Scheduling with Shared Resources

Unmesh D. Bordoloi

Resources

What is a resource?

- a data structure used by a task during its execution
- variable ...
- an area of main memory
- set of registers of peripheral device

A shared resource

- is used by more than one task
- does not allow simultaneous access
- requires *mutual exclusion*

Critical Section

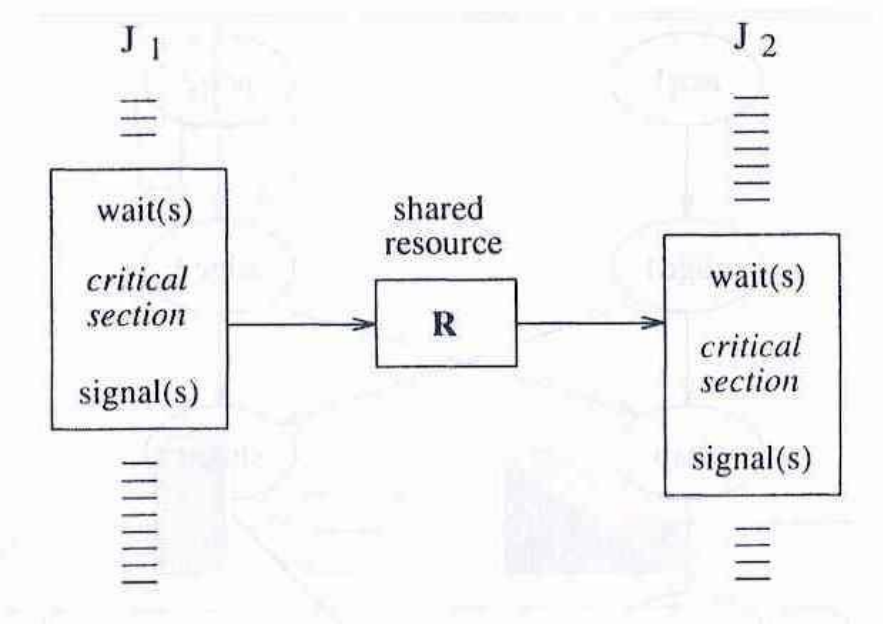
Critical section

is the piece of code belonging to a task that is executing under mutual exclusion constraints

Mutual exclusion is enforced by semaphores

- wait(s): Task is blocked if s=0
- signal(s): Task can access the resource if s=1

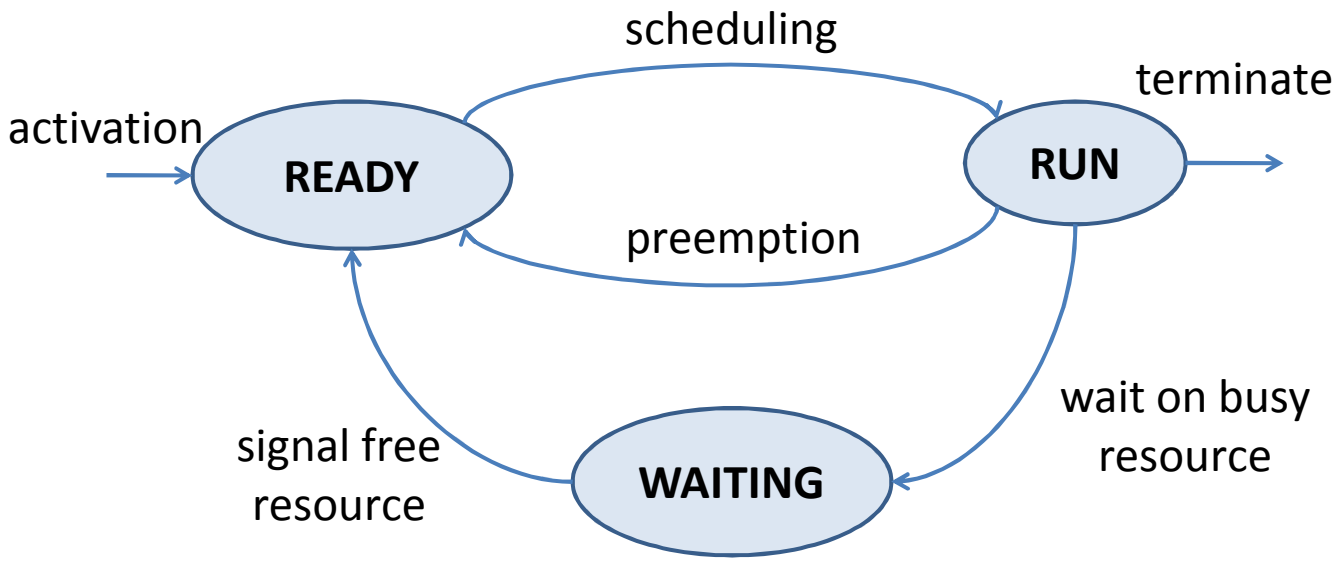
Critical Sections



States of a Task

- A task waiting for an exclusive resource is blocked on that resource
- All tasks blocked on the same resource are kept in a queue with the semaphore of that resource
- When a running tasks executes a *wait* primitive on a locked semaphore, it enters a *wait* state
- When a task currently using resource executes a signal, the semaphore is released
- When a tasks leaves the waiting state because the semaphore has been released it goes to the ready state
- The CPU is then assigned to highest priority tasks amongst the tasks in the ready state

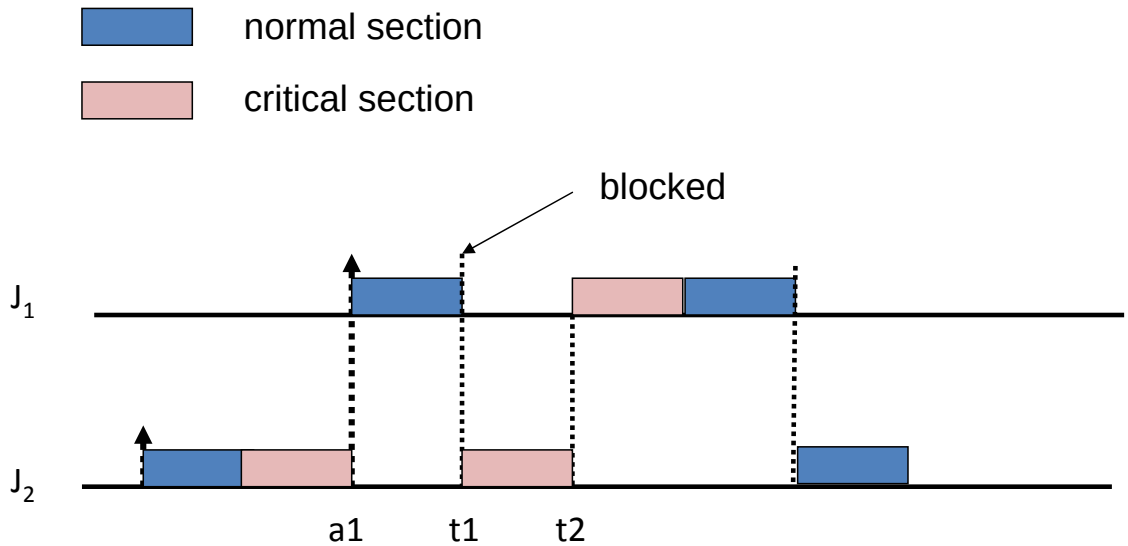
States


scheduling
terminate
activation
READY
RUN
preemption
wait on busy
resource
signal free
resource
WAITING

Shared Resources

Critical sections, states of the task ... we discussed them as necessary backgrounds. What we really want to discuss now is:
(i) Shared resources can cause problems in scheduling
and (ii) How to fix them?

Blocking

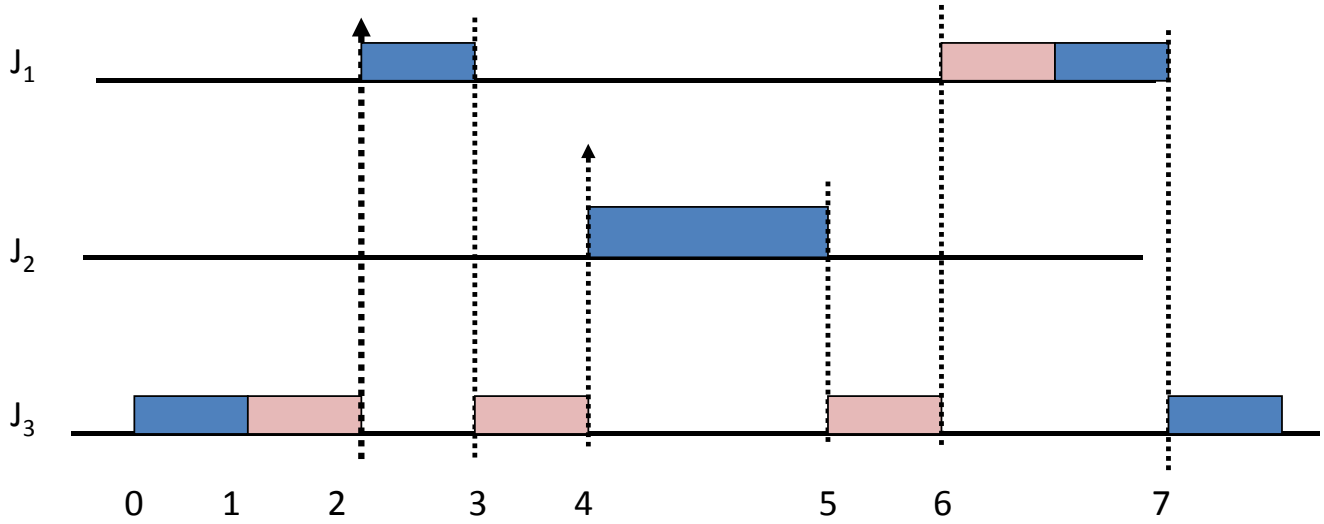


1. J_1 has higher priority than J_2
2. Uniprocessor preemptive environment

Priority Inversion

- Simply considering the blocking delay is not enough
- We must also avoid *priority inversion*

Priority Inversion



- $J1 > J2 > J3$
- $[3, 6]$ is the priority inversion period
- J1 has to wait for the execution of J2 and critical section of J3

Need for Resource Access Protocols

- Multiple tasks are running
- in uniprocessor preemptive environment
- with shared resources
- which may lead to priority inversion!
- Need for resource access protocols!

Avoiding Inversion

Naive protocol

- Prohibit preemption
- Works well only if critical section are short
- Might block higher priority processes that do not even use any shared resources
- Hence, such a naive protocol is not enough. What we need follows...

