Real-Time Scheduling (Part 2)
(Working Draft)

Insup Lee
Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
www.cis.upenn.edu/~lee/

CIS 541, Spring 2010

PERIODIC SERVERS FOR APERIODIC TASKS

Mixed Periodic and Aperiodic Task Systems

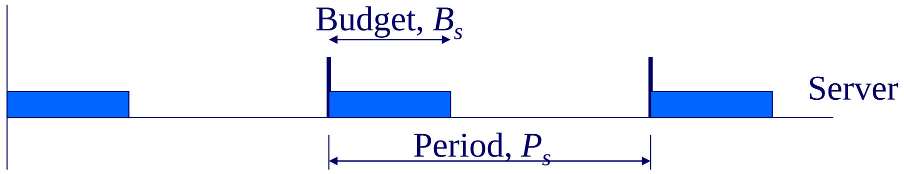
- Question: how to execute aperiodic tasks without violating schedulability guarantees given to periodic tasks?

Mixed Periodic and Aperiodic Task Systems

- Question: how to execute aperiodic tasks without violating schedulability guarantees given to periodic tasks?
- One Answer: Execute aperiodic tasks at lowest priority
  - Problem: Poor performance for aperiodic tasks

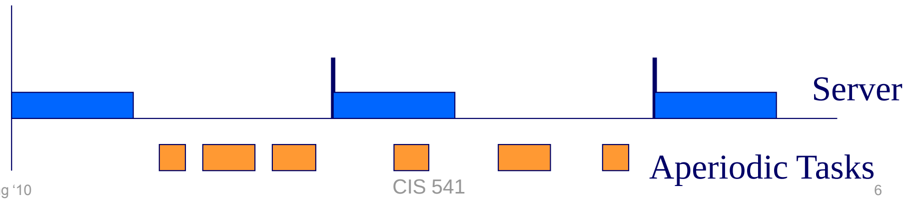
Mixed Periodic and Aperiodic Task Systems

- Idea: aperiodic tasks can be served by periodically invoked servers
- The server can be accounted for in periodic task schedulability analysis
- The server has a period P_s and a budget B_s
- Server can serve aperiodic tasks until budget expires
- Servers have different flavors depending on the details of when they are invoked, what priority they have, and how budgets are replenished



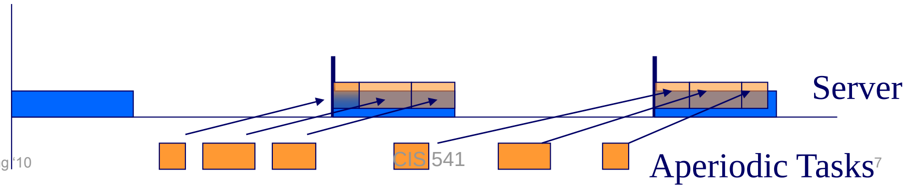
Mixed Periodic and Aperiodic Task Systems

- Idea: aperiodic tasks can be served by periodically invoked servers
- The server can be accounted for in periodic task schedulability analysis
- The server has a period P_s and a budget B_s
- Server can serve aperiodic tasks until budget expires
- Servers have different flavors depending on the details of when they are invoked, what priority they have, and how budgets are replenished



Mixed Periodic and Aperiodic Task Systems

- Idea: aperiodic tasks can be served by periodically invoked servers
- The server can be accounted for in periodic task schedulability analysis
- The server has a period P_s and a budget B_s
- Server can serve aperiodic tasks until budget expires
- Servers have different flavors depending on the details of when they are invoked, what priority they have, and how budgets are replenished

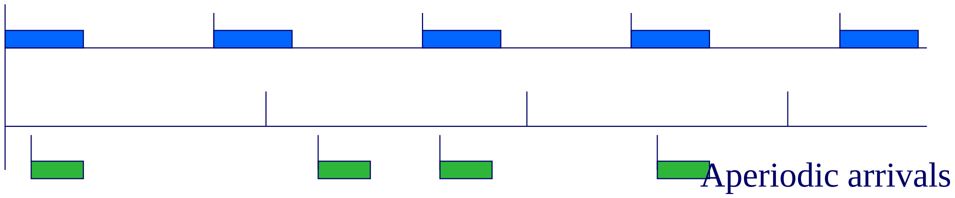


Polling Server

- Runs as a periodic task (priority set according to RM)
- Aperiodic arrivals are queued until the server task is invoked
- When the server is invoked it serves the queue until it is empty or until the budget expires then suspends itself
  - If the queue is empty when the server is invoked it suspends itself immediately.
- Server is treated as a regular periodic task in schedulability analysis

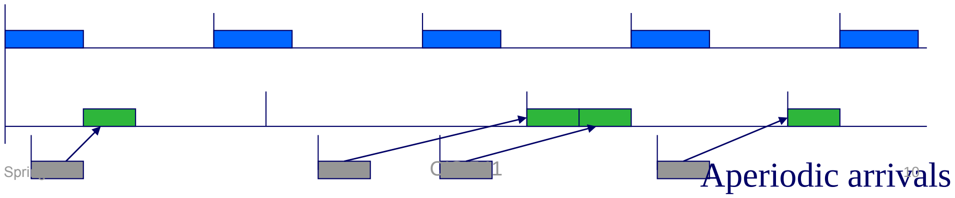
Example of a Polling Server

- Polling server:
  - Period $P_s = 5$
  - Budget $B_s = 2$
- Periodic task
  - $P = 4$
  - $C = 1.5$
- All aperiodic arrivals have C=1



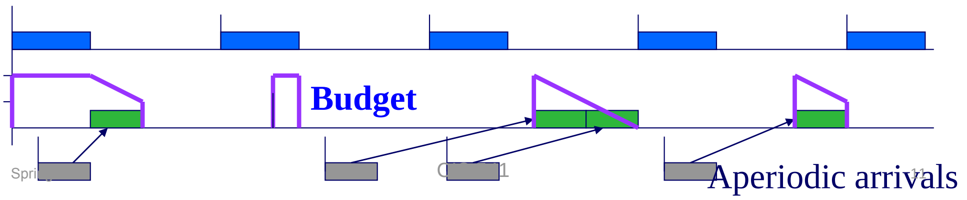
Example of a Polling Server

- Polling server:
  - Period $P_s = 5$
  - Budget $B_s = 2$
- Periodic task
  - $P = 4$
  - $C = 1.5$
- All aperiodic arrivals have C=1



