MSE | MASTER OF SCIENCE IN ENGINEERING

MA_EmbReal

Dependent Tasks

Version: 1.5

Serge Ayer – Luca Haab | 28.04.2025 | Cours MSE

Recall: Our Mission

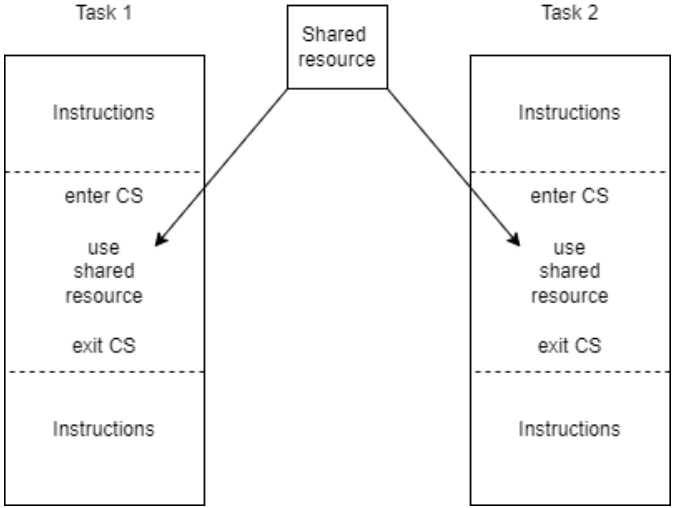
Recall: Our mission

- Program with a mix of periodic / aperiodic tasks
  - Address first scheduling of periodic tasks
  - Add aperiodic tasks
  - **Add dependencies among tasks**
- Demonstrate that a schedule is feasible given a set of tasks with their constraints and dependencies
  - Use known bounds and elaborate a feasible schedule
  - Compute bounds for blocking times
- Use the appropriate scheduling algorithm in simulation and practice
- Implement a system that meets timing constraints
  - With functional safety concepts
  - With timing constraints watchdogs

Dependent Tasks

What is the concern?

- Multi-tasking program often require to share resources among tasks
- Access to shared resources needs to be protected against concurrent access
  - Mutual exclusion among competing tasks
  - A piece of code executed under mutual exclusion is a critical section

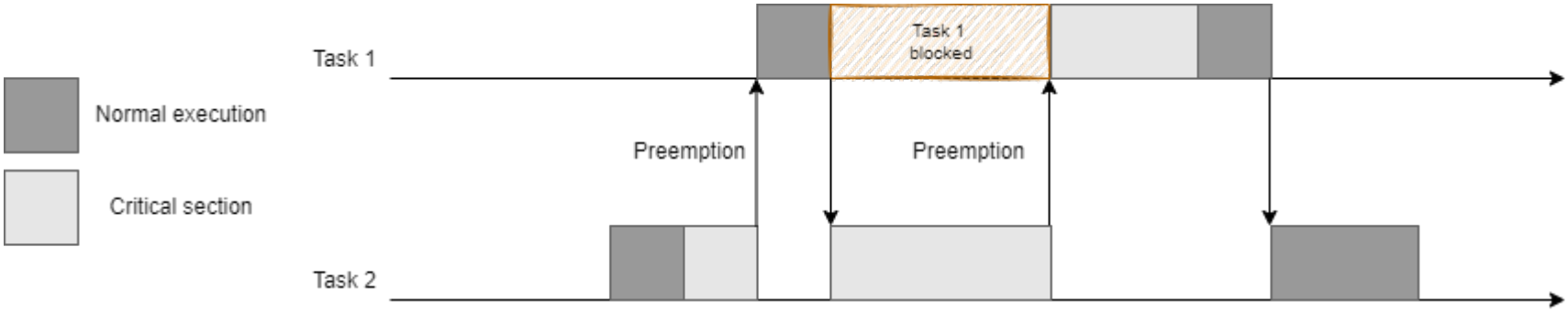


Priority Inversion

- Until now, we have considered that tasks are independent, sharing no resources, and not interacting. What if we remove this assumption?
  - Scheduling of tasks is affected!
- One main problem arises: priority inversion
  - High priority tasks may be blocked by lower priority tasks
- The main sources of priority inversion are
  - Non preemptable sections
  - Sharing resources
  - Synchronization and mutual exclusion
- In all cases, the response time (latencies) are modified

Blocking of higher priority tasks

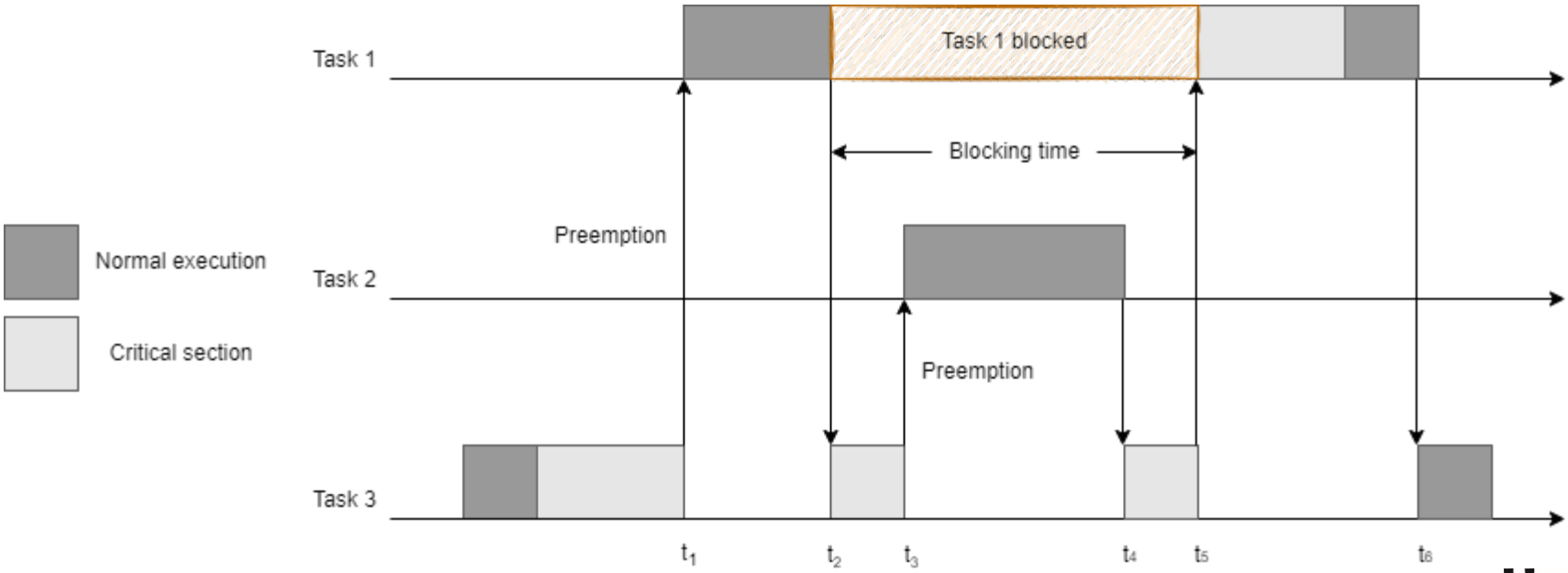
Tasks of higher priority can be blocked by lower priority tasks



