Real-Time Scheduling

Resource Access Control

[Some parts of this lecture are based on a real-time systems course of Colin Perkins

http://csperkins.org/teaching/rtes/index.html]

Current Assumptions

- Single processor
- Individual jobs
  (that possibly belong to periodic/aperiodic/sporadic tasks)
  - Jobs can be preempted at any time and never suspend themselves
- Jobs are scheduled using a priority-driven algorithm
  i.e., jobs are assigned priorities, scheduler executes jobs according to these priorities
- n resources $R_1, \ldots, R_n$ of distinct types
  - used in non-preemptable and mutually exclusive manner; *serially reusable*

Motivation & Notation

Resources may represent:

- Hardware devices such as sensors and actuators
- Disk or memory capacity, buffer space
- Software resources: locks, queues, mutexes etc.

Assume a lock-based concurrency control mechanism

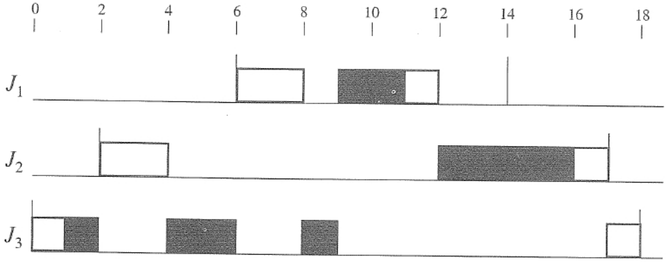
- A job wanting to use a resource R_k executes $L(R_k)$ to lock the resource R_k
- When the job is finished with the resource R_k, unlocks this resource by executing $U(R_k)$
- If lock request fails, the requesting job is **blocked** and has to wait, when the requested resource becomes available, it is unblocked

  In particular, a job holding a lock cannot be preempted by a higher priority job needing that lock

The segment of a job that begins at a lock and ends at a matching unlock is a *critical section* (CS)

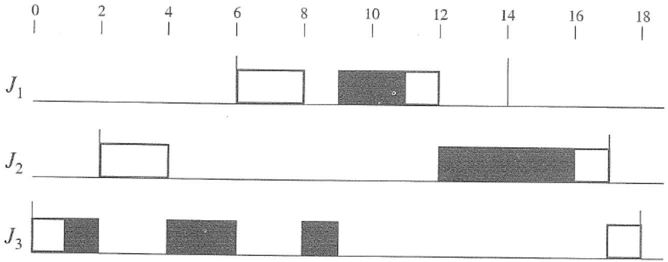
- CS must be properly nested if a job needs multiple resources

Example



- At 0, J_3 is ready and executes
- At 1, J_3 executes $L(R)$ and is granted R
- J_2 is released at 2, preempts J_3 and begins to execute
- At 4, J_2 executes $L(R)$, becomes blocked, J_3 executes
- At 6, J_1 becomes ready, preempts J_3 and begins to execute
- At 8, J_1 executes $L(R)$, becomes blocked, and J_3 executes

Example



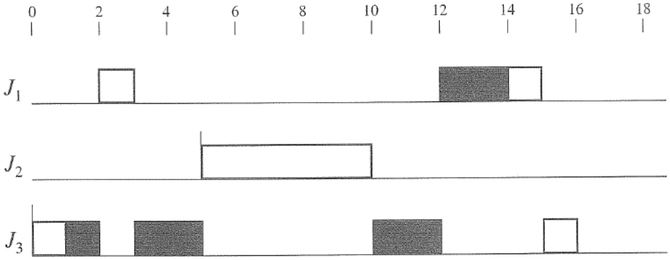
- At 9, J_3 executes $U(R)$ and both J_1 and J_2 are unblocked. J_1 has higher priority than J_2 and executes
- At 11, J_1 executes $U(R)$ and continues executing
- At 12, J_1 completes, J_2 has higher priority than J_3 and has the resource R, thus executes
- At 16, J_2 executes $U(R)$ and continues executing
- At 17, J_2 completes, J_3 executes until completion at 18

Priority Inversion

Priority inversion occurs when a low-priority job executes while some ready higher-priority job waits

Contention for resources can cause priority inversions to occur even if the jobs are preemptable, since a lower-priority job holding a lock on a resource will block a higher-priority job

