Resource Sharing

CPEN 432 Real-Time System Design

Arpan Gujarati
University of British Columbia

Blocking on an Exclusive Resource is Unavoidable

Exclusive resource R_k accessed by waiting and signalling a binary semaphore S_k

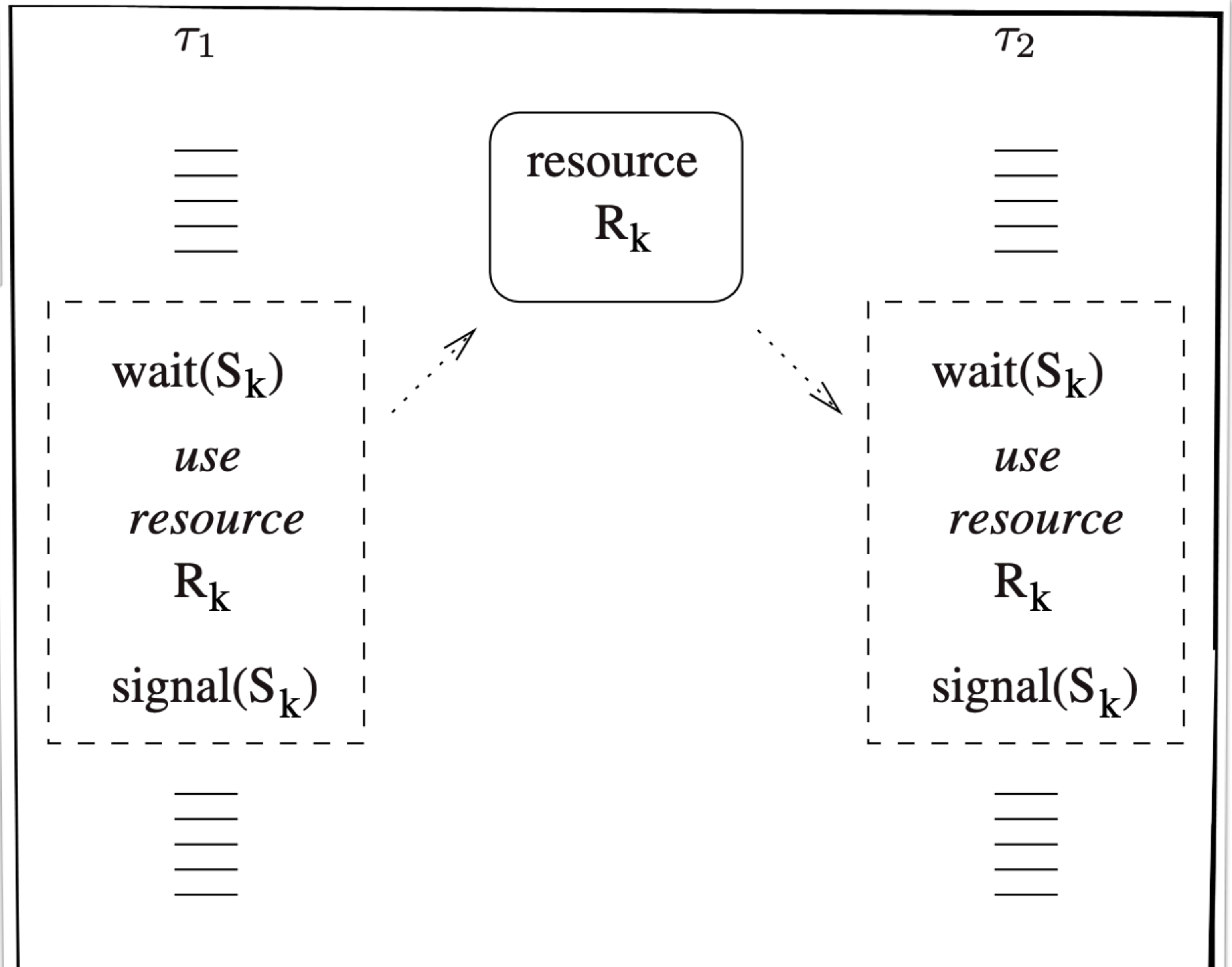

Figure 7.2 Structure of two tasks that share an exclusive resource.

Typically, the exclusive resource R_k should not be shared while a critical section using R_k is in progress

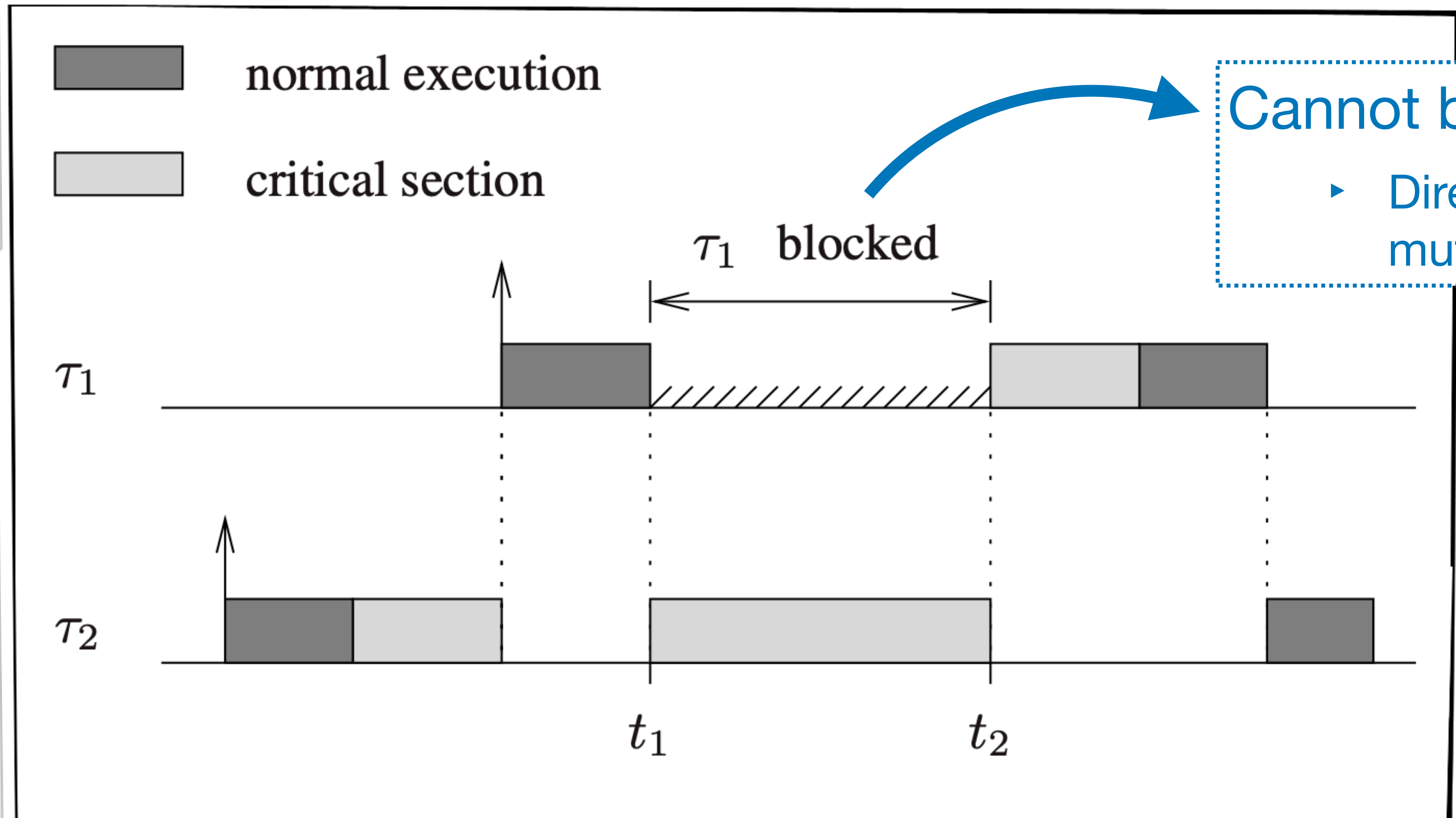

Figure 7.3 Example of blocking on an exclusive resource.

Cannot be avoided!

- Direct consequence of the mutual exclusion property

Also, easy to bound using the critical section duration

- Like the worst-case completion time C_2, we could also characterize the worst-case critical section duration of τ_2 while it uses R_k

Unbounded Blocking Due to Priority Inversion

The duration of priority inversion is unbounded

- Any intermediate priority task can preempt τ_3 and indirectly block τ_1
- If we add the completion time $C_{intermediate}$ of each intermediate-priority task $\tau_{intermediate}$ as a blocking factor in τ_1's timing analysis, the resulting system will have a very low utilization. Why?

