RMA Generalizations

Deadline Monotonic Algorithm (DMA)

- When task deadline and periods are different, i.e., $d_i \neq p_i$:
  - RMA is not an optimal scheduling algorithm.
  - DMA is optimal for such task sets
- **Essence of DMA:**
  - *Assign priorities based on task deadlines.*

Deadline Monotonic Algorithm (DMA)

- When do RMA and DMA produce identical schedules?
  - Relative deadline of every task in a task set is the same as its period.
- For arbitrary relative deadlines:
  - DMA may produce a feasible schedule even when RMA fails.
  - On the other hand, RMA will always fail if DMA fails.

Exercise

- Check for schedulability of following tasks under RMA and DMA.
  - $T_1 = (e_1 = 10\ ms,\ p_1 = 50\ ms,\ d_1 = 35\ ms)$
  - $T_2 = (e_2 = 15\ ms,\ p_2 = 100\ ms,\ d_2 = 20ms)$
  - $T_3 = (e_3 = 20\ ms,\ p_3 = 200\ ms, d_3 = 200ms)$

Solution

- RMA: Checking Liu-Layland criterion

$$\sum_{i=1}^{n} \frac{e_i}{p_i} = \sum u_i = \frac{10}{35} + \frac{15}{20} + \frac{20}{200} = \frac{1590}{1400} > 1$$

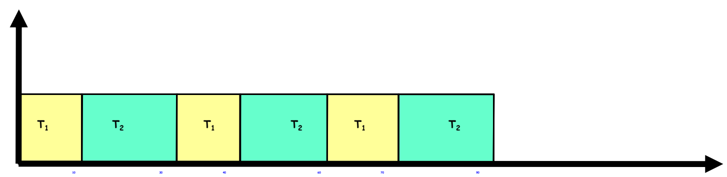
  - The task set is unschedulable
- DMA completion time check:
  - T2 meets its first deadline: 15 < 20
  - T1 meets its first deadline: 15+20=35 <=35
  - T3 meets its first deadline: 5*2+10*4+20=90<200
  - The task set is schedulable.

Overhead Due to Context Switching

- Context switching of tasks consumes some time:
  - So far we neglected the overhead due to context switching.
- When a task arrives:
  - It preempts the currently running lower priority task.
  - There may be no preemption if the CPU was idle or a higher priority task was running.

Overhead Due to Context Switching

- In the worst case, each task incurs **at most two** context switches:
  1. When it runs possibly preempting the currently running task.
  2. When it completes.



- Let the context switching time be constant and equal c ms.
- **Effectively, the execution time of each task increases to (e_i + 2*c)**

Example

- Assume 3 periodic tasks:
  - T1: e1=10mSec, p1= d1= 50mSec
  - T2: e2=25msec, p2= d2= 150mSec
  - T3: e3=50mSec, p3= d3= 200mSec
  - Assume context switching time = 1msec
  - Determine whether the task set is schedulable.

Solution

T1: e1=10mSec, p1= d1= 50mS

T2: e2=25msec, p2=d2= 150msec

T3: e3=50mSec, p3= d3= 200mSec

- Effect of context switch:
  - Execution time of each task increases at most by 2 msec.
  - **Task T1:** 12msec <50 mSec >>Schedulable
  - **Task T2:** 27+ 12*3=63<150 msec >>Schedulable
  - **Task T3:** 52+ 12*4 + 27*2 =154<200msec >>Schedulable

Practice Problem

- Check for schedulability of following tasks under RMA.
  - $T_1 = (e_1 = 10$ ms, $d_1 = p_1 = 50$ ms)
  - $T_2 = (e_2 = 5$ ms, $d_2 = p_2 = 20$ ms)
  - $T_3 = (e_3 = 9$ ms, $d_3 = p_3 = 30$ ms)
  - Assume that context switch overhead is 1 ms.

Further RMA Generalizations

Handling Critical Tasks With Long Periods

- What if task criticalities turn out to be different from task priorities?
- Simply raising the priority of a critical task:
  - Will make the RMA schedulability check results inapplicable.
  - A solution was proposed by Sha and Raj Kumar's period transformation (1989).

Period Transformation Technique

- A critical task is logically divided into many small subtasks.
- Let Ti be a critical task that is split into k subtasks:
  - Each one has execution time ei/k and deadline di/k.
- This is done virtually at a conceptual level:
  - Rather than making any changes physically to the task itself.

Period Transformation: Example

- Consider 2 tasks:
  - **T1:** e1=5, p1=d1=20 msec
  - **T2:** e2=8, p1=d1=30 msec
- Assume that T2 is a critical task:
  - Should not miss deadline even under transient underload.
- T2a: e2a=4, p2=d2=15msec

Handling Aperiodic and Sporadic Tasks

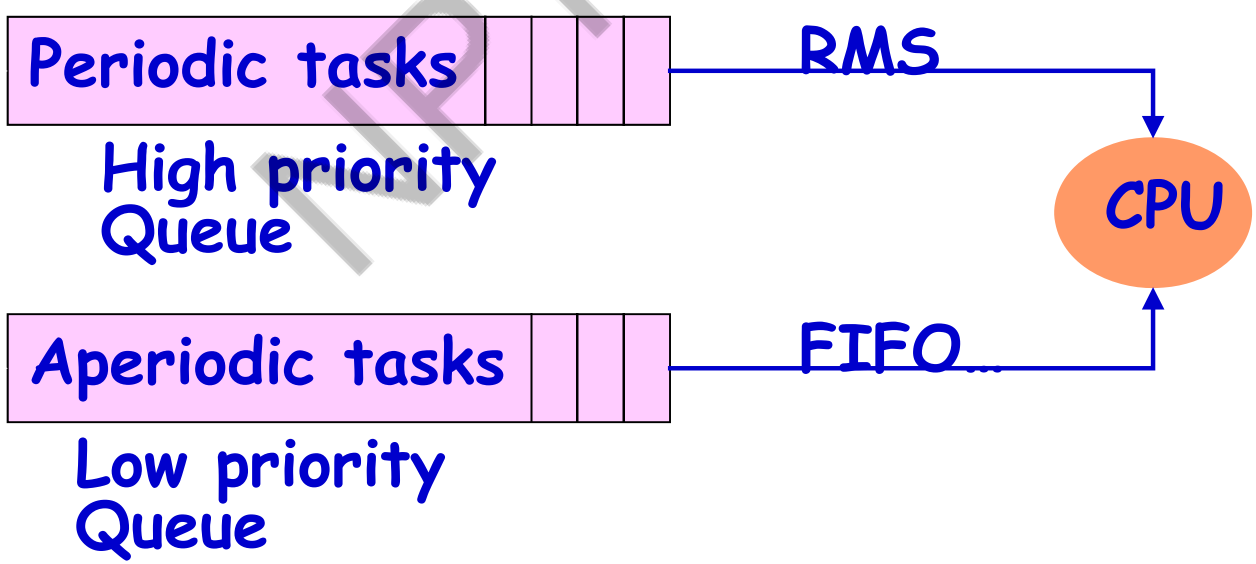
- It is difficult to assign high priority values to sporadic tasks:
  - A burst of sporadic task arrivals could overload the system:
    - Cause many tasks to miss deadlines.
    - Violate basic RMA premises
  - Low priorities can be accorded:
    - But some sporadic tasks might be critical.
  - **Periodic server** technique may be used.

Sporadic Tasks

- Two kinds of sporadic tasks:
  - **High priority:** Emergency events
  - **Non-critical:** Background jobs (logging)
- Background jobs can be deferred during transient overload:
  - Tolerate long response time anyway.
- High priority tasks:
  - ***An obvious way is to handle these is by converting them to periodic tasks.***

Simple Background Scheduling Technique

- Aperiodic tasks are executed:
  - When there is no periodic task to execute.
- Simple, but these tasks may face long delays



Periodic Server

- A periodic server is a:
  - High priority periodic task
  - Created to handle multiple sporadic tasks that have deadlines associated with them.
  - The period and execution time is decided based on the characteristics of the sporadic tasks.
- There can be multiple periodic servers in a system.

Types of Periodic Servers

- Various types of periodic servers have been proposed:
  - **Static Servers:**
    - Polling Server
    - Deferable Server
    - Priority Exchange
    - Sporadic server (POSIX-RT)
  - **Dynamic Servers**
- Slack stealer

Polling Server

- If there are no sporadic tasks at an invocation of the server (as per RMA):
  - The server suspends itself --- gets invoked again at its next period.
- If there are enough sporadic tasks at an invocation,
  - It serves up to e_s time.

