Distributed Operating Systems

Synchronization in Parallel Systems

Marcus Völp

2009

Topics

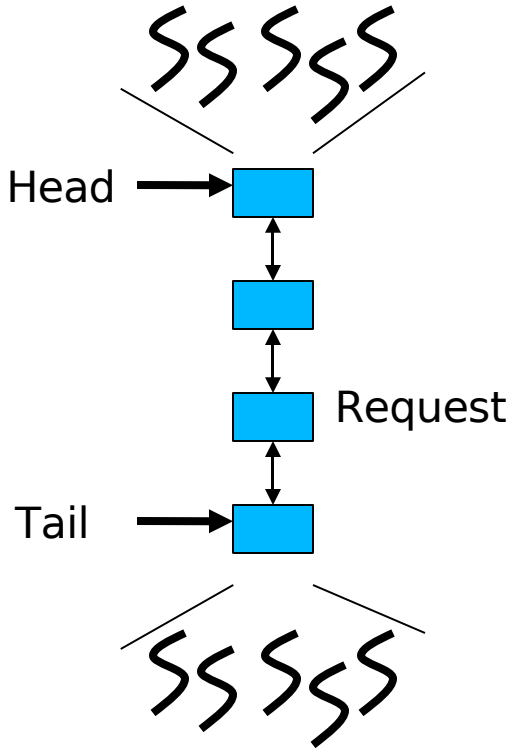
- Synchronization
- Locking
- Analysis / Comparison

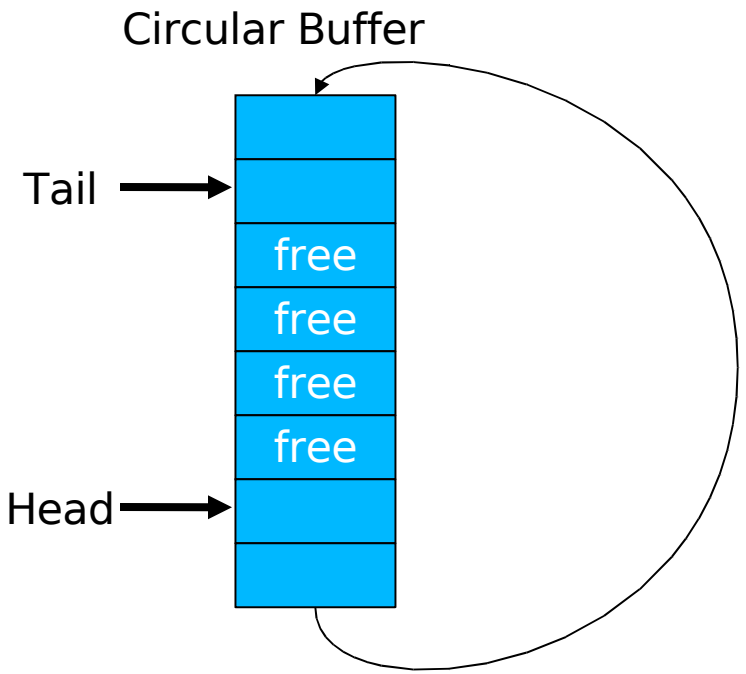
Overview

- Introduction
- Hardware Primitives
- Locking
  - Spin Lock (Test & Set Lock)
  - Test & Test & Set Lock
  - Ticket Locks
  - MCS Locks
- Lock-free Synchronization
- Special Issues
  - Timeouts
  - Reader Writer Locks
  - Lockholder Preemption
  - Monitor, Mwait
- Performance

