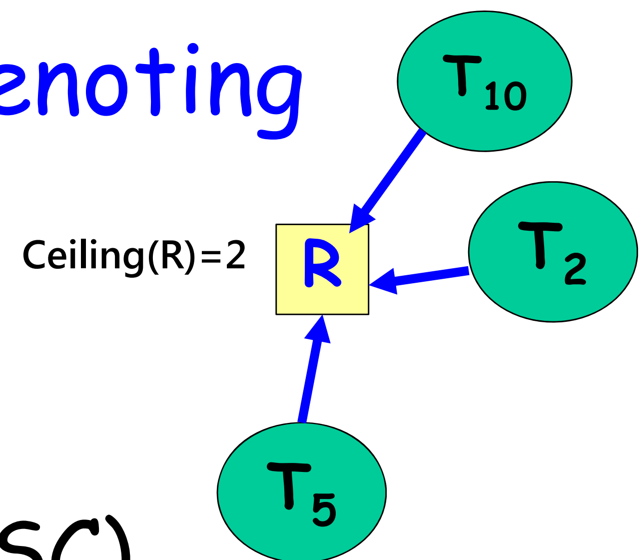
# Priority Ceiling Protocol

# 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.

# PCP: Resource Request Rule

- Resource request rule has two clauses:
  - **Resource grant clause**
    - Applied when a task requests a resource.
  - **Inheritance clause**
    - Applied when a task is made to wait for a resource.

# PCP: Resource Grant Clause

- Unless a task holds a resource that set the CSC (current system ceiling):
  - It is not allowed to lock a resource unless its priority is greater than CSC.

# PCP: Resource Grant Clause

- If a task $T_i$ requests a resource CR:
  - If it is holding a resource whose ceiling priority equals CSC.
    - It is granted access to CR
  - Otherwise, $T_i$ is not granted CR, unless $pri(T_i) >$ CSC.

# PCP: Resource Grant Clause

cont…

- If $T_i$ is granted access to the resource $CR_j$,
  - Then, if CSC<Ceil(CRj), then CSC is set to Ceil($CR_j$).

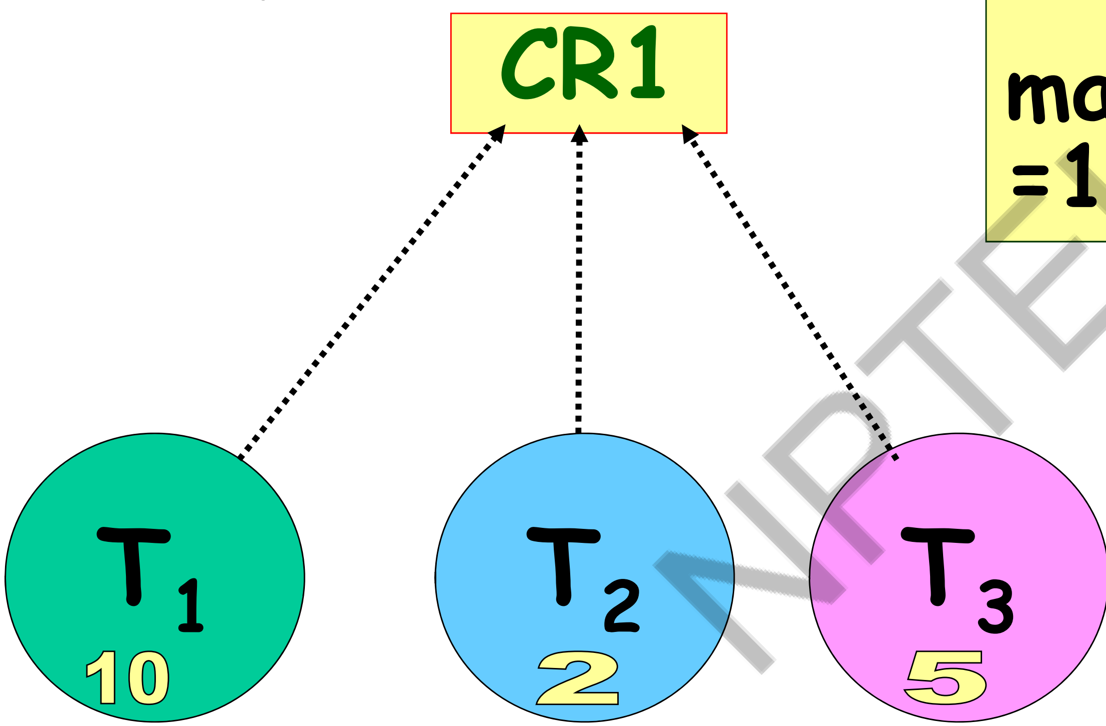
# PCP: Inheritance Clause

- If a task is prevented from locking a resource:
  - The task holding the resource inherits the priority of the blocked task:
    - If the priority of the task holding the resource is lower than that of the blocked task.

# PCP: Resource Release Rule

- If a task $T_i$ releases a critical resource $CR_j$ that it was holding and if $Ceil(CR_j)$ is equal to CSC,
  - Then, CSC is set to $max(\{Ceil(CR_k) | CR_k$ is any resource remaining in use$\})$.
  - Else, CSC remains unchanged.
- The priority of $T_i$ is also updated:
  - Reverts to its original priority if holding no resource
  - Reverts to the highest priority of all tasks waiting for any resource which $T_i$ is still holding

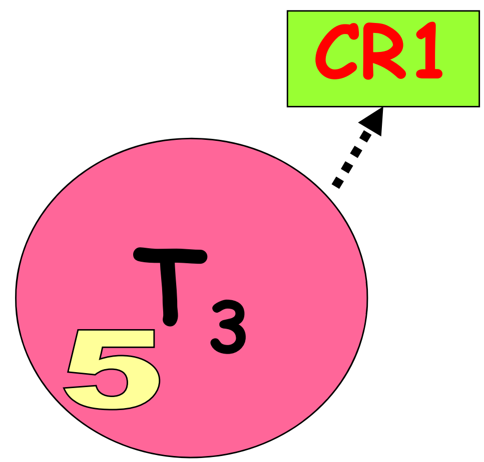
# Example




Ceil(CR1) =

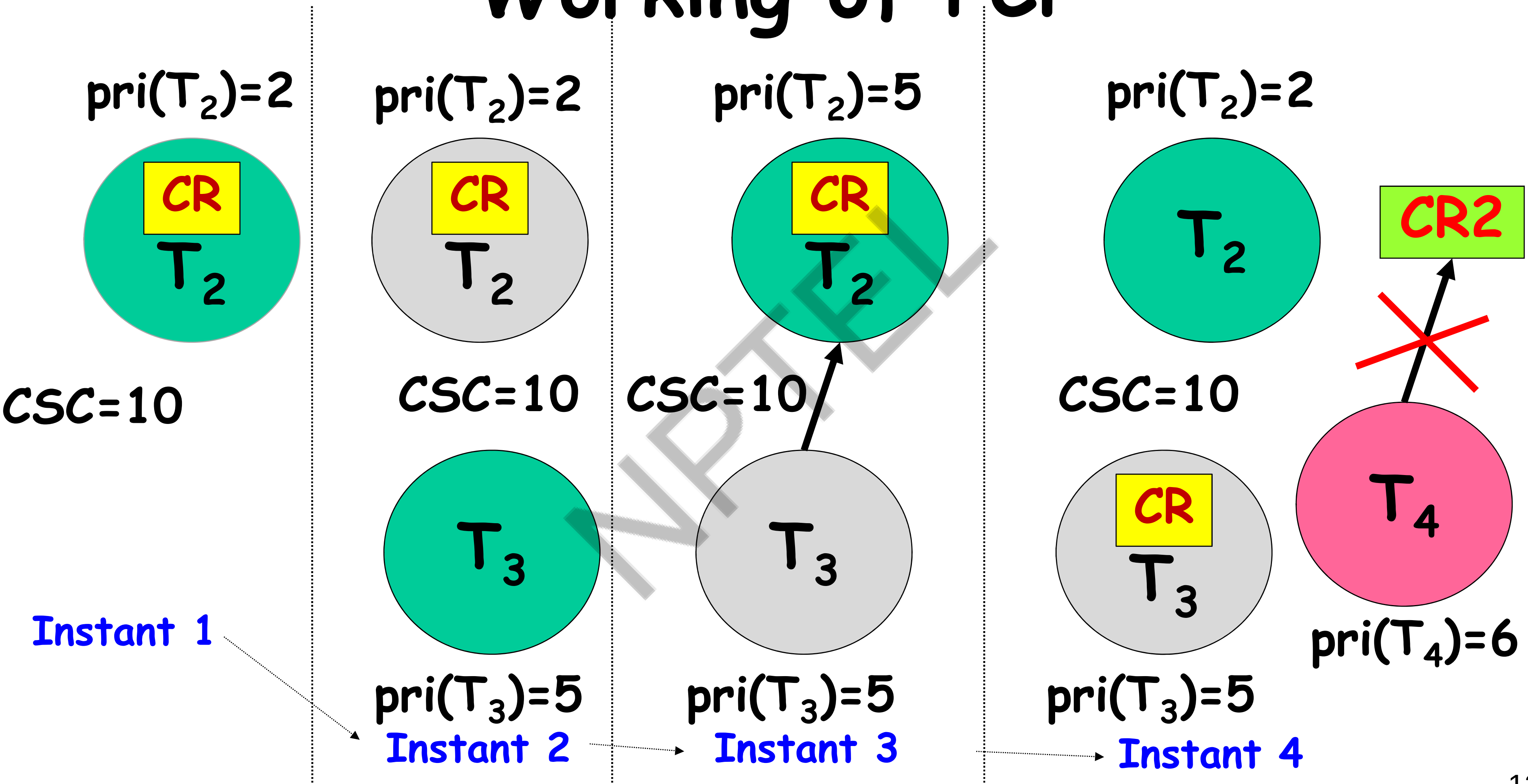
max-prio(T1,T2,T3)
=10

CSC=10




CSC=0

# Working of PCP



# Reflections on PCP

- A task when granted a resource:
  - Only CSC changes.
  - The priority of the task does not change.
- The priority of a task changes:
  - Only when the inheritance clause is applied.

