The Priority Ceiling Protocol

L. Sha, R. Rajkumar, J. Lehoczky, *Priority Inheritance Protocols: An Approach to Real-Time Synchronization,* IEEE Transactions on Computers, Vol. 39, No. 9, 1990

Restrictions on how we can lock (Wait, EnterMonitor) and unlock (Signal, LeaveMonitor) resources:

- a task must release all resources between invocations
- the computation time that a task i needs while holding semaphore s is bounded. $cs_{i,s}$ = the time length of the critical section for task i holding semaphore s
- a task may only lock semaphores from a fixed set of semaphores known a priory. $uses(i)$ $=$ the set of semaphores that may be used by task i

The protocol:

- the *ceiling* of a semaphore, $ceil(s)$, is the priority of the highest priority task that uses the semaphore
- notation: $pri(i)$ is the priority of task i
- At run-time:
  - if a task i wants to lock a semaphore s, it can only do so if $pri(i)$ is **strictly higher** than the ceilings of all semaphores currently locked by **other** tasks
  - if not, task i will be blocked (task i is said to be blocked on the semaphore, S^*, with the highest priority ceiling of all semaphores currently locked by other jobs and task i is said to be blocked by the task that holds S^*)
  - when task i is blocked on S^*, the task currently holding S^* inherits the priority of task i

Properties:

- deadlock free
- a given task i is delayed at most once by a lower priority task
- the delay is a function of the time taken to execute the critical section

Deadlock free

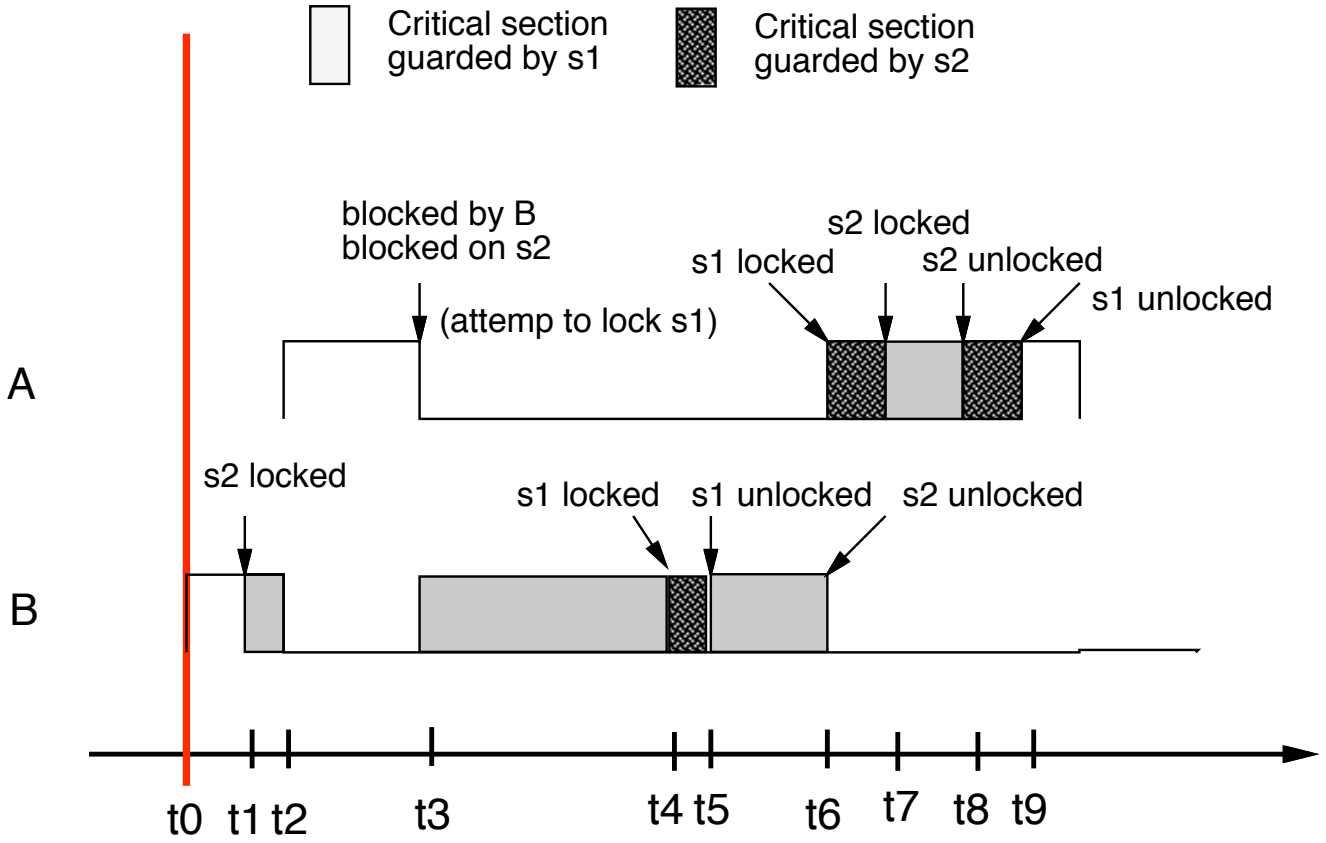
Example:

Task name	T	Priority
A	50	10
B	500	9

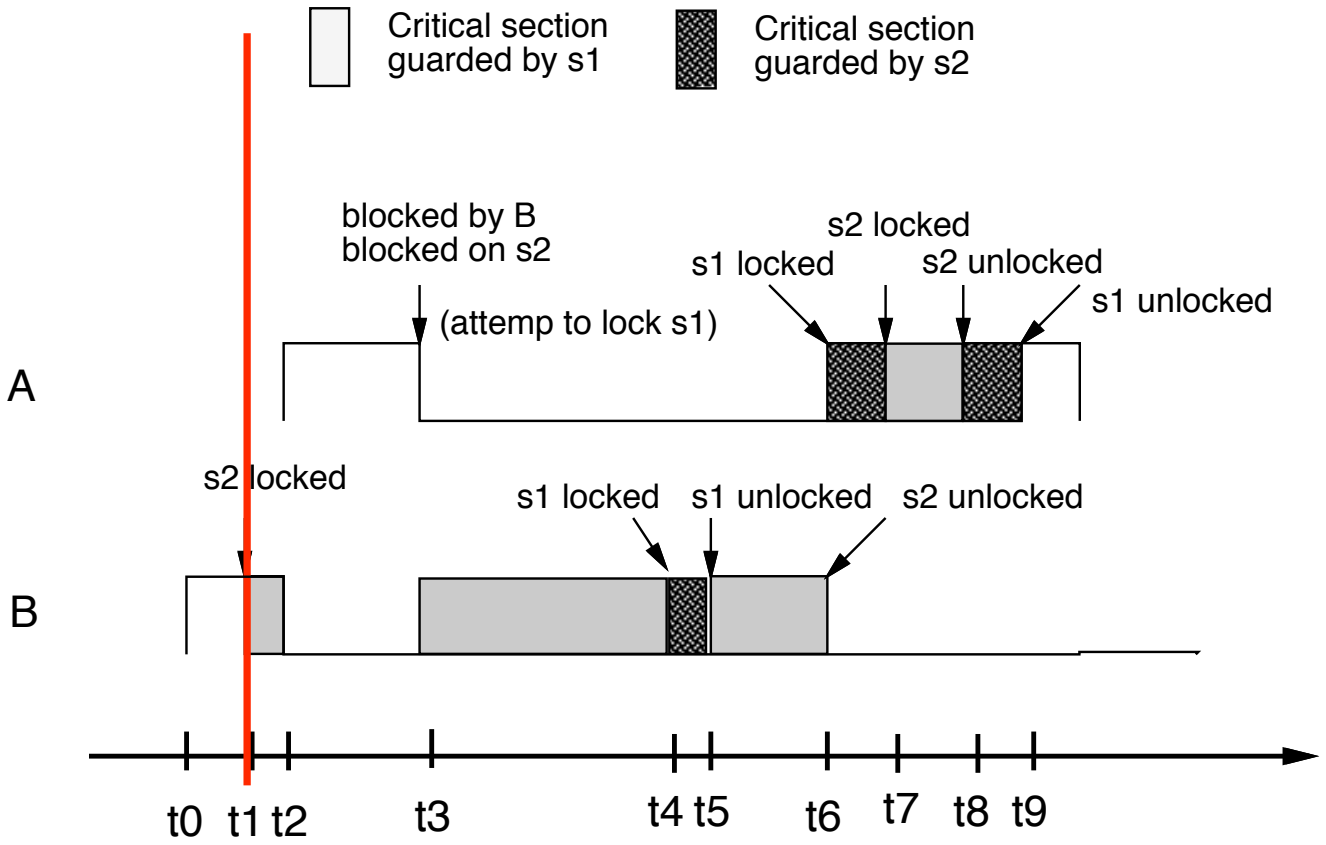
```
Task A        Task B

lock(s1)      lock(s2)
lock(s2)      lock(s1)
...           ...
unlock(s1)    unlock(s1)
unlock(s2)    unlock(s1)
```

