Distributed Operating Systems

Synchronization in Parallel Systems

Till Smejkal

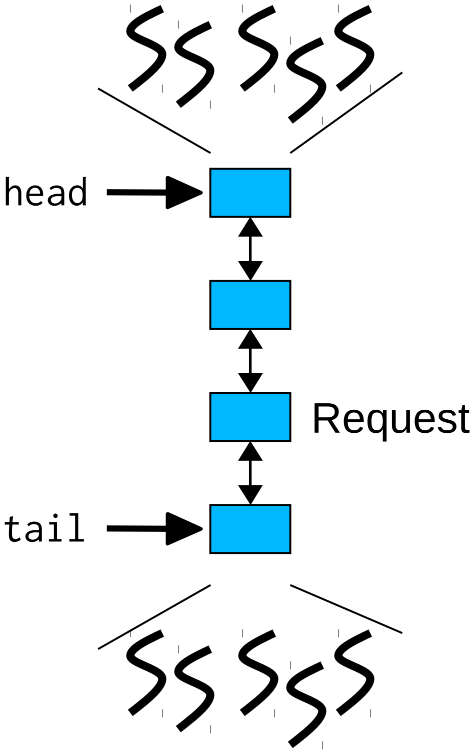
2016

TECHNISCHE
UNIVERSITÄT
DRESDEN

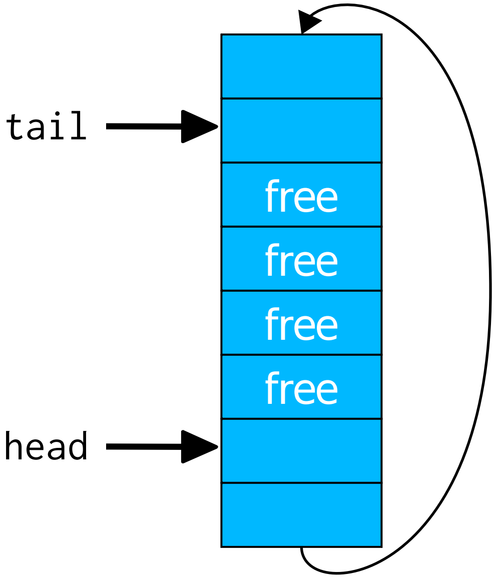
Overview

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

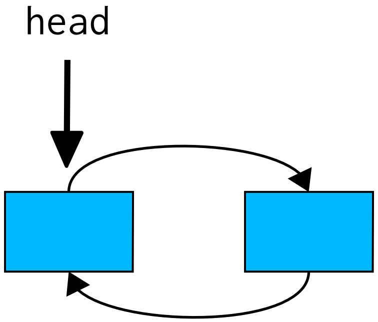
Introduction



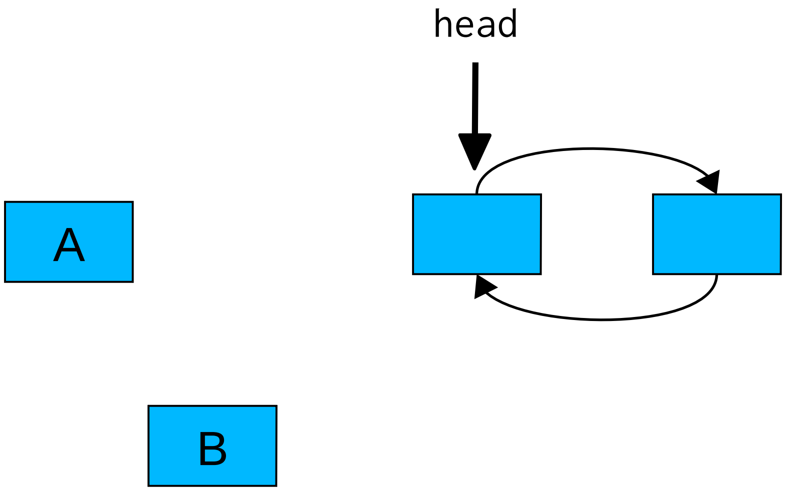
Circular Buffer



Introduction

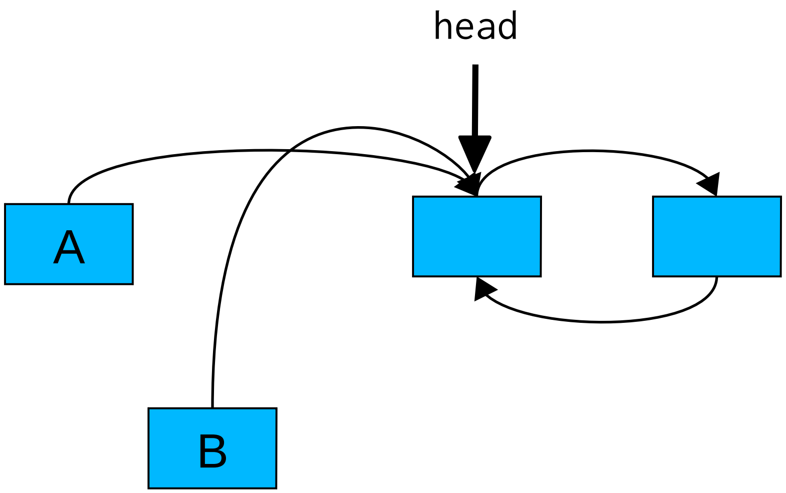


Introduction



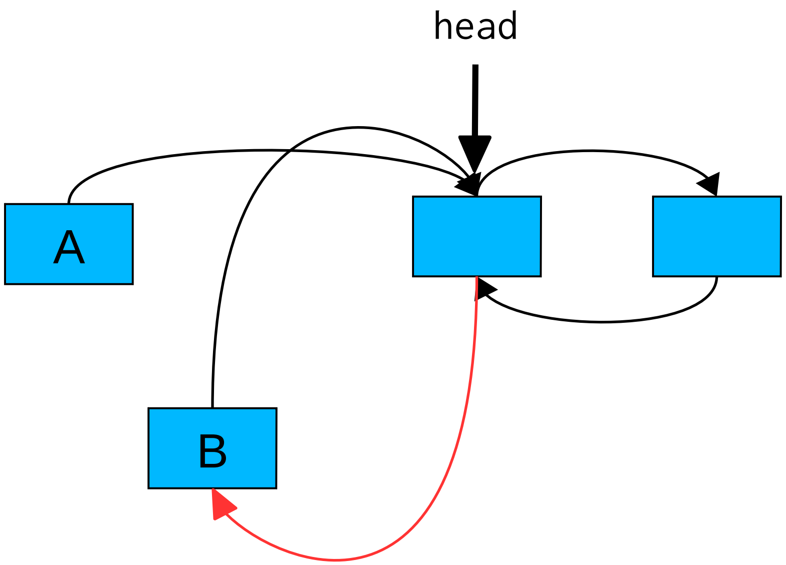
1) A,B create list element

Introduction



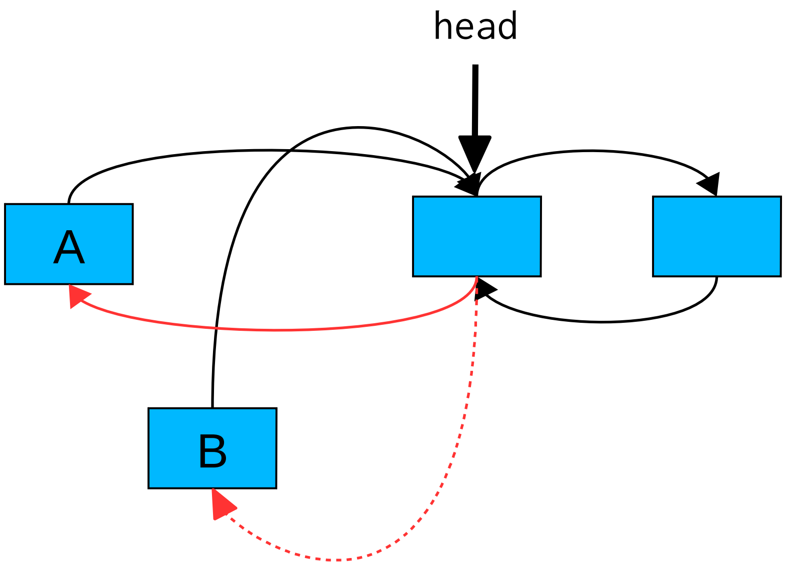
1) A,B create list element
2) A,B set `next` pointer

Introduction



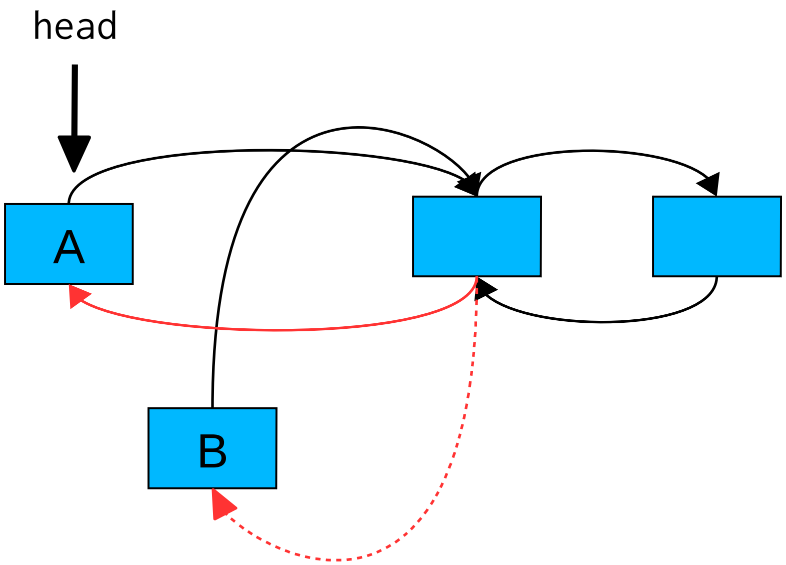
1) A,B create list element
2) A,B set `next` pointer
3) B sets `prev` pointer

Introduction



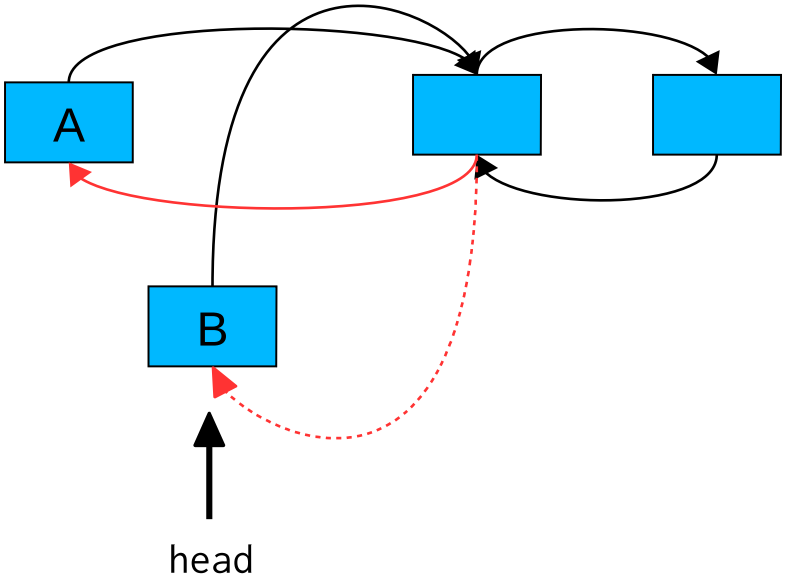
1) A,B create list element
2) A,B set `next` pointer
3) B sets `prev` pointer
4) A sets `prev` pointer

Introduction



1) A,B create list element
2) A,B set `next` pointer
3) B sets `prev` pointer
4) A sets `prev` pointer
5) A updates `head`

Introduction



1) A,B create list element
2) A,B set `next` pointer
3) B sets `prev` pointer
4) A sets `prev` pointer
5) A updates `head`
6) B updates `head`

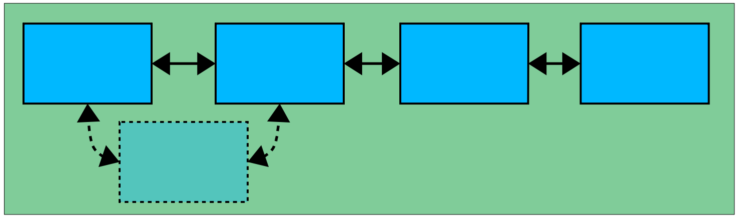
Mutual Exclusion – w/ Locks

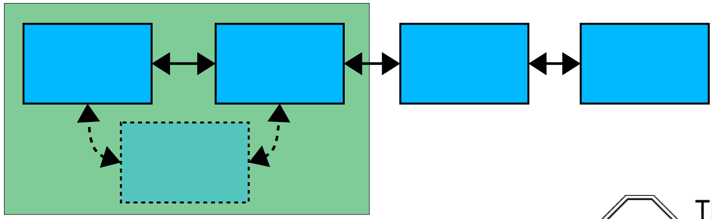
coarse grained (lock entire list)

```
lock(list);
list->insert_element(ele);
unlock(list);
```

fine grained (lock list elements)

```
retry:
  lock(head);
  if (trylock(head->next)) {
    head->insert_element(ele);
    unlock(head->next);
  } else {
    unlock(head);
    goto retry;
  }
  unlock(head)
```





Mutual Exclusion – w/o Locks

Decker / Peterson

- atomic stores, atomic loads
- sequential consistent memory (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

- A, B are atomic if A || B = A;B **or** B;A
- Read-Modify-Write instructions are typically not atomic

```
       A                          B
add &x, 1       ||        mov &x, 2          (x = 0)
```

Atomic Hardware Instructions

- A, B are atomic if A || B = A;B **or** B;A
- Read-Modify-Write instructions are typically not atomic

```
      A                          B
load &x -> Reg                                 (x = 0)
add Reg, 1        ||        store 2 -> &x
store Reg -> &x
```

Atomic Hardware Instructions

- A, B are atomic if A || B = A;B **or** B;A
- Read-Modify-Write instructions are typically not atomic

```
        A                          B
load &x -> Reg                                         (x = 0)
add Reg, 1
                           store 2 -> &x
store Reg -> &x
```

x == 1

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 a RMW instruction have completed
  - Intel Pentium 3 and older x86 CPUs

```
lock; add &x, 1; unlock
```

- Cache Lock
  - Delay snoop traffic until all memory accesses of a RMW instruction have completed
  - Intel Pentium 4 and newer x86 CPUs

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
  - ARM, Alpha

```
retry:
  load_linked &x -> Reg;
  modify Reg;
  if (!store_conditional(Reg -> &x))
    goto retry
```

- Hardware Transactional Memory
  - Install watchdog for all memory used by the transaction
  - Discard changes on write-write or write-read conflicts
  - Intel TSX, IBM BlueGeneQ

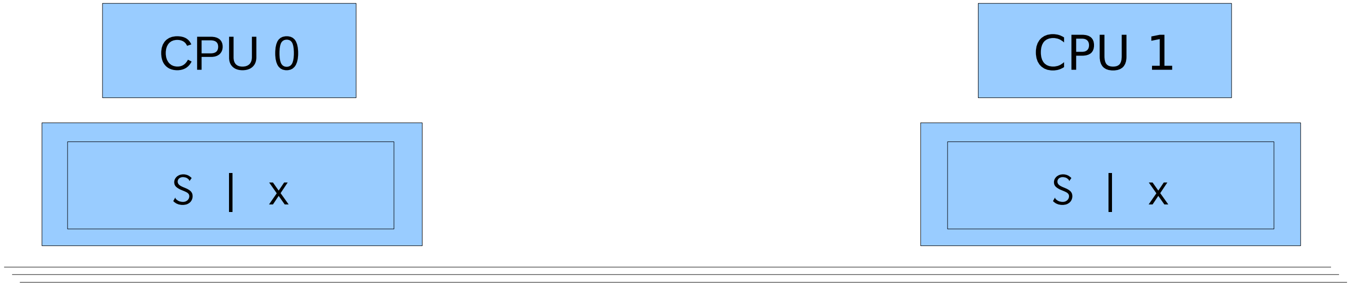
Atomic Hardware Instructions

How to make instructions atomic?

- Cache Lock
  - Delay snoop traffic until all memory accesses of a RMW instruction have completed
  - Can be achieved with the M(O)ESI Cache Coherence protocol

```
add &x, 1
    1. read_for_ownership(&x)
    2. load &x -> Reg
    3. add Reg, 1
    4. store Reg -> &x
```

```
mov &x, 2

    2. store 2 -> &x
```



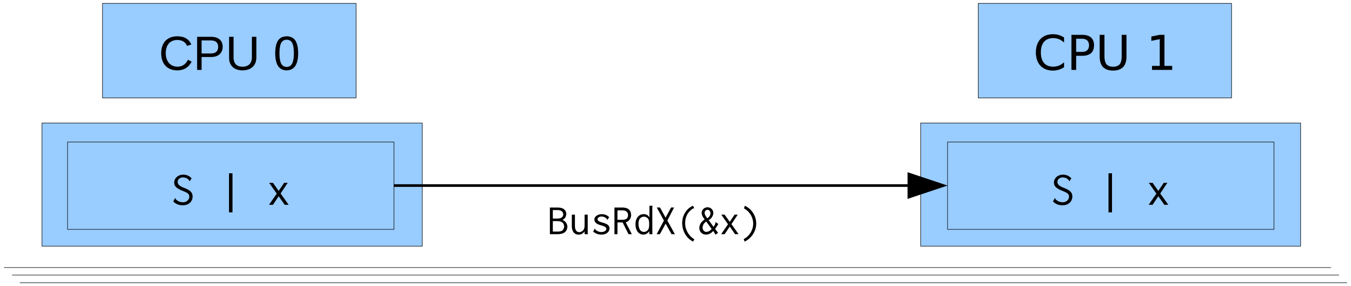
Atomic Hardware Instructions

How to make instructions atomic?

- Cache Lock
  - Delay snoop traffic until all memory accesses of a RMW instruction have completed
  - Can be achieved with the M(O)ESI Cache Coherence protocol

```
add &x, 1
    1. read_for_ownership(&x)
    2. load &x -> Reg
    3. add Reg, 1
    4. store Reg -> &x
```

```
mov &x, 2

    2. store 2 -> &x
```



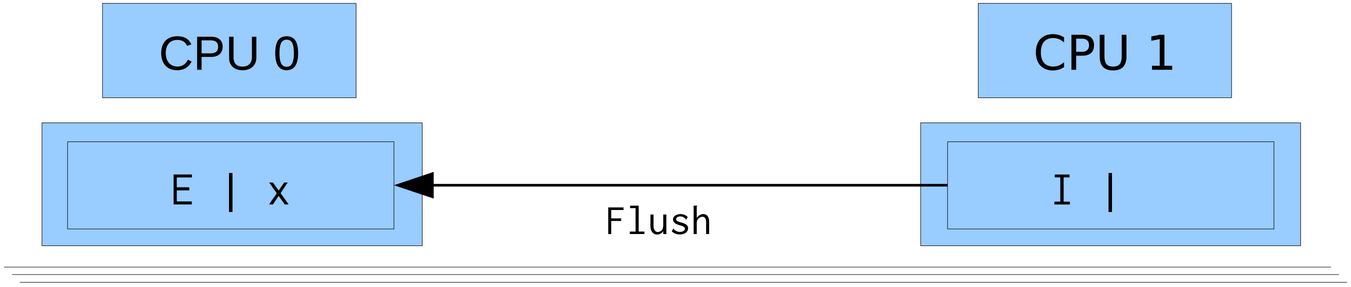
Atomic Hardware Instructions

How to make instructions atomic?

- Cache Lock
  - Delay snoop traffic until all memory accesses of a RMW instruction have completed
  - Can be achieved with the M(O)ESI Cache Coherence protocol

```
add &x, 1
    1. read_for_ownership(&x)
    2. load &x -> Reg
    3. add Reg, 1
    4. store Reg -> &x
```

```
mov &x, 2

    2. store 2 -> &x
```



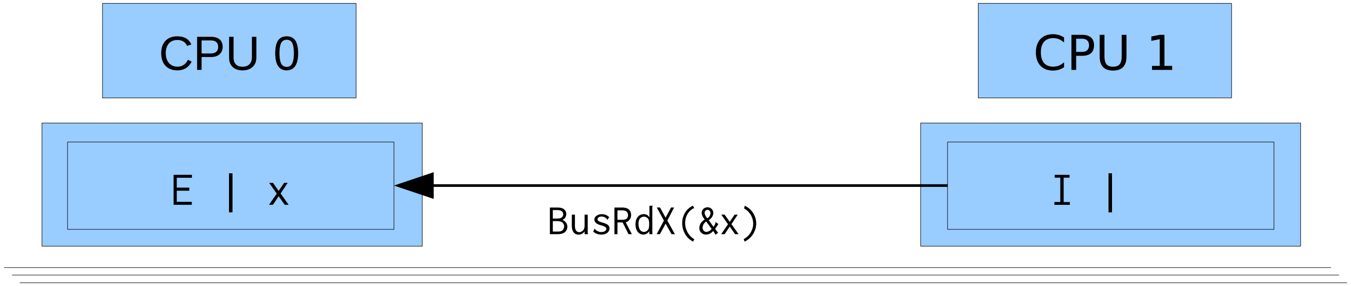
Atomic Hardware Instructions

How to make instructions atomic?

