Multi-Object Synchronization
Multi-Object Programs

- What happens when we try to synchronize across multiple objects in a large program?
  - Each object with its own lock, condition variables
  - Is locking modular?
- Performance
- Semantics/correctness
- Deadlock
- Eliminating locks
Synchronization Performance

• A program with lots of concurrent threads can still have poor performance on a multiprocessor:
  – Overhead of creating threads, if not needed
  – Lock contention: only one thread at a time can hold a given lock
  – Shared data protected by a lock may ping back and forth between cores
  – False sharing: communication between cores even for data that is not shared
Topics

• Multiprocessor cache coherence
• MCS locks (if locks are mostly busy)
• RCU locks (if locks are mostly busy, and data is mostly read-only)
Multiprocessor Cache Coherence

• Scenario:
  – Thread A modifies data inside a critical section and releases lock
  – Thread B acquires lock and reads data
• Easy if all accesses go to main memory
  – Thread A changes main memory; thread B reads it
• What if new data is cached at processor A?
• What if old data is cached at processor B
Write Back Cache Coherence

- Cache coherence = system behaves as if there is one copy of the data
  - If data is only being read, any number of caches can have a copy
  - If data is being modified, at most one cached copy
- On write: (get ownership)
  - Invalidate all cached copies, before doing write
  - Modified data stays in cache (“write back”)
- On read:
  - Fetch value from owner or from memory
Cache State Machine

- Invalid
- Read miss
- Peer write
- Write miss
- Peer write
- Peer read
- Exclusive (writable)
- Read-Only
- Write hit
Directory-Based Cache Coherence

• How do we know which cores have a location cached?
  – Hardware keeps track of all cached copies
  – On a read miss, if held exclusive, fetch latest copy and invalidate that copy
  – On a write miss, invalidate all copies

• Read-modify-write instructions
  – Fetch cache entry exclusive, prevent any other cache from reading the data until instruction completes
// A counter protected by a spinlock
Counter::Increment() {
    while (test_and_set(&lock))
        ;
    value++;
    lock = FREE;
    memory_barrier();
}
A Simple Test of Cache Behavior

Array of 1K counters, each protected by a separate spinlock

– Array small enough to fit in cache

• Test 1: one thread loops over array
• Test 2: two threads loop over different arrays
• Test 3: two threads loop over single array
• Test 4: two threads loop over alternate elements in single array
Results (64 core AMD Opteron)

One thread, one array 51 cycles
Two threads, two arrays 52
Two threads, one array 197
Two threads, odd/even 127
Reducing Lock Contention

• Fine-grained locking
  – Partition object into subsets, each protected by its own lock
  – Example: hash table buckets

• Per-processor data structures
  – Partition object so that most/all accesses are made by one processor
  – Example: per-processor heap

• Ownership/Staged architecture
  – Only one thread at a time accesses shared data
  – Example: pipeline of threads
What If Locks are Still Mostly Busy?

• MCS Locks
  – Optimize lock implementation for when lock is contended

• RCU (read-copy-update)
  – Efficient readers/writers lock used in Linux kernel
  – Readers proceed without first acquiring lock
  – Writer ensures that readers are done

• Both rely on atomic read-modify-write instructions
The Problem with Test and Set

Counter::Increment() {
    while (test_and_set(&lock)) {
        value++;
        lock = FREE;
        memory_barrier();
    }
}

What happens if many processors try to acquire the lock at the same time?

- Hardware doesn’t prioritize FREE
Counter::Increment() {
    while (lock == BUSY && test_and_set(&lock)) {
        value++;
        lock = FREE;
    }
    memory_barrier();
}

What happens if many processors try to acquire the lock?

