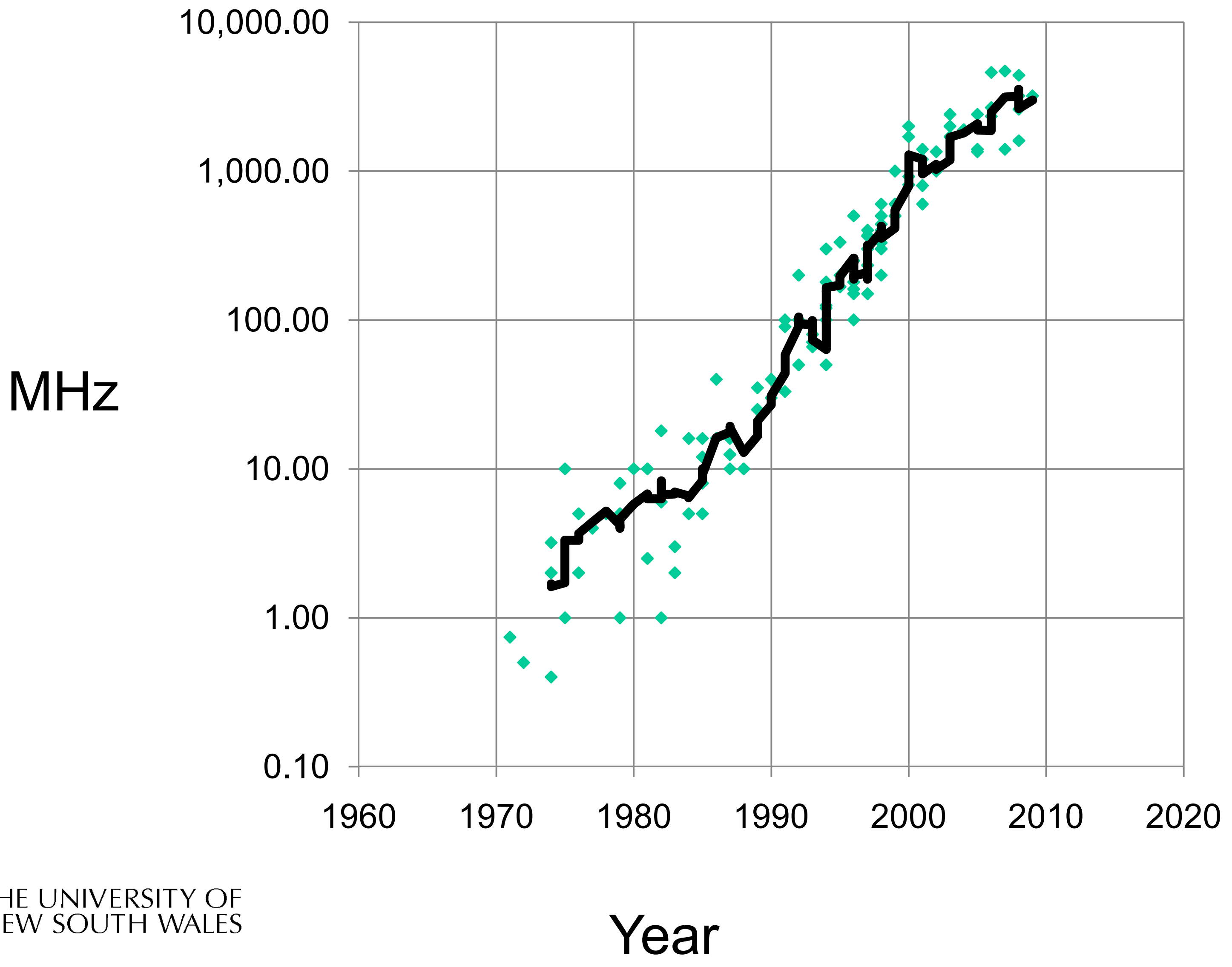
Multiprocessor Systems

Chapter 8, 8.1

CPU clock-rate increase slowing



THE UNIVERSITY OF
NEW SOUTH WALES

Multiprocessor System

- We will look at *shared-memory multiprocessors*
  - More than one processor sharing the same memory
- 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
- Disks and other hardware can be expensive
  - Can share hardware between CPUs

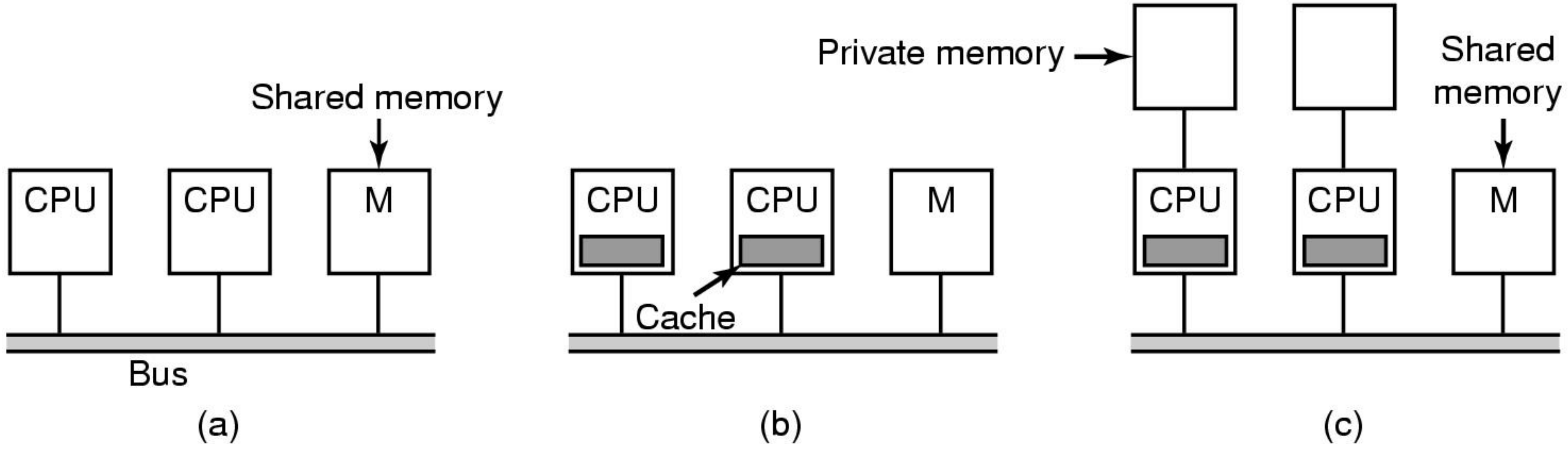
Types of Multiprocessors (MPs)

- UMA MP
  - Uniform Memory Access
    - Access to all memory occurs at the same speed for all processors.
- NUMA MP
  - Non-uniform memory access
    - Access to some parts of memory is faster for some processors than other parts of memory
- We will focus on UMA

Bus Based UMA

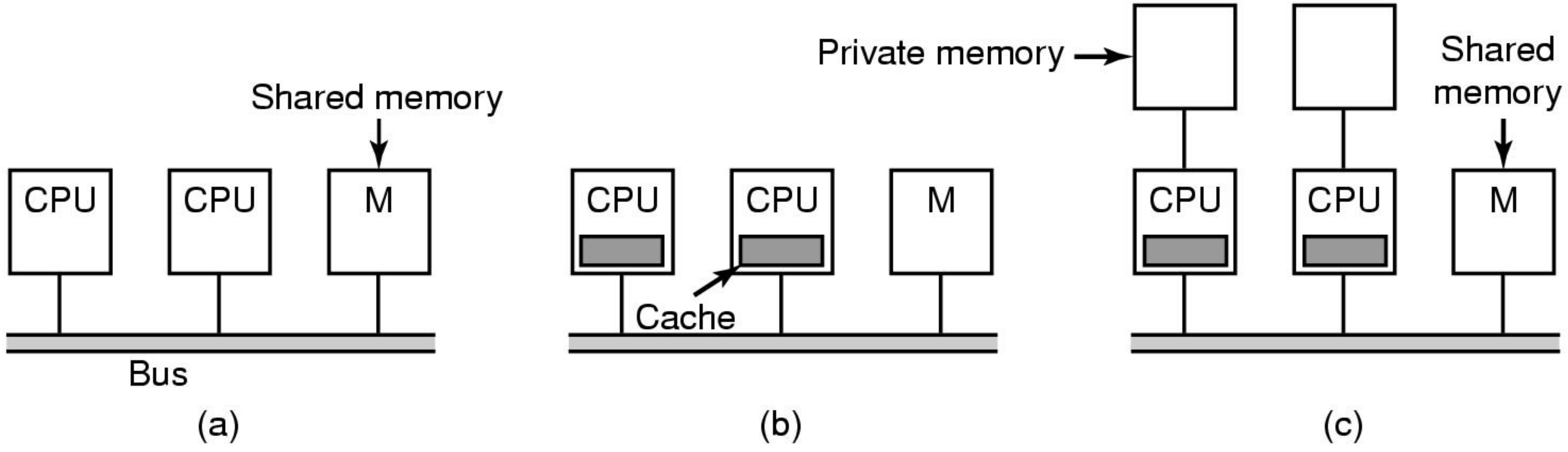
Simplest MP is more than one processor on a single bus connect to memory (a)