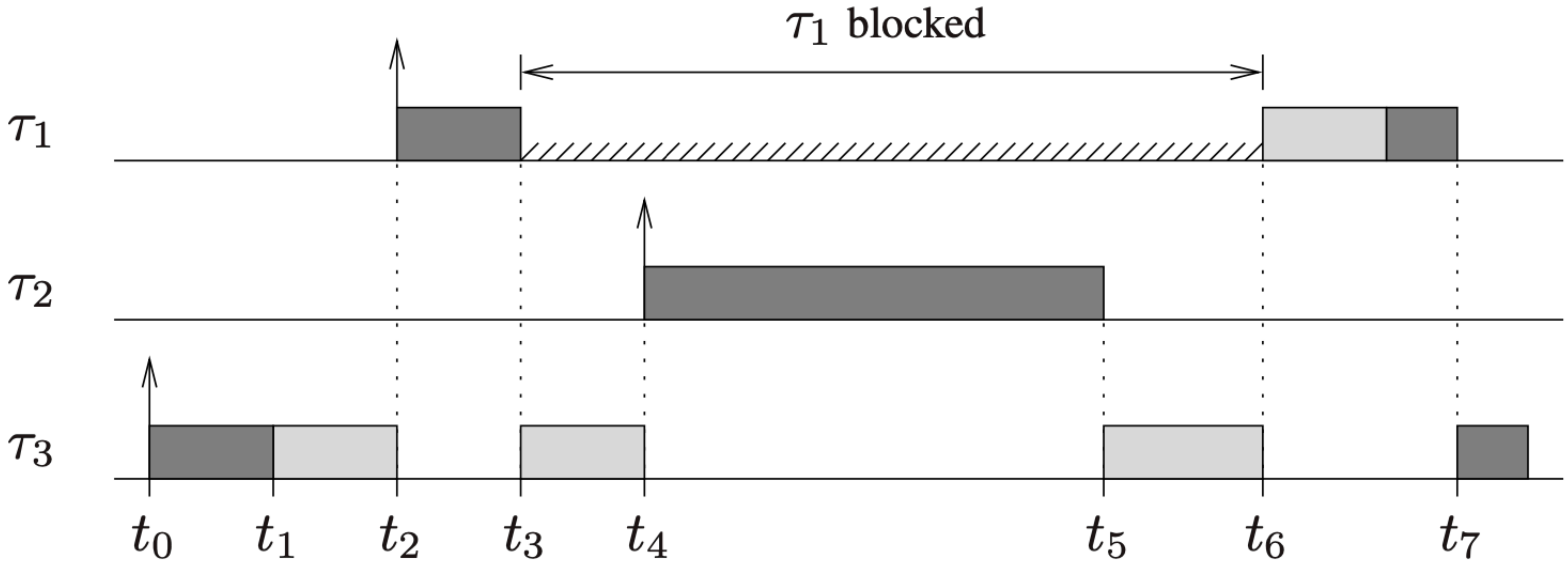

Figure 7.4 An example of priority inversion.

How to Prevent Unbounded Priority Inversions?

- Key idea …

Terminology

- Task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ consists of n periodic tasks
- Each task is characterized by a period T_i and worst-case completion time C_i
- The tasks cooperate through m shared resources $R_1, R_2, \ldots, R_m$
- Each resource R_k is guarded by a distinct **binary semaphore** S_k
  - All critical sections using R_k start and end with operations $wait(S_k)$ and $signal(S_k)$
- Each task is assigned a fixed **base priority** P_i (e.g., using RM)
  - Assumption: priorities are unique and $P_1 > P_2 > \ldots > P_n$
- Each task also has an **effective priority** p_i $(\geq P_i)$
  - It is initially set to P_i and can be **dynamically updated**
- B_i denotes the maximum blocking time task τ_i can experience
  - B_i goes into the fixed-priority response-time analysis (recall from previous lectures)
- $z_{i,k}$ denotes any arbitrary critical section of τ_i guarded by semaphore S_k
  - $Z_{i,k}$ denotes the longest among all these critical sections
  - $\delta_{i,k}$ denotes the length of this longest critical section $Z_{i,k}$

Non-Preemptive Protocol (NPP)

Observation

Blocking caused by the **preemption** of a running, **resource-holding** job

- E.g., τ_3 preempted by τ_2 at time t_4 while holding the shared resource

Key idea: **Disable preemption** before acquiring a shared resource; **reenable upon exit** of critical section

When a task τ_i acquires a resource R_k, its dynamic priority is raised to the level of the highest priority, i.e., $p_i(R_k) = \max_{\forall h}\{P_h\}$

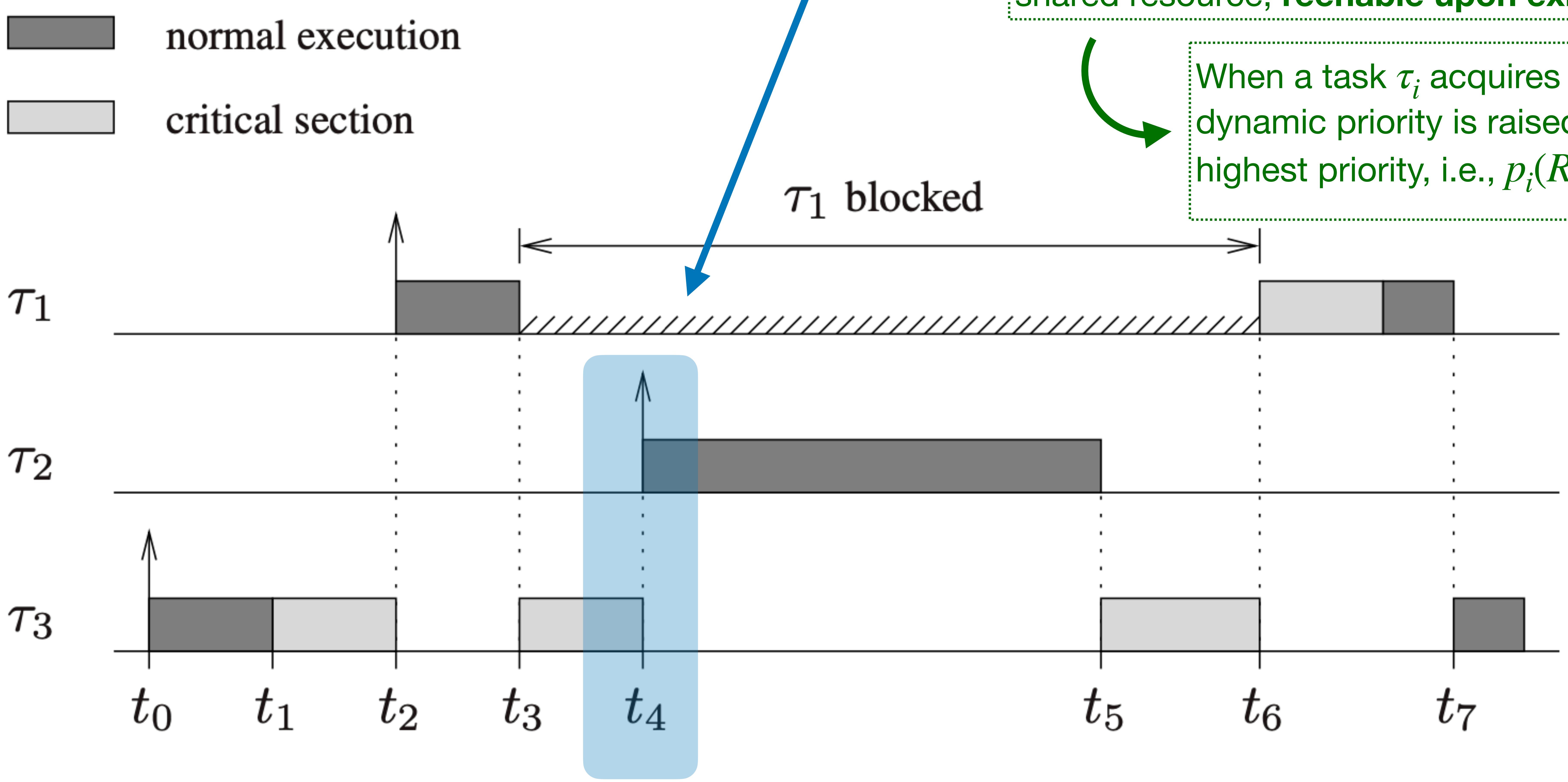

Figure 7.4 An example of priority inversion.

Example

Priority inversion **bounded by critical section length**

- How can we **formally** define τ_i's blocking time bound B_i?

