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**

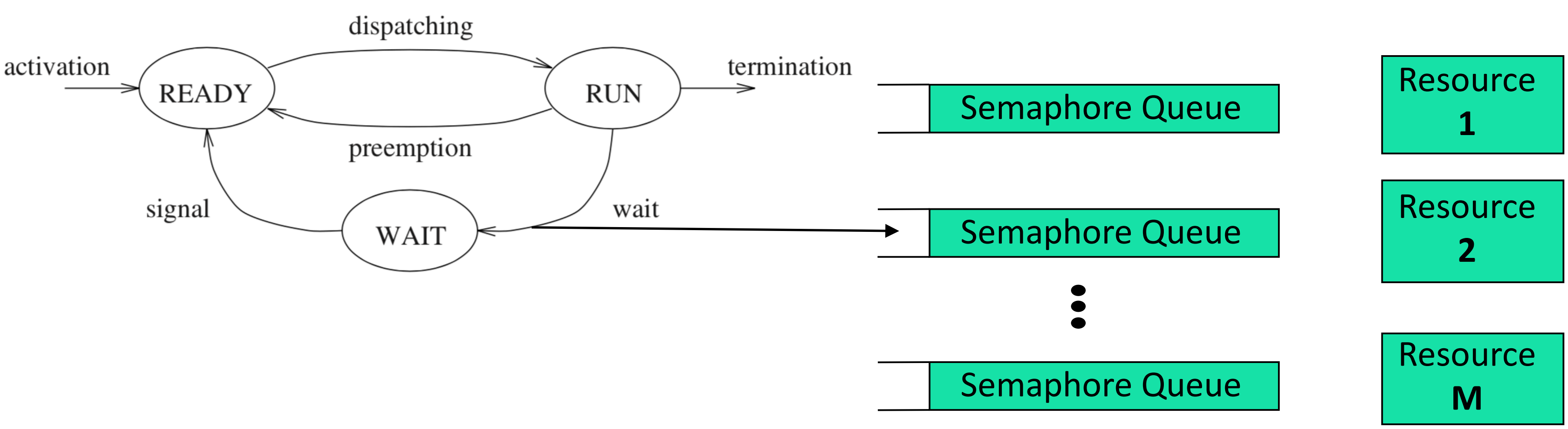
- Now: Evaluating schedulability in combination with protocols for avoiding unbounded priority inversion
- **Priority ceiling protocol** to prevent deadlocks
- **Stack-based resource policy**
    - Improves on other policies
    - Extends to EDF

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

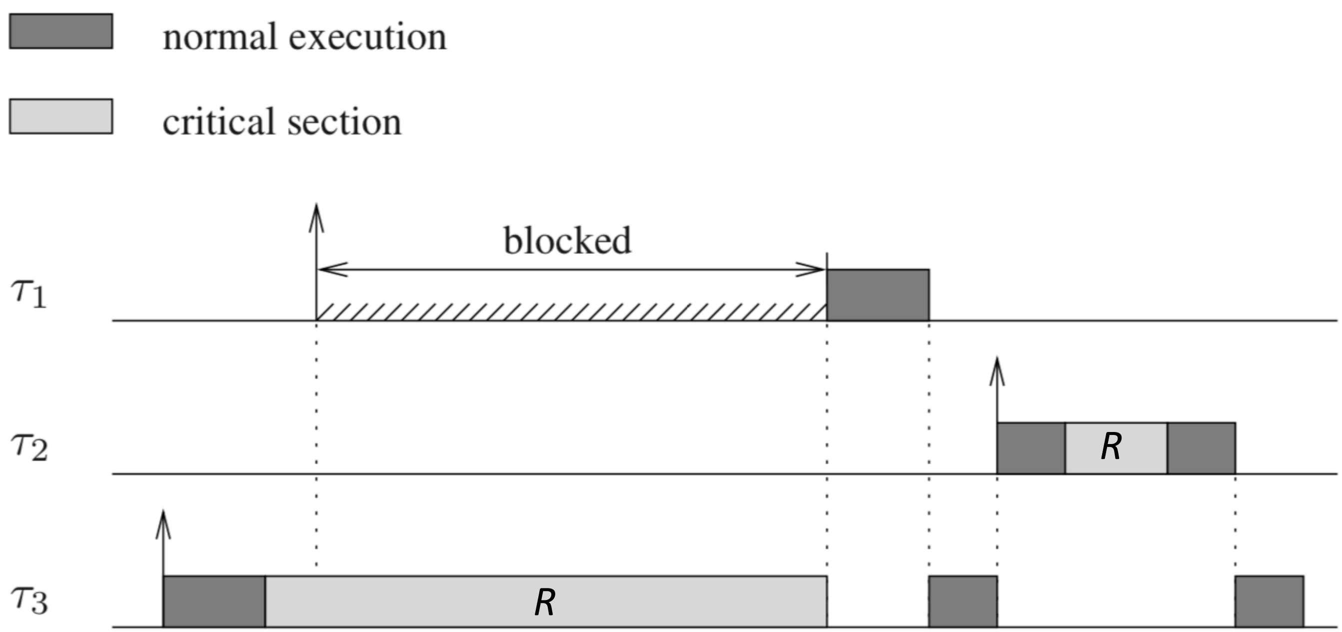
General Model and Assumptions

- **Assumption:** Each resource has one instance only (binary semaphores)
- **Assumption:** Resource requests are *properly nested*
- **Assumption:** *We have perfect knowledge of all task resource requirements*
- *Except for SRP, all protocols are designed for **static-priority** scheduling*
- Each resource has a semaphore queue



Approach #1: Non-Preemptive Protocol (NPP)

- Whenever a task requests a resource, make it the highest priority task *for the duration of its critical section*
- **The good:** Easy to Implement
- **The bad:** unnecessarily blocks higher priority tasks that do not request resources



Non-Preemptive Protocol: Blocking time computation

- A task T_i can be blocked only by a ***lower priority*** task that has requested a resource before T_i's arrival
- **Key:** *Whenever a task is in a critical section, it cannot be preempted*
- The lower priority task resumes its static priority as soon as it releases all resources in the critical section
- As soon as the lower priority task releases the resource, the highest priority task available will acquire the processor and it will not be blocked again
- **Conclusion:** Worst case blocking time is the duration of the longest critical section of any lower priority task

$\delta_{j,k}$: duration of *longest* critical section of task j using resource R_k

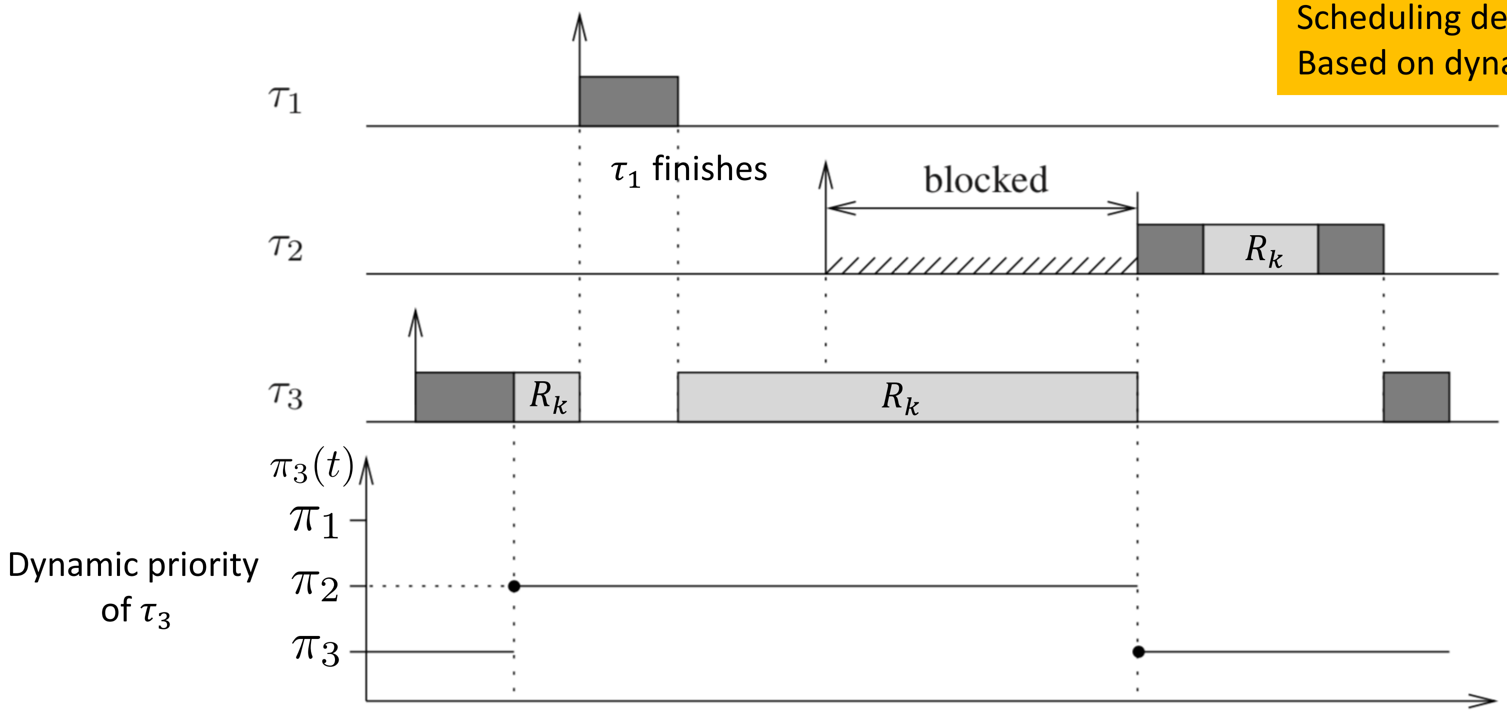
$$\boldsymbol{B_i = \max\{\delta_{j,k}: \pi_j < \pi_i, J_j \text{ uses resource } R_k\}}$$

