Scheduling
Hybrid Task Sets

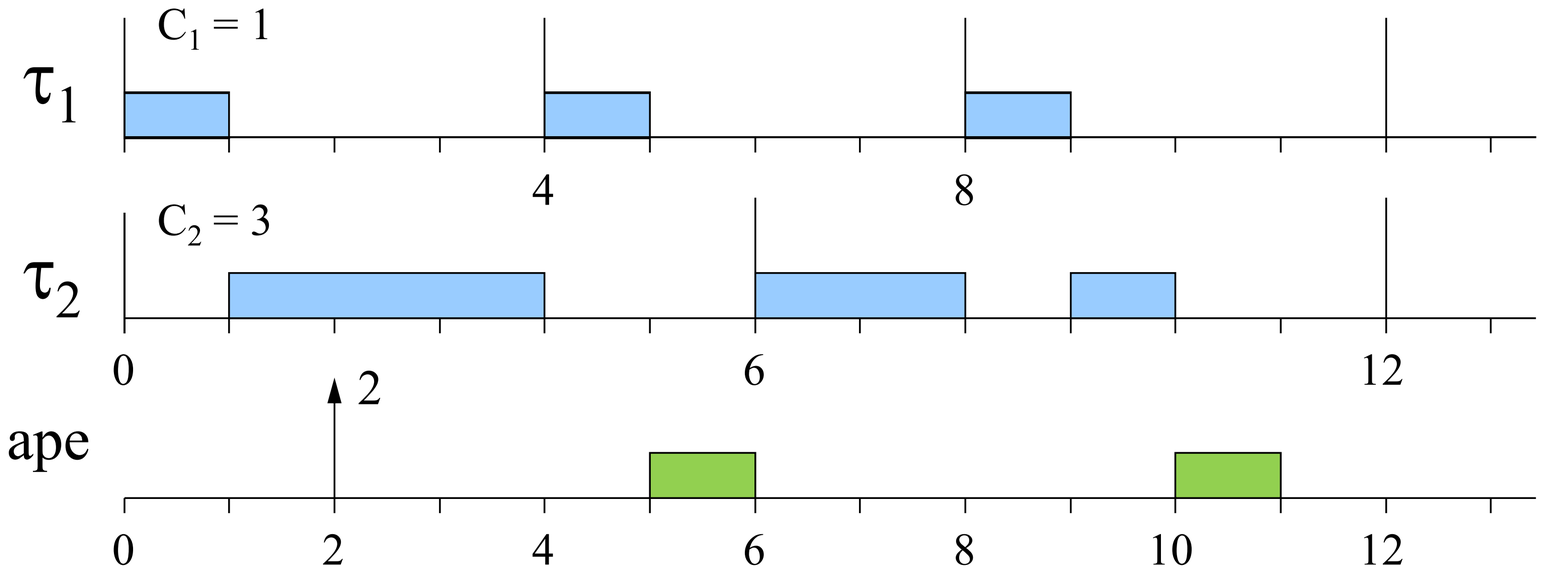
Periodic Tasks + Aperiodic Tasks

Handling aperiodic tasks

- Aperiodic tasks are typically activated by the arrival of external events (notified by interrupts).
- From one hand, one objective of the kernel is to reduce the response time of aperiodic tasks (interrupt latency).
- On the other hand, aperiodic task execution should not jeopardize schedulability.

Background service

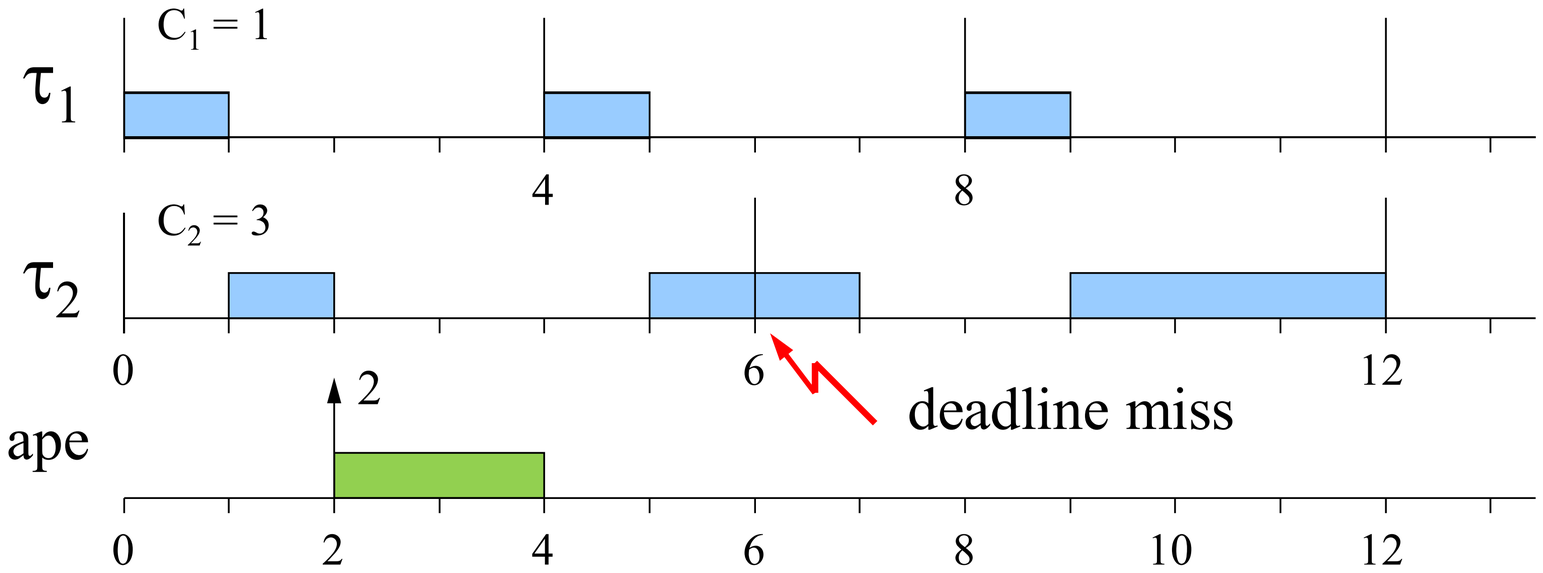
If aperiodic jobs are scheduled in background (i.e., during idle times left by periodic tasks) their response times are too long:



Response Time = 9

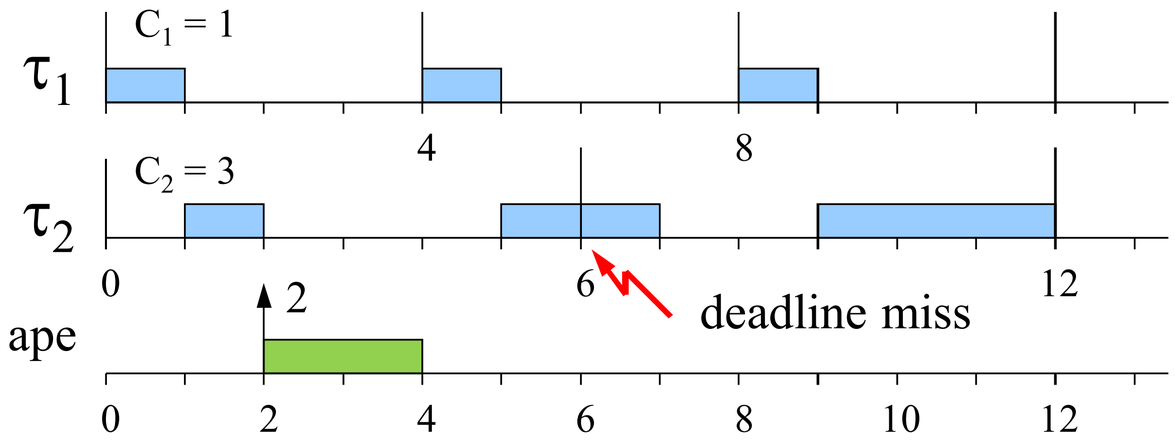
Immediate service

On the other hand, if interrupts service routines are scheduled at the highest priority, the other tasks can miss their deadlines:



Response Time = 2

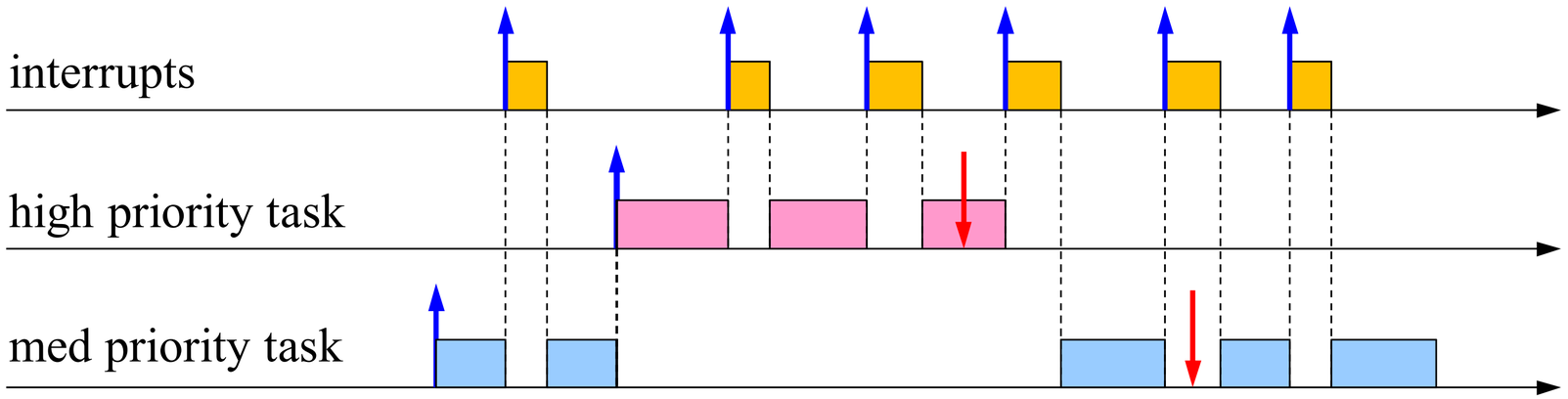
Immediate service



A problem related to a wrong service of aperiodic tasks occurred in the **Apollo 11** computing system, during the historical mission on the Moon, in the landing phase.

What happened exactly?

- During landing, sensors generated several interrupts.
- Intensive execution of drivers caused transient overload conditions.
- Task execution was delayed too much causing several timeout errors (deadline misses).



HARD aperiodic tasks

- 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.

WCET = Worst-Case Execution Time
MIT = Minimum Interarrival Time

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 response time of each aperiodic request
  - → performing an on-line guarantee

How can we achieve these goals?

Aperiodic Servers

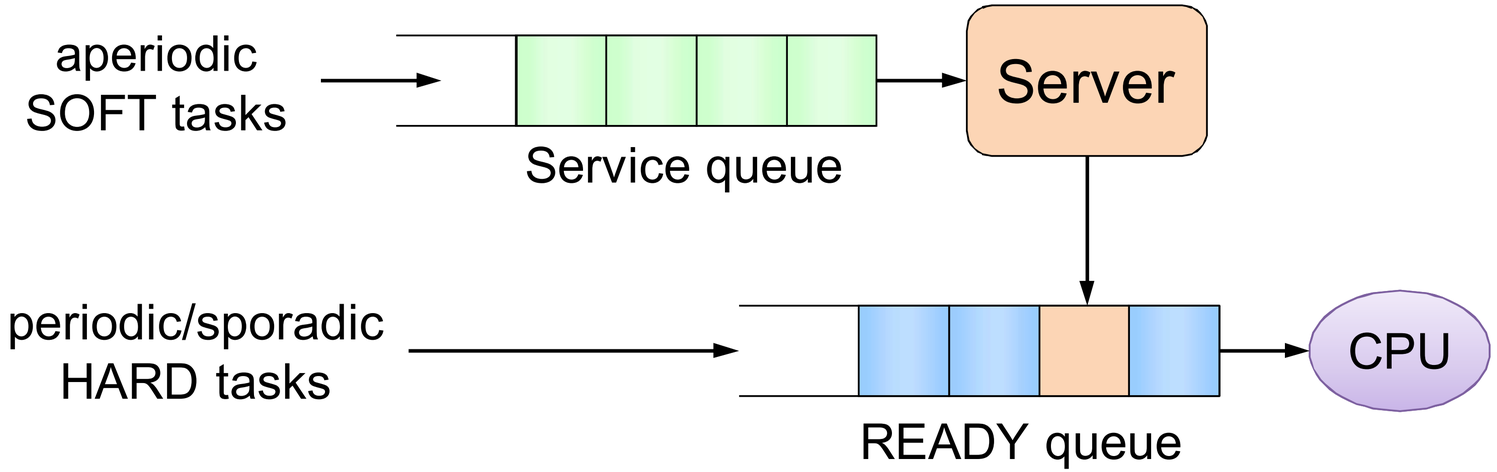
A method to limit the interference of aperiodic tasks on periodic tasks is to manage them through an aperiodic server.

- An aperiodic server can be view as a periodic task managing the execution of aperiodic tasks.
- It is typically described by 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.

Aperiodic Servers

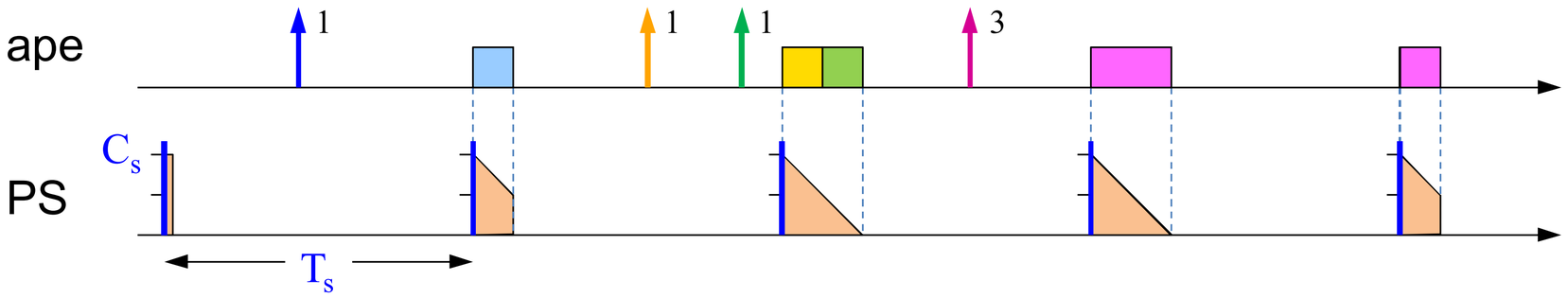
Fixed-priority servers

- 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

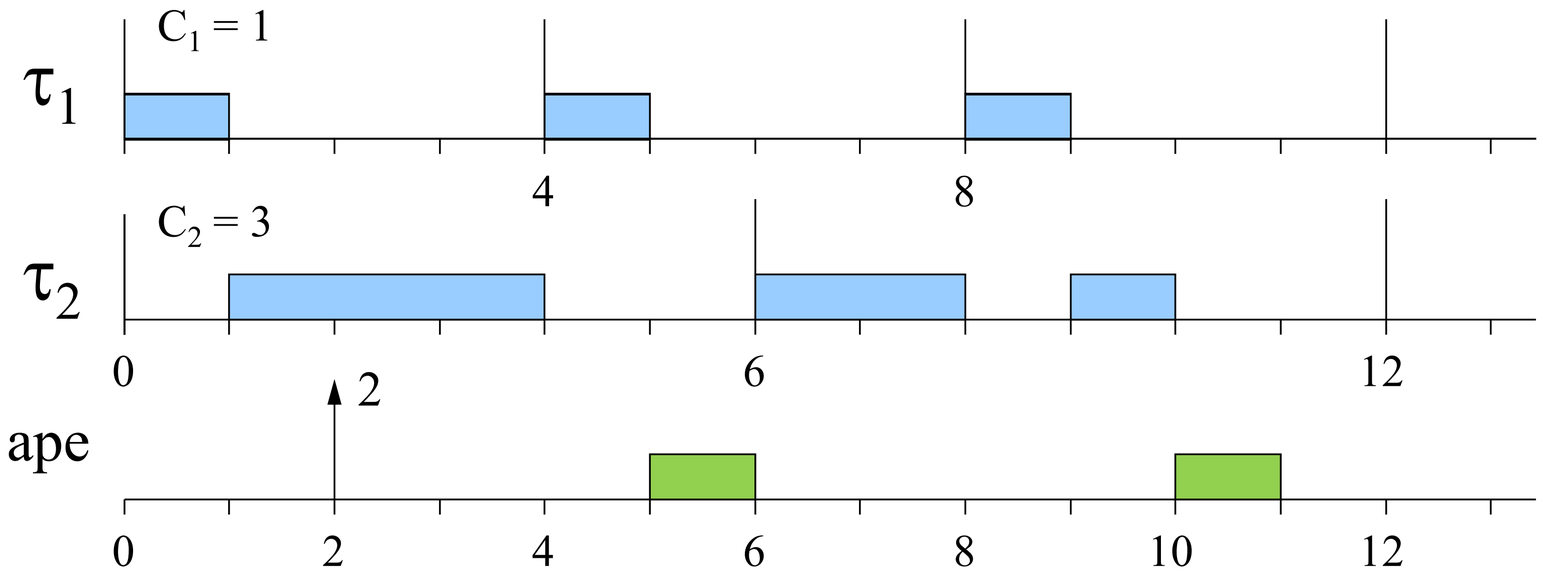
Polling Server (PS)



- At the beginning of each period, the budget is recharged at its maximum value.
- When the server is scheduled and there are no pending jobs, C_s is discharged to zero.
- When the server is scheduled and there are pending jobs, they are served as long as $C_s > 0$ (C_s is consumed proportionally).
- When the server budget is completely consumed ($C_s = 0$) the server is suspended until the beginning of the next period.

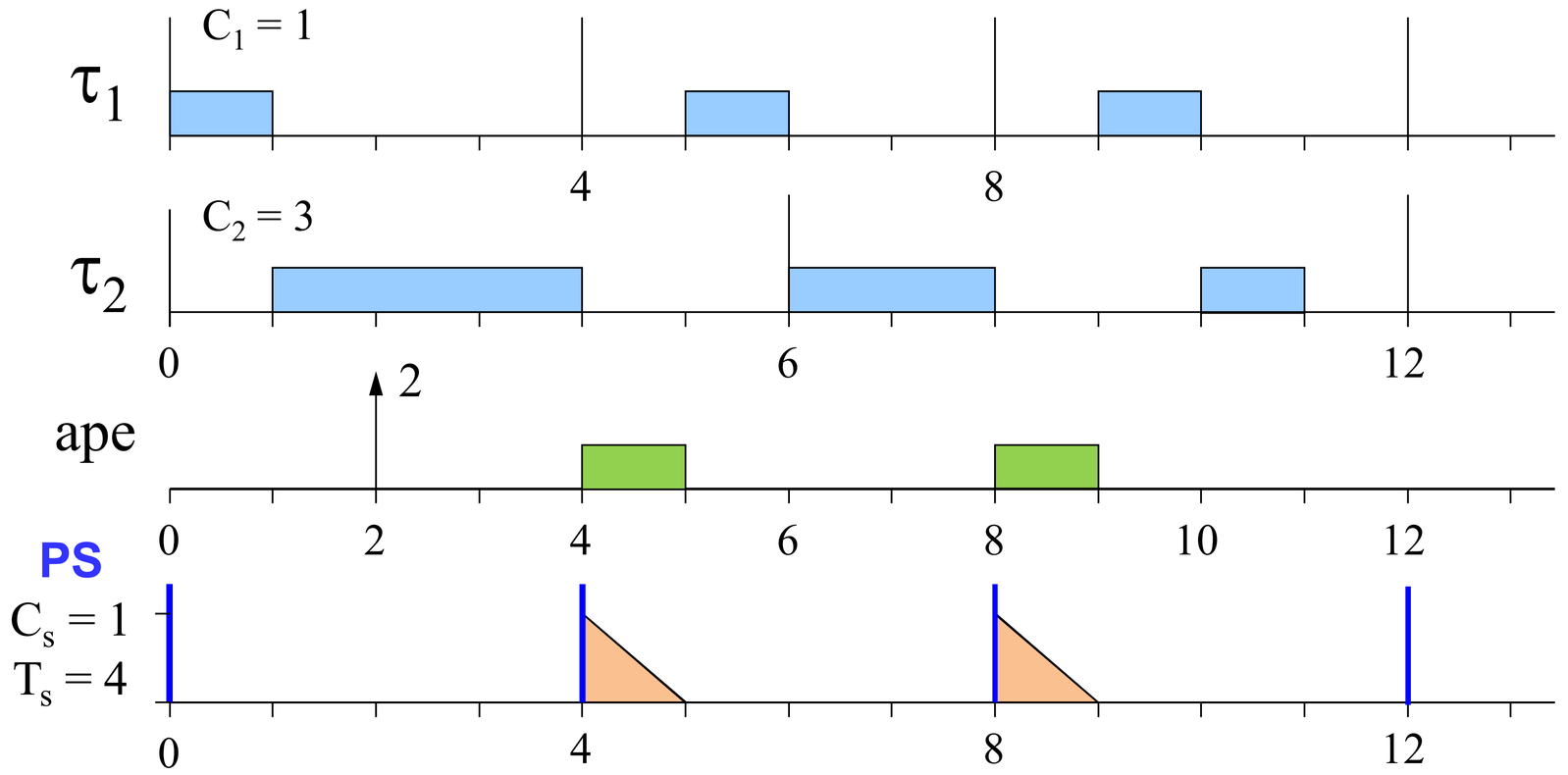
Background service

Let’s take the initial example:



