Lecture 10

PRIORITY INHERITANCE PROTOCOL

The *Priority Inheritance Protocol* (PIP), proposed by Sha, Rajkumar and Lehoczky [SRL90], avoids unbounded priority inversion by modifying the priority of those tasks that cause blocking. In particular, when a task τ_i blocks one or more higher-priority tasks, it temporarily assumes (*inherits*) the highest priority of the blocked tasks. This prevents medium-priority tasks from preempting τ_i and prolonging the blocking duration experienced by the higher-priority tasks.

The Priority Inheritance Protocol can be defined as follows:

1. **Tasks are scheduled based on their active priorities. Tasks with the same priority are executed in a First Come First Served discipline.**
2. **When task τ_i tries to enter a critical section $z_{i,k}$ and resource R_k is already held by a lower-priority task τ_j , then τ_i is blocked. τ_i is said to be blocked by the task τ_j that holds the resource. Otherwise, τ_i enters the critical section $z_{i,k}$.**
3. **When a task τ_i is blocked on a semaphore, it transmits its active priority to the task, say τ_j , that holds that semaphore. Hence, τ_j resumes and executes the rest of its critical section with a priority $p_j = p_i$. Task τ_j is said to *inherit* the priority of τ_i. In general, a task inherits the highest priority of the tasks it blocks. That is, at every instant,**

   $$p_j(R_k) = \max\{P_j, \max_h \{P_h | \tau_h \text{ is blocked on } R_k\}\}.$$

4. **When τ_j exits a critical section, it unlocks the semaphore, and the highest-priority task blocked on that semaphore, if any, is awakened. Moreover, the active priority of τ_j is updated as follows: if no other tasks are blocked by τ_j , p_j is set to its nominal priority P_j ; otherwise it is set to the highest priority of the tasks blocked by τ_j , according to Equation (7.8).**
5. **Priority inheritance is transitive; that is, if a task τ_3 blocks a task τ_2, and τ_2 blocks a task τ_1, then τ_3 inherits the priority of τ_1 via τ_2.**

Example:

There are five jobs and two resources *Black* and *Shaded*. The parameters of the jobs and their critical sections are listed in part (a). As usual, jobs are indexed in decreasing order of their priorities: The priority π_i of J_i is i , and

the smaller the integer, the higher the priority. In the schedule in part (b) of this figure, black boxes show the critical sections when the jobs are holding *Black*. Shaded boxes show the critical sections when the jobs are holding *Shaded*.

1. At time 0, job *J*5 becomes ready and executes at its assigned priority 5. At time 1, it is granted the resource *Black*.

2. At time 2, *J*4 is released. It preempts *J*5 and starts to execute.

3. At time 3, *J*4 requests *Shaded*. *Shaded*, being free, is granted to the job. The job continues to execute.

4. At time 4, *J*3 is released and preempts *J*4. At time 5, *J*2 is released and preempts *J*3.

5. At time 6, *J*2 executes *L(Black)* to request *Black*; *L(Black)* fails because *Black* is in use by *J*5. *J*2 is now directly blocked by *J*5. According to rule 3, *J*5 inherits the priority 2 of *J*2. Because *J*5's priority is now the highest among all ready jobs, *J*5 starts to execute.

6. *J*1 is released at time 7. Having the highest priority 1, it preempts *J*5 and starts to execute.

7. At time 8, *J*1 executes *L(Shaded)*, which fails, and becomes blocked. Since *J*4 has *Shaded* at the time, it directly blocks *J*1 and, consequently, inherits *J*1's priority 1. *J*4 now has the highest priority among the ready jobs *J*3, *J*4, and *J*5. Therefore, it starts to execute.

8. At time 9, *J*4 requests the resource *Black* and becomes directly blocked by *J*5. At this time the current priority of *J*4 is 1, the priority it has inherited from *J*1 since time 8.

Therefore, *J*5 inherits priority 1 and begins to execute.

9. At time 11, *J*5 releases the resource *Black*. Its priority returns to 5, which was its priority when it acquired *Black*. The job with the highest priority among all unblocked jobs is *J*4. Consequently, *J*4 enters its inner critical section and proceeds to complete this and the outer critical section.

