Synchronization II: EventBarrier, Monitor, and a Semaphore

COMPSCI210 Recitation

4th Mar 2013

Vamsi Thummala

Check point: Mission in progress

- Master synchronization techniques
- Develop best practices for writing synchronization code
- Write solid concurrent code

So far: “Too much milk” example

- Need for mutual exclusion (mutex/lock)
  - Ensures only one thread thread access critical section

“Too much milk”: Need for Mutex

```
lock ()
if (noNote && noMilk){
  leave note “at store”
  unlock ()
  buy milk
  lock ()
  remove note
  unlock ()
} else {
  unlock ()
}
```

Not holding lock

```
lock ()
if (noNote && noMilk){
  leave note “at store”
  unlock ()
  buy milk
  lock ()
  remove note
  unlock ()
} else {
  unlock ()
}
```

Only hold lock while handling shared resource.

“Too much milk” first extension

- Constraint:
  - Landon and Melisa take strict turns to buy the milk
  - Similar to Ping-Pong example
- Mutexes are not sufficient to impose constraints

Raise the level of abstraction

1. Mutual exclusion
   - Ensure one thread access the critical section
   - Use locks
2. Ordering constraints
   - Describe “before-after” relationships
   - One thread waits for another
   - Use monitors: a lock + a conditional variable

Other abstractions support by a language/OS

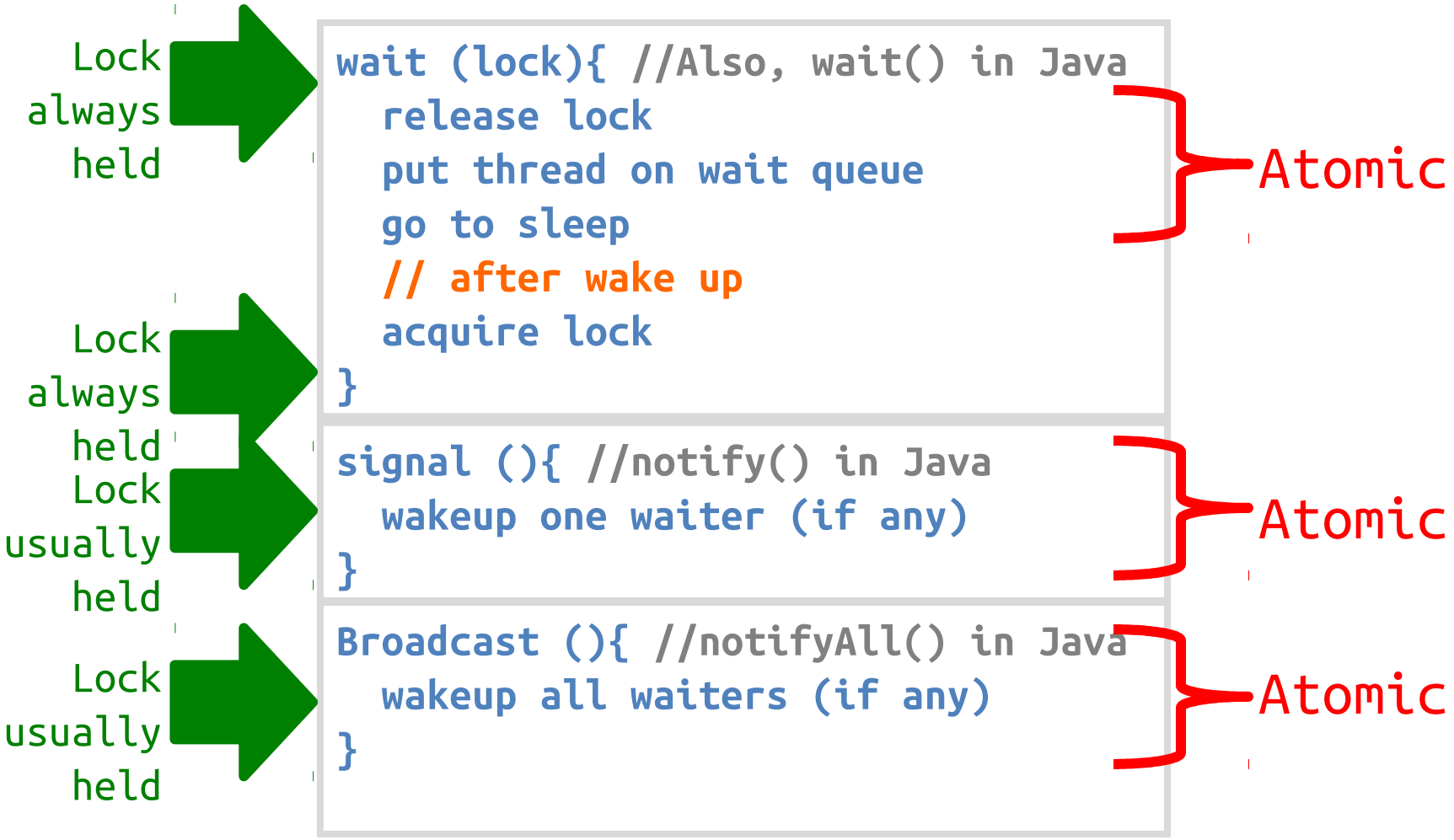
- Software transaction memory
  - Transaction support within shared memory
  - Analogous to db transaction support
  - Active area of research
  - “volatile” in Java
    - But please do not use for the labs. We want you to learn and handle synchronization explicitly
- Actors aka asynchronous communication via message passing
  - Erlang

