Principles of Software Construction: Objects, Design, and Concurrency

Concurrency Part I: Primitives

Michael Hilton **Bogdan Vasilescu**

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

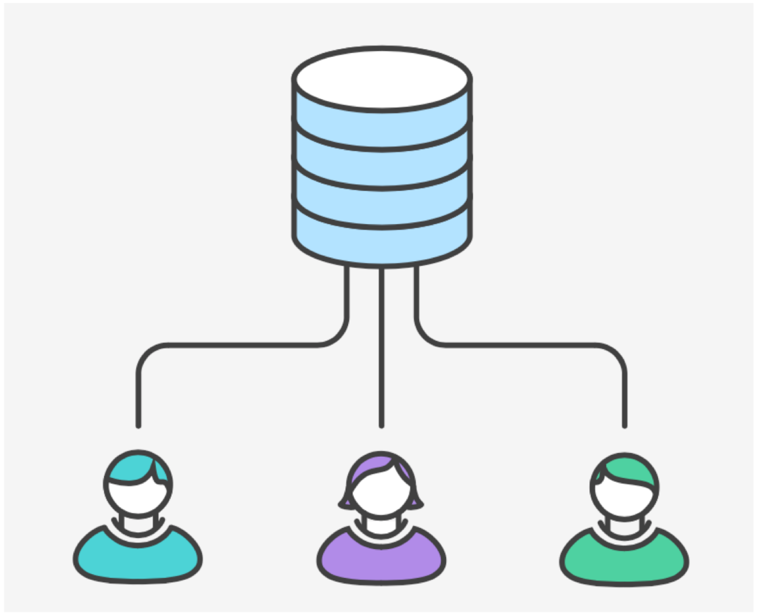
Administrivia

- HW 5b due Tuesday April 9th
- No class on Thursday next week

Last Tuesday

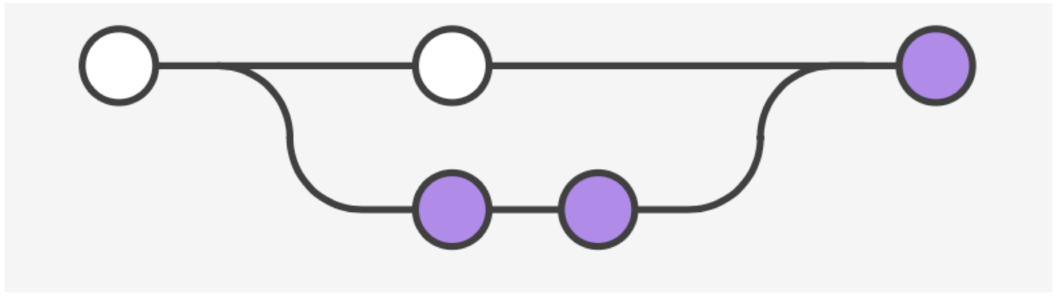
1. Centralized workflow

- Central repository to serve as the single point-of-entry for all changes to the project
- Default development branch is called master
  - all changes are committed into master
  - doesn't require any other branches

