Lab 3 for uC/OS-II: Ceiling Priority Protocol

Prof. Li-Pin Chang

ESSLab@NCTU

Objective

- To implement Ceiling Priority Protocol for ucOS’s mutex locks

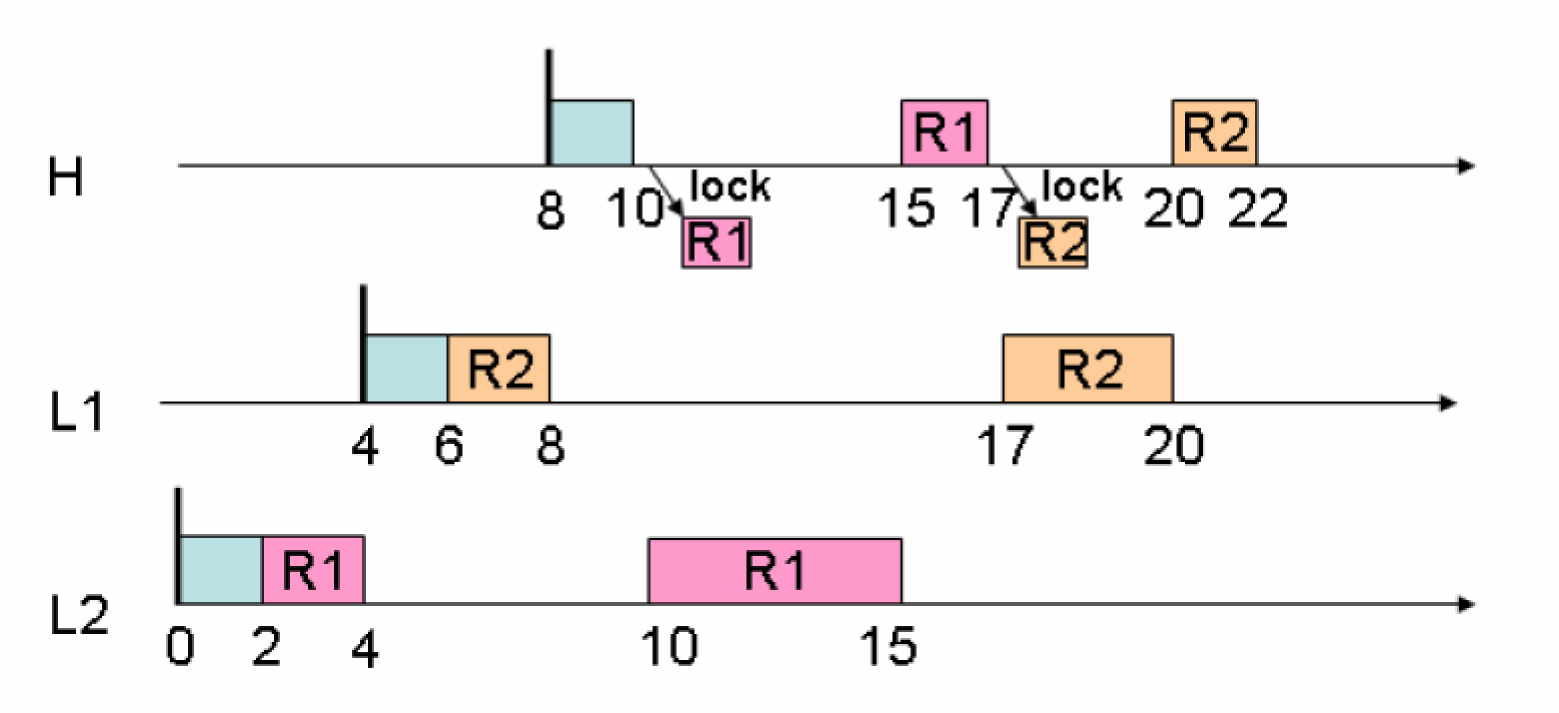
uC/OS Mutex Locks

- A mutex lock is associated with a "priority"
  - Its priority is higher than the highest locker
  - E.g., if T_3 and T_4 share a lock, the priority of the lock should be set to 2
  - When T4 blocks T3, then T4's priority becomes 2