Note: in this slide deck, tasks with smaller indices have higher priorities

Unlimited blocking time (priority inversion)

Blocking time of higher priority tasks may be unbounded



Solutions to Priority Inversion

- Tasks are ’forced’ to follow certain rules when locking and unlocking a mutex
  - This is about requesting and releasing resources.
- The rules are often called Resource Access Protocols
  - There are several existing such protocols
- What about the RTX scheduling algorithm?
  - RTX implements the priority inheritance mechanism

Resource Access Protocols

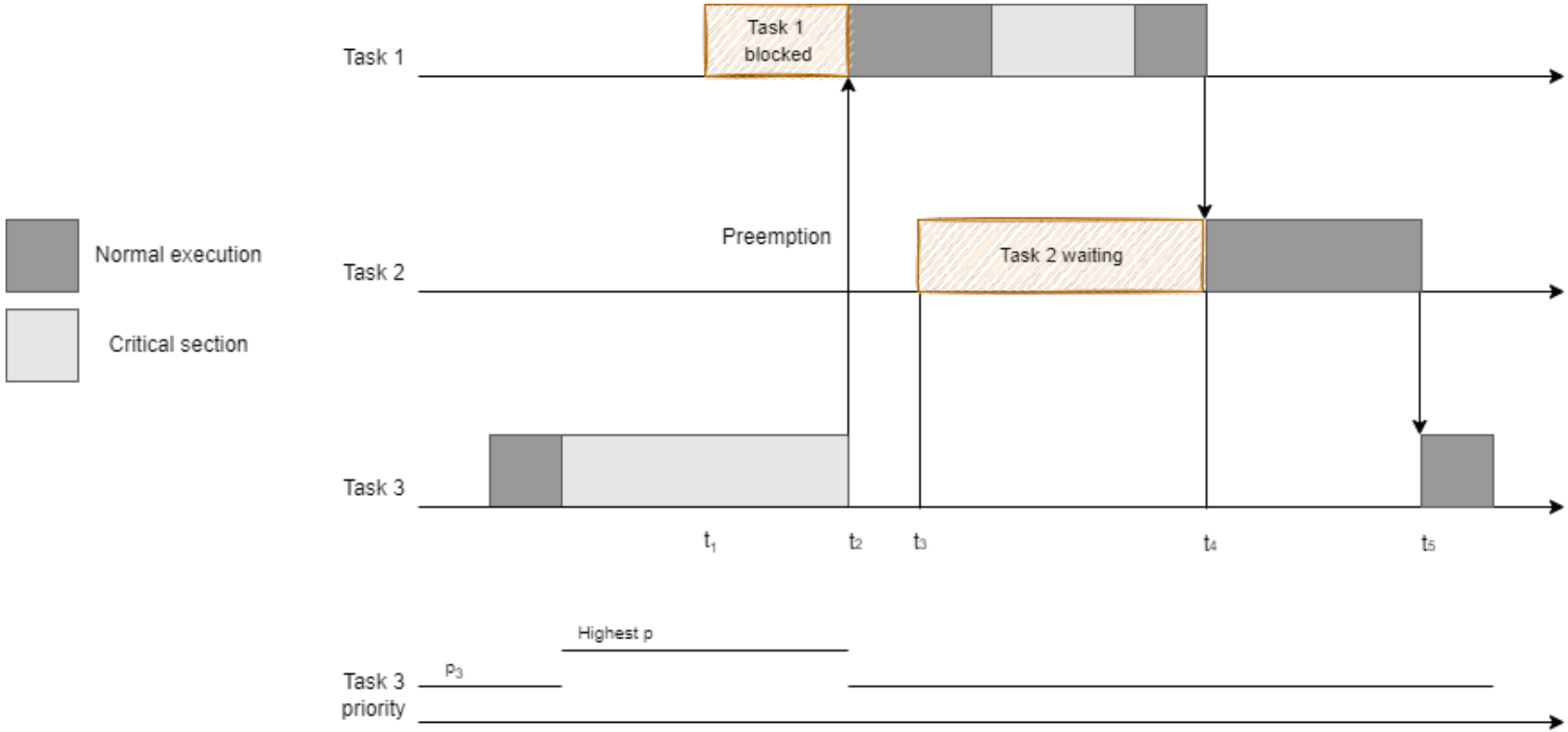
- Need to consider the following points
  - Is the priority inversion bounded?
  - Does the protocol avoid deadlock?
  - Does the protocol avoid unnecessary blocking?
  - Is it easy to calculate the blocking time upper bound?
  - What is the maximum number of blocking?
  - Is it easy to implement?

Non-Preemptive Protocol (NPP)

Principle: disallow pre-emption during the execution of any critical section

- A task is assigned the highest priority if it succeeds in locking a critical section
- The task is assigned its own priority when it releases the critical section

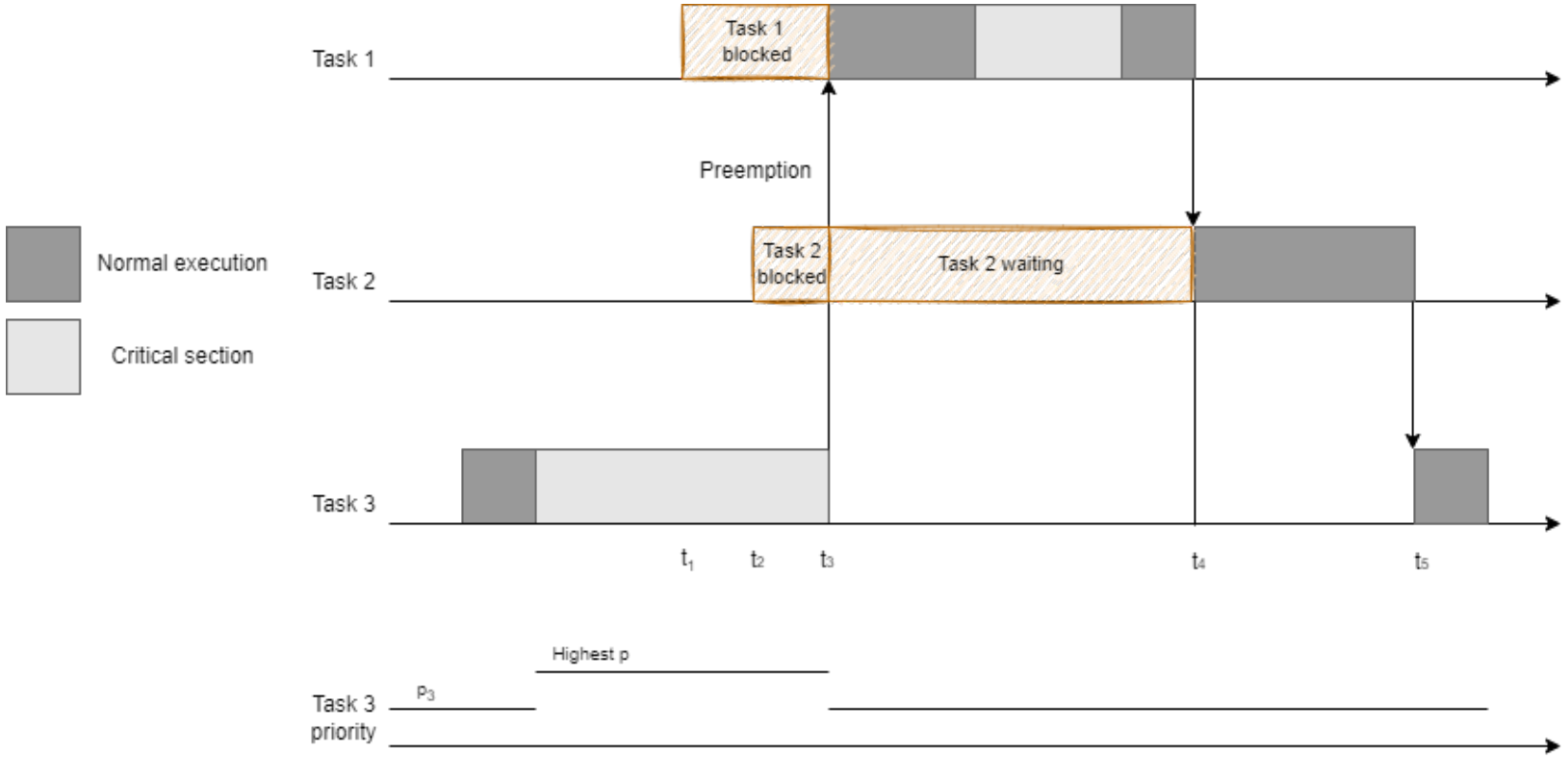
Non-Preemptive Protocol (NPP)



