Resource Sharing

CPEN 432 Real-Time System Design

Arpan Gujarati
University of British Columbia

Unavoidable Blocking on an Exclusive Resource

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

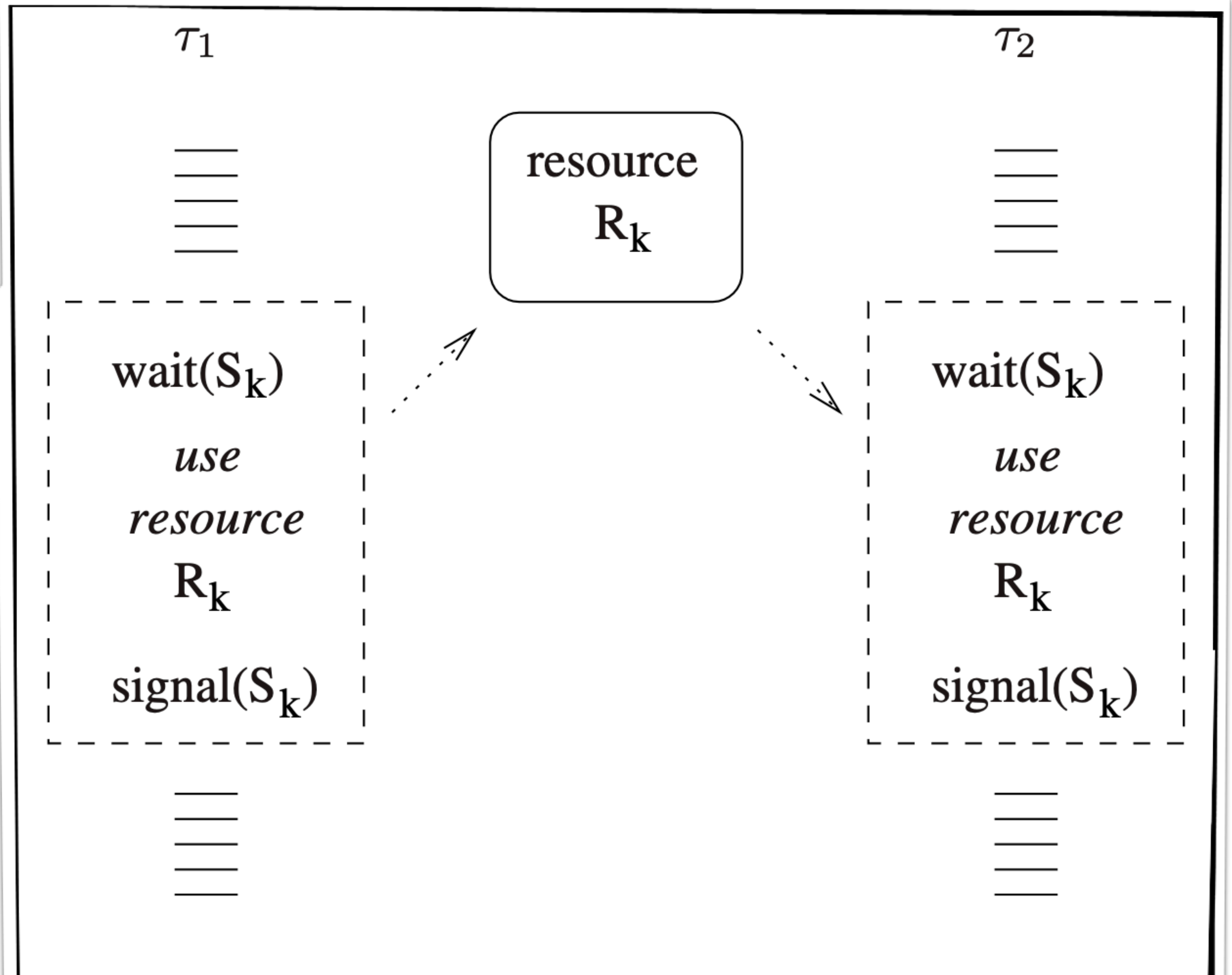

Figure 7.2 Structure of two tasks that share an exclusive resource.

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

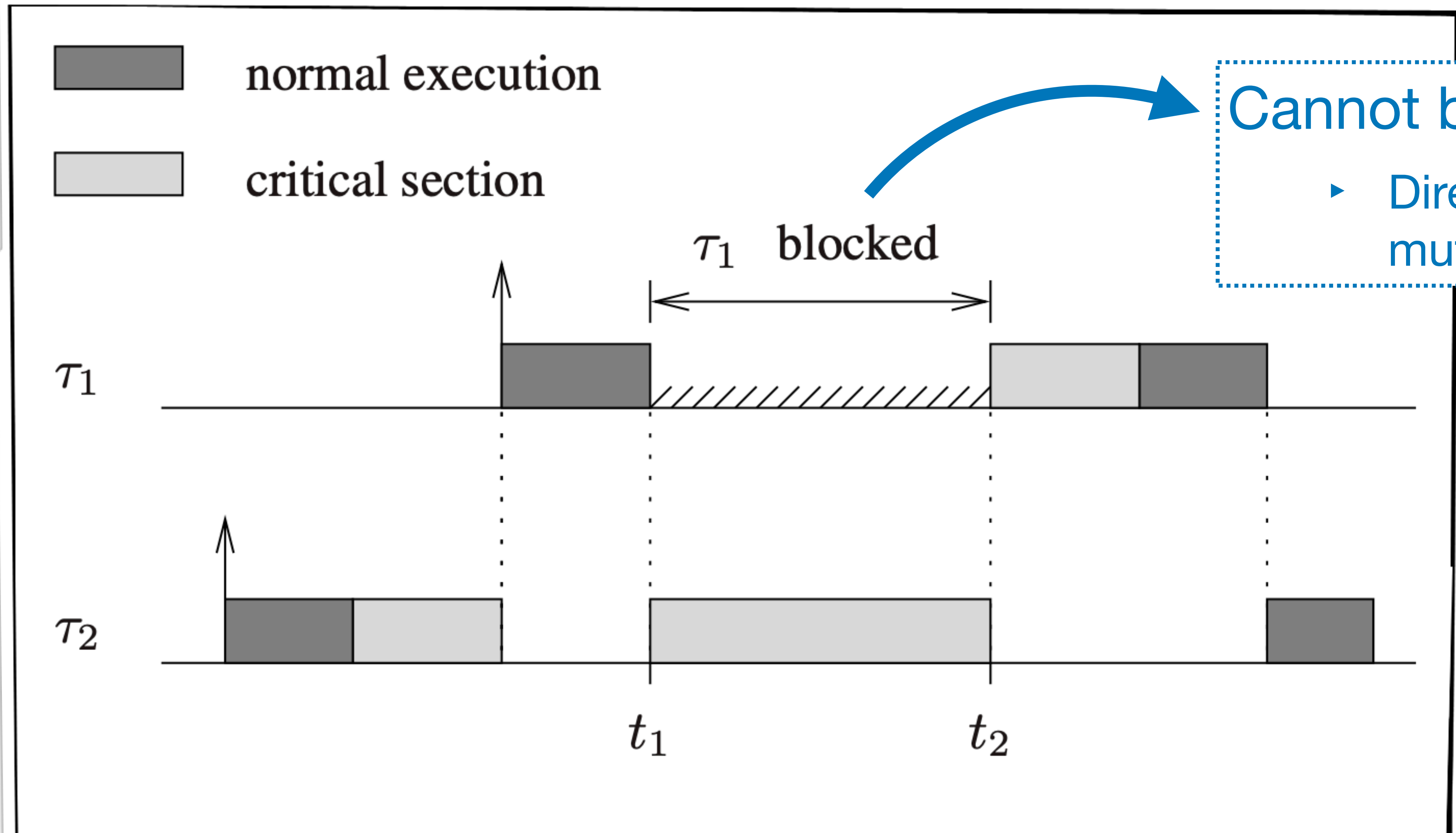

Figure 7.3 Example of blocking on an exclusive resource.

