Principles of Software Construction: Objects, Design, and Concurrency

Part 3: Concurrency

Concurrency, Part 2

Josh Bloch Charlie Garrod

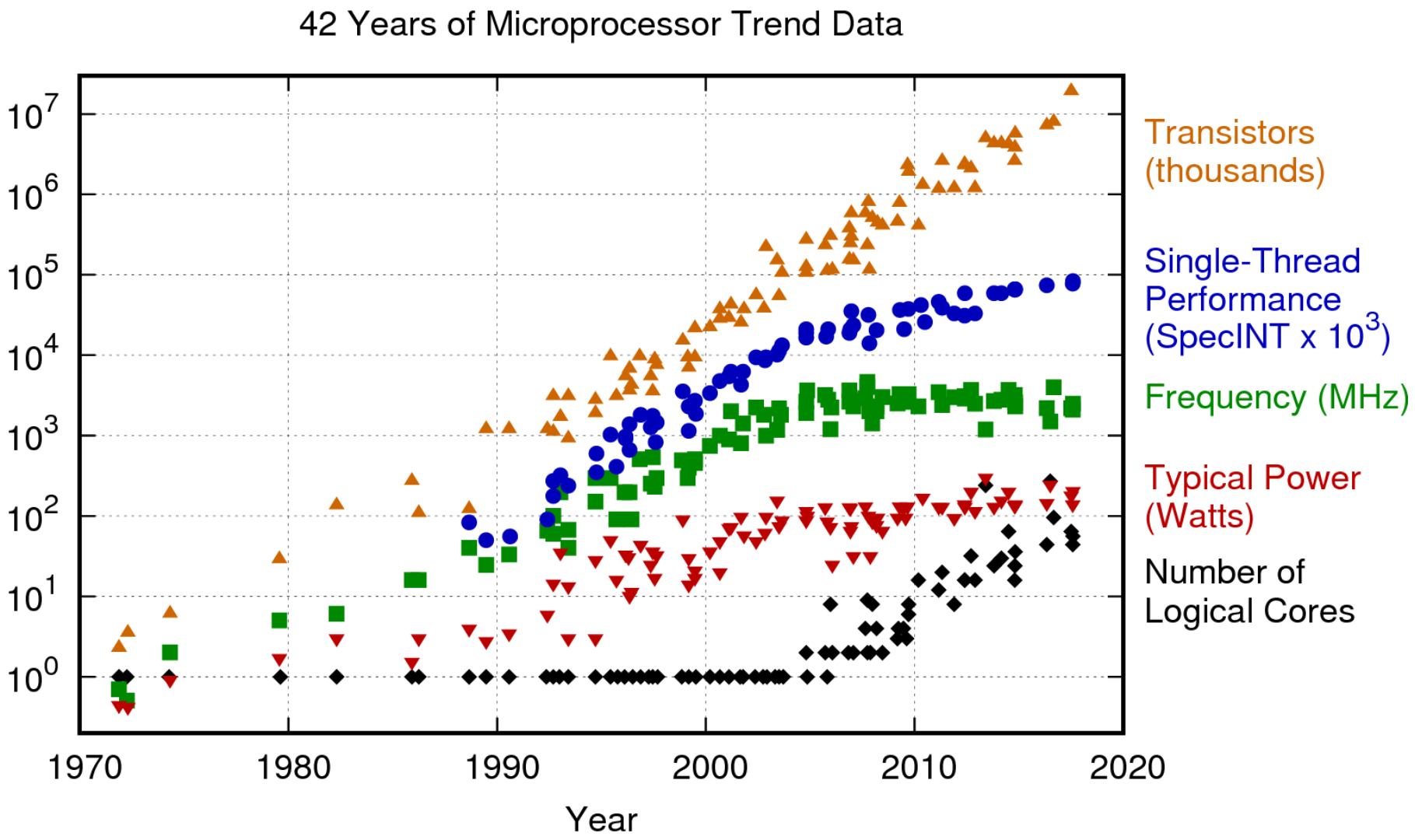
Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Second midterm currently in progress
  - Due tonight (4/8) at 11:59 EDT
- HW 5a due next Tuesday 4/13
  - pdf and planning doc in github by 9:00am Eastern time
  - Presentations at scheduled time (you signed up)
- Reading due next Tues: Java Concurrency In Practice, 11.3-4

Key concepts from last Tuesday



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

A concurrency bug with an easy fix

```
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;
    }
}
```

