Distributed Operating Systems

Synchronization in Parallel Systems

Marcus Völp

2013

Topics

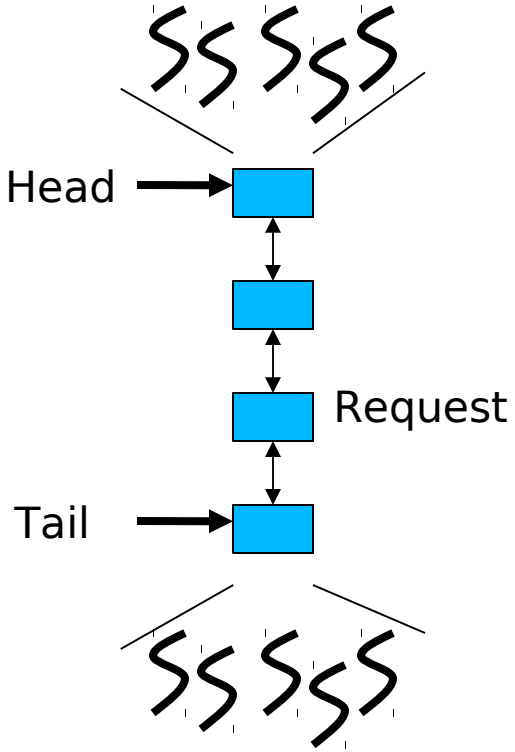
- Synchronization
- Locks
- Performance

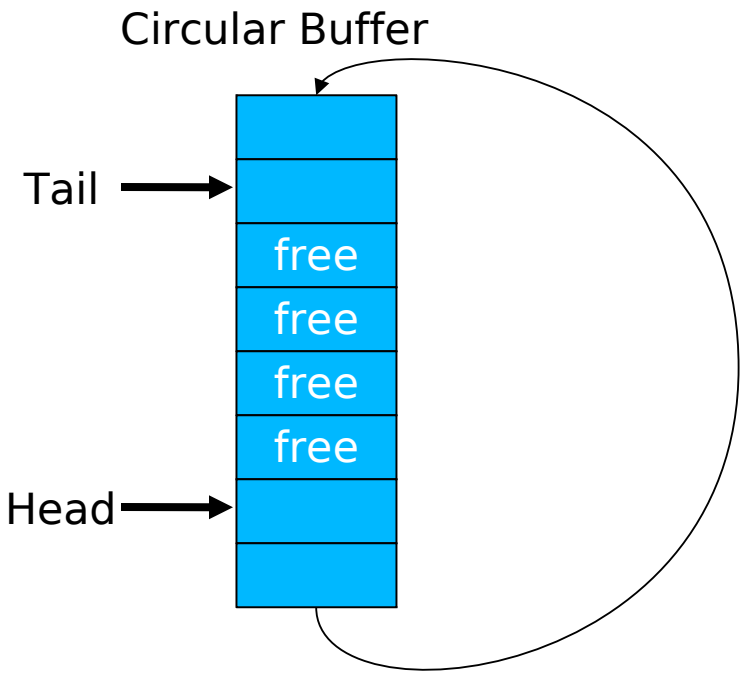
Overview

- Introduction
- Hardware Primitives
- Synchronization with Locks (Part I)
  - Properties
  - Locks
    - Spin Lock (Test & Set Lock)
    - Test & Test & Set Lock
    - Ticket Locks
- Synchronization without Locks
- Synchronization with Locks (Part II)
    - MCS Locks
    - Performance
  - Special Issues
    - Timeouts
    - Reader Writer Locks
    - Lockholder Preemption
    - Monitor, Mwait

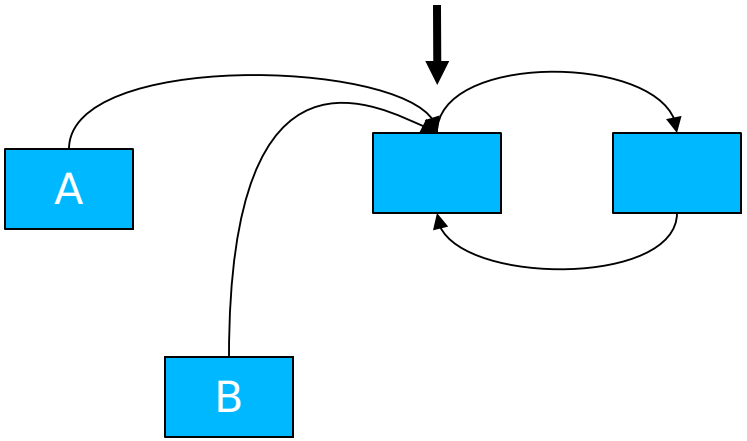
Introduction

- Example: Request Queue



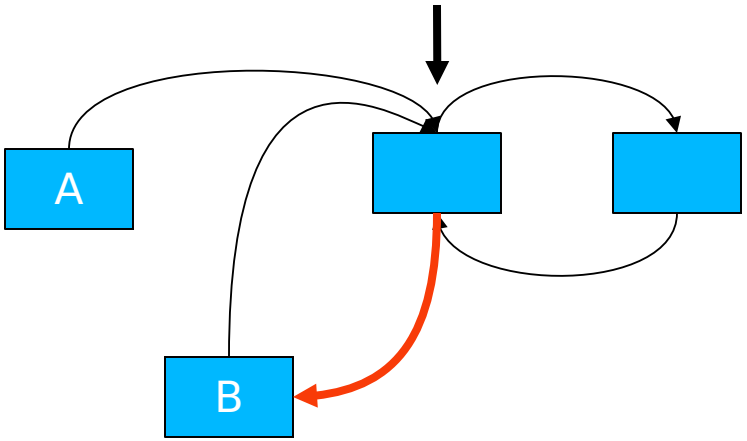


Introduction



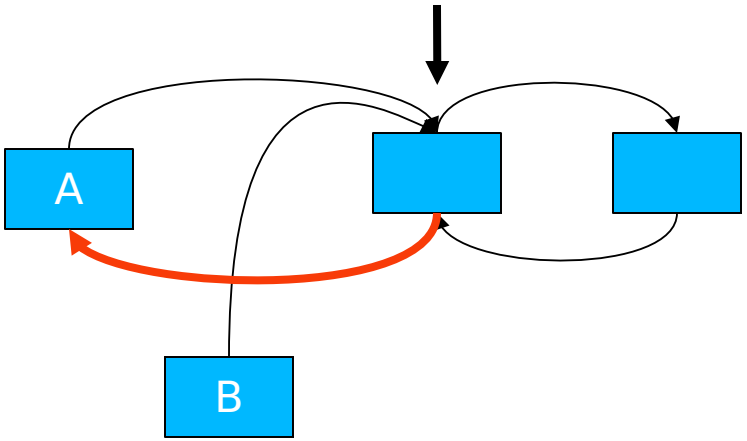
1) A,B create list elements
2) A,B set next pointer to head

Introduction



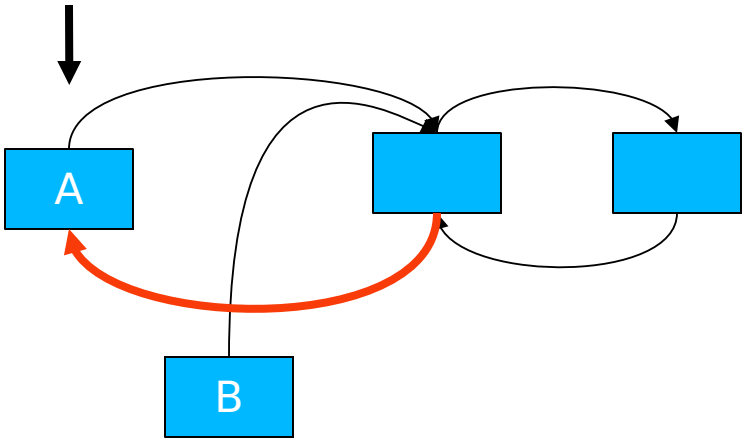
1) A,B create list elements
2) A,B set next pointer to head
3) B set prev pointer

Introduction



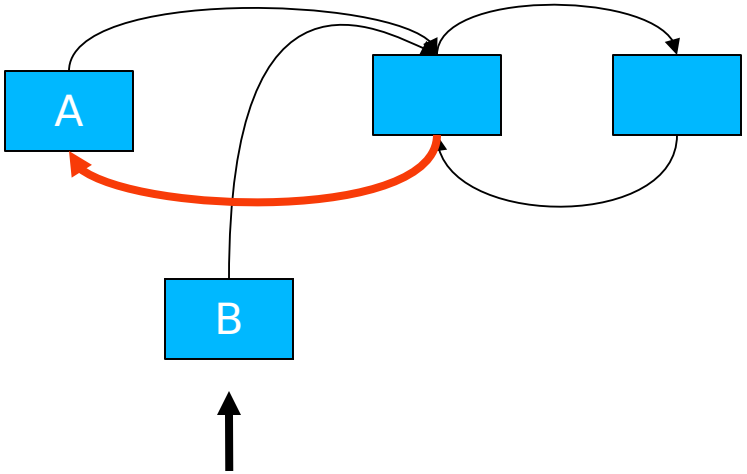
1) A,B create list elements
2) A,B set next pointer to head
3) B set prev pointer
4) A set prev pointer

Introduction



1) A,B create list elements
2) A,B set next pointer to head
3) B set prev pointer
4) A set prev pointer
5) A update head pointer

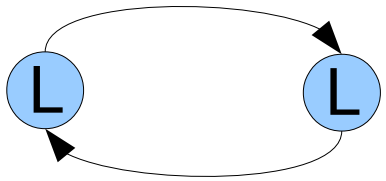
Introduction



1) A,B create list elements
2) A,B set next pointer to head
3) B set prev pointer
4) A set prev pointer
5) A update head pointer
6) B update head pointer

Introduction

- First Solution
  - Locks
    - coarse grained: lock entire list
      - lock(list);
        list->insert_element;
        unlock(list);
    - fine grained: lock list elements
      - retry:
        lock(head);
        if (trylock(head->next)) {
          head->insert_element;
          unlock(head->next);
        } else {
          unlock(head);
          goto retry;
        }



Mutual Exclusion
without Locks / Atomic Read-Modify-Write Instructions

- Last lecture: Decker / Peterson
  - requires:
    - atomic stores, atomic loads
    - sequential consistency (or memory fences)

```
bool flag[2] = {false, false};
int turn = 0;

void entersection(int thread) {
      int other = 1 - thread;                    /* id of other thread; thread in {0,1}*/
      flag[thread] = true;                       /* show interest */
      turn= other;                               /* give precedence to other thread */
      while (turn == other && flag[other]) {}; /* wait */
}

void leavesection(int thread) {
      flag[thread] = false;
}
```

