Resource Sharing

CPEN 432 Real-Time System Design

Arpan Gujarati
University of British Columbia

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 $Z_{i,k}$

The Priority Ceiling Protocol (PCP)

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

Analytically, PCP is better than PIP

- Like PIP …

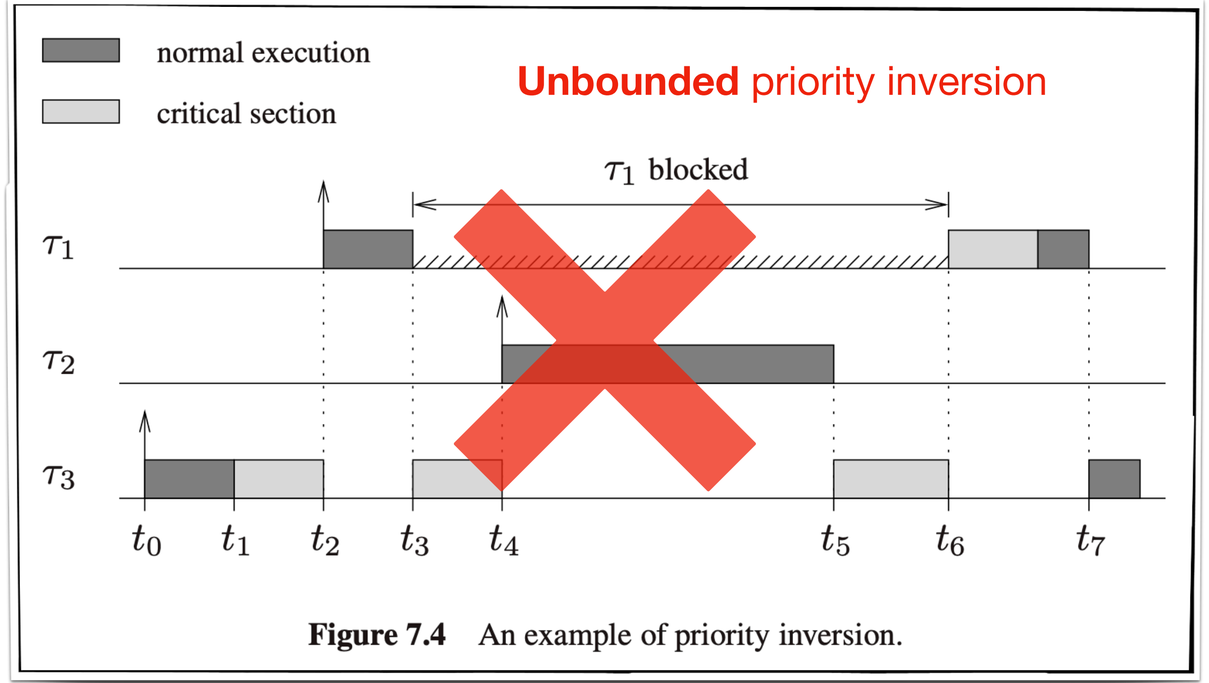


Figure 7.4 An example of priority inversion.

