Resource Sharing

CPEN 432 Real-Time System Design

Arpan Gujarati
University of British Columbia

Terminology

- Task set $\tau = \{\tau_1, \tau_2, \dots, \tau_n\}$ 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 $R_1, R_2, \dots, R_m$
- 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 $P_1 > P_2 > \dots > P_n$
- 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 $Z_{i,k}$

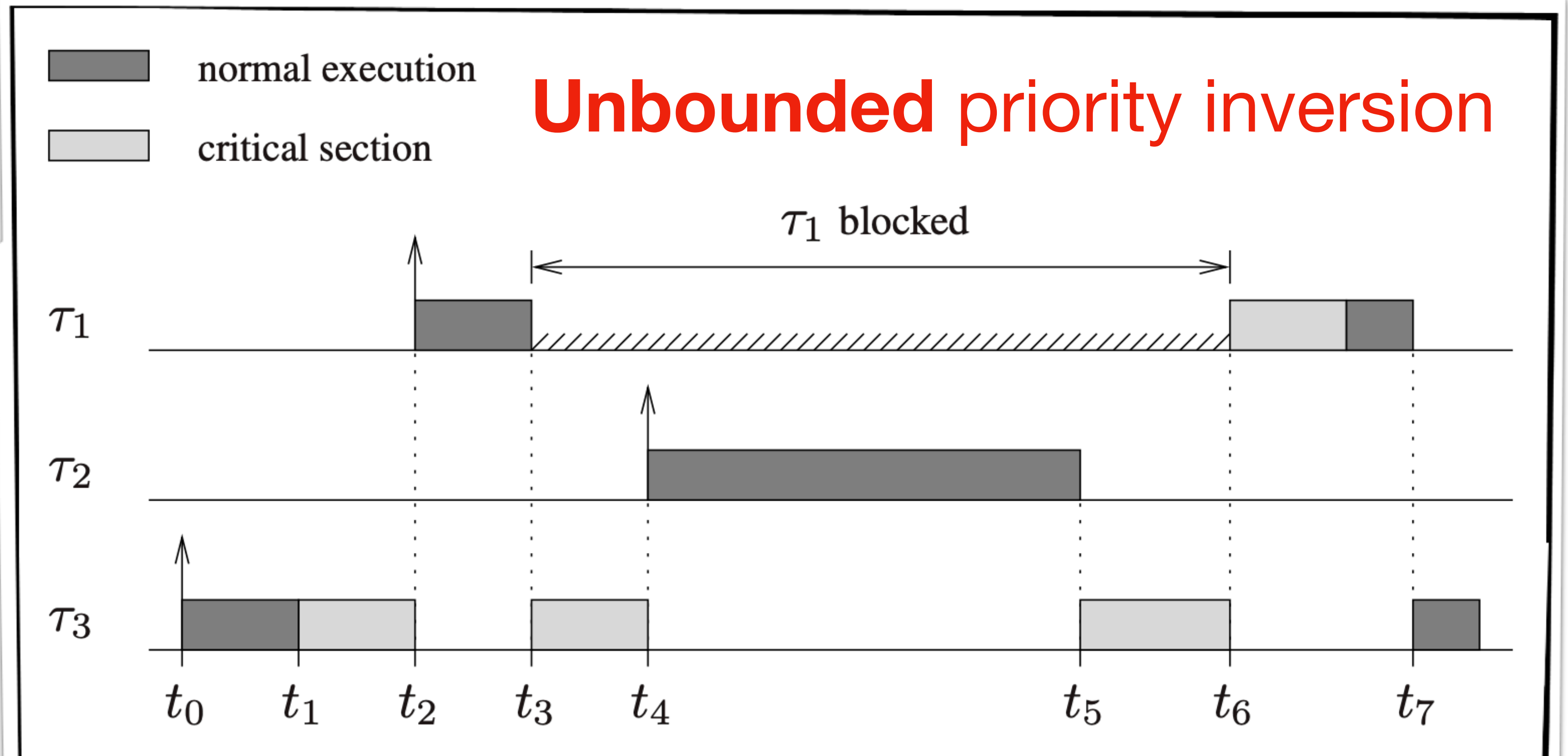


Figure 7.4 An example of priority inversion.

