artist

Network of Excellence on
Embedded Systems Design

scuola superiore Sant'Anna di studi universitari e di perfezionamento

Welcome to the Course on
Real-Time Kernels for Microcontrollers:
Theory and Practice

Retis
Real-Time Systems Laboratory

Scuola Superiore Sant'Anna, Pisa

June 23-25, 2008

Course Program

Day 1: **Monday, June 23**

Morning:	RT scheduling & resource management	(G. Buttazzo)
	RT kernels for embedded systems	(P. Gai)
Afternoon:	The FLEX development board	(M. Marinoni)
	Erika kernel and the OSEK standard	(P. Gai)

Day 2: **Tuesday, June 24**

Morning:	Developing RT appl. with Erika	(P. Gai – M. Marinoni)
Afternoon:	Laboratory practice	(P. Gai – M. Marinoni)

Day 3: **Wednesday, June 25**

Morning:	Embedded Systems and Wireless Communication	(P. Pagano – G. Franchino)
Afternoon:	Laboratory practice	(P. Pagano – G. Franchino)

Lecture Schedule

09:00	Morning Lecture - Part 1
11:00	Coffee Break
11:15	Morning Lecture - Part 2
13:00	Lunch Break
14:30	Afternoon Lecture - Part 1
16:15	Coffee Break
16:30	Afternoon Lecture - Part 2
18:00	End of Lectures

Real-Time Scheduling and Resource Management

Giorgio Buttazzo

E-mail: buttazzo@sssup.it

Real-Time Systems Laboratory

Scuola Superiore Sant'Anna

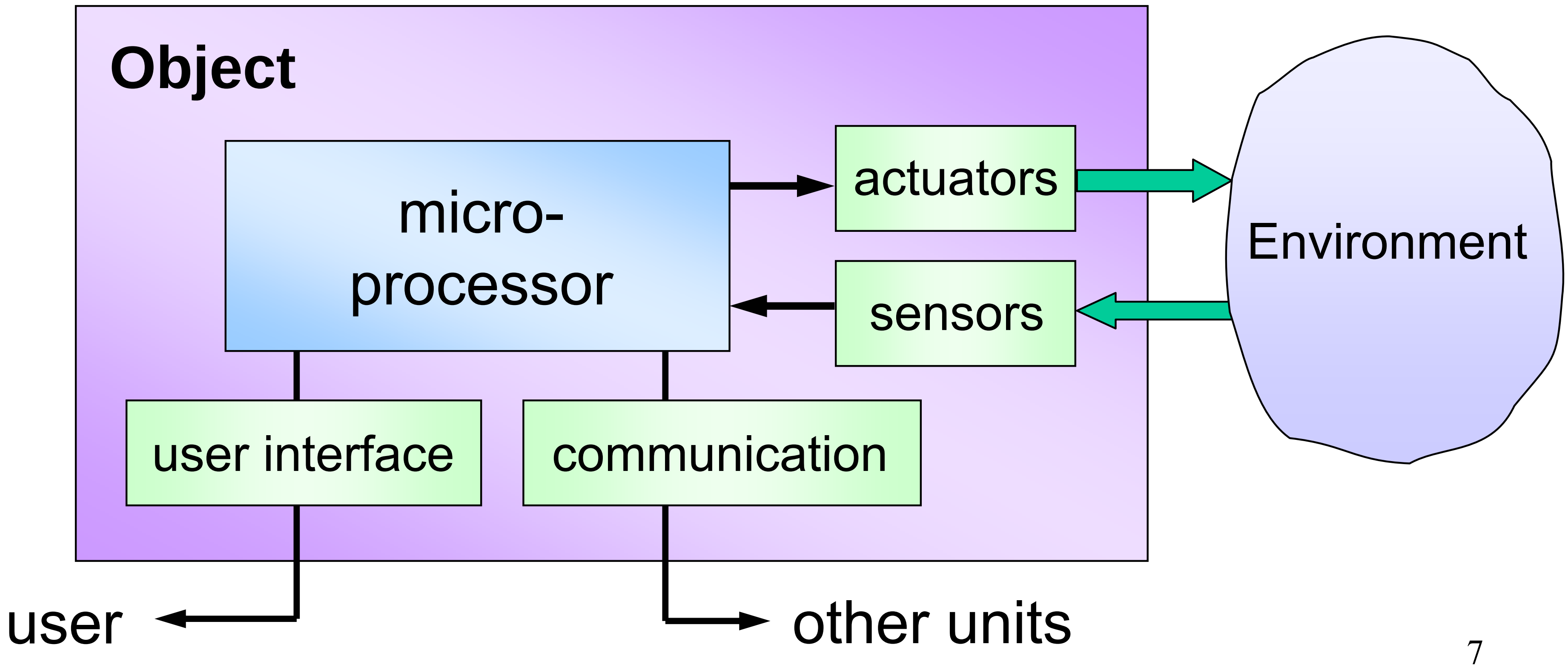
Goal

Provide some background of RT theory for implementing control applications:

- Background and basic concepts
- Modeling real-time activities
- Real-Time Task Scheduling
- Timing analysis
- Handling shared resources

What’s an Embedded System?

⇒ It is a computing system hidden in an object to control its functions, enhance its performance, manage the available resources and simplify the interaction with the user.



What’s special in Embedded Systems?

- Stringent constraints on space, weight, energy, cost
  - ⇒ Scarce resources (processing power, memory)
  - ⇒ Efficient resource usage at the OS level
- Interaction with the environment
  - ⇒ High responsiveness and timing constraints
  - ⇒ Schedulability analysis and predictable behavior (RTOS)
- Robustness (tolerance to parameter variations)
  - ⇒ Overload management and system adaptation, to cope with variable resource needs and high load variations.

… and many others

- mobile robot systems
- small embedded devices
  - ⇒ cell phones
  - ⇒ videogames
  - ⇒ smart sensors
  - ⇒ intelligent toys





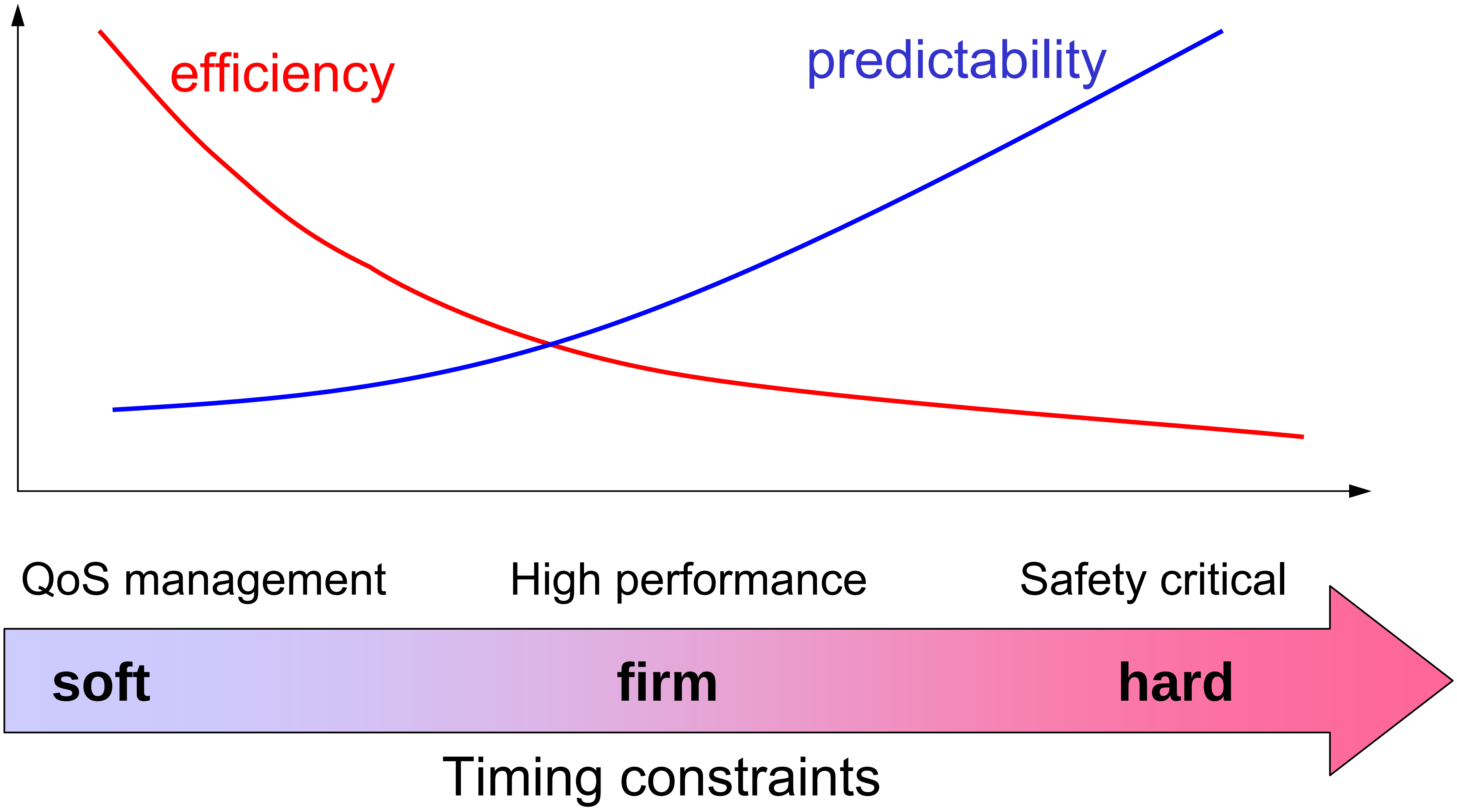

Criticality

QoS management | High performance | Safety critical

soft | **firm** | **hard**

Timing constraints

System Requirements

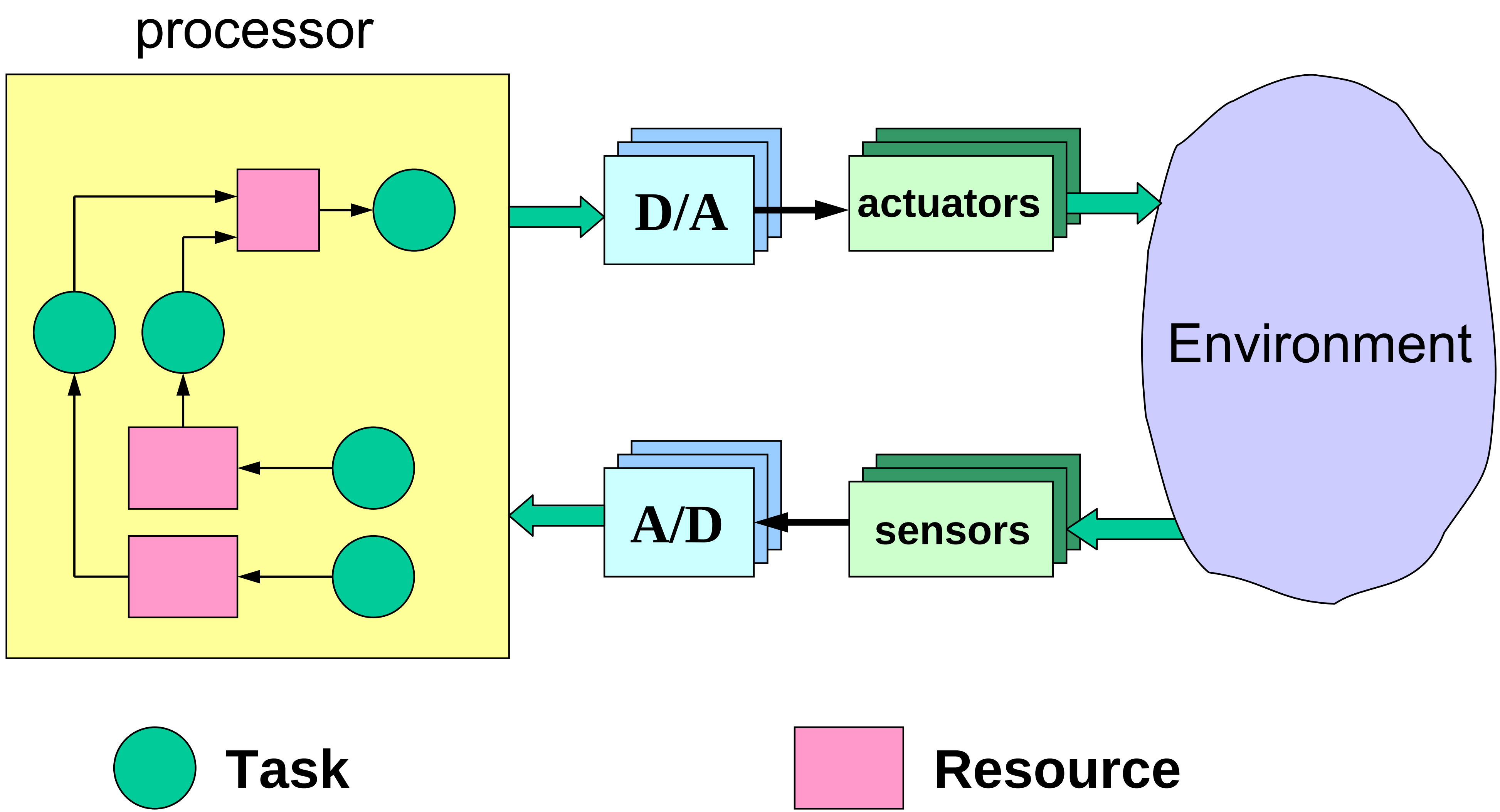


Importance of the RTOS

The Operating System is responsible for:

- managing the available resources in an efficient way (memory, devices, energy);
- Enforcing timing constraints on computational activities;
- Providing a standard programming interface to develop portable applications.
- Providing suitable monitoring mechanisms to trace the system evolution to support debugging.

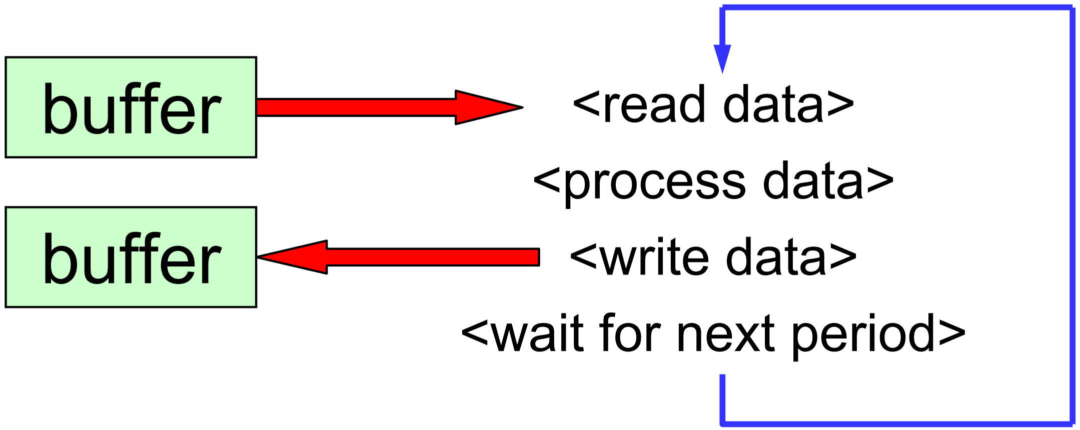
Software Vision



Activation modes

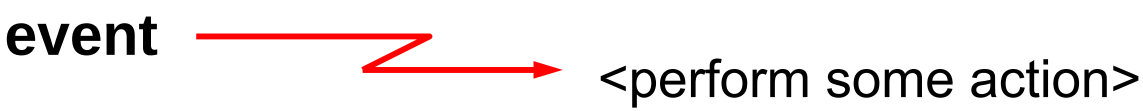
Periodic tasks: **(time driven)**

tasks are automatically activated by the kernel at regular time intervals:



Aperiodic tasks: **(event driven)**

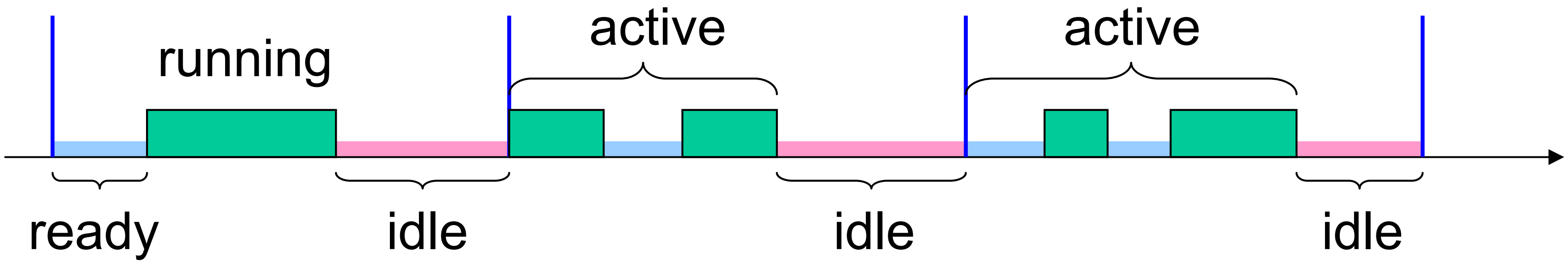
tasks are activated upon the arrival of an event (interrupt or explicit activation)



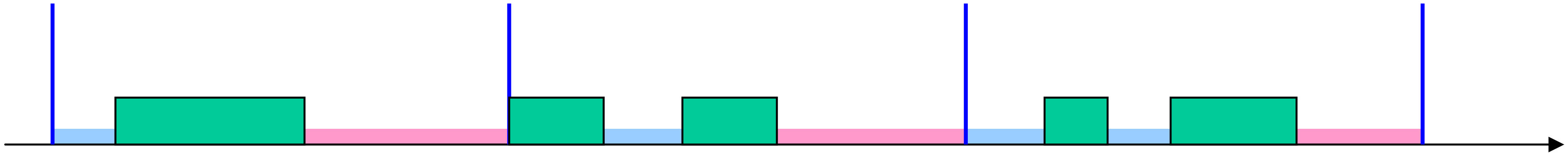
OS support for periodic tasks

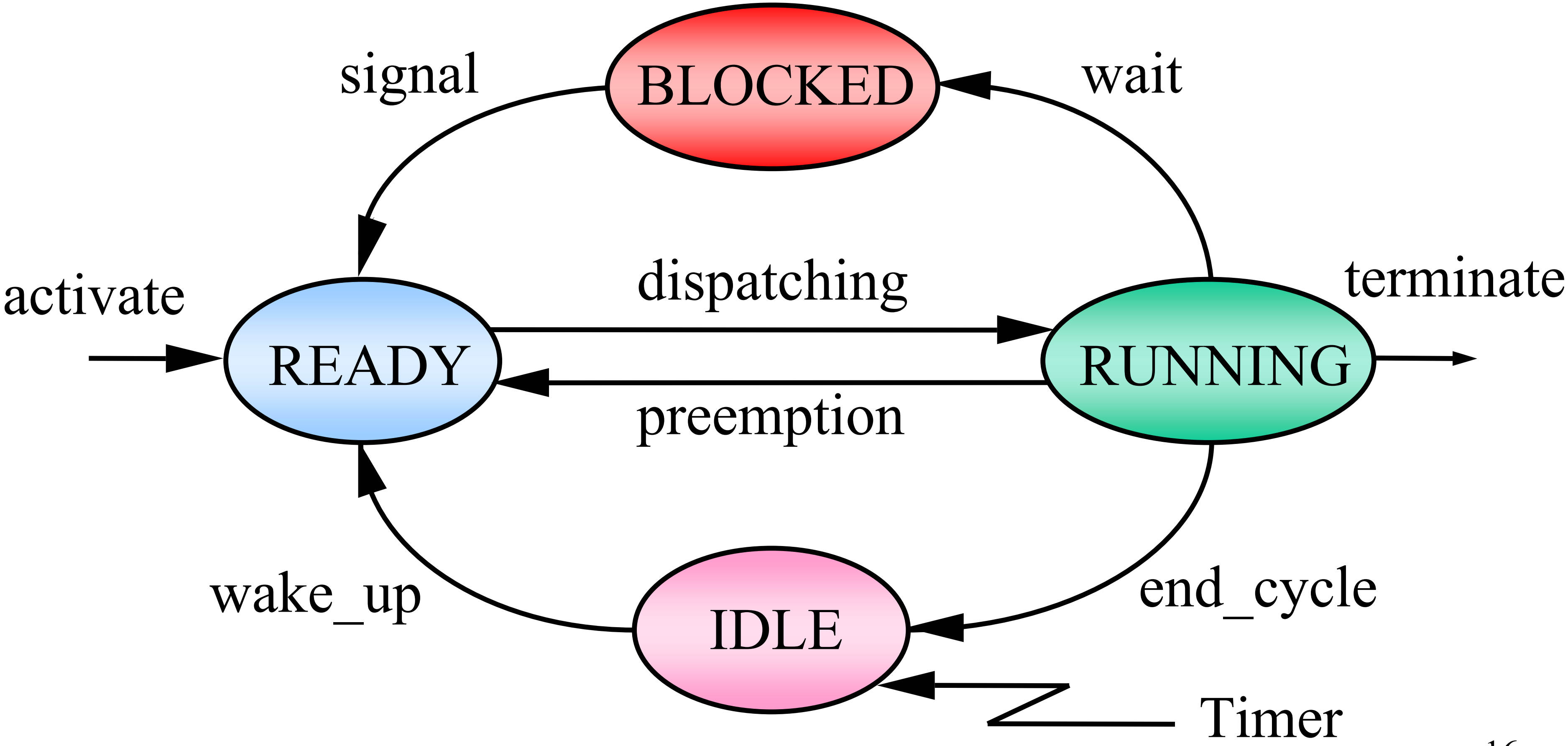
task τ_i

```
while (condition) {
        ______
        ______
        ______
        ______
        ______
        ______
        ______
        ______
        wait_for_next_period();
}
```

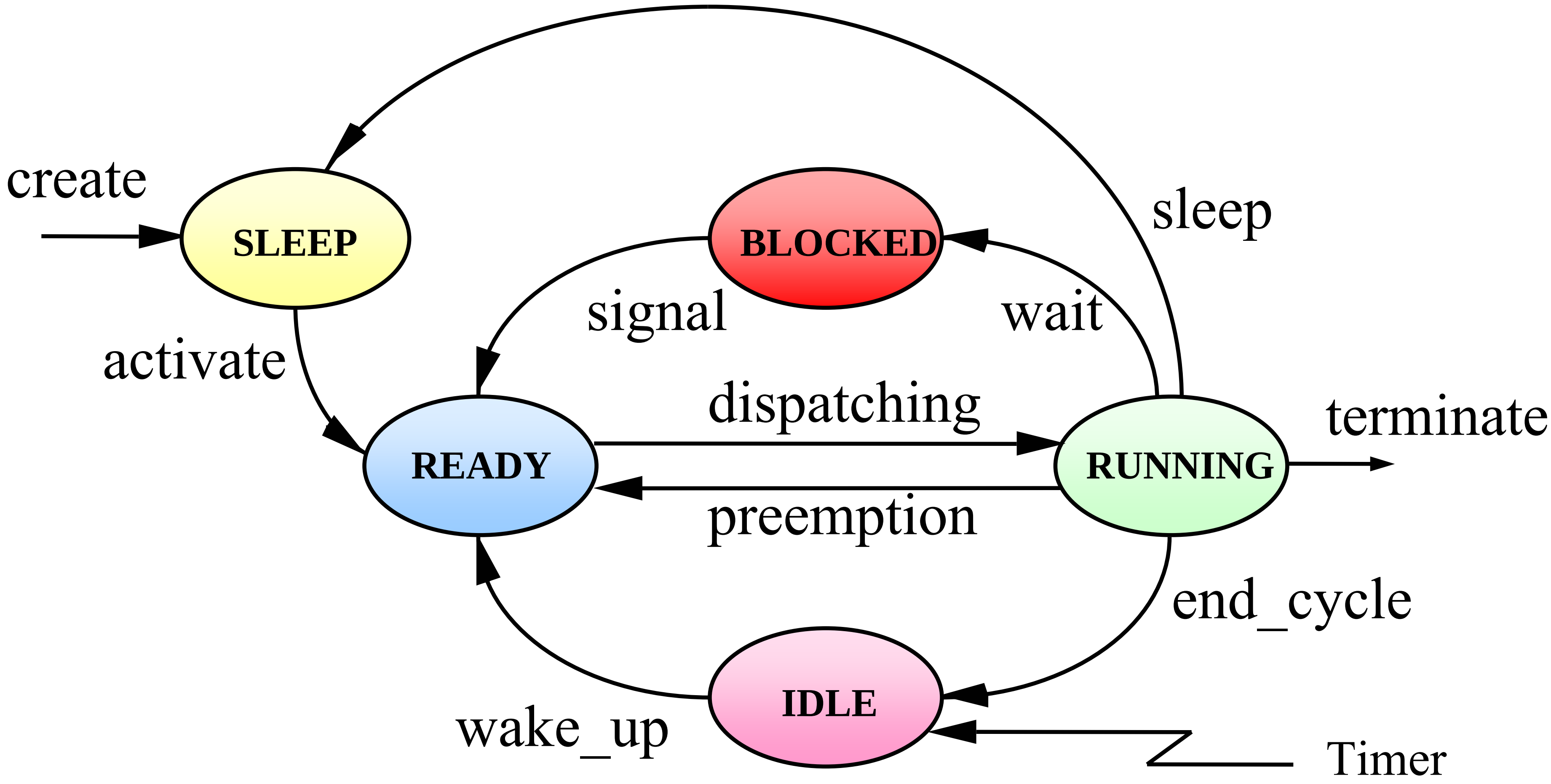


The IDLE state



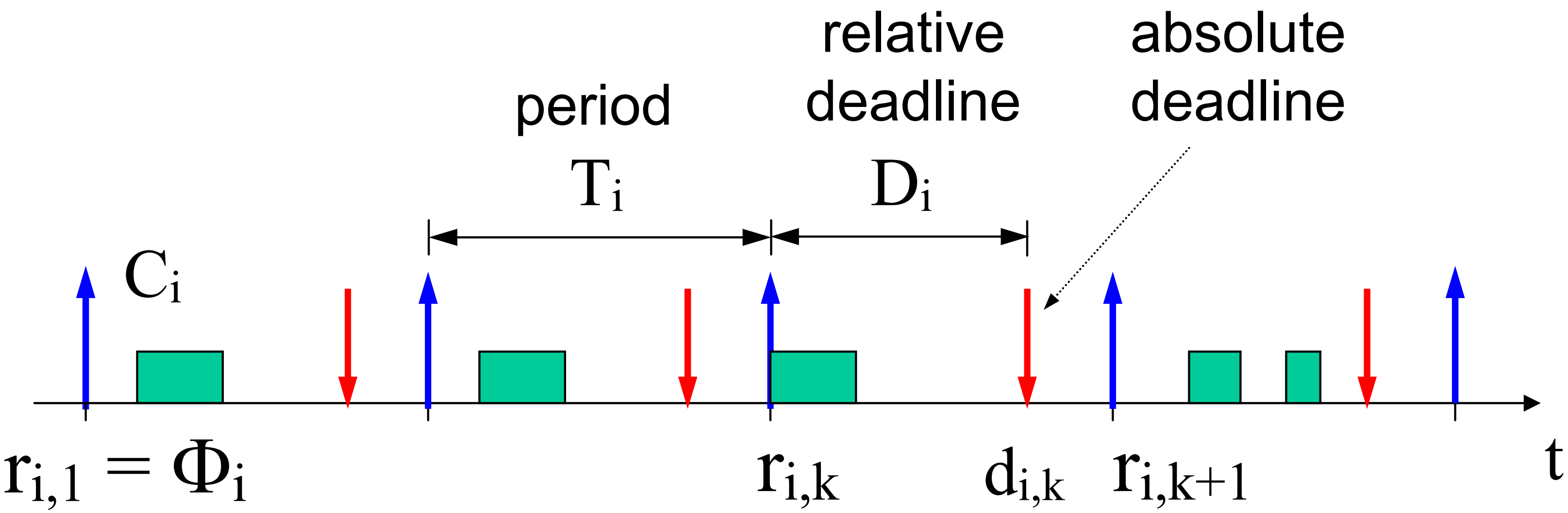


SLEEP state


create
SLEEP
BLOCKED
sleep
signal
wait
activate
dispatching
terminate
READY
RUNNING
preemption
end_cycle
IDLE
wake_up
Timer

