CMSC 433 – Programming Language Technologies and Paradigms

Concurrency
What is Concurrency?

• Simple definition
  – *Sequential* programs have one thread of control
  – *Concurrent* programs have many
Concurrency vs. Parallelism

- **Concurrency**
  - Logically simultaneous processing
  - Does not imply multiple physical processing elements (PPEs)

- **Parallelism**
  - Physically simultaneous processing
  - Involves multiple PPEs and/or independent devices

- Will ignore this distinction from here on out
Benefits of Concurrency

• Exploiting multiple processors
  – Processor speeds are not increasing as fast as they used to
  – Multi-CPU machines becoming standard
  – Can’t take full advantage of multiple CPUs without concurrent software
Benefits of Concurrency (cont.)

• For some problems, concurrency provides a very natural programming model

• For example, problems involving many, largely independent actors or actions, e.g.,
  – Simulations
  – Compute servers
• Isolates and simplifies tasks
• For instance servers typically interact with multiple clients
  – High performance, non-concurrent implementations have to multiplex (switch between) clients
  – Concurrent servers can handle each client in a separate thread of control
Concurrency Risks

• Concurrency is notoriously complex
  – Behaviors can depend on the relative timing of concurrent events
  – Bugs can be hard to reproduce

• Over the next few weeks we will discuss
  – What these difficulties are
  – How we can build correct concurrent systems
Implementing Concurrent Systems

• Overview
• What are threads?
  – Conceptual view
  – Java implementation
• Thread concerns
  – Safety and Liveness
• Threading design patterns
Computation Abstractions

A computer
Processes vs. Threads

Processes don’t easily share data

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

- **Conceptually**: It’s a concurrent 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 (*aka* the main thread)
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
  – Can do this with processes, but they are more heavyweight
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 …

• Threads are part of the Java language specification

• In other languages (e.g., C), threads are a platform specific add-on
Java Threads

• Every application has at least one thread
  – The “main” thread, started by the JVM to run the application’s main() method

• That code can create other threads
  – Explicitly, by 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: set alarms that will be triggered in the future
  – Input: Time $t$ (in seconds from now) and a Message $m$
  – Result: We’ll see $m$ printed after $t$ seconds
Example: Synchronous Alarms

• See: ThreadTestSync.java
Making It Threaded

• See: ThreadTestThreaded.java
Alternative: The **Runnable Interface**

- Extending **Thread** prohibits you from extending 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)**
Thread Example Revisited

• See: ThreadTestRunnable.java
• **run()** doesn’t take parameters
• Can “pass parameters” to the new thread by storing them as fields
  – In a Thread subclass or in a **Runnable** object
  – Example: the msg and timeout fields in the AlarmThread and AlarmRunnable classes
Thread Scheduling

• 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?
Thread Scheduling

• Multithreaded process scheduling:
  – More processors than threads
    • Can have each thread runs on its own processor
    • Splits a process across multiple CPU’s
    • Exploits hardware-level concurrency
  – More threads than processors
    • Need to share CPU in slices of time
One process per CPU
Scheduling Example (2)

Threads shared between CPU’s

CPU 1
- p1
- p2

CPU 2
- p1
- p2

p2 threads: [bars]

p1 threads: [bars]
Scheduling Consequences

• Parallelism
  – 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 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 \texttt{yield()} the CPU
  – Call to \texttt{yield()} may be ignored; you can’t depend on it

• \textit{Preemptive schedulers} can de-schedule the current thread at any time

• Threads are de-scheduled whenever they block (e.g., on a lock or on I/O) or go to sleep
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 for 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 typically get preference.
  – Oftentimes, threads waiting for I/O are also preferred
Simple Thread Methods

- `void start()`
- `boolean isAlive()`
- `void setPriority(int newPriority)`
  - Scheduler might/might not respect priority
- `void join()` throws `InterruptedException`
  - Waits for a thread to die/finish
Example: Threaded, Sync Alarm

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

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

    // wait (in secs) asynchronously
    if (m != null) {
        // start alarm thread
        Thread t = new AlarmThread(m, tm);
        t.start();
        // wait for the thread to complete
        t.join();
    }
}
```
Simple Static Thread Methods

• void yield()
  – Hint to 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
• Assuming the ClientSimulator is fixed
  – What factors accounted for the program’s running time?
  – What possibilities might exist to speed things up?
A MultiThreaded Logging Server

• Accepts records from client
• Creates a thread to process the record
• Organization
  – LoggingServerUtils
    • LoggingServerCore.java
    • DataRecord.java
    • MsgHandler.java
  – Client
    • ClientSimulator.java
  – Server
    • MultiThreadedServer.java
Let’s Look at the Code

• MultiThreadedLoggingServer
Ungraded Assignment

• Download the code
• Read and understand how it works
• Run this code and observe its performance
  – How does this code perform relative to the SingleThreadedServer?
  – What might account for any performance differences?
  – Are there any problems/concerns with the multiThreadedServer’s behavior?