- Bus bandwidth becomes a bottleneck with more than just a few CPUs



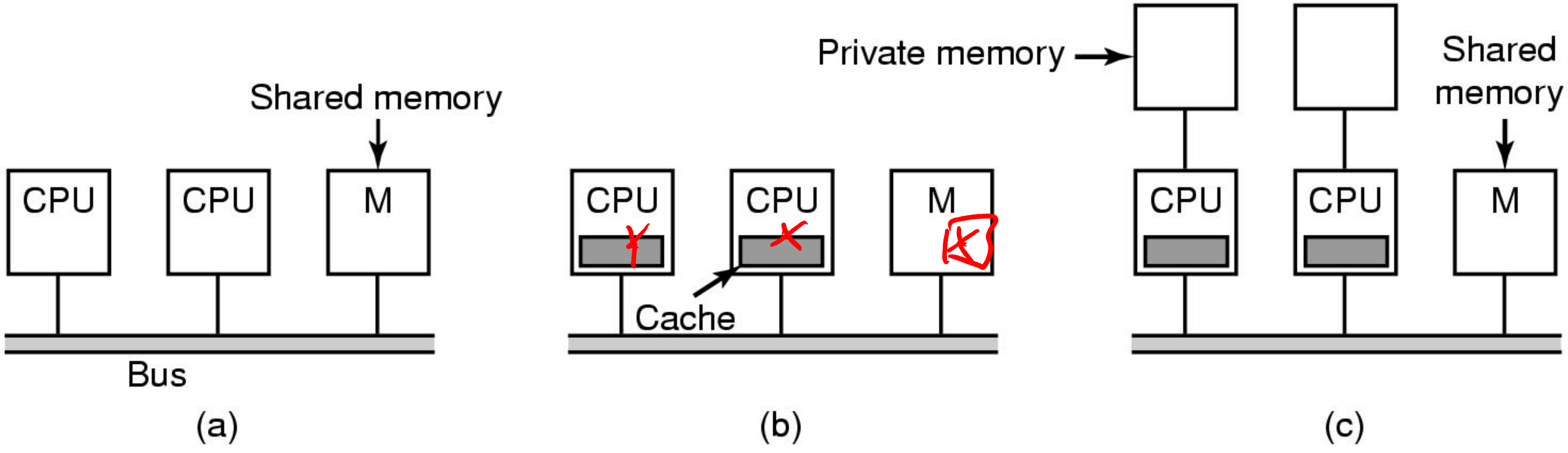
Bus Based UMA

- Each processor has a cache to reduce its need for access to memory (b)
  - Hope is most accesses are to the local cache
  - Bus bandwidth still becomes a bottleneck with many CPUs



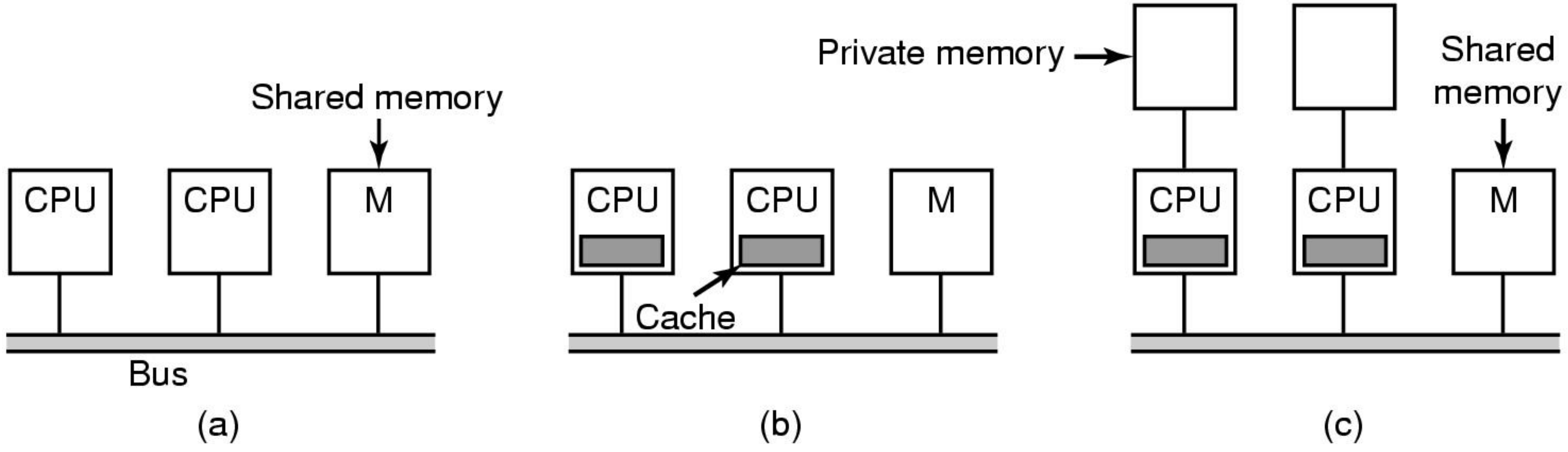
Cache Consistency

- 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)?



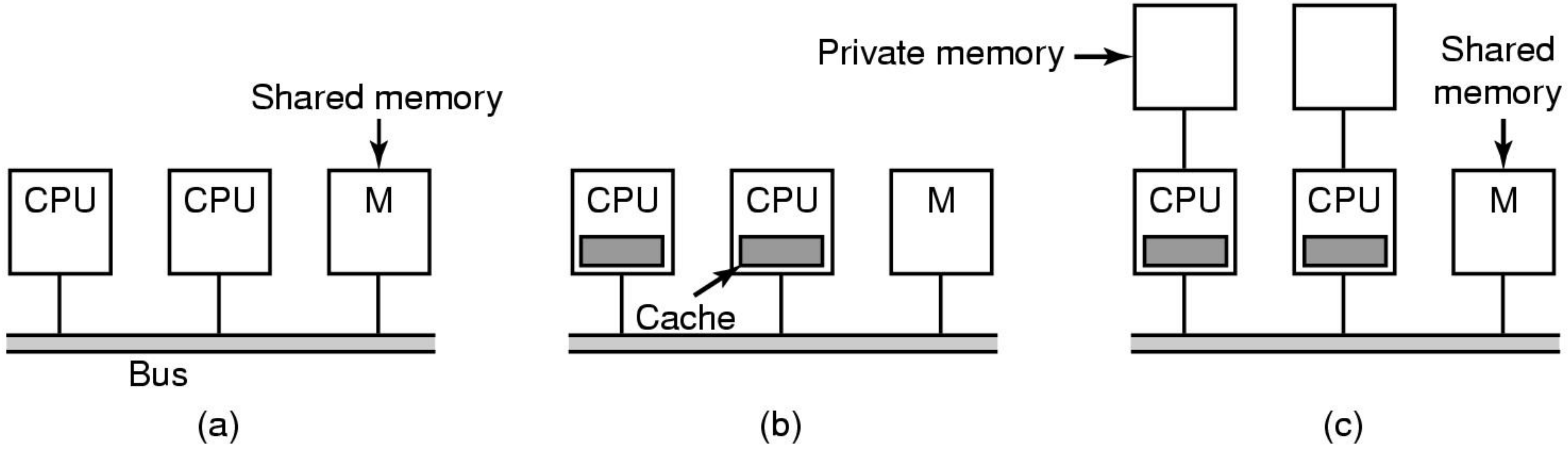
Cache Consistency

- 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



Bus Based UMA

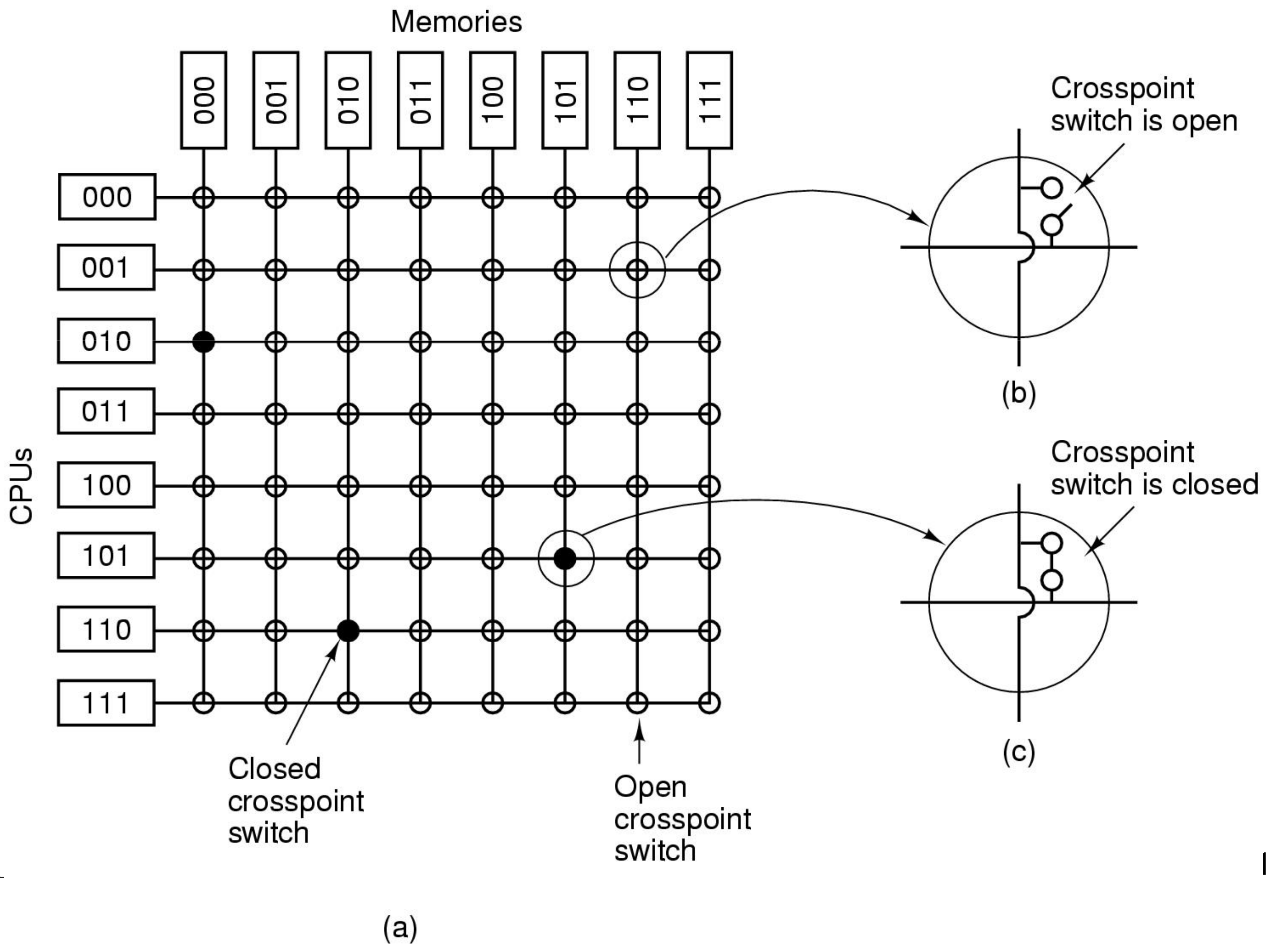
- To further scale the number processors, we give each processor private local memory
  - Keep private data local on off the shared memory bus
  - Bus bandwidth still becomes a bottleneck with many CPUs with shared data
  - Complicate application development
    - We have to partition between private and shared variables