Polling Server: Schedulability Analysis

- Include Ts in the task set and perform schedulability test
  - Of course, Schedulability of periodic tasks decreases
- Poorer response time for aperiodic tasks

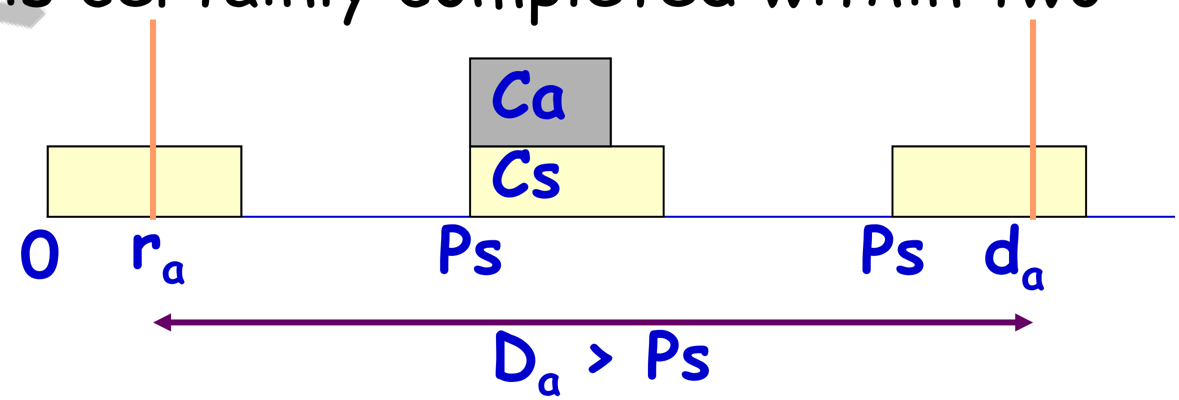
Polling Server: Schedulability Analysis

- Schedulability analysis involves
  - Schedulability of periodic tasks
  - Schedulability of sporadic tasks
- Schedulability analysis:
  - Introduce a periodic task corresponding to the server.

$$\Sigma_{i=1 \text{ to } n} (Ci / Pi) + (Cs / Ps) \leq (n+1)[2^{1/(n+1)} - 1]$$

Polling Server: Schedulability Analysis

- Sporadic task guarantees:
  - Sporadic task A_i, arrived at r_a, with computation time C_a and deadline D_a.
  - May have to wait for one period before receiving service,
  - if $C_a \leq Cs$ the request is certainly completed within two server periods.
  - $2P_s \leq D_a$



Resource Sharing Among Tasks

Introduction

- So far, the only shared resource among tasks that we considered was CPU.

- CPU is serially reusable. That is,
  - Can be used by one task at a time
  - **A task can be preempted at any time without affecting its correctness.**

Non-premptable Resources and Critical Sections

- What are some examples of non-preemptable resources?
  - Files, data structures, devices, etc.

What is a critical section?

- A piece of code in which a shared non-preemptable resource is accessed:
  - Called a **critical section** in the operating systems literature.

Critical Section Execution

- What is the traditional operating system solution to execute critical sections?
  - **Semaphores.**
- However, this solution does not work well in real-time systems --- causes:
  - **Priority inversion**
  - **Unbounded priority inversion**

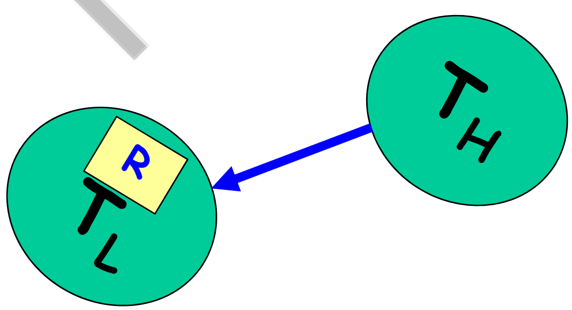
Priority Inversion

- A task instance executing its critical section:
  - Cannot be preempted.

- **Consqeuence:** A higher priority task keeps waiting:
  - While the lower priority task progresses with its computations.

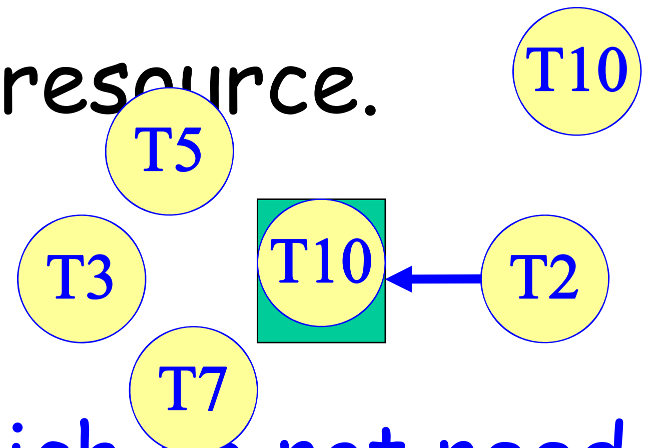
Priority Inversion

- When a resource needs to be shared in the exclusive mode.
  - A task may be blocked by a lower priority task which is holding the resource.

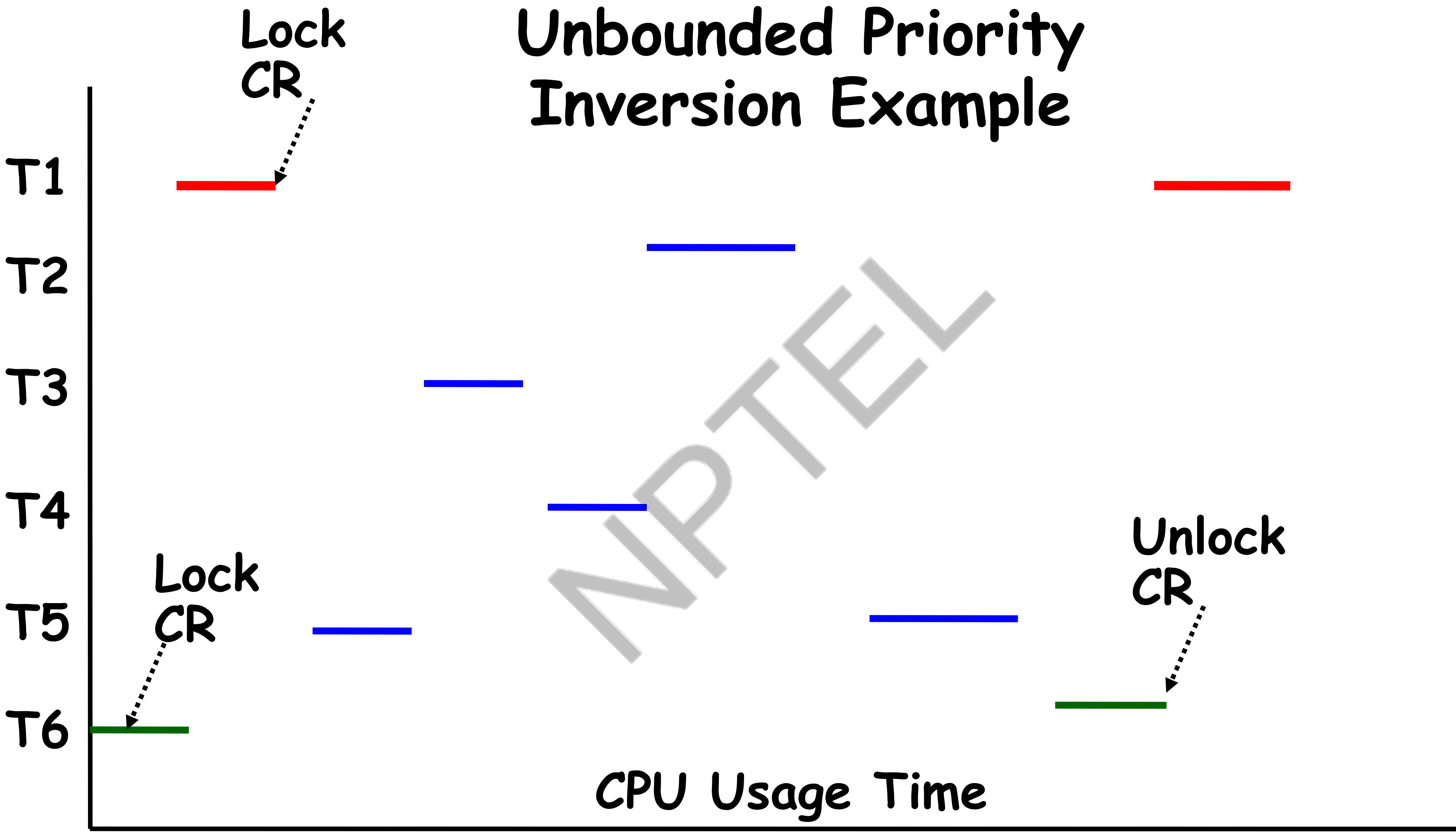


Unbounded Priority Inversion

- Consider the following situation:
  - A low priority task is holding a resource.
  - A high priority task is waiting
  - Intermediate priority tasks which do not need the resource repeatedly preempt the low priority task from CPU usage.



