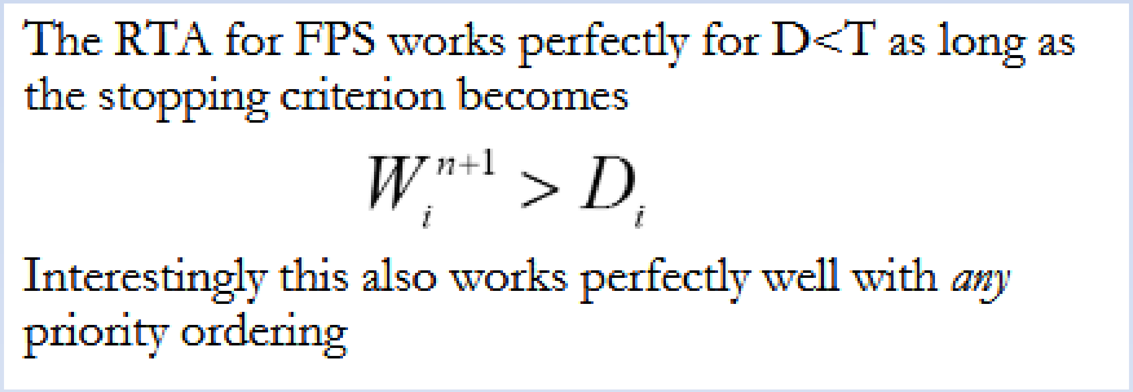
A sufficient schedulability test for a given task set (and with respect to the fixed-priorities scheduling schemes, DM in this case, since the deadlines are smaller than the periods) is given by:

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

In our case, we have: $2/5 + 2/4 + 4/8 = 1.4 \nleq 0.78 = 3(2^{1/3} - 1)$ failed!.

Fixed Priority Preemptive scheduling

Fixed Priority Preemptive scheduling (FPS) is a fixed priority scheduling policy in the sense that a task may be assigned any arbitrary priority.


The RTA for FPS works perfectly for D<T as long as the stopping criterion becomes

$$W_i^{n+1} > D_i$$

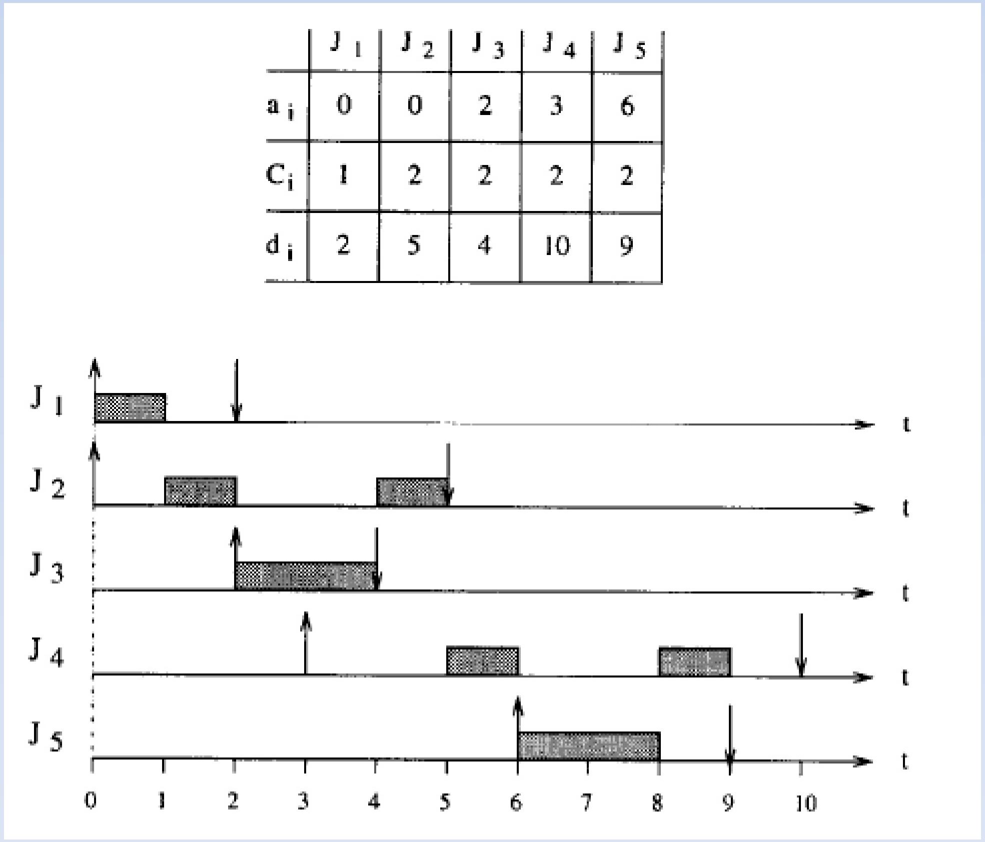
Interestingly this also works perfectly well with *any* priority ordering

:

DYNAMIC PRIORITY ALGORITHMS

Earliest Deadline First (EDF)

Priority assignment based on absolute deadline of jobs
Job with closest deadline assigned highest priority



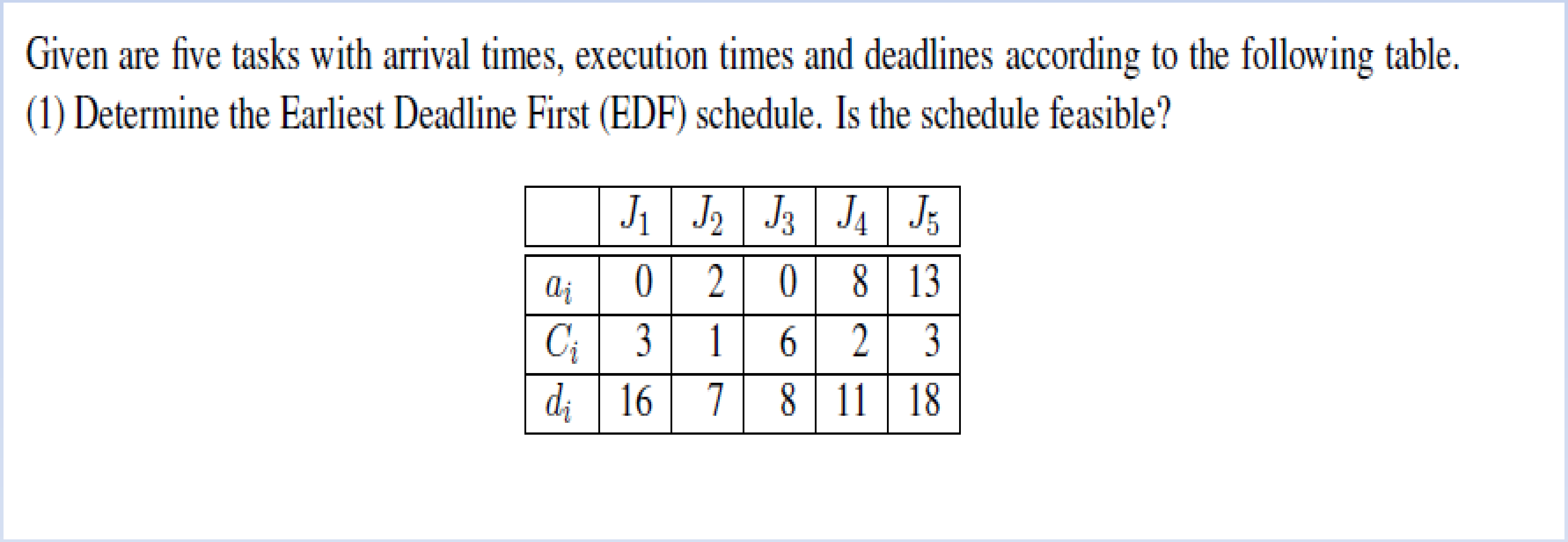
	J_1	J_2	J_3	J_4	J_5
a_i	0	0	2	3	6
C_i	1	2	2	2	2
d_i	2	5	4	10	9

Advantages:

- does not depend on periodicity;
- deadlines less than periods
- can be directly used for periodic + aperiodic tasks

Illustration on non-periodic tasks:

Question: Real-time scheduling with EDF



Given are five tasks with arrival times, execution times and deadlines according to the following table.
(1) Determine the Earliest Deadline First (EDF) schedule. Is the schedule feasible?

	J_1	J_2	J_3	J_4	J_5
a_i	0	2	0	8	13
C_i	3	1	6	2	3
d_i	16	7	8	11	18

Response: Real-time scheduling with EDF

EDF (Earliest Deadline First) is optimal in the sense of feasibility (it minimizes the maximum lateness under following assumptions: scheduling algorithm is preemptive, the tasks are independent, and may have arbitrary arrival times). The *Horn's rule* says that given a set of n independent tasks with arbitrary arrival times any algorithm that at any instant executes the task with earliest absolute deadlines among the ready tasks is optimal with respect to the maximum lateness.

(1) The EDF schedule is feasible, and the respective schedule is shown in the Figure 3.

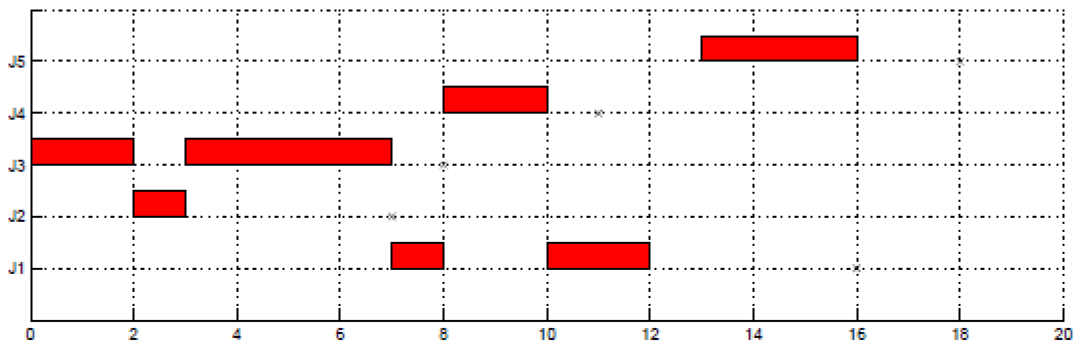
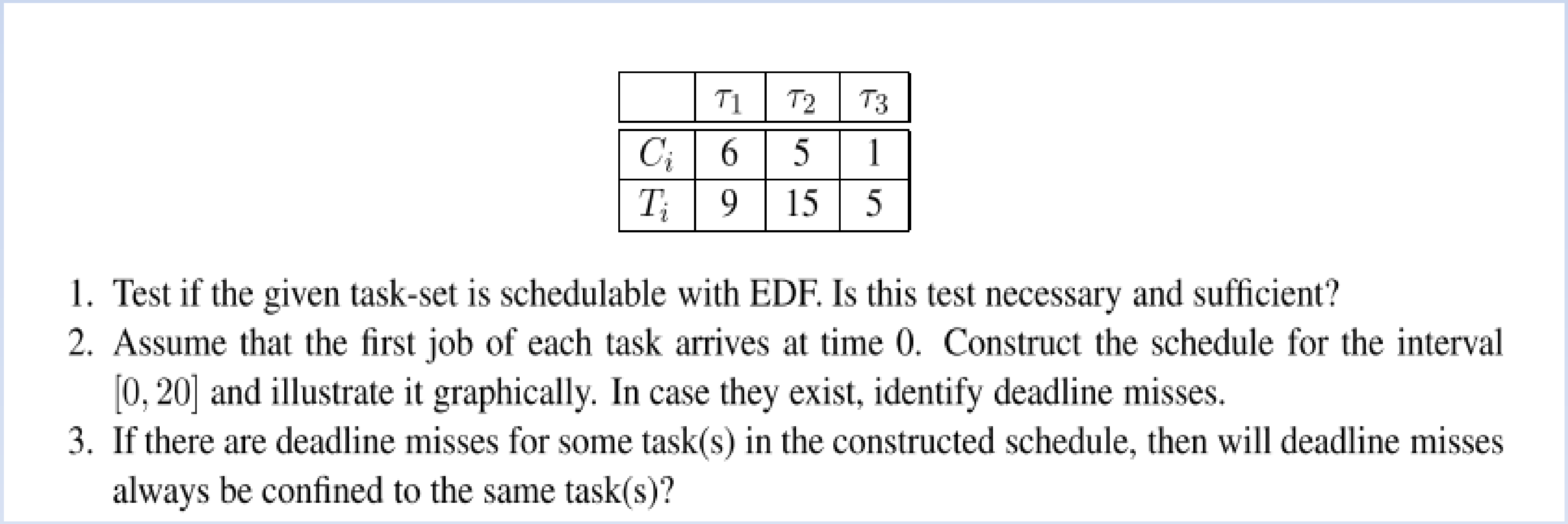


Figure 3: EDF schedule.

:

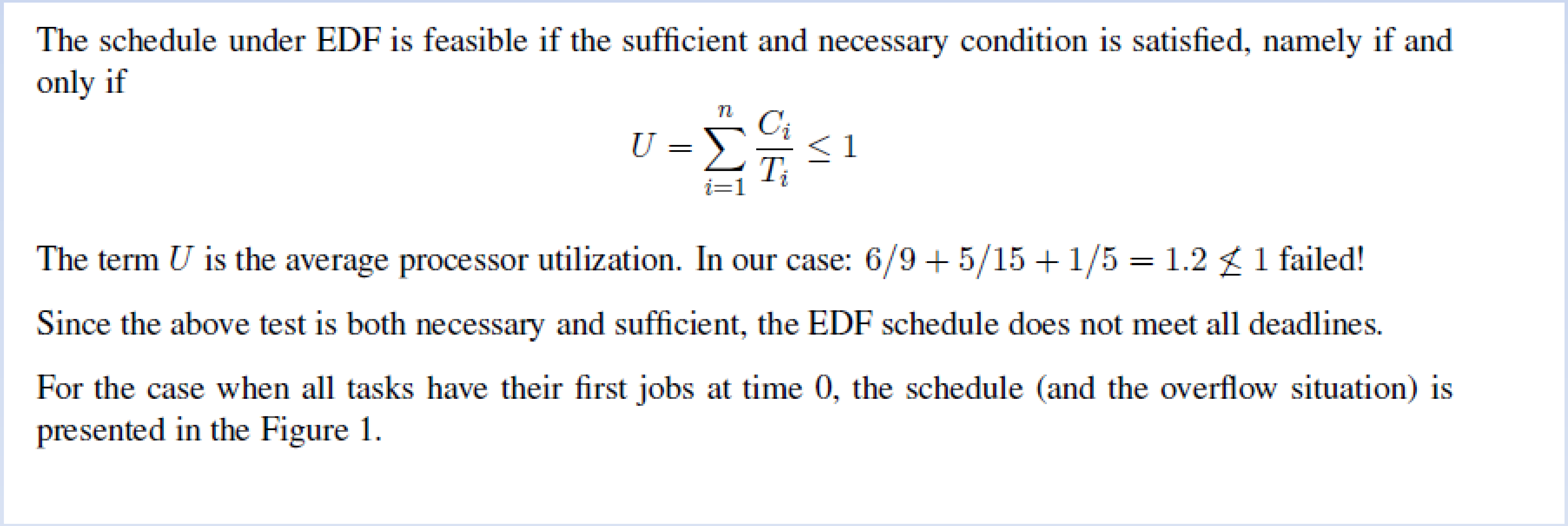
Exercise: Schedulability test with EDF



	τ_1	τ_2	τ_3
C_i	6	5	1
T_i	9	15	5

1. Test if the given task-set is schedulable with EDF. Is this test necessary and sufficient?
2. Assume that the first job of each task arrives at time 0. Construct the schedule for the interval $[0, 20]$ and illustrate it graphically. In case they exist, identify deadline misses.
3. If there are deadline misses for some task(s) in the constructed schedule, then will deadline misses always be confined to the same task(s)?

Response: Schedulability test with EDF



The schedule under EDF is feasible if the sufficient and necessary condition is satisfied, namely if and only if

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

The term U is the average processor utilization. In our case: $6/9 + 5/15 + 1/5 = 1.2 \not\leq 1$ failed!

Since the above test is both necessary and sufficient, the EDF schedule does not meet all deadlines.

For the case when all tasks have their first jobs at time 0, the schedule (and the overflow situation) is presented in the Figure 1.

Response: Schedulability test with EDF

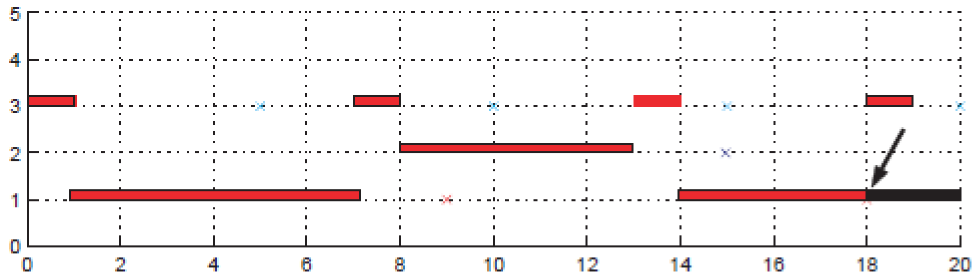


Figure 1: EDF schedule (not feasible).

EDF is an optimal scheduling algorithm: no other algorithm can schedule a set of periodic tasks if the set cannot be scheduled by EDF. Thus, the given task-set is not schedulable!

The schedulability test does not tell us *which* task misses deadlines. So, it is possible that for different arrival times of the jobs, different tasks miss deadlines. In the schedule of Figure 1, τ_1 missed its deadline. But consider the schedule where the first jobs arrive at times 0, 3, and 4, for τ_1, τ_2, τ_3, respectively. In the EDF schedule for $[0, 20]$, we first have a deadline miss for task τ_2.

