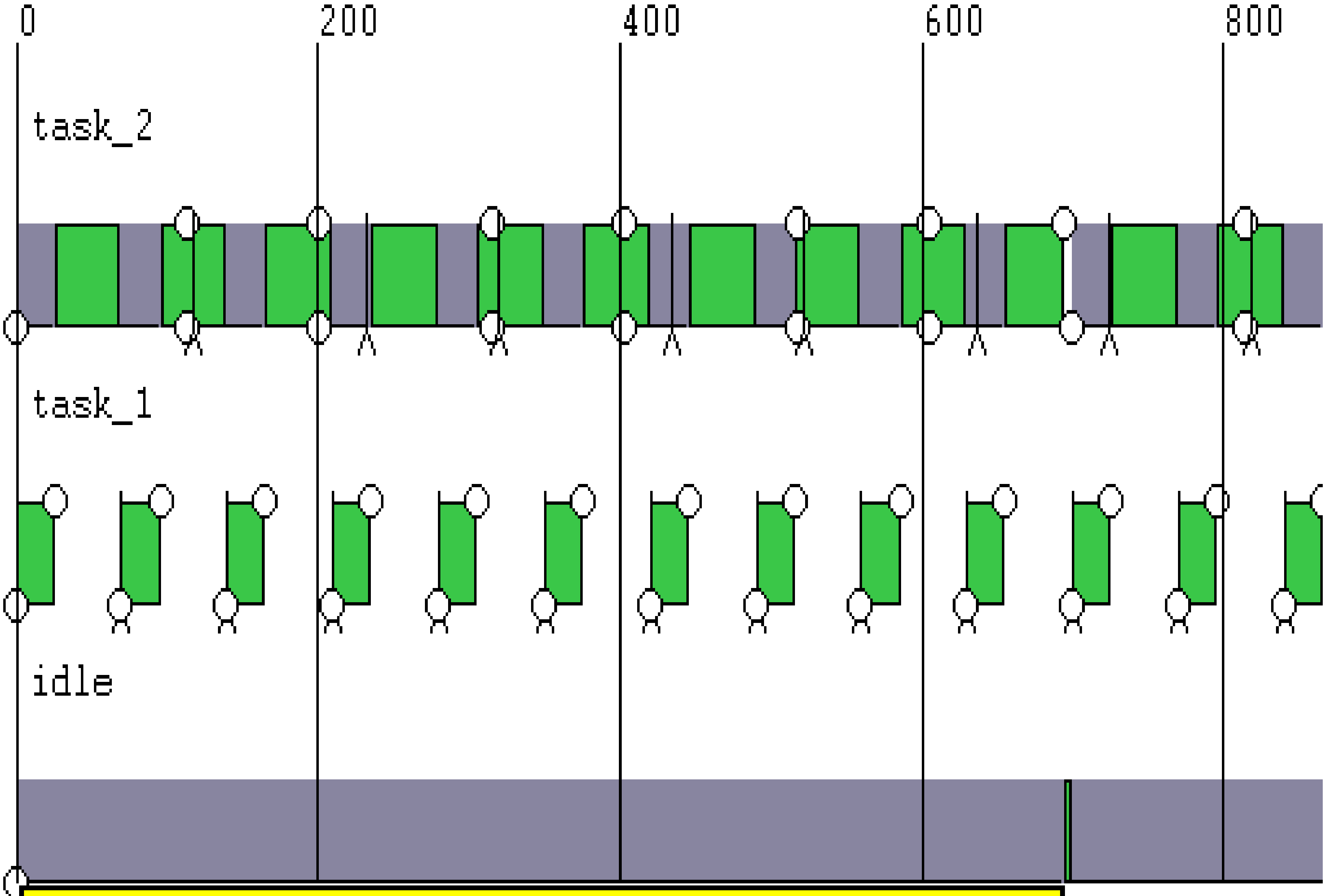
CIS 721 - Real-Time Systems
Lecture 11: Priority Inheritance

Mitch Neilsen
neilsen@ksu.edu

Lehoczky's Arbitrary Deadline Example

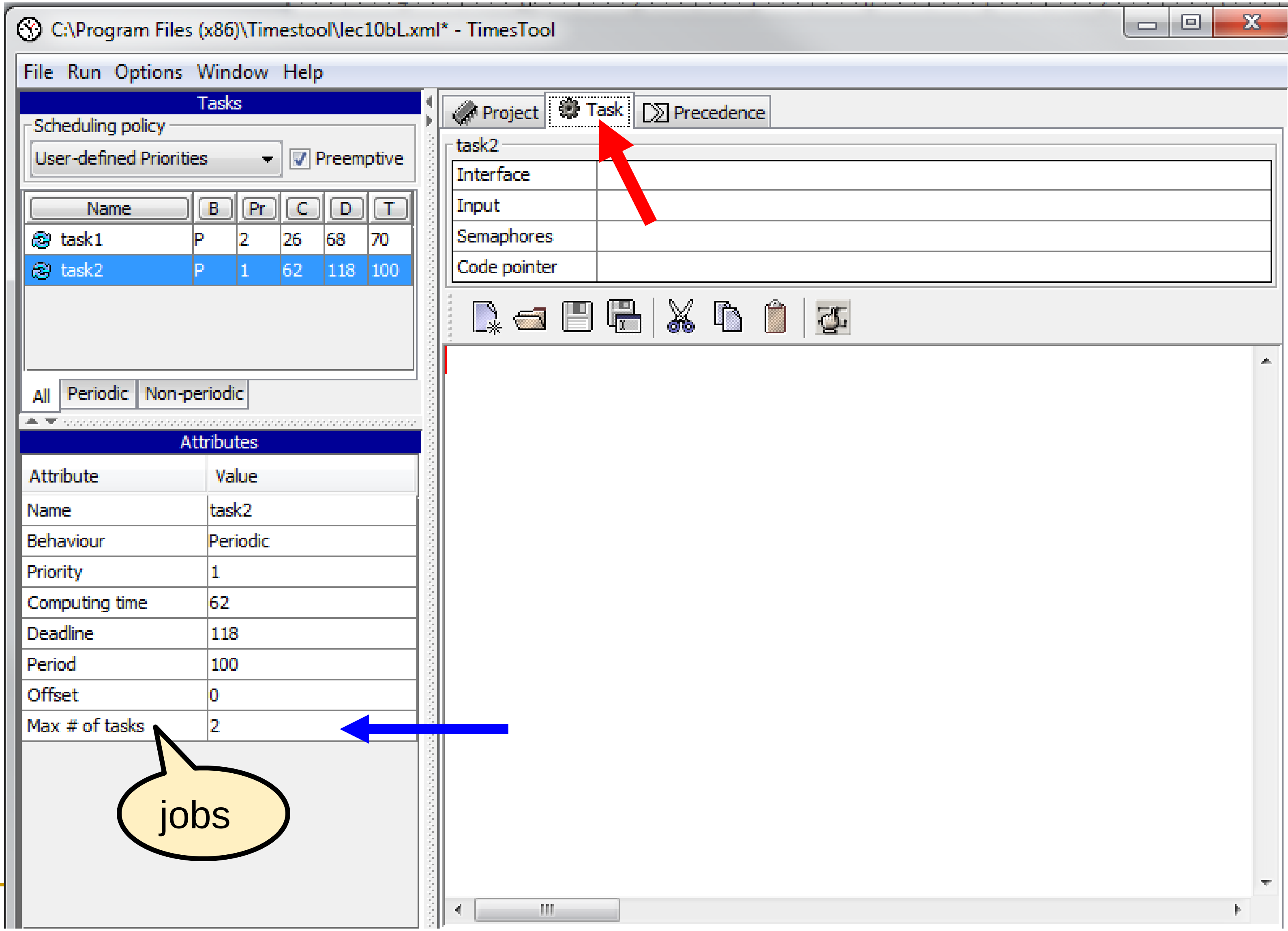
Task τ_i	Period T_i	Run-Time C_i	Phase φ_i	Deadline D_i
τ_1	70	26	0	68
τ_2	100	62	0	118

