SYSC 5701
Operating System Methods for Real-Time Applications

Access Control: PCP

Winter 2014

Resource-Sharing Dependencies

- **A job cannot proceed (is blocked) because of resource-sharing synchronization**
- **Resource-sharing requires mutually exclusive access to the resource**
- **Can cause priority inversions**

Resources

- **serially reusable "units" of resource**
  - **eg. binary semaphore has one unit**
    - **counting semaphore has *count* units**
- **grant mutually exclusive right to access a unit**
- **once a unit is granted to a job, must not be reused by other jobs until released**
- **Recall management of mutual exclusion "unit" in monitors!**

Access to Resources

- **job requests resource(s)**
  → job "*locks*" the resource(s)
- **lock is managed by o/s (kernel)**
- **if resource(s) not available job is blocked**
- **eventually, job is granted the resource(s) and is unblocked**
- **when finished with resource(s)**
  → job "*unlocks*" resource(s) for reuse

Access (more)

related material from earlier in course:

- **semaphores, IPC**
- **monitors**

constructs to enable programs to control locking and unlocking of resources

- **critical sections** — parts of programs that require locking
- **mutual exclusion** — the desired effect

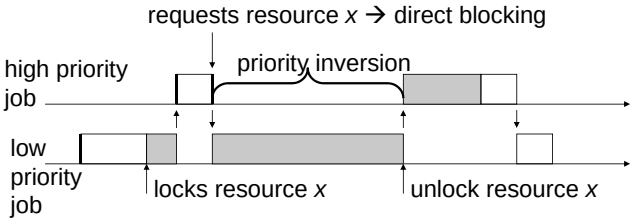
Result: task can have <u>interdependencies</u> when accessing resources

Access Control Protocol

- **resource conflict:**
  two jobs require same resource type
  - **jobs must contend for the resource**
- **access control protocol: set of rules for**
  1. **granting resources**
  2. **scheduling jobs requesting resources**

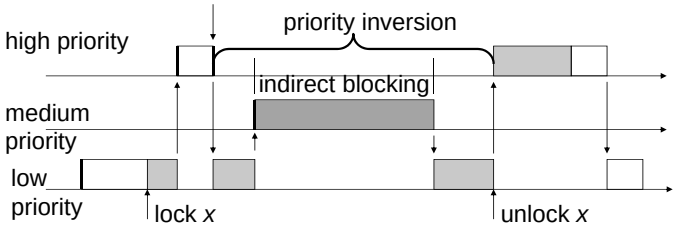
Priority Inversion

- **a higher priority job is prevented from executing by a lower priority job**
  - **the priority relationship is inverted!**



Unbounded Priority Inversion

- **duration of priority inversion is not a function of the time for low priority job to execute the relevant critical section**



Worst Case Job Response Time

- **preemption time: delay due to higher priority job**
- **execution time: time to do job's work**
- **blocking time: time spent blocked**
  - **hopefully, blocking time is a simple function of delays while lower-priority jobs execute critical sections**
    - **if not, then difficult to compute (unbounded)**

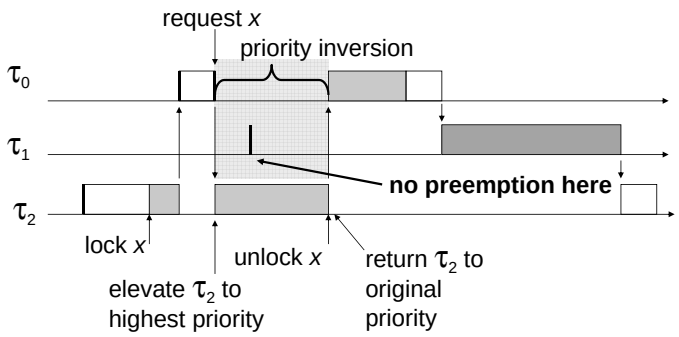
Avoiding Unbounded Priority Inversion

1. **disable preemption**
2. **priority inheritance protocol**
3. **priority ceiling protocol**

