Resource Management

Advanced Operating Systems
Lecture 6
Lecture Outline

- Definitions of resources
- Resource access control for static systems
  - Priority inheritance protocol
  - Basic priority ceiling protocol
  - Stack-based priority ceiling protocol
- Resource access control for dynamic systems
- Effects on scheduling
- Implementation choices
Resources

• A system has $\rho$ types of resource $R_1, R_2, \ldots, R_\rho$
  • A plentiful resource has no effect on scheduling, and is ignored
  • Resources used by jobs in a non-preemptive and mutually exclusive manner; resources are serially reusable

• Access to resources is controlled using locks
  • Jobs attempt to lock a resource before starting to use it, and unlock the resource afterwards; the time the resource is locked is the critical section
  • If a lock request fails, the requesting job is blocked; a job holding a lock cannot be preempted by a higher priority job needing that lock
  • Critical sections may nest if a job needs multiple simultaneous resources
Jobs *contend* for a resource if they try to lock it at once:

- **Priority inversion** occurs when a low-priority job executes while some ready higher-priority job waits
- **Deadlock** can result from piecemeal acquisition of resources
  - The classic solution is to impose a fixed acquisition order over the set of lockable resources, and all jobs attempt to acquire the resources in that order (typically LIFO order)

Priority inversion

EDF schedule of $J_1$, $J_2$ and $J_3$ sharing a resource protected by locks (blue shading indicated critical sections). The red lines indicate release times and deadlines of jobs.
Inheritance of Priority

• A standard scheduling algorithm gives each job an assigned priority

• At some time $t$, a job $J_k$ has a current priority, $\pi_k(t)$; this can differ from its assigned priority
  • Due to uncontrolled priority inversion, or controlled inheritance of priority for purposes of managing blocking time
  • This can obviously affect correctness of the schedule
Resource Contention – Timing Anomalies

• Resource contention can cause timing anomalies due to priority inversion and deadlock – potentially arbitrary duration, and can seriously disrupt timing.

• Cannot eliminate anomalies, but protocols exist to control them:
  • Priority inheritance protocol
  • Basic priority ceiling protocol
  • Stack-based priority ceiling protocol
Priority Inheritance Protocol

• Aim: control inheritance of priority during resource access to reduce the duration of timing anomalies

• Constraints:
  • Works with any pre-emptive, priority-driven scheduling algorithm
  • Does not require any prior knowledge of resource requirements
  • Does not prevent deadlock, but if some other mechanism used to prevent deadlock, ensures that no job can block indefinitely due to uncontrolled priority inversion
Priority Inheritance: Scheduling Rules

- Jobs scheduled according to *current* priority
  - At release time, current priority equals assigned priority of the job
  - Current priority remains equal to assigned priority, unless priority-inheritance is invoked:
    - When a job, $J$, becomes blocked, the job $J_l$ which blocks $J$ inherits the current priority $\pi(t)$ of $J$
    - $J_l$ executes at its inherited priority until it releases $R$; at that time, the priority of $J_l$ returns to its priority $\pi_l(t')$ at the time $t'$ when it acquired the resource $R$

- When a job $J$ requests a resource $R$ at time $t$:
  - 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 Protocol: Example

What does the schedule look like?

Jobs 1, 2, 4, 5 acquire resource after 1 time unit
Job 4 acquires blue after further 2 units

<table>
<thead>
<tr>
<th>Job</th>
<th>$r_i$</th>
<th>$e_i$</th>
<th>$\pi_i$</th>
<th>Critical Sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J_1$</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>[red; 1]</td>
</tr>
<tr>
<td>$J_2$</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>[blue; 1]</td>
</tr>
<tr>
<td>$J_3$</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>$J_4$</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>[red; 4 [blue; 1.5]]</td>
</tr>
<tr>
<td>$J_5$</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>[blue; 4]</td>
</tr>
</tbody>
</table>
Priority Inheritance Protocol: Example

Jobs may block for many different reasons…

- $J_3$ blocked because $J_5$ inherits priority of $J_2$
- $J_3$ preempted by $J_2$
- $J_2$ directly blocked by $J_5$ due to the lock $J_5$ has on the resource
- Transitive blocking: $J_5$ blocks $J_4$ blocks $J_1$