Unbounded Priority Inversion Example



Unbounded Priority Inversion

- Number of priority inversions suffered by a high priority task:
  - Can become unbounded:
  - A high priority task can miss its deadline.
  - In the worst case:
    - The high priority task might have to wait indefinitely.

Unbounded Priority Inversions

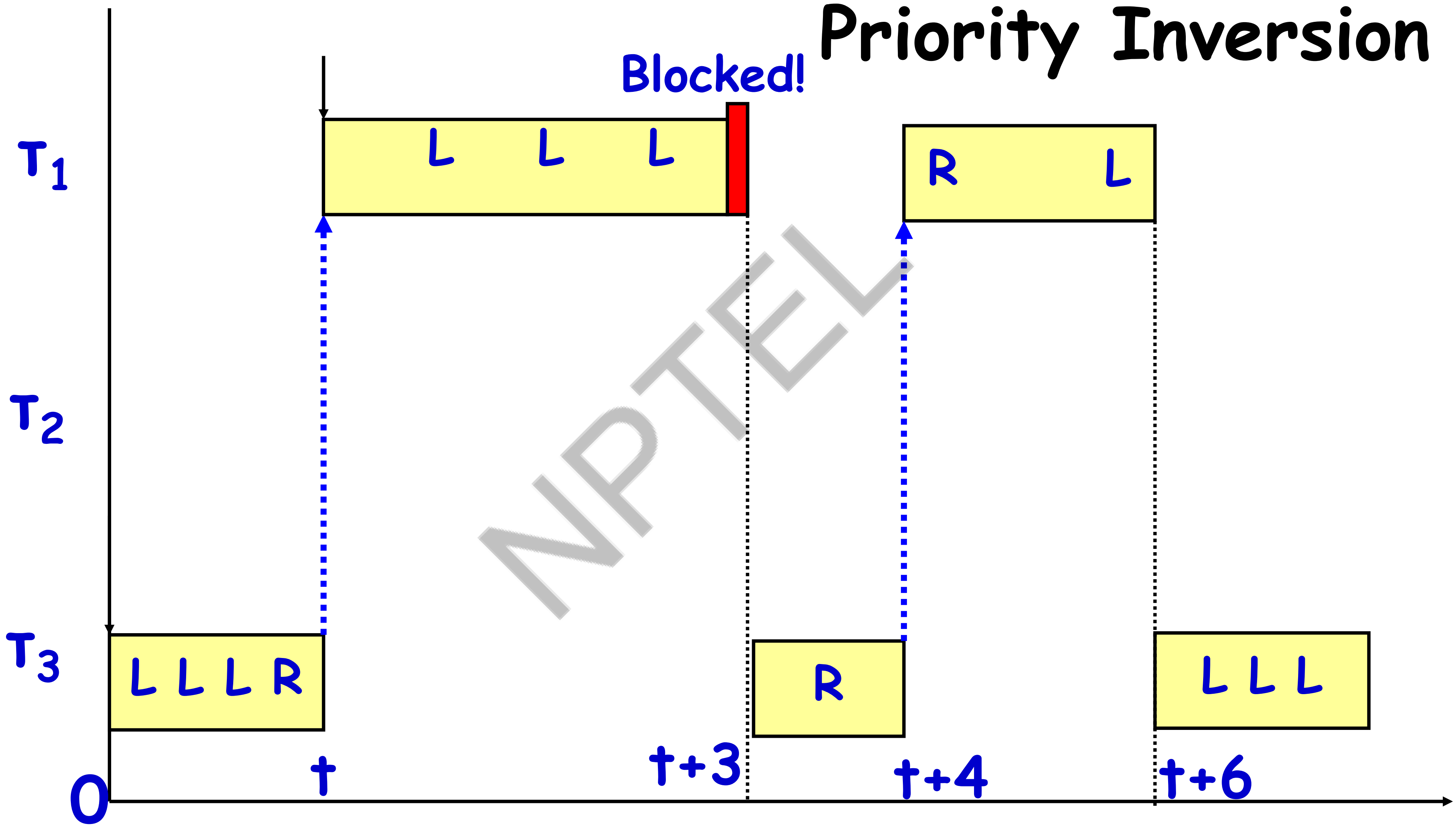
- Unless handled properly:
  - Priority inversions by a critical task can be too many causing it to miss its deadline.
- Most celebrated example:
  - Mars path finder

Example

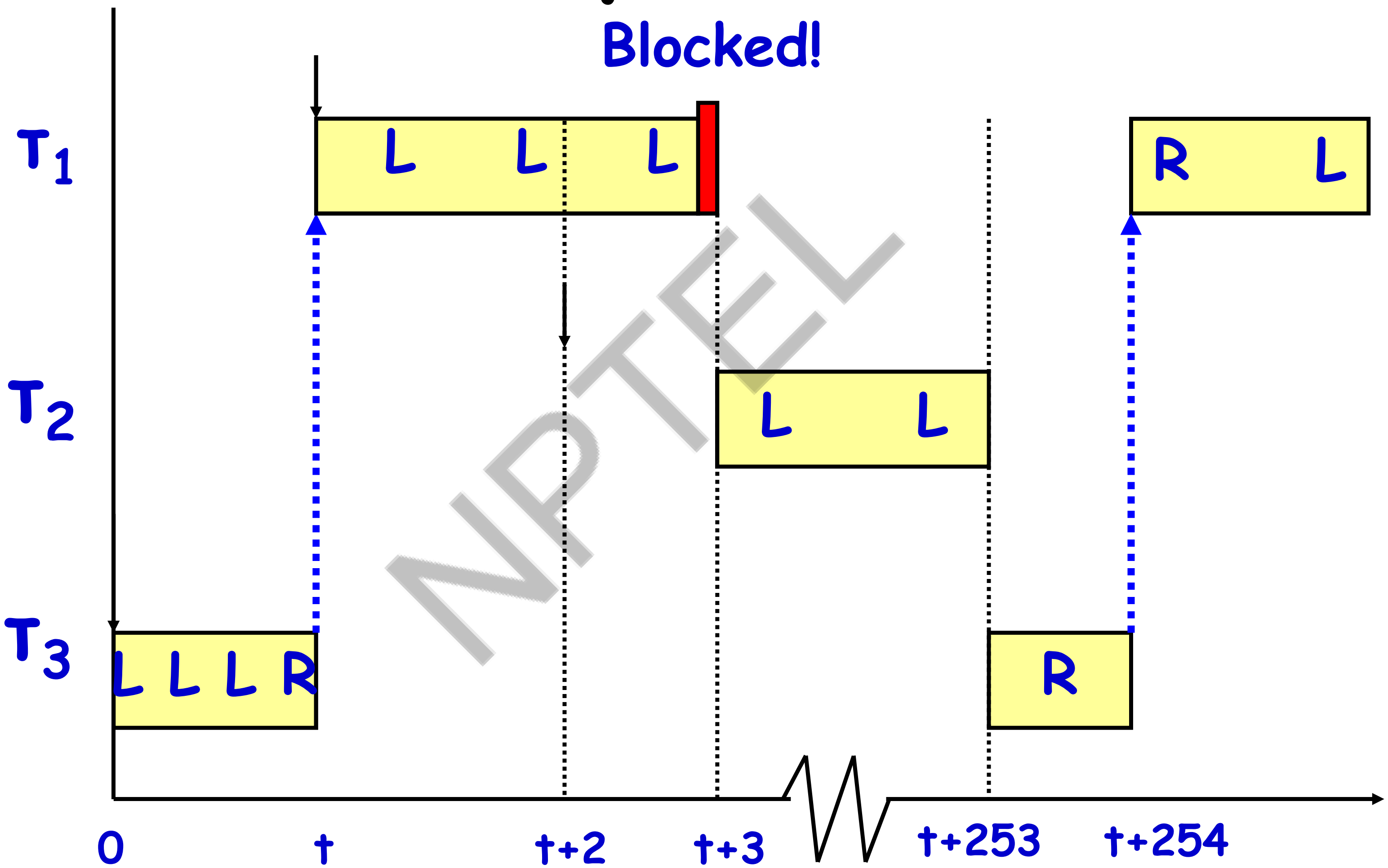
- Suppose that tasks τ_1 and τ_3 share some data in exclusive mode.
- Access to the data is restricted using *semaphore* x:
  - **Each task executes the following code:**
    - **do local processing (L)**
    - **P(x) //sem_wait**
      - **access shared resource (R)**
    - **V(x) //sem_signal**
    - **do more local processing (L)**

