Embedded and Real-Time Systems

Spring 2021

Hamed Farbeh

farbeh@aut.ac.ir

Department of Computer Engineering

Amirkabir University of Technology

Lecture 12

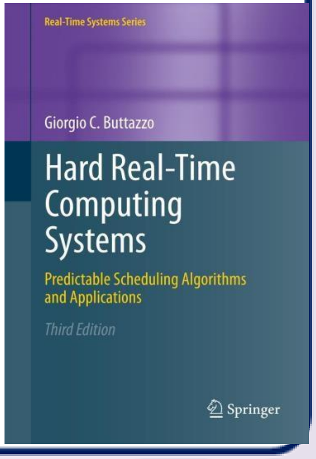
Copyright Notice

This lecture is adopted from

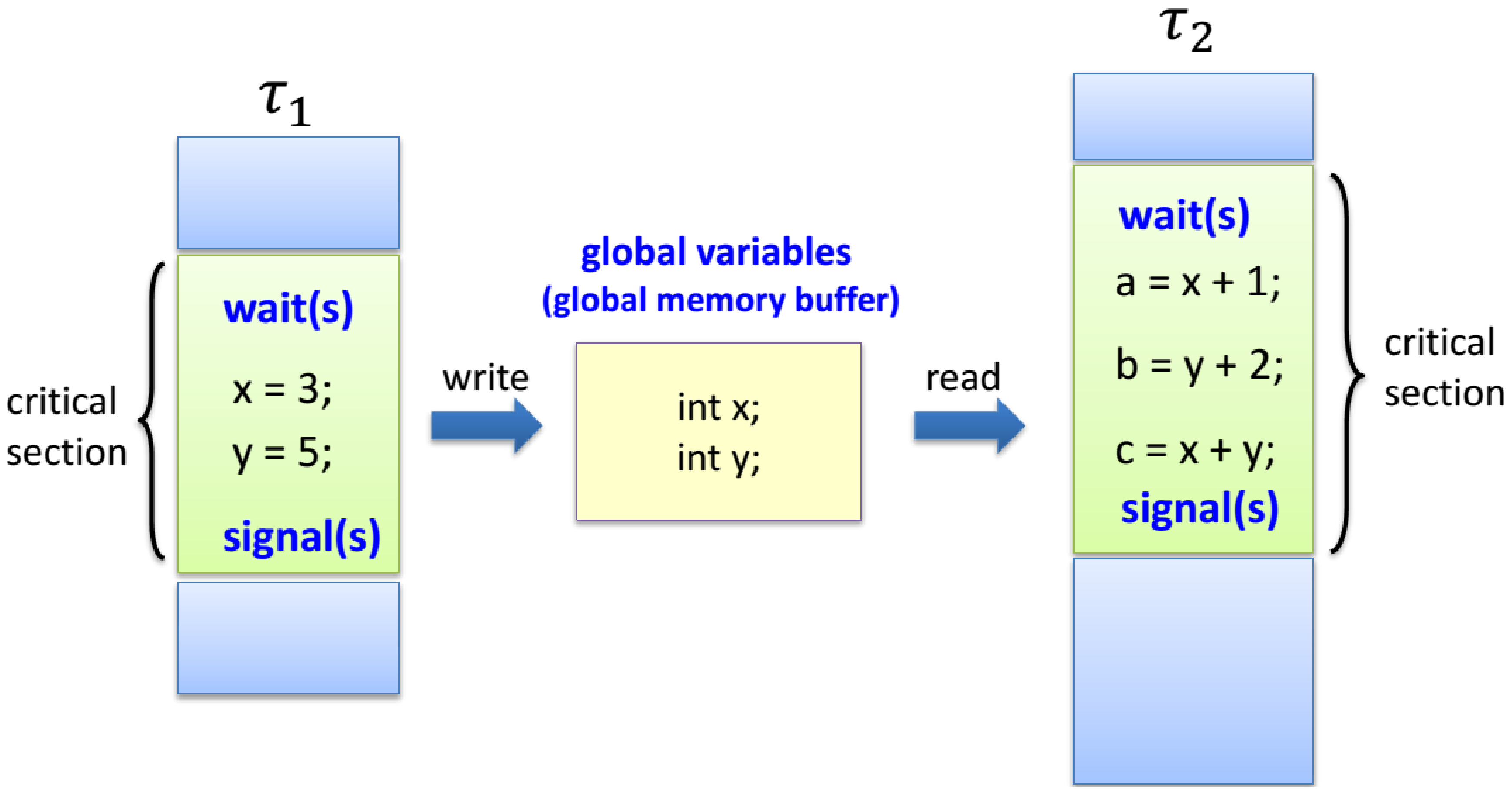
IN4343 Real-Time Systems Course 2018 – 2019, Mitra Nasri, Delft University of Technology

Handling Shared Resources

Buttazzo's book, chapter 7

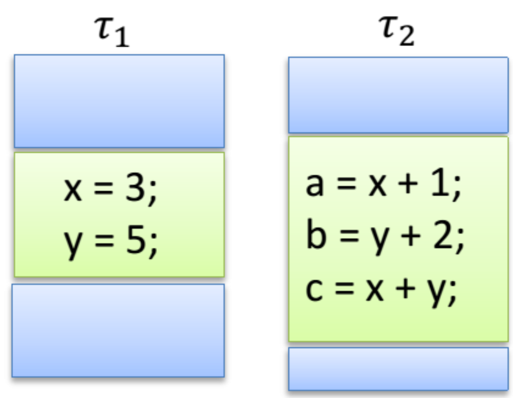


Critical sections



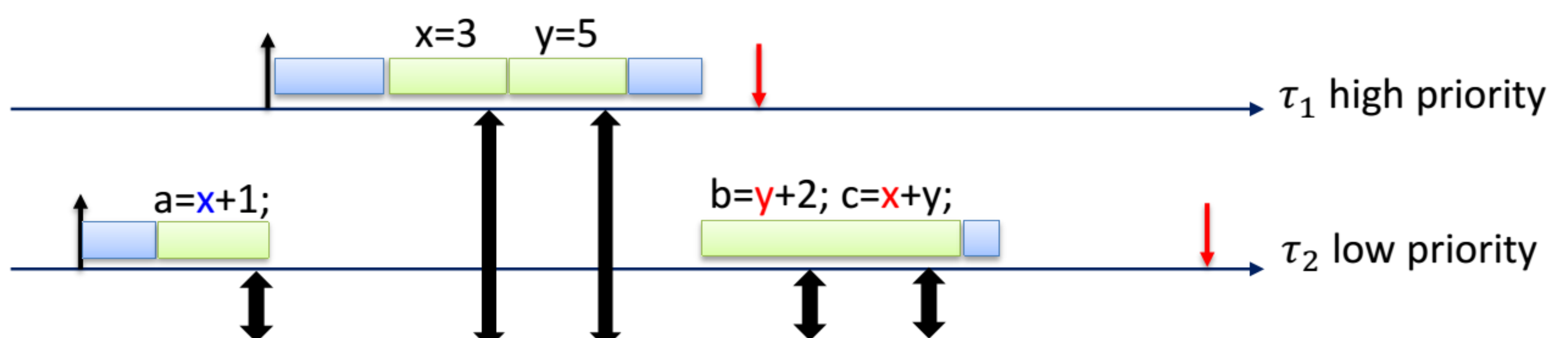
If the system supports concurrent execution (e.g., preemptive scheduling), then the access to the shared resources must be protected, e.g., by Semaphores

Why do we need to protect shared resources?



Imagine that (x, y) is the position of a robot in the room

Solution?



x	0	0	3	3	3	3
y	0	0	0	5	5	5
a	0	1	1	1	1	1
b	0	0	0	0	7	7
c	0	0	0	0	0	8

Data inconsistency!

Task τ_2 has used data from two different locations

Semaphores

- Each shared resource is protected by a different semaphore.
  - s = 1 => free resource
  - s = 0 => busy (locked) resource

τ_1

```
wait(S)
x = 3;
y = 5;
signal(S)

wait(B)
a = t + 3;
b = 5;
signal(B)
```

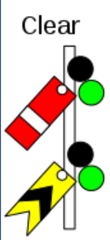
wait(s):

if s == 0, **then**

The task must be blocked on a queue of the semaphore. The queue management policy depends on the OS (usually it is FIFO or priority-based).

else

set s = 0.

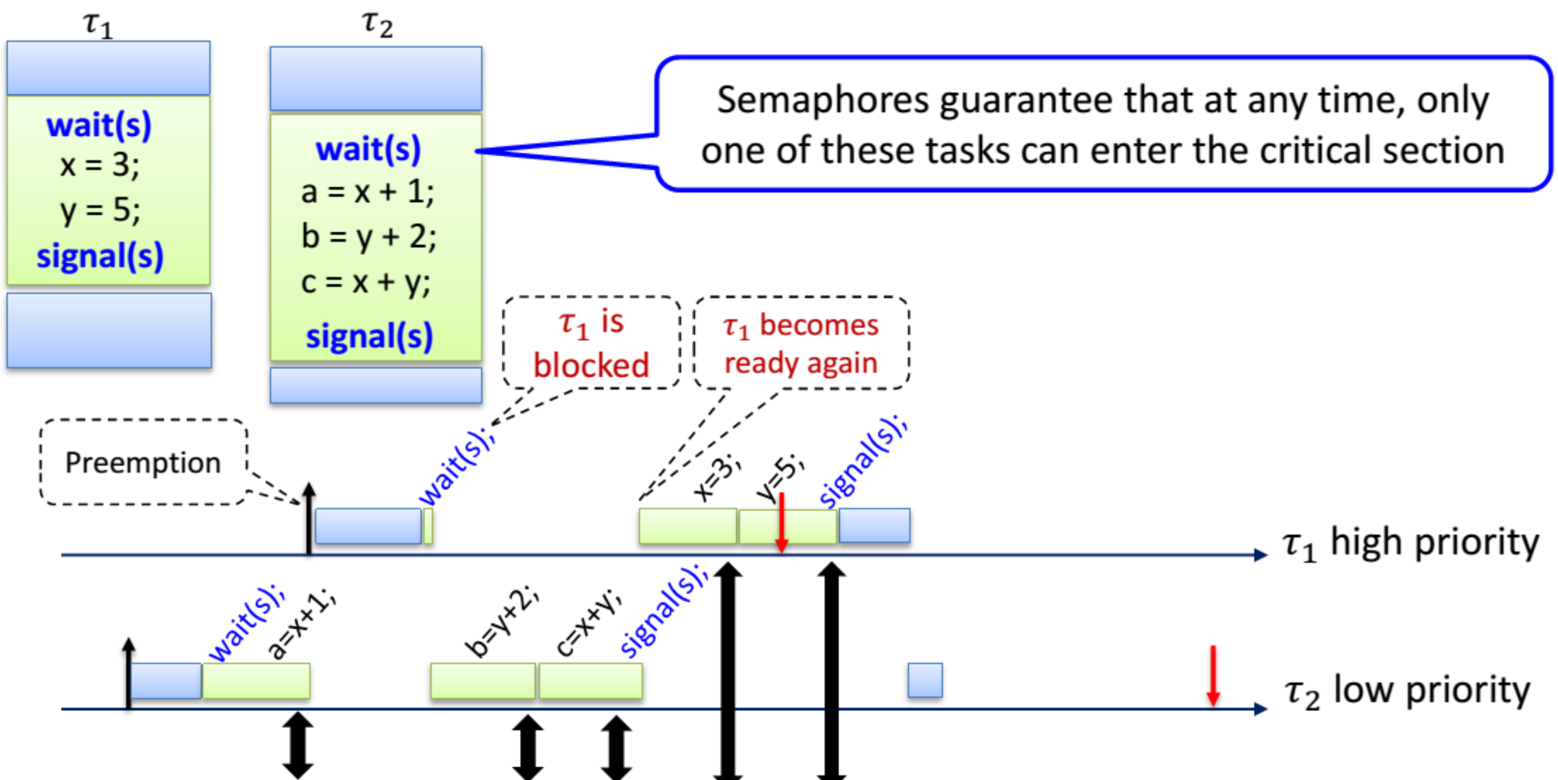


signal(s):

if there are blocked tasks, **then** the first in the queue is awaken (s remains 0),

else set s = 1.