114, 102, 116, 104, 118, 106, 94 - task_2 WCRT

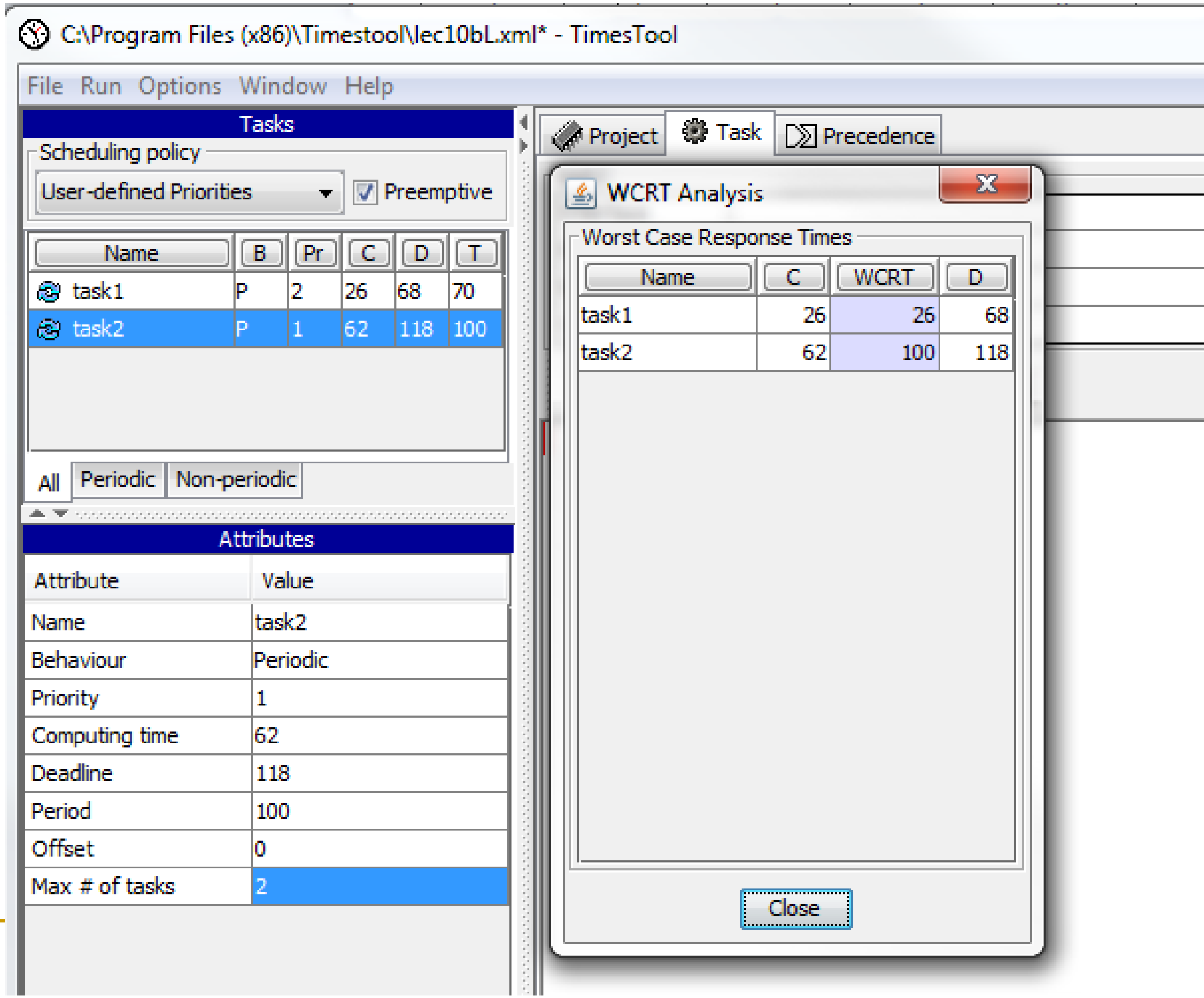


Level-i Busy Period for task_2 = [0, 694]

