Locking

Don Porter

Portions courtesy Emmett Witchel
Too Much Milk: Lessons

• Software solution (Peterson’s algorithm) works, but it is unsatisfactory
  – Solution is complicated; proving correctness is tricky even for the simple example
  – While thread is waiting, it is consuming CPU time
  – Asymmetric solution exists for 2 processes.

• How can we do better?
  – Use hardware features to eliminate busy waiting
  – Define higher-level programming abstractions to simplify concurrent programming
Concurrent Quiz

If two threads execute this program concurrently, how many different final values of X are there?

Initially, X == 0.

Thread 1
```c
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}
```

Thread 2
```c
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}
```

Answer:
A. 0  
B. 1  
C. 2  
D. More than 2
Schedules and Interleavings

• Model of concurrent execution
• Interleave statements from each thread into a single thread
• If **any** interleaving yields incorrect results, some synchronization is needed

If X==0 initially, X == 1 at the end. WRONG result!
Mutual exclusion ensures only safe interleavings

- *When is mutual exclusion too safe?*
Introducing Locks

- **Locks** – implement mutual exclusion
  - Two methods
    - Lock::Acquire() – wait until lock is free, then grab it
    - Lock::Release() – release the lock, waking up a waiter, if any

- With locks, too much milk problem is very easy!
  - Check and update happen as one unit (exclusive access)

```c
Lock.Acquire();
if (noMilk) {
    buy milk;
}
Lock.Release();
```

How can we implement locks?
How do locks work?

• Two key ingredients:
  – A hardware-provided atomic instruction
    • Determines who wins under contention
  – A waiting strategy for the loser(s)
Atomic instructions

• A “normal” instruction can span many CPU cycles
  – Example: ‘a = b + c’ requires 2 loads and a store
  – These loads and stores can interleave with other CPUs’ memory accesses

• An atomic instruction guarantees that the entire operation is not interleaved with any other CPU
  – x86: Certain instructions can have a ‘lock’ prefix
  – Intuition: This CPU ‘locks’ all of memory
  – Expensive! Not ever used automatically by a compiler; must be explicitly used by the programmer
Atomic instruction examples

• Atomic increment/decrement (x++ or x--)
  – Used for reference counting
  – Some variants also return the value x was set to by this instruction (useful if another CPU immediately changes the value)

• Compare and swap
  – if (x == y) x = z;
  – Used for many lock-free data structures
Atomic instructions + locks

• Most lock implementations have some sort of counter

• Say initialized to 1

• To acquire the lock, use an atomic decrement
  – If you set the value to 0, you win! Go ahead
  – If you get < 0, you lose. Wait 😞
  – Atomic decrement ensures that only one CPU will decrement the value to zero

• To release, set the value back to 1
Waiting strategies

• Spinning: Just poll the atomic counter in a busy loop; when it becomes 1, try the atomic decrement again

• Blocking: Create a kernel wait queue and go to sleep, yielding the CPU to more useful work
  – Winner is responsible to wake up losers (in addition to setting lock variable to 1)
  – Create a kernel wait queue – the same thing used to wait on I/O
    • Reminder: Moving to a wait queue takes you out of the scheduler’s run queue
Which strategy to use?

• Main consideration: Expected time waiting for the lock vs. time to do 2 context switches
  – If the lock will be held a long time (like while waiting for disk I/O), blocking makes sense
  – If the lock is only held momentarily, spinning makes sense
• Other, subtle considerations we will discuss later
Reminder: Correctness Conditions

- **Safety**
  - Only one thread in the critical region

- **Liveness**
  - Some thread that enters the entry section eventually enters the critical region
  - Even if other thread takes forever in non-critical region

- **Bounded waiting**
  - A thread that enters the entry section enters the critical section within some bounded number of operations.