Disadvantages of PIP

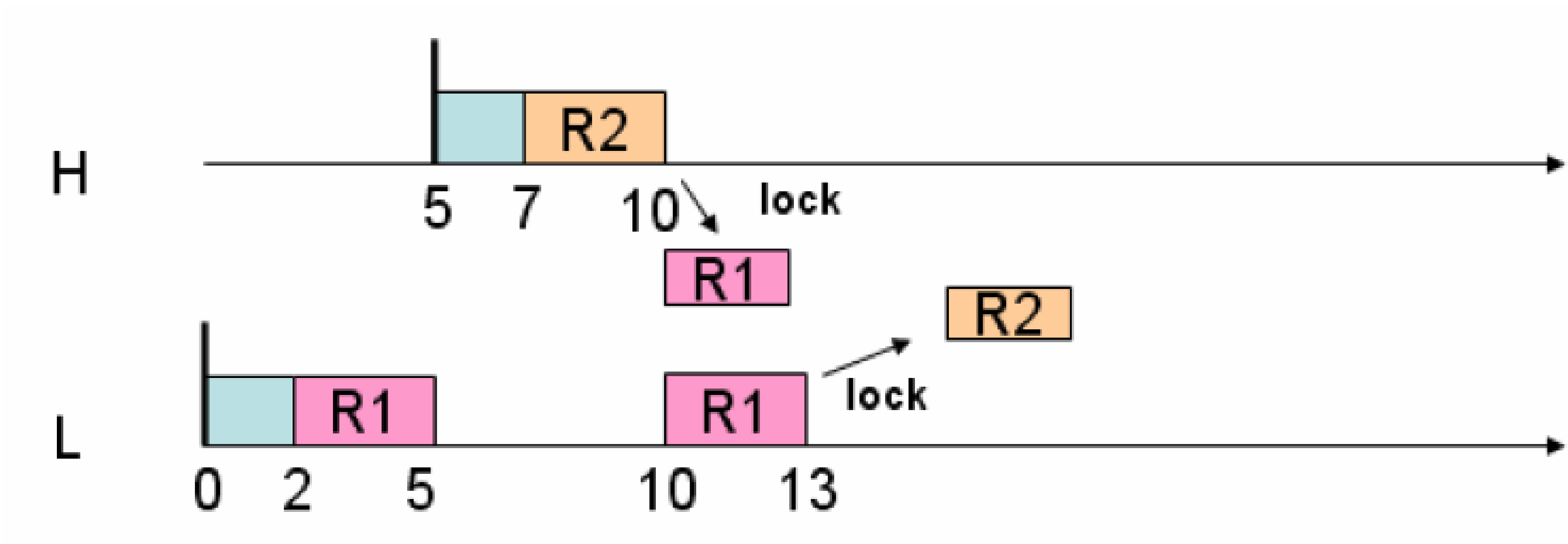
- The “PIP” avoids uncontrolled priority inversion, but it has two disadvantages
  - A high priority task can be blocked multiple times
  - Deadlocks are possible