Atomic Hardware Instructions

- [Lipton 95] a, b are atomic if A || B = A;B or B;A
- Read-Modify-Write Instructions are typically not atomic:
  - A || B (x = 0)

```
add &x, 1        ||   mov &x, 2
```

are typically executes as:

```
load &x  → Reg
add Reg + 1      ||   store 2 → &x
store Reg → &x
```

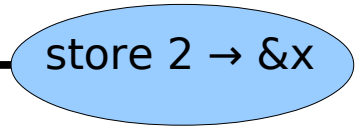
Atomic Hardware Instructions

- [Lipton 95] a, b are atomic if A || B = A;B or B;A
- Read-modify-write Instructions are typically not atomic:
  - A || B (x = 0)

```
add &x, 1   ||   mov &x, 2
```

are typically executes as:

```
load &x  → Reg
add Reg + 1
store Reg → &x
```



  - Above interleaving for A || B => x = 1
  - but A;B => x = 2,
    B;A => x = 3

Atomic Hardware Instructions

- How to make instructions atomic
  - **Bus lock**
    - Lock memory bus until all memory accesses of an RMW instruction have completed (e.g., Intel Pentium 3 and older x86 CPUs)

      lock; add [eax], 1
  - **Cache Lock**
    - Delay snoop traffic until all memory accesses of RMW instruction have completed (e.g., Intel Pentium 4 and newer x86 CPUs)
  - **Observe Cache**
    - Install cache watchdog on load
    - Abort store if watchdog has detected a concurrent access; retry OP (e.g., ARM, Alpha, monitor + mwait on x86)

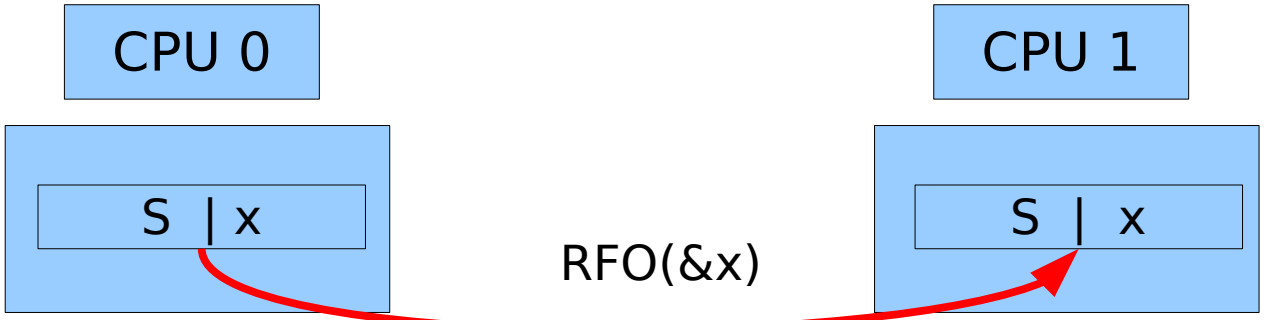
      ```
      retry:
          load_linked &x → R;
            modify R;
          if (! store_conditional(R → &x))
            goto retry:
      ```
  - **HW Transactional Memory**
    - watchdog for multiple cachelines
    - discard changes on concurrent access

Atomic Hardware Instructions

- How to make instructions atomic
  - **Observe Cache**
    - Delay snoop traffic until all memory accesses of RMW instruction have completed (e.g., Intel Pentium 4 and newer x86 CPUs)
  - last lecture: M(O)ESI Cache Coherence Protocol

add &x, 1

1. read_for_ownership(&x) [→ E] [→ I]
2. load &x → R
3. add R += 1
4. store R → &x



Atomic Hardware Instructions

- How to make instructions atomic
  - **Cache Lock**
    - Delay snoop traffic until all memory accesses of RMW instruction have completed (e.g., Intel Pentium 4 and newer x86 CPUs)
  - last lecture: M(O)ESI Cache Coherence Protocol

<u>add &x, 1</u>

1. read_for_ownership(&x) [→ E] [→ I]
2. load &x → R
3. add R += 1
4. store R → &x

Atomic Hardware Instructions

- How to make instructions atomic
  - **Cache Lock**
    - Delay snoop traffic until all memory accesses of RMW instruction have completed (e.g., Intel Pentium 4 and newer x86 CPUs)
  - last lecture: M(O)ESI Cache Coherence Protocol

add &x, 1

1. read_for_ownership(&x) [→ E]
2. load &x → R
3. add R += 1
4. store R → &x

[→ I]

read / write &x

Atomic Hardware Instructions

- How to make instructions atomic
  - **Cache Lock**
    - Delay snoop traffic until all memory accesses of RMW instruction have completed (e.g., Intel Pentium 4 and newer x86 CPUs)
  - last lecture: M(O)ESI Cache Coherence Protocol

<u>add &x, 1</u>

1. read_for_ownership(&x) [→ E]
2. load &x → R
3. add R += 1
4. store R → &x

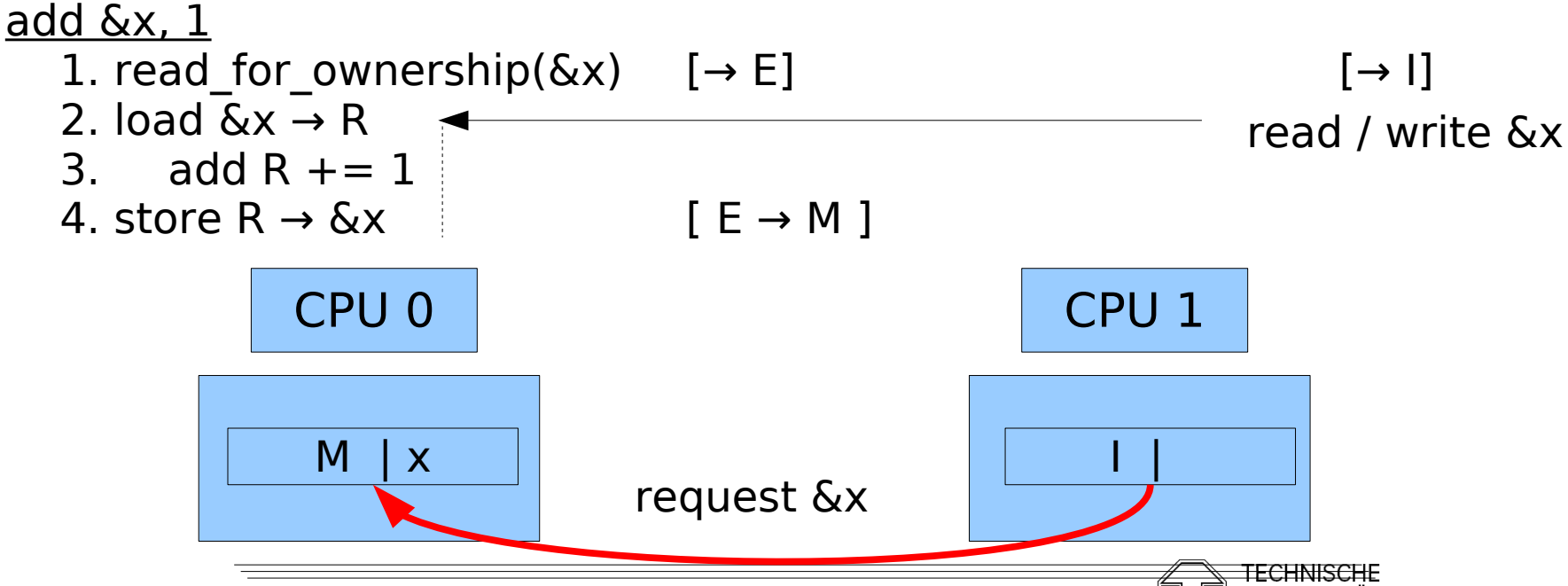
[→ I]

read / write &x

delay reply until store completes

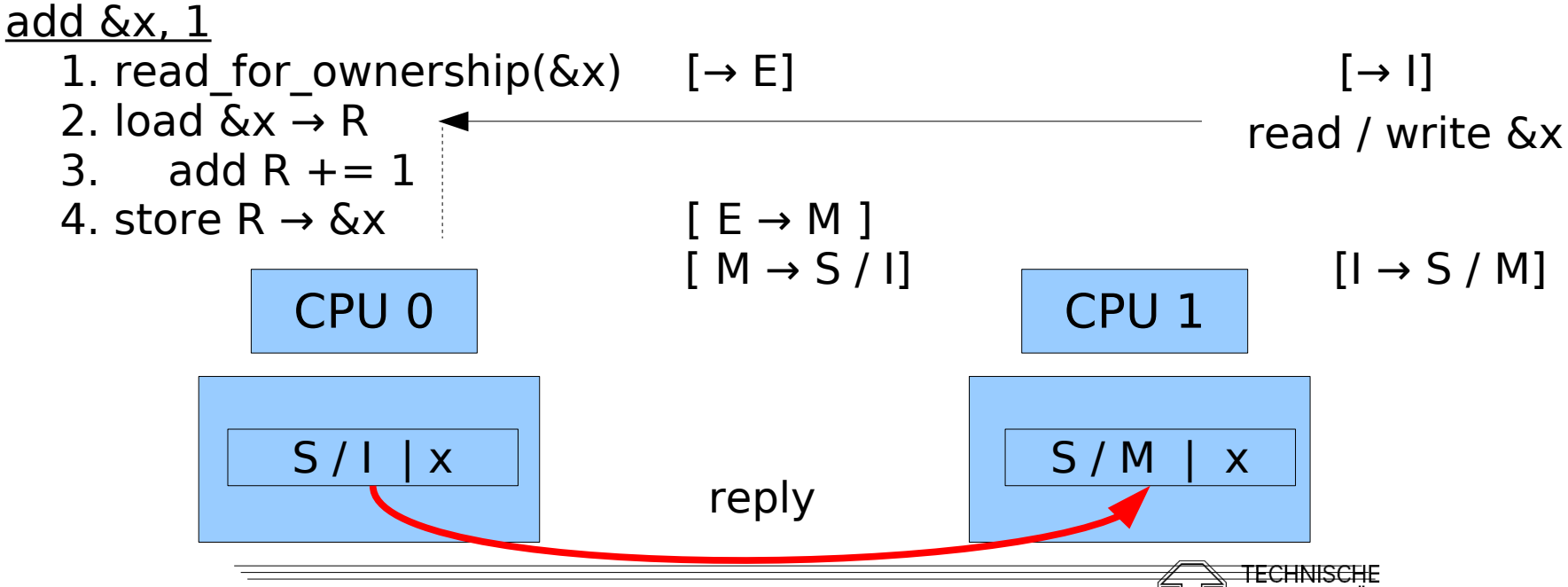
Atomic Hardware Instructions

- How to make instructions atomic
  - **Cache Lock**
    - Delay snoop traffic until all memory accesses of RMW instruction have completed (e.g., Intel Pentium 4 and newer x86 CPUs)
  - last lecture: M(O)ESI Cache Coherence Protocol



