2IMN20 - Real-Time Systems

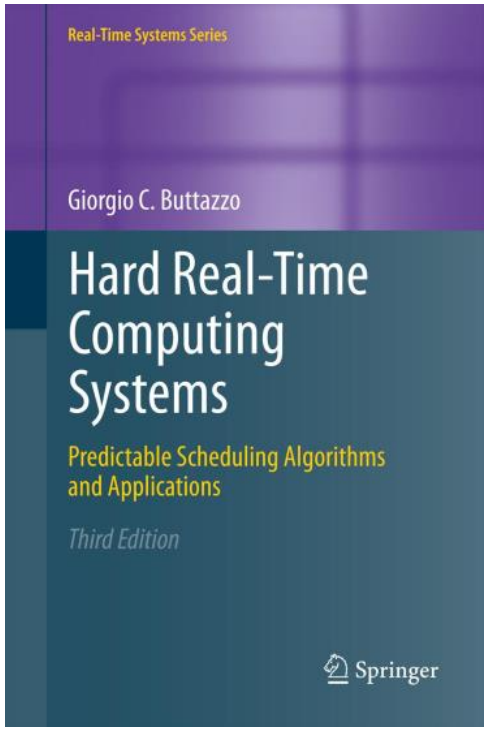
Shared resource access protocols

Nidhi Srinivasan

2023-2024

Reference book

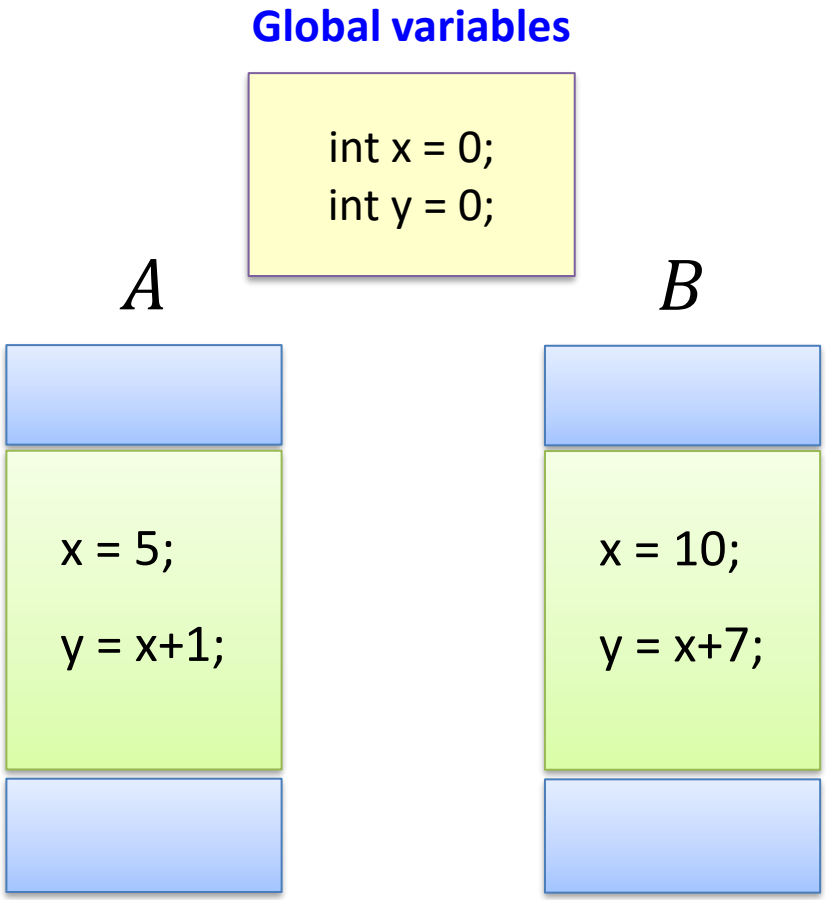
Chapter 7



Disclaimer:
Many slides were provided by Dr. Mitra Nasri

Some slides have been taken from Giorgio Buttazzo's course

Code predictability



Possible results after A:

- (x=5, y=6)

Possible results after B:

- (x=10, y=17)

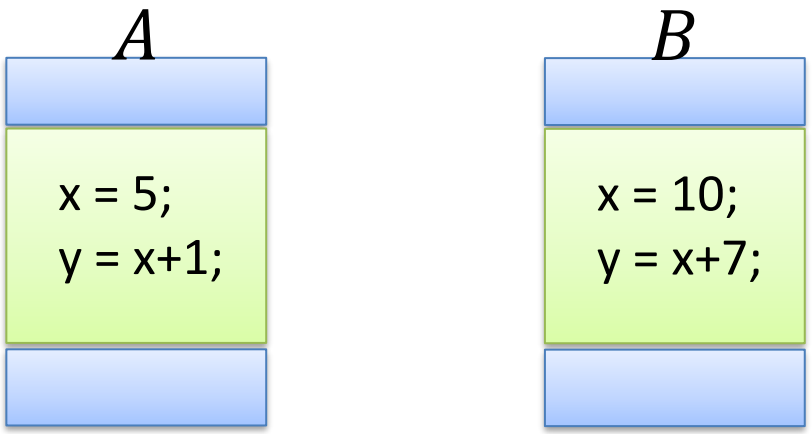
Possible results after A and B:

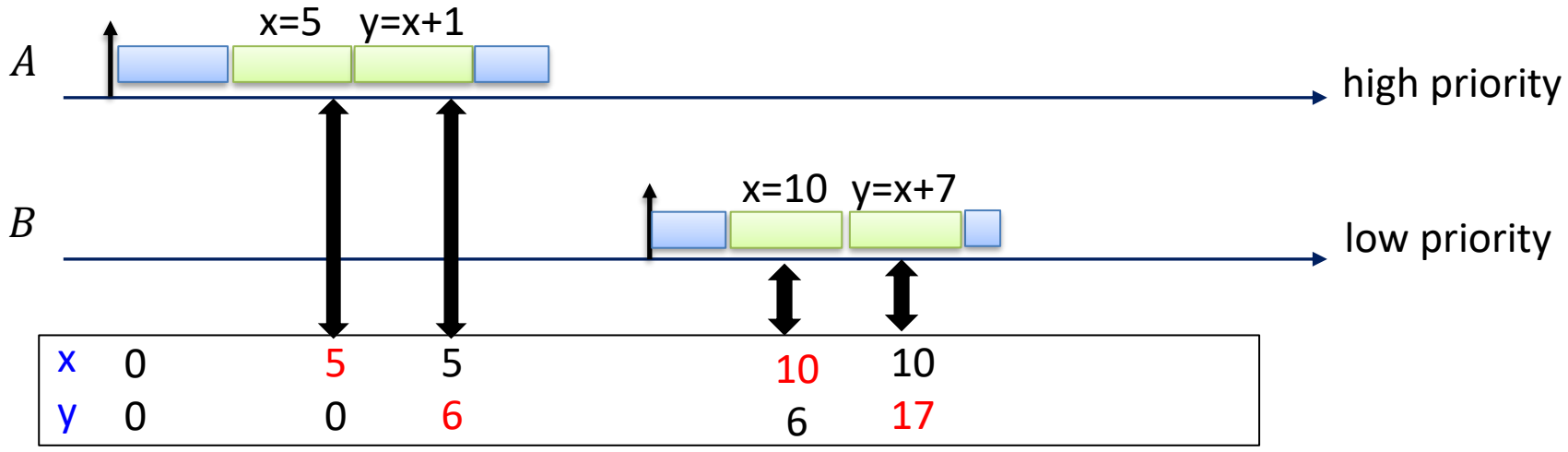
?

Execute A and B together

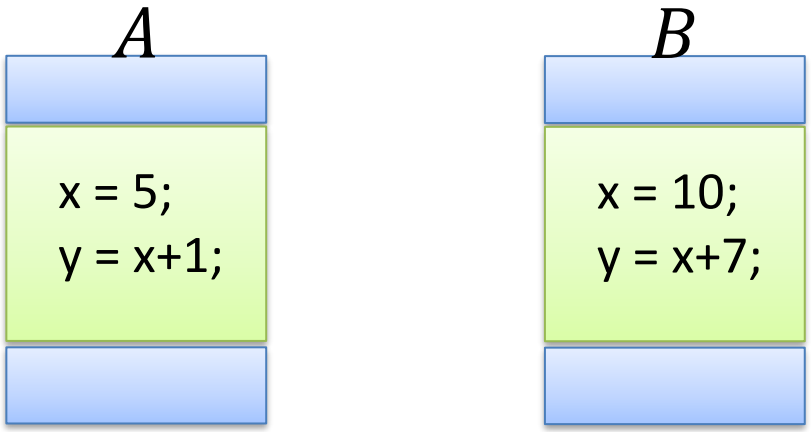
Possible results after A and B:

- (x=10, y=17)



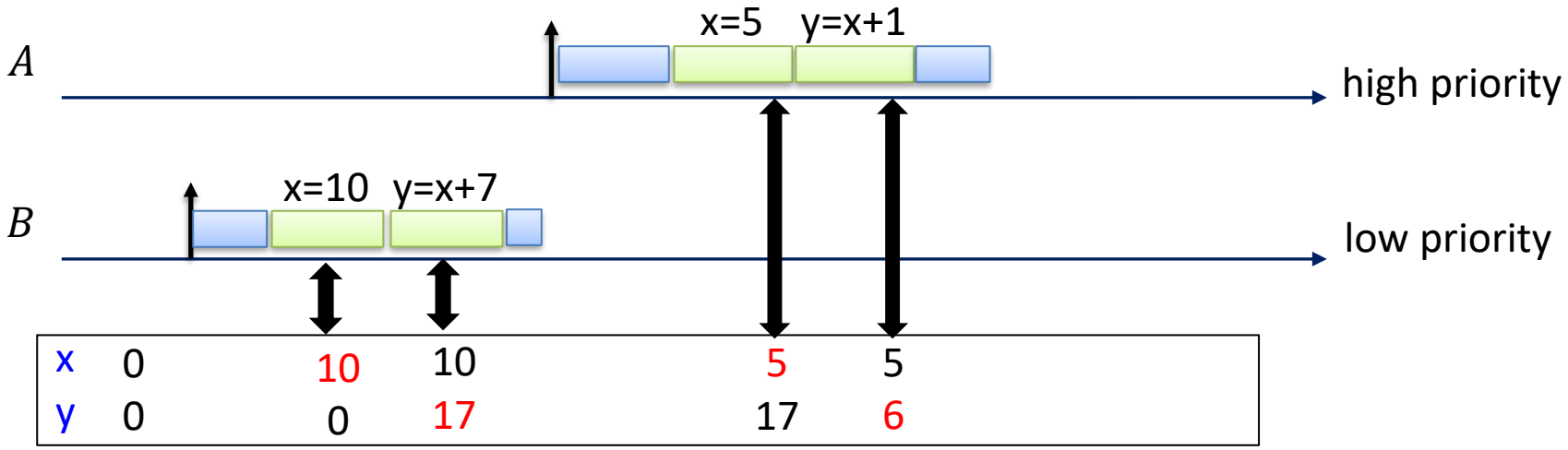


Execute A and B together

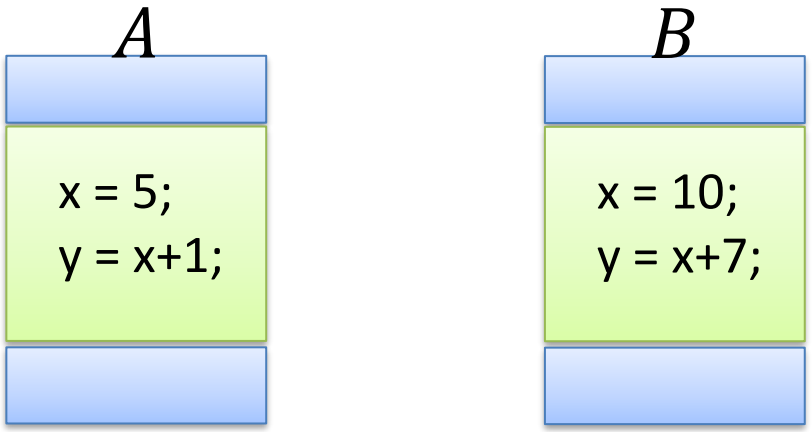


Possible results after A and B:

- (x=10, y=17)
- (x=5, y=6)

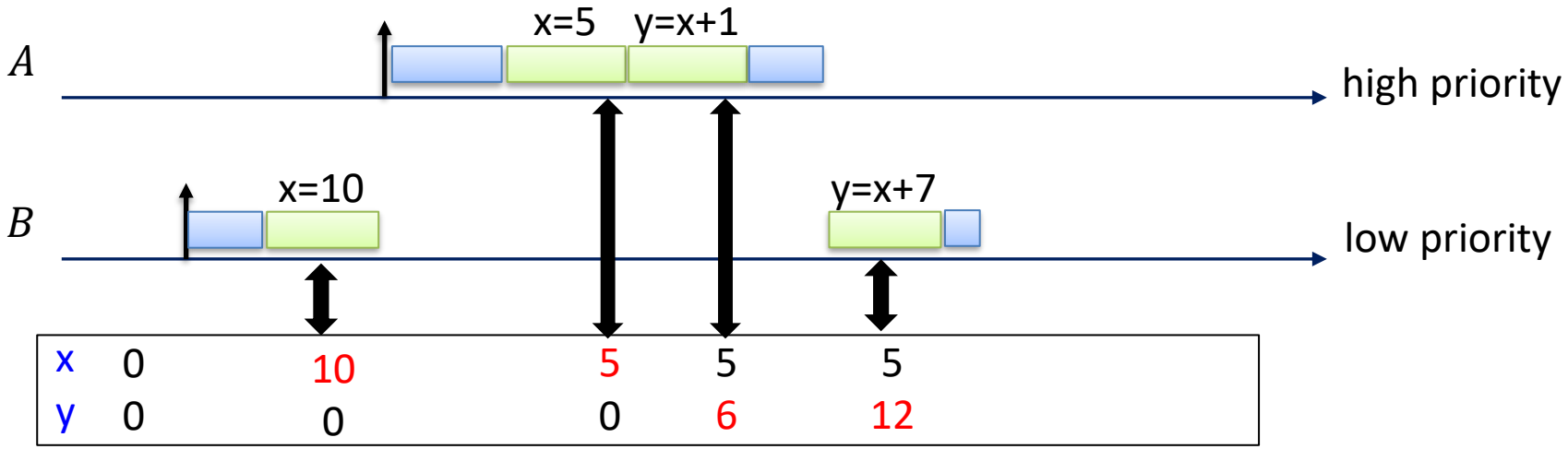


Execute A and B together

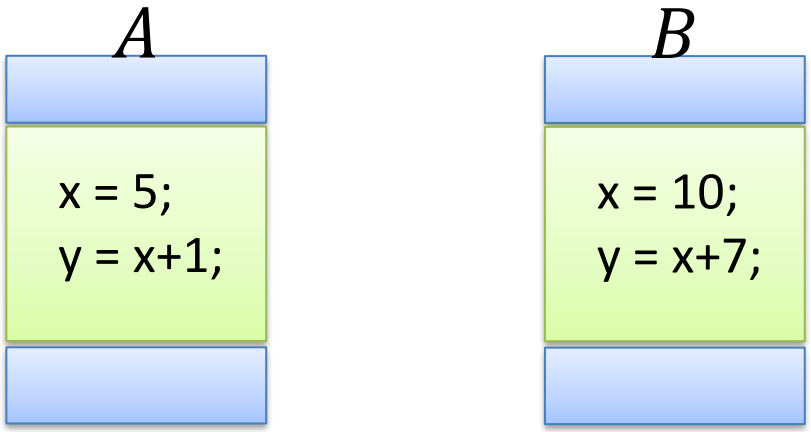


Possible results after A and B:

- (x=10, y=17)
- (x=5, y=6)
- (x=5, y=12)

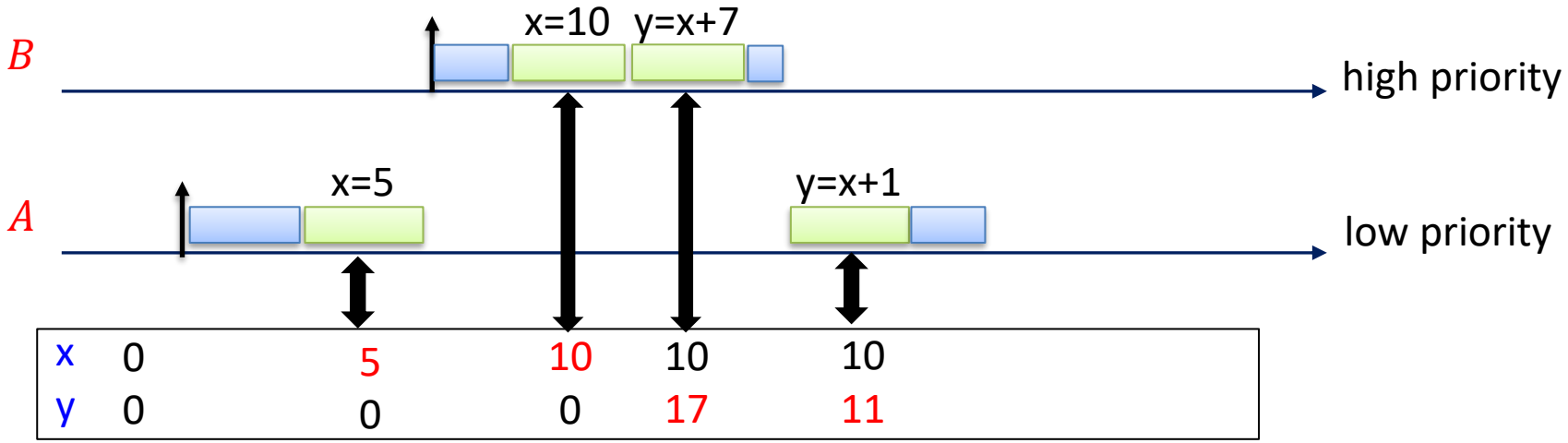


Execute A and B together

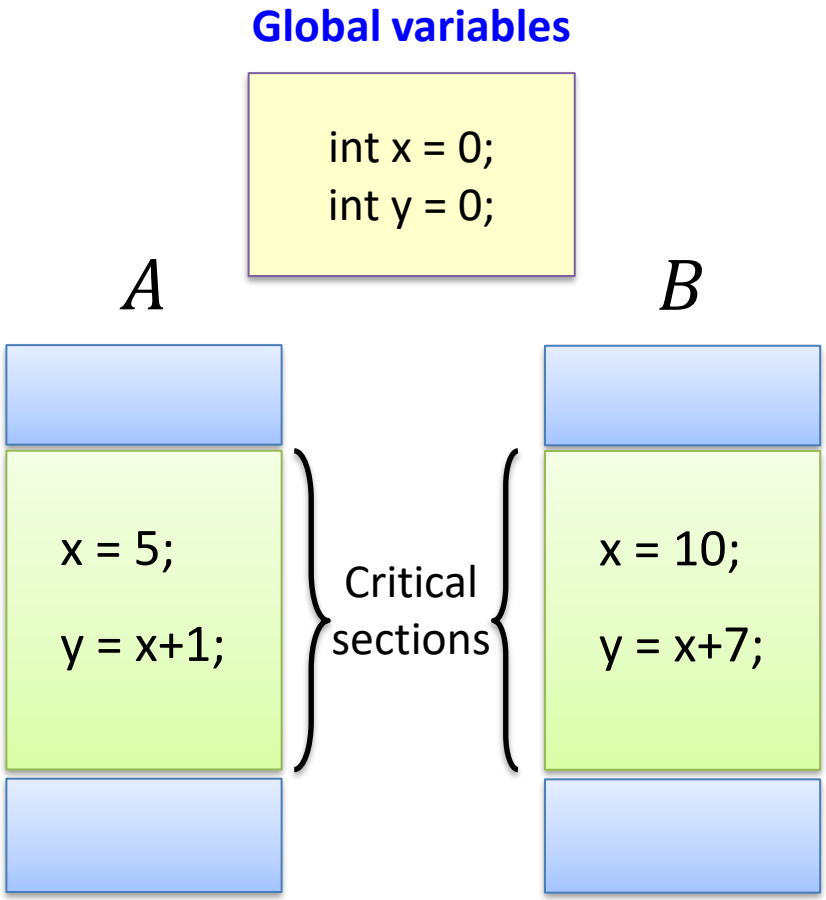


Possible results after A and B:

- (x=10, y=17)
- (x=5, y=6)
- (x=5, y=12)
- (x=10, y=11)



Code predictability



Possible results after A:

- (x=5, y=6)

Possible results after B:

- (x=10, y=17)

Possible results after A and B:

- (x=10, y=17)
- (x=5, y=6)
- (x=5, y=12) ← Only under preemptive scheduling!
- (x=10, y=11) ← Only under preemptive scheduling!

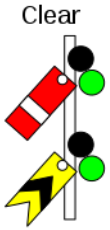
For preemptive scheduling we need to protect resources!

Typical choice: semaphores

- Here: Global variables are on a *shared resource*
- Each shared resource is protected by a different *semaphore*.
  - S = 1 ⇒ free resource
  - S = 0 ⇒ busy (locked) resource

wait(S):

if S == 0, **then**
The task must be blocked on a queue of the semaphore.
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.

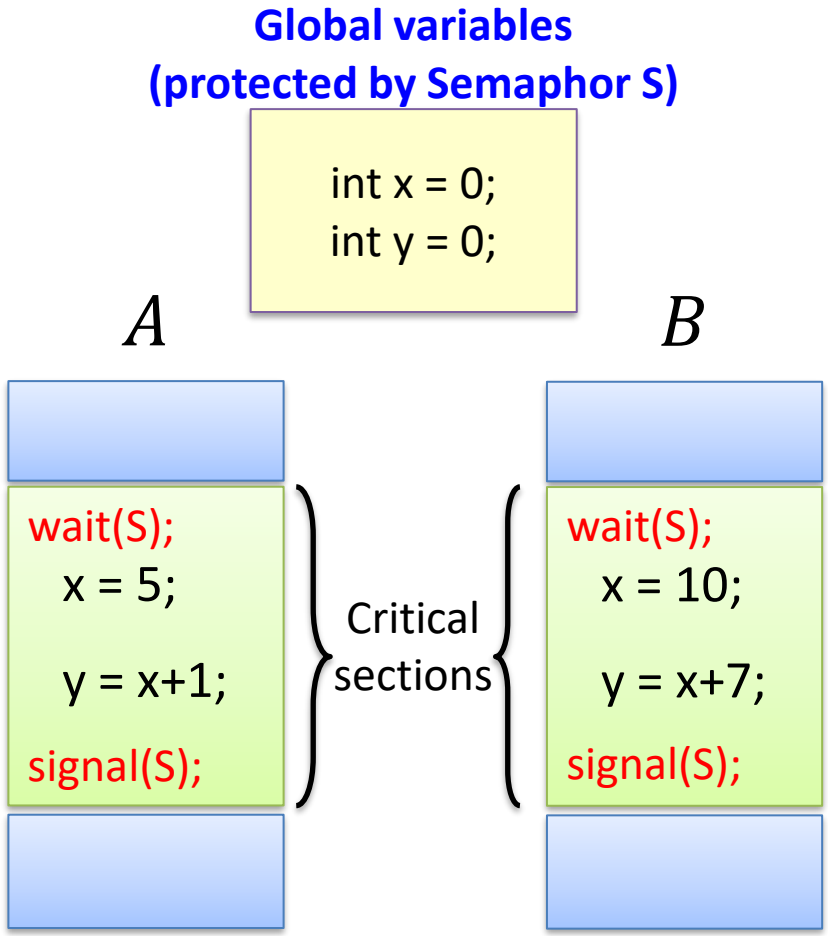
A

```
wait(S);
  x = 5;
  y = x+1;
signal(S);
```

B

```
wait(S);
  x = 10;
  y = x+7;
signal(S);
```

Code predictability



Possible results after A:

- (x=5, y=6)

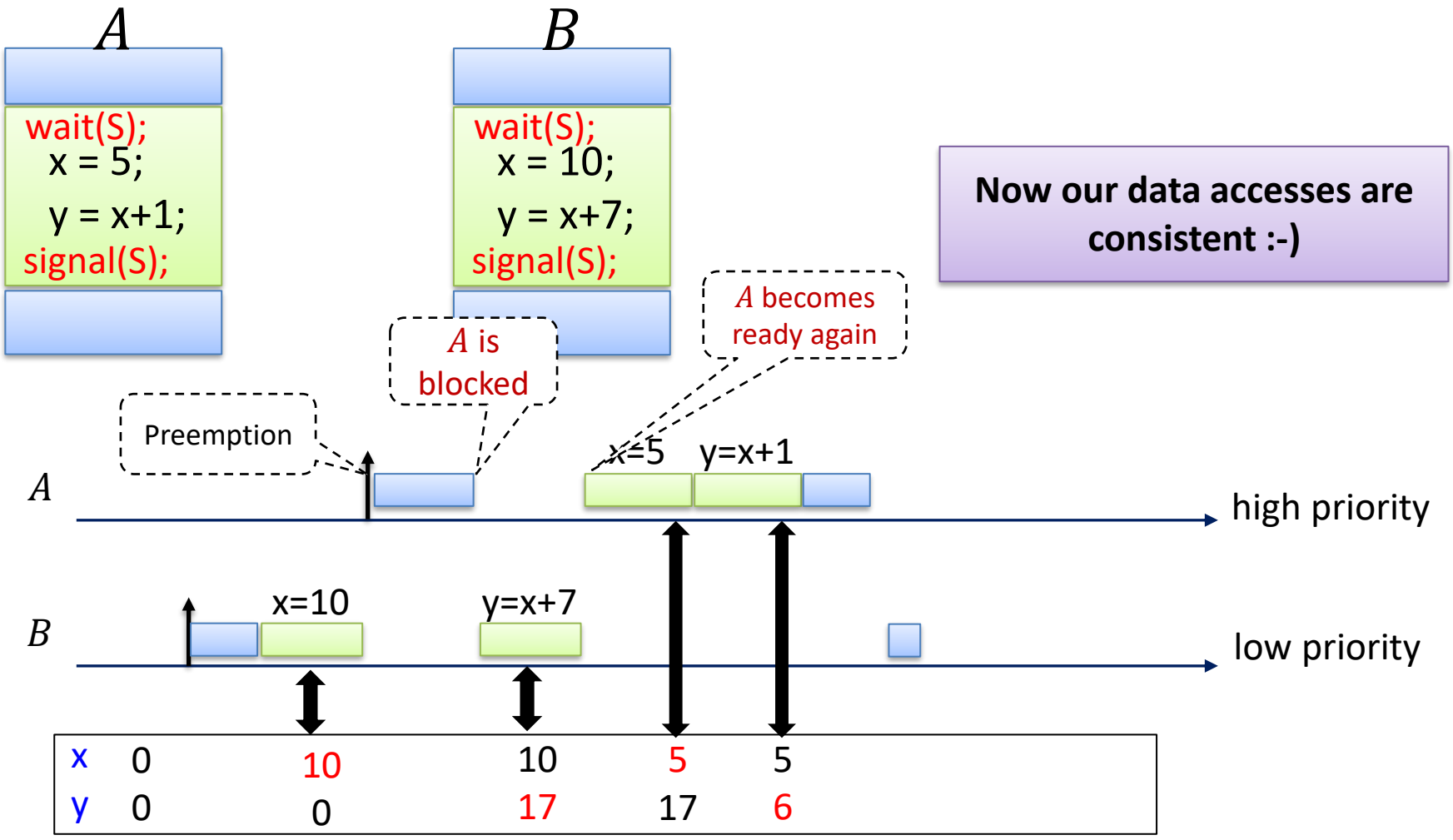
Possible results after B:

- (x=10, y=17)

Possible results after A and B:

- (x=10, y=17)
- (x=5, y=6)
- ~~(x=5, y=12)~~ ← Only under preemptive scheduling!
- ~~(x=10, y=11)~~ ← Only under preemptive scheduling!