Periodic task model

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



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

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

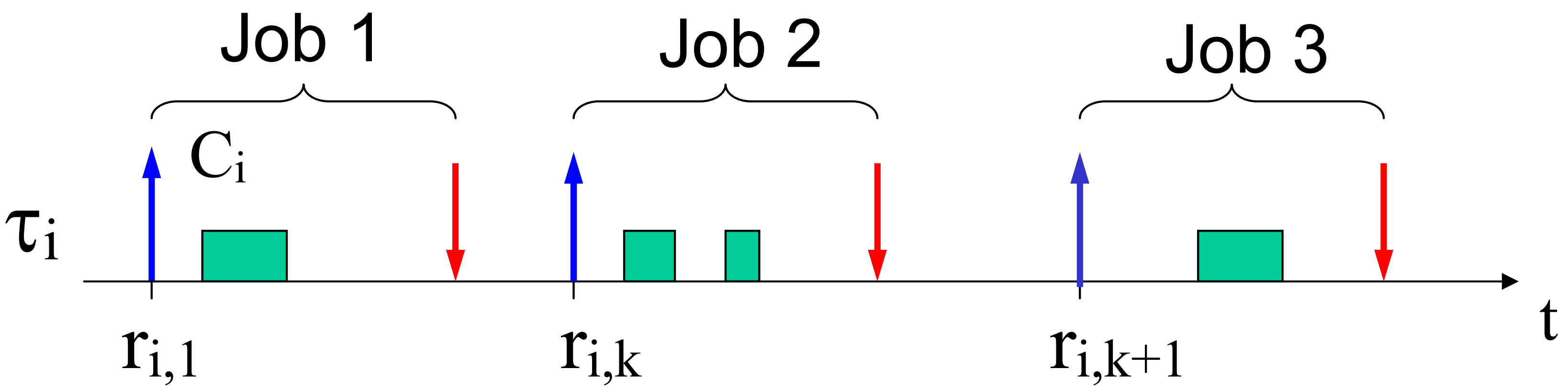
often

$\Phi_i = 0$

$D_i = T_i$

Aperiodic task model

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



Periodic Task Scheduling

- We have ***n*** periodic tasks: $\{\tau_1 , \tau_2 \ldots \tau_n\}$

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

Goal

- Execute all tasks within their deadlines
- Verify feasibility before runtime

Assumptions

- Tasks are execute in a single processor
- Tasks are independent (do not block or self-suspend)
- Tasks are synchronous (all start at the same time)
- Relative deadlines are equal to periods ($D_i = T_i$)

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

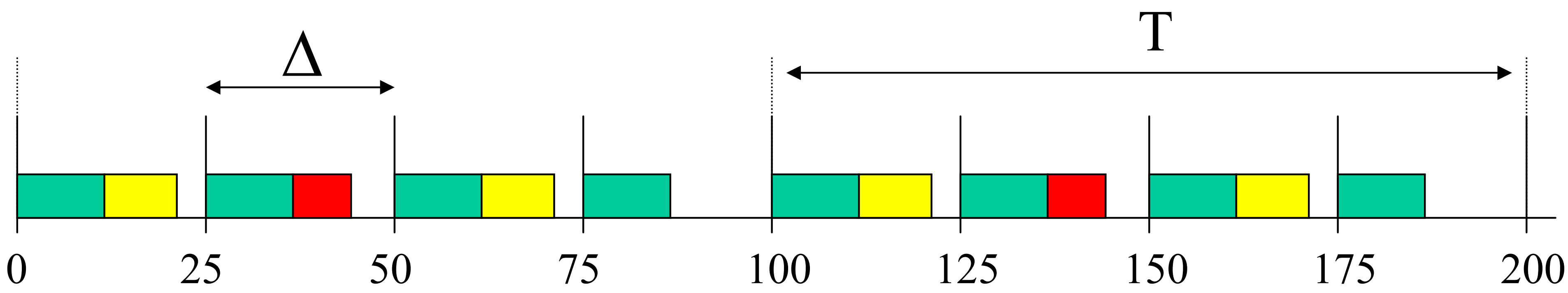
- The time axis is divided in intervals of equal length (***time slots***).
- Each task is statically allocated in a slot in order to meet the desired request rate.
- The execution in each slot is activated by a timer.

Example

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

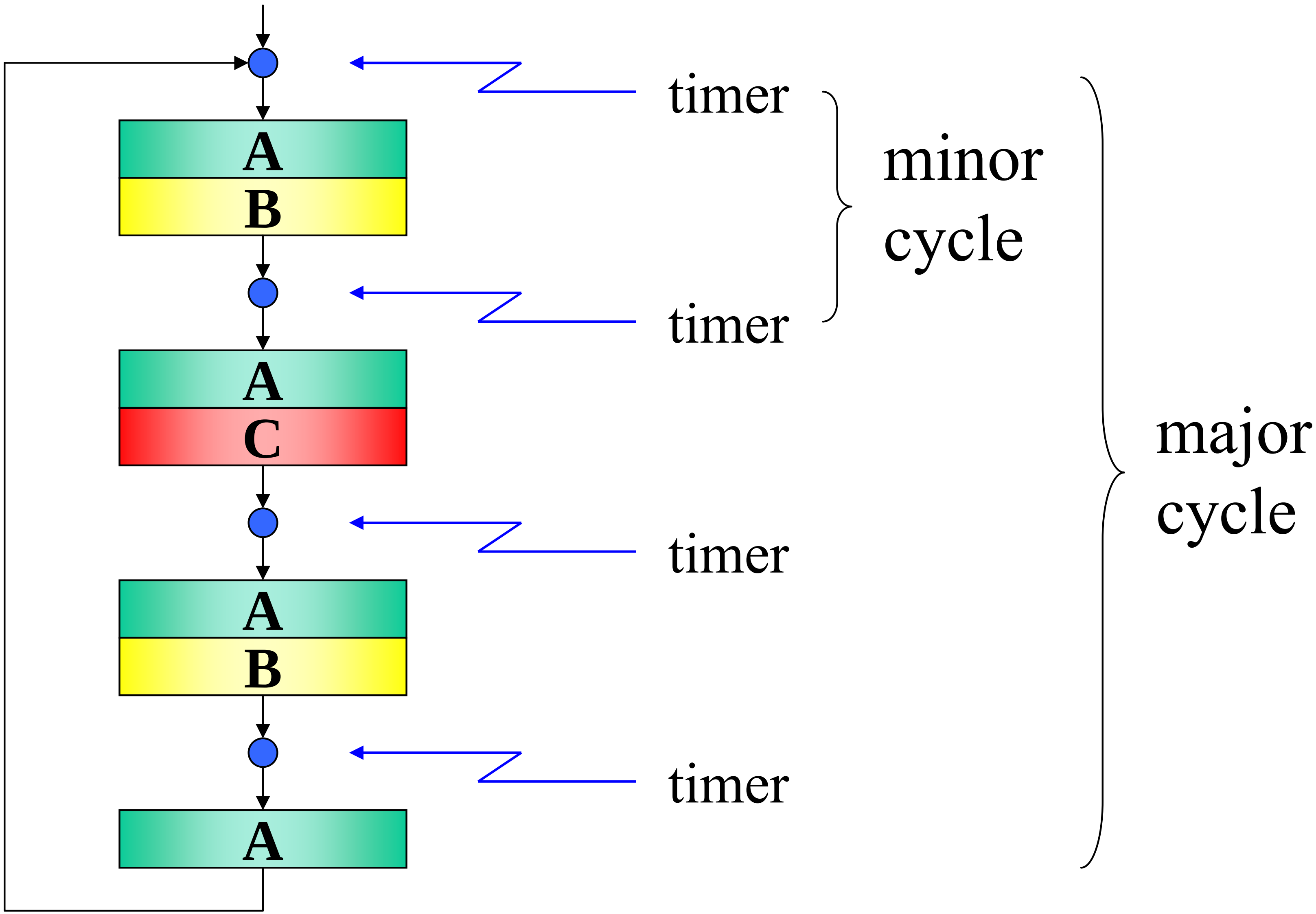
Δ = GCD (minor cycle)

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

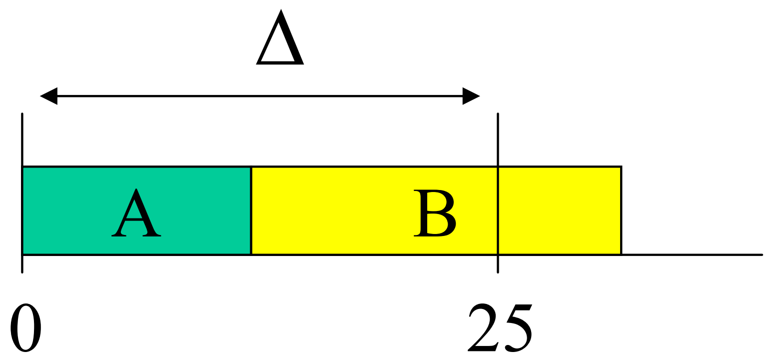
What do we do during task overruns?

- 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

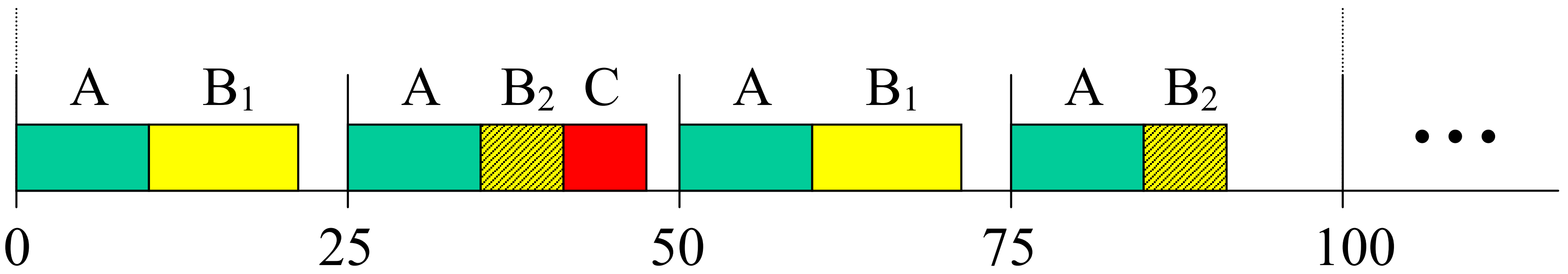
If one or more tasks need to be upgraded, we may have to re-design the whole schedule again.

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:

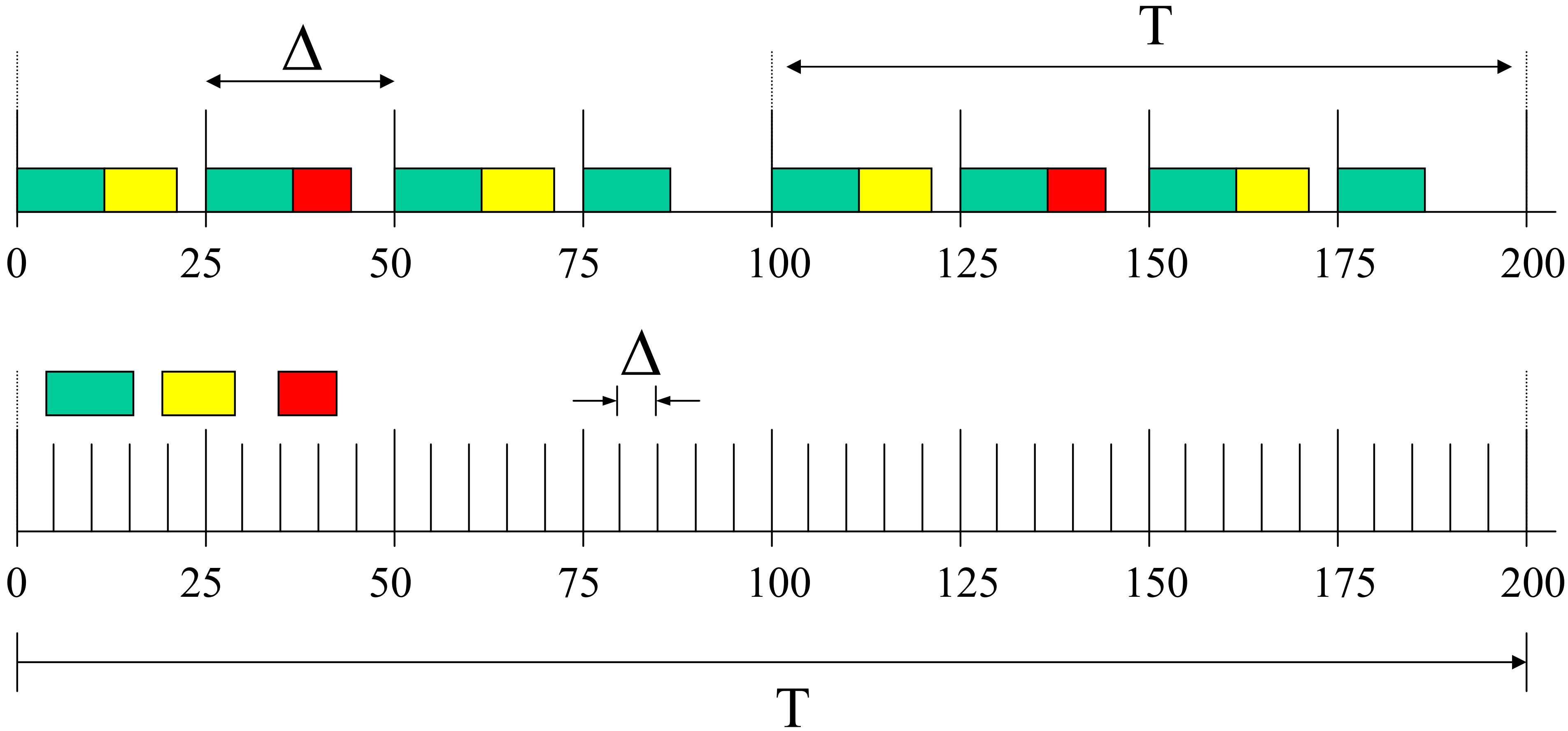
task	T	T
A	25 ms	25 ms
B	50 ms	**40 ms**
C	100 ms	100 ms
	before	**after**

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

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

40 sync. per cycle!

Example



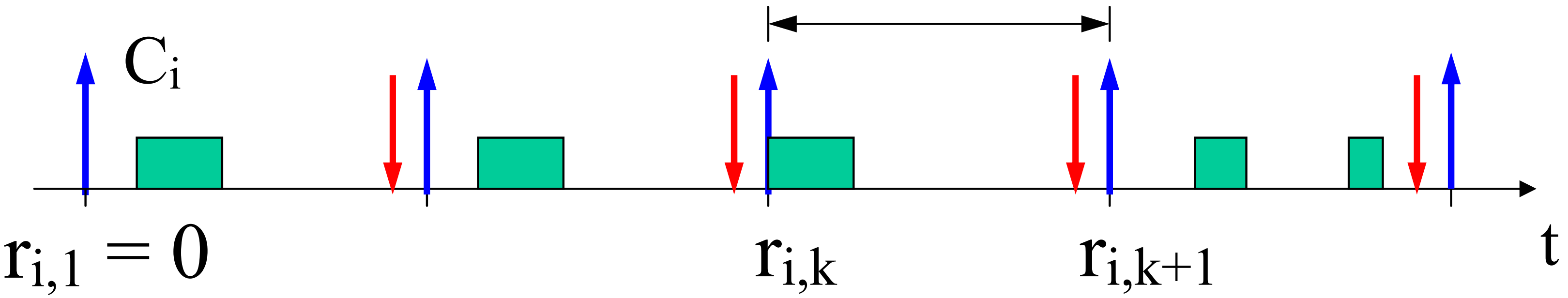
Priority Scheduling

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):**

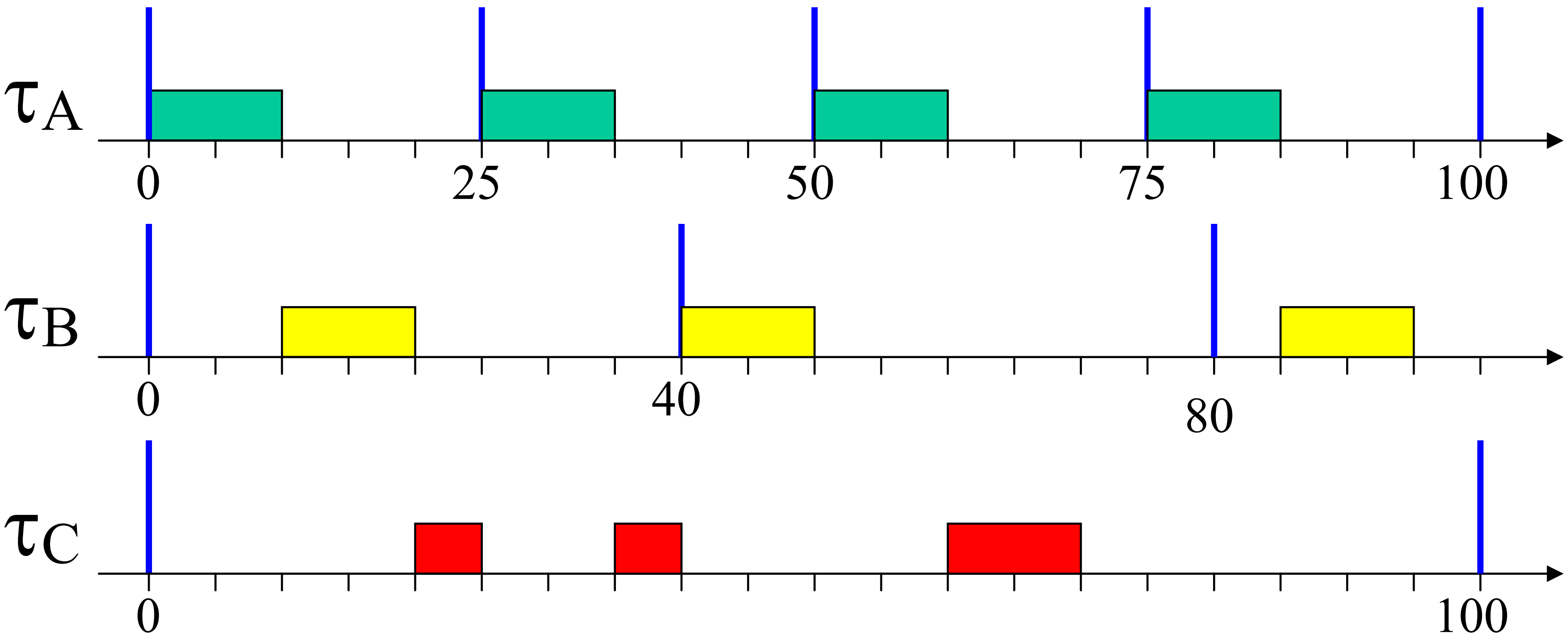
  $\mathbf{p_i \propto 1/T_i}$ (static)

- **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 fixed 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**

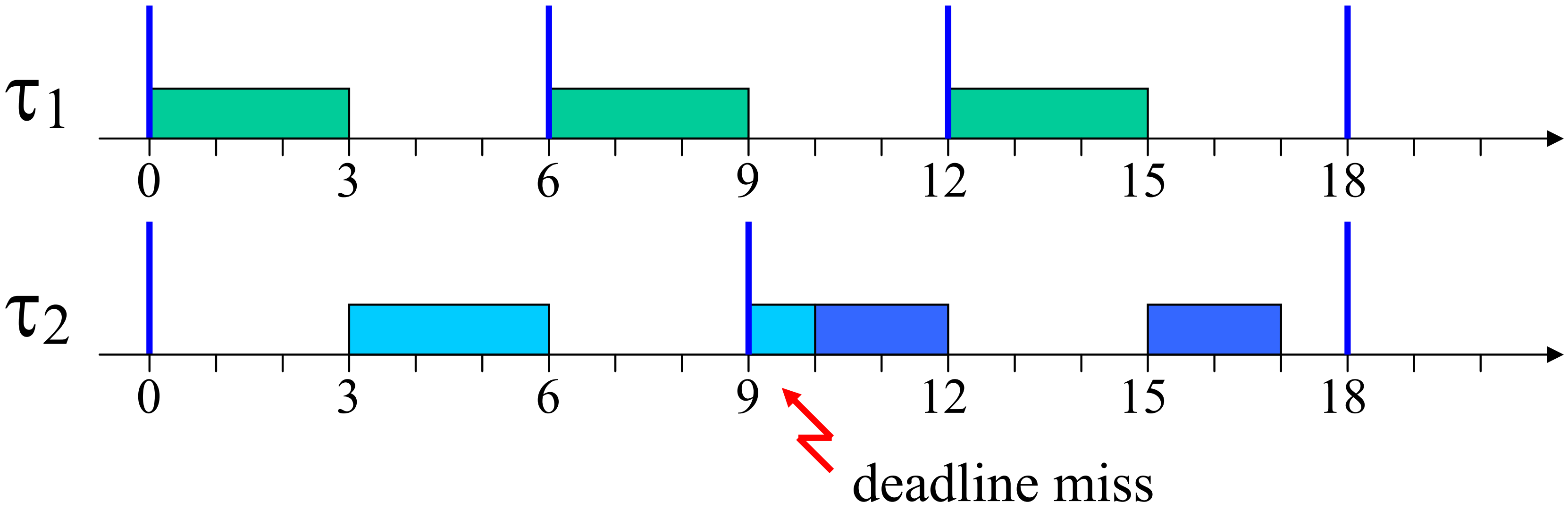
A necessary condition

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.

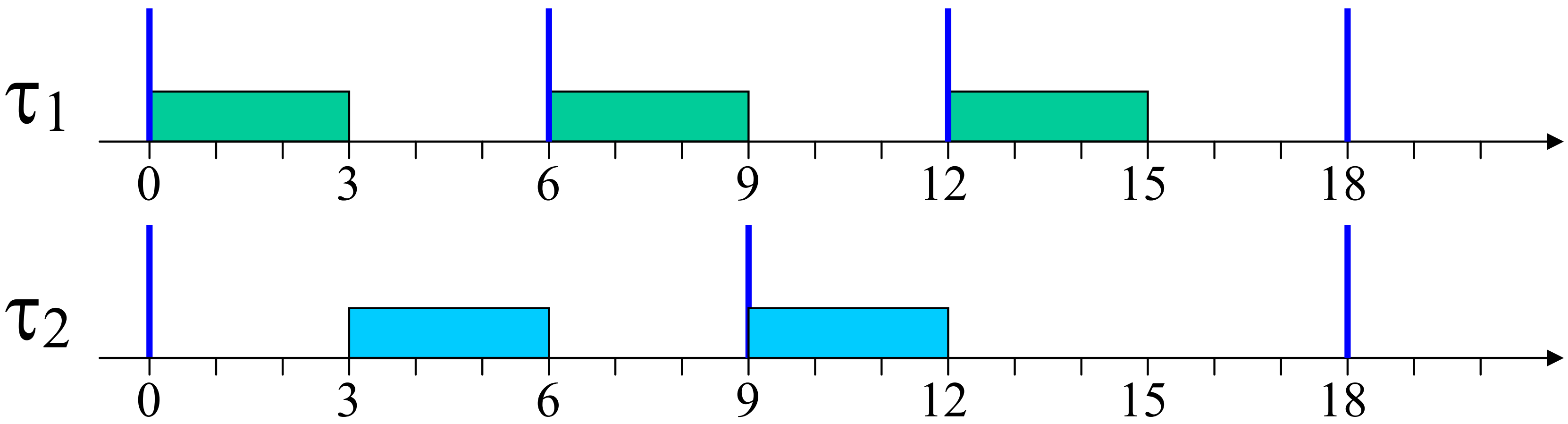
An unfeasible RM schedule

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



Utilization upper bound

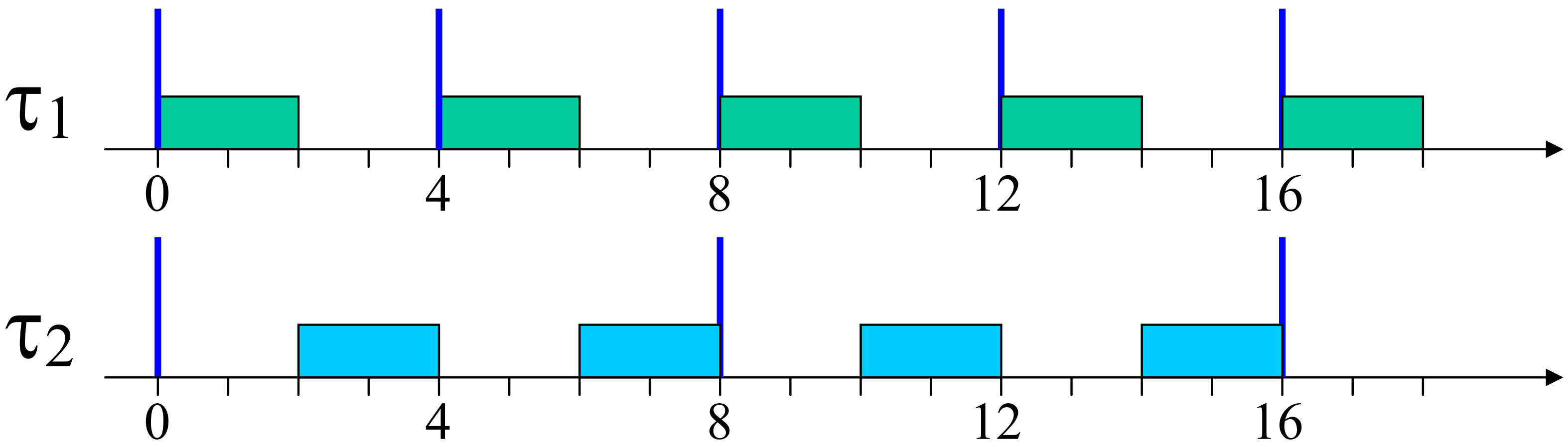
$$U_p = \frac{3}{6} + \frac{3}{9} = 0.833$$



