Real-Time Systems

L. Sha et. al., “Priority Inheritance Protocols : An Approach to Real-Time Synchronization”, IEEE Transactions on Computers, 39(9), 1990

Contents

- Background and Motivation
- Objectives
- The Priority Inversion Problem
- PIP (Priority Inheritance Protocol)
- PCP (Priority Ceiling Protocol)
- Schedulability Analysis

Background and Motivation

- A direct application of commonly used synchronization primitives such as semaphores, monitors, or the Ada rendezvous can lead to uncontrolled *priority inversion*.
  - a situation in which a higher priority job is blocked by lower priority jobs for an *indefinite* period of time
- Priority inversion is a serious problem in real-time systems by adversely affecting both the schedulability and predictability of real-time systems.
  - Sources of priority inversion
    - non-preemptible regions of code
    - interrupts
    - non-unique priorities for some tasks (if there are not enough priority levels)
    - FIFO queues
    - synchronization and mutual exclusion

Objectives

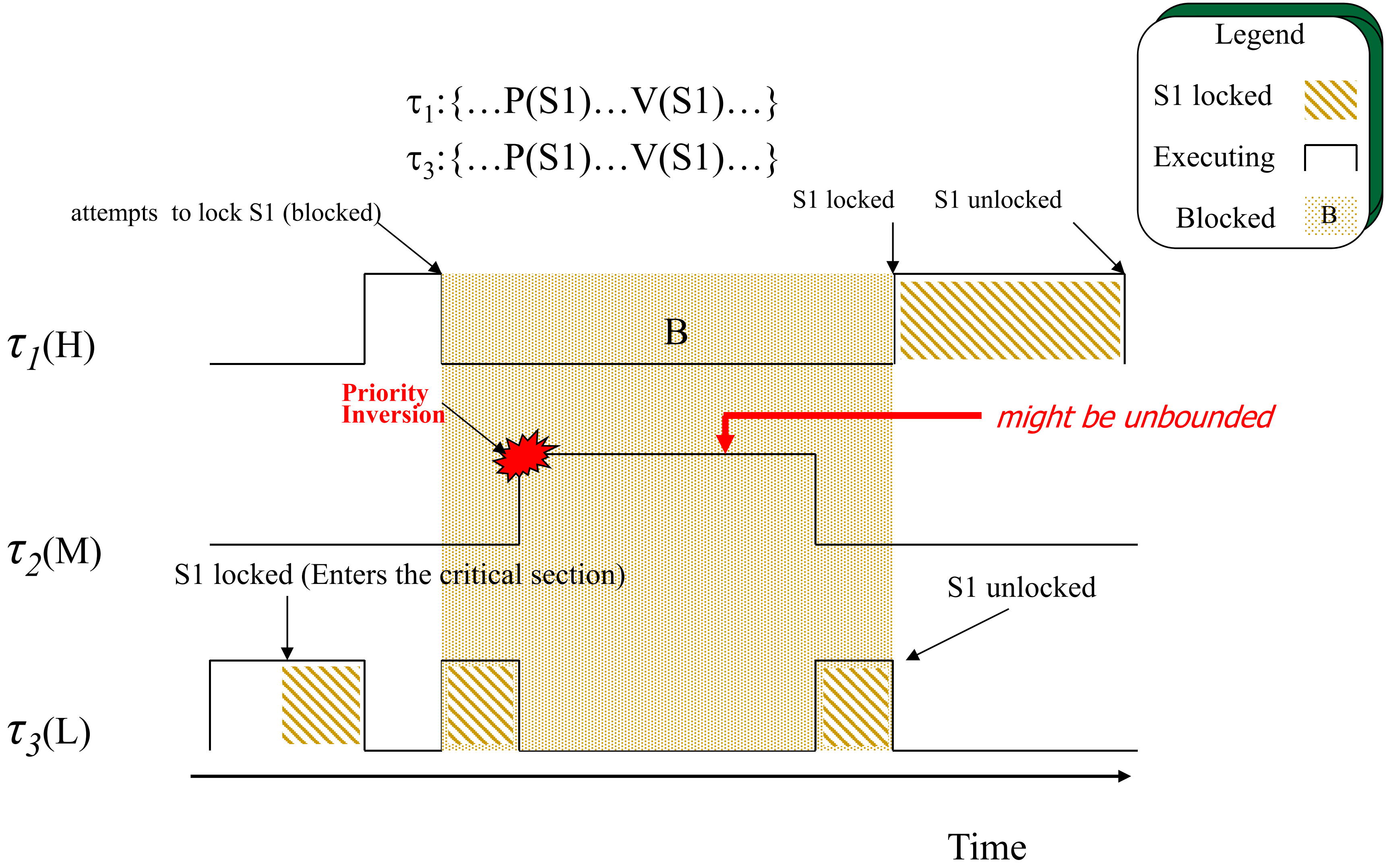
- Investigate the synchronization problem in the context of priority-driven preemptive scheduling
- Show that both protocols solve this uncontrolled priority inversion problem
  - PIP (Priority Inheritance Protocol)
  - PCP (Priority Ceiling Protocol)

  → To prevent unbounded priority inversion
- Schedulability Test with RMS

Priority Inversion Problem

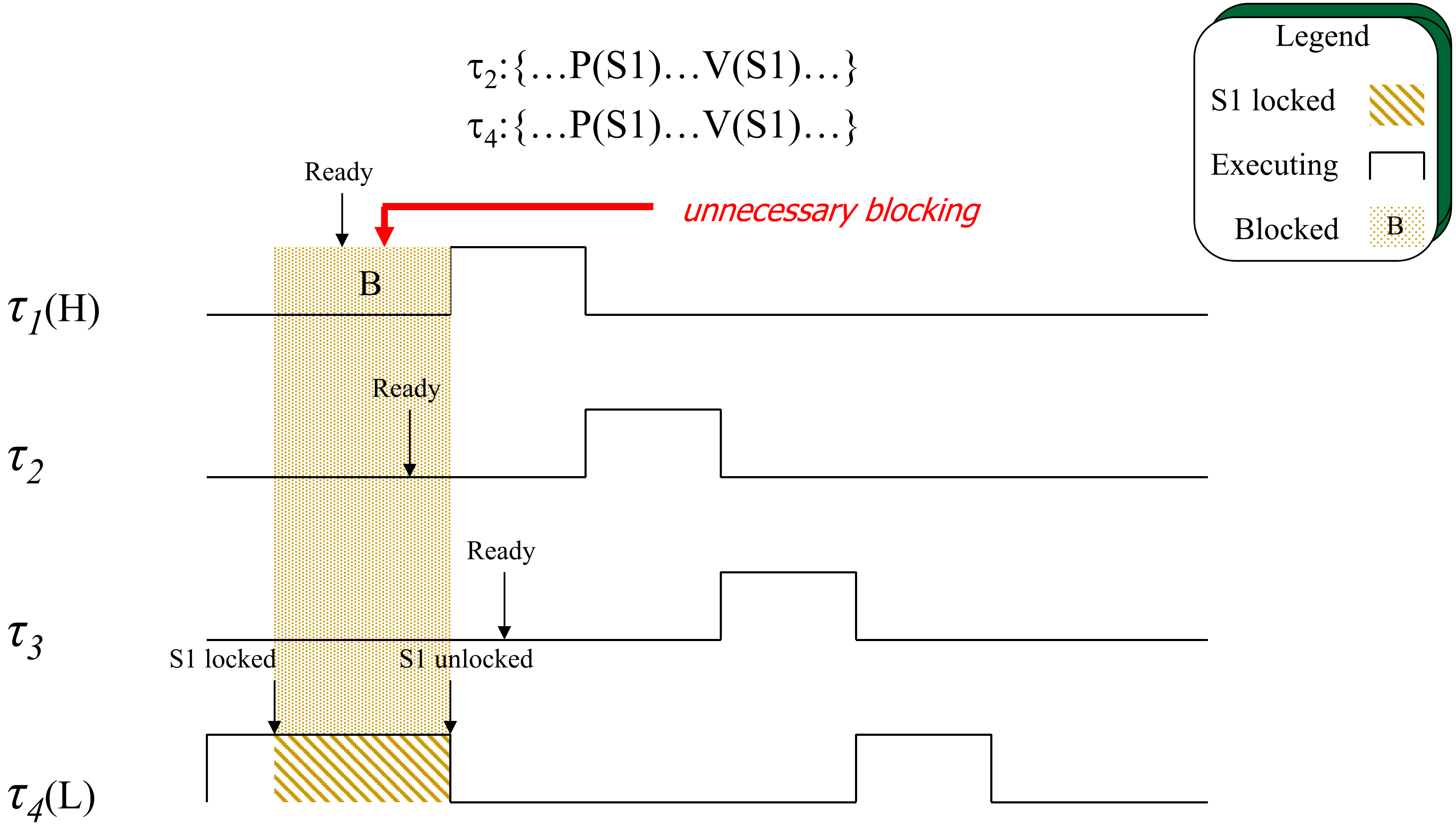
- Terms
  - Priority inversion
    - Phenomenon where a higher priority job is blocked by lower priority jobs
    - Common situation arises when two jobs attempt to access shared data.
      - To maintain consistency, the access must be serialized.
  - Blocking
    - A form of priority inversion where a higher priority job must wait for the processing of a lower priority job
- Critical sections are sections of code that use a resource, during which other tasks must not use that resource.
  - It is possible to guard critical sections by locking semaphores