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

<u>Unbounded</u> 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 analysis will be sound but extremely pessimistic

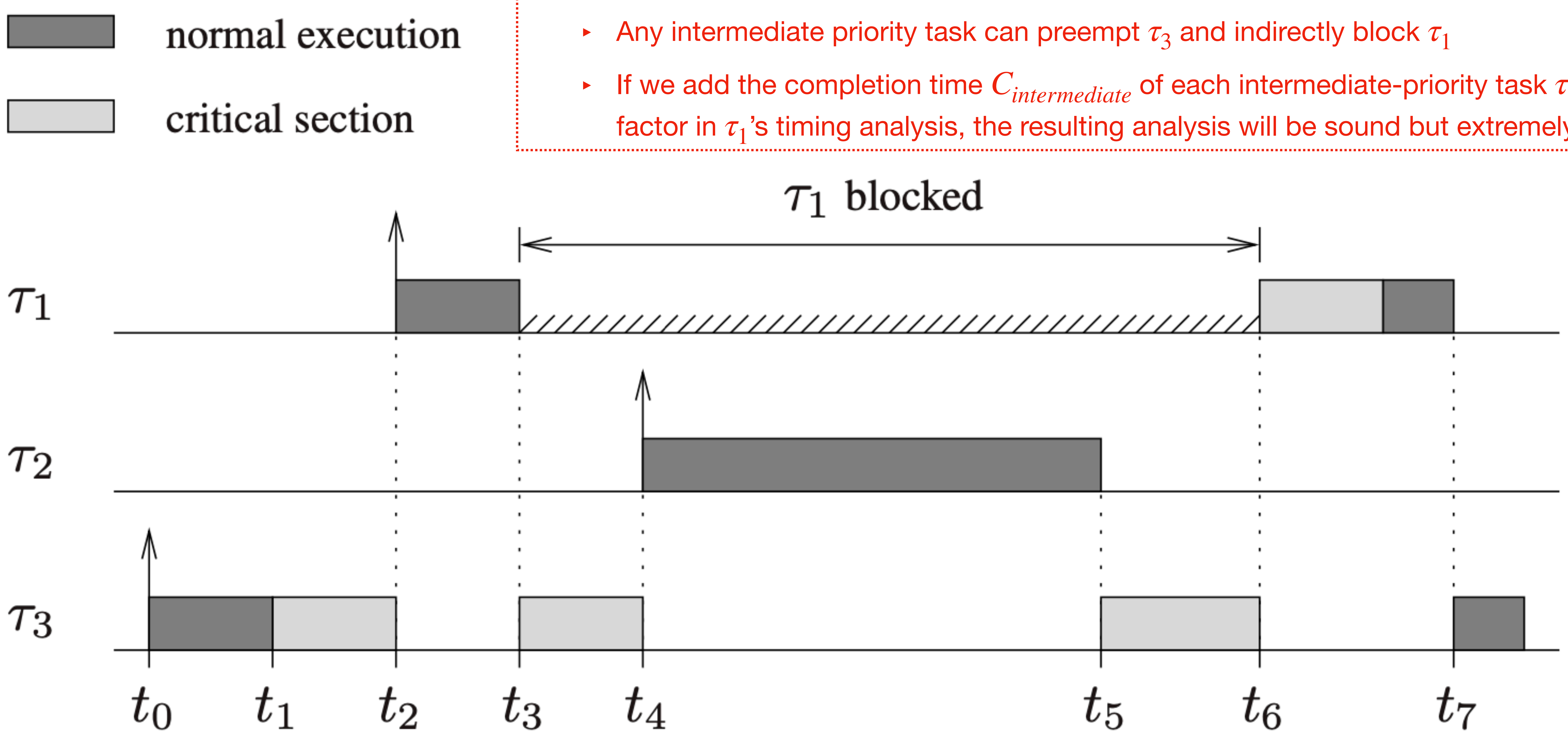

Figure 7.4 An example of priority inversion.

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., $p_i(R_k) = \max_{\forall h}\{P_h\}$

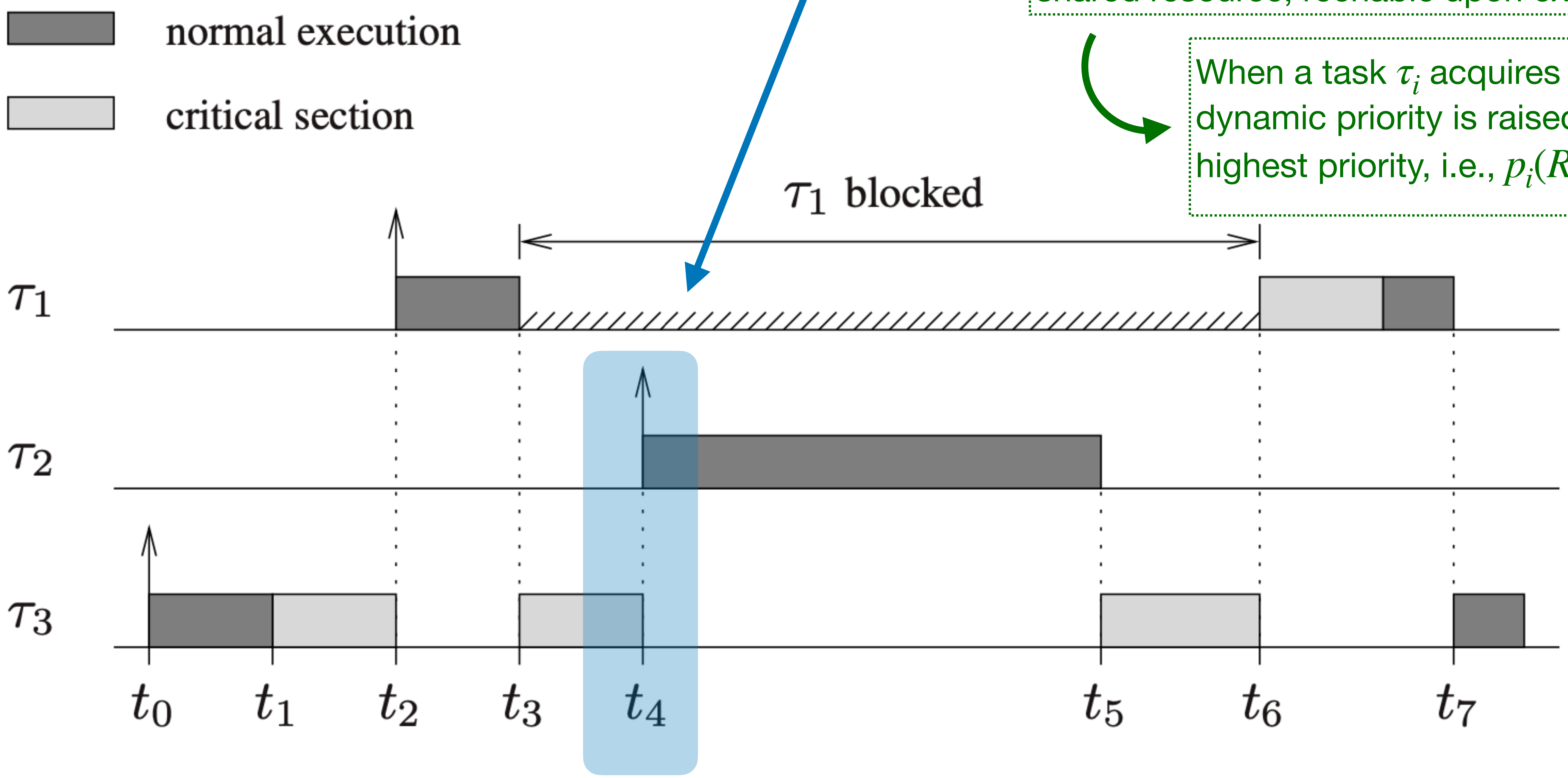

Figure 7.4 An example of priority inversion.

