Advances in Programming Languages

Lecture 8: Concurrency

Ian Stark

School of Informatics
The University of Edinburgh

Thursday 11 October 2018
Semester 1 Week 4

THE UNIVERSITY *of* EDINBURGH

https://wp.inf.ed.ac.uk/apl18
https://course.inf.ed.ac.uk/apl

Outline

1. Opening
2. Concurrency
3. Basics of concurrency in Java
4. Closing

Outline

1. Opening
2. Concurrency
3. Basics of concurrency in Java
4. Closing

Homework from Monday

1. Read This

The instructions for the APL written coursework assignment. https://is.gd/apl18coursework

2. Write This

The outline draft for your assignment, following in detail the instructions above.

What's in the course?

The lectures cover four sample areas of "advances in programming languages":

- Types: Parameterized, Polymorphic, Dependent, Refined
- Programming for Concurrency
- Augmented Languages for Correctness and Certification
- Programming for Memory and Thread Safety with Rust

Programming-Language Techniques for Concurrency

This is the first of a block of lectures looking at programming-language techniques for concurrent programs and concurrent architectures.

- Introduction, basic Java concurrency
- Concurrency abstractions
- Concurrency in different languages
- Current challenges in concurrency

Outline

1 Opening

2 Concurrency

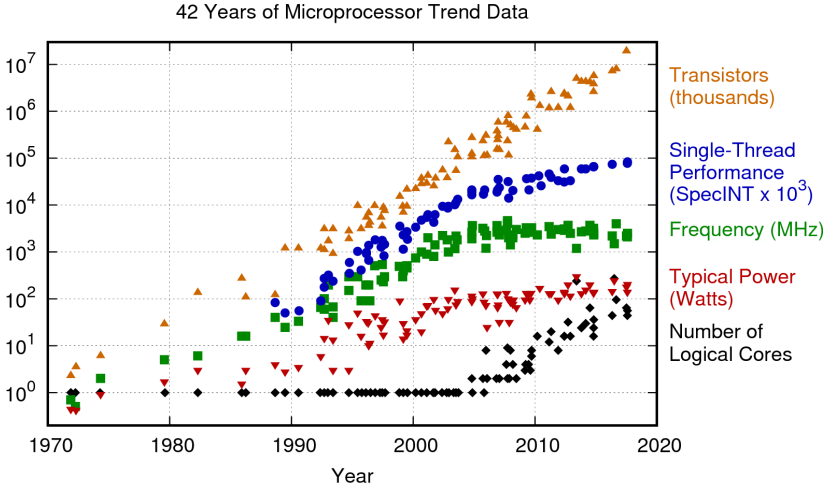
3 Basics of concurrency in Java

4 Closing

The free lunch is over

For the past 30 years, computer performance has been driven by Moore's Law; from now on, it will be driven by Amdahl's Law.

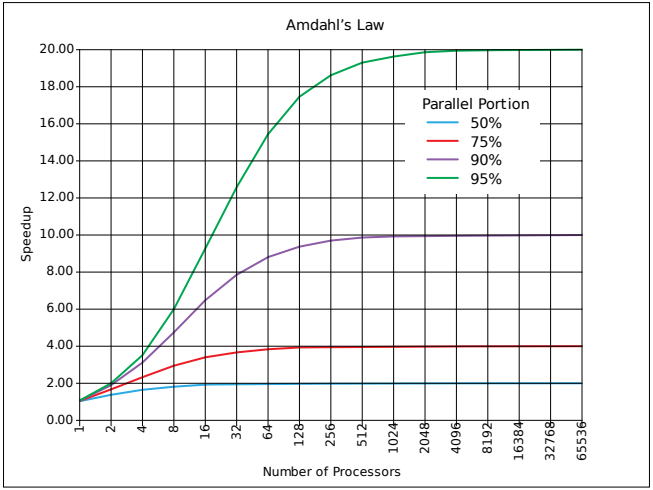
Doron Rajwan, Intel
Power Management Architect



The free lunch is over

For the past 30 years, computer performance has been driven by Moore's Law; from now on, it will be driven by Amdahl's Law.