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

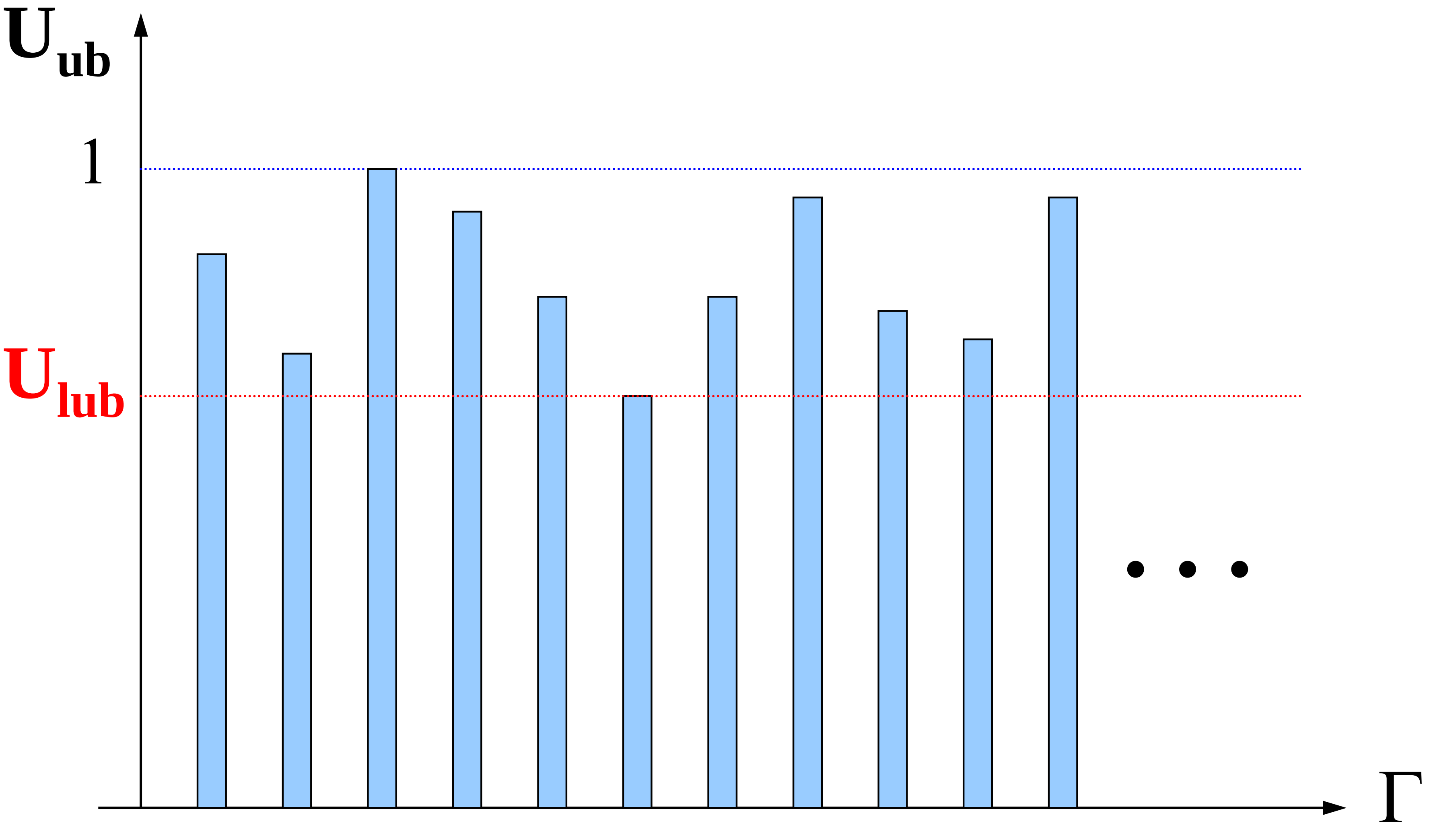
A different upper bound

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



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 $\mathbf{U_{lub} < U_p \leq 1}$ 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** $$\sum_{i=1}^{n} \frac{C_i}{T_i} \le n\left(2^{1/n} - 1\right)$$

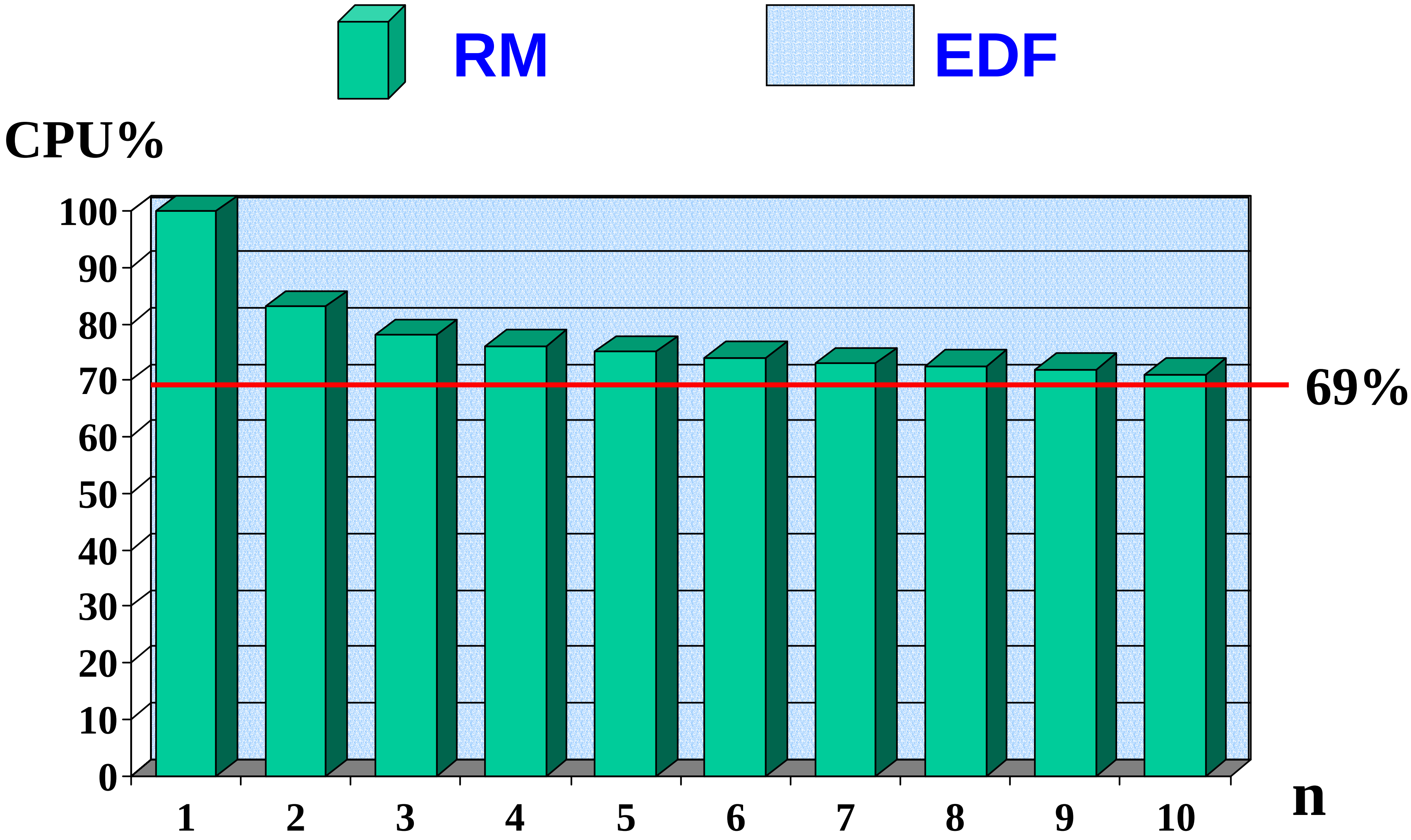
under EDF **if and only if** $$\sum_{i=1}^{n} \frac{C_i}{T_i} \le 1$$

RM bound for large *n*

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

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

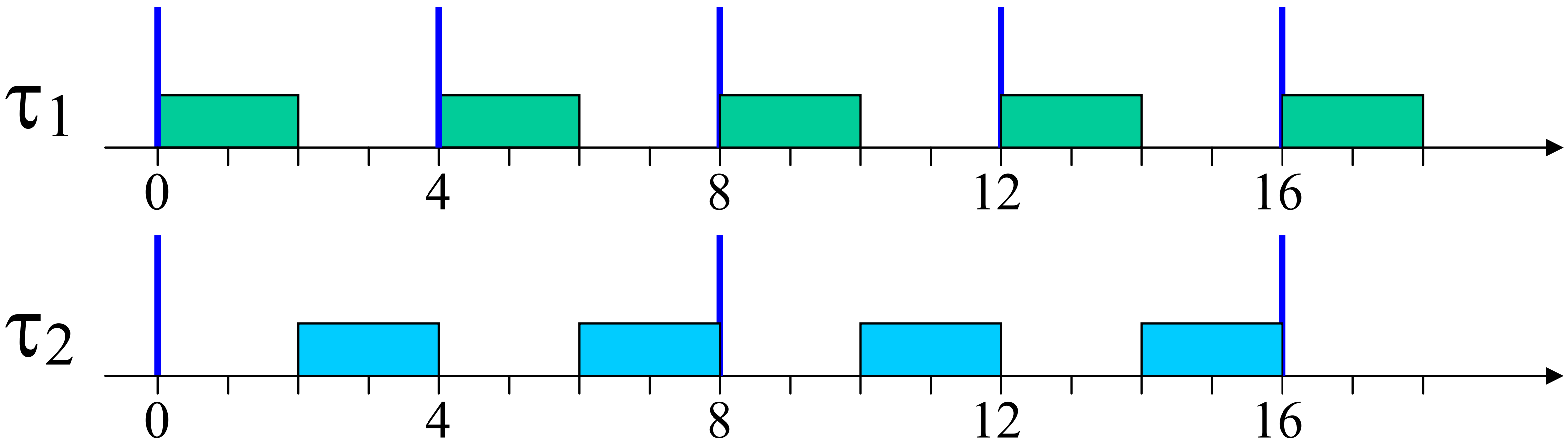
Schedulability bound



A special case

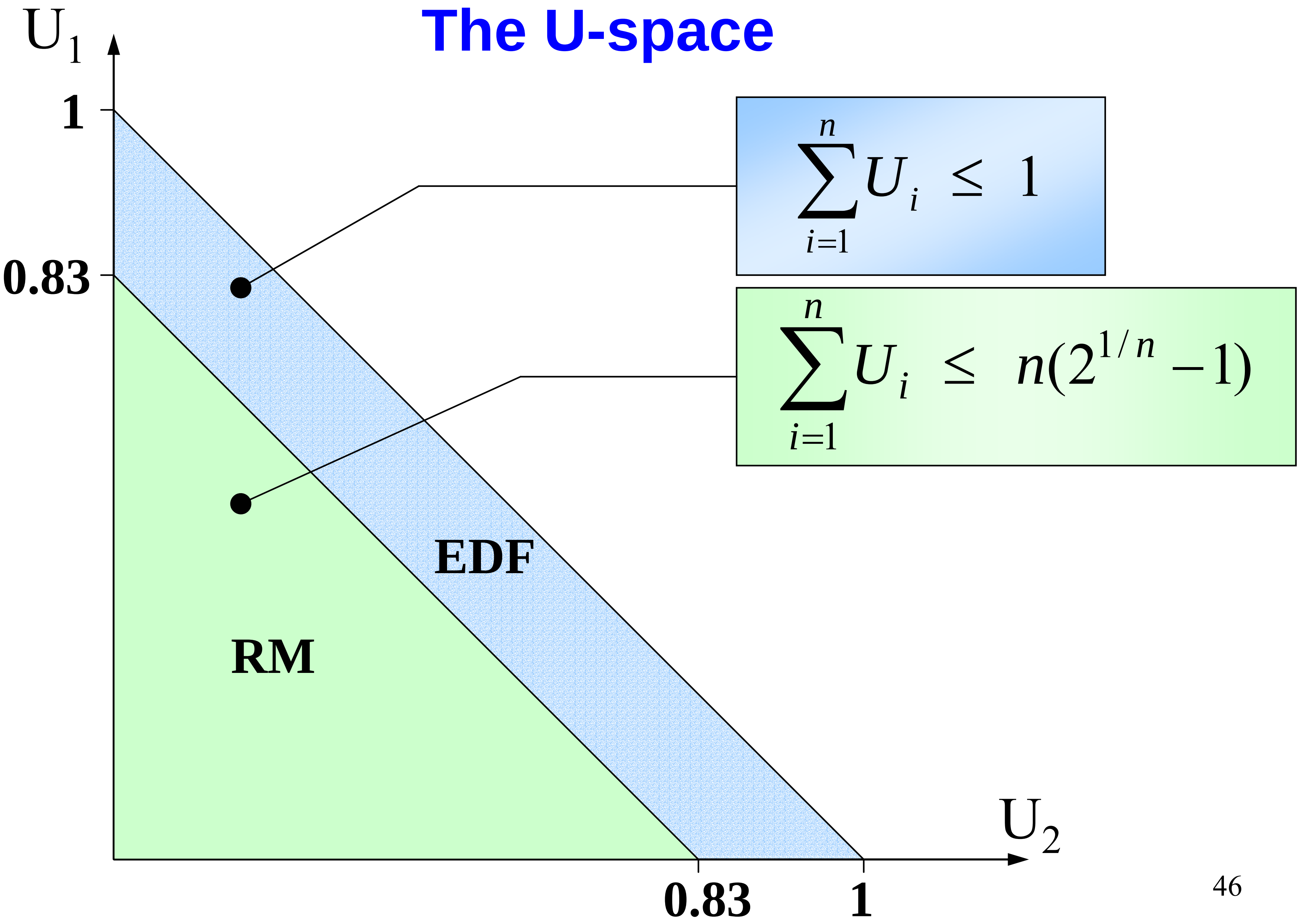
If tasks have harmonic periods $U_{lub} = 1$.

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



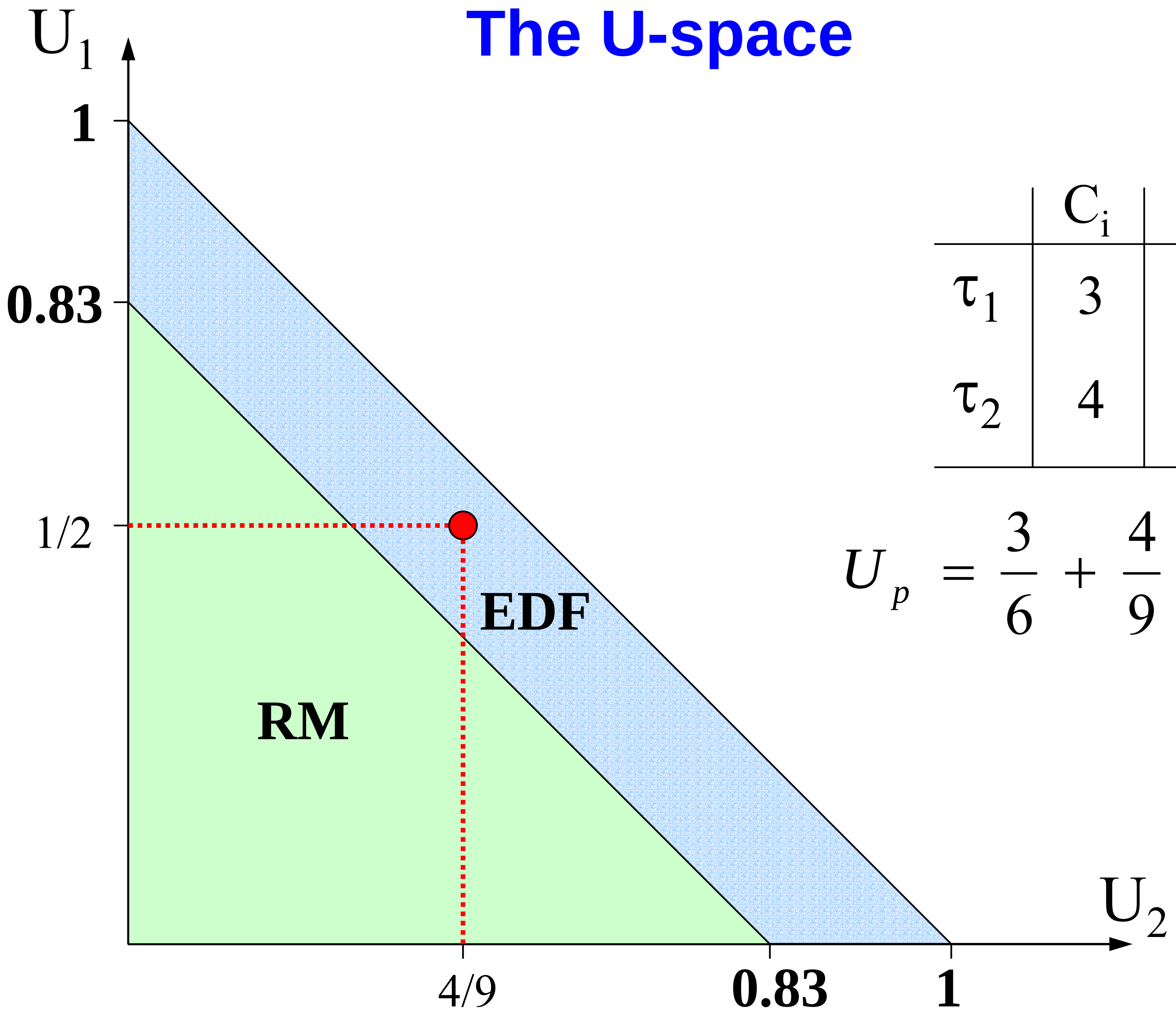
Schedulability region

The U-space



Schedulability region

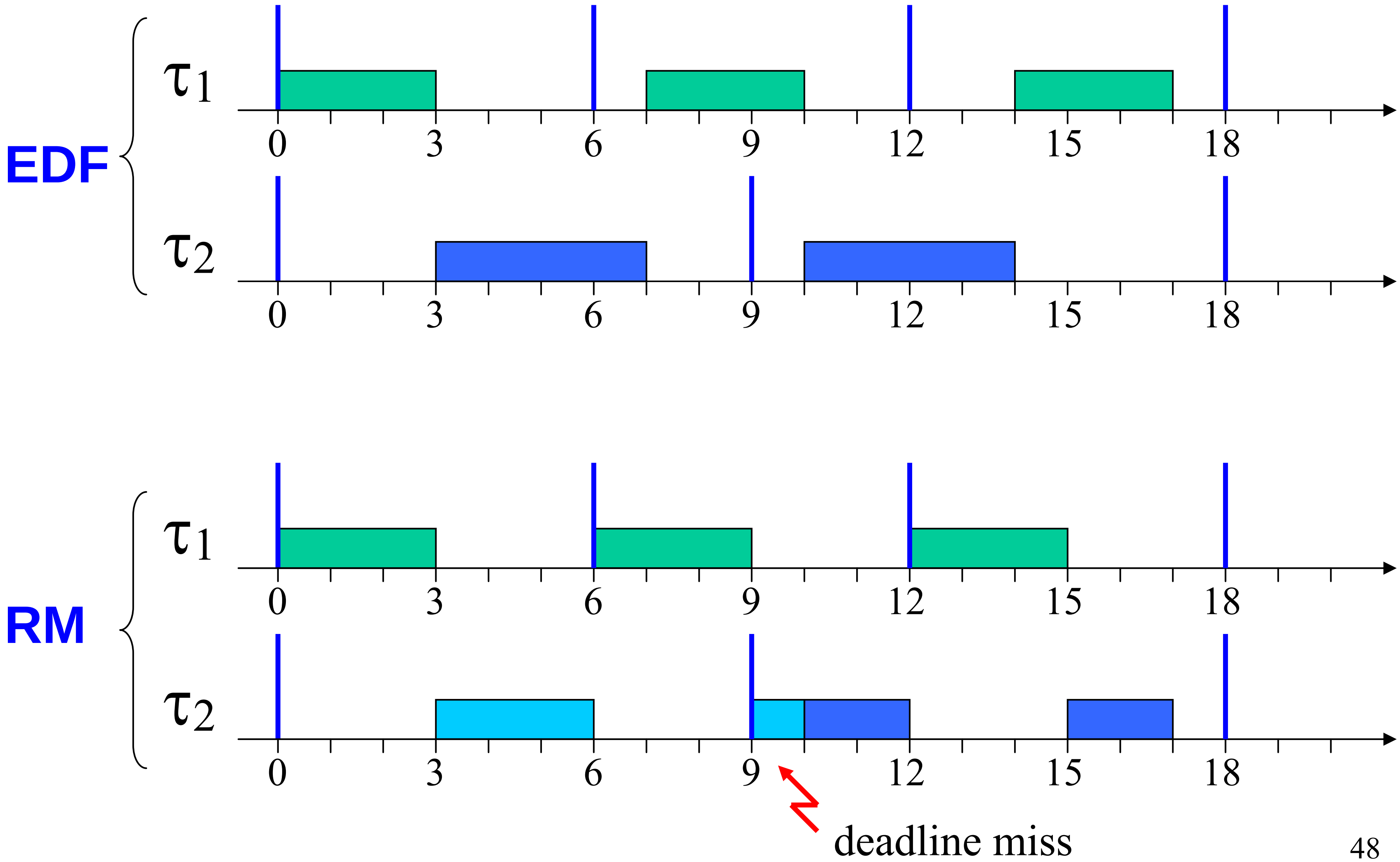
The U-space



	C_i	T_i
τ_1	3	6
τ_2	4	9

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

Schedule



EDF
τ1
0 3 6 9 12 15 18
τ2
0 3 6 9 12 15 18
RM
τ1
0 3 6 9 12 15 18
τ2
0 3 6 9 12 15 18
deadline miss

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.

⇕

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

EDF Optimality

EDF is **optimal** among <u>all</u> algorithms:

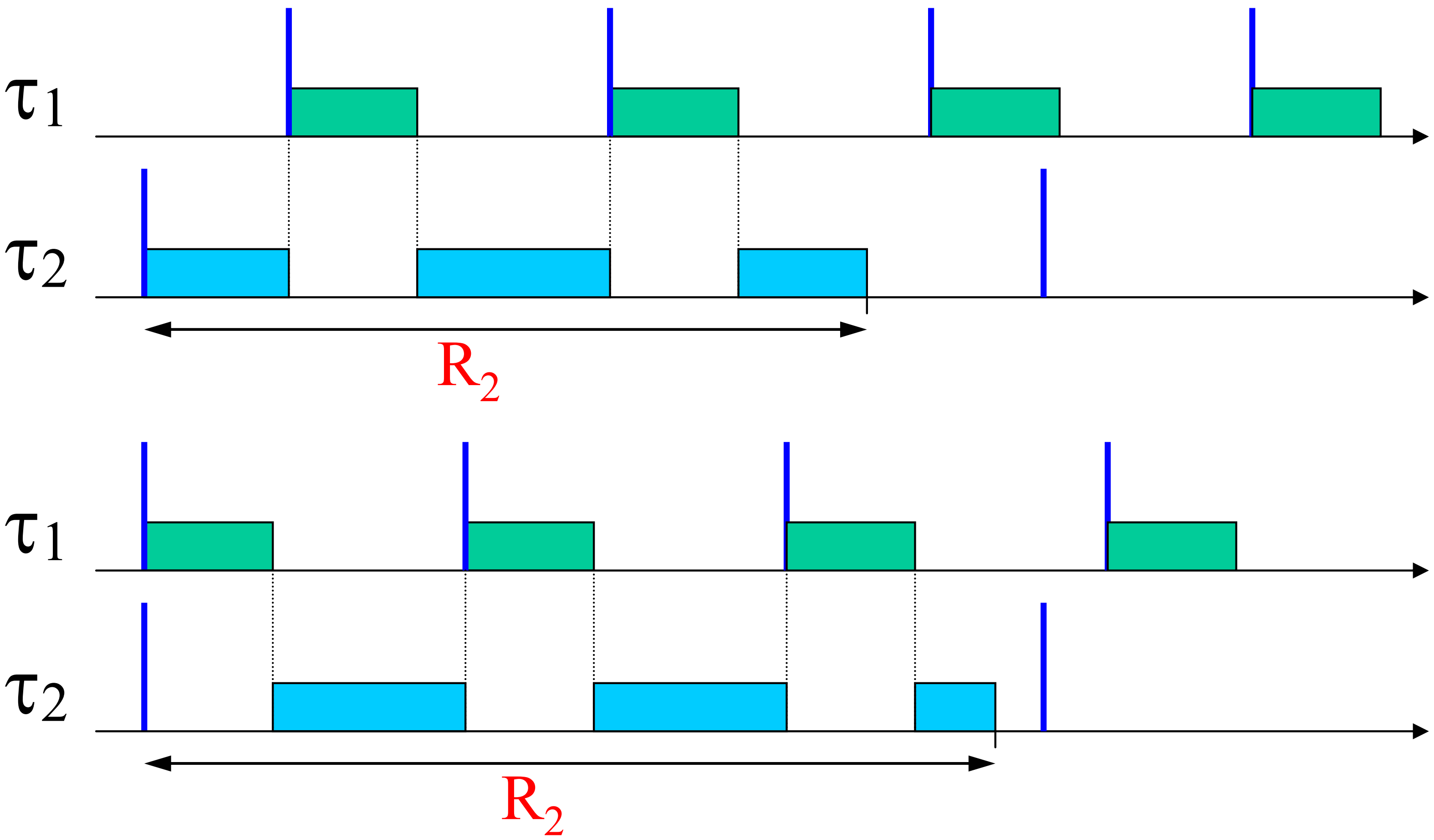
If there exists a feasible schedule for Γ, then EDF will generate a feasible schedule.

$\Updownarrow$

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.