Example

Priority inversion **bounded by critical section length**

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

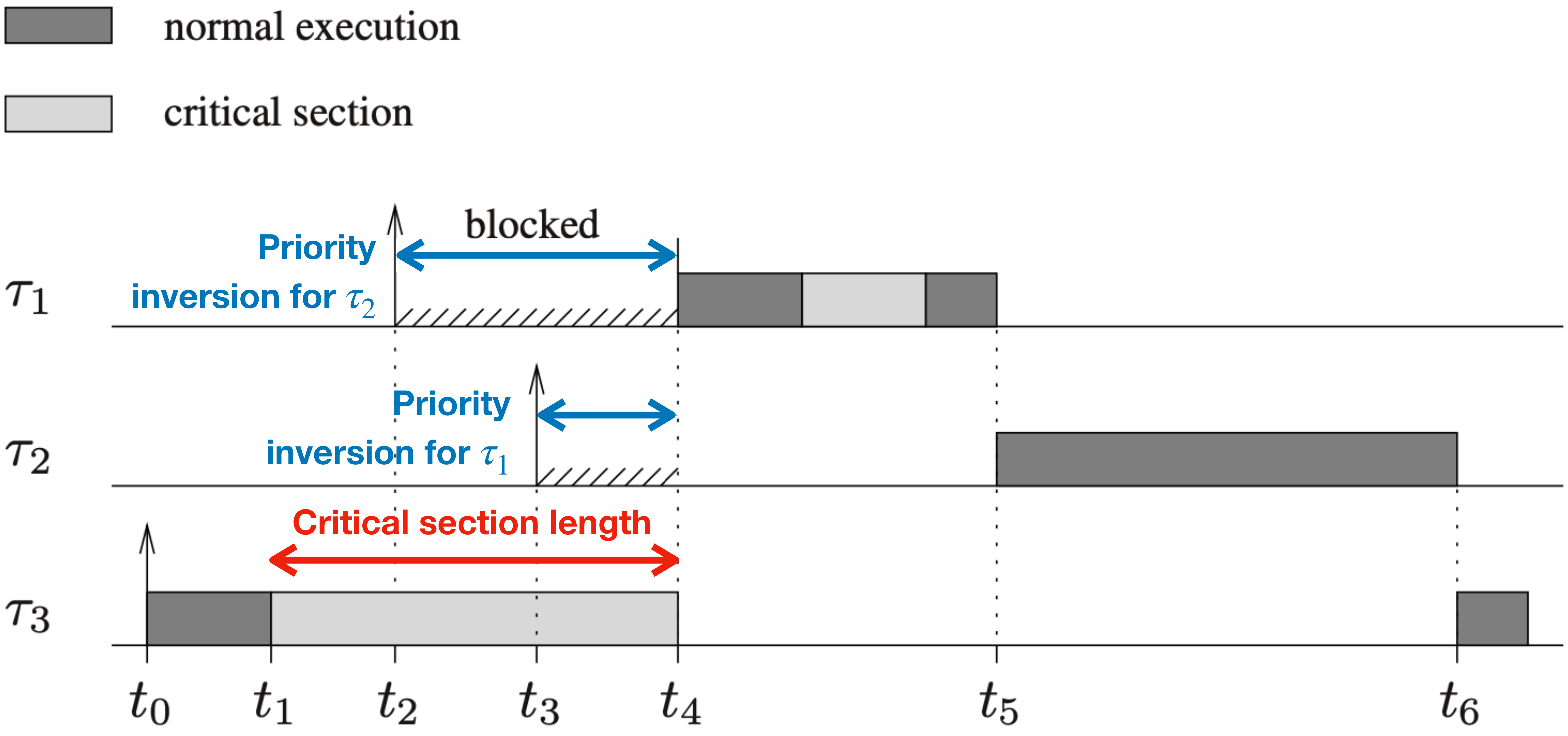

Figure 7.5 Example of NPP preventing priority inversion.

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?

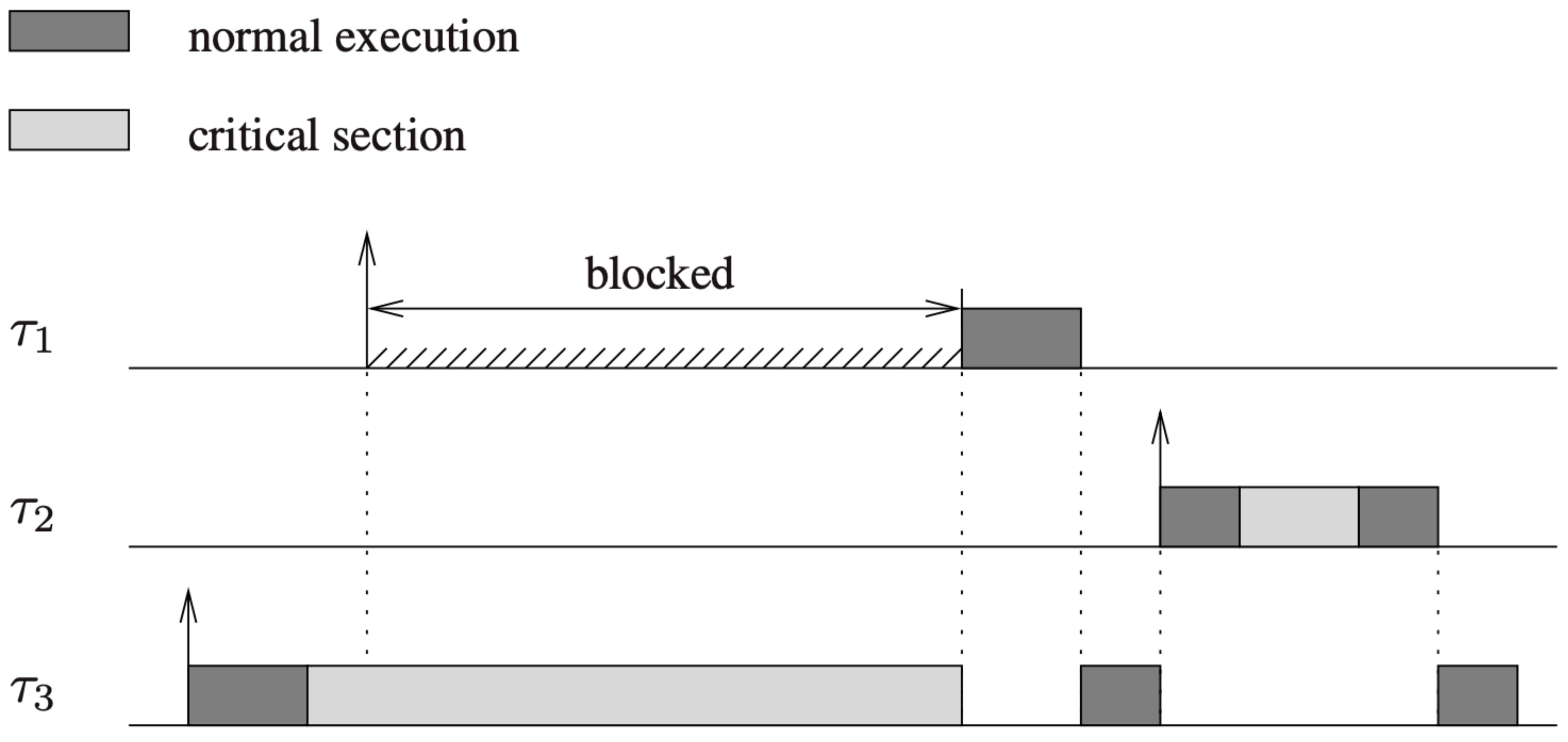

Figure 7.6 Example in which NPP causes unnecessary blocking on τ_1.

