Resource Management

(slides are based on Prof. Dr. Jian-Jia Chen and http://www.freertos.org)

Anas Toma

LS 12, TU Dortmund

December 20, 2018

[Reference: Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Giorgio C. Buttazzo]

Outline

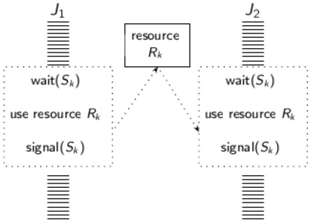
Terminology

Resource: Any software structure that can be used by a task.

- e.g. data structures, variables, memory arias, files, registers, I/O units, the processor, etc.
- Types:
  - **Private**: dedicated to one task.
  - **Shared**: used by more than one task.

Critical section: A piece of code that accesses a shared resource which must not be concurrently accessed by more than one thread of execution.

- Synchronization mechanism is required at the entry and the exit of the critical section to ensure exclusive use.

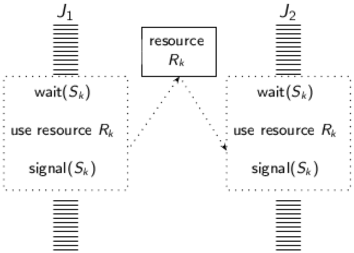


Terminology (cont.)

- Any task that needs to enter a critical section must wait until no other task is **holding** the resource, i.e., the critical section becomes **free**.
  - Waiting task $\rightarrow$ **blocked**

Exclusive resource: A shared resource protected against concurrent accesses.

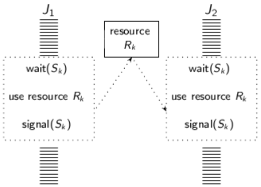
- Synchronization is required $\rightarrow$ *Semaphore* can be used
  - e.g. semaphore S_k protects the resource R_k in the figure
  - functions: *wait()* and *signal()*



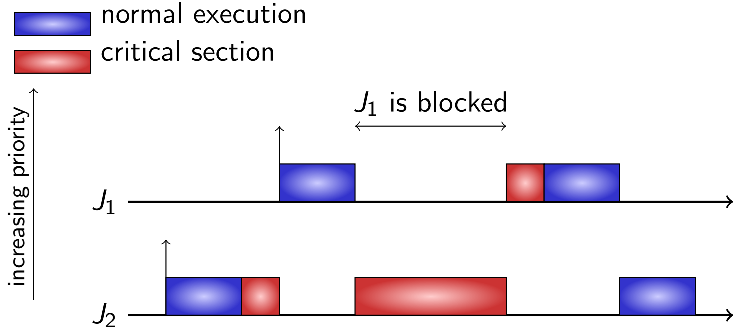
Race Condition: A situation where the outcome of the execution depends on the particular order of process scheduling.

Accessing an Exclusive Resource

- J_1: Normal execution - Accessing the resource R_k - Normal execution
- J_2: Normal execution - Accessing the resource R_k - Normal execution

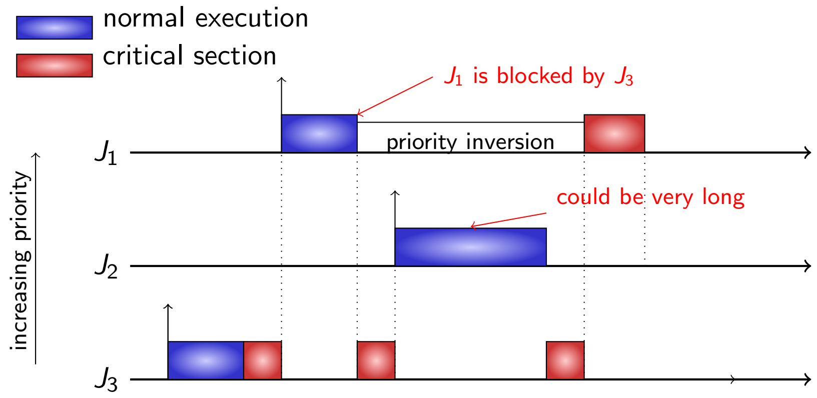


Priority inversion?



