Principles of Software Construction: Objects, Design, and Concurrency

Part 3: Concurrency

Introduction to concurrency: Concurrency challenges

Charlie Garrod Chris Timperley

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 5a due 9 a.m. tomorrow
- Midterm exam available on Gradescope
  - Regrade requests due Monday, 18 November
- Reading due today:
  - Java Concurrency in Practice, Sections 11.3 and 11.4

Winter is coming discussion

Key concepts from last Tuesday

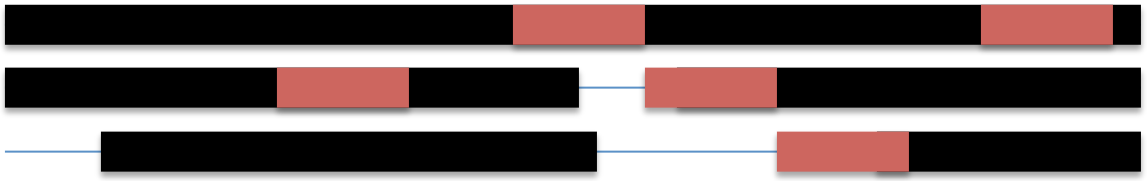
A concurrency bug with an easy fix:

```
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 synchronized 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



Atomicity

- 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
- In Java, integer increment is not atomic

```
i++;
```

is actually

1. Load data from variable `i`
2. Increment data by `1`
3. Store data to variable `i`

Yet another example: 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(42);
        stopRequested = true;
    }
}
```

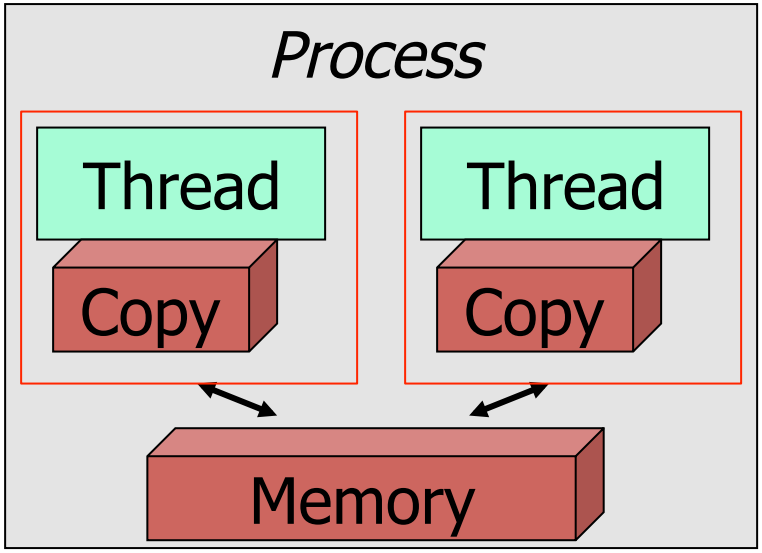
What went wrong?

- 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:

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

becomes:

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



Today

- Midterm exam 2 recap
- More basic concurrency in Java
  - Some challenges of concurrency
- Concurrency puzzlers
- Still coming soon:
  - Higher-level abstractions for concurrency
  - Program structure for concurrency
  - Frameworks for concurrent computation

A liveness problem: poor performance

```
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 synchronized 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 synchronized long balance() {
        return balance;
    }
}
```

A proposed fix?: *lock splitting*

```
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…

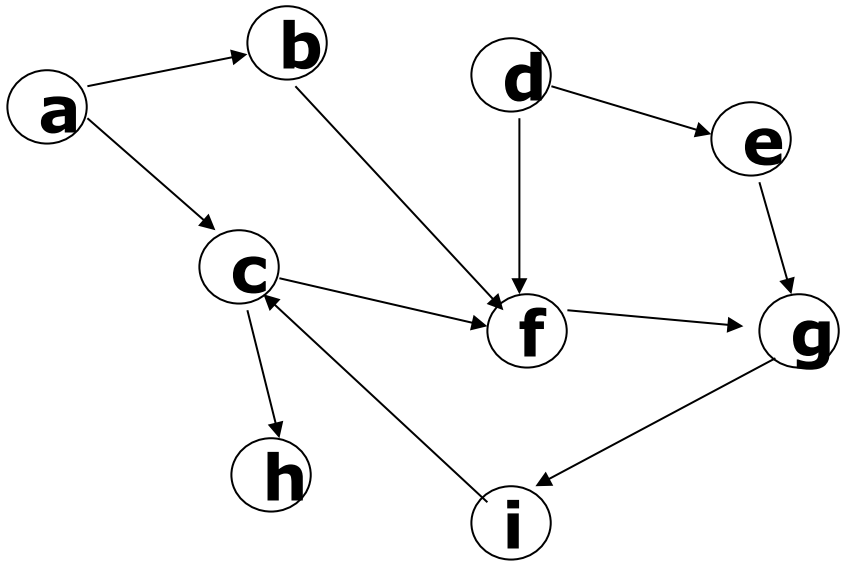
A liveness problem: deadlock

```
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;
            }
        }
    }
    …
}
```

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 iff 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 = first == source ? 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 = first == source ? dest : source;
    synchronized (first) {
        synchronized (second) {
            source.balance -= amount;
            dest.balance += amount;
        }
    }
  } …
```

An easy fix: Use a private lock

```
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 = first == source ? dest : source;
    synchronized (first.lock) {
        synchronized (second.lock) {
            source.balance -= amount;
            dest.balance += amount;
        }
    }
  } …
```

Concurrency and information hiding

- Encapsulate an object's state: Easier to implement invariants
  - Encapsulate synchronization: Easier to implement synchronization policy

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 = first == source ? dest : source;
    synchronized (first.lock) {
        synchronized (second.lock) {
            source.balance -= amount;
            dest.balance += amount;
        } …
```

An aside: Java Concurrency in Practice annotations

- `@ThreadSafe`
- `@NotThreadSafe`
- `@GuardedBy`
- `@Immutable`

Today

- Midterm exam 2 recap
- More basic concurrency in Java
  - Some challenges of concurrency
- Concurrency puzzlers
- Still coming soon:
  - Higher-level abstractions for concurrency
  - Program structure for concurrency
  - Frameworks for concurrent computation

Puzzler: “Racy Little Number”

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(2, number);
       });
       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(2, number);
       });
       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(2, number); // JUnit never sees the 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;
    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 well-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 varies

What does it print?

(a) `PingPong`

(b) `PongPing`

(c) It varies

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`
- `java.lang.Thread` should not have implemented `Runnable`

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