Multi-Object Synchronization

Chapter 6 OSPP
Part I
The problem: want multiple locks

- Why: programs need to manipulate multiple shared objects

- What is the problem with one big lock?

- Linux still has a Big Kernel Lock (BKL) that will lock everything!
  - Rarely used
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
• Performance
• Semantics/correctness
• Deadlock
• Eliminating locks
Synchronization Performance

- A program with lots of concurrent threads can still have poor performance on a multiprocessor:
  - 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 the cache within each core
  - False sharing: communication between cores even for data that is not shared
Web Server Lock

• In a memory cache that is accessed 5% of the time with a single lock

• On a multiprocessor suppose getting the lock is 4 times slower (get lock from another cache)

• Need careful design of shared locking
Reducing Lock Contention

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

• Per-processor data structures: partition by core
  – Partition object so that most/all accesses are made by one processor: reduces false sharing, but cross cache access
  – Example: per-processor heap

• Ownership/Staged architecture: partition by op
  – Only one thread at a time accesses shared data
  – Example: pipeline of threads
Thread Pipelines

- Benefits
  - Modularity
  - Cache locality
- Problems:
Lock Contention

• Still a major issue on a multiprocessor
• Busy locks can hamper performance
  – Everyone wants to access “Bieber” object
• MCS locks (if locks are mostly busy)
• RCU locks (if locks are mostly busy, and data is mostly read-only)
• We’ve seen opts for when lock was mostly FREE (fastpath)
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”
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?
Test (and Test) and Set Performance

![Graph showing performance comparison of lock mechanisms](graph.png)
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 (give FREE a chance)

• 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, “fail”, keep looping
- 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 Lock Implementation

MCSLock::acquire() {
    Queue *oldTail = tail;
    myTCB->next = NULL;
    myTCB->needToWait = TRUE;
    // become tail only if CAS not called
    while (!compareAndSwap(&tail, oldTail, &myTCB)) {
        oldTail = tail;
    }
    if (oldTail != NULL) {
        oldTail->next = myTCB;
        memory_barrier();
        // key: spinning on sep. var!
        while (myTCB->needToWait)
            ;
    }
}

MCSLock::release() {
    if (!compareAndSwap(&tail, myTCB, NULL)) {
        // spin until a acquirer shows up!
        while (myTCB->next == NULL)
            ;
        myTCB->next->needToWait=FALSE;
    }
}

bool cas (int *p, int old, new) {
    if (*p != old) {
        return false;
    }
    *p = new;
    return true;
}
MCS In Operation

a) TAIL → NIL

b) A: next needToWait
    ┌───────┬───────┐
    │ NIL   │ FALSE │
    └───────┴───────┘

B: TAIL

c) A: B FALSE
    ┌───┐
    │   │
    └───┘

B: NIL TRUE
TAIL

d) A: B FALSE
    ┌───┐
    │   │
    └───┘

B: C TRUE
    ┌───┐
    │   │
    └───┘

C: NIL TRUE
TAIL

e) B: C FALSE
    ┌───┐
    │   │
    └───┘

C: NIL TRUE
    ┌───┐
    │   │
    └───┘

f) NIL FALSE
    ┌───┐
    │   │
    └───┘

TAIL
Read-Copy-Update

• **Goal:** very fast reads to shared data
  – Reads proceed without first acquiring a lock
  – `RL.acquire()`
  – 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
RCU

- Really delay writers ... solution ->
- RCU implementation integrated with thread scheduler (unlike RW lock)
  - Quiescent fig. 6.12 (students can look at)
Multi-object Atomicity

- A->subtract(100);
- B->add(100);
- Want both to happen or not at all
- Solutions:
Multi-object Atomicity

- Acquire-all, Release-all

- 2PC: Acquire as needed; Release only after all locks are acquired

- Take 5105!
Deadlock Definition

- Resource: any (passive) entity 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 (recursive waiting)

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock1.acquire();</td>
<td>lock2.acquire();</td>
</tr>
<tr>
<td>lock2.acquire();</td>
<td>lock1.acquire();</td>
</tr>
<tr>
<td>lock2.release();</td>
<td>lock1.release();</td>
</tr>
<tr>
<td>lock1.release();</td>
<td>lock2.release();</td>
</tr>
</tbody>
</table>
Nested waiting: 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();
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.
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
  – Telephone circuit setup: p. 303
  – Internet router: p. 303 (conservative: drop pkts)

• 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
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
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
  – Rarely used BUT provides great insight into states
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
Optimistic Approach

• Optimize case with limited contention

• Proceed without the resource
  – Requires robust exception handling code
  – Amazon example p. 300

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

Diagram showing the relationships between threads and resources, indicating a deadlock situation.
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
Next Week

- Scheduling!
- OSPP Chapter 7

- Have a great weekend!