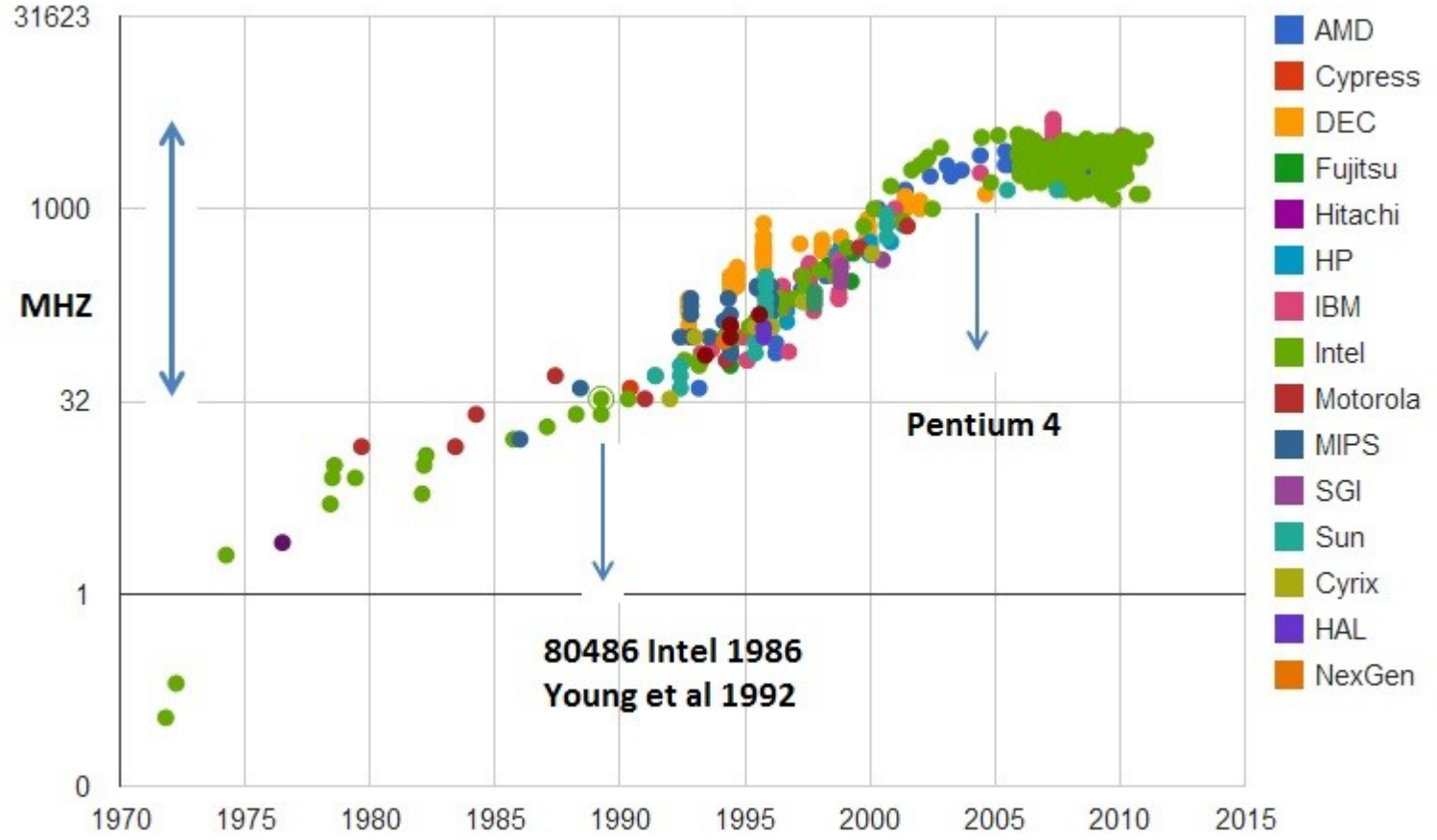
Multiprocessor Systems

Chapter 8, 8.1

Learning Outcomes

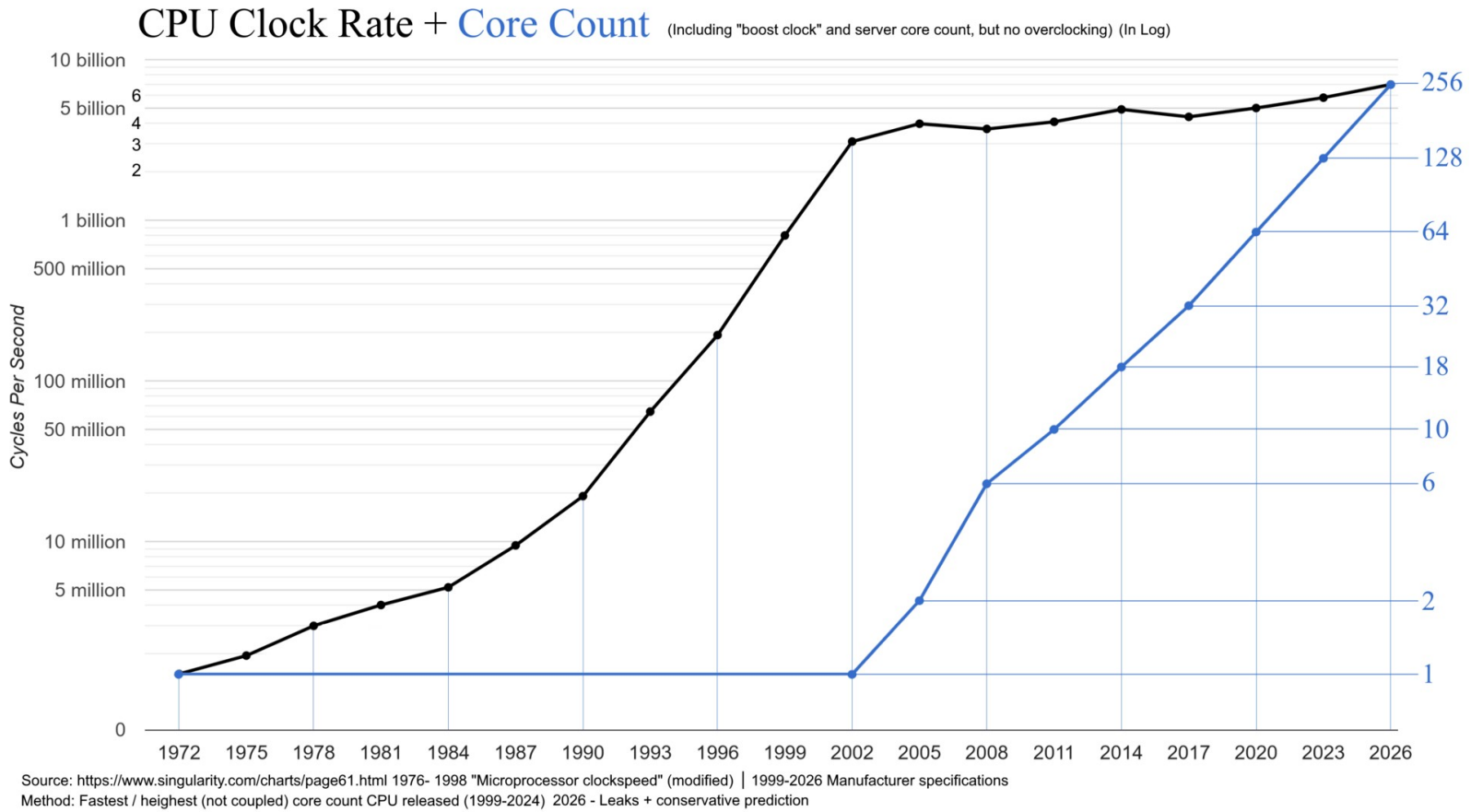
- An understanding of the structure and limits of multiprocessor hardware.
- An appreciation of approaches to operating system support for multiprocessor machines.
- An understanding of issues surrounding and approaches to construction of multiprocessor synchronisation primitives.

CPU clock-rate increase slowing

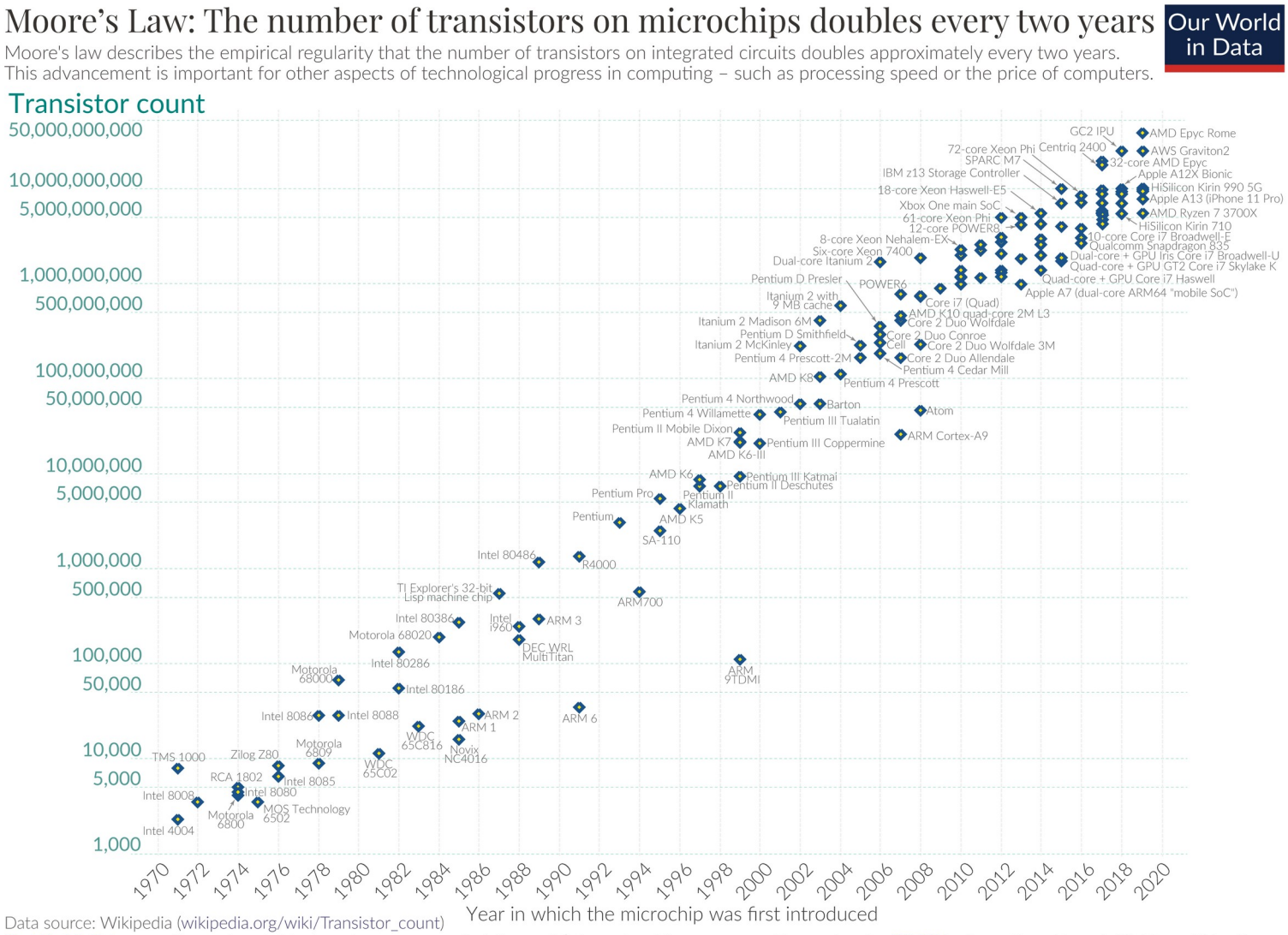


From wikipedia - by Newhorizons msk - For CPU scaling study, CC0

CPU clock-rate increase slowing



Moore’s Law continuing



Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world’s largest problems.