- Cache Lock
  - Delay snoop traffic until all memory accesses of a RMW instruction have completed
  - Can be achieved with the M(O)ESI Cache Coherence protocol

```
add &x, 1
    1. read_for_ownership(&x)
    2. load &x -> Reg
    3. add Reg, 1
    4. store Reg -> &x
```

```
mov &x, 2

    2. store 2 -> &x
```



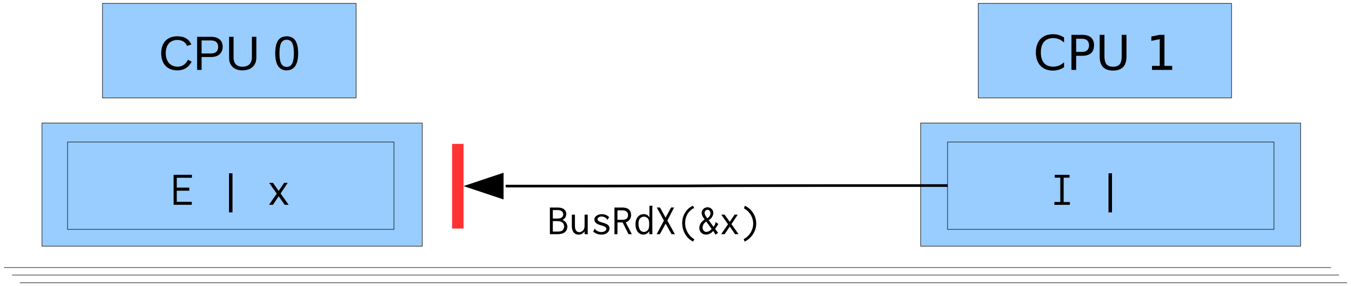
Atomic Hardware Instructions

How to make instructions atomic?

- Cache Lock
  - Delay snoop traffic until all memory accesses of a RMW instruction have completed
  - Can be achieved with the M(O)ESI Cache Coherence protocol

```
add &x, 1
    1. read_for_ownership(&x)
    2. load &x -> Reg
    3. add Reg, 1
    4. store Reg -> &x
```

```
mov &x, 2

    2. store 2 -> &x
```



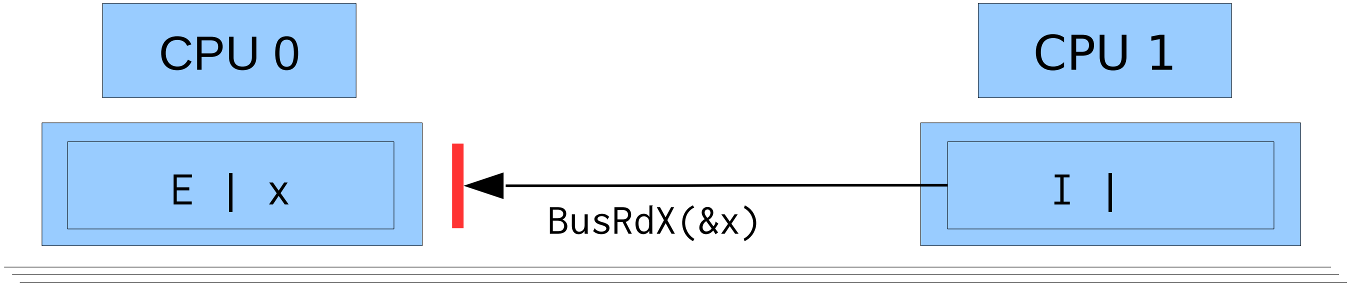
Atomic Hardware Instructions

How to make instructions atomic?

- Cache Lock
  - Delay snoop traffic until all memory accesses of a RMW instruction have completed
  - Can be achieved with the M(O)ESI Cache Coherence protocol

```
add &x, 1
    1. read_for_ownership(&x)
    2. load &x -> Reg
    3. add Reg, 1
    4. store Reg -> &x
```

```
mov &x, 2

    2. store 2 -> &x
```



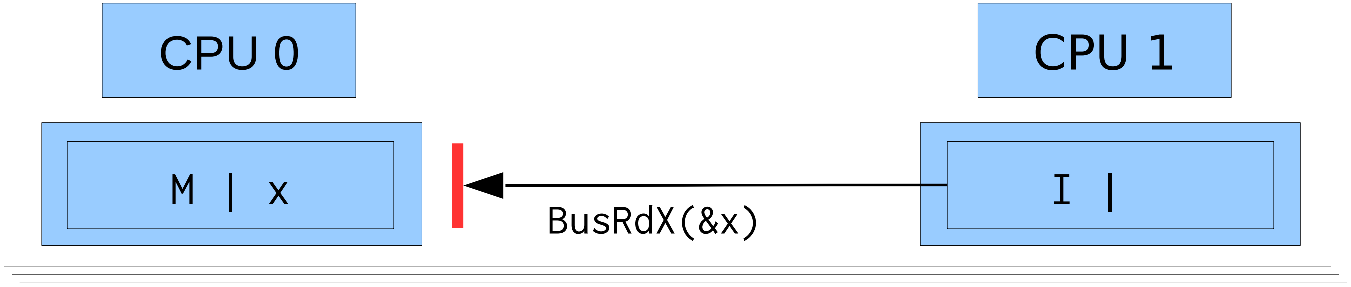
Atomic Hardware Instructions

How to make instructions atomic?

- Cache Lock
  - Delay snoop traffic until all memory accesses of a RMW instruction have completed
  - Can be achieved with the M(O)ESI Cache Coherence protocol

```
add &x, 1
    1. read_for_ownership(&x)
    2. load &x -> Reg
    3. add Reg, 1
    4. store Reg -> &x
```

```
mov &x, 2

    2. store 2 -> &x
```



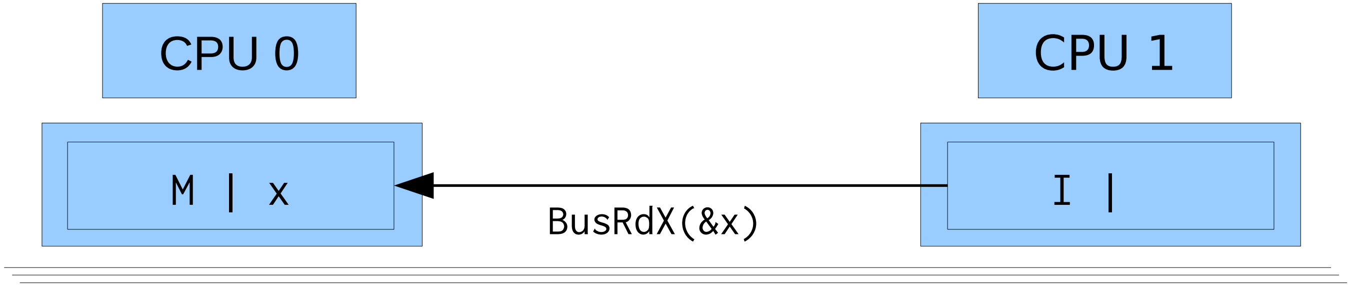
Atomic Hardware Instructions

How to make instructions atomic?

- Cache Lock
  - Delay snoop traffic until all memory accesses of a RMW instruction have completed
  - Can be achieved with the M(O)ESI Cache Coherence protocol

```
add &x, 1
    1. read_for_ownership(&x)
    2. load &x -> Reg
    3. add Reg, 1
    4. store Reg -> &x
```

```
mov &x, 2

    2. store 2 -> &x
```



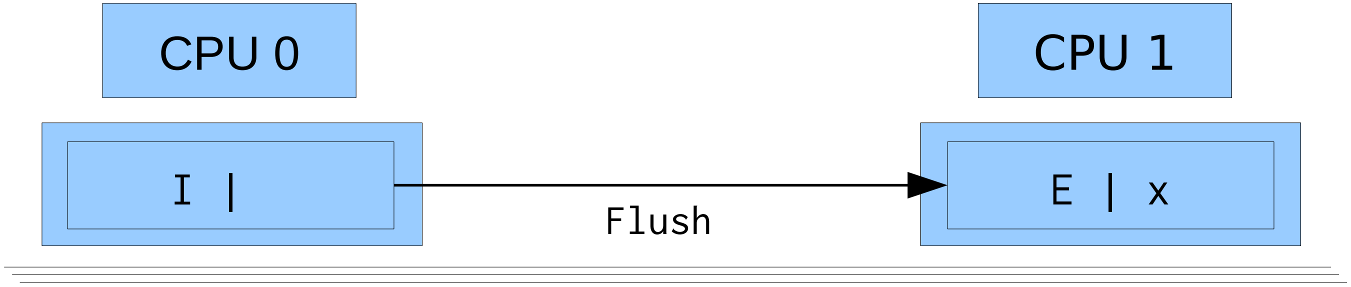
Atomic Hardware Instructions

How to make instructions atomic?

- Cache Lock
  - Delay snoop traffic until all memory accesses of a RMW instruction have completed
  - Can be achieved with the M(O)ESI Cache Coherence protocol

```
add &x, 1
    1. read_for_ownership(&x)
    2. load &x -> Reg
    3. add Reg, 1
    4. store Reg -> &x
```

```
mov &x, 2

    2. store 2 -> &x
```



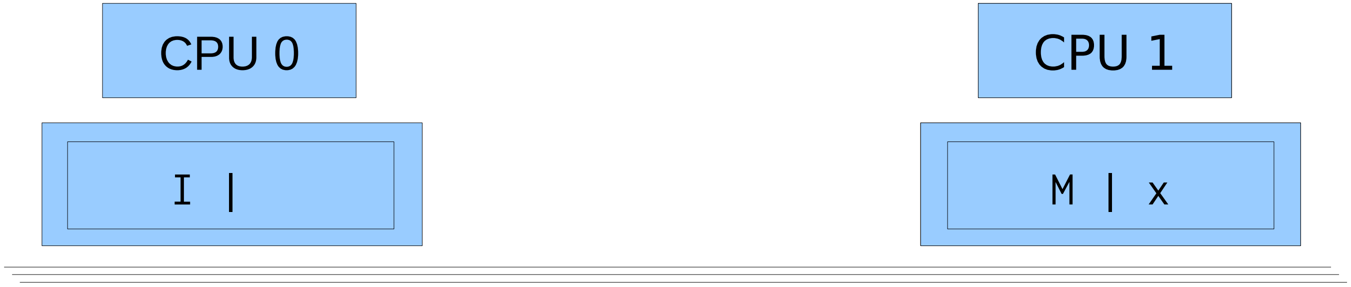
Atomic Hardware Instructions

How to make instructions atomic?

- Cache Lock
  - Delay snoop traffic until all memory accesses of a RMW instruction have completed
  - Can be achieved with the M(O)ESI Cache Coherence protocol

```
add &x, 1
    1. read_for_ownership(&x)
    2. load &x -> Reg
    3. add Reg, 1
    4. store Reg -> &x
```

```
mov &x, 2

    2. store 2 -> &x
```



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

```
add &x, 1
    1. load_linked &x -> Reg
    2. add Reg, 1
    3. store_conditional Reg -> &x
```

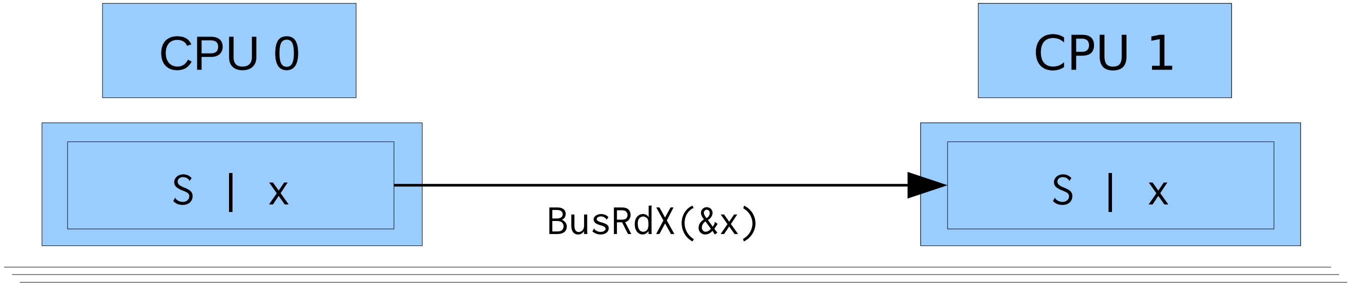
CPU 0
S | x
CPU 1
S | x

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

```
add &x, 1
    1. load_linked &x -> Reg
    2. add Reg, 1
    3. store_conditional Reg -> &x
```

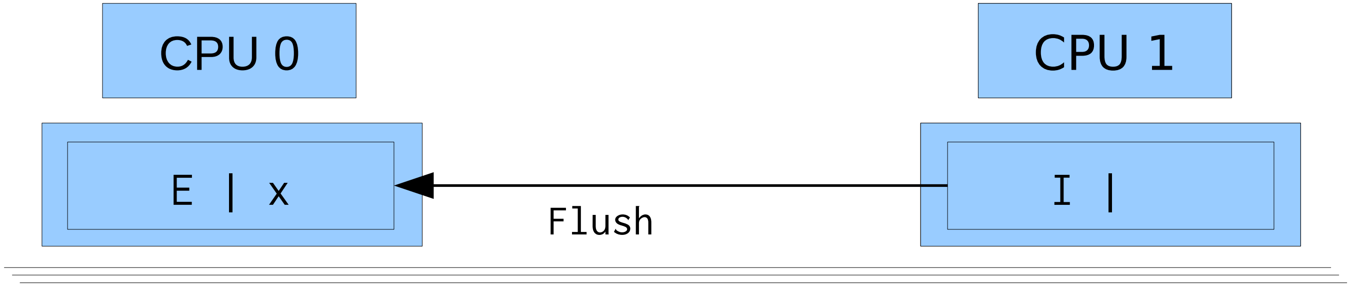


Atomic Hardware Instructions

How to make instructions atomic?

- Observe Cache
    - Install cache watchdog on load
    - Abort store if watchdog has detected a concurrent access; retry OP

```
add &x, 1
    1. load_linked &x -> Reg
    2. add Reg, 1
    3. store_conditional Reg -> &x
```

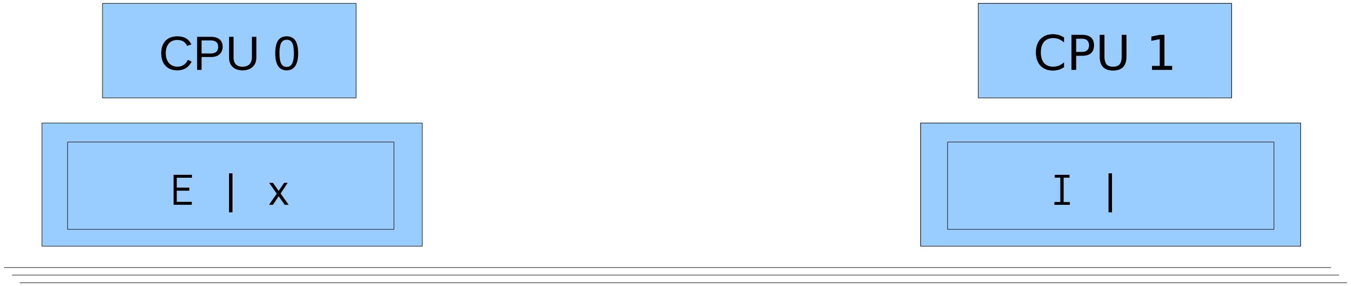


Atomic Hardware Instructions

How to make instructions atomic?

- Observe Cache
  - Install cache watchdog on load
  - Abort store if watchdog has detected a concurrent access; retry OP

```
add &x, 1
    1. load_linked &x -> Reg
    2. add Reg, 1
    3. store_conditional Reg -> &x
```



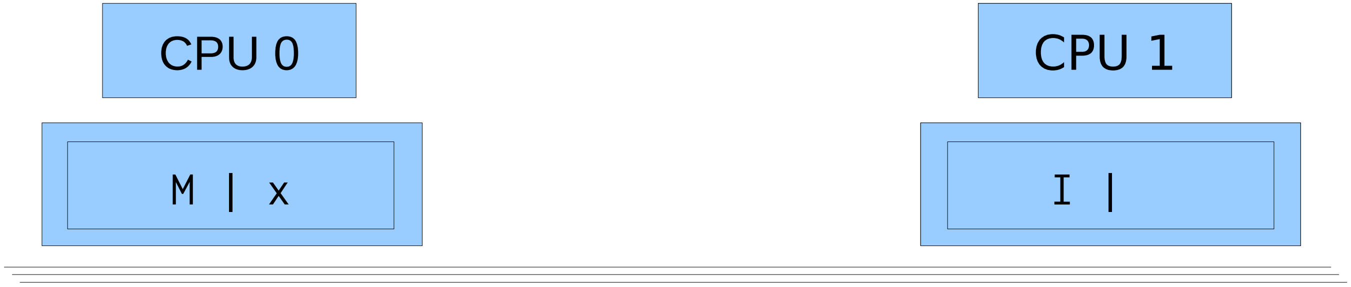
Atomic Hardware Instructions

How to make instructions atomic?

- Observe Cache
  - Install cache watchdog on load
  - Abort store if watchdog has detected a concurrent access; retry OP

```
add &x, 1
    1. load_linked &x -> Reg
    2. add Reg, 1
    3. store_conditional Reg -> &x
```

→ **Ok**



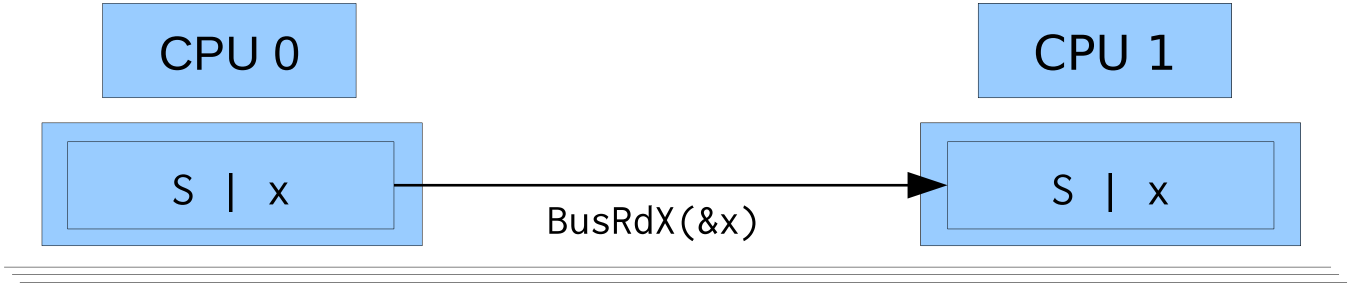
Atomic Hardware Instructions

How to make instructions atomic?

- Observe Cache
  - Install cache watchdog on load
  - Abort store if watchdog has detected a concurrent access; retry OP

```
add &x, 1
    1. load_linked &x -> Reg
    2. add Reg, 1
    3. store_conditional Reg -> &x
```

```
mov &x, 2

    2. store 2 -> &x
```



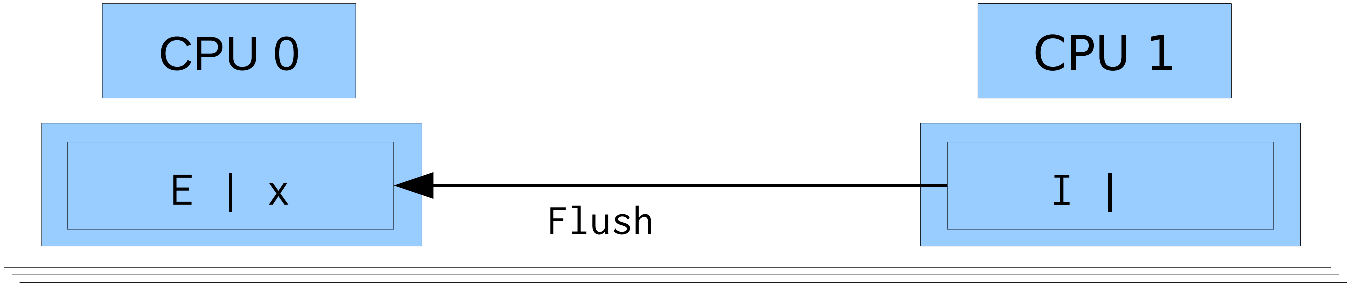
Atomic Hardware Instructions

How to make instructions atomic?

