Synchronization Primitives

- Spin Locks
  - Used for mutual exclusion around critical sequential sections
  - Spin locks used for nonscheduled processes, basic primitive for building complicated locks

- Barriers
  - Used to ensure all threads have left a parallel section
  - Similar to pthreads join()
Spin Locks: Evaluation

- Fairness
  - Starvation
  - FIFO Ordering

- Scalability
  - Network load
  - Space

- Hardware Requirements
  - Cache Coherence
  - Instruction Set

- Overhead

- Spin Locks
  - Test and Set
  - Ticket Lock
  - Array based Queuing Lock
  - MCS Lock
Test and Set Lock

type lock = (unlocked, locked)

procedure acquirelock (L : lock)
    delay : integer = 1
    while (testandset (L) == locked) // returns old value
        pause delay // pause for short time
    delay = delay*2 // exponential back off

procedure releaselock (L : lock)
    lock = unlocked
Spin Locks: Evaluation

- Fairness
  - Starvation
  - FIFO Ordering
- Scalability
  - Network load
  - Space
- Hardware Requirements
  - Cache Coherence
  - Instruction Set
- Overhead

- Improvements
  - Backoff
    - Exponential
  - test and test_and_set
Ticket Lock

Note:
Global Variable

Increment
Ticket Lock

type lock = record
    nextticket : integer = 0 // the next ticket to be issued
    nowserving : integer = 0 // ticket number of thread with lock

procedure acquirelock (L : lock)
    myticket : integer = fetchandincrement(L->nextticket)
    // returns old value, arithmetic overflow is harmless
    loop
        pause (myticket – (L->nowserving) )
        // consume this many units of time
        // on most machines subtraction works correctly
        // despite overflow
        if (L->nowserving == myticket)
            return

procedure releaselock (L : lock)
    L->nowserving = L->nowserving + 1
Spin Locks: Evaluation

- Fairness
  - Starvation
  - FIFO Ordering
- Scalability
  - Network load
  - Space
- Hardware Requirements
  - Cache Coherence
  - Instruction Set
- Overhead

- Improvements
  - Backoff
    - Exponential
    - Proportional

- Hardware Requirements
  - Overhead
Array Based Queuing Lock

Note:
Dynamically Assigned Slot

Notify next using array
Array Based Queuing Lock
(Andersen)

type lock : record

slots : array [0...numprocs-1] of (haslock, mustwait) = (haslock, mustwait, mustwait, ... mustwait)
   // each element of slots should lie in a different memory module or cache line
nextslot : integer = 0

// function argument myplace points to a private variable in an enclosing scope

procedure acquirelock (L : lock, myplace : integer)

myplace = fetchandincrement (L->nextslot)
   // returns old value, this provides my assigned slot to spin on
if (myplace mod numprocs == 0) // decrement periodically to avoid overflow
   atomicadd (L->nextslot, -numprocs)
myplace = myplace mod numprocs
repeat while (L->slots[myplace] == mustwait) // spin until L->slots[myplace]==haslock
L->slots[myplace] == mustwait // init for next time

procedure releaselock (L : lock, myplace : integer)

L->slots[(myplace+1) mod numprocs] = haslock // give lock to next thread
Spin Locks: Evaluation

- Fairness
  - Starvation
  - FIFO Ordering
- Scalability
  - Network load
  - Space
- Hardware Requirements
  - Cache Coherence
  - Instruction Set
- Overhead
MCS Lock

type qnode = record
  next : qnode
  locked : Boolean

end type lock = qnode

// parameter I below points to a qnode record allocated in an enclosing scope in shared
// memory locally accessible to the invoking processor

procedure acquirelock (L : lock, I : qnode)

I->next = nil
predecessor : qnode = fetchandstore (L, I)
if (predecessor != nil) // queue was non empty
  I->locked = true
  predecessor->next = I
repeat while (I->locked) // spin on local qnode

procedure releaselock (L : lock, I : qnode)

if (I->next == nil ) // no known successor
  if ( compareandswap (L, I, nil) )
    return // compareandswap succeeds if no thread has enqueued, we can exit
  // else, we spin until next thread registers with us, then unlock it and exit
repeat while (I->next == nil)
I->next->locked = false
MCS Lock

(a) L

(b) 1(R) → L

(c) 1(R) → 2-B → 3-B → L

(d) 2(R) → 3-B → L

(e) 3(R) → L

(e') 3(E) → L → 4-B → 5-B
Spin Locks: Evaluation

- Fairness
  - Starvation
  - FIFO Ordering
- Scalability
  - Network load
  - Space
- Hardware Requirements
  - Cache Coherence
  - Instruction Set
- Overhead
## Advantage Comparison