- **Failure atomicity**
  - It is OK for a thread to die in the critical region
  - Many techniques do not provide failure atomicity
Example: Linux spinlock (simplified)

1: lock; decb slp->slock // Locked decrement of lock var
   jns 3f // Jump if not set (result is zero) to 3
2: pause // Low power instruction, wakes on
    // coherence event
   cmpb $0,slp->slock // Read the lock value, compare to zero
   jle 2b // If less than or equal (to zero), goto 2
   jmp 1b // Else jump to 1 and try again
3: // We win the lock
Rough C equivalent

while (0 != atomic_dec(&lock->counter)) {
    do {
        // Pause the CPU until some coherence
        // traffic (a prerequisite for the counter
        // changing) saving power

    } while (lock->counter <= 0);
}
Why 2 loops?

- Functionally, the outer loop is sufficient
- Problem: Attempts to write this variable invalidate it in all other caches
  - If many CPUs are waiting on this lock, the cache line will bounce between CPUs that are polling its value
    - This is VERY expensive and slows down EVERYTHING on the system
  - The inner loop read-shares this cache line, allowing all polling in parallel
- This pattern called a Test&Test&Set lock (vs. Test&Set)
Test & Set Lock

// Has lock

```
while (!atomic_dec(&lock->counter))
```

Cache Line “ping-pongs” back and forth
Test & Test & Set Lock

// Has lock

Unlock by writing 1

while (lock->counter <= 0))

CPU 0

Cache

0x1000

Memory Bus

CPU 1

read

CPU 2

read

0x1000

RAM

Line shared in read mode until unlocked
Why 2 loops?

• Functionally, the outer loop is sufficient

• Problem: Attempts to write this variable invalidate it in all other caches
  – If many CPUs are waiting on this lock, the cache line will bounce between CPUs that are polling its value
    • This is VERY expensive and slows down EVERYTHING on the system
  – The inner loop read-shares this cache line, allowing all polling in parallel

• This pattern called a Test&Test&Set lock (vs. Test&Set)
Implementing Blocking Locks

```cpp
Lock::Acquire() {
    while (test&set(lock) == 1) // spin
        ;
}

Lock::Release() {
    *lock := 0;
}

With busy-waiting

Lock::Acquire() {
    while (test&set(q_lock) == 1) {
        Put TCB on wait queue for lock;
        Lock::Switch(); // dispatch thread
    }
}

Without busy-waiting, use a queue

Lock::Release() {
    *q_lock = 0;
    if (wait queue is not empty) {
        Move 1 (or all?) waiting threads to ready queue;
    }
}
```

Must only one thread be awaked? Is this code fair?
Best Practices for Lock Programming

• When you enter a critical region, check what may have changed while you were spinning
  – Did Jill get milk while I was waiting on the lock?

• Always unlock any locks you acquire
Implementing Locks: Summary

• Locks are higher-level programming abstraction
  – Mutual exclusion can be implemented using locks

• Lock implementations have 2 key ingredients:
  – Hardware instruction: atomic read-modify-write
  – Blocking mechanism
    • Busy waiting, or
      – Cheap Busy waiting important
    • Block on a scheduler queue in the OS

• Locks are good for mutual exclusion but weak for coordination, e.g., producer/consumer patterns.
Why locking is also hard (Preview)

- **Coarse-grain locks**
  - Simple to develop
  - Easy to avoid deadlock
  - Few data races
  - Limited concurrency

- **Fine-grain locks**
  - Greater concurrency
  - Greater code complexity
  - Potential deadlocks
    - Not composable
  - Potential data races
    - Which lock to lock?

```c
// WITH FINE-GRAIN LOCKS
void move(T s, T d, Obj key)
{
    LOCK(s);
    LOCK(d);
    tmp = s.remove(key);
    d.insert(key, tmp);
    UNLOCK(d);
    UNLOCK(s);
}
```

Thread 0
```
move(a, b, key1);
move(b, a, key2);
```

Thread 1
```
DEADLOCK!
```