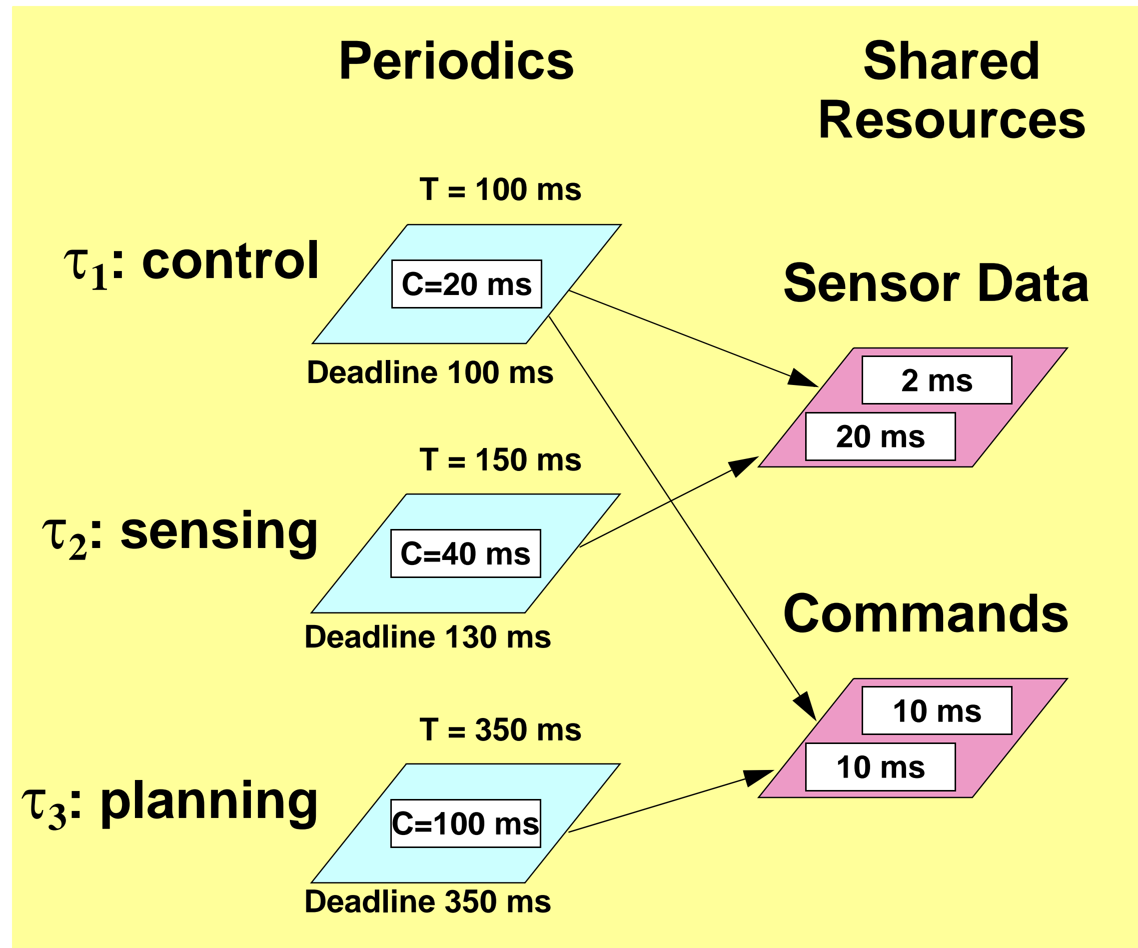
5. Synchronization

UNIVERSIDAD DE CANTABRIA

5.1 Unbounded priority inversion
5.2 Synchronization protocols
5.3 Non-preemptible sections
5.4 Immediate priority ceiling
5.5 Priority inheritance
5.6. Priority ceiling protocol
5.7 Comparison of synchronization protocols
5.8 Analyzing blocking times
5.9 Modelling synchronization
5.10 Lessons learned

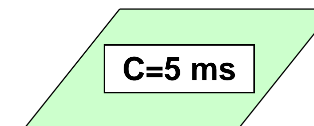
Sample Problem: Synchronization



Aperiodics

Emergency

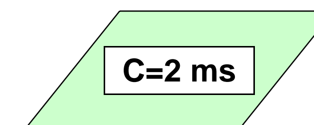
Minimum Interarrival Time = 50 msec



Deadline = 6 msec after arrival

I/O-processing

Average Interarrival Time = 40 msec



Desired Average Response Time = 20 ms

© José Carlos Palencia, Héctor Pérez Tijero 13/Feb/12

Notes:

UNIVERSIDAD DE CANTABRIA

In this section we will consider the same periodic tasks as before:

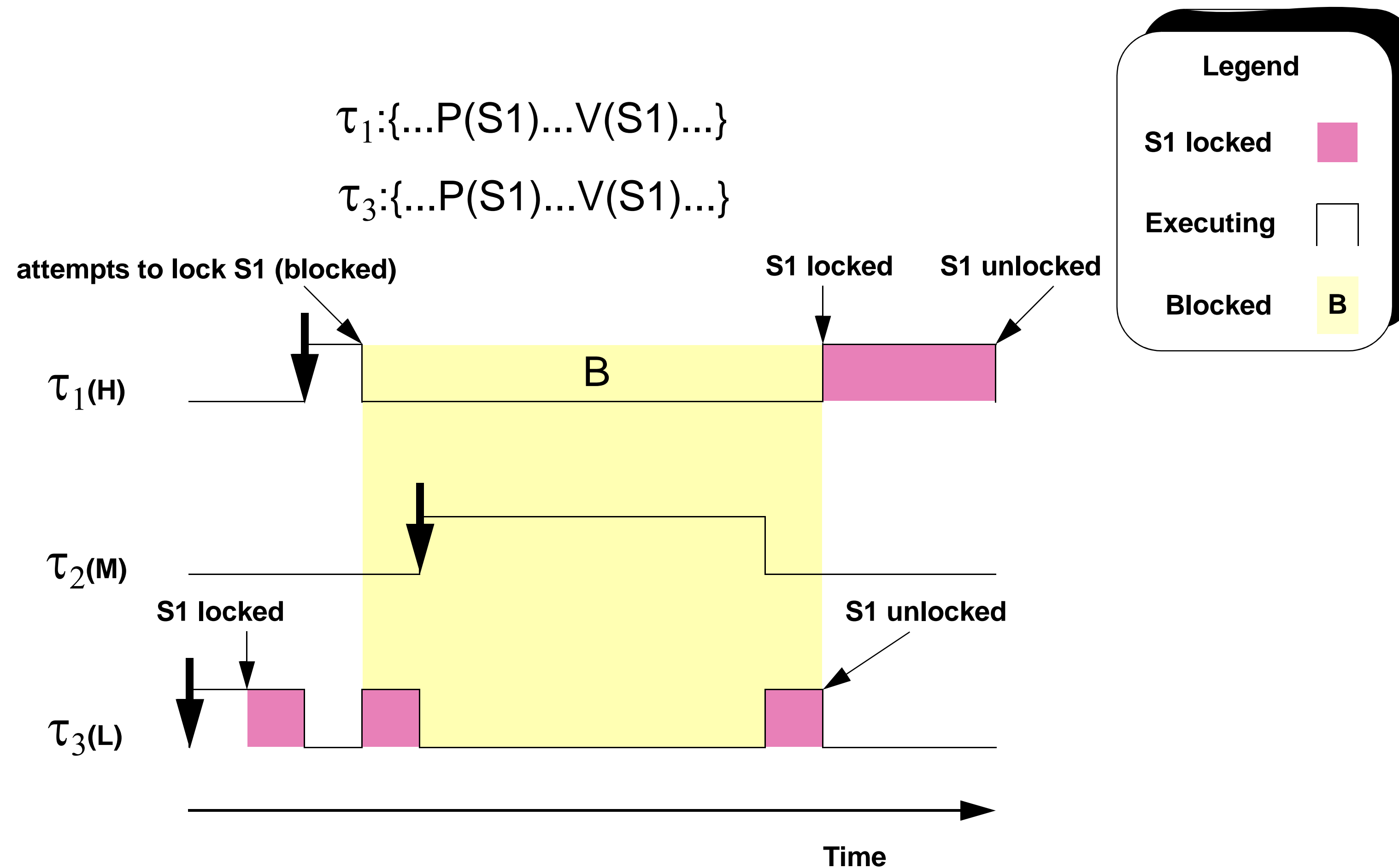
- Periodic task τ_1: execution time = 20 msec; period = 100 msec; deadline is at the end of each period.
- Periodic task τ_2: execution time = 40 msec; period = 150 msec; deadline is 20 msec before the end of each period.
- Periodic task τ_3: execution time = 100 msec; period = 350 msec; deadline is at the end of each period.

However, in addition, we consider the blocking effects of task synchronization:

- Tasks τ_1 and τ_2 share access to a common data server; τ_2 reserves the server for exclusive use during 20 msec., and τ_1 during 2 msec.
- Tasks τ_1 and τ_3 share access to a communication server; both τ_1 and τ_3 may block the server for exclusive use during 10 msec. each.

We will examine the effect of this blocking on the schedulability of the periodic tasks.

5.1 Unbounded Priority Inversion: Synchronization Model



Notes:

UNIVERSIDAD DE CANTABRIA

Critical sections are sections of code that use a resource, during which other tasks must not use that resource. It is possible to guard critical sections by disabling interrupts and prohibiting preemption, but this can have undesirable effects.