critical section

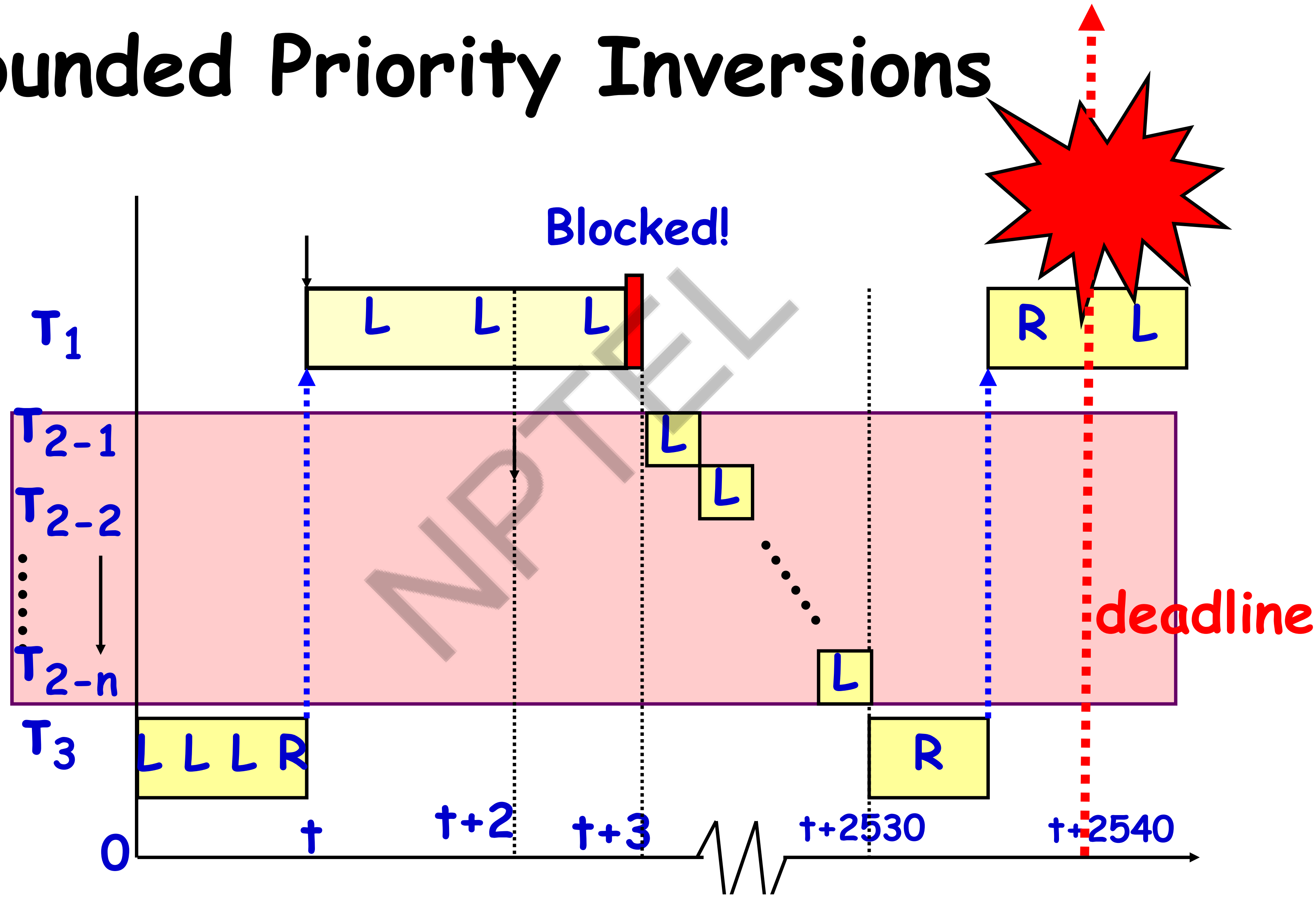
Priority Inversion



Two Priority Inversions



Unbounded Priority Inversions



Mars Pathfinder

- Landed on the Mars surface on July 4th, 1997.
  - Bounced onto the Martian surface surrounded by airbags.

  - Deployed the Sojourner rover.
  - Gathered and transmitted voluminous data back to Earth:
    - Included the panoramic pictures released by NASA and now available on the Web.

Mars Pathfinder: NASA

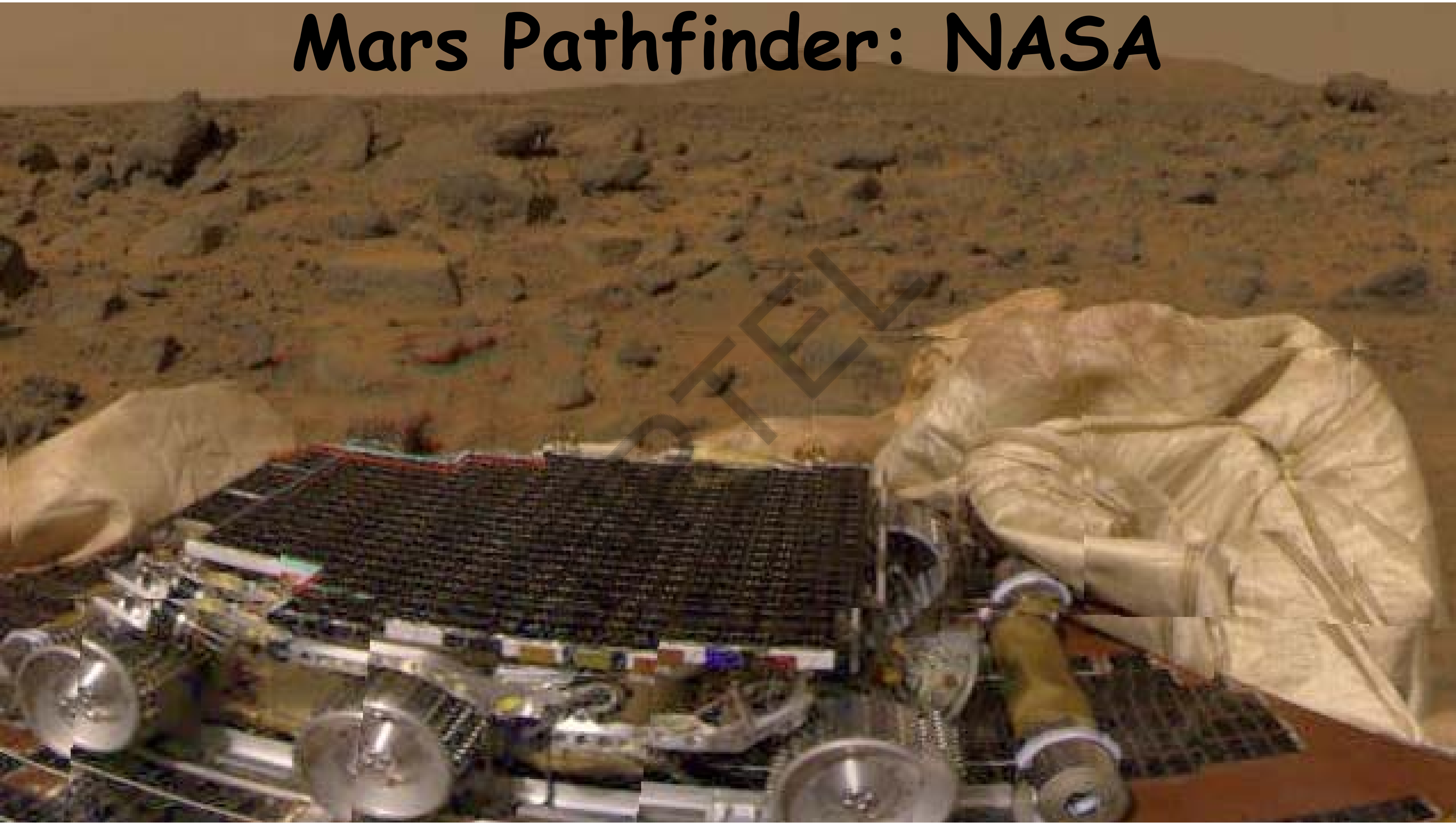



Image From Mars Pathfinder: NASA

Mars Pathfinder Bug

- Pathfinder began experiencing frequent system resets:
  - Each time resulted in loss of data.
- The newspapers reported these failures using terms such as:
  - Software glitches
  - The computer was trying to do too many things at once, etc.