Priority Inversion: Illustrated



Priority Inversion: Non-preemptive

τ_2:{…P(S1)…V(S1)…}
τ_4:{…P(S1)…V(S1)…}

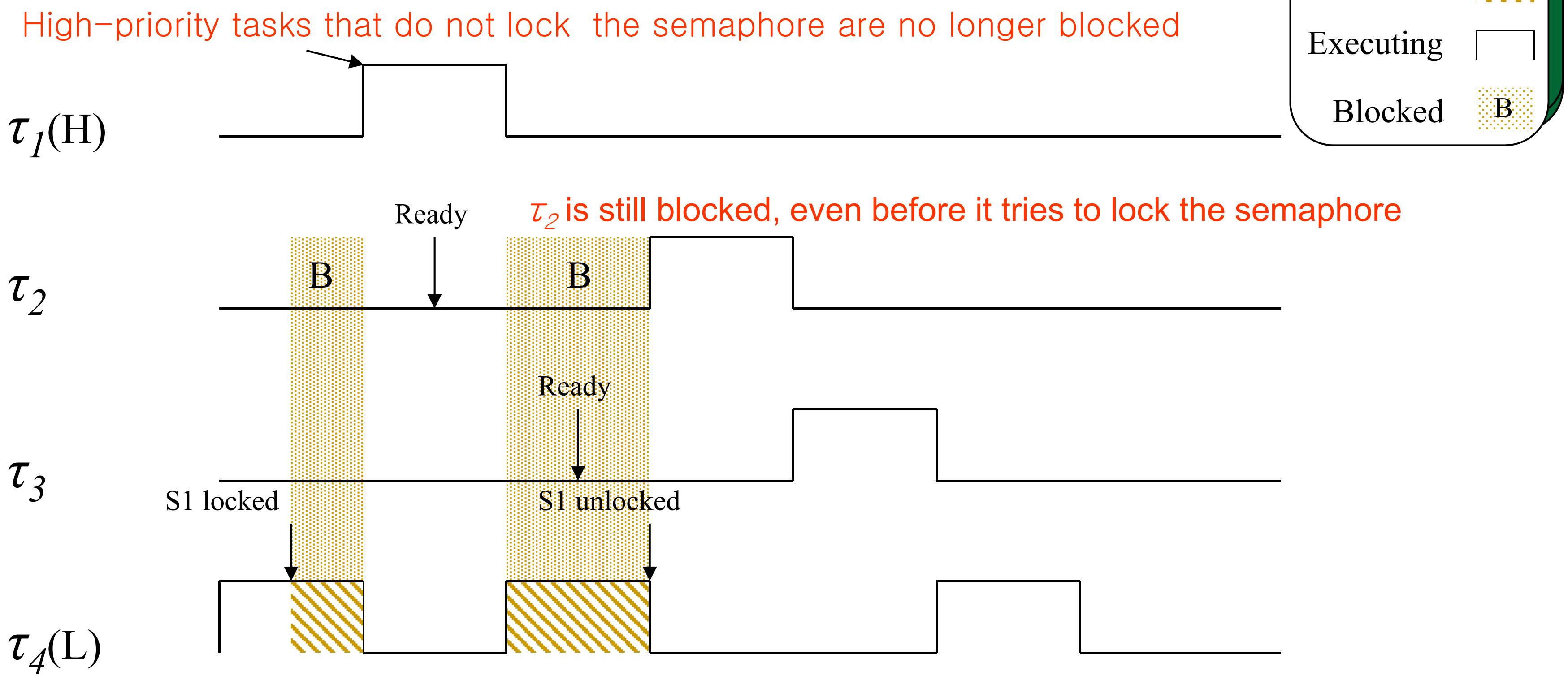


Disallow preemption during the execution of all critical sections!!

Highest-Lockers Priority Protocol

τ_2:{…P(S1)…V(S1)…}

τ_4:{…P(S1)…V(S1)…}



Execute τ_4 using the priority of the highest-priority task that may lock the semaphore (τ_2)

Problem Statement

- Notations
  - J_i : job
  - $z_{i,j}$: j^{th} critical section of job J_i
  - $S_{i,j}$: semaphore guarding the critical section $z_{i,j}$
  - $R_{i,j}$: resource associated with $z_{i,j}$
  - $z_{i,j} \subset z_{i,k}$: $z_{i,j}$ is entirely contained in $z_{i,k}$

Assumptions

- Jobs $J_1, J_2, \ldots, J_n$ are listed in descending order of nominal priority, with J_1 having the highest nominal priority
- Jobs do not suspend themselves
- The critical sections used by any task are properly nested
  - that is, given any pair $z_{i,j}$ and $z_{i,k}$, then either $z_{i,j} \subset z_{i,k}$, $z_{i,k} \subset z_{i,j}$, or $z_{i,j} \cap z_{i,k} = \emptyset$
- Critical sections are guarded by binary semaphores.
  - This means that only one job at a time can be within the critical section corresponding to a particular semaphore S_k

Basic Priority Inheritance Protocol (PIP)

Basic Inheritance Protocol (BIP)

- Basic idea
  - When a job J blocks one or more higher priority jobs, it ignores its original priority assignment and executes its critical section at the highest priority level of all the jobs it blocks
  - Allows lower-priority tasks to <u>temporarily</u> inherit higher priorities: *if they are executing within a critical section* and *if they are blocking a higher-priority task*.
    - Prevents medium-priority tasks from preempting the low-priority task and prolonging the blocking duration experienced by a high-priority task.
  - After exiting its critical section, job J returns to its original priority level

Priority Inheritance Protocol (PIP)

- Definition of the basic protocol
  - 1) Job J, which has the highest priority among the jobs ready to run, is assigned the processor.