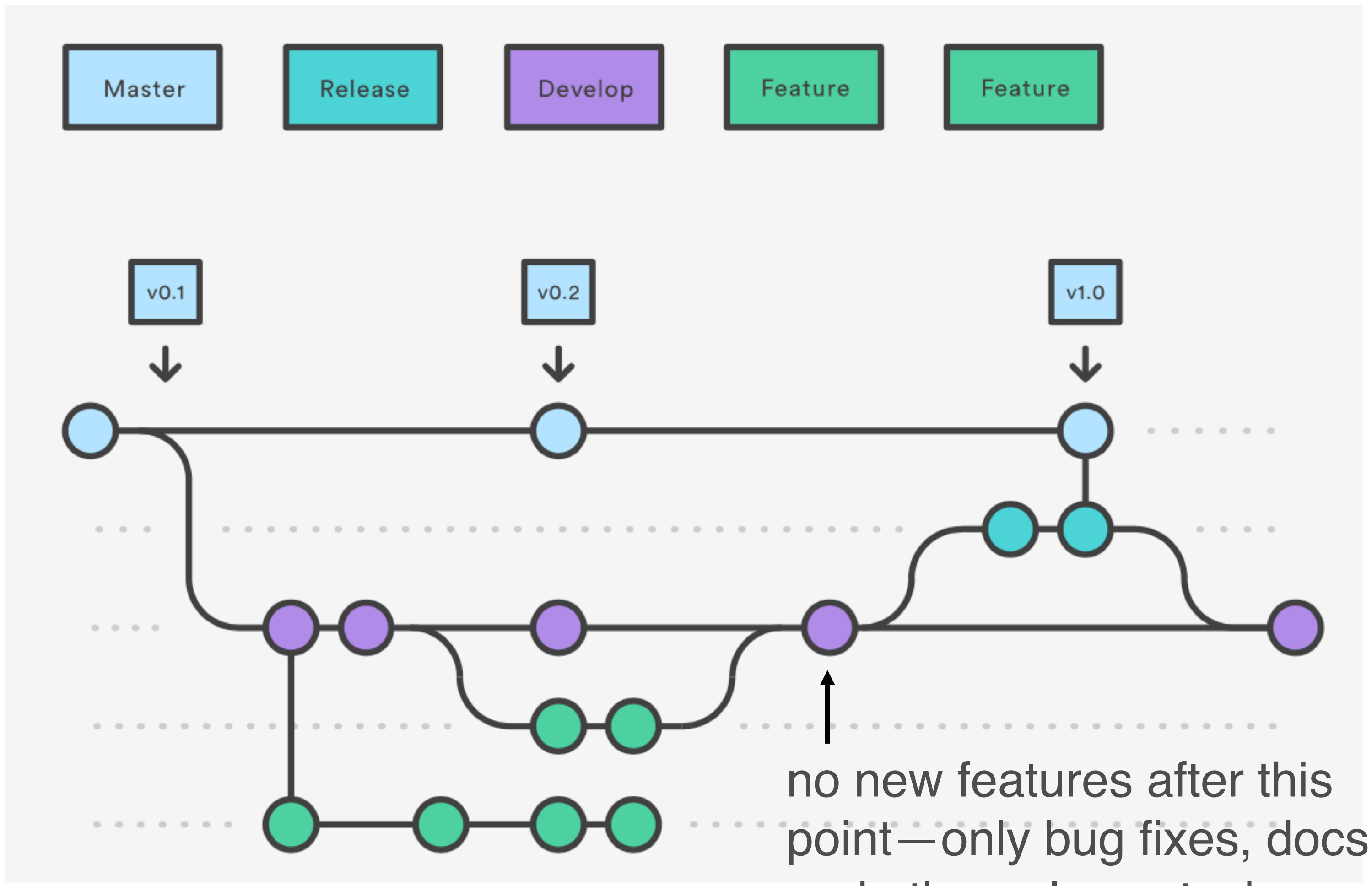


2. Git Feature Branch Workflow

- *All* feature development should take place in a dedicated branch instead of the master branch
- Multiple developers can work on a particular feature without disturbing the main codebase
  - master branch will never contain broken code (enables CI)
  - Enables pull requests (code review)