Execute A and B together



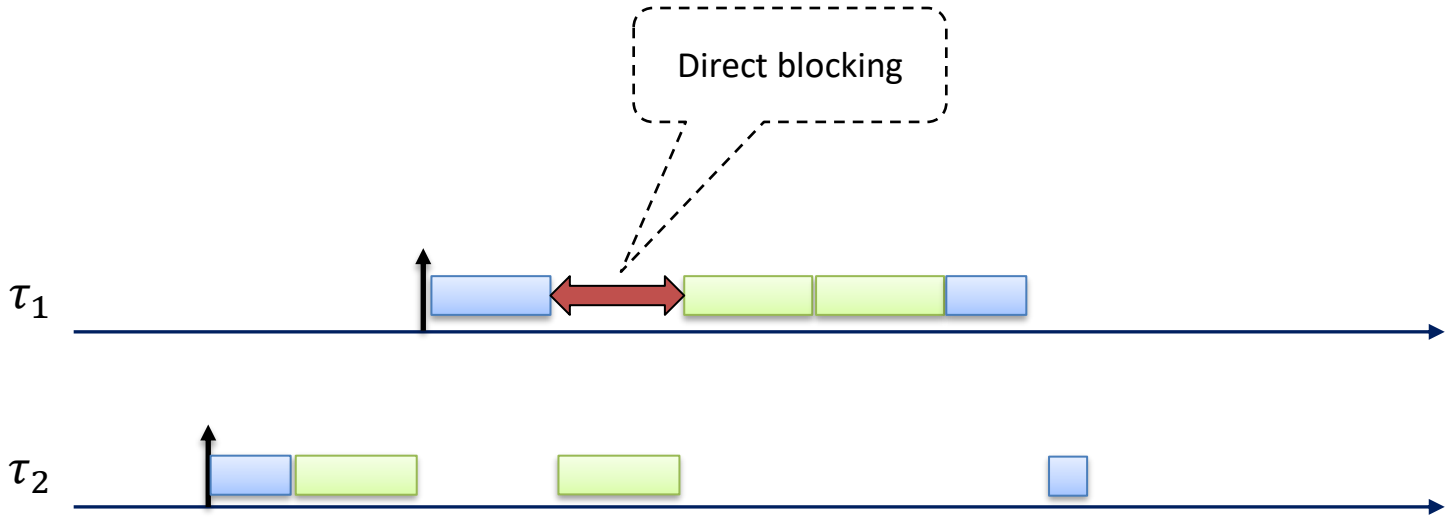
Now our data accesses are consistent :-)

<u>Homework</u>: What if *B* has higher priority and *A* enters critical section first?

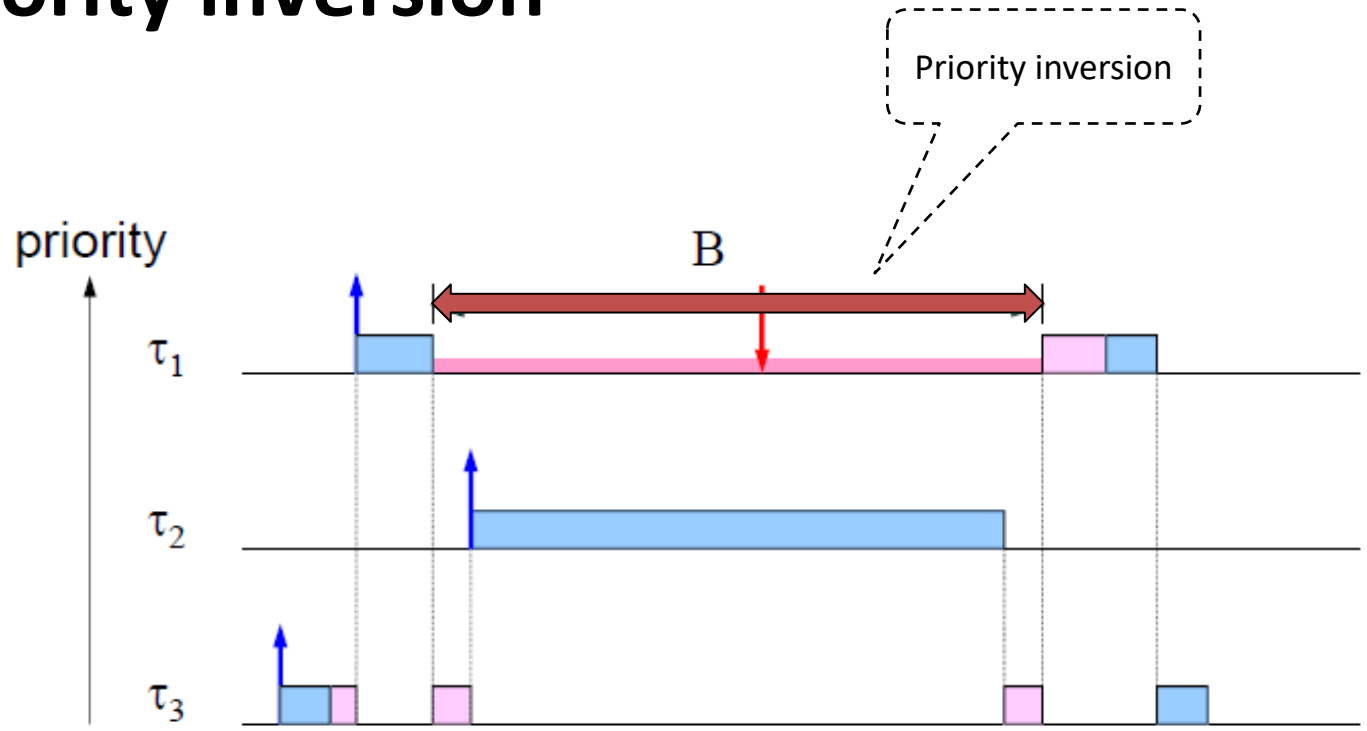
There is a certain cost that comes with using semaphores …

- Direct blocking
- Priority inversion
- Deadlocks

Direct blocking



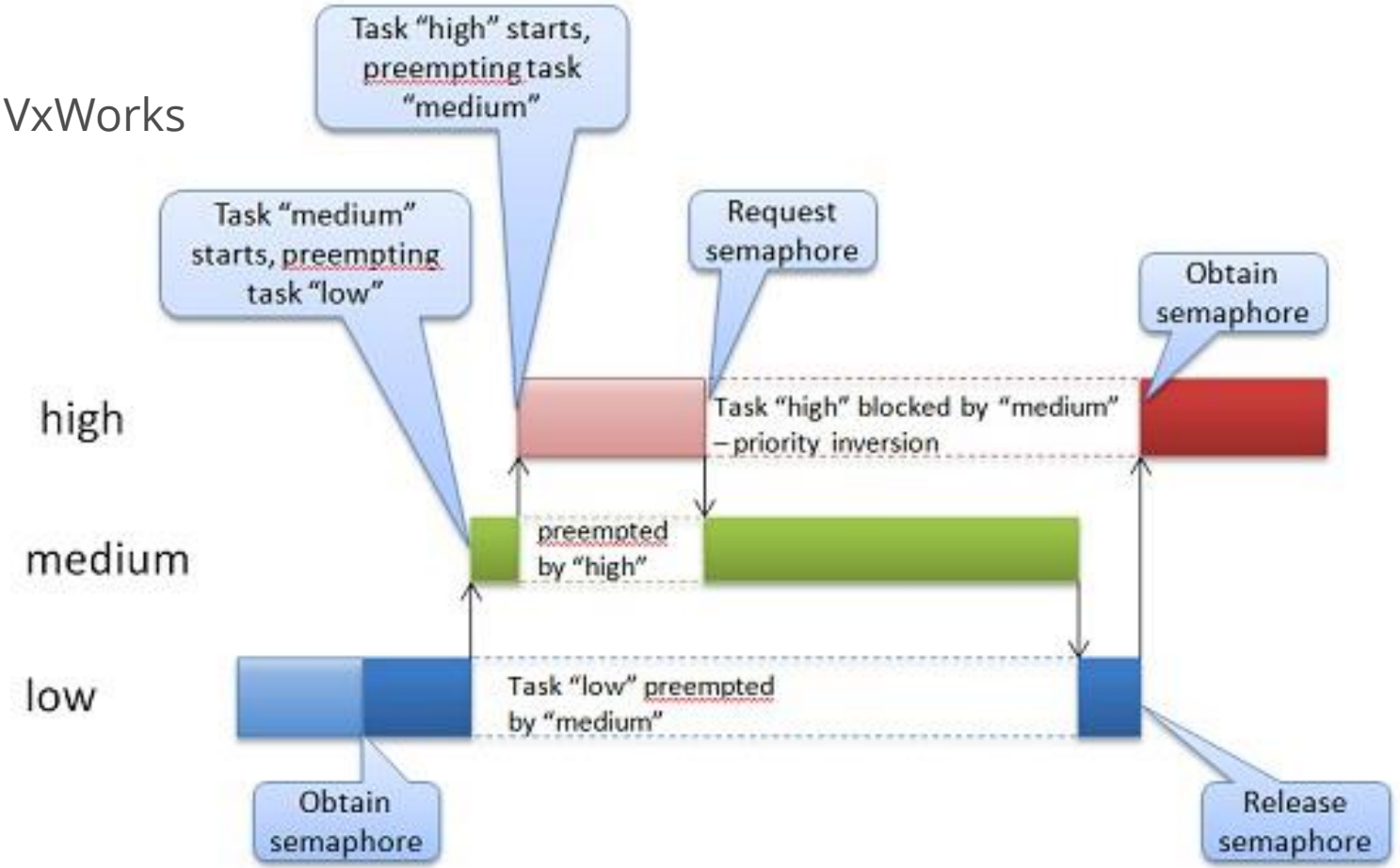
Priority inversion



A higher priority task (τ_1) is waiting for a resource, but we let a lower priority task (τ_2) who does not have anything to do with that resource run!

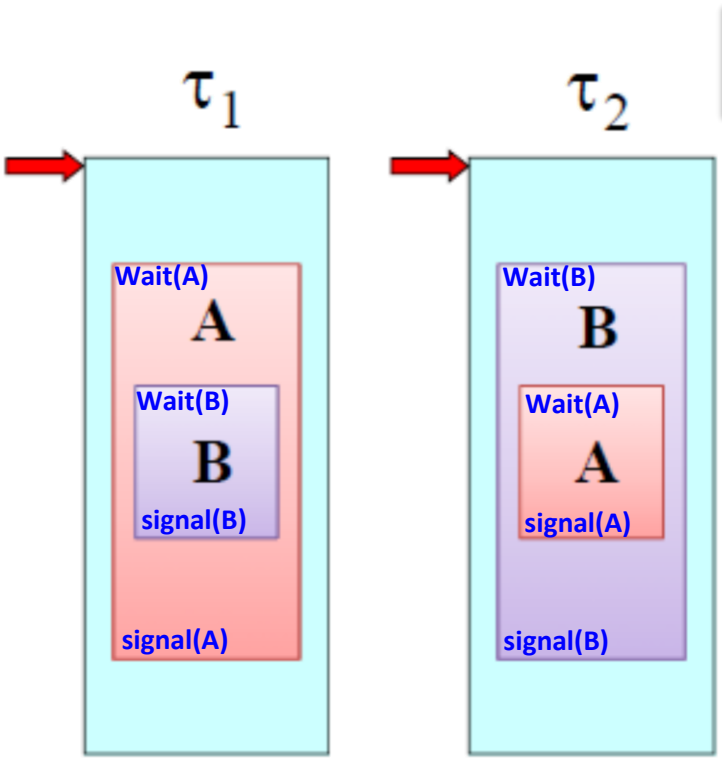
Mars PathFinder's cause of failure: (Priority inversion)

RTOS: VxWorks

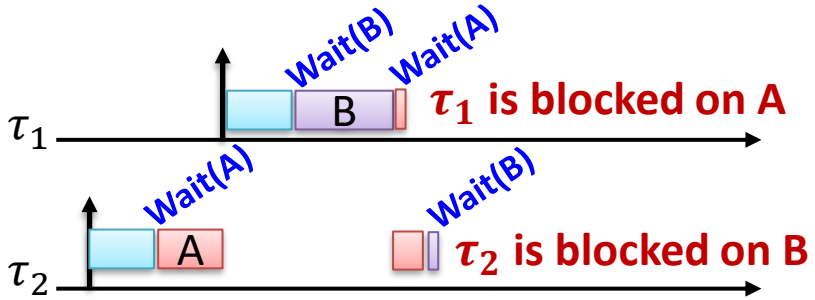


Read the story here: https://www.rapitasystems.com/blog/what-really-happened-to-the-software-on-the-mars-pathfinder-spacecraft

Deadlocks

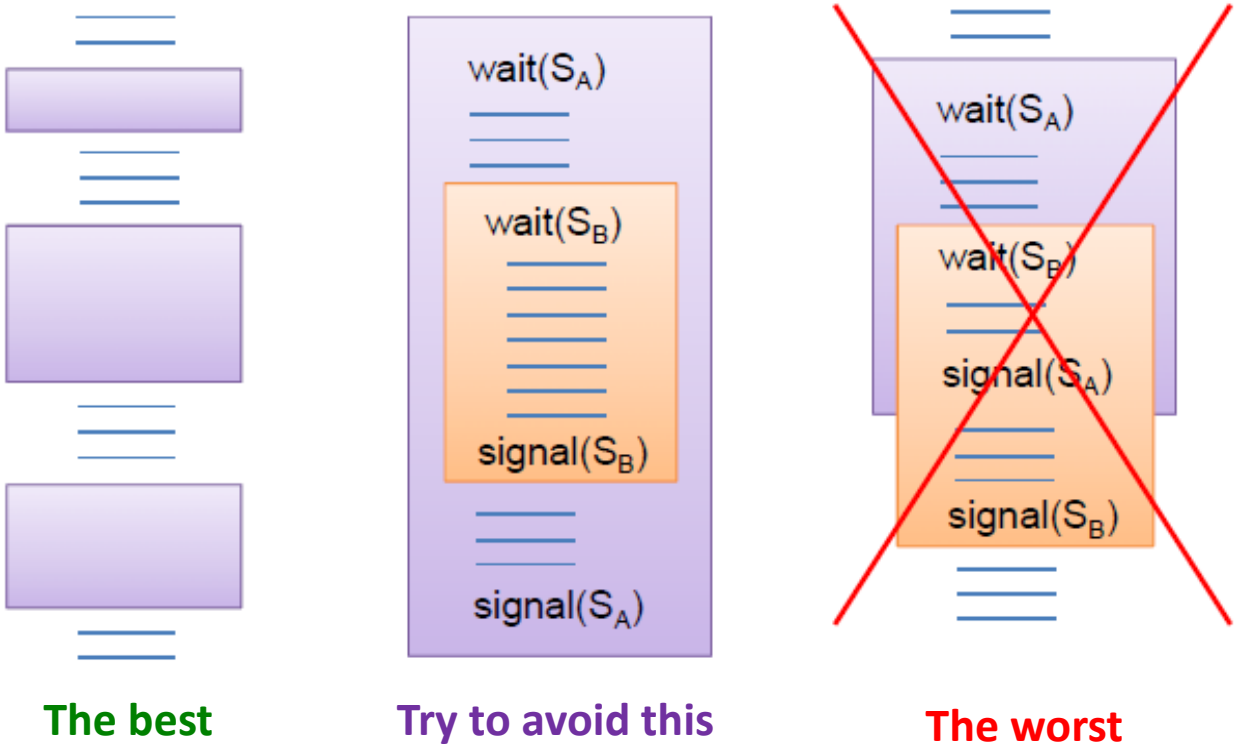


$$P_1 > P_2$$



Deadlock!