Resource access protocols

Under static priorities

- Priority inheritance
- Priority ceiling

Under dynamic priorities

- Stack resource

Resource access protocols

Under static priorities

- Priority inheritance
- Priority ceiling

Under dynamic priorities

- Stack resource

Priority Inheritance Protocol

Tasks have nominal and active priorities

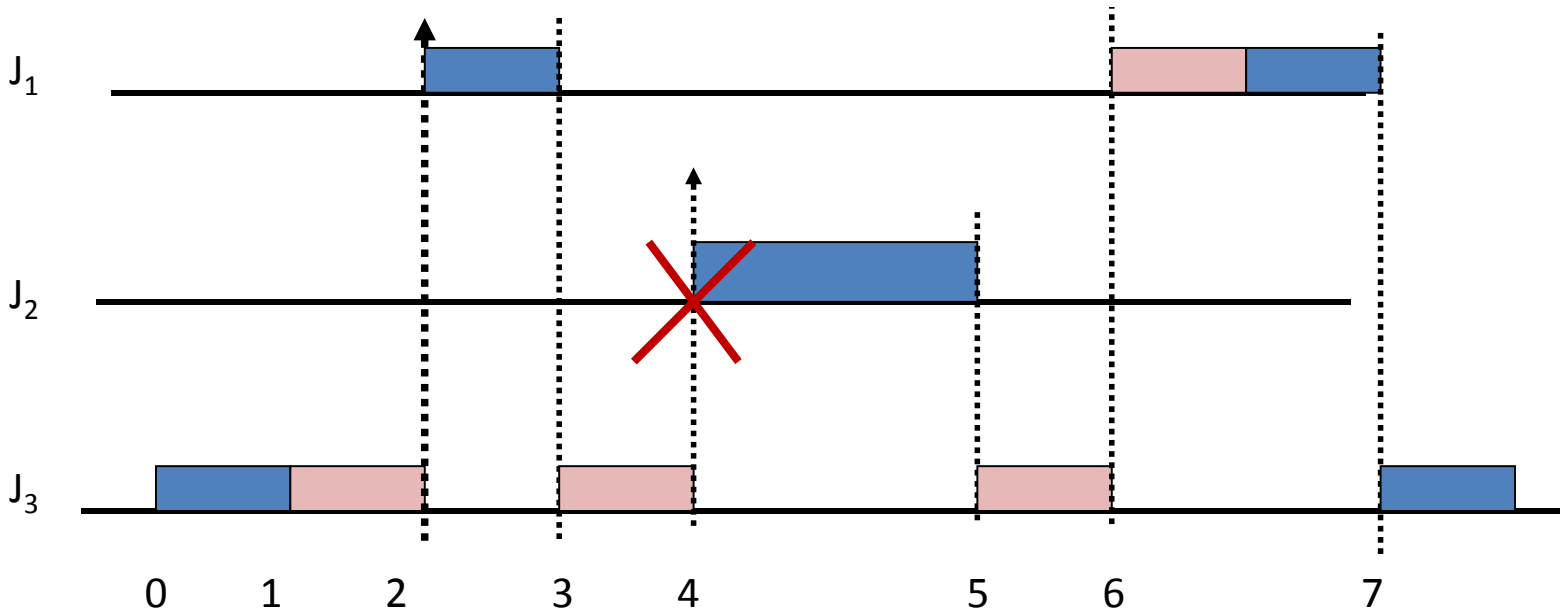
- Nominal priority : assigned by the scheduling algorithm
- Active priority : assigned by the priority inheritance protocol dynamically to avoid priority inversion

Priority Inheritance Protocol

Intuition

- When J_i blocks higher priority tasks, then its active priority is set to the highest of the priorities of the tasks it blocks
- J_i inherits - temporarily - the highest priority of the blocked tasks
- Thus, medium priority tasks which do no share resources with J_i cannot preempt J_i and cannot prolong the blocking of the higher priority tasks

Priority Inheritance Protocol - Example1

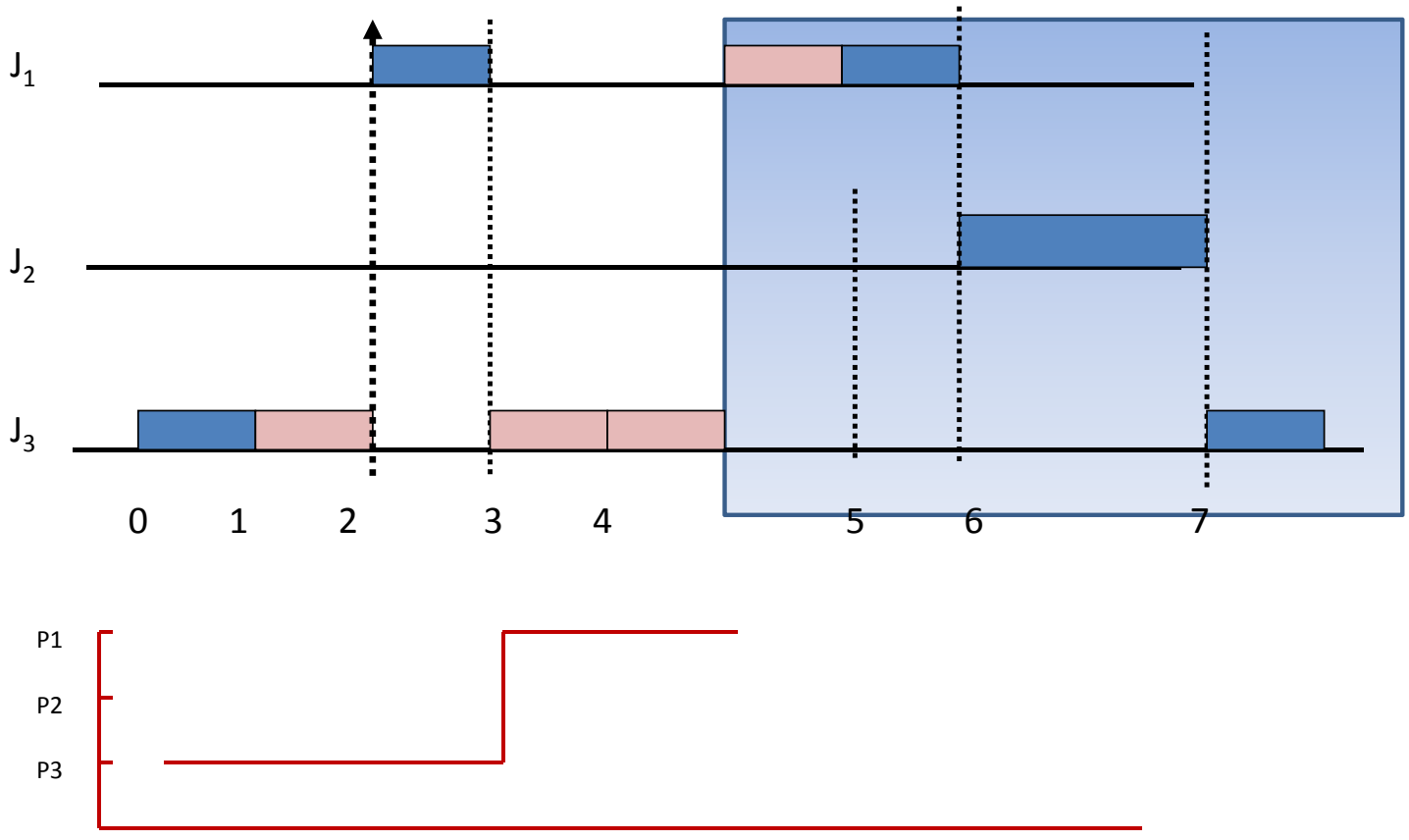


Priority Inheritance Protocol

The working of the protocol

- Jobs are scheduled based on their active priorities
- If J_i tries to enter a critical section and the corresponding resource is being held by J_j then J_i is blocked; it is said to be blocked by J_j.
- When a job is blocked on a semaphore, it transmits its active priority to the job that holds the semaphore; in general, a task inherits the highest priority of the jobs blocked by it.

Priority Inheritance Protocol - Example1



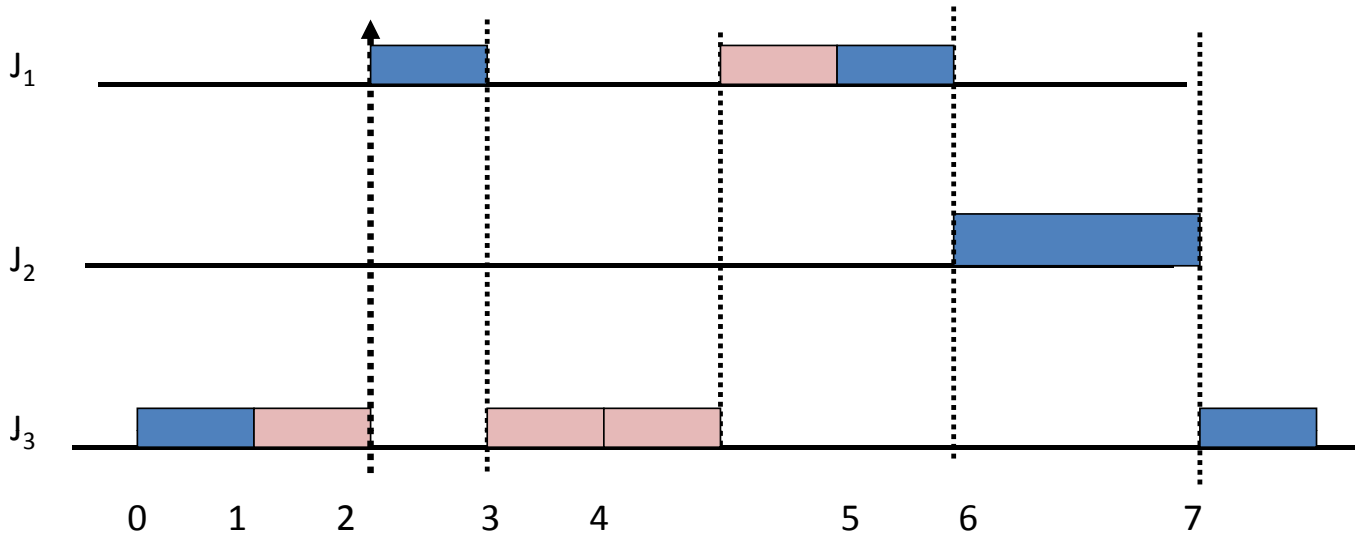
Showing the change in the priority level of the lowest priority task J3

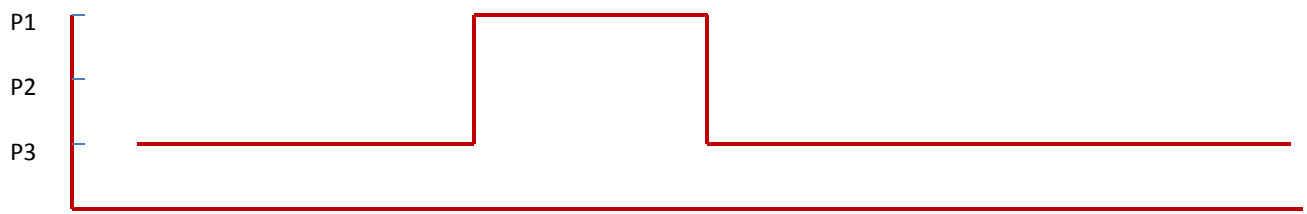
Priority Inheritance Protocol

The working of the protocol

When J_k exits a critical section, it unlocks the semaphore; the job with the highest priority that is blocked on the semaphore, if any, is awakened. The priority of J_k is set to the highest priority of the job it is currently blocking. If none, its priority is set to its nominal one.