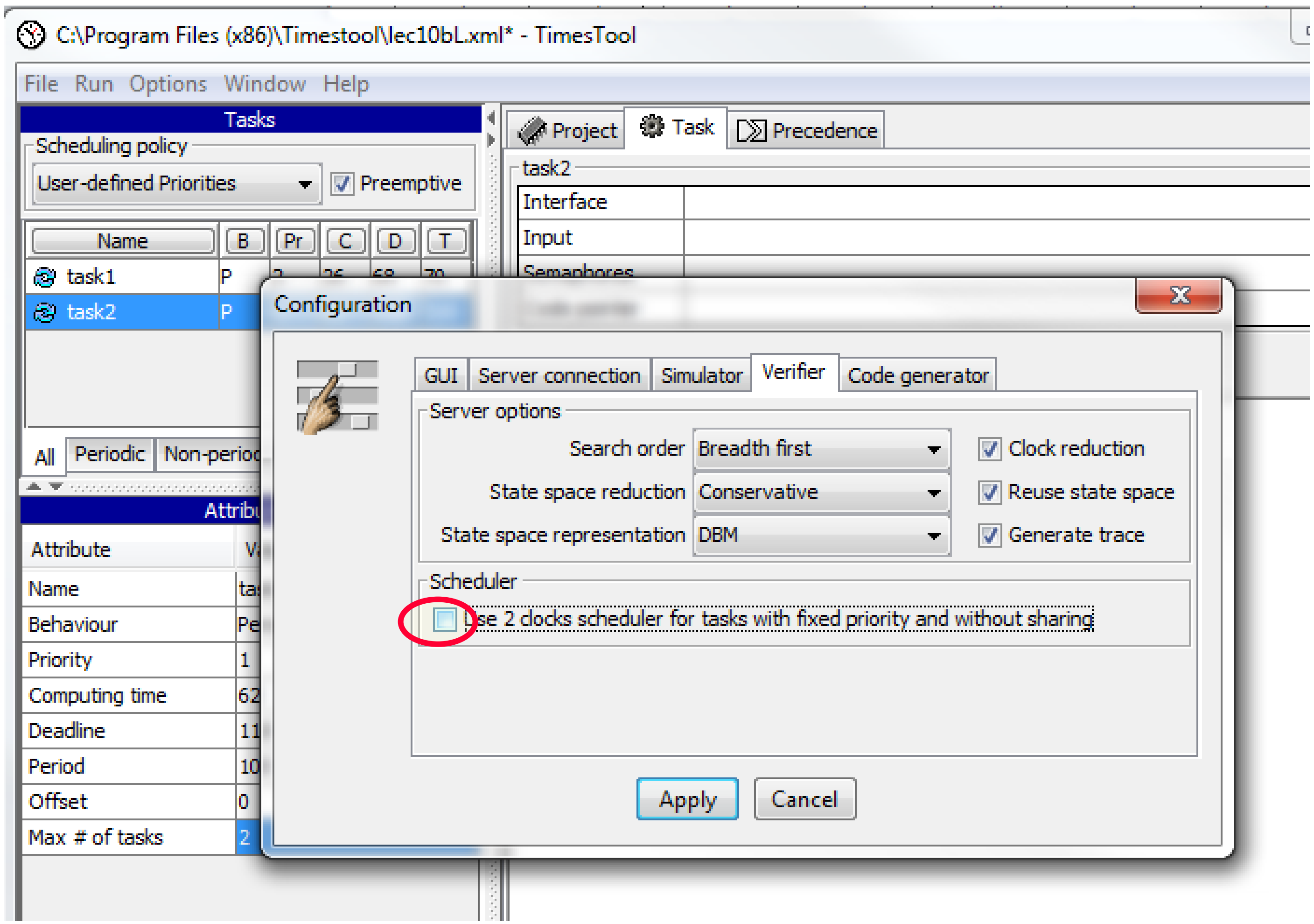
TimesTool Adendum - deadlines > periods



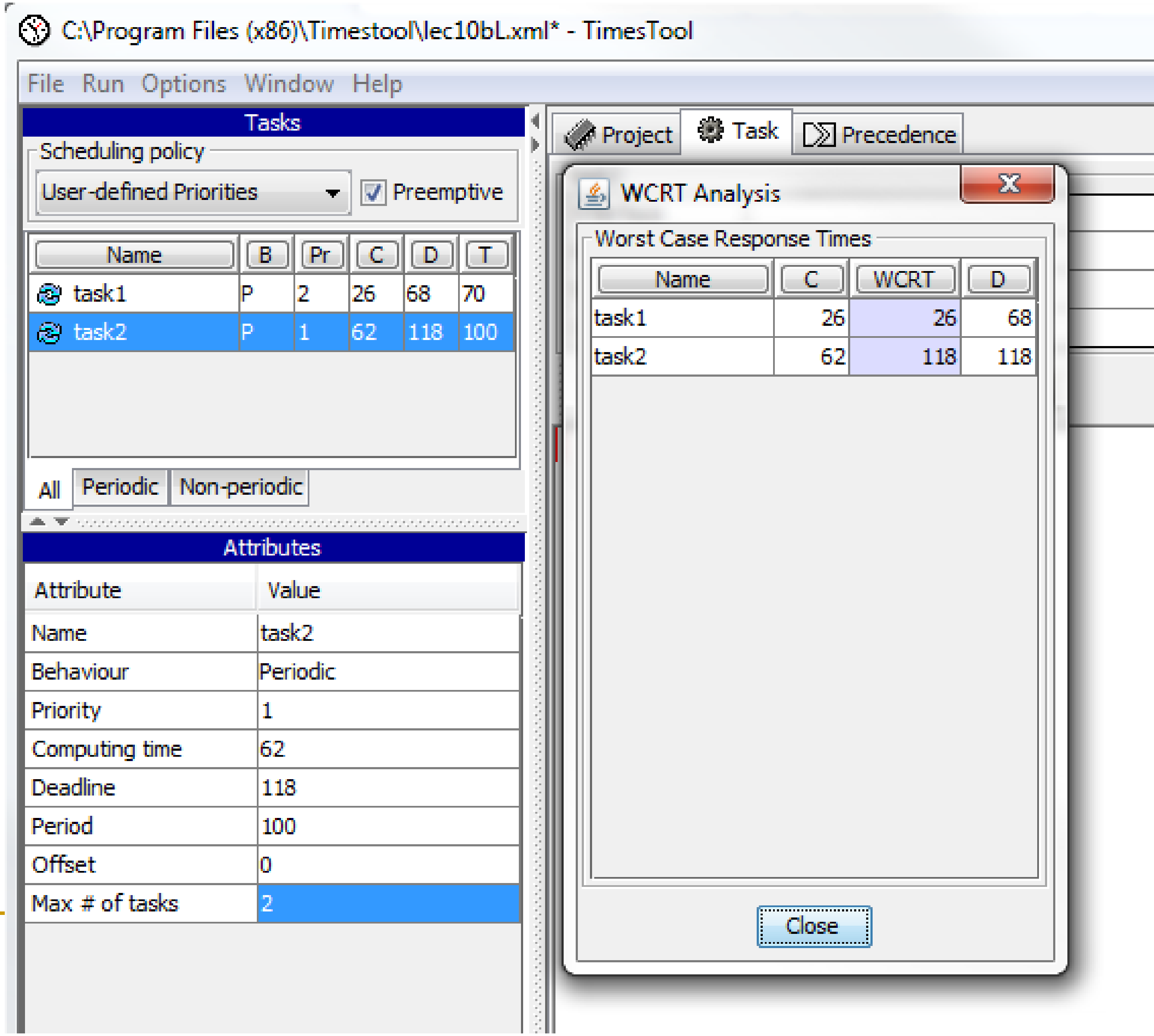
Bug: Schedulability Analysis with 2 clocks



Fix: Options + Configuration… + Verifier Tab – Uncheck Use 2 clocks scheduler



Run + Schedulability Analysis Success



Outline

Resources and Resource Access Control (Ch. 8)

- Basic Priority Inheritance Protocol
- NonPreemptive Critical Sections (NPCS)
- Basic (Original) Priority Ceiling Protocol
- Stack-Based Priority Ceiling (Ceiling Priority) Protocol

Priority Inheritance Protocols

- L. Sha, R. Rajkumar, J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", IEEE Transactions on Computers, Vol. 39, No. 9, pp. 1175-1185, 1990

Sharing Resources (Ch. 8)

- Many applications require concurrent access to shared resources. To synchronize access to a shared resource, **semaphores** can be used.
- If a task tries to lock a binary semaphore that is already locked, then the task is **blocked**.
- How much does blocking due to resource sharing impact feasibility?
- Can scheduling algorithms be modified to support resource sharing?

Example (with no blocking)

Task τ_i	Period T_i	Deadline D_i	Run-Time C_i
τ_1	50	10	5
τ_2	500	500	250
τ_3	3000	3000	1000

With no blocking:

Task	C	T	D	Prio	PT	B	S	F	WCRT
1	5	50	10	3	3	0	0	5	5
2	250	500	500	2	2	0	5	280	280
3	1000	3000	3000	1	1	0	280	2500	2500

Example (continued)

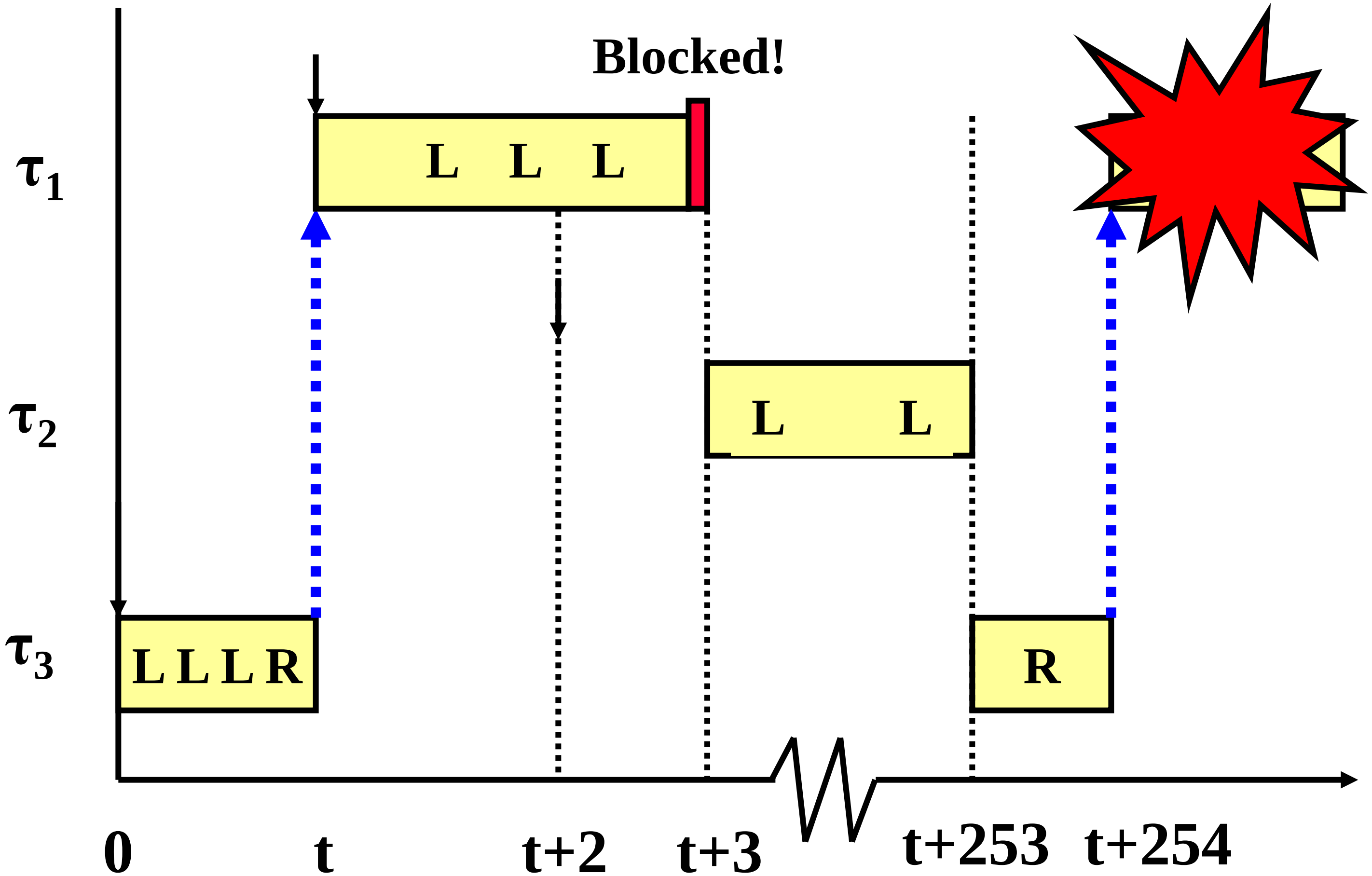
- Suppose that tasks τ_1 and τ_3 share some data.
- Access to the data is restricted using *semaphore* **s**:
  - **each task executes the following code:**
    - **do local work (L)**
    - **sem_wait(s) (or P(s))** critical section
      - **access shared resource (R)**
    - **sem_signal(s) (or V(s))**
    - **do more local work (L)**