Related Anecdotes

- There are fascinating pictures out there of multi-core layouts
  - e.g. https://www.cpushack.com/2018/03/24/making-multicore-a-slice-of-sandy/
- There are plenty of auxiliary reasons to go multi-core
  - c.f. A conversation I had about a multi-core engine controller
- In this lecture we will discuss multi-CPU and multi-core
  - Similar but different communication layout
  - Similar but different performance concerns

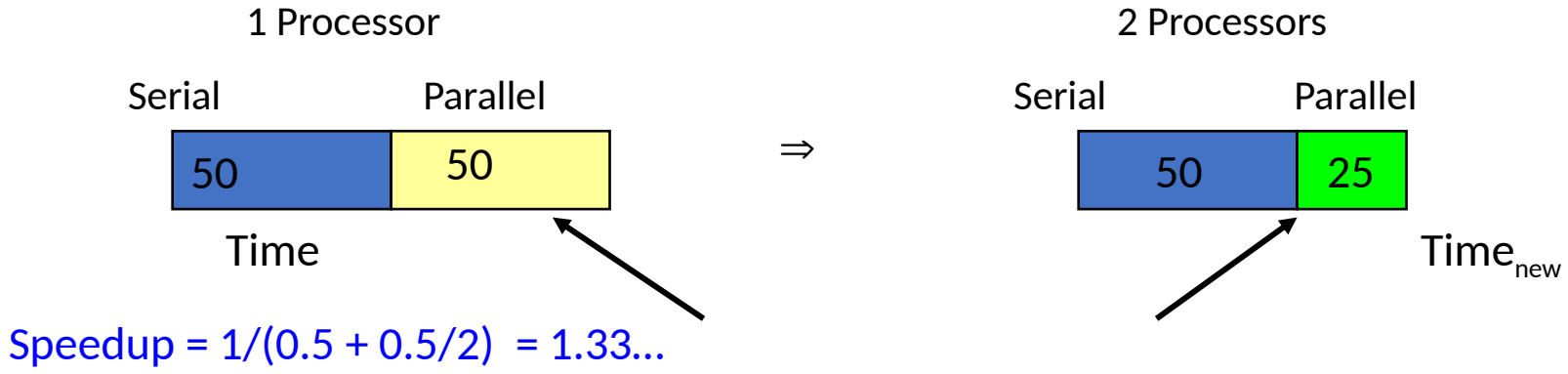
Multiprocessor System

- We will look at *shared-memory multiprocessors*
  - More than one processor sharing the same memory
- Because: a single CPU can only go so fast
  - Use more than one CPU to improve performance
  - Assumes
    - Workload can be parallelised
    - Workload is not I/O-bound or memory-bound
- Because: other computer hardware is expensive
  - e.g. disks, memory, network connection
  - Can share hardware between CPUs

Amdahl's law

- Given a proportion *P* of a program that can be made parallel, and the remaining serial portion *(1-P)*, speedup by using *N* processors

$$\frac{1}{(1-P)+\frac{P}{N}}$$

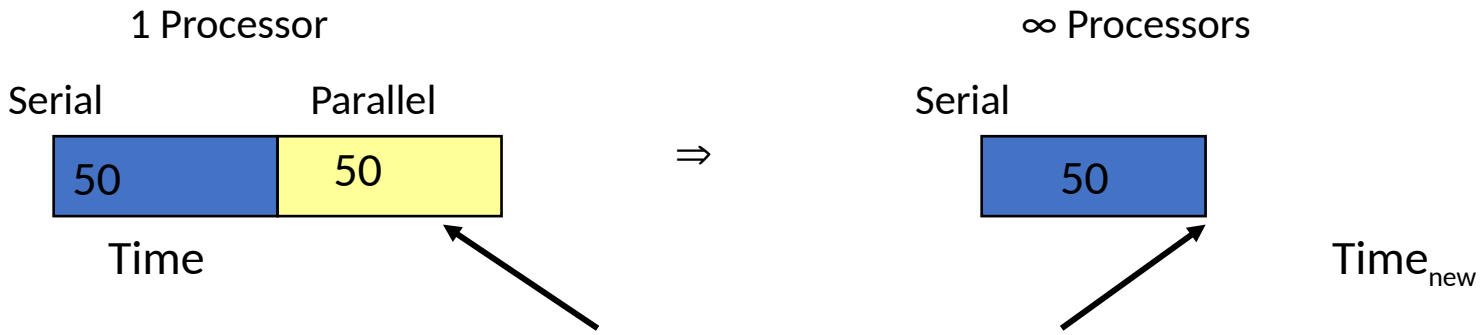


Speedup = 1/(0.5 + 0.5/2) = 1.33…

Amdahl's law

- Given a proportion *P* of a program that can be made parallel, and the remaining serial portion *(1-P)*, speedup by using *N* processors

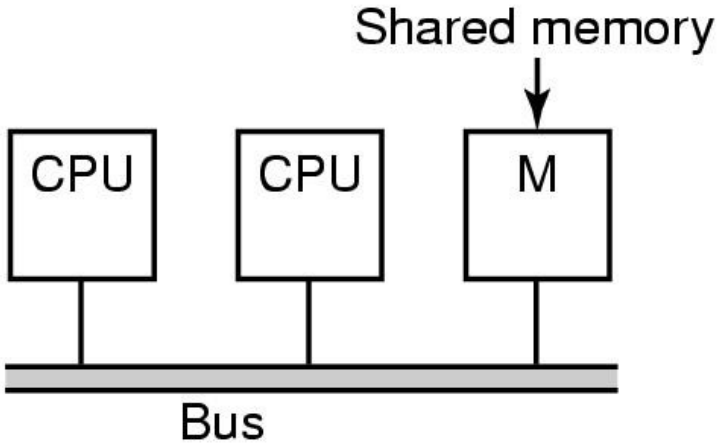
$$\frac{1}{(1-P)+\frac{P}{N}}$$



Speedup = 1/(0.5 + 0) = 2

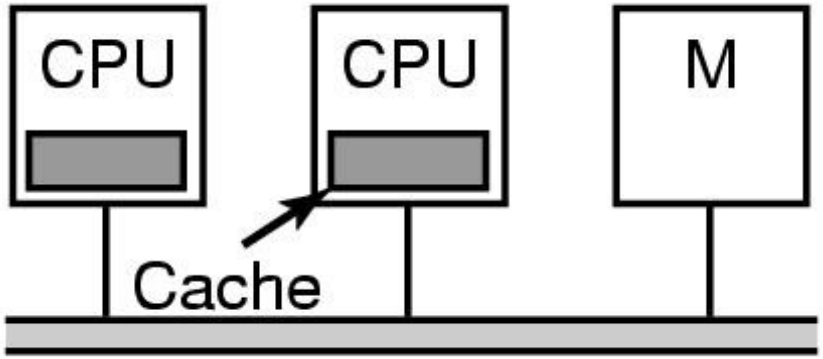
Bus-Based Uniform Memory Access Multiprocessors

- Simplest MP, multiple processors on a single bus with memory.
  - Bus controller or mechanism resolves parallel access.
  - Access to all memory occurs at roughly the same speed for all processors.
  - Bus bandwidth becomes a bottleneck with more than just a few CPUs.



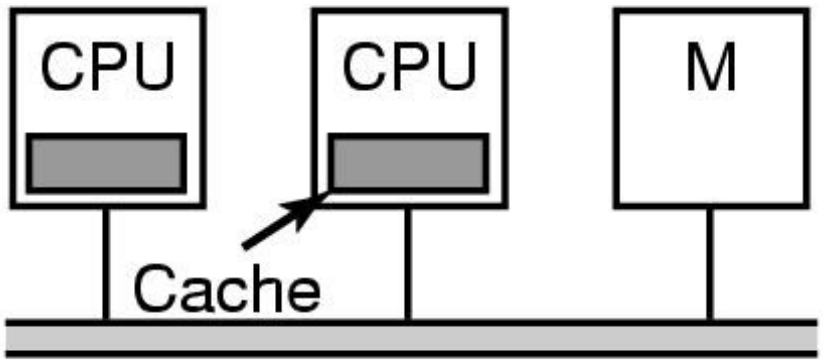
Multiprocessor caches

- Each processor has a cache to reduce its need for access to memory.
  - Hopefully most accesses are performed in the local cache.
  - Bus bandwidth still becomes a bottleneck with many CPUs.
  - Software behaviour affects the CPU/memory tradeoff.



Cache Consistency

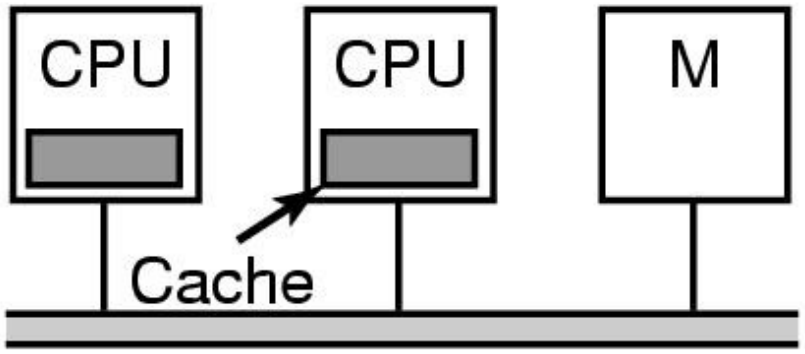
- What happens if one CPU writes to address 0x1234 (and it is stored in its cache) and another CPU reads from the same address (and gets what is in its cache)?



(b)

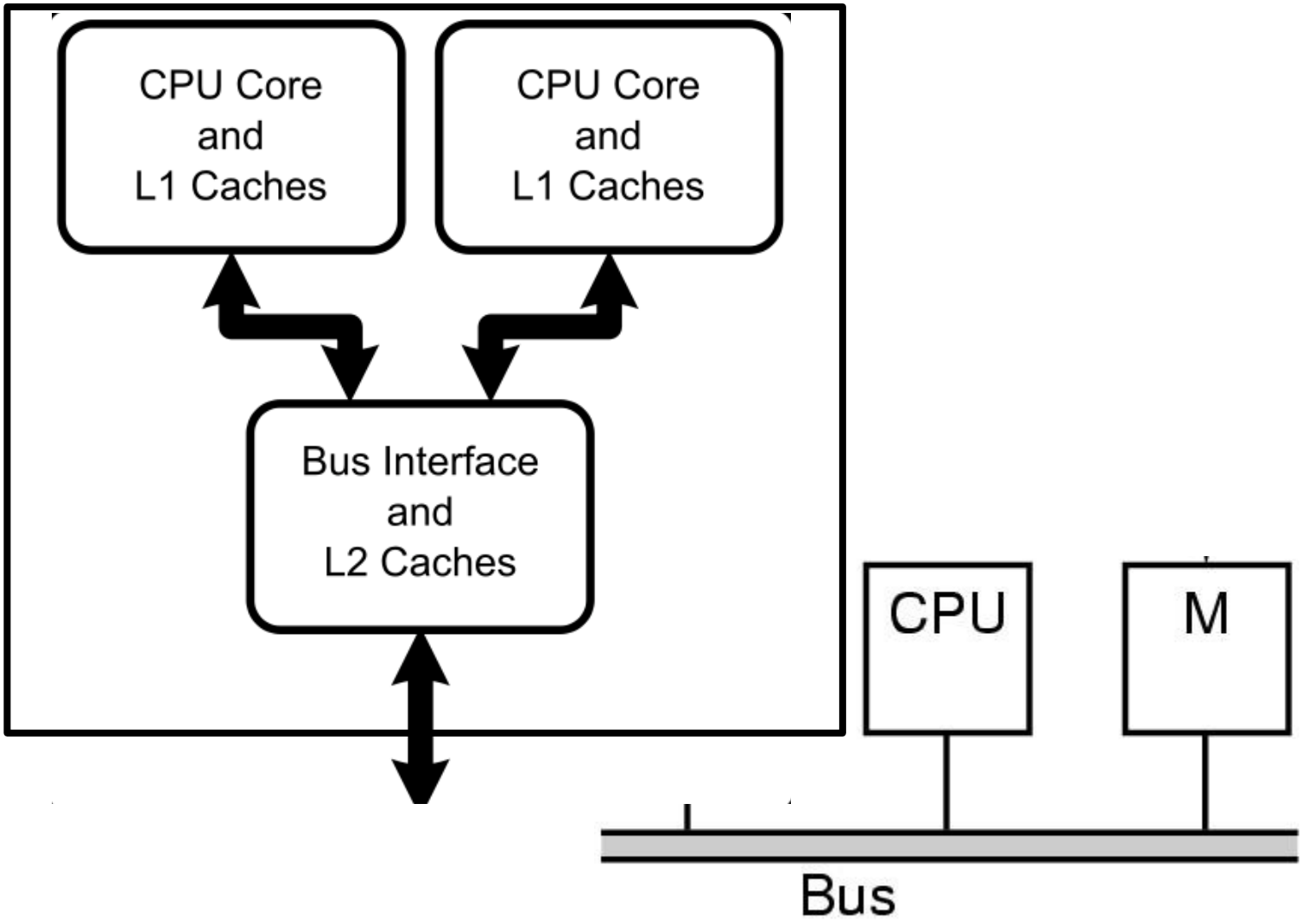
Cache Consistency

- Cache consistency is usually handled by the hardware.
  - Writes to one cache propagate to, or invalidate appropriate entries on other caches
  - Cache transactions also consume bus bandwidth



(b)

Multi-core Processor



Bus-Based UMA Multiprocessors

(UMA = Uniform Memory Architecture)

- With only a single shared bus, scalability can be limited by the bus bandwidth of the single bus.
  - Caching helps, but only helps so much.
- Alternative bus architectures (NUMA) do exist.
  - They can improve bandwidth available.
  - Nonetheless, bus bandwidth is limited.