Doron Rajwan, Intel
Power Management Architect

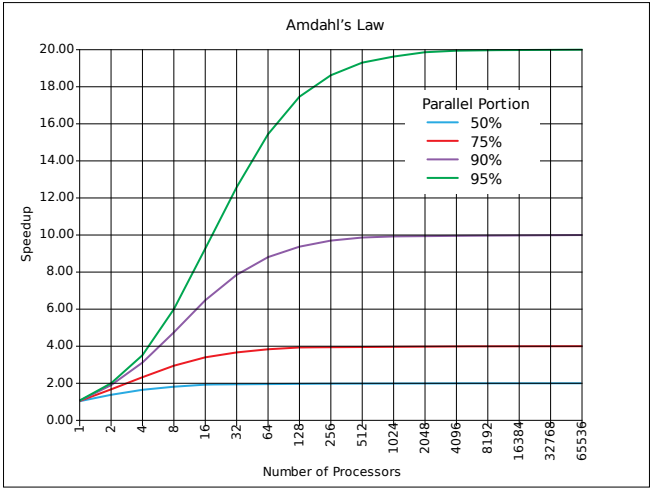


Daniels220, Wikimedia Commons

The free lunch is over

Concurrency is the next major revolution in how we write software. . . . The vast majority of programmers today don't grok concurrency, just as the vast majority of programmers 15 years ago didn't yet grok objects.

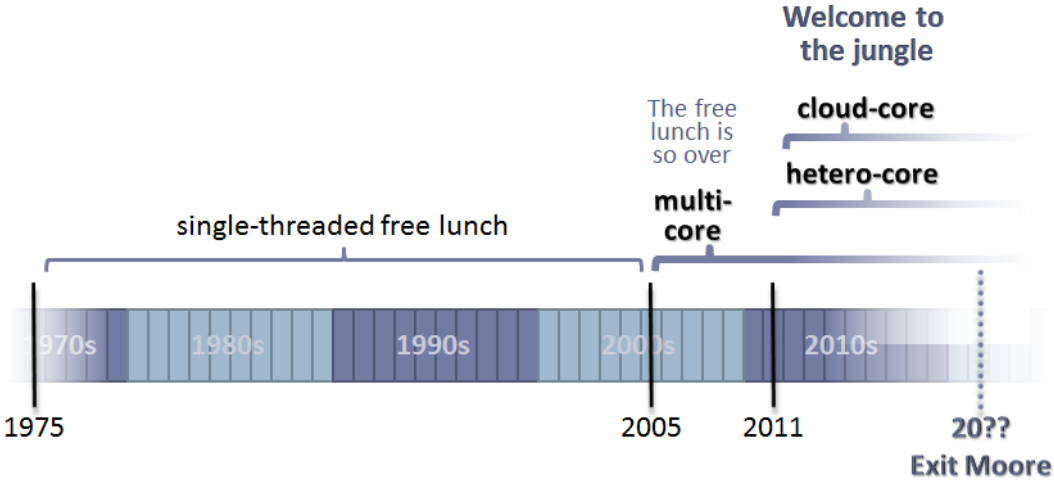
Herb Sutter
Microsoft, C++ ISO chair



Daniels220, Wikimedia Commons

The free lunch is so over

https://herbsutter.com/welcome-to-the-jungle/



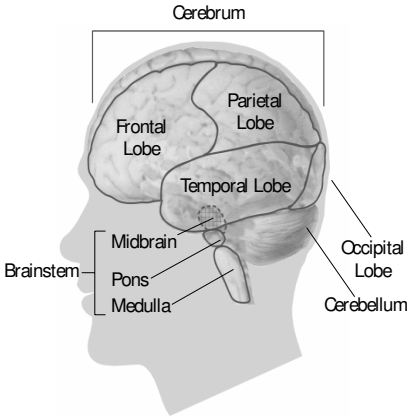
Why write concurrent programs?

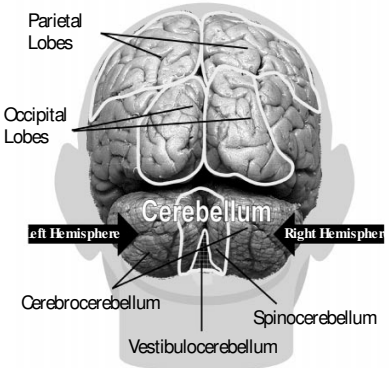
Concurrent programming is about writing code that can handle doing more than one thing at a time. There are various motivations, such as:

- Separation of duties (fetch data, render images)
- Efficient use of mixed resources (disk, memory, network)
- Responsiveness (GUI, sensors and devices, embedded systems)
- Speed (multiprocessing, hyperthreading, multi-core, GPGPU)
- Scale (distribution, cloud, elasticity)
- Serving multiple clients (database engine, web server)

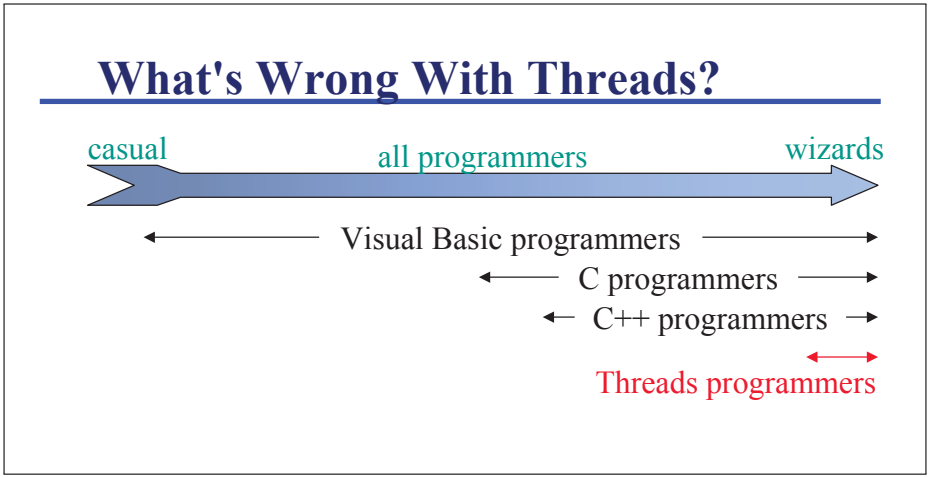
Note that the aims here are different to *parallel programming*, which is generally about the efficient (and speedy) processing of large sets of data.

Other computational modalities have this too





How hard can it be?



John Ousterhout: *Why Threads Are A Bad Idea (for most purposes)*
USENIX Technical Conference, invited talk, 1996

It's hard to walk and chew gum

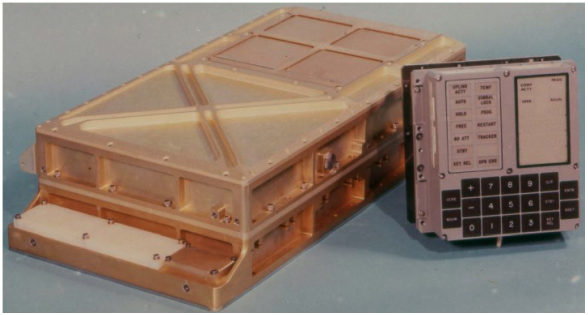
Concurrent programming offers much, including entirely new problems.

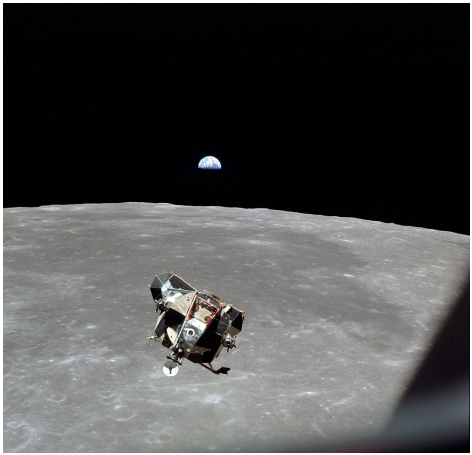
Interference — code that is fine on its own may fail if run concurrently.
Liveness — making sure that a program does anything at all.
Starvation — making sure that all parts of the program make progress.
Fairness — making sure that everyone makes reasonable progress.
Scalability — making sure that more workers means more progress.
Causality — making sure that things happen in a particular order.
Safety — making sure that the program always does the right thing.
Specification — just working out what is "the right thing" can be tricky.

