School of Computer Science & Engineering

COMP9242 Advanced Operating Systems

2025 T3 Week 04

Multicore, Memory Ordering, Locking

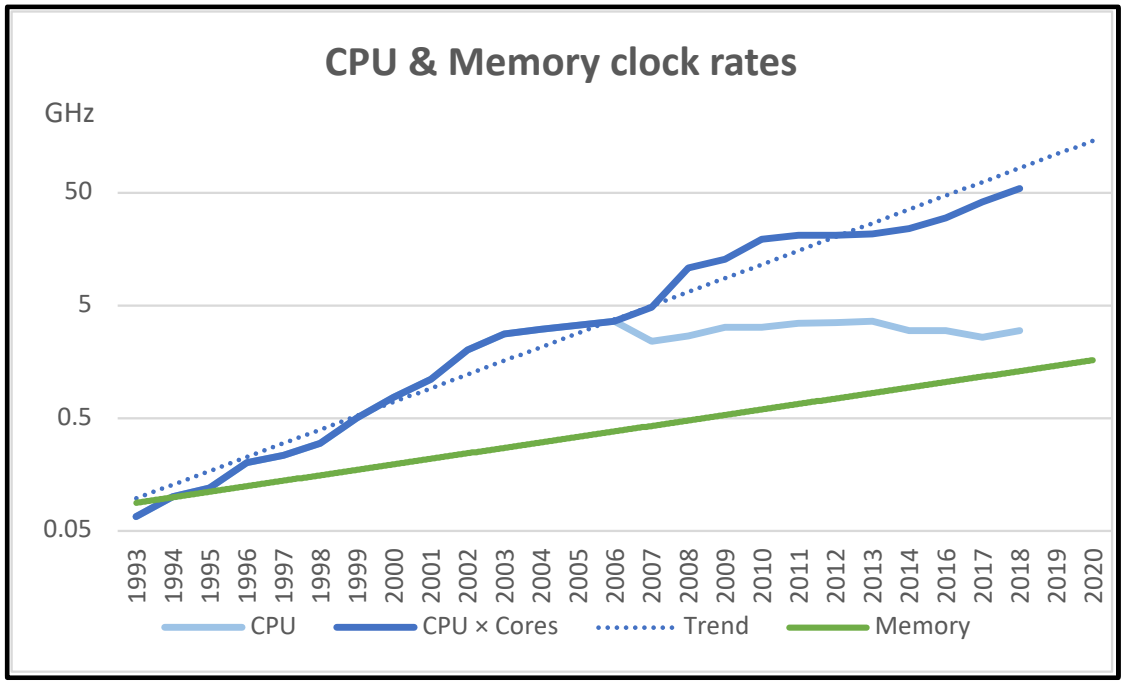
Gernot Heiser

Copyright Notice

Today's Lecture

- Multicore cache coherency challenges
- Hardware-based cache coherency
- Memory ordering
- Locking primitives
- Locking in microkernels

Reminder: Memory Wall



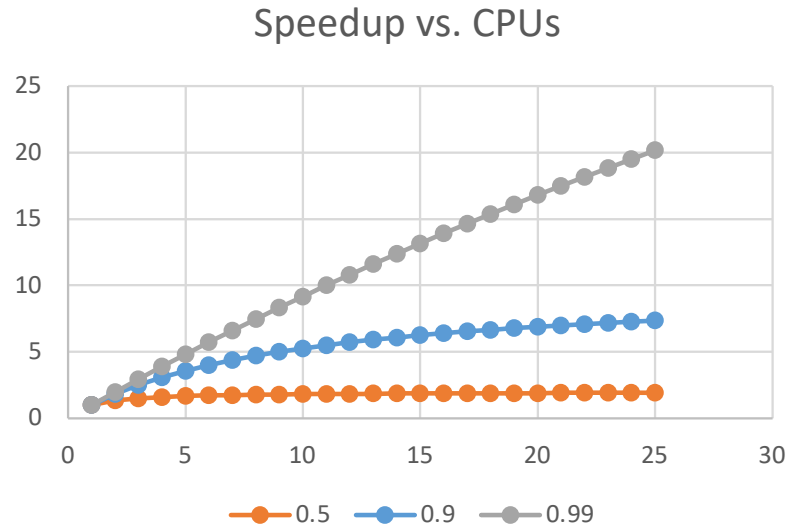
Speed gap still widens by approx 18% per year!

The Age of Multicores

- CPU Clock rates stopped growing early in the century
- Feature sizes kept shrinking
- Architects used chip area for increasing number of cores
- Can use for:
  - increasing degree of multiprogramming
  - parallelise applications

Amdahl's Law

- Given:
  - Parallelisable fraction P
  - Number of processor N
  - Speed up S
- $S(N) = \frac{1}{(1-P)+\frac{P}{N}}$
- $S(\infty) = \frac{1}{(1-P)}$
- Parallel computing takeaway:
  - Useful for small numbers of CPUs (N)
  - Or, high values of P

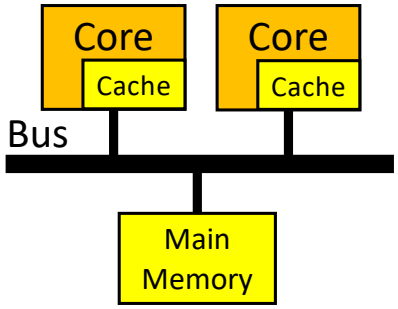


OS has no choice – needs to support all cores!

Types of Multiprocessors

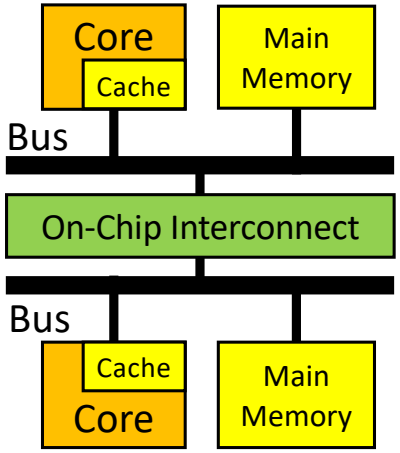
Independent systems from OS PoV

Symmetric multi-processor (SMP)



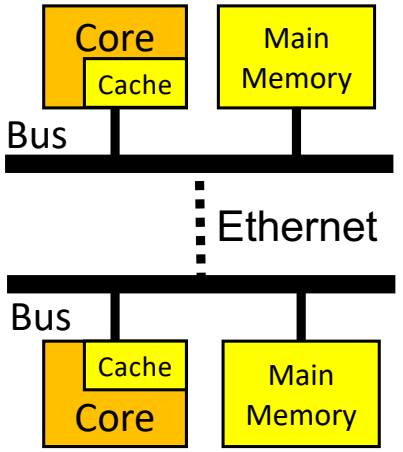
- Local caches
- Unified memory

Non-uniform memory architecture (NUMA)