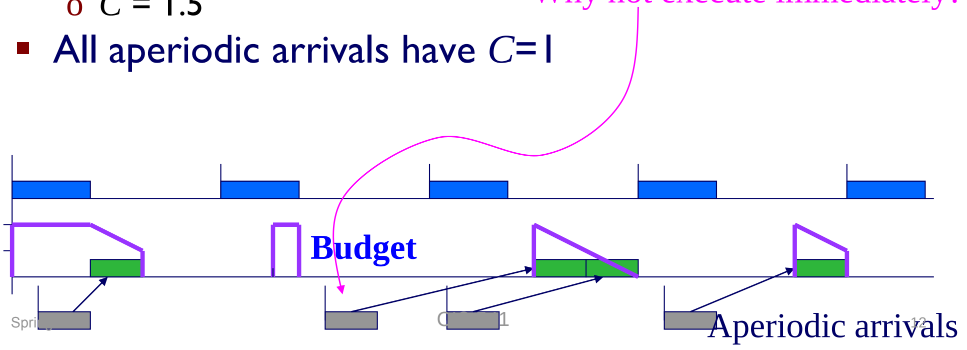
Example of a Polling Server

- Polling server:
  - Period $P_s = 5$
  - Budget $B_s = 2$
- Periodic task
  - $P = 4$
  - $C = 1.5$
- All aperiodic arrivals have $C=1$



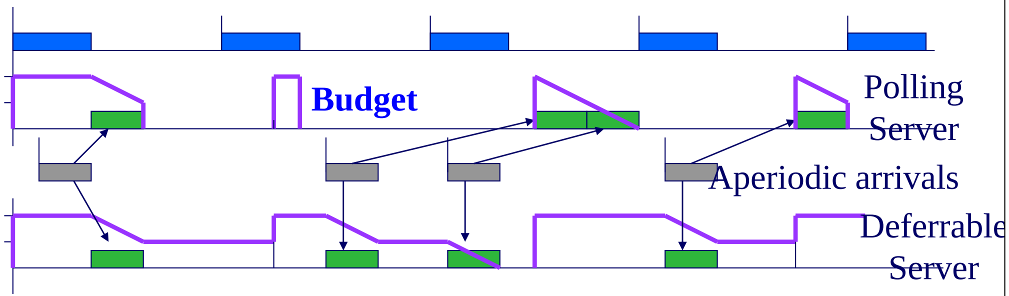
Example of a Polling Server

- Polling server:
  - Period $P_s = 5$
  - Budget $B_s = 2$
- Periodic task
  - $P = 4$
  - $C = 1.5$
- All aperiodic arrivals have $C=1$

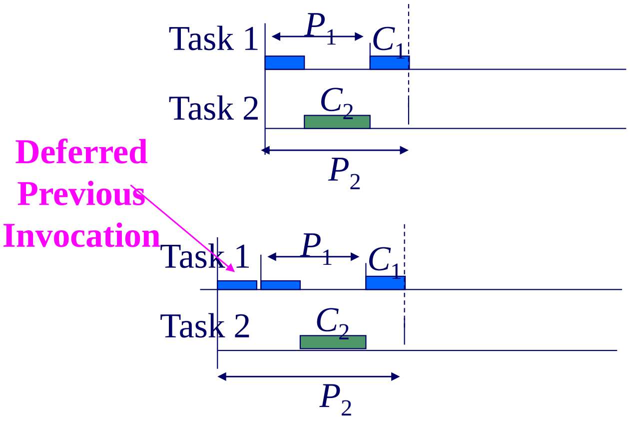

Why not execute immediately?

Deferrable Server

- Keeps the balance of the budget until the end of the period
- Example (continued)



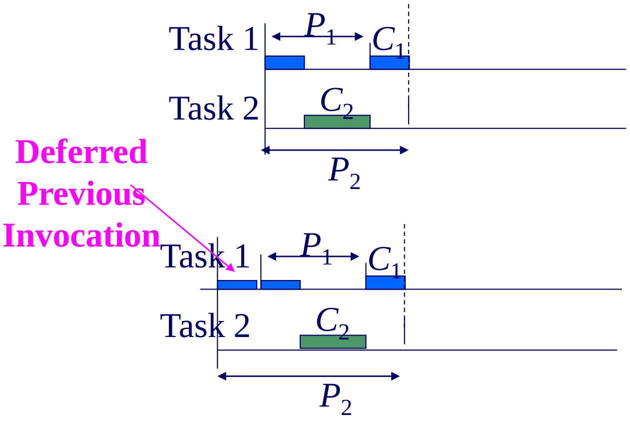
Worst-Case Scenario

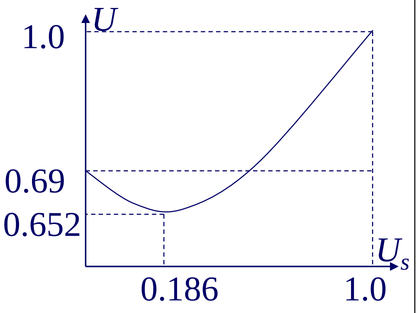


$$U_p \le \ln\left(\frac{2U_s+1}{U_s+1}\right)$$

Exercise: Derive the utilization bound for a deferrable server plus one periodic task

Worst-Case Scenario





$$U_p \le \ln\left(\frac{2U_s+1}{U_s+1}\right)$$

Exercise: Derive the utilization bound for a deferrable server plus one periodic task

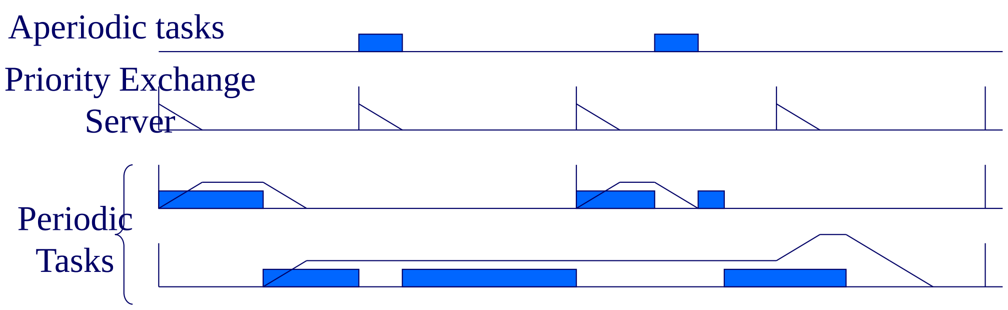
---

Priority Exchange Server

- Like the deferrable server, it keeps the budget until the end of server period
- Unlike the deferrable server the priority slips over time: When not used, the priority is exchanged for that of the executing periodic task