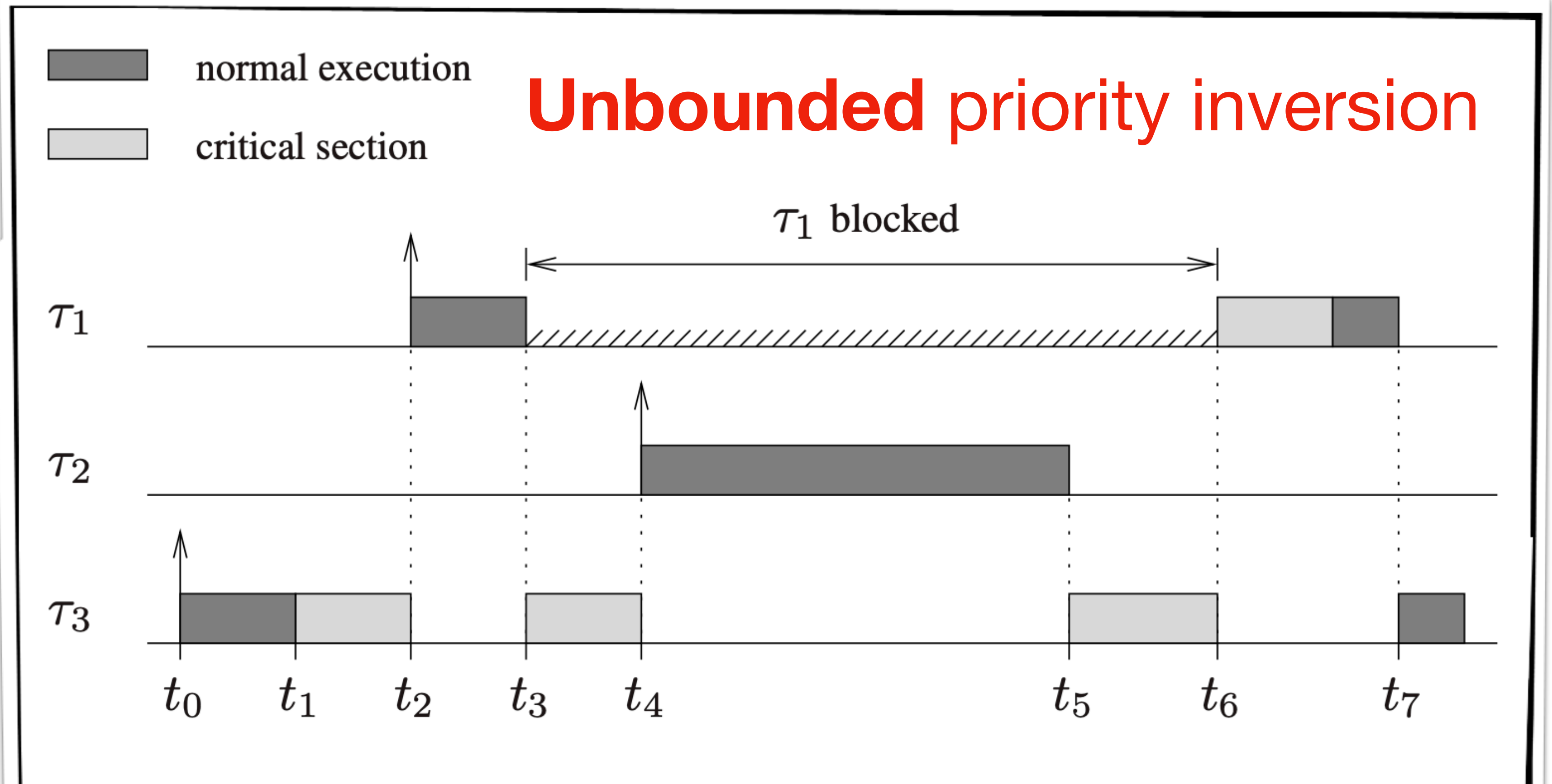

Figure 7.4 An example of priority inversion.

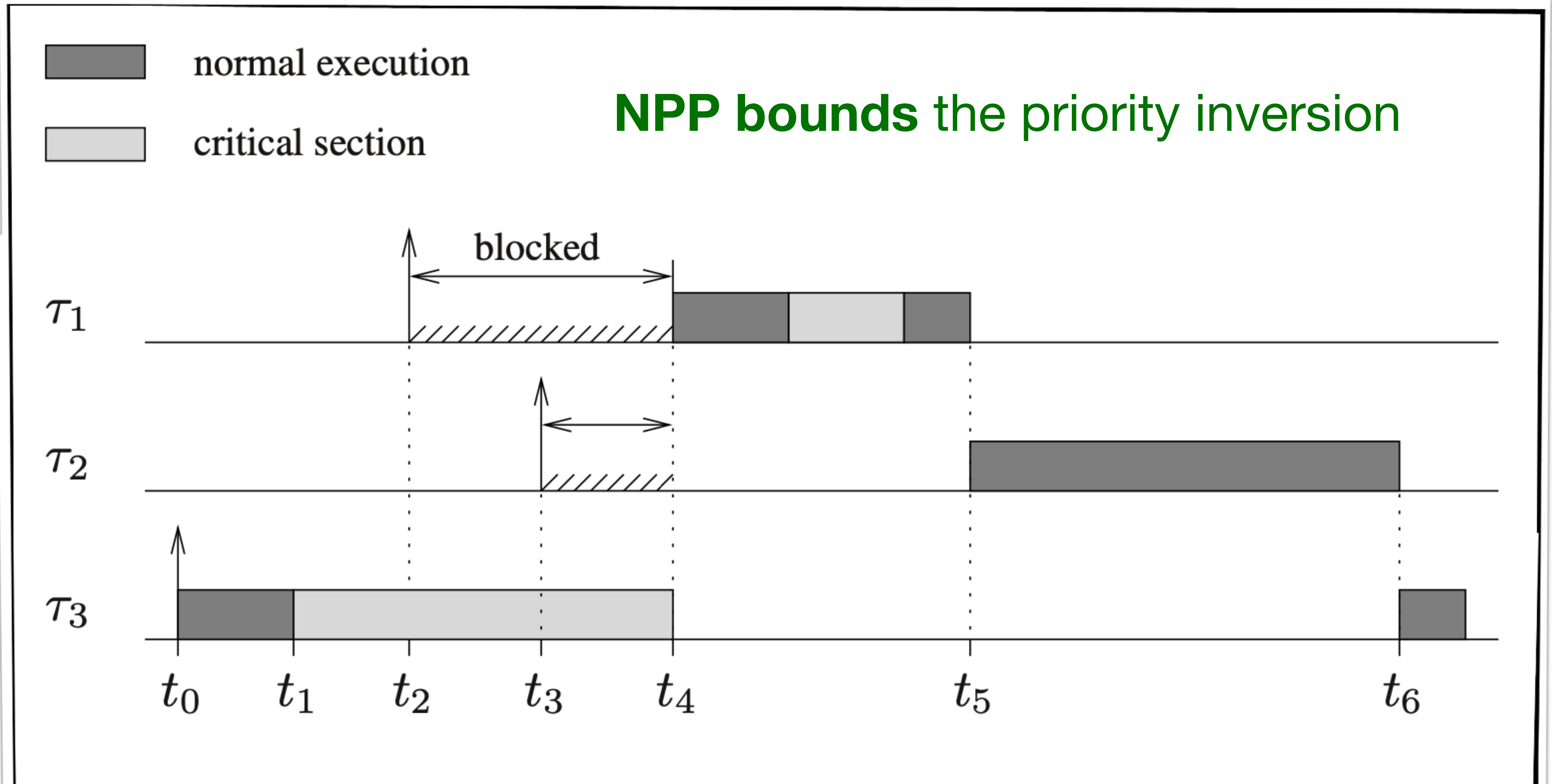

Figure 7.5 Example of NPP preventing priority inversion.