Hints: avoid cross-cutting critical sections



The best **Try to avoid this** **The worst**

Assumption for the remainder of this lecture:
No cross-cutting critical sections!

Cost of using semaphores

- **Direct blocking**
  - Higher priority task *waits* for *signal* from lower priority task

Unavoidable!

- **Priority inversion**
  - Medium priority task preempts lower priority task which directly blocks higher priority task
  - Invertion of priorities although no shared resources (*indirect blocking*)
- **Deadlocks**
  - Tasks wait for the signal from each other

How can we prevent/mitigate these effects?

We are going to design

resource access policies

that are a better fit for real-time systems!

Key design aspects of an access protocol

1: What is the "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

2- What is the "progress rule"?

Determines how to execute inside a critical section.

Example: inside critical section, execute non-preemptively

3- What is the "release rule"?

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

Example: At exit, enable preemption

Resource access protocols

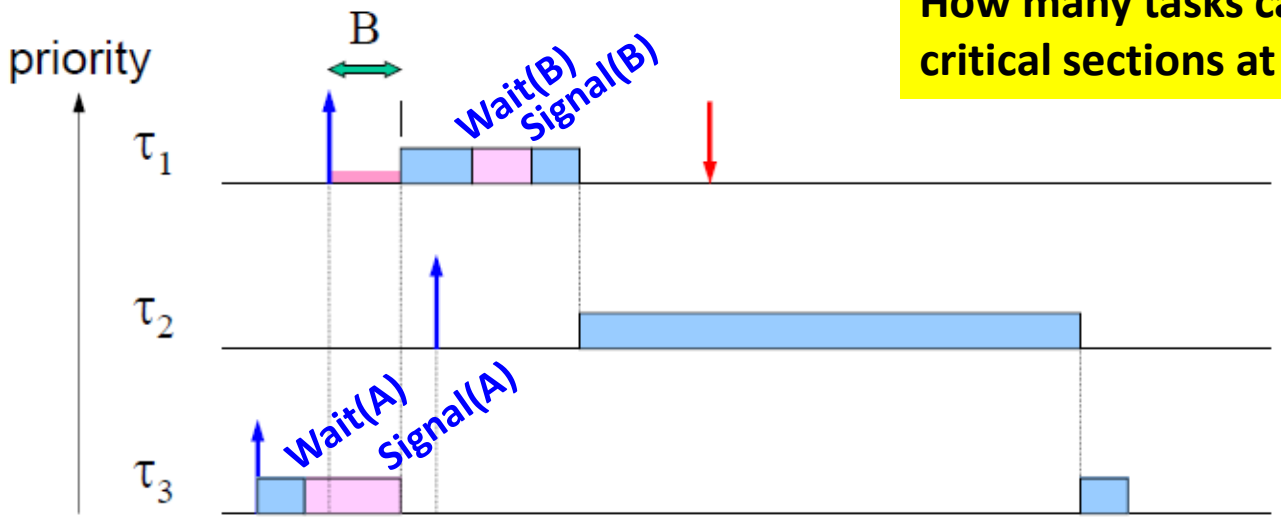
- Classical semaphores (No protocol)
- Non-Preemptive Protocol (**NPP**)
- Highest-Locker Priority (**HLP**)
  - Also known as Immediate-Priority Ceiling (IPC).
- Priority Inheritance Protocol (**PIP**)
- Priority Ceiling Protocol (**PCP**)
- Stack Resource Policy (**SRP**)

There will certainly be some exam questions from these protocols

Non-preemptive protocol (NPP)

High-level idea

Whenever a task accesses a resource, it enters the 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 own release 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.

Any idea for implementing the "priority boosting" that NPP needs?

- Access Rule: A task never blocks at the entrance of a critical section, but at its own release 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 be 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:

On wait(s): $p_i \leftarrow \max\{P_1, \dots, P_n\}$
On signal(s): $p_i \leftarrow P_i$

NPP: Blocking behavior

Blocking NPP

Under NPP, a task can be blocked by **one** critical section of a lower priority task.

$$B_i = \max\{l(Z) | Z \ from \ a \ lower \ priority \ task\}$$

Blocking that any job of task τ_i can experience

NPP: pro & cons

What are the advantages?

ADVANTAGES: simplicity and efficiency.

- **Semaphore queues are not needed**, because tasks never block on a wait(s).
- It **prevents deadlocks** and **priority inversion**.
- It is transparent to the programmer (programmer does not need to provide any info).

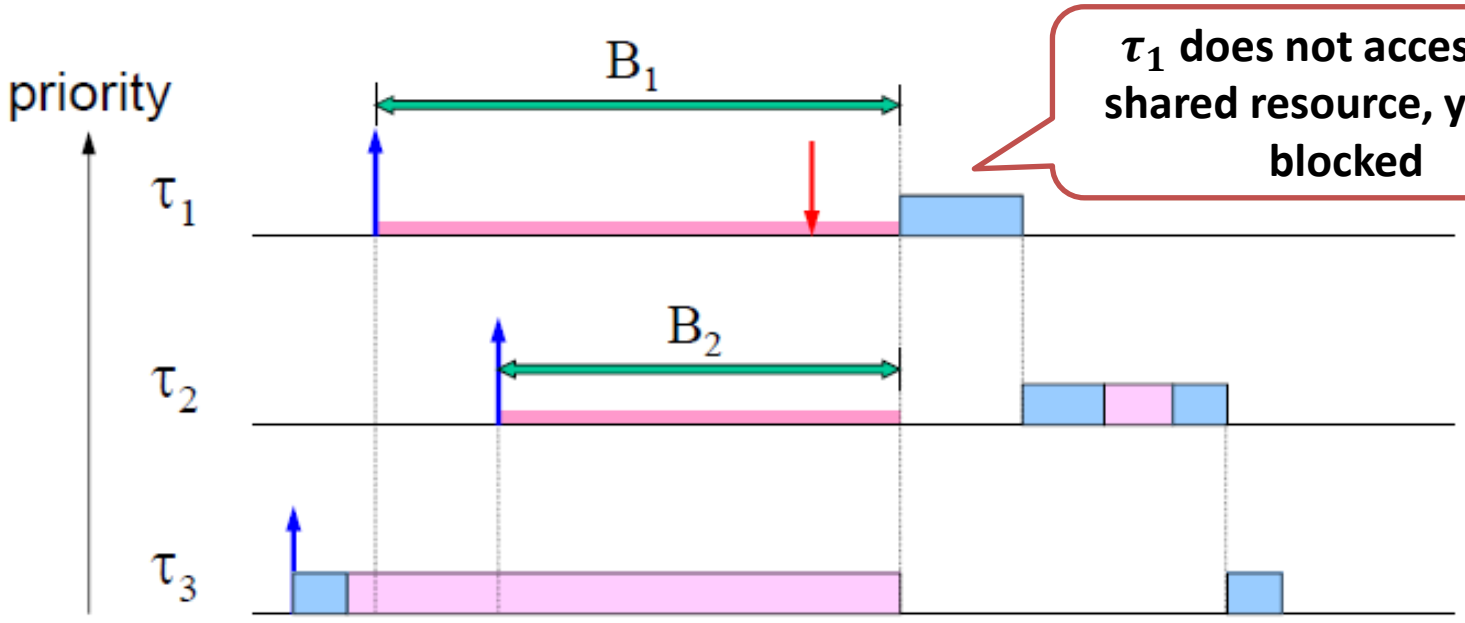
What are the disadvantages?

PROBLEMS:

1. A long critical section may delay **all high-priority tasks even if they do not need the same resource**

2. Tasks may be blocked **even if they do not use any shared resource**.

NPP: problems



τ_1 does not access any shared resource, yet it is blocked

τ_3 delays *all* higher priority tasks

Resource access protocols

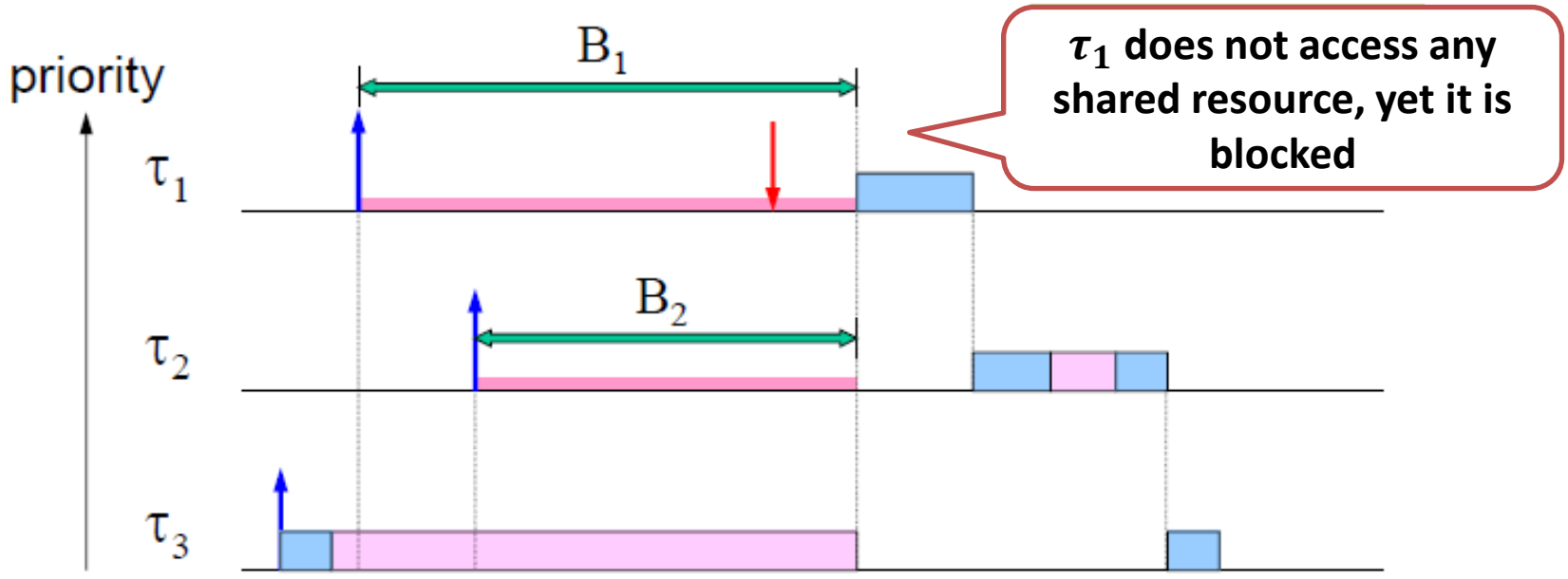
- Classical semaphores (No protocol)
- Non-Preemptive Protocol (**NPP**)
- Highest-Locker Priority (**HLP**)
  - Also known as Immediate-Priority Ceiling (IPC).
- Priority Inheritance Protocol (**PIP**)
- Priority Ceiling Protocol (**PCP**)
- Stack Resource Policy (**SRP**)

There will certainly be some exam questions from these protocols

Let's solve this problem:

Any idea?

Save τ_1 from being blocked by extending the NPP protocol



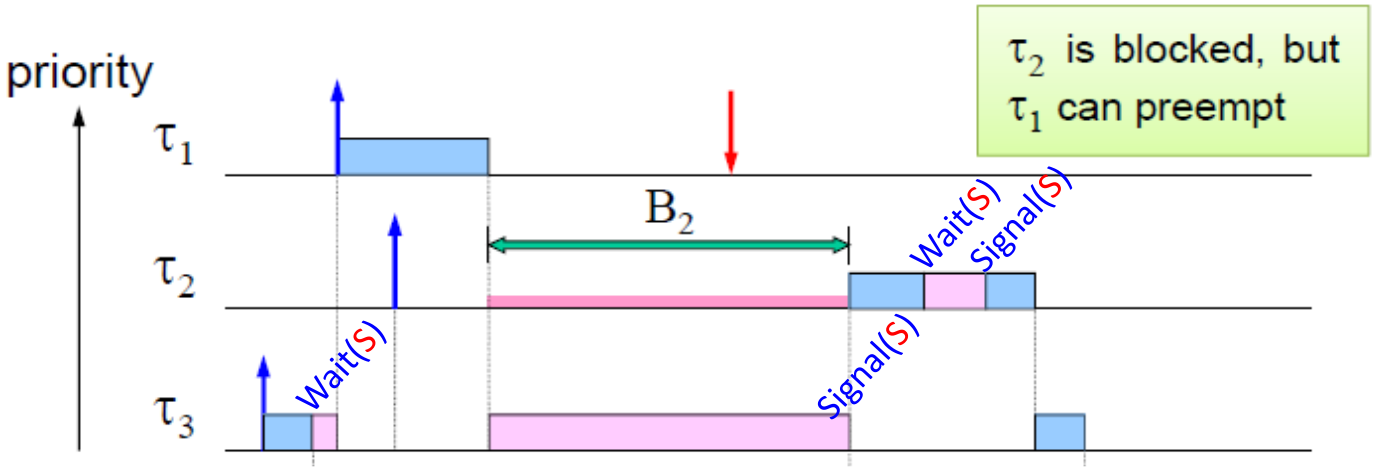
Assumptions: this system has only 3 tasks and the only tasks that may access the shared resource S are τ_2 and τ_3.