Disable All Preemption

- **disable preemption during critical sections**
- **effectively elevate job in critical section to highest priority (cannot be preempted)**
- **priority elevation only needed when higher-priority jobs are requesting the relevant critical section – in other cases, the lower priority job should be preemptable by higher-priority jobs**
- **OK if critical sections are <u>very short</u> relative to shortest deadlines**

Disable Preemption for Unbounded PI Example



Variation on Disable Preemption: Priority Ceiling Emulation

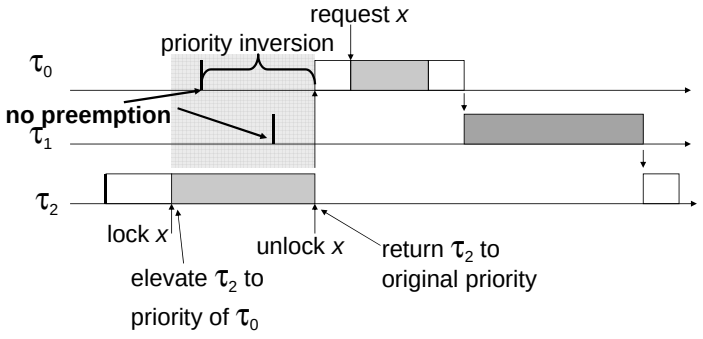
Priority Ceiling:

- the priority ceiling of resource R_i is the highest priority of all jobs that require access to R_i at any time during their operation
- denote $\Pi(R_i)$
- Q: do any jobs with priority higher than $\Pi(R_i)$ access R_i ?

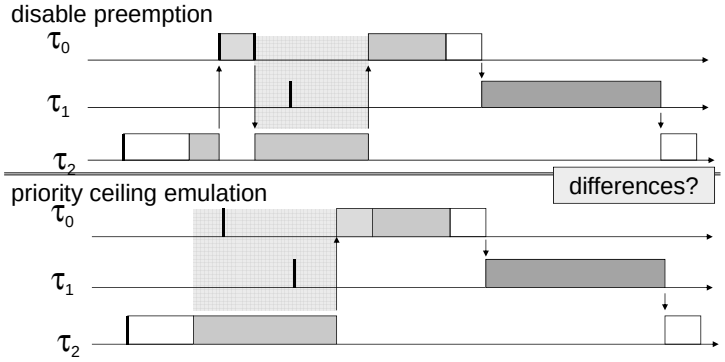
Priority Ceiling Emulation

- in critical section, job runs at priority = priority ceiling for the resource
  - i.e. no job that <u>might</u> request access to the resource is able to run!
- job in critical section disables all jobs that <u>might</u> access critical section
- at end of critical section, job returns to original priority
- jobs at priority higher than the ceiling are still eligible to run

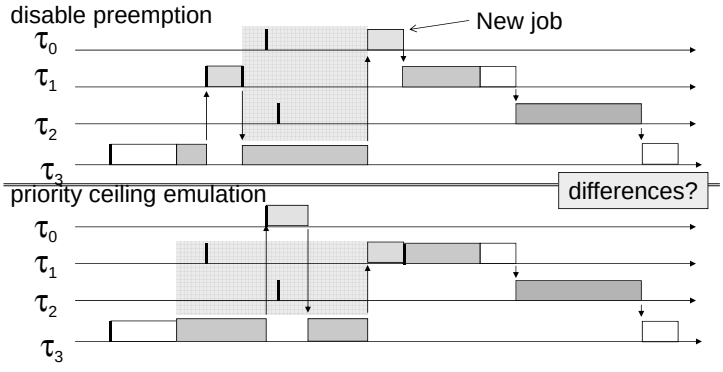
Priority Ceiling Emulation for Unbounded PI Example