# PCP: An Analysis

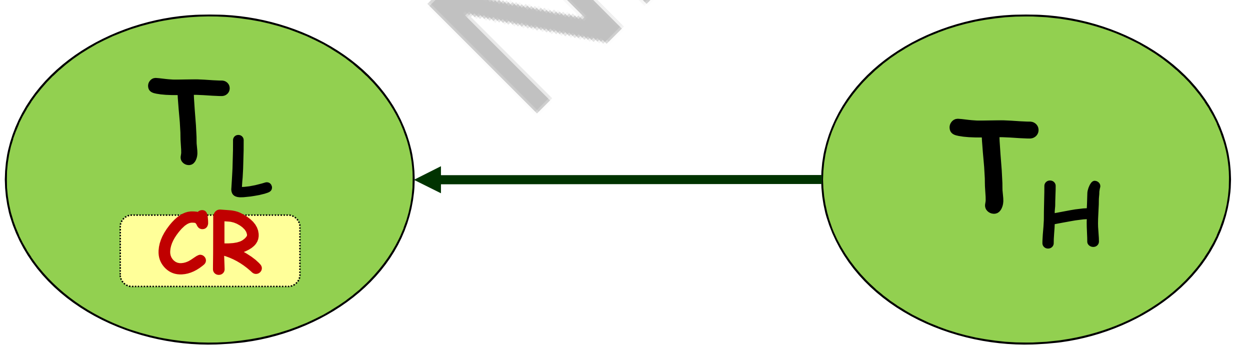
- Prevents deadlocks.
- Prevents chain blocking.
- Prevents unbounded priority inversion.
- Limits inheritance-related inversion.

# Types of Priority Inversions in PCP

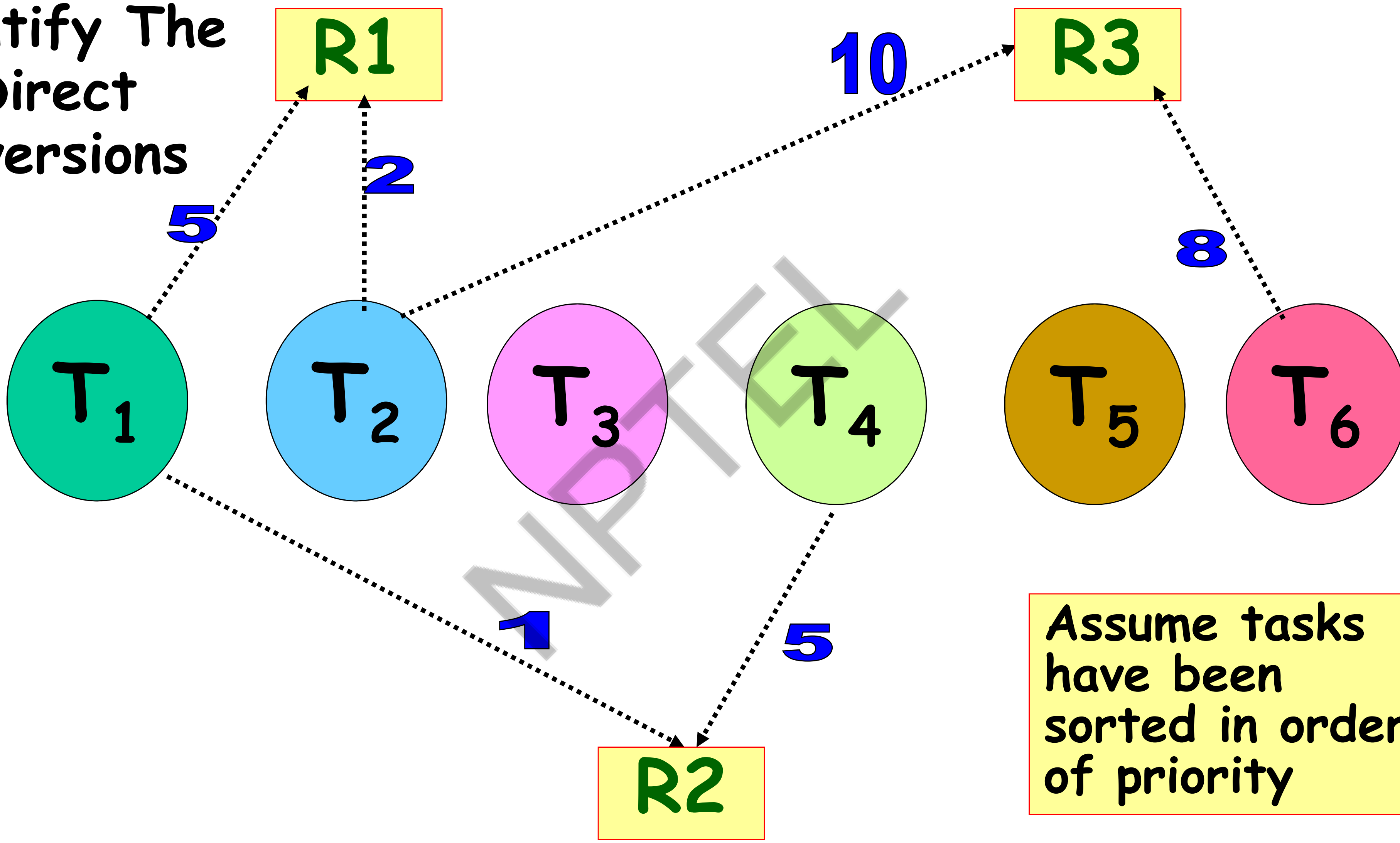
- Direct inversion
- Inheritance-related inversion
- Avoidance-related inversion

# Direct Inversion

- Consider a lower priority task is holding the resource CR:
  - Higher priority task waits for the resource.



# Identify The Direct Inversions




Assume tasks have been sorted in order of priority

# PCP: An Analysis

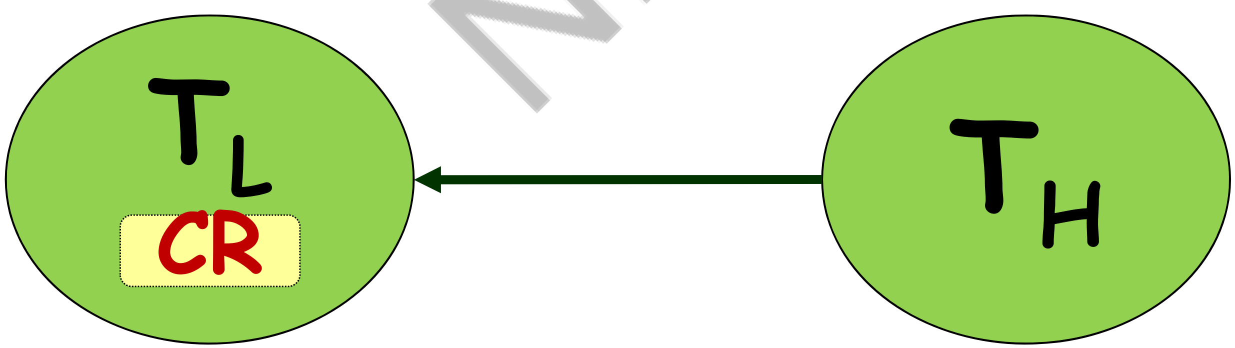
- Prevents deadlocks.
- Prevents chain blocking.
- Prevents unbounded priority inversion.
- Limits inheritance-related inversion.

