Schedulability with resource sharing

Priority inheritance protocol
Priority ceiling protocol
Stack resource policy

Lecture overview

- We have discussed the occurrence of **unbounded priority inversion**
- We know about **blocking** and **blocking times**

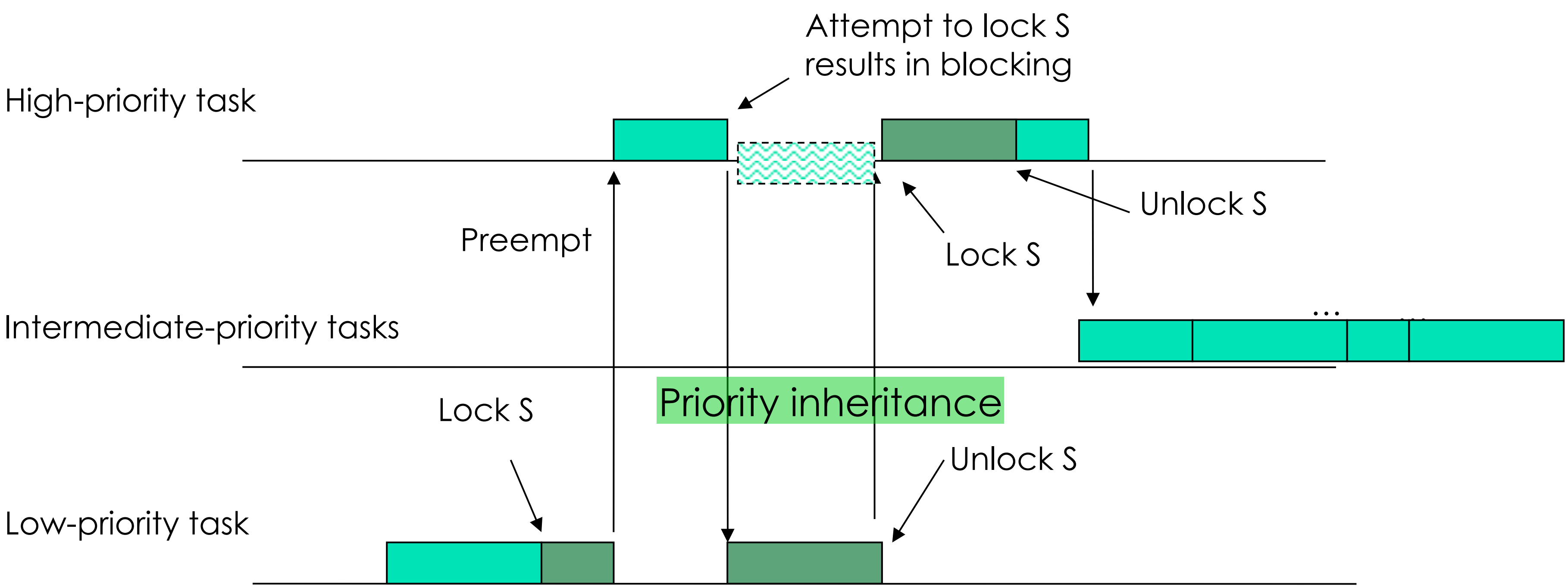
- Now: Evaluating schedulability in combination with protocols for avoiding unbounded priority inversion
- **Priority ceiling protocol** to prevent deadlocks
- **Stack-based resource policy**
  - Improves on other policies
  - Extends to EDF

Blocking

- Tasks have synchronization constraints
  - Use semaphores to protect critical sections
- Blocking can cause a *higher priority task to wait for a lower priority task to unlock a resource*
  - We always assumed that higher priority tasks can preempt lower priority tasks
  - To make rules consistent, we discussed the priority inheritance approach

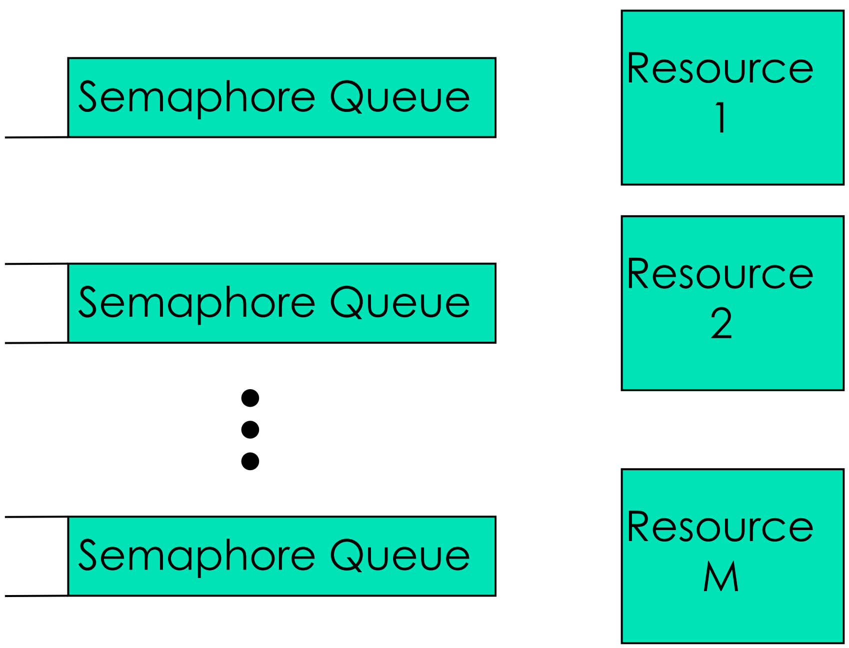
The priority inheritance protocol

- Allow a task to **inherit the priority** of the highest priority task that it is blocking



Maximum blocking time

- If all critical sections are of equal length, B
  - Blocking time = *B x min(N, M)*
  - Why?
  - And what if the critical sections are of differing lengths?