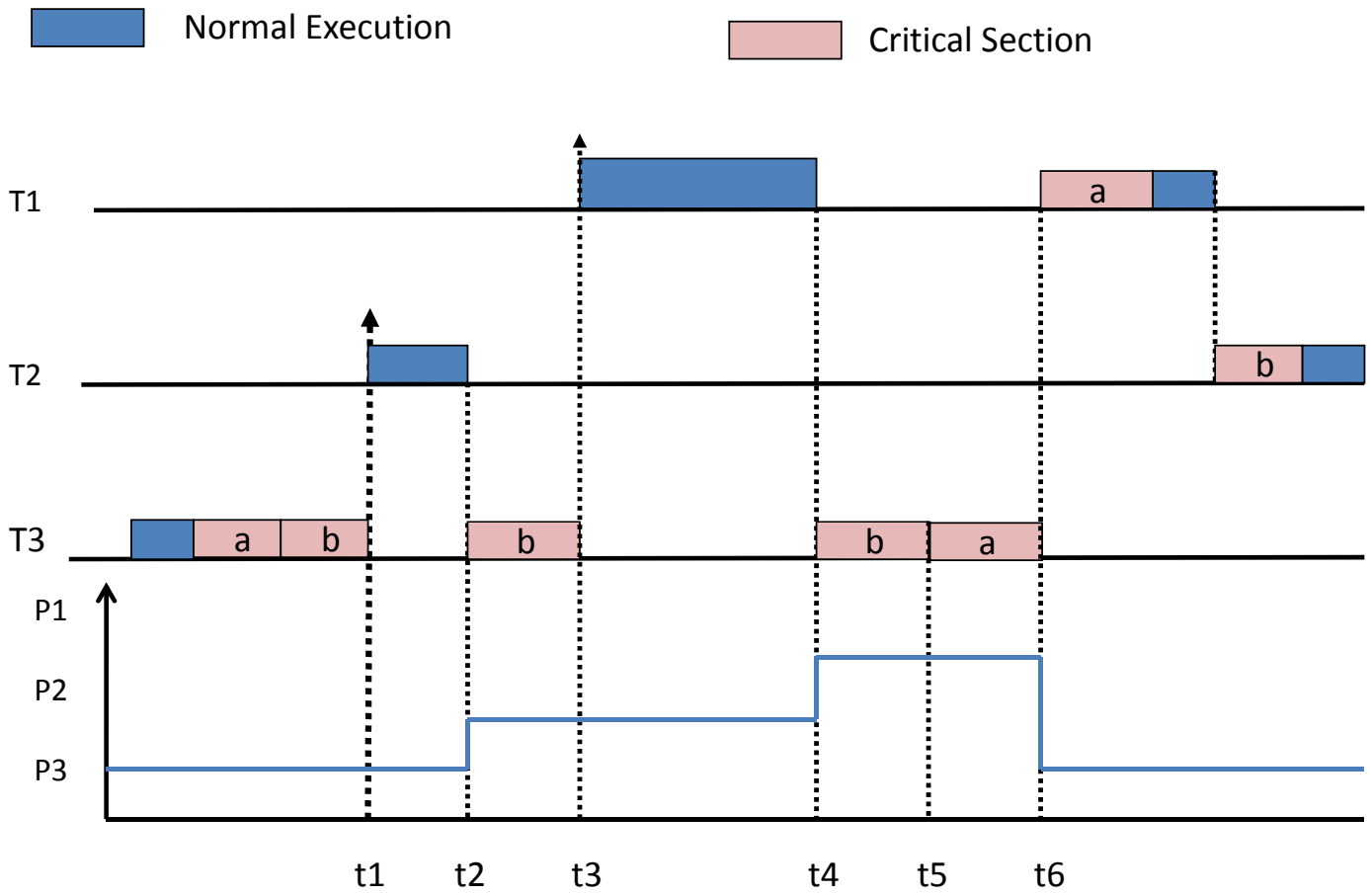
Priority Inheritance Protocol - Example1





Showing the change in the priority level of the lowest priority task J3

Priority Inheritance Protocol - Example2
Nested Critical Sections



Priority Inheritance Protocol

Advantages

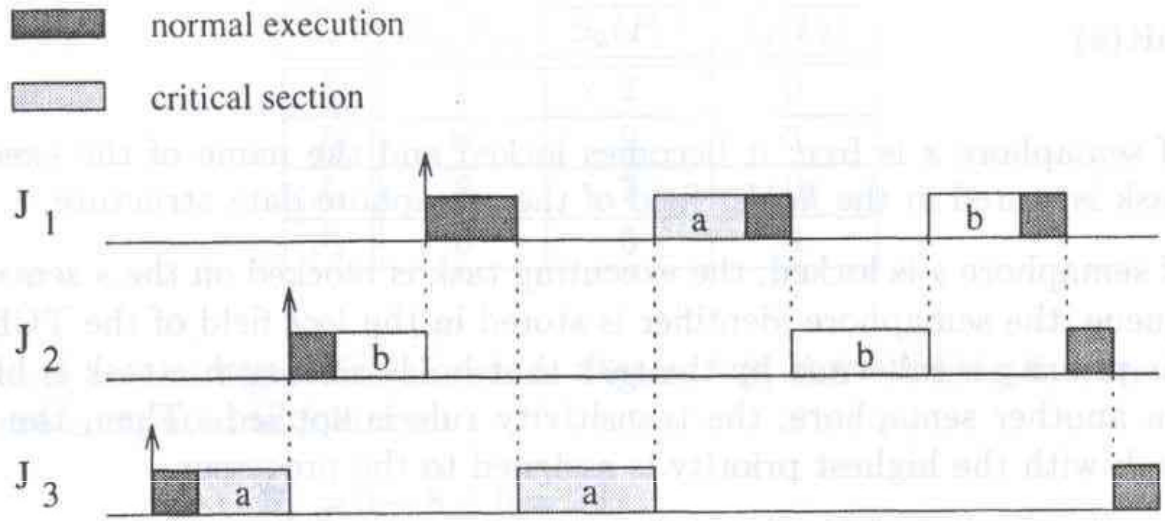
- Under the priority inheritance protocol, a job J can be blocked for at most the duration of min (n, m) critical sections, where n is the number of lower-priority jobs that could block J and m is the number of distinct semaphores that can be used to block J.
- Proof is omitted but it is important to note that the blocking time may be bounded.
- Unlike the naive protocol, the blocking time can never be as long as the WCET of a lower priority task.

Priority Inheritance Protocol

Disadvantages

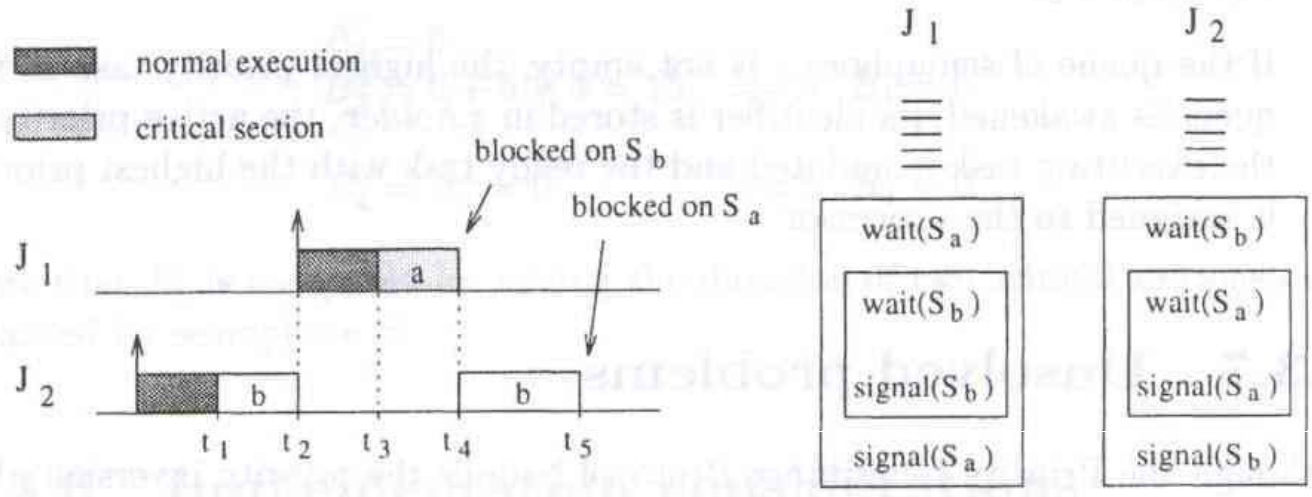
- Chained Blocking: J can get blocked on n critical sections held by n distinct lower priority jobs.
- Deadlocks:

Priority Inheritance Protocol
Chained Blocking



While Priority Inheritance Protocol bounds the priority inversion phenomenon, the blocking duration can still be substantial due to chain blocking. In this figure, when attempting to use its resources, $J1$ is blocked for the duration of two critical sections, once to wait $J3$ to release S_a and then to wait J2 to release S_b. This is called chain blocking.

Priority Inheritance Protocol
Deadlock



The Priority Inheritance Protocol cannot prevent deadlocks.

