

Mutual Exclusion and Synchronization in Java

ICS432 - Fall 2008 Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Java and Monitors

- Java implements (a restricted) Monitor abstraction
- You can declare a class and have it behave like the monitors we described in the previous set of slides
- Each Java Object has a monitor within it
 - There is no "Monitor" class since the Object class incorporates the functionality of a monitor
 - Actually there is, but for "Managed Beans"
- Some of the object's methods can be declared **synchronized**
 - A class can have both synchronized and non-synchronized methods
- Synchronized methods are executed in mutual exclusion
 - In other terms, synchronized methods are the monitor's methods as well as the object's methods

Java and Monitors

- Each Java monitor has **one** condition variable
 - In the previous set of slides we considered multiple condition variables per monitor
 - In Java, since there is only one, it's not even declared/accessable (it has no name)
- Operations on the condition variable are:
 - **notify();**
 - **notifyAll();**
 - **wait();**
 - Always used inside a synchronized method
- Let's look at a simple example

Synchronized Methods

```
public class SomeClass {  
    public synchronized void SomeMethod() {  
        ...  
    }  
  
    public synchronized void SomeOtherMethod() {  
        ...  
    }  
}
```

- number of current activations of SomeMethod() + number of current activations of SomeOtherMethod() ≤ 1 at all times
 - Coarse-grain mutual exclusion (It's a Monitor)
 - Implemented internally with a single lock

Example

```
public class Counter {  
    private int value;  
    public Counter() {  
        value = 0;  
    }  
  
    public synchronized void increment() {  
        value++;  
    }  
  
    public synchronized void decrement() {  
        value--;  
    }  
  
    public synchronized int getValue() {  
        return value;  
    }  
}
```

```
Counter counter = new Counter();  
  
// Thread 1  
...  
counter.increment();  
...  
  
// Thread 2  
...  
counter.decrement();  
...
```

- Class counter is a monitor
- It is thread-safe
- Class users do not have to worry about anything

Synchronized Statements

- It is not always convenient to have an entire method be used in mutual exclusion
 - Perhaps there are only a few "critical" statements in the method
 - And in fact, we have seen that the shorter the critical sections the better
 - The solution could be to put the critical statements in their own methods
 - But then we artificially create more method calls
 - Which in turns harms performance
- As a result, Java provides ways to have synchronized statements inside non-synchronized methods

Synchronized Statements

```
public class Counter {
    private int value;
    public Counter() {
        value = 0;
    }

    public void increment() {
        System.out.println("hello");
        synchronized(this) {
            value++;
        }
        // do some other thing
    }
    ...
}
```

- Two threads can print "hello" at the same time
- But only one can increment the value at a time
- The synchronized(this) statement makes it possible to make short critical sections and thus maximize concurrency

Only One Monitor?

- Having only one monitor per object can be a problem
- Say you define a class in which not all methods need to be in mutual exclusion, but only some of them
- Example:
 - Methods f1() and f2() should be executed in mutual exclusion
 - Methods f3() and f4() should be executed in mutual exclusion
 - But f1() and f3() can be executed concurrently

Example: Two Counters

```
public class TwoCounters {
    private int value1, value2;
    public TwoCounters() {
        value1 = 0; value2 = 0;
    }

    public void increment1() {
        synchronized(this) {
            value1++;
        }
    }

    public void increment2() {
        synchronized(this) {
            value2++;
        }
    }
}
```

- This solution is correct, but overly restrictive
- Two threads should be allowed to update two different counters simultaneously
- This is similar to the "reader/writer" problem in which we wanted some threads to be in mutual exclusions and some not to be
- In this case, with a single monitor there is no way to do this
- Therefore, we need to have multiple monitors
- Problem: The TwoCounters object has only one monitor

Synchronizing on Multiple Objects

- When synchronizing statements one uses the **synchronized(this)** statement
- The "this" specifies that one uses the monitor of the current object
- This is standard, but in fact one can synchronize on the monitor of **any** object
- This is really against the definition of a monitor we saw previously
 - The whole point was that everything was encapsulated within a single monitor object
- But it's necessary to achieved higher concurrency as in the previous example

Example: Two Counters

```
public class TwoCounters {
    private int value1, value2;
    private Object lock1, lock2;

    public TwoCounters() {
        value1 = 0; value2 = 0;
        lock1 = new Object();
        lock2 = new Object();
    }

    public void increment1() {
        synchronized(lock1) {
            value1++;
        }
    }

    public void increment2() {
        synchronized(lock2) {
            value2++;
        }
    }
}
```

