



TECHNISCHE
UNIVERSITÄT
DRESDEN

Faculty of Computer Science Institute for System Architecture, Operating Systems Group

Distributed Operating Systems

Synchronization

Marcus Völp

SS2014

- How to update objects consistently?
 - that may require multiple writes for the update
 - that may be larger than one cacheline

- How to build locks?
- How to make the implementations scale?
 - Cache effects
 - Reader/Writer Locks

- Lock-free synchronization
- Transactional memory

[Lipton 95] a, b are atomic if $A \parallel B = A;B$ or $B;A$

Read-Modify-Write Instructions are typically not atomic:

A		B	
add &x, 1		mov &x, 2	(initially: x = 0)

is typically executes as:

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

How to make instructions atomic

Bus lock

Lock memory bus until all memory accesses of a 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)

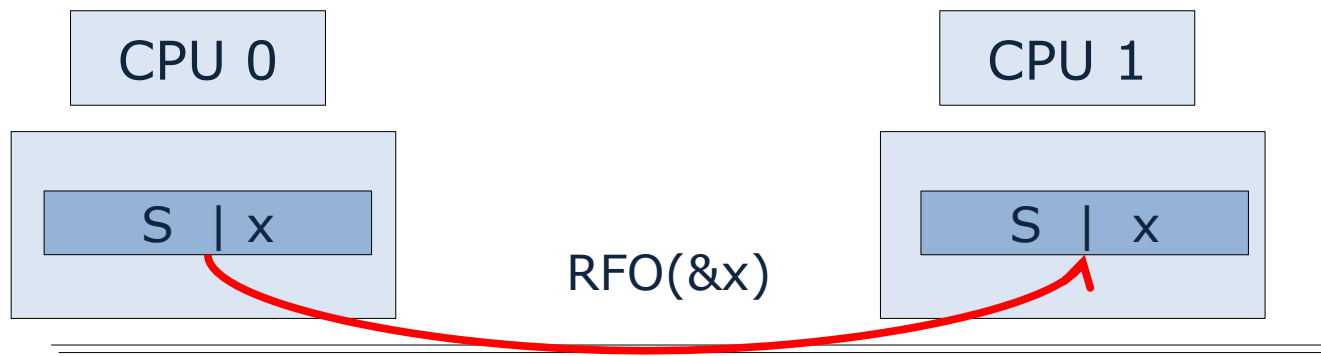
How to make instructions atomic

Cache Lock

last lecture: M(O)ESI Cache Coherence Protocol

add &x, 1

1. read_for_ownership(&x)
2. load &x → R
3. add R += 1
4. store R → &x
5. continue responding to snoop traffic



How to make instructions atomic

Cache Lock

last lecture: M(O)ESI Cache Coherence Protocol

add &x, 1

1. read_for_ownership(&x)
2. load &x → R
3. add R += 1
4. store R → &x
5. continue responding to snoop traffic



How to make instructions atomic

Cache Lock

last lecture: M(O)ESI Cache Coherence Protocol

add &x, 1

1. read_for_ownership(&x)
2. load &x → R
3. add R += 1
4. store R → &x
5. continue responding to snoop traffic



How to make instructions atomic

Cache Lock

last lecture: M(O)ESI Cache Coherence Protocol

add &x, 1

1. read_for_ownership(&x)
2. load &x → R
3. add R += 1
4. store R → &x
5. continue responding to snoop traffic



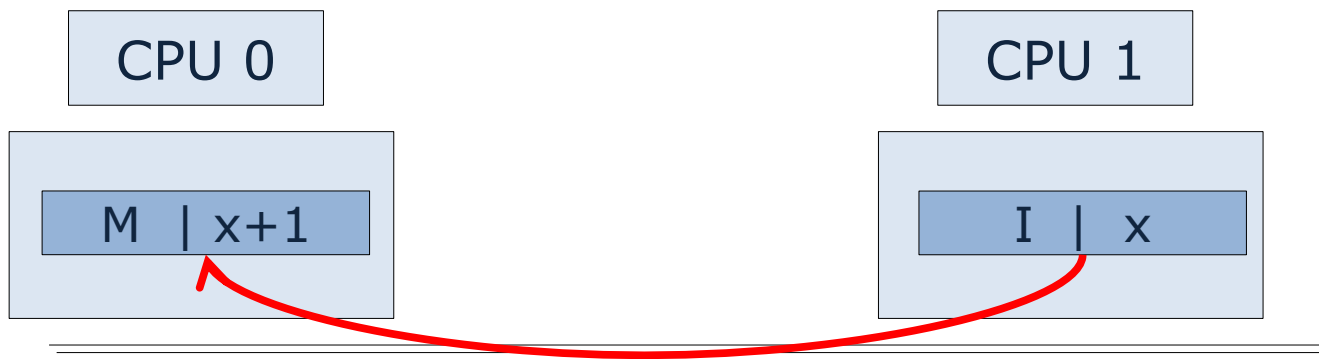
How to make instructions atomic

Cache Lock

last lecture: M(O)ESI Cache Coherence Protocol

add &x, 1

1. read_for_ownership(&x)
2. load &x → R
3. add R += 1
4. store R → &x
5. continue responding to snoop traffic



