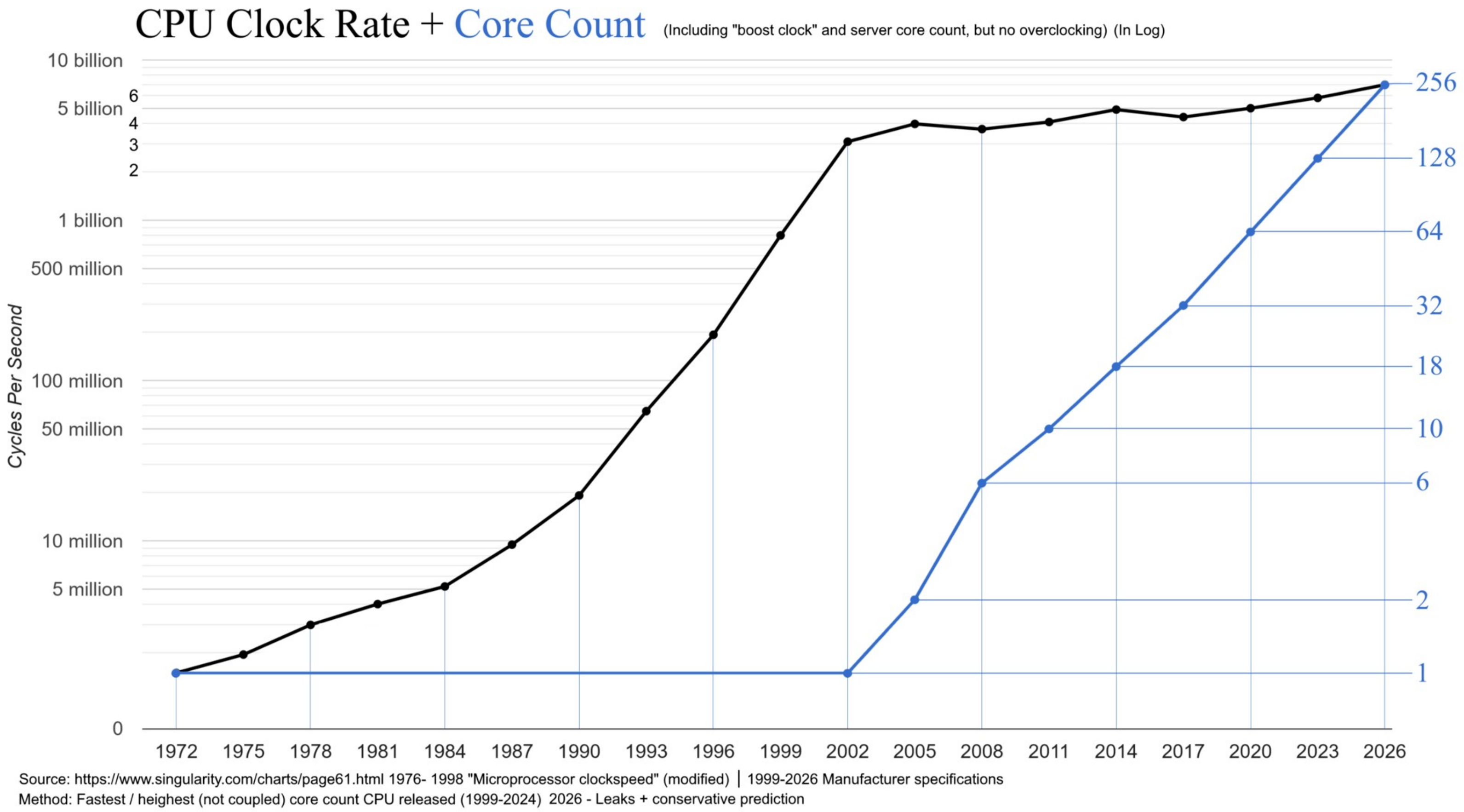
Multiprocessor Systems II

Chapter 8, 8.1

Learning Outcomes

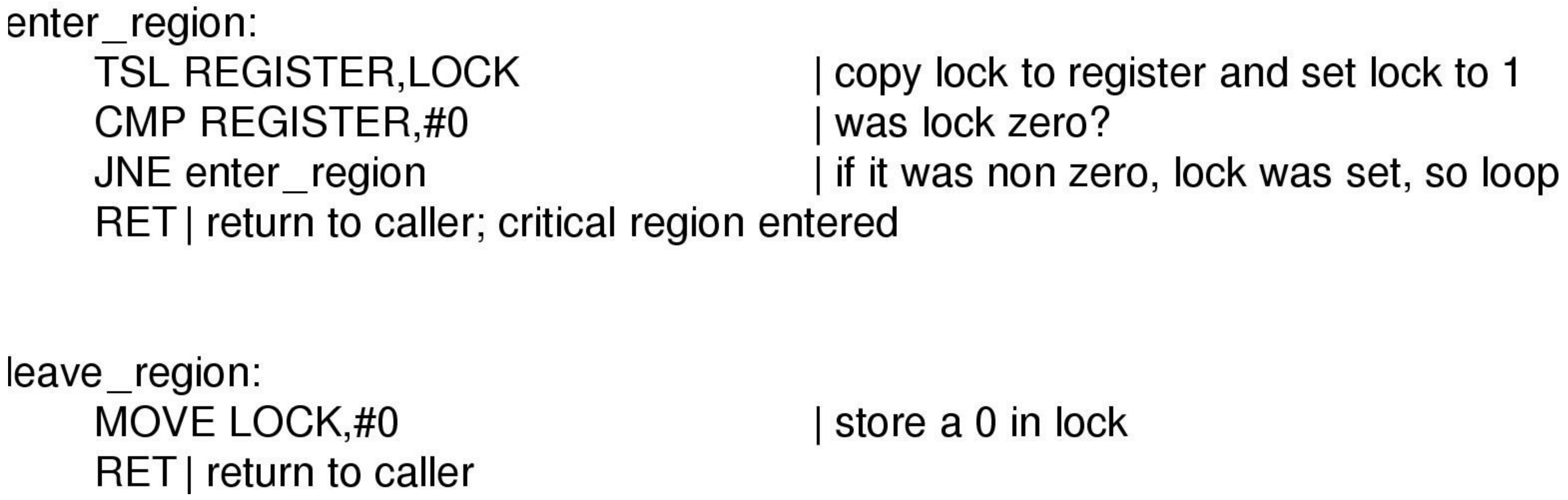
- An understanding of the structure and limits of multiprocessor hardware.
- An appreciation of approaches to operating system support for multiprocessor machines.
- An understanding of issues surrounding and approaches to construction of multiprocessor synchronisation primitives.

