Introduction to Java Threads

4003-243
What is a Process?

- Here’s what happens when you run this Java program and launch 3 instances while monitoring with `top`.
- On a single CPU architecture, the operating system manages how processes share CPU time.

```java
public class MyProgram {
    public static void main(String args[]) {
        int i = 0;
        while (true) {
            i = i + 1;
        }
    }
}
```
What is a Process?

• Besides running your program, the Java interpreter process must do other tasks
  − Example: manage memory for your code, including garbage collection

• How does the interpreter perform multiple tasks within a single process?

  *threads*
What is a Thread?

- A **thread** is a flow of execution
- Java has built-in **multithreading**
  - Multiple tasks run concurrently in 1 process
- Multiple processes and threads share CPU time
Java Threads

• When a program executes, the JVM starts a thread to run `main()`
  - This is the “main thread” of execution

• Each thread is an instance of a class that:
  – EITHER Extends the `Thread` class,
  – OR Implements the `Runnable` interface

• The new object is a **Runnable object**
  – Another name used is **active object**
Creating Threads by Extending Thread

• An example class that extends the Thread

```java
// Custom thread class
class MyThread extends Thread {
    public MyThread(...) {
        ...
    }
    public void run() {
        // Run the thread
        ...
    }
}
```

```java
// Client class
class Client {
    public void method(...) {
        // Create thread1
        MyThread t1 = new MyThread(...);
        t1.start();
        // Create thread2
        MyThread t2 = new MyThread(...);
        t2.start();
    }
}
```
Creating Threads by Extending Thread

• A program to create and run 3 threads
  – First thread prints the letter a 5000 times
  – Second thread prints the letter b 5000 times
  – Third thread prints integers 1 through 5000

• Make one thread class to handle the first two threads, PrintChar

• The third thread will be implemented by the PrintNum class

• See TestThread/TestThread.java
Creating Threads by Implementing Runnable

- An active class may implement the Runnable interface

```java
// Custom thread class
public class MyThread implements Runnable {
    public MyThread(...) {
        ...
    }

    // Override the run method
    // in Thread
    public void run() {
        // Run the thread
        ...
    }
}

// Client class
public class Client {
    public void method(...) {
        // Create thread1
        Thread t1 = new Thread (new MyThread(...));

        // Start thread1
        t1.start();
    }

    public void method(...) {
        // Create thread2
        Thread t2 = new Thread (new MyThread(...));

        // Start thread2
        t2.start();
    }
}
```
Creating Threads by Implementing Runnable

- If a runnable class already inherits from a specific superclass, it must implement the Runnable interface
  - The interface specifies one method: `run()`
- Wrap the runnable class in a Thread object to use it

```java
Thread t1 = new Thread( new PrintChar( 'a', 5000 ) );
Thread t2 = new Thread( new PrintChar( 'b', 5000 ) );
Thread t3 = new Thread( new PrintInt( 5000 ) );
```

- See TestRunnable/TestRunnable.java
Thread: Selected Methods

**java.lang.Runnable**

**java.lang.Thread**

- Thread()
- Thread(target: Runnable)
- run(): void
- start(): void
- interrupt(): void
- isAlive(): boolean
- setPriority(p: int): void
- join(): void
- sleep(millis: long): void
- yield(): void
- isInterrupted(): boolean
- currentThread(): Thread

*Underlined methods are static*
Thread Control - \textit{yield}

- Use \textit{yield()} to release the CPU
- See TestYield/TestYield.java
Thread Control - sleep

- **sleep()** pauses thread execution for a specified number of milliseconds
- See TestSleep/TestSleep.java

```
sleep( 500 )
```

Thread #1

```
ZzZz...waiting...ZzZz...
```

Thread #2

```
T2 executes
```

Time's up. T1 wakes and runs

```
T2 stops executing
```

T1
Thread Control - \texttt{join}

- \texttt{join()} makes one thread wait for another thread to finish
- See TestJoin/TestJoin.java
Thread States