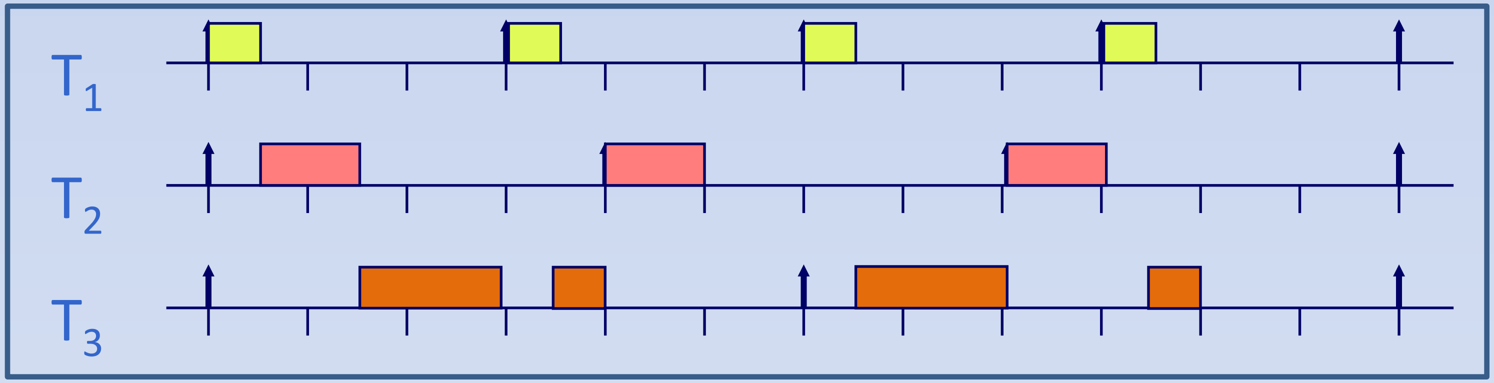
Exercises on scheduling periodic tasks

Three periodic tasks, $T_1 = (0.5,3)$, $T_2 = (1,4)$, $T_3 = (2,6)$

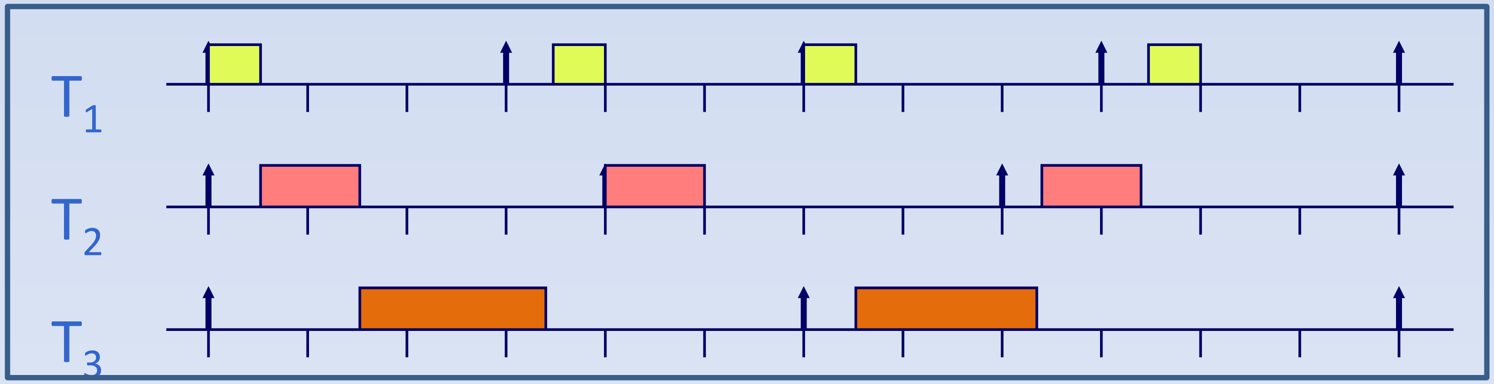
Give the RM schedules and the EDF schedules for this task set

Response

RM



EDF



:

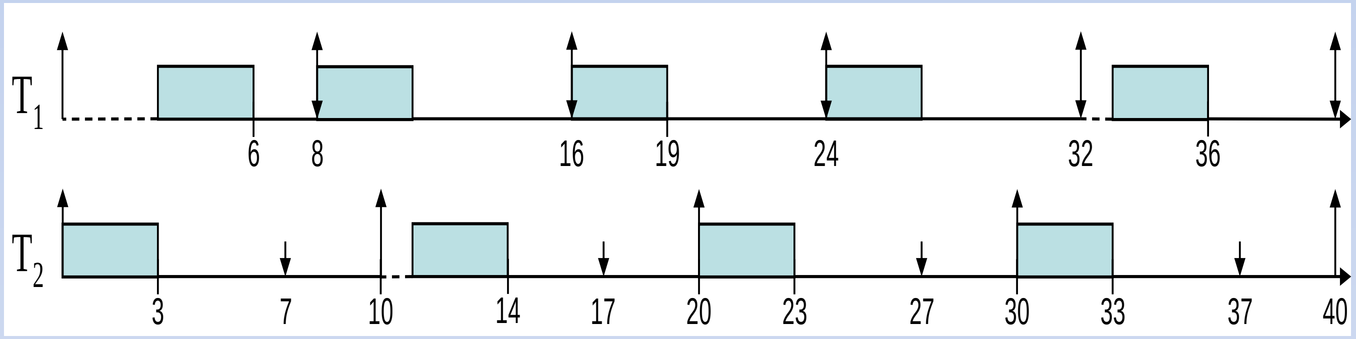


Illustration on periodic tasks:

Earliest Deadline First is an optimal scheduling algorithm

Drawback: It is not possible to know a priori which tasks will fail deadlines

:

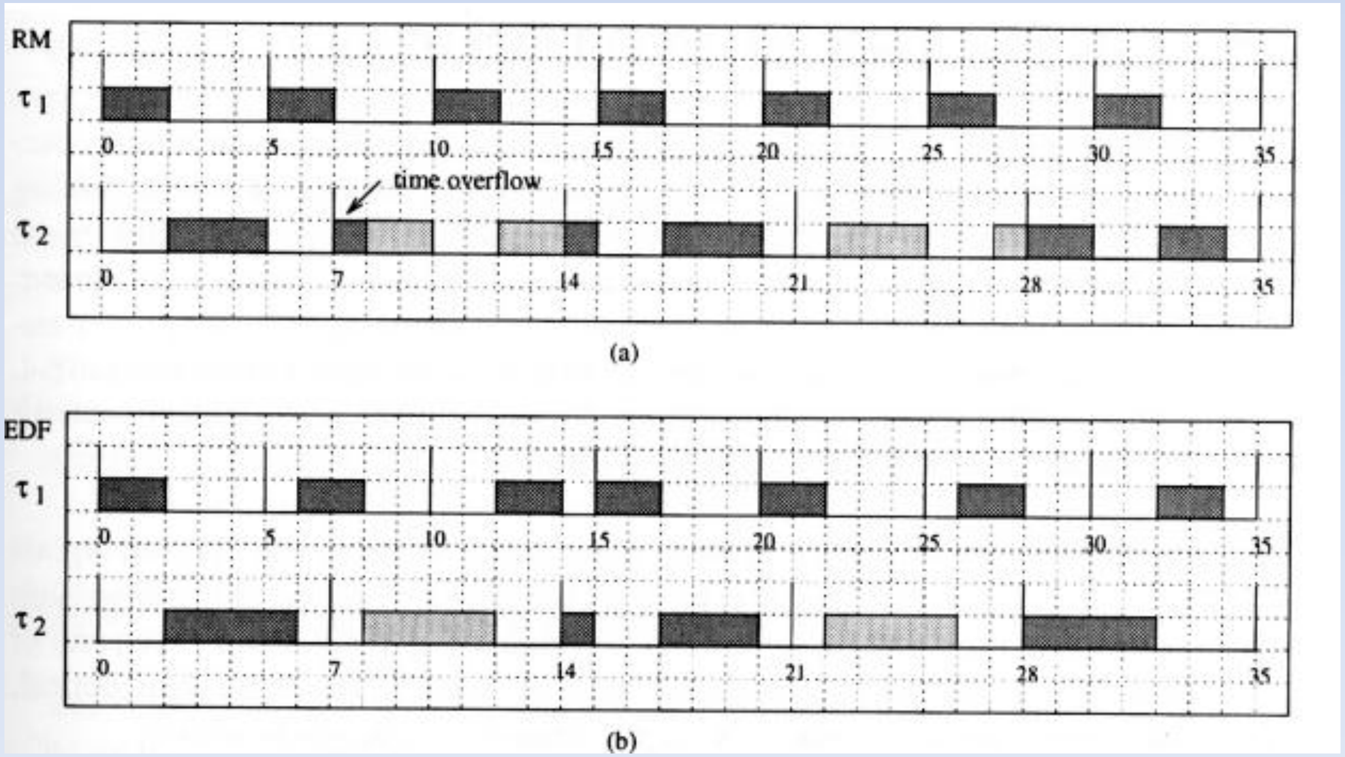
Utilization based test

A set of periodic tasks is schedulable with EDF if $Ch \leq 1$

with $Ch = \sum C_i / D_i$

A set of periodic tasks is schedulable with EDF if and only if $U \leq 1$

with $U = \sum C_i / T_i$



CPU Load based test

For D ≤T, the bigger period during which the CPU is permanently used (i.e. without interruption, idle time) corresponds to the scenario in which all tasks are activated synchronously. This period is called **synchronous busy period** and has duration L

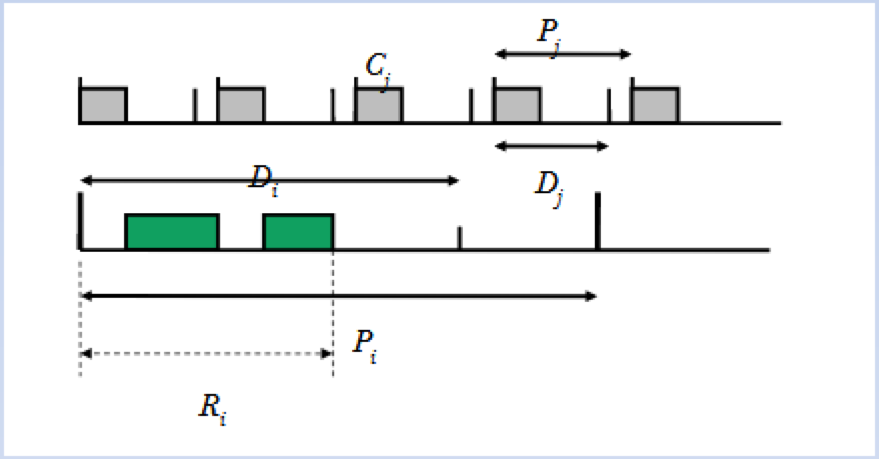
L can be computed by the following iterative method, which returns the first instant since the synchronous activation in which the CPU completes all the submitted jobs

$$L(0)=\sum_i C_i$$

$$L(m+1)=\sum_i \left(\left\lceil \frac{L(m)}{T_i} \right\rceil * C_i\right)$$

:

CPU Load based (Processor demand) test



Here, Interference is due to only those tasks with earlier deadlines

Knowing L, we have to guarantee the load condition :

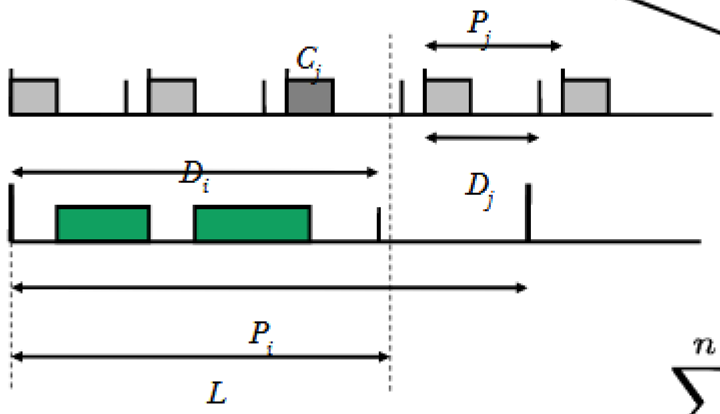
$$h(t) \leq t, \forall_{t \in [0,L]} \Rightarrow \text{Task ser is schedulable (synchronous activations)}$$

In which h(t) is the load function

$$h(t) = \sum_{i=1..n} max(0, 1 + \lfloor \frac{(t - D_i)}{T_i} \rfloor) * C_i$$

CPU Load based (Processor demand) test

- Consider demand on the processor due to tasks whose deadlines have passed
- Within any time interval, *L*, the demand must be less than *L*.



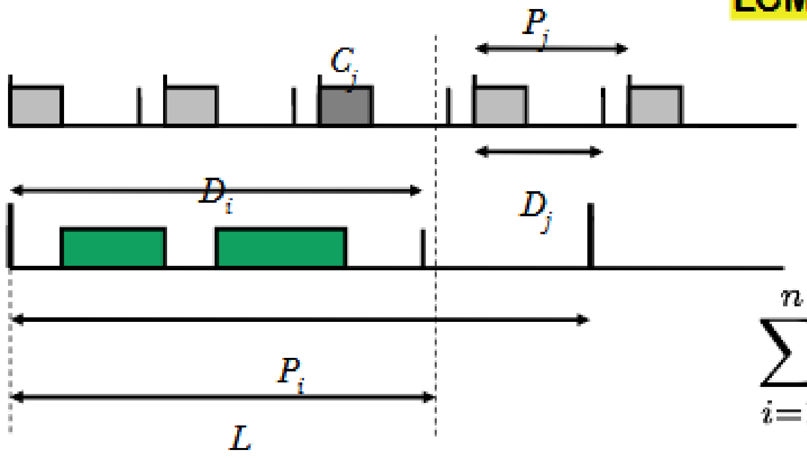
The execution time demanded by jobs with deadline less than ***L*** over an interval of length ***L*** cannot be greater than ***L***

$$\sum_{i=1}^{n} \left\lfloor \frac{L + P_i - D_i}{P_i} \right\rfloor C_i \leq L$$

CPU Load based (Processor demand) test

- Observation 1: Sufficient to check up to the hyperperiod (schedule repeats itself)
- Observation 2: Check only at absolute deadlines

LCM of all task periods

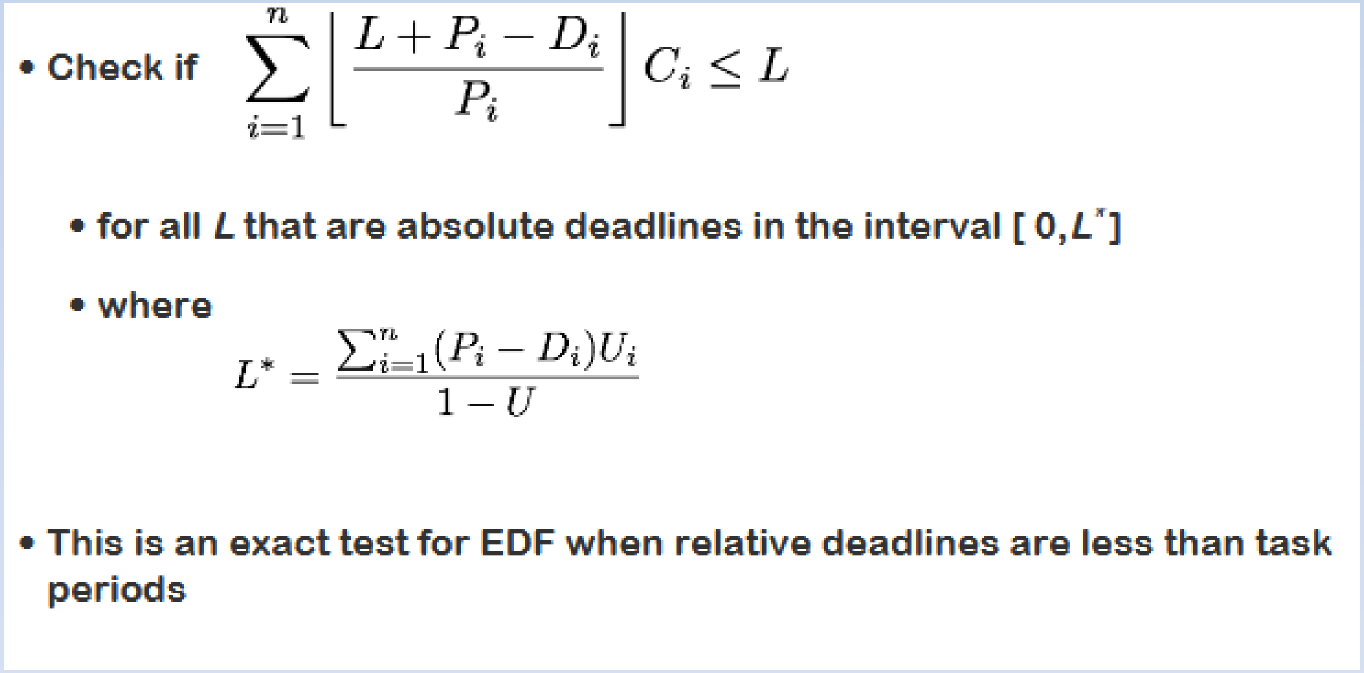


$$\sum_{i=1}^{n} \left\lfloor \frac{L + P_i - D_i}{P_i} \right\rfloor C_i \leq L$$

5

:

CPU Load based (Processor demand) test



- Check if $$\sum_{i=1}^{n} \left\lfloor \frac{L + P_i - D_i}{P_i} \right\rfloor C_i \leq L$$
    - for all *L* that are absolute deadlines in the interval [0,L^*]
    - where
$$L^* = \frac{\sum_{i=1}^{n} (P_i - D_i) U_i}{1 - U}$$
- This is an exact test for EDF when relative deadlines are less than task periods

:

Example using processor demand

- Consider the following task set
  - T_1: (C_1=1, P_1=3, D_1=2)
  - T_2: (C_2=2, P_2=7, D_2=5.5)
  - T_3: (C_3=2, P_3=10, D_3=6)
- The task set has a hyperperiod of 210
- However, we only need to test deadlines up to $L^* = \frac{\sum_{i=1}^{n}(P_i - D_i)U_i}{1-U}$
- *U* = 86/105 = 0.8190; *L** = 8.63
- At *L=2*:

$$\lfloor \frac{L + P_1 - D_1}{P_1} \rfloor C_1 = \lfloor \frac{2+3-2}{3} \rfloor 1 = 1$$

- At *L=5.5*: (also need to check at *L=5*; not shown here)

$$\lfloor \frac{L + P_1 - D_1}{P_1} \rfloor C_1 + \lfloor \frac{L + P_2 - D_2}{P_2} \rfloor C_2 = \lfloor \frac{5.5+3-2}{3} \rfloor 1 + \lfloor \frac{5.5+7-5.5}{7} \rfloor 2 = 2+2=4$$

Example using processor demand

- Consider the following task set
  - T_1: (C_1=1, P_1=3, D_1=2)
  - T_2: (C_2=2, P_2=7, D_2=5.5)
  - T_3: (C_3=2, P_3=10, D_3=6)
- The task set has a hyperperiod of 210
- However, we only need to test deadlines up to $L^* = \frac{\sum_{i=1}^{n}(P_i - D_i)U_i}{1-U}$
- *U* = 86/105 = 0.8190; *L** = 8.63
- At *L=2*:

$$\lfloor\frac{L+P_1-D_1}{P_1}\rfloor C_1 = \lfloor\frac{2+3-2}{3}\rfloor 1 = 1$$

- At *L=5.5*: (also need to check at *L=5*; not shown here)

$$\lfloor\frac{L+P_1-D_1}{P_1}\rfloor C_1 + \lfloor\frac{L+P_2-D_2}{P_2}\rfloor C_2 = \lfloor\frac{5.5+3-2}{3}\rfloor 1 + \lfloor\frac{5.5+7-5.5}{7}\rfloor 2 = 2+2=4$$

:

- At *L=6*:

$$\lfloor\frac{6+3-2}{3}\rfloor 1+\lfloor\frac{6+7-5.5}{7}\rfloor 2+\lfloor\frac{6+10-6}{10}\rfloor 2=6$$

- At *L=8*:

$$\lfloor\frac{8+3-2}{3}\rfloor 1+\lfloor\frac{8+7-5.5}{7}\rfloor 2+\lfloor\frac{8+10-6}{10}\rfloor 2=3+2+2=7$$

No more absolute deadlines < 8.63. We are done!

