Synchronization: Performance and Multi-Object
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
  – Usual lock semantics
  – Optimized for case that locks are mostly busy
• RCU locks
  – Relaxed semantics (somewhat like readers/writers)
  – Optimized for 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

• Caching
  – 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

- Read miss
- Peer write
- Write miss
- Peer write
- Read-Only
- Peer read
- Write hit
- Exclusive (writable)
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 Simple Critical Section

// 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)

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>One thread, one array</td>
<td>51 cycles</td>
</tr>
<tr>
<td>Two threads, two arrays</td>
<td>52</td>
</tr>
<tr>
<td>Two threads, one array</td>
<td>197</td>
</tr>
<tr>
<td>Two threads, odd/even</td>
<td>127</td>
</tr>
</tbody>
</table>
Reducing Lock Contention

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

• **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**
  - Memory system-aware, optimized lock implementation for when lock is contended

- **RCU (read-copy-update)**
  - Efficient readers/writers lock used in Linux kernel
  - Readers never block
  - Writer updates while readers operate (!)

- Both rely on atomic read-modify-write instructions
Lock Performance: 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
The Problem with Test and Test and Set

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 between caches
Test (and Test) and Set Performance

![Graph showing performance of Test and Set Locks and MCS Locks.](image-url)
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

TCB {
  TCB *next; // next in line
  bool needToWait;
}
MCSLock {
  Queue *tail = NULL; // end of line
}
MCS In Operation

a) TAIL → NIL

b) A:  
   next | needToWait
   NIL  | FALSE

T AIL

c) A:  
   B   | FALSE

T AIL

B:  
   NIL  | TRUE

d) A:  
   B   | FALSE

T AIL

B:  
   C   | TRUE

C:  
   NIL  | TRUE

e) B:  
   C   | FALSE

T AIL

C:  
   NIL  | TRUE

f) T AIL

NIL  | FALSE
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;
    }
}
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 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
Deadlock Definition

- Resource: any (passive) thing needed by a thread to do its job (CPU, disk space, memory, lock)
  - Preemptable: can be taken away by OS
  - Non-preemptable: must leave with thread
- Starvation: thread waits indefinitely
- Deadlock: circular waiting for resources
  - Deadlock => starvation, but not vice versa
Example: two locks

Thread A
lock1.acquire();
lock2.acquire();
lock2.release();
lock1.release();

Thread B
lock2.acquire();
lock1.acquire();
lock1.release();
lock2.release();
Bidirectional Bounded Buffer

Thread A

buffer1.put(data);
buffer1.put(data);
buffer2.get();
buffer2.get();

Thread B

buffer2.put(data);
buffer2.put(data);
buffer1.get();
buffer1.get();

Suppose buffer1 and buffer2 both start almost full.
Two locks and a condition variable

Thread A

lock1.acquire();
...
lock2.acquire();
while (need to wait) {
    condition.wait(lock2);
}
lock2.release();
...
lock1.release();

Thread B

lock1.acquire();
...
lock2.acquire();
condition.signal(lock2);
...
lock2.release();
...
lock1.release();
Yet another Example
Dining Lawyers

Each lawyer needs two chopsticks to eat. Each grabs chopstick on the right first.
Necessary Conditions for Deadlock

• Limited access to resources
  – If infinite resources, no deadlock!

• No preemption
  – If resources are virtual, can break deadlock

• Multiple independent requests
  – “wait while holding”

• Circular chain of requests
Question

• How does Dining Lawyers meet the necessary conditions for deadlock?
  – Limited access to resources
  – No preemption
  – Multiple independent requests (wait while holding)
  – Circular chain of requests

• How can we modify Dining Lawyers to prevent deadlock?
Preventing Deadlock

• Exploit or limit program behavior
  – Limit program from doing anything that might lead to deadlock

• Predict the future
  – If we know what program will do, we can tell if granting a resource might lead to deadlock

• Detect and recover
  – If we can rollback a thread, we can fix a deadlock once it occurs
Exploit or Limit Behavior

• Provide enough resources
  – How many chopsticks are enough?

• Eliminate wait while holding
  – Release lock when calling out of module

• Eliminate circular waiting
  – Lock ordering: always acquire locks in a fixed order
  – Example: move file from one directory to another
Example

Thread 1

1. Acquire A
2.
3. Acquire C
4.
5. If (maybe) Wait for B

Thread 2

1.
2. Acquire B
3.
4. Wait for A

How can we make sure to avoid deadlock?
Deadlock Dynamics

• Safe state:
  – For any possible sequence of future resource requests, it is possible to eventually grant all requests
  – May require waiting even when resources are available!

• Unsafe state:
  – Some sequence of resource requests can result in deadlock

• Doomed state:
  – All possible computations lead to deadlock
Possible System States

Unsafe

Deadlock

Safe
Question

- What are the doomed states for Dining Lawyers?
- What are the unsafe states?
- What are the safe states?
Communal Dining Lawyers

- $n$ chopsticks in middle of table
- $n$ lawyers, each can take one chopstick at a time
- What are the safe states?
- What are the unsafe states?
- What are the doomed states?
Communal Mutant Dining Lawyers

• N chopsticks in the middle of the table
• N lawyers, each takes one chopstick at a time
• Lawyers need k chopsticks to eat, \( k > 1 \)

• What are the safe states?
• What are the unsafe states?
• What are the doomed states?
Predict the Future

• Banker’s algorithm
  – State maximum resource needs in advance
  – Allocate resources dynamically when resource is needed -- wait if granting request would lead to deadlock
  – Request can be granted if some sequential ordering of threads is deadlock free
Banker’s Algorithm

• Grant request iff result is a safe state
• Sum of maximum resource needs of current threads can be greater than the total resources
  – Provided there is some way for all the threads to finish without getting into deadlock
• Example: proceed iff
  – total available resources - # allocated >= max remaining that might be needed by this thread in order to finish
  – Guarantees this thread can finish
Detect and Repair

• Algorithm
  – Scan wait for graph
  – Detect cycles
  – Fix cycles

• Proceed without the resource
  – Requires robust exception handling code

• Roll back and retry
  – Transaction: all operations are provisional until have all required resources to complete operation
Detecting Deadlock
Non-Blocking Synchronization

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

• General method using compareAndSwap
  – Create copy of data structure
  – Modify copy
  – Swap in new version iff no one else has
  – Restart if pointer has changed
Lock-Free Bounded Buffer

tryget() {
    do {
        copy = ConsistentCopy(p);
        if (copy->front == copy->tail)
            return NULL;
        else {
            item = copy->buf[copy->front % MAX];
            copy->front++;
        }
    } while (compareAndSwap(&p, p, copy));
    return item;
}