3. GitFlow release branches



no new features after this point—only bug fixes, docs, and other release tasks

Today: Concurrency, motivation and primitives

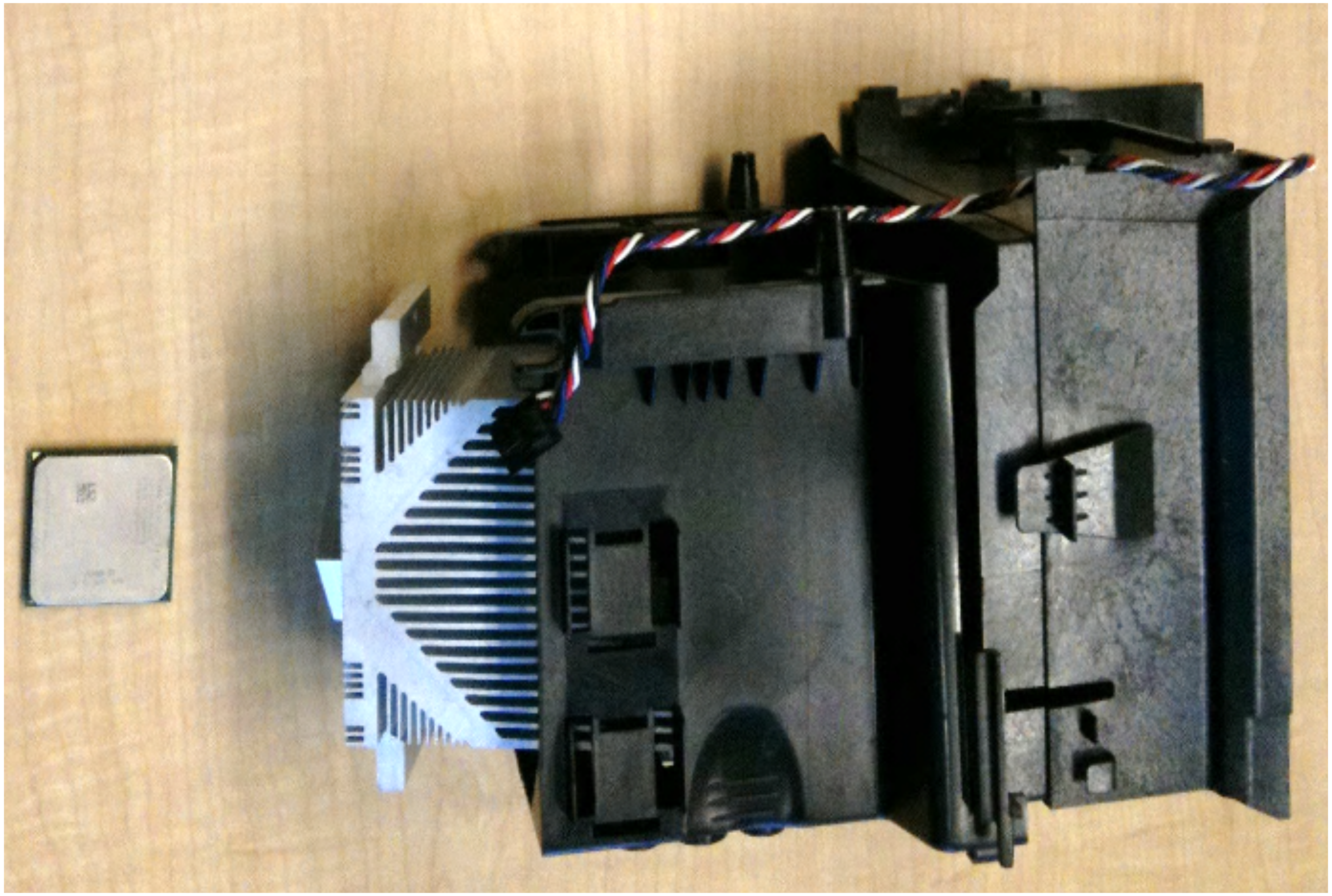
- The backstory
  - Motivation, goals, problems, …
- Basic concurrency in Java
- Later (probably not today):
  - Higher-level abstractions for concurrency
  - Program structure for concurrency
  - Frameworks for concurrent computation

Power requirements of a CPU

- Approx.: **C**apacitance * **V**oltage^{2} * **F**requency
- To increase performance:
  - More transistors, thinner wires
    - More power leakage: increase **V**
  - Increase clock frequency **F**
    - Change electrical state faster: increase **V**
- *Dennard scaling*: As transistors get smaller, power density is approximately constant…
  - …until early 2000s
- Heat output is proportional to power input