Debugging Mars Pathfinder

- The real-time kernel used was VxWorks (Wind River Systems Ltd.)
  - RMA scheduling of threads was used
- Pathfinder contained:
  - Information sharing through shared memory
  - Information shared among different spacecraft components.

Debugging Mars Pathfinder

- VxWorks can be run in trace mode:
  - Interesting system events: context switches, uses of synchronization objects, and interrupts are recorded.
- JPL engineers spent hours running exact spacecraft replica in lab:
  - Replicated the precise conditions under which the reset occurred.

Debugging Mars Pathfinder

- VxWorks mutex object:
  - A boolean parameter indicates whether priority inheritance should be performed.
- It became clear to the JPL engineers:
  - **Turning ON priority inheritance would prevent the resets.**
  - Initialization parameters were stored in global variables.
  - A short C program was uploaded to the spacecraft.

Solution for Simple Priority Inversion

- What is the longest duration for which a simple priority inversion can occur?
  - **Bounded by the duration for which a lower priority task needs to use the resource in exclusive mode.**

Solution to Simple Priority Inversion

- Can simple priority inversion be tolerated by careful programming?
  - Limit the time for which a task executes its critical section.
  - **A simple priority inversion can be taken care of by careful programming, but not unbounded priority inversion.**

Protocols for Resource Sharing Among Tasks

Basic Priority Inheritance Protocol

Sha and Rajkumar 1990

- Main idea behind this scheme:
  - Since a task in a critical section cannot be preempted.
  - **It should be allowed to complete as early as possible.**

Priority Inheritance Protocol

- How do you make a task complete as early as possible?
  - **Raise its priority, so that low priority tasks are not able to preempt it.**
- By how much should its priority be raised?
  - **Make its priority as much as that of the task it is blocking.**

Protocols for Resource Sharing Among Tasks

Basic Priority Inheritance Protocol

Sha and Rajkumar 1990

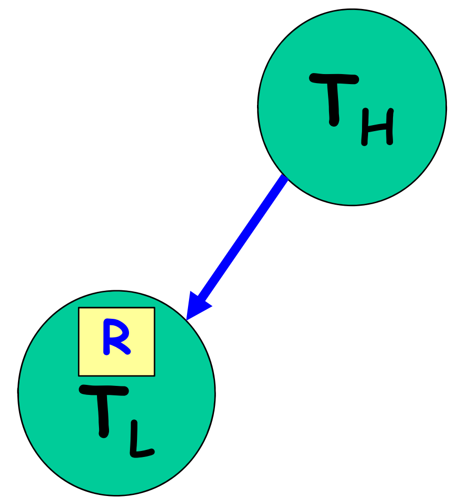
- Main idea behind this scheme:
  - Since a task in a critical section cannot be preempted.
  - **It should be allowed to complete as early as possible.**

Priority Inheritance Protocol

- How do you make a task complete as early as possible?
  - **Raise its priority, so that low priority tasks are not able to preempt it.**
- By how much should its priority be raised?
  - **Make its priority as much as that of the task it is blocking.**

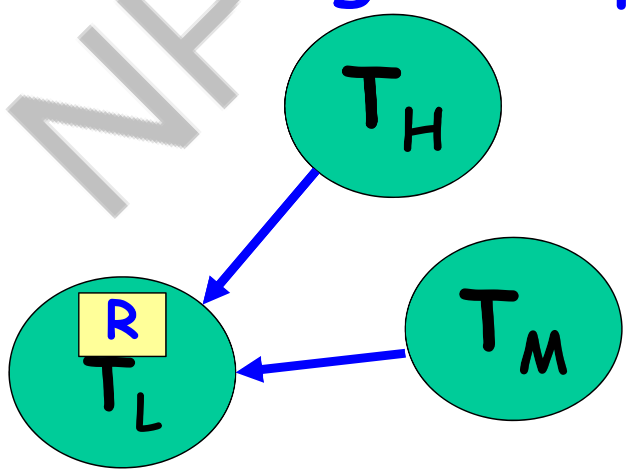
Priority Inheritance Protocol Sha and Rajkumar

- When a resource is already under use:
  - Requests to lock the resource by different tasks are queued in FIFO order.
  - Inheritance clause applied each time after a higher priority task blocks.



Inheritance Clause

- The priority of the task in the critical section:
  - Raised to equal the highest priority task in the queue.



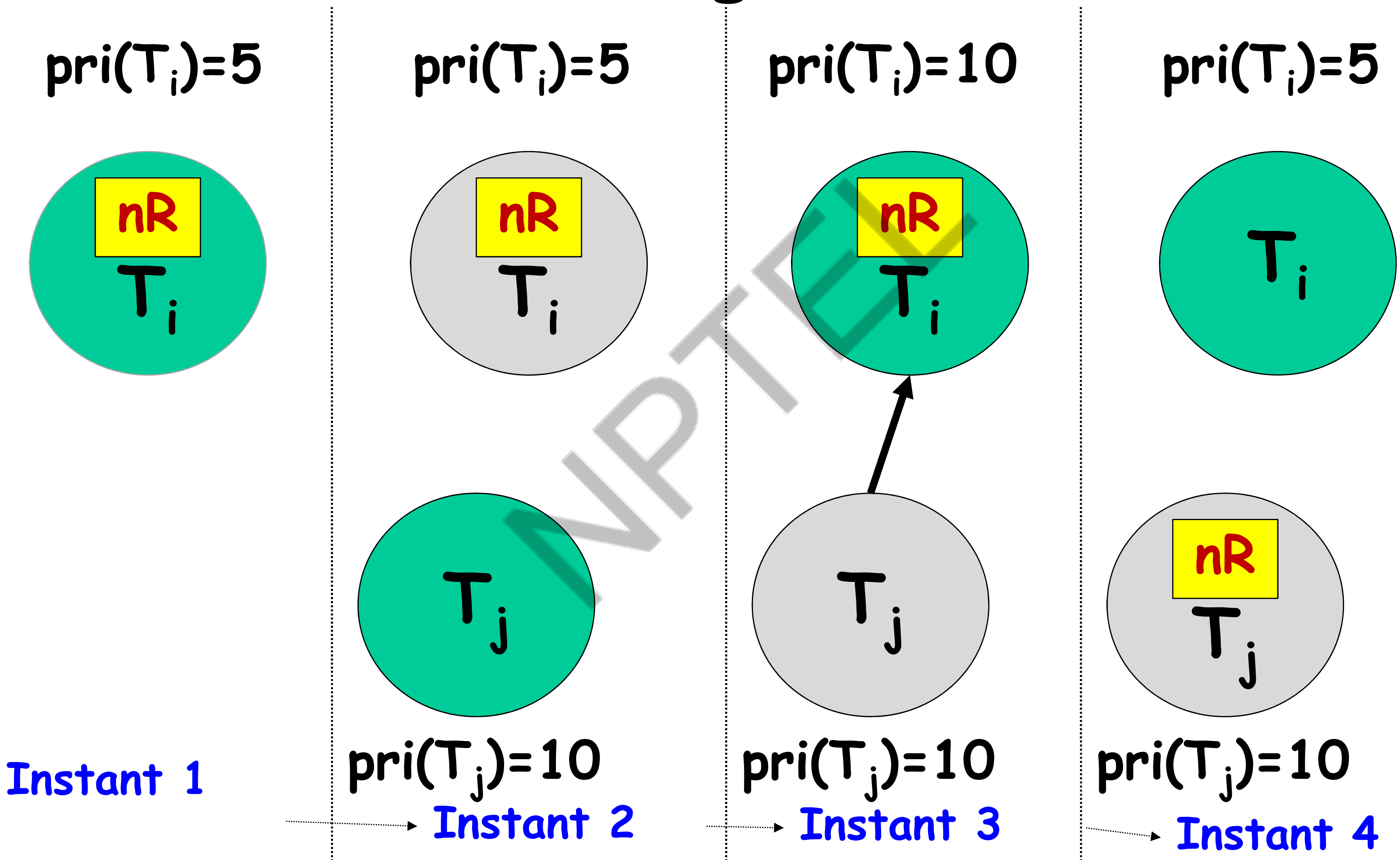
Priority Inheritance Protocol (PIP)
Sha and Rajkumar

- As soon as the task releases the resource,
  - It gets back its original priority value if it is holding no other critical resource.
- In case it is holding other critical resources:
  - It inherits priority of the highest priority task waiting for that resource.