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**)

Maximum blocking time

- If all critical sections are of equal length, B
  - Blocking time = $B \times min(N, M)$
  - Why?
- And what if the critical sections are of differing lengths?
  - Find the maximum length critical section for each resource
  - Add the top $min(N, M)$ sections in size
  - The total priority inversion time experienced by Task T_i is denoted B_i
- **Remember: when computing the blocking time, you need only consider tasks with lower priority.**
  - **And a task may be blocked at most once by a lower priority task.**

Schedulability tests

- For the fixed-priority scheduling case
  - We can use the Liu & Layland bound with some modifications
- For task T_k: we need to consider the blocking by lower priority tasks

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

Each instance of a task may experience blocking (worst case); equivalent to increasing the execution time of the task by the blocking time.

For task T_k, we need to consider:
a) preemption by higher priority tasks
b) blocking from lower priority tasks
c) bound for T_k involves only k tasks

Why do we test each task separately? Why can we not have one utilization bound test like we did earlier?

Example: blocking and schedulability

- Consider the following set of tasks, which share resources R_1, R_2 and R_3
  - Relative deadline are equal to periods; tasks scheduled using RM policy
  - T_1: P_1=20, e_1=3, uses R_1 and R_2 separately for 1 time unit each
  - T_2: P_2=30, e_2=6, uses R_2 and R_3 simultaneously for 2 time units
  - T_3: P_3=50, e_2=10, uses R_1 and R_3 separately for 3 and 4 time units respectively
  - T_4: P_4=80, e_2=8, uses R_2 for 5 time units

Is there a difference?

We will see that there is no difference in this example. In other cases, maybe.

Without resource constraints

$$U = \frac{3}{20} + \frac{6}{30} + \frac{10}{50} + \frac{8}{80} = 0.65 < 0.69$$

The task set satisfies the Liu and Layland bound; easily schedulable by RM

Example: blocking and schedulability

- Consider the following set of tasks, which uses resources R_1, R_2 and R_3
  - Relative deadline are equal to periods; tasks scheduled using RM policy
  - T_1: P_1=20, e_1=3, uses R_1 and R_2 separately for 1 time unit each
  - T_2: P_2=30, e_2=6, uses R_2 and R_3 simultaneously for 2 time units
  - T_3: P_3=50, e_2=10, uses R_1 and R_3 separately for 3 and 4 time units respectively
  - T_4: P_4=80, e_2=8, uses R_2 for 5 time units

$$\frac{B_k}{P_k} + \sum_{i=1}^{k} \frac{e_i}{P_i} \leq k(2^{1/k} - 1)$$

With resource constraints
T_1 can potentially be blocked by T_2, T_3 and T_4

It can be blocked by T_2 on resource R_2 for upto 2 time units
It can be blocked by T_3 on resource R_1 for upto 4 time units
It can be blocked by T_4 on resource R_2 for upto 5 time units

The worst-case wait for R_1 is 3 units (only T_3 can block T_1)
The worst-case wait for R_2 is 5 units (T_2 can block T_1 for 2 units or T_4 can block T_1 for 5 units)
Maximum wait for resources is B_1 = 3+5 = 8

$$\frac{8}{20} + \frac{3}{20} < 1$$

T_1 is schedulable

Example: blocking and schedulability

- Consider the following set of tasks, which uses resources R_1, R_2 and R_3
  - Relative deadline are equal to periods; tasks scheduled using RM policy
  - T_1: P_1=20, e_1=3, uses R_1 and R_2 separately for 1 time unit each
  - T_2: P_2=30, e_2=6, uses R_2 and R_3 simultaneously for 2 time units
  - T_3: P_3=50, e_2=10, uses R_1 and R_3 separately for 3 and 4 time units respectively
  - T_4: P_4=80, e_2=8, uses R_2 for 5 time units

$$\frac{B_k}{P_k} + \sum_{i=1}^{k} \frac{e_i}{P_i} \leq k(2^{1/k} - 1)$$

With resource constraints
***T*₂** can be blocked by T_3 and T_4
T_3 can block T_2 in two ways:
directly on R_3 (upto 4 units)
by obtaining priority of T_1 when using R_1 (upto 3 units) (push-through)
T_4 can block T_2 in two ways:
directly when using R_2 (upto 5 units)
by obtaining priority of T_1 when using R_2 (upto 5 units) (push-through)
The worst-case blocking by T_3 is 4 time units
The worst-case blocking by T_4 is 5 time units
Maximum wait for resources is B_2 = 5+4 = 9

A low priority task can block a high priority task at most once. With priority inheritance, it will get a higher priority and continue till it releases the lock. Therefore, it can block a high priority task at most once.

