Handling Resource Sharing and Dependencies Among Real-Time Tasks
Real-Time Systems Design (CS 6414)

Sumanta Pyne

Assistant Professor
Computer Science and Engineering Department
National Institute of Technology Rourkela
pynes@nitrkl.ac.in

February 4, 2021

Overview

- Scheduling disscussed so far on independent tasks
- Rare in real-life applications
- Task often have explicit dependencies among themselves
- Implicit dependencies are more common
- Tasks might become interdependent for several reasons
- Dependency arises when one task needs result of another to proceed
- Example: In a fly-by-wire aircraft
  - positional error computation task need results of current position task
  - positional error computation after current position determination over
- Even no explicit data exchanges involved tasks need to run in an order
- Example: System initialization task need to run first before others
- Dependency restricts applicability of results on task scheduling
- EDF and RMA impose no contraints on order of task execution
- EDF or RMA might violate constraints due to task dependencies
- Need to extend EDF and RMA to cope with intertask dependencies

Overview

- Discussed CPU scheduling not satisfactory for shared critical resources
- Task using critical resources can't be preempted at any time
  - without risking correctness of computed results
- New methods to schedule critical resources among tasks are required
- How critical resources may shared among a set of real-time tasks ?
- Problems arise if traditional resource sharing deployed in real-time
- How EDF and RMA augmented for tasks with dependencies ?

Resource Sharing Among Real-Time Tasks

- Real-time tasks need to share resources among themselves
- Shared resources used by individual tasks in exclusive mode
- Task using a resource can't immediately hand over resource to another
- It can do so only after completing use of resource
- If task preempted before completion of usage then resource corrupted
- Examples - files, devices, certain data structures
- These are non-preemptable or *critical resources*
- Non-preemptable resources referred as *critical sections*
- In OS it is section of code exclusively uses non-preemptable resource

Resource Sharing Among Real-Time Tasks

- Sharing critical resources require a different set of rules
- Compared to resource such as CPU shared among tasks
- CPU shared among tasks with cyclic, EDF and RMA scheduling
- CPU is a *serially reusable resource*
- Once a task gains access to serially reusable resource uses exclusively
- Two different tasks can't run on one CPU at same time
- Another feature, task executing on CPU preempted and restarted later
- A serially reusable resource is used in exclusive mode
- Task using it preempted from it, later allowed no affect on correctness
- A non-preemptable resource also used in exclusive mode
- Task using non-preemptable resources can't preempted resource usage
- Otherwise resource become inconsistent, lead to system failure
- Low priority task using non-preemptable resource, high priority waits
- Hence EDF & RMA for sharing serially reusable resources (e.g. CPU)
  - can't satisfactorily used to share non-preemptable resources

Priority Inversion

- Mutual exclusion of data and resources in traditional systems
  - semaphores, locks and monitors
- If used for resource sharing in a real-time applications then
  - simple priority inversion and unbounded priority inversion can occur
- Unbounded priority inversions cause deadline miss, system failure
- Simple priority inversion
  - When a lower priority task already holding a non-preemptable resource
  - A higher priority task needing same resource has to wait
  - Can't make progress with its computations
  - Remains blocked until lower priority task releases
  - In such situation higher priority task undergo simple priority inversion
  - On account of lower priority task for duration it waits
  - Unavoidable two or more tasks share a non-preemptable resource
  - A single simple priority is not difficult to cope with
  - Duration bounded by longest duration lower task need critical resource

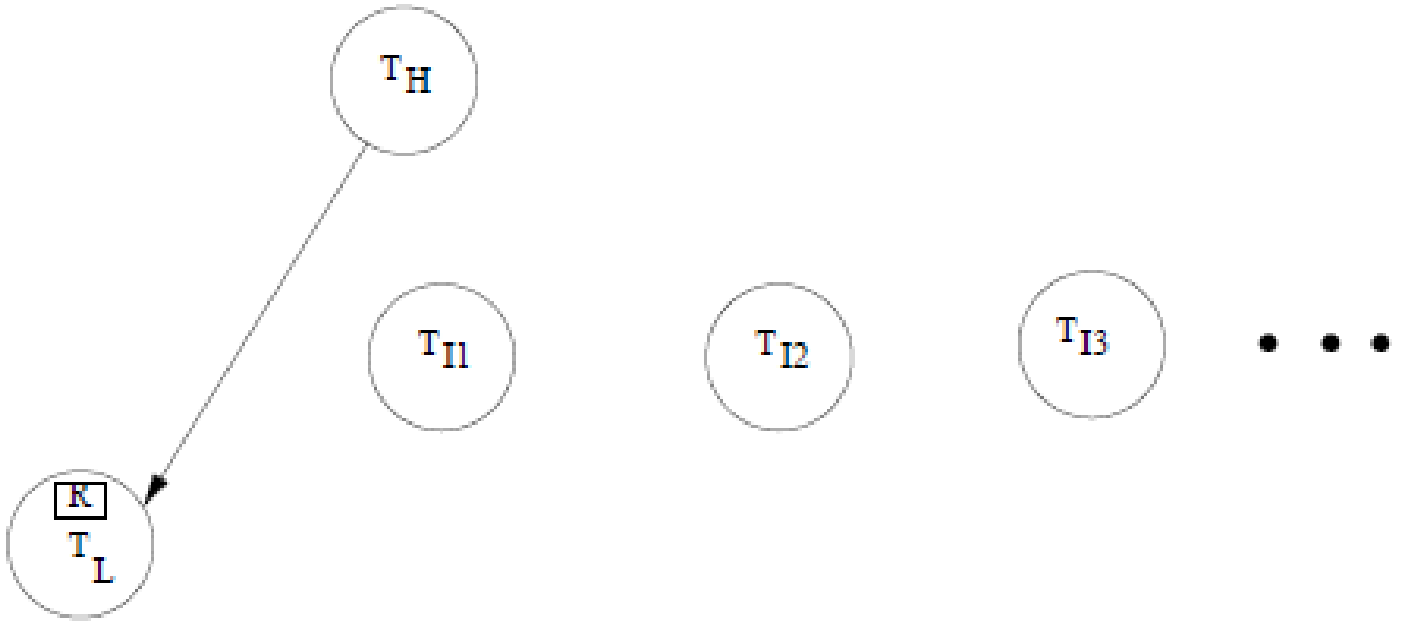
Simple Priority Inversion

- Simple priority inversion delays higher priority task
- Duration for task blocks made very small
- If tasks are restricted to very brief periods of critical section usage
- Can be tolerated through careful programming
- More serious resource sharing issue - unbounded priority inversion

