Threads and Synchronization

(thanks to Doug Lea for some slides)

Overview

• What are threads?
  – Concept
  – Basic Java mechanisms

• Thread concerns
  – Safety and Liveness
  – Use of synchronization and signaling

• Threading design patterns
Computation Abstractions

A computer

Processes vs. Threads

Processes do not share data

Threads share data within a process
So, What Is a Thread?

- **Conceptually**: it is a parallel computation occurring within a process
- **Implementation view**: it’s a program counter and a stack. The heap and static area are shared among all threads
- All programs have at least one thread (main)

Why Multiple Threads?

- Performance:
  - Parallelism on multiprocessors
  - Concurrency of computation and I/O
- Can easily express some programming paradigms
  - Event processing
  - Simulations
- Keep computations separate, as in an OS
  - But - why not use processes?
Why Not Multiple Threads?

• Complexity:
  – Dealing with safety, liveness, composition
  – The root of the problem is shared state

• Overhead
  – Higher resource usage
  – May limit performance compared to direct event processing
    • context switching, locking, etc.

Programming Threads

• Threads are available in many languages
  – C, C++, Objective Caml, Java, SmallTalk …

• In many languages (e.g., C and C++), threads are a platform specific add-on
  – Not part of the language specification

• Part of the Java language specification
Java Threads

- Every application has at least one thread
  - The “main” thread, started by the JVM to run the application’s `main()` method
- The code executed by `main()` can create other threads
  - Explicitly, using the `Thread` class
  - Implicitly, by calling libraries that create threads as a consequence
    - RMI, AWT/Swing, Applets, etc.

Java Threads: Creation

- To explicitly create a thread
  - Instantiate a `Thread` object
    - An object of class `Thread` or a subclass of `Thread`
  - Invoke the object’s `start()` method
    - This will start executing the `Thread`’s `run()` method concurrently with the current thread
  - Thread terminates when its `run()` method returns
Java Threads: Creation

Running Example: Alarms

• Goal: let us set alarms that will be triggered in the future
  – Input: Time t (seconds) and message m
  – Result: We’ll see m printed after t seconds
Example: Synchronous alarms

while (true) {
    System.out.print("Alarm> ");

    // read user input
    String line = b.readLine();
    parseInput(line); // sets timeout

    // wait (in secs)
    try {
        Thread.sleep(timeout * 1000);
    } catch (InterruptedException e) { }
    System.out.println("(+timeout+") "+msg);
}

Making It Threaded (1)

public class AlarmThread extends Thread {
    private String msg = null;
    private int timeout = 0;

    public AlarmThread(String msg, int time) {
        this.msg = msg;
        this.timeout = time;
    }

    public void run() {
        try {
            Thread.sleep(timeout * 1000);
        } catch (InterruptedException e) { }
        System.out.println("(+timeout+") "+msg);
    }
}
Making It Threaded (2)

while (true) {
    System.out.print("Alarm> ");

    // read user input
    String line = b.readLine();
    parseInput(line);
    if (m != null) {
        // start alarm thread
        Thread t = new AlarmThread(m,tm);
        t.start();
    }
}

Alternative: The Runnable Interface

- Extending Thread prohibits a different parent
- Instead implement Runnable
  - Declares that the class has a void run() method
- Construct a Thread from the Runnable
  - Constructor Thread(Runnable target)
  - Constructor Thread(Runnable target, String name)
public class AlarmRunnable implements Runnable {
    private String msg = null;
    private int timeout = 0;

    public AlarmRunnable(String msg, int time) {
        this.msg = msg;
        this.timeout = time;
    }

    public void run() {
        try {
            Thread.sleep(timeout * 1000);
        } catch (InterruptedException e) { }
        System.out.println("("+timeout+") "+msg);
    }
}

while (true) {
    System.out.print("Alarm> ");

    // read user input
    String line = b.readLine();
    parseInput(line);
    if (m != null) {
        // start alarm thread
        Thread t = new Thread(
            new AlarmRunnable(m,tm));
        t.start();
    }
}
Notes: Passing Parameters

• `run()` doesn’t take parameters
• We “pass parameters” to the new thread by storing them as private fields
  – In the extended class
  – Or in the `Runnable` object
  – Example: the time to wait and the message to print in the `AlarmThread` class