Resource Access Protocols

- Some resources (data structures, physical resources) need to be protected, so that when a process starts using them, no other process can use them until the first process is done.
- Method: locks (semaphores).

Locks and deadlocks

P:

```java
do {
    wait (L1)
    wait (L2)
    use (R1)
    use (R2)
    release (L2)
    release (L1)
} 
```

Q:

```java
    do {
    wait (L2)
    wait (L1)
    use (R2)
    use (R1)
    release (L1)
    release (L2)
} 
```

- Assume that P1 and P2 try to get access to critical resources R1, R2.
- They acquire L1, L2 in different order.

Possible result: P gets L1, Q gets L2, deadlock!

The cure? Total ordering of locks.

- Choose a total order of all locks: L1 < L2 < ... < Ln
- When you need both L1 and L2, get first the lock that comes first in the order.
- Problems:
  - How to decide an order if locks protect, say, access to Java objects created dynamically?
  - How to enforce/check that the policy is respected?

Scheduling with locks

- Even when one-way locking is used, priority inversion problem:
Priority inheritance protocol

- Jobs i have two priorities: their nominal priority $P_i$ and their (possibly larger) active priority $Q_i$.
- Jobs are scheduled according to their active priority.
- Initially, $Q_i = P_i$ for all jobs.
- Locks: $lck_1, lck_2, \ldots$, each guarding a critical section.

Priority inheritance

- Define a relation <$> between processes, such that $i <$> j$ if j holds a lock on which i is waiting.
- Let $<^*$ be the reflexive transitive closure of $<$.
- Let $Q_k = \max\{P_i \mid i <$>^* k\}$

Priority inheritance: implementation

Data structure:

- With each lock (semaphore) $s$:
  - s.holder: process that has the lock
  - s.waiting: list of waiting processes
- With each process $p$:
  - p.waiting: lock for which it is waiting
  - p.holding: list of locks that it is holding.

Priority inheritance: implementation

priority_increase (j, p);
   set $Q_j = \max(Q_j, p)$;
   priority_increase (j, wait.holder, $Q_j$)

Priority inheritance: implementation

process k calls wait(s):
   if s.holder = 0, then {
      s.holder = k;
      append (k, holding, s);
   } else (/* s.holder != 0 */)
      append (s.waiting, k); k.wait = s;
      call priority_increase (s.holder, Q_i);
      suspend; /* waits to be woken up */
      s.holder = k; k.wait = 0;
      append (k, holding, s);
   }

Priority inheritance: implementation

Process k does signal(s):
   s.holder = 0
   remove (k, holding, s);
   $Q_k = \max(P_k, \text{dyn_priority}(k, \text{holding}))$;
   unlock s;
   if s.wait != 0 {
      j = highest priority process in s.wait;
      remove (s.wait, j);
      wakeup (j); /* Tell j it can enter s */
   }
Priority inheritance: implementation

dyn_priority (sem_list):
p = minimum_priority;
for each s in sem_list {
    p = max (p, max (Qi | i ∈ s.wait));
}
return p;

Types of blocking

Schedulability analysis

- Fact 1: Job Jₖ waits for at most one completion of a critical section of job
  that blocks it (directly or indirectly), regardless of how many times Jₖ is
  using each lock.
- Proof: since k has higher priority, once Jₖ gets out, Jₖ will not let it
  begin something else.

Schedulability analysis

- Fact 2: If job k uses lock lck, then it can
  be blocked for the duration of at most one
  critical section guarded by lck, regardless
  how many times lck is used by how many
  processes.
- Proof: k has higher priority (by definition)
  that all processes that try to block it.
  Hence, the first time k tries to get a lock
  lck, it will be blocked for at most the time
  it takes for the lower-priority job inside to
  get out. The following times, no lower-
  priority process may have gotten inside a
  section guarded by lck.

Computing the maximum
blocking time

Case for non-recursive locks
 ONLY

- For each lock s, define C(s) (the ceiling
  of s) by C(s) = max (Pj; j uses s).
- D[i,s] = maximum duration in process i
  of a critical section guarded by lock s.

Computation of blocking time

For each process:

\[ B^\text{proc}_i = \sum_{j=i+1}^{m} \max_{k} \{ D_{j,k} | C(s_k) \geq P_i \} \]

\[ B^\text{locks}_i = \sum_{k=1}^{m} \max_{j \neq i} \{ D_{j,k} | C(s_k) \geq P_i \} \]

\[ B_i = \min \{ B^\text{proc}_i, B^\text{locks}_i \} \]
Example

<table>
<thead>
<tr>
<th>lck₁</th>
<th>lck₂</th>
<th>lck₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>J₁</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>J₂</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>J₃</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>J₄</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

• J₁, processes: 9+8+6 = 23
• J₁, locks: 9+8 = 17
• Blocking bound: 17

Example

<table>
<thead>
<tr>
<th>lck₁</th>
<th>lck₂</th>
<th>lck₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>J₁</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>J₂</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>J₃</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>J₄</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

• J₂, processes: 8+6 = 14
• J₂, locks: 8+7+4 = 19
• Blocking bound: 14

Example

<table>
<thead>
<tr>
<th>lck₁</th>
<th>lck₂</th>
<th>lck₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>J₁</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>J₂</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>J₃</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>J₄</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