Atomic Hardware Instructions

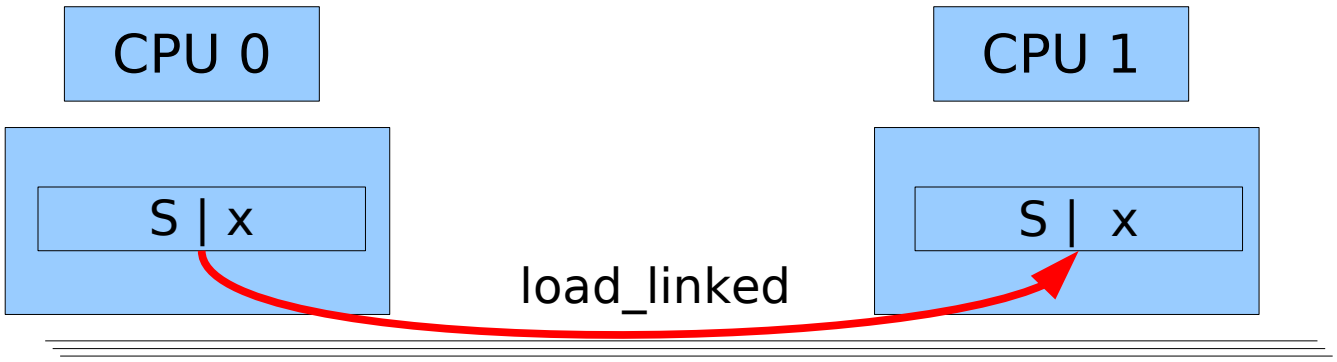
- How to make instructions atomic
  - **Cache Lock**
    - Delay snoop traffic until all memory accesses of RMW instruction have completed (e.g., Intel Pentium 4 and newer x86 CPUs)
  - last lecture: M(O)ESI Cache Coherence Protocol



Atomic Hardware Instructions

- How to make instructions atomic
  - **Observe Cache**
    - Install cache watchdog on load
    - Abort store if watchdog has detected a concurrent access; retry OP (e.g., ARM, Alpha, monitor + mwait on x86)

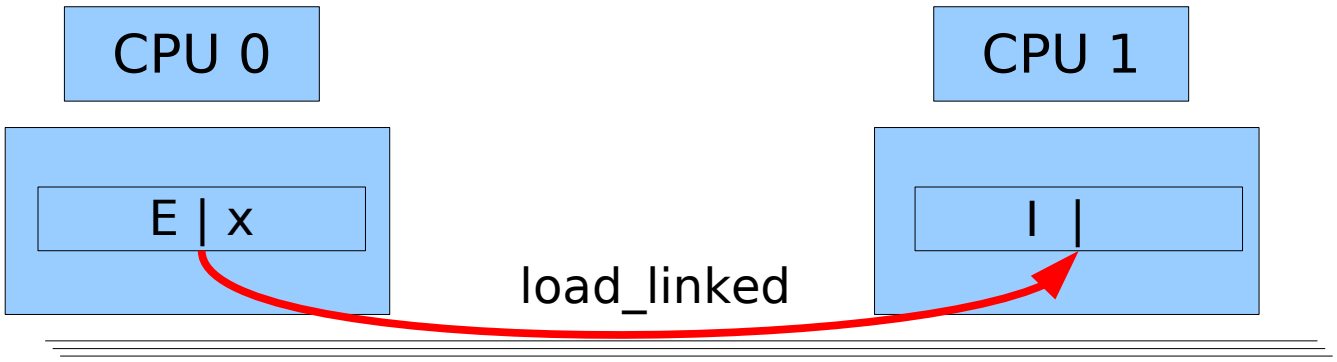
1. load_linked &x → R [→ E] [→ I]
2. add R += 1
3. store_conditional R → &x [if (E) → M else abort]



Atomic Hardware Instructions

- How to make instructions atomic
  - **Observe Cache**
    - Install cache watchdog on load
    - Abort store if watchdog has detected a concurrent access; retry OP (e.g., ARM, Alpha, monitor + mwait on x86)

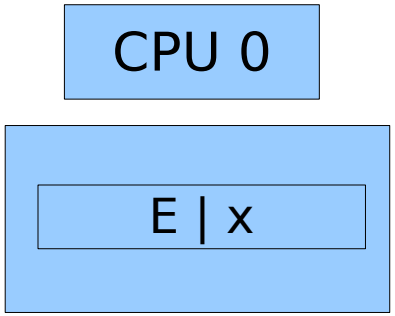
1. load_linked &x → R [→ E] [→ I]
2. add R += 1
3. store_conditional R → &x [if (E) → M else abort]



Atomic Hardware Instructions

- How to make instructions atomic
  - **Observe Cache**
    - Install cache watchdog on load
    - Abort store if watchdog has detected a concurrent access; retry OP (e.g., ARM, Alpha, monitor + mwait on x86)

1. load_linked &x → R [→ E] [→ I]
2. add R += 1
3. store_conditional R → &x [if (E) → M else abort]

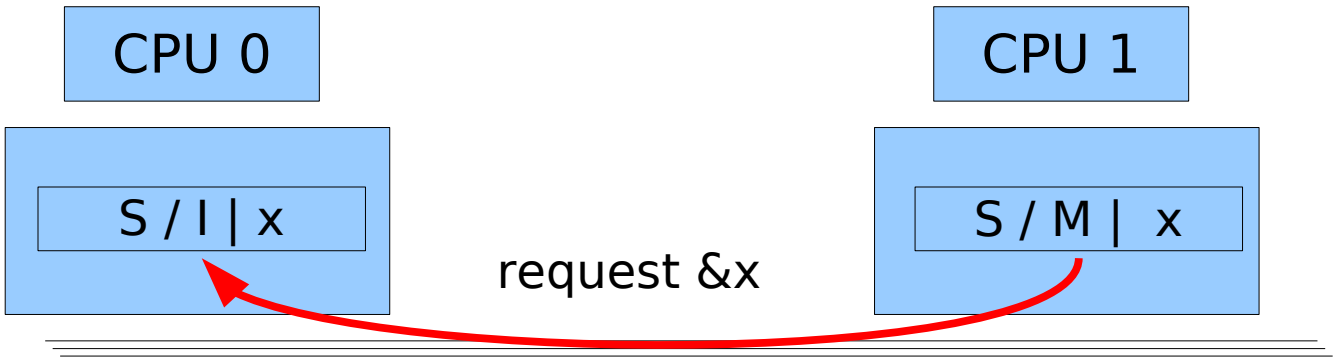


&x is only in local cache
=> store_conditional succeeds

Atomic Hardware Instructions

- How to make instructions atomic
  - **Observe Cache**
    - Install cache watchdog on load
    - Abort store if watchdog has detected a concurrent access; retry OP (e.g., ARM, Alpha, monitor + mwait on x86)

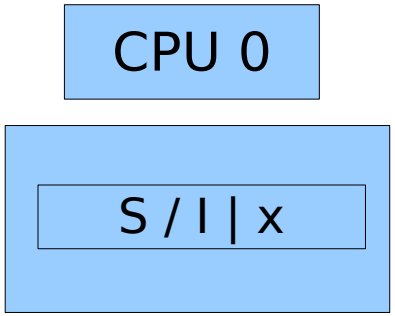
1. load_linked &x → R [→ E] [→ I]
2. add R += 1 ◄ read / write &x
3. store_conditional R → &x [if (E) → M else abort]



Atomic Hardware Instructions

- How to make instructions atomic
  - **Observe Cache**
    - Install cache watchdog on load
    - Abort store if watchdog has detected a concurrent access; retry OP (e.g., ARM, Alpha, monitor + mwait on x86)

1. load_linked &x → R [→ E]
2. add R += 1 ◄ read / write &x [→ I]
3. store_conditional R → &x [if (E) → M else abort]



&x can be in remote caches
=> store_conditional fails

Atomic Hardware Instructions

- Read-Modify-Write Instructions
  - bit test and set - bts (bit)
    - if (bit clear) { set bit ; return true; } else { return false; }
  - Exchange - swap (mem, R)
    - &mem → tmp; R → &mem; tmp → R;
  - fetch and add - xadd (mem, R)
    - &mem → tmp; &mem += R; return tmp;
  - compare and swap - cas (mem, expected, desired)
    - if (&mem == expected) {
      desired → &mem; return true;
      } else {
      return false;
      }
  - double “address” compare and swap –
    cas (mem1, mem2, exp1, exp2, des1, des2)
    - swap mem1 ↔ des1, mem2 ↔ des2 iff
      mem1 == exp1 & mem2 == exp2

Overview

- Introduction
- Hardware Primitives
- Synchronization with Locks (Part I)
  - Properties
  - Locks
    - Spin Lock (Test & Set Lock)
    - Test & Test & Set Lock
    - Ticket Locks
- Synchronization without Locks
- Synchronization with Locks (Part II)
    - MCS Locks
    - Performance
  - Special Issues
    - Timeouts
    - Reader Writer Locks
    - Lockholder Preemption
    - Monitor, Mwait

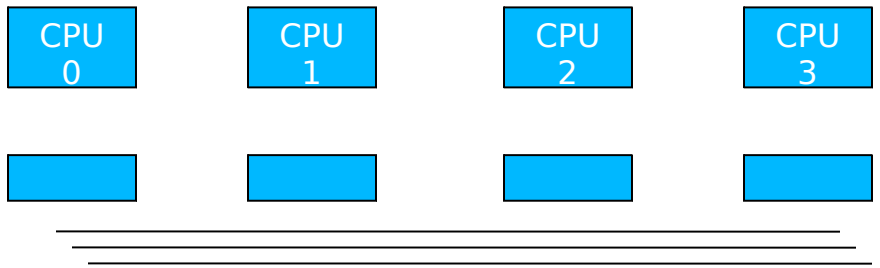
Synchronization with Locks

- Properties
  - **overhead**
    - fine-grained locking => critical sections are short
    - minimize overhead to take the lock if it is free
  - **fairness**
    - every thread should obtain the lock after a finite amount of time
    - (real-time:) … latest after x * |CS| seconds
  - **timeouts / abort lock() operation**
    - kill threads that compete for the lock
    - run fixup code if thread fails to acquire the lock before timeout
  - **reader / writer locks**
    - concurrent readers may enter the lock at the same time
  - **lockholder preemption**
    - avoid blocking other threads on a descheduled lockholder
  - **priority inversion**
    - ! Not covered in this lecture (RTS / MKK)
  - **spinning vs. blocking**
    - release CPU while others hold the lock

Synchronization with Locks

- Spin Lock (Test and Set Lock)
  - atomic swap

```
lock (lock_var & L) {
  do {
     reg = 1;
     swap (L, reg)
  } while (reg == 1);
}

unlock (lock_var & L) {
  L = 0;
}
```