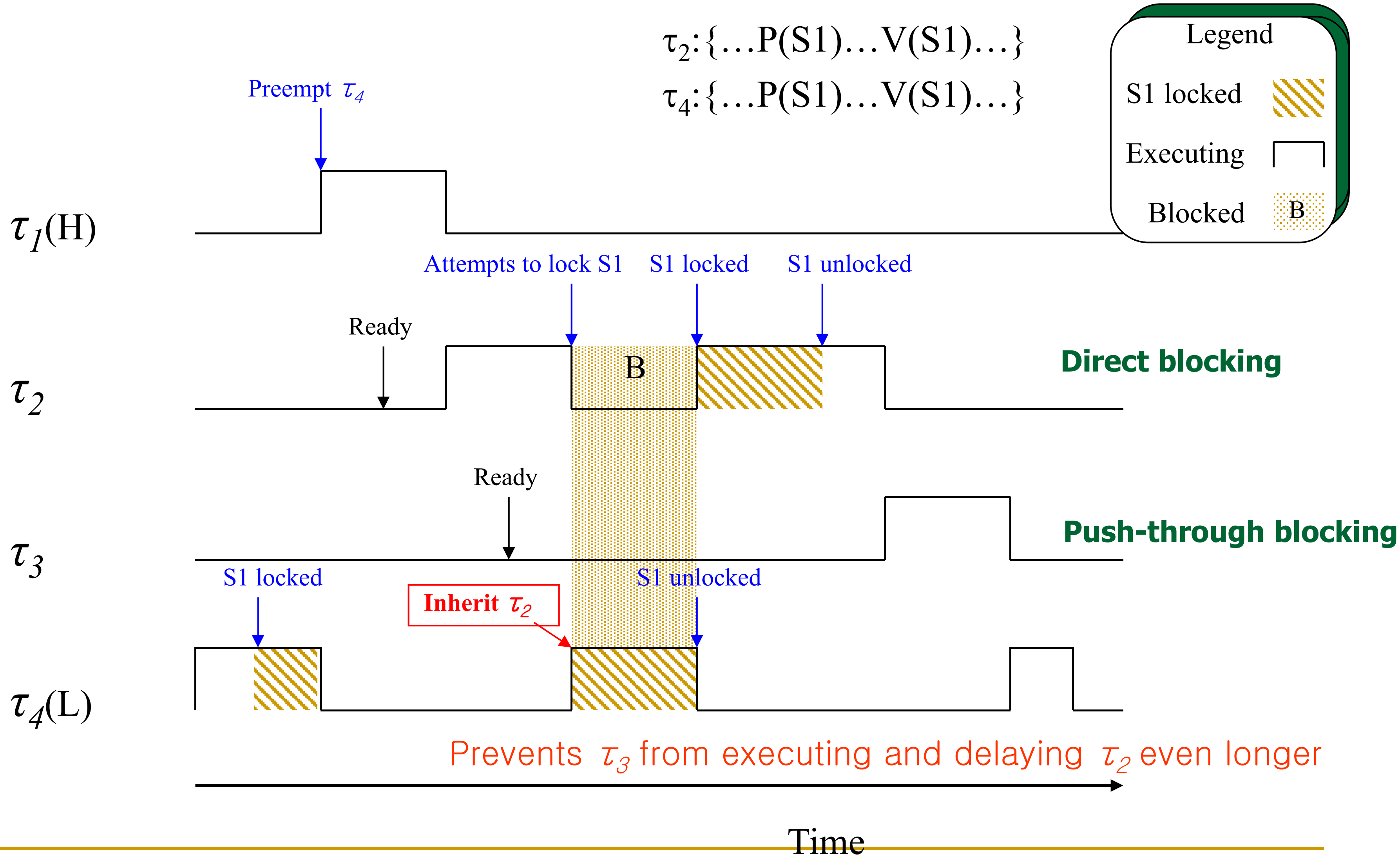
    - Job J will be blocked, if semaphore S has been already locked.

    - When Job J exits its critical section, critical section unlocked, and the highest priority job blocked by job J will be awakened
  - 2) If job J blocks higher priority jobs, J inherits the highest priority of the jobs blocked by J

    - When J exits a critical section, it resumes the priority it had at the point of entry into the critical section
  - 3) Priority inheritance is transitive.

    - Job J_3 blocks J_2, and J_2 blocks J_1, J_3 would inherit the priority of J_1 via J_2
  - 4) Job J can preempt another job J_L if job J is not blocked and its priority is higher than the priority, inherited or assigned, at which job J_L is executing