Priority Exchange Server

Example

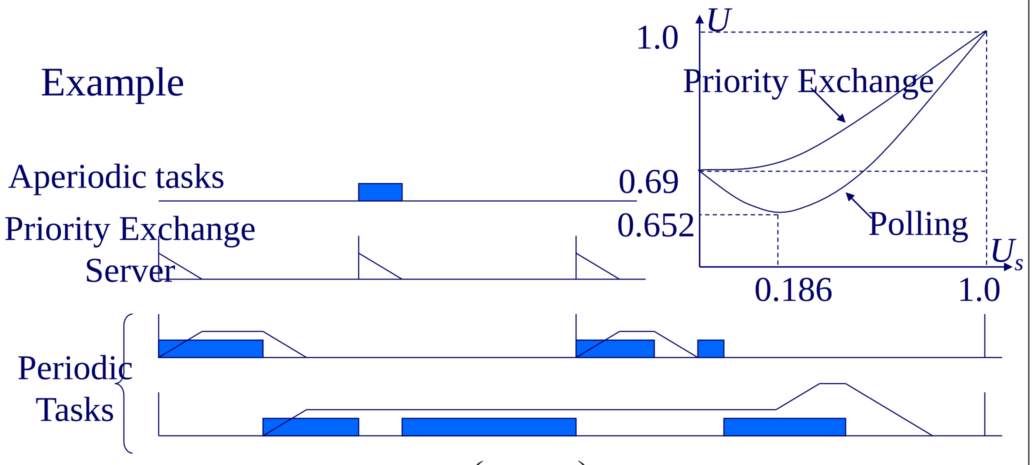

Aperiodic tasks

Priority Exchange Server

Periodic Tasks

Priority Exchange Server

Example


Aperiodic tasks

Priority Exchange Server

Periodic Tasks

$$U_p \le \ln\left(\frac{2}{U_s+1}\right)$$

Sporadic Server

- Server is said to be *active* if it is in the *running* or *ready* queue, otherwise it is *idle*.
- When an aperiodic task comes and the budget is not zero, the server becomes active
- Every time the server becomes *active*, say at t_A, it sets replenishment time one period into the future, $t_A + P_s$ (but does not decide on replenishment amount).
- When the server becomes idle, say at t_I, set replenishment amount to capacity consumed in $[t_A, t_I]$

$$U_p \le \ln\left(\frac{2}{U_s + 1}\right)$$

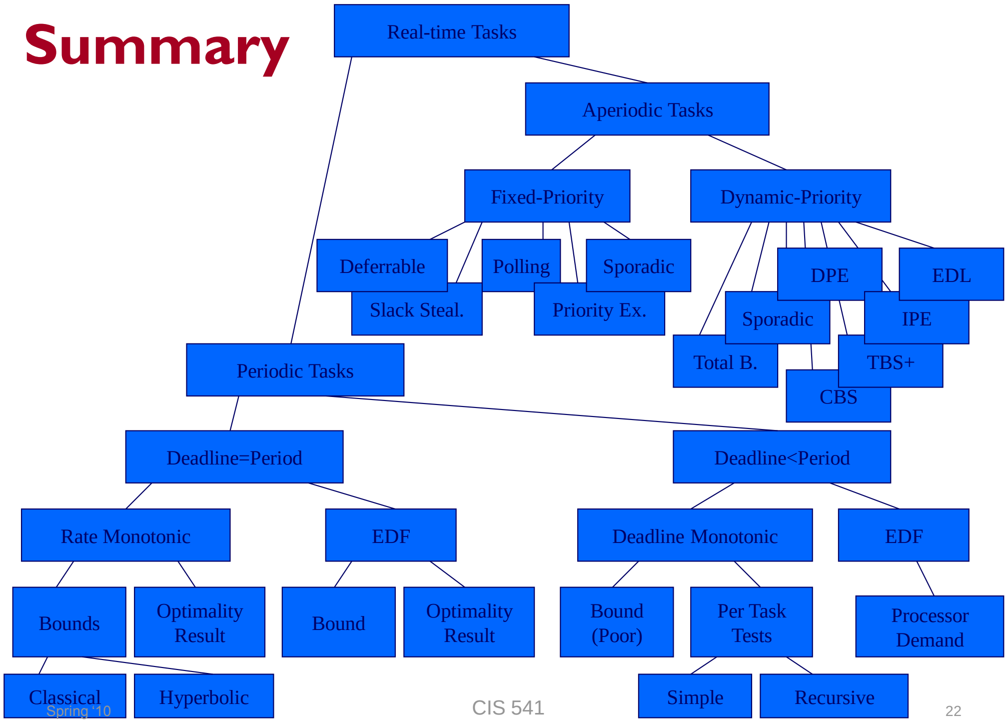
Slack Stealing Server

- Compute a slack function $\mathrm{A}(t_s, t_f)$ that says how much total slack is available
- Admit aperiodic tasks while slack is not exceeded

Aperiodic Servers

Dynamic Priority

Summary



Real-time Tasks

Aperiodic Tasks

Fixed-Priority

Dynamic-Priority

Deferrable

Polling

Sporadic

Slack Steal.

Priority Ex.

DPE

EDL

Sporadic

IPE

Total B.

TBS+

CBS

Periodic Tasks

Deadline=Period

Deadline<Period

Rate Monotonic

EDF

Deadline Monotonic

EDF

Bounds

Optimality Result

Bound

Optimality Result

Bound (Poor)

Per Task Tests

Processor Demand

Classical

Hyperbolic

Simple

Recursive

PRIORITY INVERSION

Resources and Blocking

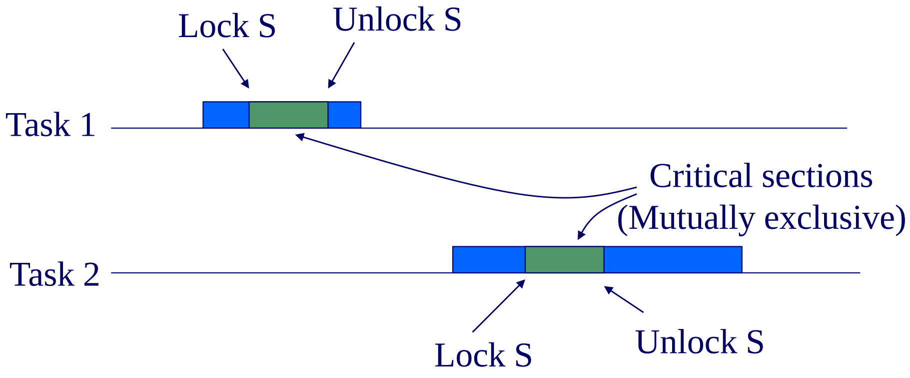
Priority Inheritance
Priority Ceiling
Slack Resource Policy

The Problem

- Tasks have synchronization constraints
  - Semaphores protect critical sections
- Blocking can cause a higher-priority task to wait on a **lower-priority** one to unlock a resource
  - Problem: In all previous derivations we assumed that a task can only wait for **higher-priority** tasks not **lower-priority** tasks
- Question
  - What is the maximum amount of time a higher-priority task can wait for a lower-priority task?
  - How to account for that time in schedulability analysis?