Priority Ceiling

Extension of Priority Inheritance Protocol to handle chained blocking and deadlocks

Intuition

- Avoid multiple blocking i.e.,
- Once a task enters a critical section, it can not be blocked by lower priority tasks till its completion
- This addresses the evil twins

Priority Ceiling

Extension of Priority Inheritance Protocol to handle chained blocking and deadlocks

Intuition

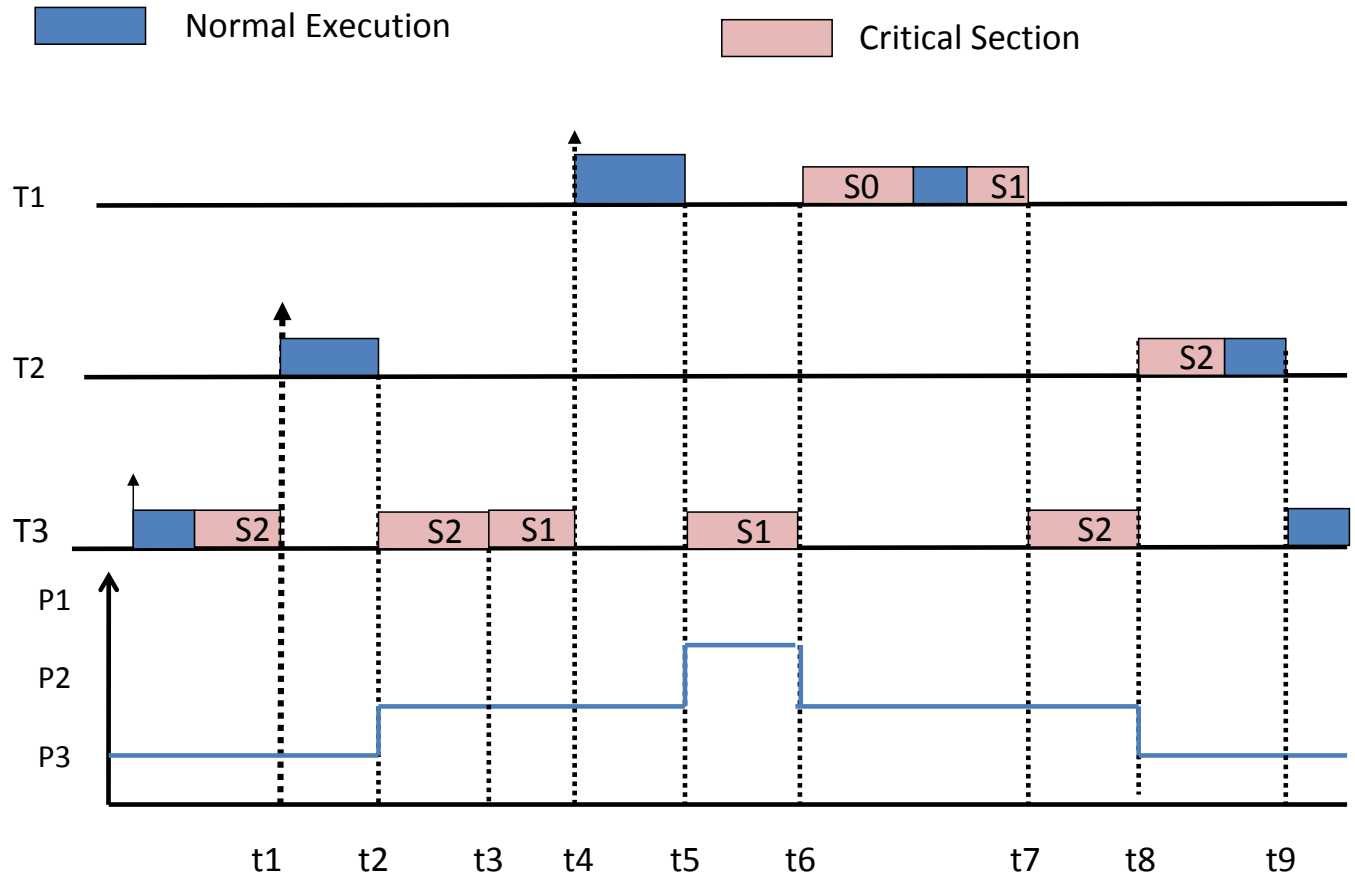
- A task is not allowed to enter a critical section if there are already locked semaphores which could block it eventually
- Hence, once a task enters a critical section, it can not be blocked by lower priority tasks till its completion
- This is achieved by assigning priority ceiling ...

Priority Ceiling

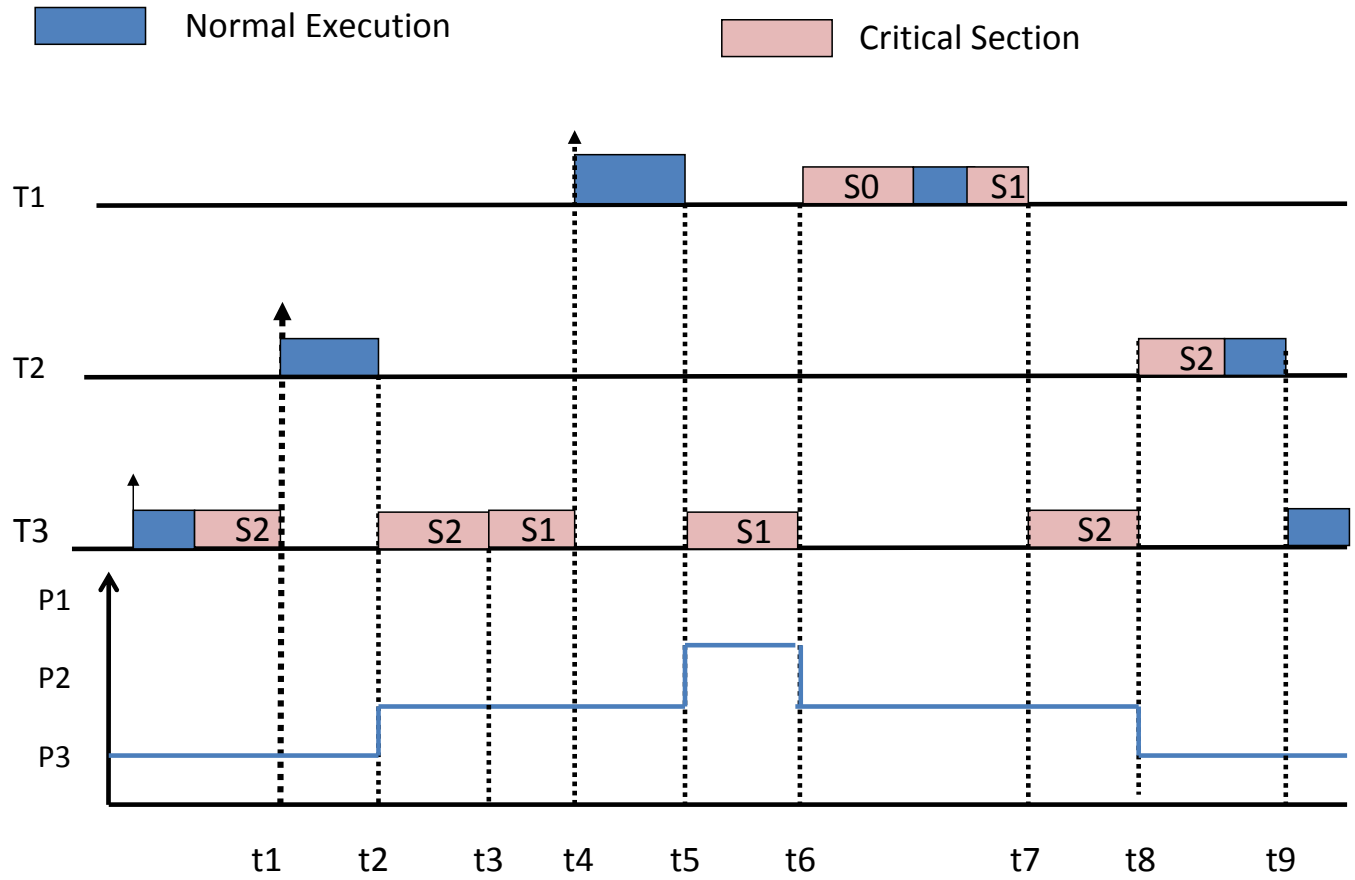
Protocol

- Each semaphore S_k is assigned a priority ceiling $C(S_k)$. It is the priority of the highest priority task that can lock S_k. This is a static value.

Priority Ceiling Protocol Example

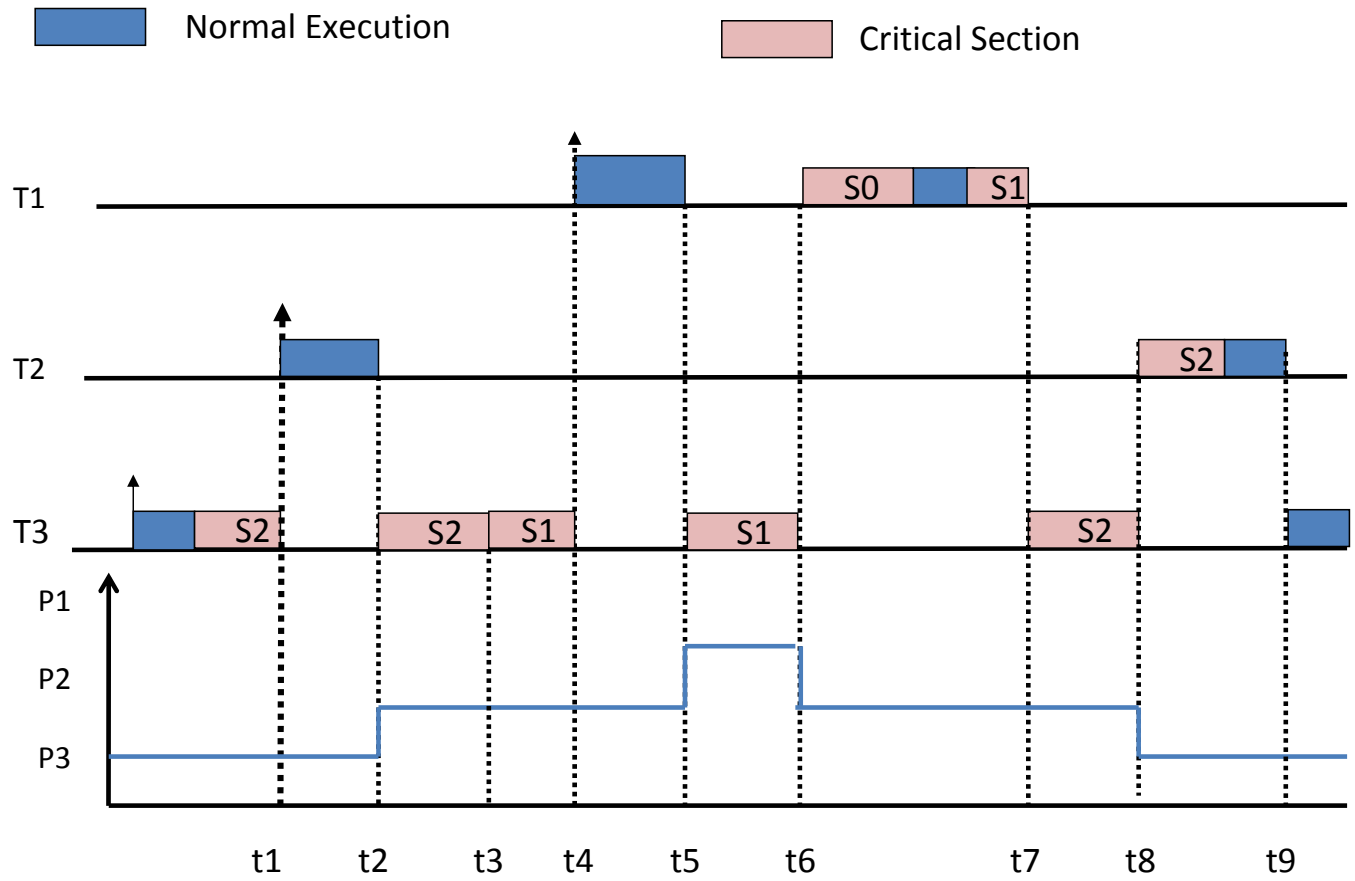


Priority Ceiling Protocol Example



C(S0=?) C(S1=?) C(S2=?)

