Principles of Software Construction: Objects, Design, and Concurrency

Part 3: Concurrency

Introduction to concurrency

Charlie Garrod Chris Timperley

Carnegie Mellon University
School of Computer Science

institute for
SOFTWARE
RESEARCH

Administrivia

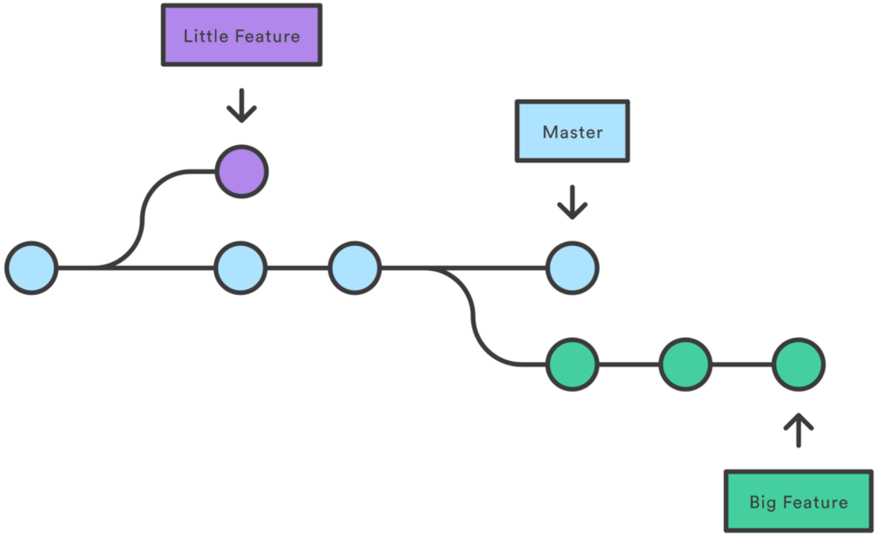
- Homework 5 team sign-up deadline Thursday
  - Team sizes, presentation slots…
- Midterm exam in class Thursday (31 October)
  - Review session Wednesday, 30 October, 6-8 p.m. in HH B131
- Next required reading due Tuesday
  - Java Concurrency in Practice, Sections 11.3 and 11.4
- Homework 5 frameworks discussion

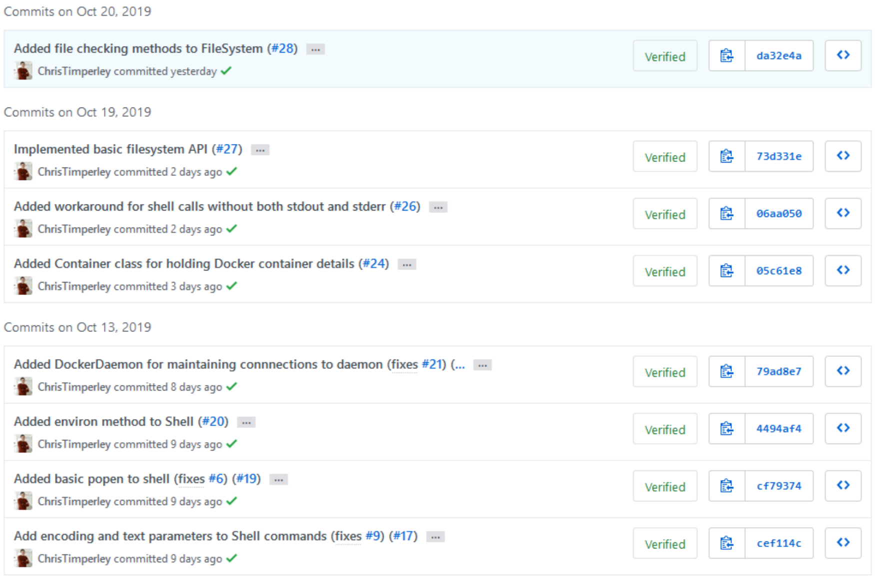
Key concepts from last Thursday

Challenges of working as a team: Aligning expectations

- How do we make decisions?