Tasks

- **Task τ_1 L L L R L**
- **Task τ_2 L L ... L**
- **Task τ_3 L L L R R L ... L**

- **L = Local computation unit while NOT using the shared resource**
- **R = Resource in use in critical section**

Priority Inversion



Blocking

- **Blocking** occurs when a higher priority task is waiting on a lower priority task; e.g., **blocking** occurs when a lower priority task is holding a resource that is requested by a higher priority task or when a lower priority task is non-preemptive and is executing when a higher priority task arrives for execution.

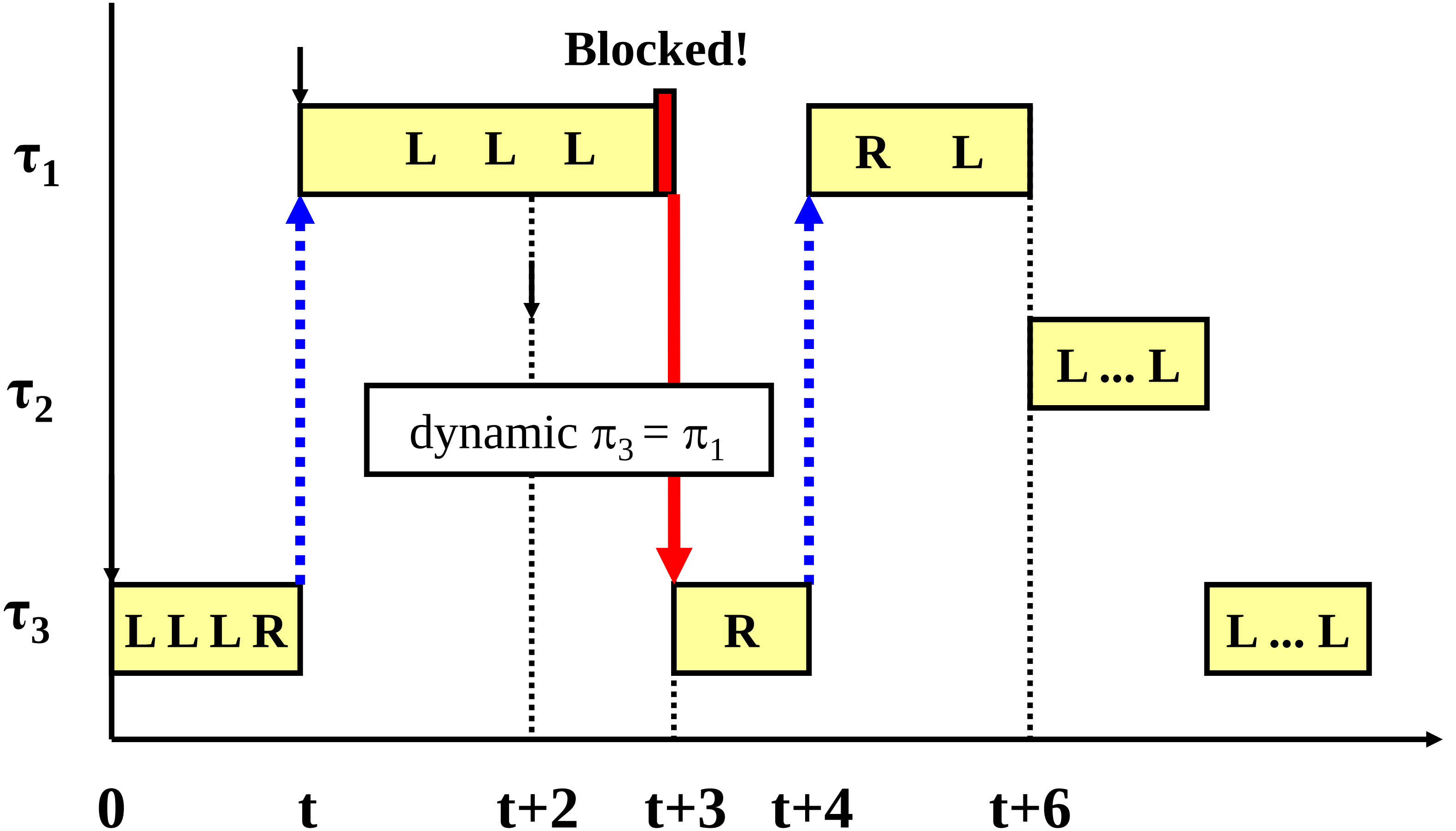
Priority Inheritance Protocols

- **Protocols**
  - Basic Priority Inheritance Protocol
  - NonPreemptive Critical Sections (NPCS)
  - Basic (Original) Priority Ceiling Protocol
  - Stack-Based Priority Ceiling Protocol
- **Analysis**
  - Utilization-Based Test
  - Response Time Analysis

Basic Priority Inheritance Protocol

- For each resource (semaphore), a list of blocked tasks must be stored in a priority queue.
- A task τ_i uses its assigned priority, unless it is in its critical section and blocks some higher priority tasks; when blocking occurs, task τ_i **inherits** the highest dynamic priority of all tasks it blocks.
- Priority inheritance is **transitive**; that is, if task τ_i blocks τ_j and τ_j blocks τ_k, then τ_i can inherit the priority of τ_k (π_k).

Priority Inheritance



Types of Blocking

- **Direct** - task τ_1 and τ_2 use a shared resource. If the low priority task is in its critical section, then it **directly** blocks the high priority task.
- **Indirect (push-through)** - if a low priority task inherits the priority of a high priority task, a medium priority task can be blocked while the low priority task is in its critical section.

```
periodic task_1
  period        20   deadline 10   offset 3
  priority 1
  [2,2]
  inheritance::respop(sem)
  pop(sem)
  [1,1]
  inheritance::resvop(sem)
  vop(sem)
  [1,1]
endper
periodic task_2
  period        20   deadline 15    offset 2
  priority 2
  inheritance::respop(sem)
  pop(sem)
  [1,1]
  inheritance::resvop(sem)
  vop(sem)
  [2,2]
endper
periodic task_3
  period        20   deadline 20
  priority 3
  [1,1]
  inheritance::respop(sem)
  pop(sem)
  [4,4]
  inheritance::resvop(sem)
  vop(sem)
  [1,1]
endper
```

Example: lec12a.str

Task τ_i	Period T_i	Deadline D_i	Run-Time C_i	Offset O_i
τ_1	20	10	4 LLRL	3
τ_2	20	15	3 RLL	2
τ_3	20	20	6 LRRRRL	0