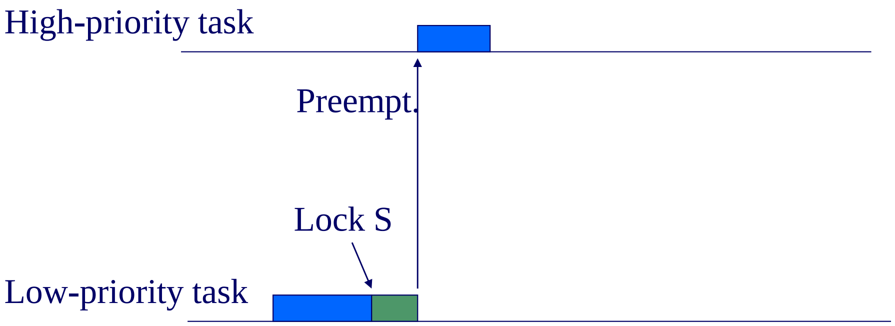
Mutual Exclusion Constraints

- Tasks that lock/unlock the same semaphore are said to have a mutual exclusion constraint



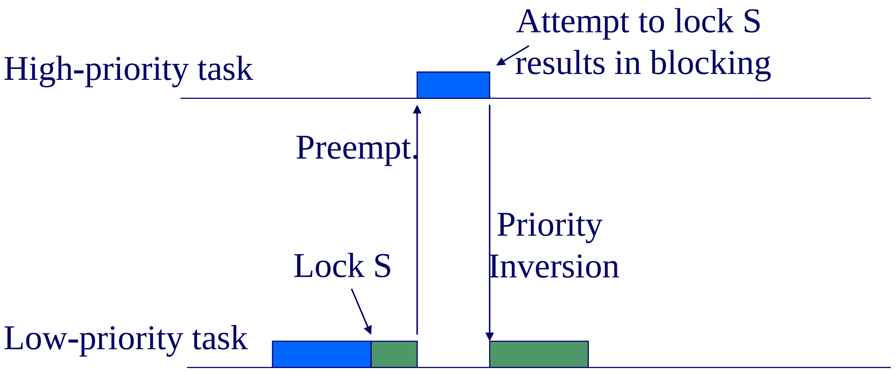
Priority Inversion

- Locks and priorities may be at odds. Locking results in priority inversion



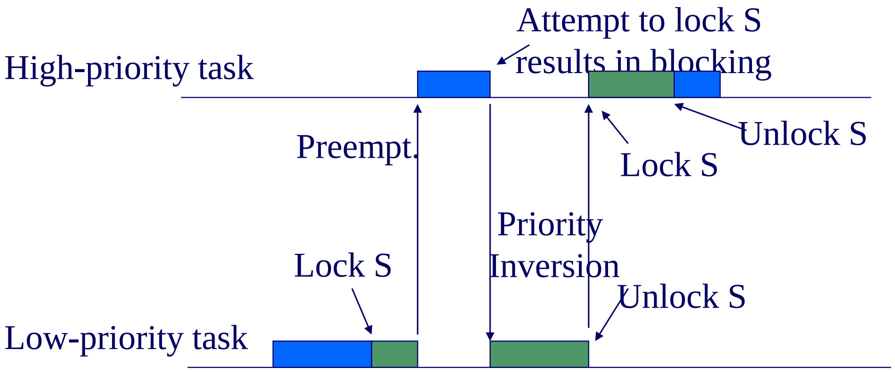
Priority Inversion

- Locks and priorities may be at odds. Locking results in priority inversion



Priority Inversion

- How to account for priority inversion?



Attempt to lock S
results in blocking

High-priority task

Preempt.

Lock S

Unlock S

Priority
Inversion

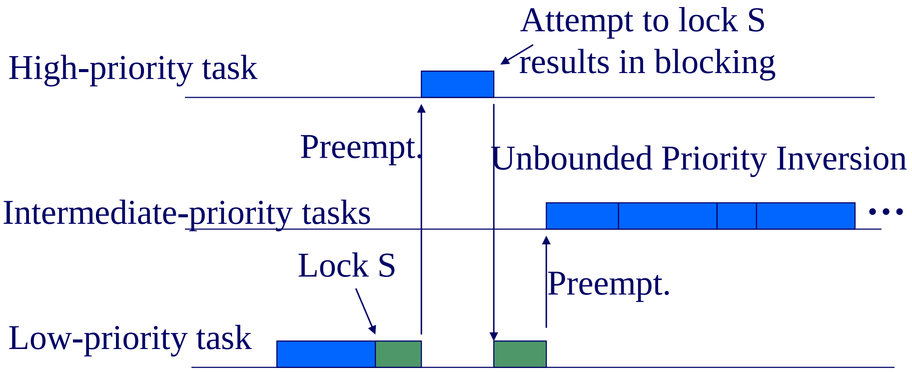
Lock S

Unlock S

Low-priority task

Unbounded Priority Inversion

- Consider the case below: a series of intermediate priority tasks is delaying a higher-priority one



Attempt to lock S
results in blocking

High-priority task

Preempt.

Unbounded Priority Inversion

Intermediate-priority tasks

…

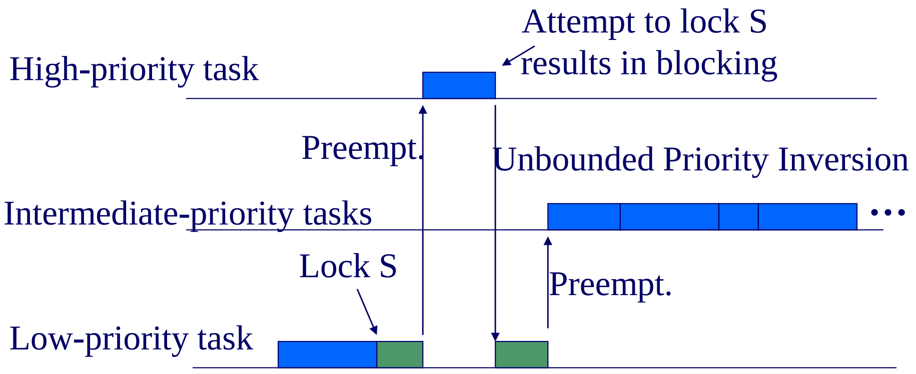
Lock S

Preempt.

Low-priority task

Unbounded Priority Inversion

- How to prevent unbounded priority inversion?



Attempt to lock S results in blocking

High-priority task

Preempt.

Unbounded Priority Inversion

Intermediate-priority tasks

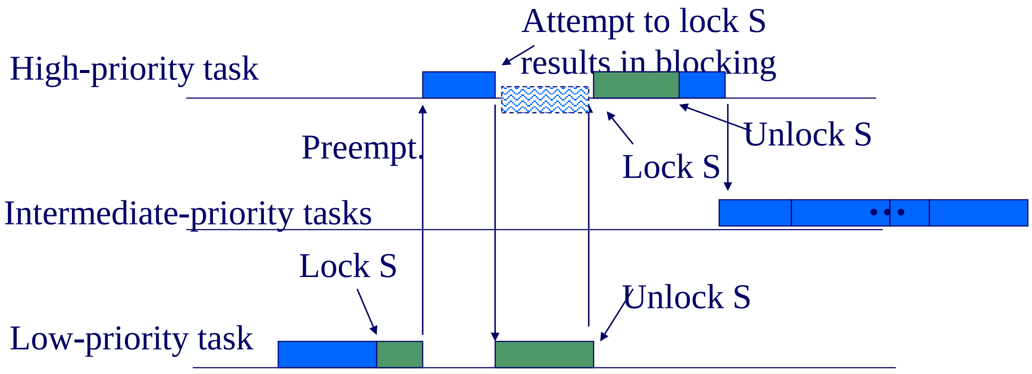
Lock S

Preempt.