Priority Ceiling Emulation <u>vs.</u> Disable Preemption Example 1



Priority Ceiling Emulation <u>vs.</u> Disable Preemption Example 2



Basic Priority Inheritance Protocol

- while a job_{low} is holding any resource: raise its priority to the highest priority of any job requesting any resource held by job_{low}
- dynamic → raise at time higher priority job requests the resource
- when unlock a resource: assign job_{low} the higher of (1) its original priority, or (2) the highest priority of a job requesting a resource held by job_{low}

Lowering Priority Scenario

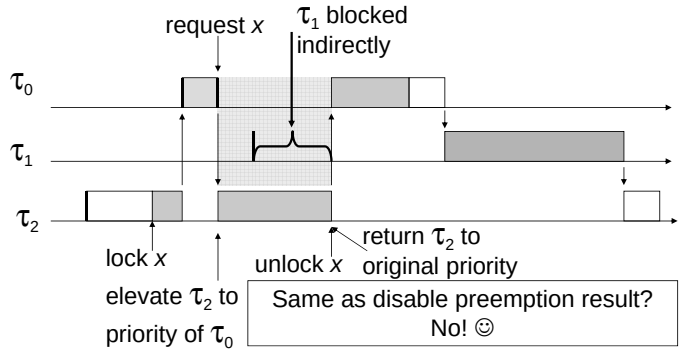
- Suppose job_{low} holds resource 1 and it is then requested by job with priority τ_2

  → raise job_{low} to priority τ_2
- Now job_{low} acquires resource 2 and it is then requested by job with priority τ_1

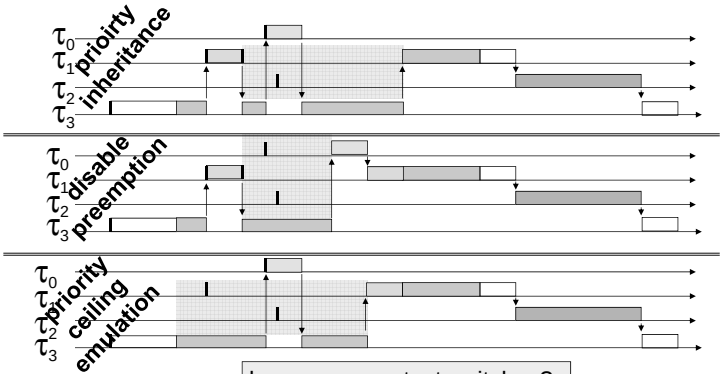
  → raise job_{low} to priority τ_1
- Now job_{low} releases resource 1

  → what should be priority of job_{low} now?
  - How does this fit with the rule on previous slide?

Basic Priority Inheritance for Unbounded PI Example



What about All Three?



how many context switches?

response times?

blocking?

Is Blocking a Function of Time to Execute Critical Sections?

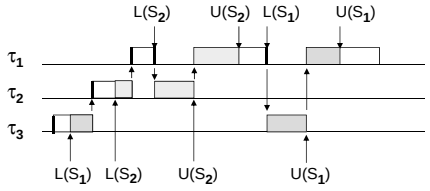
- suppose *m* critical sections accessed by task τ
- job of τ can be blocked directly, at most, m times
  - not counting indirect blocking!
- if *n* tasks at lower priority than τ
  - job of τ can be blocked, at most, at one critical section in each of the n tasks
  - blocking time is bounded, ☺ but ... may suffer from "chain blocking" – blocks each time attempt to access a critical section

Chain Blocking Example

τ_1 : $L(S_2)$ $U(S_2)$ $L(S_1)$ $U(S_1)$

τ_2 : $L(S_2)$ $U(S_2)$

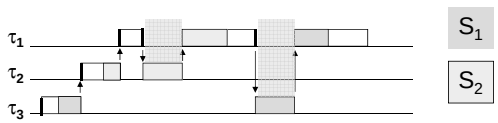
τ_3 : $L(S_1)$ $U(S_1)$



τ_1 blocked each time it tries to access a critical section

Chain Blocking for Disable Preemption & Priority Inheritance?