Unbounded Priority Inversion

- Problem faced by programmers of real-time & embedded systems
- Can upset all calculations on worst case response time
- Cause task to miss its deadline
- When higher priority task waits for lower to release resources needed
- Intermediate priority tasks preempt lower from CPU usage repeatedly
- Lower priority task can't complete critical resource usage
- Higher priority task waits indefinitely for required resource release
- Consider real-time application with high & low priority tasks TH & TL
- Assume TH and TL need to share a critical resource R
- Tasks TI1,TI2,TI3,. . . have priorities intermediate between TH & TL
- Intermediate tasks do not need resource R in their computations
- Assume TL starts executing at some instant, locks non-preemptable R

Unbounded Priority Inversion

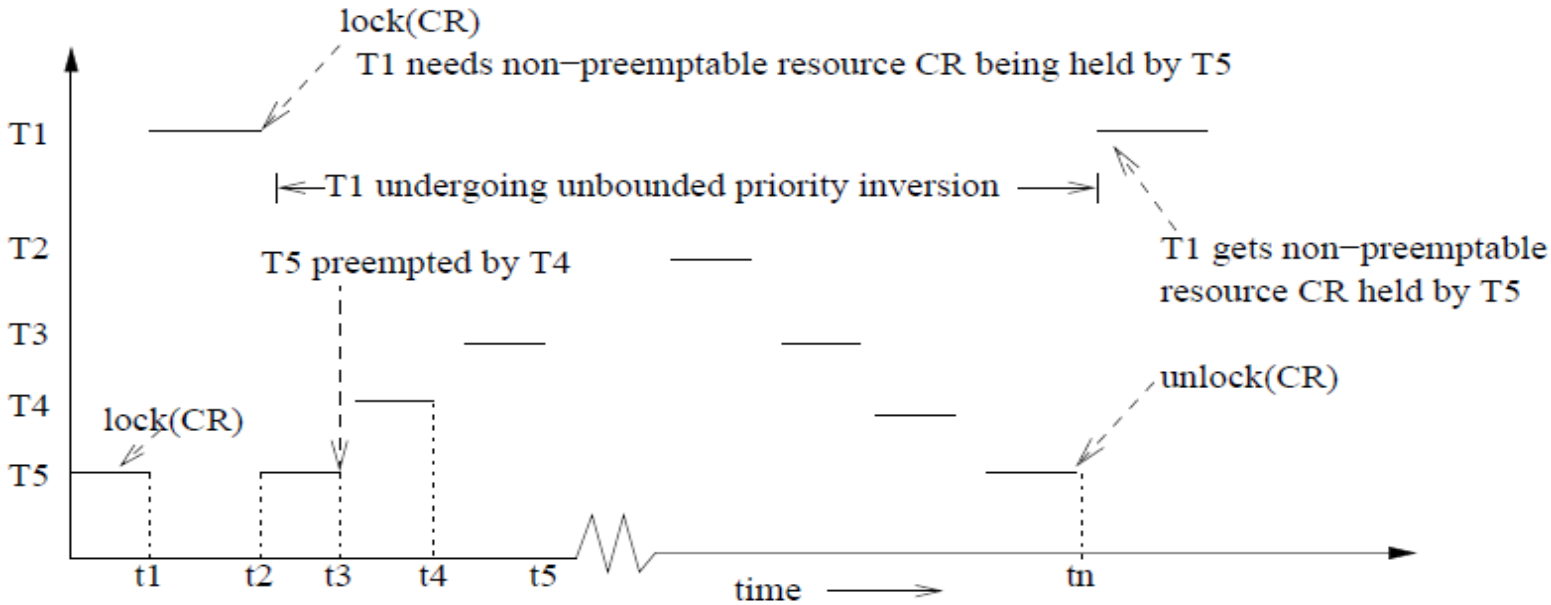


- Soon after TH is ready, preempts TL, starts executing, needs R
- TH blocked for R held by TL, TL continues execution
- TL may be preempted from CPU usage by TI1,TI2,TI3,...
- TI1,TI2,TI3,... become ready, do not require R

Unbounded Priority Inversion

- TH waits for
  - TL to complete R usage, and
  - TI1,TI2,TI3,... to complete computation preempting TL
- Result TH to wait for R for a considerable amount of time
- In worst case, TH might wait indefinitely for
  - TL to complete usage of R, and
  - if TL is repeatedly preempted by TI1,TI2,TI3,... not needing R
- In such scenario, TH undergoes *unbounded priority inversion*

Timing Diagram - Unbounded Priority Inversion



- prior(T1)>prior(T2)>prior(T3)>prior(T4)>prior(T5)
- At t0: T5 executes, uses non-preemptable critical resource CR
- At t1: T1 arrives, preempts T5, starts to execute
- At t2: T1 requests for CR, blocks as CR held by T5
- T1 blocks, T5 resumes execution
- At t3: T4 don't need CR preempts T5 from CPU, starts to execute
- T3 preempt T4 and so on
- T1 suffers multiple priority inversions

