Java Threads - Synchronization

Università di Modena e Reggio Emilia

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

ING

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 -= amount;
    }
}
```

Synchronization

- Each person (i.e., thread) does these steps
  1. Decide an amount to withdrawal
  2. Check the balance of the account
  3. If there's enough money, 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

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

Race condition

- A problem arising whenever two or more threads share the same resource (typically an object) and one thread *"races in"* too quickly before another operation has been completed

ING

Preventing Race Conditions

- We must guarantee that the steps comprising the withdrawal process are NEVER split apart
- Withdrawal must be an atomic operation
  - Any withdrawal (accomplished by one thread) must be completed before any other thread is allowed to act on the account
  - Regardless of the number of actual instructions!

Synchronized

- Developers can't guarantee that a single thread will stay running during a whole operation (supposed to be atomic for avoiding race conditions). In fact, developers can not control the scheduler (excluding the case of calling yield()).
- In Java, **synchronized** methods are used to protect access to resources that are accessed concurrently. Only one thread at a time can access
- The modifier **synchronized** can be applied either to a method or an 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

- Every object in Java has ONE built-in lock
- Entering a synchronized non-static method means getting the lock of the object. If one thread gets the lock, all other threads have to wait to enter ALL the synchronized code until the lock is released (the first thread exits the synchronized method)
- Entering a synchronized static method means getting the lock of the class instead of an object. Useful when the shared resource is defined static as well.

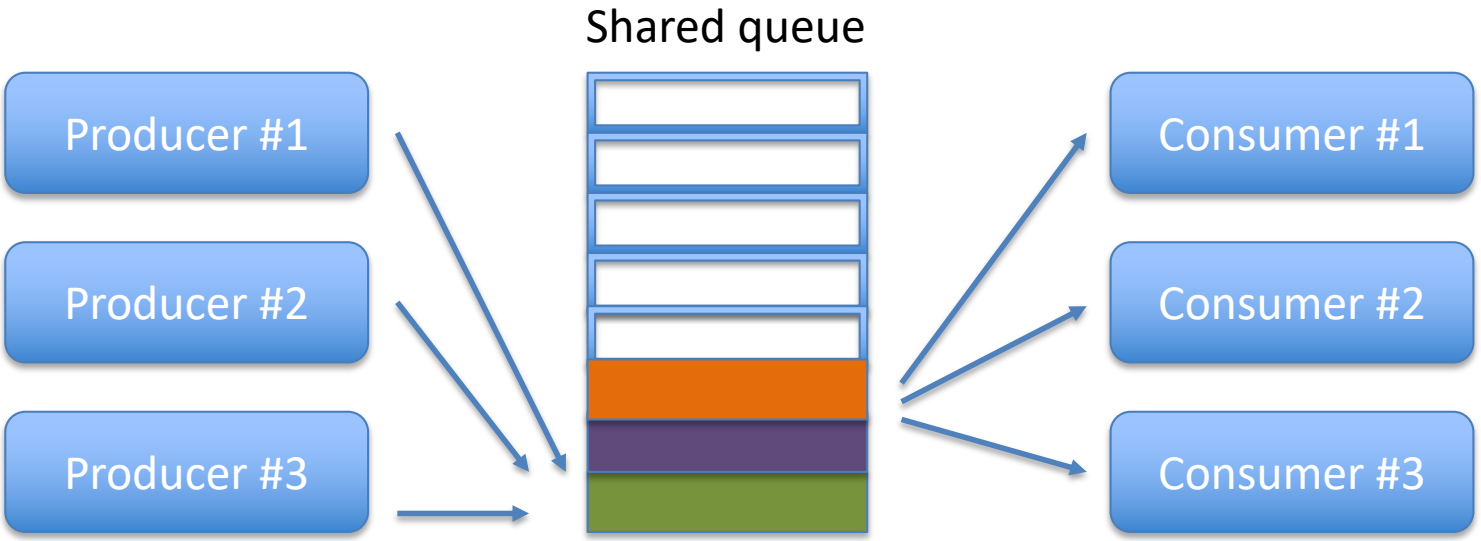
Synchronization and Locks

- Whenever an object lock has been acquired by one thread, other threads can still access the class's non-synchronized methods. Methods that don't access critical data don’t need to be synchronized
- Details:
  - Threads going to sleep don't release locks!