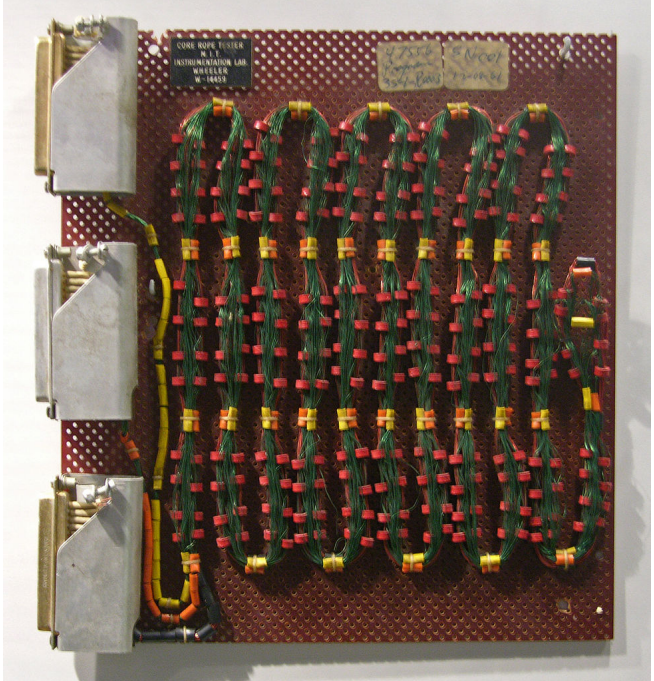
Concurrent programming is hard, and although there is considerable research, and even progress, on how to do it well, it is often wise to avoid doing it (explicitly) yourself unless absolutely necessary.

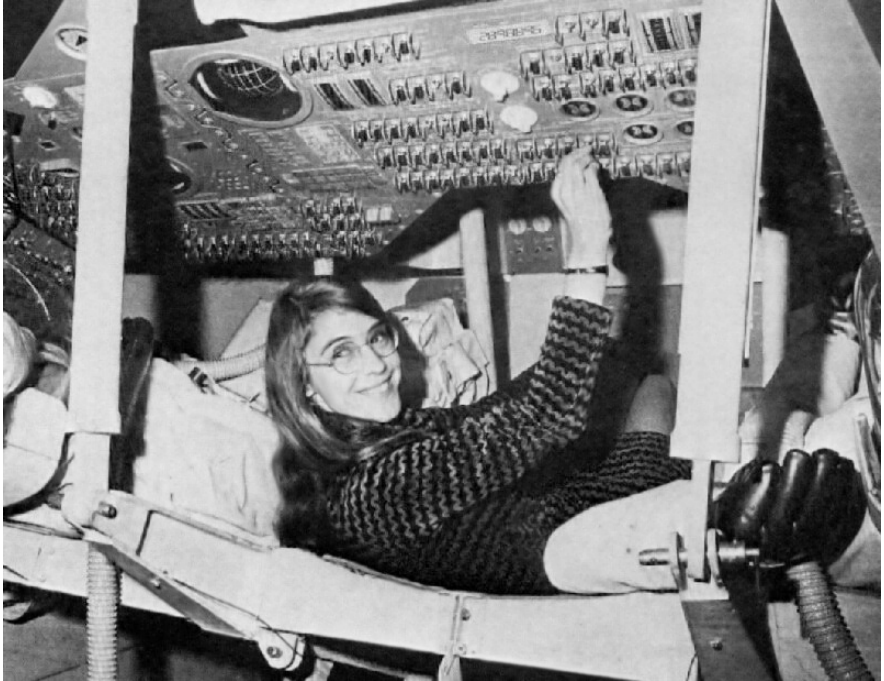
The greatest performance improvement of all
is when a system goes from not-working to working — *Ousterhout*

Concurrent programming in practice






CORE ROPE TESTER
M.I.T.
INSTRUMENTATION LAB
WHEELER
W-18459



Clients

- Abadon
- ABB
- Access Research
- Adtranz Signal
- Advanced Methods
- AGS Nynex
- Apogee Communications Technologies
- Ariel Technologies
- AT&T
- Auburn University
- BDM International Inc.
- BMC3 Engineering
- Boeing
- Boston University
- Bowe Software Solutions
- Lockheed Martin
- Logicon
- Loral
- Los Alamos National Laboratories
- Massachusetts Institure of Technology (MIT)
- McDonnell Douglas
- Mitre
- Motorola
- MRJ, Inc.
- NASA
- Newbridge Networks
- NR Pty, Ltd.
- NRI & NCC Co. Ltd.
- NSA
- Pacific States Marine Fisheries Commission

PRESIDENT OF THE

Women of NASA
MARGARET HAMILTON
KATHERINE JOHNSON
SALLY RIDE
NANCY GRACE ROMAN
MAE JEMISON
By @20tauri
Celebrate the history of women in science, technology, and math!