- Local caches
- Local memory, globally accessible
- Locality reduces contention

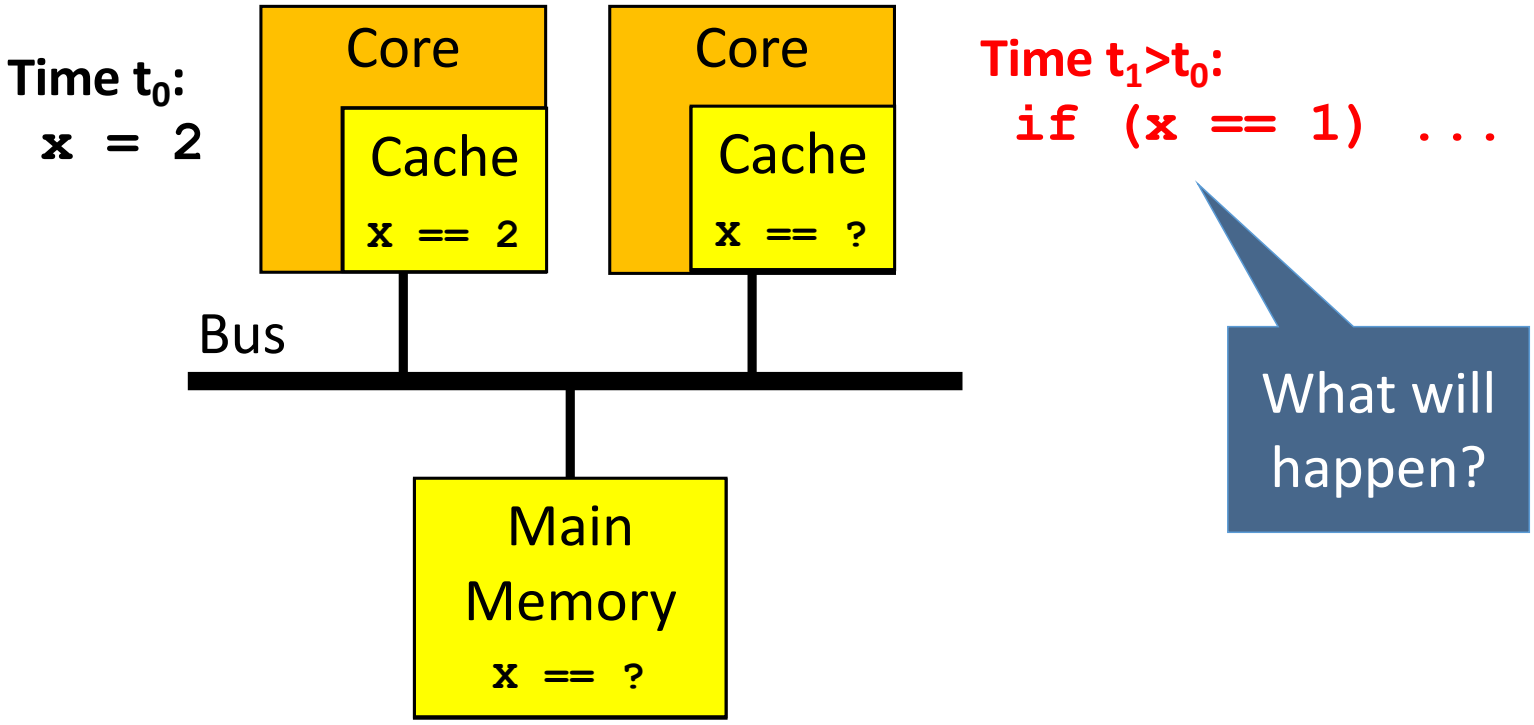
Distributed System



- Local caches
- Local memory

HW-Based Cache Coherency

Cache Coherency



Problematic Example

Assume initially a=b=0

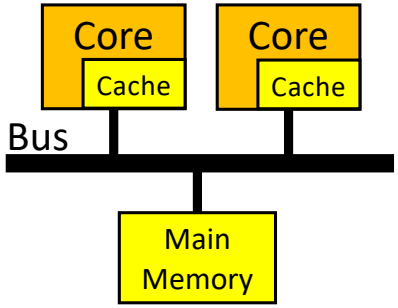
CPU A

```
a = 1
if (b == 0) then {
  /* critical section */
  a = 0
} else {
...
```

CPU B

```
b = 1
if (a == 0) then {
  /* critical section */
  b = 0
} else {
...
```

What can go wrong?



Memory Model: Sequential Consistency

"The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]

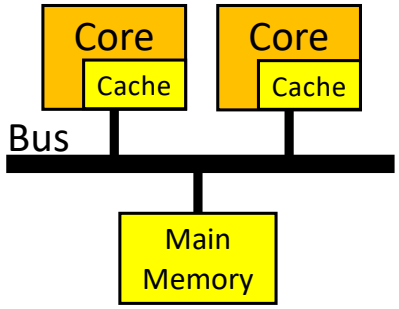
With Sequential Consistency

CPU A

```
a = 1
if (b == 0) then {
  /* critical section */
  a = 0
} else {
...
```

CPU B

```
b = 1
if (a == 0) then {
  /* critical section */
  b = 0
} else {
...
```



Sequential consistency
- ensures correct behaviour
- requires coherent caches

Keeping Caches Coherent: Write-Through

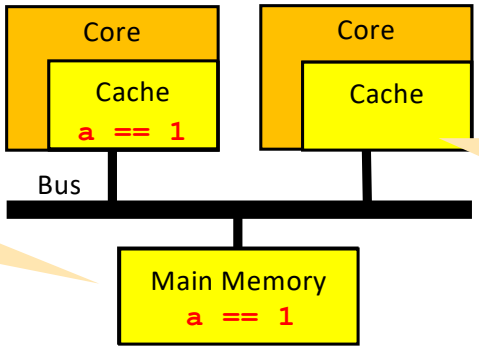
CPU A

```
a = 1
if (b == 0) then {
  /* critical section */
  a = 0
} else {
...
```

CPU B

```
b = 1
if (a == 0) then {
  /* critical section */
  b = 0
} else {
...
```

✓Cache coherent with memory
❖Long write latency



Write immediately propagated to memory

Core snoops on bus, invalidates stale lines

How Keep Caches Coherent??