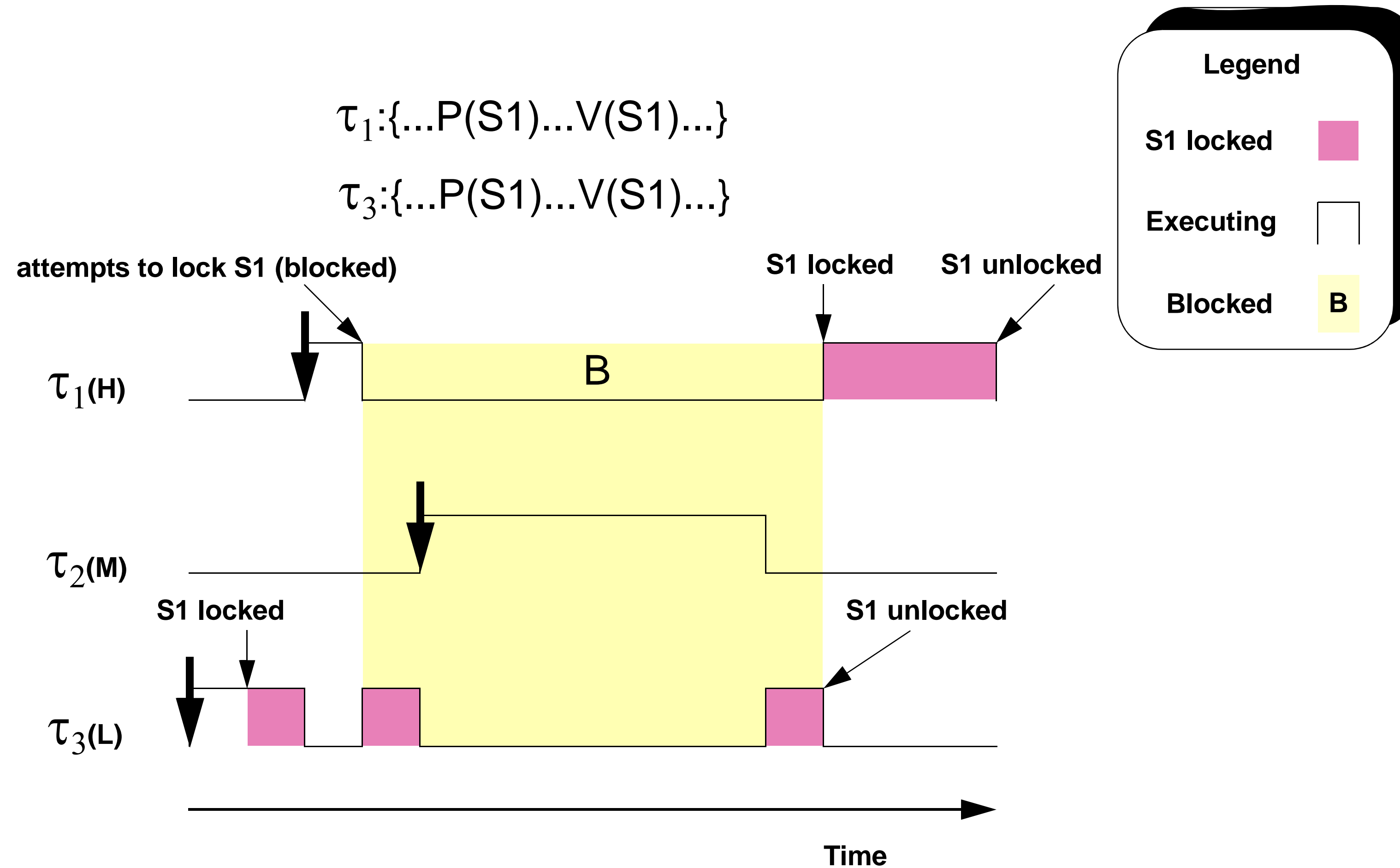
We model synchronization (or more accurately, mutual exclusion) using the concept of critical sections guarded by semaphores. Suppose two tasks τ_1 and τ_3 must avoid interfering with each other. We call the section of code in τ_1 that must be executed without interference from τ_3 a critical section. Likewise, the section of code in τ_3 that must execute without interference from τ_1 is called a critical section. A semaphore may be used to guard these critical sections as follows:

- To enter a critical section, a task must lock the semaphore S protecting the critical section. We use P(S) to indicate locking semaphore S.
- Locks on semaphores are exclusive; if S is already locked, the task must wait. The task unlocks S upon exiting the critical section, which allows some other task to lock S. We use V(S) to indicate unlocking S.

We will discuss several *synchronization protocols* that provide rules for:

- When to grant a lock request on a free semaphore.
- What priority to use when executing the critical section and how to queue the tasks waiting for a semaphore.

Unbounded Priority Inversion



Notes:

UNIVERSIDAD DE CANTABRIA

A very simple synchronization protocol is:

- Always grant a lock request on a free semaphore.
- Execute the critical section at the priority of the task locking the semaphore.

Looking closer at the above example of synchronization, we can see what can go wrong with this simple protocol. In this example a low-priority task τ_3 and a high-priority task τ_1 share a semaphore. An intermediate-priority task τ_2 does not need the semaphore.

- τ_3 locks the semaphore and begins executing its critical section.
- Later, τ_1 preempts τ_3, executes for a while, and then tries to lock the semaphore.
- τ_1 is blocked, because τ_3 already has locked the semaphore; τ_3 resumes execution.
- τ_2 preempts τ_3 and runs however long it wishes; τ_1 is blocked for the entire duration.

The high-priority task τ_1 is blocked for the duration of τ_3's critical section, **plus** the entire duration of τ_2's execution. Clearly, τ_1 is experiencing priority inversion. In fact, the duration of priority inversion is not bounded by the duration of critical sections, since any intermediate-priority task that can preempt τ_3 will indirectly block τ_1. We call this problem **unbounded priority inversion**.

5.2. Synchronization Protocols

UNIVERSIDAD DE CANTABRIA

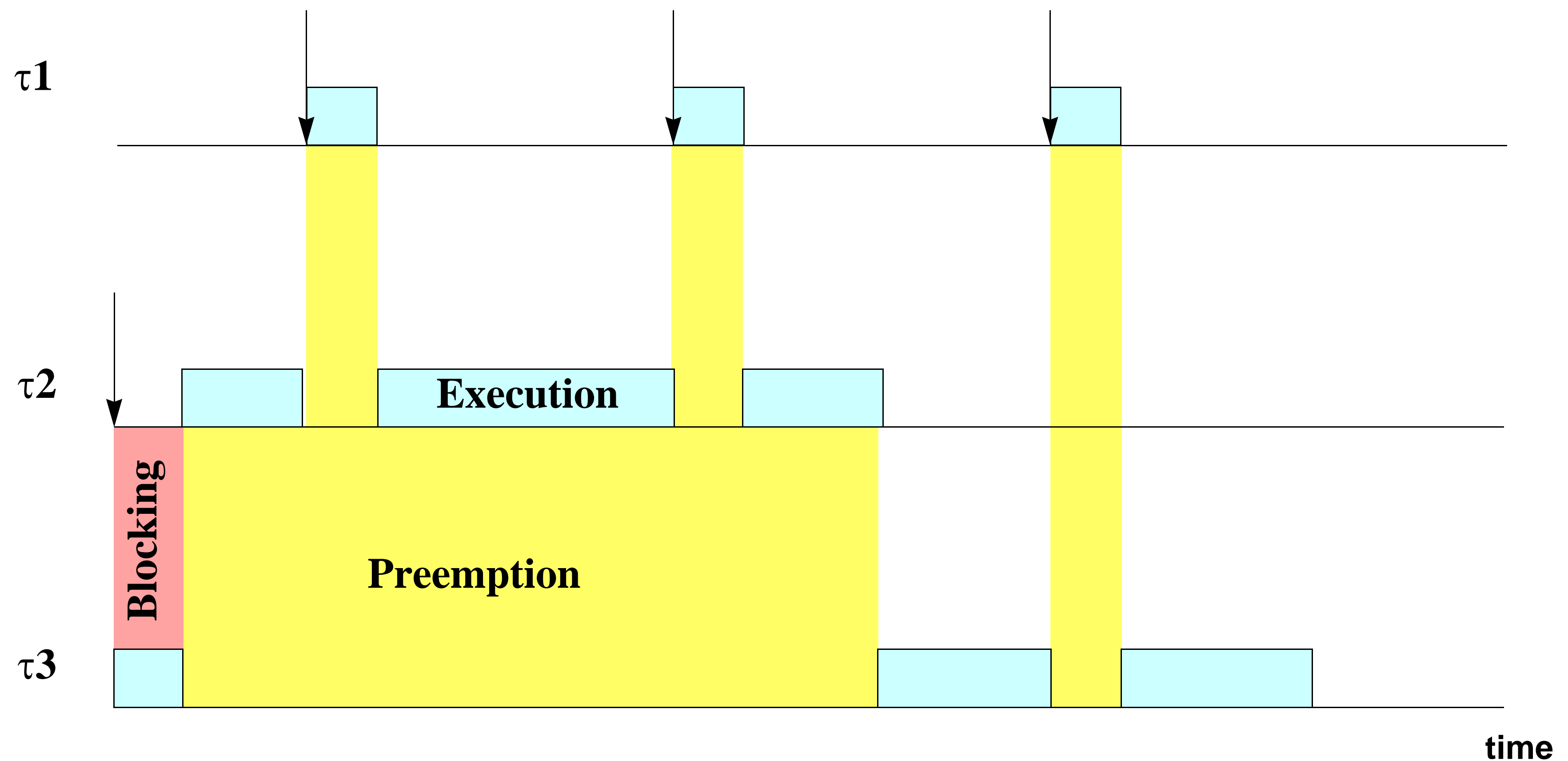
The following are synchronization protocols for fixed priority systems:

- No preemption
- Immediate priority ceiling, or priority protect protocol
- Basic priority inheritance
- Priority ceiling protocol

Each protocol prevents unbounded priority inversion.

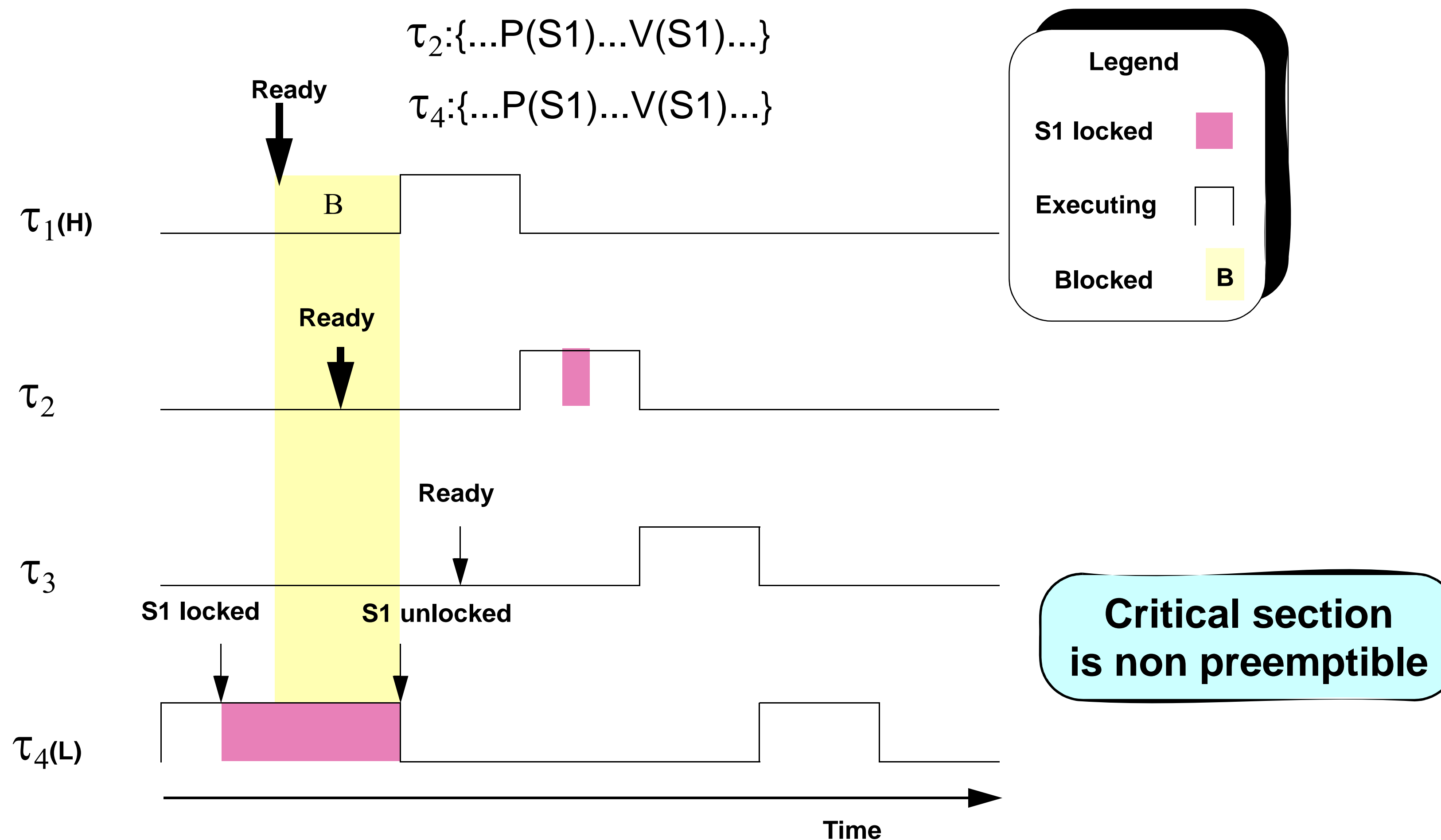
Elements that influence the response time