Priority Ceiling Protocol Example



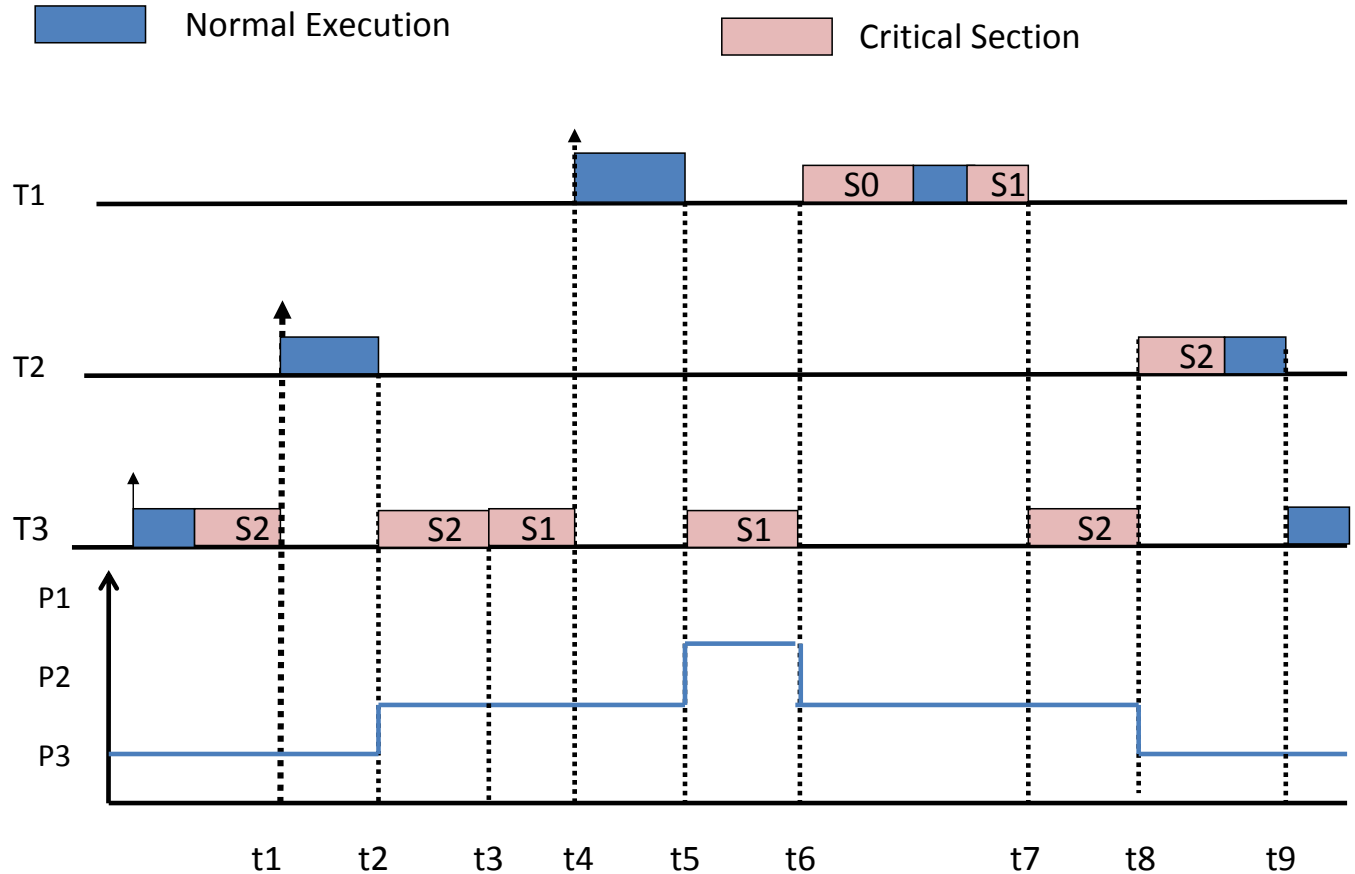
C(S0=P1) C(S1=P1) C(S2=P2)

Priority Ceiling

Protocol

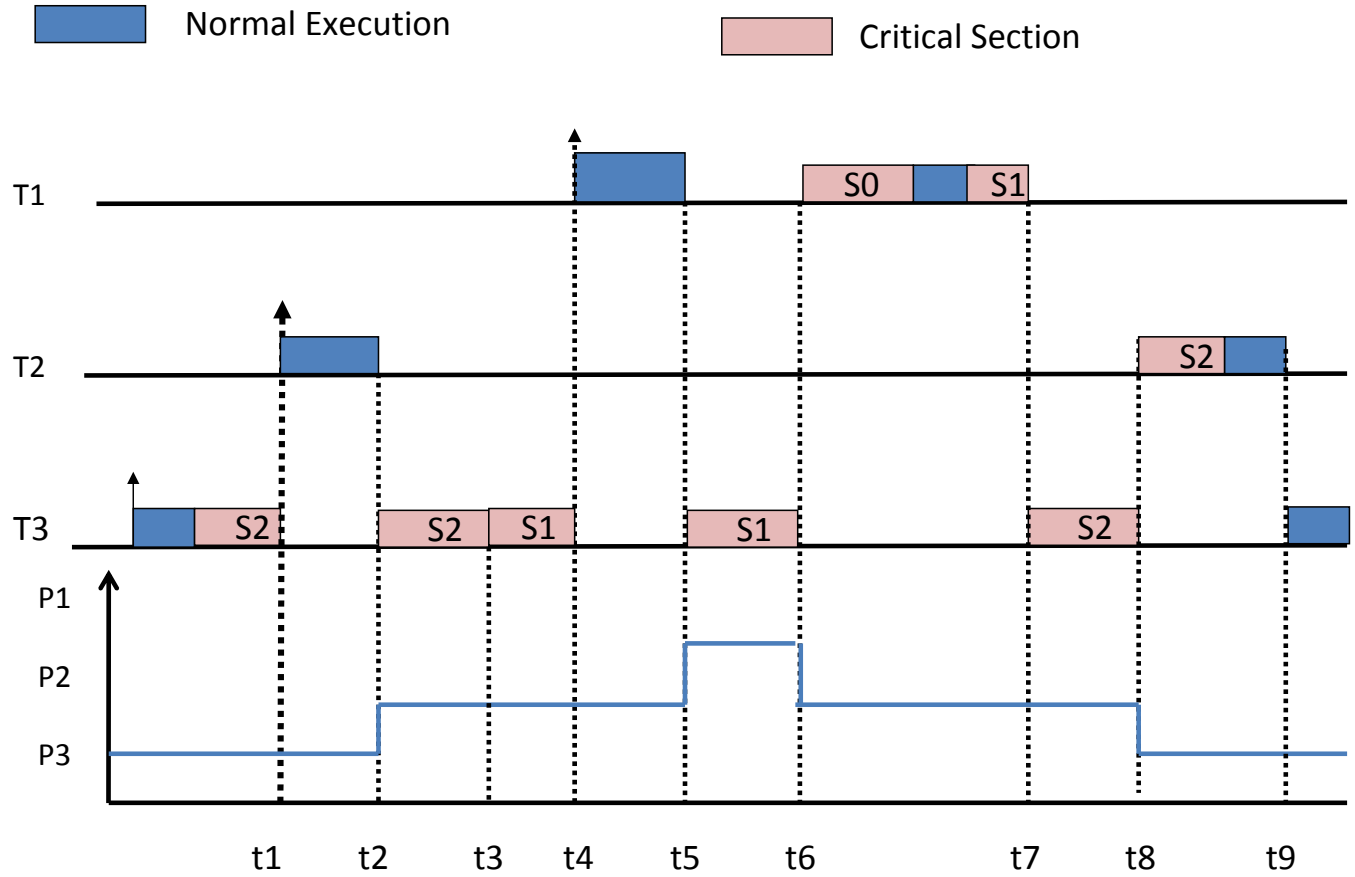
- Suppose J is currently running and it wants to lock the semaphore S_k. J is allowed to lock S_k only if the priority of J is strictly higher than the priority ceiling $C(S^*)$ of the semaphore S^* where:
  - S^* is the semaphore with the highest priority ceiling among all the semaphores which are currently locked by jobs other than J
  - In this case, J is said to blocked by the semaphore S^* (and the job currently holding S^*)
  - When J gets blocked by S^* then the priority of J is transmitted to the job J^* that currently holds S^*

Priority Ceiling Protocol Example



t2: T2 can not lock S2. Currently T3 is holding S2 and C(S2) = P2 and the current priority of T2 is also P2

