Synchronization (II)

Instructor: Josep Torrellas
CS533
Ticket Lock with Delay

• Introduce delay on each processor between probes
• Not exponential backoff --> delays multiply
• Better have a delay = f(number of processors waiting) = f(difference between R and T)

• Do not use the average time of processors in critical section, but the minimum!
**Ticket Lock**

```pascal
 type lock = record
   next-ticket : unsigned integer := 0
   now-serving: unsigned integer := 0
  end;

 procedure acquire-lock( L: ^lock)
  my-ticket : unsigned integer := fetch-and-increment(&L -->
   next-ticket)
  // returns the old value; arithmetic overflow is harmless
  loop
    pause (my-ticket - L-->now-serving)
    // consume this many units of time
    // on most machines, subtractions works correctly despite ovf
  if   (L-->now-serving == my-ticket) return

 procedure release-lock(L: ^lock)
  L-->now-serving ++
```
Array Based Queuing Locks

• Disadvantages of Ticket locks:
  – not possible to obtain the lock with an expected constant number of transactions

• ABQLs:
  – processors use the atomic operation to obtain the address of a location to spin on.
  – Each processor spins on a different location

• In a different cache line!
\textbf{type} lock = record
\begin{align*}
\text{slots: array[0…numproc-1] of (has-lock,must-wait)} \\
:= (\text{has-lock, must-wait, must-wait,……, must-wait})
\end{align*}

//each element of slots should lie in diff cache line
\text{next-slot: integer := 0}

//parameter my-place, below, points to a priv variable
\text{procedure acquire-lock(L: ^lock, my-place: ^integer)}
\begin{align*}
\text{my-place^ := fetch-and-increment(&L \rightarrow next-slot)} \\
// returns the old value \\
\text{my-place^ := my-place^ mod numproc}
\end{align*}

\text{repeat while (L\rightarrow slots[my-place^] == must-wait) } //spin
\begin{align*}
\text{L\rightarrow slots[my-place^] := must-wait} \\
//init for next one
\end{align*}

\text{procedure release-lock(L: ^lock, my-place: ^integer)}
\text{L\rightarrow slots[(my-place^ + 1) mod numproc]} := \text{has-lock}
ABQL

• Advantages:
  – FIFO
  – Little network traffic: each processor spins on its own variable

• Disadvantage:
  – space per lock is linear with the number of processors
MCS Lock

• Same advantages as ABQL
• In addition, less space
• Requires fetch&store
Centralized Barrier

• Each processor
  – updates shared state to indicate its arrival
  – polls that state, to see when all procs arrived

• Since barrier has to be used repeatedly:
  – state must end as it started
Centralized Barriers (Cont)

- Each processor must spin twice per instance
  - 1) to ensure that all processors have left the previous barrier
  - 2) to ensure that all processors have arrived at the current barrier
Centralized Barriers (Cont)

- Optimization: may eliminate one of the spinning episodes by reversing the sense of the barrier

- Last processor resets the count and reverses sense

- Consecutive barriers do not interfere because all operations on count are done before the sense is toggled
Example of Code

shared count : integer := P
shared sense : boolean := true
processor private local-sense : boolean := true

procedure central-barrier
    local-sense := non local-sense  // each proc toggles its own sense
    if (fetch-and-decrement (&count) == 1)
        count := P
        sense := local-sense  // last processor toggles global sense
    else
        repeat until sense == local-sense

Centralized Barriers

• Disadvantage:
  – spinning on a single shared location (sense)
• If caches, sense can be replicated efficiently: only updated once (as opposed to spin locks)
• If not caches or a limited directory: lot of traffic
Barrier with Adaptive Backoff

- Delay between successive polling:
  - Adv: less traffic
  - Dsv: more latency (killer if lots of processors)

- Not scalable!
Software Combining Tree Barrier

• Shared variable represented as a tree of vars
• Each node of the tree in a different cache line
• Processors divided into groups
• Each group assigned to a leaf of the tree
• Each processor updates the state of its leaf
• If last one to arrive in a group, it continues up the tree to update the parent
Software Combining Tree Barrier

• Writes into a tree are used to determine that all processors have reached the barrier
• Reads out of a second tree allow the processors to continue past barrier
• The two trees can be combined
How it Works

• The processor that reaches the root of the tree begins a reverse wave of updates to lock-sense flags
• As soon as it awakes, each processor retraces its path through the tree unblocking its siblings
type node = record
  k: integer // fan-in of this node
  count: integer // initialized to k
  locksense: boolean // initially false
  parent: ^node // pointer to parent node; nil if root

shared nodes: array [0..P-1] of node
  // each element of nodes allocated in a different cache line
processor private sense : boolean := true
processor private mynode: ^node // my group’s leaf in the combining tree

procedure combining-barrier
  combining-barrier-aux (mynode) // join the barrier
  sense := not sense // for next barrier
procedure combining-barrier-aux (nodepointer: ^node) 
with nodepointer^ do 
  if fetch-and-decrement(&count) == 1  //last one to reach this node 
    if parent != nil 
      combining-barrier-aux(parent) 
    count := k  //prepare for next barrier 
    locksense := not locksense  // release waiting processors 
  repeat until locksense = sense
Software Combining Tree Barrier

• Adv: decreases contention and prevents tree saturation in interconnection networks (distributes the accesses)

• Dsv: spinning on memory locations that
  – cannot be statically determined (past leaf level)
  – on which other processors are also spinning

  … this is a problem if no caches or limited directory. Otherwise caches cache the spinning
Performance of spin locks/barriers

- Fig 15 of MCS paper
- Fig 19 of MCS paper