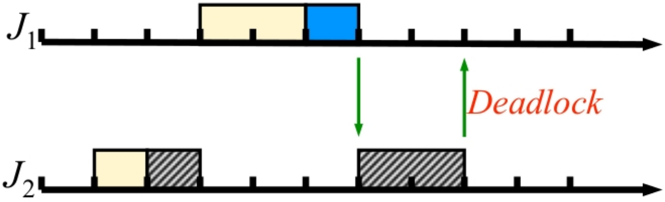
Uncontrolled priority inversion:



High priority job (J_1) can be blocked by low priority job (J_3) for unknown amount of time depending on middle priority jobs (J_2)

Deadlock

Deadlock can result from piecemeal acquisition of resources: classic example of two jobs J_1 and J_2 both needing both resources R and R'



- J_2 locks R' and J_1 locks R
- J_1 tries to get R' and is blocked
- J_2 tries to get R and is blocked

The classic solution is to impose a fixed locking order over the set of lockable resources, and all jobs attempt to lock the resources in that order

Controlling Timing Anomalies

Contention for resources causes timing anomalies due to priority inversion and deadlock

Several protocols exist to control the anomalies

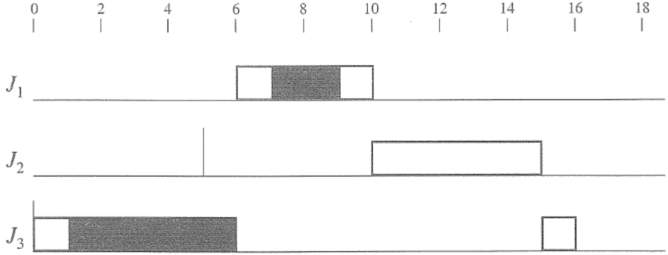
- Non-preemptive CS
- Priority inheritance protocol
- Priority ceiling protocol
- ....

Non-preemptive Critical Sections

The protocol: when a job locks a resource, it is scheduled with highest priority in a non-preemptive manner

Example 1

Jobs J_1, J_2, J_3 with release times $2, 5, 0$, resp., and with execution times $4, 5, 7$, resp.



Disadvantage: every job can be blocked by every lower-priority job with a critical section, even if there is no resource conflict

Priority-Inheritance Protocol

Idea: adjust the scheduling priorities of jobs during resource access, to reduce the duration of timing anomalies

Features:

- Works with any preemptive, priority-driven scheduling algorithm
- Does not require any prior knowledge of the jobs resource requirements
- Does not prevent deadlock, but if some other mechanism is used to prevent deadlock, ensures that no job can block indefinitely due to uncontrolled priority inversion

Priority-Inheritance Protocol

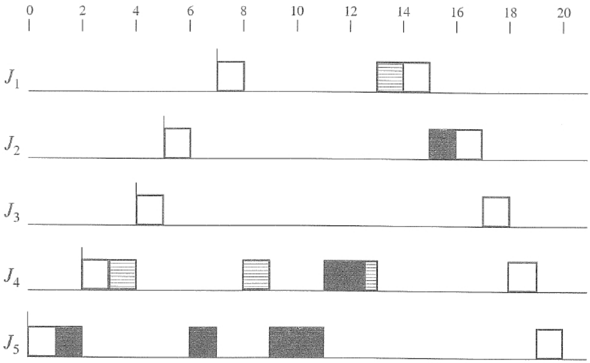
Notation:

- *assigned priority* = priority assigned to a job according to a standard scheduling algorithm
- At any time t, each ready job J_k is scheduled and executes at its *current priority* $\pi_k(t)$ which may differ from its assigned priority and may vary with time
  - The current priority $\pi_k(t)$ of a job J_k may be raised to the higher priority $\pi_h(t)$ of another job J_h
  - In such a situation, the lower-priority job J_k is said to *inherit* the priority of the higher-priority job J_h, and J_k executes at its inherited priority $\pi_h(t)$