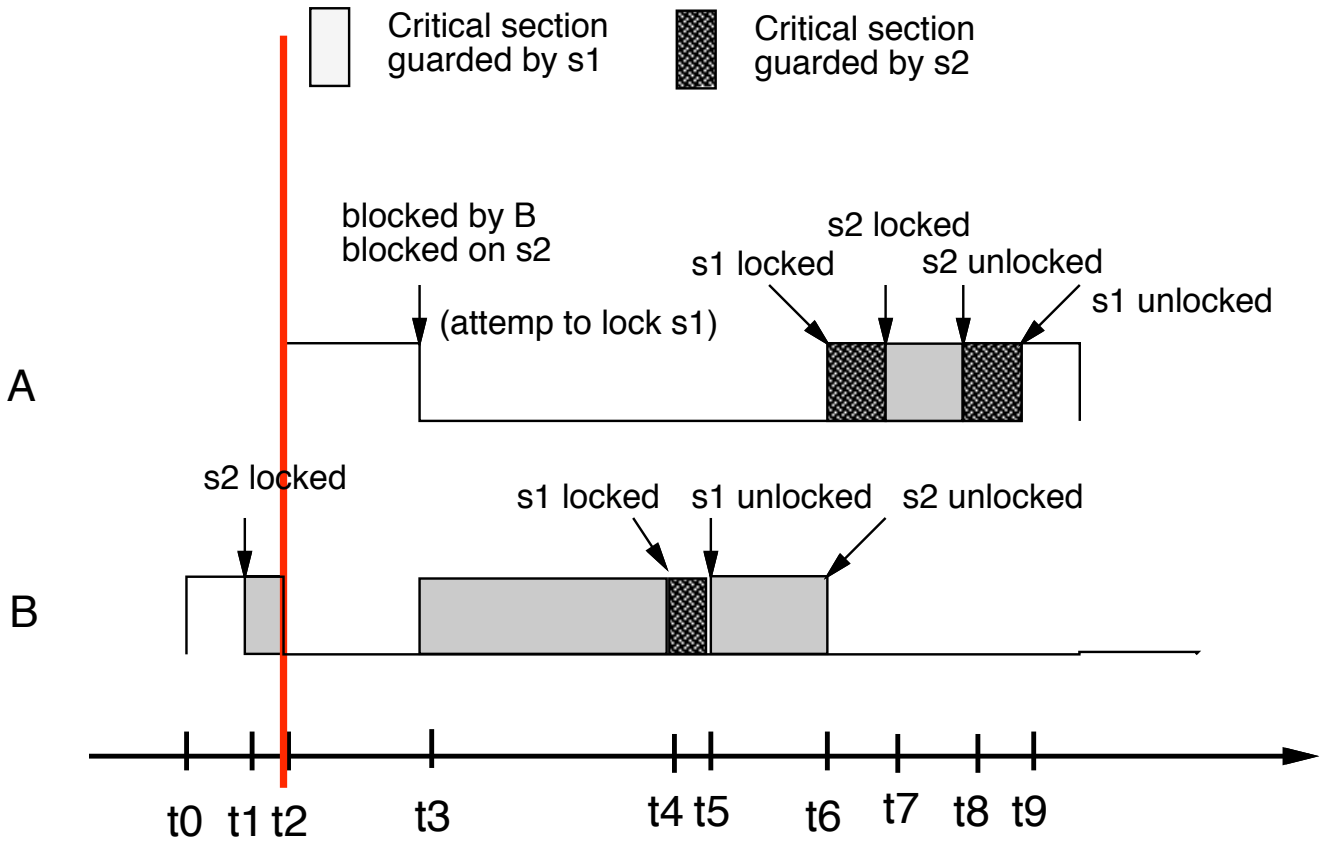
$$ceil(s_1) = 10, ceil(s_2) = 10$$



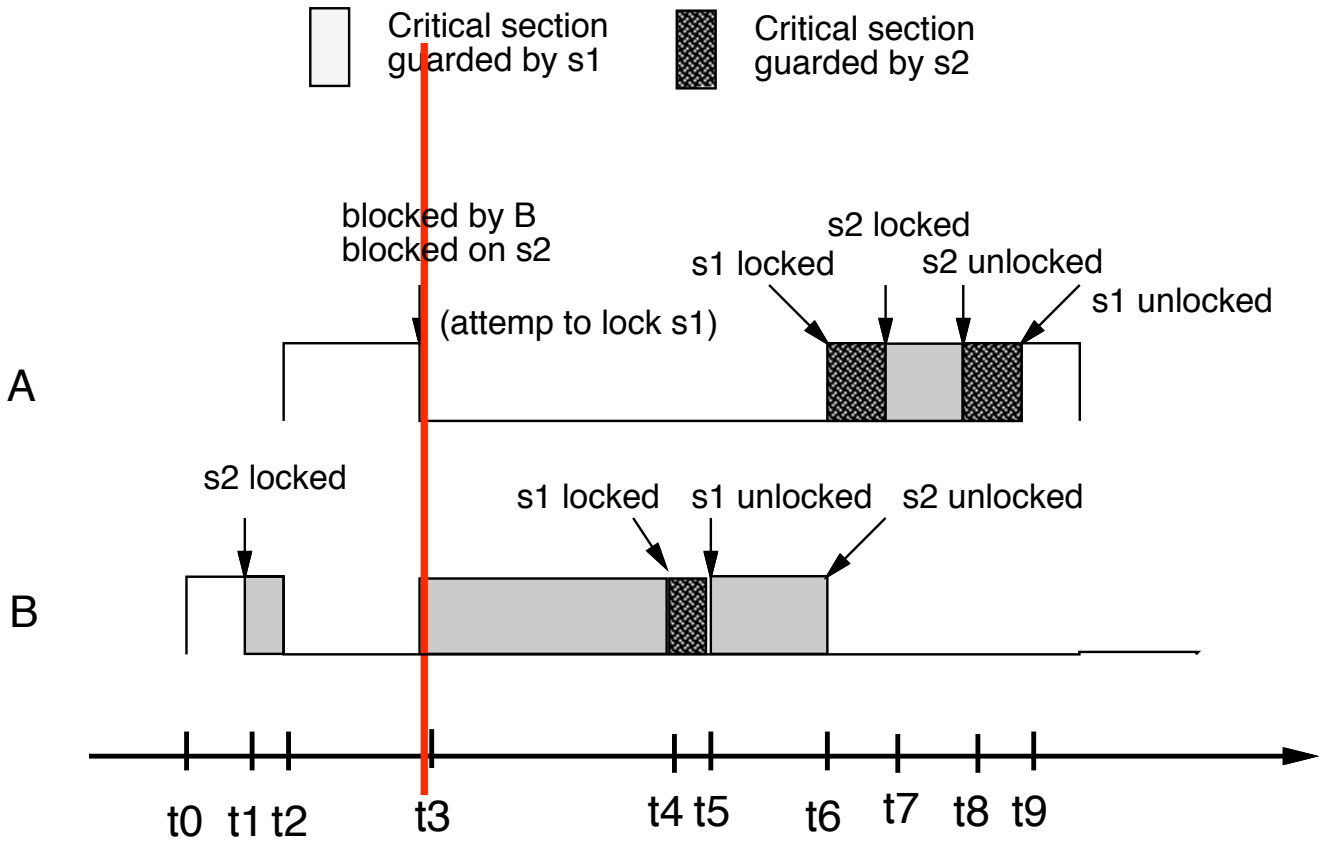
- t_0: B starts executing



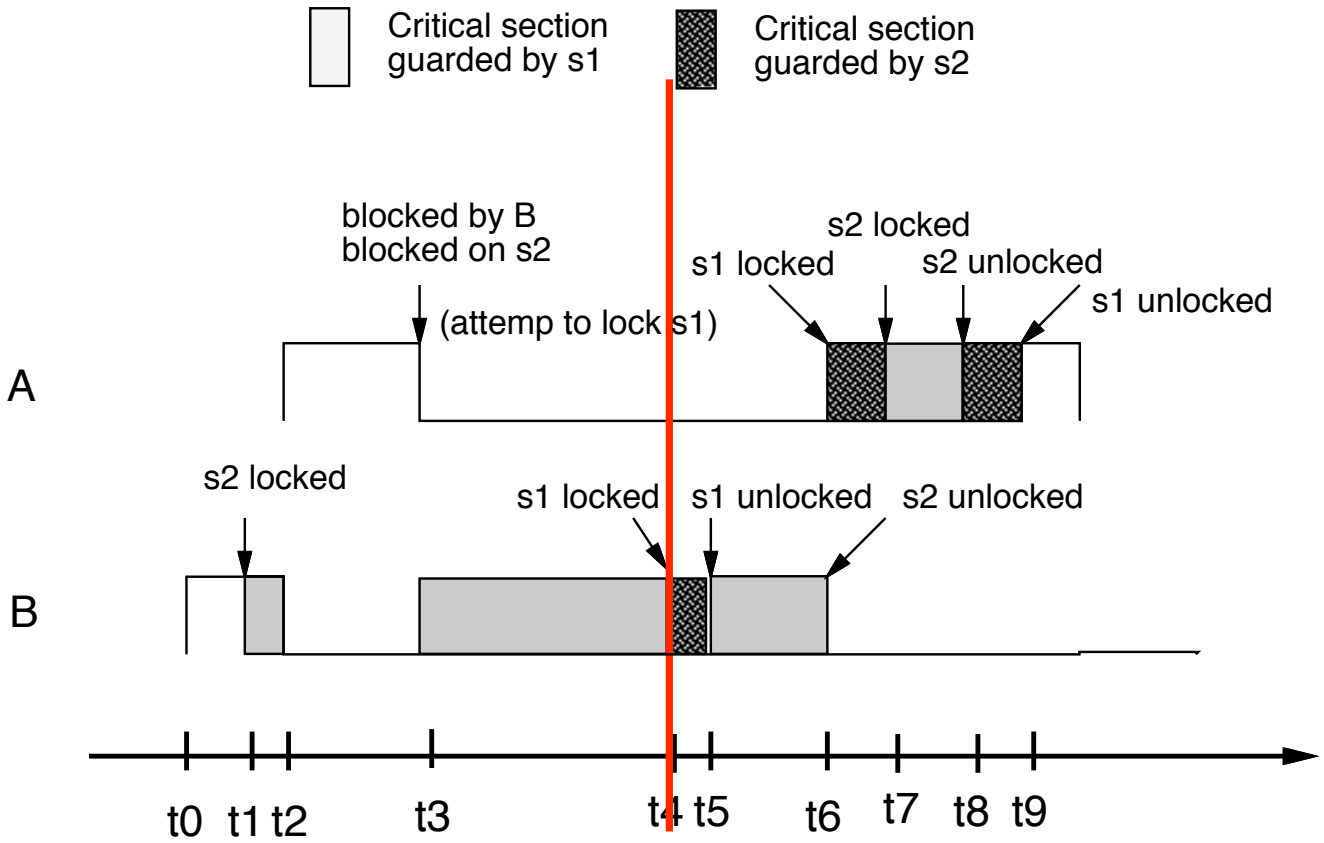
- t_1: B attempts to lock s_2. It succeeds since no lock is held by another task.