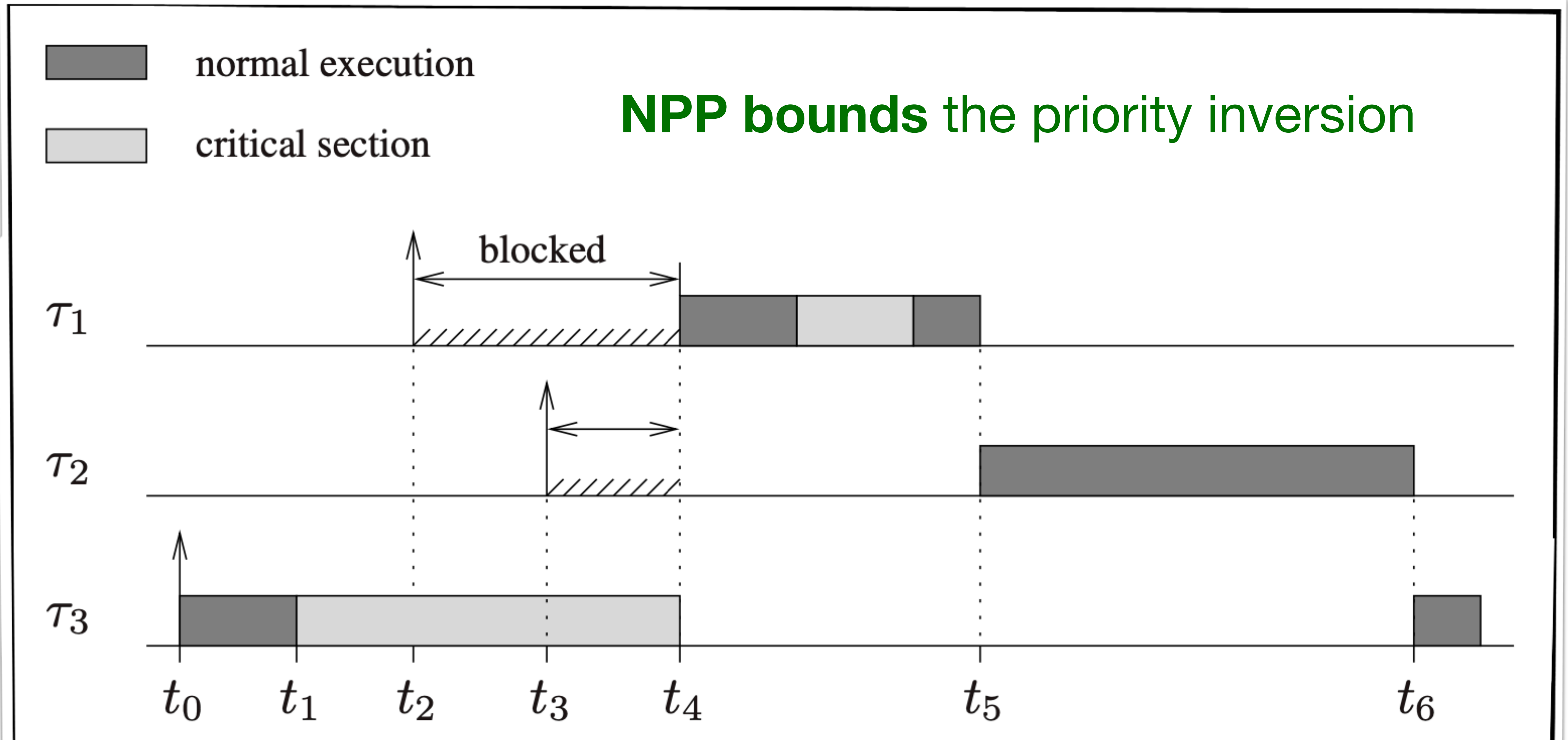


Figure 7.5 Example of NPP preventing priority inversion.