Note: If no lower priority task uses any resources, then $B_i = 0$; i.e., $\max \emptyset = 0$

Approach #2: Highest Locker Priority (HLP)

- When a task requests resource R_k, elevate its priority to the priority of the highest priority task that ever shares (uses) resource R_k
- Avoids the unnecessary blocking of higher priority tasks that do not need resources (present in NPP)

Scheduling decisions are
Based on dynamic priorities $\pi_i(t)$



Highest Locker Priority: Blocking time computation

- When a task requests resource R_k, elevate its priority to the priority of the highest priority task that ever shares resource R_k
- **Observation:** *Task T_i can be blocked only by lower priority tasks that use a resource that is used by a task with priority **greater than or equal** to T_i*
- **Claim:** *A task T_i can be blocked for at most the duration of a **single** critical section of at most one lower priority task that uses a resource that is used by a task with priority $\geq \pi_i$*
- ***Ceiling of resource*** R_k *is the priority of the highest priority task that uses R_k*
    - $C(R_k) = \max_{i \in [n]} \{\pi_i : T_i \text{ uses } R_k\}$
- ***Claim recast***: *A task T_i can be blocked for at most the duration of a **single** critical section of at most one lower priority task that ever uses a resource R_k with $C(R_k) \geq \pi_i$*

Highest Locker Priority: Blocking time computation

- **Claim:** *A task T_i can be blocked for at most the duration of a* ***single*** *critical section of* ***at most one*** *lower priority task that ever uses a resource R_k with $C(R_k) \geq \pi_i$*

$$B_i = \max\{\delta_{j,k} : \pi_j < \pi_i, T_j \text{ uses } R_k, C(R_k) \geq \pi_i\}$$

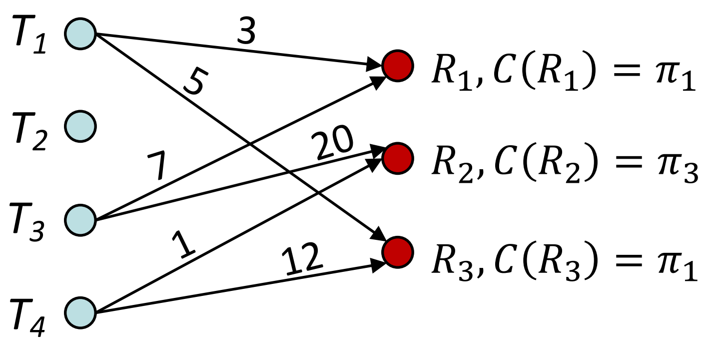
Proof of Claim

- Suppose task T_i is blocked by **two** critical sections
- Then both critical sections must belong to two **different** lower priority tasks (why?)

resource R_a used by task T_1 $\quad\quad \pi_1 < \pi_i \leq C(R_a) \quad (*)$
resource R_b used by task T_2 $\quad\quad \pi_2 < \pi_i \leq C(R_b)$

- Since T_i is blocked on both resources, it must be that tasks T_1 and T_2 were in their critical sections when task T_i arrived
- Then one of T_1 or T_2 must have preempted the other inside its critical section –> say T_1 preempted T_2 while T_2 is in its critical section using R_b
- This means that $\pi_1 > C(R_b)$
- But $\pi_i > \pi_1 \rightarrow \pi_i > C(R_b)$ *contradicts* $(*)$

A useful tool: The ***resource graph***



B_1 =?
B_2 = ?
B_3 = ?
$B_4 = 0$

$$B_i = \max\{\delta_{j,k} : \pi_j < \pi_i, T_j \text{ uses } R_k, C(R_k) \geq \pi_i\}$$

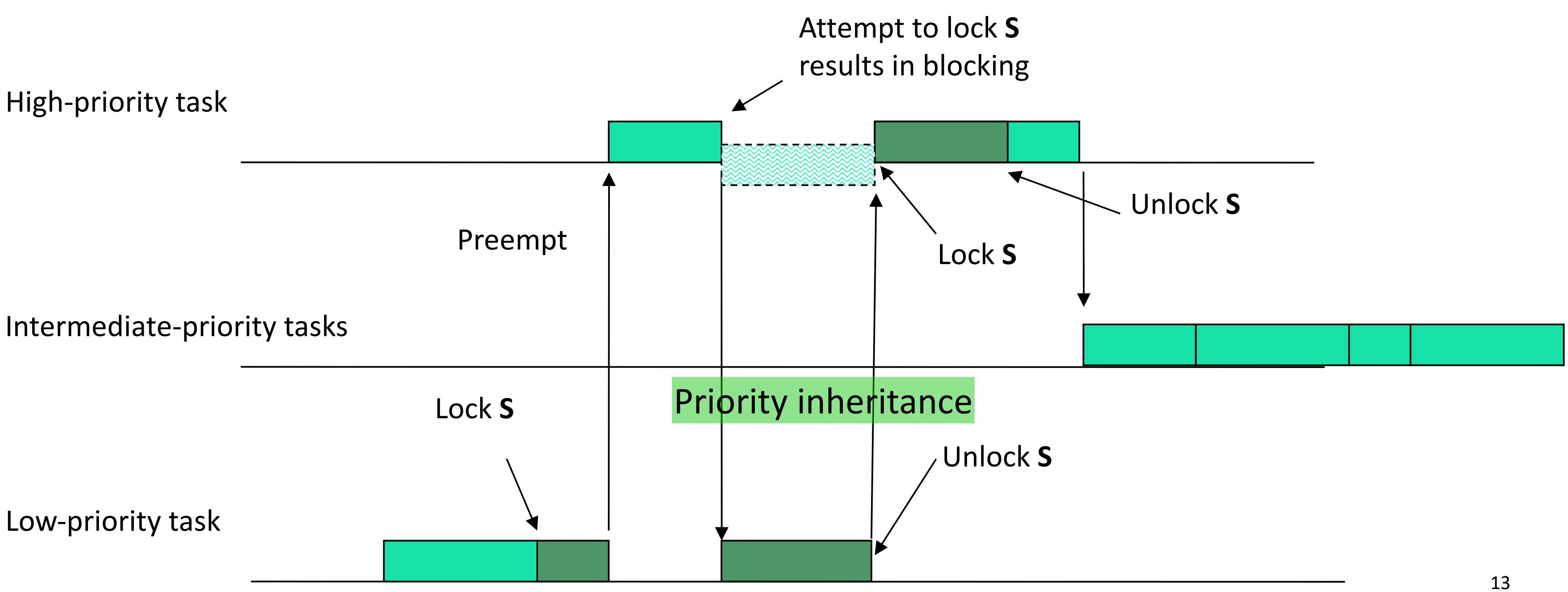
- List jobs in priority order and resources in any order, creating a node for each
- Create an edge between task T_j and resource R_k if T_j uses R_k
- Label arc (T_j, R_k) with the length of the *longest* critical section of T_j that uses R_k, $\delta_{j,k}$ (even if critical sections are nested)
- Label each resource node R_k by its ceiling $C(R_k)$

Highest Locker Priority: Problems

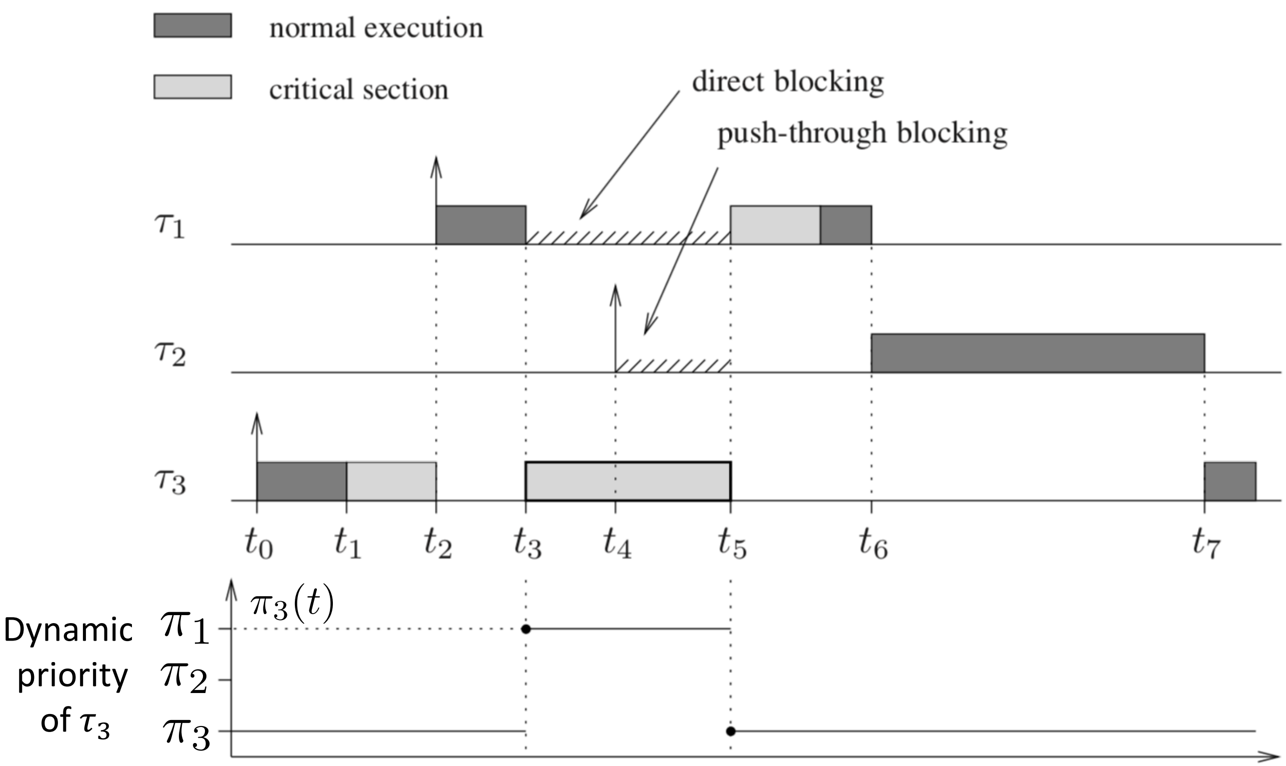
- HLP causes *unnecessary blocking*
  - A higher priority task is blocked on its ***arrival,*** not when it attempts to request the resource
  - **Solution:** delay blocking until the task ***attempts to request*** shared resource → **PIP**!