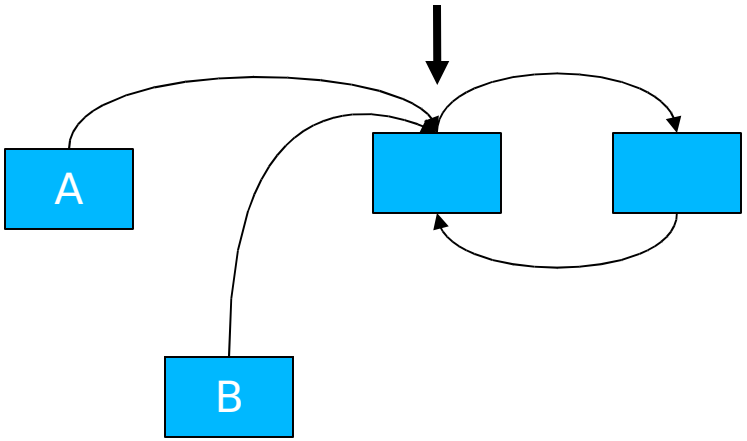
Introduction

- An example: Request Queue



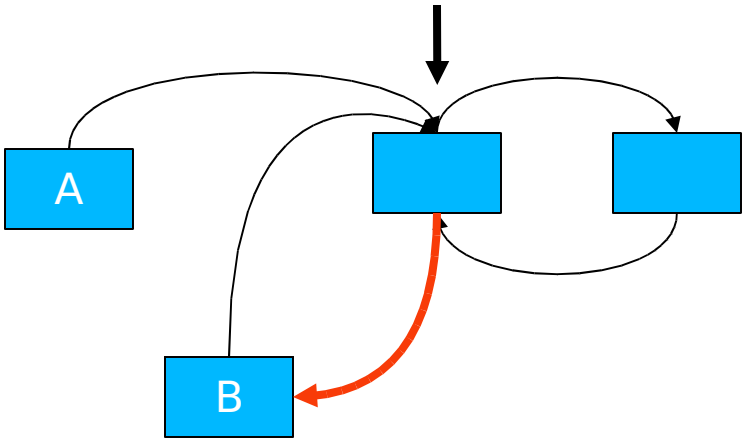


Introduction



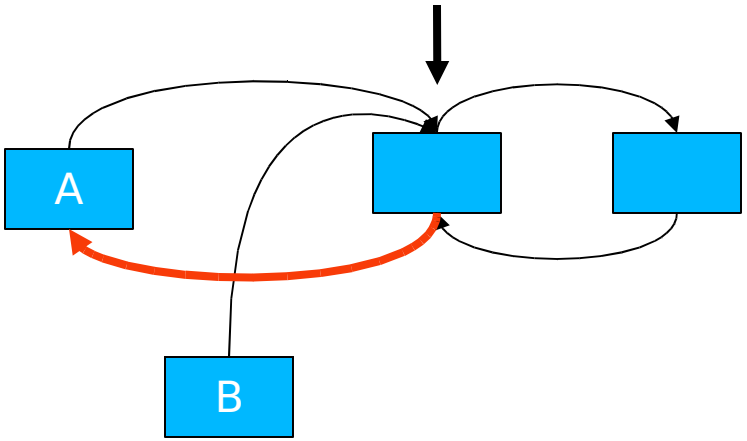
1) A,B create list elements
2) A,B set next pointer to head

Introduction



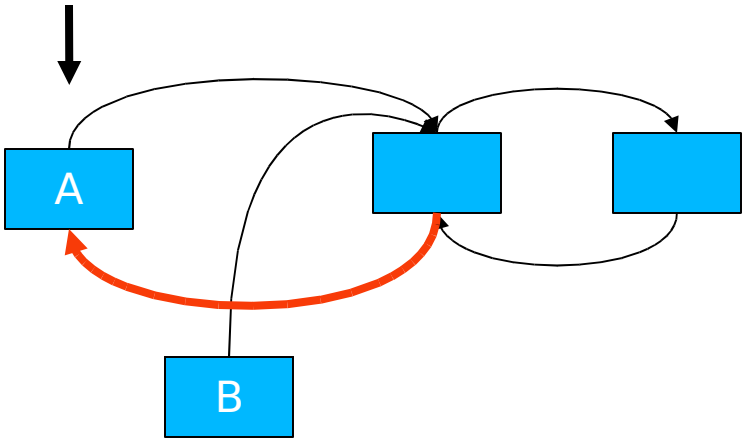
1) A,B create list elements
2) A,B set next pointer to head
3) B set prev pointer

Introduction



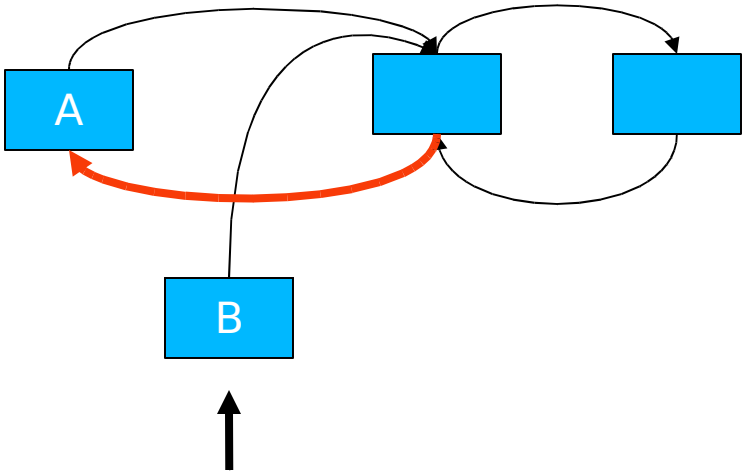
1) A,B create list elements
2) A,B set next pointer to head
3) B set prev pointer
4) A set prev pointer

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

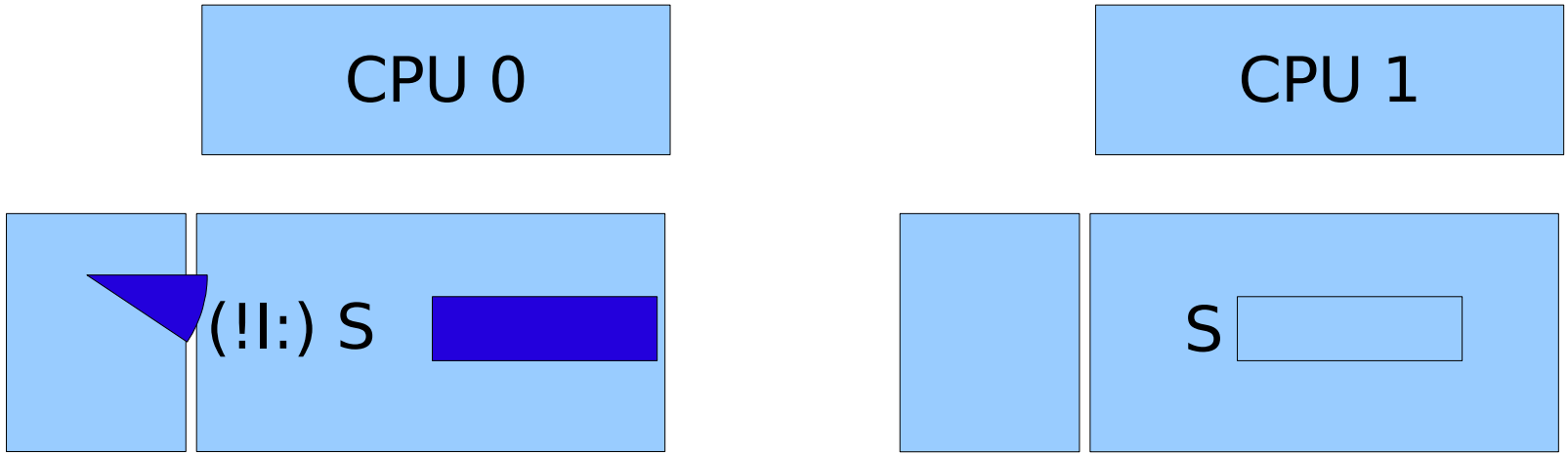
- First Solution
  - Locks
    - coarse grain: lock entire list
    - fine grain: lock list elements

```
lock_list;
  insert_element;
unlock_list;

lock head
if (try_lock head->next) {
  insert_element
  unlock head->next
}
unlock head
retry if trylock failed
```

