Conditions Variables in Java

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Condition Variable in Objects?

- Remember that in Java every Object has inside it a “hidden” lock?
- Well, in Java every Object has inside it a “hidden” condition variable as well!
- Technically we say that Java implements monitors
- A concept proposed in the 70s (popularized by Hoare)
- A monitor is an abstract data type in which
  - All methods are in mutual exclusion
  - There is a hidden condition variable
- The idea was to make concurrency easier (don’t have to declare locks and condition variables, never forget to unlock)
  - It’s really more about software engineering than anything else

Java Condition Variables

- Each Java monitor encapsulates **one** condition variable

- Operations on the condition variable are:
  - □ **notify();** **// a.k.a. signal()**
  - □ **notifyAll();** **// a.k.a. broadcast()**
  - □ **wait();** **// releases the lock, waits, and re-acquires the lock**
  - □ **These are methods that EVERY object has**
  - □ Always to be used **inside** a synchronized method
    - Since condition variables and locks are intertwined, as we saw in the previous set of lecture notes
- Let’s look at a simple example…

Example

```
public class MyThread extends Thread {

  public void run() {
    while (true) {
      synchronized(this) {
        this.wait();
      }
      System.out.println(“Awakened”);
    }
  }
}
```

```
. . .

MyThread thread = new
MyThread();
thread.start();

while (true) {
  Thread.sleep(1000);
  synchronized (thread) {
    thread.notify();
  }
}
```

Example

```
public class MyThread extends Thread {

  public void run() {
    while (true) {
      synchronized(this) {
        this.wait();
      }
      System.out.println(“Awakened”);
    }
  }
}
```

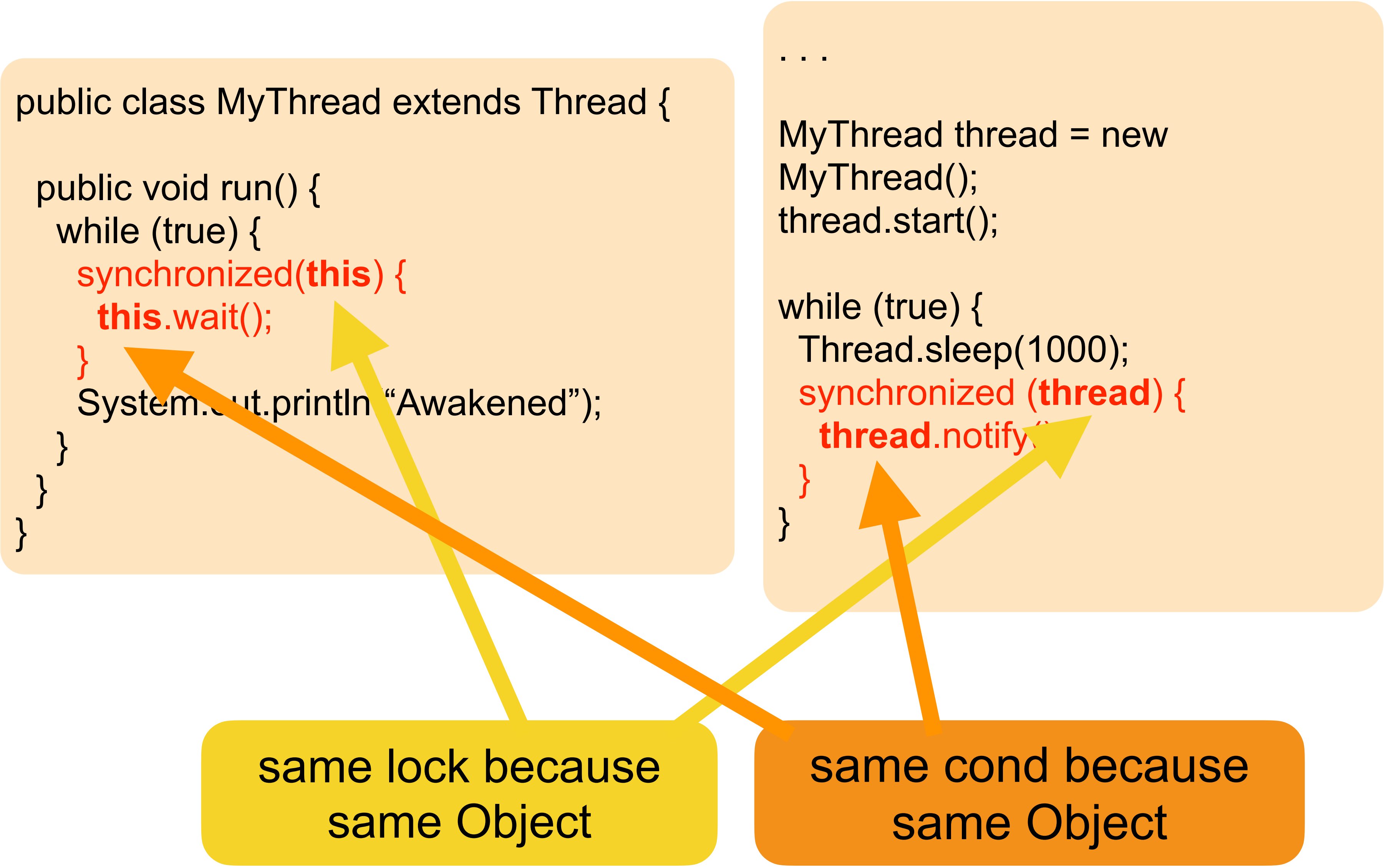
```
. . .

MyThread thread = new
MyThread();
thread.start();

while (true) {
  Thread.sleep(1000);
  synchronized (thread) {
    thread.notify();
  }
}
```