Snooping caches:

- Needs write-through caches
- Needs cheap “broadcast” to all cores

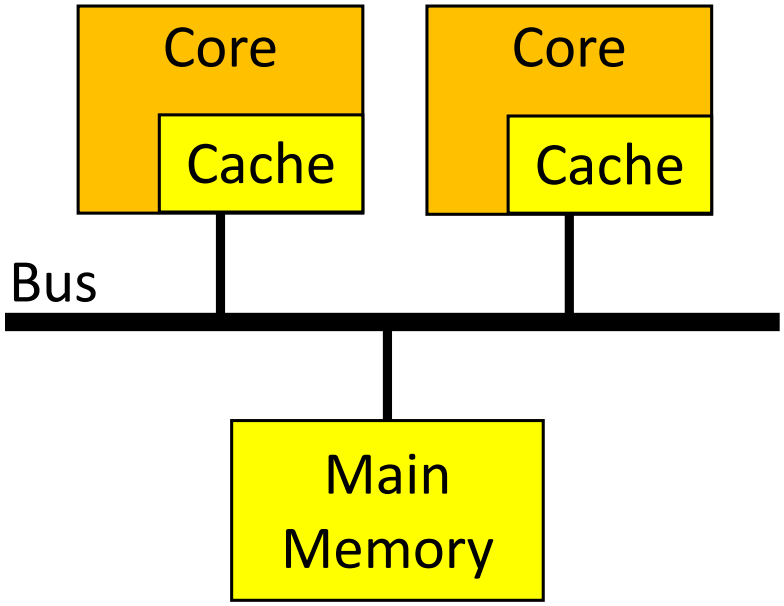
Alternatives:

- Better performance by more complexity, less strict assumptions
- Use memory bus for messaging between local caches
- Example: MESI & variants

MESI Cache Coherence Protocol

Each cache line is in one of four states:

- Invalid (I):
  - Holds no useful data
- Exclusive (E):
  - Line is only in this cache
  - Line is *clean* – consistent with RAM
- Shared (S):
  - Line is in at least one other cache
  - Line is clean
- Modified (M):
  - The line is only in only this cache
  - Line is *dirty* – modified wrt RAM



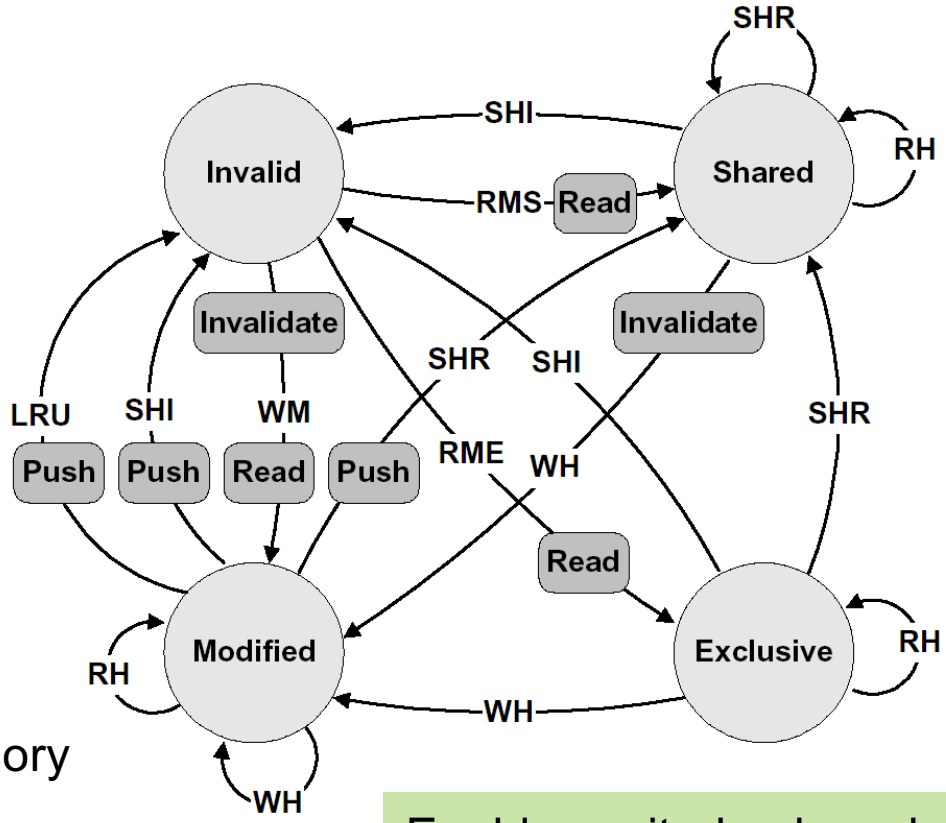
MESI (with snooping & broadcast invalidate)

Events:

- RH = Read Hit
- RMS = Read miss, shared
- RME = Read miss, exclusive
- WH = Write hit
- WM = Write miss
- SHR = Snoop hit on read
- SHI = Snoop hit on invalidate
- LRU = LRU replacement

Bus Transactions:

- Push = Write cache line back to memory
- Invalidate = Broadcast invalidate
- Read = Read cache line from memory



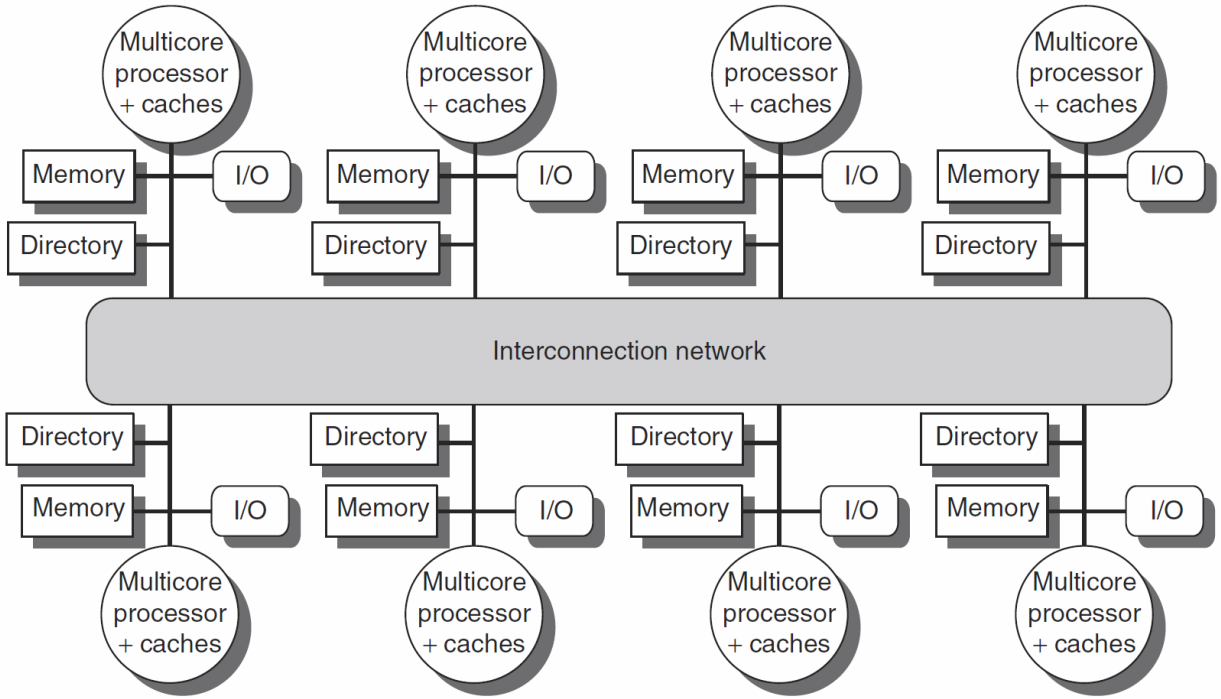
Enables write-back caches
⇒ less memory traffic