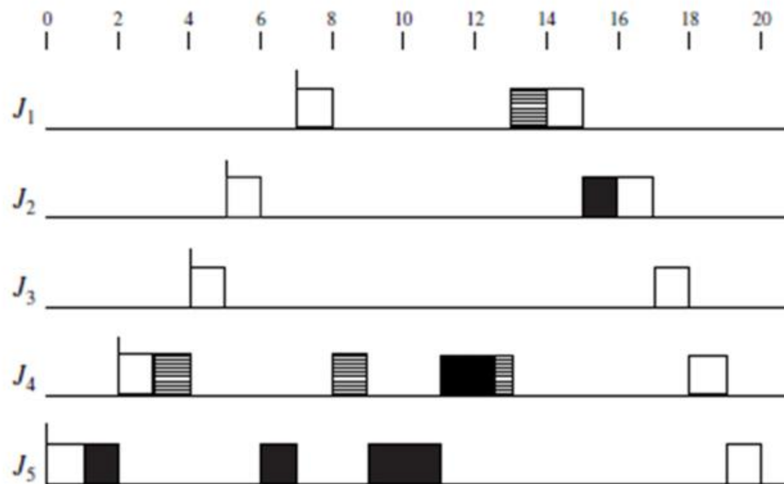
10. At time 13, *J*4 releases *Shaded*. The job no longer holds any resource; its priority returns to 4, its assigned priority. *J*1 becomes unblocked, acquires *Shaded*, and begins to execute.

11. At time 15, *J*1 completes. *J*2 is granted the resource *Black* and is now the job with the highest priority. Consequently, it begins to execute.

12. At time 17, *J*2 completes. Afterwards, jobs *J*3, *J*4, and *J*5 execute in turn to completion.

Job	r_i	e_i	π_i	Critical Sections
J_1	7	3	1	[*Shaded*; 1]
J_2	5	3	2	[*Black*; 1]
J_3	4	2	3	
J_4	2	6	4	[*Shaded*; 4 [*Black*; 1.5]]
J_5	0	6	5	[*Black*; 4]

(a)



(b)

FIGURE 8–8 Example illustrating transitive inheritance of priority inheritance. (a) Parameters of jobs. (b) Schedule under priority inheritance.

From this example, we notice that a high-priority task can experience two kinds of blocking:

Direct blocking. It occurs when a higher-priority task tries to acquire a resource already held by a lower-priority task. Direct blocking is necessary to ensure the consistency of the shared resources.

Push-through blocking. It occurs when a medium-priority task is blocked by a low-priority task that has inherited a higher priority from a task it directly blocks. Push-through blocking is necessary to avoid unbounded priority inversion.

Note that in most situations when a task exits a critical section, it resumes the priority it had when it entered. This, however, is not always true. Consider the example illustrated in Figure 7.9. Here, task $\tau 1$ uses a resource *Ra* guarded by a semaphore *Sa*, task $\tau 2$ uses a resource *Rb* guarded by a semaphore *Sb*, and task $\tau 3$ uses both resources in a nested fashion (*Sa* is locked first). At time $t1$, $\tau 2$ preempts $\tau 3$ within its nested critical section;

hence, at time *t*2, when *τ*2 attempts to lock *Sb*, *τ*3 inherits its priority, *P*2. Similarly, at time *t*3, *τ*1 preempts *τ*3 within the same critical section, and at time *t*4, when *τ*1 attempts to lock *Sa*, *τ*3 inherits the priority *P*1. At time *t*5, when *τ*3 unlocks semaphore *Sb*, task *τ*2 is awakened but *τ*1 is still blocked; hence, *τ*3 continues its execution at the priority of *τ*1. At time *t*6, *τ*3 unlocks *Sa* and, since no other tasks are blocked, *τ*3 resumes its original priority *P*3