Priority-Inheritance Protocol

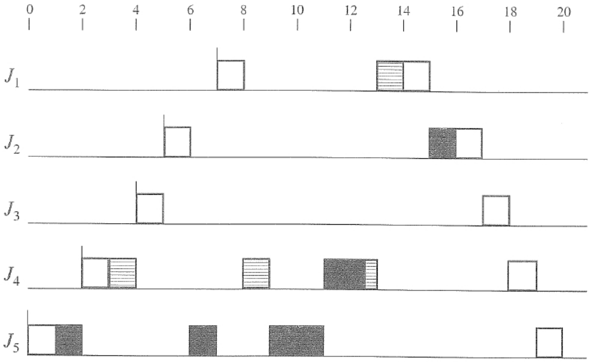
- Scheduling rules:
  - Jobs are scheduled in a preemptable priority-driven manner *according to their current priorities*
  - At release time, the current priority of a job is equal to its assigned priority
  - The current priority remains equal to the assigned priority, except when the priority-inheritance rule is invoked
- **Priority-inheritance rule**:
  - When job J_h becomes blocked on a resource R, the job J_k which blocks J_h inherits the current priority $\pi_h(t)$ of J_h; all priorities are updated transitively
  - J_k executes at its inherited priority until it releases R; at that time, the priority of J_k returns to its priority $\pi_k(t')$ at the time t' when it acquired the resource R
    (note that $\pi_k(t')$ may still be an inherited priority)
- Resource allocation: when a job J requests a resource R at t:
  - If R is free, R is allocated to J until J releases it
  - If R is not free, the request is denied and J is blocked
  - J is only denied R if the resource is held by another job

Priority-Inheritance Example



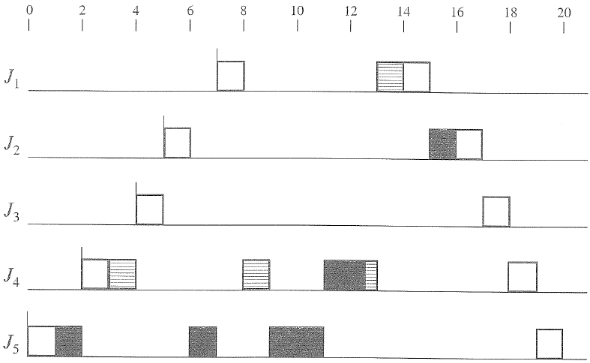
- At 0, J_5 starts executing at priority 5, at 1 it executes $L(\mathit{Black})$
- At 2, J_4 preempts J_5 and executes
- At 3, J_4 executes $L(\mathit{Shaded})$, J_4 continues to execute
- At 4, J_3 preempts J_4; at 5, J_2 preempts J_3
- At 6, J_2 executes $L(\mathit{Black})$ and is blocked by J_5. Thus J_5 inherits the priority 2 of J_2 and executes

Priority-Inheritance Example



- At 8, J_1 executes $L(\textit{Shaded})$ and is blocked by J_4. Thus J_4 inherits the priority 1 of J_1 and executes
- At 9, J_4 executes $L(\textit{Black})$ and is blocked by J_5. Thus J_5 inherits the **current** priority 1 of J_4 and executes
- At 11, J_5 executes $U(\textit{Black})$, its priority returns to 5 (the priority before locking *Black*). Now J_4 has the highest priority (1) and executes

Priority-Inheritance Example



- At 13, J_4 executes $U(Shaded)$, its priority returns to 4. J_1 has now the highest priority and executes
- At 15, J_1 completes, J_2 is granted *Black* and has the highest priority and executes
- At 17, J_2 completes, afterwards J_3, J_4, J_5 complete.

Priority-Inheritance – Blocking Time

$z_{\ell,k}$ = the k-th critical section of J_ℓ

A job J_h is blocked by $z_{\ell,k}$ if J_h has higher priority than J_ℓ but has to wait for J_ℓ to exit $z_{\ell,k}$ in order to continue

$\beta^*_{h,\ell}$ = the set of all maximal critical sections $z_{\ell,k}$ that may block J_h
we assume that CS are properly nested, maximal CS which may block J_h is the one which is not contained within any other CS which may block J_h

Theorem 2
Let J_h be a job and let $J_{h+1}, \ldots, J_{h+m}$ be jobs with lower priority than J_h. Then J_h can be blocked for at most the duration of one critical section in each *of $\beta^*_{h,\ell}$ where $\ell \in \{h+1, \ldots, h+m\}$.*