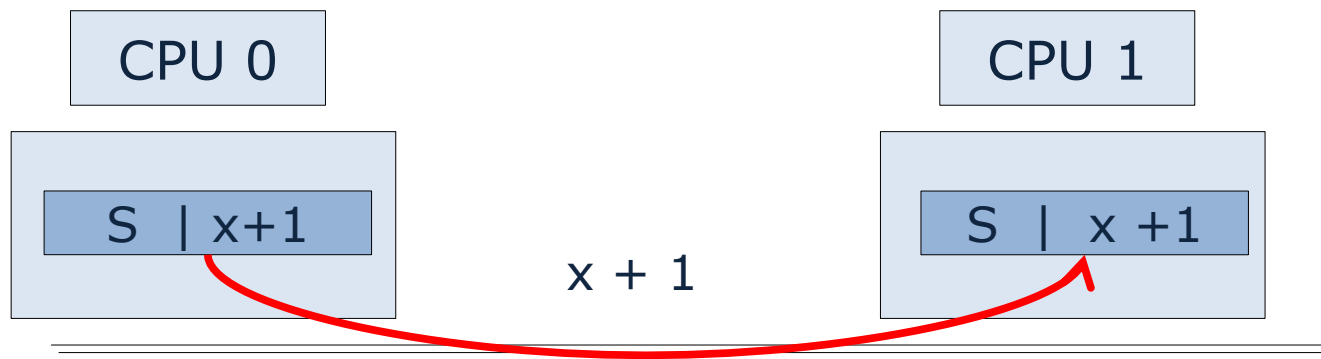
How to make instructions atomic

Cache Lock

last lecture: M(O)ESI Cache Coherence Protocol

add &x, 1

1. read_for_ownership(&x)
2. load &x \rightarrow R
3. add R += 1
4. store R \rightarrow &x
5. continue responding to snoop traffic



How to make instructions atomic

Observe Cache

- install cache watchdog on load
 - abort store if watchdog has detected a concurrent access
 - retry operation
- (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 (later)

How to make instructions atomic

Observe Cache

load linked &x, R

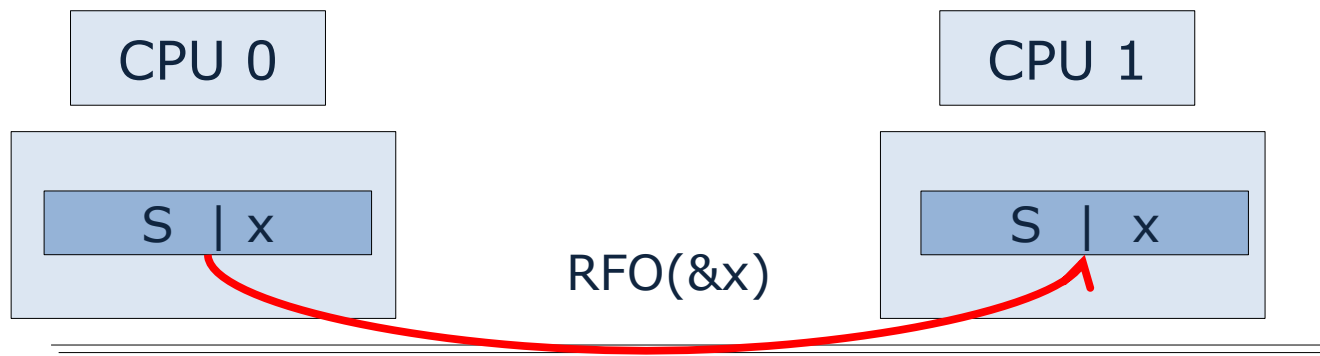
1. read_for_ownership(&x)

2. load &x → R

add R += 1

store conditional R, &x

3. try to store R → &x



How to make instructions atomic

Observe Cache

load linked &x, R

1. read_for_ownership(&x)

2. load &x → R

add R += 1

store conditional R, &x

3. try to store R → &x



How to make instructions atomic

Observe Cache

load linked &x, R

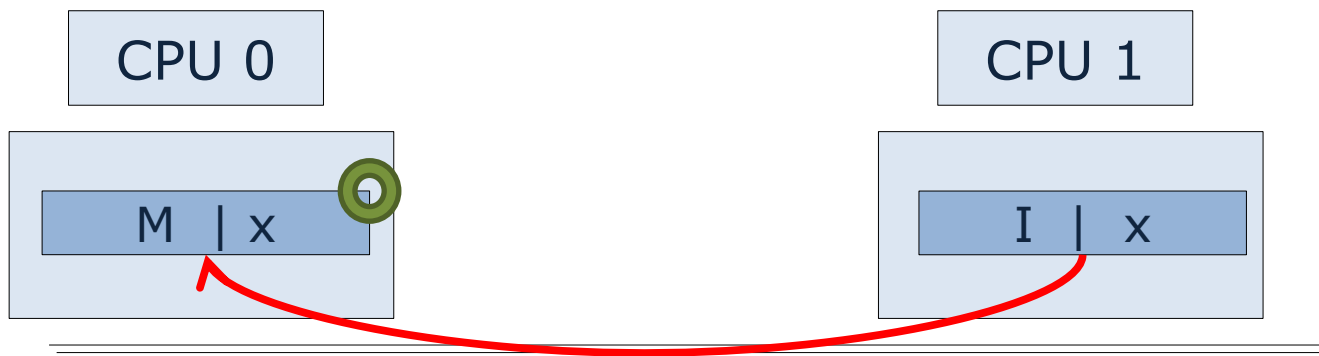
1. read_for_ownership(&x)

2. load &x → R

add R += 1

store conditional R, &x

3. try to store R → &x



How to make instructions atomic

Observe Cache

load linked &x, R

1. read_for_ownership(&x)

2. load &x → R

add R += 1

store conditional R, &x

3. try to store R → &x



How to make instructions atomic

Observe Cache

load linked &x, R

1. read_for_ownership(&x)

2. load &x → R

add R += 1

store conditional R, &x

3. try to store R → &x



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
```

Synchronization with Atomic Reads and Writes: Dekker's Algorithm

```
bool flag0 = false;           // intention to enter
bool flag1 = false;
int turn = 0;                 // who's next?
```

CPU0

```
P: flag0 = true;
while (flag1) {
    If (turn == 1) {
        flag0 = false;
        goto P;
    }
}
// Critical section
flag0 = false;
turn = 1;
```

CPU1