Bus Based UMA

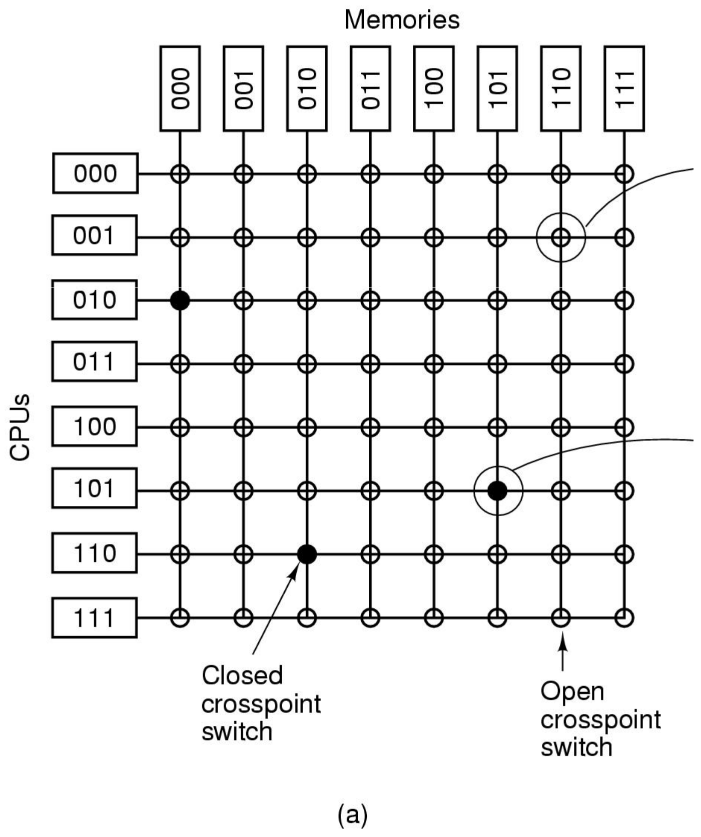
- With only a single shared bus, scalability is limited by the bus bandwidth of the single bus
  - Caching only helps so much
- Alternative bus architectures do exist on high-end machines

UMA Crossbar Switch



UMA Crossbar Switch

- Pro
  - Any CPU can access any available memory with less blocking
- Con
  - Number of switches required scales with n^2.
    - 1000 CPUs need 1000000 switches



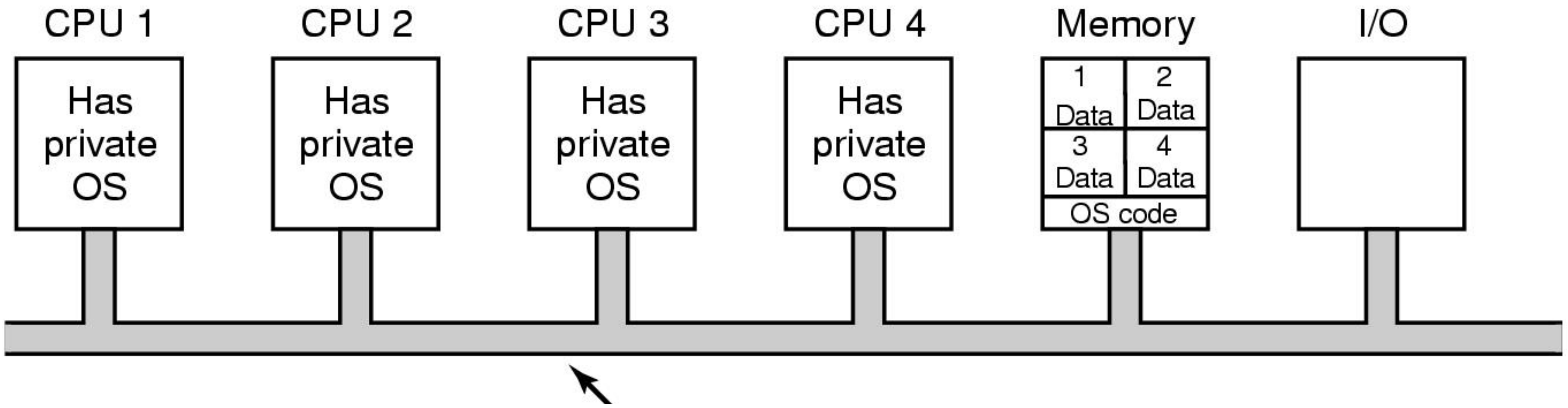
THE UNIVERSITY OF NEW SOUTH WALES

Summary

- Multiprocessors can
  - Increase computation power beyond that available from a single CPU
  - Share resources such as disk and memory
- However
  - Shared buses (bus bandwidth) limit scalability
    - Can be reduced via hardware design
    - Can be reduced by carefully crafted software behaviour
      - Good cache locality together with private data where possible
- Question
  - How do we construct an OS for a multiprocessor?
    - What are some of the issues?

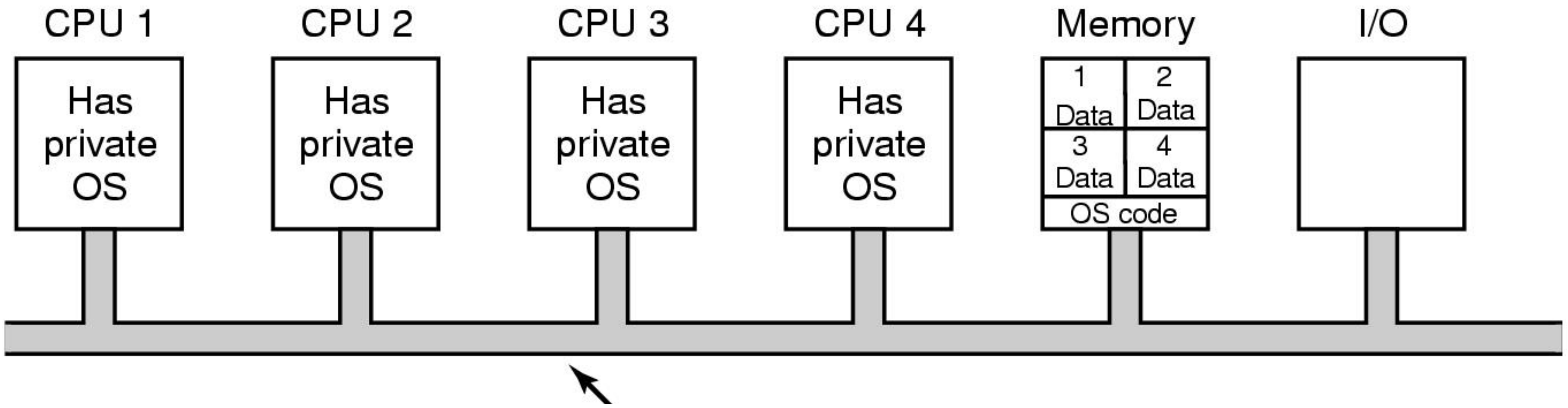
Each CPU has its own OS

- Statically allocate physical memory to each CPU
- Each CPU runs its own independent OS
- Share peripherals
- Each CPU (OS) handles its processes system calls



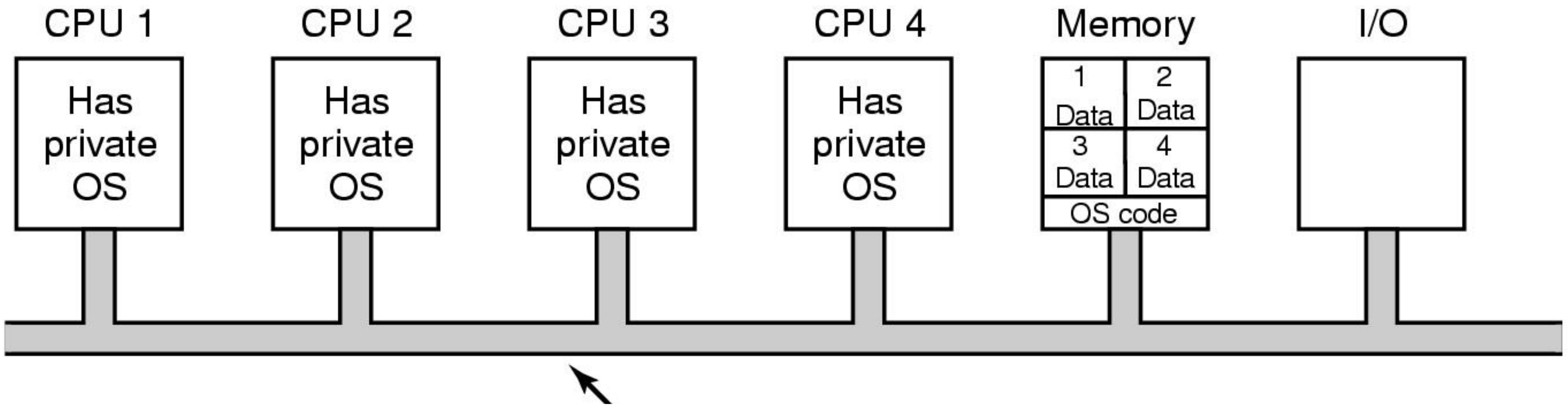
Each CPU has its own OS

- Used in early multiprocessor systems to 'get them going'
  - Simpler to implement
  - Avoids concurrency issues by not sharing