- Now we have a distinct monitor for each counter
- Note that these monitors are encapsulated within Object objects
- I name these objects lock1 and lock2 just to remind myself that they are used exclusively for mutual exclusion
 - I could have used any object in the program really, but it's typically not very readable
- Some programmers find this confusing, but you only have to remember that all Java objects incorporate a monitor

Synchronized Class

- So far, we have seen mutual exclusion over objects, i.e., instances of classes
- Sometimes one wants a particular method to be in mutual exclusion **over all instances** of the class
- Example: only one of the counter objects in the whole program can update some global sum of all the counters
- One way to do this is to encapsulate the global sum in its own class, with its own monitor
- Another way is to declare a "class method", i.e., a static method, as synchronized
- Let's see an example

Example

```
public class Counter {
    private int value;
    static private int sum = 0;

    public Counter() {
        value = 0;
    }

    public void increment() {
        synchronized(this) { value++; }
    }

    public static synchronized void updateSum() {
        sum += value;
    }
}
```

- Only one Counter object can update the sum class variable at a time

Summary So Far

- The synchronized keyword is the way to implement mutual exclusion
 - At the class level
 - At the method level
 - At the statement level
- One often creates objects just for the purpose of using their monitors for mutual exclusion
 - which in effect simulates the basic lock mechanisms, and goes a bit against the whole monitor philosophy

Wait/Notify in Java

- Since each Java object has a monitor, each object also has a condition variable
- Each Java object inherits three methods from the Object class
 - wait(): the calling thread blocks
 - notify(): unblock the first blocked thread
 - notifyAll(): unblock all blocked threads
- These three methods must be called within synchronized methods/blocks
- In good condition variable fashion, when a thread calls wait() it releases the lock on the object
- When a thread calls notify() it keeps the lock and continues executing
 - Since it is in a synchronized method/block
- The awakened thread(s) will acquire the lock whenever possible

Example

```
public class MyThread extends Thread {
    public void run() {
        while (true) {
            synchronized(this) {
                this.wait();
            }
            System.out.println("Awakened");
        }
    }
}
```

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

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

Where are we now?

- At this point, we know how to do the two fundamental things for concurrency in Java:
 - Mutual exclusion: synchronized
 - Signaling: wait/notify
- So far, we're almost in line with our discussion of the clean monitor model from the previous set of slides
 - Besides the fact that we can basically declare locks
- Unfortunately, there are other concepts that are Java-specific that a programmer must understand to write good concurrent Java code
- These concepts are confusing to many programmers, so let's look at them
 - The dreaded volatile keyword
 - Ways to stop, suspend, resume threads

The volatile Keyword

- For optimization purposes, the JVM allows threads to cache copies of instance or class variables
 - So while you think of two threads "share" a variable in your program, they may each work on their own copy!!
 - Working on a private copy, in particular on a multi-processor system, could lead to much better performance
 - Each processor has its own copy in its own cache
- The problem is that if a thread sets a variable to some value, other threads *may* not see that value!
 - They will eventually, but the delay may be significant
- To avoid this problem, Java provides the **volatile** keyword
- When a class/instance variable is declared volatile, the JVM will not allow threads to cache the value
- Instead, **the variable is kept in a single location** (Java calls that location "main memory")
 - As a result, volatile variables may have a higher access overhead than regular variables because they can never be cached

The volatile Keyword

- Volatile variables are synchronized across threads: **Each read of a volatile will see the last write to that volatile**

```
public class SomeClass {
    private int var1;
    private volatile int var2;

    public int get1() {
        return var1;
    }

    public int get2() {
        return var2;
    }
}
```

```
SomeClass stuff;
...
// Thread 1
System.out.println(stuff.get1());
...
System.out.println(stuff.get2());
...
...
// Thread 2
System.out.println(stuff.get1());
...
System.out.println(stuff.get2());
...
```

may see different values!!!

