

Introduction to Concurrency

CSCI 5828: Foundations of Software Engineering
Lecture 04 — 01/26/2012

Goals

- Present an overview of concurrency in software systems
 - Review the benefits and challenges associated with designing and implementing concurrent software systems
 - Review material from Chapter 1 of our concurrency textbook
 - review some of the code examples presented in that chapter

But first, Homework 1...

- Let's grade Homework 1

Why worry about concurrency? (I)

- “Concurrency is hard and I’ve only ever needed single-threaded programs. Why should I care about it?”
- Changing environment
 - multi-core computers (including handheld devices)
 - use of computing clusters to solve problems on the rise
- Performance
 - Growth rates for chip speed are flat;
 - You can’t wait 18 months for a 2x speed-up anymore
 - Instead, chips are becoming “wider”
 - More cores, wider bus (more data at a time), more memory

Why worry about concurrency? (II)

- Since chips are not getting faster (in the same way they used to)
 - a single-threaded, single-process application is not going to see any significant performance gains from new hardware
- Instead, software will only see performance gains from new hardware
 - if it is designed to do more work in parallel
 - as the number of processors available to it increases
- THIS IS NOT EASY
 - the application's computations must be amenable to parallelization
 - that is, it must be possible to break its work into tasks that can run at the same time with no need to coordinate with each other

Why worry about concurrency? (III)

- If you can design your system in this way, you pave the way to seeing linear speed ups as the number of processors increases
 - That is a system on n -cores will be n times faster than the same system running on a single core CPU
 - Recently, laptops are shipping with 8 cores, smart phones with 2-4 cores, desktops with 12-16 cores
 - Some threading frameworks will allow you to send tasks to your machine's graphics card and these cards can have hundreds to thousands of cores (although, granted, they are designed to be used by very specific types of algorithms)
- However, it is very hard to achieve linear speed-ups, but performance gains can still be quite significant

In addition...

- Concurrent programming is becoming hard to ignore
- lots of application domains in which concurrency is the norm
 - Embedded software systems, robotics, “command-and-control”, high-performance computing (use of clusters), ...
 - Web programming often requires concurrency (AJAX)
 - Web browsers are themselves examples of multi-threaded GUI applications
 - without threads the UI would block as information is downloaded

BUT...