• The JVM manages thread scheduling and execution using thread states
• **Runnable** is the only state in which the thread *may* be executing on the CPU
  - On a single core/processor system, only one thread is actually running at any moment
• Other thread states record the situation of a thread with respect to its execution
• `t.isAlive()` returns **true** if the thread is **not** in the **New** or **Terminated** state
Java Thread States

New

Runnable

Blocked

Terminated

Thread tries to access a locked, synchronized block

Block becomes unlocked

*NOTE: Running is not an actual Thread.State enum value. It represents the thread currently executing.
**isAlive()**

```java
public class WorkerThread extends Thread {
    private int result = 0;

    public void run() {
        // Perform a complicated, time-consuming calculation
        // and store the answer in the variable result
    }

    public static void main(String args[]) {
        WorkerThread t = new WorkerThread();
        t.start();

        while (t.isAlive()) {
            System.out.println(result);
        }
    }
}
```

- This solution works, but there is a better way!
Introduction to Threads

Coordinating Thread Execution
Threads and Methods

- If multiple threads run inside a program, they might call the same method.
- If they call the same method at the same time, what happens? The threads could:
  - Read different results from member fields
  - Overwrite field values with incorrect values
- To make sure only one thread executes inside a method, you must lock the object.
  - The `synchronized` keyword, when applied to a method, locks the object instance.
Resource Conflict Scenario

• Example Situation:
  - A program launches 100 threads, each of which adds a penny to an account
  - Assume the account is initially empty

• What might happen?

```
java.lang.Thread

AddAPenny
+ run() : void

100

100

AccountConflict
- bank : Account
  - Thread : Thread[]
+ main(args : String[]) : void

Account
+ balance : int
+ getBalance() : int
+ deposit(amt : int) : void
```
Resource Conflict Scenario

- One thread could update the value of the common, shared balance before other(s) complete their update individually
  - This is called a **race condition**
  - A class is **thread-safe** if it does not cause a race condition in the presence of multiple threads

<table>
<thead>
<tr>
<th>Step</th>
<th>Balance</th>
<th>Thread[ 0 ]</th>
<th>Thread[ 1 ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>In execution</td>
<td>Actions</td>
<td>Actions</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>newBalance = balance+1</td>
<td>newBalance = balance+1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>balance = newBalance</td>
<td>balance = newBalance</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>balance = newBalance</td>
<td>balance = newBalance</td>
</tr>
</tbody>
</table>
Critical Region

- To avoid race conditions, program code must prevent more than one thread from executing in a **critical region**
  - Code in this region may change *shared state*
- Only one thread must be allowed to enter the `deposit` method at a time

Risky, unsynchronized version of Account.deposit():

```java
public void deposit(int amount) {
    int newBalance = balance + amount;
    balance = newBalance;
}
```
Synchronizing Instance Methods

To synchronize an instance method means a thread must get a lock on the instance before beginning to execute it.

- Makes deposit thread-safe in Account

```java
class AddAPenny implements Runnable {
    public void run() {
        account.deposit(1);
    }
}

public class Account {
 ...
    public synchronized void deposit(int amount) {
        ...
    }
}
```

Synchronize on the account object
Synchronizing Instance Methods

• The scenario with synchronization:
  - Thread 0 gets there first; thread 1 blocks

Thread[0]

Acquire lock on account

Enter deposit

Release the lock

balance = 1

Thread[1]

Blocks trying to acquire lock on account

Thread[1] cannot execute here

Acquires lock on account

Enter deposit

Release the lock

balance = 2
Thread Synchronization

• These programs illustrate and solve the resource conflict problem:
  – AccountConflict.java
    • 100 threads add a penny to one account
    • Lack of synchronization loses money
  – AccountSync.java
    • 100 threads add a penny without error
    • A synchronized method keeps threads from corrupting the state of the Account
      – See Synchronization/*.java
Synchronized Statements

- A **synchronized block** can lock inside a portion of a method
  - This may shorten lock time to increase concurrency and improve performance
  - Smaller critical region → better performance

In AccountSync.java

```java
class Account {
    public void deposit( int amt ) {
        synchronized( this ) { 
            int newBalance = balance + amount;
            Balance = newBalance;
        }
    }
}
```
Contestion & Starvation