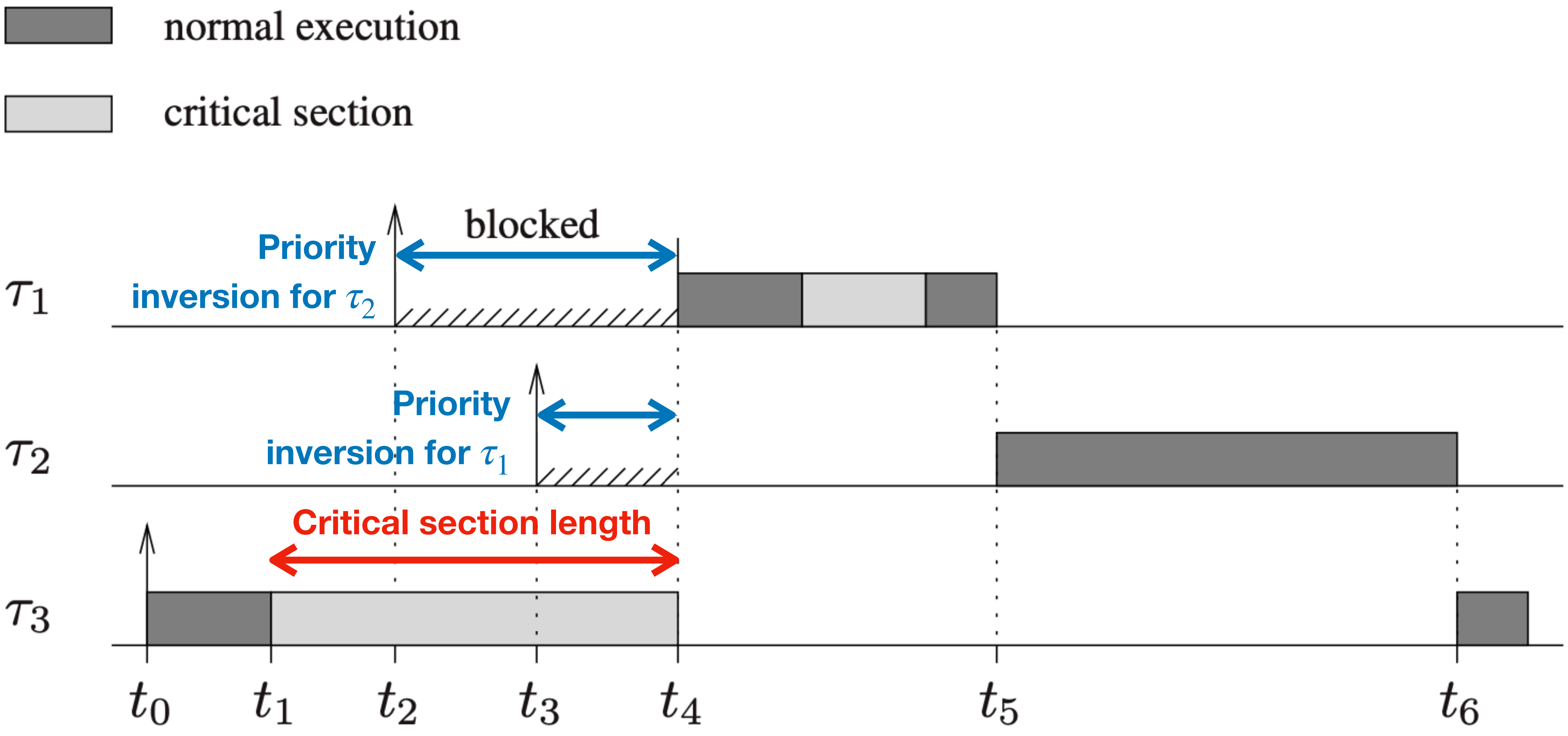

Figure 7.5 Example of NPP preventing priority inversion.

NPP Benefits & Limitations

- Most **simple** way to prevent unbounded priority inversions
- Can be realized by **disabling/reenabling interrupts**
  - Raising task priorities is a useful abstraction but needn't be implemented in this case
- Limitations
  - Turning off interrupts risks **large interrupt latency**
  - All tasks effected
    - Even **independent tasks blocked** due to priority inversion

What if high-frequency tasks cannot tolerate blocking even due to a single, **long** non-preemptive section?

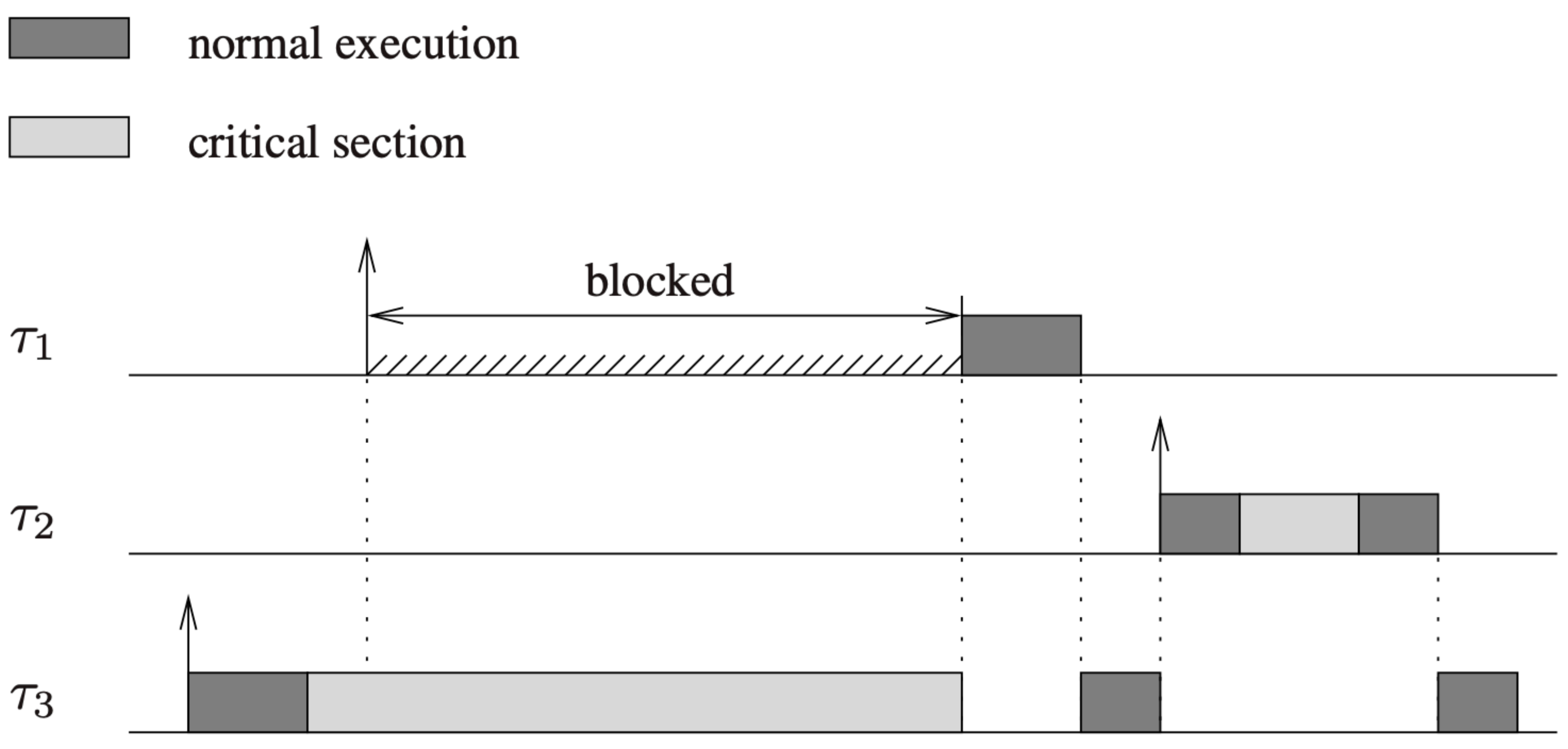

Figure 7.6 Example in which NPP causes unnecessary blocking on τ_1.

