Lecture 11

PRIORITY CEILING PROTOCOL

The Priority Ceiling Protocol (PCP) was introduced by Sha, Rajkumar, and Lehoczky [SRL90] to bound the priority inversion phenomenon and prevent the formation of deadlocks and chained blocking.
The basic idea of this method is to extend the Priority Inheritance Protocol with a rule for granting a lock request on a free semaphore. To avoid multiple blocking, this rule does not allow a task to enter a critical section if there are locked semaphores that could block it. This means that once a task enters its first critical section, it can never be blocked by lower-priority tasks until its completion In order to realize this idea, each semaphore is assigned a *priority ceiling* equal to the highest priority of the tasks that can lock it. Then, a task *τ i* is allowed to enter a critical section only if its priority is higher than all priority ceilings of the semaphores currently locked by tasks other than *τi*.
The Priority Ceiling Protocol can be defined as follows:

1. Each semaphore *Sk* is assigned a priority ceiling *C*(*Sk*) equal to the highest priority of the tasks that can lock it. Note that *C*(*Sk*) is a static value that can be computed off-line:

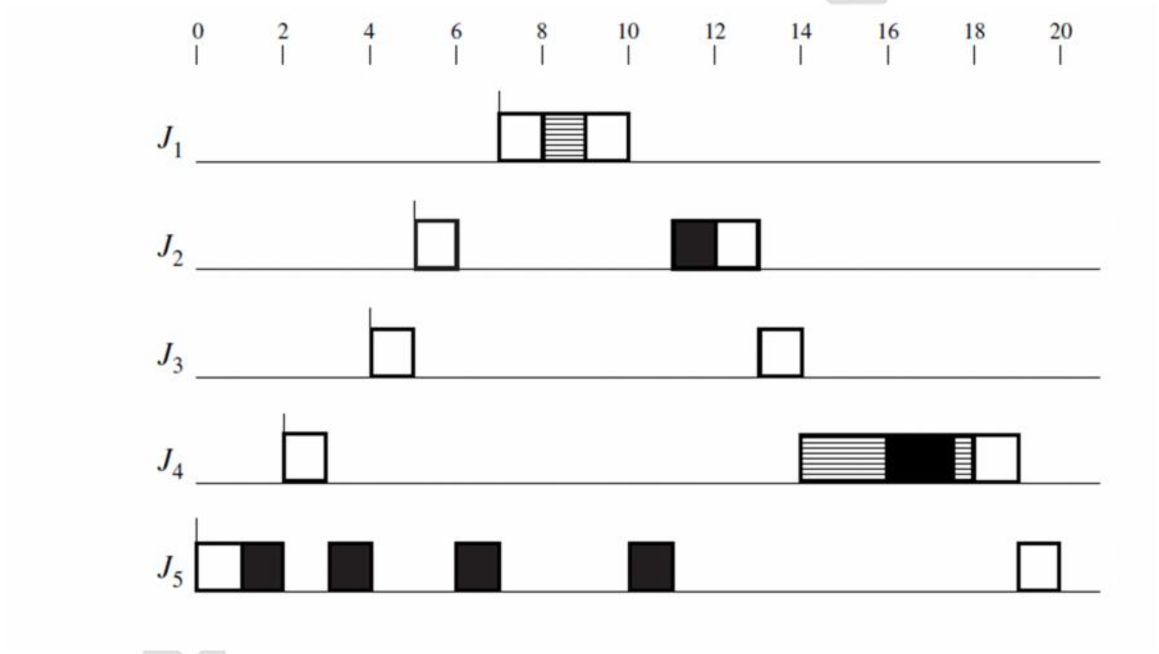
C(*Sk*) def = max *I {Pi* | *Sk* □ *σi}*. (7.13)