Highest-locker priority (HLP) protocol

Protocol

High-level idea

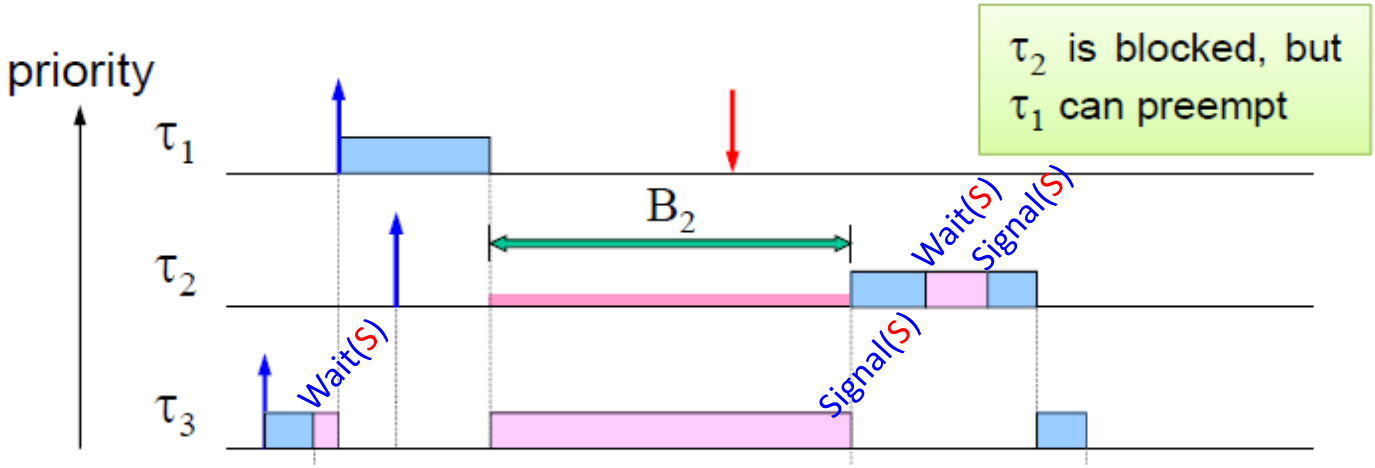
When a task accesses a resource (e.g., wait(S)), its priority upgrades to the priority of the highest-priority task that may use that resource *S*



- Access Rule: A task never blocks at the entrance of a critical section, but at its own release time.
- Progress Rule: Inside the critical section for a resource R, the task executes at the highest priority of the tasks that use R (we need to know those tasks).
- Release Rule: At exit, the dynamic priority of the task is reset to its nominal priority P_i.

Highest-locker priority (HLP) protocol

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



Priority assigned to τ_i when it uses semaphore S:

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

What is $p_3(S)$?

It is P_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)*:

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

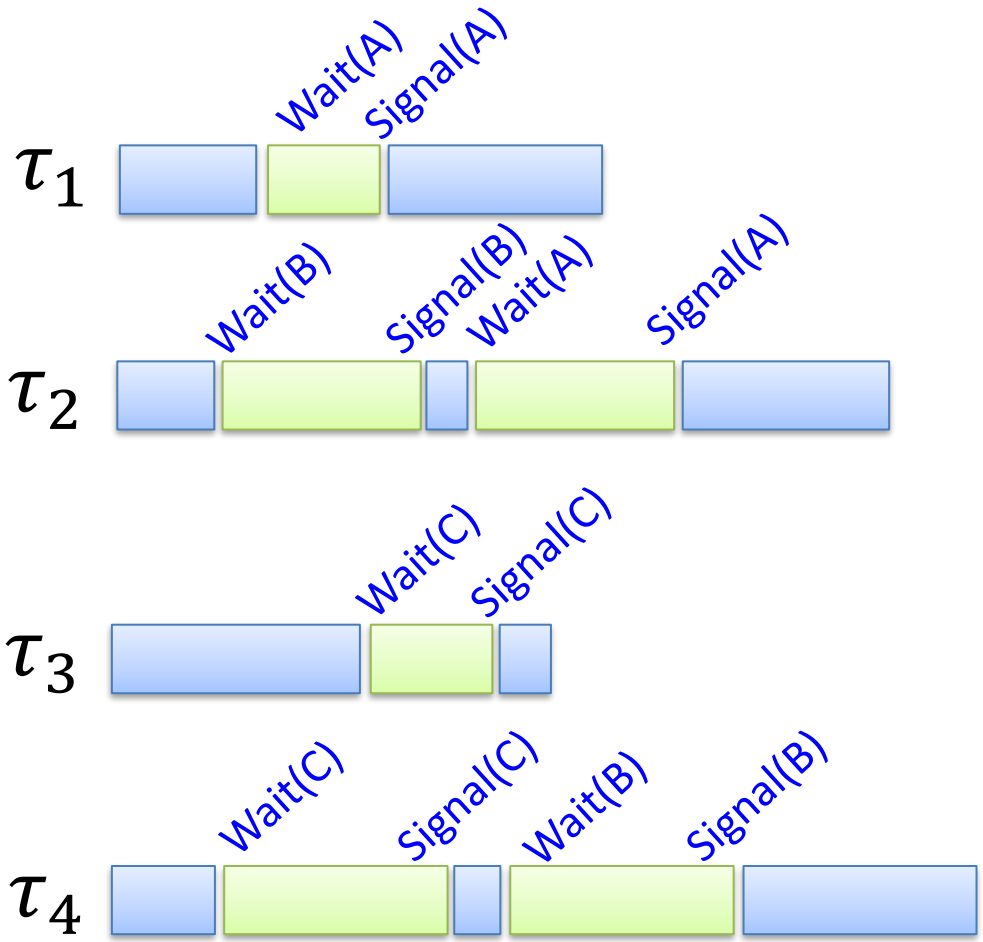
Change the wait and signal primitives as follows:

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

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

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

HLP: Example



Consider HLP protocol:

What is $p_1(A)$? P_1

What is $p_2(A)$? P_1

What is $p_2(B)$? P_2

What is $p_3(C)$? P_3

What is $p_4(C)$? P_3

What is $p_4(B)$? P_2

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

HLP: Blocking behavior

Blocking HLP

Under HLP, a task τ_i can be blocked by one critical section of a lower priority task if the resource ceiling is larger than or equal to P_i.

$$B_i = \max\{l(Z) | Z \ from \ a \ lower \ priority \ task, C(Z) \geq P_i\}$$

HLP: pro & cons

What are the advantages?

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.

What are the disadvantages?

PROBLEMS:

1. A task could be blocked even if it "may" not access a critical section (similar to NPP). But less than with NPP!

2. Programmer must provide information about the resources used by the tasks (to define the ceilings).

Resource access protocols

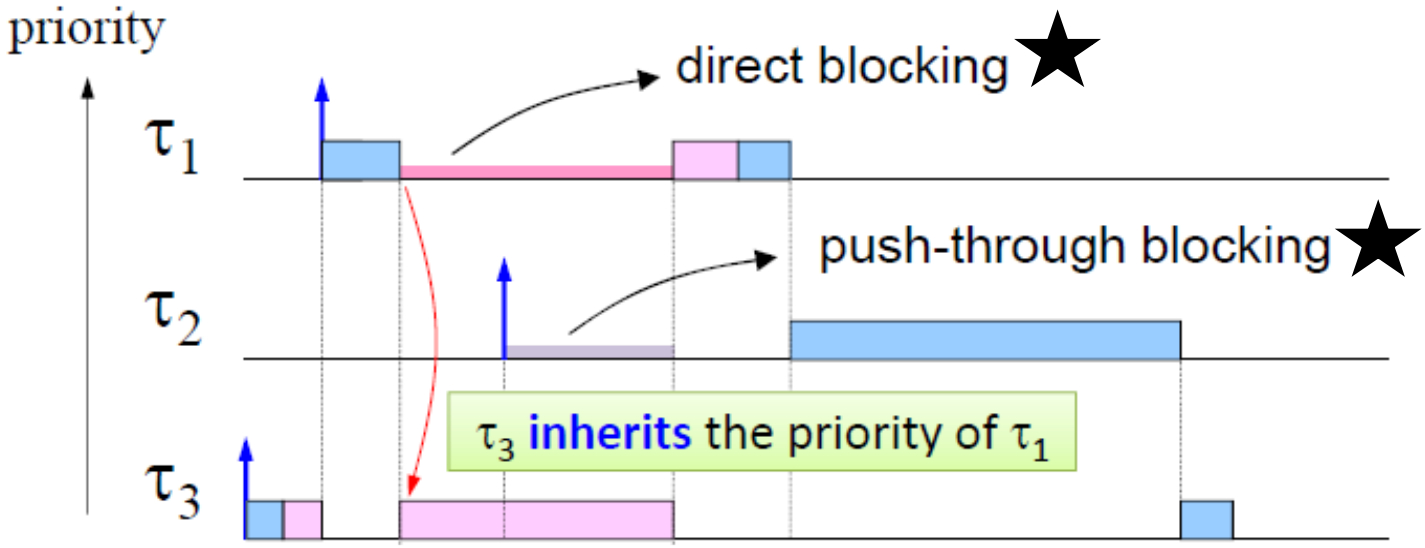
- Classical semaphores (No protocol)
- Non-Preemptive Protocol (**NPP**)
- Highest-Locker Priority (**HLP**)
  - Also known as Immediate-Priority Ceiling (IPC).
- Priority Inheritance Protocol (**PIP**)
- Priority Ceiling Protocol (**PCP**)
- Stack Resource Policy (**SRP**)

There will certainly be some exam questions from these protocols

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

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

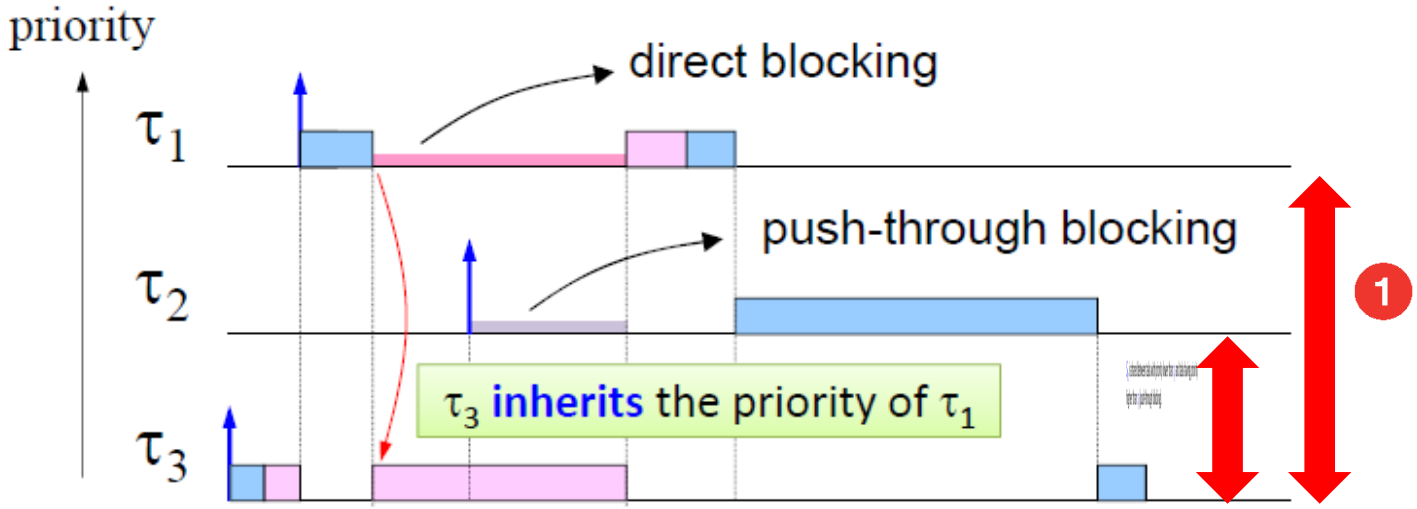
```
wait (S)    if (S == 0) {
                <suspend the calling task τc in the semaphore queue>
                <find the task τk that is locking the semaphore S>
                pk = max{Pc, pk}     // τk inherits the priority of τc if pk > Pc
                <call the scheduler>
            }
            else S = 0;
```

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

Identifying blocking resources

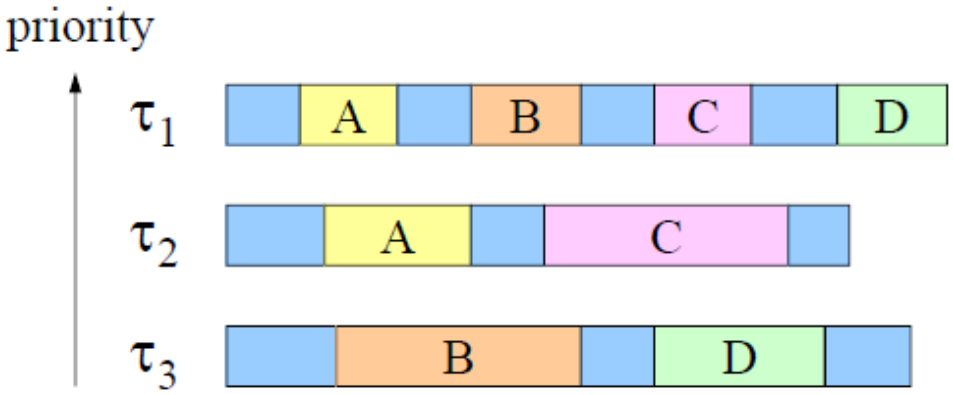
Under PIP, a task τ_i can be **blocked** on a semaphore S_k **only if**:

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).



