CS18000: Problem Solving and Object-Oriented Programming

Concurrency
Video 1
History of Concurrency, Java Threads
Concurrent Programming and Synchronization

Threads
Time Slicing

• In early days -- one program ran at a time
• Along came windowed operating systems
• Giving illusion of multiple programs running at a time
• Email tool, web browser, text editor
• 1 second = 1000 milliseconds
• Time slicing -- running each program a few hundred milliseconds
• Humans are so slow that it appeared that all programs were running simultaneously
Mobile Devices

- Illusion worked well as programs became more and more complex
- Simply make the CPU run faster
- Along came mobile devices -- laptops and phones
- Run on batteries
- As the CPU runs faster battery runs down quicker and gets very hot
Multiple Cores

- How about putting 2 or more CPUs (cores) on same chip?
- If run 2 cores at half speed, same as 1 core at full speed
- Slower cores consume less battery and generate less heat
- Now most laptops and phones have 4 or more cores
- If you have 4 apps running, they may actually be running simultaneously
- But, what if you have more than 4
- Time slicing is alive and well -- some apps are in a queue waiting to get a core
Threads

• Multiple cores can even be used to speed up one program
• A program can have more than one part (thread) running simultaneously
• What if a database is spread over 3 files?
• Run 3 threads on 3 cores ... each looking for the same item
• Program runs (approximately) 3 times faster
Sequential vs. Concurrent

• Sequential:
  – A single “thread of execution” weaves its way through your program
  – A single PC (“program counter”) identifies the current instruction being executed

• Concurrent:
  – Multiple “threads of execution” are running simultaneously through your program
  – Multiple PCs are active, one for each thread
Java Threads

• Thread class with run() method
• import java.lang.*;
• Allows creation and manipulation of threads
  – Thread t = new Thread();
• Three important methods:
  – t.start(): start the thread referenced by t
  – t.join(): “join with” (wait for) the running thread t
  – t.run(): called by start() in a different thread
• Note: Your code does not call run() directly; instead, the start() method calls run() as part of the new thread sequence
public class MainThread {
    public static void main(String[] args) {
        Thread t = Thread.currentThread();
        System.out.printf("main thread = %s\n", t);

        System.out.printf("going to sleep...\n");
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.printf("ah, that was nice\n");

        System.out.printf("letting someone else run\n");
        Thread.yield();
        System.out.printf("back\n");
    }
}
How to Create Threads

• Create a class that implements the Runnable interface

  public class MyTask implements Runnable {
    public void run() { ... }
  }

  Thread t = new Thread(new MyTask());
Example: MyTask

```java
public class MyTask implements Runnable {
    public static void main(String[] args) {
        MyTask m = new MyTask();
        Thread t = new Thread(m);

        t.start();
    }

    public void run() {
        System.out.printf("now in %s\n", Thread.currentThread());
    }
}
```
Video 2
Task and Domain Decomposition
Concurrent Programming and Synchronization

Runnables
Using Concurrent Processing

• How do you break down a large problem into pieces?
• Need to \textit{decompose} the problem into pieces
• Two approaches to decomposition
  – By the tasks to be done
  – By the data to be processed
Task Decomposition

- Split task into multiple subtasks
- Each subtask runs different code
- Each subtask runs on its own core (processor)
- Primary benefit: responsiveness
  - GUI is one task
  - Background computation a different task
  - GUI has its own core, so is always responsive
Domain Decomposition

• Domain:
  – Input examined by the problem
• Divide domain into pieces (subdomains)
• Each subtask runs the
  – same code but
  – on different input
• Each subdomain is given to a task running on a different core
• Primary benefit: raw speed
Examples: Task Decomposition

• Updating the screen of a video game
  – One task processes player moves
  – One task updates the display
  – Two tasks communicate as necessary

• Can a student register for a class?
  – One task determines if student has pre-reqs
  – One task decides if student has class at same time
  – One task determines if a seat is available in class
  – Combine results when each task is done
Task Decomposition Example: Video Game Updates

- Core 1
  - A: start game
  - B: process move
  - C: update score
  - E: end game
  - D: repaint screen

- Core 2
  - D: repaint screen

(a) [Diagram showing task flow]
(b) [Diagram showing task flow]
Examples: Domain Decomposition

• Factoring a large number
  – Trial divide up to square root of number
  – Assign blocks of trial divisors to separate tasks
  – First task to divide with 0 remainder stops process

• Finding words in a word search puzzle
  – Divide word list into subsets
  – Assign each subset to a separate task
  – Tasks search the puzzle grid, recording hits
Domain Decomposition Example: Matrix Multiplication

Using domain decomposition to compute the matrix product $A \times B$.

The top half is computed on Core 1 and the bottom half on Core 2.
Task Decomposition

```java
public class Model implements Runnable {
    // This run method keeps track of where the characters are, what
direction they are moving and at what speed
    public void run() {...}
}

public class View implements Runnable {
    // This run method updates the GUI showing where each character is
right now
    public void run() {...}
}
```
public class Game {
    public static void main(String[] args) {
        // Thread data keeps track of where the characters are, what direction they
        // are moving and at what speed
        Thread data = new Thread(new Model(...));
        // Thread gui updates the GUI showing each character
        Thread gui = new Thread(new View(...));

        // Start the data thread. It will receive information from the GUI about use
        // of controls, mouse clicks, etc.
        data.start();
        // Start the gui thread. It will receive information from Model class methods
        // about where the characters are, what direction they are moving and at what
        // speed
        gui.start();
        ...
    }
}
Video 1
Synchronization Using join()
Concurrent Programming and Synchronization

