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

Q: do {
    wait (L2)
    wait (L1)
    use (R1)
    use (R2)
    release (L1)
    release (L2)
}

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

The cure? Total ordering of locks.

- Choose a total order of all locks: L_1 < L_2 < ... < L_n
- When you need both L_i and L_j, 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:
  high priority: unbounded wait
  low priority
Priority inheritance protocol

- Each job $i$ has two priorities: the nominal priority $P_i$, and the (possibly higher) 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, ..., $ each guarding a critical section.

Priority inheritance

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

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.

**process $k$ calls wait($s$):**

```plaintext
if s.holder = ∅, then {
  s.holder = k;
  append (k.holding, s);
} else { /* s.holder ≠ ∅ */
  append (s.waiting, k);
  k.wait = s;
  call priority_increase (s.holder, Q_k);
  suspend; /* waits to be woken up */
  s.holder = k; s.wait = ∅;
  append (k.holding, s);
}
```

**Process $k$ does signal($s$):**

```plaintext
s.holder = ∅
remove (k.holding, s);
Q_k = max(P_k, dyn_priority(k.holding));
unlock s;
if s.wait ≠ ∅ {
  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 \in s.wait});
  }
  return p;

Types of blocking

Schedulability analysis

- **Fact 1**: Job $J_k$ waits for at most one completion of a critical section of job that blocks it (directly or indirectly), regardless of how many times $J_k$ is using each lock.
- **Proof**: since $k$ has higher priority, once $J_i$ gets out, $J_k$ 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 of 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 $C(s) = \max (P_j; j \text{ uses } s)$.
- $D_{is} = \text{maximum duration in process } i \text{ of a critical section guarded by lock } s$.

Computation of blocking time

For each process:

$$B_{i}^{\text{proc}} = \sum_{j=i+1}^{n} \max \{D_{j,s_k} | C(s_k) \geq P_i\}$$

$$B_{i}^{\text{locks}} = \sum_{k=1}^{m} \max \{D_{j,k} | C(s_k) \geq P_i\}$$

$$B_i = \min \{B_{i}^{\text{proc}}, B_{i}^{\text{locks}}\}$$
Example

<table>
<thead>
<tr>
<th>J_1</th>
<th>lck_1</th>
<th>lck_2</th>
<th>lck_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>J_2</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>J_3</td>
<td>0</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>J_4</td>
<td>8</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>J_5</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

- J_1 processes: 9+8+6 = 23
- J_1 locks: 9+8 = 17
- Blocking bound: 17

Example

<table>
<thead>
<tr>
<th>J_1</th>
<th>lck_1</th>
<th>lck_2</th>
<th>lck_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>J_2</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>J_3</td>
<td>0</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>J_4</td>
<td>8</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>J_5</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

- J_2 processes: 8+6 = 14
- J_2 locks: 7+4 = 11
- Blocking bound: 11

Example

<table>
<thead>
<tr>
<th>J_1</th>
<th>lck_1</th>
<th>lck_2</th>
<th>lck_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>J_2</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>J_3</td>
<td>0</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>J_4</td>
<td>8</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>J_5</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

- J_3 processes: 6
- J_3 locks: 6+5 = 11
- Blocking bound: 6

Example

<table>
<thead>
<tr>
<th>J_1</th>
<th>lck_1</th>
<th>lck_2</th>
<th>lck_3</th>
<th>lck_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>J_2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>J_3</td>
<td>0</td>
<td>9</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>J_4</td>
<td>8</td>
<td>7</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>J_5</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>11</td>
</tr>
</tbody>
</table>

The true lower bound is computed looking at a selection that never repeats locks nor jobs. Complexity: NP-complete (I bet).

Schedulability analysis: RM

Theorem: consider a set of n periodic tasks J_1 ... J_n where P_1 > ... > P_n, and assume that for all 1 \leq i \leq 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 $C_k/T_k$ from higher priority jobs.

Schedulability analysis: RM

- **Theorem**: consider a set of $n$ periodic tasks $J_1, ..., J_n$ where $P_1 > ... > P_n$, and for $1 \leq i \leq n$ let $R_i$ be the least fixpoint of

$$R_i = C_i + B_i + \sum_{j=1}^{i-1} \left\lfloor \frac{R_j}{T_j} \right\rfloor C_j$$

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

Chained blocking

<table>
<thead>
<tr>
<th>lock 1</th>
<th>lock 2</th>
</tr>
</thead>
</table>

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 $P_i > P^*$. Note that $P^*$ is independent from the desired lock $s$!

Example: without ceiling

<table>
<thead>
<tr>
<th>P=3</th>
<th>wait1</th>
<th>wait2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P=2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P=1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: with ceiling

<table>
<thead>
<tr>
<th>P=3</th>
<th>wait1</th>
</tr>
</thead>
<tbody>
<tr>
<td>P=2</td>
<td></td>
</tr>
<tr>
<td>P=1</td>
<td></td>
</tr>
</tbody>
</table>

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 $P_m$, 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 $P_i = 4 - i$. Then, $J_3$ will inherit the priority of $J_1$; however, this contradicts the previous lemma, that says that once $J_1$ enters W, since W has ceiling at least 3, then $J_2$ cannot enter its region and block $J_3$.

Properties of Priority Ceiling

Theorem: The priority ceiling protocol prevents deadlocks.

Proof: A deadlock can only happen when there is a cycle of blocked processes. Consider the shortest cycle: $J_1$ blocks $J_2$, and vice-versa; $J_1$ has higher priority. Then, who entered first?
- If $J_2$, then the ceiling was raised to $P_1$; then $J_1$ could not have entered.
- If $J_1$, then $J_2$ could not have entered. In both cases, we reach a contradiction.

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 $J_1$ and $J_2$, where $P_2 < P_1$. 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 $P_1 > C^*_{2} \geq P_i$. This contradicts $P_i > P_1$.

Properties of Priority Ceiling

Theorem: under the priority ceiling protocol, a critical section $Z_{j,k}$ in job $J_k$ and guarded by lock $S_k$ can block a job $J_i$ only if $P_i < P_k$ and $C(S_k) \geq P_i$.

Hence, define

$$B_i = \max_{j,k} \{ D_{j,k} | P_i < P_k, C(S_k) \geq P_i \}$$

where $D_{j,k}$ is the duration of $Z_{j,k}$.

Which critical sections can block a job?

Process $J_k$ calls wait(s):

1. Find the lock $s'$ having max ceiling $C'$ among the locks held by processes other than $J_k$.
2. If $P_k \leq C'$, 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.
3. If $P_k > C'$, then $J_k$ locks $s$ and enters the associated critical section.
Priority Ceiling: Implementation

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