The priority inheritance protocol

- Allow a task to **inherit the priority** of the highest priority task that it is blocking
- When a hp task is blocked as it attempts to acquire a resource that is held by a lower priority task, it **transfers** its priority to that lower priority task



The priority inheritance protocol



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**)

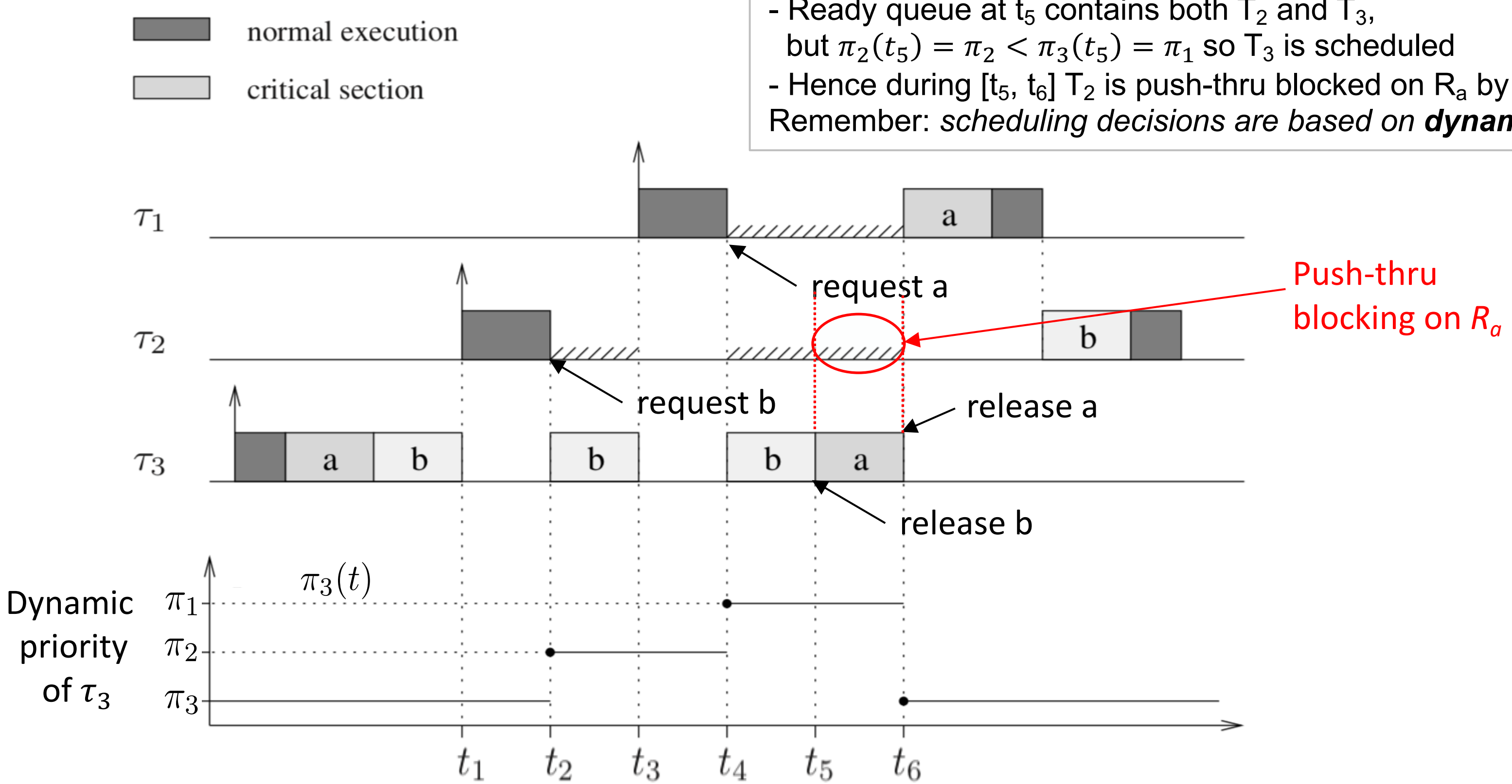
The priority inheritance protocol

- Resource **Release** Rule: When a task **releases** a resource, its dynamic priority $\pi(t)$ is set to the highest priority of the tasks **currently blocked by it**

- **Q:** When a task exits a critical section, does it always resume the priority it had when it entered?

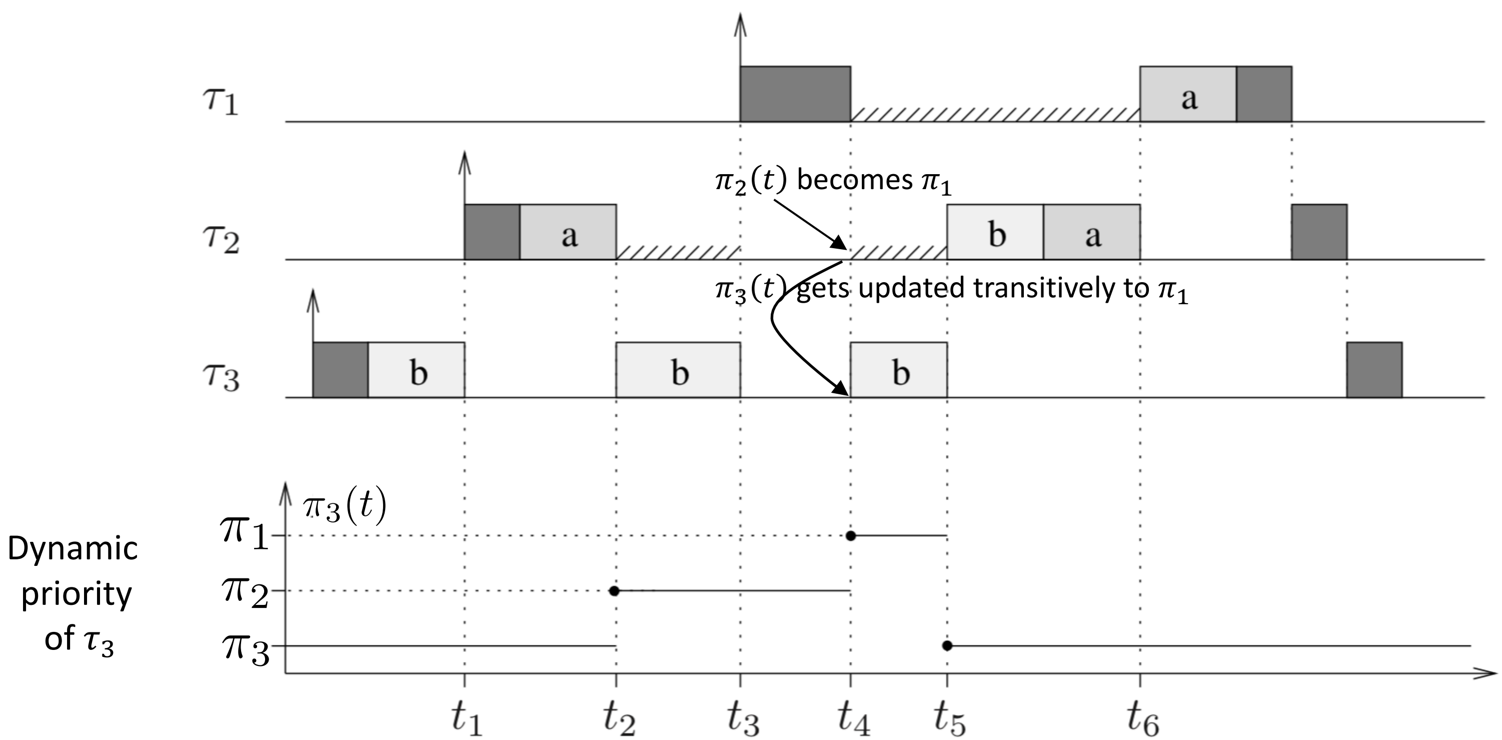
- At t_5, T_2 is put back in the ready queue
- Ready queue at t_5 contains both T_2 and T_3, but $\pi_2(t_5) = \pi_2 < \pi_3(t_5) = \pi_1$ so T_3 is scheduled
- Hence during $[t_5, t_6]$ T_2 is push-thru blocked on R_a by T_3

Remember: *scheduling decisions are based on* ***dynamic*** *priorities*



The priority inheritance protocol

- Priority inheritance is *transitive*
  - *However, transitive priority inheritance can occur only in the presence of **nested critical sections** (proof in book Lemma 7.2)*



Maximum blocking time

- **Claim1:** *If there are* $\boldsymbol{\ell_i}$ ***lower-priority*** *tasks that can block task* τ_i*, then* τ_i *can be blocked for at most the duration of* $\boldsymbol{\ell_i}$ ***critical sections*** *(one for each of the* ℓ_i *lower-priority tasks), regardless of the number of semaphores used by* τ_i
  - A critical section $z_{j,k}$ of a lower priority task T_j can block T_i if it causes either direct or push-thru blocking to T_i
- **Claim2:** *If there are* $\boldsymbol{s_i}$ ***distinct semaphores*** *that can block task* τ_i*, then* τ_i *can be blocked for at most the duration of* $\boldsymbol{s_i}$ ***critical sections****, one for each of the* s_i *semaphores, regardless of the number of critical sections used by* τ_i

- Then, if all critical sections are of equal length, b_i
  - Blocking time $B_i = b_i \times \min(\ell_i, s_i)$
- What if the critical sections are of differing lengths?