# Types of Priority Inversions in PCP

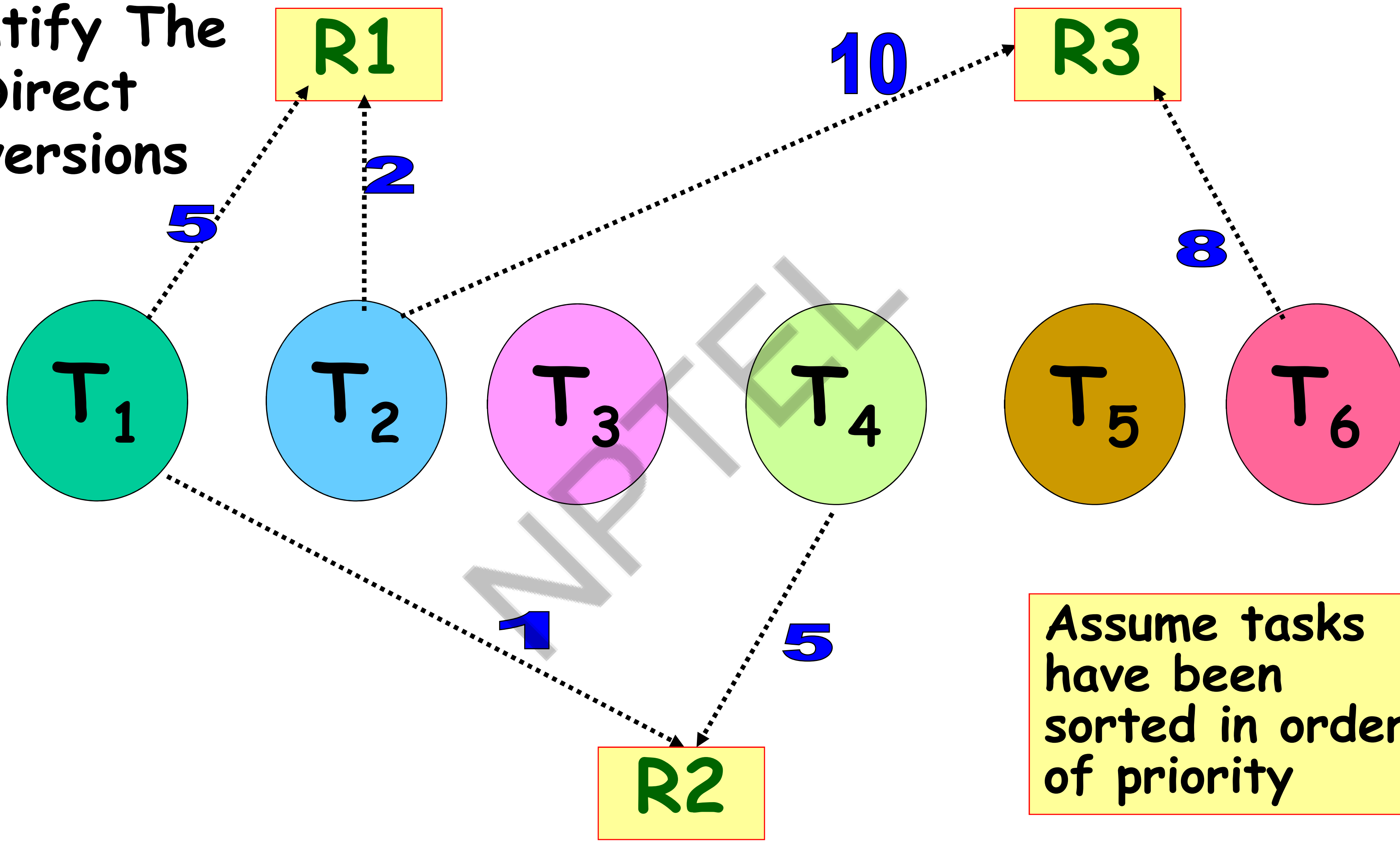
- Direct inversion
- Inheritance-related inversion
- Avoidance-related inversion

# Direct Inversion

- Consider a lower priority task is holding the resource CR:
  - Higher priority task waits for the resource.



