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 two people (represented by two threads) each one having an ATM card linked to the same account

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

Synchronization

- Each person (i.e., thread) does these steps
  1. Decide an amount to withdrawal
  2. Check the account balance.
  3. If there's enough money in the account, withdrawal the decided amount
- What happens if the scheduler suspends one thread between step 2 and step 3 and the other one gets executed?

Synchronization

Synchronization

- Homer decide to withdraw 100$ and verifies that the account contains 125$!
- Homer enters the status RUNNABLE
- Marge decide to withdraw 120$ and verifies that the account contains 125$!
- Marge withdraws 120$
- Homer enters the status RUNNING
- Homer withdraw 100$ (he already checked!) but the ATM gives him only 5$

Synchronization

ING

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.
- Withdrawal must be an atomic operation:
  - Any withdrawal must be completed before any other thread is allowed to act on the account
  - …regardless of the number of actual instructions

Synchronized

- You can't guarantee that a single thread will stay running during a whole operation (supposed to be atomic). Developers do not control the scheduler.
- The modifier synchronized can be applied to a method or a code block
- Synchronized locks a code block: only one thread can access

Synchronization and Locks

- Every object in Java has one built-in lock
- Enter a synchronized non-static method means getting the lock of the object. If one thread gets the lock, the other threads have to wait to enter the synchronized code until the lock is 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

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


Is equivalent to


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

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

Thread-safe classes

- A thread-safe class is class that is safe (works properly) when accessed by multiple threads. Critical sections (i.e., sections containing race conditions) are encapsulated in synchronized methods.
  - Vector is thread-safe (all methods are synchronized) version of ArrayList

Synchronized code

- The most common problem of concurrent access is the producer-consumer problem
  - The producer thread pushes elements into a shared object. The consumer thread fetches them
- There are several ways to synchronize access to a shared object. Basically:
  - Use synchronize in producers and consumers to lock the shared object
  - Use thread-safe classes as shared objects (they use synchronized on their methods)

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.

Synchronization using Object

- Producers and consumers might be able to lock (acquire exclusive access) a shared resource but still be unable to progress.
  - E.g., a producer with a full buffer, a consumer with an empty buffer
- For avoiding a waste of computational resources we can use
  - yield()
  - wait()/notify()

Synchronization using Object

- 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.

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.

Thread issues

- If code is correctly synchronized, it still might not work. The main issues are:
  - Deadlock (indefinite wait)
  - Livelock (threads running but no work gets done)
  - Starvation (thread never executes)

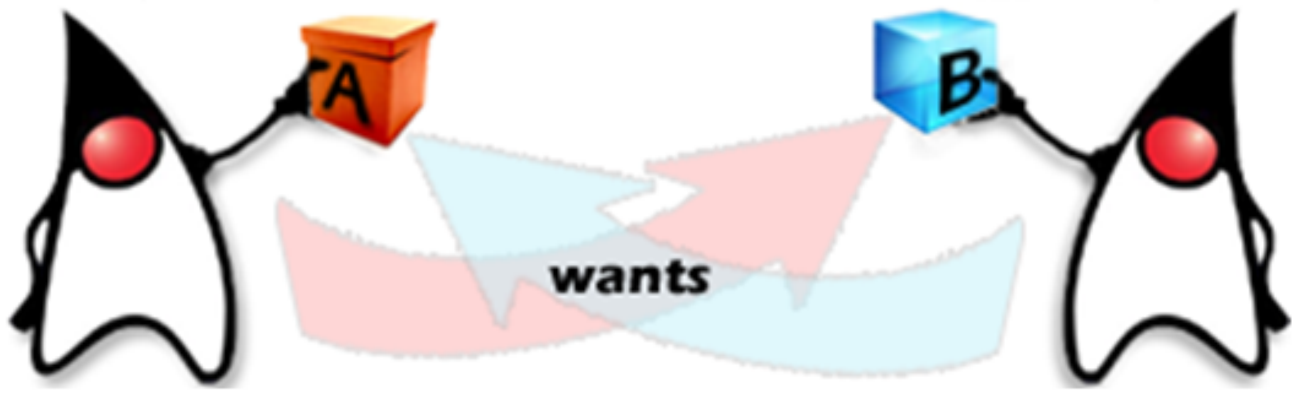
Deadlock

- 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

Thread 1 is holding Resource A

Thread 2 is holding Resource B



but wants Resource B

but wants Resource A

Livelock

- A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then livelock may result.
- As with deadlock, livelocked threads are unable to make further progress.
- However, the **threads are not blocked** — they are simply too busy responding to each other to resume work.

Starvation

- Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress.
- For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

Starvation

Running Java Thread
Starving Thread
Higher Priority Threads waiting...

Recap: Threads Are Hard

- Synchronization:
  - Must coordinate access to shared data with locks.
  - Forgot a lock? Enjoy corrupted data.
- Achieving good performance is hard:
  - Simple locking yields low concurrency.
  - Fine-grained locking increases complexity
- Threads not well supported:
  - Standard libraries not thread-safe.
- Hard to debug :
  - Data and Timing dependencies
  - Few debugging tools