Lecture 2
Introduction/Review of Concurrency
Running a Sequential Program

- Executable
  *Machine instructions to be performed*
- Program counter
  *Next instruction to be executed*
- Stack
  *Current variable definitions*
- Heap
  *Dynamically allocated data structures*
- Control flow
  *Sequence of instructions performed during an execution*
Java Memory Model

- **Stack**
  - Local variables
  - Method parameters

- **Heap**
  - Objects!
  - Every call to `new` allocates space on heap

- Class-typed variables reference heap or null
Concurrent Programs

• Multiple control flows!
• Programs with multiple control flows can be
  – Concurrent
  – Parallel
  – Distributed
• Control flows are either
  – Processes
  – Threads
Concurrent vs. Parallel vs. Distributed

• Concurrent
  \# of control flows unrelated to \# of physical processors

• Parallel
  \# of control flows ≤ \# of physical processors; each flow has its own processor

• Distributed
  Multiple machines connected via network
Processes vs. Threads

- **Processes**
  - Possess own heap
  - Communicate via *IPC* (= inter-process communication mechanisms)
    - Sockets
    - Message passing
    - Etc.
- **Threads**
  - Contained within processes
  - Possess own stack, program counter
  - Share heap with other threads in same process
  - Communicate via shared memory
- **Historically**
  - Process management handled by operating system
  - Processes were single-threaded
(Single-Threaded) Processes

Process 1

- Thread
- Memory
- Heap
- Stack
- IPC

Process 2

- Thread
- Memory
- Heap
- Stack
- IPC
Multi-threaded Process
Scheduling

• Execution requires processor
  – Running a thread requires using a processor
• What decides which thread gets which processor?
  – **Scheduler** (part of JVM and OS)
    • Scheduling policy decides which threads run where, and when
  – **Pre-emptive** schedulers can interrupt one thread and let another run on a given processor
    • Interrupted thread is “suspended”: its stack, program counter are saved so that thread can be re-activated later
    • Stack, program of new thread are loaded and new thread activated
    • This is called a **context switch**
Threads, Processes and Processors

• Do processes run on single machine? Yes
• Do processes run on a single processor? Not necessarily
  – Different threads can run on different processors
  – Scheduler makes this decision
• Do threads run on a single processor?
  – Usually
  – Some schedulers support thread migration (why?)
A Reference Model for Distributed / Parallel / Concurrent Programs
Example thread scheduling (1)

One process and all its threads on a single CPU
Example thread scheduling (2)

Threads of a process allowed to run on either CPU

CPU 1

- p1
- p2

CPU 2

- p1
- p2

p2 threads:  p1 threads:
Language Support for Concurrency

• Many languages support concurrency!
  C, C++, C#, OCaml, Java, SmallTalk, Python, ...

• Traditionally: process / thread management handled via system calls to operating system
  – Not part of core language (e.g. C)
  – Platform-specific, non-portable, since different OS’s have different mechanisms

• Modern languages (e.g. Java) include mechanisms for thread management directly
Java Concurrency

• Support for multi-threading, processes
  – Process = running instance of Java Virtual Machine
  – Objects live on heap, can be shared by threads in same process

• Every Java program starts with one thread running `main`

• This course: focus is on thread programming
Java Concurrency: Threads Are Objects

• Object class is `Thread`, which is part of `java.lang` package (automatically imported!)

• Thread objects include:
  – `public void run()` executed when thread runs
  – `public void start()` to launch the thread
    • Other methods that we will study later
  – Two main constructors (more later):
    • `Thread()` create a thread (used by subclasses)
    • `Thread(Runnable r)` create a thread whose run method will execute `r.run`

• **Homework**: read javadoc
  – [http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html)
Thread Creation in Java

• Create an object `t` in class `Thread` with desired functionality in `run()` method
• Invoke `t.start()`
• This starts a thread that runs the `t.run()` method!
“Desired Functionality in run()”?

• Two approaches
  – **Subclassing from** Thread
  – **Implementing** Runnable interface

• In the former: **override** run()

• In the second
  – Define a class implementing the Runnable interface
  – Use relevant constructor in Thread on objects in this class
    
    Thread (Runnable target)
    Thread (Runnable target, String name)
Thread Implementation via Subclassing (Inheritance)

```java
public class HelloWorldThread extends Thread {
    public void run () {
        System.out.println ("Thread says Hello World!");
    }
}
```

**New class** `HelloWorldThread` is introduced

- Extends `Thread` class
- Uses overriding to redefine `run()` method to do what we want
Thread Implementation via Runnable

```java
public class HelloWorldRunnable implements Runnable {
    public void run () {
        System.out.println("Runnable says Hello World!");
    }
}
```

- **Runnable** is an interface in java.lang containing only:
  ```java
  public void void run ()
  ```

- **This class implements** **Runnable** by providing each object with a `run()` method

- **Constructor for** **Thread** class can now be called with objects in this class
Thread Creation

Thread h1 = new HelloWorldThread ();
Thread h2 = new Thread (new HelloWorldRunnable ());
h1.start ();
h2.start ();

• h1 is thread object created from subclass of Thread
• h2 is thread object created from Runnable object
• Output is two instances of “Hello World!”
# Subclassing or Runnable?