$$\frac{9}{30} + \left(\frac{3}{20} + \frac{6}{30}\right) = 0.65 < 0.82$$

T_2 is schedulable

Example: blocking and schedulability

- Consider the following set of tasks, which uses resources R_1, R_2 and R_3
  - Relative deadline are equal to periods; tasks scheduled using RM policy
  - T_1: P_1=20, e_1=3, uses R_1 and R_2 separately for 1 time unit each
  - T_2: P_2=30, e_2=6, uses R_2 and R_3 simultaneously for 2 time units
  - T_3: P_3=50, e_2=10, uses R_1 and R_3 separately for 3 and 4 time units respectively
  - T_4: P_4=80, e_2=8, uses R_2 for 5 time units

$$\frac{B_k}{P_k} + \sum_{i=1}^{k} \frac{e_i}{P_i} \leq k(2^{1/k} - 1)$$

With resource constraints
T_3 can be blocked by T_4
even when it shares no resource with T_4 (lower priority task)
Notice that T_4 may execute with priority of T_1 (priority inheritance)
T_4 may execute with the priority of T_1 for at most 5 time units
Classic case of push-through blocking
Maximum blocking due to T_4 is 5 time units; B_3 = 5

$$\frac{5}{50} + \left(\frac{3}{20} + \frac{6}{30} + \frac{10}{50}\right) = 0.65$$

T_3 is schedulable

Example: blocking and schedulability

- Consider the following set of tasks, which uses resources R_1, R_2 and R_3
  - Relative deadline are equal to periods; tasks scheduled using RM policy
  - T_1: P_1=20, e_1=3, uses R_1 and R_2 separately for 1 time unit each
  - T_2: P_2=30, e_2=6, uses R_2 and R_3 simultaneously for 2 time units
  - T_3: P_3=50, e_2=10, uses R_1 and R_3 separately for 3 and 4 time units respectively
  - T_4: P_4=80, e_2=8, uses R_2 for 5 time units

$$\frac{B_k}{P_k} + \sum_{i=1}^{k} \frac{e_i}{P_i} \leq k(2^{1/k} - 1)$$

With resource constraints
T_4 can never be blocked
because it is the lowest priority task
Maximum wait for resources is B_4 = 0

$$\left(\frac{3}{20} + \frac{6}{30} + \frac{10}{50} + \frac{8}{80}\right) = 0.65$$

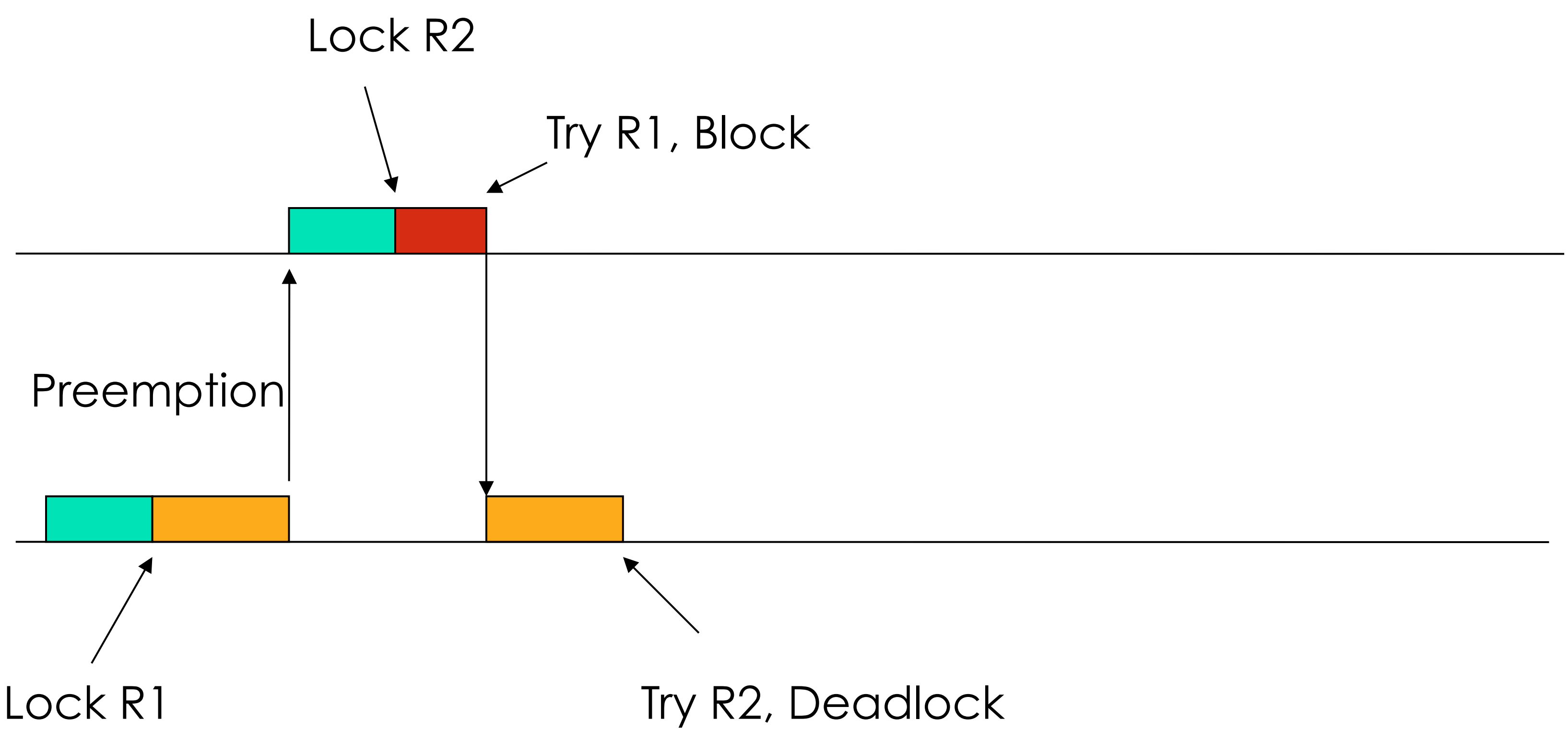
T_4 is schedulable

General approach to computing blocking times

- For a high-priority task
  - Examine all tasks with lower priority
    - Determine the worst-case blocking that it may offer (consider the highest priority that it can inherit)
  - Examine all semaphores/resources
    - Determine the worst-case blocking due to that resource
    - Consider lower-priority tasks that may inherit a higher priority when they hold the semaphore