:

EDF VS. DM

Similarities:

- Both algorithms are optimal within their class
- Both are easy to implement in terms of priority queues
- Both have simple utilization-based schedulability tests
- Both can be extended in similar ways (next lecture)

Advantages of EDF:

- close relation to terminology of real-time specifications,
- directly applicable to sporadic, interrupt-driven tasks
- greater CPU utilization

Drawbacks of EDF:

- It exhibits random behavior under transient overload (but so does RM, in f act, in a different way). DM predictably skips low priority tasks under constant Overload
- Operating systems generally don't support it

:

Least Laxity First (LLF)

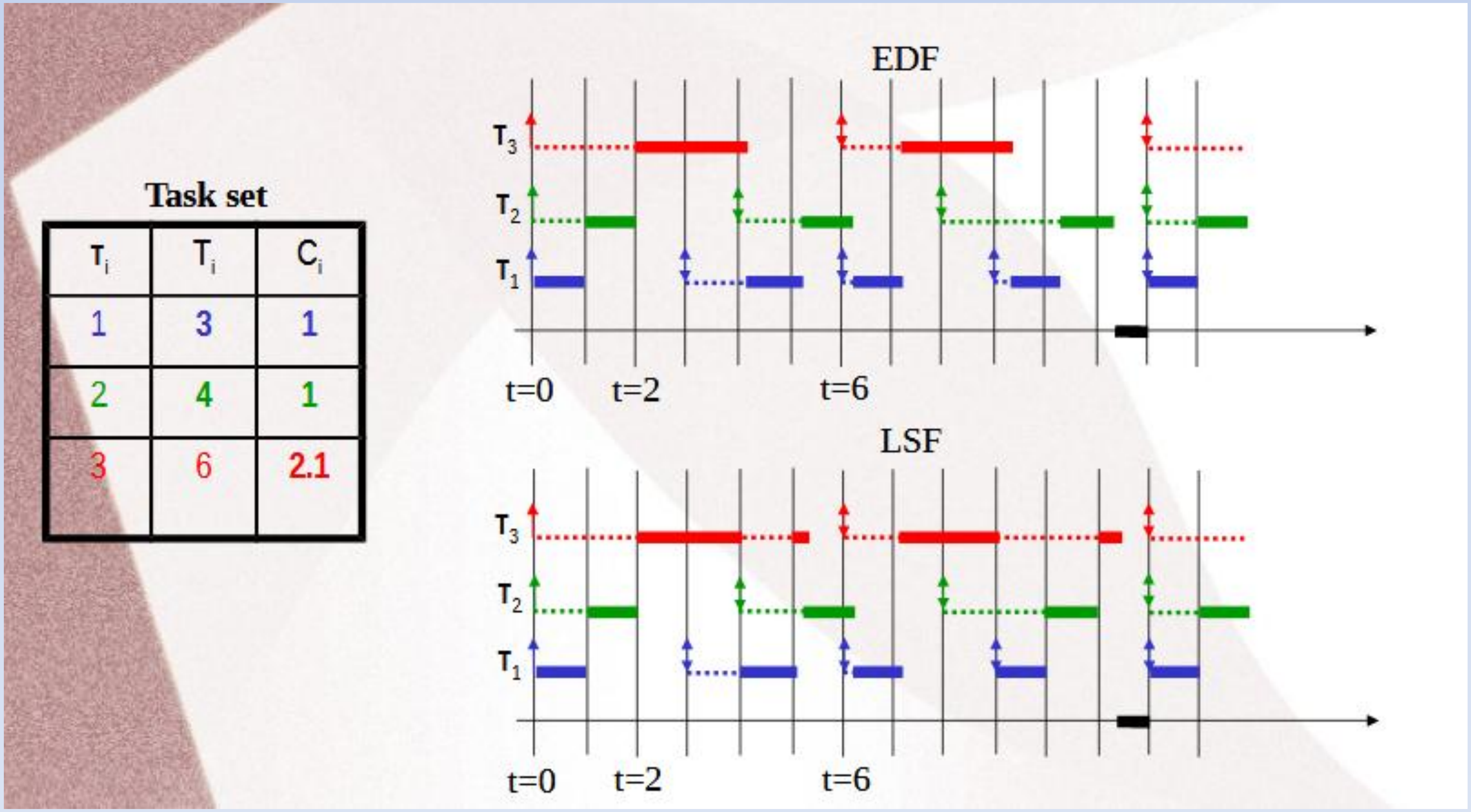
- Laxity = Absolute Deadline – Worst case computation time
- Priority assignment based on laxity of jobs
- Job with minimum laxity assigned highest priority
- Schedulable utilization = 1

Remarks:

- Dynamic Priority algorithms EDF and LLF provide better processor utilization than Fixed Priority algorithms
- Identical schedulability checks for EDF and LLF
- Prohibitive overheads with LLF

Least Laxity First is an optimal scheduling algorithm

:



Task set

τ_i	T_i	C_i
1	3	1
2	4	1
3	6	2.1

:

HANDLING SHARED RESOURCES

- Resource - something needed to advance execution of a task
- shared resource - resource used by several tasks
- mutually exclusive resource - shared resource that can be used by only one task at a time
- critical section (CS) - piece of code executed under mutual exclusion constraint

- Typical example: semaphore (wait : to lock, signal : to unlock)

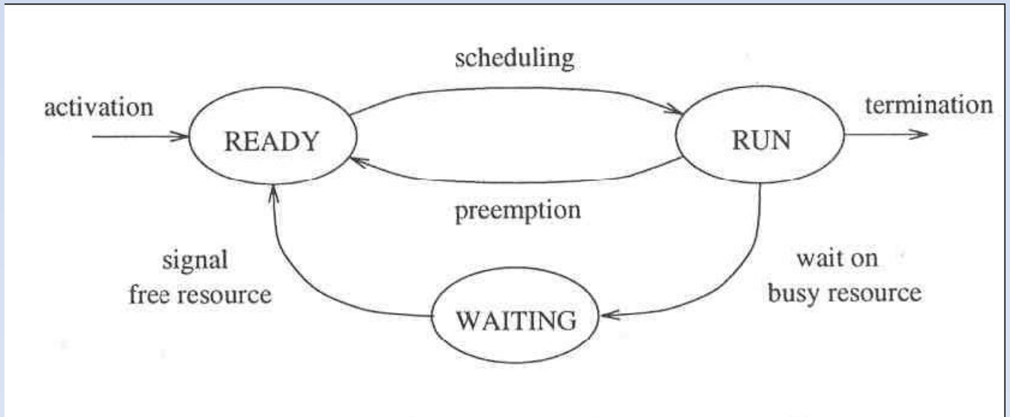
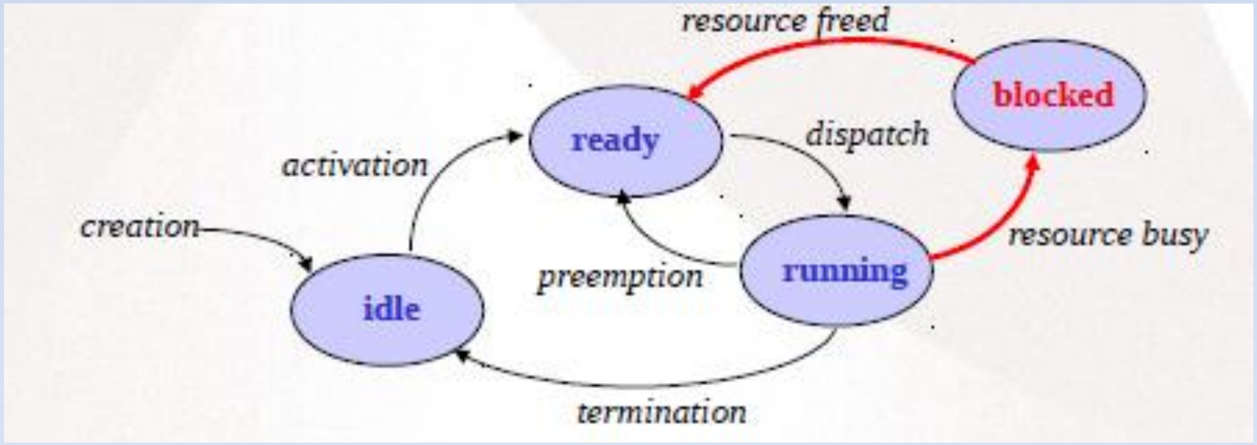


Figure 2.11 Waiting state caused by resource constraints.

Examples of common resources: data structures, variables, main memory area, file, set of registers, I/O unit, … .

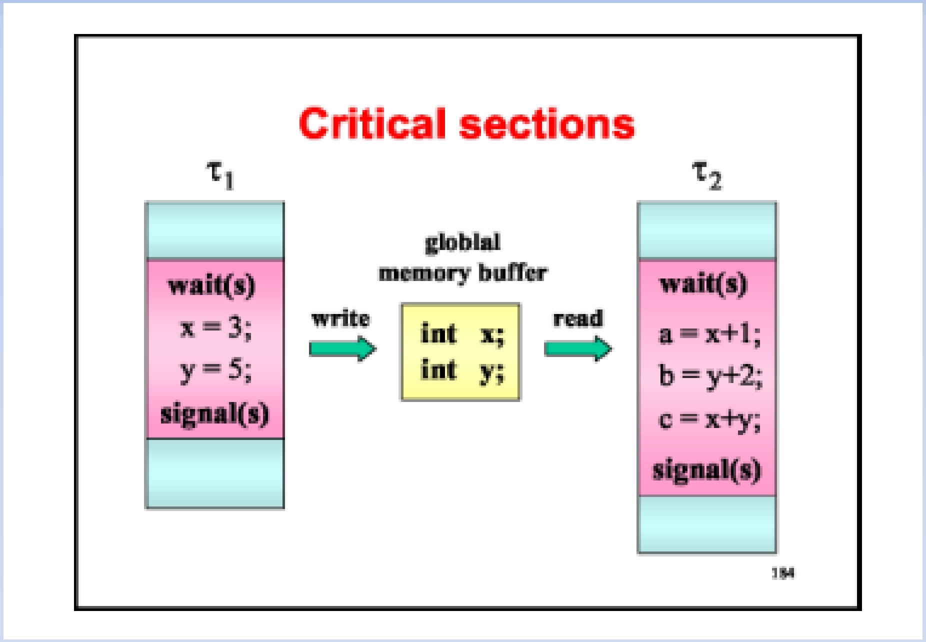
:

When a running task tries to access a shared resource (e.g. a buffer, a communication port) that is already taken (i.e. in use) by another task, the first one is **blocked**.
When the resource becomes free, the blocked task becomes again ready for execution. To handle this scenario the state diagram is updated as follows:



Examples of common resources: data structures, variables, main memory area, file, set of registers, I/O unit, … .

:



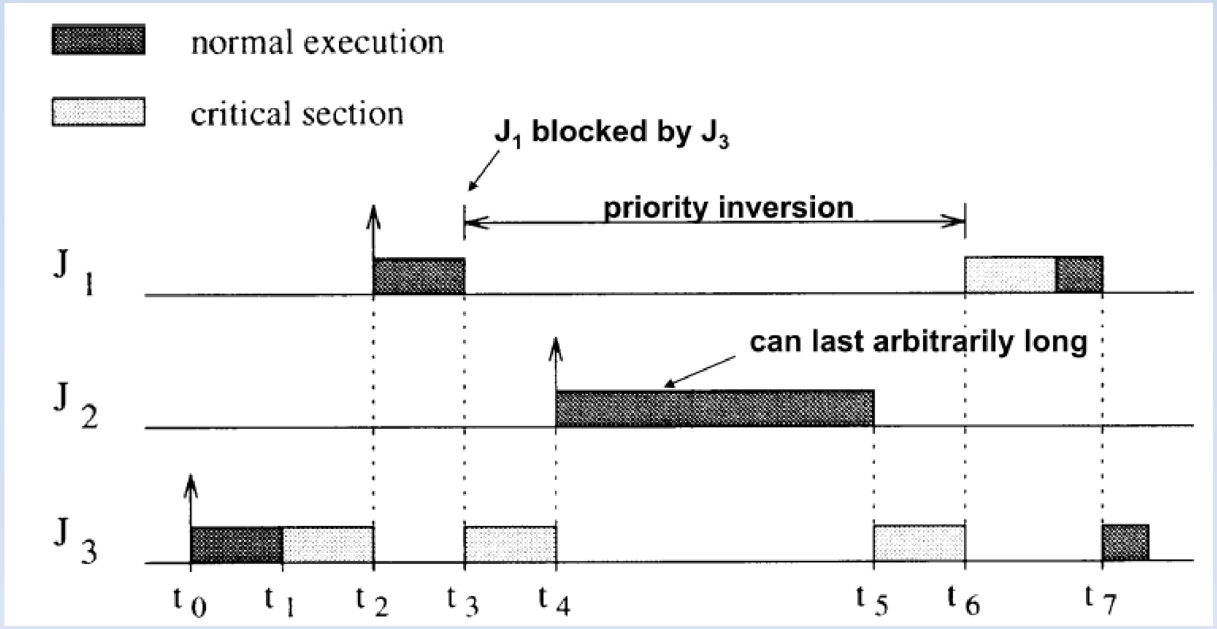
A task waiting for an exclusive resource is said to be **blocked** on that resource. Otherwise, it proceeds by entering the critical section and holds the resource. When a task leaves a critical section, the associated resource becomes free.

Each exclusive resource Ri must be protected by a different semaphore Si and each critical section operating on a resource must begin with a wait(Si) primitive and end with a signal(Si) primitive.

All tasks blocked on the same resource are kept in a queue associated with the semaphore. When a running task executes a wait on a locked semaphore, it enters a waiting state, until another tasks executes a signal primitive that unlocks the semaphore.

Priority inversion

A high priority task is blocked by a lower priority task for an unbounded interval of time.



:

Solution: Introduce a concurrency control protocol for accessing critical sections called

Resource Access Protocols

Under fixed priorities

- Non Preemptive Protocol (NPP)
- Highest Locker Priority (HLP)
- Priority Inheritance Protocol (PIP)
- Priority Ceiling Protocol (PCP)

Under EDF

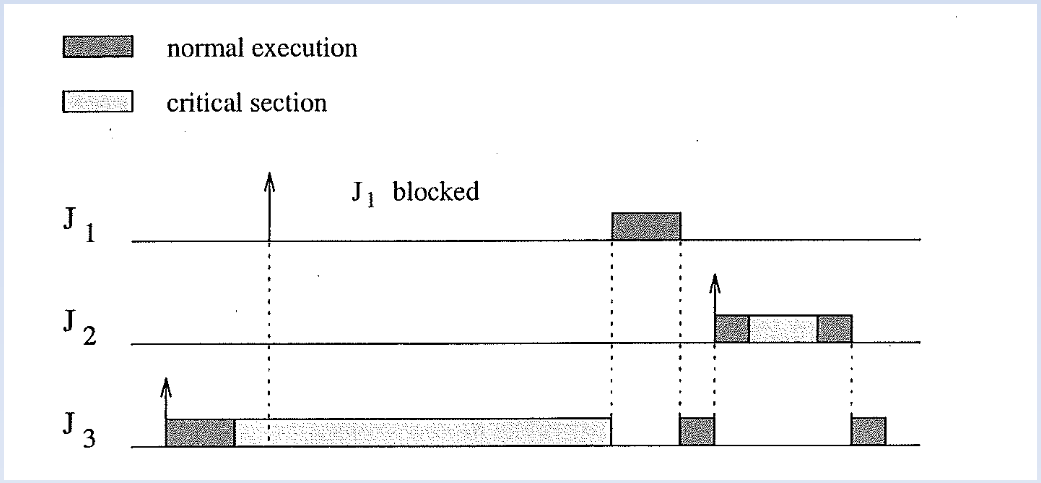
- Stack Resource Policy (SRP)

NON PREEMPTIVE PROTOCOL

•Preemption is forbidden in critical sections.
•Implementation: when a task enters a CS, its priority is increased at the maximum value.

Advantages:
•simplicity,
• no semaphore needed

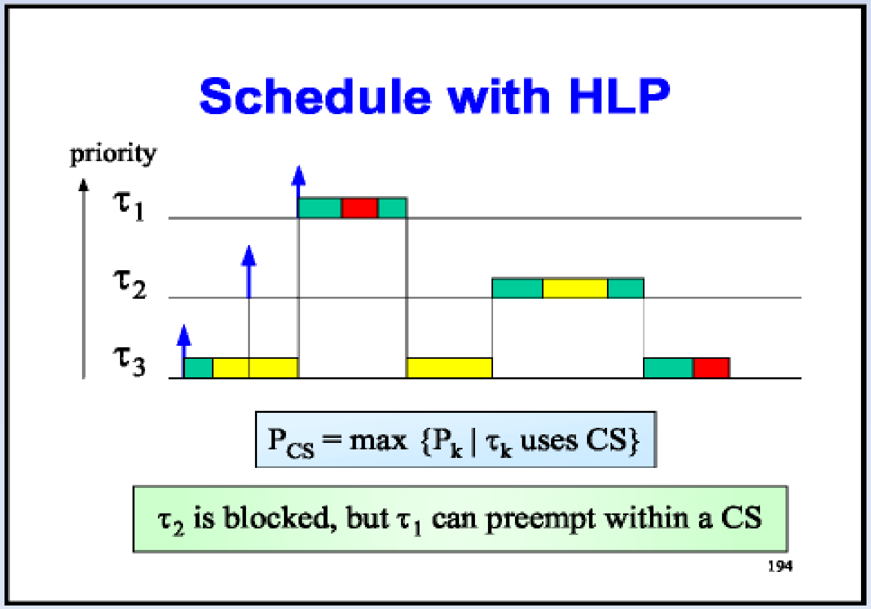
Problems: high priority tasks that do not use CS may also block



HIGHEST LOCKER PRIORITY

A task in a CS gets the highest priority among the tasks that use it.

Features:
•Simple implementation.
•A task is blocked when attempting to preempt, not when entering the CS.



$P_{CS} = \max \{P_k \mid \tau_k \text{ uses CS}\}$

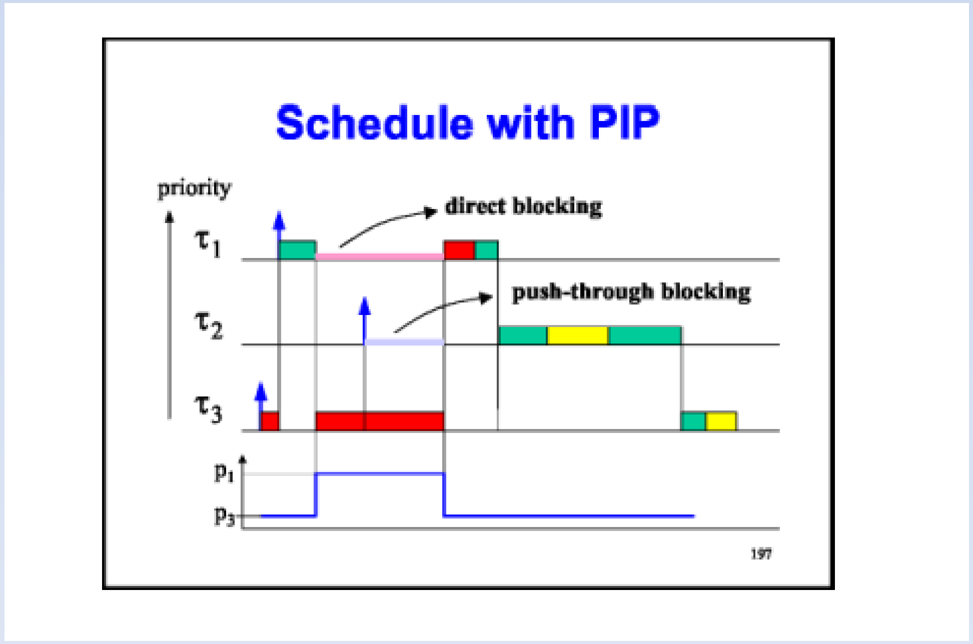
τ_2 is blocked, but τ_1 can preempt within a CS

PRIORITY INHERITANCE PROTOCOL

•The Priority Inheritance Protocol (PIP) has been introduced in 1990.

•A task in a CS increases its priority only if it blocks other tasks.

•A task in a CS inherits the highest priority among those tasks it blocks.