Summary

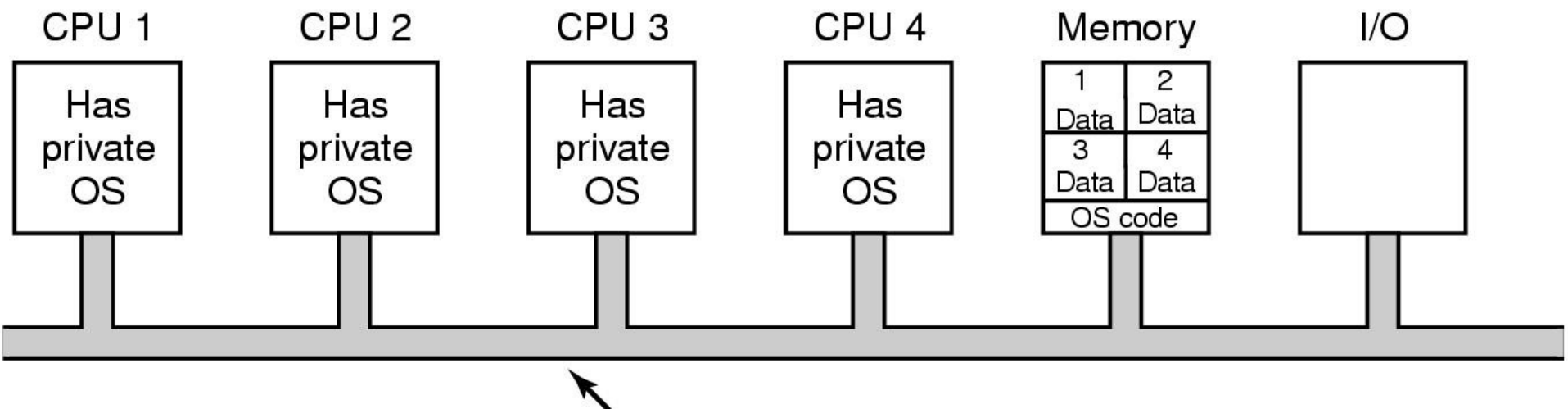
- Multiprocessors can
  - Increase computation power beyond that available from a single CPU
  - Share resources such as disk and memory
- However
  - Assumes a workload which can run in parallel
  - Assumes not I/O or memory limited
  - Shared buses (bus bandwidth) limit scalability
    - Bus bandwidth can be boosted via hardware design
    - Bus contention can be reduced by careful software design
      - Good cache locality together with limited data sharing where possible

Question

- How do we construct an OS for a multiprocessor?
  - What are some of the issues?

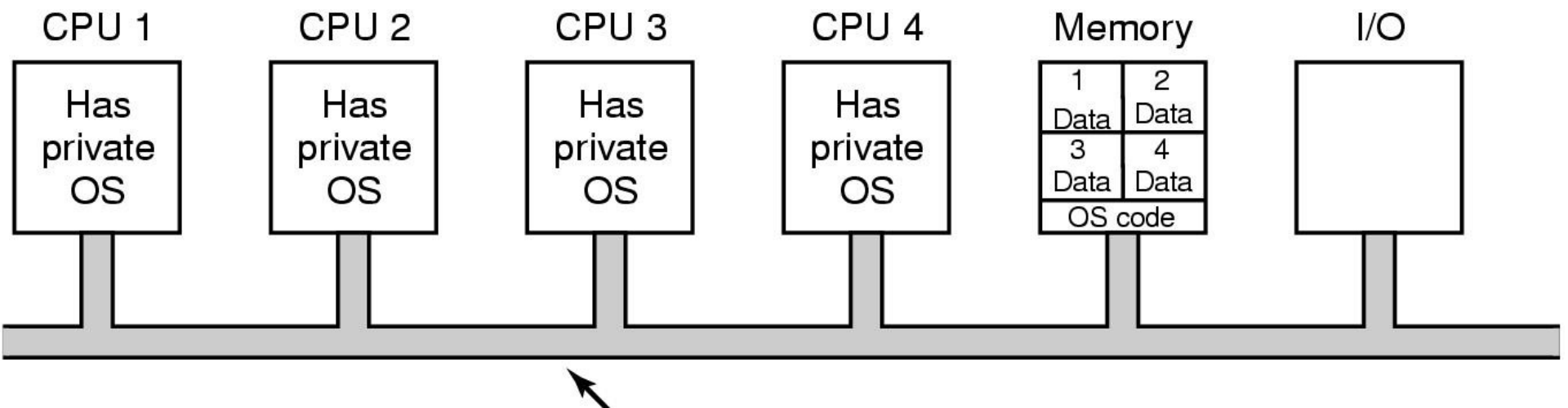
Each CPU has its own OS?

- Statically allocate physical memory to each CPU
- Each CPU runs its own independent OS
- Share peripherals
  - Or create a cross/OS virtualised “peripheral”
- Each CPU (OS) handles its processes system calls



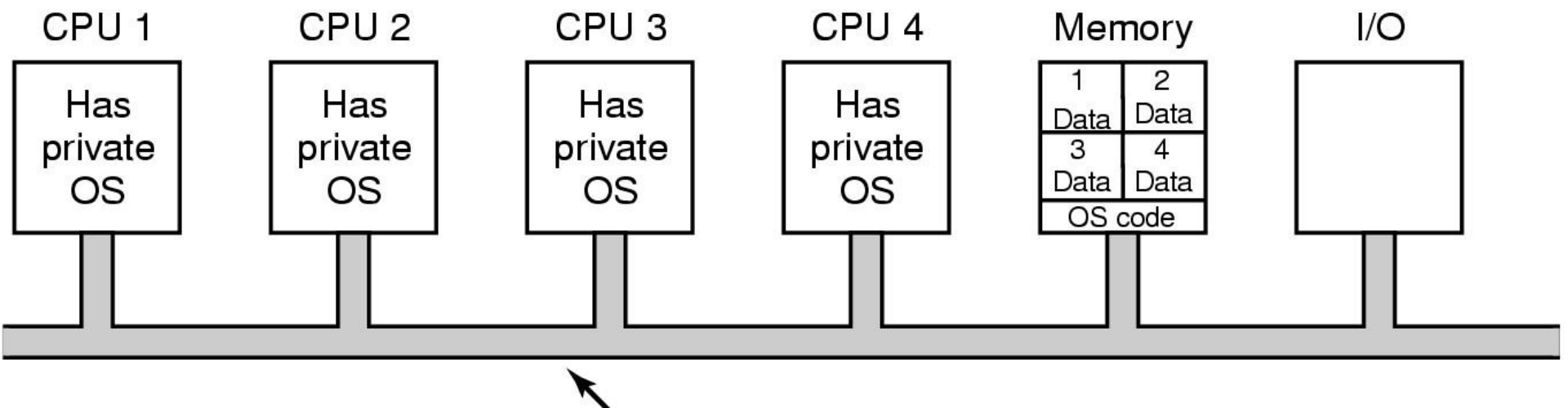
Each CPU has its own OS

- Used in early multiprocessor systems to ‘get them going’
  - Simpler to implement
  - Avoids CPU-based concurrency issues by not sharing
  - Scales – no shared serial sections
  - Modern analogy, virtualisation in the cloud.



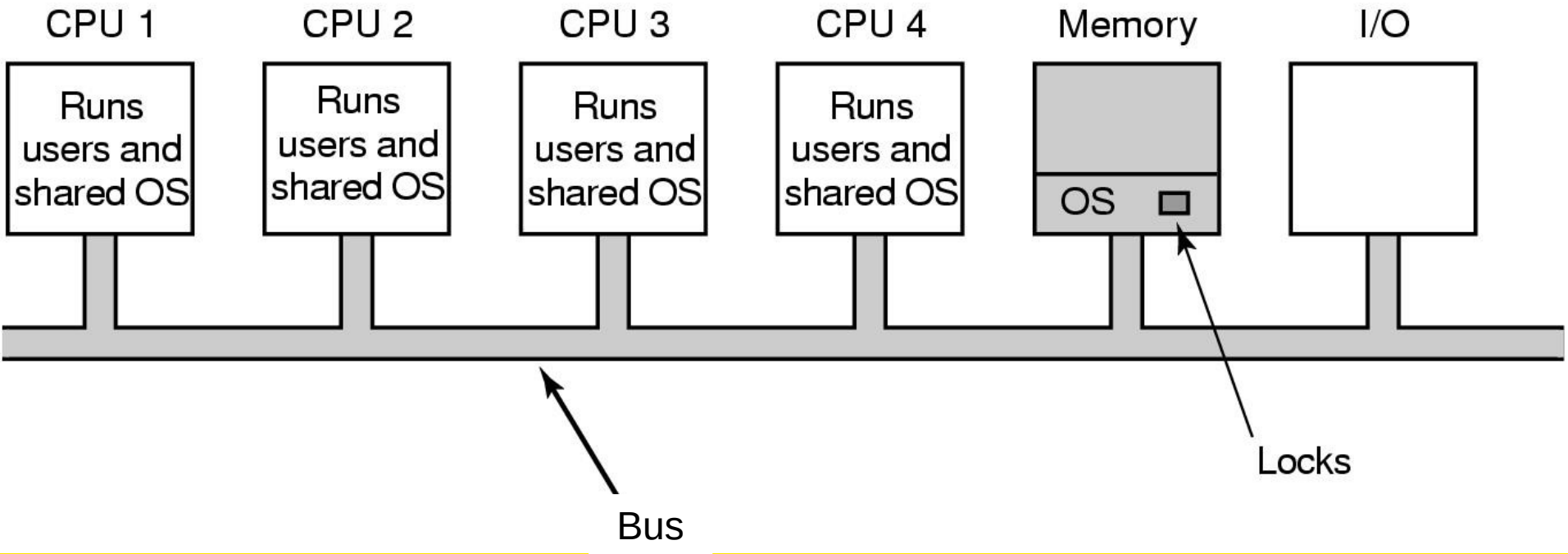
Issues

- Each processor has its own scheduling queue
  - We can have one processor overloaded, and the rest idle
- Each processor has its own memory partition
  - We can a one processor thrashing, and the others with free memory
    - No way to move free memory from one OS to another



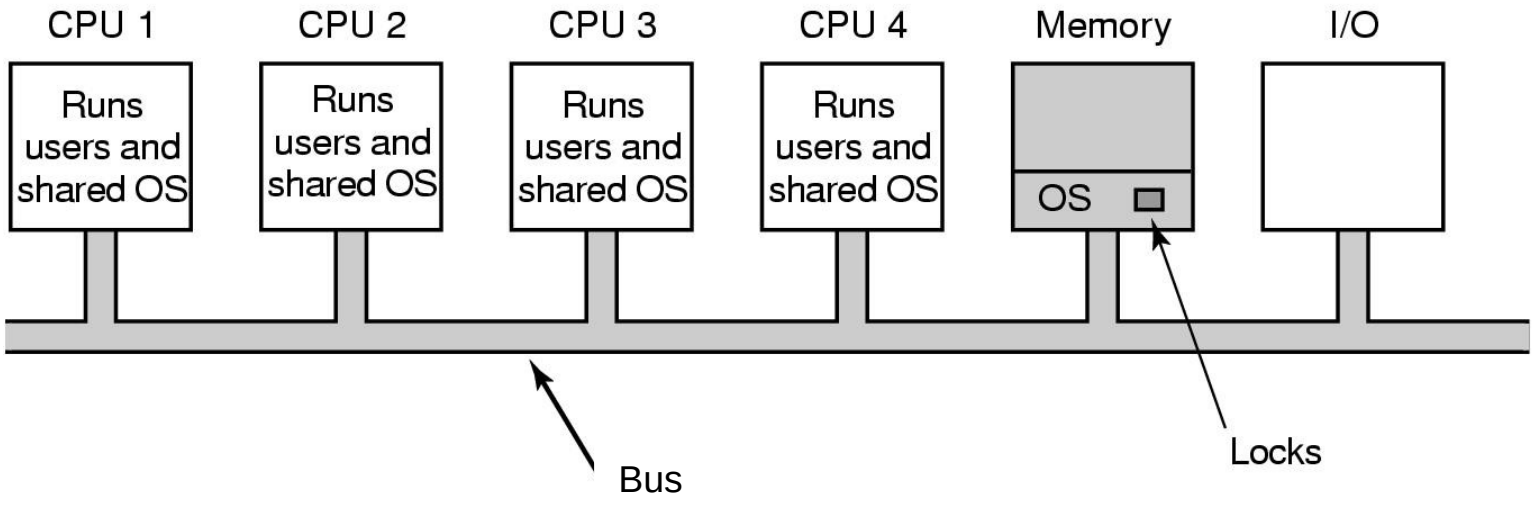
Symmetric Multiprocessors (SMP)

- OS kernel runs on all processors
  - Load and resources are balanced between all processors
    - Including kernel execution
- Issue: *Real* concurrency in the kernel
  - Need carefully applied synchronisation primitives to avoid disaster



