CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

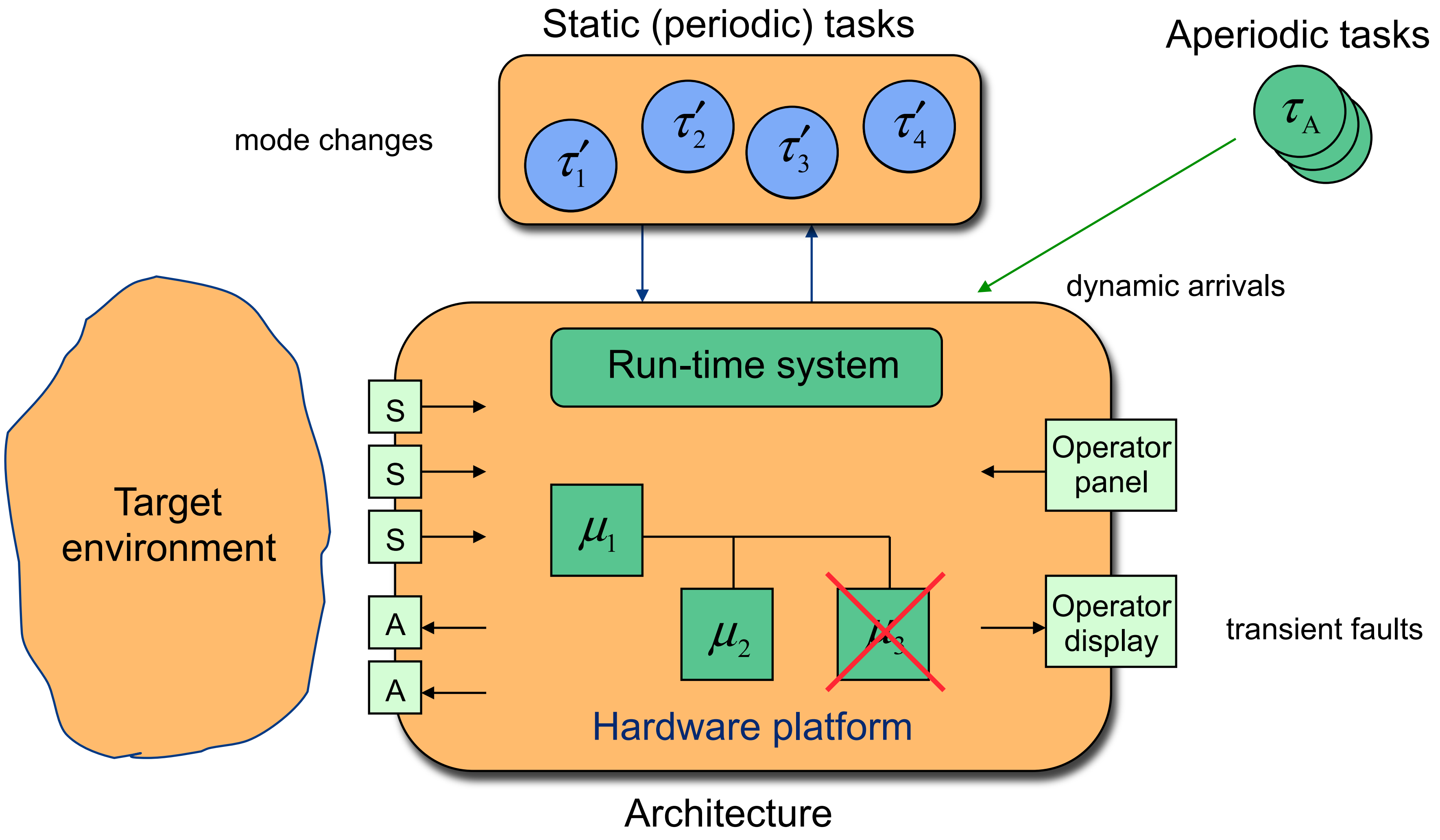
Dependable Real-Time Systems

Lecture #10

Professor Jan Jonsson

Department of Computer Science and Engineering
Chalmers University of Technology

Handling on-line changes



Handling on-line changes

Origins of on-line changes:

- Changing task characteristics:
  - Tasks execute shorter/longer than their assumed WCET.
  - Tasks increase/decrease the values of their static parameters as a result of, for example, a mode change.
- Dynamically arriving tasks:
  - Aperiodic tasks (with characteristics known *a priori*) arrive
  - New tasks (with characteristics not known *a priori*) enter the system at run-time.
- Changing hardware configuration:
  - Transient/intermittent/permanent hardware faults
  - Controlled hardware re-configuration (mode change)