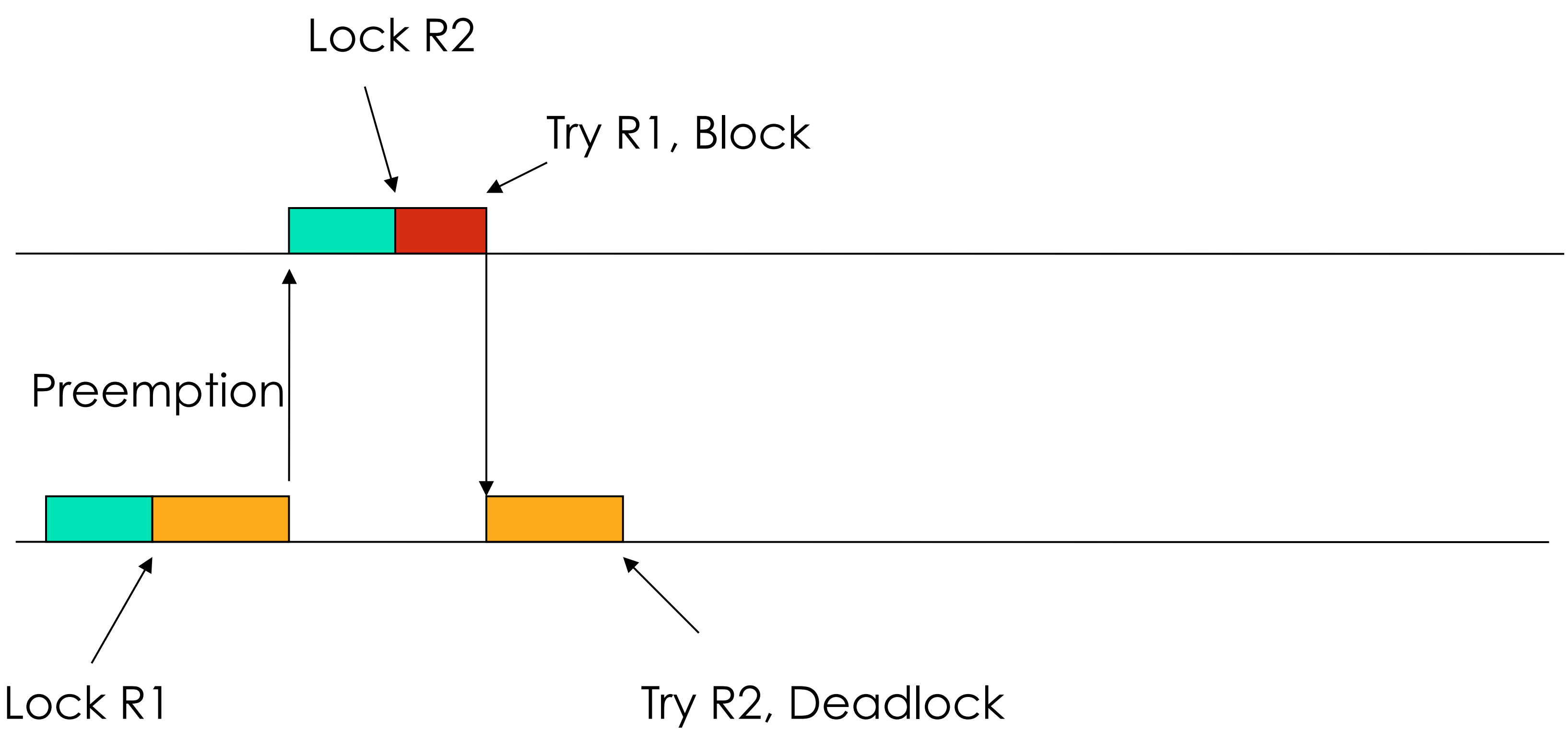
Does priority inheritance solve all problems?

- Actually, not all problems
- We can still have a deadlock if resources are locked in opposing orders
- As we saw two lectures back



Deadlocks

- Can attribute it to sloppy programming
- But can we solve the problem in a different way
- Avoid deadlocks by designing a suitable protocol

