Principles of Software Construction: Concurrency, Part 1

Josh Bloch

Charlie Garrod
Administrivia

• Midterm review tomorrow 7-9pm, HH B103
• Midterm on Thursday
• HW 5 team signup deadline **tonight**
  – If you’re still looking for a homework 5 team, come to front of room after class
Foundations of the Software Engineering minor

- Core computer science fundamentals
- Building good software
- Organizing a software project
  - Development teams, customers, and users
  - Process, req’ts, estimation, management, & methods
- The larger context of software
  - Business, society, policy
- Engineering experience
- Communication skills
  - Written and oral
SE minor requirements

• Prerequisite: 15-214
• Two core courses
  – 15-313 Foundations of SE (fall semesters)
  – 15-413 SE Practicum (spring semesters)
• Three electives
  – Technical
  – Engineering
  – Business or policy
• Software engineering internship + reflection
  – 8+ weeks in an industrial setting, then
  – 17-413
To apply to be a Software Engineering minor

- Email clegoues@cs.cmu.edu
  - Your name, Andrew ID, expected grad date, QPA, and minor/majors
  - Why you want to be a SE minor
  - Proposed schedule of coursework

- Spring applications due Friday, 07 November 2016
  - Only 15 SE minors accepted per graduating class

- More information at:
  - http://isri.cmu.edu/education/undergrad/
“A Big Delight in Every Byte”

class Delight {
    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE;
            b < Byte.MAX_VALUE; b++) {
            if (b == 0x90) {
                System.out.print("Joy! ");
            }
        }
    }
}
What Does It Print?

class Delight {
    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE;
             b < Byte.MAX_VALUE; b++) {
            if (b == 0x90) {
                System.out.print("Joy! ");
            }
        }
    }
}

(a) Joy!
(b) Joy! Joy!
(c) Nothing
(d) None of the above
What Does It Print?

(a) Joy!
(b) Joy! Joy!
(c) Nothing
(d) None of the above

Program compares a byte with an int; byte is promoted with surprising results
Another Look

*bytes are signed; range from -128 to 127*

class Delight {
    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE; 
            b < Byte.MAX_VALUE; b++) {
            if (b == 0x90) // (b == 144)
                System.out.print("Joy! ");
        }
    }
}

// (byte)0x90 == -112
// (byte)0x90 != 0x90
You Could Fix it Like This...

- Cast `int` to `byte`
  ```java
  if (b == (byte)0x90)
      System.out.println("Joy!");
  ```

- Or convert `byte` to `int`, suppressing sign extension with mask
  ```java
  if ((b & 0xff) == 0x90)
      System.out.println("Joy!");
  ```
...But This is Even Better

```java
public class Delight {
    private static final byte TARGET = 0x90; // Won’t compile!
    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE; b < Byte.MAX_VALUE; b++)
            if (b == TARGET)
                System.out.print("Joy!");
    }
}
```

Delight.java:2: possible loss of precision
found   : int
required: byte
    private static final byte TARGET = 0x90; // Won’t compile!
    ^
The Best Solution, Debugged

```java
public class Delight {
    private static final byte TARGET = (byte) 0x90; // Fixed
    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE; b < Byte.MAX_VALUE; b++)
            if (b == TARGET)
                System.out.print("Joy!");
    }
}
```
The Moral

• **byte values are signed 😞**

• Be careful when mixing primitive types

• **Compare like-typed expressions**
  – Cast or convert one operand as necessary
  – Declared constants help keep you in line

• For language designers
  – Don’t violate principle of least astonishment
  – Don’t make programmers’ lives miserable
Key concepts from Tuesday...

• Java I/O is a bit of a mess
  – There are many ways to do things
  – Use readers/writers most of the time
  – Use Scanner for casual use

• Reflection is tricky, but Class.forName and newInstance go a long way
Outline

I. Introduction to concurrency
II. Threading Basics
III. Synchronization
What is a thread? (review)

• Short for *thread of execution*
• Multiple threads run in same program concurrently
• Threads share the same address space
  – Changes made by one thread may be read by others
• Multithreaded programming
  – Also known as shared-memory multiprocessing
Processor characteristics over time
Power requirements of a CPU

• power = capacitance $\times$ \textit{voltage}^2 $\times$ frequency

• To increase performance
  – More transistors, thinner wires
    • More power leakage: \textit{increase voltage}
  – Increase clock frequency
    • Change electrical state faster: \textit{increase voltage}

• \textit{Dennard scaling} – as transistors get smaller, power density is approximately constant...
  – ...until early 2000s

• Now: Power is super-linear in CPU performance
Failure of Dennard Scaling forced our hand

• Must reduce heat by limiting power
• Limit power by reducing frequency and/or voltage
• In other words, build slower cores...
  – ...but build more of them
• Adding cores ups power linearly with performance
• But concurrency is required to utilize multiple cores
Concurrency then and now

• In past multi-threading just a convenient abstraction
  – GUI design: event dispatch thread
  – Server design: isolate each client’s work
  – Workflow design: isolate producers and consumers
• Now: **required** for scalability and performance
We are all concurrent programmers

• Java is inherently multithreaded
• In order to utilize our multicore processors, we must write multithreaded code
• Good news: a lot of it is written for you
  – Excellent libraries exist (java.util.concurrent)
• Bad news: you still must understand fundamentals
  – to use libraries effectively
  – to debug programs that make use of them
Outline

I. Introduction to concurrency
II. Threading Basics
III. Synchronization
The Runnable interface - represents the work to be done by a thread

