Principles of Software Construction: Objects, Design, and Concurrency

Part 3: Concurrency

Introduction to concurrency, part 2

Concurrency primitives and challenges, continued

Charlie Garrod Bogdan Vasilescu

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 5a due 9 a.m. tomorrow
- 2nd midterm exam returned today
- Reading due today:
  - Java Concurrency in Practice, Sections 11.3 and 11.4

Design tools discussion

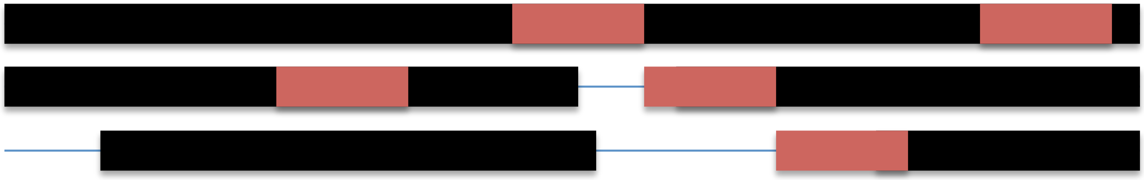
Key concepts from last Tuesday

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



Today

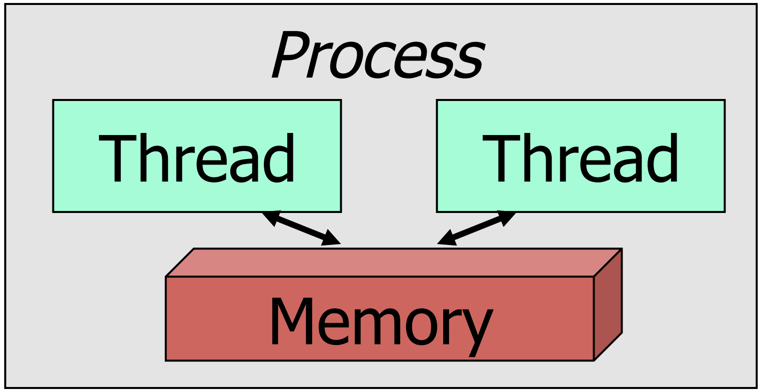
- Midterm exam 2 recap
- More basic concurrency in Java
  - Some challenges of concurrency
- Concurrency puzzlers
- Still coming soon:
  - Higher-level abstractions for concurrency
  - Program structure for concurrency
  - Frameworks for concurrent computation

Another example: serial number generation

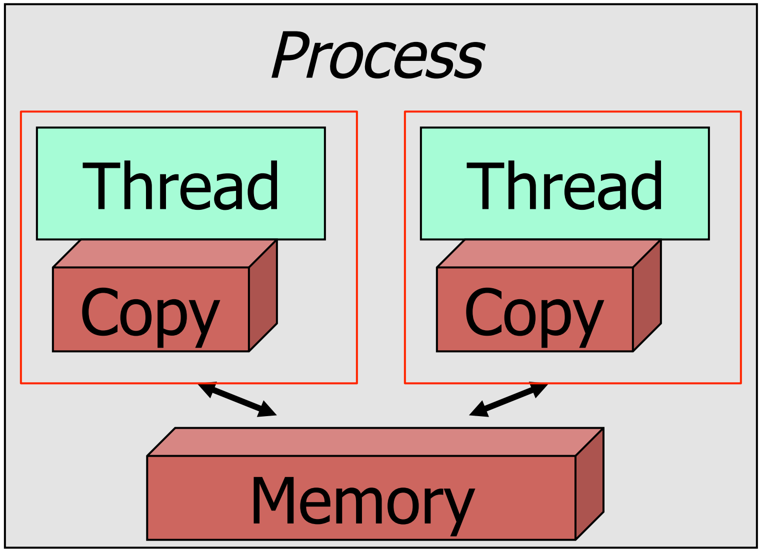
```
public class SerialNumber {
    private static long nextSerialNumber = 0;
    public static long generateSerialNumber() {
        return nextSerialNumber++;
    }

    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());
    }
}
```

Aside: Hardware abstractions

- Supposedly:
  - Thread state shared in memory



- A (slightly) more accurate view:
  - Separate state stored in registers and caches, even if shared



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`

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());
    }
}
```

Some actions are atomic

Precondition:

```
int i = 7;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

- What are the possible values for `ans`?