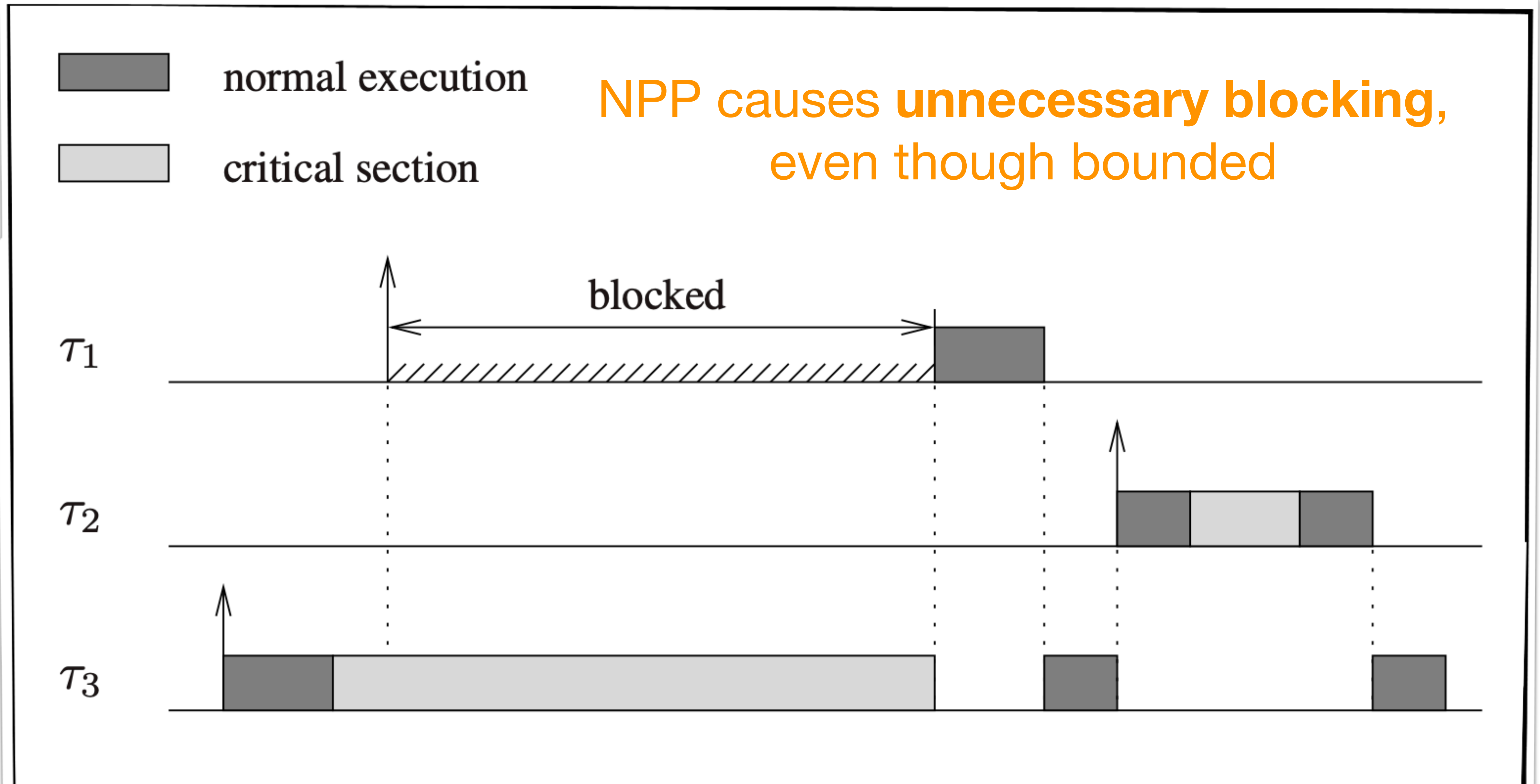

Figure 7.6 Example in which NPP causes unnecessary blocking on τ_1.

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 help 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

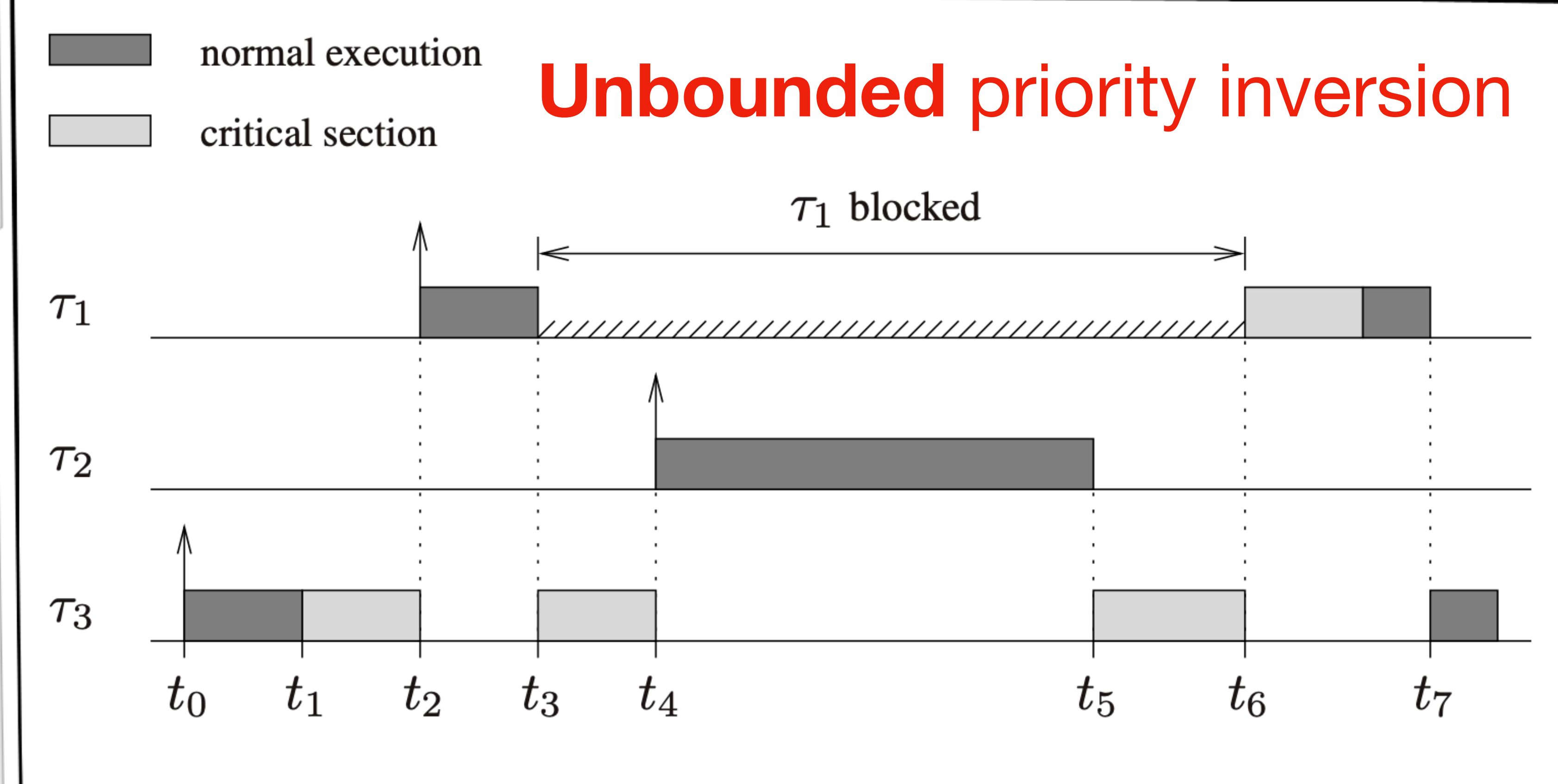

Figure 7.4 An example of priority inversion.

