Synchronization: Performance and Multi-Object

Module 8
Topics

• Readers/Writers Locks
  • Class exercise...

• Performance: 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
Readers/Writers Locks
Enabling Concurrency

• Imagine you’re creating a thread-safe implementation of some data structure
• The interface is read(key) and put(key, value)
• Each instance of the data structure contains a single mutex that is used to restrict concurrent operations

• Does put() need to obtain the mutex?
• Does read() need to obtain the mutex?
Enabling Concurrency

• Imagine you’re creating a thread-safe implementation of some data structure
• The interface is read(key) and put(key, value)
• Each instance of the data structure contains a single mutex that is used to restrict concurrent operations

• Does put() need to obtain the mutex?
• Does read() need to obtain the mutex?

• Usually the answer to both questions is “yes”
Readers/Writers Locks

- Mutex has semantics “one thread at a time”

- Suppose we want semantics
  - any number of readers but no writers
  - OR
  - just one writer

- Readers/writers locks support this
  - lock for read or lock for write

- Interface:
  - startRead() ... doneRead
  - startWrite() ... doneWrite()
R/W Locks Implementation

• Take a few minutes and implement them
  • In teams
  • (Heaven help us...)

• The text advocates a “monitor style” programming discipline
  • Implement an abstract data type as a class
  • Each instance contains a lock
  • Every method acquires the lock as the first thing it does
  • Every method releases the lock as the last thing it does
  • What should your code do if it needs to wait?
void startRead() {
    lock.lock();
    while ( numWriters > 0 ) wait(readWaitCV, lock);
    numReaders++;
    lock.unlock();
}

void endRead() {
    lock.lock();
    if ( --numReaders == 0 ) signal(writeWaitCV);
    lock.unlock();
}
R/W Locks Implementation

```java
void startWrite() {
    lock.lock();
    while (numWriters > 0 || numReaders > 0)
        wait(writeWaitCV, lock);
    numWriters = 1;
    lock.unlock();
}
void endWrite() {
    lock.lock();
    numWriters = 0;
    broadcast(readWaitCV);
    signal(writeWaitCV);
    lock.unlock();
}
```
R/W Lock Implementation

- What’s good about my implementation?
  - It works!

- What’s bad about my implementation?
  - “starvation”

- What alternative semantics might you want?
Synchronization Performance: Caches
Multi-threaded/core Performance

Parallel, numerical application. N reflects size of data (granularity).
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
Multicore Caching

Main Memory

Memory Interconnect

Cache

Core
Performance: Multiprocessor Cache Coherence

• (Cache) Coherence vs. (Memory) Consistency
  • Consistency: view of values in multiple locations across cores
    • last module
  • Coherence: view of a single location’s value across cores
    • this module

• 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

• With 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

Invalid → Peer write

Peer write → Write miss

Write miss → Peer write

Peer write → Peer read

Peer read → Shared (Read-only)

Shared (Read-only) → Write hit

Write hit → Exclusive (writable)

Exclusive (writable) → Peer write

This is one simple example of a possible state machine.
Cache Coherence

• How do we know which cores have a location cached?
  • Snooping – shared bus; all cores see transactions
  • Directory Based
    • Better scalability than snooping
    • 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
  • Atomically fetch cache entry exclusive and update
    • prevents any other cache from reading or writing the data until instruction completes
A Simple Critical Section

// A counter protected by a spinlock
Counter::Increment() {
    while (test_and_set(&lock))
        ;
    value++;
    memory_barrier();
    lock = FREE;  // atomic write
}
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

time to execute one Increment()
Lock Performance:
The Problem with Test-and-Set

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

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

- Threads trying to get lock acquire cache line ownership
- Hardware doesn’t prioritize FREE
Test-and-Test-and-Set

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

What happens if many processors try to acquire the lock?
  • Lock value pings among caches
Test(-and-Test)-and-Set Performance

![Graph showing the performance of Test-And-Set Lock and Test-And-Test-And-Set Lock over the number of processors. The graph indicates that Test-And-Set Lock has better performance than Test-And-Test-And-Set Lock.]
Some Possible 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

• Reduce Lock Contention
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
Test(-and-Test)-and-Set Performance

Graph showing the time to execute a critical section for different lock mechanisms as the number of processors increases. The graph compares Test-And-Set Lock, Test-And-Test-And-Set Lock, and MCS Lock.
Background: Atomic CompareAnd Swap Instruction

• CompareAndSwap( memory address, comparison value, update value );

• Atomically
  if ( value at memory address == comparison value ) {
    value at memory address = update value;
    return true;
  }
else return false;
MCS Lock

```c
// thread control block (per thread)
TCB {
    TCB *next;       // next in line
    bool needToWait; // per-thread flag
}

MCSLock {
    Queue *tail = NULL; // end of line
}
```

- 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

- Threads spin on their private needToWait flags

- Lock is passed by thread releasing the lock setting
  `next->needToWait = FALSE;`
MCS In Operation

For this to work, must be able to do each required operation in one (atomic) CompareAndSwap instruction.
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;
    }
}