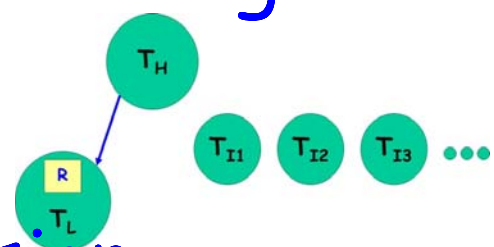
Identify The Direct Inversions



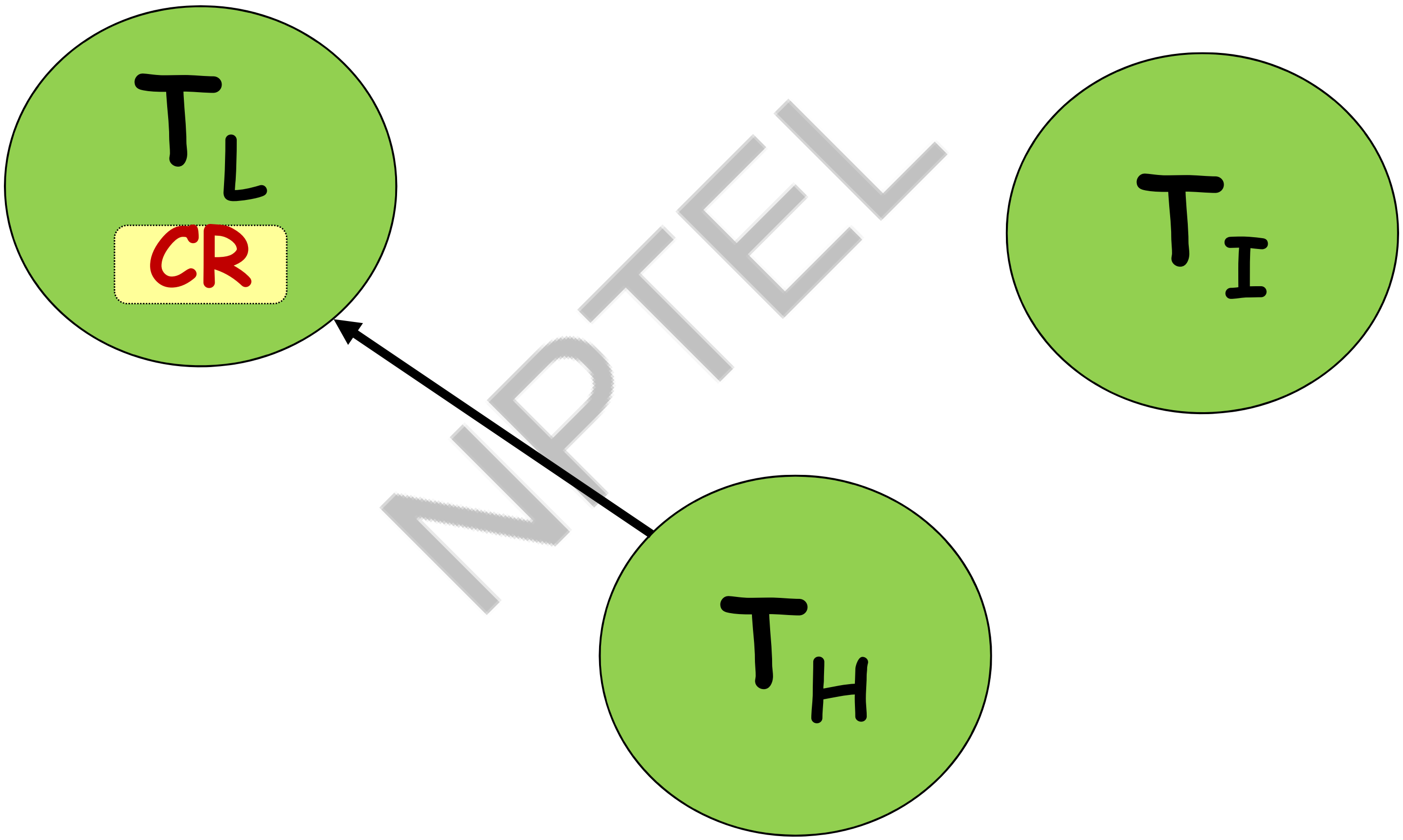

Assume tasks have been sorted in order of priority

# Inheritance-Related Inversion

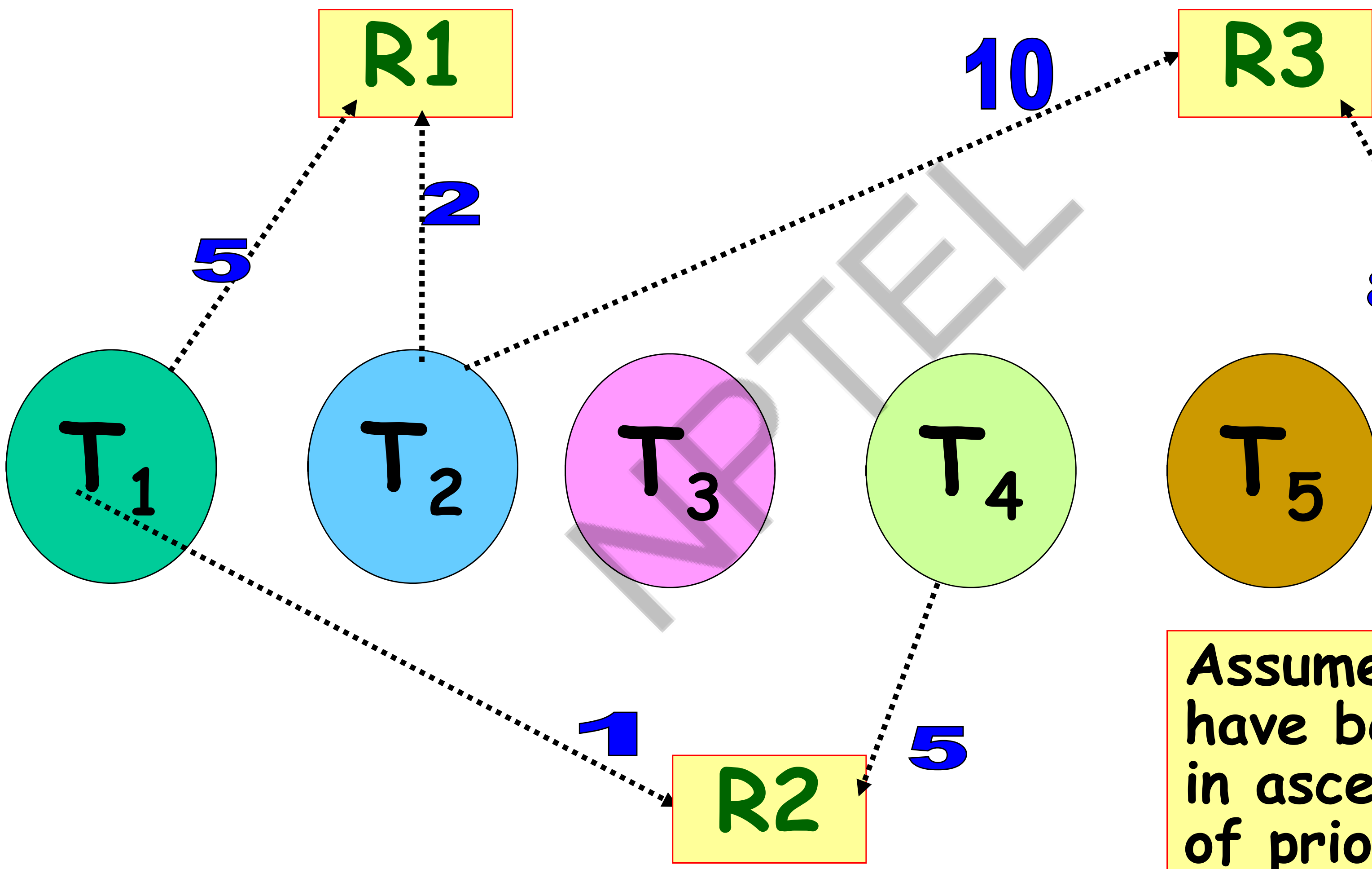
- When a low priority task is holding a resource and a high priority task is waiting for resource:
  - The priority of the low priority task is raised.
  - An intermediate priority task not needing that resource:
    - Undergoes inheritance-related inversion.



# Inheritance-Related Inversion



# Identify The Inheritance-Related Inversions

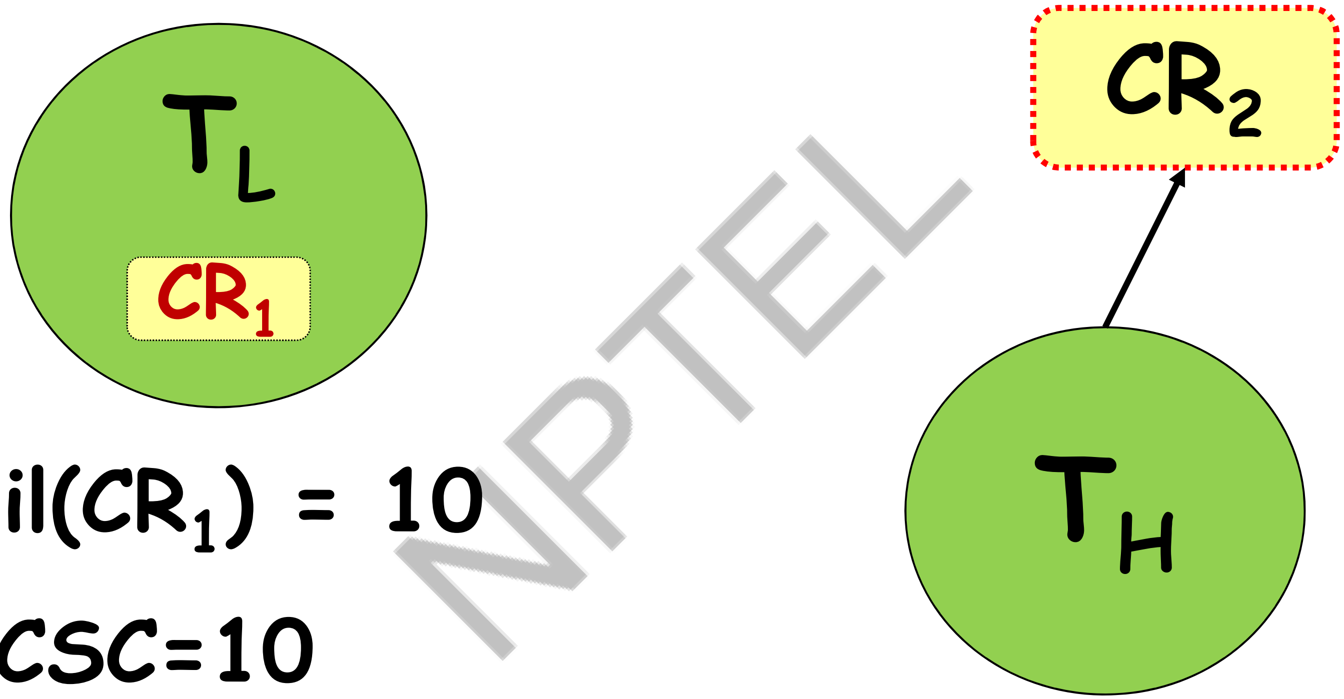



Assume tasks have been sorted in ascending order of priority

# Avoidance-Related Inversion

- Consider a low priority task that is holding a resource:
  - CSC is made equal to the ceiling of the resource being held.
  - A higher priority task, whose priority is lower than the CSC, needs a resource currently not in use:
    - Undergoes avoidance-related inversion
    - Due to the resource grant rule
- Also called priority ceiling-related or deadlock-avoidance inversion.

