Java Threads - Synchronization

Università di Modena e Reggio Emilia

Prof. Nicola Bicocchi (nicola.bicocchi@unimore.it)

Synchronization

- What happens when two different threads are accessing the same data ?
- Imagine that two people each have ATM cards, but both cards are linked to only one account

```
class Account {
   private int balance;
   public int getBalance() {return balance;}
   public void withdraw(int amount) {
         balance = balance - amount;
   }}
```

Synchronization

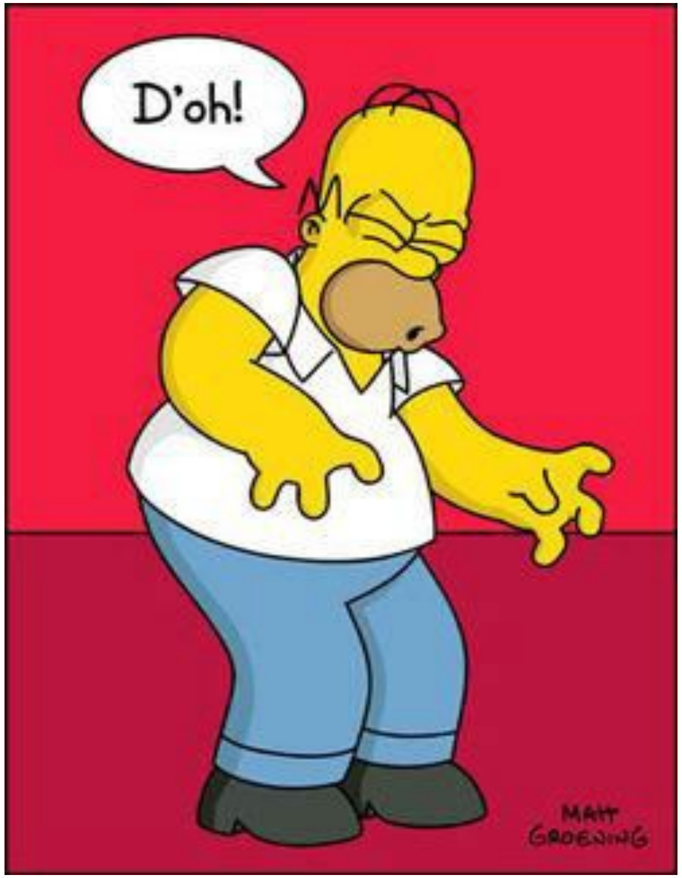
- Before one of them makes a withdrawal, first check the balance to be certain there's enough to cover the withdrawal
  - Check the balance.
  - If there's enough in the account, make the withdrawal
- What happens if something separates step 1 from step 2 ?

Synchronization

Synchronization

- Marge checks the balance and there is enough
- Before she withdraws money, Homer checks the balance and also sees that there's enough for his withdrawal. He is seeing the account balance before Marge actually debits the account…
- Both Marge and Homer believe there's enough to make their withdrawals !
- If Marge makes her withdrawal…
  - …there isn't enough in the account for Homer's withdrawal
  - … but he thinks there is since when he checked, there was enough!

Synchronization



Race condition

- A problem happening whenever:
  - Many threads can access the same resource (typically an object's instance variable)
  - This can produce corrupted data if one thread *"races in"* too quickly before an operation has completed.

Preventing Race Conditions

- We must guarantee that the two steps of the withdrawal are NEVER split apart.
- It must be an atomic operation:
  - It is completed before any other thread code that acts on the same data
  - …regardless of the number of actual instructions

Preventing Race Conditions

- You can't guarantee that a single thread will stay running during the atomic operation.
- But even if the thread running the atomic operation moves in and out of the running state, no other running thread should be able to act on the same data.
- How to protect the data:
  - Mark the variables private
  - Synchronize the code that modifies the variables

Synchronized

- The modifier synchronized can be applied to a method or a code block
- It locks* a code block: only one thread can access

Remember: the simplest lock implementation is a semaphore (provided by OSs).

Synchronization and Locks

- Every object in Java has one built-in lock
- Enter a synchronized non-static method => get the lock of the current object code we're executing.
- If one thread got the lock, other threads have to wait to enter the synchronized code until the lock has been released (thread exits the synchronized method)
- Not all methods in a class need to be synchronized.
- Once a thread gets the lock on an object, no other thread can enter any of the synchronized methods in that class (for that object).

Synchronization and Locks

- Multiple threads can still access the class's non-synchronized methods
  – Methods that don't access the data to be protected, don't need to be synchronized
  – Threads going to sleep, don't release locks
- A thread can acquire more than one lock, e.g.
  – a thread can enter a synchronized method, then immediately invoke a synchronized method on another object

Synchronization and Locks

```
public synchronized void doStuff() {
    System.out.println("synchronized");
}
```

Is equivalent to

```
public void doStuff() {
    synchronized(this) {
        System.out.println("synchronized");
}   }
```

Synchronization and Locks

```
public static synchronized int getCount() {
    return count;
}
```

Is equivalent to

```
public static int getCount(){
    synchronized(MyClass.class) {
    return count;
}
```

When Do I Need To Synchronize?