Concurrency control with Java's *intrinsic* locks

- `synchronized (lock) { … }`
  - Synchronizes entire block on object `lock`; cannot forget to unlock
  - Intrinsic locks are *exclusive*: One thread at a time holds the lock
  - Intrinsic locks are *reentrant*: A thread can repeatedly get same lock
- `synchronized` on an instance method
  - **Equivalent to** `synchronized (this) { … }` for entire method
- `synchronized` on a static method in class Foo
  - **Equivalent to** `synchronized (Foo.class) { … }` for entire method

Another concurrency bug: serial number generation

```
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?

- An action is *atomic* if it is indivisible
  - Effectively, it happens all at once
    - No effects of the action are visible until it is complete
    - No other actions have an effect during the action
- Java's ++ (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

Pop quiz – Does this fix work? If not, why not?

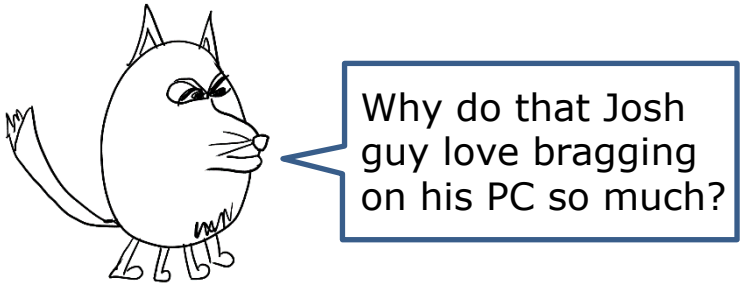
```
public class SerialNumber {
    private static volatile long nextSerialNumber = 0;

    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());
    }
}
```

It does not – `volatile` provides *only* the communications effects of synchronization (no mutual exclusion)

- But the increment operator (`i++`) is not atomic
  - It's a read followed by an increment followed by a write
- So you need mutual exclusion
  - As provided by Java's intrinsic locks (`synchronized`)
  - Or an equivalent concurrent abstraction
- As usual, `java.util.concurrent` is your best bet
  - `AtomicLong` is *significantly* better than `synchronized`
  - I ran a benchmark : 7.1 ns for `AtomicLong` vs. 13 ns for `synchronized` (Ryzen 9 3900x, 24 Java threads, 24 hyperthreads, 12 cores)



A third concurrency bug: cooperative thread termination

```
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(5);
        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
- **JVMs can and do perform this optimization ("hoisting"):**

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

becomes:

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

Pop quiz – what's wrong with this "fix"?

```
public class StopThread {
    private static boolean stopRequested;

    private static synchronized void requestStop() {
        stopRequested = true;
    }

    private static 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(5);
        requestStop();
    }
}
```

You **must** lock write and read!

*Otherwise, locking accomplishes **nothing***

```
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();
    }
}
```

Today

- More basic concurrency in Java
  - Some challenges of concurrency
- Still coming soon:
  - Higher-level abstractions for concurrency
  - Program structure for concurrency
  - Frameworks for concurrent computation

“Fixed” BankAcccount program performs poorly. Why?

```
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;
    }
}
```

A liveness problem: poor performance

```
public class BankAccount {
    private long balance;

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

A proposed fix: *lock splitting*

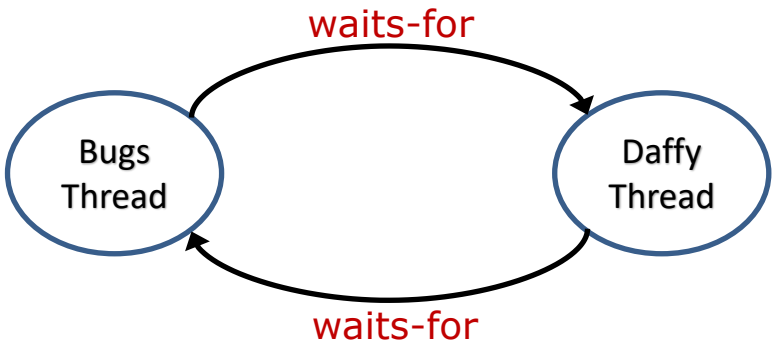
Does this work?