# Avoidance-Related Inversion

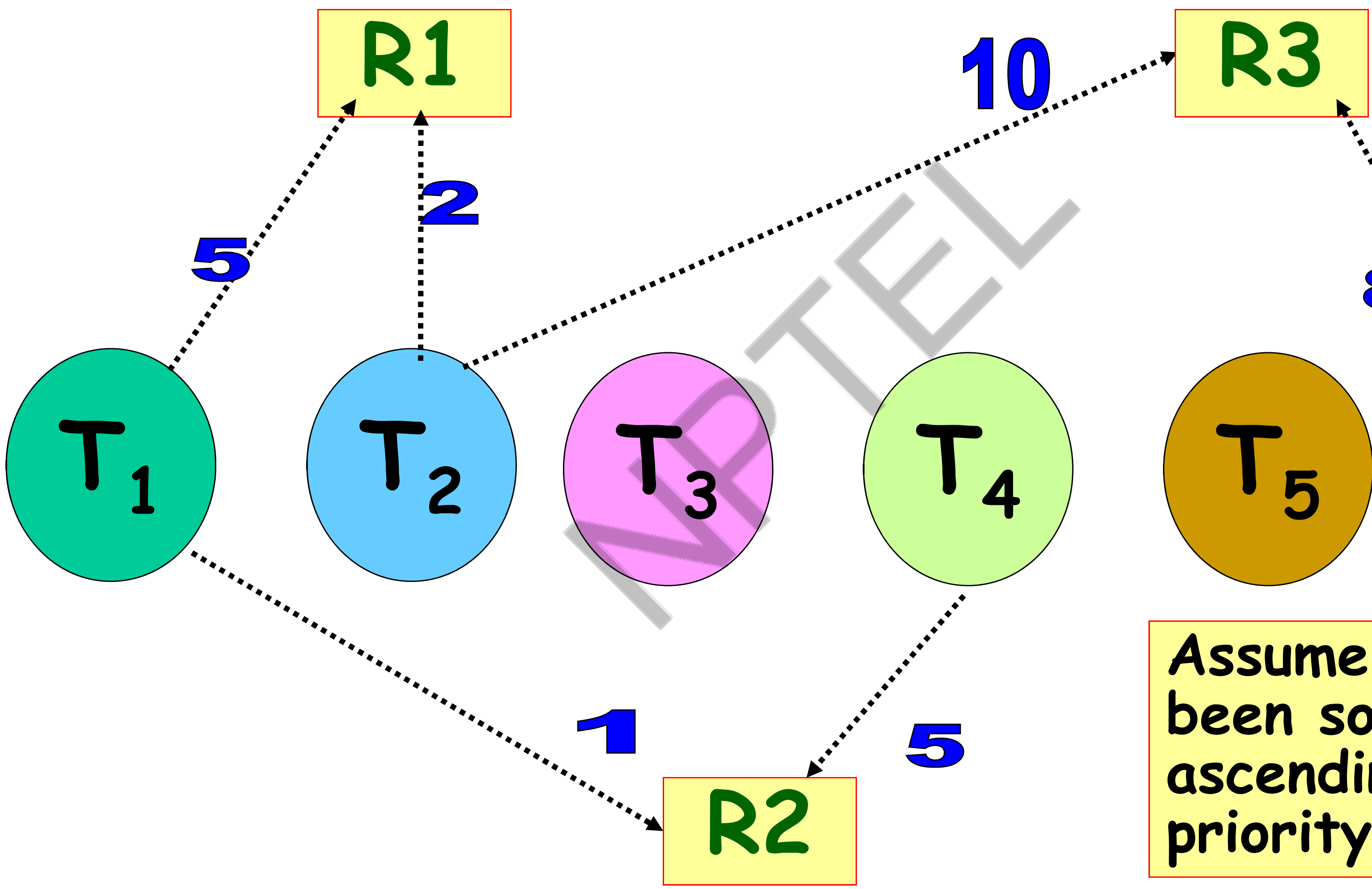



$Ceil(CR_1) = 10$

$CSC=10$

$pri(T_H) = 8$

# Identify The Avoidance-Related Inversions

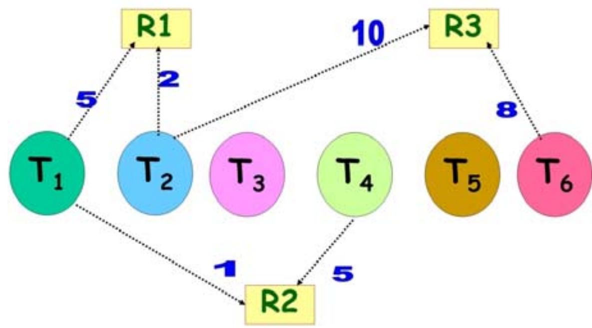





Assume tasks have been sorted in ascending order of priority

# Avoidance-Related Inversion

- **Theorem:** Tasks are single-blocking under PCP.
    - Once a task acquires a resource, it does not undergo any priority inversion.




- Corollary 1:
    - Under PCP a task can undergo at most one priority inversion during its execution.

# Why is PCP Deadlock Free?

- Deadlocks occur only when
  - Different tasks hold parts of each other's required resources.
  - Then they request for the resources being held by each other.
- Under PCP, when a task holds some resource,
  - No other task can hold a resource it may need.

## How is Unbounded Priority Inversion Avoided?

- A task suffers unbounded priority inversion, when
  - It is waiting for a lower priority task to release a resources required by it.
  - In the meanwhile intermediate priority tasks preempt the low priority task from CPU usage.

## How is Unbounded Priority Inversion Avoided?

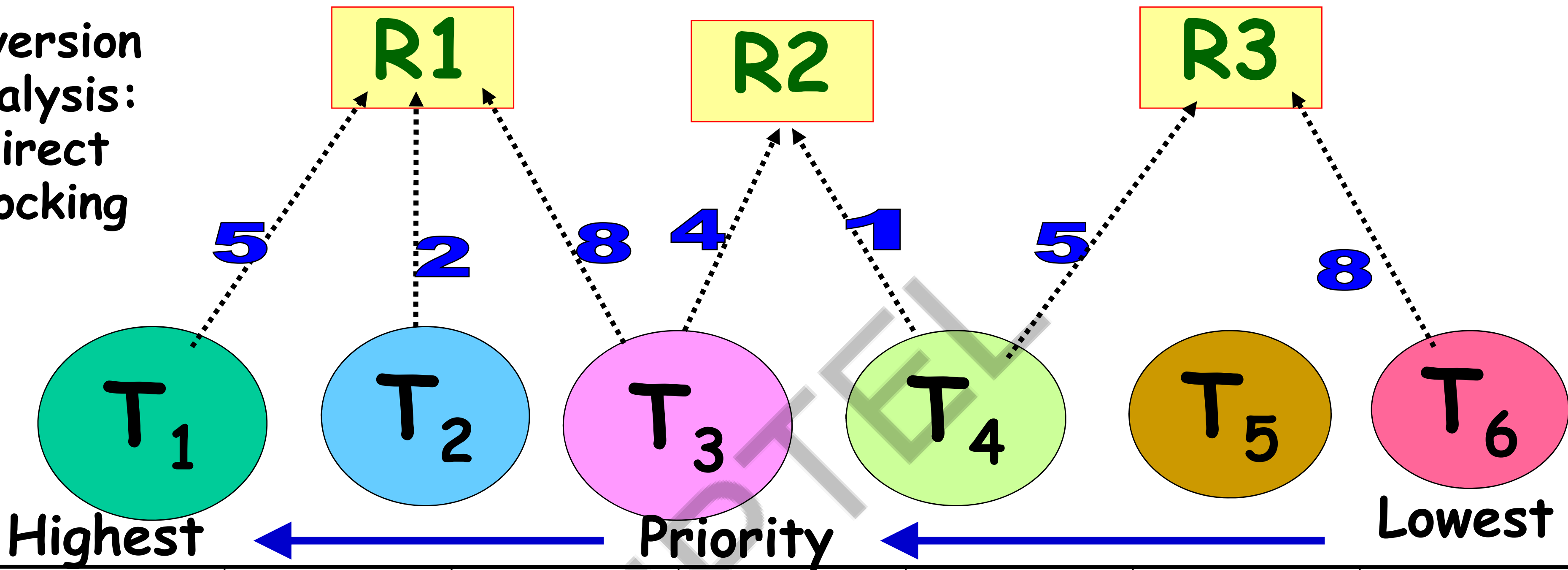
- **Inheritance clause:** Whenever a high priority task waits for a resource held by a low priority task,
  - The lower priority task inherits the priority of high priority task.
  - Intermediate priority tasks can not preempt the low priority task from CPU usage.

# How is Chain Blocking Avoided?

- Already we have seen:
  - Resource sharing among tasks under PCP is single blocking.
  - This gives the clue as to how chain blocking is avoided.

# Analysis Priority Ceiling Protocol

## Inversion Analysis: Direct Blocking

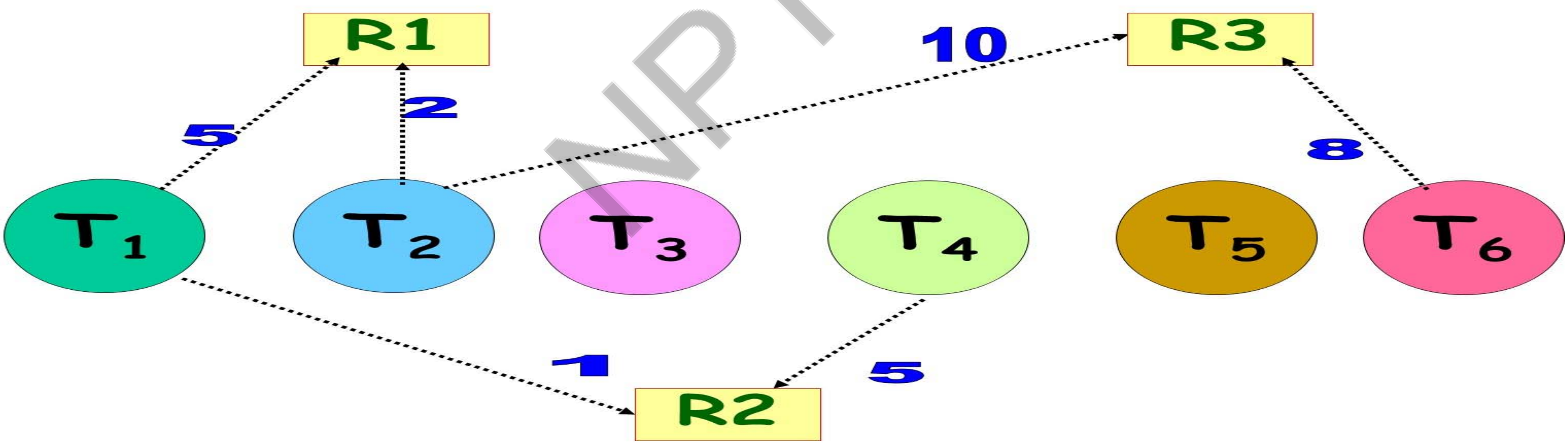



| | T1 | T2 | T3 | T4 | T5 | T6 |
|---|---|---|---|---|---|---|
| T1 | | 2 | 8 | | | |
| T2 | | | 8 | | | |
| T3 | | | | 1 | | |
| T4 | | | | | | 8 |
| T5 | | | | | | |
| T6 | | | | | | |