One option: fix the symptom

- Dissipate the heat

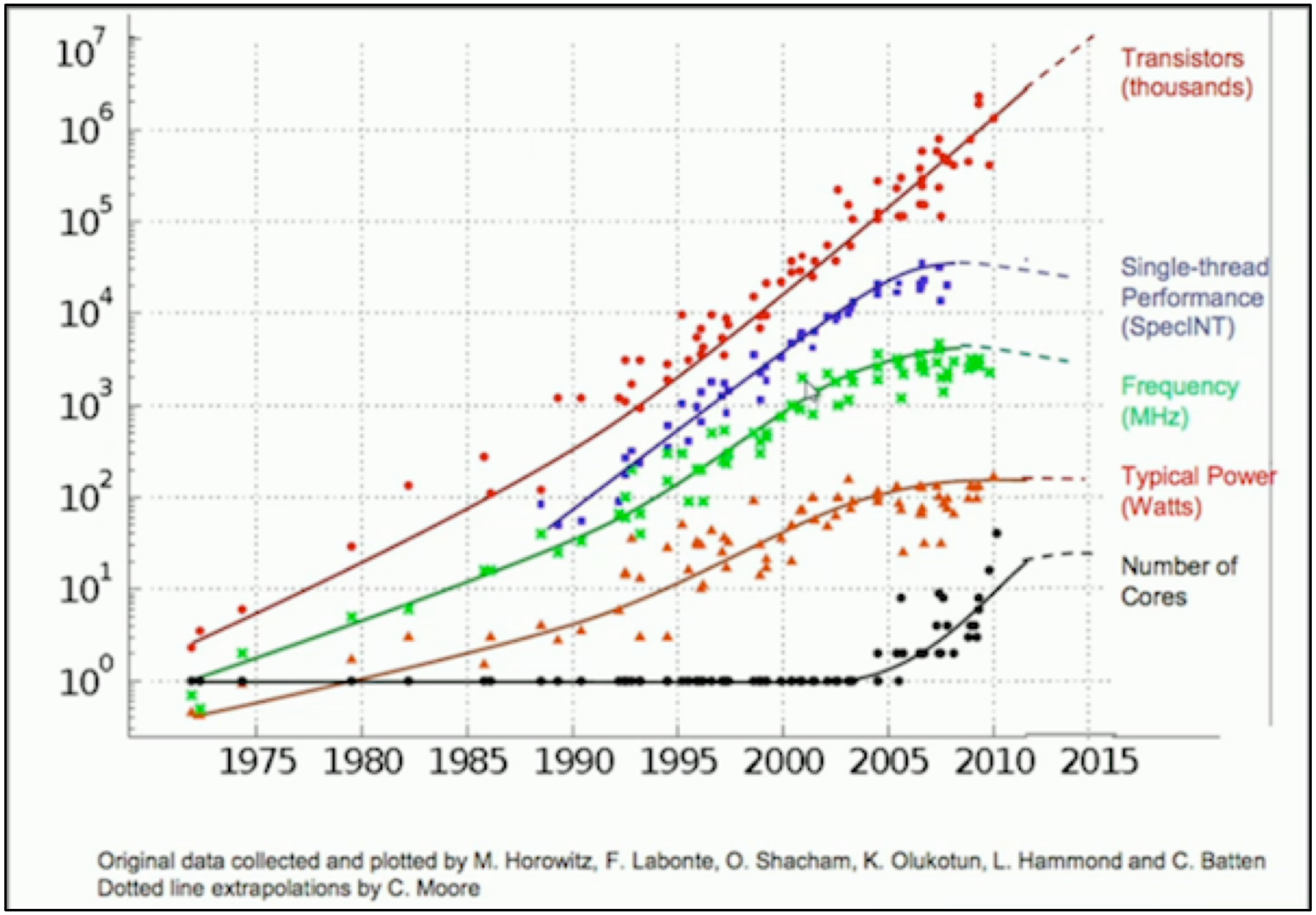


One option: fix the symptom

- Better: Dissipate the heat with liquid nitrogen
  - Overclocking by Tom's Hardware's 5 GHz project