CPU clock-rate increase slowing



Source: https://www.singularity.com/charts/page61.html 1976- 1998 "Microprocessor clockspeed" (modified) | 1999-2026 Manufacturer specifications
Method: Fastest / heighest (not coupled) core count CPU released (1999-2024) 2026 - Leaks + conservative prediction

Recall Mutual Exclusion with Test-and-Set



```
enter_region:
        TSL REGISTER,LOCK                    | copy lock to register and set lock to 1
        CMP REGISTER,#0                      | was lock zero?
        JNE enter_region                     | if it was non zero, lock was set, so loop
        RET | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0                         | store a 0 in lock
        RET | return to caller
```

Entering and leaving a critical region using the TSL instruction

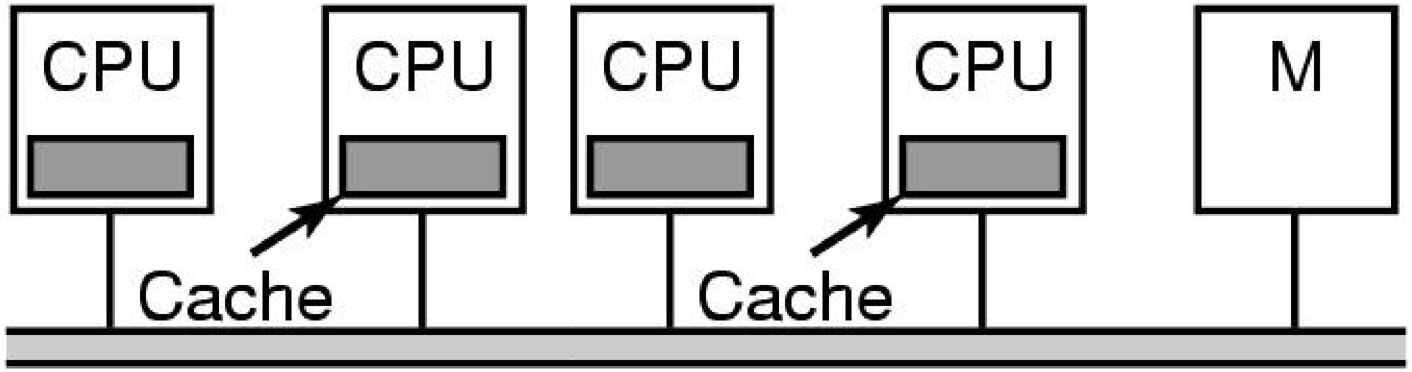
Test-and-Set

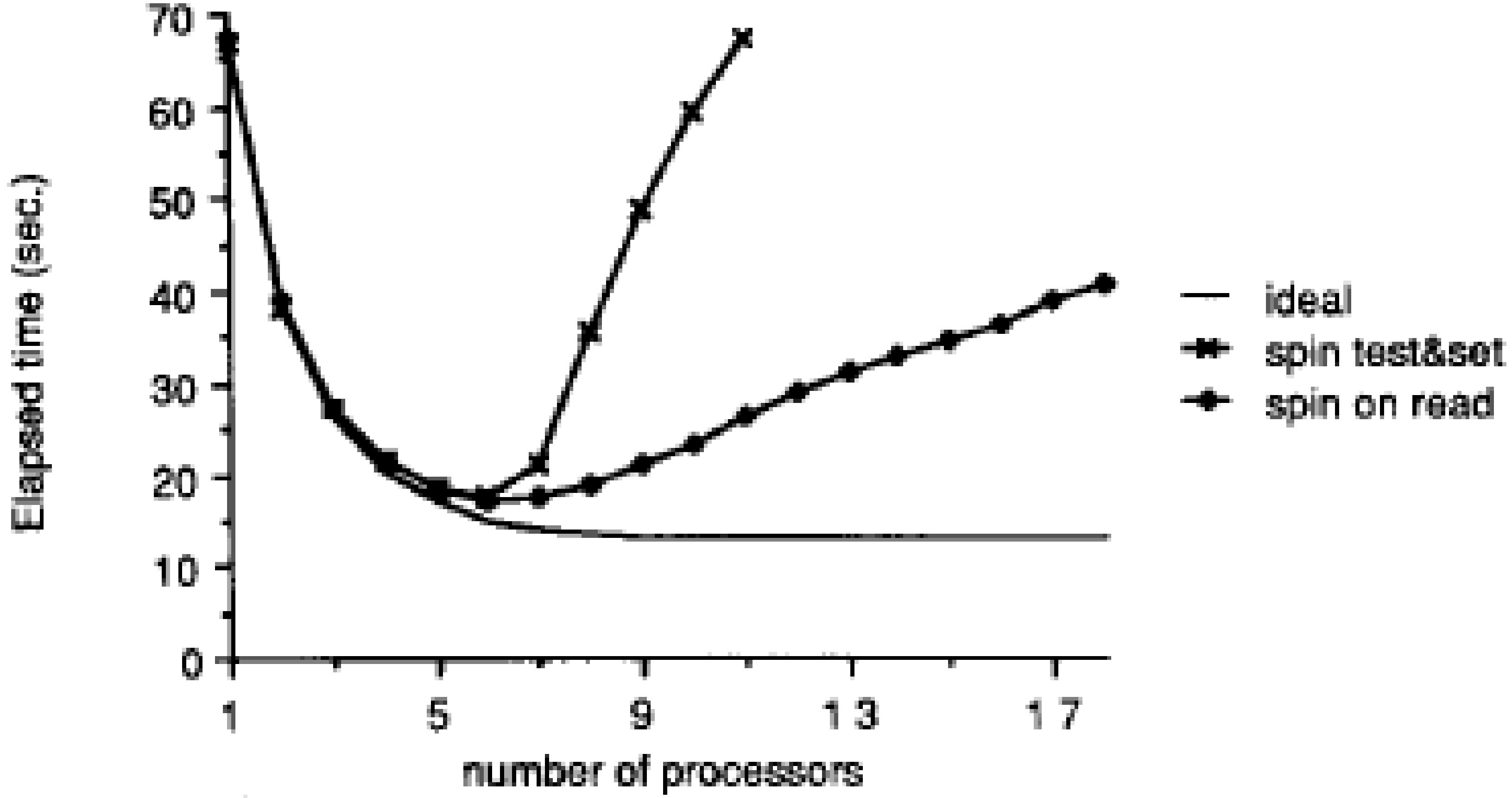
- Hardware guarantees that the instruction executes atomically on a CPU.
    - Atomically: As an indivisible unit.
  - The instruction can not stop half way through