General approach to computing blocking times

- What if the critical sections are of differing lengths?
- Will consider a safe approximation to blocking time.
- **Assumption***: no nested critical sections*
- 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

Maximum blocking time

This is just a safe approximation
(upper bound on exact blocking time)
Exact blocking time computation is intractable

- What if the critical sections are of differing lengths?
- $\delta_{j,k}$: length of *longest* critical section among all those of T_j guarded by sempahore S_k
- Let $z_{j,k}$ denote the critical section (CS) whose length is $\delta_{j,k}$

- **(1)** Blocking due to lower priority tasks that ***can block*** T_i (**claim1**)
  - A task T_j ***can block*** T_i if it has lower priority than T_i and uses some resource R_k that is also used by a task with priority greater than or equal to T_i

$$B_i^{\ell} = \sum_{j=i+1}^{n} \max_{k \in \{1,\ldots,m\}} \{\delta_{j,k} : z_{j,k} \text{ is max. length CS that can block } T_i\}$$

- **(2)** Blocking due to semaphores that can block T_i (**claim 2**)
  - A resource ***can block*** T_i if it is used by a lower priority task **and** a task with priority $\geq \pi_i$

$$B_i^{s} = \sum_{k=1}^{m} \max_{j>i} \{\delta_{j,k} : z_{j,k} \text{ is max. length CS that can block } T_i\}$$

$$B_i = \min\left(B_i^{\ell}, B_i^{s}\right)$$

Simplifying matters

- Use resource ceilings (very useful device)
- Recall: $C(R_k) = \max_{i \in [n]} \{\pi_i : T_i \text{ uses } R_k\}$
- **Claim:** *In the absence of nested critical sections, a critical section $z_{j,k}$ of τ_j using resource R_k can block τ_i only if $\pi_j < \pi_i \leq C(R_k)$*
  - Proof in text; Lemma 7.5

$$B_i^{\ell} = \sum_{j=i+1}^{n} \max_k \{\delta_{j,k} : z_{j,k} \text{ is max. length CS that can block } T_i\}$$

$$B_i^{s} = \sum_{k=1}^{m} \max_{j>i} \{\delta_{j,k} : z_{j,k} \text{ is max. length CS that can block } T_i\}$$

$$B_i^{\ell} = \sum_{j=i+1}^{n} \max_k \{\delta_{j,k} : C(R_k) \geq \pi_i\}$$

$$B_i^{s} = \sum_{k=1}^{m} \max_{j>i} \{\delta_{j,k} : C(R_k) \geq \pi_i\}$$

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

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

Each instance of a task might 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
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?

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

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

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 up to 6 time units (because it might wait for T_3)

It can be blocked by T_3 on resource R_1 for up to 3 time units

It can be blocked by T_4 on resource R_2 for up to 5 time units

Then maximum wait on lower priority tasks is $\boldsymbol{B_1^{\ell} = 6 + 3 + 5 = 14}$

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 6 units (T_2 can block T_1 for 6 units or T_4 can block T_1 for 5 units)

Then maximum wait for resources is $\boldsymbol{B_1^{s} = 3 + 6 = 9}$

Then $B_1 = \min(14, 9) = 9$

$$\frac{9}{20} + \frac{3}{20} < 1$$

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 $\boldsymbol{B_2 = 5 + 4 = 9 = B_2^{\ell}}$ (check for yourself that $\boldsymbol{B_2^s = 12}$)

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 might execute with priority of T_1 (priority inheritance)

T_4 might 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

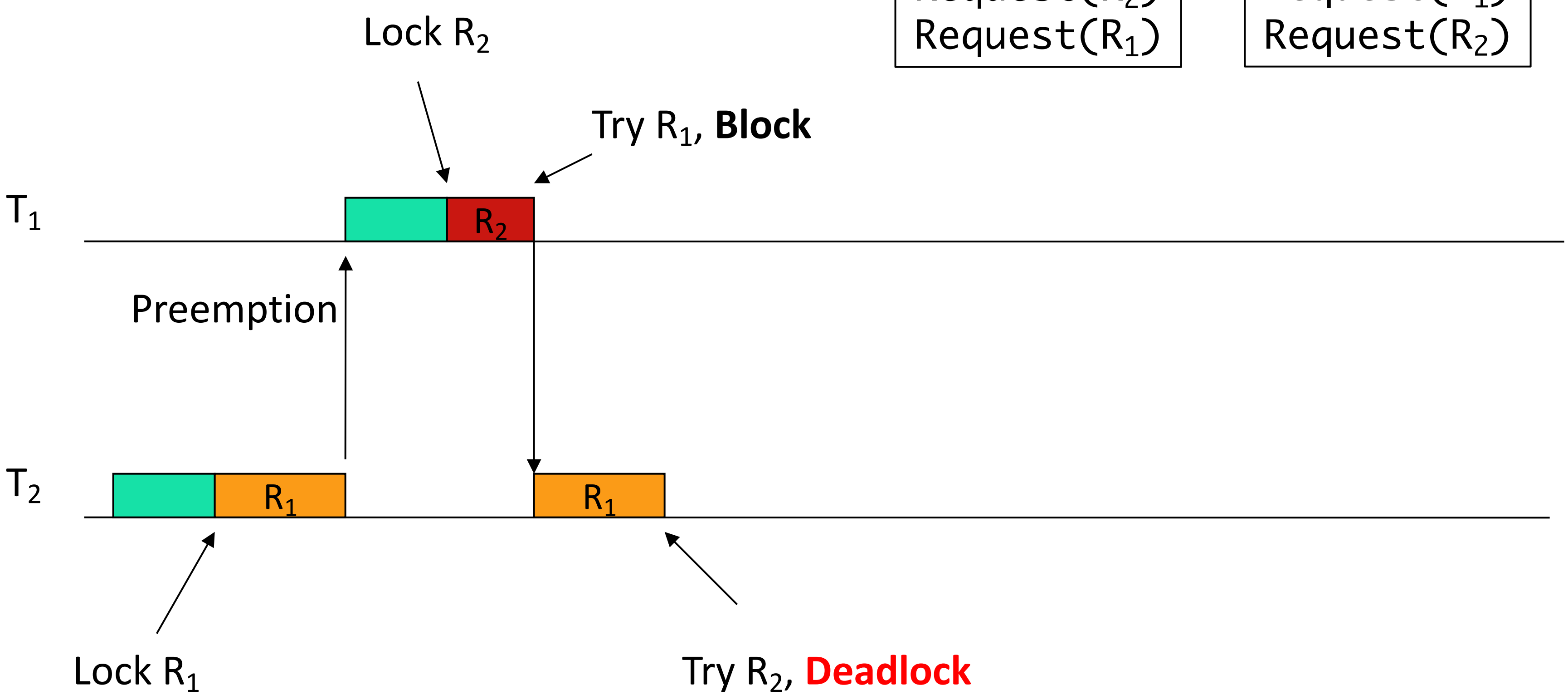
$$\left(\frac{3}{20} + \frac{6}{30} + \frac{10}{50} + \frac{8}{80}\right) = 0.65$$

T_4 is schedulable

Does priority inheritance solve all problems?