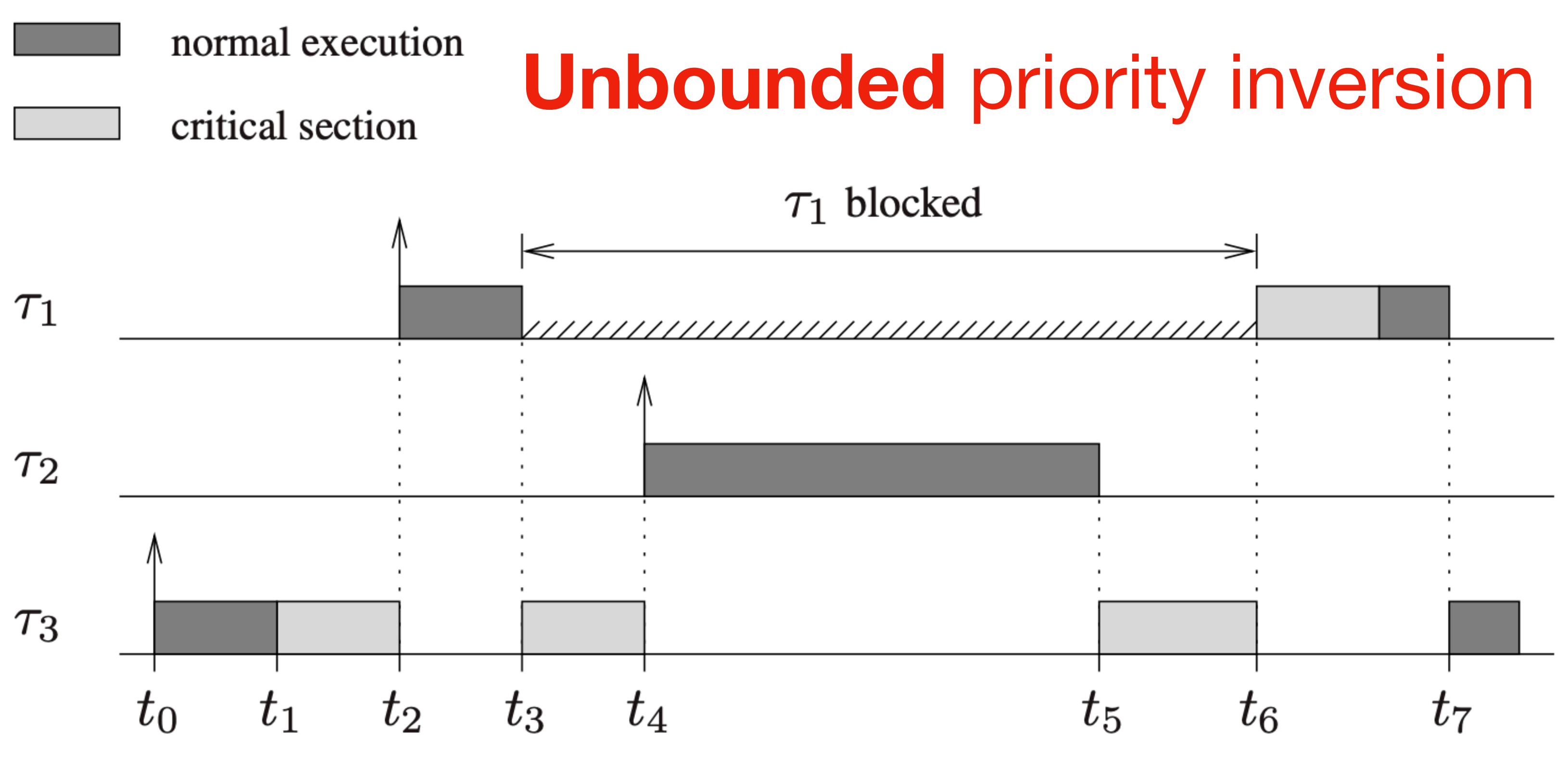

Figure 7.4 An example of priority inversion.

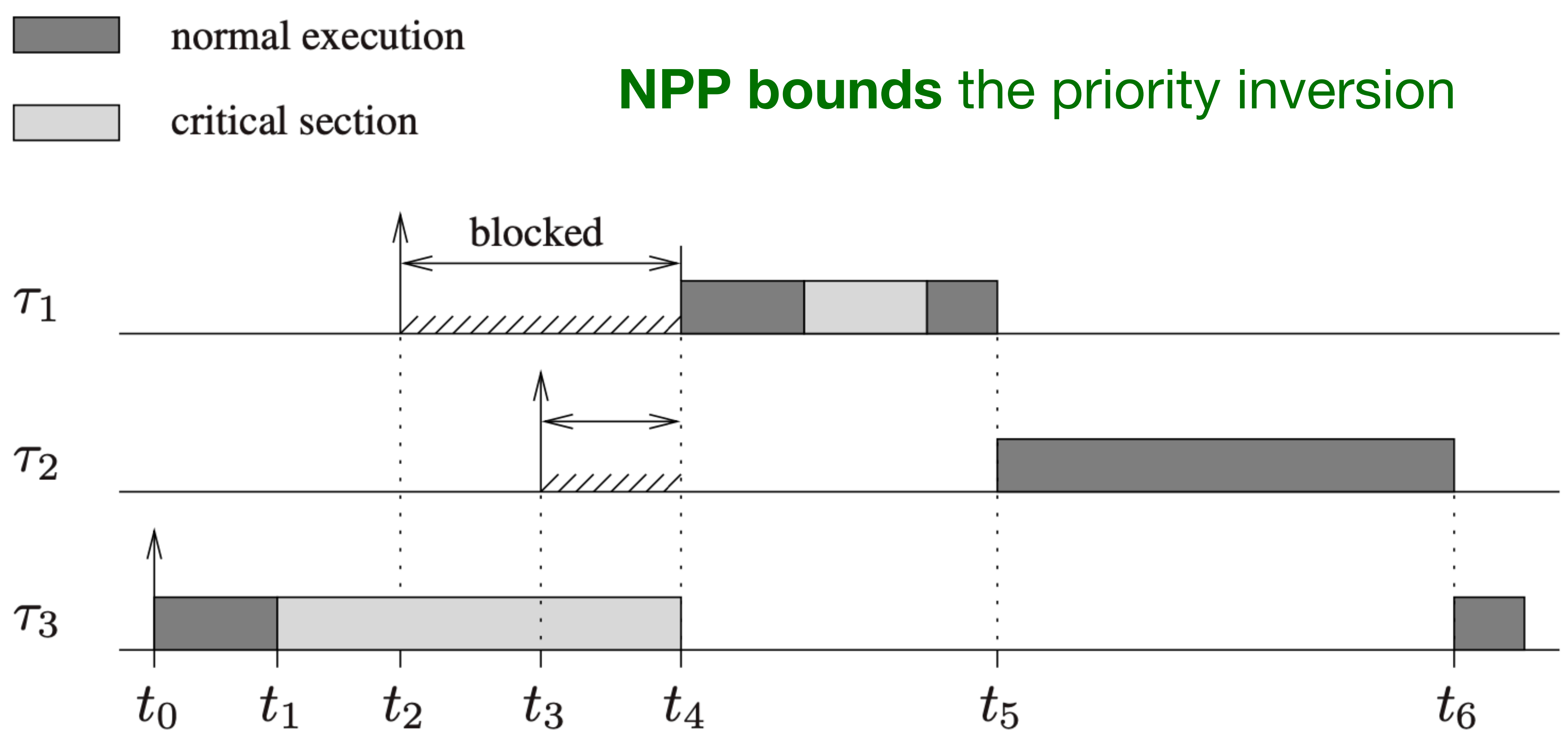

Figure 7.5 Example of NPP preventing priority inversion.

