Principles of Software Construction: Objects, Design, and Concurrency

Concurrency: Motivation and Primitives

Christian Kästner **Bogdan Vasilescu**

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia (1)

- Signup procedure and deadline for HW5a
  - Teams of 2 or 3. Form your team and sign up for a presentation time by Thursday, Mar 30, 11:59pm.
    - You may utilize the "Search for Teammates" thread @5 to help you find teammates.
    - Stick around after class today if you don't have partners yet.
    - Two places to sign up: Google Sheet & GitHub repo. See @652
  - Short presentation (max 10 min, 6 slides or fewer) in recitation on Wednesday, April 5 in front of your classmates.
    - Goal: illustrate how you achieve reuse in a domain
    - Describe domain, examples of plugins, decisions regarding generality vs specificity, overall project structure (e.g., how are plugins loaded), plugin interfaces
    - Similar to design review sessions

Administrivia (2)

- Second midterm, Thursday Mar 30 in class.
- **Midterm review session today 7:30pm in GHC 4401**.
- Concurrency not tested on the midterm
  - But everything in the course including readings is fair game
  - We will focus on the middle part of the course and the things that you had more chances to practice
    - e.g. more UML/design than API design

Administrivia (3)

- Good discussion on Piazza about the Reading Quiz question 2 (Chapter 6 of "Beautiful Code"):
- Q: "What are some of the mentioned mechanisms that can help ensure backward-compatibility?"
  - *A: Using design patterns*
  - A: Using interfaces
  - A: Controlling visibility
- A: Using design patterns:
  - Book: "Provide well-defined 'hook points' that permit extensibility in the places where you intend it to occur."
  - Example of how the Observer pattern can be used to provide such hook points

Administrivia (4)

- Commit messages are (one of) your primary means of communication with the rest of the team.
  - This will become more obvious in HW5.

HW4b; Oops forgot to save. (Also bus is here)

Woke up and dreamt of some bugs. They were there.

HW 4b update (...kill me)

dropped my laptop, then I banged it on a table. Was reminded of impor...

Intro to Java

Git, CI

UML

Static Analysis

Performance

GUIs

GUIs

More Git

Design

Part 1: Design at a Class Level	Part 2: Designing (Sub)systems	Part 3: Designing Concurrent Systems
Design for Change: Information Hiding, Contracts, Design Patterns, Unit Testing	Understanding the Problem	Concurrency Primitives, Synchronization
Design for Reuse: Inheritance, Delegation, Immutability, LSP, Design Patterns	Responsibility Assignment, Design Patterns, GUI vs Core, Design Case Studies	Designing Abstractions for Concurrency
	Testing Subsystems	Distributed Systems in a Nutshell
	Design for Reuse at Scale: Frameworks and APIs	