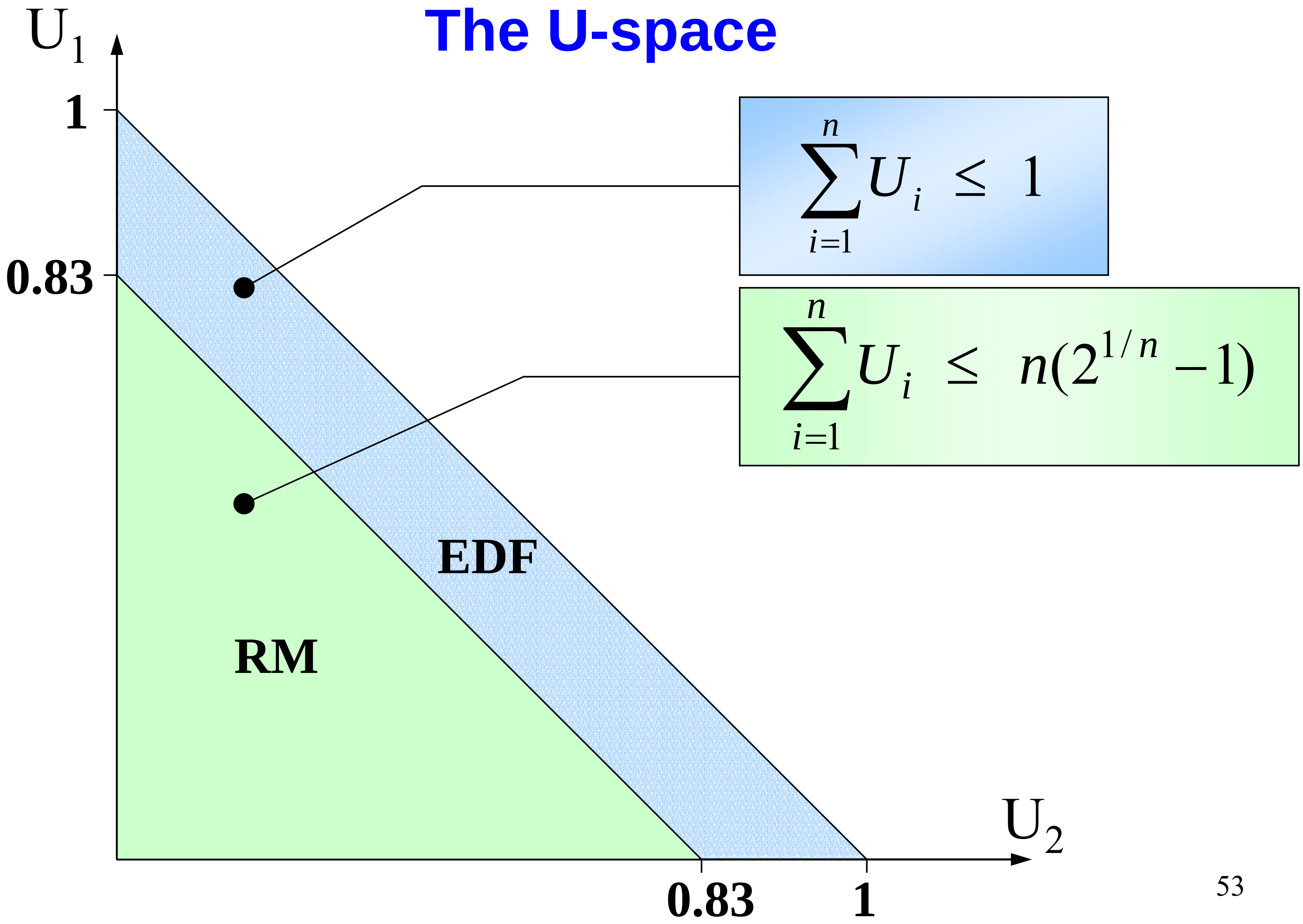
The Hyperbolic Bound

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

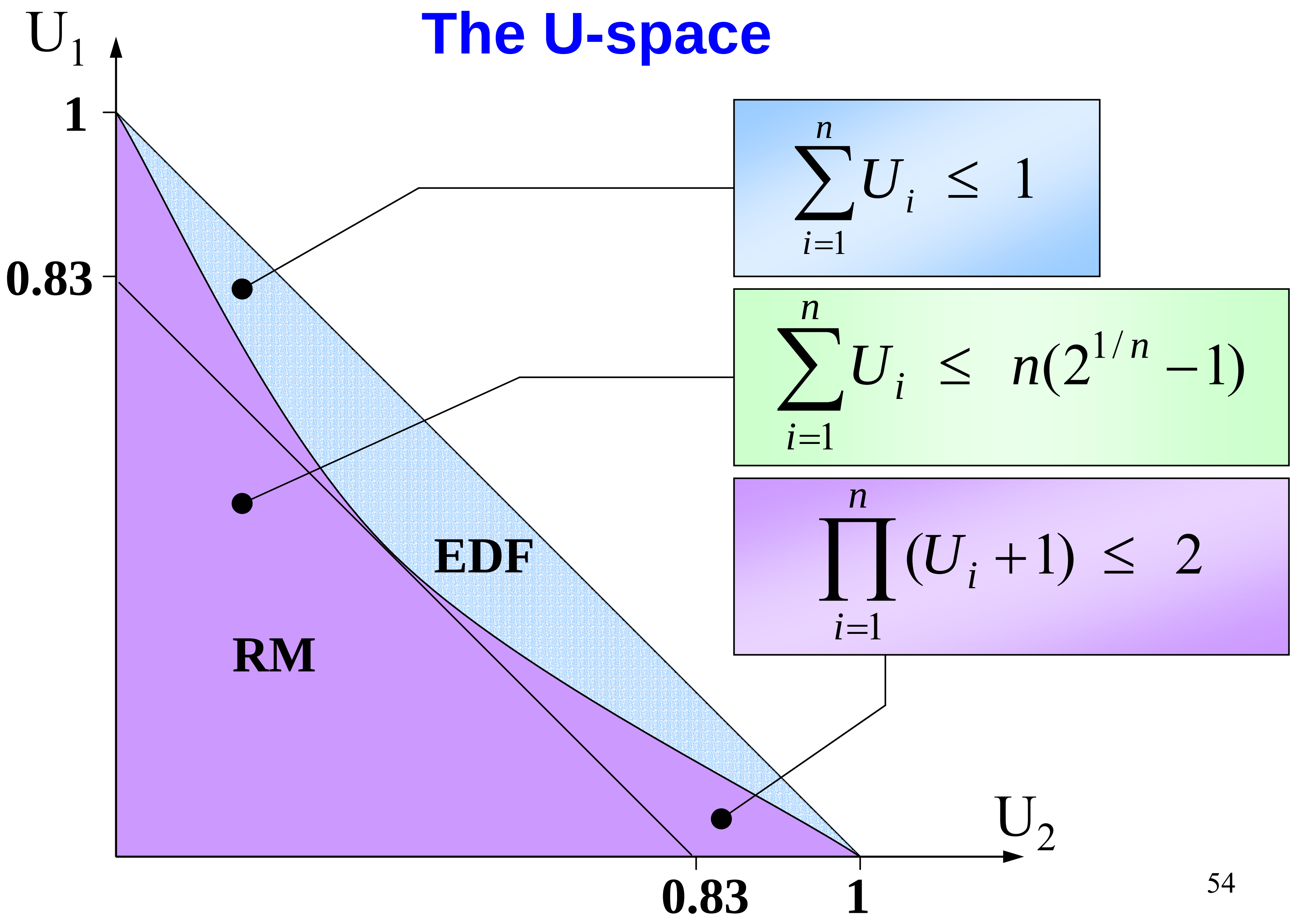
$$\prod_{i=1}^{n} (U_i + 1) \leq 2$$

Schedulability region

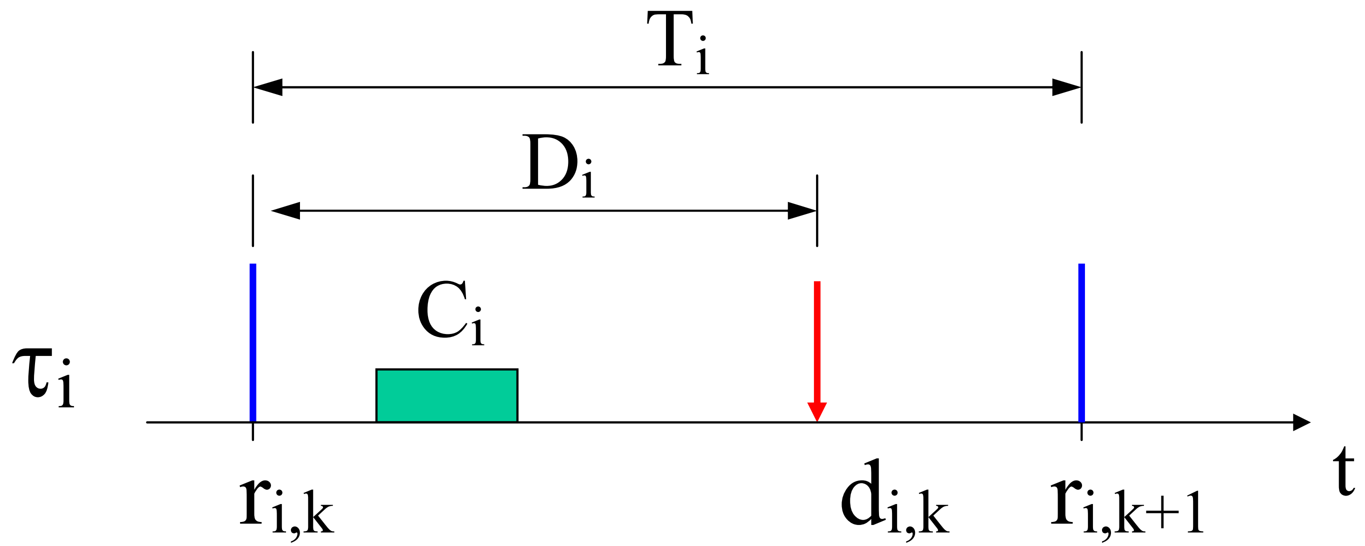
The U-space



Schedulability region



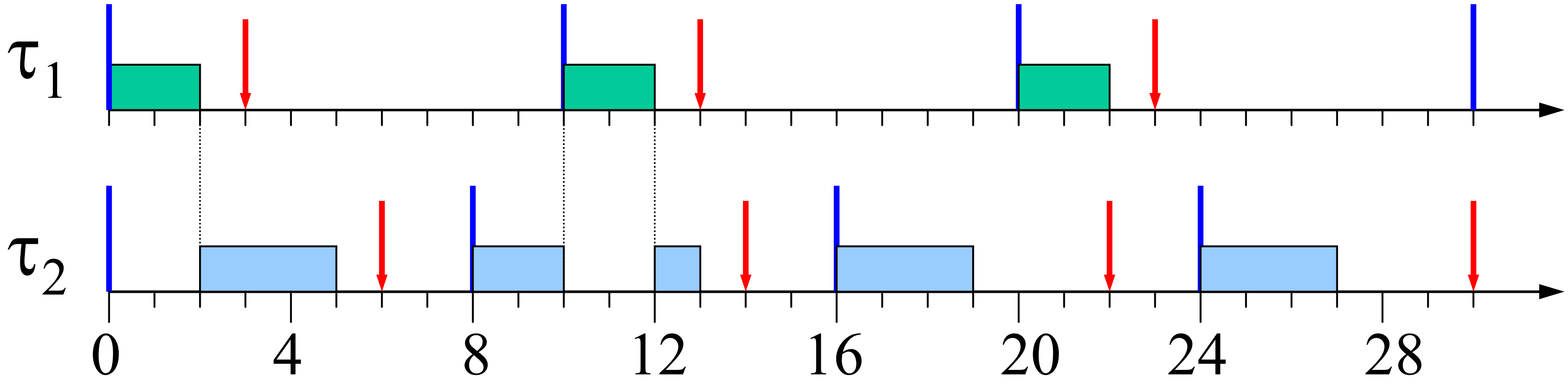
Extension to tasks with D < T



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

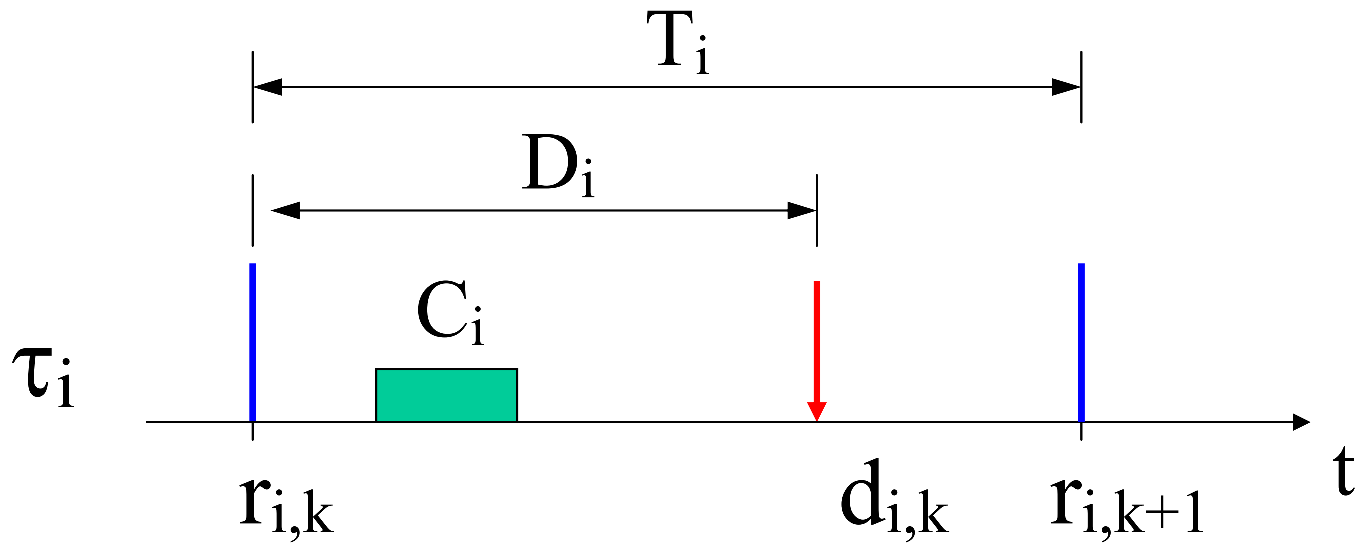


Problem with the Utilization Bound

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

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:

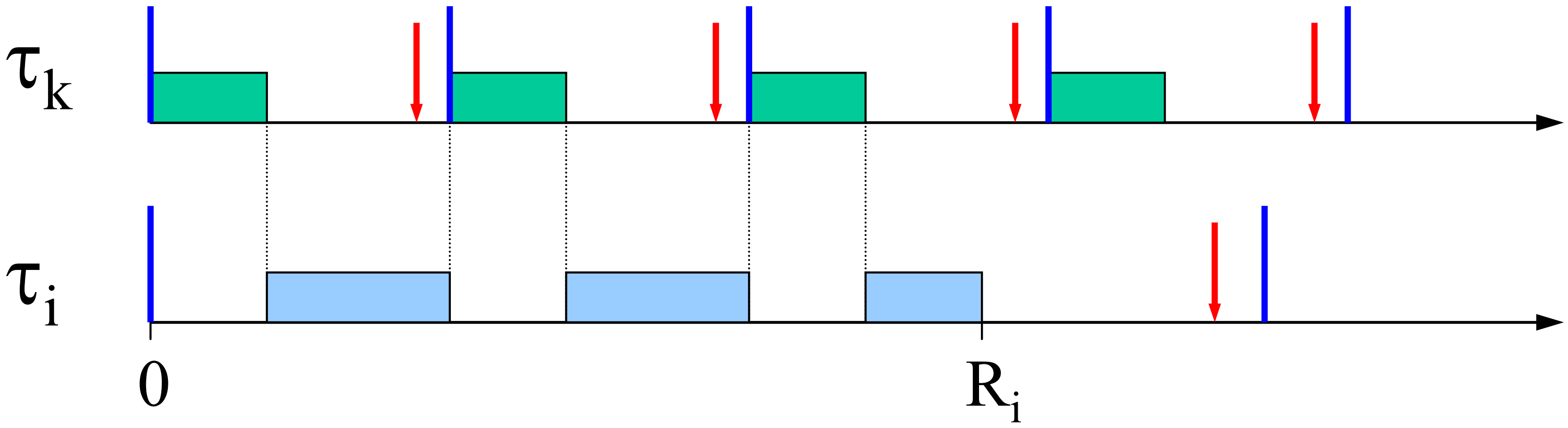
$$I_i = \sum_{D_k < D_i} C_k$$

- compute its response time as

$$R_i = C_i + I_i$$

- verify if $R_i \le D_i$

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

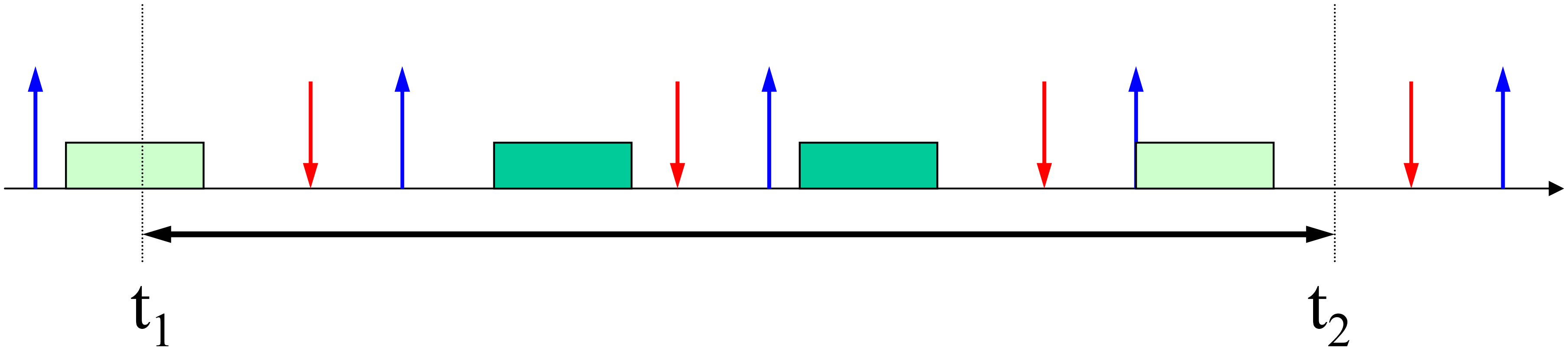
[Baruah, Howell, Rosier 1990]

For checking the existence of feasibile schedule and for **EDF**

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

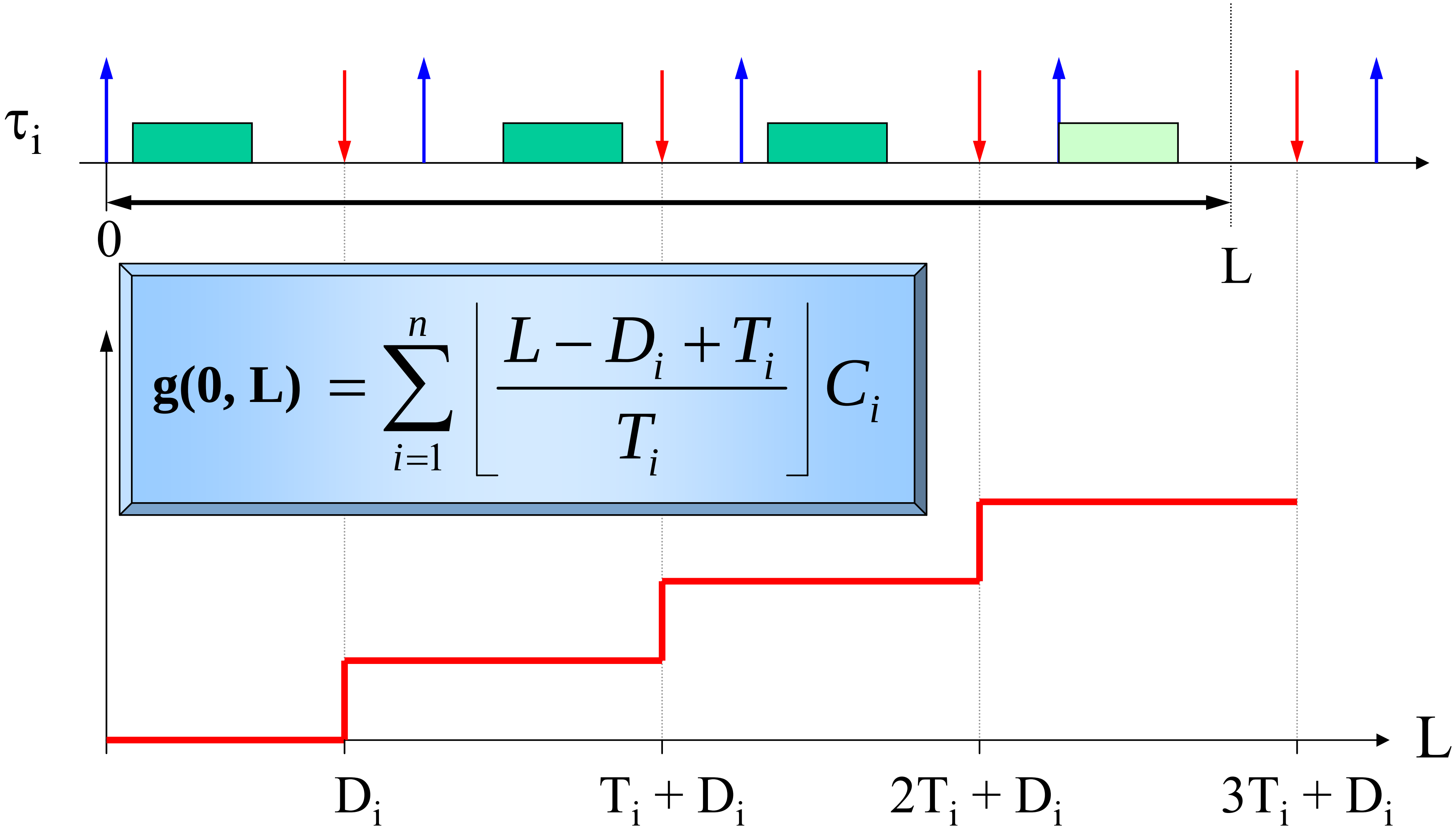


The demand in $[t_1, t_2]$ is the computation time of those jobs started at or after t_1 with deadline less than or equal to t_2:

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

Processor Demand

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



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

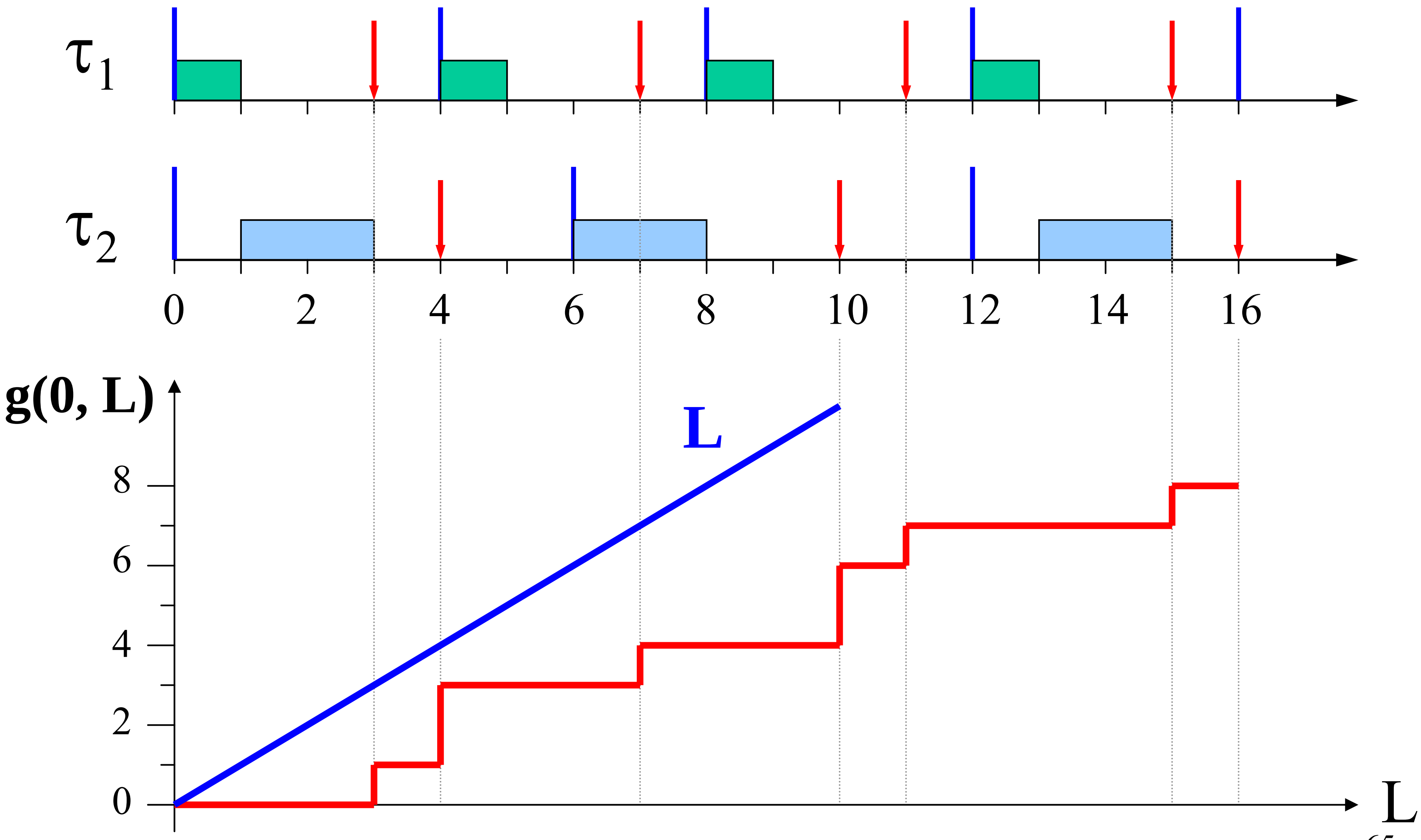
Processor Demand Test

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

Question

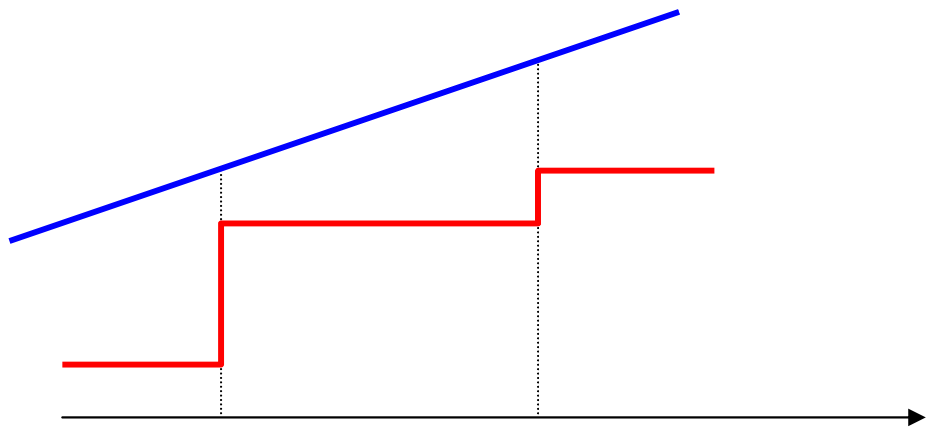
How can we bound the number of intervals in which the test has to be performed?