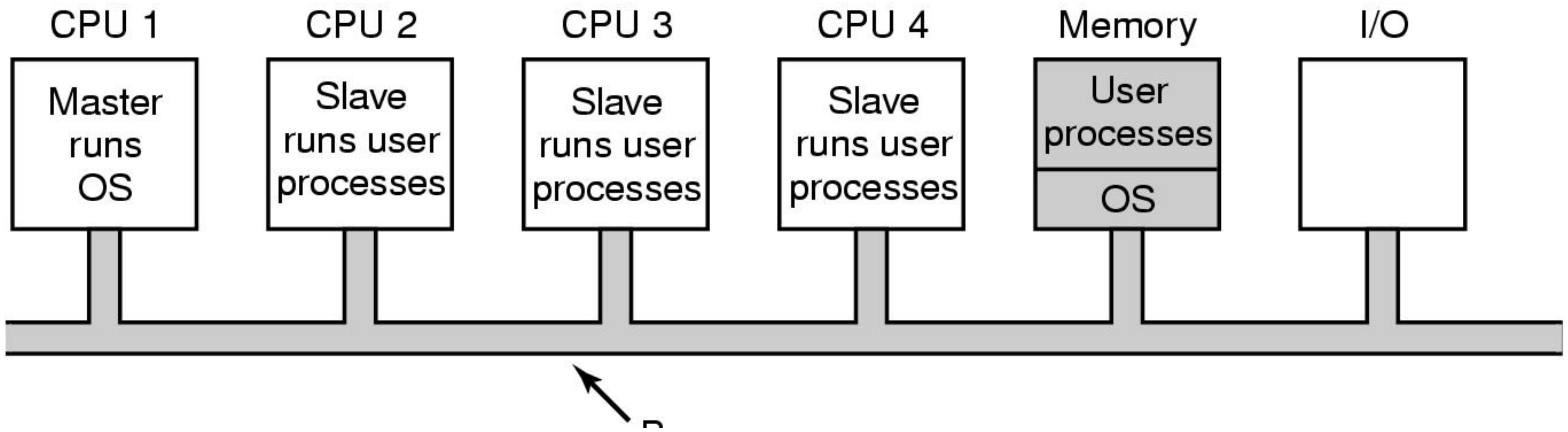
Issues

- 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
- Consistency is an issue with independent disk buffer caches and potentially shared files



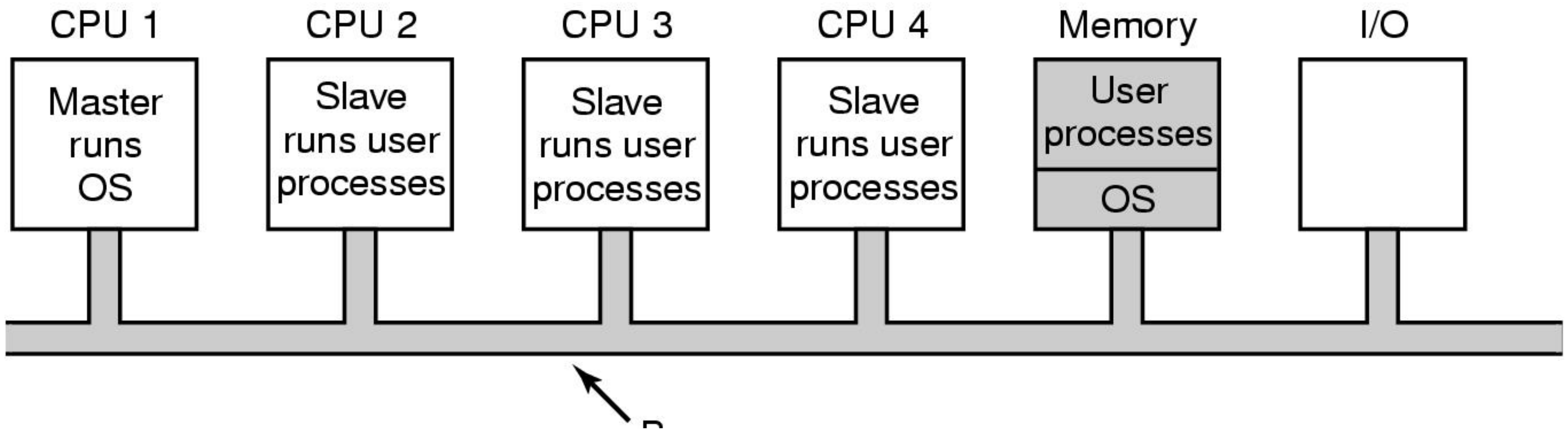
Master-Slave Multiprocessors

- OS (mostly) runs on a single fixed CPU
  - All OS tables, queues, buffers are present/manipulated on CPU 1
- User-level apps run on the other CPUs
  - And CPU 1 if there is spare CPU time
- All system calls are passed to CPU 1 for processing


CPU 1
CPU 2
CPU 3
CPU 4
Memory
I/O
Master runs OS
Slave runs user processes
Slave runs user processes
Slave runs user processes
User processes
OS

Master-Slave Multiprocessors

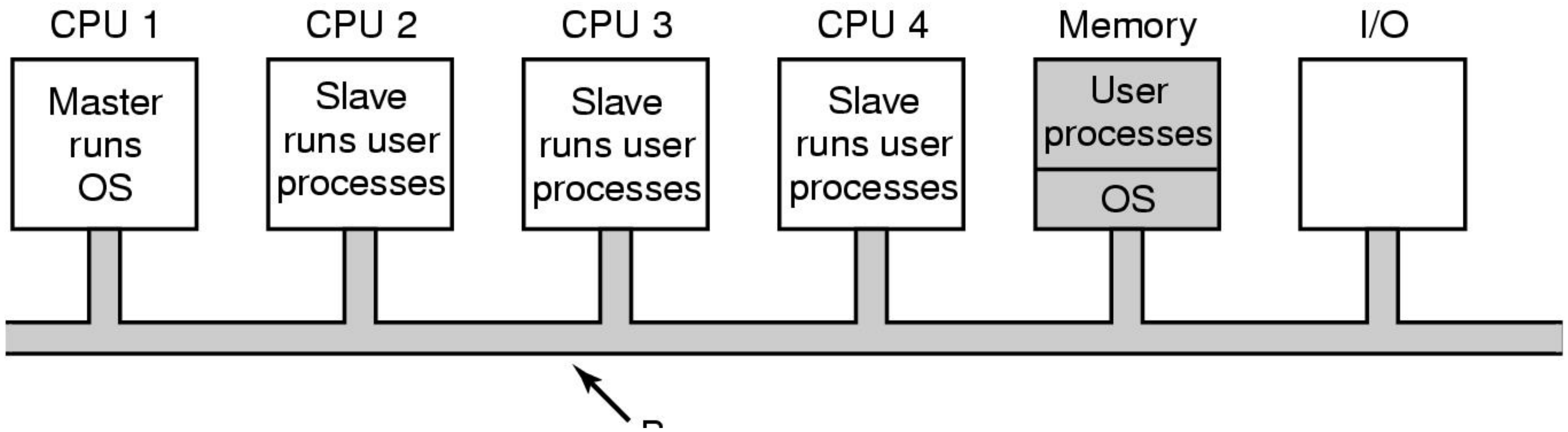
- Very little synchronisation required
  - Only one CPU accesses kernel data
- Simple to implement
- Single, centralised scheduler
  - Keeps all processors busy
- Memory can be allocated as needed to all CPUs



CPU 1
Master runs OS
CPU 2
Slave runs user processes
CPU 3
Slave runs user processes
CPU 4
Slave runs user processes
Memory
User processes
OS
I/O

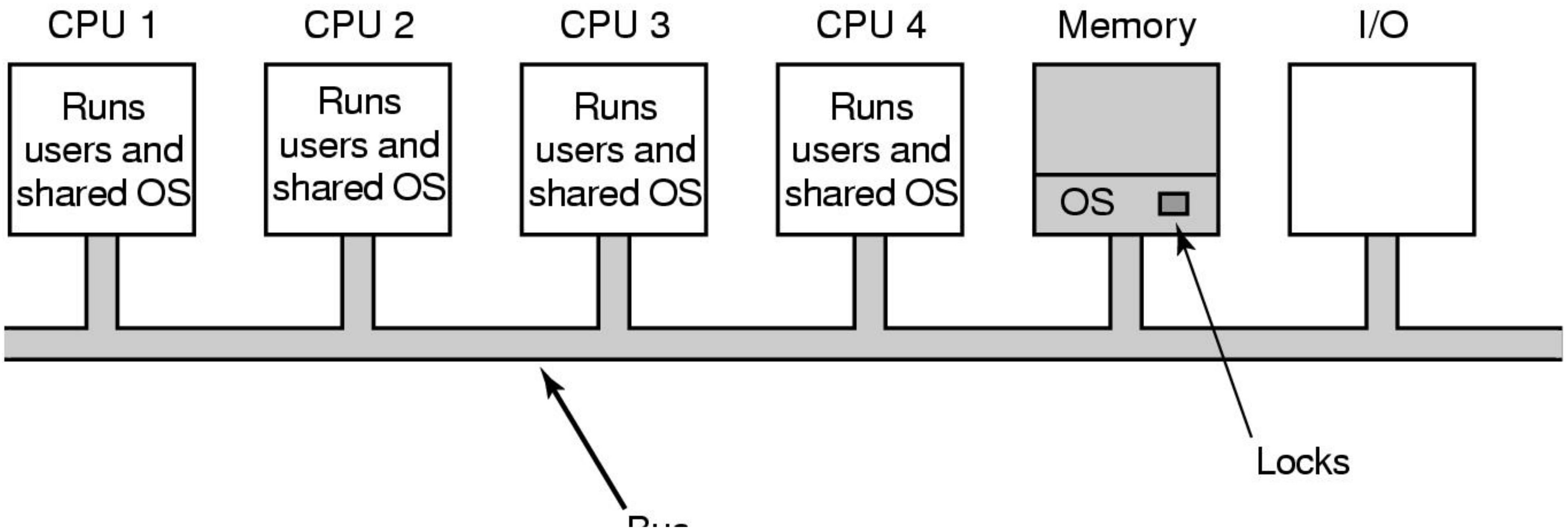
Issue

- Master CPU can become the bottleneck


CPU 1
Master runs OS
CPU 2
Slave runs user processes
CPU 3
Slave runs user processes
CPU 4
Slave runs user processes
Memory
User processes
OS
I/O

