CS533: Synchronization (II)

Josep Torrellas

University of Illinois in Urbana-Champaign

February 24, 2015

Alternative Approach

- No fancy hardware
- Software smarts: distribute the lock and spin locally
- Use locks and barriers

Simple Test & Set

- Processor tests and set it. Returns old value
- Polling loop around a boolean variable

```
While(Test & Set(lock)){}
```

- Disadvantages:
  - Contention for the lock
  - Frequent accesses using the expensive RMW

Test & Test & Set

- Issue a Test & Set only when a previous read has indicated that the Test & Set may succeed
- Advantages:
  - Reduce the number of RMWs
  - Reduce the number of cache misses
- Disadvantage:
  - Still several processors may attempt the Test & Set

Test & Set With Delay

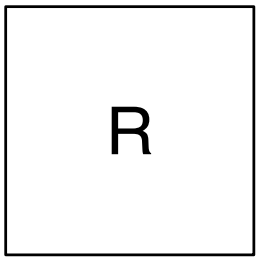
- Reduce the use of caches/networks by delaying consecutive probes of the lock after failure
- Example:
  - Constant delay
  - Backoff on unsuccessful probes
  - Exponential backoff (best)

Example

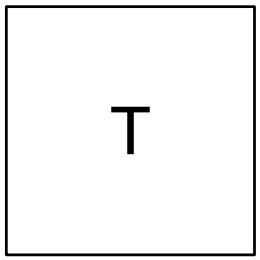
```
type lock = (unlocked, locked)
procedure acquire_lock (L: ^lock)
    integer delay :=1
    while (Test&Set(L) == locked) {
        pause (delay)
    delay := delay*2
    }

procedure release_lock(L: ^lock)
    lock^ := unlocked
```

Ticket Lock

- In Test & Test & Set, still a lot of RMW is possible (e.g. every processor every lock release)
- Use Ticket Lock:



Requests to Acquire the lock



Times the lock has been released

```
Acquire: ticket=Fetch & Increment(R)
    wait until T == ticket
    ...
    T++
```

Ticket Lock (II)

Advantages:

- Only 1 Fetch & Op per lock operation (probing is done with reads only)
- FIFO scheme: grant lock to processors in order they requested it $\rightarrow$ fair, no starvation

Disadvantages:

- Still a lot of cache or network contention through polling

Ticket Lock with Delay

- Introduce delay on each processor between probes
- Not exponential backoff $\rightarrow$ delays multiply
- Better have a delay $= f(\textit{number of processors waiting}) = f(\textit{difference between Request Number and Tickets Satisfied})$
- Do not use the average time of processors in critical section, but the minimum!

Ticket Lock

```
type lock = record
    next_ticket : unsigned integer := 0
    now_serving: unsigned integer := 0

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
- Array-Based Queuing Locks (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!

Example

```
type lock = record
    slots: array[0...numproc-1] of (has_lock,must_wait)
    := (has_lock, must_wait, must_wait, ... , must_wait)
        //each element of slots should lie in diff cache line
        next_slot: integer := 0
        //parameter my_place, below, points to a priv variable

procedure acquire_lock( L: ^lock, my_place: ^integer)
    my_place^ := fetch_and_increment(&L --> next_slot)
    // returns the old value
    my_place^ := my_place^ mod numproc
    repeat while (L-->slots[my_place^] == must_wait) //spin
    L-->slots[my_place^] := must_wait //init for next one

procedure release_lock(L: ^lock, my_place: ^integer)
    L-->slots[(my_place^ + 1) mod numproc] := 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
  - FIFO ordering of lock acquisitions
- Spins on locally-accessible flags
- In addition, less space than ABQL
- Requires fetch & store, meaning atomic swap, and benefits from compare & swap

MCS Lock (fig 6(a))

- Every processor using the lock
  - allocates a qnode record with a queue_link and a flag
- Processors holding or waiting for the lock are chained by links
- Each processor spins on a local (shared) variable
- The lock contains a pointer to the qnode of the processor at the tail of the queue or nil (if lock free)
- Each processor in the queue holds the address of the record for the processor behind it in the queue

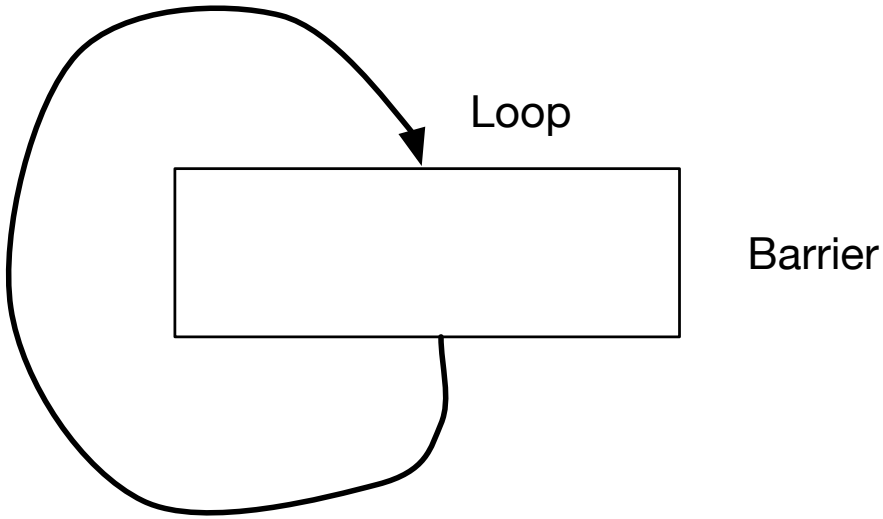
Centralized Barrier

- 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 (Continued)