:

Advantage:
Relatively easy to implement because One additional field on the TCB, the inherited priority

Blocking: a delay caused by a lower priority task

Types of blocking:

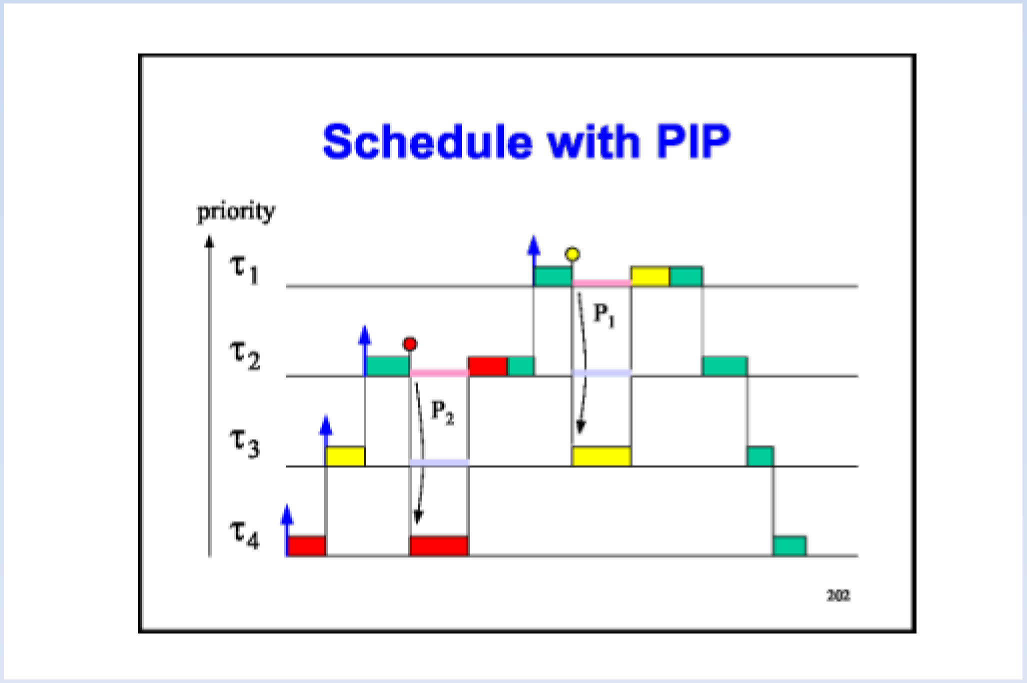
- **Direct blocking**
  A task blocks on a locked semaphore
- **Push-through blocking (chain blocking)**
  A task blocks because a lower priority task inherited a higher priority.

A task τ_i can be blocked by those semaphores used by lower priority tasks and

- directly shared with T_i (direct blocking) or
- shared with tasks having priority higher than T_i (push-through blocking).

Bounding blocking times:

If n is the number of tasks with priority less than T_i and m is the number of semaphores on which T_i can be blocked, then T_i can be blocked at most during min(n,m) critical sections.



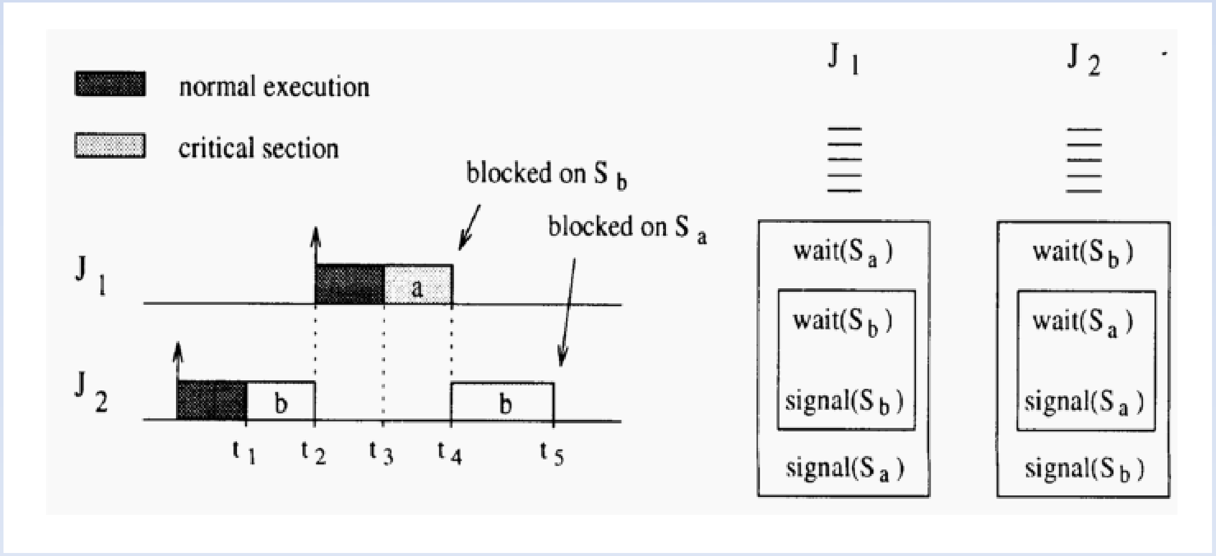
Remarks on PIP:

Advantages:

- It is transparent to the programmer.
- It bounds priority inversion.

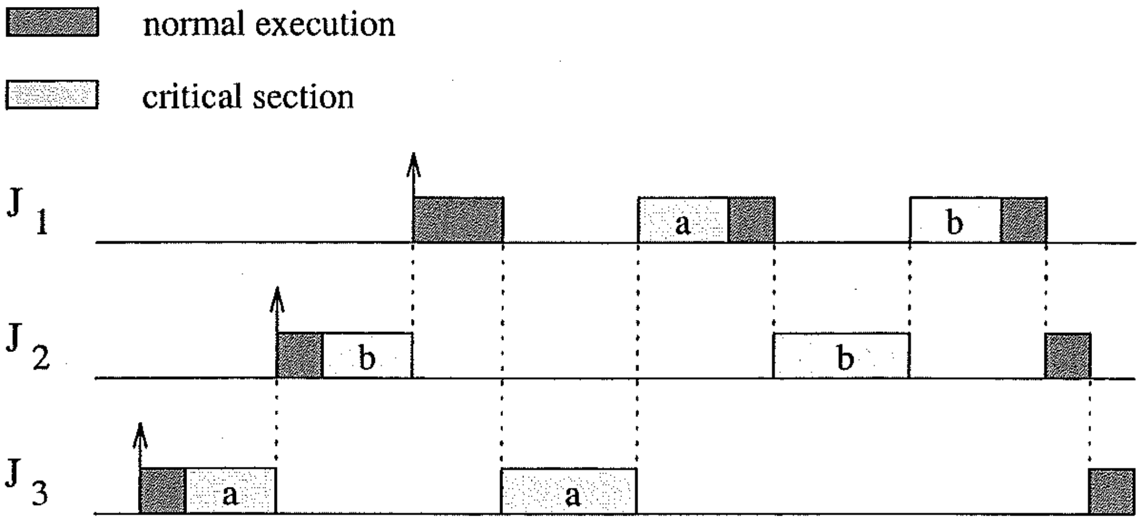
Problems:

- It does not avoid **deadlocks**



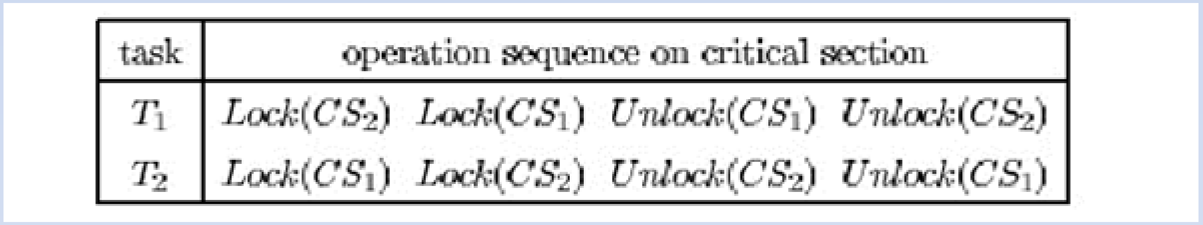
Problems:

- It does not avoid **chain blocking**



Exercise on deadlock with PIP

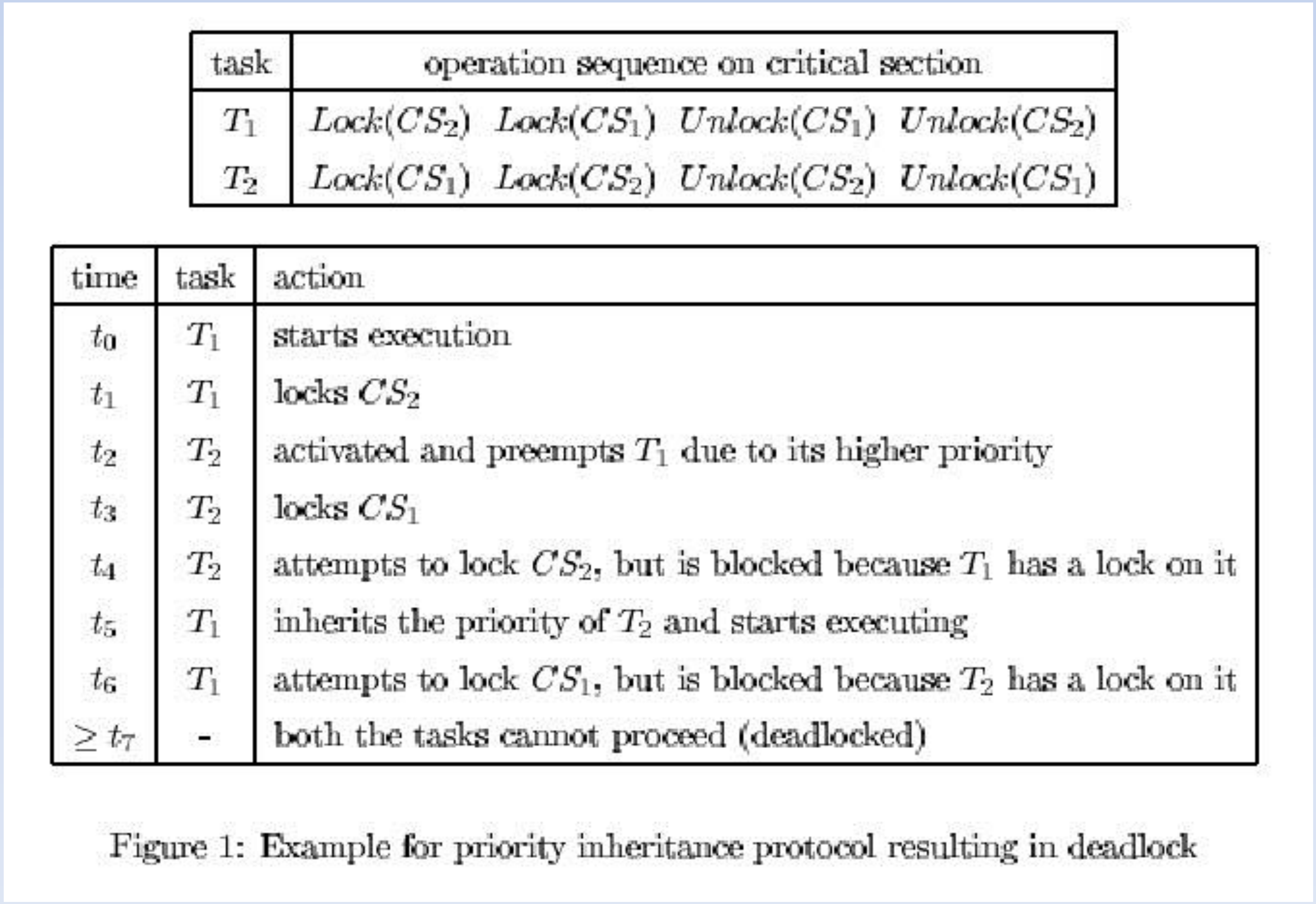
Assume the following actions of the two tasks and assume that T2 > T1 (i.e., T2 has the highest priority)



task	operation sequence on critical section			
T_1	$Lock(CS_2)$	$Lock(CS_1)$	$Unlock(CS_1)$	$Unlock(CS_2)$
T_2	$Lock(CS_1)$	$Lock(CS_2)$	$Unlock(CS_2)$	$Unlock(CS_1)$

Draw the schedule and prove that there is a deadlock.

Solution: Exercise on deadlock with PIP



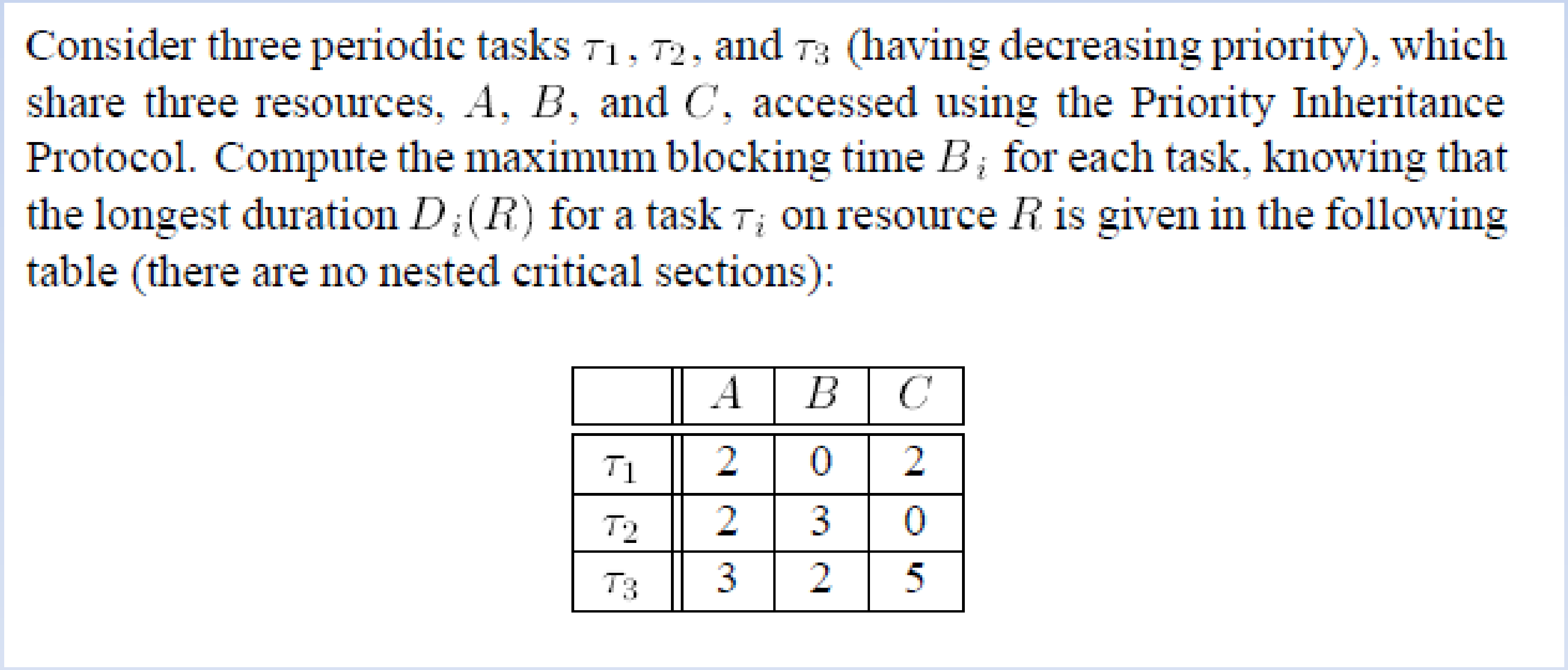
task	operation sequence on critical section			
T_1	$Lock(CS_2)$	$Lock(CS_1)$	$Unlock(CS_1)$	$Unlock(CS_2)$
T_2	$Lock(CS_1)$	$Lock(CS_2)$	$Unlock(CS_2)$	$Unlock(CS_1)$

time	task	action
t_0	T_1	starts execution
t_1	T_1	locks CS_2
t_2	T_2	activated and preempts T_1 due to its higher priority
t_3	T_2	locks CS_1
t_4	T_2	attempts to lock CS_2, but is blocked because T_1 has a lock on it
t_5	T_1	inherits the priority of T_2 and starts executing
t_6	T_1	attempts to lock CS_1, but is blocked because T_2 has a lock on it
$\geq t_7$	-	both the tasks cannot proceed (deadlocked)

Figure 1: Example for priority inheritance protocol resulting in deadlock

:

Exercise: computation of the blocking time with with PIP



Consider three periodic tasks τ_1, τ_2, and τ_3 (having decreasing priority), which share three resources, A, B, and C, accessed using the Priority Inheritance Protocol. Compute the maximum blocking time B_i for each task, knowing that the longest duration $D_i(R)$ for a task τ_i on resource R is given in the following table (there are no nested critical sections):

	A	B	C
τ_1	2	0	2
τ_2	2	3	0
τ_3	3	2	5

PRIORITY CEILING PROTOCOL (PCP)

Principles:

- The Priority Ceiling Protocol (PCP) has been introduced in 1990.
- Can be viewed as PIP + access test.
- A task can enter a CS only if it is free and there is no risk of chained blocking.

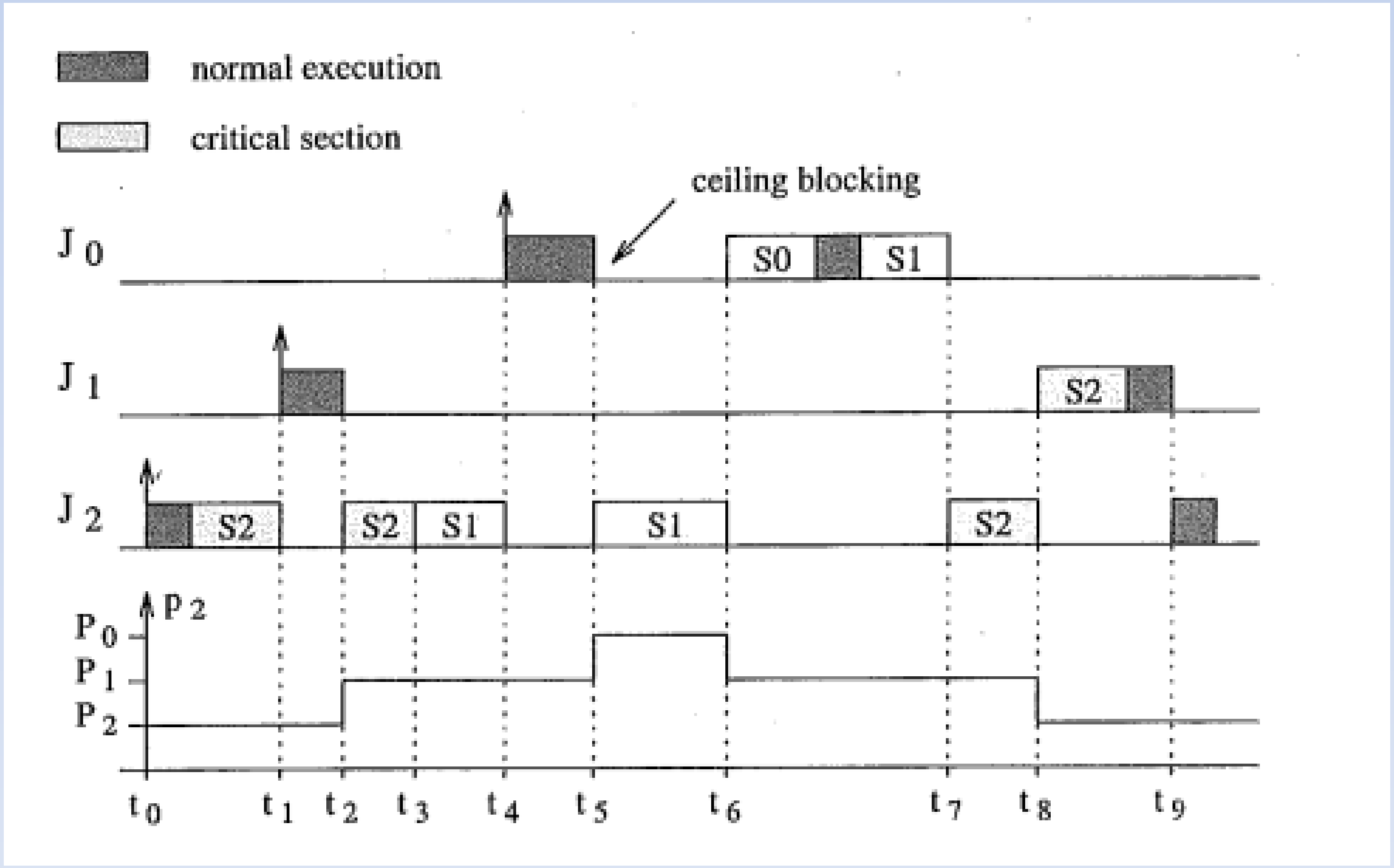
Implementation:

Extension of PIP with one additional rule about access to free semaphores, inserted to guarantee that all required semaphores are free.

