Multi-threading

19 Mar 2010
CMPT166
Dr. Sean Ho
Trinity Western University

Outline for today

- Leftover from last time: UDP send/receive
- Threads
  - States threads can be in
  - Tasks vs. threads
  - In Java: Runnable, Thread
  - Anonymous objects and classes
  - In Swing
- Dividing up the work: managing threads

Receiving a UDP packet

- Create a DatagramSocket (in java.net):
  - **sock = new DatagramSocket(*port*);**
- Create a DatagramPacket to store the data:
  - **byte payload[] = new byte[100];**
  - **packet = new DatagramPacket(payload, payload.length);**
- Wait (block) for a packet:
  - **sock.receive(packet);**
- Read info from packet:
  - **packet.getData(), .getLength(), .getAddress(), .getPort()**

Sending a UDP packet

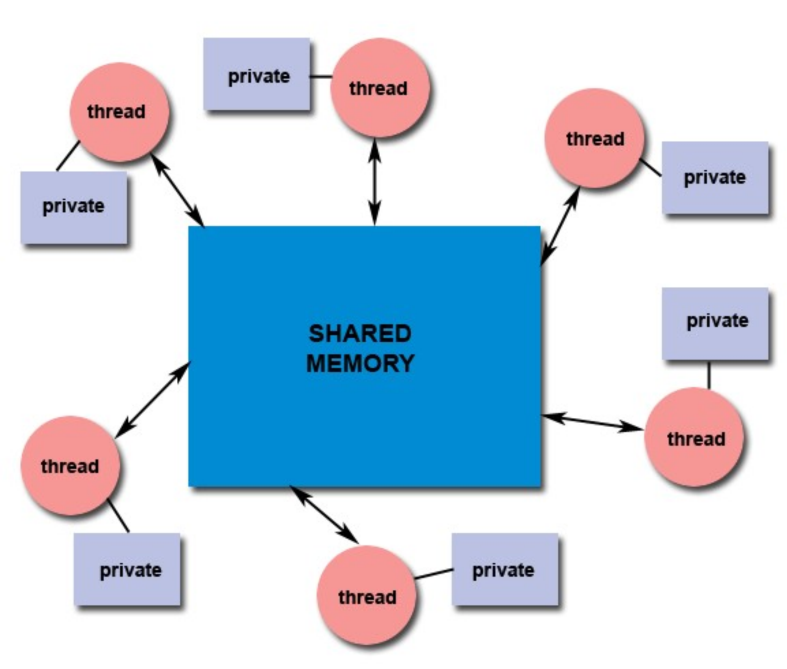
- Prepare payload:
  - **String msg = “Hello, World!”;**
  - **byte[] payload = msg.getBytes();**
- Package payload:
  - **packet = new DatagramPacket(**
    payload, payload.length,
    ***hostname*, *port*);**
- Send packet:
  - **socket.send(packet);**

Multithreading

- Concurrency is running multiple tasks at the same time
  - Downloading a file, watching a movie, checking email
  - One server talking to multiple clients
- Threads are individual tasks (objects) that may run concurrently
- Multithreading is built-in to Java ≥1.5