Why is this fast?
- Under low lock contention
- At high lock contention
Read-Copy-Update Locks
Read-Copy-Update

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

- Multiple concurrent versions
  - Readers may see old version for a limited time

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

- Relies on integration with thread scheduler
  - Guarantee all readers complete within grace period, and then garbage collect old version
Read-Copy-Update

The diagram illustrates the Read-Copy-Update process. When an update is published, all reads are directed to the old version, indicated by 'Read (Old)'. During the grace period, reads are performed on both the old and new versions, denoted by 'Read (Old or New)'. When the grace period ends, reads are directed to the new version, shown as 'Read (New)', and any old versions are deleted, indicated by 'Delete (Old)'. The timeline is marked with 'Time'.
RCU Lock Basic Idea

• Atomic update to switch to next version of data structure
  • No problem

• Don’t know if any readers are still using an old version
  • Problem: can’t “clean up” old versions as part of publishing new versions

• Solution: version generation numbers
  • Increment a generation number (counter) associated with data structure each time a new version is published
  • Each thread exposes the highest version number it has seen
  • So... just wait until all threads are saying they’ve seen at least version N to clean up versions before N

• RCU Locks: do that, but on a processor basis rather than a thread basis
Read-Copy-Update Implementation

• Readers *disable interrupts on entry*
  • Guarantees they complete critical section in a timely fashion
  • Prevents scheduler from running on that core
  • No read or write lock

• Writer
  • Acquire write lock
    • One writer at a time
  • Compute new data structure
  • Publish new version with atomic instruction
  • Release write lock
  • Wait for scheduler time slice on each CPU
  • Only then, garbage collect old version of data structure
Writer Operation

**WriteLock();** // only one writer at a time

<*prepare updated data structure>*

**publish(updated data structure);** // make new version visible by CAS // pointer

**WriteUnlock();** // allow another writer to start

**synchronize();** // wait until all readers are at at least the version // you published

<*free anything that needs freeing from the version you replaced>*
RCU Lock Implementation

```c
void ReadLock() { disableInterrupts(); }
void ReadUnlock() { enableInterrupts(); }

void WriteLock() { writerSpin.lock(); }
void WriteUnlock() { writerSpin.unlock(); }

void publish( void **pp1, void *p2) {
    memory_barrier();
    *pp1 = p2;   // atomic assignment needed...
    memory_barrier();
}
```
RCU Lock Implementation

// called after each modification (after releasing write lock)
void synchronize() {
    c = atomicIncrement(globalCounter);
    for (p=0; p<NUM_CORES; p++ )
        while (PER_PROC_VAR(quiescentCount,p) < c)
            sleep(10);  // about a scheduling quantum
}

// called by scheduler
void QuiescentState() {
    memory_barrier();
    PER_PROC_VAR(quiescentCount) = globalCounter;
    memory_barrier();
}
RCU Lock Question

- We require that the new version of the update be published with a single, atomic instruction

- So, why do we need a write lock?
  - Why not just produce the updated data structure without a lock and then install it using the atomic instruction?
Deadlock
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();
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

1. Limited access to resources
   • If infinite resources, no deadlock!

2. No preemption
   • If resources are preemptable, can break deadlock

3. Hold and Wait
   • Threads don’t voluntarily give up resources

4. Circular chain of requests
Question

• How does Dining Lawyers meet the necessary conditions for deadlock?
  • Limited access to resources
  • No preemption
  • Hold and wait
  • Circular chain of requests

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

• Make sure at least one of the four conditions can’t hold by
  • 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
  • Acquire all locks at once, or none

• 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. Acquire C
3. If (cond) Acquire B

Thread 2

1. Acquire B
2. Wait for A

How can we “pause” thread execution to 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 (perhaps by delaying some requests)

• **Unsafe state:**
  
  • Some sequence of resource requests can result in deadlock, even if you reserve all remaining resources for one chosen thread (in case it will want them)

• **Doomed state:**
  
  • All possible computations lead to deadlock
Possible System States
Dining Lawyers

• What are the unsafe states?

• What are the safe states?

• What are the doomed states for Dining Lawyers?

• Note: In Dining Lawyers we know exactly what each thread will do. This dynamic approach to deadlock prevention is oriented toward situations where threads conditionally acquire resources
  • “I need up to two chopsticks (but sometimes I use just one chopstick to stab my food and I’m done)”
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?
Maybe 1, Maybe 2 Chopstick Lawyers

- Lawyers in a circle with a chopstick between each adjacent pair
- "Nobody’s going to tell me in what order I have to pick up my chopsticks!"
- If a lawyer is holding one chopstick, the next thing s/he might do is
  - Try to get the other chopstick, or
  - Stab and then put down the chopstick s/he is holding

- Note that the situation where every lawyer is holding a chopstick is not (necessarily) deadlock
- Note that the situation where every lawyer is holding one chopstick and waiting to get a second is deadlock
General Method: Banker’s Algorithm

