SMP & Locking

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’ll focus on UMA

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

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 solution is used
    - Directory based – typically for non-bus interconnects
    - Snooping – typically for bus-based architectures
Snooping

- Each cache "broadcasts" transactions on the bus
- Each cache monitors the bus for transactions that affect its state
- Typically use "MESI" protocol in bus-based architectures
- How does snooping work with multiple levels of caches?
  - inclusion property: In \supset Ln
  - multi-level snooping

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.

Directory-based coherence example

- Each memory block has a home node
- Home node keeps directory of cache that have a copy
- Pro
  - Invalidation/update messages can be directed explicitly
  - Requires more storage to keep directory
    - E.g. each 256 bits or memory requires 32 bits of directory
- Con
  - At least one CPU must load the other’s new value

Interaction with Memory Architecture

- Example: critical section
  /* counter++ */
  load r1, counter
  add r1, r1, 1
  store r1, counter
  /* unlock(mutex) */
  store zero, mutex
- Relies on all CPUs seeing update of counter before update of mutex
- Depends on assumptions about ordering of stores to memory

Memory Models: Strong Ordering

- Loads and stores execute in program order
- Memory accesses of different CPUs are sequentialised
- Traditionally used by many architectures
  - At least one CPU must load the other’s new value
Other Memory Models

- Modern hardware features can interfere with store order:
  - write buffer (or store buffer or write-behind buffer)
  - instruction reordering (out-of-order completion)
  - superscalar execution
  - Pipelining
- Each CPU keeps its own data consistent, but how about others?
  - Multiprocessing
  - DMA

Total Store Ordering

- Stores go to write buffer to hide memory latency
- Loads read from write buffer if possible
-Stores are guaranteed to occur in FIFO order

Partial Store Ordering

- All stores go to write buffer
- Loads read from write buffer if possible
- Redundant stores are cancelled
  - Store to mutex can overtake store to counter
  - Need to use memory barrier
  - Failure to do so will introduce subtle bugs:
    - changing process state after saving context
    - initiating I/O before setting up parameter buffer

Observation

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

Focus on locking in the Common Case

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

Kernel Locking

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

- Expensive communication
  - distributed data structures
- Fast communication
  - shared data structures

Scheduling domain
- Hardware thread contexts of an SMT core
- Multiple cores on a chip with fast cache migration, inter-core interrupts
- Separate domains where communication is slow
- Multiple cores without shared caches
  - Bus-connected processors

Lock Granularity

- Fine-grained vs coarse-grained?
  - tradeoff is highly dependent on
    - length of system calls
    - number of fine-grained locks required
    - cost of individual locks
    - ...

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.

Hardware Provided Locking Primitives

```c
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
```

Spin locks

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

- Busy waits. Good idea?
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 contented by the same CPU.
  - local contention can be prevented
    - by design
    - by turning off interrupts
- Assumptions:
  - 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.

Spinning versus Switching

- Blocking and switching
  - to another process takes time
  - Cache contains current process not new
  - 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

Spin forever

The general approaches taken are

- Spinning 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

Alternative: Conditional Lock (TryLock)

```c
bool cond lock (volatile lock t *l) {
    if (test and set(l))
        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!

More Appropriate Mutex Primitive:

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

- 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.
Common Multiprocessor Spin Lock

```c
void mp spinlock (volatile lock_t *l) {
    cli(); // prevent preemption
    while (test_and_set(l)) ; // look
}
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


Compares Simple Spinlocks

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

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

- 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

Benchmark

```c
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)
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 lock holder is slower
  – Slower lock holder results in more contention

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.

Examining Inserting Delays

<table>
<thead>
<tr>
<th>TABLE III: DELAY AFTER SPINNH NOTICES RELEASED LOCK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE IV: DELAY BETWEEN EACH REFERENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

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.

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)

MCS Locks
• Each CPU enqueues its own private lock variable into a queue and spins on it.
  - No reclamation
  - CPU 3 spins on this (private) lock
  - CPU 2 spins on this (private) lock
  - CPU 4 spins on this (private) lock
  - Shared memory
  - CPU 1 holds the next lock

MCS Lock
• Requires
  – compare_and_swap()
  – exchange()
    • Also called fetch_and_store()

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?

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

Idea

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

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?
Protocol Selection

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

Changing Protocol

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

When to change protocol

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

Results

![Graph showing lock performance metrics for different protocols over processors]

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

• Preemption safe lock
  – It never spins for more than a constant time
  – empleys 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

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.
  – Historical L4 provides similar scheme

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

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

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

11 Processor SGI challenge
Loop consisting of critical and non-critical sections
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
  - To a lesser extent, avoiding passing lock to preempted process in the case of scalable queue locks