NUMA Directory-Based Coherence

Idea: Each node keeps per-line directory of caches that have a copy of the node's memory (bitmap of cores)

Pro: Directed messages instead of snooping & broadcast

Con: High memory overhead



Computer Architecture A Quantitative Approach *Fifth Edition* **John L. Hennessy, David A. Patterson**

Summary: Hardware-Based Coherency

- ✓Consistent view of memory across the machine
- ✓A dirty line is only cached once, clean lines can be replicated
- ✓A **read** will get the result of the last **write** to the memory hierarchy
- ❖Scales poorly

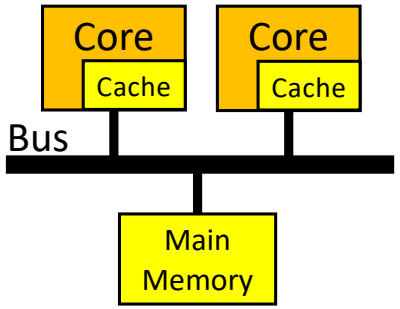
Memory Ordering

Memory Ordering

A tail of a critical section:

```
    /* assuming lock already held */
   /* counter++ */
load r1, counter
add r1, r1, 1
store r1, counter
   /* unlock(mutex) */
store zero, mutex
```

Assumes all cores see update to counter before update to mutex



Memory Model 1: Strong Ordering

Remember: Sequential consistency:

> The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Traditional model

Assume initially x = y = 0

Core 0	Core 1
`store 1, x`	`store 1, y`
`load r1, y`	`load r2, x`

Post-Lecture Comment

Apologies for confusing myself on the next page in the lecture! I've tried to clarify so even I can understand it (and will hopefully get it right next time!)

Potential Interleavings

```
store 1, X
load r1, Y
store 1, Y
load r2, X
r1=0, r2=1
```

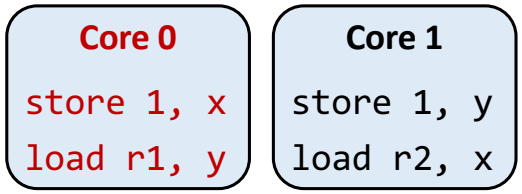
```
store 1, X
store 1, Y
load r1, Y
load r2, X
r1=1,r1=1
```

```
store 1, X
store 1, Y
load r2, X
load r1, Y
r1=1,r2=1
```

```
store 1, Y
load r2, X
store 1, X
load r1, Y
r1=1, r2=0
```

```
store 1, Y
store 1, X
load r2, X
load r1, Y
r1=1,r2=1
```

```
store 1, Y
store 1, X
load r1, Y
load r2, X
r1=1,r2=1
```

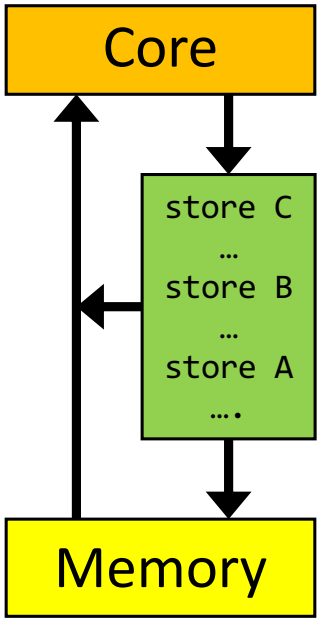


At least one core must load other core's new value – x0=0, y=0 impossible!

Remember the Write Buffer?

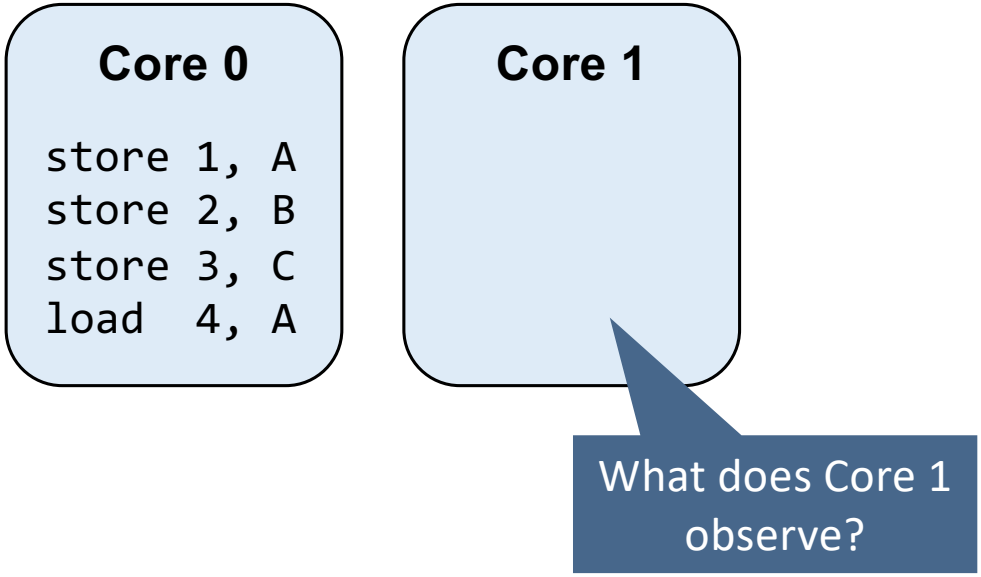
Core

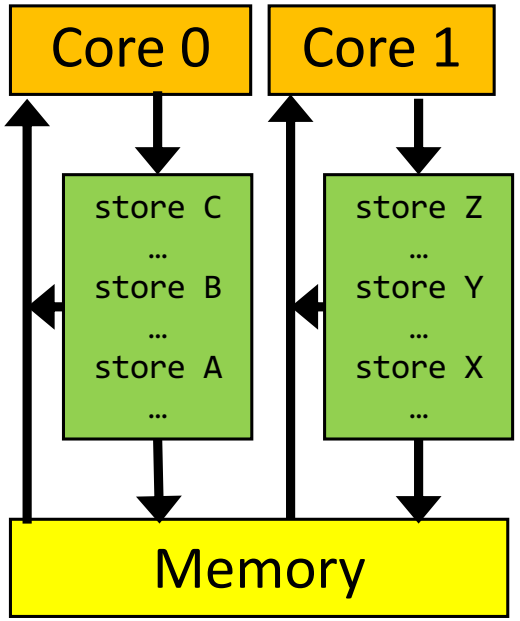
```
store 1, A
store 2, B
store 3, C
load  4, A
```