Hardware Primitives

- How to make instructions atomic
  - Bus lock
    - Lock memory bus for duration of single instruction (e.g., lock add)
  - **Cache Lock**
    - x86 > Pentium 4: processor may not grab the bus lock but execute atomic operation on single cacheline; subsequent accesses to this cacheline are delayed
  - Observe Cache (ARM v6, Alpha, x86: monitor, mwait)
    - Load Linked: Load value and watch location
    - Store Conditional: Store value if no other store has accessed location
- Atomic operations
  - loads, stores
  - swap (XCHG)
  - bit test and set (BTS)
    - **if** bit clear **then** set bit; return true **else** return false
  - compare and swap (CAS m, old, new)
    - **if** m == old **then** m := new ; return true **else** new := m, return false

Load Linked – Store Conditional

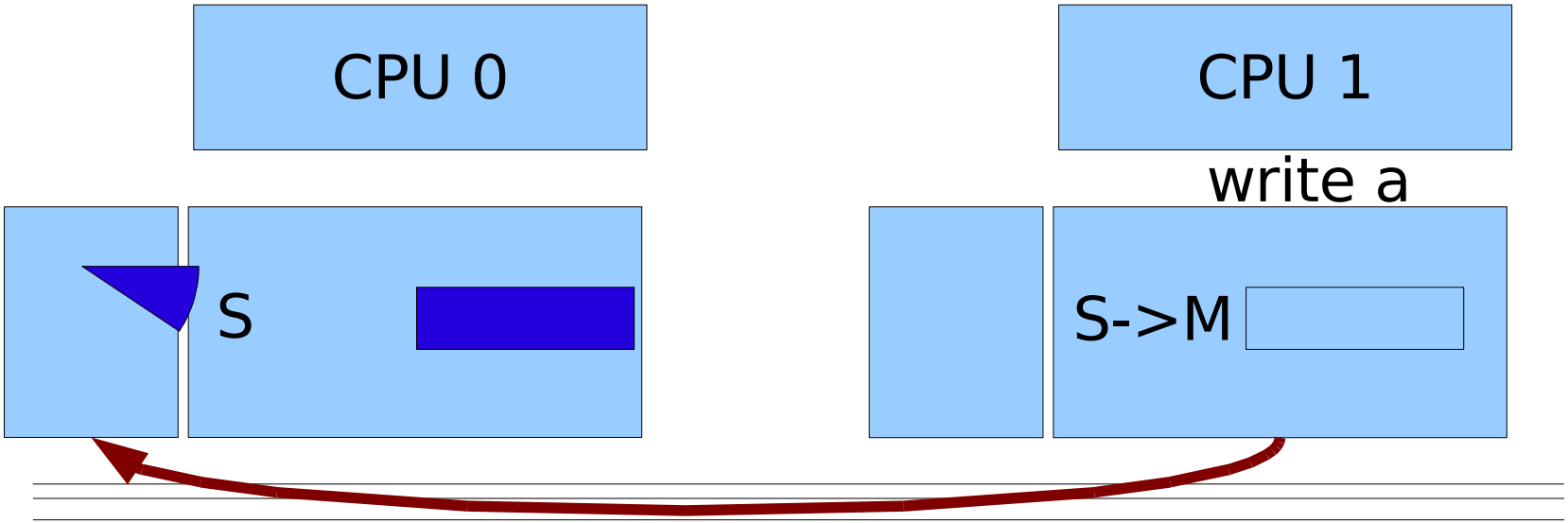


load linked a → reg

modify reg

store conditional reg → a

Load Linked – Store Conditional