```
public class BankAccount {
    private long balance;

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

A liveness problem: deadlock

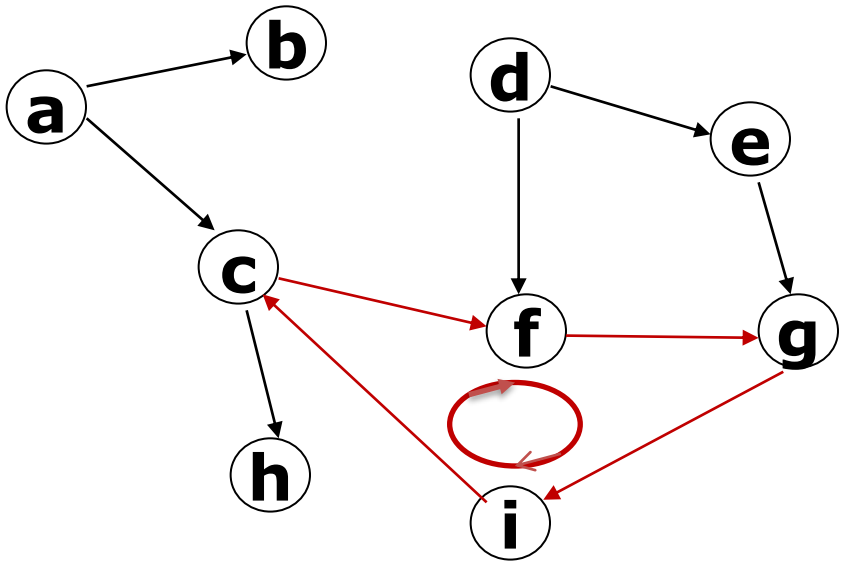
- A possible interleaving of operations:
  - `bugsThread` locks the `daffy` account
  - `daffyThread` locks the `bugs` account
  - `bugsThread` waits to lock the `bugs` account…
  - `daffyThread` waits to lock the `daffy` account…




```
synchronized(source) {
    synchronized(dest) {
        source.balance -= amount;
        dest.balance   += amount;
    }
}
```

Avoiding deadlock

- The ***waits-for graph*** represents dependencies between threads
  - Each node in the graph represents a thread
  - An edge T1⟶T2 represents that thread T1 is waiting for a lock T2 owns
- Deadlock has occurred if the waits-for graph contains a cycle
- One way to avoid deadlock: locking protocols that avoid cycles



Avoiding deadlock by ordering lock acquisition

```
public class BankAccount {
    private long balance;
    private final long id = SerialNumber.generateSerialNumber();

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

    static void transferFrom(BankAccount source,
                             BankAccount dest, long amount) {
        BankAccount first  = source.id < dest.id ? source : dest;
        BankAccount second = source.id < dest.id ? dest : source;
        synchronized (first) {
            synchronized (second) {
                source.balance -= amount;
                dest.balance += amount;
            }
        }
    }
}
```

Another subtle problem: The lock object is exposed

```
public class BankAccount {
    private long balance;
    private final long id = SerialNumber.generateSerialNumber();

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

    static void transferFrom(BankAccount source,
                             BankAccount dest, long amount) {
        BankAccount first  = source.id < dest.id ? source : dest;
        BankAccount second = source.id < dest.id ? dest : source;
        synchronized (first) {
            synchronized (second) {
                source.balance -= amount;
                dest.balance += amount;
            }
        }
    }
}
```

Concurrency and encapsulation

- Encapsulate an object's state – guarantee invariants
- But locks *are* state!
- Encapsulate synchronization – guarantee synchronization policy

An easy fix: Use a `private` lock contained in object

```
public class BankAccount {
    private long balance;
    private final long id = SerialNumber.generateSerialNumber();
    private final Object lock = new Object();

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

    static void transferFrom(BankAccount source,
                             BankAccount dest, long amount) {
        BankAccount first  = source.id < dest.id ? source : dest;
        BankAccount second = source.id < dest.id ? dest : source;
        synchronized (first.lock) {
            synchronized (second.lock) {
                source.balance -= amount;
                dest.balance += amount;
            }
        }
    }
}
```

An aside: Java Concurrency in Practice annotations

```
@ThreadSafe public class BankAccount {
    @GuardedBy("lock") private long balance;
    private final long id = SerialNumber.generateSerialNumber();
    private final Object lock = new Object();

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

    static void transferFrom(BankAccount source,
                             BankAccount dest, long amount) {
        BankAccount first  = source.id < dest.id ? source : dest;
        BankAccount second = source.id < dest.id ? dest : source;
        synchronized (first.lock) {
            synchronized (second.lock) {
                source.balance -= amount;
                dest.balance += amount;
            }
        }
    }
}
```

An aside: Java Concurrency in Practice annotations

- For classes

  @Immutable

  @ThreadSafe

  @NotThreadSafe

- For fields

  @GuardedBy

Interlude – Ye Olde Puzzler

Puzzler: “Racy Little Number”