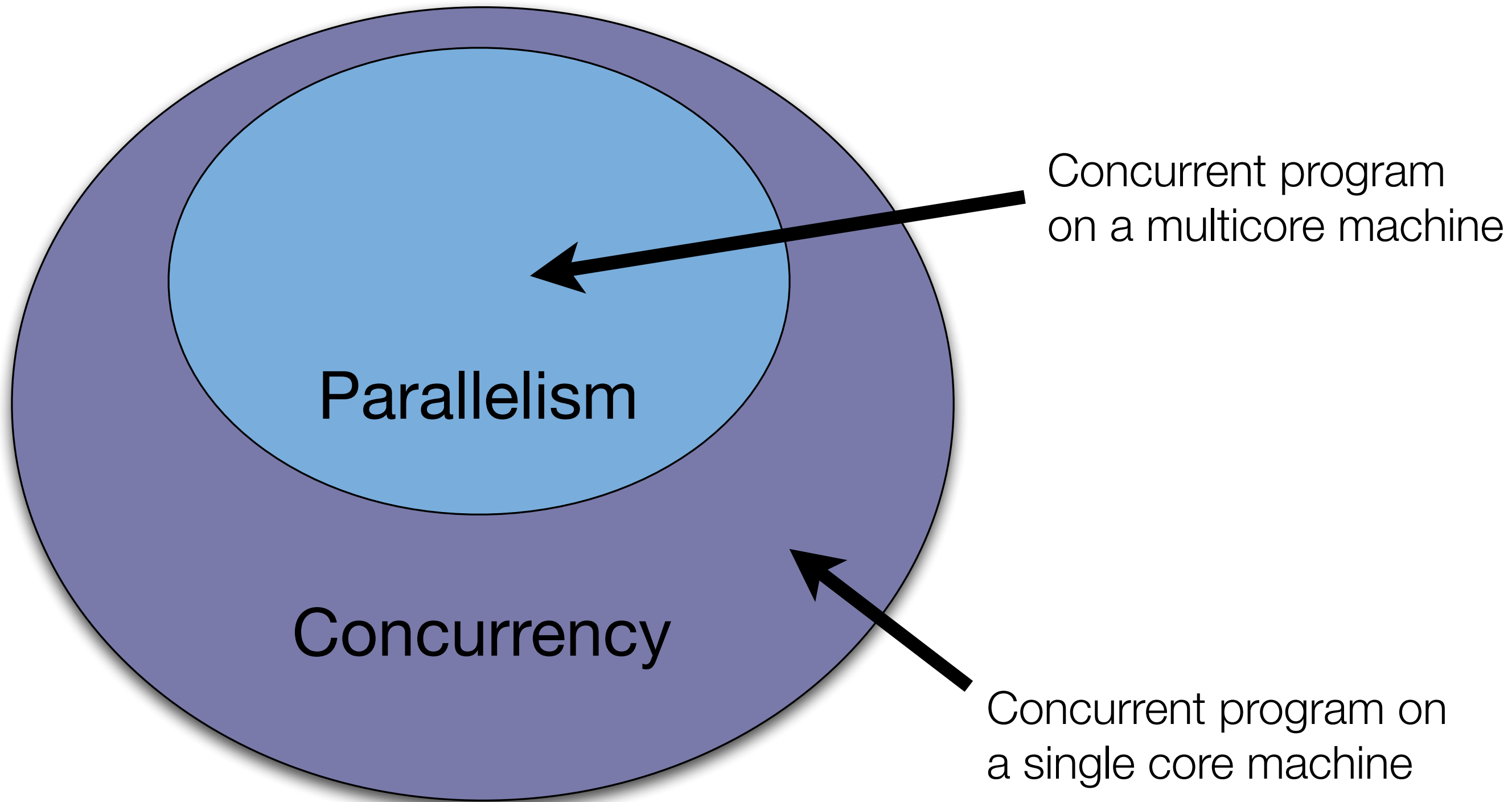
- “A DEEP CHASM OPENS BEFORE YOU...” ‡
 - Concurrency is HARD
- While concurrency is widespread, it is error prone
- Programmers trained for single-threaded programming face unfamiliar problems
 - synchronization, race conditions, deadlocks, “memory barriers”, etc.
- Let’s review some terminology

‡ — Taken from Cocoa Programming For Mac OS X, 4th Edition by Aaron Hillegass and Adam Preble