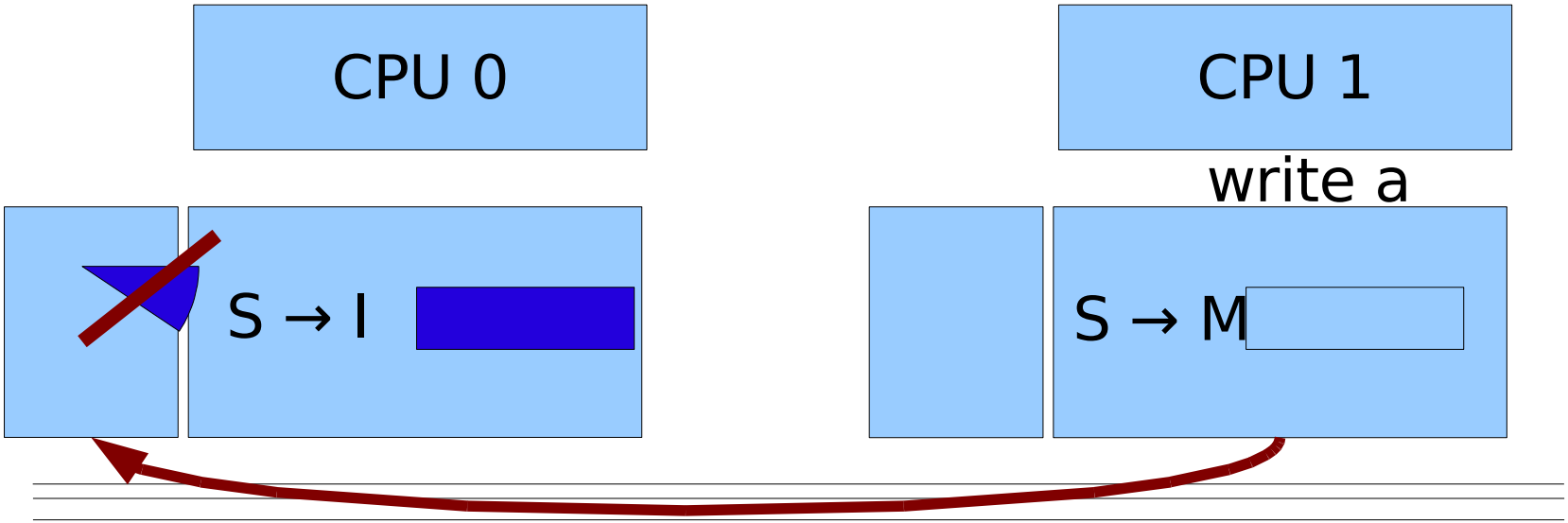
load linked a → reg

modify reg

store a

store conditional reg → a

Load Linked – Store Conditional



load linked a → reg

modify reg

store conditional reg → a

store a

Mutual Exclusion without Locks

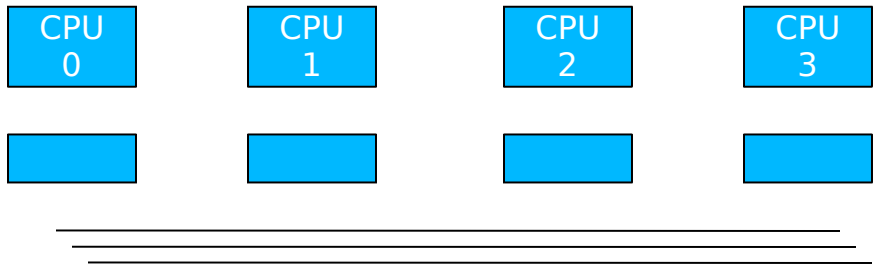
- Last week: Decker / Peterson
  - **only** atomic stores, atomic loads
  - if memory is sequentially consistent
    - (memory fences otherwise)

```
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;
}
```

Locking Algorithms

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

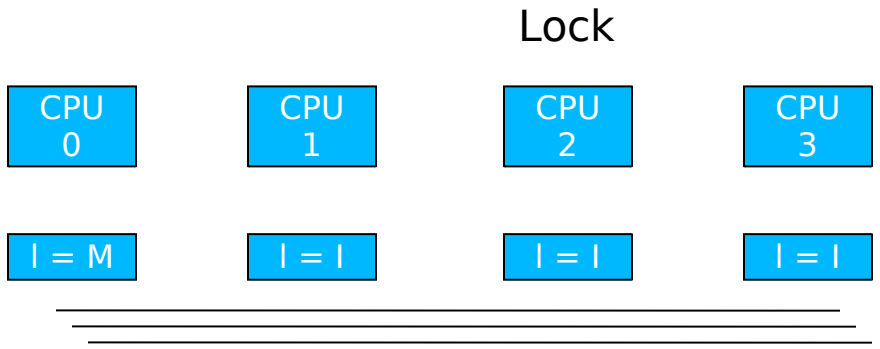
```
lock (lock_var & l) {
  do {
     reg = 1;
     swap (l, reg)
  } while (reg == 1);
}

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



Locking Algorithms

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

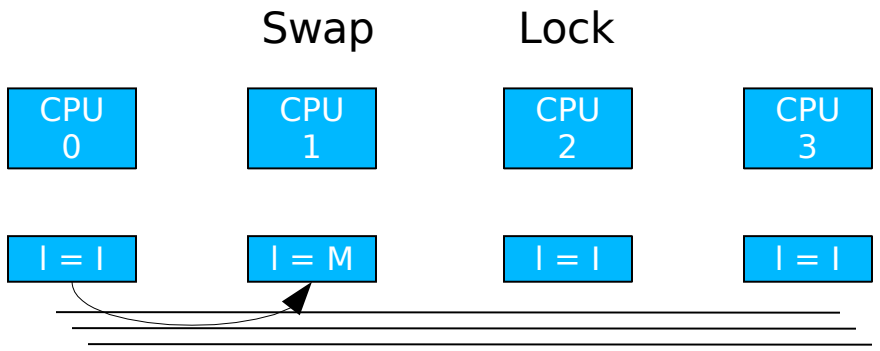
```
lock (lock_var & l) {
  do {
     reg = 1;
     swap (l, reg)
  } while (reg == 1);
}

