artist

scuola superiore Sant'Anna di studi universitari e di perfezionamento

Graduate Course on Embedded Control Systems: Theory and Practice

Retis Real-Time Systems Laboratory

Scuola Superiore Sant'Anna, Pisa

8-12 June 2009

Program at glance

- Day 1: **Mon. July 8 – Real-Time Day**
- Day 2: **Tue. July 9 – Platform Day**
- Day 3: **Wed. July 10 – Control Day**
- Day 4: **Thu. July 11 – Networks Day**
- Day 5: **Fri. July 12 – Judgment Day**

Mon. July 8 – Real-Time Day

09:00	Introduction to real-time systems	(Giorgio Buttazzo)
10:30	Coffee Break	
11:00	Real-time scheduling and resource management	
13:00	Lunch Break	
14:00	Aperiodic Scheduling and Reservations	
16:00	Break	
16:30	dsPic architecture: overview	(Mauro Marinoni)
18:30	End of Session	

Tue. July 9 – Platform Day

08:30 Operating systems for micro-controllers (Paolo Gai)

10:30 Coffee Break

11:00 The OSEK standard (Paolo Gai)

13:00 Lunch Break

14:00 The Erika kernel (Paolo Gai)

16:00 Break

16:30 Lab practice on Flex and Erika (Mauro Marinoni)

18:30 End of Session

Wed. July 10 – Control Day

08:30 Integrated control and scheduling (Karl-Erik Arzen)

10:30 Coffee Break

11:00 Control of computing systems (Karl-Erik Arzen)

13:00 Lunch Break

14:00 Lab practice on control (Anton Cervin)

16:00 Break

16:30 Lab practice on control (Anton Cervin)

18:30 End of Session

Thu. July 11 – Network Day

08:30 Real-Time Networks (Luis Almeida)

10:30 Coffee Break

11:00 Medium Access Control (Luis Almeida)

13:00 Lunch Break

14:00 Networked control systems (Anton Cervin)

16:00 Break

16:30 Lab practice on networks (Anton Cervin)

18:30 End of Session

Fri. July 12 – Judgment Day

09:00 Final Exam (3 credits)

13:00 Lunch Break

14:00 Lab practice

18:00 Closing remarks and certificates

Real-Time Scheduling

Giorgio Buttazzo

Scuola Superiore Sant'Anna, Pisa

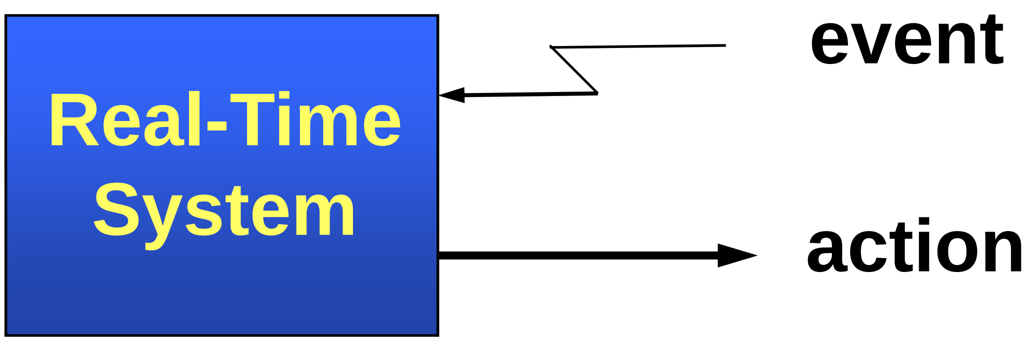
E-mail: buttazzo@unipv.it

Goal

Provide some background of RT theory that you can apply for implementing RT control applications:

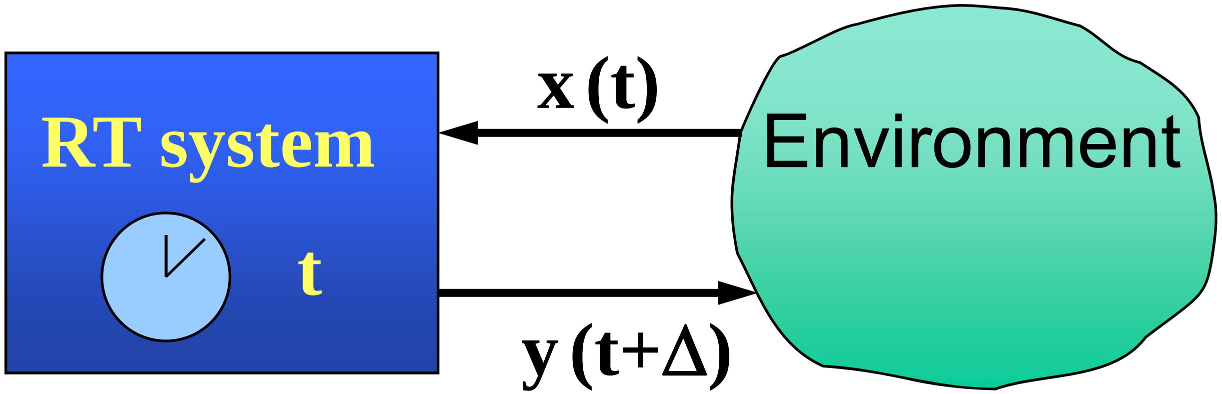
- Terminology and models
- Basic results on periodic scheduling
- Aperiodic task handling
- Inter-task communication
- Overload and QoS management

Real-Time system



A computing system able to respond to events within precise timing constraints.

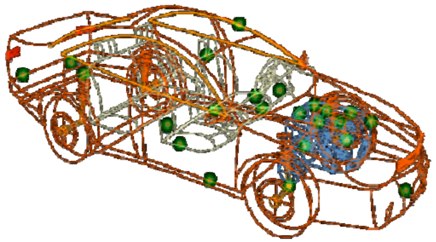
Real-Time system



It is a system in which the correctness depends not only on the output values, but also on the **time** at which results are produced.

Typical applications

- flight control systems
- robotics
- automotive applications
- multimedia systems
- virtual reality
- small embedded devices
  - ⇒ cell phones
  - ⇒ digital TV
  - ⇒ videogames
  - ⇒ intelligent toys



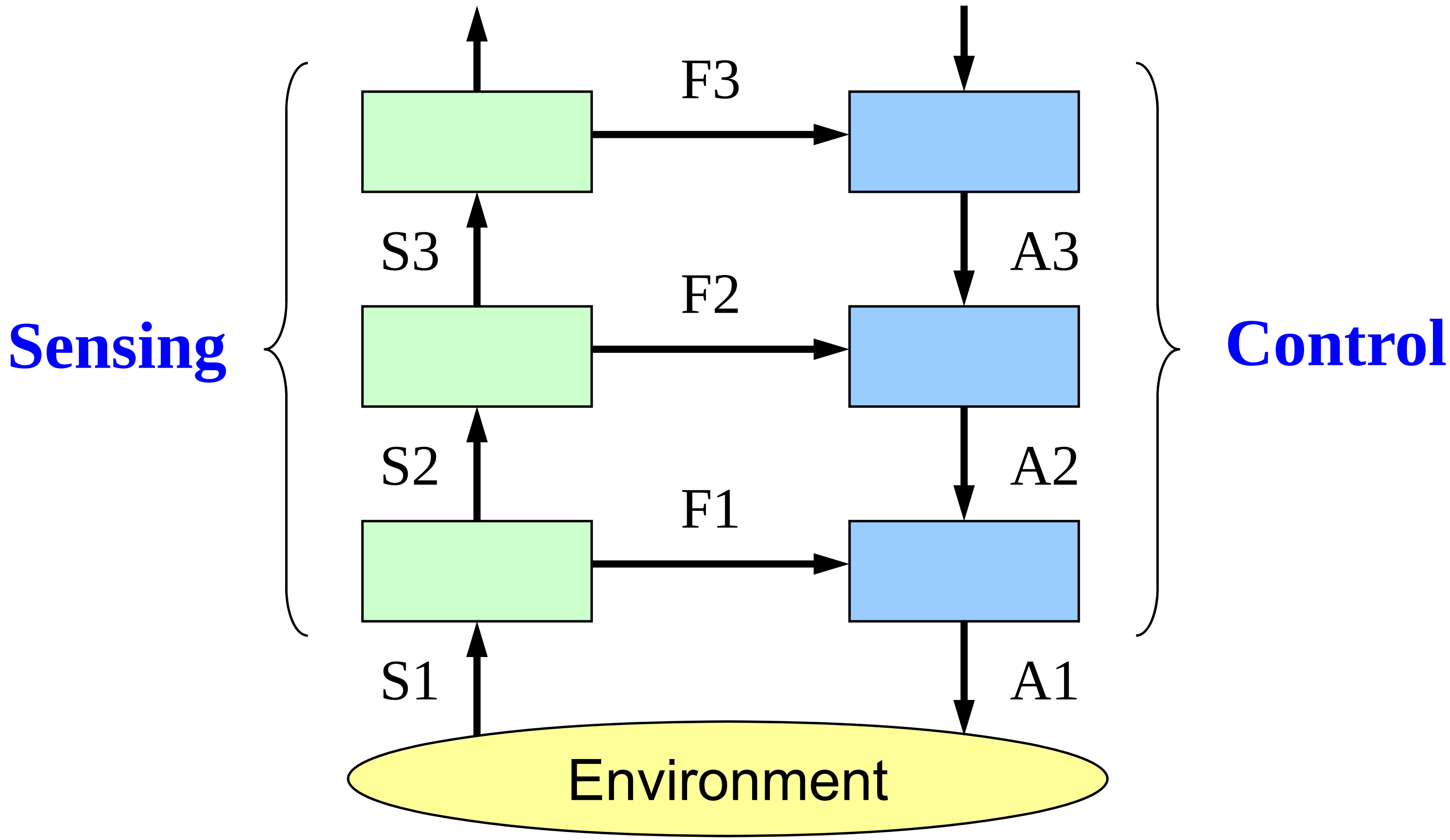
Implications

- Timing constraints are imposed by the dynamics of the environment.

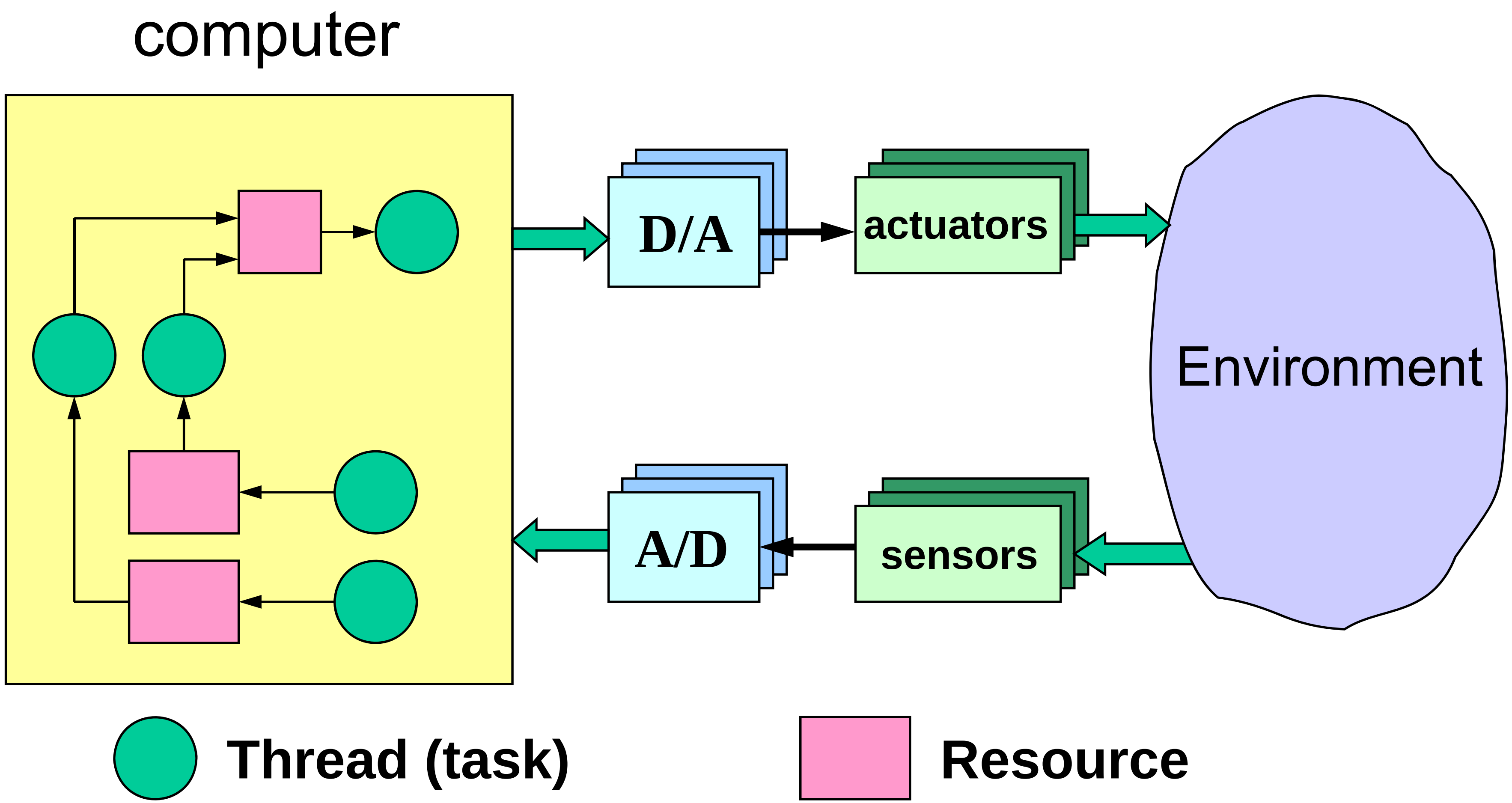
- The tight interaction with the environment requires the system to react to events within precise timing constraints.

➡ The operating system is responsible for enforcing such constraints on task execution.

Multi-level feedback control



Software Vision



Traditional Approach

- In spite of this large application domain, most of RT applications are designed using empirical techniques:
  - assembly programming
  - timing through dedicated timers
  - control through driver programming
  - priority manipulations

The resulting SW can be very efficient, but …

Disadvantages

1. Tedious programming which heavily depends on programmer's ability

2. Difficult code understanding

$$\text{Readability} \propto \frac{1}{\text{efficiency}}$$

An efficient C program

```
int a[1817];
main(z,p,q,r)
{
 for(p=80;q+p-80;p-=2*a[p])
  for(z=9;z--;)
   q=3&(r=time(0)+r*57)/7,q=q?q-1?
     q-2?1-p%79?-1:0:p%79-77?
   1:0:p<1659?79:0:p>158?-79:0,
     q?!a[p+q*2]?a[p+=a[p+=q]=q]=q :0:0;
 for(;q++-1817;)
  printf(q%79?"%c":"%c\n"," Û"[!a[q-1]]);
}
```

Disadvantages

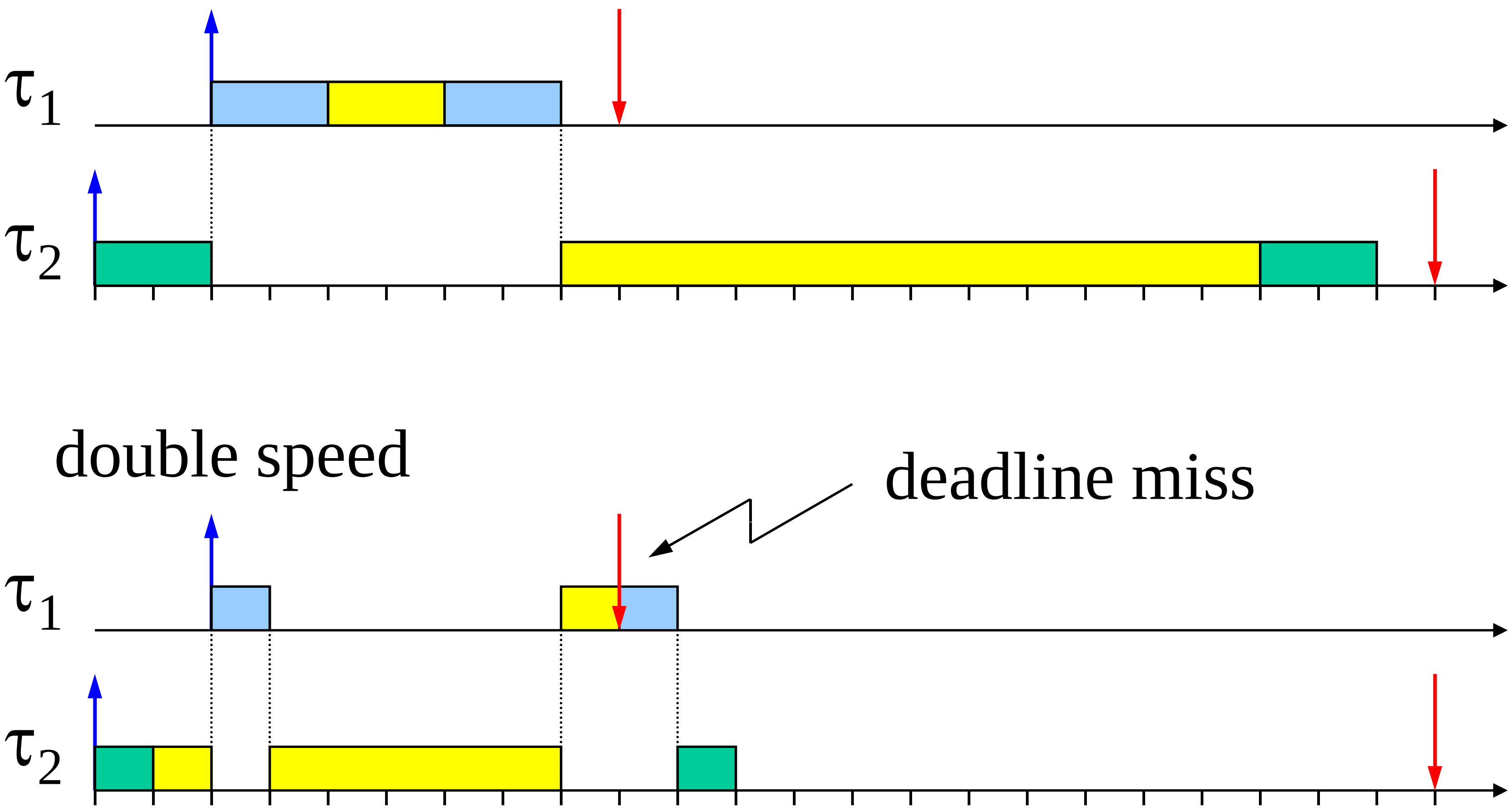
3. Difficult software maintainability
   - Complex appl.s consists of millions lines of code
   - Code understanding takes more that re-writing
   - But re-writing is VERY expensive and bug prone
4. Difficult to verify timing constraints without explicit support from the OS and the language

Implications

- Such a way of programming RT applications is very dangerous.
- It may work in most situations, but the risk of a failure is high.
- When the system fails is very difficult to understand why.

→ **low reliability**

Real-Time ≠ Fast



Speed vs. Predictability

- The objective of a real-time system is to guarantee the timing behavior of each individual task.
- The objective of a fast system is to minimize the average response time of a task set.

But …

Don't trust average when you have to guarantee individual performance

Lessons learned

- Tests are not enough for real-time systems
- Intuitive solutions do not always work
- Delay should not be used in real-time tasks

A safe approach:

♦ use predictable kernel mechanisms

♦ analyze the system to predict its behavior

Achieving predictability

- The operating system is the part most responsible for a predictable behavior.

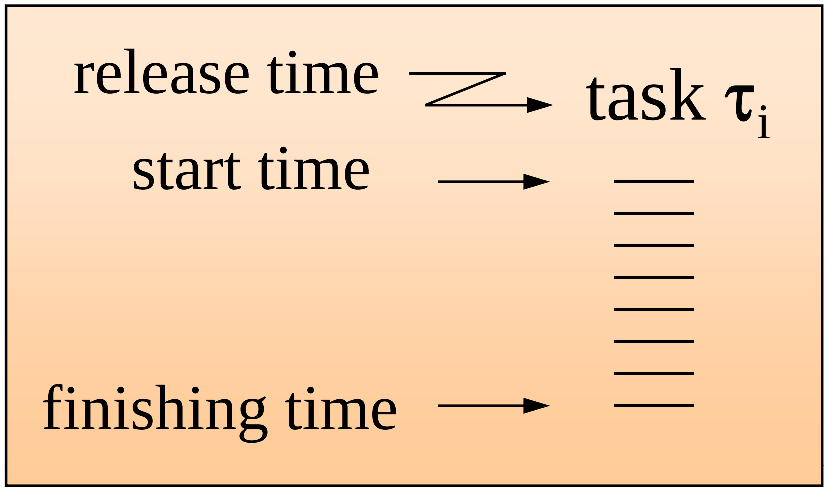
- Concurrency control must be enforced by:
  - ⇒ appropriate scheduling algorithms
  - ⇒ appropriate syncronization protocols
  - ⇒ efficient communication mechanisms
  - ⇒ predictable interrupt handling
  - ⇒ overload management

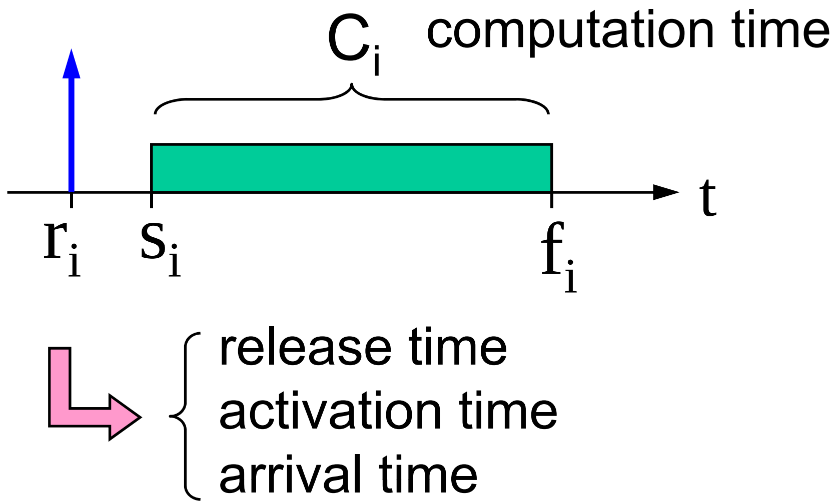
Let’s review the main scheduling results

Terminology

- **Task** (or thread)

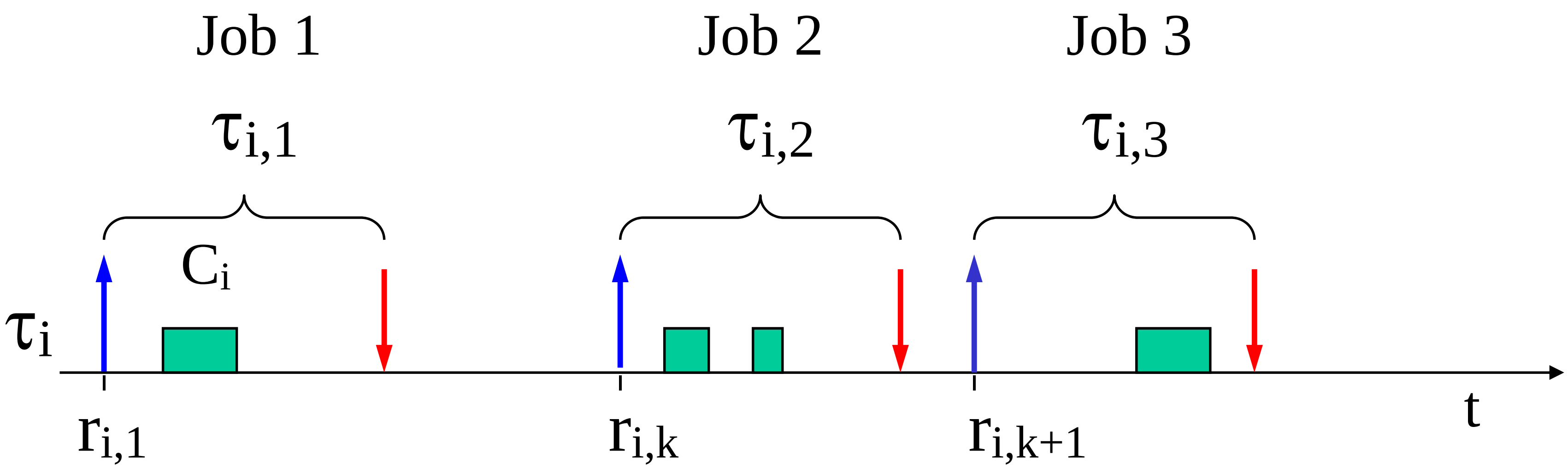
  is a sequence of instructions that in the absence of other activities is continuously executed by the processor until completion.





Tasks and jobs

A task is an infinite sequence of instances (jobs):



Activation modes

- **Time driven**: **periodic** tasks

  the task is automatically activated by the kernel at regular intervals.

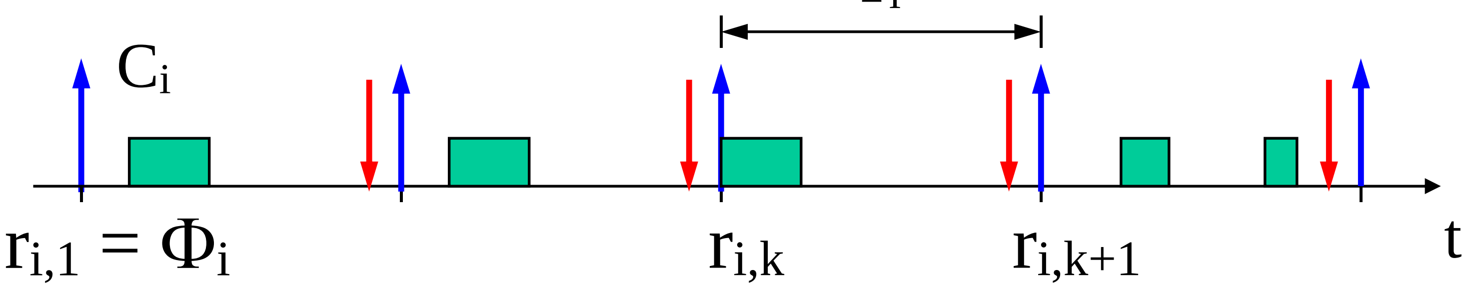
- **Event driven**: **aperiodic** tasks

  the task is activated upon the arrival of an event or through an explicit invocation of the activation primitive.

Periodic task model

$$\begin{cases} r_{i1} = \Phi_i \\ r_{i,k+1} = r_{i,k} + T_i \end{cases}$$

$\tau_i\,(C_i\,,\,T_i\,,\,D_i\,)$

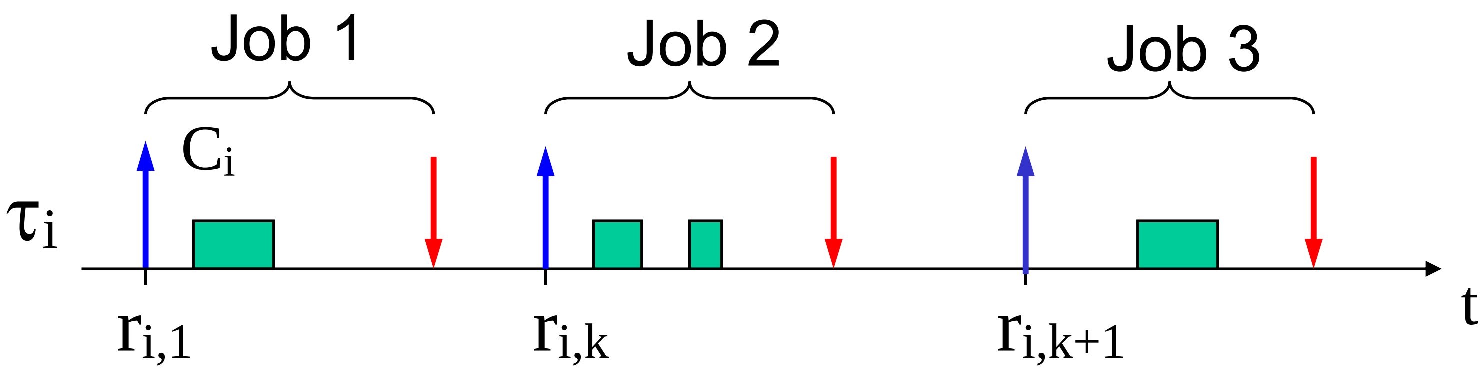


$$r_{i,k} = \Phi_i + (k-1)\,T_i$$
$$d_{i,k} = r_{i,k} + D_i$$

$$\left(\begin{array}{c} \text{often} \\ D_i = T_i \end{array} \right)$$

Aperiodic task model

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



Scheduling

- A scheduling algorithm is said to be:
  - **preemptive**: if the running task can be temporarely suspended in the ready queue to execute a more important task.
  - **non preemptive**: if the running task cannot be suspended until completion.

Schedule

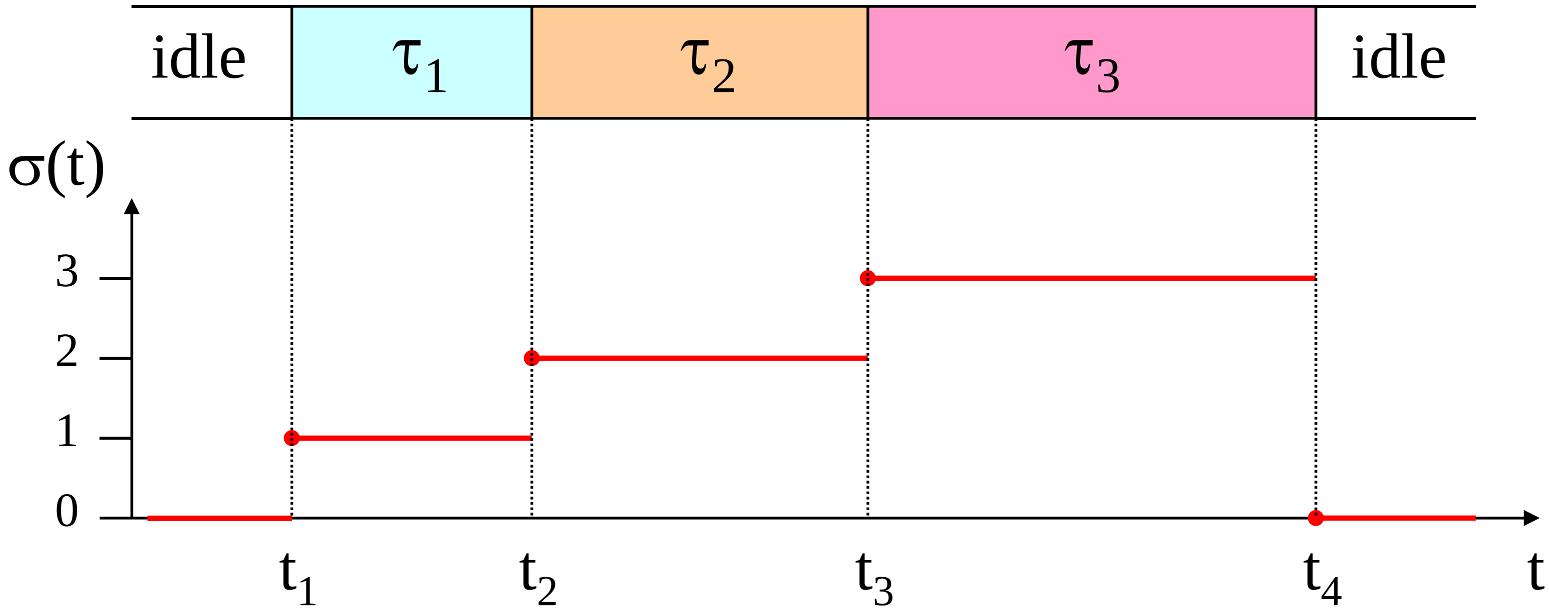
A **schedule** is a particular assignment of tasks to the processor.

Given a task set $\Gamma = \{\tau_1, \ldots, \tau_n\}$, a schedule is a mapping $\boldsymbol{\sigma : \mathbf{R}^+ \rightarrow \mathbf{N}}$ such that $\forall t \in \mathbf{R}^+, \; \exists t_1, t_2 :$

$$t \in [t_1, t_2) \quad e \quad \forall t' \in [t_1, t_2) : \sigma(t) = \sigma(t')$$

$$\sigma(t) = \begin{cases} k > 0 & \text{if } \tau_k \text{ is running} \\ 0 & \text{if the processor is idle} \end{cases}$$

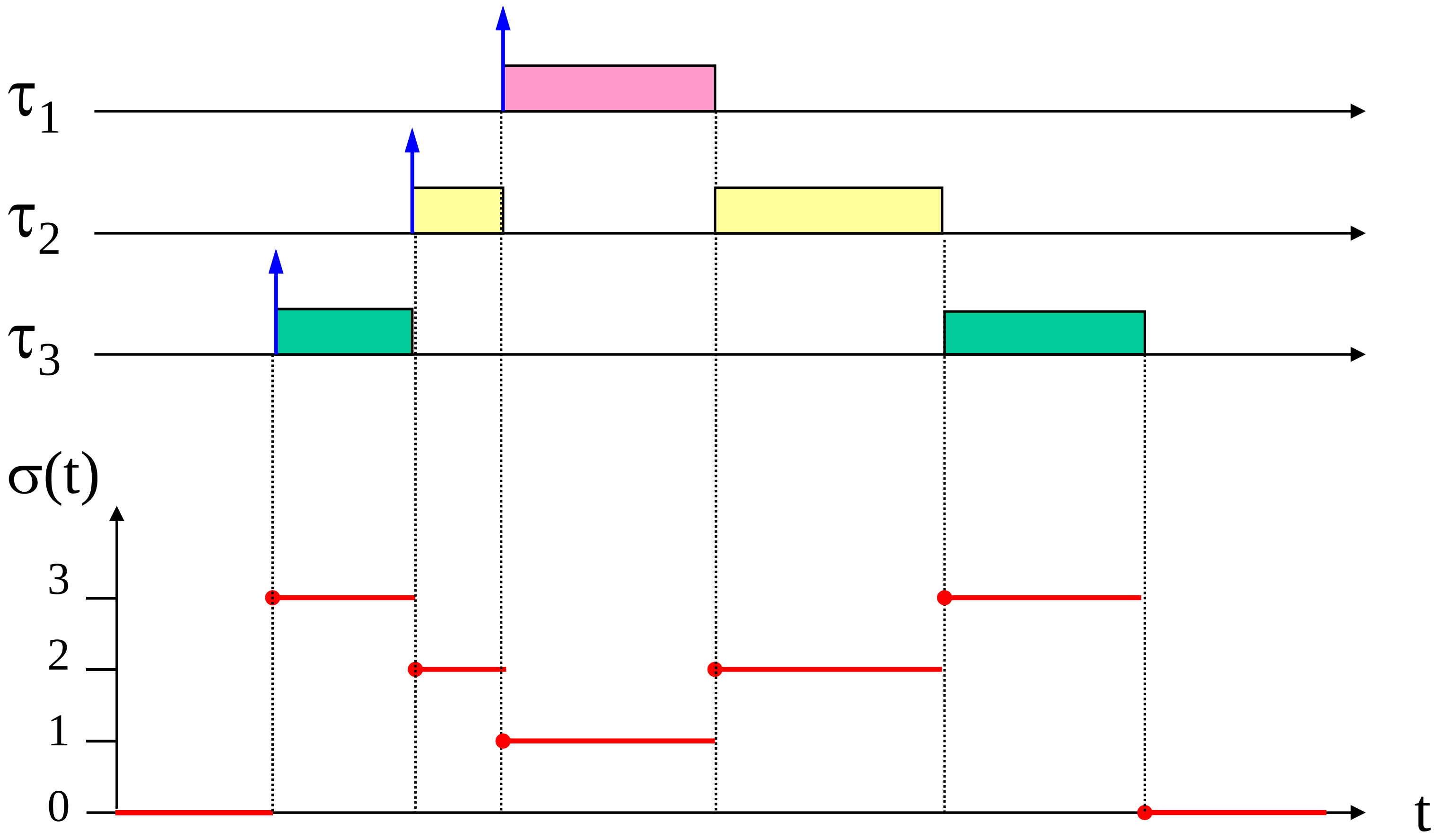
A sample schedule



At time t_1, t_2, t_3, e t_4 a **context switch** is performed.

Each interval $[t_i, t_{i+1})$ is called a **time slice**.

A preemptive schedule



Definitions

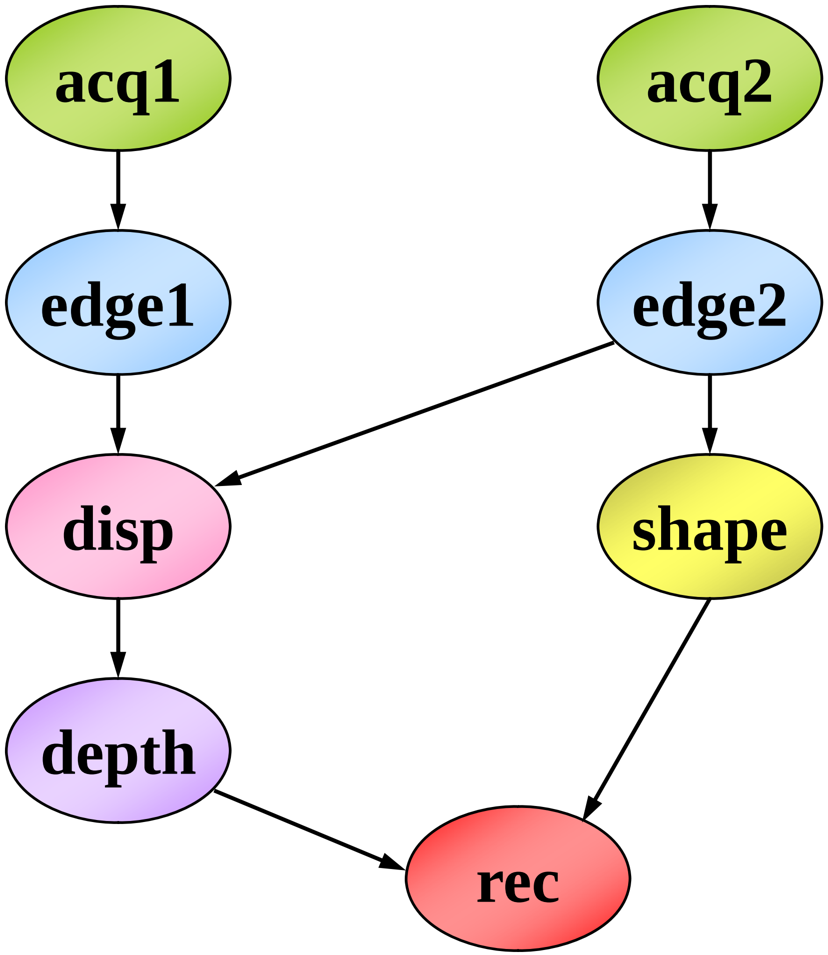
- A schedule σ is said to be **feasible** if all the tasks are able to complete within a set of constraints.

- A set of tasks Γ is said to be **schedulable** if there exists a feasible schedule for it.

Types of constraints

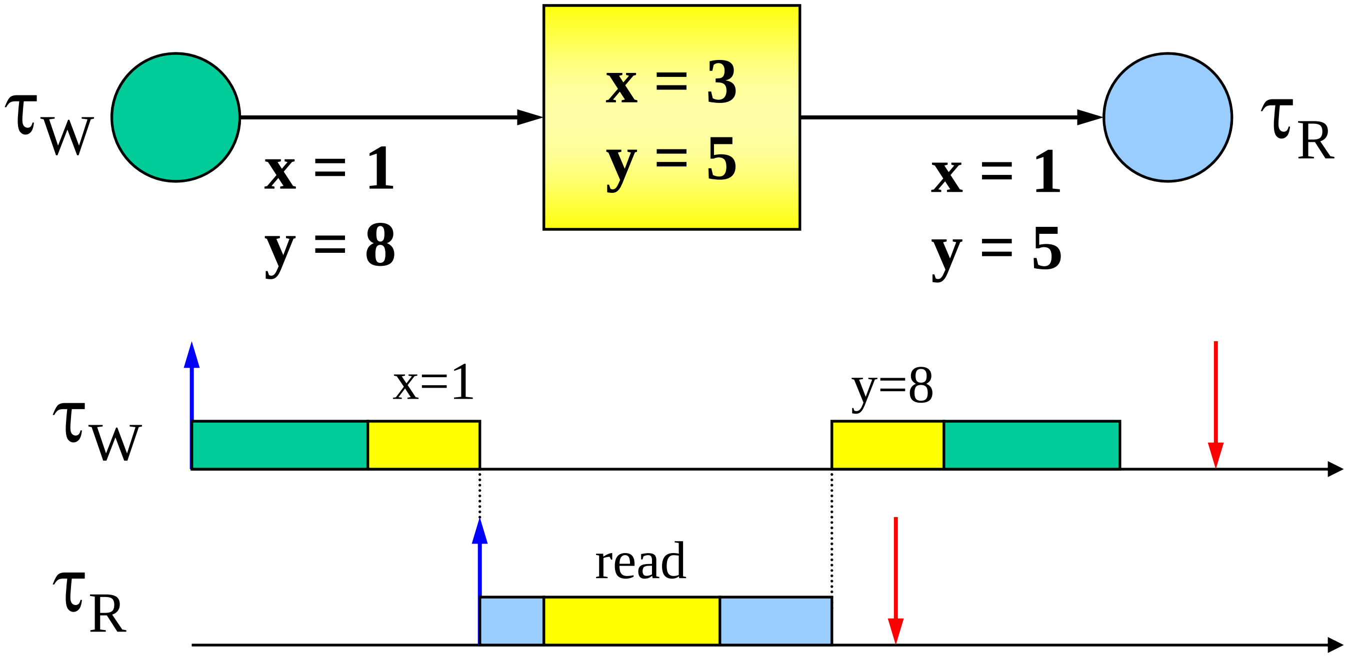
- **Timing constraints**
  - activation, completion, jitter.
- **Precedence constraints**
  - they impose an ordering in the execution.
- **Resource constraints**
  - they enforce a synchronization in the access of mutually exclusive resources.

Precedence graph



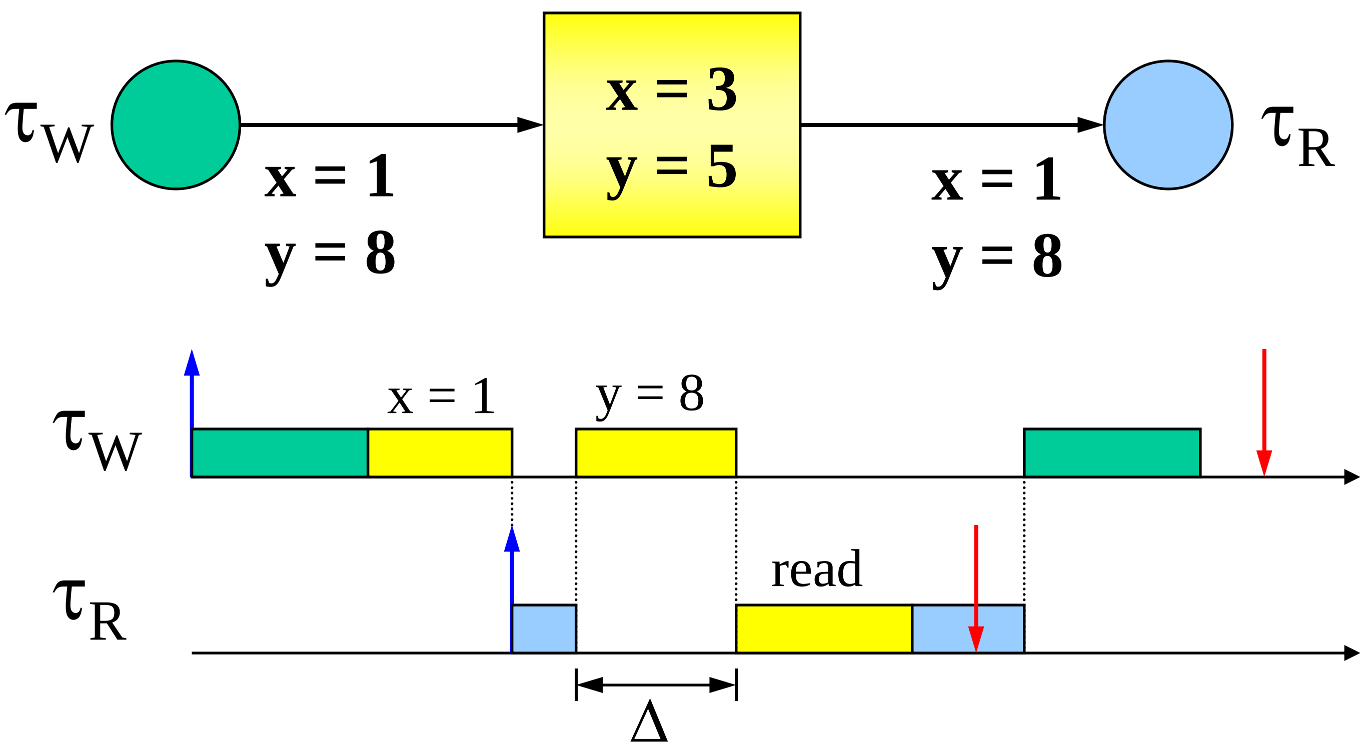
Resource constraints

To preserve data consistency, shared resources must be accessed in mutual exclusion:



Mutual exclusion

However, mutual exclusion introduces extra delays:



Timing constraints

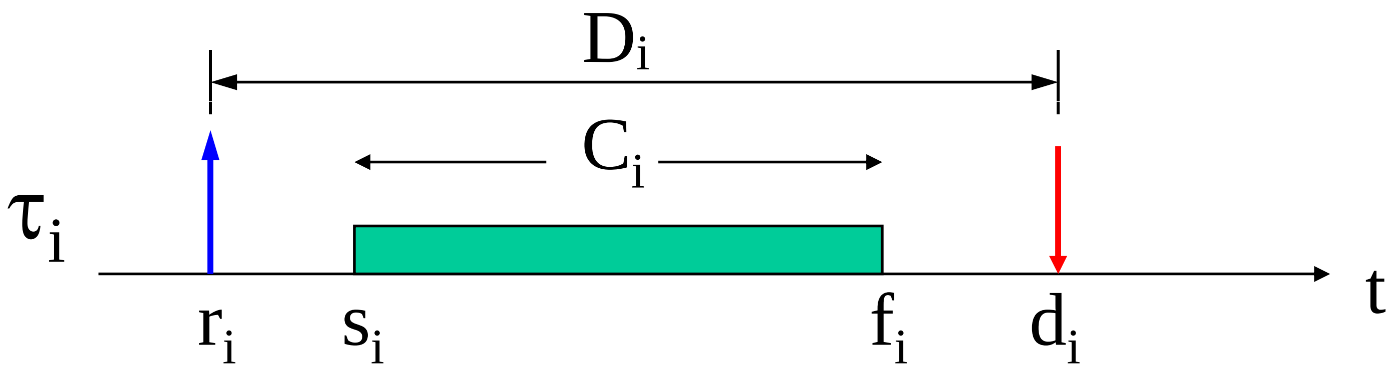
Can be explicit or implicit.

- **Explicit constraints**
  - Are included in the specification of the system activities.