```
P: flag1 = true;
while (flag0) {
    If (turn == 0) {
        flag1 = false;
        goto P;
    }
}
// Critical section
flag1 = false;
turn = 0;
```

Dekker's Algorithm on z Series



z Series: later reads can bypass earlier writes unless both are to the same memory location

```
bool flag0 = false;            // intention to enter
bool flag1 = false;
int turn = 0;                 // who's next?
```

CPU0

CPU1

```
P: flag0 = true;
while (flag1) {
    If (turn == 1) {
        flag0 = false;
        goto P;
    }
}
// Critical section
flag0 = false;
turn = 1;
```

```
P: flag1 = true;  Buffered
while (flag0) { 
    If (turn == 0) {
        flag1 = false;
        goto P;
    }
}
// Critical section
flag1 = false;
turn = 0;
```

Properties to achieve

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 de-scheduled lockholder

priority inversion

! Not covered in this lecture (RTS / MKK)

spinning vs. blocking

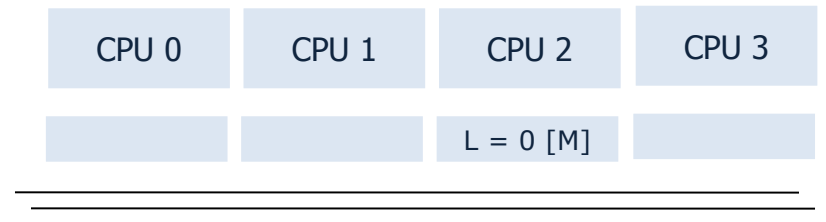
release CPU while others hold the lock

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;  
}
```



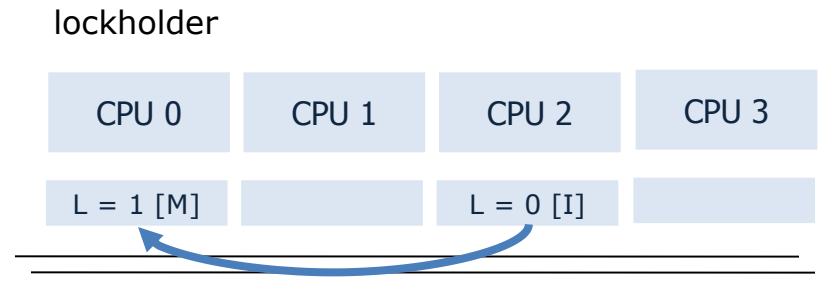
- Pro: 1 cheap atomic OP to acquire the lock
Cons: high bus traffic while lock is held

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;  
}
```



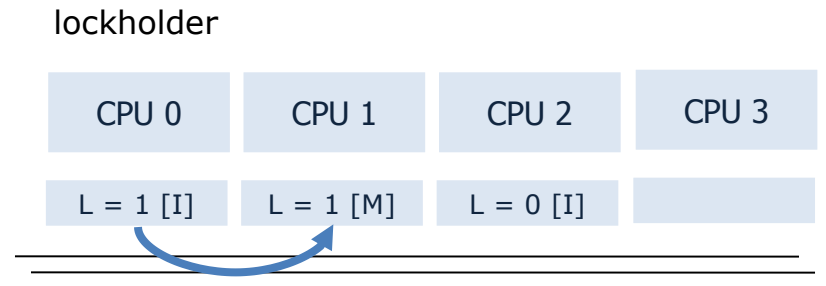
- Pro: 1 cheap atomic OP to acquire the lock
Cons: high bus traffic while lock is held

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;  
}
```



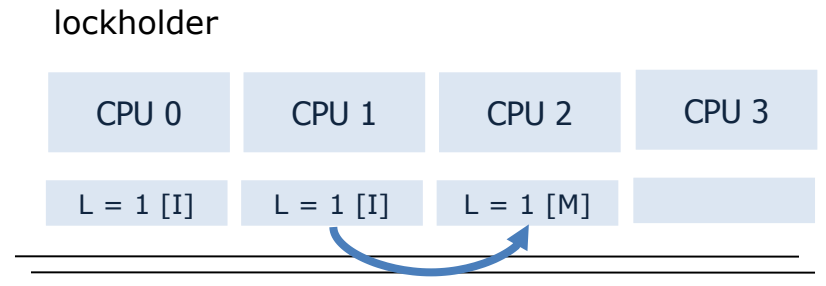
Pro: 1 cheap atomic OP to acquire the lock
Cons: high bus traffic while lock is held

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;  
}
```



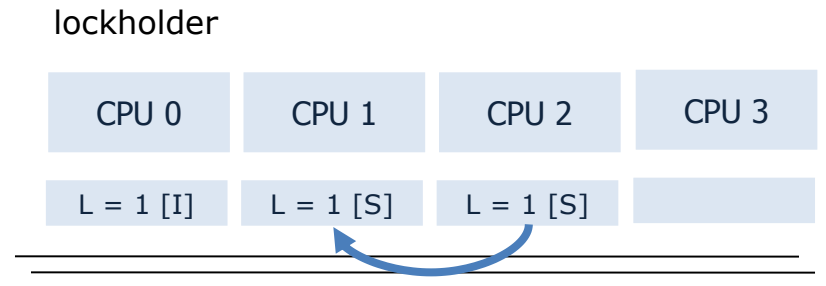
- Pro: 1 cheap atomic OP to acquire the lock
Cons: high bus traffic while lock is held

Spin Lock (Test and Test and Set Lock)

atomic swap