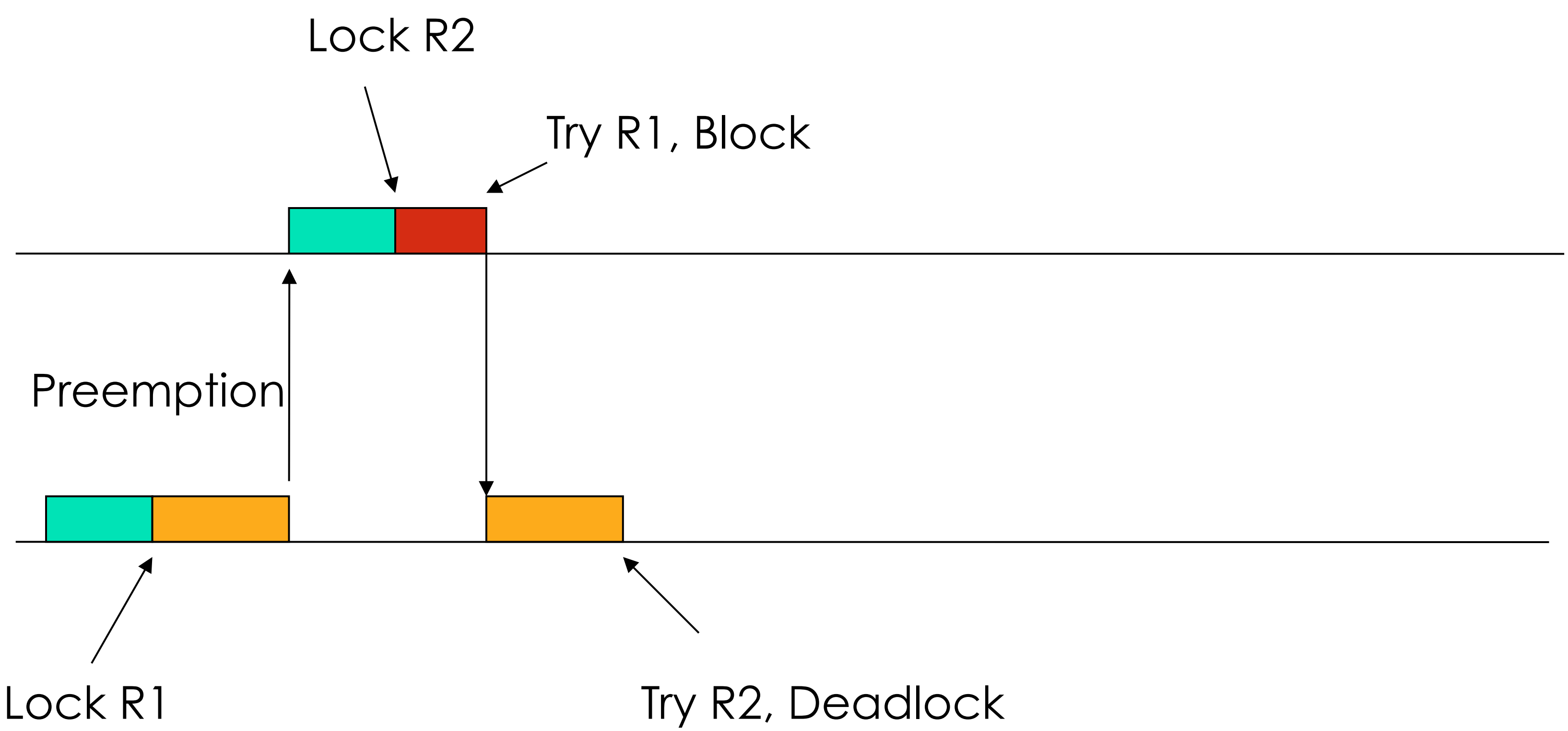


Priority ceiling protocol

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

Deadlocks?

- A deadlock can occur if two tasks locked semaphores in opposite order. Can it occur with the priority ceiling protocol?