• J₃, processes: 6
• J₃, locks: 6+5+4 = 15
• Blocking bound: 6

Example

<table>
<thead>
<tr>
<th>lck₁</th>
<th>lck₂</th>
<th>lck₃</th>
<th>lck₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>J₁</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>J₂</td>
<td>0</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>J₃</td>
<td>8</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>J₄</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

• J₄, processes: 0
• J₄, locks: 0
• Blocking bound: 0

Schedulability analysis: RM -utilization method-

• Theorem: consider a set of n periodic tasks J₁, ..., Jᵣ where P₁ > P₂ > ... > Pᵣ, and assume that for all 1 ≤ i ≤ n we have:

\[
\frac{B_i}{T_i} + \sum_{k=1}^{i} \frac{C_k}{T_k} \leq i(2^{1/i} - 1)
\]

then the set of periodic tasks is schedulable using Priority Inheritance and Rate Monotonic.
Proof:

- If the criterion holds, then a job \( i \) has enough time even if lasted for \( C_i + B_i \), taking into account the preemption \( C_i / T_k \) from higher priority jobs.

Schedulability analysis: RM -response time method-

- Theorem: consider a set of \( n \) periodic tasks \( J_1, J_2, \ldots, J_n \), where \( P_1 > P_2 > \cdots > P_n \), and for \( 1 \leq i \leq n \) let \( R_i \) be the least fixpoint of

\[
R_i = C_i + B_i + \sum_{j=1}^{i-1} \frac{R_j}{T_j} C_j
\]

then, if \( R_i < D_i \) for all \( 1 \leq i \leq n \), the processes are schedulable by RM or DM.

Chained blocking

Priority Ceiling Protocol

- Each lock \( s \) has a priority ceiling \( C(s) \): highest priority of task that will lock it.
- When a task \( i \) wants to get a lock \( s \), we compute the set \( H_i \) of locks held by tasks different from \( i \), and we compute

\[
P^* = \max \{ C(s') \mid s' \in H_i \} \]

- The process \( i \) gets the lock \( s \) only if \( P_i > P^* \).

Example: without ceiling

Example: with ceiling

Here, the ceiling is set to 3. The process with \( P=2 \) cannot enter, even though it wants another lock.
Properties of Priority Ceiling

**Lemma:** If task \( k \) is preempted in a critical section \( Z \) by a job \( j \) that then enters a critical section \( W \), then \( k \) cannot inherit a priority greater than that of \( j \) until \( j \) leaves \( W \).

**Proof:** If \( k \) inherits a priority greater than that of \( j \) before \( j \) completes \( Z \), it means that there is a job \( m \) that blocks on \( Z \) and raises the priority of \( k \) so that \( P_k > P_j \). But then the ceiling of \( Z \) would be at least \( P_k \), and \( j \) would not have been able to enter \( W \).

Properties of Priority Ceiling

**Theorem:** The priority ceiling protocol prevents transitive blocking.

**Proof:** Suppose that a transitive block occurs: \( J_1 \) blocks \( J_2 \), and \( J_2 \) blocks \( J_3 \). Assume \( P_i < 4 \). Then, \( J_3 \) will inherit the priority of \( J_2 \); however, this contradicts the previous lemma, that says that once \( J_3 \) enters \( W \), since \( W \) has ceiling at least 3, then \( J_3 \) cannot enter its region and block \( J_3 \).

Properties of Priority Ceiling

**Theorem:** Under the priority ceiling protocol, a job can be blocked for at most the duration of a critical section.

**Proof:** Suppose that \( J_1 \) is blocked by two lower-priority jobs \( J_2 \) and \( J_3 \), where \( P_2 < P_3 < P_1 \). Then, \( J_2 \) enters the critical section first, and \( C_2 \) is the ceiling of that section. Hence, we must have \( P_1 > C_2 \).

Moreover, since \( J_1 \) can be blocked by \( J_2 \), we have \( P_1 > C_2 \geq P_3 \). This contradicts \( P_1 > P_3 \).

Properties of Priority Ceiling

**Theorem:** under the priority ceiling protocol, a critical section \( Z_{jk} \) in job \( J_j \) and guarded by lock \( S_k \) can block a job \( J_j \) only if \( P_j < P_k \) and \( C(S_k) \geq P_k \).

**Hence, define**

\[ B_j = \max_{i,k} \{ D_{jk} \mid P_j < P_k < C(S_k) \} \]

where \( D_{jk} \) is the duration of \( Z_{jk} \).

Properties of Priority Ceiling

**Priority Ceiling: Implementation**

Process \( J_k \) calls \( \text{wait}(s) \):

- Find the lock \( s^* \) having max ceiling \( C \) among the locks held by processes other than \( J_k \).
- If \( P_k < C \), transfer \( P_k \) to the process that holds \( s^* \), insert \( J_k \) into the ready queue, and execute the ready job (other than \( J_k \)) with the highest priority.
- If \( P_k > C \), then \( J_k \) locks \( s \) and enters the associated critical section.
Priority Ceiling: Implementation

Process $J_i$ calls signal(s):
- Remove $s$ from the list of locked locks.
- If no other jobs are blocked by $J_i$, then set $P_i$ to the nominal priority of $J_i$; otherwise set $P_i$ to the highest priority of jobs are blocked by $J_i$.
- Let $P^*$ be the highest priority among ready jobs. If $P_i < P^*$, insert $J_i$ in the ready queue and execute the ready job (different from $J_i$) with highest priority.

Homework due Fri April 26

- Do an exercise on checking schedulability: without priority inheritance, with priority inheritance, with ceiling. Make them compare between RM and EDF.