http://www.tomshardware.com/reviews/5-ghz-project,731-8.html

Processor characteristics over time



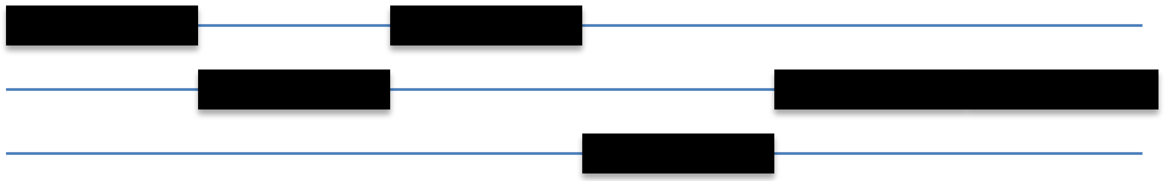
Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Concurrency then and now

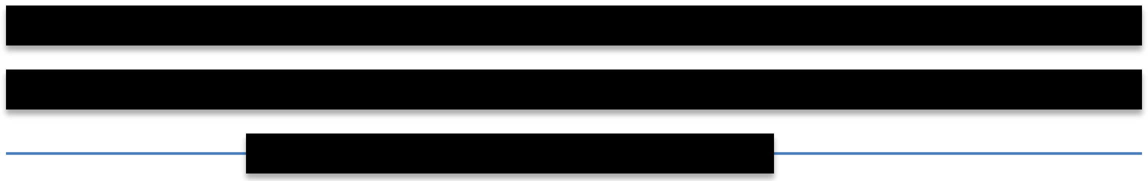
- In the past, multi-threading just a convenient abstraction
  - GUI design: event dispatch thread
  - Server design: isolate each client's work
  - Workflow design: isolate producers and consumers
- Now: required for scalability and performance

Aside: Concurrency vs. parallelism, visualized

- Concurrency without parallelism:



- Concurrency with parallelism:



Basic concurrency in Java

- An interface representing a task

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

- A class to execute a task in a thread

```
public class Thread {
    public Thread(Runnable task);
    public void start();
    public void join();
    …
}
```

makes sure that thread is terminated before the next instruction is executed by the program

A simple threads example

```java
public interface Runnable {  // java.lang.Runnable
    public void run();
}

public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);  // Number of threads;

    Runnable greeter = new Runnable() {
        public void run() {
            System.out.println("Hi mom!");
        }
    };
    for (int i = 0; i < n; i++) {
        new Thread(greeter).start();
    }
}
```

A simple threads example

```
public interface Runnable {  // java.lang.Runnable
    public void run();
}

public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);  // Number of threads;

    Runnable greeter = () -> System.out.println("Hi mom!");
    for (int i = 0; i < n; i++) {
        new Thread(greeter).start();
    }
}
```

A simple threads example

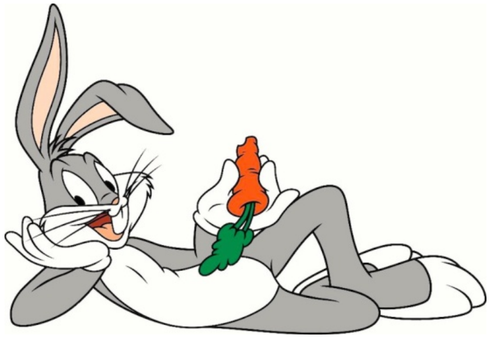
```
public interface Runnable {  // java.lang.Runnable
    public void run();
}

public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);  // Number of threads;

    for (int i = 0; i < n; i++) {
        new Thread(() -> System.out.println("Hi mom!")).start();
    }
}
```

Another 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;
    }
}
```

Another example: Money-grab (2)

```
public static void main(String[] args) throws InterruptedException {
    BankAccount bugs = new BankAccount(1_000_000);
    BankAccount daffy = new BankAccount(1_000_000);

    Thread bugsThread = new Thread(()-> {
        for (int i = 0; i < 1_000_000; i++)
            transferFrom(daffy, bugs, 1);
    });