- Observe Cache
  - Install cache watchdog on load
  - Abort store if watchdog has detected a concurrent access; retry OP

```
add &x, 1
    1. load_linked &x -> Reg
    2. add Reg, 1
    3. store_conditional Reg -> &x
```

```
mov &x, 2

    2. store 2 -> &x
```



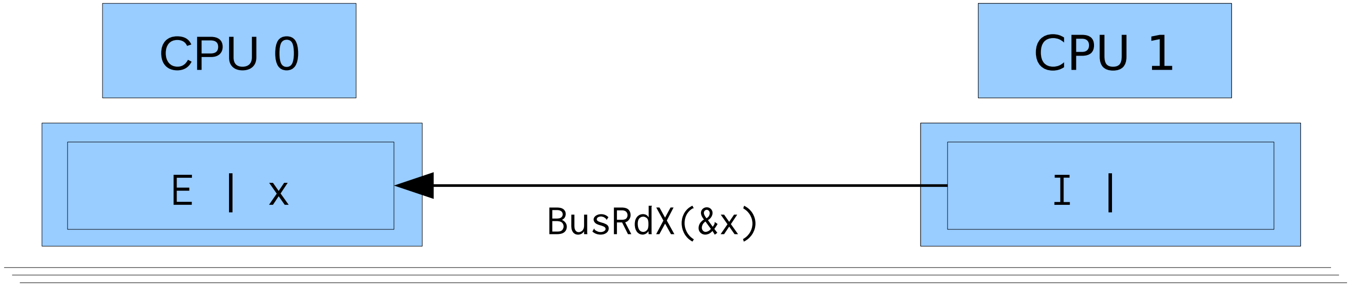
Atomic Hardware Instructions

How to make instructions atomic?

- Observe Cache
  - Install cache watchdog on load
  - Abort store if watchdog has detected a concurrent access; retry OP

```
add &x, 1
    1. load_linked &x -> Reg
    2. add Reg, 1
    3. store_conditional Reg -> &x
```

```
mov &x, 2

    2. store 2 -> &x
```



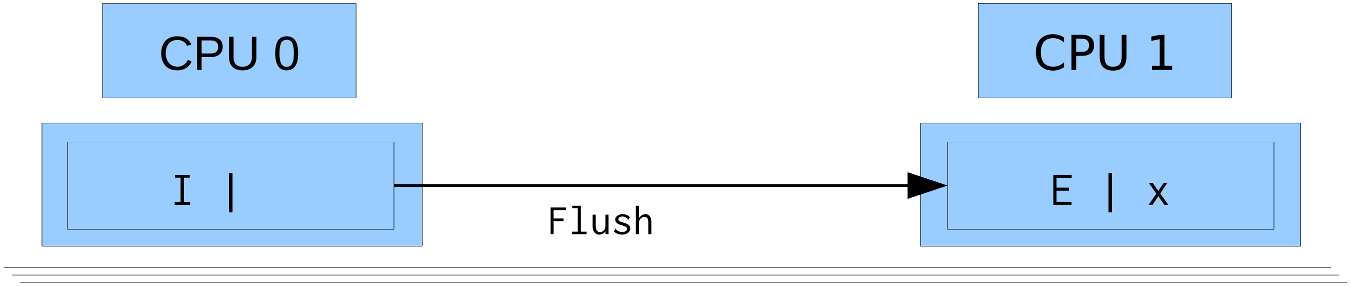
Atomic Hardware Instructions

How to make instructions atomic?

- Observe Cache
  - Install cache watchdog on load
  - Abort store if watchdog has detected a concurrent access; retry OP

```
add &x, 1
    1. load_linked &x -> Reg
    2. add Reg, 1
    3. store_conditional Reg -> &x
```

```
mov &x, 2

    2. store 2 -> &x
```



Atomic Hardware Instructions

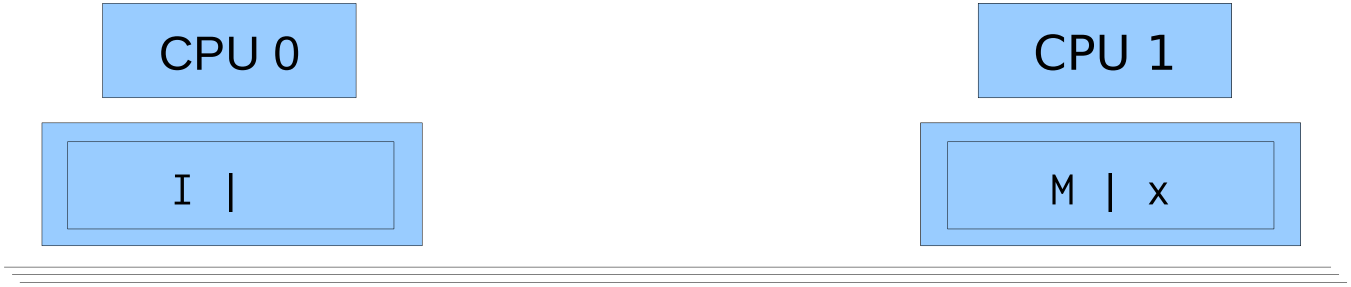
How to make instructions atomic?

- Observe Cache
  - Install cache watchdog on load
  - Abort store if watchdog has detected a concurrent access; retry OP

```
add &x, 1
    1. load_linked &x -> Reg
    2. add Reg, 1
    3. store_conditional Reg -> &x
```

→ **Abort**

```
mov &x, 2

    2. store 2 -> &x
```



Atomic Hardware Instructions

- Bit Test and Set

```
bit_test_and_set (bit) {
    if (bit clear) {
        set bit ; return true;
    } else { return false; }
}
```

- Exchange

```
swap (mem, Reg) {
    mov &mem, tmp;
    mov Reg, &mem;
    mov tmp, Reg;
}
```

- Fetch and Add

```
xadd (mem, Reg) {
    mov &mem, tmp;
    add &mem, Reg;
    return tmp;
}
```

Atomic Hardware Instructions

- Compare and Swap

```
cas (mem, expected, desired) {
    if (&mem == expected) {
        mov desired, &mem; return true;
    } else { return false; }
}
```

- Double Compare and Swap

```
cas (mem1, mem2, exp1, exp2, des1, des2) {
    if (&mem1 == exp1 && &mem2 == exp2) {
        mov des1, &mem1;
        mov des2, &mem2;
        return true;
    } else { return false; }
}
```

Overview

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

Synchronization w/ Locks

Properties

- Overhead
  - Taking a lock should be cheap (<10% of CS)
  - Minimize overhead if lock is currently free
- Fairness
  - Every thread should be able to obtain the lock after a finite amount of time
- Abort lock()-operations
  - Abort locking after a specified timeout
  - Stop threads which are currently waiting for a lock
- Concurrent access to CS
  - Support that multiple threads can enter the lock at the same time

Synchronization w/ Locks

Properties

- Lock-holder preemption
  - Preemption of the thread currently executing the CS
- Priority inversion
  - Prevent higher priority thread from executing because of lower priority thread holding shared lock

    → Not covered in this lecture! (See MKC or RTS)
- Spinning vs. Blocking
  - Release CPU while waiting for the lock to be free again
  - Latency and performance implications

Synchronization w/ Locks

Spin Lock (Test & Set Lock)

```
void lock (lock_t *l) {
    do {
        int tmp = 1;
        swap (l->lock, tmp);
    } while (tmp == 1);
}

void unlock (lock_t *l) {
    l->lock = 0
}
```

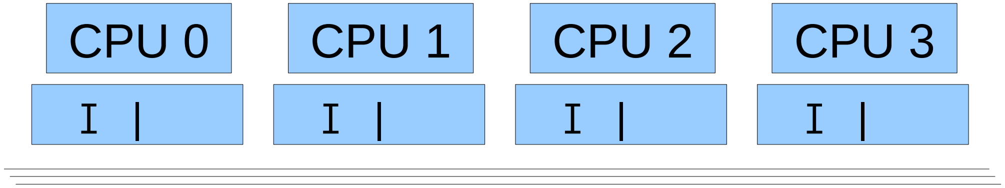
+ only one cheap atomic OP required

– high cache bus traffic while lock is held:

Synchronization w/ Locks

Spin Lock (Test & Set Lock)

```
void lock (lock_t *l) {
    do {
        int tmp = 1;
        swap (l->lock, tmp);
    } while (tmp == 1);
}

void unlock (lock_t *l) {
    l->lock = 0
}
```



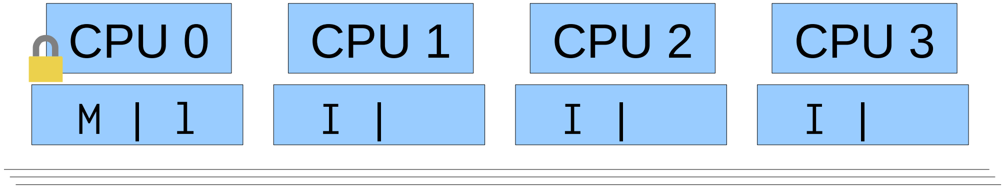
+ only one cheap atomic OP required

– high cache bus traffic while lock is held

Synchronization w/ Locks

Spin Lock (Test & Set Lock)

```
void lock (lock_t *l) {
    do {
        int tmp = 1;
        swap (l->lock, tmp);
    } while (tmp == 1);
}

void unlock (lock_t *l) {
    l->lock = 0
}
```



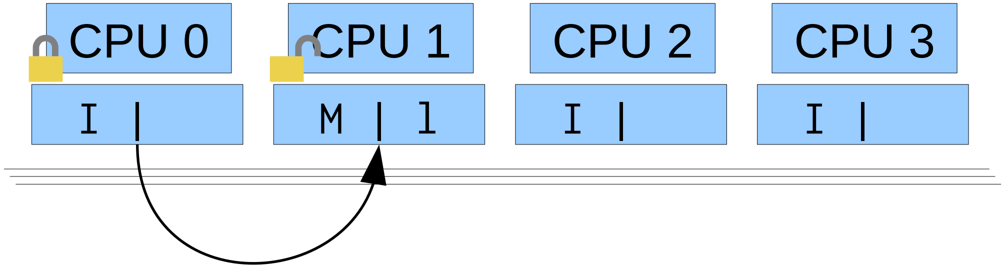
+ only one cheap atomic OP required

– high cache bus traffic while lock is held

Synchronization w/ Locks

Spin Lock (Test & Set Lock)

```
void lock (lock_t *l) {
    do {
        int tmp = 1;
        swap (l->lock, tmp);
    } while (tmp == 1);
}


void unlock (lock_t *l) {
    l->lock = 0
}
```



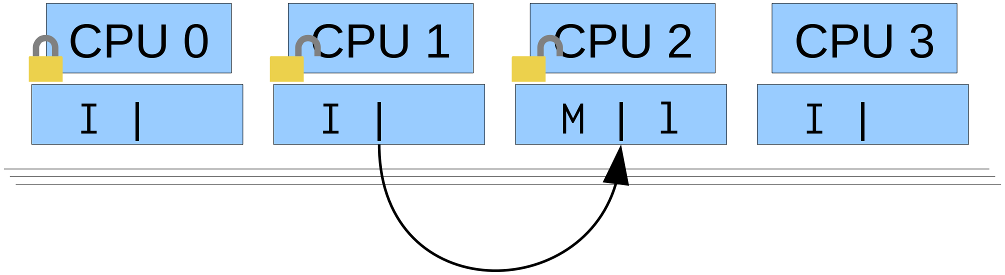
+ only one cheap atomic OP required

– high cache bus traffic while lock is held

Synchronization w/ Locks

Spin Lock (Test & Set Lock)

```
void lock (lock_t *l) {
    do {
        int tmp = 1;
        swap (l->lock, tmp);
    } while (tmp == 1);
}

void unlock (lock_t *l) {
    l->lock = 0
}
```



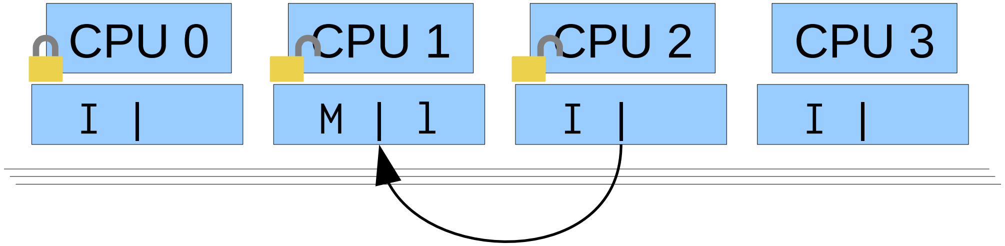
+ only one cheap atomic OP required

– high cache bus traffic while lock is held

Synchronization w/ Locks

Spin Lock (Test & Set Lock)

```
void lock (lock_t *l) {
    do {
        int tmp = 1;
        swap (l->lock, tmp);
    } while (tmp == 1);
}

void unlock (lock_t *l) {
    l->lock = 0
}
```



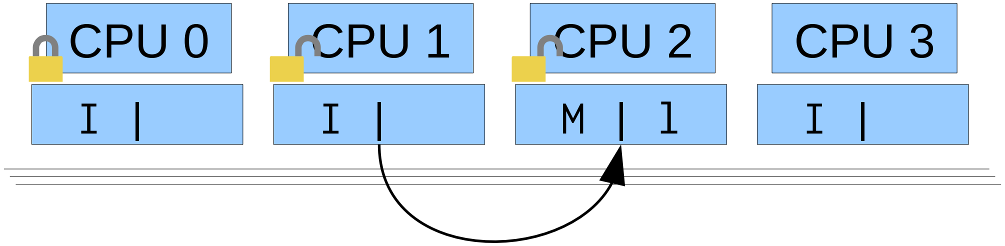
+ only one cheap atomic OP required

– high cache bus traffic while lock is held

Synchronization w/ Locks

Spin Lock (Test & Set Lock)

```
void lock (lock_t *l) {
    do {
        int tmp = 1;
        swap (l->lock, tmp);
    } while (tmp == 1);
}

void unlock (lock_t *l) {
    l->lock = 0
}
```



+ only one cheap atomic OP required

– high cache bus traffic while lock is held

Synchronization w/ Locks

Spin Lock (Test & Test & Set Lock)

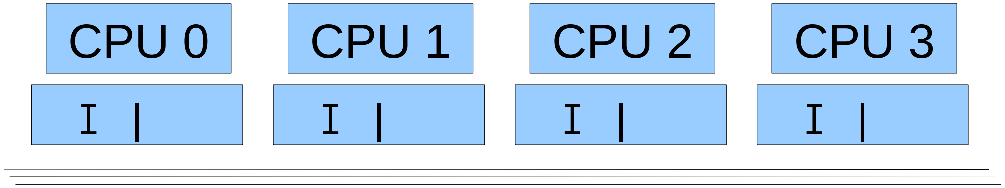
```
void lock (lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap (l->lock, tmp);
    } while (tmp == 1);
}

void unlock (lock_t *l) {
    l->lock = 0
}
```

+ spins locally while lock is held by other CPU
+ like Test & Set Lock but with fewer cache bus traffic

Synchronization w/ Locks

Spin Lock (Test & Test & Set Lock)

```
void lock (lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap (l->lock, tmp);
    } while (tmp == 1);
}

void unlock (lock_t *l) {
    l->lock = 0
}
```



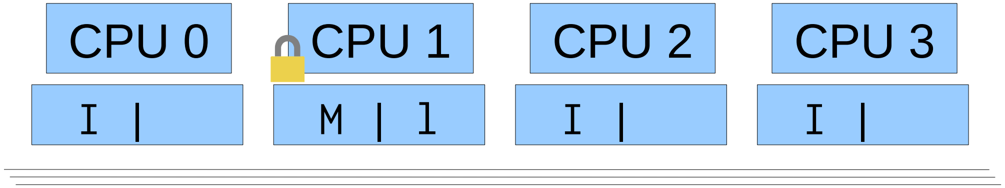
+ spins locally while lock is held by other CPU

+ like Test & Set Lock but with fewer cache bus traffic

Synchronization w/ Locks

Spin Lock (Test & Test & Set Lock)

```
void lock (lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap (l->lock, tmp);
    } while (tmp == 1);
}

void unlock (lock_t *l) {
    l->lock = 0
}
```

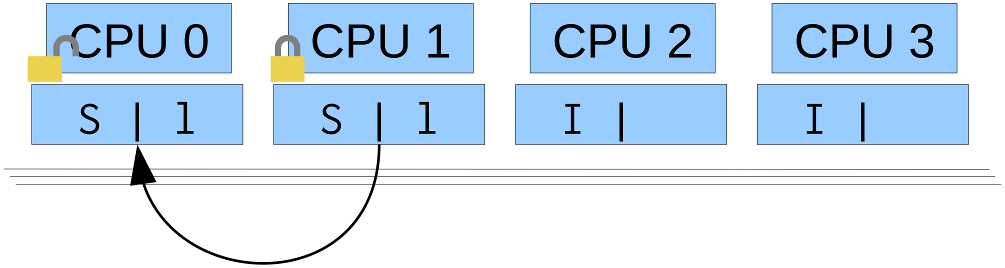


+ spins locally while lock is held by other CPU
+ like Test & Set Lock but with fewer cache bus traffic

Synchronization w/ Locks

Spin Lock (Test & Test & Set Lock)

```
void lock (lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap (l->lock, tmp);
    } while (tmp == 1);
}

void unlock (lock_t *l) {
    l->lock = 0
}
```

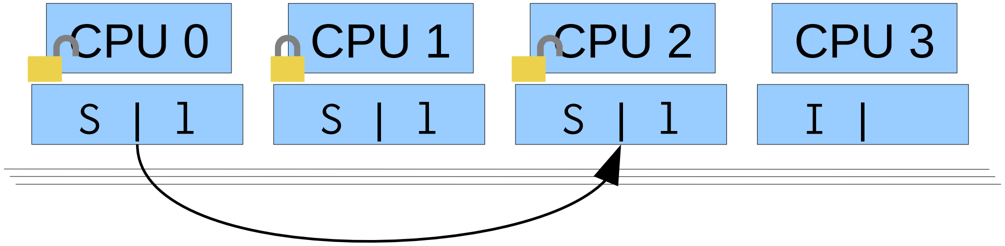


+ spins locally while lock is held by other CPU
+ like Test & Set Lock but with fewer cache bus traffic

Synchronization w/ Locks

Spin Lock (Test & Test & Set Lock)

```
void lock (lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap (l->lock, tmp);
    } while (tmp == 1);
}

void unlock (lock_t *l) {
    l->lock = 0
}
```

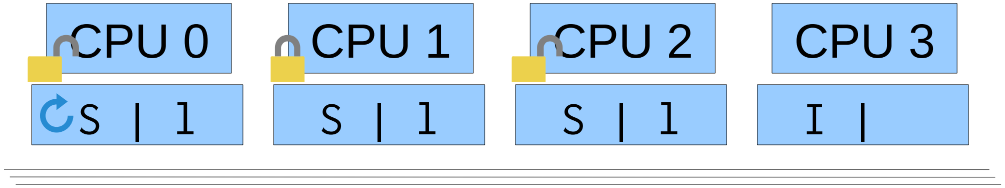


+ spins locally while lock is held by other CPU
+ like Test & Set Lock but with fewer cache bus traffic

Synchronization w/ Locks

Spin Lock (Test & Test & Set Lock)

```
void lock (lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap (l->lock, tmp);
    } while (tmp == 1);
}

void unlock (lock_t *l) {
    l->lock = 0
}
```

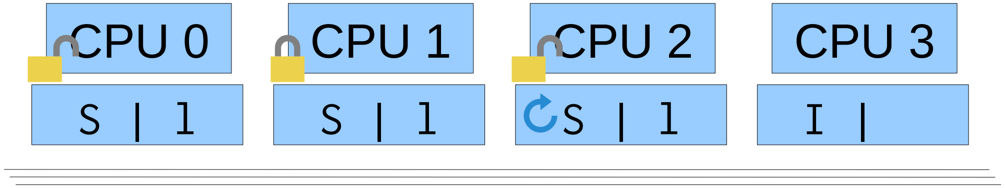


+ spins locally while lock is held by other CPU
+ like Test & Set Lock but with fewer cache bus traffic

Synchronization w/ Locks

Spin Lock (Test & Test & Set Lock)

```
void lock (lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap (l->lock, tmp);
    } while (tmp == 1);
}

void unlock (lock_t *l) {
    l->lock = 0
}
```