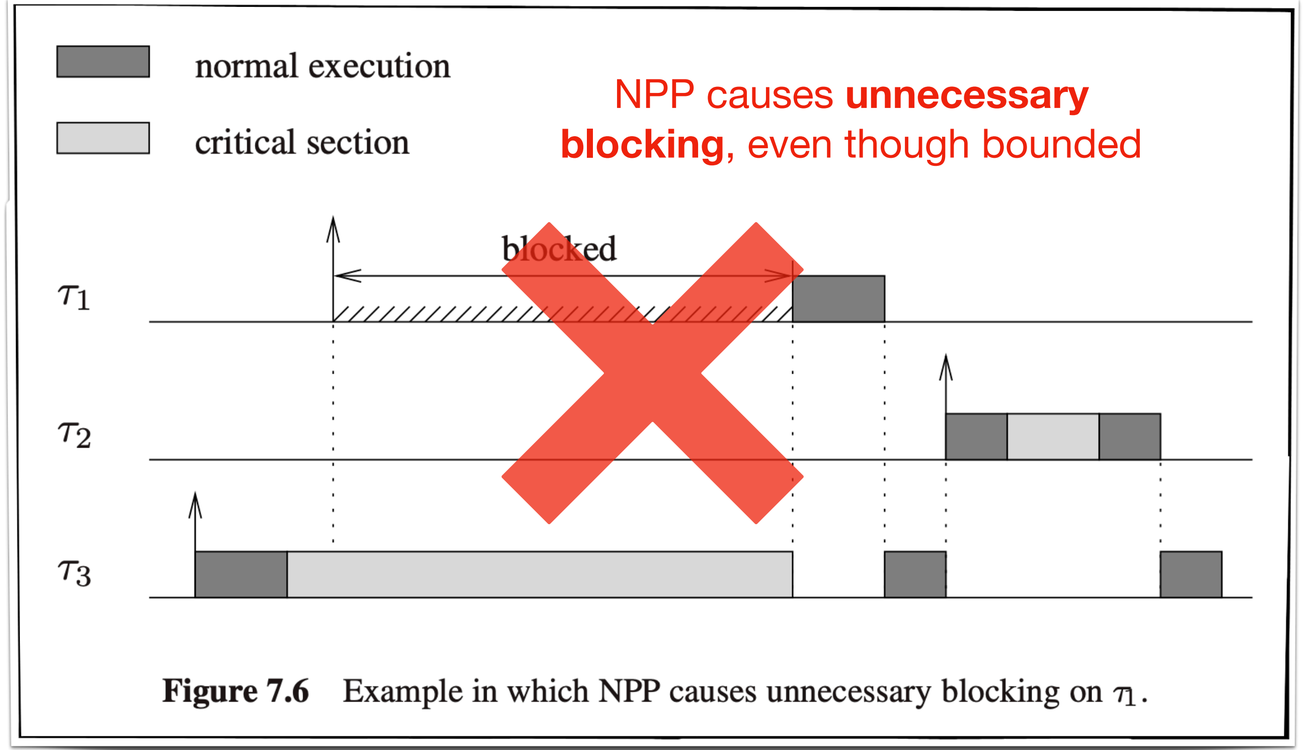
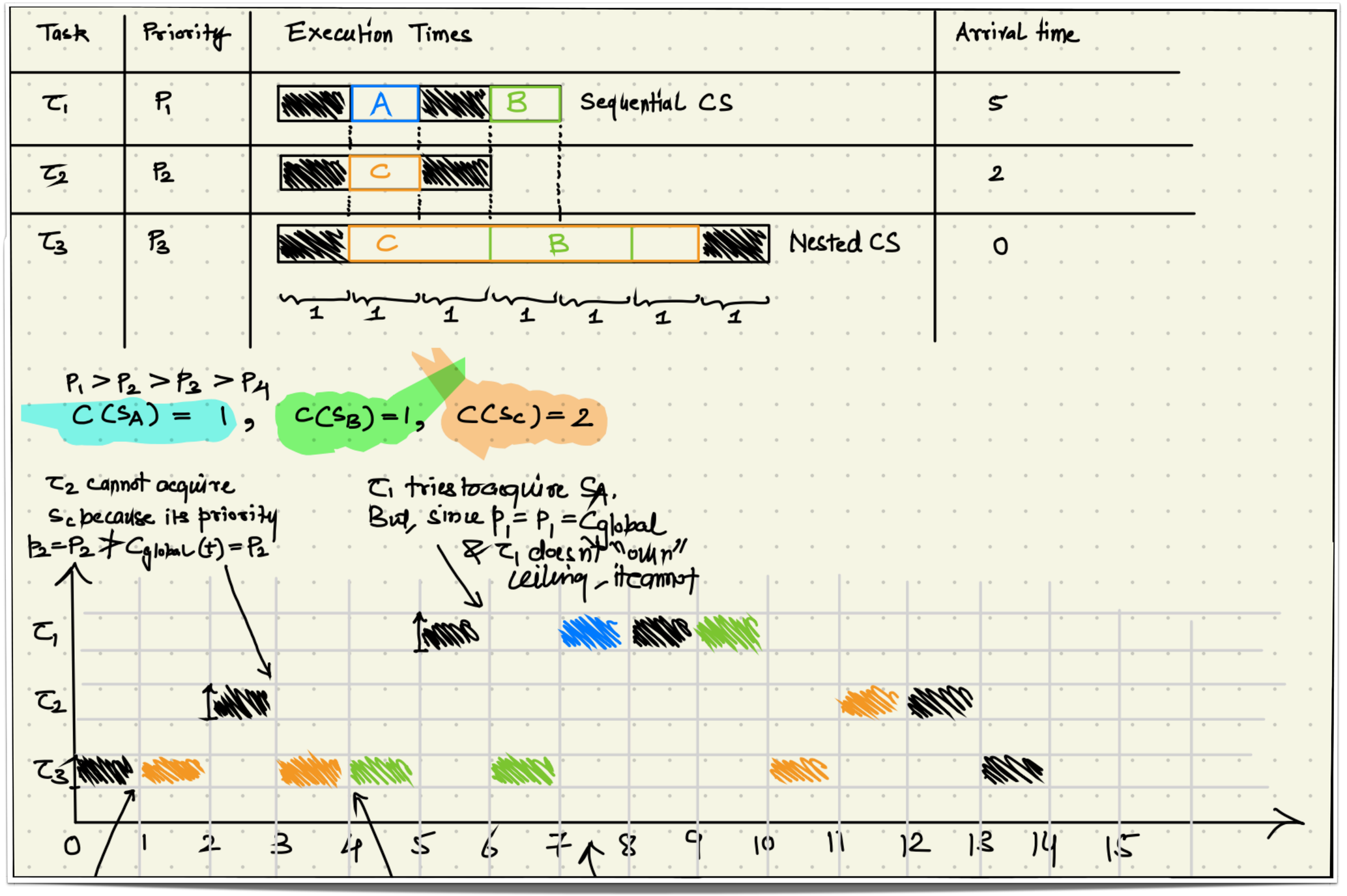