Examples

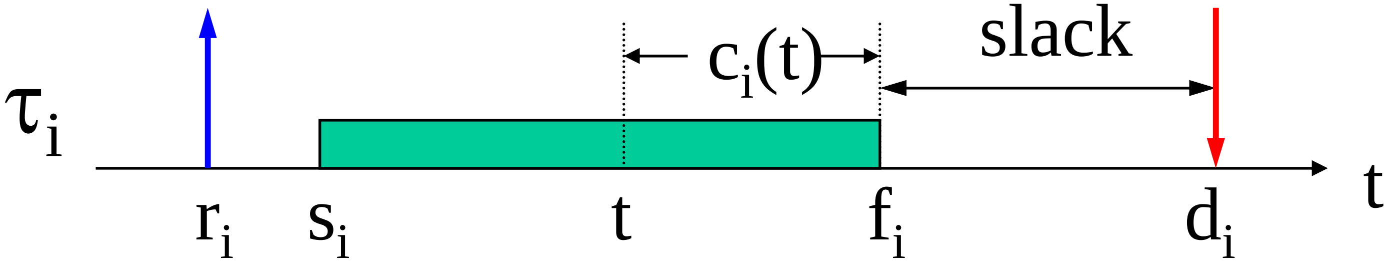
- open the valve **in** 10 seconds
- send the position **within** 40 ms
- read the altimeter **every** 200 ms

Real-Time tasks



$\mathbf{r_i}$	release time (arrival time $\mathbf{a_i}$)
$\mathbf{s_i}$	start time
$\mathbf{C_i}$	worst-case execution time (wcet)
$\mathbf{d_i}$	absolute deadline
$\mathbf{D_i}$	relative deadline
$\mathbf{f_i}$	finishing time

Other parameters



Lateness: $L_i = f_i - d_i$

Tardiness: $\max(0, L_i)$

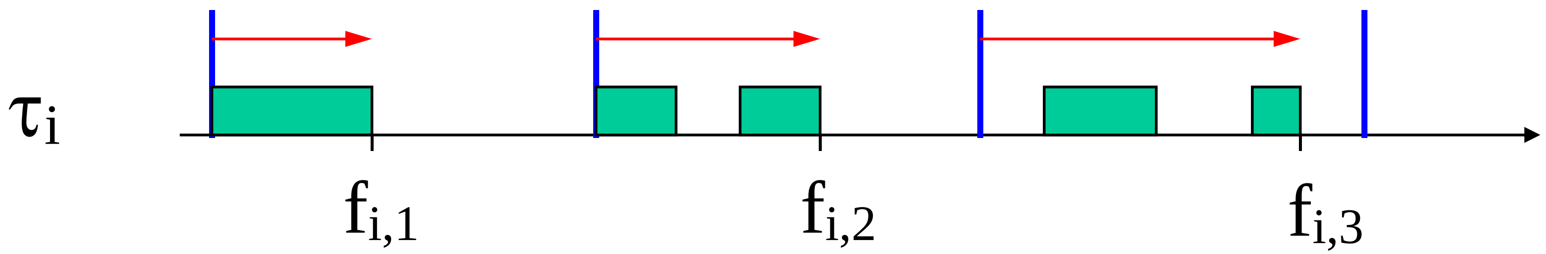
Residual wcet: $c_i(t)$ $\quad c_i(r_i) = C_i$

Laxity (o slack): $d_i - t - c_i(t)$

Jitter

It is the time variation of a periodic event:

Finishing-time Jitter

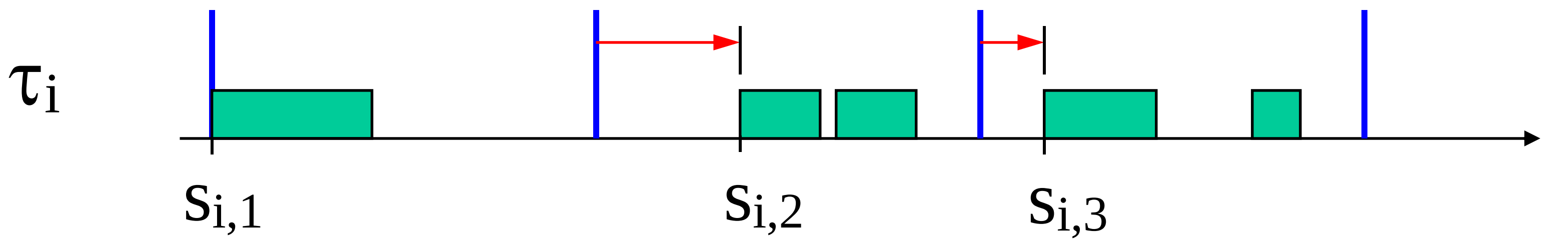


Absolute: $$\max_k (f_{i,k} - r_{i,k}) - \min_k (f_{i,k} - r_{i,k})$$

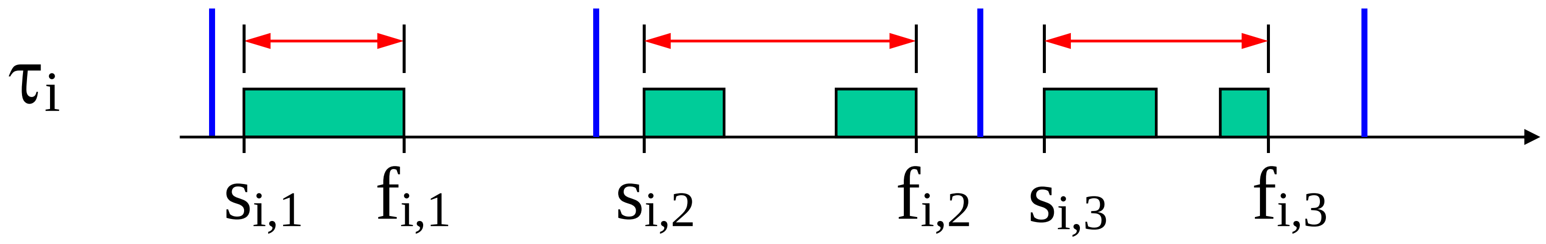
Relative: $$\max_k \left| (f_{i,k} - r_{i,k}) - (f_{i,k-1} - r_{i,k-1}) \right|$$

Other types of Jitter

Start-time Jitter



Completion-time Jitter (I/O jitter)



Task Criticality

HARD tasks

All jobs must meet their deadlines. Missing a deadline may cause catastrophical effects.

SOFT tasks

Missing deadlines is not desired but causes only a performace degradation.

An operating system able to handle hard tasks is called a **hard real-time** system.

Typical HARD tasks

- sensory acquisition
- low-level control
- sensory-motor planning

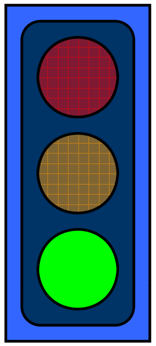
Typical SOFT tasks

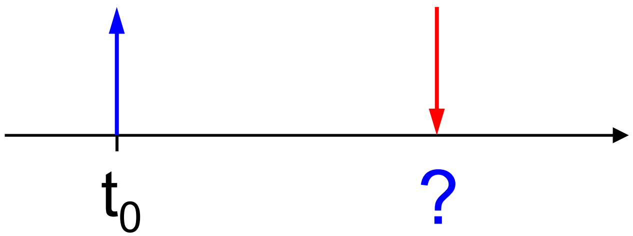
- reading data from the keyboard
- user command interpretation
- message displaying
- graphical activities

- **Implicit constraints**
  - do not appear in the system specification, but must be respected to meet the requirements.

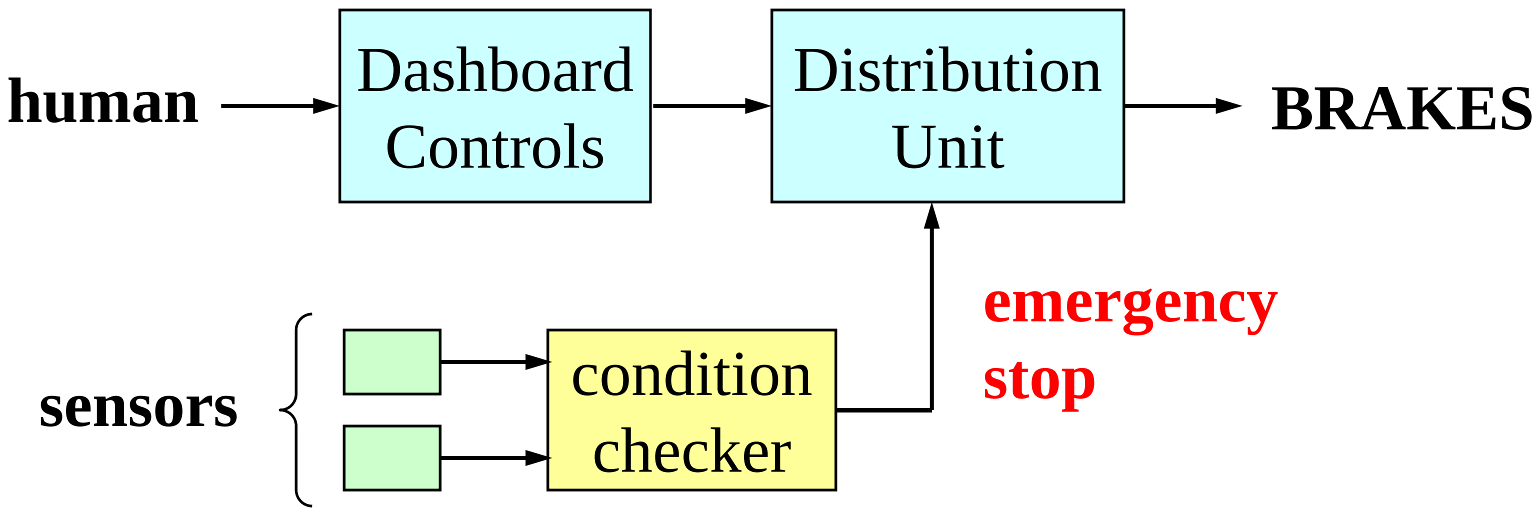
Example

What's the time validity of a sensory data?

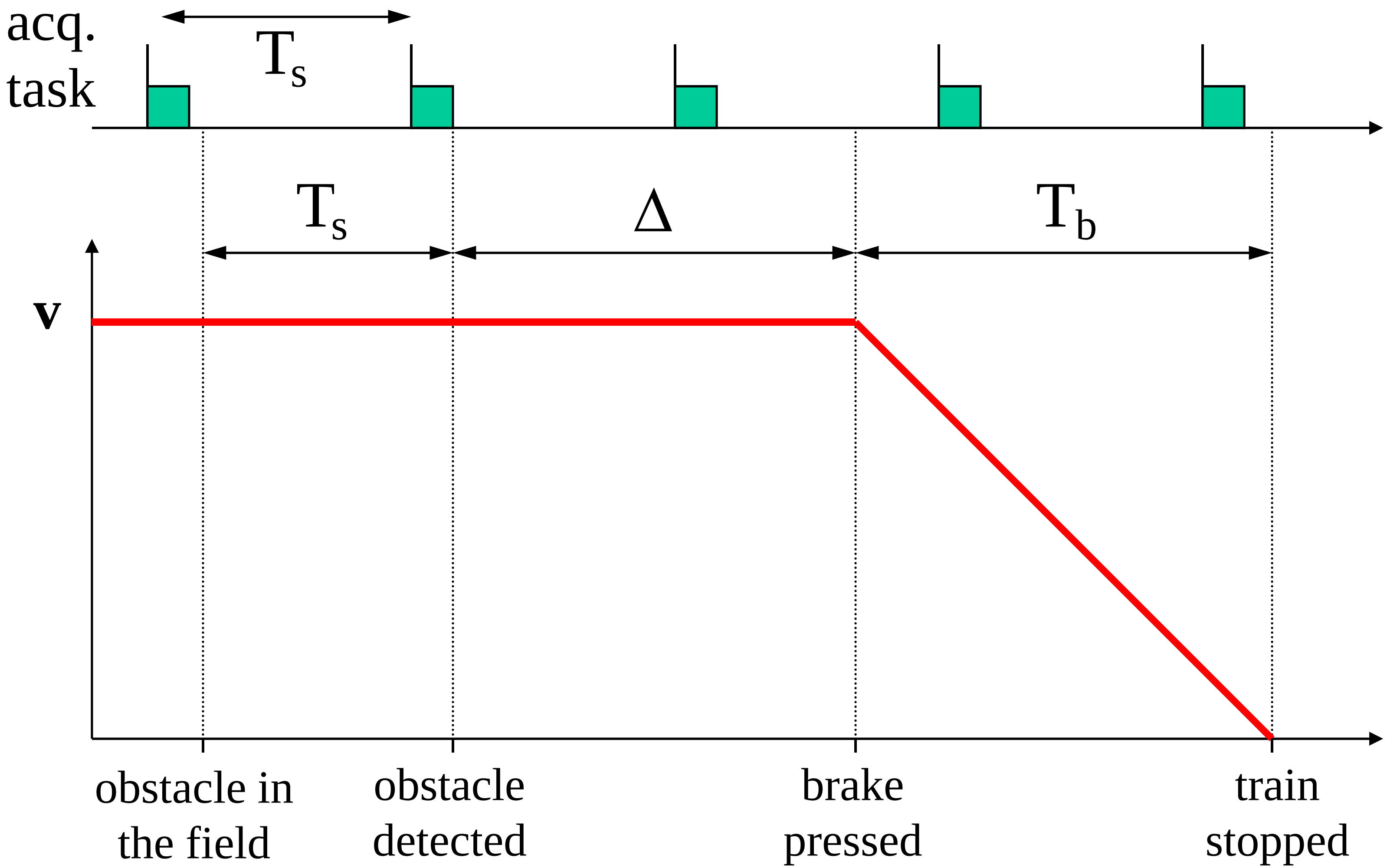




Example: automatic breaking



Worst-case reasoning



D = sensor visibility

$$\boxed{v(T_s + \Delta) + X_b < D}$$

$$\begin{cases} X_b = vt - \frac{1}{2}at^2 \\ v = a\,t \end{cases}$$

$$a = \mu g$$

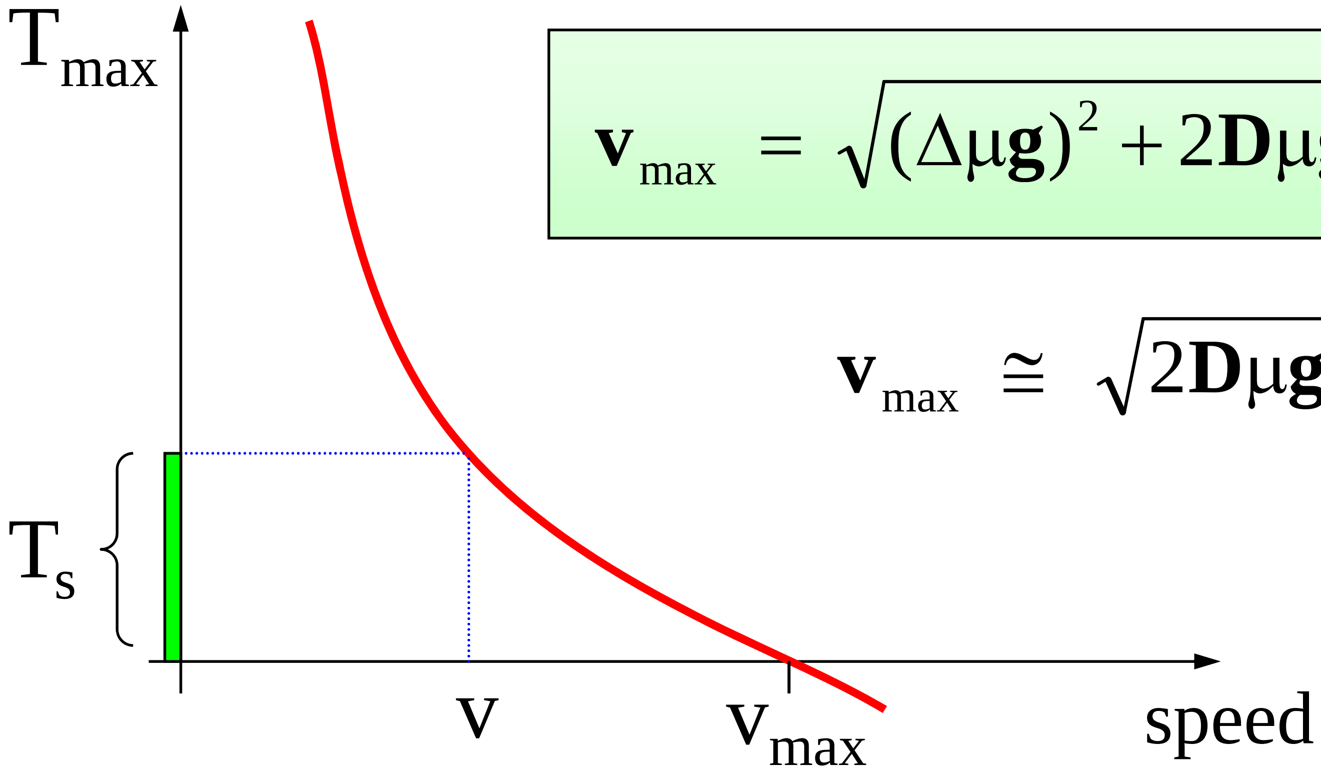
$$\boxed{X_b = \frac{v^2}{2\mu g}}$$

$$\boxed{v(T_s + \Delta) + \frac{v^2}{2\mu g} < D}$$

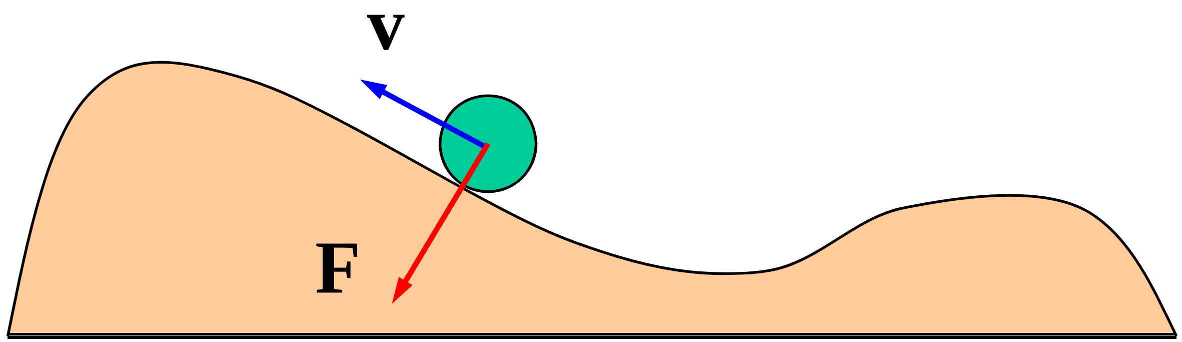
$$T_s < \frac{D}{v} - \frac{v}{2\mu g} - \Delta$$

$$\mathbf{v}_{\max} = \sqrt{(\Delta\mu\mathbf{g})^2 + 2\mathbf{D}\mu\mathbf{g}} - \Delta\mu\mathbf{g}$$

$$\mathbf{v}_{\max} \cong \sqrt{2\mathbf{D}\mu\mathbf{g}}$$



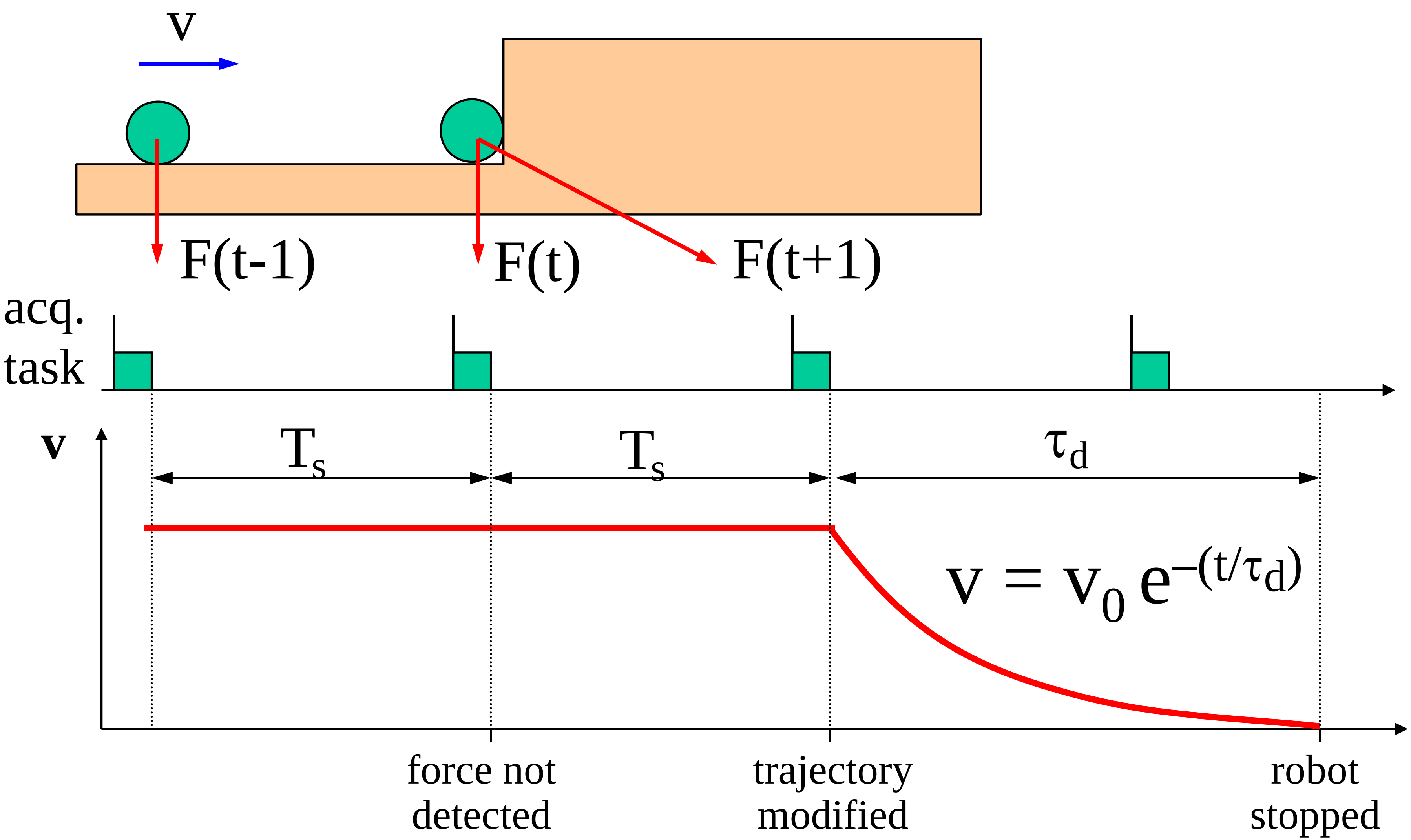
Example 2: contour following



Goal

Move at velocity **v** along the surface tangent, exerting a force **$F < F_{max}$** along its normal direction.

Worst-case reasoning



Lenght covered by the robot after the contact:

$$L = vT_s + x_f$$

$$x_f = \int_0^\infty v(t)dt = \int_0^\infty v_0 e^{-t/\tau_d} dt = -v_0 \tau_d (e^{-\infty} - e^0) = v_0 \tau_d$$

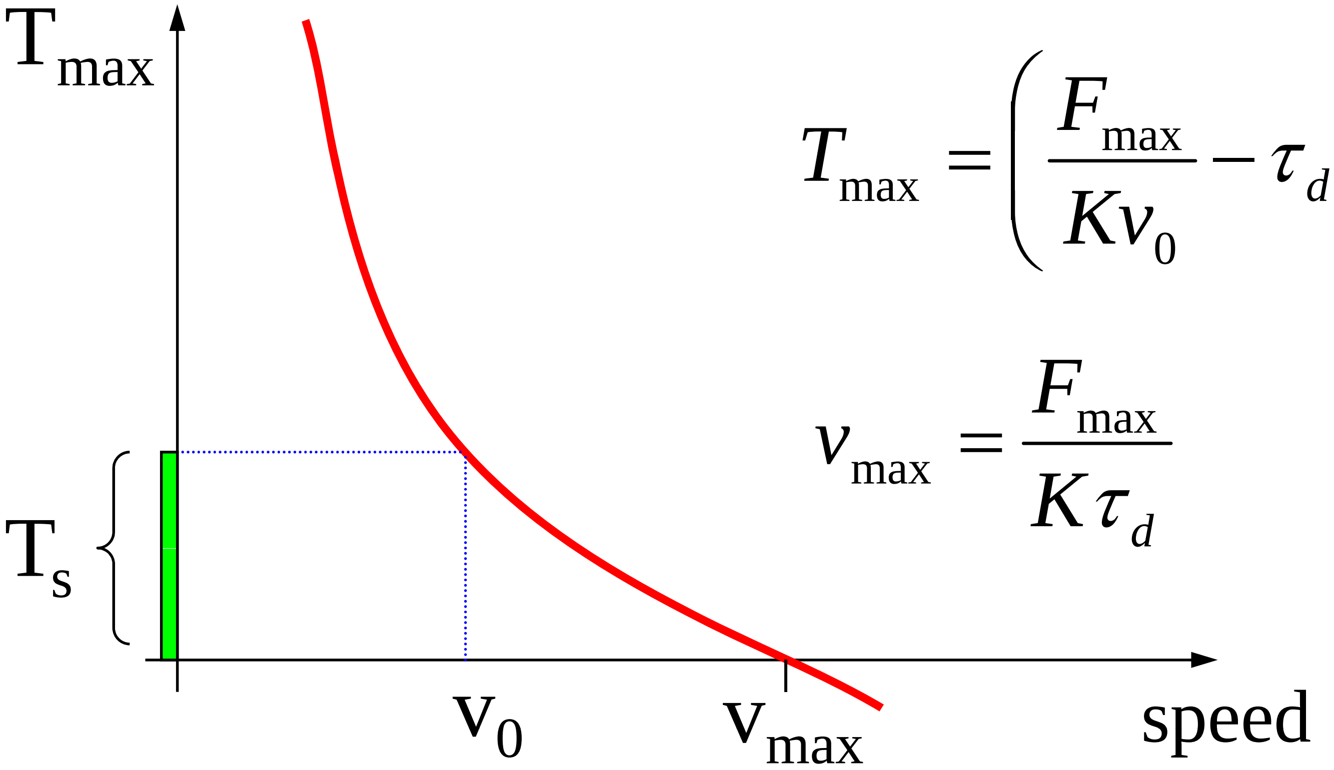
$$L = v(\mathrm{T_s} + \tau_\mathrm{d})$$

Force on the robot tool: (K = elastic coefficient)

$$F = KL = v(\mathrm{T_s} + \tau_\mathrm{d}) < \mathrm{F_{max}}$$

Condition on the sampling period:

$$\boxed{T_s < \frac{F_{max}}{K v_0} - \tau_d}$$

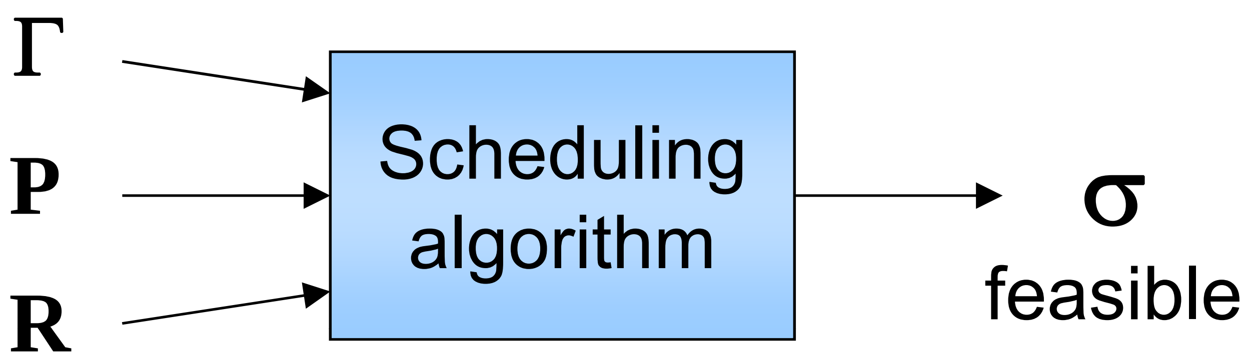


$$T_{max} = \left(\frac{F_{max}}{K v_0} - \tau_d \right)$$

$$v_{max} = \frac{F_{max}}{K \tau_d}$$

The general scheduling problem

Given a set Γ of n tasks, a set **P** of m processors, and a set **R** of r resources, find an assignment of **P** and **R** to Γ which produces a feasible schedule.



Complexity

- In 1975, Garey and Johnson showed that the general scheduling problem is NP hard.
- However, polynomial time algorithms can be found under particular conditions.

Complexity

It's important to find polynomial time algorithms.

number of tasks $n = 30$
elementary step $= 1\mu s$

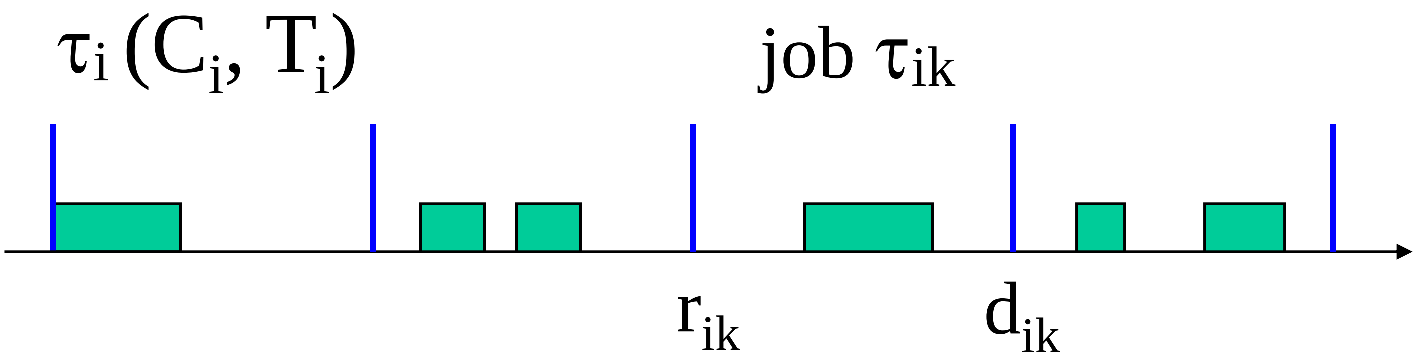
- Alg. 1: $O(n)$ **30 μs**
- Alg. 2: $O(n^6)$ **12 min**
- Alg. 3: $O(6^n)$ **7 billions of years**

Simplifying assumptions

- Single processor
- Omogeneous task sets
- Fully preemptive tasks
- Simultanoeus activations
- No precedence constraints
- No resource constraints

Periodic Task Scheduling

Problem formulation



For each periodic task, guarantee that:

- each job τ_{ik} is activated at $r_{ik} = (k-1)T_i$
- each job τ_{ik} completes within $d_{ik} = r_{ik} + D_i$

Timeline Scheduling (cyclic scheduling)

It has been used for 30 years in military systems, navigation, and monitoring systems.

Examples

- Air traffic control
- Space Shuttle
- Boeing 777

Timeline Scheduling

Method

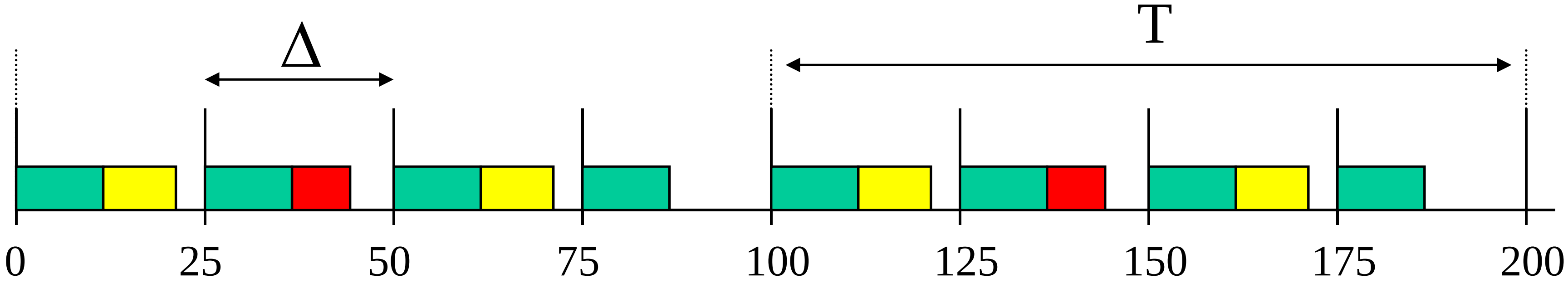
- The time axis is divided in intervals of equal length (***time slots***).
- Each task is statically allocated in a slot in order to meet the desired request rate.
- The execution in each slot is activated by a timer.

Example

task	f	T
A	40 Hz	25 ms
B	20 Hz	50 ms
C	10 Hz	100 ms

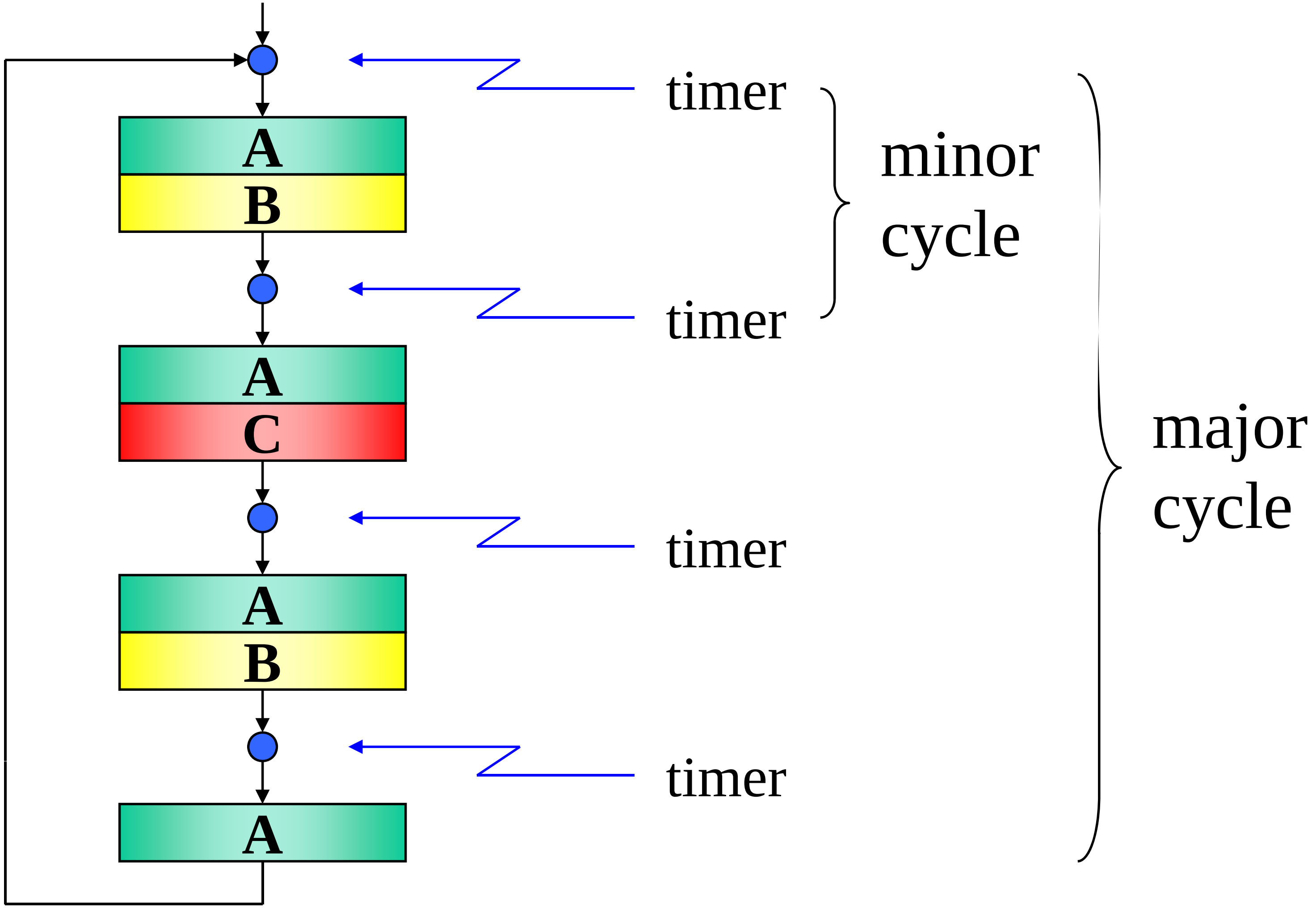
Δ = GCD (minor cycle)

T = lcm (major cycle)



Guarantee: $\begin{cases} C_A + C_B \leq \Delta \\ C_A + C_C \leq \Delta \end{cases}$

Implementation



Timeline scheduling

Advantages

- Simple implementation (no real-time operating system is required).
- Low run-time overhead.
- It allows jitter control.

Timeline scheduling

Disadvantages

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

Problems during overloads

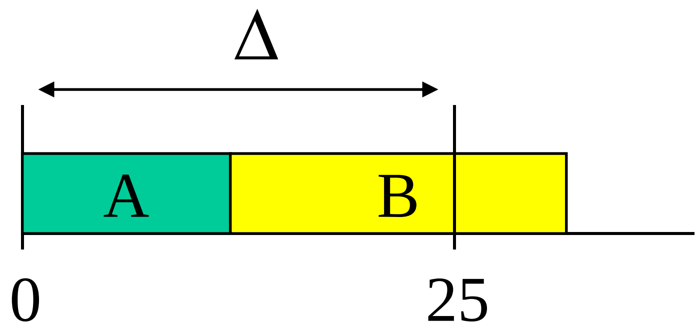
What do we do during task overruns?

- Let the task continue
  - we can have a ***domino effect*** on all the other tasks (timeline break)
- Abort the task
  - the system can remain in inconsistent states.

Expandibility

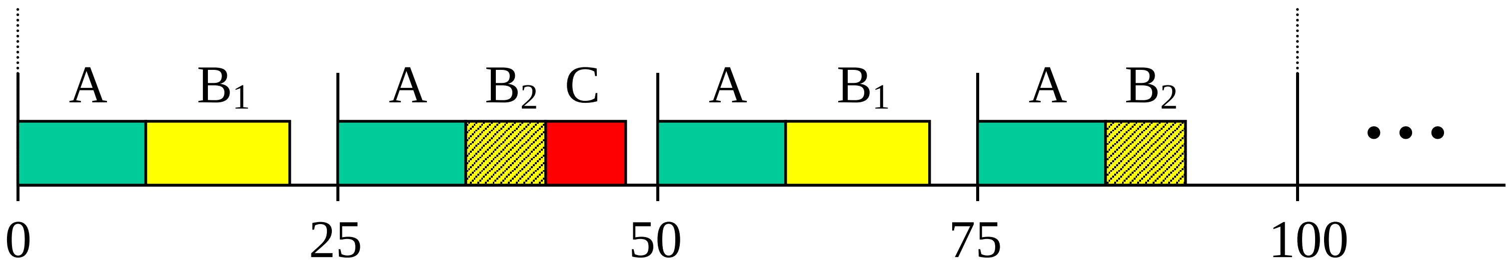
If one or more tasks need to be upgraded, we may have to re-design the whole schedule again.

Example: B is updated but $C_A + C_B > \Delta$



Expandibility

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



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

Expandibility

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

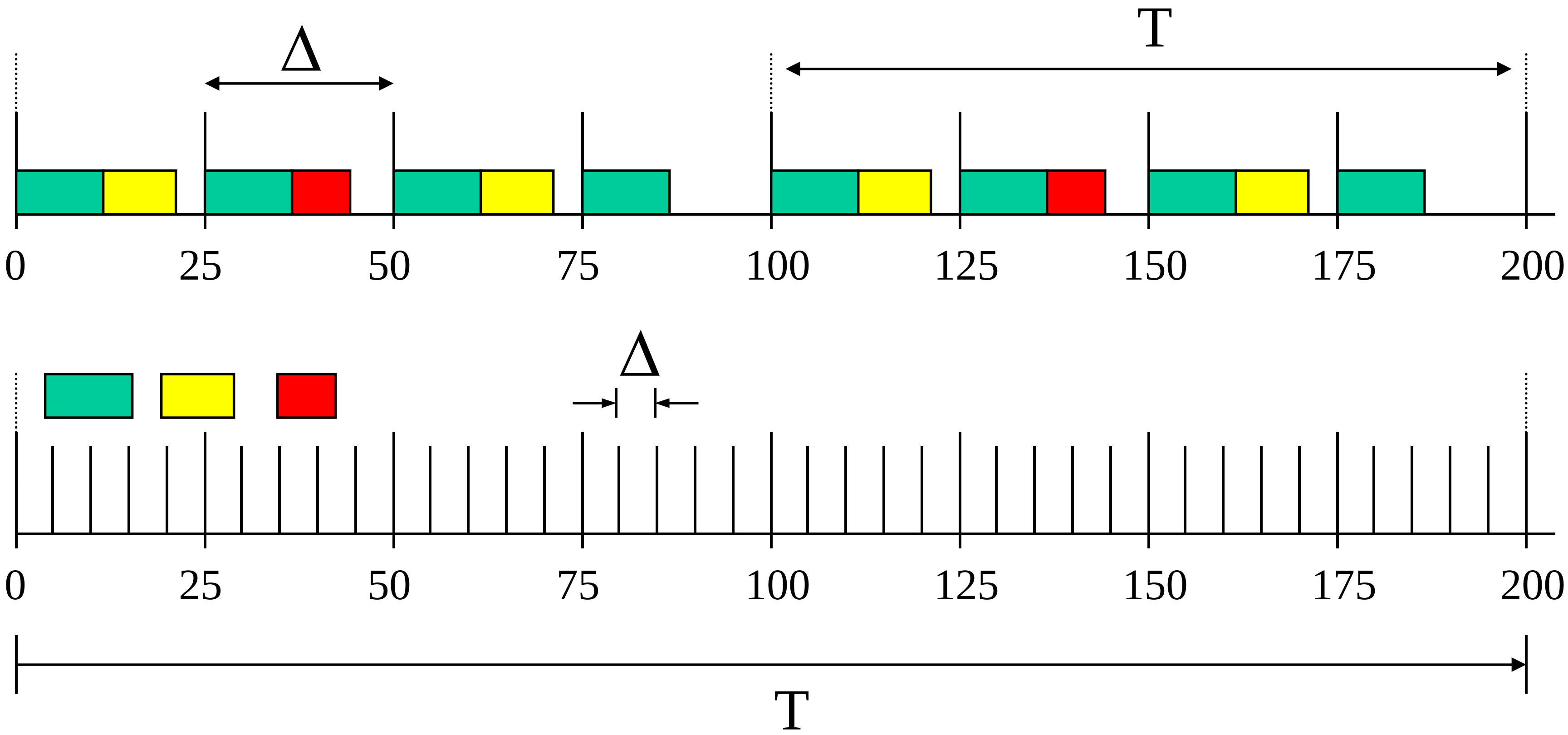
task	T	T
A	25 ms	25 ms
B	50 ms	**40 ms**
C	100 ms	100 ms
	before	**after**

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

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

40 sync. per cycle!

Example



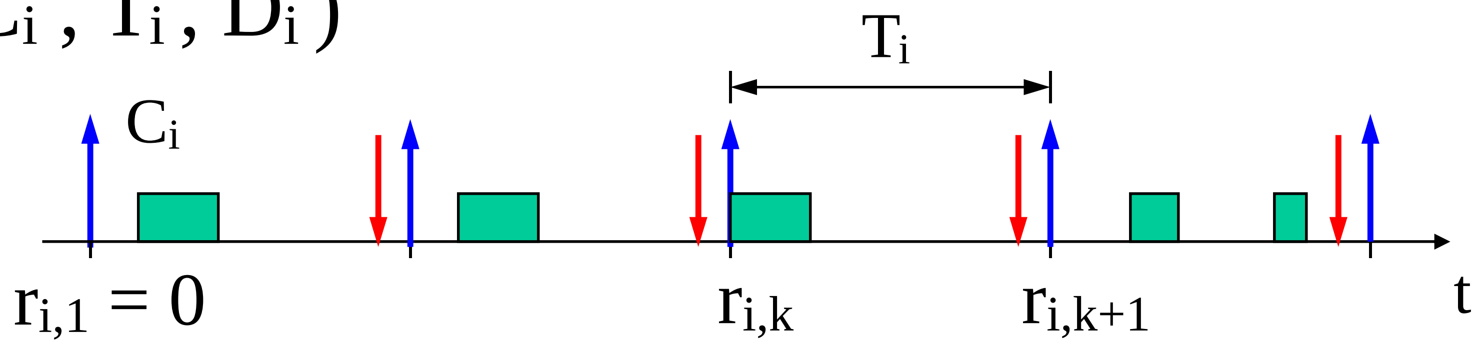
Priority Scheduling

Method

- Each task is assigned a priority based on its timing constraints.
- We verify the feasibility of the schedule using analytical techniques.
- Tasks are executed on a priority-based kernel.

Priority Assignments

$\tau_i\,(C_i\,,\,T_i\,,\,D_i\,)$



$D_i = T_i$

- **Rate Monotonic (RM):**

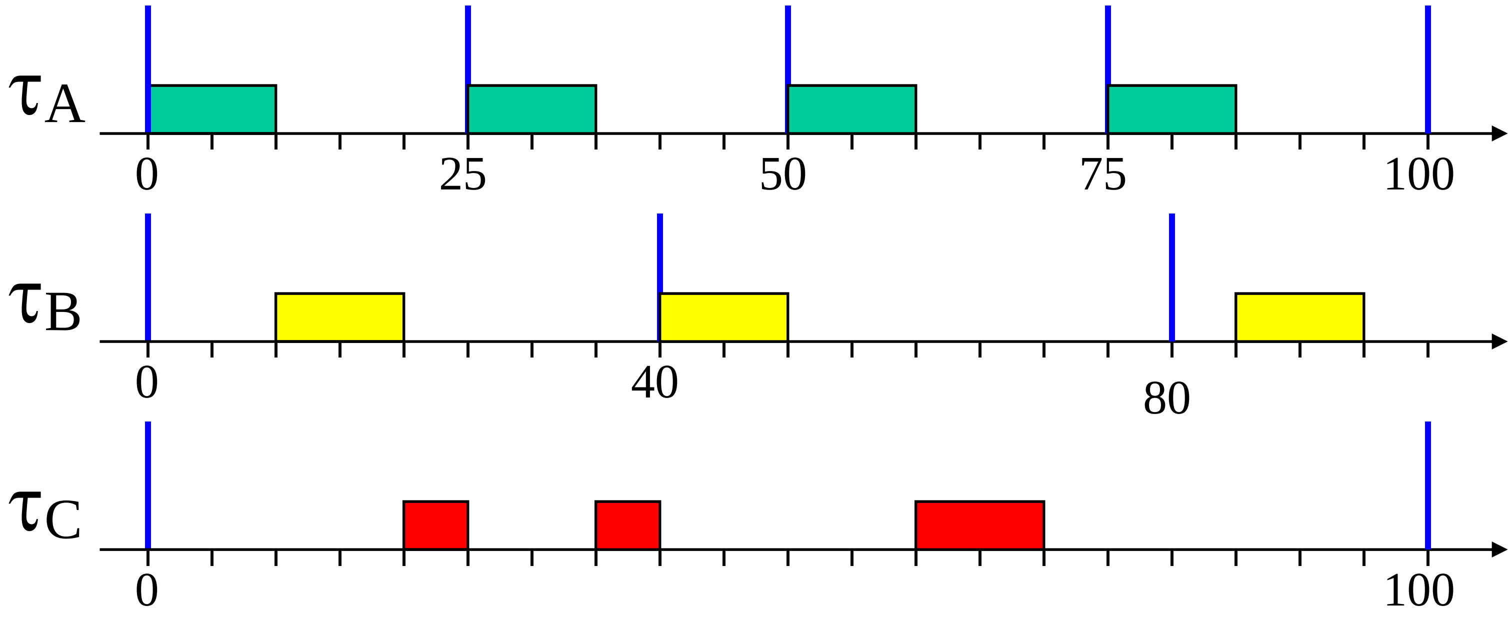
  $\mathbf{p_i \propto 1/T_i}$ (static)

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

  $\mathbf{p_i \propto 1/d_i}$ (dynamic) $d_{i,k} = r_{i,k} + D_i$

Rate Monotonic (RM)

- Each task is assigned a <u>fixed</u> priority proportional to its rate.