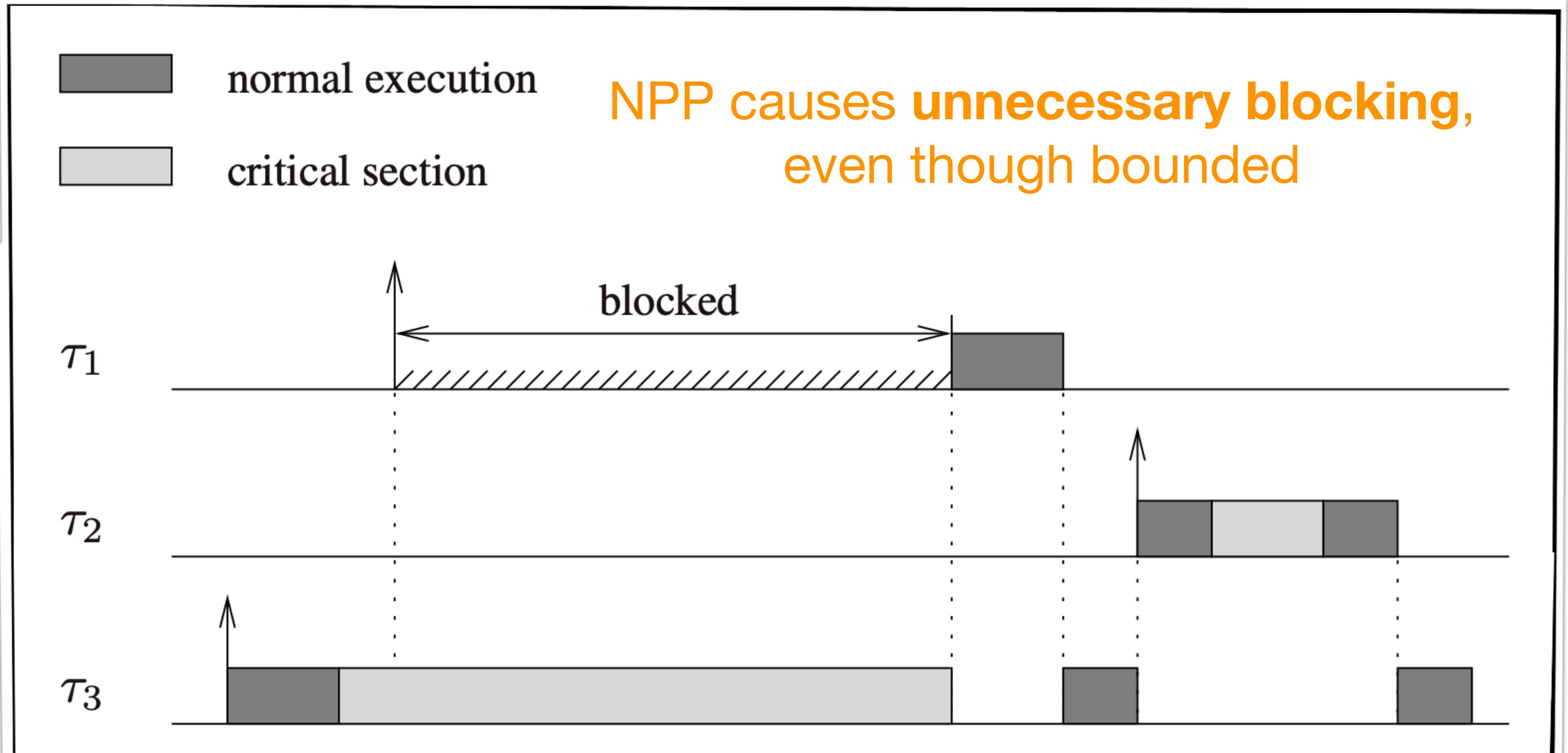


Figure 7.6 Example in which NPP causes unnecessary blocking on τ_1.

What's next?

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

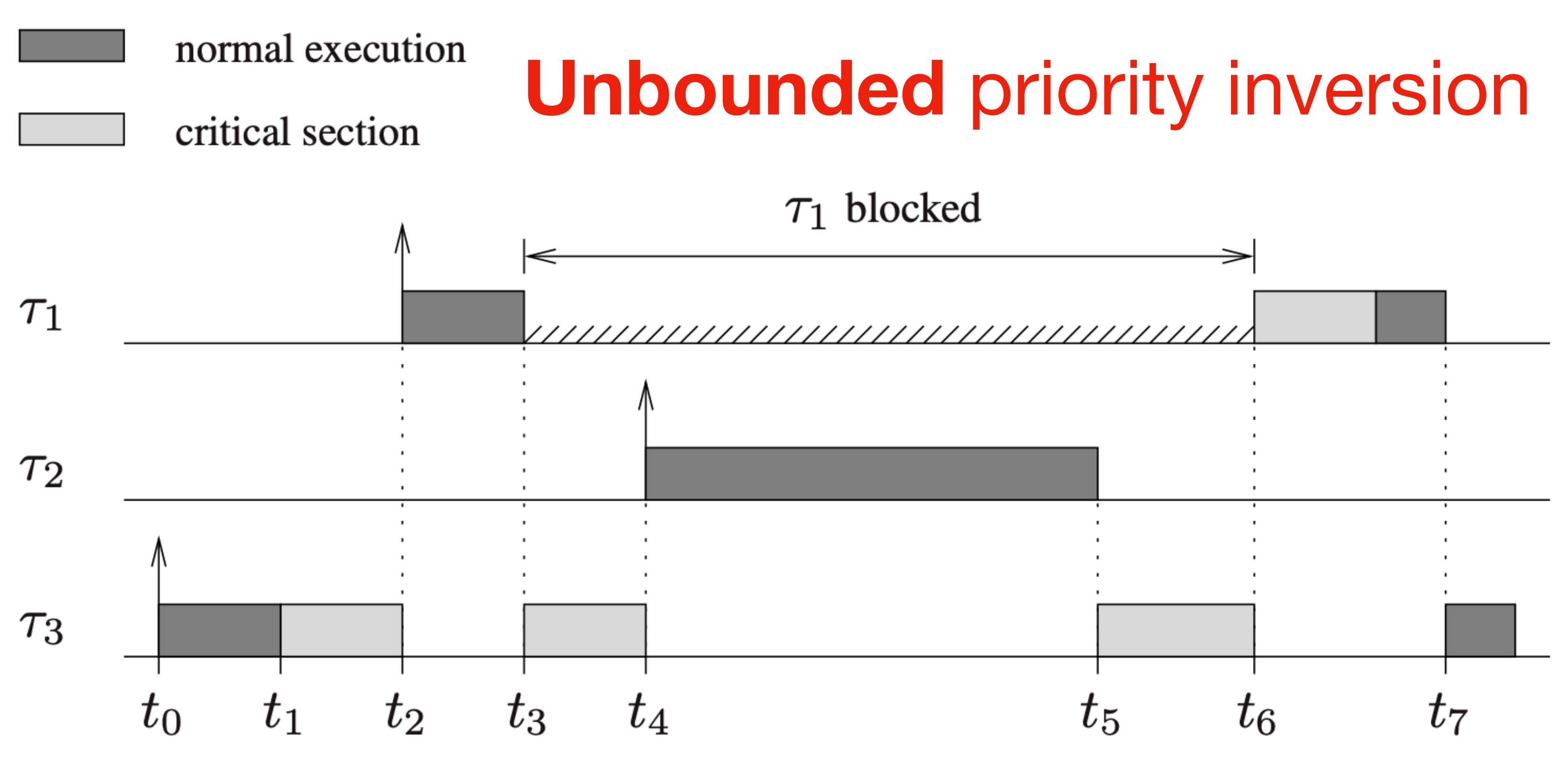


Figure 7.4 An example of priority inversion.