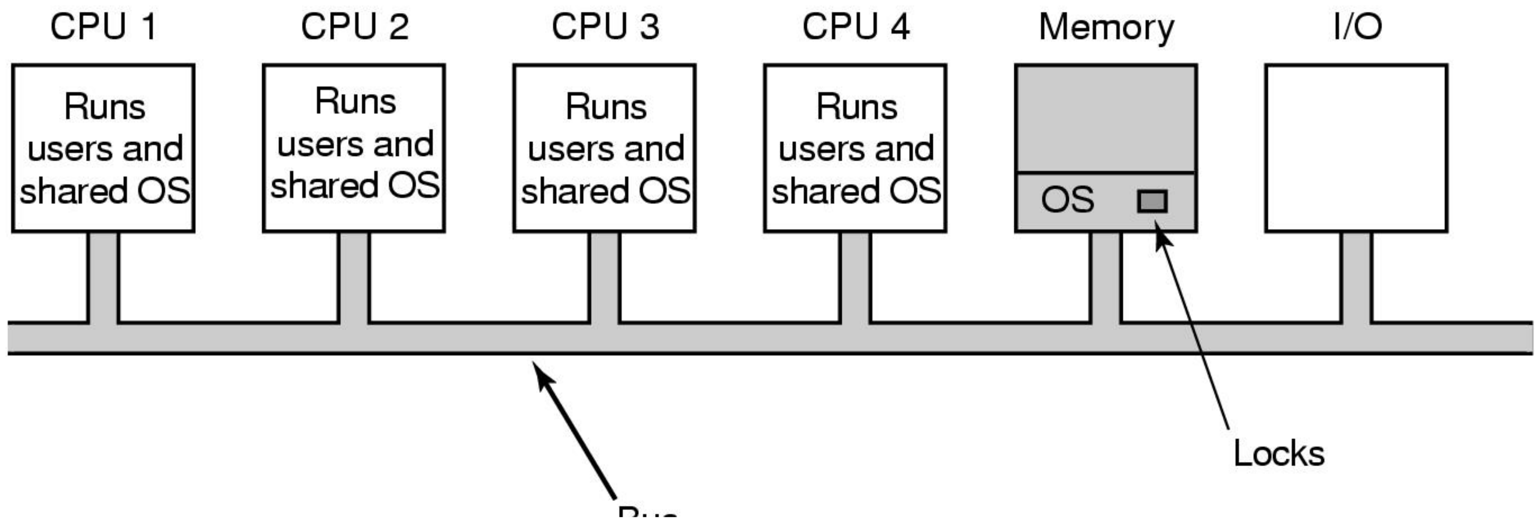
Symmetric Multiprocessors (SMP)

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



Symmetric Multiprocessors (SMP)

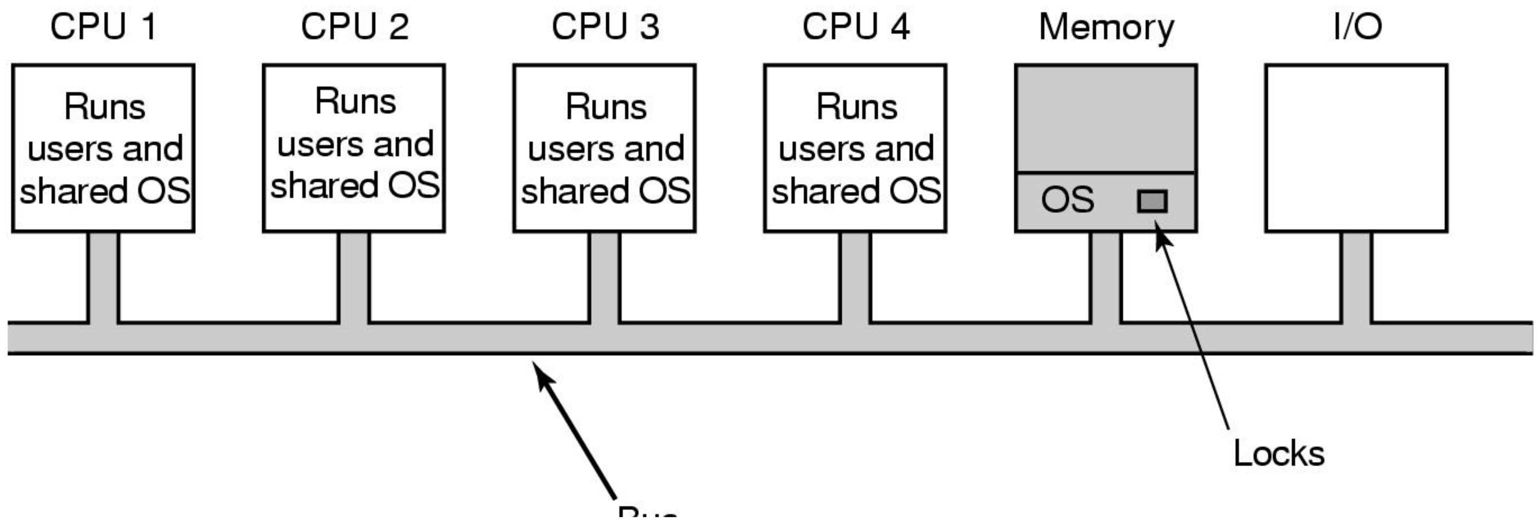
- One alternative: A single mutex that make the entire kernel a large critical section
  - Only one CPU can be in the kernel at a time
  - Only slight better solution than master slave
    - Better cache locality
    - The “big lock” becomes a bottleneck when in-kernel processing exceed what can be done on a single CPU



Symmetric Multiprocessors (SMP)

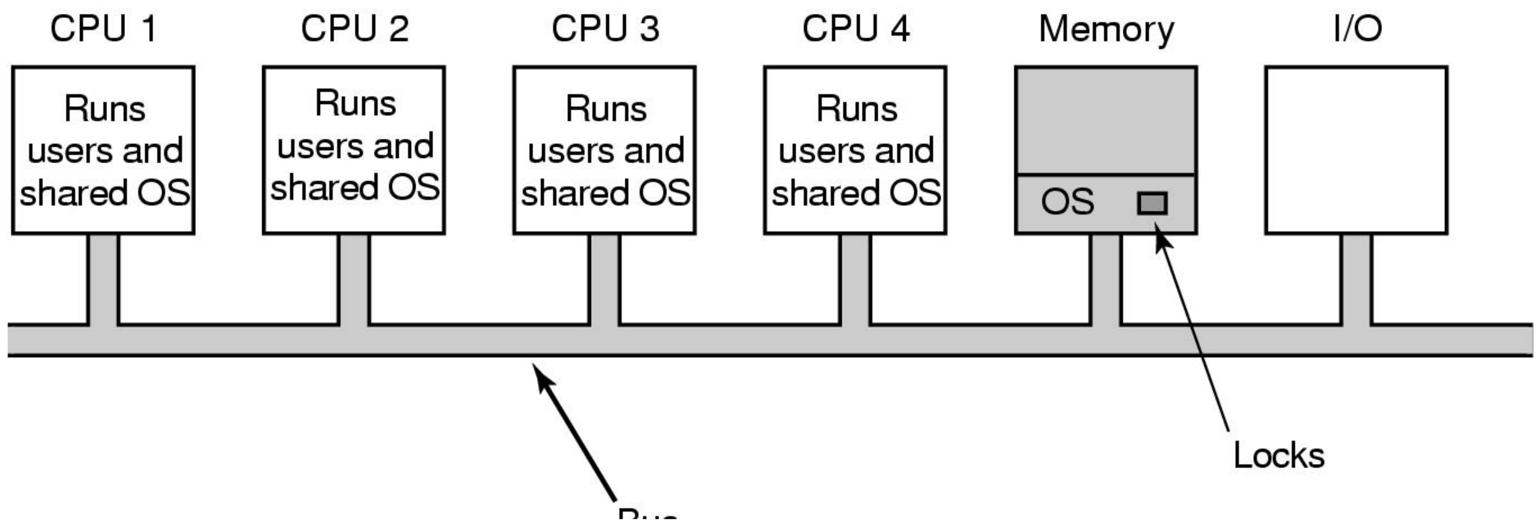
- Better alternative: identify largely independent parts of the kernel and make each of them their own critical section
  - Allows more parallelism in the kernel

- Issue: Difficult task
  - Code is mostly similar to uniprocessor code
  - Hard part is identifying independent parts that don't interfere with each other



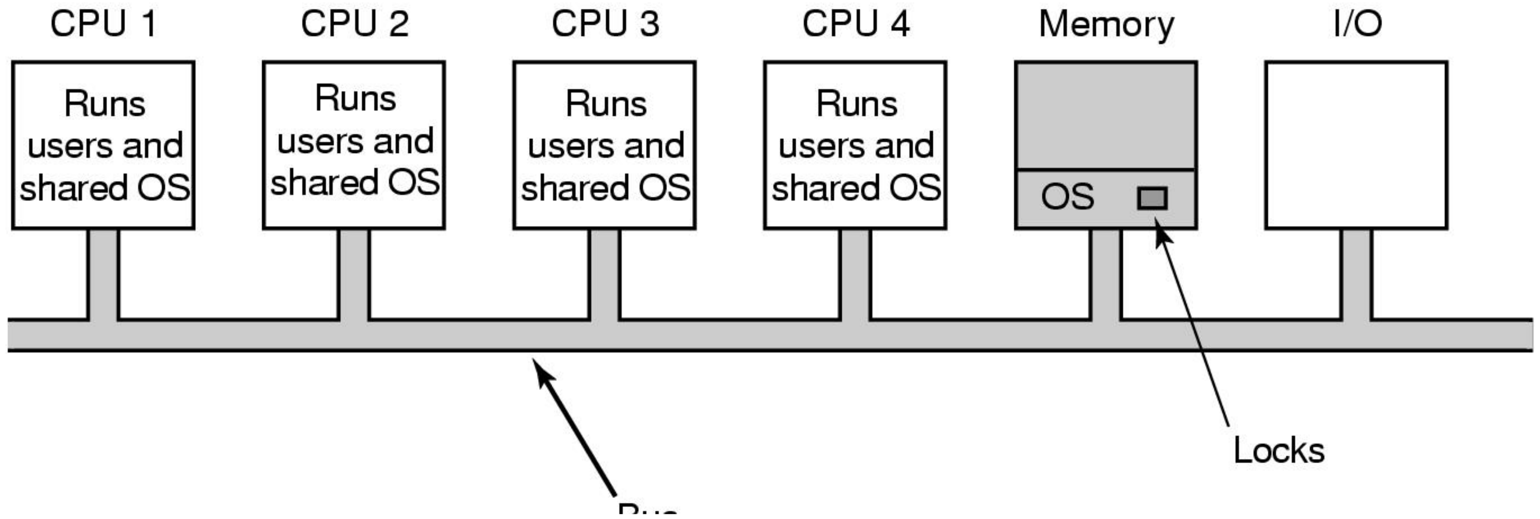
Symmetric Multiprocessors (SMP)