- still a problem!
- previous example:

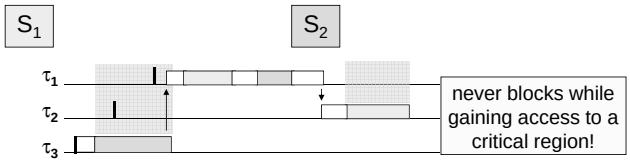


Same behaviour for both? WHY?

Chain Blocking for Priority Ceiling Emulation?

- never blocks on request!
- resource always available (GOOD!) … WHY?
- for previous example:

$\Pi(S_1) = \pi(\tau_1)$ $\Pi(S_2) = \pi(\tau_1)$



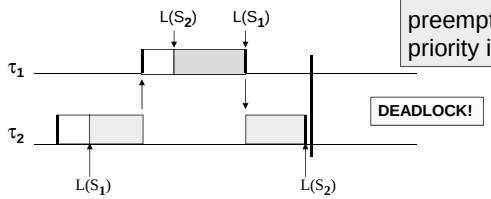
Potential Deadlock

- loop of tasks blocked waiting for each other

τ_1 : $L(S_2)$ $L(S_1)$ $U(S_1)$ $U(S_2)$

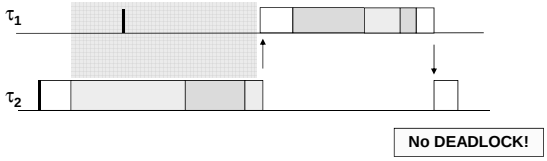
τ_2 : $L(S_1)$ $L(S_2)$ $U(S_2)$ $U(S_1)$

problem for disable preemption & priority inheritance



Deadlock for Priority Ceiling Emulation?

- no!
- resource always available WHY?



Performance vs. Penalty

- priority ceiling emulation looks "best" in terms of performance
  - penalty? → may delay higher priority jobs <u>even though no conflict would occur</u>
    → unnecessary priority inversion!
- disable preemption & priority inheritance
  - only elevate priority when a conflict occurs → avoids <u>unnecessary</u> priority inversion

Basic Priority Ceiling Protocol

- combine priority ceiling emulation with priority inheritance protocol
  - ✓ priority ceiling
  - ✓ inheritance only when conflict

meaning of "conflict" is key!

- current priority ceiling: $\hat{\Pi}(t)$

  highest priority ceiling of <u>all</u> resources <u>currently in use</u>

Basic Priority Ceiling Protocol Rules

Scheduling Rule:

- job released at <u>assigned</u> priority
- preemptive and priority driven at job's current priority

Allocation Rule: when J requests R at time t

- if R already locked – request denied and J blocked

Allocation Rule (con't)

if R is free:

i. **if priority of J at t > $\hat{\Pi}(t)$, allocate R to J**
   - **J does not access any of the held resources!**

ii. **else: if J is the job holding the resource(s) whose priority ceiling = $\Pi(t)$, allocate R to J**
   - **Not possible for different jobs to hold resources with same priority ceiling! (see i. and iii.)**

iii. **otherwise: request denied and J is blocked**

Allocation Rule (paraphrasing)

- **a job <u>cannot</u> acquire a resource unless its priority is higher than the ceilings of all other resources currently acquired by other jobs**
- **if priority higher than ceilings, then job will not request access to any of the other active resources (by the definition of a ceiling!)**
- **when request denied and job is blocked, higher priority jobs <u>might still be able to acquire resource</u>! (deny access ≠ FIFO blocking)**

Priority-Inheritance Rule

- **while a job_{low} is holding any resource: raise its priority to the highest priority of any job requesting any resource held by job_{low}**
- **dynamic: at time t when a job J becomes blocked, the job J_{low} which blocks J inherits the current priority of J**
- **J_{low} executes at inherited priority until t′ when it releases every resource whose priority ceiling is greater or equal to the inherited priority**
- **at t′: priority of J_{low} falls to the higher of (1) its original priority, or (2) the priority of the highest job requesting one of the resources still held by J_{low}**