Non-Preemptive Protocol (NPP)

- Advantages:
  - This bounds priority inversion:
    - For a given task, the bound is the maximal length of any single critical section belonging to lower priority tasks.
  - It limits the number of blocking of any task to one
  - It is deadlock free
  - Implementation is easy and transparent
- But:
  - It introduces unnecessary blocking:
    - Low priority tasks may block high priority tasks including those that do not require access to shared resources.

Non-Preemptive Protocol (NPP)



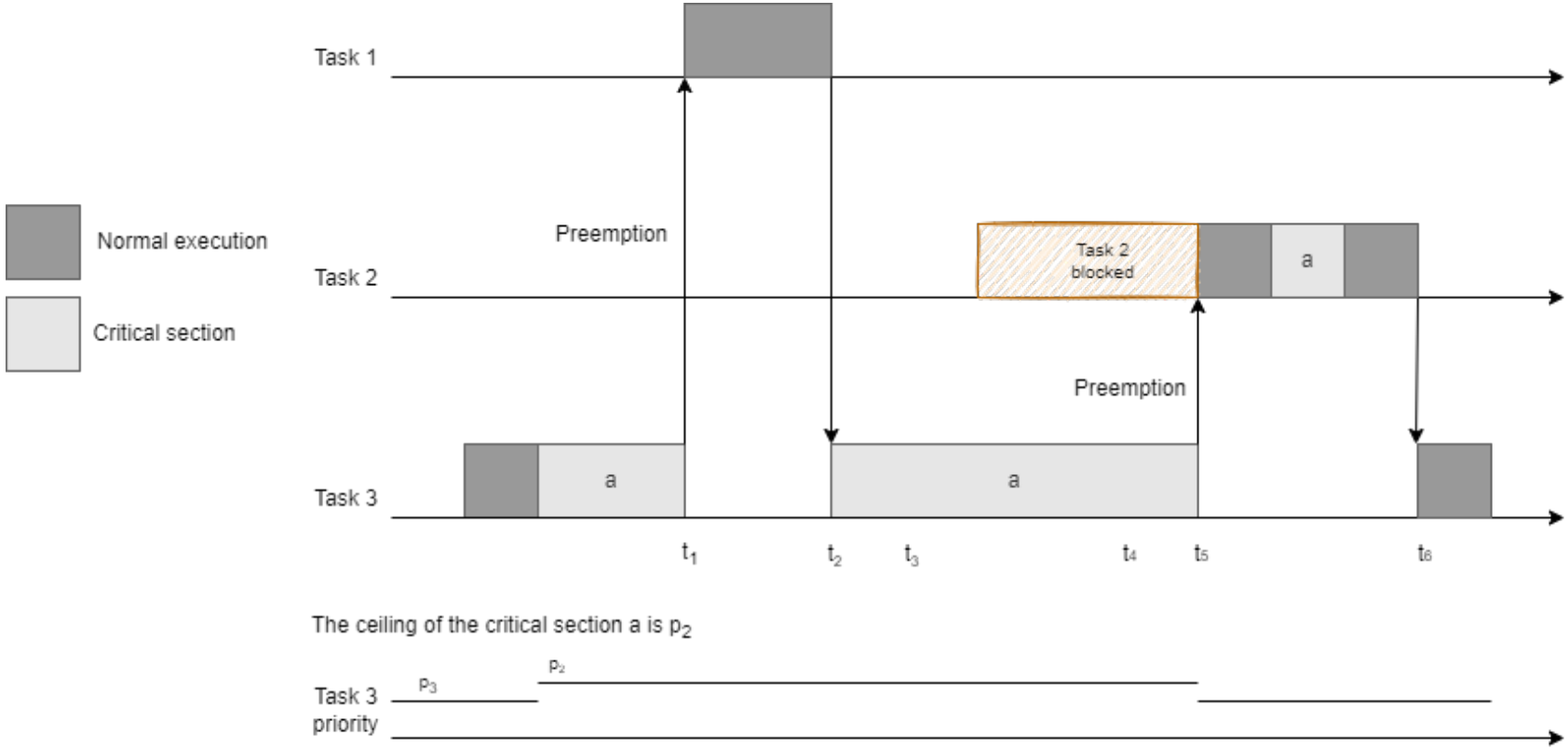
Highest Locker's Priority Protocol

- Idea: define the ceiling C(S) of a critical section S to be the highest priority of all tasks that use S during execution.
  - Note that C(S) must be calculated statically (off-line).
- Whenever a task succeeds in holding a critical section S, its priority is changed dynamically to the maximum of its current priority and C(S)
- When it finishes with S, it sets its priority back to what it was before.

	priority	use
Task 1	H	S3
Task 2	M	S1, S
Task 3	L	S1, S2
Task 4	Lower	S2, S

C(S1)=M
C(S2)=L
C(S3)=H
C(S)=M

Highest Locker Priority Protocol (HLP)