Basic Definitions

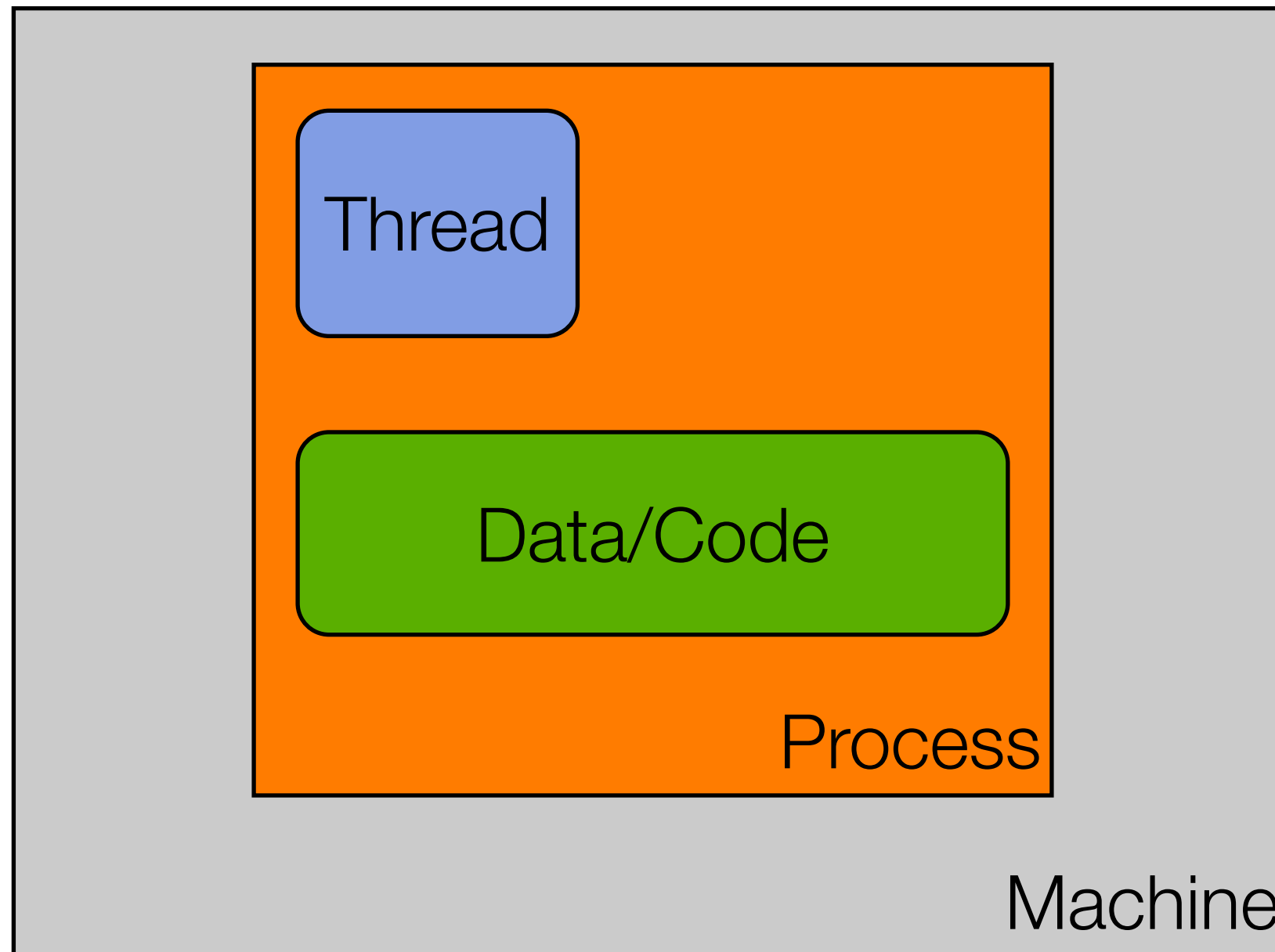
- When we execute a **program**, we create a **process**
 - A **sequential** program has a **single thread of control**
 - A **concurrent** program has **multiple threads of control**
- A single computer can have multiple processes running at once;
 - If that machine **has a single processor**, then **the illusion of multiple processes running at once** is just that: **an illusion**
 - That illusion is maintained by the operating system; it coordinates access to the single processor by the various processes; only one process runs at a time
- If a machine **has more than a single processor**, then **true parallelism** can occur
 - you can have **N processes running simultaneously on a machine that has N processors**

Thus...