How can we verify feasibility?

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

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

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

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

- U_p is a misure of the **processor load**

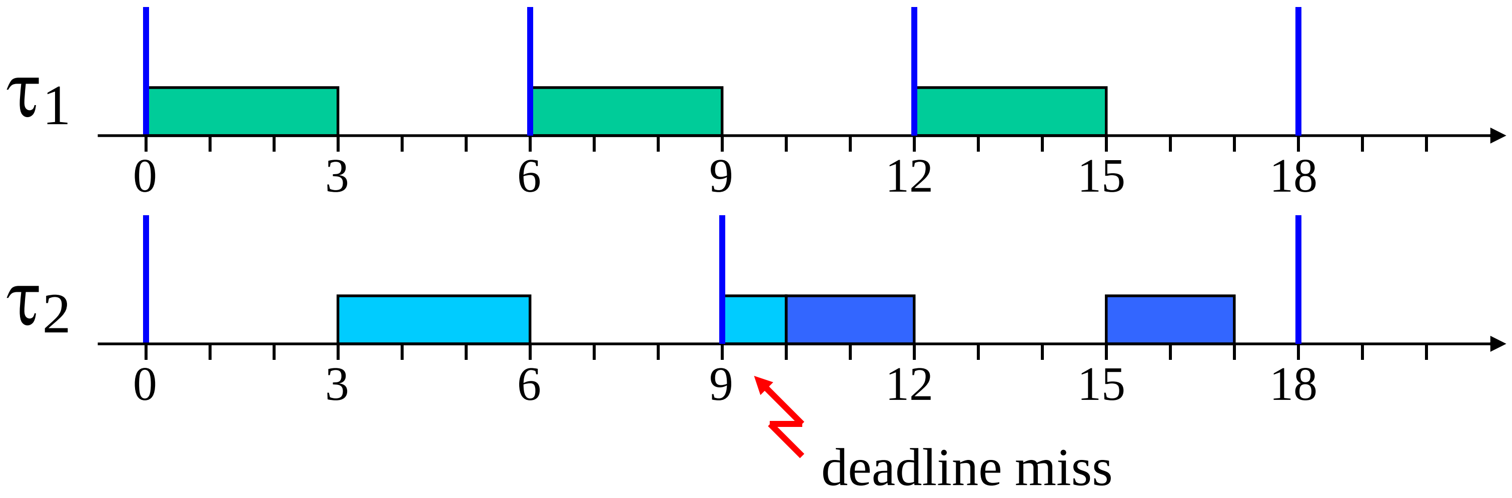
A necessary condition

If $\mathbf{U_p > 1}$ the processor is overloaded hence the task set cannot be schedulable.

However, there are cases in which $\mathbf{U_p < 1}$ but the task is not schedulable by RM.

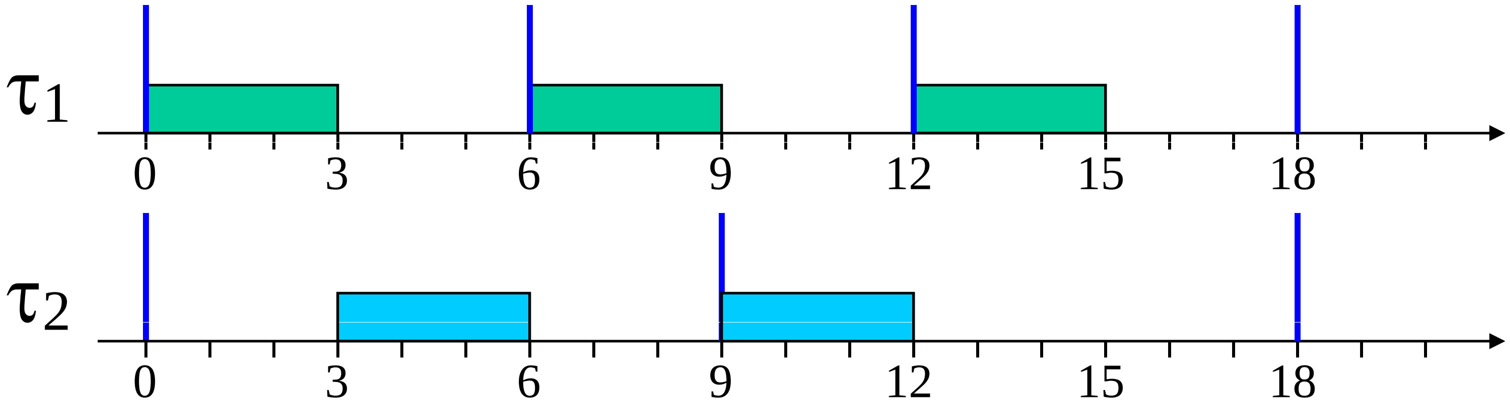
An unfeasible RM schedule

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.944$$



Utilization upper bound

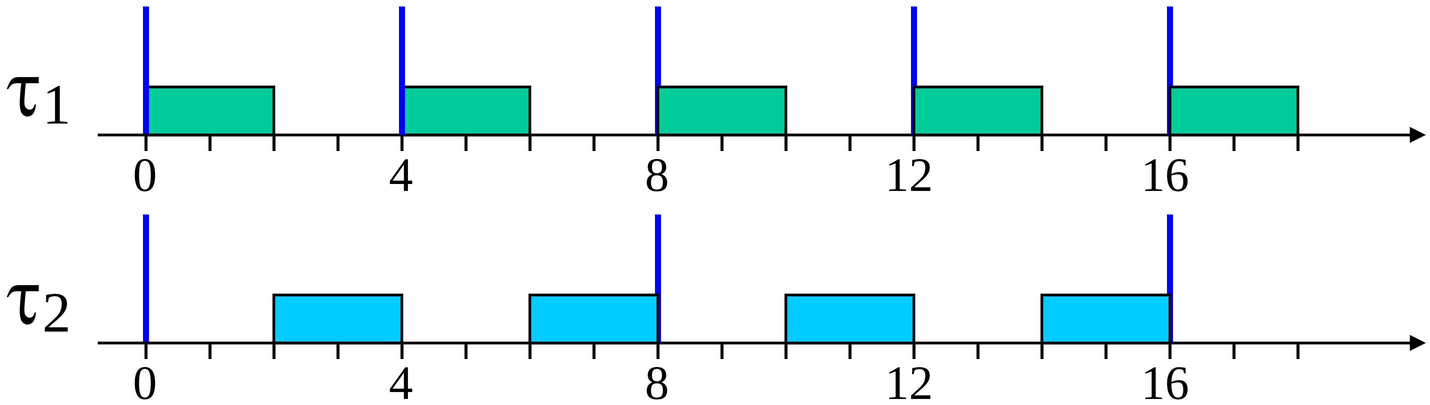
$$U_p = \frac{3}{6} + \frac{3}{9} = 0.833$$



NOTE: If C_1 or C_2 is increased, τ_2 will miss its deadline!

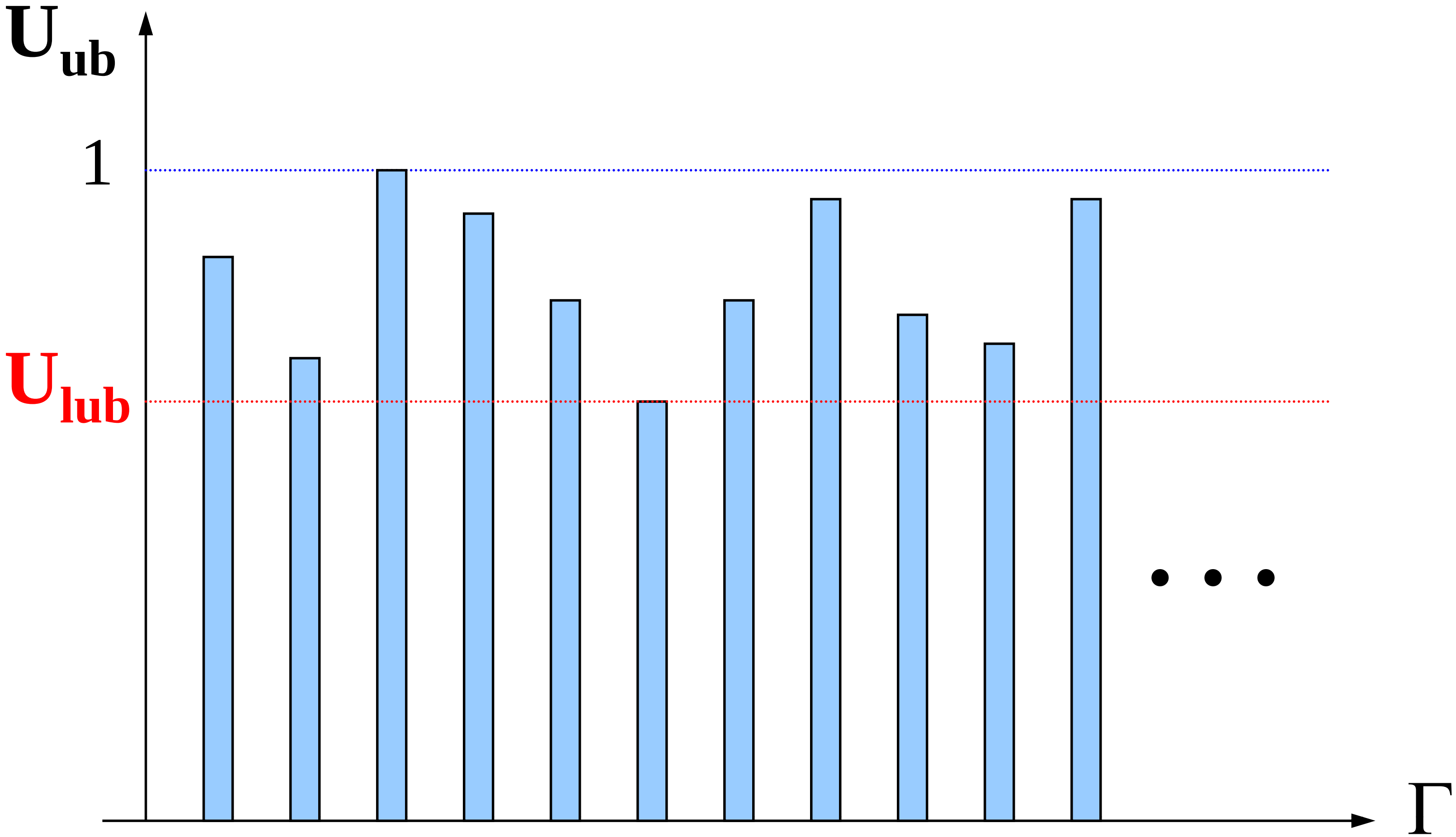
A different upper bound

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



The upper bound U_{ub} depends on the specific task set.

The least upper bound



A sufficient condition

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

NOTE

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

Basic results

Assumptions:
- Independent tasks
- $\Phi_i = 0$ $\quad D_i = T_i$

In 1973, Liu & Layland proved that a set of ***n*** periodic tasks can be feasibly scheduled

under RM **if**

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

under EDF **if and only if**

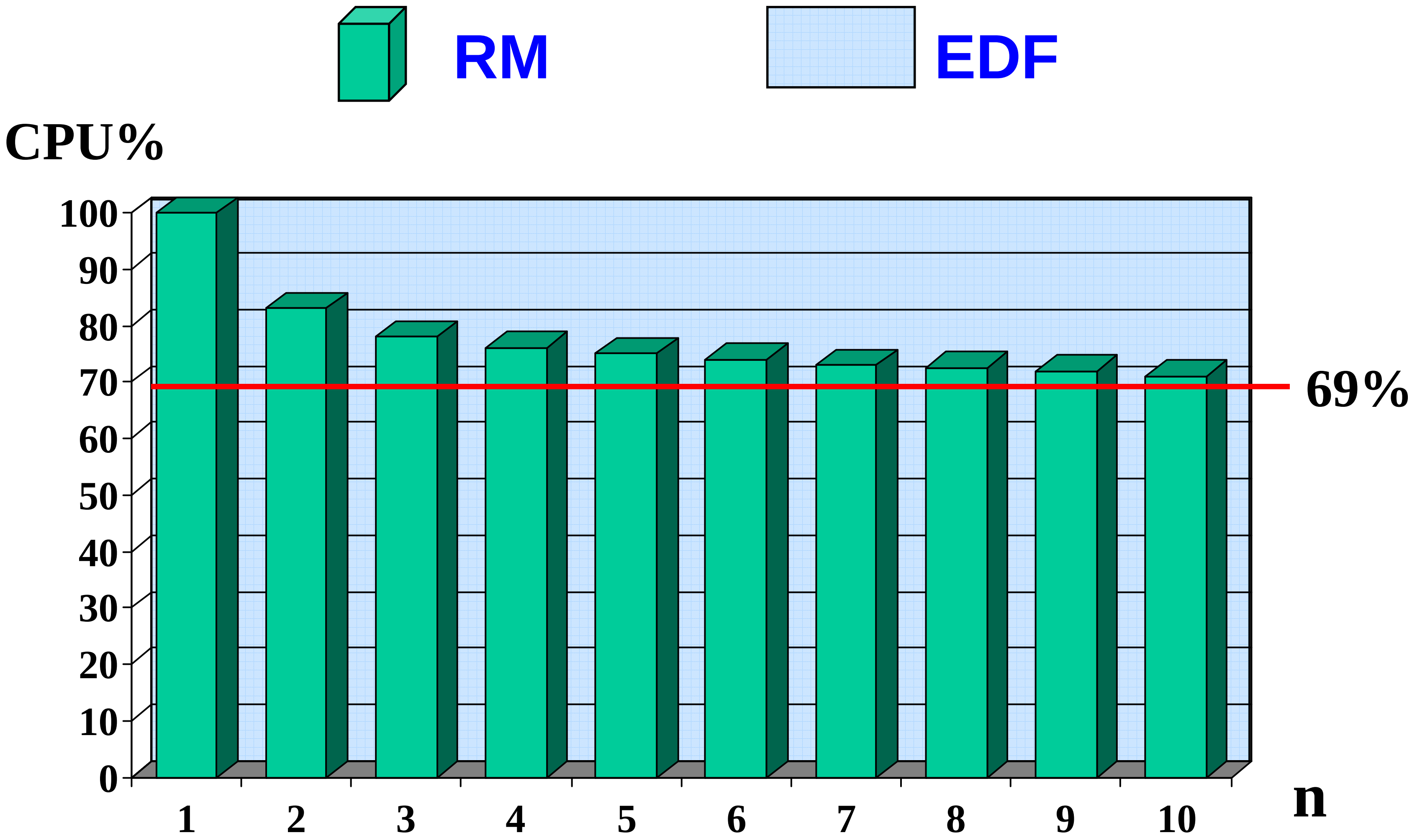
$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

RM bound for large *n*

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

for n $\rightarrow \infty$ $\quad$ $U_{\text{lub}} \rightarrow$ *ln 2*

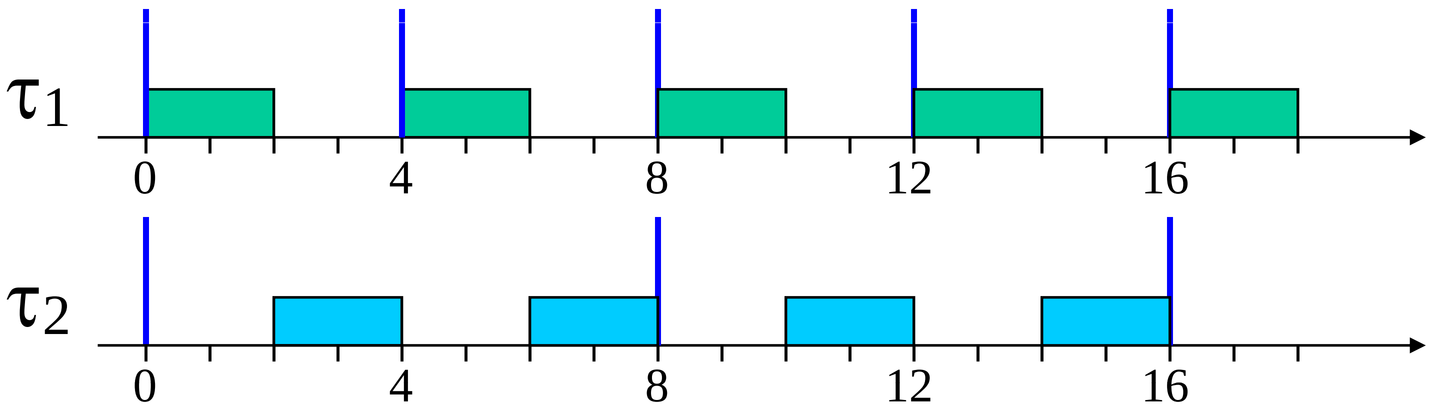
Schedulability bound



A special case

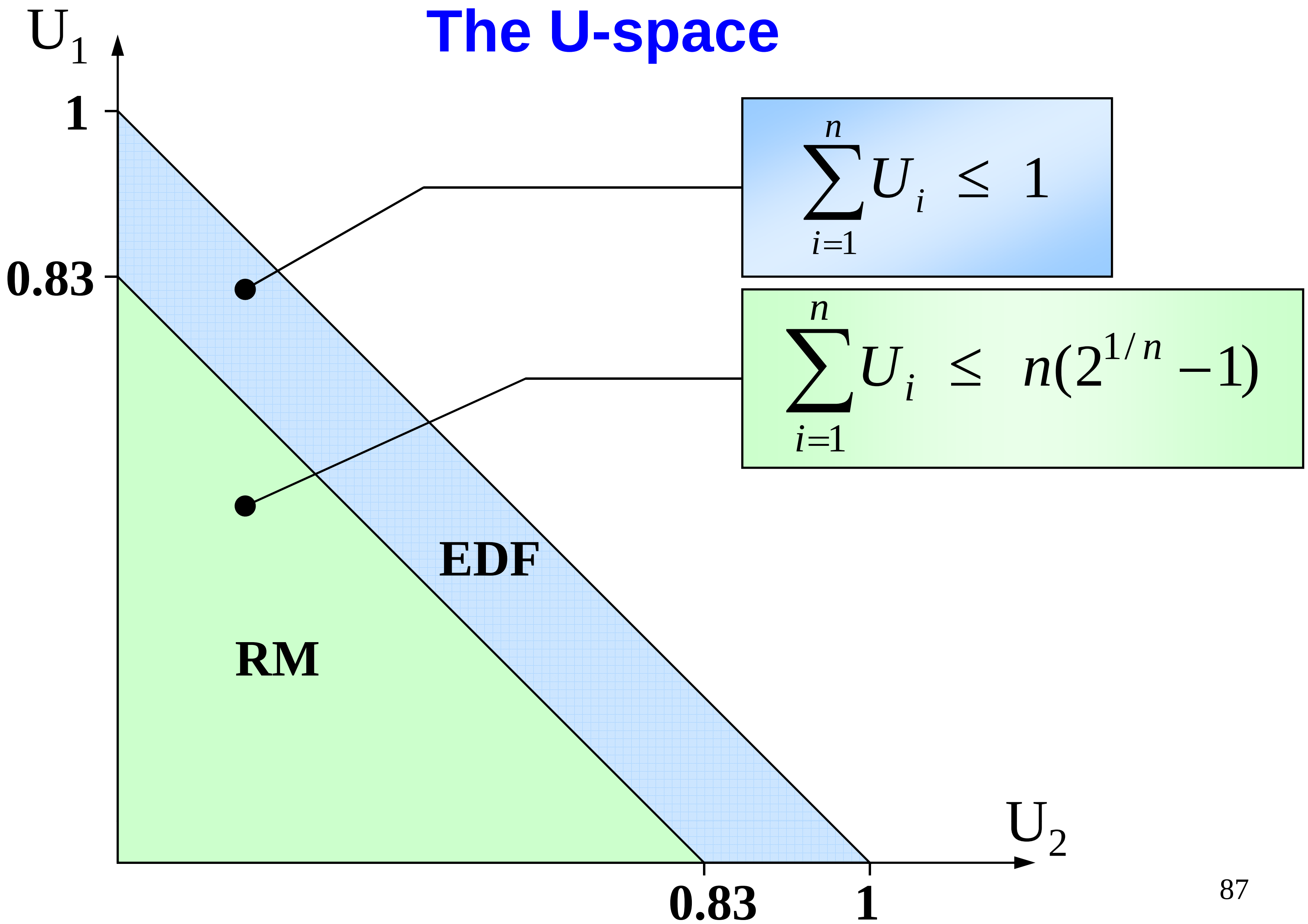
If tasks have harmonic periods $U_{lub} = 1$.

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



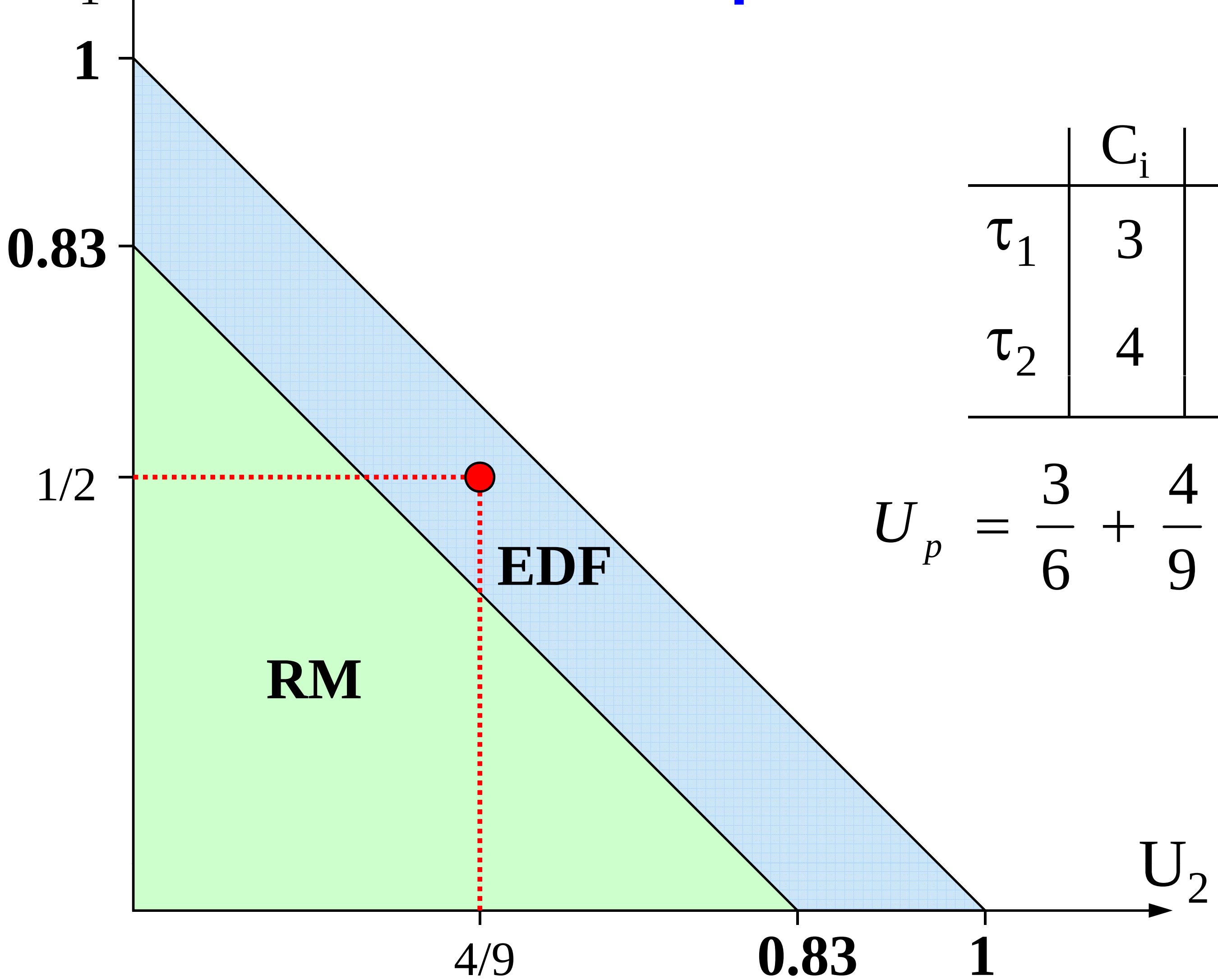
Schedulability region

The U-space



Schedulability region

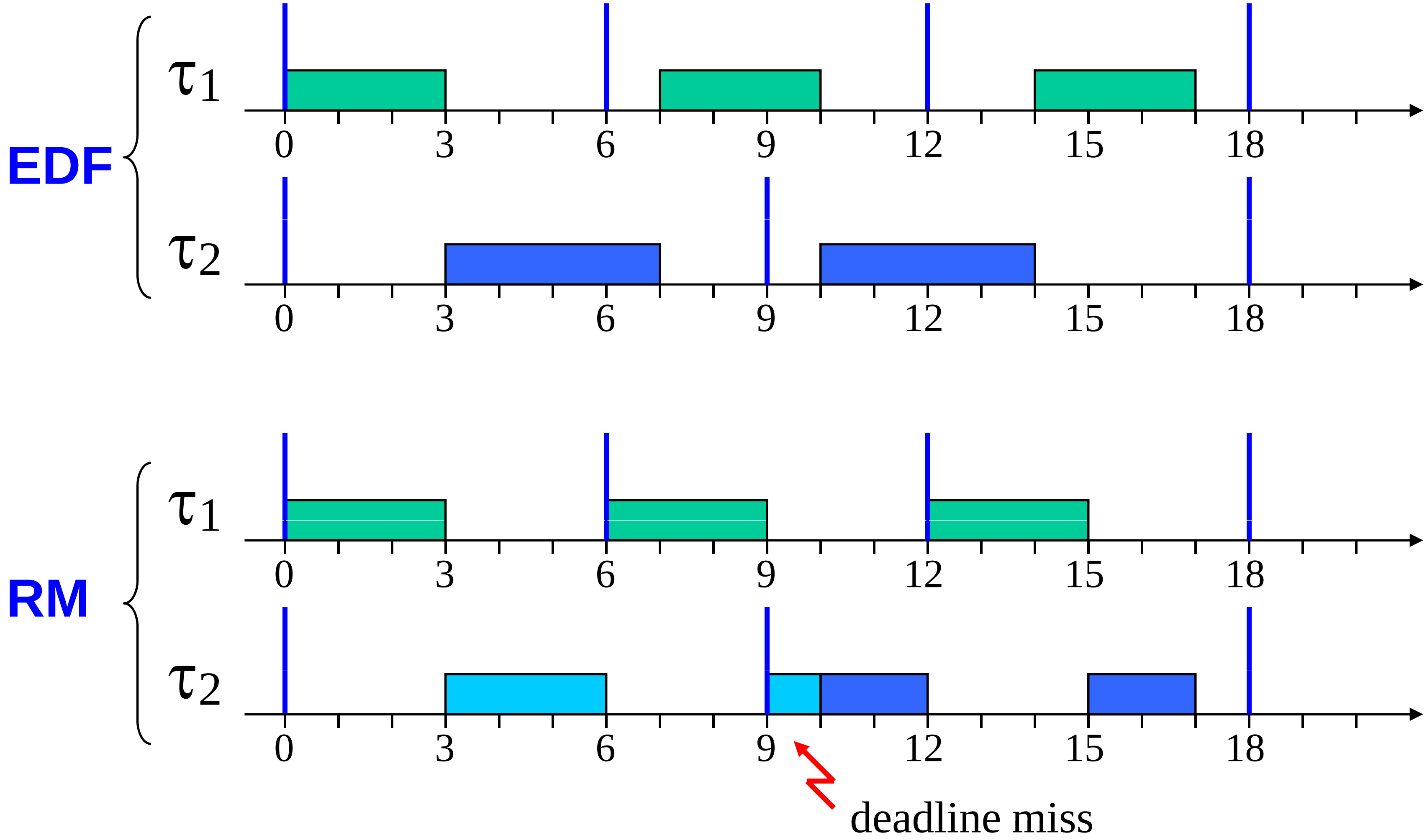
The U-space



	C_i	T_i
τ_1	3	6
τ_2	4	9

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.94$$

Schedule



RM Optimality

RM is **optimal** among all fixed priority algorithms:

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

$\Updownarrow$

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

EDF Optimality

EDF is **optimal** among all algorithms:

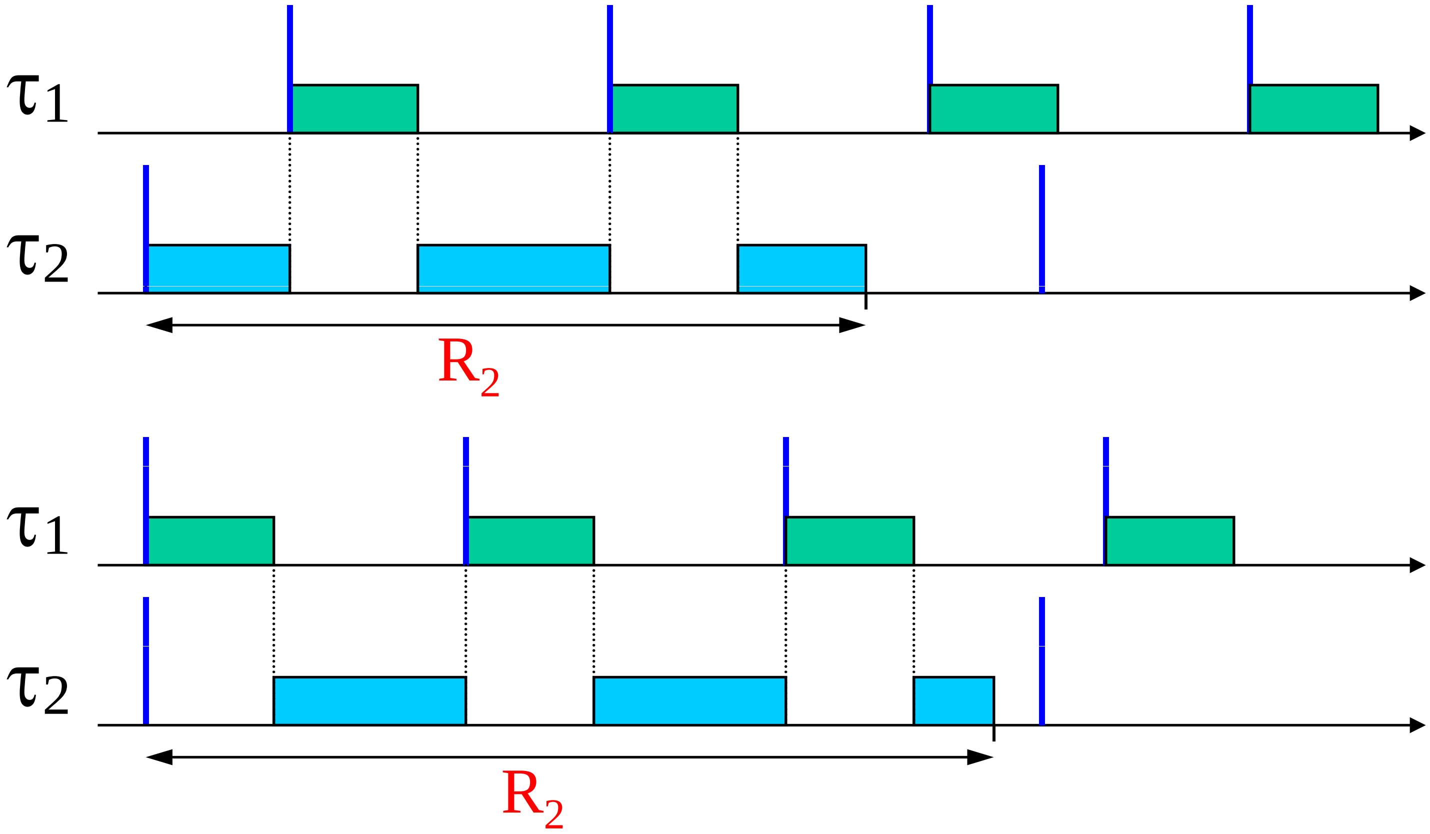
If there exists a feasible schedule for Γ, then EDF will generate a feasible schedule.

$\Updownarrow$

If Γ is not schedulable by EDF, then it cannot be scheduled by any algorithm.

Critical Instant

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



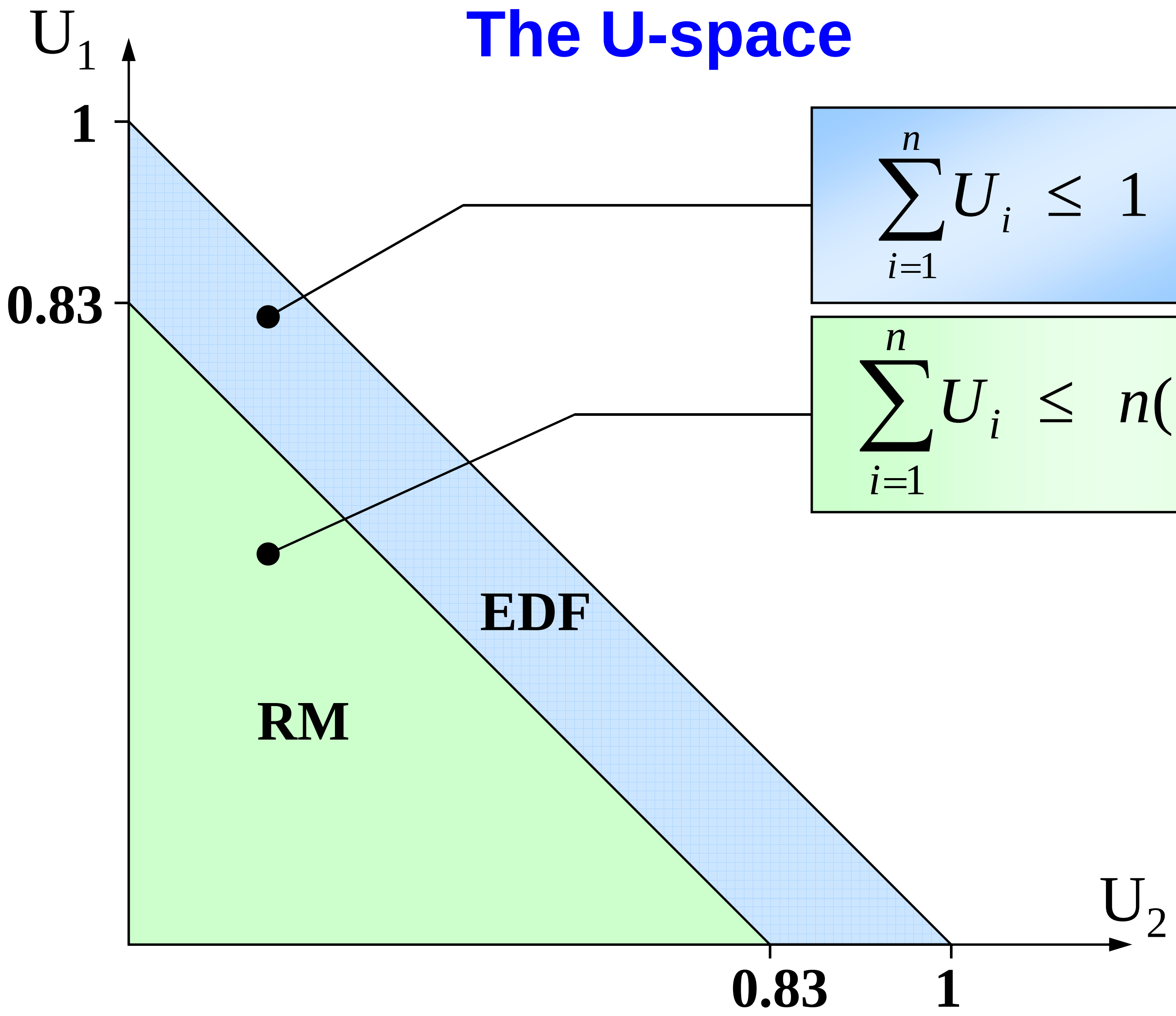
The Hyperbolic Bound

- In 2000, **Bini et al.** proved that a set of *n* periodic tasks is schedulable with RM if:

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

Schedulability region

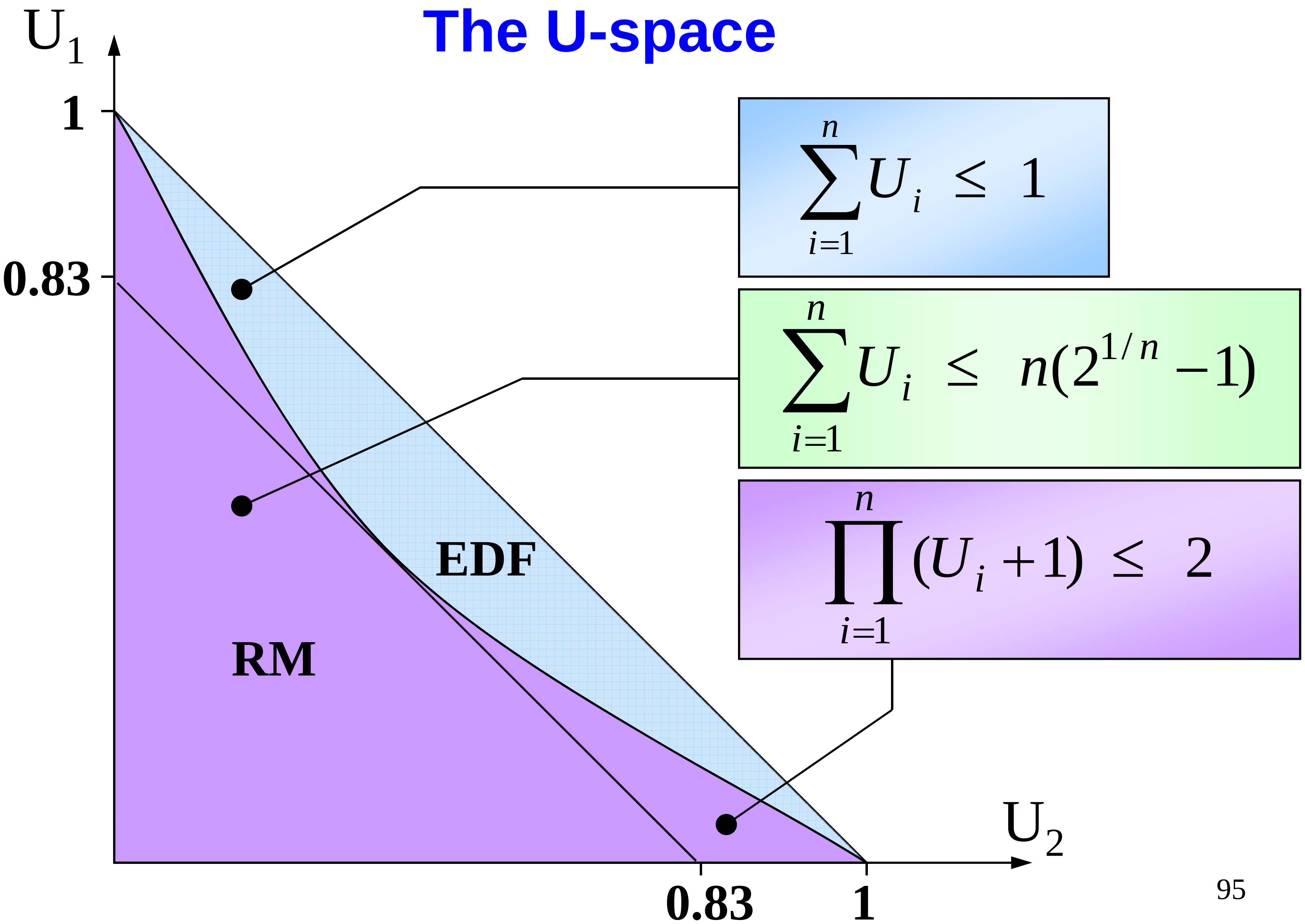
The U-space



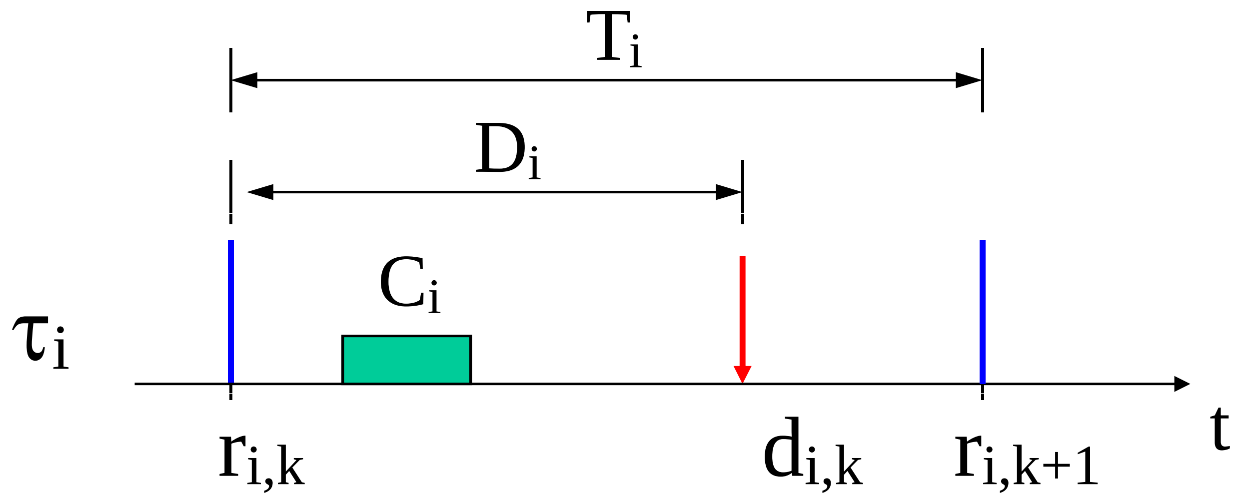
$$\sum_{i=1}^{n} U_i \leq 1$$

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

Schedulability region



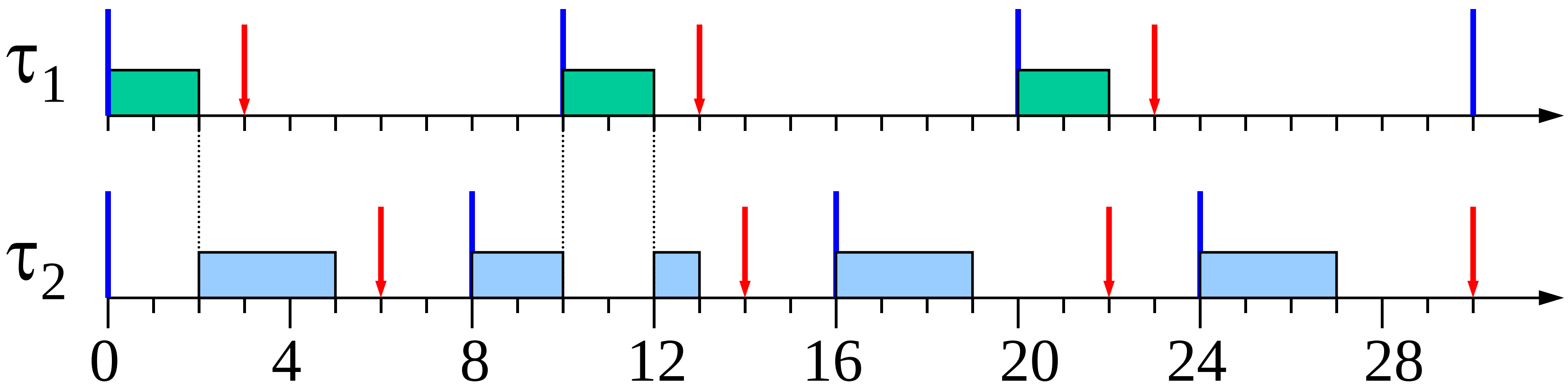
Extension to tasks with D < T



Scheduling algorithms

- Deadline Monotonic: $\mathbf{p_i \propto 1/D_i}$ (static)
- Earliest Deadline First: $\mathbf{p_i \propto 1/d_i}$ (dynamic)

Deadline Monotonic

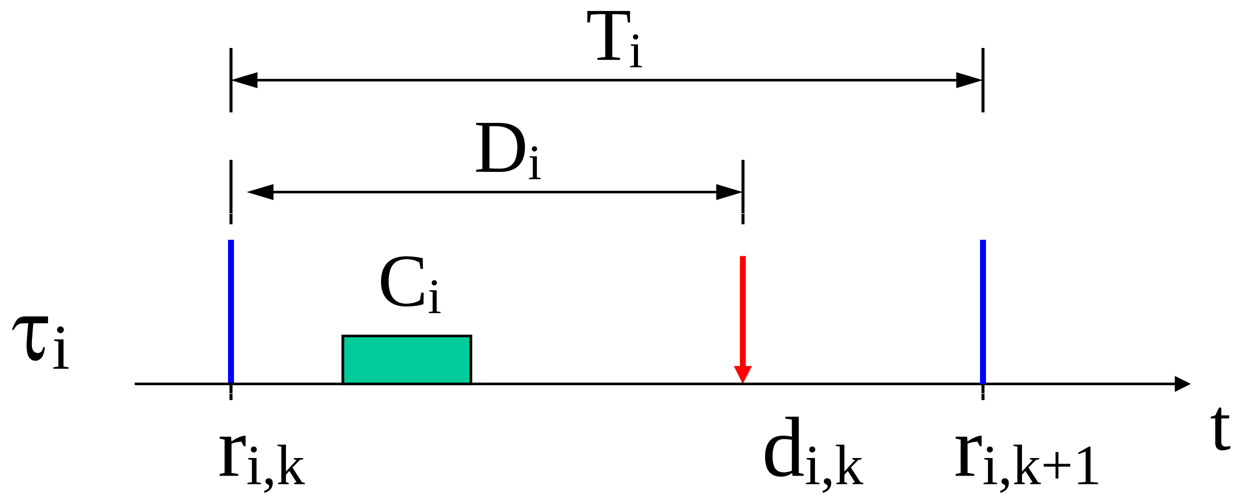


Problem with the Utilization Bound

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

but the task set is schedulable.

How to guarantee feasibility?