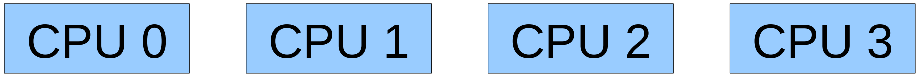


+ spins locally while lock is held by other CPU

+ like Test & Set Lock but with fewer cache bus traffic

Synchronizatio w/ Locks

Fairness – Test & Set Locks



CPU 0

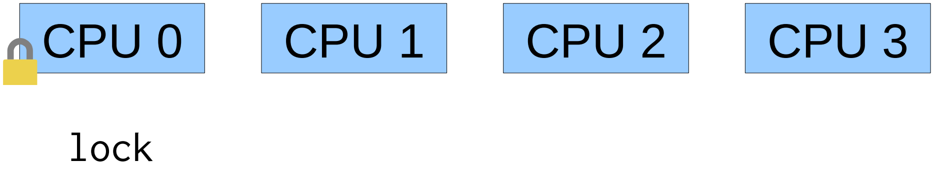
CPU 1

CPU 2

CPU 3

Synchronizatio w/ Locks

Fairness – Test & Set Locks



CPU 0

CPU 1

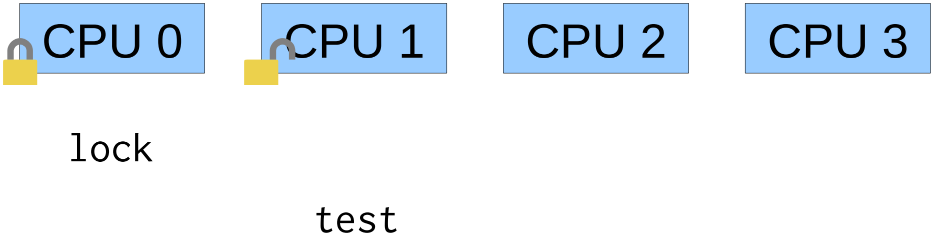
CPU 2

CPU 3

`lock`

Synchronizatio w/ Locks

Fairness – Test & Set Locks



CPU 0

CPU 1

CPU 2

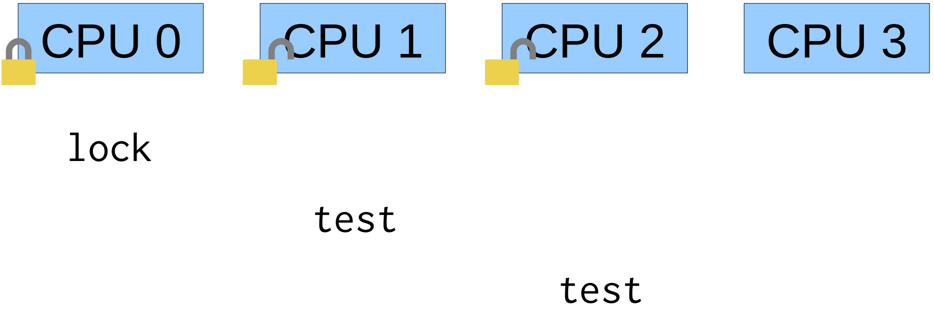
CPU 3

```
lock
```

```
test
```

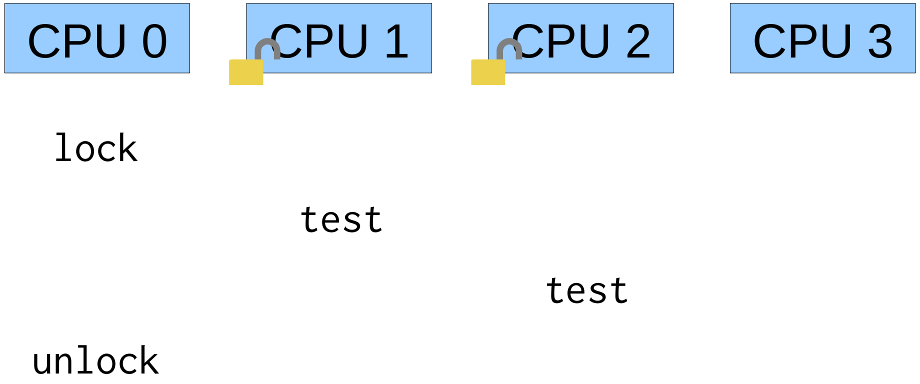
Synchronizatio w/ Locks

Fairness – Test & Set Locks



Synchronizatio w/ Locks

Fairness – Test & Set Locks

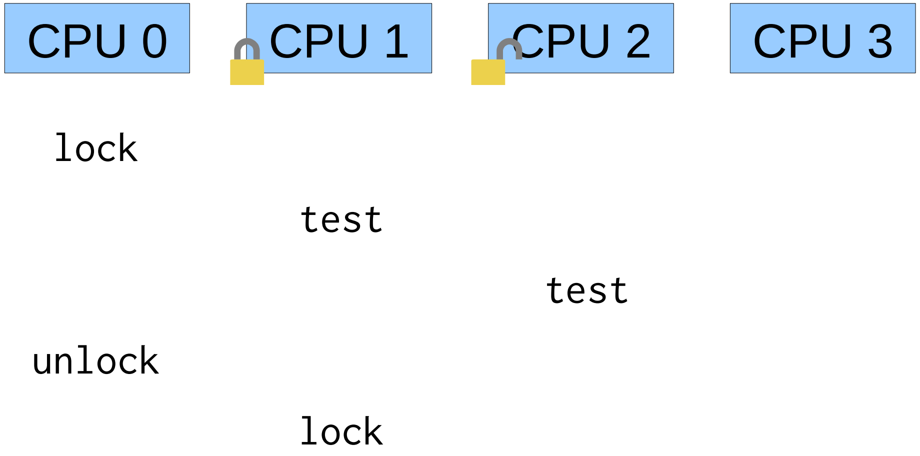


CPU 0 CPU 1 CPU 2 CPU 3

```
lock
            test
                        test
unlock
```

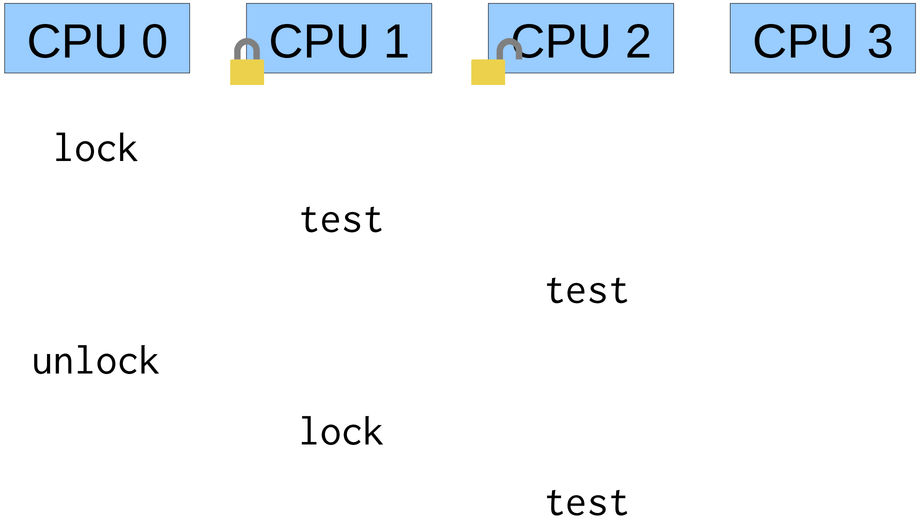
Synchronizatio w/ Locks

Fairness – Test & Set Locks



Synchronizatio w/ Locks

Fairness – Test & Set Locks



CPU 0	CPU 1	CPU 2	CPU 3
lock			
	test		
		test	
unlock			
	lock		
		test	

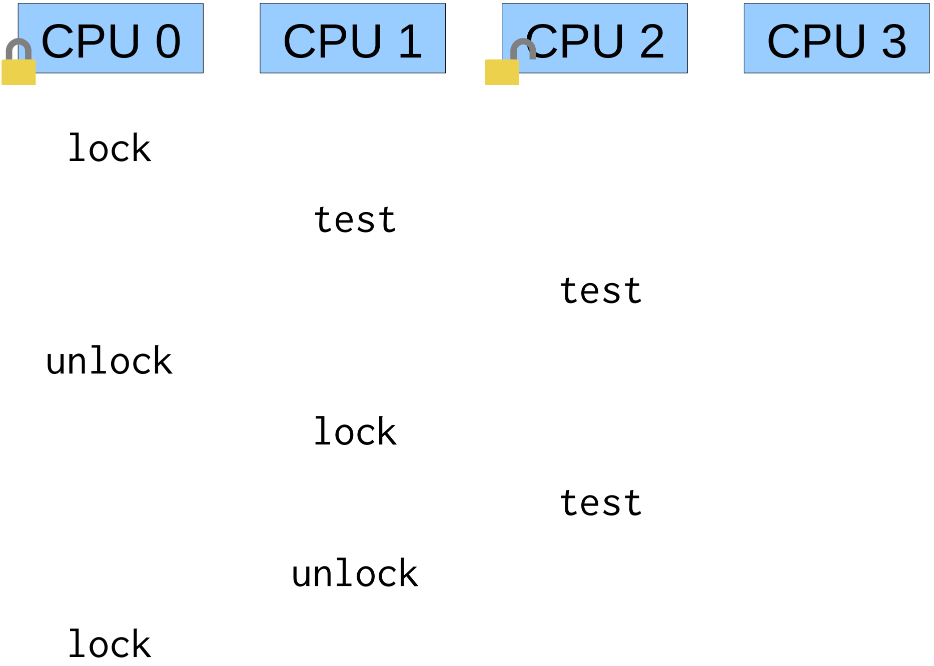
Synchronizatio w/ Locks

Fairness – Test & Set Locks

CPU 0	CPU 1	CPU 2	CPU 3
`lock`			
	`test`		
		`test`	
`unlock`			
	`lock`		
		`test`	
	`unlock`		

Synchronizatio w/ Locks

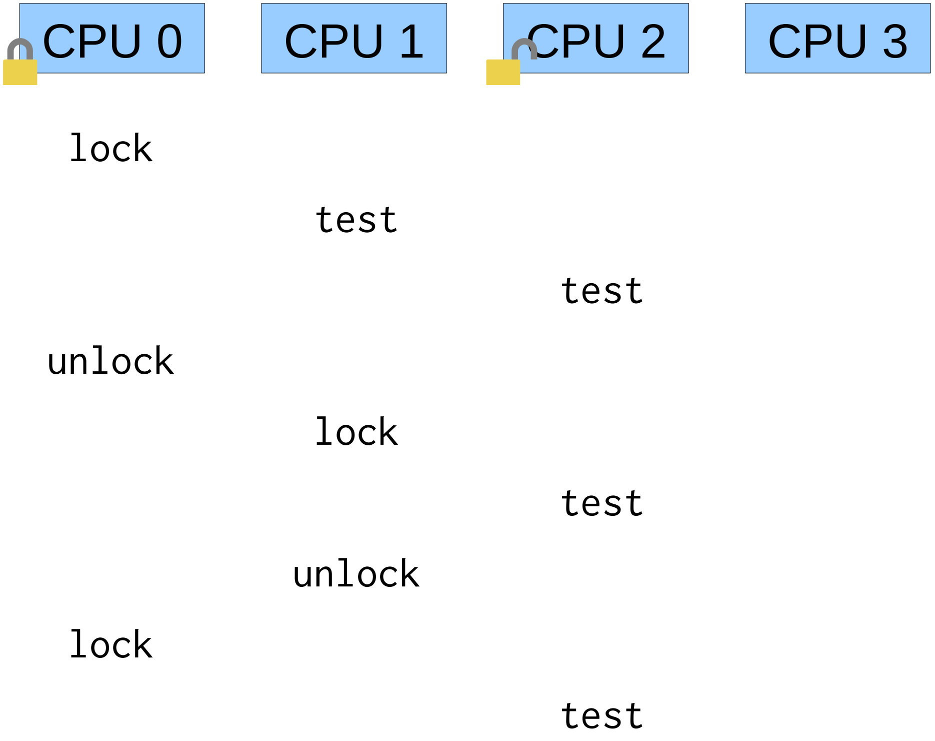
Fairness – Test & Set Locks



CPU 0	CPU 1	CPU 2	CPU 3
lock			
	test		
		test	
unlock			
	lock		
		test	
	unlock		
lock			

Synchronizatio w/ Locks

Fairness – Test & Set Locks



CPU 0	CPU 1	CPU 2	CPU 3
`lock`			
	`test`		
		`test`	
`unlock`			
	`lock`		
		`test`	
	`unlock`		
`lock`			
		`test`	

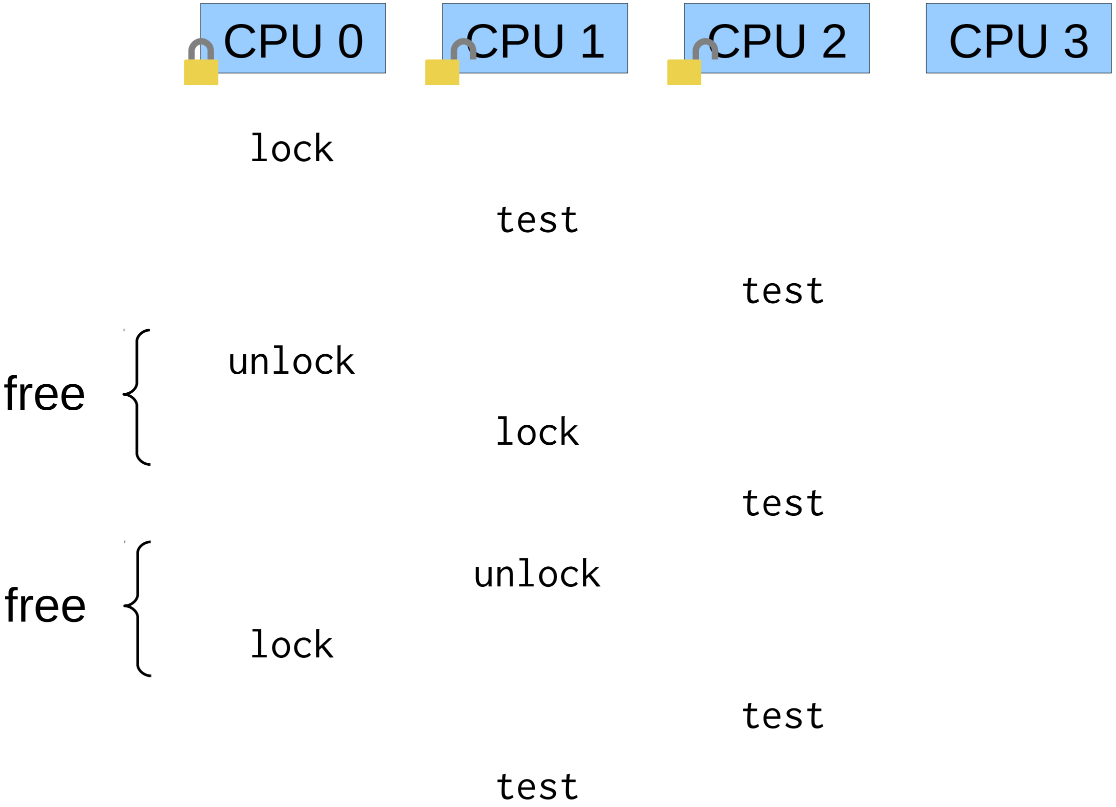
Synchronizatio w/ Locks

Fairness – Test & Set Locks

CPU 0	CPU 1	CPU 2	CPU 3
`lock`			
	`test`		
		`test`	
`unlock`			
	`lock`		
		`test`	
	`unlock`		
`lock`			
		`test`	
	`test`		

Synchronizatio w/ Locks

Fairness – Test & Set Locks



	CPU 0	CPU 1	CPU 2	CPU 3
	lock			
		test		
			test	
free	unlock			
		lock		
			test	
free		unlock		
	lock			
			test	
		test		

Although the lock was free multiple times CPU2 did not get it.

→ Test & Set Locks are not fair!

Synchronization w/ Locks

Ticket Locks

```
struct ticket_lock_t {
    int next_ticket;
    volatile int cur_ticket;
};

void lock (ticket_lock_t *l) {
    int my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket != my_ticket);
}

void unlock (ticket_lock_t *l) {
    l->cur_ticket++;
}
```

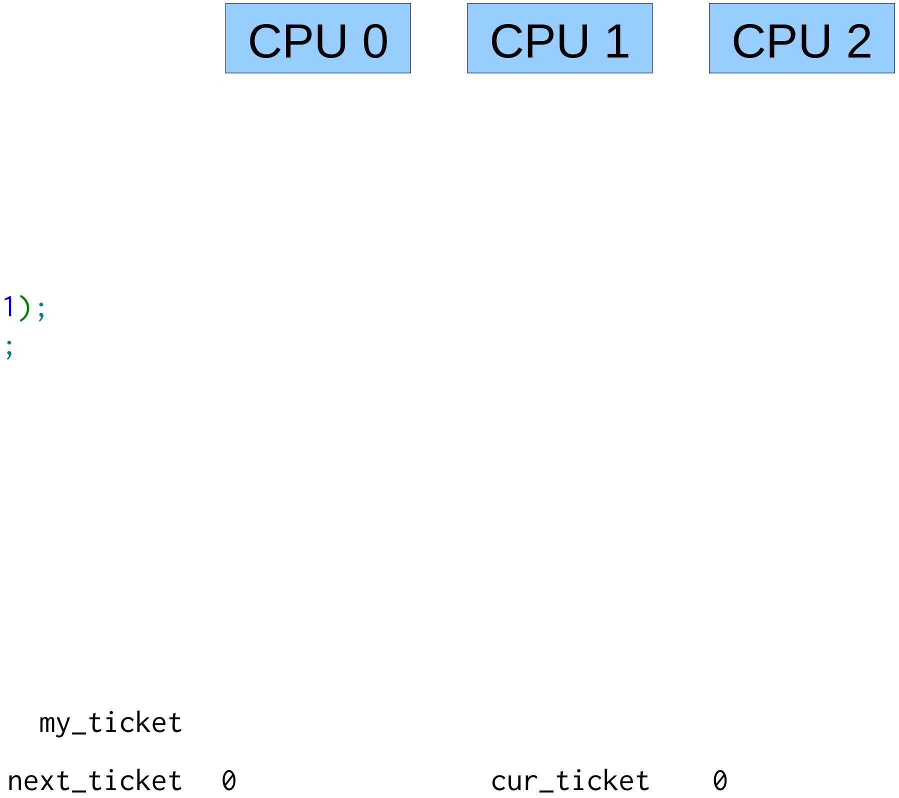
+ similarly cheap as Test & Set Lock

+ ensures fairness between threads

Synchronization w/ Locks

Ticket Locks

```
struct ticket_lock_t {
    int next_ticket;
    volatile int cur_ticket;
};

void lock (ticket_lock_t *l) {
    int my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket != my_ticket);
}

void unlock (ticket_lock_t *l) {
    l->cur_ticket++;
}
```



CPU 0 CPU 1 CPU 2

my_ticket

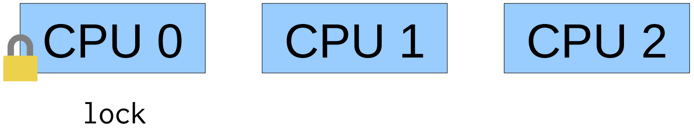
next_ticket 0

cur_ticket 0

Synchronization w/ Locks

Ticket Locks

```
struct ticket_lock_t {
    int next_ticket;
    volatile int cur_ticket;
};

void lock (ticket_lock_t *l) {
    int my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket != my_ticket);
}

void unlock (ticket_lock_t *l) {
    l->cur_ticket++;
}
```