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class LittleTest {
    int number;

    @Test
    public void test() throws InterruptedException {
        number = 0;
        Thread t = new Thread(() -> {
            assertEquals(number, 2);
        });
        number = 1;
        t.start();
        number++;
        t.join();
    }
}
```

How often does this test pass?

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class LittleTest {
    int number;

    @Test
    public void test() throws InterruptedException {
        number = 0;
        Thread t = new Thread(() -> {
            assertEquals(number, 2);
        });
        number = 1;
        t.start();
        number++;
        t.join();
    }
}
```

(a) It always fails

(b) It sometimes passes

(c) It always passes

(d) It always hangs

How often does this test pass?

(a) It always fails

(b) It sometimes passes

(c) It always passes – but it tells us nothing

(d) It always hangs

JUnit doesn't see assertion failures in other threads

Another look

```
import org.junit.*;
import static org.junit.Assert.*;

public class LittleTest {
    int number;

    @Test
    public void test() throws InterruptedException {
        number = 0;
        Thread t = new Thread(() -> {
            assertEquals(number, 2); // JUnit never sees exception!
        });
        number = 1;
        t.start();
        number++;
        t.join();
    }
}
```

How do you fix it? (1)

```
// Keep track of assertion failures during test
volatile Exception exception;
volatile Error error;

// Triggers test case failure if any thread asserts failed
@After
public void tearDown() throws Exception {
    if (error != null)
        throw error;         // In correct thread
    if (exception != null)
        throw exception;     //  "    "        "
}
```

How do you fix it? (2)

```
Thread t = new Thread(() -> {
    try {
        assertEquals(2, number);
    } catch(Error e) {
        error = e;
    } catch(Exception e) {
        exception = e;
    }
});
```

Now it sometimes passes*

*YMMV (It's a race condition)

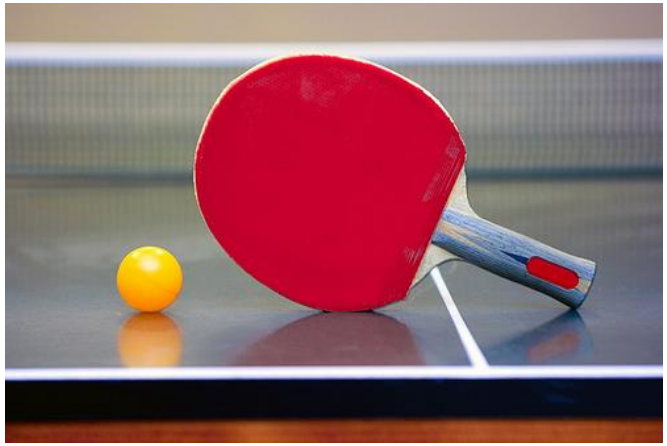
The moral

- JUnit does not support concurrent tests
  - You might get a false sense of security
- Concurrent clients beware…

Puzzler: “Ping Pong”

```
public class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread( () -> pong() );
        t.run();
        System.out.print("Ping");
    }

    private static synchronized void pong() {
        System.out.print("Pong");
    }
}
```



What does it print?

```
public class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread( () -> pong() );
        t.run();
        System.out.print("Ping");
    }

    private static synchronized void pong() {
        System.out.print("Pong");
    }
}
```

(a) PingPong

(b) PongPing

(c) It hangs

(c) None of the above

What does it print?

(a) `PingPong`

(b) `PongPing`

(c) It hangs

(d) None of the above

Not a multithreaded program!

Another look

```
public class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread( () -> pong() );
        t.run();  // An easy typo!
        System.out.print("Ping");
    }

    private static synchronized void pong() {
        System.out.print("Pong");
    }
}
```

How do you fix it?

```
public class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread( () -> pong() );
        t.start();
        System.out.print("Ping");
    }

    private static synchronized void pong() {
        System.out.print("Pong");
    }
}
```

Now prints PingPong

The moral

- Invoke `Thread.start`, not `Thread.run`
  - Can be very difficult to diagnose
- **This is a severe API design bug!**
- **`Thread` should not have implemented `Runnable`**
  - This confuses is-a and has-a relationships
  - `Thread`'s `runnable` should have been private
- `Thread` flagrantly violates the "Minimize accessibility" principle

Summary

- Concurrent programming can be hard to get right
  - Easy to introduce bugs even in simple examples
- Coming soon:
  - Higher-level abstractions for concurrency
  - Program structure for concurrency
  - Frameworks for concurrent computation