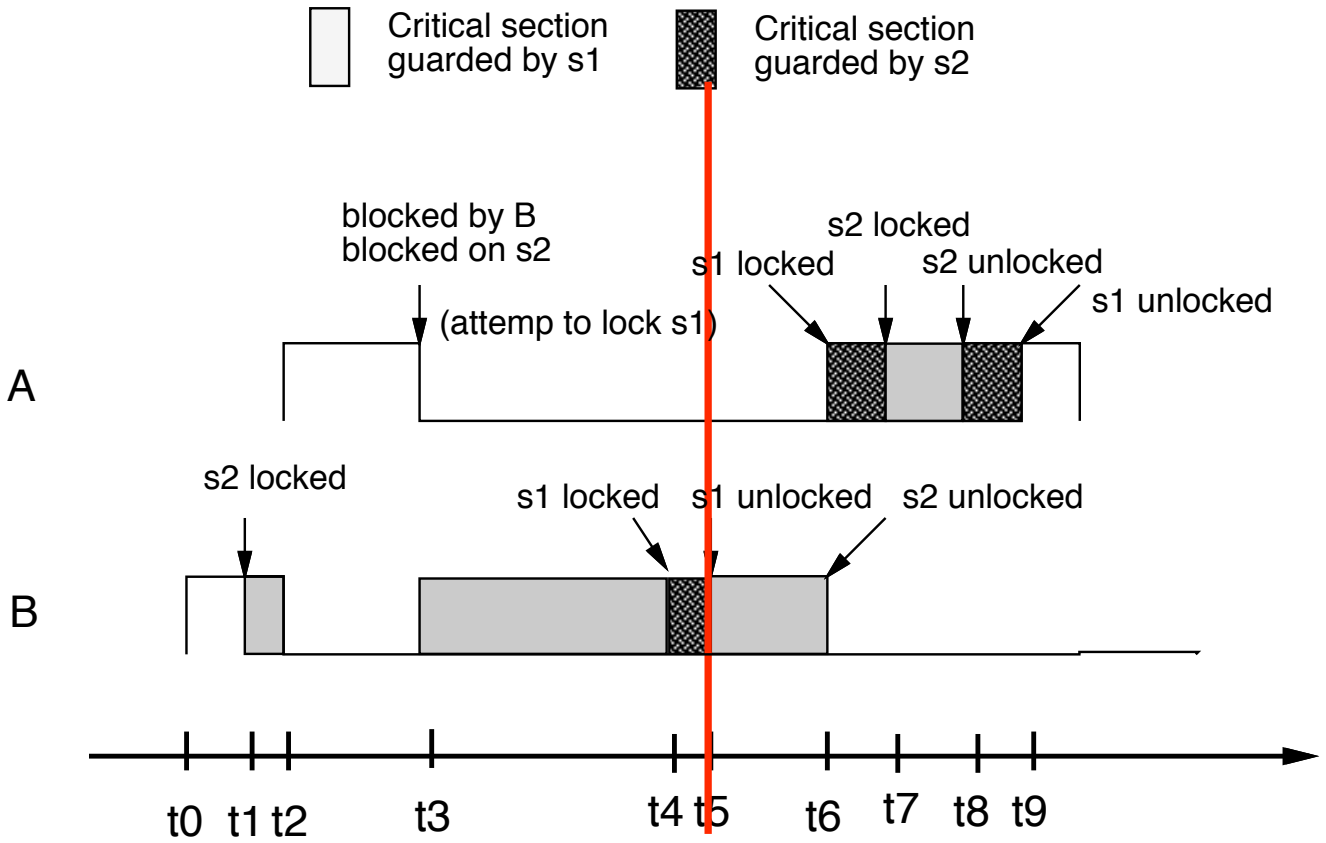
- t_2: A preempts B



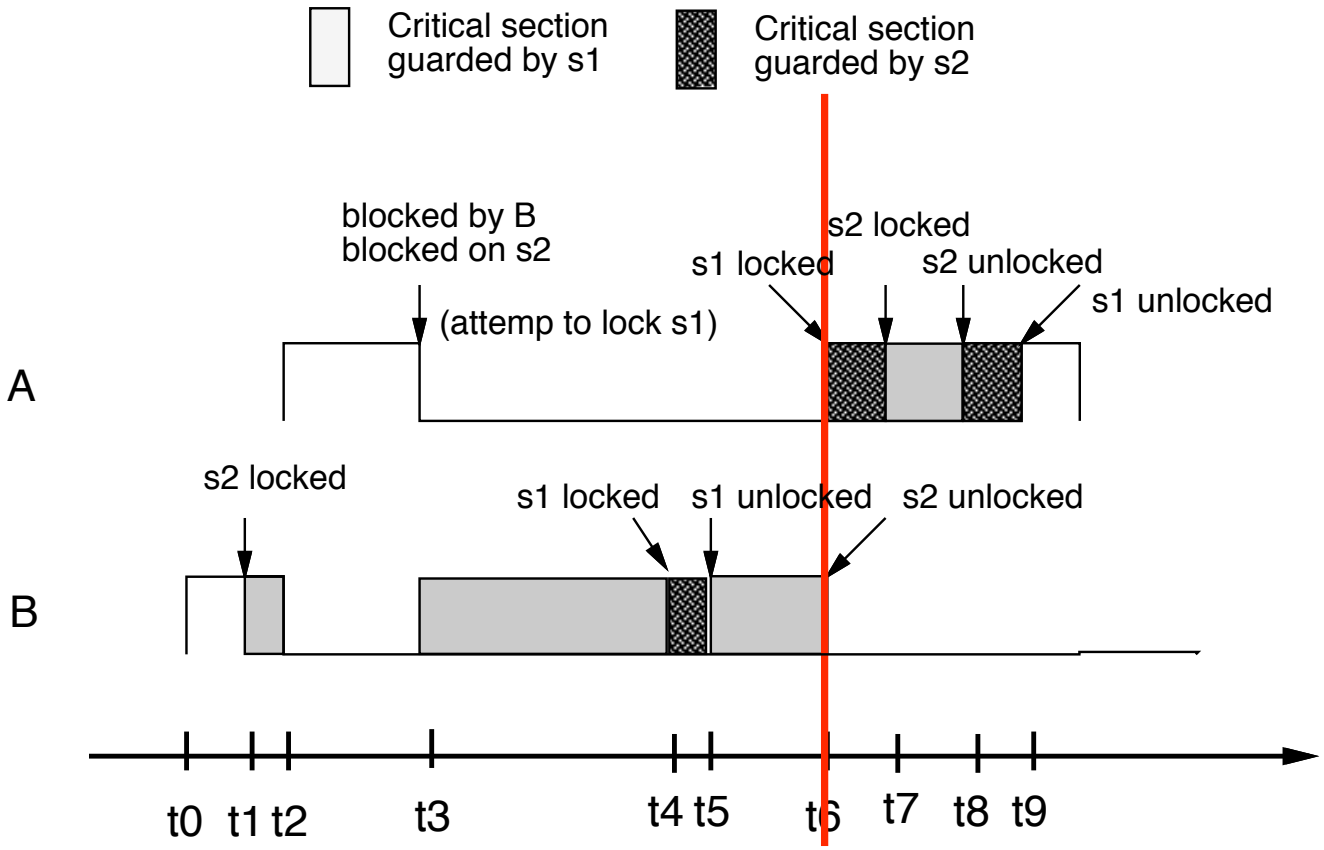
- t_3: A tries to lock s_1. A fails since A's priority (10) is not strictly higher than the ceiling of s_2 (10) that is held by B
- A is blocked by B
- A is blocked on s_2
- The priority of B is raised to 10.