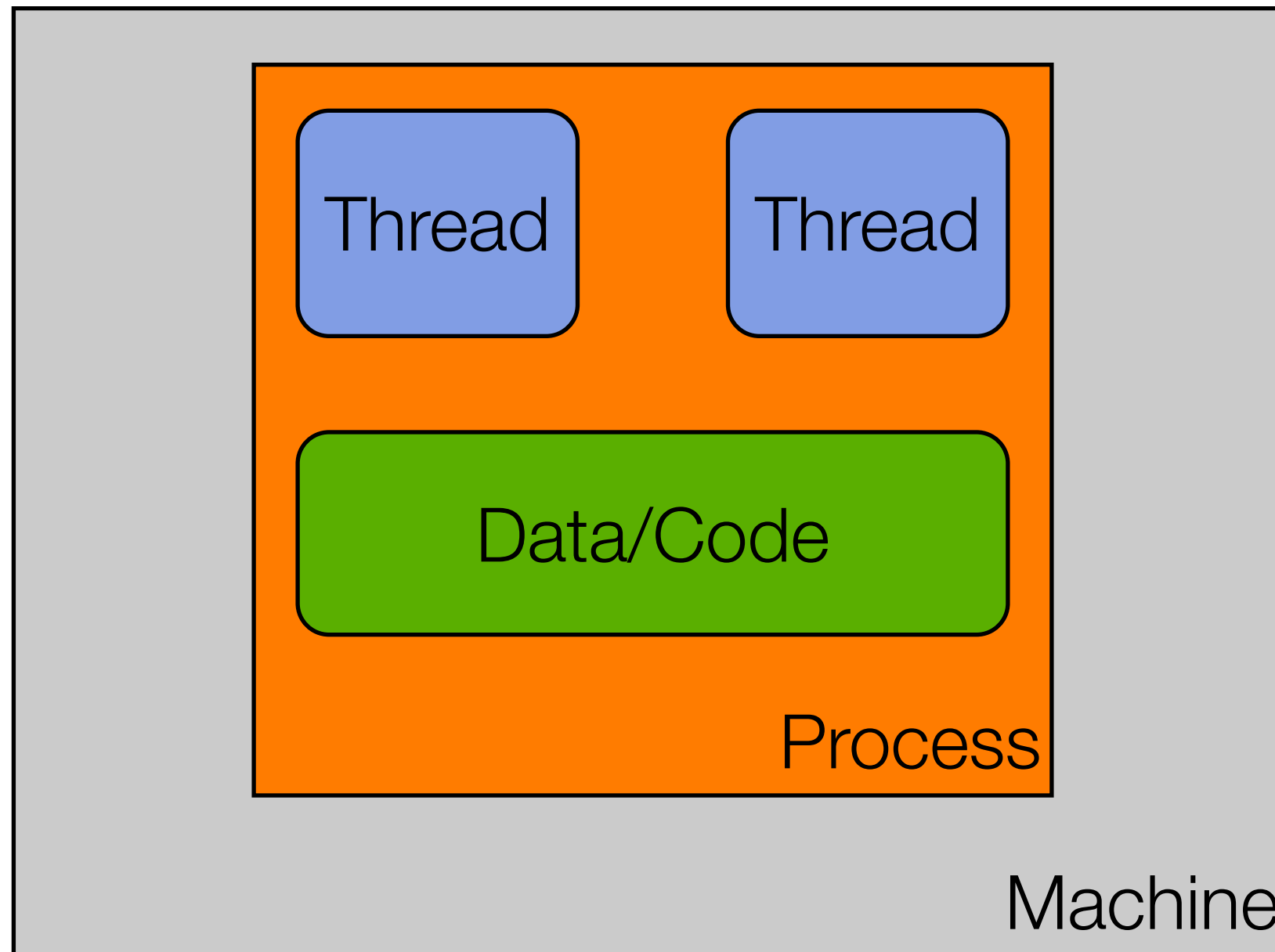
Basics: **Single** Thread, **Single** Process, **Single** Machine

Note:
Repetition
is good!



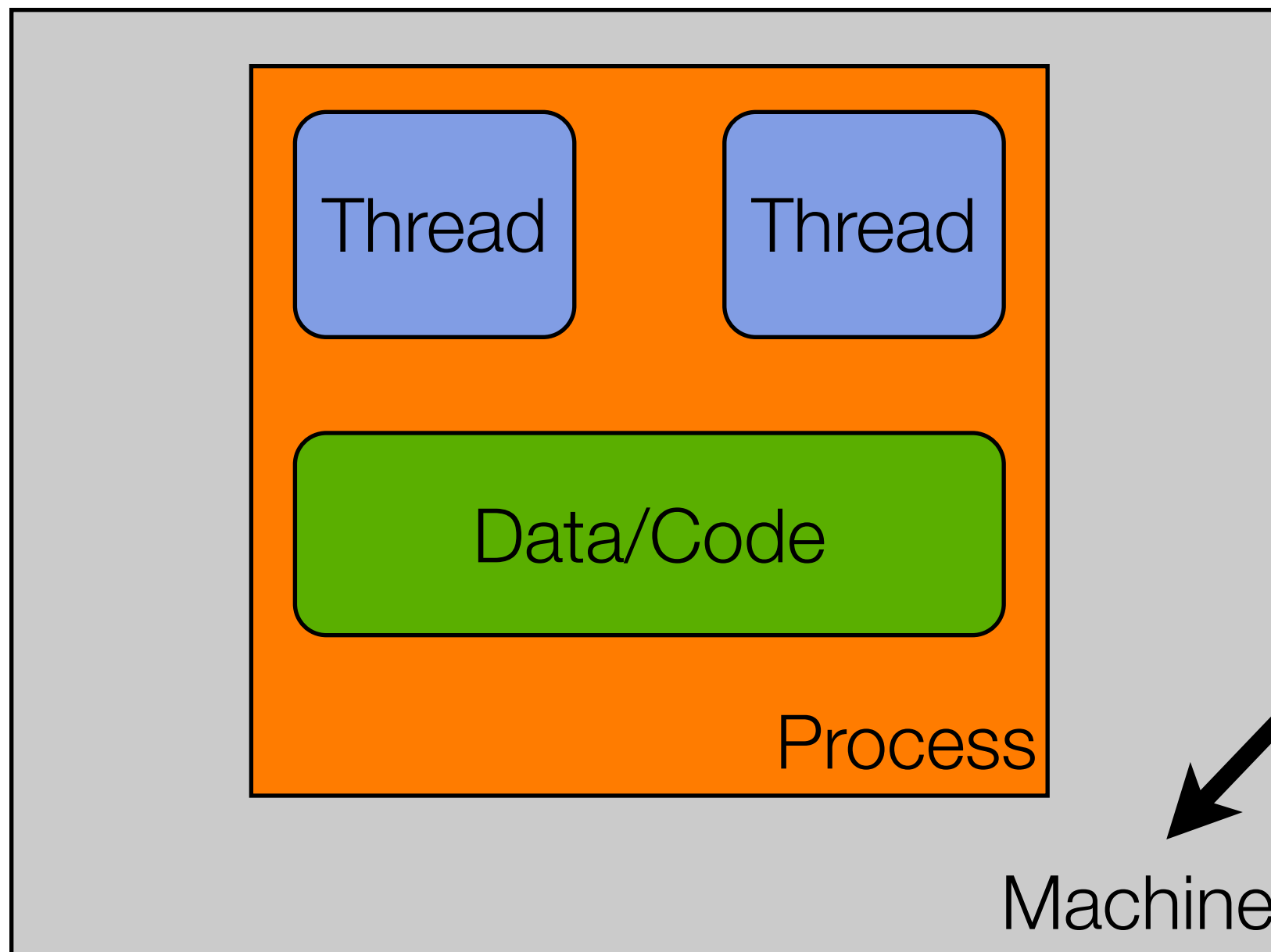
Sequential Program == Single Thread of Control