CDDA TVLA
ATA LLA
LFA MTA
CTA CTM
TWA FIA
MEN WALK ON MOON

It's hard to walk and chew gum

Concurrent programming offers much, including entirely new problems.

Interference — code that is fine on its own may fail if run concurrently.
Liveness — making sure that a program does anything at all.
Starvation — making sure that all parts of the program make progress.
Fairness — making sure that everyone makes reasonable progress.
Scalability — making sure that more workers means more progress.
Causality — making sure that things happen in a particular order.
Safety — making sure that the program always does the right thing.
Specification — just working out what is "the right thing" can be tricky.

Concurrent programming is hard, and although there is considerable research, and even progress, on how to do it well, it is often wise to avoid doing it (explicitly) yourself unless absolutely necessary.

The greatest performance improvement of all
is when a system goes from not-working to working — *Ousterhout*

Processes, Threads, Tasks, Events

All operating systems and many programming languages provide for concurrent programming, usually through a notion of *process*, *thread*, *task*, or *event* handler.

The idea is that a process/thread/task/handler captures a single *flow of control*. A concurrent program will have many of these at a time.

A *scheduler* directs which threads are running at any time, and how control passes among them.

There are many design tradeoffs here, concerning memory separation, mutual protection, communication, scheduling, signalling, ...

Usually processes are "heavyweight" (e.g., separate memory space), threads "lightweight" (shared memory) and tasks lightest. But usage is not precise or consistent. Any complete system will include multiple layers of concurrency, possibly of different kinds.

The concurrency presented to a programmer may not match that available on any particular hardware; and there will be a cost in maintaining the illusion of a persistent flow of control. I think it's possible the model of multiple concurrent threads may itself become outdated.

Critical Section

A key issue with multiple explicit threads is to avoid interference through shared memory.

```
void moveBy(int dx, int dy) {
    System.out.println("Moving by "+dx+","+dy);
    x = x+dx;
    y = y+dy;
    System.out.println("Completed move");
}

void moveTo(int newx, int newy) {
    System.out.println("Moving to "+newx+","+newy);
    x = newx;
    y = newy;
    System.out.println("Completed move");
}
```

Critical Section

A key issue with multiple explicit threads is to avoid interference through shared memory.

```
void moveBy(int dx, int dy) {
    .
    .
  x = x+dx;
  y = y+dy;
    .
    .
}
```

```
void moveTo(int newx, int newy) {
    .
    .
  x = newx;
  y = newy;
    .
    .
}
```

Because both methods access the fields x and y, it is vital that these two *critical sections* of code are not executing at the same time.

Critical Section

A key issue with multiple explicit threads is to avoid interference through shared memory.

```
void moveBy(int dx, int dy) {
    .
    .
  x = x+dx;
  y = y+dy;
    .
    .
}
```

```
void moveTo(int newx, int newy) {
    .
    .
  x = newx;
  y = newy;
    .
    .
}
```

Interference can result in erroneous states — perhaps violating object invariants and causing crashes, or possibly going undetected until later.

Critical Section

A key issue with multiple explicit threads is to avoid interference through shared memory.

```
void moveBy(int dx, int dy) {
    .
    .
  x = x+dx;
  y = y+dy;
    .
    .
}
```

```
void moveTo(int newx, int newy) {
    .
    .
  x = newx;
  y = newy;
    .
    .
}
```

Debugging can be difficult because scheduling is often non-deterministic.

The only thing worse than a problem that happens all the time is a problem that doesn't happen all the time — *Ousterhout*

Locks and more

There are many ways to ensure critical sections do not interfere, and refinements to make sure that these constraints do not disable desired concurrency.

- Locks
- Mutexes
- Semaphores
- Condition variables
- Monitors *etc.*

Constructs such as these can ensure *mutual exclusion* or enforce *serialization* orderings.

They are implemented on top of other underlying locking mechanisms, either in hardware (test-and-set, compare-and-swap,...) or software (spinlock, various busy-wait algorithms).

Outline

1 Opening

2 Concurrency

3 Basics of concurrency in Java

4 Closing

Concurrency in Java

Java supports concurrent programming as an integral part of the language: threading is always available, and a range of libraries provide high-level abstractions over the basic primitives.