Response Time = 9

RM + Polling Server



Response Time = 7

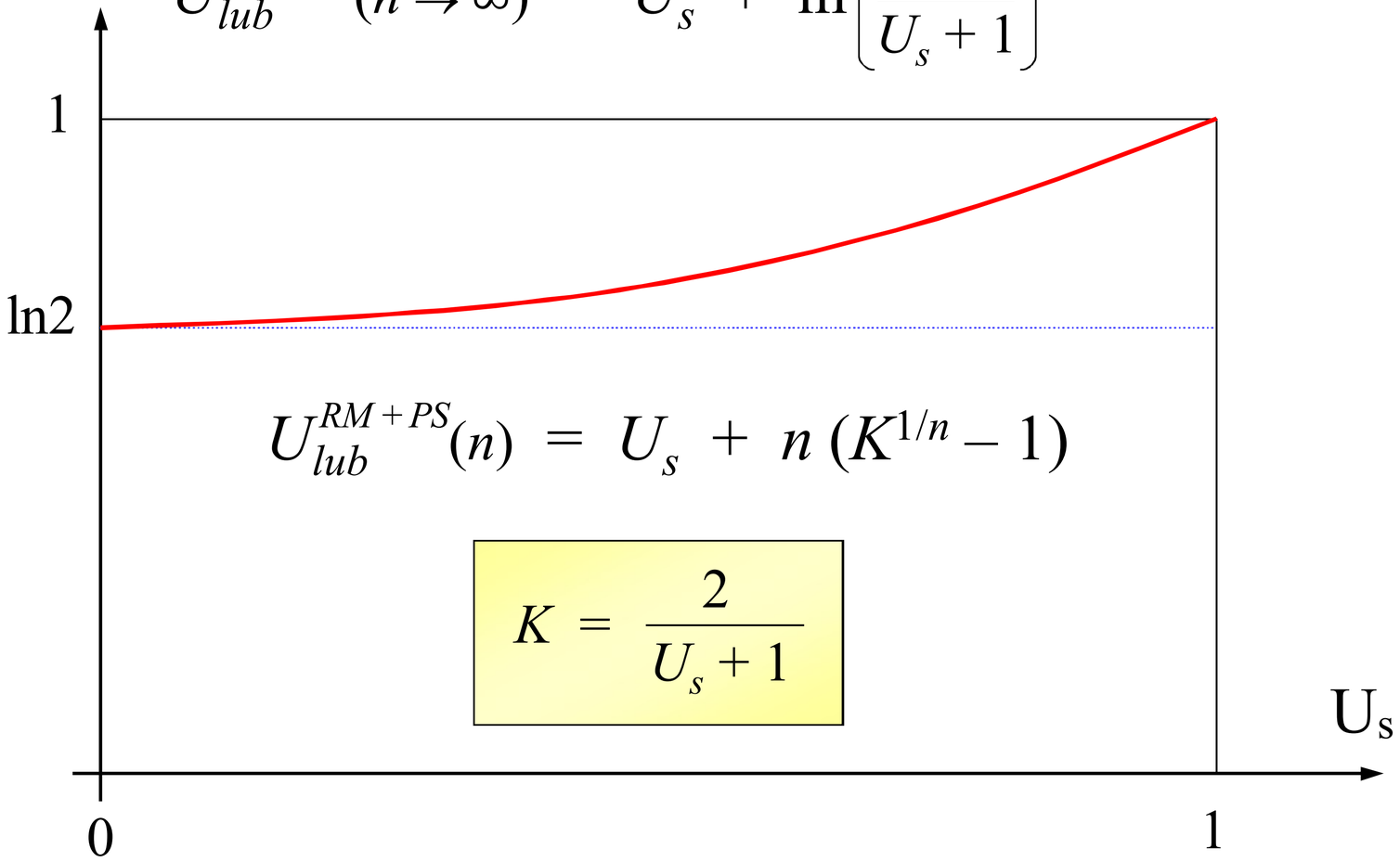
PS properties

- 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:

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

RM+PS schedulability

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



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

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

Analysis with Hyperbolic Bound

A set of periodic tasks is schedulable by Rate Monotonic in the presence of a Polling Server with utilization U_s if

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

Defining $P = \prod_{i=1}^{n}(U_i+1)$

the maximum server utilization that guarantees the schedulability of the periodic task set is

$$U_s^{\max} = \frac{2}{P} - 1$$

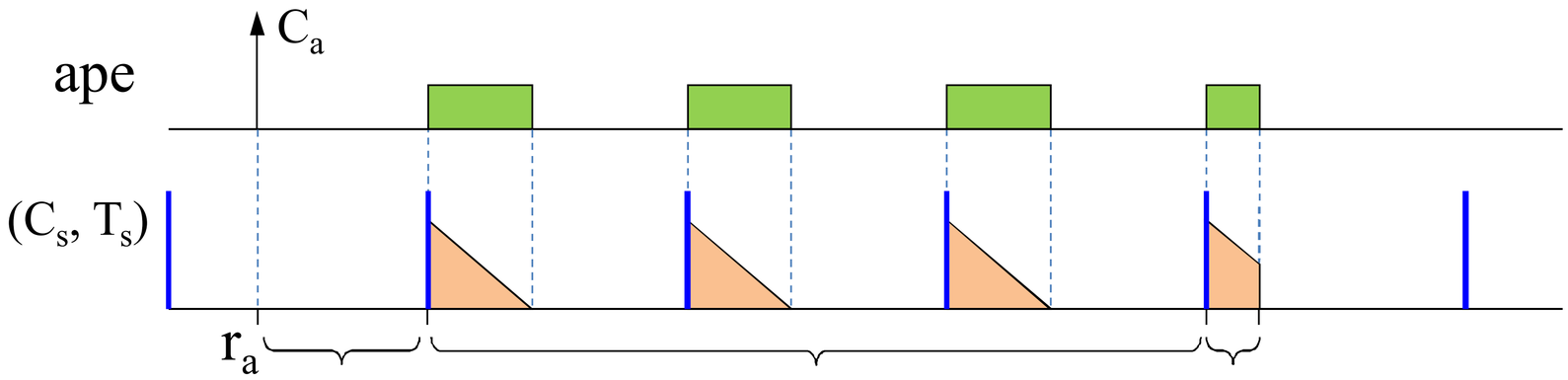
RTA of hard periodic tasks with PS

And the response time of a periodic task can be computed as follows:

$$R_i^{(s)} = C_i + \underbrace{\left\lceil \frac{R_i^{(s-1)}}{T_s} \right\rceil C_s}_{\text{Interference of the server}} + \underbrace{\sum_{k=1}^{i-1} \left\lceil \frac{R_i^{(s-1)}}{T_k} \right\rceil C_k}_{\text{Interference of the hp tasks}}$$

Response time of sporadic tasks under PS

Consider a PS running at the highest priority and an aperiodic job arriving when the server is idle:



initial delay

$$\Delta_a = \left\lceil \frac{r_a}{T_s} \right\rceil T_s - r_a$$

full service periods

$$F_a = \left\lceil \frac{C_a}{C_s} \right\rceil - 1$$

final chunk

$$\delta_a = C_a - F_a C_s$$

$$R_a = \Delta_a + C_a + F_a (T_s - C_s)$$

Complete RM tests

Liu & Layland test:

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

Hyperbolic test:

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

Response Time:

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

Designing server parameters

1. Determine U_s^{max} using $\prod_{i=1}^{n}(U_i + 1) \leq K$

   Defining $P = \prod_{i=1}^{n}(U_i + 1)$ we have $\begin{cases} U_{PS}^{max} = \frac{2}{P} - 1 \\ U_{DS}^{max} = \frac{2 - P}{2P - 1} \end{cases}$

2. Set $U_s \leq U_s^{max}$

3. Set $T_s = \min(T_1, \ldots, T_n)$

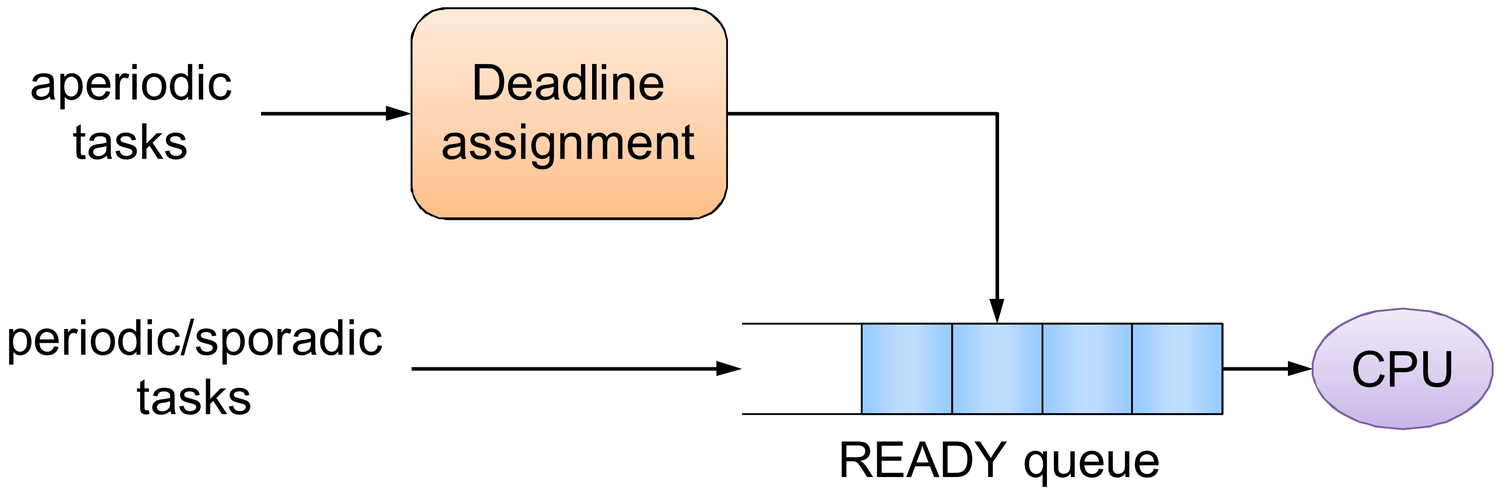
4. Set $C_s = U_s T_s$

Dynamic Priority Servers

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*

$$U_p + U_s \leq 1$$

Deadline assignment rule

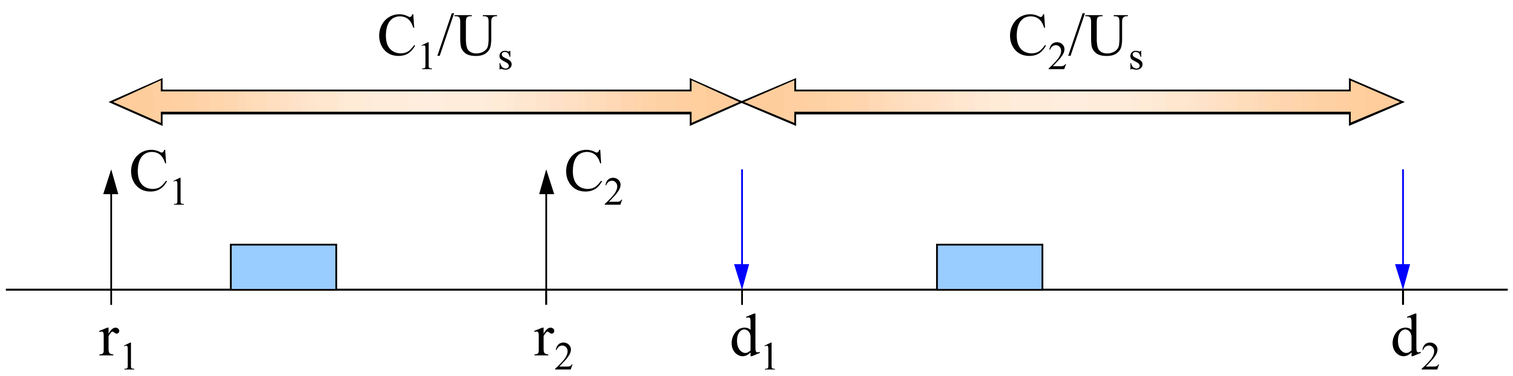
- Deadline has to be assigned not to jeopardize periodic tasks.
- A safe relative deadline is equal to the minimum period that can be assigned to a new periodic task with utilization U_s:

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

- Hence, the absolute deadline can be set as:

$$d_k = r_k + C_k / U_s$$

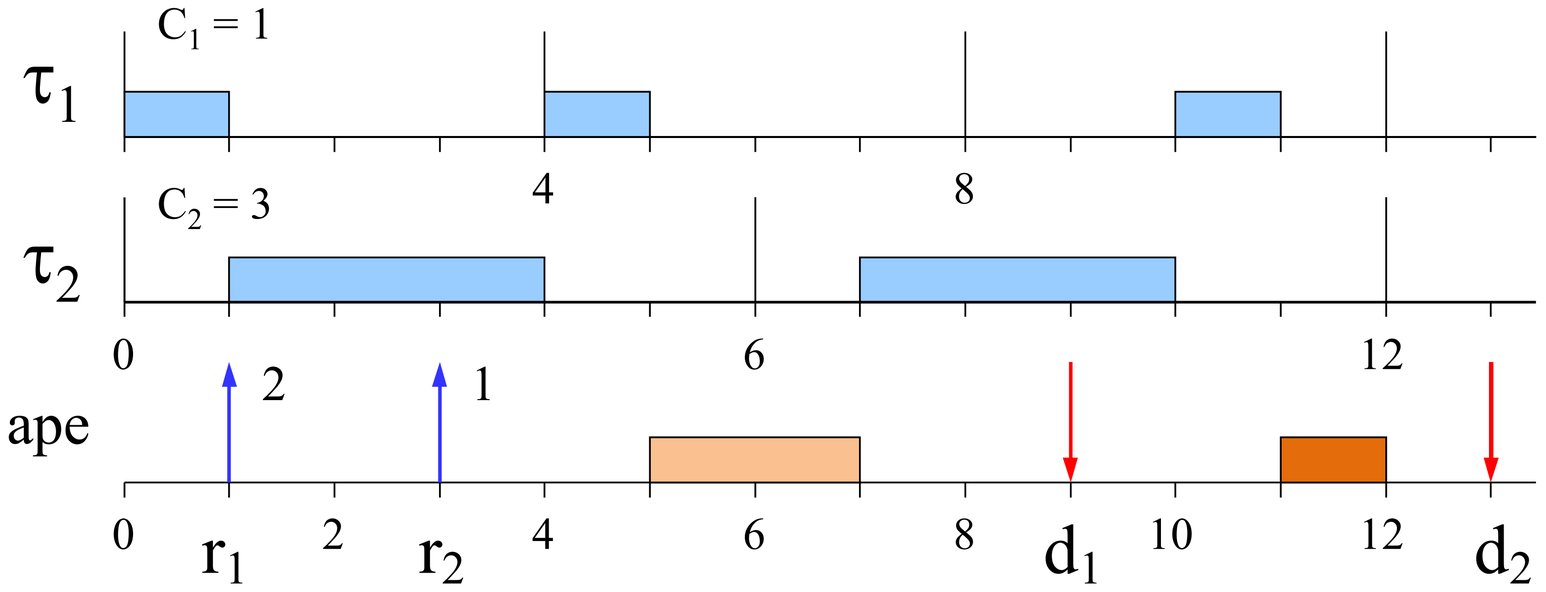
Deadline assignment rule



- 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

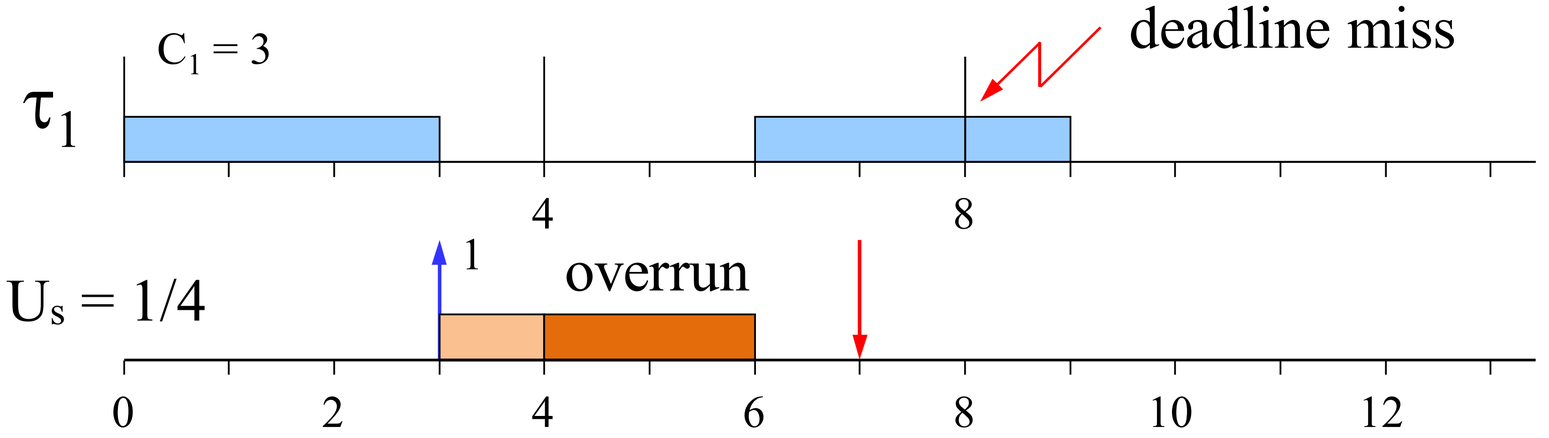


$$U_s = 1 - U_p = 1/4$$

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

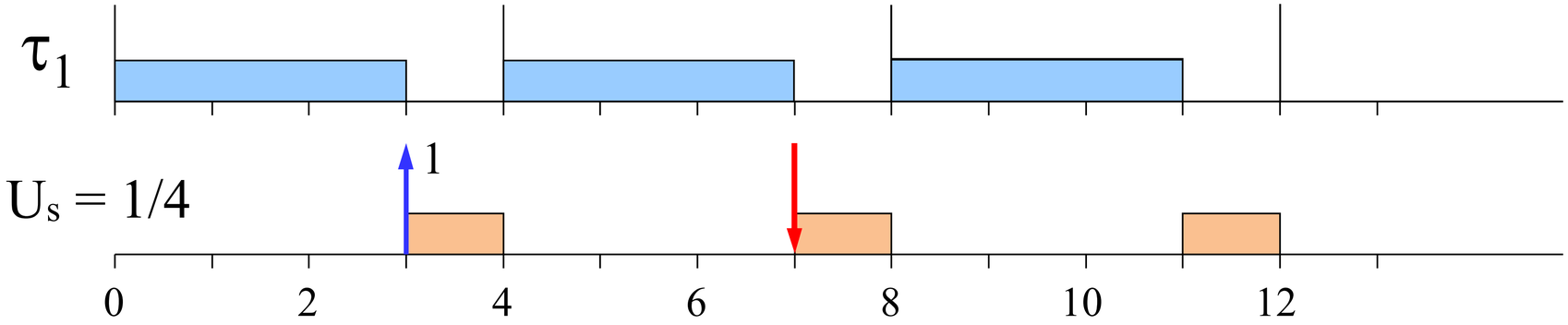
Problems with the TBS

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



Overrun handling

- If a job executes more than expected (i.e., consumes its budget) it must be delayed by decreasing its priority or postponing its deadline.



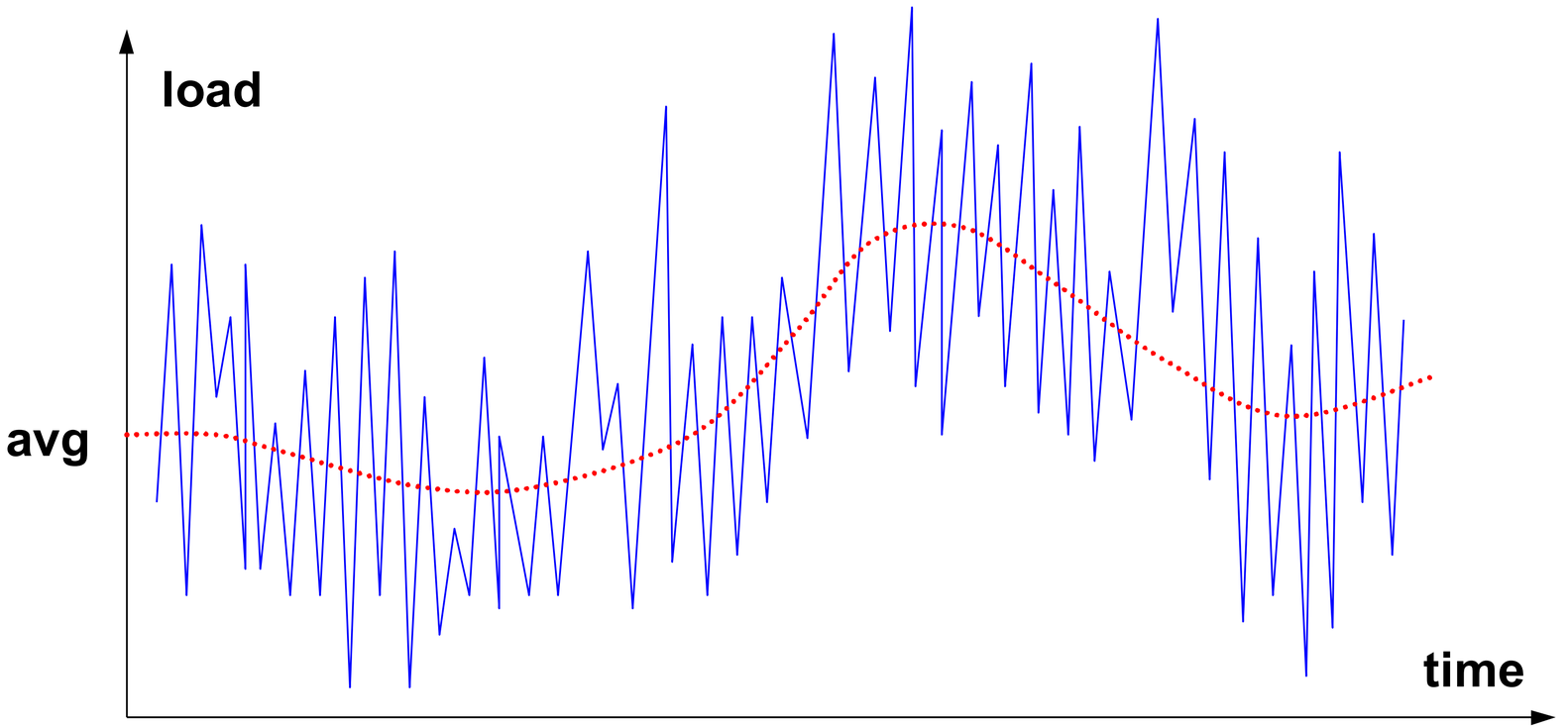
Solution: task isolation

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

Load Control Methods

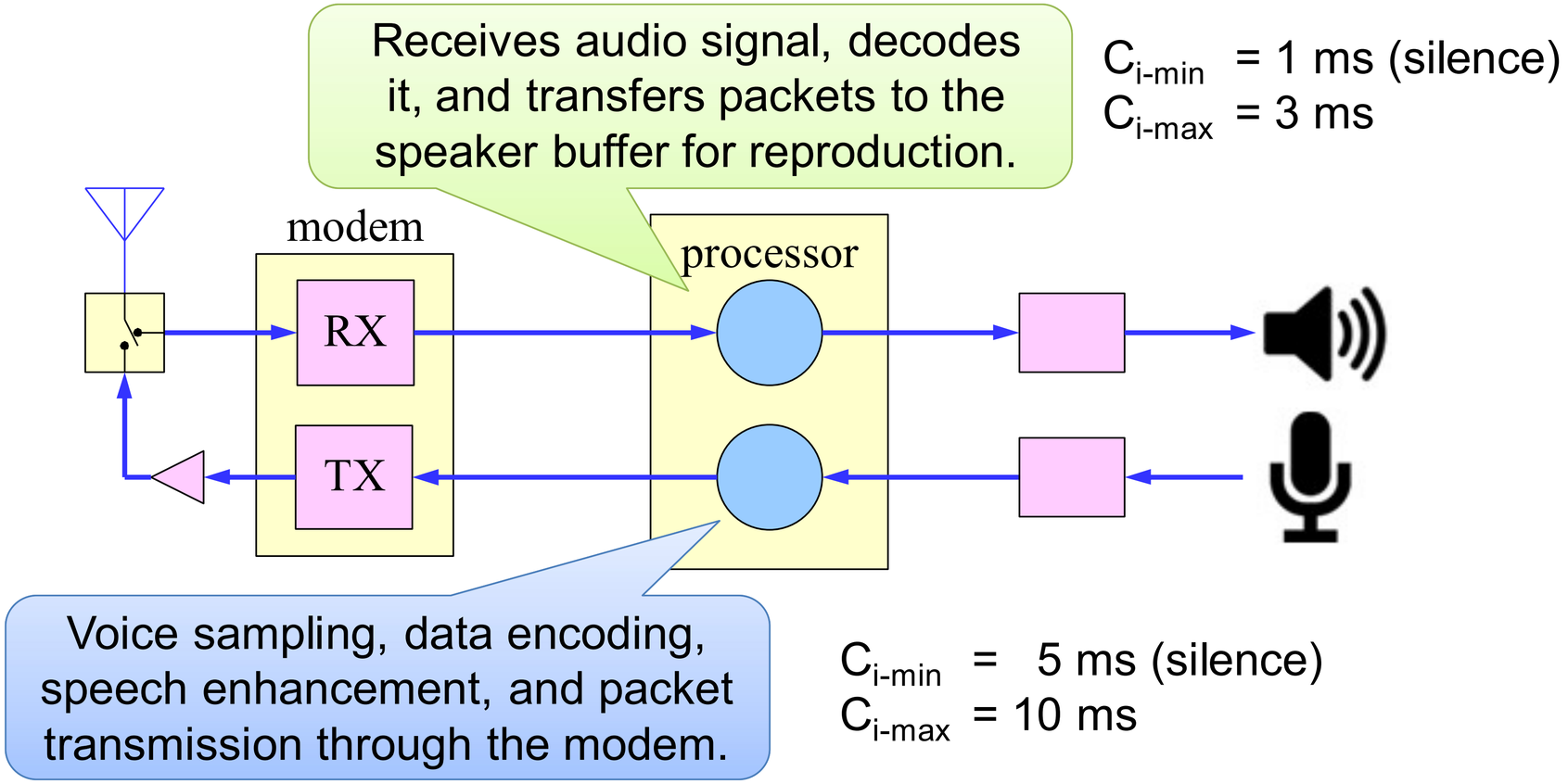
Dynamic workload

Several applications (e.g., multimedia systems) are characterized by highly variable computational requirements:



Example: phone call

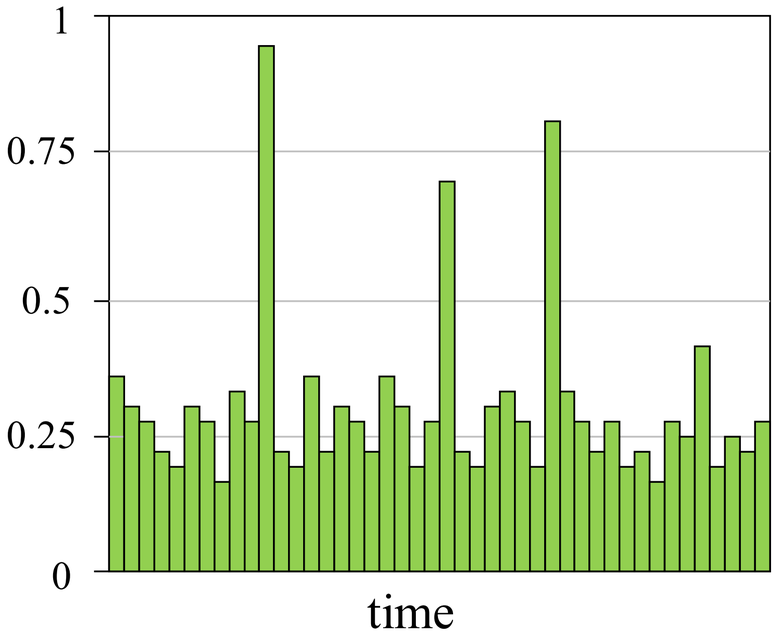
It consists at least of 2 periodic tasks, executed every 20 ms:



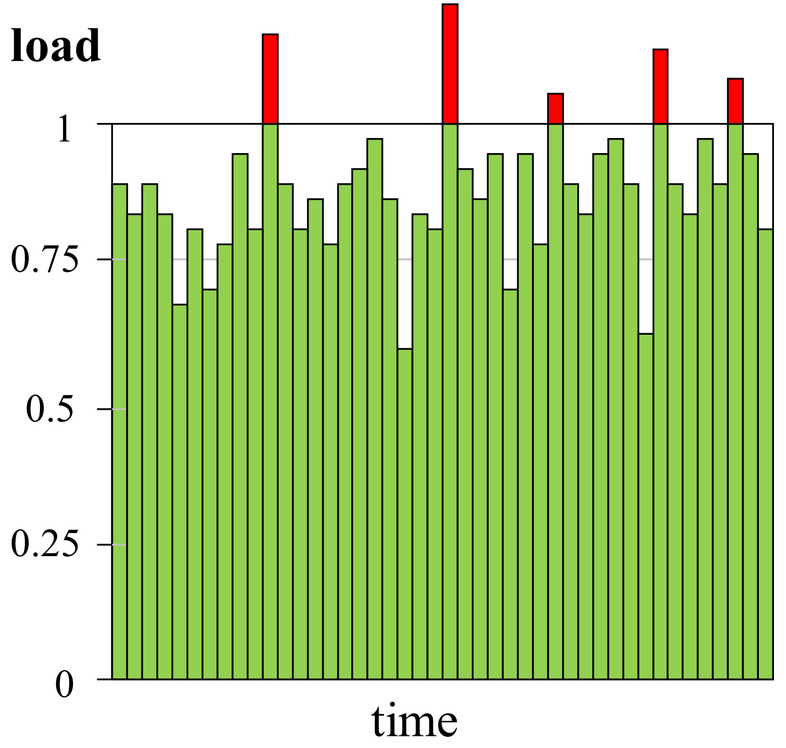
Overall CPU bandwidth: 30-65%

Load and design assumptions

System designed under worst-case assumptions



System designed under average-case assumptions



A matter of cost

- High predictability and low efficiency means wasting resources $\Rightarrow$ **high cost**
  - it can be justified only for very critical systems

- High efficiency in resource usage requires the system to:
  - handle and tolerate overloads
  - adapt for graceful degradation
  - plan for exception handling mechanisms

Definitions

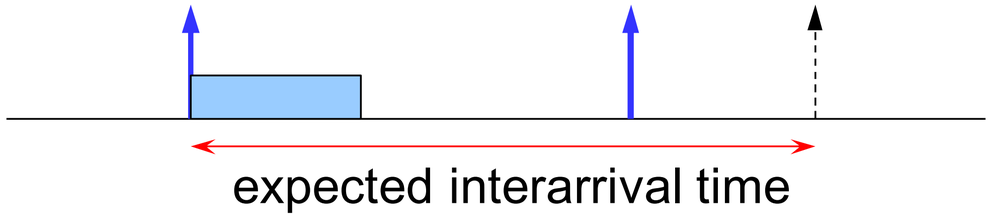
Overrun: Situation in which a task exceeds its expected utilization.

Execution Overrun

The task computation time exceeds its expected value:

Activation Overrun

The next task activation occurs before its expected time:

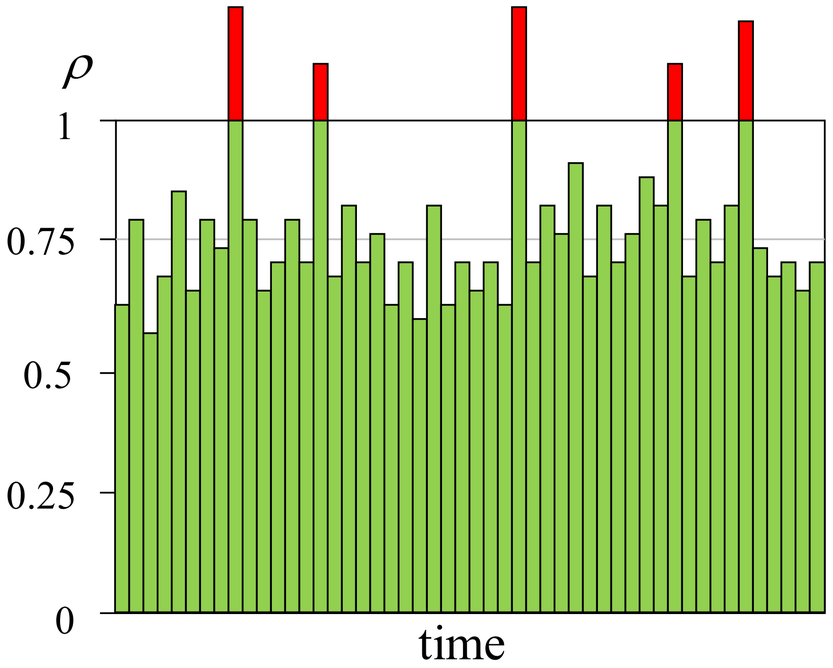


Definitions

Overload: Situation in which $\rho(t) > 1$.

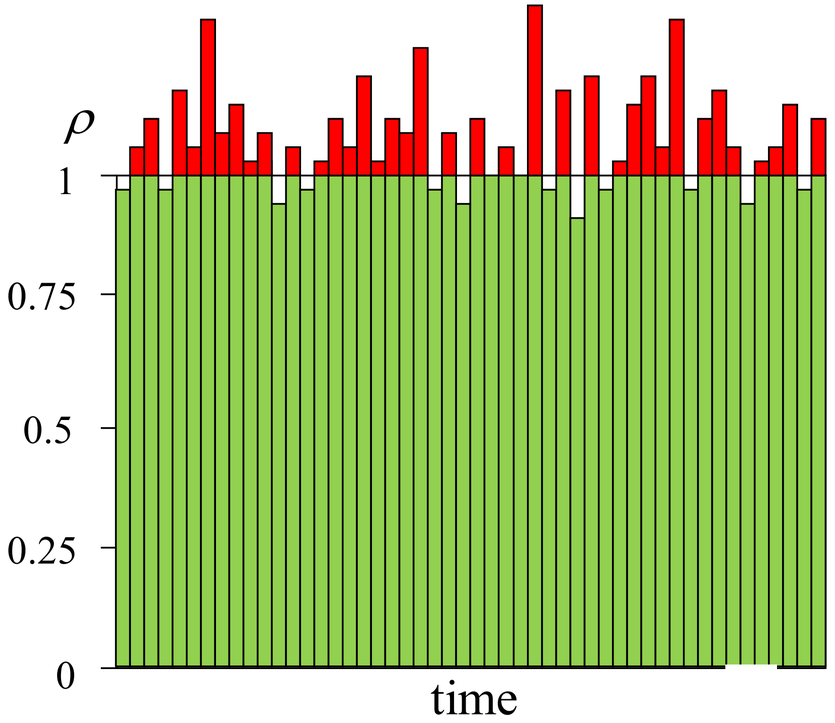
Transient Overload

$\rho_{max} > 1$ but $\rho_{avg} \leq 1$



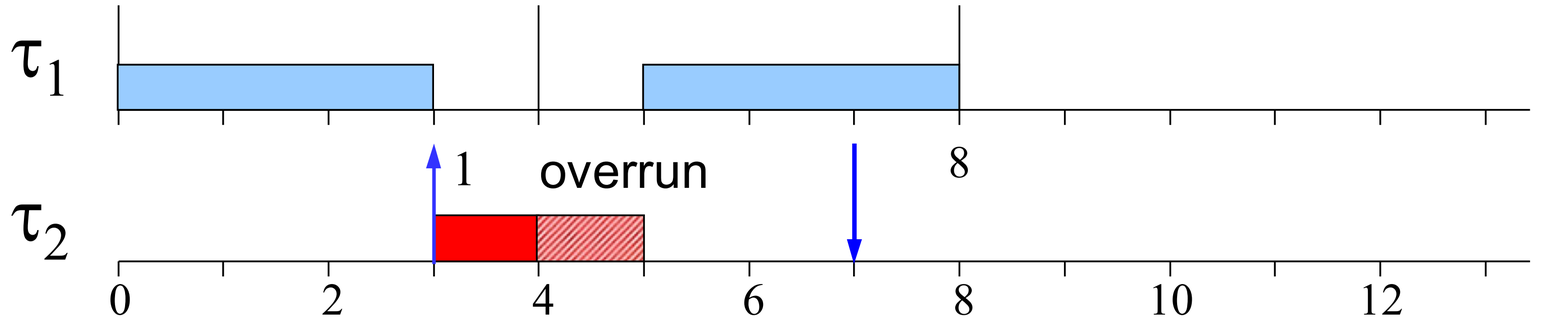
Permanent Overload

$\rho_{avg} > 1$

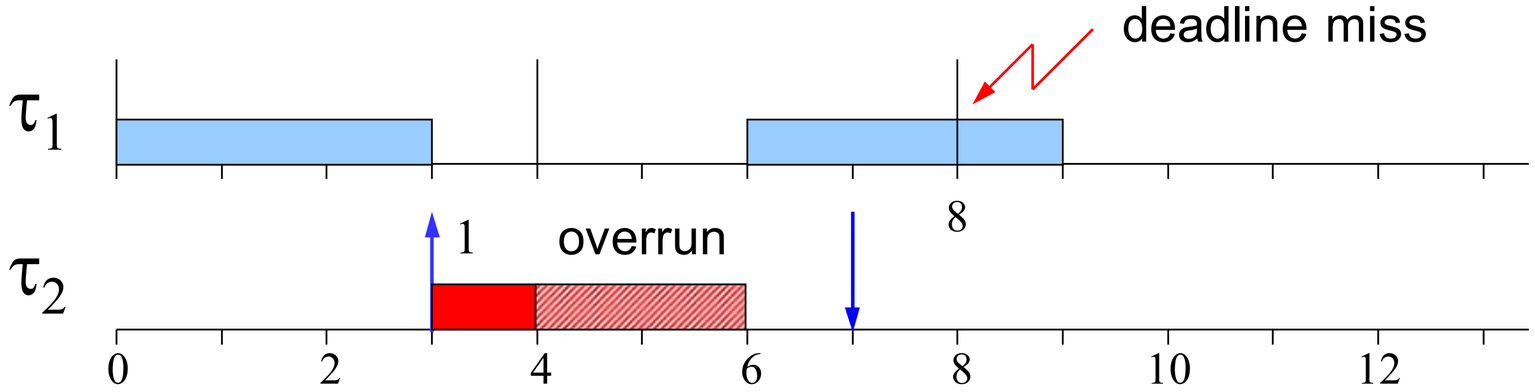


Consequences of overruns

A task overrun may not cause an overload:

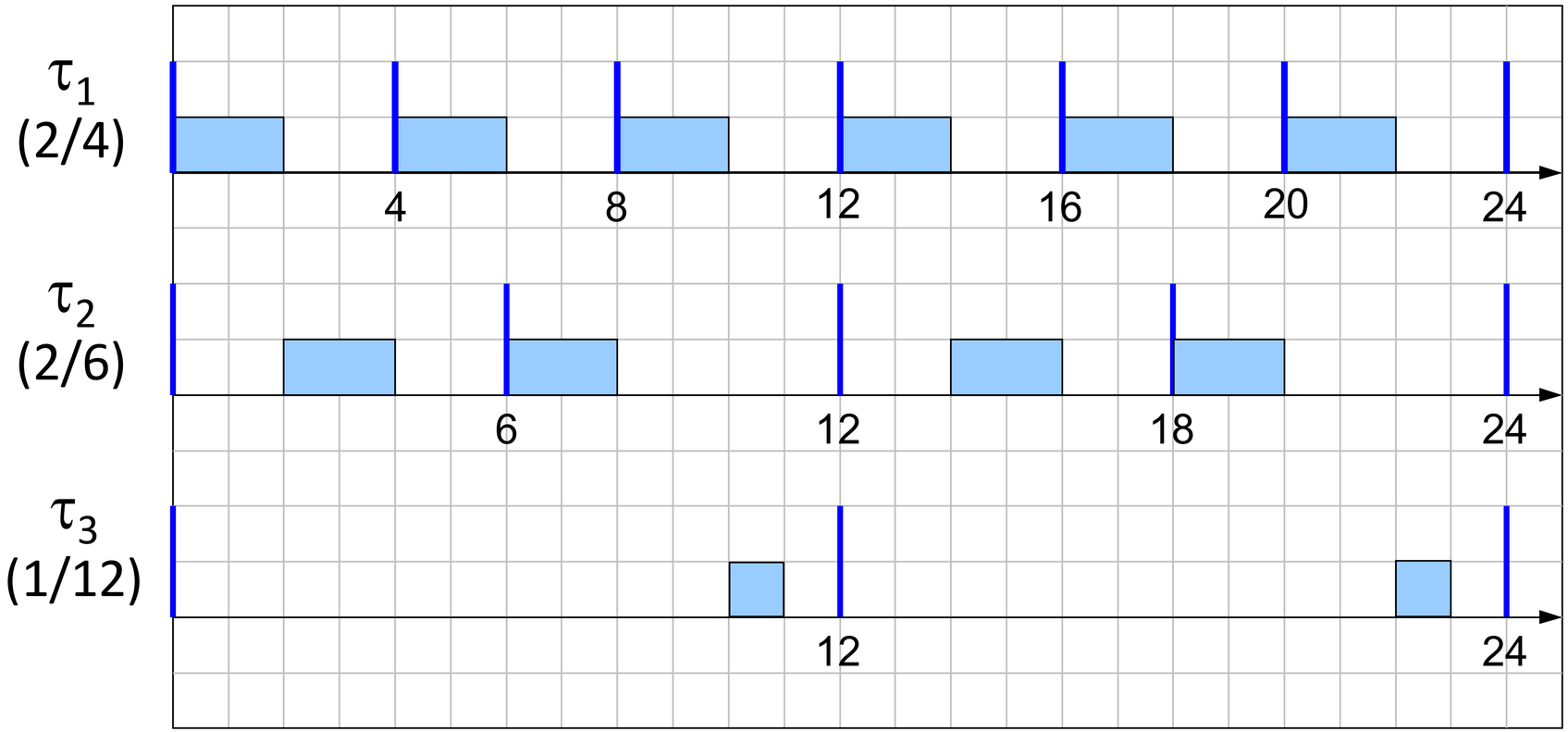


But in general may delay the execution of other tasks, causing a deadline miss:



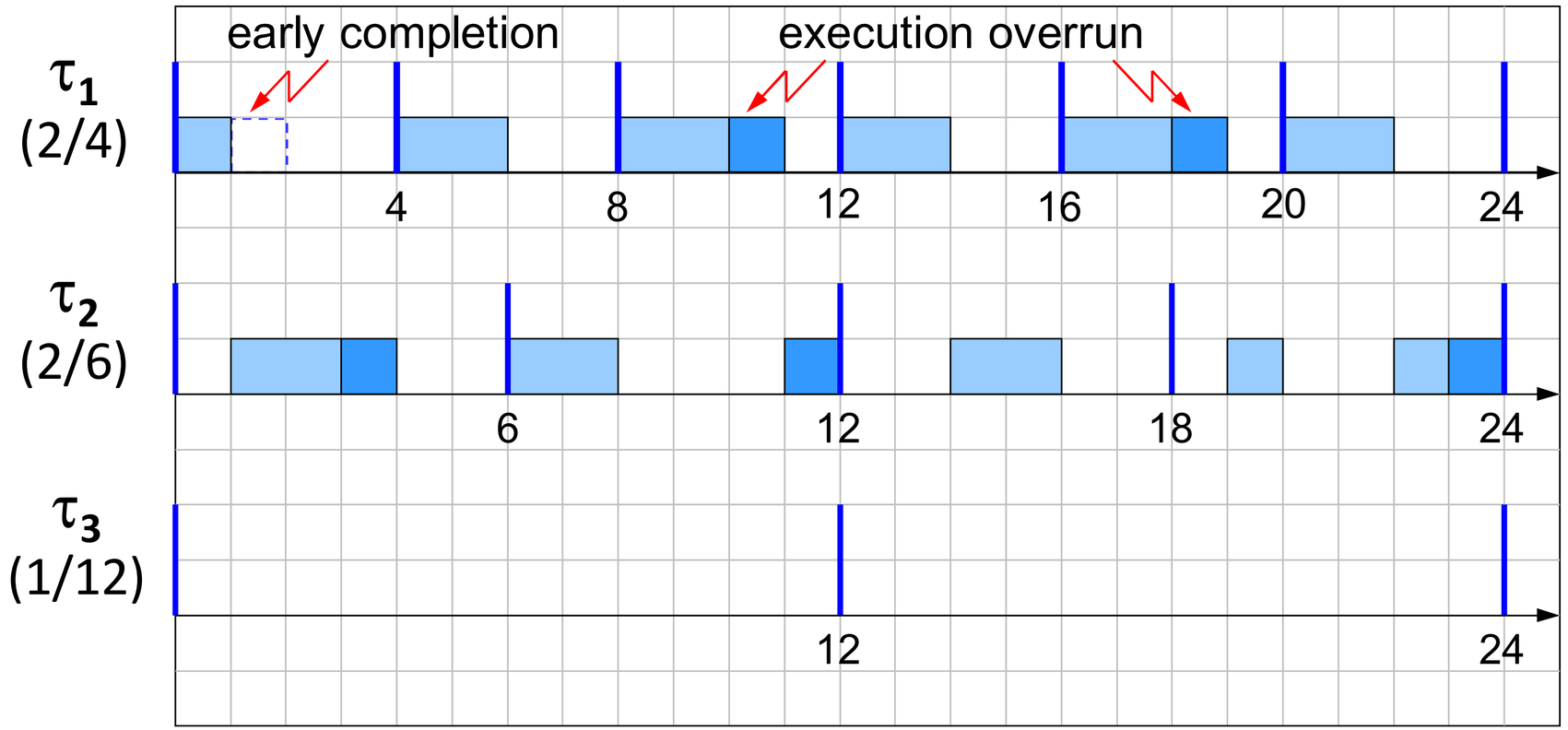
Consequences of overruns

Consider the following applications with $U < 1$:

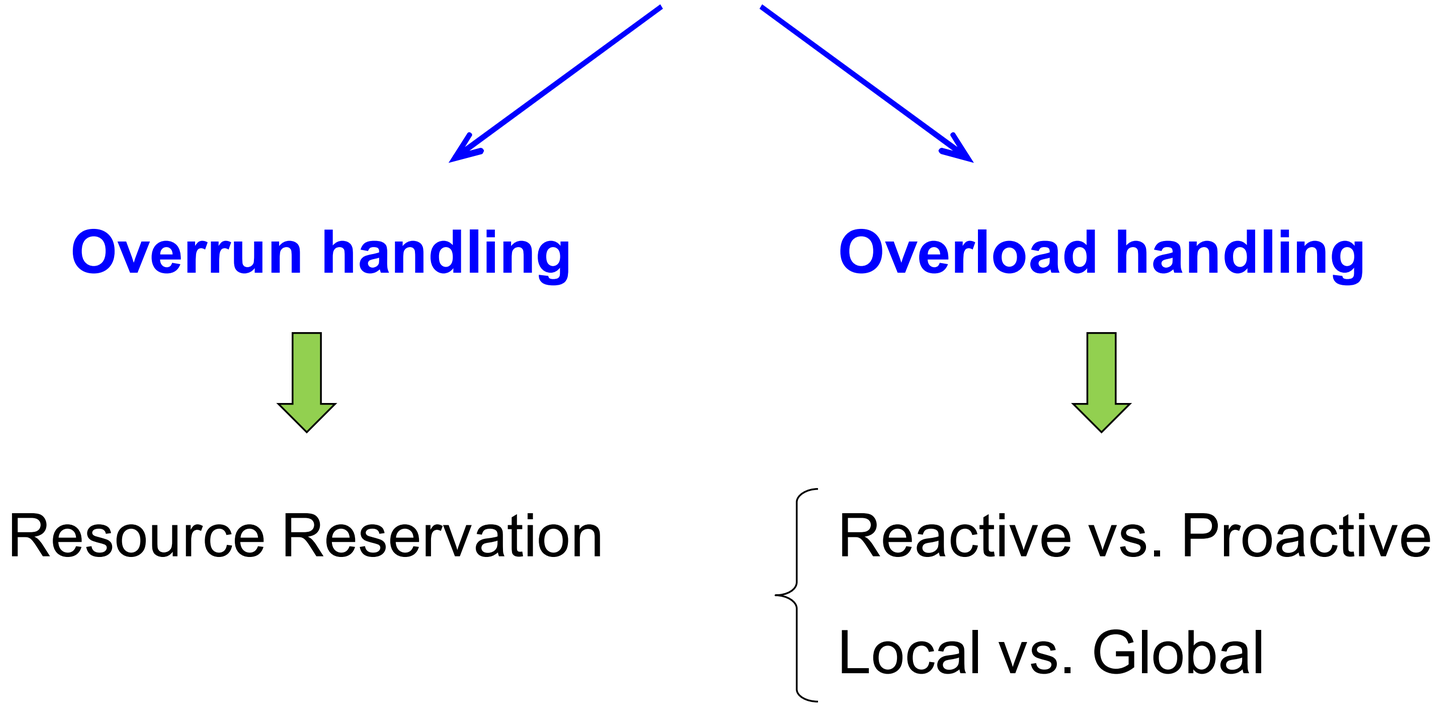


Consequences of overruns

$U < 1$ but sporadic overruns can prevent τ_3 to run:



Load control methods



Resource Reservation Methods

Providing temporal isolation

Transient overloads can be efficiently handled through **Resource Reservation**: an operating system mechanism to bound the resource consumption of tasks in order to limit the interference caused to other tasks.