Reducing Bus Contention

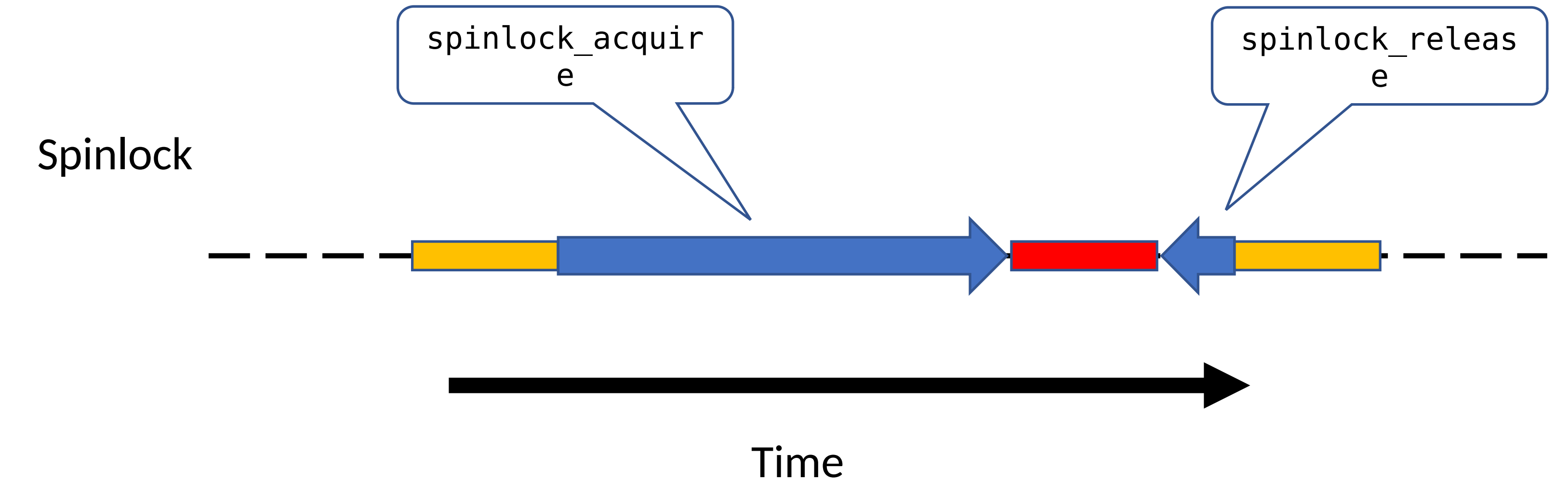
- Read before TSL
    - Spin reading the lock variable waiting for it to change
    - When it does, use TSL to acquire the lock
- Allows lock to be shared read-only in all caches until its released
    - no bus traffic until actual release
- No race conditions, as acquisition is still with TSL.

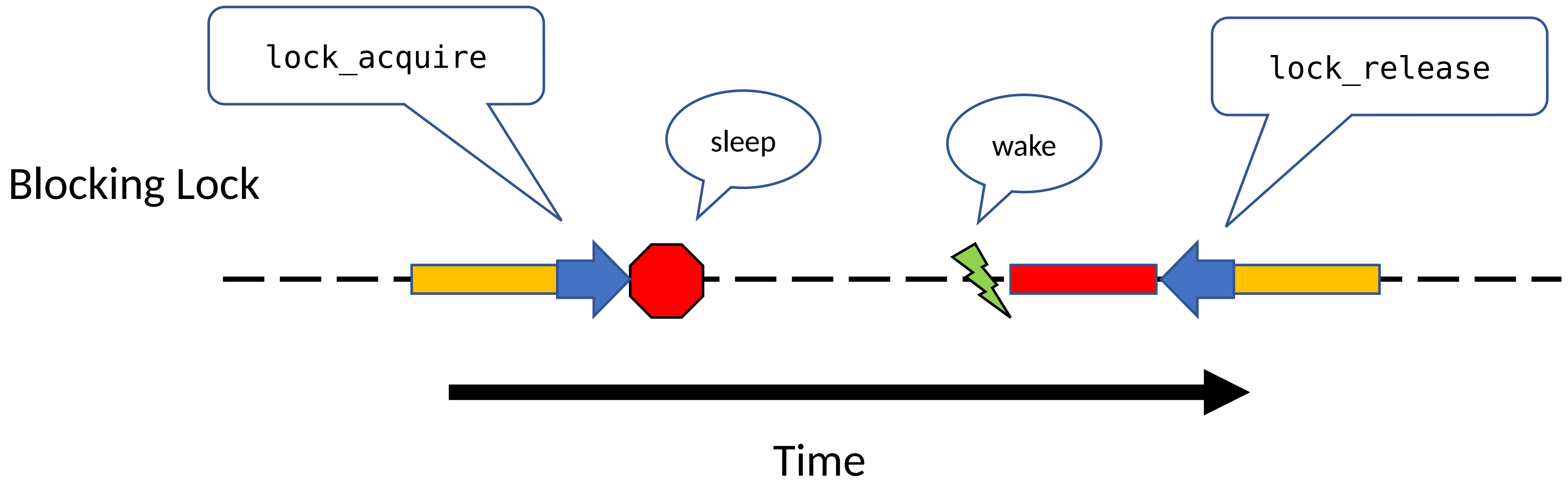
```
start:
while (lock == 1)
   ;
r = TSL(lock);
if (r == 1)
  goto start;
```