Handling on-line changes

Consequences of on-line changes:

- Overload situations:
  - Changes in workload/architecture characteristics causes the accumulated processing demands from all tasks to exceed the capacities of the available processors.
  - Question: How do we reject certain tasks in a way such that the inflicted damage is minimized?
- Scheduling anomalies:
  - Changes in workload/architecture causes non-intuitive negative effects on system performance.
  - Question: Can we design sustainable feasibility tests that can guarantee that such a change does not result in a task set, that was previously deemed schedulable, becoming unschedulable?

Handling overload conditions

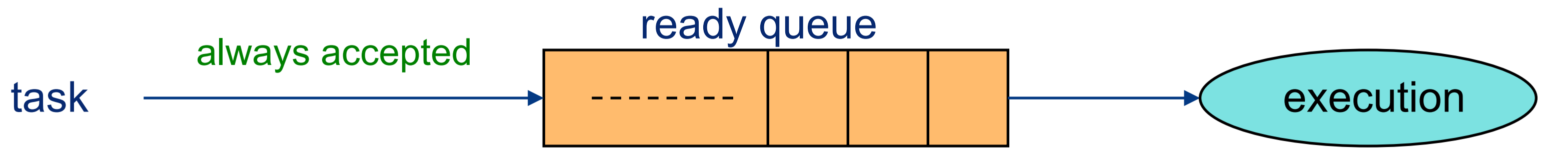
How do we handle a situation where the system becomes temporarily overloaded?

- Best-effort schemes:
  - No prediction for overload conditions.
- Guarantee schemes:
  - Processor load is controlled by continuous acceptance tests.
- Robust schemes:
  - Different policies for task acceptance and task rejection.
- Negotiation schemes:
  - Modifies workload characteristics within agreed-upon bounds.

Handling overload conditions

Best-effort schemes:

Includes those algorithms with no predictions for overload conditions. A new task is always accepted into the ready queue so the system performance can only be controlled through a proper priority assignment.



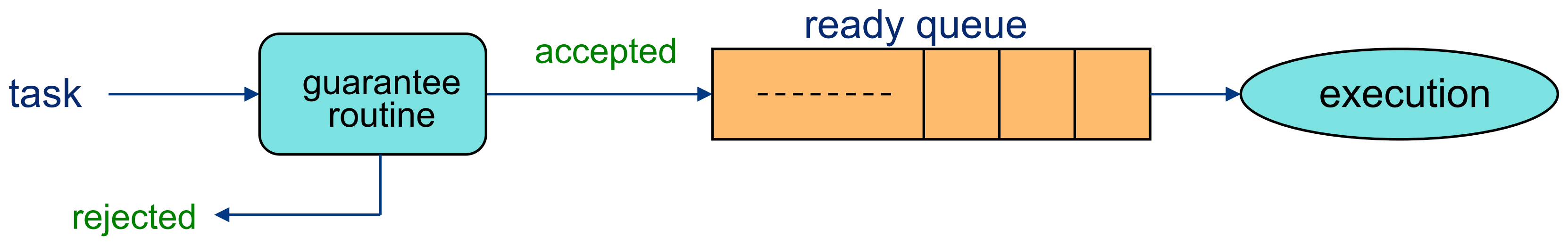
Example:

In case of overload, the tasks with the least value (importance, criticality) are removed.

Handling overload conditions

Guarantee schemes:

Includes those algorithms in which the load on the processor is controlled by an acceptance test executed at each task arrival. If the task set is found schedulable, the new task is accepted; otherwise, it is rejected.