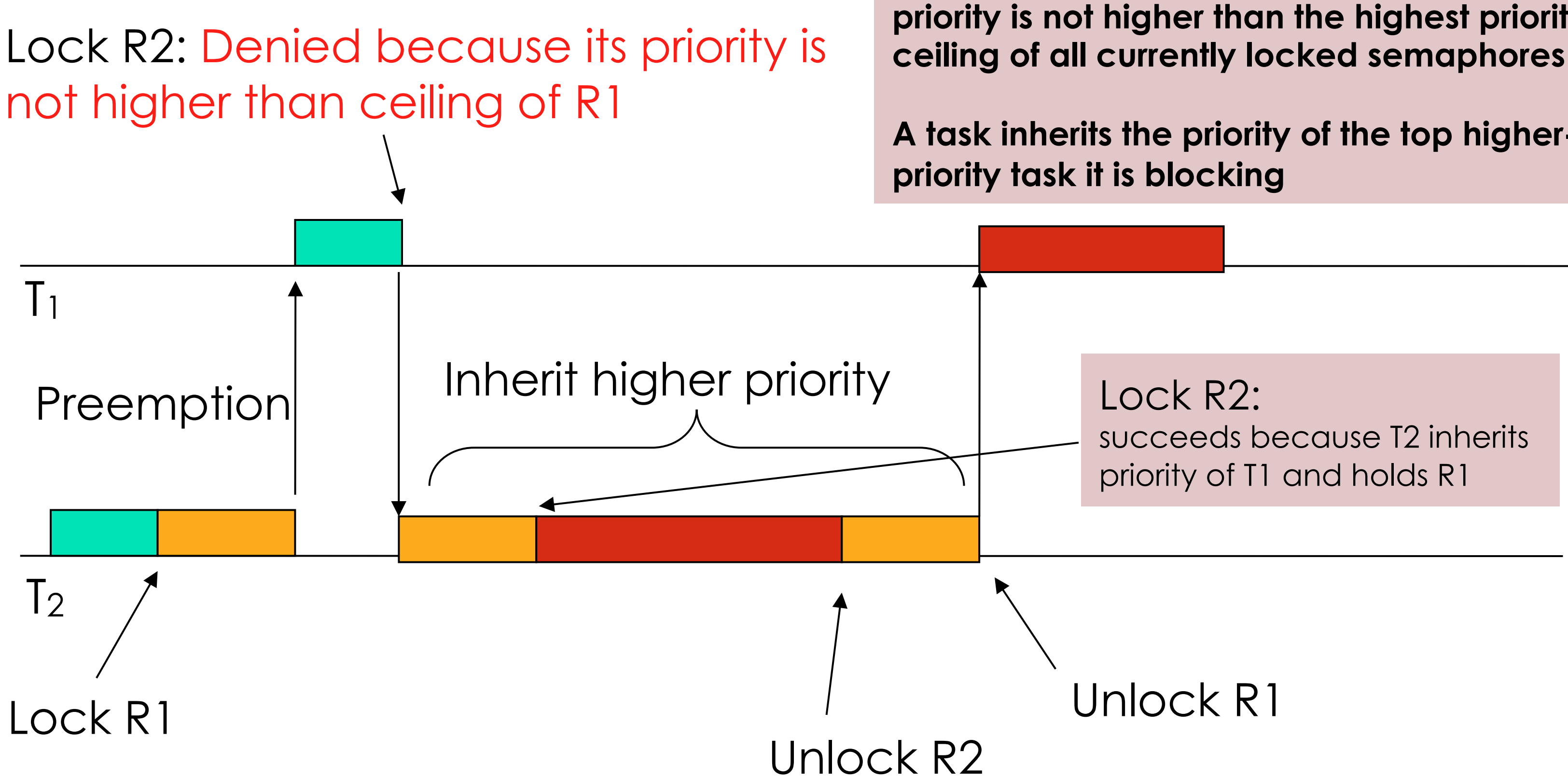


Priority ceilings

• T_1 and T_2 use R_1 and R_2: the priority ceiling of a resource is the priority of the highest priority task that uses it, therefore the priority ceilings of R_1 and R_2 are the same: the priority of T_1

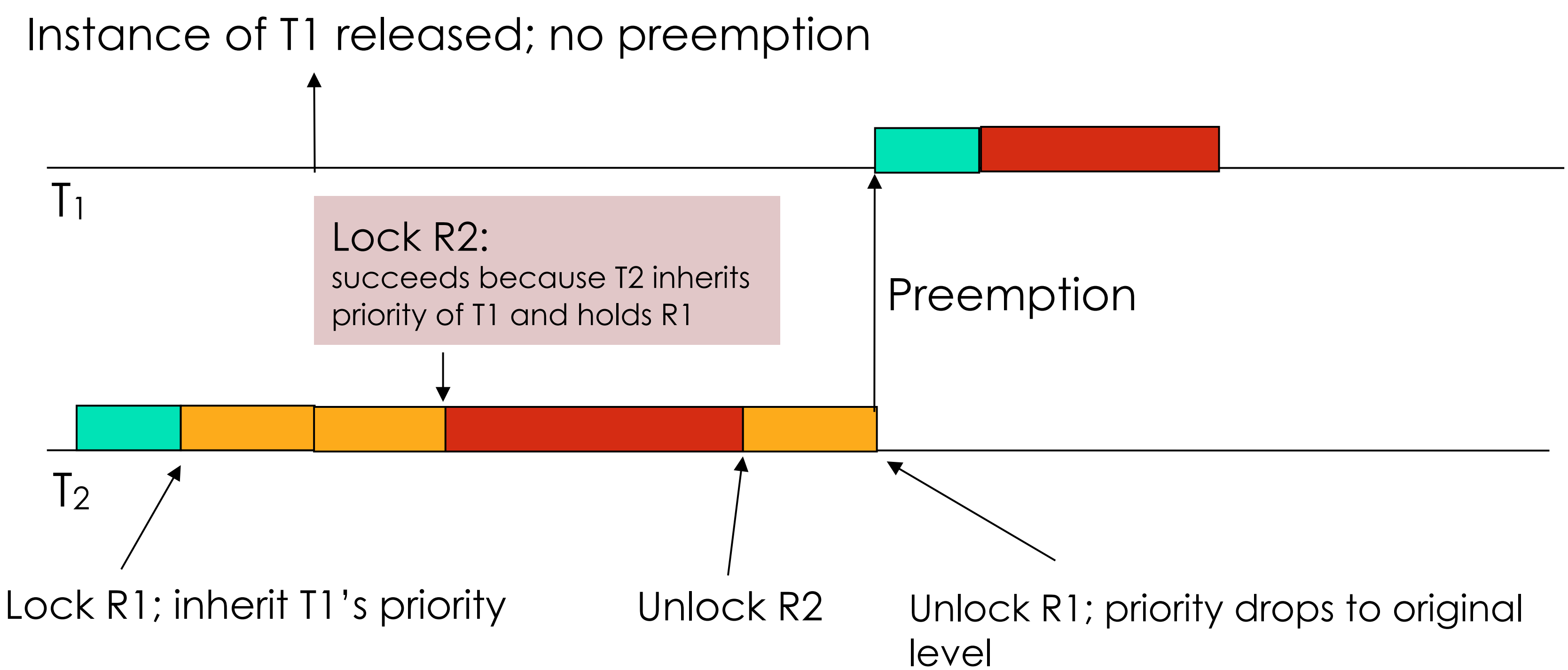
A task that requests lock R_k is denied if its priority is not higher than the highest priority ceiling of all currently locked semaphores

A task inherits the priority of the top higher-priority task it is blocking



Immediate inheritance

- Priority ceiling protocol with slight difference: when a semaphore is locked, the locking task raises its priority to the ceiling priority of the semaphore (**immediate inheritance**). When the semaphore is unlocked the task's priority is restored.