Inheritance Blocking

- How does PIP prevent unbounded priority inversions?
- **Priority of low priority task raised to high value:**
  - **Intermediate priority tasks can no longer preempt it.**

Working of PIP



Shortcomings of the Basic Priority Inheritance Scheme

- PIP suffers from two important drawbacks:
  - Susceptible to chain blocking.
  - Does nothing to prevent deadlocks.

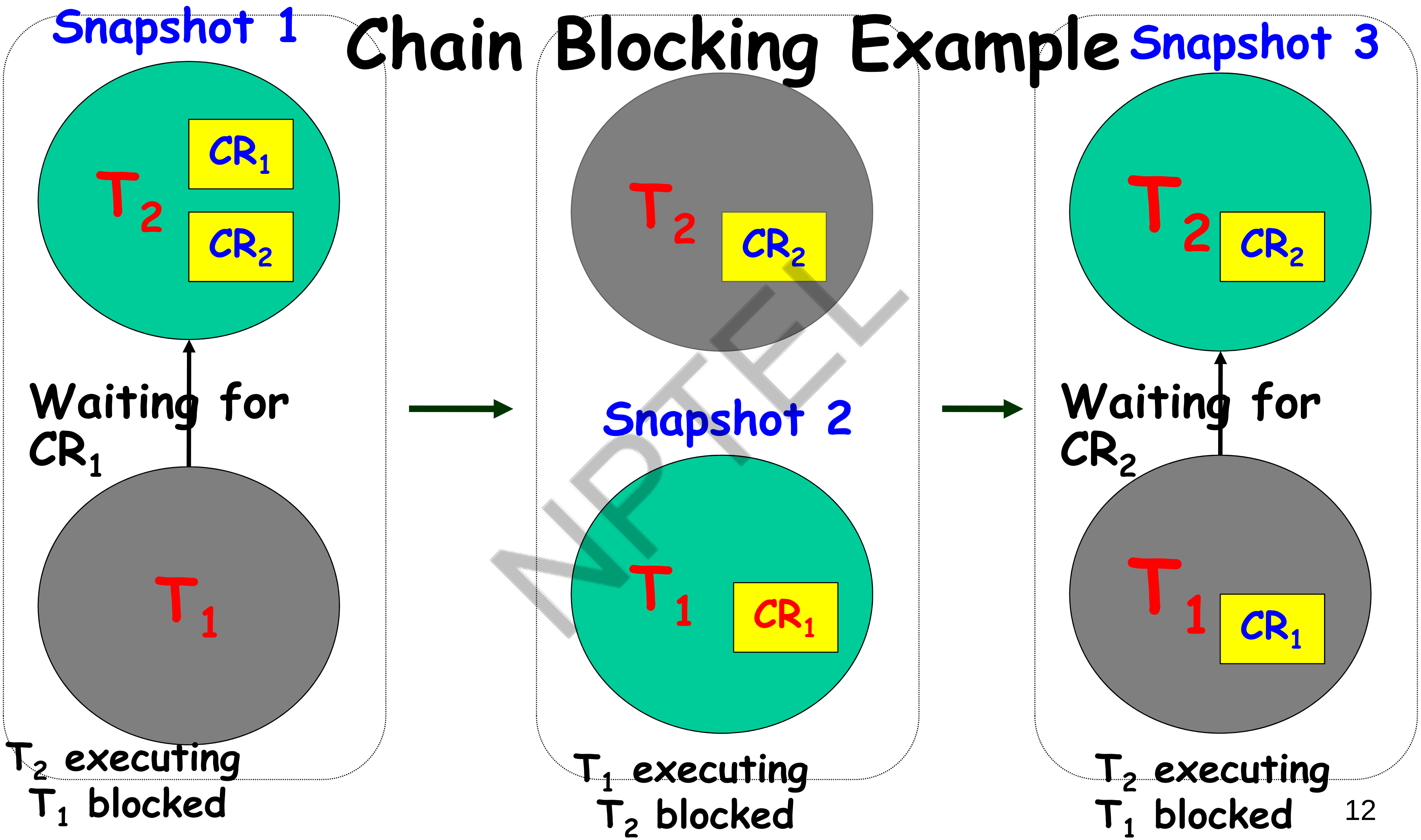
Deadlocks

- Consider two tasks T_1 and T_2 accessing critical resources CR_1 and CR_2.
- Assume:
    - T1 has a higher priority than T2
    - T2 starts running first
- T_1: Lock R_1, … Lock R_2, …Unlock R_2, Unlock R_1
- T_2: Lock R_2, … Lock R_1, …Unlock R_1, Unlock R_2

Chain Blocking

- A task needing to use a set of resources undergoes chain blocking, if :
  - **Each time it needs a resource, it undergoes priority inversion.**
- Example:
  - Assume a high-priority task T_1 needs several resources

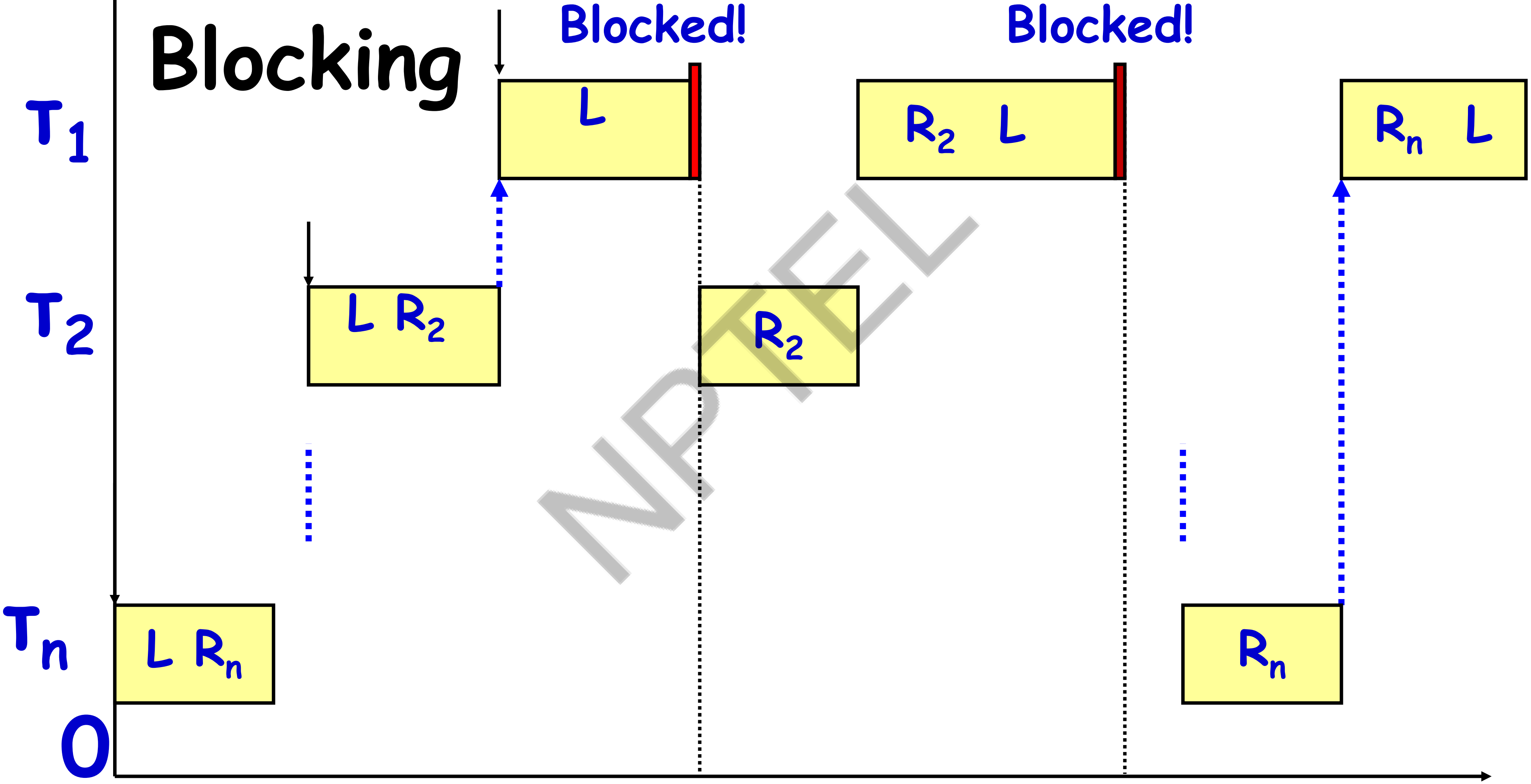
Chain Blocking Example



Chain Blocking Example

- Task τ_1 : L R_2 L R_3 L R_4 L ... L R_n L,
- Task τ_2 : L R_2 R_2,
- Task τ_3 : L R_3 R_3,
- Task τ_4 : L R_4 R_4,
- ...
- Task τ_{n-1} : L R_{n-1} R_{n-1},
- Task τ_n : L R_n R_n,