- Stores & invalidates go to write buffer to hide latency
- Loads read from write buffer

Write Buffer and SMP





Total Store Ordering (eg x86)

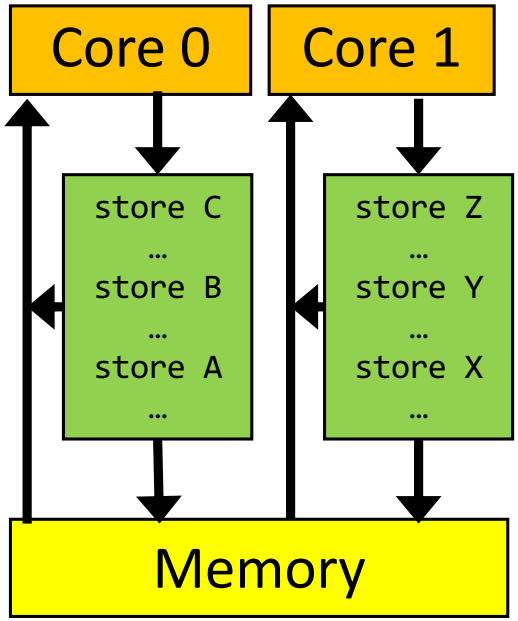
Core 0

```
store 1, A
store 2, B
store 3, C
```

Core 1

```
A = 1
B = 2
C = 3
```

Stores guaranteed to be observed in FIFO order



Total Store Ordering (eg x86)

Core 0

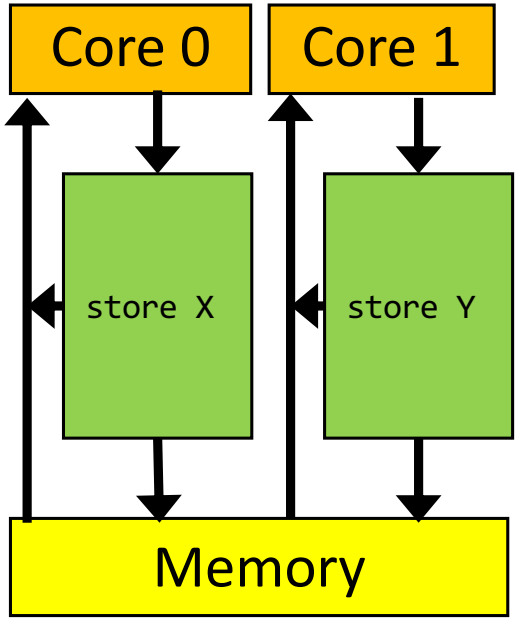
```
store 1,  X
load  r1, Y
```

Core 1

```
store 1,  Y
load  r2, X
```

```
load r1, Y
load r2, X
store 1, X
store 1, Y
```

- Stores buffered, don't appear on other core in time
- X=0, Y=0 is possible!



Memory Fences (Barriers)

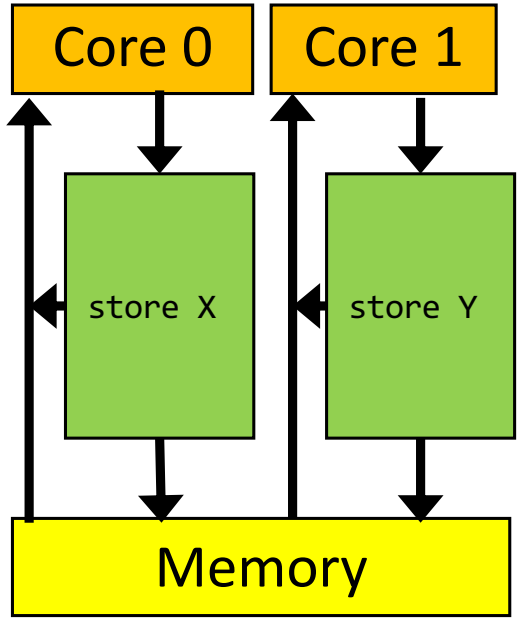
Core 0

```
store 1,  X
fence
load  r1, Y
```

Core 1

```
store 1,  Y
fence
load  r2, X
```

- Prevent re-ordering across fence
- Drains write buffer



Partial Store Ordering (e.g. Arm)

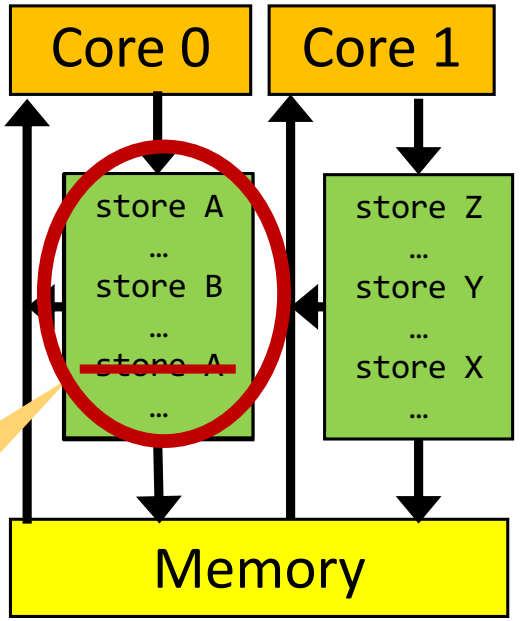
Core 0 does

```
store BUSY, A
store VAL,  B
store IDLE, A
```

Core 1 sees

```
B = VAL
A = IDLE
```

Stores go missing and appear out of order – need memory barrier



Redundant stores cancelled or merged!

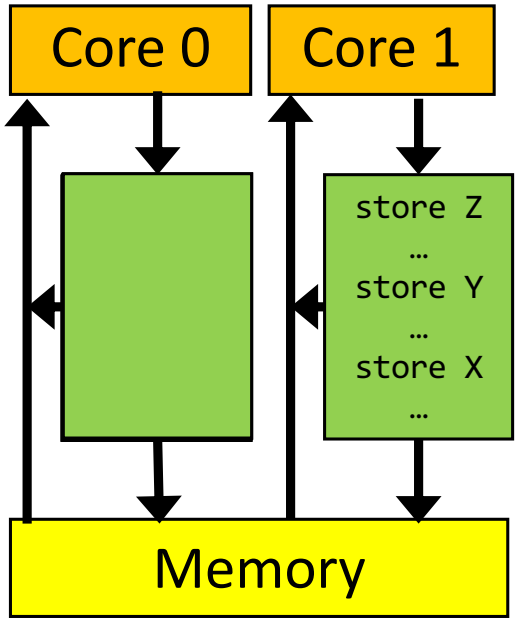
Partial Store Ordering (e.g. Arm)