- 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



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
- 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

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 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
```

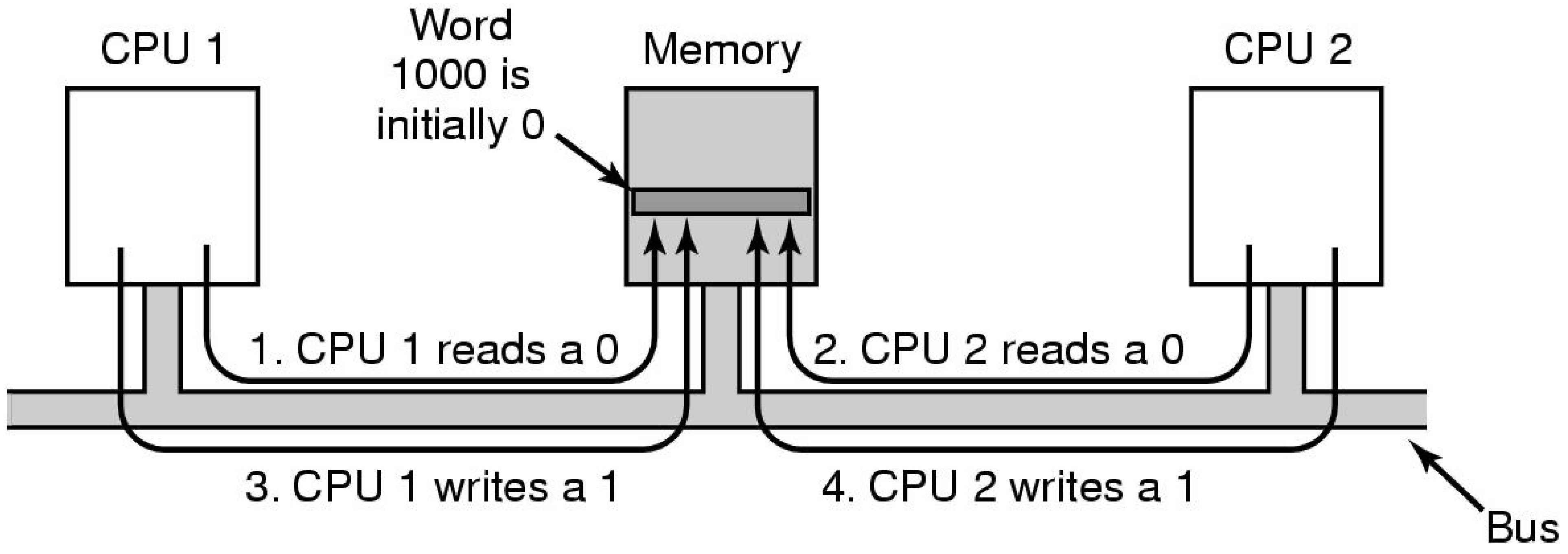
Entering and leaving a critical region using the TSL instruction

Test-and-Set

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

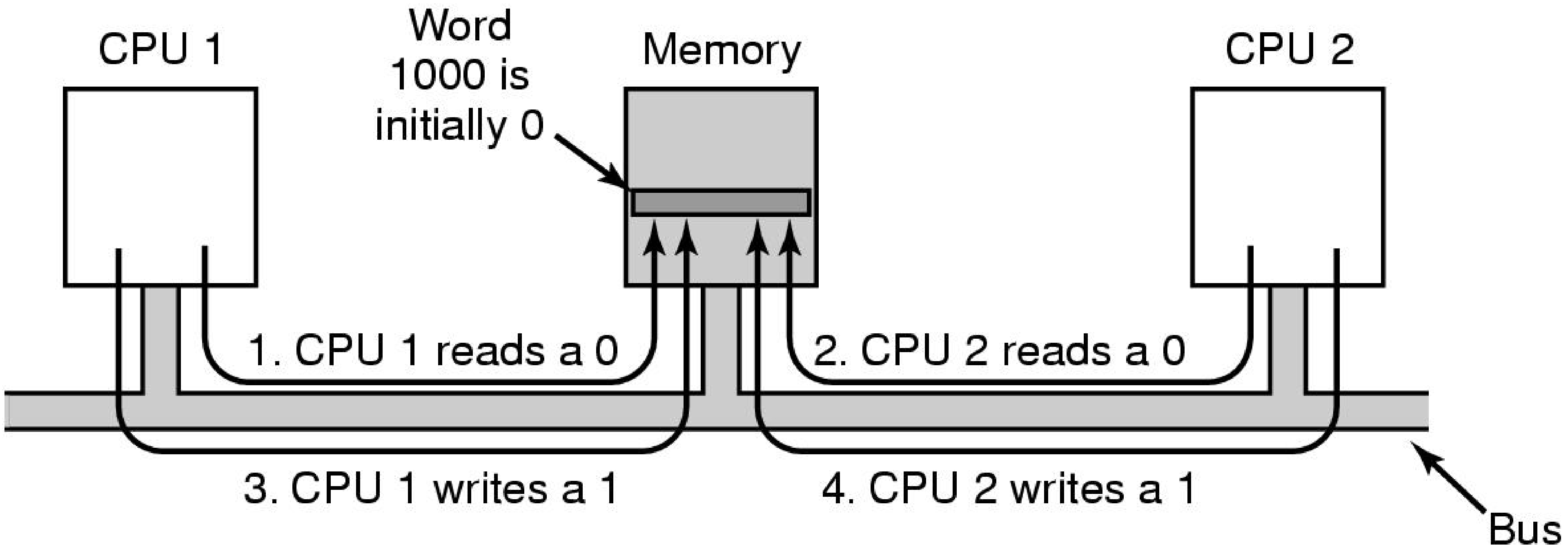
Test-and-Set on SMP

- It does not work without some extra hardware support



Test-and-Set on SMP

- A solution:
  - Hardware *locks* the bus during the TSL instruction to prevent memory accesses by any other CPU

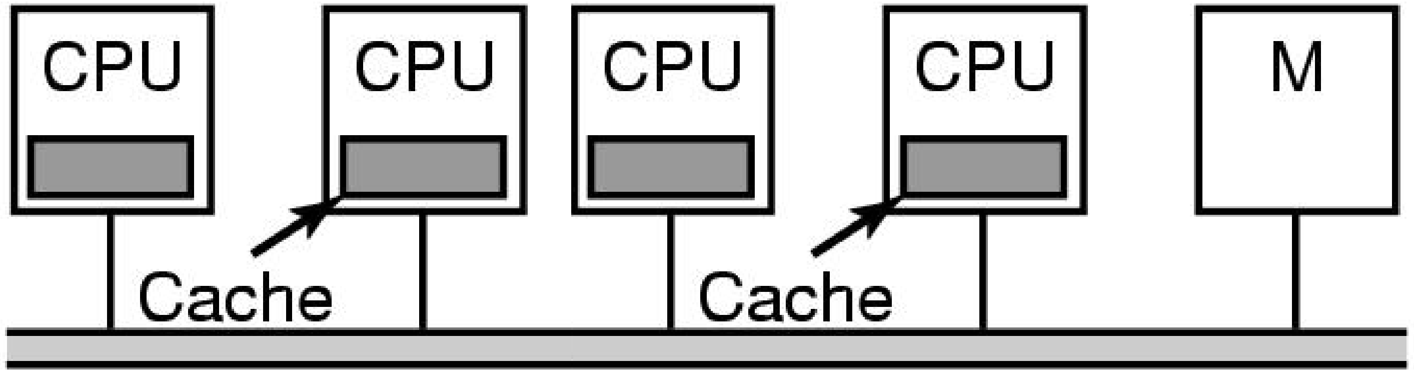


Test-and-Set on SMP

- 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 bus locking which slows all other CPUs down
      - Independent of whether other CPUs need a lock or not
      - Causes bus contention

Test-and-Set on SMP

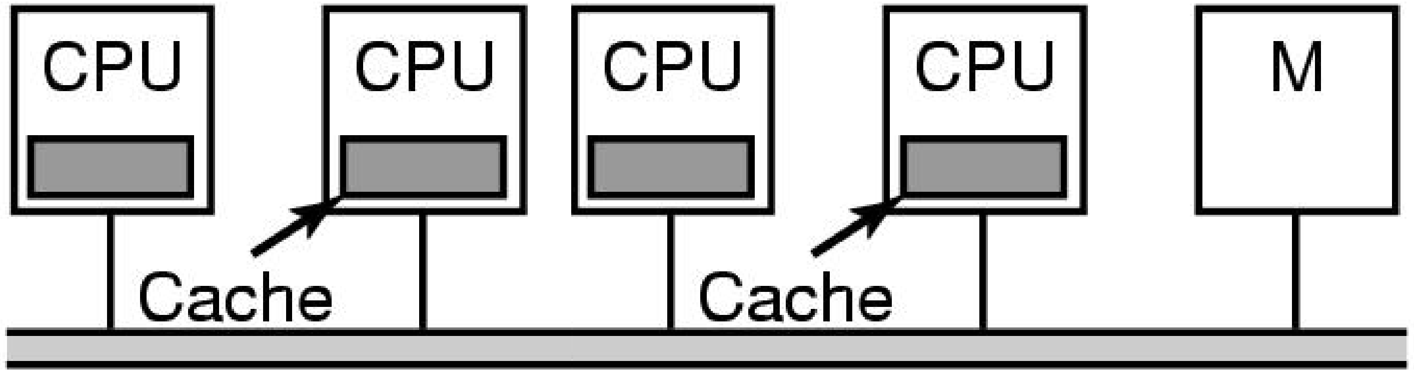
- Caching does not help reduce bus contention
  - Either TSL still locks the bus
  - Or TSL requires exclusive access to an entry in the local cache
    - Requires invalidation of same entry in other caches, and loading entry into local cache
    - Many CPUs performing TSL simply bounce a single exclusive entry between all caches using the bus



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;
```