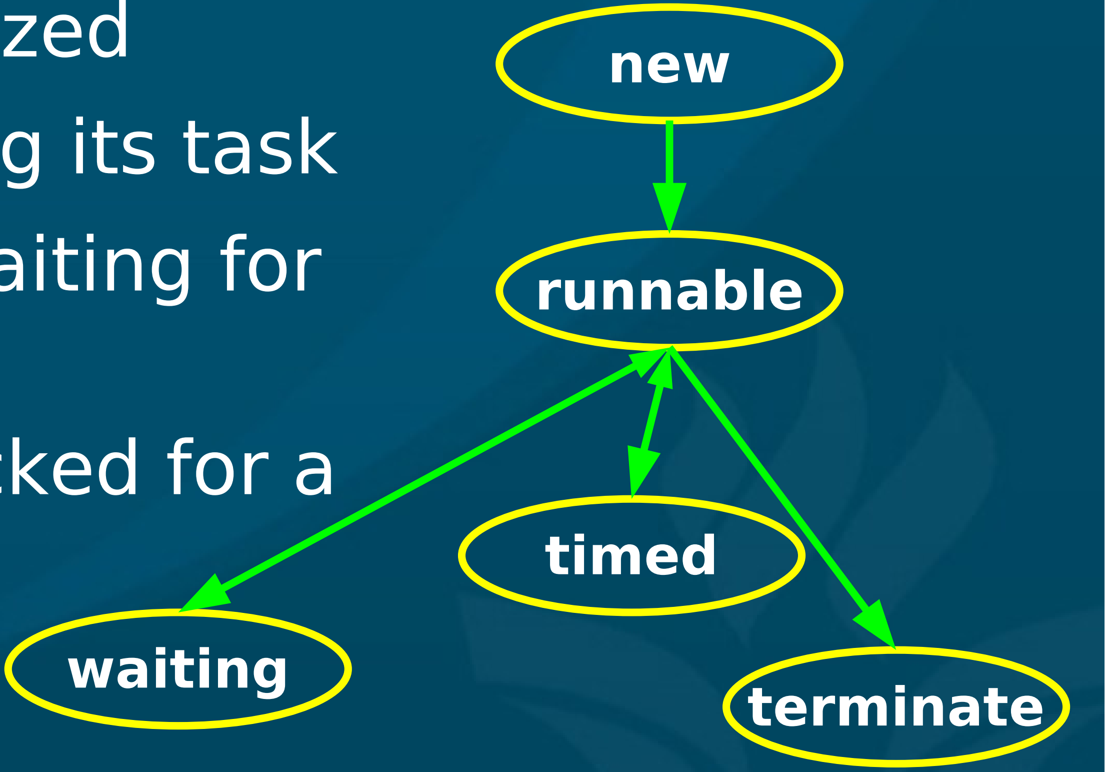
Thread model of parallelism

- Threads are lightweight processes
- Threads allow concurrency
  - Make use of multiple processors
  - But still useful even on uniprocessor
- Threads use shared memory
  - Synchronization issues for shared objects
    - Thread-safe code?
  - May also have local (private) variables



Thread state diagram

- Threads can be in one of five states:
  - New: not yet initialized
  - Runnable: executing its task
  - Waiting: blocked waiting for another thread
  - Timed waiting: blocked for a fixed time
  - Terminated



Task scheduling

- Create as many threads as you like
- But # of processors limits # of running threads
    - Multi-core; Hyper-threading
- Scheduler assigns runnable threads to processors
    - Part of operating system, not Java VM
    - Scheduler can preempt running threads to allow others to run
    - Each thread has a priority ("nice" value)
        - Lower priority threads might get starved

Tasks vs. threads

- Distinction between a task and a thread:
- Task is work that needs to be done
  - in Java: the Runnable interface
- Thread is a process that can perform the work
  - in Java: the Thread class
- Define the tasks as run() methods in classes
- Create threads by instantiating Thread (or subclasses of it)
  - Assign a Runnable task to the thread

Threads in Java: Runnable

- Define a class with the Runnable interface
  - **class NumCruncher implements Runnable**
  - Define (override) the method run():
  - **public void run() { … }**
- Create an instance of Thread that uses an instance of your class:
  - **Thread crunch =**
    new Thread(new NumCruncher());
- Start the thread:
  - **crunch.start();**
- No imports needed: all in java.lang

The Thread class

- Thread implements Runnable, so you may also subclass Thread:
  - **class NumCruncher extends Thread {**
    public void run() { … }
- Then just call start() directly on your object:
  - **NumCruncher cr = new NumCruncher();**
  - **cr.start();**
- Runnable is the interface; Thread is a class
- The Thread class also has static utility methods:
  - **Thread.sleep(100);** **// wait for 100ms**

Example: PrintTask

```
import java.util.Random;
class PrintTask implements Runnable {
    private int sleepTime;
    private String name;
    private static Random gen = new Random();
    public PrintTask( String name ) {
        this.name = name;
        this.sleepTime = gen.nextInt( 5000 );
    }
    public void run() {
        System.out.println( name + “: good night!” );
        Thread.sleep( sleepTime );
        System.out.println( name  + “: good morning!” );
    }
}
```

Short-hand: anonymous

- Instantiate a thread and start it in one line:

  -
    ```
    (new NumCruncher()).start();
    ```
  - The instance is an anonymous object
- Even shorter: use an anonymous class

  -
    ```
    (new Thread() {
            public void run() { … }
       }).start();
    ```
  - Defines an anonymous subclass of Thread
    - Inner class (defined within enclosing class)
  - Creates an anonymous instance of it
  - Starts the thread object

Example: starting a Swing app

- We've used anonymous classes before as a thread-safe way of initializing a Swing GUI:

  -
    ```
    public static void main( String[] args ) {
      SwingUtilities.invokeLater(
          new Runnable() {
              public void run() {
                  new Histogram();
              }
          } );
    ```
- invokeLater() runs the task on a thread designated for interaction with the Swing GUI

Multithreading for Swing

- If an event handler (e.g., ActionListener) is very slow, the whole GUI is blocked waiting for it
- So create worker threads for these callbacks
- Use inner class for access to private attributes
- Use anonymous class for one-off tasks:

```
public void ActionPerformed() {
    (new Thread() {
        public void run() {
            /* do long operation, e.g. network */
        }
    }).start();
}
```

Warning: shared objects

- Swing programs have multiple threads:
  - Init thread (main() setup before GUI)
  - Event dispatch thread (interacts w/GUI)
  - Any worker threads you create
- If multiple threads try to modify a shared object, errors may occur!
- Only the event dispatch thread should access the GUI (change widget text, etc.)
  - Worker threads may ask the event dispatch thread to update the GUI
- More details in Swing tutorial: concurrency

How to divide up the work?

- Master/worker:
  master thread assigns work to worker threads
  - Master typically handles UI, input
  - Static or dynamic worker pool
- Coworkers: all threads are peers:
  - Main thread participates in doing work
- Pipeline: each thread works on a different part of the task: e.g., automobile assembly line
  - Function parallelism vs. data parallelism