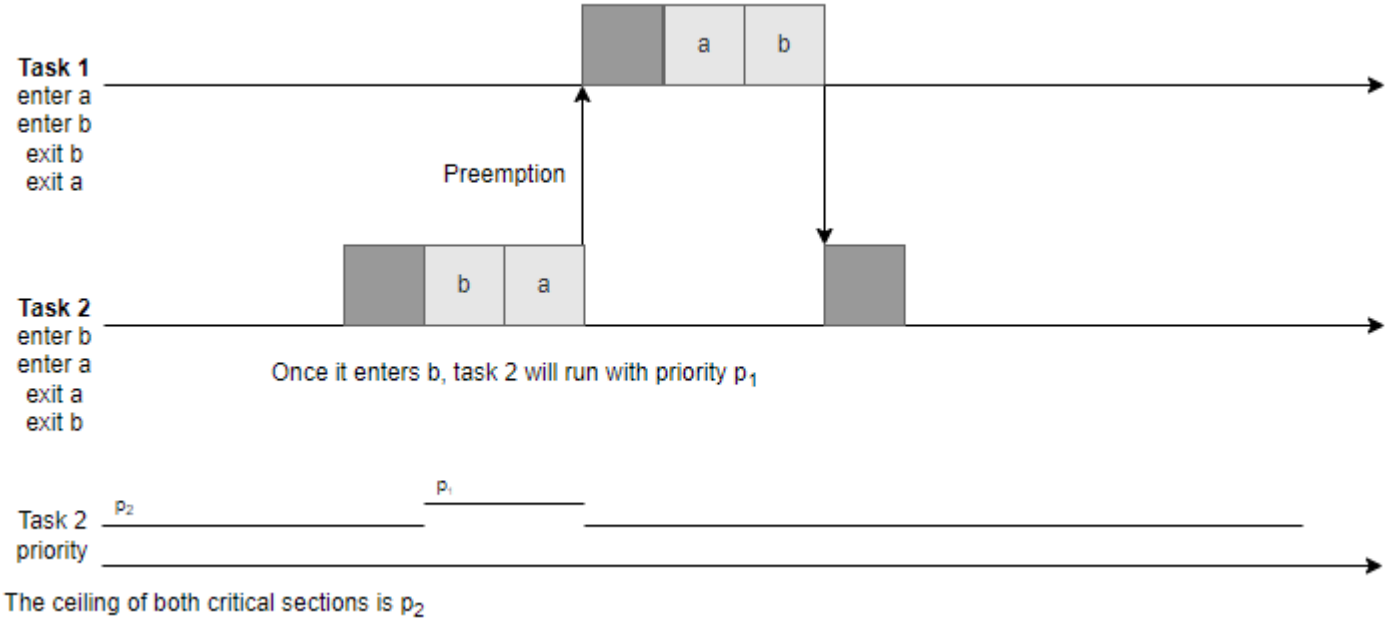


Highest Locker Priority Protocol (HLP)

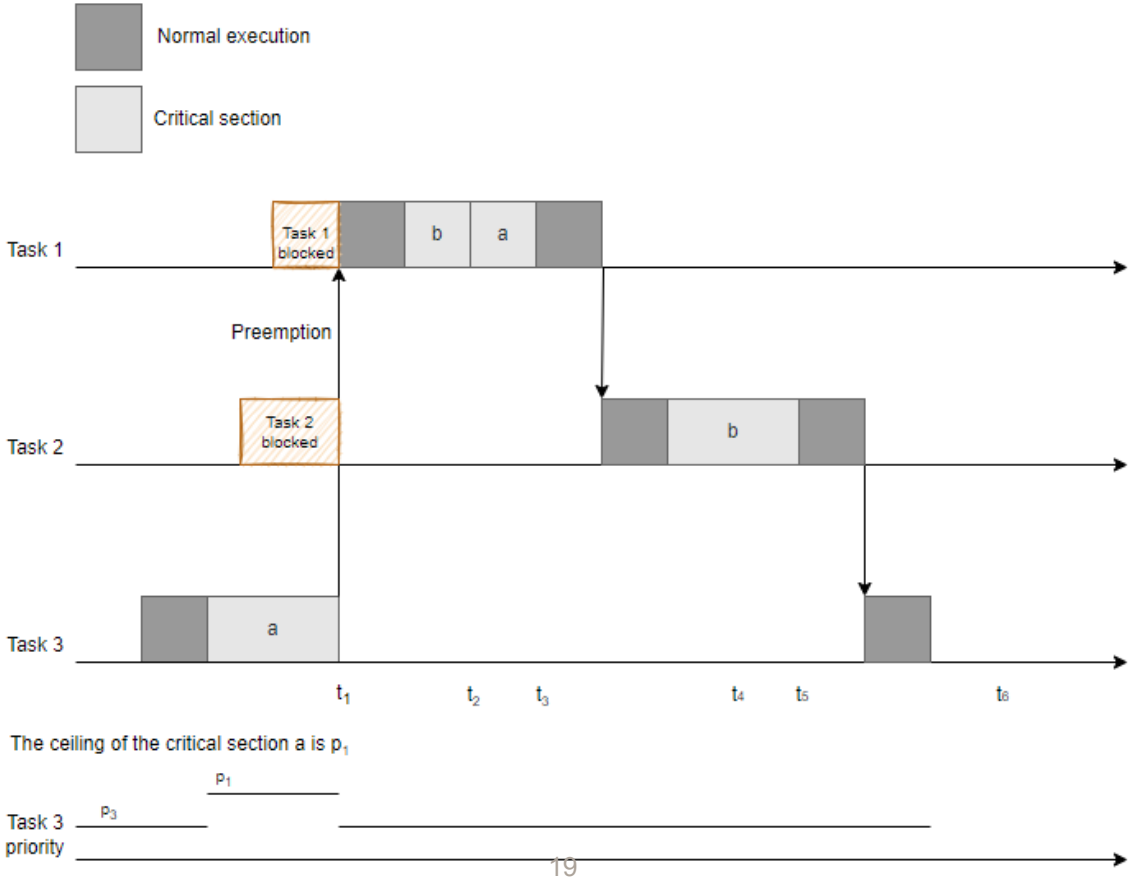
- Advantages:
  - It introduces a bound to the blocking time.
  - It limits the number of blocking of any task to one.
  - It is deadlock free
- But:
  - Implementation is not transparent
  - It still introduces unnecessary blocking
    - Since blocking happens at task arrival – it may block a task that does not access the critical section or access it much later.

No deadlock with HLP

- Once task 2 gets CS b, it runs with priority p_1
- Task 1 will be blocked and cannot get CS a before task 2



