Principles of Software Construction:

Concurrency, Part 1

Josh Bloch Charlie Garrod

School of
Computer Science

institute for
SOFTWARE
RESEARCH

Administrivia

- Midterm review tomorrow 7-9pm
- Midterm on Thursday
- If you're still looking for a homework 5 team, come to front of room after class

PSA: One-line, high-quality hash functions

Tip o' the hat to William Chargin

```
public class Thing {
    private int x;
    private double y;
    private String name;
    private List<Player> things;

    @Override
    public int hashCode() {
        return Objects.hash(x, y, name, things);
    }
}
```

This will do the Right Thing (i.e., $\sum 31^k \cdot \text{hashCode}(x_k)$), and it couldn't be easier. No excuse.

Key concepts from Tuesday…

- Java I/O is a bit of a mess
  - There are many ways to do things
  - Use readers most of the time
- Reflection is tricky, but `Class.forName` and `newInstance` go a long way

Outline

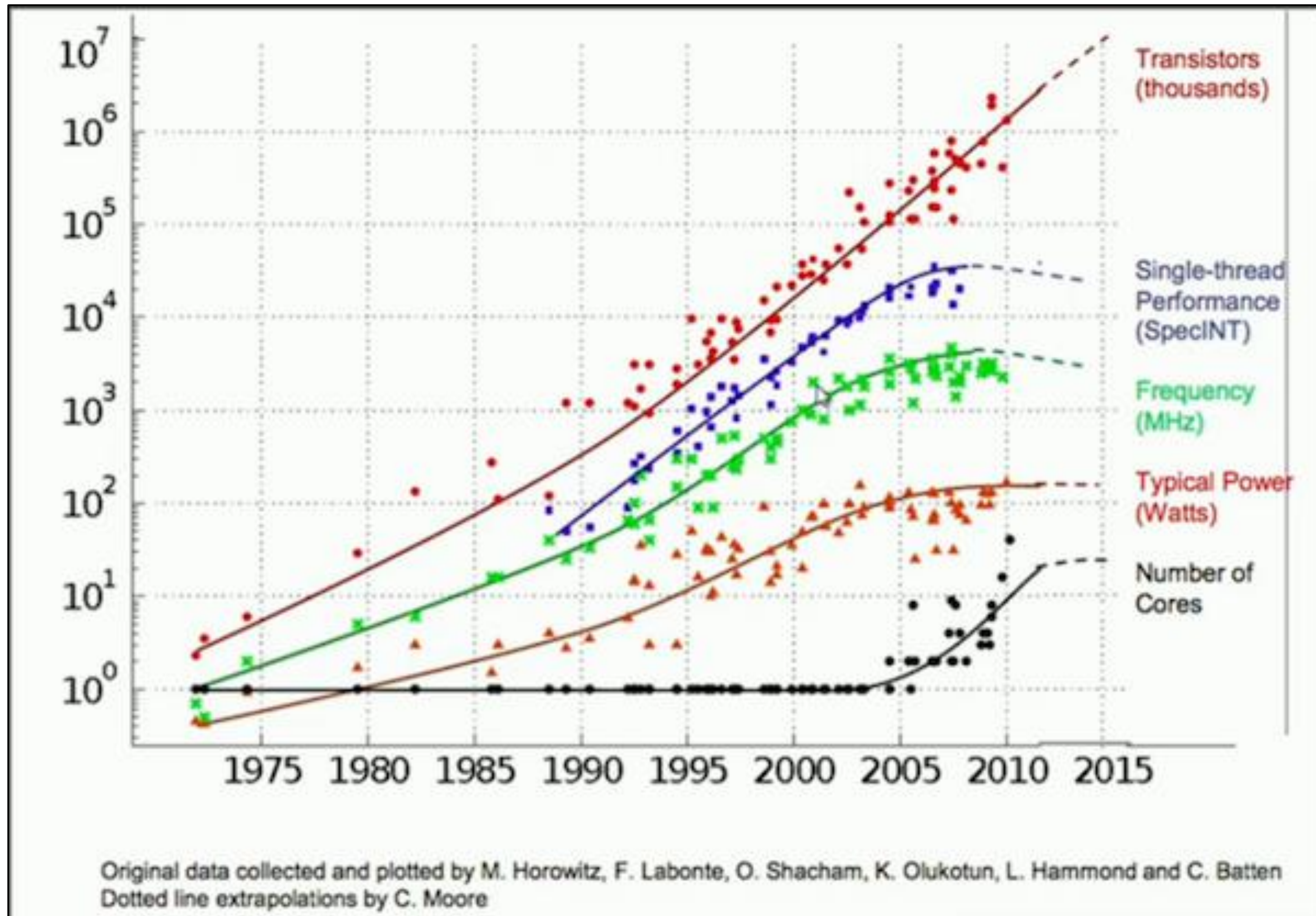
I. Introduction to concurrency
II. Threading Basics
III. Synchronization

isr institute for SOFTWARE RESEARCH

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



isr institute for SOFTWARE RESEARCH

Power requirements of a CPU

- power capacitance × voltage^2 × frequency
- To increase performance
  - More transistors, thinner wires
    - More power leakage: increase voltage
  - Increase clock frequency
    - Change electrical state faster: increase voltage
- Dennard scaling: As transistors get smaller, power density is approximately constant…
  - …until early 2000s
- Now: Power super-linear in CPU performance

Failure of Dennard Scaling forced our hand

- Must reduce heat by limiting power input
- Limit power by reducing frequency and voltage
- In other words, build slower cores…
  - …but build more of them
- Adding cores increases power linearly with perf
- But concurrency is required to utilize multiple cores

Concurrency then and now

- In past multi-threading just a convenient abstraction
  - GUI design: event threads
  - 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

isr institute for SOFTWARE RESEARCH

Outline

I. Introduction to concurrency
II. Threading Basics
III. Synchronization

isr institute for SOFTWARE RESEARCH

The `Runnable` interface - represents work to be done by a thread

An instance is passed to each thread when it is created

```
public interface Runnable {
    void run();
}
```


institute for SOFTWARE RESEARCH

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);
        } catch (InterruptedException ie) {
            throw new AssertionError(ie);
        }
    }
}
```


isr institute for SOFTWARE RESEARCH

Multithreaded driver (déjà vu)

```
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));
}
```


isr institute for SOFTWARE RESEARCH

Outline

I. Introduction to concurrency
II. Threading Basics
III. Synchronization

isr institute for SOFTWARE RESEARCH

Example: Money-Grab (1)

```
public 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?

```
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

isr institute for SOFTWARE RESEARCH

It's easy to fix!

```
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?

```
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());
    }
}
```


institute for SOFTWARE RESEARCH

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

isr institute for SOFTWARE RESEARCH

Again, the fix is easy

```
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!

```
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:

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

becomes:

```
if (!done)
    while (true)
        /* do something */ ;
```


isr institute for SOFTWARE RESEARCH

How do you fix it?

```
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
  - And some things that look atomic aren't (e.g., `val++`)