volatile and atomicity

- The subtlety is that having a variable declared volatile does not enforce that it can be updated atomically at all
- Having multiple threads do “var++” on volatile variable var is still a race condition
 - Thread #1 reads the value
 - Thread #2 reads the value
 - Thread #1 writes the value
 - At this point all threads reading the value see the new value
 - Thread #2 writes the value
 - We still have a lost update
 - In spite of Thread #2 always seeing the newest value
- Therefore, **volatile** does not subsume **synchronized**

volatile and synchronized

- Should we declare every shared variable volatile?
- Turns out, we don't have to because synchronized ALSO synchronizes memory
 - Acquiring a lock forces all variable values to be updated with “main memory” values
 - Releasing a lock forces all written variable values to be written to “main memory”
- In this sense, volatile is a weaker form of synchronized
 - volatile synchronizes a single variable, not the whole thread's memory
 - and of course, volatile doesn't imply any critical section of code
- So in all the examples we've seen so far we never needed to worry about “will the thread see the last value?” because accesses to shared variables were within synchronized methods!

volatile and synchronized

```
public class SomeValue {
    private float value = 0.0;

    synchronized void set(float v) {
        value = v;
    }

    synchronized float get() {
        return value;
    }

    ...
    SomeValue value = new SomeValue();
    ...
}
```

```
...
volatile float value = 0.0;
...
```

- A volatile variable removes the need for a small class that would be used just so that all threads see the value written last!
- You can replace the whole code on the left with the code on the right
- But not if the class on the left has an “update” method that needs to be in mutual exclusion

So When Do I Use volatile???

- Clearly, if multiple threads “update” a variable, you need synchronized methods/statements
 - In this case, there is no need for a volatile variable, because synchronized also ensures that the value written last is seen by all threads
- But if you have a variable written to by 1 thread and read by N threads, then you don't need to go synchronized, volatile will do the job
 - with less overhead to boot!
- We will see this in action on our next topic, stopping threads

What You Must Know

- The value of a “volatile” variable read by a thread always reflects the last write to that variable
- This may not be the case for variables not declared “volatile” and written/read by multiple threads
- “synchronized” also implies that a thread will see the last written value of a variable
- So, if a shared variable is written/read in “synchronized” methods, “volatile” is superfluous
- Typically, one uses volatile when
 - One thread writes the variable
 - One or more threads read the variable
- For something more involved, we use “synchronized”
 - But there may be some non-synchronized reads!

Final Word on volatile

- There is a whole section of the JVM on the JMM (Java Memory Model)
 - Many argue that the JMM must be evolved
 - Some JVMs don't even implement volatile correctly yet!
 - This is an area that's bound to evolve
 - There are many on-line discussions on this topic, and many confused people, for good reasons
 - The volatile keyword is only one of the issues
- Finally, remember that up-to-date values of non-volatile variables may be available to all threads, but that they may not
 - Depends on the compiler, the JVM, the system
- So you may write perfectly working code without ever using volatile, but sometimes you may encounter problems
 - Which adds to the already prominent concurrency "once in a while it doesn't work, or on some machine it doesn't work" problem :(

Interrupting, Stopping, etc.

- We need a way for a thread to stop, pause, resume, another thread
 - Useful for GUIs for instance if you have a "cancel" button that can cancel an ongoing operation
 - Useful for applications in which a worker thread's work becomes useless due to another worker's work
 - Useful to implement a "Pause" button
 - Generally useful to do your own thread scheduling
 - Useful to "shutdown" an application
 - etc.
- Question: How do we do it in Java?

Deprecated stop(), resume(), etc.

- Java used to provide the following methods
 - Thread.stop(): Kills a thread
 - Thread.suspend(): pauses a thread
 - Thread.resume(): resumes a paused thread
- These methods were simple to use but they are now **DEPRECATED**
- Deprecated methods are methods whose usage will be discontinued and is thus being strongly discouraged
 - In fact, you have to compile with a special flag to use deprecated methods
- **Question:** Why make such useful methods deprecated???

Deprecated stop()

- The stop() method kills a thread
- As a result, all monitors locked by the killed thread are unlocked
- These monitors were locked for a reason, mostly likely some mutual exclusion
- Therefore, the killed thread may have left some shared variables in an inconsistent state in the middle of some critical section
- When other threads continue accessing these shared variable odd things could happen
- In many programs, stop() is called "at the end", when bad things most likely won't happen
- Nevertheless, the Java team decided to remove the option of just killing a thread!