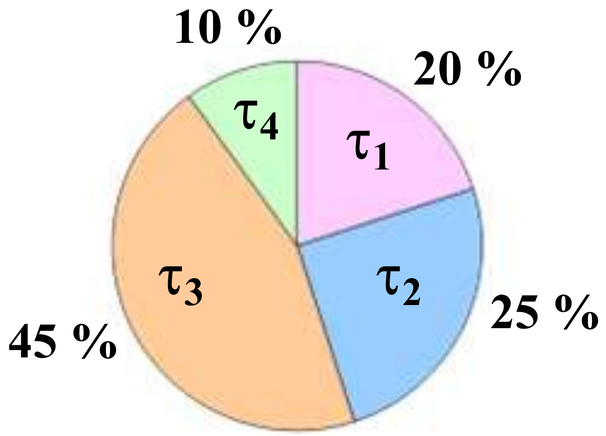
To implement Resource Reservation we have to:

1. reserve a fraction of the processor bandwidth to the tasks we want to keep under load control;

2. prevent the such tasks to use more bandwidth than the reserved fraction.

Resource Reservation

Resource partition



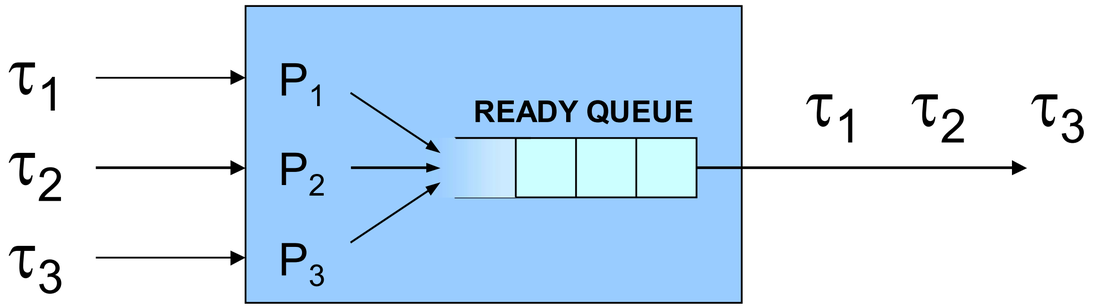
Each task receives a bandwidth α_i and behaves as it were executing alone on a slower processor of speed α_i

Resource enforcement

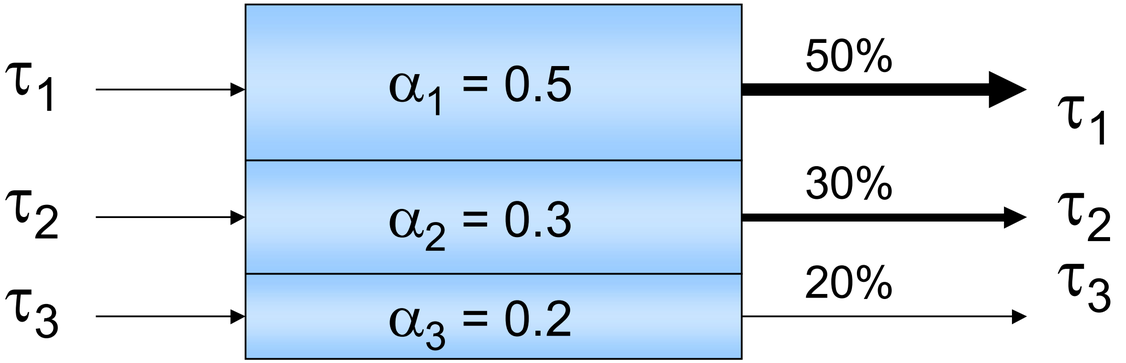
- A mechanism that prevents a task to consume more than its reserved amount.
- If a task executes more, it is delayed, preserving the resource for other tasks.

Priorities vs. Reservations

Prioritized Access

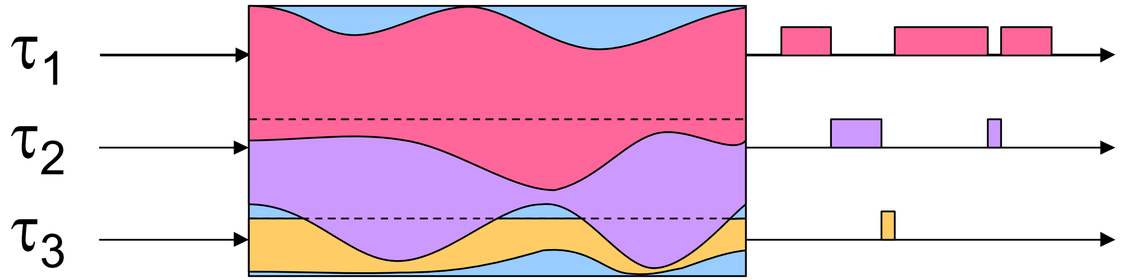


Resource Reservation

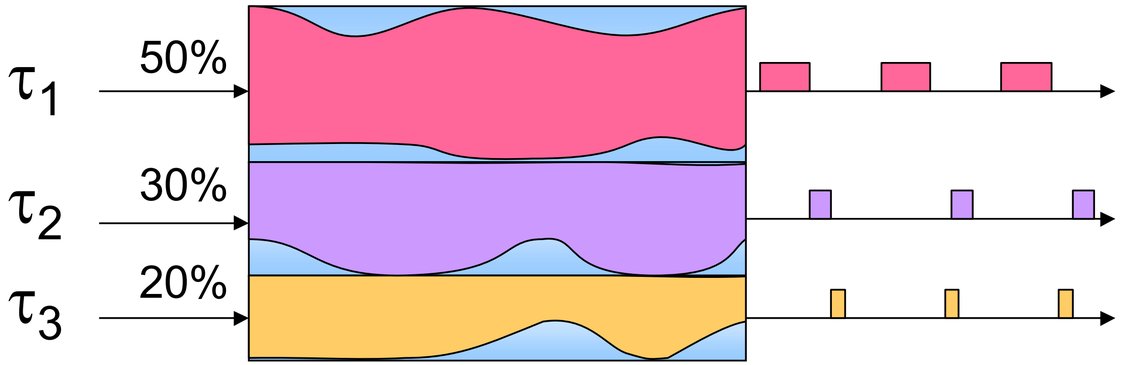


Priorities vs. Reservations

Prioritized Access



Resource Reservation



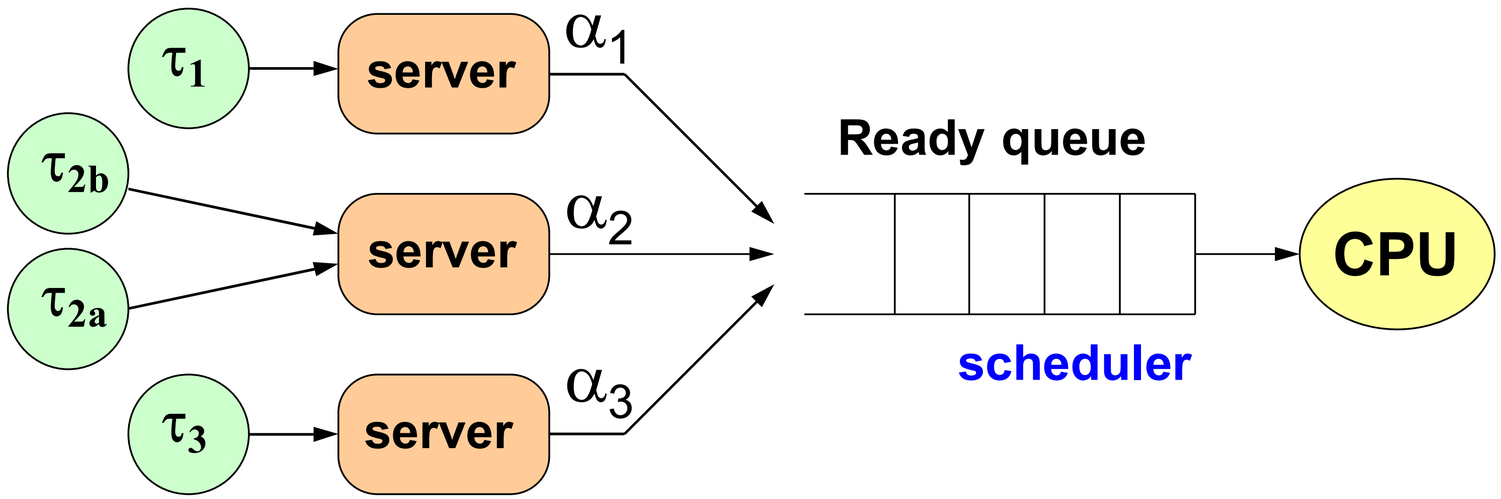
Benefits of Resource Reservation

1. Resource allocation is easier than priority mapping.

2. It provides temporal isolation: overruns occurring in a reservation do not affect other tasks.
   - Important for modularity and scalability

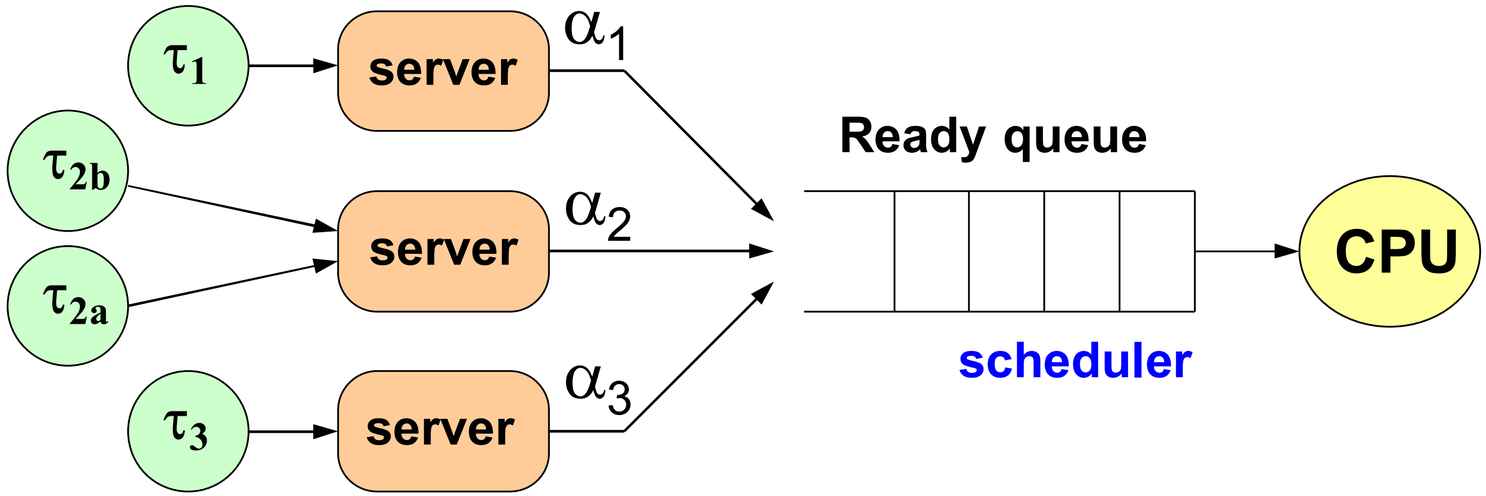
3. Simpler schedulability analysis:
   - Response times only depends on the application demand and the amount of reserved resource.

Implementing RR



	scheduler	server
Fixed priorities	**RM/DM**	**Sporadic Server**
Dynamic priorities	**EDF**	**CBS**

Analysis under RR



If a processor is partitioned into *n* reservations, we must have that:

$$\sum_{i=1}^{n} \alpha_i \leq U_{\text{lub}}^{A}$$

where ***A*** is the adopted scheduling algorithm.

Types of reservations

HARD: when the budget is exhausted, the server is blocked until the next budget replenishment.

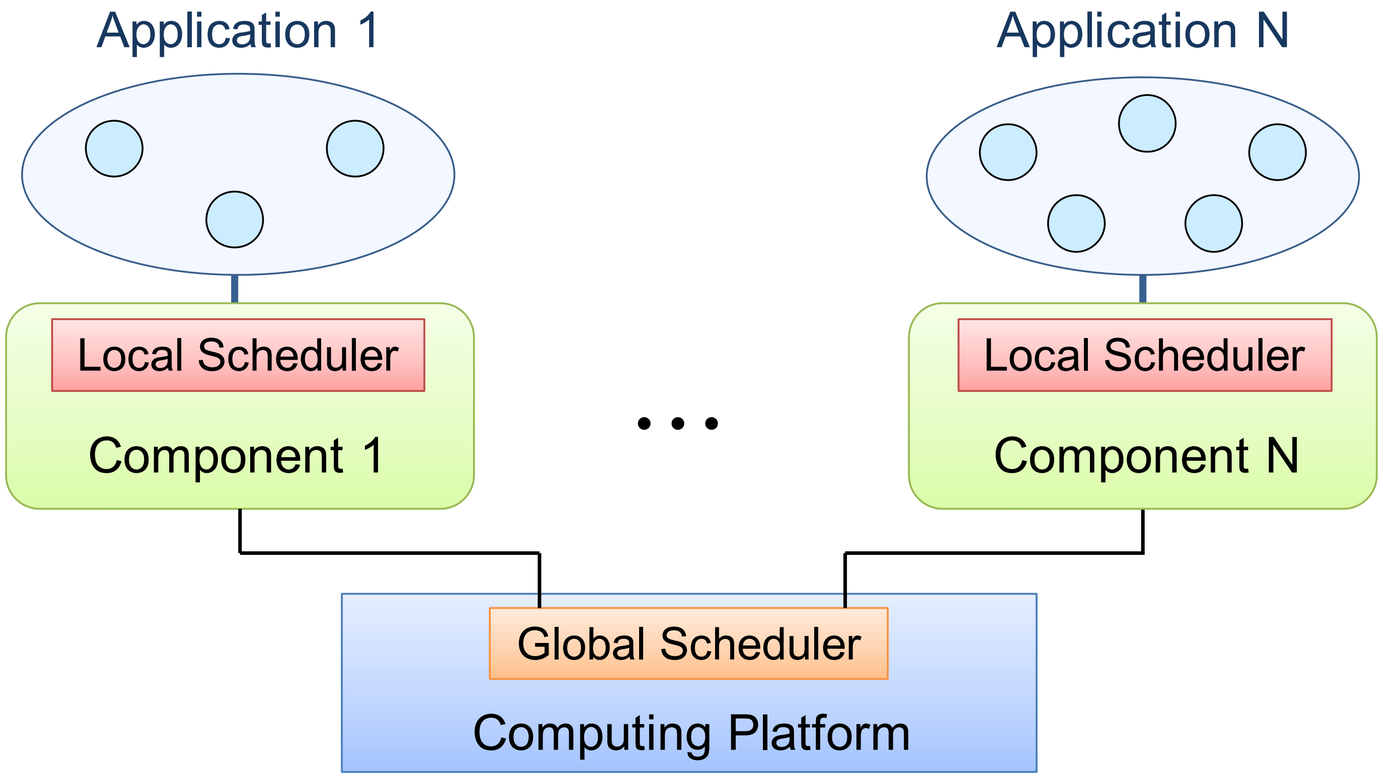
SOFT: when the budget is exhausted, the server remains active but at a lower priority level.

Note that

- a HARD reservation (Q_s, P_s) guarantees at most a budget Q_s every period P_s
- a SOFT reservation (Q_s, P_s) guarantees at least a budget Q_s every period P_s

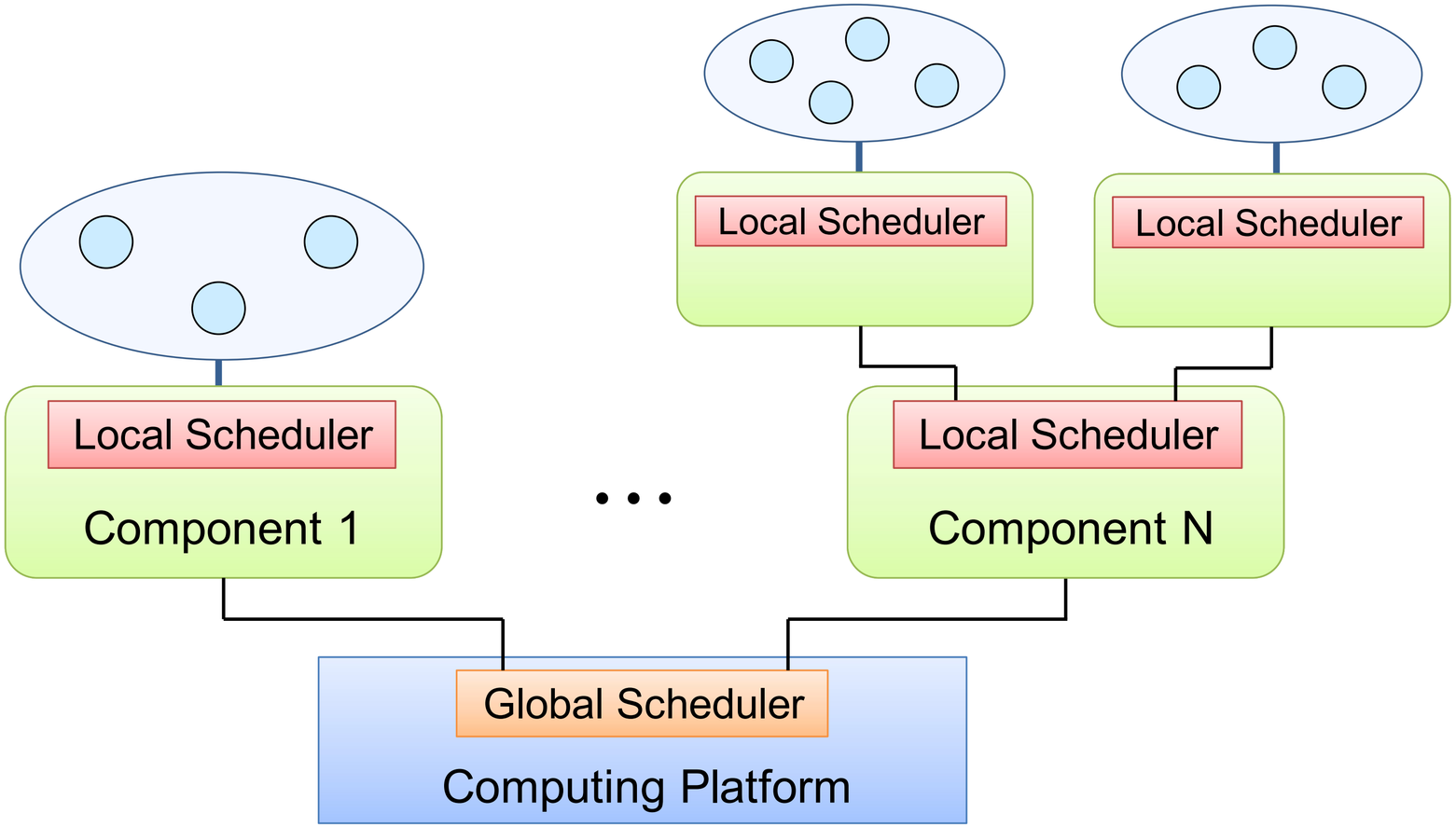
Hierarchical scheduling

Resource reservation can be used to develop hierarchical systems, where each component is implemented within a reservation:



Hierarchical scheduling

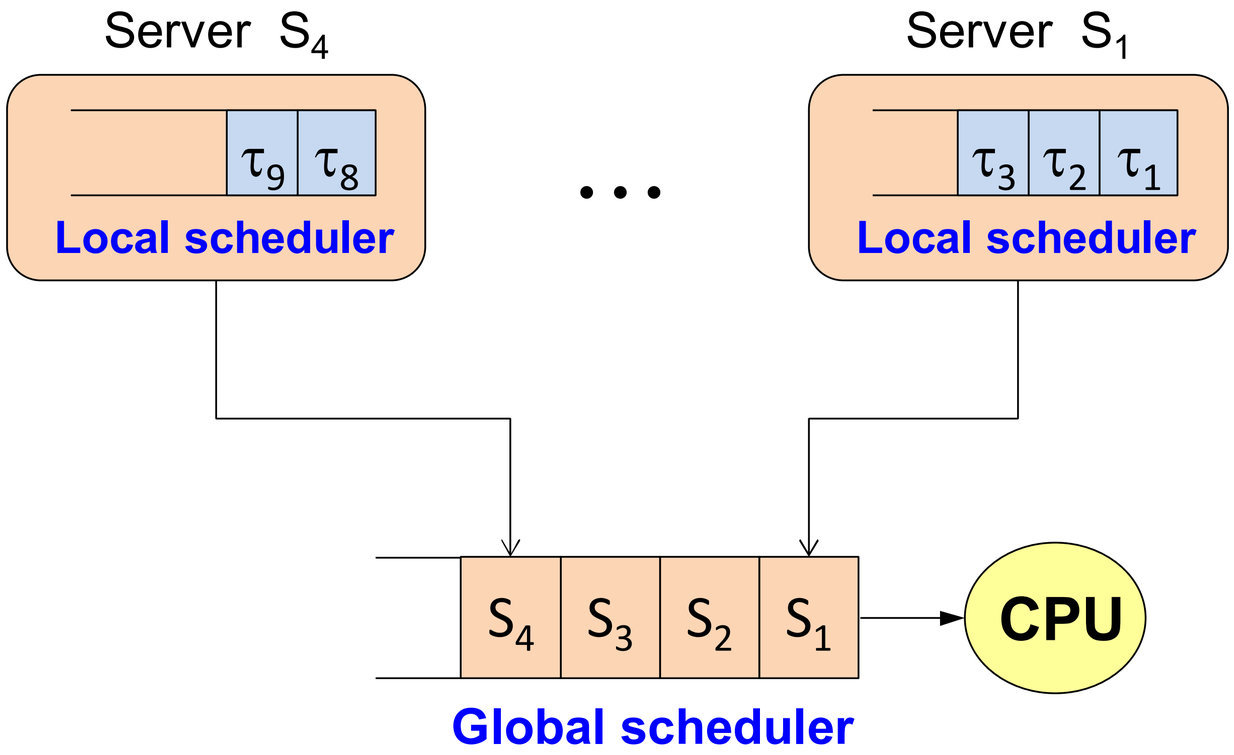
In general, a component can also be divided in other sub-components



Hierarchical scheduling

Global scheduler: the one managing the system ready queue

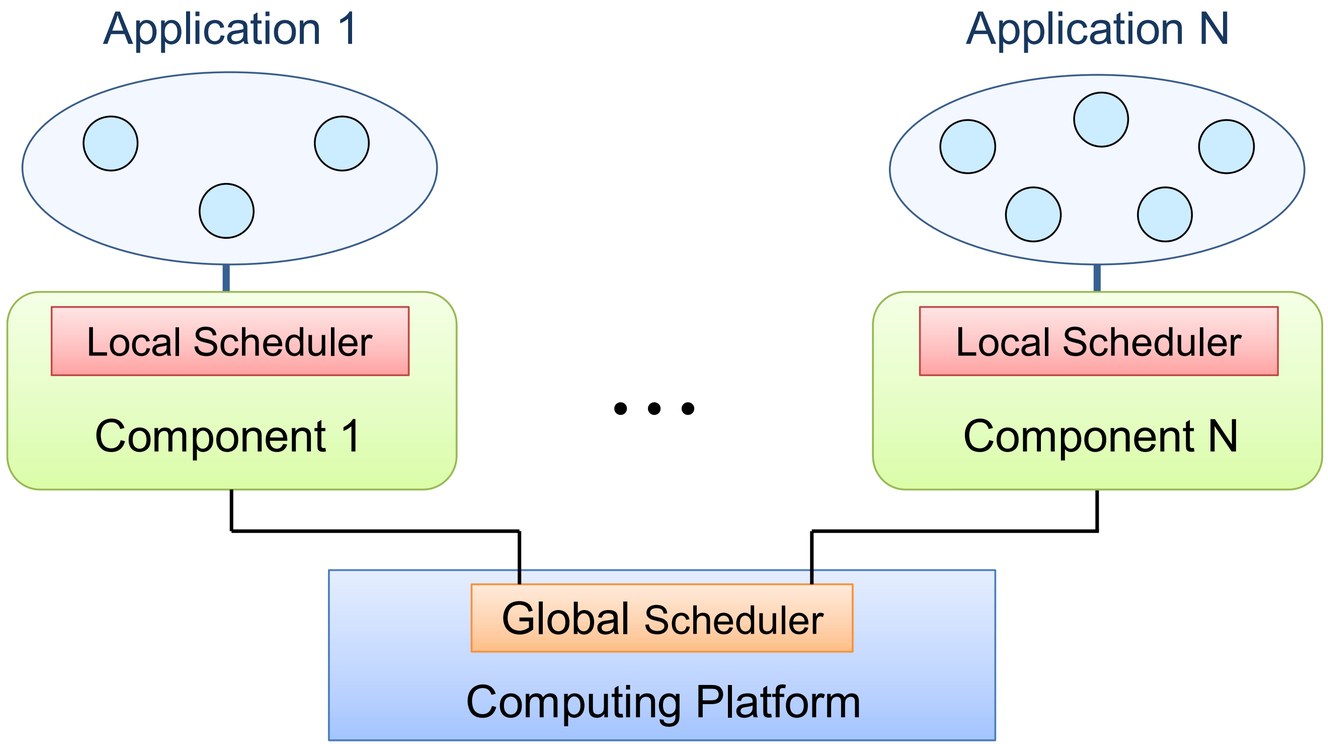
Local schedulers: the ones managing the servers queues



Hierarchical Analysis

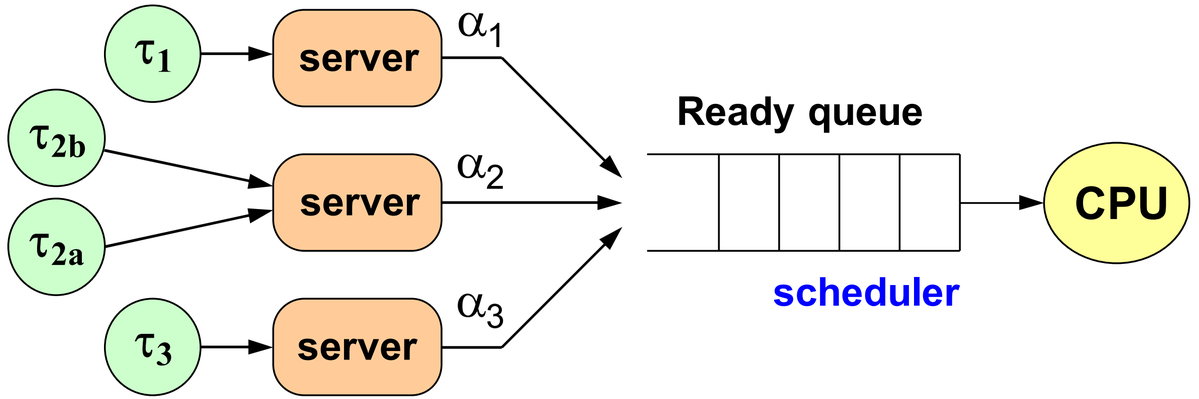
Global analysis: Servers must be schedulable by the global scheduler running on the physical platform.

Local analysis: Applications must be scheduled by the local schedulers running on the components.



Global Analysis

Global analysis: Servers must be schedulable by the global scheduler running on the physical platform.



If a processor is partitioned into *n* reservations, we must have that:

$$\sum_{i=1}^{n} \alpha_i \leq U_{\text{lub}}^{A}$$

where ***A*** is the adopted scheduling algorithm.

Local Analysis under RR

Processor Demand Criterion (under EDF)

$$\forall t \in D \qquad dbf(t) \leq t$$

Timing behavior of the application

processing time available in $[0, t]$

Workload Analysis (under fixed priorities):

$$\forall i = 1, \ldots, n$$

$$\exists\ t \in (0, D_i] : W_i(t) \leq t$$

Local Analysis under RR

Under EDF, the analysis of an application within a reservation is done through the Processor Demand Criterion:

$$\forall t > 0, \quad dbf(t) \leq t$$

Under Fixed Priority Systems (FPS), the analysis is done through the Workload Analysis:

$$\forall i = 1,\ldots,n \quad \exists\ t \in (0, D_i] : W_i(t) \leq t$$

The difference is that in an interval of length t the processor is only partially available.

Local Analysis under RR

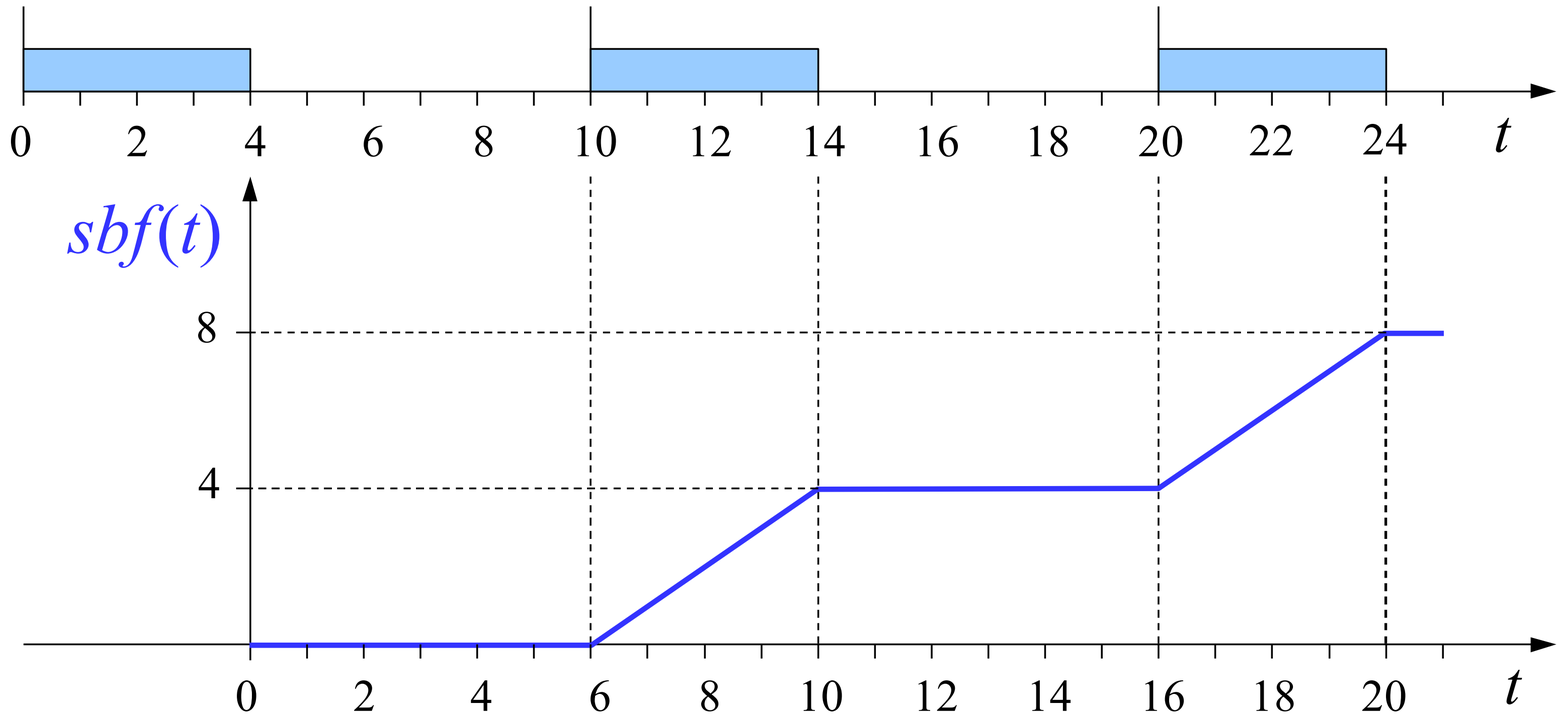
To describe the time available in a reservation, we need to identify, for any interval $[0, t]$, the minimum time allocated in the worst-case situation.

Supply bound function $sbf(t)$*:*

minimum amount of time available in reservation R_k *in every time interval of length* t*.*

Example: Static time partition

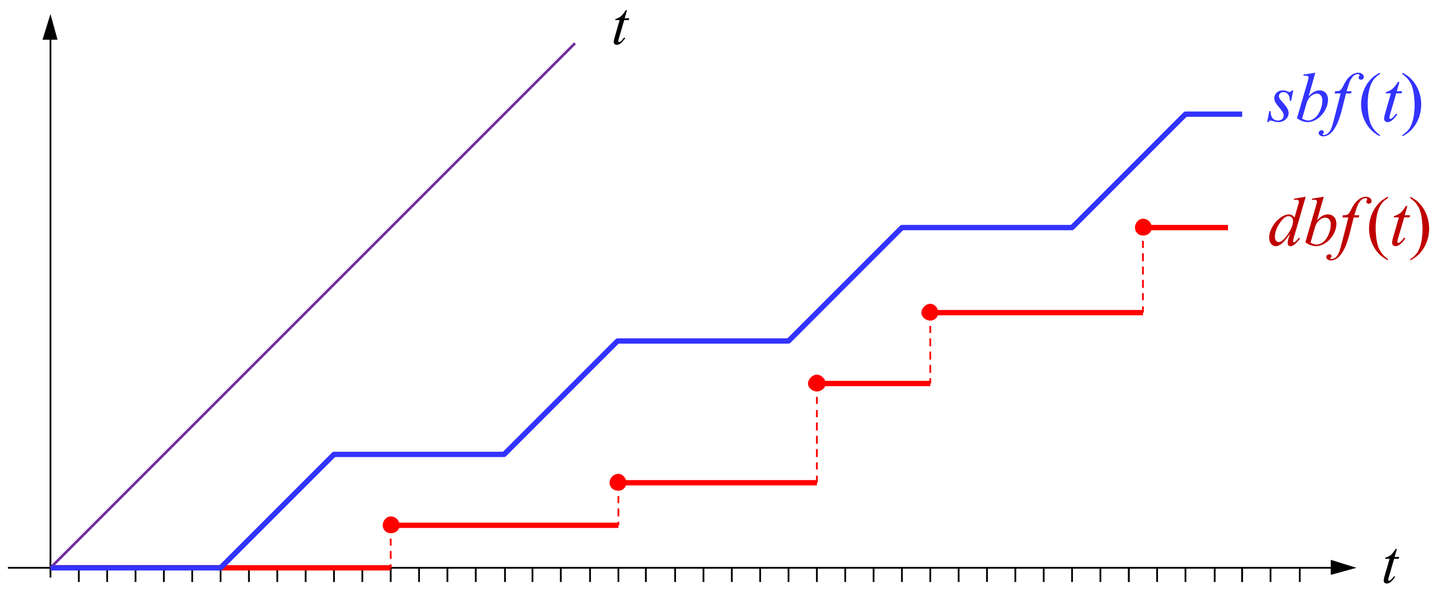
Example of reservation providing 4 units of time every 10 (bandwidth = 0.4):



Analysis under RR

Hence the Processor Demand Criterion can be reformulated as follows:

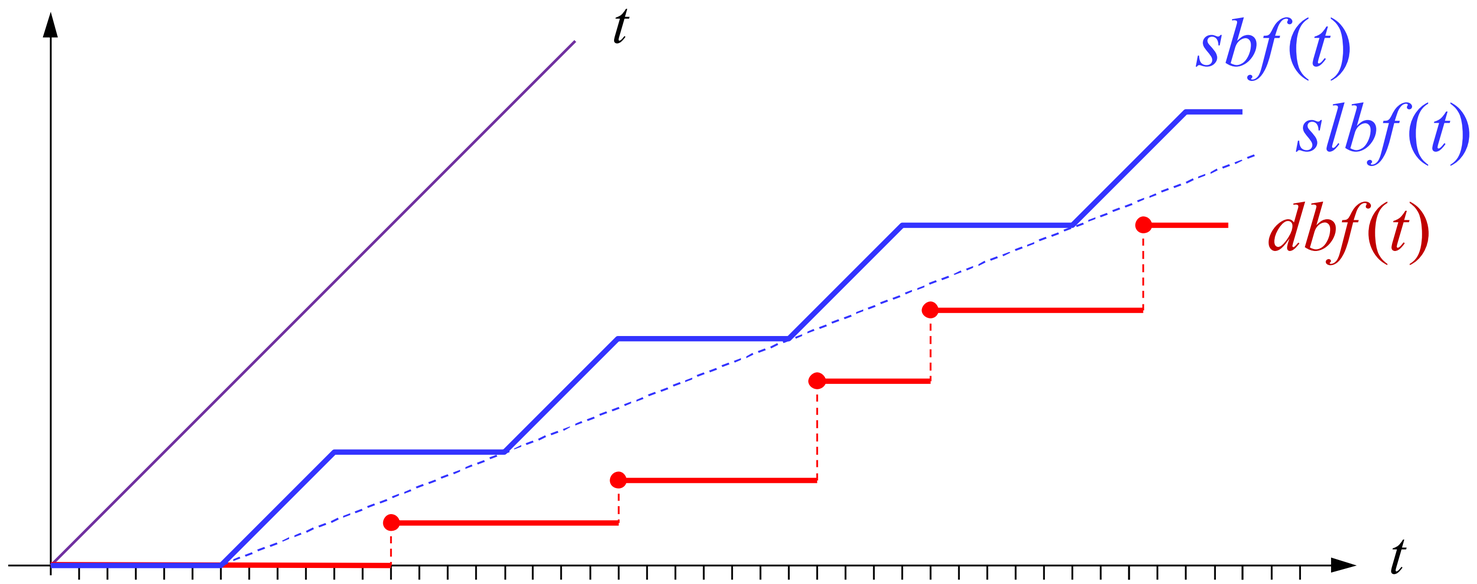
$$\forall t > 0, \quad dbf(t) \leq sbf(t)$$



Local Analysis under RR

A simpler sufficient test, can be done by replacing $sbf(t)$ with a lower bound, called supply lower-bound function $slbf(t)$:

$$\forall t > 0, \quad dbf(t) \leq slbf(t)$$



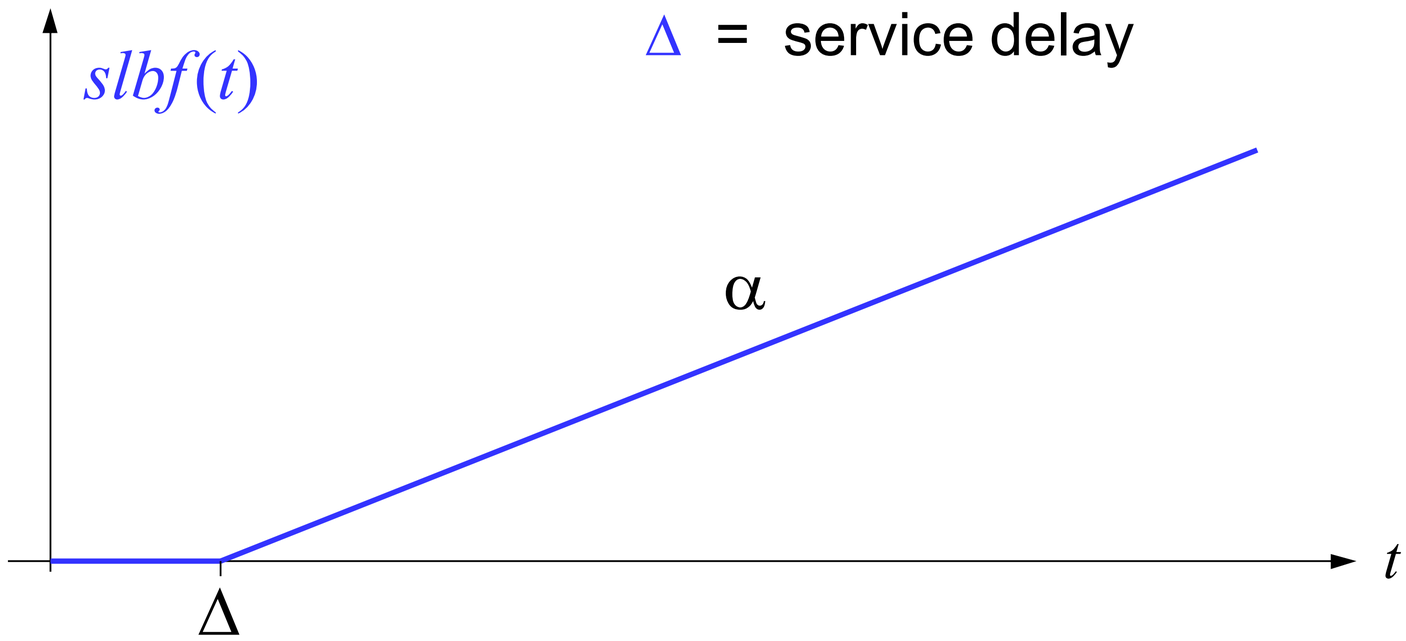
Supply lower-bound function

A supply bound function has the following form:

$$slbf(t) = \max\{0,\ \alpha(t-\Delta)\}$$

α = bandwidth

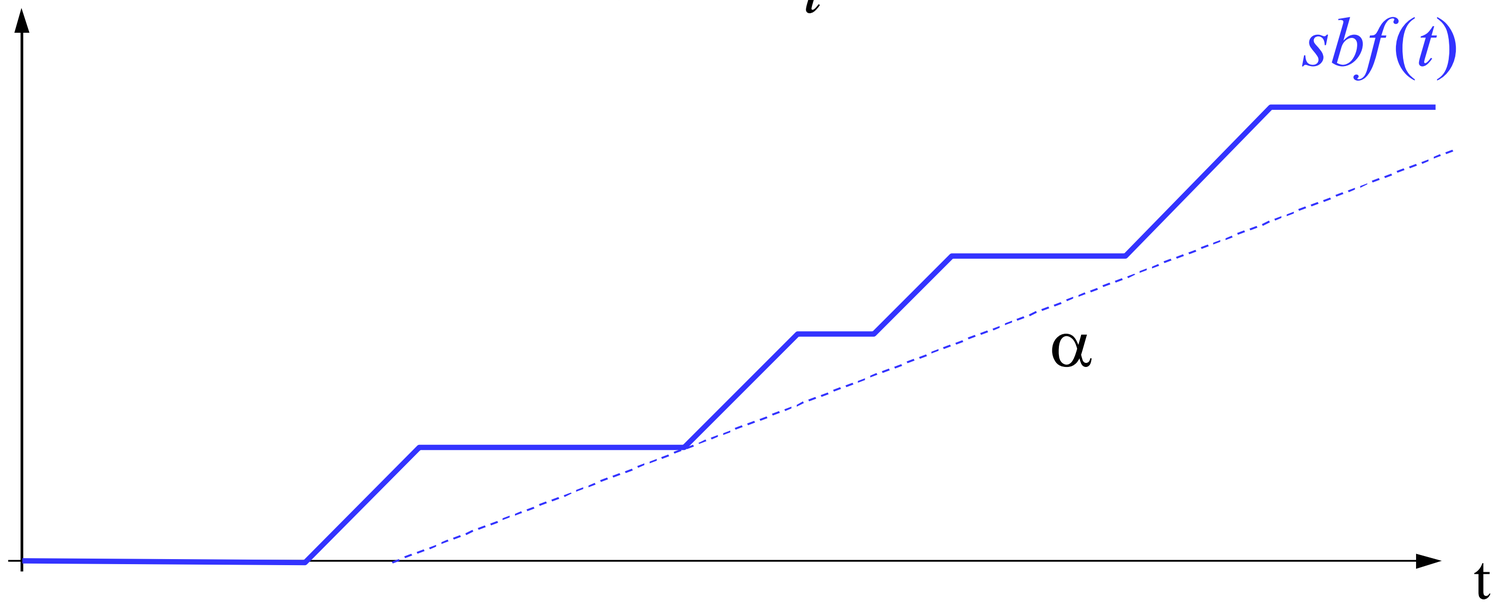
Δ = service delay



Deriving α and Δ

Given a generic supply function $sbf(t)$, the bandwidth α is the equivalent slope computed for long intervals:

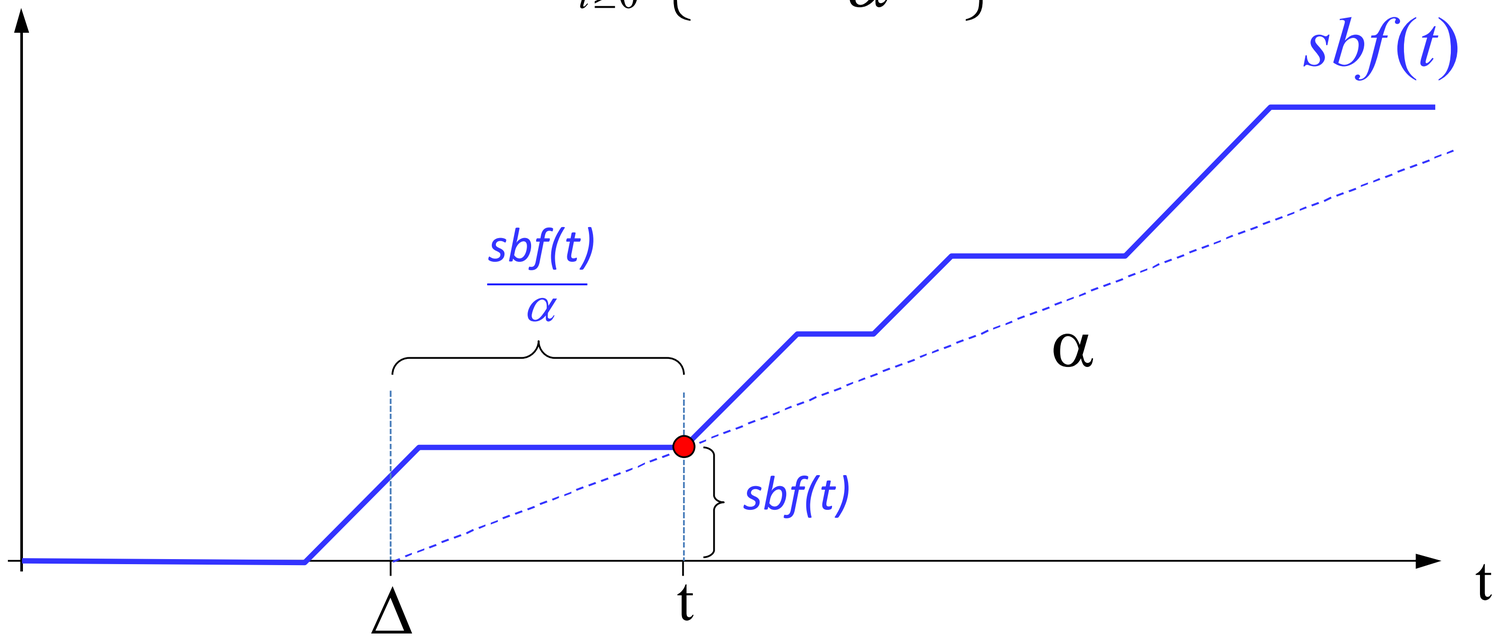
$$\alpha = \lim_{t \to \infty} \frac{sbf(t)}{t}$$



Deriving α and Δ

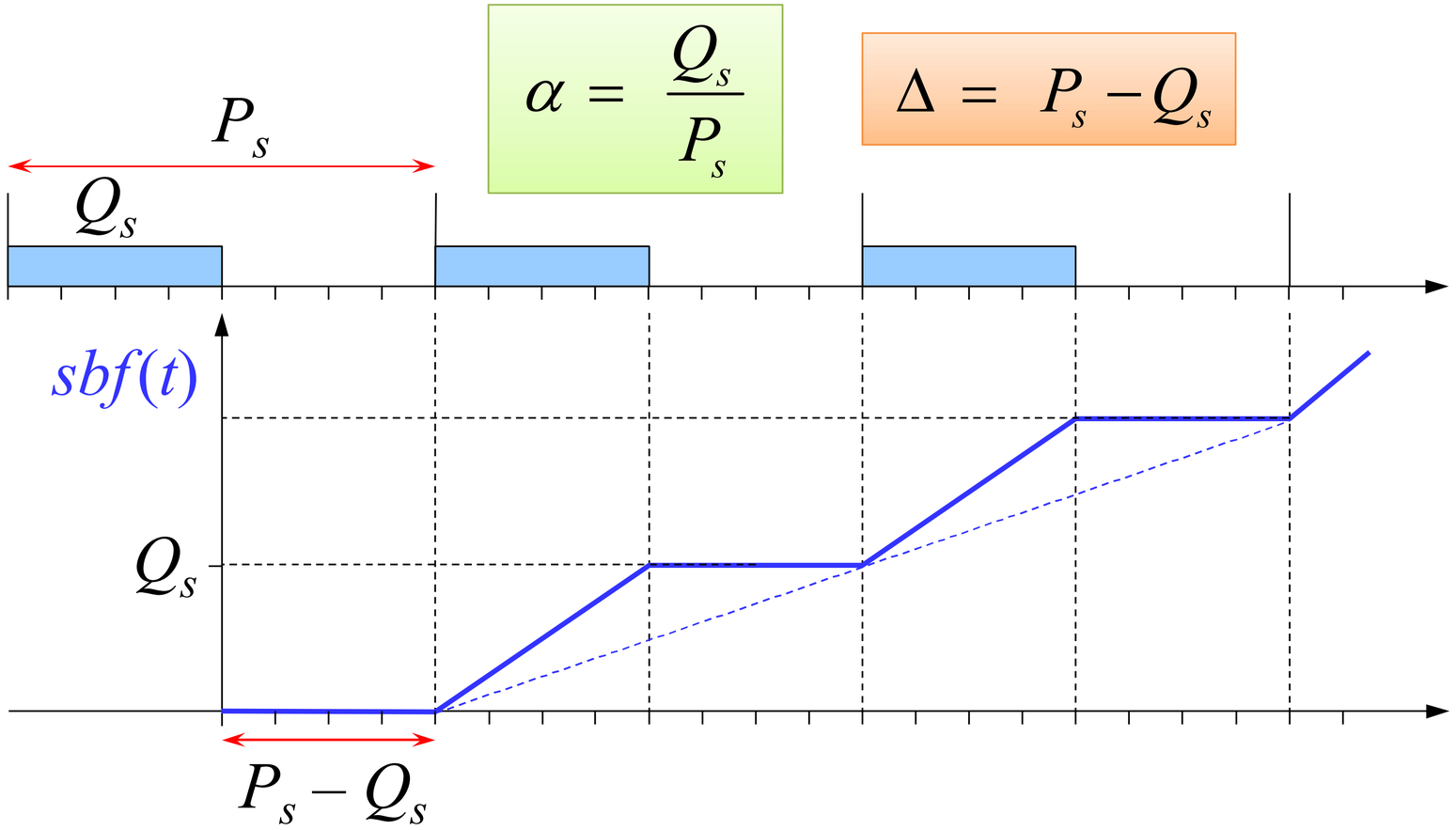
While the delay Δ is the highest intersection with the time axis of the line of slope α touching the $sbf(t)$:

$$\Delta = \sup_{t \geq 0} \left\{ t - \frac{sbf(t)}{\alpha} \right\}$$



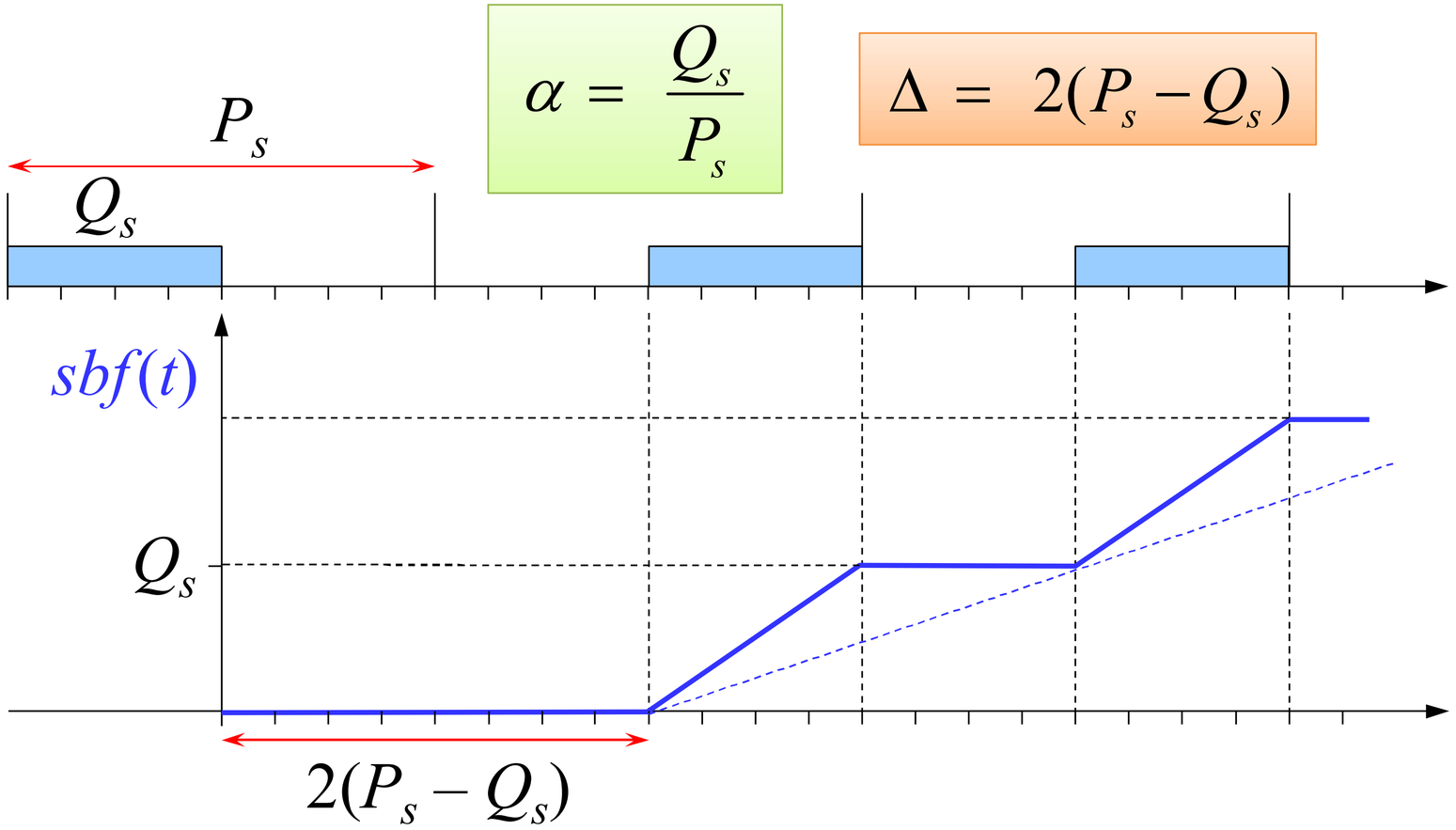
Example: Periodic Server

For a periodic server with budget Q_s and period P_s running at the highest priority, we have:



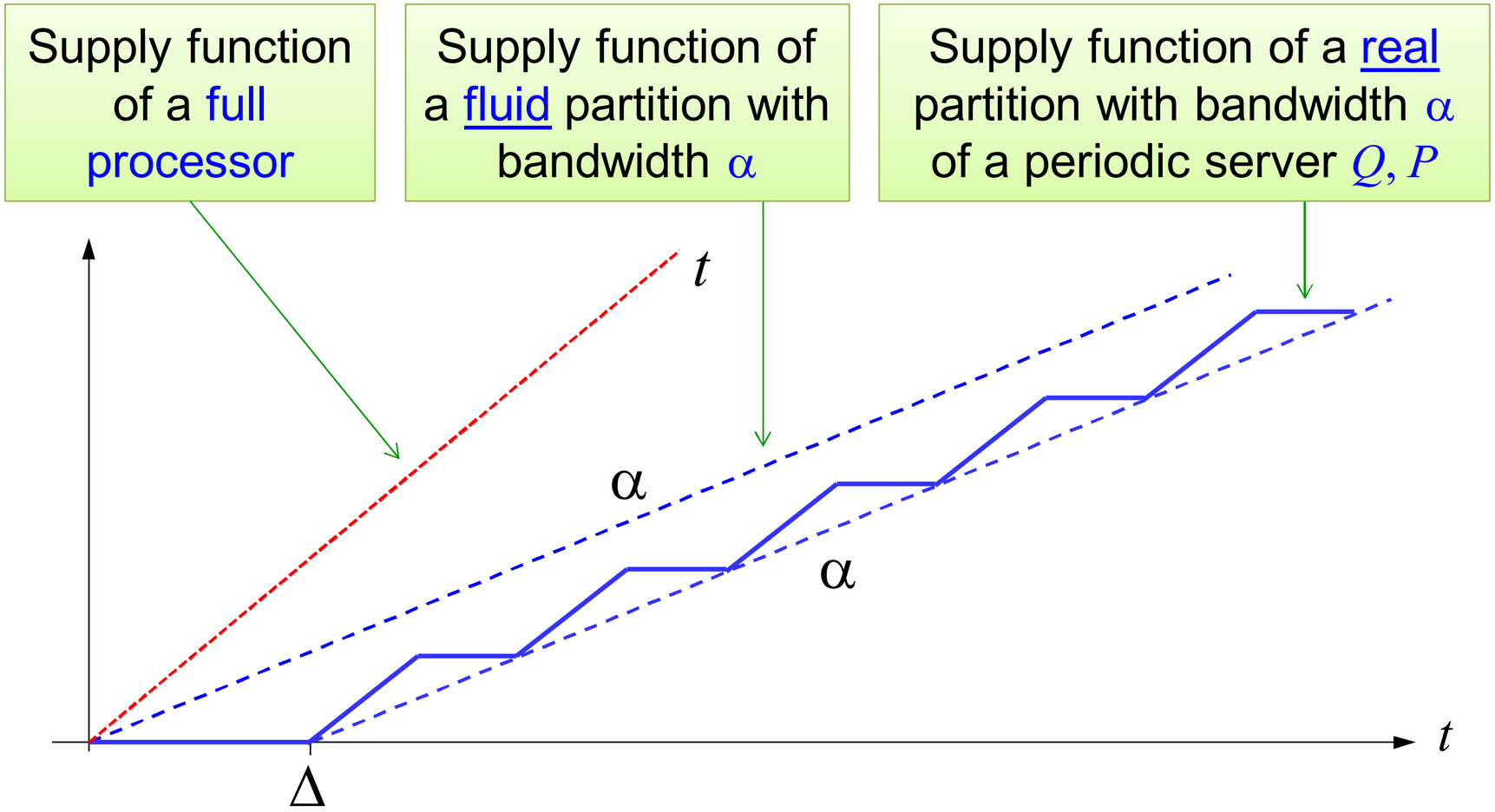
Example: Periodic Server

For a periodic server with budget Q_s and period P_s running at unknown priority, we have:



Observation

It is worth comparing the following supply functions:

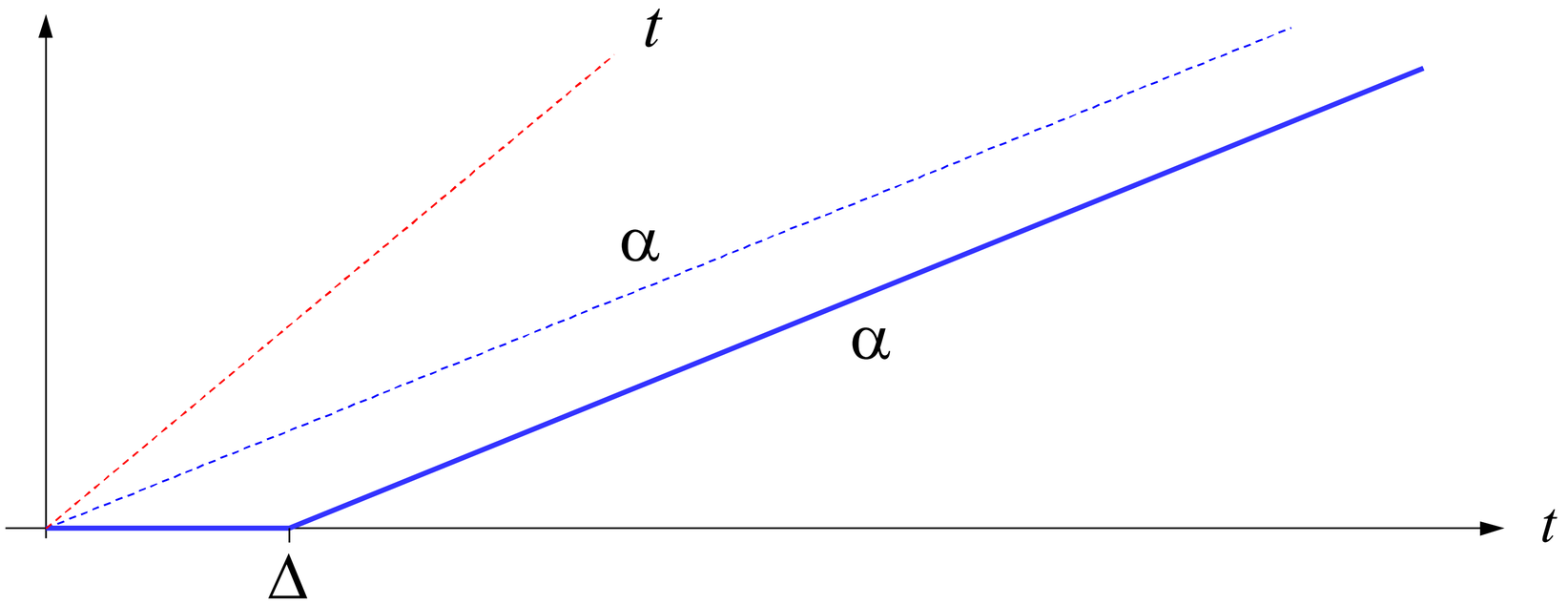


Observation

In a periodic server with bandwidth α, we have that:

$$\begin{cases} Q = \alpha P \\ \Delta = 2(P - Q) = 2P(1 - \alpha) \end{cases} \Rightarrow$$

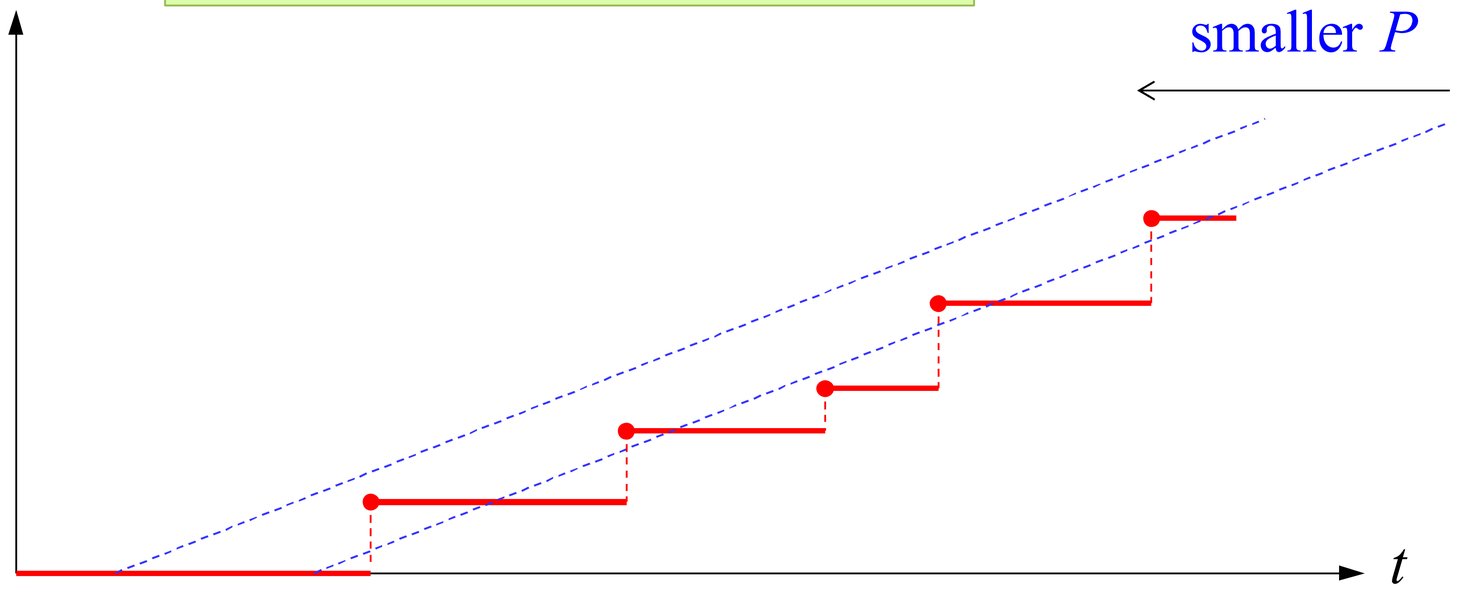
The delay Δ is proportional to the server period P