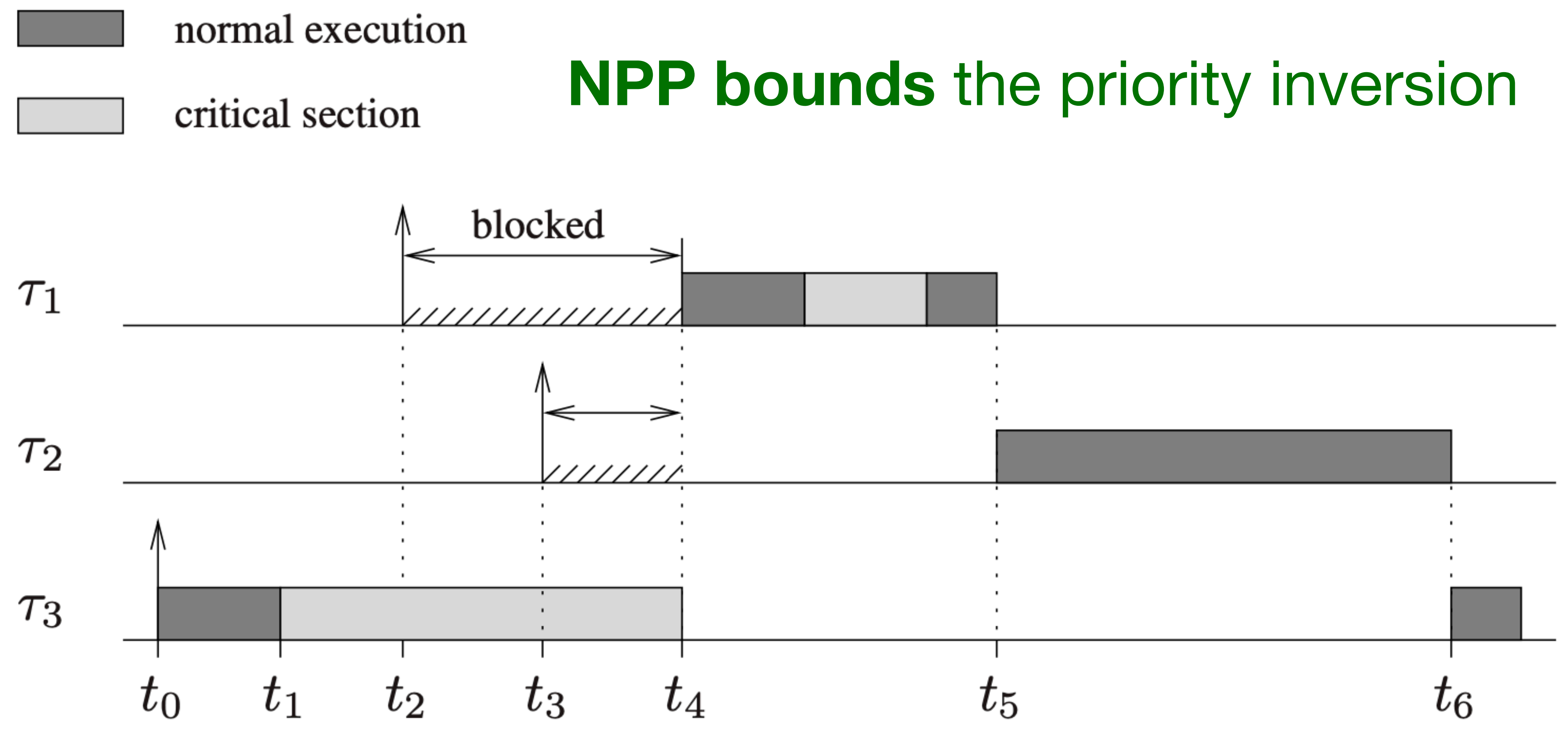


Figure 7.5 Example of NPP preventing priority inversion.