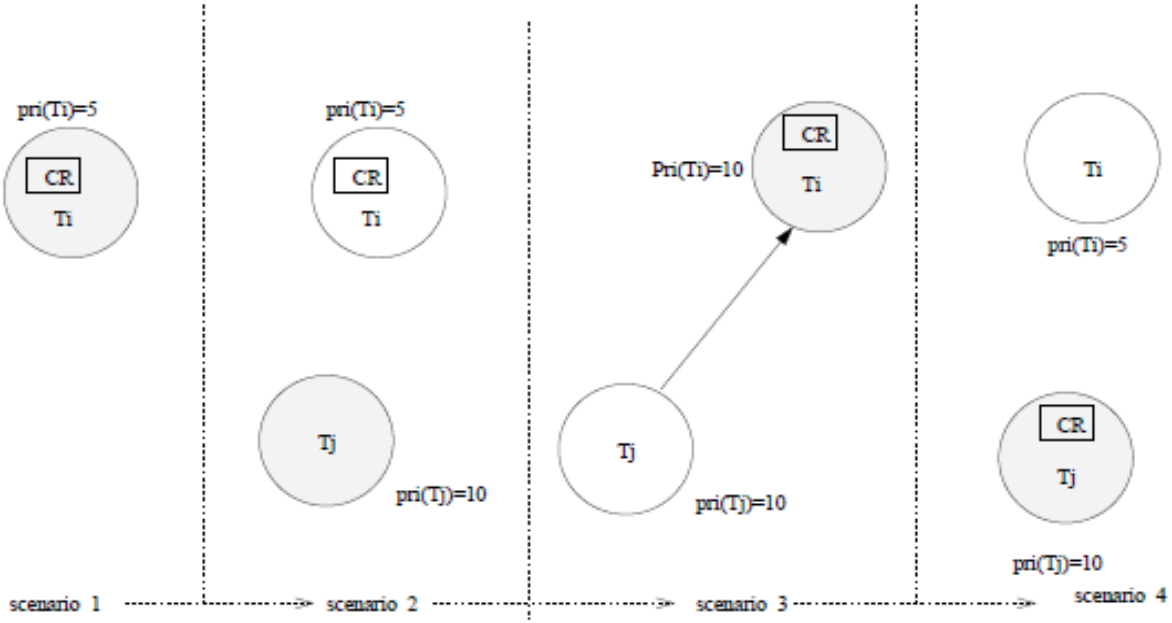
Dealing with Unbounded Priority Inversion

- Unbounded Priority Inversion from traditional sharing techniques
- Avoid preempt low priority task with critical resource by intermediate
- Raise priority level of low priority task equalled to waiting high priority
- Low priority task made to inherit priority of waiting high
- *Priority inheritance protocol (PIP)* by Sha and Rajkumar
  - When a task suffers priority inversion
  - priority of lower priority task holding resource raised
  - enables task to complete critical resource usage at earliest
  - without preemptions from intermediate priority tasks
  - When several tasks waiting for resource
  - task holding resource inherits highest priority of all waiting
  - As priority of low priority task holding resource equalled to all waiting
  - intermediate priority tasks can't preempt it
  - Thus unbounded priority inversion is avoided
  - Task on inheriting priority of waiting higher priority releases resource
  - gets back its original priority value if holding no other critical resources
  - it inherits priority of highest priority task waiting for resources it helds

Priority Inversion Protocol

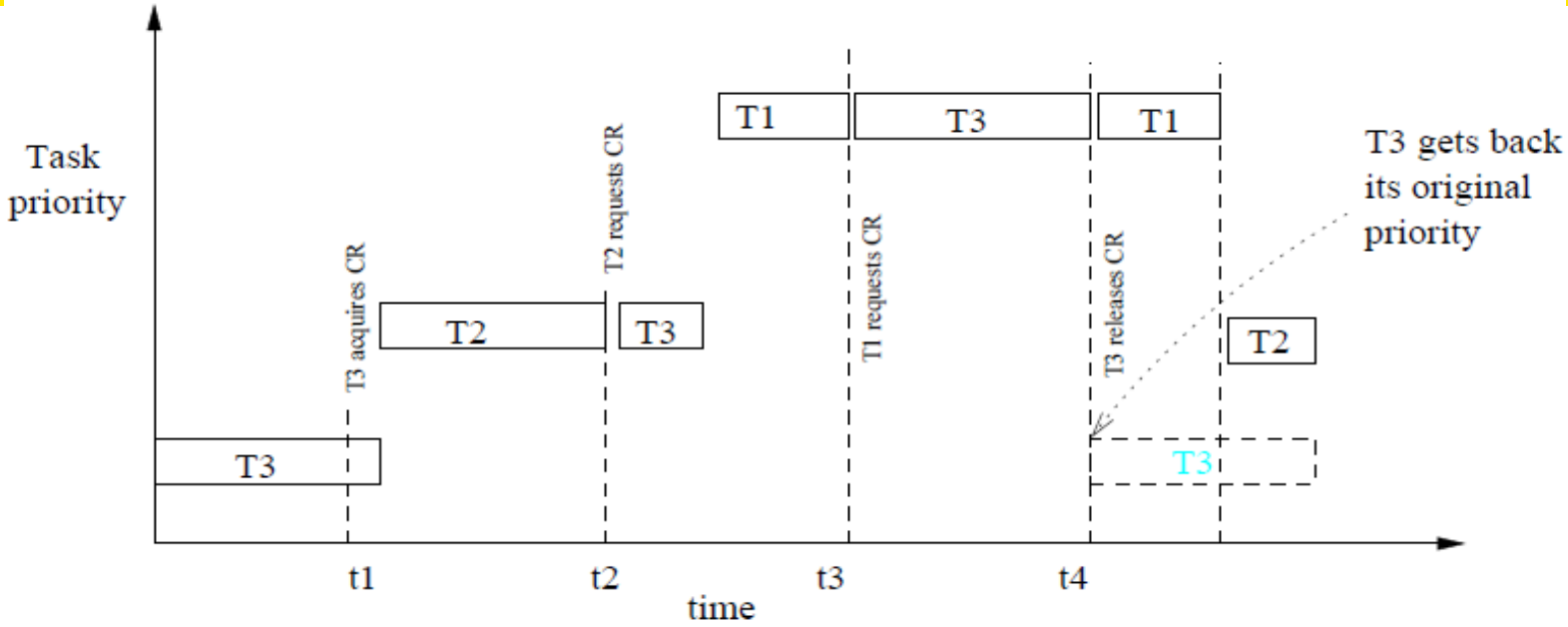
if *the required resource is free* **then**
  grant it;
end
if *the required resource is being held by a higher priority task* **then**
  wait for the resource;
end
if *the required resource is held by a lower priority task* **then**
  wait for the resource;
  the low priority task holding the resource acquires highest priority
    of all tasks waiting for the resource;
end

Working of Priority Inversion Protocol



- Executing task shaded, blocked task unshaded
- Priority of Ti and Tj change due to priority inheritance
- prior(Ti) = 5, prior(Tj) = 10, critical resource CR
- Scenario 1: Ti executes, acquires CR
- Scenario 2: Tj is ready, higher priority, executes
- Scenario 3: Tj blocked requesting CR, Ti inherit's Tj's priority
- Scenario 4: Ti unlocks CR, gets original priority, Tj executes with CR

