

cse

SMP & Locking

THE UNIVERSITY OF
NEW SOUTH WALES

cse

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

COMP9242 2

THE UNIVERSITY OF
NEW SOUTH WALES

cse

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

(a) (b)

COMP9242

3

THE UNIVERSITY OF
NEW SOUTH WALES

cse

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

(a) (b)

COMP9242

4

THE UNIVERSITY OF
NEW SOUTH WALES

cse

NUMA

Hennessy & Patterson, Computer Architecture, a quantitative approach, 2nd ed.

COMP9242

5

THE UNIVERSITY OF
NEW SOUTH WALES

cse

Cache Coherency

- 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)?
 - Can be thought of as replication and migration of data between CPUs

(b)

6

THE UNIVERSITY OF
NEW SOUTH WALES

cse

Simplistic Goal

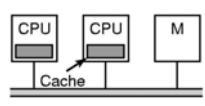
- Ideally, a read produces the result of the last write to the particular memory location?
 - Approaches that avoid the issue in software also avoid exploiting replication for parallelism
 - Typically, a hardware-based solution is used
 - Directory based
 - Snooping

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 7

cse

Snooping

- Each cache "broadcasts" transactions on the bus
- Each cache monitors the bus for transactions that affect its state



THE UNIVERSITY OF NEW SOUTH WALES COMP9242 8

cse

Example Coherence Protocol MESI

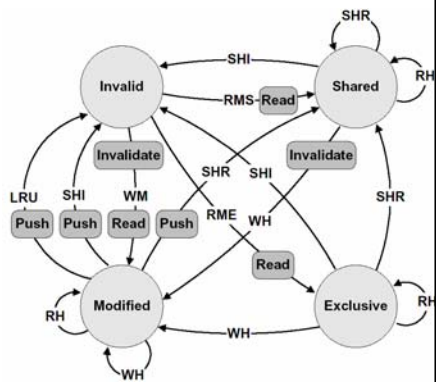
Each cache line is in one of four states

- Modified (M)
 - The line is valid in the cache and in only this cache.
 - The line is modified with respect to system memory—that is, the modified data in the line has not been written back to memory.
- Exclusive (E)
 - The addressed line is in this cache only.
 - The data in this line is consistent with system memory.
- Shared (S)
 - The addressed line is valid in the cache and in at least one other cache.
 - A shared line is always consistent with system memory. That is, the shared state is shared-unmodified; there is no shared-modified state.
- Invalid (I)
 - This state indicates that the addressed line is not resident in the cache and/or any data contained is considered not useful.

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 9

cse

- 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



THE UNIVERSITY OF NEW SOUTH WALES COMP9242 9

cse

Directory-based coherence example

- Each memory block has a home node
- Home node keeps directory of cache that have a copy
 - E.g., a bitmap of processors per memory block
- Pro
 - Invalidation/update messages can be directed explicitly
- Con
 - Requires more storage to keep directory
 - E.g. each 256 bits or memory requires 32 bits of directory

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 11

cse

Observation

- Locking primitives require exclusive access to the "lock"
- Care required to avoid excessive bus/interconnect traffic

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 12

cse

Focus on locking in the Common Case

- Bus-based UMA, per-CPU write-back caches, snooping coherence protocol.

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 13

cse

Kernel Locking

- Several CPUs can be executing kernel code concurrently.
- Need mutual exclusion on shared kernel data.
- Issues:
 - Lock implementation
 - Granularity of locking

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 14

cse

Mutual Exclusion Techniques

- Disabling interrupts (CLI — STI).
 - Unsuitable for multiprocessor systems.
- Spin locks.
 - Busy-waiting wastes cycles.
- Lock objects.
 - Flag (or a particular state) indicates object is locked.
 - Manipulating lock requires mutual exclusion.

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 15

cse

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 *)`
- `v = load_linked(lock *) / bool store_conditional(int, lock *)`
 - LL/SC can be used to implement all of the above

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 16

cse

Spin locks

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

- Busy waits. Good idea?