- Two threads executing the same method at the same time may:
  - use different copies of local vars => no problem
  - access fields that contain shared data
- To make a thread-safe class:
  - methods that access changeable fields need to be synchronized.
  - access to static fields should be done from static synchronized methods.
  - access to non-static fields should be done from non-static synchronized methods

Synchronize the code yourself

```
class NameDropper implements Runnable{
    public void run() {
        String name = nl.removeFirst();
        System.out.println(name);
    }
 }

public static void main(String[] args) {
    fNameList nl = new NameList();
    nl.add("Jacob");
    Thread t1 = new Thread(new NameDropper()); t1.start();
    Thread t2 = new Thread(new NameDropper()); t2.start();
}
```

ING

Synchronize the code yourself

```
public class NameList {
    private List names = new LinkedList();

    public synchronized void add(String name) {
        names.add(name);
    }

    public synchronized String removeFirst() {
        if (names.size() > 0)
            return (String) names.remove(0);
        else
            return null;
        }
    }
}
```

Collections.synchronized*

```
public class NameList {
    private List names =
    Collections.synchronizedList(new LinkedList());

    public void add(String name) {
        names.add(name);
    }
    public String removeFirst() {
        if (names.size() > 0)
            return (String) names.remove(0);
        else return null;
    }
}
```

Collections.synchronized*

public static <T> List<T> synchronizedList(List<T> list)

Returns a synchronized (thread-safe) list backed by the specified list. In order to guarantee serial access, it is critical that all access to the backing list is accomplished through the returned list.

It is imperative that the user manually synchronize on the returned list when iterating over it:

```
List list = Collections.synchronizedList(new ArrayList());
   ...
synchronized (list) {
   Iterator i = list.iterator(); // Must be in synchronized block
   while (i.hasNext())
      foo(i.next());
}
```

Failure to follow this advice may result in non-deterministic behavior.

Deadlock!

- Deadlock occurs when two threads are blocked, with each waiting for the other's lock.
  - Neither can run until the other gives up its lock, so they wait forever
- Poor design can lead to deadlock
  - It is hard to debug code to avoid deadlock
  - Model checking could be a solution (problem: state space explosion)

Deadlock!

```
public class DeadlockProducer {
private static class Resource {   public int value; }
      private Resource resourceA = new Resource();
      private Resource resourceB = new Resource();
      public int read() {
            synchronized(resourceA) {  // May deadlock here !
                  synchronized(resourceB) {
                        return resourceB.value + resourceA.value;
      } } }
      public void write(int a, int b) {
            synchronized(resourceB) {   // May deadlock here !
                  synchronized(resourceA) {
                        resourceA.value = a; resourceB.value = b;
      } } }
}
```

Livelock!

- A livelock happens when threads are actually running, but no work gets done
  - what is done by a thread is undone by another
- Example: each thread already holds one object and needs another that is held by the other thread.
  - What if each thread unlocks the object it owns and picks up the object unlocked by the other thread ?
  - They can run forever in lock-step!

Starvation!

- Wait/notify primitives of the Java language do not guarantee liveness (=> starvation)
- When wait() method is called
  - thread releases the object lock prior to commencing to wait
  - and it must be reacquired before returning from the method, post notification

Starvation!

- Once a thread releases the lock on an object (following the call to wait), it is placed in a object's wait-set
  - Implemented as a queue by most JVMs
  - When a notification happens, a new thread will be placed at the back of the queue
- By the time the notified thread actually gets the monitor, the condition for which it was notified may no longer be true …
  - It will have to wait again
  - This can continue indefinitely => Starvation

Synchronization in Object class

- void wait()
  - Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
- void notify()
  - Wakes up a single thread that is waiting on this object's lock.
- void notifyAll()
  - Wakes up all threads that are waiting on this object's lock.

Mutual Exclusion

- A thread invokes wait() or notify() on a particular object, and the thread must currently hold the lock on that object
  - Called from within a synchronized context
- A thread owns a critical region when he has called wait() and it has not released the object yet (calling notify)
- A critical region is nonsignaled when is owned by a thread, signaled otherwise.

Wait

- The wait() method lets a thread say:

"There's nothing for me to do now, so put me in your waiting pool and notify me when something happens that I care about."

- When calling wait() :
  - On a signaled object lock thread keeps executing
  - On a nonsignaled object lock thread is suspended

Notify

- The notify() method send a signal to one of the threads that are waiting in the same object's waiting pool.
- The notify() method CANNOT specify which waiting thread to notify.
- The method notifyAll() is similar but only it sends the signal to all of the threads waiting on the object.

Recap: Threads Are Hard

- Synchronization:
  - Must coordinate access to shared data with locks.
  - Forget a lock? Corrupted data.
- Deadlock:
  - Circular dependencies among locks.
  - Each process waits for some other process: system hangs.

Recap: Threads Are Hard

- Achieving good performance is hard:
  - Simple locking yields low concurrency.
  - Fine-grained locking increases complexity
  - O.S. limits performance (context switches)
- Threads not well supported:
  - Hard to port threaded code (PCs?   Macs?).
  - Standard libraries not thread-safe.
- Hard to debug :
  - data dependencies
  - timing dependencies
  - Few debugging tools

ING