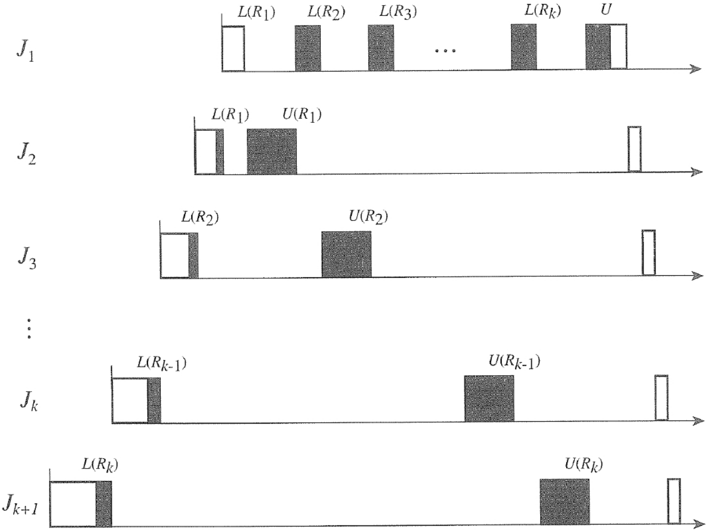
Proof.

Lemma 3
J_h can be blocked by J_ℓ only if J_ℓ is executing within a critical section $z_{\ell,k}$ of $\beta^*_{h,\ell}$ when J_h is released

It follows that J_h can be blocked by a lower priority job J_ℓ for at most duration of one critical section of $\beta^*_{h,\ell}$

□

Priority-Inheritance – The Worst Case



Properties of Priority-Inheritance Protocol

- Simple to implement, does not require prior knowledge of resource requirements
- Jobs exhibit two types of blocking
  - Direct blocking due to resource locks
    i.e., a job J_ℓ locks a resource R, J_h executes $L(R)$ is directly blocked by J_ℓ on R
  - Priority-inheritance blocking
    i.e., a job J_h is preempted by a lower-priority job that inherited a higher priority
- Jobs may exhibit transitive blocking
- Deadlock is *not* prevented
- Can reduce blocking time compared to non-preemptable CS but does not guarantee to minimize blocking

Priority-Ceiling Protocol

The goal: to furhter reduce blocking times due to resource contention

- in its basic form priority-ceiling protocol works under the assumption that the priorities of jobs and resources required by all jobs are known apriori

  can be extended to dynamic priority (job-level fixed priority), see later

Notation:

- The *priority ceiling* of any resource R_k is the highest priority of all the jobs that require R_k and is denoted by $\Pi(R_k)$
- At any time t, the current priority ceiling $\Pi(t)$ of the system is equal to the highest priority ceiling of the resources that are in use at the time
- If all resources are free, $\Pi(t)$ is equal to Ω, a newly introduced priority level that is lower than the lowest priority level of all jobs

Priority-Ceiling Protocol

The scheduling and priority-inheritance rules are the same as for priority-inheritance protocol

- Scheduling rules:
  - Jobs are scheduled in a preemptable priority-driven manner *according to their current priorities*
  - At release time, the current priority of a job is equal to its assigned priority
  - The current priority remains equal to the assigned priority, except when the priority-inheritance rule is invoked
- **Priority-inheritance rule**:
  - When job J_h becomes blocked on a resource R, the job J_k which blocks J_h inherits the current priority $\pi_h(t)$ of J_h; also priorities are inherited transitively
  - J_k executes at its inherited priority until it releases R; at that time, the priority of J_k returns to its priority $\pi_k(t')$ at the time t' when it acquired the resource R

    (note that $\pi_k(t')$ may still be an inherited priority)

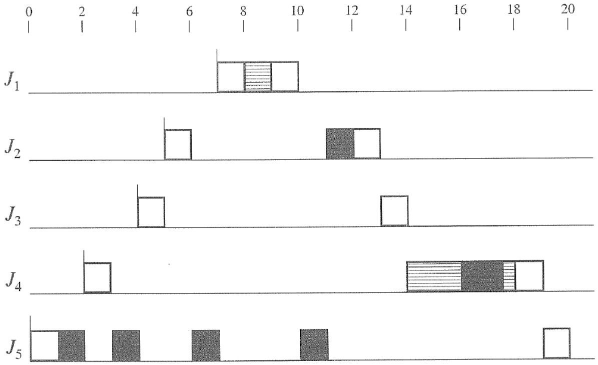
Priority-Ceiling Protocol

Resource allocation rule:

- When a job J requests a resource R held by another job, the request fails and the requesting job blocks
- When a job J requests a resource R at time t, and that resource is free:
  - If J's priority $\pi(t)$ is higher than current priority ceiling $\Pi(t)$, R is allocated to J
  - If J's priority $\pi(t)$ is not higher than $\Pi(t)$, R is allocated to J only if J is the job holding the resource(s) whose priority ceiling is equal to $\Pi(t)$, otherwise J is blocked
    (Note that only one job may hold the resources whose priority ceiling is equal to $\Pi(t)$)

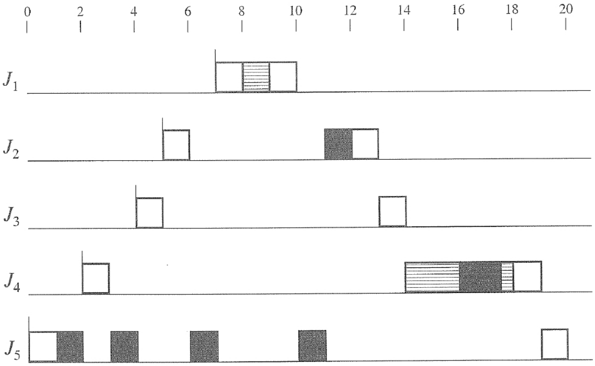
Note that unlike priority-inheritance protocol, the priority-ceiling protocol can deny access to an available resource

Priority-Ceiling Protocol



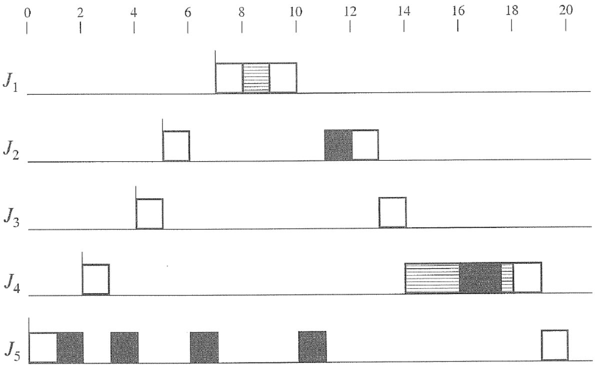
- At 1, $\Pi(t) = \Omega$, J_5 executes $L(Black)$, continues executing
- At 3, $\Pi(t) = 2$, J_4 executes $L(Shaded)$; because the ceiling of the system $\Pi(t)$ is higher than the current priority of J_4, job J_4 is blocked, J_5 inherits J_4's priority and executes at priority 4
- At 4, J_3 preempts J_5; at 5, J_2 preempts J_3. At 6, J_2 requests *Black* and is directly blocked by J_5. Consequently, J_5 inherits priority 2 and executes until preempted by J_1

Priority-Ceiling Protocol



- At 8, J_1 executes $L(\textit{Shaded})$, its priority is higher than $\Pi(t) = 2$, its request is granted and J_1 executes; at 9, J_1 executes $U(\textit{Shaded})$ and at 10 completes
- At 11, J_5 releases *Black* and its priority drops to 5; J_2 becomes unblocked, is allocated *Black* and executes

Priority-Ceiling Protocol



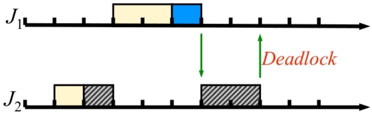
- At 14, J_2 and J_3 complete, J_4 is granted *Shaded* (because its priority is higher than $\Pi(t) = \Omega$) and executes
- At 16, J_4 executes $L(\mathit{Black})$ which is free, the priority of J_4 is not higher than $\Pi(16) = 1$ but J_4 is the job holding the resource whose priority ceiling is equal to $\Pi(16)$. Thus J_4 gets *Black*, continues to execute; the rest is clear