Priority Ceiling Protocol Example



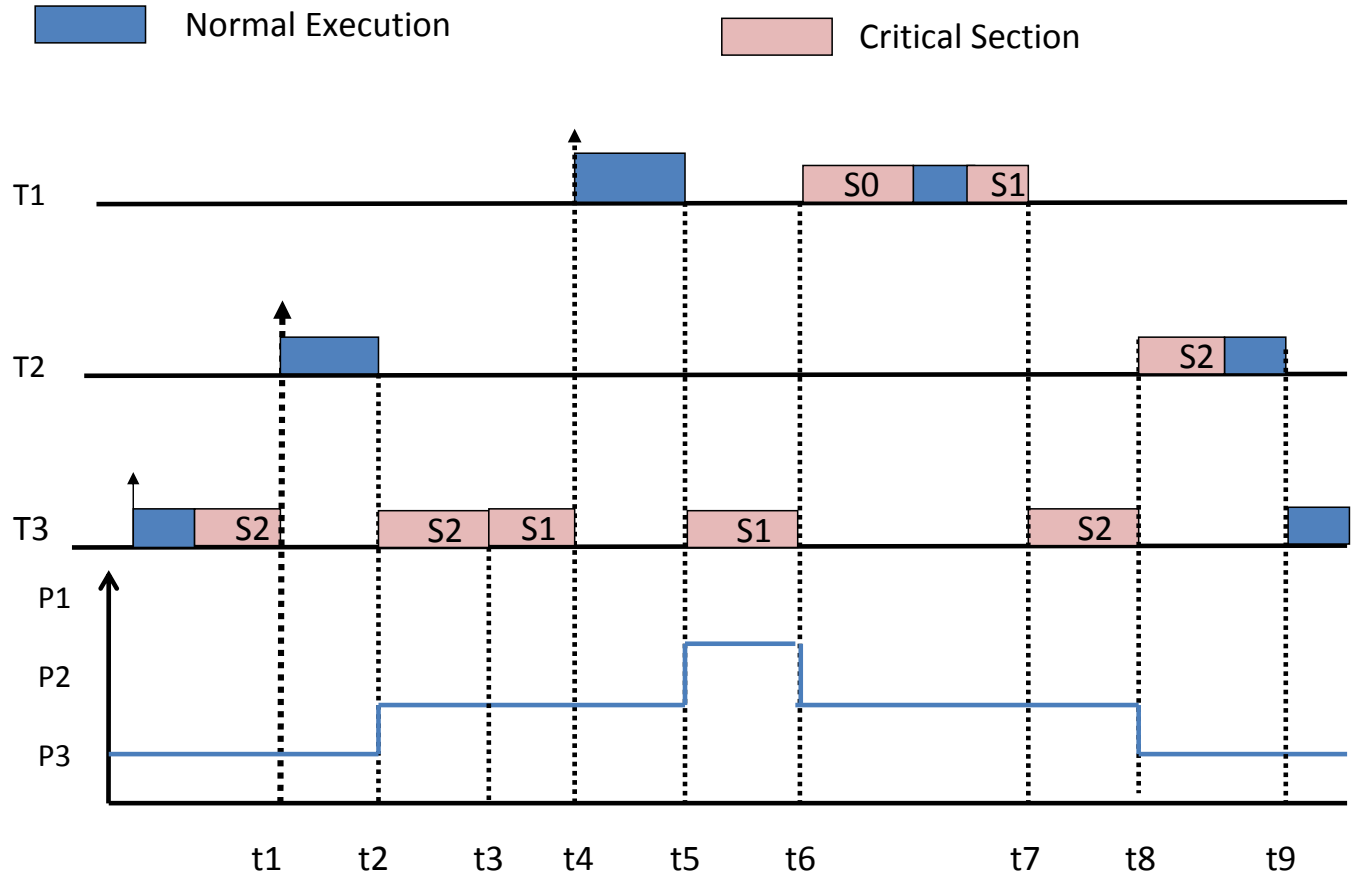
t5 : T1 can not lock S0. Currently T3 is holding S2 and S1 and C(S1) = T1 and the current priority of T1 is also P1. The (inherited) priority of T3 is now P1

Priority Ceiling

Protocol

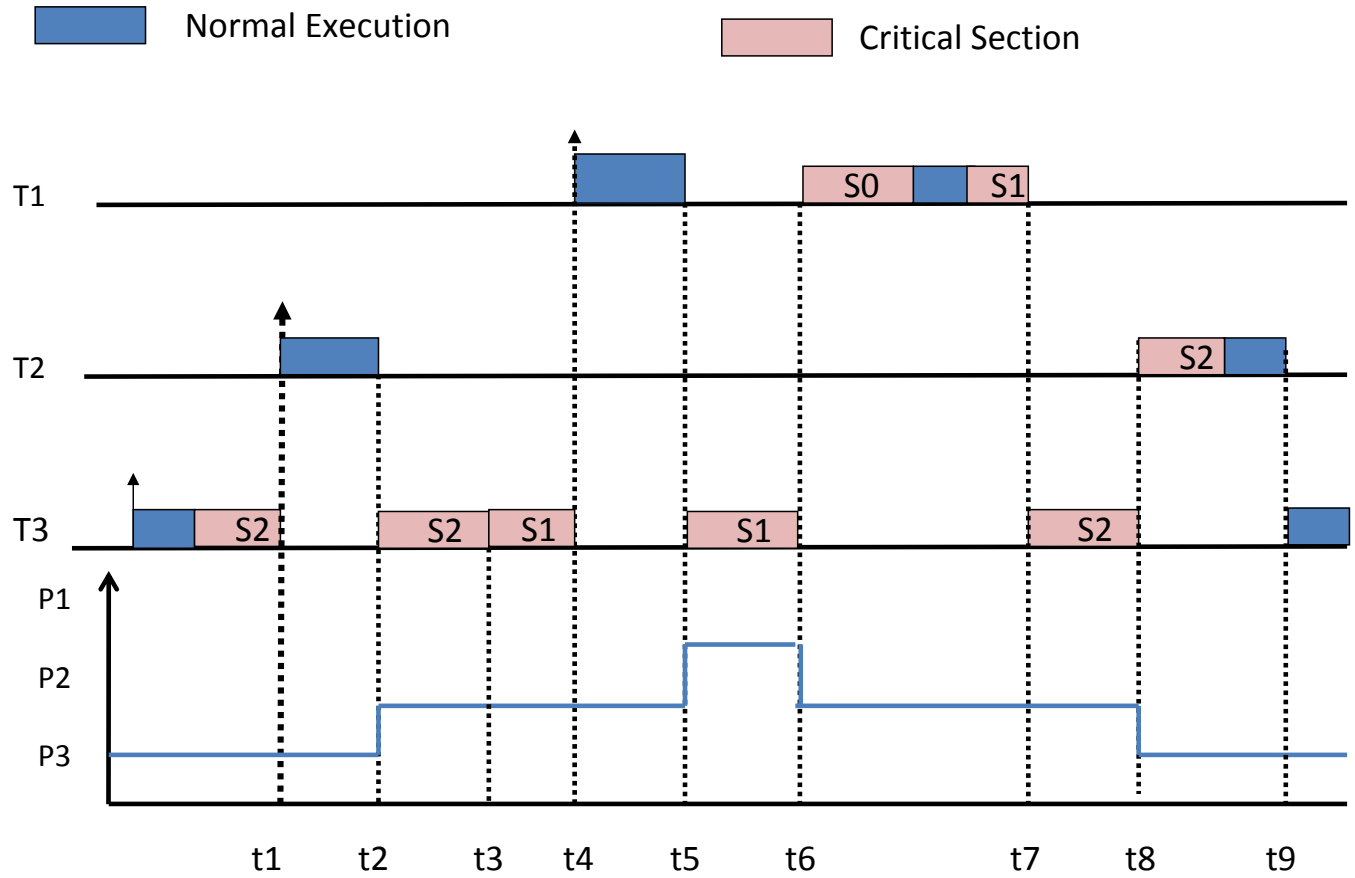
- When J^* leaves a critical section guarded by S^* then it unlocks S^* and the highest priority job, if any, which is blocked by S^* is awakened
- The priority of J^* is set to the highest priority of the job that is blocked by some semaphore that J^* is still holding. If none, the priority of J^* is set to be its nominal one

Priority Ceiling Protocol Example



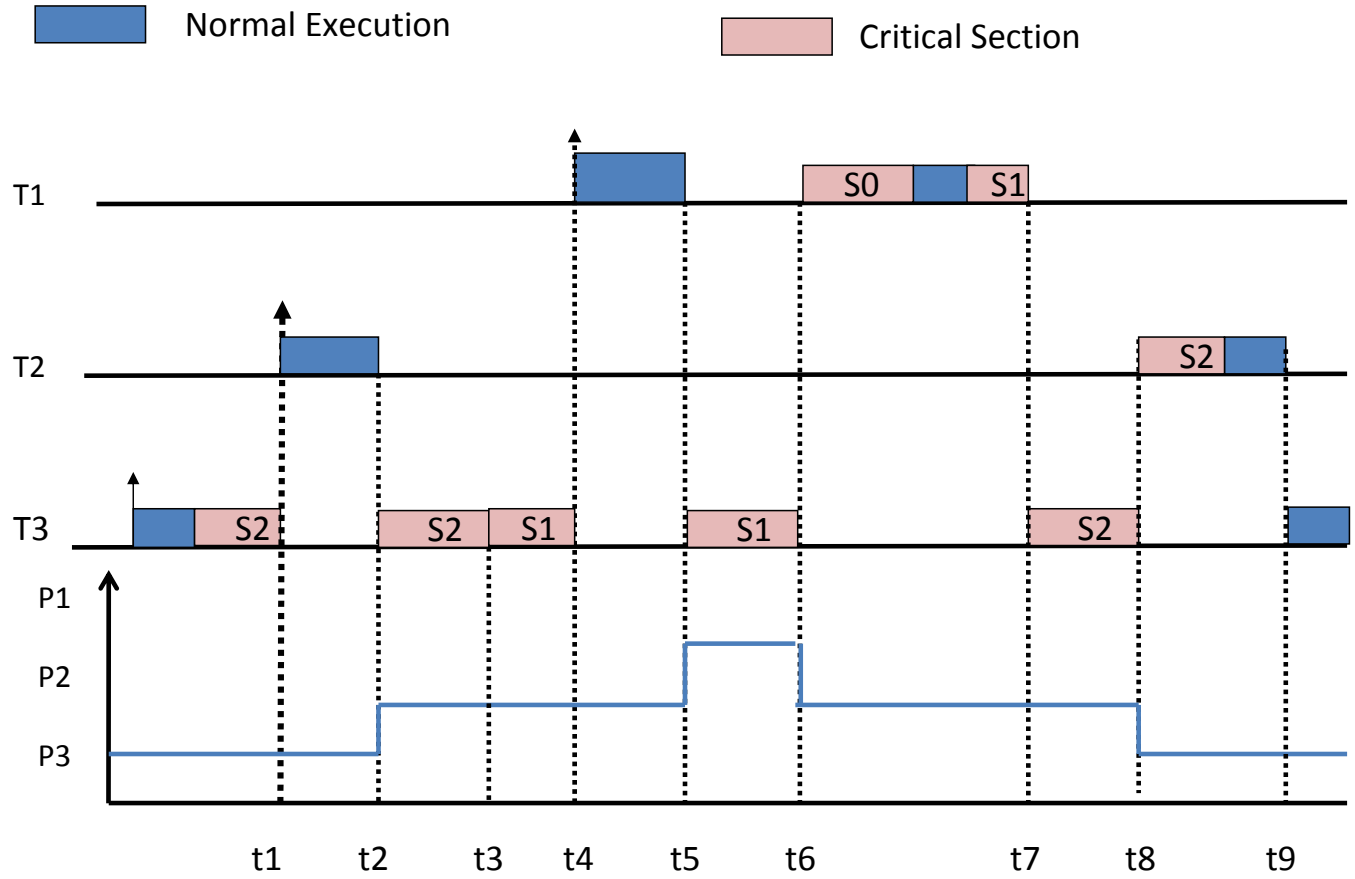
t6 : T3 unlocks S1. It awakens T1. But T3s (inherited) priority is now only P2 while $P1 > C(S2) = P2$. So T1 preempts T3 and runs to completion

Priority Ceiling Protocol
Example



t7 : T3 resumes execution with priority P2

Priority Ceiling Protocol
Example



t8 : T3 unlocks S2 and goes back to its nominal priority P3. So T2 preempts T1 and runs to completion

Schedulability Analysis for PIP and PCP

$\forall i, 1 \leq i \leq n$
$\Sigma_{k=1}^{i} C_k/T_k + B_i/T_i \leq i(2^{1/i} - 1)$
B_i is the maximum blocking times according to the respective protocols.