UNIVERSIDAD
DE CANTABRIA



5.3. Non-Preemption Protocol

UNIVERSIDAD DE CANTABRIA



© José Carlos Palencia, Héctor Pérez Tijero 13/Feb/12

Notes:

UNIVERSIDAD DE CANTABRIA

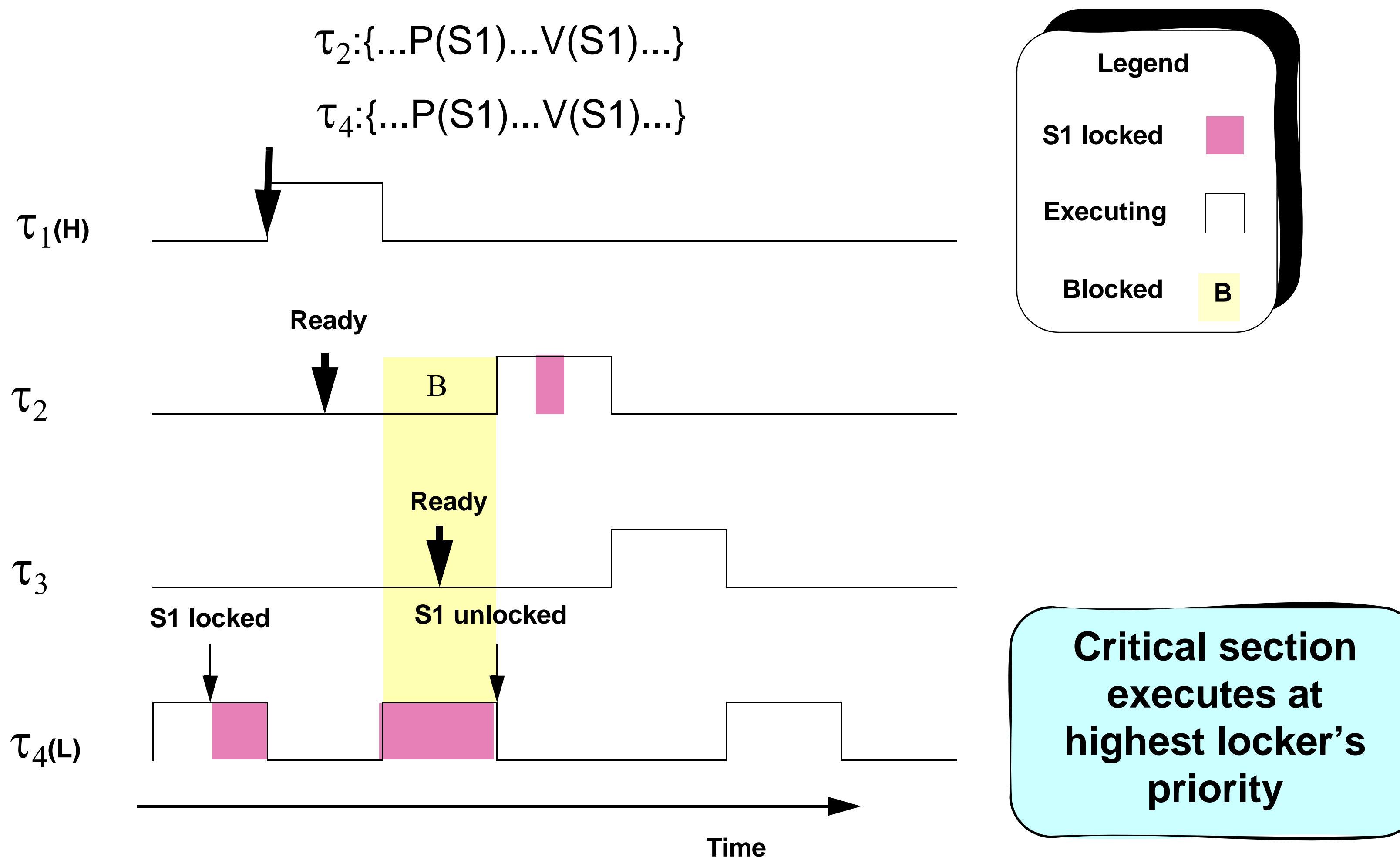
One solution to the unbounded priority inversion problem in a single processor environment is to disallow preemption during the execution of all critical sections. This protocol unnecessarily blocks all high-priority tasks, including those tasks, such as τ_1, which have higher priority than any tasks that share the semaphore.

In this example, when τ_4 is executing its critical section, all other tasks with priority higher than τ_4 will be blocked, some unnecessarily.

- τ_1 is blocked even though it will never try to lock the semaphore.
- τ_2 is blocked even before it tries to lock the semaphore.

Observe that the blocking of τ_3 is necessary; it is the price to be paid to eliminate unbounded priority inversion.

5.4 Immediate Priority Ceiling (IPC)



© José Carlos Palencia, Héctor Pérez Tijero 13/Feb/12

Notes:

Another solution is to execute critical regions using the priority of the highest-priority task that may lock the semaphore. (Arrival of equal-priority tasks should not preempt the currently executing task.)

This protocol has an advantage over the "no preemption" protocol, in that very high-priority tasks that do not lock the semaphore are no longer blocked. Compare this slide to the previous one and note that τ_1 executes immediately. However, note that τ_2 is still blocked, even before it tries to lock the semaphore.

Again, the blocking of τ_3 is necessary to avoid unbounded priority inversion.

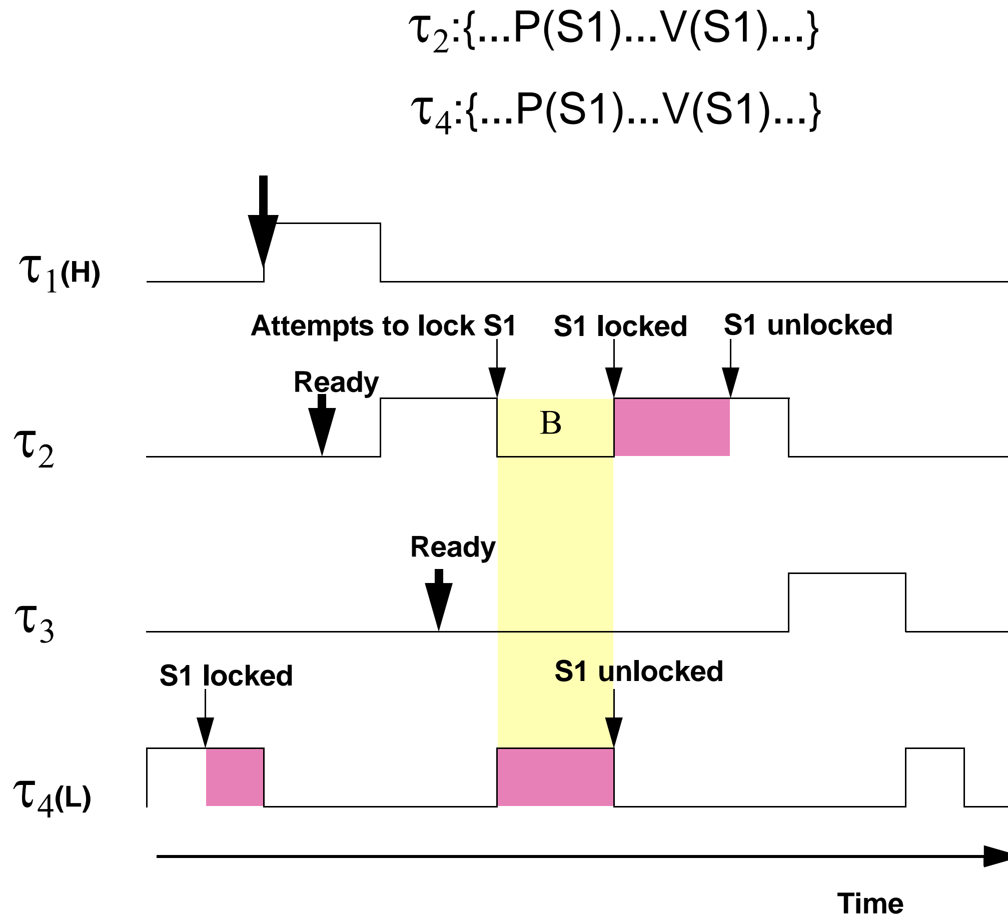
© José Carlos Palencia, Héctor Pérez Tijero 13/Feb/12

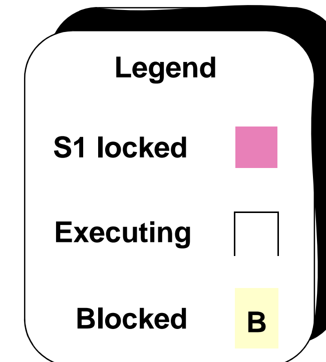
5.5 Basic Inheritance Protocol (BIP)

UNIVERSIDAD DE CANTABRIA

τ_2:{...P(S1)...V(S1)...}

τ_4:{...P(S1)...V(S1)...}





Critical section inherits priority of all tasks which it is blocking

Notes:

UNIVERSIDAD DE CANTABRIA

A third solution is to start out executing the critical section at the priority of the task that locked the semaphore and to increase the execution priority as needed in order to avoid unbounded priority inversion.