Use simple **branch-based development**





Commits on Oct 20, 2019

Added file checking methods to FileSystem (#28) …
ChrisTimperley committed yesterday ✓
Verified da32e4a

Commits on Oct 19, 2019

Implemented basic filesystem API (#27) …
ChrisTimperley committed 2 days ago ✓
Verified 73d331e

Added workaround for shell calls without both stdout and stderr (#26) …
ChrisTimperley committed 2 days ago ✓
Verified 06aa050

Added Container class for holding Docker container details (#24) …
ChrisTimperley committed 3 days ago ✓
Verified 05c61e8

Commits on Oct 13, 2019

Added DockerDaemon for maintaining connnections to daemon (fixes #21) (… …
ChrisTimperley committed 8 days ago ✓
Verified 79ad8e7

Added environ method to Shell (#20) …
ChrisTimperley committed 9 days ago ✓
Verified 4494af4

Added basic popen to shell (fixes #6) (#19) …
ChrisTimperley committed 9 days ago ✓
Verified cf79374

Add encoding and text parameters to Shell commands (fixes #9) (#17) …
ChrisTimperley committed 9 days ago ✓
Verified cef114c

Create a new branch for each feature.

- allows parallel development
- no dealing with half-finished code
- no merge conflicts!

Every commit to “master” should pass your CI checks.

build passing

Semester overview

- Introduction to Java and O-O
- Introduction to **design**
  - **Design** goals, principles, patterns
- **Design**ing classes
  - **Design** for change
  - **Design** for reuse
- **Design**ing (sub)systems
  - **Design** for robustness
  - **Design** for change (cont.)
- **Design** case studies
- **Design** for large-scale reuse
- Explicit concurrency

- Crosscutting topics:
  - Modern development tools: IDEs, version control, build automation, continuous integration, static analysis
  - Modeling and specification, formal and informal
  - Functional correctness: Testing, static analysis, verification

Today: Concurrency, motivation and primitives

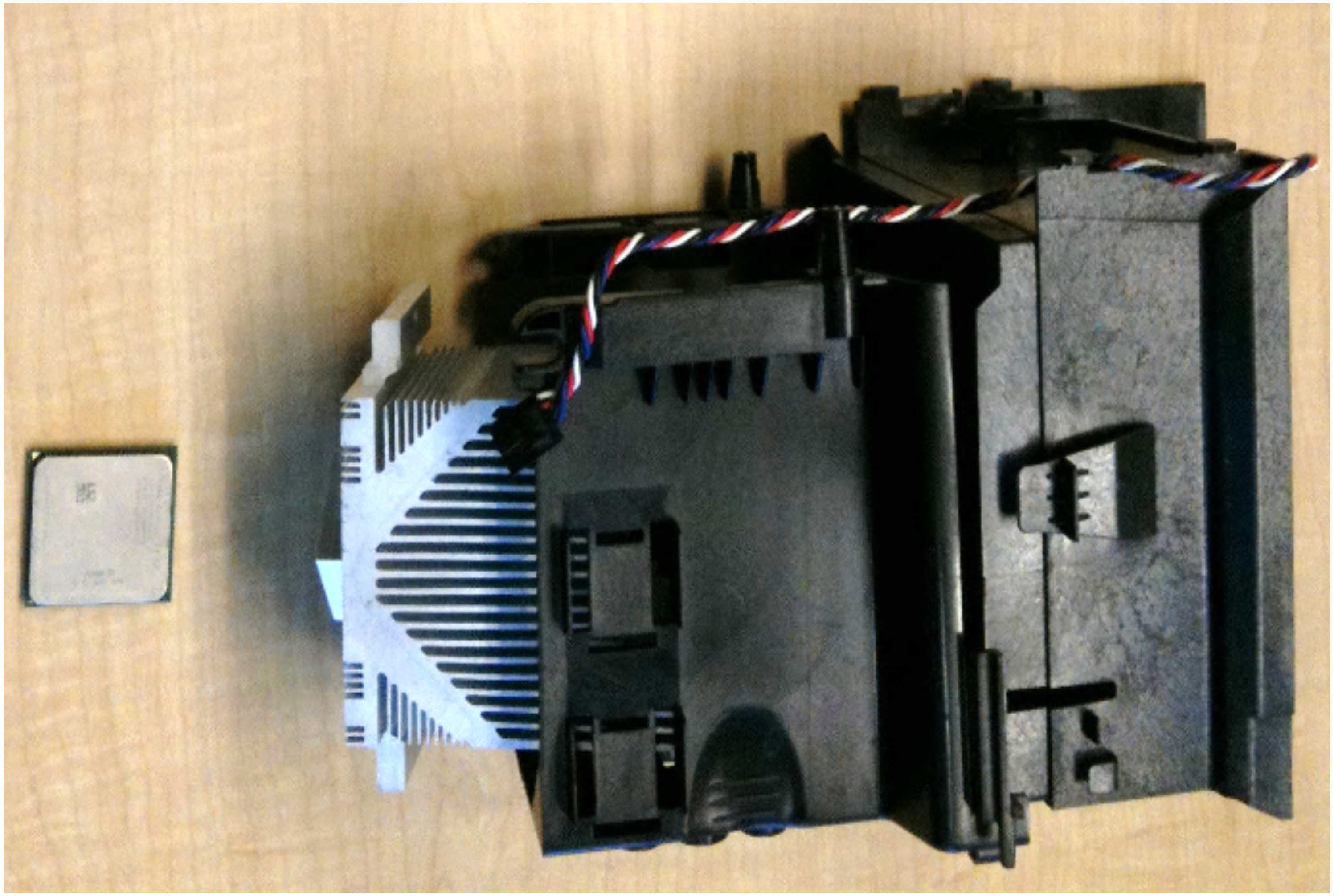
- The backstory
  - Motivation, goals, problems, …
- Concurrency primitives in Java
- Coming soon (not today):
  - Higher-level abstractions for concurrency
  - Program structure for concurrency
  - Frameworks for concurrent computation