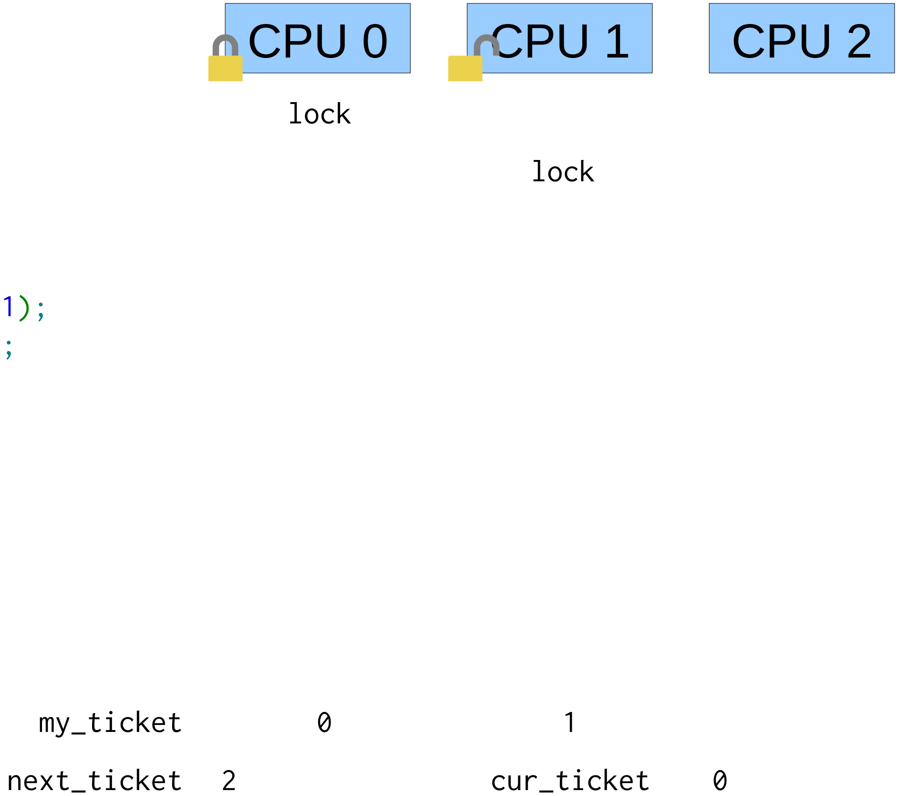


```
           my_ticket      0

         next_ticket   1                cur_ticket    0
```

Synchronization w/ Locks

Ticket Locks

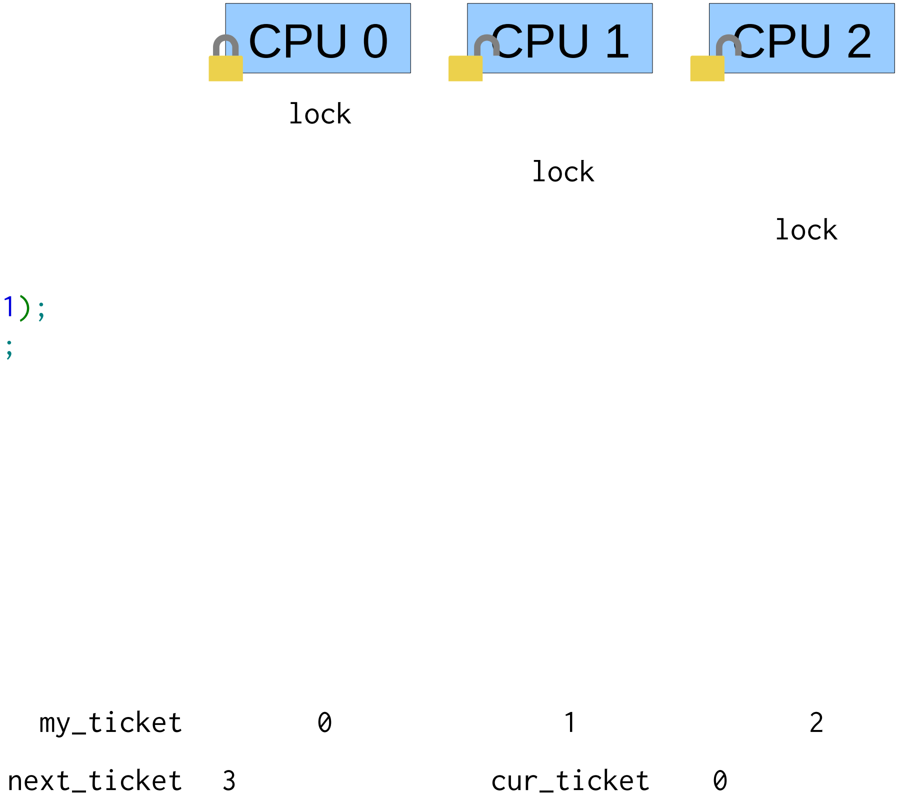
```
struct ticket_lock_t {
    int next_ticket;
    volatile int cur_ticket;
};

void lock (ticket_lock_t *l) {
    int my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket != my_ticket);
}

void unlock (ticket_lock_t *l) {
    l->cur_ticket++;
}
```



Synchronization w/ Locks

Ticket Locks

```
struct ticket_lock_t {
    int next_ticket;
    volatile int cur_ticket;
};

void lock (ticket_lock_t *l) {
    int my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket != my_ticket);
}

void unlock (ticket_lock_t *l) {
    l->cur_ticket++;
}
```



CPU 0

CPU 1

CPU 2

lock

lock

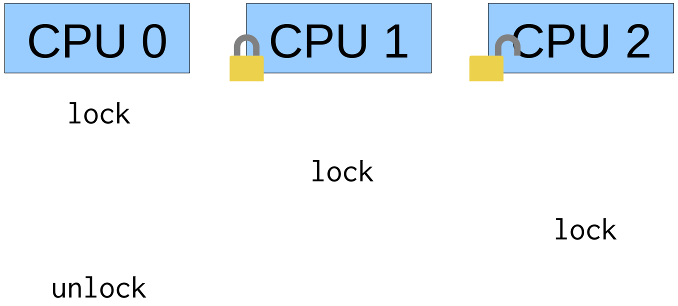
lock

```
my_ticket       0           1           2
next_ticket   3         cur_ticket    0
```

Synchronization w/ Locks

Ticket Locks

```
struct ticket_lock_t {
    int next_ticket;
    volatile int cur_ticket;
};

void lock (ticket_lock_t *l) {
    int my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket != my_ticket);
}

void unlock (ticket_lock_t *l) {
    l->cur_ticket++;
}
```



CPU 0 CPU 1 CPU 2

lock

lock

lock

unlock

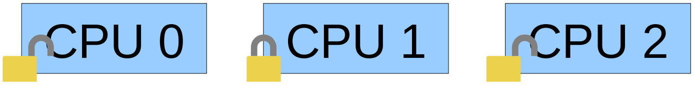
my_ticket 1 2

next_ticket 3 cur_ticket 1

Synchronization w/ Locks

Ticket Locks

```
struct ticket_lock_t {
    int next_ticket;
    volatile int cur_ticket;
};

void lock (ticket_lock_t *l) {
    int my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket != my_ticket);
}

void unlock (ticket_lock_t *l) {
    l->cur_ticket++;
}
```



CPU 0 CPU 1 CPU 2

lock

lock

lock

unlock

lock

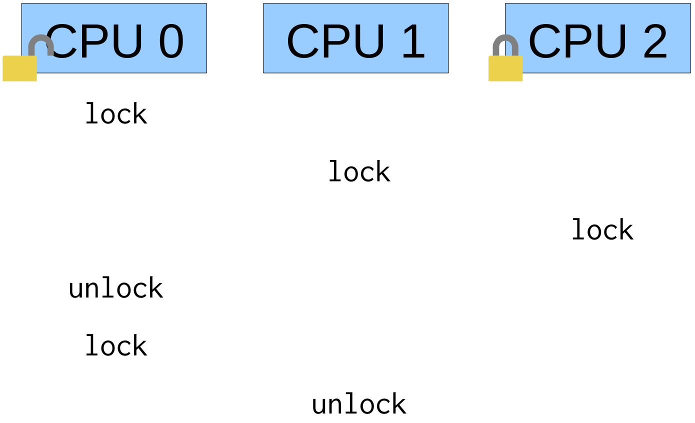
my_ticket 3 1 2

next_ticket 4 cur_ticket 1

Synchronization w/ Locks

Ticket Locks

```
struct ticket_lock_t {
    int next_ticket;
    volatile int cur_ticket;
};

void lock (ticket_lock_t *l) {
    int my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket != my_ticket);
}

void unlock (ticket_lock_t *l) {
    l->cur_ticket++;
}
```



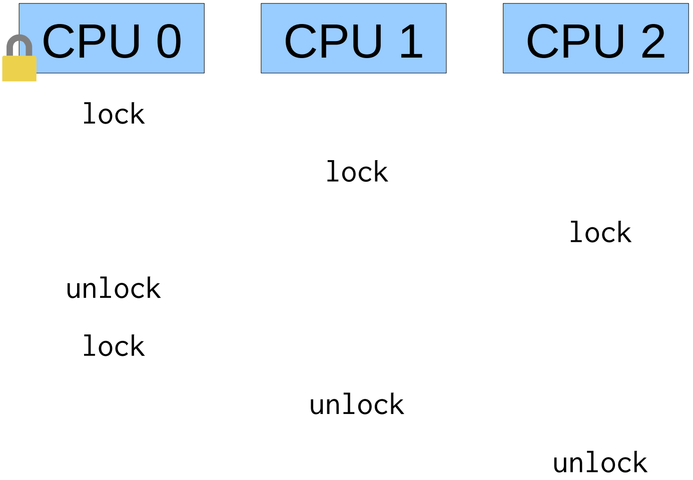
my_ticket 3 2

next_ticket 4 cur_ticket 2

Synchronization w/ Locks

Ticket Locks

```
struct ticket_lock_t {
    int next_ticket;
    volatile int cur_ticket;
};

void lock (ticket_lock_t *l) {
    int my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket != my_ticket);
}

void unlock (ticket_lock_t *l) {
    l->cur_ticket++;
}
```



CPU 0

CPU 1

CPU 2

lock

lock

lock

unlock

lock

unlock

unlock

my_ticket 3

next_ticket 4

cur_ticket 3

Synchronization w/ Locks

Ticket Locks

```
struct ticket_lock_t {
    int next_ticket;
    volatile int cur_ticket;
};

void lock (ticket_lock_t *l) {
    int my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket != my_ticket);
}

void unlock (ticket_lock_t *l) {
    l->cur_ticket++;
}
```

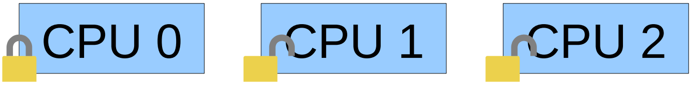
– unnecessary bus traffic on ticket increase

– abort of lock operation is difficult to implement

Synchronization w/ Locks

Ticket Locks

```
struct ticket_lock_t {
    int next_ticket;
    volatile int cur_ticket;
};

void lock (ticket_lock_t *l) {
    int my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket != my_ticket);
}

void unlock (ticket_lock_t *l) {
    l->cur_ticket++;
}
```



```
  my_ticket      0              1              2
next_ticket   3             cur_ticket    0
```

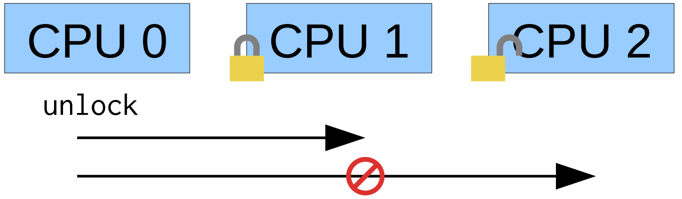
– unnecessary bus traffic on ticket increase

– abort of lock operation is difficult to implement

Synchronization w/ Locks

Ticket Locks

```
struct ticket_lock_t {
    int next_ticket;
    volatile int cur_ticket;
};

void lock (ticket_lock_t *l) {
    int my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket != my_ticket);
}

void unlock (ticket_lock_t *l) {
    l->cur_ticket++;
}
```



my_ticket 1 2

next_ticket 3 cur_ticket 1

– unnecessary bus traffic on ticket increase

– abort of lock operation is difficult to implement

Overview

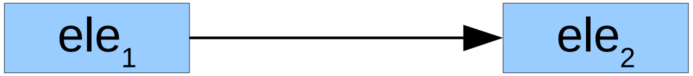
- Introduction
- Hardware Primitives
- Synchronization with Locks (Part I)
    - Properties
    - Locks
        - Spin Lock (Test & Set Lock)
        - Test & Test & Set Lock
        - Ticket Lock
- Synchronization without Locks
- Synchronization with Locks (Part II)
        - MCS Lock
        - Performance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

Synchronization w/o Locks

Lock-free Data Structures

- Single-Linked List

```
void insert(ele_t *new_ele, ele_t *prev) {
    do {
        new_ele->next = prev->next;
    } while (!cas(&(prev->next), new_ele->next, new_ele));
}
```

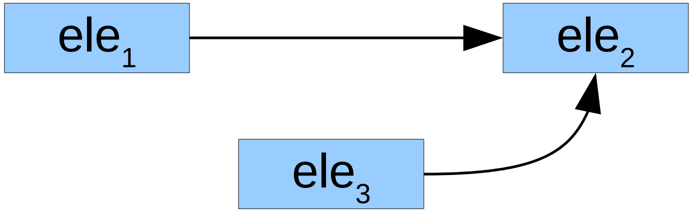


Synchronization w/o Locks

Lock-free Data Structures

- Single-Linked List

```
void insert(ele_t *new_ele, ele_t *prev) {
    do {
        new_ele->next = prev->next;
    } while (!cas(&(prev->next), new_ele->next, new_ele));
}
```

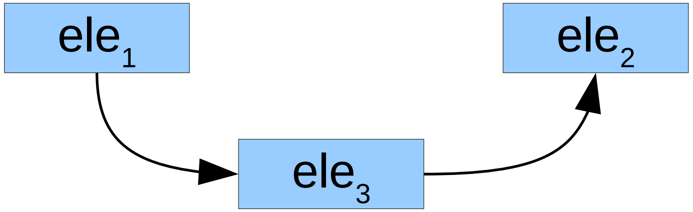


Synchronization w/o Locks

Lock-free Data Structures

- Single-Linked List

```
void insert(ele_t *new_ele, ele_t *prev) {
    do {
        new_ele->next = prev->next;
    } while (!cas(&(prev->next), new_ele->next, new_ele));
}
```



Synchronization w/o Locks

Lock-free Data Structures

- Single-Linked List

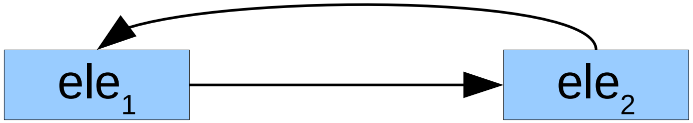
```
void insert(ele_t *new_ele, ele_t *prev) {
    do {
        load_linked(prev->next);
        new_ele->next = prev->next
    } while (!store_conditional(&(prev->next), new_ele);
}
```

Synchronization w/o Locks

Lock-free Data Structures

- Single-Linked List
- Double-Linked List

```
void insert(ele_t *new_ele, ele_t *prev) {
    do {
        auto next = prev->next;
        new_ele->next = next;
        new_ele->prev = prev;
    } while (!dcas(&(prev->next), &(next->prev),
                   new_ele->next, new_ele->prev,
                   new_ele, new_ele);
}
```

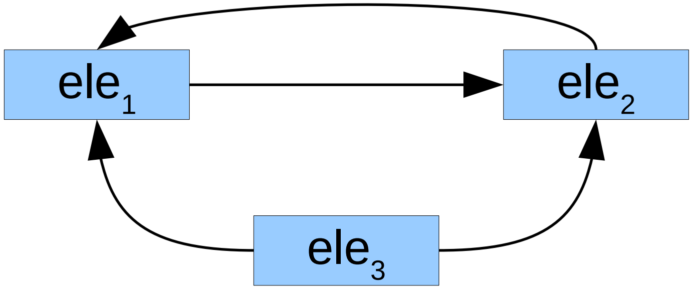


Synchronization w/o Locks

Lock-free Data Structures

- Single-Linked List
- Double-Linked List

```
void insert(ele_t *new_ele, ele_t *prev) {
    do {
        auto next = prev->next;
        new_ele->next = next;
        new_ele->prev = prev;
    } while (!dcas(&(prev->next), &(next->prev),
                   new_ele->next, new_ele->prev,
                   new_ele, new_ele);
}
```

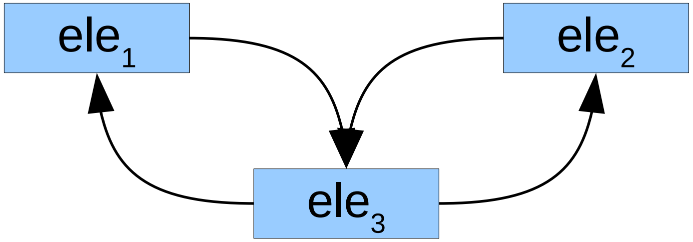


Synchronization w/o Locks

Lock-free Data Structures

- Single-Linked List
- Double-Linked List

```
void insert(ele_t *new_ele, ele_t *prev) {
    do {
        auto next = prev->next;
        new_ele->next = next;
        new_ele->prev = prev;
    } while (!dcas(&(prev->next), &(next->prev),
                   new_ele->next, new_ele->prev,
                   new_ele, new_ele);
}
```



Synchronization w/o Locks

Lock-free Data Structures

- Single-Linked List
- Double-Linked List
- Binary Trees
- ...

Not using locks does not solve all problems of locks!

e.g. Fairness → Wait-free Data Structures

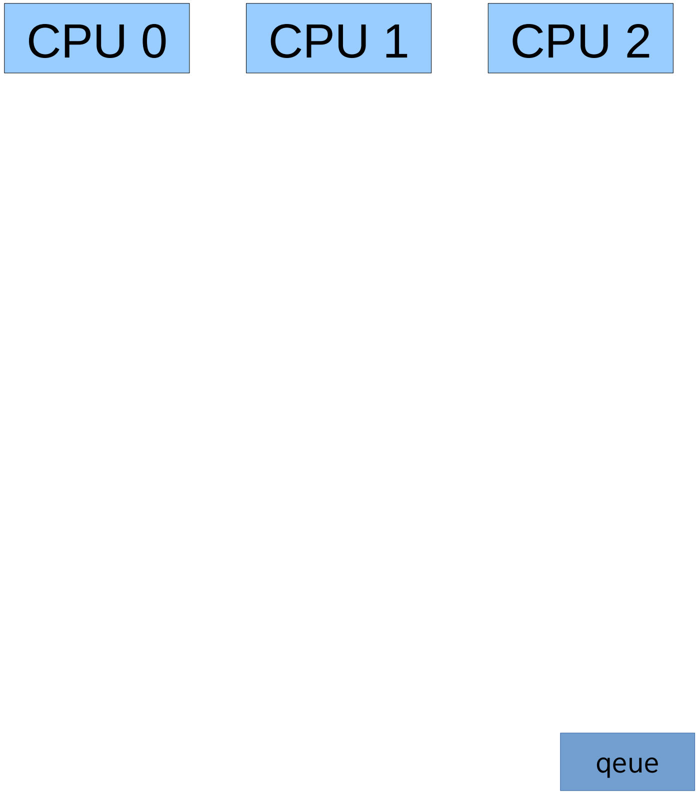
Overview

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

Synchronization w/ Locks

MCS-Lock – fair local spinning lock – Mellor-Crummey and Scott

```
struct mcs_node_t {          struct mcs_lock_t {
    mcs_node_t* next;            mcs_node_t* queue;
    bool free;               };
};


void mcs_lock(mcs_lock_t* l, mcs_node_t* cur) {
    cur->next = NULL;
    cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}


void mcs_unlock(mcs_lock_t* l, mcs_node_t* cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```



Synchronization w/ Locks

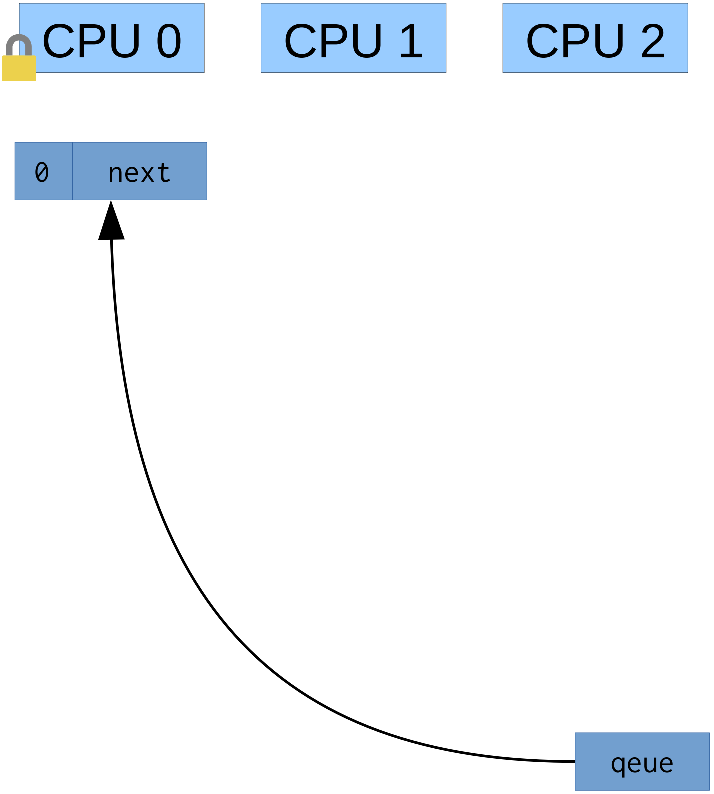
MCS-Lock – fair local spinning lock – Mellor-Crummey and Scott