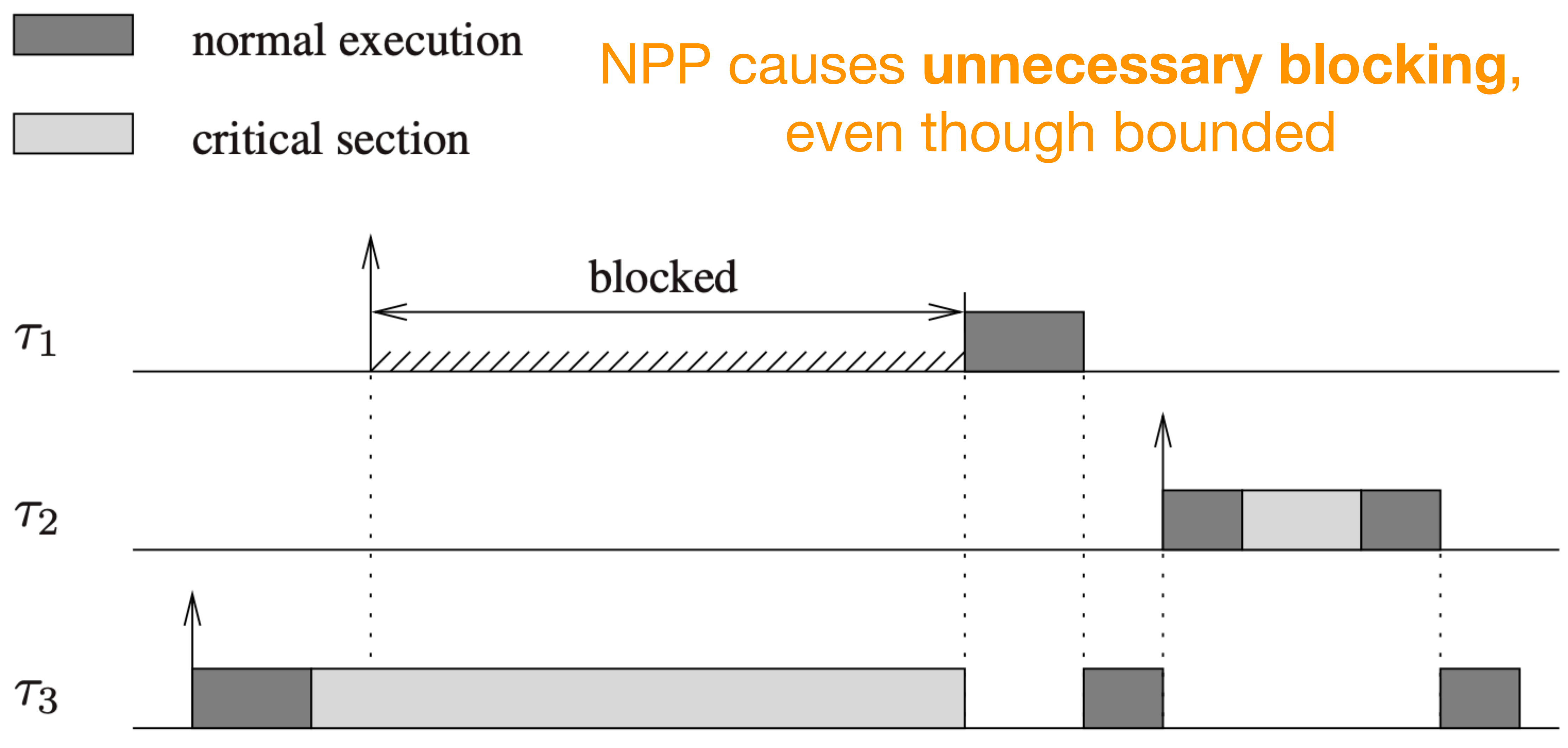

Figure 7.6 Example in which NPP causes unnecessary blocking on τ_1.

What’s next?

The Priority Inheritance Protocol (PIP)

Protocol Definition

- Unlike NPP, resource holding jobs remain **fully preemptive**
- Tasks are scheduled based on their effective priorities
  - For scheduling purposes, τ_i's priority is considered to be p_i and not P_i
- Suppose task τ_i tries to enter a critical section by acquiring resource R_k
  - Case 1: R_k is already held by a lower-priority task $\tau_j \implies \tau_i$ is **blocked** by τ_j
  - Case 2: R_k is already held by a higher-priority task $\tau_j \implies \tau_i$ is **interfered** by τ_k
  - Case 3: R_k is not held by any task $\implies \tau_i$ **enters** the critical section
- For Case 1,τ_j **inherits** τ_i's effective priority
  - τ_j's dynamic priority is updated as $p_j = p_i$
- In general, τ_j inherits the **highest priority of among all tasks that it blocks**
  - At any point of time, $p_j(R_k) = \max \{P_j, \max_{\forall h} \{p_h \,|\, \tau_h \text{ is blocked on } R_k\}\}$

Example 1

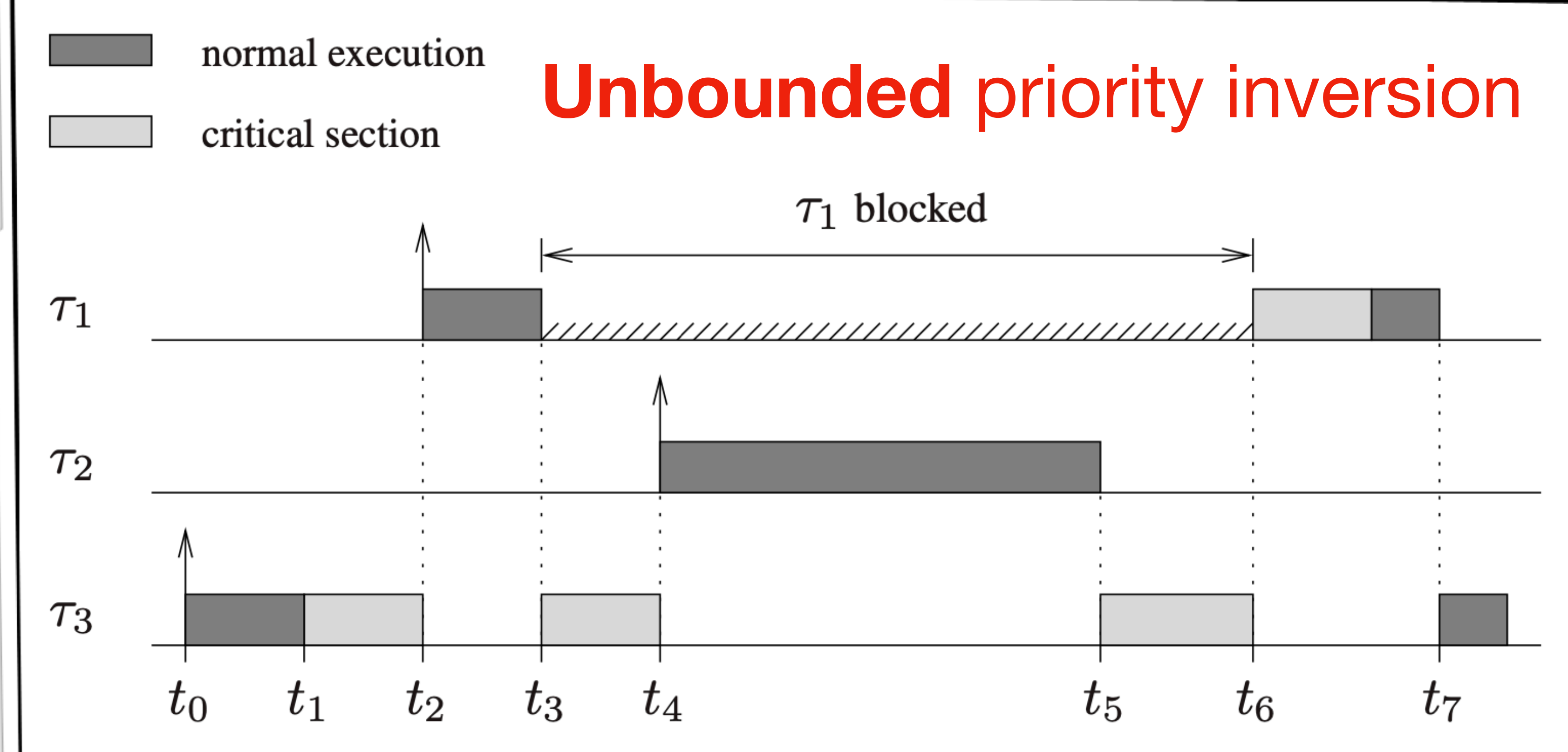

Figure 7.4 An example of priority inversion.