Priority Changes Under Priority Inversion Protocol



- T3 initially locks CR, preempted by T2, T2 requests CR at t2
- T3 holds CR, T2 blocks, T3 inherits priority of T2
- T2 and T3 at same priority levels
- Before T3 complete use of CR, it is preempted by higher priority T1
- T1 requests CR at t3 and T1 blocks as CR still held by T3
- At this point T2 and T1 waits for CR
- T3 inherits priority of highest priority waiting task T1
- T3 completes CR usage at t4, releases CR, back to original priority

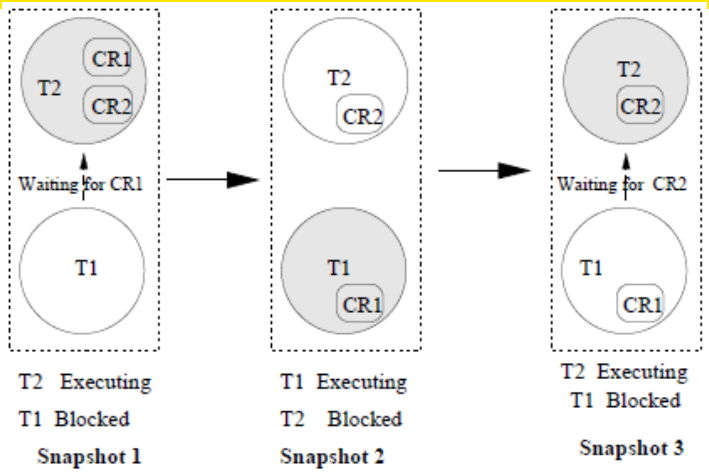
Priority Inheritance Protocol

- A lower priority task retains inherited priority
- Until it holds resource required by waiting higher priority task
- When more than one higher priority task waiting for same resource
- Task holding resource inherits maximum of priority of all waiting tasks
- Real-time tasks share resources without unbounded priority inversion
- Two important problems: dealock and chain blocking
- PIP is susceptable to chain blocking and doesn't prevent deadlocks

Deadlock

- Consider sequence of actions by tasks T1 and T2
- Need shared critical resources CR1 and CR2
  - T1: Lock CR1, Lock CR2, Unlock CR2, Unlock CR1
  - T2: Lock CR2, Lock CR1, Unlock CR1, Unlock CR2
- Assume prior(T1)>prior(T2)
- T2 starts running first, locks CR2 when T1 is not ready
- T1 arrives, preempts T2, starts executing
- T1 locks CR1, tries to lock CR2 held by T2
- T1 blocks, T2 inherits T1's priority by PIP
- T2 resumes execution, needs to lock CR1 held by T1
- Both T1 and T2 are deadlocked

Chain Blocking



- Each time a task needs a resource it undergoes priority inversion
- A task needs n resources for computation
- Might undergo priority inversion n times to acquires all resources
- Assume T1 needs several resources, prior(T1)>priot(T2)
- T2 holds CR1 and CR2, T1 arrives and request to lock CR1
- T1 undergoes priority inversion, T2 inherits T1's priority
- T2 releases CR1, gains original priority, T1 locks CR1
- T1 requests to lock CR2 held by T2, undergoes priority inversion
- T1 waits until T2 releases CR2
- Multiple priority inversion to lock resources leads to chain blocking

Highest Locker Protocol (HLP)

- An extension of PIP, overcomes some shortcomings of PIP
- Every critical resource assigned a *ceiling priority* value
- Critical priority of a critical resource max. priority all tasks req. it
- Task acquires res., its prior. immediately equalled to ceil. prior. of res.
- If task holds multiple resources, inherits max. ceil. prior of resource
- Implicit assumption - res. reqd. by all task known before compile time
- Ceil(Ri) is ceiling priority of a resource Ri, pri(Tj) is priority of task Tj
- FCFS policy
  - a task runs to completion while equal priority task waits
  - for equal priority tasks, Ceil(Ri)=max({pri(Tj)/Ti needs Ri})
  - holds when higher priority values indicate higher priority (Windows OS)
  - If higher priority values indicate lower priority (Unix)
  - then ceiling priority is min. prior. of all tasks needing resource (Unix)
  - That is Ceil(Ri)=min({pri(Tj)/Ti needs Ri})

Highest Locker Protocol (HLP)

- Time-sliced Round Robin policy
  - Equal priority tasks execute for one time slice in round-robin fasion
  - For larger priority value indicates higher priority
  - Ceil(Ri)=max({pri(Tj)/Ti needs Ri}) + 1
  - For larger priority value indicates lower priority
  - Ceil(Ri)=min({pri(Tj)/Ti needs Ri}) + 1
- CR1 shared by T1, T5 and T7, CR2 shared by T5 and T7
- pri(T1)=10, pri(T2)=5, pri(T7)=2
- Priority of CR1 maximum of priorities of T1, T5 and T7
- Ceil(CR1)=max({10,5,2})=10
- As soon as T1, T5, or T7 acquires CR1, its priority raised to 10
- Rule of inheritance of priority
  - Any task that acquires resource inherits corresponding ceiling priority
- If task holds more than one resource
  - its priority is maximum of all ceiling priorities of all resources holding it
  - Example: Ceil(CR2)=max({5,2})=5
- A task holding both CR1 and CR2
  - inherit larger of two ceiling priorities, i.e. 10

Highest Locker Protocol

- Soon after acquiring a resource operates a task at the ceiling priority
- Eliminate unbounded priority inversions, deadlock, chain blocking
- Theorem 1: When HLP is used for resource sharing, once a task gets a resource required by it, it is not only blocked any further.
  Proof: Let T1 and T2 need to share CR1 and CR2
  T1 acquires CR1, T1's priority becomes Ceil(CR1) by HLP
  T1 requires CR2, T2 already holding CR2, pri(T2) raised to Ceil(CR2)
  Ceil(CR2)>pri(T1) because Ceil(CR2) = max({pri(T1),pri(T2)}) + 1
  T2 being higher priority should execute, T1 should not have run
  Contradiction to assume T1 executing while T2 holding CR2
  Similarly, when T1 acquires resource, all res. reqd. by T1 must be free
  A task blocks once for all its res. req. for any set of tasks & res. req.