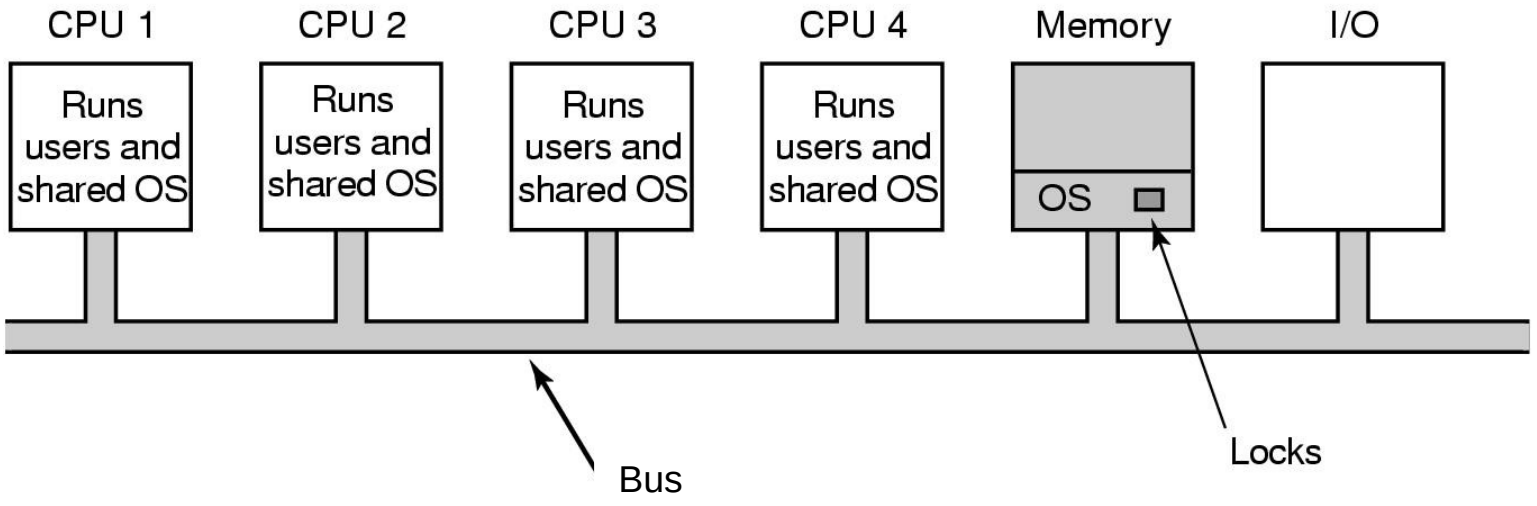
Symmetric Multiprocessors (SMP)

- One option: A single mutex that make the entire kernel a large critical section
  - Only one CPU can be in the kernel at a time
  - The “big lock” becomes a bottleneck when in-kernel processing exceeds what can be done on a single CPU



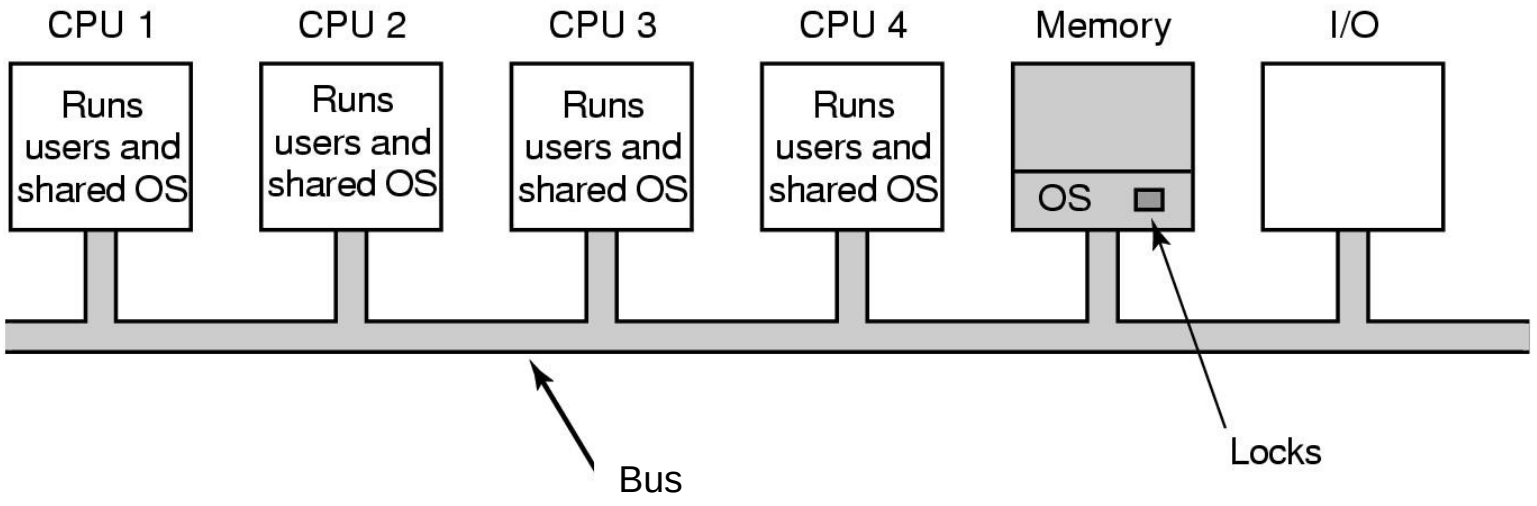
Symmetric Multiprocessors (SMP)

- Better alternative: identify largely independent parts of the kernel and make each of them their own critical section
  - Allows more parallelism in the kernel
- Issue: Difficult task
  - Code is mostly similar to uniprocessor code
  - The hard part is identifying independent parts of the OS
    - Remember all the inter-dependencies between OS subsystems.



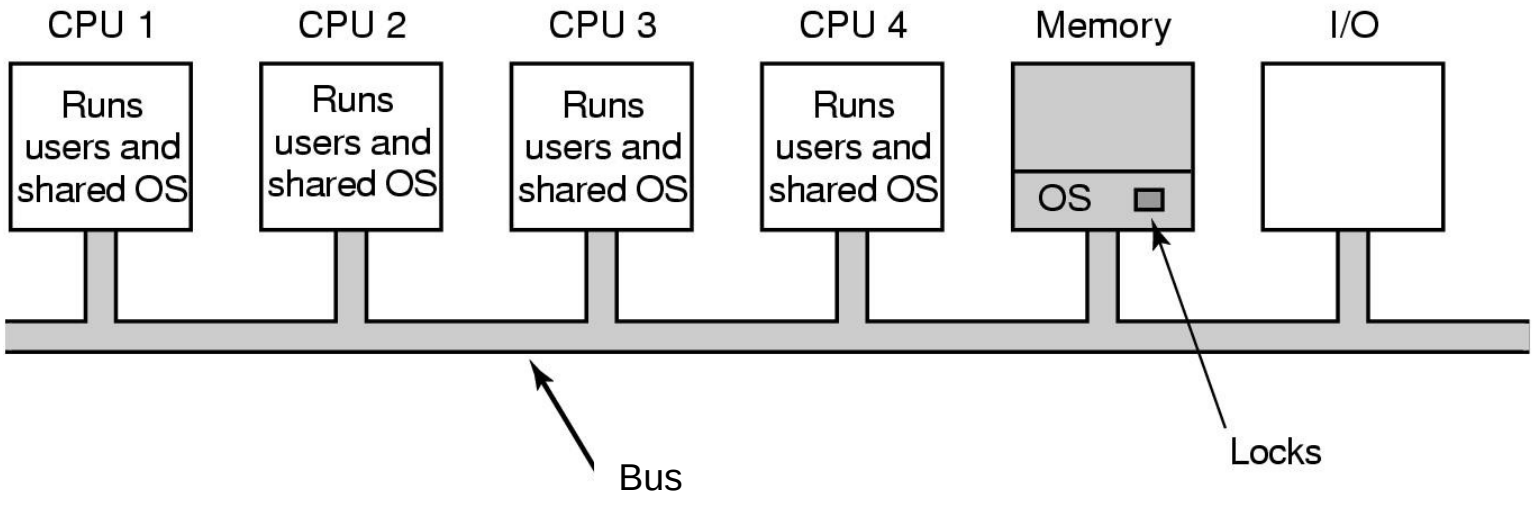
Symmetric Multiprocessors (SMP)

- Example:
  - Associate a mutex with independent parts of the kernel
  - Some kernel activities require more than one part of the kernel
    - Need to acquire more than one mutex
    - Great opportunity to deadlock!!!!
  - Results in potentially complex lock ordering schemes that must be adhered to



Symmetric Multiprocessors (SMP)

- Example:
  - Given a “big lock” kernel, we divide the kernel into two independent parts with a lock each
    - Good chance that one of those locks will become the next bottleneck
    - Leads to more subdivision, more locks, more complex lock acquisition rules
      - Subdivision in practice is (in reality) making more code multithreaded (parallelised)



Real life Scalability Example

- Early 1990's, CSE wanted to run 80 X-Terminals off one or more server machines
- Winning tender was a 4-CPU bar-fridge-sized machine with 256M of RAM
  - Eventual config 6-CPU and 512M of RAM
  - Machine ran fine in all pre-session testing

Real life Scalability Example

- Students + assignment deadline = machine unusable

Real life Scalability Example

- To fix the problem, the tenderer supplied more CPUs to improve performance (number increased to 8)
  - No change????

- Eventually, machine was replaced with
  - Three 2-CPU pizza-box-sized machines, each with 256M RAM
  - Cheaper overall
  - Performance was dramatically improved!!!!!
  - Why?

Real life Scalability Example

- Paper:
  - Ramesh Balan and Kurt Gollhardt, “A Scalable Implementation of Virtual Memory HAT Layer for Shared Memory Multiprocessor Machines”, Proc. 1992 Summer USENIX conference

- The 4-8 CPU machine hit a bottleneck in the single threaded VM code
  - Adding more CPUs simply added them to the wait queue for the VM locks, and made others wait longer
- The 2 CPU machines did not generate that much lock contention and performed proportionally better.

Lesson Learned

- Building scalable multiprocessor kernels is hard
- Lock contention can limit overall system performance

SMP Linux similar evolution

- Linux 2.0 Single kernel big lock (1996)
- Linux 2.2 Big lock with interrupt handling locks
- Linux 2.4 Big lock plus some subsystem locks
- Linux 2.6 most code moved outside the big lock, data-based locking, lots of scalability tuning, etc..
- Big lock removed in kernel version 2.6.39
  - released 2011

Multiprocessor Synchronisation

- Given we need synchronisation, how can we achieve it on a multiprocessor machine?
  - Unlike a uniprocessor, disabling interrupts does not work.
    - **It does not prevent other CPUs from running in parallel**
  - Need special hardware support

Recall: Mutual Exclusion via Test-and-Set

```
enter_region:
    TSL r0, lock          | copy lock to r0, 1 to lock
    CMP r0, #0            | compute whether lock was 0
    JNE enter_region      | if not, loop again
    RET                   | done, return to caller


leave_region:
    MOVE lock, #0         | store 0 (unlocked) in lock
```

Entering and leaving a critical region using the TSL instruction.

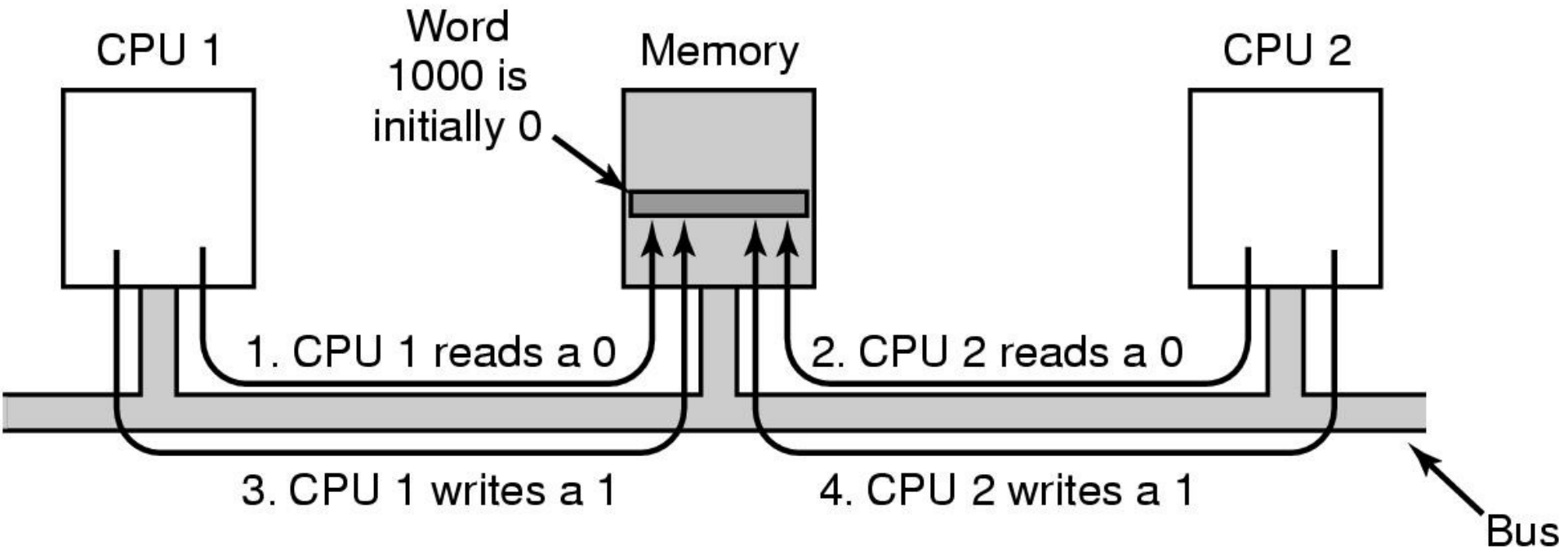
(From week 1 concurrency slides.)