Figure 7.6 Example in which NPP causes unnecessary blocking on τ_1.

Analytically, PCP is better than PIP

- In addition, unlike PIP
  - PCP prevents transitive blocking
  - PCP prevents deadlocks
  - A task τ_i can be blocked for **at most** the duration of **one** critical section

PCP Example


Task
Priority
Execution Times
Arrival time
τ_1
P_1
A
B
Sequential CS
5
τ_2
P_2
C
2
τ_3
P_3
C
B
Nested CS
0
1 1 1 1 1 1 1
$P_1 > P_2 > P_3 > P_4$
$C(S_A) = 1$,
$C(S_B) = 1$,
$C(S_C) = 2$
τ_2 cannot acquire S_C because its priority $P_2 = P_2 \neq C_{global}(t) = P_2$
τ_1 tries to acquire S_A. But, since $P_1 = P_1 = C_{global}$ & τ_1 doesn't "own" ceiling - it cannot
τ_1
τ_2
τ_3
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

What's Wrong with Context Switches?

- Each contended critical section causes *two additional context switches*.
  ➞ Regular preemption: LO-HI-LO
  ➞ With critical section: LO-HI-LO-HI-LO
- Context switches can be a main source of overhead: OS context switching code, possibly TLB flushes, **loss of cache affinity**.
- Switching *back* to a preempted job implies the need for **separate function call stacks** for each task ➞ *memory* overhead. *Why?*

➞ Avoiding LO-HI-LO-HI-LO context switches avoids runtime overheads and enables single-stack designs.

Stack Resource Policy (SRP)

The Stack Resource Policy (SRP)

Observation: if a preempting job requires a locked resource, then a LO-HI-LO-HI-LO context switch sequence becomes **inevitable** *only if the preempting job is allowed to start executing*.

Solution: do not allow jobs to commence execution until all (possibly) required resources are available.
→ No more LO-HI-LO-HI-LO context switch sequences…

SRP Definition[Ba91]