  - A thread can acquire more than one lock. For example, a thread can enter a synchronized method, then immediately invoke a synchronized method on another object (deadlock prone!)

Synchronized code

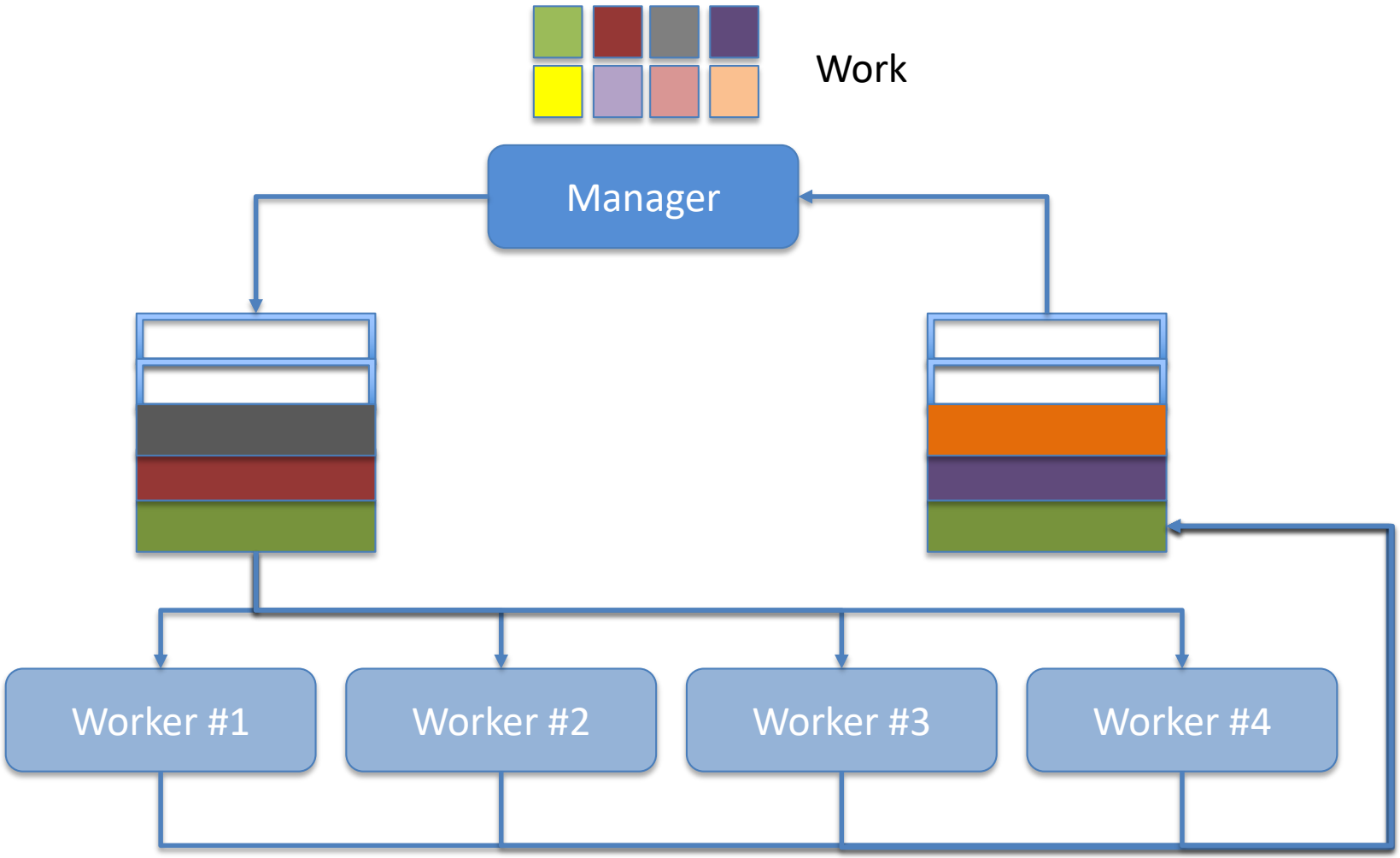
- Despite threads can be used for solving a number of real-world problems, most of them can be conceptually assimilated to two main patterns:
  - The producer-consumer pattern, where the producer thread pushes elements into a shared object and the consumer thread fetches (consumes) them
  - The manager-worker pattern, where a manager decomposes a complex task into subtask, and assigns them to worker threads.