BIP Illustrated

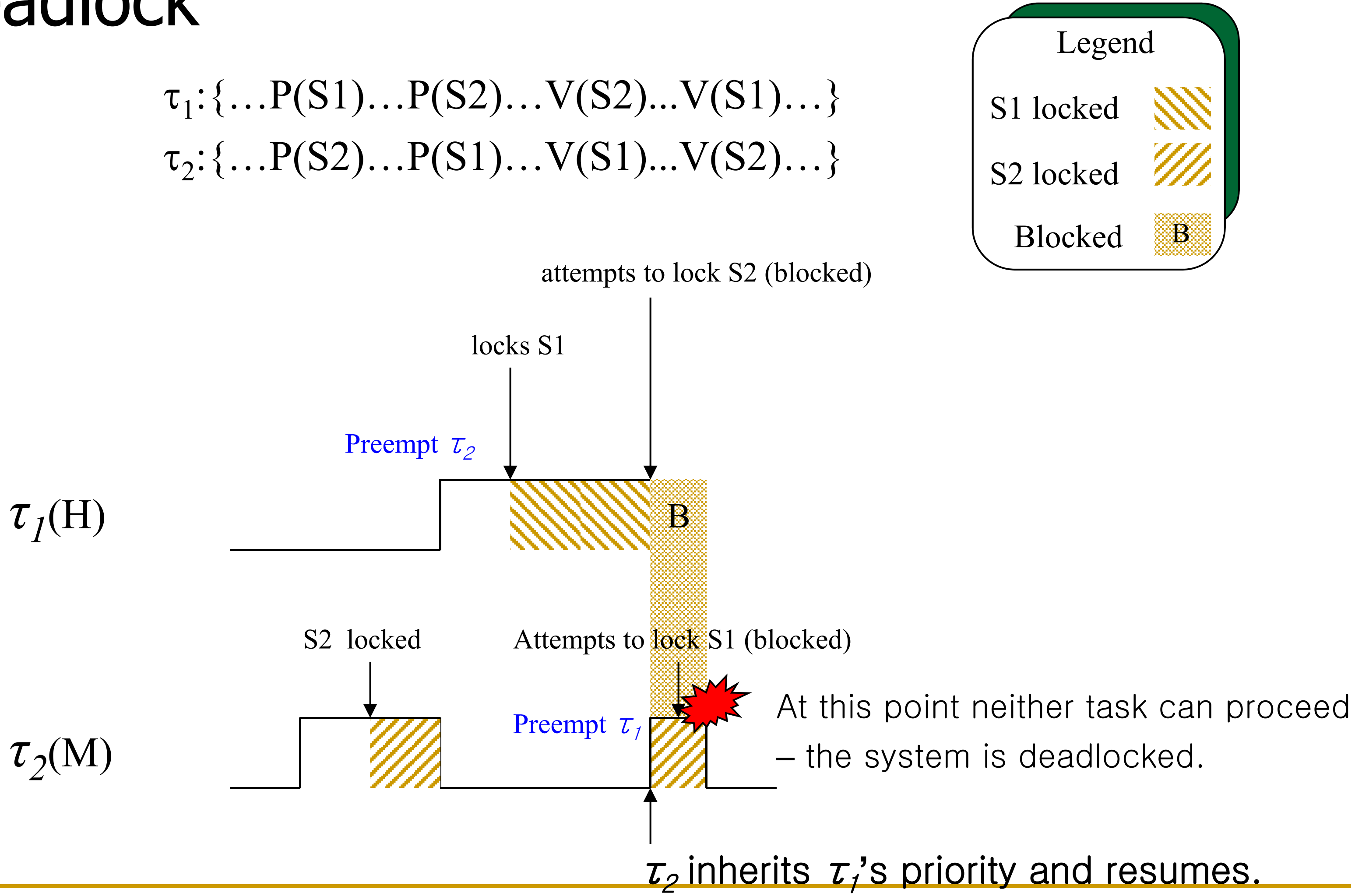


Potential Drawback

- Deadlock

τ_1:{…P(S1)…P(S2)…V(S2)...V(S1)…}

τ_2:{…P(S2)…P(S1)…V(S1)...V(S2)…}



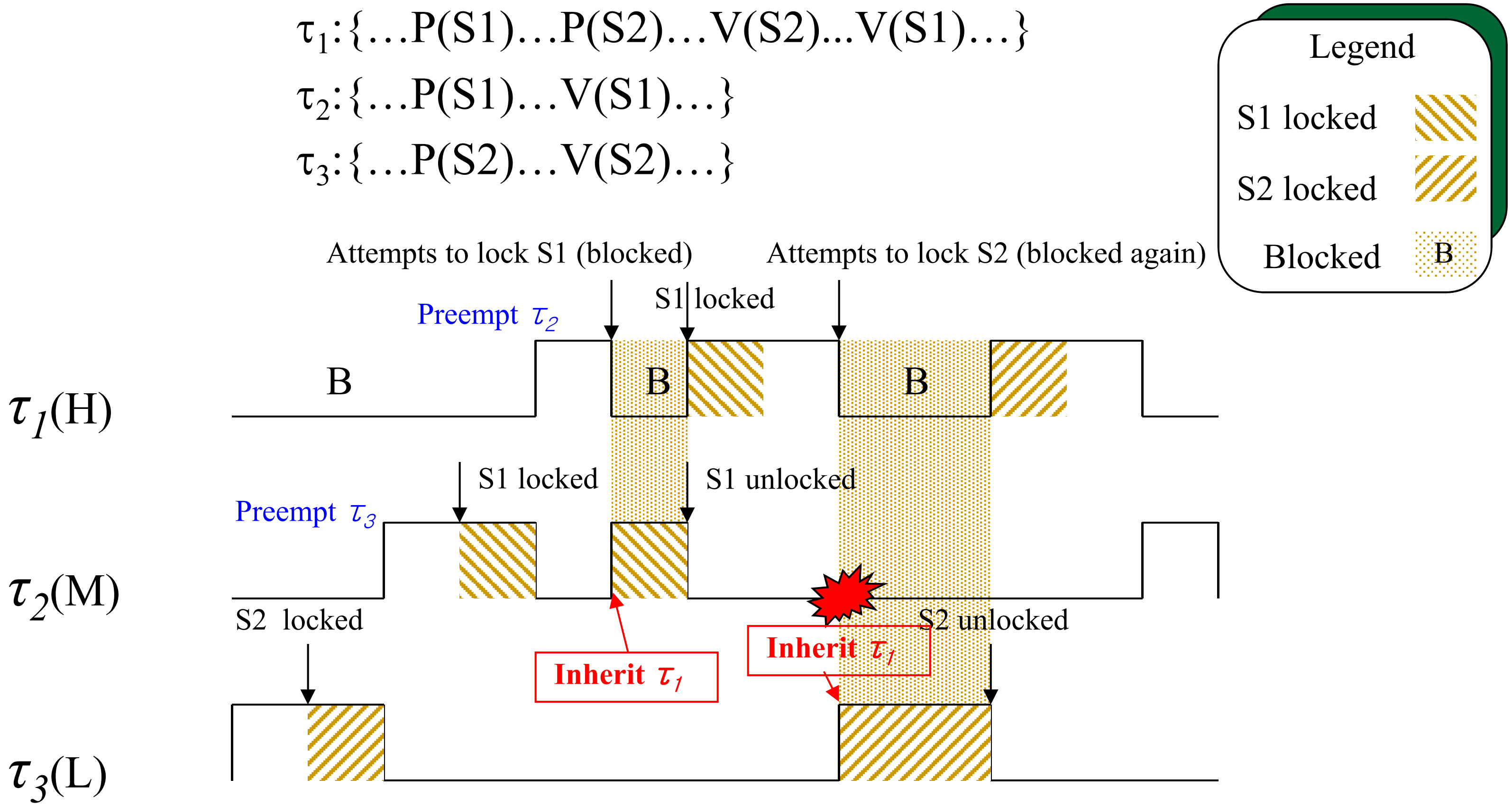
Potential Drawback

- Chained Blocking

τ_1:{…P(S1)…P(S2)…V(S2)...V(S1)…}

τ_2:{…P(S1)…V(S1)…}

τ_3:{…P(S2)…V(S2)…}



When a high-priority task shares more than one semaphore with lower-priority tasks, it may be blocked on each request to lock a semaphore --> chained blocking.

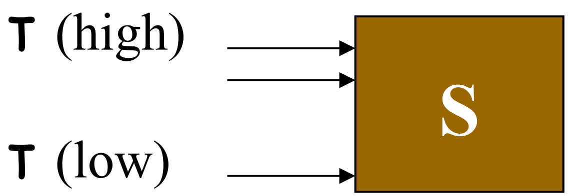
Priority Ceiling Protocol (PCP)

Priority Ceiling Protocol (PCP)

- Goal
  - To prevent the formation of deadlocks and of chained blocking
- Basic idea
  - When a job J preempts the critical section of another job and executes its own critical section z, the priority of J should be guaranteed to be higher than the inherited priorities of all the preempted critical sections
  - If this condition cannot be satisfied, job J is denied entry into the critical section z and suspended, and the job that blocks J inherits J's priority
  - Allow a job J to start a new critical section only if J's priority is higher than all priority ceilings of all the semaphores locked by jobs other than J

Priority Ceiling

- Priority ceiling of a semaphore S
  - simply the priority of the highest priority task that <u>may</u> lock semaphore S
- System ceiling
  - the maximum ceiling of all semaphores currently locked by other tasks
- The idea behind PCP
  - To create a total priority ordering of executing and suspended critical sections



priority ceiling of semaphore S is the priority of T (high)

Priority Ceiling - Example

Critical section	Accessed by	Priority ceiling
S_1	τ_1, τ_2	$P(\tau_1)$
S_2	τ_1, τ_2, τ_3	$P(\tau_1)$
S_3	τ_3	$P(\tau_3)$
S_4	τ_2, τ_3	$P(\tau_2)$

PCP Rules (1)

- *Preemption*: A task with a higher execution priority always preempts tasks with lower execution priorities.
- *Ceiling*: A task cannot enter its critical section unless its priority is higher than the system ceiling.
- *Inheritance*: A lower priority task that blocks a higher priority task J_h inherits the priority of task J_h.