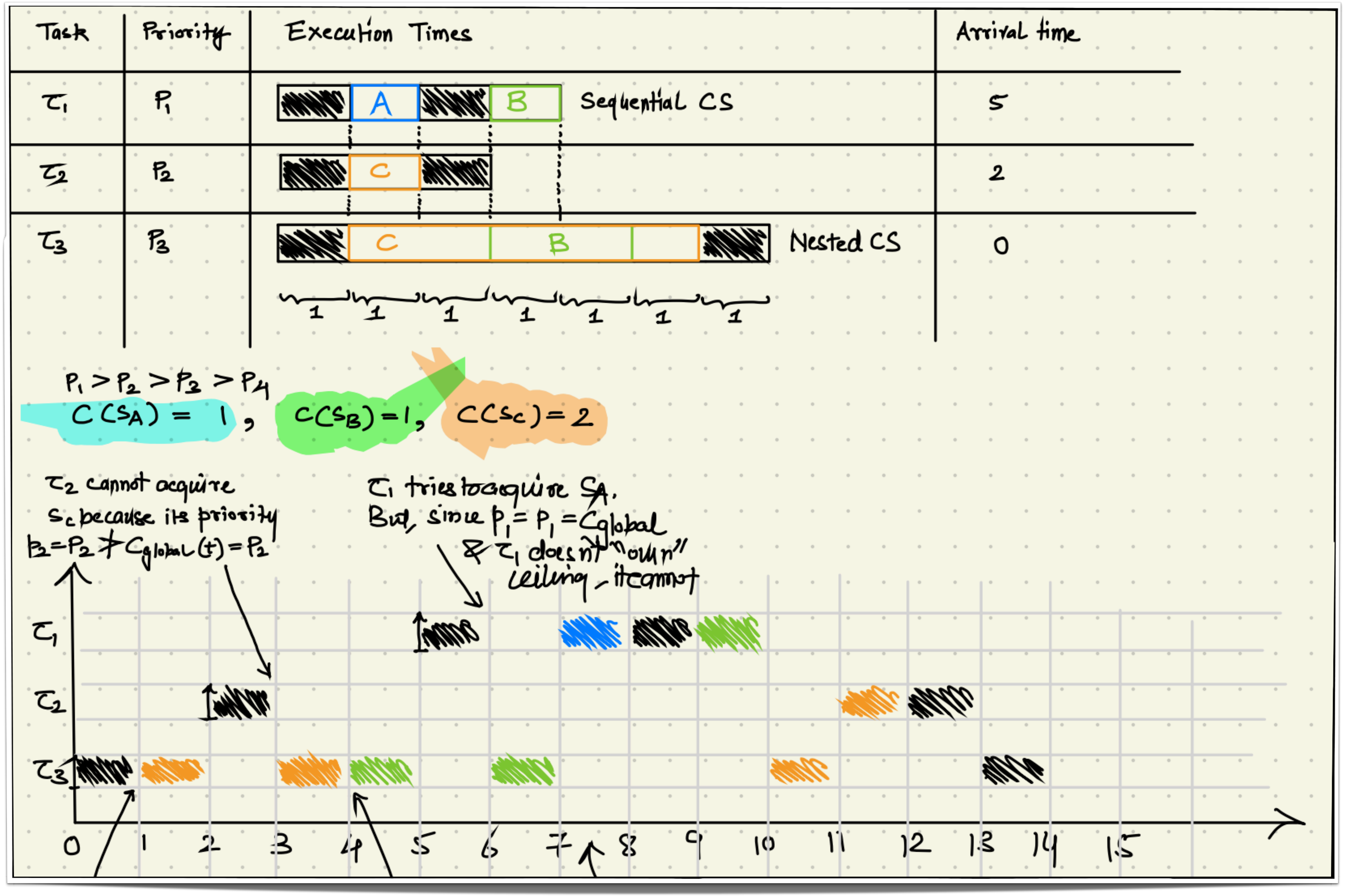
1. Define priority ceilings and system ceilings as under the PCP.
2. When a job is released, it may not commence execution until its (base) priority exceeds the system ceiling (or *preemption threshold*).
3. Whenever a job requires a resource, it gains access immediately.

[Ba91] T. Baker (1991). Stack-based scheduling for realtime processes. Real-Time Systems, 3(1):67–99.

~~PCP~~ SRP 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**~~ **immediately**
  - **Task τ_i may commence its execution 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

PCP Example



Task
Priority
Execution Times
Arrival time
τ_1
P_1
A
B
Sequential CS
5
τ_2
P_2
C
2
τ_3
P_3
C
B
Nested CS
0
1 1 1 1 1 1 1
$P_1 > P_2 > P_3 > P_4$
$C(S_A) = 1$,
$C(S_B) = 1$,
$C(S_C) = 2$
τ_2 cannot acquire S_C because its priority $P_2 = P_2 \ngtr C_{global}(t) = P_2$
τ_1 tries to acquire S_A. But, since $P_1 = P_1 = C_{global}$ & τ_1 doesn't "own" ceiling - it cannot
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

SRP Blocking Analysis

The bound on *worst-case* pi-blocking under the SRP is *identical* to the PCP's bound.

$$B_i = max\{Z_{j,k} \mid P_j < P_i \ \ and \ \ C_{global}(S_k) \geq P_i\}$$

- The *actual* pi-blocking differs under the SRP and the PCP.
  → The SRP moves blocking to an earlier point in time.
  → On average, the PCP may yield slightly less blocking. (Why?)