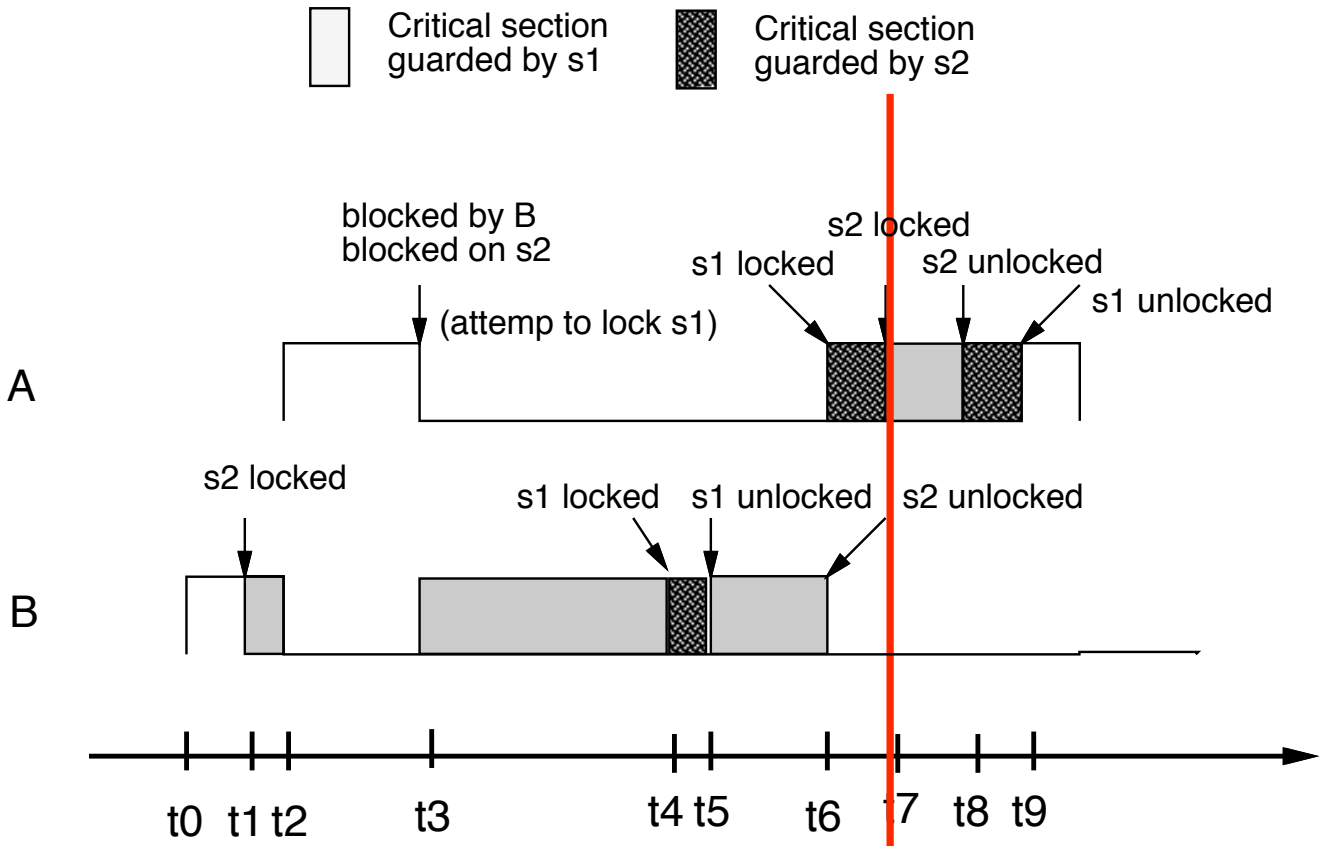
- t_4: B attempts to lock s_1. B succeeds since there are no locks held by any other tasks.



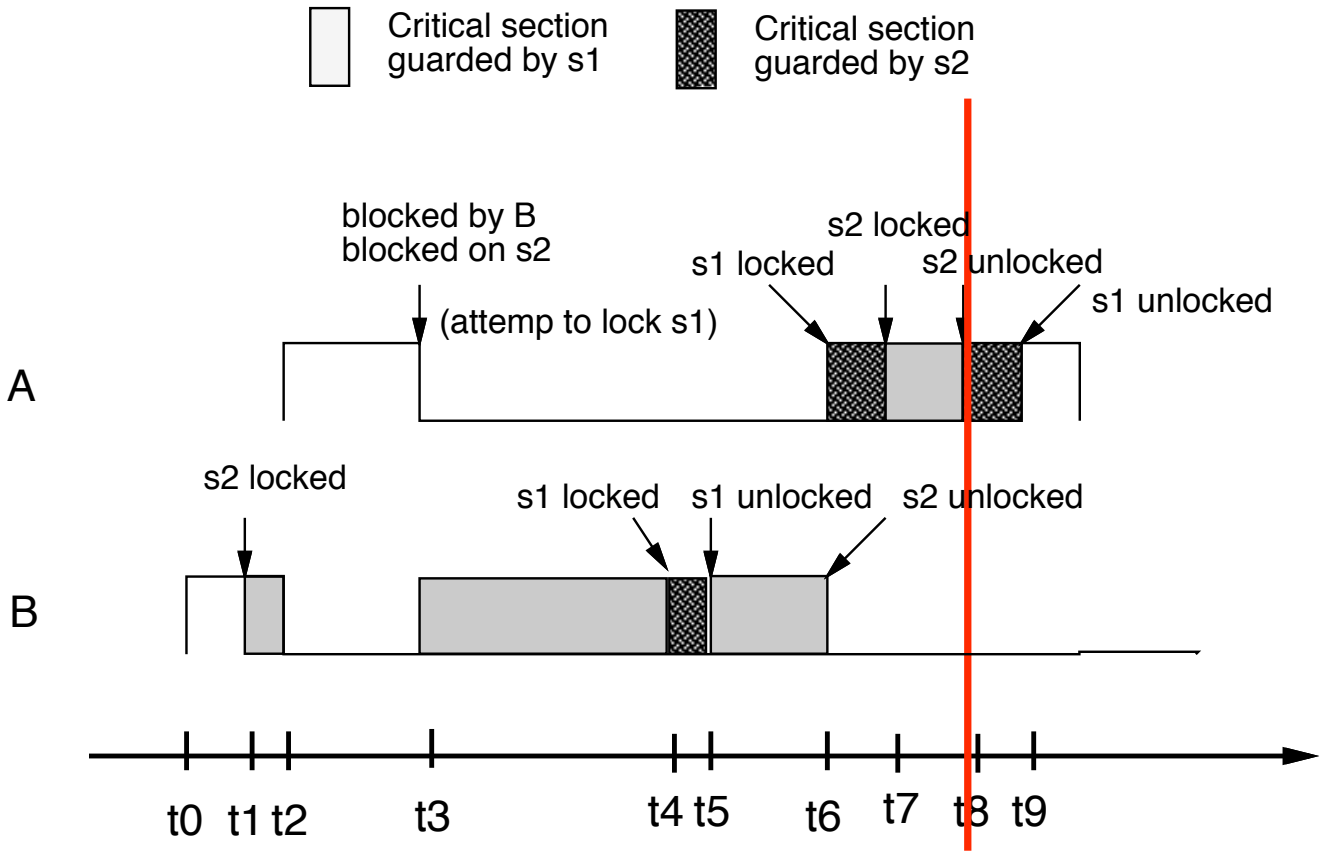
- t_5: B unlocks s_1



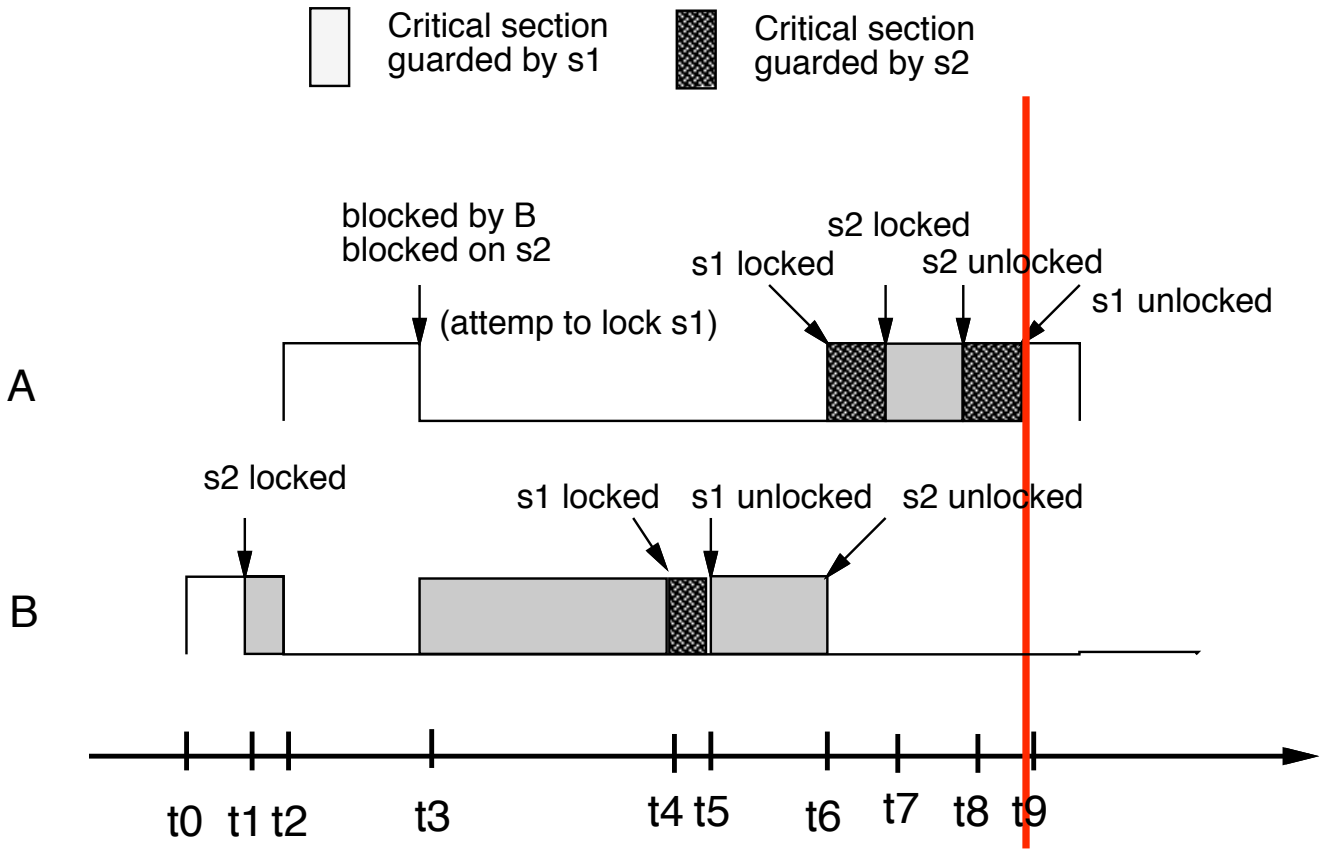
- t_6: B unlocks s_2
- The priority of B is lowered to its assigned priority (9)
- A preempts B, attempts to lock s_1 and succeeds