Highest Locker Protocol

- By Theorem 1 HLP tasks are *single blocking*
- A task is blocked atmost once for all resource requests
- Once task get res. may preempt by higher prior. task not sharing res.
- "single blocking" - blocking on account of resource sharing
- No deadline and chain blocking under HLP is valid
  - once a task releases a resource, it does not acquire any further
- Request and release phases for a task
  - First it acquires all resources it needs
  - Then releases resources
- Corollary 1: Under HLP, before a task can acquire one resource, all the resources that might be required by it must be free
- Corollary 2: A task cannot undergo chain blocking in HLP

HLP vs. PIP

- In PIP several tasks req. a CR in use, queued in order of request
- When a CR becomes free, longest waiting task in queue granted CR
- Every CR is associated with a queue of waiting tasks
- In HLP no such queue is needed
- When a task req. CR, it executes Ceil(CR)
- Other tasks needing CR do not get chance to execute and req. for CR
- HLP prevents deadlocks
- By Corollary 2
- When a task gets a CR all other resources required by it must be free

Shortcomings of HLP

- HLP solves - unbounded prior. inver., deadlock, chain blocking
- Opens up possibility for inheritance-related inversion
- Inheritance-related inversion occurs
  - when priority value of low priority task holding resource
  - is raised to a high value by a ceiling rule
  - intermediate tasks not needing resource
  - undergoes inheritance-related inversion
- Let pri(T1)=1, pri(T2)=2, pri(T3)=3, pri(T4)=4, pri(T5)=5
- pri(T1)<pri(T2)<pri(T3)<pri(T4)<pri(T5)
- T1, T2, T3 need CR1; T1, T4, T5 need CR2
- Ceil(CR1)= max(1,2,3)=3, Ceil(CR2)= max(1,4,5)=5
- T1 acquires CR2 pri(T1)=5, T2 & T3 not run due lower priority
- In such situation T2 and T3 undergo inheritance-related inversion

HLP usage

- HLP is rarely used in real-life applications
- Problem of inheritance-related inversion often severe
- Makes protocol unstable
- Prior. of very low prior. tasks raised to very high while res. acquiring
- Inter. prior. tasks not req. any res. undergo inherit-related invers.
- Miss their deadline
- HLP is foundation of *priority ceiling protocol* (PCP)
- PCP widely used in real-time application developments

Priority Ceiling Protocol (PCP)

- Extends PIP and HLP
- Solves - bounded priority inversion, chain blocking, deadlocks
- Minimizes inheritance-related inversions
- PIP is a greedy approach unlike PCP
- In PIP when request for resource is made, it is allocated if free
- In PCP request may not be granted even if it is free
- PCP associates Ceil(CRi) with every CRi
- Maximum of all priority values of all tasks might use CRi
- An OS variable CSC (Current System Ceiling) used
- CSC keeps track of max. ceil. value of all resources in use at any time
- CSC=max({Ceil(CRi)/CRi is currently in use})
- At system start, CSC initialized 0 - lower prior. than least prior. task

Resource Sharing under PCP

- Two rules handle resource requests - resource grant and resource release
- Resource grant rule - Two clauses applied when Tj req. to lock CRi
  1. Resource Request Clause
     (a) If Tj holding CRi and Ceil(CRi)=CSC, then Tj granted access to CRi
     (b) o.w., Tj not granted CRi, unless pri(Tj)>CSC
     in both (a) and (b)
     If Tj granted CRi & if CSC<Ceil(CRi), then CSC←Ceil(CRi)
  2. Inheritance clause
     - When Tj prevented locking Ri by failing to meet resource grant clause
     - Tj blocks and Tk holding Ri inherits pri(Tj) if pri(Tk)<pri(Tj)
- Resource Release Clause
  - If Tj releases CRi and Ceil(CRi)=CSC, then
    CSC=max({Ceil(CRI)/CRI is currently in use $\forall l \neq i$ })
    else CSC remains unchanged
- Tj releasing CRi gets back its original prior,
  or max. prior waiting tasks which it is still holding,
  whichever is high

PCP vs. HLP

- PCP is very similar to HLP
- In PCP - Tj granted CRi, Tj does not acquire Ceil(CRi) immediately
- In PCP - pri(Tj) does not change on acquiring CRi, only CSC changes
- pri(Tj) changes by inheritance clause of PCP only when
  - one or more tasks wait for CRi held by Tj
- Tasks requesting CRi block under identical situations for PCP & HLP
- Only difference with PCP, Tj can be blocked from entering critical
  - if $\exists$ CRi held by Tk, Ceil(CRi)$\geq$pri(Tj)
- Prevents unnecessary inheritance blockings caused due to
  - pri(Tj) acquiring CRi raised to very high Ceil(CRi) on acquiring CRi
- In PCP instead of raising prior(Tj) on acquiring CRi, CSC$\leftarrow$Ceil(CRi)
- Comparing CSC with pri(Tj) requesting CRi - avoid deadlocks
- If no comparison with CSC as in PIP then
  - a higher pri(Tk) may later lock CRi required by Tj
  - leads to deadlock, where each task holds resources required by other

Example 1

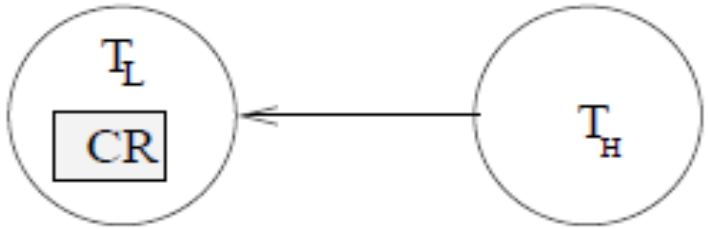
Consider a system consisting of four real-time tasks T1, T2, T3, T4. These four tasks share two non-preemptable resources CR1 and CR2. Assume CR1 is used by T1, T2 and T3, and CR2 is used by T1 and T4. Assume that the priority values of T1, T2, T3, and T4 are 10, 12, 15 and 20, respectively. Assume FCFS scheduling among equal priority tasks, and that higher priority values indicate higher priorities.

- Solution. Ceil(CR1)=max(pri(T1),pri(T2),pri(T3))=15, Ceil(CR2)=max(pri(T1),pri(T4))=20.
  Consider an instant in exection of system in which T1 is executing after acquiring CR1. When T1 acquires CR1, CSC=Ceil(CR1)=15.
  Two alternate situations might arise during execution of tasks.
  Situation 1: Assume T4 ready.
  pri(T4)>pri(T1), preempts T1 and starts to execute.
  After sometime T4 requests CR2. pri(T4)>CSC (i.e. 20>15), T4 granted CR2 by the resource clause. CSC set to 20.
  When T4 completes execution, T1 gets a chance to execute.