• A situation known as **contention** may occur when threads compete for execution time on the CPU
  – A thread might not give other threads a chance to run
    • That may lead to **starvation**

• What's needed are ways to make threads coordinate and cooperate with each other
Thread Cooperation – wait(), notify() and notifyAll()

- Three `Object` methods support coordination among active threads
  - `Object.wait()` blocks a thread
  - `Object.notify()` 'wakes up' a thread waiting on some condition
  - `Object.notifyAll()` wakes up all threads waiting on some condition

- These methods must be called inside a `synchronized` method or block that has locked the object instance
Thread Cooperation – wait(), notify() and notifyAll()

- `public final void wait()` throws `InterruptedException`
  - Forces the thread to wait until a `notify` or `notifyAll` method is called on the object
  - If notify never happens, thread blocks forever
- `public final void notify()`
  - Awakens one thread waiting on the object
  - Which one wakens is implementation dependent
- `public final void notifyAll()`
  - Wakes all threads waiting on the object
  - Scheduling decides which gets notified first
Thread Cooperation – \(\text{wait}(), \text{notify}()\) and \text{notifyAll}()\

Threads running within one process

- **Thread #1**
  - \(\text{obj.wait}()\)
  - T1 notified and resumes

- **Thread #2**
  - T2
  - \(\text{obj.notify}()\)

- **Thread #3**
  - T3
  - \(\text{obj.wait}()\)
  - T3 is still waiting...
Threads and Interruptions

- A thread may be interrupted
  - If the thread is not alive, there's no effect
  - If the thread is in the **Runnable** state:
    - Set its interrupted status and move it to **Blocked**
  - If the thread is **Blocked**:
    - Interrupt clears the thread's interrupted status
    - Throws an **InterruptedException**
      - The thread's Blocked state has been interrupted
- Result: `wait()`, `sleep()` and `join()` calls require `try {...} catch (InterruptedException exc) syntax`
Thread Cooperation

- Synchronization ensures mutual exclusion in critical regions to avoid race conditions.
- Threads also need a way to cooperate without requiring threads to finish:
  - The wait/notify mechanisms can coordinate the execution of multiple threads.
- Calls to `wait()`, `notify()` and `notifyAll()` must be in a `synchronized` method or block:
  - Otherwise: `IllegalMonitorStateException`
Two kinds of thread synchronization are:
- **Mutual exclusion** uses `synchronized` to keep threads from interfering with one another when sharing data
- **Cooperation** uses `wait` and `notify` to allow threads to safely coordinate and share data

The structure that Java uses to support synchronization is called a **monitor**
- A **monitor** controls access to data and coordinates multiple threads' use of data
- A **room** where 1 thread executes at a time
Monitor Structure

• The monitor structure combines synchronized, wait and notify to coordinate

Thread 1 executing `aMonitor.methodA`

```java
synchronized methodA(){
    while ( condition ) {
        try {
            // Wait for condition
            this.wait();
        }
        catch ((InterruptedException ex)){
            ex.printStackTrace();
        }
    }
    // Do something critical…
}
```

Resumes

Thread 2 executing `aMonitor.methodB`

```java
synchronized methodB(){
    // Do things…

    // When condition is false, then
    this.notify();
}
```

Or `this.notifyAll()` to wake up all threads waiting on this `aMonitor`
Producer-Consumer

- The **producer-consumer** threading problem is a *classic* in operating systems
  - The **producer** creates new elements and sends them to one or more consumers
  - A **consumer** takes an element out of the collection, when available, and uses it

- Carefully design a solution to **avoid**:
  - Producer sending items that consumers miss
  - Consumer getting the same item many times
**Producer-Consumer**

- See ProducerConsumer1/ ProducerConsumer2/ and ProducerConsumer3/

### General structure

<table>
<thead>
<tr>
<th><strong>Producer</strong></th>
<th><strong>Consumer</strong></th>
</tr>
</thead>
</table>
| - cubbyHole : CubbyHole  
- number : int  
| - cubbyHole : CubbyHole  
- number : int  |
| + run() : void  |
| + run() : void  |

<table>
<thead>
<tr>
<th><strong>CubbyHole</strong></th>
<th><strong>ProducerConsumerTest</strong></th>
</tr>
</thead>
</table>
| - contents : int  
- available : boolean  
| + main(args : String[]) : void  
| + get() : int  
+ put() : int  

- 4/27/12
Consumer Waiting Behavior

In this sequence, the consumer must wait.