Low-priority task

Priority Inheritance Protocol

- Let a task inherit the priority of any higher-priority task it is blocking



Attempt to lock S results in blocking

High-priority task

Unlock S

Preempt.

Lock S

Intermediate-priority tasks

Lock S

Unlock S

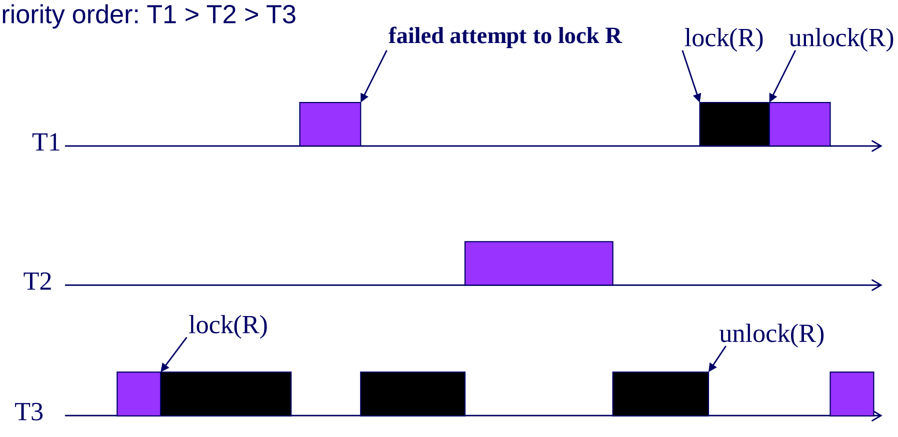
Low-priority task

Priority Inversion and the MARS Pathfinder

- Landed on the Martian surface on July 4th, 1997
- Unconventional landing – boucing into the Martian surface
- A few days later, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system reset, each resulting in losses of data
- What happened:
  - Pathfinder has an "information bus"
  - The meteorological data gathering task ran as an infrequent, low priority thread, and used the information bus to publish its data (while holding the mutex on bus).
  - A communication task that ran with medium priority.
  - It is possible for an interrupt to occur that caused (medium priority) communications task to be scheduled during the short interval of the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread.
  - After some time passed, a watch dog timer goes off, noticing that the data bus has not been executed for some time, it concluded that something had gone really bad, and initiated a total system reset.

The Priority Inversion Problem

Priority order: T1 > T2 > T3

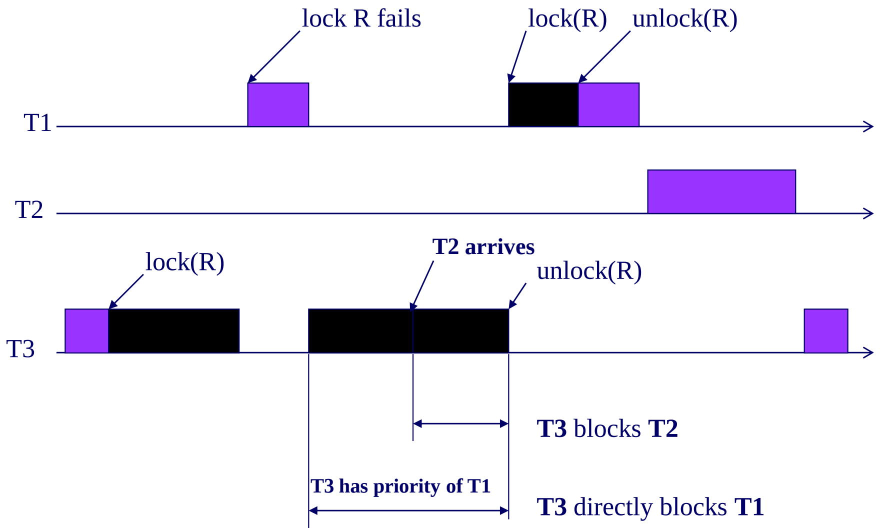


T2 is causing a higher priority task T1 wait !

Priority Inversion

1. T1 has highest priority, T2 next, and T3 lowest
2. T3 comes first, starts executing, and acquires some resource (say, a lock).
3. T1 comes next, interrupts T3 as T1 has higher priority
4. But T1 needs the resource locked by T3, so T1 gets blocked
5. T3 resumes execution (this scenario is still acceptable so far)
6. T2 arrives, and interrupts T3 as T2 has higher priority than T3, and T2 executes till completion
7. In effect, even though T1 has priority than T2, and arrived earlier than T2, T2 delayed execution of T1
8. This is “priority inversion” !! Not acceptable.
9. Solution T3 should inherit T1’s priority at step 5

Priority Inheritance Protocol

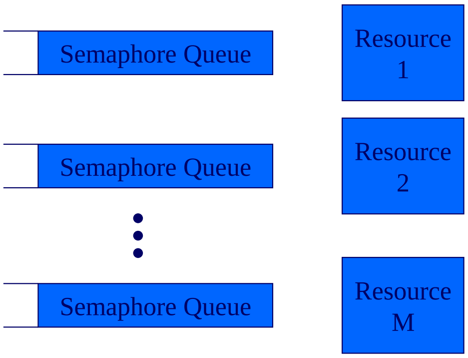


Priority Inheritance Protocol

- Question: What is the longest time a task can wait for lower-priority tasks?
- Answer: ?

Computing the Maximum Priority Inversion Time

- Consider the instant when a high-priority task that arrives.
  - What is the most it can wait for lower priority ones?



If I am a task, priority inversion occurs when

(a) Lower priority task holds a resource I need (direct blocking)

(b) Lower priority task inherits a higher priority than me because it holds a resource the higher-priority task needs (push-through blocking)

Schedulability Test

$$\forall i, 1 \le i \le n,$$

$$\frac{B_i}{P_i} + \sum_{k=1}^{i} \frac{C_k}{P_k} \le i(2^{1/i} - 1)$$

Schedulability Test

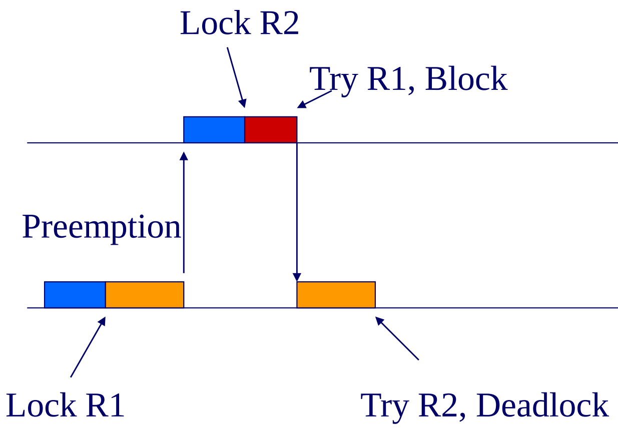
$$\forall i, 1 \le i \le n,$$

$$\frac{B_i}{T_i} + \sum_{k=1}^{i} \frac{C_k}{P_k} \le i(2^{1/i} - 1)$$

Why do we have to test each task separately? Why not just one utilization-based test like it used to?