$J_1$

$J_2$

$J_3$

$J_4$

$J_5$
Properties of Priority Inheritance Protocol

- Simple to implement, needs no prior knowledge of resource requirements
- Jobs exhibit different types of blocking
  - Direct blocking due to resource locks
  - Priority-inheritance blocking
  - Transitive blocking
- Lower blocking time than prohibiting preemption during critical sections, but does not guarantee to minimise blocking
- Deadlock is not prevented: need to manage lock acquisition order in addition
Basic Priority Ceiling Protocol

• Sometimes want to further reduce blocking times
• Basic priority ceiling protocol does this, provided:
  • The assigned priorities of all jobs are fixed (e.g. RM scheduling, not EDF)
  • The resources required by all jobs are known a priori
• Need two additional terms to define the protocol:
  • 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 $\Omega$, a nonexistent priority level that is lower than the lowest priority level of all jobs
Basic Priority Ceiling: Scheduling Rules (1)

- Scheduling rules:
  - Priority-driven scheduling; jobs can be preempted
  - The current priority of a job equals its assigned priority, except when the priority-inheritance rule (see next slide) is invoked

- 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 $R$ that is available:
    - if $J$'s priority $\pi(t)$ is higher than current priority ceiling $\Pi(t)$:
      - $R$ is allocated to $J$
    - else
      - if $J$ is the job holding the resource(s) whose priority ceiling is equal to $\Pi(t)$:
        - $R$ is allocated to $J$
      - else
        - the request is denied, and $J$ becomes blocked
  - Unlike priority inheritance: can deny access to an available resource
Basic Priority Ceiling: Scheduling Rules (2)

- Priority-inheritance rule:
  - When the requesting job, $J$, becomes blocked, the job $J_l$ which blocks $J$ inherits the current priority $\pi(t)$ of $J$
  - $J_l$ executes at its inherited priority until the time when it releases every resource whose priority ceiling is equal to or higher than $\pi(t)$; then, the priority of $J_l$ returns to its priority $\pi_l(t')$ at the time $t'$ when it was granted the resource(s)
Basic Priority Ceiling Protocol: Example

What does the schedule look like?

<table>
<thead>
<tr>
<th>Job</th>
<th>$r_i$</th>
<th>$e_i$</th>
<th>$\pi_i$</th>
<th>Critical Sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J_1$</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>[red; 1]</td>
</tr>
<tr>
<td>$J_2$</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>[blue; 1]</td>
</tr>
<tr>
<td>$J_3$</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>$J_4$</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>[red; 4] [blue; 1.5]</td>
</tr>
<tr>
<td>$J_5$</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>[blue; 4]</td>
</tr>
</tbody>
</table>

$J_4$ requests red but is denied, since $\pi_4 < \Pi$.
$J_2$ requests blue but is blocked since locked by $J_5$ (which inherits priority 2).

Significant reduction in execution time for some tasks compared to priority inheritance.

$\Pi$ remains =1 since red in use.
Basic Priority Ceiling Protocol: Properties

• If resource access in a system of preemptable, fixed priority jobs on one processor is controlled by the priority-ceiling protocol:
  • Deadlock can never occur
  • A job can be blocked for at most the duration of one critical section: there is no transitive blocking

• Differences between the priority-inheritance and priority-ceiling protocols:
  • Priority inheritance is greedy, while priority ceiling is not
    • The priority ceiling protocol may withhold access to a free resource, causing a job to be blocked by a lower-priority job which does not hold the requested resource – termed avoidance blocking
  • The priority ceiling protocol forces a fixed order onto resource accesses, thus eliminating deadlock
Stack-based Priority Ceiling Protocol

• Basic priority ceiling protocol performs well, but is complex, and has high context switch overheads
• Stack-based priority ceiling protocol has lower cost
• Defining rules:
  • Ceiling: When all resources are free, $\Pi(t) = \Omega$; $\Pi(t)$ updated each time a resource is allocated or freed
    - $\Pi(t)$ current priority ceiling of all resources in currently use; $\Omega$ non-existing lowest priority level
  • Scheduling:
    - After a job is released, it is blocked from starting execution until its assigned priority is higher than $\Pi(t)$
    - Non-blocked jobs are scheduled in a pre-emptive priority manner; tasks never self-yield
  • Allocation: when a job requests a resource, it is allocated
    - The allocation rule looks greedy, but the scheduling rule is not
Stack-based Priority Ceiling Protocol

What does the schedule look like?