Core 0 does

```
store BUSY, A
fence
store VAL,  B
fence
store IDLE, A
```

Core 1 sees

```
A = BUSY
B = VAL
A = IDLE
```



Reality more complex:

- Read barriers
- Write barriers
- …

Memory-Ordering Zoo

Type	Alpha	ARM v7	Arm v8	PA RISC	POWER	RISC-V RVWMO	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA-64	IBM zSeries
Loads reord. aft. loads	Y	Y	Y	Y	Y	Y	Y				Y		Y	
Loads reord. aft. stores	Y	Y	Y	Y	Y	Y	Y				Y		Y	
Stores reord. aft. stores	Y	Y	Y	Y	Y	Y	Y	Y			Y		Y	
Stores reord. aft. loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reord. w. loads	Y	Y	opt.		Y	opt.	Y						Y	
Atomic reord. w. stores	Y	Y	opt.		Y	opt.	Y	Y					Y	
Depend. loads reord.	Y													
Incoherent instr. cache pipeline	Y	Y	opt.		Y	Y	Y	Y	Y	Y	Y		Y	Y

MP Hardware Take Away (1/2)

- Each core sees sequential execution of own code
- Other cores see execution affected by
  - Store order and write buffers
  - Cache coherence model
  - Out-of-order execution
- Systems software needs to understand:
  - Specific system (cache, coherence, etc..)
  - Synch mechanisms (barriers, test&set, etc).

… to build cooperative, correct, and scalable parallel code

MP Hardware Take Away (2/2)

- Library sync primitives will be implemented as required by HW
  - Using them you correctly can ignore memory model!
- However, racy code (eg lock-free algorithms) is dangerous!
  - Must understand HW memory model
  - Must add fences as required by HW
  - Easy to get wrong with partial store order!

Locking Primitives

Locking Issues

- Exclusivity of lock-state update – how?
- Scalability – minimise coherency traffic

Mutual Exclusion Techniques

- Disabling interrupts (CLI — STI)
  - OK to prevent pre-emption intra-core
  - Useless for avoiding inter-core concurrency
- Spin locks
  - Busy-waiting wastes cycles
  - May be ok for short critical sections
- Synchronisation objects (locks, semaphores)
  - Flag (or a particular state) indicates object is locked.
  - Manipulating lock requires mutual exclusion

Not needed inside OS unless allow nested exceptions!

Hardware-Provided Locking Primitives

```
int test_and_set(lock *);
int compare_and_swap(int c, int v, lock *);
int exchange(int v, lock *)
int atomic_inc(lock *)
```

Directly map onto instructions

- Bypass/flush cache, write buffer
- Lock bus (optimised on some recent processors)

⇒ generally scale poorly

Hardware-Provided Locking Primitives

- Save lock address in per-core register
- Load from address and return

```
v = load_linked (lock *)
bool store_conditional (int, lock *)
```

Need re-try loop ⇒ spinning!

Any store & coherency traffic:
- Invalidate register if address matches value

- If address matches value perform store & return TRUE
- Else return FALSE

Spin Locks

```
void lock (volatile lock_t *l) {
  while (test_and_set(l));
}
void unlock (volatile lock_t *l) {
  *l = 0;
}
```

Similar with LL/SC

Somewhat better:

- Test & test & set lock

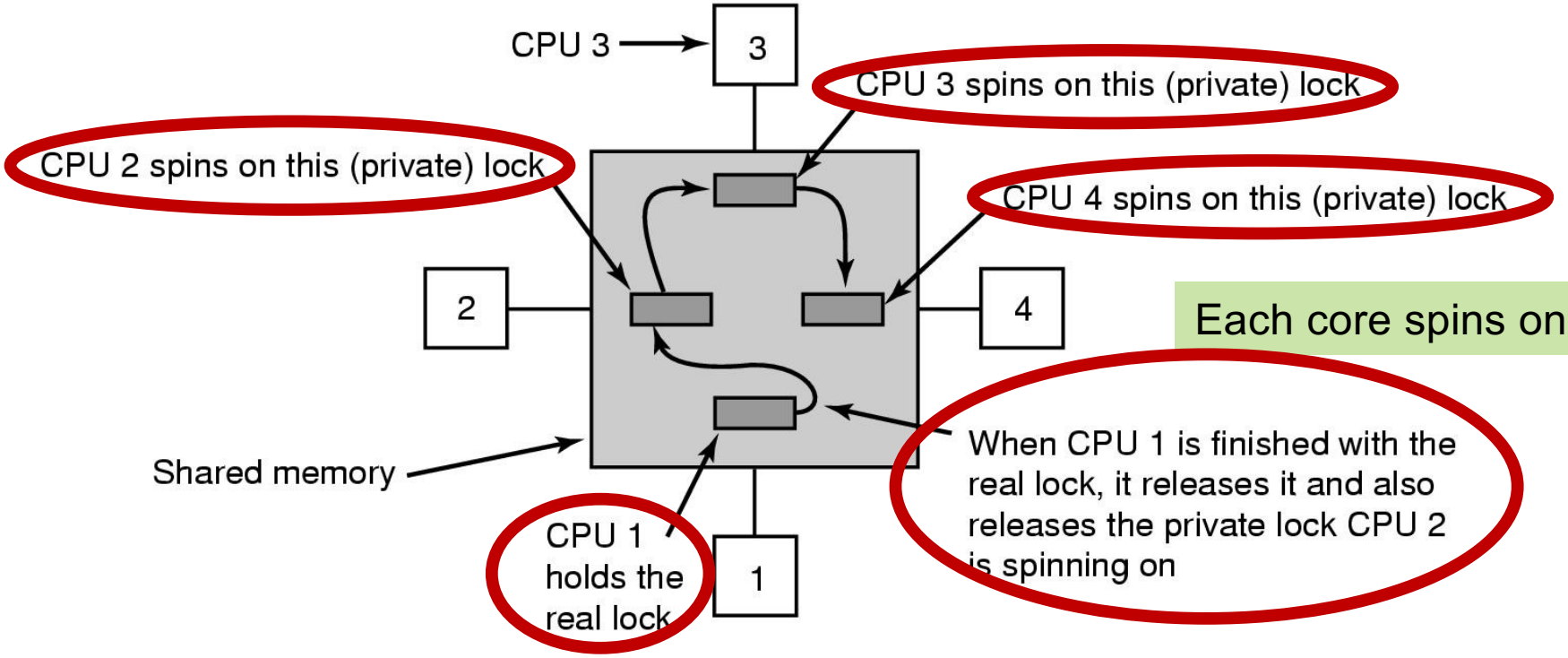
Still scalability issues due to bus traffic!