Priority Inversion

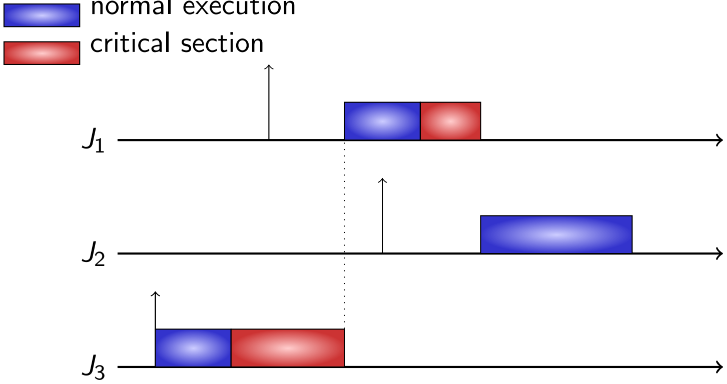
A higher priority job is *blocked* by a lower-priority job.



Naïve Solution for Priority Inversion

Disallow preemption during critical sections

- It is simple
- But, it creates unnecessary blocking, as unrelated tasks may be blocked

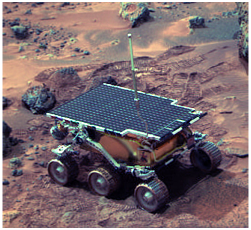


Case Study: MARS Pathfinder Problem

- Mission Name: Mars Pathfinder
- Budget: 250 million USD (1997)
- Mission Length: ˜30 sol (solar days on Mars)
- Rover Name: Sojourner
    - On-board processor: Intel 80C85 CPU 2 MHz, 512 kB RAM, 176 kB of flash
    - RTOS: VxWorks

Problem:

Not long after the rover started gathering meteorological data, the vehicle began experiencing **total system resets**, each resulting in losses of data.



Case Study: MARS Pathfinder Problem (cont.)

- "VxWorks provides **preemptive** priority scheduling
- Tasks were executed as threads with **priorities** reflecting their relative urgency
- Pathfinder contained an information bus, i.e., a **shared memory area**, used for passing information between different components"

- Tasks:
  - A bus management task ran frequently to move data in and out of the information bus → **high priority**
    - Access to the bus was synchronized with mutual exclusion locks (mutexes)
  - The meteorological data gathering task ran as an infrequent → **low priority**
    - Acquire a mutex - write to the bus (to publish the data) - release the mutex
  - Communication and other tasks → **medium priority**.

Case Study: MARS Pathfinder Problem (cont.)

- “Most of the time this combination worked fine.
- It was possible for an interrupt to occur that caused the medium priority tasks (communication and others) to be scheduled while the high priority task (information bus thread) was blocked waiting for the low priority task (meteorological data thread).
- The long-running communications task (with higher priority than the meteorological task) would prevent it from running $\rightarrow$ preventing the blocked information bus task from running $\rightarrow$ expired its deadline (reported by the OS) $\rightarrow$ executed the appropriate action i.e. system reset.
- The rover would forget the rest of the commands, i.e. sitting idle for rest of the sol till new commands were uploaded.
- This scenario is a classic case of priority inversion.”

[Ref: Wilner, David. “Vx-files: What really happened on mars”. Keynote at RTSS 97. Vol. 41. 1997.]

Solution: Resource Access Protocols

- Main idea
  - Modify (increase) the priority of those tasks/jobs that cause blocking.
  - When a job J_j blocks one or more higher-priority jobs, it temporarily assumes a higher priority.
- Methods
  - Priority Inheritance Protocol (PIP), for fixed-priority scheduling
  - Priority Ceiling Protocol (PCP), for fixed-priority scheduling
  - Stack Resource Policy (SRP), for both fixed- and dynamic-priority scheduling
  - others.....