<table>
<thead>
<tr>
<th>Job</th>
<th>$r_i$</th>
<th>$e_i$</th>
<th>$\pi_i$</th>
<th>Critical Sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J_1$</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>[1 ; 1]</td>
</tr>
<tr>
<td>$J_2$</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>[1 ; 1]</td>
</tr>
<tr>
<td>$J_3$</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>$J_4$</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>[1 ; 4 [1 ; 1.5]]</td>
</tr>
<tr>
<td>$J_5$</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>[1 ; 4]</td>
</tr>
</tbody>
</table>

Jobs blocked from starting since $\pi_i < \Pi$

Context switches are reduced compared to the basic priority ceiling protocol; no jobs finish later, but many jobs start later
Stack-based Priority Ceiling Protocol

- **Characteristics:**
  - When a job starts to run, all the resource it will ever need are free (since otherwise the ceiling would be ≥ priority)
    - No job ever blocks waiting for a resource once its execution has begun
    - Implies low context switch overhead
  - When a job is pre-empted, all the resources the pre-empting job will require are free, ensuring it will run to completion; deadlock cannot occur
  - Longest blocking time provably not worse than the basic priority ceiling protocol, i.e., not worse than the duration of one critical section
Choice of Resource Management Protocol

- If tasks never self yield, the stack based priority ceiling protocol is a better choice than the basic priority ceiling protocol
  - Simpler
  - Reduce number of context switches
  - Can also be used to allow sharing of the run-time stack, to save memory resources

- Both perform better than basic priority inheritance
  - Assuming fixed priority scheduling, resource usage known in advance
Resources in Dynamic Priority Systems

• The priority ceiling protocols assume fixed priority scheduling

• In a dynamic priority system, the priorities of the periodic tasks change over time, while the set of resources required by each task remains constant
  • As a consequence, the priority ceiling of each resource changes over time
  • Example:

```
π(T₁) = 1
π(T₁) = 2
π(T₁) = 1
```

• What happens if $T₁$ uses resource $X$, but $T₂$ 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 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
- Proven to work and have the same properties as priority ceiling protocol in fixed priority systems
  - But very inefficient, since priority ceilings updated frequently
  - May be better to use priority inheritance protocol, accept longer blocking
Maximum Duration of Blocking

• Assume $J_1$ and $J_2$ contend for a resource, $R$, where $J_1$ is the higher priority job
  • Worst case blocking time $\rightarrow$ duration of $J_2$’s critical section over $R$

• When using priority inheritance protocol, $J_2$ might be transitively blocked for the duration of the next priority job’s critical section
  • Worst case: it is blocked by every other lower priority job, for the full duration of each lower priority job’s critical section
Maximum Duration of Blocking

• The priority ceiling protocols implement avoidance blocking, and so do not exhibit transient blocking
  • Block for at most the duration of one low priority critical section
    • Direct blocking: low priority jobs locks resource; can be blocked for up to the duration of the critical section of that job
    • Avoidance blocking: resource is free, but priority ceiling rules deny access

• Calculate worst case blocking duration:
  • Simple:
    • Assume can block for duration of longest critical section of lower priority jobs
    • Probably overestimates blocking duration; likely not too significant
  • More efficient:
    • Trace direct conflicts with lower priority jobs, find longest critical section
    • Trace indirect conflicts with lower priority jobs that may inherit priority and cause avoidance blocking, find longest critical section
    • Greatest of these is maximum possible blocking time
Effect of Scheduling Tests

- Jobs that block for resource access affect whether a system can be scheduled

- How to adjust scheduling test?
  - Incorporate maximum blocking time as part of execution time of job; scheduling test then runs as normal
  - Priority ceiling protocols clearly preferred where possible
Implementation Choices

- POSIX real-time extensions provide useful baseline functionality
  - Priority scheduling abstraction, to implement Rate Monotonic schedules
  - A mutex abstraction using either priority inheritance or priority ceiling protocols to arbitrate resource access

- Similar, sometimes more advanced features, provided by other real-time operating systems
  - Example: Ada real-time package supports the priority ceiling protocol
Summary

- Defined resources, timing anomalies, and need for resource access control
- Operation of resource access control protocols:
  - Priority inheritance protocol
  - Basic priority ceiling protocol
  - Stack-based priority ceiling protocol
- Maximum duration of blocking
- Impact on scheduling tests