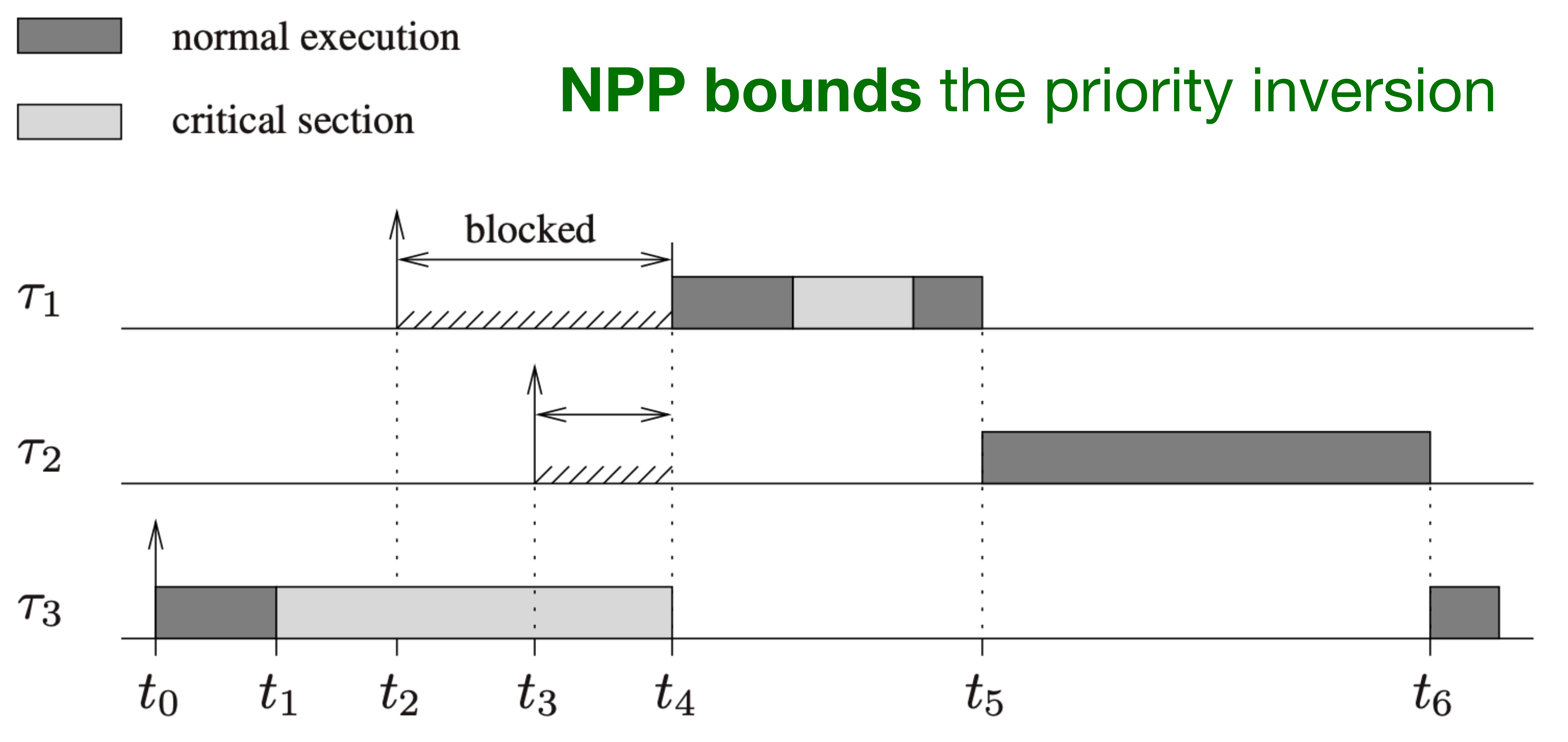

Figure 7.5 Example of NPP preventing priority inversion.

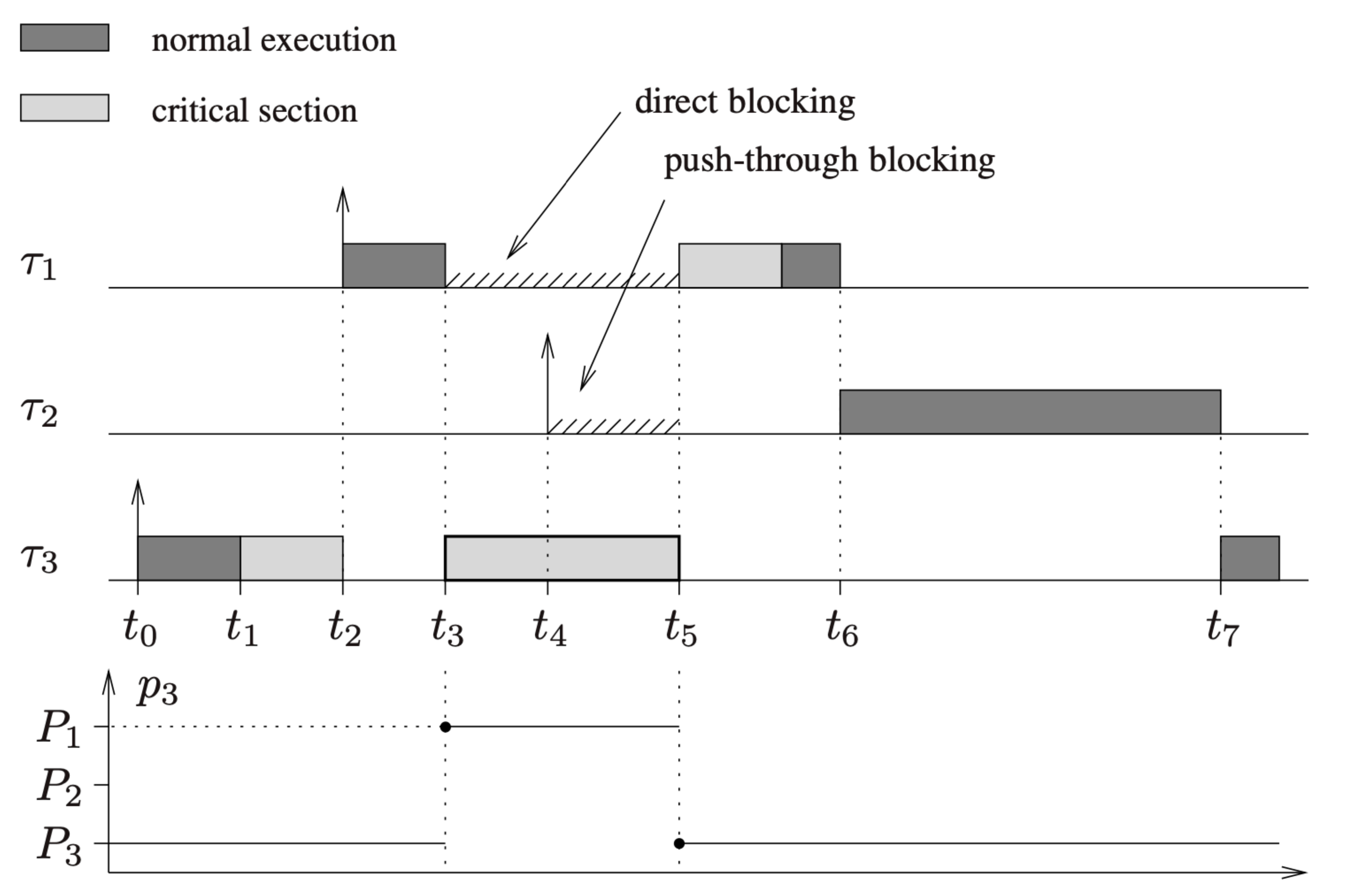

Figure 7.8 Example of Priority Inheritance Protocol.

Example 2

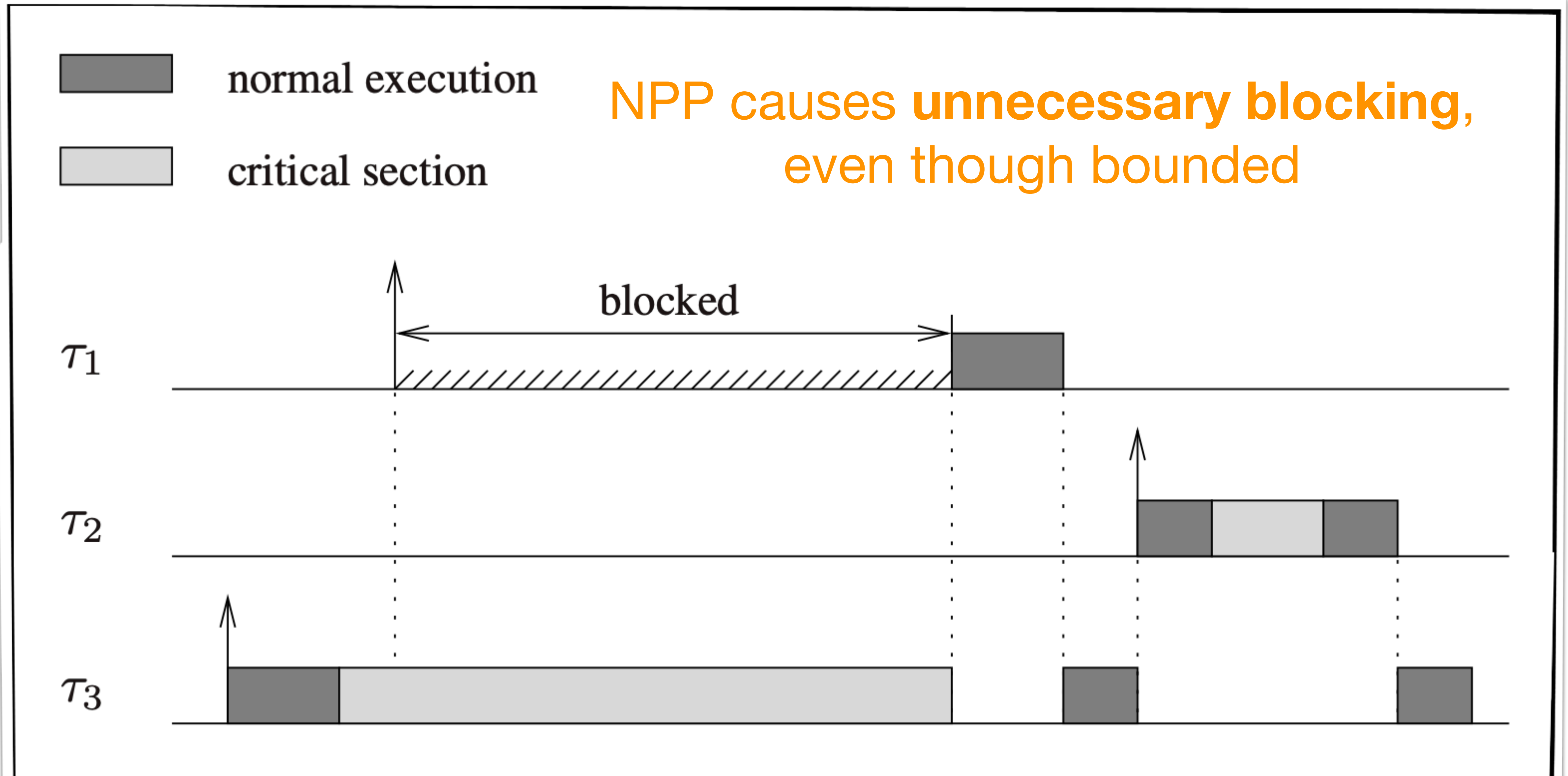

Figure 7.6 Example in which NPP causes unnecessary blocking on τ_1.