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:

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

Bounding complexity

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

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

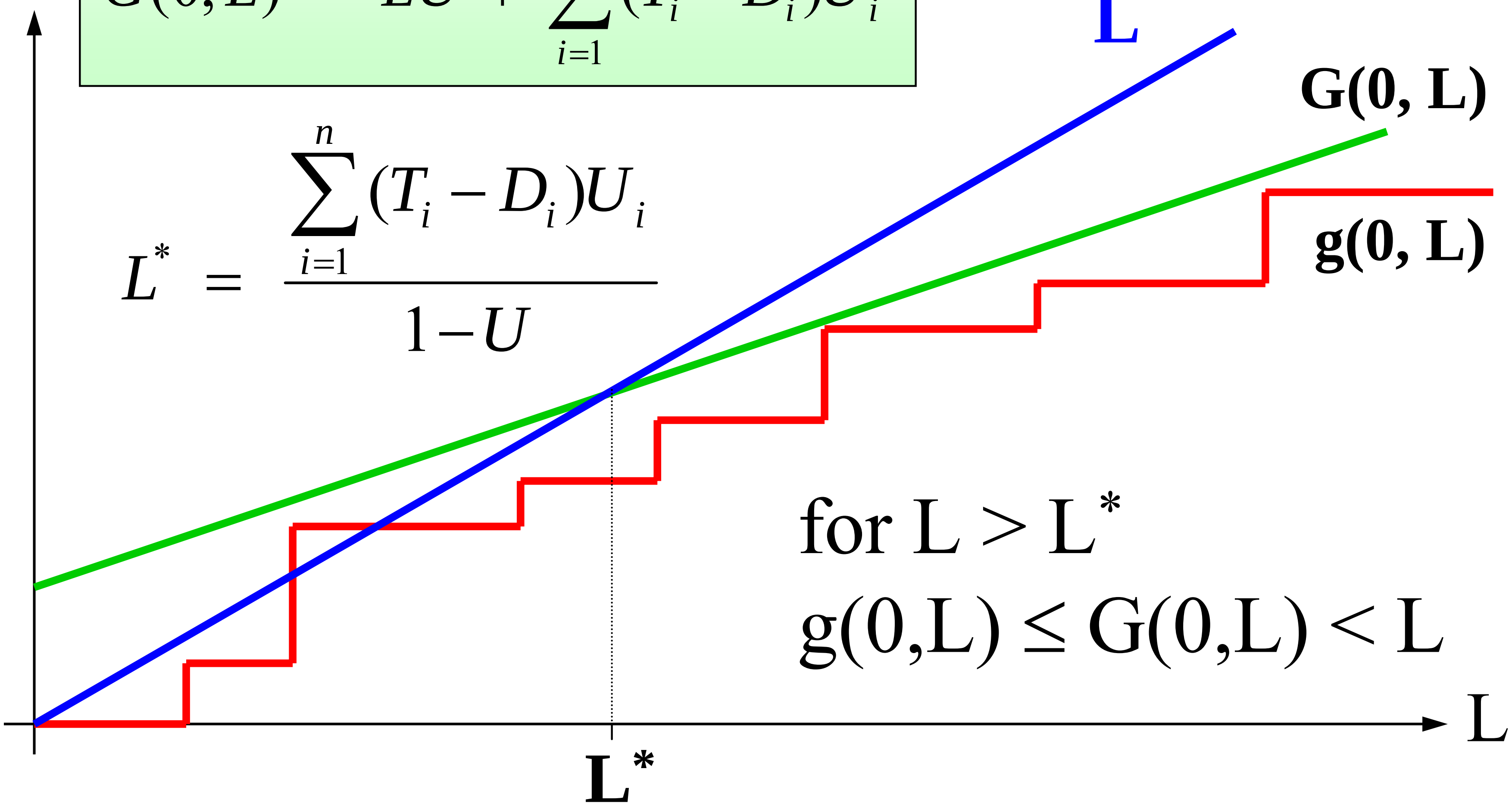
$$= \sum_{i=1}^{n} L \frac{C_i}{T_i} + \sum_{i=1}^{n} (T_i - D_i) \frac{C_i}{T_i}$$

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

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 $L > L^*$

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

Processor Demand Test

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

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

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

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

Summarizing: RM vs. EDF

	$D_i = T_i$	$D_i \le T_i$
RM	***Suff.: polynomial*** $O(n)$ LL: $\Sigma U_i \le n(2^{1/n} - 1)$ HB: $\Pi(U_i + 1) \le 2$ ***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:*** $O(n)$ $\Sigma U_i \le 1$	***pseudo-polynomial*** Processor Demand Analysis $\forall L > 0, \quad g(0, L) \le L$

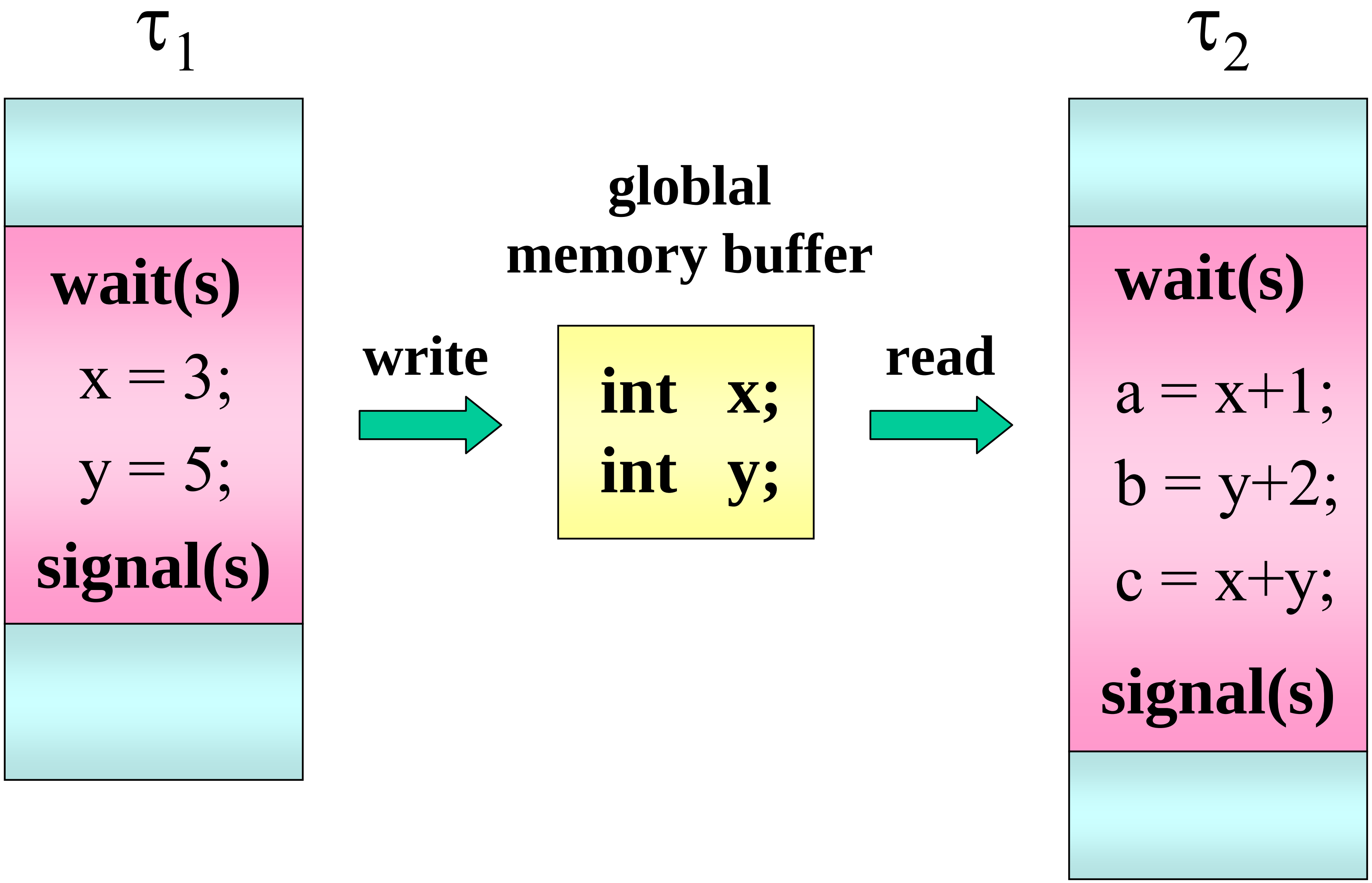
Inter-task communication mechanisms

- Shared memory
- Message passing ports
- Asynchonous buffers

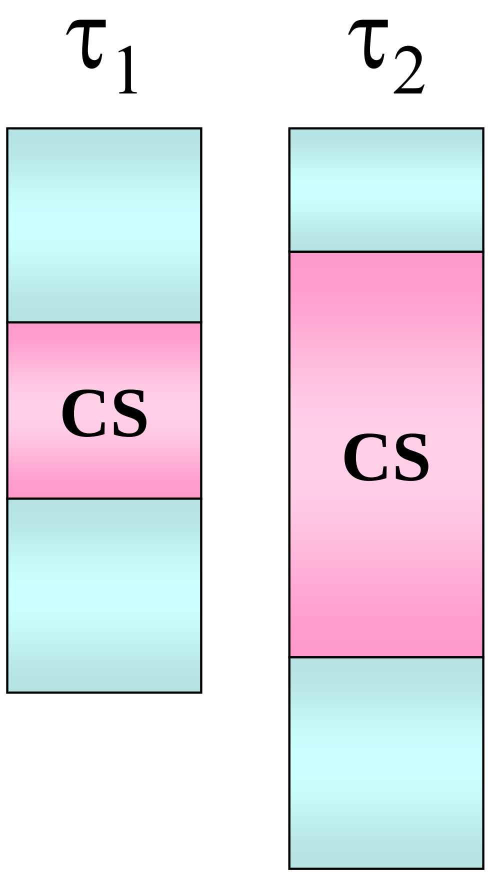
Handling shared resources

Problems caused by mutual exclusion

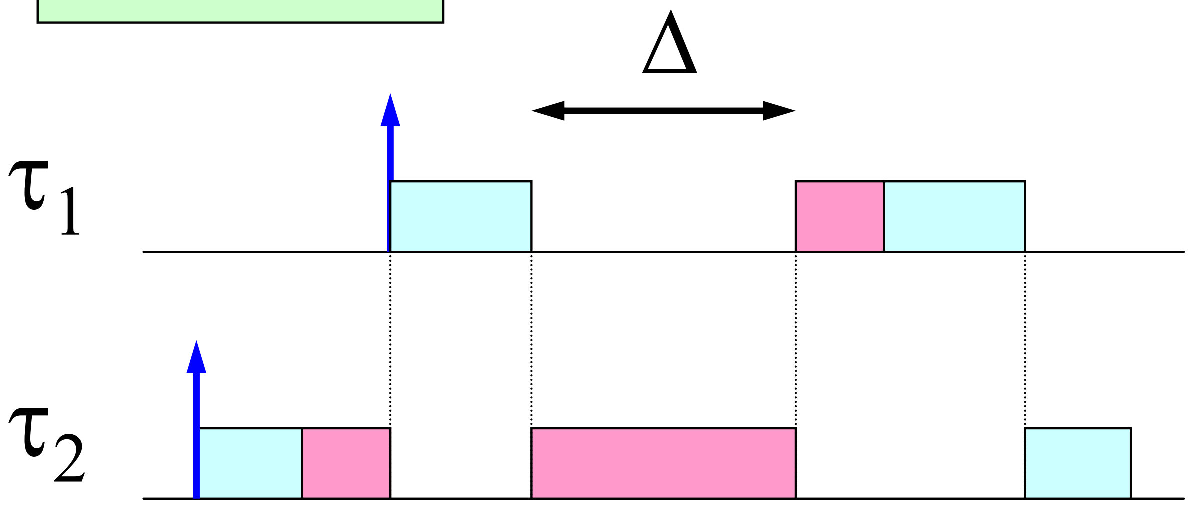
Critical sections



Blocking on a semaphore

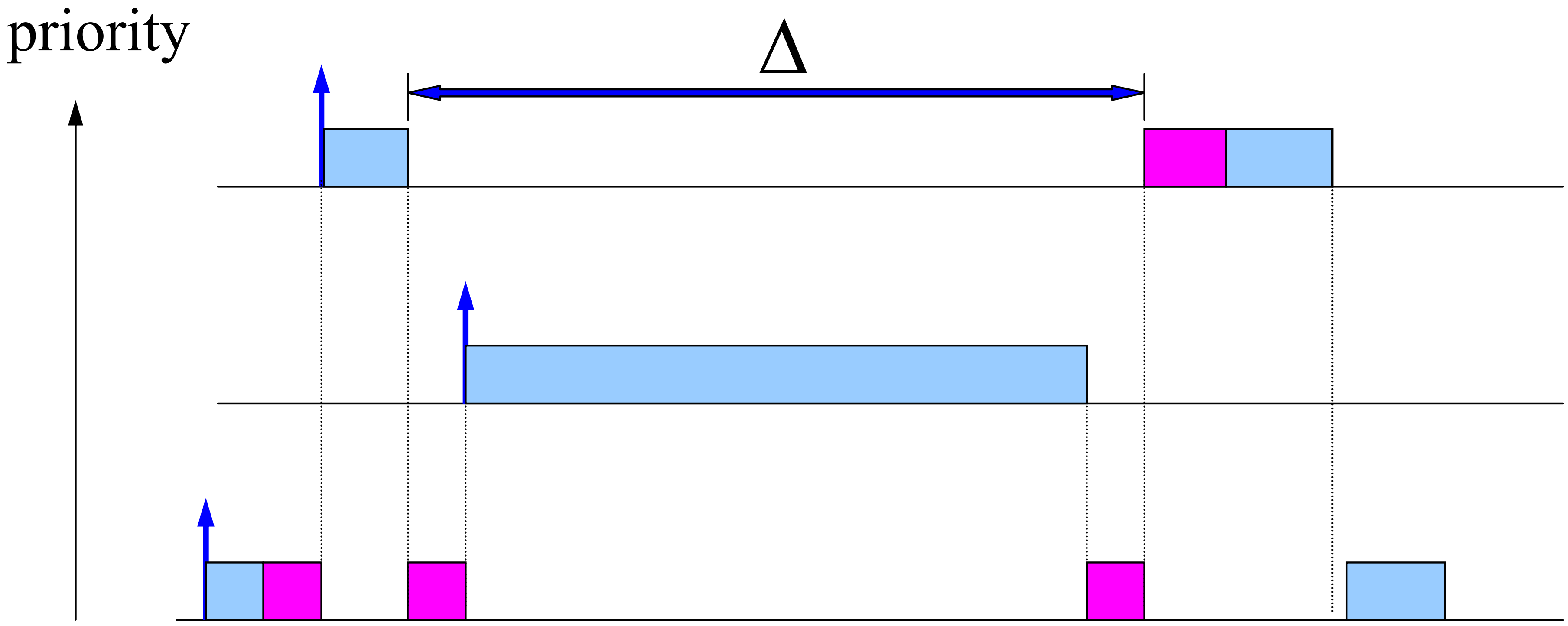


$p_1 > p_2$



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

Priority Inversion



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

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)

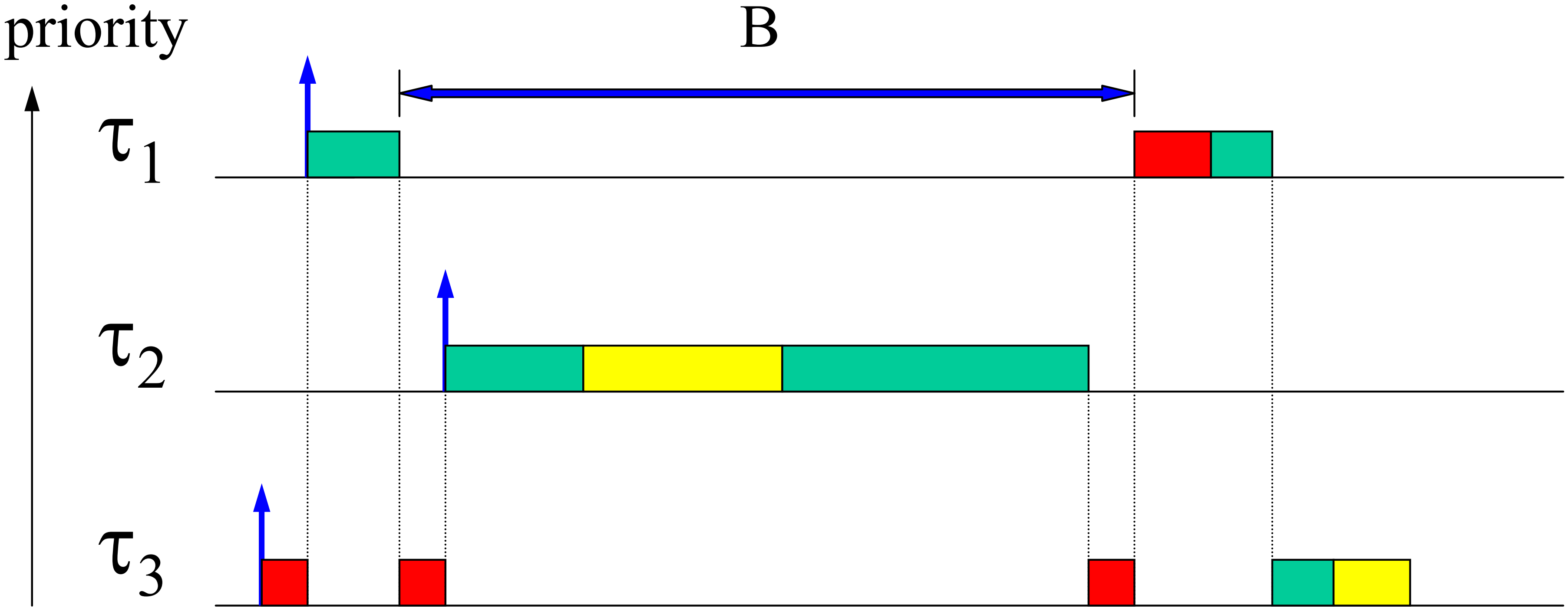
Non Preemptive Protocol

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

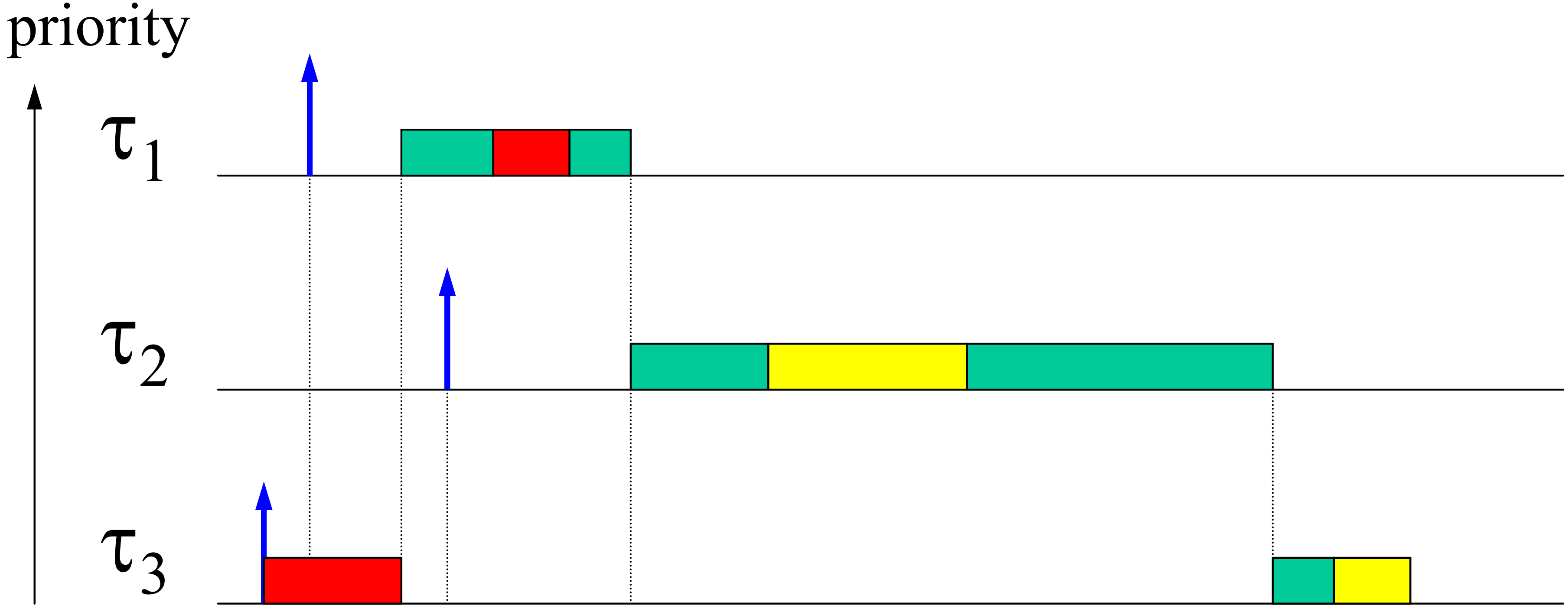
ADVANTAGES: simplicity

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

Conflict on critical section

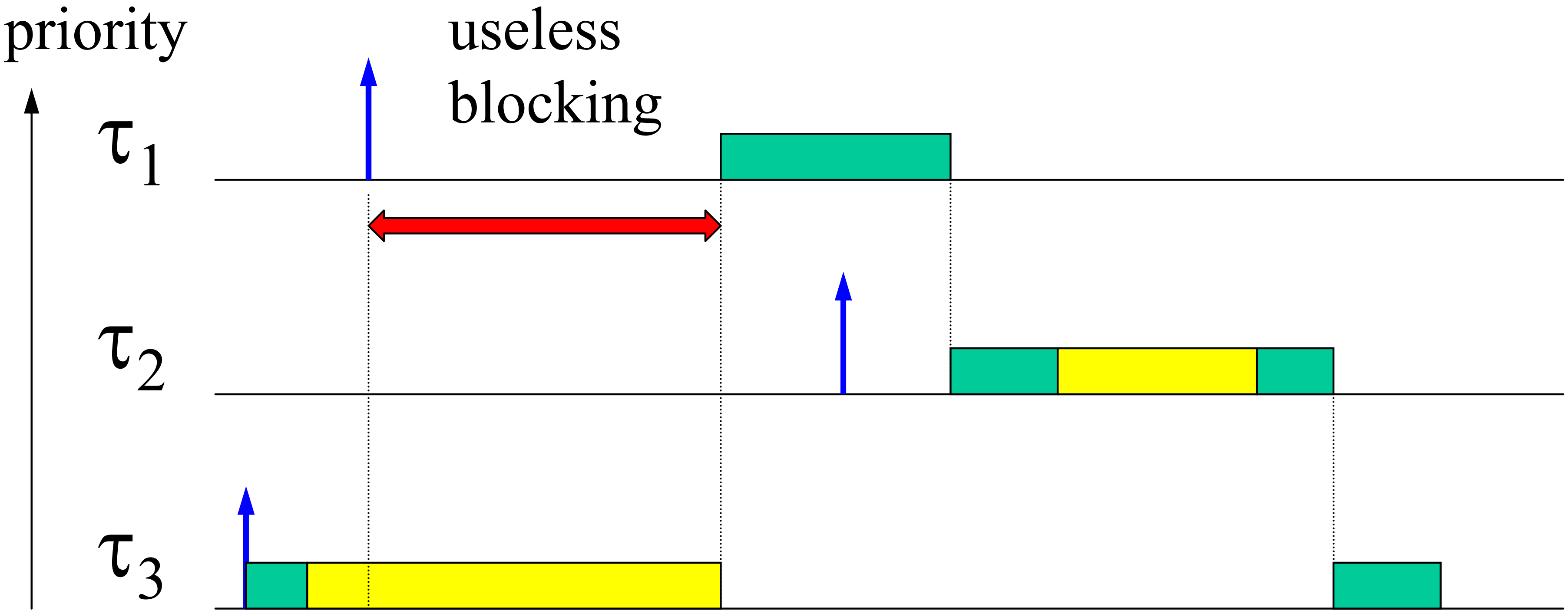


Schedule with NPP



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

Problem with NPP



τ_1 cannot preemt, although it could

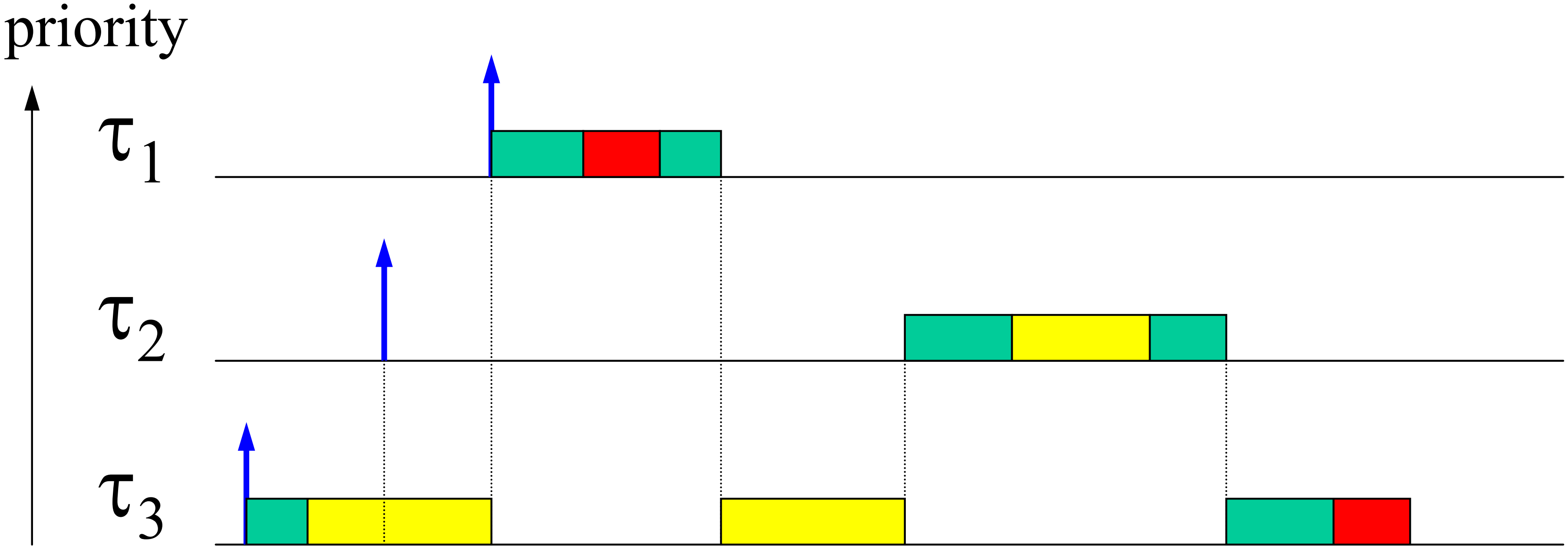
Highest Locker Priority

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.

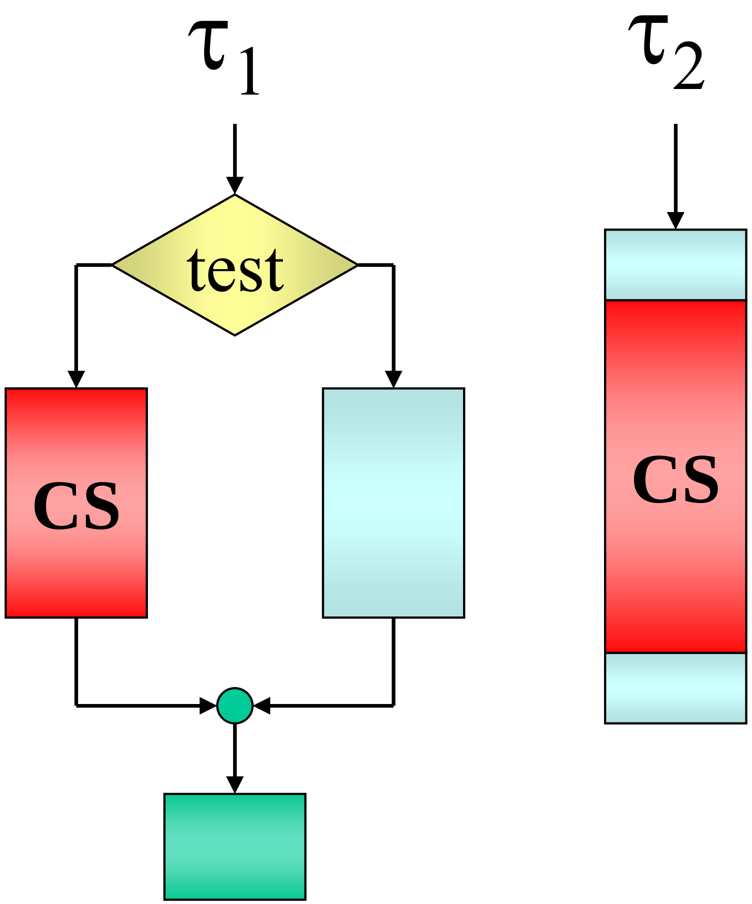
Schedule with HLP



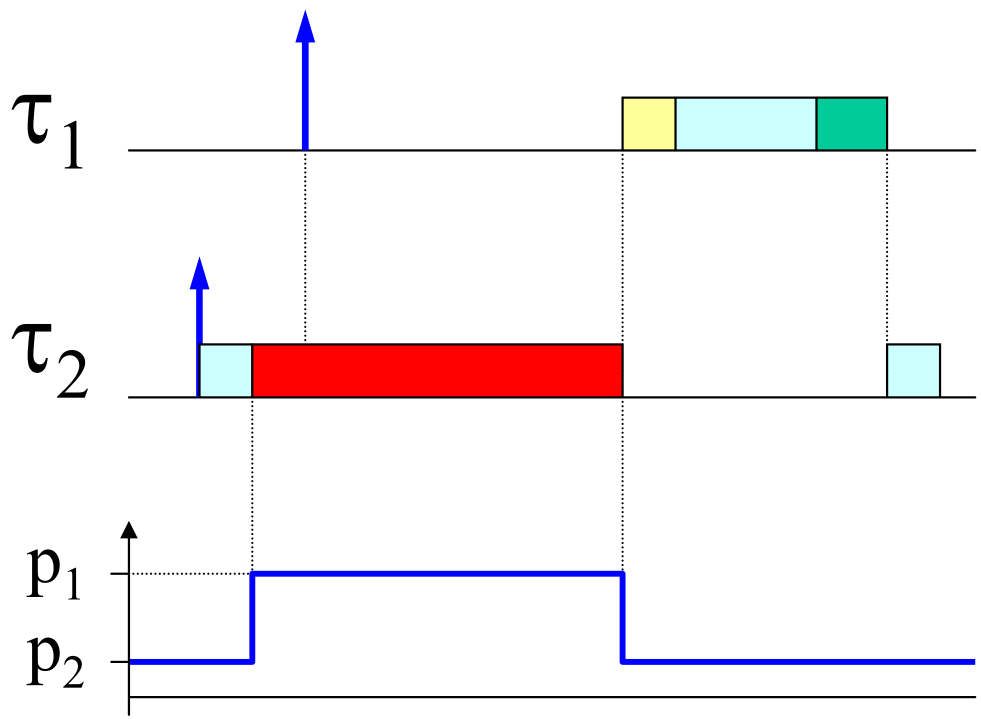
$$P_{CS} = \max\ \{P_k \mid \tau_k \text{ uses CS}\}$$

τ_2 is blocked, but τ_1 can preempt within a CS

Problem with HLP



τ_1 blocks just in case ...