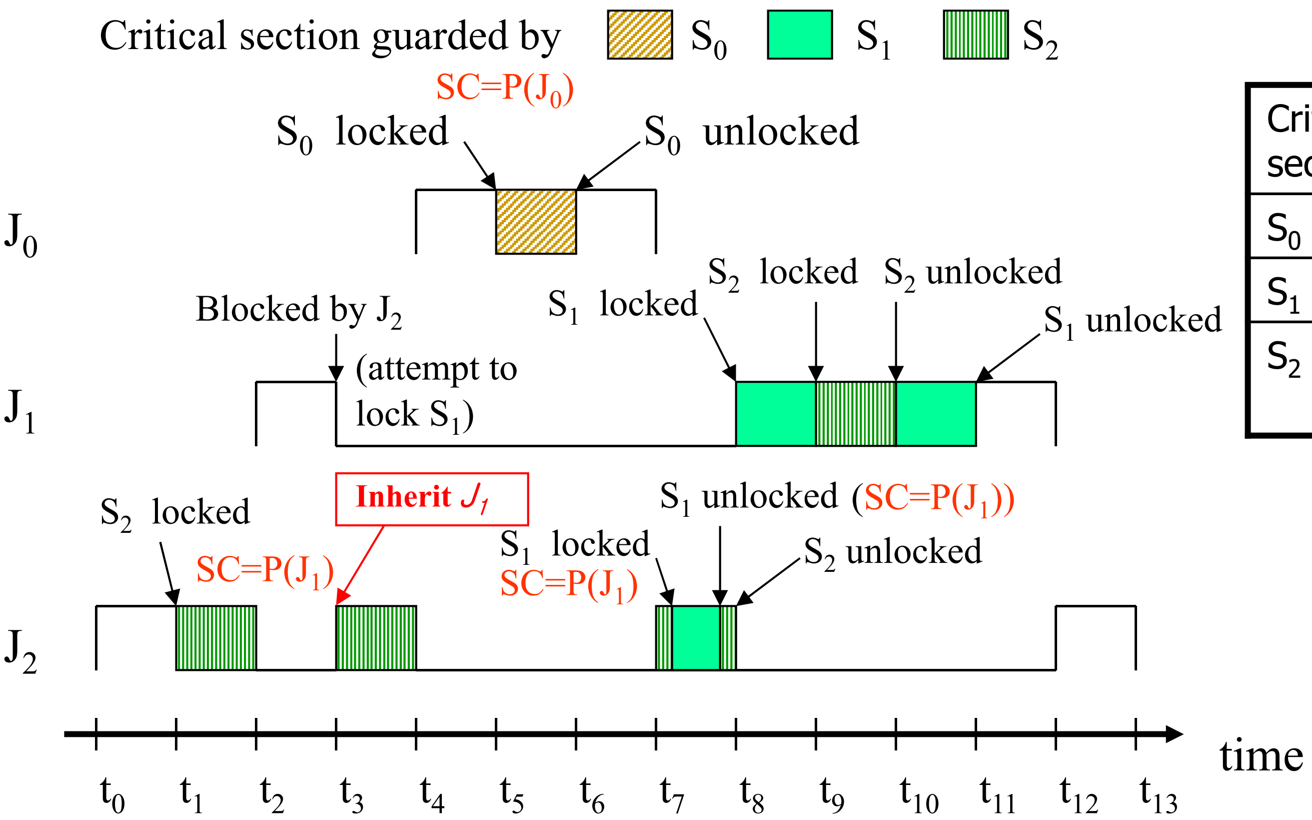
  [When there is only one semaphore, PCP works just like BIP.]

PCP Rules (2)

1. When J wants to enter the c. s.
   - its priority must be higher than system ceiling
   - when exiting, wake up the highest priority task among the blocked
2. While running inside the c. s.
   - always inherits highest priority among the blocked
3. *When J does not want to enter the c. s.*
   - can preempt a lower priority task
4. When J completes execution normally
   - if there are many tasks of same priority ready for execution, schedule the task blocking other tasks

Example 1

$J_0 = \{ \ldots, P(S_0), \ldots, V(S_0), \ldots \}$
$J_1 = \{ \ldots, P(S_1), \ldots, P(S_2), \ldots, V(S_2), \ldots, V(S_1), \ldots \}$
$J_2 = \{ \ldots, P(S_2), \ldots, P(S_1), \ldots, V(S_1), \ldots, V(S_2), \ldots \}$



Critical section	Accessed by	Priority ceiling
S_0	J_0	$P(J_0)$
S_1	J_1, J_2	$P(J_1)$
S_2	J_1, J_2	$P(J_1)$