Instead of priority inversion we have push-through blocking.
(same for NPP and HLP)

PIP: example 2

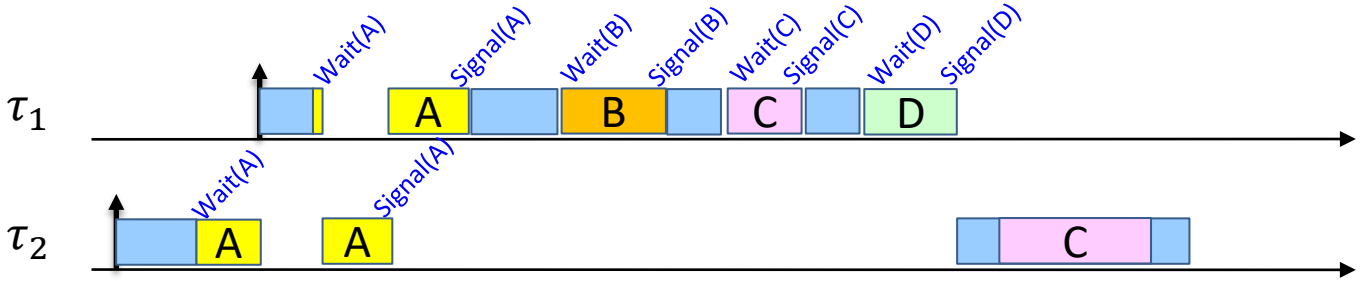


Assume that any release scenario is possible

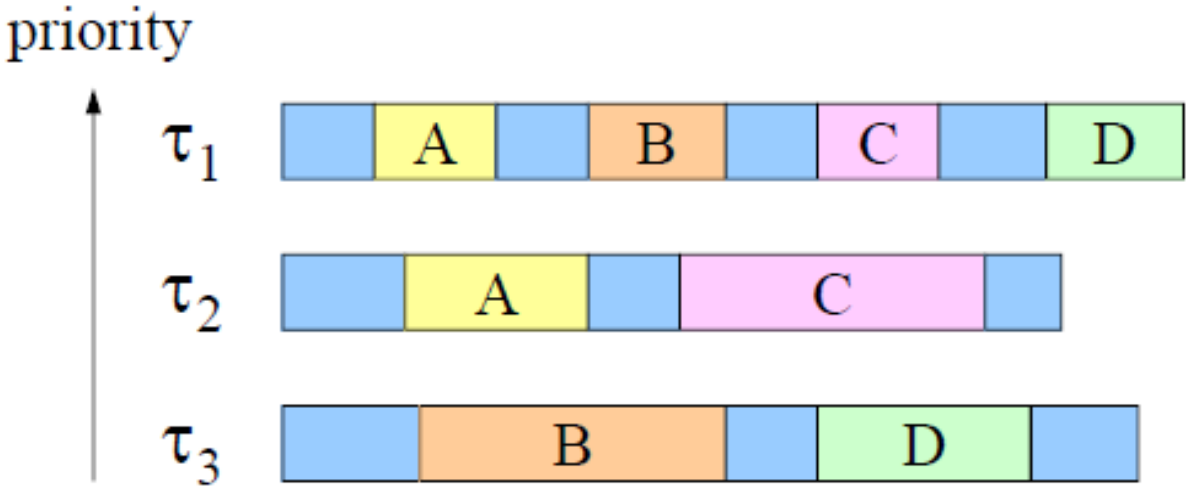
Which tasks can block τ_1? τ_2 (on A or C) and τ_3 (on B or D)

How many times can any low-priority task 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.



PIP: example 2



Which tasks can block τ_3? τ_3 cannot be blocked! It is already the lowest-priority task

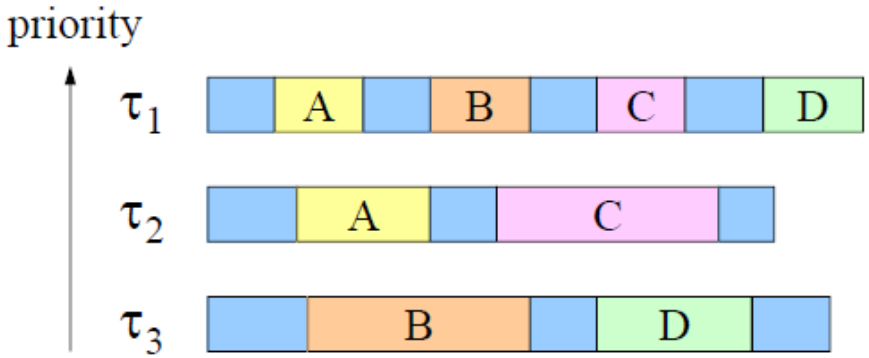
Which tasks can block τ_2? τ_3 (on B or D) because of indirect blocking

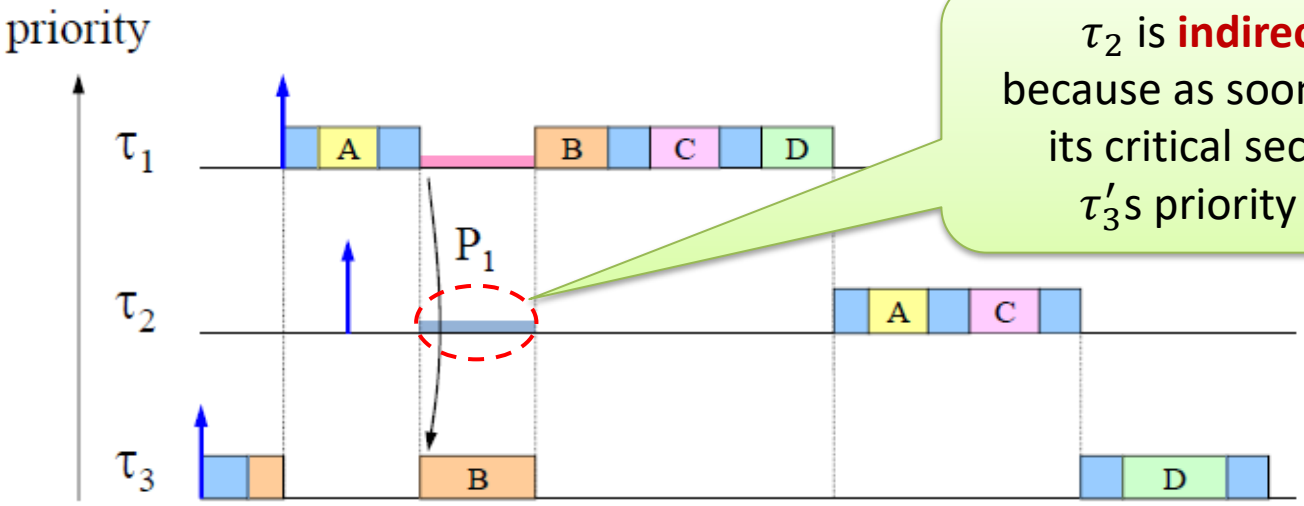
Example

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

Part 1. Is it possible that under the PIP protocol, $\boldsymbol{\tau_3}$ **directly** blocks $\boldsymbol{\tau_2}$ 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.

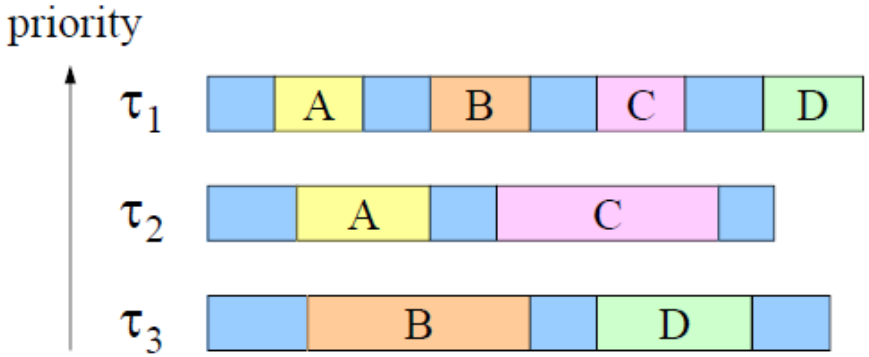




τ_2 is **indirectly blocked** by τ_3 because as soon as τ_1 wants to enter its critical section on resource B, τ_3's priority is upgraded to P_1

Example (homework)

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 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).

For the computation of B_i we refer to Buttazo's book Chapter 7.6.3

PIP: pro & cons

What are the advantages?

ADVANTAGES:

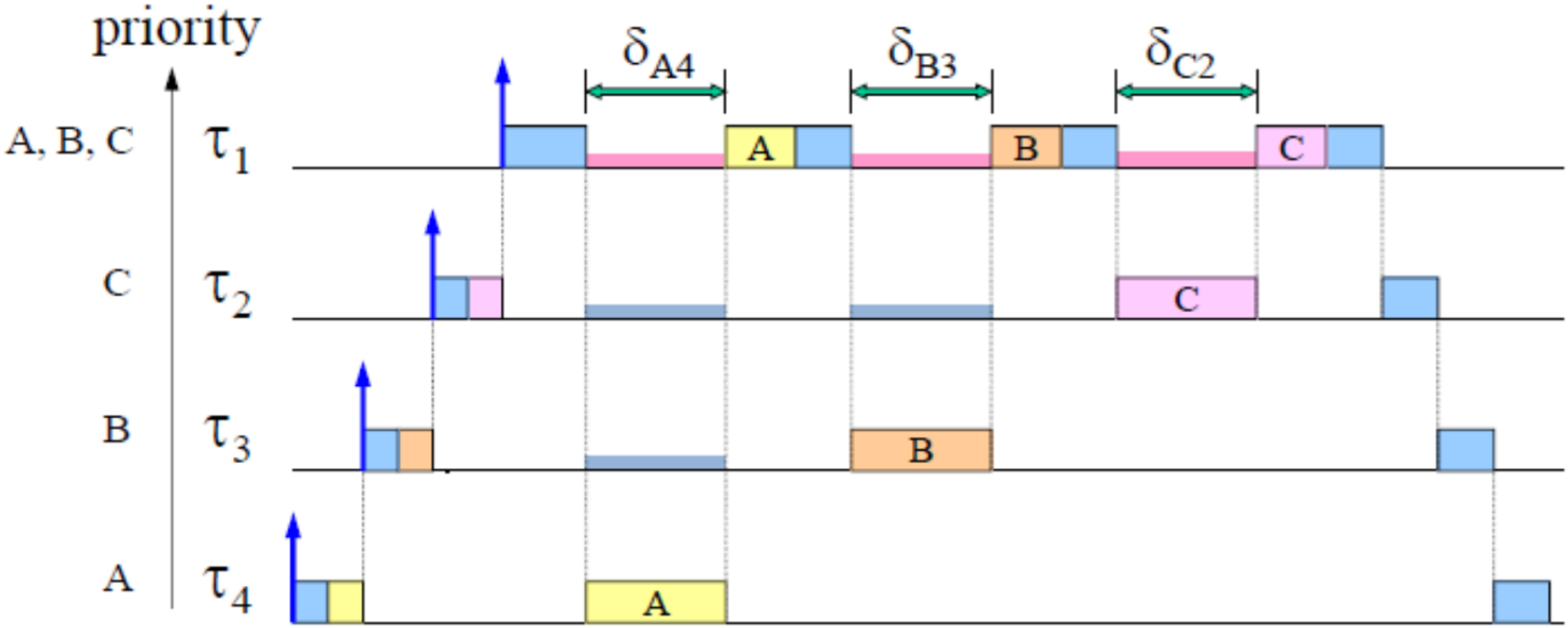
- It removes the pessimisms of NPP and HLP **(a task is blocked only when it is really needed).**
- It is transparent to the programmer .

What are the disadvantages?

PROBLEMS:

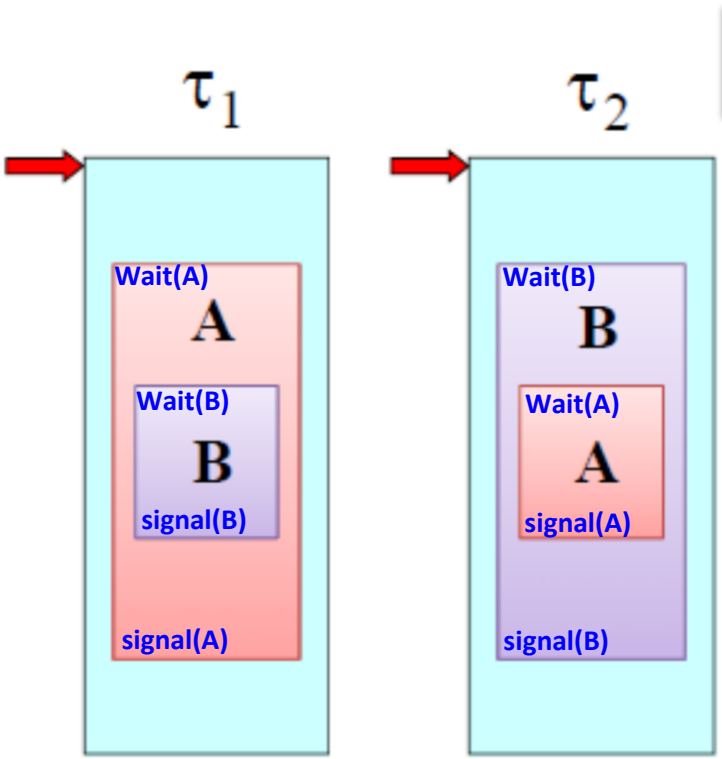
1. More complex to implement (especially to support nested critical sections).
2. It is prone to chained blocking.
3. It does not avoid **deadlocks**.★

PIP: Chained blocking problem

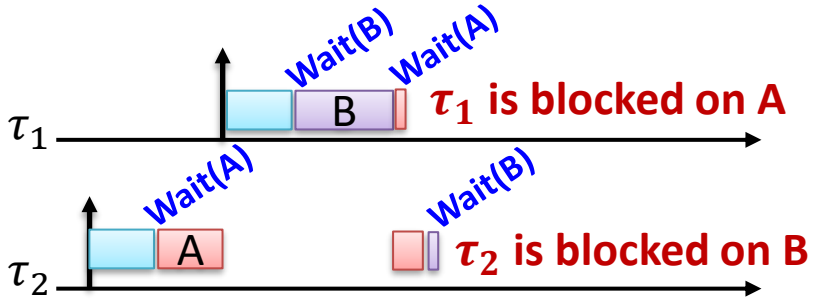


NOTE: τ_1 can be blocked at most once for each lower-priority task.

With PIP we can still have deadlocks!