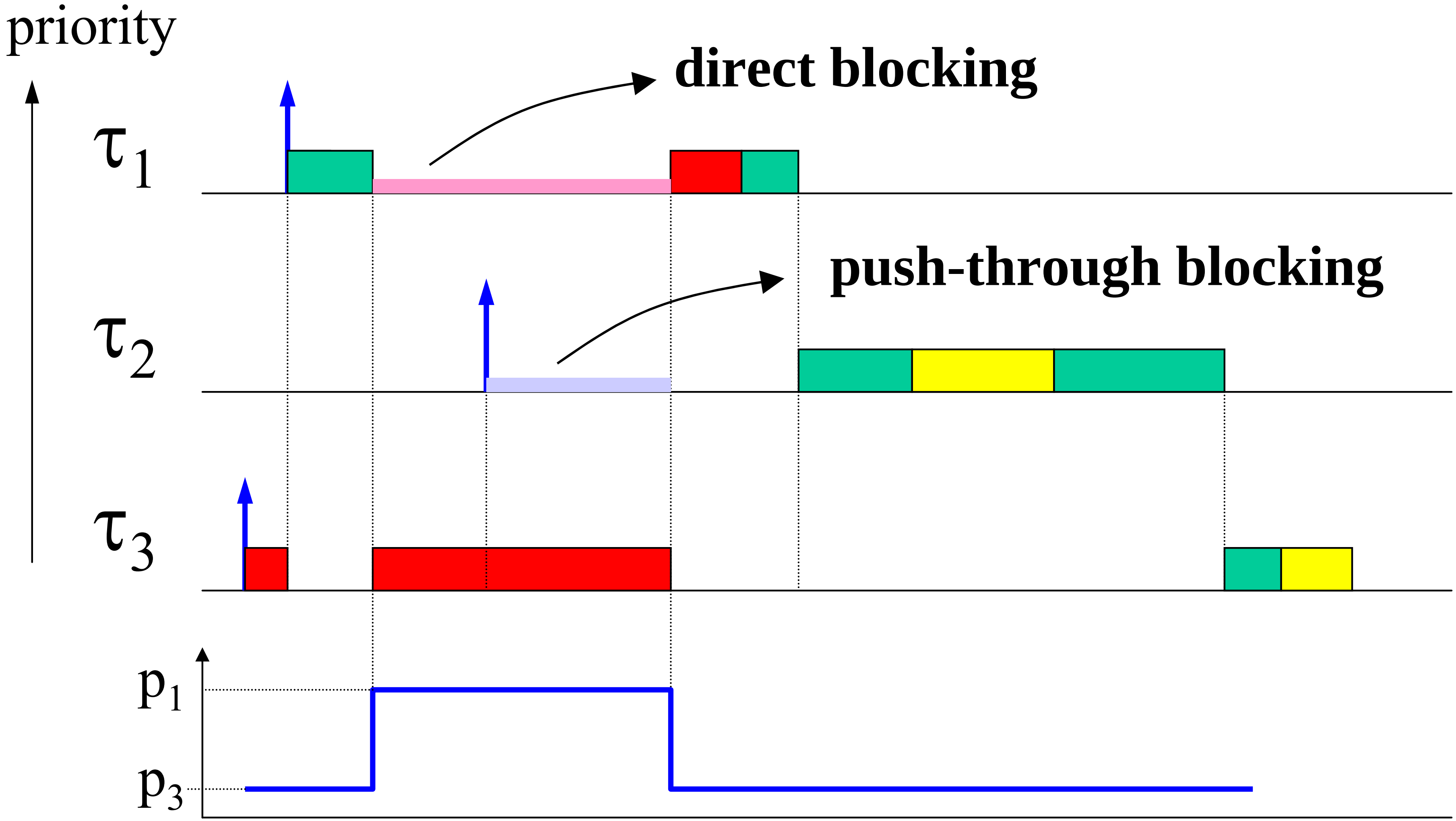
Priority Inheritance Protocol

[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

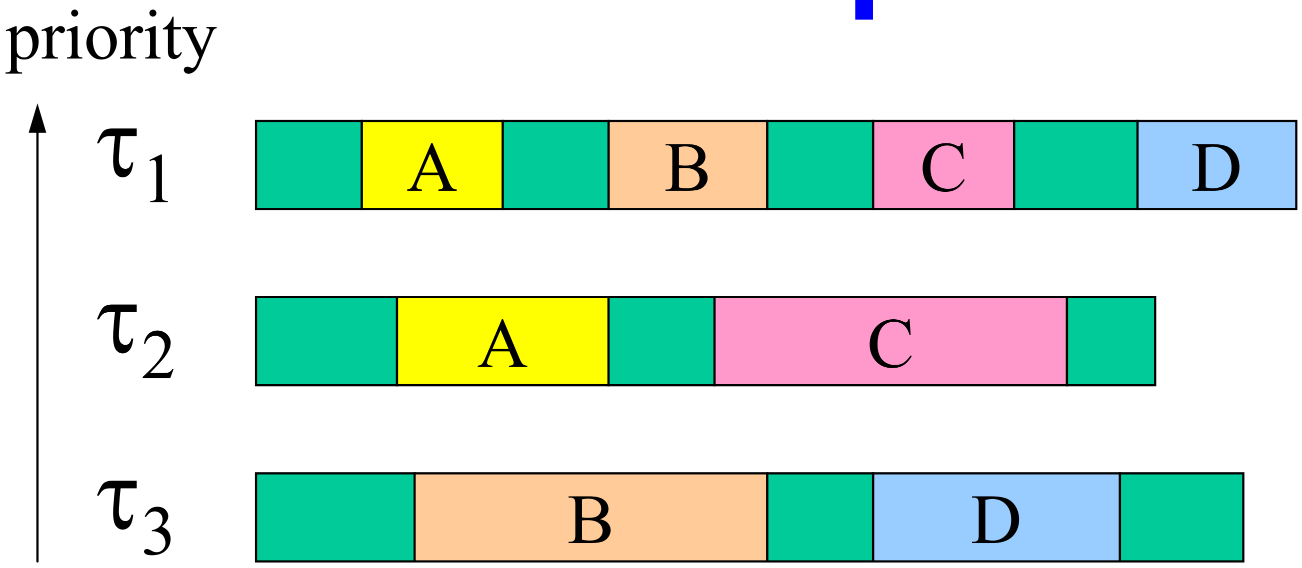
Theorem: τ_i can be blocked at most once by each lower priority task

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**

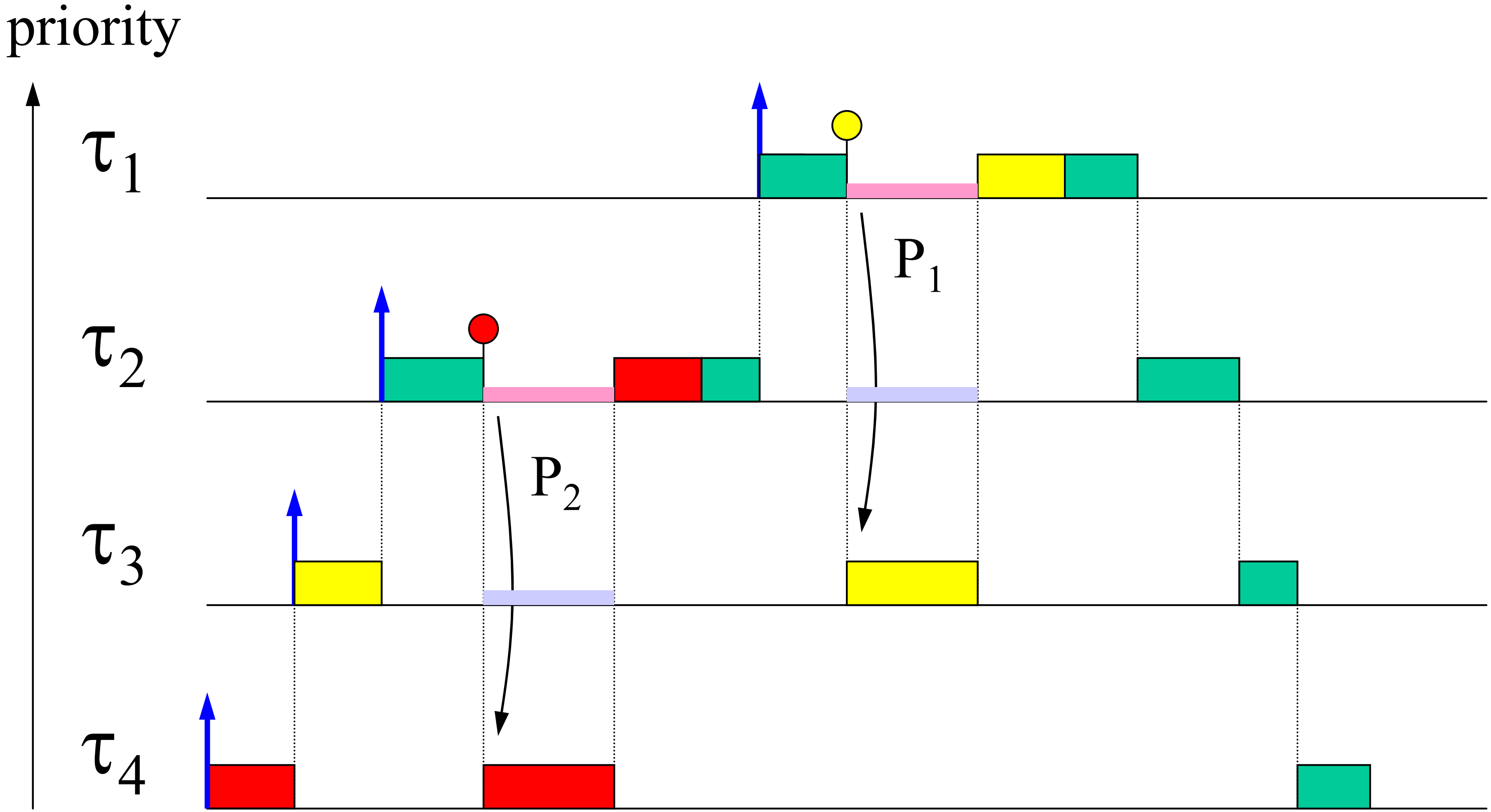
Theorem: τ_i can be blocked at most for the duration of **min(n,m)** critical sections

Example



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

Schedule with PIP



Remarks on PIP

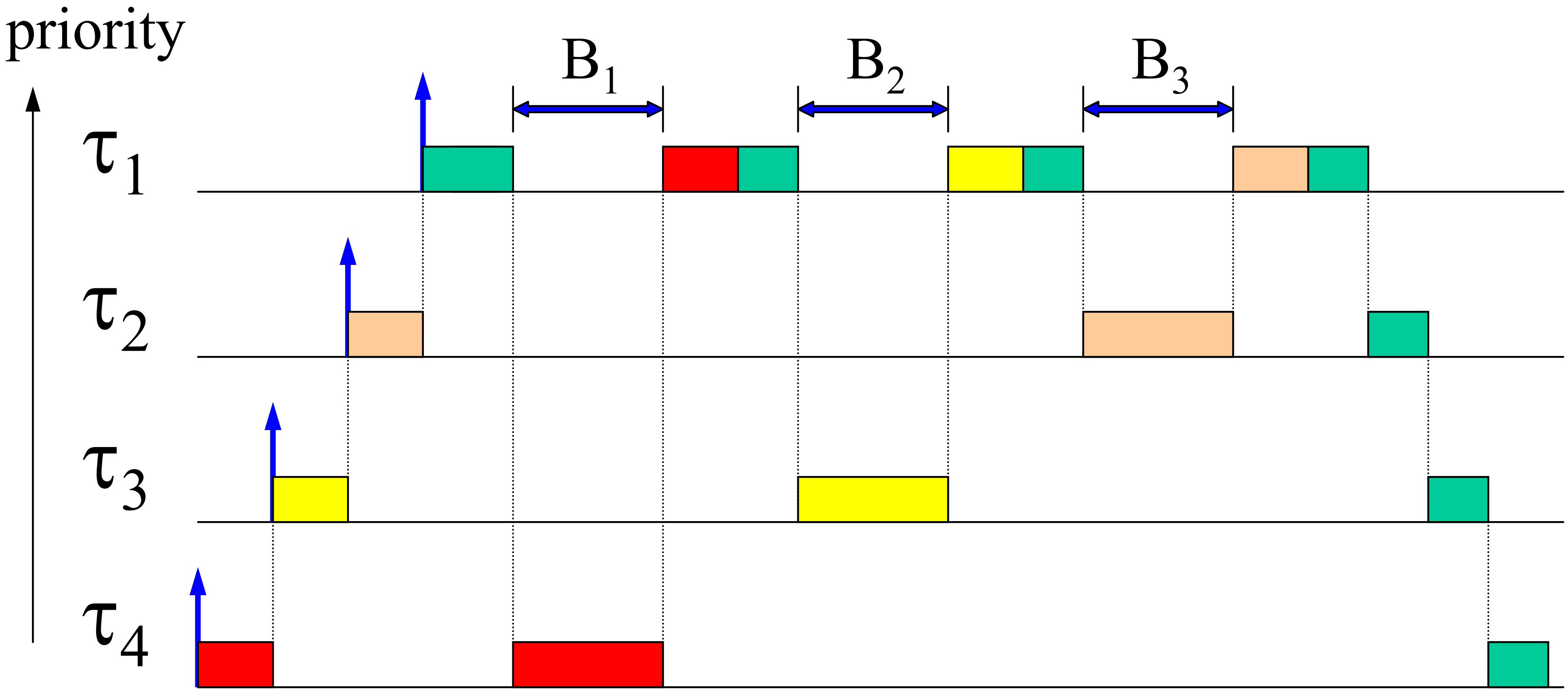
ADVANTAGES

- 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

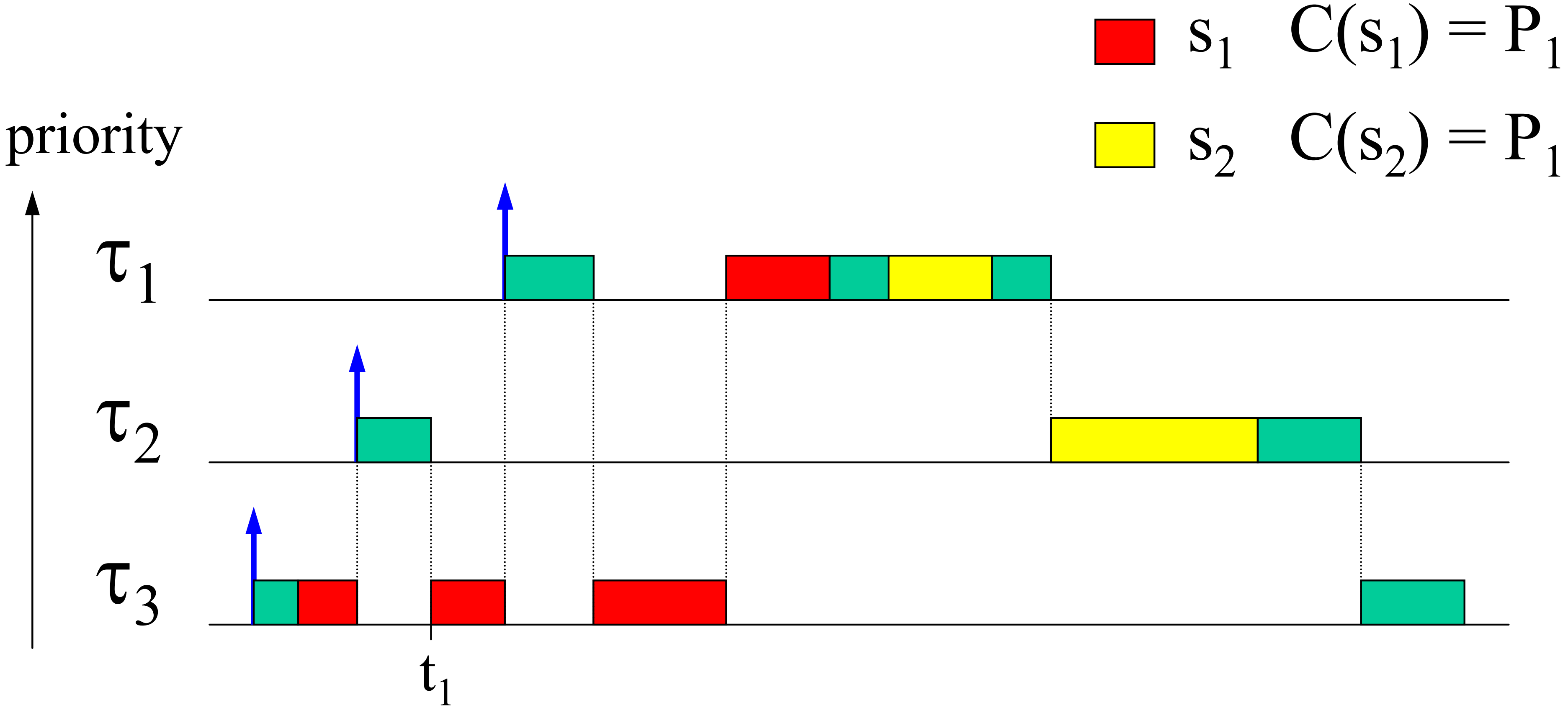
- Each semaphore s_k is assigned a ceiling:

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

- 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 $P_2 < C(s_1)$

PCP properties

Theorem 1

Under PCP, each task can block at most once.

Theorem 2

PCP prevents chained blocking.

Theorem 3

PCP prevents deadlocks.

Remarks on PCP

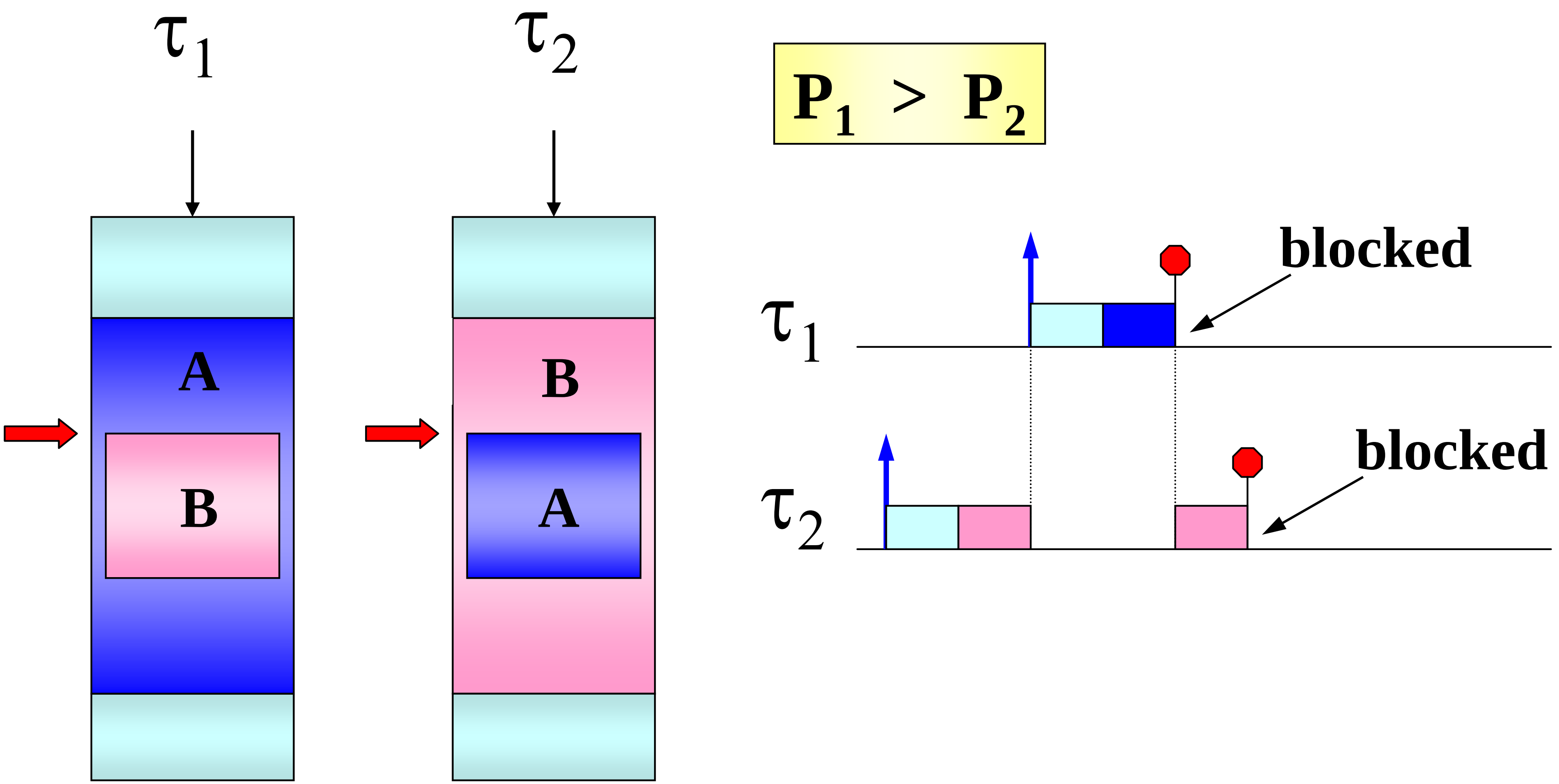
ADVANTAGES

- Blocking is reduced to <u>only one</u> CS
- It prevents deadlocks

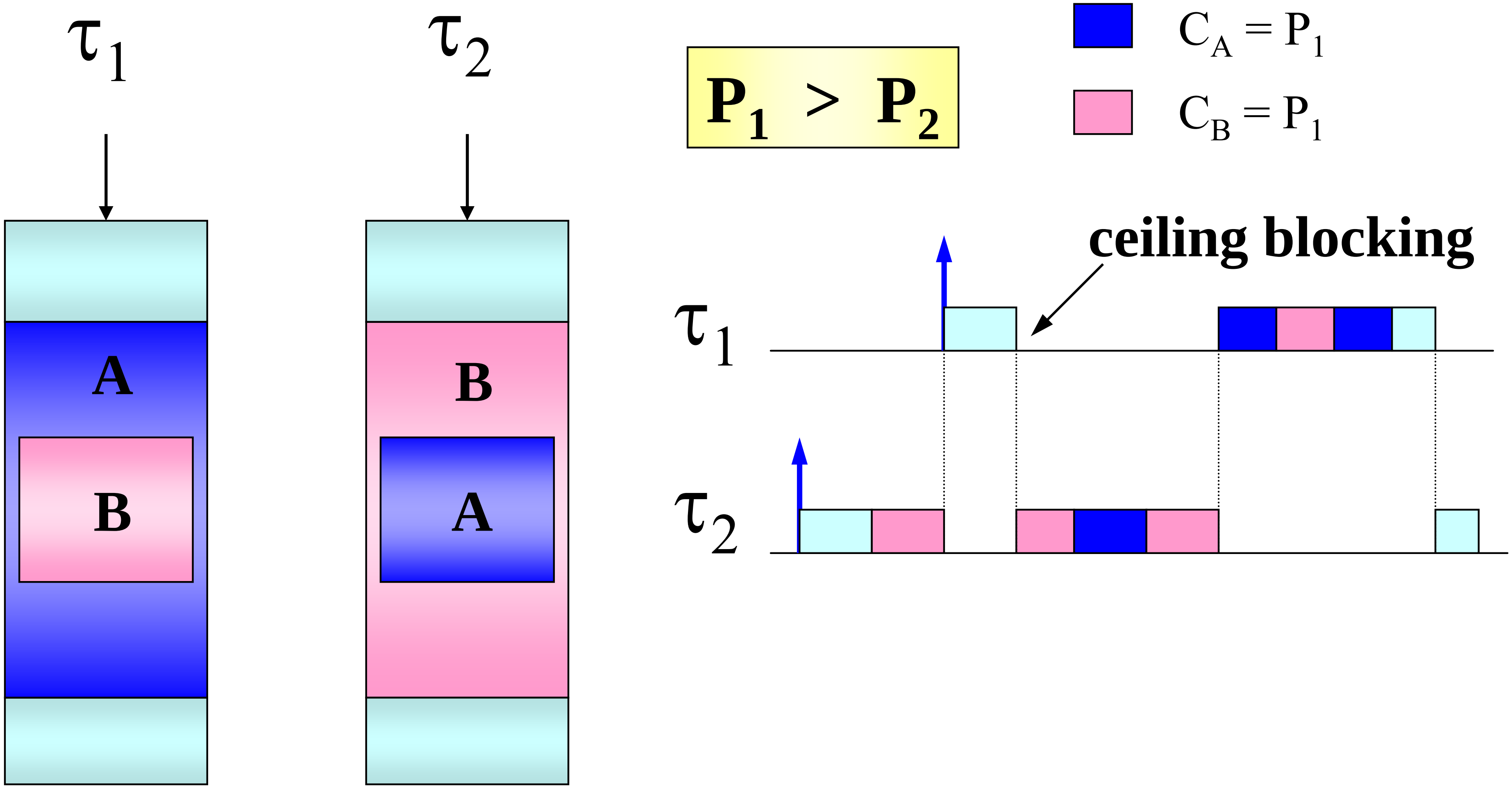
PROBLEMS

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

Typical Deadlock



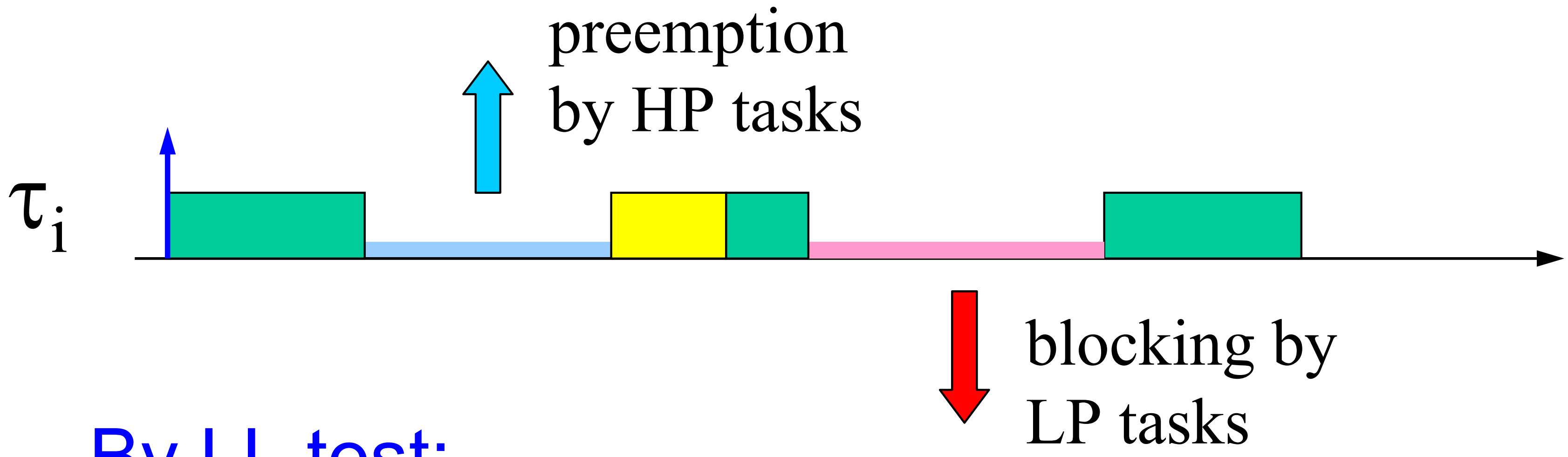
Deadlock avoidance with PCP



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.