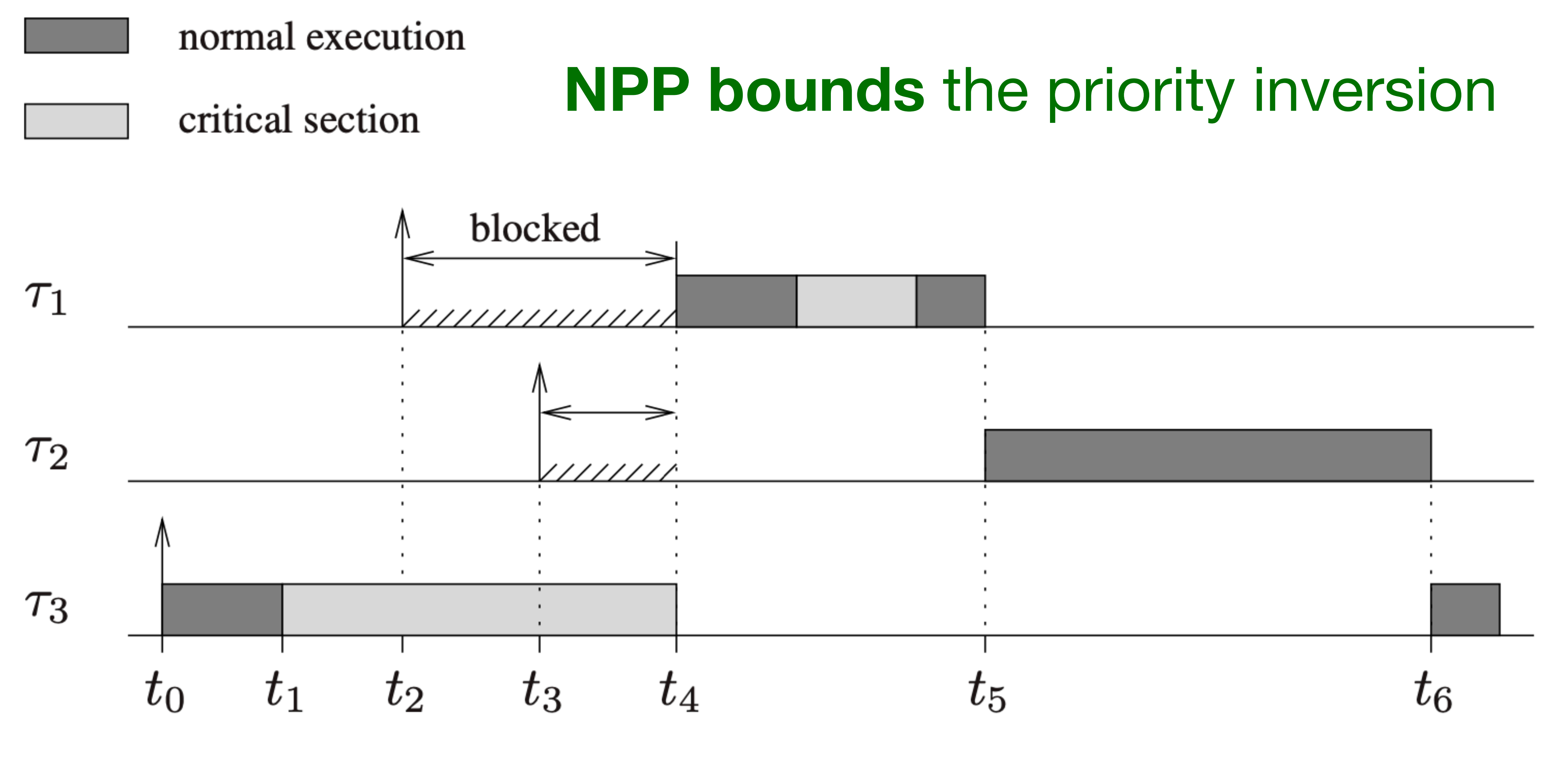

Figure 7.5 Example of NPP preventing priority inversion.

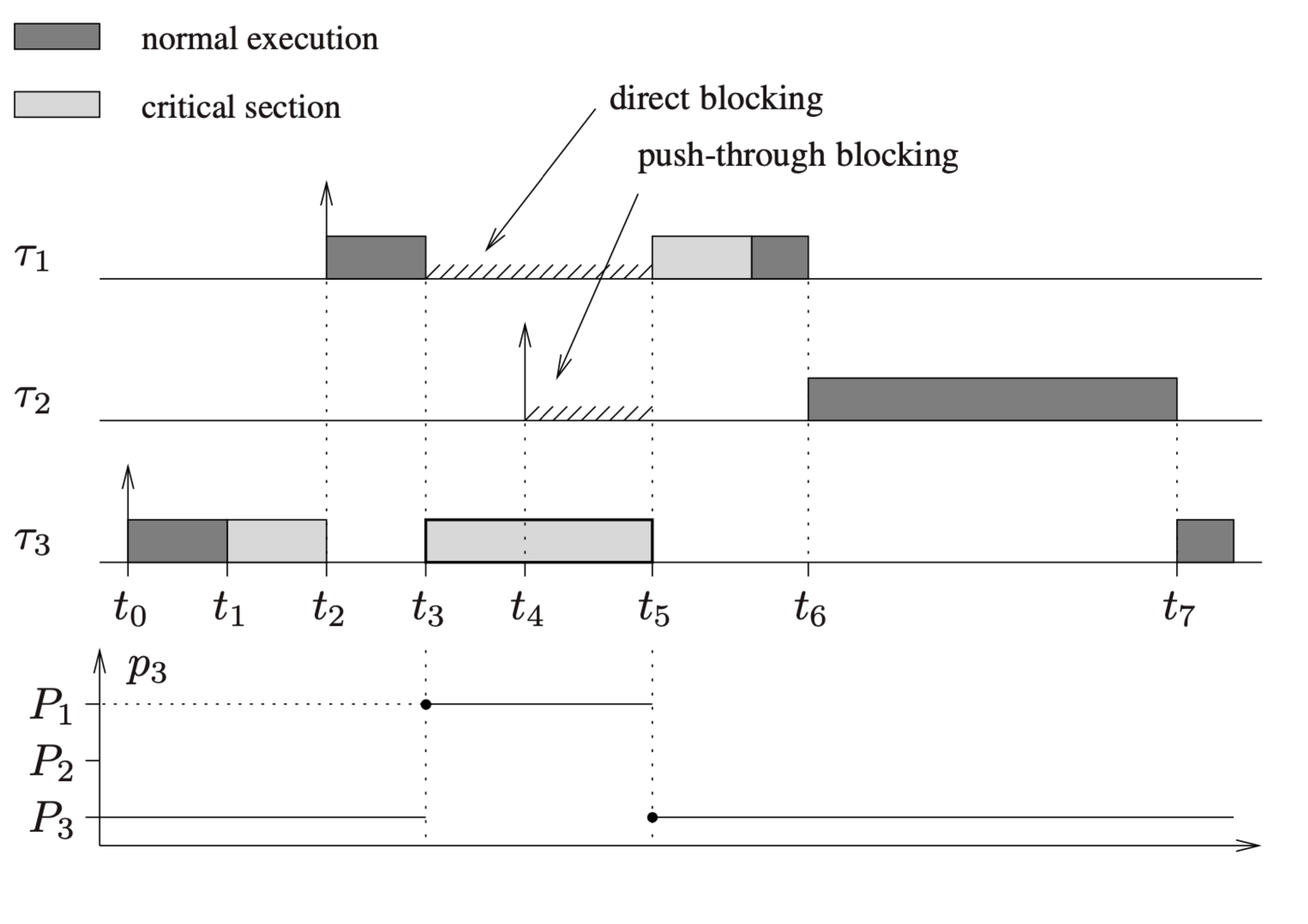

Figure 7.8 Example of Priority Inheritance Protocol.