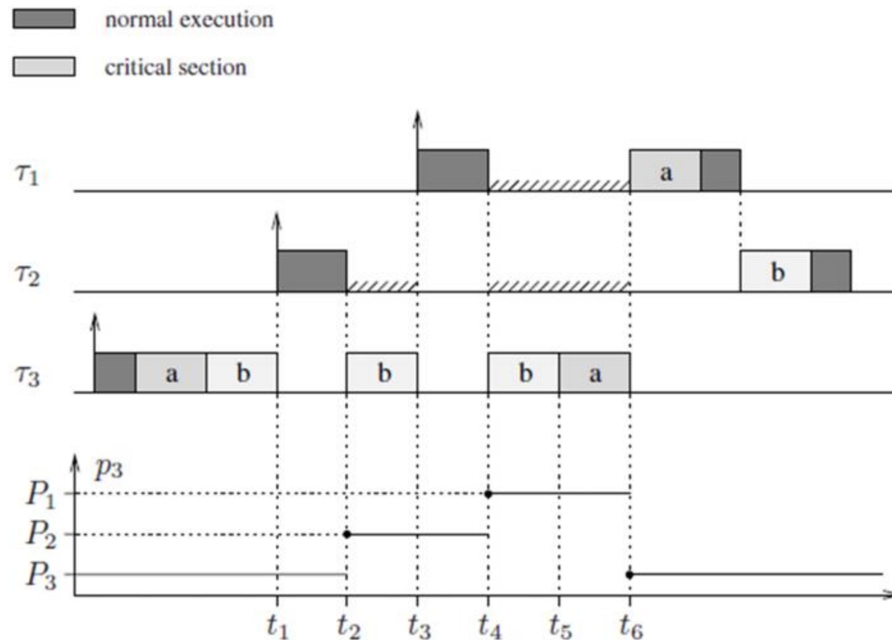


Figure 7.9 Priority inheritance with nested critical sections.

An example of transitive priority inheritance is shown in Figure 7.10. Here, task *τ* 1 uses a resource *Ra* guarded by a semaphore *Sa*, task *τ*3 uses a resource *Rb* guarded by a semaphore *Sb*, and task *τ*2 uses both resources in a nested fashion (*Sa* protects the external critical section and *Sb* the internal one). At time *t*1, *τ*3 is preempted within its critical section by *τ*2, which in turn enters its first critical section (the one guarded by *Sa*), and at time *t*2 it is blocked on semaphore *Sb*. As a consequence, *τ*3 resumes and inherits the priority *P*2. At time *t*3, *τ*3 is preempted by *τ*1, which at time *t*4 tries to acquire *Ra*. Since *Sa* is locked by *τ*2, *τ*2 inherits *P*1. However, *τ*2 is blocked by *τ*3; hence, for transitivity, *τ*3 inherits the priority *P*1 via *τ*2. When *τ*3 exits its critical section, no other tasks are blocked by it; thus it resumes its nominal priority *P*3. Priority *P*1 is now inherited by *τ*2, which still blocks *τ*1 until time *t*6.

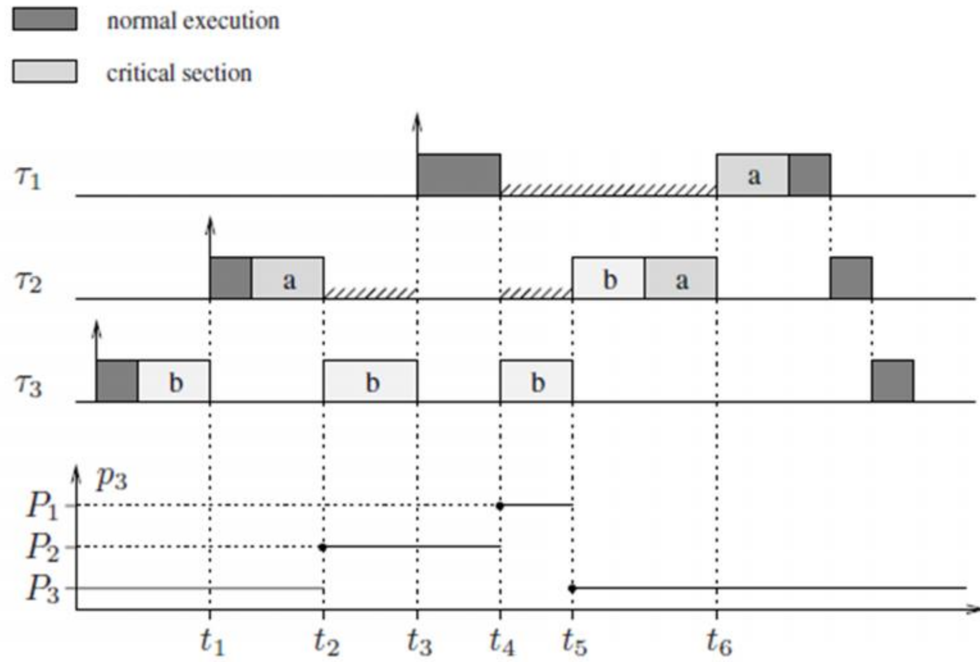


Figure 7.10 Example of transitive priority inheritance.

esired effect of the protocol