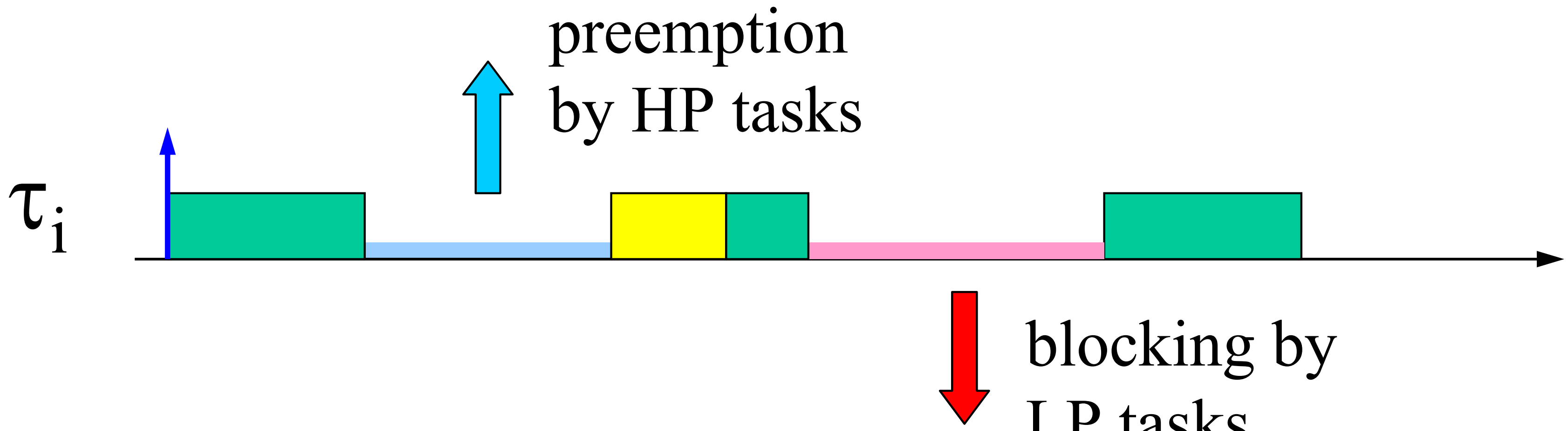
Guarantee with RM (D = T)



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: $\forall i \quad R_i \leq D_i$

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

Resource Sharing under EDF

The protocols analyzed so far have been originally developed for fixed priority scheduling schemes. However:

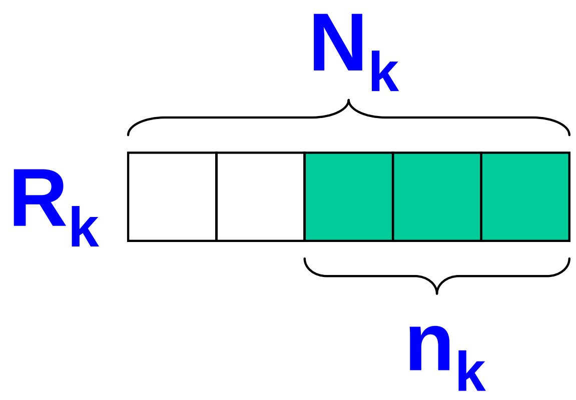
- NPP can also be used under EDF
- PIP has been extended under EDF by Spuri (1997).
- PCP has been extended under EDF by Chen-Lin (1990)
- In 1990, Baker proposed a new protocol that works both under fixed and dynamic priorities.

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 $\mathbf{R_k}$:
  - $\Rightarrow$ Maximum units: $\mathbf{N_k}$
  - $\Rightarrow$ Available units: $\mathbf{n_k}$



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

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

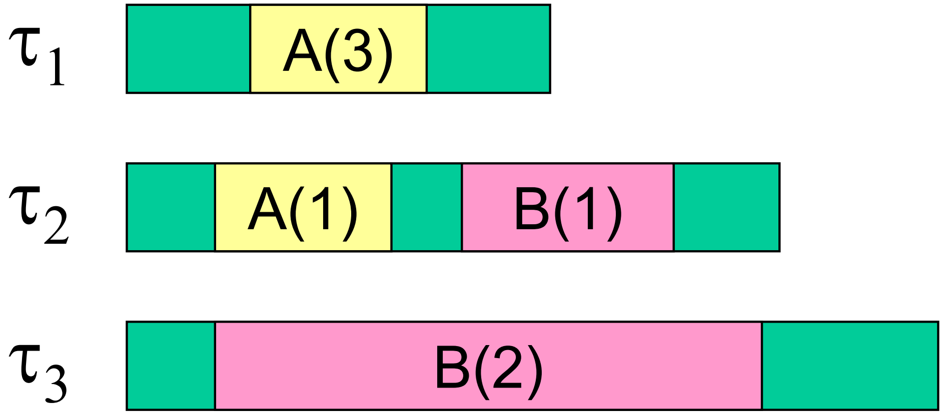
$$\Pi_s = \max_k \{ C_k(n_k) \}$$

SRP Rule

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

Computing Resource Ceilings

	N_R
R_A	3
R_B	2



	D_i	π_i	μ_A	μ_B
τ_1	10	3	3	0
τ_2	15	2	1	1
τ_3	20	1	0	2

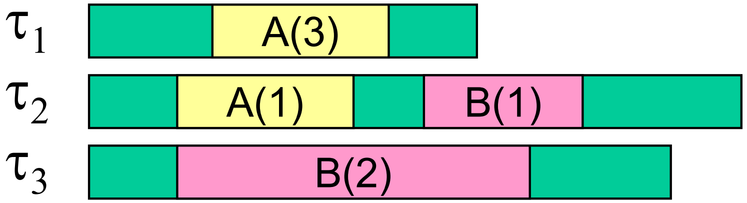
Computing Resource Ceilings

	N_R
R_A	3
R_B	2

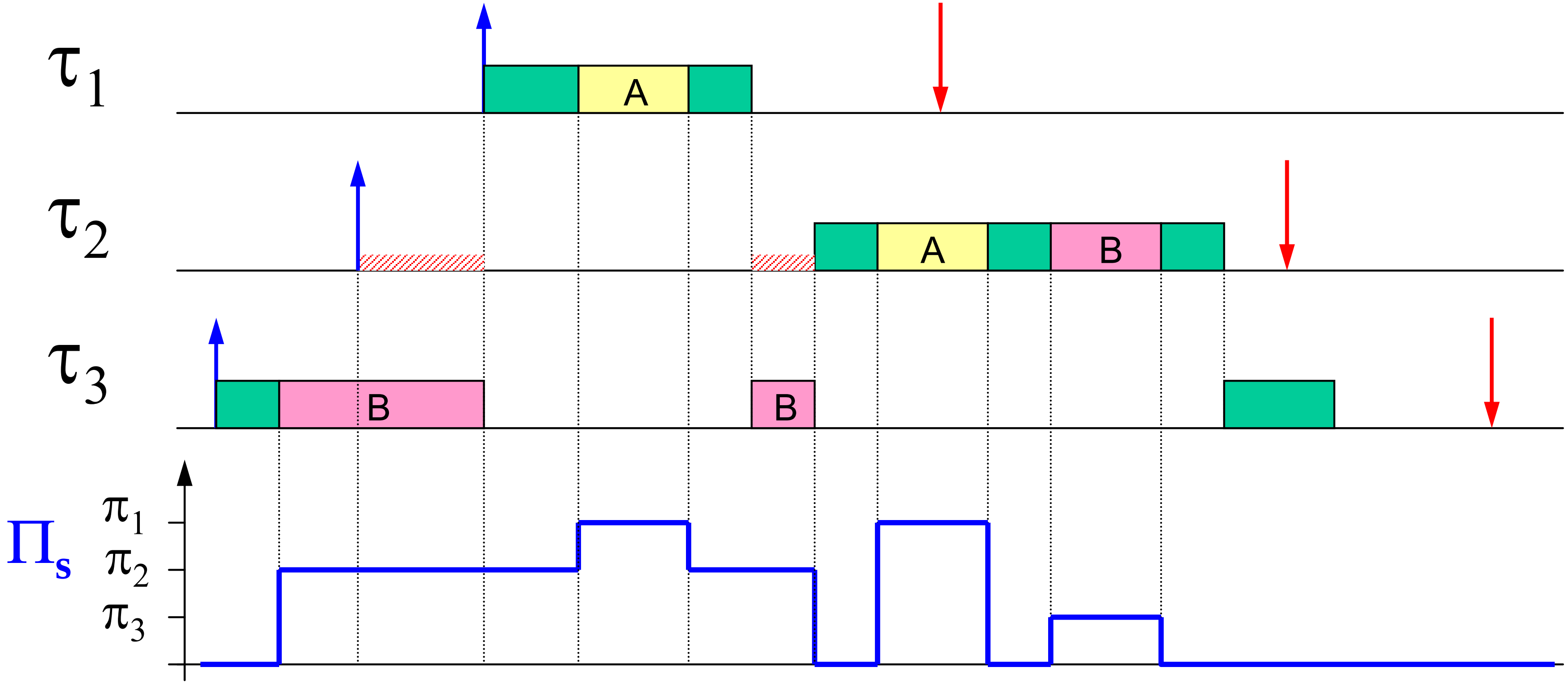
	D_i	π_i	μ_A	μ_B
τ_1	10	3	3	0
τ_2	15	2	1	1
τ_3	20	1	0	2

	$C_R(3)$	$C_R(2)$	$C_R(1)$	$C_R(0)$
R_A	0	3	3	3
R_B	-	0	1	2

Schedule with SRP

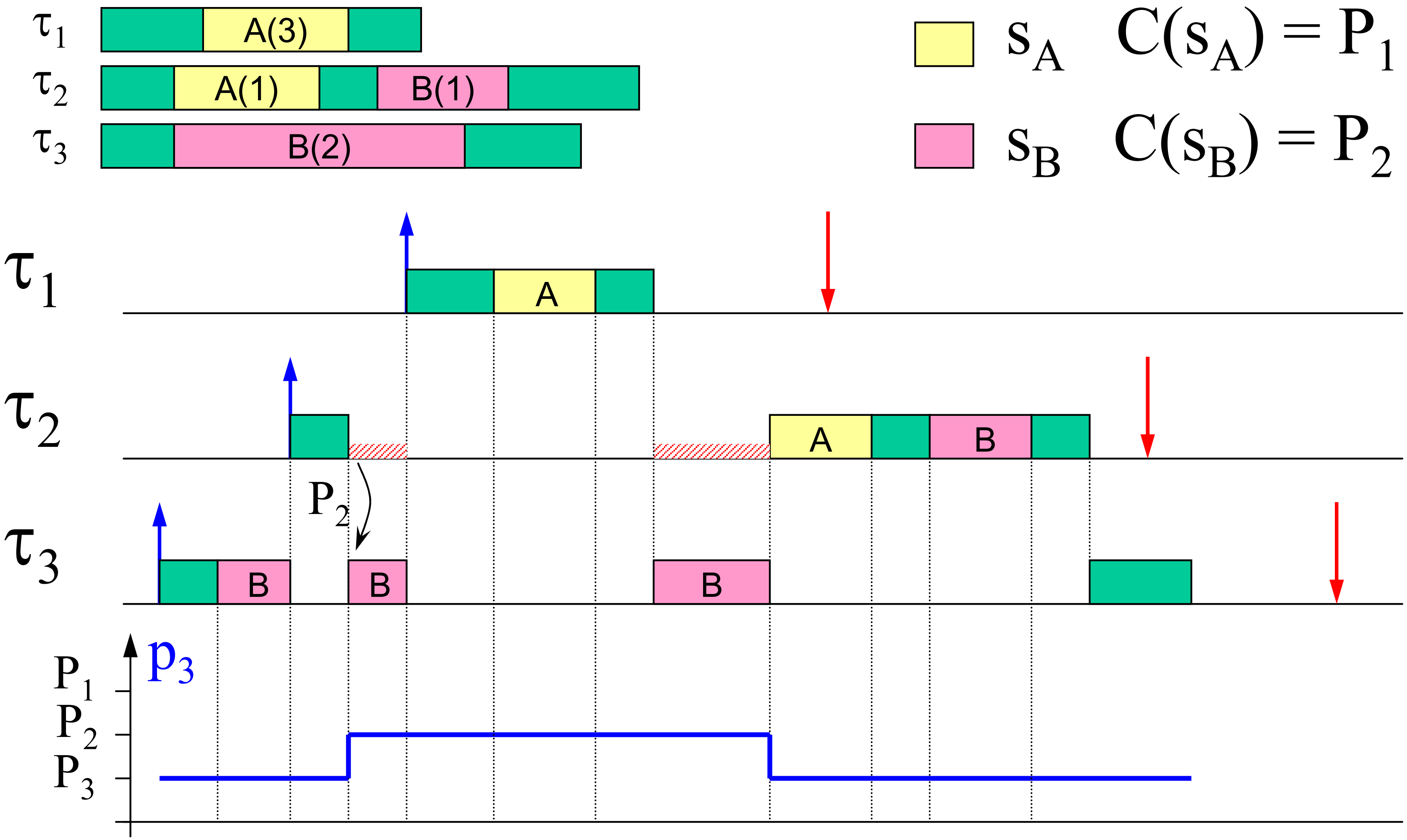


	N_R	$C_R(3)$	$C_R(2)$	$C_R(1)$	$C_R(0)$
R_A	3	0	3	3	3
R_B	2	-	0	1	2



A task blocks when attempting to preempt

Schedule with PCP



A task is blocked when accessing a resource

SRP Properties

Lemma

If $\pi_i > C_R(n_k)$ then there exist enough units of R

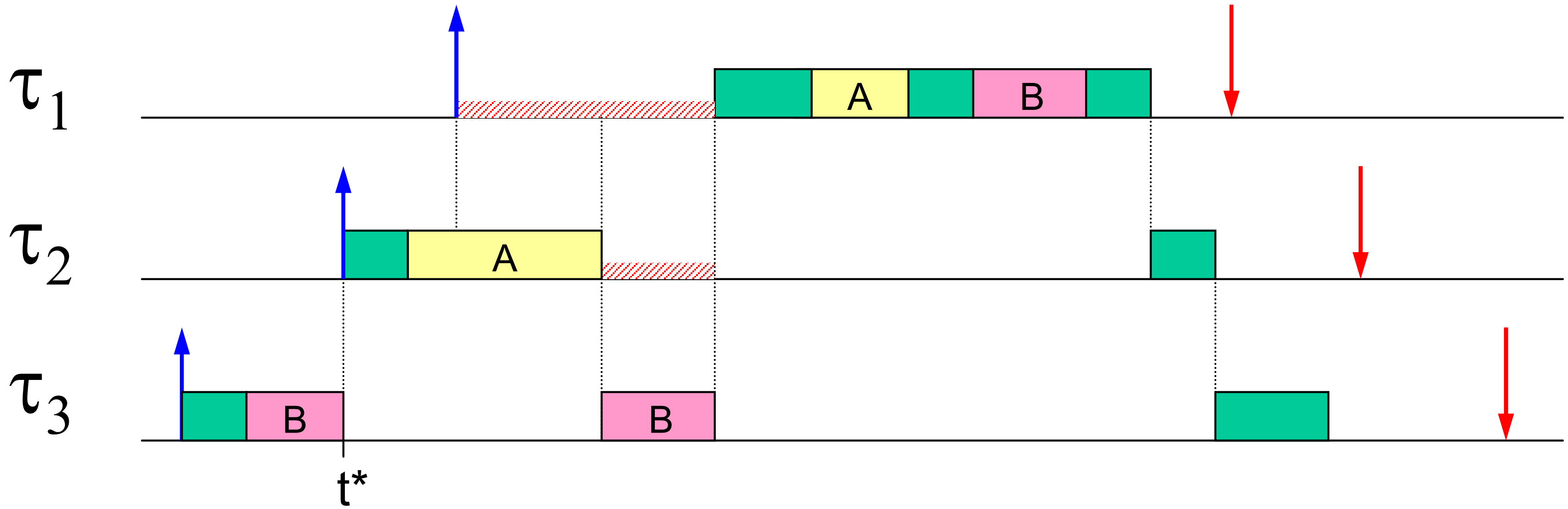
1. to satisfy the requirements of τ_i
2. to satisfy the requirements of all tasks that can make preemption on τ_i

SRP Properties

Theorem 1

Under SRP, each task can block at most once.

Consider the following scenario where τ_1 blocks twice:



This is not possible, because τ_2 could not preempt τ_3 because, at time t*, $\pi_2 < \Pi_s$

SRP Properties

Theorem 2

If $\pi_i > \Pi_s$ then τ_i will never block once started.

Proof

Since $\Pi_s = \max\{C_R(n_k)\}$, then there are enough resources to satisfy the requirements of τ_i and those of all tasks that can preempt τ_i .

Question

If a task can never block once started, can we get rid of the wait / signal primitives?

SRP Properties

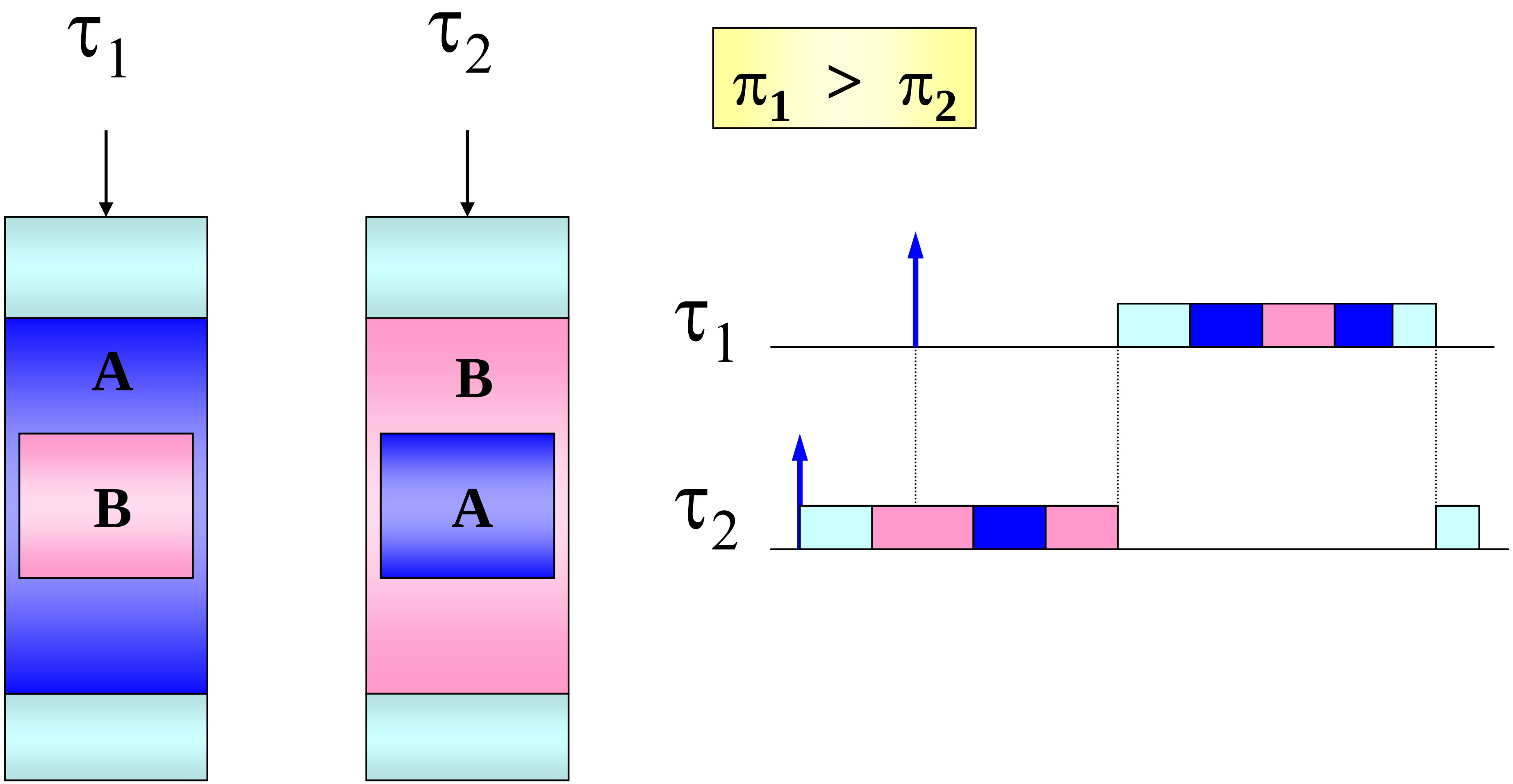
Theorem 3

SRP prevents deadlocks.

Proof

From Theorem 2, if a task can never block once started, then no deadlock can occur.

Deadlock avoidance with SRP



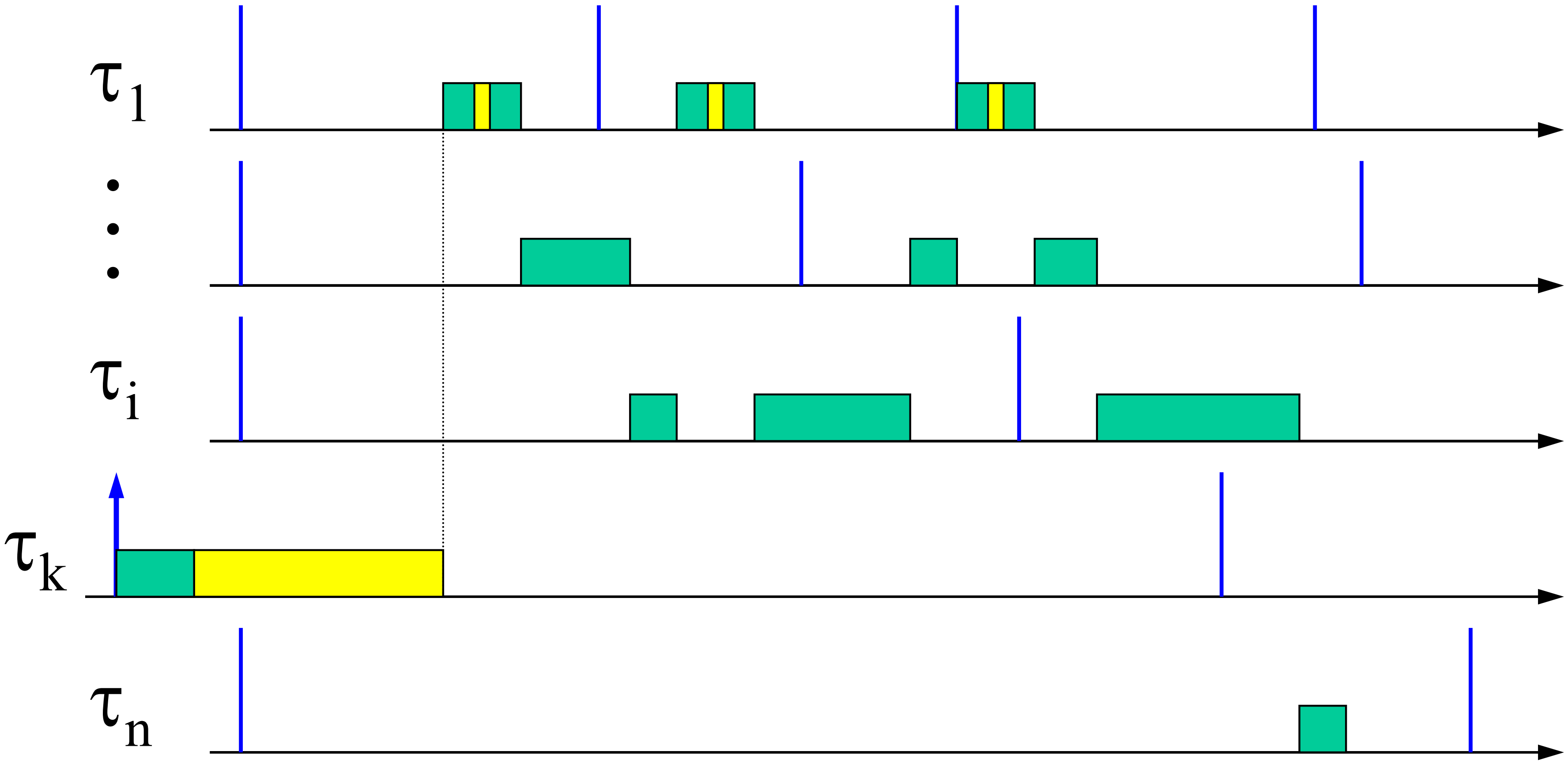
Schedulability Analysis under EDF

When $D_i = T_i$

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

B_i can be computed as under PCP and refers to the length of longest critical section that can block τ_i.

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 $\forall L \in D$

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

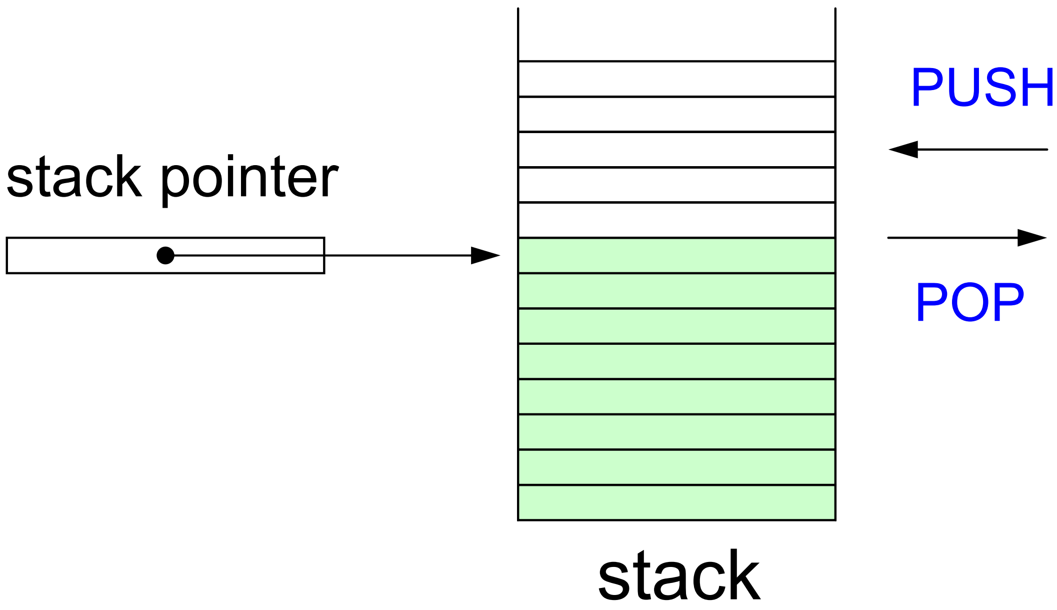
where $D = \{d_k \mid d_k \leq \min(H, L^*)\}$

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

Stack Sharing

Each task normally uses a private stack for

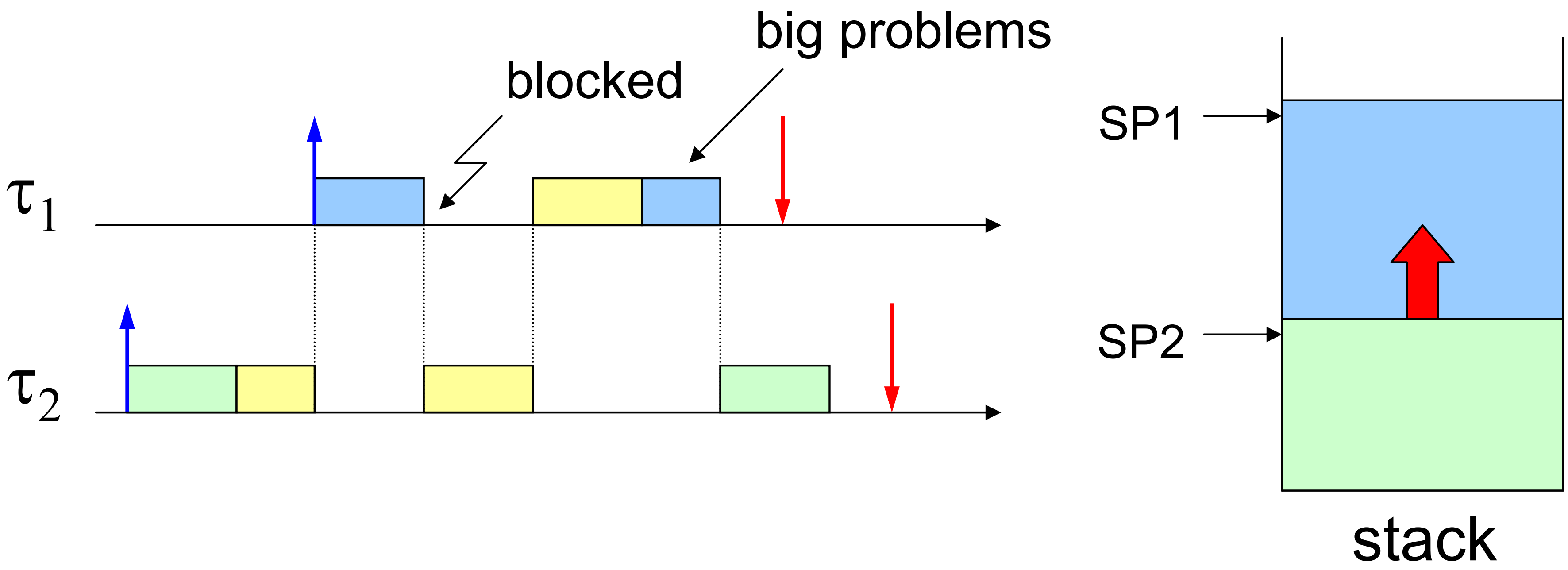
- saving context (register values)
- managing functions
- storing local variables



Stack Sharing

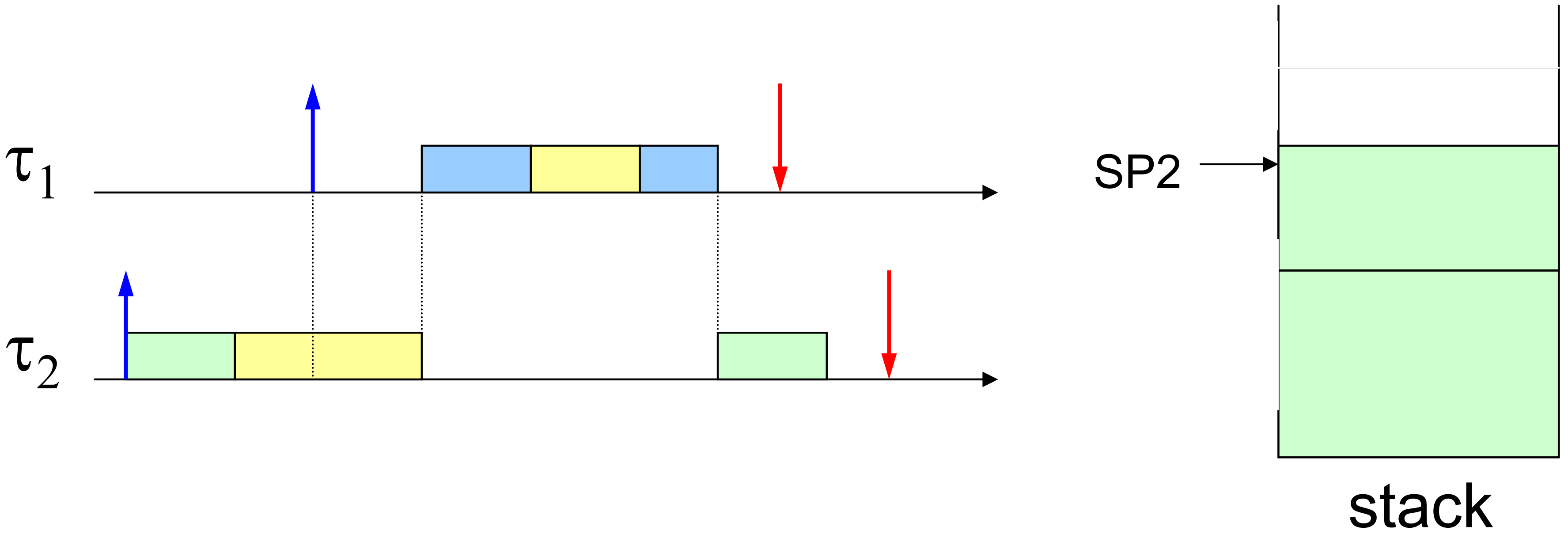
Why stack cannot be normally shared?