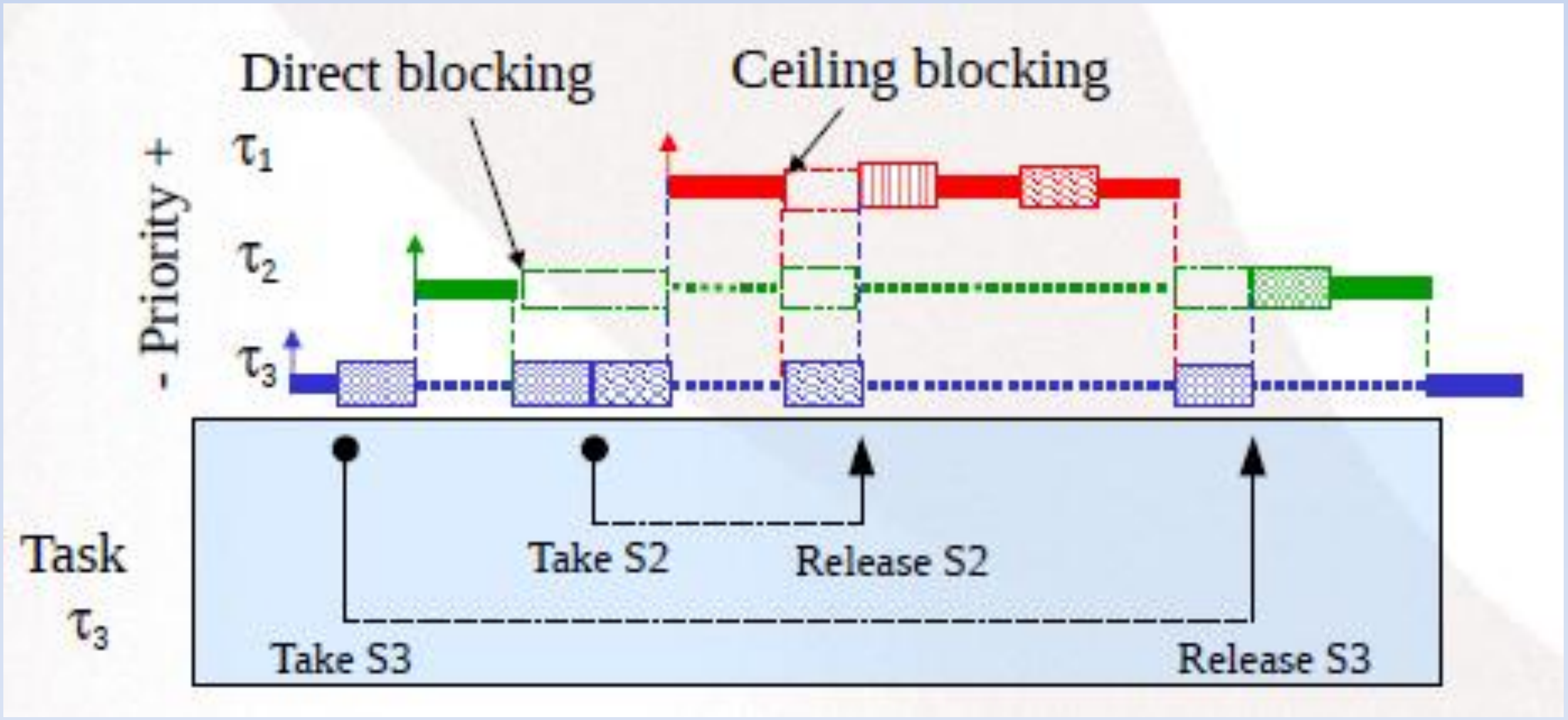
● For each semaphore is defined a **priority *ceiling***, which equals the priority of the maximum priority task that uses it.

● A task can only **take a semaphore** if this one **is free** and if its **priority** is **greater than the ceilings** of all semaphores currently taken.

:


normal execution
critical section
ceiling blocking
J_0
J_1
J_2
S0
S1
S2
S2
S1
S1
S2
S2
P_2
P_0
P_1
P_2
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9

Other example



:

- This protocol is the same as the priority inheritance protocol, except that a task *T_i* can also be blocked from entering a critical section if any other task is currently holding a semaphore whose priority ceiling is greater than or equal to the priority of task *T_i*.

- Prevents mutual deadlock among tasks

- **A task can be blocked by lower priority tasks at most once**

Drawback:

Much harder to implement than PiP. On the TCB it requires one additional field for the inherited priority and another one for the semaphore where the task is blocked.

It also requires a structure to the semaphores, their respective ceilings and the identification of the tasks that are using them

Schedulability test with PCP

$$\forall_{1\leq 1\leq n} \sum_{k=1}^{i} \frac{C_k}{T_k} + \frac{B_i}{T_i} \leq i(2^{\frac{1}{i}} - 1)$$

$$\sum_{1=1}^{n} \frac{C_i}{T_i} + \max_{i=1\ldots n} \frac{B_i}{T_i} \leq n(2^{\frac{1}{n}} - 1)$$

$$R_{wc_i} = C_i + B_i + \sum_{k=1}^{i-1} \lceil \frac{R_{wc_i}}{T_k} \rceil C_k$$

Example:

e.g.	C_i	T_i	B_i
τ_1	5	30	9
τ_2	15	60	8
τ_3	20	80	6
τ_4	20	100	0

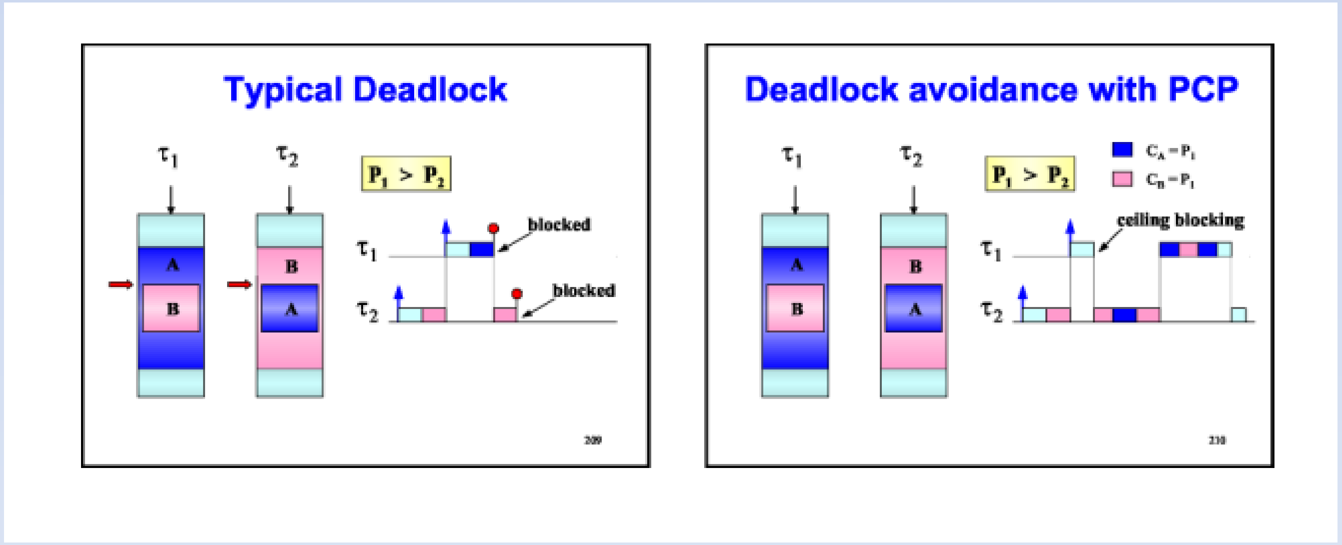
$\leftarrow$

e.g.	S_1	S_2	S_3
τ_1	1	2	0
τ_2	0	9	3
τ_3	8	7	0
τ_4	6	5	4

:

Advantages:

- Blocking is reduced to only one CS because once a task enters its first critical section, it can never be blocked by lower priority tasks until its completion.
- It prevents deadlocks



Problems:
It is not transparent to the programmer: semaphores need ceilings

:

Exercise on PCP

Take the previous example and apply the PCP protocol.

Solution of exercise on PCP

For the previous example, the priority ceiling for both CS_1 and CS_2 is the priority of T_2.

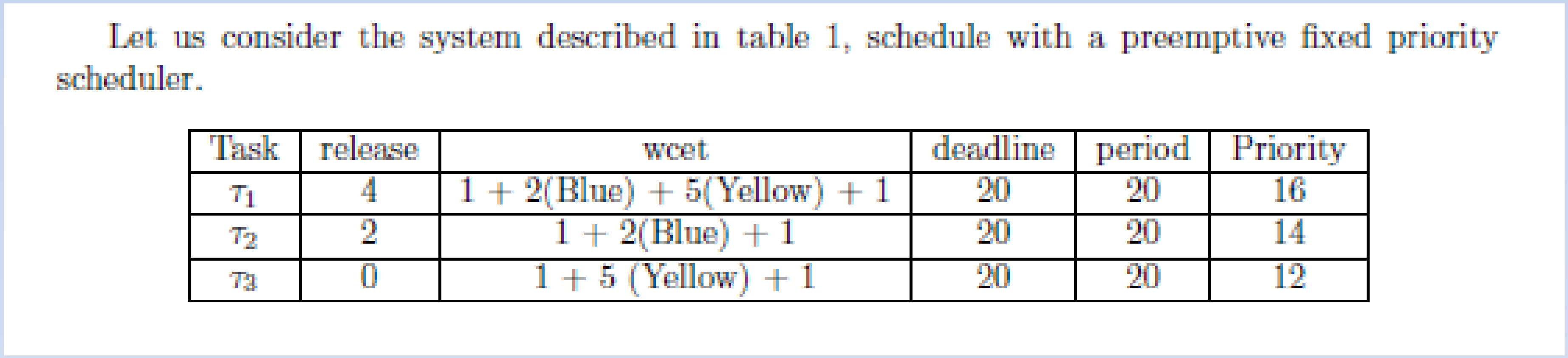
From time t_0 to t_2, the operations are the same as before.

At time t_3, T_2 attempts to lock CS_1, but is blocked since CS_2 (which has been locked by T_1) has a priority ceiling equal to the priority of T_2.

Thus T_1 inherits the priority of T_2 and proceeds to completion, thereby preventing deadlock situation.

:

Exercise on PCP



Let us consider the system described in table 1, schedule with a preemptive fixed priority scheduler.

Task	release	wcet	deadline	period	Priority
τ_1	4	1 + 2(Blue) + 5(Yellow) + 1	20	20	16
τ_2	2	1 + 2(Blue) + 1	20	20	14
τ_3	0	1 + 5 (Yellow) + 1	20	20	12

- draw and explain execution with PIP
- draw and explain execution with PCP

Schedulability test with resource constraints:

- We select a scheduling algorithm and a resource access protocol.
- We compute the maximum blocking times (B_i) for each task T_i.
- We perform the guarantee test including the blocking terms.

Other protocols:

Stack Resource policy (1991) (see book of Buttazzo, chapter 7)

:

APERIODIC TASK SERVICING

PROBLEM TO BE SOLVED

Periodic tasks are time-driven and execute critical control activities with hard timing constraints aimed at guaranteeing regular activation rates.

Aperiodic tasks are usually event-driven and may have hard, soft or non-real real time requirements.

When dealing with hybrid task sets, the objective of the RTOS is to guarantee the schedulability of all critical tasks in worst case conditions and provide good average response times for soft or non-real time activities.

The load on the machine is composed of :

- A static load → periodic tasks
- A dynamic load → aperiodic tasks
-

:

Aperiodic tasks are tasks which arrive **once** and at **unpredictables times**.

An aperiodic task is either:

- **A soft aperiodic task** with no deadline
- **A hard aperiodic task** with a deadline to be met (also called sporadic task)

The objectives of the scheduler are:

1. To meet the deadlines of periodic tasks in all situations
2. To verify on-line that a hard aperiodic task can be scheduled while meeting its deadline
3. To minimize the response time of the soft aperiodic task

:

BASIC SERVERS

Immediate service:

•the best response time but deadlines of periodic tasks are violated

Background service:

•All deadlines are met but the longest response time for the aperiodic task

•execute aperiodic tasks when no periodic ones are executing

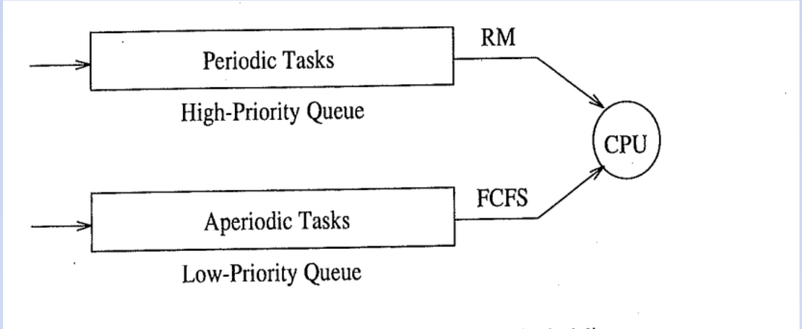
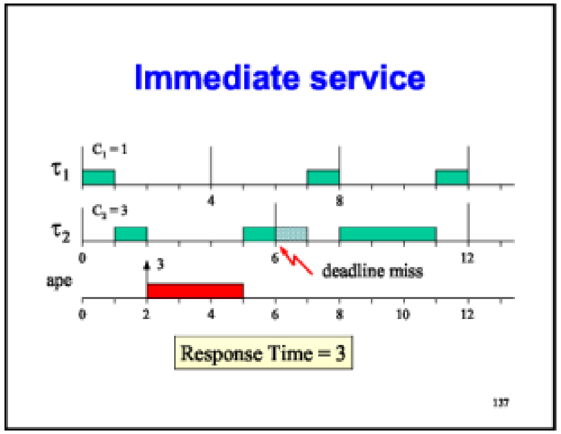
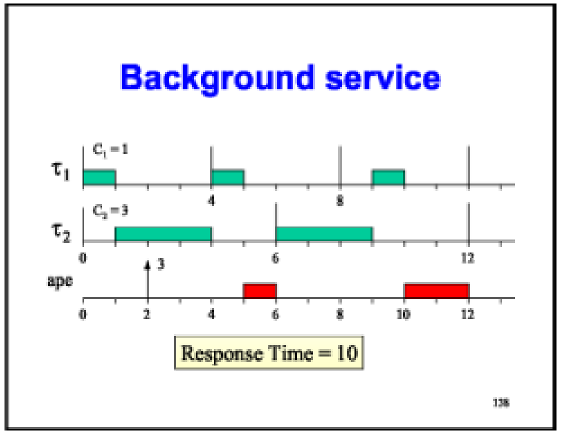


Figure 5.2 Scheduling queues required for background scheduling.





Advantages:

- no disturbance of periodic tasks (and their feasibility)
- simple run-time mechanisms
- queue for periodics
- queue for aperiodics – FCFS

Inconvenients :

- no guarantees
- the longest response time

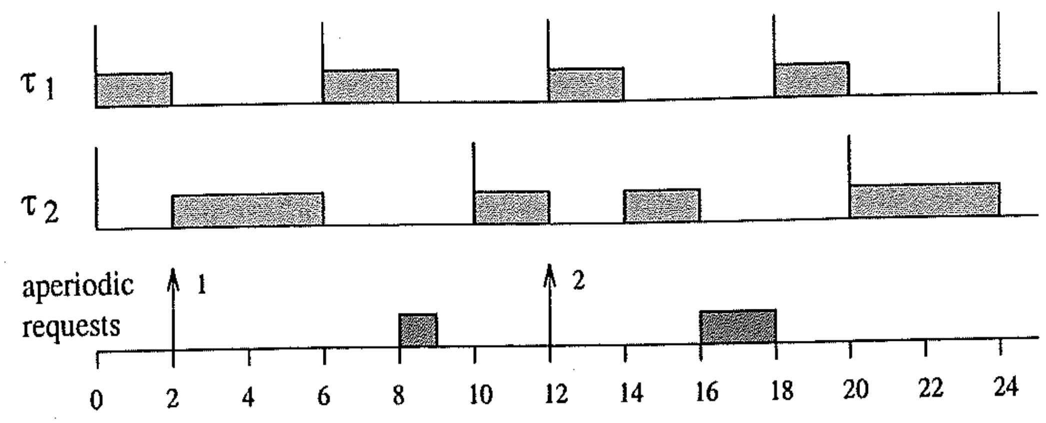


Figure 5.1 Example of background scheduling of aperiodic requests under Rate Monotonic.

:

FIXED PRIORITY SERVERS

Polling Server

The average response time of aperiodic tasks can be improved through the use of **a server**, that is a periodic task whose purpose is to service aperiodic requests as soon as possible.

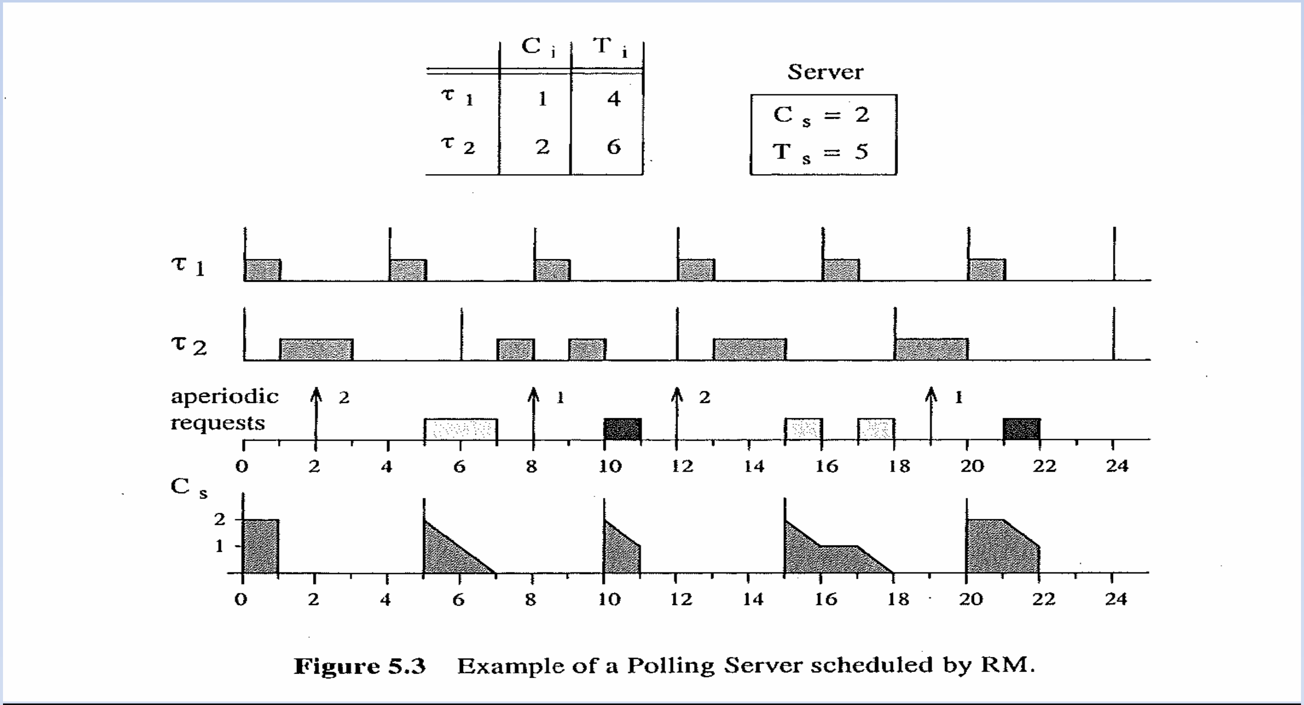
A server is a periodic task that is characterized by:

- A computation time C_s also called **capacity**
- A period T_s

The server is scheduled with the other periodic tasks and once active, serves the aperiodic tasks within the limit of its capacity.
If no aperiodic tasks are pending PS suspends itself until the beginning of the next period.