    Thread daffyThread = new Thread(()-> {
        for (int i = 0; i < 1_000_000; i++)
            transferFrom(bugs, daffy, 1);
    });

    bugsThread.start(); daffyThread.start();
    bugsThread.join(); daffyThread.join();
    System.out.println(bugs.balance() - daffy.balance());
}
```



What went wrong?

- Daffy & Bugs threads had a *race condition* for shared data
  - Transfers did not happen in sequence
- Reads and writes interleaved randomly
  - Random results ensued

Safety, Liveness, Performance

CONCURRENCY HAZARDS

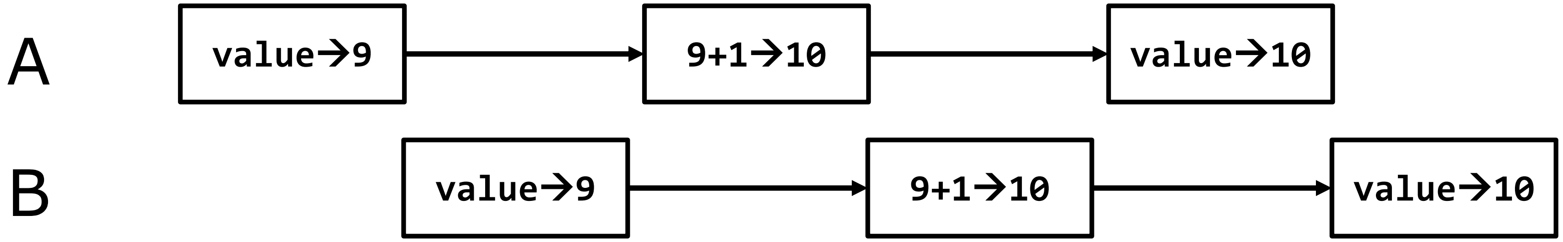
1. Safety Hazard

- 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`



Aside: 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`

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.

2. 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!

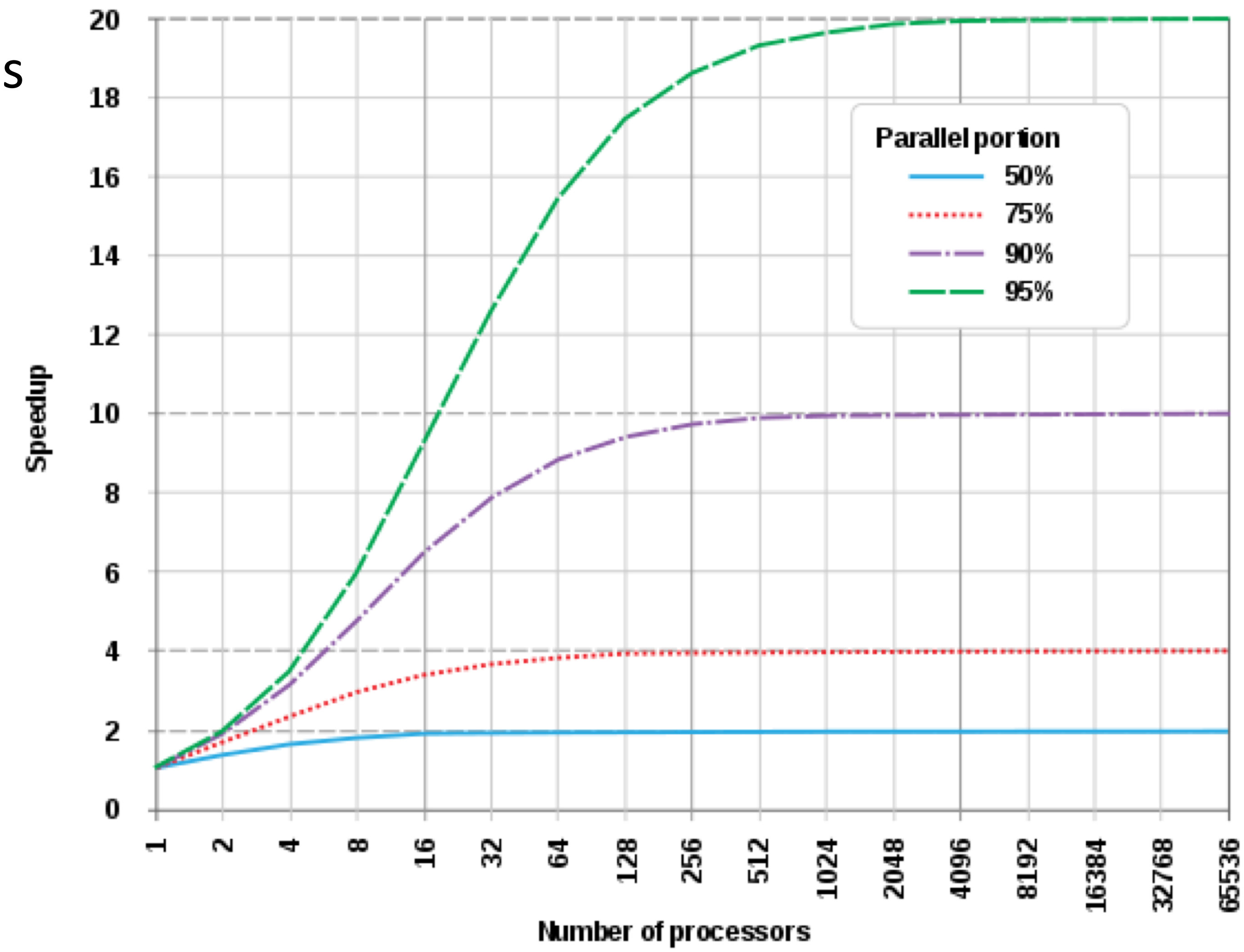
3. Performance Hazard

- 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

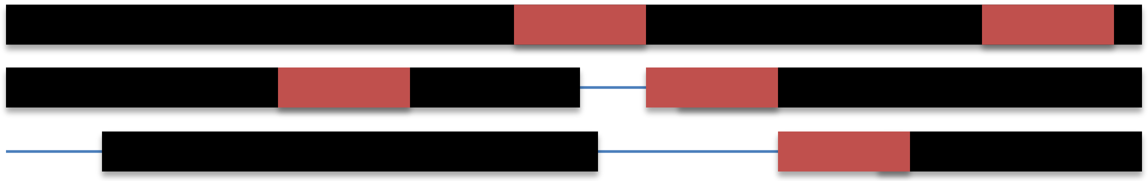
Synchronization for Safety

- 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.

An easy fix: Synchronized access (visibility)