The basic inheritance protocol allows lower-priority tasks to temporarily inherit higher priorities: *if* they are executing within a critical section and *if* they are blocking a higher-priority task. This prevents medium-priority tasks from preempting the low-priority task and prolonging the blocking duration experienced by a high-priority task.

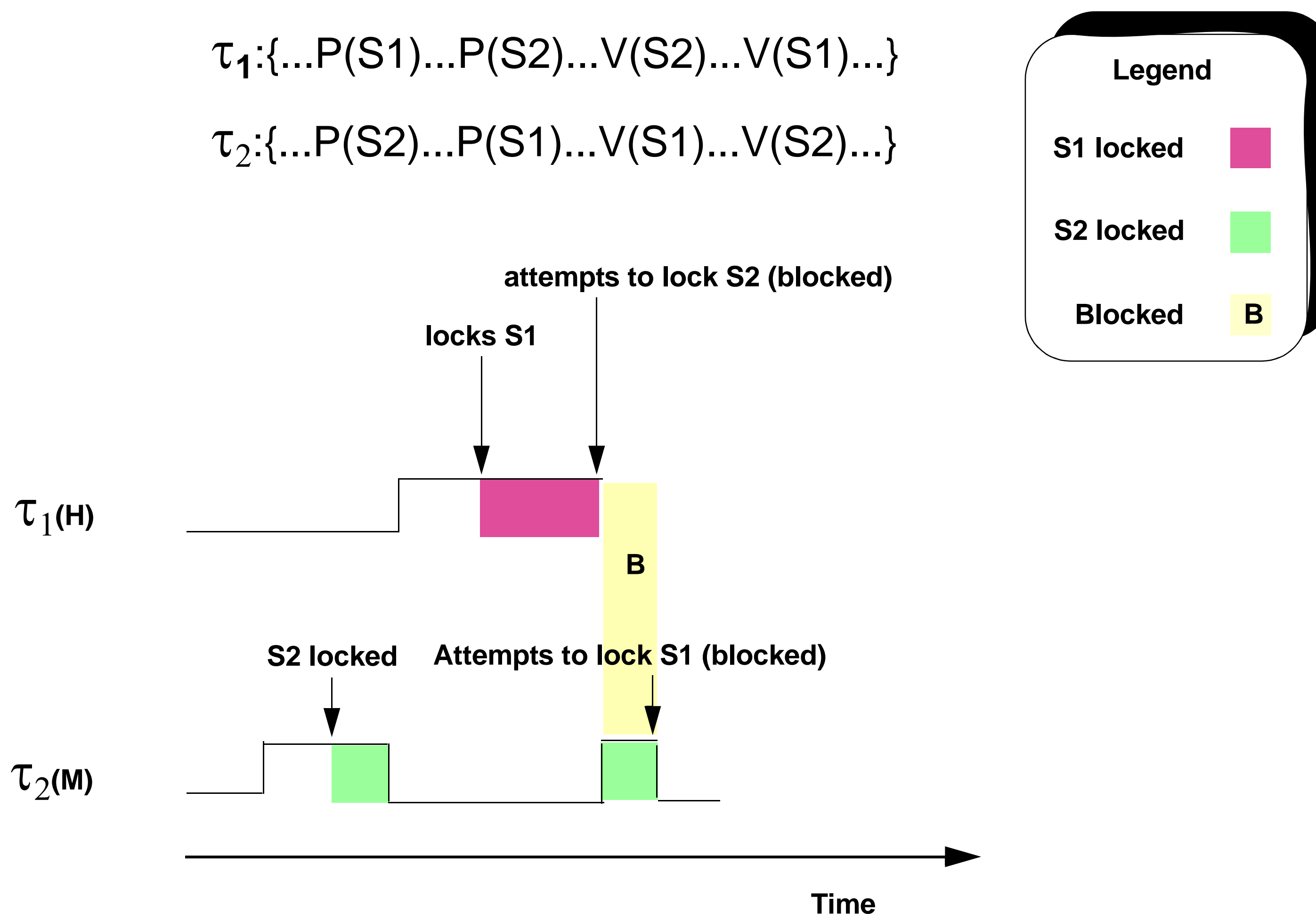
The operation of BIP is shown here:

1. τ_1 preempts τ_4 executing in its critical section; τ_1 executes to completion since it does not need semaphore S1.
2. τ_2 continues preemption of τ_4 (still in its critical section), but τ_2 is blocked when it tries to lock S1; at this point, τ_4 resumes execution, but at τ_2's priority. This effectively prevents τ_3 from executing and delaying τ_2 even longer.
3. When τ_4 unlocks S1, it drops back to its original priority; τ_2 locks S1 and executes.
4. Eventually, τ_2 completes and τ_3 is allowed to run.

This protocol seems to be an improvement, since now τ_2 is not delayed until necessary.

© José Carlos Palencia, Héctor Pérez Tijero 13/Feb/12

Deadlock: Using BIP



Notes:

UNIVERSIDAD DE CANTABRIA

A problem of the BIP protocol is the potential deadlock.

Two tasks τ_1 and τ_2 share two semaphores S1 and S2, which they lock in opposite order. It is possible for τ_1 to lock S1 and τ_2 to lock S2 and for each task to be blocked waiting for the other. This slide shows a possible execution under BIP:

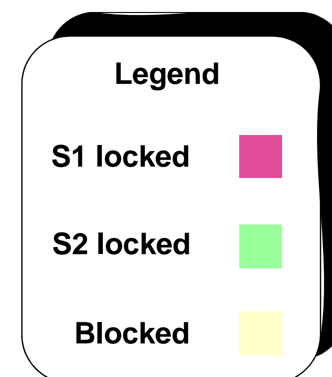
1. τ_2 executes and locks S2.
2. τ_1 preempts τ_2 (in its critical section) and locks S1.
3. Later, τ_1 tries to lock S2 and is blocked.
4. τ_2 inherits τ_1's priority and resumes.
5. Later, τ_2 tries to lock S1 and is blocked.

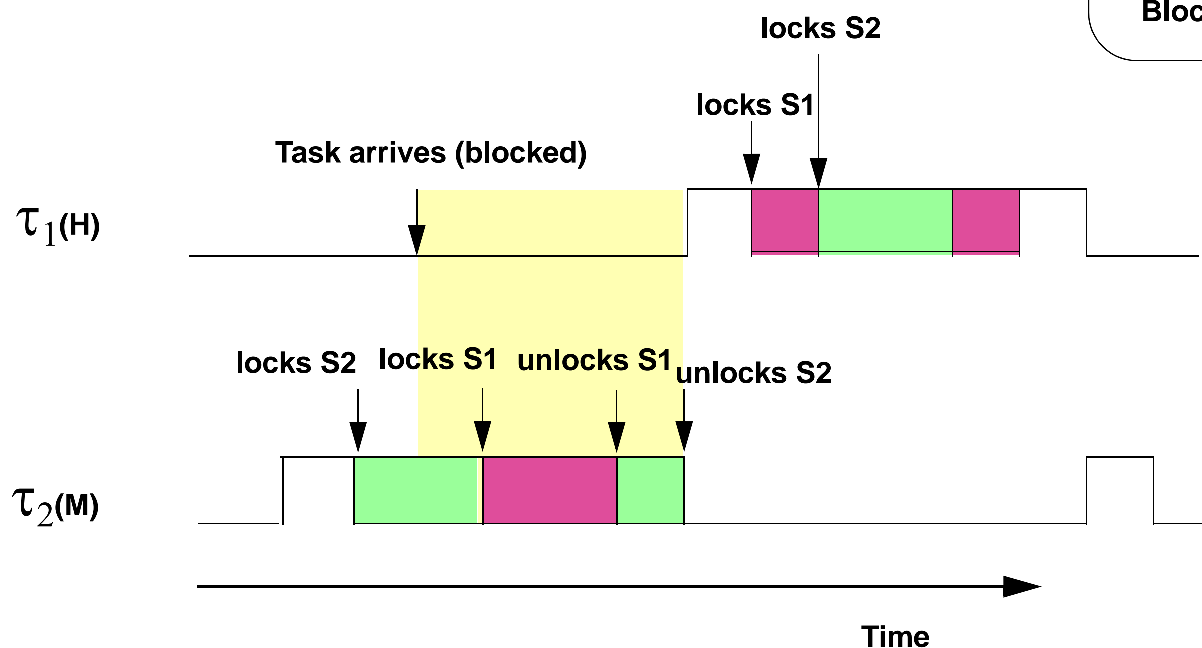
At this point neither task can proceed — the system is deadlocked.

Deadlock Avoidance: Using IPC

τ_1:{...P(S1)...P(S2)...V(S2)...V(S1)...}

τ_2:{...P(S2)...P(S1)...V(S1)...V(S2)...}





© José Carlos Palencia, Héctor Pérez Tijero 13/Feb/12

Notes:

UNIVERSIDAD DE CANTABRIA

This slide shows the deadlock avoidance when IPC is used.

The ceiling of both S1 and S2 is equal to H = τ_1's priority. Execution proceeds as follows:

1. τ_2 begins execution and locks S2. Its priority is raised to the ceiling.
2. τ_1 is activated but cannot preempt τ_2 because it does not have enough priority.
3. τ_2 continues execution; later, τ_2 locks S1; it continues at the same ceiling priority.
4. Eventually, τ_2 unlocks S1 and then S2; τ_2 drops to its original priority.
5. τ_1 can now execute; it is granted the lock on S1 and begins execution of its critical section; later, τ_1 locks S2.
6. Eventually, τ_1 completes and τ_2 resumes.

IPC has avoided the deadlock between τ_1 and τ_2.

5.6 Priority Ceiling Protocol (PCP)

UNIVERSIDAD DE CANTABRIA

The priority ceiling protocol is based on BIP, but uses a different rule for when to grant the lock request on a free semaphore. It avoids the chained blocking and the deadlock.