Resource access protocols

Under static priorities

- Priority inheritance
- Priority ceiling

Under dynamic priorities

- Stack resource

Stack Resource Policy

Extends PCP

- SRP supports dynamic priority scheduling
- PCP blocks a task at the time it makes the resource request, while SRP blocks task at the time it attempts to preempt. We will revisit this issue later, once we go through SRP

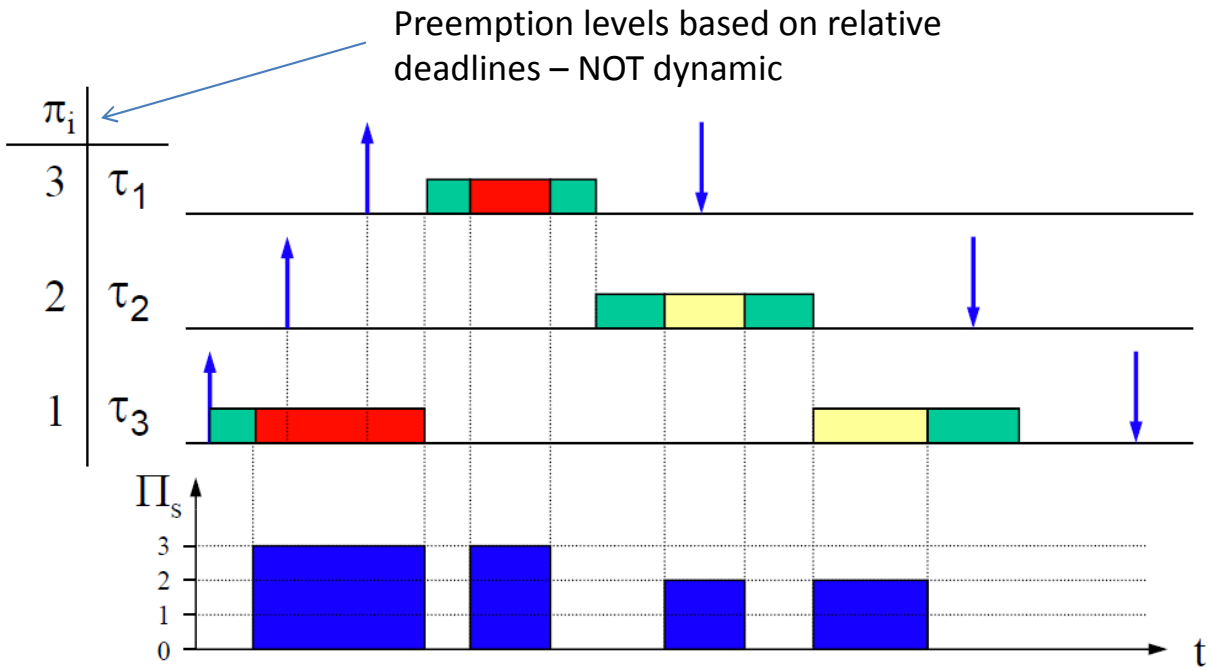
Stack Resource Policy

We need to understand three definitions: Preemption level, Resource ceiling and System Ceiling

Preemption Level

- is a static value
- $\pi \propto 1/D_i$ i.e., tasks with larger deadlines have lower preemption level. Intuition: they can be easily preempted

Stack Resource Policy: Example



Stack Resource Policy

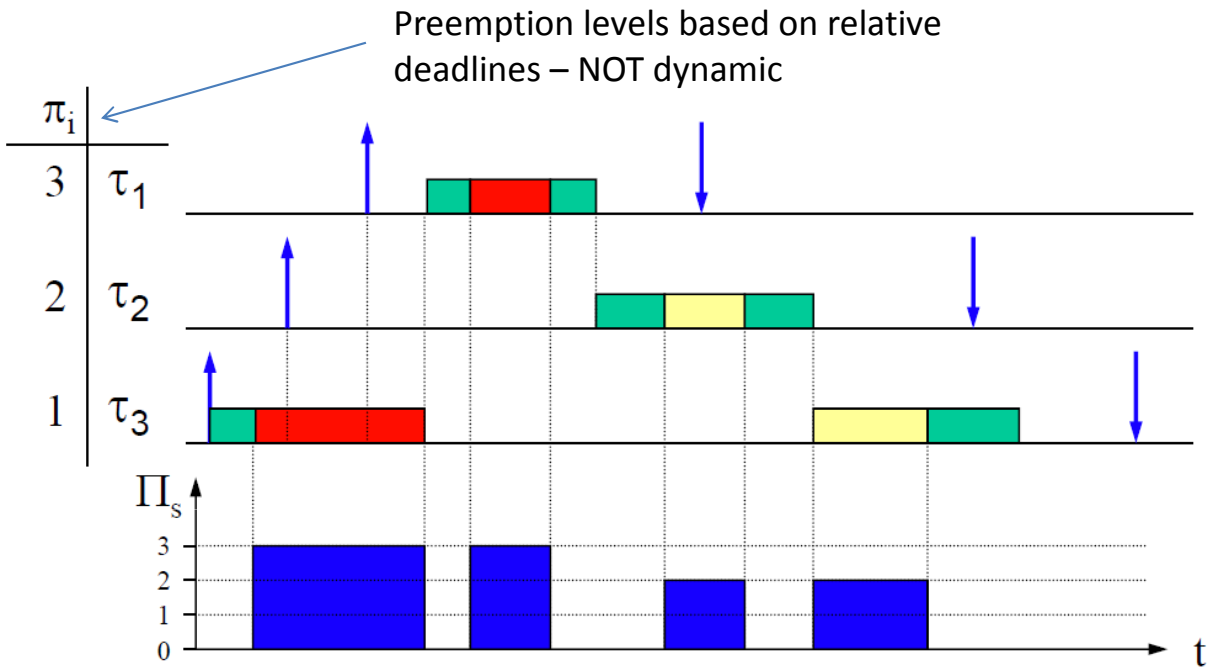
Resource ceiling

of a resource is the highest preemption level from amongst of all tasks that may access that resource. Note: (i) this is associated with each resource (ii) this is static

System ceiling

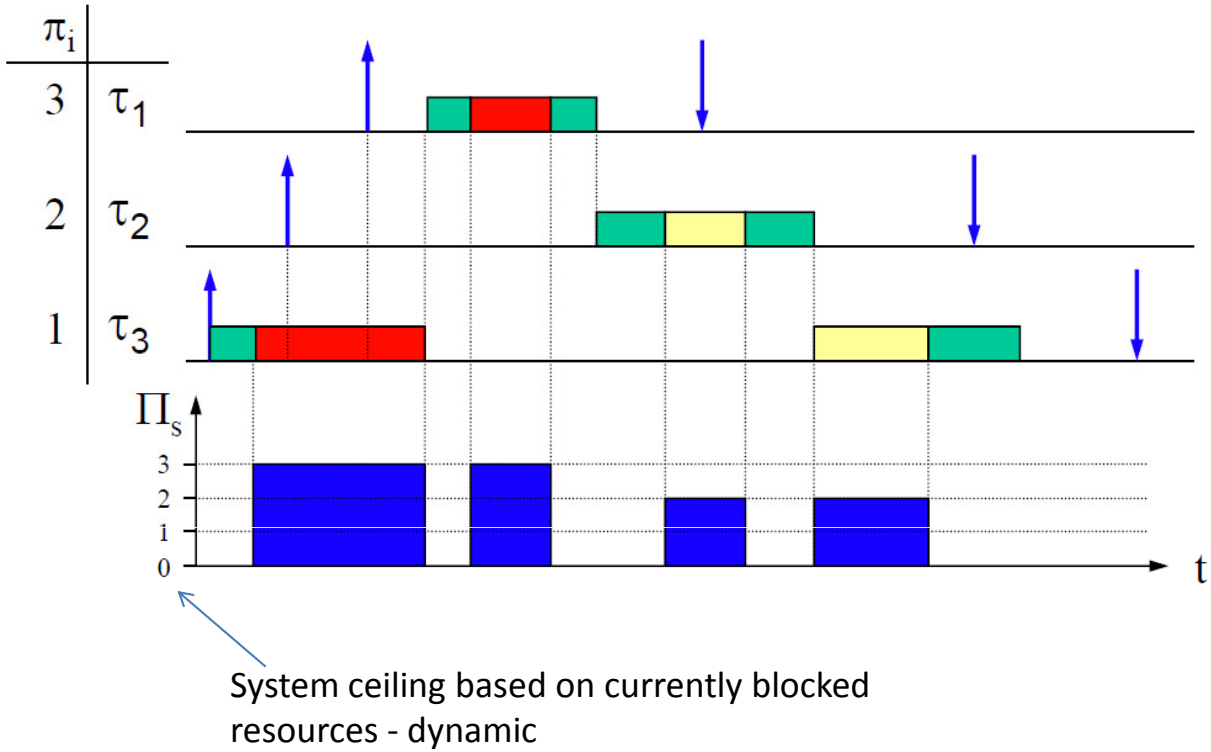
is the highest resource ceiling level from amongst of resources that are currently blocked. Note: (i) this is not associated with each resource but with the system (ii) this is a dynamic parameter that can change every time a resource is accessed or released

Stack Resource Policy: Example



What is the resource ceiling for the red resource? for the yellow resource?