```
@ThreadSafe
public class BankAccount {

     @GuardedBy(“this”)
    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;
    }
}
```

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);
    }
}
```

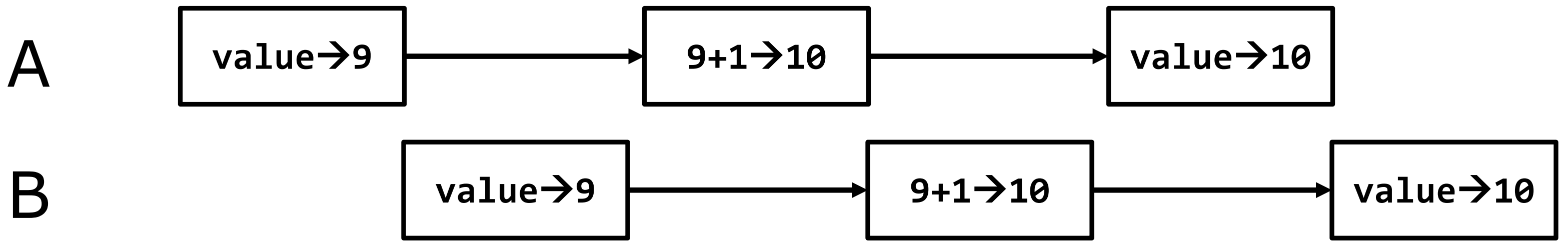
Is this thread safe?

```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);
    }
}
```

Is this thread safe?

```
@NotThreadSafe
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



```
@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);
   }
```

To be continued …