Scenario 1: Multiple blocking in ucOS2 PIP



Task H is in turn blocked by task L1 and task L2

Scenario 2: Deadlock in uCOS-2 PIP

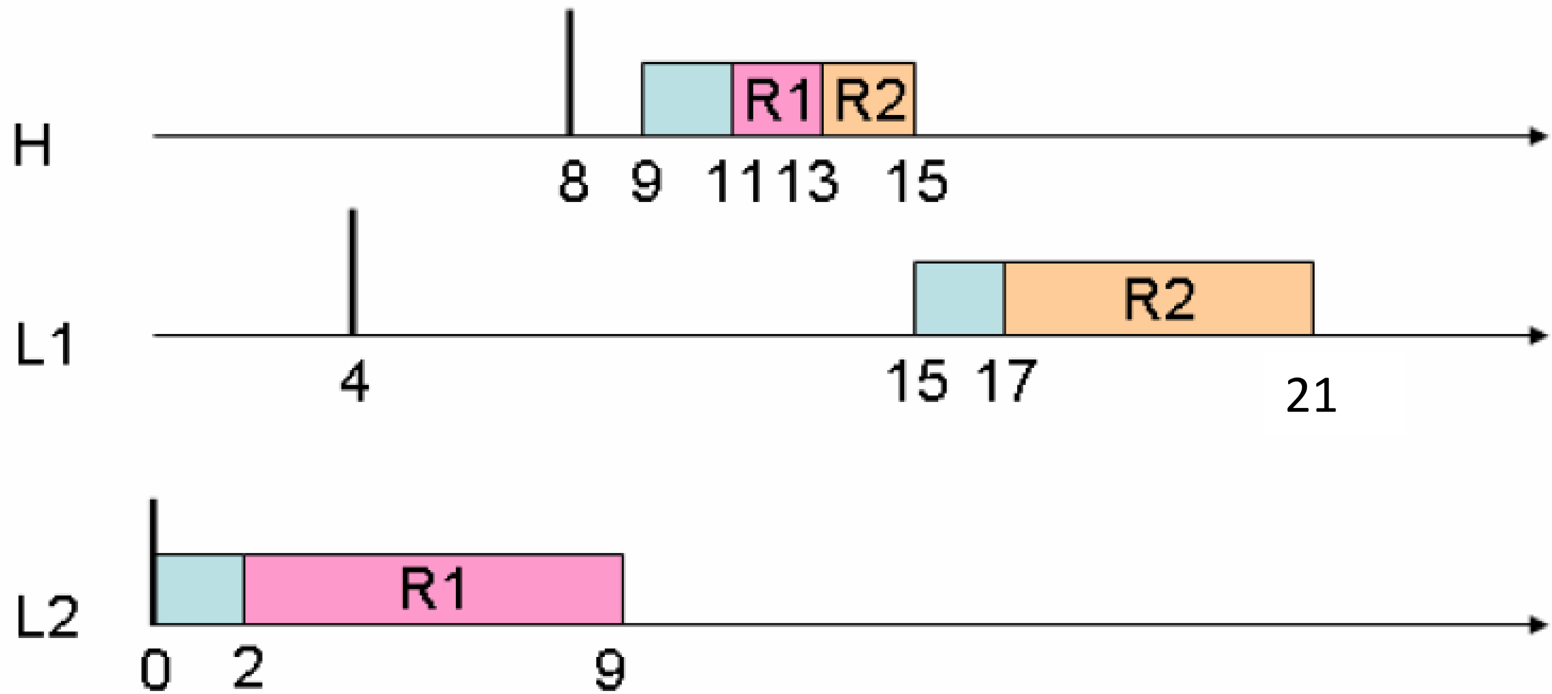


A deadlock

Ceiling Priority Protocol

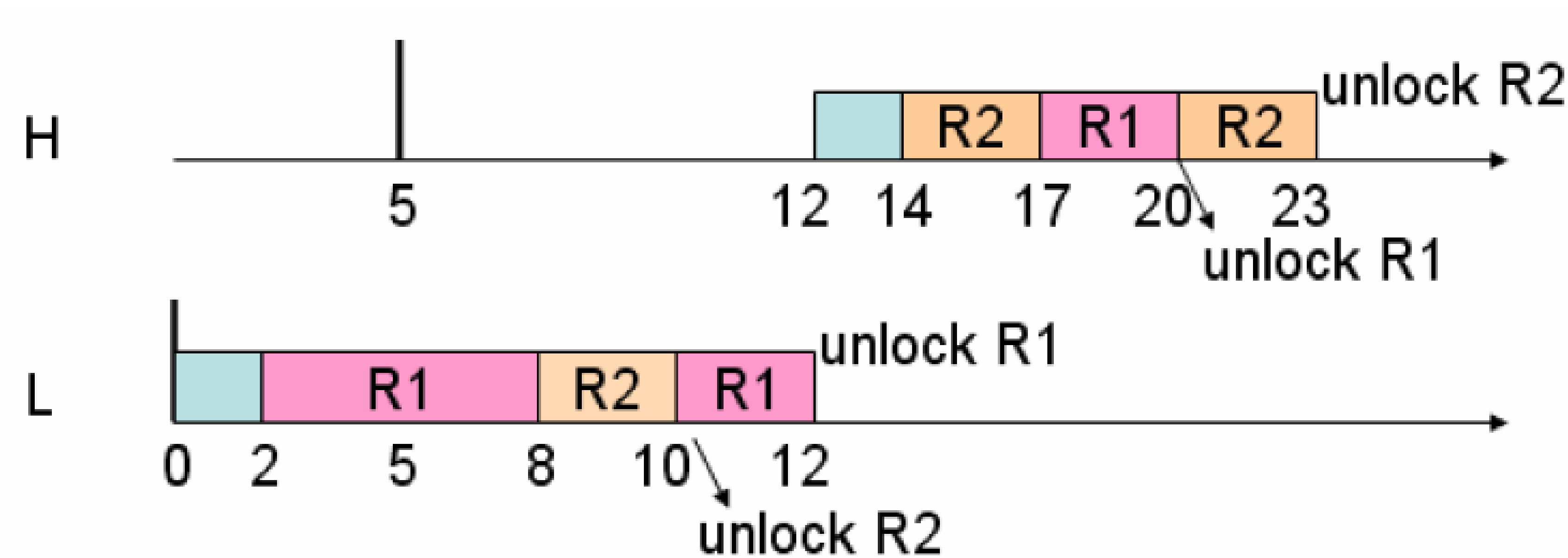
- Highest-Locker Protocol
- When a task acquires a mutex lock, its priority becomes the highest among all lockers' priorities
- In uC/OS, the we use the mutex's priority as the highest-locker's priority
  - Immediately higher than all lockers' priorities