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



Locking Algorithms

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

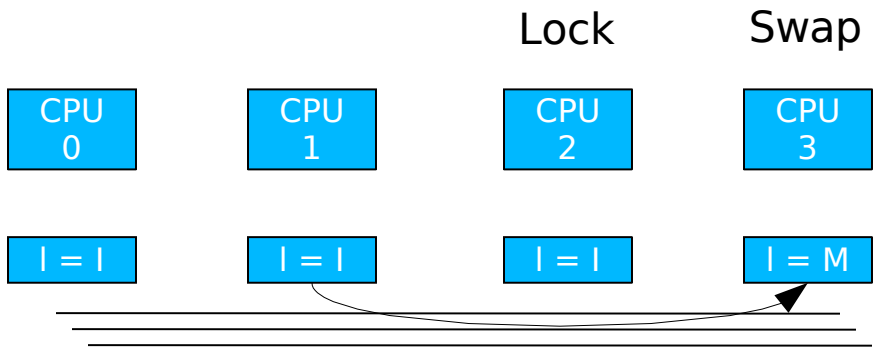
```
lock (lock_var & l) {
  do {
     reg = 1;
     swap (l, reg)
  } while (reg == 1);
}

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



Locking Algorithms

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

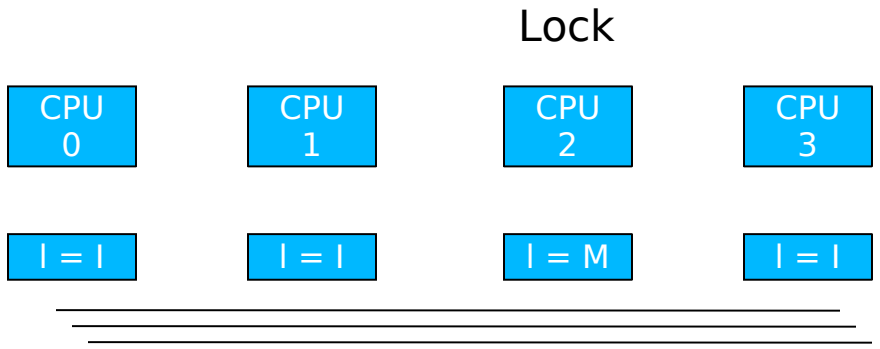
```
lock (lock_var & l) {
  do {
     reg = 1;
     swap (l, reg)
  } while (reg == 1);
}

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



Locking Algorithms

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

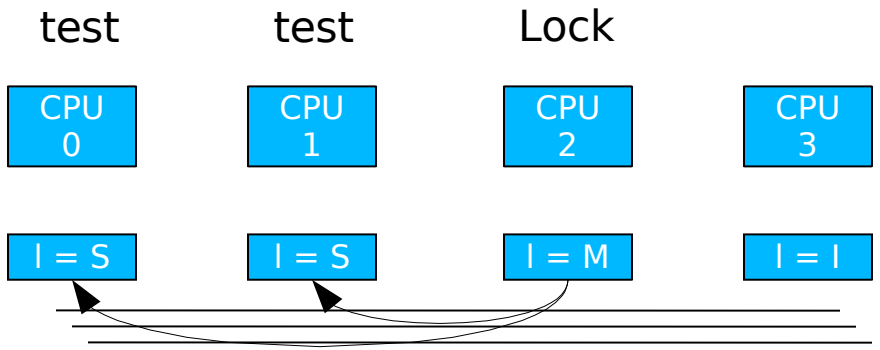
```
lock (lock_var & l) {
  do {
     reg = 1;
     do { } while (l == 1);
     swap (l, reg)
  } while (reg == 1);
}

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



Locking Algorithms

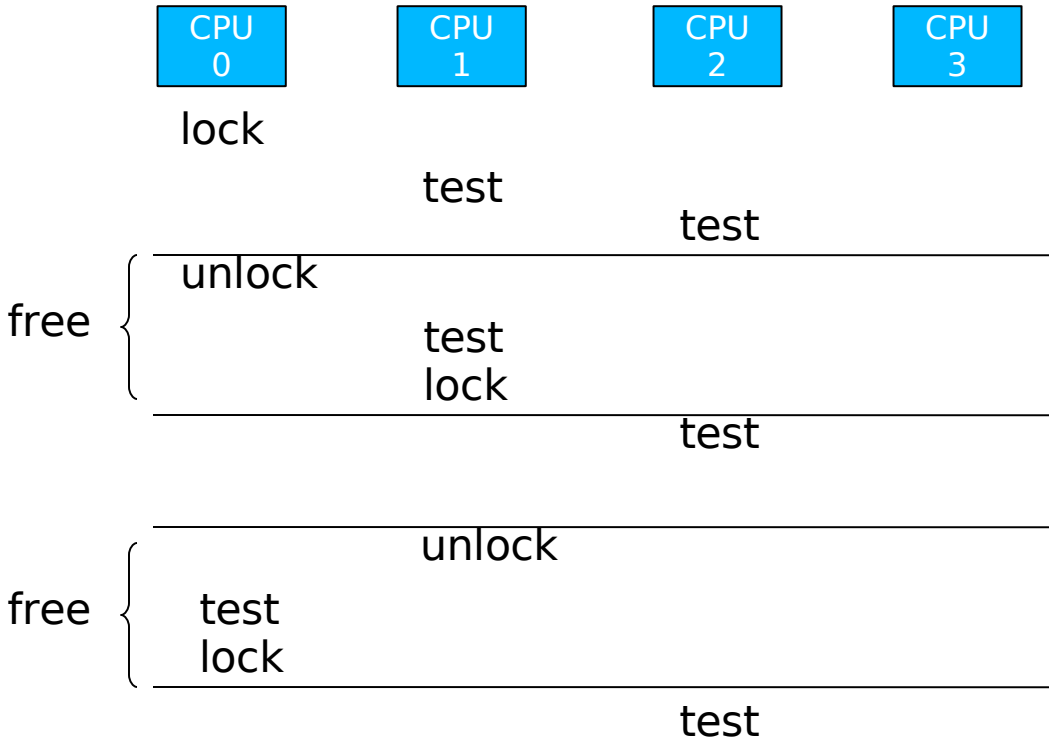
- Spin Lock (Test and Test and Set Lock)
  - atomic swap

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

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



Locking Algorithms

- Fairness



Locking Algorithms

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

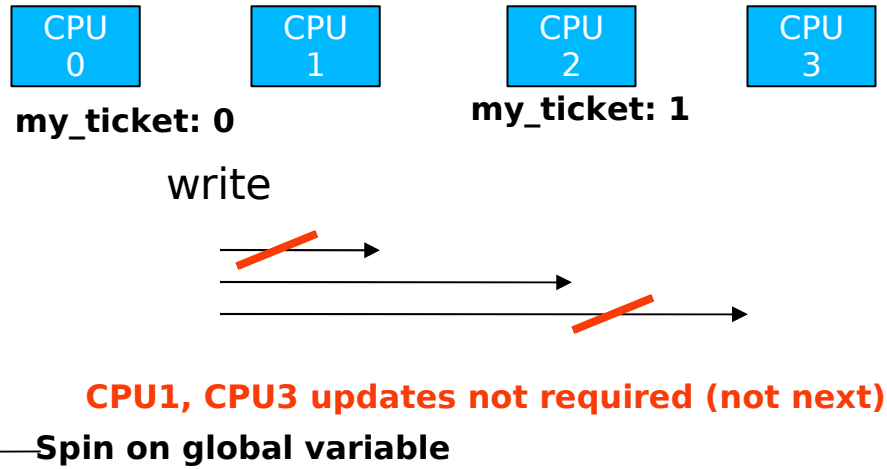
```
lock_struct {
  next_ticket,
  current_ticket
}