Example 1 (continued)

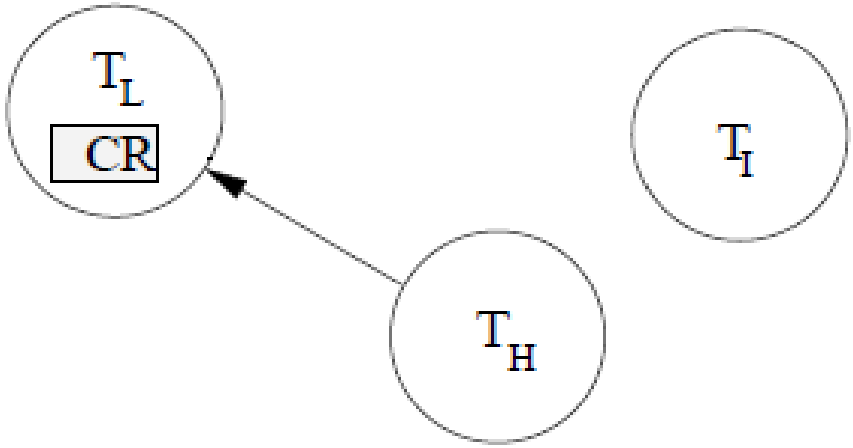
- Situation 2: Assume T3 ready.
  pri(T3)>pri(T1), T3 preempts T1 and starts to execute.
  After sometime T3 requests CR1. pri(T3)$\not>$CSC (i.e. 15$\not>$15),
  T3 not granted CR1. T3 is blocked.
  T1 inherits pri(T3) by PCP inheritance clause.
  pri(T1) changes from 10 to 15.

Different types of Priority Inversions under PCP



- Direct Inversion
  - Occurs when high priority TH waits for lower priority TL to release CR
  - TH waits till TL finishes CR usage and releases it
  - TL directly causes TH undergo inversion by holding CR needed by TH

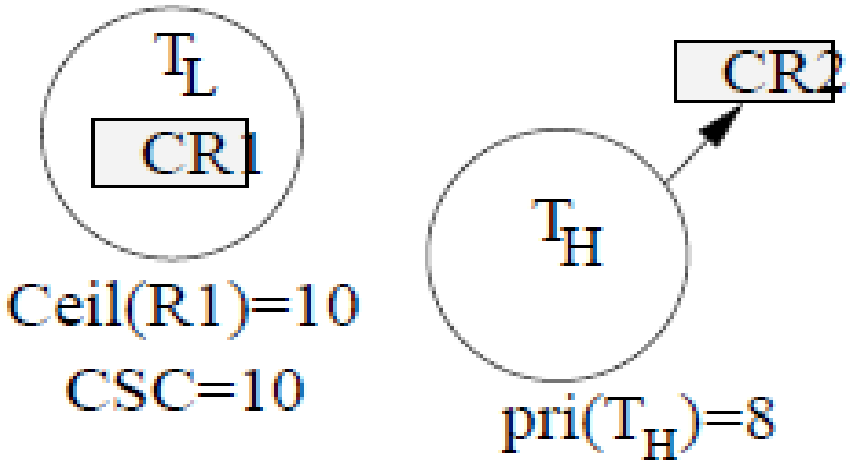
Inheritance-Related Inversion



- TL and TL needs CR, TI does not need CR
- TL acquires CR
- CSC set to Ceil(CR) by resource request inversion clause
- Later TH requests CR, TH blocked on CR by request resource clause
- By inheritance clause, TL inherits pri(TH)
- TI not needing CR cannot execute due to raised pri(TL)
- TI undergoes inheritance-related inversion

Avoidance-Related Inversion

- pri(Tj)>pri(Tk), Tk currently executing
- Tj needs CR not in use, not granted access as pri(Tj)<CSC
- Tj undergoes avoidance-related inversion



- TL and TH need CR1 and CR2 during computation
- TL presently using CR1
- When CR1 granted CSC is set to Ceil(CR1)
- When TH requests CR2, TH blocks since pri(TH)<CSC
- Not allowing TH to access CR1 precludes the possibility
  - TH may request CR1 and TL may request CR2
- Avoidance-related inversion avoid deadlocks

Avoidance-Related Inversion - when no deadlock is possible

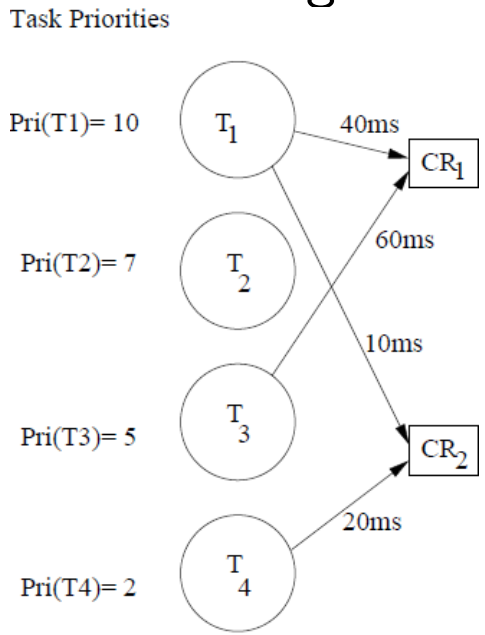
- Tasks T1, T2, T3, T4
- pri(T1)=2, pri(T2)=4, pri(T3)=5, pri(T4)=10
- T1 and T4 share CR1, T2 and T3 share CR2
- Ceil(CR1)=10 and Ceil(CR2)=5
- T1 first acquires CR1
- CSC$\leftarrow$Ceil(CR1)=10 by resource grant clause
- If T2 requests CR2, it is refused access since pri(T2)$<$CSC
- T2 suffers avoidance inversion
- Though no deadlock is possible
- Even if T2 permitted access to CR2

Duration of task inversion

- Assume: Once Tj releases CRi, Tj does not acquire any other CR
- Resource acquire phase - Tj acquires CRs, no release
- Resource release phase - Tj releases CRs, no acquiring
- Unless tasks follow this
- Difficult to determine a quantitative bound on inversion time

Example 2

A system has four tasks: T1, T2, T3, T4. These tasks need two critical resources CR1 and CR2. Assume that the priorities of the four tasks are as follows: pri(T1)=10, pri(T2)=7, pri(T3)=5, pri(T4)=2. The four tasks: T1, T2, T3 and T4 have been arranged in decreasing order of their priorities. That is, pri(T1)>pri(T2)>pri(T3)>pri(T4). The exact resource requirements of these tasks and the duration for which the tasks need the two resources have been shown in the figure.