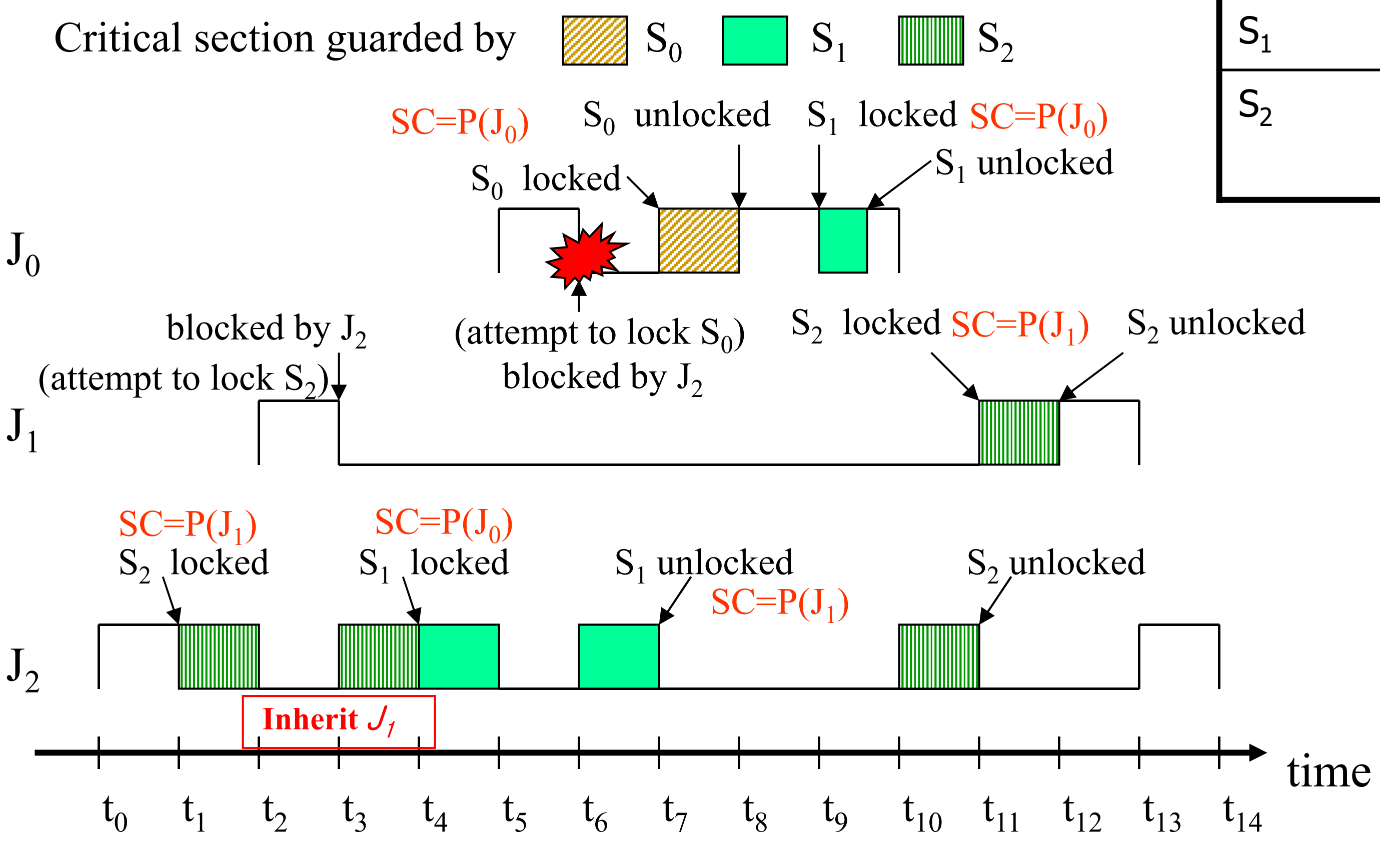
Example 2

$J_0 = \{\ \ldots, P(S_0), \ldots, V(S_0), \ldots, P(S_1), \ldots, V(S_1), \ldots\ \}$

$J_1 = \{\ \ldots, P(S_2), \ldots, V(S_2), \ldots\ \}$

$J_2 = \{\ \ldots, P(S_2), \ldots, P(S_1), \ldots, V(S_1), \ldots, V(S_2), \ldots\ \}$

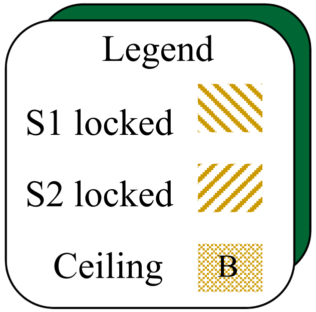
Critical section	Accessed by	Priority ceiling
S_0	J_0	$P(J_0)$
S_1	J_0, J_2	$P(J_0)$
S_2	J_1, J_2	$P(J_1)$



Deadlock Avoidance

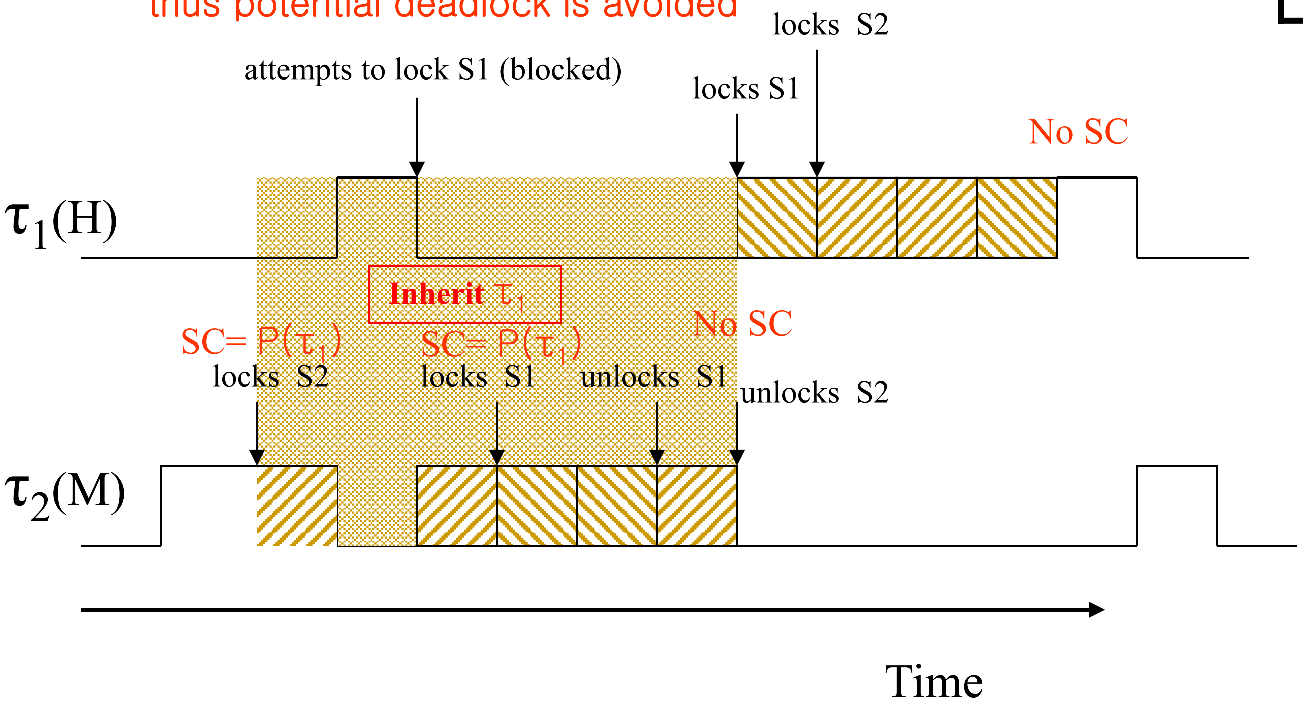
τ_1:{…P(S1)…P(S2)…V(S2)...V(S1)…}

τ_2:{…P(S2)…P(S1)…V(S1)...V(S2)…}



Critical section	Accessed by	Priority ceiling
S_1	τ_1 τ_2	$P(\tau_1)$
S_2	τ_1 τ_2	$P(\tau_1)$