Observation

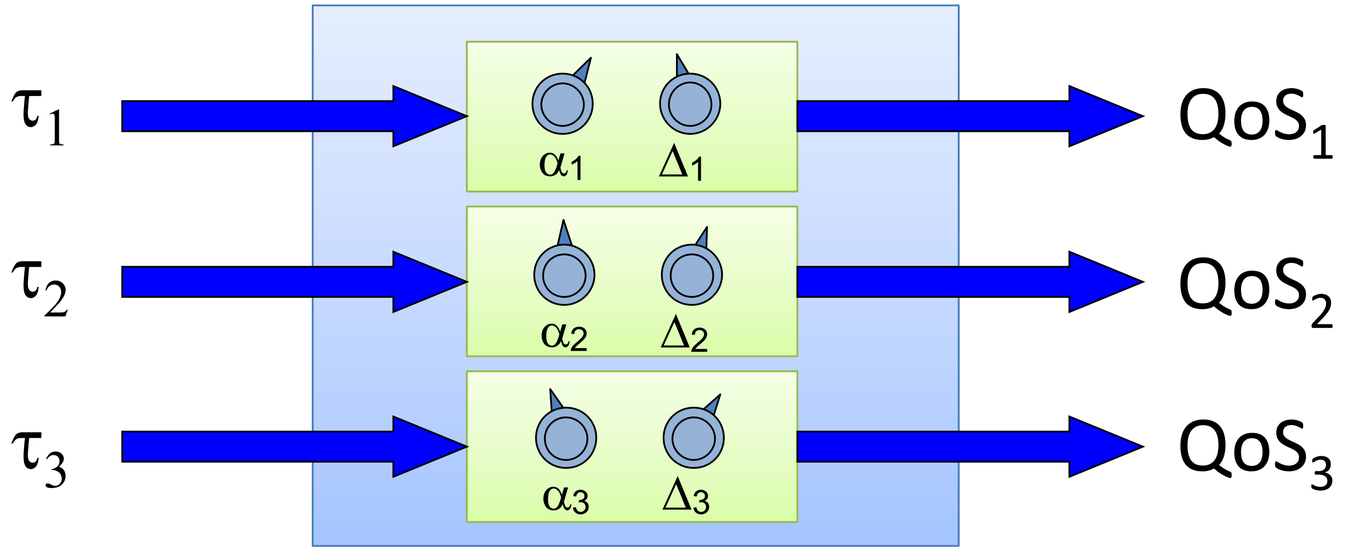
Note that, for a given bandwidth α, reducing P reduces the delay Δ and improves schedulability, tending to a fluid reservation, but …

smaller periods generates
higher runtime overhead



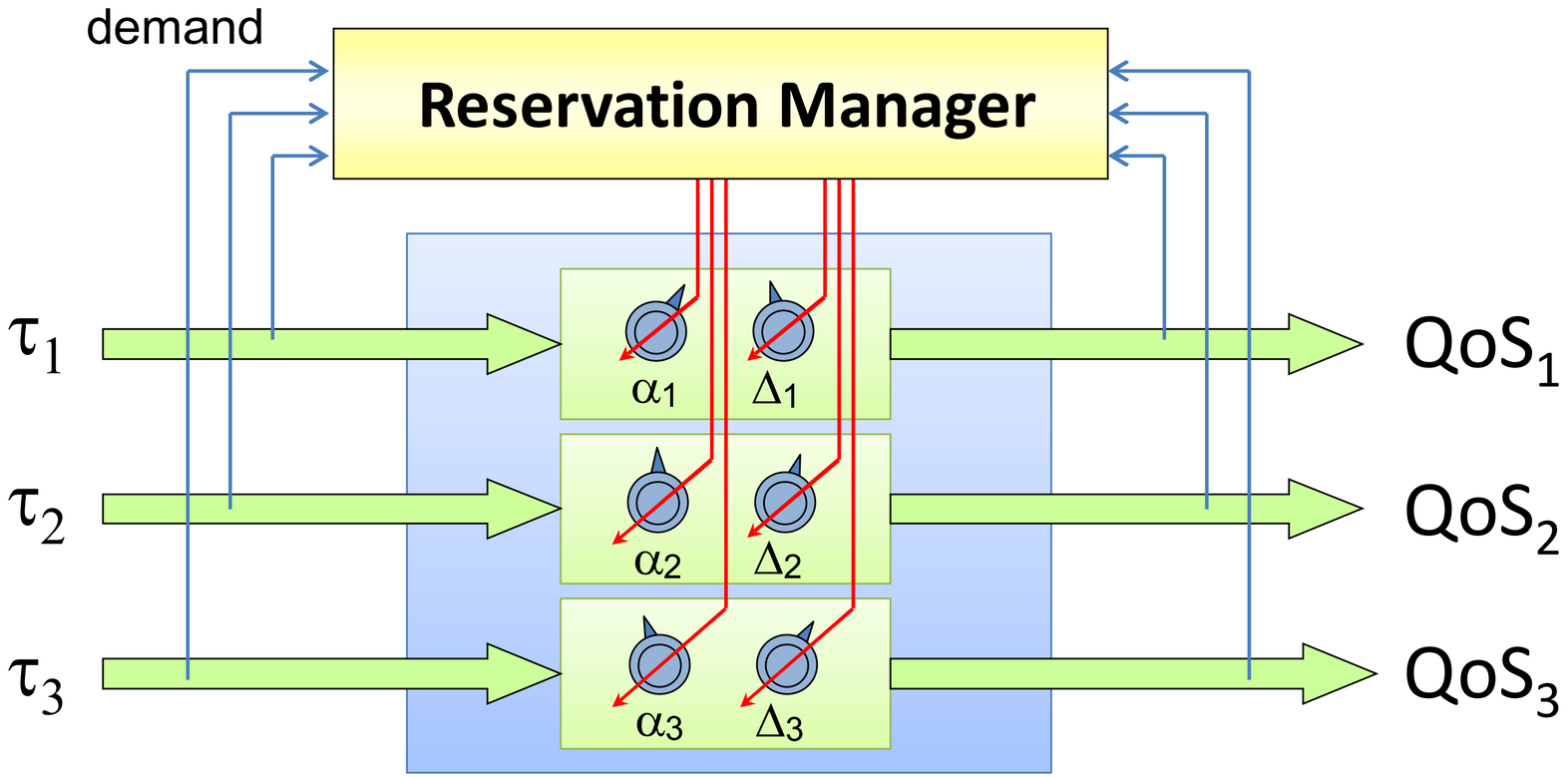
Reservation interface

Note that the (α, Δ) parameters offer an alternative interface, which is independent of the implementation mechanism (static partitions or Q-P server):



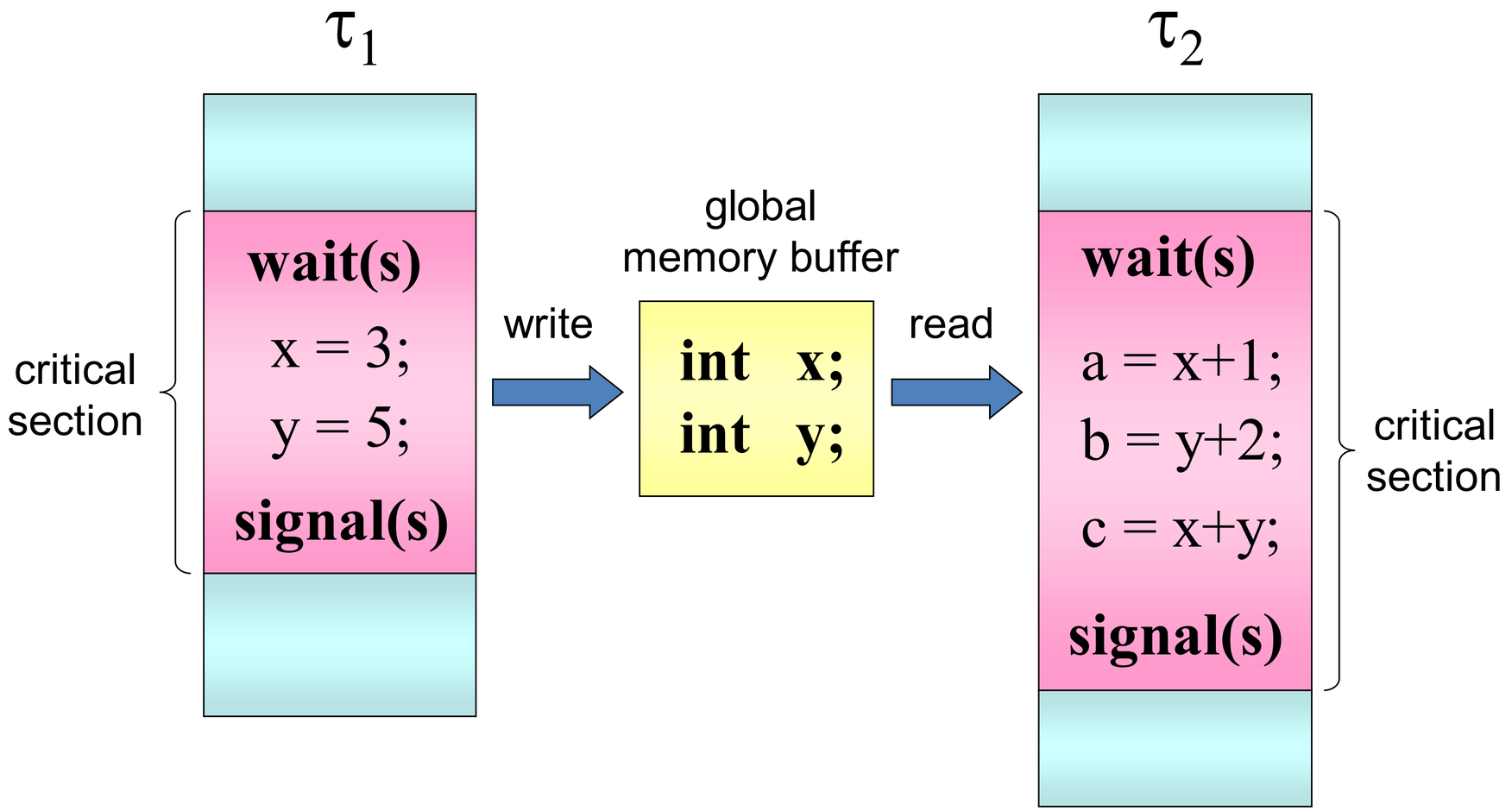
Adaptive QoS Management

Reservation parameters can be changed at run time by a Reservation Manager:



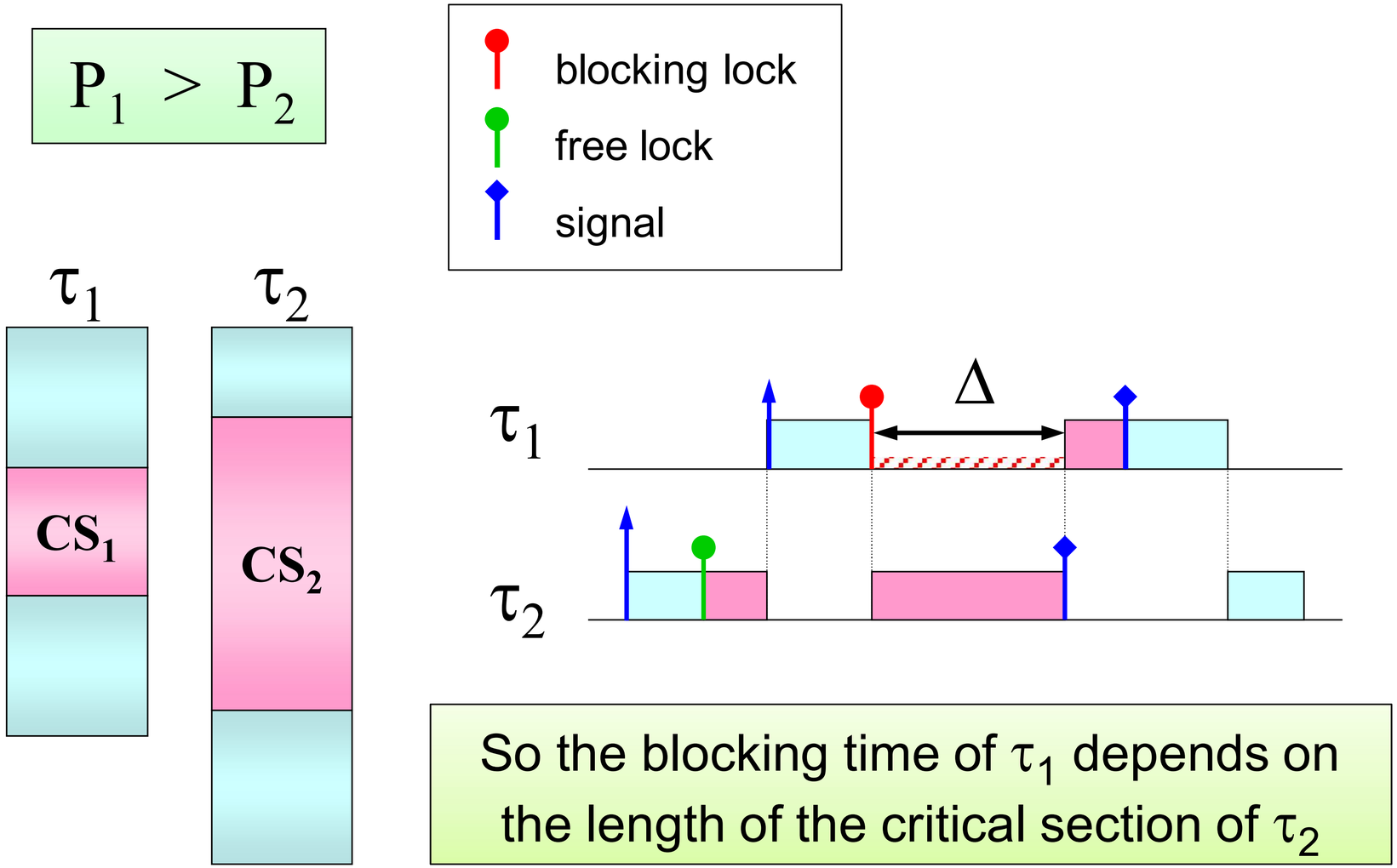
Handling
Shared Resources

Critical sections



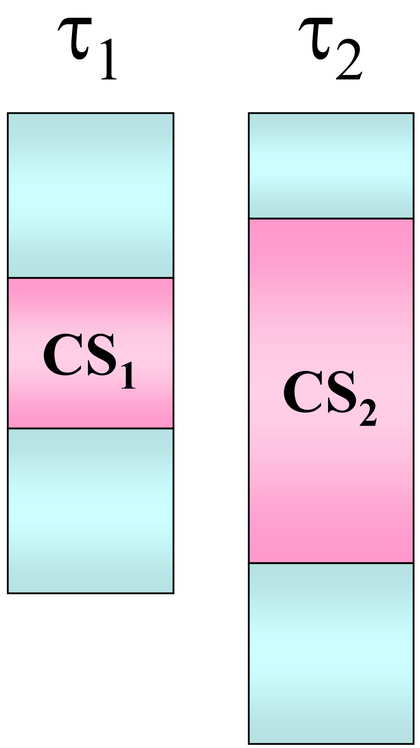
Blocking on a semaphore

$P_1 > P_2$

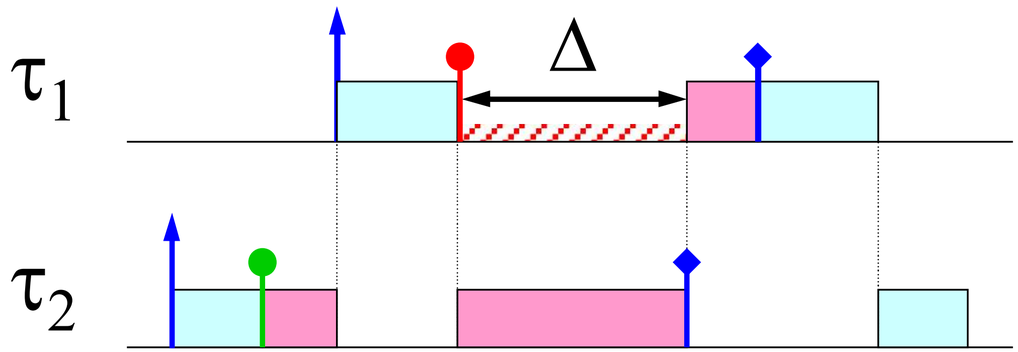


So the blocking time of τ_1 depends on the length of the critical section of τ_2

How long is blocking time?

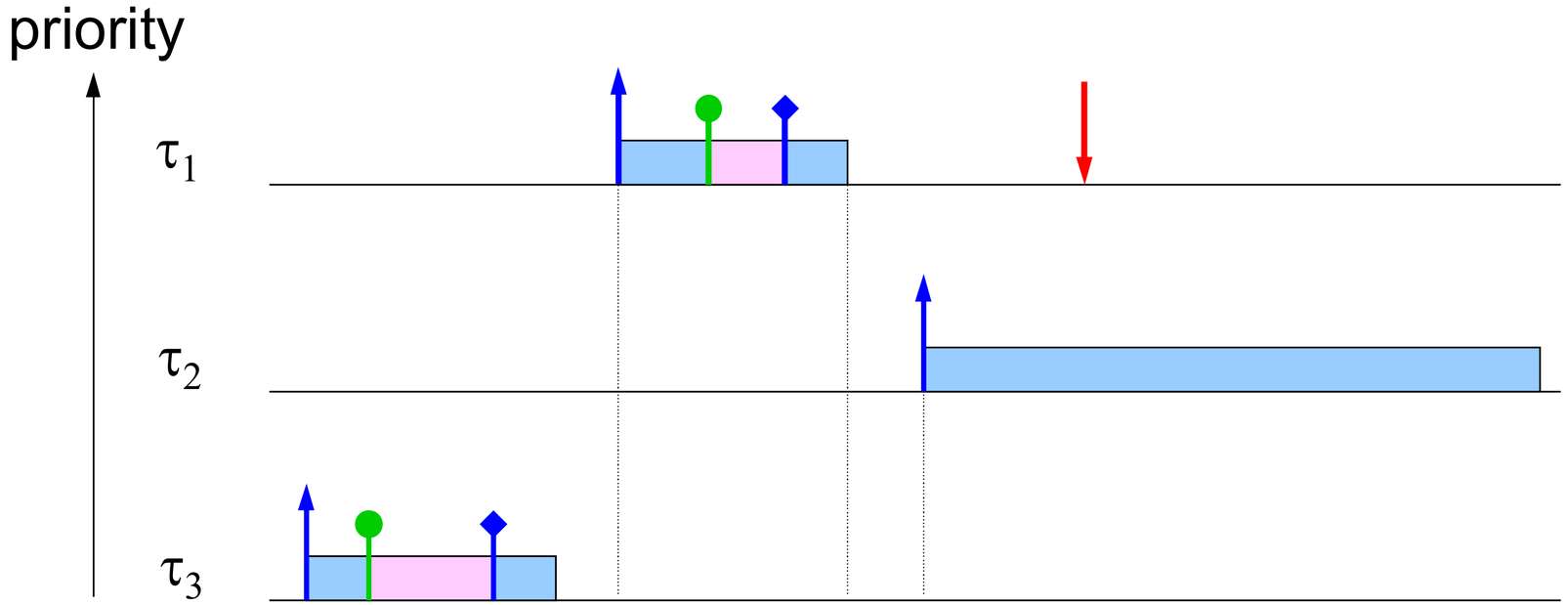


$P_1 > P_2$

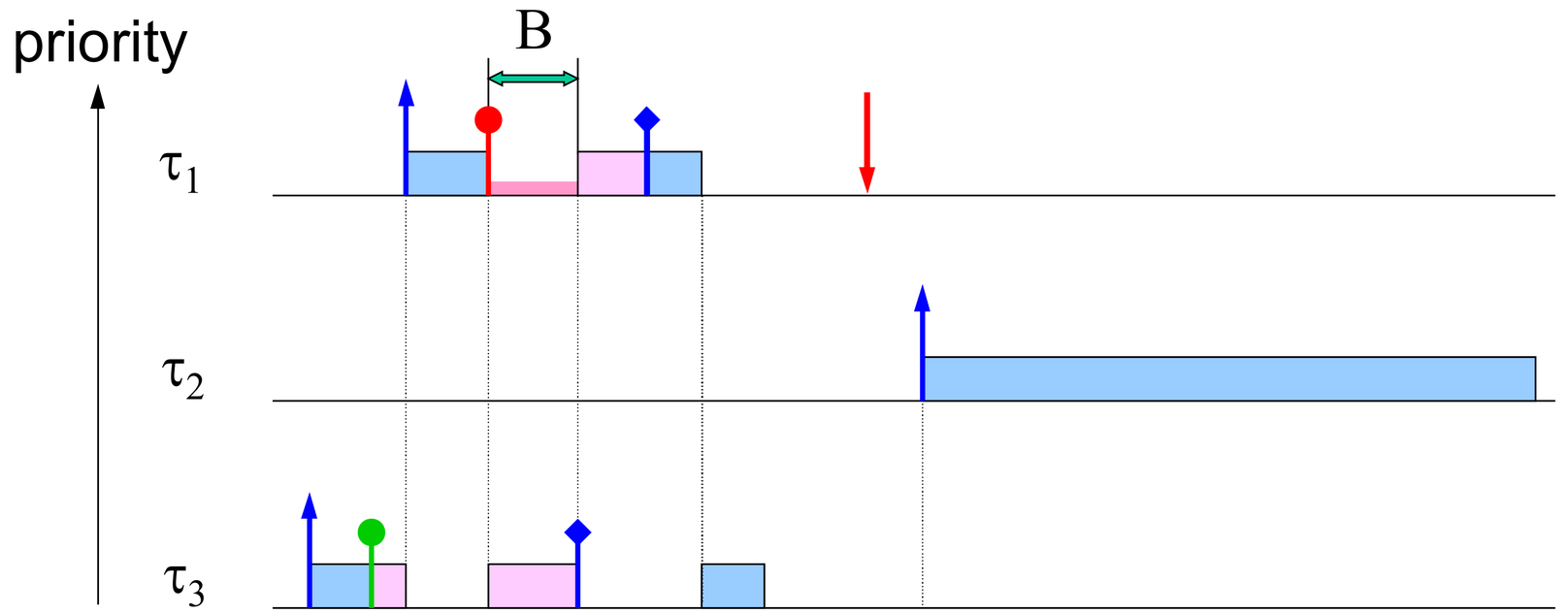


It seems that the maximum blocking time for τ_1 is equal to the length of the critical section (CS_2) of τ_2, but …

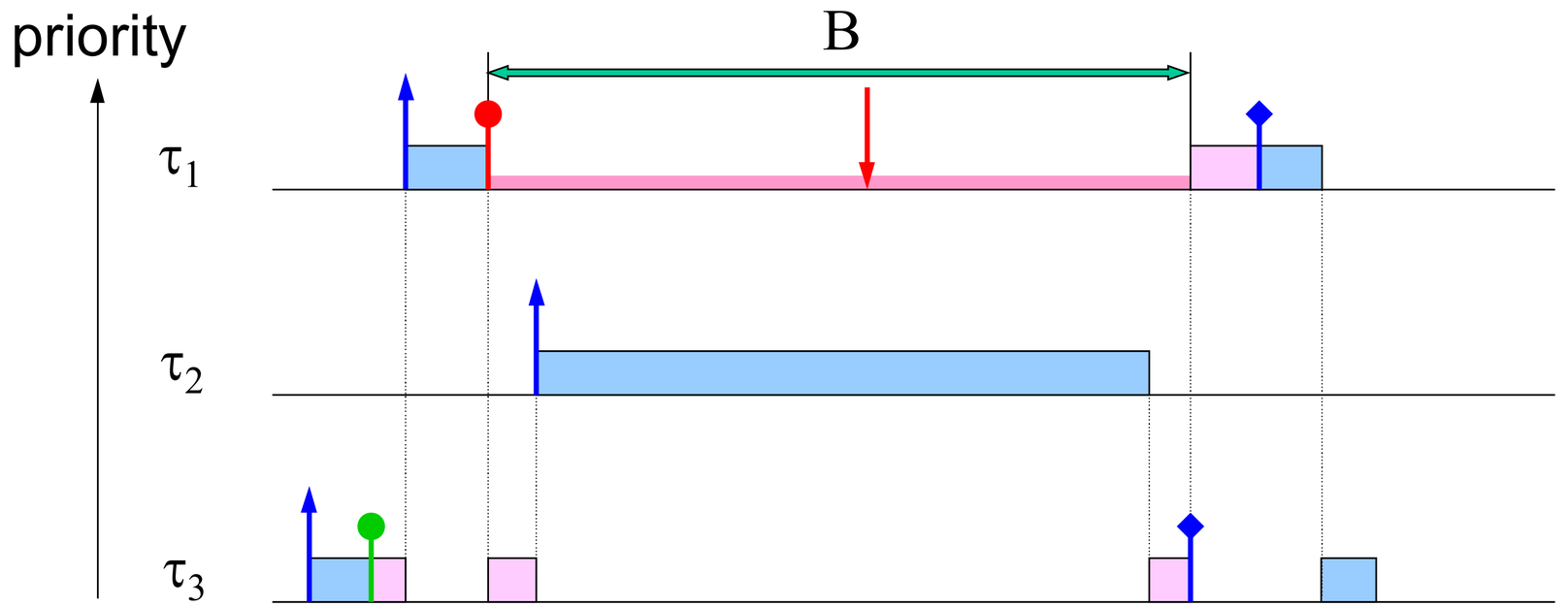
Schedule with no conflicts



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.