Remarks:
If an aperiodic task occurs just after the server has suspended, it must wait until the next period.

:



	C_i	T_i
τ_1	1	4
τ_2	2	6

Server: $C_s = 2$, $T_s = 5$

Figure 5.3 Example of a Polling Server scheduled by RM.

If periodic tasks are scheduled by RM, the schedulability test is:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} + \frac{C_s}{T_s} \leq (n+1)[2^{1/(n+1)} - 1].$$

Exercise on the Polling server

Let us consider two periodic tasks and a server with the following parameters:

Task set: $T_i = (c_i, p_i)$

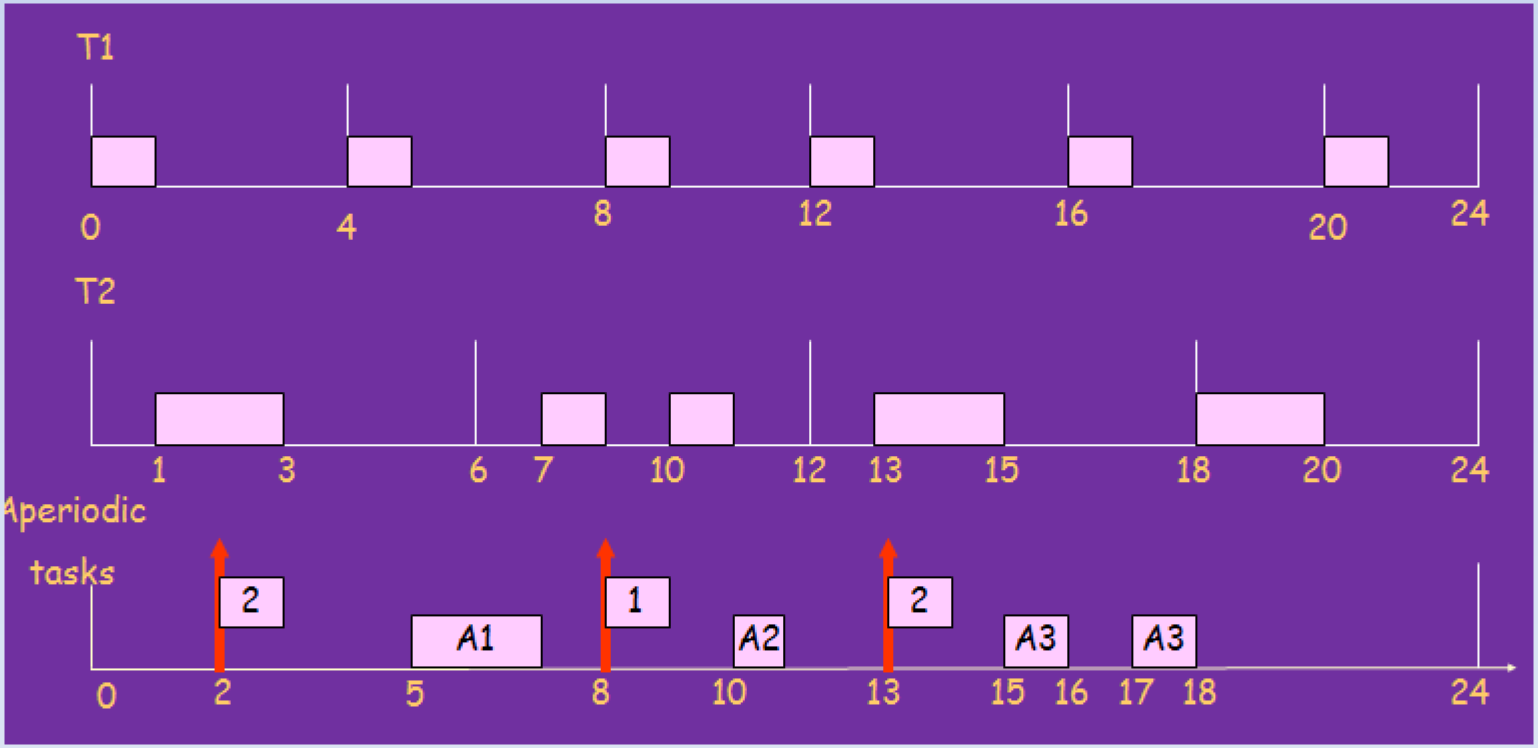
$T1 = (1,4)$, $T2 = (2,6)$ and <u>$Ts = (2,5)$</u>

Consider that three aperiodic task arrive as follows:

- At time 2 with computation time 2
- At time 8 with computation time 1
- At time 13 with computation 2

Draw the schedule and compute the average response time for the three aperiodic tasks

Solution for the exercise on the Polling server



Deferrable Server

Description :
The Deferrable Server (DS) has been introduced in 1987 to improve the average response time with respect to Polling

DS uses a periodic task, the server, usually with the highest priority.

DS preserves its capacity if no requests are pending upon the invocation of the server.

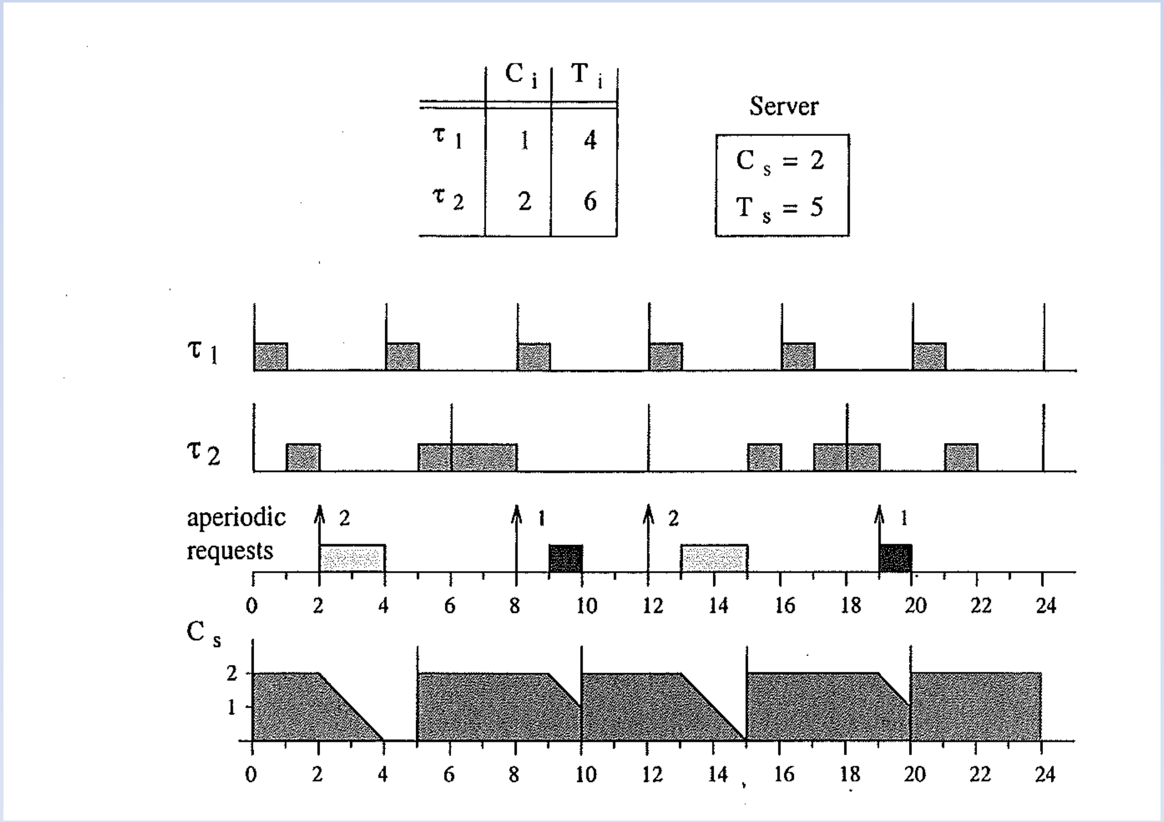
The capacity is maintained until the end of the period, so that aperiodic tasks can be serviced at the server's priority as long as the capacity has not been exhausted.

:

The **average response time** to aperiodic requests is **improved** with respect to the PS, since it is possible to use the capacity of the DS during the whole period, provided that its capacity is not exhausted.

However, there is a **negative impact** on the schedulability of the periodic tasks.
The reason for this impact is that the **delayed executions** increase the **load** on the **future**.

Example of a Deferrable server scheduled by RM:



	C_i	T_i
τ_1	1	4
τ_2	2	6

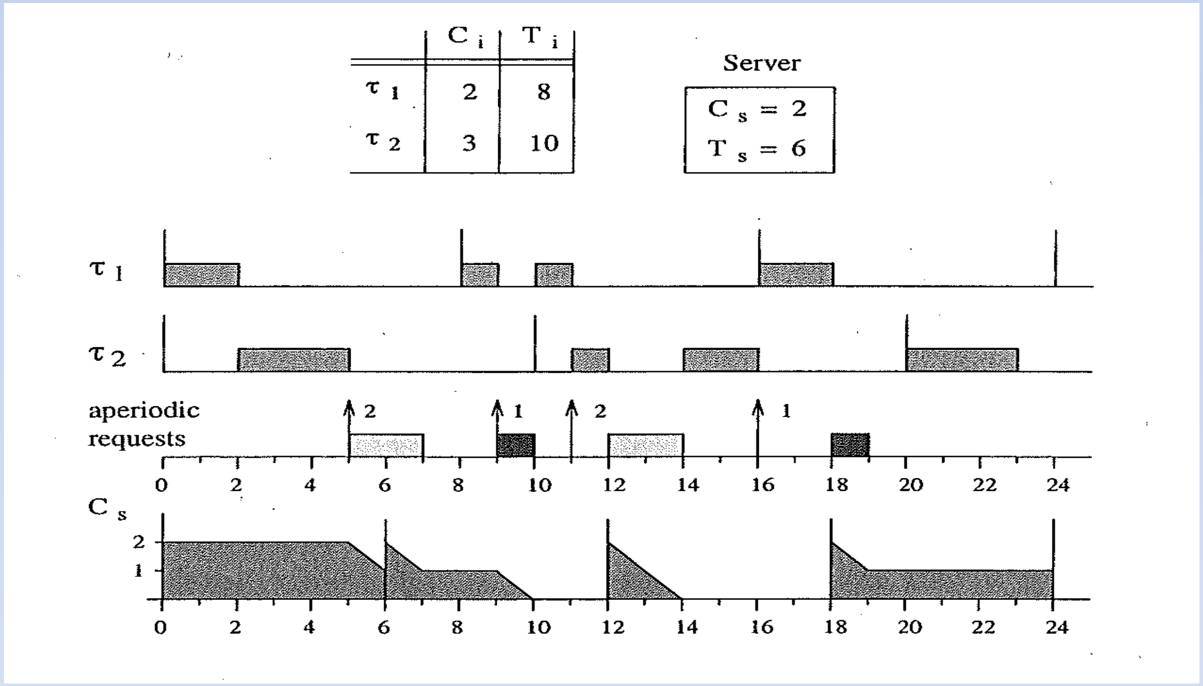
Server

$C_s = 2$

$T_s = 5$

:

Example of a high priority server:



	C_i	T_i
τ_1	2	8
τ_2	3	10

Server

$C_s = 2$
$T_s = 6$

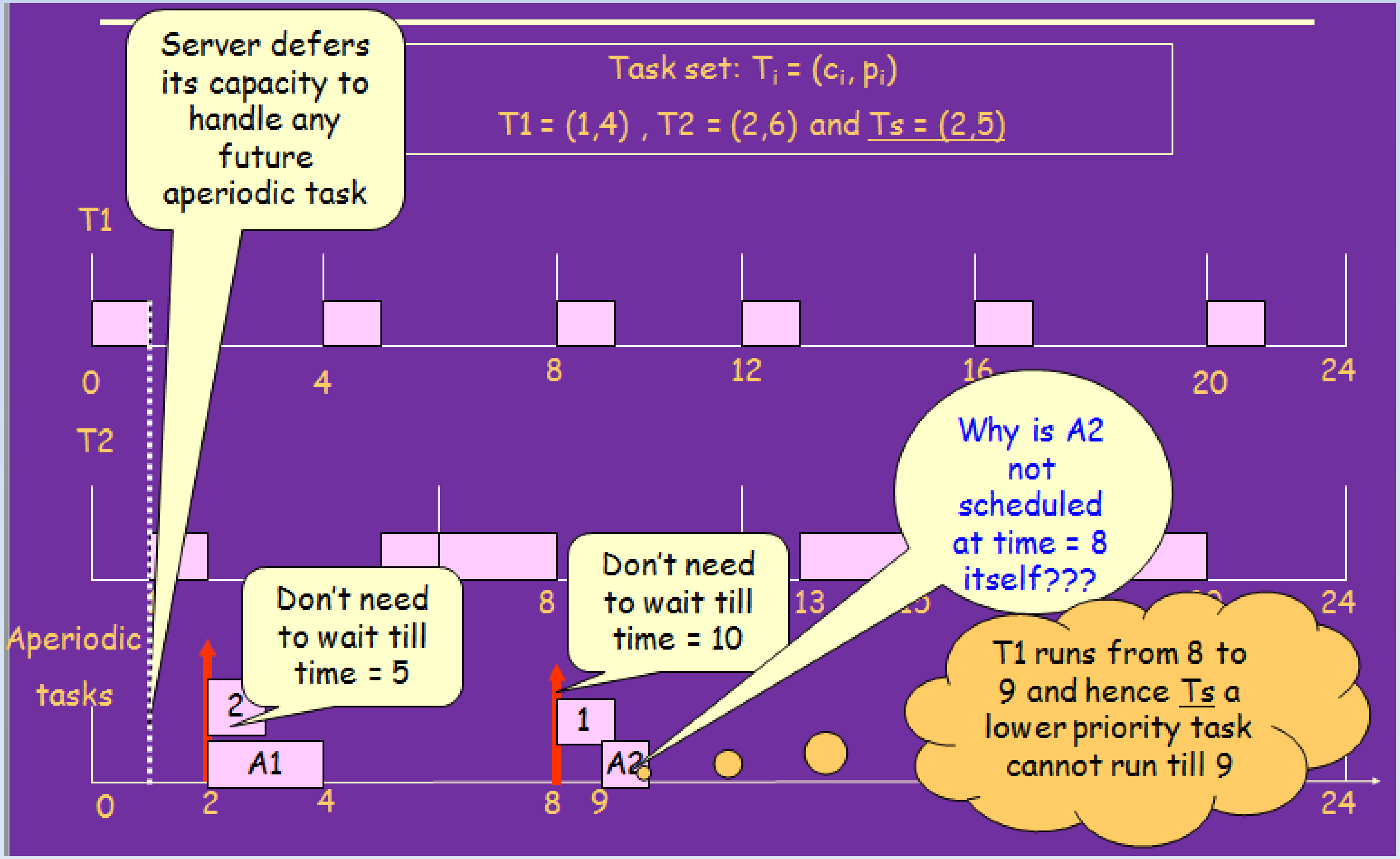
For a set of n periodic tasks and a Deferrable Server with utilization factors U_p and U_s, the schedulability is guaranteed under RM **if**

$U_p \leq \ln (U_s + 2/ 2U_s + 1)$

Exercise on the Deferrable server

Take the previous exercise and apply this server.

Solution for the exercise on the Deferrable server


Server defers its capacity to handle any future aperiodic task
Task set: $T_i = (c_i, p_i)$
T1 = (1,4) , T2 = (2,6) and Ts = (2,5)
T1
0
4
8
12
16
20
24
T2
8
13
24
Aperiodic tasks
2
A1
1
A2
0
2
4
8
9
24
Don't need to wait till time = 5
Don't need to wait till time = 10
Why is A2 not scheduled at time = 8 itself???
T1 runs from 8 to 9 and hence Ts a lower priority task cannot run till 9

:

Sporadic Server (SS)

Principles:
The Sporadic Server has been introduced in 1989.

The SS algorithm creates a high priority task for servicing aperiodic tasks and like DS preserves the server capacity at its high priority level until an aperiodic task occurs.

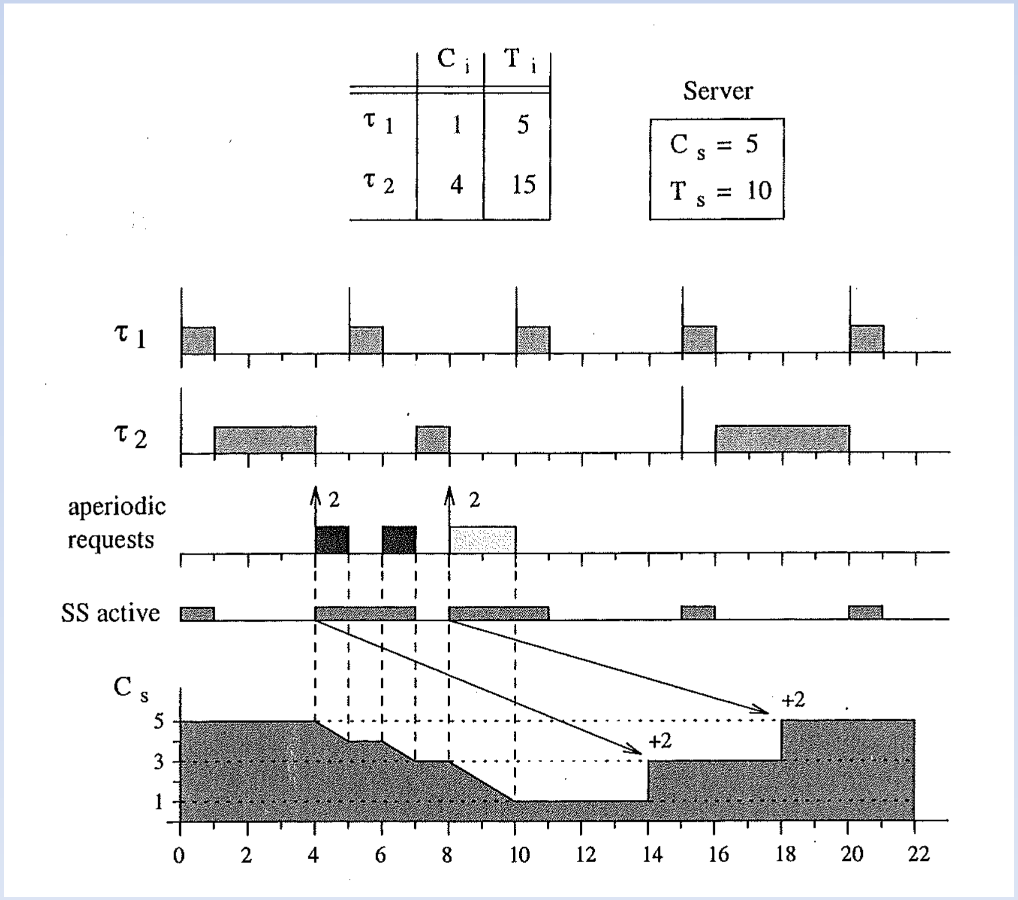
However SS differs from DS in the way it replenishes its capacity.