<table>
<thead>
<tr>
<th>Lock</th>
<th>FIFO</th>
<th>Network Traffic</th>
<th>Space</th>
<th>Instruction Set</th>
<th>Overhead Ratio (single CPU compared to TSL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test and Set</td>
<td>No</td>
<td>O(n*t)</td>
<td>O(k)</td>
<td>test_and_set</td>
<td>1.0</td>
</tr>
<tr>
<td>Ticket</td>
<td>Yes</td>
<td>O(n*t)</td>
<td>O(k)</td>
<td>fetch_and_increment</td>
<td>1.11</td>
</tr>
<tr>
<td>Array Queue</td>
<td>Yes</td>
<td>O(n)</td>
<td>O(#cpus)</td>
<td>fetch_and_add</td>
<td>1.88</td>
</tr>
<tr>
<td>MCS</td>
<td>Yes</td>
<td>O(n)</td>
<td>O(n)</td>
<td>compare_and_swap</td>
<td>2.04</td>
</tr>
</tbody>
</table>

* n = number of threads contending for lock  
  t = number of polls made in spin  
  k = constant
Speed Comparison

Figure 4: Performance of spin locks on the Butterfly (empty critical section).
Overhead Comparison

Figure 5: Performance of selected spin locks on the Butterfly (empty critical section).
Barriers
Centralized Barrier (Naive)

count
Centralized Barrier (Sense Reversal)
Centralized Barrier

- Ops on critical path = $O(P)$
- Space = $O(1)$
- Network transactions = $O(P)$ with broadcast coherent caches, unbounded otherwise
- `fetch_and_increment` needed
Centralized Barrier

Pros

● Simple
● Less traffic on broadcast-based cache-coherent architectures
● Constant Space

Cons

● Many busy-wait accesses – significant network traffic
● Memory contention
● Cache thrashing
● Does not scale
Software-Combining Tree Barrier
Software Combining Tree Barrier

- Nodes scattered across different memory units or cache lines
- Ops on critical path = $O(\log P)$
- Space = $O(P)$
- Network transactions = $O(P)$ with broadcast coherent caches, unbounded otherwise
- `fetch_and_increment` needed
Software Combining Tree Barrier

Pros
• Significant decrease in memory contention
• Prevents tree saturation in multi-stage interconnects

Cons
• Processors spin on memory locations that cannot be statically determined
• Processors spin on memory locations that other processes spin on
Dissemination Barrier

- Uses two sets of variables to avoid double spinning
- Network transactions = $O(P \log P)$
- Ops in critical path = $O(\log P)$
- Space = $O(P \log P)$
- No need for fetch_and_Φ instructions
Dissemination Barrier

Pros

• All spinning is local
• Eliminates remote spinning

Cons

• With distributed shared memory and no coherent caches, must scatter state across memory banks
Tournament Barrier

- Combining tree of fan-in two
- Rather than waiting on a node, losers drop out and wait on shared variable
- Winners statically determined
- Tournament winner changes shared state
Tournament Barrier

- Network transactions = $O(P)$
- Ops in critical path = $O(\log P)$
- Space = $O(P)$ or $O(P \log P)$
- No need for fetch_and_Φ instructions
Tournament Barrier

Pros

- Efficient when broadcast is used for cache consistency
- Eliminates remote spinning

Cons

- Heavy interconnect traffic if no coherent caches
MCS Barrier - Arrival
MCS Barrier

- Network transactions = $2p - 2 = O(P)$, theoretical minimum without broadcast
- Space = $O(P)$
- Ops on critical path = $O(\log P)$
- No need for fetch_and_\Phi instructions
MCS Barrier

• With coherent caches, could replace wakeup with spin on global variable

• Minimal network traffic
# Advantage Comparison

<table>
<thead>
<tr>
<th><strong>Barrier</strong></th>
<th><strong>Critical Path Length</strong></th>
<th><strong># of Network Transactions</strong></th>
<th><strong>Space</strong></th>
<th><strong>Instruction Set</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized (count)</td>
<td>$O(P)$</td>
<td>$O(P)$ or Unbounded</td>
<td>$O(1)$</td>
<td>fetch_and_increment</td>
</tr>
<tr>
<td>Software Combining Tree</td>
<td>$O(\log P)$</td>
<td>$O(P)$ or Unbounded</td>
<td>$O(P)$</td>
<td>fetch_and_increment</td>
</tr>
<tr>
<td>Dissemination</td>
<td>$O(\log P)$</td>
<td>$O(P \log P)$</td>
<td>$O(P \log P)$</td>
<td>Atomic read/write</td>
</tr>
<tr>
<td>Tournament</td>
<td>$O(\log P)$</td>
<td>$O(P)$</td>
<td>$O(P \log P)$</td>
<td>Atomic read/write</td>
</tr>
<tr>
<td>MCS Tree</td>
<td>$O(\log P)$</td>
<td>$O(P)$</td>
<td>$O(P)$</td>
<td>Atomic read/write</td>
</tr>
</tbody>
</table>
Speed Comparison (Non Cache Coherent)

Figure 9: Performance of selected barriers on the Butterfly.
Figure 10: Performance of barriers on the Symmetry.
Questions?
Images Cited

- Mellor Crummey, John. Comp 422 Lecture Slides, Rice University (Dissemination Barrier Image)