$P_1 < P_2$



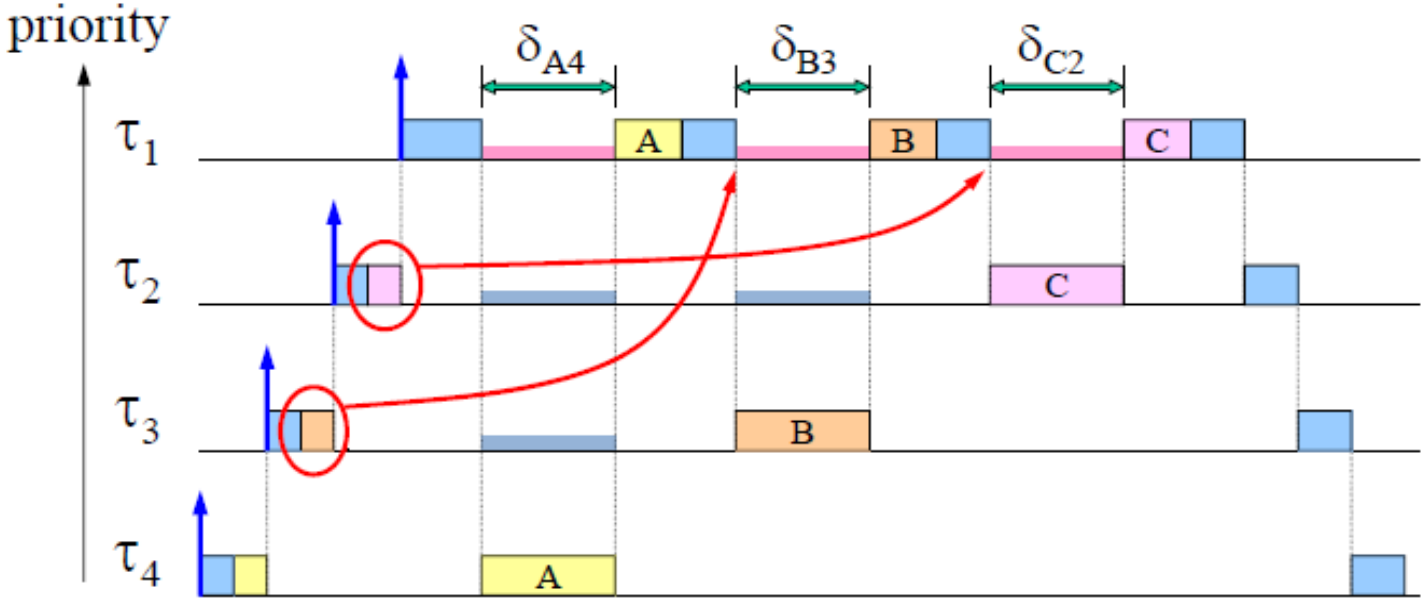
Deadlock!

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**)

There will certainly be some exam questions from these protocols

Avoiding “chained blocking problem”



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.

How to do that?

By adding an admission test to PIP!

Don't let a task τ_i enter a critical section if there might be a higher-priority task that, in the future, may potentially want to access any of the resources that are locked at the moment.

Thus:

A task τ_i can enter a critical section **only if** its priority is higher than the priority of the highest-priority task that may potentially access any of the currently locked semaphores!

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 the resource usage by high-priority tasks, each resource is assigned a **resource ceiling**:

Any task that can potentially request accessing S_k

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

$C(S_k)$ is the highest priority among the priority of tasks that can potentially request accessing S_k

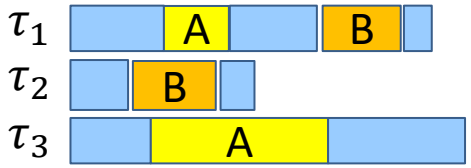
A task τ_i can enter a critical section **only if** its priority is higher than the maximum ceiling of the locked semaphores:

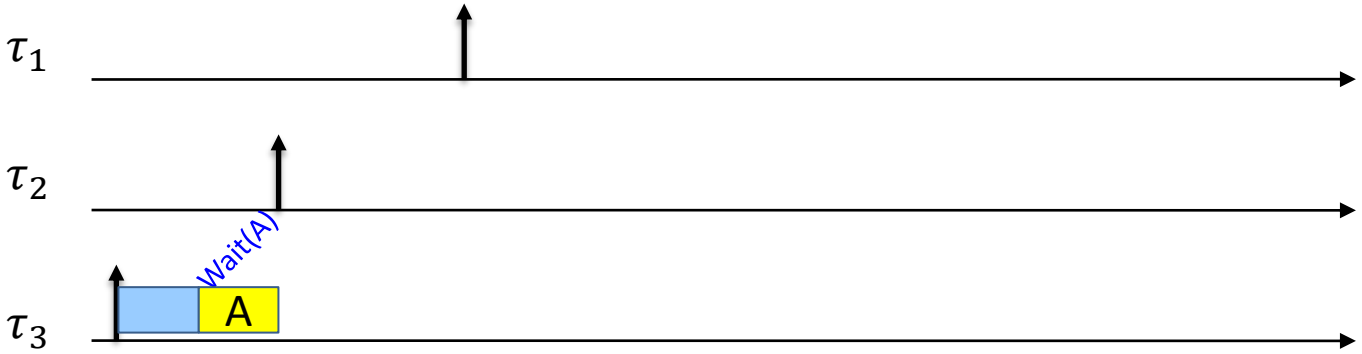
PCP access test:

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

The set of semaphores that are currently locked by any task other than τ_i

PCP: example





Admission test for τ_3

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

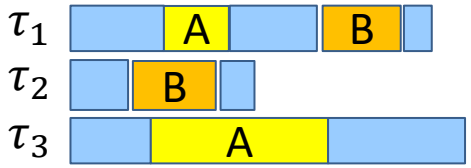
?

The test passes because no resource has been locked so far

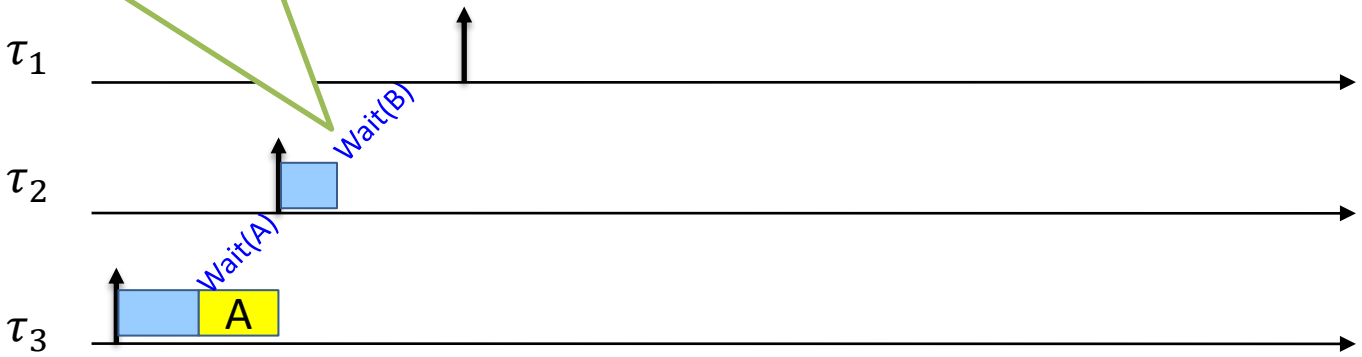
$C(A) = \max\{P_j \mid \forall \tau_j, \tau_j \text{ uses } A\}$ P_1

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



Admission test for τ_2 ?

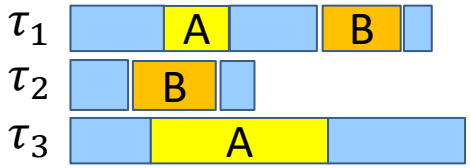
$P_2 > \max\{C(S_k) \mid S_k \in \{A, B\} \text{ and } S_k \text{ locked by tasks } \neq \tau_2\}$

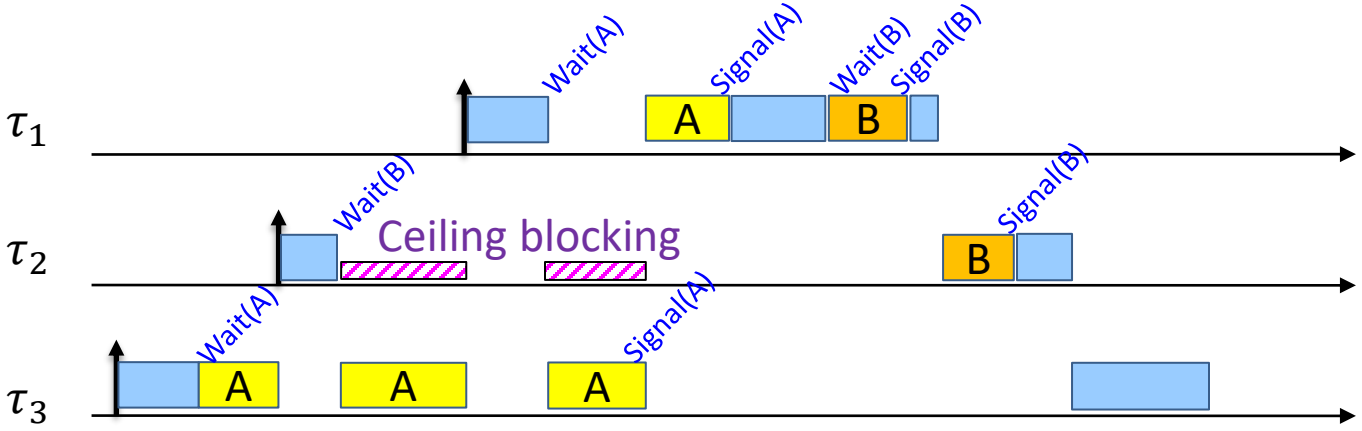
$= C(A) = P_1$

$C(A) = \max\{P_j \mid \forall \tau_j, \tau_j \text{ uses } A\}$ P_1

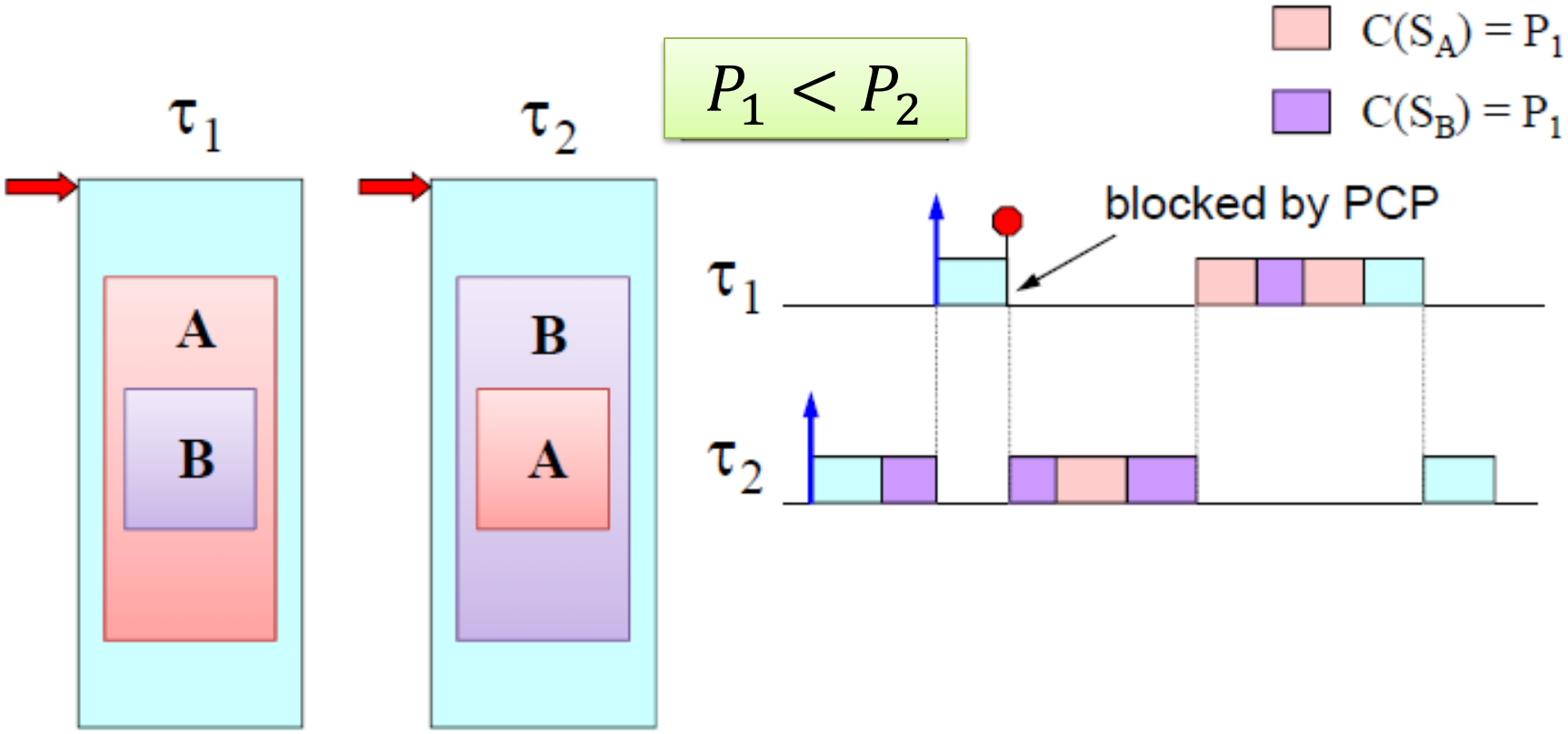
$C(B) = \max\{P_j \mid \forall \tau_j, \tau_j \text{ uses } B\}$ P_1

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: deadlock avoidance



PCP: properties

Theorem 1

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

Theorem 2

PCP prevents chained blocking.

Theorem 3

PCP **prevents** deadlocks.

PCP: Blocking behavior

Blocking PCP

Under PCP, a task τ_i can be blocked by one critical section of a lower priority task if the resource ceiling is larger than or equal to P_i.

$$B_i = \max\{l(Z) | Z \ from \ a \ lower \ priority \ task, C(Z) \geq P_i\}$$

PCP: pro & cons

What are the advantages?

ADVANTAGES:

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

What are the disadvantages?

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.