Whereas DS periodically replenish its capacity to its full value at the beginning of each server period, SS replenishes its capacity only after it has been consumed by aperiodic task execution.

The implementation **complexity** of a sporadic server is higher than the one of PS and DS, due to the computation of the replenishment instants and, more importantly, to the **complex timer management**

:

Example of a medium-priority Sporadic Server



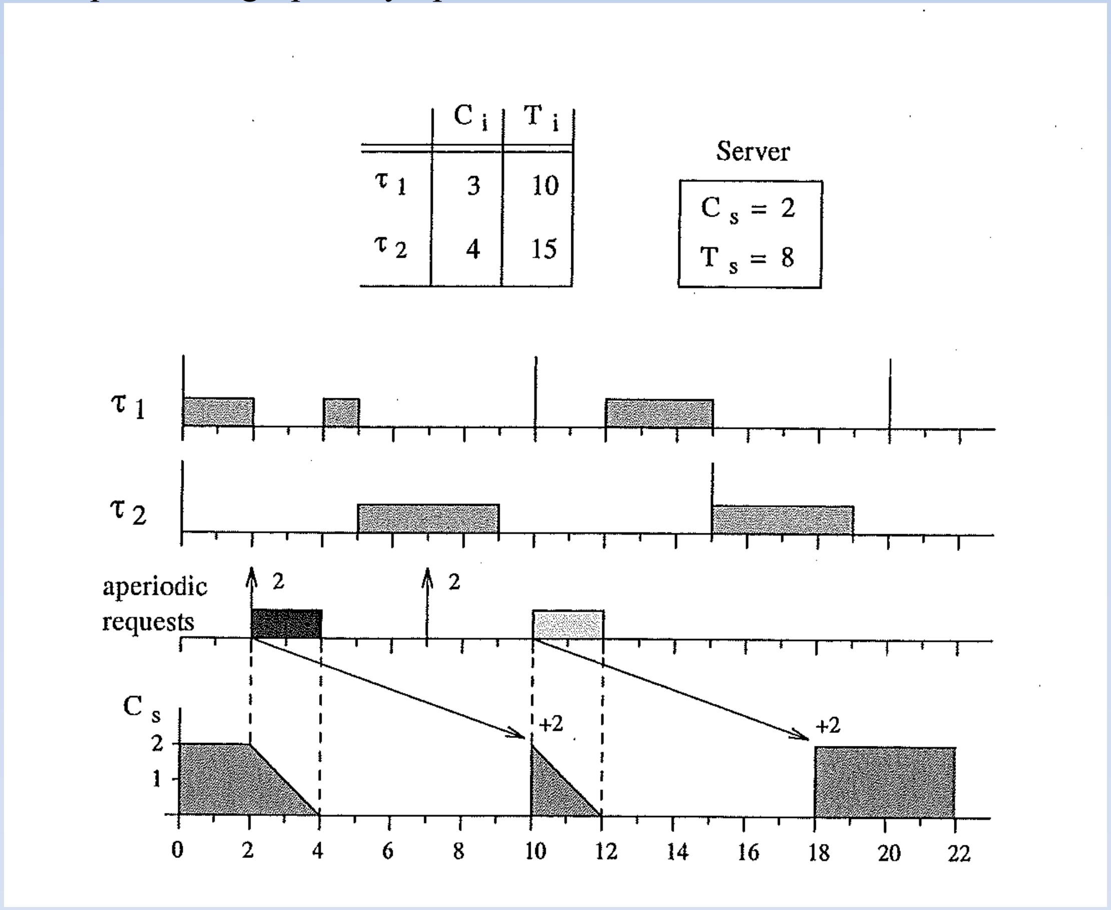
	C_i	T_i
τ_1	1	5
τ_2	4	15

Server

$C_s = 5$

$T_s = 10$

Example of a high-priority Sporadic Server



	C_i	T_i
τ_1	3	10
τ_2	4	15

Server

$C_s = 2$

$T_s = 8$

Using RM+SS and giving higher priority to the server:

$$U_p + U_s \le U_s + n\left(\left(\frac{2}{U_s+1}\right)^{\frac{1}{n}} - 1\right)$$

Exercise on the Sporadic Server

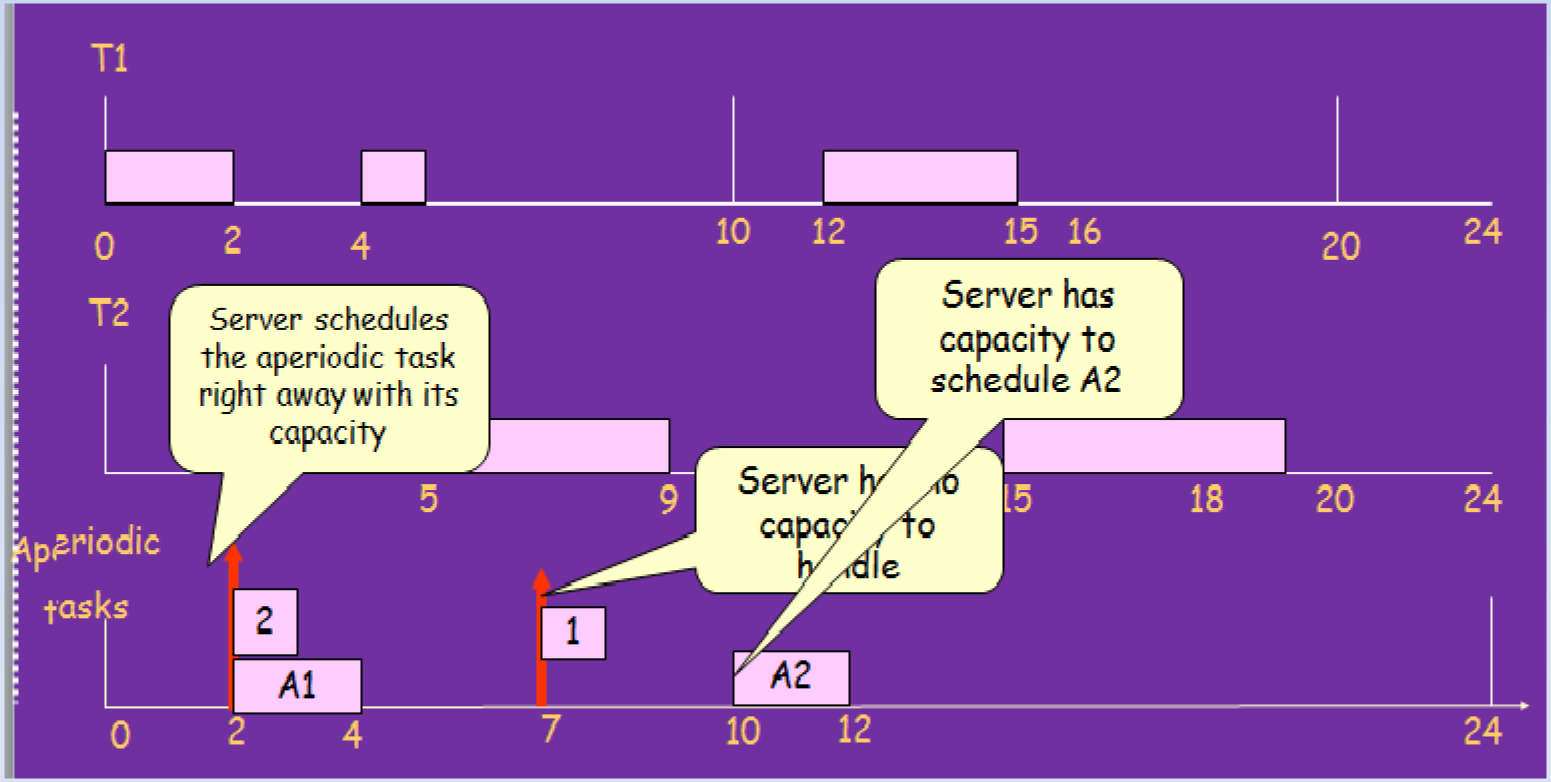
Task set: $T_i = (c_i, p_i)$

T1 = (3,10) , T2 = (4,15) and Ts = (2,8)

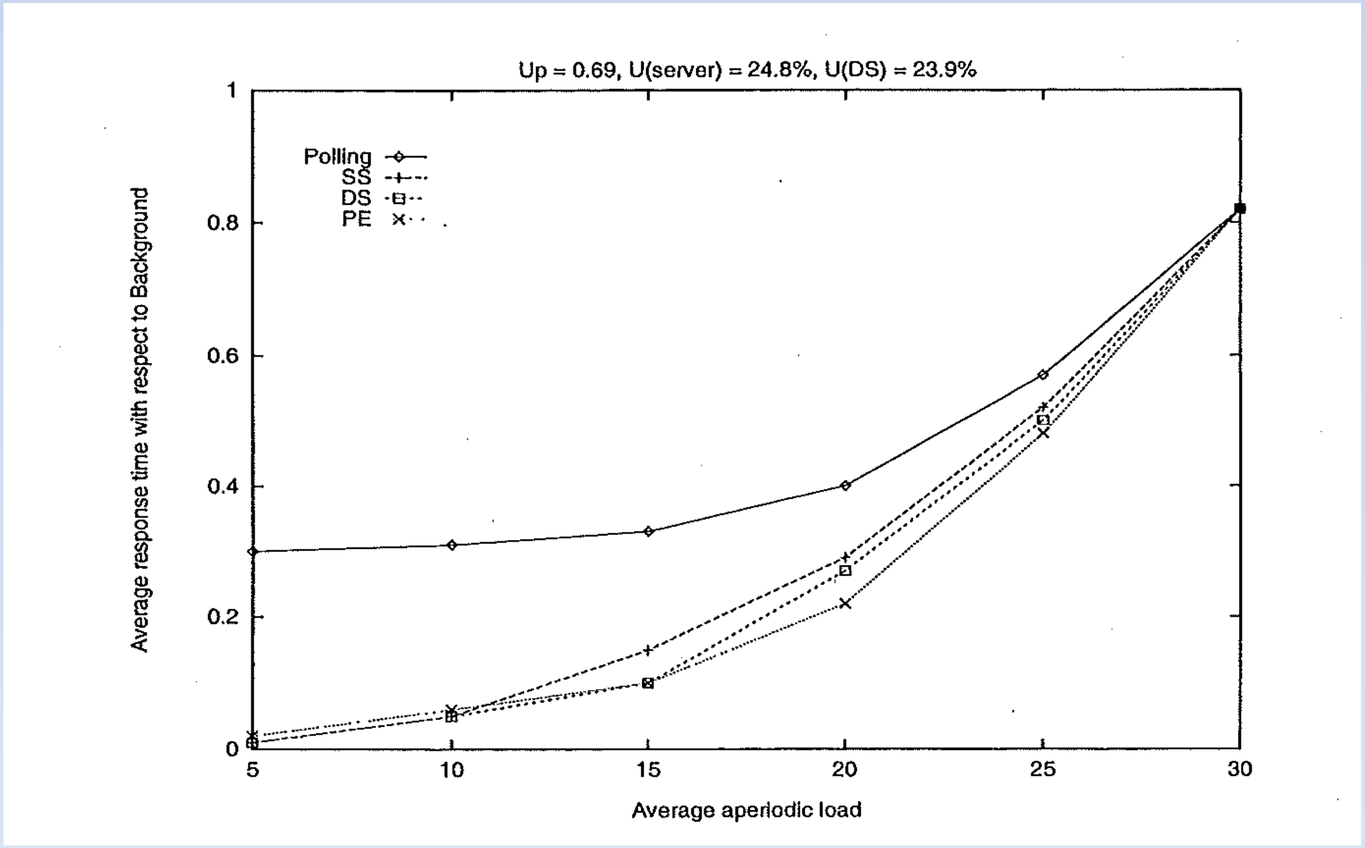
Let us consider the following aperiodic tasks:

- Arrives at time 2 with computation 2
- Arrives at time 7 with computation 1

Solution for exercise on the Sporadic Server


T1
0
2
4
10
12
15
16
20
24
T2
Server schedules the aperiodic task right away with its capacity
5
9
Server has capacity to schedule A2
15
18
20
24
Aperiodic tasks
2
A1
1
A2
0
2
4
7
10
12
24

Comparison of BS, SS, DS and PS:



:

DYNAMIC PRIORITY SERVERS

Dynamic schedulers are characterized by higher schedulability bounds

→ allow the processor to be better utilized

→ increase the size of aperiodic servers

→ enhance aperiodic responsiveness

Dynamic Sporadic Server

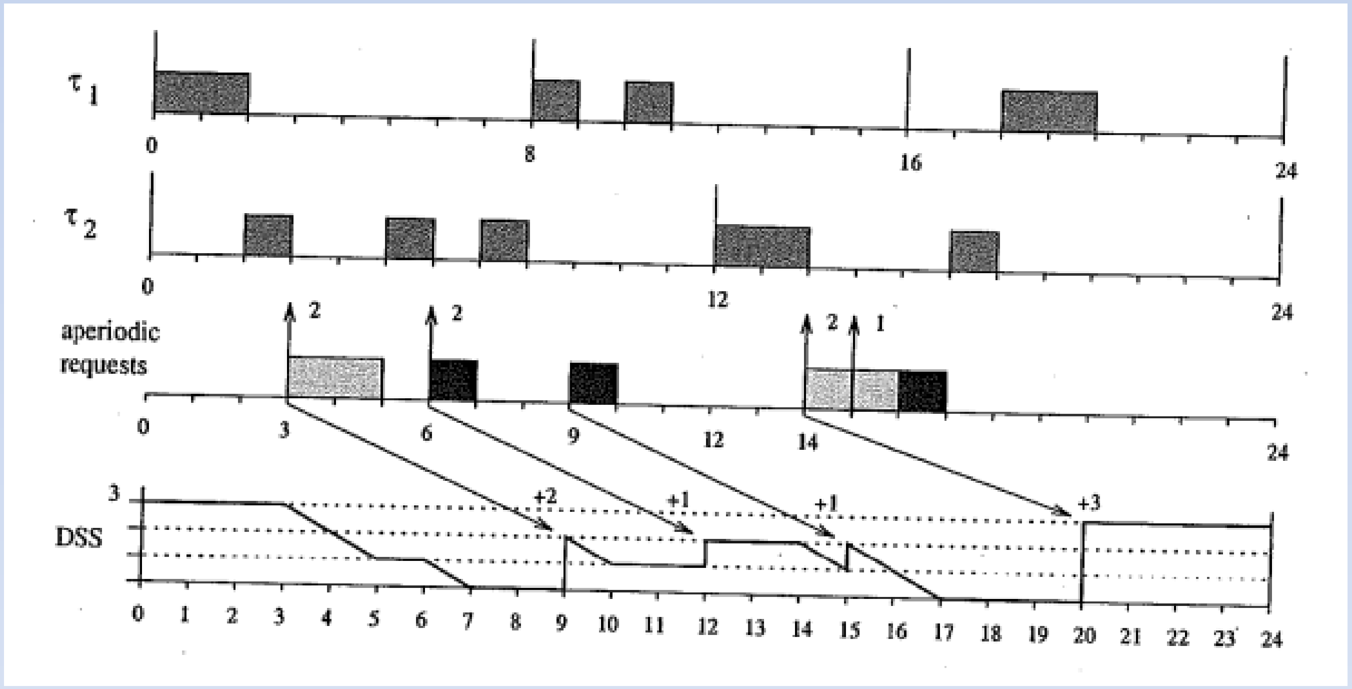
DSS is characterized by a period and a capacity which is preserved for aperiodic requests

DSS has a dynamic priority assigned through a suitable deadline

Given a set of n periodic tasks with processor utilization U_p and Dynamic Sporadic Server with processor utilization U_s,

the whole set is schedulable **if and only if** $U_p + U_s \leq 1$

:



τ_1
0
8
16
24
τ_2
0
12
24
aperiodic requests
2
2
2
1
0
3
6
9
12
14
24
DSS
3
+2
+1
+1
+3
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

Earliest Deadline Late server (Slack Stealing server)

The EDL server has been proposed by Chetto in 1989. It is a **Slack Stealing** server.

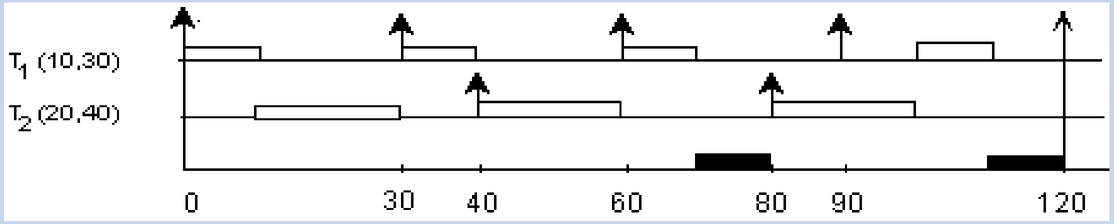
It uses the available slack of periodic tasks for advancing the execution of the aperiodic requests.
It is based on the following property: In any interval [0, t], EDL guarantees the maximum available idle time.

The EDL server is optimal.