```
struct mcs_node_t {
    mcs_node_t* next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t* queue;
};
```

```
void mcs_lock(mcs_lock_t* l, mcs_node_t* cur) {
    cur->next = NULL;
    cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
```

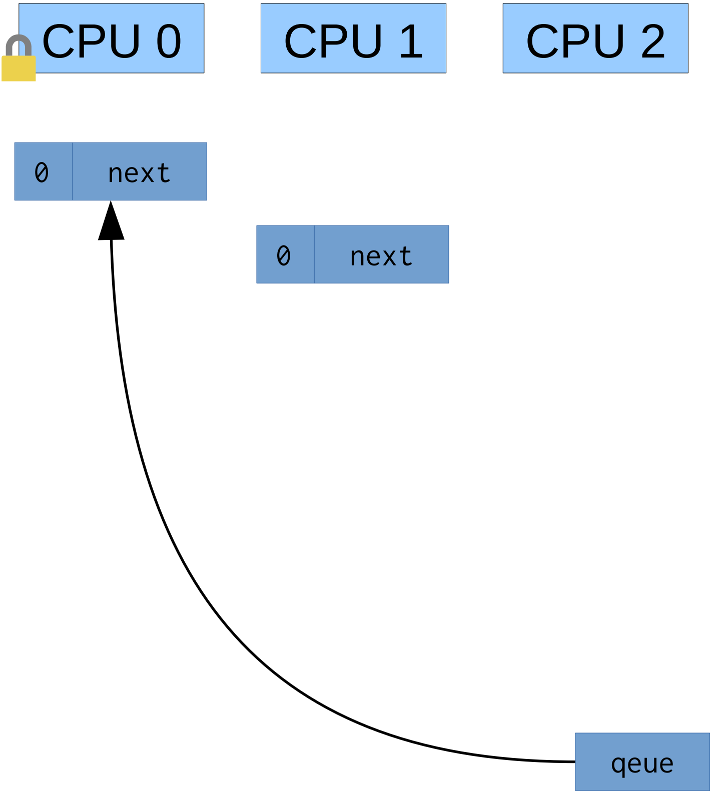
```
void mcs_unlock(mcs_lock_t* l, mcs_node_t* cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```



Synchronization w/ Locks

MCS-Lock – fair local spinning lock – Mellor-Crummey and Scott

```
struct mcs_node_t {          struct mcs_lock_t {
    mcs_node_t* next;            mcs_node_t* queue;
    bool free;               };
};


void mcs_lock(mcs_lock_t* l, mcs_node_t* cur) {
    cur->next = NULL;
    cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}


void mcs_unlock(mcs_lock_t* l, mcs_node_t* cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```



Synchronization w/ Locks

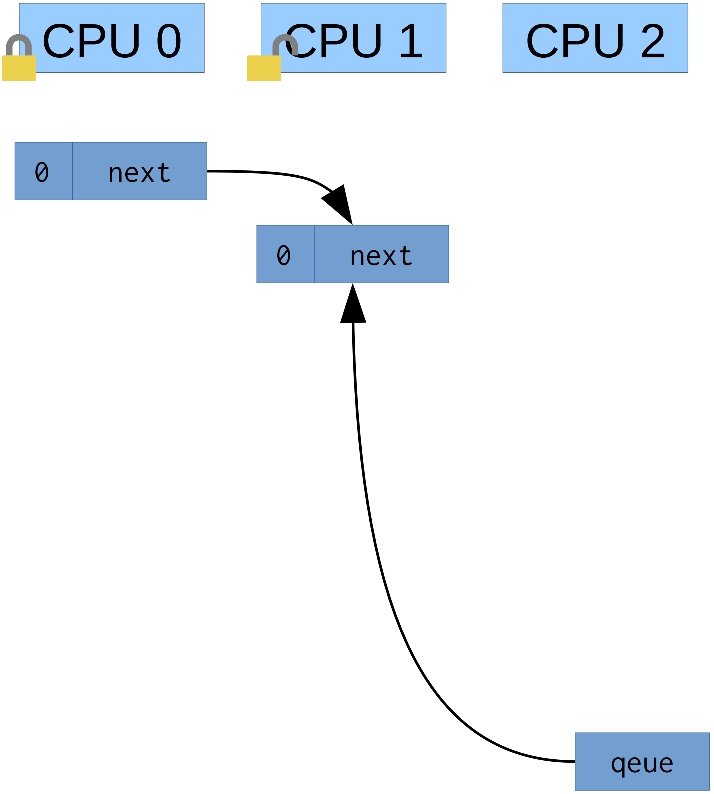
MCS-Lock – fair local spinning lock – Mellor-Crummey and Scott

```
struct mcs_node_t {
    mcs_node_t* next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t* queue;
};
```

```
void mcs_lock(mcs_lock_t* l, mcs_node_t* cur) {
    cur->next = NULL;
    cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
```

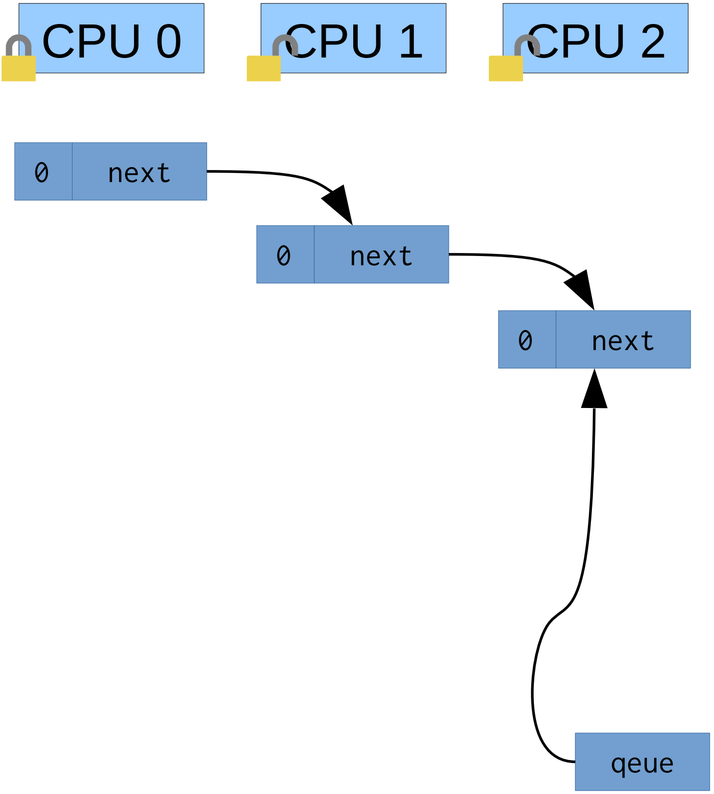
```
void mcs_unlock(mcs_lock_t* l, mcs_node_t* cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```



Synchronization w/ Locks

MCS-Lock – fair local spinning lock – Mellor-Crummey and Scott

```
struct mcs_node_t {          struct mcs_lock_t {
    mcs_node_t* next;            mcs_node_t* queue;
    bool free;               };
};


void mcs_lock(mcs_lock_t* l, mcs_node_t* cur) {
    cur->next = NULL;
    cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}


void mcs_unlock(mcs_lock_t* l, mcs_node_t* cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```



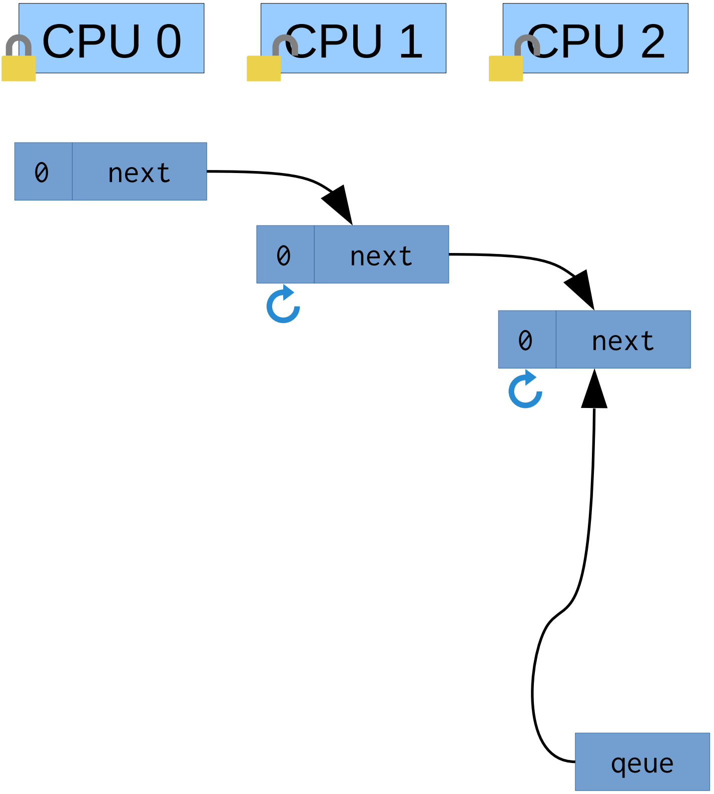
Synchronization w/ Locks

MCS-Lock – fair local spinning lock – Mellor-Crummey and Scott

```
struct mcs_node_t {            struct mcs_lock_t {
    mcs_node_t* next;              mcs_node_t* queue;
    bool free;                 };
};
```

```
void mcs_lock(mcs_lock_t* l, mcs_node_t* cur) {
    cur->next = NULL;
    cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
```

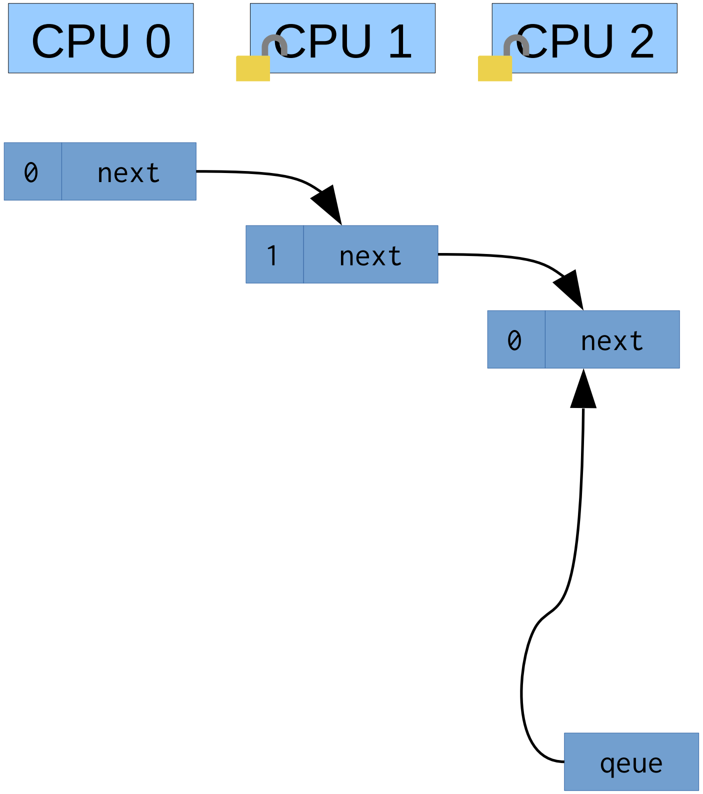
```
void mcs_unlock(mcs_lock_t* l, mcs_node_t* cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```



Synchronization w/ Locks

MCS-Lock – fair local spinning lock – Mellor-Crummey and Scott

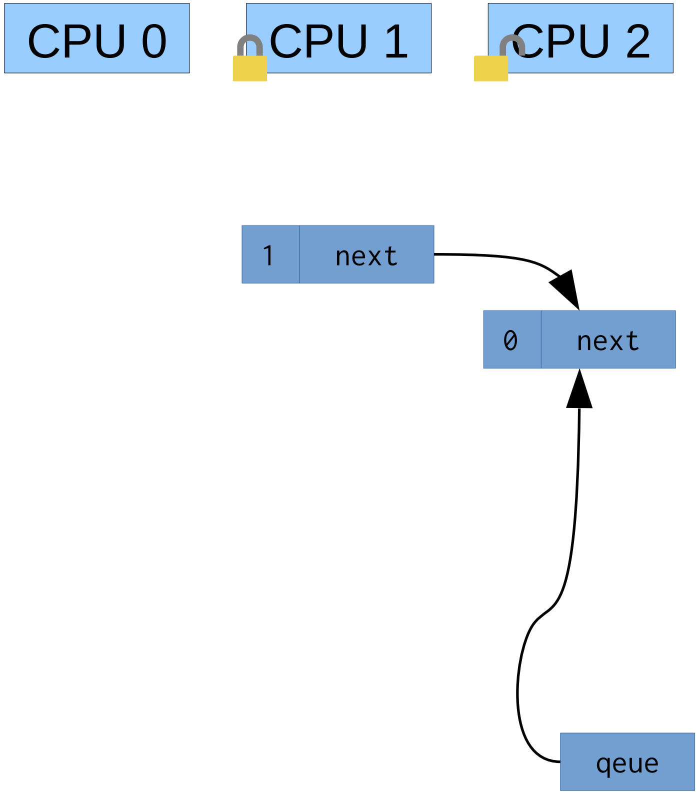
```
struct mcs_node_t {            struct mcs_lock_t {
    mcs_node_t* next;              mcs_node_t* queue;
    bool free;                 };
};


void mcs_lock(mcs_lock_t* l, mcs_node_t* cur) {
    cur->next = NULL;
    cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}


void mcs_unlock(mcs_lock_t* l, mcs_node_t* cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```



Synchronization w/ Locks

MCS-Lock – fair local spinning lock – Mellor-Crummey and Scott

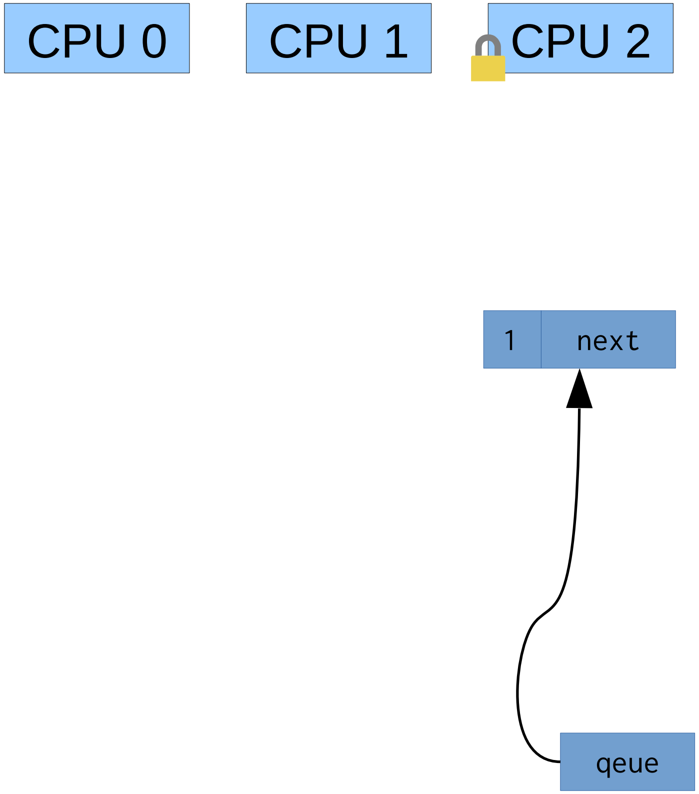
```
struct mcs_node_t {            struct mcs_lock_t {
    mcs_node_t* next;              mcs_node_t* queue;
    bool free;                 };
};


void mcs_lock(mcs_lock_t* l, mcs_node_t* cur) {
    cur->next = NULL;
    cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}


void mcs_unlock(mcs_lock_t* l, mcs_node_t* cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```



Synchronization w/ Locks

MCS-Lock – fair local spinning lock – Mellor-Crummey and Scott

```
struct mcs_node_t {          struct mcs_lock_t {
    mcs_node_t* next;            mcs_node_t* queue;
    bool free;               };
};


void mcs_lock(mcs_lock_t* l, mcs_node_t* cur) {
    cur->next = NULL;
    cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}


void mcs_unlock(mcs_lock_t* l, mcs_node_t* cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```



Synchronization w/ Locks

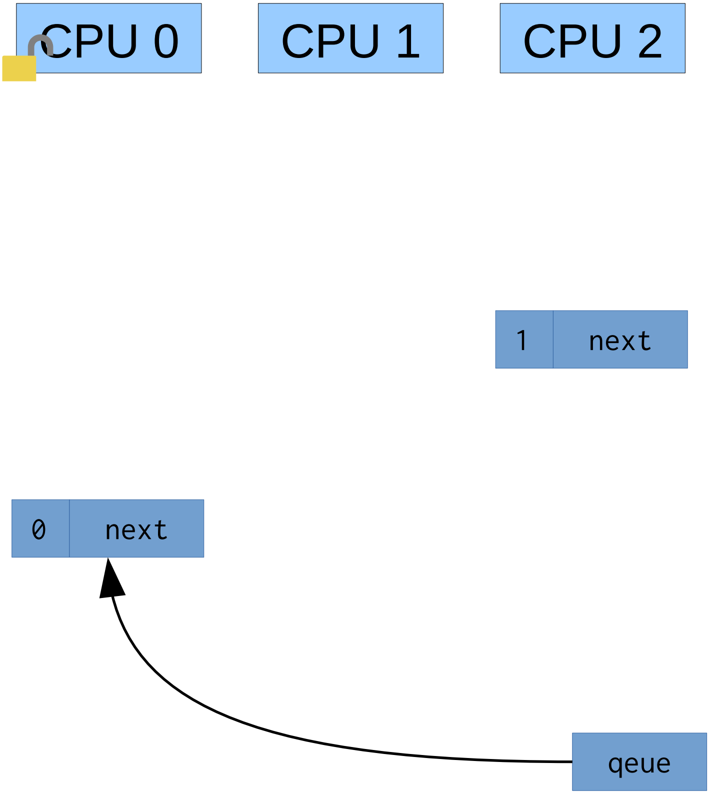
MCS-Lock – fair local spinning lock – Mellor-Crummey and Scott

```
struct mcs_node_t {
    mcs_node_t* next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t* queue;
};
```

```
void mcs_lock(mcs_lock_t* l, mcs_node_t* cur) {
    cur->next = NULL;
    cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
```

```
void mcs_unlock(mcs_lock_t* l, mcs_node_t* cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```



Synchronization w/ Locks

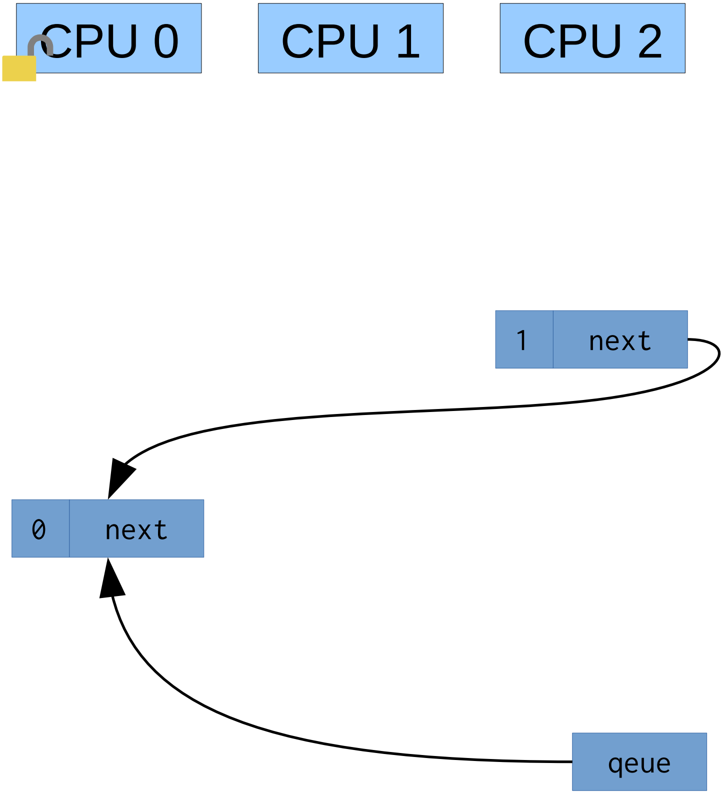
MCS-Lock – fair local spinning lock – Mellor-Crummey and Scott

```
struct mcs_node_t {
    mcs_node_t* next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t* queue;
};
```

```
void mcs_lock(mcs_lock_t* l, mcs_node_t* cur) {
    cur->next = NULL;
    cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
```

```
void mcs_unlock(mcs_lock_t* l, mcs_node_t* cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```



TECHNISCHE UNIVERSITÄT DRESDEN

Synchronization w/ Locks

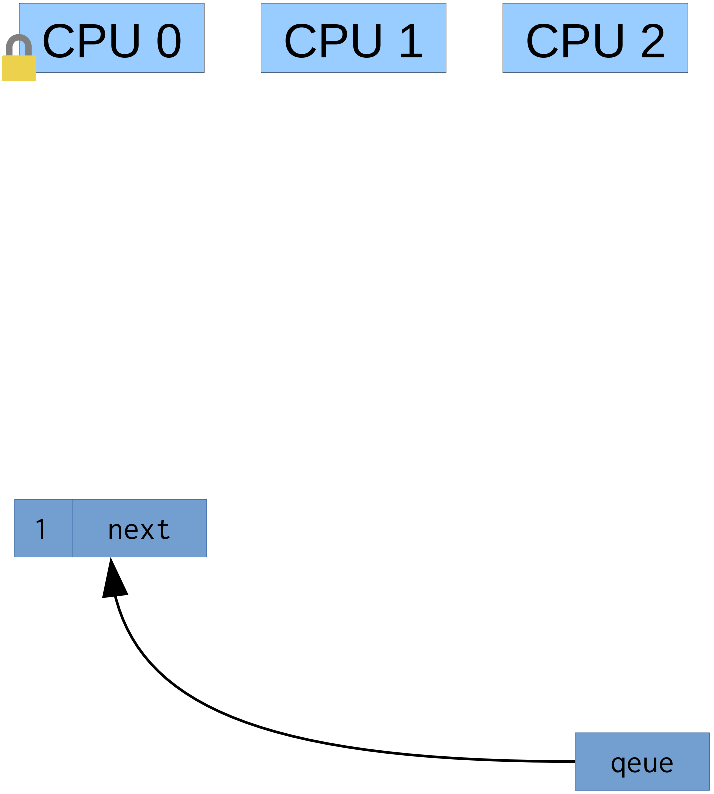
MCS-Lock – fair local spinning lock – Mellor-Crummey and Scott