Deprecated suspend()

- Consider a thread that has locked several monitors
- Then this thread is suspended by a call to suspend()
- The monitors that thread had locked cannot be released otherwise mutual exclusion executions of synchronized routines could be compromised
- But if they are not released, then other threads cannot acquire them
- Therefore, we may have deadlocks
- Here again, the Java people decided to remove Thread.suspend()
 - And Thread.resume() as well

So what do we do???

- This is all well and good, but we still need to kill threads, to suspend them and to resume them later!!
- Let's start with stopping a thread
- The point of deprecating the stop() method is that a thread shouldn't be stopped just at any point of its execution
- This means that the programmer must specify when a thread could be stopped
- During its execution, a thread should check regularly whether it should stop or not
- This can be done by the means of a shared variable
- Let's see this on an example

Killing a Thread

```
public class MyThread extends Thread {
    private boolean stopped = false;

    public void stopThread() {
        stopped = true;
    }

    public void run() {
        while(true) {
            <do some work>
            if (stopped) {
                break;
            }
        }
        <clean up>
        return;
    }
}
```

```
...
MyThread thread
thread = new MyThread();
...
thread.start();
...
// stop the thread
thread.stopThread();
...
```

Killing a Thread

```
public class MyThread extends Thread {
    private boolean stopped = false;

    public void stopThread() {
        stopped = true;
    }

    public void run() {
        while(true) {
            <do some work>
            if (stopped) {
                break;
            }
        }
        <clean up>
        return;
    }
}
```

```
...
MyThread thread
thread = new MyThread();
...
thread.start();
...
// stop the thread
thread.stopThread();
...
```

- There is a small problem with this code
- Anybody see what it is?

Killing a Thread

```
public class MyThread extends Thread {
    private volatile boolean stopped = false;

    public void stopThread() {
        stopped = true;
    }

    public void run() {
        while(true) {
            <do some work>
            if (stopped) {
                break;
            }
        }
        <clean up>
        return;
    }
}
```

```
...
MyThread thread
thread = new MyThread();
...
thread.start();
...
// stop the thread
thread.stopThread();
...
```

- If stopped is not declared **volatile**, there could be a delay before the thread actually realizes that stopped was set to true!
- If you don't know about volatile, you could spend days wondering why your thread doesn't die!

Killing a Thread

- So at this point, stopping a thread is a bit of a hassle but it's manageable
- If you want extreme responsiveness though, the thread should check on whether it was stopped very often
 - This could clutter the code
- But what if the thread is sometimes blocked on some operations?
 - wait(), sleep(), join(), etc.
- Clearly, while a thread is blocked it cannot check to see whether it was stopped or not
- Sometimes, you want the thread to die while it is blocked because something bad could happen if it waits until it's unblocked to check whether it should abort or not

The interrupt() method

- The Thread class provides an interrupt() method
 - This is a very confusing name, so beware
- Calling interrupt() causes an InterruptedException to be raised if/while the target thread is blocked
- As you see in compilation error messages, several blocking functions mandate a try block and a catch for the InterruptedException
- Example:

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    // Perhaps do something
}
```

Killing a Thread

```
public class MyThread extends Thread {
    private volatile boolean stopped = false;
    public void stopThread() {
        stopped = true;
    }

    public void run() {
        ...
        if (stopped) { return; }
        ...
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            return;
        }
        ...
    }
}
```

```
...
MyThread thread
thread = new MyThread();
...
thread.start();
...
// stop the thread
thread.stopThread();
thread.interrupt();
...
```

- To cover all bases one typically both sets the "stopped" variable to true AND call interrupt();

InterruptedException

- So now we understand the use of the InterruptedException exception
- When you write your own code, if you know that you will not interrupt your threads, then you typically do nothing when the exception is caught
 - e.g., when you do a sleep() and you have only one thread!
- But if you write code with others and are in charge of the code of one of many threads, it's always a good idea to deal with InterruptedException
- Let's now look at suspending/resuming threads

Suspending/Resuming Threads

- One uses the same technique to suspend/resume threads
- The thread has a volatile variable, say isSuspended, that the thread can check
- The thread periodically checks whether isSuspended is true
- If isSuspended is true, the thread blocks by calling wait()
- The thread can be unsuspended by setting isSuspended to false and calling notify()
- Let's see this on an example, which will use other things we've talked about so far

“Big” Example

- We wish to write a program that displays a disk and changes the disk's color in some fashion
- We wish to have a “Pause/Resume” button that controls the display
- One thread will be in charge of the display and will be paused/resumed when the button is clicked
- Let's first write the code without the button

MyPanel class