Example 2

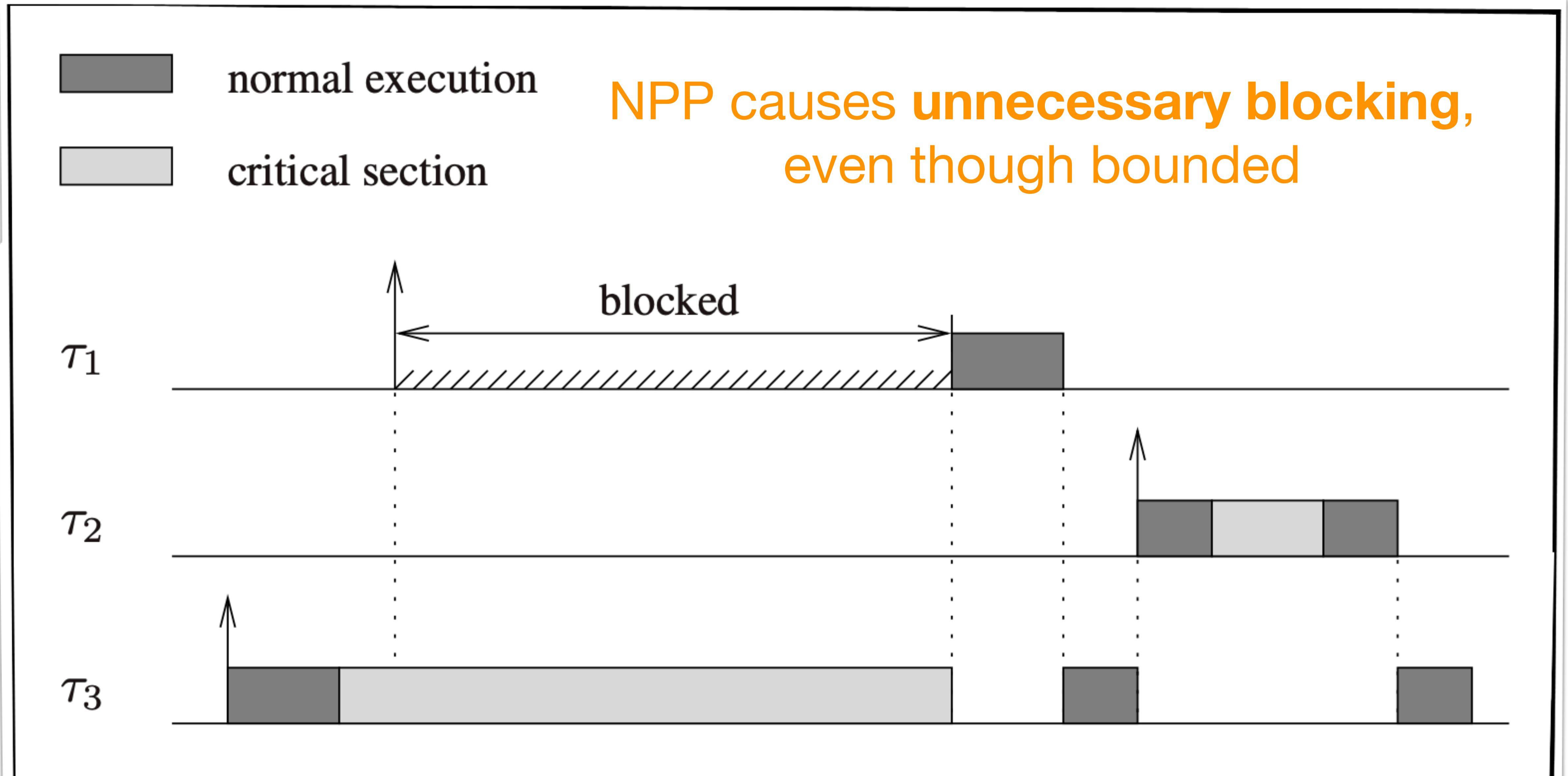

Figure 7.6 Example in which NPP causes unnecessary blocking on τ_1.

Example 3: Nested Blocking

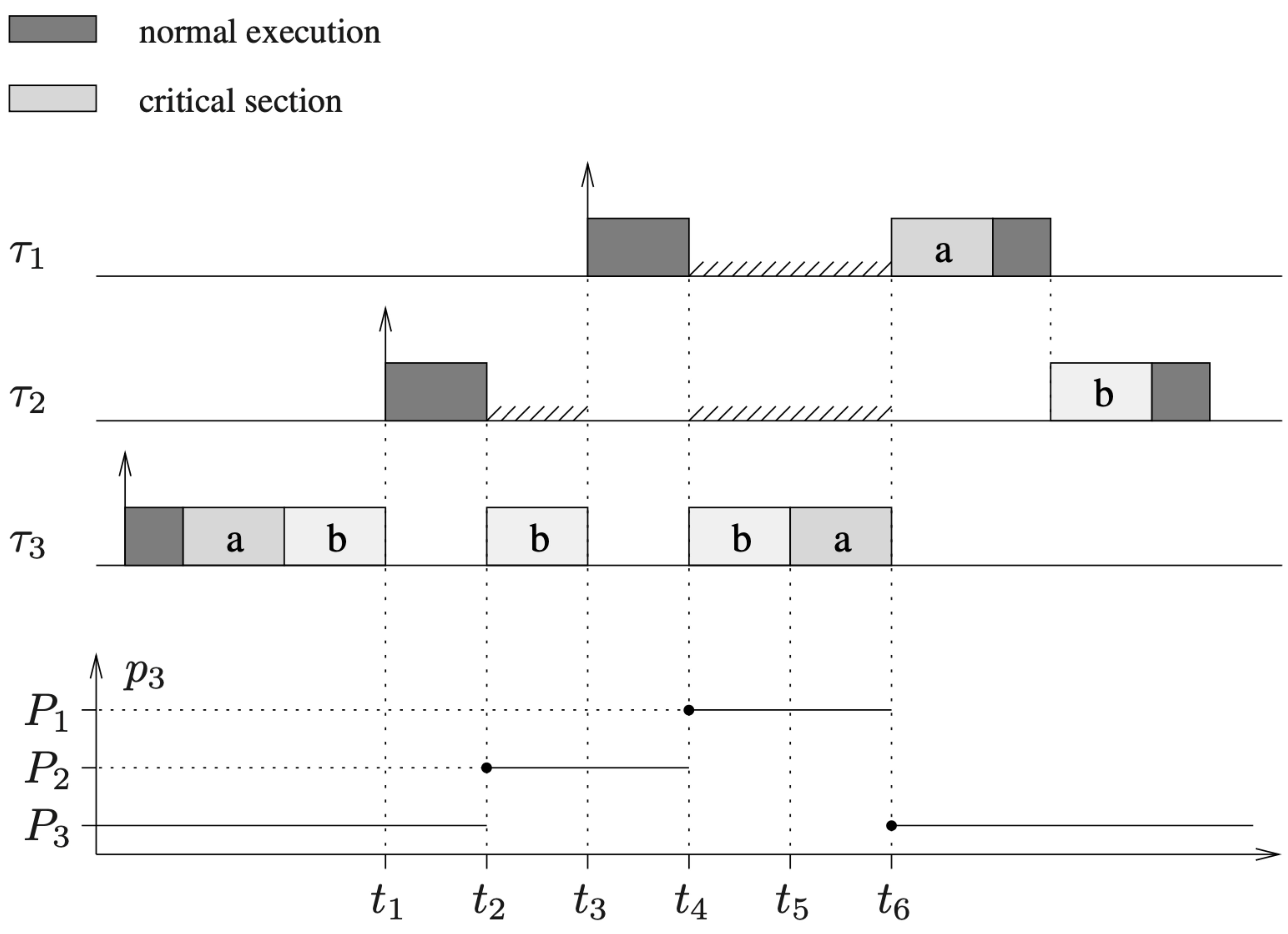

Figure 7.9 Priority inheritance with nested critical sections.