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 17

cse

Spin Lock Busy-waits Until Lock Is Released

- Stupid on uniprocessors, as nothing will change while spinning.
 - Should release (yield) CPU immediately.
- Maybe ok on SMPs: locker may execute on other CPU.
 - Minimal overhead (if contention low).
 - Still, should only spin for short time.
- Generally restrict spin locking to:
 - short critical sections,
 - unlikely to be contended by the same CPU.
 - local contention can be prevented
 - by design
 - by turning off interrupts

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 18

cse

Spinning versus Switching

- Blocking and switching
 - to another process takes time
 - Save context and restore another
 - Cache contains current process not new
 - » 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
 - ⇒ It's more efficient to spin

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 19

cse

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

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 20

cse

Interrupt Disabling

- Assume no local contention by design, is disabling interrupt important?
- Hint: What happens if a lock holder is preempted (e.g., at end of its timeslice)?
- All other processors spin until the lock holder is re-scheduled

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 21

cse

Alternative: Conditional Lock

```
bool cond lock (volatile lock t *l) {
    if (test and set(1))
        return FALSE; //couldn't lock
    else
        return TRUE; //acquired lock
}
```

- Can do useful work if fail to acquire lock.
- **But** may not have much else to do.
- Starvation: May never get lock!

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 22

cse

More Appropriate Mutex Primitive:

```
void mutex lock (volatile lock t *l) {
    while (1) {
        for (int i=0; i<MUTEX N; i++)
            if (!test and set(1))
                return;
        yield();
    }
}
```

- Spins for limited time only
 - assumes enough for other CPU to exit critical section
- Useful if critical section is shorter than N iterations.
- Starvation possible.

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 23

cse

Common Multiprocessor Spin Lock

```
void mp spinlock (volatile lock t *l) {
    cli(); // prevent preemption
    while (test and set(1)) ; // lock
}
void mp unlock (volatile lock t *l) {
    *l = 0;
    sti();
}
```

- Only good for short critical sections
- Does not scale for large number of processors
- Relies on bus-arbitrator for fairness
- Not appropriate for user-level
- Used in practice in small SMP systems

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 24

cse

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

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 25

cse

Compares Simple Spinlocks

- Test and Set


```
void lock (volatile lock_t *l) {
    while (test_and_set(l)) ;
}
```
- Test and Test and Set


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

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 26

cse

test_and_test_and_set LOCK

- Avoid bus traffic contention caused by test_and_set until it is likely to succeed
- Normal read spins in cache
- Can starve in pathological cases

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 27

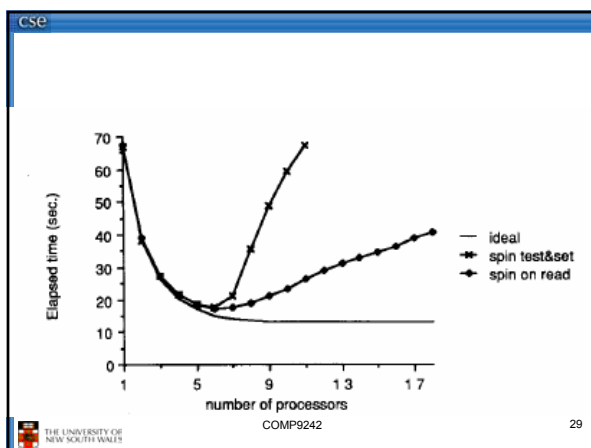
cse

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)

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 28



cse

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

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 30

cse

Idea

- Can inserting delays reduce bus traffic and improve performance
- Explore 2 dimensions
 - Location of delay
 - Insert a delay after release prior to attempting acquire
 - Insert a delay after each memory reference
 - Delay is static or dynamic
 - Static – assign delay “slots” to processors
 - Issue: delay tuned for expected contention level
 - Dynamic – use a back-off scheme to estimate contention
 - Similar to ethernet
 - Degrades to static case in worst case.

