CS601: Principles of Software Development

Multithreading. Synchronization: volatile, synchronized blocks.

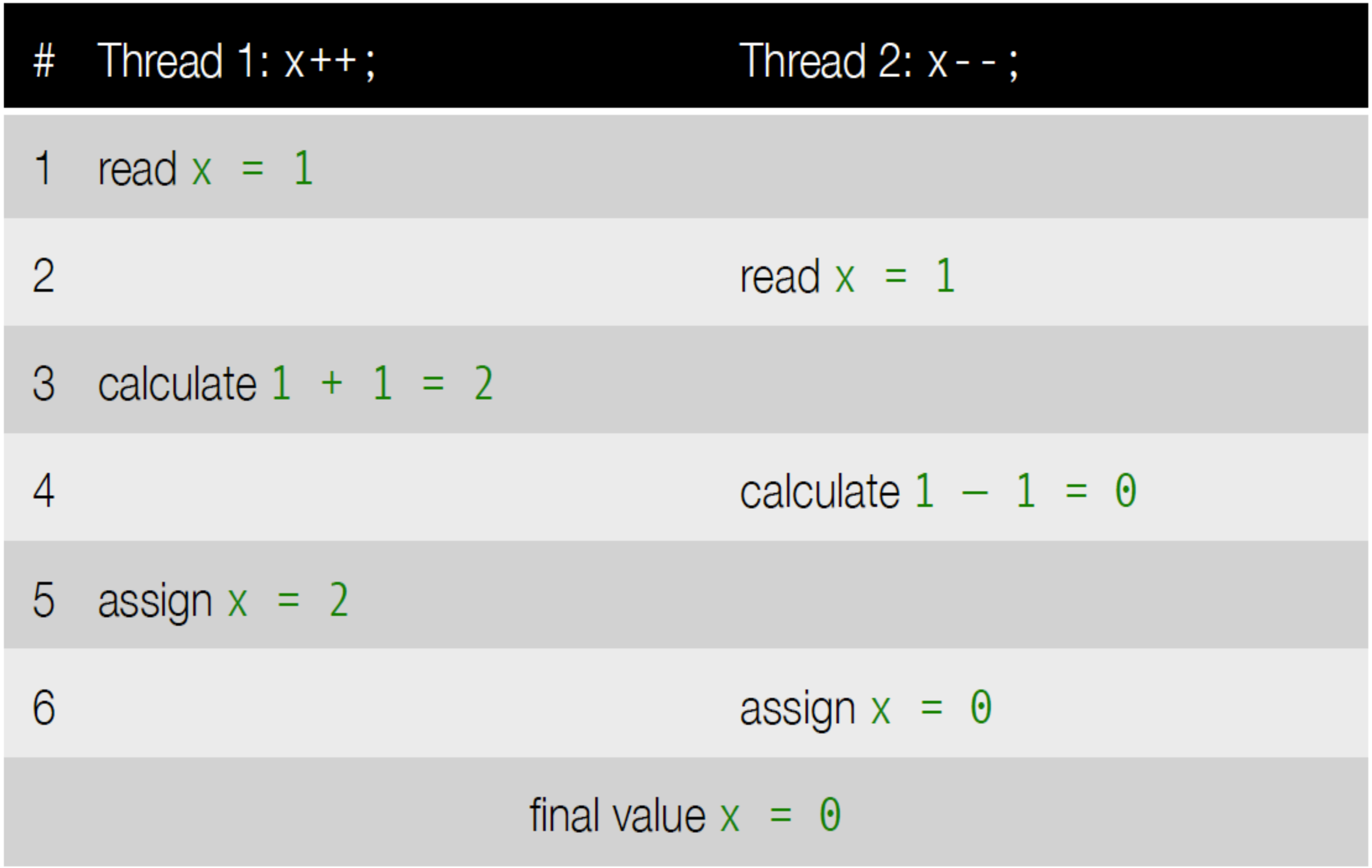
Olga A. Karpenko

Parts of this presentations are based on the materials of Prof. Engle.

Announcements

- Lab 2 due tonight
  - Reminder regarding github history!

Motivation



#	Thread 1: x++;	Thread 2: x--;
1	read x = 1	
2		read x = 1
3	calculate 1 + 1 = 2	
4		calculate 1 – 1 = 0
5	assign x = 2	
6		assign x = 0
	final value x = 0	

The image is courtesy of Prof. Engle.

Problems

- Atomicity
  - Operations x++ and x-- are not *atomic*
- Visibility
  - Shared data modified between read & use

Synchronization in Java

- Volatile variables
- Synchronized code blocks and methods
- Custom lock objects

Volatile

- All "writes" are written directly to the main memory
- All "reads" read from the main memory, not from cache
- Threads always read the latest value
- Guarantees visibility but not atomicity

From: http://tutorials.jenkov.com/java-concurrency/volatile.html

Volatile

```
public class SharedObject {

    public volatile int counter = 0;

}
```

From: http://tutorials.jenkov.com/java-concurrency/volatile.html

Use Pattern #1: Status Flag

- Read does not depend on other variables
- Write does not depend on the current value
- One state transition typically

Use Pattern #1: Status Flag

```
volatile boolean shutdownRequested;

// other code

public void shutdown() {
     shutdownRequested = true;
}

public void doWork() {
    while (!shutdownRequested) {
        // do stuff
    }
}
```

Use Pattern #1

- Example: `CalculatePrimes.java`
- Calculate as many primes as we can in ten seconds
  - One thread calculates prime numbers
  - Another one is the “timer”

Other Patterns

- http://www.ibm.com/developerworks/library/j-jtp06197/
- http://tutorials.jenkov.com/java-concurrency/volatile.html