Schedulability test for priority ceiling protocol

- The test is the same as with the priority inheritance protocol
  - Worst-case blocking time may change when compared to PIP

$$\frac{B_k}{P_k} + \sum_{i=1}^{k} \frac{e_i}{P_i} \leq k(2^{1/k} - 1)$$

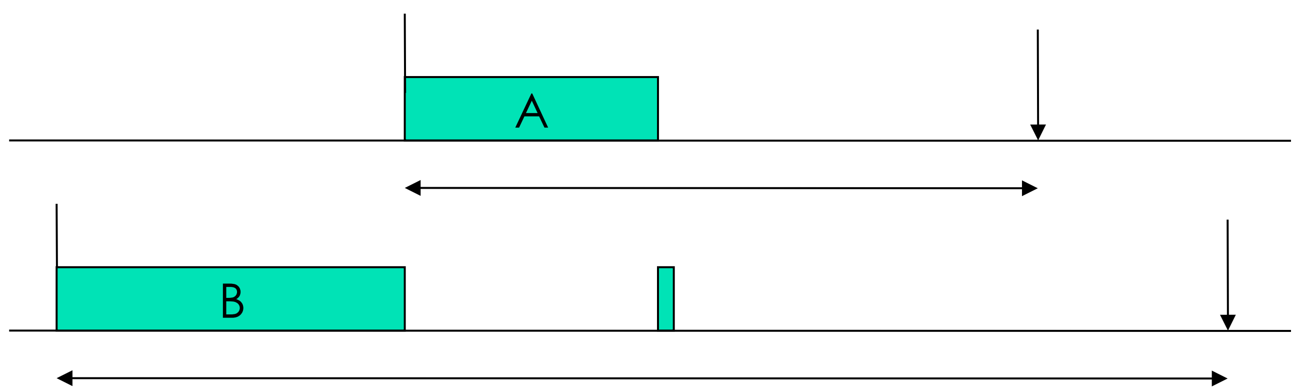
For task T_k

Stack-based resource policy

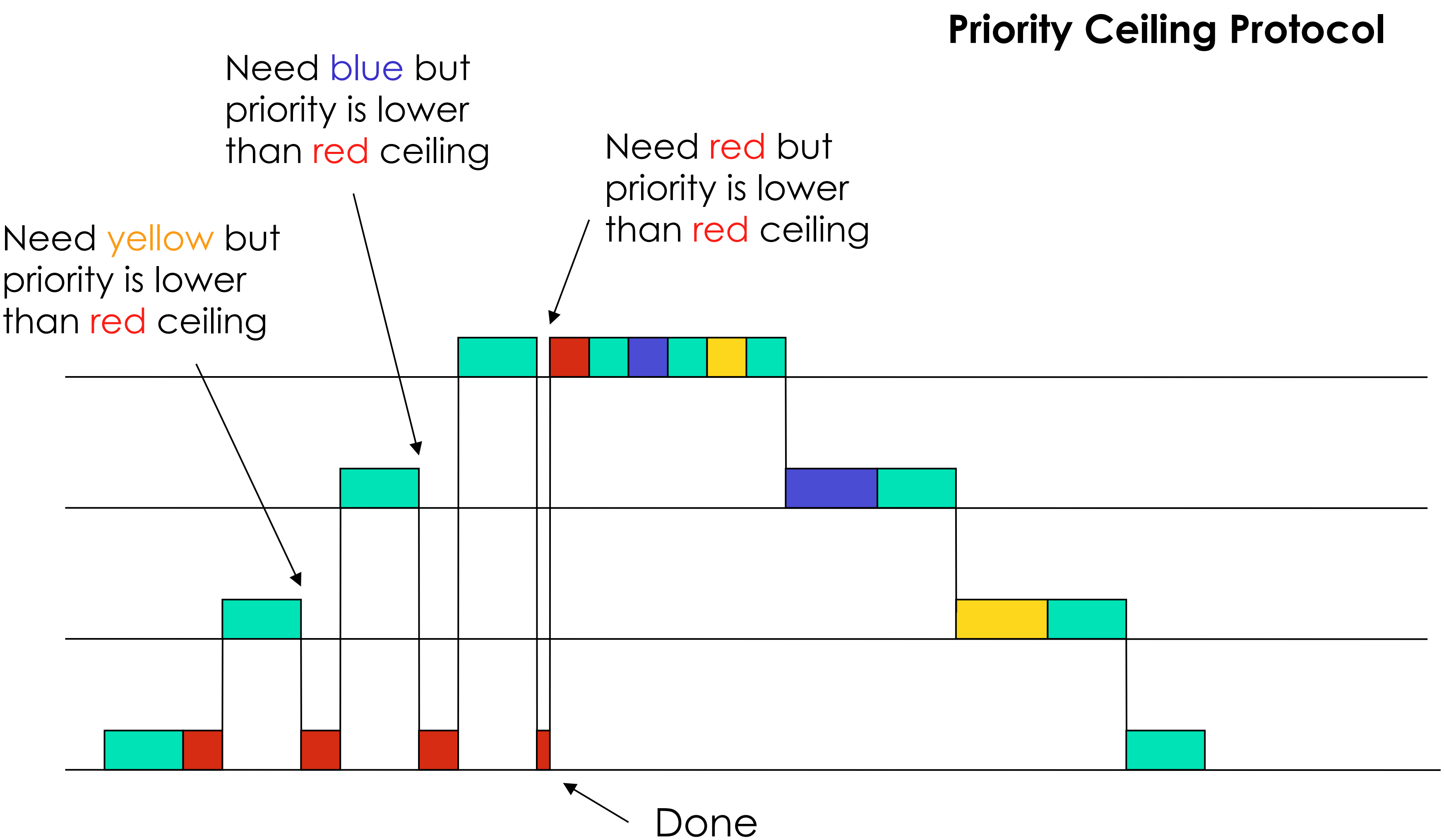
- Priority inheritance protocol and priority ceiling protocol are easy to analyze in a fixed-priority setting
- What about dynamic priority scheduling?
- **Stack-based resource policy [SRP]**
  - Preemption level: Any fixed value that satisfies the statement "if A arrives after B and priority(A) > priority(B), then PreemptionLevel(A) > PreemptionLevel(B)."
  - Resource ceiling for resource *R*: Highest preemption level of all tasks that may access the resource *R*
  - System ceiling: Highest resource ceiling among all currently locked resources
  - A task can preempt another task if
    - it has the highest priority and
    - its preemption level is higher than the system ceiling