S1 CPP: Removing Multiple Blockings



The result of applying CPP

S2 CPP: Avoiding Deadlocks



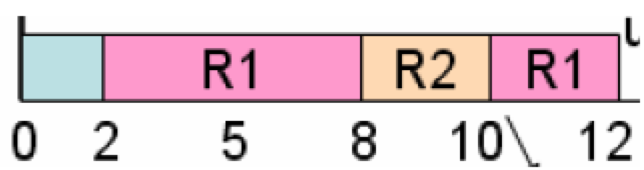
Deadlocks are avoided by using CPP

Implementation

- Reuse your code of Lab 1 (do not re-use EDF)
- Modify the following two functions
  - OSMutexPend()
    - If mutex is free, boost the locker's (caller) priority
  - OSMutexPost()
    - Restore the original priority of the locker
- Do not use OSTaskChangePrio()
  - It calls OS_Sched() and results in unexpected behaviors

Implementation

- All tasks should add proper OSTimeDly() at their beginning to emulate their arrival times
- Emulate durations of CPU execution and resource use with your code from Lab 0
  - 2 ticks → lock R1→ 6 ticks → lock R2 → 2 ticks → unlock R2 → 2 ticks → unlock R1



Output

- Similar to those in prior labs, but add lock/unlock events
- Output the results of using CPP for Scenarios 1 and 2 (shown previously)

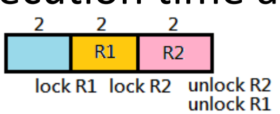
Output Example of S1

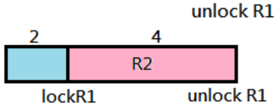
Priority initialization:
R1: 1
R2: 2
Task1: 3
Task2: 4
Task3: 5

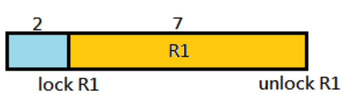
Task arrival time:
Task1: 8
Task2: 4
Task3: 0

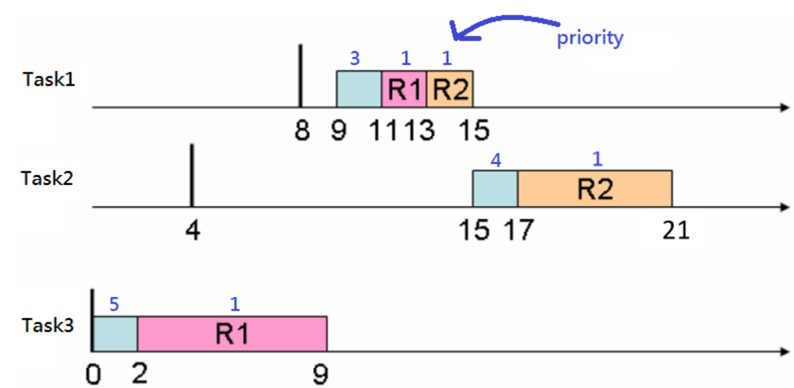
```
20    lock      R1  (Prio=5 changes to=1)
90    unlock    R1  (Prio=1 changes to=5)
90    complete      5          3
110   lock      R1  (Prio=3 changes to=1)
130   lock      R2  (Prio=1 changes to=1)
150   unlock    R2  (Prio=1 changes to=1)
150   unlock    R1  (Prio=1 changes to=3)
150   complete      3          4
170   lock      R2  (Prio=4 changes to=2)
210   unlock    R2  (Prio=2 changes to=4)
210   complete      4          19
```

Task execution time and resource used:

Task1:


Task2:


Task3:




- Here, at time 90, it is actually a “preempt” because T3 calls OSMutexPost(R1), which internally calls OS_Sched() to surrender the CPU to T1 (HPT)
- Because our prior lab defines “OS_Sched()” as “complete” so it is okay. T3 ends after unlocking R1 anyway. The same to times 150 and 220.

```
 20     lock     R1   (Prio=5 changes to=1)
 90   unlock     R1   (Prio=1 changes to=5)
 90   complete           5          3
----------------------------------------------------
110     lock     R1   (Prio=3 changes to=1)
130     lock     R2   (Prio=1 changes to=1)
150   unlock     R2   (Prio=1 changes to=1)
150   unlock     R1   (Prio=1 changes to=3)
150   complete           3          4
170     lock     R2   (Prio=4 changes to=2)
220   unlock     R2   (Prio=2 changes to=4)
220   complete           4         19
```