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

unlock (lock_var & 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

Locking Algorithms

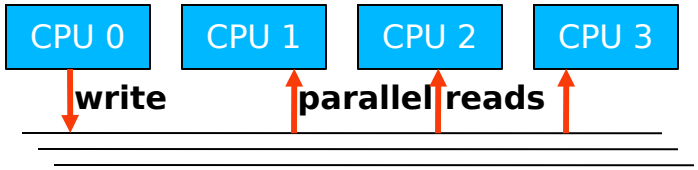
- Fairness: Ticket Lock
  - fetch and add (xadd)

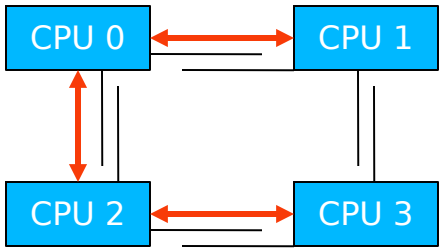
```
lock_struct {
  next_ticket,
  current_ticket
}

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

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



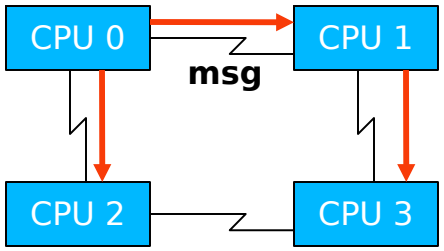
However:
- Signal all CPUs not only next
- Preemption of enlisted threads

Local Spinning





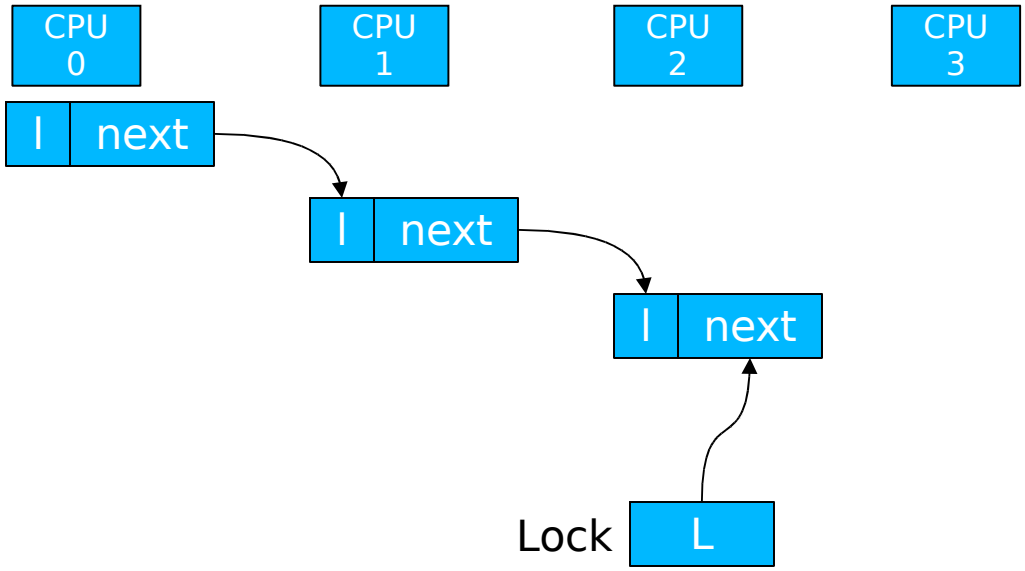
Need to propagate write on Bus 2-3 (or 1 – 3)



3 Network Messages

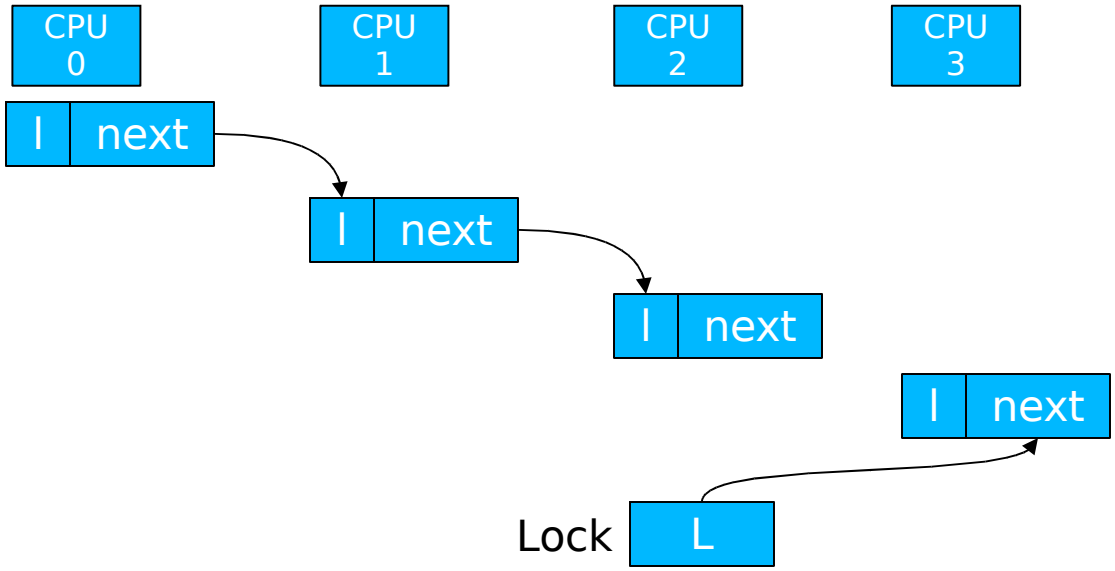
MCS-Lock by Mellor-Crummey and Scott

- Fair Lock with Local Spinning



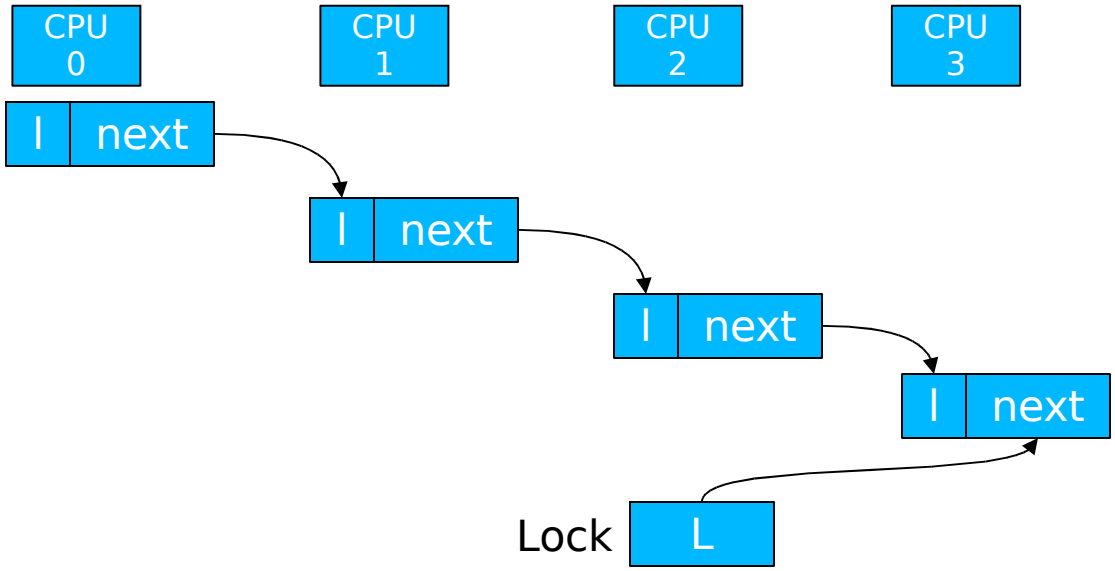
MCS-Lock (Mellor Crummey Scott)

- Fair Lock with Local Spinning



MCS-Lock (Mellor Crummey Scott)

- Fair Lock with Local Spinning

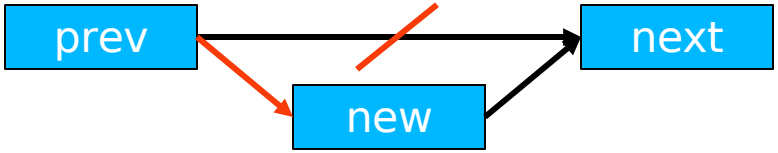


MCS Locks

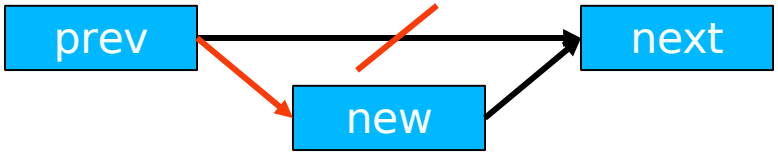
- Fair, local spinning
  - atomic compare exchange: cmpxchg (L == Old, New)

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

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

Lock-free synchronization




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




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

Lock-free Synchronization

- Load Linked, Store Conditional

```
insert (prev, new) {
  retry:
   ll (prev.next);
   new.next = prev.next;
   if (not stc (prev.next, new)) goto retry;
}
```

Overview

- Introduction
- Hardware Primitives
- Locking
  - Spin Lock (Test & Set Lock)
  - Test & Test & Set Lock
  - Ticket Locks
  - MCS Locks
- Lock-free Synchronization
- Special Issues
  - Timeouts
  - Reader Writer Locks
  - Lockholder Preemption
  - Monitor, Mwait
- Performance

Special Issues

- Timeouts
  - No longer apply for lock after timeout
  - Dequeue from MCS queue
- Reader Writer Locks
  - Lock differentiates two types of lockers:
    - reader, writer
      - Multiple readers may hold the lock at the same time
      - A writer can hold the lock only exclusively
  - Fairness
  - Improve reader latency by allowing readers to overtake writers (unfair lock)

Special Issues

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