Learning Goals

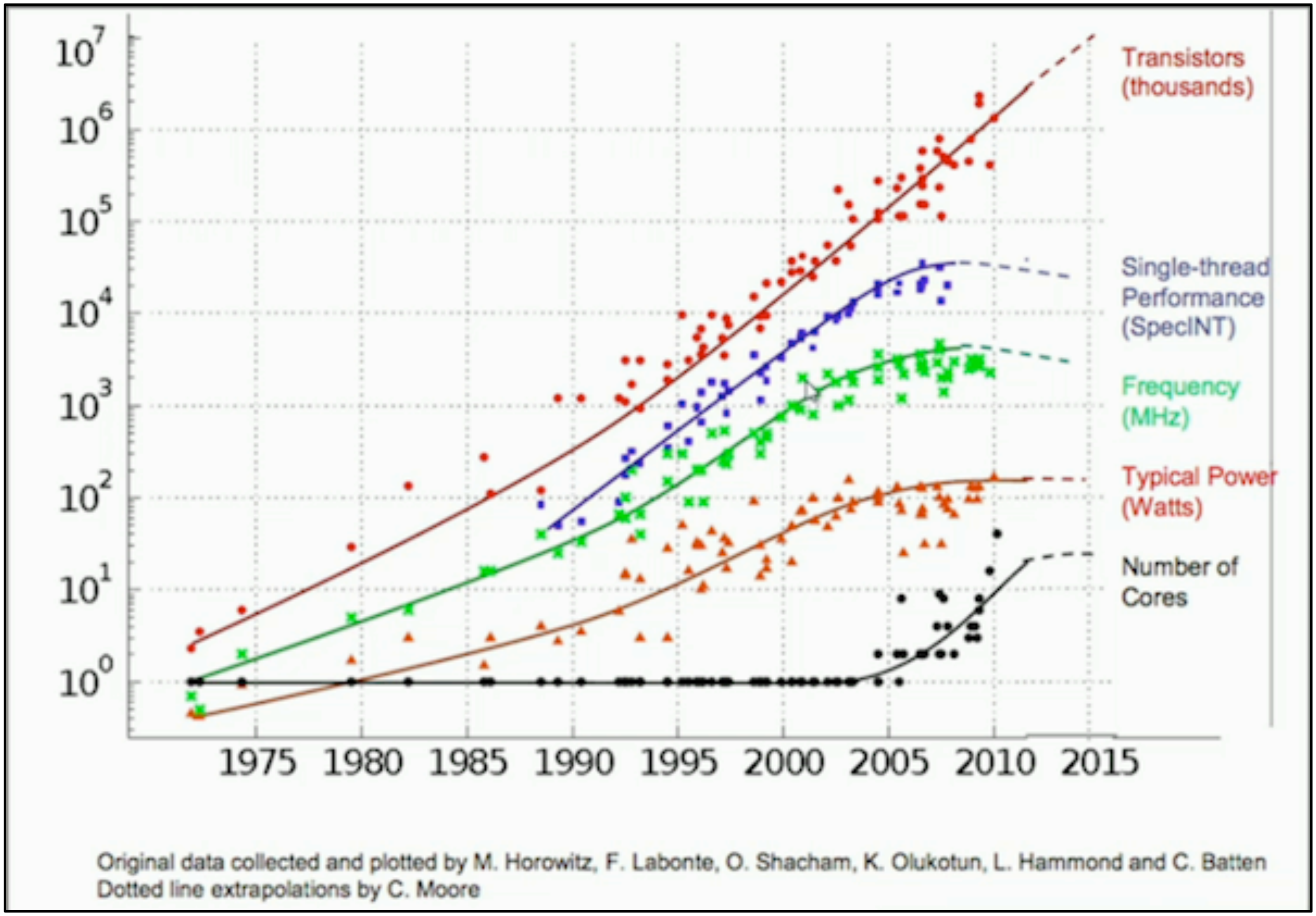
- Understand the motivation and different use cases for concurrency and parallelism
- Understand concurrency risks: safety, liveness, performance
- Understand and use Java primitives for concurrency: threads, synchronization, volatile, wait/notify

WHY CONCURRENCY

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 × 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 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

Benefits of Threads (1)

- Exploiting Multiple Processors
  - All CPUs today are multi-core
  - But basic unit of scheduling is a thread
    - A single-threaded program running on a 100-processor system is giving up access to 99% of the available CPU resources
  - Also, better throughput on single-processor systems:
    - Single-threaded program needs to wait for synchronous I/O operation to complete
    - Multi-threaded program can do something else during the blocking I/O
- Responsive User Interfaces
  - AWT, Spring have separate *event dispatch thread*
  - Long-running tasks (e.g., spell checking) can be executed in separate thread

Benefits of Threads (2)

- Simplicity of Modeling
  - Separating tasks & assigning a separate thread to each
  - Abstracting common infrastructure, as request management, load balancing, ... in concurrency frameworks
- Simplified Handling of Asynchronous Events
  - Async vs sync I/O: server that accepts socket connections from multiple clients; client `read` blocks until data is available
  - Avoiding "callback hell" (JavaScript)

Aside: JavaScript “Callback Hell”

- You don’t want the program to pause (block) while waiting for download to finish

```
var photo = downloadPhoto('http://coolcats.com/cat.gif')
// photo is 'undefined'!
```

- Store the code that should run after the download is complete in a “callback” function

```
2  downloadPhoto('http://coolcats.com/cat.gif', handlePhoto)

1  function handlePhoto (error, photo) {
     if (error) console.error('Download error!', error)
     else console.log('Download finished', photo)      4
   }

3  console.log('Download started')
```

Aside: JavaScript "Callback Hell"