- Lock value pings among caches
Test (and Test) and Set Performance

![Graph showing the time to execute a critical section vs. number of processors for different lock types: Test-And-Set Lock, Test-And-Test-And-Set Lock, and MCS Lock. The graph illustrates the performance comparison among these lock types.]
Some Approaches

• Insert a delay in the spin loop
  – Helps but acquire is slow when not much contention

• Spin adaptively
  – No delay if few waiting
  – Longer delay if many waiting
  – Guess number of waiters by how long you wait

• MCS
  – Create a linked list of waiters using compareAndSwap
  – Spin on a per-processor location
Atomic CompareAndSwap

- Operates on a memory word
- Check that the value of the memory word hasn’t changed from what you expect
  - E.g., no other thread did CompareAndSwap first
- If it has changed, return an error (and loop)
- If it has not changed, set the memory word to a new value
MCS Lock

• Maintain a list of threads waiting for the lock
  – Front of list holds the lock
  – MCSLock::tail is last thread in list
  – New thread uses CompareAndSwap to add to the tail
• Lock is passed by setting next->needToWait = FALSE;
  – Next thread spins while its needToWait is TRUE

```c
TCB {
    TCB *next;       // next in line
    bool needToWait;
}
MCSLock {
    Queue *tail = NULL; // end of line
}
```
MCS Lock Implementation

MCSLock::acquire() {
    Queue *oldTail = tail;

    myTCB->next = NULL;
    myTCB->needToWait = TRUE;
    while (!compareAndSwap(&tail, oldTail, &myTCB)) {
        oldTail = tail;
    }
    if (oldTail != NULL) {
        oldTail->next = myTCB;
        memory_barrier();
        while (myTCB->needToWait) ;
    }
}

MCSLock::release() {
    if (!compareAndSwap(&tail, myTCB, NULL)) {
        while (myTCB->next == NULL) ;
        myTCB->next->needToWait = FALSE;
    }
}
MCS In Operation

a) TAIL → NIL

b) A:  
   next  needToWait
   NIL   FALSE

c) A:  
   B  FALSE
   B:   
   NIL TRUE

d) A:  
   B  FALSE
   B:   
   C   TRUE
   C:   
   NIL TRUE

e) B:  
C:   
   NIL TRUE

f) NIL FALSE
Read-Copy-Update

• Goal: very fast reads to shared data
  – Reads proceed without first acquiring a lock
  – OK if write is (very) slow

• Restricted update
  – Writer computes new version of data structure
  – Publishes new version with a single atomic instruction

• Multiple concurrent versions
  – Readers may see old or new version

• Integration with thread scheduler
  – Guarantee all readers complete within grace period, and then garbage collect old version
Read-Copy-Update

Update is Published

Read (Old)
Read (Old)
Read (Old or New)
Write (New)
Read (Old or New)
Grace Period

Time

Grace Period Ends

Read (New)
Read (New)
Delete (Old)
Read-Copy-Update Implementation

• Readers disable interrupts on entry
  – Guarantees they complete critical section in a timely fashion
  – No read or write lock

• Writer
  – Acquire write lock
  – Compute new data structure
  – Publish new version with atomic instruction
  – Release write lock
  – Wait for time slice on each CPU
  – Only then, garbage collect old version of data structure
Non-Blocking Synchronization

• Goal: data structures that can be read/modified without acquiring a lock
  – No lock contention!
  – No deadlock!
  – (No priority inversion!)

• General method using CompareAndSwap
  – Create copy of data structure
  – Modify copy
  – Swap in new version iff no one else has already posted a change
  – Restart if pointer has changed
Push(*stack, *entry) {
  pointer_t old_top;
  do {
    old_top = stack->top;
    entry->next.ptr = old_top.ptr;
  } while (!CAS(&(stack->top),
    old_top,
    <entry,
    old_top.count>));
}

entry *Pop(Stack *stack) {
  pointer_t old_top;
  entry *top;
  do {
    old_top = stack->top;
    top = old_top.ptr;
    if ( top == NULL)
      return NULL;
  } while (!CAS(&(stack->top),
    old_top,
    <top->next.ptr,
    old_top.count+1>));
  return top;
}