τ_1 is denied to lock S1 and
thus potential deadlock is avoided

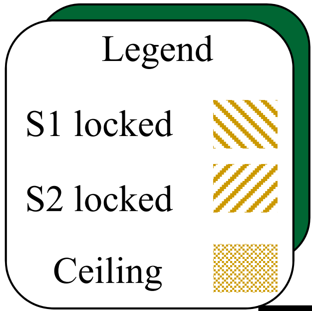


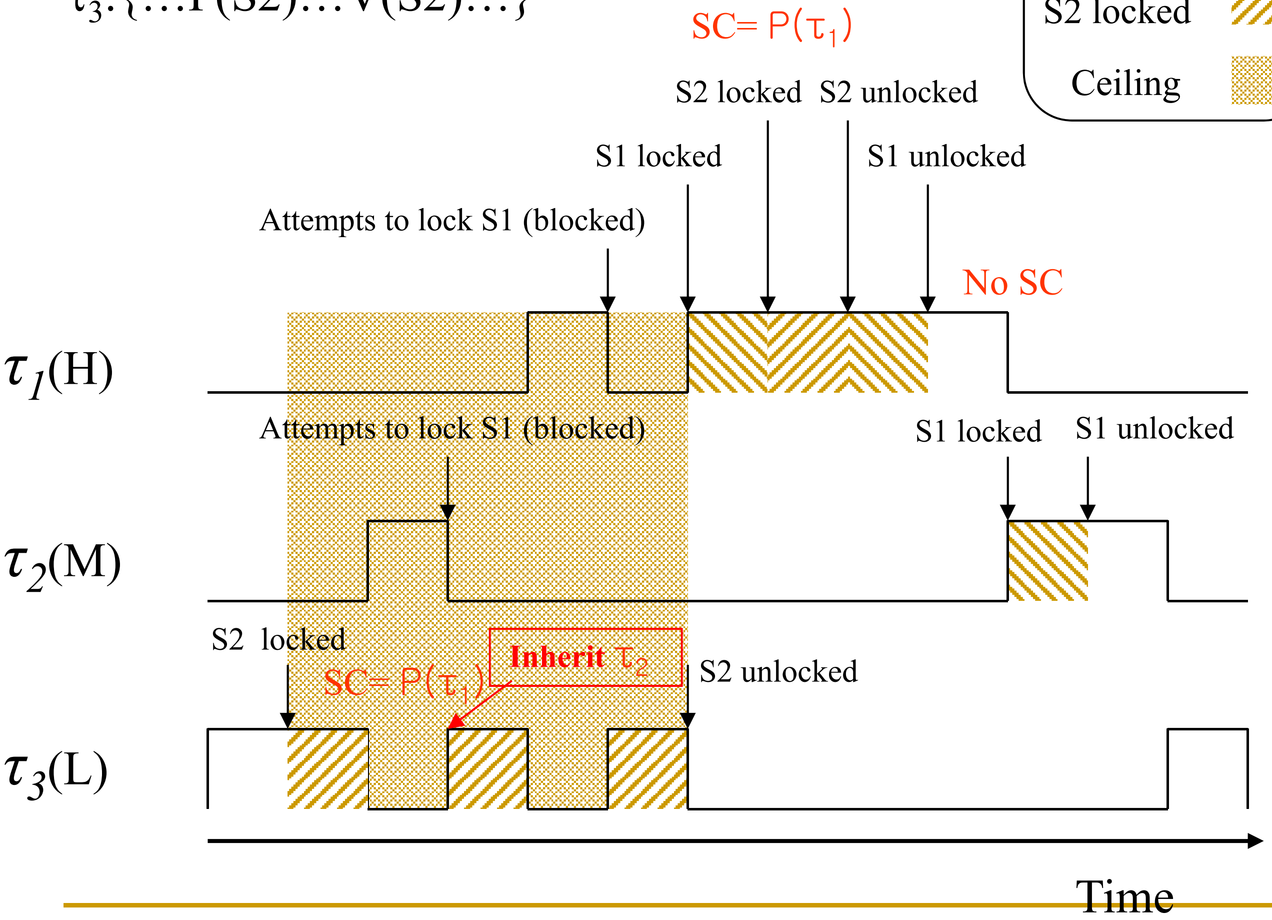
Blocked At Most Once

τ_1:{…P(S1)…P(S2)…V(S2)...V(S1)…}

τ_2:{…P(S1)…V(S1)…}

τ_3:{…P(S2)…V(S2)…}





Critical section	Accessed by	Priority ceiling
S_1	$\tau_1\ \tau_2$	$P(\tau_1)$
S_2	$\tau_1\ \tau_3$	$P(\tau_1)$

Summary of synchronization protocols

- No preemption
  - do not allow preemption during execution of critical sections
- Highest locker's priority
  - execute critical sections with the priority of the highest priority task that may lock the semaphore
- Priority inheritance
  - when a lower priority task blocks the execution of a higher priority task, it inherits the priority of the task it blocks
- Priority ceiling
  - priority inheritance plus priority ceiling rule for locking semaphores

Summary of synchronization protocols

Protocol	Bounded priority inversion	Blocked at most once	Deadlock aviodance
Nonpremptible critical sections	Yes	Yes[1]	Yes[1]
Highest locker's priority	Yes	Yes[1]	Yes[1]
Basic inheritance	Yes	No	No
Priority ceiling	Yes	Yes[2]	Yes

1 Only if tasks do not suspend within critical sections

2 PCP is not affected if tasks suspend within critical sections

Real-Time Systems

Scheduling Algorithm
– EDF (Earliest Deadline First)

[Liu73] C. L. Liu and J.W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment", Journal of the ACM, 1973

[Baruah90] S. K. Baruah, L.E.Rosier, and R.R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor", Journal of Real-time Systems, 1990

[Jeffay93] K. Jeffay and D.L. Stone, "Accounting for interrupt handling costs in dynamic priority task systems", RTSS, 1993

Contents

- EDF (Earliest Deadline First) Scheduling
- Schedulability Analysis based on Utilization Bound

EDF (Earliest Deadline First)

- Assumptions
  - Same as RM (Rate Monotonic)
- Priority Assignment
  - Priorities are assigned to tasks according to the deadlines of their current requests.
  - Dynamic priority scheduling
- Run-time activity of scheduler
  - The ready task with the highest priority, i. e., the one with the nearest deadline, is executed.
  - Sort priorities such that tasks with closer deadlines get higher priorities
- [Liu73] (Refer to RM paper, section 7)

Schedulability Analysis

Theorem

For a given set of n tasks, the EDF scheduling algorithm is feasible if and only if

$$(C_1/T_1)+(C_2/T_2)+\ldots+(C_n/T_n) = U \leq 1$$

Proof:

(1) *Only if (necessary condition)*

Let $T = T_1 T_2 \ldots T_n$,

The total demand of computation time in $[0, T]$ is

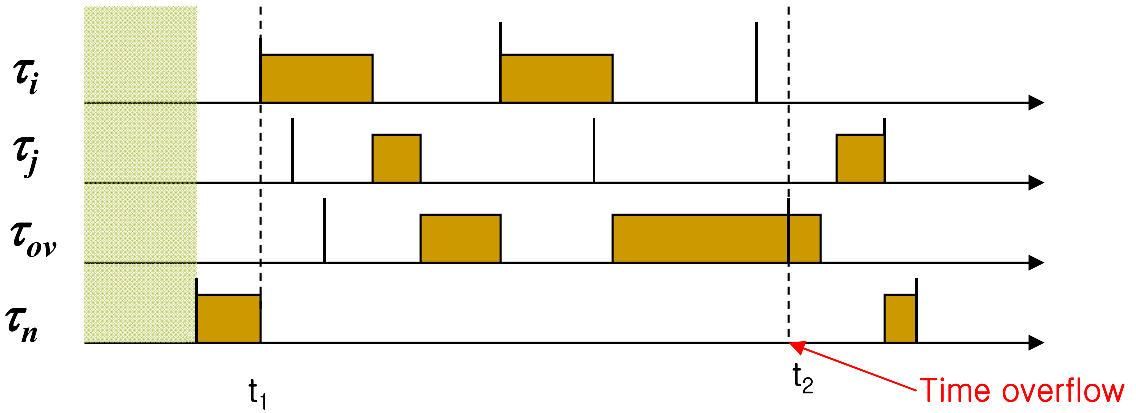
$$\sum_{i=1}^{n} \frac{C_i}{T_i} T = UT$$

If $U > 1$, there is clearly no feasible schedule for the task set since the processor demand UT exceeds the available time T.