- Callbacks can get out of hand

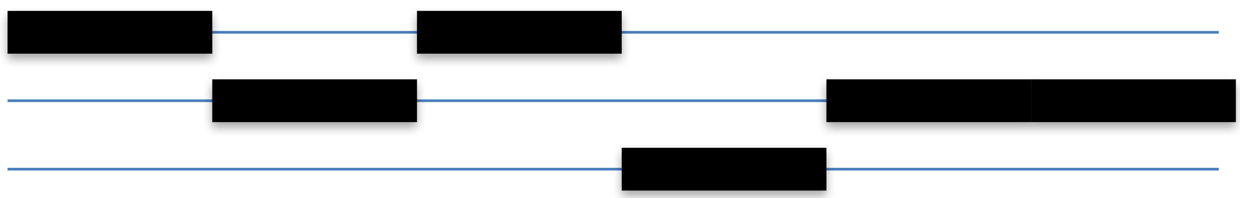
```
getData(function(a){
    getMoreData(a, function(b){
        getMoreData(b, function(c){
            getMoreData(c, function(d){
                getMoreData(d, function(e){
                    ...
                });
            });
        });
    });
});
```


From: http://stackabuse.com/avoiding-callback-hell-in-node-js/

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

Concurrency vs Parallelism



Safety, Liveness, Performance

CONCURRENCY HAZARDS

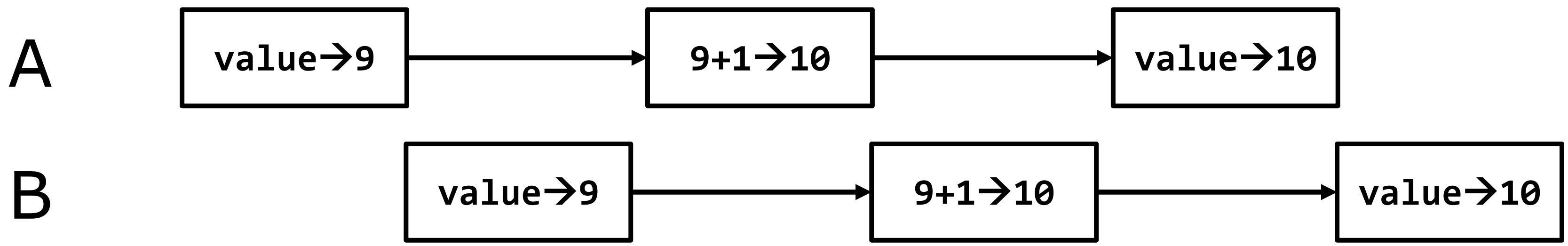
Safety Hazard

- The ordering of operations in multiple threads is **unpredictable**.

```
@NotThreadSafe
public class UnsafeSequence {
    private int value;

    public int getNext() {
        return value++;
    }
}
```

Not atomic

- Unlucky execution of `UnsafeSequence.getNext`



Thread Safety

A class is thread safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.

Liveness Hazard

- Safety: “nothing bad ever happens”
- Liveness: “something good eventually happens”

- Deadlock
  - Infinite loop in sequential programs
  - Thread A waits for a resource that thread B holds exclusively, and B never releases it → A will wait forever
    - E.g., Dining philosophers

- Elusive: depend on relative timing of events in different threads

Deadlock example

- Two threads: A does transfer(a, b, 10); B does transfer(b, a, 10)

```
class Account {
  double balance;

  void withdraw(double amount){ balance -= amount; }

  void deposit(double amount){ balance += amount; }

  void transfer(Account from, Account to, double amount){
        synchronized(from) {
            from.withdraw(amount);
            synchronized(to) {
                to.deposit(amount);
            }
        }
  }
}
```

Execution trace:
A: lock a (v)
B: lock b (v)
A: lock b (x)
B: lock a (x)
A: wait
B: wait

Deadlock!