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 31

cse

Examining Inserting Delays

TABLE III
DELAY AFTER SPINNER NOTICES RELEASED LOCK

Lock	<pre>while (lock = BUSY or TestAndSet (Lock) = BUSY) begin while (lock = BUSY) ; Delay (); end;</pre>
------	---

TABLE IV
DELAY BETWEEN EACH REFERENCE

Lock	<pre>while (lock = BUSY or TestAndSet (lock) = BUSY) Delay ();</pre>
------	--

cse

Queue Based Locking

- Each processor inserts itself into a waiting queue
 - It waits for the lock to free by spinning on its own separate cache line
 - Lock holder frees the lock by “freeing” the next processors cache line.

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 33

cse

Results

number of processors	spin on read	static release	backoff rel.	static ref.	backoff ref.	queue
1	0.5	0.5	0.5	0.5	0.5	0.5
5	1.5	0.5	0.5	0.5	0.5	0.5
9	3.5	0.5	0.5	0.5	0.5	0.5
13	10.0	0.5	0.5	0.5	0.5	0.5
17	18.0	0.5	0.5	0.5	0.5	0.5

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 34

cse

Results

- Static backoff has higher overhead when backoff is inappropriate
- Dynamic backoff has higher overheads when static delay is appropriate
 - as collisions are still required to tune the backoff time
- Queue is better when contention occurs, but has higher overhead when it does not.
 - Issue: Preemption of queued CPU blocks rest of queue (worse than simple spin locks)

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 35

cse

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

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 36

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)

Shared memory

CPU 1 holds the real lock

CPU 2 spins on this (private) lock

CPU 3 spins on this (private) lock

CPU 4 spins on this (private) lock

When CPU 1 is finished with the real lock, it releases it and also releases the private lock CPU 2 is spinning on