- t_7: A attempts to lock s_2. Succeeds



- t_8: A unlocks s_2



- t_9: A unlocks s_1

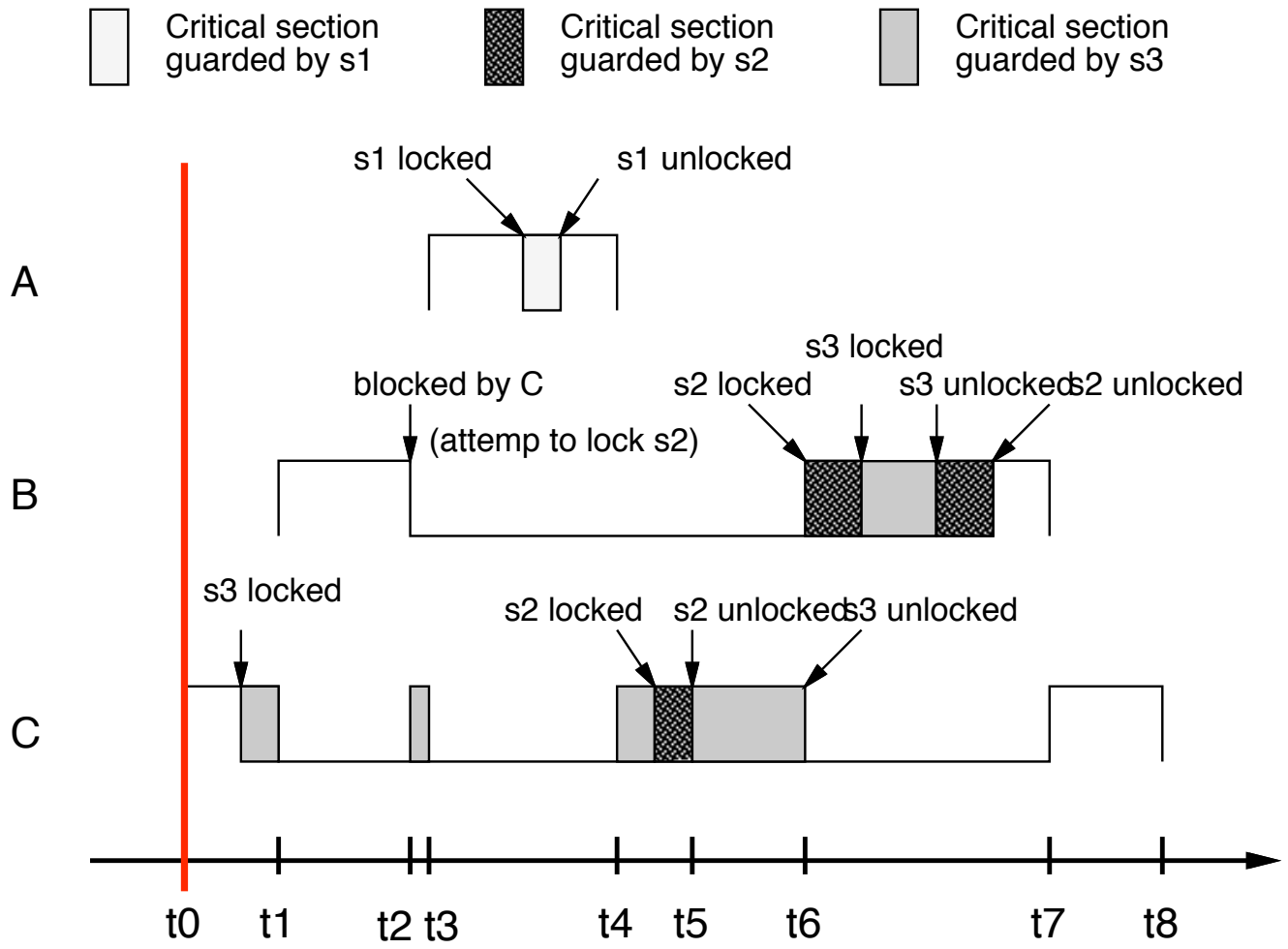
Example:

Task name	T	Priority
A	50	10
B	500	9
C	3000	8

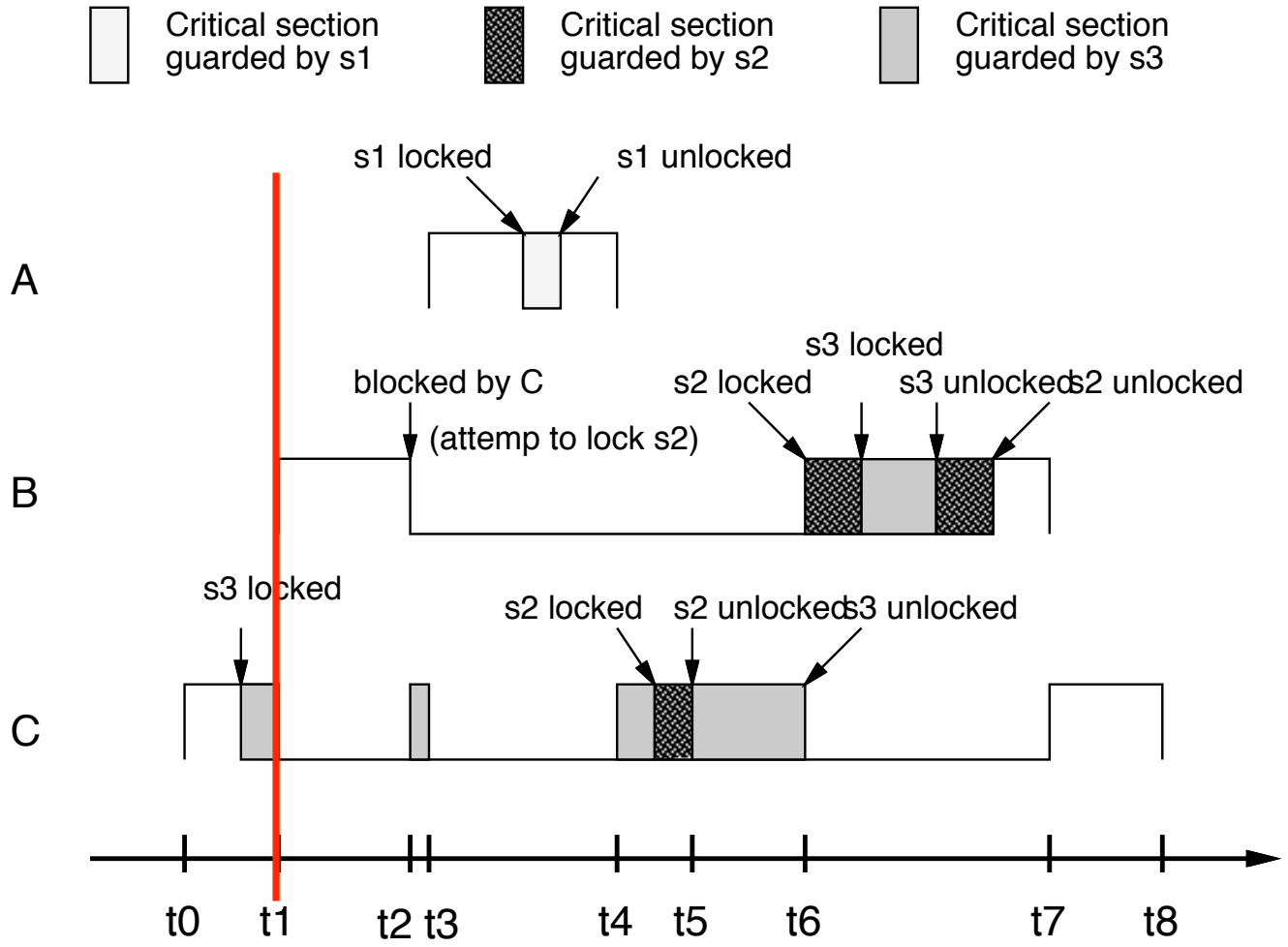
```
Task A       Task B       Task C

lock(s1)     lock(s2)     lock(s3)
..           ..           ..
unlock(s1)   lock(s3)     lock(s2)
..           ..           ..
             unlock(s3)   unlock(s2)
             ..           ..
             unlock(s2)   unlock(s3)
```