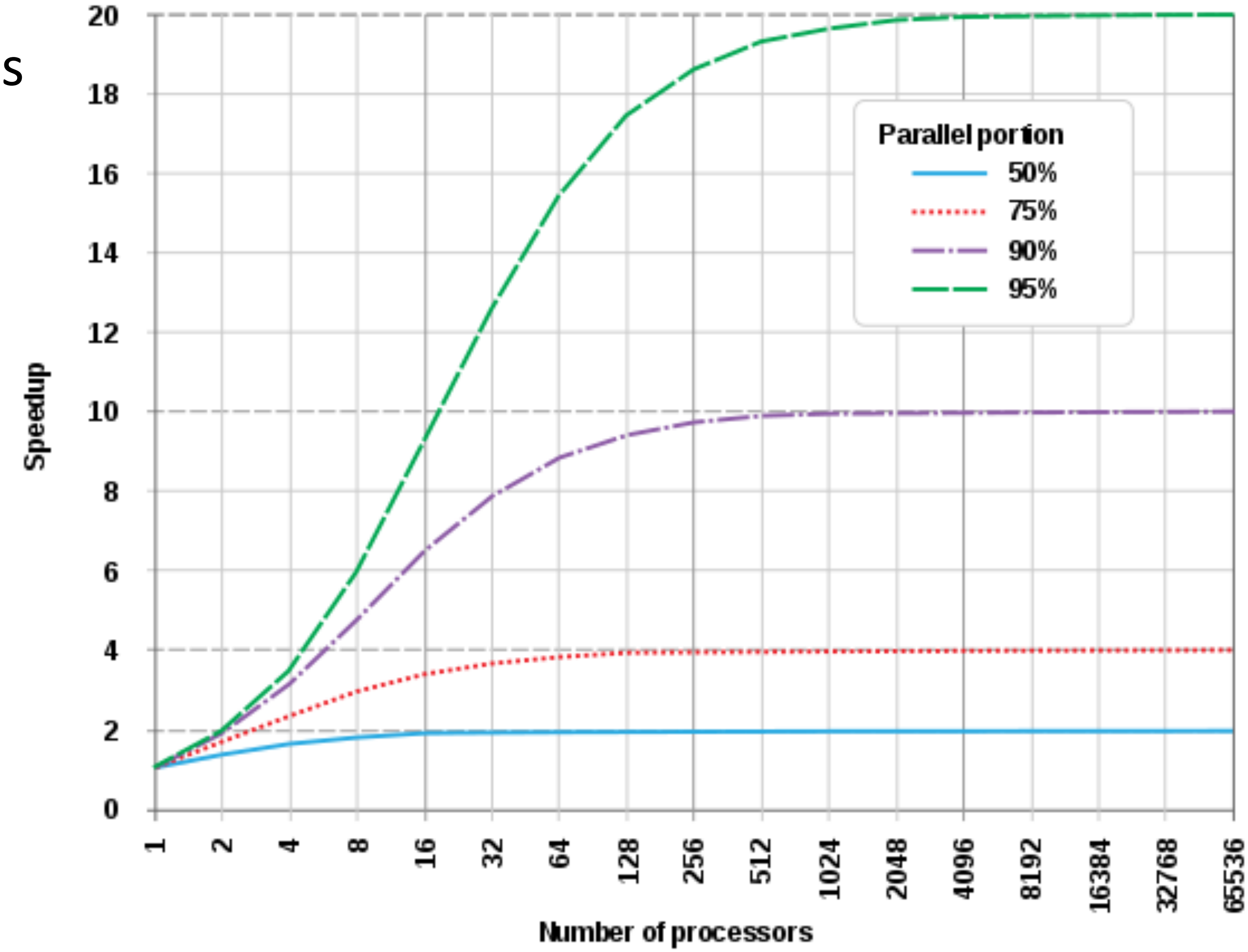
Performance Hazard

- Liveness: “something good eventually happens”
- Performance: we want something good to happen quickly

- Multi-threading involves runtime overhead:
  - Coordinating between threads (locking, signaling, memory sync)
  - Context switches
  - Thread creation & teardown
  - Scheduling
- Not all problems can be solved faster with more resources
  - One mother delivers a baby in 9 months

Amdahl's law

- The speedup is limited by the serial part of the program.



How fast can this run?

- N threads fetch independent tasks from a shared work queue

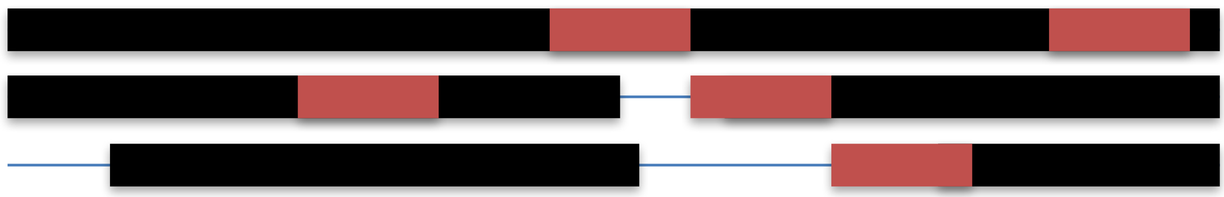
```
public class WorkerThread extends Thread {
    ...

    public void run() {
        while (true) {
            try {
                Runnable task = queue.take();
                task.run();
            } catch (InterruptedException e) {
                break; /* Allow thread to exit */
            }
        }
    }
}
```

JAVA PRIMITIVES: ENSURING VISIBILITY AND ATOMICITY

Synchronization for Safety

- If multiple threads access the same mutable state variable without appropriate synchronization, the program is broken.
- There are three ways to fix it:
  - Don't share the state variable across threads;
  - Make the state variable immutable; or
  - Use synchronization whenever accessing the state variable.