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

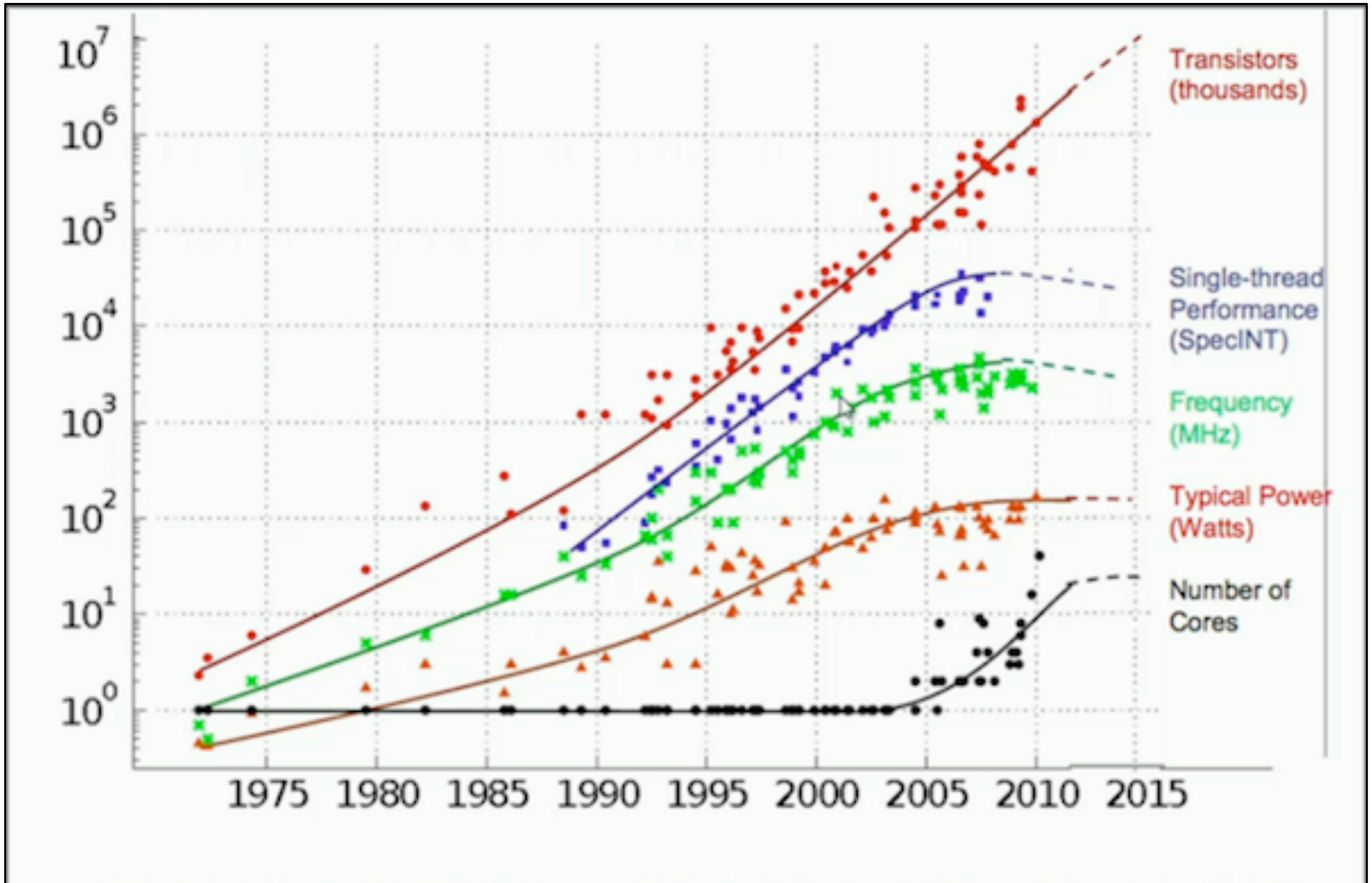


One option: fix the symptom

- 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

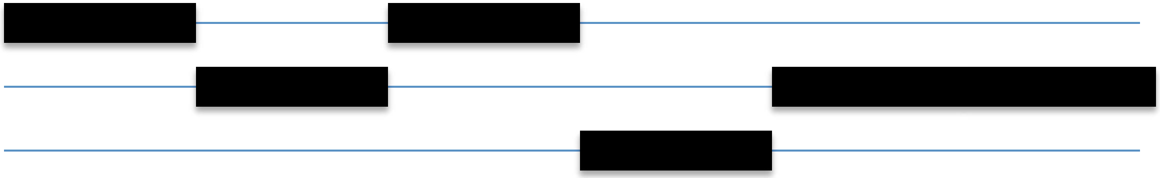
- In the past, multi-threading just a convenient abstraction
  - GUI design: event dispatch thread
  - Server design: isolate each client's work
  - Workflow design: isolate producers and consumers
- Now: required for scalability and performance

We are all concurrent programmers

- Java is inherently multithreaded
- To utilize modern 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

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

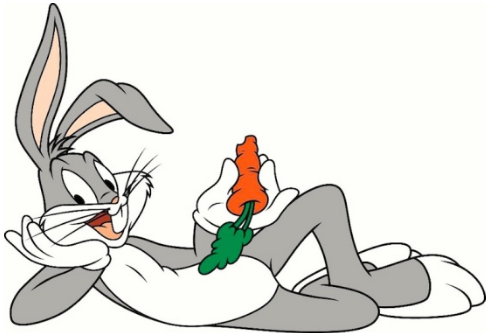
```
public class Thread {
    public Thread(Runnable task);
    public void start();
    public void join();
    …
}
```

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;
    }
}
```

Example: Money-grab (2)

```
public static void main(String[] args) throws InterruptedException
   {
    BankAccount bugs = new BankAccount(100);
    BankAccount daffy = new BankAccount(100);

    Thread bugsThread = new Thread(()-> {
        for (int i = 0; i < 1_000_000; i++)
            transferFrom(daffy, bugs, 100);
    });

    Thread daffyThread = new Thread(()-> {
        for (int i = 0; i < 1_000_000; i++)
            transferFrom(bugs, daffy, 100);
    });

    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

The challenge of concurrency control

- Not enough concurrency control: *safety failure*
  - Incorrect computation
- Too much concurrency control: *liveness failure*
  - Possibly no computation at all (*deadlock* or *livelock*)

Shared mutable state requires concurrency control

- Three basic choices:
  1. Don't mutate: share only immutable state
  2. Don't share: isolate mutable state in individual threads
  3. If you must share mutable state: *limit concurrency to achieve safety*