Basics: **Multiple** Thread, Single Process, Single Machine



Concurrent Program == Multiple Threads of Control

Multi-Thread: **But is it truly parallel?**



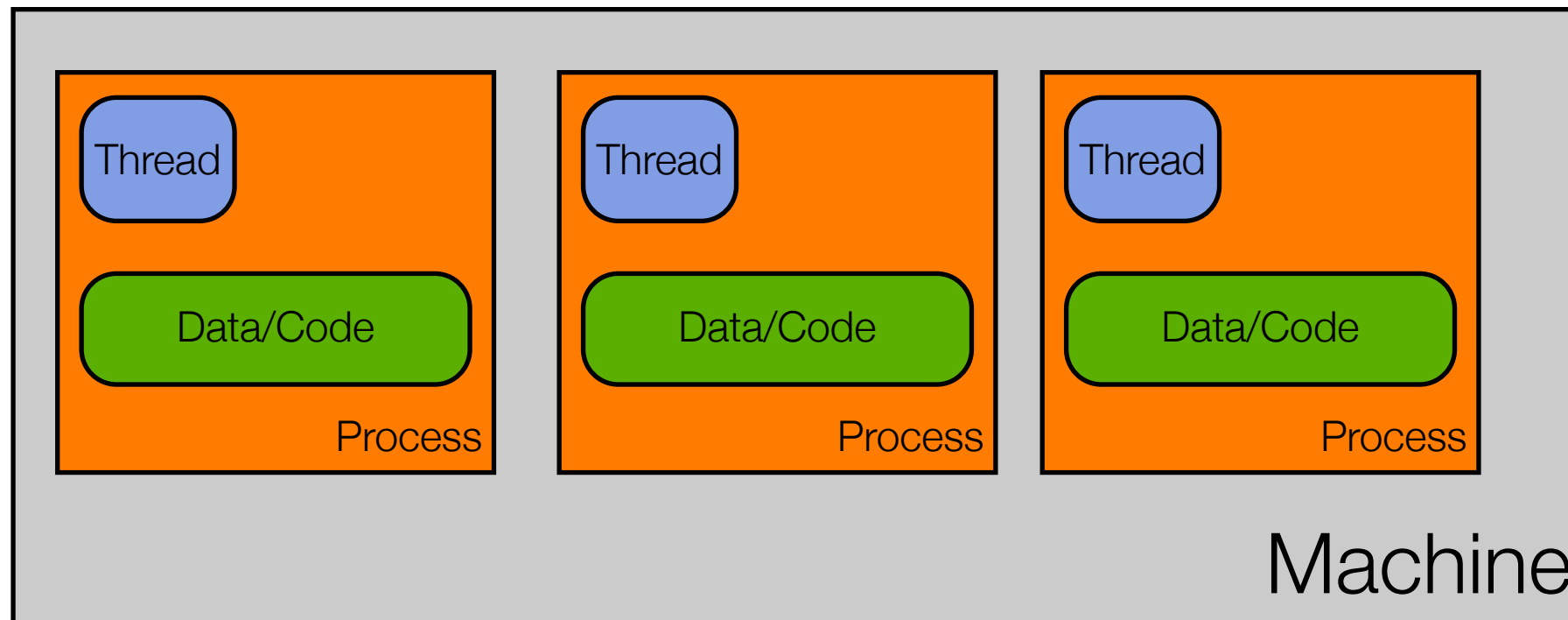
We may have multiple threads in this process, but we may not have events truly occurring in parallel. Why not?