Chain Blocking

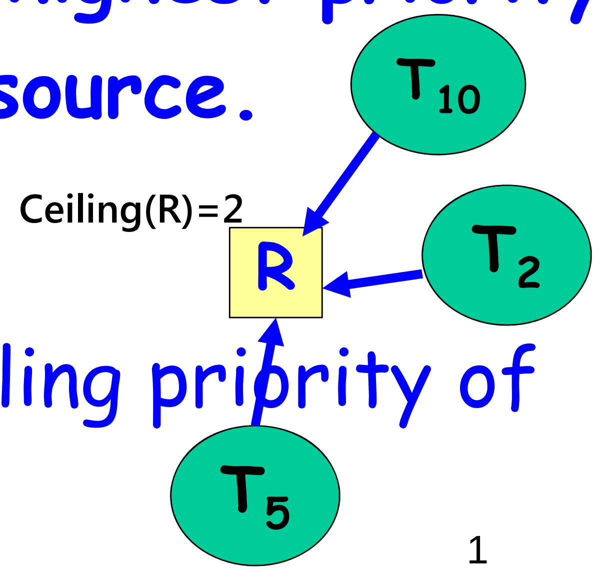


Properties of PIP

- **Theorem 1:**
  - If a task Ta can be blocked by k lower priority tasks T1…Tk due to a single resource usage,
    - Then the worst case duration for which it can block is max(ei); where ei is the critical section time of task Ti.
- **Theorem 2:**
  - If a task needs to use k critical resources:
    - The maximum duration for which it can block is ∑ max(ej) for each critical resource.
    - ei is the longest execution duration by a task for resource ri

Highest Locker Protocol

- During the design of a system
  - A ceiling priority value is assigned to all resources.
  - **The ceiling priority is equal to the highest priority of all tasks needing to use that resource.**
- When a task acquires a resource:
  - Its priority value is raised to the ceiling priority of that resource.

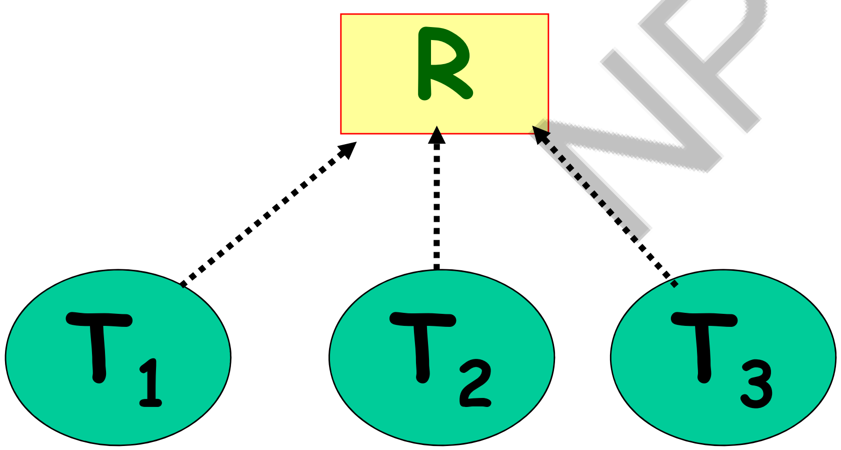


Highest Locker Protocol (HLP)

- Addresses the shortcomings of PIP:
  - However, introduces new complications.
  - Addressed by Priority Ceiling Protocol (PCP).
  - Easier to first understand working of HLP and then PCP.
- During the design of a system:
  - A ceiling priority value is assigned to all critical resources.
  - The ceiling priority is equal to the highest priority of all tasks using that resource.

Ceiling Priority of a Resource

- When a task acquires a resource:
  - Its priority value is raised to the ceiling priority of that resource.

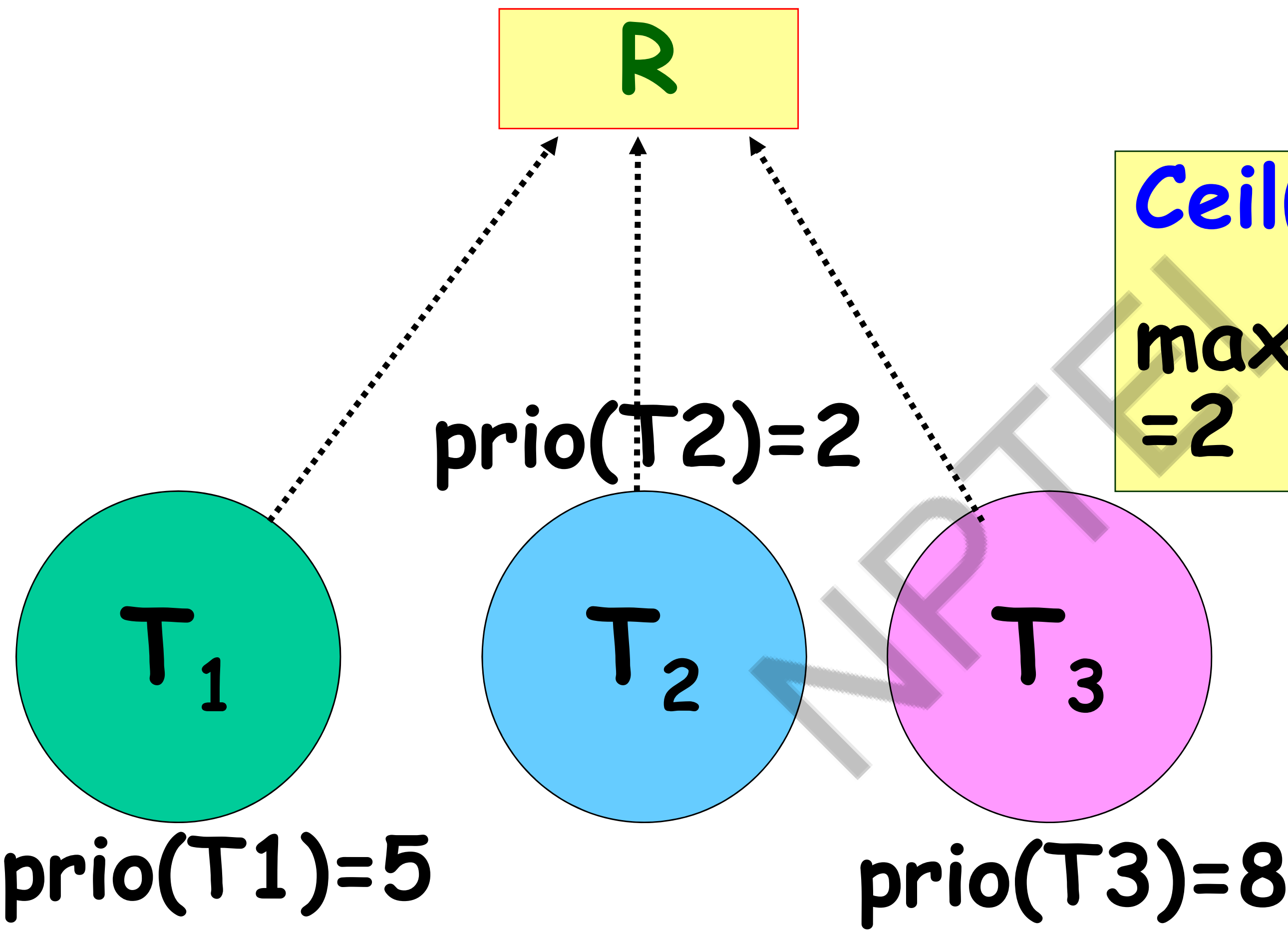


Ceil(R) =

max-prio(T1,T2,T3)