- Fixed priority: Response Time Analysis (RTA)
- EDF: Processor Demand Criterion (PDC)

Response Time Analysis
[Audsley '90]

- For each task τ_i compute the interference due to higher priority tasks:

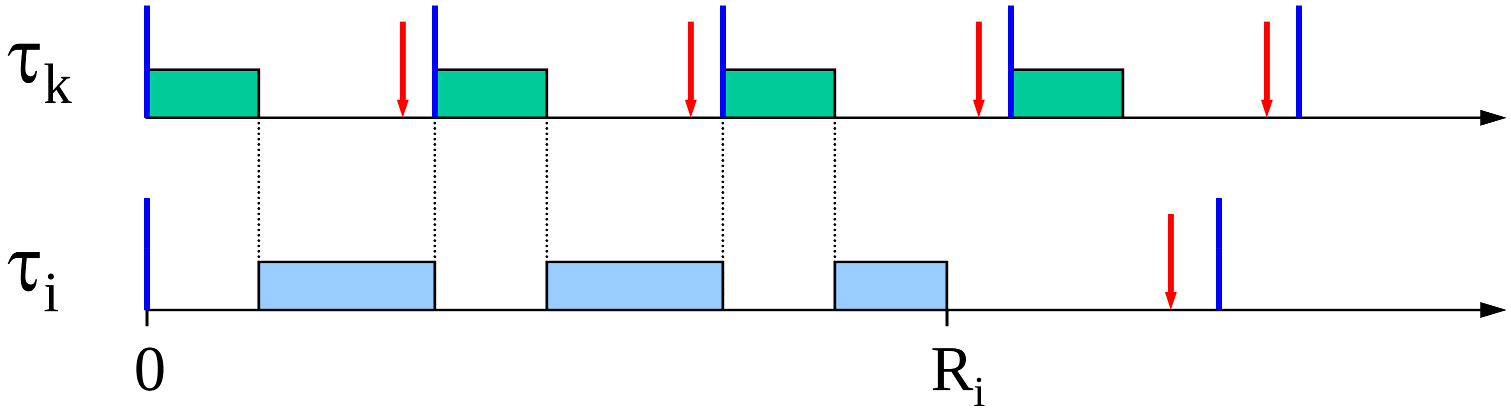
$$I_i = \sum_{D_k < D_i} C_k$$

- compute its response time as

$$R_i = C_i + I_i$$

- verify if $R_i \leq D_i$

Computing the interference



Interference of τ_k on τ_i
in the interval $[0, R_i]$:

$$I_{ik} = \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

Interference of high
priority tasks on τ_i:

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

Computing the response time

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

Iterative solution:

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

iterate until

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

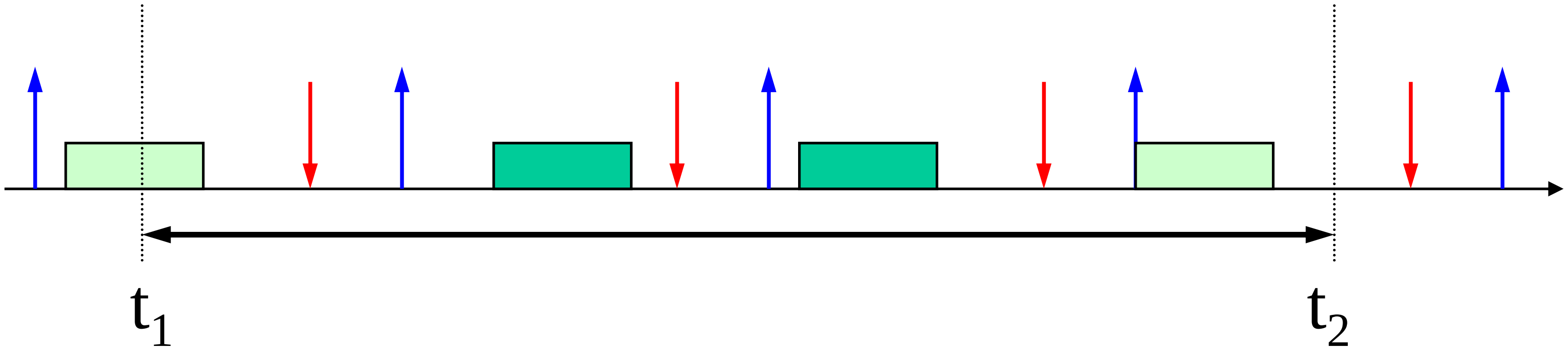
Processor Demand Criterion

[Baruah, Howell, Rosier 1990]

In any interval of time, the computation demanded by the task set must be no greater than the available time.

$$\forall t_1, t_2 > 0, \quad g(t_1, t_2) \leq (t_2 - t_1)$$

Processor Demand

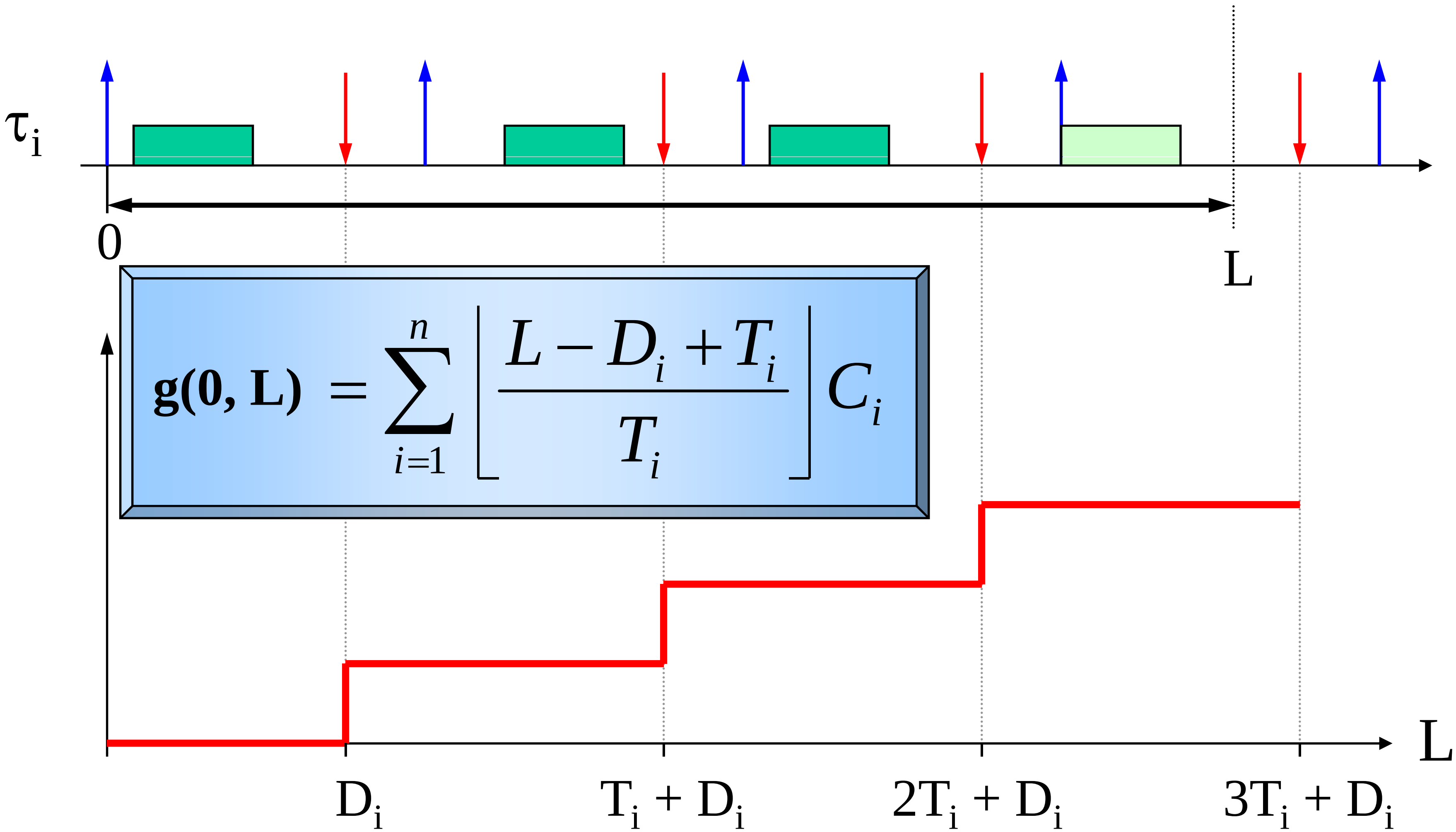


The demand in $[t_1, t_2]$ is the computation time of those jobs started at or after t_1 with deadline less than or equal to t_2:

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

Processor Demand

For synchronous task sets we can only analyze intervals [0,L]



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

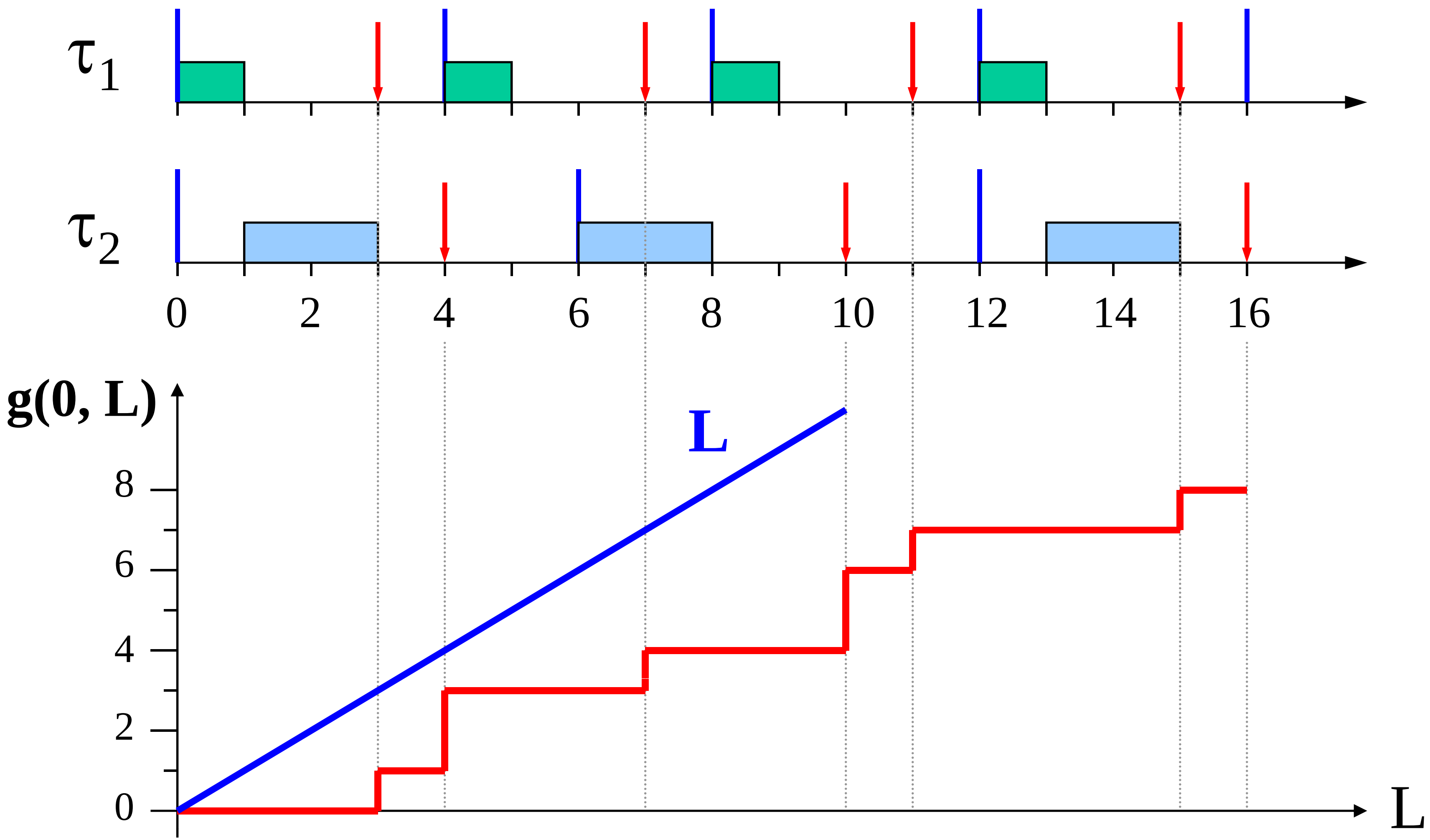
Processor Demand Test

$$\forall L > 0 \qquad \sum_{i=1}^{n} \left\lfloor \frac{L - D_i + T_i}{T_i} \right\rfloor C_i \;\leq\; L$$

Question

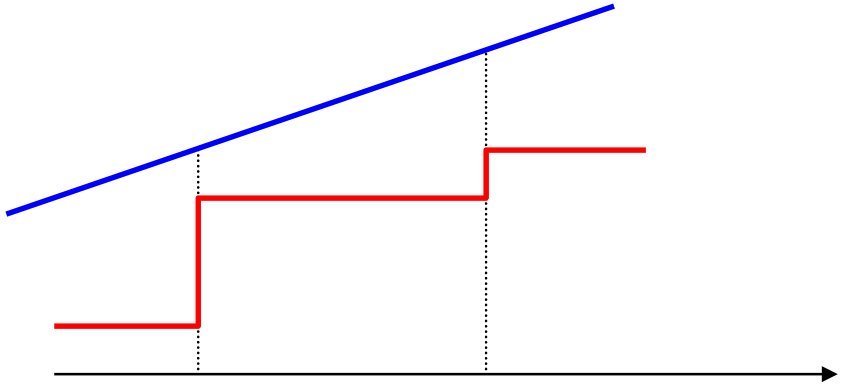
How can we bound the number of intervals in which the test has to be performed?

Example



Bounding complexity

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



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

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

Bounding complexity

- Moreover we note that: $g(0, L) \leq G(0, L)$

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

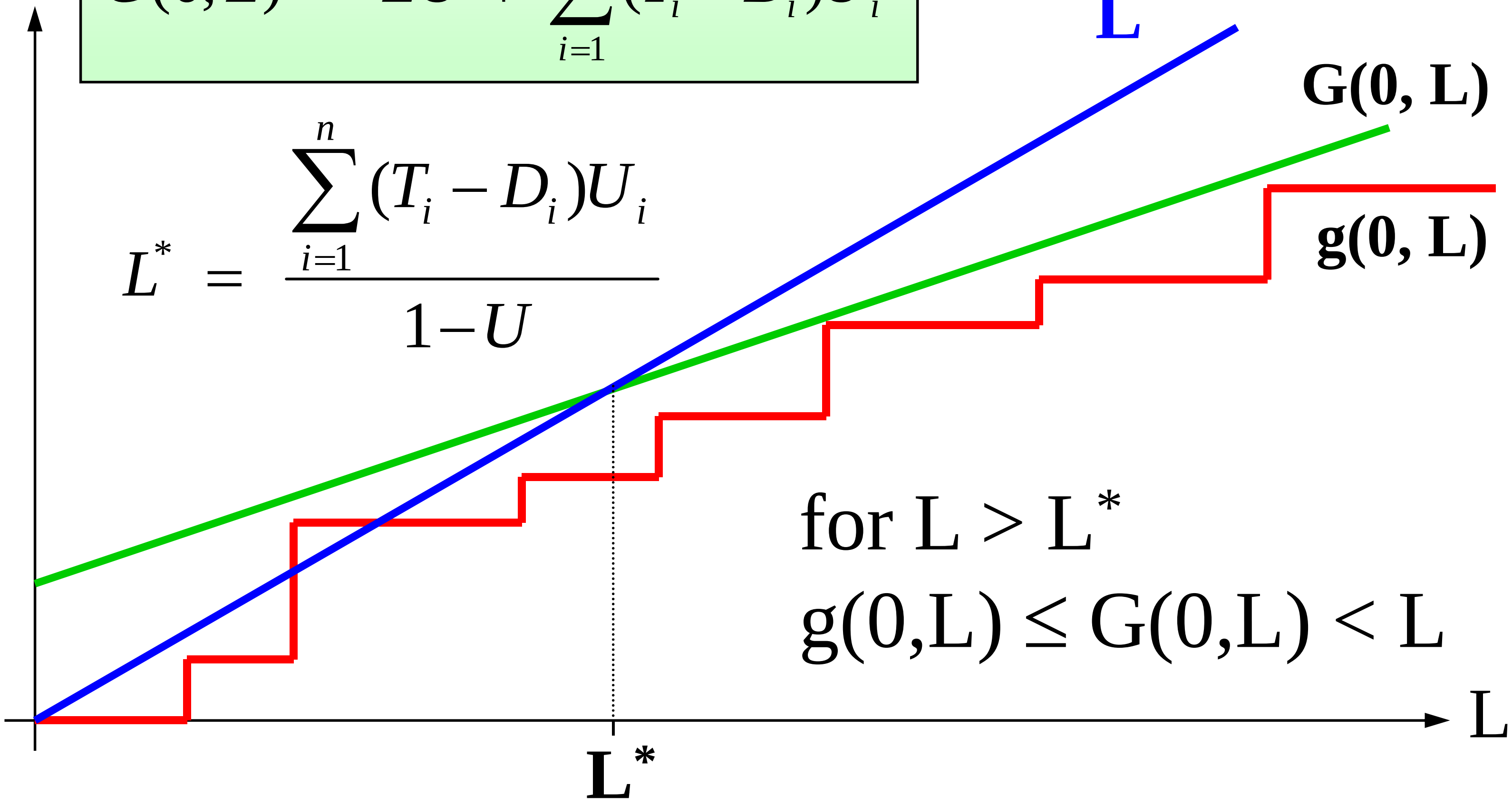
$$= \sum_{i=1}^{n} L \frac{C_i}{T_i} + \sum_{i=1}^{n} (T_i - D_i) \frac{C_i}{T_i}$$

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

Limiting L

$$G(0, L) = LU + \sum_{i=1}^{n} (T_i - D_i) U_i$$

$$L^* = \frac{\sum_{i=1}^{n} (T_i - D_i) U_i}{1 - U}$$



for $L > L^*$

$g(0,L) \le G(0,L) < L$

Processor Demand Test

A set of *n* periodic tasks with D ≤ T is schedulable by EDF if and only if

$$U < 1 \qquad \text{AND} \qquad \forall L > 0 \quad \sum_{i=1}^{n} \left\lfloor \frac{L - D_i + T_i}{T_i} \right\rfloor C_i \le L$$

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

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

Summarizing: RM vs. EDF

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

Questions

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

Main reason

- RM is simpler to implement on top of commercial (fixed priority) kernels.
- EDF requires explicit kernel support for deadline scheduling, but gives other advantages.

Advantages of EDF

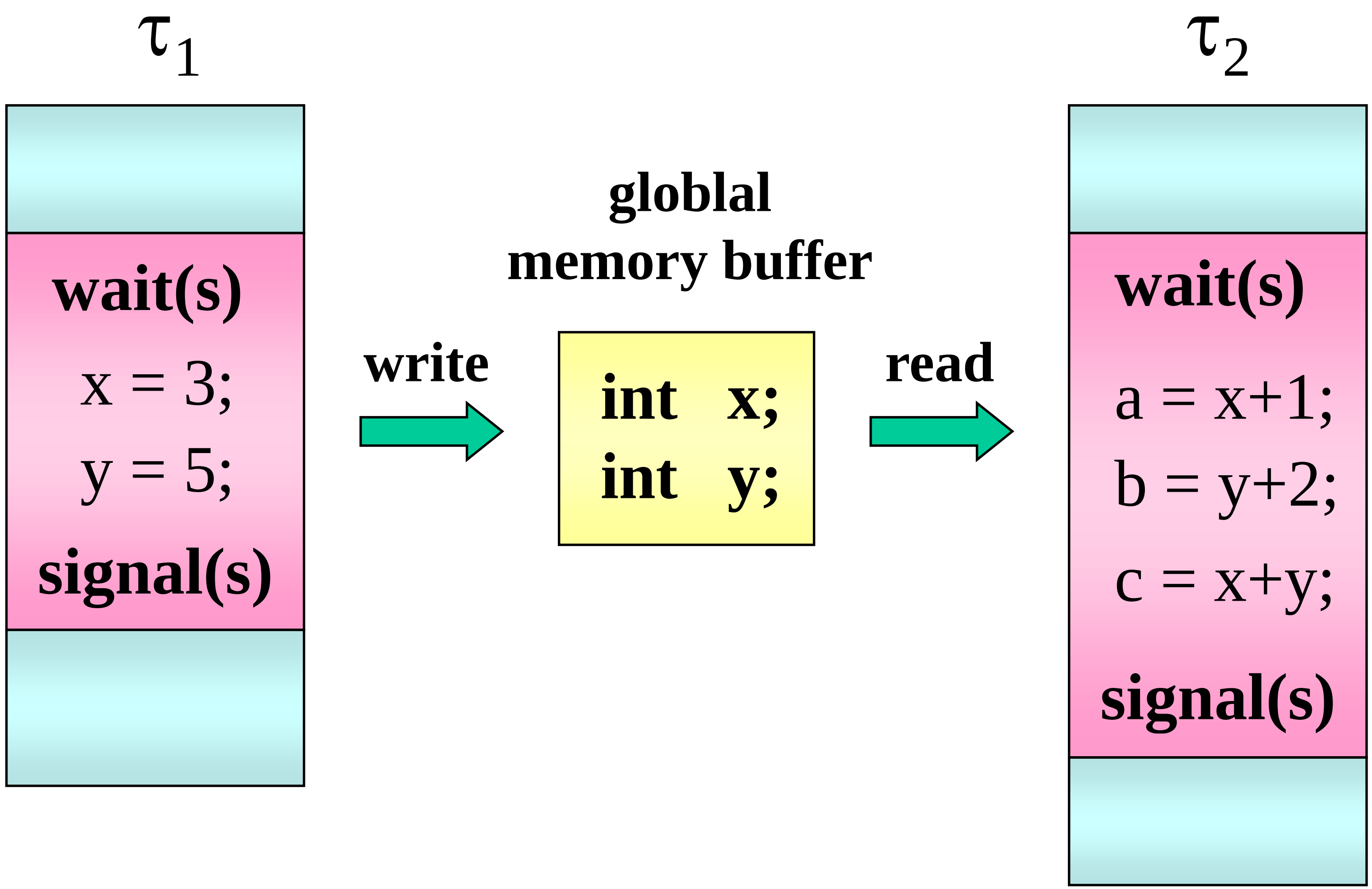
However, EDF offers the following advantages with respect to RM:

- Less overhaed due to preemptions;
- More flexible behavior in overload situations;
- More uniform jitter control;
- Better aperiodic responsiveness.

Handling shared resources

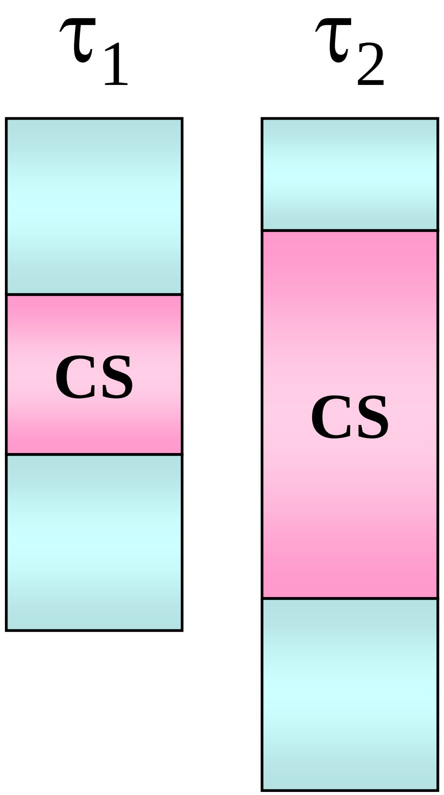
Problems caused by mutual exclusion

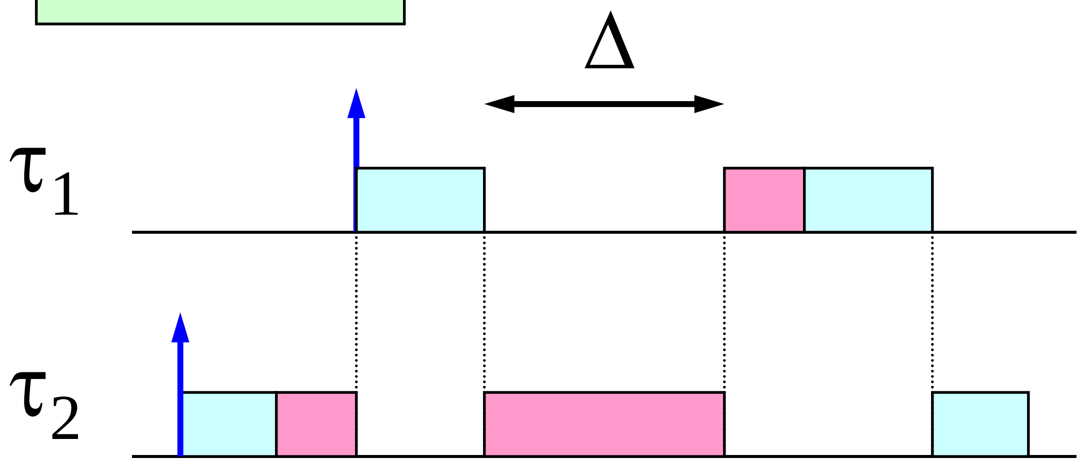
Critical sections



Blocking on a semaphore

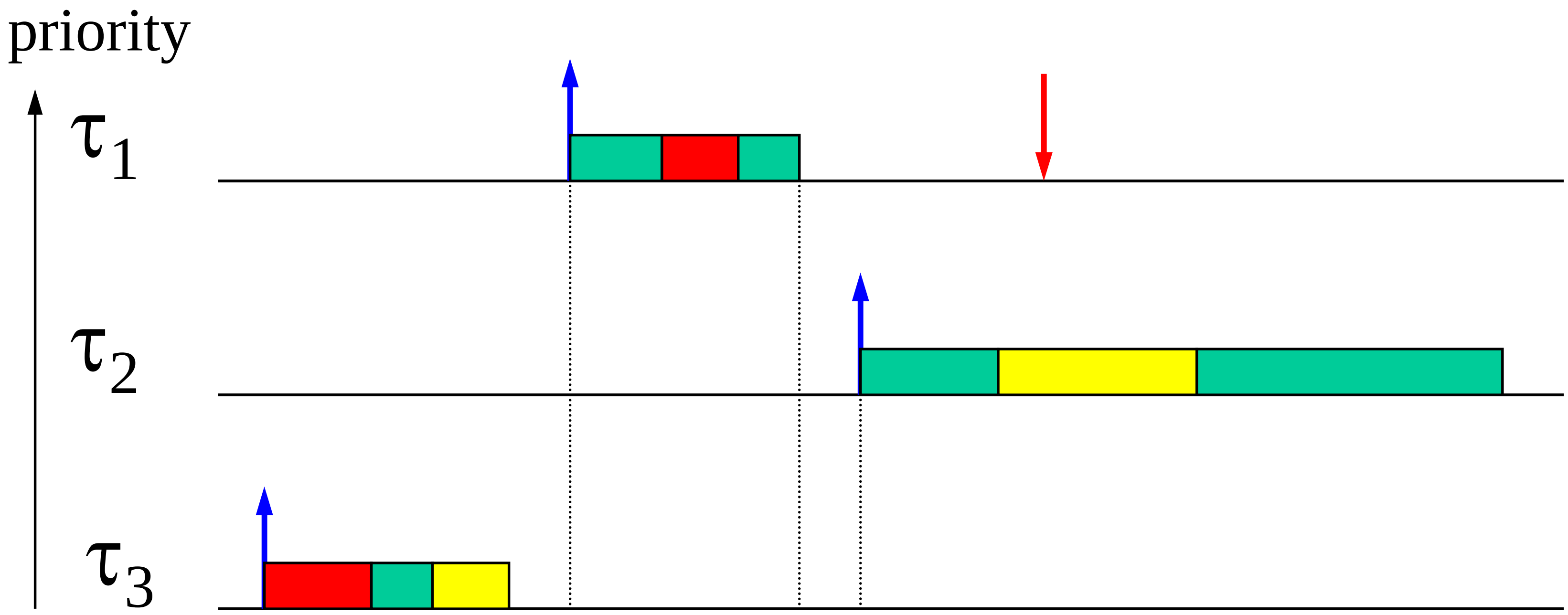
$p_1 > p_2$



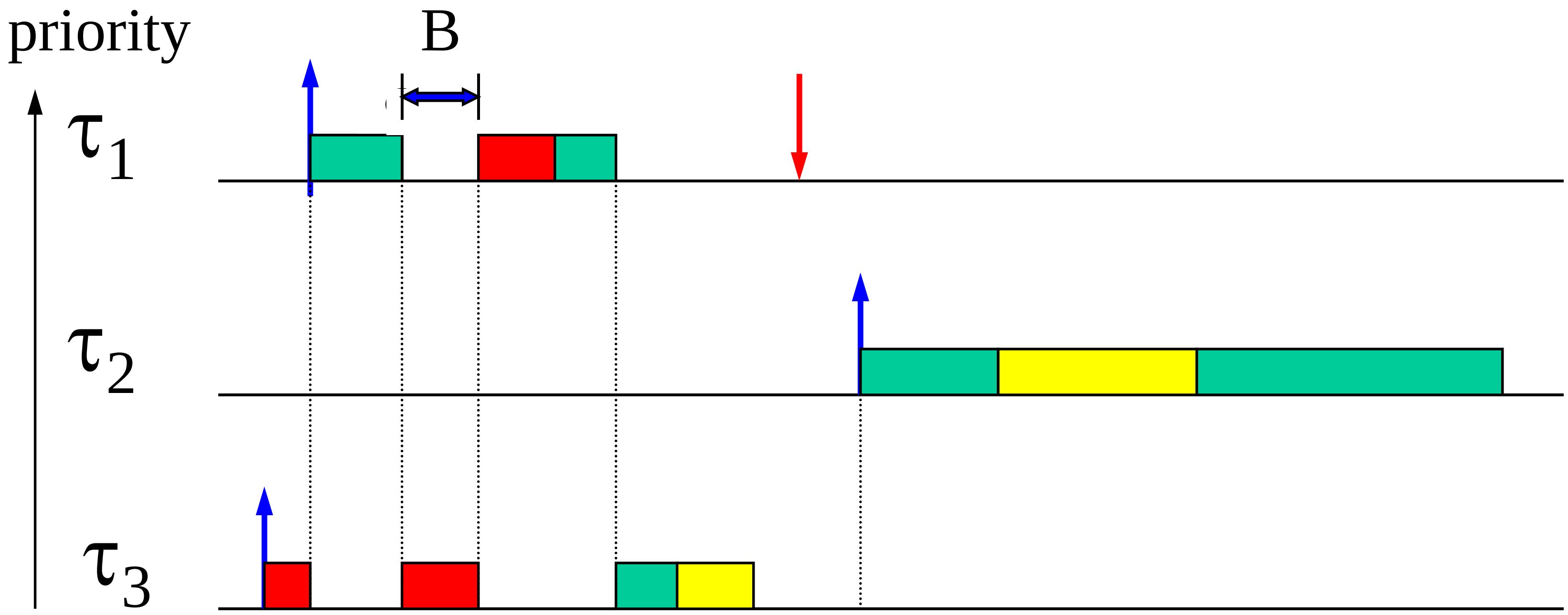


It seems that the maximum blocking time for τ_1 is equal to the length of the critical section of τ_2, but …

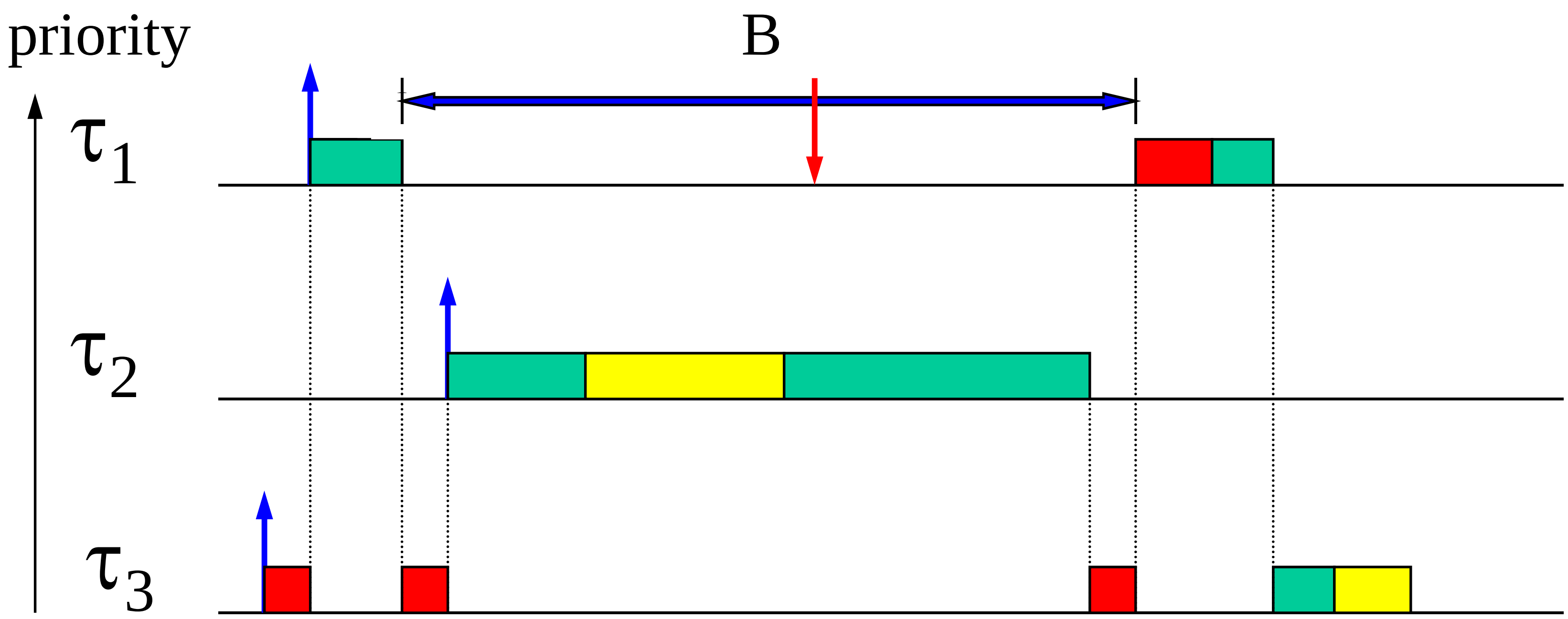
Schedule with no conflicts



Conflict on a critical section



Conflict on a critical section



Priority Inversion

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

Solution

Introduce a concurrency control protocol for accessing critical sections.

Resource Access Protocols

Under fixed priorities

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

Under EDF

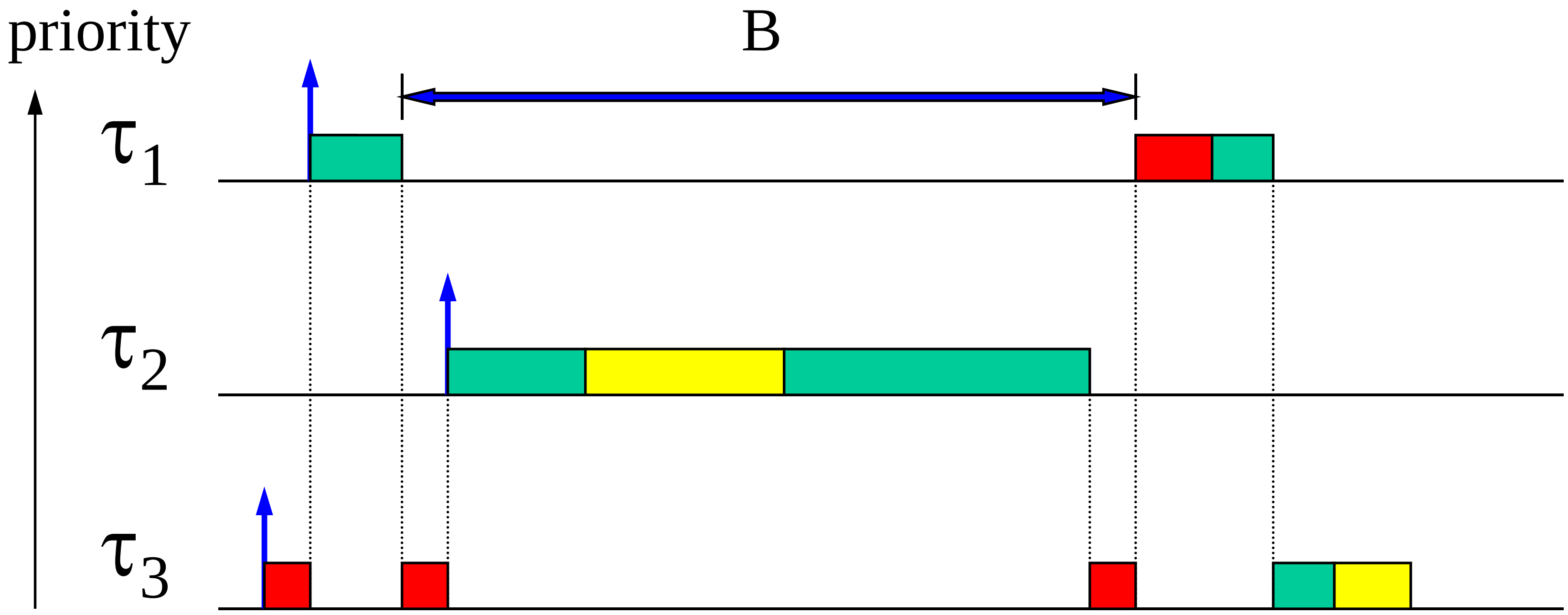
- Stack Resource Policy (SRP)

Non Preemptive Protocol

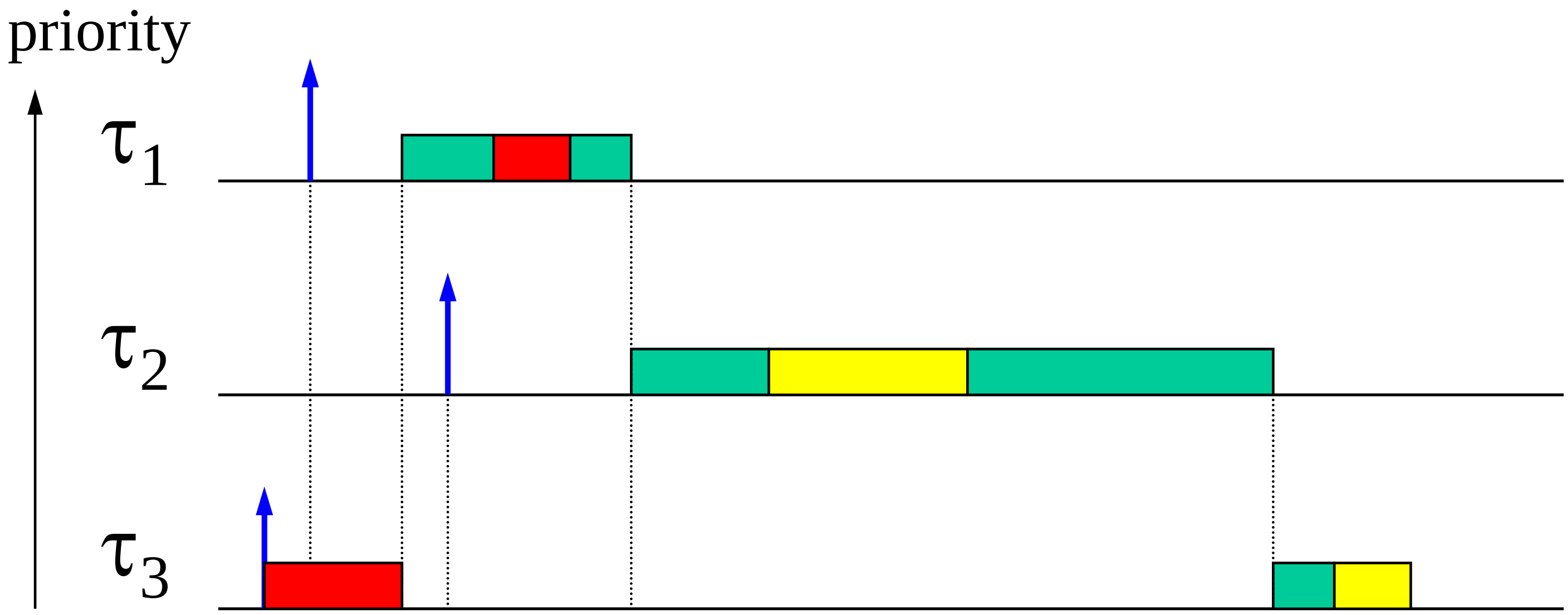
- Preemption is forbidden in critical sections.
- Implementation: when a task enters a CS, its priority is increased at the maximum value.

ADVANTAGES:	simplicity
PROBLEMS:	high priority tasks that do not use CS may also block

Conflict on critical section

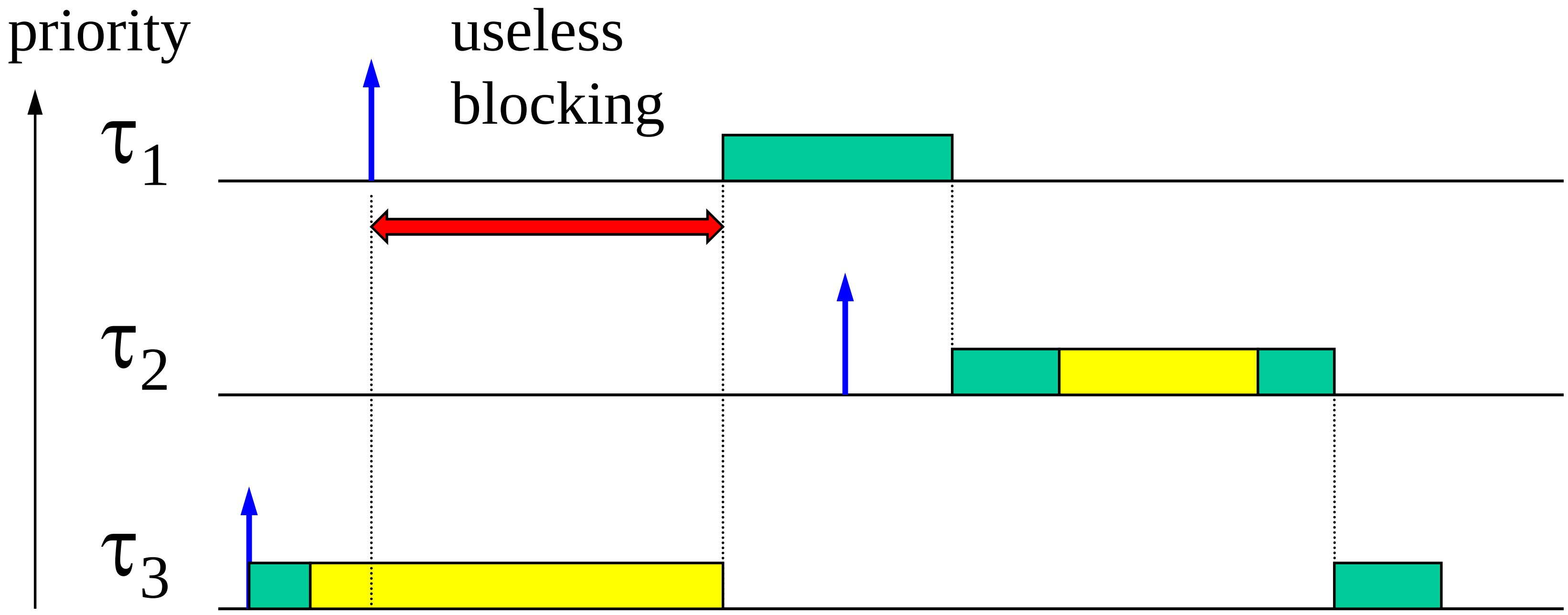


Schedule with NPP



$$P_{CS} = \max\{P_1, \ldots P_n\}$$

Problem with NPP



τ_1 cannot preemt, although it could

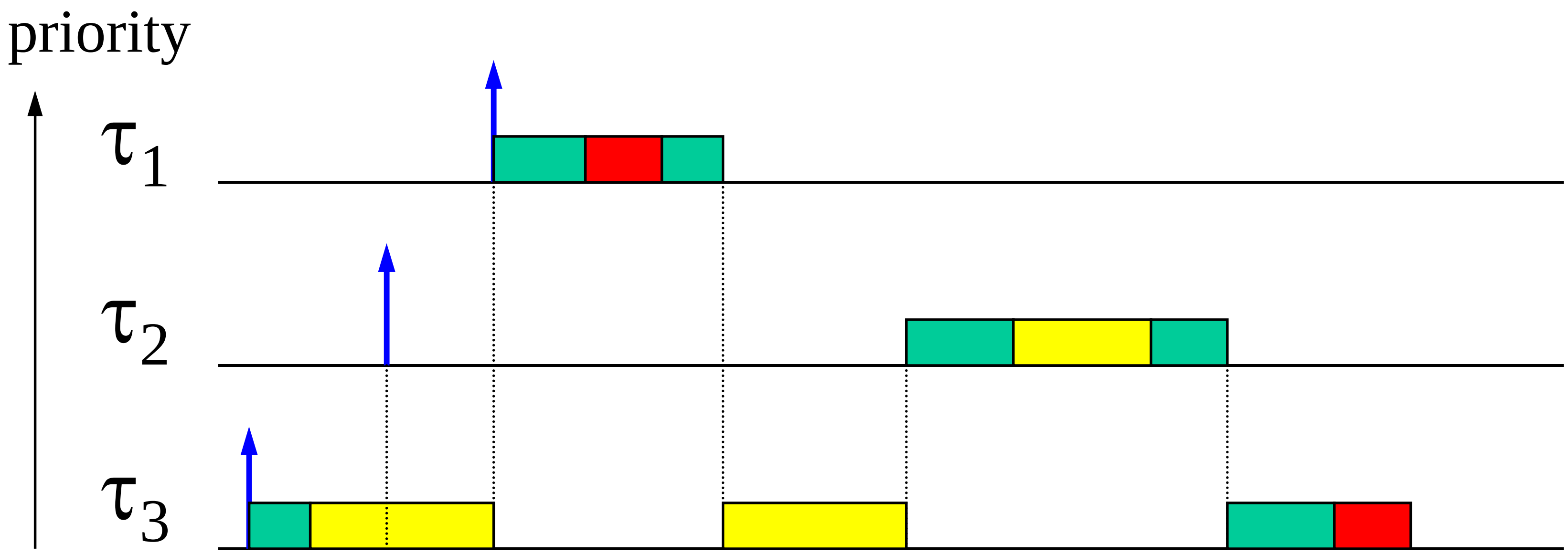
Highest Locker Priority

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.

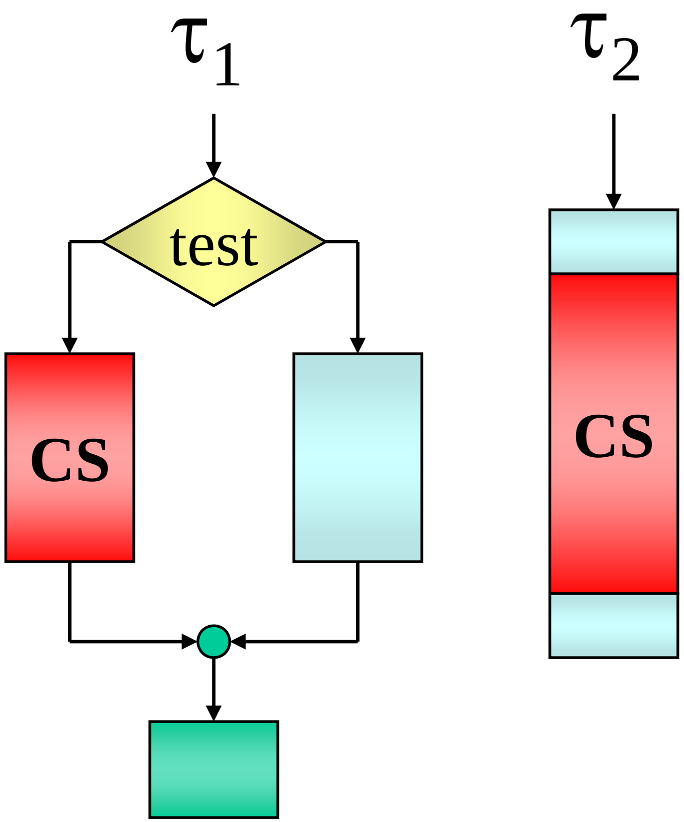
Schedule with HLP



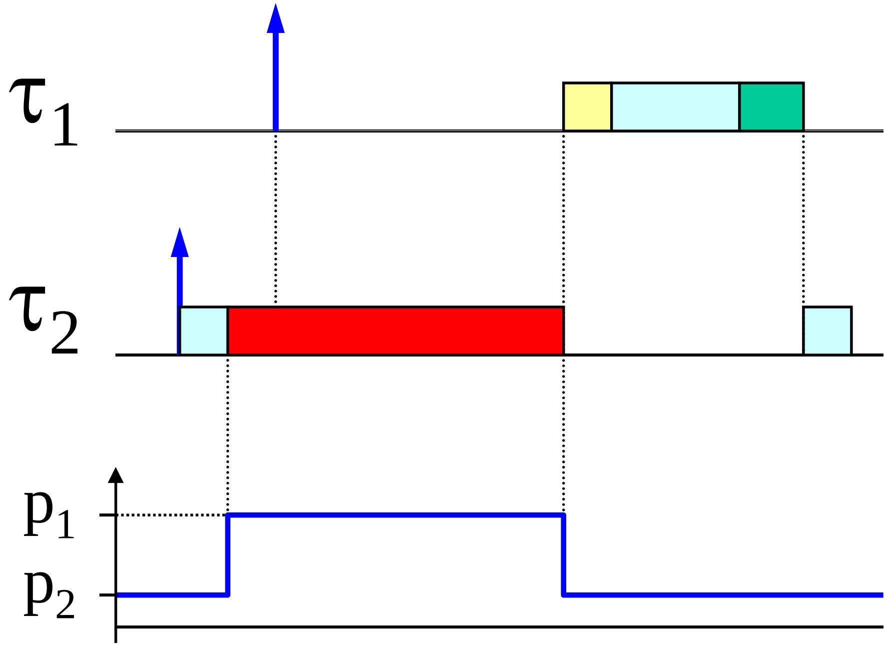
$$P_{CS} = \max \{P_k \mid \tau_k \text{ uses CS}\}$$

τ_2 is blocked, but τ_1 can preempt within a CS

Problem with HLP



τ_1 blocks just in case ...