Producer Consumer



ING

Manager Workers



Synchronized code

- There are two main ways to grant atomic access to a shared object:
  - Use synchronize explicitly to lock the shared object within the threads code (@see Account example)
  - Use a thread-safe class as shared objects (they use synchronized on their methods) (@see GranularityExample)

Thread-safe classes

- A thread-safe class is class that is safe (works properly) when accessed by multiple threads. Critical sections (i.e., sections possibly generating race conditions) are encapsulated in synchronized methods.
  - Interface List: ArrayList (unsafe), Vector (safe)
  - Interface Queue: LinkedList (unsafe), ConcurrentLinkedQueue (safe), ArrayBlockingQueue (safe)

Collections.synchronized*

- Return a synchronized (thread-safe) Collection/Map backed by the specified Collection/Map. In order to guarantee serial access, it is critical that all access to the backing Collection/Map is accomplished through the returned Collection/Map.
- It is imperative that the user manually synchronize on the returned Collection/Map when iterating over it! Failure to follow this advice may result in non-deterministic behavior.

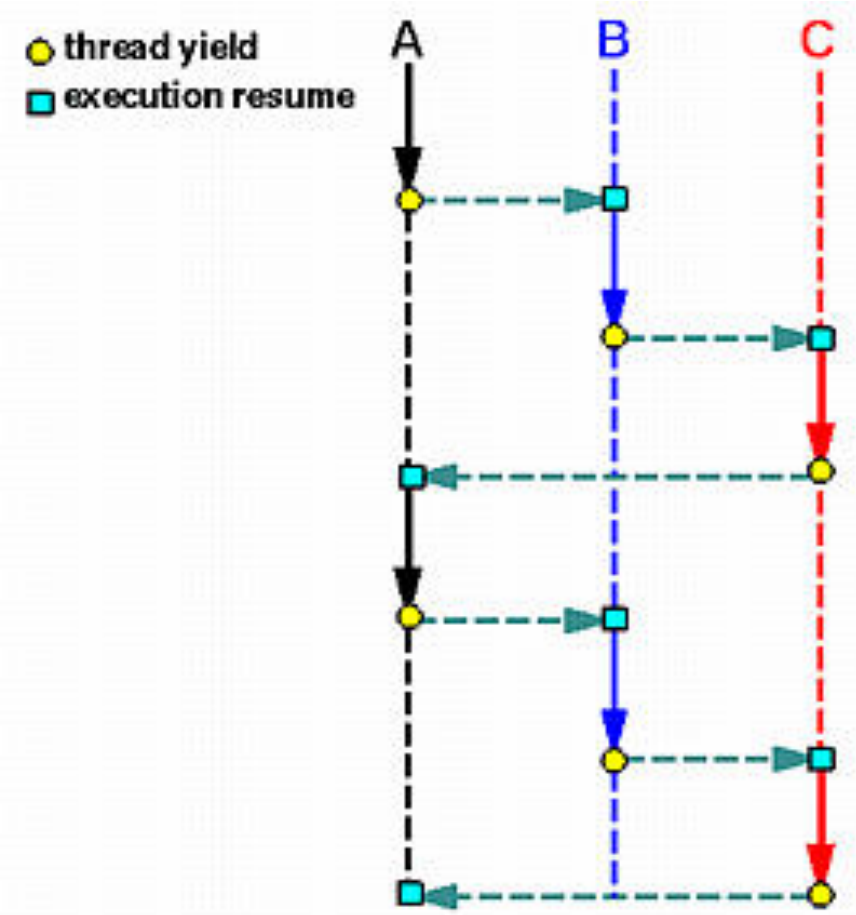
```
List list = Collections.synchronizedList(new ArrayList());
      ...
