Lecture 2
Introduction to 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

Process 2

Thread

Memory

Heap

Stack

IPC
Multi-threaded Process
Running a Multi-Process / Multi-Threaded Application

- Execution requires processor
- Running a thread requires using a processor
- What decides which thread gets which processor?
  - Scheduler (part of operating system)!
  - Scheduling policy decides which threads run 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 \textit{context switch}
Threads, Processes and Processors

• Do processes run on a 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
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 has at least one thread: main

• This course: focus is on thread programming
Java 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 is launched
  - `public void start ()` to launch the thread
  - Other methods that we will study later
  - Constructors, of which more later, but here are two:
    - `Thread()` create a thread
    - `Thread(String name)` create a thread with the given name
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 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>• Easy access to Thread methods when implementing <code>run()</code></td>
<td>• Can inherit from another class besides Thread when creating Runnable object</td>
</tr>
<tr>
<td></td>
<td>• No need for creating intermediate object</td>
<td>• Protects other Thread methods (e.g. <code>start()</code>)</td>
</tr>
<tr>
<td><strong>CONS</strong></td>
<td>• Cannot inherit from another class</td>
<td>• Harder to access non-static Thread methods when defining Runnable objects</td>
</tr>
<tr>
<td></td>
<td>• Danger of overriding other methods in Thread class (e.g. <code>start()</code>)</td>
<td>• Must create intermediate Runnable object in order to create Thread</td>
</tr>
</tbody>
</table>
Thread States

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

• Answer
  
  ```java
  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/6/docs/api/java/lang/Thread.State.html]
Thread State Example Revisited

Thread \( h_1 \) = new HelloWorldThread (); // state is NEW
\( h_1 \).start (); // state is RUNNABLE
\( h_1 \).start (); // Error!

- When \( h_1 \) is created, its state is NEW
- After \( h_1 \).start () is called, the state is RUNNABLE
- \( h_1 \).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.isAlive()` equivalent to
    
    ```java
    (t.getState() != NEW) && (t.getState() != TERMINATED)
    ```

- **void join()**
  - Blocks until thread terminates, then terminates
  - `t.join()` very similar to
    ```java
    while (t.isAlive ()) { }
    ```

- **void join (int millis)**
  - Like `t.join()` except that if `t` has not terminated in `millis` milliseconds, then `t.join(millis)` nevertheless terminates
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
Thread Execution

• Once threads are started, what determines when they are eligible for execution?
• Answer: scheduler!
  – Operating system routine responsible for allocating processor time to threads
  – If there are more processors than threads, could allocate each thread to its own processor
  – If there are more threads than processors, time-slicing may be needed to *interleave* access to processors
    • Each thread executes for a while, then is pre-empted
    • Exact scheme also takes account of priorities, whether or not threads are blocked
    • What if thread is in the middle of something “atomic”?
Scheduling Example (1)

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

Threads of a process allowed to run on either CPU

<table>
<thead>
<tr>
<th>CPU 1</th>
<th>p1 threads:</th>
<th>CPU 2</th>
<th>p1 threads:</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td></td>
<td>p1</td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td></td>
<td>p2</td>
<td></td>
</tr>
</tbody>
</table>

p2 threads: | p1 threads: |
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

• **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.
    ```java
    try { ... sleep (1000);}...
    catch (InterruptedException e) { ... }
    ```
  – ... or include a **throws** directive in your method declaration, e.g.
    ```java
    public void myMethod throws InterruptedException (...)
    ```

©2012 University of Maryland
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