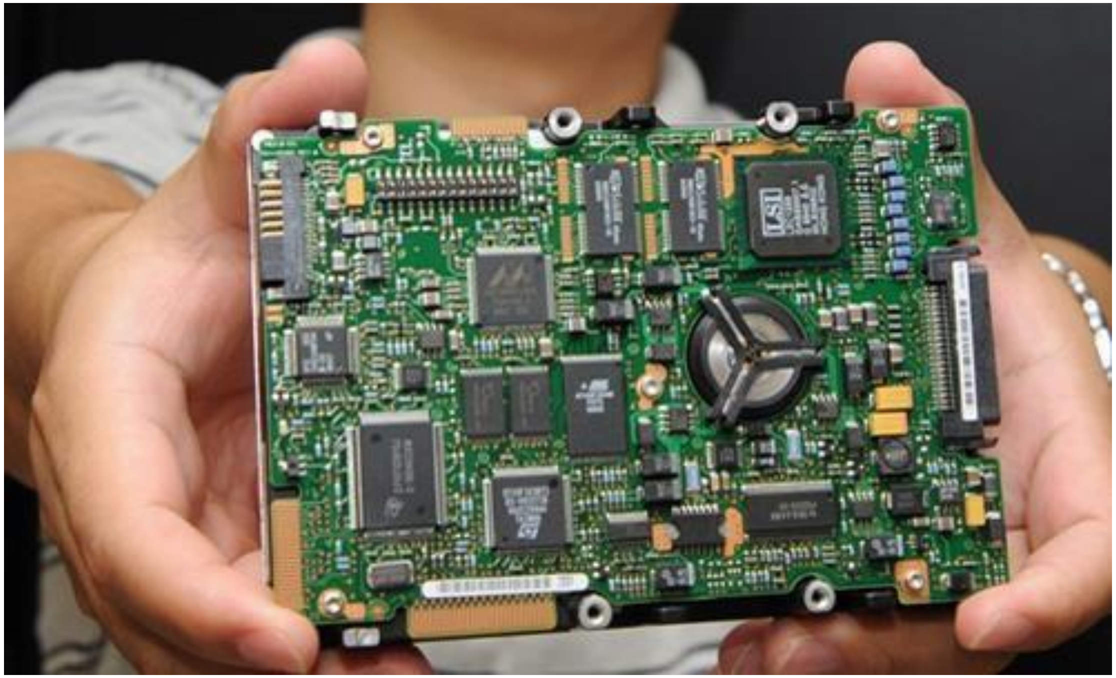
Using semaphores to protect shared resources



x	0	0	0	0	3	3
y	0	0	0	0	0	5
a	0	1	1	1	1	1
b	0	0	2	2	2	2
c	0	0	0	0	0	0

No data inconsistency

Guidelines for real-time systems engineers



Hints: shorten critical sections

Make critical sections as short as possible.

```
int     x, y;   // these are global shared variables
mutex   s;      // this is the semaphore to protect them
```

```
task    reader() {
int     i;              // these are local variables
float   d, v[DIM];

        ...
        wait(s);
        d = sqrt(x*x + y*y);
        for (i=0; i++; i<DIM) {
             v[i] = i*(x + y);
             if (v[i] < x*y) v[i] = x + y;
        }
        signal(s);
        ...
}
```

Can we shorten this critical section?

critical section length

Hints: shorten critical sections

A possibility is to **copy global variables** into local variables:

```
task    reader() {
int     i;                    // these are local variables
float   d, v[DIM];
float   a, b;                 // two new local variables
        ...
        wait(s);              // copy global vars
        a = x; b = y;         // to local vars
        signal(s);
        d = sqrt(a*a + b*b);       // make computation
        for (i=0; i++; i<DIM) {    // using local vars
              v[i] = i*(a + b);
              if (v[i] < a*b) v[i\] = a + b;
        }
        wait(s);              // copy local vars
        x = a; y = b;         // to global vars
        signal(s);
        ...
}
```

critical section length

critical section length

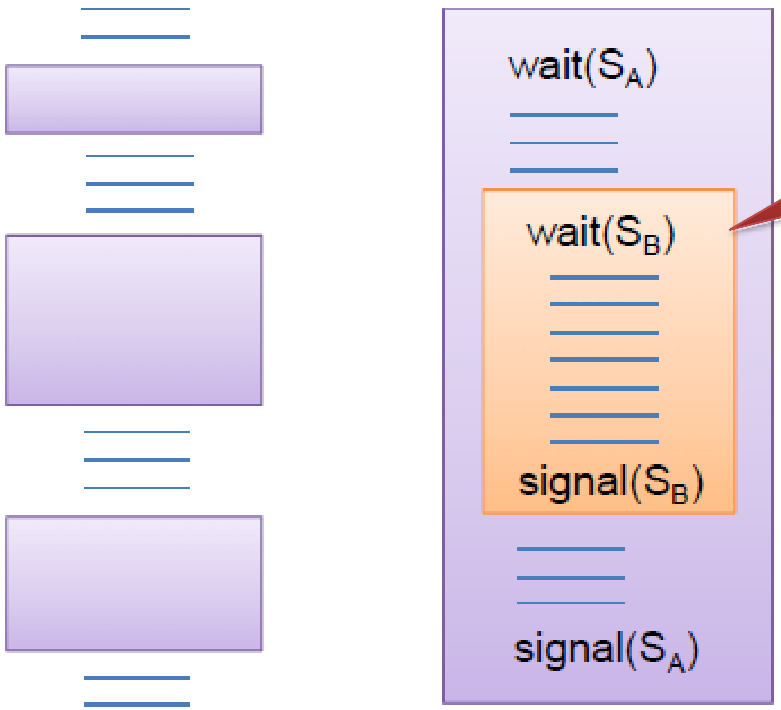
Hints: avoid critical sections across loops or conditions

What can go wrong here?

```
...
wait(s);
results = x + y;
while (result > 0){
      v[i] = i*(x + y);
      if (v[i] < x*y)
            results = results - y;
      else
            signal(s);
}
...
```

this code is very UNSAFE since “signal” could never be executed, and τ_1 could be blocked forever!

Hints: avoid nested critical sections



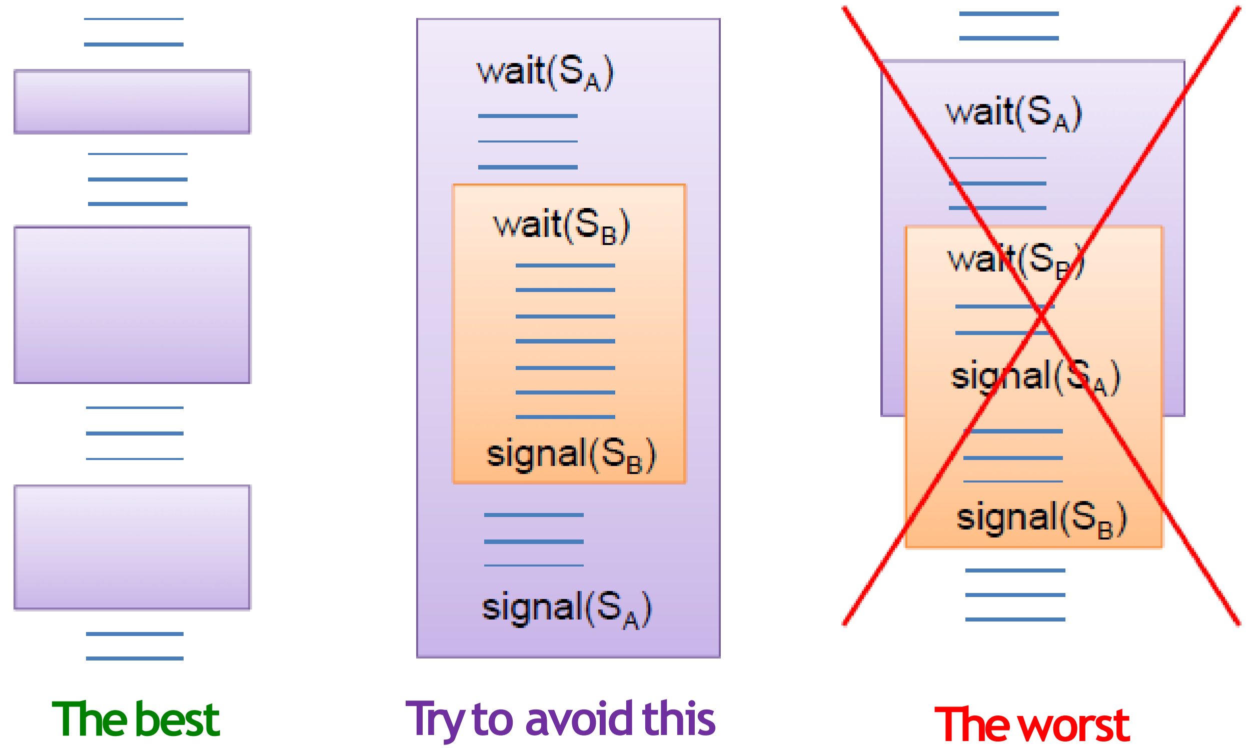
Why is it not advised?

Because to reach the inner critical section the task must acquire 2 locks: S_A and S_B.

While the task holds the first lock S_A and waits for the second one S_B, no other task can access the first lock S_A!

Hints: avoid cross-cutting critical sections

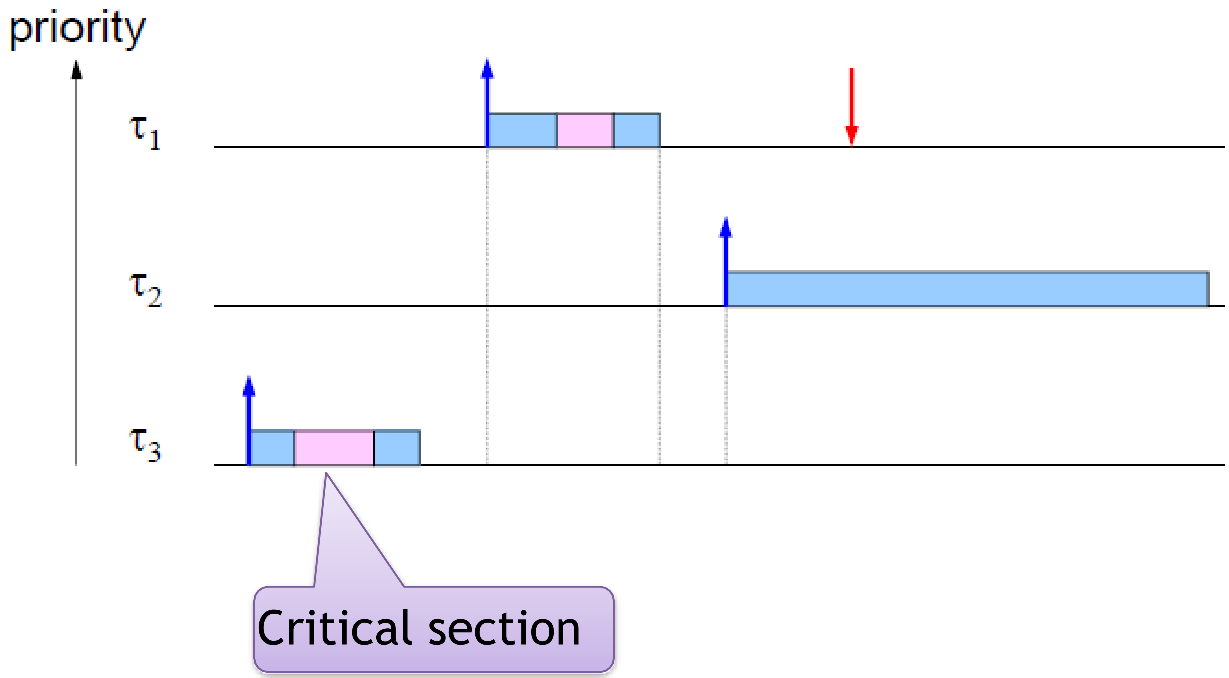
- Make critical sections as short as possible.
- Avoid making critical sections across loops or conditional statements.
- Try to avoid nested critical sections.
- If nested critical sections are unavoidable, at least avoid cross-cutting critical sections.
  - Because it makes the analysis very hard



Do these guidelines solve the “blocking” problem?