```
lock (lock_var & L) {  
  do {  
    reg = 1;  
    swap (L, reg)  
    if (reg == 0) break;  
    while (L == 1) {};  
  } while (true);  
}
```

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

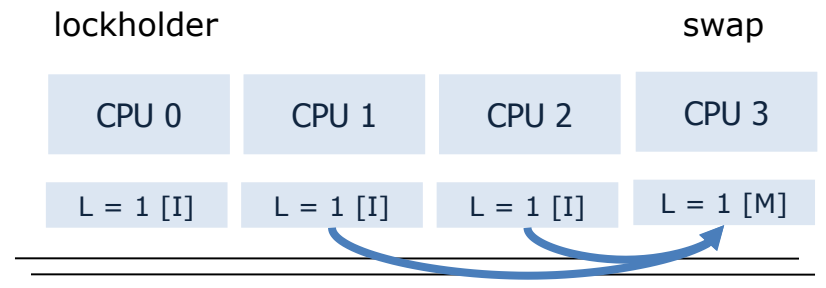


Spin Lock (Test and Test and Set Lock)

atomic swap

```
lock (lock_var & L) {  
  do {  
    reg = 1;  
    swap (L, reg)  
    if (reg == 0) break;  
    while (L == 1) {};  
  } while (true);  
}
```

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

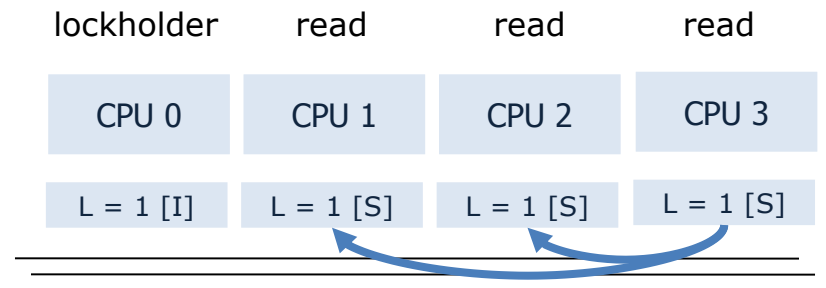


Spin Lock (Test and Test and Set Lock)

atomic swap

```
lock (lock_var & L) {
  do {
    reg = 1;
    swap (L, reg)
    if (reg == 0) break;
    while (L == 1) {};
  } while (true);
}
```

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

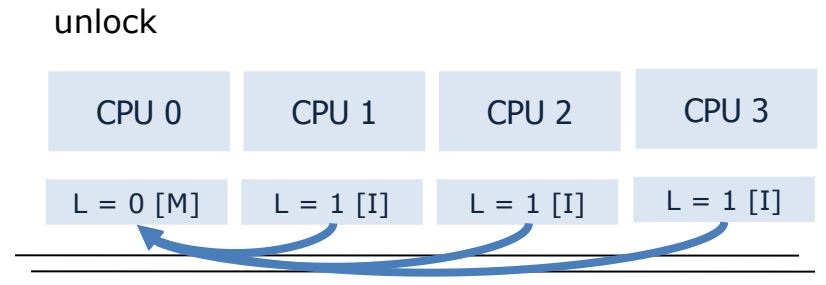


Spin Lock (Test and Test and Set Lock)

atomic swap

```
lock (lock_var & L) {  
  do {  
    reg = 1;  
    swap (L, reg)  
    if (reg == 0) break;  
    while (L == 1) {};  
  } while (true);  
}
```

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

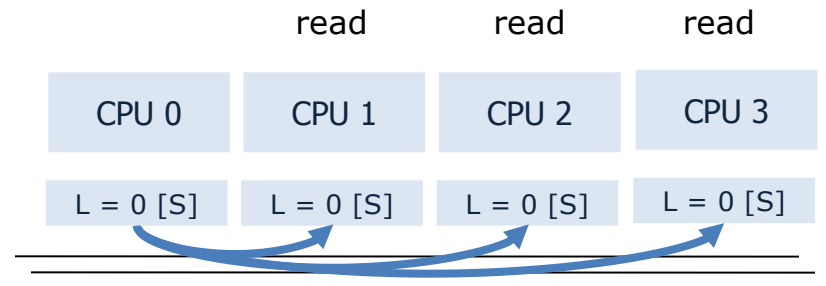


Spin Lock (Test and Test and Set Lock)

atomic swap