1.0  int= get()

1.1 [available == false ] wait

1.2 put(value)

1.3 [available == false ] notifyAll

1.4  int= get()

1.5 notifyAll

1.6 [available == true ] value

When the producer tries to put when the cubbyhole is ‘not full’ -- i.e. there is space to put more.

When the consumer tries to get when nothing is available, the cubbyhole monitor makes the consumer's thread wait.

Notification from the producer's put operation finally causes the consumer to come out of its wait and get the newly available data value. This is NOT A SECOND get() call; the thread was blocked waiting for something to notify the (consumer) thread when there is a new value available.
Producer Waiting Behavior

In this sequence, the producer must wait.

Notification from the consumer's get operation finally causes the producer to come out of its wait and put another data value into the cubby. This is NOT A SECOND put() call; the thread was blocked waiting for something to notify the (producer) thread when the cubby can accept the put of a new value.

A consumer eventually comes along to get what's in the cubby and notify any waiting threads after changing the cubby's available state to false.
Introduction to Threads

Threads and Swing

When an event occurs, how does the listener code execute?

Who is executing these GUI event handlers?

The short answer is: Threads
AWT Event Thread

- All GUI tasks are handled by a single thread: the **AWT event dispatcher**
- The code in a listener actually executes *within* the event dispatching thread

```
AWT event thread

actionPerformed(ActionEvent e)

draw a JButton
```
AWT Event Thread

- What code should the AWT thread run?
  - It is good to do things in an AWT handler that cause *GUI components* to be redrawn.
AWT Event Thread

- What if another thread has GUI work?
  - Example: update a component
- Need: schedule GUI work on the main event thread or risk erroneous behavior

```
AWT event thread

paint a button component

// ok

user thread

update a JTextArea

Danger: update outside the AWT event loop
```
AWT GUI Event Processing vs. User Processing

- A listener responds to events generated by a thread **not running on the main event dispatching thread**
  - AWT event dispatcher may conflict with a user thread

Danger: update outside AWT event loop
SwingUtilities.invokeLater()

- To schedule a GUI interaction for the event thread, use the non-blocking call:
  SwingUtilities.invokeLater(Runnable task)
- The Runnable object’s run() method performs the GUI task you want done
- The invokeLater method queues the task object for execution later
  – The AWT event thread will then execute it

- See javax.swing.SwingUtilities
SwingUtilities.invokeLater()

- SwingUtilities.invokeLater() is a **non-blocking call**
- SwingUtilities.invokeLater() **blocks current thread**

```java
SwingUtilities.invokeLater(updateTask)
```

// updates JList from right place at right time
Nested and Anonymous Classes

```java
class DoIt implements Runnable {
    public void run() {
        inputDoc.remove(0, inputDoc.getLength());
    }
}
SwingUtilities.invokeLater(new DoIt());
```

Or an anonymous Runnable class can update

```java
SwingUtilities.invokeLater(new Runnable() { // a class and object with no name...
    public void run() {
        ... 
        inputDoc.remove(0, inputDoc.getLength());
        ...
    }
}); // end of invokeLater() call
```
Nested and Anonymous Classes

A nested class becomes a separate byte code file as:

```
OuterClassName$InnerClassName.class
```

An anonymous class file becomes a numbered .class file:

```
OuterClassName$1.class
OuterClassName$2.class
```
Exercise

• What happens with this code fragment?
  – Will myThread ever run?
  – How could you make sure myThread runs?

```java
public class ThreadQuestion {
    public static void main( String args[] ) {
        new Thread( "myThread" ) {
            public void run() {
                System.out.println( "I’m running!" );
            }
        }.start();
    }
}
```