No chained blocking with HLP

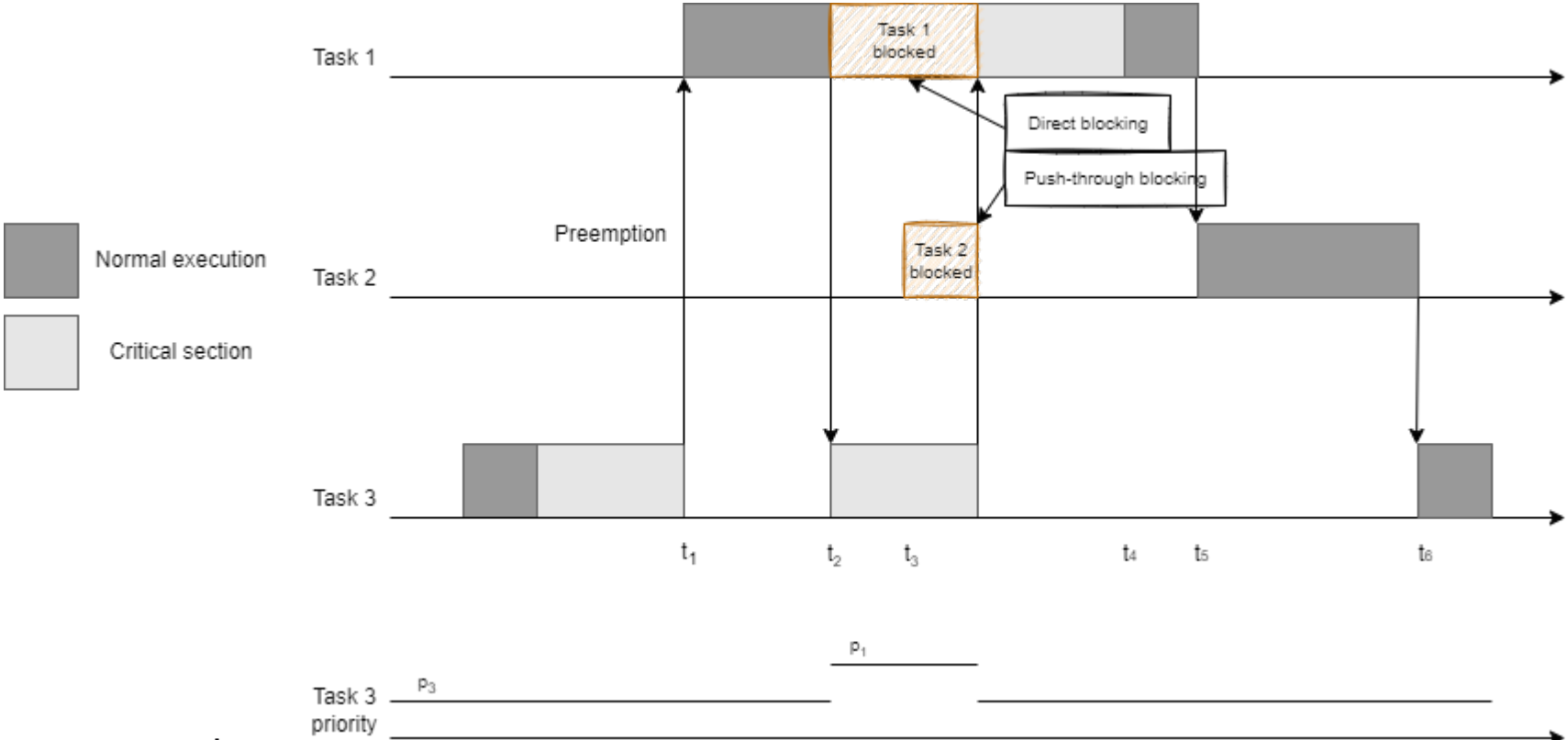


Priority Inheritance Protocol (PIP)

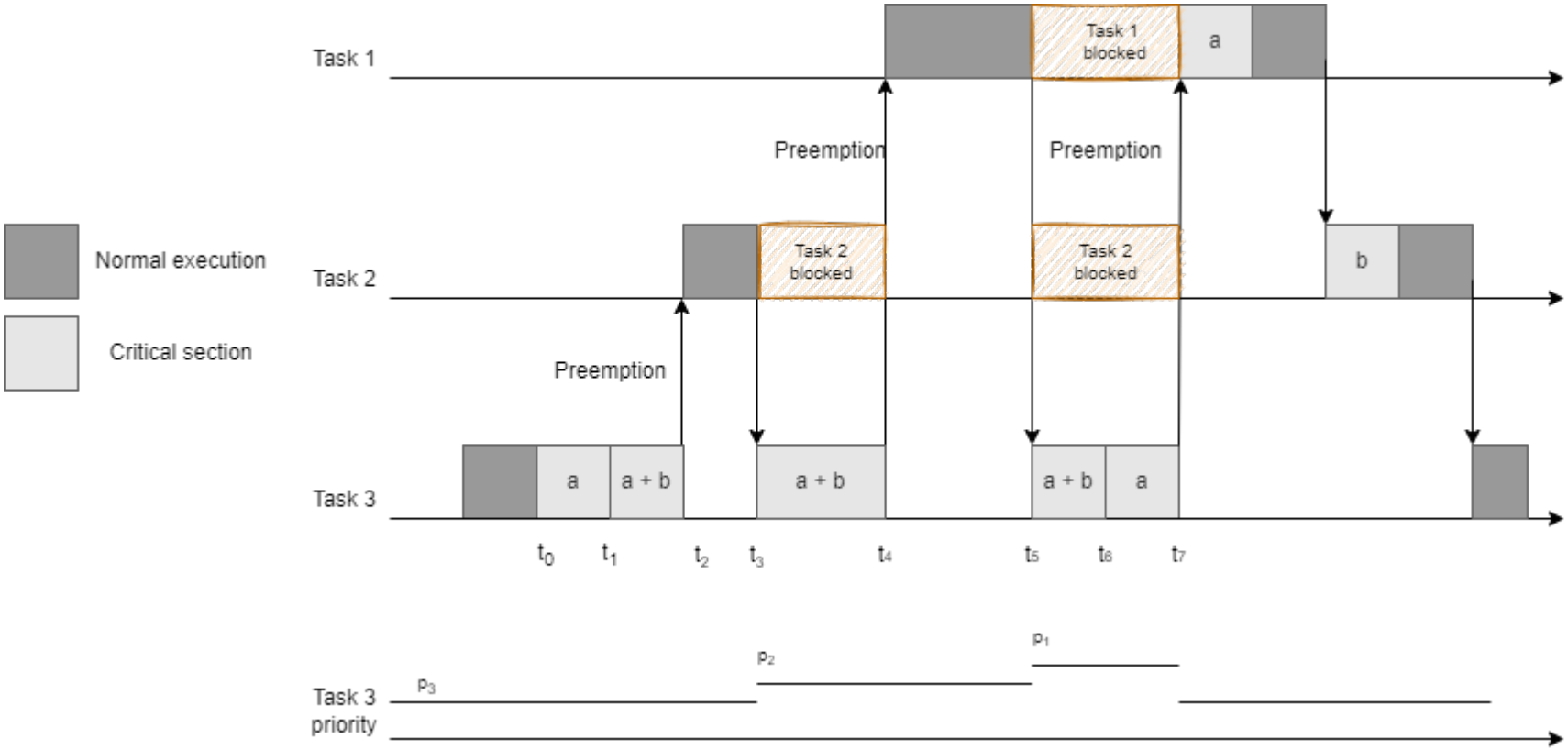
Principle: when a task blocks a task with higher priority, it temporarily inherits the highest priority of the blocked tasks.

- A task inherits the highest priority of the tasks it blocks
- The priority of a task leaving a critical section is updated as:
  - The highest-priority task blocked by this CS is unblocked
  - If no other task is blocked by the CS, its priority is set to its nominal value,
  - If other tasks are blocked by the CS, its priority is set to the highest-priority of the tasks blocked
- This prevents medium-priority tasks from preempting lower priority tasks and thus prolonging the blocking duration experienced by the higher-priority tasks.