Outline

Priority Inheritance Protocol (PIP)

When a lower-priority job J_j blocks a higher-priority job, the priority of job J_j is *promoted* to the priority level of highest-priority job that job J_j blocks.

For example, if the priority order is $J_1 > J_2 > J_3 > J_4 > J_5$,

- When job J_4 blocks jobs J_2 and J_3, the priority of J_4 is promoted to the priority level of J_2.
- When job J_5 blocks jobs J_1 and J_3, the priority of J_5 is promoted to the priority level of J_1.

Priority Inheritance Protocol (PIP)

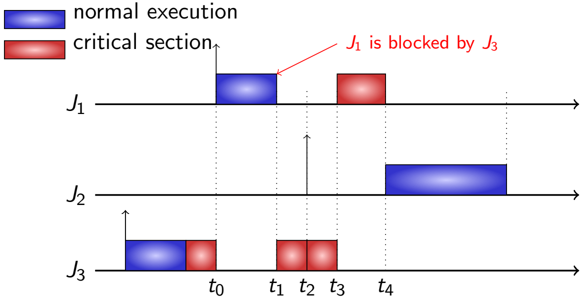
When a lower-priority job J_j blocks a higher-priority job, the priority of job J_j is *promoted* to the priority level of highest-priority job that job J_j blocks.

For example, if the priority order is $J_1 > J_2 > J_3 > J_4 > J_5$,

- When job J_4 blocks jobs J_2 and J_3, the priority of J_4 is promoted to the priority level of J_2.
- When job J_5 blocks jobs J_1 and J_3, the priority of J_5 is promoted to the priority level of J_1.

Priority inheritance solved the Mars Pathfinder problem: the VxWorks operating system used in the pathfinder implements a flag for the calls to mutex primitives. This flag allows priority inheritance to be set to on. When the software was shipped, it was set to off.

Example of PIP

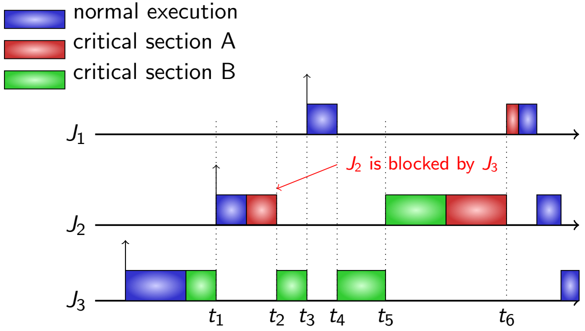


- t_0: J_1 arrives and preempts J_3, since J_1 does not want to enter the critical section
- t_1: J_1 locks the semaphore and tries to enter the critical section. J_1 is blocked by J_3, and J_3 inherits J_1's priority
- t_2: J_2 arrives and has a lower priority than J_3, since J_3 inherited J_1's priority.
- t_3: J_3 leaves its critical section, and J_1 now preempts J_3.
- t_4: J_1 finishes, and J_2 is the highest-priority task.

Blocking in PIP

- *Direct Blocking*: higher-priority job tries to acquire a resource held by a lower-priority job.
- *Push-through Blocking*: medium-priority job is blocked by a lower-priority job that has a higher priority from a job it directly blocks
- *Transitive Blocking*: higher-priority job is blocked by a medium-priority job due to nested critical sections.

