

CMPE 117

Lecture:
Resource access protocols

© Luca de Alfaro, 2002

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

- Assume that P_1 and P_1 try to get access to critical resouces R_1, R_2 .
- They acquire L_1, L_2 in different order.

Locks and deadlocks

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

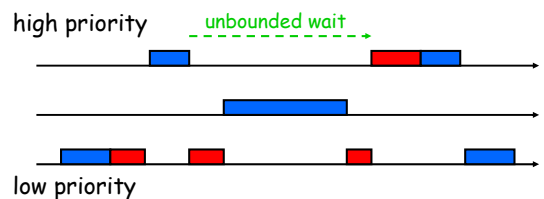
Possible result: P gets L_1 , Q gets L_2 ,
deadlock!

The cure? Total ordering of locks.

- Choose a total order of all locks: $L_1 < L_2 < \dots < L_n$
- 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**:



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, \dots , 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.

Priority inheritance: implementation

```
process k calls wait(s):
  if s.holder =  $\emptyset$ , then {
    s.holder = k;
    append (k.holding, s);
  } else { /* s.holder  $\neq \emptyset$  */
    append (s.waiting, k); k.wait = s;
    call priority_increase (s.holder,  $Q_k$ );
    suspend; /* waits to be woken up */
    s.holder = k; k.wait =  $\emptyset$ ;
    append (k.holding, s);
  }
```

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 does signal(s):

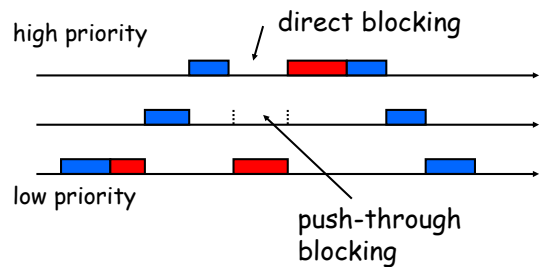
```
s.holder =  $\emptyset$ 
remove (k.holding, s);
 $Q_k = \max\{P_k, \text{dyn\_priority}(k.\text{holding})\}$ ;
unlock s;
if s.wait  $\neq \emptyset$  {
  j = highest priority process in s.wait;
  remove (s.wait, j);
  wakeup (j); /* Tell j it can enter s */
}
```

Priority inheritance: implementation

```

dyn_priority (sem_list):
  p = minimum_priority;
  for each s in sem_list {
    p = max (p, max {Q_i | i ∈ 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 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 $C(s) = \max \{P_j : j \text{ uses } s\}$.
- $D_{i,s}$ = maximum duration in process i of a critical section guarded by lock s .

Computation of blocking time

For each process:

$$B_i^{proc} = \sum_{j=i+1}^n \max_k \{D_{j,s_k} \mid C(s_k) \geq P_i\}$$

$$B_i^{locks} = \sum_{k=1}^m \max_{j>i} \{D_{j,k} \mid C(s_k) \geq P_i\}$$

$$B_i = \min \{B_i^{proc}, B_i^{locks}\}$$

Example

	lck ₁	lck ₂	lck ₃
J ₁	1	2	0
J ₂	0	9	3
J ₃	8	7	0
J ₄	6	5	4

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

Example

	lck ₁	lck ₂	lck ₃
J ₁	1	2	0
J ₂	0	9	3
J ₃	8	7	0
J ₄	6	5	4

- J₂, processes: 8+6 = 14
- J₂, locks: 7+4 = 11
- Blocking bound: 11

Example

	lck ₁	lck ₂	lck ₃
J ₁	1	2	0
J ₂	0	9	3
J ₃	8	7	0
J ₄	6	5	4

- J₃, processes: 6
- J₃, locks: 6+5 = 11
- Blocking bound: 6

Example

	lck ₁	lck ₂	lck ₃
J ₁	1	2	0
J ₂	0	9	3
J ₃	8	7	0
J ₄	6	5	4

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

Example

	lck ₁	lck ₂	lck ₃	lck ₄
J ₁	1	2	1	1
J ₂	0	9	3	3
J ₃	8	7	0	4
J ₄	6	5	4	11
J ₅	4	3	10	5

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

Schedulability analysis: RM -utilization method-

- **Theorem:** consider a set of n periodic tasks J₁ ... J_n where P₁ > ... > P_n, 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 it lasted for $C_i + B_i$, taking into account the preemption C_k/T_k from higher priority jobs.

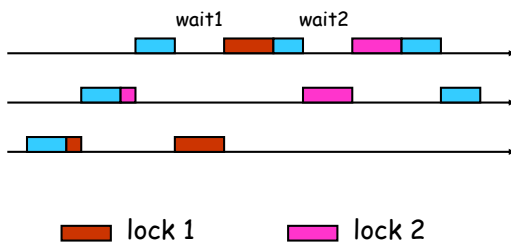
Schedulability analysis: RM -response time method-

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

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

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

Chained blocking

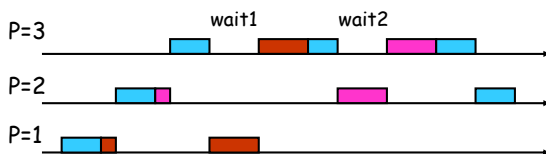


Priority Ceiling Protocol

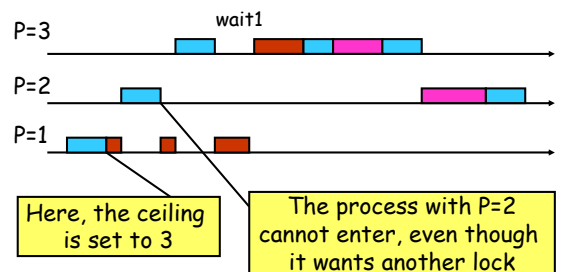
- Each lock s has a priority ceiling $C(s)$: highest priority of task that will lock it.
- When a task i wants to get a lock s , we compute the set H_i of locks held by tasks different from i , and we compute

$$P^* = \max \{C(s') \mid s' \in H_i\}$$
- The process i gets the lock s only if $P_i > P^*$. Note that P^* is independent from the desired lock s !

Example: without ceiling



Example: with ceiling



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 $P_i = 4-i$. 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_i is blocked by two lower-priority jobs J_1 and J_2 , where $P_2 < P_1 < P_i$. Then, J_2 enters the critical section first, and C_2^* is the ceiling of that section. Hence, we must have $P_i > C_2^*$. Moreover, since J_i can be blocked by J_2 , we have $P_i > C_2^* \geq P_1$. This contradicts $P_i > P_1$.

Which critical sections can block a job?

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

$$B_i = \max_{j,k} \{D_{j,k} \mid P_j < P_i, C(S_k) \geq P_i\}$$
 where $D_{j,k}$ is the duration of $Z_{j,k}$

Priority Ceiling: Implementation

Process J_k calls `wait(s)`:

- Find the lock s^* having max ceiling C^* among the locks held by processes other than J_k
- 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.
- 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.