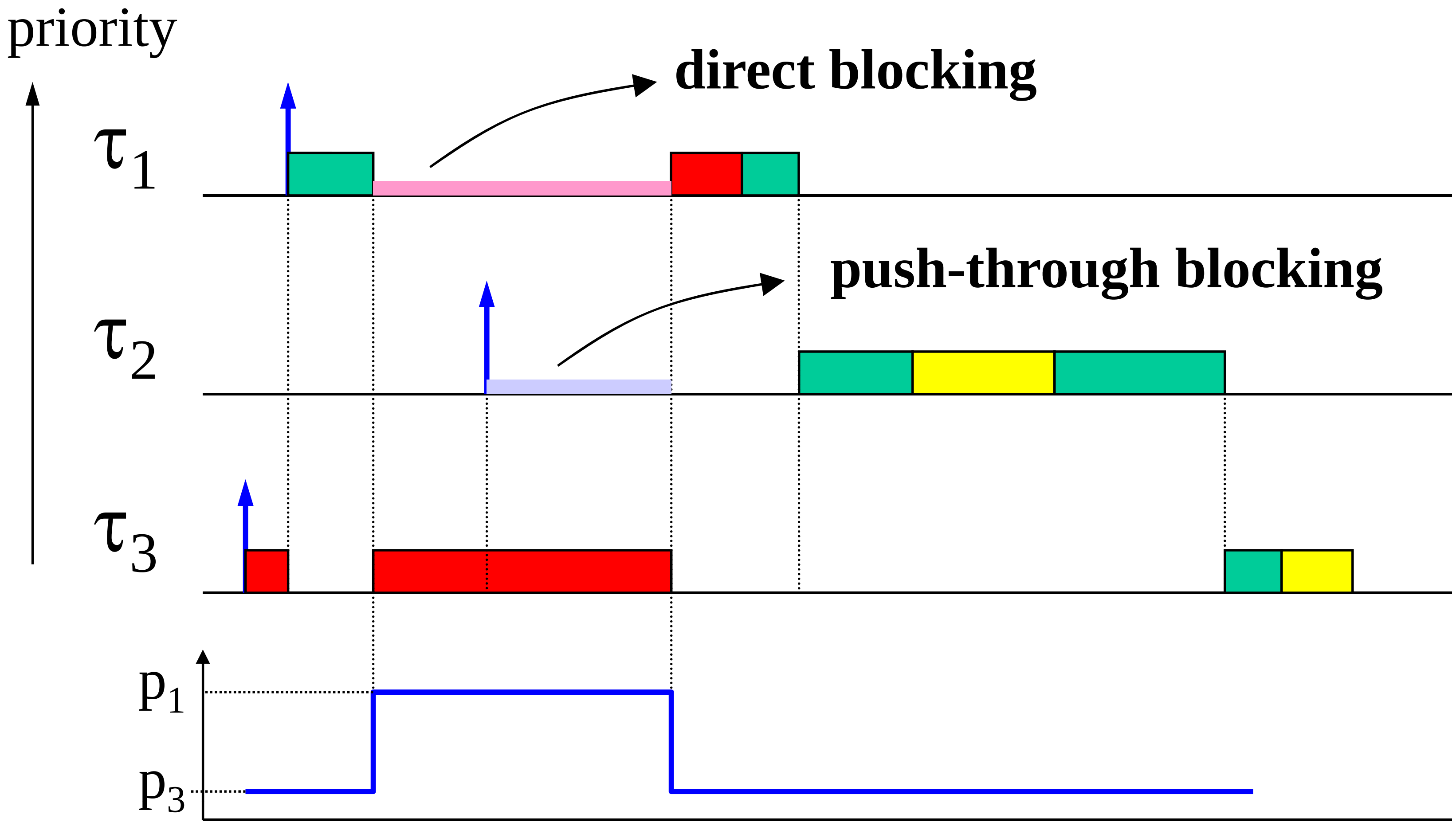
Priority Inheritance Protocol

[Sha, Rajkumar, Lehoczky, 90]

- 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.

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

Schedule with PIP



Types of blocking

- **Direct blocking**

  A task blocks on a locked semaphore

- **Push-through blocking**

  A task blocks because a lower priority task inherited a higher priority.

BLOCKING:

a delay caused by a lower priority task

Identifying blocking resources

- A task τ_i can be blocked by those semaphores used by lower priority tasks and
  - directly shared with τ_i (direct blocking) or
  - shared with tasks having priority higher than τ_i (push-through blocking).

Theorem: τ_i can be blocked at most once by each of such semaphores

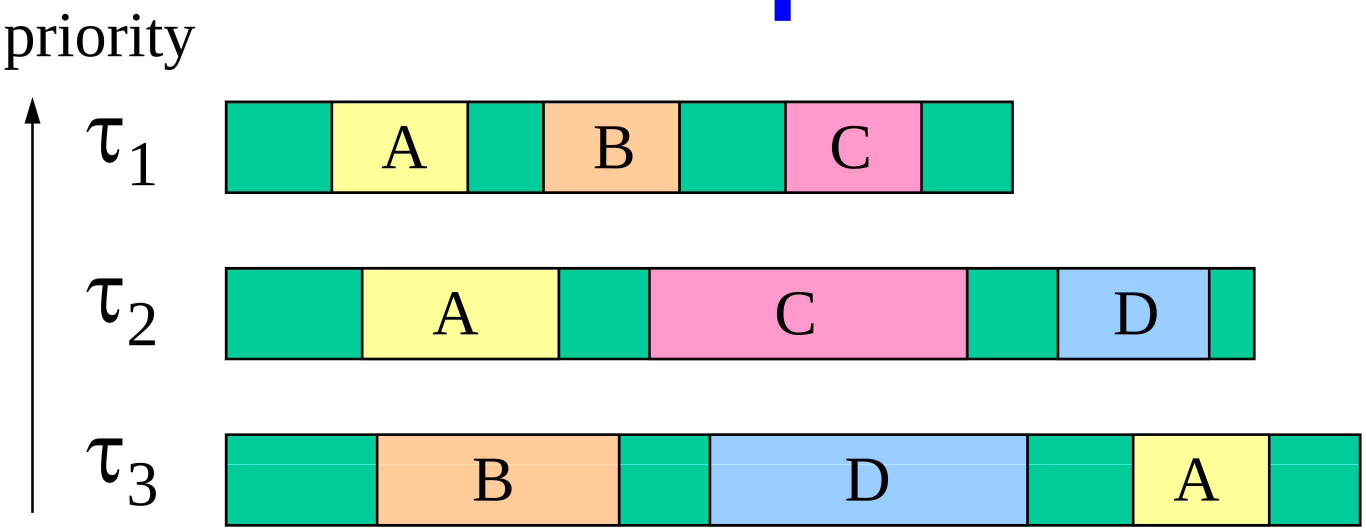
Theorem: τ_i can be blocked at most once by each lower priority task

Bounding blocking times

- If **n** is the number of tasks with priority less than τ_i
- and **m** is the number of semaphores on which τ_i can be blocked, **then**

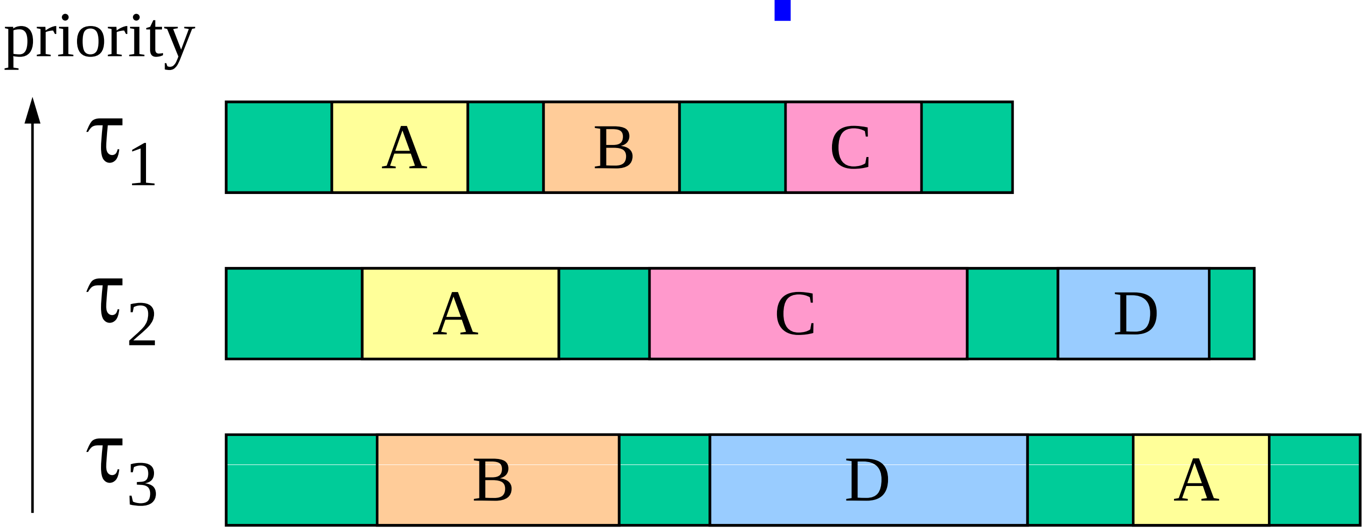
Theorem:	τ_i can be blocked at most for the duration of **min(n,m)** critical sections

Example



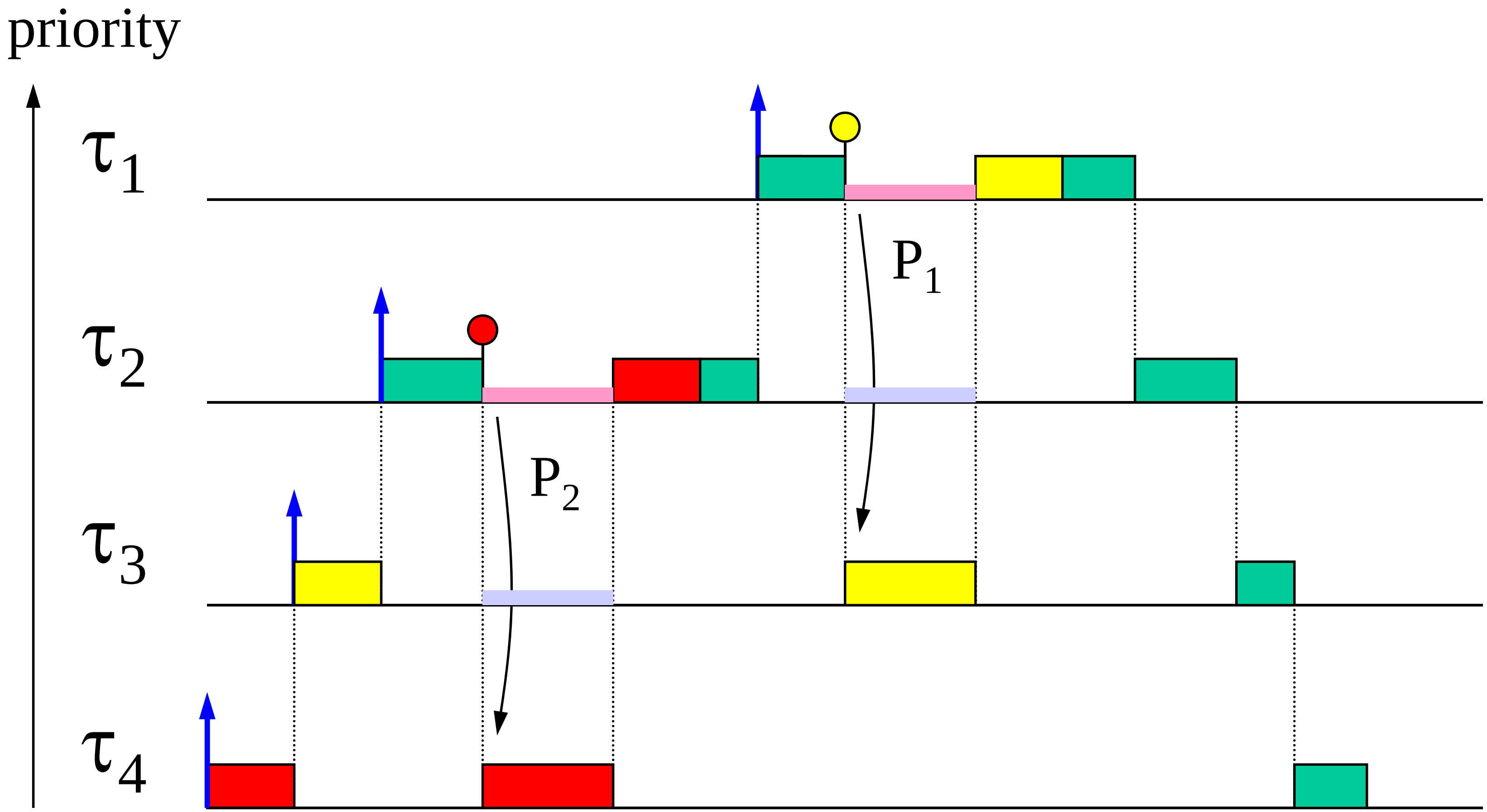
- τ_1 can be blocked once by τ_2 (on A_2 or C_2) and once by τ_3 (on A_3 or B_3)
- τ_2 can be blocked once by τ_3 (on A_3, B_3 or D_3)
- τ_3 cannot be blocked

Example



- $B_1 = \delta(C_2) + \delta(B_3)$
- $B_2 = \delta(D_3)$
- $B_3 = 0$

Schedule with PIP



Remarks on PIP

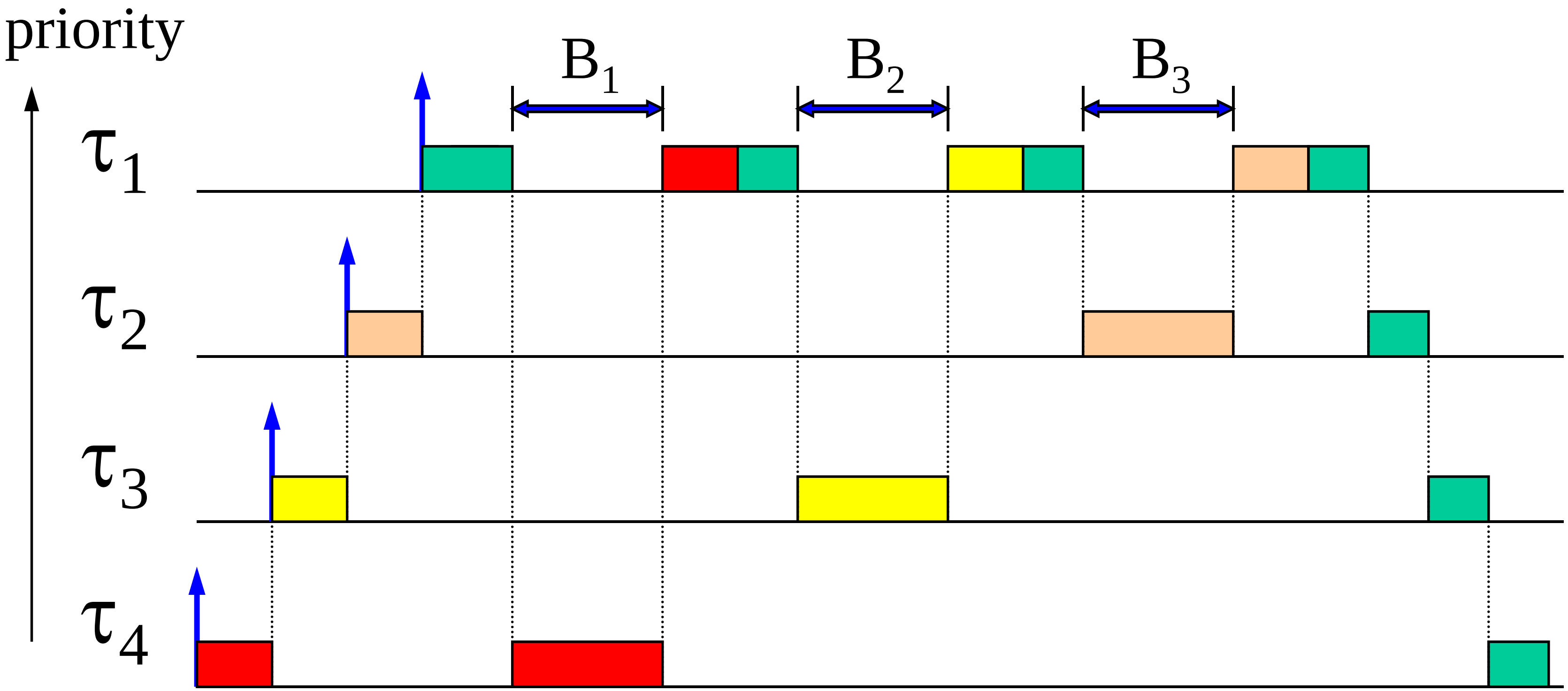
ADVANTAGES

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

PROBLEMS

- It does not avoid deadlocks and chained blocking.

Chained blocking with PIP



Theorem: τ_i can be blocked at most once by each lower priority task

Priority Ceiling Protocol

- 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.

To prevent chained blocking, a task may stop at the entrance of a free CS (***ceiling blocking***).

Resource Ceilings

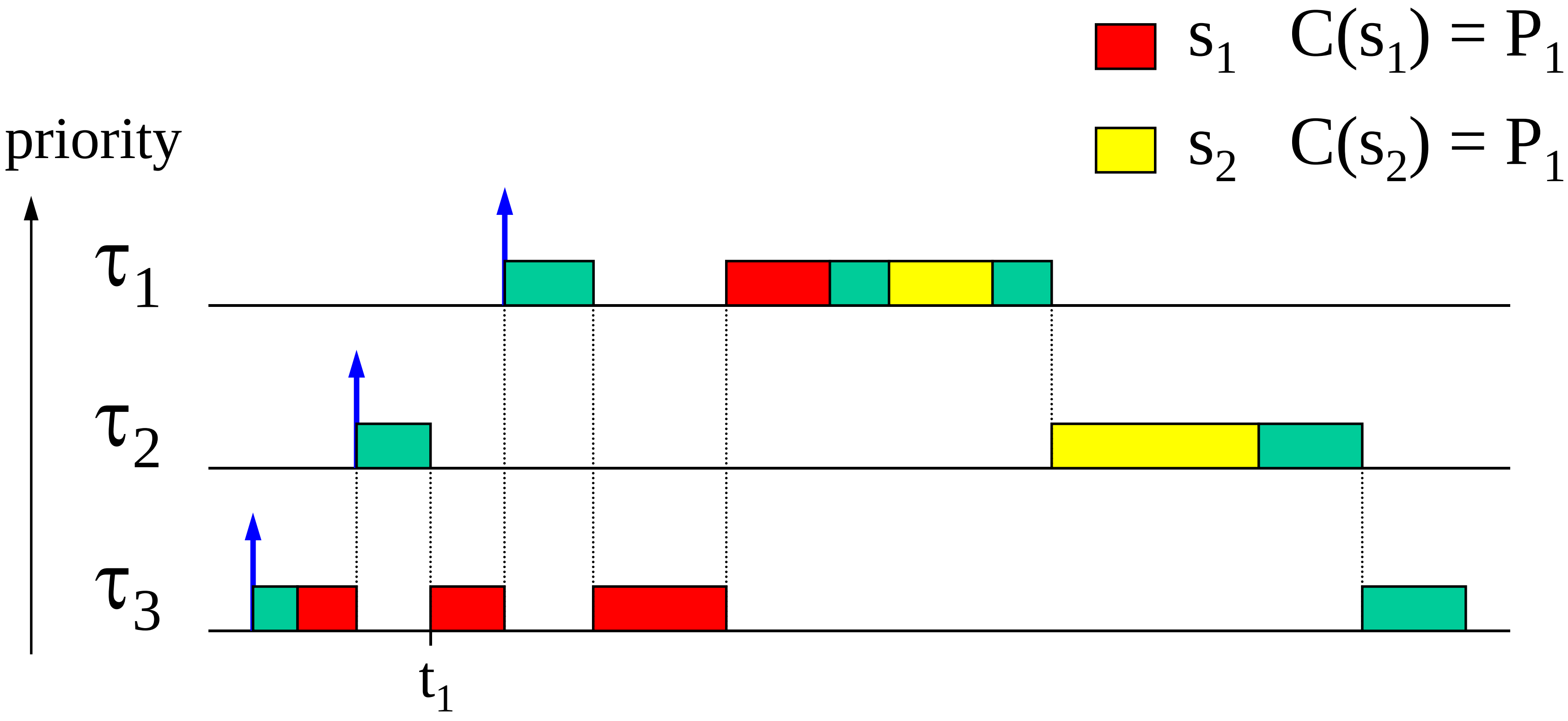
- Each semaphore s_k is assigned a ceiling:

$$C(s_k) = \max \{P_j : \tau_j \text{ uses } s_k\}$$

- A task τ_i can enter a CS only if

$$P_i > \max \{C(s_k) : s_k \text{ locked by tasks} \neq \tau_i\}$$

Schedule with PCP



t_1: τ_2 is blocked by the PCP, since $P_2 < C(s_1)$

Remarks on PCP

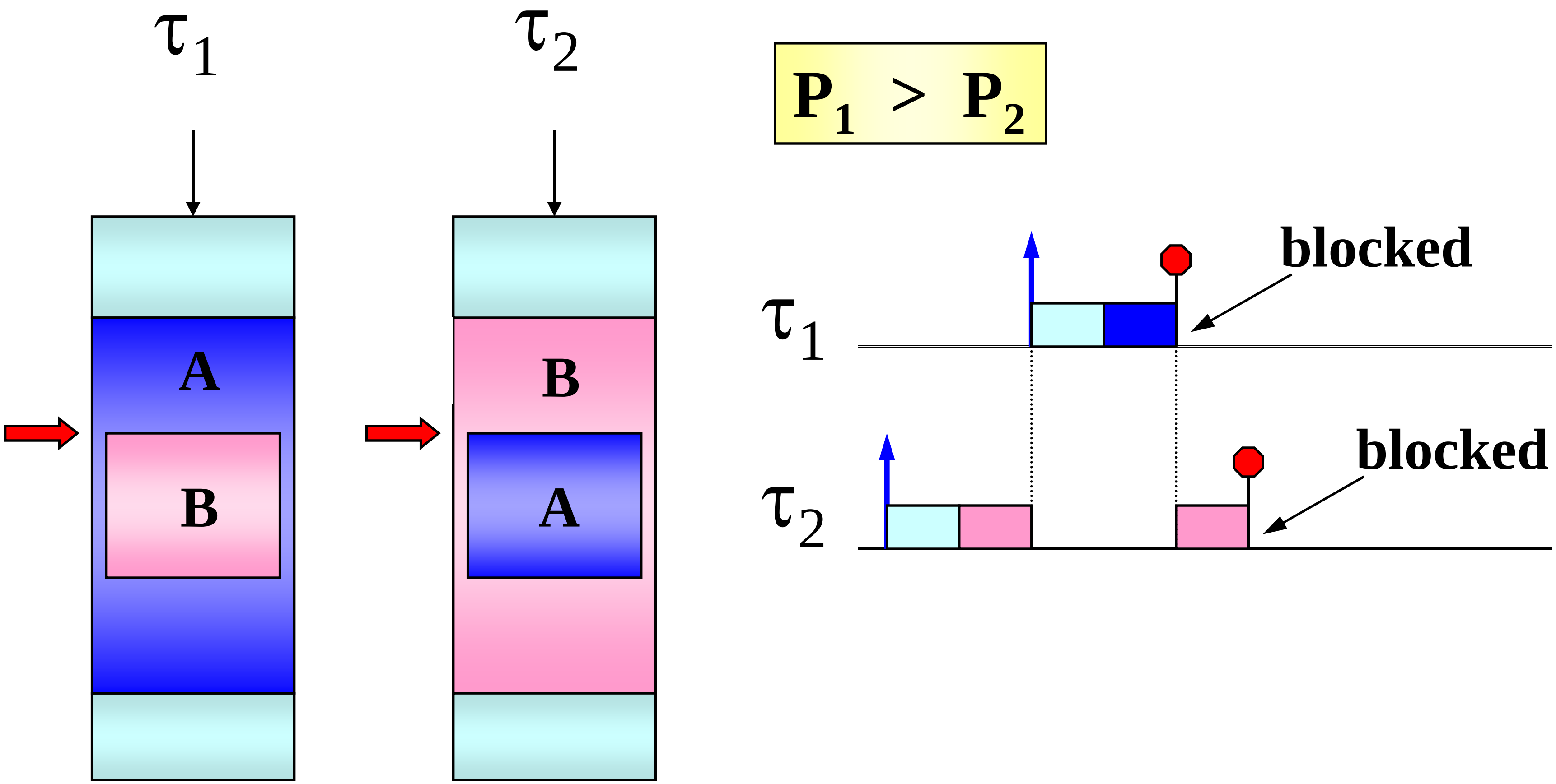
ADVANTAGES

- Blocking is reduced to only one CS
- It prevents deadlocks

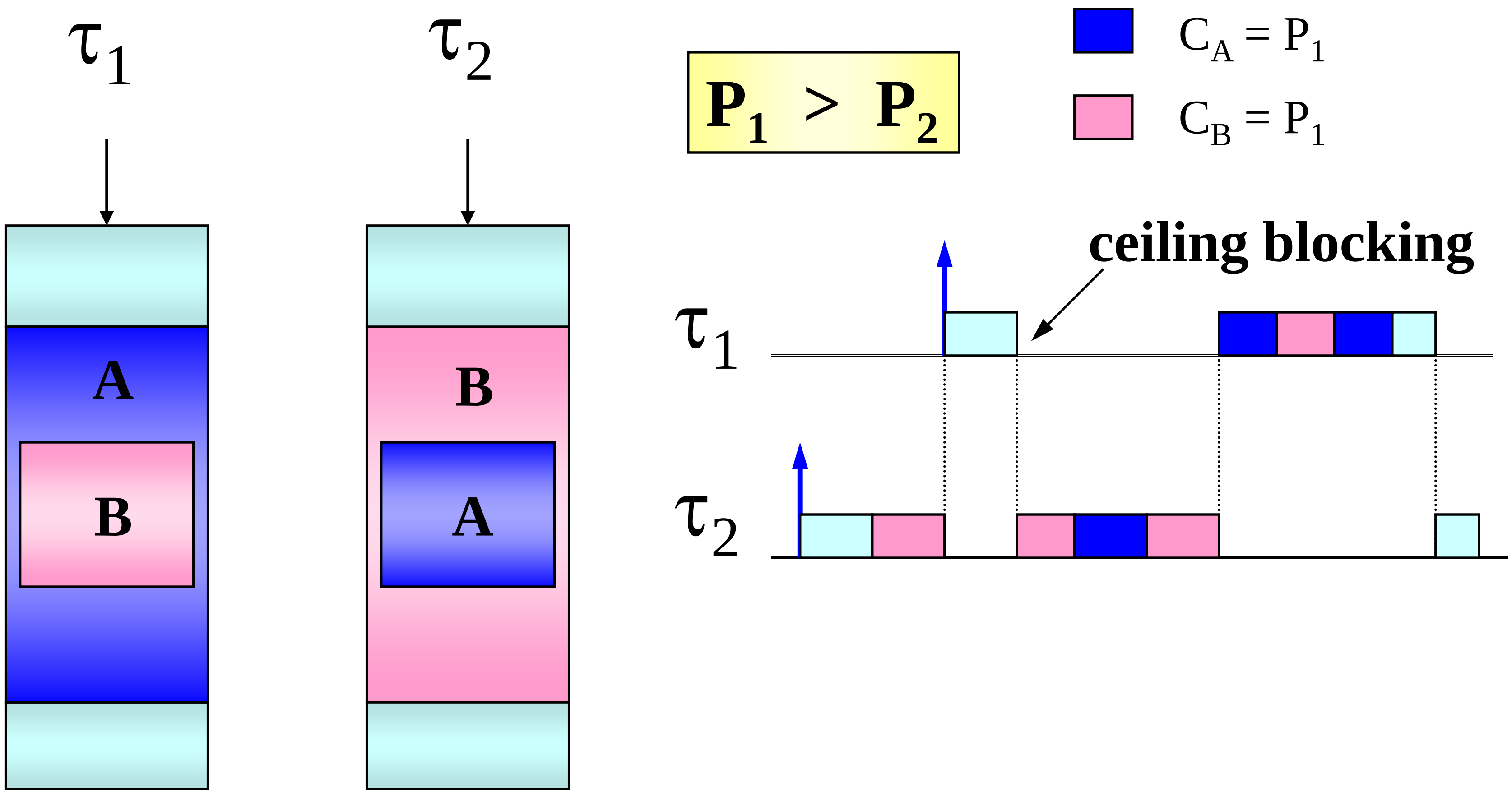
PROBLEMS

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

Typical Deadlock



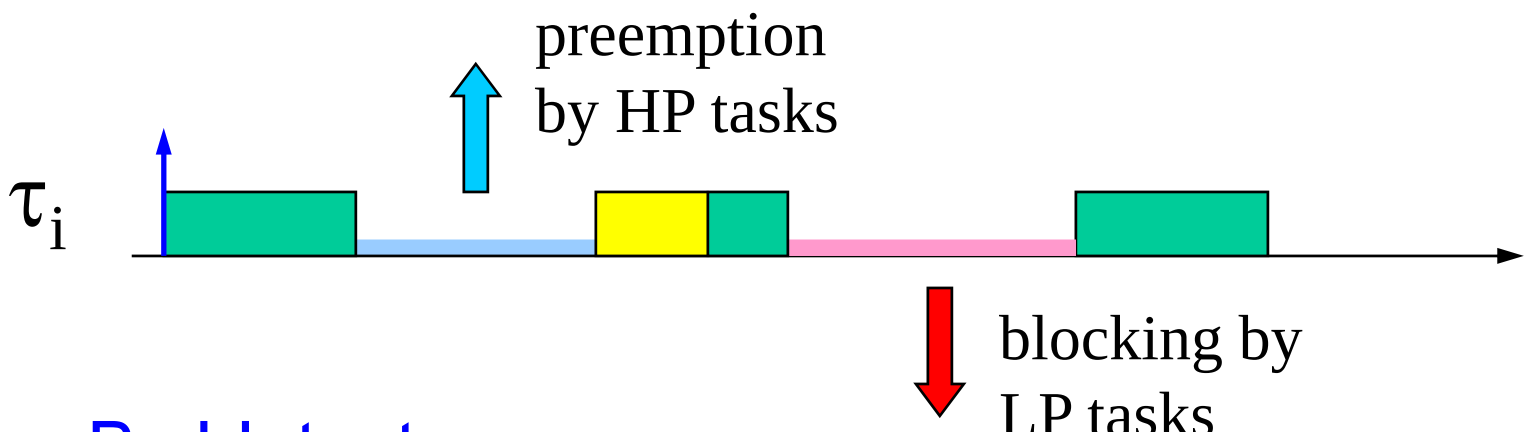
Deadlock avoidance with PCP



Guarantee with resource constraints

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

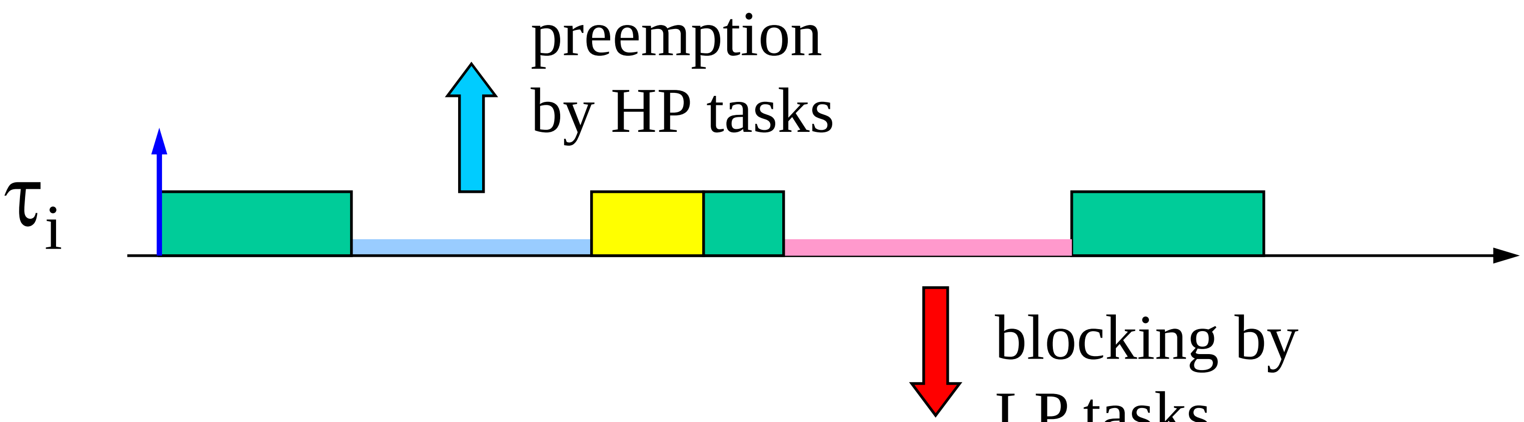
Guarantee with RM (D = T)



By LL test:

$$\forall i \quad \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i + B_i}{T_i} \leq i\left(2^{1/i} - 1\right)$$

Guarantee with RM (D ≤ T)



By RTA test: $\forall i \quad R_i \leq D_i$

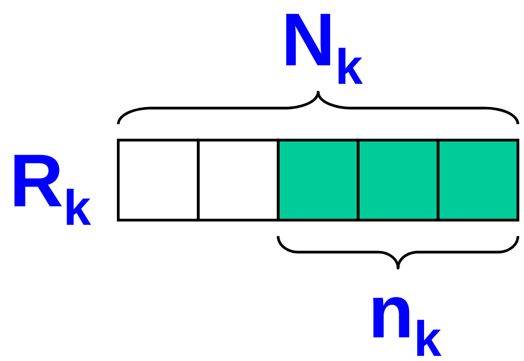
$$R_i = C_i + B_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

Stack Resource Policy [Baker 1990]

- It works both with fixed and dynamic priority
- It limits blocking to 1 critical section
- It prevents deadlock
- It supports multi-unit resources
- It allows stack sharing
- It is easy to implement

Stack Resource Policy [Baker 90]

- For each resource R_k:
  - ⇒ Maximum units: N_k
  - ⇒ Available units: n_k



- For each task τ_i the system keeps:
  - ⇒ its resource requirements: $\mu_i(R_k)$
  - ⇒ a priority p_i: RM $p_i \propto 1/T_i$ EDF $p_i \propto 1/d_i$
  - ⇒ a static preemption level: $\pi_i \propto 1/D_i$

Stack Resource Policy [Baker 90]

Resource ceiling

$$C_k(n_k) = \max_j \left\{ \pi_j : n_k < \mu_j(R_k) \right\}$$

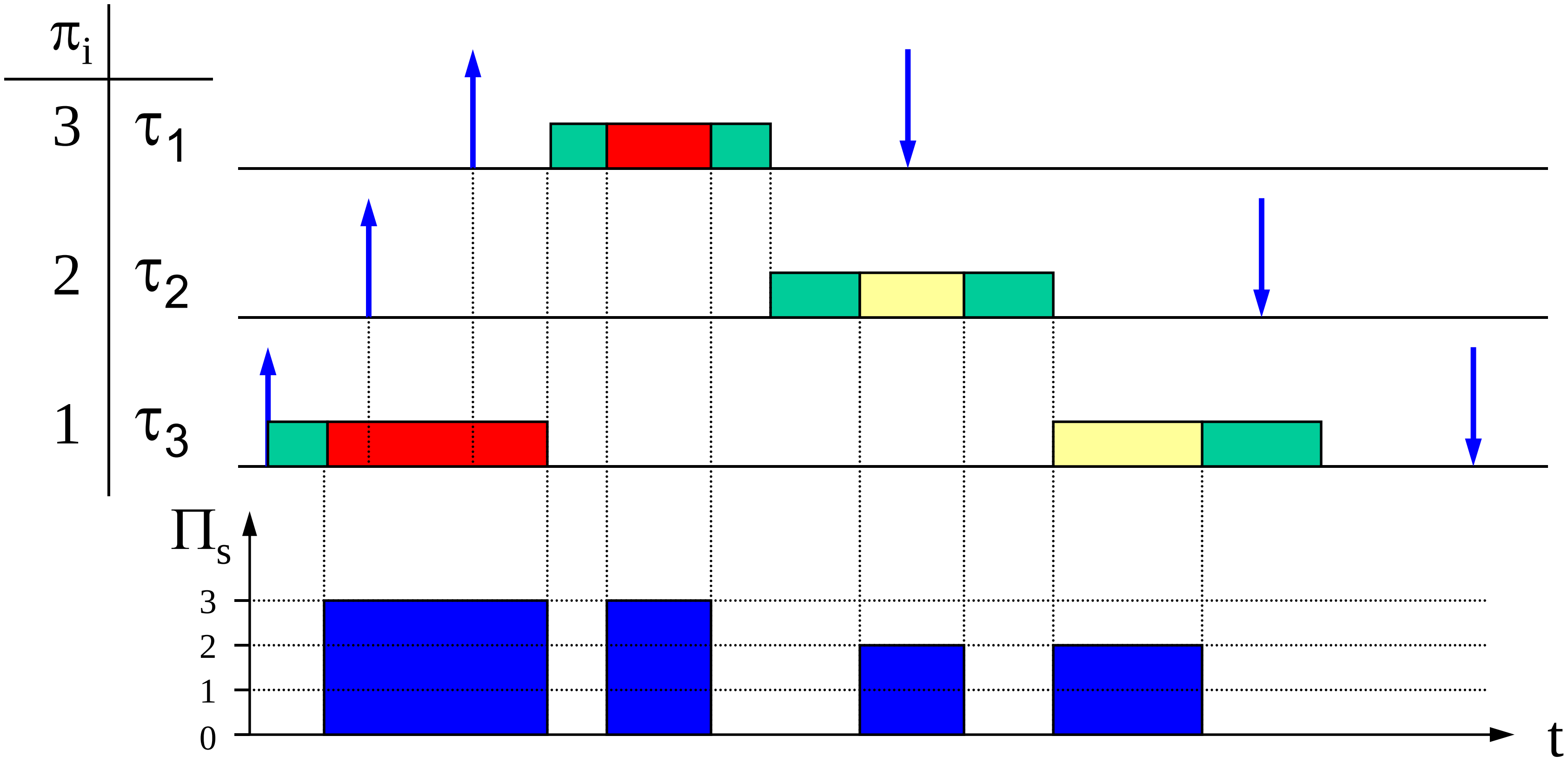
System ceiling

$$\Pi_s = \max_k \{ C_k(n_k) \}$$

SRP Rule

A job cannot preempt until p_i is the highest and $\pi_i > \Pi_s$

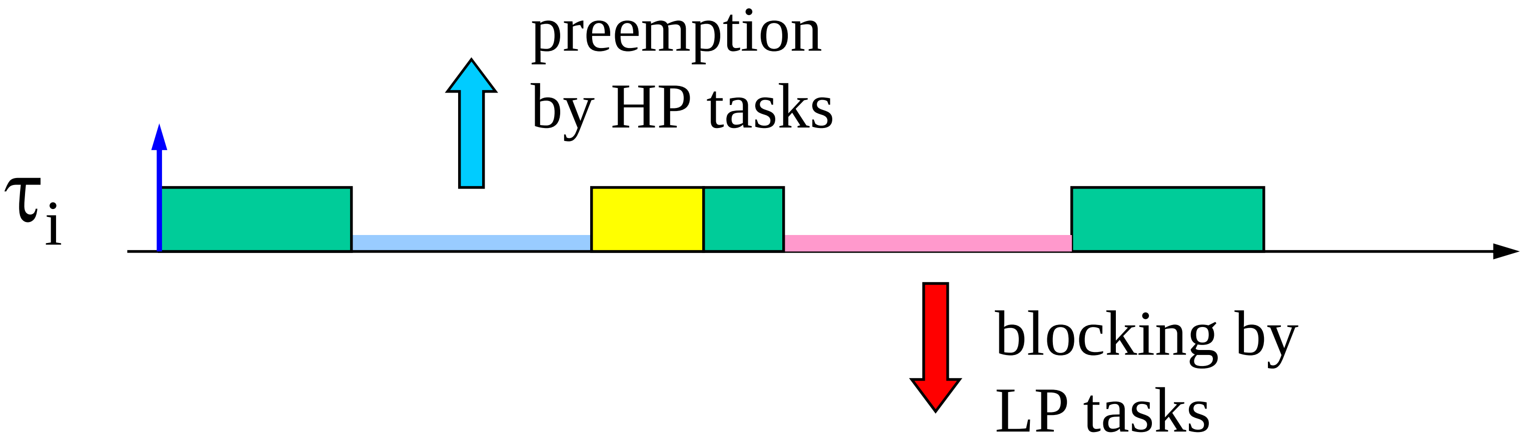
Example



SRP: Notes

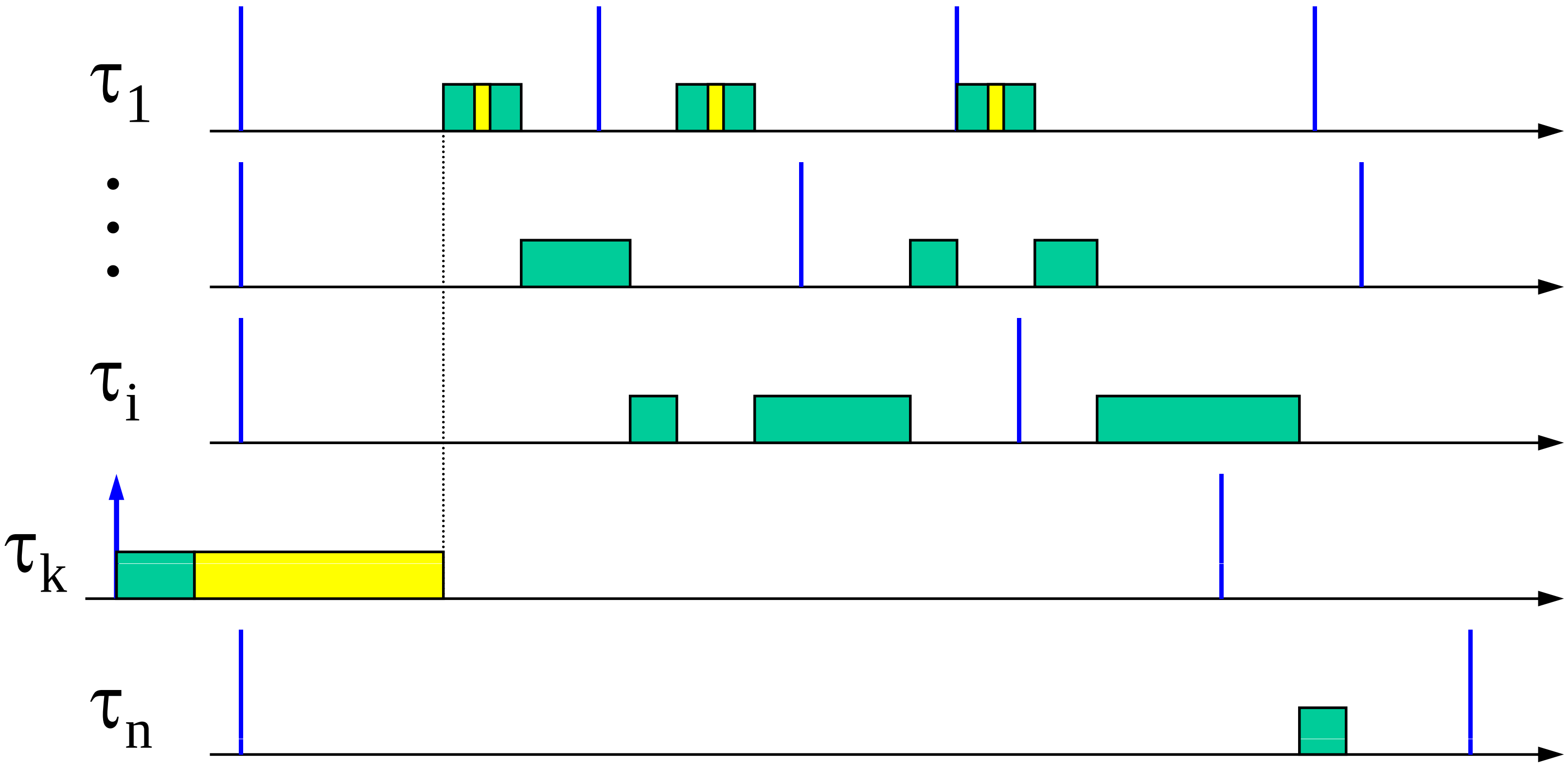
- Blocking always occurs at preemption time
- A task never blocks on a wait primitive (semaphore queues are not needed)
- Semaphores are still needed to update the system ceiling
- Early blocking allows stack sharing

EDF Guarantee (D$_i$ = T$_i$)



$$\forall i \qquad \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i + B_i}{T_i} \leq 1$$

EDF Guarantee: PD test $(D_i \leq T_i)$



Tasks are ordered by decreasing preemption level

Schedulability Analysis under EDF

When $D_i \leq T_i$

A task set is schedulable if $U < 1$ and $\forall L \in D$

$$\forall i \quad B_i + \sum_{k=1}^{n} \left\lfloor \frac{L + T_k - D_k}{T_k} \right\rfloor C_k \leq L$$

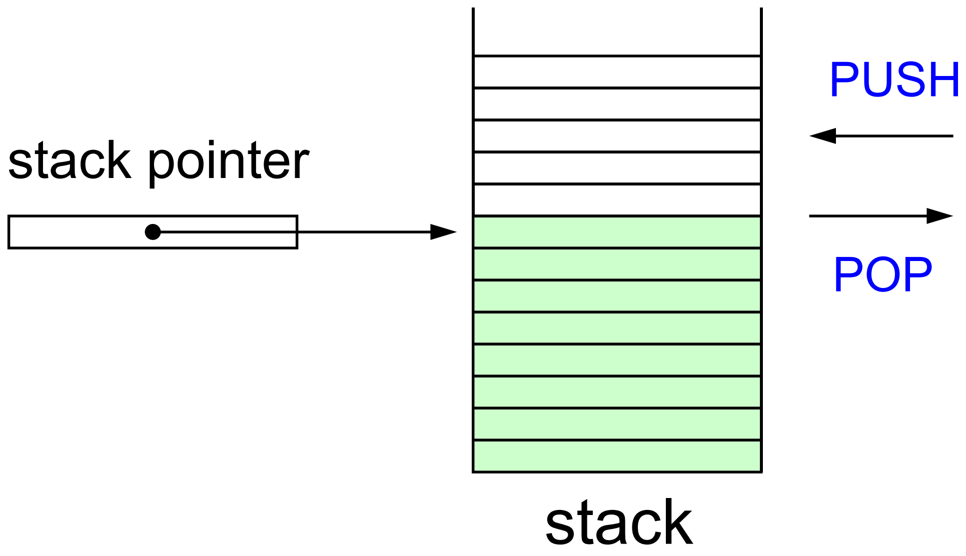
where $D = \{d_k \mid d_k \leq \min(H, L^*)\}$

$$H = lcm(T_1, \ldots, T_n) \qquad L^* = \frac{\sum_{i=1}^{n} (T_i - D_i) U_i}{1 - U}$$

Stack Sharing

Each task normally uses a private stack for

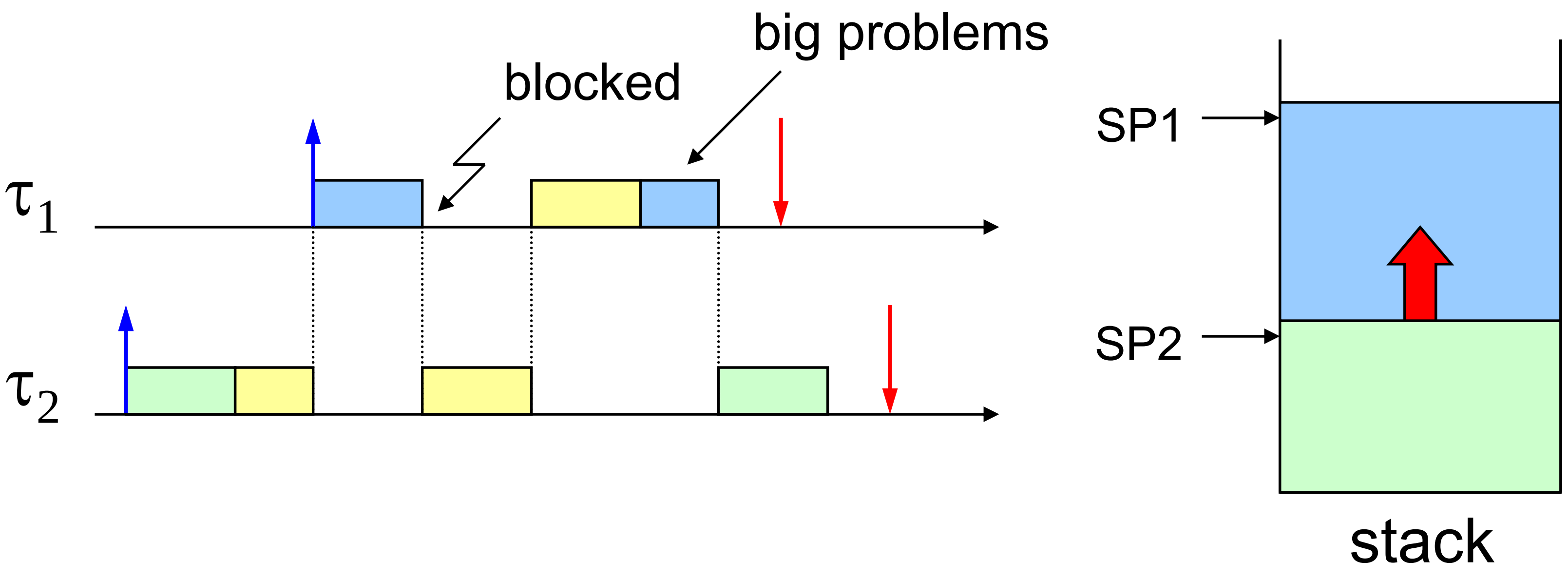
- saving context (register values)
- managing functions
- storing local variables



Stack Sharing

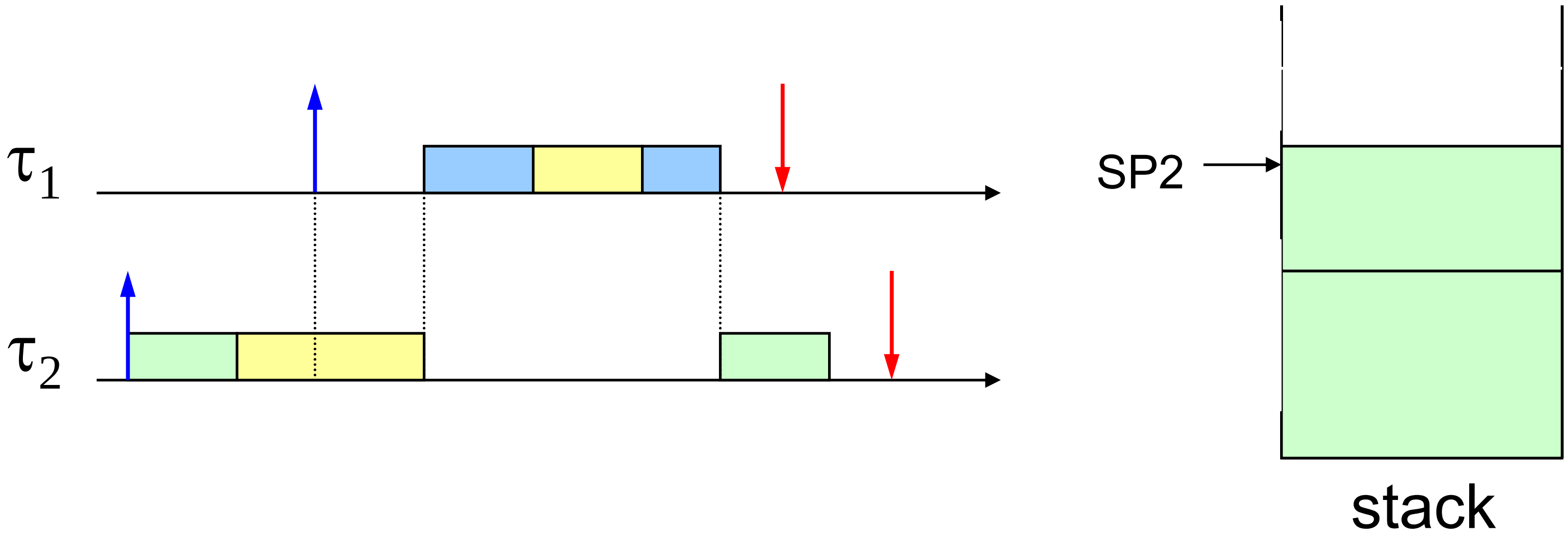
Why stack cannot be normally shared?