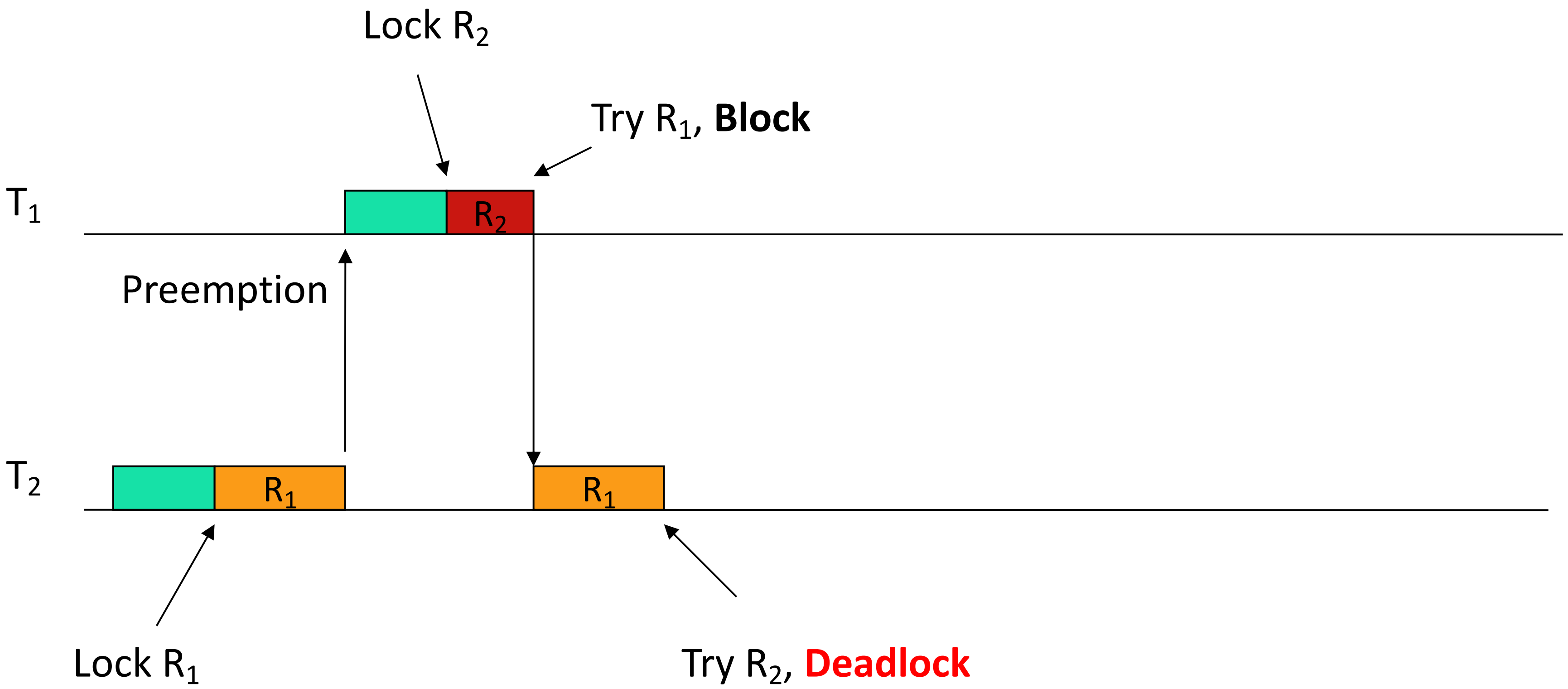
- Actually, not all problems
- We can still have a deadlock if resources are locked in opposing orders

T_1	T_2
Request(R_2) Request(R_1)	Request(R_1) Request(R_2)



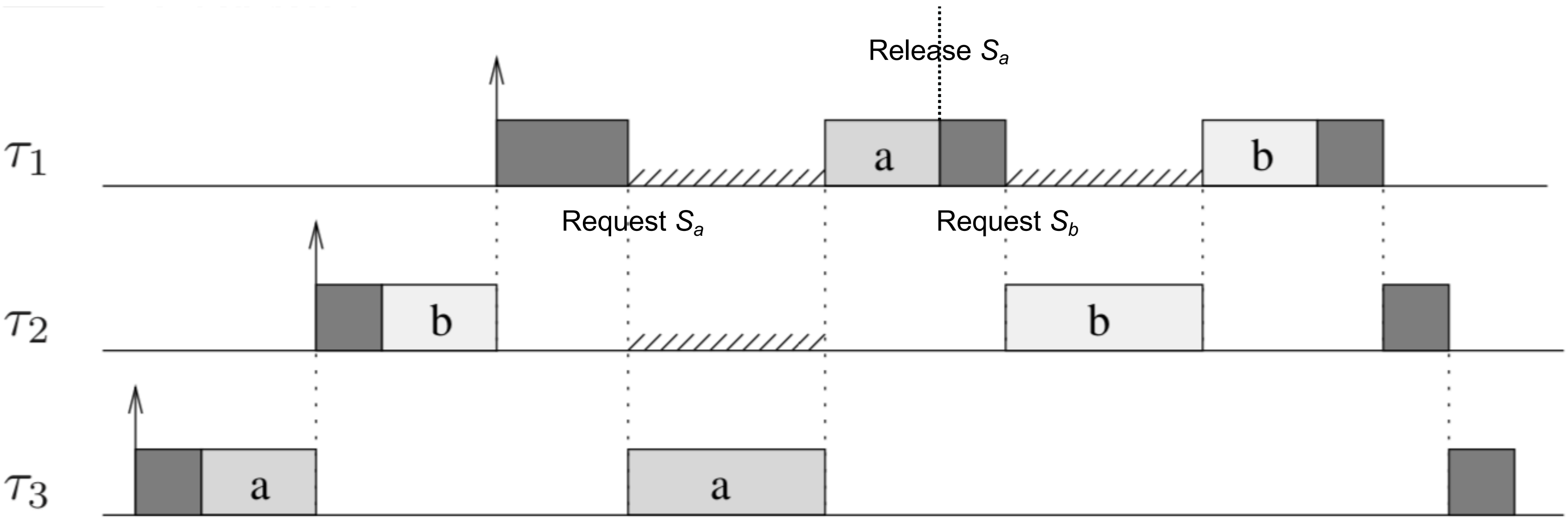
Deadlocks

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



Another problem with PIP: *Chained blocking*

- When τ_1 attempts to use its resources, it is blocked for the duration of **2** critical sections:
  - once to wait for τ_3 to release S_a
  - and then to wait for τ_2 to release S_b
- In the worst case, if τ_1 accesses *n* distinct semaphores that have been locked by *n* lower-priority tasks, τ_1 will be blocked for the duration of *n* critical sections.



Avoiding Multiple Blocking

- When a task enters a critical section, make sure that there are sufficient resources to satisfy its maximum resource requirements
- **Consequence**: When a task enters a critical section, it cannot be blocked on resources
- Do not allow a task to enter a critical section if there are locked resources that can block it
- **Meaning**: do not allow task T_i to enter a critical section at time t if there is a locked resource R_k with $C(R_k) \geq \pi_i$
- Iff allow task T_i to enter a critical section at time t if $\pi_i > C(R_k)$ for **every** locked resource R_k
- Iff allow task T_i to enter a critical section if
$$\pi_i > \max\{C(R_k): R_k \text{ locked at time } t\} \equiv C(t)$$

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 strictly higher than the highest priority ceiling of all ***currently*** locked semaphores (let us say this belongs to semaphore R_h; *Can there be more than one?*)
  - 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

Priority ceiling protocol

- **Recall: Priority Ceiling** of resource R_k: $C(R_k) = \max_{i \in [n]}\{\pi_i : T_i \text{ uses } R_k\}$
- Suppose task T_i requests a resource R_k at time t
- Let $R_h = \text{ argmax}_j\{C(R_j)\text{: resource } R_j \text{ is locked at time } t\ \}$
  - Can there be more than one such R_h?
- Define **System Ceiling** as the highest ceiling of currently locked semaphores → $C(t) = C(R_h) = \max\{C(R_j): R_j \text{ locked at time } t\}$
  - System ceiling updated whenever a resource is acquired/released
- If $\pi_i \leq C(t)$, then T_i is denied access to the resource
  - **Exception:** If $\pi_i \leq C(t)$ but T_i is the task locking R_h then grant T_i access to R_k (o/w T_i will block itself!)
  - T_i is said to be blocked by the task holding semaphore R_h
  - T_i then trasfers its priority to task holding R_h

Priority ceiling protocol

- To avoid multiple blocking, this rule does not allow a task to enter a critical section if there are locked semaphores that could block it.
- This means that *once a task enters its first critical section, it can **never** be blocked by lower-priority tasks until its completion*

Similarity to PIP
Priority Inheritance rule

Fundamental difference from PIP
PIP is *greedy*, PCP is not!
In what sense?

A task can be blocked on a *free* resource in PCP
Impossible in PIP

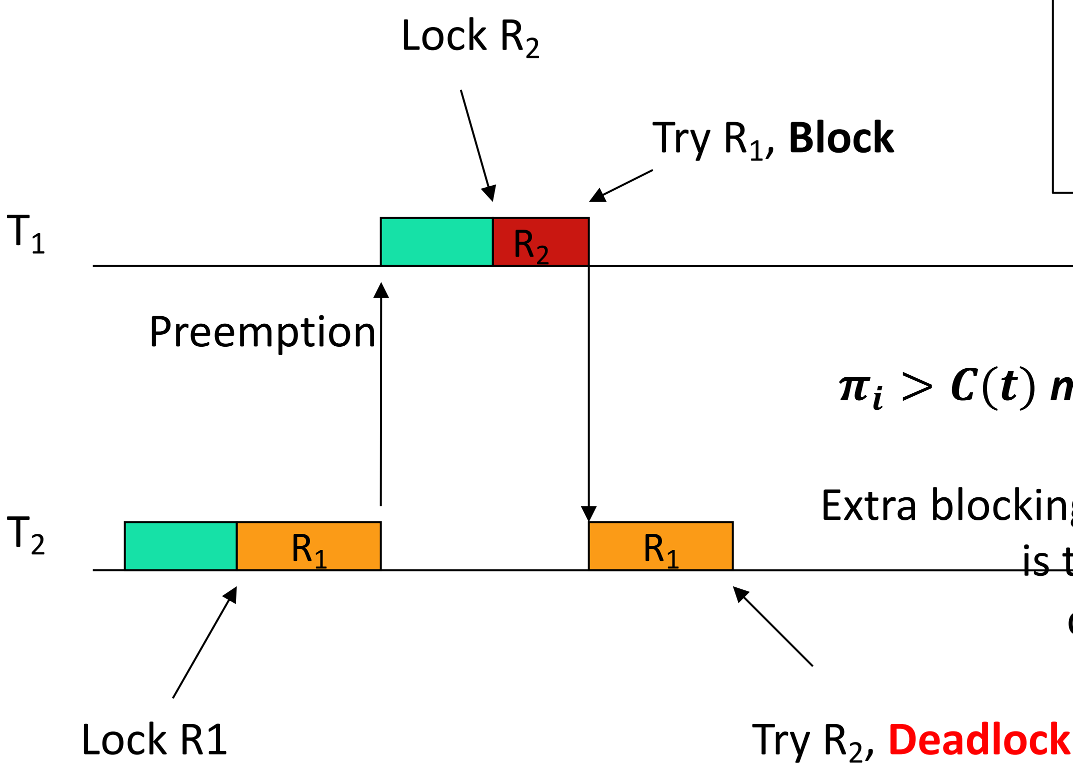
Extra blocking caused by non-greediness of PCP is the price to avoid deadlocks & chained blocking called ***avoidance blocking*** *or* ***ceiling blocking***

Deadlocks?