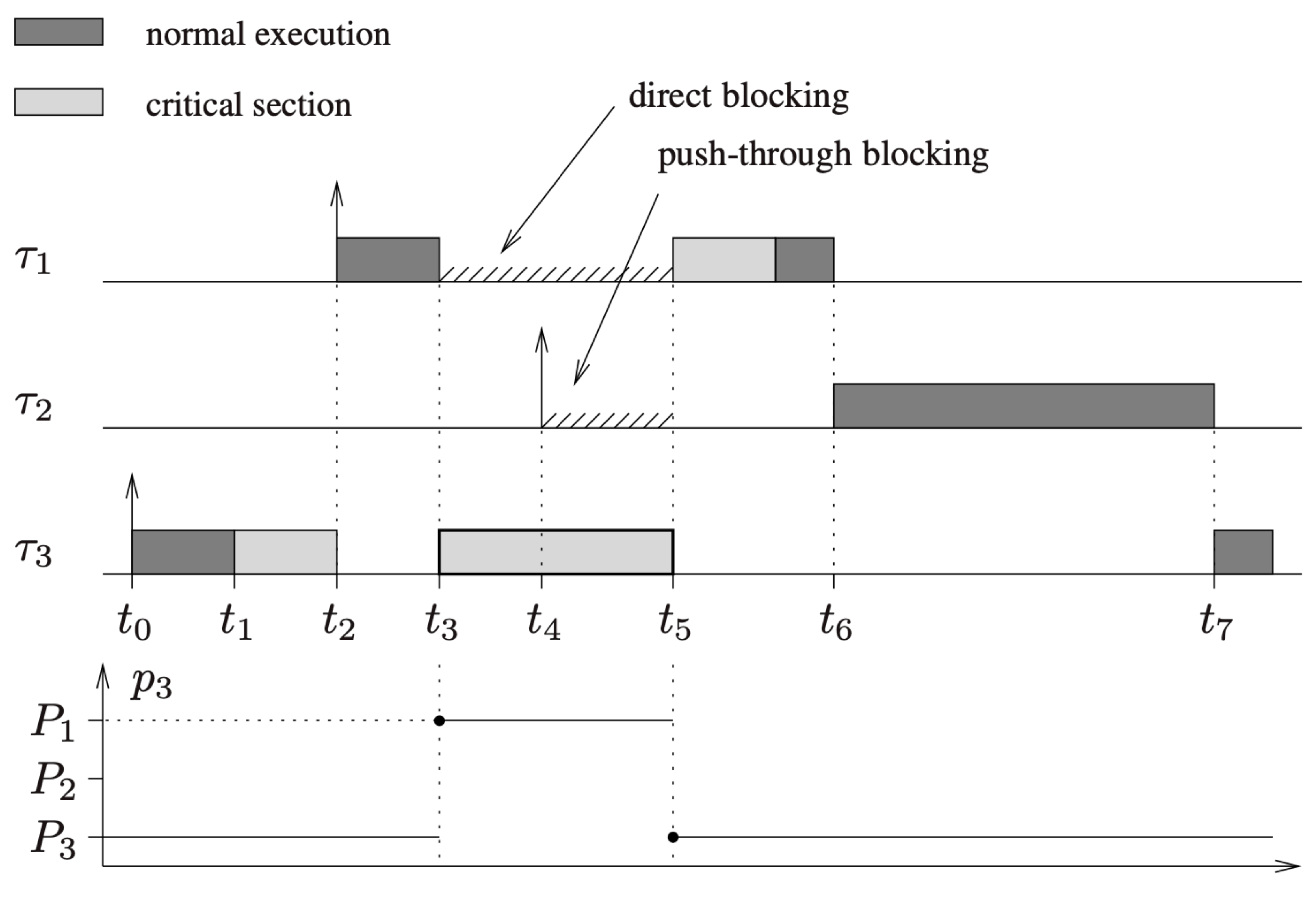


Figure 7.8 Example of Priority Inheritance Protocol.

Properties of PIP [1/5]

A semaphore S_k can cause push-through blocking to task T_i, only if S_k is accessed both by a task with priority lower than P_i and by a task with priority higher than P_i.

