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 signalling
- Threading design patterns

Computation Abstractions

Processes vs. Threads

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
  - Java OS
Why not multiple threads?

- Complexity:
  - Dealing with safety, liveness, composition
- Overhead
  - Higher resource usage
- We’ll compare threads to their alternatives a bit later …

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
  - Invoke the object’s start() method
    - This will start executing the Thread’s run() method concurrently with the current thread
  - Thus, need to provide a run() method
    - Easy: subclass Thread and override run().

Example: Synchronous alarms

```java
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)

```java
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)

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

    // read user input
    String line = b.readLine();
    // creates a AlarmThread to wait timeout secs
    Thread t = parseInput(line);
    // wait for alarm concurrently
    if (t != null)
        t.start();
}
```

Alternative: the Runnable interface

- Extending Thread prohibits a different parent
- Instead implement Runnable
  - declares that the class has a void run() method
- Can construct a new Thread
  - and give it an object of type Runnable as an argument to the constructor
  - Thread(Runnable target)
  - Thread(Runnable target, String name)

Thread example revisited

```java
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);
    }
}
```

Change is in parseInput

- Old parseInput does
  - return new AlarmThread(m,t);
- New parseInput does
  - return new Thread(new AlarmRunnable(m,t));
- Code in while loop doesn’t change

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 the Runnable object
- Example: the time to wait and the message to print in the AlarmThread class
Once a new thread is created, how does it interact with existing threads?

This is a question of scheduling:
- Given N processors and M threads, which thread(s) should be run at any given time?

OS schedules a single-threaded process on a single processor
- Multithreaded process scheduling:
  - One thread per processor
    - Effectively splits a process across CPU's
    - Exploits hardware-level concurrency
  - Many threads per processor
    - Need to share CPU in slices of time

Thread scheduling
- When multiple threads share a CPU, must decide:
  - When the current thread should stop running
  - What thread to run next
- A thread can voluntarily yield() the CPU
  - call to yield may be ignored; don’t depend on it
- Preemptive schedulers can de-schedule the current thread at any time
  - Not all JVMs use preemptive scheduling, so a thread stuck in a loop may never yield by itself. Therefore, put yield() into loops
- Threads are de-scheduled whenever they block (e.g., on a lock or on I/O) or go to sleep

Concurrency
- Different threads from the same application can be running at the same time on different processors

Interleaving
- Threads can be pre-empted at any time in order to schedule other threads
Thread Lifecycle

- While a thread executes, it goes through a number of different phases
  - New: created but not yet started
  - Runnable: is running, or can run on a free CPU
  - Blocked: waiting for I/O or on a lock
  - Sleeping: paused for a user-specified interval
  - Terminated: completed

Which thread to run next?

- The scheduler looks at all of the runnable threads, including threads that were unblocked because
  - A lock was released
  - I/O became available
  - They finished sleeping, etc.
- Of these threads, it considers the thread’s priority. This can be set with `setPriority()`. Higher priority threads get preference.
  - Oftentimes, threads waiting for I/O are also preferred.

Simple thread methods

- `void start()`
- `boolean isAlive()`
- `void setPriority(int newPriority)`
  - thread scheduler might respect priority
- `void join()` throws InterruptedException
  - waits for a thread to die/finish

Example: threaded, sync alarm

```java
while (true) {
  System.out.println("Alarm> ");

  // read user input
  String line = b.readLine();
  Thread t = parseInput(line);

  // wait (in secs) asynchronously
  if (t != null)
    t.start();

  // wait for the thread to complete
  t.join();
}
```

Simple static thread methods

- `void yield()`
  - Give up the CPU
- `void sleep(long milliseconds)` throws InterruptedException
  - Sleep for the given period
- `Thread currentThread()`
  - Thread object for currently executing thread
- All apply to thread invoking the method

Daemon threads

- `void setDaemon(boolean on)`
  - Marks thread as a daemon thread
  - Must be set before thread started
- By default, thread acquires status of thread that spawned it
- Program execution terminates when no threads running except daemons