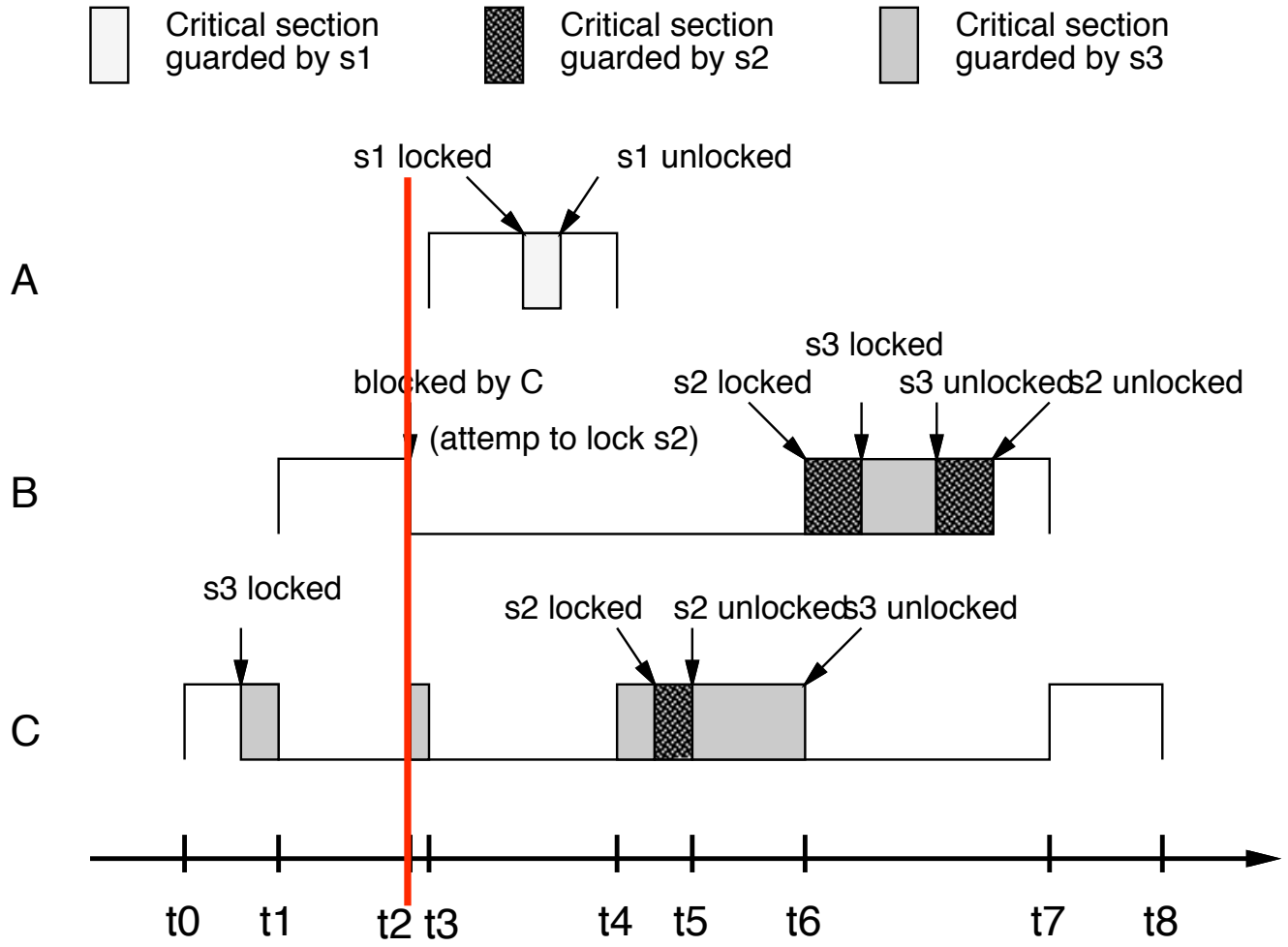
$$ceil(s_1) = 10, ceil(s_2) = ceil(s_3) = 9$$



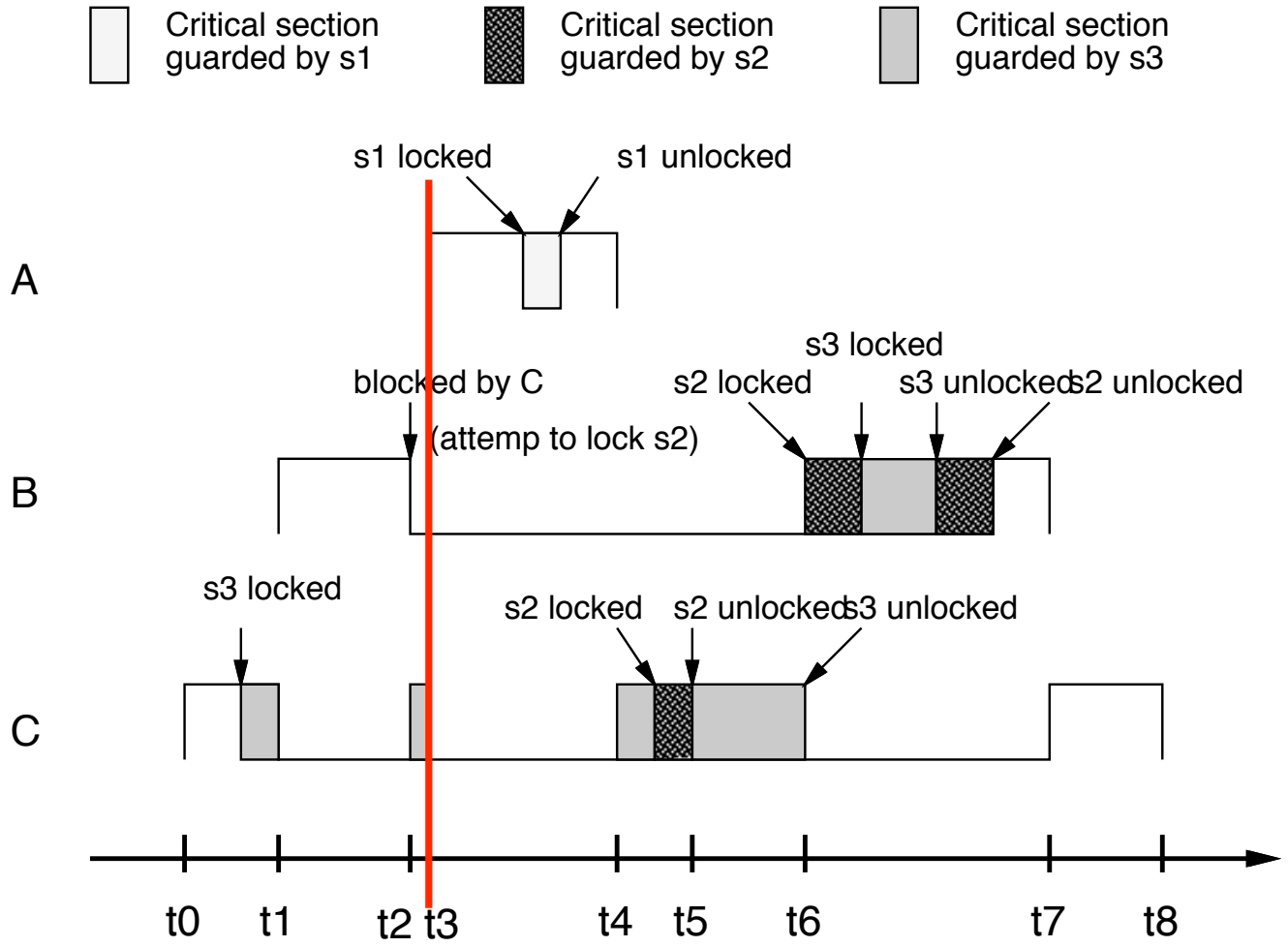
- t_0: C starts execution and then locks s_3