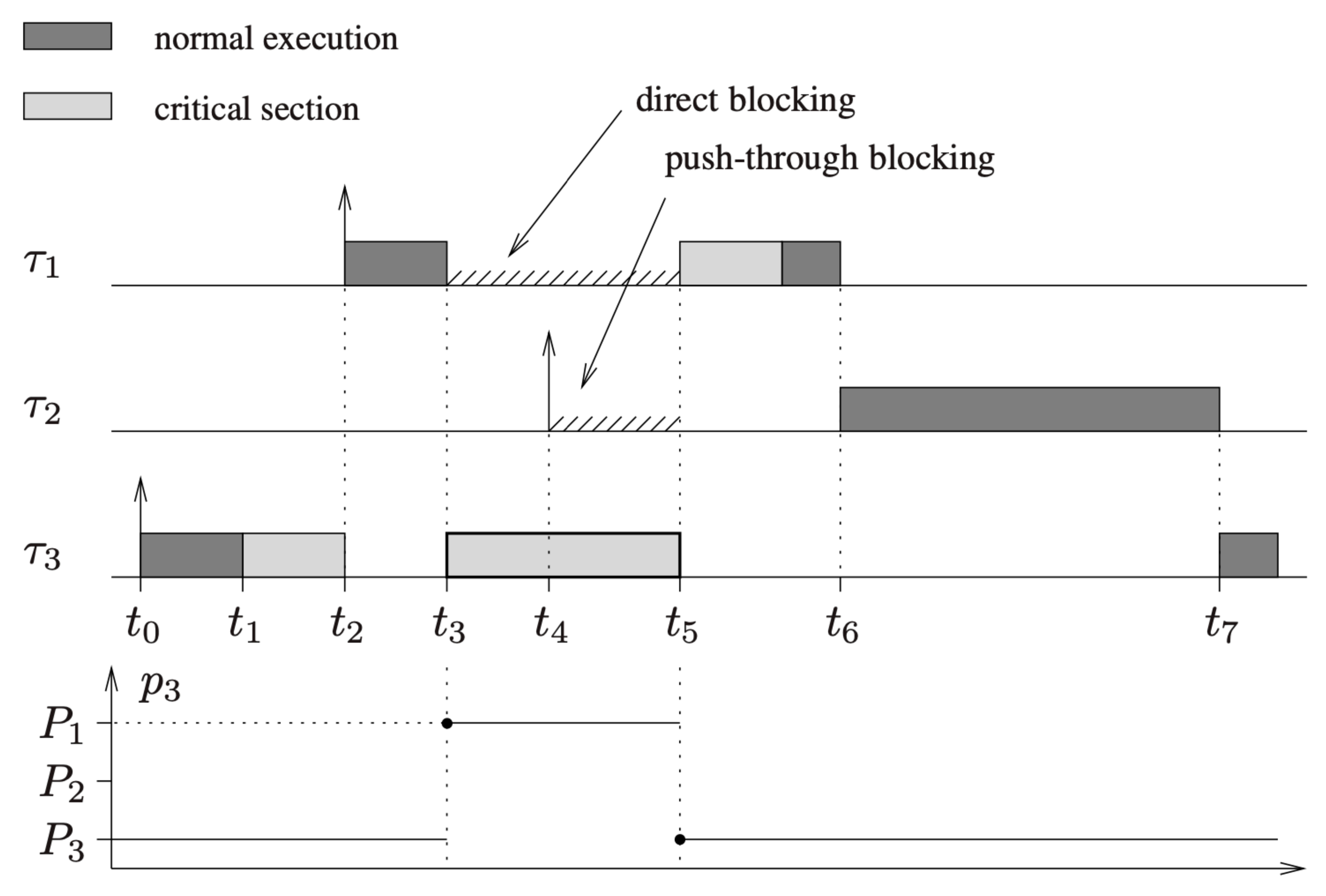


Figure 7.8 Example of Priority Inheritance Protocol.

Properties of PIP [2/5]

Transitive priority inheritance can occur only in the presence of nested critical sections.

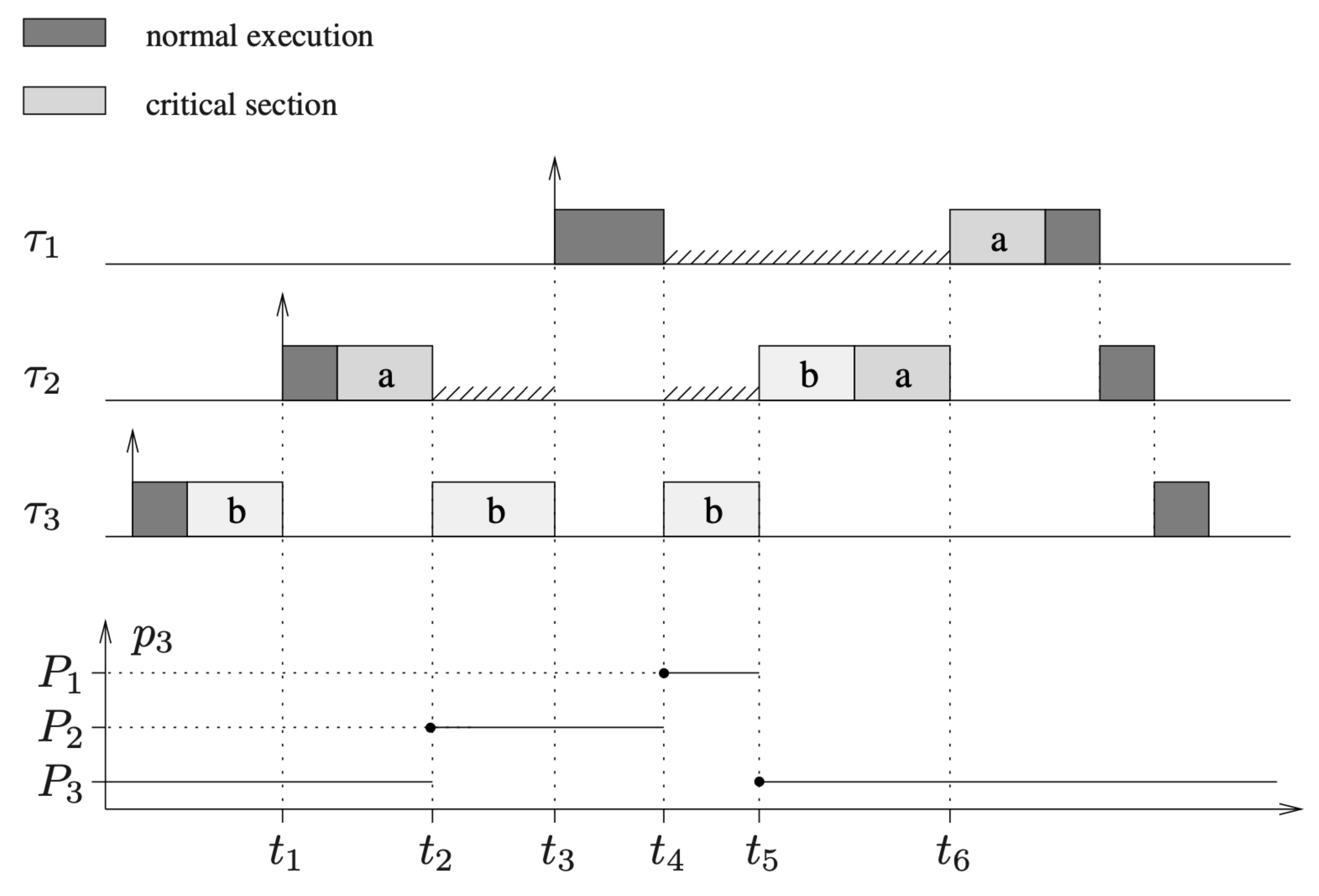


Figure 7.10 Example of transitive priority inheritance.