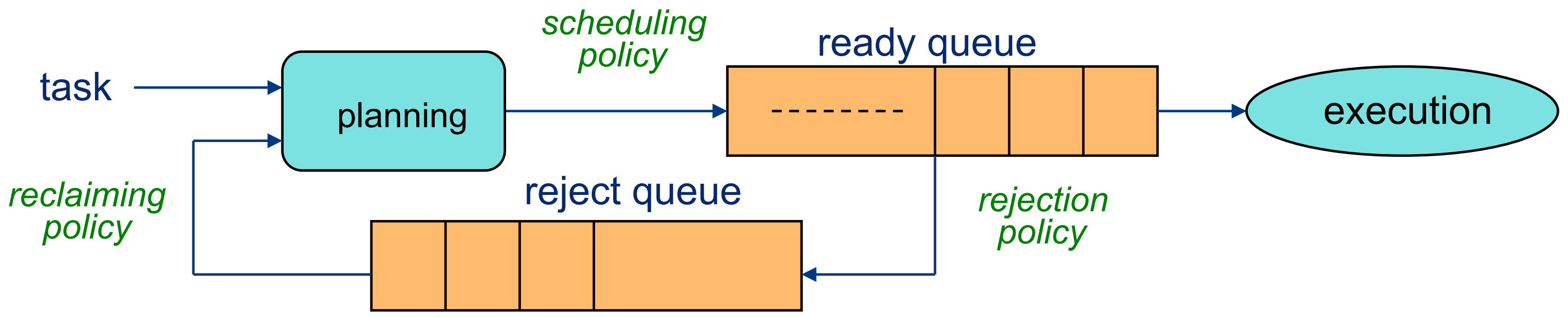
Example:

If a new task arrival cannot be guaranteed it is rejected (and distributed scheduling may be attempted).

Handling overload conditions

Robust schemes:

Includes those algorithms that separate timing constraints and importance by considering two different policies: one for task acceptance and one for task rejection.



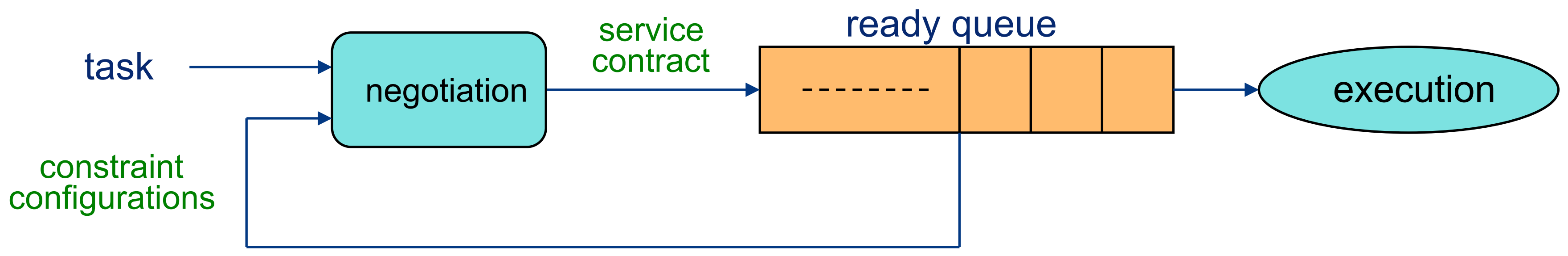
Example:

Consider deadline tolerance for acceptance of a task, but consider value (importance, criticality) for rejection.

Handling overload conditions

Negotiation schemes:

Includes those algorithms that attempt to modify timing constraints and/or importance within certain specified limits in an attempt to provide requested functionality.



Example:

Provide primary and alternate quality-of-service levels (constraint configurations) for each task, and in case of overload change to an alternate service level.

Handling overload conditions

Cumulative value:

The cumulative value of a scheduling algorithm A is a performance measure with the following quality:

$$\boxed{\Gamma_A = \sum_{i=1}^{n} v(f_i)}$$

Competitive factor:

A scheduling algorithm A has a <u>competitive factor</u> φ_A if and only if it can guarantee a cumulative value

$$\Gamma_A \geq \varphi_A \Gamma^*$$

where Γ^* is the cumulative value achieved by an optimal clairvoyant scheduler.