Protocol key aspects.

Access Rule:	Decides whether to block and when.
Progress Rule:	Decides how to execute inside a critical section.
Release Rule:	Decides how to order the pending requests of the blocked tasks.

Other aspects

Analysis:	estimates the worst-case blocking times.
Implementation:	finds the simplest way to encode the protocol rules.

Resource Access Protocols

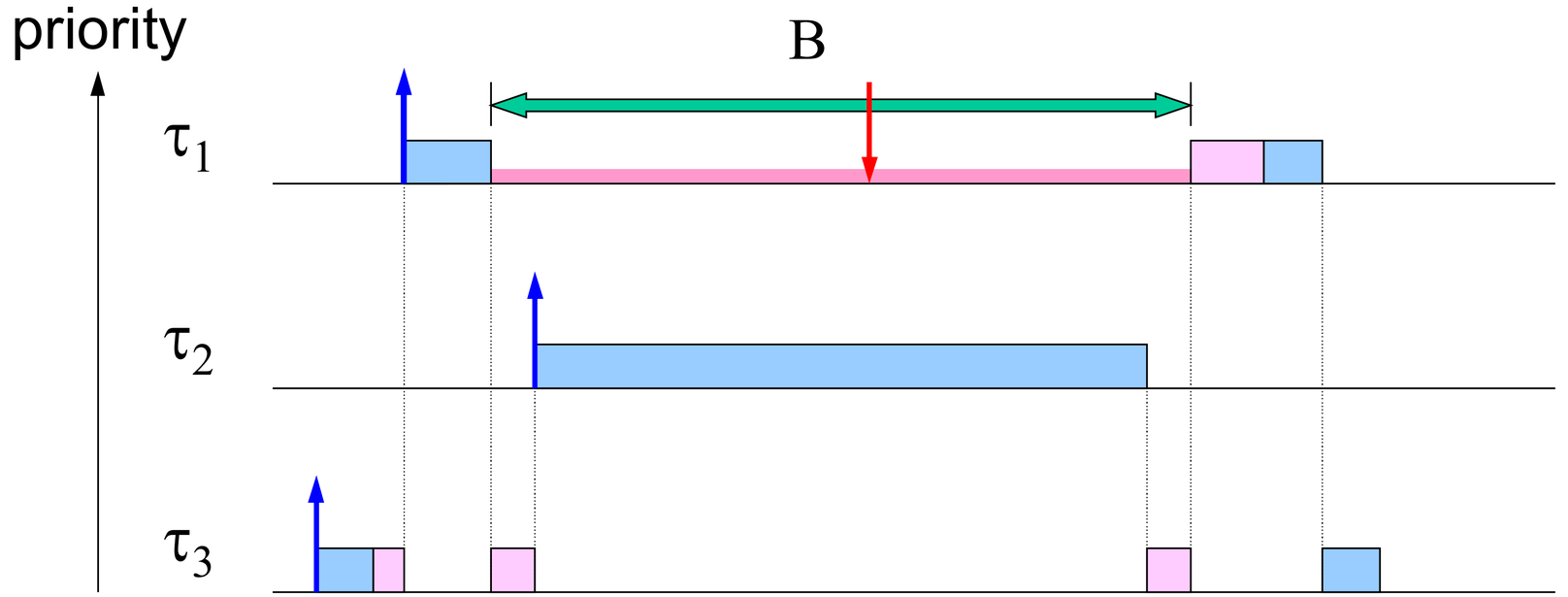
- Classical semaphores (No protocol)
- Non Preemptive Protocol (**NPP**)
- Highest Locker Priority (**HLP**)
- Priority Inheritance Protocol (**PIP**)
- Priority Ceiling Protocol (**PCP**)
- Stack Resource Policy (**SRP**)

Non Preemptive Protocol

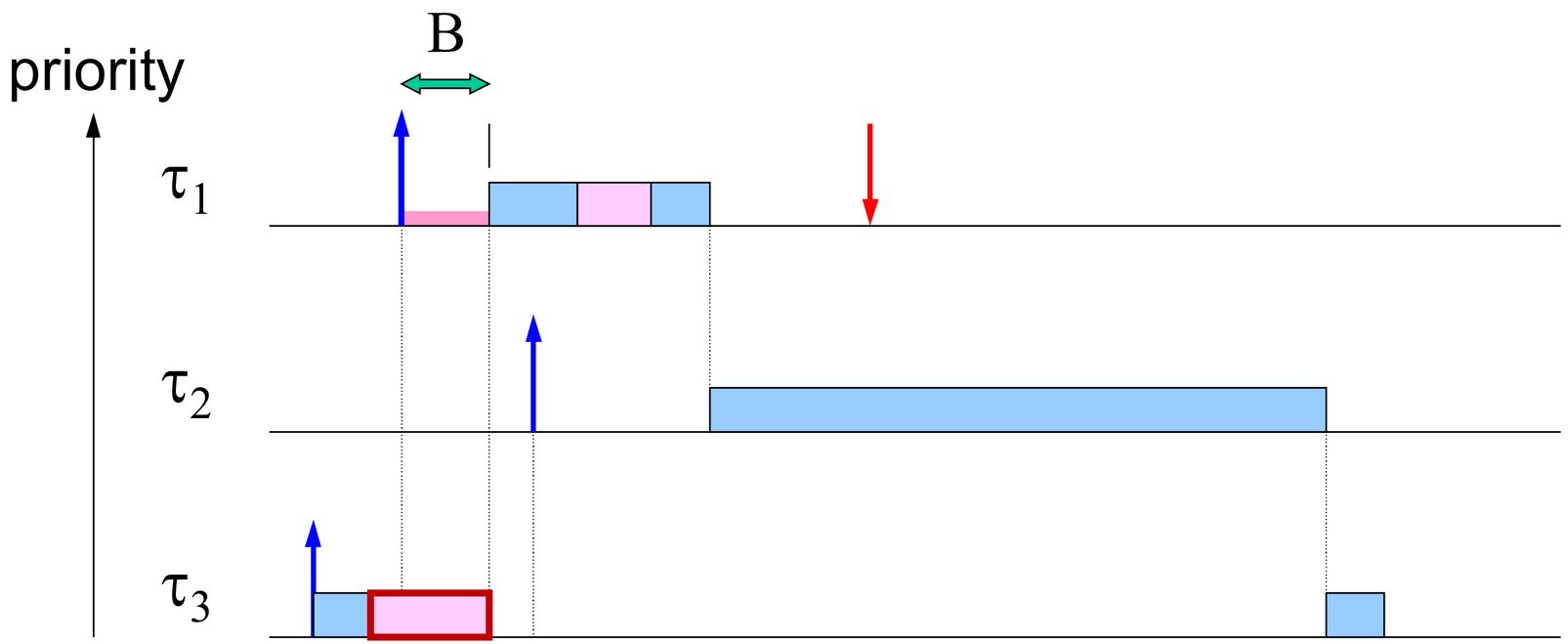
Access Rule:	A task never blocks at the entrance of a critical section, but at its activation time.
Progress Rule:	Disable preemption when executing inside a critical section.
Release Rule:	At exit, enable preemption so that the resource is assigned to the pending task with the highest priority.

Conflict on a critical section

(using classical semaphores)



NPP: example



NPP: implementation notes

Each task τ_i must be assigned two priorities:

- a nominal priority P_i (fixed) assigned by the application developer;
- a dynamic priority p_i (initialized to P_i) used to schedule the task and affected by the protocol.

Then, the protocol can be implemented by changing the behavior of the wait and signal primitives:

wait(s): $p_i = \max(P_1, \ldots, P_n)$

signal(s): $p_i = P_i$

NPP: pro & cons

ADVANTAGES: simplicity and efficiency.

- Semaphores queues are not needed, because tasks never block on a wait(s).
- Each task can block at most on a single critical section.
- It prevents deadlocks and allows stack sharing.
- It is transparent to the programmer.

PROBLEMS:

1. Tasks may block even if they do not use resources.
2. Since tasks are blocked at activation, blocking could be unnecessary (pessimistic assumption).

Priority Ceiling Protocol

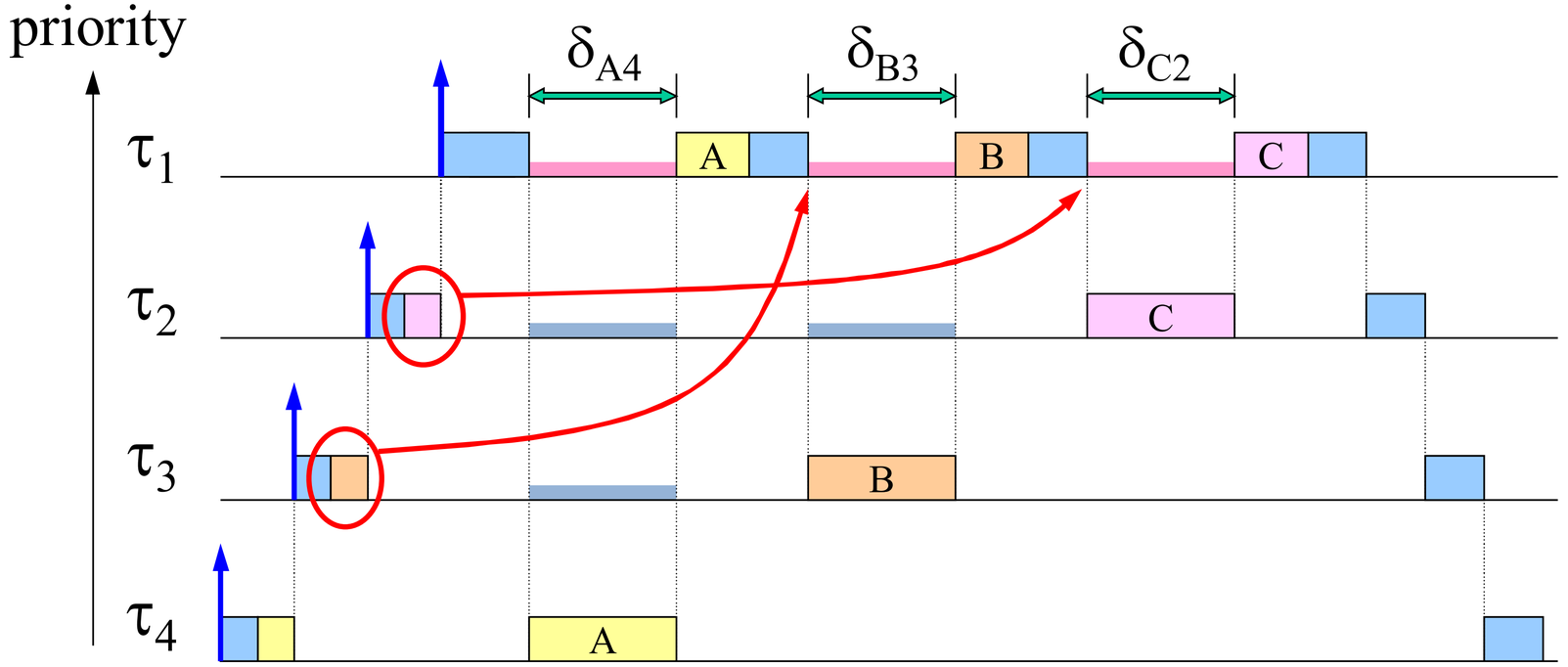
Access Rule:	A task can access a resource only if it passes the PCP access test.
Progress Rule:	Inside resource R, a task executes with the highest priority of the tasks blocked on R.
Release Rule:	At exit, the dynamic priority of the task is reset to its nominal priority P_i.

NOTE: **PCP** can be viewed as **PIP** + access test

Reference

L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization", *IEEE Transactions on Computers*, 39(9), September 1990.

Avoiding chained blocking



To avoid multiple blocking of τ_1 we must prevent τ_3 and τ_2 to enter their critical sections (even if they are free), because a low priority task (τ_4) is holding a resource used by τ_1.

Resource Ceilings

To keep track of resource usage by high-priority tasks, each resource is assigned a resource ceiling:

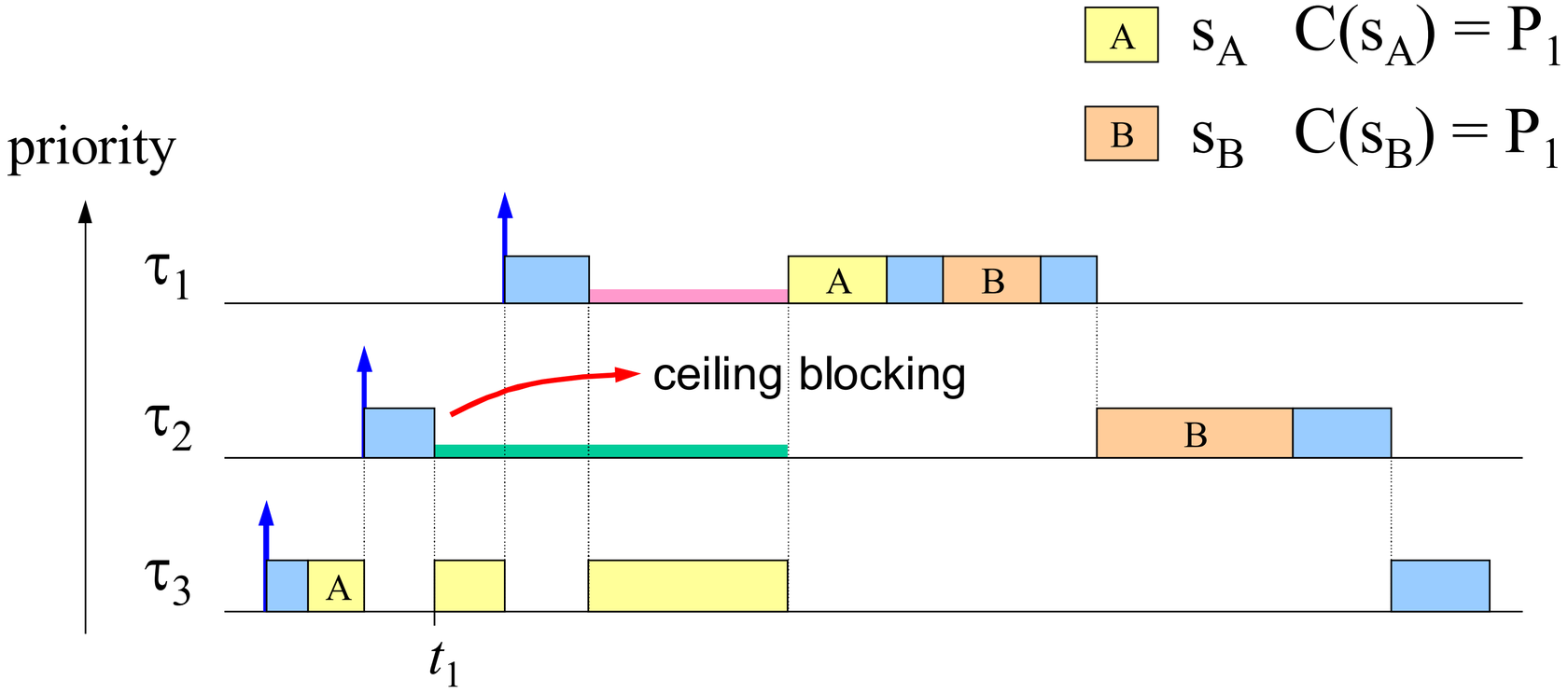
$$C(s_k) = max \{ P_i \mid \tau_i \ \ uses \ s_k \}$$

Then a task τ_i can enter a critical section only if its priority is higher than the maximum ceiling of the locked semaphores:

PCP access test

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

PCP: example



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

PCP: properties

Theorem 1

Under PCP, a task can block at most on a single critical section.

Theorem 2

PCP prevents chained blocking.

Theorem 3

PCP prevents deadlocks.

PCP: pro & cons

ADVANTAGES:

- It limits blocking to the length of a single critical section.
- It prevents deadlocks when using nested critical sections.

PROBLEMS:

- It is complex to implement (like PIP).
- It can create unnecessary blocking (it is pessimistic like HLP).
- It is not transparent to the programmer: resource ceilings must be specified in the source code.

Analysis under shared resources

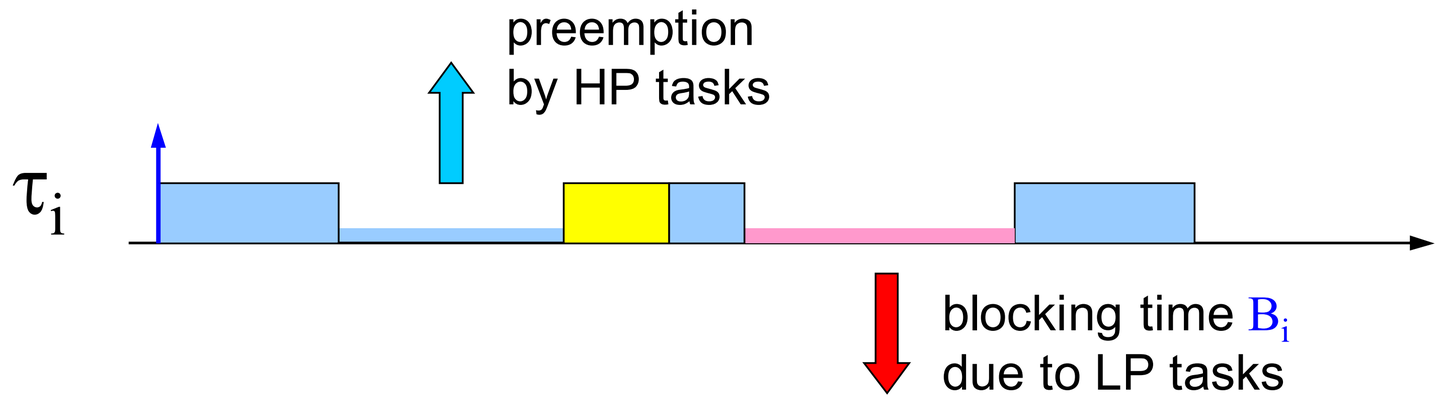
1. Select a scheduling algorithm to manage tasks and a protocol for accessing shared resources.

2. Compute the maximum blocking time B_i for each task.

3. Perform the guarantee test including the blocking terms.

Analysis under RM

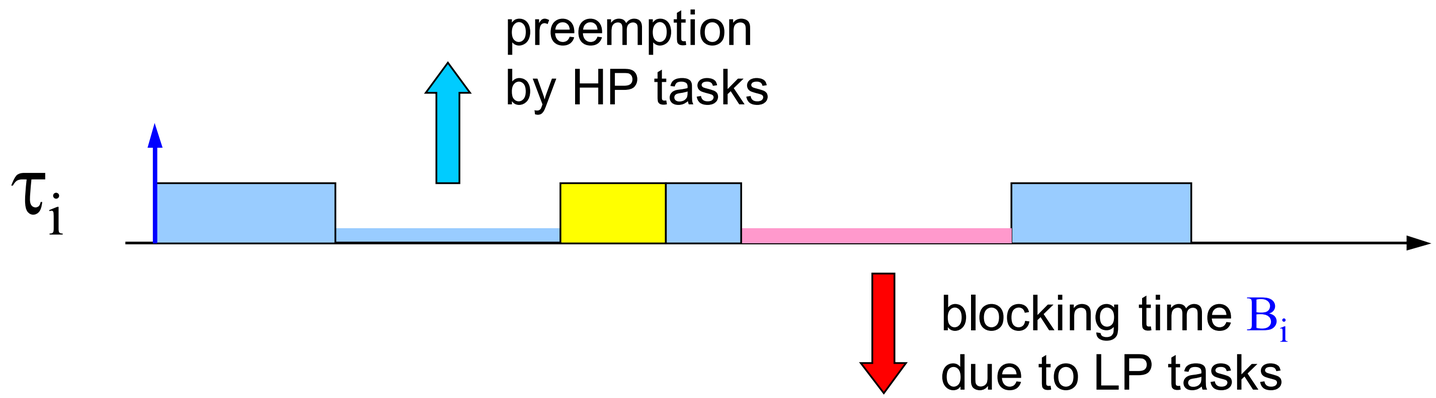
Once we identify the maximum blocking time B_i for each task, we can extend the LL schedulability test as follows:



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

Hyperbolic Bound

Similarly, we can extend the HB schedulability test as follows:



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

Response Time Analysis

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

Iterative solution:

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

iterate while

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

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 access protocol, called Stack Resource Policy (**SRP**) that works both under fixed priorities and EDF.

Resource ceiling for EDF

Each resource R_k is assigned a dynamic ceiling equal to the highest preemption level of the tasks that may block on R_k:

$$C_R(n_k) = \max \{0, \pi_i : n(R_k) < \mu_i(R_k)\}$$

NOTE: $C_R(n_k)$ increases only when a resource is locked

$C_R(n_k)$ decreases only when a resource is released

Schedulability analysis under EDF

When $D_i = T_i$

A task set is schedulable if

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

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

Schedulability analysis under EDF

When $D_i \leq T_i$

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

$$B(L) + \sum_{k=1}^{n} \left\lfloor \frac{L + T_k - D_k}{T_k} \right\rfloor C_k \leq L$$

where: $B(L) = \max \{ \delta_{jh} \mid (D_j > L) \text{ and } (D_h \leq L) \}$

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

$$\begin{cases} D_{max} = max(D_1, \ldots, D_n) \\ H = lcm(T_1, \ldots, T_n) \end{cases} \qquad L^* = \frac{\sum_{i=1}^{n} (T_i - D_i) U_i}{1 - U}$$

Comparison

Prot	prio.	pessim.	α_i	block. instant	trans-parent	dlock avoid.	stack sharing	impl
NPP	any	high	1	on arrival	YES	YES	YES	easy
HLP	fixed	medium	1	on arrival	NO	YES	YES	easy
PIP	fixed	low	min (n_i, m_i)	on access	YES	NO	NO	hard
PCP	fixed	low	1	on access	NO	YES	NO	hard
SRP	any	medium	1	on arrival	NO	YES	YES	easy