Exclusion



Synchronization allows parallelism while ensuring that certain segments are executed in isolation. Threads wait to acquire lock, may reduce performance.

Stateless objects are always thread safe

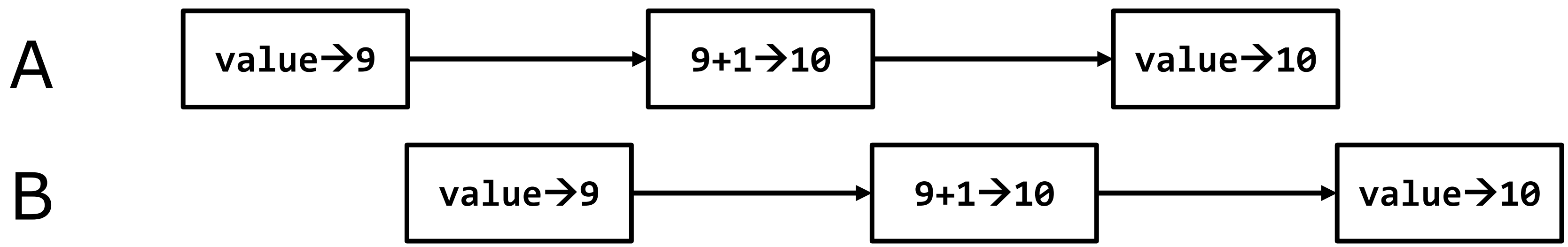
- Example: stateless factorizer
  - No fields
  - No references to fields from other classes
  - Threads sharing it cannot influence each other

```
@ThreadSafe
public class StatelessFactorizer implements Servlet {

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        encodeIntoResponse(resp, factors);
    }
}
```

Is this thread safe?

```java
public class CountingFactorizer implements Servlet {
    private long count = 0;

    public long getCount() { return count; }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        ++count;
        encodeIntoResponse(resp, factors);
    }
}
```

Non atomicity and thread (un)safety




```java
@NotThreadSafe
public class UnsafeCountingFactorizer implements Servlet {
    private long count = 0;

    public long getCount() { return count; }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        ++count;
        encodeIntoResponse(resp, factors);
    }
}
```

Non atomicity and thread (un)safety

- Stateful factorizer
  - Susceptible to *lost updates*
  - The ++count operation is not atomic (read-modify-write)

```
@NotThreadSafe
public class UnsafeCountingFactorizer implements Servlet {
    private long count = 0;

    public long getCount() { return count; }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        ++count;
        encodeIntoResponse(resp, factors);
    }
}
```

Enforcing atomicity: Intrinsic locks

- `synchronized(lock) { … }` synchronizes entire code block on object `lock`; cannot forget to unlock
- The `synchronized` modifier on a method is equivalent to `synchronized(this) { … }` around the entire method body
- Every Java object can serve as a lock
- At most one thread may own the lock (mutual exclusion)
  - `synchronized` blocks guarded by the same lock execute atomically w.r.t. one another

Fixing the stateful factorizer

```
@ThreadSafe
public class UnsafeCountingFactorizer
        implements Servlet {
    @GuardedBy(“this”)
    private long count = 0;

    public long getCount() {
        synchronized(this){
            return count;
        }
    }

    public void service(ServletRequest req,
                        ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        synchronized(this) {
            ++count;
        }
        encodeIntoResponse(resp, factors);
    }
}
```

For each mutable state variable that may be accessed by more than one thread, **all** accesses to that variable must be performed with the **same** lock held. In this case, we say that the variable is **guarded by** that lock.