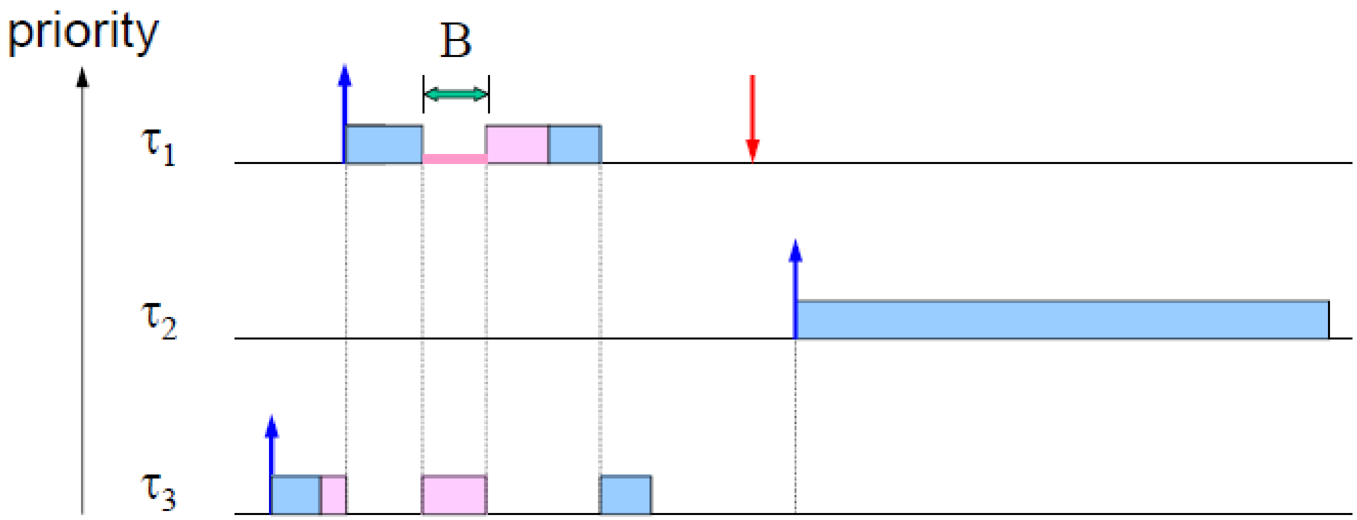
Impact on schedulability

Schedule with no conflicts



Impact on schedulability

Conflict on a critical section

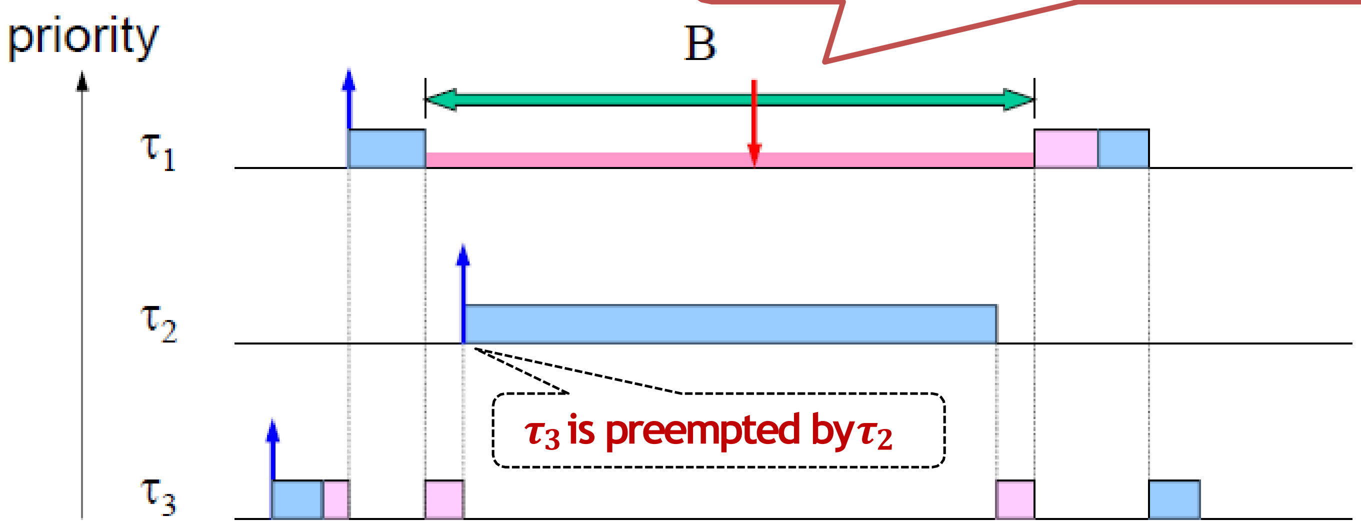


In this case, it is a direct blocking from the low-priority task to the high-priority task

Impact on schedulability

Conflict on a critical section

τ_1 is **"indirectly" blocked** by τ_2.
Indirect blocking happens when a low priority task blocks a high- priority one and they do not share any resource



This situation is called "**priority inversion**":
A high-priority task is blocked by a lower-priority task

Solution

Introduce a **concurrency control protocol** for accessing critical sections.

Key aspects in designing an access protocol

Access Rule:

Determines when to block or whether to block or not.

Example: if a task is in a critical section, then block the task that has just arrived

Progress Rule:

Determines how to execute inside a critical section.

Example: inside critical section, execute non-preemptively

Release Rule:

Determines how to order the pending requests of the blocked tasks.

Example: At exit, enable preemption

Resource access protocols

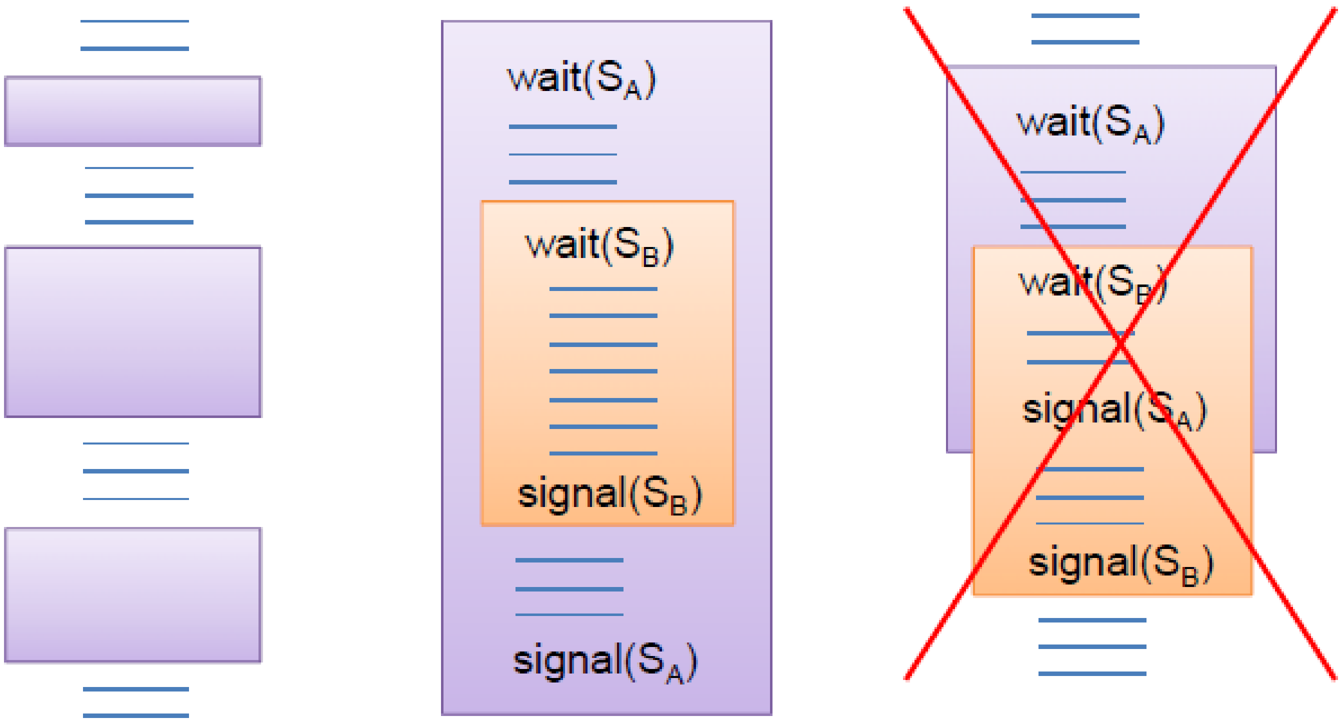
- Classical semaphores (No protocol)
- Non-Preemptive Protocol (**NPP**)
- Highest-Locker Priority (**HLP**)
- Priority Inheritance Protocol (**PIP**)
- Priority Ceiling Protocol (**PCP**)
- Stack Resource Policy (**SRP**)

  (will not be covered in the exam)

There will certainly be some exam questions from these protocols

Assumption

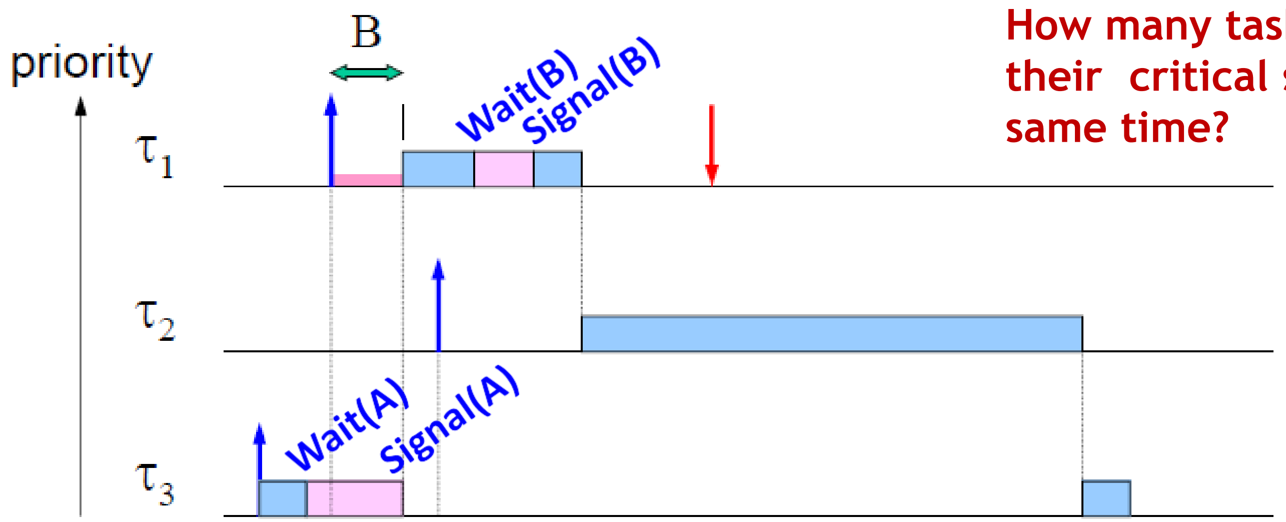
Critical sections are correctly accessed by tasks:



Non-preemptive protocol (NPP)

High-level idea

Whenever a task accesses a resource, it enters a non-preemptive mode until it releases the resource.