Handling overload conditions

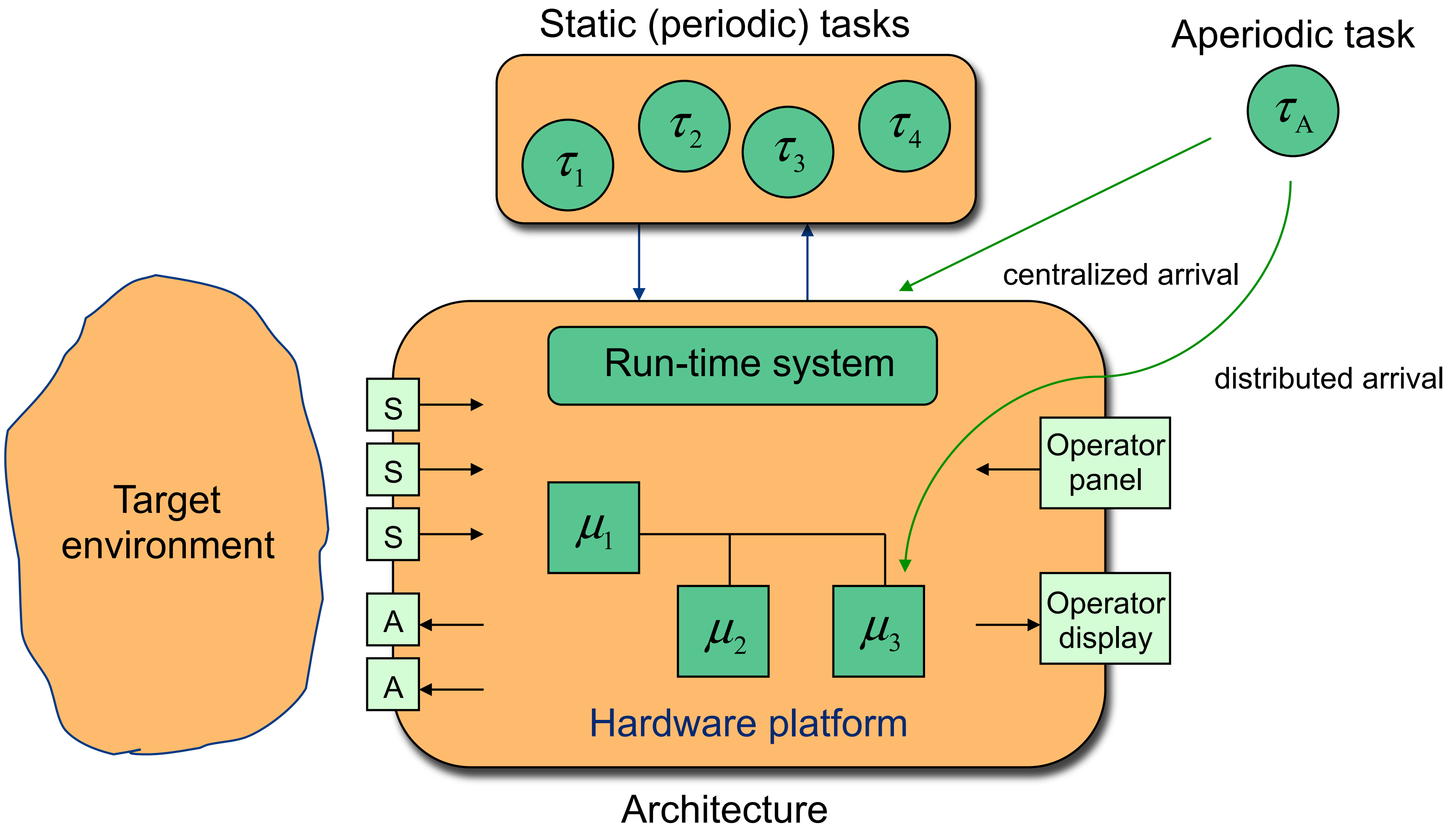
Limitations of on-line schedulers: (Baruah et al., 1992)

In systems where the loading factor is greater than 2 and tasks' values are proportional to their computation times, no on-line algorithm can guarantee a competitive factor greater than 0.25.

Observations:

- If the overload is of infinite duration, no on-line algorithm can guarantee a competitive factor greater than zero.
- Even for intermittent overloads, plain EDF has a <u>zero</u> competitive factor.
- The Dover algorithm has optimal competitive factor (Koren & Shasha, 1992)
- Having the best competitive factor among all on-line algorithms does not mean having the best performance in <u>any</u> load condition.

Handling aperiodic tasks



Handling aperiodic tasks

Aperiodic task model:

- Spatial:
  - The aperiodic task arrival is handled centralized; this is the case for multiprocessor servers with a common run-time system.
  - The aperiodic task arrival is handled distributed; this is the case for distributed systems with separate run-time systems.
- Temporal:
  - The aperiodic task is assumed to only arrive once; thus, it has no period.
  - The actual arrival time of an aperiodic task is not known in advance (unless the system is clairvoyant).
  - The actual parameters (e.g., WCET, relative deadline) of an aperiodic task may not be known in advance.