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**)

There will certainly be some exam questions from these protocols

Stack Resource Policy (SRP)

High-level idea

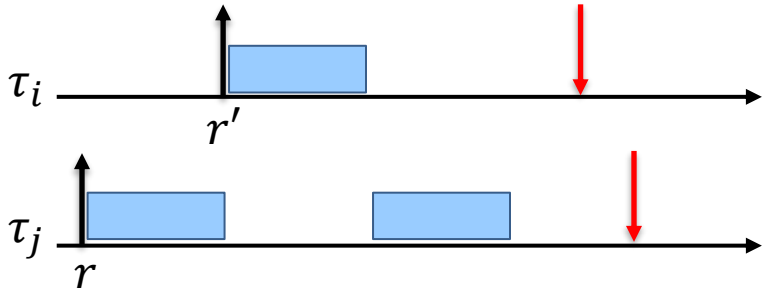
Extend PCP to Earliest-Deadline-First (EDF).

Do you see any problems here?

PROBLEM:

EDF has dynamic priorities!

SRP: Preemption levels



A job of τ_i can preempt a job of τ_j under EDF

$\Leftrightarrow r < r'$ and $r + D_j > r' + D_i$

$\Rightarrow D_i < D_j$

Definition:
We define the preemption level π_i of task τ_i such that:

$$\pi_i > \pi_j \Leftrightarrow D_i < D_j$$

SRP: Definition

- Define ceiling based on preemption level:

$$C(S_k) = \max\{\pi_j | \pi_j \; uses \; S_k\}$$

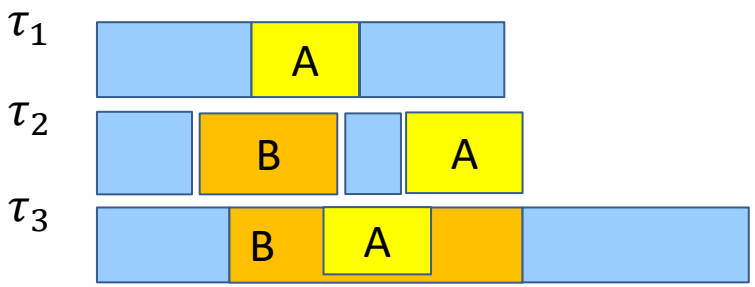
SRP *preemption* test:

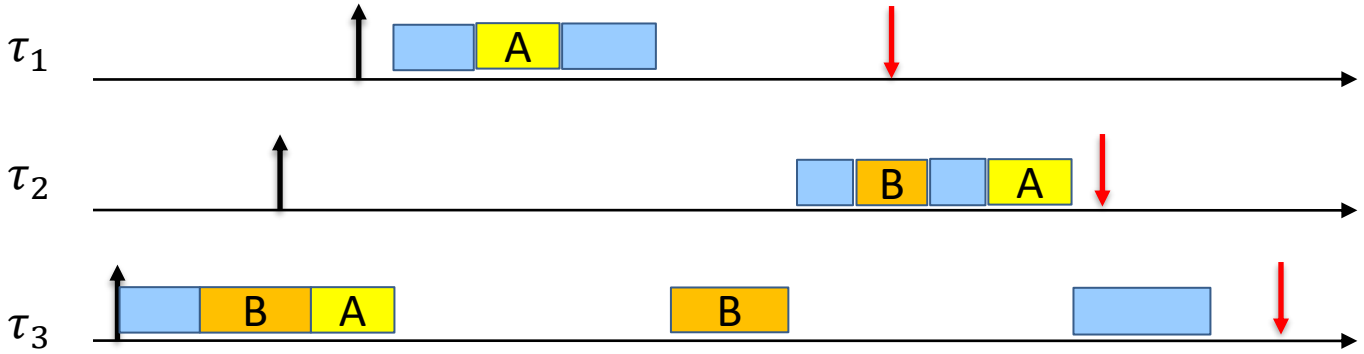
$$\pi_i > \max\{C(S_k) \mid S_k \text{ locked by tasks } \neq \tau_i\}$$

Attention:
We test at <u>preemption attempt</u> not at resource request (PCP)!

<u>Homework</u>: Formulate *access rule, progress rule* and *release rule* for SRP by yourself.

SRP: Example



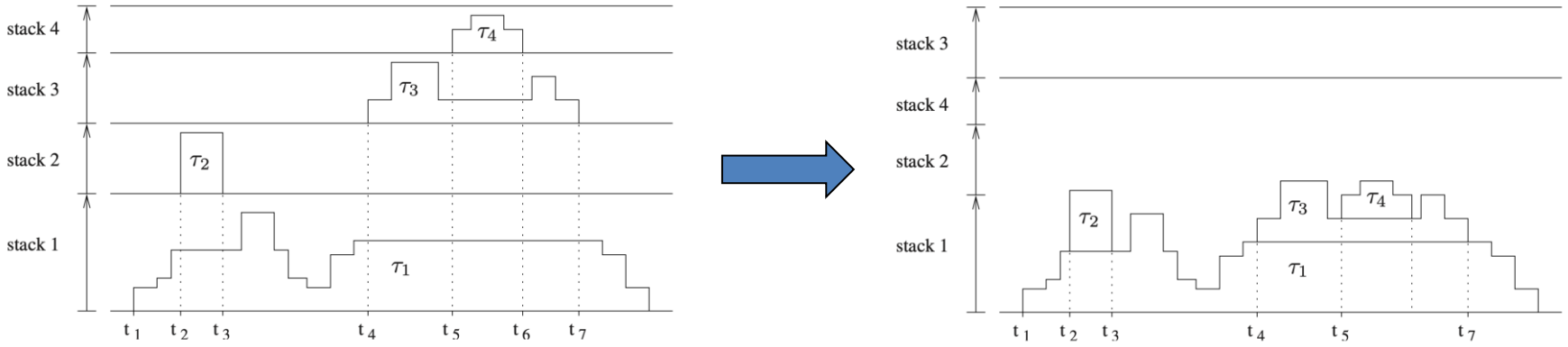


$\pi_1 > \pi_2 > \pi_3$

$C(A) = \pi_1, C(B) = \pi_2$

SRP: Properties

SRP allows sharing of runtime stack!



(From Buttazo's reference book Figures 7.21 and 7.22)

SRP: Properties

Same as PCP!

Theorem 1

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

Theorem 2

SRP prevents chained blocking.

Theorem 3

SRP **prevents** deadlocks.

SRP: Blocking behavior

Blocking SRP

Under SRP, a task τ_i can be blocked by one critical section of a task with lower preemption level if, the resource ceiling is larger than or equal to π_i.

$$B_i = \max\{l(Z) | Z \ task \ with \ lower \ preemption \ level, C(Z) \geq \pi_i\}$$

SRP: One more note …

- SRP even allows for **multi-unit resources**
  (= a certain number of tasks can access the same resource at the same time)

- This is achieved by extending the ceiling function to account for the number of available slots.

SRP: pro & cons

What are the advantages?

ADVANTAGES:

- It limits blocking to the length of a single critical section.
- It avoids deadlocks when using nested critical sections.
- Applicable to EDF
- Stack sharing and multi-unit resources are allowed

What are the disadvantages?

PROBLEMS:

1. More complex to implement.
2. It can create **unnecessary blocking** (it is pessimistic like HLP, PCP).
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 ?
SRP	any	medium	Arrival	no	yes	Harder ?

How do we analyze tasks scheduled by resource access protocols?

?

Analysis

High-level idea: Include the blocking factor B_i!

Theorem 1: Liu and Layland for FP

$$\forall i = 1, \dots, n: \sum_{h:\, P_h > P_i} \frac{C_h}{T_h} + \frac{C_i + B_i}{T_i} \leq i\left(2^{\frac{1}{i}} - 1\right)$$

Theorem 2: Hyperbolic test for FP

$$\forall i = 1, \dots, n: \prod_{h:\, P_h > P_i} \left(\frac{C_h}{T_h} + 1\right)\left(\frac{C_i + B_i}{T_i} + 1\right) \leq 2$$

Analysis

High-level idea: Include the blocking factor B_i!

Theorem 3: Response time analysis for FP

$$R_i^{(0)} = C_i + B_i$$

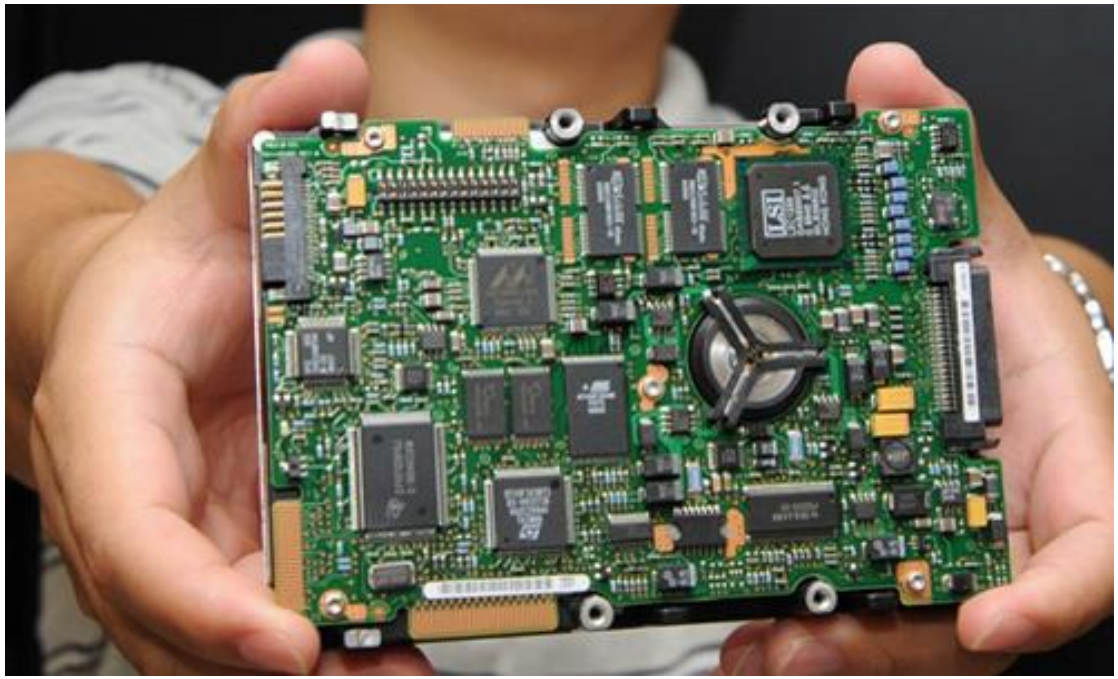
$$R_i^{(s)} = C_i + B_i + \sum_{h:\, P_h > P_i} \left\lceil \frac{R_i^{(s-1)}}{T_h} \right\rceil C_h$$

Theorem 4: Utilization based test for EDF

$$\forall i = 1, \ldots, n: \quad \sum_{h:\, P_h > P_i} \frac{C_h}{T_h} + \frac{C_i + B_i}{T_i} \leq 1$$

How would the RTA for non-preemtive tasks change?

General guidelines for real-time system engineers



Hint 1: Shorten critical sections

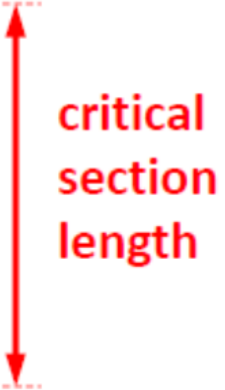
Make critical sections as short as possible.

```
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?



Hint 1: 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;
        }
        ...
}
```

critical section length

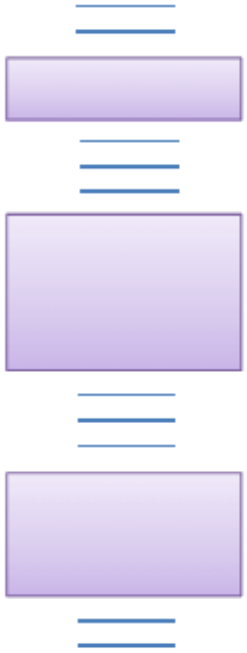
Hint 2: Avoid critical sections across loops or conditions

What can go wrong here?

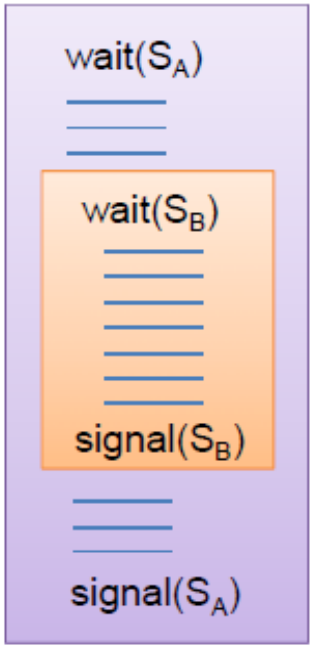
```
...
wait(s);
results = x + y;
while (result > 0) {
      v[i] = i*(x + y);
      if (v[i] < x*y)
            results = results - y;
      else
            signal(s);
}
...
```

this code is very UNSAFE since "signal" could never be executed, and the shared resource *s* could remain blocked forever!

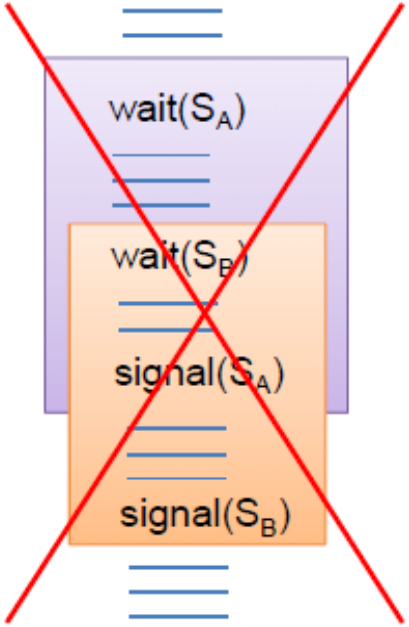
Hint 3: Avoid nested and cross-cutting critical sections



The best



Try to avoid this



The worst

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!

Also:
Potential deadline misses!

Guideline

1. Make critical sections as short as possible.

2. Avoid making critical sections across loops or conditional statements.

3. Try to avoid nested critical sections.
   - If nested critical sections are unavoidable, at least avoid cross-cutting critical sections.

Resource access protocols help you to control the behavior of critical sections.