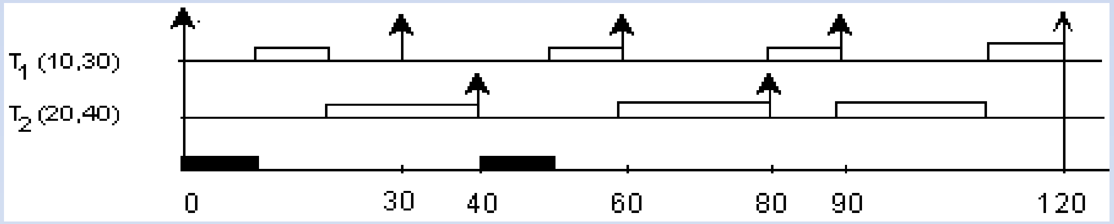
Chetto and Chetto 1989

:

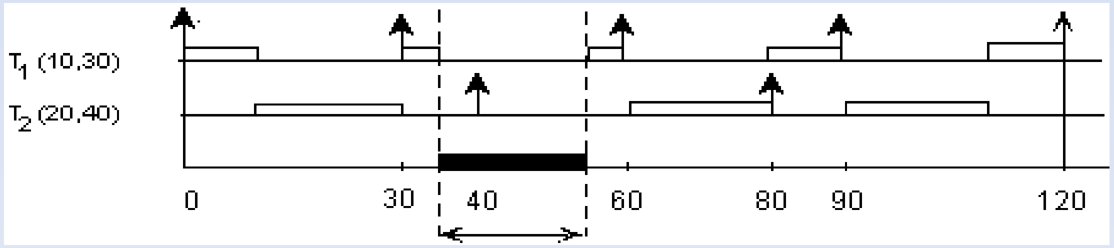
Principles of EDS (Earliest Deadline Soon)



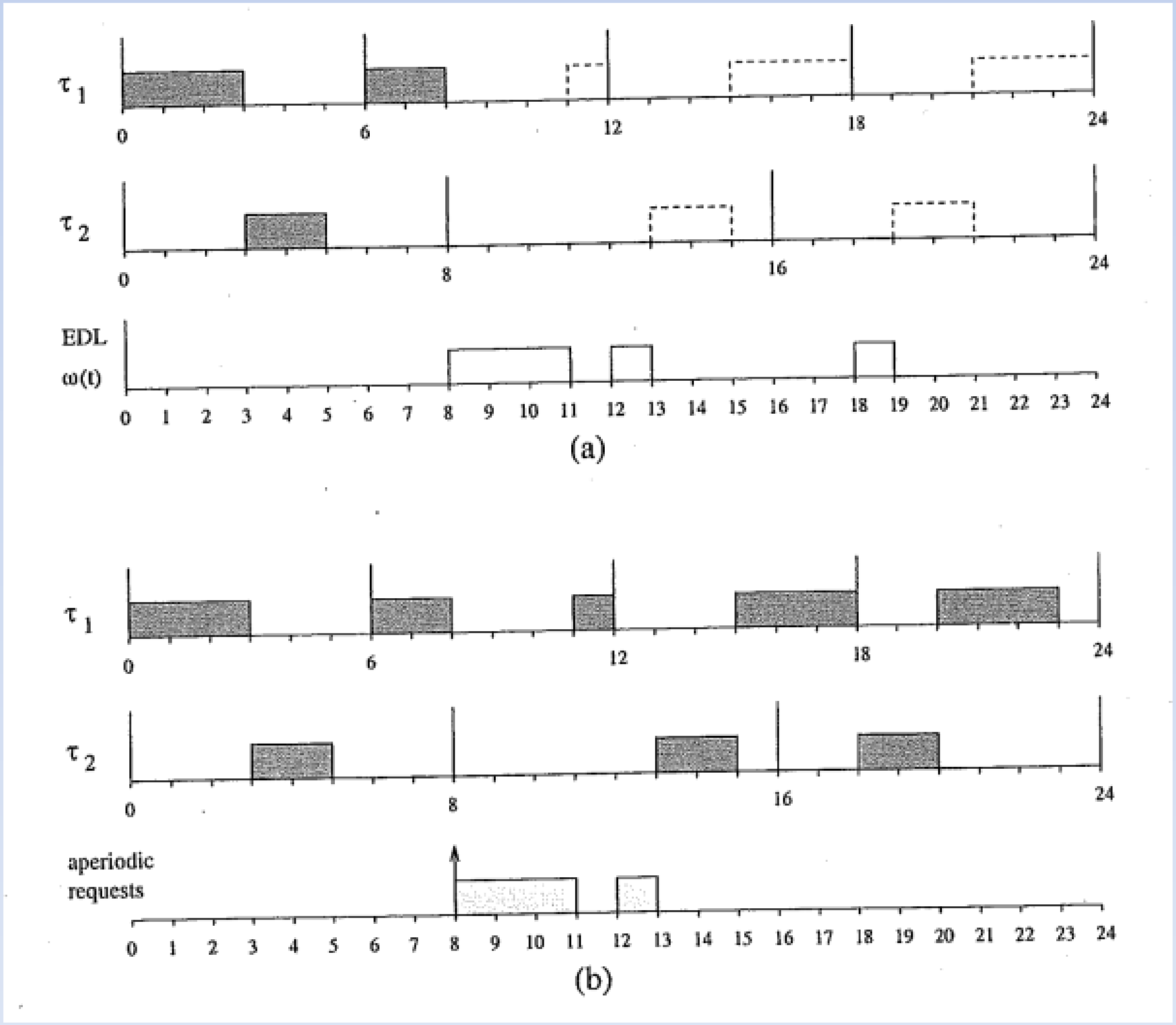
Principles of EDL (Earliest Deadline Late)



Maximum laxity at current time with EDL



:



τ1
τ2
EDL
ω(t)
(a)
aperiodic
requests
(b)
0
6
8
12
16
18
24

:

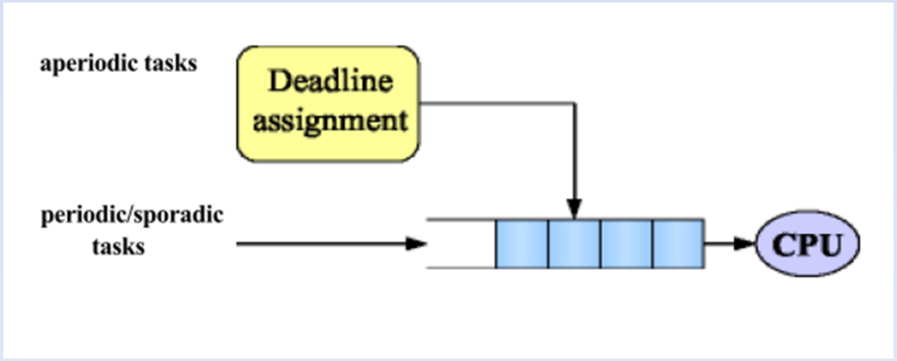
The Total Bandwidth Server (TBS)

- The Total Bandwidth Server (TBS) has been introduced in 1994.
- The ***Total Bandwidth Server*** is a **dynamic priority** server which has the objective of executing the aperiodic requests **as soon as possible** while **preserving** the **bandwidth** assigned to it, to not disturb the periodic tasks.

It was developed for EDF systems.

- Each occurring aperiodic request with computation time C_k and arrival time r_k is assigned a deadline d_k so as not to jeopardize periodic tasks

The following formula is used : $\mathbf{d_k = \max(r_k, d_{k-1}) + C_k / U_s}$



:

Advantages:

- TBS is **simple** to implement and has low overhead, since it only requires a simple computation (deadline for each arrival). Then the aperiodic request is inserted in the ready queue and handled as any other task.

● The **average response time** to aperiodic requests is **smaller** than the one obtained with dynamic- priority versions of fixed-priority servers.

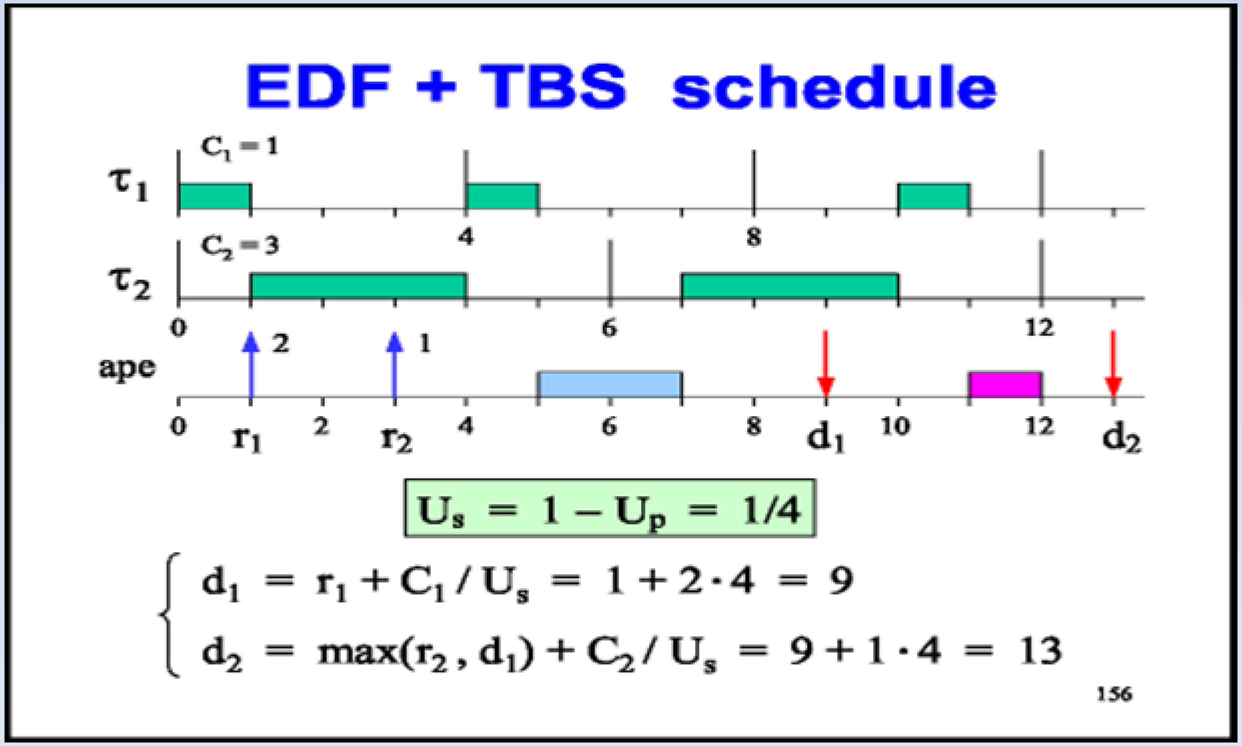
● The **impact** on the schedulability of the periodic task set is equal to the one of a periodic task with utilization equal to the one granted to the server.

Drawback:

Requires a priori knowledge of computation times of aperiodic tasks

:

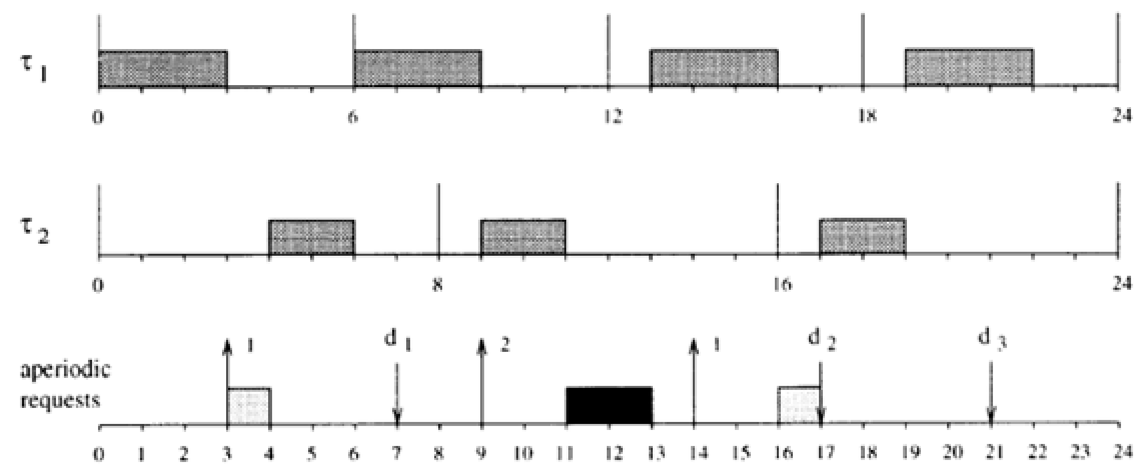
Periodic tasks with processor utilization U_p and a TBS with processor utilization U_s are guaranteed under EDF if and only if $U_p + U_s \leq 1$



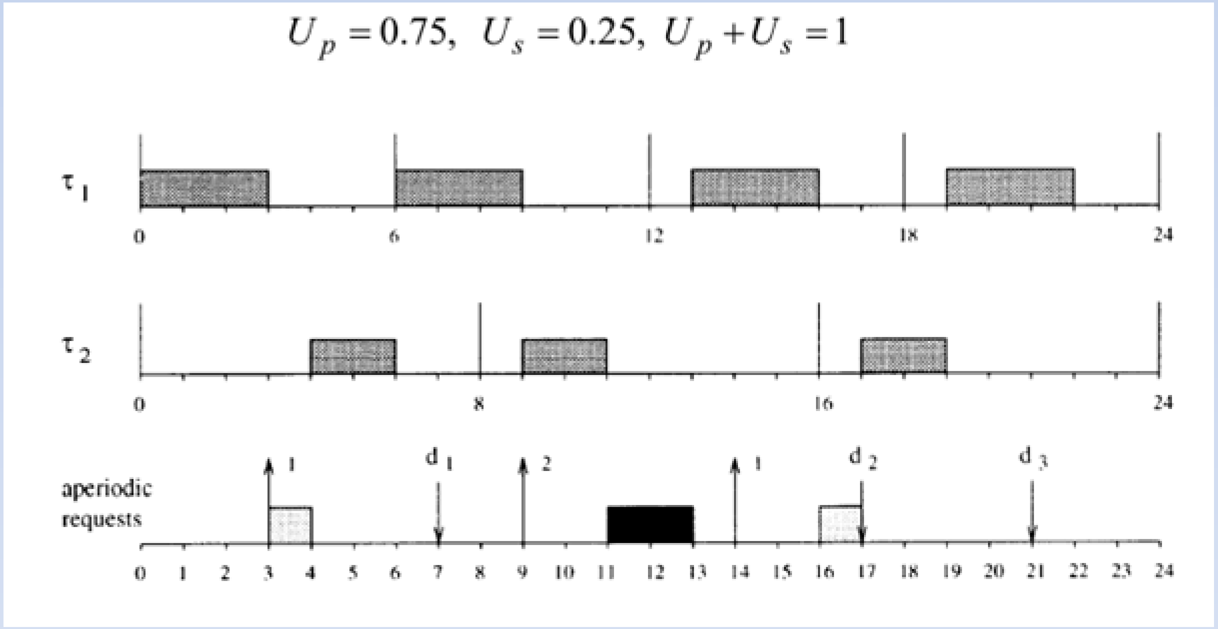
$U_s = 1 - U_p = 1/4$

$$\begin{cases} d_1 = r_1 + C_1 / U_s = 1 + 2 \cdot 4 = 9 \\ d_2 = \max(r_2, d_1) + C_2 / U_s = 9 + 1 \cdot 4 = 13 \end{cases}$$

$$U_p = 0.75, \; U_s = 0.25, \; U_p + U_s = 1$$



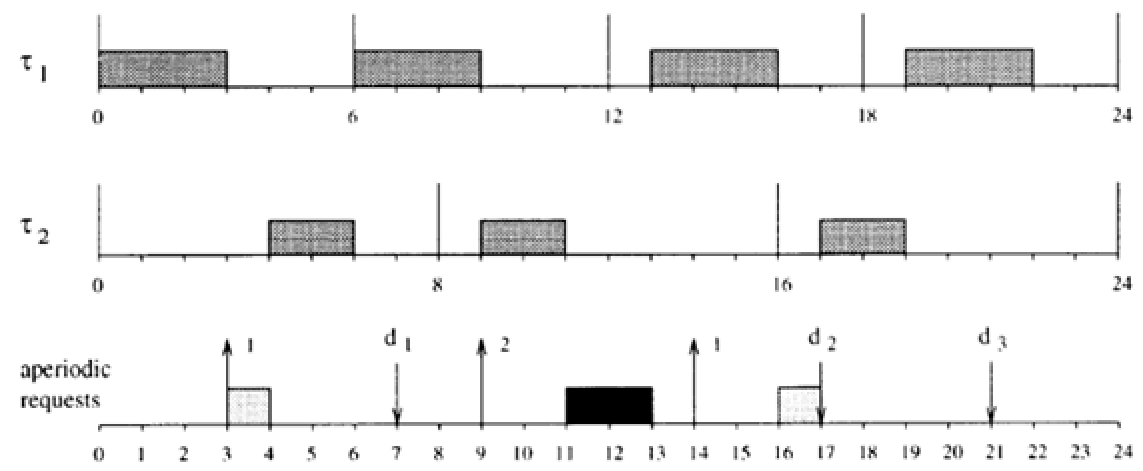
:



$$U_p = 0.75,\ \ U_s = 0.25,\ \ U_p + U_s = 1$$

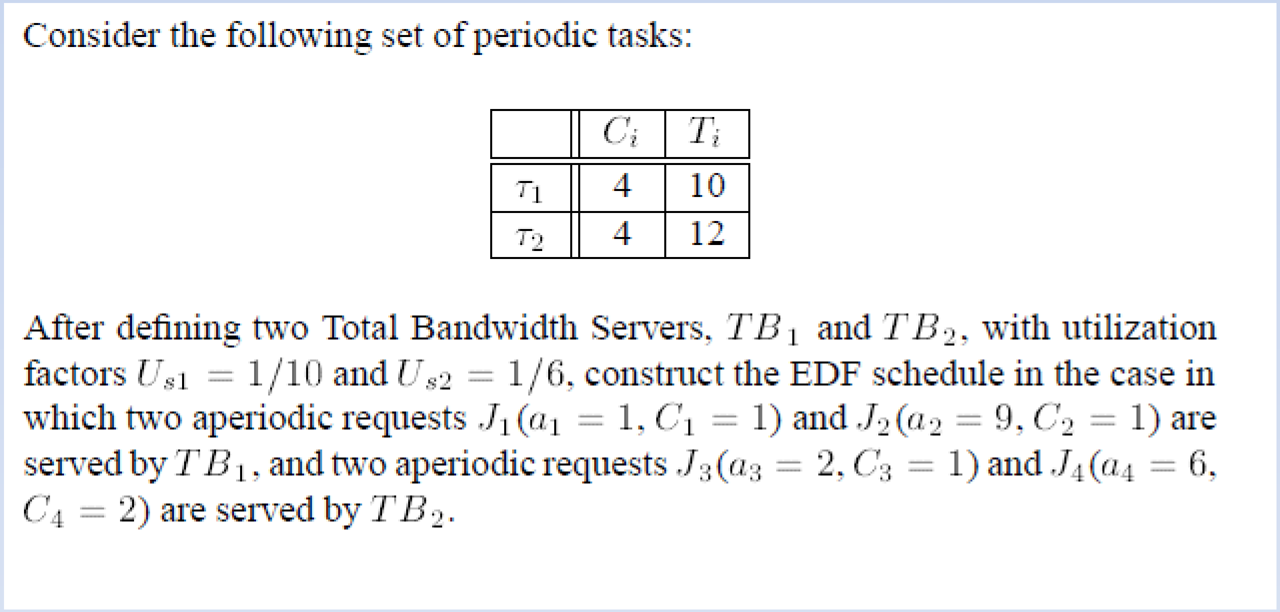
:

$$U_p = 0.75,\ \ U_s = 0.25,\ \ U_p + U_s = 1$$



:

Exercise on the TBS aperiodic server



Consider the following set of periodic tasks:

	C_i	T_i
τ_1	4	10
τ_2	4	12

After defining two Total Bandwidth Servers, TB_1 and TB_2, with utilization factors $U_{s1} = 1/10$ and $U_{s2} = 1/6$, construct the EDF schedule in the case in which two aperiodic requests $J_1(a_1 = 1, C_1 = 1)$ and $J_2(a_2 = 9, C_2 = 1)$ are served by TB_1, and two aperiodic requests $J_3(a_3 = 2, C_3 = 1)$ and $J_4(a_4 = 6, C_4 = 2)$ are served by TB_2.

:

SCHEDULING WITH PRECEDENCE CONSTRAINTS

Modification of the timing parameters

We express the task dependencies with usual task parameters:
(a) Assign priorities according to dependencies (Chetto 1990)

(b) Change release time and deadlines according to dependenices (Chetto 1990)

The idea is to make tasks independent with modified timing parameters

This method assumes either aperiodic tasks or tasks with the same period.

Modification of the parameters

With Rate Monotonic:

Parameter updates:

If τi precedes τj then priority (τi)> priority(τj)

:

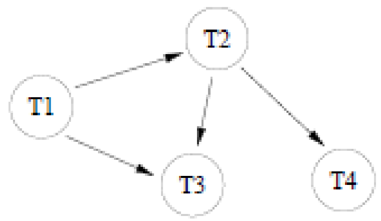
Modification of the parameters

With Earliest Deadline First:

Parameter updates:

Modification of deadline:
$D_i^* = \min(D_i, \min(D_j^* - C_j$ such that τ_j is a successor of τ_i).

Illustration



	C_i	D_i	D_i^*
T_4	2	14	14
T_3	1	8	8
T_2	2	10	7
T_1	1	5	5

- Example : EDF + aperiodic tasks.

  - $D_4^* = 14; D_3^* = 8;$
  - $D_2^* = min(D_2, D_3^* - C_3, D_4^* - C_4) = min(10, 8-1, 14-2) = 7;$
  - $D_1^* = min(D_1, D_2^* - C_2, D_3^* - C_3) = min(5, 7-2, 8-1) = 5;$

Example taken from the course of Frank Singhoff, University of Brest)