Schedulability Analysis (cont'd)

Proof (cont'd):

(2) *if (sufficient condition)*



Assume that the condition U <1 is satisfied and yet the task set is not schedulable

$$C_p(t_1,t_2) = \sum_{r_k \ge t_1, d_k \le t_2} C_k = \sum_{i=1}^{n} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i$$

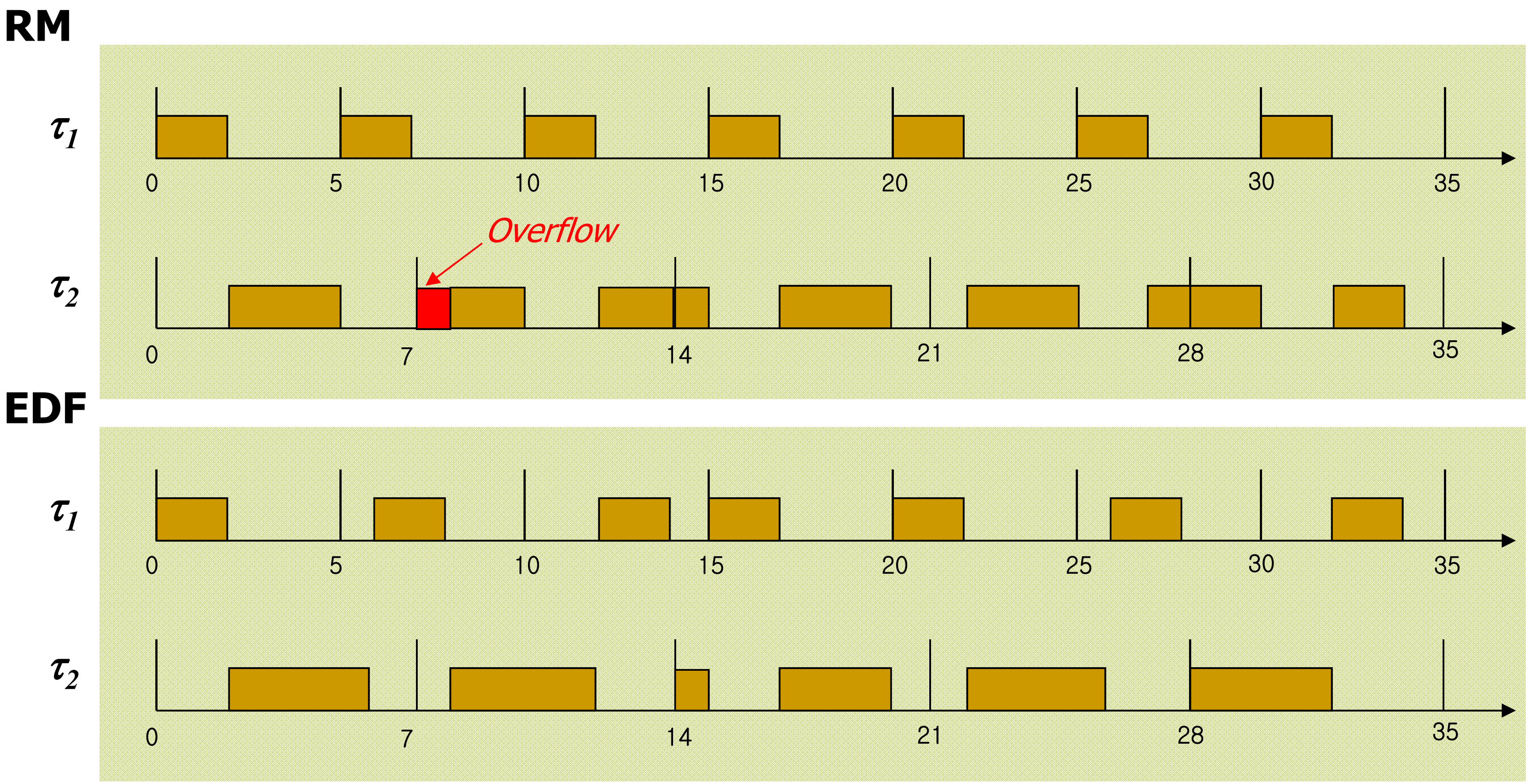
$$C_p(t_1,t_2) = \sum_{i=1}^{n} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i \le \sum_{i=1}^{n} \frac{t_2 - t_1}{T_i} C_i = (t_2 - t_1)U$$

$$(t_2 - t_1) < C_p(t_1,t_2) \le (t_2 - t_1)U$$

That is $U > 1$, which is a contradiction.

Example

- Task set $\tau=\{\tau_1, \tau_2\}$, $\tau_1=(2,5,5)$ and $\tau_2=(4,7,7)$



[Least/Minimum] [Slack Time/Laxity] First

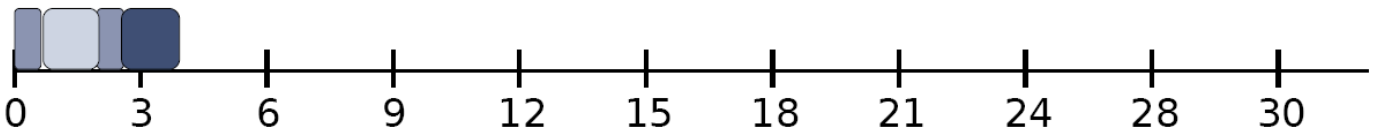
- **Select the task whose slack is the lowest**
- **Slack Time = Laxity:**
  (time to deadline - remaining execution time required to reach deadline)
- **slack time: *d-x-t***
  - ***x* remaining execution time of a job**
  - ***d* absolute deadline**
  - ***t* current time**
  - **dynamic per job, dynamic at task level**
    - **also optimal (analog EDF definition)**

MLF (LSF) Scheduling

- two versions:
  - Strict: slacks are computed at all times
    - ***Each instruction (prohibitively slow)***
    - ***Each timer "tick"***
  - Non-strict: slacks computed only at events (release and completion)
- Scheduler checks slacks of all ready jobs and reorders queue

Non-Strict Example

- T_1: (0.75, 2), T_2: (1.5, 5), T_3: (1.5, 5.1)



- t=0 all Jobs are released
  - **T_1,J_1: 1.25** **T_2,J_1: 3.5** d=5 x=1.5 **T_3,J_1:** d=5.1 x=1.5 **3.6**
  - d.h. T_2,J_1 higher priority than T_3,J_1
- t=2 T_1,J_2 released
  - **T_1,J_2: 1.25** **T_2,J_1: 2.75** d=5 x=0.25 **T_3,J_1: 1.6** d=5.1 x=1.5
  - d.h. T_2,J_1 lower priority than T_3,J_1
- t=2.75 T_1,J_2 completed
  - **T_1,J_2:** **T_2,J_1: 2** d=5 x=0.25 **T_3,J_1: 0.85** d=5.1 x=1.5

Reading Assignment

- **T.P. Baker and Alan Show, “The Cyclic Executive Model and Ada”, Real-Time Systems Journal, 1989**
- **J. P. Lehoczky, L. Sha, and J. K. Strosnider, “Enhanced Aperiodic Responsiveness in Hard Real-Time Environments”, IEEE RTSS, 1987**
- **B. Sprunt, L. Sha, and J. Lehoczky, “Aperiodic Task Scheduling for Hard-Real-Time Systems”, Journal of Real-Time Systems, 1989**