The class Thread encapsulates a thread, which can run arbitrary code. Threads have unique identifiers, names, and integer priorities.

Parent code can spawn multiple child threads, and then wait for individual children to terminate or (co-operatively) interrupt them.

```
class Watcher
  extends Thread {
    public void run() {
    // check what's going on
     ...
    }
}
```

```
class WorkerJob
  implements Runnable {
    public void run() {
    // do some work
     ...
    }
}
```

```
Watcher watcher = new Watcher();
watcher.start();
...
WorkerJob job = new WorkerJob();
Thread worker = new Thread(job).start();
...
work.join();
watcher.interrupt();
```

Synchronized methods

Java provides mutual exclusion for critical sections through the **synchronized** primitive.

```
synchronized void moveBy(int dx, int dy) {
    System.out.println("Moving by "+dx+","+dy);
    x = x+dx;
    y = y+dy;
    System.out.println("Completed move");
}

synchronized void moveTo(int newx, int newy) {
    System.out.println("Moving to "+newx+","+newy);
    x = newx;
    y = newy;
    System.out.println("Completed move");
}
```

Synchronized methods

Two move methods cannot now execute at the same time on the same object.

```
synchronized void moveBy(int dx, int dy) {
    System.out.println("Moving by "+dx+","+dy);
    x = x+dx;
    y = y+dy;
    System.out.println("Completed move");
}

synchronized void moveTo(int newx, int newy) {
    System.out.println("Moving to "+newx+","+newy);
    x = newx;
    y = newy;
    System.out.println("Completed move");
}
```

Synchronized methods

Each **synchronized** method must *acquire* a lock before starting and *release* it when finished.

```
synchronized void moveBy(int dx, int dy) {
    System.out.println("Moving by "+dx+","+dy);
    x = x+dx;
    y = y+dy;
    System.out.println("Completed move");
}

synchronized void moveTo(int newx, int newy) {
    System.out.println("Moving to "+newx+","+newy);
    x = newx;
    y = newy;
    System.out.println("Completed move");
}
```

Synchronized blocks

Every Java object has an implicit associated lock, used by its **synchronized** methods. This can also be used to arrange exclusive access to any block of code:

```
void moveBy(int dx, int dy) {
  System.out.println("Moving by "+dx+","+dy);
  synchronized(this) {
    x = x+dx;              // Only this section of the
    y = y+dy;              // code is critical
  }
  System.out.println("Completed move");
}
```

The locking object need not be **this**. Sometimes a lock may be better on a contained (or "owned") object, or may be needed on a containing (or "owning") object.

Condition variables

Java refines critical regions with basic *condition variables*.

- Synchronized code that finds things are not suitable for it to proceed may wait() on the condition variable associated with its lock. This blocks the code and releases the lock.
- Another thread can acquire the lock and do some work. Because this may change the situation for the other thread, it should notify() or notifyAll() other threads of this.
- Threads waiting on the condition variable will be made runnable again, and can check to see if they are now ready to proceed.

Having threads block saves on busy waiting, and ensures that they only wake when there is something to check.

A simple blocking method

```
class Pigeonhole {

  private Object contents = null;

  synchronized void put (Object o) {

    while (contents != null)    // Wait until the pigeonhole is empty
      try { wait(); }
      catch (InterruptedException ignore) { return; }

    contents = o;               // Fill the pigeonhole
    notifyAll();                // Tell anyone who might be interested
  }
 ...
}
```

Outline

1. Opening
2. Concurrency
3. Basics of concurrency in Java
4. Closing

Summary

- Concurrency is essential (efficiency, responsiveness, speed).
- But can be tricky (interference, deadlock, fairness, . . .).
- Concurrency can be in the language, in libraries, or in both.
- Java provides shared-memory concurrency in the language:
  - Locks and critical regions with **synchronized**;
  - Condition variables with wait, notify and notifyAll;
  - Explicit spawning of multiple threads.
- Java provides further concurrency abstractions in a library: java.util.concurrent.

Homework for Monday !

1. Do This

Find out what a *data race* is. What happens to C or C++ code with a data race?

2. Read This

Read the first three sections of the Java Concurrency tutorial.

http://java.sun.com/docs/books/tutorial/essential/concurrency

Processes and Threads • Thread Objects • Synchronization

Have another Java concurrency tutorial to recommend? Great! Post on Piazza or mail me.