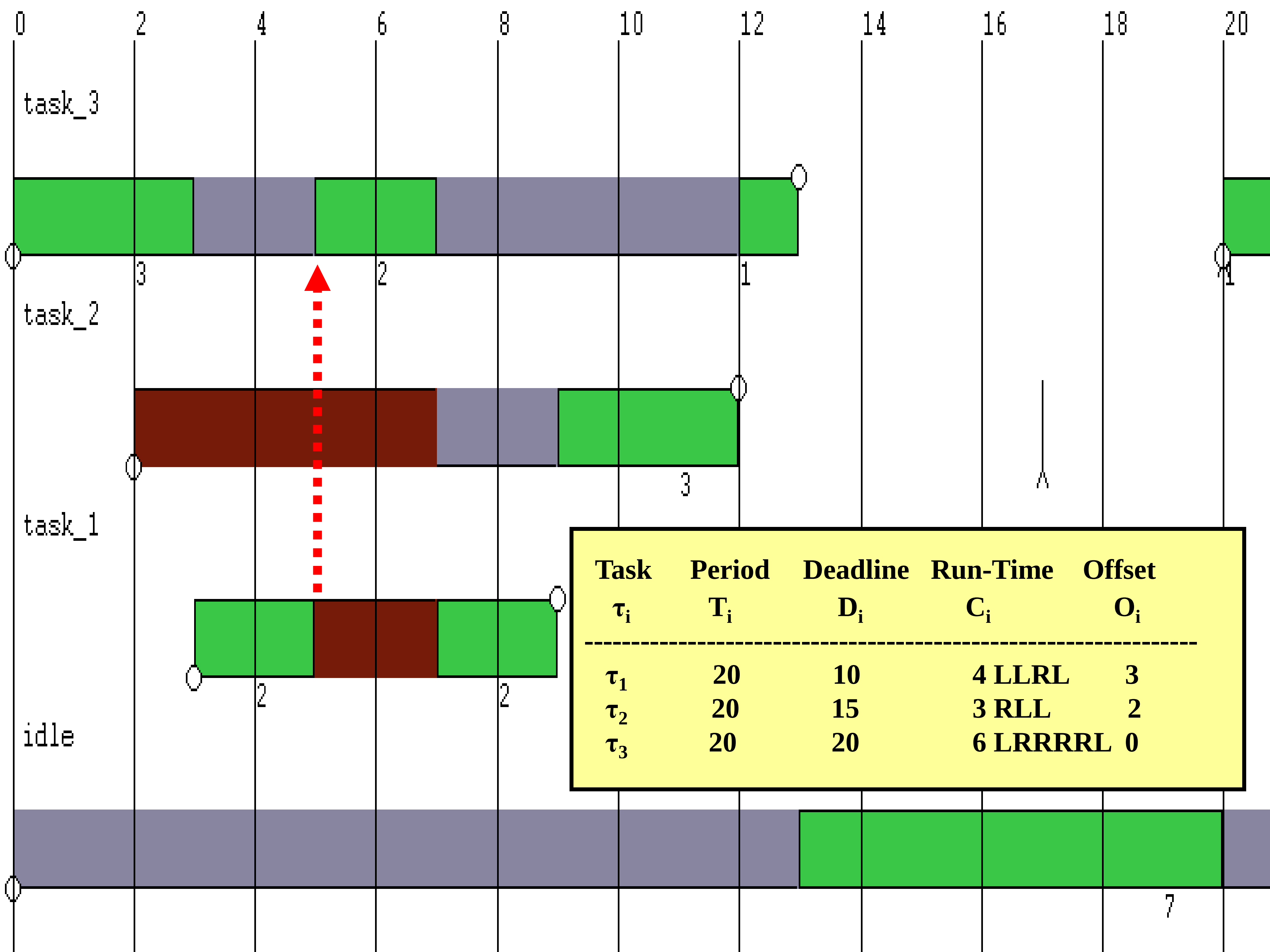
```
system
    node node_1
      processor proc_1

          semaphore sem = 1

          /*  Whan a task becomes blocked on a semaphore,    */
          /* the priorites of any tasks which are locking    */
          /* that semaphore are raised to the priority of the      */
          /* task which will become blocked.                 */
          resource inheritance
            variable locker
            variable waiter
            variable myused

            method respop (sema)
            /* If the semaphore is not available, then any task    */
            /* which is locking the semaphore, and has a lower     */
            /* priority than the caller, has its priority raised to      */
            /* that of the caller.                              */
                if cnt of sema = 0 then
                  for locker in locking of sema max 999
                      if effpri of locker > effpri of mytask then
                        effpri of locker := effpri of mytask
            endmet

            method resvop (sema)
            /* Reset the priority of the caller to the highest     */
            /* priority amongst its base priority and the          */
            /* priorities of any tasks blocked on any other        */
            /* semaphores which the caller holds.                  */
                effpri of mytask := baspri of mytask
                for myused in locking of mytask max 999
                  if myused != sema then
                      for waiter in waiting of sema max 999
                        if effpri of waiter < effpri of mytask then
                            effpri of mytask := effpri of waiter
            endmet
          endres
```



Task τ_i	Period T_i	Deadline D_i	Run-Time C_i	Offset O_i
τ_1	20	10	4 LLRL	3
τ_2	20	15	3 RLL	2
τ_3	20	20	6 LRRRRL	0

Properties of Priority Inheritance

- Under the basic priority inheritance protocol, if there are *m* semaphores that can block a job J in a task, then J can be blocked at most *m* times; e.g., on each semaphore at most once.

Problems

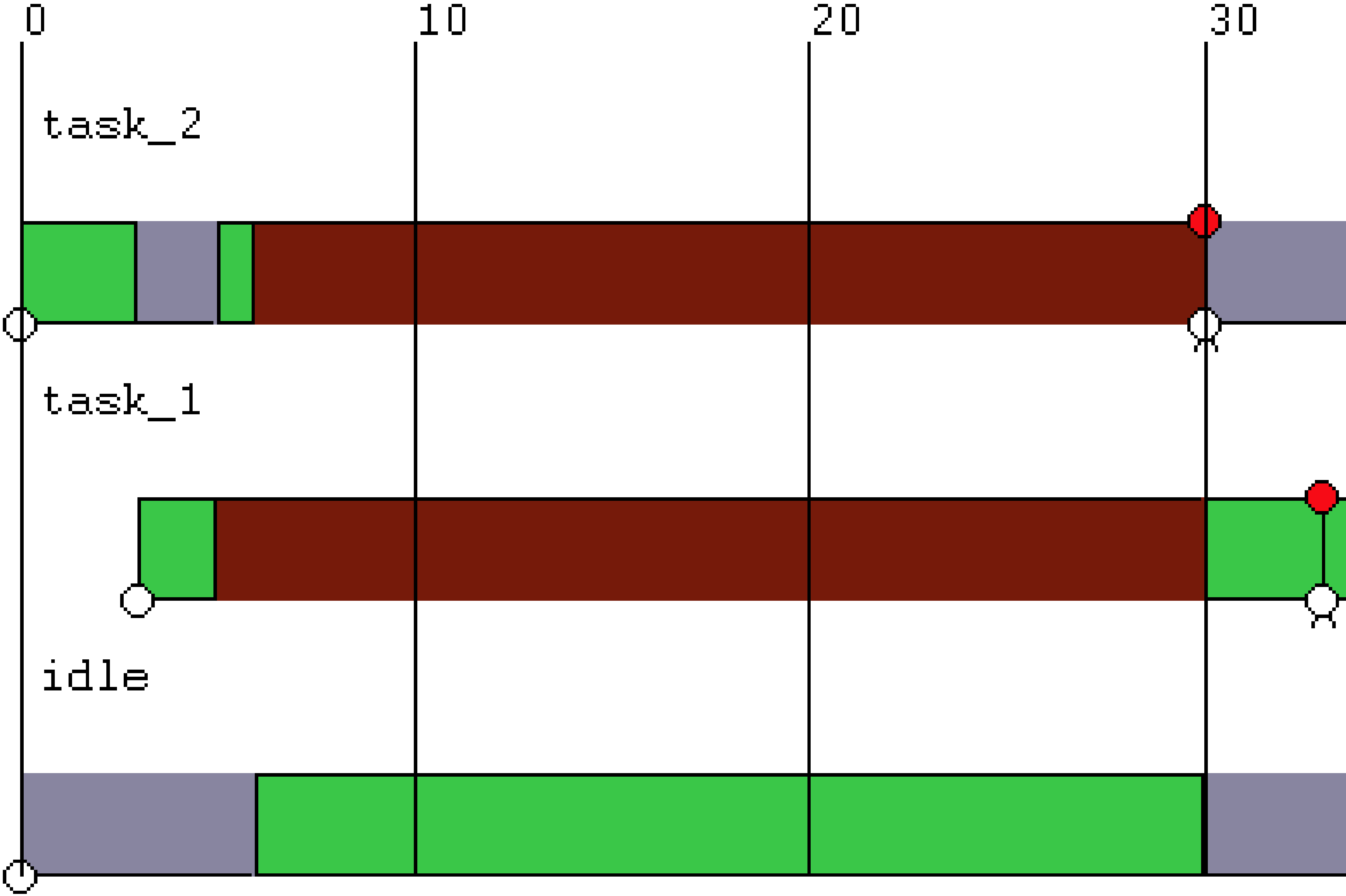
- The Basic **Priority Inheritance Protocol** has two problems:
  - **Deadlock** - two tasks need to access a pair of shared resources simultaneously. If the resources, say A and B, are accessed in opposite orders by each task, then deadlock may occur.
  - **Blocking Chain** - the blocking duration is bounded (by at most the sum of critical section times), but that may be substantial.

```
periodic task_1
  period        30   deadline 30    offset 3
  priority 1
  inheritance::respop(sem1)
  pop(sem1)
  [2,2]
  inheritance::respop(sem2)
  pop(sem2)
  [2,2]
  inheritance::resvop(sem2)
  vop(sem2)
  [1,1]
  inheritance::resvop(sem1)
  vop(sem1)
  [1,1]
endper

periodic task_2
  period        30   deadline 30    offset 0
  priority 2
  [2,2]
  inheritance::respop(sem2)
  pop(sem2)
  [2,2]
  inheritance::respop(sem1)
  pop(sem1)
  [2,2]
  inheritance::resvop(sem1)
  vop(sem1)
  [1,1]
  inheritance::resvop(sem2)
  vop(sem2)
  [1,1]
endper
```