How many tasks can be in their critical sections at the same time?

Only one task!

- Access Rule: A task never blocks at the entrance of a critical section, but at its activation time.
- Progress Rule: Disable preemption when executing inside a critical section.
- Release Rule: At exit, enable preemption so that the resource is assigned to the pending task with the highest priority.

NPP: implementation notes

A possible method to implement NPP protocol:

- Each task τ_i must have two priorities:
  - a nominal priority P_i (fixed) assigned by the application developer;
  - a dynamic priority p_i (initialized to P_i) used to schedule the task and affected by the protocol.

- Then, the protocol can be implemented by changing the behavior of the wait and signal primitives:

wait(s): $p_i = \min\{P_1, \ldots, P_n\}$

signal(s): $p_i = P_i$

NPP: pro & cons

ADVANTAGES: simplicity and efficiency.

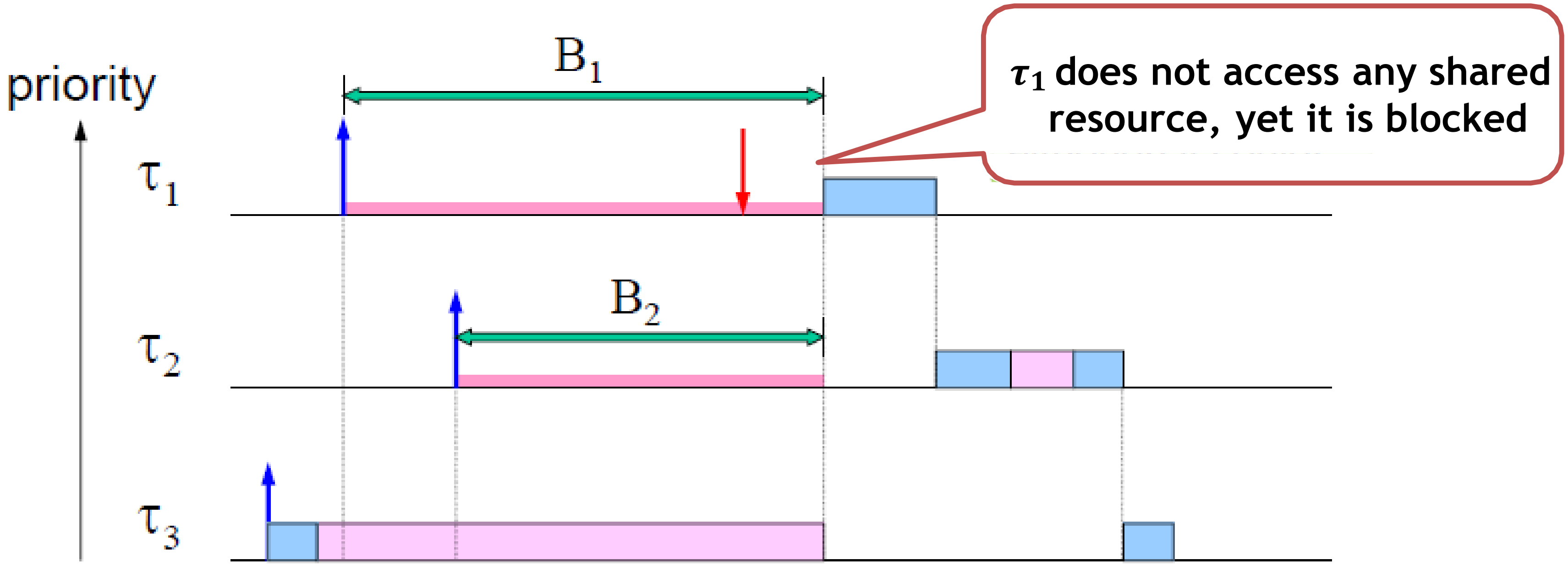
- Semaphore queues are not needed, because tasks never block on a wait(s).
- Each task can block at most on a single critical section.
- It **prevents deadlocks** and **allows stack sharing**.
- It is transparent to the programmer.

PROBLEMS:

1. Tasks may be blocked **even if they do not use any shared resource**.
2. Long critical sections delay all high-priority tasks
3. A task could be blocked even if it “may” not access a critical section

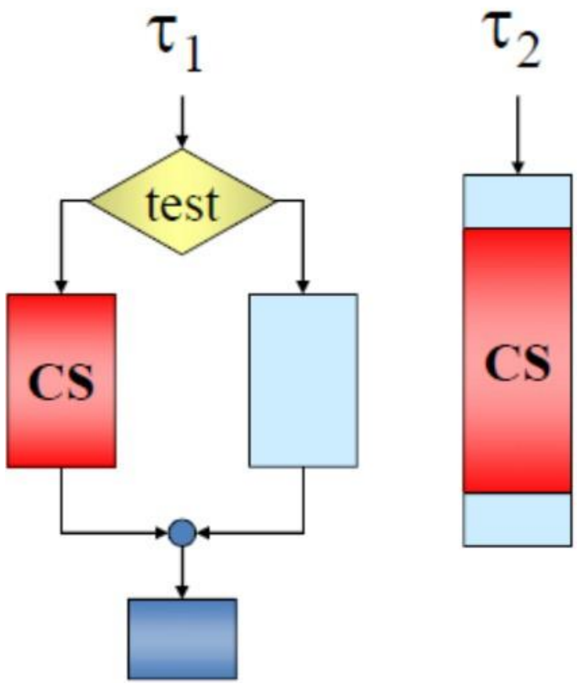
NPP: problems

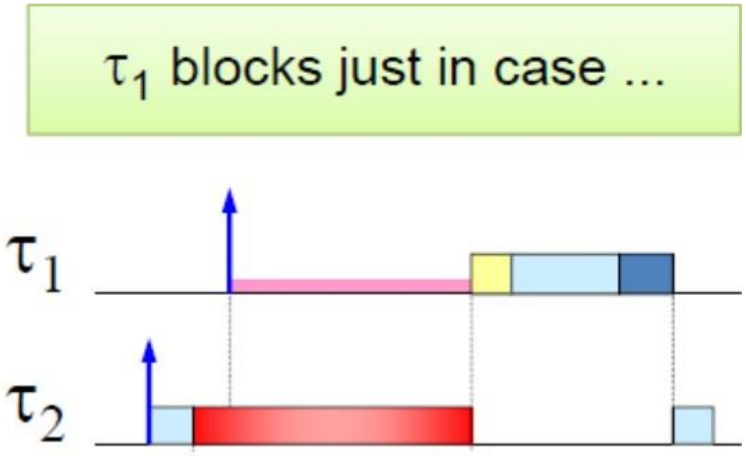
- Tasks may be blocked even if they do not use any shared resource.
- Long critical sections delay all high priority tasks



NPP: problems

A task could be blocked even if it "**may**" not access a critical section

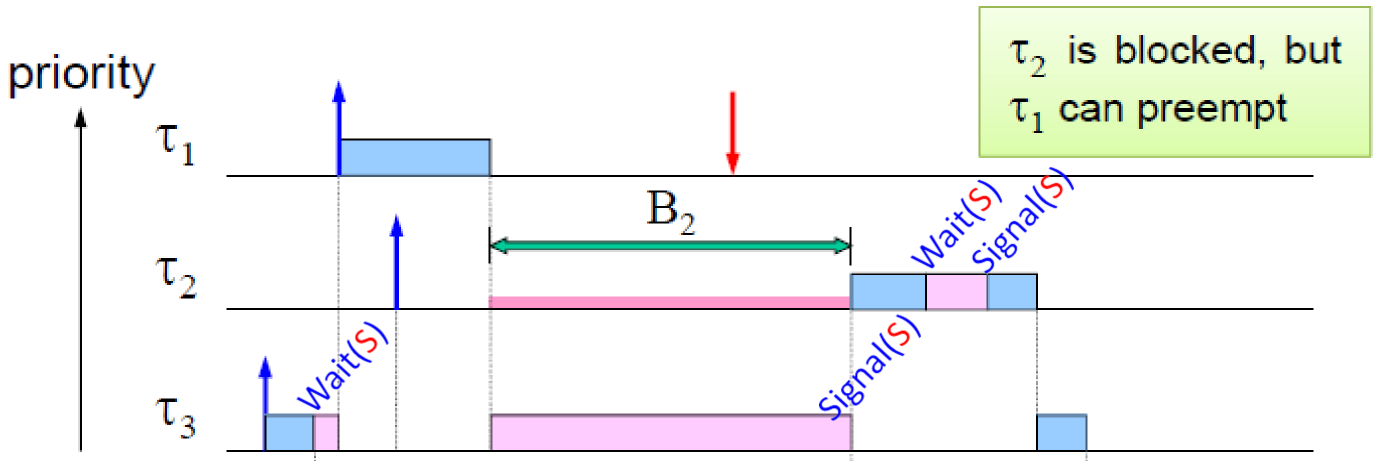




Highest-locker priority (HLP) protocol

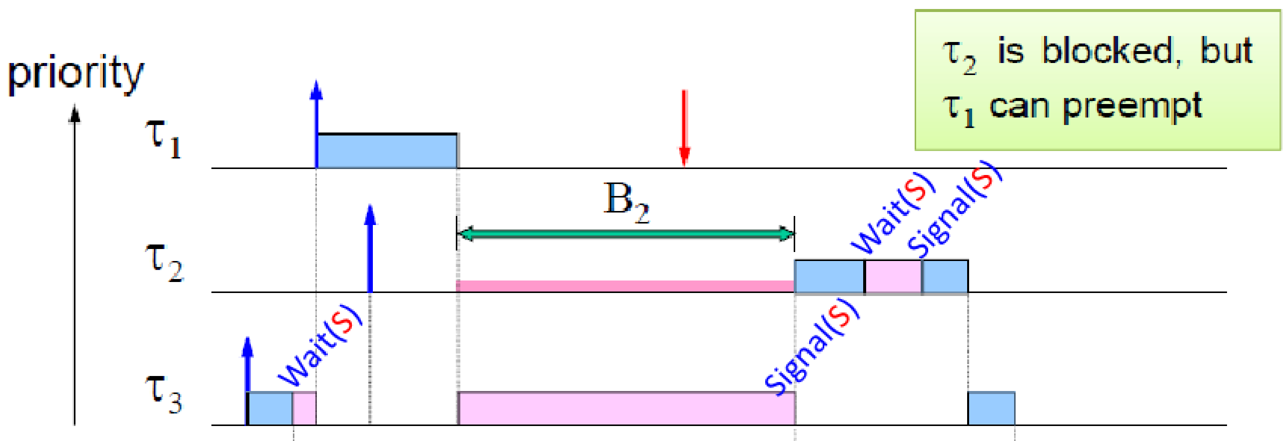
High-level idea

When a task accesses a resource (e.g., wait(S)), its priority upgrades to the priority of the highest-priority task that <u>may use</u> the resource *S*



- Access Rule: A task never blocks at the entrance of a critical section, but at <u>its activation time</u>.
- Progress Rule: Inside the critical section for resource R, the task executes at the highest priority of the <u>tasks that use</u> R.
- Release Rule: At exit, the dynamic priority of the task is reset to its nominal priority P_i .

Highest-locker priority (HLP) protocol



Priority assigned to τ_i when it uses semaphore S:

$$p_i(S) = \min\{P_j \mid \forall \tau_j, \tau_j \text{ uses } S\}$$

What is $p_3(S)$?