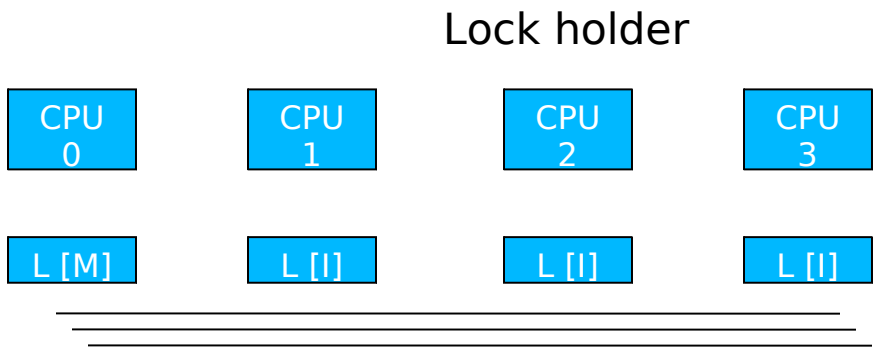


Pro: 1 cheap atomic OP to acquire the lock
Cons: high bus traffic while lock is held

Synchronization with Locks

- Spin Lock (Test and Set Lock)
  - atomic swap

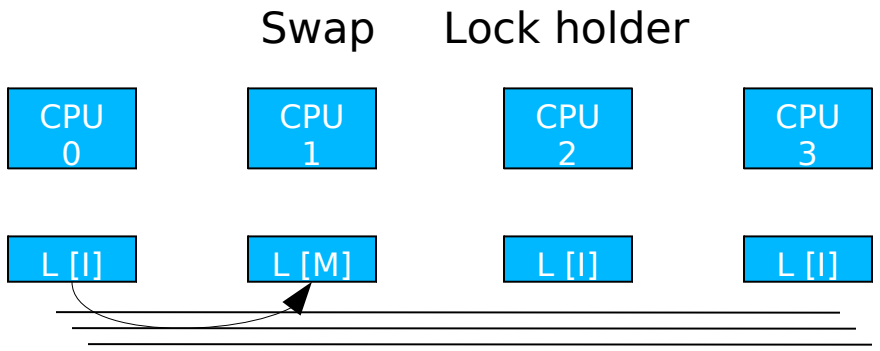
```
lock (lock_var & L) {
  do {
     reg = 1;
     swap (L, reg)
  } while (reg == 1);
}

unlock (lock_var & L) {
  L = 0;
}
```



Synchronization with Locks

- Spin Lock (Test and Set Lock)
  - atomic swap

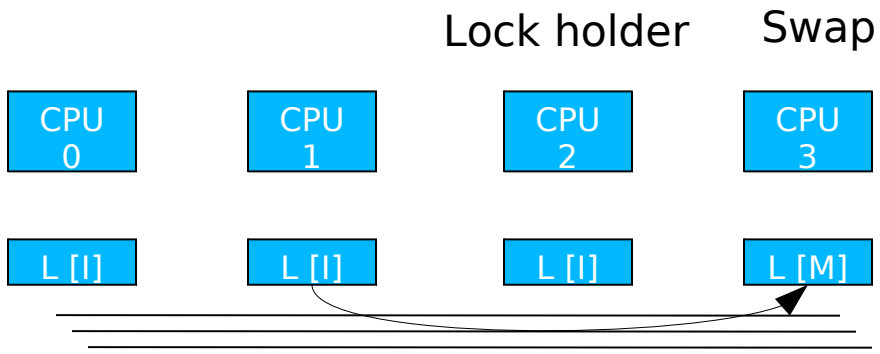
```
lock (lock_var & L) {
  do {
     reg = 1;
     swap (L, reg)
  } while (reg == 1);
}

unlock (lock_var & L) {
  L = 0;
}
```



Synchronization with Locks

- Spin Lock (Test and Set Lock)
  - atomic swap

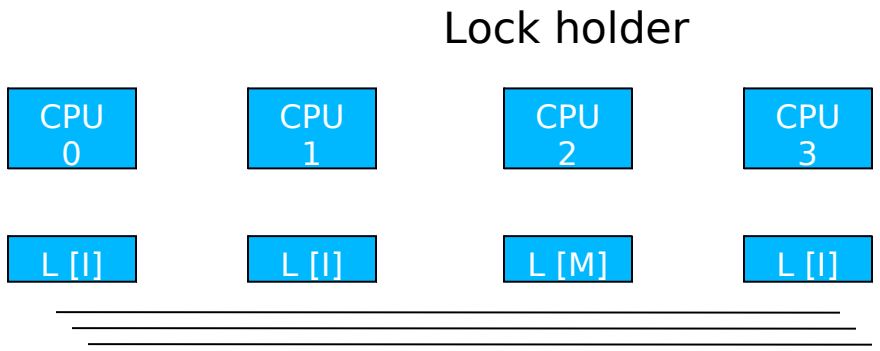
```
lock (lock_var & L) {
  do {
     reg = 1;
     swap (L, reg)
  } while (reg == 1);
}

unlock (lock_var & L) {
  L = 0;
}
```



Synchronization with Locks

- Spin Lock (Test and Test and Set Lock)
  - atomic swap

```
lock (lock_var & L) {
  do {
     reg = 1;
     do { } while (L == 1);
     swap (L, reg)
  } while (reg == 1);
}

unlock (lock_var & L) {
  L = 0;
}
```

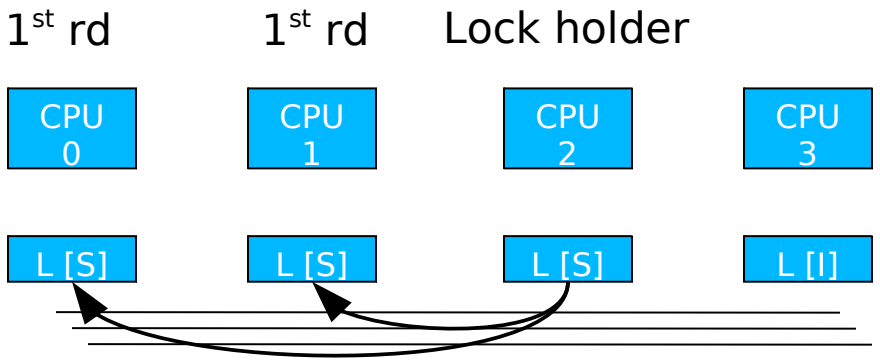


Spin locally while lock is held
=> reduces bus traffic

Synchronization with Locks

- Spin Lock (Test and Test and Set Lock)
  - atomic swap

```
lock (lock_var & L) {
  do {
     reg = 1;
     do { } while (L == 1);
     swap (L, reg)
  } while (reg == 1);
}

unlock (lock_var & L) {
  L = 0;
}
```

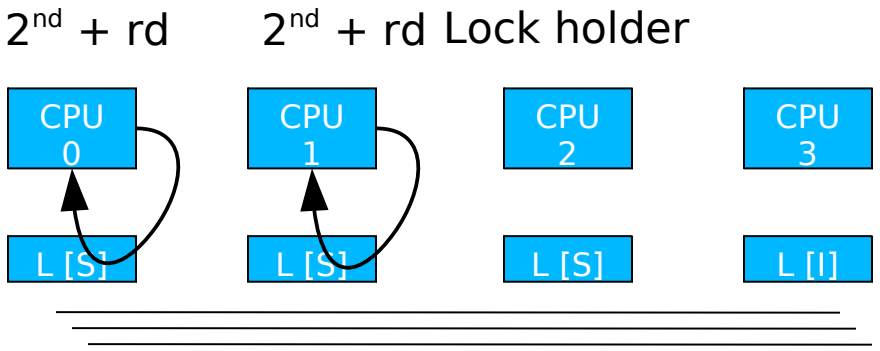


Spin locally while lock is held
=> reduces bus traffic

Synchronization with Locks

- Spin Lock (Test and Test and Set Lock)
  - atomic swap

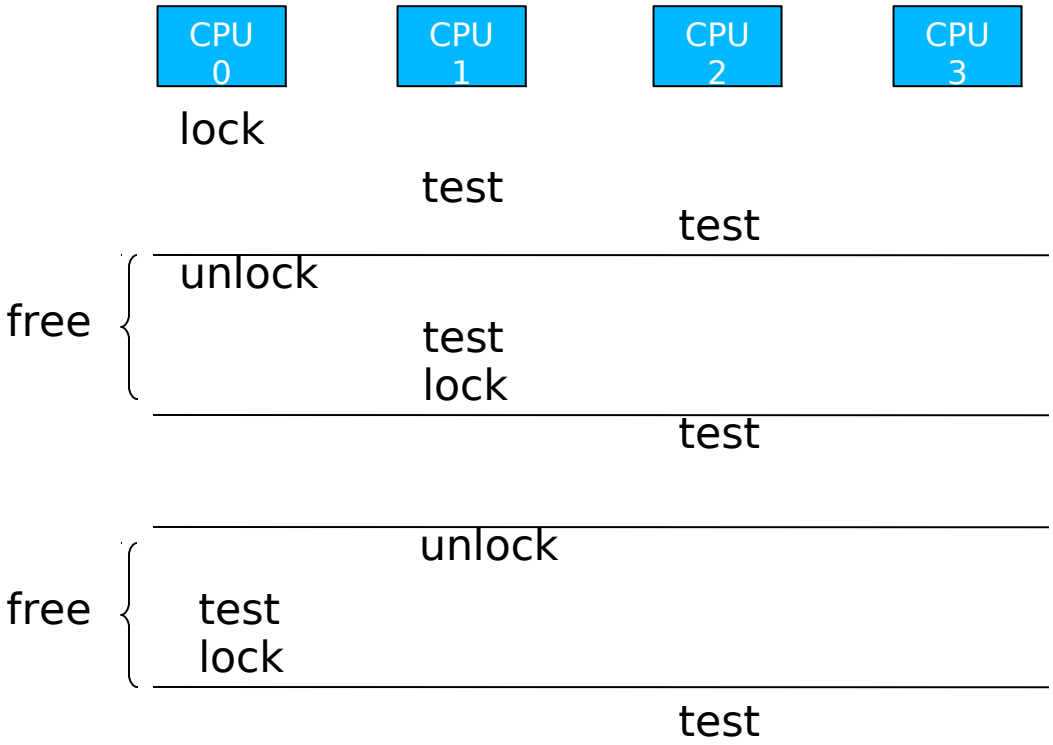
```
lock (lock_var & L) {
  do {
     reg = 1;
     do { } while (L == 1);
     swap (L, reg)
  } while (reg == 1);
}

unlock (lock_var & L) {
  L = 0;
}
```