Compute the different types of inversions that each task might have to undergo in the worst case.

Example 2 - Solution

T1 requires CR1 for 40 msec and CR2 for 10 msec.
T3 requires CR1 for 60 msec and T4 requires CR2 for 20 msec.
Assume FCFS scheduling among equal priority tasks.
Ceil(CR)=max({pri{Ti}/Ti may need CR})
Ceil(CR1)=max({10,5})=10 and Ceil(CR2)=max({10,2})=10

- T1 can suffer direct inversion
  - due to CR1 by T3 for 60 msec or
  - due to CR2 by T4 for 20 msec
- No inheritance-related and deadlock-avoidance inversions for T1
  - since pri(T1) is highest
- T2 will not suffer direct inversion since it does not need any CR
- T2 can suffer inheritance-related inversion
  - due to T3 for 60 msec when T3 acquires CR1 and T1 waits for CR1
  - due to T4 for 20 msec when T4 acquires CR2 and T1 waits for CR2
- T2 cannot suffer deadlock-avoidance inversion
  - since it does not require any CR

Example 2 - Solution (continued)

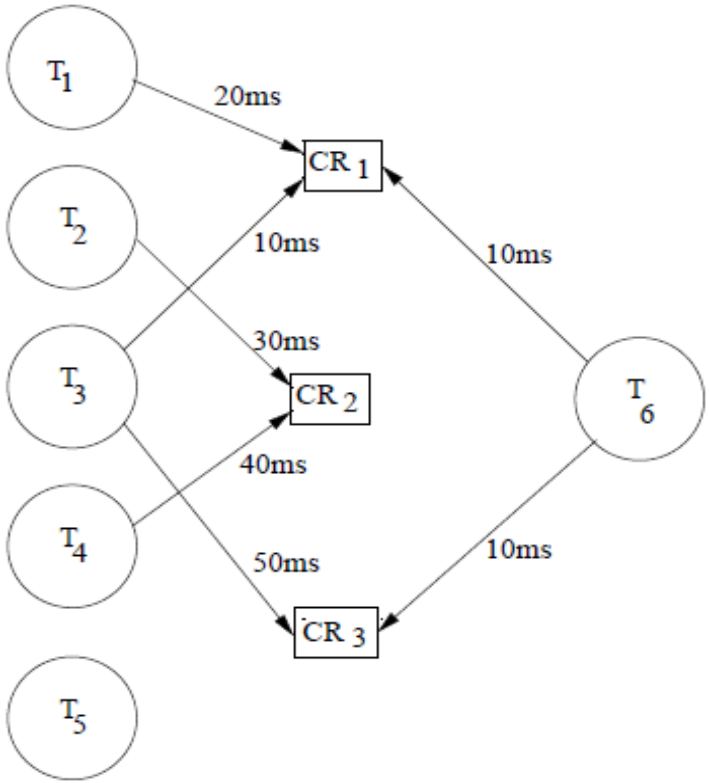
- T3 will not suffer direct inversion
  - since it does not share any CR with a lower priority task
- T3 can suffer inheritance-related and deadlock-avoidance inversions
  - due to T4 for atmost 20 msec
- T4 will not suffer any priority inversion as pri(T4) is lowest
- Maximum duration for each task suffering inversions

Priority Inversions

Task	Direct			Inheritance			Avoidance		
	T_2	T_3	T_4	T_2	T_3	T_4	T_2	T_3	T_4
T_1	x	60	40	x	x	x	x	x	x
T_2	x	x	x	x	60	20	x	x	x
T_3	x	x	x	x	x	20	x	x	20

Example 3

Let us consider a system with a set of periodic real-time tasks T1,. . . ,T6. The resources and computing requirements of these tasks have been shown in the figure



Assume that the tasks T1,. . . ,T6 have been arranged in decreasing order of their priorities. Compute the different types of inversions that a task might have to undergo.

Example 3 - Solution

- Maximum duration each task suffers different inversions

Task	Direct					Inheritance					Avoidance				
	T_2	T_3	T_4	T_5	T_6	T_2	T_3	T_4	T_5	T_6	T_2	T_3	T_4	T_5	T_6
T_1	x	10	x	x	10	x	x	x	x	x	x	x	x	x	x
T_2	x	x	40	x	x	x	10	x	x	10	x	10	x	x	10
T_3	x	x	x	x	10	x	x	40	x	10	x	x	40	x	10
T_4	x	x	x	x	x	x	x	x	x	10	x	x	x	x	10
T_5	x	x	x	x	x	x	x	x	x	10	x	x	x	x	x

Properties of an inversion table

- Each inversion table is an upper triangular matrix
  - all lower triangular items are zero
  - since lower priority tasks do not suffer inversions due to higher
- Avoidance table is similar to inheritance table
  - except when a task does not need any resource
  - tasks not needing any resource never request
  - they cannot suffer avoidance inversions
- A task suffer at best one of direct, inheritance, or avoidance inver.
  - from Corollary 1 Theorem 2
- Total duration for which Ti may be blocked by lower priority tasks
  - bti=max{(bidj),(biij),(biaj)}
  - bidj is blocking Ti by lower priority Tj due to direct inversion
  - biij is blocking Ti by lower Tj due to inheritance-related inversion
  - biaj is blocking Ti by lower Tj due to avoidance-related inversion
- bti based on largest entry in corresponding row of Ti
- Response time of Ti = ei+bssi+bti+$\sum$ceil(pi/pj)*ej, 1$\leq$j$\leq$n-1

Important features of PCP

- Theorem 2 Tasks are single blocking under PCP.