It is 2 because τ_2 is the highest-priority task that uses *S*

HLP: implementation notes

- Each task τ_i is assigned a nominal priority P_i and a dynamic priority p_i.
- Each semaphore S is assigned a resource ceiling *C(S)*:

$$C(S) = \min\{P_j \mid \forall \tau_j, \tau_j \text{ uses } S\}$$

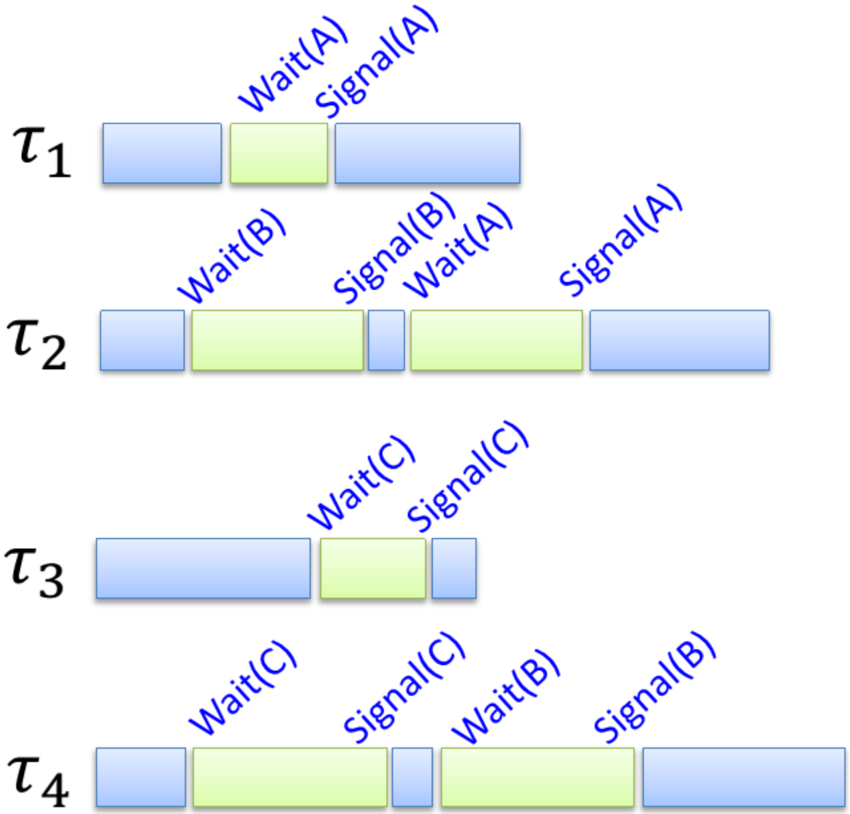
Change the wait and signal primitives as follows:

Wait (S): $p_i = C(S)$
Signal (S): $p_i = P_i$

Note: **HLP** is also known as Immediate-Priority Ceiling (**IPC**).

Exam examples

$p_i(S) \ = \min\{P_j \mid \forall \tau_j, \tau_j \text{ uses } S\}$



Consider HLP protocol:

What is $p_1(A)$? 1

What is $p_2(A)$? 1

What is $p_2(B)$? 2

What is $p_3(C)$? 3

What is $p_4(C)$? 3

What is $p_4(B)$? 2

Reminder: $P_1 < P_2 < P_3 < \cdots < P_n$

HLP: pro & cons

ADVANTAGES: simplicity and efficiency.

- Semaphores queues are not needed, because tasks never block on a wait(s).
- Each task can block at most on a single critical section.
- It **prevents deadlocks** and allows stack sharing.

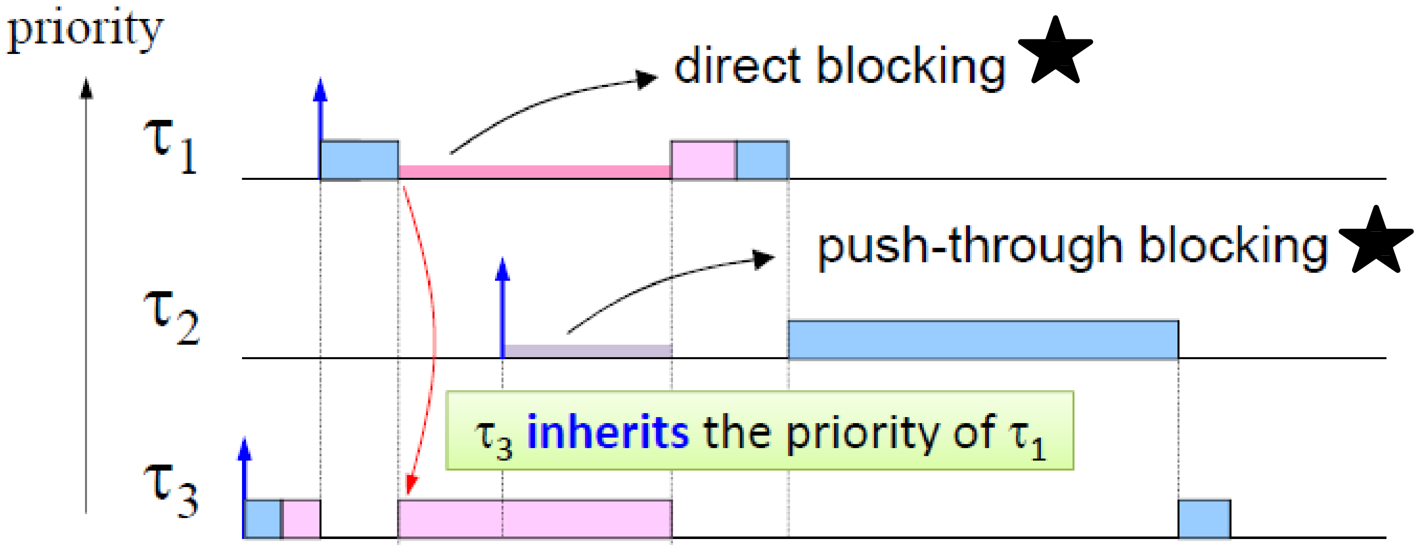
PROBLEMS:

1. A task could be blocked even if it "may" not access a critical section (similar to NPP).
2. It is not transparent to programmers (due to ceilings).

Priority-inheritance protocol (PIP)

High-level idea

Whenever a task accesses a resource *S* that is locked by another task, the priority of the locking task upgrades to the priority of the highest-priority task that is currently blocked on resource *S*.



- Access Rule: A task blocks at the entrance of a critical section if the resource is locked.
- Progress Rule: Inside resource R, a task executes with the highest priority of the tasks blocked on R.
- Release Rule: At exit, the dynamic priority of the task is reset to its nominal priority P_i.

PIP: types of blocking

- **Direct blocking**
  - A task blocks on a locked semaphore
- **Indirect blocking (push-through blocking)**
  - A task is blocked because a lower-priority task inherited a higher priority.

Blocking:
a delay caused by lower-priority tasks

PIP: implementation notes

- Inside a resource S the dynamic priority p_i is set to

$$p_i(S) = \min\{P_j \mid \tau_j \text{ is blocked on } S\}$$

wait (S)

```
if (S == 0) {
    <suspend the calling task τ_c in the semaphore queue>
    <find the task τ_k that is locking the semaphore S>
    p_k = min{P_c, p_k}      // τ_k inherits the priority of τ_c if p_k > P_c
    <call the scheduler>
}
else S = 0;
```

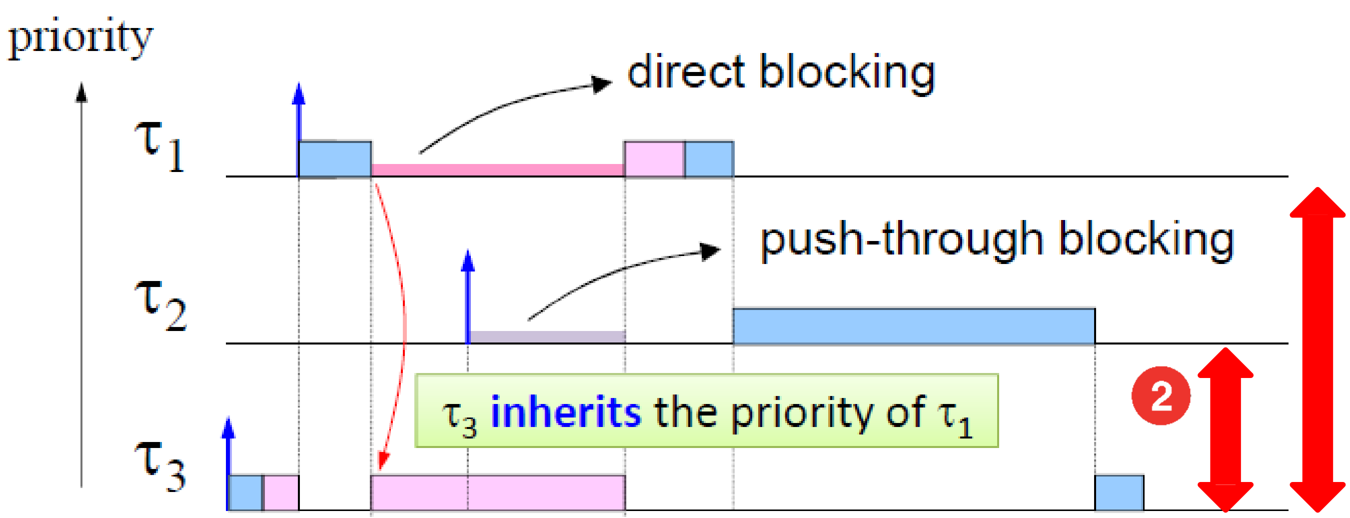
signal (S)

```
if (there are blocked tasks) {
    <awake the highest-priority task in the semaphore queue>
    p_i = P_i
    <call the scheduler>
    }
else S = 1;
```

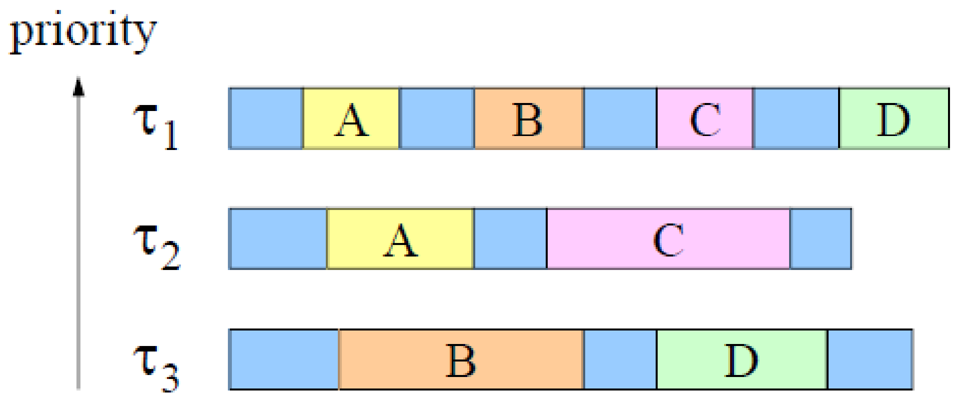
Identifying blocking resources

Under PIP, a task τ_i can be **blocked** on a semaphore S_k **only if**:

1. S_k is directly shared between τ_i and lower-priority tasks (direct blocking), or
2. S_k is shared between tasks with priority lower than τ_i and tasks having priority higher than τ_i (push-through blocking).