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

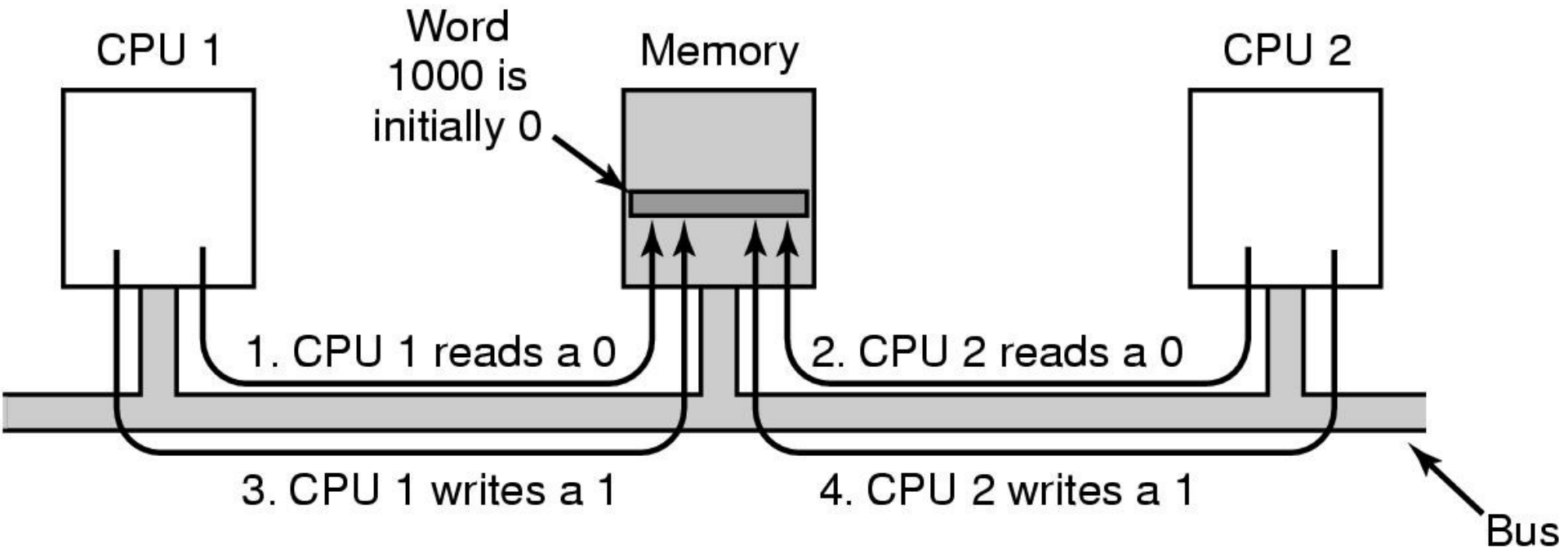
Test-and-Set on SMP

- It does not work without some extra hardware support



Test-and-Set on SMP

- A solution:
  - Hardware blocks other CPUs from accessing the bus during the TSL instruction to prevent memory accesses by any other CPU.
    - TSL has mutually exclusive access to memory for duration of instruction.

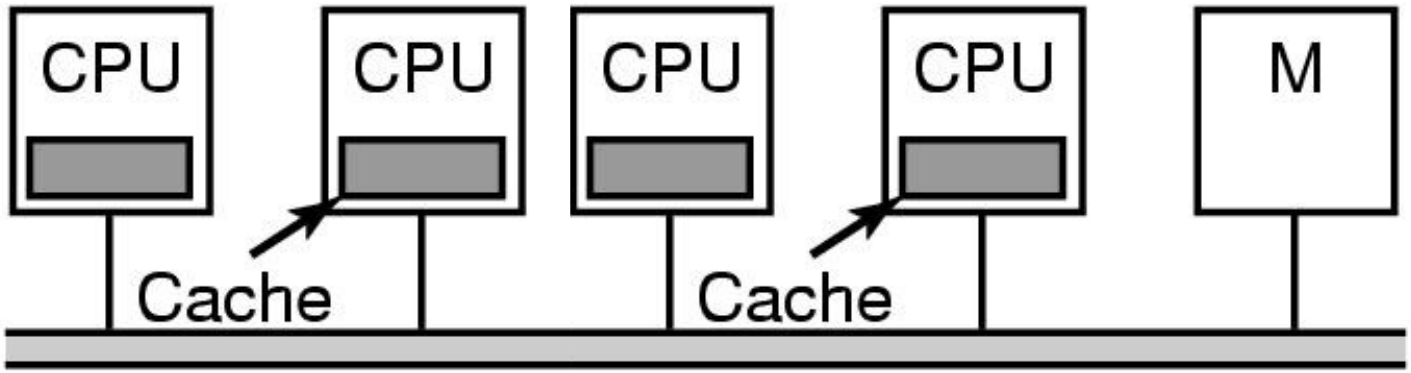


Test-and-Set on SMP

- Test-and Set is a busy-wait synchronisation primitive
  - Called a ***spinlock***
- Issue:
  - Lock contention leads to spinning on the lock
    - Spinning on a lock requires blocking the bus which slows all other CPUs down
      - Independent of whether other CPUs need a lock or not
      - Causes bus contention

Test-and-Set on SMP

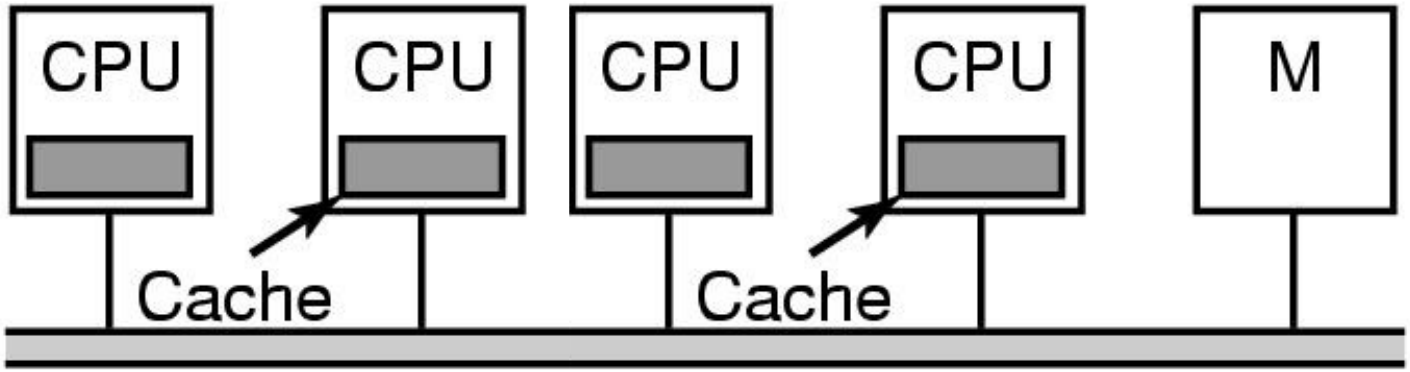
- Caching does not help reduce bus contention.
  - Either TSL blocks the bus.
  - Or TSL requires exclusive access to the lock in the local cache.
    - Doesn't solve the performance problem.
    - Detailed explanation to come in later slides.



Reducing 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;
```



Thomas Anderson, “The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors”, *IEEE Transactions on Parallel and Distributed Systems*, Vol 1, No. 1, 1990

Comparison of Simple Spinlocks

- Test and Set

```
void lock (volatile lock_t *l) {
 while (test_and_set(l)) ;
}
```

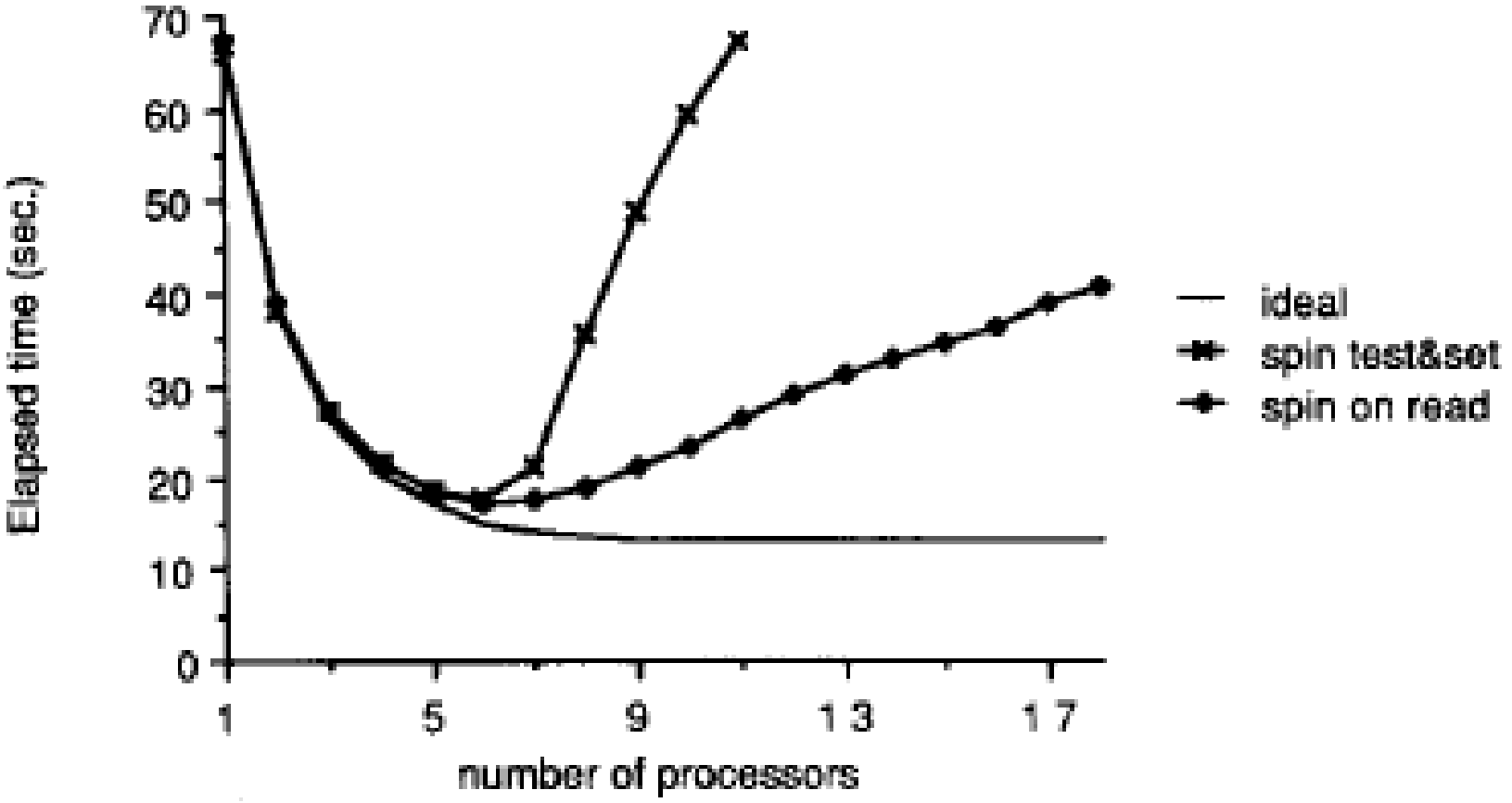
- Read before Test and Set

```
void lock (volatile lock_t *l) {
 while (*l == BUSY || test_and_set(l)) ;
}
```

Benchmark