Direct Inversion

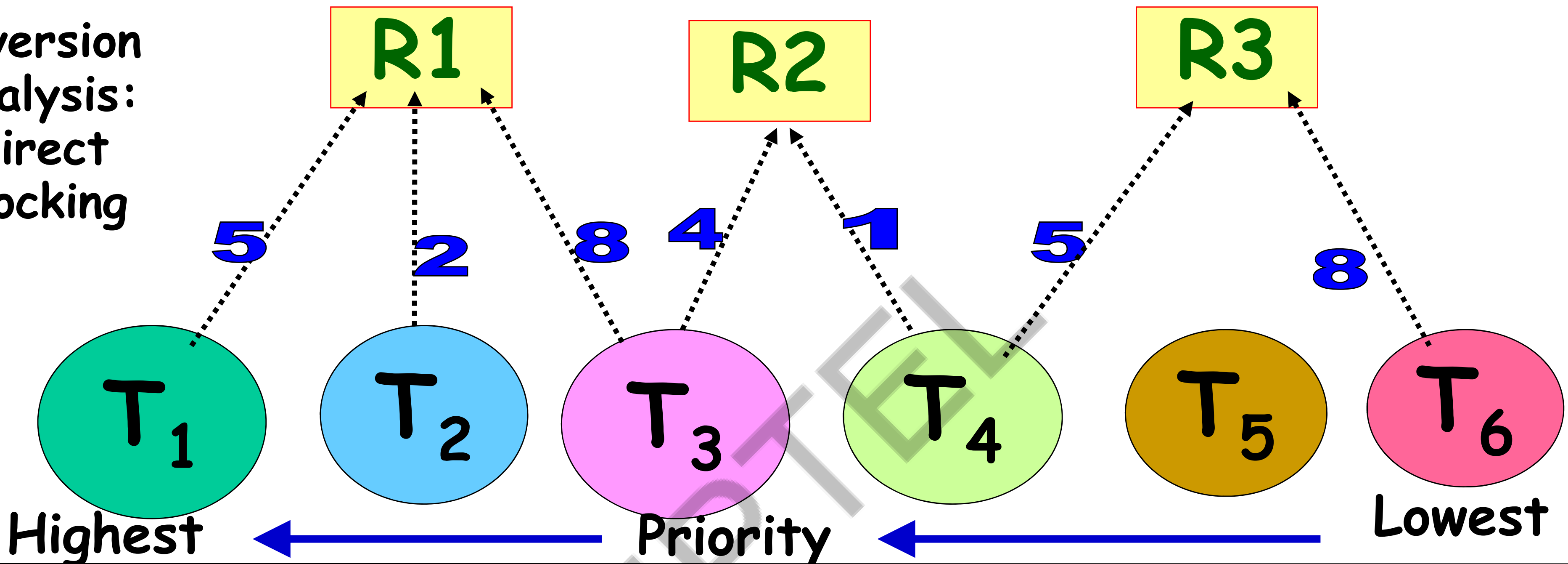
| | T1 | T2 | T3 | T4 | T5 | T6 |
|---|---|---|---|---|---|---|
| T1 | | | | | | |
| T2 | | | | 5 | | |
| T3 | | | | 5 | | 8 |
| T4 | | | | | | 8 |
| T5 | | | | | | 8 |
| T6 | | | | | | |

**Inheritance Inversion**



# Analysis Priority Ceiling Protocol

Inversion Analysis: Direct Blocking



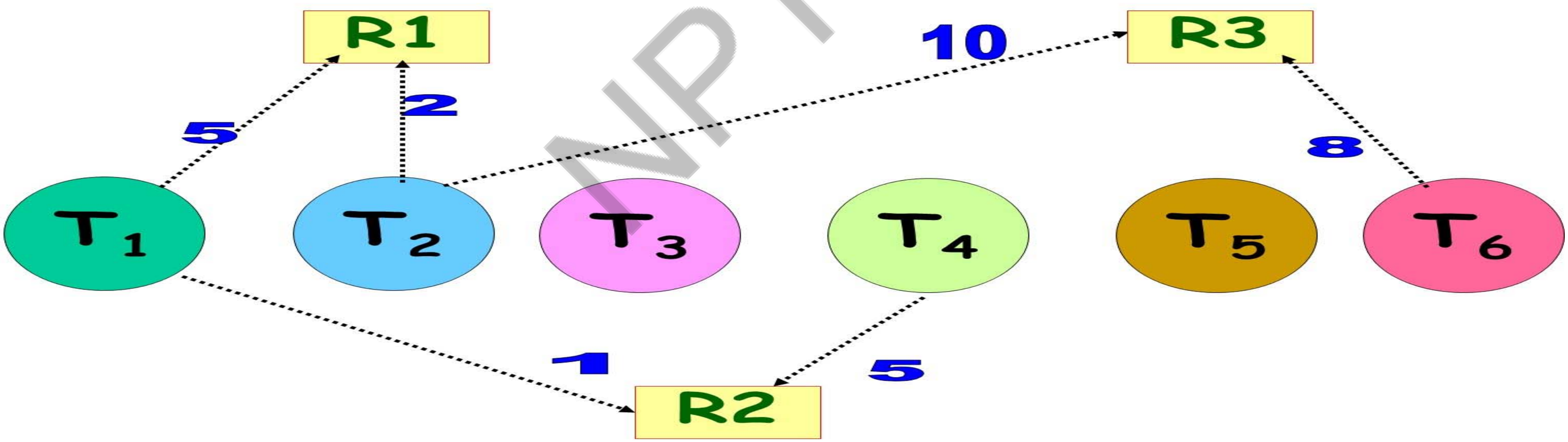

| | T1 | T2 | T3 | T4 | T5 | T6 |
|---|---|---|---|---|---|---|
| T1 | | 2 | 8 | | | |
| T2 | | | 8 | | | |
| T3 | | | | 1 | | |
| T4 | | | | | | 8 |
| T5 | | | | | | |
| T6 | | | | | | |

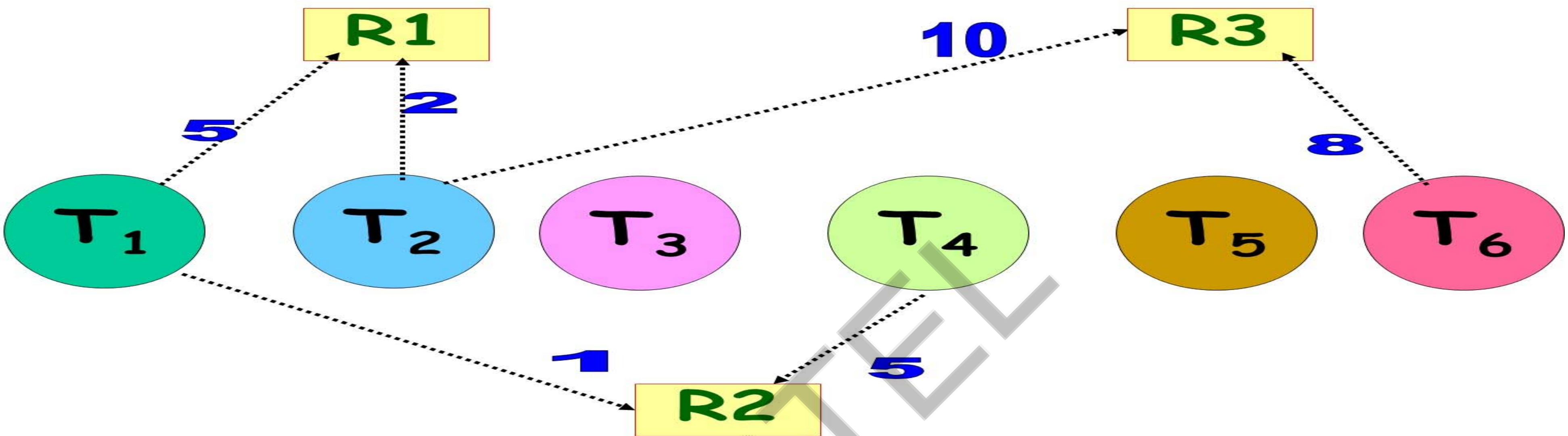
Direct Inversion

| | T1 | T2 | T3 | T4 | T5 | T6 |
|---|---|---|---|---|---|---|
| T1 | | | | | | |
| T2 | | | | 5 | | |
| T3 | | | | 5 | | 8 |
| T4 | | | | | | 8 |
| T5 | | | | | | 8 |
| T6 | | | | | | |

**Inheritance Inversion**








| | T1 | T2 | T3 | T4 | T5 | T6 |
|---|---|---|---|---|---|---|
| T1 | | | | 5 | | |
| T2 | | | | 5 | | 8 |
| T3 | | | | | | |
| T4 | | | | | | 8 |
| T5 | | | | | | |
| T6 | | | | | | |