same lock because
same Object

Example



```
public class MyThread extends Thread {

  public void run() {
    while (true) {
      synchronized(this) {
        this.wait();
      }
      System.out.println(“Awakened”);
    }
  }
}
```

```
. . .

MyThread thread = new
MyThread();
thread.start();

while (true) {
  Thread.sleep(1000);
  synchronized (thread) {
    thread.notify();
  }
}
```

same lock because same Object

same cond because same Object

Example

```
public class MyThread extends Thread {

  public void run() {
    while (true) {
      synchronized(this) {
        this.wait();
      }
      System.out.println(“Awakened”);
    }
  }
}
```

```
. . .

MyThread thread = new
MyThread();
thread.start();

while (true) {
  Thread.sleep(1000);
  synchronized (this) {
    this.notify();
  }
}
```

Not the same “this”

HORRIBLE BUG
Different Object

Example: A barrier

- In the previous set of lecture notes we saw how to implement a barrier in pseudo-code
- Remember: it’s about threads waiting for each other until N threads have “reached” (i.e., called) the barrier
- Let’s develop a barrier live in Java…
  - Before we look at the solution on the next slide

Example: A barrier

```
public class Barrier {
    private int maxNumThreads;
    private int numCalls = 0;

    public Barrier(int numThreads) {
        this.maxNumThreads = numThreads;
   }

    public synchronized void call()  {
        numCalls++;
        if (numCalls == maxNumThreads) {
            this.notifyAll();
            numCalls = 0;
        } else {
            try {
                this.wait();
            } catch (InterruptedException ignore) { }
        }
    }
 }
```

java.util.concurrent.CyclicBarrier

- Turns out, the barrier abstraction is so useful that Java provides it in the java.util.concurrent package
- It’s called CyclicBarrier
  - Cyclic because it can be re-used, like the one implemented in the previous slide
- It provides a few useful features
  - e.g., a way to call the barrier with a timeout

Flashback to the Blocking Lock!

- At the beginning of the previous set of lecture notes we tried using a blocking lock for communication purposes
  - I gave some arguments to say it was conceptually a bad idea (which you have to trust me on a bit)
  - And then we said “here are condition variables”
- But I also said that, strangely, the implementation of a blocking lock really ressembles that of a condition variables
- Just for kicks, let’s implement, in Java, a blocking lock using the (hidden) spinlock and a condition variable inside a BlockingLock object
- Let’s do it live (an implementation is on the Web site, likely 100% similar to what we’re about to write)
  - It’s basically a super-simple barrier!

Abstraction but Less Flexibility

- Each Java Object has one lock and one condition variable in it
- In the previous set of lecture notes we have seen a Producer/Consumer implementation that uses one lock and TWO condition variables “associated” to the same lock
- Therefore, you simply cannot do this using only synchronized, wait, and notify
- You have to go brute-force:
  - Using the same condition variable for all events
  - Always call notifyAll()
  - e.g., If a Producer puts an item in the buffer, it will wake up ALL threads
- This is not very efficient (imagine waking up 1000 threads for whom the event is irrelevant!)
  - The curse of high-level “easy” abstractions

Abstraction: good or bad?