An instance is passed to each thread when it is created

```java
public interface Runnable {
    void run();
}
```
A simple example: running a task asynchronously

public class Background {
    public static void runInBackground(Runnable task) {
        Thread t = new Thread(task);
        t.start();
    }

    // Sample use
    public static void main(String[] args) {
        runInBackground(Background::slowTask);
    }

    private static void slowTask() {
        try {
            TimeUnit.SECONDS.sleep(5); // Represents computation
        } catch (InterruptedException ie) {
            throw new AssertionError(ie);
        }
    }
}

public static void main(String[] args) throws InterruptedException {
    int n = Integer.parseInt(args[0]);
    int wordsPerThread = words.length / n;
    Thread[] threads = new Thread[n];
    String[][] results = new String[n][];
    for (int i = 0; i < n; i++) {
        int start = i == 0 ? 0 : i * wordsPerThread - 2;
        int end = i == n-1 ? words.length : (i + 1) * wordsPerThread;
        int m = i; // Only constants can be captured by lambdas
        threads[i] = new Thread(() -> {
            results[m] = cryptarithms(words, start, end);
        });
    }
    for (Thread t : threads) t.start();
    for (Thread t : threads) t.join();
    System.out.println(Arrays.deepToString(results));
}
Outline

I. Introduction to concurrency
II. Threading Basics
III. Synchronization
Example: Money-Grab (1)

```java
class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }

    static void transferFrom(BankAccount source, BankAccount dest, long amount) {
        source.balance -= amount;
        dest.balance += amount;
    }

    public long balance() {
        return balance;
    }
}
```
Example: Money-Grab (2)

What would you expect this to print?

```java
public static void main(String[] args) throws InterruptedException {
    BankAccount bugs = new BankAccount(100);
    BankAccount daffy = new BankAccount(100);

    Thread bugsThread = new Thread(() -> {
        for (int i = 0; i < 1000000; i++)
            transferFrom(daffy, bugs, 100);
    });

    Thread daffyThread = new Thread(() -> {
        for (int i = 0; i < 1000000; i++)
            transferFrom(bugs, daffy, 100);
    });

    bugsThread.start();
    daffyThread.start();
    bugsThread.join();
    daffyThread.join();
    System.out.println(bugs.balance + daffy.balance());
}
```
What went wrong?

• Daffy & Bugs threads were stomping each other
• Transfers did not happen in sequence
• Constituent reads and writes interleaved randomly
• Random results ensued
It’s easy to fix!

```java
public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }

    static synchronized void transferFrom(BankAccount source, BankAccount dest, long amount) {
        source.balance -= amount;
        dest.balance += amount;
    }

    public long balance() {
        return balance;
    }
}
```
Example: serial number generation

What would you expect this to print?

```java
public class SerialNumber {
    private static long nextSerialNumber = 0;
    public static long generateSerialNumber() {
        return nextSerialNumber++;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread threads[] = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 1_000_000; j++)
                    generateSerialNumber();
            });
            threads[i].start();
        }
        for(Thread thread : threads) thread.join();
        System.out.println(generateSerialNumber());
    }
}
```
What went wrong?

• The ++ (increment) operator is not atomic!
  – It reads a field, increments value, and writes it back
• If multiple calls to `generateSerialNumber` see the same value, they generate duplicates
Again, the fix is easy

```java
public class SerialNumber {
    private static int nextSerialNumber = 0;
    public static synchronized int generateSerialNumber() {
        return nextSerialNumber++;
    }

    public static void main(String[] args) throws InterruptedException{
        Thread threads[] = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 1_000_000; j++)
                    generateSerialNumber();
            });
            threads[i].start();
        }
        for(Thread thread : threads) thread.join();
        System.out.println(generateSerialNumber());
    }
}
```
But you can do better!

`java.util.concurrent` is your friend

```java
public class SerialNumber {
    private static AtomicLong nextSerialNumber = new AtomicLong();
    public static long generateSerialNumber() {
        return nextSerialNumber.getAndIncrement();
    }
    public static void main(String[] args) throws InterruptedException{
        Thread threads[] = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 1_000_000; j++)
                    generateSerialNumber();
            });
            threads[i].start();
        }
        for(Thread thread : threads) thread.join();
        System.out.println(generateSerialNumber());
    }
}
```
Example: cooperative thread termination

How long would you expect this to run?

public class StopThread {
    private static boolean stopRequested;

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested) {
                /* Do something */
            }
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
What went wrong?

• In the absence of synchronization, there is no guarantee as to when, if ever, one thread will see changes made by another!

• VMs can and do perform this optimization:

```java
while (!done)
    /* do something */ ;
```

becomes:

```java
if (!done)
    while (true)
        /* do something */ ;
```
public class StopThread {
    private static boolean stopRequested;
    private static synchronized void requestStop() {
        stopRequested = true;
    }
    private static synchronized boolean stopRequested() {
        return stopRequested;
    }

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested()) {
                /* Do something */
            });
        backgroundThread.start();
        TimeUnit.SECONDS.sleep(1);
        requestStop();
    }
}
You can do better (?)

volatile is synchronization sans mutual exclusion

public class StopThread {
    private static volatile boolean stopRequested;

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested)
                /* Do something */
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
Summary

• Like it or not, you’re a concurrent programmer
• Ideally, avoid shared mutable state
• If you can’t avoid it, synchronize properly
  – Failure to do so causes safety and liveness failures
  – If you don’t sync properly, your program won’t work
• Even atomic operations require synchronization
  – e.g., stopRequested = true
  – And some things that look atomic aren’t (e.g., val++)

15-214