Transitive Blocking



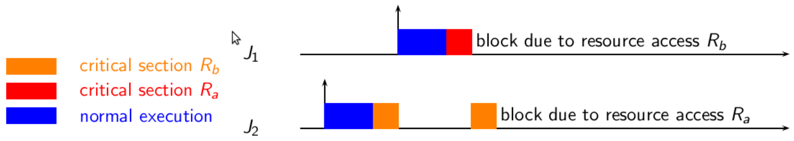
Three jobs with two semaphores A and B:

- J_1: normal execution, wait A, signal A, normal execution
- J_2: normal execution, wait A, wait B, signal B, signal A, normal execution
- J_3: normal execution, wait B, signal B, normal execution

Problem of PIP

- PIP might cause a *deadlock* if there are multiple resources

- Example: Two jobs with two semaphores a and b
  - J_1: normal execution, wait a, wait b, signal b, signal a, normal execution
  - J_2: normal execution, wait b, wait a, signal a, signal b, normal execution



Solution: Use Priority Ceiling Protocol (PCP)

Outline

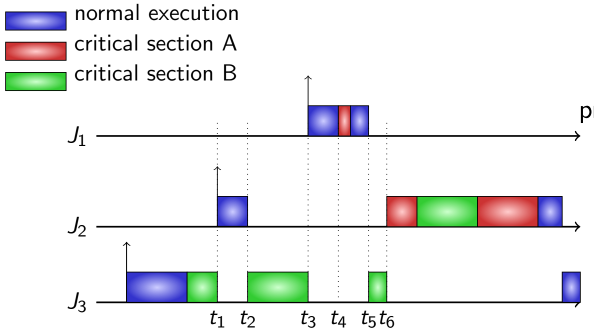
Priority Ceiling Protocol (PCP)

- Two key assumptions:
  - The assigned priorities of all jobs are fixed.
  - The resources required by all jobs are known a priori before the execution of any job begins.
- Definition: The priority ceiling of a resource R is the highest priority of all the jobs that require R, and is denoted $\Pi(R)$.
- Definition: The current priority ceiling $\Pi'(t)$ of the system is equal to the highest priority ceiling of the resources currently in use at time t, or Ω if no resources are currently in use (Ω is a priority lower than any real priority).
- Use the priority ceiling to decide whether a higher priority can allocate a resource or not.

PCP

1. Scheduling Rule
   - Every job J is scheduled based on the current priority $\pi(t, J)$.
2. Allocation Rule: Whenever a job J requests a resource R at time t, one of the following two conditions occurs:
   - R is held by another job and J becomes blocked.
   - R is free:
     - If J's priority $\pi(t, J)$ is higher than the current priority ceiling $\Pi'(t)$, R is allocated to J.
     - Otherwise, only if J is the job holding the resource(s) whose priority ceiling equals $\Pi'(t)$, R is allocated to J
     - Otherwise, J becomes blocked.
3. Priority-inheritance Rule: When J becomes blocked, the job J_l that blocks J inherits the current priority $\pi(t, J)$ of J. J_l executes at its inherited priority until it releases every resource whose priority ceiling is $\geq \pi(t, J)$ (or until it inherits an even higher priority); at that time, the priority of J_l returns to its priority $\pi(t', J_l)$ at the time t' when it was granted the resources.

Example 1 of PCP



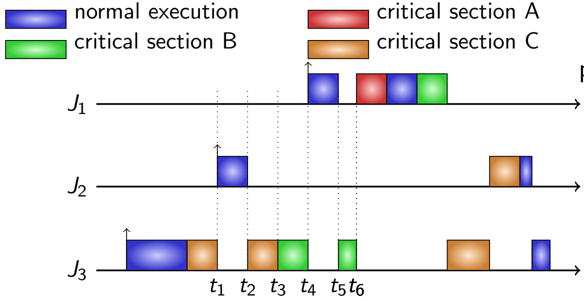
priority ceilings:

- A: priority level 1
- B: priority level 2

Three jobs with two semaphores A and B.

- J_1: normal execution, wait A, signal A, normal execution
- J_2: normal execution, wait A, wait B, signal B, signal A, normal execution
- J_3: normal execution, wait B, signal B, normal execution

Example 2 of PCP



priority ceilings:

- A: priority level 1
- B: priority level 1
- C: priority level 2

Three jobs with three semaphores A, B, and C.