Monitor: A Lock + CV

- Conditional variable: Maintains state
  - Queue of waiting threads on a lock
- Internal atomic actions

```
// begin atomic
release lock
put thread on wait queue
go to sleep
// end atomic
```

Condition variable operations



“Too much milk”: strict alternate constraint

```
synchronized buyMilk {
  if(isMilkPurchased) {
    notify()
    wait()
    buy milk
 }
```

```
synchronized buyMilk {
  if(isMilkPurchased) {
    notify()
    wait()
    buy milk
 }
```

What if they are multiple threads in two pools?

“Too much milk”: second extension – multiple threads in two pools

and his friends

```
synchronized buyMilk {
  while(hasMilkPurchased) {
    notify(MelisaPool)
    wait(MelisaPool)
    buy milk
 }
```

and her friends

```
synchronized buyMilk {
  while(hasMilkPurchased) {
    notify(LandonPool)
    wait(LandonPool)
    buy milk
 }
```

Some coding practices

- (Almost) never sleep()
- (Always) loop always before you leap!

```
while(CV is true) {
     wait()
}
```

- Avoid using synchronized(this)
  - Lock is held and released in between a method
  - Code hard to read/follow
  - Instead divide the code into modules and synchronize on methods

Semaphore

- Alternative to monitor
- Two operations
  - P() // Down
  - V() // UP
- No separation of locking and coordination/scheduling unlike monitors
  - Everything expressed using P(), V() including mutex (binary semaphore)
  - Best fit when synchronization involves some form of counting (resources)
  - CV can represent any condition: need not be counting
- CV and semaphore type can implement one another

“Too much milk”: first extension taking turns using semaphores

```
boolean isMilkPurchased = false
buyMilk {
   isMilkPurchased.V()
   isMilkPurchased.P()
}
```

```
boolean isMilkPurchased = false
buyMilk {
   isMilkPurchased.P()
   isMilkPurchased.V()
}
```

Asymmetric code

“Too much milk”: first extension taking turns using semaphores

```
boolean isMilkPurchased = true
buyMilk {
   isMilkPurchased.P()
   isMilkPurchased.V()
}
```

```
boolean isMilkPurchased = false
buyMilk {
   isMilkPurchased.P()
   isMilkPurchased.V()
}
```

Symmetric code

What is the right synchronization primitive?

- Should I use a CV or a Semaphore or a EventBarrier?
- Some problems are better expressed using semaphores, but in general, CVs are much better abstraction
- EventBarrier is useful when multiple threads has to synchronize in phases
  - Will revisit

Semaphores using CVs

```
class Semaphore {
    private unsigned int _count;

    public Semaphore(unsigned int count) { _count = count; }

    public synchronized void P() {

    }

    public synchronized void V() {

    }
}
```

Semaphores using CVs

```
class Semaphore {
    private unsigned int _count;

    public Semaphore(unsigned int count) { _count = count; }

    public synchronized void P() { //Down
        while (_count <= 0)
            try { wait(); } catch (Exception e) {}
        _count--;
    }

    public synchronized void V() { //Up
        _count++;
        notify();
    }
}
```