2. Let *τi* be the task with the highest priority among all tasks ready to run; thus, *τ I* is assigned the processor.
3. Let *S*□ be the semaphore with the highest ceiling among all the semaphores currently locked by tasks other than *τi* and let *C*(*S*□) be its ceiling.
4. To enter a critical section guarded by a semaphore *Sk*, *τi* must have a priority higher than *C*(*S*□). If *Pi* ≤ *C*(*S*□), the lock on *Sk* is denied and *τi* is said to be blocked on semaphore *S*□ by the task that holds the lock on *S*□.
5. When a task *τi* is blocked on a semaphore, it transmits its priority to the task, say *τj* , that holds that semaphore. Hence, *τj* resumes and executes the rest of its critical section with the priority of *τi*. Task *τj* is said to *inherit* the priority of *τi*.
6. In general, a task inherits the highest priority of the tasks blocked by it. That is, at every instant,

pj(*Rk*) =max*{Pj* , max *h{Ph*|*τh* is blocked on *Rk}}*. (7.14)

7. When *τj* exits a critical section, it unlocks the semaphore and the highest-priority task, if any, blocked on that semaphore is awakened. Moreover, the active priority of *τj* is updated as follows: if no other tasks are blocked by *τj* , *pj* is set to the nominal priority *Pj* ; otherwise, it is set to the highest priority of the tasks blocked by *τj* , according to Equation (7.14).
8. 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:
Figure 8–10 shows the schedule of the system of jobs whose parameters are listed in Figure 8–8(a) when their accesses to resources are controlled by the priority-ceiling



As stated earlier, the priority ceilings of the resources *Black* and *Shaded* are 2 and 1, respectively.

1. In the interval *(*0*,* 3], this schedule is the same as the schedule shown in Figure 8–8, which is produced under the basic priority-inheritance protocol. In particular, the ceiling of the system at time 1 is . When *J*5 requests *Black*, it is allocated the resource according to (i) in part (b) of rule 2. After *Black* is allocated, the ceiling of the system is raised to 2, the priority ceiling of *Black*.

2. At time 3, *J*4 requests *Shaded*. *Shaded* is free; however, because the ceiling ˆ *_(*3*)* (= 2) of the system is higher than the priority of *J*4, *J*4's request is denied according to (ii) in part (b) of rule 2. *J*4 is blocked, and *J*5 inherits *J*4's priority and executes at priority 4.

3. At time 4, *J*3 preempts *J*5, and at time 5, *J*2 preempts *J*3. At time 6, *J*2 requests *Black* and becomes directly blocked by *J*5. Consequently, *J*5 inherits the priority 2; it executes until *J*1 becomes ready and preempts it. During all this time, the ceiling of the system remains at 2.

4. When *J*1 requests *Shaded* at time 8, its priority is higher than the ceiling of the system. Hence, its request is granted according to (i) in part (b) of rule 2, allowing it to enter its critical section and complete by the time 10. At time 10, *J*3 and *J*5 are ready. The latter has a higher priority (i.e., 2); it resumes.

5. At 11, when *J*5 releases *Black*, its priority returns to 5, and the ceiling of the system drops to . *J*2 becomes unblocked, is allocated *Black* [according to (i) in part (b) of rule 2], and starts to execute.

6. At time 14, after *J*2 and *J*3 complete, *J*4 has the processor and is granted the resource *Shaded* because its priority is higher than , the ceiling of the system at the time. It starts to execute. The ceiling of the system is raised to 1, the priority ceiling of *Shaded*.

7. At time 16, *J*4 requests *Black*, which is free. The priority of *J*4 is lower than ˆ _*(*16*)*, but *J*4 is the job holding the resource (i.e., *Shaded*) whose priority ceiling is equal to ˆ _*(*16*)*. Hence, according to (ii) of part (b) of rule 2, *J*4 is granted *Black*. It continues to execute. The rest of the schedule is self-explanatory.

Comparing the schedules in Figures 8–8 and 8–10, we see that when priority-ceiling protocol is used, *J*4 is blocked at time 3 according to (ii) of part (b) of rule 2. A consequence is that the higher priority jobs *J*1, *J*2, and *J*3 all complete earlier at the expense of the lower priority job *J*4. This is the desired effect of the protocol