- J_1: normal execution, wait A, signal A, normal execution, wait B, signal B, normal execution
- J_2: normal execution, wait C, signal C, normal execution
- J_3: normal execution, wait C, wait B, signal B, signal C, normal execution

Outline

Events in Real-time Systems

- Regular events or known arrival patterns
  - Periodic tasks
  - Time-driven
  - Usually hard timing constraints, i.e., critical tasks
  - Scheduled under RM, EDF, etc.
- Irregular events or unknown arrival patterns
  - Aperiodic tasks
  - Event-driven
  - Hard, soft, or non-real-time requirements
  - Offline schedulability guarantee based on maximum arrival rate
    - e.g. minimum inter-arrival time for sporadic tasks
    - Maximum arrival rate cannot be bounded? $\rightarrow$ No guarantee!
      $\rightarrow$ We need online guarantee
- **Objective**: Provide a guarantee to schedule hard real-time tasks and provide good average response times for soft and non real-time tasks

Backgroud Scheduling

- Handle aperiodic soft-real time tasks when there are no periodic tasks ready for execution
- Two ready queues:
    - Higher priority queue for periodic tasks.
    - Lower priority queue for aperiodic tasks
- Figures 5.1 and 5.2, page 121 in "Hard Real-Time Computing Systems" book

+ Simple

− The response time of aperiodic requests can be too long for high periodic loads

⇒ Use resource reservation servers to improve the average response time of aperiodic tasks

Well-Known Servers

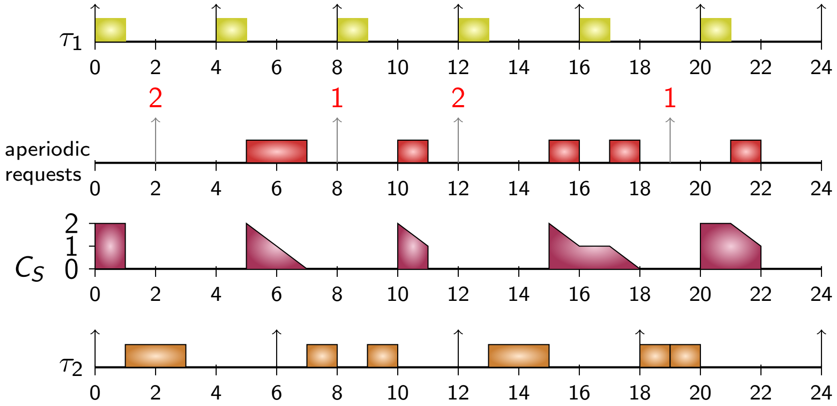
- Servers for fixed-priority systems
  - Polling Server (PS): provide a fixed execution budget that is only available at pre-defined times.
  - Deferrable Server (DS): provide a fixed budget, in which the budget replenishment is done periodically.
  - Sporadic Server (SS): provide a fixed budget, in which the budget replenishment is performed only if it was consumed.
  - Others: priority exchange (PE) server, slack stealer, etc.
- Servers for dynamic-priority systems
  - Total bandwidth server (TBS): provide a fixed utilization for executing jobs, in which the deadline for execution is *dependent* on the execution time of jobs.
  - Constant bandwidth server (CBS): provide a fixed utilization for executing jobs, in which the deadline for execution is *independent* on the execution time of jobs.
  - Others: dynamic priority exchange (DPE) server, dynamic slack stealer, dynamic sporadic server, etc.

Polling Server (PS)

- Improve the average response time of aperiodic tasks
  - compared to background scheduling
- **Behavior**: periodic task i that is dedicated to serve aperiodic requests within a limited budget. It has:
  - period T_{S_i}
  - capacity (computation time) C_{S_i}
- The server is scheduled with the same algorithm used for the periodic tasks
  - The sequencing of aperiodic requests does not depend on the scheduling algorithm used for periodic tasks
    - Any other policy can be used, e.g., according to arrival time, computation time, deadline, etc.
- **Consumption rule**:
  - **Only** upon activation, it executes the **pending aperiodic events** until either (1) there is no more requests in the aperiodic ready queue or (2) the capacity C_{S_i} is exhausted.