The ceiling rule: A task cannot lock a free semaphore unless its priority is strictly greater than the "system" ceiling

- **the maximum ceiling of all semaphores currently locked by other tasks**

Notes:

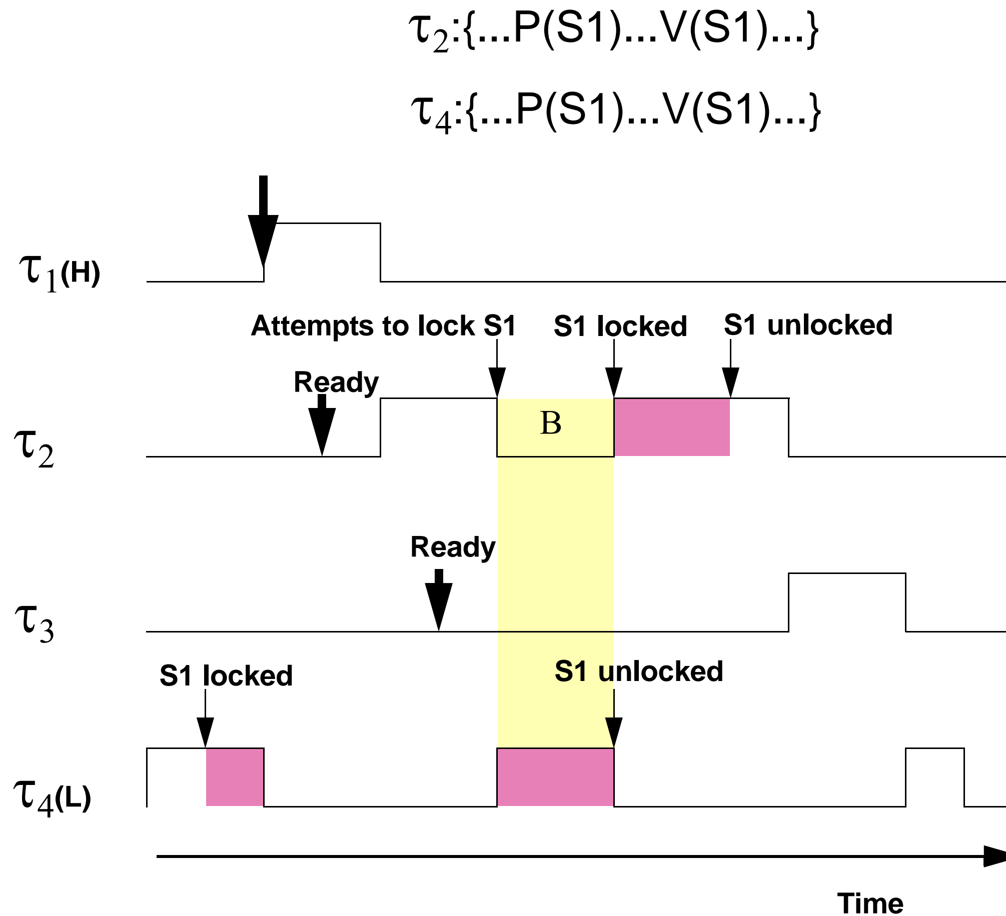
The priority ceiling protocol is based on BIP, but uses a different rule for when to grant a lock request on a free semaphore. The rule is as follows:

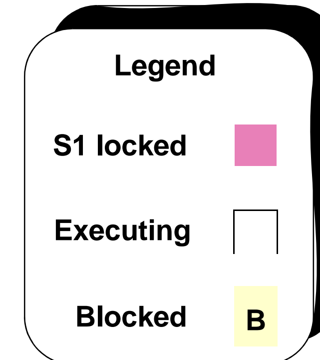
- Ceiling: each semaphore is assigned a priority ceiling equal to the highest locker's priority.
- Ceiling rule: a task cannot lock a free semaphore unless its priority is strictly greater than the "system" ceiling, which is the maximum ceiling of all semaphores currently locked by other tasks.

© José Carlos Palencia, Héctor Pérez Tijero 13/Feb/12

Priority Ceiling Protocol (PCP) (cont.)

UNIVERSIDAD DE CANTABRIA





Basic Inheritance plus the ceiling rule

Notes:

UNIVERSIDAD DE CANTABRIA

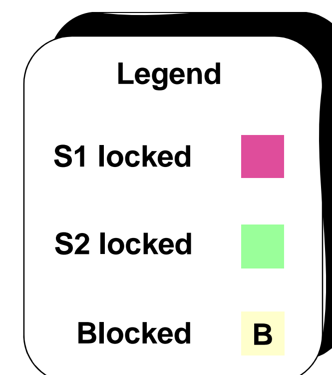
When there is only one semaphore, PCP works just like BIP, and so this diagram looks the same as the last slide. However, the explanation is a bit different:

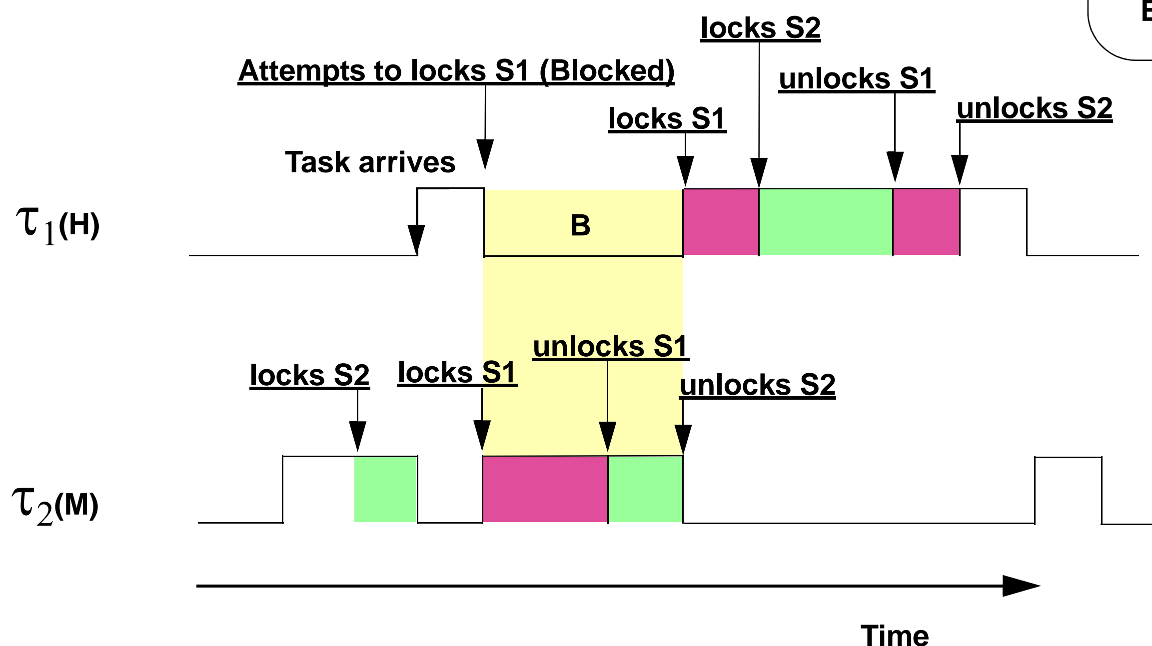
- The ceiling of S1 is equal to the priority of τ_2. When τ_4 locks S1 the “system” ceiling becomes the priority of τ_2.
- τ_1 preempts τ_4 in its critical section, since it does not try to lock S1 (the ceiling rule does not apply).
- τ_2 preempts τ_4 in its critical section, but τ_2 is blocked when it tries to lock S1, since S1 is not free; τ_4 resumes execution at τ_2’s priority; τ_3 cannot preempt τ_4 (executing at τ_2’s priority).
- Eventually, τ_4 unlocks S1 and drops to its old priority; τ_2 locks S1 and executes.

Deadlock Avoidance: Using PCP

τ_1:{...P(S1)...P(S2)...V(S2)...V(S1)...}

τ_2:{...P(S2)...P(S1)...V(S1)...V(S2)...}





Notes:

The priority ceiling protocol has another nice property: it prevents mutual deadlock. This slide show the deadlock avoidance property of PCP.

We note in passing that PCP still works if either τ_1 or τ_2 suspend within a critical section, because the "system" ceiling is not affected in this case.

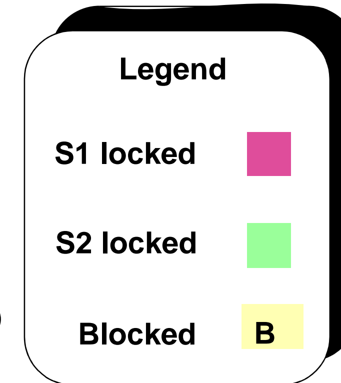
© José Carlos Palencia, Héctor Pérez Tijero 13/Feb/12

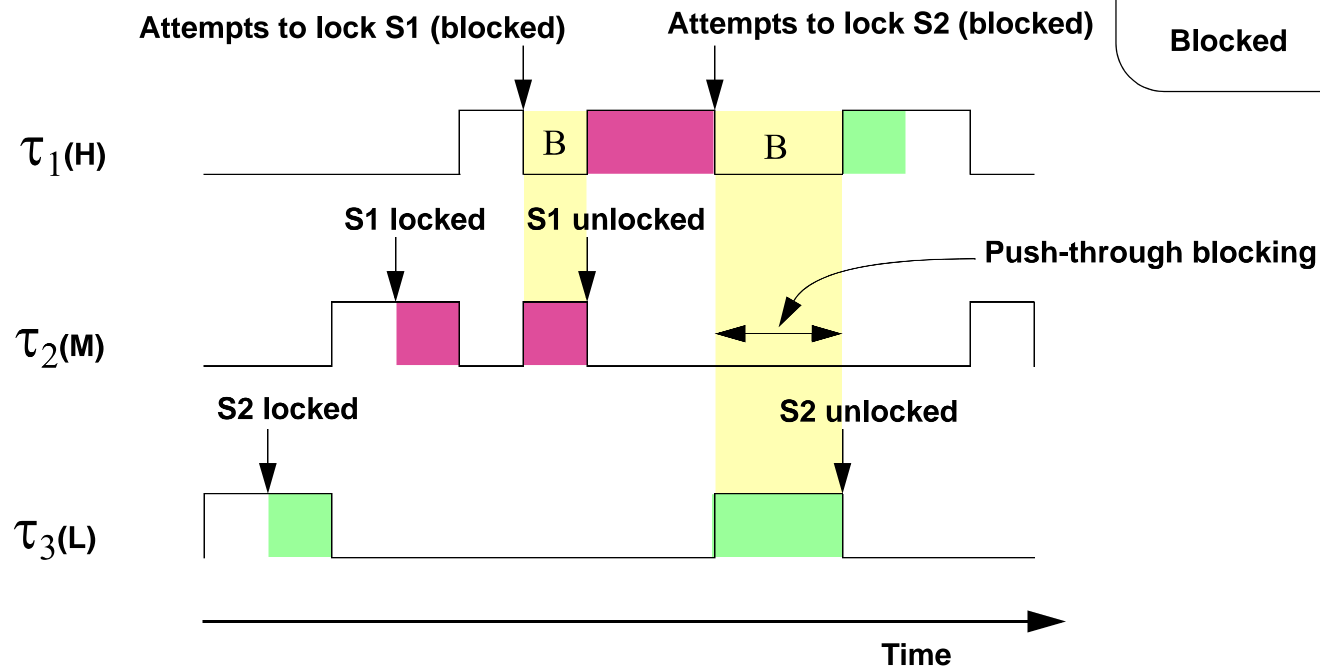
Example Of Chained Blocking (BIP)

τ_1:{...P(S1)...P(S2)...V(S2)...V(S1)...}

τ_2:{...P(S1)...V(S1)...}

τ_3:{...P(S2)...V(S2)...}





Notes:

UNIVERSIDAD DE CANTABRIA

Although BIP solves the unbounded priority inversion problem described in slide , it is still possible for a high-priority task to experience unnecessary blocking. When a high-priority task shares more than one semaphore with lower-priority tasks, it may be blocked on each request to lock a semaphore. We call this phenomenon *chained blocking*.