```
struct mcs_node_t {
    mcs_node_t* next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t* queue;
};
```

```
void mcs_lock(mcs_lock_t* l, mcs_node_t* cur) {
    cur->next = NULL;
    cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
```

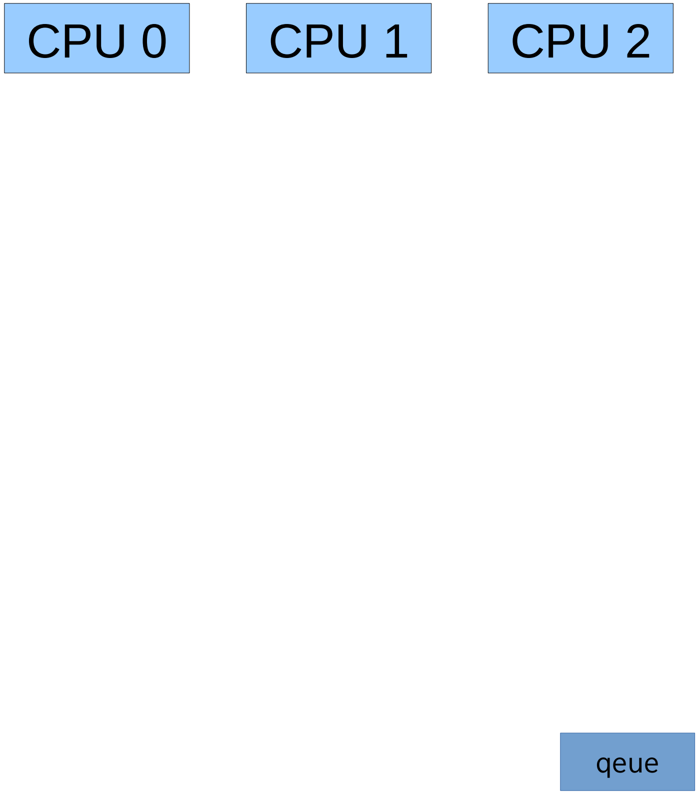
```
void mcs_unlock(mcs_lock_t* l, mcs_node_t* cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```



Synchronization w/ Locks

MCS-Lock – fair local spinning lock – Mellor-Crummey and Scott

```
struct mcs_node_t {          struct mcs_lock_t {
    mcs_node_t* next;            mcs_node_t* queue;
    bool free;               };
};


void mcs_lock(mcs_lock_t* l, mcs_node_t* cur) {
    cur->next = NULL;
    cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}


void mcs_unlock(mcs_lock_t* l, mcs_node_t* cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```



CPU 0

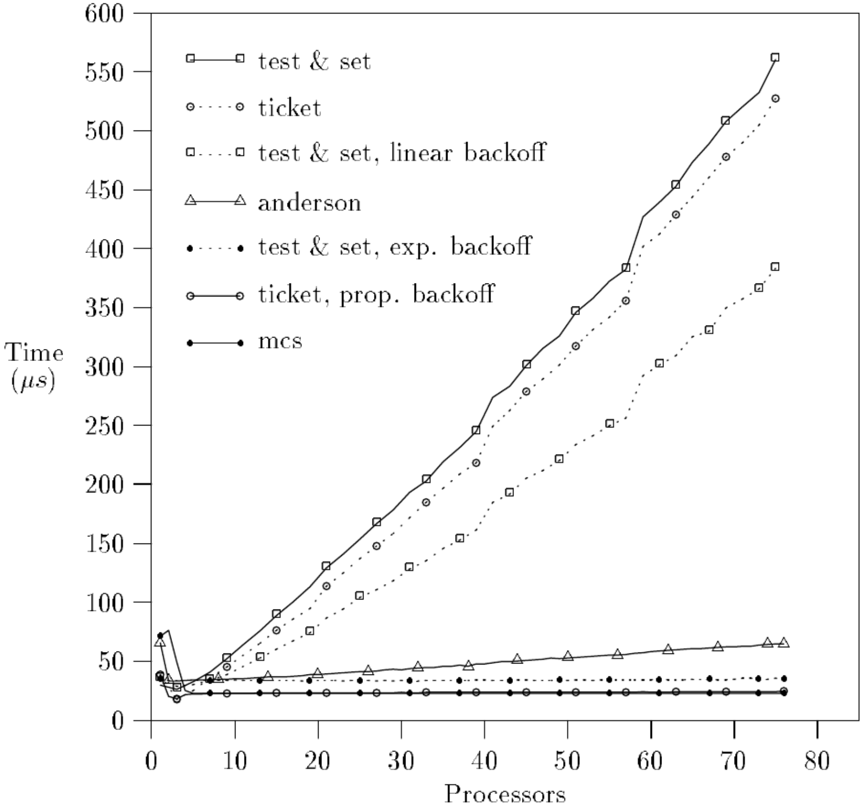
CPU 1

CPU 2

qeue

Synchronization w/ Locks

MCS-Lock – Performance



<u>Figure 1</u> Comparisson of diffirent lock implementations.

Mellor-Crummey, Scott [1991]: “Algorithms for Scalable Synchronization on Shared Memory Multiprocessors”

Synchronization w/ Locks

MCS-Lock – Performance

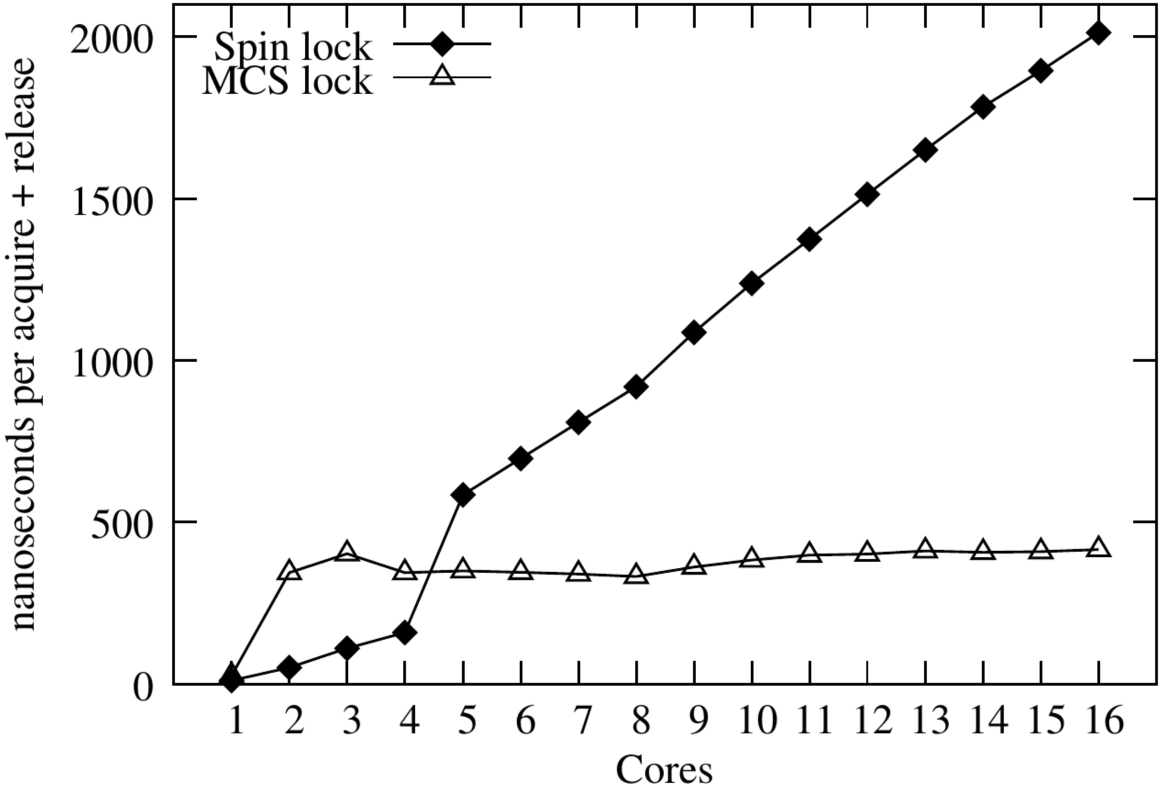


Figure 2 Comparisson of the overhead of spin-locks and MCS-locks on an 16 core AMD Opteron.
Boyd-Wickizer et al. [2008]: “Corey: An Operating System for Many Cores”

Overview

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

Synchronization w/ Locks

Timeouts – Abort lock()-Operation

- Give up locking after a specified timeout
- Stop threads which are currently waiting for a lock

Synchronization w/ Locks

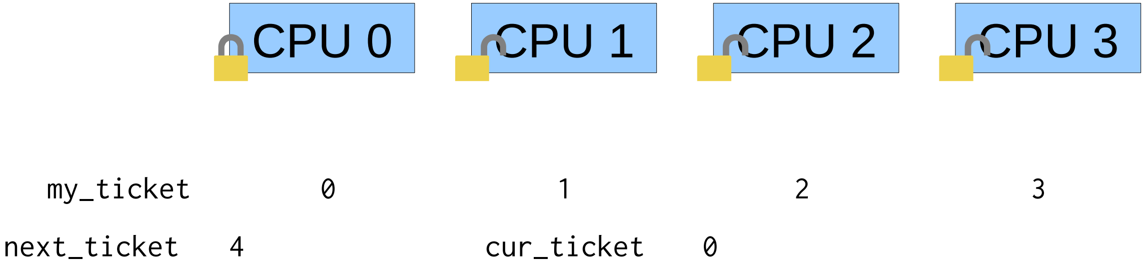
Timeouts – Abort lock()-Operation

- Give up locking after a specified timeout
- Stop threads which are currently waiting for a lock
  - Test & Set Locks → stop trying to acquire lock

Synchronization w/ Locks

Timeouts – Abort lock()-Operation

- Give up locking after a specified timeout
- Stop threads which are currently waiting for a lock
  - Test & Set Locks
  - Ticket Lock

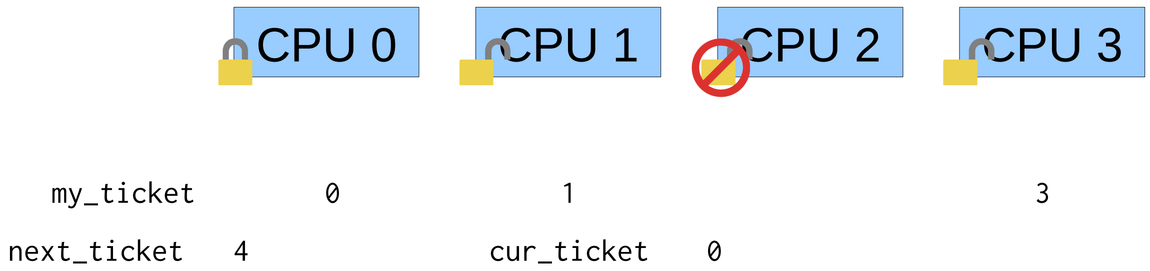


```
my_ticket         0          1          2          3
next_ticket   4        cur_ticket   0
```

Synchronization w/ Locks

Timeouts – Abort lock()-Operation

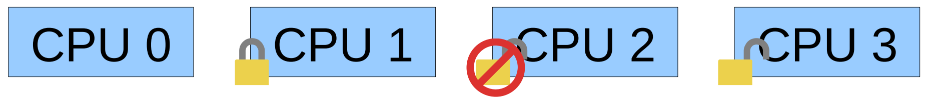
- Give up locking after a specified timeout
- Stop threads which are currently waiting for a lock
  - Test & Set Locks
  - Ticket Lock → stop trying to acquire lock



Synchronization w/ Locks

Timeouts – Abort lock()-Operation

- Give up locking after a specified timeout
- Stop threads which are currently waiting for a lock
  - Test & Set Locks
  - Ticket Lock → stop trying to acquire lock

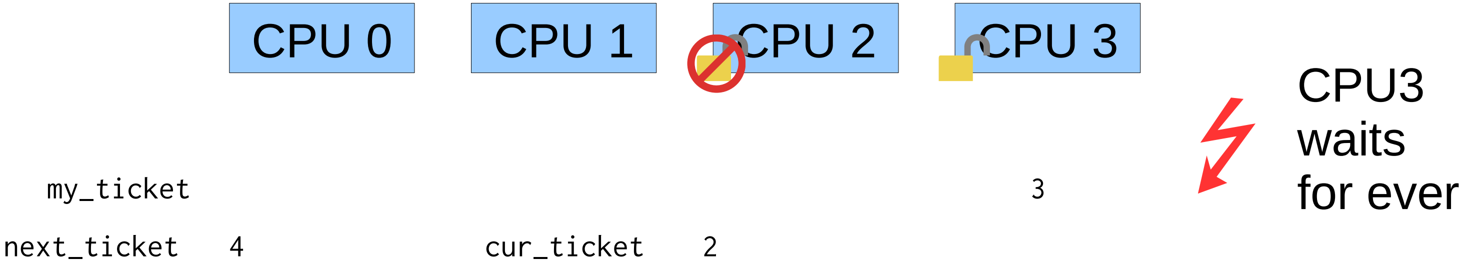


```
my_ticket                 1                        3
next_ticket    4           cur_ticket    1
```

Synchronization w/ Locks

Timeouts – Abort lock()-Operation

- Give up locking after a specified timeout
- Stop threads which are currently waiting for a lock
  - Test & Set Locks
  - Ticket Lock → stop trying to acquire lock

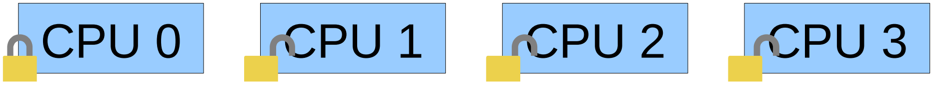


Synchronization w/ Locks

Timeouts – Abort lock()-Operation

- Give up locking after a specified timeout
- Stop threads which are currently waiting for a lock

  - Test & Set Locks
  - Ticket Lock → stop trying to acquire lock + increase `cur_ticket`



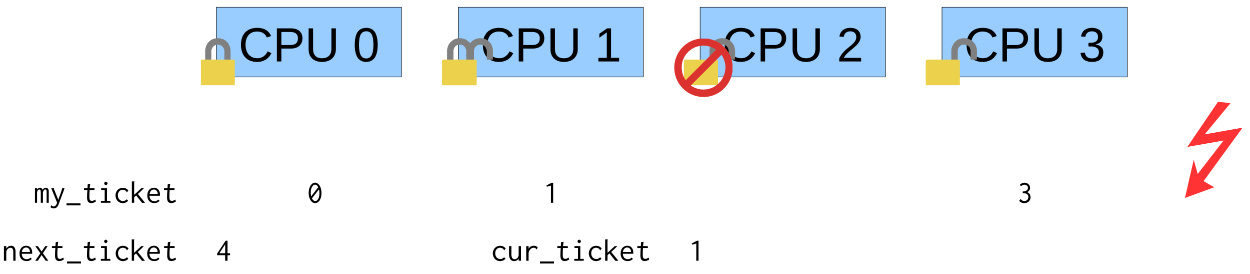
CPU 0 CPU 1 CPU 2 CPU 3

`my_ticket`		0	1	2	3
`next_ticket`	4	`cur_ticket`	0		

Synchronization w/ Locks

Timeouts – Abort lock()-Operation

- Give up locking after a specified timeout
- Stop threads which are currently waiting for a lock
    - Test & Set Locks
    - Ticket Lock → stop trying to acquire lock + increase `cur_ticket`



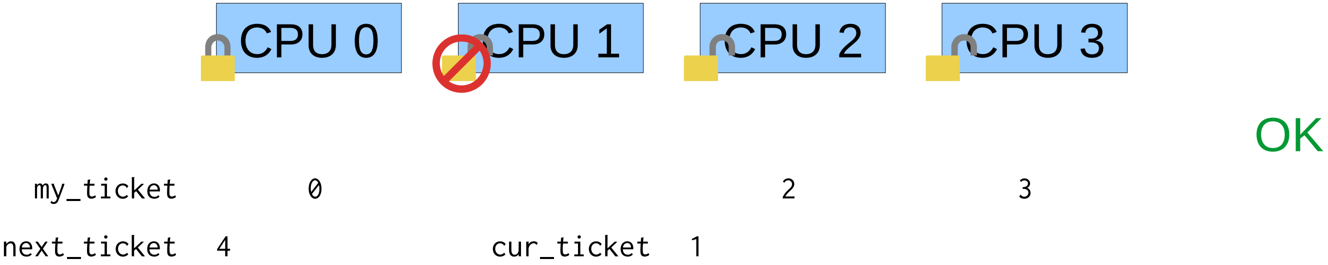
CPU1 can enter CS

Synchronization w/ Locks

Timeouts – Abort lock()-Operation

- Give up locking after a specified timeout
- Stop threads which are currently waiting for a lock

  - Test & Set Locks
  - Ticket Lock → stop trying to acquire the lock + increase `cur_ticket`



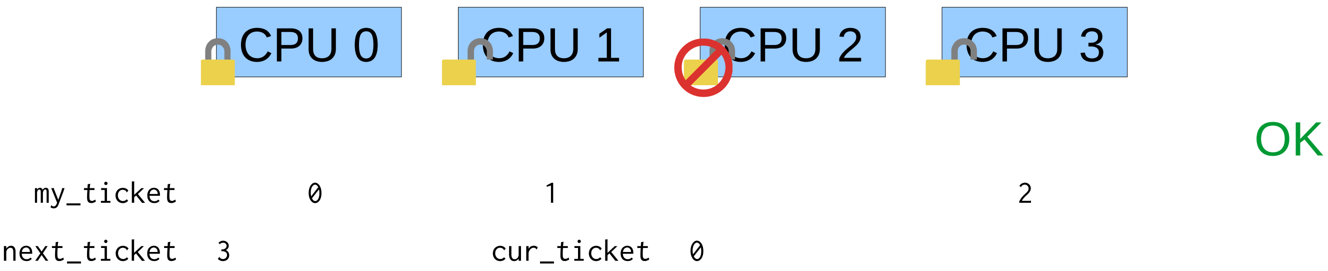
```
my_ticket      0                                2          3
next_ticket  4            cur_ticket  1
```

Synchronization w/ Locks

Timeouts – Abort lock()-Operation

- Give up locking after a specified timeout
- Stop threads which are currently waiting for a lock

  - Test & Set Locks
  - Ticket Lock → stop trying to acquire the lock + alter `next_ticket` and `my_ticket`



CPU 0 CPU 1 CPU 2 CPU 3

OK