Example 3: Nested Blocking

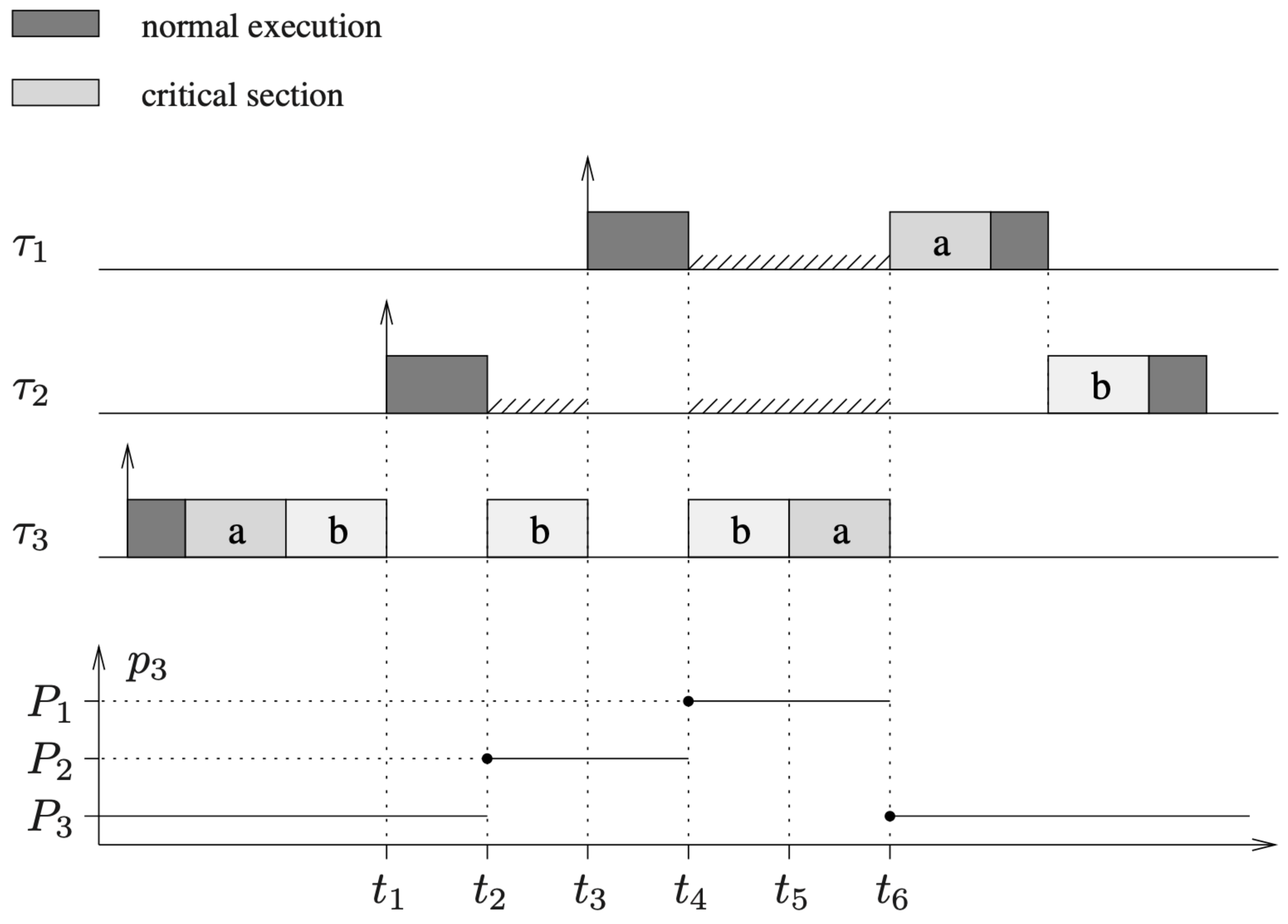

Figure 7.9 Priority inheritance with nested critical sections.

Example 4: Transitive Blocking

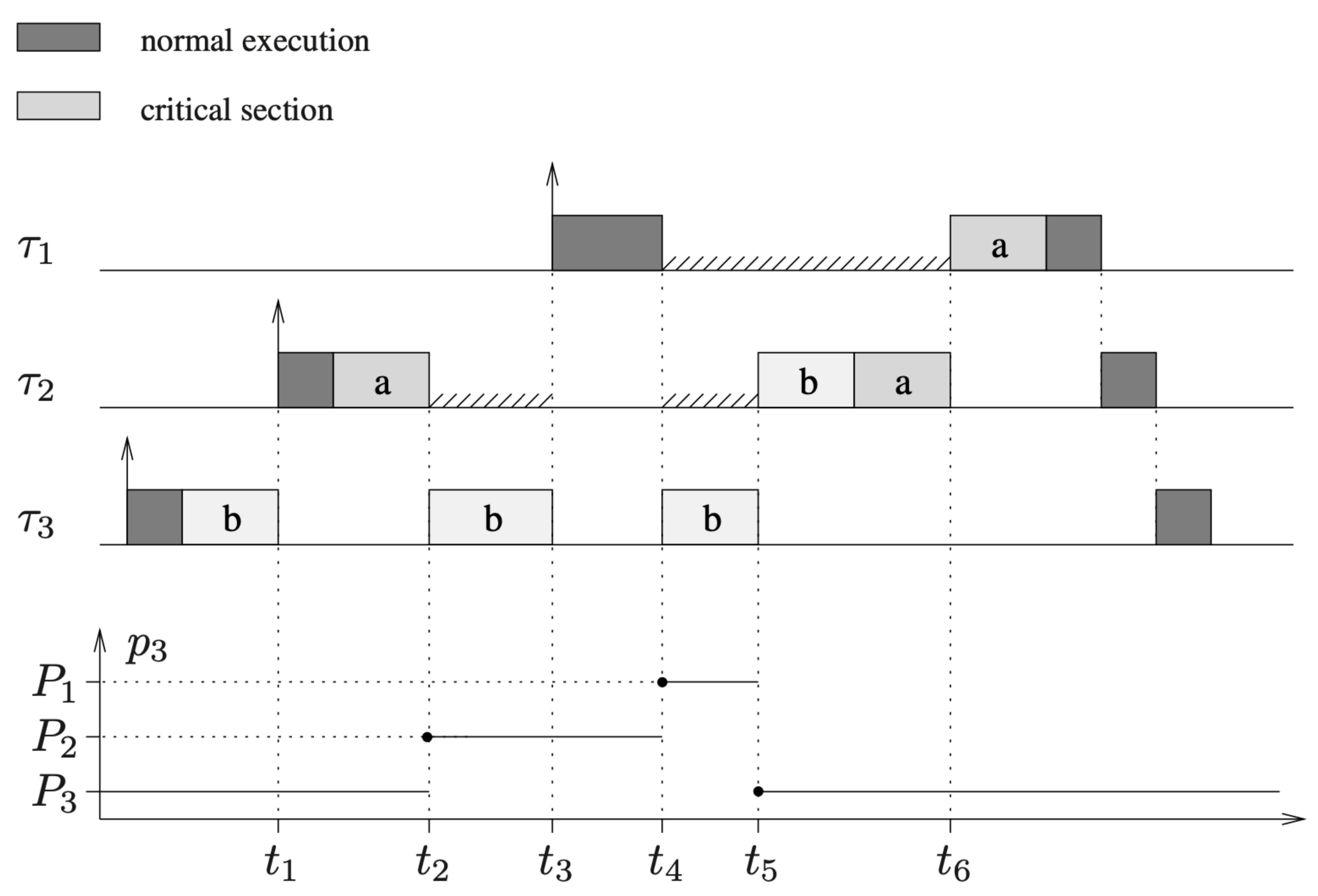

Figure 7.10 Example of transitive priority inheritance.