```
lock (lock_var & L) {
  do {
    reg = 1;
    swap (L, reg)
    if (reg == 0) break;
    while (L == 1) {};
  } while (true);
}
```

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

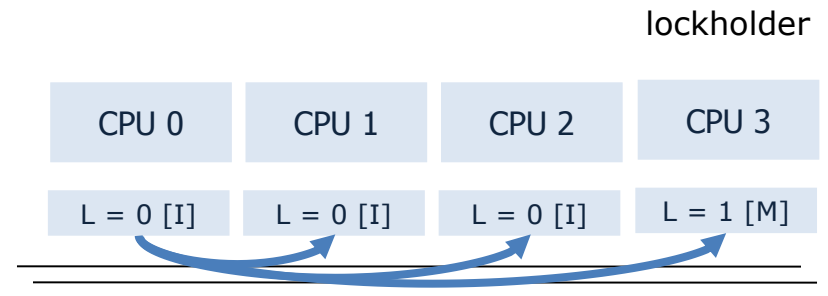


Spin Lock (Test and Test and Set Lock)

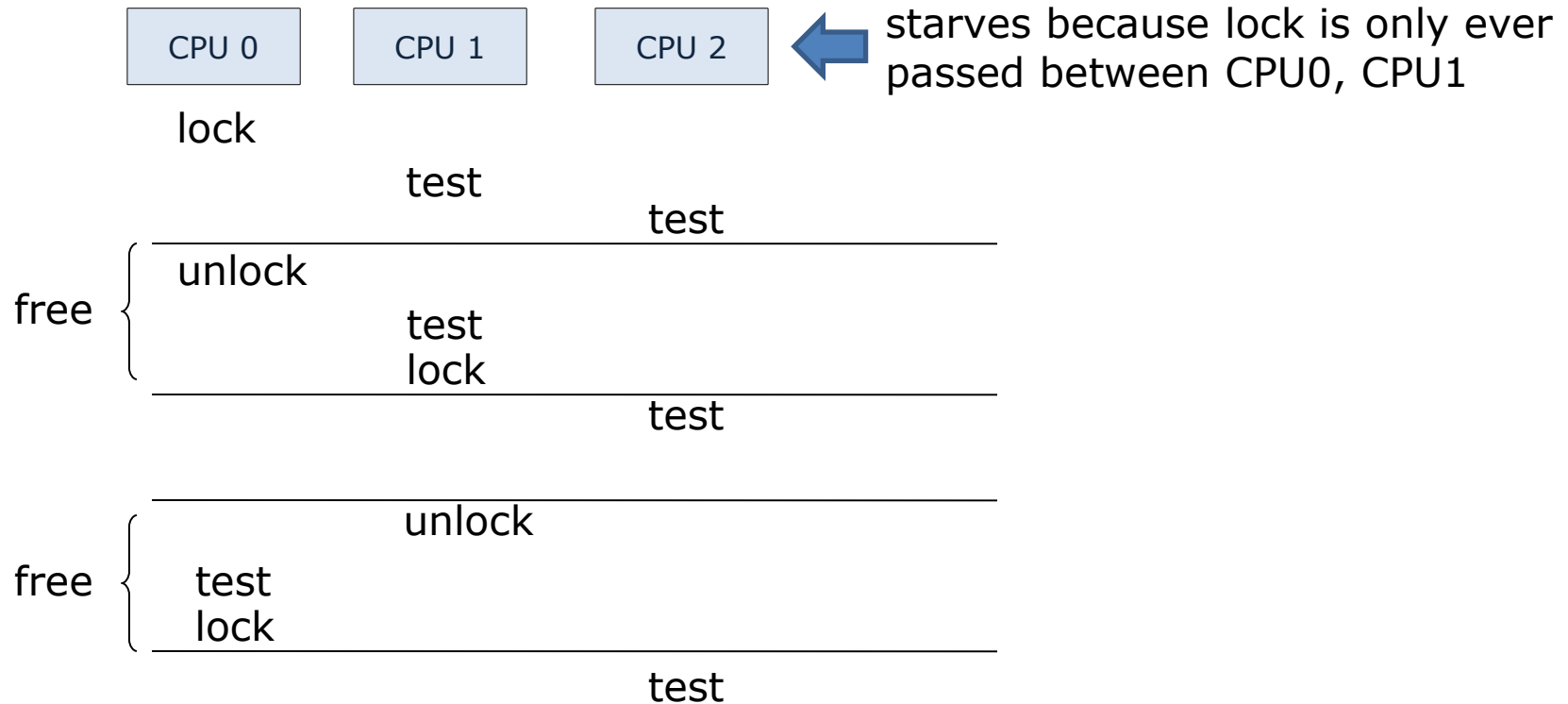
atomic swap

```
lock (lock_var & L) {
  do {
    reg = 1;
    swap (L, reg)
    if (reg == 0) break;
    while (L == 1) {};
  } while (true);
}
```

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



■ Fairness



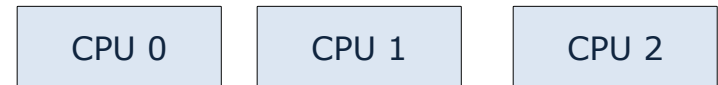
Fairness: Ticket Lock

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

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 ++;
}

```

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

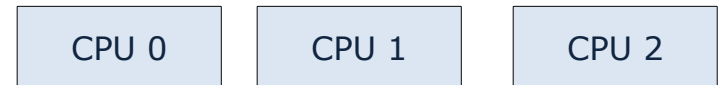
Fairness: Ticket Lock

fetch and add (xadd)

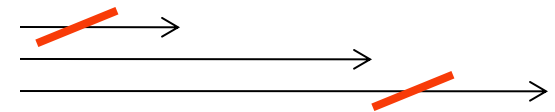
```
lock_struct {
    next_ticket,
    current_ticket
}
```

```
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 ++;
}
```



only lockholder writes current ticket



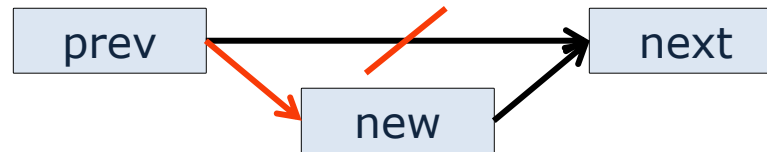
CPU1, CPU3 updates not required (not next)

← **Spin on global variable**

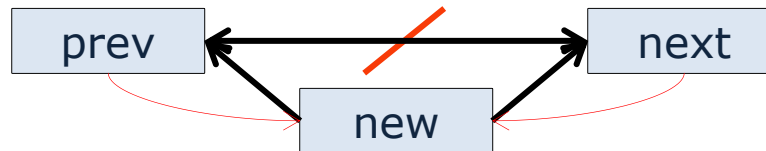
However:

- signal all CPUs not only next
- abort / timeout of competing threads

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;
}
```

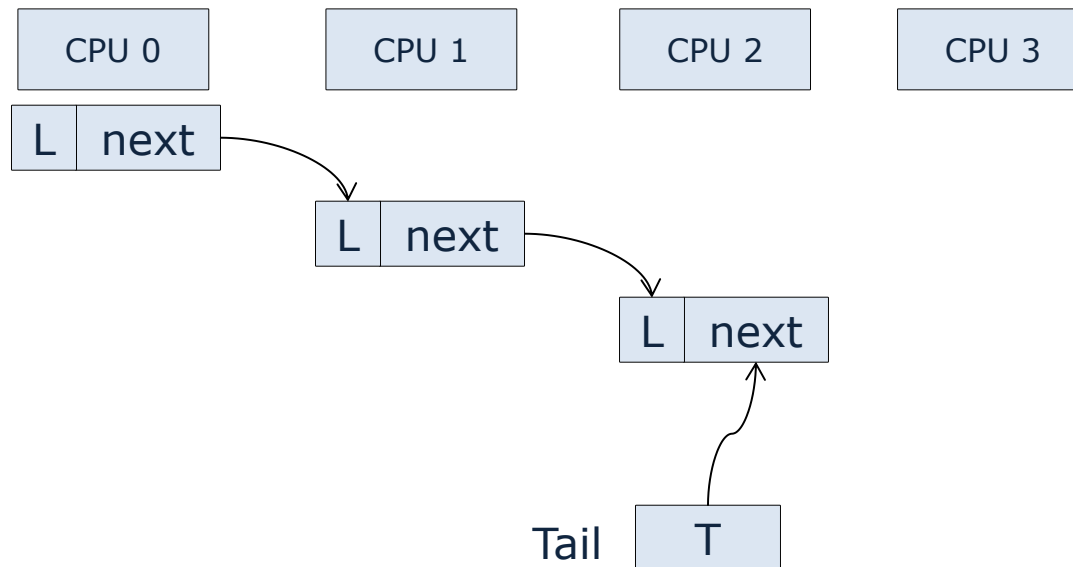


```
insert(new_elem, prev) {
  retry:
    next = prev.next;
    new_elem.next = prev.next;
    new_elem.prev = prev;
    if (not DCAS(prev.next == next && next.prev == prev,
                 prev.next = new_elem, next.prev = new_elem))
      goto retry;
}
```

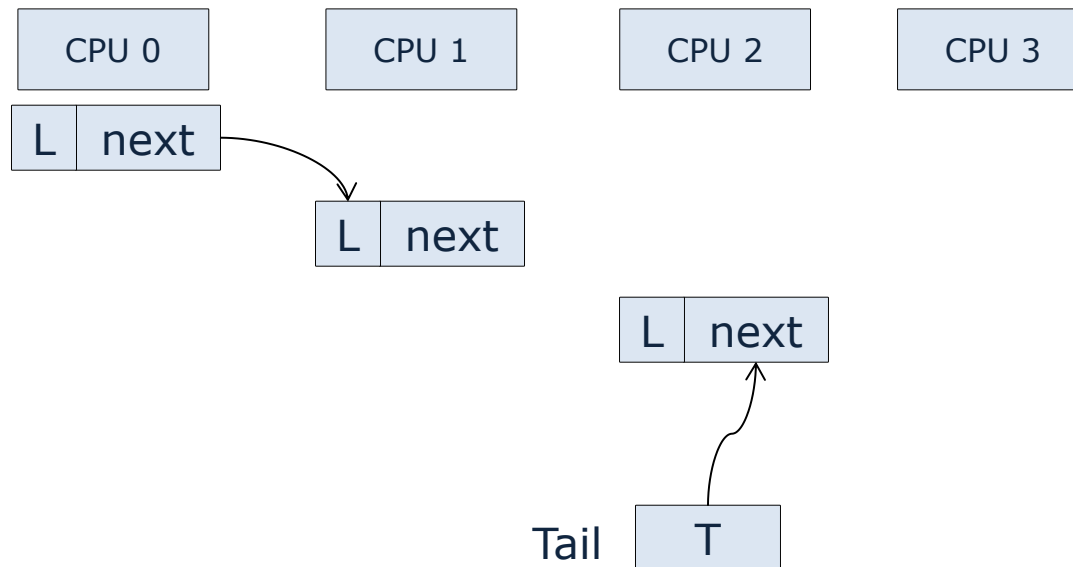
Load Linked, Store Conditional

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

Fairness + Local Spinning by Mellor-Crummey and Scott



Fairness + Local Spinning by Mellor-Crummey and Scott



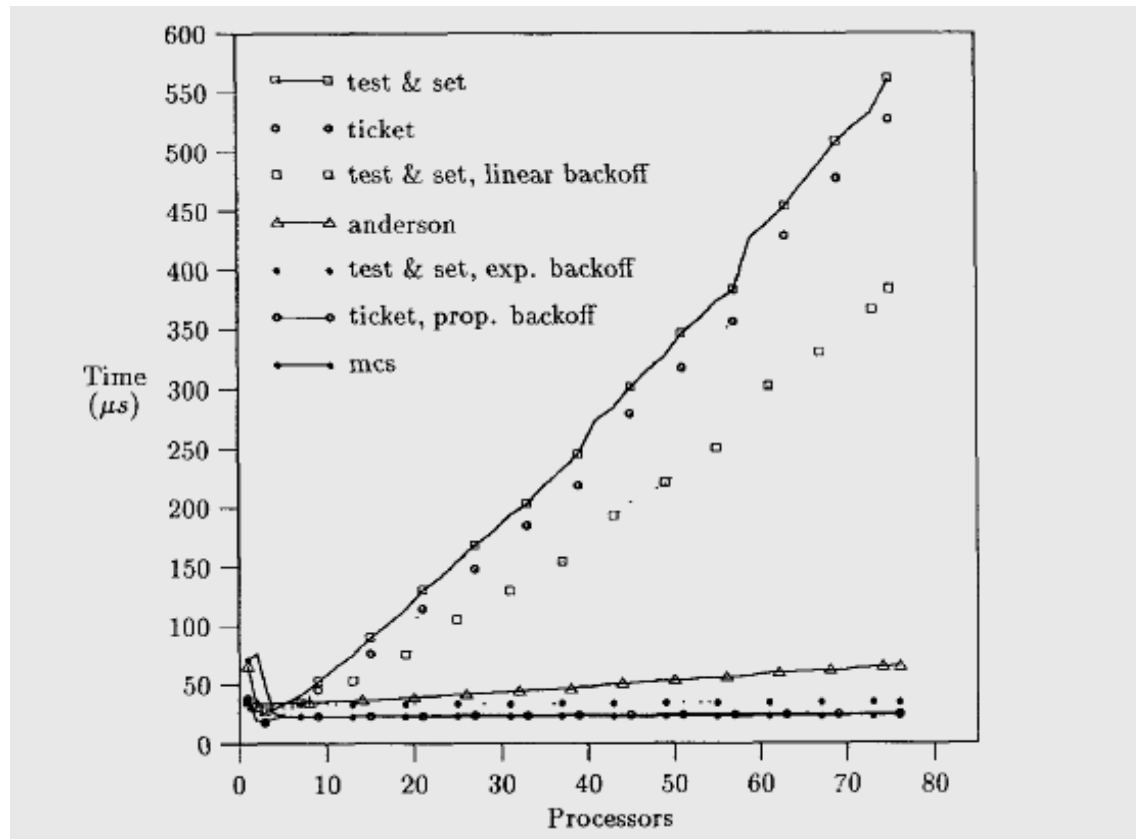
Fair, local spinning

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);
    }
}
```