In this example, τ_1 shares semaphore S1 with τ_2 and semaphore S2 with τ_3. A chain of blocking occurs when τ_1 must first wait to lock S1 and then later must wait to lock S2. The sequence of events is:

1. τ_3 executes and locks S2
2. τ_2 preempts τ_3 executing in its critical region and locks S1.
3. τ_1 preempts τ_2 and executes until it tries to lock S1; τ_1 is blocked.
4. τ_2 resumes, inherits τ_1's priority and executes until it unlocks S1.
5. τ_1 locks S1 and executes until it tries to lock S2; τ_1 is blocked again.
6. τ_3 resumes, inherits τ_1's priority and executes until it unlocks S2.
7. τ_1 resumes and completes; eventually τ_2 and then τ_3 complete.

© José Carlos Palencia, Héctor Pérez Tijero

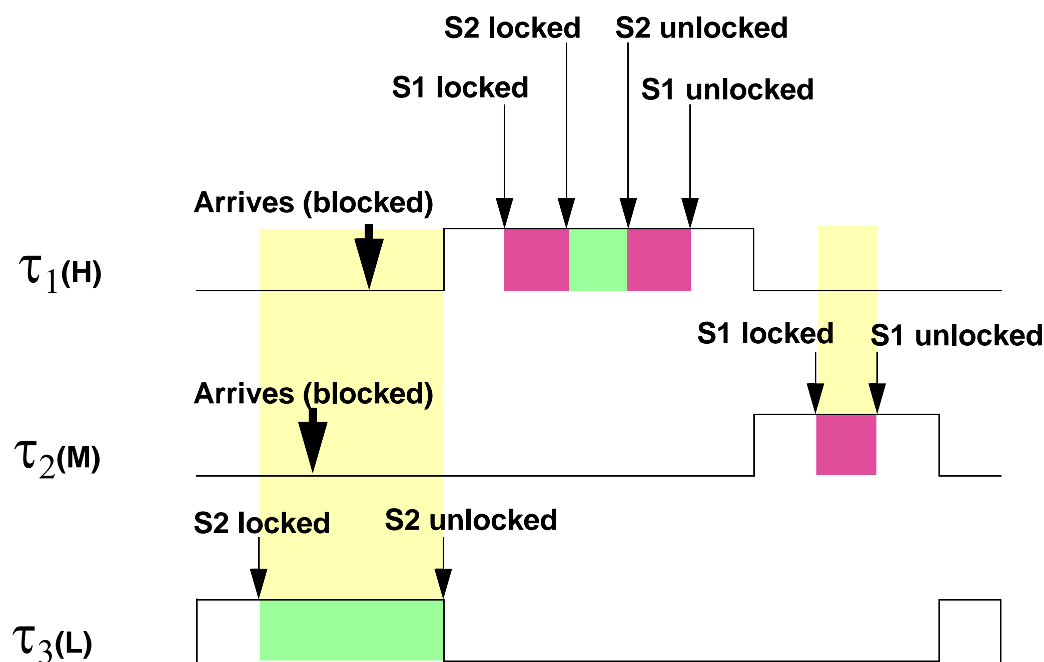
Blocked At Most Once (IPC)

UNIVERSIDAD DE CANTABRIA

τ_1:{...P(S1)...P(S2)...V(S2)...V(S1)...}

τ_2:{...P(S1)...V(S1)...}

τ_3:{...P(S2)...V(S2)...}



© José Carlos Palencia, Héctor Pérez Tijero 13/Feb/12

Notes:

UNIVERSIDAD DE CANTABRIA

This slide shows the same situation as slice 17, but now the immediate priority ceiling protocol is being used. First, note that the ceilings of both semaphores are equal to H, the priority of τ_1. The sequence of events is:

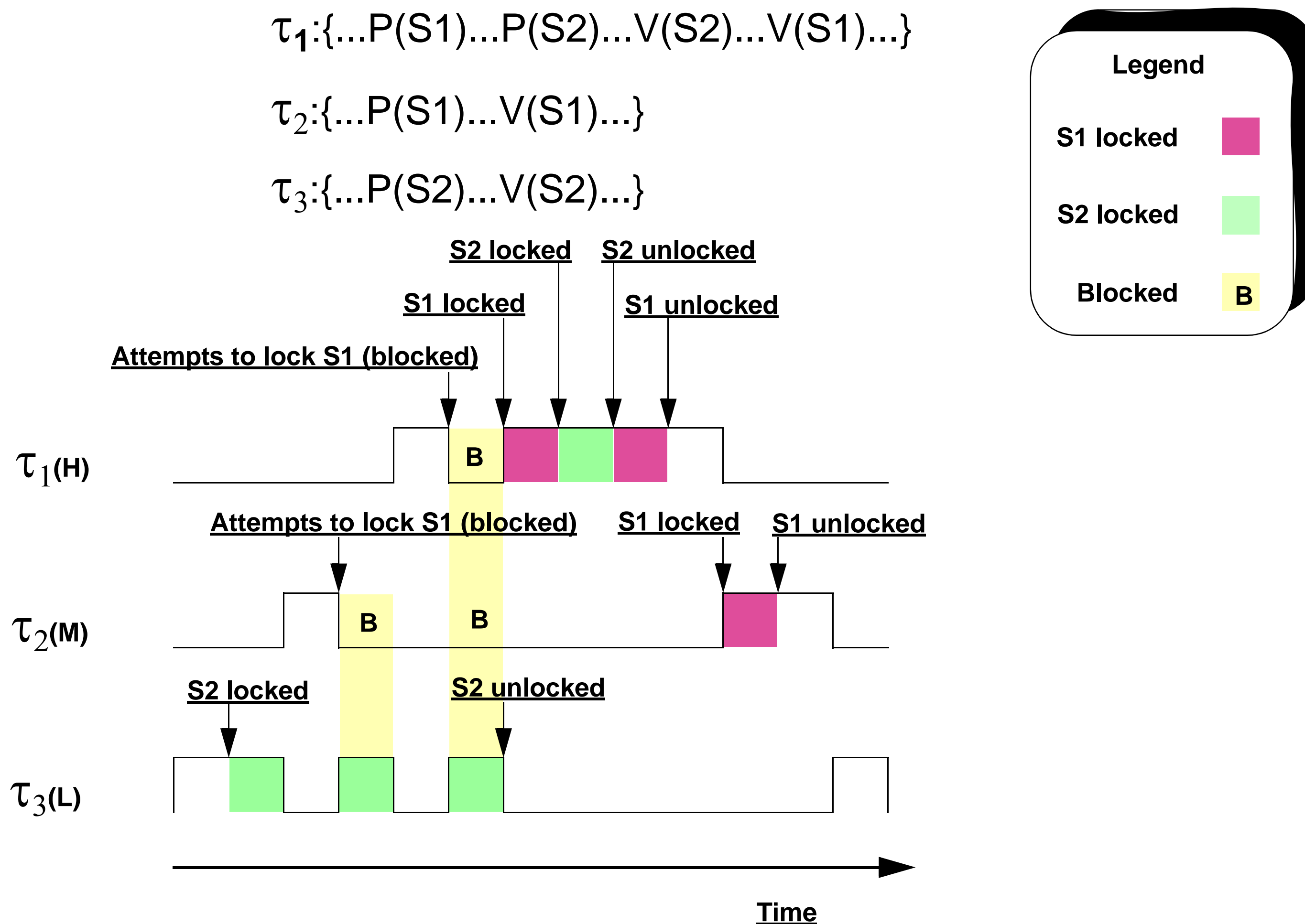
1. τ_3 executes and locks S2; at this point its priority rises to the ceiling
2. When τ_2 tries execute, it can't because it does not have enough priority. Therefore it is prevented from locking any other semaphore.
3. When τ_1 tries to execute it can't, because it does not have enough priority
4. When τ_3 unlocks S2, then τ_1 can proceed. When it tries to lock semaphores S1 and S2 it finds them unlocked, so it is not blocked.

Several observations are worth making:

- A task may only get delayed at the beginning, for one critical section at most.
- When a task starts executing, all the semaphores that it needs are unlocked.
- These properties eliminate chained blocking, and reduce the number of context switches. In fact, there are no context switches caused by synchronization under this protocol.

Blocked At Most Once (PCP)

UC UNIVERSIDAD DE CANTABRIA



Notes:

UNIVERSIDAD DE CANTABRIA

This slide shows the same situation as slice 27, but now the priority ceiling protocol is being used. First, note that the ceilings of both semaphores are equal to H, the priority of τ_1. The sequence of events is:

1. τ_3 executes and locks S2; at this point the “system” ceiling is H = priority of τ_1.
2. When τ_2 tries to lock S1, the ceiling rule prevents the lock, since τ_2's priority is not greater than H. τ_3 inherits τ_2's priority and continues execution.
3. When τ_1 tries to lock S1, the ceiling rule prevents the lock, since τ_1's priority is not greater than H.
4. τ_3 inherits τ_1's priority and continues execution. Eventually, τ_3 unlocks S2; τ_1 is allowed to resume and to lock S1. Later, τ_1 is granted an immediate lock on S2 and executes to completion.

Several observations are worth making:

- A task trying to lock a chain of semaphores is blocked until all lock requests can be granted (so the ceiling rule is applied when the task has not yet locked any semaphores).
- Once the task is allowed to lock the first semaphore, it executes to completion (with no further checks on semaphore state needed). These properties should result in fewer context switches and lower overhead, and should eliminate chained blocking.

© José Carlos Palencia, Héctor Pérez Tijero 13/Feb/12

Context Switches Caused by Synchronization

UNIVERSIDAD DE CANTABRIA

In a single processor, there are no context switches due to synchronization for the non preemption or the IPC protocols

- the blocking time occurs before the task starts executing its job
- once the task has started, if there is no suspension inside the critical sections, the task finds all the resources it needs unlocked
- there is no context switch overhead

For the BIP or PCP protocols there are suspensions caused by synchronization

- we have to add two context switches to each critical section contributing to the blocking term
    - one critical section for PCP, possibly a few for BIP

© José Carlos Palencia, Héctor Pérez Tijero 13/Feb/12

Summary of properties

UNIVERSIDAD DE CANTABRIA

Dynamic systems (no precalculated ceilings):

- *BIP* has less blocking than Non-Preemption

Static systems (ceilings are precalculated):

- *IPC* has less overhead than PCP
- Critical sections are designed to not suspend themselves
  - this is desirable in any case

© José Carlos Palencia, Héctor Pérez Tijero

5.7 Comparison of Synchronization Protocols

UNIVERSIDAD DE CANTABRIA

Protocol	Non preemption	IPC	BIP	PCP
Requires precalculating ceilings	No	Yes	No	Yes
Blocked at most once	Yes[2]	Yes[2]	No	Yes[2]
Avoids deadlock	Yes[1]	Yes[1]	No	Yes
Avoids unnecessary blocking	No	Yes	Yes	Yes
Worst-case context switch overhead	0	0	$2C_s$ per critical section	$2C_s$

[1] **Only if tasks do not suspend within critical sections**
[2] **Only if tasks do not suspend**

Notes:

UNIVERSIDAD
DE CANTABRIA

This slide summarizes some of the important properties of the four synchronization protocols introduced in this section. All four protocols eliminate unbounded priority inversion by preventing intermediate-priority tasks from preempting critical sections of lower-priority tasks.