Properties of PIP [3/5]

If there are l_i lower-priority tasks that can block a task τ_i, then τ_i can be blocked for at most the duration of l_i critical sections, one for each of the l_i lower- priority tasks, regardless of the number of semaphores used by τ_i.

Properties of PIP [4/5]

If there are s_i *distinct semaphores that can block a task* τ_i*, then* τ_i *can be blocked for at most the duration of* s_i *critical sections, one for each of the* s_i *semaphores, regardless of the number of critical sections used by* τ_i*.*

Properties of PIP [5/5]

Under the Priority Inheritance Protocol, a task τ_i can be blocked for at most the duration of $\alpha_i = min(l_i, s_i)$ critical sections, where l_i is the number of lower-priority tasks that can block τ_i and s_i is the number of distinct semaphores that can block τ_i.

Computing Blocking Time B_i [1/2]

- A precise evaluation of the blocking factor B_i is quite complex because each critical section of the lower-priority tasks may interfere with τ_i via **direct blocking**, **push-through blocking**, or **transitive inheritance**

- Simplified algorithm
  - Assumes **no nested critical sections**, hence **no transitive inheritance**

Computing Blocking Time B_i [2/2]

- Semaphores that can **directly** block τ_i and that are shared by the lower-priority task τ_j are $\sigma^{dir}_{i,j} = \sigma_i \cap \sigma_j$
- Semaphores that can block τ_i by **push-through** and that are shared by the lower-priority task τ_j are $\sigma^{pt}_{i,j} = \cup_{h:P_h>P_i} \sigma_h \cap \sigma_j$
- Semaphores that can block τ_i either **directly or by push-through** and that are shared by the lower-priority task τ_j
  - $\sigma_{i,j} = \sigma^{dir}_{i,j} \cup \sigma^{dir}_{i,j} = \cup_{h:P_h \geq P_i} \sigma_h \cap \sigma_j$
- Longest critical sections used by lower-priority task τ_j that can block τ_i either directly or by push-through is
  - $\gamma_{i,j} = \{Z_{j,k} | R_k \in \sigma_{i,j}\}$
- Lll critical sections that can block τ_i either directly or by push-through is $\gamma_i = \cup_{j:P_j<P_i} \gamma_{i,j}$
- B_i is given by the largest sum of the lengths of the α_i critical sections in γ_i
  - The sum should contain only terms $\delta_{i,k}$ referring to different tasks and different semaphore

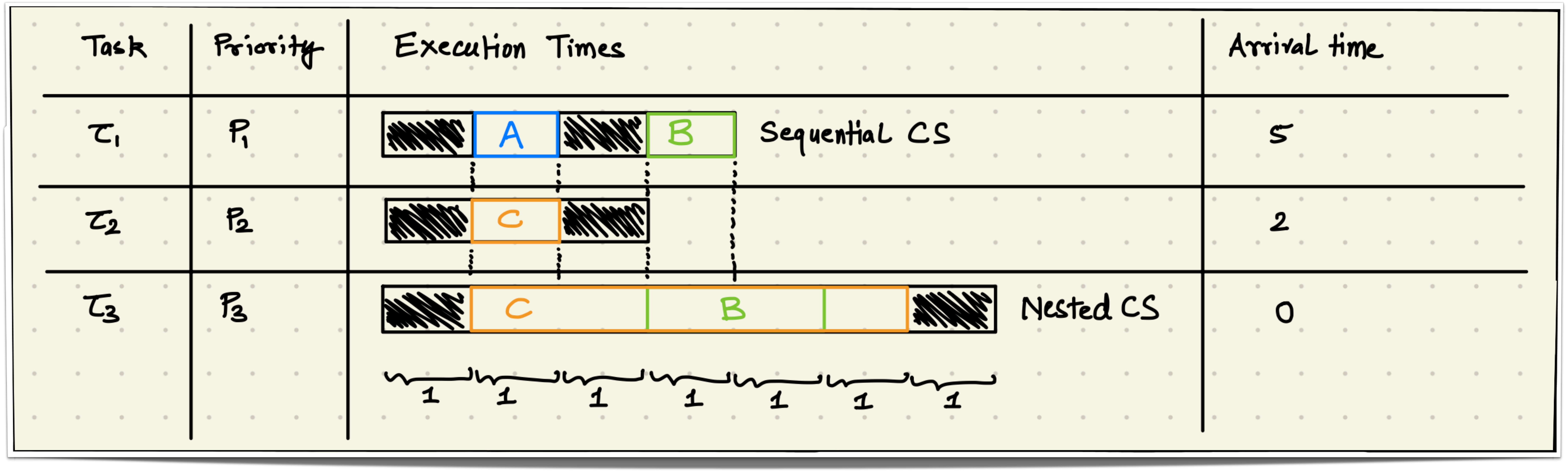
The Priority Ceiling Protocol (PCP)