Avoidance Inversion

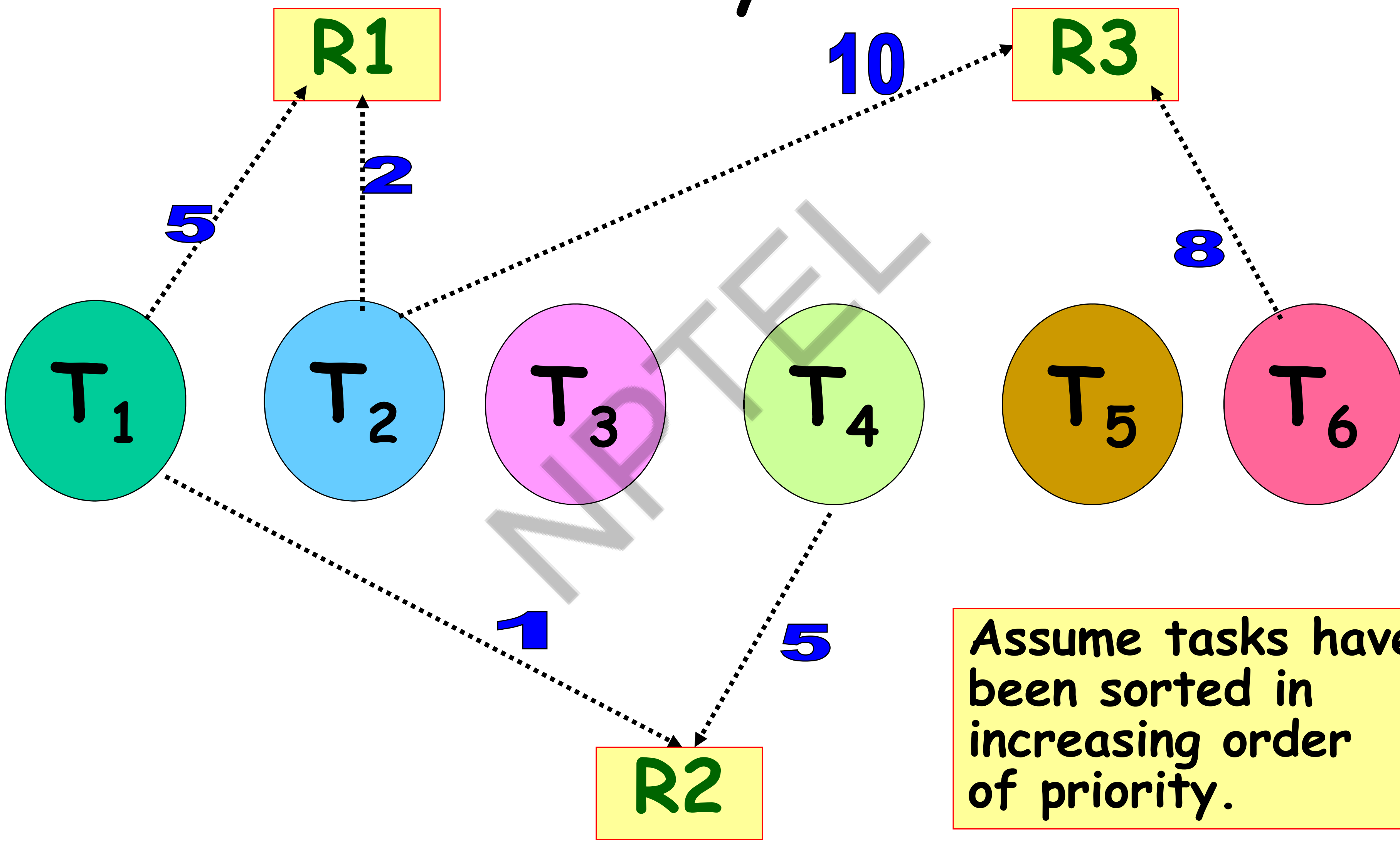
# Analysis of Inversions

- Each inversion table is an upper triangular matrix:
  - Why?
  - A task does not suffer any inversions due to higher priority task.

# Maximum Inversion for a Task

- A task can suffer:
  - At best one of direct, inheritance, or avoidance-related inversion.
- Therefore the maximum inversion that a task can suffer is:
  - **The maximum of the entries of the corresponding row of the inversion table.**

# Inversion Analysis Exercise




Assume tasks have been sorted in increasing order of priority.

# Liu and Lehoczky Condition Under Resource Sharing

- Let $b_i$ denote:
  - The longest time for which a task $T_i$ can undergo priority inversions due to resource sharing.
  - $T_i$ will meet its first deadline if,

$$\left(b_i + e_i + \sum_{j=1}^{i-1} \left\lceil \frac{p_i}{p_j} \right\rceil * e_j\right) \leq p_i$$

where $p_i$ is the period of task $T_i$ and

$$p_1 \leq p_2 \leq p_3 \leq \ldots p_n$$

# PCP for Dynamic Priority Systems

- The priority ceiling values need to change dynamically with time.
  - **A solution:** Each time the priority of a task changes:
    - Update the priority ceiling of each resource and the current system ceiling.
  - However, this would incur unacceptably high processing overhead.

# Comparison of Resource Sharing Protocols

- PIP:
  - Simplest --- requires minimal support from the OS.
  - Effectively overcomes the unbounded priority inversion problem.
  - **However, tasks may suffer from chain blockings and deadlocks.**

# Comparison of Resource Sharing Protocols

- HLP:
  - Requires moderate support from the OS.
  - Solves the chain blocking and deadlock problems.
  - However, intermediate priority tasks:
    - **May suffer from inheritance-related inversions.**

# Comparison of Resource Sharing Protocols

- PCP:
  - Overcomes shortcomings of PIP.
    - Free from deadlocks and chain blocking.
  - Low inheritance-related inversions.
  - Priority of a task on acquiring a resource does not change :
    - Until a higher priority task requests the resource.

## Muliprocessor Scheduling: A Difficult Problem

- “The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors” [Liu’69]

# Multiprocessor RT Task Scheduling

- Most theoretical results reported over last 30 years
- Many are NP-hard problems
- Few optimal results
- Heuristic approaches
- Simplified task models

# The Multiprocessor Scheduling Problem

- Actually two separate problems.
  - Task assignment problem:
    - How to assign tasks to processors?
  - Scheduling problem:
    - How to schedule the tasks on the processor to which it has been assigned.

# Optimal Schedulers?

- Optimal schedulers for uniprocessors:
  - **Static --- RMA**
  - **Dynamic --- EDF**
- What are their complexities?
  - **Linear** for RMA
  - **Log n** for EDF
- General real-time task scheduling in multiprocessor /distributed systems :**NP hard**

# A Few Important Task Assignment Algorithms

- Static:
  - Utilization balancing algorithm
  - Next-fit algorithm for RMA
  - Bin packing algorithm for EDF
- Dynamic:
  - Focused addressing and bidding algorithm
  - Buddy algorithm

# Utilization Balancing Algorithm

- Maintain tasks in a queue:
  - In increasing order of their utilizations.
- Remove tasks one by one from queue:
  - Allocates to the lowest utilized process.
- Usually, $\bar{u} \neq u_i$
- Used when tasks in the node are scheduled using EDF.      Why?

# A Few Basic Issues in Real-Time Operating Systems

# Basic Requirements of an RTOS

- Real-time priority levels
- Real-time task scheduling policy
- Support for resource sharing protocols
- Low task preemption times:
    - Of the order of milli/micro seconds
- Interrupt latency requirements

# A Few Basic Issues in Real-Time Operating Systems

# Basic Requirements of an RTOS

- Real-time priority levels
- Real-time task scheduling policy
- Support for resource sharing protocols
- Low task preemption times:
  - Of the order of milli/micro seconds
- Interrupt latency requirements

# Additional Requirements of an RTOS

- Memory locking support
- Timers
- Real-time file system support
- Device interfacing

# Support for Real-Time Priority Levels

- **Static task priority level (or real-time priority level):**
  - Operating system does not change the programmer assigned task priorities.
- **Dynamically changing task priority:**
  - Supported by traditional operating systems.
  - The objective is to maximize system throughput.

# Task Scheduling Support

- Should allow programmers choice of real-time task schedulers such as:
  - RMS
  - EDF
  - Custom task schedulers

# Resource Sharing Support

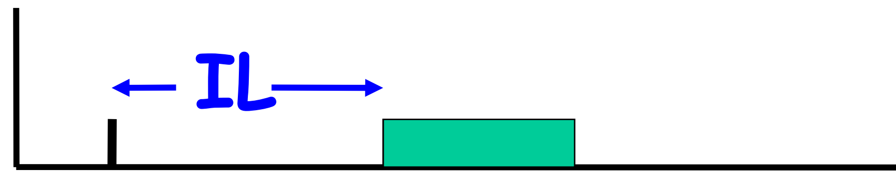
- Should support resource sharing protocols such as PCP:
  - To minimize priority inversions during resource sharing among real-time tasks.

# Task Preemption Time

- RTOS task preemption times:
  - Of the order of a few micro seconds.
- Worst case task preemption times for traditional operating systems:
  - Of the order of a second.
- The significantly large latency:
  - Caused by a non-preemptive kernel.

# Interrupt Latency Requirements

- **Interrupt latency:**




  - The time delay between the occurrence of an interrupt and the running of the corresponding ISR.
- The upper bound on interrupt latency:
  - Must be bounded and less than a few micro seconds.

# How Low Interrupt Latency is Achieved?

- Perform bulk of ISR processing:
  - As a queued low priority task called deferred procedure call (DPC).
- Support for nested interrupts requires:
  - Not only preemptive kernel routines.
  - Should be preemptive during interrupt servicing as well.

# Requirements on Memory Management

- Traditional operating systems support:
  - Virtual memory and memory protection features.
- Not supported by embedded RTOS:
  - Increase worst-case memory access time drastically.
  - Result in large memory access jitter

# Virtual Memory

- Virtual memory technique helps reduce average memory access time:
  - **But degrades the worst-case memory access time.**
  - Page faults incur significant latency.
- Without virtual memory support:
  - Providing memory protection difficult.
  - Also, memory fragmentation becomes a problem.

## Do Any RTOS Support Virtual Memory?

- RTOS for large applications need to support virtual memory :
  - To meet memory demands of heavy weight real-time tasks.
  - Support running non-real-time applications: text editors, e-mail client, Web browsers, etc.

# Memory Protection: Pros and Cons

- Advantage of a single address space:
  - Saves memory bits and also results in light weight system calls.
  - For very small embedded applications:
    - Memory overhead can be unacceptable.
- Without memory protection:
  - The cost of developing and testing a program increases.
  - Also, maintenance cost increases.