PIP Benefits & Limitations

- **No latency penalty** for high-priority independent tasks
- Widely used in practice: POSIX's PTHREAD_PRIO_INHERIT
- Limitations
    - Chained blocking
    - Deadlock

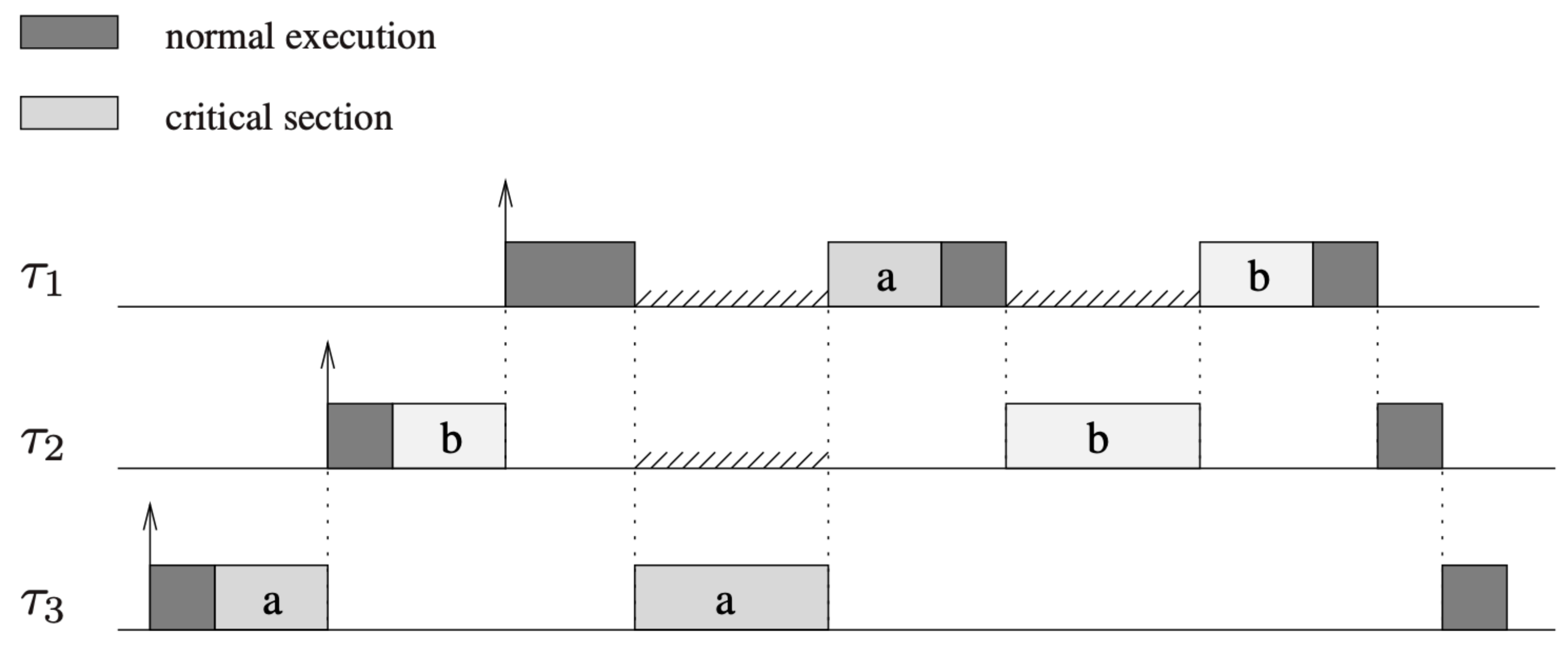

Figure 7.12 Example of chained blocking.

The Priority Ceiling Protocol (PCP)

PCP vs PIP

- The PIP is a **reactive** locking protocol
  - It only kicks in when resource contention already exists
- **Key PCP insight**
  - Better to **prevent** problematic scenarios rather **than resolve** them
- The PCP is an **anticipatory** locking protocol
  - Exploits the knowledge of resource needs at **design time** to avoids excessive blocking at runtime

Key Concepts

- **Priority ceilings**
  - Each semaphore S_k is **statically** assigned a priority ceiling $C_{static}(S_k)$
    - $C_{static}(S_k) =$ priority of the highest-priority task that **ever** accesses S_k

- **Current system ceiling**
  - At any time t, a global system ceiling $C_{global}(t)$ is dynamically computed
    - $C_{global}(t)$ = highest priority ceiling among all semaphores locked at time t OR
      (if no semaphores are locked) sentinel value P_0 that is **smaller** than all task priorities

- **Protocol**
  - Task τ_i can acquire semaphore S_k at time t only if
    - Its effective priority $p_i > C_{global}(t)$ OR $p_i = C_{global}(t)$ and τ_i "owns" the ceiling resource
    - OTHERWISE, it transmits its priority to the task τ_j that holds semaphore S_k

Example

Priority Inversion — Does It Matter?

- **What really happened on Mars Rover Pathfinder**
  - https://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html
- **What the Media Couldn’t Tell You About Mars Pathfinder**
  - https://people.cs.ksu.edu/~hatcliff/842/Docs/Course-Overview/pathfinder-robotmag.pdf

What really happened on Mars Rover Pathfinder?

- *But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data. The press reported these failures in terms such as "software glitches" and "the computer was trying to do too many things at once".*

- *This week at the IEEE Real-Time Systems Symposium I heard a fascinating keynote address by David Wilner, Chief Technical Officer of Wind River Systems. Wind River makes VxWorks, the real-time embedded systems kernel that was used in the Mars Pathfinder mission. In his talk, he explained in detail the actual software problems that caused the total system resets of the Pathfinder spacecraft, how they were diagnosed, and how they were solved. I wanted to share his story with each of you.*

- *VxWorks provides preemptive priority scheduling of threads. Tasks on the Pathfinder spacecraft were executed as threads with priorities that were assigned in the usual manner reflecting the relative urgency of these tasks.*

What really happened on Mars Rover Pathfinder?

- *Pathfinder contained an "information bus", which you can think of as a shared memory area used for passing information between different components of the spacecraft. A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes).*

- *The meteorological data gathering task ran as an infrequent, low priority thread, and used the information bus to publish its data. When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex.*

- *If an interrupt caused the information bus thread to be scheduled while this mutex was held, and if the information bus thread then attempted to acquire this same mutex in order to retrieve published data, this would cause it to block on the mutex, waiting until the meteorological thread released the mutex before it could continue. The spacecraft also contained a communications task that ran with medium priority.*

- *Most of the time this combination worked fine. However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running. After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset.*

- *This scenario is a classic case of priority inversion.*

What really happened on Mars Rover Pathfinder?

- *When created, a VxWorks mutex object accepts a boolean parameter that indicates whether priority inheritance should be performed by the mutex. The mutex in question had been initialized with the parameter off; had it been on, the low-priority meteorological thread would have inherited the priority of the high-priority data bus thread blocked on it while it held the mutex, causing it be scheduled with higher priority than the medium-priority communications task, thus preventing the priority inversion. Once diagnosed, it was clear to the JPL engineers that using priority inheritance would prevent the resets they were seeing.*

- *VxWorks contains a C language interpreter intended to allow developers to type in C expressions and functions to be executed on the fly during system debugging. The JPL engineers fortuitously decided to launch the spacecraft with this feature still enabled. By coding convention, the initialization parameter for the mutex in question (and those for two others which could have caused the same problem) were stored in global variables, whose addresses were in symbol tables also included in the launch software, and available to the C interpreter. A short C program was uploaded to the spacecraft, which when interpreted, changed the values of these variables from FALSE to TRUE. No more system resets occurred.*

What really happened on Mars Rover Pathfinder?

- *Finally, the engineer's initial analysis that "the data bus task executes very frequently and is time-critical -- we shouldn't spend the extra time in it to perform priority inheritance" was exactly wrong. It is precisely in such time critical and important situations where correctness is essential, even at some additional performance cost.*

- *David told us that the JPL engineers later confessed that one or two system resets had occurred in their months of pre-flight testing. They had never been reproducible or explainable, and so the engineers, in a very human-nature response of denial, decided that they probably weren't important, using the rationale "it was probably caused by a hardware glitch".*

- *David also said that some of the real heroes of the situation were some people from CMU who had published a paper he'd heard presented many years ago who first identified the priority inversion problem and proposed the solution. He apologized for not remembering the precise details of the paper or who wrote it. Bringing things full circle, it turns out that the three authors of this result were all in the room, and at the end of the talk were encouraged by the program chair to stand and be acknowledged. They were Lui Sha, John Lehoczky, and Raj Rajkumar. When was the last time you saw a room of people cheer a group of computer science theorists for their significant practical contribution to advancing human knowledge? :-) It was quite a moment.*