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

Compares Simple Spinlocks

- Test and Set

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

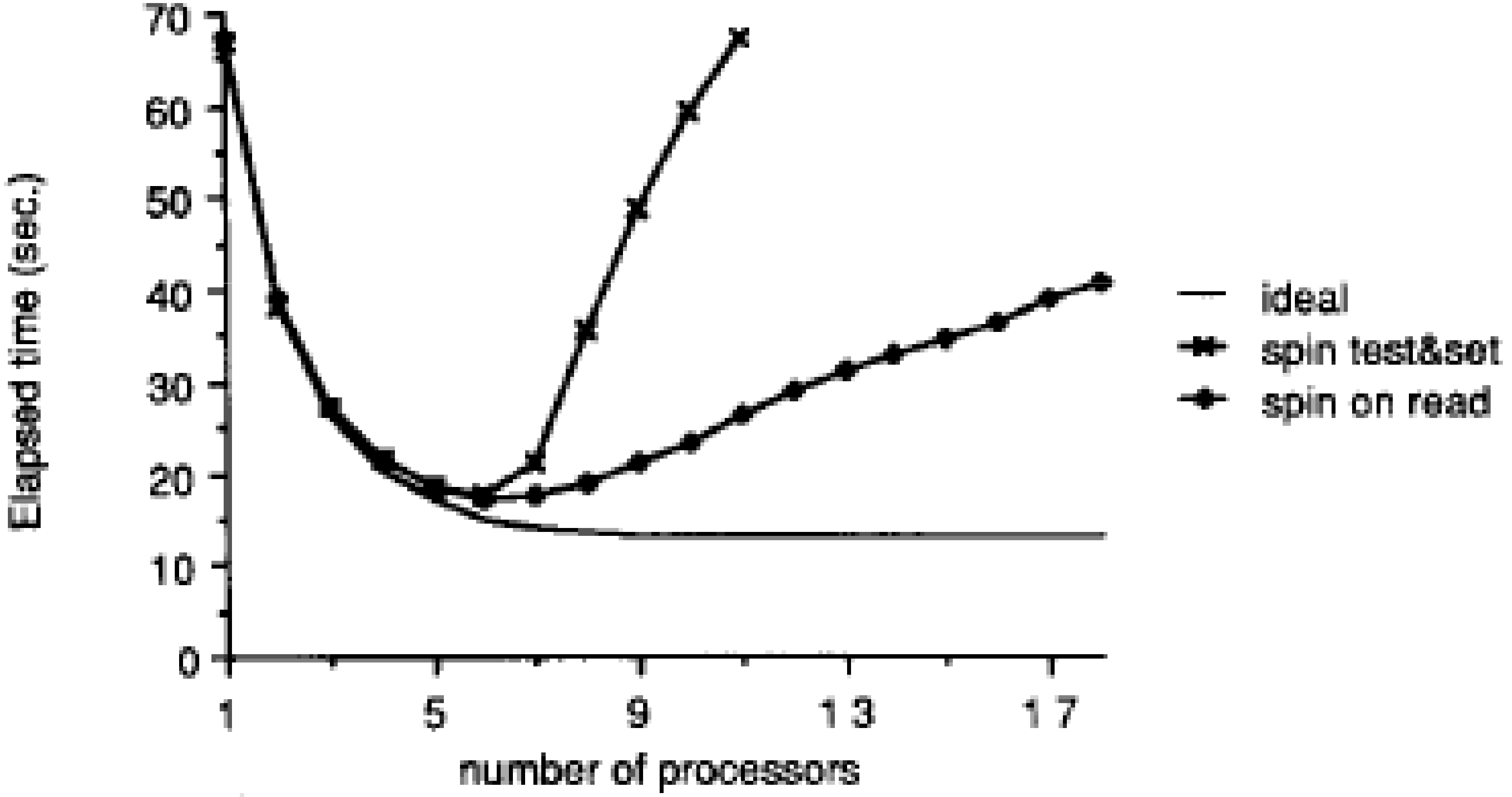
- Read before Test and Set

```
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)


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

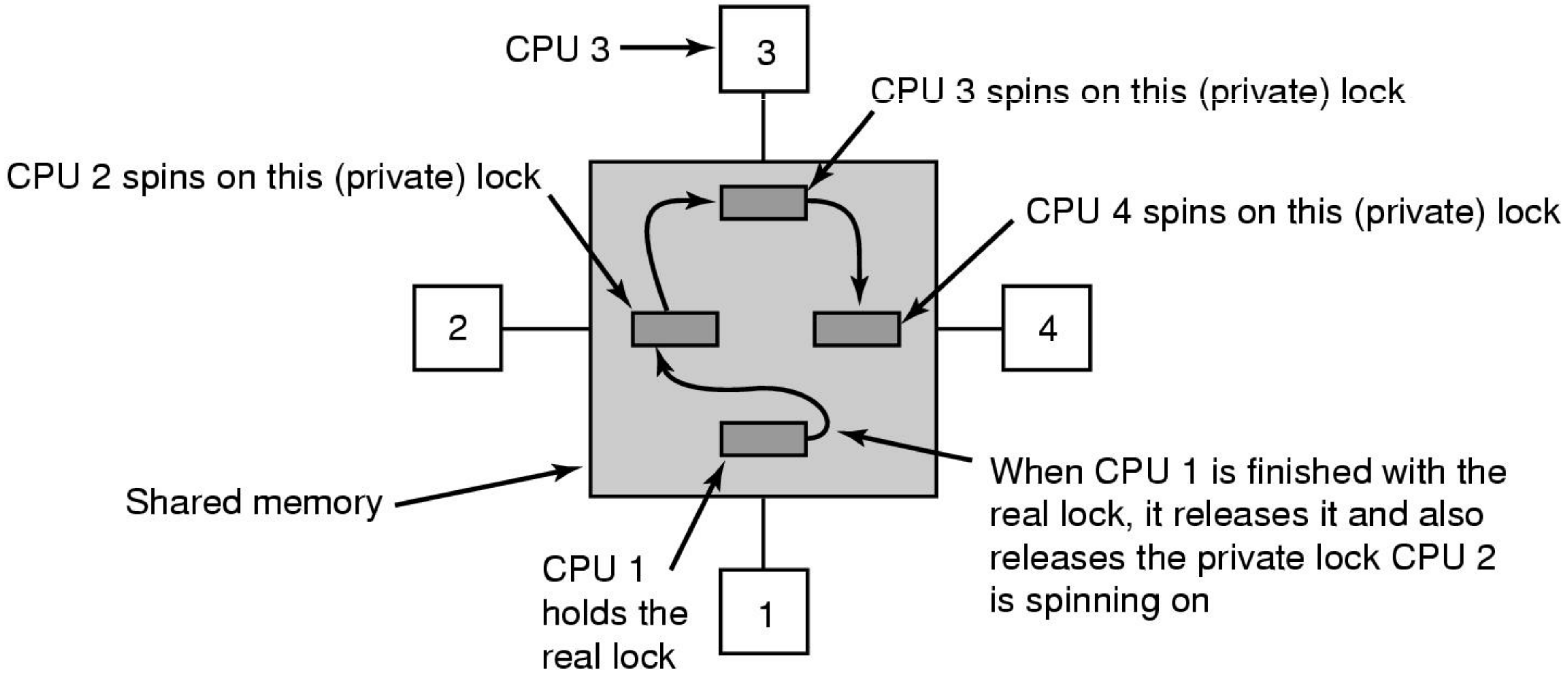
Results

- Test and set performs poorly once there is enough CPUs to cause contention for lock
  - Expected
- Test and Test and Set performs better
  - Performance less than expected
  - 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 missed as they test and set $\Rightarrow$ lock holder is slower
  - Slower lock holder results in more contention

- 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 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()

```
type qnode = record
    next : ^qnode
    locked : Boolean
type lock = ^qnode