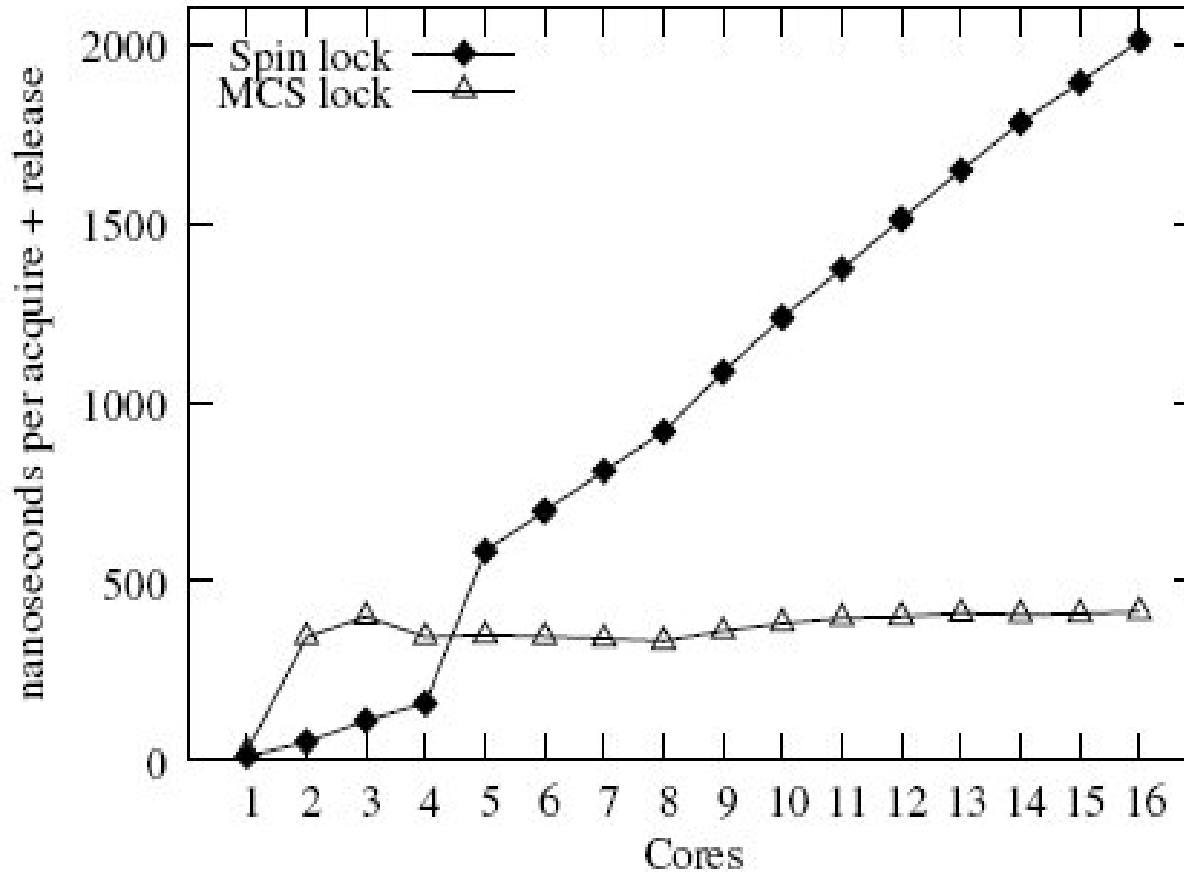
```
unlock (Node * & T, Node * I) {
    if (!I->next) {
        if (cmpxchg (T == I, 0)) return; // no waiting cpu
        do { } while (!I->next); // spin until the following process
                                // updates the next pointer
    }
    I->next->Lock = true;
}
```

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 $\log_4(\text{depth})$ switched network; Anderson: array-based queue lock

Source: [corey 08]



16 core AMD Opteron

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)