```
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

- Lockholder preemption
  - Spinning-time of other CPUs increase by the preemption time of lockholders
  - E.g., no packets can be sent when the OS network thread is preempted while it holds the lock of the xmit queue

    => do not preempt lock holders

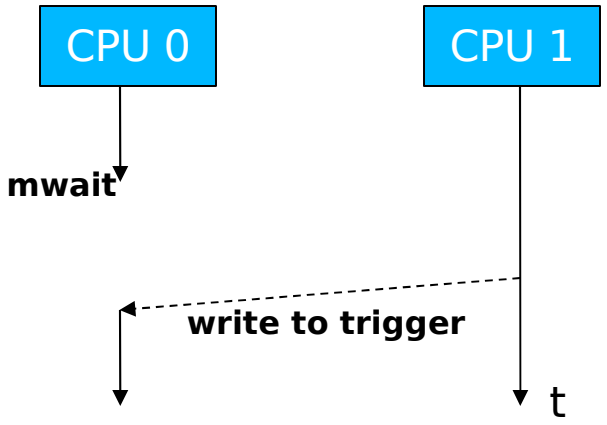
```
spin_lock(lock_var) {
  pushf; // store whether interrupts were already closed
  do {
     popf;
     reg = 1;
     do {} while (lock_var == 1);
     pushf;
     cli;
     swap(lock_var, reg);
  } while (reg == 1);
}
```

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

Special Issues

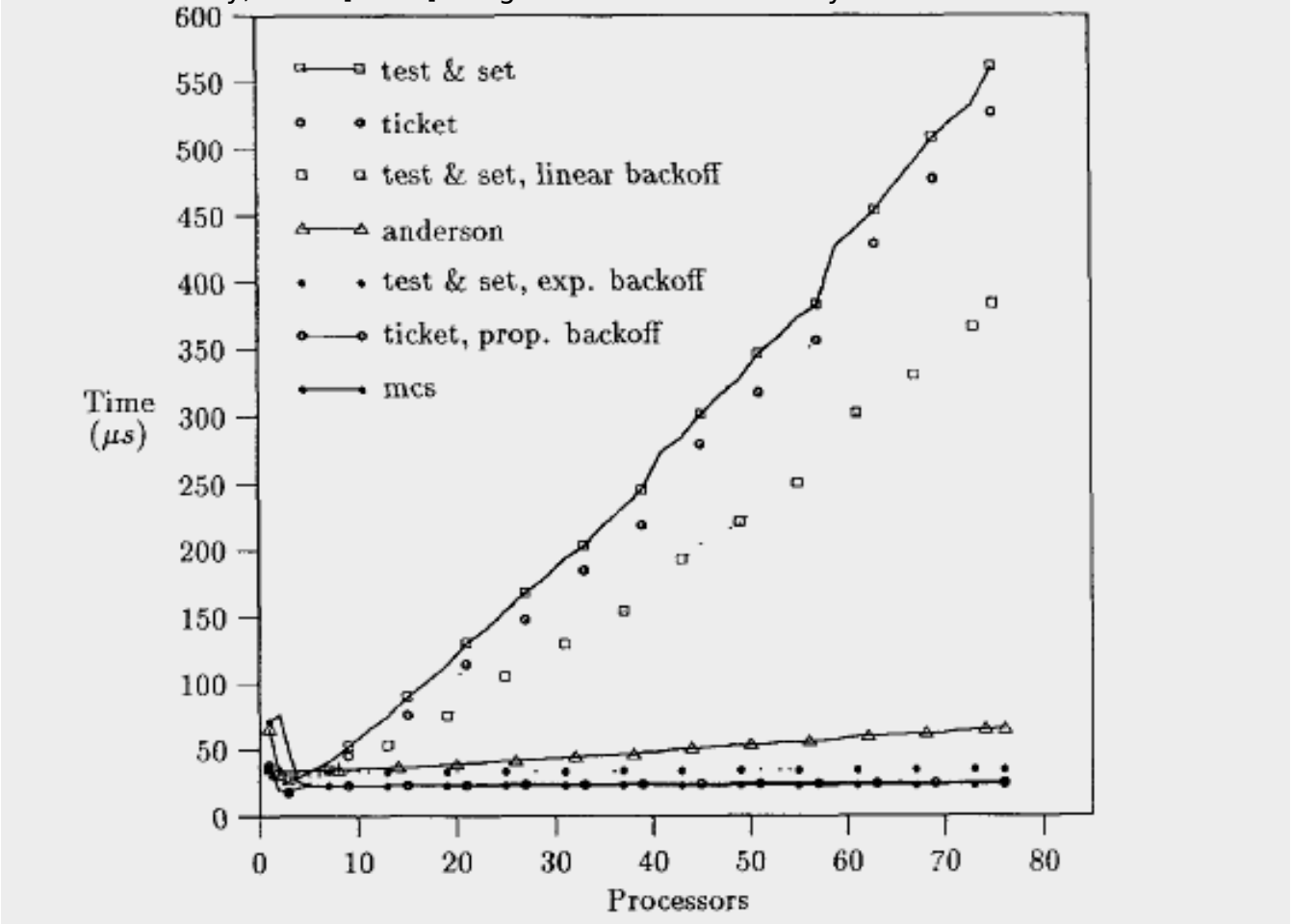
- Monitor, Mwait
  - Stop CPU / HT while waiting for lock (signal)
    - Saves power
    - Frees up processor resources (HT)
  - Monitor: watch cacheline
  - Mwait: stop CPU / HT until:
    - cacheline has been written, or
    - interrupt occurs

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



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

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
- Reactive Synchronization Algorithms for Multiprocessors
  B. Lim, A. Agarwal
- Lock Free Data Structures
  John D. Valois (PhD Thesis)