PROOF Consider Ti needs a set of resources SR = {Ri}.
For each Ri Ceil(Ri)≥ pri(Ti). Assume Ti acquires Ri, Tj already holding Rj, and Ri, Rj ∈ SR. This leads Ti to block after acquiring Ri. When Tj locked Rj CSC should have been set to atleast pri(Ti), by resource grant clause of PCP. Ti could not have been granted Ri. This is a contradiction with the premise. Therefore, when Ti acquires Ri all resources required by Ti must be free. Once Ti acquires Ri, it cannot undergo inheritance-related inversion. Assume lower pri(Tk) holding Rz and higher pri(Th) waiting for Rz. This is not possible as Ceil(Rz)≥pri(Th). Thus CSC≥pri(Th).
Ti prevented Ri access in first place. This is a contradiction with the premise. Therefore, it is not possible Ti undergoes inheritance-related inversion after it acquires Ri. Similarly, Ti cannot suffer avoidance inversion after acquiring Ri. Thus once Ti acquires Ri it can't undergo any inversion $\implies$ PCP tasks are single blocking.

- Corollary 1. Under PCP a task can undergo atmost one inversion during its execution

How is deadlock avoided in PCP ?

- Deadlock - tasks holds each other's required resources at same time
- Under PCP by Theorem 2
  - When one task is executing with some resources
  - other tasks cannot hold a resource that may ever be needed by the task
- When a task granted a resource, all its required resources must be free
- This prevents the possibility of any deadlock

How is unbounded priority inversion avoided ?

- A higher priority task suffers unbounded priority inversion
  - when waiting for a lower priority task to release resources required by it
  - intermediate priority tasks preempt low priority task from CPU usage
- In PCP
  - whenever a higher priority task waits for resouces used by lower
  - executing lower priority task is made to inherit priority of higher
  - so intermediate priority tasks cannot preempt lower from CPU usage
  - Therefore, unbounded priority inversions cannot occur under PCP

How is chain blocking avoided in PCP ?

- By Theorem 2,
  - resource sharing among tasks under PCP is single blocking
- Therefore, chain blocking cannot occur under PCP

Using PCP in Dynamic Priority Systems

- So far assumed task priorities are static
- Task's priority does not change for its entire lifetime
- In fynamic priority systems priority of a task might change with time
- Ceil(CR) needs to be recomputed each time a task's priority changes
- CSC and inherited priority values of tasks holding CR need change
- Introduces a high run-time overhead
- Use of PCP in dynamic priority systems unattractive

Comparison of Resource Sharing Protocols

- Priority Inheritance Protocol (PIP)
  - Simple protocol
  - Overcomes unbounded priority inversion
  - Requires minimal support from operate system
  - Suffers from chain blocking
  - Does not prevent deadlocks
- Highest Locker Protocol (HLP)
  - Requires moderate support from operating system
  - Solves chain blocking and deadlock of PIP
  - Intermediate priority tasks undergo large inheritance-related inversions
  - Causes intermediate priority tasks to miss deadlines
- Priority Ceiling Protocol (PCP)
  - Overcomes shortcomings of PIP and HLP
  - Free from deadlock and chain blocking
  - Priority of task changed when higher priority task request resource
  - Suffers much lower inheritance-related inversions that HLP

Handling Task Dependencies

- So far tasks considered as independent
- No constraint on order in execution of different tasks
- Practical situations one task need results from another
- Tasks might have to carried out in certain order for proper functioning
- Develop a schedule for set of tasks used in table-driven scheduling

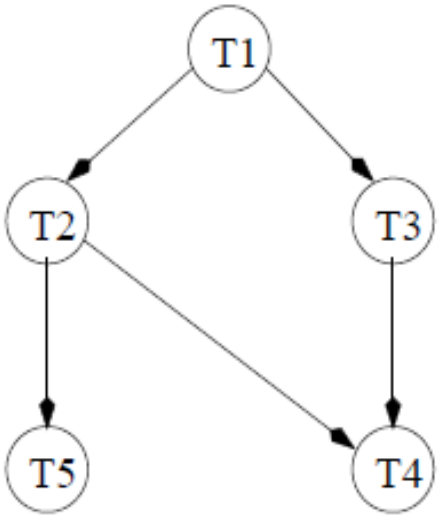
Table-driven Algorithm

- Given a set of periodic real-time tasks with dependencies
- Main steps of a simple algorithm for determining a feasible schedule
  1. Sort task in increasing order of their deadlines, without violating any precedence contraints and store the sorted tasks in a linked list.
  2. Repeat
     - Take up the task having largest deadline and not yet scheduled (i.e., scan the task list of step 1 from left). Schedule it as late as possible.

     until all tasks are scheduled.
  3. Move the schedule of all tasks to as much left (i.e., early) as possible without disturbing their relative positions in the schedule.

Example 4

Determine a feasible schedule for the real-time tasks of a task set {T1,T2,. . . ,T5} for which the precedence relations have been given for use with a table-driven scheduler. The execution times of the tasks T1,T2,. . . ,T5 are: 7, 10, 5, 6, 2 and the corresponding deadlines are 40, 50, 25, 20, 8, respectively.



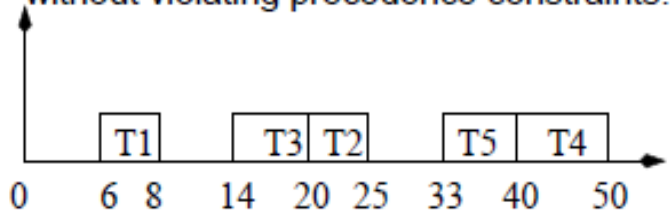
Precedence Relationship Among Tasks

Solution

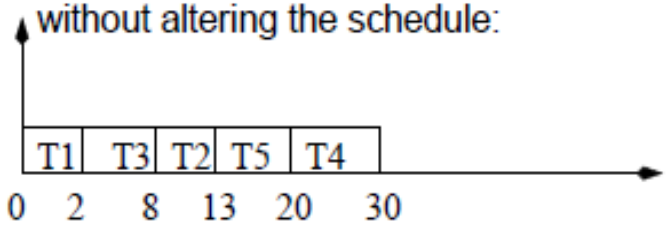
Step1: Arrangement of tasks in ascending order:

T1 T3 T2 T5 T4

Step 2: Schedule tasks as late as possible without violating precedence constraints:



Step 3: Move tasks as early as possible without altering the schedule:



EDF and RMA-based Schedulers

- Task precedence contraints can be handled in both EDF and RMA
- Require following modification to algorithm
  - Do not enable a task until all its predecessors complete their execution
  - Check tasks waiting to be enabled after every task completes
- Achievable schedulable utilization of tasks with dependencies
  - lower compared to utilization achieved by independent tasks

Thank you