Lowering Priority Scenario

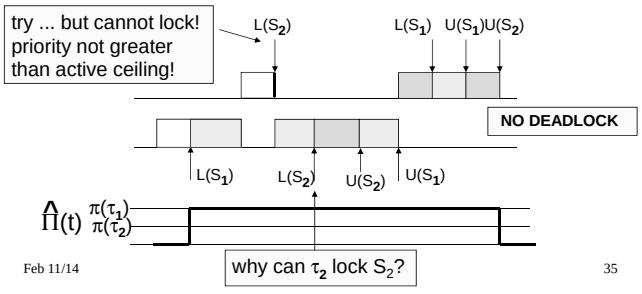
- **Suppose job_{low} holds resource 1 with priority ceiling π_3 which is then requested by job with priority π_4 (rules?)**
  → **raise job_{low} to priority π_4**
- **Now job_{low} acquires resource 2 with priority ceiling π_1 which is then requested by job with priority π_2 (rules?)**
  → **raise job_{low} to priority π_2**
- **Now job_{low} releases resource 1**
  → **what should be priority of job_{low} now? (rules?)**
- **Now job_{low} releases resource 2**
  → **what should be priority of job_{low} now? (rules?)**
- **What if resources released in other order? (rules?)**

Recall Previous Deadlock Example

τ_1 : $L(S_2)$ $L(S_1)$ $U(S_1)$ $U(S_2)$

τ_2 : $L(S_1)$ $L(S_2)$ $U(S_2)$ $U(S_1)$

$\Pi(S_1) = \pi(\tau_1)$ $\Pi(S_2) = \pi(\tau_1)$



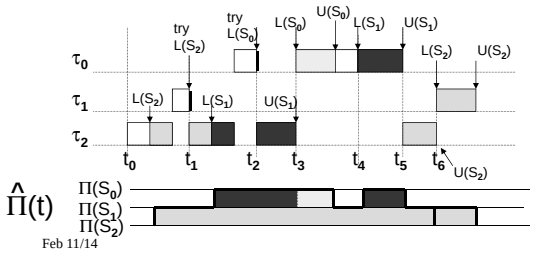
Another Example

τ_0 : $L(S_0)$ $U(S_0)$ $L(S_1)$ $U(S_1)$

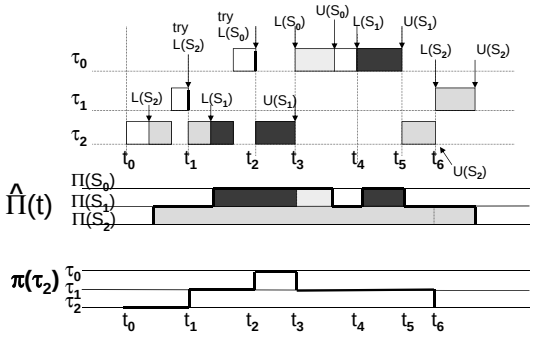
τ_1 : $L(S_2)$ $U(S_2)$

τ_2 : $L(S_2)$ $L(S_1)$ $U(S_1)$ $U(S_2)$

$\Pi(S_0) = \pi(\tau_0)$ $\Pi(S_1) = \pi(\tau_0)$ $\Pi(S_2) = \pi(\tau_1)$



Priority Inheritance in Example



Behaviour

- t_0 : $\pi(\tau_2) = \tau_2$
- t_1 : $\Pi(S_2) = \pi(\tau_1)$ ∴ block, bump τ_2 to $\pi(\tau_1)$
- t_2 : $\Pi(S_1) = \pi(\tau_0)$ ∴ block, bump τ_2 to $\pi(\tau_0)$
- t_3 : $\pi(\tau_2)$ returns to $\pi(\tau_1)$, τ_0 resumes
- t_4 : S_1 free, only other active resource is S_2
  $\Pi(S_2) = \pi(\tau_1) \rightarrow \hat{\Pi}(t_4)$ and $\pi(\tau_0) > \hat{\Pi}(t_4)$
  ∴ allocate S_1 to τ_0
- t_5 τ_2 resumes
- t_6 : $\pi(\tau_2)$ returns to $\pi(\tau_2)$, τ_1 resumes