Stack-based resource policy with EDF

- Priority is inversely proportional to the absolute deadline
- Preemption level is inversely proportional to the relative deadline
- Observe that:
  - If A arrives after B and Priority(A) > Priority(B) then PreemptionLevel(A) > PreemptionLevel(B)



Priority ceiling vs. stack-based resource policy

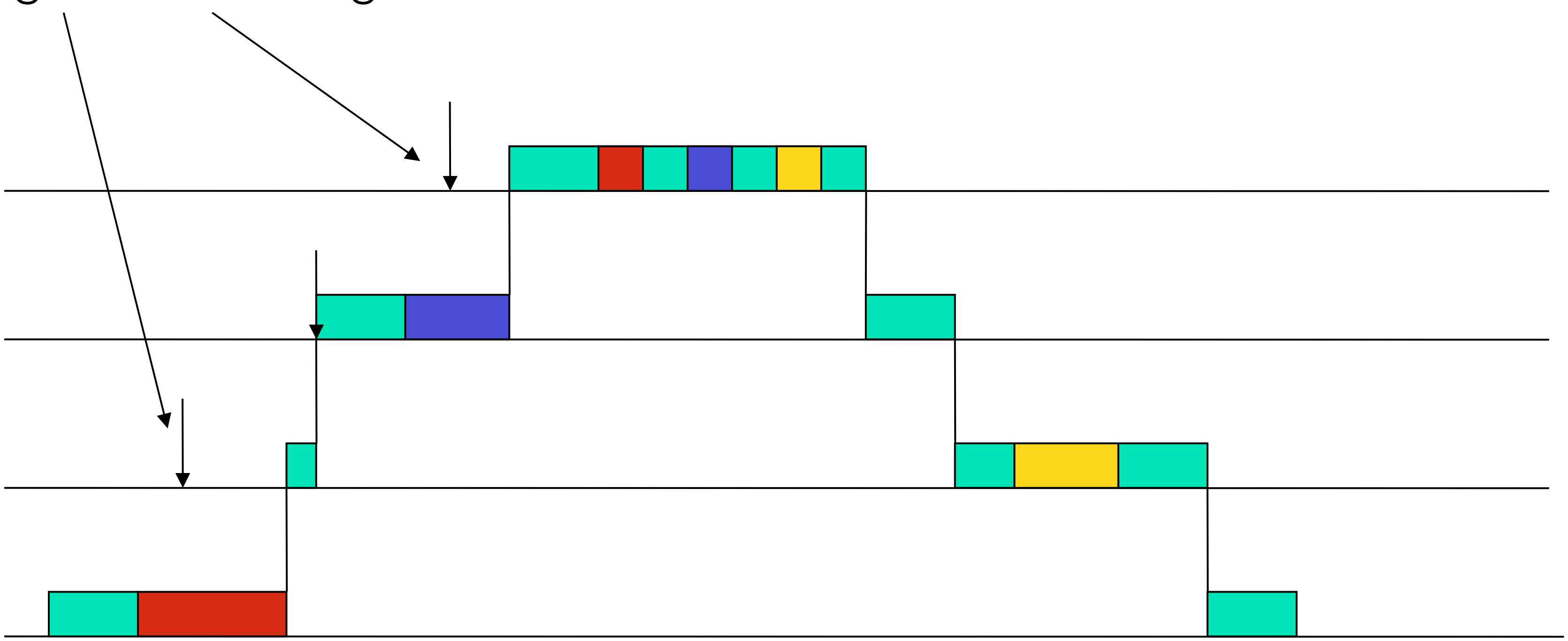


Priority ceiling vs. stack-based resource policy

Stack-based Resource Policy

Notice that SRP is similar to immediate inheritance in PCP.
However, with no static priority levels, it needs a preemption level.

Can't preempt.
Preemption level is not
higher than ceiling.



Analysis with EDF and SRP

- As simple as other protocols

$$\frac{\boxed{B_k}}{P_k} + \sum_{i=1}^{k} \frac{e_k}{P_k} \leq 1$$

Maximum blocking due to task with lower preemption level; in the case of EDF: with period P_j such that $P_k < P_j$.

For task T_k

Tasks are sorted such that the task with shortest period is T_1 and so on.

Highlights

- Schedulability analysis needs to account for blocking due to low priority tasks
- Priority inheritance protocol (PIP) may not prevent deadlocks
- Deadlocks can be prevented with the priority ceiling protocol (PCP)
- To deal with dynamic priority policies (such as EDF), we need a different policy: the stack-based resource policy (SRP)
- SRP (and the immediate inheritance version of the PCP) have efficient implementations
  - Reduce the number of context switches
  - SRP also prevents deadlocks (note the similarities between PCP and SRP)