Handling aperiodic tasks

Approaches for handling aperiodic tasks:

- Server-based approach:
  - Reserve capacity to a "server task" that is dedicated to handling aperiodic tasks.
  - All aperiodic tasks are accepted, but can only be handled in a best-effort fashion ⇒ no guarantee on schedulability
- Server-less approach:
  - A schedulability test is made on-line for each arriving aperiodic task ⇒ guaranteed schedulability for accepted task.
  - Rejected aperiodic tasks could either be dropped or forwarded to another processor (in case of multiprocessor systems)

Handling aperiodic tasks

Challenges in handling aperiodic tasks:

- Server-based approach:
  - How do we reserve enough capacity to the server task without compromising schedulability of hard real-time tasks, while yet offering good service for future aperiodic task arrivals?
- Server-less approach:
  - How do we design a schedulability test that accounts for arrived aperiodic tasks (remember: they do not have periods)?
  - To what other processor do we off-load a rejected aperiodic task (in case of multiprocessor systems)?

Aperiodic servers

Handling (soft) aperiodic tasks on uniprocessors:

- Static-priority servers:
  - Handles aperiodic/sporadic tasks in a system where periodic tasks are scheduled based on a static-priority scheme (RM).
- Dynamic-priority servers:
  - Handles aperiodic/sporadic tasks in a system where periodic tasks are scheduled based on a dynamic-priority scheme (EDF).
- Slot-shifting server:
  - Handles aperiodic/sporadic tasks in a system where periodic tasks are scheduled based on a time-driven scheme.

Primary goal: to minimize the response times of aperiodic tasks in order to increase the likelihood of meeting their deadlines.

Static-priority servers

Background scheduling:

Schedule aperiodic activities in the background; that is, when there are no periodic task instances to execute.

Advantage:

– Very simple implementation

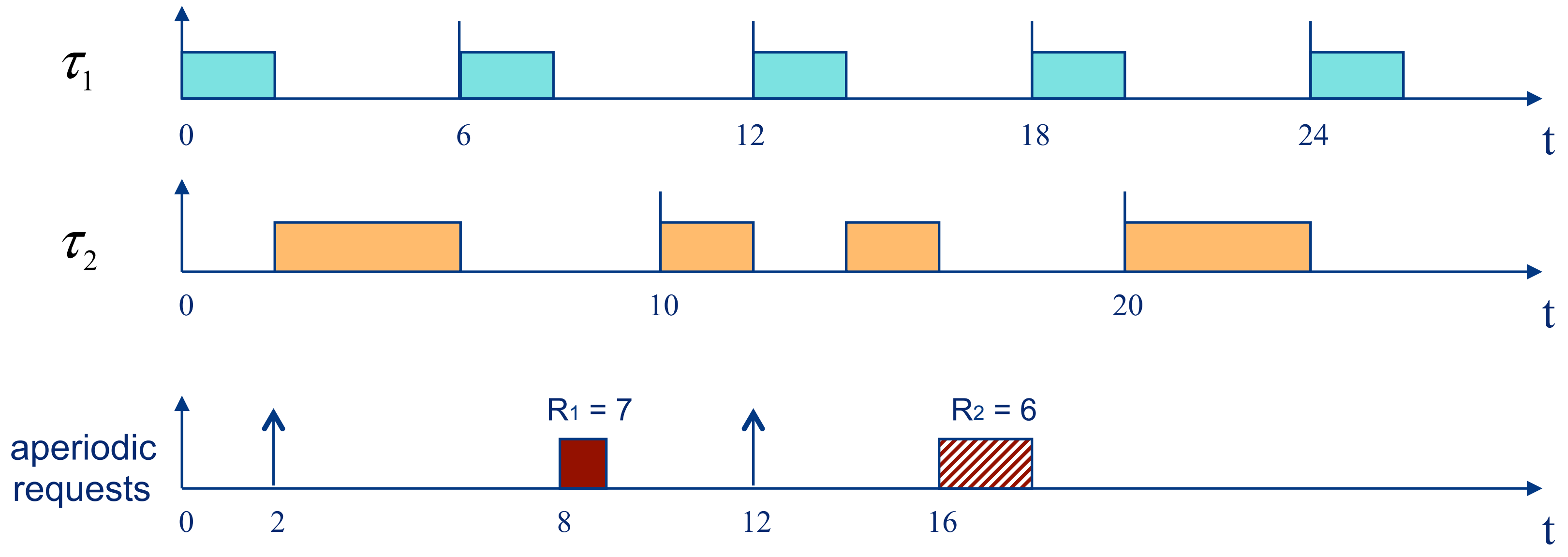
Disadvantage:

– Response time can be too long

Static-priority servers

Background scheduling:

$$\tau_1 = \{ C_1 = 2, T_1 = 6 \}$$
$$\tau_2 = \{ C_2 = 4, T_2 = 10 \}$$
$$U = 2/6 + 4/10 \approx 0.73$$



Static-priority servers

Polling Server (PS): (Lehoczky, Sha & Strosnider, 1987)

Service aperiodic tasks using a dedicated task with a period T_s and a capacity C_s.