Scalable Spin Locks

- Issue: Spinning on shared location ⇒ non-scalable bus traffic

Idea: Only spin on uncontented data
- Have queue of lines to spin locally
- Lock-holder advances queue
- Guarantees fairness (FIFO)

Queueing – MCS Lock



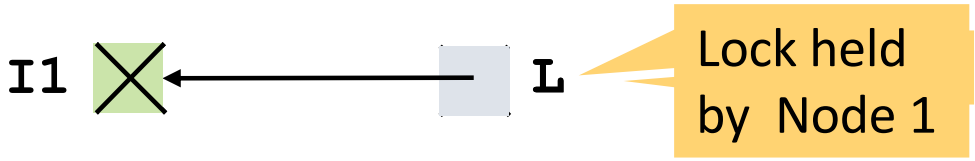
John Mellor-Crummey and Michael Scott, "Algorithms for Scalable Synchronisation on Shared-Memory Multiprocessors", *ACM Transactions on Computer Systems*, Vol. 9, No. 1, 1991

MCS Acquire

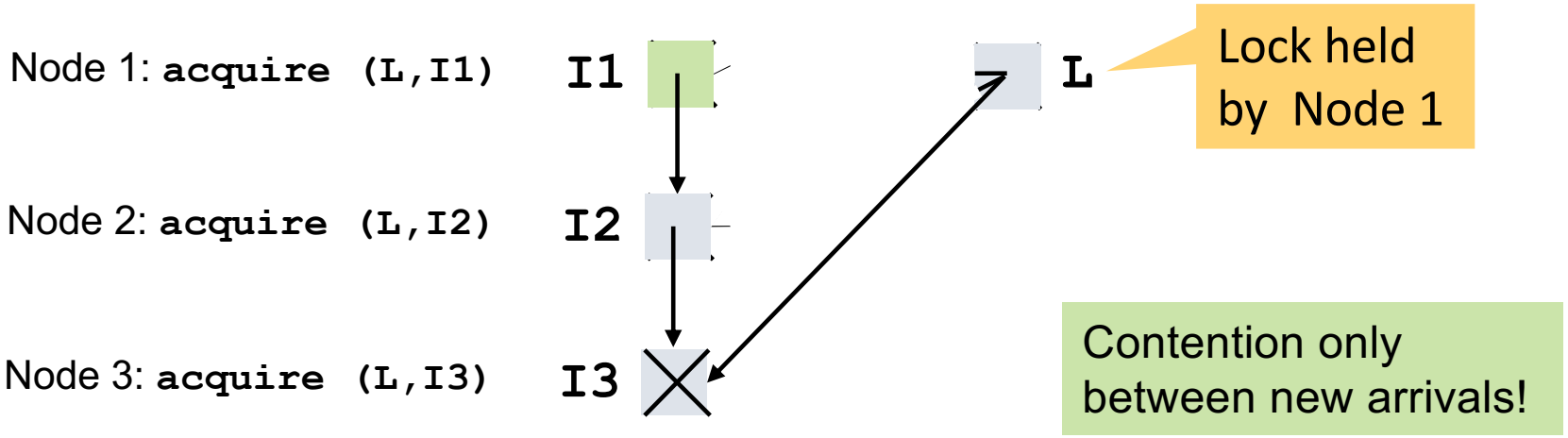
Lock free

MCS Acquire

Node 1: `acquire (L,I1)`

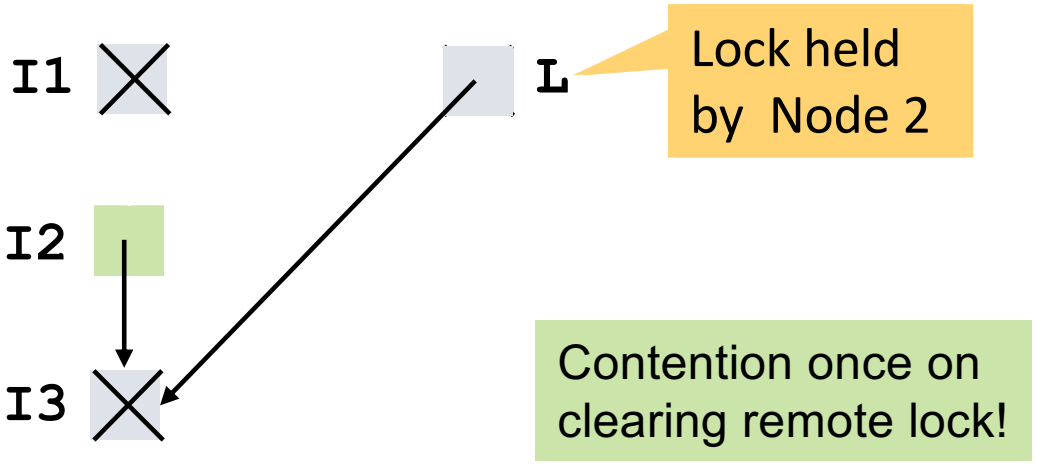


MCS Acquire



MCS Release

Node 1: `release (L,I1)`



Sample MCS Code for Arm

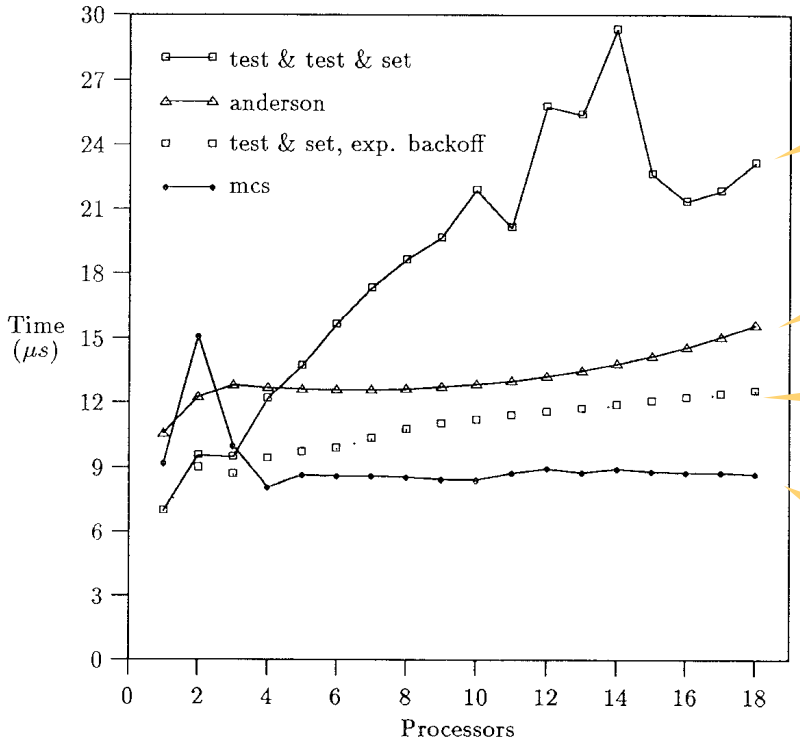
```
void mcs_acquire(mcs_lock *L, mcs_qnode_ptr I)
{
    I->next = NULL;
    MEM_BARRIER;
    mcs_qnode_ptr pred = (mcs_qnode*) SWAP_PTR( L, (void *)I);
    if (pred == NULL) {
      /* lock was free */
      MEM_BARRIER;
      return;
    }
    I->waiting = 1; // word on which to spin
    MEM_BARRIER;
    pred->next = I; // make pred point to me
}
```