Synchronization
Unpredictability in Thread Execution

• Thread execution may be interrupted
  – “Time slicing” of threads (and processes) prevents one thread from “hogging” the CPU
  – Higher priority activities may interrupt the thread: e.g., I/O
• Multiple threads do not always proceed at the same rate
• Coordination of multiple threads a challenge
• Java provides low-level and high-level tools to deal with synchronization of threads
public class Interleave implements Runnable {
    private char c;

    public Interleave(char c) {
        this.c = c;
    }

    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.printf("%c", c);
            for (int j = 0; j < 1000; j++)
                Math.hypot(i, j);
        }
        System.out.printf("%c", Character.toUpperCase(c));
    }

    // ... continued on next slide ...
Example: Interleave (2)

// ... continued from previous slide ...

public static void main(String[] args) {
    while (true) {
        Thread t1 = new Thread(new Interleave('a'));
        Thread t2 = new Thread(new Interleave('b'));
        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) { .... }
        System.out.println();
    }
}
Join: Wait for a Thread to Finish

• A simple kind of synchronization
• For Thread t:
  
  t.join();

• Blocks the “current thread”—the one that called t.join()—until Thread t completes (returns from run())
• join() may throw an InterruptedException, so generally is in try-catch clause
Join using Try-Catch Clause

```java
try {
    t.join();
} catch (InterruptedException e) {
    e.printStackTrace(); // example
}
```
public class FindStudent implements Runnable {
    private static Student stu = null; // stu will be returned
    private static boolean found = false; // haven’t found stu yet
    private static int student; // who we are looking for
    private Fileserver fileserver; // where to look
    private Student maybe = null; // maybe this is the one

    public FindStudent (Fileserver f) {fileserver=f;};

    public Student search (int student) throws StudentNotFoundException {
        this.student = student;
        Thread t1 = new Thread (new FindStudent (fileserver1));
        Thread t2 = new Thread (new FindStudent (fileserver2));
        Thread t3 = new Thread (new FindStudent (fileserver3));
    }
Fileserver

may be

null

found

false

student [257461982]
Search for a Student using Threads

t1.start();
t2.start();
t3.start();
try {
    t1.join();
    t2.join();
    t3.join();
} catch (InterruptedException e) { .... }

if (found)
    return stu;
else
    throw new StudentNotFoundException
        (Integer.toString(student));
public void run() {
    while (more to read on fileserver) {
        if (found) return;
        // read the next Student record, maybe points to it
        ...
        if (maybe.getID() == student) {
            stu = maybe;
            found = true;
        }
    }
}
Video 2
Race Conditions
Salary

Supervisor + $5,000

40.00

Head of Software Engineering + $10,000
Synchronization Problem: Race Condition

• As threads “race” through execution, their instructions are interleaved at the nanosecond level
  – Byte codes within a thread always executed in relative order, as expected
  – Byte codes between threads not executed in predictable absolute order

• Causes problems when accessing and updating shared data
public class RaceCondition implements Runnable {
    private static int counter;

    public static void main(String[] args) {
        counter = 0;

        Thread t1 = new Thread(new RaceCondition());
        Thread t2 = new Thread(new RaceCondition());

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) { e.printStackTrace(); }

        System.out.printf("counter = %d\n", counter);
    }
    // ... run() method on next slide ...
}
Example: RaceCondition (2)

```java
public void run() {
    for (int i = 0; i < 10000; i++) {
        counter++;
    }
}
```
Two Threads Updating a Counter

• Thread 1
  
  ```
  int t1 = counter;
  t1 = t1 + 1;
  counter = t1;
  ```

• Thread 2
  
  ```
  int t2 = counter;
  t2 = t2 + 1;
  counter = t2;
  ```
Counter

8471

8472

8473

→ 1

8471

8472
Counter

\[ t_1 \]

\[ \text{8471} \]

\[ \text{8471} \]

\[ \epsilon_1 \]

\[ \text{8471} \]

\[ \epsilon_2 \]
Solution: Synchronize Threads

• Java keyword “synchronized”
• Allows two or more threads to use a common object to avoid race conditions
• Syntax:
  
  ```java
  synchronized (object) {
      statements;  // modify shared data here
  }
  ```
• Among all threads synchronizing using the same object, only one thread can be “inside” the block of statements at a time
public class NoRaceCondition implements Runnable {
    private static int counter = 0;
    private static Object gateKeeper = new Object();

    public static void main(String[] args) {
        Thread t1 = new Thread(new NoRaceCondition());
        Thread t2 = new Thread(new NoRaceCondition());
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) { e.printStackTrace(); }
        System.out.printf("counter = %d\n", counter);
    }

    public void run() {
        for (int i = 0; i < 10000; i++) {
            synchronized (gateKeeper) { counter++; }
        }
    }
}
Shared Memory Architecture

• Two paradigms for supporting concurrent or parallel processing
  • Message Passing: processes
    – Messages sent between separate processes
    – Generally, one process per program
    – May run on different physical computers
  • Shared Memory: threads
    – Single program
    – All threads share the same memory space
    – This approach is what we are using in Java
Thread States

• A Java thread goes through several states in its lifetime:
  – New thread: created but not yet started
  – Runnable: started and available to be run
  – Not Runnable: sleeping, waiting for i/o, etc.
  – Terminated: returned from the run() method
• t.sleep(n) puts the current thread to sleep for n milliseconds; allows other threads to run
• t.yield() “gives up” the CPU, letting another thread run
Thread States

- **New Thread**
  - start()

- **Runnable**
  - Thread.yield()
  - run() method finishes
  - Thread.sleep()
  - wait()
  - wake up notify()

- **Not Runnable**

- ** Terminated **