PIP: example 2

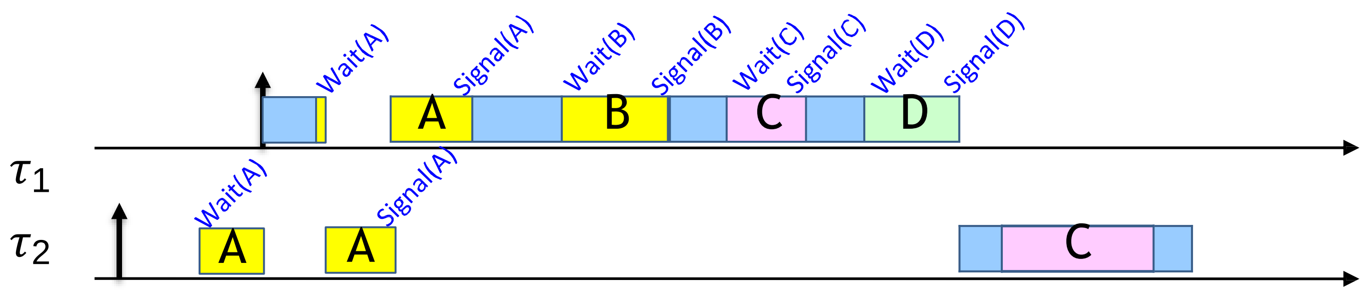


Assume that any release scenario is possible

Which tasks can block τ_1? τ_2 (on A2 or C2) and τ_3 (on B3 or D3)

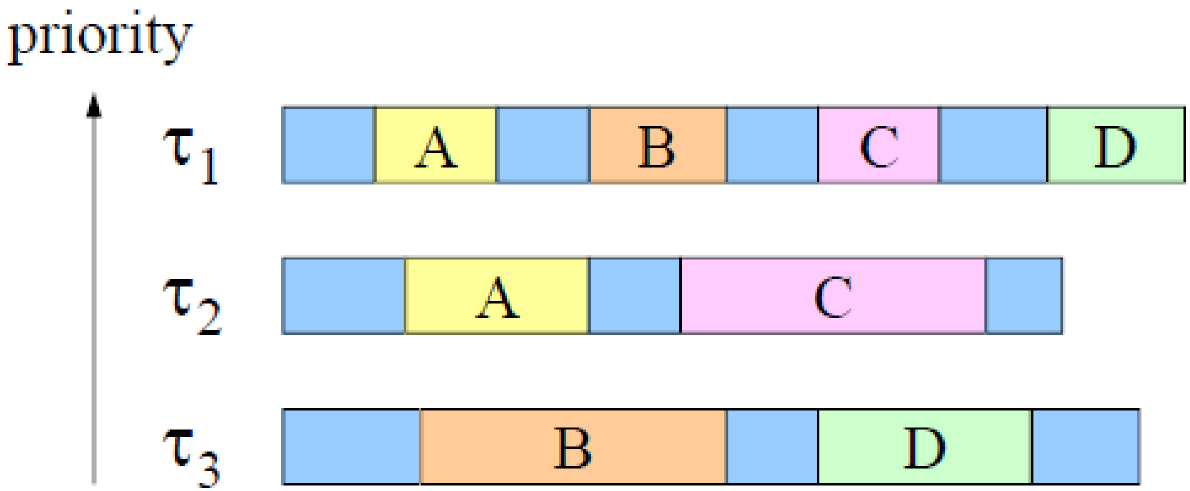
How many time any low-priority task can block a high-priority task?

ONLY once! Because right after the end of the critical section of that low-priority task, the high-priority task starts its execution and then no other low-priority task can preempt the high-priority one.



Notation guide:
B2 = the access of task 2 to resource B

PIP: example 2

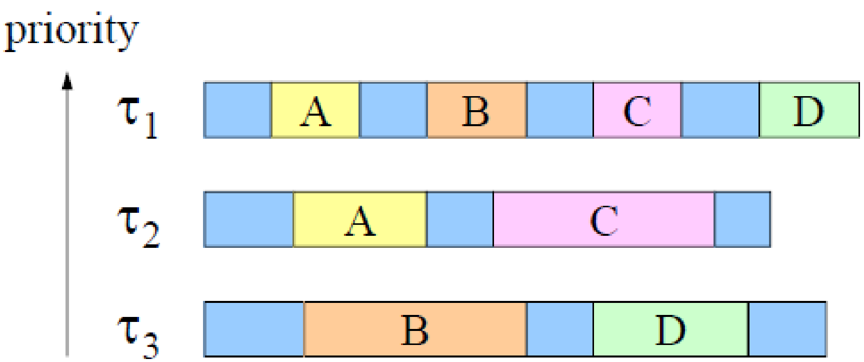


Which tasks can block τ_3? τ_3 cannot be blocked

Which tasks can block τ_2? τ_3 (on B3 or D3)

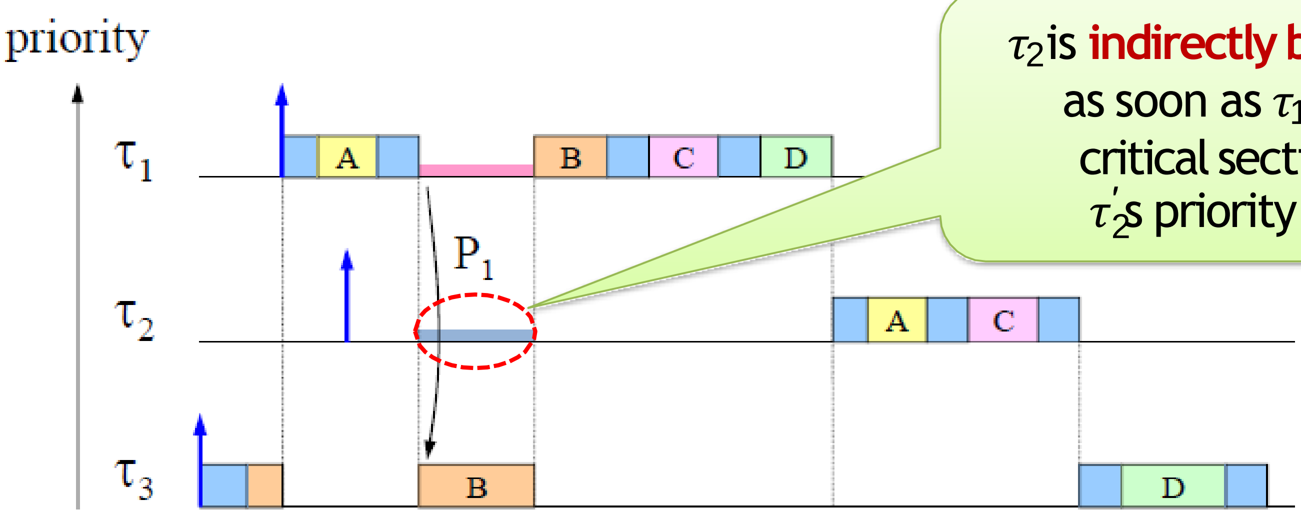
Exam example

Given the following resource accesses protected by semaphores A, B, C, and D,



Part 1. Is it possible that under the PIP protocol, τ_3 **directly** blocks τ_2 on any resource?

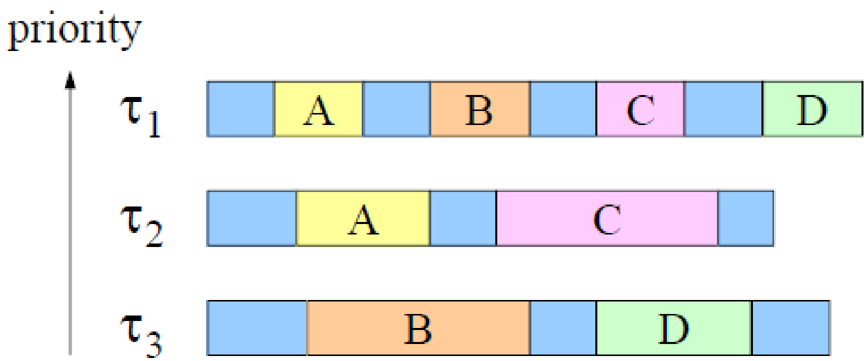
Part 2. Generate an **execution scenario** in which τ_2 is **indirectly blocked** by τ_3 on the access of τ_3 to resource B.



τ_2 is **indirectly blocked** by τ_3 because as soon as τ_1 wants to enter its critical section on resource B, τ_2's priority is upgraded to 1

Exam example (Quiz!)

Under PIP, generate an **execution scenario** in which τ_2 is **indirectly blocked** by τ_3 on the access of τ_3 to resource D.



Identifying blocking resources

Lemma 1: A task τ_i can be blocked at most once by a lower priority task.

If there are n_i tasks with priority lower than τ_i, then τ_i can be blocked at most at most n_i times, independently of the number of critical sections that can block τ_i.

Identifying blocking resources

Lemma 2: A task τ_i can be blocked at most once on a semaphore S_k.

↓

If there are m_i distinct semaphores that can block a task τ_i, then τ_i can be blocked at most m_i times, independently of the number of critical sections that can block τ_i.

Bounding blocking times

Theorem:

τ_i can be blocked at most for $\alpha_i = \min(n_i, m_i)$ critical sections.

n_i = number of tasks with priority less than τ_i

m_i = number of semaphores that can block τ_i (either directly or indirectly).

PIP: pro & cons

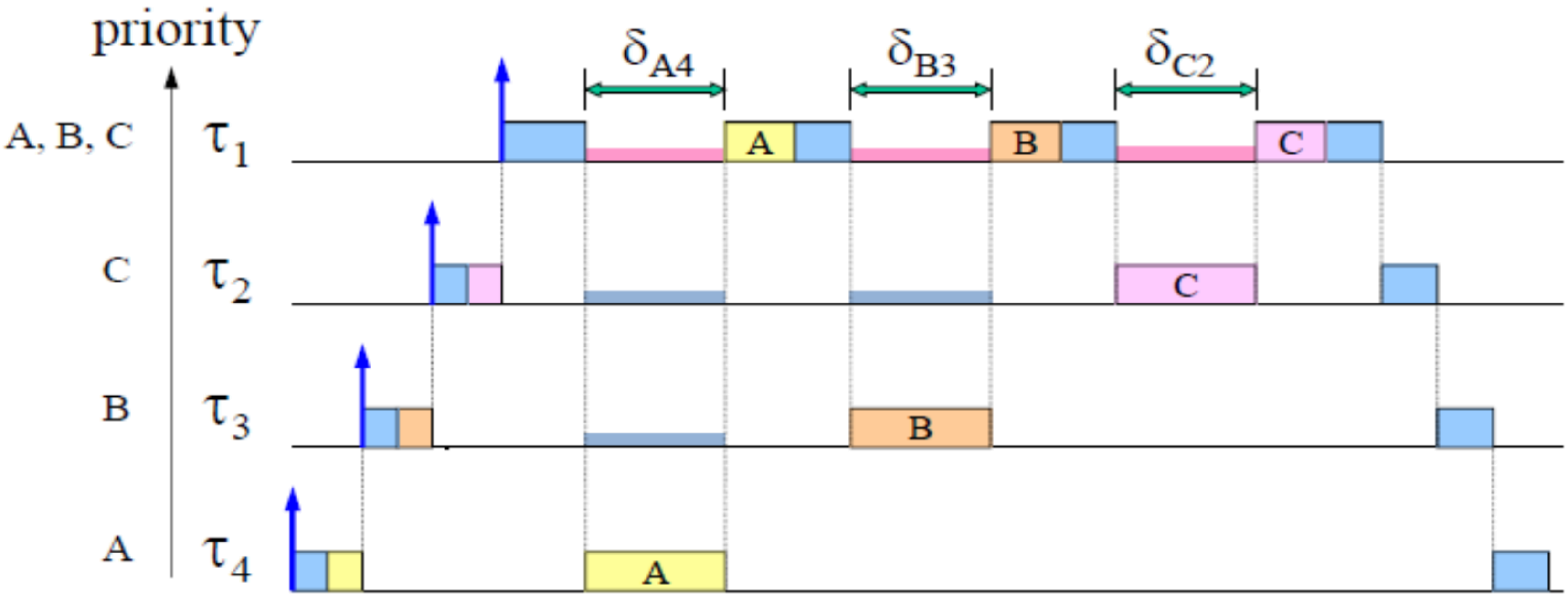
ADVANTAGES:

- It removes the pessimisms of NPP and HLP **(a task is blocked only when really needed).**
- It is transparent to the programmer.