<table>
<thead>
<tr>
<th></th>
<th>Subclassing</th>
<th>Runnable</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PROS</strong></td>
<td>• Can call Thread methods when via <code>this</code> implementing <code>run()</code></td>
<td>• Can inherit from another class besides <code>Thread</code> when creating <code>Runnable</code> object</td>
</tr>
<tr>
<td></td>
<td>• No need for creating intermediate object</td>
<td>• Protects other <code>Thread</code> methods (e.g. <code>start()</code>)</td>
</tr>
<tr>
<td><strong>CONS</strong></td>
<td>• Cannot inherit from another class</td>
<td>• Must call non-static <code>Thread</code> methods via <code>Thread.currentThread()</code> inside <code>run()</code></td>
</tr>
<tr>
<td></td>
<td>• Danger of overriding other methods in <code>Thread</code> class (e.g. <code>start()</code>)</td>
<td>• Must create intermediate <code>Runnable</code> object in order to create <code>Thread</code></td>
</tr>
</tbody>
</table>

9/9/2013 ©2012, 2013 University of Maryland
In truth

• Shouldn’t make Thread objects directly!
• Instead, try to use factory methods (e.g., from java.util.concurrent) to which you pass Runnable objects
  – This is a task-oriented view, as opposed to a thread-oriented view
  – Allows details of parallel resources to be handled by libraries/run-time: more scalable
currentThread ()

static Thread currentThread ()
  – Returns thread of current execution
  – Useful when implementing thread operations via Runnable, as you can get access to thread info at runtime
Thread States

• What happens if we do the following?
  Thread h1 = new HelloWorldThread ();
  h1.start ();
  h1.start ();
  h1.start ();

• Answer
  Exception in thread "main"
  java.lang.IllegalThreadStateException

• What?
  – Not every method is legal on every Thread object
  – The state of the object determines this validity
  – In this case, you cannot start a thread that has already been started
Thread States

- Accessible via method `Thread.State getState()`
- `Thread.State` is an enumerated type recording state of thread object
  - NEW
    A thread that has not yet started is in this state.
  - RUNNABLE
    A thread executing in the Java virtual machine is in this state.
  - BLOCKED
    A thread that is blocked waiting for a monitor lock is in this state.
  - WAITING
    A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
  - TIMED_WAITING
    A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
  - TERMINATED
    A thread that has exited is in this state.

[Quoted from http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html]
Thread State Example Revisited

Thread h1 = new HelloWorldThread (); // state is NEW
h1.start (); // state is RUNNABLE
h1.start (); // Error!

• **When h1 is created, its state is NEW**
• **After h1.start () is called, the state is RUNNABLE**
• **h1.start () can only be called when state is NEW!**
More on Thread States

• Some `Thread` methods (e.g. `start`) only applicable when object is in correct state
• The states `NEW`, `RUNNABLE`, `TERMINATED` are probably easiest to understand
• We will learn about the states `BLOCKED`, `WAITING`, `TIMED_WAITING` later
Other Thread State Methods

• **boolean isAlive()**
  – Returns *true* if thread has been started but is not terminated
  – \( t \text{.isAlive}() \) equivalent to
    \[
    (t \text{.getState}() \neq \text{NEW}) \&\& (t \text{.getState}() \neq \text{TERMINATED})
    \]

• **void join()**
  – Blocks until thread terminates, then terminates
  – \( t \text{.join}() \) very similar to
    \[
    \text{while } (t \text{.isAlive}()) \{ \}
    \]

• **void join (int millis)**
  Like \( t \text{.join}() \) except that if \( t \) has not terminated in millis milliseconds, then \( t \text{.join(millis)} \) nevertheless terminates
Thread death

• Originally, Java allowed one thread to kill/pause another thread
  – Via thread.stop, thread.destroy, thread.stop, thread.suspend/resume
• No longer allowed!
  – These methods are deprecated because they are inherently unsafe: could cause races or deadlocks
Threads and Process Termination

- A process (JVM) terminates when “there is nothing left that has to be done”
- When does this hold?
  - When the main thread terminates?
  - When all threads terminate?
  - When “the important” threads terminate?
- Java answer: when all user threads terminate
User Threads vs. Daemon Threads

• In Java, every thread object is by default a *user thread*

• A Java process can terminate if and only if all user threads (including, but not only, main) have terminated

• Threads may be changed to *daemon threads* using method `setDaemon (boolean on)`
  – If the only nonterminated threads are daemons, then the JVM will terminate
  – Daemon threads should only be used for “background work” needed while “useful” computation is being performed (e.g. updating status bars, etc.)

• `setDaemon ()` is only valid if thread state is `NEW`; otherwise, `IllegalThreadStateException` thrown
More on Thread Termination

• When a thread object terminates, the object still remains!
  – Thread state is TERMINATED ...
  – ... but object still exists
Methods for Interacting with Scheduler

- **void setPriority** (int newPriority)
  Set priority to given value (must be between MIN_PRIORITY and MAX_PRIORITY: see below)
- **int getPriority** ()
  Return priority value
- **static void yield** ()
  “Hint” to scheduler that thread can give up processor
- **static void sleep** (int millis)
  Block for millis milliseconds
- **static int MIN_PRIORITY**
  Smallest (lowest) priority
- **static int MAX_PRIORITY**
  Largest (highest) priority
- **static int NORM_PRIORITY**
  Default priority
InterruptedException

• (More on this later, but for now:)
• Thrown by some Thread methods (e.g. sleep())
  – Raised when a method is interrupted while sleeping
  – We will see about interruptions later
• When you call such a method, you must either
  – Catch the exception, e.g.
    try { ... sleep (1000);...}  
catch (InterruptedException e) { ... }
  – ... or include a throws directive in your method declaration, e.g.
    public void myMethod throws InterruptedException (...){...}