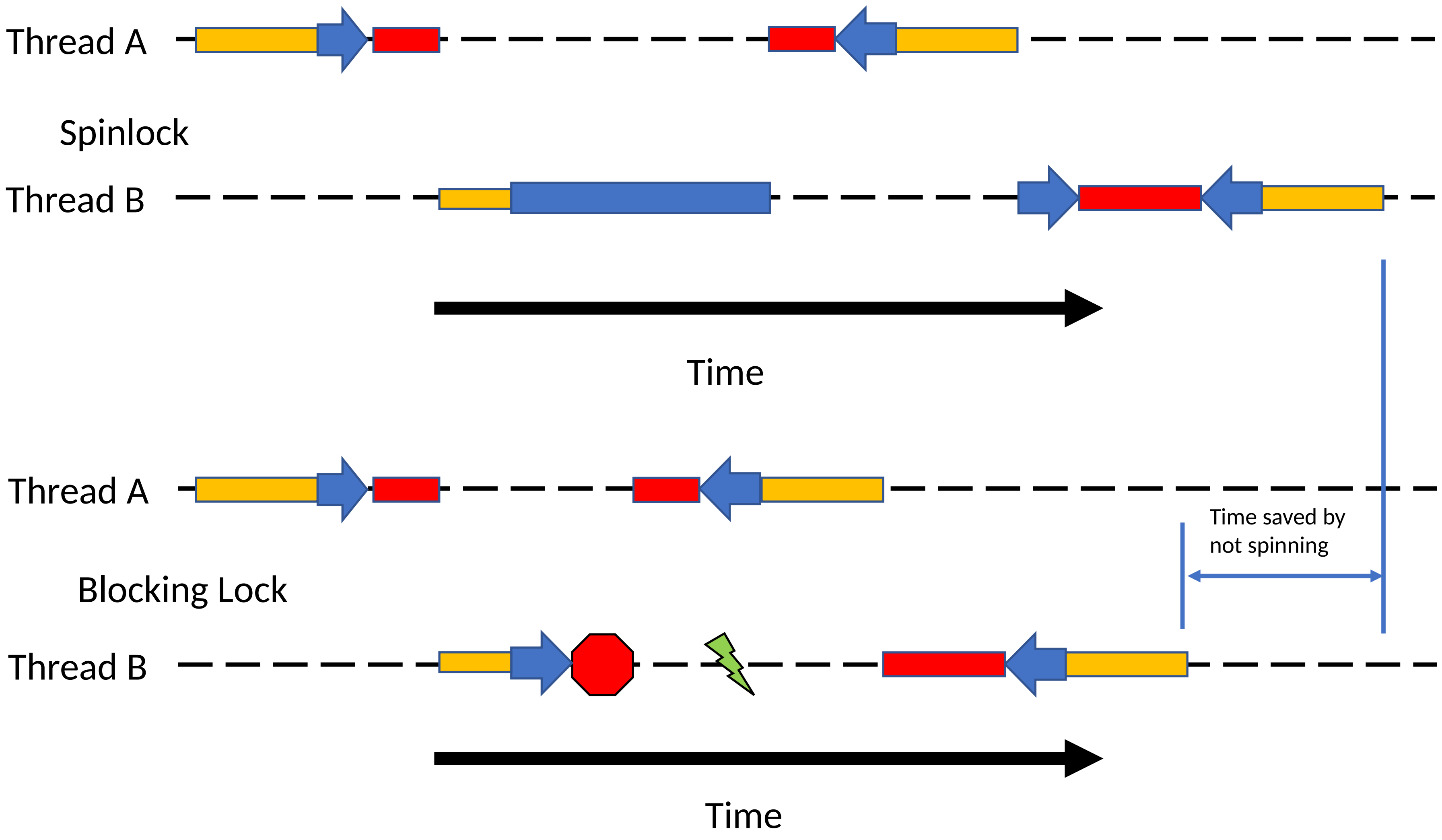

Elapsed time (sec.)
70
60
50
40
30
20
10
0
1
5
9
13
17
number of processors
ideal
spin test&set
spin on read