synchronized (list) {
    Iterator i = list.iterator(); // Must be in synchronized block
    while (i.hasNext())
        foo(i.next());
}
```

Synchronization using Object

- Threads might be able to acquire exclusive access a shared resource but still be unable to progress. For example, a producer with a full queue, a consumer with an empty queue
- To avoid waste of computational resources we can use:
  - yield()
  - wait()/notify()

yield()

- The method yield() make the currently running thread back to Runnable state
  - It allows other threads to get their turn
  - However, it might have no effect at all. In fact, there's no guarantee the yielding thread won't be scheduled again for execution.



wait()

- wait() is an instance method that's used for thread synchronization.
- wait() can be called on any object, as it's defined right on java.lang.Object
- wait() can only be called from a synchronized block. It releases the lock on the object so that another thread can jump in and acquire a lock.
- Simply put, wait() method lets a thread say:

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

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 sends a signal to all the threads waiting on the object.
- Simply put, notify() method lets a thread say:

“Something has changed here. Feel free to continue what you were trying to do”

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)

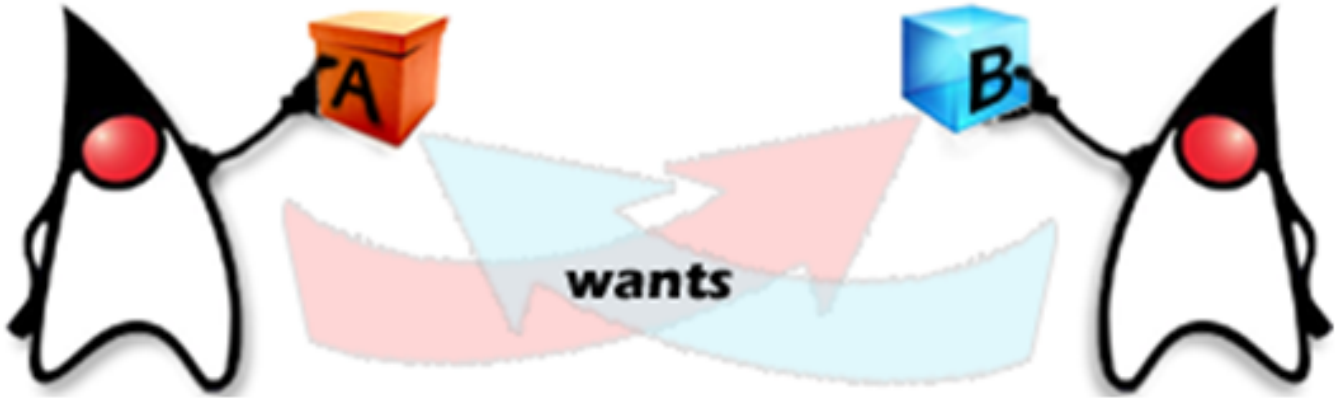
Deadlock

- Deadlock occurs when two threads are blocked, with each other 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

ING

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

ING

Recap: Threads Are Hard

- Synchronization
  - Must coordinate access to shared data with locks. Forgot a lock? Enjoy little predictable runtime errors
- Performance issues
  - Simple locking yields low concurrency
  - Fine-grained locking increases complexity
- Hard to debug