Problem: Deadlock

Deadlock occurs if two tasks locked two semaphores in opposite order



Lock R2

Try R1, Block

Preemption

Lock R1

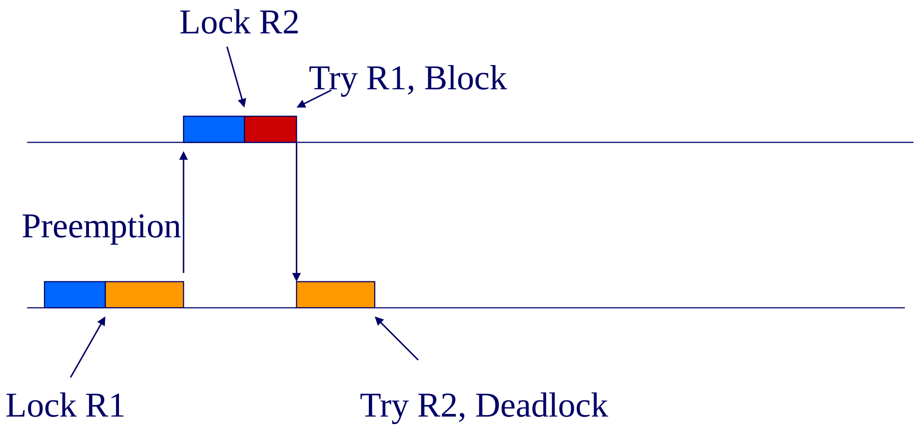
Try R2, Deadlock

Priority Ceiling Protocol

- Definition: The priority ceiling of a semaphore is the highest priority of any task that can lock it
- A task that requests a lock R_k is denied if its priority is not higher than the highest priority ceiling of all currently locked semaphores (say it belongs to semaphore R_h)
  - The task is said to be blocked by the task holding lock R_h
- A task inherits the priority of the top higher-priority task it is blocking

Problem: Deadlock?

Deadlock used to occur if two tasks locked two semaphores in opposite order. Can it still occur in priority ceiling?



Lock R2

Try R1, Block

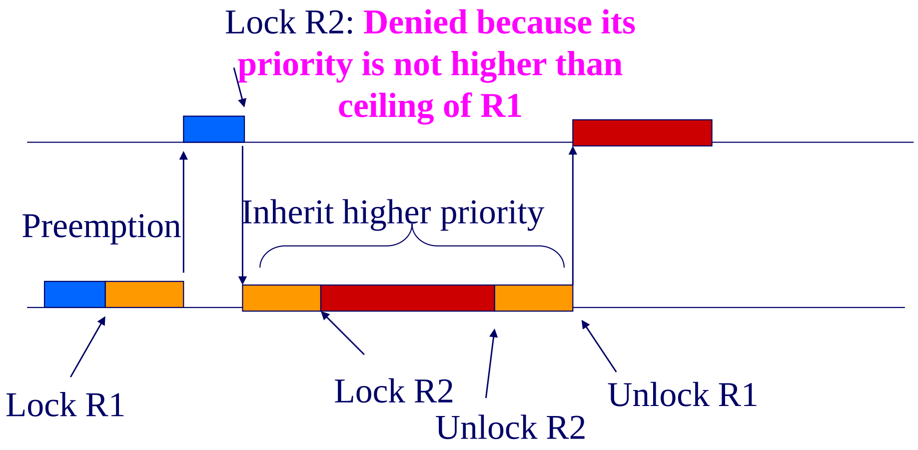
Preemption

Lock R1

Try R2, Deadlock

Problem: Deadlock?

Deadlock used to occur if two tasks locked two semaphores in opposite order. Can it still occur in priority ceiling?



Lock R2: **Denied because its priority is not higher than ceiling of R1**

Preemption

Inherit higher priority

Lock R1

Lock R2

Unlock R2

Unlock R1

Slack Resource Policy

- Priority:
  - Any static or dynamic policy (e.g., EDF, RM, …)
- Preemption Level
  - Any *fixed value* that satisfies: If A arrives after B and Priority (A) > Priority (B) then PreemptionLevel (A) > PreemptionLevel (B)
- Resource Ceiling
  - Highest preemption level of all tasks that might access the resource
- System Ceiling
  - Highest resource ceiling of all currently locked resources
- A task can preempt another if:
  - It has the highest priority
  - Its preemption level is higher than the system ceiling

