

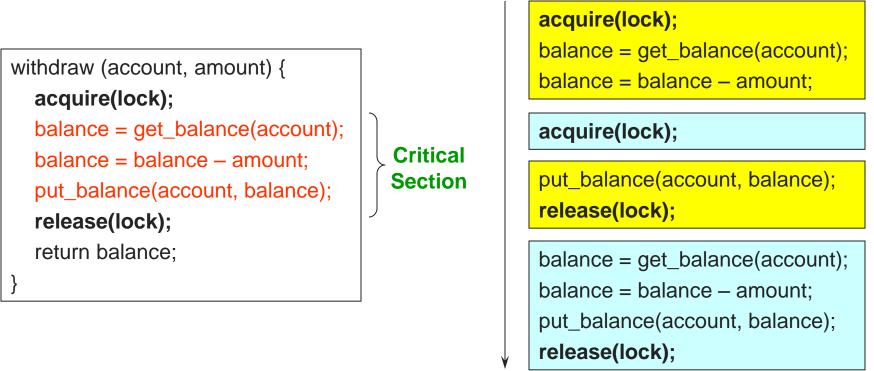
Winter 2016

Lecture 8: Synchronization



- A lock is an object in memory providing two operations
 - acquire(): before entering the critical section
 - release(): after leaving a critical section
- Threads pair calls to acquire() and release()
 - Between acquire()/release(), the thread holds the lock
 - acquire() does not return until any previous holder releases
 - What can happen if the calls are not paired?

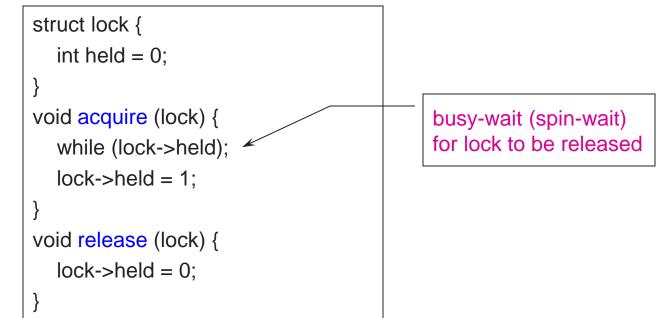
Using Locks



- Why is the "return" outside the critical section? Is this ok?
- What happens when a third thread calls acquire?

Implementing Locks (1)

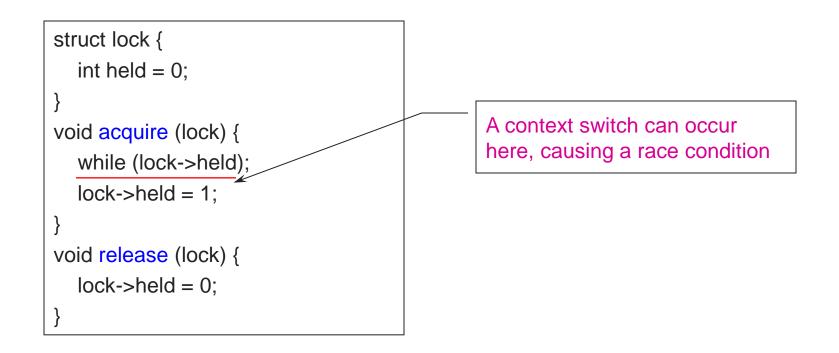
• How do we implement locks? Here is one attempt:



- This is called a spinlock because a thread spins waiting for the lock to be released
- Does this work?

Implementing Locks (2)

• No. Two independent threads may both notice that a lock has been released and thereby acquire it.

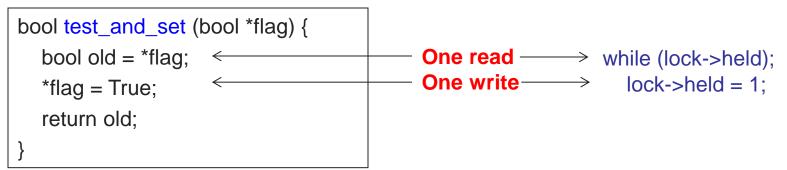


Implementing Locks (3)

- 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
 - An atomic operation is one which executes as though it could not be interrupted
 - Code that executes "all or nothing"
- How do we make them atomic?
- Need help from hardware
 - 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:
 - Record the old value
 - Set the value to indicate available
 - Return the old value
- Hardware executes it atomically!



- When executing test-and-set on "flag"
 - What is value of flag afterwards if it was initially False? True?
 - What is the return result if flag was initially False? True?

Using Test-And-Set (Spinlocks)

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

```
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:

```
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
 - This is what PintOS does
 - Don't want interrupts disabled between acquire and release

Next time...

- Semaphores, monitors and other synchronization primitives
 - ♦ Read Chapter 5.4 5.7