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

T_1	T_2

`Request(R2)`
`Request(R1)`
 	
`Signal(R1)`
`Signal(R2)`
	`Request(R1)`
	`Request(R2)`
	
	`Signal(R2)`
	`Signal(R1)`



$\pi_i > C(t)$ ***means that this cannot happen!***

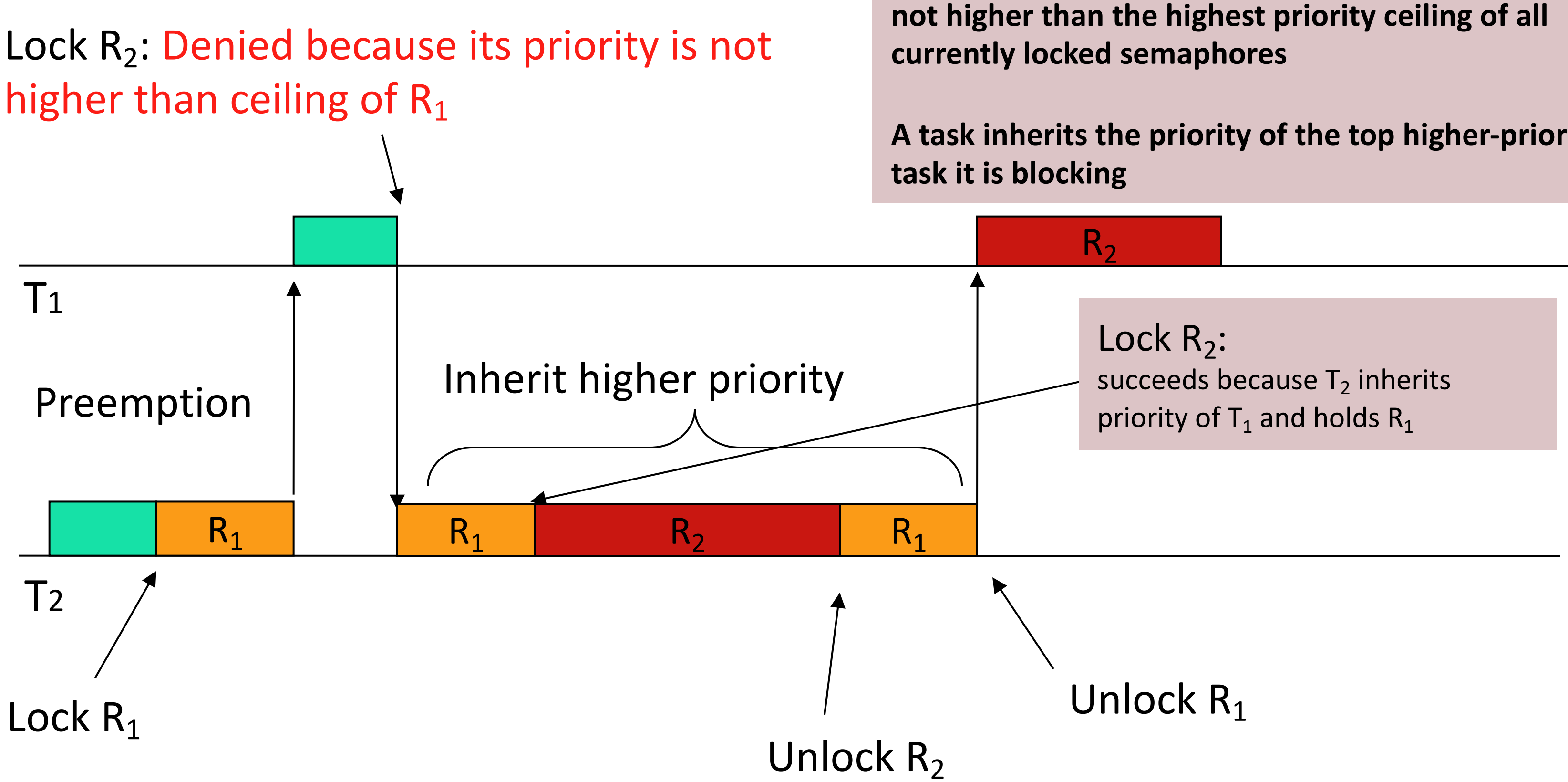
Extra blocking caused by non-greedy nature of PCP is the price to avoid deadlocks called ***avoidance blocking***

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



PCP blocking time computation

- A task can be blocked by the duration of at most one critical section of at most one lower priority task
- Much simpler to compute than PIP
- Should consider the three types of blocking and take the max of them
- Resource graph to our rescue!

Schedulability test for priority ceiling protocol

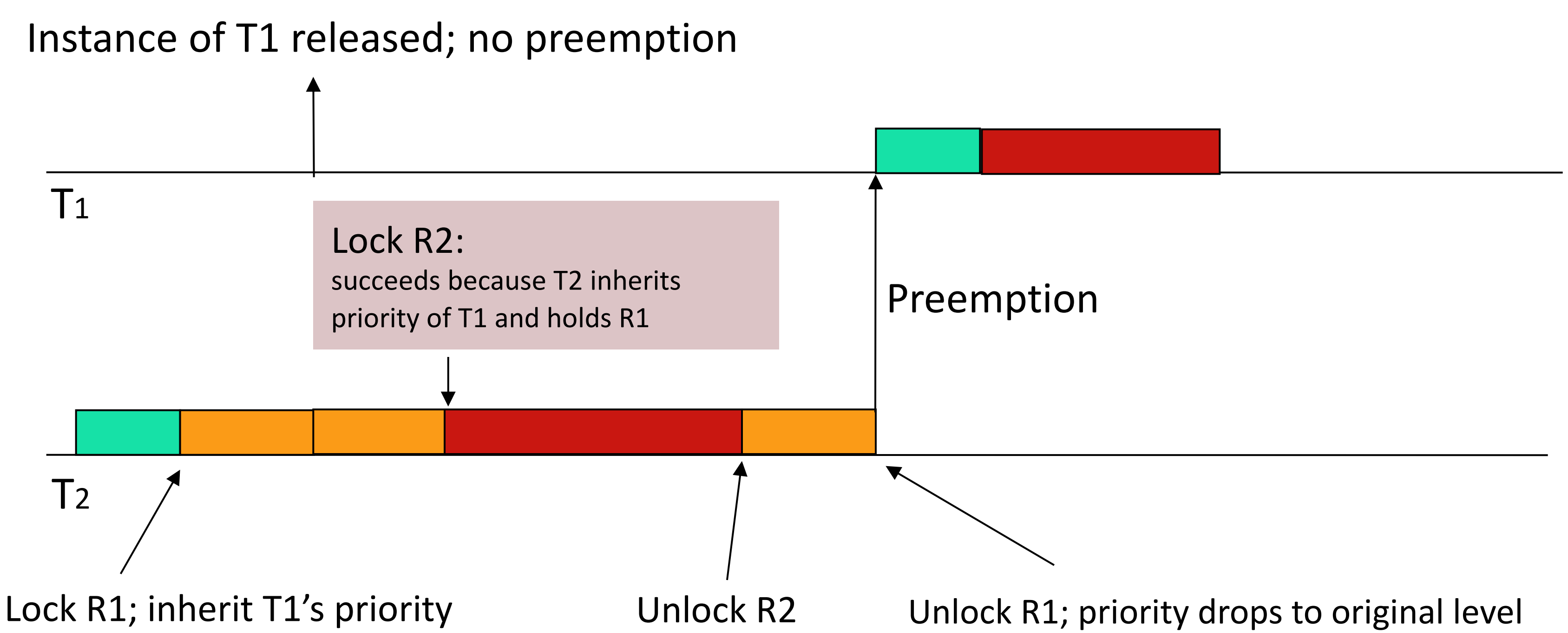
- The test is the same as with the priority inheritance protocol
  - Worst-case blocking time may change when compared to PIP