Suppose tasks share a resource: A



Stack Sharing

Why stack can be shared under SRP?



Saving Stack Size

To really save stack size, we should use a small number of preemption levels.

100 tasks

10 Kb stack per task

$\rightarrow$ stack size = 1 Mb

10 preemption levels

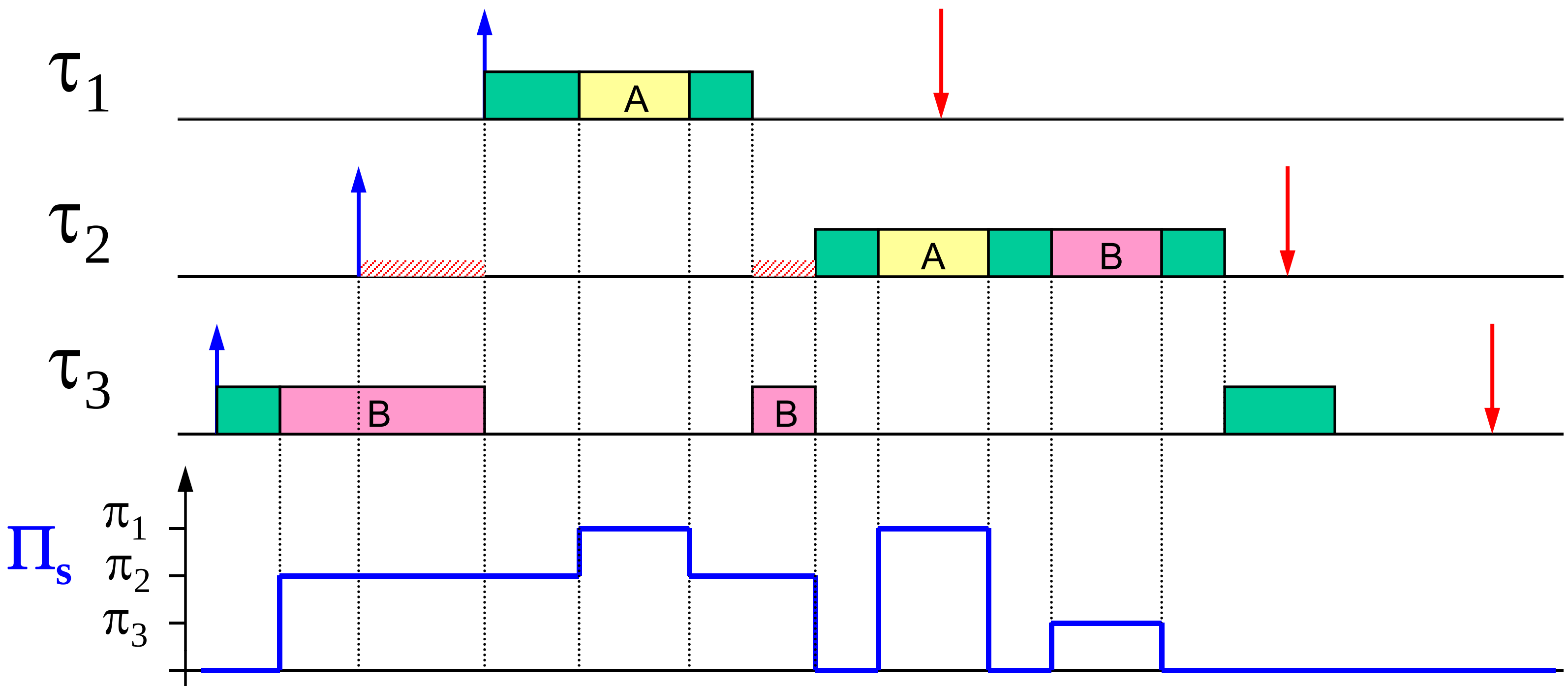
10 tasks per group

$\rightarrow$ stack size = 100 Kb

stack saving = 90 %

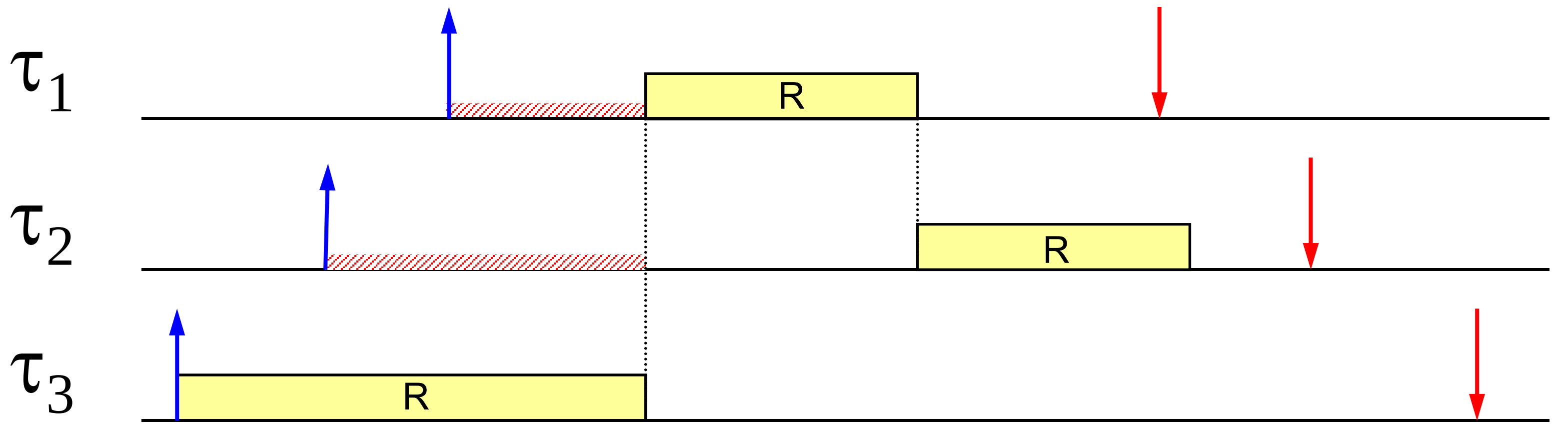
NOTE on SRP

- SRP for fixed priorities and single-unit resources is equivalent to Higher Locker Priority.
- It is also referred to as Immediate Priority Ceiling



Non-preemtive scheduling

It is a special case of preemptive scheduling where all tasks share a single resource for their entire duration.



The max blocking time for task τ_i is given by the largest C_k among the lowest priority tasks:

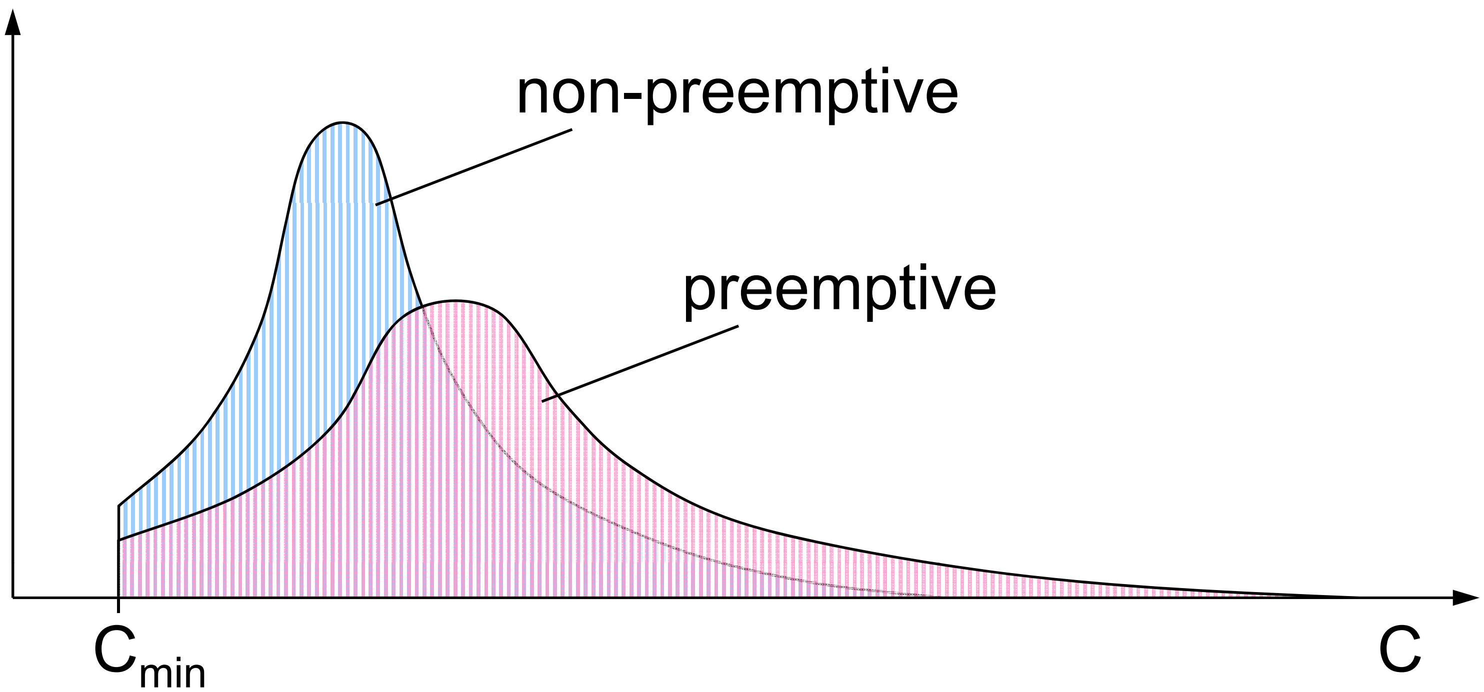
$$B_i = \max\{C_k : P_k < P_i\}$$

Advantages of NP scheduling

- Reduces runtime overhead
  - Less context switches
  - No semaphores are needed for critical sections
- Reduces stack size, since no more than one task can be in execution.
- Preserves program locality, improving the effectiveness of
  - Cache memory
  - Pipeline mechanisms
  - Prefetch queues

Advantages of NP scheduling

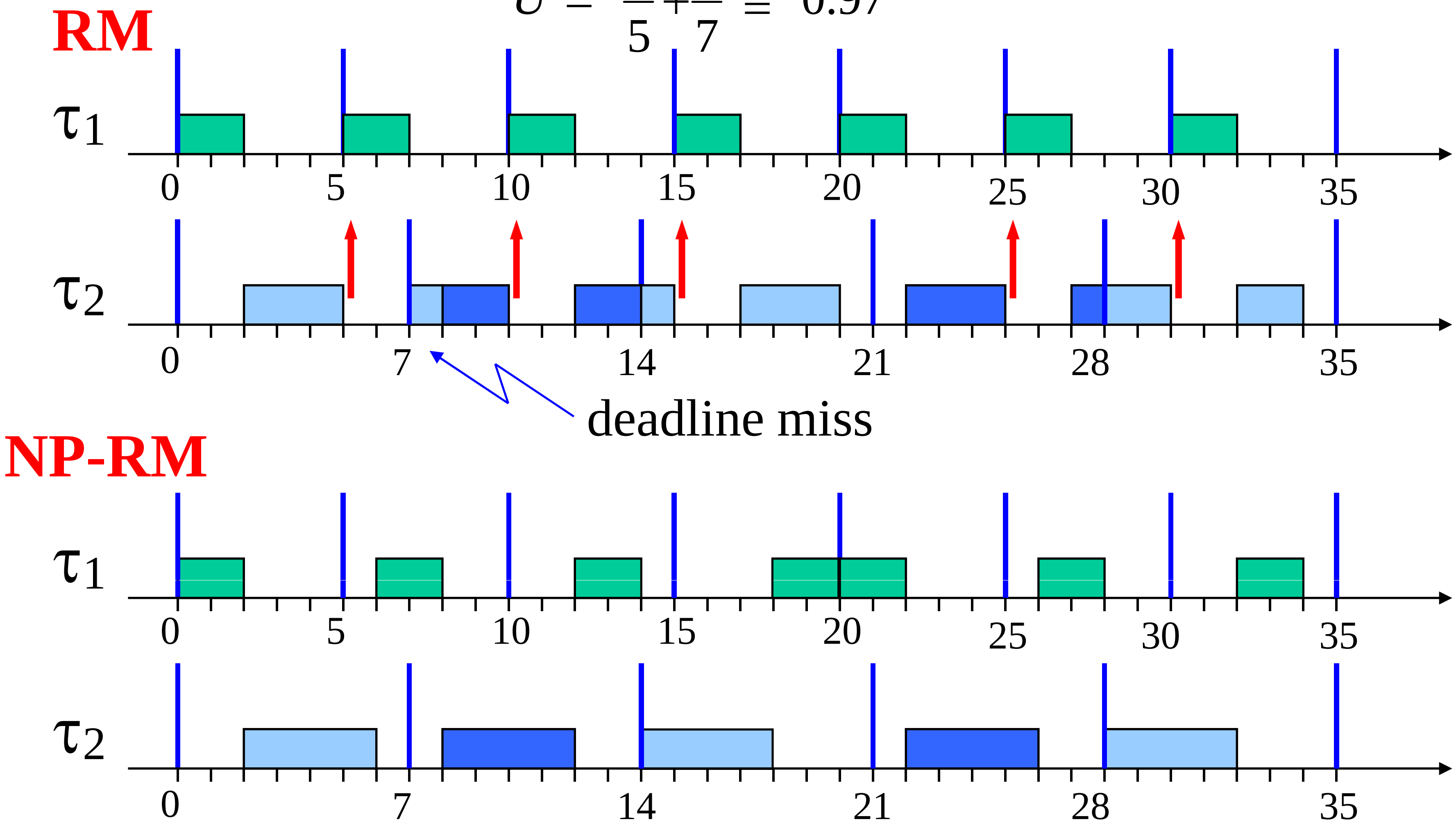
- As a consequence, task execution times are
  - Smaller
  - More predictable (less variable)



Advantages of NP scheduling

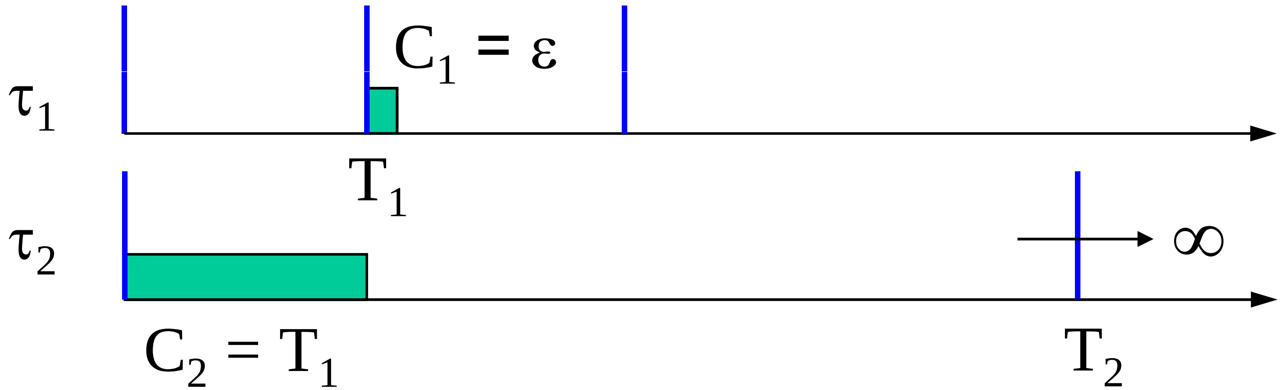
In fixed priority systems can improve schedulabiilty:

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



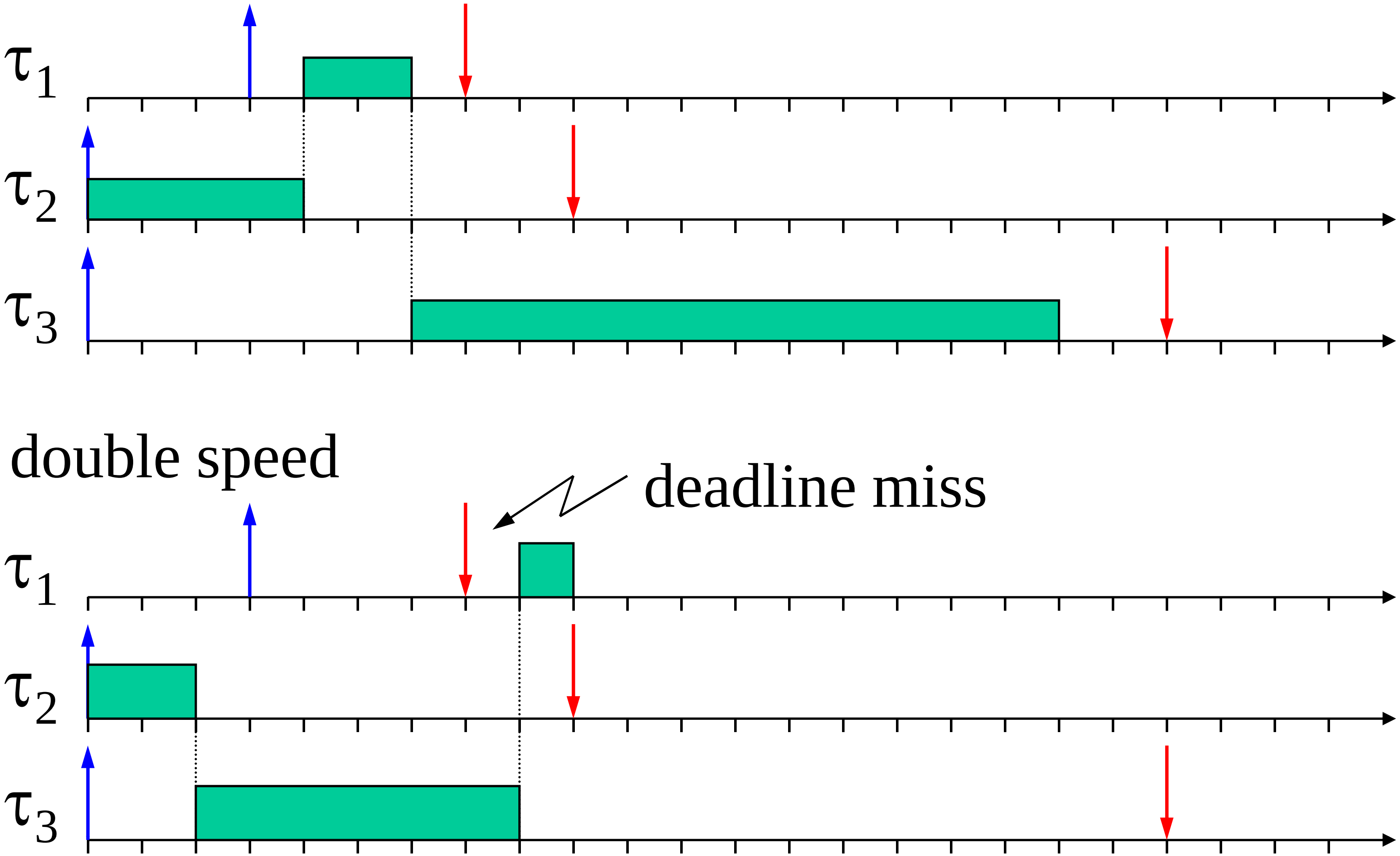
Disadvantages of NP scheduling

- In general, NP scheduling reduces schedulability.
- The utilization bound under non preemptive scheduling drops to zero:



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

Non preemptive scheduling anomalies



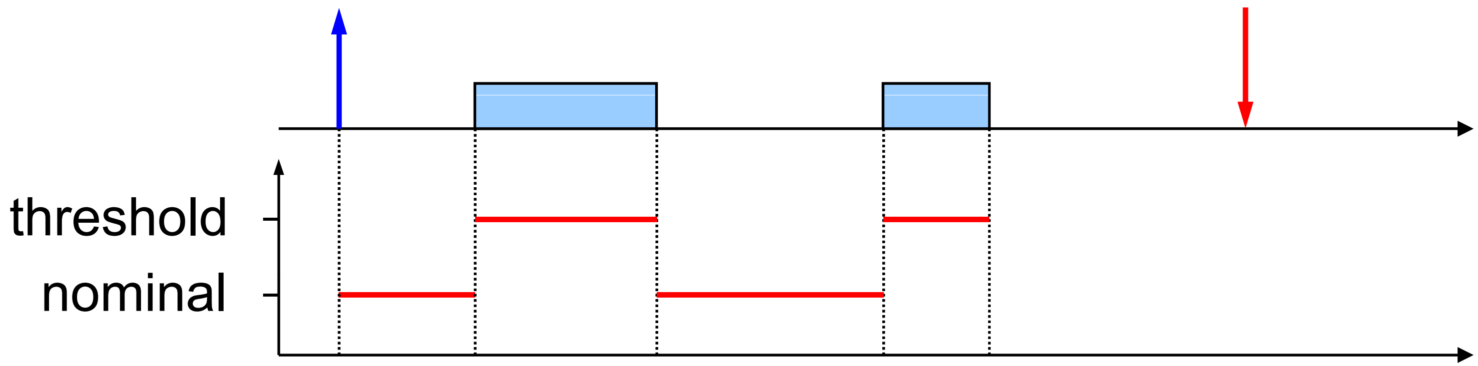
Trade-off solutions

Preemption thresholds

Each task has two priorities:

- Nominal priority (ready priority): used to enqueue the task in the ready queue
- Threshold priority: used for task execution

nominal priority $\leq$ threshold priority

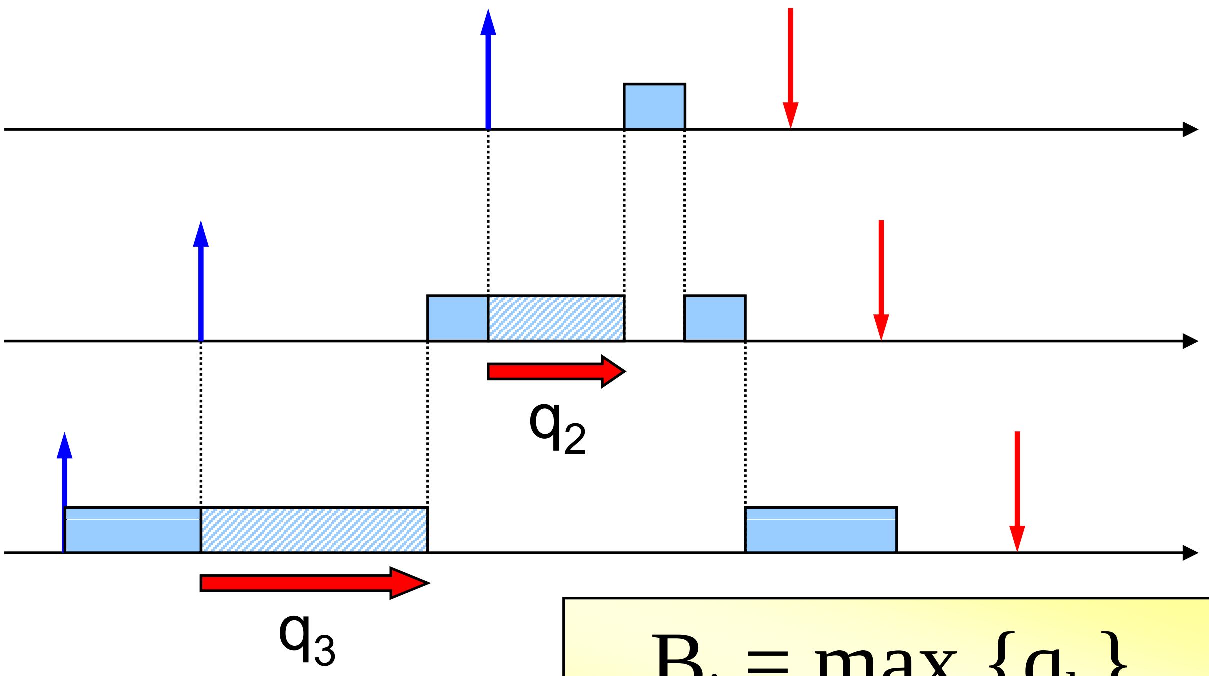


Trade-off solutions

Deferred preemption

Each task can defer preemption up to q_i

NP regions are floating in the code



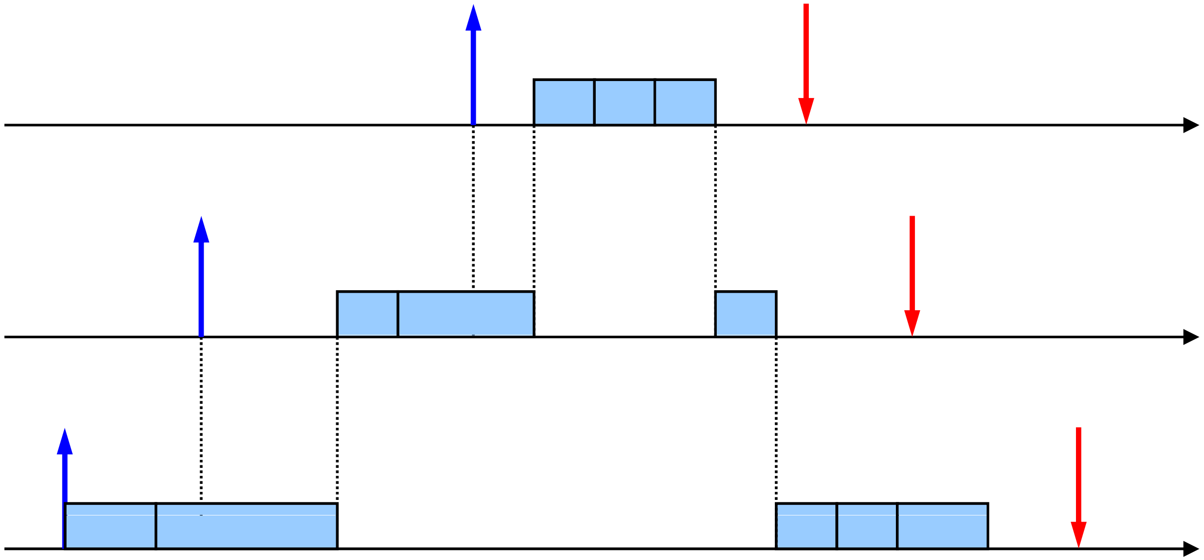
$$B_i = \max_{k>i} \{q_k\}$$

Trade-off solutions

Fixed preemption points

A task can only be preempted in fixed points

and it is divided in m_i chunks: $q_{i,1}$ ... q_{i,m_i}



$$B_i = \max_{k>i} \{q_k^{max}\}$$

Interesting problem

Given a preemptively feasible task set, reduce preemptions as much as possible still preserving schedulability.

→ Reducing context switch costs and WCETs

This means finding the longest non-preemtive chunk for each task that can still preserve schedulability.

- Under EDF → Baruah - ECRTS 2005
- Under Fix. Pr. → Yao et. al. - RTCSA 2009

Handling Aperiodic Tasks

Handling Criticality

- Aperiodic tasks with **HARD** deadlines must be guaranteed under worst-case conditions.
- Off-line guarantee is only possible if we can bound interarrival times (**sporadic tasks**).
- Hence **sporadic tasks** can be guaranteed as periodic tasks with $C_i = WCET_i$ and $T_i = MIT_i$

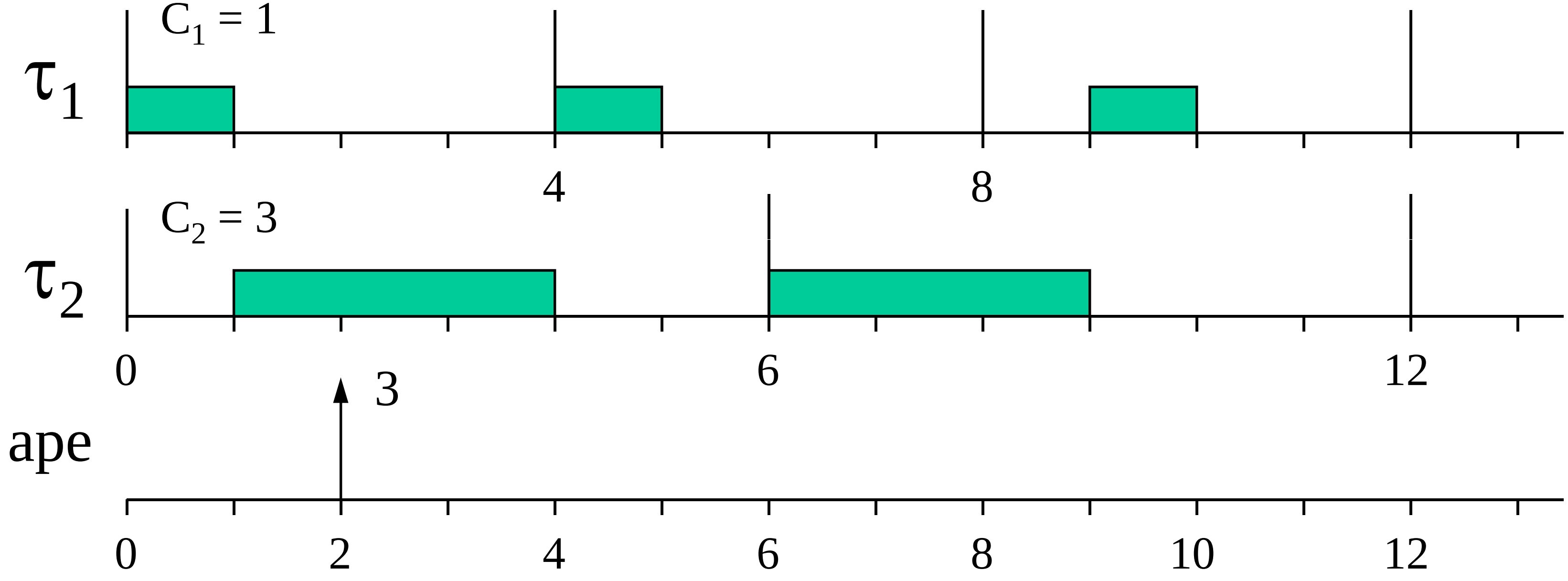
$$\left(\begin{array}{lcl} \text{WCET} & = & \text{Worst-Case Execution Time} \\ \text{MIT} & = & \text{Minimum Interarrival Time} \end{array} \right)$$

SOFT aperiodic tasks

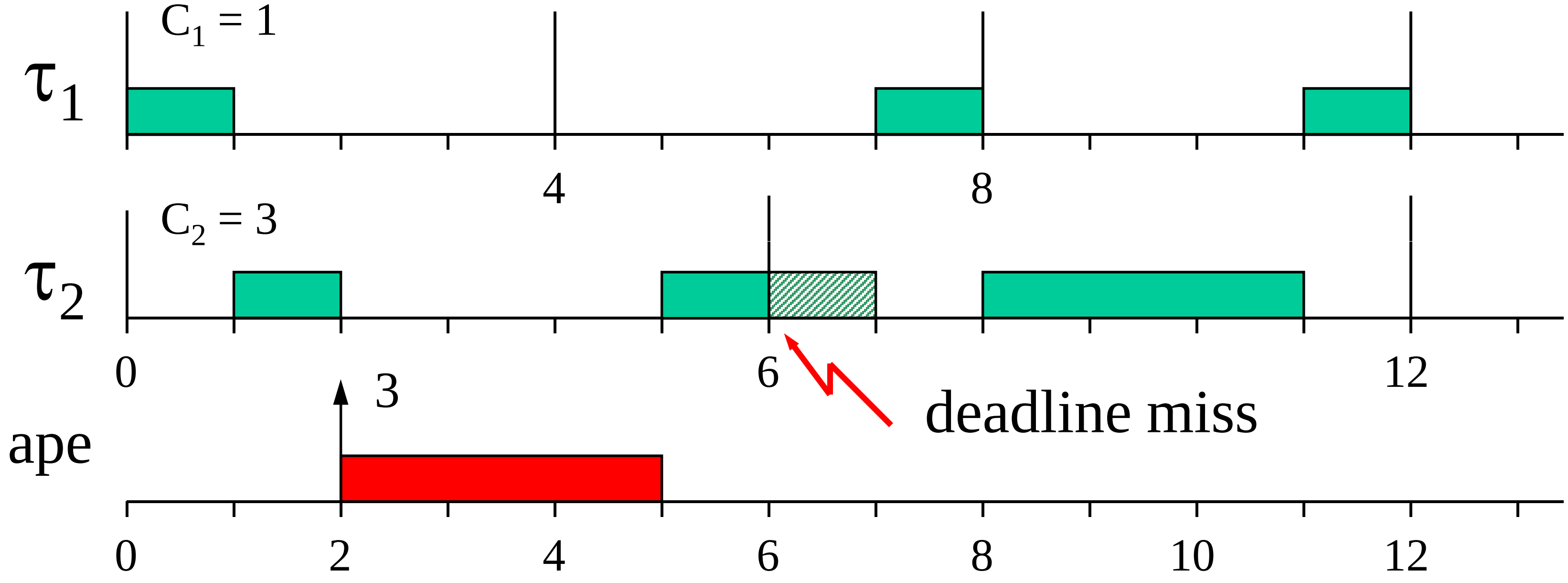
- Aperiodic tasks with **SOFT** deadlines should be executed as soon as possible, but without jeopardizing HARD tasks.
- We may be interested in
  - → minimizing the average response time
  - → performing an on-line guarantee

Periodic Scheduling

(EDF)

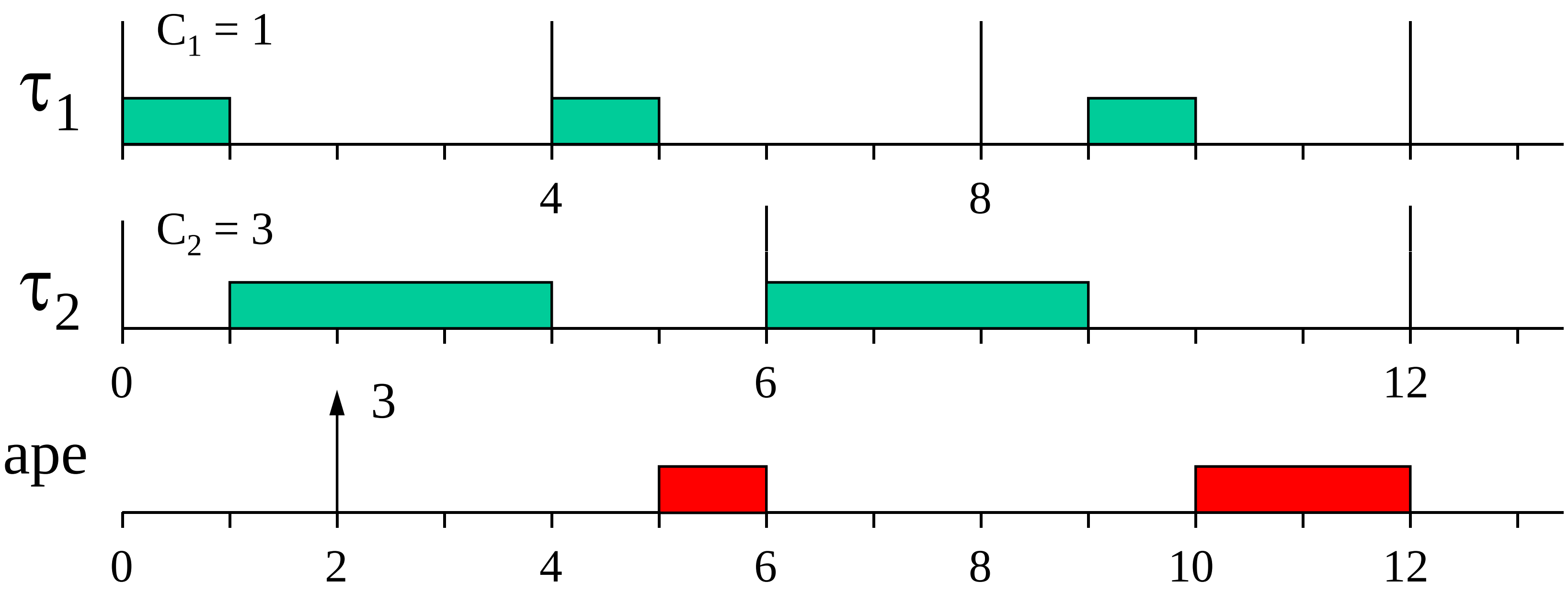


Immediate service



Response Time = 3

Background service



Response Time = 10

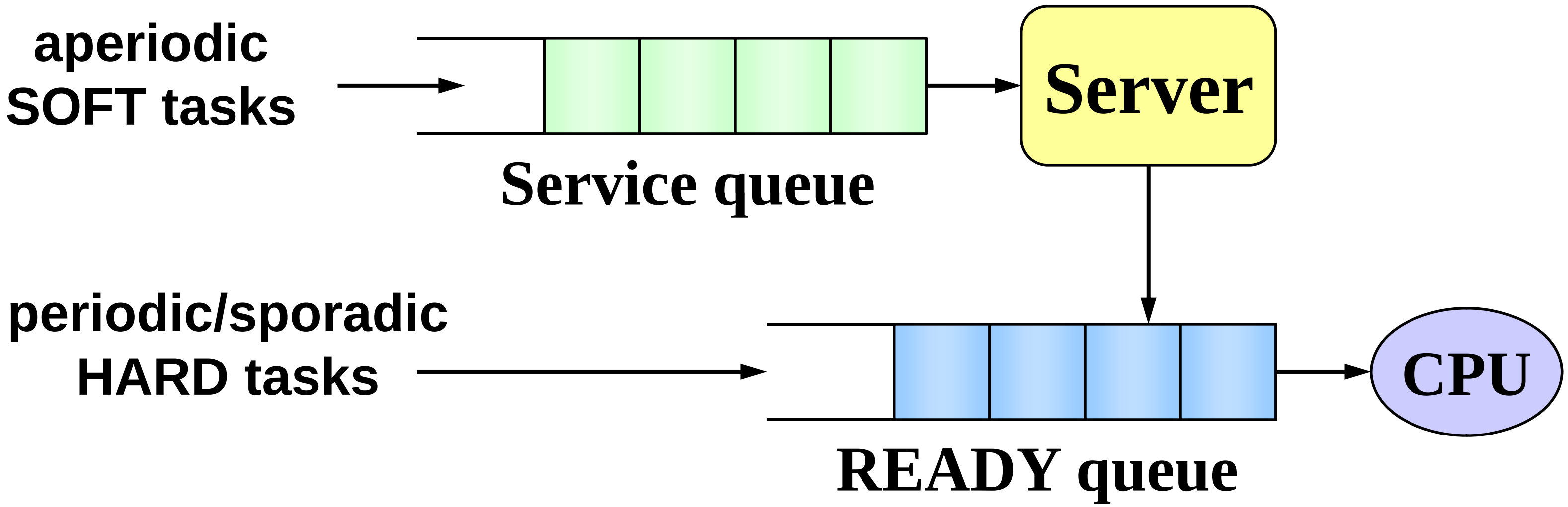
Aperiodic Servers

- A server is a kernel activity aimed at controlling the execution of aperiodic tasks.
- Normally, a server is a periodic task having two parameters:

$$\begin{cases} C_s & \text{capacity (or budget)} \\ T_s & \text{server period} \end{cases}$$

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

Aperiodic service queue



- The server is scheduled as any periodic task.
- Priority ties are broken in favor of the server.
- Aperiodic tasks can be selected using an arbitrary queueing discipline.

Fixed-priority Servers

- Polling Server
- Deferrable Server
- Sporadic Server
- Slack Stealer

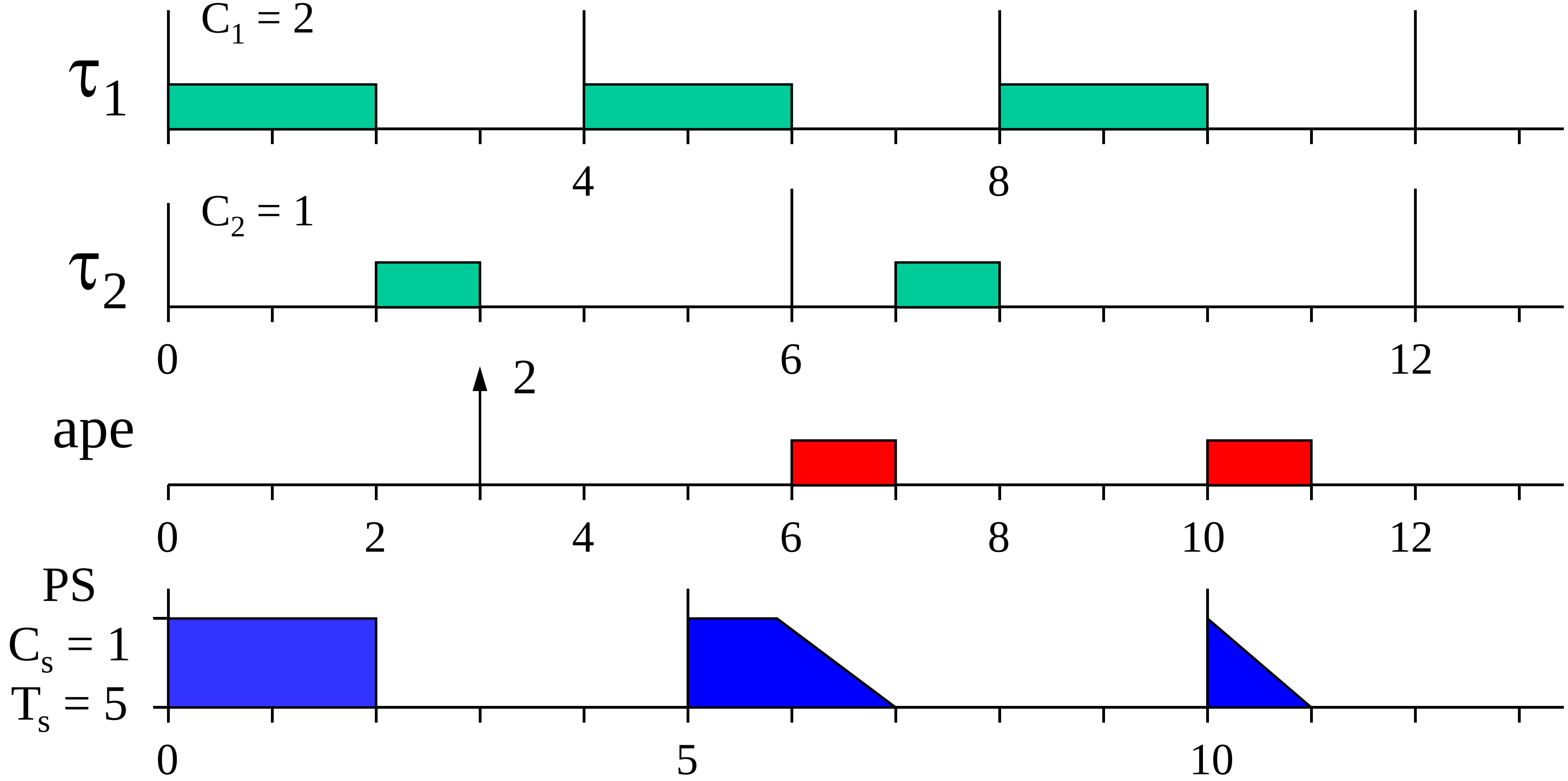
Dynamic-priority Servers

- Dynamic Polling Server
- Dynamic Sporadic Server
- Total Bandwidth Server
- Tunable Bandwidth Server
- Constant Bandwidth Server

Polling Server (PS)

- At the beginning of each period, the budget is recharged at its maximum value.
- Budget is consumed during job execution.
- When the server becomes active and there are no pending jobs, C_s is discharged to zero.
- When the server becomes active and there are pending jobs, they are served until $C_s > 0$.