Stack Resource Policy: Example



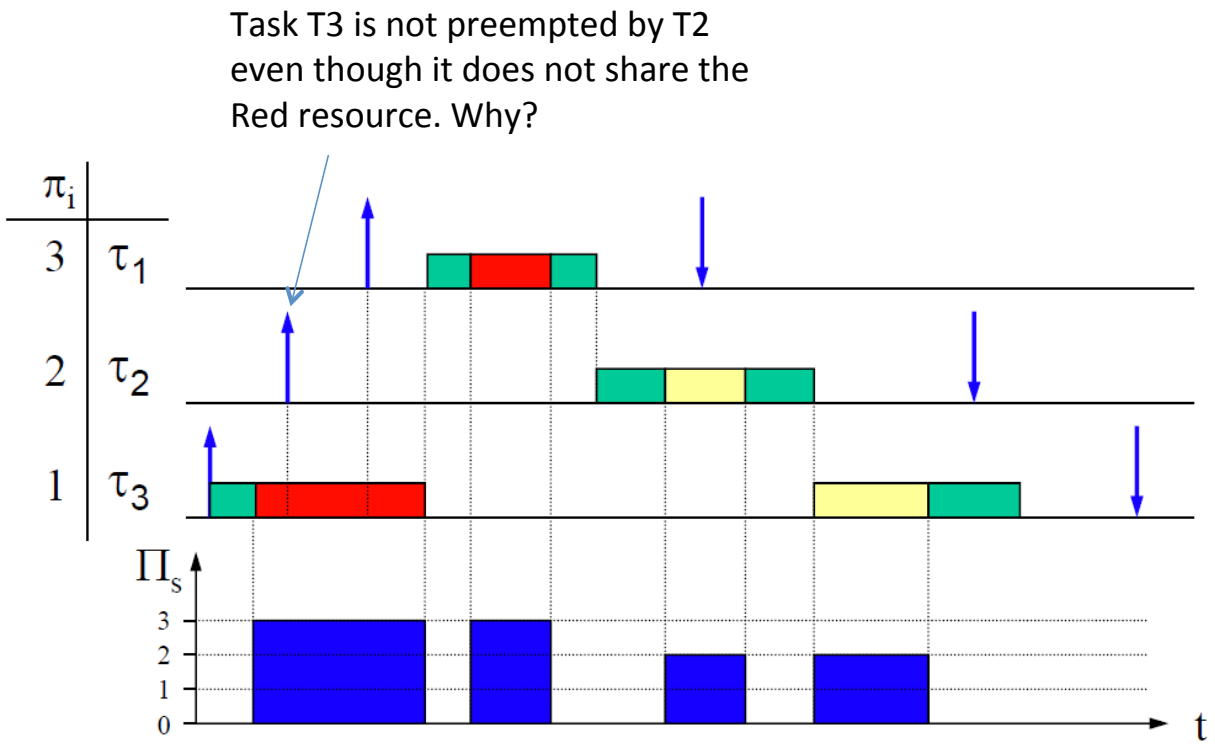
Resource ceiling for the red resource is 3. For the yellow resource, it is 2. Based on this, we can see how the system ceiling varies dynamically

Stack Resource Policy

A task can preempt another task if

- it has the highest priority
- and its preemption level is higher than the system ceiling

Stack Resource Policy: Example



Stack Resource Policy

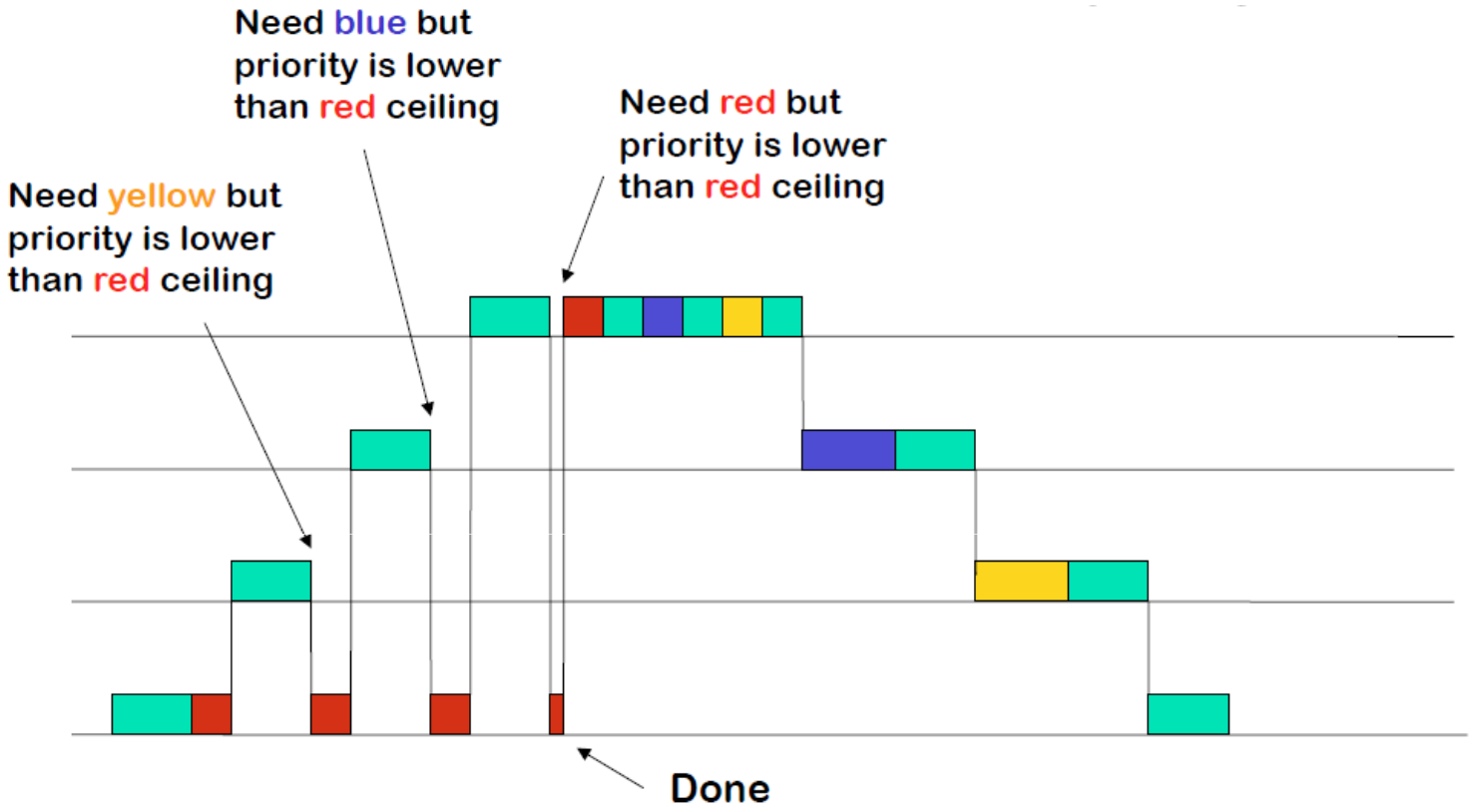
Intuition

- When a job needs a resource that is not available, it is blocked at the time it attempts to preempt, rather than later. (Think - preemption levels!)
- To prevent multiple priority inversions, a job is not allowed to start until the resource currently available are sufficient to meet the maximum requirement of every job that could preempt it. (Think - system ceilings)

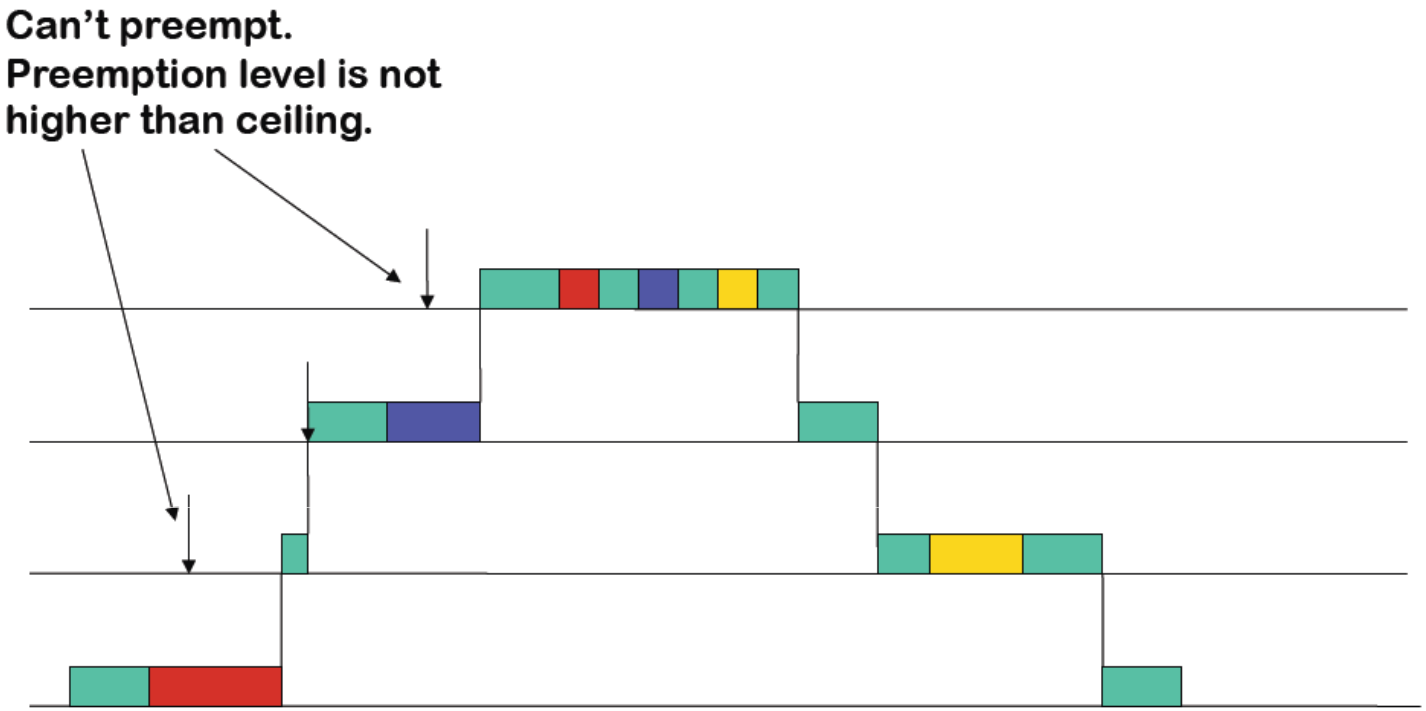
Stack Resource Policy vs. Priority Ceiling

SRP is an extension of the PCP.
Let us look at an example to see the main difference

Stack Resource Policy vs. Priority Ceiling



Stack Resource Policy vs. Priority Ceiling



SRP reduces preemptions compared to PCP.

Stack Resource Policy

Why is it called *Stack* Resource Policy?

Hint: A job cannot be blocked by jobs with lower premption levels - they can resume only when the job completes. Hence, if there are tasks on the same preemption level, they can never occupy stack space on the same time. Higher the number of tasks on the same preemption level, larger the stack space saving!

Schedulability Analysis for SRP

$\forall i, 1 \leq i \leq n$
$\Sigma_{k=1}^{i} C_k/T_k + B_i/T_i \leq 1$
B_i is the maximum blocking time suffered.