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

PCP 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**



Properties of PCP [1/4]

- PCP prevents transitive blocking

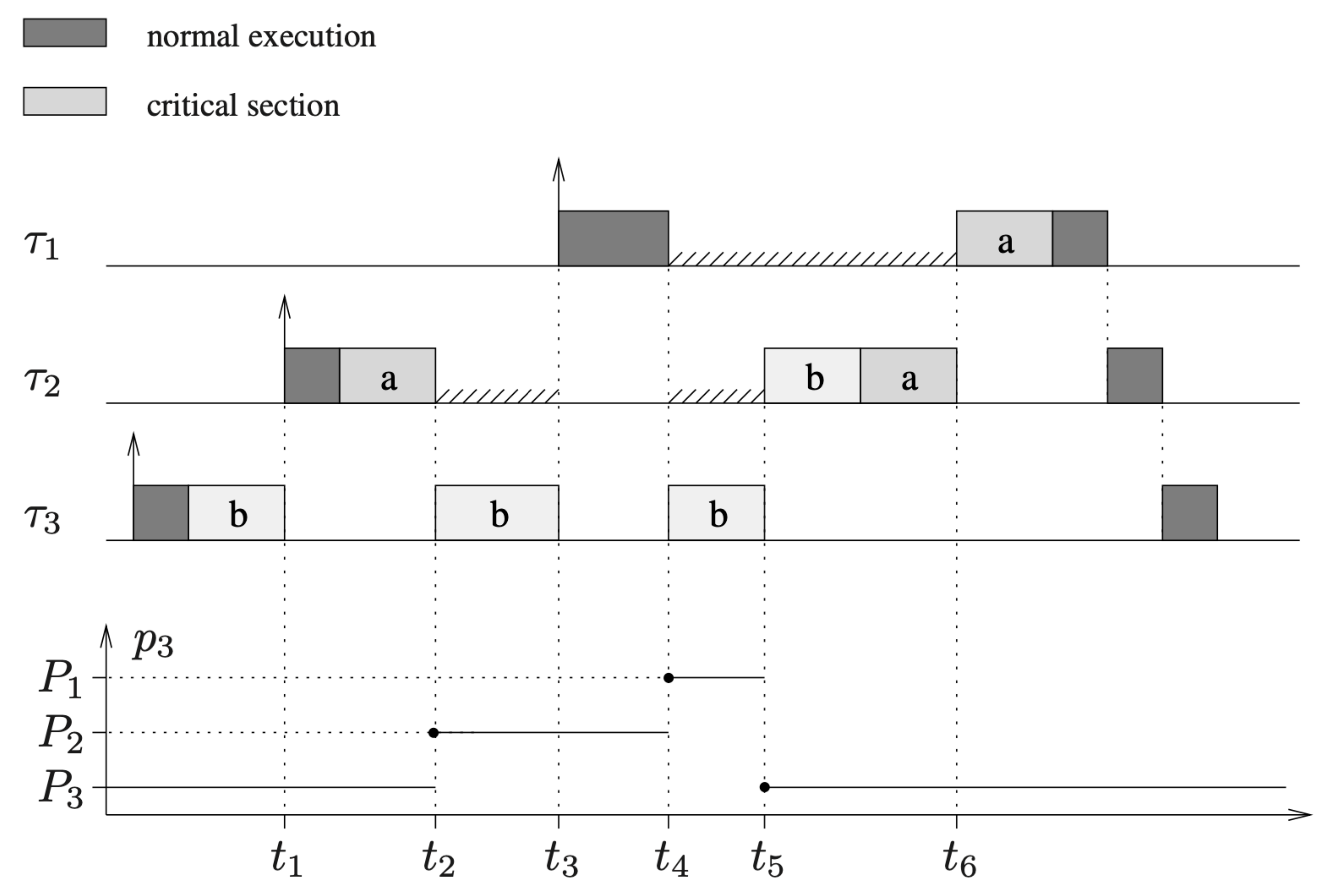


Figure 7.10 Example of transitive priority inheritance.

Properties of PCP [2/4]

- PCP prevents deadlocks

Properties of PCP [3/4]

- A task τ_i can be blocked for **at most** the duration of **one** critical section

Properties of PCP [4/4]

- A critical section $z_{i,k}$ belonging to task τ_j and guarded by semaphore S_k can block a task τ_i only if $P_j < P_i$ and $C_{global}(S_k) \geq P_i$

Computing Blocking Time B_i

Schedulability Analysis with Resource Sharing

Key Ideas

- Schedulability analysis of task τ_i
  - **Inflate the computation time** C_i of by the blocking factor B_i
- All exact tests (both necessary and sufficient) become **only sufficient**
  - Blocking conditions are derived in worst-case scenarios that differ for each task and **may never occur simultaneously**
- Examples
  - RM utilization bound $\forall i = 1, \ldots, n : \sum_{h:P_h > P_i} \frac{C_h}{T_h} + \frac{C_i + B_i}{T_i} \leq i(2^{1/i} - 1)$ (for EDF, replace RHS with 1)
  - Response-time analysis $R_i^{(s)} = C_i + B_i + \sum_{h:P_h > P_i} \left\lceil \frac{R_i^{(s-1)}}{T_h} \right\rceil C_h$