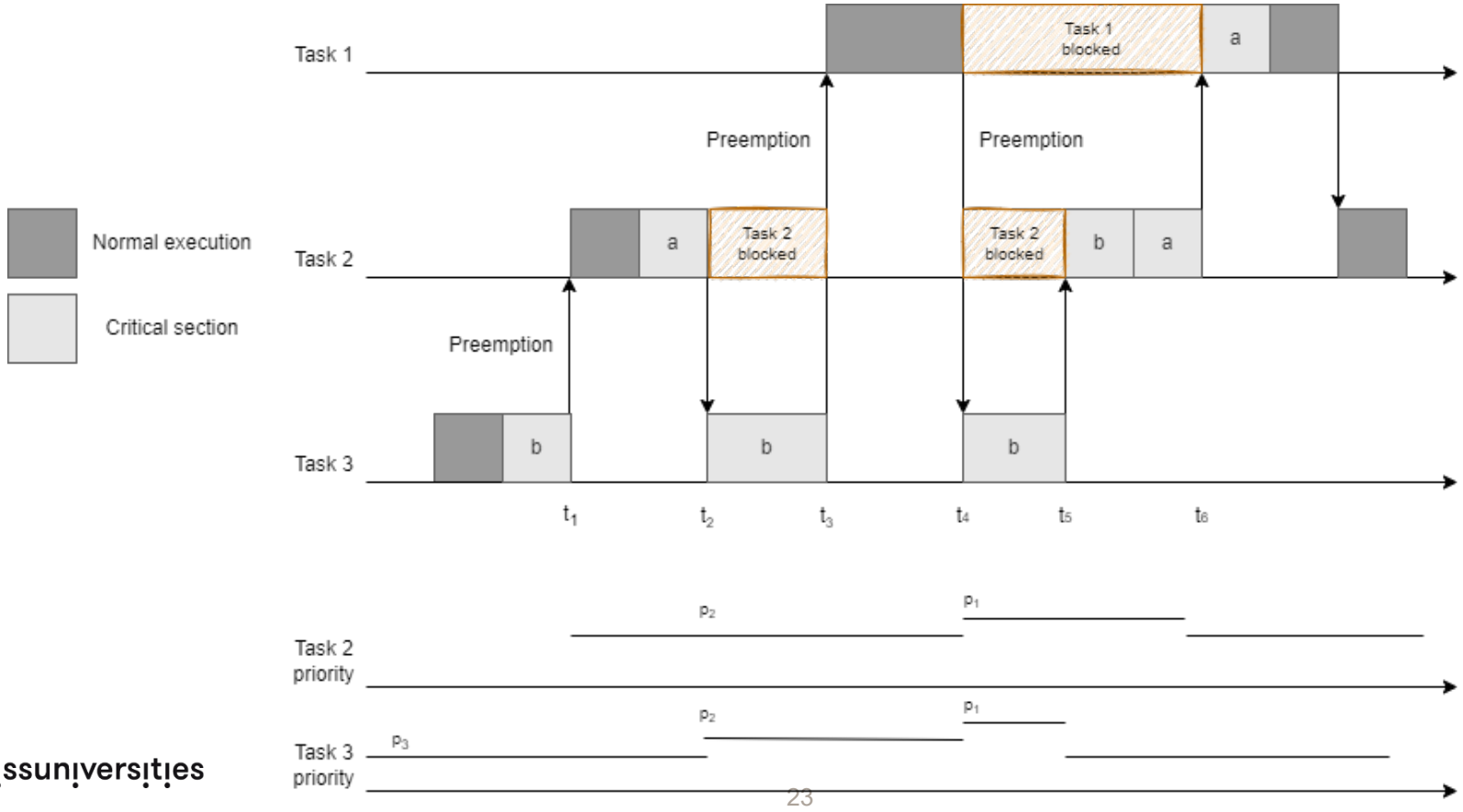
Priority Inheritance Protocol (PIP)



Priority Inheritance Protocol (PIP)



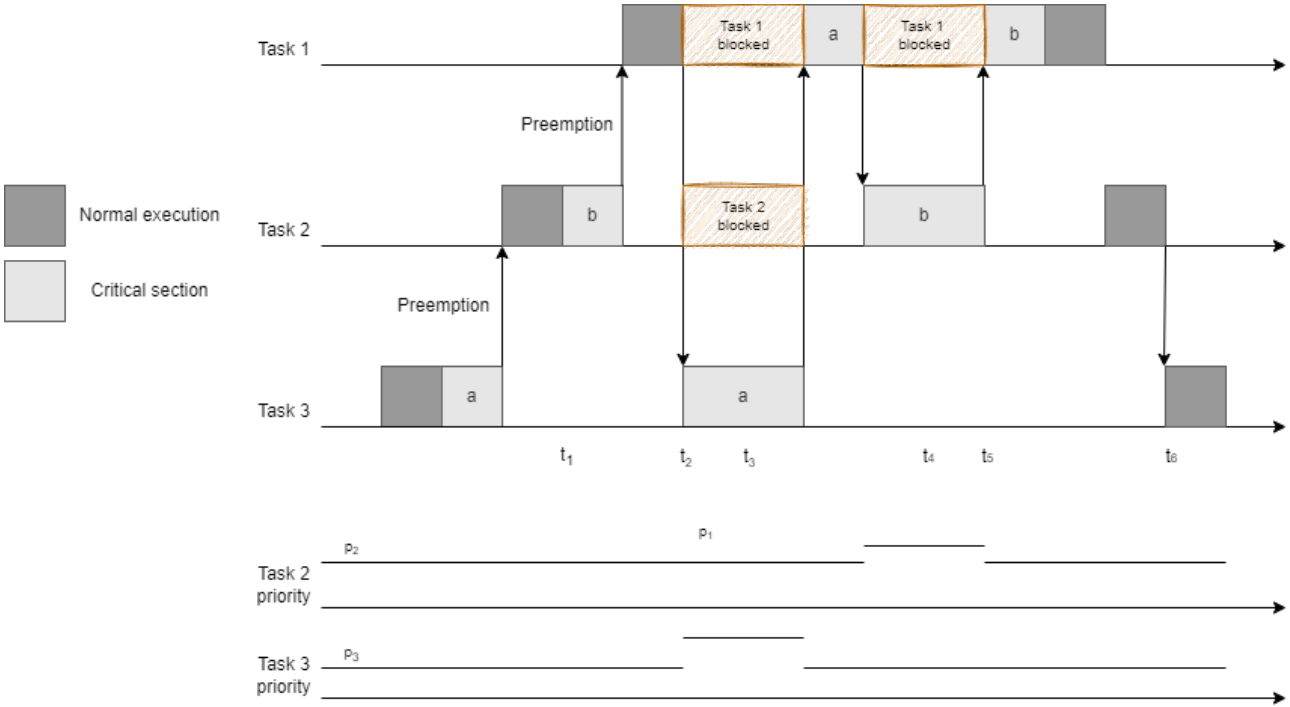
Priority Inheritance Protocol (PIP)