Example: lec12b.str

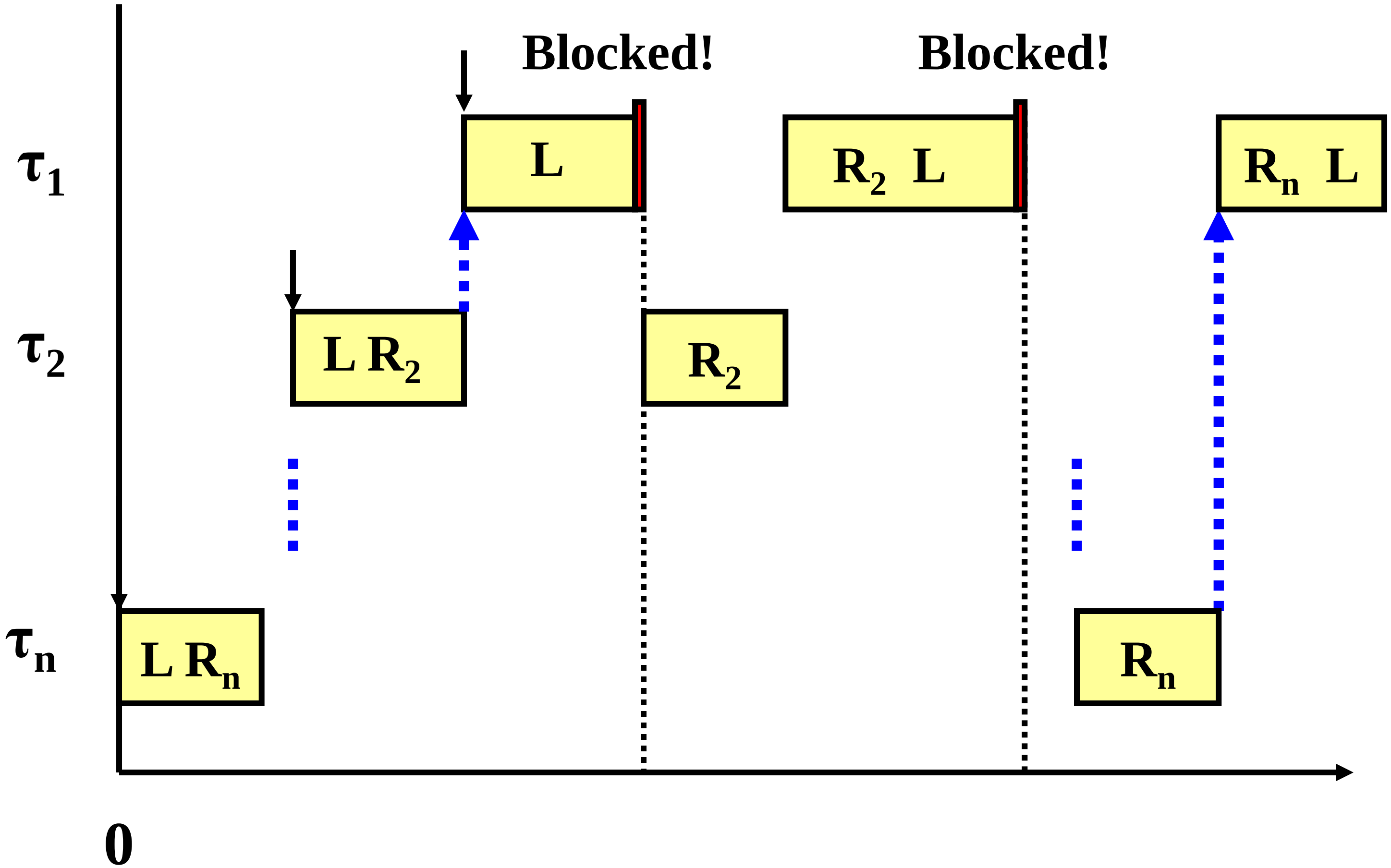
Deadlock!


0
10
20
30
task_2
task_1
idle

Blocking Chain Example

- Task τ_1 : L R_2 L R_3 L R_4 L ... L R_n L, $\varphi_1 = 2(n-1)$
- Task τ_2 : L R_2 R_2, $\varphi_2 = 2(n-2)$
- Task τ_3 : L R_3 R_3, $\varphi_3 = 2(n-3)$
- Task τ_4 : L R_4 R_4, $\varphi_4 = 2(n-4)$
- ...
- Task τ_{n-1} : L R_{n-1} R_{n-1}, $\varphi_{n-1} = 2$
- Task τ_n : L R_n R_n, $\varphi_n = 0$