Priority-Ceiling Protocol

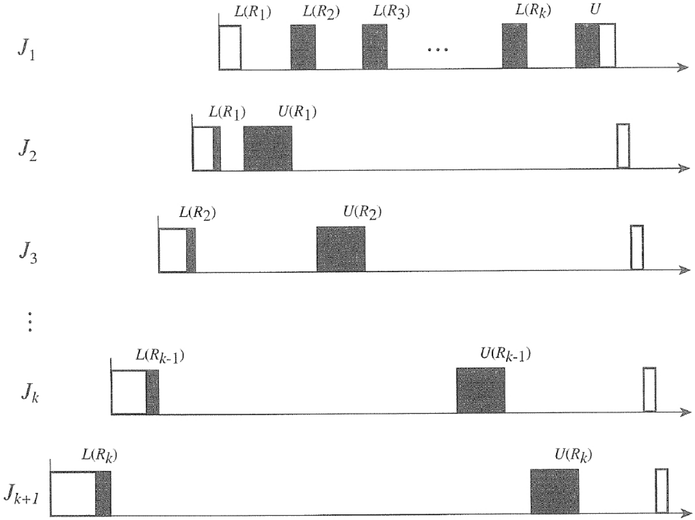
Theorem 4

Assume a system of preemptable jobs with fixed assigned priorities. Then

- *deadlock may never occur,*
- *a job can be blocked for at most the duration of one critical section.*



J1
Deadlock
J2



J1
L(R1)
L(R2)
L(R3)
L(Rk)
U
J2
L(R1)
U(R1)
J3
L(R2)
U(R2)
Jk
L(Rk-1)
U(Rk-1)
Jk+1
L(Rk)
U(Rk)

Differences between the priority-inheritance and priority-ceiling

- Priority-inheritance is greedy, while priority ceiling is not

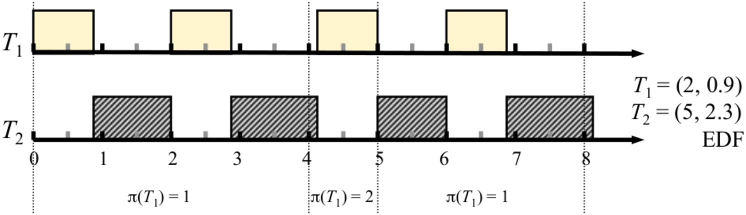
  The priority-ceiling protocol may withhold access to a free resource, i.e., a job can be blocked by a lower-priority job which does not hold the requested resource – *avoidance blocking*
- The priority ceiling protocol forces a fixed order onto resource accesses thus eliminating deadlock

Resources in Dynamic Priority Systems

The priority ceiling protocol assumes fixed and known priorities

In a dynamic priority system, the priorities of the periodic tasks change over time, while the set of resources is required by each task remains constant

- As a consequence, the priority ceiling of each reasource changes over time



What happens if T_1 uses resource X, but T_2 does not?

- Priority ceiling of X is 1 for $0 \leq t \leq 4$, becomes 2 for $4 \leq t \leq 5$, etc. even though the set of resources is required by the tasks remains unchanged

Resources in Dynamic Priority Systems

- If a system is job-level fixed priority, but task-level dynamic priority, a priority ceiling protocol can still be applied
  - Each job in a task has a fixed priority once it is scheduled, but may be scheduled at different priority to other jobs in the task (e.g. EDF)
  - Update the priority ceilings of all jobs each time a new job is introduced; use until updated on next job release
- Has been proven to prevent deadlocks and no job is ever blocked for longer than the length of one critical section
  - But: very inefficient, since priority ceilings updated frequently
  - May be better to use priority inheritance, accept longer blocking

Schedulability Tests with Resources

How to adjust schedulability tests?

Add the blocking times to execution times of jobs; then run the test as normal

The blocking time b_i of a job J_i can be determined for all three protocols:

- non-preemptable CS $\Rightarrow$ b_i is bounded by the maximum length of a critical section in lower priority jobs
- priority-inheritance $\Rightarrow$ b_i is bounded by the total length of the m longest critical sections where m is the number of jobs that may block J_i

  (For a more precise formulation see Theorem 2.)
- priority-ceiling $\Rightarrow$ b_i is bounded by the maximum length of a critical section