PIP Properties

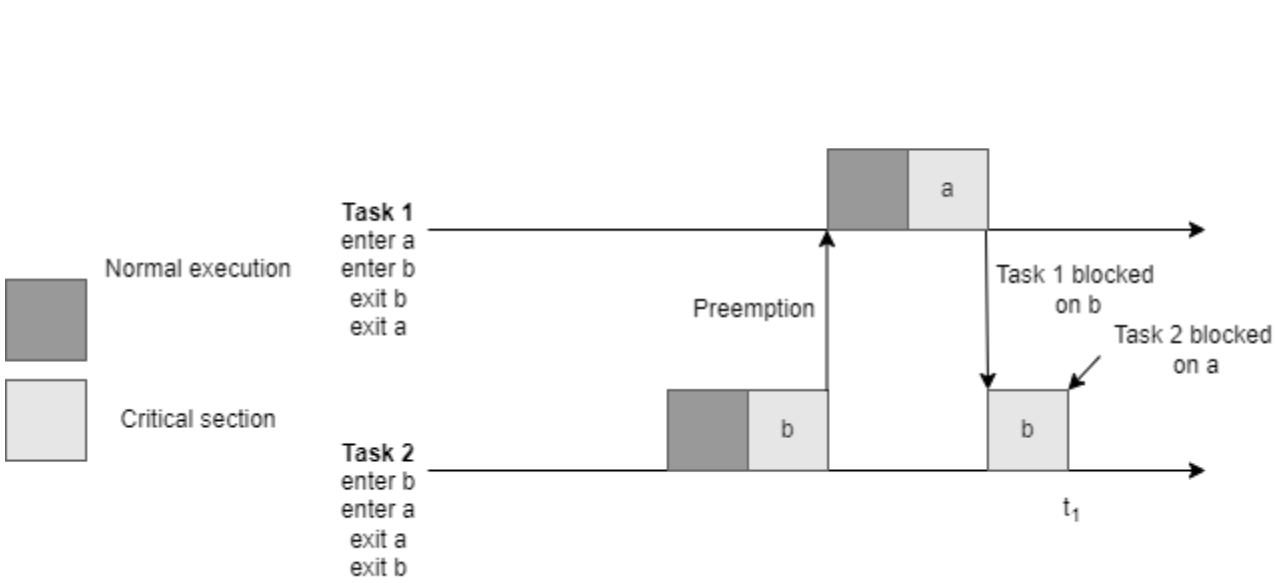
- Push-through blocking
  - A medium-priority task is blocked by a low-priority task that has inherited a higher priority
  - It can happen only if a critical section is accessed both by a task with lower priority and by a task with higher priority
  - Necessary to avoid unbounded priority inversion
- Transitive inheritance
  - Can happen only in the presence of nested CSs
- Blocking time upper bound
  - A task can be blocked for at most the duration of α critical sections, where α is $\min(l, s)$
    - l is the number of lower-priority tasks that can block the task
    - s is the number of distinct semaphores that can block the task

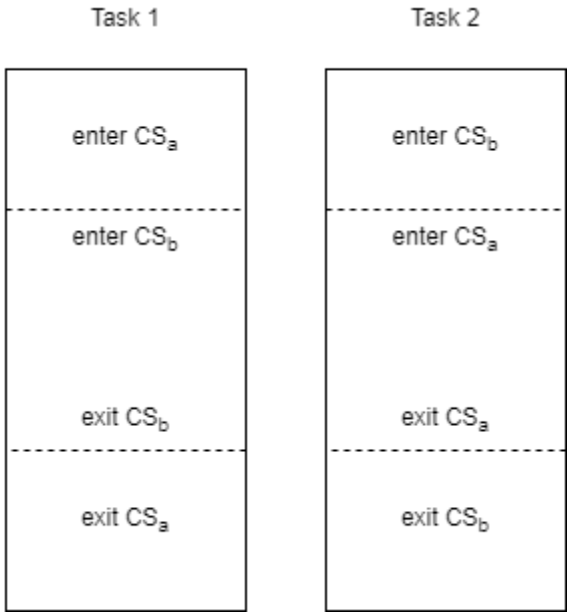
PIP Issue: Chained Blocking



In the worst case, a high priority task may be blocked once by each of the lower priority task