MCS Lock

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

```

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

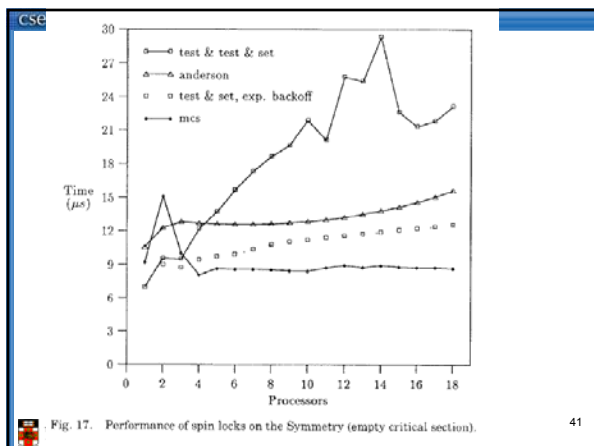
// parameter I, below, points to a qnode record allocated
// (in an enclosing scope) in shared memory locally-accessible
// to the invoking processor

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
 - Anderson's array based queue
 - test and set with exponential back-off
 - MCS



Confirmed Trade-off

- Queue locks scale well but have higher overhead
- Spin Locks have low overhead but don't scale well
- What do we use?

cse

- Beng-Hong Lim and Anant Agarwal, "Reactive Synchronization Algorithms for Multiprocessors", *ASPLOS VI*, 1994

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 43

cse

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 44

cse

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 45

cse

Idea

- Can we dynamically switch locking methods to suit the current contention level???

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 46

cse

Issues

- How do we determine which protocol to use?
 - Must not add significant cost
- How do we correctly and efficiently switch protocols?
- How do we determine when to switch protocols?

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 47

cse

Protocol Selection

- Keep a "hint"
- Ensure both TTS and MCS lock are never free at the same time
 - Only correct selection will get the lock
 - Choosing the wrong lock will result in a retry which can get it right next time
 - Assumption: Lock mode changes infrequently
 - hint cached read-only
 - infrequent protocol mismatch retries

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 48

cse

Changing Protocol

- Only lock holder can switch to avoid race conditions
 - It chooses which lock to free, TTS or MCS.

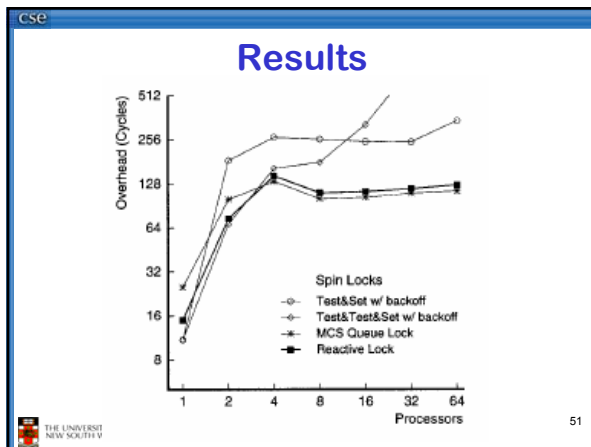
THE UNIVERSITY OF NEW SOUTH WALES COMP9242 49

cse

When to change protocol

- Use threshold scheme
 - Repeated acquisition failures will switch mode to queue
 - Repeated immediate acquisition will switch mode to TTS

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 50



cse

Have we found the perfect locking scheme?

- No!!
- What about preemption of the lock holder?
- For queue-based locking scheme, we switch to the next in queue:
 - What happens if the next in queue is preempted?
 - Multiprogramming increases chance of preemption, even though contention may not be high

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 52

cse

Kontothanassis, Wisniewski, and Scott, "Scheduler-Conscious Synchronisation", *ACM Transactions on Computer Systems*, Vol. 15, No. 1, 1997

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 53

cse

- Preemption safe lock
 - it never spin for more than a constant time
 - employs only kernel extension to avoid its own preemption in critical sections
- Scheduler conscious lock
 - interacts with the scheduler to determine or alter state of other threads

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 54

cse

Preemption Control

- Share state/interface between kernel and lock primitive such that
 - Application can indicate no preemption
 - set a *unpreemptable_self* bit
 - Kernel does not preempt lock holders
 - If time slice expires, *warning* bit is set
 - If time slices expires again, preemption occurs
 - If lock finds warning bit set, it yields to reset it.
 - L4 provides similar scheme

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 55

cse

Scheduler Conscious

- Two extra states of **other** threads
 - preempted: Other thread is preempted
 - unpreemptable_other: Mark other thread as unpreemptable so we can pass the lock on
 - State is visible to lock contenders

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 56

cse

Examined

- TAS-B
 - Test and set with back-off
- TAS-B-PS
 - Test and set with back-off and uses kernel interface to avoid preemption of lock holder
- Queue
 - Standard MCS lock
- Queue-NP
 - MCS lock using kernel interface to avoid preemption of lock holder
- Queue-HS
 - Queue-NP + handshake to avoid hand over to preempted process
 - Receiver of lock must ack via flag in lock within bounded time, otherwise preemption assumed

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 57

cse

Examined

- Smart-Q
 - Uses "scheduler conscious" kernel interface to avoid passing lock to preempted process
 - Also marks successor as unpreemptable_other
- Ticket
 - Normal ticket lock with back-off
- Ticket-PS
 - Ticket lock with back-off and preemption safe using kernel interface, and a handshake.
- Native
 - Hardware supported queue lock
- Native-PS
 - Hardware supported queue lock using kernel interface to avoid preemption in critical section

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 58

cse

11 Processor SGI challenge
Loop consisting of critical and non-critical sections

secs

multiprogramming level

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 59

cse

Conclusions

- Scalable queue locks very sensitive to degree of multiprogramming
 - Preemption of process in queue the major issue
- Significant performance benefits if
 - Avoid preemption of lock-holders
 - Avoiding passing lock to preempted process in the case of scalable queue locks

THE UNIVERSITY OF NEW SOUTH WALES COMP9242 60