```
my_ticket      0        1                   2
next_ticket  3        cur_ticket  0
```

Very tricky to implement!

Synchronization w/ Locks

Timeouts – Abort lock()-Operation

- Give up locking after a specified timeout
- Stop threads which are currently waiting for a lock

  - Test & Set Locks → stop trying to acquire the lock
  - Ticket Lock → stop trying to acquire the lock + alter `next_ticket` and `my_ticket`
  - MCS-Lock → dequeue from the queue of waiters (exercise)

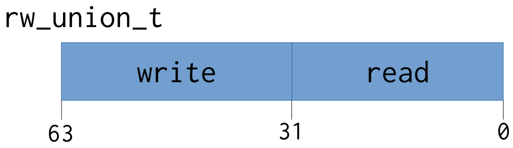
Synchronization w/ Locks

Reader Writer Locks

- Lock differentiates two types of lock holders:
  - Readers
    - Do not modify the object
    - Multiple can use the object at the same time
  - Writers
    - Modify the object
    - Must have exclusive access to the object (no other readers or writers)
- Locks can have different level of fairness
  - Readers and writers use the object in the order they appear → fair
  - Later readers overtake earlier writers → unfair for writers
  - Later writers overtake earlier readers → unfair for readers

Synchronization w/ Locks

Fair Ticket Reader Writer Lock

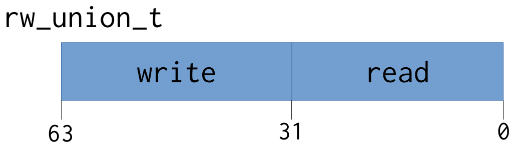


```
struct rw_lock_t {
    rw_union_t cur_ticket;
    rw_union_t next_ticket;
};

void lock_read(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket.write != my_ticket.write);
}

void lock_write(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket.write), 1);
    do {} while (l→cur_ticket != my_ticket);
}

void unlock_read(rw_lock_t *l) {
    xadd(&(l->cur_ticket.read), 1);
}

void unlock_write(rw_lock_t *l) {
    l->cur_ticket.write++;
}
```

Synchronization w/ Locks

Fair Ticket Reader Writer Lock

```
struct rw_lock_t {
    rw_union_t cur_ticket;
    rw_union_t next_ticket;
};

void lock_read(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket.write != my_ticket.write);
}

void lock_write(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket.write), 1);
    do {} while (l→cur_ticket != my_ticket);
}

void unlock_read(rw_lock_t *l) {
    xadd(&(l->cur_ticket.read), 1);
}

void unlock_write(rw_lock_t *l) {
    l->cur_ticket.write++;
}
```



CPU 0

CPU 1

CPU 2

my_ticket

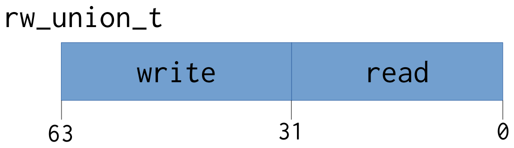
next_ticket 0|0

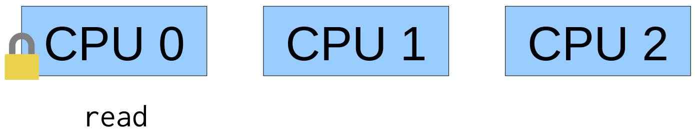
cur_ticket 0|0

Synchronization w/ Locks

Fair Ticket Reader Writer Lock

```
struct rw_lock_t {
    rw_union_t cur_ticket;
    rw_union_t next_ticket;
};

void lock_read(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket.write != my_ticket.write);
}

void lock_write(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket.write), 1);
    do {} while (l→cur_ticket != my_ticket);
}

void unlock_read(rw_lock_t *l) {
    xadd(&(l->cur_ticket.read), 1);
}

void unlock_write(rw_lock_t *l) {
    l->cur_ticket.write++;
}
```



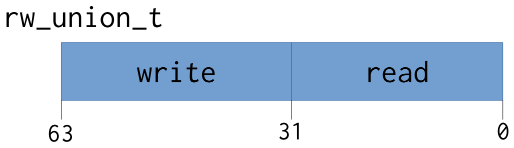


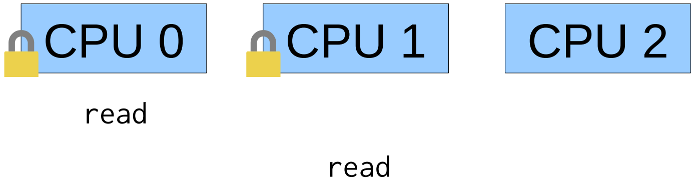
```
                 my_ticket       0|0
               next_ticket   0|1              cur_ticket     0|0
```

Synchronization w/ Locks

Fair Ticket Reader Writer Lock

```
struct rw_lock_t {
    rw_union_t cur_ticket;
    rw_union_t next_ticket;
};

void lock_read(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket.write != my_ticket.write);
}

void lock_write(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket.write), 1);
    do {} while (l→cur_ticket != my_ticket);
}

void unlock_read(rw_lock_t *l) {
    xadd(&(l->cur_ticket.read), 1);
}

void unlock_write(rw_lock_t *l) {
    l->cur_ticket.write++;
}
```





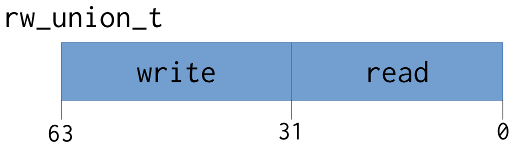
my_ticket 0|0 0|1

next_ticket 0|2 cur_ticket 0|0

Synchronization w/ Locks

Fair Ticket Reader Writer Lock

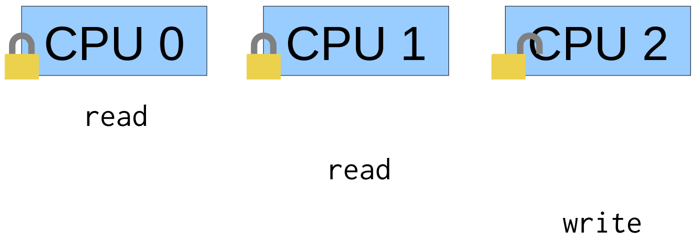
```
struct rw_lock_t {
    rw_union_t cur_ticket;
    rw_union_t next_ticket;
};

void lock_read(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket.write != my_ticket.write);
}

void lock_write(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket.write), 1);
    do {} while (l→cur_ticket != my_ticket);
}

void unlock_read(rw_lock_t *l) {
    xadd(&(l->cur_ticket.read), 1);
}

void unlock_write(rw_lock_t *l) {
    l->cur_ticket.write++;
}
```



rw_union_t

write	read

63 31 0



read

read

write

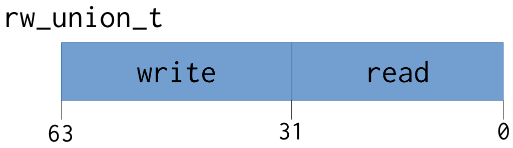
my_ticket 0|0 0|1 0|2

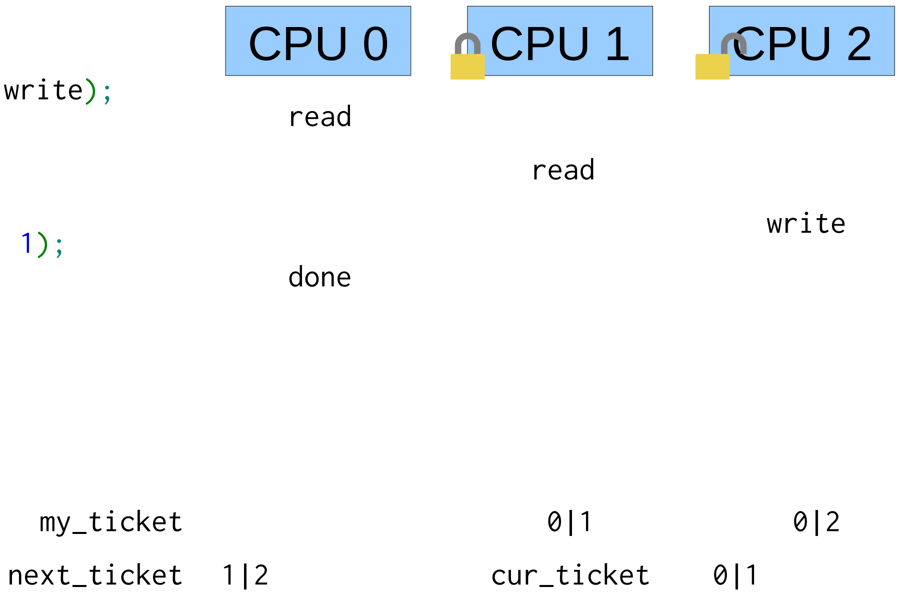
next_ticket 1|2 cur_ticket 0|0

Synchronization w/ Locks

Fair Ticket Reader Writer Lock

```
struct rw_lock_t {
    rw_union_t cur_ticket;
    rw_union_t next_ticket;
};

void lock_read(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket.write != my_ticket.write);
}

void lock_write(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket.write), 1);
    do {} while (l→cur_ticket != my_ticket);
}

void unlock_read(rw_lock_t *l) {
    xadd(&(l->cur_ticket.read), 1);
}

void unlock_write(rw_lock_t *l) {
    l->cur_ticket.write++;
}
```

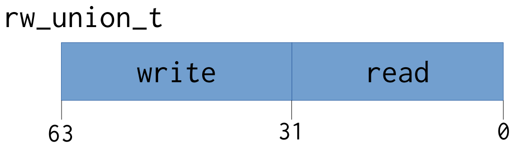


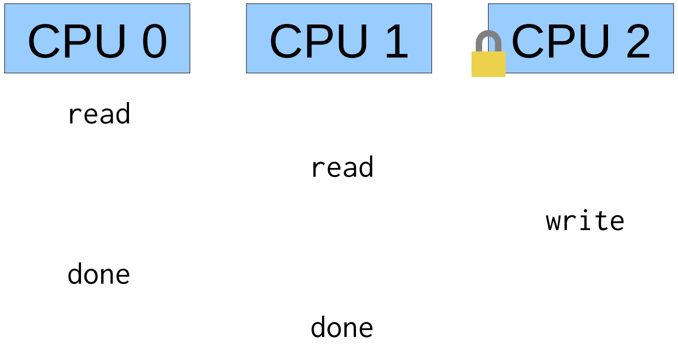


Synchronization w/ Locks

Fair Ticket Reader Writer Lock

```
struct rw_lock_t {
    rw_union_t cur_ticket;
    rw_union_t next_ticket;
};

void lock_read(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket.write != my_ticket.write);
}

void lock_write(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket.write), 1);
    do {} while (l→cur_ticket != my_ticket);
}

void unlock_read(rw_lock_t *l) {
    xadd(&(l->cur_ticket.read), 1);
}

void unlock_write(rw_lock_t *l) {
    l->cur_ticket.write++;
}
```





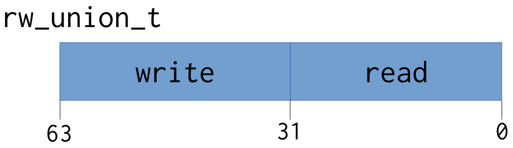
my_ticket 0|2

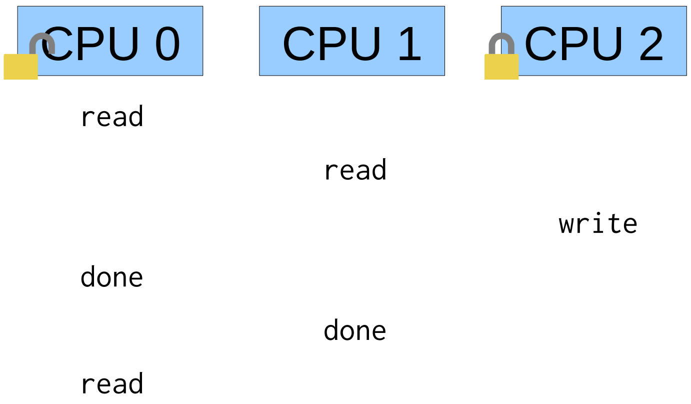
next_ticket 1|2 cur_ticket 0|2

Synchronization w/ Locks

Fair Ticket Reader Writer Lock

```
struct rw_lock_t {
    rw_union_t cur_ticket;
    rw_union_t next_ticket;
};

void lock_read(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket.write != my_ticket.write);
}

void lock_write(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket.write), 1);
    do {} while (l→cur_ticket != my_ticket);
}

void unlock_read(rw_lock_t *l) {
    xadd(&(l->cur_ticket.read), 1);
}

void unlock_write(rw_lock_t *l) {
    l->cur_ticket.write++;
}
```





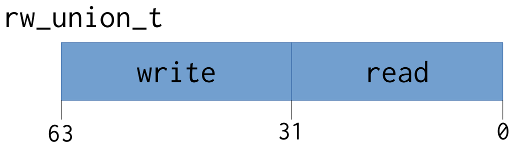
my_ticket 1|2 0|2

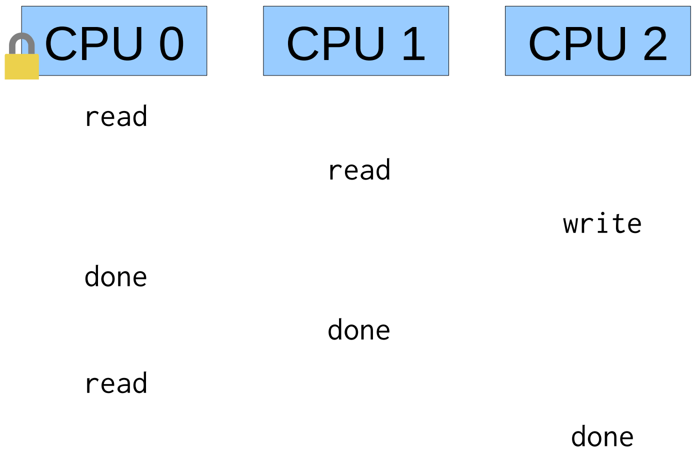
next_ticket 1|3 cur_ticket 0|2

Synchronization w/ Locks

Fair Ticket Reader Writer Lock

```
struct rw_lock_t {
    rw_union_t cur_ticket;
    rw_union_t next_ticket;
};

void lock_read(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket.write != my_ticket.write);
}

void lock_write(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket.write), 1);
    do {} while (l→cur_ticket != my_ticket);
}

void unlock_read(rw_lock_t *l) {
    xadd(&(l->cur_ticket.read), 1);
}

void unlock_write(rw_lock_t *l) {
    l->cur_ticket.write++;
}
```





my_ticket 1|2

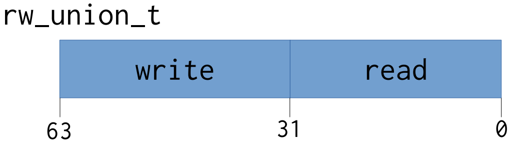
next_ticket 1|3

cur_ticket 1|2

Synchronization w/ Locks

Fair Ticket Reader Writer Lock

```
struct rw_lock_t {
    rw_union_t cur_ticket;
    rw_union_t next_ticket;
};

void lock_read(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket), 1);
    do {} while (l->cur_ticket.write != my_ticket.write);
}

void lock_write(rw_lock_t *l) {
    auto my_ticket = xadd(&(l->next_ticket.write), 1);
    do {} while (l→cur_ticket != my_ticket);
}

void unlock_read(rw_lock_t *l) {
    xadd(&(l->cur_ticket.read), 1);
}

void unlock_write(rw_lock_t *l) {
    l->cur_ticket.write++;
}
```



Difficult to get right!

- No overflows within each counter
  - Counters must be large enough so that all threads can be readers or writers
- No overflow from one counter to another
  - Read counter must not overflow into write counter → protection bit

Synchronization w/ Locks

Lockholder Preemption

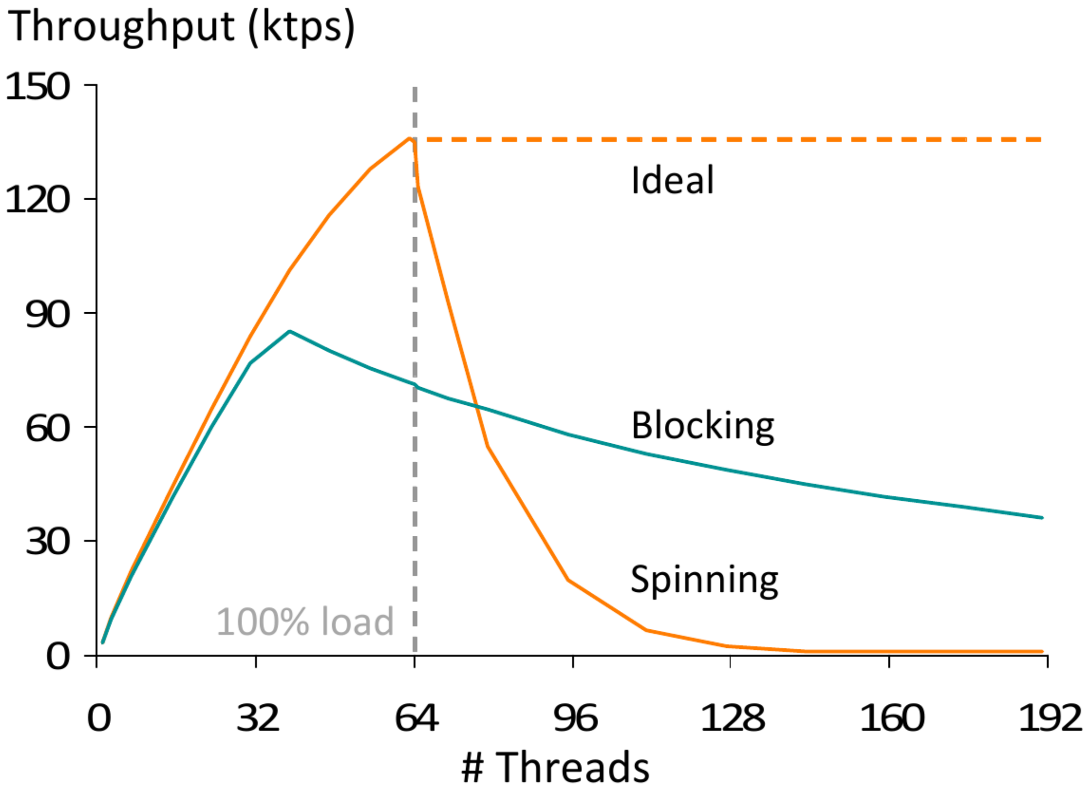


Figure 3 Comparison of the operation throughput using blocking or spinning primitives.
Johnson et al. [2010]: “Decoupling Contention Management from Scheduling”

Synchronization w/ Locks

Lockholder Preemption

- Spinning time of a CPU is increased by the time the current lockholder can not execute
  - Lockholder gets preempted by other spinning threads on the same CPU
  - Especially bad for Ticket and MCS-Locks

- Blocking instead of spinning reduces the load on the system and can thereby help preventing lockholder preemption
- Prohibit preemption of the lockholder by disabling interrupts while being in CS
  - `cli` + `sti` in combination with `pushf` + `popf`

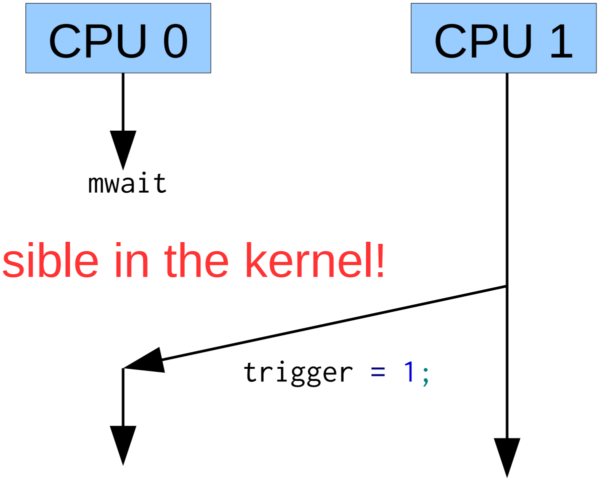
Only possible in the kernel! Very dangerous!

Synchronization w/ Locks

Monitor, Mwait

- Possibility to stop the CPU/HT while waiting for a lock (only x86)
  - Can be used to put a CPU in a sleep state
  - Allows better usage of the remaining resources
- `monitor` – watches a given cache line
- `mwait` – stops CPU/HT until write to monitored cache line or interrupt

```
while (trigger != 1) {
    monitor(&trigger);
    if (trigger != 1)
        mwait
}
```



Only possible in the kernel!

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. Scott
- *Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors*
  John M. Mellor-Crummey, Michael L. 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

References

- *Decoupling Contention Management from Scheduling*
  F.R. Johnson, R. Stoica, A. Ailamaki, T. Mowry
- *Corey: An Operating System for Many Cores*
  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