Spinning Locks versus Blocking Locks





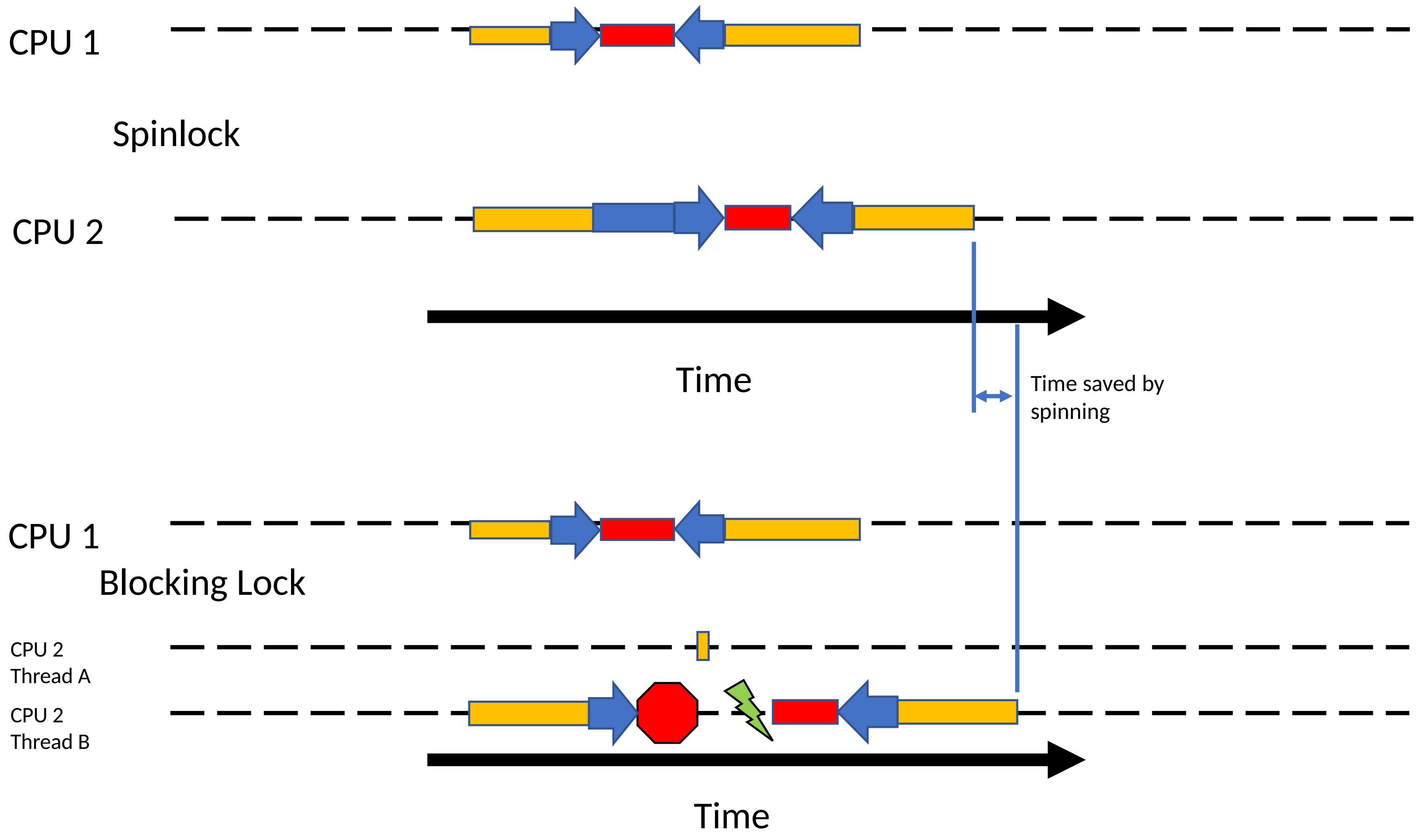
Uniprocessor: Spinning versus Blocking



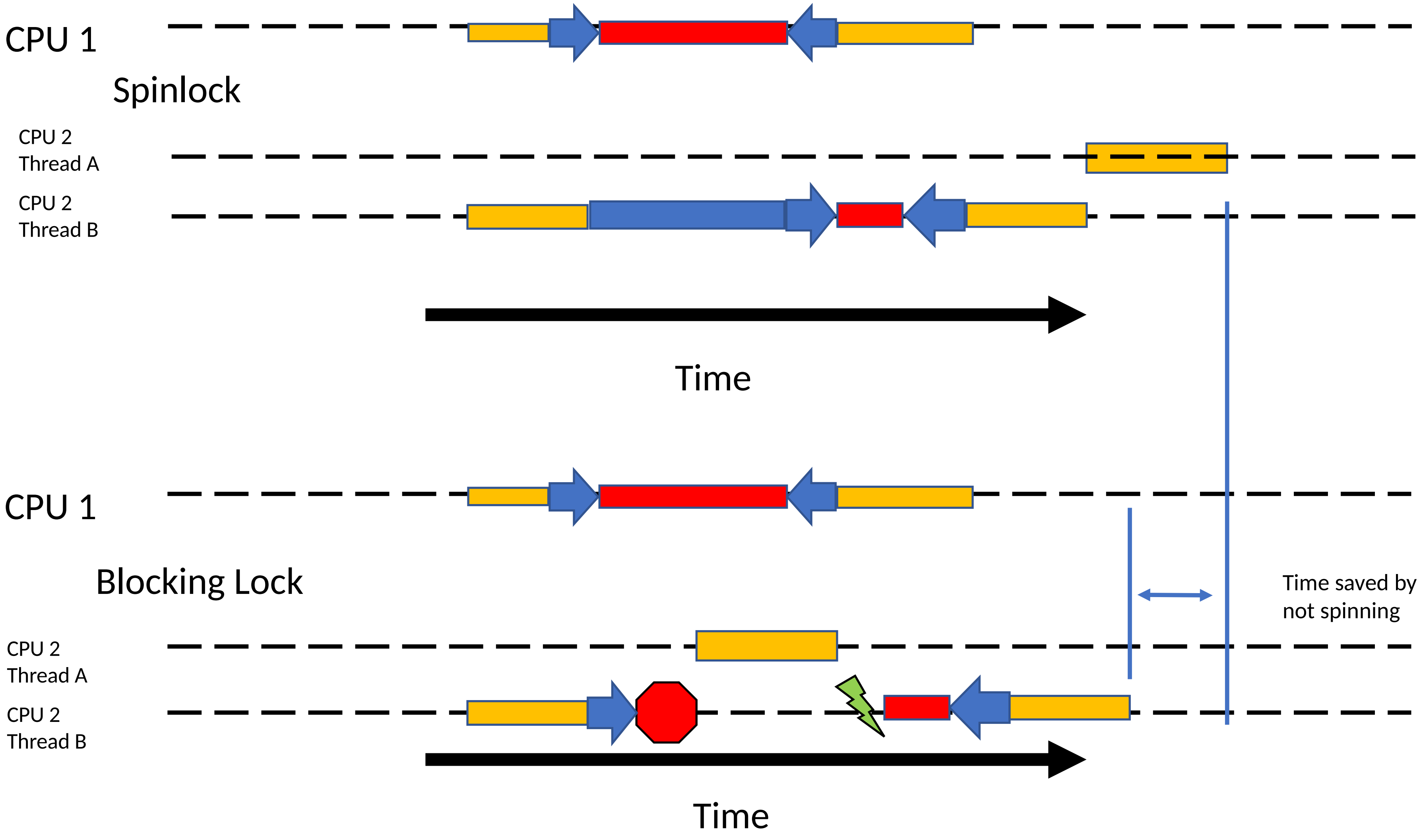
Spinning versus Blocking and Switching

- Spinning (busy-waiting) on a lock makes no sense on a uniprocessor
  - The was no other running process to release the lock
  - Blocking and (eventually) switching to the lock holder is the only sensible option.
- On multiprocessor systems, the decision to spin or block is not as clear.
  - The lock might be held by another running CPU and be freed in the near future while the current task spins

Multiprocessor: Spinning versus Blocking



Multiprocessor: Spinning versus Blocking



Spinning versus Switching

- Switching to another process takes time
    - Save context and restore another
    - Cache relevant to current process not new process
        - Adjusting the cache working set also takes time
    - TLB is similar to cache
  - Blocking and resuming requires two switches
- Spinning wastes CPU time directly

- Trade off
  - Might the lock be held for longer than 2x switch overhead?
    - Yes, it’s probably more efficient to block
    - No, it’s probably more efficient to spin

⇒ Spinlocks expect critical sections to be short

⇒ No waiting for I/O within a spinlock

⇒ No nesting locks within a spinlock

Preemption and Spinlocks

- Critical sections synchronised via spinlocks are expected to be short
  - Avoid other CPUs wasting cycles spinning
- What happens if the spinlock holder is preempted at end of holder's timeslice?
  - Mutual exclusion is still guaranteed
  - Other CPUs will spin until the holder is scheduled again!!!!!

⇒ Within the OS, Spinlock implementations disable interrupts in addition to acquiring locks

- avoids lock-holder preemption
- avoids spinning on a uniprocessor

A Hybrid Lock

- Suppose we want to implement a user level lock
  - System has test-and-set (similar story for other ops)
  - System calls take ~ 200 cycles
  - Thread switches take ~ 2000 cycles
- Simple strategy:
  - Attempt to take the lock with test-and-set
  - If not, **spin** with read/test-and-set for ~ 1000 cycles
  - Then trigger a thread-switch
    - e.g. wait on an OS-level object

Bonus Content

- Some additional content on multi-core locking for today
    - Invented Cache Protocol
    - Ticket locks
    - MCS/Queue locks

    - Note this content is not assessible this year

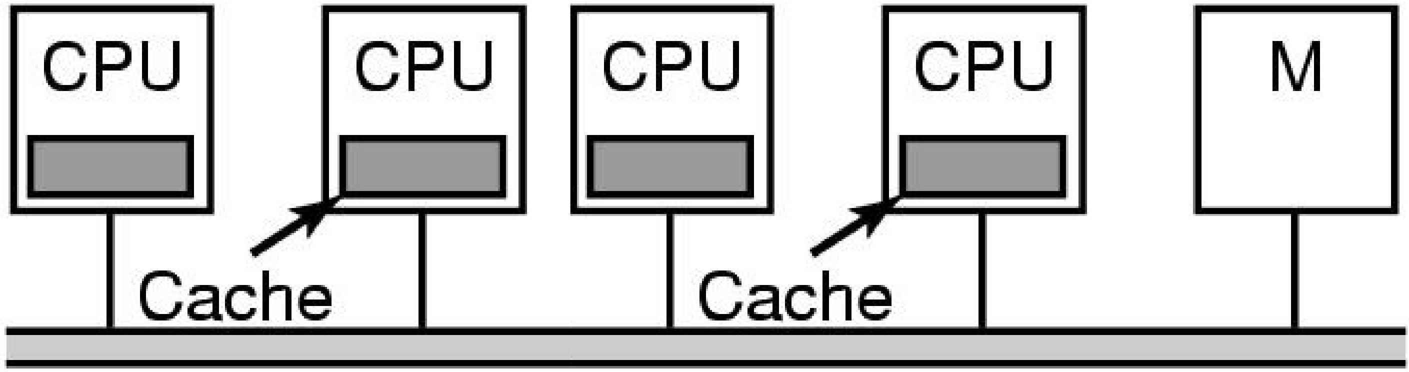
Cache State (Invented)

