Resource Access Control in Real-time Systems

Advanced Operating Systems (M)
Lecture 8
Lecture Outline

• Definitions of resources
• Resource access control for static systems
  • Basic priority inheritance protocol
  • Basic priority ceiling protocol
  • Enhanced priority ceiling protocols
• Resource access control for dynamic systems
• Effects on scheduling
• Implementing resource access control
Resources

• A system has \( \rho \) types of resource \( R_1, R_2, \ldots, R_{\rho} \)
  • Each resource comprises \( n_k \) indistinguishable units; plentiful resources have no effect on scheduling and so are ignored
  • Each unit of resource is used in a non-preemptive and mutually exclusive manner; resources are serially reusable
  • If a resource can be used by more than one job at a time, we model that resource as having many units, each used mutually exclusively

• 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
Contention for Resources

- Jobs *contend* for a resource if they try to lock it at once

![Diagram showing contention and priority inversion]

- **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)
Timing Anomalies

- As seen, contention for resources can cause timing anomalies due to priority inversion and deadlock.
- Unless controlled, these anomalies can be arbitrary duration, and can seriously disrupt system timing.

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

- Aim: to adjust the scheduling priorities of jobs 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 the jobs’ 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

- We discuss the *basic* priority-inheritance protocol which assumes there is only 1 unit of resource
Basic Priority-Inheritance Protocol

• Assumptions (for all of the following protocols):
  • Each resource has only 1 unit
  • The priority assigned to a job according to a standard scheduling algorithm is its assigned priority
  • 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 \( \pi_l(t) \) of a job \( J_l \) may be raised to the higher priority \( \pi_h(t) \) of another job \( J_h \). In such a situation, the lower-priority job \( J_l \) is said to inherit the priority of the higher-priority job \( J_h \), and \( J_l \) executes at its inherited priority \( \pi_h(t) \)
Basic Priority-Inheritance Protocol

• Jobs are pre-emptively scheduled 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:
    • When a job, \( J \), becomes blocked, the job \( J_i \) which blocks \( J \) inherits the current priority \( \pi(t) \) of \( J \)
    • \( J_i \) executes at its inherited priority until it releases \( R \); at that time, the priority of \( J_i \) returns to its priority \( \pi(t') \) 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
Basic Priority-Inheritance Protocol

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>$[\boxed{7}; 1]$</td>
</tr>
<tr>
<td>$J_2$</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>$[\boxed{5}; 1]$</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>$[\boxed{4}; \boxed{1.5}]$</td>
</tr>
<tr>
<td>$J_5$</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>$[\boxed{0}; 4]$</td>
</tr>
</tbody>
</table>
Properties of the 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 desirable to further reduce blocking times due to resource contention

• The *basic priority-ceiling* protocol provides a means to do 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 Protocol

• 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 Protocol

• 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

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>[7; 1]</td>
</tr>
<tr>
<td>$J_2$</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>[5; 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>[2; 4 [3; 1.5]]</td>
</tr>
<tr>
<td>$J_5$</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>[0; 4]</td>
</tr>
</tbody>
</table>
Basic Priority-Ceiling Protocol

• 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
Enhancing the Priority Ceiling Protocol

- The basic priority ceiling protocol performs well, but is complex, and can result in high context switch overheads

- This has led to two modifications to the protocol:
  - The stack-based priority ceiling protocol
  - The ceiling priority protocol
Stack-Based Priority Ceiling Protocol

• Based on original work to allow jobs to share a run-time stack, extended to control access to other resources

• 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>[( \pi ); 1]</td>
</tr>
<tr>
<td>( J_2 )</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>[( \pi ); 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>[( \pi ); 4 [\pi ]; 1.5]]</td>
</tr>
<tr>
<td>( J_5 )</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>[( \pi ); 4]</td>
</tr>
</tbody>
</table>

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 $\geq$ 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 Priority Ceiling 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 give better performance than priority inheritance protocol
  - Assuming fixed priority scheduling, resource usage known in advance
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:

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 required by the tasks remains unchanged.
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
Effects on Scheduling Tests

- Jobs which block due to 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
Implementing Resource Access Control

• Have focussed on resource access control algorithms which can be implemented by an operating system

• How are these made available to applications?
  • Some implemented by the operating system
  • Some implemented at the application level
POSIX Mutex API

- Control access to resource using a mutex
  - A mutex is embedded in an object at a location of the programmers choosing to control access to that object/resource
  - Basic API:

```c
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int proto);
int pthread_mutexattr_getprotocol(pthread_mutexattr_t *attr, int *proto);
```
Can specify resource access protocol for a mutex:

- Use `pthread_mutexattr_setprotocol()` during mutex creation
  
  - **PTHREAD_PRIO_INHERIT**: Priority inheritance protocol applies
  
  - **PTHREAD_PRIO_PROTECT**: Priority ceiling protocol applies
  
  - **PTHREAD_PRIO_NONE**: Priority remains unchanged

- If the priority ceiling protocol is used, can adjust the ceiling to match changes in thread priority (e.g. dynamic priority scheduling):
  
  - `pthread_mutexattr_getprioceiling(…)`
  
  - `pthread_mutexattr_setprioceiling(…)`

Used with POSIX real-time scheduling:

- Allow implementation of fixed priority scheduling with a known resource access control protocol
  
- Controls priority inversion, scheduling; allows reasoning about a system
POSIX also defines a condition variable API:

```c
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           struct timespec *wait_time);

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Combine a condition variable with a mutex to wait for a condition to be satisfied:

```c
lock associated mutex
while (condition not satisfied) {
    wait on condition variable
}
do work
unlock associated mutex
```
Implementation Summary

• As seen, many approaches to implementing resource access control

• POSIX provides 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
  • Examples: Ada supports the priority ceiling protocol; QNX supports message based priority inheritance
Summary

- Defined resources, explaining timing anomalies and the need for resource access control
- Illustrated operation of three resource access control protocols:
  - Basic priority inheritance protocol
  - Basic priority ceiling protocol
  - Stack-based priority ceiling protocol
- Discussed impact on scheduling tests
- Implementation of resource access control in POSIX applications