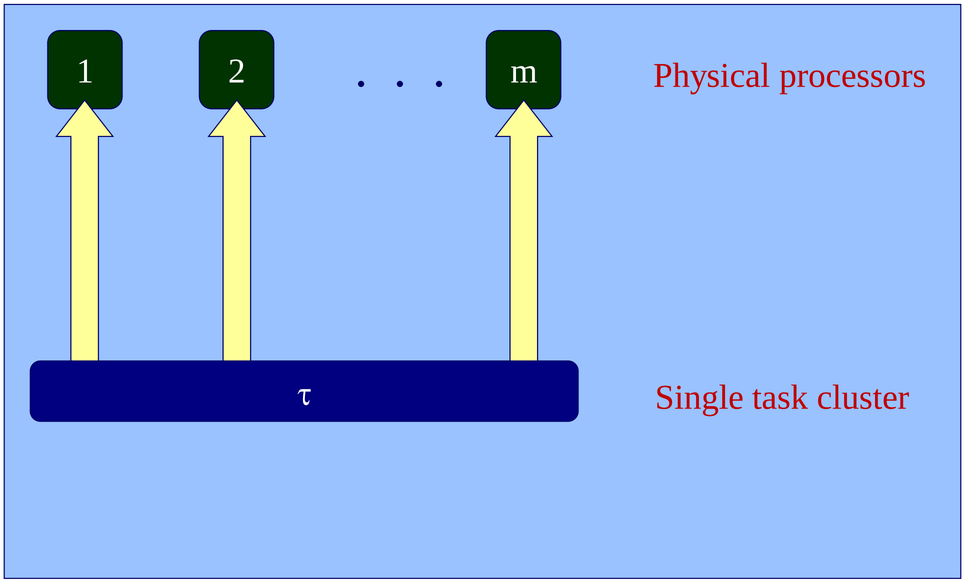
MULTI-PROCESSOR SCHEDULING

Multiprocessor Scheduling

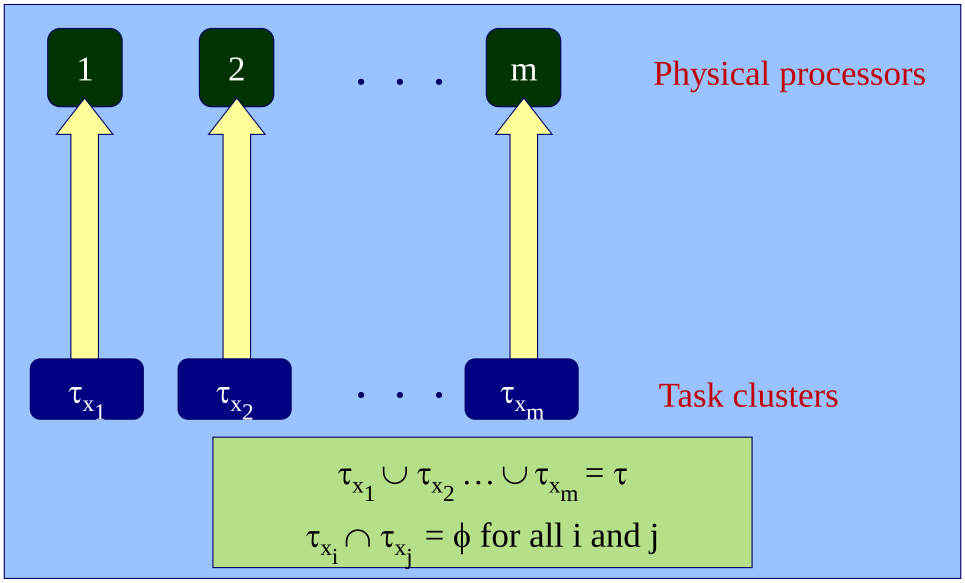
- Why consider multiprocessors
  - Better tradeoff between computational power and costs (energy,fabrication)
  - Ability to exploit inherent concurrency in software

- Problem statement
  - Constrained deadline sporadic task system
    - $W = \{\tau_1,\ldots,\tau_n\}$, where $\tau_i=(T_i,C_i,D_i)$ and $C_i <= D_i <= T_i$
    - C_i units must be supplied non-concurrently
  - Identical, unit-capacity multiprocessor platform
    - m processors
  - How can W be scheduled on these m processors?

Global Scheduling



Partitioned Scheduling

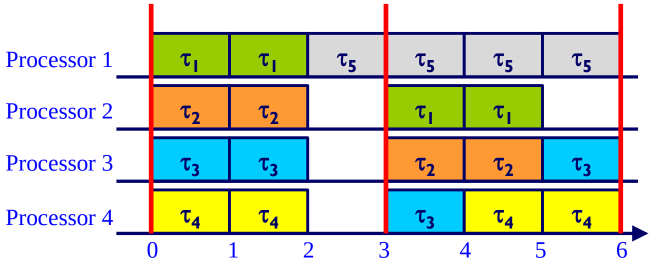


$$\tau_{x_1} \cup \tau_{x_2} \ldots \cup \tau_{x_m} = \tau$$

$$\tau_{x_i} \cap \tau_{x_j} = \phi \text{ for all i and j}$$

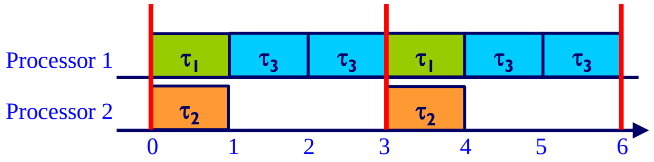
Counter-example for Partitioned

- Task set and number of processors
  - $\tau_1=\tau_2=\tau_3=\tau_4=(3,2,3)$ and $\tau_5=(6,4,6)$, m=4
- No partitioning technique will work
  - Some processor must be assigned two tasks which is not possible
- Schedulable under global LLF



Counter-example for Global

- Task set and number of processors
  - $\tau_1=\tau_2=(3,1,3)$ and $\tau_3=(7,6,7)$, m=2
- Global EDF cannot schedule this task set

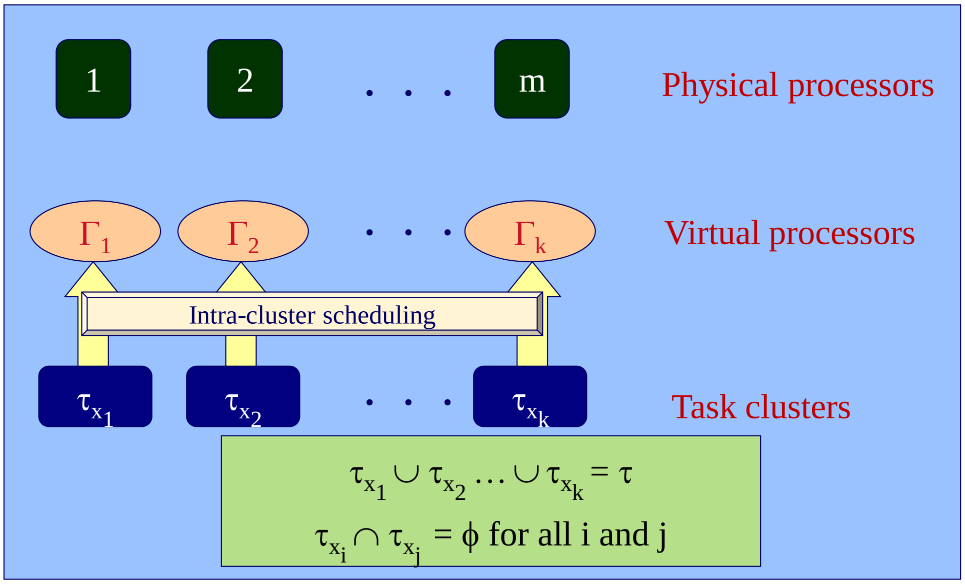


- Partitioned EDF can
  - τ_3 on processor 1, and the other two on processor 2

Partitioned vs. Global

- Two extreme cases of task-processor clustering
  - one-one (partitioned) vs. one-all (global)
- Both have advantages and disadvantages
  - Partitioned: Low preemptions, but low resource utilization
  - Global: High resource utilization with high preemptions
- Optimal scheduling on identical platforms
  - Only developed for implicit deadline task systems ($D_i = T_i$ for all i)
  - All known optimal schedulers are global strategies (Pfair [BCPV96])
  - Problem open for constrained deadline periodic task systems
    - Shown to be impossible for sporadic task systems
- Can we support general task-processor clustering through the concept of platform virtualization (hierarchical scheduling)?