Spin locally while lock is held
=> reduces bus traffic

Synchronization with Locks

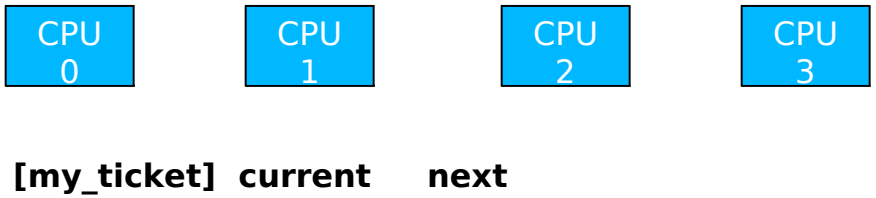
- Fairness



Synchronization with Locks

- Fairness: Ticket Lock
  - fetch and add (xadd)

```
lock_struct {
  next_ticket,
  current_ticket
}

ticket_lock (lock_struct & l) {
  my_ticket = xadd (&l.next_ticket, 1)
  do { } while (l.current_ticket != my_ticket);
}

unlock (lock_struct & l) {
  current_ticket ++;
}
```



Synchronization with Locks

- Fairness: Ticket Lock
  - fetch and add (xadd)

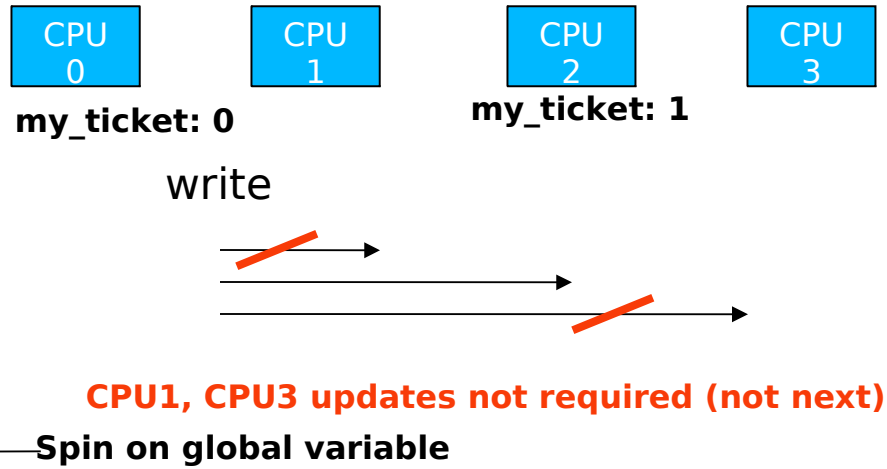
```
lock_struct {
  next_ticket,
  current_ticket
}

ticket_lock (lock_struct & l) {
  my_ticket = xadd (&l.next_ticket, 1)
  do { } while (l.current_ticket != my_ticket);
}

unlock (lock_struct & l) {
  current_ticket ++;
}
```

[my_ticket]	current	next	
	0	0	
L.CPU0 [0]:	0	1	=> Lockholder = CPU0
L.CPU1 [1]:	0	2	
L.CPU2 [2]:	0	3	
U.CPU0 [0]:	1	3	=> Lockholder = CPU1
L.CPU3 [3]:	1	4	
L.CPU0 [4]:	1	5	
U.CPU1 [1]:	2	5	=> Lockholder = CPU 2

Synchronization with Locks

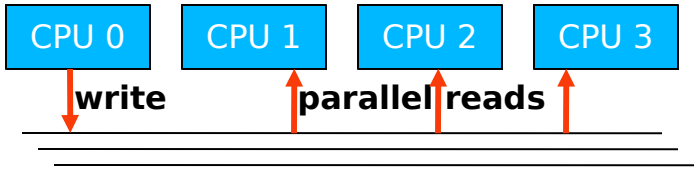
- Fairness: Ticket Lock
  - fetch and add (xadd)

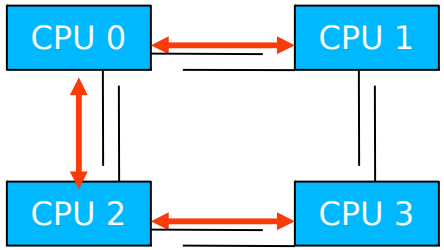
```
lock_struct {
  next_ticket,
  current_ticket
}

ticket_lock (lock_struct & l) {
  my_ticket = xadd (&l.next_ticket, 1)
  do { } while (l.current_ticket != my_ticket);
}

unlock (lock_struct &l) {
  current_ticket ++;
}
```



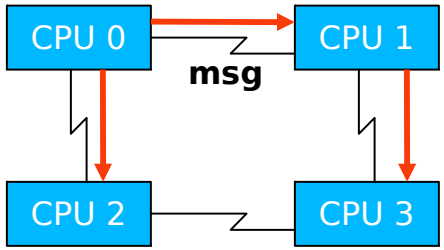
However:
- Signal all CPUs not only next
- Abort / timeout of competing threads

More Local Spinning

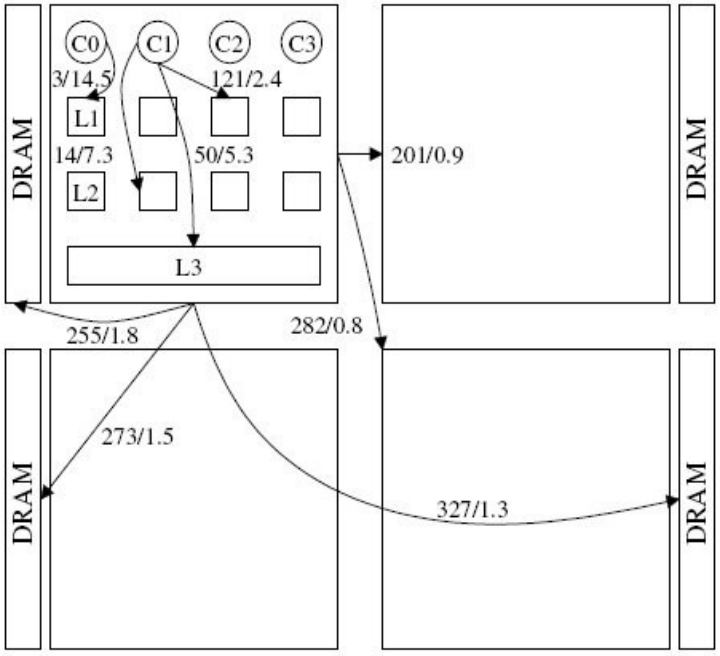




Need to forward write on Bus 2-3



3 Network Messages



16 core AMD Opteron:
4 chips with 4 cores + partitioned RAM
internal crossbar to access L1 / L2 on local chip

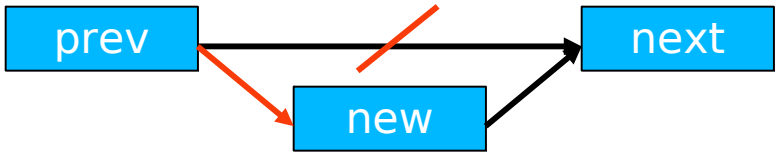
source: [corey08]

Overview

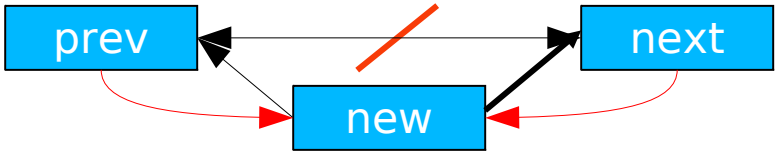
- Introduction
- Hardware Primitives
- Synchronization with Locks (Part I)
  - Properties
  - Locks
    - Spin Lock (Test & Set Lock)
    - Test & Test & Set Lock
    - Ticket Locks
- Synchronization without Locks
- Synchronization with Locks (Part II)
    - MCS Locks
    - Performance
  - Special Issues
    - Timeouts
    - Reader Writer Locks
    - Lockholder Preemption
    - Monitor, Mwait

Synchronization without Locks

- A quick intermezzo to lock-free synchronization




```
insert(new_elem, prev) {
  retry:
    new_elem.next = prev.next;
    if (not CAS(prev.next == prev.next, new_elem)) goto retry;
}
```




```
insert(new_elem, prev) {
  retry:
    next = prev.next;
    new_elem.next = prev.next;
    new_elem.prev = prev;
    if (not DCAS(prev.next == next && next.prev ==prev,
                 prev.next = new_elem, next.prev = new_elem))
      goto retry;
}
```

Synchronization without Locks

- Load Linked, Store Conditional

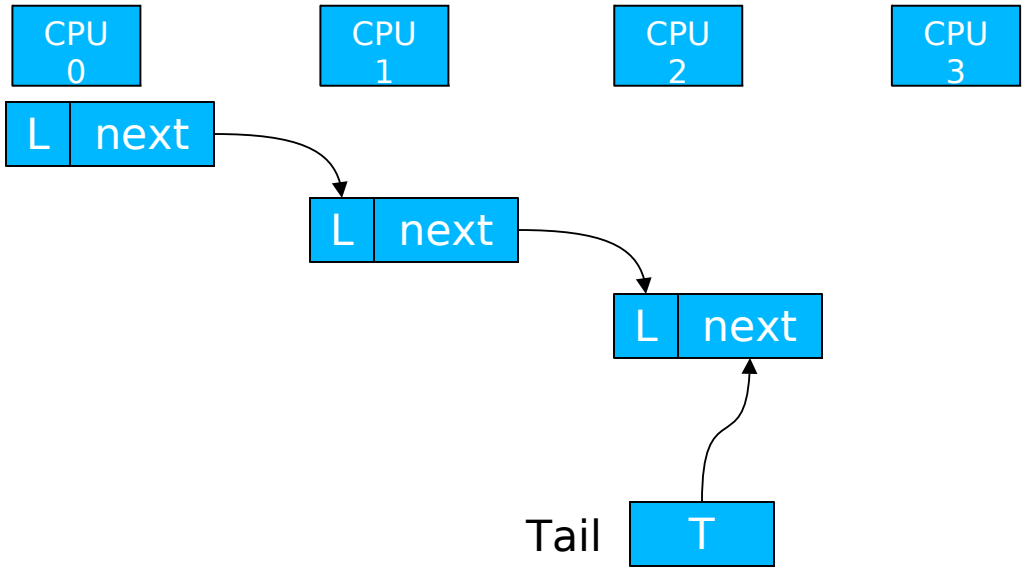
```
insert (prev, new_elem) {
  retry:
   load_linked (prev.next);
   new_elem.next = prev.next;
   if (! store_conditional (prev.next, new_elem)) goto retry;
}
```

Overview

- Introduction
- Hardware Primitives
- Synchronization with Locks (Part I)
  - Properties
  - Locks
    - Spin Lock (Test & Set Lock)
    - Test & Test & Set Lock
    - Ticket Locks
- Synchronization without Locks
- Synchronization with Locks (Part II)
    - MCS Locks
    - Performance
  - Special Issues
    - Timeouts
    - Reader Writer Locks
    - Lockholder Preemption
    - Monitor, Mwait