An Example of PS

$\tau_1 = (1, 4, 4)$, $\tau_2 = (2, 6, 6)$.
Polling server: $C_S = 2$ and $T_S = 5$.
Priority: $\tau_1 > PS > \tau_2$.



Properties of PS

- Schedulability Guarantee:
  - Suppose that there are n periodic tasks and a polling server with utilization $U_S = \frac{C_S}{T_S}$ and the RM scheduling algorithm is adopted.
  - The schedulability of the periodic task set is guaranteed if

$$U_S + \sum_{i=1}^{n} \frac{C_i}{T_i} \leq U_{lub}(RM, n+1) = (n+1)(2^{\frac{1}{n+1}} - 1).$$

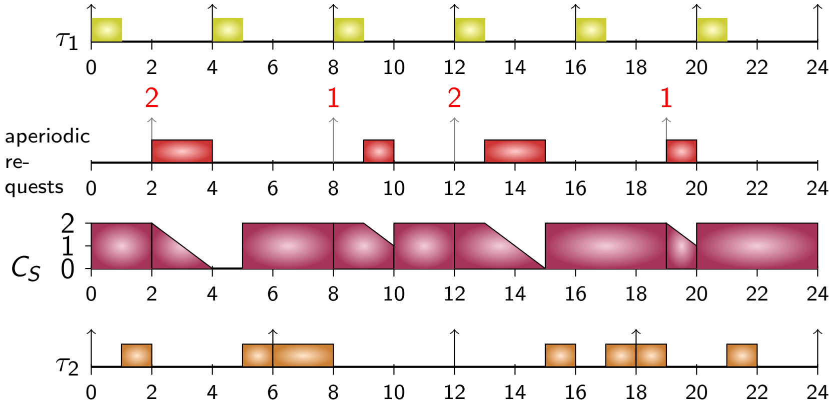
  - The proof can be done by imaging that a periodic task represents the polling server, which is executed for at most C_i time units after it is granted for execution.
  - Since the capacity is greedily set to 0 if there is no request for the polling server to execute, the periodic task, that represents the polling server, can be imaged as early completion, instead of task suspension, of the task.

Deferrable Server (DS)

- Improve the average response time of aperiodic tasks
  - compared to polling server
- **Behavior**: periodic task
  - period: T_{S_i}
  - capacity (computation time): C_{S_i}
- **Replenishment rule**:
  - Upon activation of the task (periodic replenishment at the multiple of T_{S_i})
- **Consumption rule**:
  - Aperiodic requests are served when the server still has capacity
  - DS preserves its capacity if no requests are pending
  - The capacity is maintained until the end of the period
- Shorter response times can be achieved by creating a DS having the highest priority among the periodic tasks

An Example of DS

$\tau_1 = (1, 4, 4)$, $\tau_2 = (2, 6, 6)$.
Deferrable server: $C_S = 2$ and $T_S = 5$.
Priority: $\tau_1 > DS > \tau_2$.



Schedulability Guarantee of DS

- Suppose that there are n periodic tasks and a deferrable server with utilization $U_S = \frac{C_S}{T_S}$ and the RM scheduling algorithm is adopted.

- RM analysis is incorrect for such a case, since the periodic task, that represents the deferrable server, can be imaged as self-suspension (e.g., time 1 in the example) of the task.

Sporadic Server (SS)

- Improve the average response time of aperiodic tasks without degrading the utilization bound of the periodic task set.