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 (Continued)

- 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

Examples 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:
  - spining on a single shared location (sense)
- If caches, sense can be replicated efficiently: only update once (as opposed to spin locks)
- If not caches or a limited directory: a lot of traffic

Barrier with Adaptive Backoff

- Delay between successive polling
- Advantage: less traffic
- Disadvantage: more latency (killer if lots of processors)
- Not scalable!

Software Combining Tree Barrier

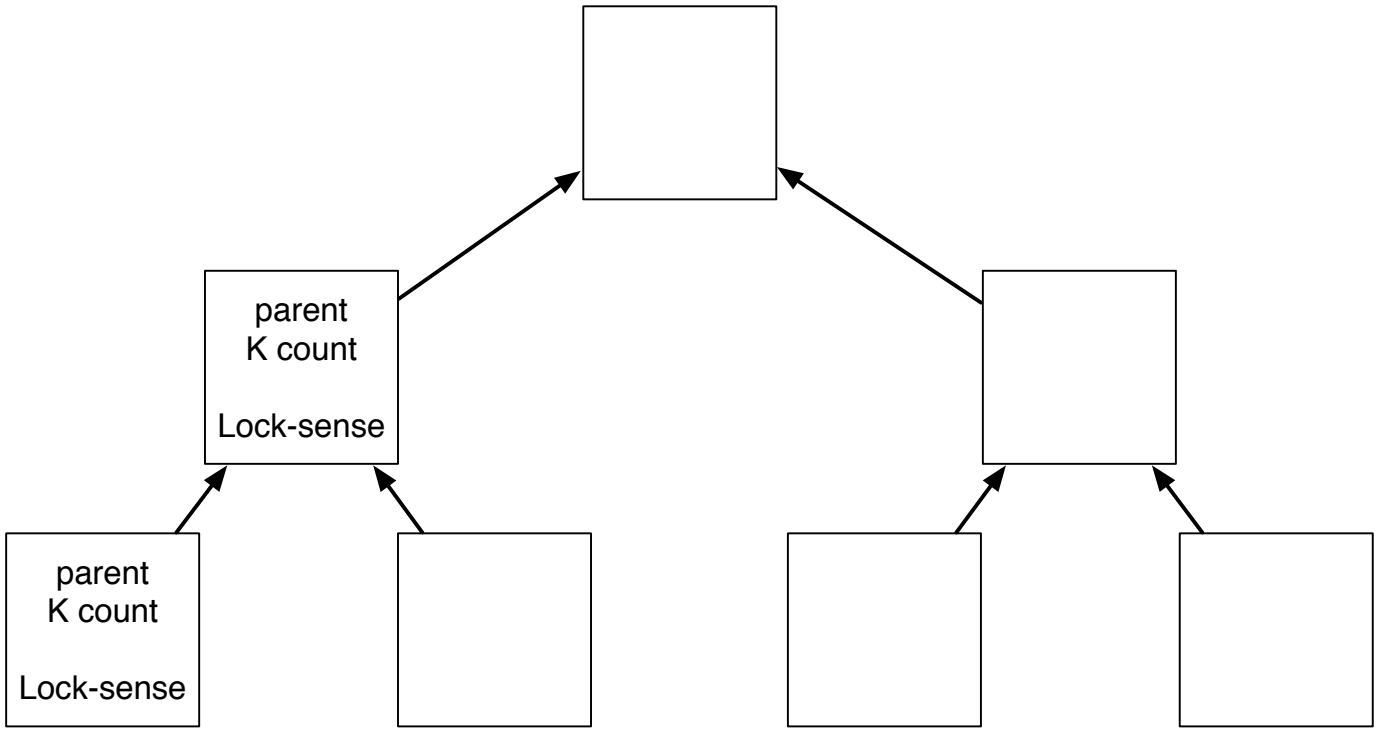
- Shared variable represented as a tree of vars
- Each node of the tree in a different cache line
- Processors divided into groups
- Each group assigned to a leaf of the tree
- Each processor updates the state of its leaf
- If last one to arrive in a group, it continues up the tree to update the parent

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



Example

```
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 rool

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 groups s leaf in
    ↪ the combining tree

procedure combining_barrier
     combining_barrier_aux (mynode) // join the barrier
     sense := not sense // for next barrier
```

Example

```
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

- Advantage: decreases contention and prevents tree saturation in interconnection networks (distributes the access)
- Disadvantage: spining 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