- To understand why Test-Test-Set saves inter-cache bandwidth
- This is an invented cache state

R/V	W/X	D	L	V Addr	Contents

- Like TLB, associative mapping + status bits
  - Read/Valid
  - Write/Exclusive
  - Dirty
  - Locked

```
start:
while (lock == 1)
   ;
r = TSL(lock);
if (r == 1)
  goto start;
```



Cache Status Bits

R/V	W/X	D	L	V Addr	Contents

- Read/Valid
  - This is a valid cache entry, and can be read
- Write/Exclusive
  - This entry can be written. No other entry is valid.
- .Dirty
  - This entry contains writes not yet written to memory.
- Locked
  - ???

Invalidate Operations

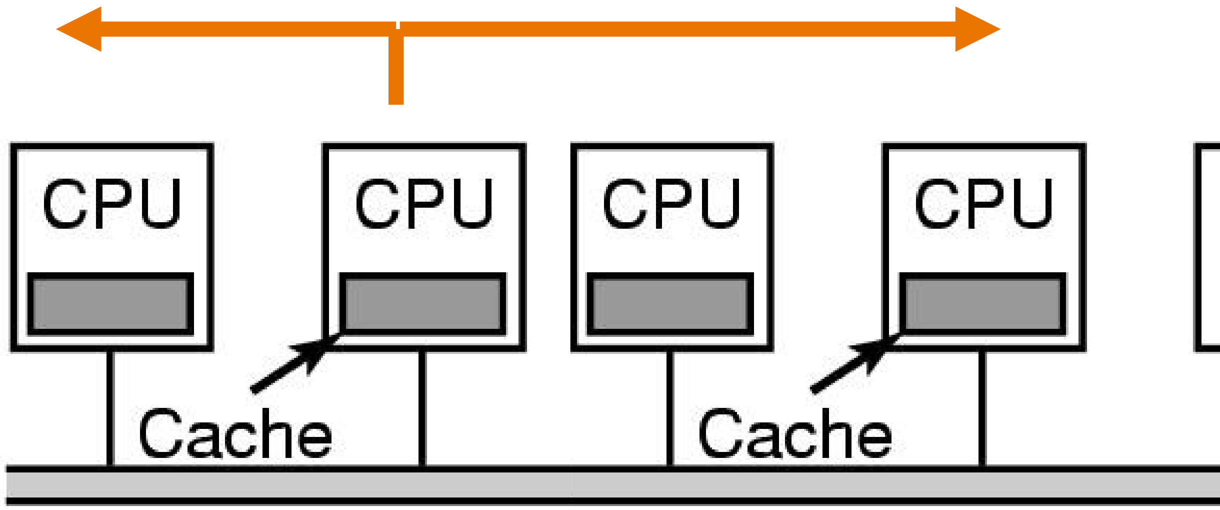
- CPU 1
  - Write @ 0x1f00

R/V	W/X	D	L	V Addr	Contents
1	0	0	0	0x1f00	0x0000

- CPU 2

R/V	W/X	D	L	V Addr	Contents
1	0	0	0	0x1f00	0x0000

- CPU 1 broadcasts
  - INVAL(ALL)
- CPU 2
  - ACK



Cache Status Bit Invariants

R/V	W/X	D	L	V Addr	Contents

- W/X **implies** R/V
- W/X **implies no other** R/V
- D **implies** W/X
- R/V **and not** D **implies** **(** Contents **equals** Memory **)**