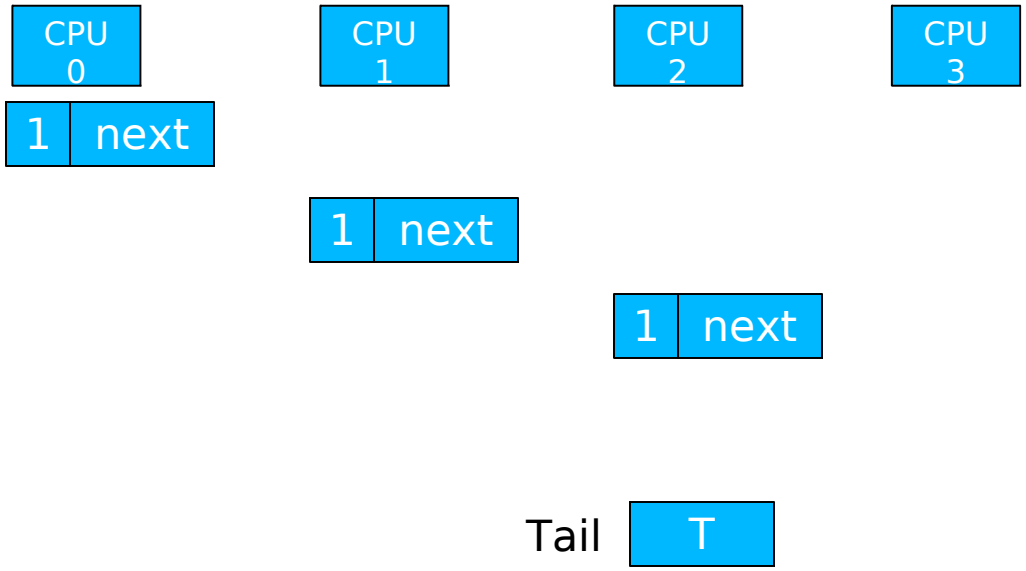
MCS-Lock

- Fairness + Local Spinning
  by Mellor-Crummey and Scott



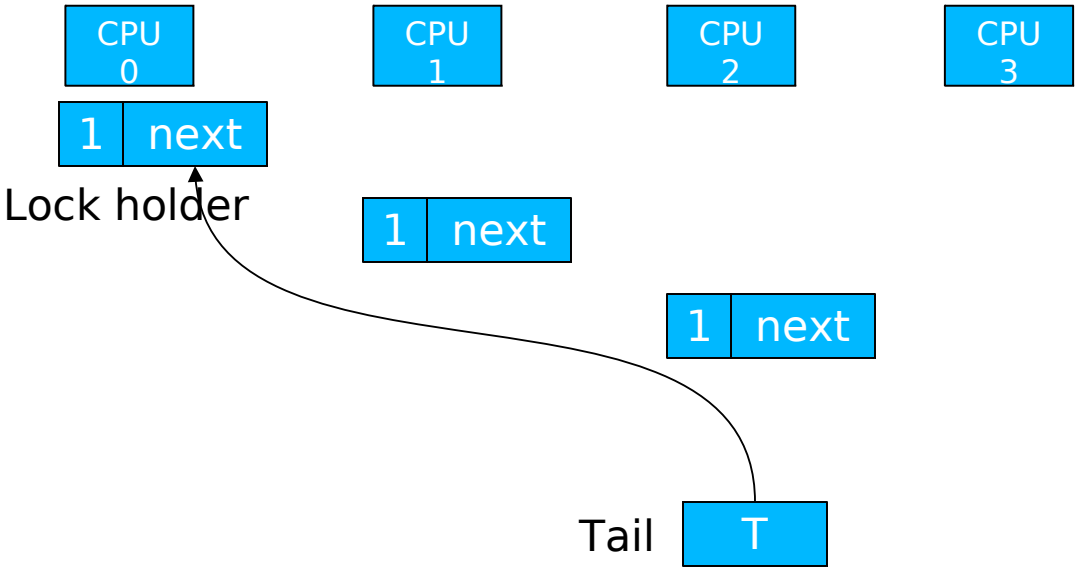
MCS-Lock

- Fairness + Local Spinning
  by Mellor-Crummey and Scott



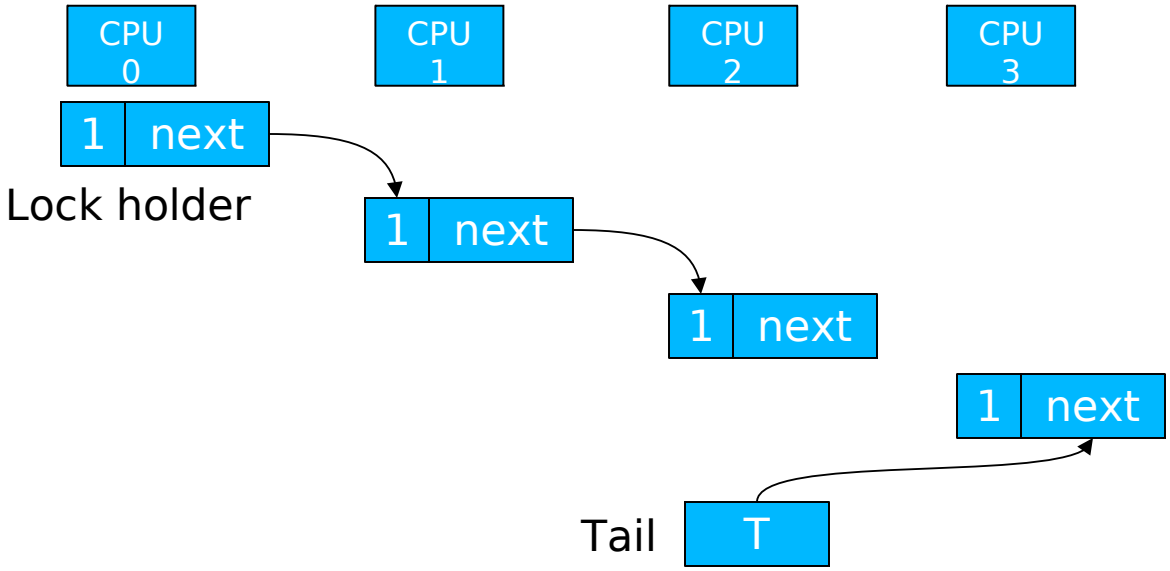
MCS-Lock

- Fairness + Local Spinning
  by Mellor-Crummey and Scott



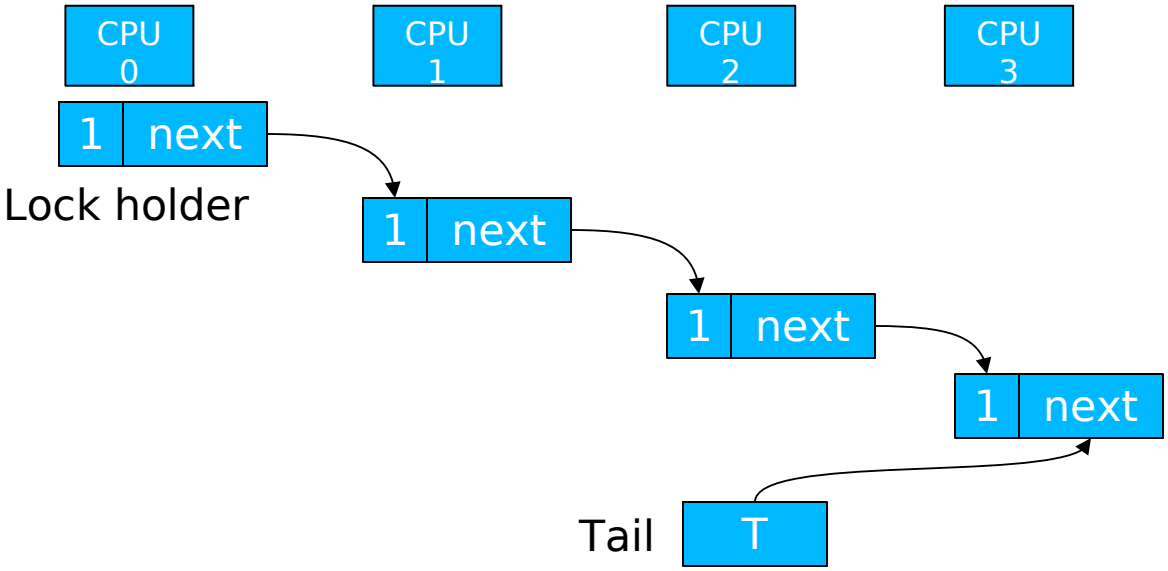
MCS-Lock

- Fair Lock with Local Spinning



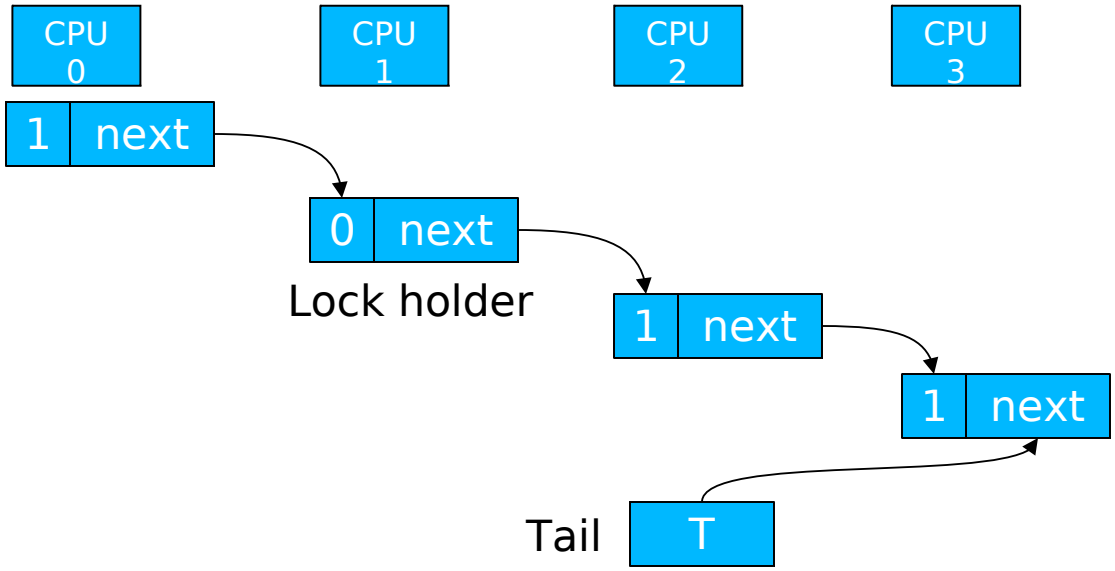
MCS-Lock

- Fair Lock with Local Spinning



MCS-Lock

- Fair Lock with Local Spinning



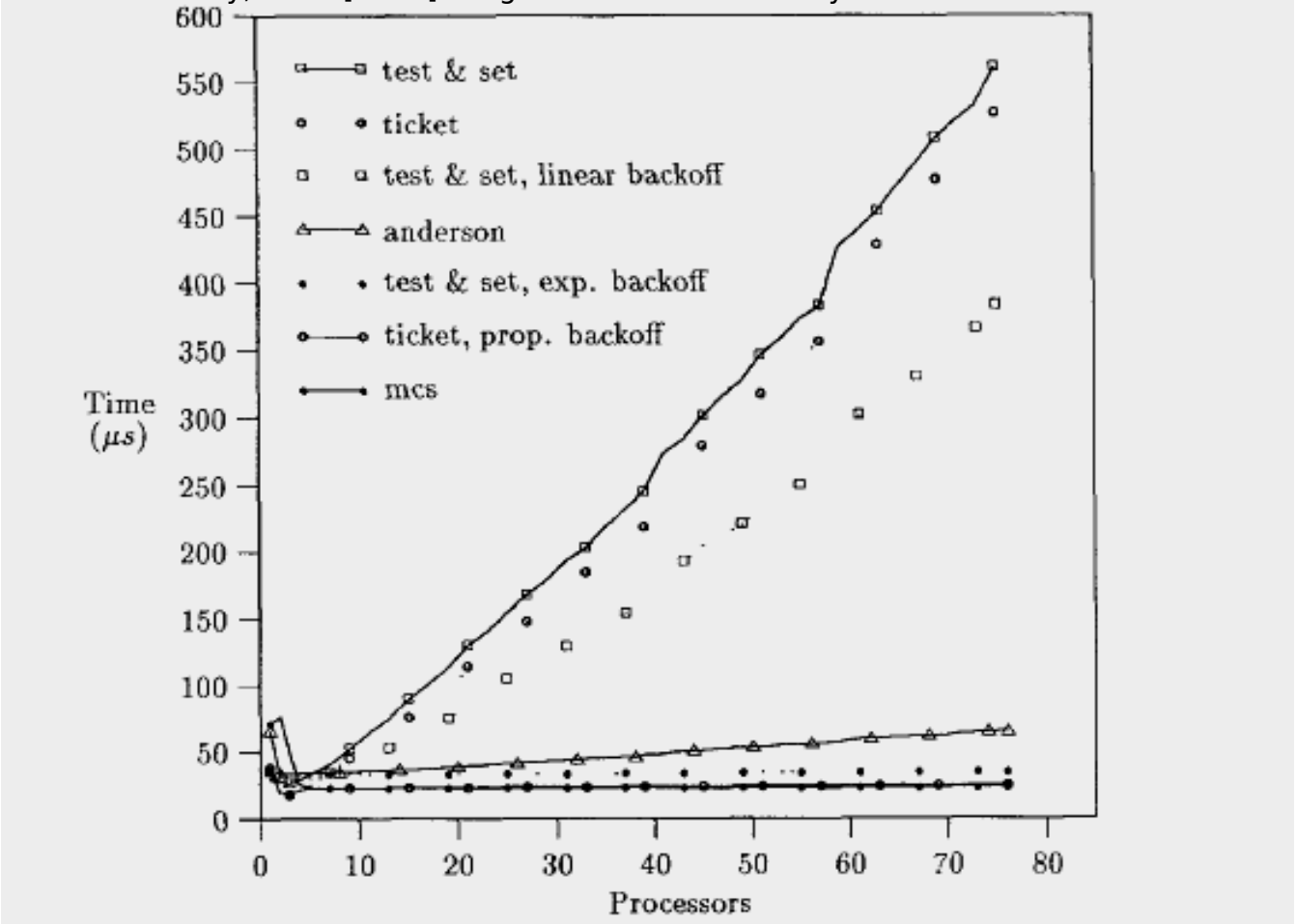
MCS Locks

- Fair, local spinning
  - atomic compare exchange:
    cmpxchg (T == Expected, Desired)

```
lock(Node * & T, Node * I) {
  I->next = null;
  I->Lock = false;
  Node * prev = swap(T, I);
   if (prev) {
      prev->next = I;
      do {} while (I->Lock == false);
   }
}

unlock (Node * & T, Node * I) {
  if (!I->next) {
     if (cmpxchg (T == I, 0)) return;  // no waiting cpu
     do { } while (!I->next);          // spin until the following process
                                          updates the next pointer
  }
  I->next->Lock = true;
}
```

Performance