Atomic exchange:

- Returns prev value of L
- Sets L to I

MCS Benchmarks

Take-away
- MCS lock scales well under contention
- … but has higher overhead



Test & test & set

[Anderson ‘90]

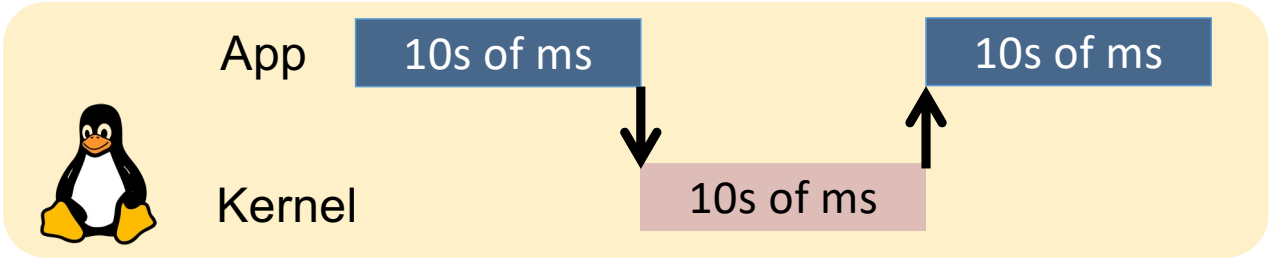
Test & test w exp. backoff

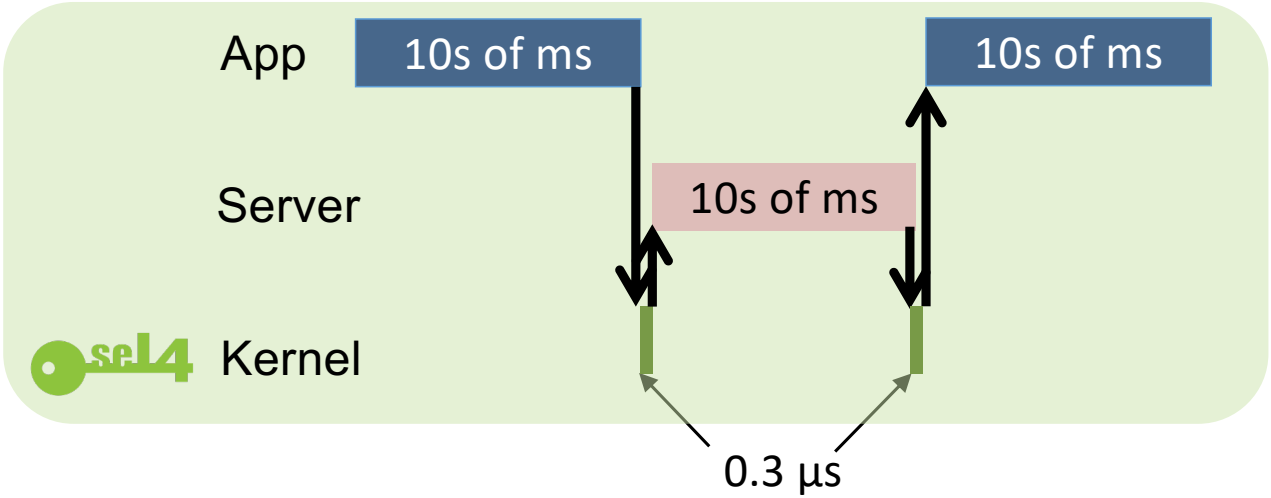
MCS lock

John Mellor-Crummey and Michael Scott, “Algorithms for Scalable Synchronisation on Shared-Memory Multiprocessors”, *ACM Transactions on Computer Systems*, Vol. 9, No. 1, 1991

Locking and Microkernels

Monolithic vs Microkernel Execution





Monolithic kernels have long system calls ⇒ fine-grained locking is essential

"For a microkernel, a big lock is fine!" [Peters et al, APSys'15]

Even with many cores?

Highly Abstracted View of seL4 Syscall

1. save state
2. acquire kernel lock
3. look-up caps, destination state
4. if (can-proceed) {
5. maybe update cap state
6. maybe update thread state
7. copy data [PPC only]
8. } else fail
9. release kernel lock
10. restore state
11. return

Fast syscall accesses 10-20ish cache lines, takes 500–1000 cycles

Fast syscall **writes** 1–2 **shared** cache lines

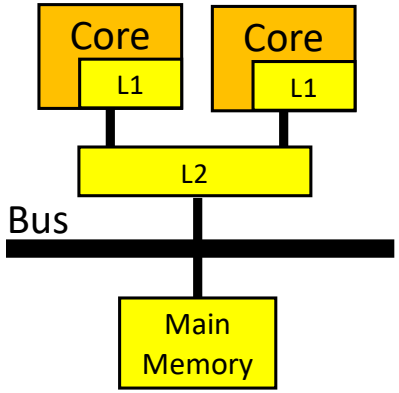
What is the cost of writing shared data?

Approximate Cost of Writing Shared Lines

Arm (shared L2 cache)

- ≤8 closely coupled cores
- Cache-line migration latency: 10–20 cycles
- 2 lines ≈ 2.5% of base syscall

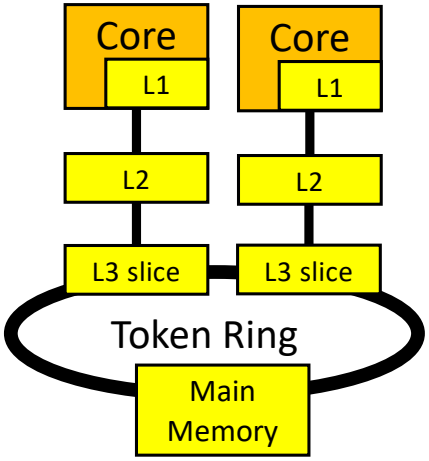
No problem!



Intel (private L2, sliced L3 cache)

- 100s loosely coupled cores
- Cache-line migration latency: 100s–1000s cycles
- 2 lines ≳ cost of base syscall

Insane!



Microkernel Shared-State Take-Aways

1. Global state is no problem for small number of closely coupled cores (eg Arm)
2. Global state **dominates** syscall cost for loosely coupled cores – inevitable for large core counts

Microkernel minimality principle:
A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality. [Liedtke SOSP'95]

Can we take shared state out of the kernel?

(Clustered) Multikernel

Present multicore as a NUMA system!