If no aperiodic tasks need service in the beginning of the PS period, PS suspends itself until beginning of next period. Unused server capacity is used by periodic tasks.

Advantage:

– Much better average response time

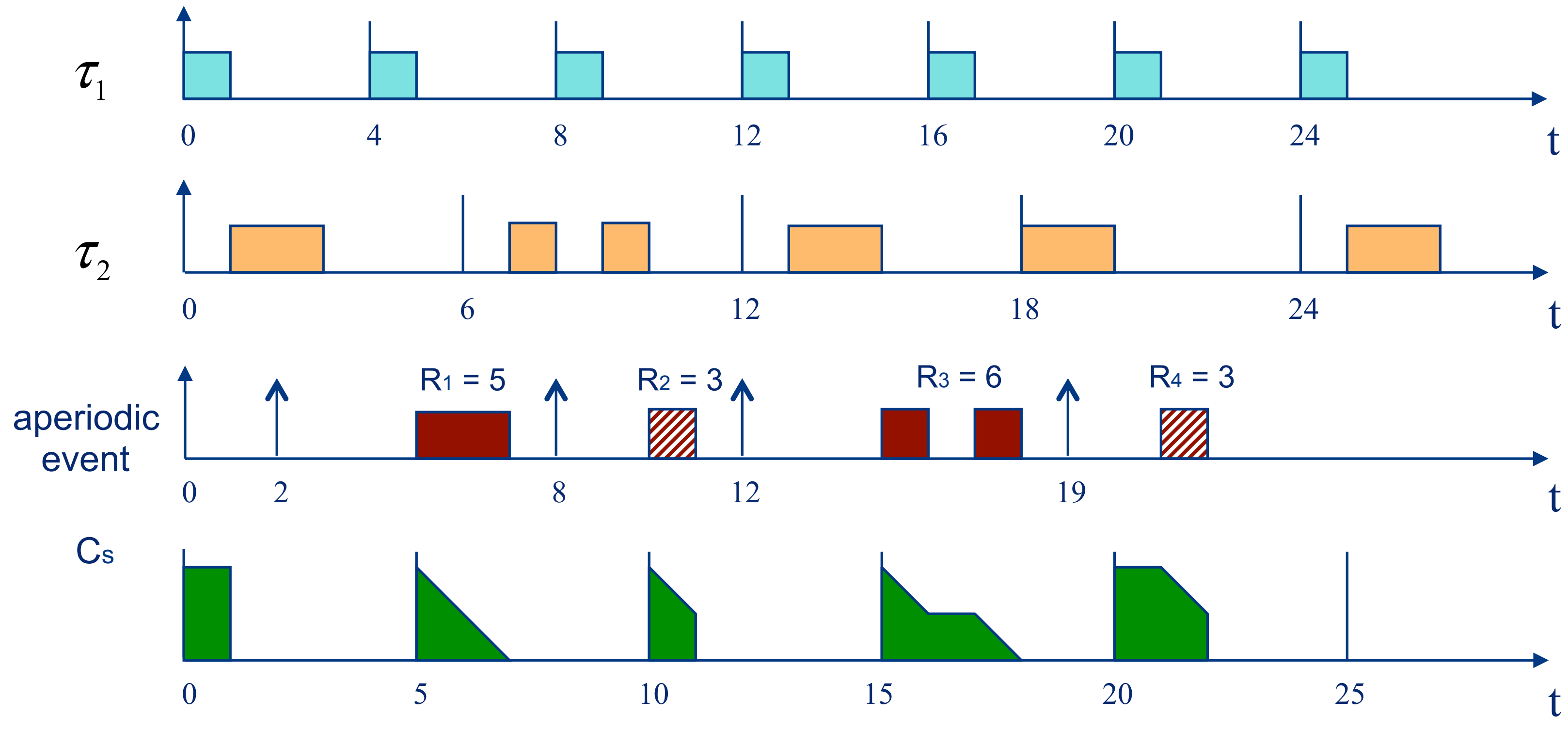
Disadvantage:

– If no aperiodic request occurs at beginning of server period, the entire server capacity for that period is lost.