Basic Setup

- There is a resource manager that “owns” all the resources
  - There can be many types of resources, all controlled by one manager
- Threads request resources from the manager and return them to the manager
  - Requests/allocations are one resource at a time

Dynamic Operation

- Threads state maximum resource needs to manager when they start
- Threads request resources dynamically as needed
- Manager delays granting request if doing so could lead to deadlock
- Manager grants request if some sequential ordering of threads is deadlock free
Banker’s Algorithm

• Grant request iff result is a safe state
  • **Simple Example:** proceed if total available resources - # allocated >= max remaining that might be needed by this thread in order to finish

• More generally, allocate resource if manager can find a way for all threads to eventually finish, even if each asks for its maximum request, even if the requested resource is allocated
  • Otherwise, don’t allocate and block thread until it’s safe to grant its request

• **Why would you want to go through all this trouble?**
  • Sum of maximum resource needs of sthreads can be greater than the total resources
  • No static ordering (or any other) constraints on acquiring resources that have to be respected in the code
Another Approach: 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
Yet Another Approach: Non-blocking algorithms

• An algorithm is **non-blocking** if a slow thread cannot prevent another faster thread from making progress
  - Using locks is not non-blocking because a thread may acquire the lock and then run really really slowly
    • (Why?)

• Non-blocking algorithms are often built on an atomic hardware instruction, Compare And Swap (CAS)

```cpp
bool CAS(ptr, old, new) {
    if ( *ptr == old ) { *ptr = new; return true; }
    return false;
}
```
Example: Non-blocking atomic integer

```c
int atomic_int_add(atomic_int *p, int val) {
    int oldval;
    do {
        oldval = *p;
    } while ( !CAS(p, oldval, oldval+val);
}
```

• What happens if multiple threads execute this concurrently?
  • Does every thread make progress?
  • Does at least one thread make progress in bounded number of steps?
Why Non-blocking?

Two words: No locks!

- With locks, what happens if a thread is pre-empted while holding a lock?
- With locks, deadlock might be possible. Is it possible when there are no locks?

- Priority inversion
  - Assume threads have been assigned priorities, and we’d like to preferentially allocate cores to the highest priority runnable threads
  - Now suppose a low priority thread holds a lock needed by a high priority thread
    - A medium priority thread might even steal the core from the thread holding the lock!

- Alternative solution: priority inheritance
  - Raise the priority of a thread holding a lock to the maximum priority of any thread waiting for the lock
Why not non-blocking?
(Non-blocking FIFO implementation)

```
structure pointer
{ptr: pointer to node, count: unsigned integer}
structure node
{value: data type, next: pointer}
structure queue
{Head: pointer, Tail: pointer}
```

```
initialize(Q: pointer to queue)
node = new_node()
node->next.ptr = NULL
Q->Head = Q->Tail = node
# Allocate a free node
# Make it the only node in the linked list
# Both Head and Tail point to it
```

Pointers are stored with a generation number in one 8-byte quantity
(32-bit pointer + 32-bit generation number)

*From Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms by Michael & Scott.*
Non-blocking FIFO: enqueue()

enqueue(Q: pointer to queue, value: data type)
E1:   node = new node()
E2:   node->value = value
E3:   node->next.ptr = NULL
E4:   loop
E5:     tail = Q->Tail
E6:     next = tail.ptr->next
E7:     if tail == Q->Tail
E8:       if next.ptr == NULL
E9:         if CAS(&tail.ptr->next, next, <node, next.count+1>)
E10:           break
E11:       endif
E12:     else
E13:       CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)
E14:     endif
E15:   endloop
E16:   CAS(&Q->Tail, tail, <node, tail.count+1>)

# Allocate a new node from the free list
# Copy enqueued value into node
# Set next pointer of node to NULL
# Keep trying until Enqueue is done
# Read Tail.ptr and Tail.count together
# Read next ptr and count fields together
# Are tail and next consistent?
# Was Tail pointing to the last node?
# Try to link node at the end of the linked list
# Enqueue is done. Exit loop
# Tail was not pointing to the last node
# Try to swing Tail to the next node
# Enqueue is done. Try to swing Tail to the inserted node
Non-blocking FIFO: dequeue

dequeue(Q: pointer to queue, pvalue: pointer to data type): boolean

D1: loop
D2:   head = Q->Head
D3:   tail = Q->Tail
D4:   next = head->next
D5:   if head == Q->Head
D6:     if head.ptr == tail.ptr
D7:       if next.ptr == NULL
D8:         return FALSE
D9:     endif
D10:    CAS(&Q->Tail, tail, <next.ptr, tail.count+1>))
D11:    # Tail is falling behind. Try to advance it
D12:   else
D13:     # Read value before CAS, otherwise another dequeue might free the next node
D14:     *pvalue = next.ptr->value
D15:     if CAS(&Q->Head, head, <next.ptr, head.count+1>)
D16:       break
D17:     endif
D18:   endif
D19: endloop
D20: return TRUE
Performance Results

Figure 3: Net execution time for one million en-queue/dequeue pairs on a dedicated multiprocessor.

12 processor Silicon Graphics Challenge