```
for i = 1 .. 1,000,000 {
  lock(l)
  crit_section()
  unlock()
  compute()
}
```

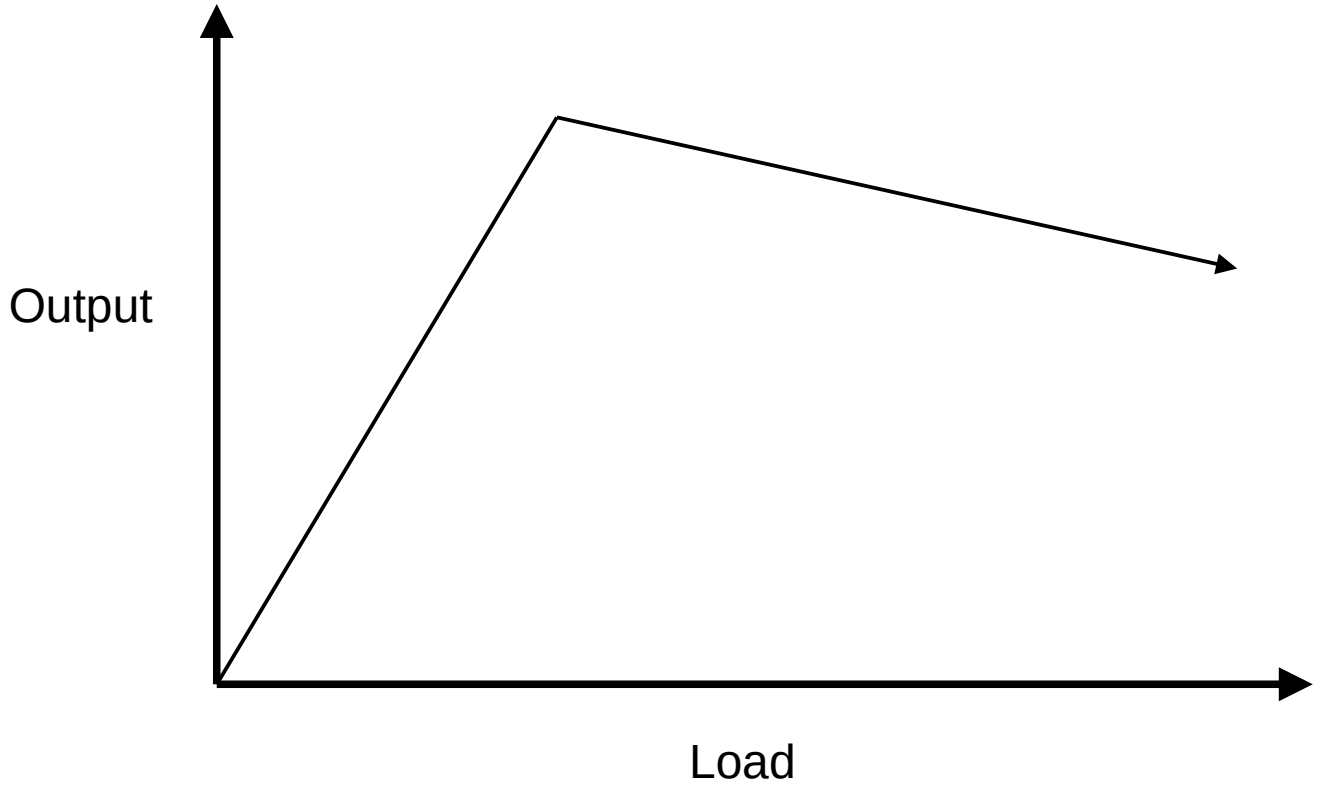
- Compute chosen from uniform random distribution of mean 5 times critical section
- Measure elapsed time on Sequent Symmetry (20 CPU 30386, coherent write-back invalidate caches)


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

Results (of Anderson's Investigation)

- Test and set performs poorly once there are enough CPUs to cause contention for lock.
  - As expected.
- Read before Test and Set performs better.
  - Performance less than was expected.
  - There is still significant contention on lock when CPUs notice release and all attempt acquisition.
- Critical section performance degenerates.
  - Critical section requires bus traffic to modify shared structure.
  - Lock holder competes with CPU that's waiting as they test and set, so the lock holder is slower.
  - Slower lock holder results in more contention.

A Performance Pattern



Explaining Invalidate Contention

- Let's have a look at the difference between the test-and-set lock and the test-and-test-and-set lock in the case where the caches implement exclusive access.
- We will do this by showing an example cache protocol.
  - Note: this protocol is entirely made up for the purposes of explanation.

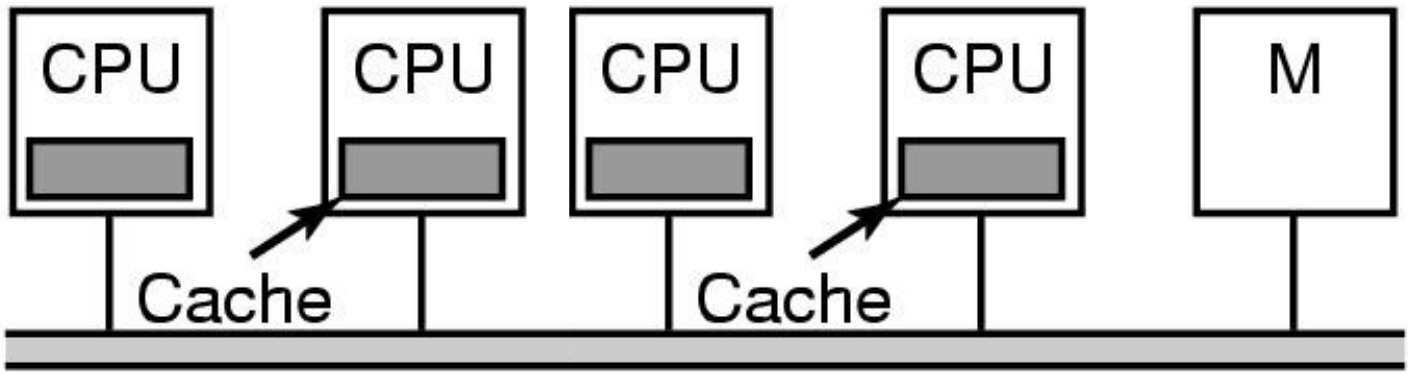
Cache State (Invented)

- 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

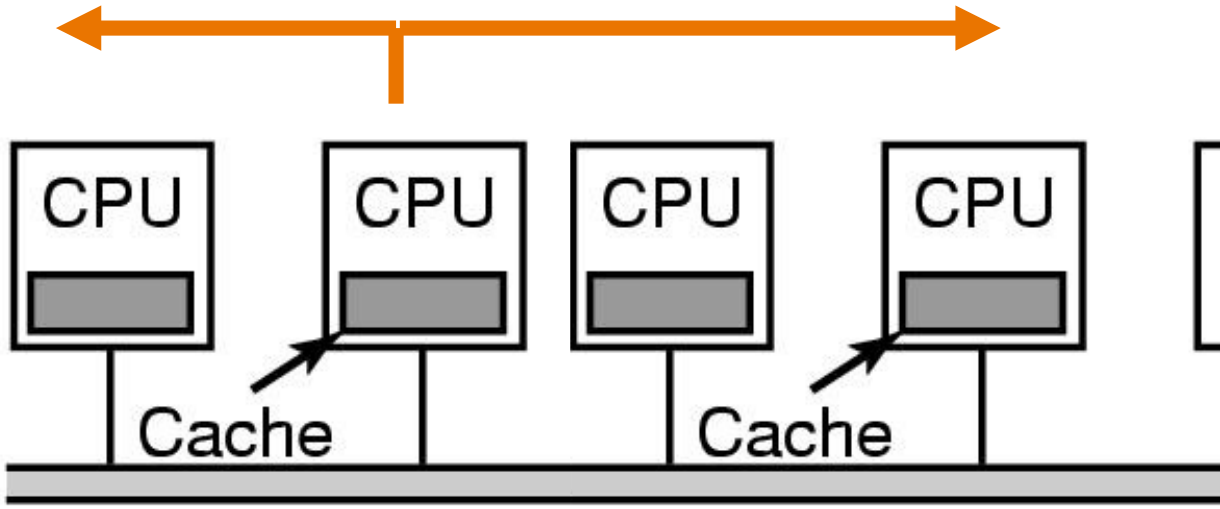
- Write @ 0x1f00

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

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

- CPU 1 broadcasts
  - INVAL(ALL) @ 0x1f00