PROBLEMS:

1. More complex to implement (especially to support nested critical sections).
2. It is prone to chained blocking.
3. It does not avoid **deadlocks**. ★

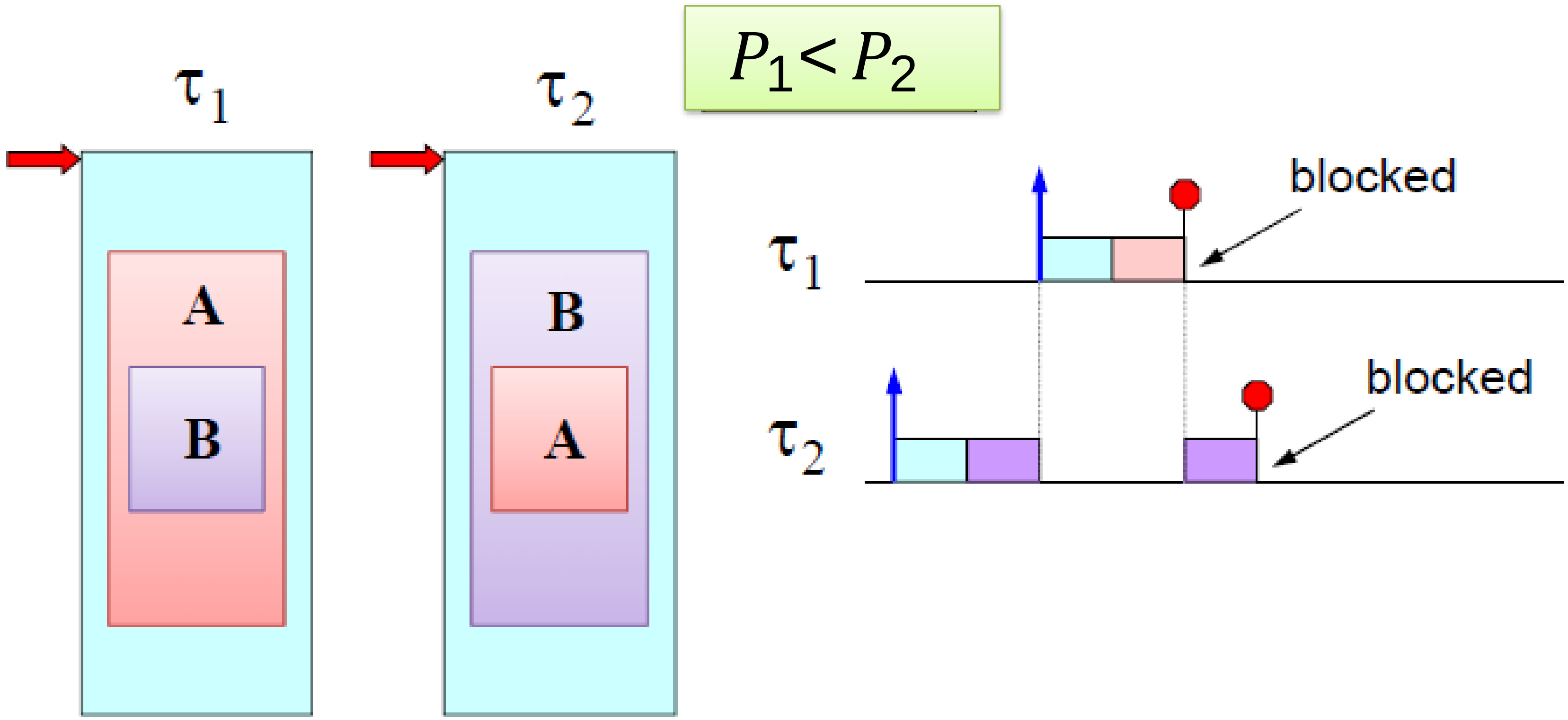
PIP: Chained blocking problem



NOTE: τ_1 can be blocked at most once for each lower priority task.

Typical deadlock

- It can only occur with nested critical sections:



Agenda

- Classical semaphores (No protocol)
- Non-Preemptive Protocol (**NPP**)
- Highest-Locker Priority (**HLP**)
- Priority Inheritance Protocol (**PIP**)
- Priority Ceiling Protocol (**PCP**)
- Stack Resource Policy (**SRP**)

  (will not be covered in the exam)

There will certainly be some exam questions from these protocols

Priority Ceiling Protocol (PCP)

High-level idea

A task can access a resource only if it passes the PCP access test.

If the test is passed, the rest is like PIP protocol.

PCP can be viewed as **PIP** + access test

- Access Rule: A task can access a resource only if it passes the PCP access test.
- Progress Rule: Inside resource R, a task executes with the highest priority of the tasks blocked on R.
- Release Rule: At exit, the dynamic priority of the task is reset to its nominal priority P_i .

PCP implementation

To keep track of resource usage by high-priority tasks, each resource is assigned a **resource ceiling**:

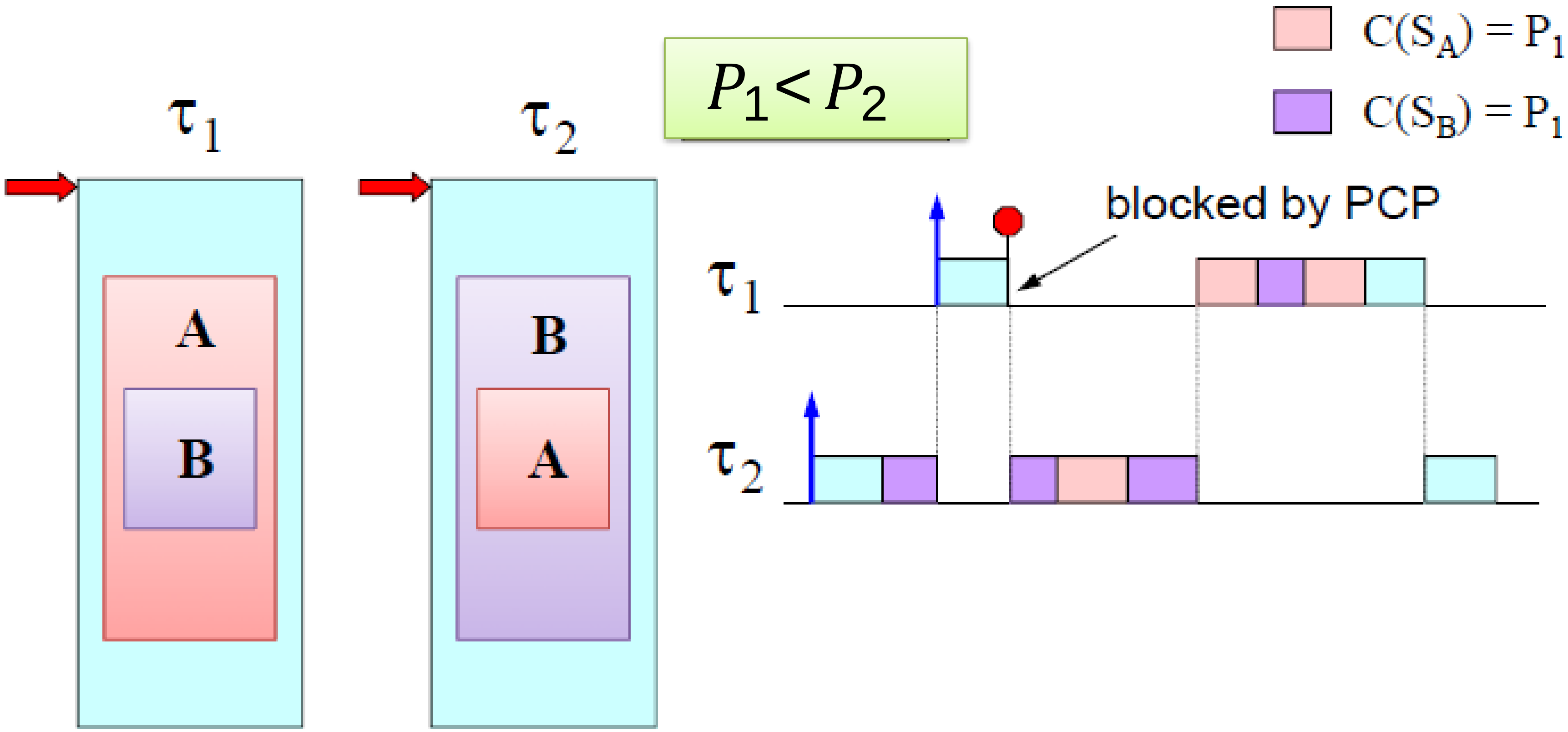
$$C(S_k) = \min\{P_j \mid \forall \tau_j, \tau_j \text{ uses } S_k\}$$

A task τ_i can enter a critical section **only if** its priority is higher than the maximum ceiling of the locked semaphores:

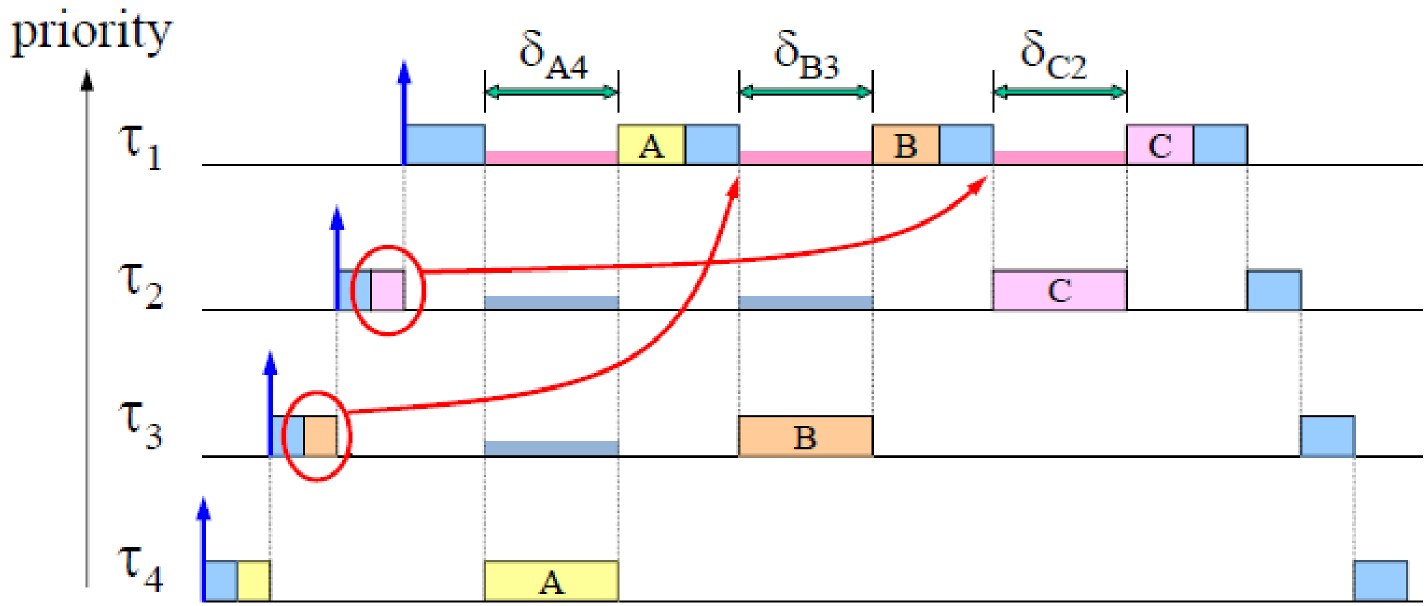
PCP access test:

$$P_i < \min\{C(S_k) \mid S_k \text{ locked by tasks } \neq \tau_i\}$$

PCP: deadlock avoidance



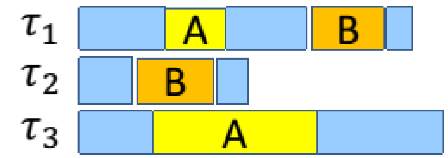
Avoiding chained blocking

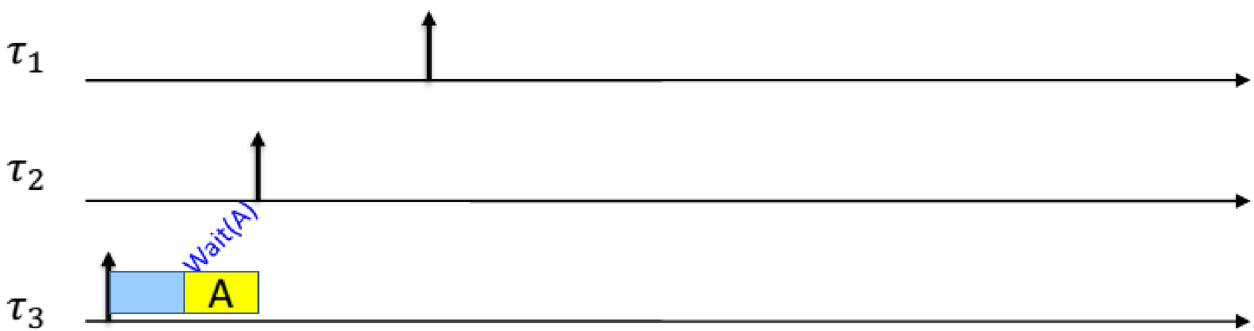


How can we avoid chained blocking?

To avoid multiple blocking of τ_1, **we must prevent τ_3 and τ_2 to enter their critical sections** (even if they are free), because a low priority task (τ_4) is holding a resource used by τ_1.

PCP: example





$P_3 < \min\{C(S_k) \mid S_k \in \{A, B\}$ and S_k locked by tasks $\neq \tau_3\}$

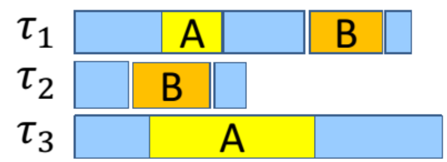
?

The test passes because no resource has been locked so far

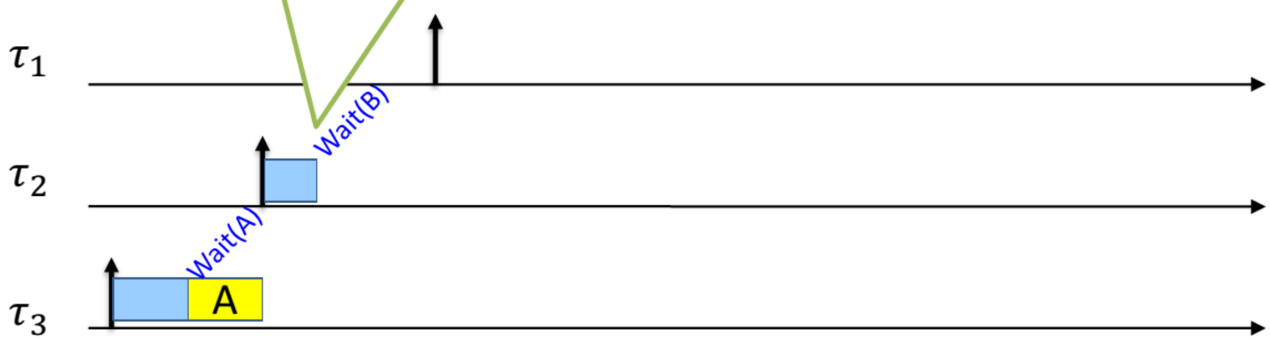
$C(A) = \min\{P_j \mid \forall \tau_j, \tau_j$ uses $A\}$ 1

$C(B) = \min\{P_j \mid \forall \tau_j, \tau_j$ uses $B\}$ 1

PCP: example



τ_2 does **not get the permission** to enter its critical section even though it was accessing a resource that was not locked



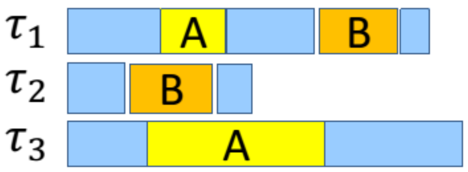
$$P_2 < \min\{C(S_k) \mid S_k \in \{A, B\} \text{ and } S_k \text{ locked by tasks } \neq \tau_2\}$$

?

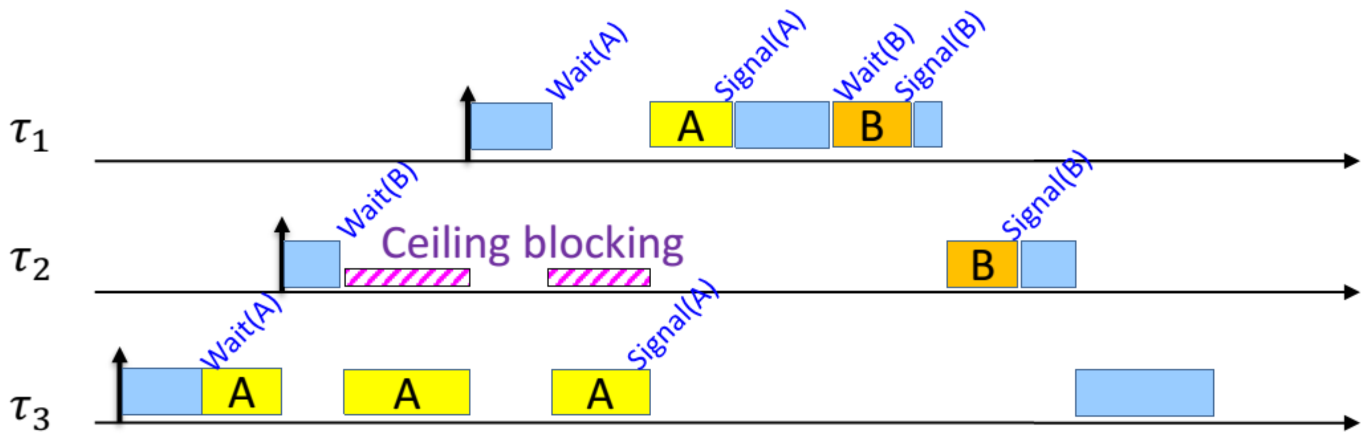
$C(A) = \min\{P_j \mid \forall \tau_j, \tau_j \text{ uses } A\}$ 1

$C(B) = \min\{P_j \mid \forall \tau_j, \tau_j \text{ uses } B\}$ 1

PCP: example

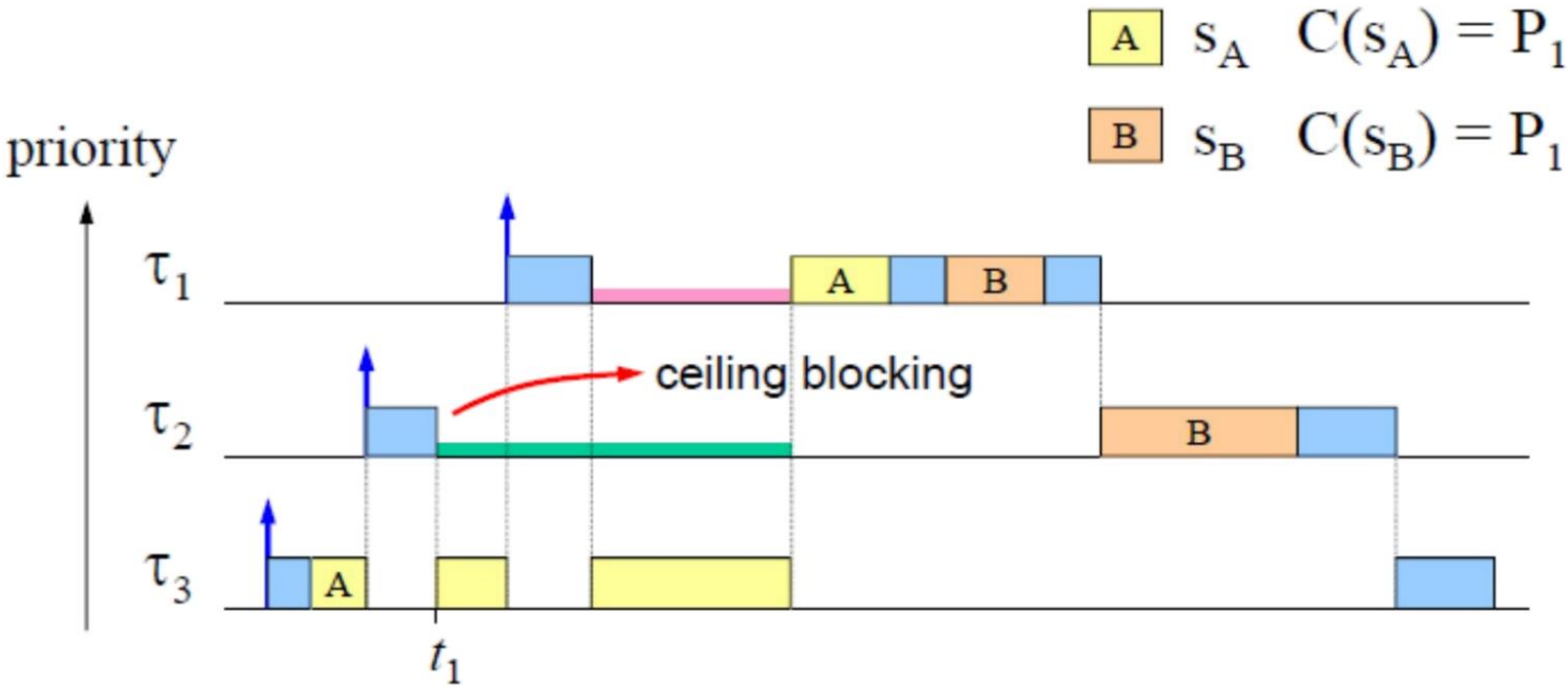


τ_1
A
B
τ_2
B
τ_3
A



τ_1
Wait(A)
Signal(A)
Wait(B)
Signal(B)
A
B
τ_2
Wait(B)
Ceiling blocking
Signal(B)
B
τ_3
Wait(A)
Signal(A)
A
A
A

PCP: another example



At t_1: τ_2 is blocked by the PCP, since $P_2 > C(S_A)$

PCP: properties

Theorem

Under PCP, a task can be blocked at most on a single critical section.

Theorem

PCP prevents chained blocking.

Theorem

PCP **prevents** deadlocks.

PCP: pro & cons

ADVANTAGES:

- It limits blocking to the length of a single critical section.
- It avoids deadlocks when using nested critical sections.

PROBLEMS:

1. More complex to implement (like PIP).
2. It can create **unnecessary blocking** (it is pessimistic like HLP).
3. It is **not transparent** to the programmer: resource ceilings must be specified in the source code.

Summary

protocol	Compatible with scheduling algorithm	pessimism	Blocking at	Transparent to user	Deadlock free	implementation
NPP	any	high	Arrival	yes	yes	easy
HLP	FP	medium	Arrival	no	yes	easy
PIP	FP	low	Resource access	yes	no	hard
PCP	FP	medium	Resource access	no	yes	harder