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

For task T_k

Recall: Highest Locking Protocol (HLP) = PCP with Immediate inheritance

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

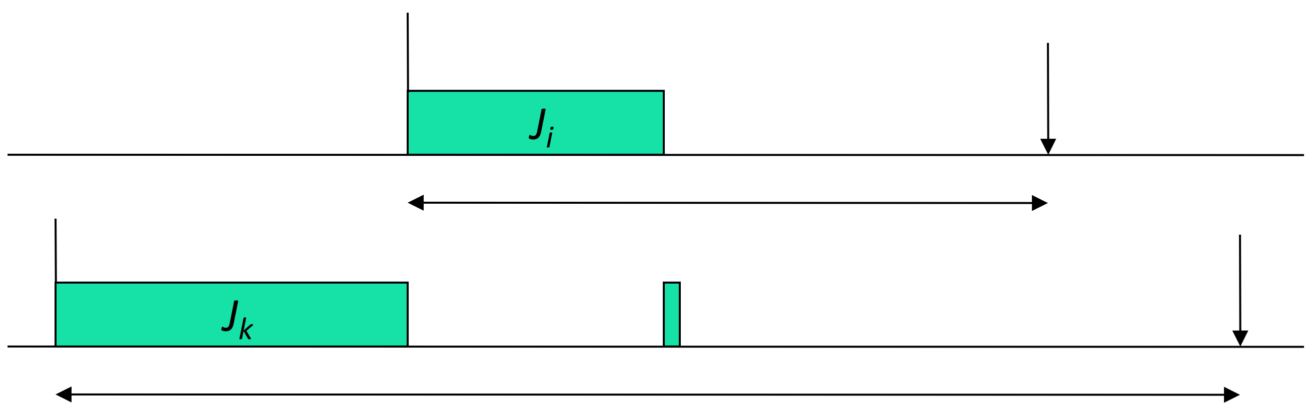


Stack-based resource policy

- Let us attempt to support *dynamic-priority* systems
- Does PCP extend directly?
- Task priorities in dynamic-task (equivalently fixed-job) priority systems might change at every invocation
  - Resource ceilings are no longer static: Must be updated potentially at every invocation. High runtime overhead!
- **Observation:** That a job J_h has a higher priority than another job J_l and they both require some resource does not imply that J_l can directly block J_h
  - This blocking can occur **only when it is possible for J_h to *preempt* J_l**
- When determining whether a free resource can be granted to a job, it is **not necessary** to be concerned with the resource requirements of all higher-priority jobs; *only those that can preempt the job*

Stack-based resource policy

- Since for resource contention purposes we only care about the jobs that a job can possibly preempt, let us identify *the event that causes a job to be preempted in any task-dynamic priority scheduling scheme*
- In a dynamic-task policy, when can a job preempt another job?



Stack-based resource policy

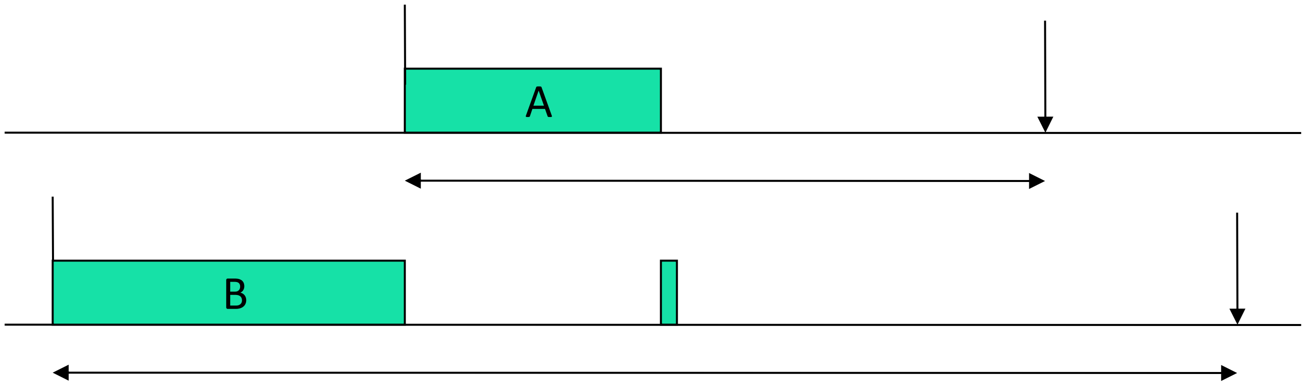
- A quantity that encodes a job's ability to preempt other jobs
- $(*)$ Formally, we want to associate job J_k with quantity ψ_k such that
  if $\psi_k \le \psi_i$, then it is **not** possible for J_k to preempt J_i
- J_k cannot preempt $J_i \Leftrightarrow$ either $r_k \le r_i$ or $\pi_k \le \pi_i$
- Then $(*)$ translates to:

$$\boxed{(**) \quad \text{if } r_k > r_i \text{ and } \pi_k > \pi_i\text{, then } \psi_k > \psi_i \text{ (it's possible for } J_k \text{ to preempt } J_i\text{)}}$$

- A ψ_k satisfying $(**)$ is called the **preemption level** of job J_k
- **Q:** How does ψ_k look like for EDF?

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)`



Stack-based resource policy

- The preemption level ψ_i of J_i is any quantity satisfying the statement:

  if $r_k > r_i$ and $\pi_k > \pi_i$, then $\psi_k > \psi_i$

- **Q:** How does ψ_i look like for EDF?

- EDF:
  - $\pi_k > \pi_i$ iff $r_k + D_k < r_i + D_i$
  - So $r_i < r_k$ implies $r_i + D_k < r_i + D_i \Rightarrow D_k < D_i$
  - $\psi_k > \psi_i \Leftrightarrow D_k < D_i$

  - For EDF, this quantity is for the entire **task**, not only a job!

- *The possibility that a task preempts other tasks remains constant throughout all its invocations*

  - *Task's preemption level is static; can be computed offline once and for all*

- *EDF is one such **fixed preemption-level** system*

  - In such systems, the potentials of resource contentions do not change with time, just as in fixed-priority systems, and hence can be analyzed *statically*

Stack-based resource policy

- In fixed-**preemption** level systems, the set of critical sections that can block T_i are
$$\{z_{j,k}: \psi_i > \psi_j, C(R_k) \geq \psi_i\}$$
- **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

- Resource ceiling $C(R_k) = \max\{\psi_i : T_i \text{ uses } R_k\}$
- System ceiling $C(t) = \max\{C(R_k) : \text{resource } R_k \text{ is being used at time } t\}$

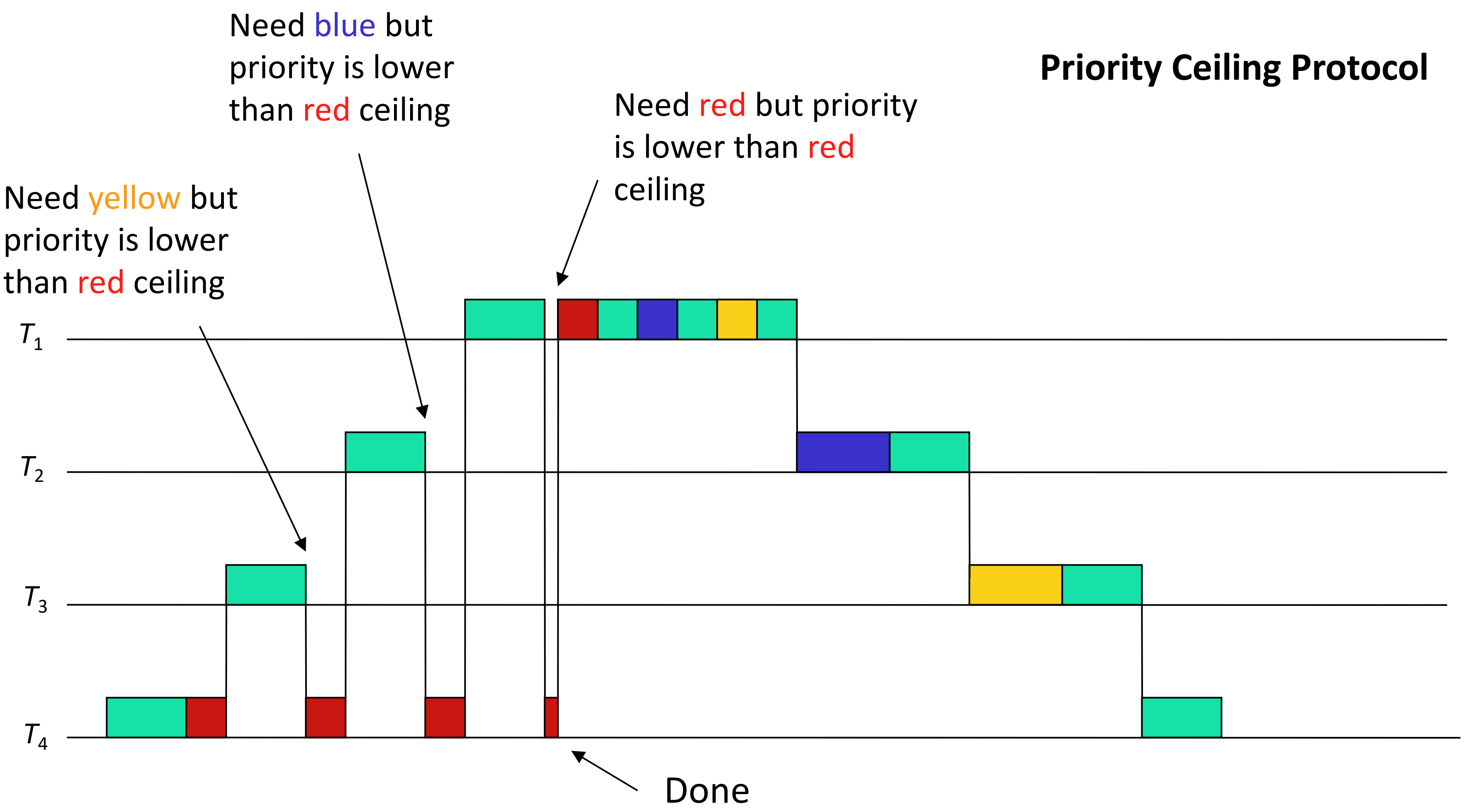
SRP Preemption Test

A task can preempt another task if
- it has the highest priority and
- its preemption level is higher than the system ceiling