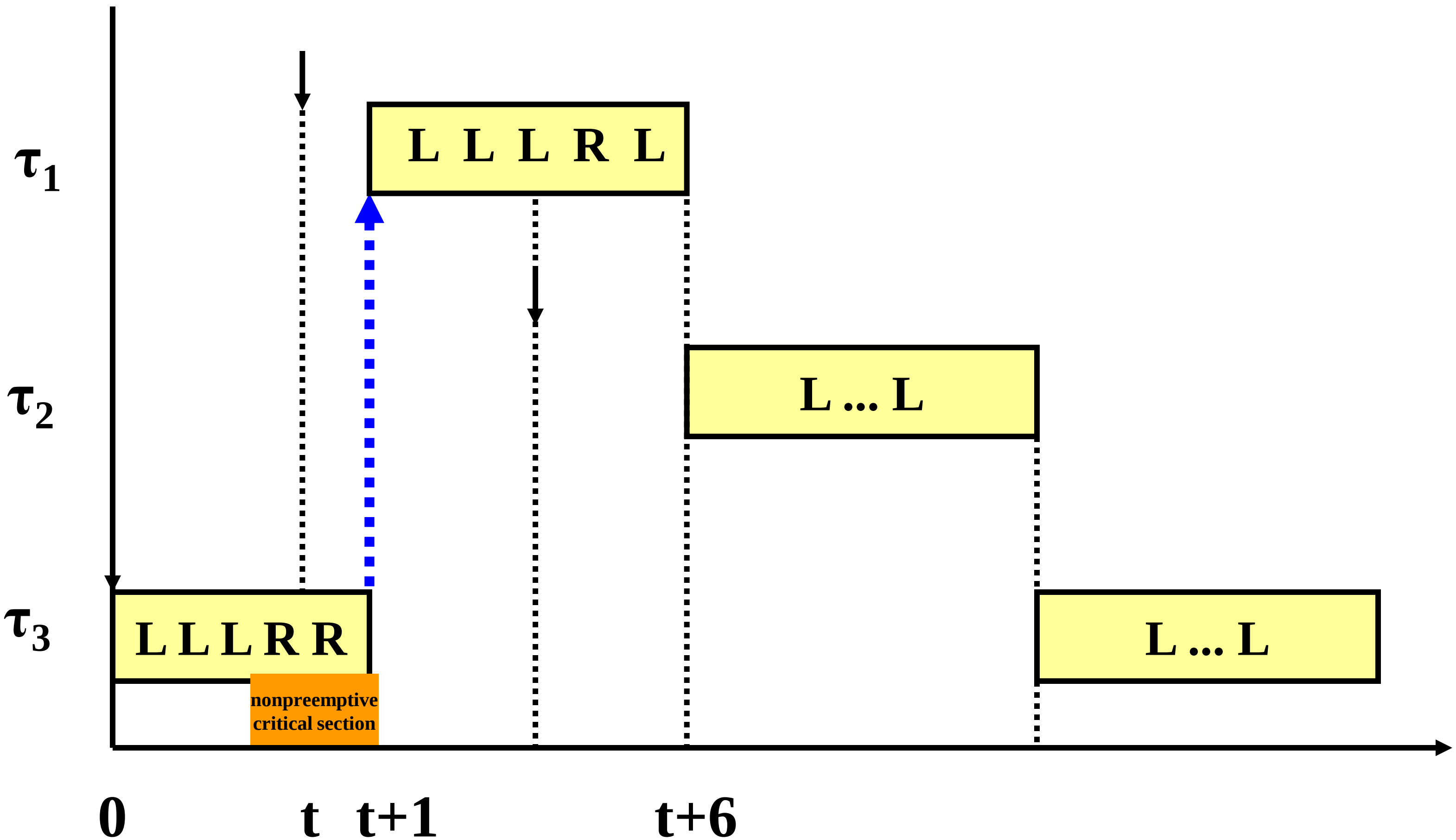
Priority Inheritance - Blocking Chain



NonPreemptive Critical Sections (NPCS)

- All critical sections are executed nonpreemptively.
- When a job requests a resource, it is always allocated the resource.
- When a job holds a resource (in a critical section), it executes at a priority higher than any other task.
- Since no job is ever preempted when it holds a resource, deadlock can never occur.
- The blocking time due to resource conflicts is the maximum critical section time over **all** lower priority tasks.

NonPreemptive Critical Sections



Advantages and Disadvantages

- **Advantages:**
  - Simplicity
  - Use with fixed-priority and dynamic-priority systems
- **Disadvantages:**
  - Every task can be blocked by every lower priority task, even when there is no resource sharing between the tasks.
- **Idea behind Priority Ceiling Protocols:** Only allow blocking when tasks share resources.

Priority Ceiling Protocols (PCP)

- A higher priority task can be blocked **at most once** during each job by a lower priority task.
- Deadlocks are prevented.
- Transitive blocking is prevented.
- Mutually exclusive access to shared resources is supported.

Semaphore Requirements

- Tasks cannot hold locks on a semaphore between invocations of a job.
- Tasks must lock and unlock semaphores in a “nested” or “pyramid” fashion:
  - Let P(S) = L(S) = lock(S) = sem_wait(S).
  - Let V(S) = U(S) = unlock(S) = sem_signal(S).
  - Example: $P(s_1);P(s_2);P(s_3);...;V(s_3);V(s_2);V(s_1);$

Original PCP

- The protocol uses the notion of a **system-wide semaphore ceiling priority.**
- Each task has a **static default priority** assigned.
- Each resource (semaphore) has a **static ceiling priority** defined to be the maximum static priority of any task that uses it.
- Each task has a **dynamic priority** equal to the maximum of its own default priority and any priority it inherits due to blocking a higher priority task.

Original PCP

- At run-time, if a task wants to lock a semaphore s, its priority must be strictly higher than the ceilings of all semaphores currently locked by other tasks (unless it is the task holding the lock on the semaphore with the highest ceiling).
- If this condition is not satisfied, then the task is blocked on s.
- When a task is blocked on semaphore s, the task currently holding s inherits the priority of the blocked task.

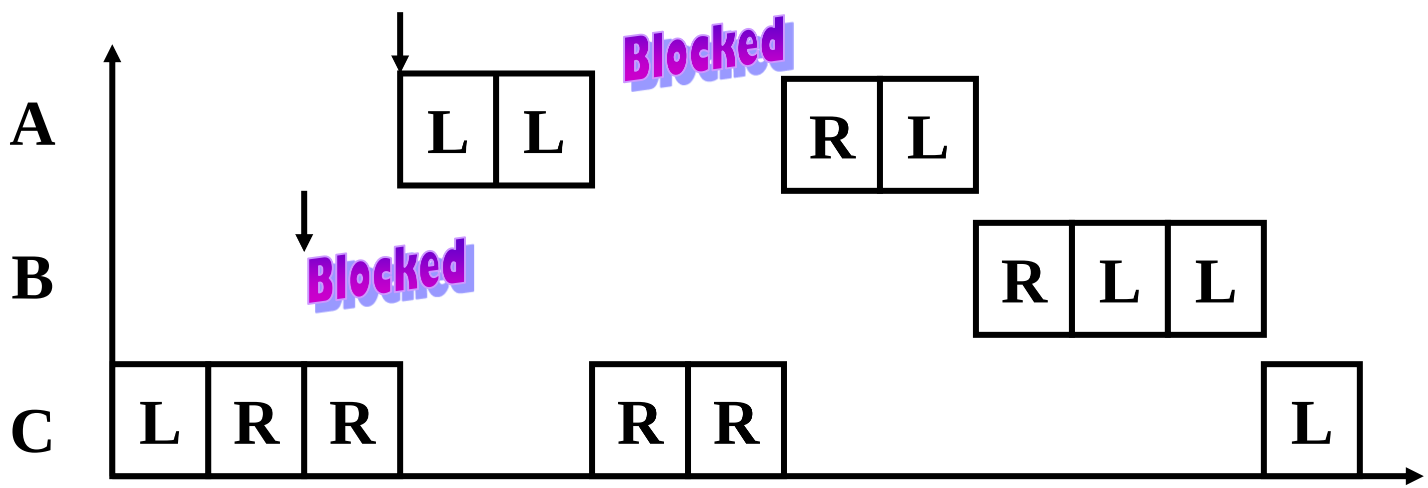
Notes

- A task can lock a semaphore only if its dynamic (effective) priority is higher than the ceiling of **any** resource currently locked by another task (system-wide max).
- A task’s dynamic priority can only **increase** if blocking will occur.
- The main difference between basic priority inheritance and basic priority ceiling protocols is the basic priority inheritance protocol allows a job to lock a resource whenever the resource is free, but the PCP protocols may not.

Example #1 – Original PCP

Task A: LLRL, priority = 3 (high), arrival time = 3
Task B: RLL, priority = 2 (medium), arrival time = 2
TaskC: LRRRRL, priority = 1 (low), arrival time = 0

L = local computation, R = access shared resource



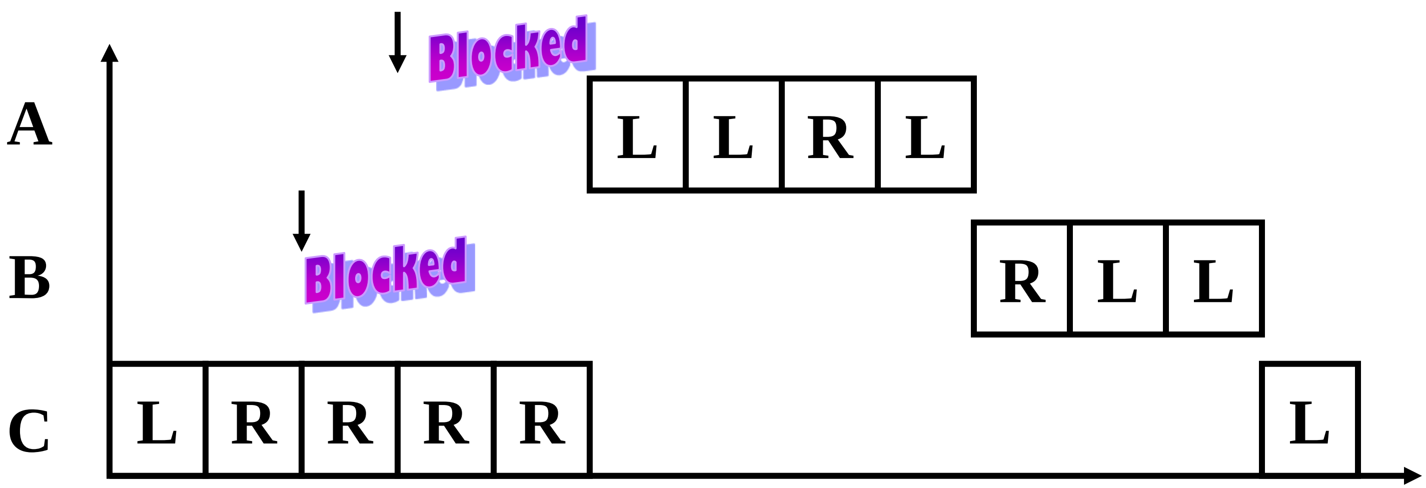
Immediate PCP (IPCP) (Ceiling-Priority Protocol)

- Each task has a static default priority assigned.
- Each resource has a static ceiling value defined.
- Each task has a dynamic priority that is the maximum of its own static priority and the ceiling value of **any resource it has locked**.
- Blocking occurs **prior to the initial execution** of a task during each invocation. Tasks should not voluntarily suspend themselves while holding a resource because no job is ever blocked once its execution begins.