Points Seen in Example

- a job is blocked by a (possibly nested) critical section of <u>at most</u> one lower priority job
- ceiling blocking occurs – a job is prevented from entering a critical section by ceiling of an active resource → not because the requested resource was busy !
  - e.g. t_2: τ_0 is blocked even though S_0 is free

Properties of Basic Priority Ceiling Protocol

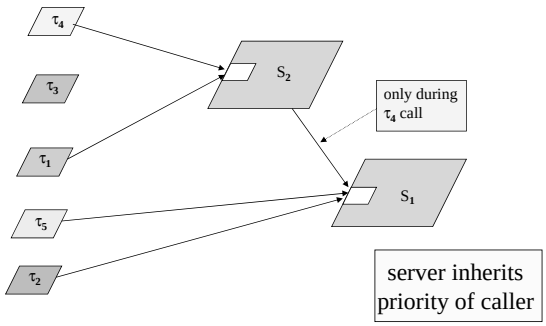
- no deadlock !
- job blocks in at most one critical section
  - blocking is bounded
  - no chain blocking → shorter blocking bound than Priority Inheritance Protocol
- once acquire first resource, all resources needed will be available when requested

Implementation of Basic Priority Ceiling Protocol

- don't need "lock" queues (e.g. semaphore queue)
- maintain queue of tasks that are ready-to-run or blocked – maintain in priority order
- task at head is current task
- need list of active resources – ordered by ceiling priority (includes task that locked the resource, and highest priority of any task blocked waiting for the resource)
- *lock* and *unlock* manipulate queue and list
- need analysis of critical section use – establish priority ceilings prior to run-time

Why is a single queue sufficient?

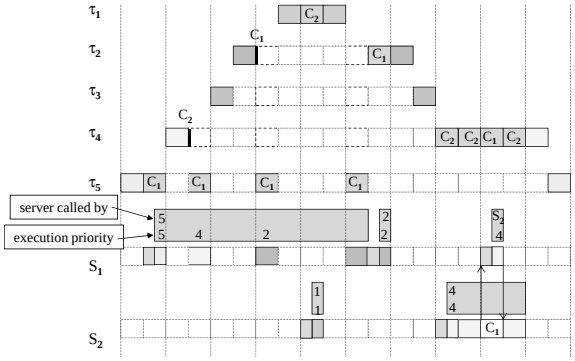
Example Using Client / Server (similar to Liu text)



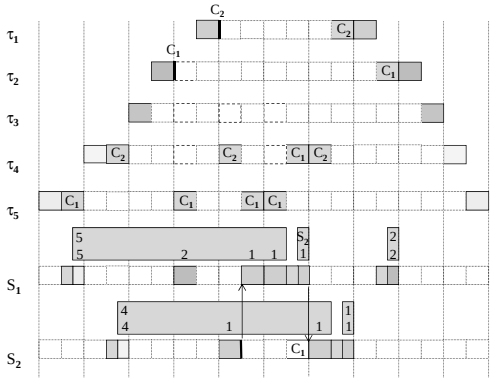
Ceilings in Example

- Ceiling(S_1) = max {τ_2 , τ_4 , τ_5} (τ_4 indirect)
  = Priority(τ_2)
- Ceiling(S_2) = max {τ_1 , τ_4 }
  = Priority(τ_1)
- τ_3 does not access servers

Basic Priority Ceiling Protocol Behaviour



Basic Priority Inheritance Same Example



Response Comparison