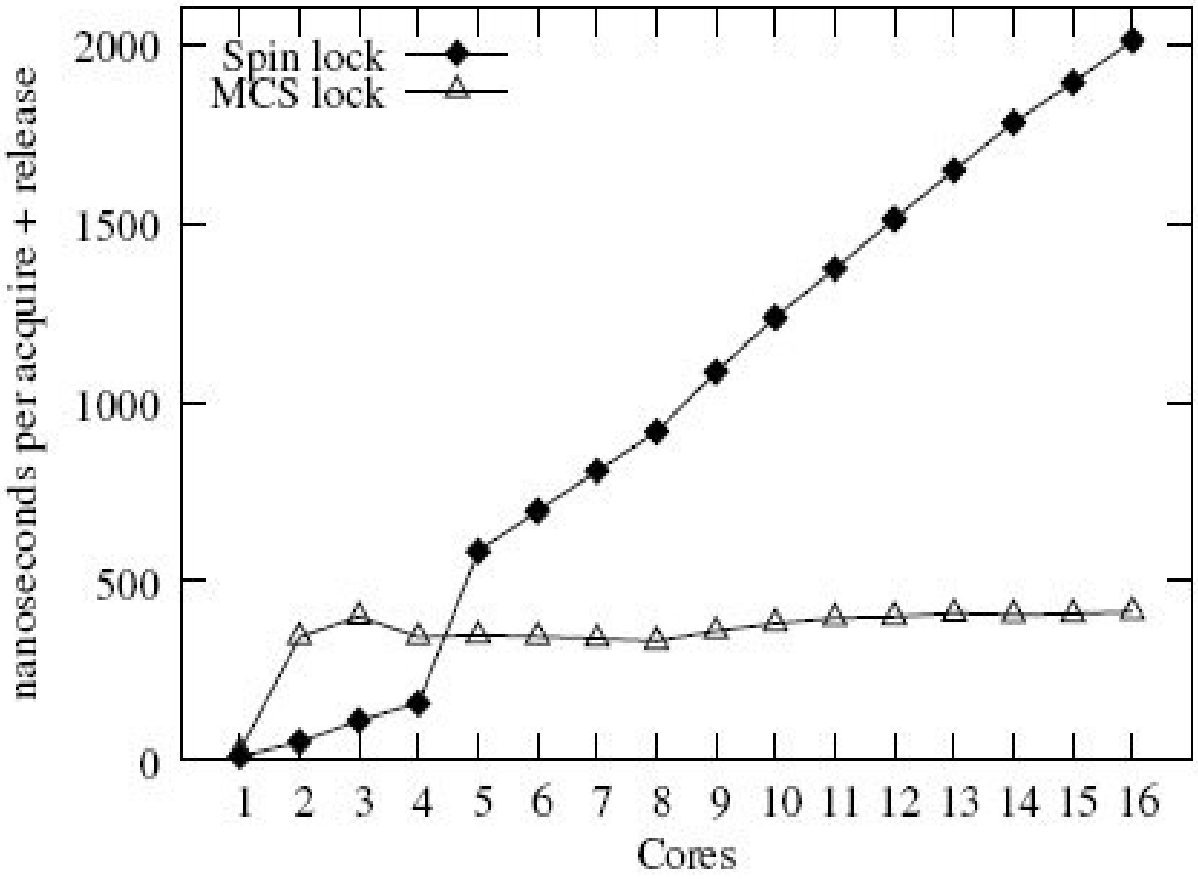
Source: Mellor Crummey, Scott [1990]: “Algorithms for Scalable Synchronization on Shared Memory Multiprocessors”



on BBN Butterfly: 256 nodes, local memory; each node can access other memory through log4(depth) switched network
Anderson: array-based queue lock

Performance

Source: [corey 08]



16 core AMD Opteron

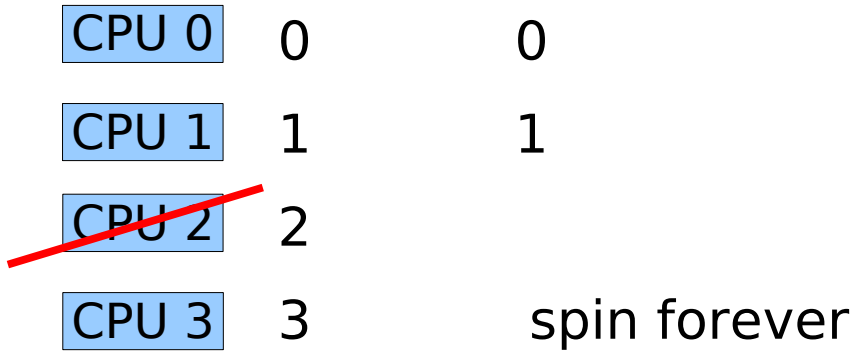
Overview

- Introduction
- Hardware Primitives
- Synchronization with Locks (Part I)
  - Properties
  - Locks
    - Spin Lock (Test & Set Lock)
    - Test & Test & Set Lock
    - Ticket Locks
- Synchronization without Locks
- Synchronization with Locks (Part II)
    - MCS Locks
    - Performance
  - Special Issues
    - Timeouts
    - Reader Writer Locks
    - Lockholder Preemption
    - Monitor, Mwait

Special Issues

- No longer apply for lock
  - after timeout
  - to kill / signal competing thread

  - Spin Lock: (trivial: stop spinning)
  - Ticket Lock: my_ticket current

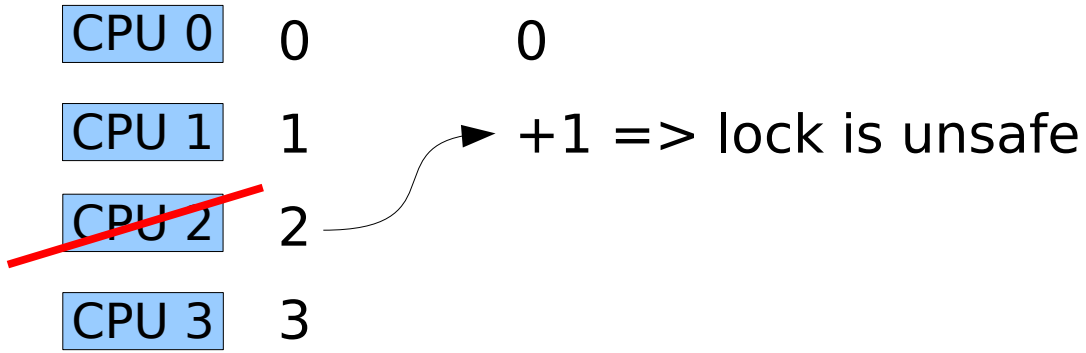


  - MCS Lock: (see Exercises)
    - dequeue nodes of competing threads

Special Issues

- No longer apply for lock
  - after timeout
  - to kill / signal competing thread