Fixing the stateful factorizer

```
@ThreadSafe
public class UnsafeCountingFactorizer
        implements Servlet {
    @GuardedBy("this")
    private long count = 0;

    public synchronized long getCount() {
            return count;
    }

    public void service(ServletRequest req,
                        ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        synchronized(this) {
            ++count;
        }
        encodeIntoResponse(resp, factors);
    }
}
```

For each mutable state variable that may be accessed by more than one thread, **all** accesses to that variable must be performed with the **same** lock held. In this case, we say that the variable is **guarded by** that lock.

Fixing the stateful factorizer

```
@ThreadSafe
public class UnsafeCountingFactorizer
        implements Servlet {
    @GuardedBy("this")
    private long count = 0;

    public synchronized long getCount() {
            return count;
    }

    public synchronized void service(
                        ServletRequest req,
                        ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
            ++count;
        encodeIntoResponse(resp, factors);
    }
}
```

For each mutable state variable that may be accessed by more than one thread, **all** accesses to that variable must be performed with the **same** lock held. In this case, we say that the variable is **guarded by** that lock.

What's the difference?

```
public synchronized void service(ServletRequest req,
                                 ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i);
        ++count;
    encodeIntoResponse(resp, factors);
}
```

```
public void service(ServletRequest req,
                    ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        synchronized(this) {
            ++count;
        }
        encodeIntoResponse(resp, factors);
    }
```

Private locks

```
@ThreadSafe
public class UnsafeCountingFactorizer
        implements Servlet {
    private final Object lock = new Object();
    @GuardedBy(“lock”)
    private long count = 0;

    public long getCount() {
        synchronized(lock){
            return count;
        }
    }

    public void service(ServletRequest req,
                        ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        synchronized(lock) {
            ++count;
        }
        encodeIntoResponse(resp, factors);
    }
}
```

For each mutable state variable that may be accessed by more than one thread, **all** accesses to that variable must be performed with the **same** lock held. In this case, we say that the variable is **guarded by** that lock.

Does this deadlock?

```java
public class Widget {
    public synchronized void doSomething() {...}
}

public class LoggingWidget extends Widget {
    public synchronized void doSomething() {
        System.out.println(toString() + ": calling doSomething");
        super.doSomething();
    }
}
```

No: Intrinsic locks are reentrant

- A thread can lock the same object again while already holding a lock, i.e., a synchronized method can call another synchronized method in the same object

```
public class Widget {
    public synchronized void doSomething() {...}
}

public class LoggingWidget extends Widget {
    public synchronized void doSomething() {
        System.out.println(toString() + ": calling doSomething");
        super.doSomething();
    }
}
```

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(5);
        stopRequested = true;
    }
}
```

What could have gone 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 */ ;
```

How do you fix it?

```
public class StopThread {
    @GuardedBy(“StopThread.class”)
    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(5);
        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;
    }
}
```

Volatile keyword

- Tells compiler and runtime that variable is shared and operations on it should not be reordered with other memory ops
  - A read of a volatile variable **always returns the most recent write by any thread**

- Volatile is **not a substitute for synchronization**
  - Volatile variables can only guarantee visibility
  - Locking can guarantee both visibility and atomicity

Summary: Synchronization

- **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++`)

JAVA PRIMITIVES: WAIT, NOTIFY, AND TERMINATION

Guarded methods

- What to do on a method if the precondition is not fulfilled (e.g., transfer money from bank account with insufficient funds)
  - throw exception (**balking**)
  - wait until precondition is fulfilled (**guarded suspension**)
  - wait and timeout (combination of balking and guarded suspension)

Guarded suspension

- Block execution until a given condition is true
- For example,
  - pull element from queue, but wait on an empty queue
  - transfer money from bank account as soon sufficient funds are there
- Blocking as (often simpler) alternative to callback

Monitor Mechanics in Java

- Object.wait() – suspends the current thread's execution, releasing locks
- Object.wait(timeout) – suspends the current thread's execution for up to *timeout* milliseconds
- Object.notify() – resumes one of the waiting threads
- See documentation for exact semantics

Monitor Example

```
class SimpleBoundedCounter {
  protected long count = MIN;
  public synchronized long count() { return count; }
  public synchronized void inc() throws InterruptedException {
        awaitUnderMax(); setCount(count + 1);
  }
  public synchronized void dec() throws InterruptedException {
        awaitOverMin(); setCount(count - 1);
  }
  protected void setCount(long newValue) { // PRE: lock held
        count = newValue;
        notifyAll(); // wake up any thread depending on new value
  }
  protected void awaitUnderMax() throws InterruptedException {
        while (count == MAX) wait();
  }
  protected void awaitOverMin() throws InterruptedException {
        while (count == MIN) wait();
  }
}
```

Never invoke wait outside a loop!