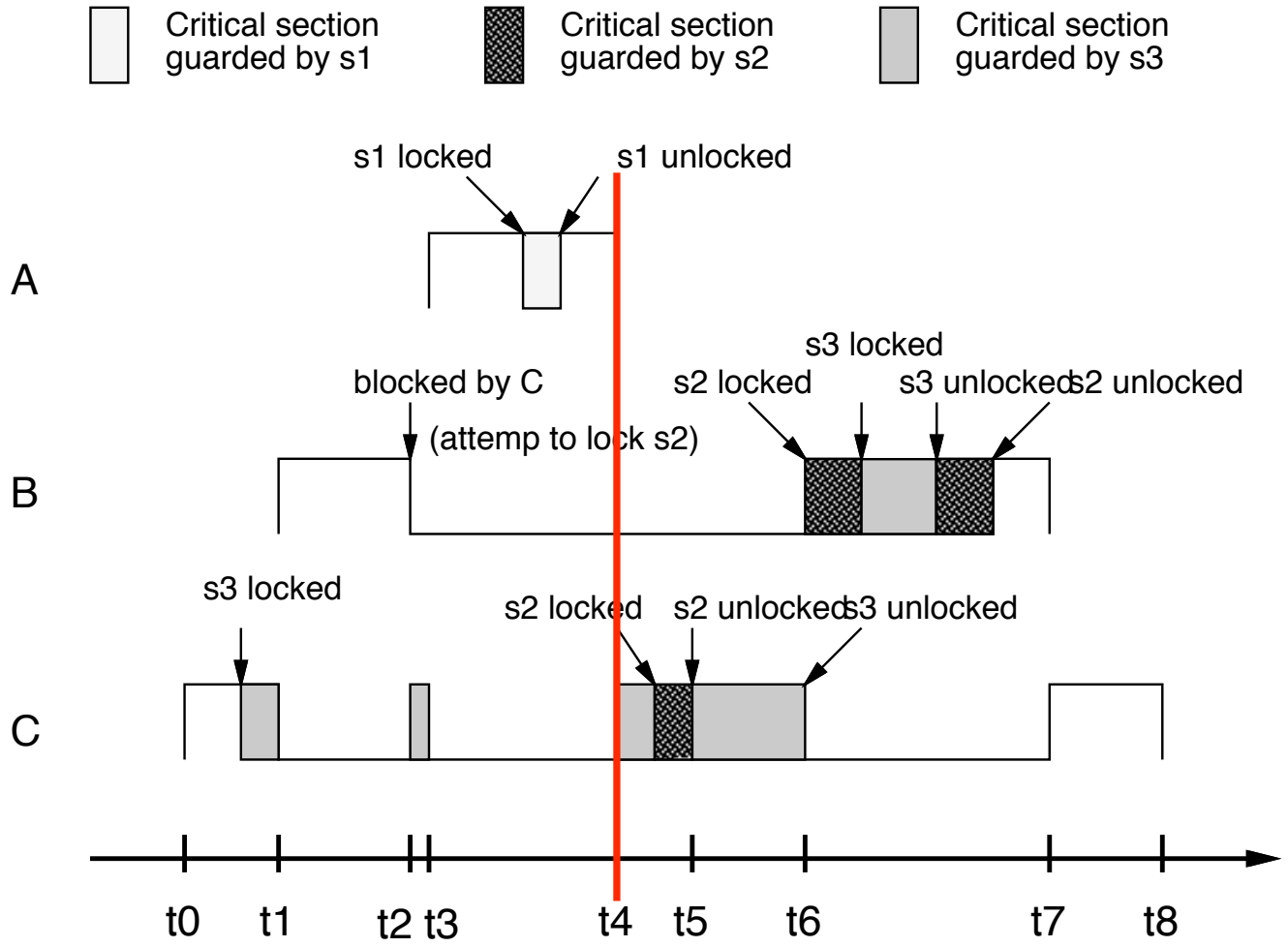
- t_1: B preempts C



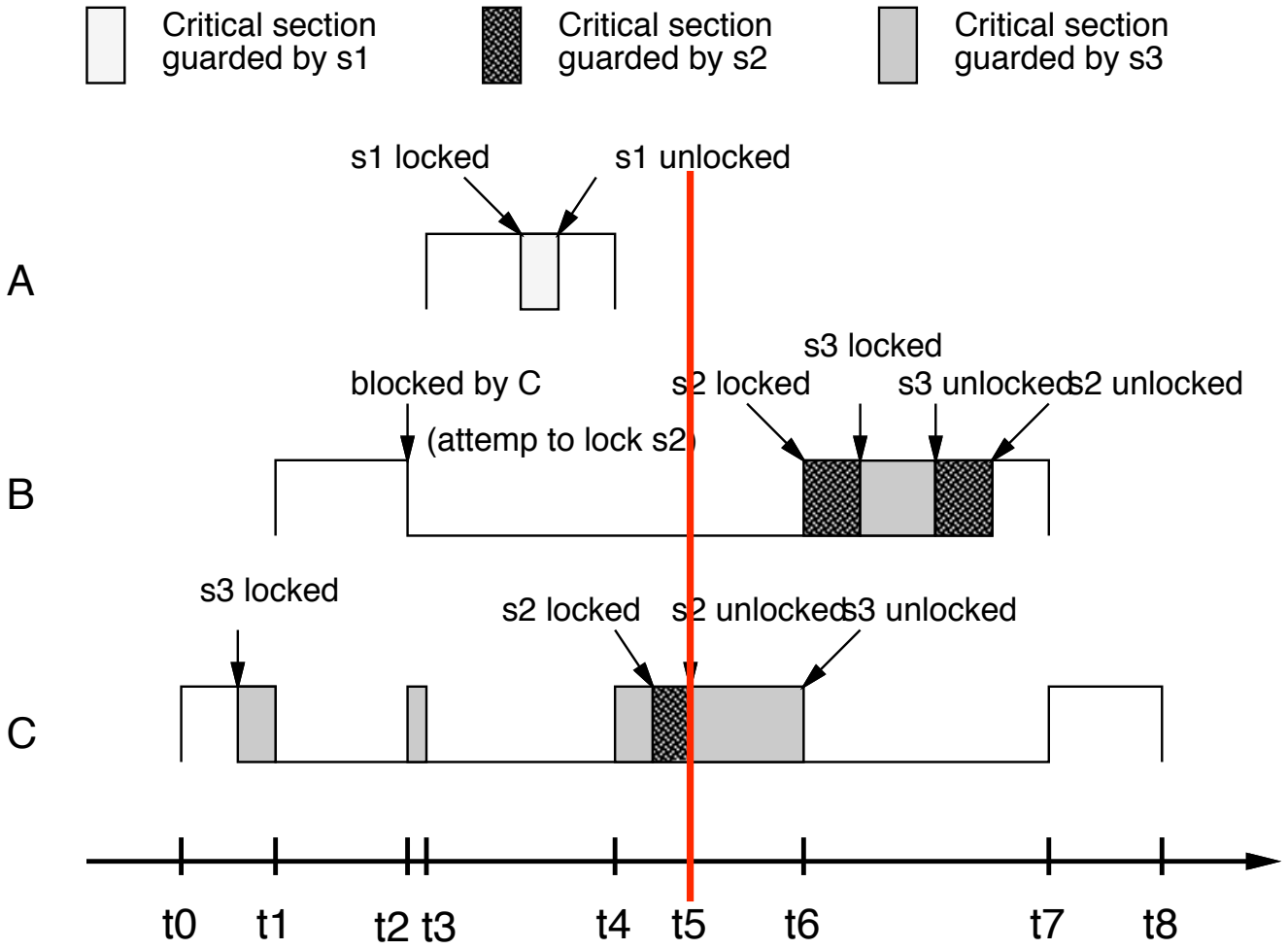
- t_2: B tries to lock s_2. B fails (the priority of B is not strictly higher than the ceiling of s_3 that is held by C) and blocks on s_3 (B is blocked by C). C inherits the priority of B.



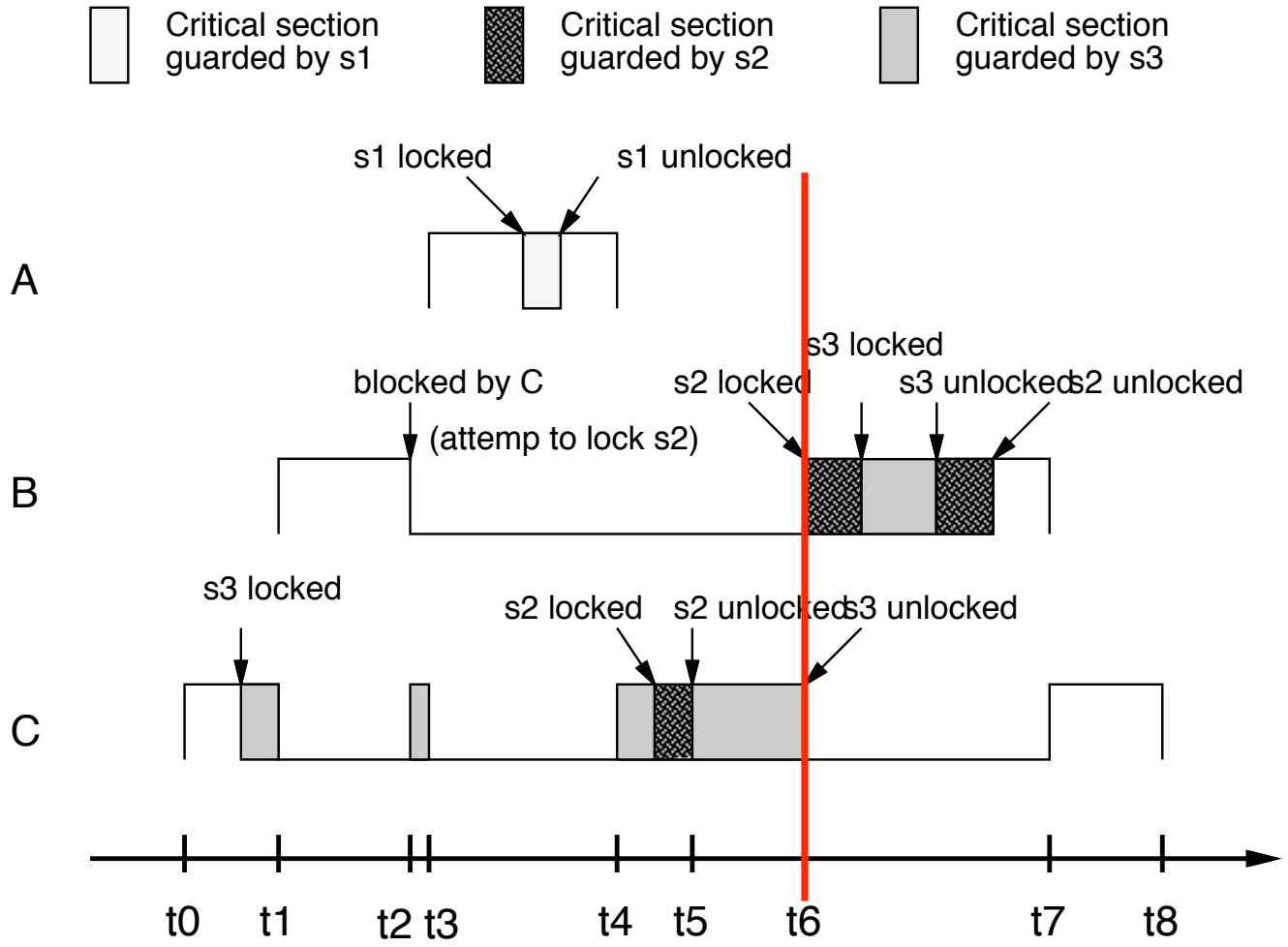
- t_3: A preempts C. Later is tries to lock s_1 and succeeds (the priority of A is higher than the ceiling of s_3).