Example #2 (IPCP)

Task A: LLRL, priority = 3 (high), arrival time = 3
Task B: RLL, priority = 2 (medium), arrival time = 2
TaskC: LRRRRL, priority = 1 (low), arrival time = 0

L = local computation, R = access shared resource



Review: Priority Ceiling Protocols

- Each task has a static default priority.
- Each resource has a static ceiling priority equal to the maximum priority of all tasks that use it.
- A task has a dynamic priority equal to the maximum of its own default priority and any priority it inherits due to **blocking** a higher priority task. The difference is **when does** the dynamic priority change:
  - immediately (before execution) (IPCP), or
  - when blocking occurs (OPCP).

PCP (continued)

- A task can lock a resource only if its dynamic priority is higher than the ceiling of any currently locked resource (system-wide priority ceiling).
- IPCP requires fewer context switches. It is called the **Priority Protect Protocol** in POSIX.

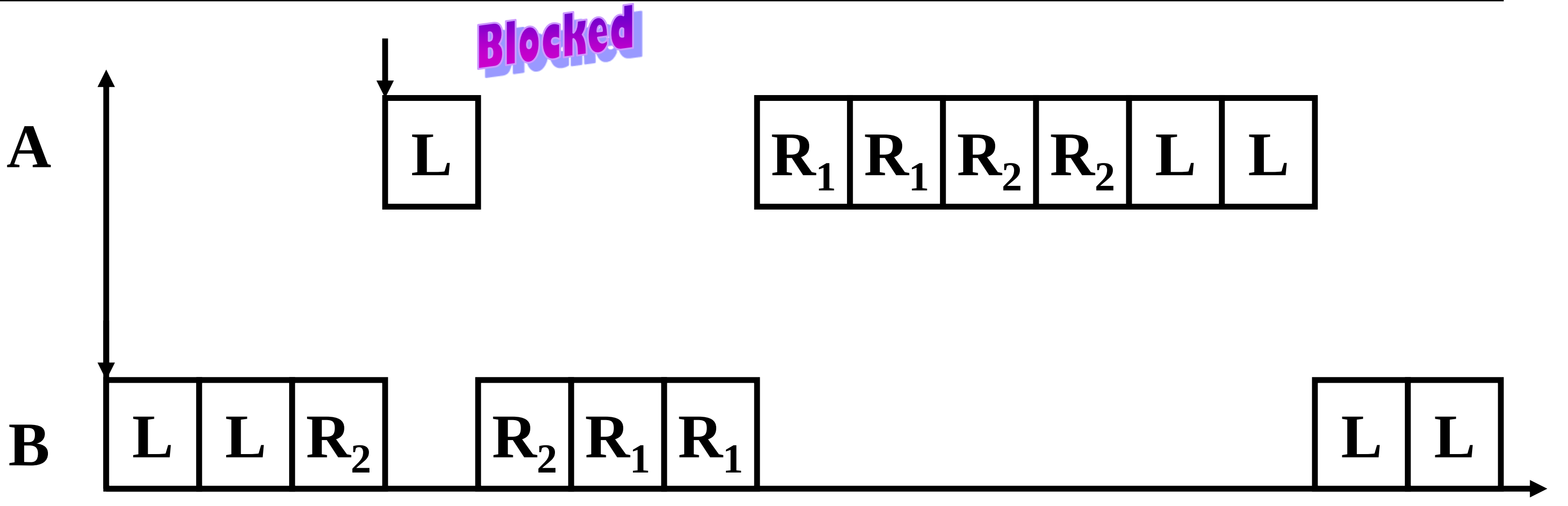
Example #3 and #4

- **Task A: $LR_1R_1R_2R_2LL$, priority = high, arr. time = 3**
  - **local computation (L)**
  - **sem_wait (S_1)**
  - **Access resource R_1**
  - **sem_wait (S_2)**
  - **Access resource R_2**
  - **sem_signal (S_2)**
  - **sem_signal (S_1)**
  - **local computation (LL)**
- **Task B: $LLR_2R_2R_1R_1LL$, priority = low, arr. time = 0**

Example #3: OPCP

Task A: $LR_1R_1R_2R_2LL$, priority = high, arr. time = 3
Task B: $LLR_2R_2R_1R_1LL$, priority = low, arr. time = 0

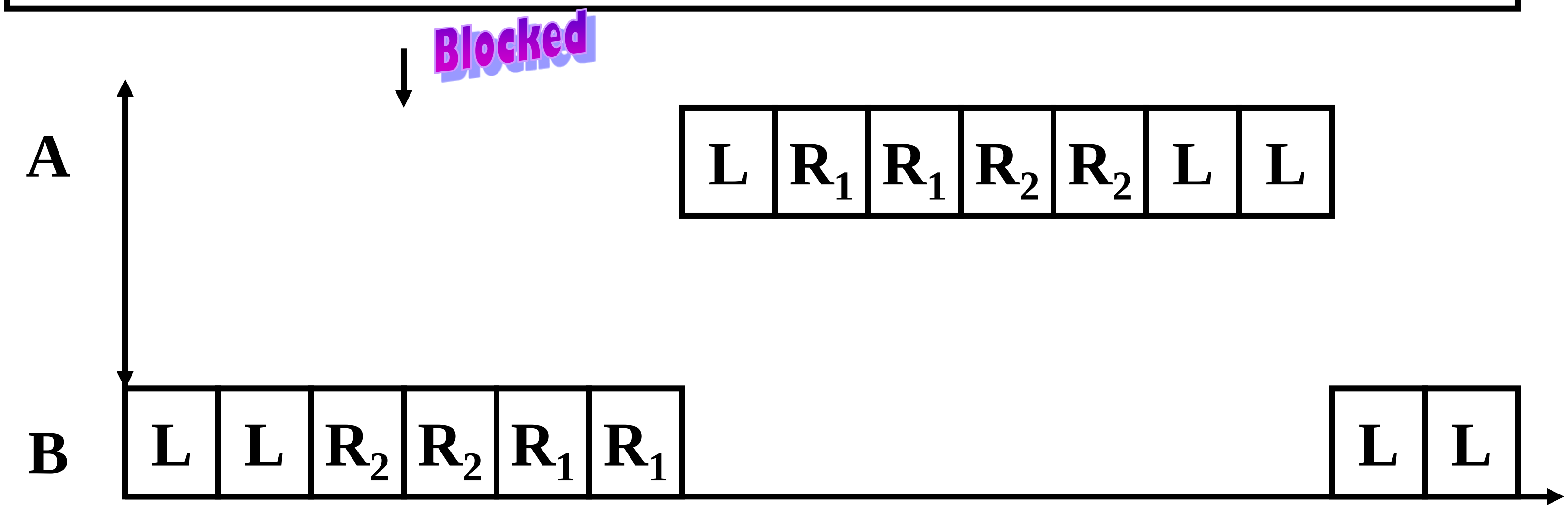
L = local computation, R_i = access shared resource i



Example #4: IPCP

Task A: $LR_1R_1R_2R_2LL$, priority = high, arr. time = 3
Task B: $LLR_2R_2R_1R_1LL$, priority = low, arr. time = 0

L = local computation, R_i = access shared resource i



Summary

- Read Ch. 8.
- Read Sha's paper on priority inheritance protocols.