Fair Ticket Reader-Writer Lock

co-locate reader tickets and writer tickets

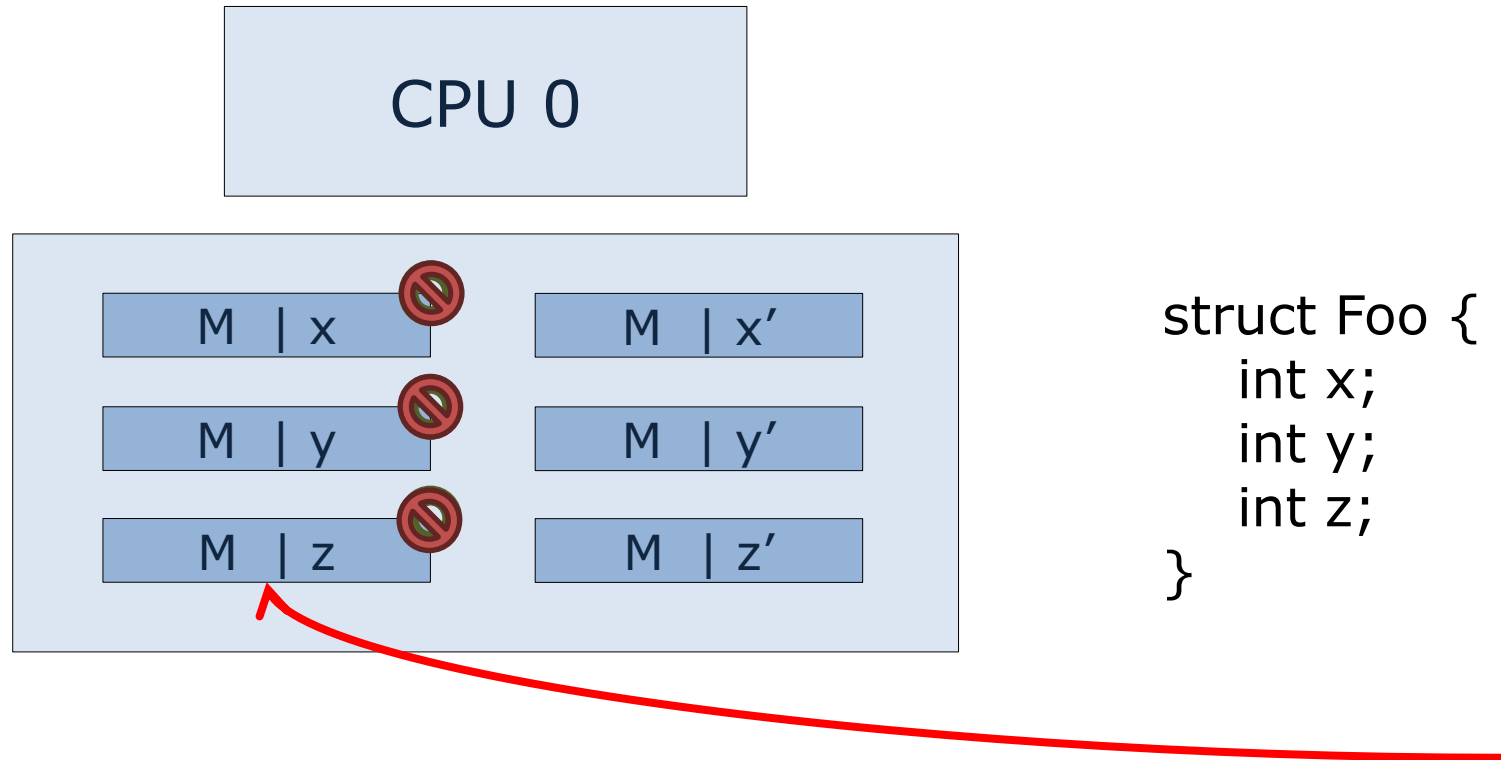
```
lock read (next, current) {
  my_ticket = xadd (next, 1);
  do {} while (current.write != my_ticket.write);
}
```

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

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

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

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




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) {  
  
    do {  
        sti; // enable interrupts  
        reg = 1;  
        do {} while (lock_var == 1);  
  
        cli; // disable interrupts  
        swap(lock_var, reg);  
    } while (reg == 1);  
}  
  
spin_unlock(lock_var) {  
    lock_var = 0;  
    sti;  
}
```



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;
}
```

MWait:

stop CPU / Hyperthread

wait for cacheline to be written

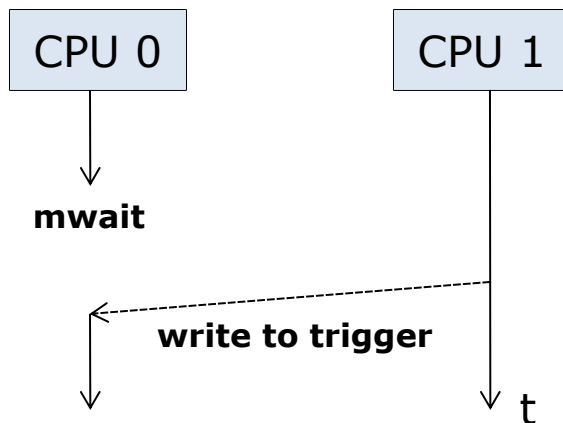
=> save power by allowing CPU to enter sleep state

=> free resources for other hyperthread

Monitor:

set watchdog to cacheline

watchdog may also be triggered by other events (interrupts)



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

- A Primer on Memory Consistency and Cache Coherence
Sorin, Hill, Wood; 2011
- [atomic<> Weapons](#): The C++ Memory Model and Modern Hardware (Video)
Sutter; 2013
- [Shared memory consistency models: a tutorial](#)
Adve, Gharachorloo; 1996
- [IA Memory Model](#)
Richard Hudson; Google Tech Talk 2008
- [Memory Ordering in Modern Microprocessors](#)
McKenney; Linux Journal 2005
- How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs
Lampert, 1979
- [PowerPC Storage Model](#)

Scheduler-Conscious Synchronization

Leonidas Kontothanassis, Robert Wisniewski, Michael Scott

Scalable Reader- Writer Synchronization for Shared-Memory Multiprocessors

John M. Mellor-Crummey, Michael L. Scott

Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors

John Mellor-Crummey, Michael Scott

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)

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