- CPU 2
  - ACK

What is the new cache state?



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

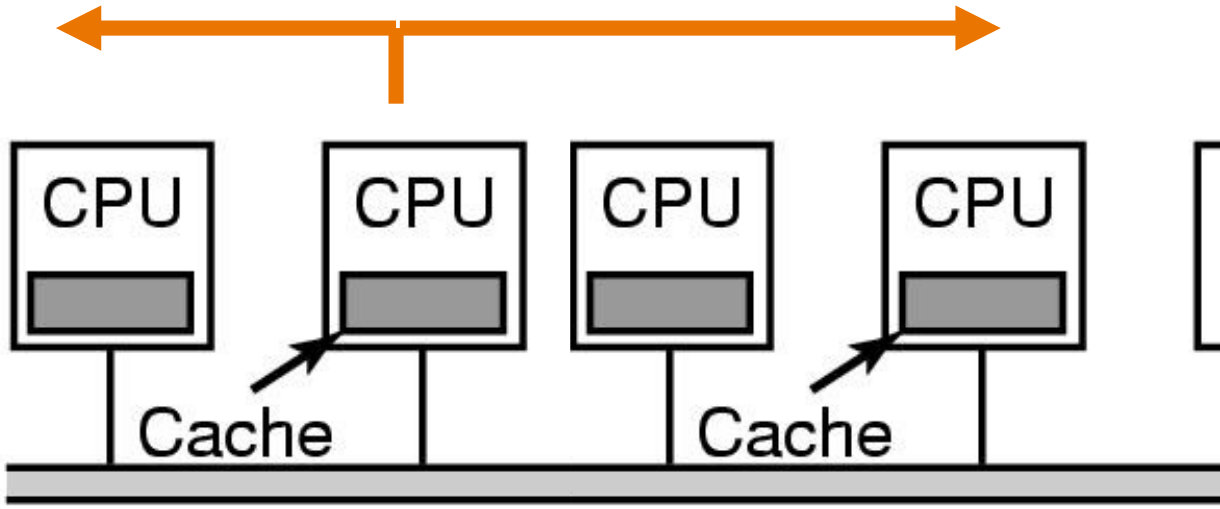
- Read @ 0x1f00

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

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

- CPU 1 broadcasts
  - INVAL(EXCL) @ 0x1f00
- CPU 2
  - ACK
- CPU 1 can read memory.

  What is the new cache state?



Invalidate to Read II

CPU 1

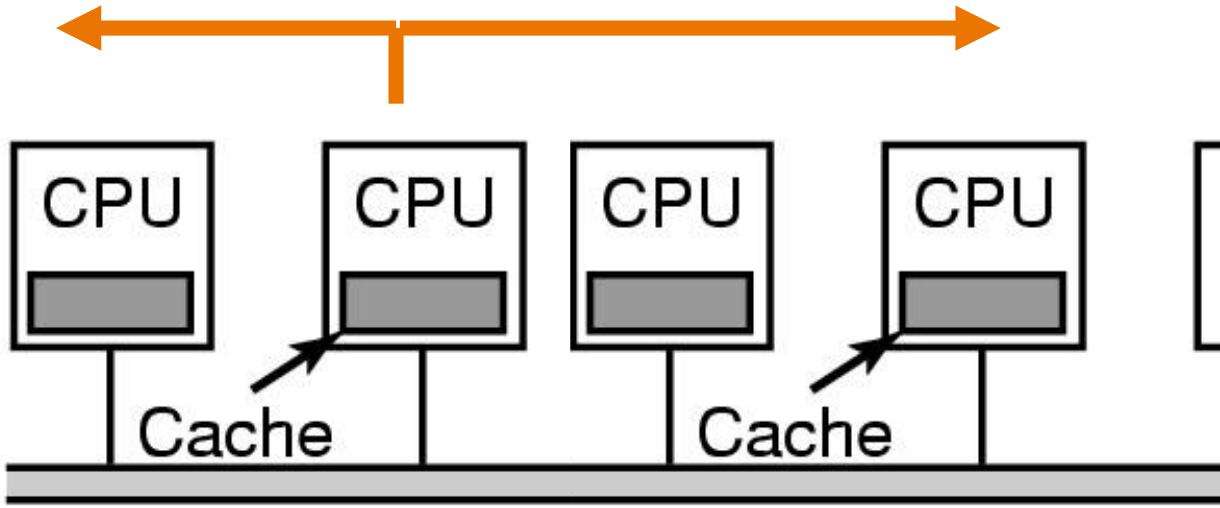
- Read @ 0x1f00

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

- CPU 1 broadcasts
  - INVAL(EXCL) @ 0x1f00

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

- CPU 2
  - WAIT!!
- Execution waits for the dirty cache contents to be written to memory.



Lock Status Bit

R/V	W/X	D	L	V Addr	Contents

- What about the lock status bit?
- Like D, 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

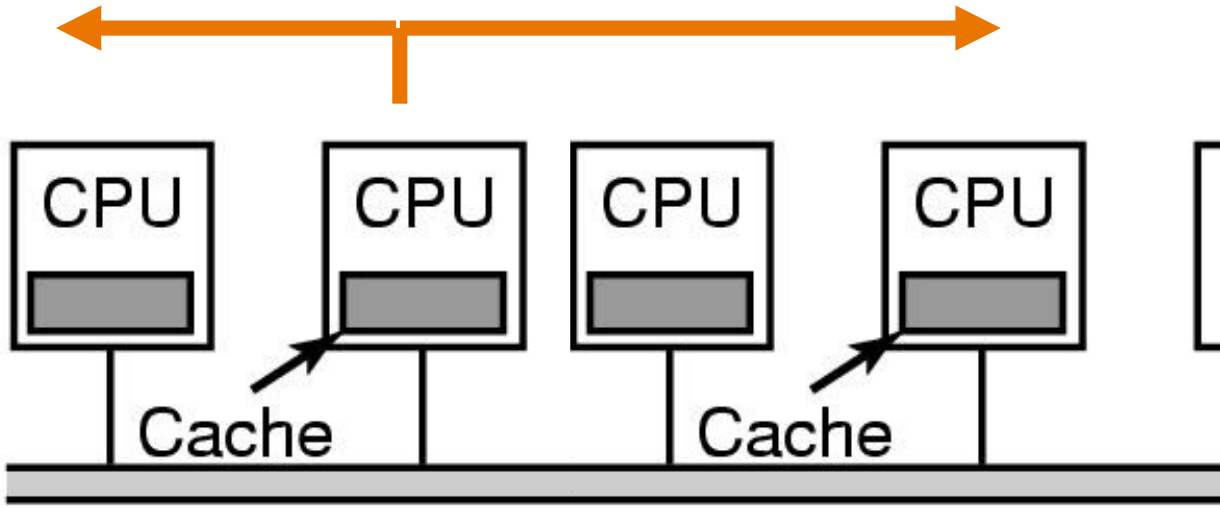
- TSL @ 0x1f00
- INVAL(ALL)&Lock

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

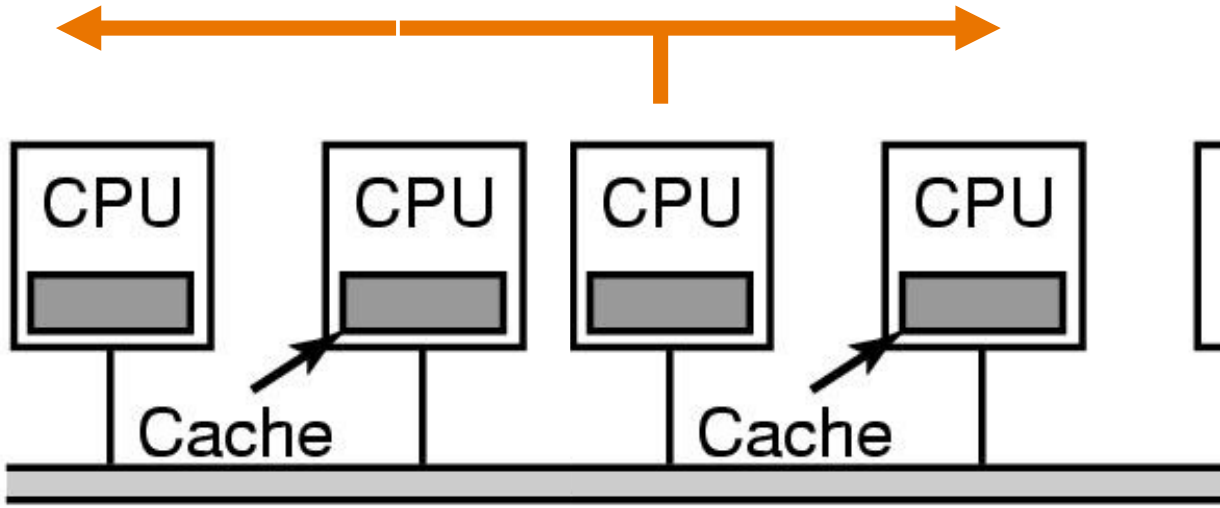
- Has lock, moves on

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

Unlock-Time Invalidate Contention

- As Anderson discovered, the test-and-test-and-set lock also has performance issues for many CPUs/cores.
- The problem is the contest between tasks waiting for the lock when it is released.

- Some additional lock types avoid this problem.
  - Ticket locks
  - MCS/Queue locks

  - Note that ticket locks, MCS/Queue locks and the invented cache protocol are all “bonus content” not assessible this year.

Ticket Lock

Consists of two numbers.

- **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