# Memory Locking

- **Memory locking:**
  - Prevents a page from being swapped from memory to hard disk.
- In the absence of memory locking support:
  - Even critical tasks can suffer large memory access jitter.

# Asynchronous I/O

- Traditional read(), write() system calls:
  - Synchronous I/O.
  - Process is blocked while it waits for the results.
- Asynchronous I/O:
  - Non-blocking I/O.
  - **aio_read(), aio_write()**

# RTOS For Embedded Systems

- Embedded systems have small memories:
  - Obviously, OS with large footprints cannot be used in embedded applications.
- Power saving feature is desirable.

## Embedded Systems: A Basic Question

- RTOS and application programs are stored in a flash memory:
  - Is it necessary to have a RAM?
  - Yes, otherwise memory access times would be very high --- programs would run very slow.

# Using Traditional Unix as RTOS

- Two major shortcomings of Unix would pose as obstacle:
  - Nonpreemptable kernel
  - Dynamic priority levels

# Nonpreemptable Kernel

- A user program executes in the kernel mode:
  - When it makes a system call
- Original Unix developers:
  - Did not visualize use of Unix in real-time and multiprocessor systems.

# Nonpreemptable Kernel

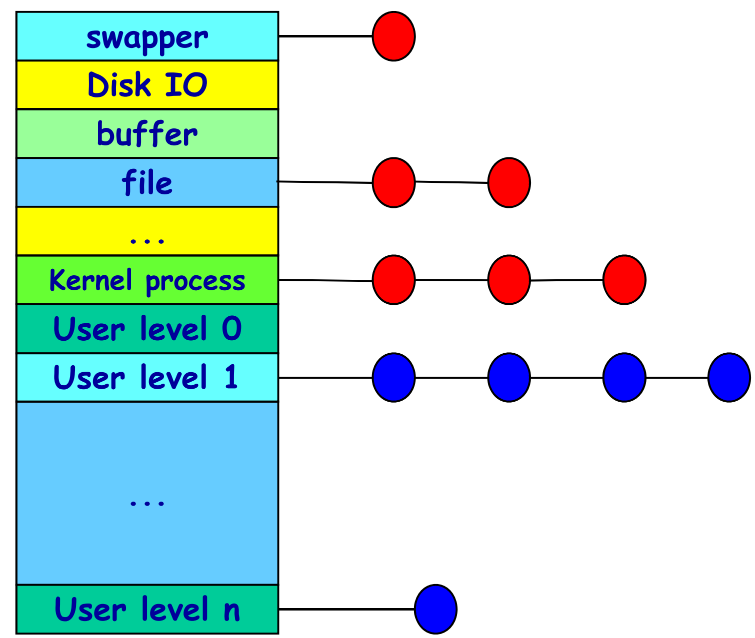
- Whenever the system is in the kernel mode:
  - All interrupts are disabled.
- **Why?**
  - **This is an easy and efficient way to preserve the integrity of kernel data.**

# Nonpreemptable Kernel

- In a real-time application:
  - A nonpreemptable kernel can cause deadline misses.
  - task preemption time = time spent in kernel mode + context switch time
  - This can be of the order of a second in the worst case.

# Dynamic Priorities

- The Unix scheduler maintains a multilevel feedback queue.
- The system by default:
  - Uses a 1 second time slice.
- Task priorities:
  - Recomputed after each interval.



# Unix Scheduling Policy

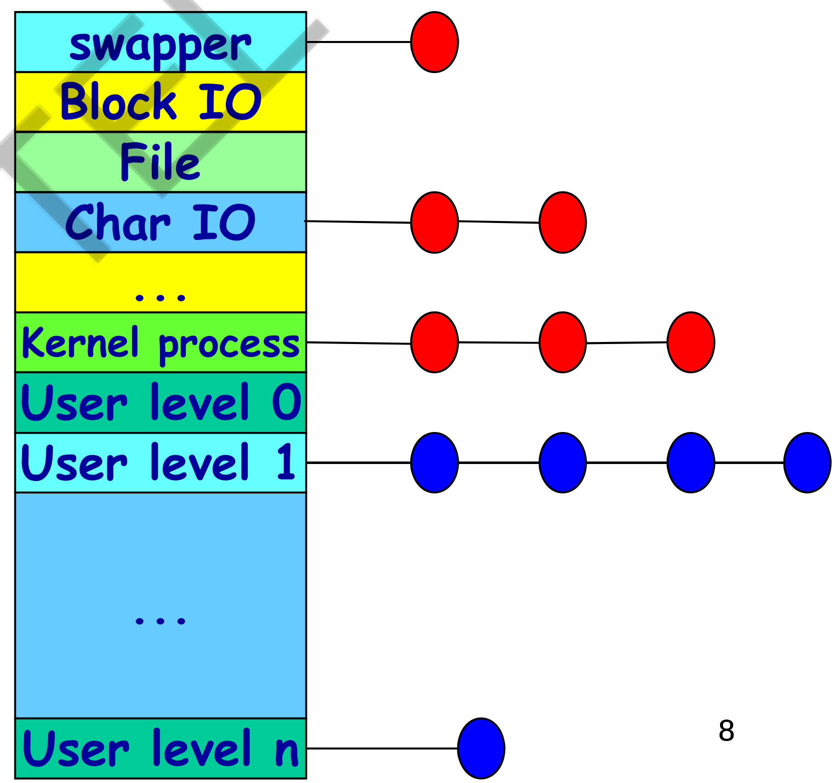
- $Pr(Ti,j)=CPU(Ti,j)+Base(Ti)+nice(Ti)$
- $CPU(Ti,j)=U(Ti,j-1)/2+CPU(Ti,j-1)/2$
  - **Pr(Ti,j)** : Priority of Ti at the jth instant
  - **CPU(Ti,j)**: History of CPU usage by Ti with geometrically decreasing weights
  - **U(Ti,j)** : CPU utilization in the jth instant
  - **Base(Ti)** : Used to separate different tasks into bands
  - **nice(Ti)**: Set by the programmer

# History of CPU Usage

- $CPU(Ti,j)=U(Ti,j)/2+CPU(Ti,j-1)/2$
  - By unfolding the recurrence
- $CPU(Ti,j)=U(Ti,j)/2+U(Ti,j-1)/4+CPU(Ti,j-2)/4$
- Or, $CPU(Ti,j)=U(Ti,j)/2+U(Ti,j-1)/4+U(Ti,j-2)/8 \ldots$
- Geometrically reduced weightage for past CPU utilization history.

# Base Priorities

- Different base priorities segregate tasks into the following base bands:
  - Swapper
  - Block I/O
  - File manipulation
  - Character I/O
  - Kernel process
  - User processes



# The Central Idea

- I/O speed is very slow compared to CPU speed:
  - **I/O is the bottleneck**
- I/O channels should be kept as busy as possible.
- To increase average throughput:
  - **Raise the priority of I/O intensive tasks.**

# The Consequence

- I/O bound processes gravitate towards higher and higher priorities:
  - If a real-time task spends most of its time in computation:
    - It can miss its deadline.
  - This is unacceptable for hard real-time applications.

# Unix V as RTOS

- Suitable only for soft real-time applications:
  - Clearly unsuitable for hard real-time applications.
  - Deadlines of the order of milli/micro seconds.

# Main Deficiencies of Unix V

- **Task preemption time of the order of a second:**
  - Preemption is disabled during system calls.
- **Dynamic recomputation of priorities**
- **Resource sharing problem:**
  - High-priority tasks may wait for a low-priority task to release resources.

# Other Deficiencies of Unix V

- Inefficient interrupt processing
- Unsatisfactory support of device interfacing
- Unsatisfactory timers
- Lack of real-time file support

# Example 2

- On account of every context switch :
  - Assume an overhead of 1 msec.
  - Compute the completion time of TB.

# Practice Questions

- What do you understand by priority inversion?
- (T/F) When several tasks share a set of critical resources,
  - Is it possible to avoid priority inversion altogether by using a suitable task scheduling algorithm?

# Practice Question

- When priority inheritance is used in the resource sharing protocol:
  - What do you understand by inheritance-related inversion?
- When a set of real-time tasks share certain critical resources using the priority inheritance protocol:
  - The highest priority task does not suffer any inversions. **(T/F)**

# Practice Question

- When priority inheritance scheme is used, a task needing a resource undergoes priority inversions due to:
  - A higher priority task holding the resource
  - A lower priority task holding the resource
  - An equal priority task holding the resource
  - Either a higher or a lower priority task holding the resource

# Practice Question

- Using semaphores of traditional operating systems, what is the maximum duration for which a task may undergo priority inversion:
    1. Longest duration for which a higher priority task uses a shared resource.
    2. Longest duration for which a lower priority task uses a shared resource.
    3. Sum of the durations for which different lower priority tasks may use the shared resource.
    4. Greater than the longest duration for which a lower priority task uses a shared resource

# Identify TRUE or FALSE

- When a set of real-time tasks share certain critical resources using priority ceiling protocol (PCP):
  - The highest priority task does not suffer any inversions.
- Under PCP, a task not requiring any resource:
  - May still undergo priority inversion for some duration.