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

Concurrency control with Java's *intrinsic* locks

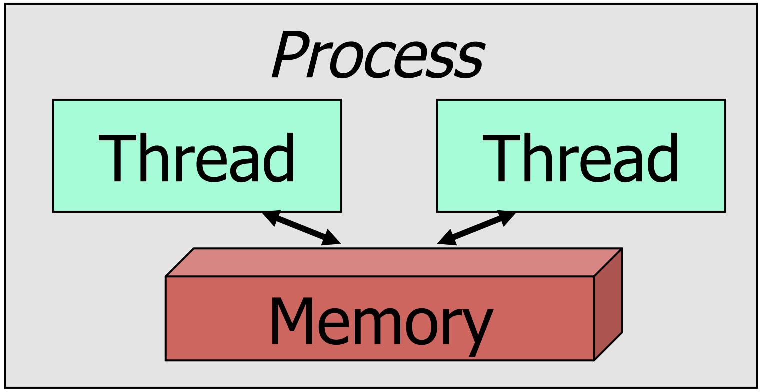
- `synchronized (lock)` { … }
  - Synchronizes entire block on object `lock`; cannot forget to unlock
  - Intrinsic locks are *exclusive*: One thread at a time holds the lock
  - Intrinsic locks are *reentrant*: A thread can repeatedly get same lock
- `synchronized` on an instance method
  - Equivalent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

Another example: serial number generation

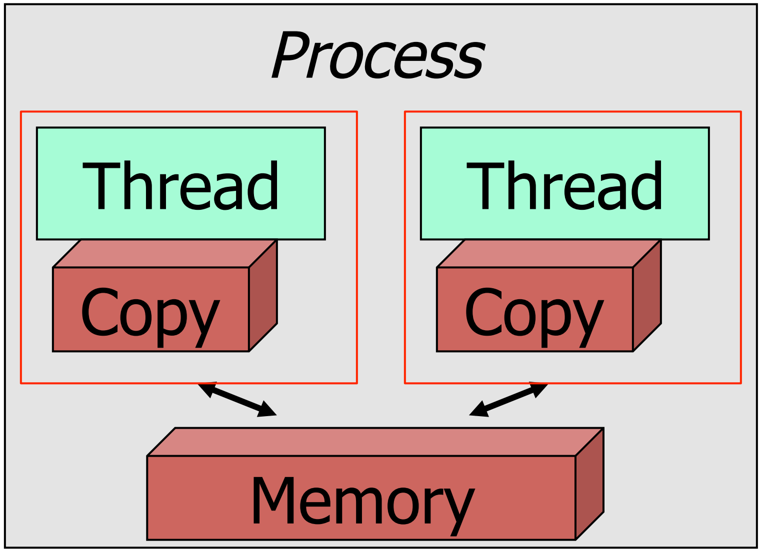
```
public class SerialNumber {
    private static long nextSerialNumber = 0;
    public static long generateSerialNumber() {
        return nextSerialNumber++;
    }

    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;
    }
}
```

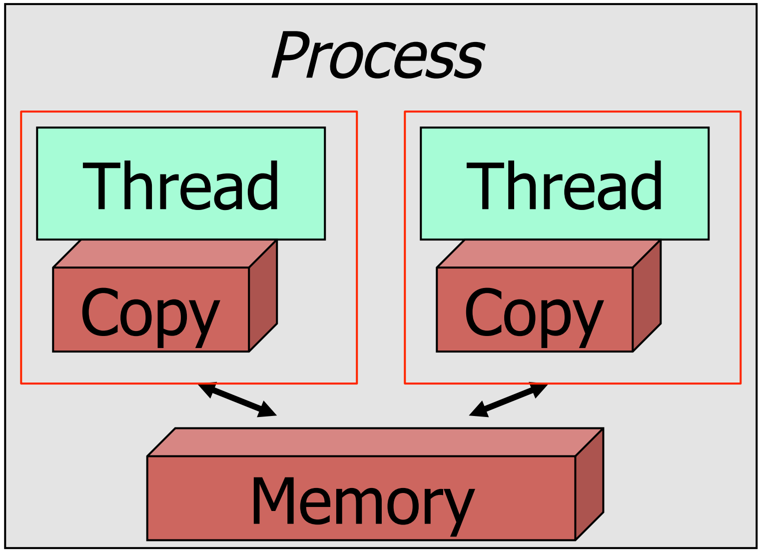
What went wrong?

- 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;
    }
}
```

Summary

- Like it or not, you're a concurrent programmer
- Ideally, avoid shared mutable state
  - If you can't avoid it, synchronize properly
- Even atomic operations require synchronization
  - e.g., `stopRequested = true`
- Some things that look atomic aren't (e.g., `val++`)