- Java tries to hide concurrency by using monitors
- However, to truly understand how things work, many think it's useful to actually think of locks and condition variables underneath the abstractions
- And also, the previous slides shows that abstraction can be unwieldy
- For locks, we have seen that `java.util.concurrent` provides Lock classes
- It also provides a method to create condition variables
- Because a condition variable must be associated to a lock, this method is part of the Lock interface
- Let's see an example….

java.util.concurrent Conditions

```
Lock lock = new ReentrantLock();
Condition cond  = lock.newCondition();
```

```
…
lock.lock();
try {
   cond.signal();
}  finally {
  lock.unlock();
}
…
```

```
…
lock.lock();
try {
   cond.await();
}  finally {
  lock.unlock();
}
…
```

- Note that in all code fragments shown in these slides, I do not show the try-catch for InterruptedException

Pausing/Resuming Java Threads

- In the Java Thread module we talked about deferred thread cancelation
    - How to stop a thread using a volatile boolean
- Now that we have condition variables we can learn how to pause (and resume) a Java Thread
- Just like Thread.stop(), Thread.pause() and Thread.resume() have been deprecated for a long time
- And so we need deferred thread pausing
- Do do this, we use a condition variable

Pausing/Resuming Java Threads

- Approach:
  - The thread has a **volatile** variable
  - The thread periodically checks whether the variable is set to true
  - If isSuspended is true, the thread blocks by calling wait()
  - The thread can be unsuspended by setting the variable to false and calling notify()
- Let’s see this in Java…
  - Using the built-in condition variable in a monitor

Pausable Java Thread Example

```
public class PausableThread extends Thread {
  private volatile boolean isPaused = false;

  public void pause() {
    this.isPaused = true;
  }

  public void unPause() {
    this.isPaused = false;
    this.notify();
  }

  public void run() {
    while(true) {
      . . .
      while (isPaused) {
        try {
          this.wait();
        } catch  (InterruptedException e) { }
      }
    }
  }
}
```

```
Thread t = new
    PausableThread();
t.start();
. . .
t.pause();
. . .
t.unPause();
. . .
```

Pausable Java Thread Example

```
public class PausableThread extends Thread {
  private volatile boolean isPaused = false;

  public void pause() {
    this.isPaused = true;
  }

  public void unPause() {
    this.isPaused = false;
    this.notify();
  }

  public void run() {
    while(true) {
      . . .
      while (isPaused) {
        try {
          this.wait();
        } catch  (InterruptedException e) { }
      }
    }
  }
}
```

```
ead();
```

This code will not compile because notify() and wait() need to be called from synchronized methods/blocks!

Pausable Java Thread Example

```
public class PausableThread extends Thread {
  private boolean isPaused = false;

  public synchronized void pause() {
    this.isPaused = true;
  }

  public synchronized void unPause() {
    this.isPaused = false;
    this.notify();
  }

  public synchronized void checkPaused() {
    while (isPaused) {
      try {
        this.wait();
      } catch  (InterruptedException e) { }
    }
  }

public void run() {
    while(true) {
       . . .
      checkPaused();
    }
  }
}
```

- And now, we no longer need the **`volatile`**!
  - To be 100% sure we make the pause() method synchronized
  - Likely paranoid since the memory fence instructions will be called by unPause() and checkPaused()
  - See Locks module ("How to use Locks in Java")

More complicated example

- Say a thread is doing, in an infinite loop:
  - print “hello”
  - sleep for 10 seconds
- We want to make this thread pausable
- The difficulty: we also want to pause the sleep!

- Let’s try to implement this live…
  - It could get a bit dicey…
  - I put an implementation on the course’s site

Conclusion

- At this point, we know how to do the two fundamental things for concurrency in Java:
  - Mutual exclusion with locks: synchronized
  - Communication with condition variables: wait/notify
- We know to do this using Java monitors, or using classes in java.util.concurrent
- We know how to stop/pause/resume a Java Thread
- Let's look at Homework Assignment #6…