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

Concurrency Quiz

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

Initially, X == 0.

Thread 1

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

Thread 2

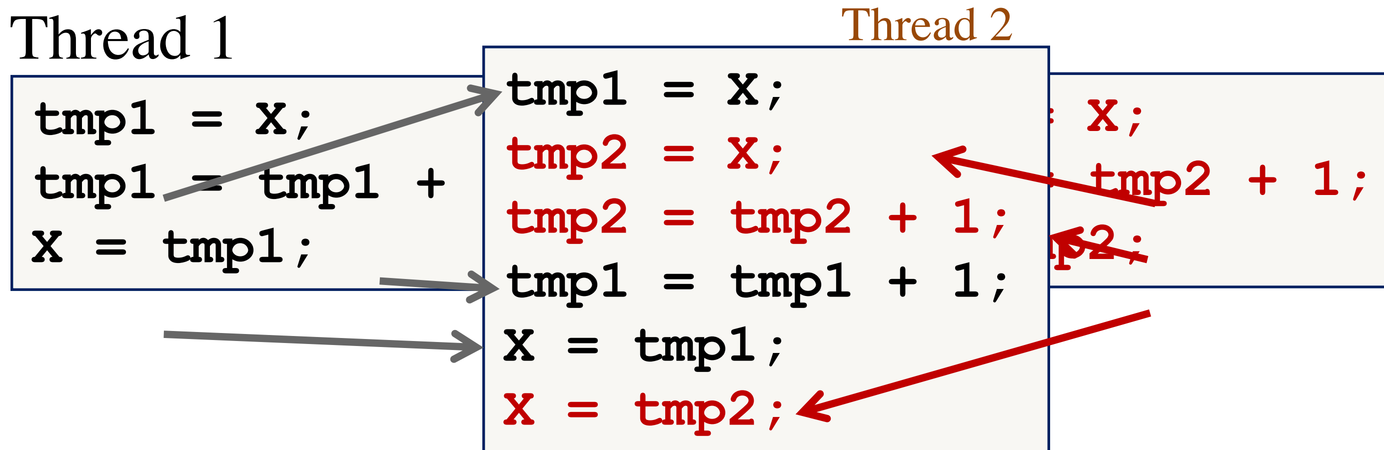
```
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}
```

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!

Locks fix this with Mutual Exclusion

```
void increment() {
    pthread_mutex_lock(&lock);
    int temp = X;
    temp = temp + 1;
    X = temp;
    pthread_mutex_unlock(&lock);
}
```

- Key abstraction: mutual exclusion while lock is held
- Goal: ”Protect” unsafe code from dangerous interleavings
  - At some loss of concurrency

Introducing Locks

- Locks – implement mutual exclusion
  - Two methods
    - pthread_mutex_lock(lock) – wait until lock is free, then grab it
    - pthread_mutex_unlock(lock) – 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)

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

```
Lock.Acquire();
x++;
Lock.Release();
```

How can we implement locks?

Performance: Between rock and hard place

- We need threads for concurrent performance
- We can't safely execute all code concurrently
  - Locks ensure that "delicate" code does not interleave with other code that could interleave unsafely
- It is safe to execute everything in one big lock
  - But *worse* performance than a single thread
  - No concurrency + overheads
- Goal: get just enough mutual exclusion for safety, but no more than strictly necessary

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” line of code (or CISC instruction) can span multiple memory operations
  - 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--)
  - ```
    int atomic_inc(int *var) {
      int rv = *var;
      *var++;
      return rv;
    }
    ```
  - Used for reference counting,
  - Returns old value that you specifically set
  - If *var is 0, and 3 threads do an atomic_inc, one will get 1, one 2, and one 3
- Atomic Test and Set:
  - `old = ts(&var)`
  - bool ts(int *) { bool ret = *int; *int = 1; return ret == 0;}
  - Sets a value to 1 atomically; returns true if you were the thread that transitioned from 0 to 1
- Compare and swap
  - Common Syntax: **`cas(&var, old, new)`**
  - {int rv = *var; if (*var == old) *var = new; return rv;}
  - 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
  - Recall: atomic_dec returns the value your thread set
  - 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

Cache Line “ping-pongs” back and forth

Test & Test & Set Lock

// Has lock

while (lock->counter <= 0))

CPU 0

Unlock by writing 1

CPU 1

CPU 2

read

read

Cache

Cache

0x1000

Memory Bus

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

```
pthread_mutex_lock() {
while (ts(lock) == 1)
    ; // spin
}
```

With busy-waiting

```
pthread_mutex_unlock() {
    *lock := 0;
}
```

```
pthread_mutex_lock() {
while (ts(q_lock) == 1) {
  Put TCB on wait queue for lock;
}
```

Without busy-waiting, use a queue

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

Must only one thread be awakened? Is this code fair?

Reader/writer locks

- Simple optimization: If I am just reading, we can let other readers access the data at the same time
  - Just no writers
- Writers require mutual exclusion

History: Semaphores

- Semaphores implement k-way exclusion
  - Where k >= 1
- History: Semaphores were the first lock
- Today: A binary (k=1) semaphore *is* a lock
  - Often a blocking lock
- Non-binary semaphores are rarely useful
  - k identical resources typically need k mutual exclusion locks
    - Not k threads interleaving with each other on any of k resources
- Worth knowing the term for interview “trivia”

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?

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

Thread 0	Thread 1
`move(a, b, key1);`	
	`move(b, a, key2);`

DEADLOCK!