Static-priority servers

Polling Server:

$\tau_1 = \{1,4\}$ $\tau_S = \{2,5\}$
$\tau_2 = \{2,6\}$ $U \approx 0.98$



Static-priority servers

Deferrable Server (DS): (Lehoczky, Sha & Strosnider, 1987)

Service aperiodic tasks using a dedicated task with a period T_s and a capacity C_s.

Server maintains its capacity until end of period so that requests can be serviced as capacity is not exhausted.

Advantage:

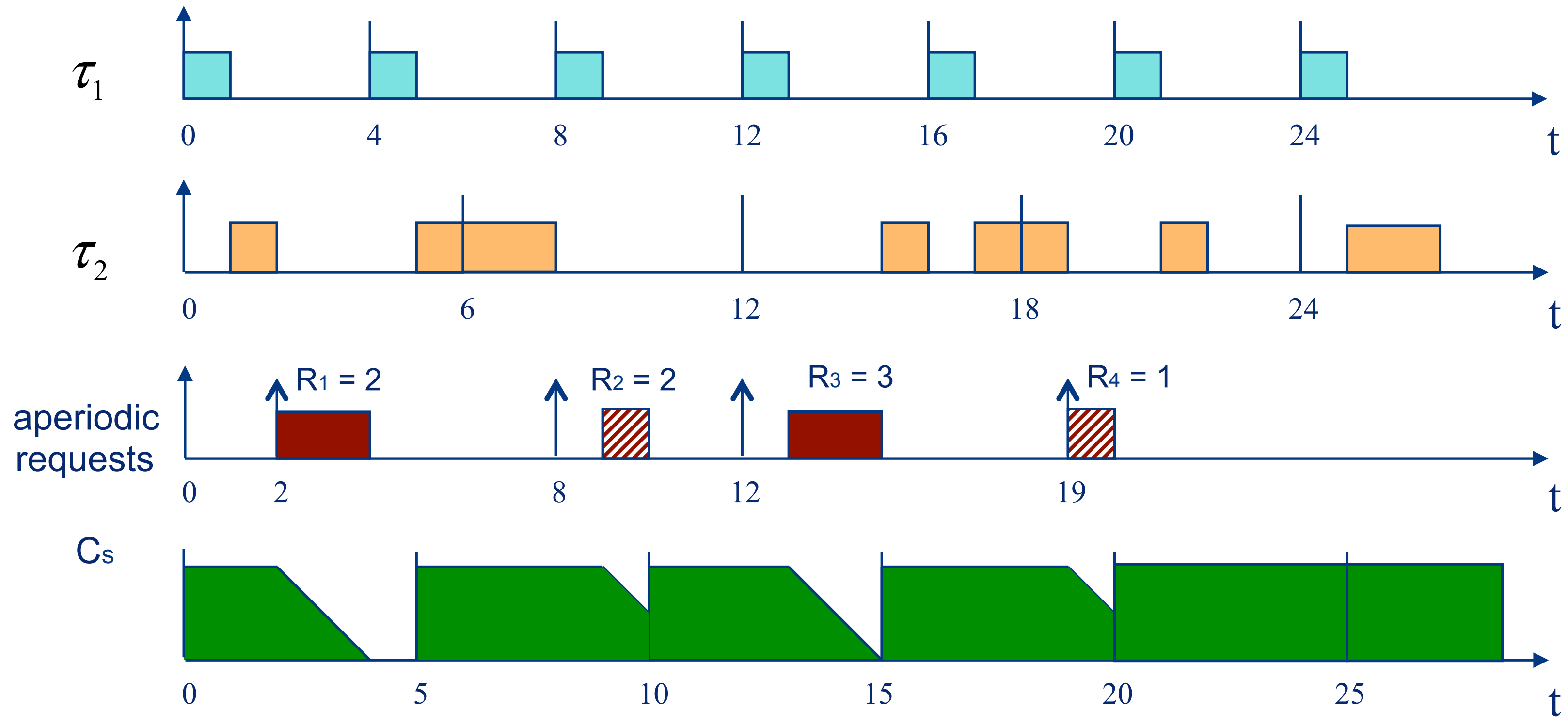
– Even better average response time because capacity is not lost

Static-priority servers

Deferrable Server:

$\tau_1 = \{1,4\}$ $\tau_S = \{2,5\}$

$\tau_2 = \{2,6\}$ $U \approx 0.98$



Static-priority servers

Feasibility test for RM + DS:

A set of n periodic tasks and one aperiodic server are schedulable using RM if the processor utilization does not exceed:

$$U_{RM+DS} = U_S + n\left(\left(\frac{U_S + 2}{2U_S + 1}\right)^{1/n} - 1\right)$$

Static-priority servers

Feasibility test for RM + DS:

Rules-of-thumb:

$$n \rightarrow \infty \Rightarrow U_{RM+DS} \approx 0.652 \quad \left(\text{for } U_S = 0.186\right)$$

$$U_{RM+DS} \leq U_{RM} \quad \left(\text{for } U_S \leq 0.4\right)$$
$$U_{RM+DS} > U_{RM} \quad \left(\text{for } U_S > 0.4\right)$$

(Other) Static-priority servers

Priority Exchange Server: (Lehoczky, Sha & Strosnider, 1987)

Preserves its capacity by temporarily exchanging it for the execution time of a lower-priority periodic task.

Sporadic Server: (Sprunt, Sha & Lehoczky, 1989)

Replenishes its capacity only after it has been consumed by aperiodic task execution.

Slack Stealing: (Lehoczky & Ramos-Thuel, 1992)

Does not use a periodic server task. Instead, it employs a run-time mechanism that can delay the execution of periodic tasks to make room for a aperiodic task, without violating the priority ordering of those tasks.

Static-priority servers

Non-existence of optimal servers: (Tia, Liu & Shankar, 1995)

For any set of periodic tasks ordered on a given static-priority scheme and aperiodic requests ordered according to a given aperiodic queuing discipline, there does not exist any on-line algorithm that minimizes the response time of every soft aperiodic request.

For any set of periodic tasks ordered on a given static-priority scheme and aperiodic requests ordered according to a given aperiodic queuing discipline, there does not exist any on-line algorithm that minimizes the average response time of the soft aperiodic requests.

Dynamic-priority servers

Dynamic Priority Exchange Server: (Spuri & Buttazzo, 1994)

Preserves its capacity by temporarily exchanging it for the execution time of a lower-priority (longer deadline) task.

Dynamic Sporadic Server: (Spuri & Buttazzo, 1994)

Replenishes its capacity only after it has been consumed by aperiodic task execution.

Total Bandwidth Server: (Spuri & Buttazzo, 1994)

Assign a (possibly earlier) deadline to each aperiodic task and schedule it as a normal task. Deadlines are assigned such that the overall processor utilization of the aperiodic load never exceeds a specified maximum value U_s.

Slot-shifting server

Slot-Shifting Server: (Fohler, 1995)

Schedules aperiodic tasks in the unused time slots in a schedule generated for time-driven dispatching.

Associated with each point in time is a spare capacity that indicates by how much the execution of the next periodic task can be shifted in time without missing any deadline.

Whenever an aperiodic task arrives, task instances in the static workload may be shifted in time – by as much as the spare capacity indicates – in order to accommodate the new task.

Scheduling anomalies

Scheduling anomaly: A seemingly positive change in the system (reducing load or adding resources) causes a non-intuitive decrease in performance.

State-of-the-art :

- Uniprocessor systems:
  - Anomalies only found for non-preemptive scheduling
- Multiprocessor systems:
  - Richard's anomalies for non-preemptive scheduling
  - Execution-time-based anomalies for preemptive scheduling
  - Period-based anomalies for preemptive scheduling

Scheduling anomalies

Richard's anomalies: (Graham, 1969)

Assumptions:

- Non-preemptive scheduling
- Precedence constraints
- Restricted migration (individual task instances cannot migrate)
- Fixed execution times

Task completion times may increase as a result of:

- Changing the task priorities
- Increasing the number of processors
- Reducing task execution times
- Weakening the precedence constraints

Scheduling anomalies

Execution-time-based anomalies: (Ha & Liu, 1994)

Assumptions:

- Preemptive scheduling
- Independent tasks
- Restricted migration (individual task instances cannot migrate)
- Fixed execution times

Task completion times may increase as a result of:

- Reducing task execution times

Scheduling anomalies

Period-based anomalies: (Andersson & Jonsson, 2000)

Assumptions:

- Preemptive scheduling
- Independent tasks
- Full migration
- Fixed execution times

A task's completion time may increase as a result of:

- Increasing the period of a higher-priority task
- Increasing the period of the task itself

Consequently, sporadic tasks may not have the same worst-case schedulability behavior as periodic tasks! (this is in contrast to the uniprocessor case)