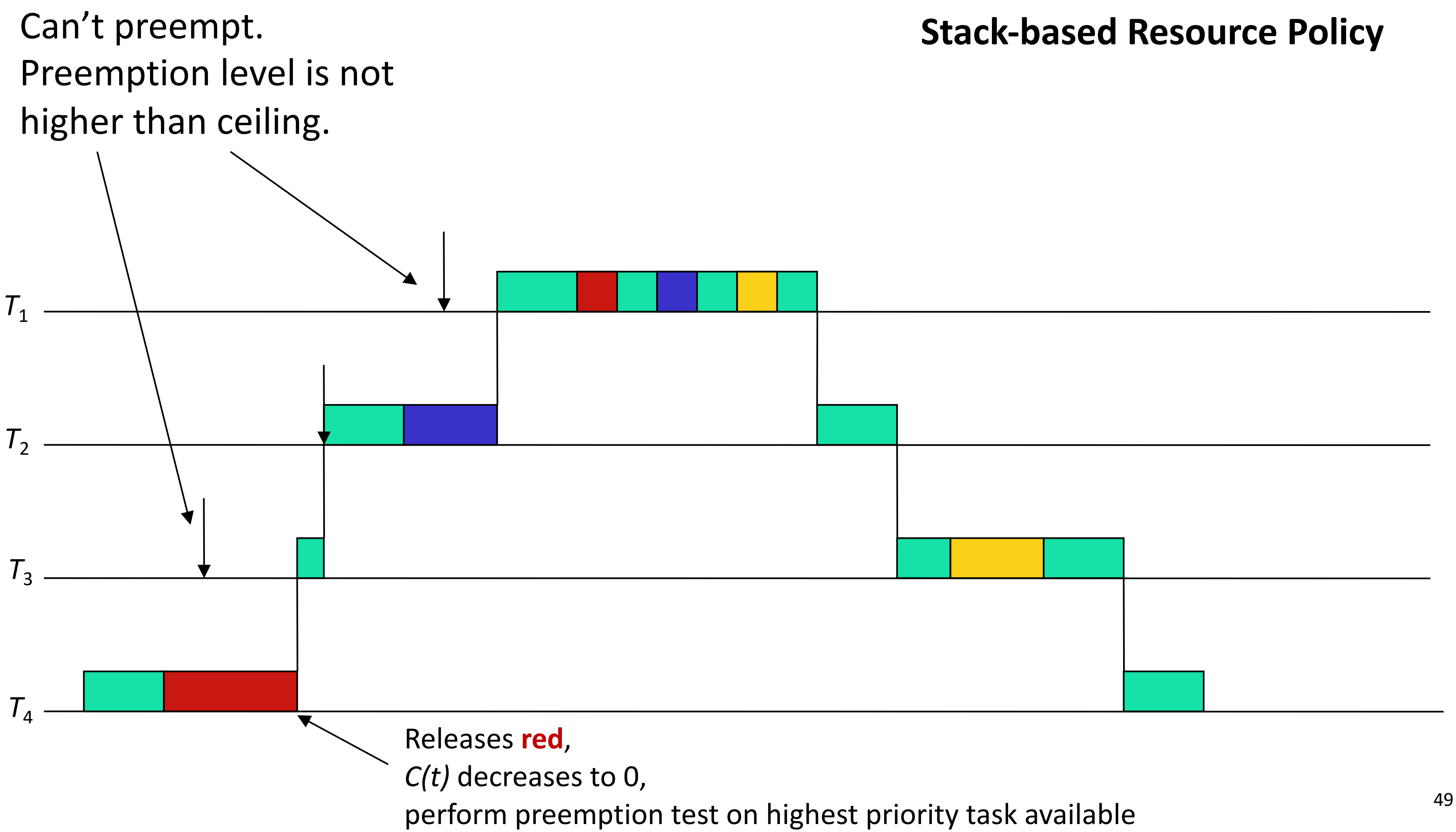
If T_i is the highest priority task at time t and $\psi_i > C(t)$ then allow T_i to preempt, otherwise block it

- Perform preemption test when a task arrives (on the arriving task) and on highest priority task when $C(t)$ decreases (a resource is released)

Priority ceiling vs. stack-based resource policy



Priority ceiling vs. stack-based resource policy

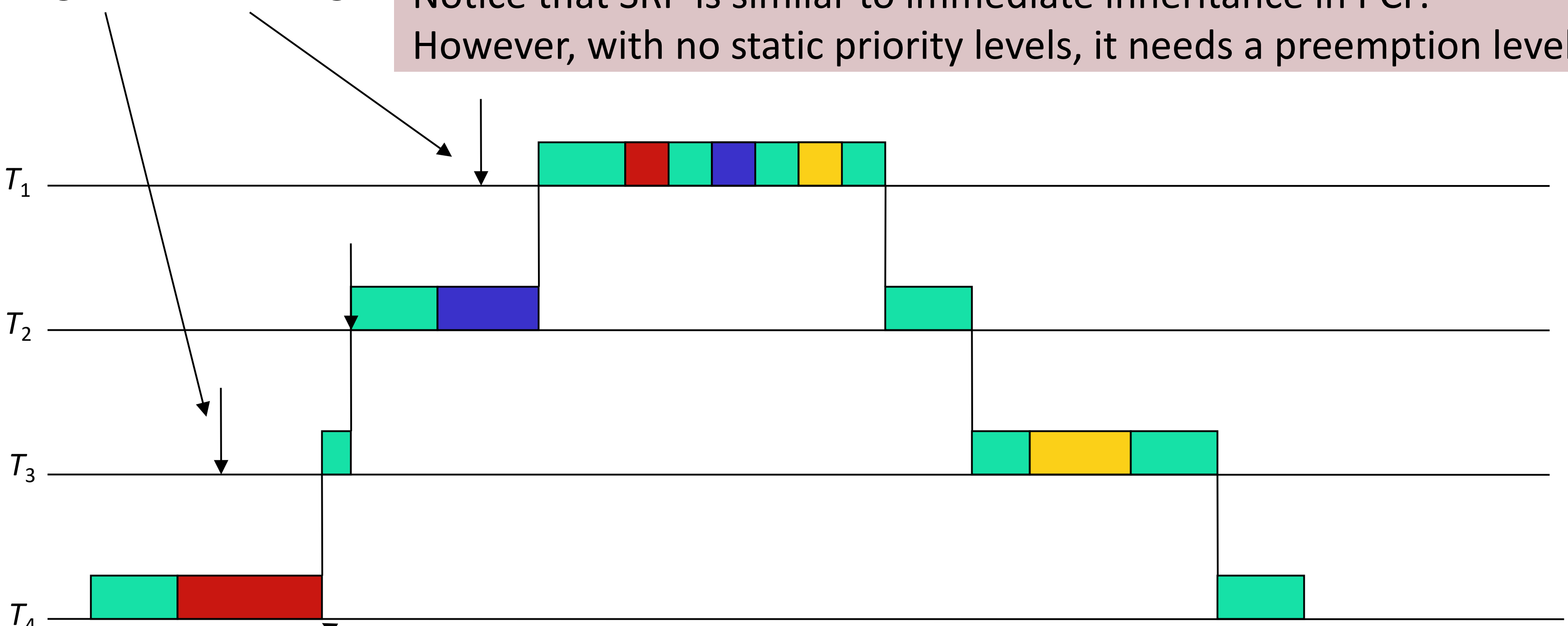


Priority ceiling vs. stack-based resource policy

Stack-based Resource Policy

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

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



Releases **red**,
C(t) decreases to 0,
perform preemption test on highest priority task available

Stack-based resource policy

- **Q:** What does it mean when a task passes the preemption test?
- **A:** the resources that are currently available are sufficient to satisfy the maximum requirement of task T_h and the maximum requirement of every task that could preempt T_h.
  - This means that once T_h starts executing, it will never be blocked for resource contention.
- **Remarks**
  - SRP avoids deadlocks. Why?
  - Resources are only allocated when a task *requests* them, not when it preempts
    - A higher-priority job may preempt and use the resources between these critical sections
  - A task can be blocked by the preemption test even though it does not require any resource. This is needed to avoid unbounded priority inversion.
  - The preemption test has the effect of imposing priority inheritance
    - an executing task that holds a resource modifies the system ceiling and resists preemption as though it inherits the priority of any tasks that might need that resource

Analysis with EDF and SRP

- As simple as other protocols

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

For task T_k

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

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

What is the “stack” in Stack-based Resource Sharing Protocol?

- **Two things:**
  1. Can be implemented using a stack. How?
  2. Allows tasks to share the run-time stack

In-class activity

Determine if the following task set can be scheduled using the *rate monotonic scheduling policy* with the *priority ceiling protocol* to control resource access.

Task	e_i	P_i	Resources used
T_1	4	10	R_1, R_2
T_2	5	20	R_2, R_3
T_3	10	35	R_3
T_4	2	40	R_1

The duration for which each resource is used by the tasks is specified in the following table. You may assume that a task locks only one resource at a time.

Resource	Duration
R_1	2
R_2	1
R_3	2

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)