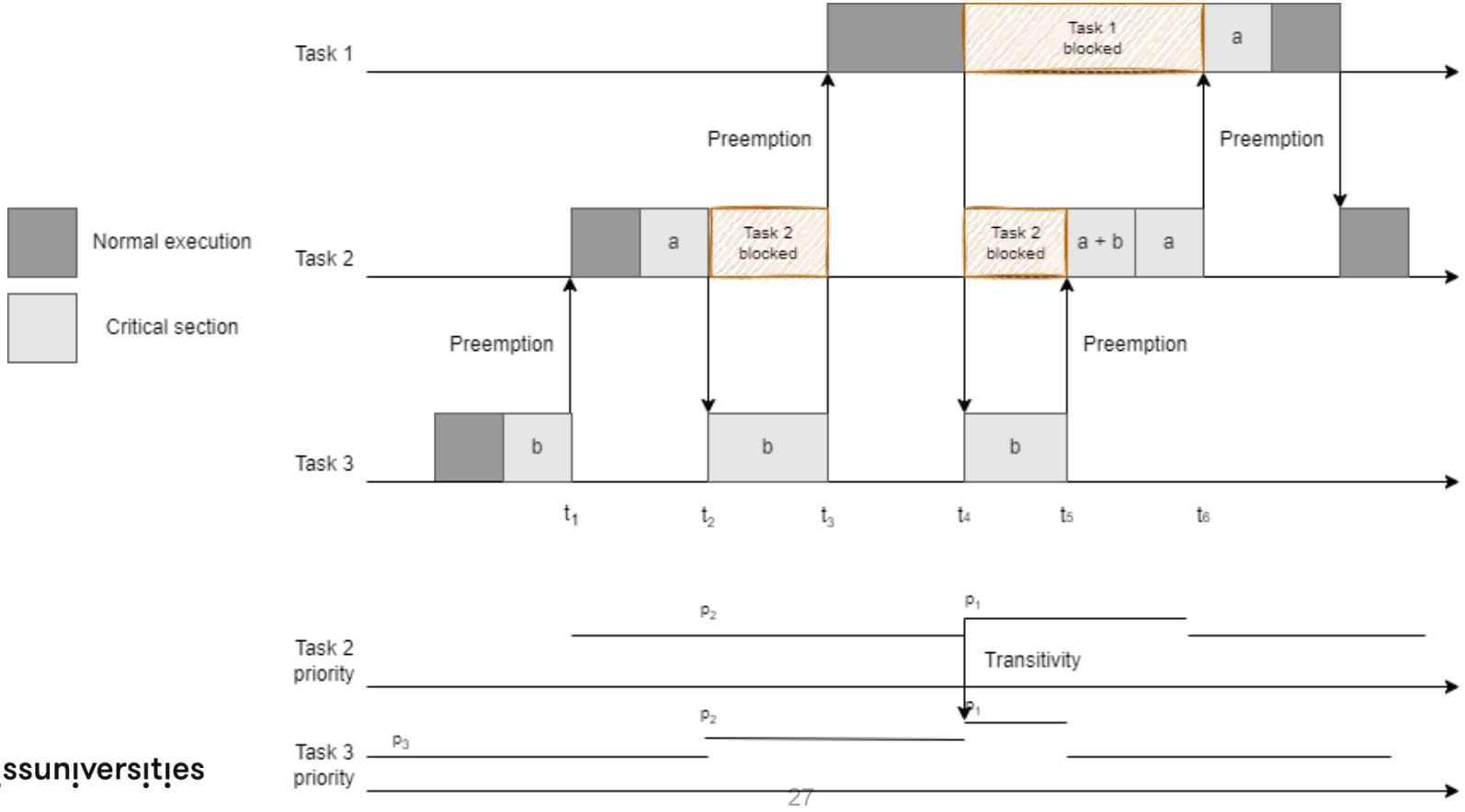
PIP Issue: Deadlock



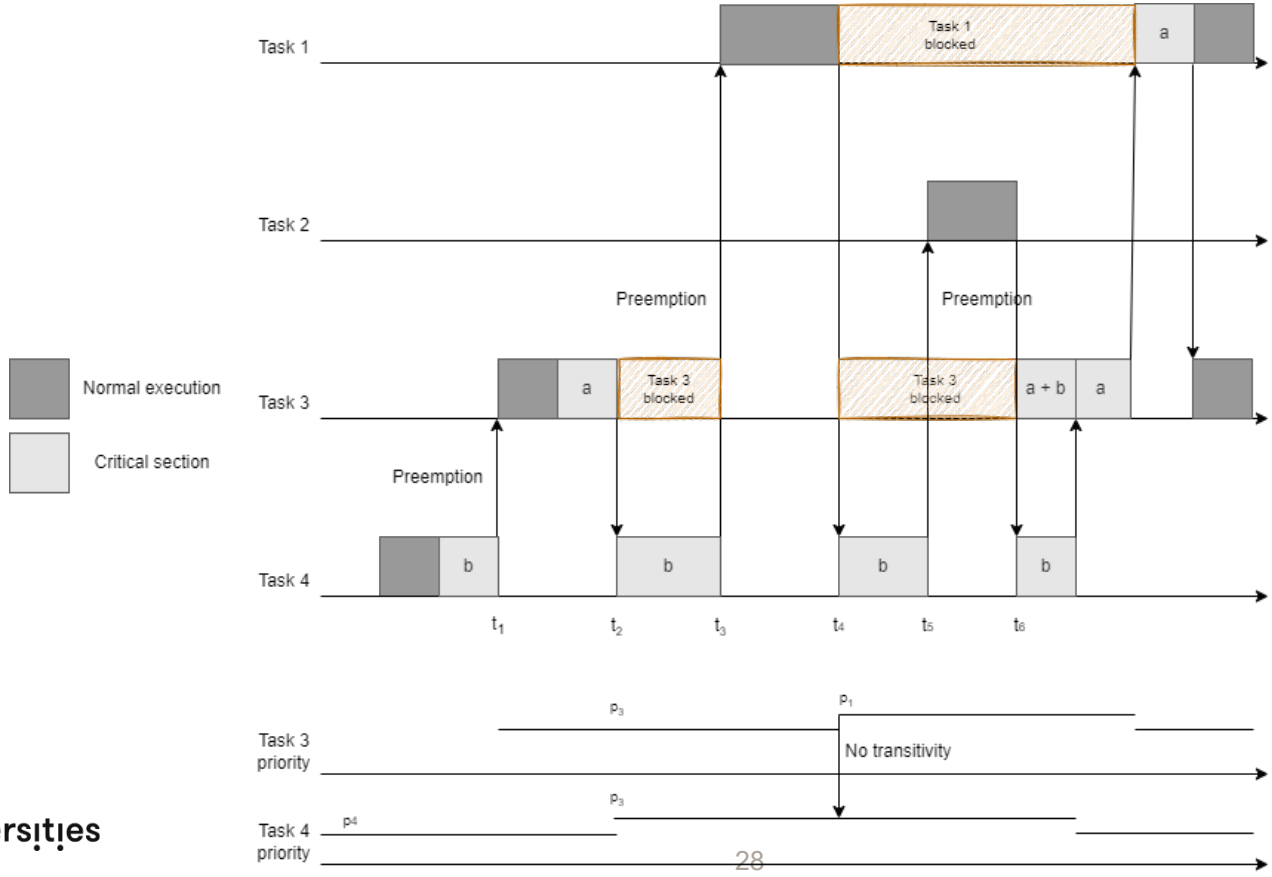


PIP does not prevent deadlock

Transitivity for PIP



Transitivity is needed



PIP Implementation

- Implementation is transparent
  - It does require changes in the kernel but not in the application or in the kernel API
- Required changes in the data structure
  - Each thread must store its nominal and active priority
  - Inheritance: Each mutex keeps track of the thread holding the mutex
  - Transitivity: Each thread keeps track of the mutex in which it is blocked

PIP Implementation

- Mutex lock
  - If Mutex is free:
    - The mutex becomes locked
    - The mutex keeps track of the thread locking the mutex.
  - If Mutex is locked:
    - The task calling the lock keeps track of the blocking mutex and it blocks.
    - The priority of the thread locking the mutex is changed to the priority of the task calling the lock.
    - If the task owning the mutex is blocked on another thread, the transitivity rule is applied.
    - The ready task with the highest priority is executed.

PIP Implementation

- Mutex unlock
  - If no thread is blocked on the Mutex, then the mutex is simply unlocked.
  - If one or more threads are blocked on the Mutex:
    - The highest-priority thread blocked on the Mutex is awakened.
    - It is stored as owner of the Mutex.
    - The priority of the thread unlocking the Mutex is updated.

Resource Access Protocols Summary

Algorithm	Chained blocking	Unnecessary Blocking	Blocking instant	Deadlock prevention	Transparency	Implementation
NPP	no	yes	on arrival	yes	yes	easy
PIP	yes	limited	on access	no	yes	medium
HLP	no	yes	on arrival	yes	no	medium

- Each protocol has advantages and disadvantages
- There are other protocols
- Most RTOS implement PIP

Schedulability Analysis with Dependencies

- Recall the schedulability test and hyperbolic test for n tasks under RM

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq \mathrm{n}(2^{\frac{1}{n}} - 1)$$

$$\prod_{i=1}^{n} (U_i + 1) \leq 2$$

- With dependent tasks, we may compute an upper bound B_i for the blocking time of each task
  - Depending on the Resource Access Protocol
- Schedulability tests may be extended to include this upper bound
  - Inflate the computation time C_i by the blocking time B_i
  - Blocking times are computed under worst-case scenarios for each task
  - Worst-case scenarios cannot happen simultaneously for each task

Schedulability Analysis with Dependencies

Rationale for extending schedulability analysis with blocking times

- In the worst case, each task can be blocked for the sum of durations of critical section of lower-priority tasks – only once.
- A task cannot be blocked by another task with higher priority

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

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

Mutex vs Semaphore

- Mutexes are often called binary semaphores
- But mutexes are NOT like semaphores
- Semaphores implement a mechanism for producer-consumer scenarios
  - The task that acquires the semaphore is often not the same as the one that releases the semaphore
  - Releasing a semaphore from an ISR context is allowed
- Mutexes implement a mechanism for exclusive access to a critical section
  - The task that locks and unlocks the mutex is the same
  - Accessing a mutex from an ISR context is not allowed
  - Mutexes are recursive/reentrant while semaphores are not
  - Mutexes implement a Resource Access Protocol (like PIP) while semaphores do not
- Implementation of semaphores and mutexes is different
  - Do NOT use a binary semaphore as a mutex !

References

- The Little Book of Semaphores, A. B. Downey (https://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf)
- Hard Real-Time Computing Systems, chapter 7, G.C. Buttazzo (https://embreal.isc.heia-fr.ch/documentation/assets/literature/Hard%20Real-Time%20Computing%20Systems.pdf)