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.

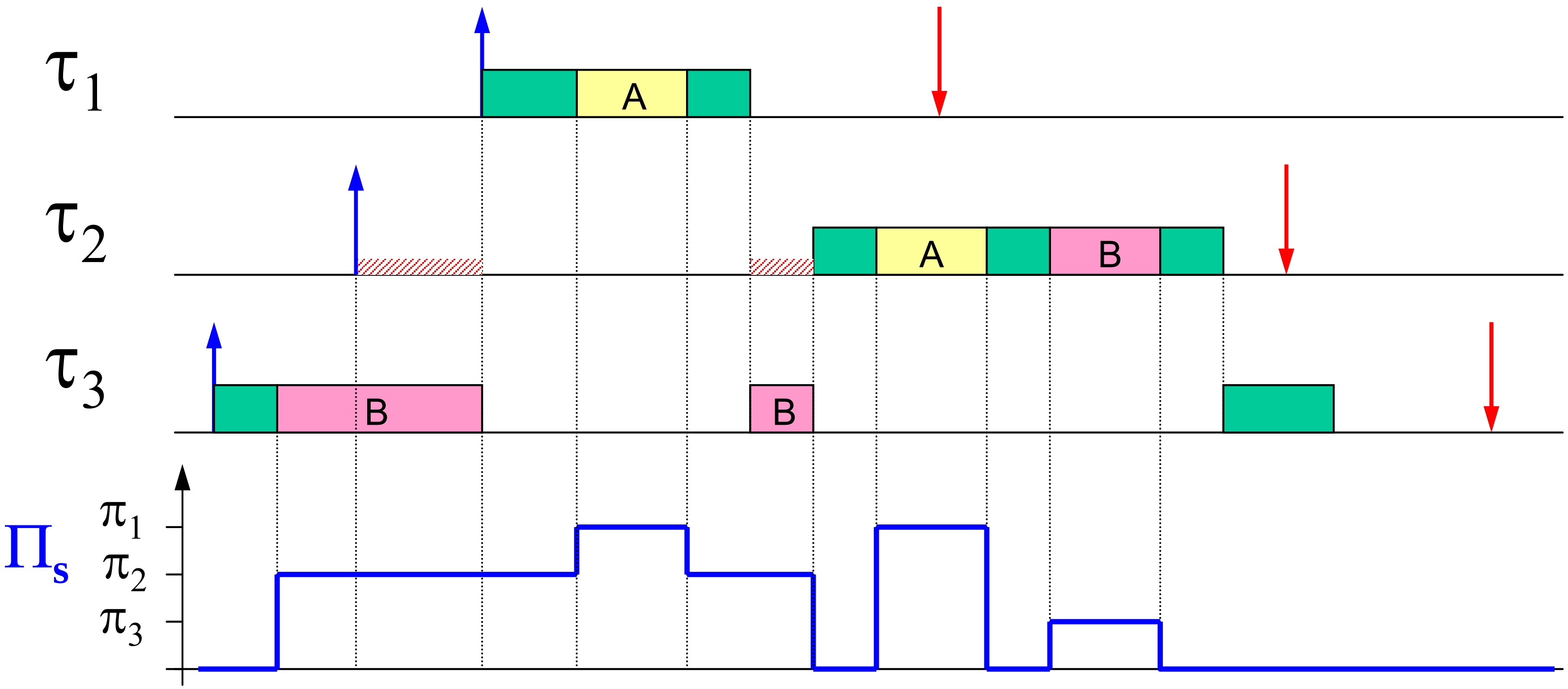
100 tasks
10 Kb stack per task
→ stack size = 1 Mb

10 preemption levels
10 tasks per group
→ stack size = 100 Kb

stack saving = 90 %

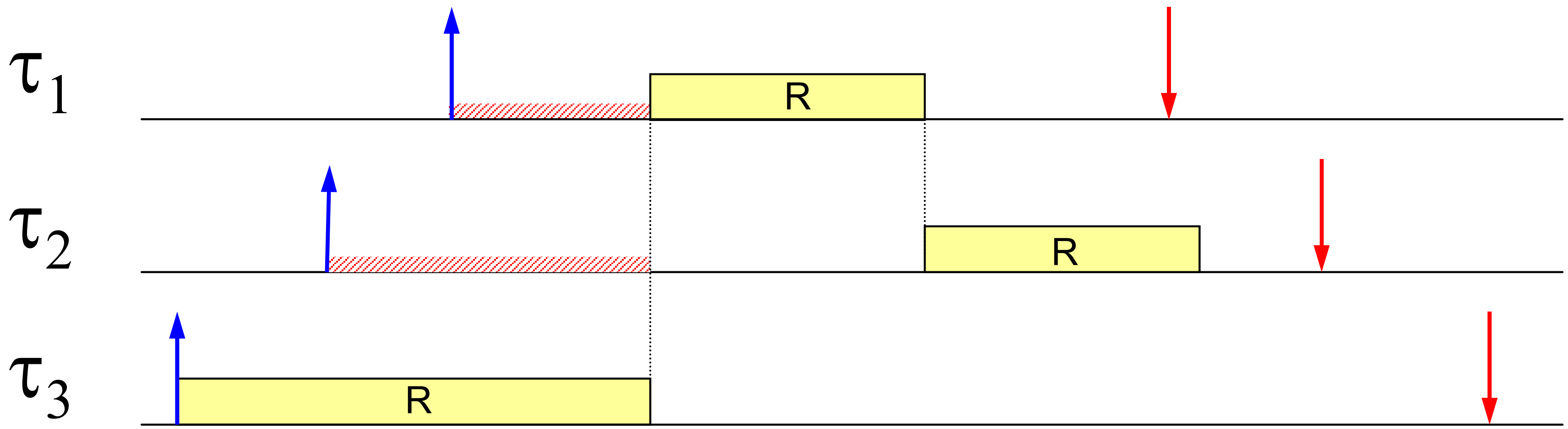
NOTE on SRP

- 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:

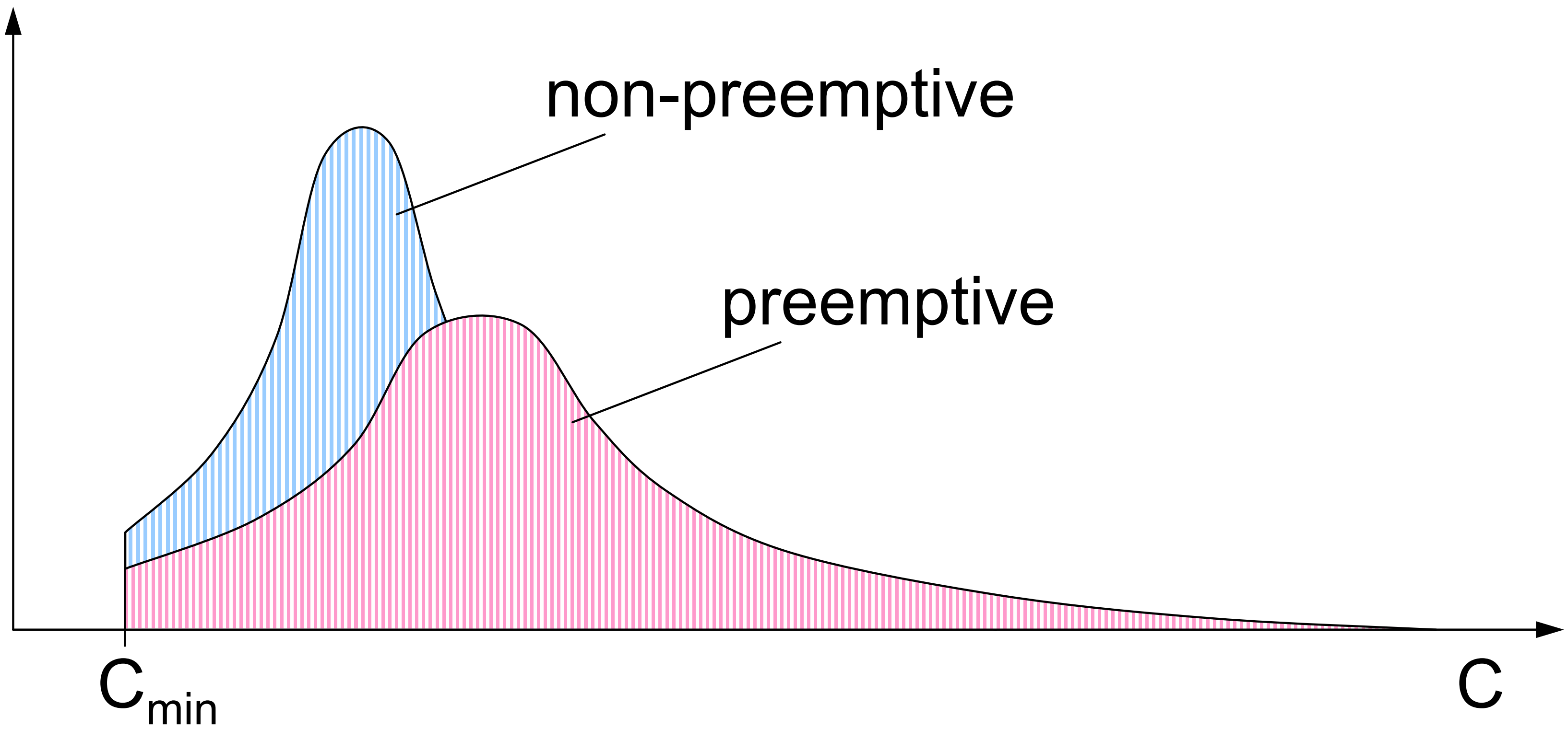
$$B_i = \max\{C_k : P_k < P_i\}$$

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

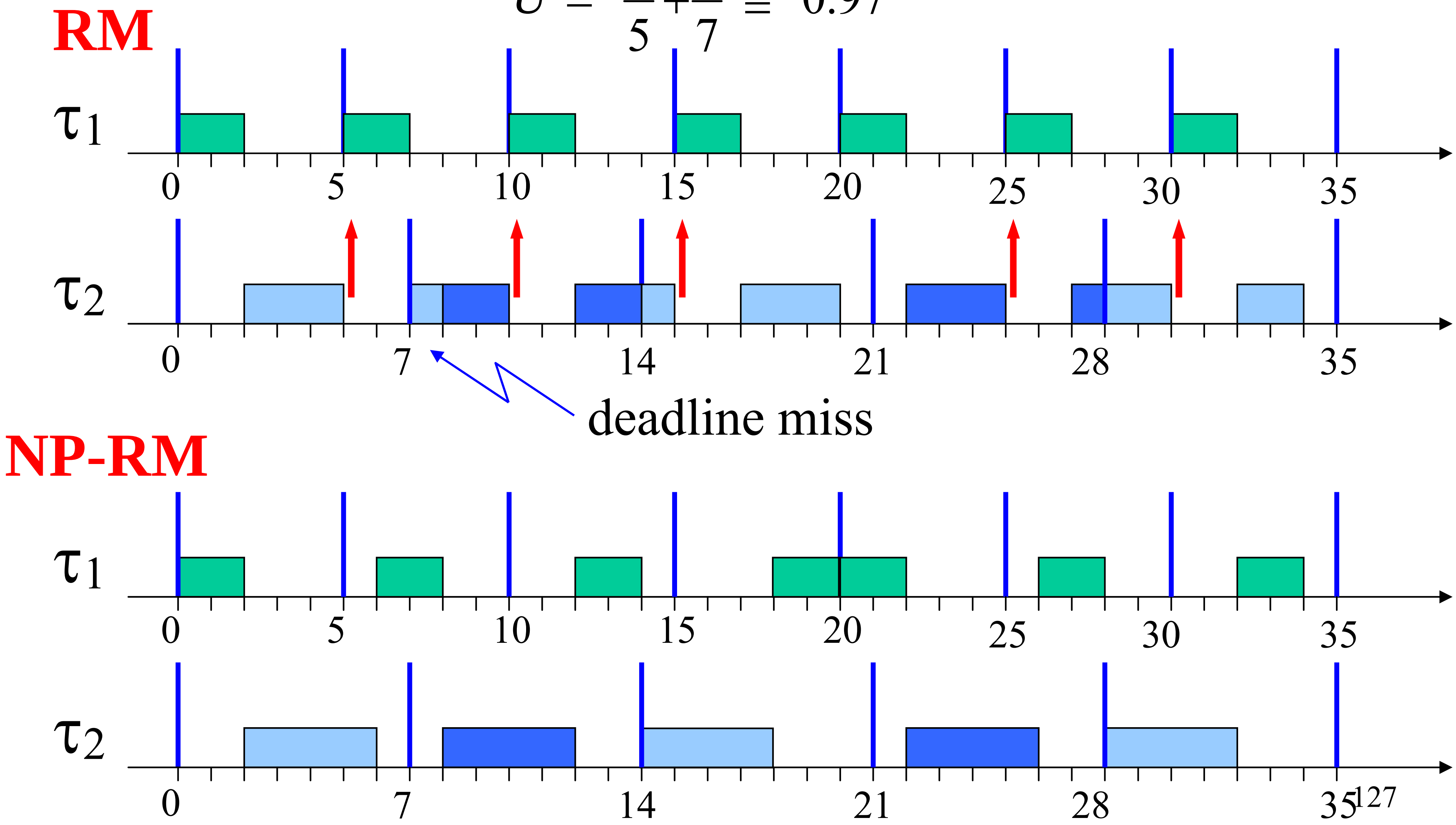
- As a consequence, task execution times are
  - Smaller
  - More predictable



Advantages of NP scheduling

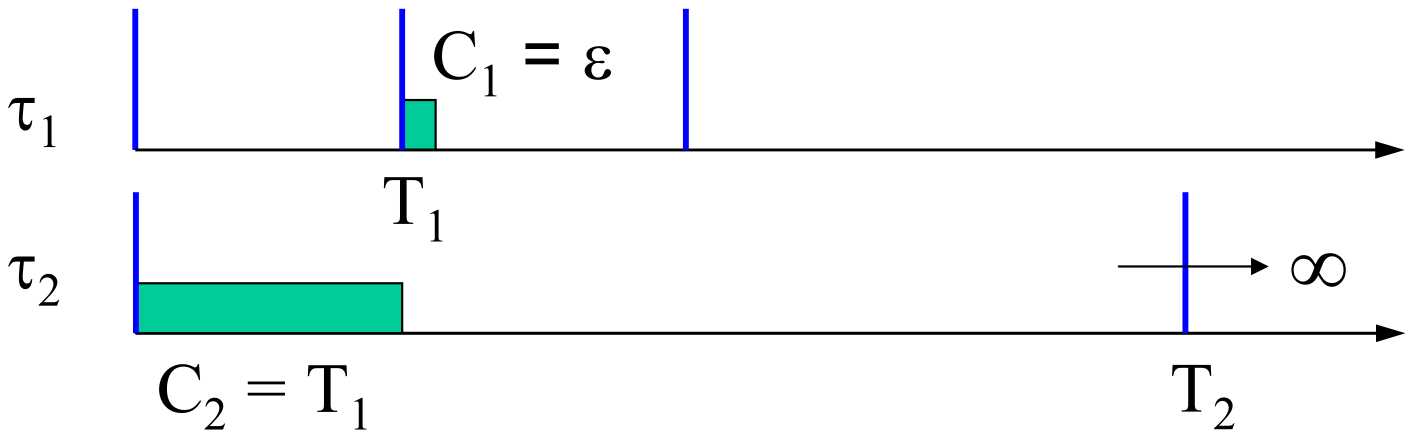
In fixed priority systems can improve schedulabiilty:

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



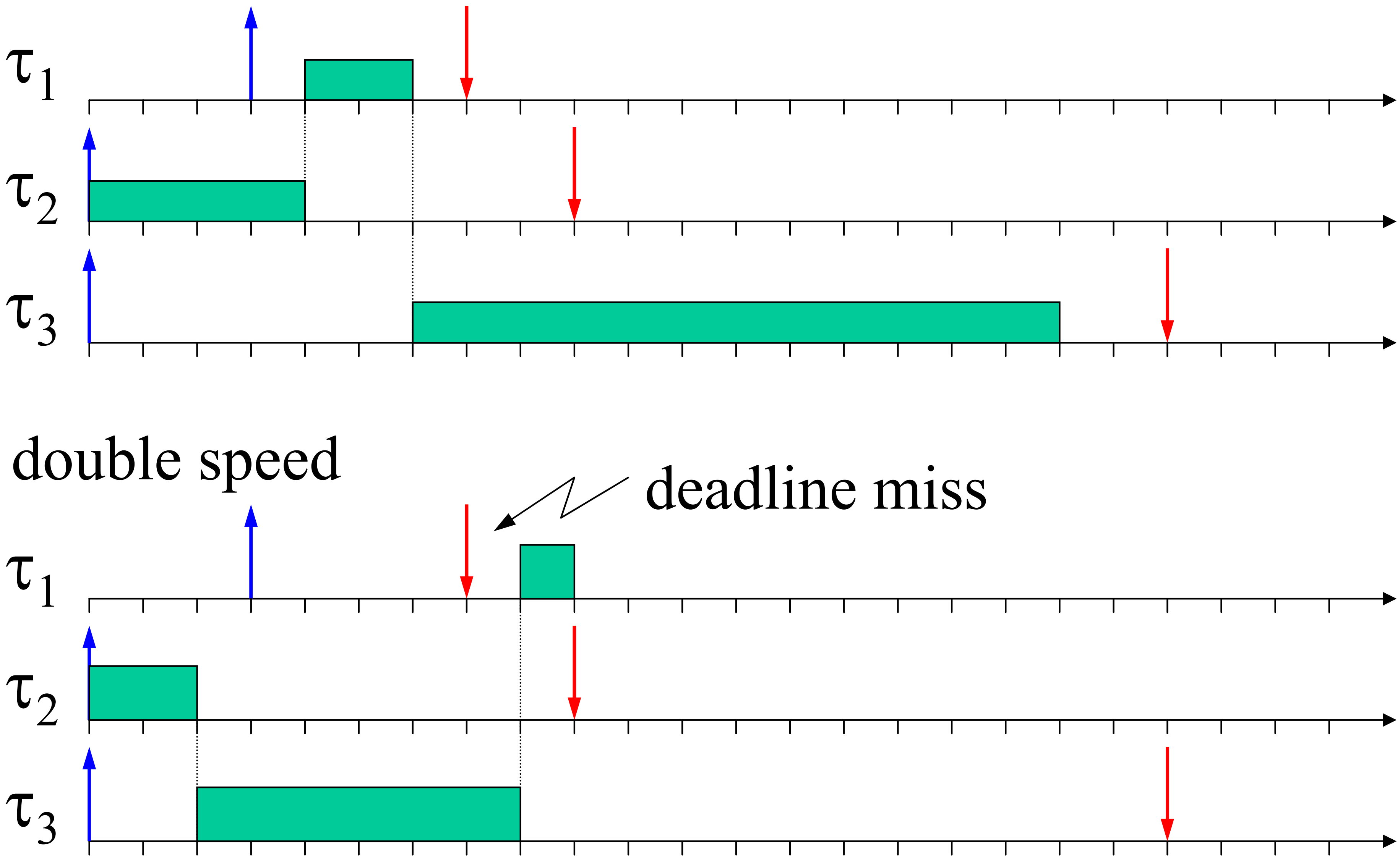
Disadvantages of NP scheduling

- 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



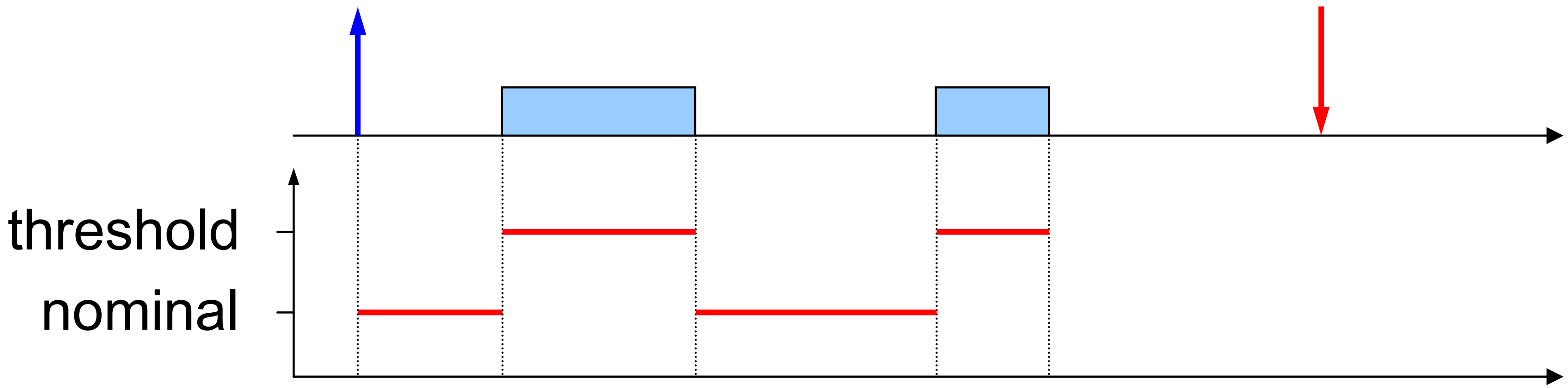
Trade-off solutions

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

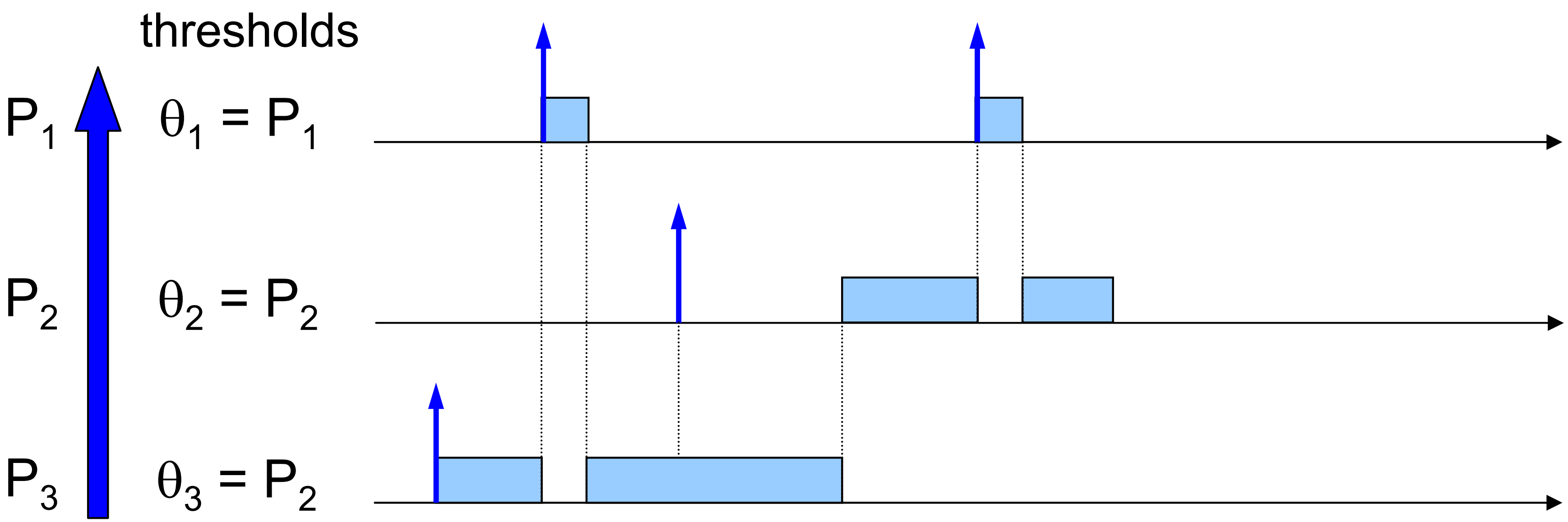
nominal priority $\leq$ threshold priority



Preemption thresholds

- Nominal pr. = threshold: $\Rightarrow$ fully preemptive
- Threshold = P_{max} $\Rightarrow$ fully non preemptive

In general:



Trade-off solutions

Tunable Preemptive Systems

- Compute the longest non-preemptive section that allows a feasible schedule.
- Allow preemption only in certain points in the code.