Invalidate to Read

CPU 1

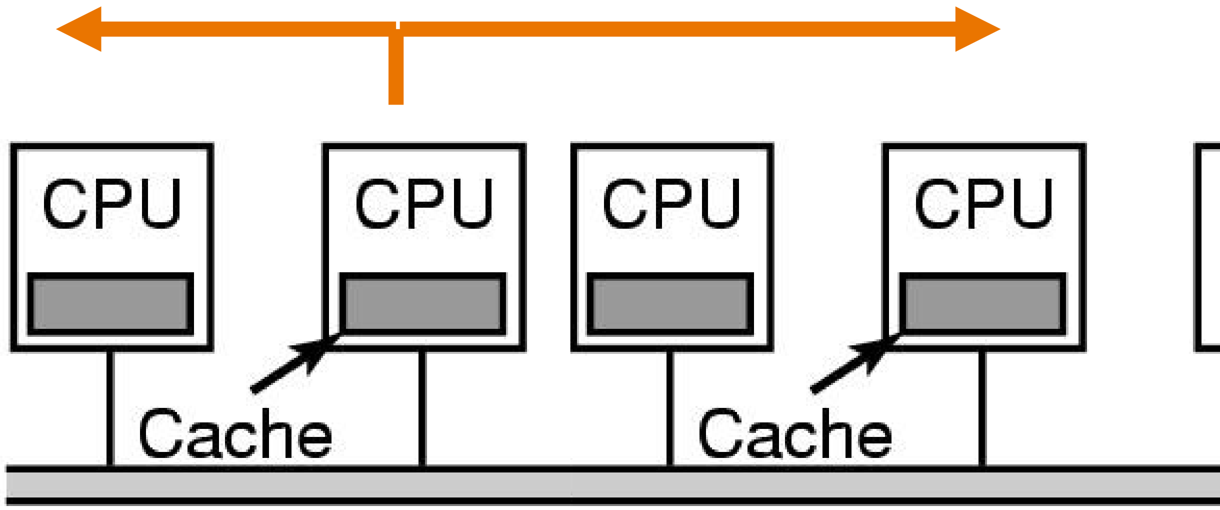
- Read @ 0x1f00

R/V	W/X	D	L	V Addr	Contents
0	0	0	0	N/A	N/A

CPU 2

R/V	W/X	D	L	V Addr	Contents
1	1	0	0	0x1f00	0x1234

- CPU 1 broadcasts
  - INVAL(EXCL)
- CPU 2
  - ACK
- CPU 1 can read memory



Invalidate to Read II

CPU 1

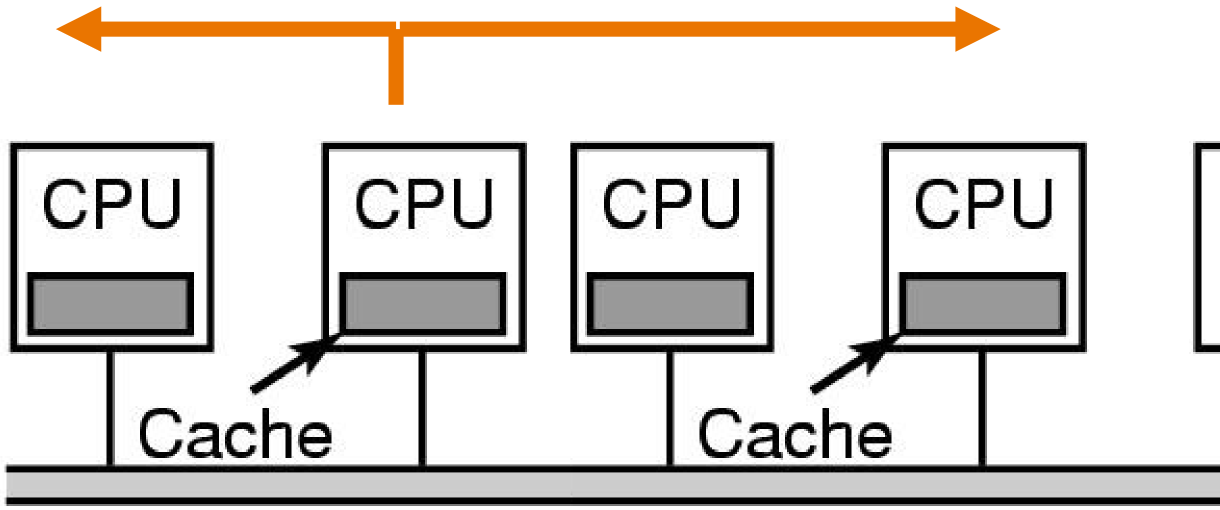
- Read @ 0x1f00

R/V	W/X	D	L	V Addr	Contents
0	0	0	0	N/A	N/A

CPU 2

R/V	W/X	D	L	V Addr	Contents
1	1	1	0	0x1f00	0x4567

- CPU 1 broadcasts
  - INVAL(EXCL)
- CPU 2
  - WAIT!!



Lock Status Bit

R/V	W/X	D	L	V Addr	Contents

- Lock status bit causes **WAIT!!**
- Prevents other caches causing a writeback & invalidate

- Used to implement atomics
- CPU test-and-set:
  - 1: Lock cache line
  - 2: Test-and-set between cache line and register
  - 3: Unlock cache line