Synchronization in Java

- Volatile variables
- Synchronized functions or code blocks
- Custom lock objects

Synchronized keyword

Synchronized Block

- A code block protected by a special "lock"
- Must have a key to the lock to enter code
  - One key may potentially unlock multiple locks
  - One lock may have potentially many keys

Synchronized Block

- A thread is blocked by the lock until it is able to get the the key
- When exiting code block, the thread returns the key

Synchronized Blocks

- Must specify object to use as a lock

```
Object lock = new Object();
// some statements

 synchronized (lock) {
       // do something
 }
```

Synchronized Blocks

- Only one thread may obtain the lock object
  - Others will be blocked
  - 2 blocks, same lock- > only 1 can be entered at a time
  - 2 blocks, different locks -> both can be entered

- Releases the lock when exits synchronized block

Examples

- WithSynchronization.java
- DataRaceSynchronized.java
- LockDemo.java

Synchronized Methods

- Can declare using a *synchronized* keyword:

```
public synchronized void func(…)
```

- The code is locked on *"this"* object

Example

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

What about Visibility?

- Consider a synchronized block
- Thread 1 exits, thread 2 enters (same lock)
- After getting the lock, Thread 2 will see all writes made by Thread 1

Concurrent Operations

- Mutual Exclusion
  - Only one thread may enter synchronized code at a time -> blocking other threads
  - Lots of blocking - no purpose in multithreading
- Conditional Synchronization
  - Only block *if* certain conditions are true
  - Uses wait() and notify()

wait()

- The thread enters “waiting” state
- Waits for some other thread to perform an action
- “Wakes up” when notify() or notifyAll() are called

wait()

- A thread releases its lock when wait() is called
- wait() must be called:
  - on **the lock object**
  - in the synchronized block
- Only return from wait() if able to reacquire a lock

notify() and notifyAll()

- Must also be called:
  - on **the lock object**
  - in the synchronized block
- Only one waiting thread woken up by notify()

wait() : Spurious wakeups

- Thread which is waiting resumes for no apparent reason
- You should always wait *while* checking some condition as follows:

```
synchronized(lock) {
    while (!condition) {
         lock.wait();
    }
 }
```

notifyAll()

- All threads waiting on the lock woken up by notifyAll()
    - - Usually used, despite sometimes being slower

Blocking Queue Example

- A Blocking Queue:
  - Blocks if trying to dequeue and it's empty
  - Blocks if trying to enqueue and it's full

- See `BlockingQueue.java`

Synchronization in Java

- Volatile variables
- Synchronized code blocks or methods
- Custom lock objects

Custom Locks

Motivation

- Need synchronization to protect
  - data (memory consistency) and
  - operations (atomicity)
- “synchronized” keyword causes blocking
  - reducing the speedup

Motivation

- Assume have a large shared data structure

Motivation

- Assume have a large shared data structure
- What operations may occur concurrently?

Motivation

- Assume have a large shared data structure
- What operations may occur concurrently?
  - Thread 1 reads A, Thread 2 reads A

Motivation

- Assume have a large shared data structure
- What operations may occur concurrently?
    - Thread 1 reads A, Thread 2 reads A
    - Thread 1 reads A, Thread 2 writes A

Motivation

- Assume have a large shared data structure
- What operations may occur concurrently
  - Thread 1 reads A, Thread 2 reads A
  - Thread 1 reads A, Thread 2 writes A
  - Thread 1 writes A, Thread 2 writes A

Motivation

- Assume have a large shared data structure
- What operations may occur concurrently?
  - Thread 1 reads A, Thread 2 reads A
  - ~~Thread 1 reads A, Thread 2 writes A~~
  - ~~Thread 1 writes A, Thread 2 writes A~~

Custom Lock Class: "MultiReadLock"

- May read to shared data structure if…
  - No other threads are writing to it
- May write to shared data structure if…
  - No other threads are reading from it
  - No other threads are writing to it
- Must track…
  - Number of active readers and writers

MultiReadLock

```
public synchronized void lockRead() {
  while (writers > 0) {
      try {
             this.wait();
      }
      catch (InterruptedException ex) {
          // log the exception
      }
  }
  readers++;
}
```

MultiReadLock

- `public synchronized void` **`lockRead()`**
  - Wait (i.e. give up lock) until no active writers
  - Use a loop to avoid spurious wakeups
    - Use wait() and notifyAll() to avoid busy-wait
  - Increase number of readers and “give” lock

- `public synchronized void` **`unlockRead()`**
  - Decrease number of readers to “free” the lock
    - Wake up threads if necessary using notifyAll()

Using MultiReadLock

```
MultiReadLock lock = new MultiReadLock ();
SharedData data = new SharedData();

lock.lockRead();  // protects read-only operations
data.read();
lock.unlockRead();

lock.lockWrite();  // protects write operations
data.read();       // or read/write operations
data.write();
lock.unlockWrite();
```

Example

```
Class SynchronizedMap {
    private final Map<String, Data> m = new TreeMap<>();
    private final MultiReadLock lock = new MultiReadLock();

    public Data get(String key) {
        lock.lockRead ();
        try { return m.get(key); }
        finally { lock.unlockRead(); }
    }

    public Data put(String key, Data value) {
        lock.lockWrite();
        try { return m.put(key, value); }
        finally { lock.unlockWrite(); }
    }
}
```

http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html

Custom Locks

- More flexible than synchronized blocks
- The con: *no one* will unlock the lock for us
  - if an exception occurs etc.
- Unlike in synchronized blocks

Unlock In a finally Block

```
MultiReadLock l = new MultiReadLock();
l.lockRead();
try {
    // whatever needs to be synchronized
}
finally {
    l.unlockRead();
}
```