CS162
Operating Systems and Systems Programming
Lecture 4 (extra)

Synchronization, Atomic operations, Locks

February 3, 2014
Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162
Definitions

• Atomic Operation: an operation that always runs to completion or not at all
  – It is *indivisible*: it cannot be stopped in the middle and state cannot be modified by someone else in the middle
Definitions

- **Atomic Operation**: an operation that always runs to completion or not at all
  - It is *indivisible*: it cannot be stopped in the middle and state cannot be modified by someone else in the middle

- **Synchronization**: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic

- **Critical Section**: piece of code that only one thread can execute at once

- **Mutual Exclusion**: ensuring that only one thread executes critical section
  - Critical section and mutual exclusion are two ways of describing the same thing
Definitions

• **Synchronization**: using atomic operations to ensure cooperation between threads

• **Critical Section**: piece of code that only one thread can execute at once

• **Mutual Exclusion**: ensuring that only one thread executes critical section

• **Lock**: prevents someone from doing something
  – Lock before entering critical section and before accessing shared data
  – Unlock when leaving, after accessing shared data
  – Wait if locked

  » **Important idea**: all synchronization involves waiting
5min Break
Too Much Milk Solution #3

- Here is a possible two-note solution:

  **Thread A**
  
  leave note A;
  
  while (note B) {
    do nothing;
  }
  
  if (noMilk) {
    buy milk;
  }
  
  remove note A;

  **Thread B**
  
  leave note B;
  
  if (noNote A) {
    if (noMilk) {
      buy milk;
    }
  }
  
  remove note B;
Naïve use of Interrupt Enable/Disable

• How can we build multi-instruction atomic operations?
  – Recall: dispatcher gets control in two ways.
    » Internal: Thread does something to relinquish the CPU
    » External: Interrupts cause dispatcher to take CPU
  – On a uniprocessor, can avoid context-switching by:
    » Avoiding internal events (although virtual memory tricky)
    » Preventing external events by disabling interrupts

• Consequently, naïve Implementation of locks:

  LockAcquire { disable Ints; }  
  LockRelease { enable Ints; }
Better Implementation of Locks by Disabling Interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

```c
int value = FREE;

Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
        // Enable interrupts?
    } else {
        value = BUSY;
    }
    enable interrupts;
}

Release() {
    disable interrupts;
    if (anyone on wait queue) {
        take thread off wait queue
        Put at front of ready queue
    } else {
        value = FREE;
    }
    enable interrupts;
}
```