Test-and-Set Lock Phase 1

CPU 1

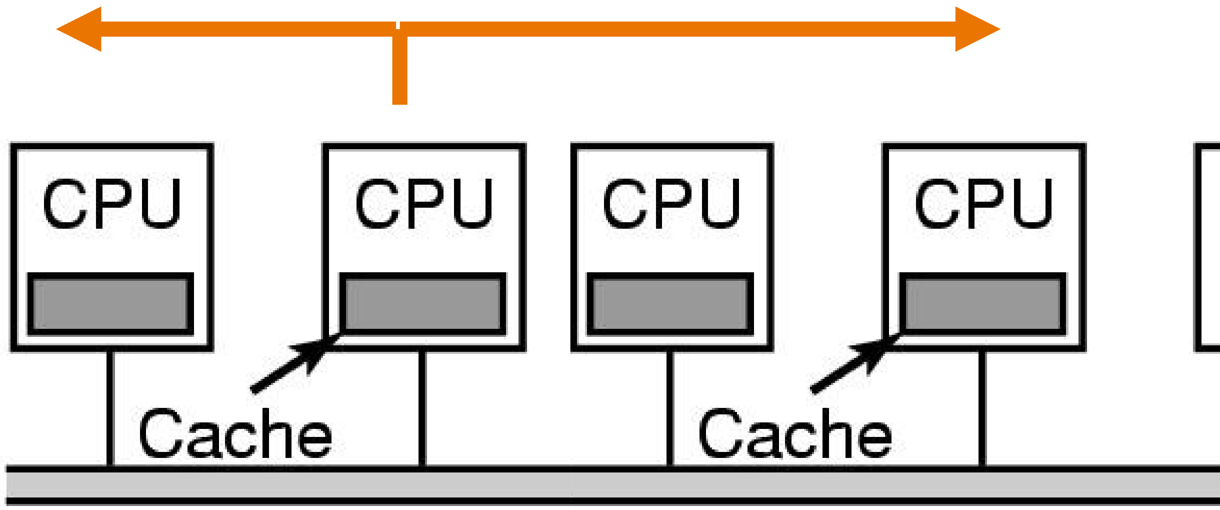
- TSL @ 0x1f00
- INVAL->Lock

R/V	W/X	D	L	V Addr	Contents
1	1	1	1	0x1f00	0 → 1

CPU 2

- TSL @ 0x1f00
- Must WAIT

R/V	W/X	D	L	V Addr	Contents
0	0	0	0	0x1f00	N/A



Test-and-Set Lock Phase 2

CPU 1

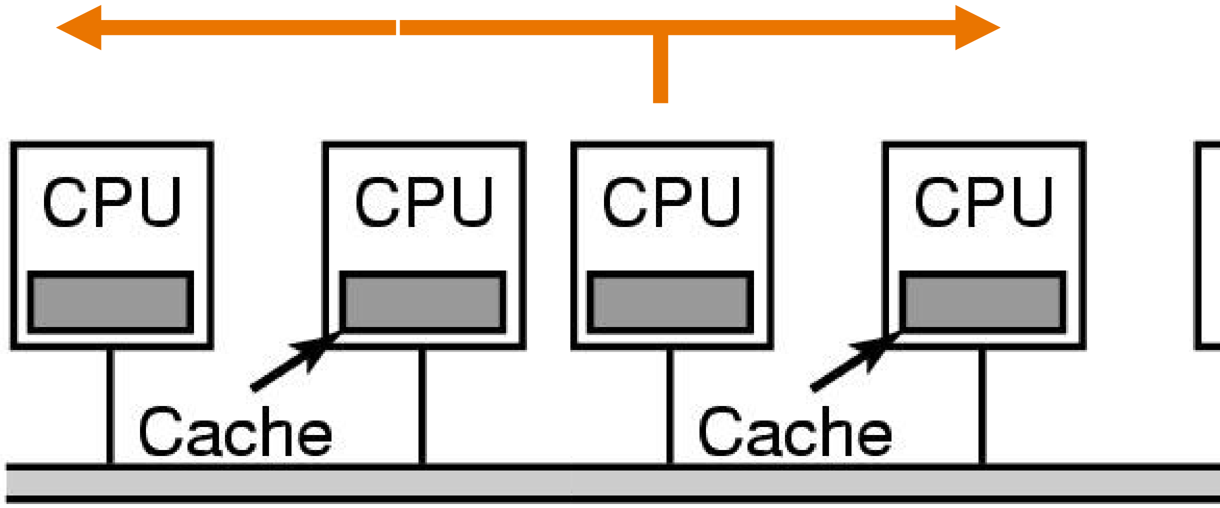
- Has lock, moves on

R/V	W/X	D	L	V Addr	Contents
0	0	0	0	0x1f00	N/A

CPU 2

- Repeats TSL @ 0x1f00
- Gets cache line
- Spins
- OK

R/V	W/X	D	L	V Addr	Contents
1	1	0	0/1	0x1f00	1



Test-and-Set Lock With 3 CPUS

CPU 1

- Has lock, moves on

R/V	W/X	D	L	V Addr	Contents
0	0	0	0	0x1f00	N/A

CPU 2

- Repeats TSL @ 0x1f00

R/V	W/X	D	L	V Addr	Contents
0/1	0/1	0	0/1	0x1f00	1

CPU 3

- Repeats TSL @ 0x1f00

R/V	W/X	D	L	V Addr	Contents
0/1	0/1	0	0/1	0x1f00	1

- Massive INVAL traffic

Test-and-Test-and-Set Lock With 3 CPUS

CPU 1

- Has lock, moves on

R/V	W/X	D	L	V Addr	Contents
1	0	0	0	0x1f00	1

CPU 2

- Repeats read @ 0x1f00

R/V	W/X	D	L	V Addr	Contents
1	0	0	0	0x1f00	1

CPU 3

- Repeats read @ 0x1f00

R/V	W/X	D	L	V Addr	Contents
1	0	0	0	0x1f00	1

- Reads remain in cache

- On unlock, INVAL flurry as CPU 2 & 3 race to take the lock

Ticket Lock

Consists of two words.

- **avail**: Next available ticket.
- **curr**: Currently active ticket.

Starting state is { **avail** = 0, **curr** = 0}.

- Lock:
  - atomic { ticket = avail; avail ++ }
  - Spin until { curr == ticket }

Ticket Lock With 3 CPUS

CPU 1

- Gets ticket 0, has lock

R/V	W/X	D	L	V Addr	Contents
1	0	0	0	0x1f00	0

CPU 2

- Gets ticket 1, waits

R/V	W/X	D	L	V Addr	Contents
1	0	0	0	0x1f00	0

CPU 3

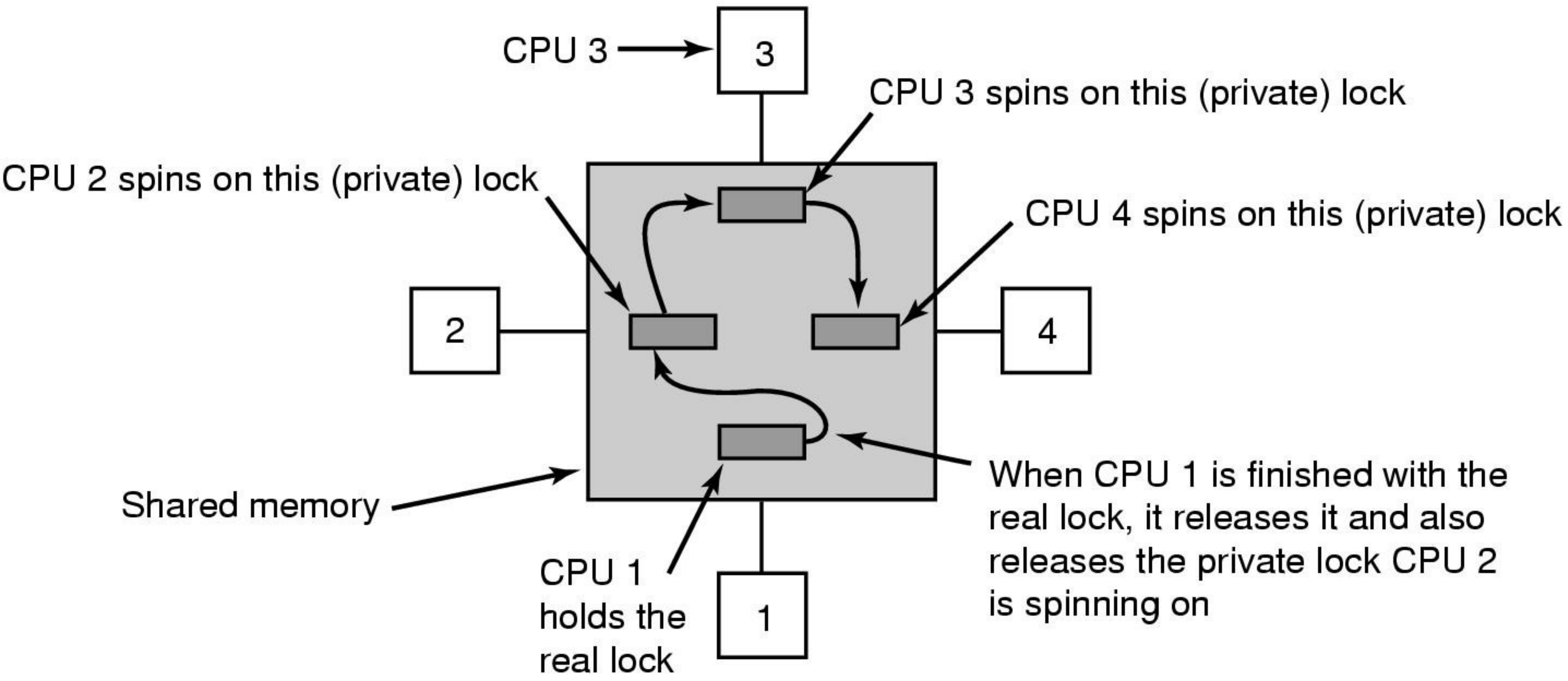
- Gets ticket 2, waits

R/V	W/X	D	L	V Addr	Contents
1	0	0	0	0x1f00	0

- CPU 1 releases by incrementing curr to 1
- Only CPU 2 attempts to proceed

MCS Locks

- Each CPU enqueues its own private lock variable into a queue and spins on it
  - No contention
- On lock release, the releaser unlocks the next lock in the queue
  - Only have bus contention on actual unlock
  - No starvation (order of lock acquisitions defined by the list)


CPU 3
3
CPU 3 spins on this (private) lock
CPU 2 spins on this (private) lock
CPU 4 spins on this (private) lock
2
4
Shared memory
When CPU 1 is finished with the real lock, it releases it and also releases the private lock CPU 2 is spinning on
CPU 1 holds the real lock
1

MCS Lock

- Requires
  - compare_and_swap()
  - exchange()
    - Also called fetch_and_store()

Today: Multiprocessing

- Need for multi-core and multi-processor systems
- Machine design, consistency and bandwidth challenges
- OS design challenges
- Synchronisation challenges on true multi-processors