  - Spin Lock: (trivial: stop spinning)
  - Ticket Lock: my_ticket current

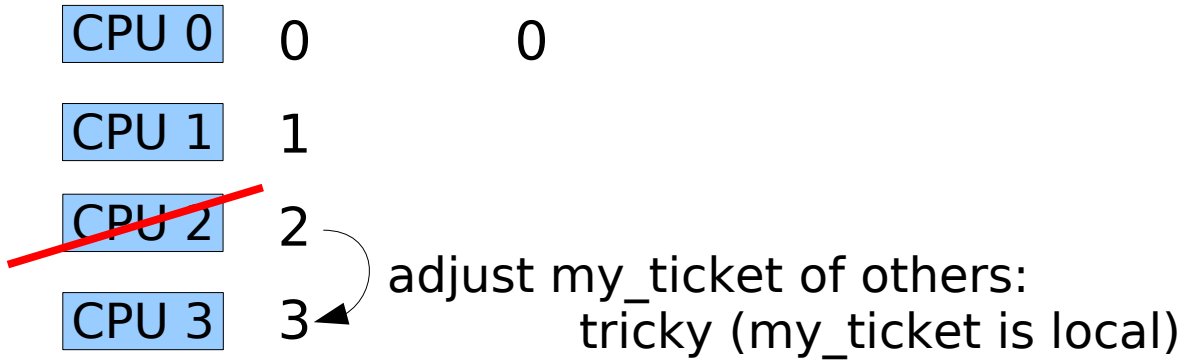


  - MCS Lock: (see Exercises)
    - dequeue nodes of competing threads

Special Issues

- No longer apply for lock
  - after timeout
  - to kill / signal competing thread

  - Spin Lock: (trivial: stop spinning)
  - Ticket Lock: my_ticket current



  - MCS Lock: (see Exercises)
    - dequeue nodes of competing threads

Special Issues

- Reader Writer Locks
  - Lock differentiates two types of lock holders:
    - Readers:
      - Don't modify the lock-protected object
      - Multiple readers may hold the lock at the same time
    - Writers:
      - Modify the protected object
      - Writers must hold the lock exclusively
  - Fairness
    - Improve reader latency by allowing readers to overtake writers (=> unfair lock)

Special Issues

- Fair Ticket Reader-Writer Lock
  - co-locate reader tickets and writer tickets

```
lock read (next, current) {
  my_ticket = xadd (next, 1);
  do {} while (current.write != my_ticket.write);
}

lock write (next, current) {
   my_ticket = xadd (next.write, 1);
   do {} while (current != my_ticket);
}


unlock_read () {
   xadd (current.read, 1);
}

unlock write () {
   current.write ++;
}
```

current	next	R0	R1	W2	R3
0 0	0 0	0 0			
	0 1		0 1		
	0 2			0 2	
	1 2				1 2

Special Issues

- Fair Ticket Reader-Writer Lock
  - combine read, write ticket in single word

Correctness of Lock:

1) no counter must overflow:
=>
max count value >=
max #threads that simultaneously attempt to acquire the lock

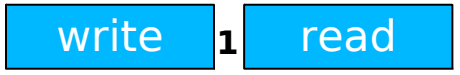
2) no overflow from read to write:

e.g., 8-bit counter: read = 0xff, write = 5

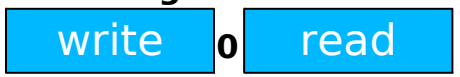
xadd(next, 1) => read = 0, write = 6

=> 1-bit to separate read from write field
always clear this bit before xadd

xadd => overflow

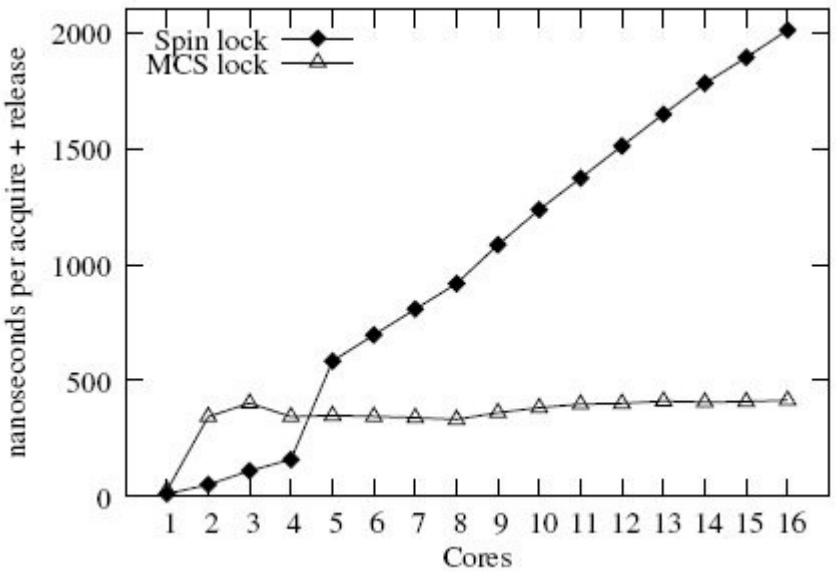


clear flag before next xadd

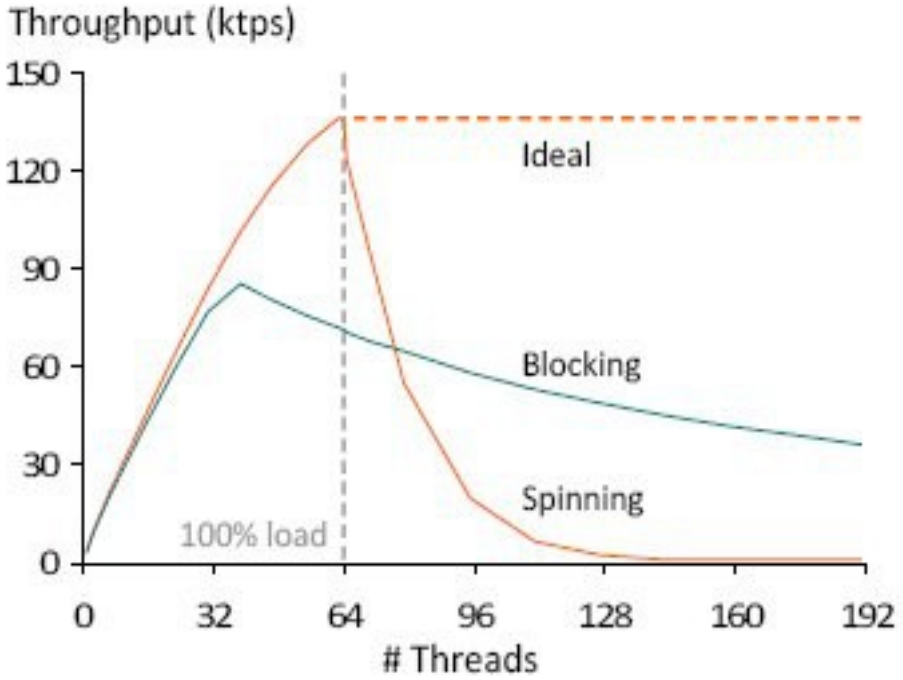


Read won't overflow again unless 2^n CPUs are preempted after clear flag (i.e. 2^n xadds in sequence)
=> Condition (1) prevents this

Special Issues



Source: [corey 08]



Source: [johnson 10]

Special Issues

- Lockholder preemption
  - Spinning-time of other CPUs increase by the time the lockholder is preempted
    - worse for ticket lock / MCS
      - grant free lock to preempted thread

    => do not preempt lock holders

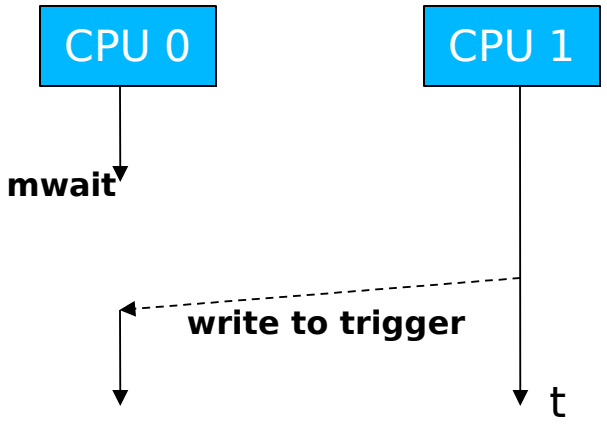
```
spin_lock(lock_var) {
  pushf; // store whether interrupts were already closed
  do {
     popf;
     reg = 1;
     do {} while (lock_var == 1);
     pushf;
     cli;
     swap(lock_var, reg);
  } while (reg == 1);
}
```

```
spin_unlock(lock_var) {
  lock_var = 0;
  popf;
}
```

Special Issues

- Monitor, Mwait
  - Stop CPU / HT while waiting for lock (signal)
    - Saves power
    - Frees up processor resources (HT)
  - Monitor: watch cacheline
  - Mwait: stop CPU / HT until:
    - cacheline has been written, or
    - interrupt occurs

```
while (trigger[0] != value) {
   monitor (&trigger[0])
   if (trigger[0] != value) {
      mwait
   }
}
```



References

- *Scheduler-Conscious Synchronization*
  LEONIDAS I. KONTOTHANASSIS, ROBERT W. WISNIEWSKI, MICHAEL L. SCOTT
- *Scalable Reader- Writer Synchronization for Shared-Memory Multiprocessors*
  John M. Mellor-Crummey, Michael L. Scottt
- *Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors*
  JOHN M. MELLOR-CRUMMEY, MICHAEL L.
- *Concurrent Update on Multiprogrammed Shared Memory Multiprocessors*
  Maged M. Michael, Michael L. Scott
- *Scalable Queue-Based Spin Locks with Timeout*
  Michael L. Scott and William N. Scherer III

References

- *Reactive Synchronization Algorithms for Multiprocessors*
  B. Lim, A. Agarwal

- *Lock Free Data Structures*
  John D. Valois (PhD Thesis)

- *Reduction: A Method for Proving Properties of Parallel Programs*
  R. Lipton - Communications of the ACM 1975

- *Decoupling Contention Management from Scheduling (ASPLOS 2010)*
  F.R. Johnson, R. Stoica, A. Ailamaki, T. Mowry

- *Corey: An Operating System for Many Cores (OSDI 2008)*
  Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, Zheng Zhang

MESI

MESI Cache Coherency Protocol