The first two protocols as well as the priority ceiling protocol exhibit the blocked-at-most-once property, but *only* if tasks do not suspend (e.g., by waiting on input/output completion). The reason is that when tasks suspend themselves, other lower-priority tasks may run and lock free semaphores that may be needed later by the suspended task. The three protocols show the blocked-at-most-once property for each contiguous segment of execution.

The first two protocols and PCP also have the property of deadlock avoidance. However, for the first two protocols, deadlock may not be avoided if the tasks suspend themselves inside critical sections. Note that PCP is not affected when tasks suspend during critical regions. Should any other task attempt to lock semaphores that are needed later by the suspended task, tasks which suspend during critical regions retain any semaphore locks, and the ceiling rule continues to apply.

Basic inheritance allows chained blocking (see slide) and does not avoid deadlock (see slide), but does tend to minimize unnecessary blocking. The priority ceiling protocol (PCP) exhibits the many desirable properties, but at the cost of a little overhead to keep track of the “system” ceiling.

5.8 Analyzing the Blocking Times

UNIVERSIDAD DE CANTABRIA

For the no preemption, immediate priority ceiling, or priority ceiling protocols:

- B_i is the maximum of all the critical sections of lower priority tasks whose ceiling is $\geq P_i$

For the basic priority inheritance protocol:

- The critical sections that may cause blocking are those of lower priority tasks whose ceiling is $\geq P_i$; ceiling may be affected by transitive or recursive inheritance.
- Only one independent (non overlapped) critical section per semaphore may cause blocking (pick the longest)
- Only one independent critical section per lower priority task may cause blocking (pick the longest)

Notes:

UNIVERSIDAD DE CANTABRIA

The rule for calculating the blocking term when the no preemption, immediate priority ceiling, or priority ceiling protocols are being used is very simple, because of the blocked-at-most-once property:

- The blocking term is the maximum duration of all the critical sections of lower priority tasks whose priority ceiling is higher than or equal to the priority of the task under analysis.

When the basic priority inheritance protocol is being used, it is usually very difficult to calculate the blocking term exactly. However, an upper bound is relatively easy to obtain:

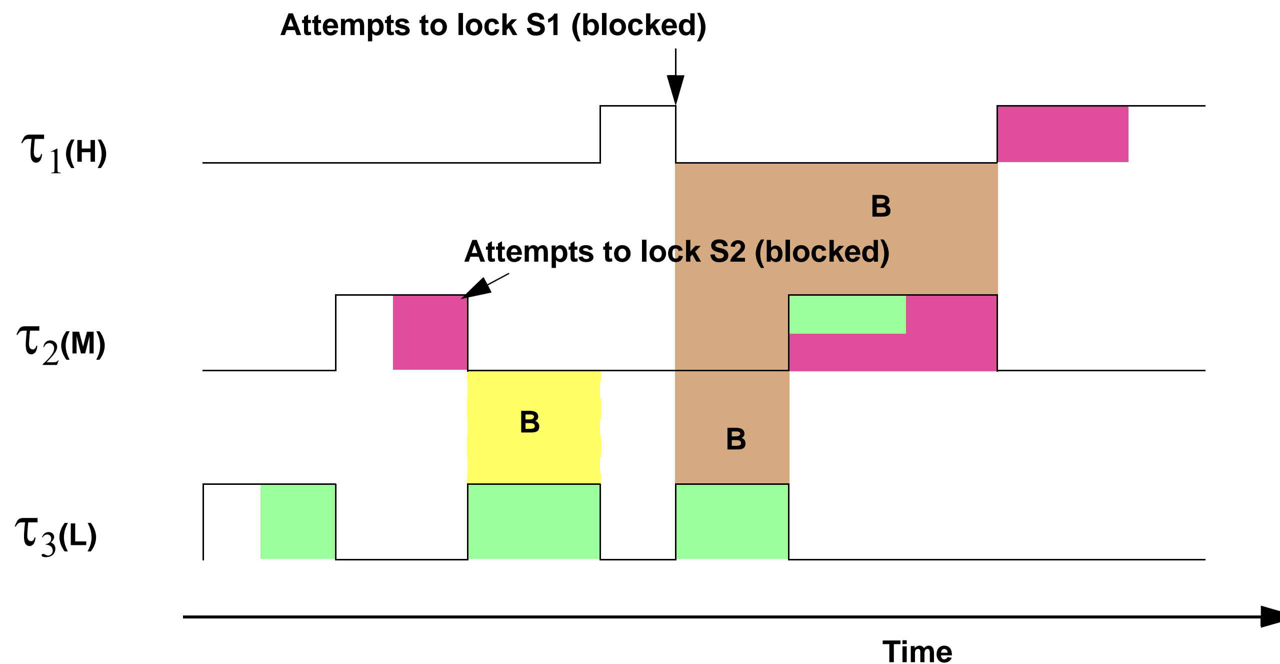
- The critical sections that may affect a task under analysis are those whose priority ceilings are higher than or equal to the priority of that task. But notice that the priority ceiling of a critical section may be higher than just the priorities of the tasks that may lock the associated resource. If BIP is being used, **transitive blocking** may occur, which may increase the priority ceiling of a critical section.
- For each resource, only one independent critical section relative to that resource may affect the task under analysis. Notice however that this applies only to independent critical sections, i.e, those that are not overlapped or nested with other critical sections.
- For each lower priority task, only one independent critical section may affect the task.

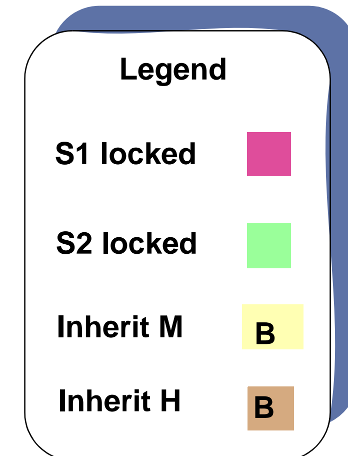
Transitive or recursive blocking (BIP).

τ_1:{...P(S1)...V(S1)...}

τ_2:{...P(S1)...P(S2)...V(S2)...V(S1)...}

τ_3:{...P(S2)...V(S2)...}





Notes:

UNIVERSIDAD DE CANTABRIA

- Transitive blocking occurs when a task t2 that is blocked waiting for another task (t3) to unlock a resource is itself holding a resource that a third higher priority task (t1) needs. In this case, the lower priority task (t3) may inherit the priority of the higher priority task (t1) via transitive inheritance through the medium priority task (t2).

Sample Problem: Using BIP

UNIVERSIDAD DE CANTABRIA

	C	T	D	B
τ_1	20	100	100	30[1]
τ_2	40	150	130	10[2]
τ_3	100	350	350	0

[1] **Task τ_1 suffers chained blocking**

[2] **Task τ_2 suffers "push-through blocking"**

© José Carlos Palencia, Héctor Pérez Tijero 13/Feb/12

Notes:

UNIVERSIDAD DE CANTABRIA

This slide applies the basic inheritance protocol to our sample problem (slide) to determine the blocking effects on τ_1 and τ_2 (by definition τ_3 cannot be blocked, since it has the lowest priority).

We observe that τ_1 can be blocked by τ_2 for at most 20 msec (because of the shared data server); τ_1 can be blocked by τ_3 for at most 10 msec (because of the shared communication server); and τ_2 can be blocked indirectly by τ_3 for at most 10 msec. (When τ_3 blocks τ_1, τ_3 inherits τ_1's priority, so τ_3 can then indirectly block τ_1.)

The blocking term for τ_1 is the sum $20 + 10 = 30$, since chained blocking can occur.

Schedulability Using BIP

UNIVERSIDAD DE CANTABRIA

Workload equations

$$W_1(t) = C_1 + B_1$$

$$W_2(t) = \left\lceil \frac{t}{T_1} \right\rceil C_1 + C_2 + B_2$$

$$W_3(t) = \left\lceil \frac{t}{T_1} \right\rceil C_1 + \left\lceil \frac{t}{T_2} \right\rceil C_2 + C_3$$

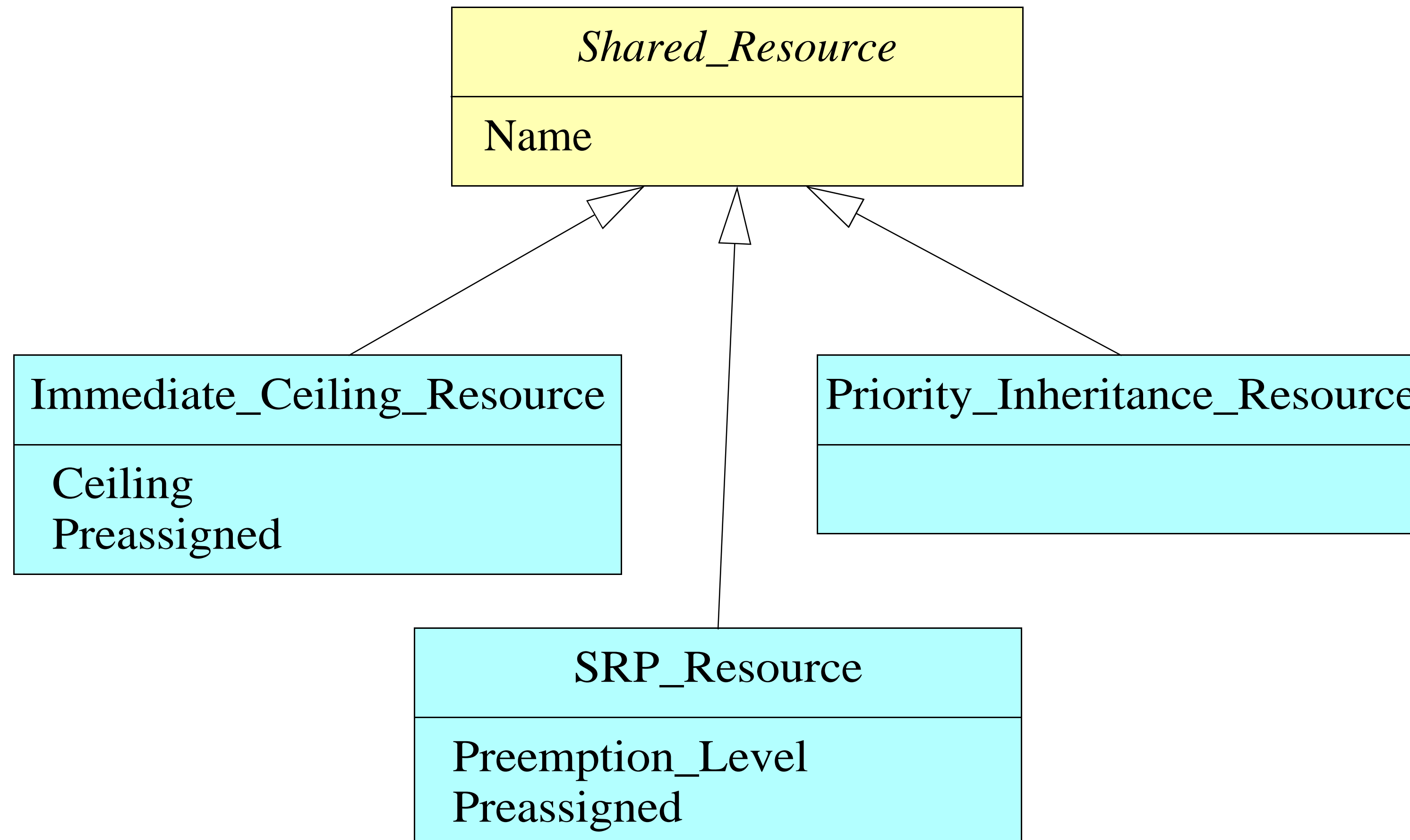
Response time analysis results:

- R_1=50; R_2=70; R_3=240

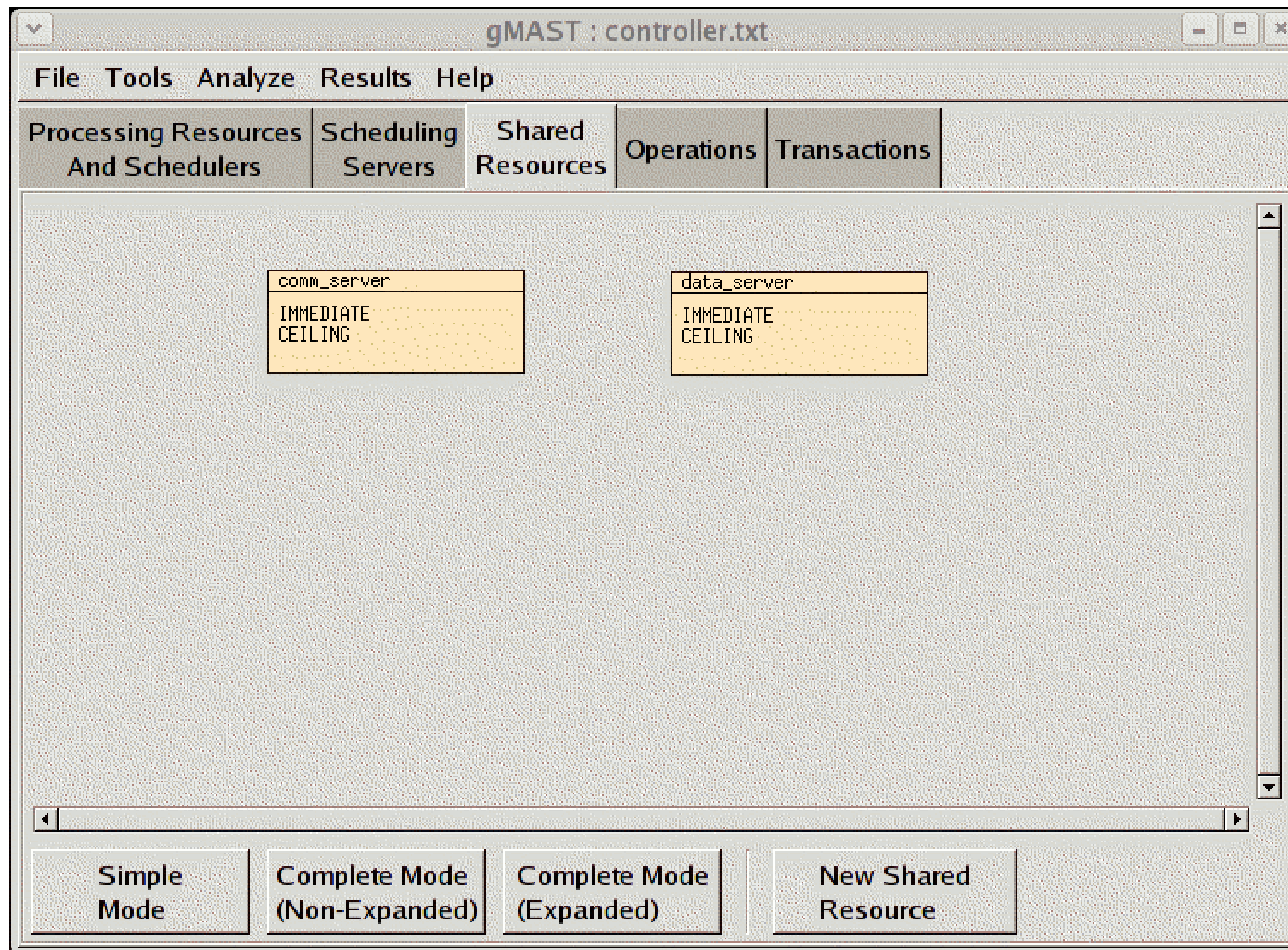
© José Carlos Palencia, Héctor Pérez Tijero 13/Feb/12

UNIVERSIDAD DE CANTABRIA

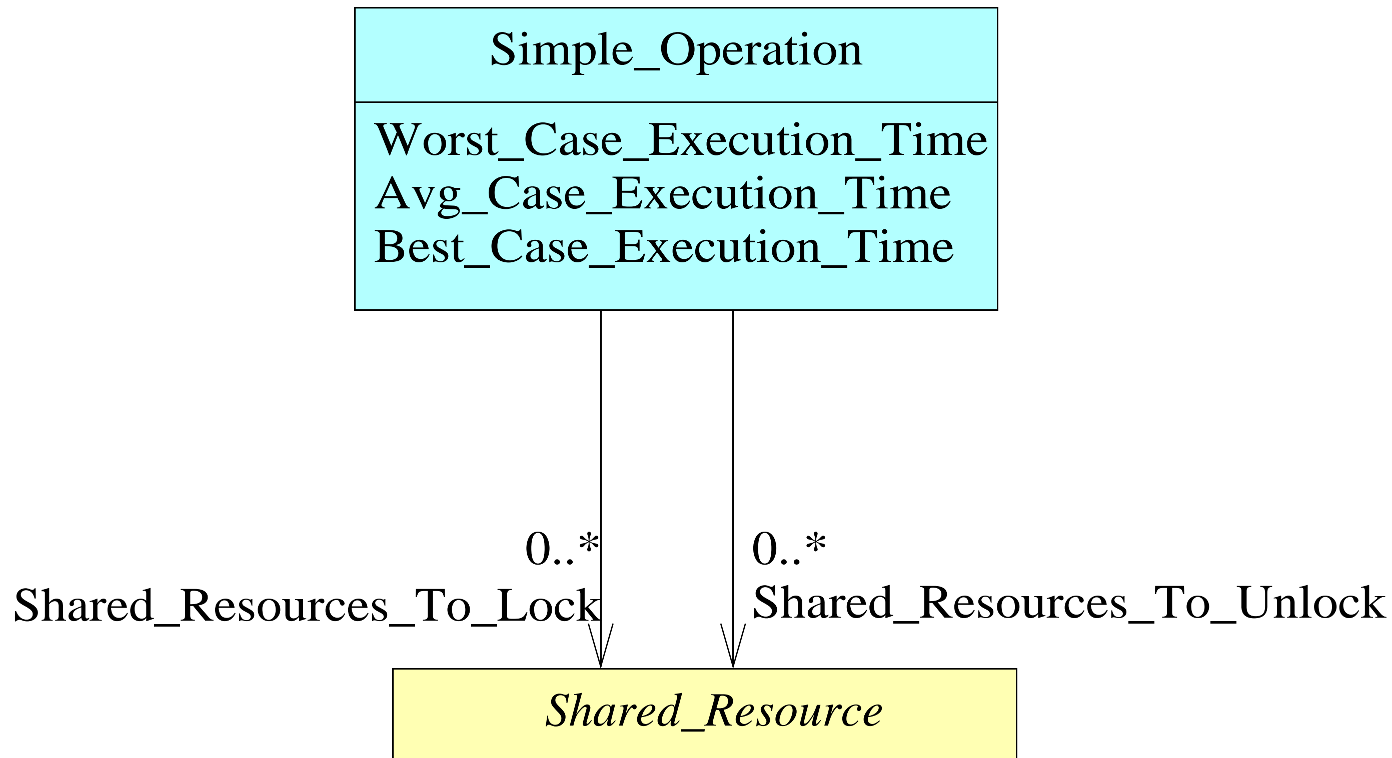
5.9 Modelling synchronization: Shared Resources (mutually exclusive)



Shared resources



Operations use shared resources



5.10. Lessons learned from basic real-time theory

UNIVERSIDAD DE CANTABRIA

In fixed-priority systems, the worst-case response time of a task is a function of:

- ***preemption*: time waiting for higher-priority tasks**
- ***execution*: time to do its own work**
- ***blocking*: time delayed by lower-priority tasks, usually because of mutual exclusion synchronization**

Deadlines are missed when utilization is over 100% (seems obvious, but happens in reality):

- **Solutions:**
  - **change the computation requirements**
  - **change the periods or interarrival times**
  - **buy a faster CPU**

© José Carlos Palencia, Héctor Pérez Tijero 13/Feb/12

Lessons learned from basic real-time theory (cont'd)

Deadlines are missed because there is too much preemption:

- A task or a task's portion runs at a higher priority than it should
  - this happens typically with interrupt service routines
- There are not enough priorities
- Solutions:
  - preemption can be minimized by choosing an optimum priority assignment
  - sometimes a task or an ISR must be split into parts whose priorities can be independently assigned
  - buy a system with user-defined priorities for I/O drivers
  - buy a system with enough priority levels

Lessons learned from basic real-time theory (cont'd)

Deadlines are missed because there is too much blocking:

- Normal semaphores suffer from unbounded priority inversion
  - this causes horrendous worst-case blocking times
- Solutions
  - disable preemption in the critical sections (may still have too much blocking)
  - priority inheritance protocol
  - immediate priority ceiling protocol
  - split long critical sections

UNIVERSIDAD
DE CANTABRIA

© José Carlos Palencia, Héctor Pérez Tijero 13/Feb/12