// parameter I, below, points to a qnode record allocated
// (in an enclosing scope) in shared memory locally-accessible
// to the invoking processor

procedure acquire_lock (L : ^lock, I : ^qnode)
    I->next := nil
    predecessor : ^qnode := fetch_and_store (L, I)
    if predecessor != nil      // queue was non-empty
        I->locked := true
        predecessor->next := I
        repeat while I->locked             // spin

procedure release_lock (L : ^lock, I: ^qnode)
    if I->next = nil        // no known successor
        if compare_and_swap (L, I, nil)
            return
            // compare_and_swap returns true iff it swapped
        repeat while I->next = nil         // spin
    I->next->locked := false
```

Selected Benchmark

- Compared
  - test and test and set
  - Others in paper
    - Anderson’s array based queue
    - test and set with exponential back-off
  - MCS

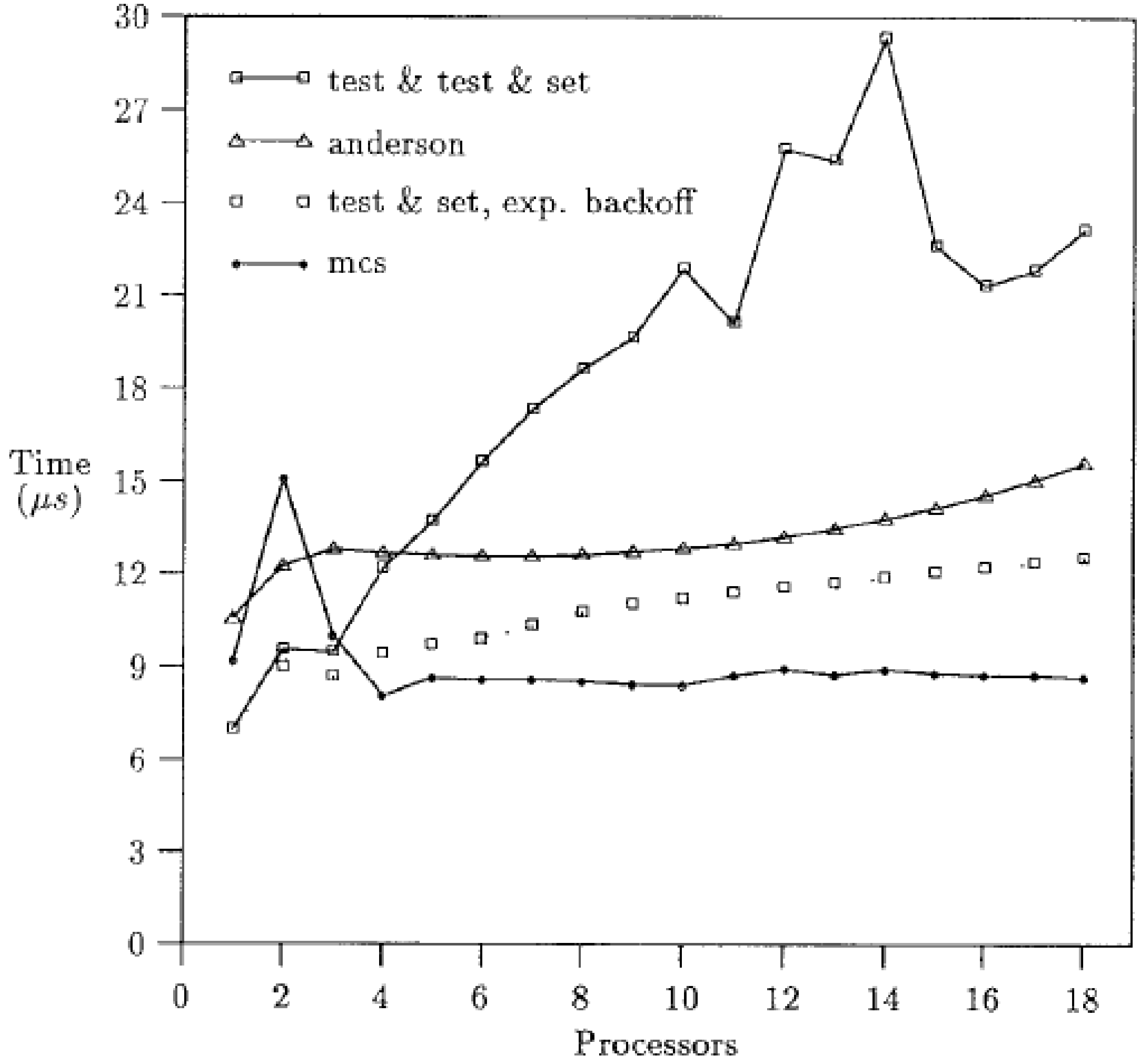


Fig. 17. Performance of spin locks on the Symmetry (empty critical section).

Confirmed Trade-off

- Queue locks scale well but have higher overhead
- Spin Locks have low overhead but don’t scale well

Other Hardware Provided SMP Synchronisation Primitives

- Atomic Add/Subtract
  - Can be used to implement counting semaphores
- Exchange
- Compare and Exchange
- Load linked; Store conditionally
  - Two separate instructions
    - Load value using *load linked*
    - Modify, and store using *store conditionally*
    - If value changed by another processor, or an interrupt occurred, then *store conditionally* failed
  - Can be used to implement all of the above primitives
  - Implemented without bus locking

Spinning versus Switching

- Remember spinning (busy-waiting) on a lock made little 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 option.
- On SMP systems, the decision to spin or block is not as clear.
  - The lock is held by another running CPU and will be freed without necessarily blocking the requestor

Spinning versus Switching

- Blocking and switching
  - to another process takes time
    - Save context and restore another
    - Cache contains current process not new process
      - » Adjusting the cache working set also takes time
    - TLB is similar to cache
  - Switching back when the lock is free encounters the same again
- Spinning wastes CPU time directly

- Trade off
  - If lock is held for less time than the overhead of switching to and back

  $\Rightarrow$ It's more efficient to spin

Spinning versus Switching

- The general approaches taken are
  - Spin forever
  - Spin for some period of time, if the lock is not acquired, block and switch
    - The spin time can be
      - Fixed (related to the switch overhead)
      - Dynamic
        - » Based on previous observations of the lock acquisition time

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!!!!!

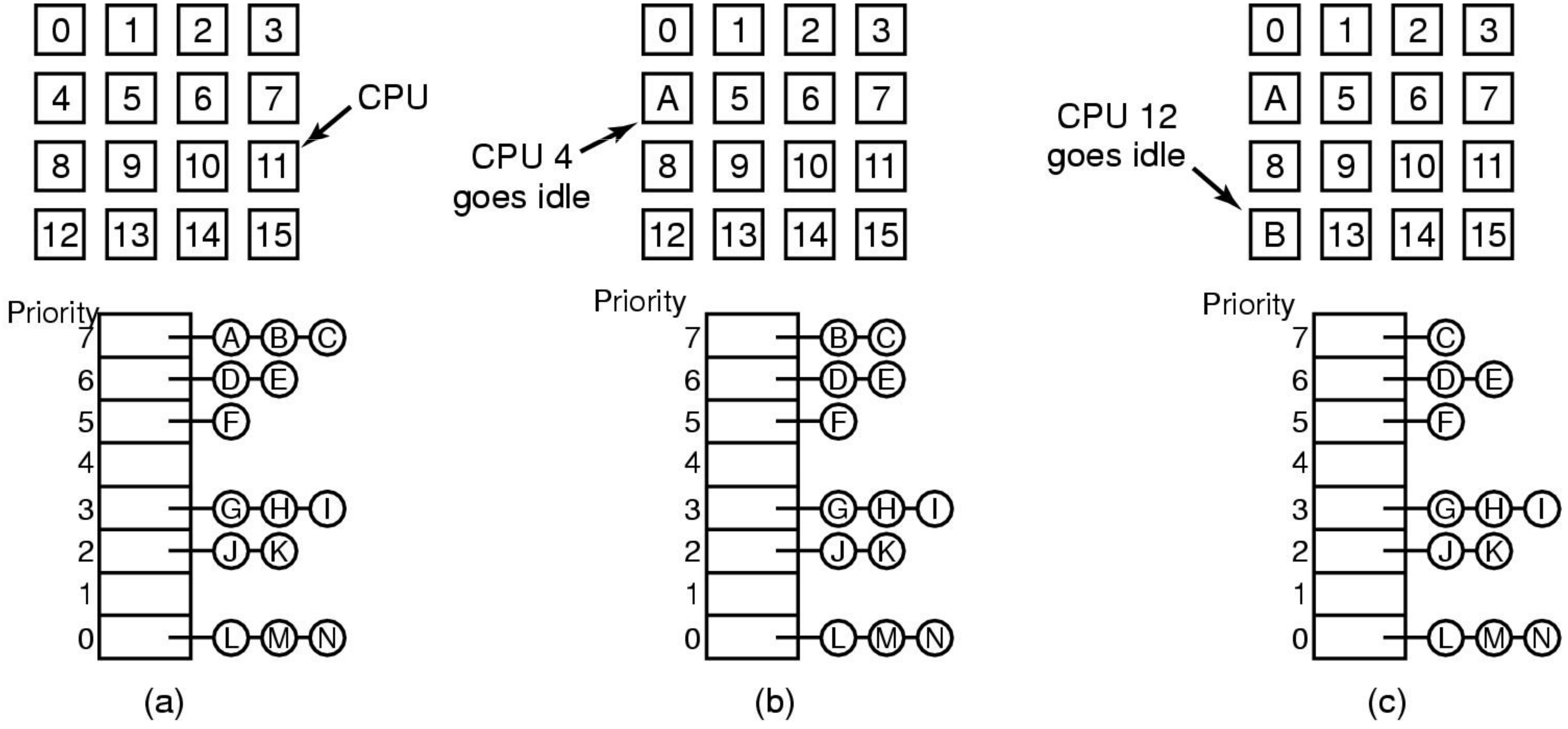
⇒ Spinlock implementations disable interrupts in addition to acquiring locks to avoid lock-holder preemption

Multiprocessor Scheduling

- Given *X* processes (or threads) and *Y* CPUs,
  – how do we allocate them to the CPUs

A Single Shared Ready Queue

- When a CPU goes idle, it take the highest priority process from the shared ready queue



Single Shared Ready Queue

- Pros
  - Simple
  - Automatic load balancing
- Cons
  - Lock contention on the ready queue can be a major bottleneck
    - Due to frequent scheduling or many CPUs or both
  - Not all CPUs are equal
    - The last CPU a process ran on is likely to have more related entries in the cache.

Affinity Scheduling

- Basic Idea
  - Try hard to run a process on the CPU it ran on last time
- One approach: *Two-level scheduling*

Two-level Scheduling

- Each CPU has its own ready queue
- Top-level algorithm assigns process to CPUs
  - Defines their affinity, and roughly balances the load
- The bottom-level scheduler:
  - Is the frequently invoked scheduler (e.g. on blocking on I/O, a lock, or exhausting a timeslice)
  - Runs on each CPU and selects from its own ready queue
    - Ensures affinity
  - If nothing is available from the local ready queue, it runs a process from another CPUs ready queue rather than go idle

Two-level Scheduling

- Pros
  - No lock contention on per-CPU ready queues in the (hopefully) common case
  - Load balancing to avoid idle queues
  - Automatic affinity to a single CPU for more cache friendly behaviour