CS153: Lock Implementations

Chengyu Song

Slides modified from Harsha Madhyvasta, Nael Abu-Ghazaleh, and Zhiyun Qian
Administrivia

- **Lab**
  - Lab1 is due this Sunday
  - Demo sessions next week
- **Homework**
  - Homework1 is due this Friday
Recap: synchronization

- Race condition can happen between threads with shared data
  - Concurrent
  - At least one write
- Synchronization is necessary to provide
  - Safety, liveness, performance
- Mechanisms
  - Building blocks: atomic read and write
  - Alternation, lock
Implementing locks: first attempt

• How do we implement locks?

```c
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (lock->held);
    lock->held = 1;
}
void release (lock) {
    lock->held = 0;
}
```

• This is called a **spinlock** because a thread spins waiting for the lock to be released.
Implementing locks: problem

- Does the first attempt work?
- No. Two independent threads may both notice that a lock has been released and thereby acquire it.

```c
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (lock->held);
    lock->held = 1;
}
void release (lock) {
    lock->held = 0;
}
```

A context switch can occur here, causing a race condition.
Implementing locks: challenges

- The problem is that the implementation of locks has critical sections, too
- How do we stop the recursion?
- The implementation of `acquire()`/`release()` must be atomic
- Recap: what are atomic operations?
  - An atomic operation is one which executes as though it could not be interrupted
  - Code that executes "all or nothing"
Implementing locks: architecture support

• How do we make `acquire()` atomic?

• Need help from hardware
  
  • More atomic instructions (e.g., test-and-set)
  
  • Disable/enable interrupts (prevents context switches)
Atomic instructions: test-and-set

• The semantics of test-and-set are:

```cpp
bool test_and_set (bool *flag) {
    bool old = *flag;   // Record the old value
    *flag = True;      // Set the value to indicate available
    return old;        // Return the old value
}
```

• Hardware executes it atomically!
Implementing locks: test-and-set

• Here is our lock implementation with test-and-set:

```c
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (test-and-set(&lock->held));
}
void release (lock) {
    lock->held = 0;
}
```

• When will the while return? What is the value of held?
Problems with spinlocks

• The problem with spinlocks is that they are wasteful
• If a thread is spinning on a lock, then the scheduler thinks that this thread needs CPU and puts it on the ready queue
• If N threads are contending for the lock, the thread which holds the lock gets only 1/N'th of the CPU
Disabling interrupts

- Another implementation of acquire/release is to disable interrupts

```c
struct lock {
}
void acquire (lock) {
    disable interrupts;
}
void release (lock) {
    enable interrupts;
}
```

- Note that there is no state associated with the lock
- Can two threads disable interrupts simultaneously?
On disabling interrupts

• Disabling interrupts blocks notification of external events that could trigger a context switch (e.g., timer)

• In a "real" system, this is only available to the kernel. Why?

• Disabling interrupts is insufficient on a multiprocessor
  • Back to atomic instructions

• Like spinlocks, only want to disable interrupts to implement higher-level synchronization primitives
  • Don't want interrupts disabled between `acquire()` and `release()`
Where we are so far

• Goal: Use **mutual exclusion** to protect **critical sections** of code that access **shared resources**

• Method: use locks (spinlocks or disable interrupts)
Where we are so far (cont.)

- Problem: Critical sections can be long

**Spinlocks:**
- Threads waiting to acquire lock spin in test-and-set loop
- Wastes CPU cycles
- Longer the CS, the longer the spin
- Greater the chance for lock holder to be interrupted

**Disabling Interrupts:**
- Should not disable interrupts for long periods of time
- Can miss or delay important events (e.g., timer, I/O)
Higher-level synchronization

- Spinlocks and disabling interrupts are useful only for very short and simple critical sections
  - These primitives are "primitive" – don’t do anything besides mutual exclusion
- Need higher-level synchronization primitives that:
  - Block waiters
  - Leave interrupts enabled within the critical section
- New building block: our atomic locks
Implementing locks: second attempt

- Block waiters, interrupts enabled in critical sections

```
struct lock {
    int held = 0;
    queue Q;
}
void acquire (lock) {
    Disable interrupts;
    if (lock->held) {
        put current thread on lock Q;
        block current thread;
    }
    lock->held = 1;
    Enable interrupts;
}
```

```c
void release (lock) {
    Disable interrupts;
    if (Q)
        remove and unblock a waiting thread;
    else
        lock->held = 0;
    Enable interrupts;
}
```

- acquire(lock)  
  ...  
  Critical section  
  ...  
  release(lock)
Implementing locks: second attempt (cont.)

- Again, disabling interrupts does not work on multiprocessor systems

```c
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (test-and-set(&lock->held)) {
        thread_yield();
    }
}
void release (lock) {
    lock->held = 0;
}
```
Problems with wait queue

- Liveness requirement: **bounded wait time**
  - How to implement the queue? FIFO?
- Performance requirement: the overhead of entering and exiting the critical section is small
  - Maintaining queue is not free
    - CPU, memory
  - Context switch is not free
    - Maybe it's better just spinning for a while
Implementing locks: advanced topics

- Test-and-set with back-off (exponential works best)
- Ticket lock
- Array-based queue
- List-based queue
  - MCS, JM Mellor-Crummey and ML Scott (2006 Edsger Dijkstra Prize)
Summary

• Architecture support
  • test-and-set, disabling interrupts
• Spinlock
• Queued lock
For next class ...

- Semaphores and monitors
- Textbook
  - Module 30, 31