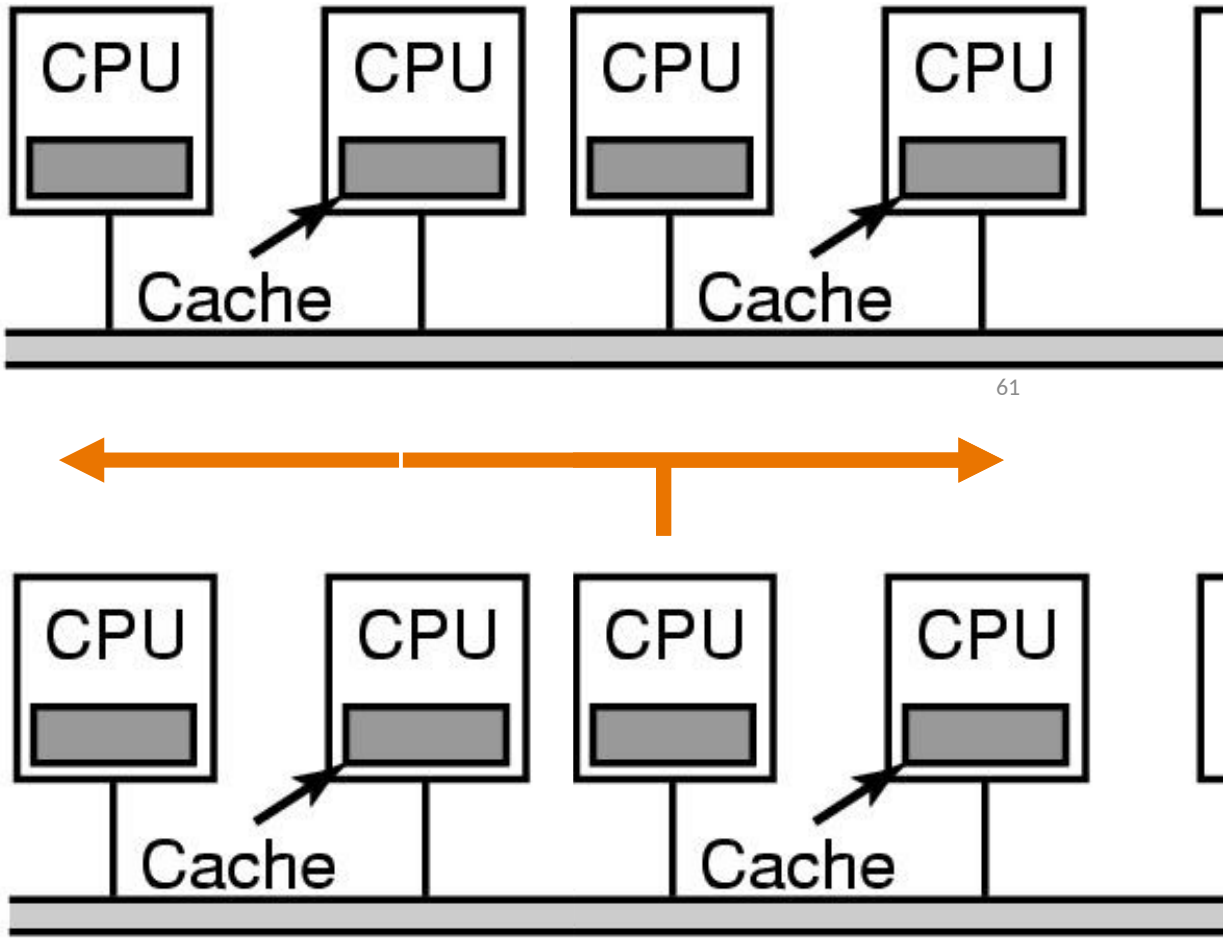
- Gets ticket 2, waits

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

Global Invalidate Issues

- With many CPUs/cores, (e.g. 16-64), global operations such as invalidate become a problem.
- Can be improved with “directory” cache coherence approaches.
- Ideal case is where two cores hand over a cache line *privately*.

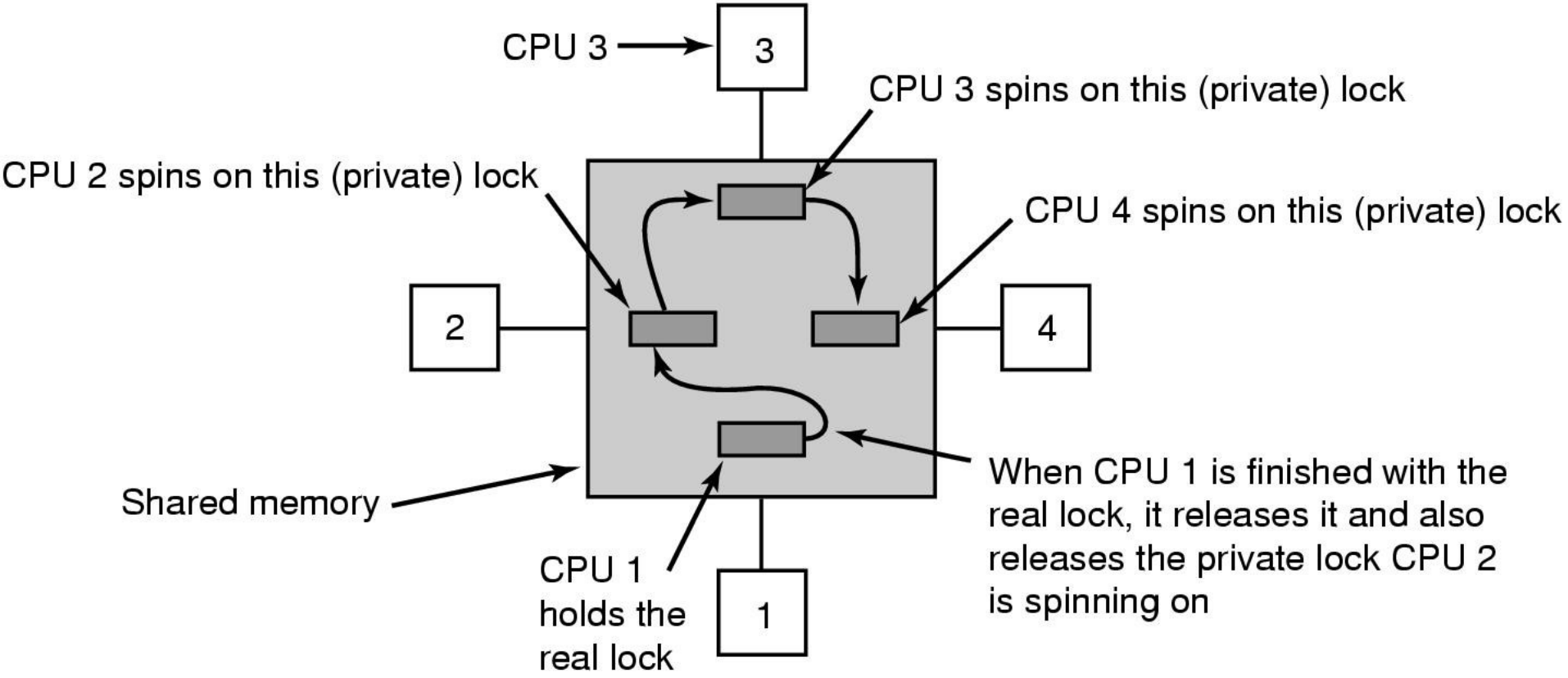


61

MCS (Queue) Lock

- Introduced by Mellor-Crummey and Scott.
- CPUs/cores that need to wait construct a linked list which structures the wait queue, using atomic operations.
- Each CPU waits on its own variable in its own structure in the queue.
  - No contention.
- On lock release, the releaser signals the next CPU by writing queue element.
  - This write does not have to broadcast.
  - No starvation (order of lock acquisitions defined by the list)

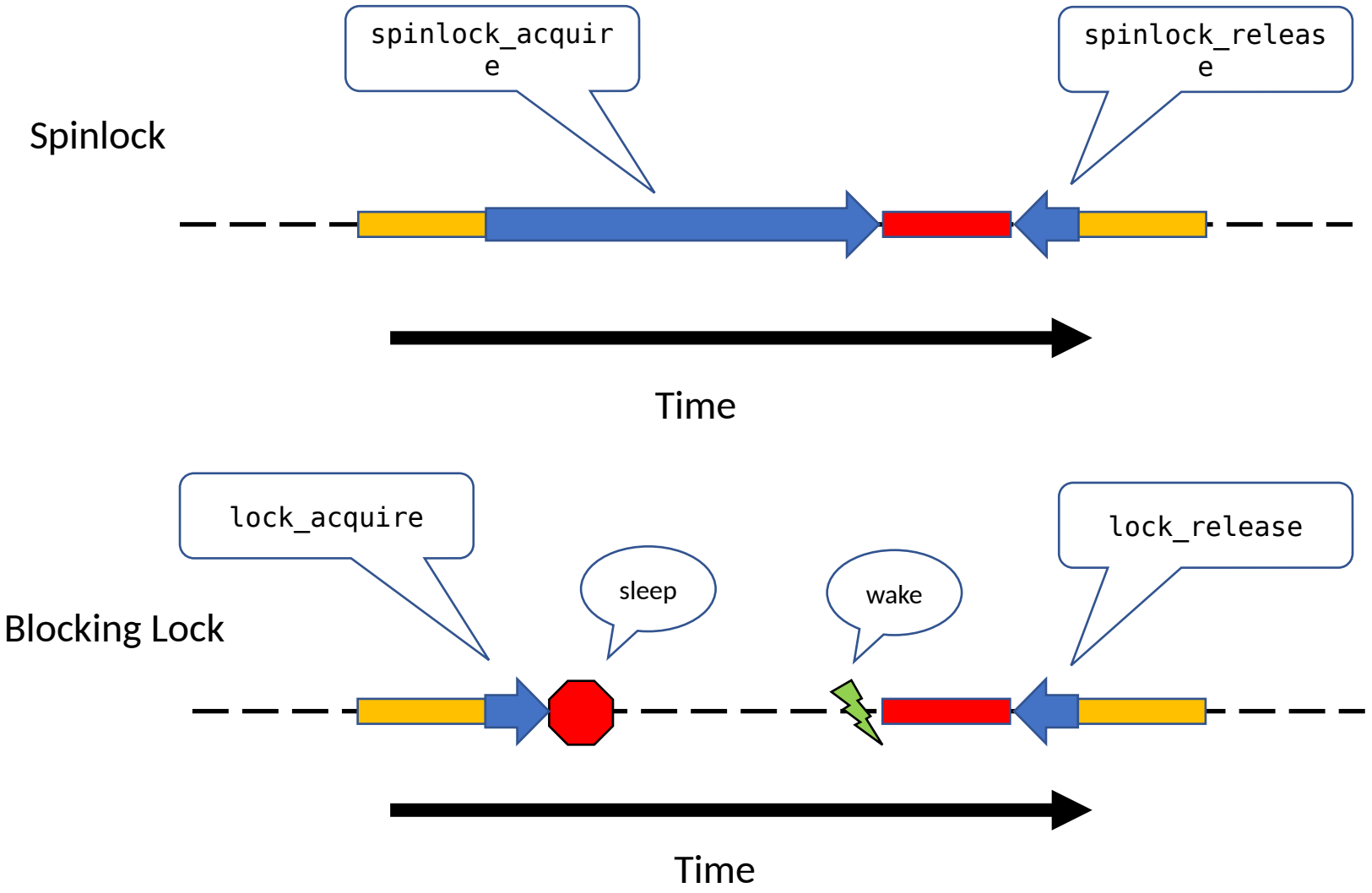
MCS (Queue) Lock



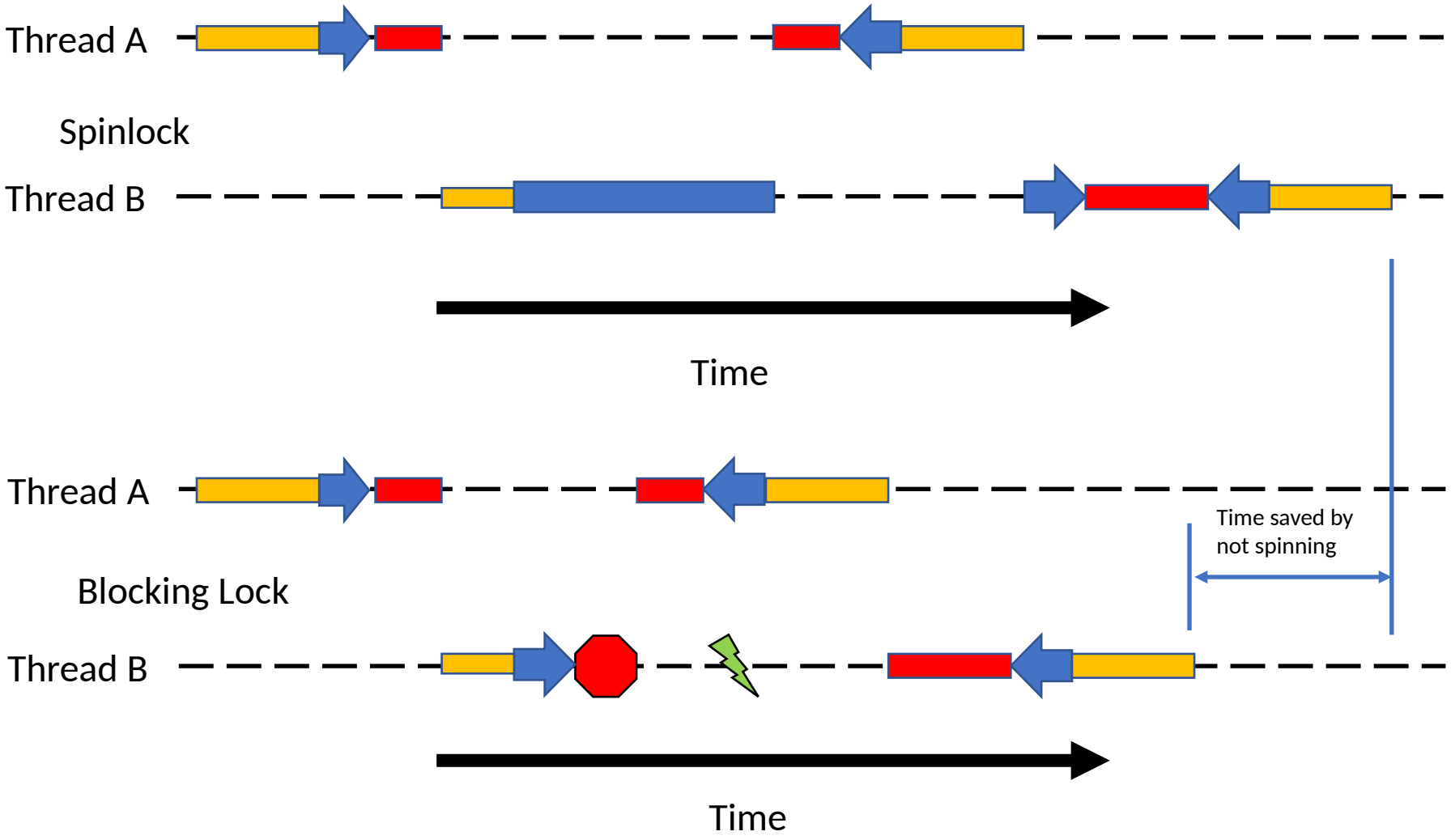
MCS Lock

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

Spinning Locks versus Blocking Locks



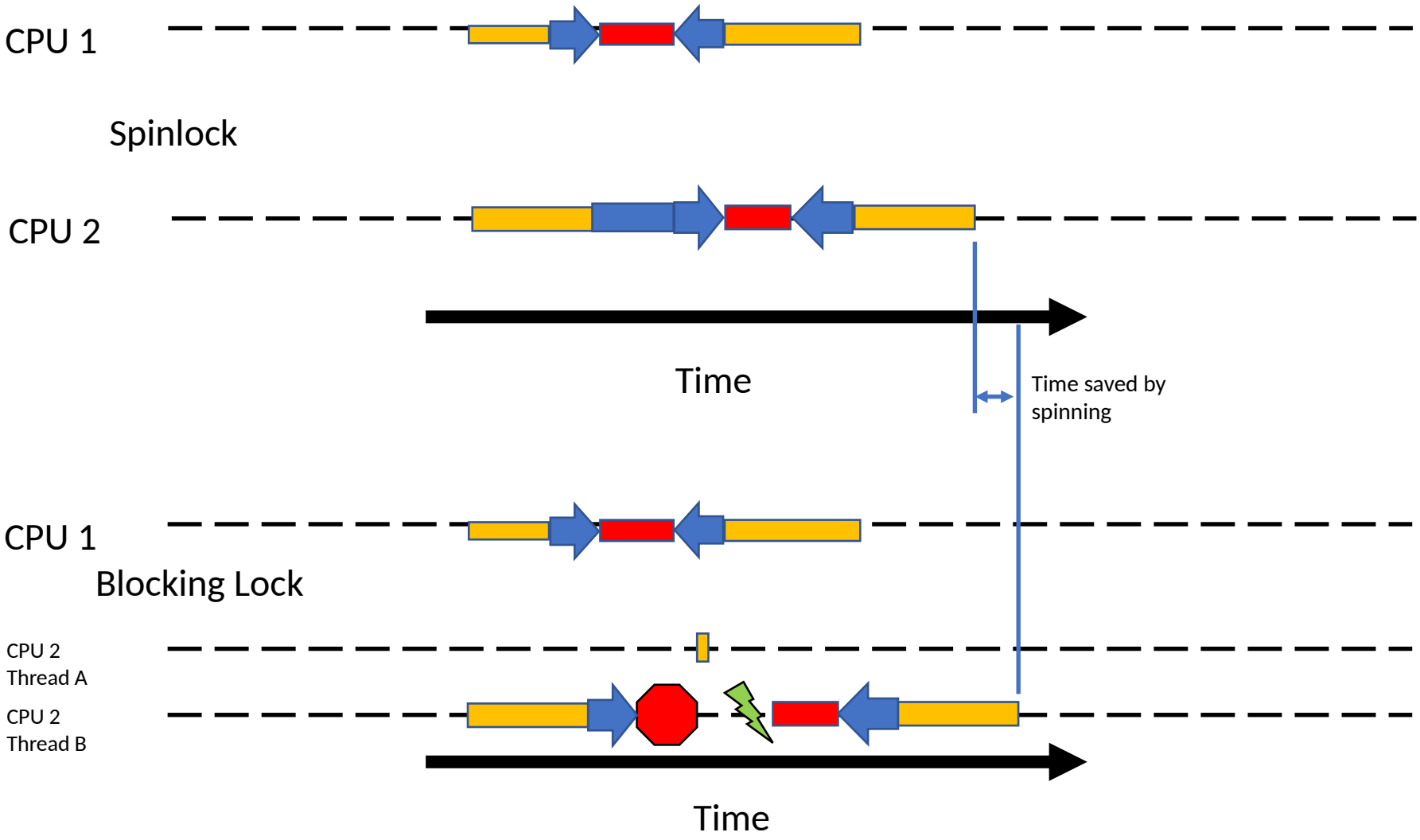
Uniprocessor: Spinning versus Blocking



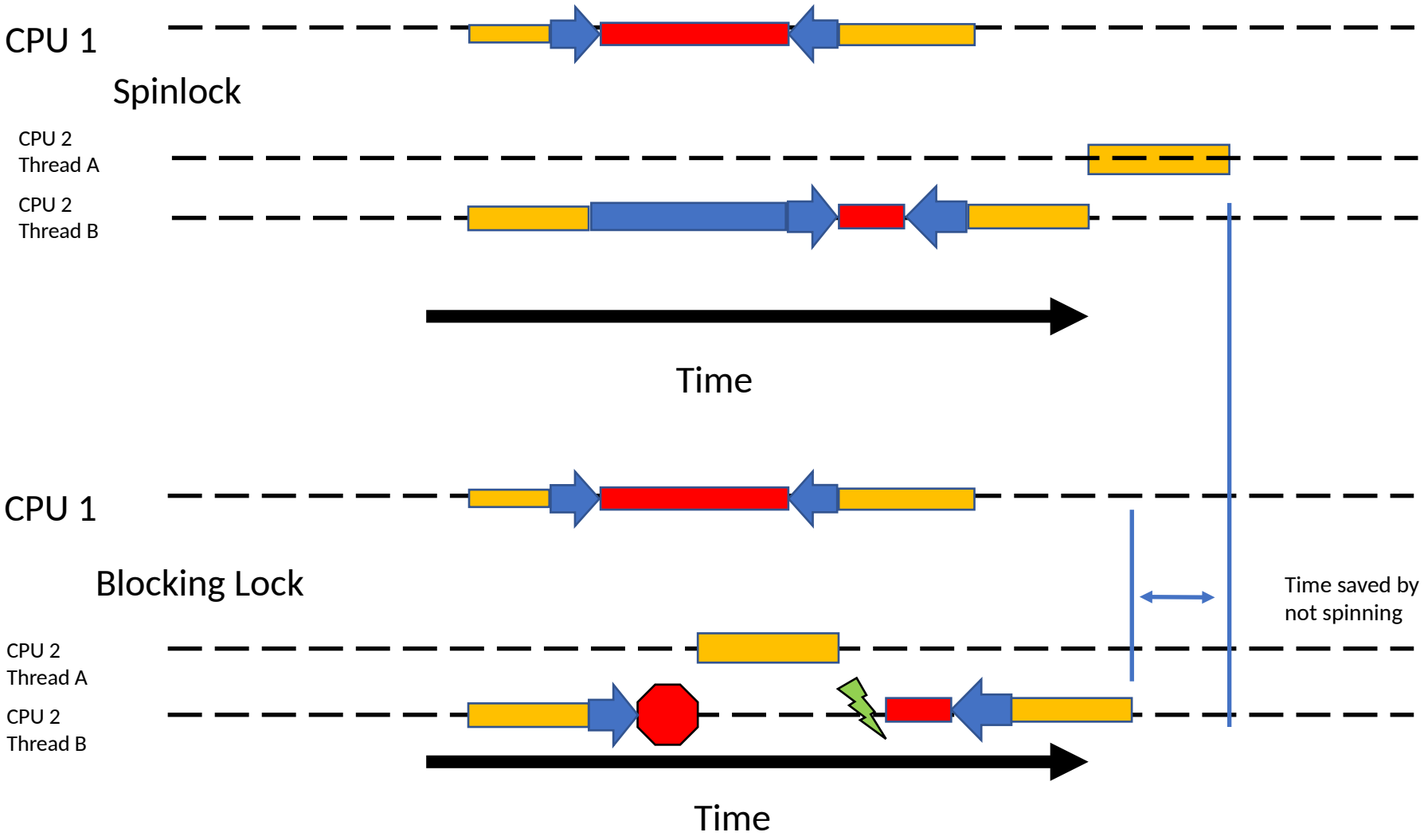
Spinning versus Blocking and Switching

- 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

This Lecture: Multiprocessing

- Need for multi-core and multi-processor systems.
  - Moore's law and MHz and similar.
- Machine design, consistency and bandwidth challenges.
- OS design challenges.
- Synchronisation challenges on true multi-processors.
  - Locks and cache coherence.
  - Spinning vs waiting tradeoffs.