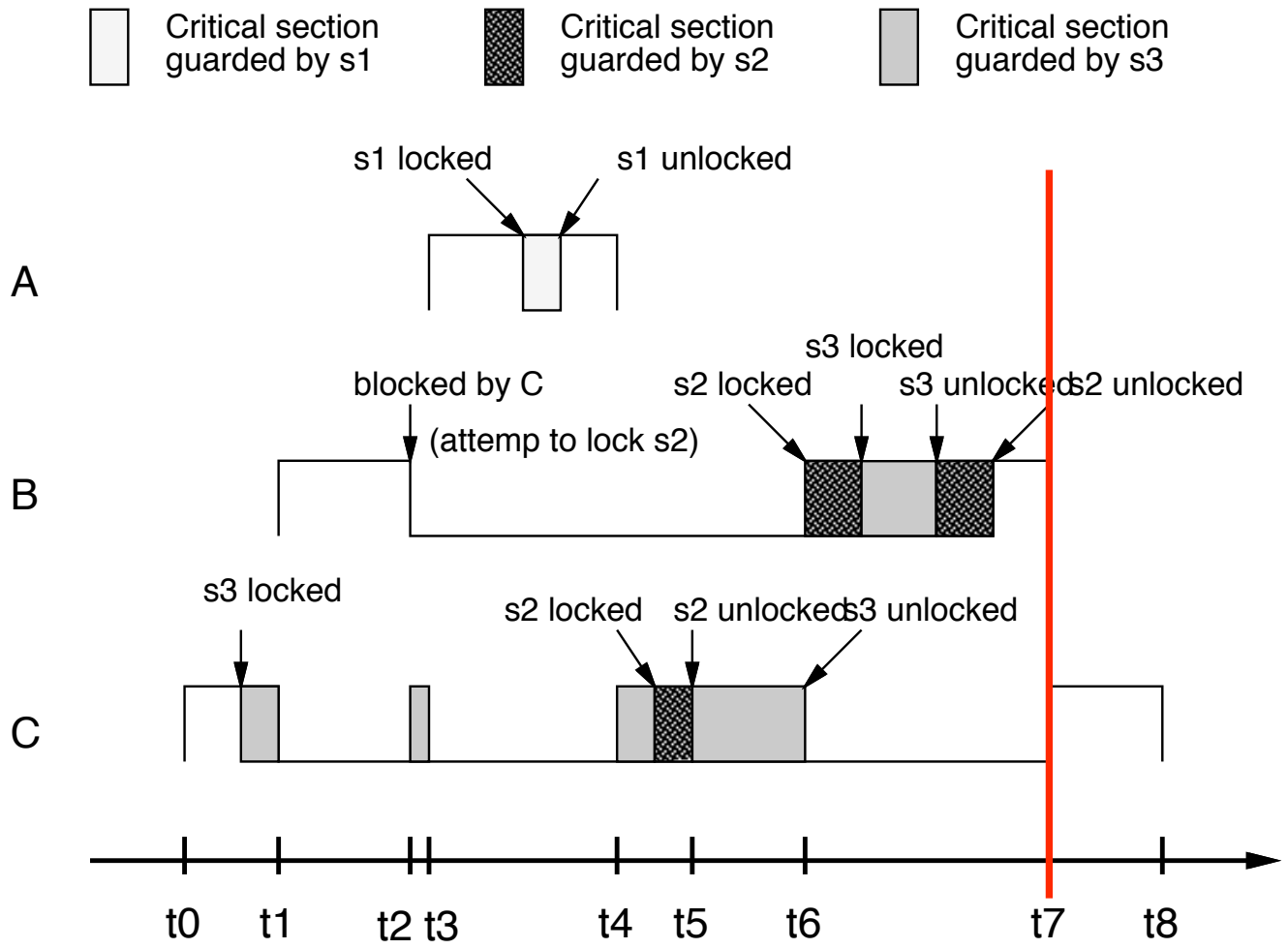
- t_4: A completes. C resumes and later tries to lock s_2 and succeeds (it is C itself that holds s_3).



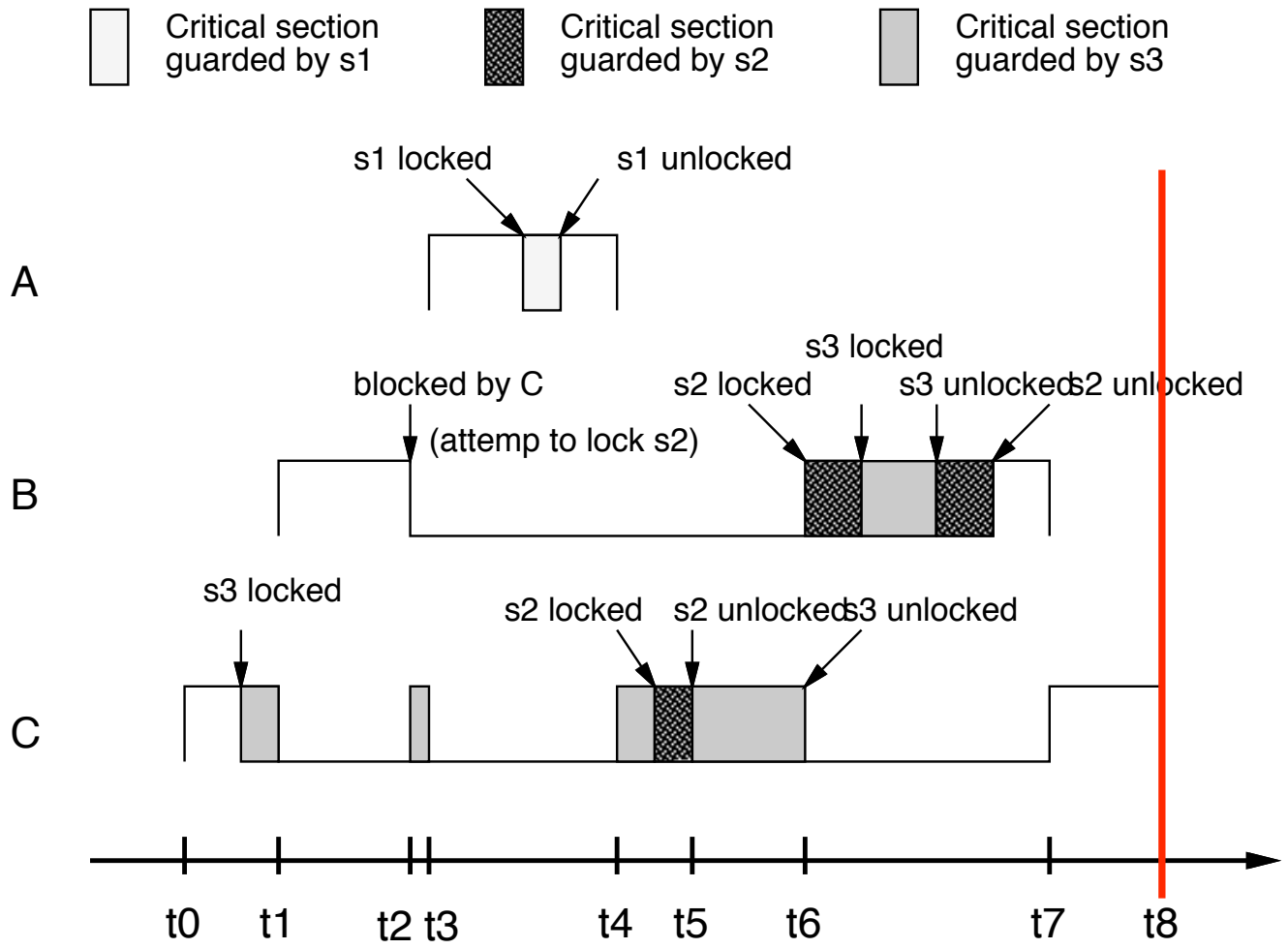
- t_5: C unlocks s_2



- t_6: C unlocks s_3, and gets back its basic priority. B preempts C, tries to lock s_2 and succeeds. Then B locks s_3, unlocks s_3 and unlocks s_2



- t_7: B completes and C is resumed.



- t_8: C completes

- A is never blocked
- B is blocked by C during the intervals $[t_2, t_3]$ and $[t_4, t_6]$. However, B is blocked for no more than the duration of one time critical section of the lower priority task C even though the actual blocking occurs over disjoint time intervals

General properties:

- with ordinary priority inheritance, a task i can be blocked for at most the duration of $\min(n, m)$ critical sections, where n is the number of lower priority tasks that could block i and m is the number of semaphores that can be used to block i
- with the priority ceiling inheritance, a task i can be blocked for at most the duration of one longest critical section
- sometimes priority ceiling introduces unnecessary blocking but the worst-case blocking delay is much less than for ordinary priority inheritance

The Immediate Inheritance Protocol

- when a task obtains a lock the priority of the task is immediately raised to the ceiling of the lock
- the same worst-case timing behavior as the priority ceiling protocol (also known as the Priority Ceiling Emulation Protocol and as the Priority Protect Protocol)
- easy to implement
- on a single-processor system it is not necessary to have any queues of blocked tasks for the locks (semaphores, monitors) – tasks waiting to acquire the locks will have lower priority than the task holding the lock and can, therefore be queued in `ReadyQueue`.

Priority Inheritance

Priority inheritance is a common, but not mandatory, feature of most Java implementations.

The Real-Time Java Specification requires that the priority inheritance protocol is implemented by default. The priority ceiling protocol is optional.