Read/Write Lock

- Improve standard lock for multiple readers:
- Read
  - Can assign locks to multiple readers, but only when:
    - no threads are requesting for write access
- Write
  - Exclusive access:
    - no other threads are reading or writing

Implementing Read/Write Lock

```
public class ReadWriteLock{

  private int _numReaders, _numWriters, _numWriteRequests = 0;

  public synchronized void acquireRead() {
  }
  public synchronized void releaseRead() {
  }
  public synchronized void acquireWrite() {
  }
  public synchronized void releaseWrite() {
  }
}
```

Implementing Read/Write Lock

```
public class ReadWriteLock{
  private int _numReaders, _numWriters, _numWriteRequests = 0;
  public synchronized void acquireRead() {
       while(_numWriters > 0 || _numWriteRequests > 0){
           wait();
       }
       readers++;
  }
  public synchronized void releaseRead() {
  }
  public synchronized void acquireWrite() {
  }
  public synchronized void releaseWrite() {
  }
}
```

Implementing Read/Write Lock

```
public class ReadWriteLock{
  private int _numReaders, _numWriters, _numWriteRequests = 0;
  public synchronized void acquireRead() {
       while(_numWriters > 0 || _numWriteRequests > 0){
           wait();
       }
       _numReaders++;
  }
  public synchronized void releaseRead() {
        _numReaders--;
        notifyAll();
  }
  public synchronized void acquireWrite() {
  }
  public synchronized void releaseWrite() {
          _numWriters--;
          notifyAll();
  }
}
```

Implementing Read/Write Lock

```
public class ReadWriteLock{
  private int _numReaders, _numWriters, _numWriteRequests = 0;
  public synchronized void acquireRead() {
       while(_numWriters > 0 || _numWriteRequests > 0){ wait();}
       _numReaders++;
  }
  public synchronized void releaseRead() {
        _numReaders--; notifyAll();
  }
  public synchronized void acquireWrite() {
        _numWriteRequests++;
        while(_numReaders > 0 || _numWriters > 0){
              wait();
        }
        _numWriteRequests--;
        _numWriters++;
  }
  public synchronized void releaseWrite() {
          _numWriters--; notifyAll();
  }
}
```

Read/Write Lock issues

- Starvation
- Lock not reentrant
  - A thread holding a lock and requesting for the same lock again will block since the lock is held
- Deadlock can occur
  - Due to lock not reentrant (1R, 2W, 1R)
- How to improve for the above?

EventBarrier: Another analogy (on piazza)

- Alice, Bob, and Charlie are three secret agents who are good in their respective domains: Math, Physics, and CS. They are given a jigsaw puzzle to solve, which demands the knowledge from all the three domains. However, due to the nature of operation involved there are certain constraints: they cannot talk to each other directly, and they cannot meet for more than 10 minutes at a time. There is an agent coordinator, who arranges rendezvous, whenever all the agents agrees to meet. They worked out a plan: all agents work independently on a certain task and notifies the coordinator when they are done with that task and want to meet (through arrive() call), and wait perpetually until the coordinator responds with details (through raise() call). Once all three agents notifies the coordinator, the coordinator send the details of rendezvous, and they all meet and synchronize on the tasks, and dissemble. With the collective new found knowledge, they start working independently again the next day, and this process continues until the puzzle is solved.

EventBarrier: Use case

- A complex computation can be divided and distributed among multiple tasks. Some parts of this computation can be I/O bound, the other parts are CPU intensive, and other are GPU operations that rely on specialized graphics chip. These partial results must be collected from various tasks for the final calculation. The result determines what other partial computations each task is to perform next.

Testing EventBarrier

- Say you have n consumers with some local variable set to "phase1". On complete(), each consumer increments their count. For example, the second iteration their local variable will be set to "phase2". But the barrier does not return until all the consumers arrived. So if you have print() statement after the barrier, you should see all the consumers printing "phase2". If some consumer prints "phase1" that means that complete() did not happen but still passed through the barrier. Hence, indicative of a bug.

“Too much milk”: thrid extension

- Practice problem

4 times in a week

```
synchronized buyMilk {


}
```

3 times in a week

```
synchronized buyMilk {


}
```

- More: http://www.cs.duke.edu/courses/compsci210/spring13/slides/recitation/sync-practice.pdf