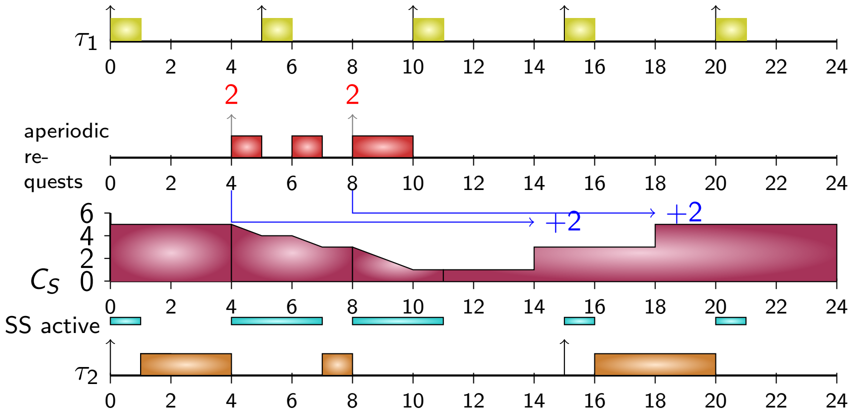
- SS differs from DS in the way it replenishes its capacity
  - DS: replenishes its capacity to **full value** periodically at the beginning of each period
  - SS: replenishes its capacity only after it has been consumed by aperiodic task execution

Sporadic Server (SS) (cont.)

- **Behavior**: sporadic task with a specified priority
  - period: T_{S_i}
  - capacity (computation time): C_{S_i}
- **Rules**:
  - $\pi_{exe}(t)$: the priority level of the task that is executing at time t
  - π_{SS}: the priority level of the sporadic server
  - The server is **Active** when $\pi_{exe}(t) \geq \pi_{SS}$
  - The server is **Idle** when the $\pi_{exe}(t) < \pi_{SS}$
  - Initially, the server is Idle and its budget is C_{S_i}. When the server becomes Active at time t_1, the **replenishment time** is set to $t_1 + T_{S_i}$
  - When the server becomes Idle at time t_2, the (next) **replenishment amount** is set to the amount of capacity consumed in time interval $[t_1, t_2]$

An Example of SS

$\tau_1 = (1, 5, 5)$, $\tau_2 = (4, 15, 15)$.
Sporadic server: $C_S = 5$ and $T_S = 10$.
Priority: $\tau_1 > SS > \tau_2$.



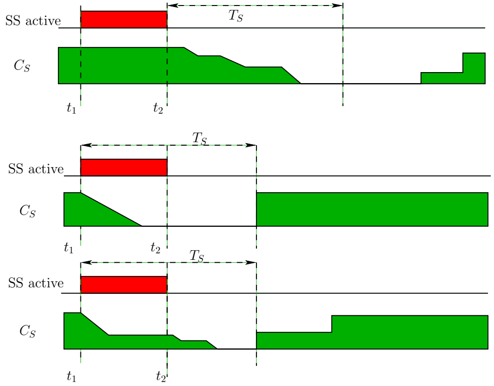
Schedulability Guarantees of SS

Theorem

If a periodic task set is schedulable, replacing a task τ_i by a sporadic server SS_i with the same period and execution time is still schedulable.

- The theorem can be proved by showing that for any type of service, SS exhibits an execution behavior equivalent to one or more periodic tasks.
- The execution behavior of the server in the interval $[t_1, t_2]$ can be described by one of the following three cases:

Schedulability Guarantees of SS (cont.)



Case 1: No capacity is consumed $\rightarrow$ Imagine that the arrival of a periodic task is delayed to arrive at time t_2.

Case 2: The server capacity is totally consumed $\rightarrow$ Imagine the behavior like a periodic real-time task.

Case 3: The server capacity is partially consumed $\rightarrow$ Imagine that there are multiple tasks with the same period but with different arrival times, and their total execution time is the same as that of τ_i.

Overview of PS, DS, and SS

	Performance	computation	memory	implementation complexity
PS	poor	excellent	excellent	excellent
DS	good	excellent	excellent	excellent
SS	excellent	good	good	good