RM + Polling Server



Response Time = 8

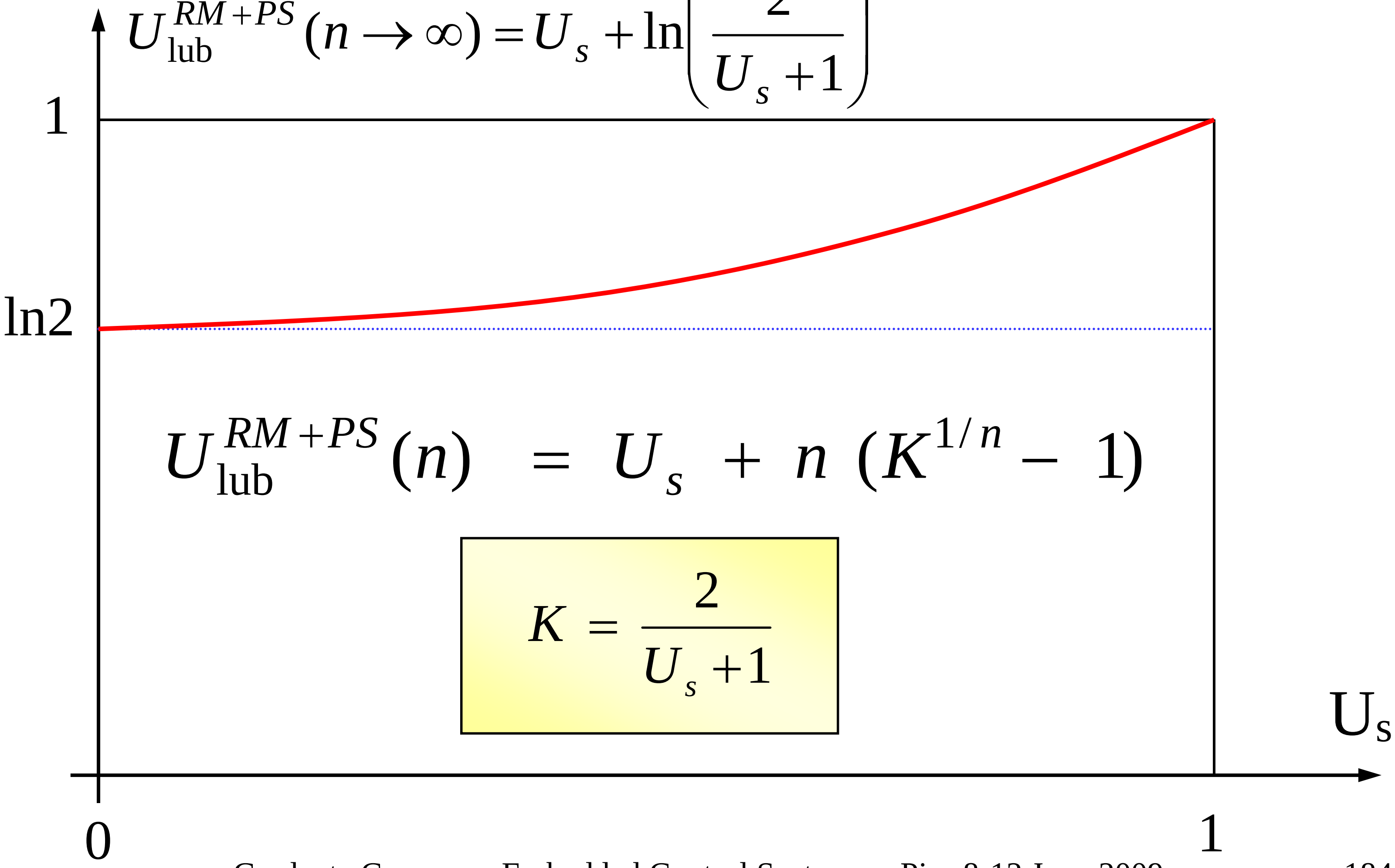
PS properties

- In the worst-case, the PS behaves as a periodic task with utilization $U_s = C_s/T_s$.
- Aperiodic tasks execute at the highest priority if $T_s = \min(T_1, \ldots, T_n)$.
- Liu & Layland analysis gives that:

$$U_{\text{lub}}^{RM+PS}(n) = U_s + n\left[\left(\frac{2}{U_s+1}\right)^{1/n} - 1\right]$$

RM + PS schedulability

$$U_{\text{lub}}^{RM+PS}(n \to \infty) = U_s + \ln\left(\frac{2}{U_s+1}\right)$$

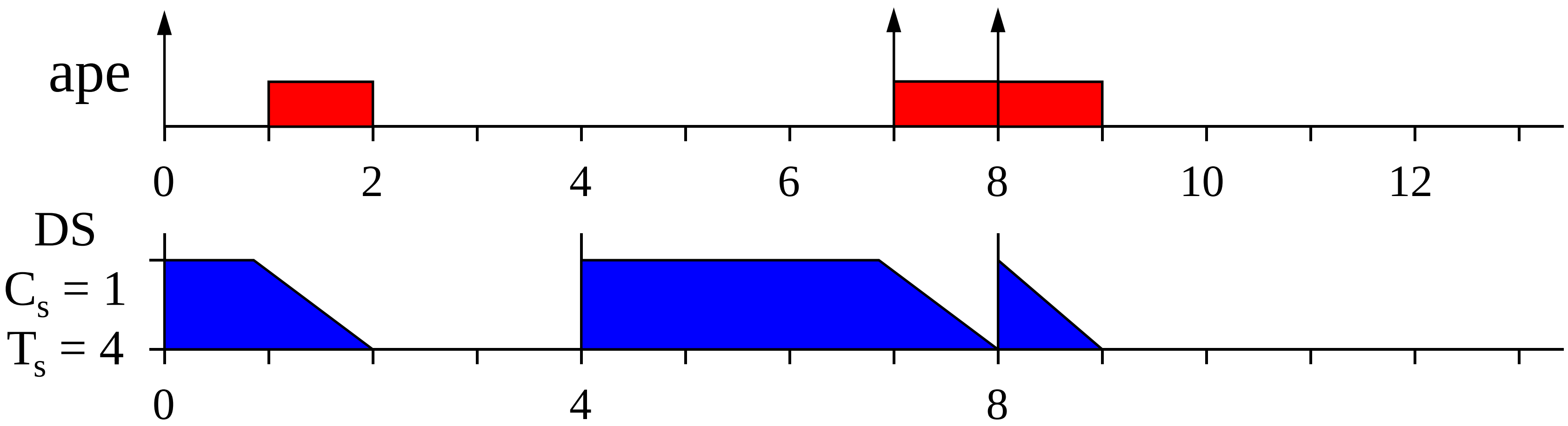


$$U_{\text{lub}}^{RM+PS}(n) = U_s + n\,(K^{1/n} - 1)$$

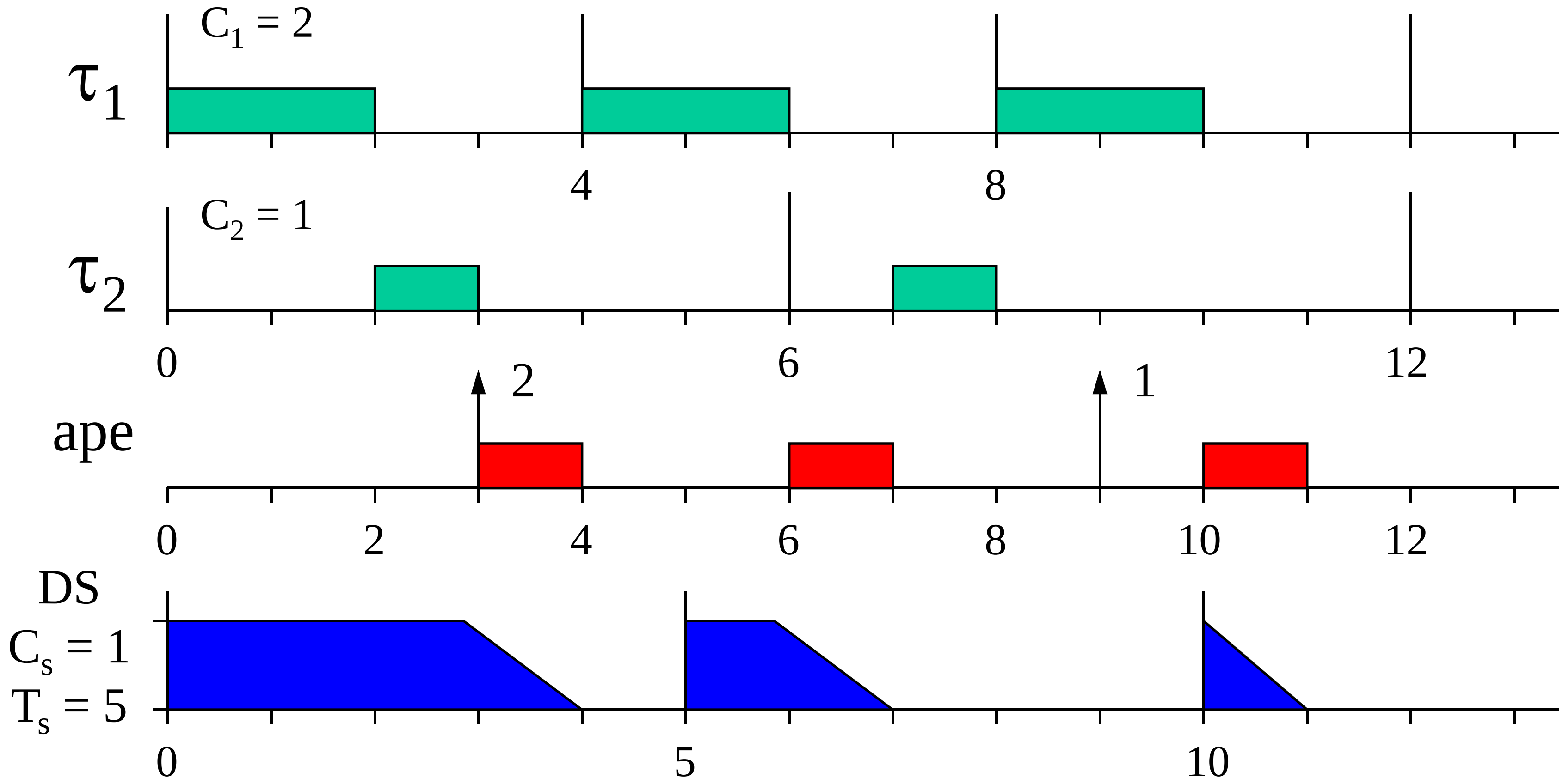
$$K = \frac{2}{U_s+1}$$

Deferrable Server (DS)

- Is similar to the PS, but the budget is not discharged if there are no pending requests.
- Keeping the budget improves responsiveness, but decreases the utilization bound.

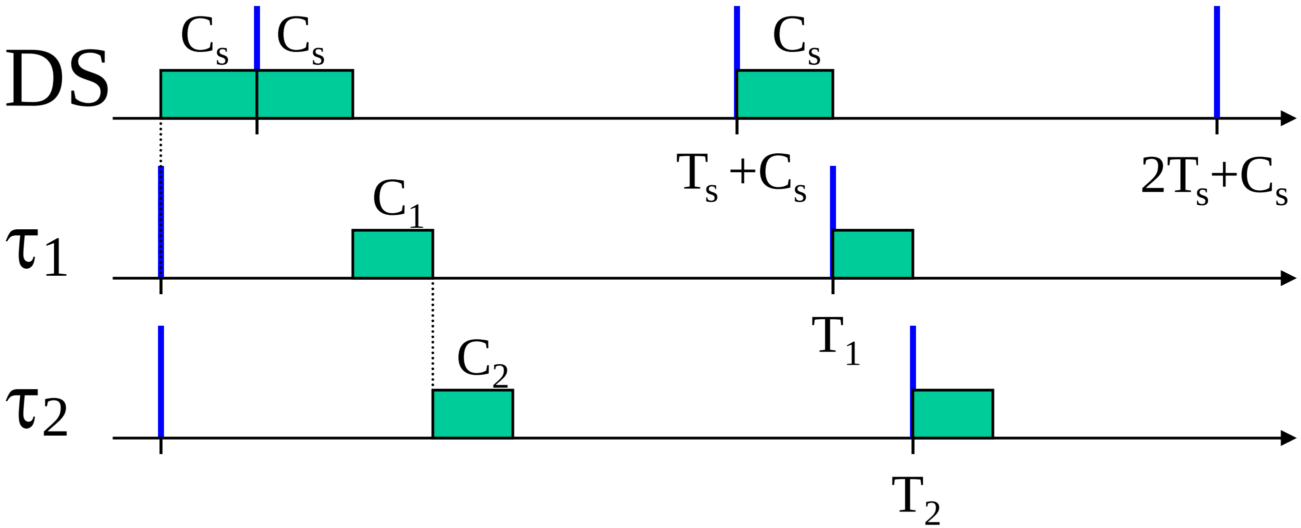


RM + Deferrable Server



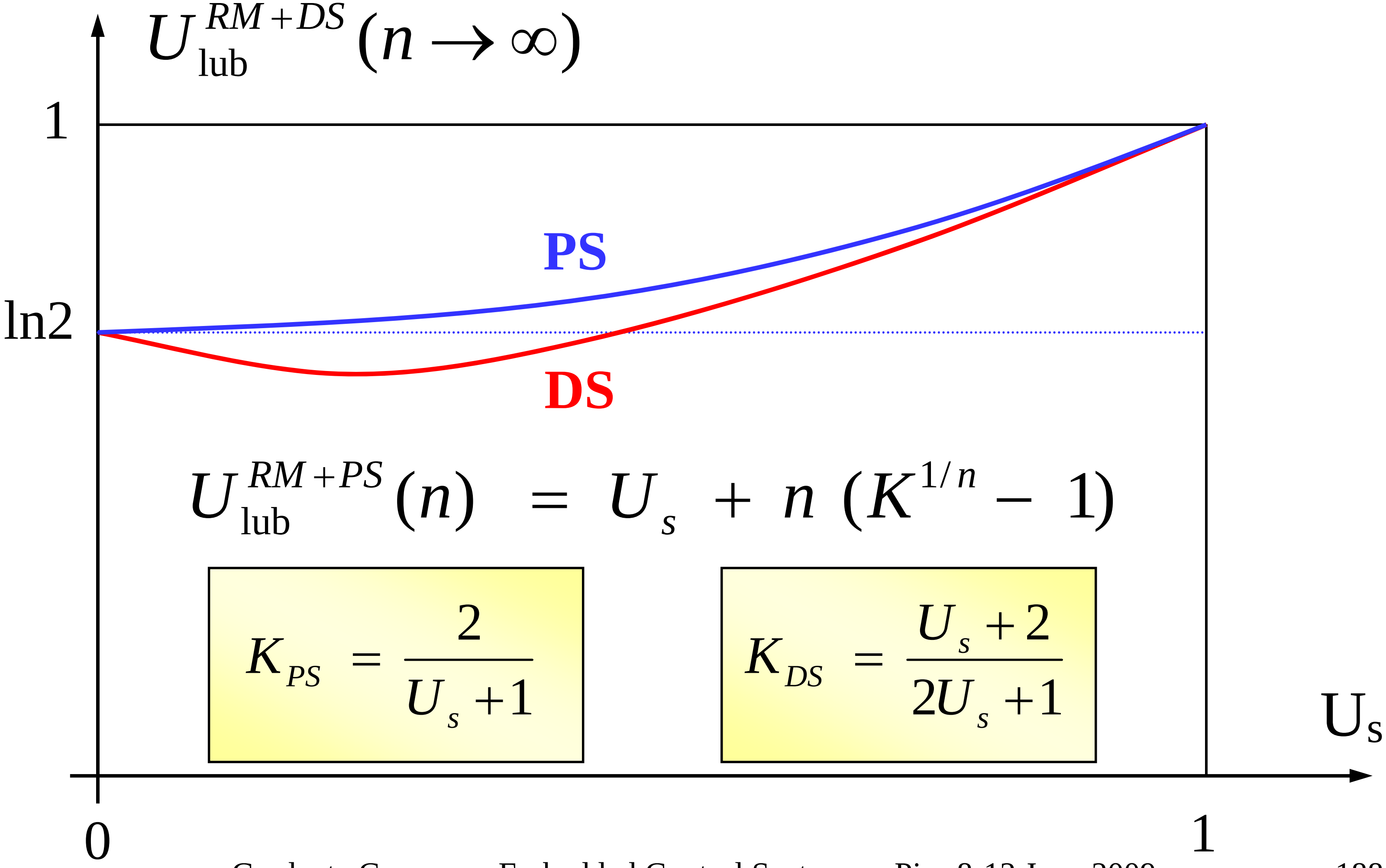
Response Time = 4

Analysis of RM + DS



$$U_{\text{lub}}^{RM+DS}(n) = U_s + n\left[\left(\frac{U_s+2}{2U_s+1}\right)^{1/n} - 1\right]$$

RM + DS schedulability



$$U_{\text{lub}}^{RM+PS}(n) = U_s + n\,(K^{1/n} - 1)$$

$$K_{PS} = \frac{2}{U_s + 1}$$

$$K_{DS} = \frac{U_s + 2}{2U_s + 1}$$

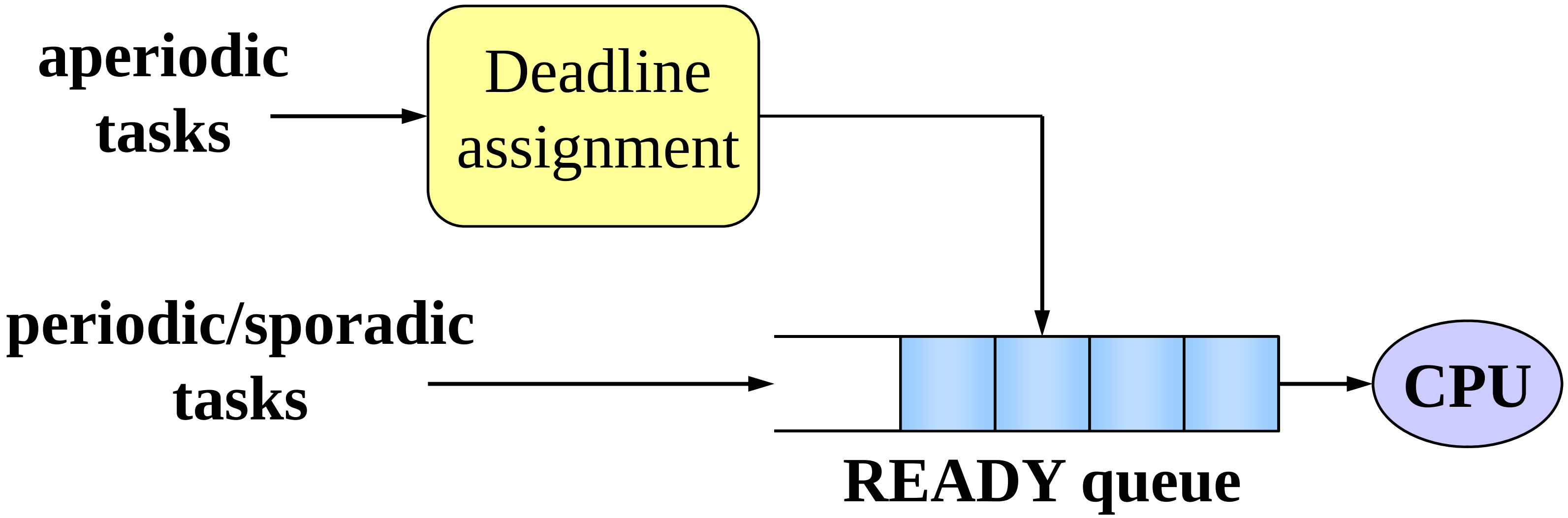
Designing server parameters

- Determine U_s^{max} from $$U_p \leq n \left[\left(\frac{U_s + 2}{2U_s + 1} \right)^{1/n} - 1 \right]$$
- Define $U_s \leq U_s^{max}$
- Define $T_s = \min(T_1, \ldots, T_n)$
- Compute $C_s = U_s T_s$

Total Bandwidth Server (TBS)

- It is a dynamic priority server, used along with EDF.
- Each aperiodic request is assigned a deadline so that the server demand does not exceed a given bandwidth U_s .
- Aperiodic jobs are inserted in the ready queue and scheduled together with the HARD tasks.

The TBS mechanism



- Deadlines ties are broken in favor of the server.
- Periodic tasks are guaranteed *if and only if*

$$\boxed{U_p + U_s \leq 1}$$

Deadline assignment rule

- Deadline has to be assigned not to jeopardize periodic tasks.

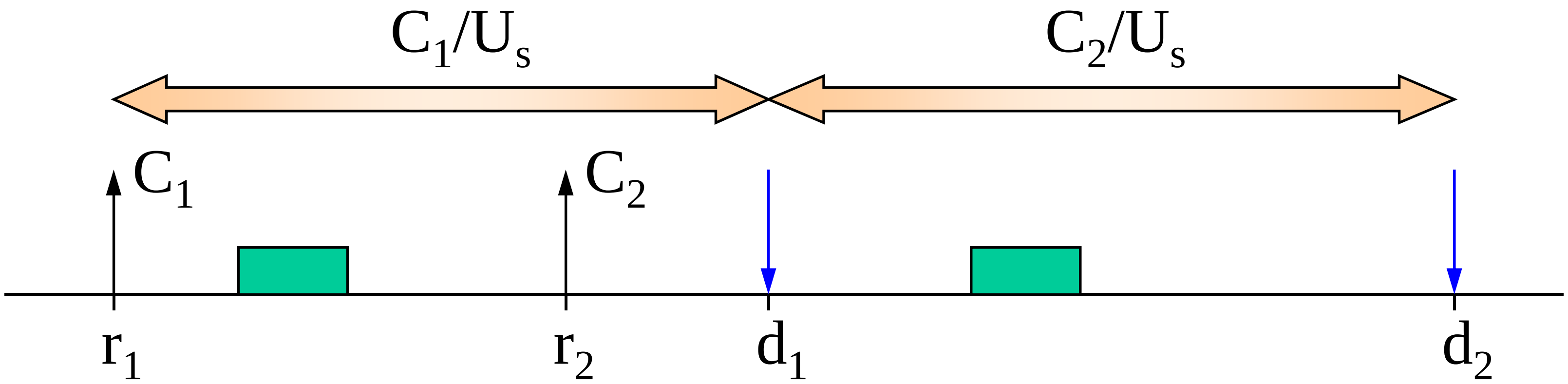
- A safe relative deadline is equal to the minimum period that can be assigned to a new periodic task with utilization U_s:

$$U_s = C_k / T_k \quad \Rightarrow \quad T_k = d_k - r_k = C_k / U_s$$

- Hence, the absolute deadline can be set as:

$$\boxed{d_k = r_k + C_k / U_s}$$

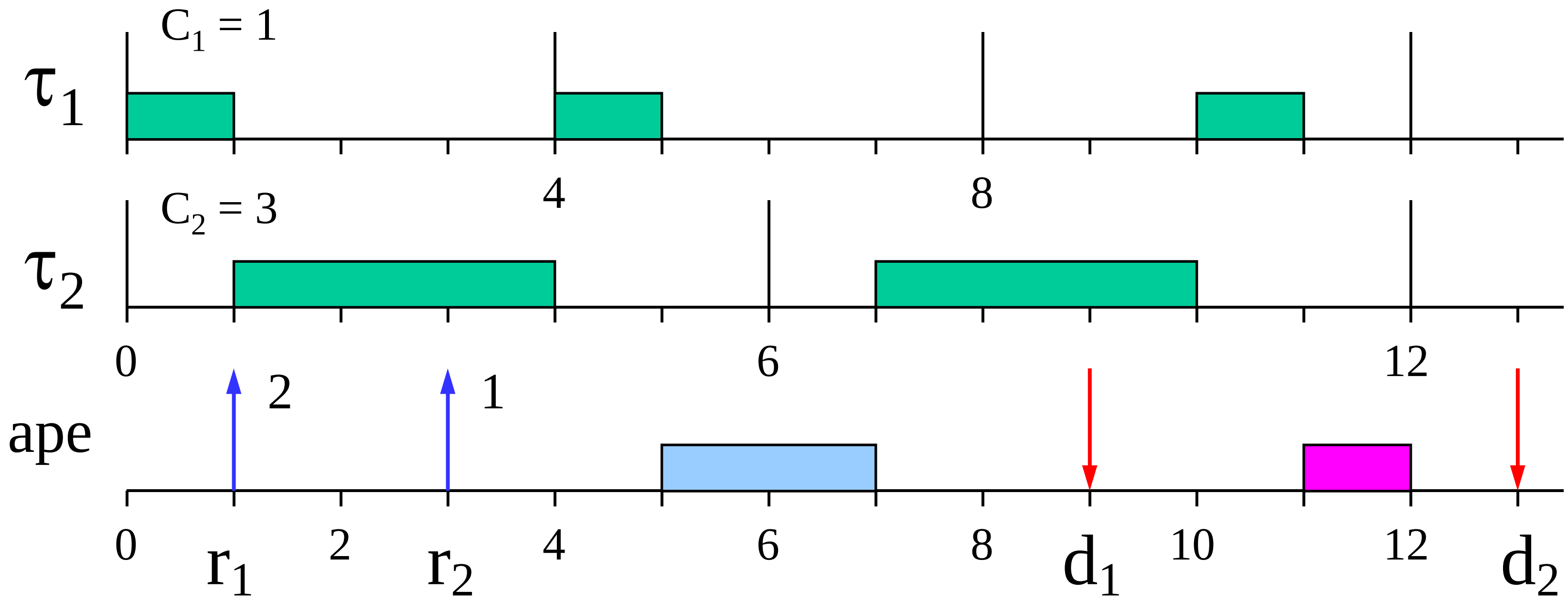
Deadline assignment rule



- To keep track of the bandwidth assigned to previous jobs, d_k must be computed as:

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

EDF + TBS schedule

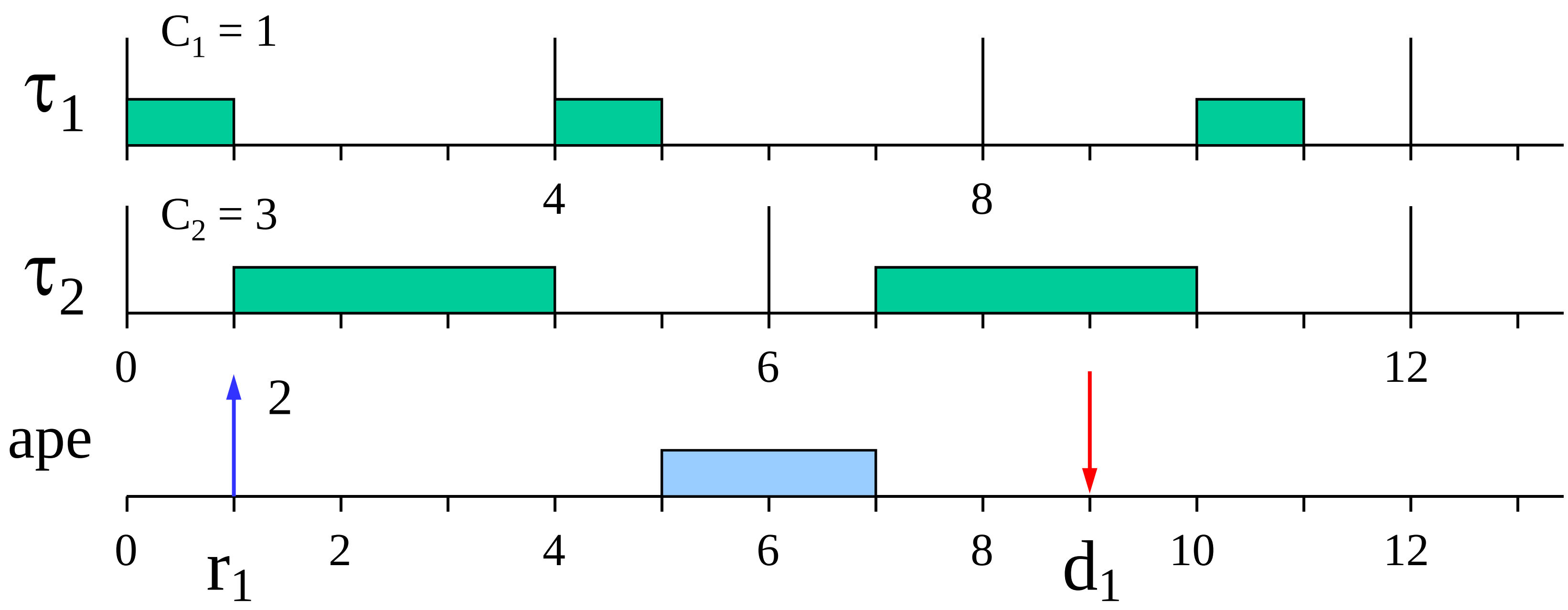


$$U_s = 1 - U_p = 1/4$$

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

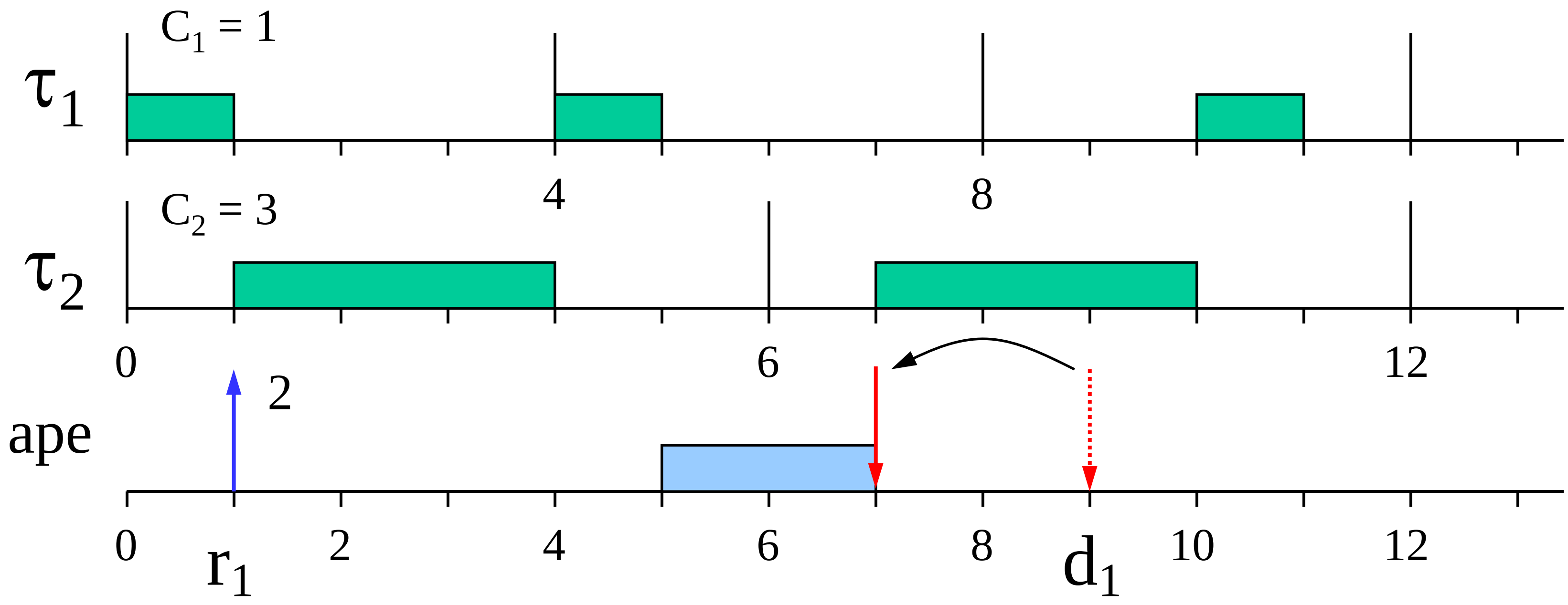
Improving TBS

- What's the minimum deadline that can be assigned to an aperiodic job?



Improving TBS

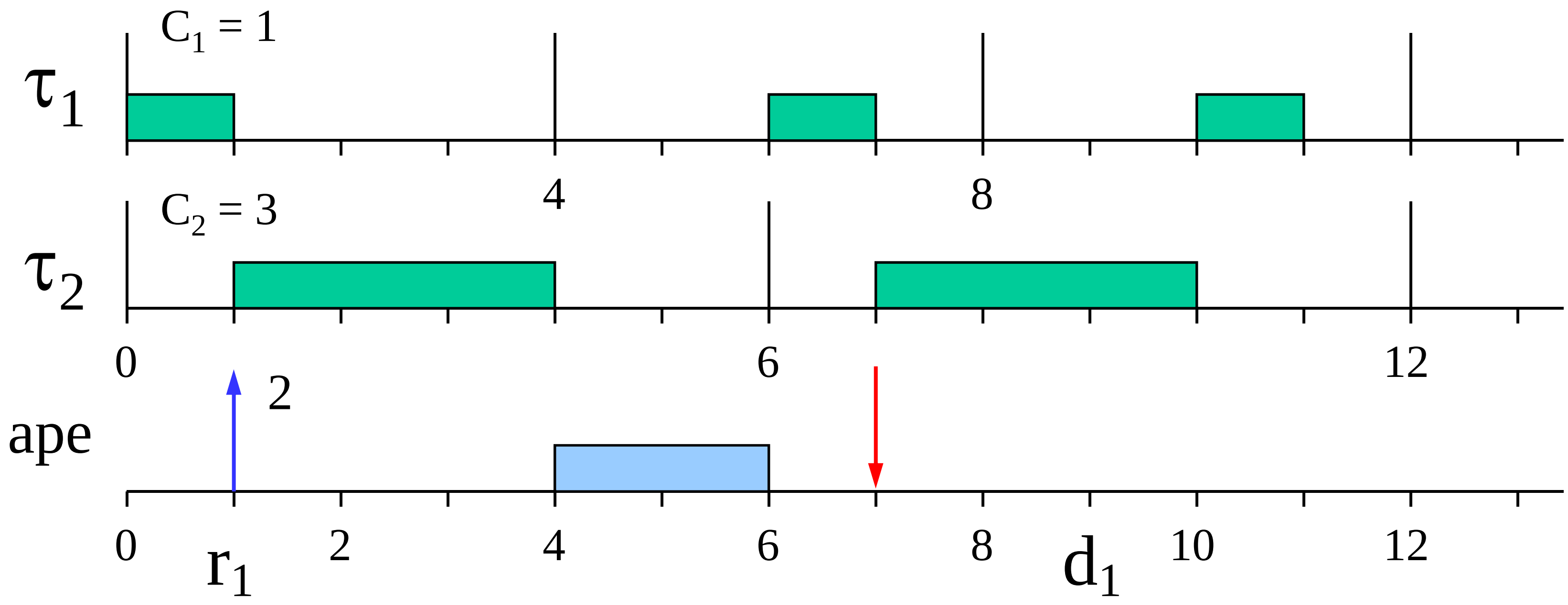
- If we freeze the schedule and advance d_1 to 7, no task misses its deadline, but the schedule is not EDF:



Feasible schedule ≠ EDF

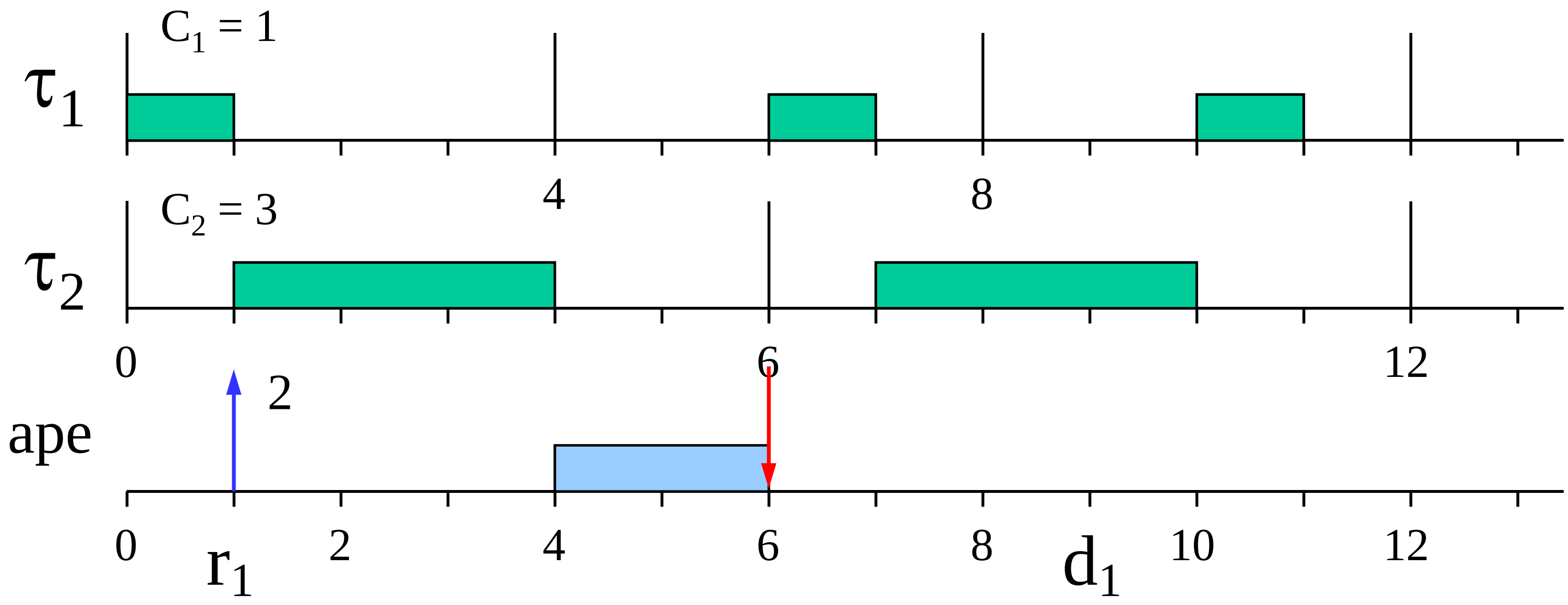
Improving TBS

- However, since EDF is optimal, the schedule produced by EDF is also feasible:



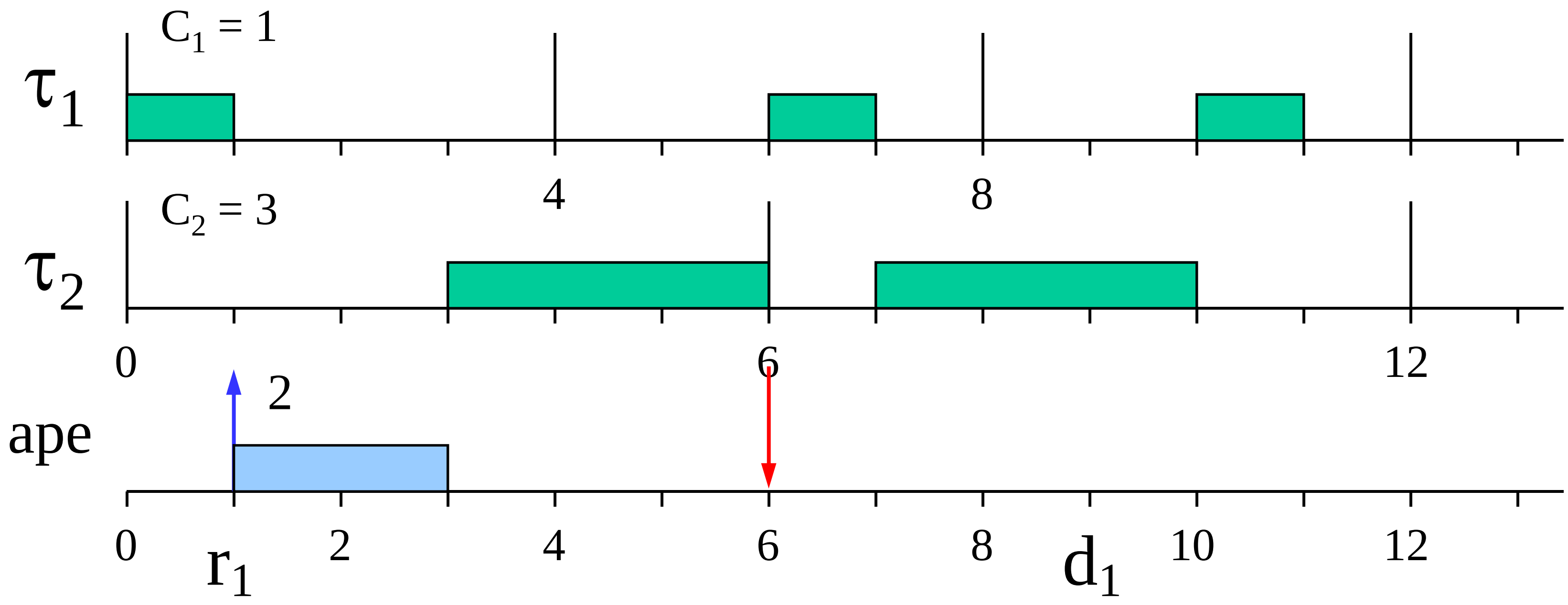
Improving TBS

- We can now apply the same argument, and advance the deadline to t = 6:



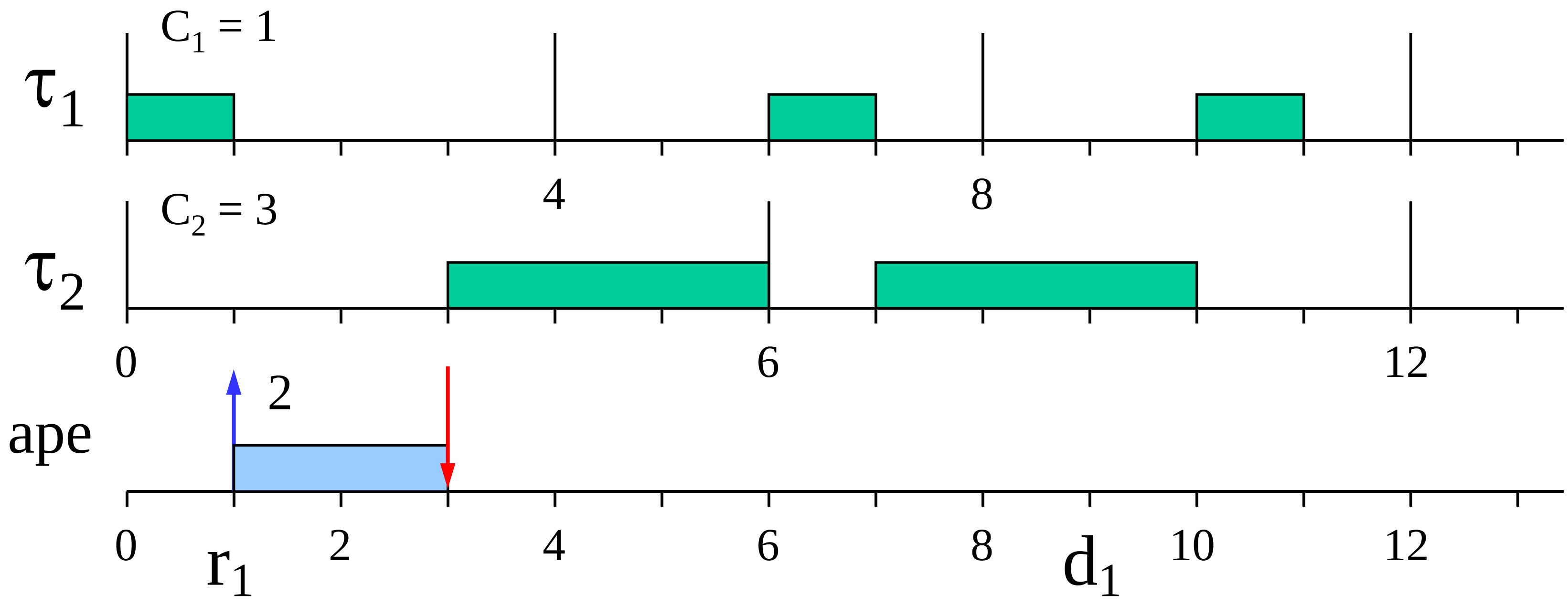
Improving TBS

- We can now apply the same argument, and advance the deadline to t = 6:



Improving TBS

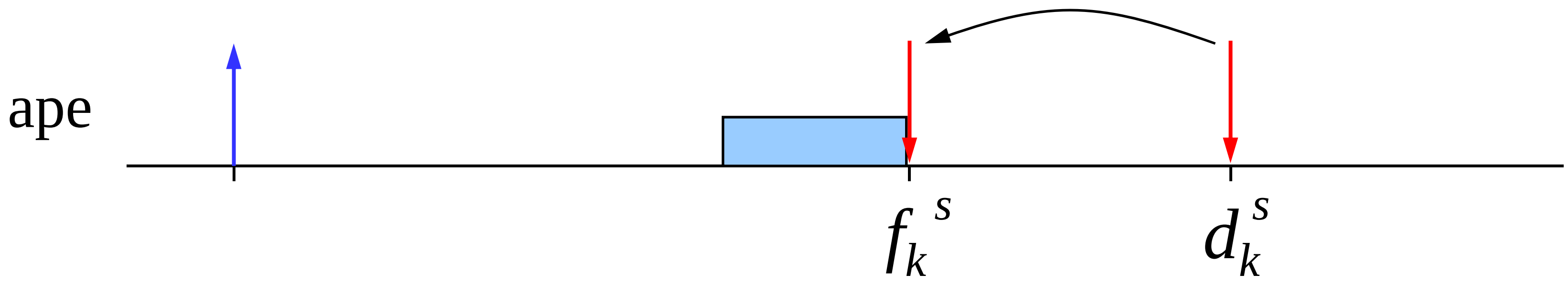
- Clearly, advancing the deadline now does not produce any enhancement:



Computing the deadline

- In general, the new deadline has to be set to the finishing time of the current job:

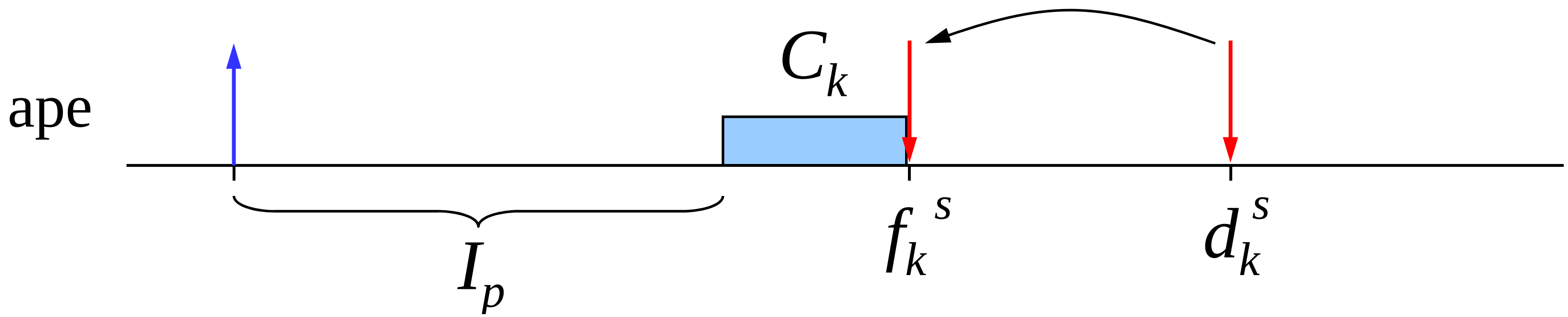
$$\begin{cases} d_k^0 = \max(r_k, d_{k-1}^0) \\ d_k^{s+1} = f_k^s = f_k(d_k^s) \end{cases}$$



Computing the deadline

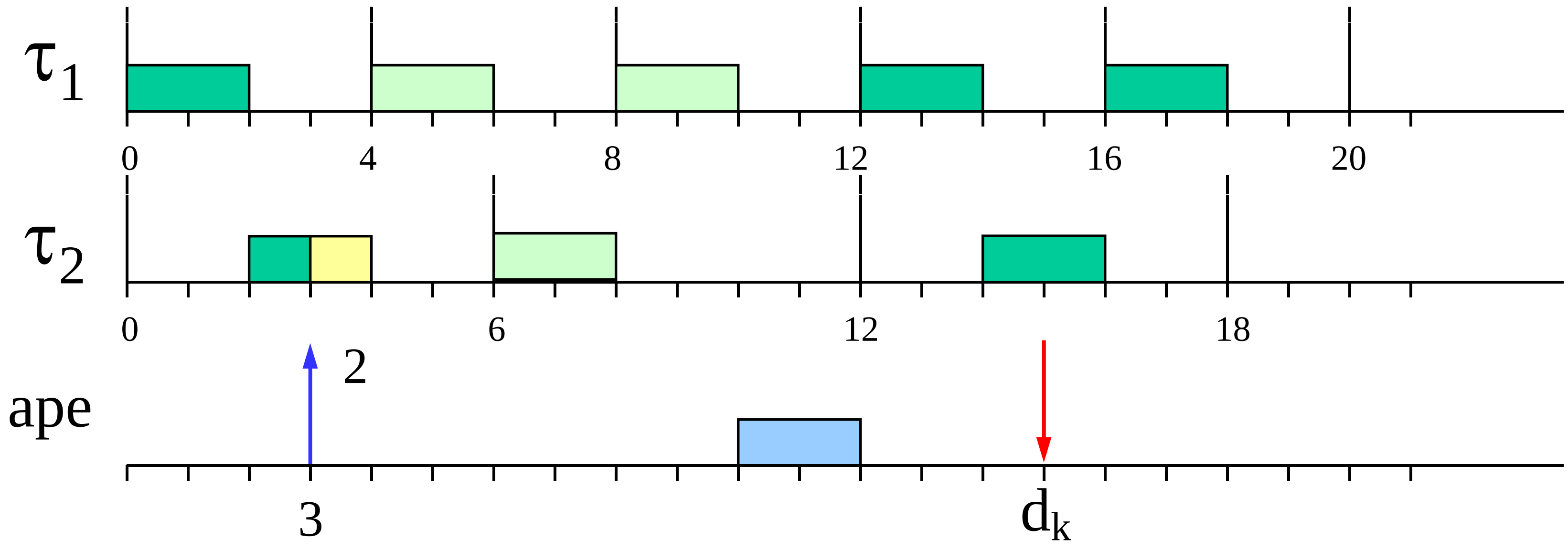
- The actual finishing time can be estimated based on the periodic interference:

$$f_k^s = C_k + I_p(r_k, d_k^s)$$



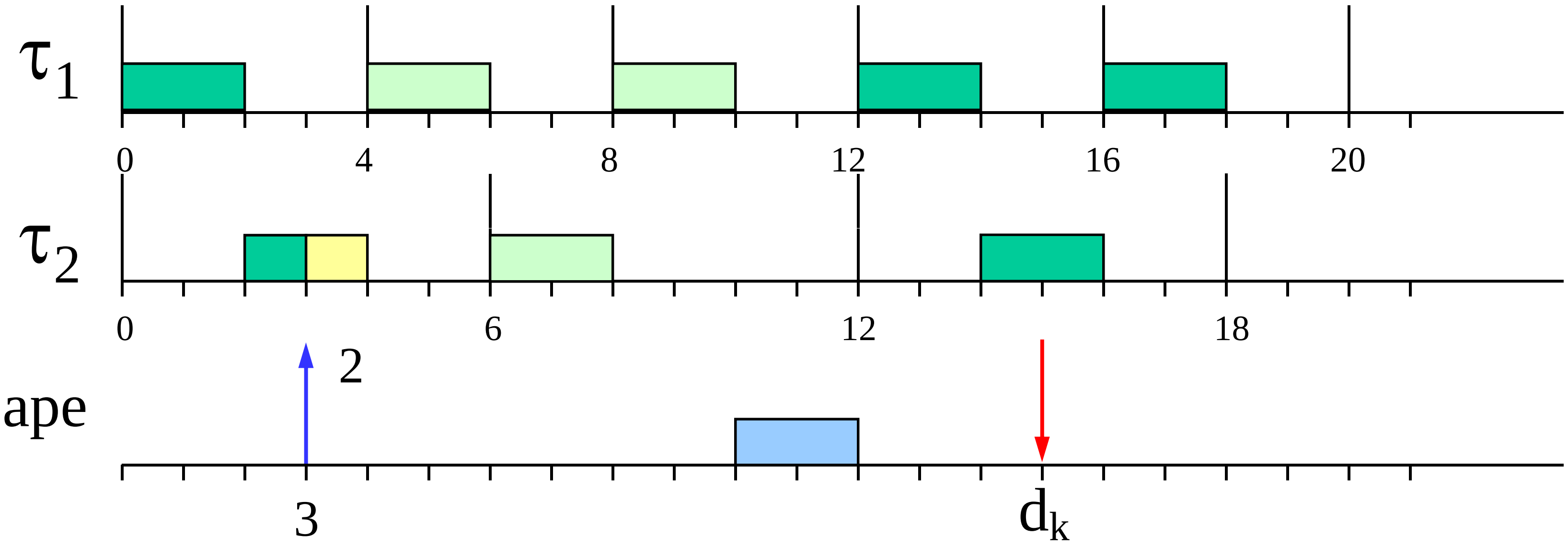
Periodic Interference

$$\begin{cases} U_p = 1/2 + 1/3 = 5/6 \\ U_s = 1 - U_p = 1/6 \end{cases} \qquad \begin{cases} C_k = 2 \\ d_k = 3 + 2/U_s = 15 \end{cases}$$



$$I_p(t, d_k^s) = I_a(t, d_k^s) + I_f(t, d_k^s)$$

Computing interference

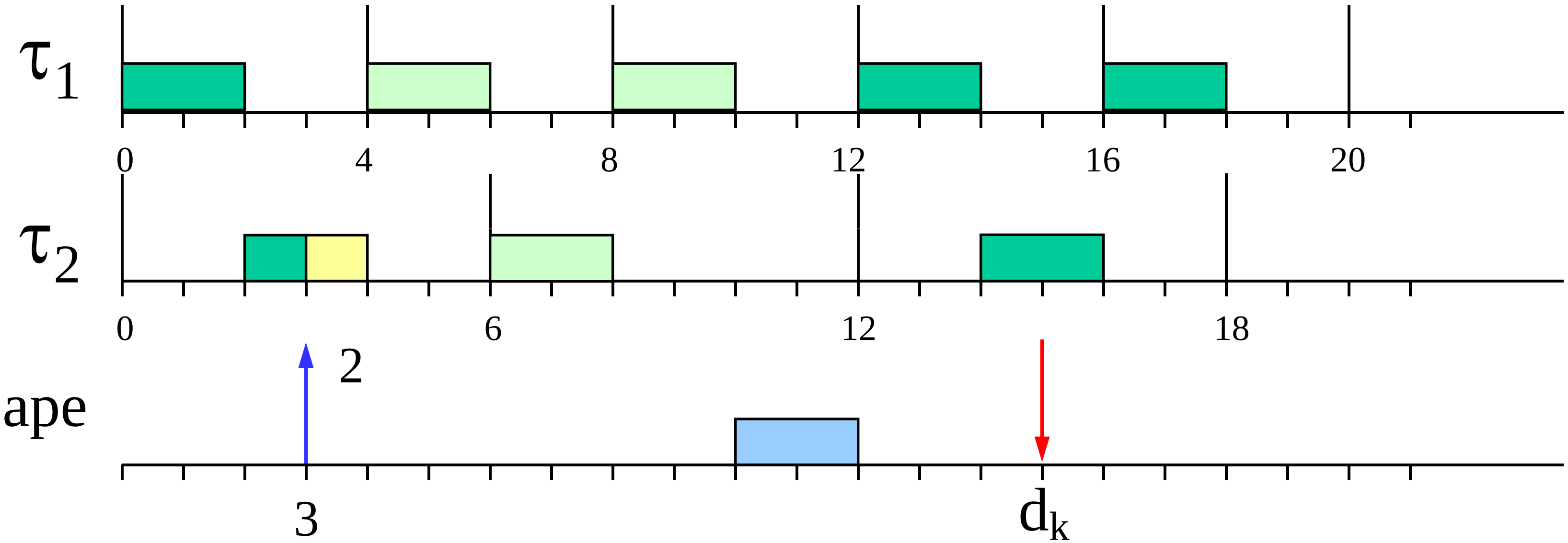


$$I_a(t, d_k^s) = \sum_{\tau_i \ active} c_i(t)$$

next$_i$(t) = next release time of task τ_i after t

$$I_f(t, d_k^s) = \sum_{i=1}^{n} \left\lfloor \frac{d_k^s - next_i(t)}{T_i} \right\rfloor C_i$$

Computing interference

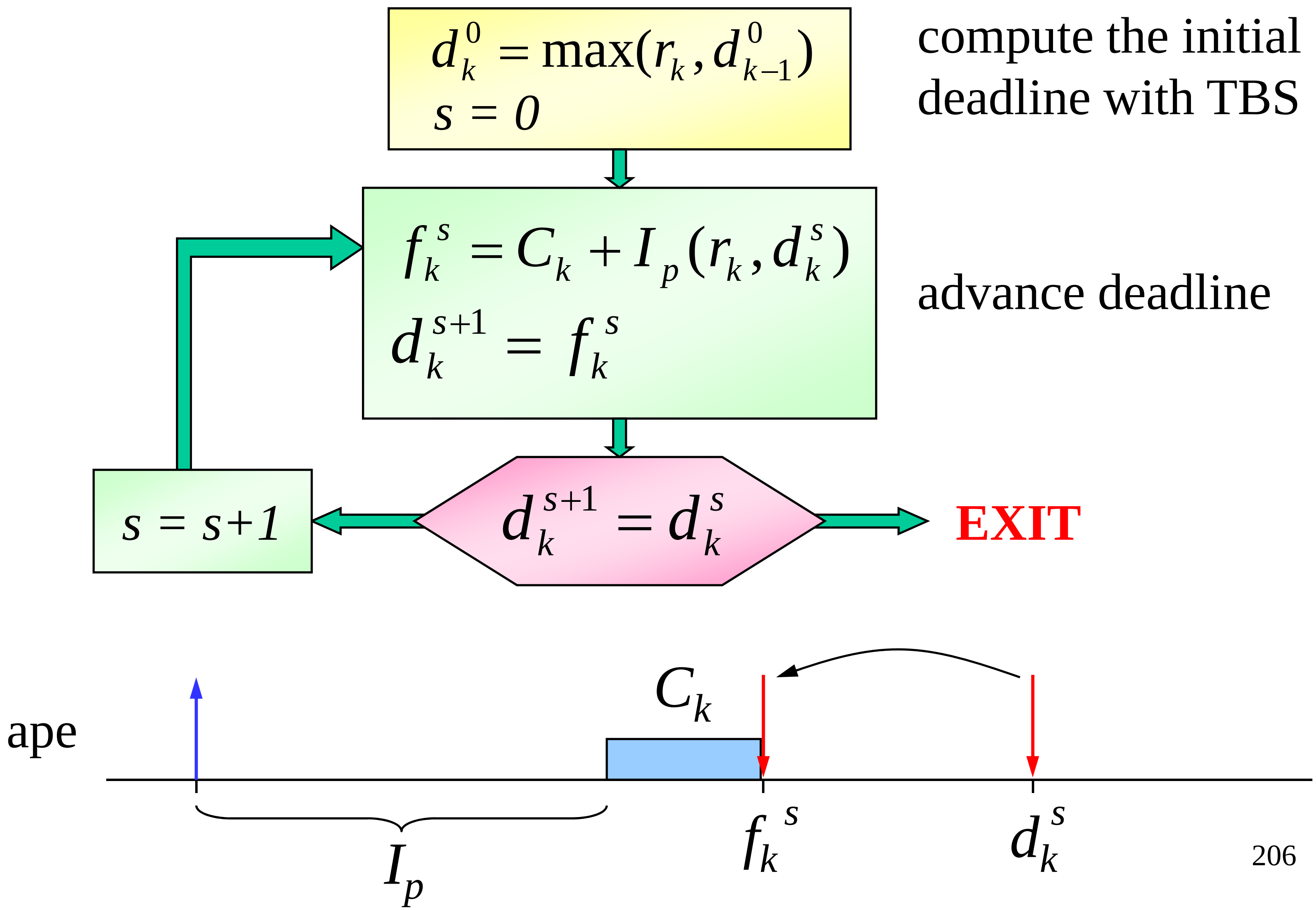


$$I_a(t, d_k^s) = \sum_{\tau_i \ active} c_i(t)$$

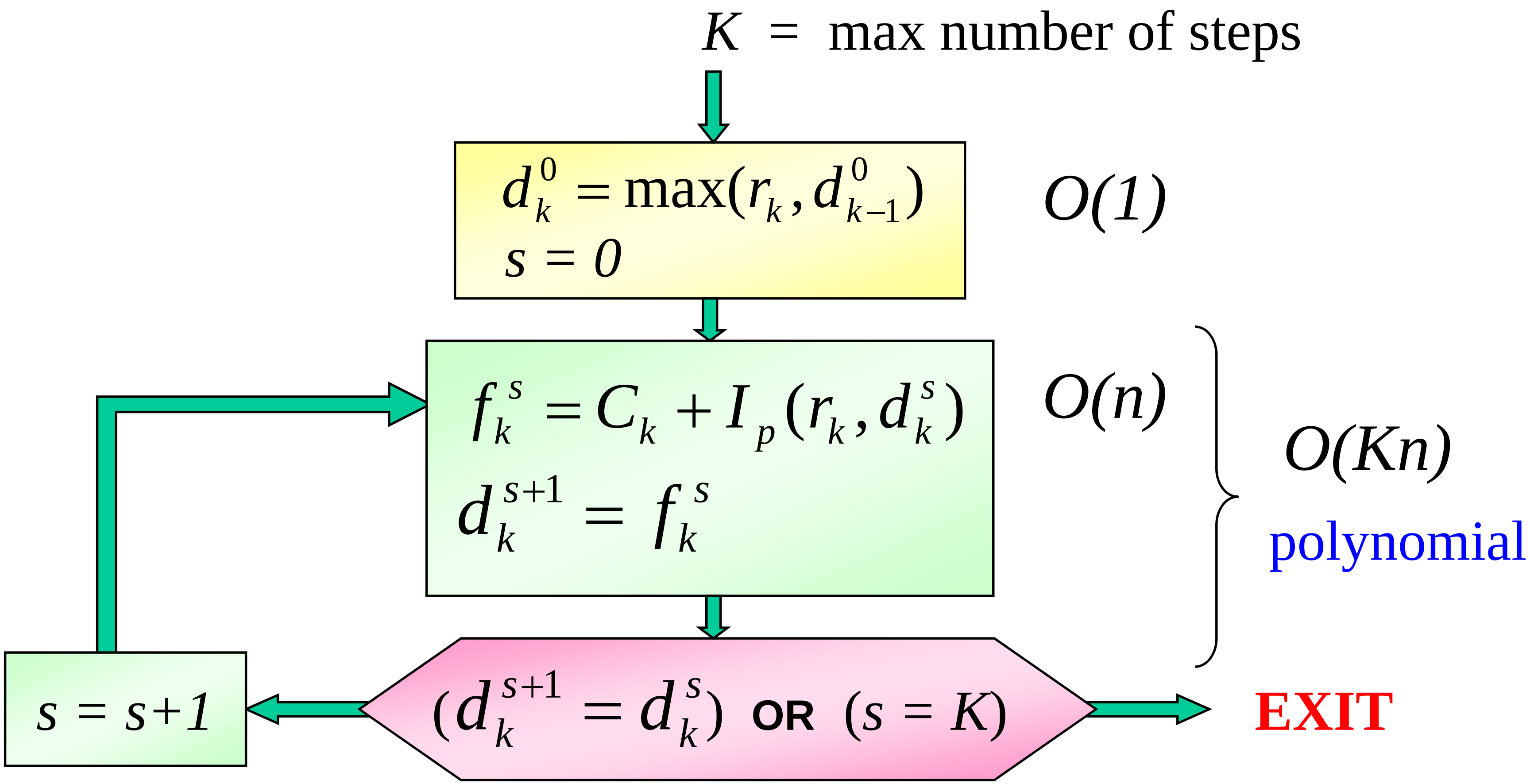
$next_i(t)$ = next release time of task τ_i after t

$$I_f(t, d_k^s) = \sum_{i=1}^{n} \left(\left\lceil \frac{d_k^s - next_i(t)}{T_i} \right\rceil - 1 \right) C_i$$

The Optimal Server

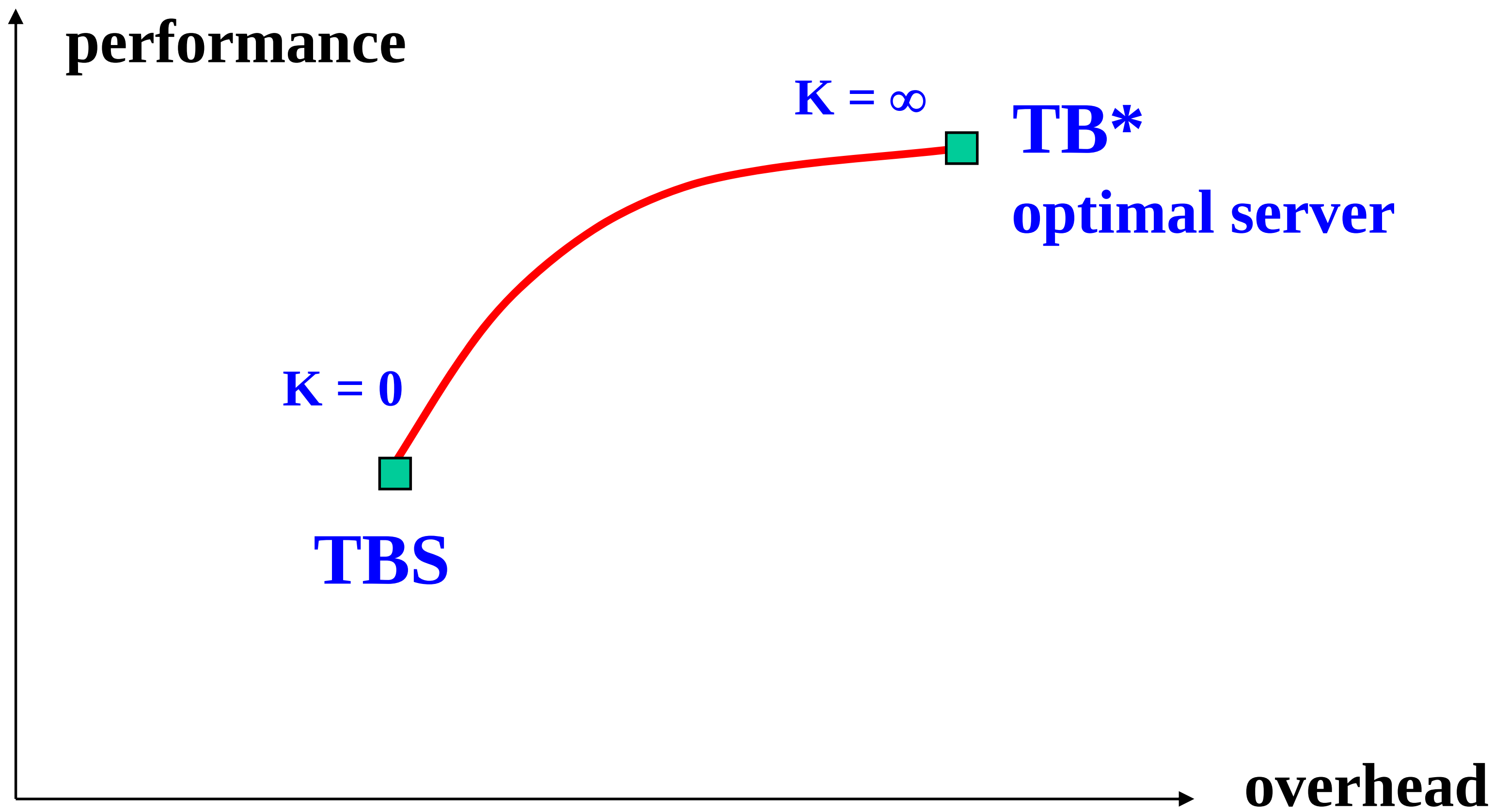


Tunable Bandwidth Server TB(K)

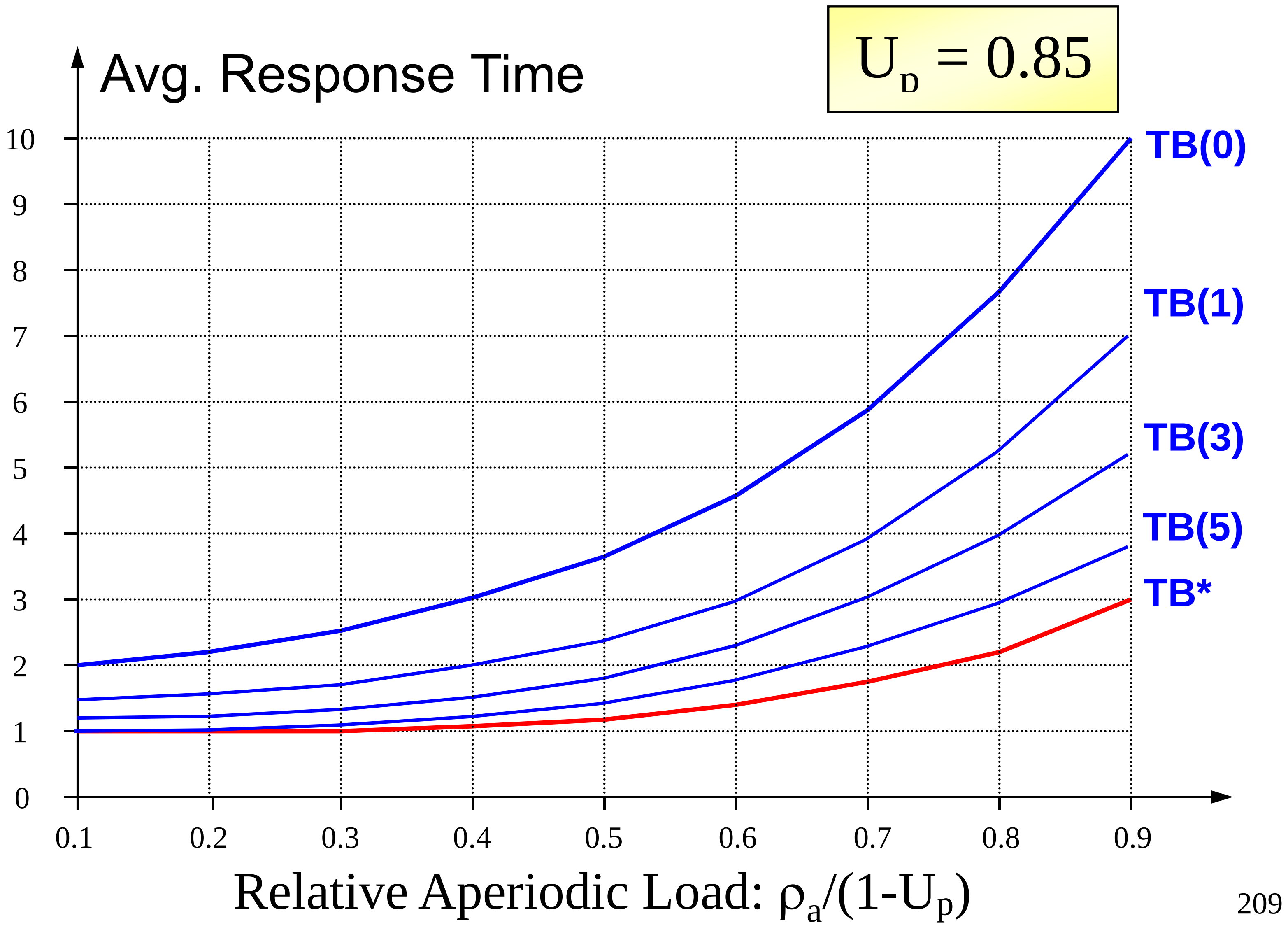


TB(0) = TBS **TB(∞) = TB***

Tuning performance vs. overhead

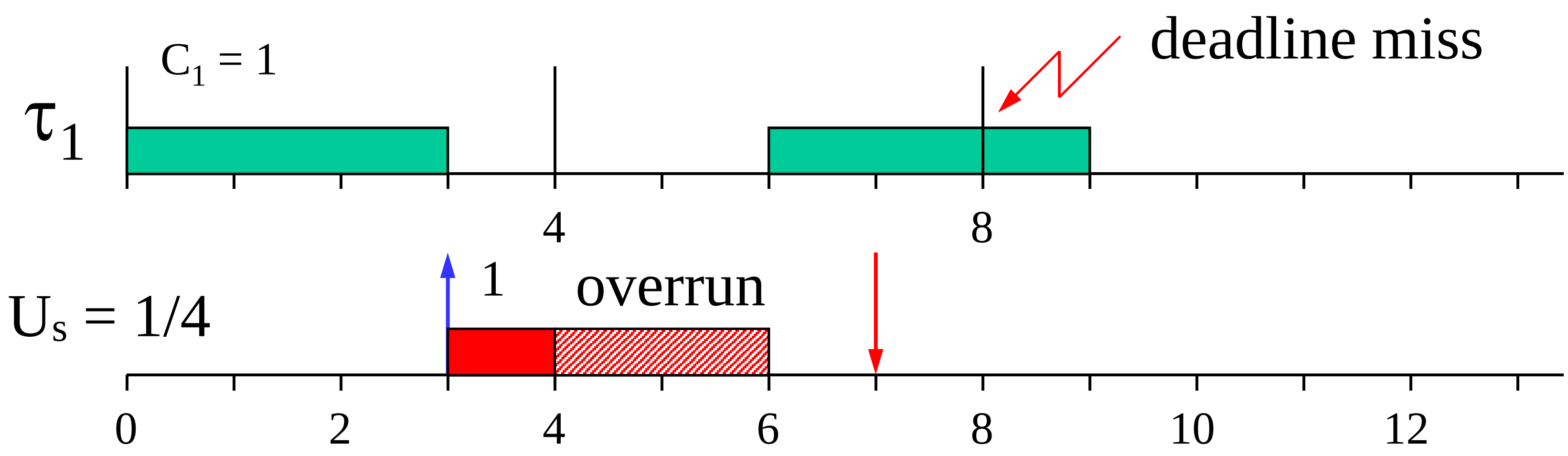


Aperiodic responsiveness



Problems with the TBS

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



Solution: temporal isolation

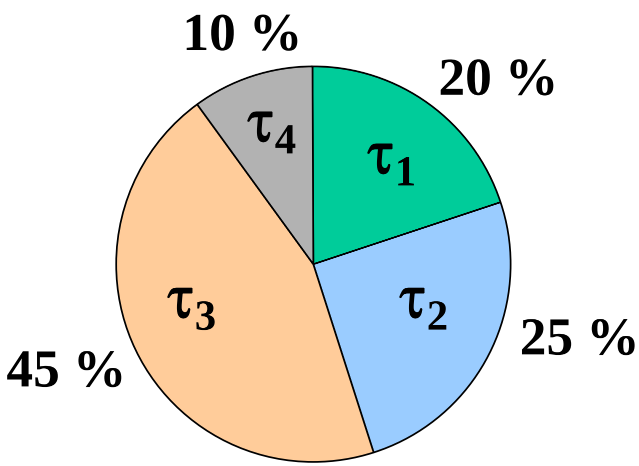
- In the presence of overruns, only the faulty task should be delayed.
- Each task τ_i should not demand more than its declared utilization ($U_i = C_i/T_i$).
- If a task executes more than expected, its priority should be decreased (i.e., its deadline postponed).

Achieving isolation

- Isolation among tasks can be achieved through a **bandwidth reservation**.
- Each task is managed by a dedicated server having bandwidth U_s .
- The server assigns priorities (or deadlines) to tasks so that they do not exceed the reserved bandwidth.

Resource Reservation

Resource partition



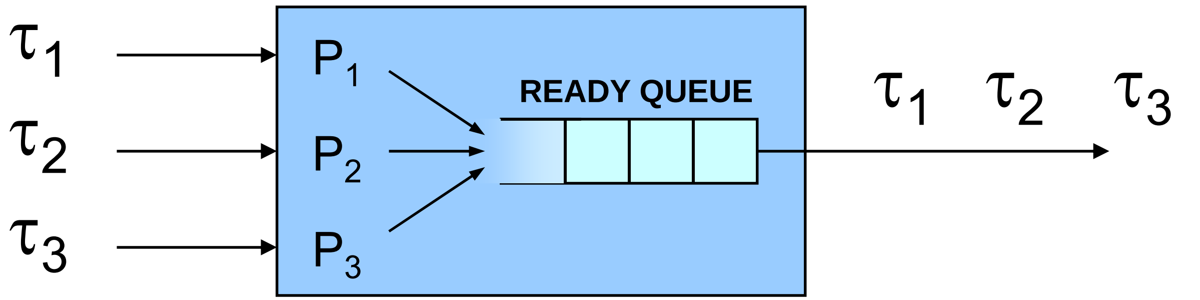
Each task receives a bandwidth U_i and behaves as it were executing alone on a slower processor of speed U_i

Resource enforcement

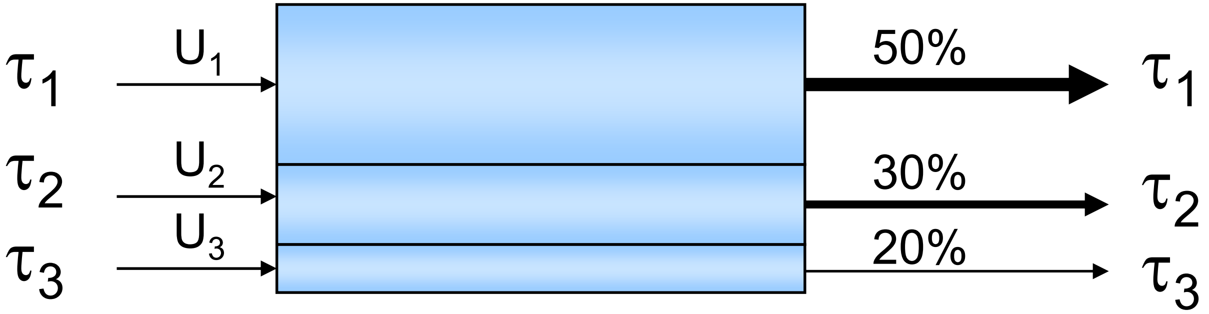
- A mechanism that prevents a task to consume more than its reserved amount.
- If a task executes more, it is delayed, preserving the resource for the other tasks.

Priorities vs. Reservations

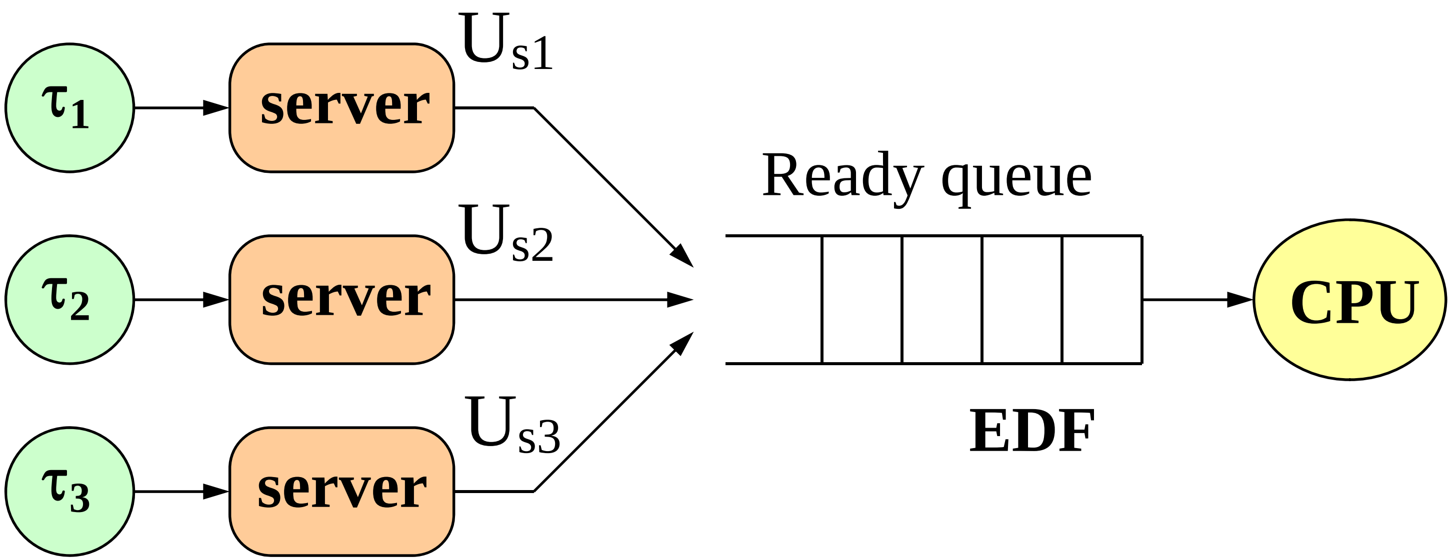
Prioritized Access



Resource Reservation



Implementation



$$U_{s1} + U_{s2} + U_{s3} \leq 1$$

Constant Bandwidth Server (CBS)

- It assigns deadlines to tasks like the TBS, but keeps track of job executions through a budget mechanism.
- When the budget is exhausted it is immediately replenished, but the deadline is postponed to keep the demand constant.

CBS parameters

Given by the user

- Maximum budget: Q_s
- Server period: T_s

$U_s = Q_s / T_s$ (server bandwidth)

Maintained by the server

- Current budget: c_s (initialized to 0)
- Server deadline: d_s (initialized to 0)

Basic CBS rules

- **Arrival of job J_k $\Rightarrow$ assign d_s**

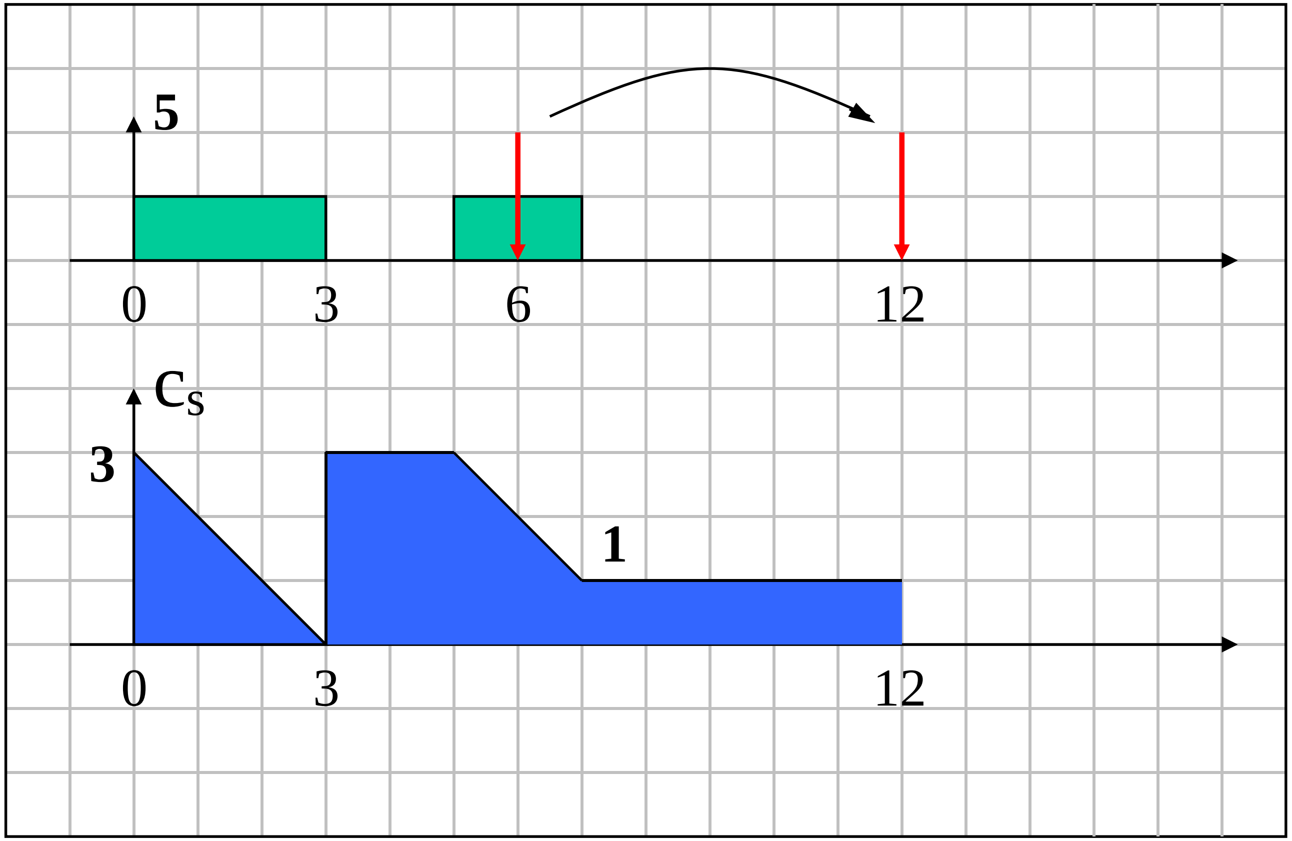
$$\textbf{if}\ (r_k + c_s/U_s \le d_s)\ \textbf{then}\ \text{recycle}\ (c_s, d_s)$$

$$\textbf{else}\quad \begin{cases} d_s = r_k + T_s \\ c_s = Q_s \end{cases}$$

- **Budget exhausted $\Rightarrow$ postpone d_s**

$$\begin{cases} d_s = d_s + T_s \\ c_s = Q_s \end{cases}$$

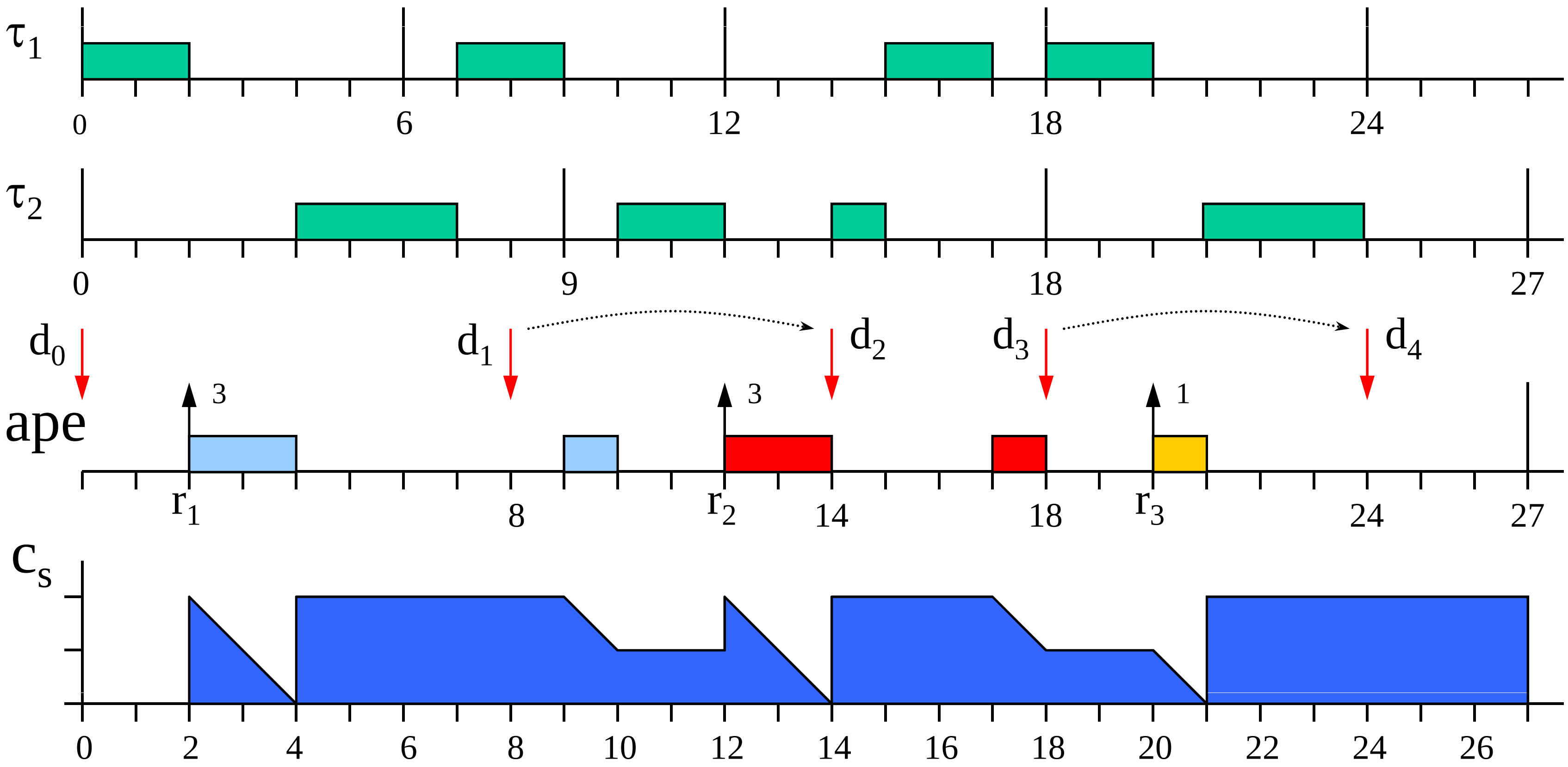
Budget exhausted



$Q_s = 3$

$T_s = 6$

EDF + CBS schedule



CBS: $Q_s = 2$, $T_s = 6$

CBS properties

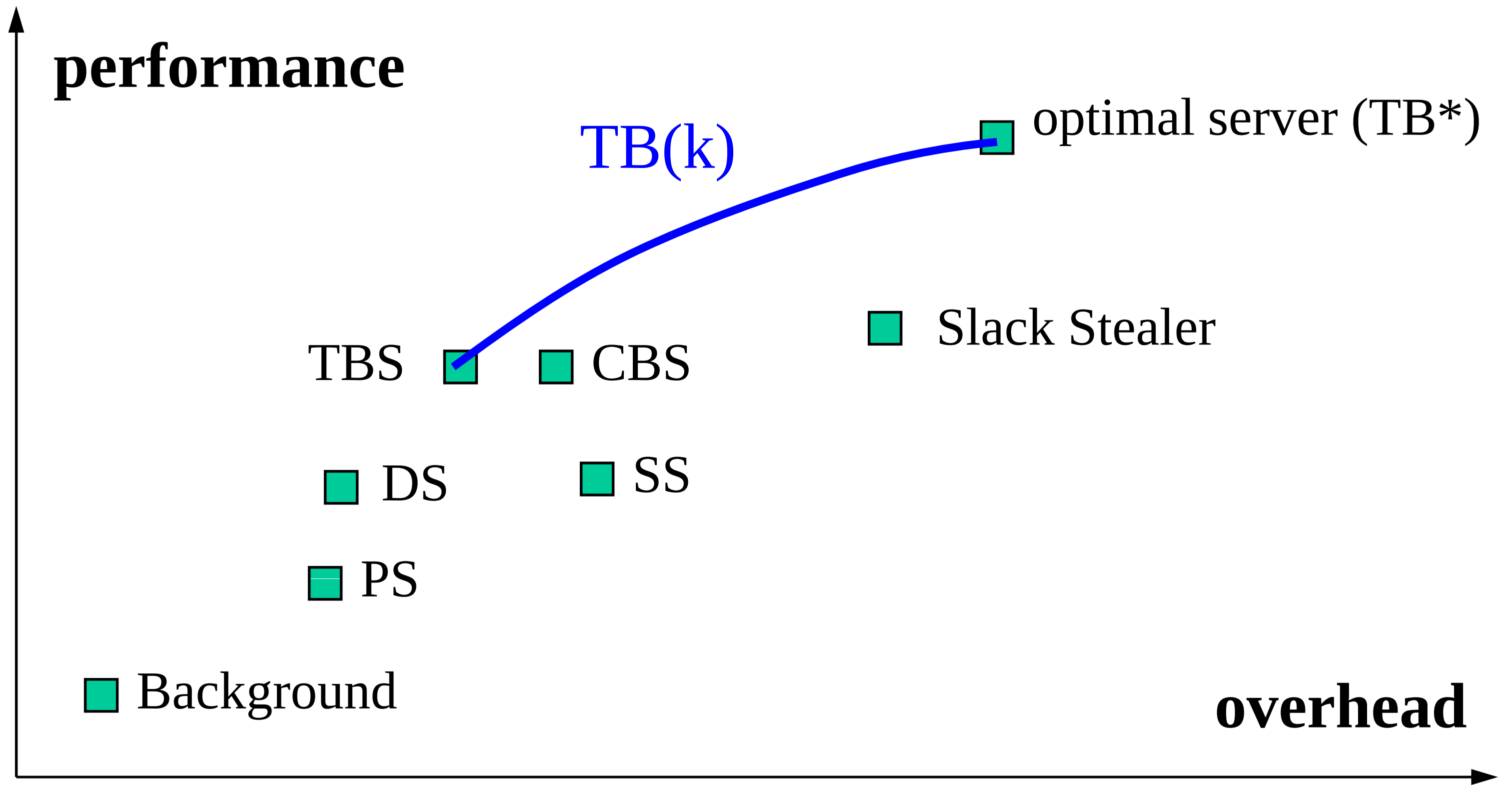
- **Bandwidth Isolation**

  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 $T_s = T_i$, iff τ_i is schedulable by EDF.

Selecting the most suitable service mechanism



It depends on the price (overhead) we want to pay to reduce task response times