Virtual Cluster-based Scheduling



1 2 . . . m

Physical processors

Γ_1 Γ_2 . . . Γ_k

Virtual processors

Intra-cluster scheduling

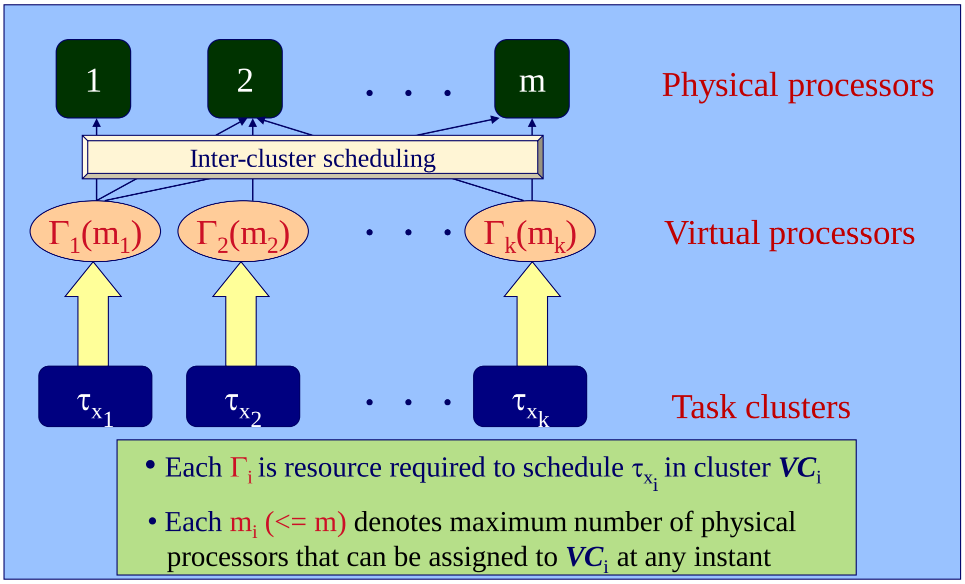
τ_{x_1} τ_{x_2} . . . τ_{x_k}

Task clusters

$$\tau_{x_1} \cup \tau_{x_2} \ldots \cup \tau_{x_k} = \tau$$

$\tau_{x_i} \cap \tau_{x_j} = \phi$ for all i and j

Virtual Cluster-based Scheduling



1 2 . . . m

Physical processors

Inter-cluster scheduling

$\Gamma_1(m_1)$ $\Gamma_2(m_2)$ . . . $\Gamma_k(m_k)$

Virtual processors

τ_{x_1} τ_{x_2} . . . τ_{x_k}

Task clusters

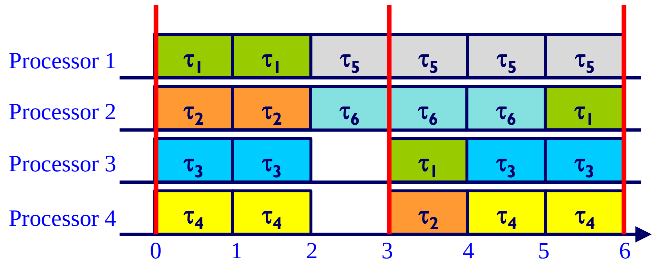
- Each Γ_i is resource required to schedule τ_{x_i} in cluster ***VC***$_i$
- Each m_i ($<= m$) denotes maximum number of physical processors that can be assigned to ***VC***$_i$ at any instant

Counter-example for Partitioned and Global

- Task set and number of processors
  - $\tau_1=\tau_2=\tau_3=\tau_4=(3,2,3)$, $\tau_5=(6,4,6)$, and $\tau_6=(6,3,6)$, m=4
- No Partitioning technique can work
  - Some processor needs to be assigned two tasks which is not possible (maximum utilization that can be assigned to any processor is 1)

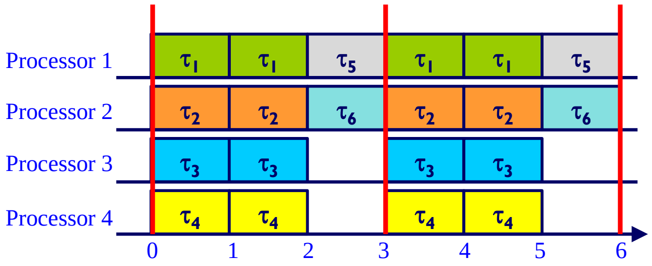
Counter-example for Partitioned and Global

- Task set and number of processors
  - $\tau_1=\tau_2=\tau_3=\tau_4=(3,2,3)$, $\tau_5=(6,4,6)$, and $\tau_6=(6,3,6)$, m=4
- Schedule under global EDF/EDZL (earliest deadline until zero laxity)/LLF
  - Task τ_2 misses its deadline



Counter-example for Partitioned and Global

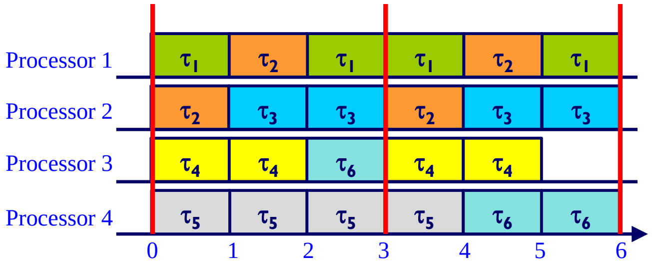
- Task set and number of processors
  - $\tau_1=\tau_2=\tau_3=\tau_4=(3,2,3)$, $\tau_5=(6,4,6)$, and $\tau_6=(6,3,6)$, m=4
- Schedule under global fp-EDF/US-EDF (highest priority to high utilization tasks $\tau_1,\ldots,\tau_5$)
  - Task τ_6 misses its deadline



Processor 1	τ_1	τ_1	τ_5	τ_1	τ_1	τ_5
Processor 2	τ_2	τ_2	τ_6	τ_2	τ_2	τ_6
Processor 3	τ_3	τ_3		τ_3	τ_3	
Processor 4	τ_4	τ_4		τ_4	τ_4	

Virtual Clustering

- Task set and number of processors
  - $\tau_1=\tau_2=\tau_3=\tau_4=(3,2,3)$, $\tau_5=(6,4,6)$, and $\tau_6=(6,3,6)$, m=4
- Schedule under clustered scheduling
  - τ_1, τ_2, τ_3 scheduled on processors 1 and 2
  - τ_4, τ_5, τ_6 scheduled on processors 3 and 4



Processor 1	τ_1	τ_2	τ_1	τ_1	τ_2	τ_1
Processor 2	τ_2	τ_3	τ_3	τ_2	τ_3	τ_3
Processor 3	τ_4	τ_4	τ_6	τ_4	τ_4	
Processor 4	τ_5	τ_5	τ_5	τ_5	τ_6	τ_6

Virtual Clustering

- Two-level hierarchical scheduler
  - Intra-cluster schedulers for tasks within clusters
  - Inter-cluster schedulers for clusters on the platform (clusters can share some physical processors)
- Concurrency bound for each cluster
  - Abstract concurrency constraints of tasks within cluster
  - Helps regulate resource access (e.g., Caches)
- Have virtual clusters been used before?
  - Supertasks[MoRa99], Megatasks[ACD06]
  - Results restricted to Pfair schedulers (not generalizable)

MODE CHANGE PROTOCOLS