Example

$Ceil(R)$ = max-prio(T1,T2,T3) =2



Highest Locker Protocol (HLP)

- If higher priority values indicate higher priority (e.g., Microsoft Windows):

$$Ceil(R_i) = \max(\{pri(T_j) \mid T_j \; needs \; R_i\})$$

- If higher priority values indicate lower priority (e.g., Unix):

$$Ceil(R_i) = \min(\{pri(T_j) \mid T_j \; needs \; R_i\})$$

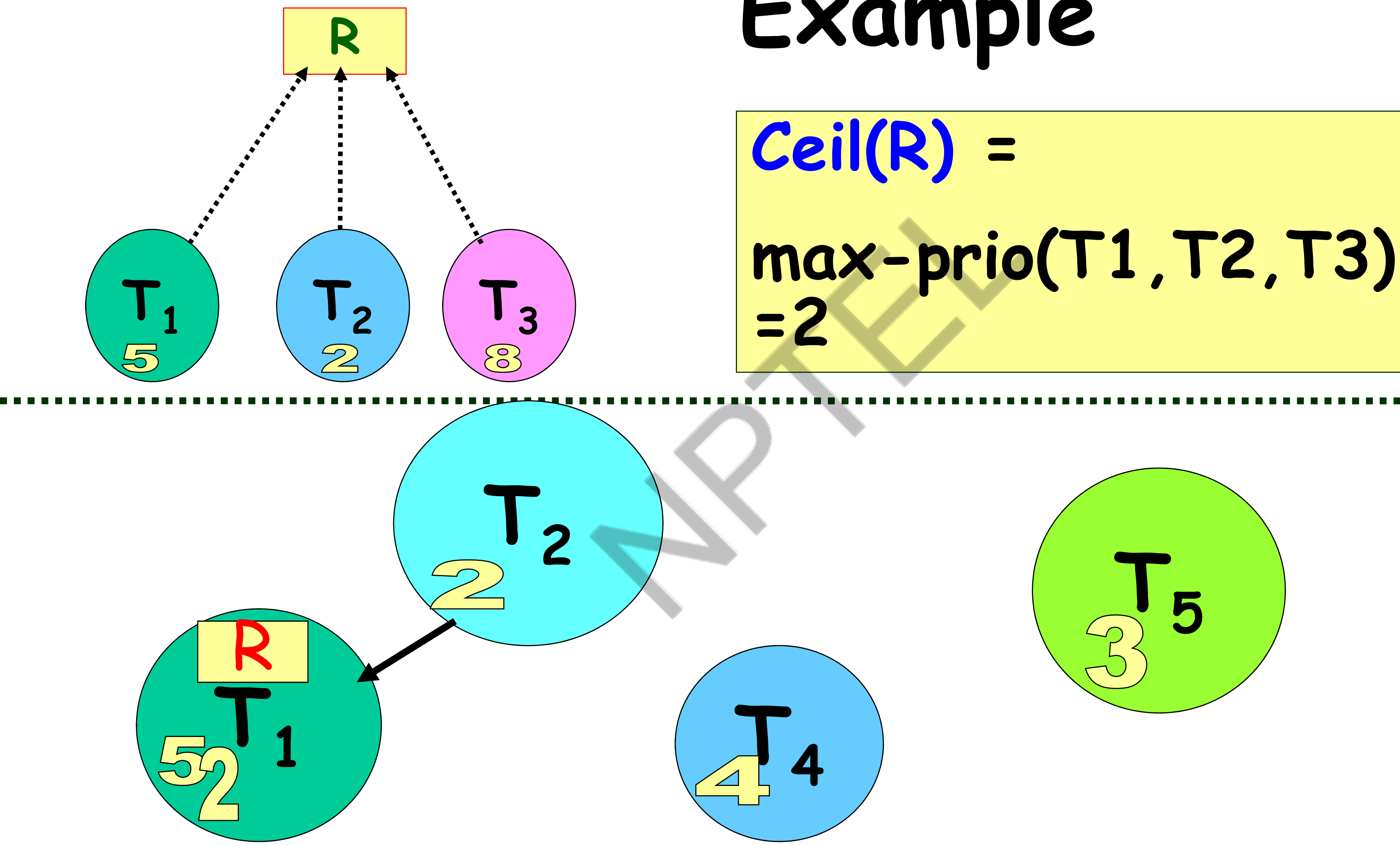
Highest Locker Protocol (HLP)

- As soon as a task acquires a resource R:
  - Its priority is raised to Ceil(R)
  - Helps eliminate the problems of:
    - Unbounded priority inversions,
    - Deadlock, and
    - Chain blocking.
- However, introduces inheritance blocking.

Example

Ceil(R) =

max-prio(T1,T2,T3)
=2



Highest Locker Protocol (HLP)

- **Theorem:**
  - When HLP is used for resource sharing:
    - Once a task gets any one of the resource required by it, it is not blocked any further.
- **Corollary 1:**
  - Under HLP, before a task is granted one resource:
    - All the resources required by it must be free.
- **Corollary 2:**
  - A task can not undergo chain blocking in HLP.

Highest Locker Protocol

- Prevents deadlock

  - **T1:** **lock R1, Lock R2, Unlock R2, Unlock R1**
  - **T2:** **lock R2, Lock R1, Unlock R1, Unlock R2**

- Prevents unbounded priority inversion.

Shortcomings of HLP

- Inheritance blocking occurs:
  - When the priority value of a low priority task holding a resource is raised to a high value.
  - Intermediate priority tasks not needing the resource:
    - Cannot execute and undergo priority inversion.

HLP: Blocking Time of a Task

- Let
  - Use(S) is the set of all tasks that use S
  - C(Tk,S) denote the computing time for the critical section for task Tk using resource S.
- The maximal blocking time B for task Ti is as follows:
  - $B = \max\{C(Tk,S) \mid Tk \text{ in } Use(S),\ pr(k) < pr(i)\}$

Usage of HLP

- POSIX supports priority locking for mutexes:
  - PTHREAD_PRIO_PROTECT mutex creation attribute:
  - example: pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_PROTECT);
- Linux does not support HLP
- The Ada programming language supports HLP:
  - Calls it "ceiling priority locking"
- Real-time Specification for Java (RTSJ) supports HLP:
  - Calls this "priority ceiling emulation"

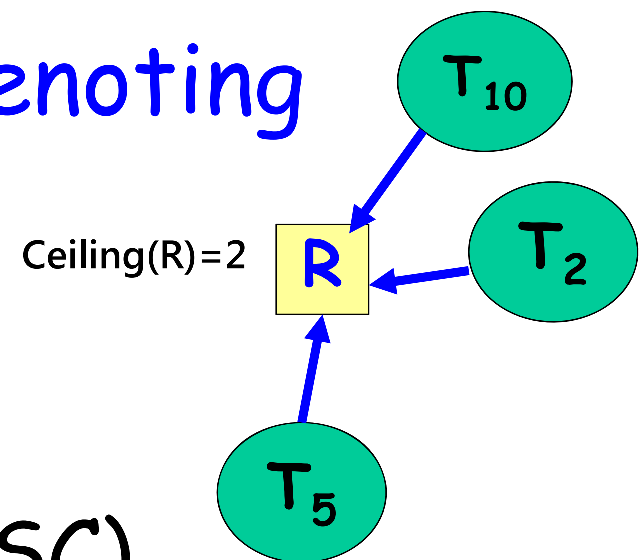
Shortcomings of HLP

- Due to the problem of inheritance-related priority inversion.:
  - HLP to be used cautiously in real applications.
  - Several intermediate priority tasks may miss their deadlines.
- Priority Ceiling Protocol:
  - Extension of PIP and HLP to overcome their drawbacks.
  - Can you list the drawbacks?

Priority Ceiling Protocol

Priority Ceiling Protocol

- Each resource is assigned a ceiling priority:
  - Like in HLP
- An operating system variable denoting highest ceiling of all locked semaphores is maintained.
  - Call it Current System Ceiling (CSC).



Priority Ceiling Protocol (PCP)

- Difference between PIP and PCP:
  - **PIP is a greedy approach**
    - Whenever a request for a resource is made, the resource is promptly allocated if it is free.
  - **PCP is not a greedy approach**
    - A resource may not be allocated to a requesting task even if it is free.

PCP: CSC

- At any instant of time,
  - CSC = max({Ceil(CR_i)|CR_i is currently in use})
  - At system start,
    - CSC is initialized to zero.
- Resource sharing among tasks in PCP is regulated by two rules:
  - Resource Request Rule.
  - Resource Release Rule.