- Loop tests condition before and after waiting
- Test before skips `wait` if condition already holds
  - Necessary to ensure **liveness**
  - Without it, thread can wait forever!
- Testing after `wait` ensure **safety**
  - Condition may not be true when thread wakens
  - If thread proceeds with action, it can destroy invariants!

All of your waits should look like this

```
synchronized (obj) {
    while (<condition does not hold>) {
        obj.wait();
    }

    ... // Perform action appropriate to condition
}
```

Why can a thread wake from a `wait` when condition does not hold?

- Another thread can slip in between `notify` & wake
- Another thread can invoke `notify` accidentally or maliciously when condition does not hold
  - This is a flaw in java locking design!
  - Can work around flaw by using private lock object
- Notifier can be liberal in waking threads
  - Using `notifyAll` is good practice, but causes this
- Waiting thread can wake up without a `notify`(!)
  - Known as a *spurious wakeup*

Interruption

- Difficult to kill threads once started, but may politely ask to stop (thread.interrupt())
- Long-running threads should regularly check whether they have been interrupted
- Threads waiting with wait() throw exceptions if interrupted
- Read documentation

```
public class Thread {
    public void interrupt() { ... }
    public boolean isInterrupted() { ... }
    ...
}
```

Interruption Example

```
class PrimeProducer extends Thread {
    private final BlockingQueue<BigInteger> queue;
    PrimeProducer(BlockingQueue<BigInteger> queue) {
        this.queue = queue;
    }
    public void run() {
        try {
            BigInteger p = BigInteger.ONE;
            while (!Thread.currentThread().isInterrupted())
                queue.put(p = p.nextProbablePrime());
        } catch (InterruptedException consumed) {
            /* Allow thread to exit */
        }
    }
    public void cancel() { interrupt(); }
}
```

For details, see Java Concurrency In Practice, Chapter 7

BUILDING HIGHER LEVEL CONCURRENCY MECHANISMS

Beyond Java Primitives

- Java Primitives (synchronized, wait, notify) are low level mechanisms
- For most tasks better higher-level abstractions exist
- Writing own abstractions is possible, but potentially dangerous – use libraries written by experts

Example: read-write locks (API)

Also known as shared/exclusive mode locks

```
private final RwLock lock = new RwLock();

lock.readLock();
try {
    // Do stuff that requires read (shared) lock
} finally {
    lock.unlock();
}

lock.writeLock();
try {
    // Do stuff that requires write (exclusive) lock
} finally {
    lock.unlock();
}
```

Example: read-write locks (Impl. 1/2)

```
public class RwLock {
    // State fields are protected by RwLock's intrinsic lock

    /** Num threads holding lock for read. */
    private int numReaders = 0;

    /** Whether lock is held for write. */
    private boolean writeLocked = false;

    public synchronized void readLock() throws InterruptedException {
        while (writeLocked) {
            wait();
        }
        numReaders++;
    }
```

Example: read-write locks (Impl. 2/2)

```
    public synchronized void writeLock() throws InterruptedException {
        while (numReaders != 0 || writeLocked) {
            wait();
        }
        writeLocked = true;
    }

    public synchronized void unlock() {
        if (numReaders > 0) {
            numReaders--;
        } else if (writeLocked) {
            writeLocked = false;
        } else {
            throw new IllegalStateException("Lock not held");
        }
        notifyAll(); // Wake any waiters
    }
}
```

Caveat: `RwLock` is just a toy!

- It has poor fairness properties
  - Readers can starve writers!
- `java.util.concurrent` provides an industrial strength `ReadWriteLock`
- More generally, avoid `wait`/`notify`
  - In the early days it was all you had
  - Nowadays, higher level concurrency utils are better

Summary

- Concurrency for exploiting multiple processors, simplifying modeling, simplifying asynchronous events
- Safety, liveness and performance hazards matter
- Synchronization on any Java object; volatile ensures visibility
- Wait/notify for guards, interruption for cancelation – building blocks for higher level abstractions

Recommended Readings

- Goetz et al. Java Concurrency In Practice. Pearson Education, 2006, Chapters 1-2
- Lea, Douglas. Concurrent programming in Java: design principles and patterns. Addison-Wesley Professional, 2000.