Example 4: Transitive Blocking

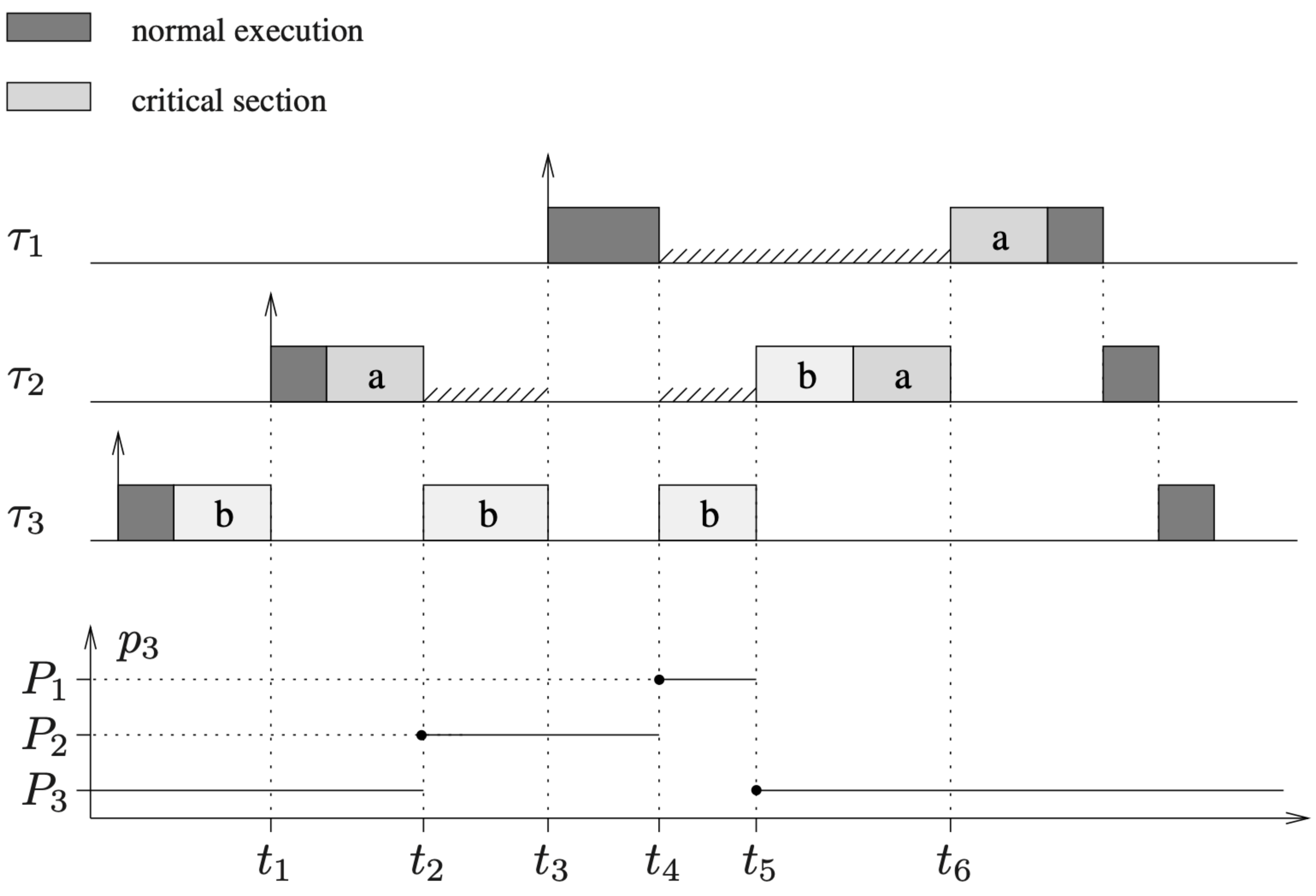

Figure 7.10 Example of transitive priority inheritance.

PIP Benefits & Limitations

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

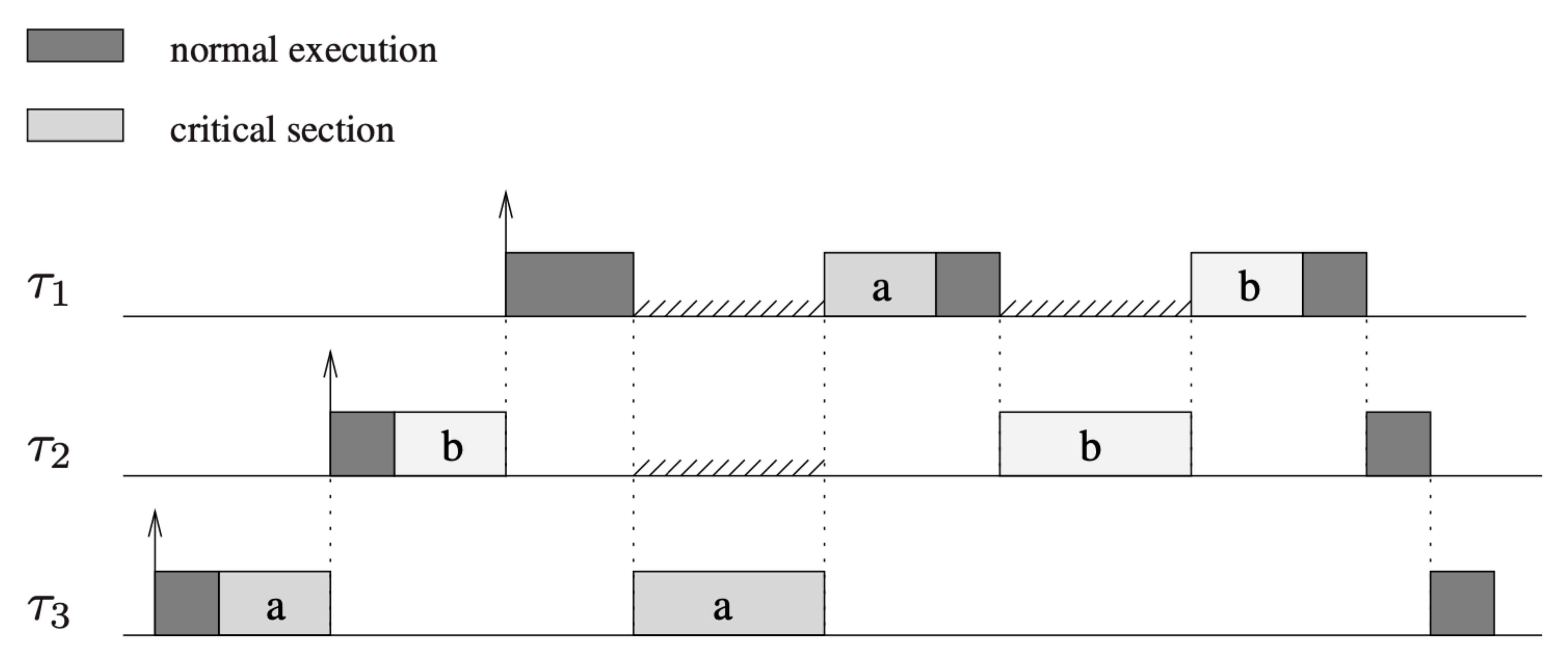

Figure 7.12 Example of chained blocking.

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