Some actions are atomic

Precondition:

```
int i = 7;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

- What are the possible values for `ans`?

i: 00000…00000111

⋮

i: 00000…00101010

Some actions are atomic

Precondition:

```
int i = 7;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

- What are the possible values for `ans`?

```
i: 00000…00000111
⋮
i: 00000…00101010
```

- In Java:
  - Reading an `int` variable is atomic
  - Writing an `int` variable is atomic
  - Thankfully, `ans: 00000…00101111` is not possible

Bad news: some simple actions are not atomic

- Consider a single 64-bit `long` value

high bits	low bits

  - Concurrently:
    - Thread A writing high bits and low bits
    - Thread B reading high bits and low bits

Precondition:

```
long i = 10000000000;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

`ans:` **01001…00000000** (10000000000)

`ans:` **00000…00101010** (42)

`ans:` **01001…00101010** (10000000042 or …)

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;
    }
}
```

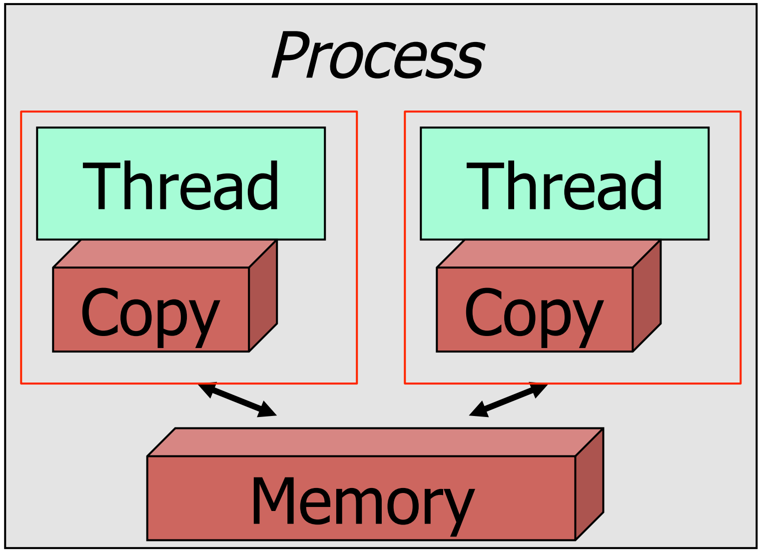
What went wrong?

- 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 */ ;
```



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(42);
        requestStop();
    }
}
```

A better(?) solution

```
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(42);
        stopRequested = true;
    }
}
```

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

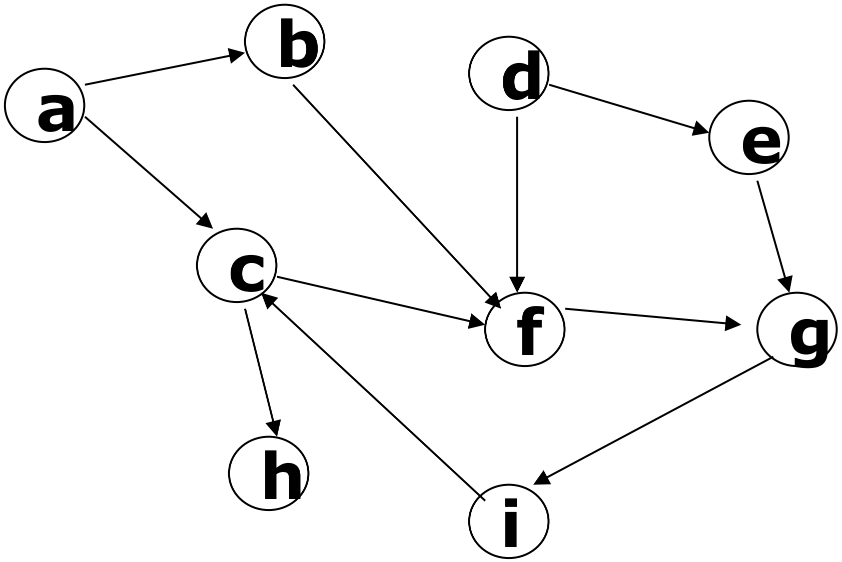
- A possible interleaving of operations:
  - `bugsThread` locks the `daffy` account
  - `daffyThread` locks the `bugs` account
  - `bugsThread` waits to lock the `bugs` account…
  - `daffyThread` waits to lock the `daffy` account…

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;
        }
    }
  } …
```


institute for SOFTWARE RESEARCH

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

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