Sharing Runtime Stacks

Example: $prio(\tau_4) > prio(\tau_3) = prio(\tau_2) > prio(\tau_1)$

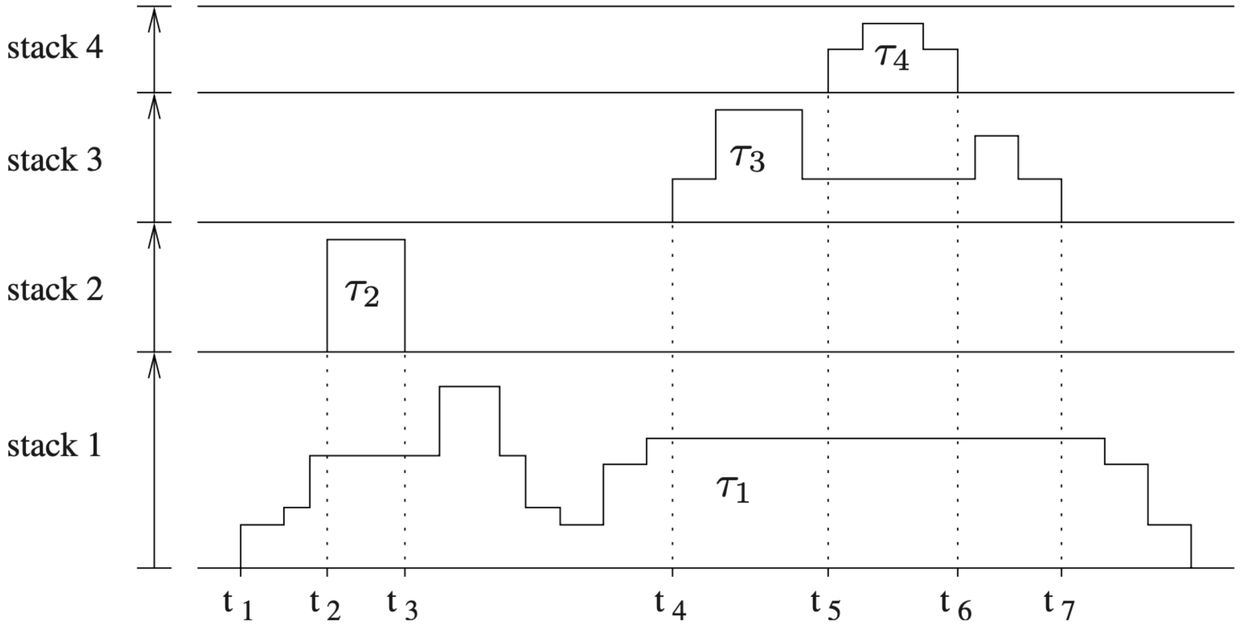


Figure 7.21 Possible evolution with one stack per task.

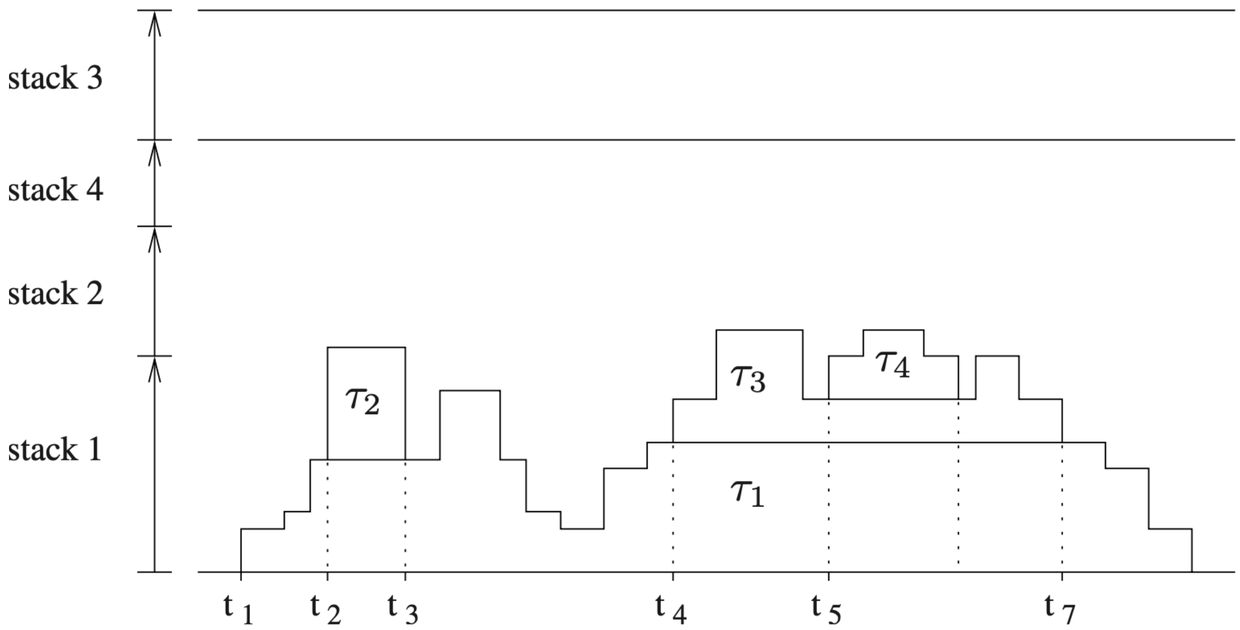


Figure 7.22 Possible evolution with a single stack for all tasks.