```
Terminal - vim - 54x24
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class MyPanel extends JPanel {
    private Color color;

    public MyPanel() {
        super();
        this.setPreferredSize(new Dimension(500,500));
    }

    public void setColor(float r, float g, float b) {
        color = new Color(r,g,b);
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(color);
        g.fillRect(500/8, 500/8, 500/2, 500/2);
    }
}
```

MyThread class

```
Terminal - vim - 58x30
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class MyThread extends Thread {
    MyPanel panel;

    public MyThread(MyPanel p) {
        panel = p;
    }

    public void run() {
        float g = (float)0.0;
        int sign = 1;
        while (true) {
            panel.setColor((float).1, g, (float).1);
            panel.repaint();
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                // do nothing
            }
            g += sign*(float)0.01;
            if (g > (float)1.0) { g = (float)1.0; sign = -1; }
            if (g < (float)0.0) { g = (float)0.0; sign = +1; }
        }
    }
}
```

Blinking class

let's look at it work

```
Terminal - vim - 44x30
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Blinking {
    private MyThread thread;

    public static void main(String args[]) {
        Blinking bd = new Blinking();
    }

    public Blinking() {
        JFrame frame = new JFrame();
        MyPanel panel = new MyPanel();

        frame.add(panel);
        frame.pack();
        frame.setVisible(true);

        thread = new MyThread(panel);
        thread.start();
        try {
            thread.join();
        } catch (InterruptedException e) {
            // do nothing
        }
    }
}
```

Adding the Button

- We need to do several things
 - Add the button to the display
 - Map the button to some action
 - Enable the thread to be interrupted and resumed
- Let's start with additions to the Blinking class

Modifications to Blinking

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Blinking implements ActionListener {

    private MyThread thread;
    private JButton button;
    private boolean isPaused = false;

    public static void main(String args[]) {
        Blinking bd = new Blinking();
    }

    public Blinking() {
        JFrame frame = new JFrame();
        frame.setLayout(new BorderLayout());
        MyPanel panel = new MyPanel();
        button = new JButton("Pause");
        button.addActionListener(this);

        frame.add(panel);
        frame.add(button);
        frame.pack();
        frame.setVisible(true);

        thread = new MyThread(panel);
        thread.start();
        try {
            thread.join();
        } catch (InterruptedException e) {
            // do nothing
        }
    }
}
```

Modifications to Blinking

all ok because
actionPerformed()
is executed by the
event dispatch thread

```
public void actionPerformed(ActionEvent event) {
    Component c = (Component)event.getSource();
    if (c == button) {
        if (isPaused) {
            thread.resumeThread();
            isPaused = false;
            button.setText("Pause");
        } else {
            thread.pauseThread();
            isPaused = true;
            button.setText("Resume");
        }
    }
}
```

Modif. to MyThread

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class MyThread extends Thread {
    MyPanel panel;
    private volatile boolean isPaused = false;

    public MyThread(MyPanel p) {
        panel = p;
    }

    public void pauseThread() {
        isPaused = true;
    }

    public void resumeThread() {
        isPaused = false;
        synchronized(this) { this.notifyAll(); }
    }

    private synchronized void checkPaused() {
        while(isPaused) {
            try {
                this.wait();
            } catch (InterruptedException e) {} // do nothing
        }
    }
}
```

Modif. to MyThread

```
public void run() {
    float g = (float)0.0;
    int sign = 1;
    while (true) {
        panel.setColor((float).1, g, (float).1);
        checkPaused();
        panel.repaint();
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            // do nothing
        }
        g += sign*(float)0.01;
        if (g > (float)1.0) { g = (float)1.0; sign = -1; }
        if (g < (float)0.0) { g = (float)0.0; sign = +1; }
    }
}
```

Conclusion

- Concurrency in Java is at the same time simple ...
 - Monitors are clean and easy to use
- and complicated
 - You can create objects just for using their monitors, i.e., creating a bunch of locks
 - Volatile
 - Stopping/Resuming
 - And the always fun invokeLater() thing we saw at the beginning of the semester
- At this point, we know everything we need to know (and more than most programmers) to write good concurrent Java code using **basic mechanisms**
 - A future lecture will talk about java.util.concurrent