It depends on the machine!

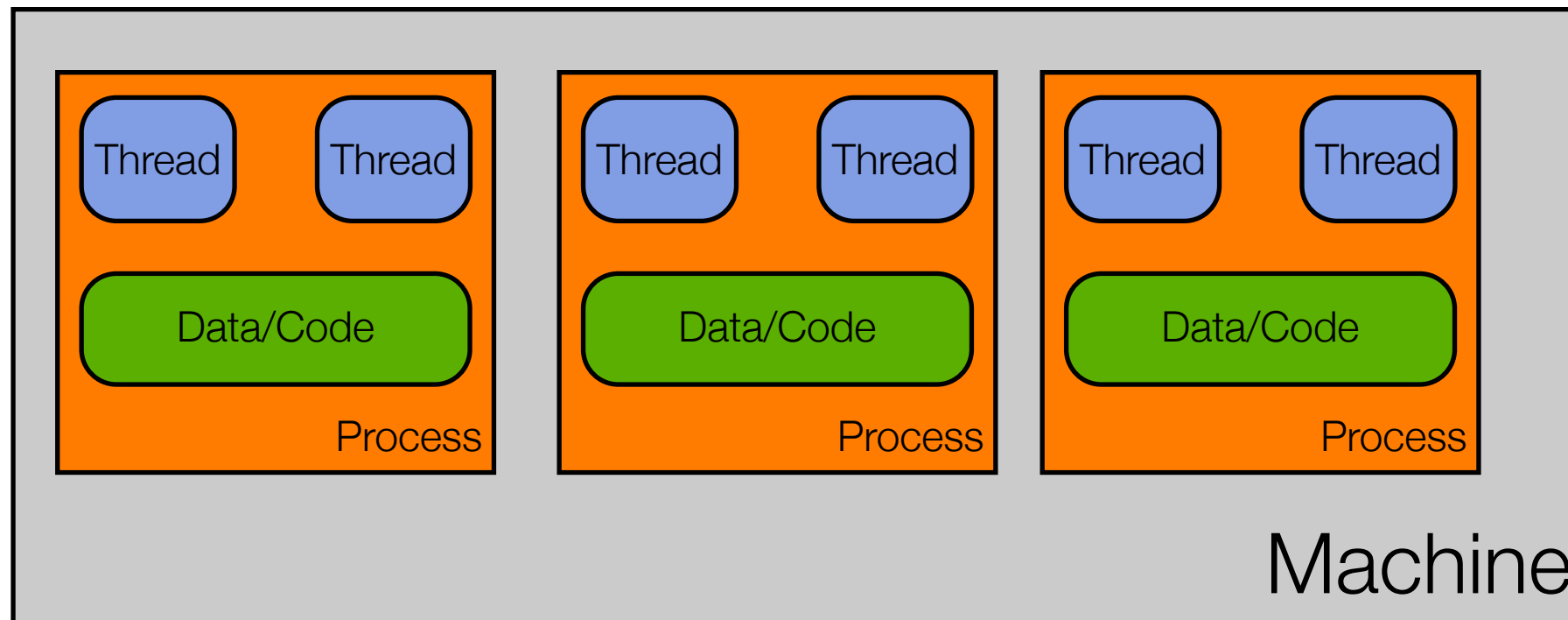
If the machine has multiple processors, then **true parallelism can occur**. Otherwise, parallelism is **simulated**

Concurrent Program == Multiple Threads of Control

Basics: Single Thread, **Multiple** Process, Single Machine

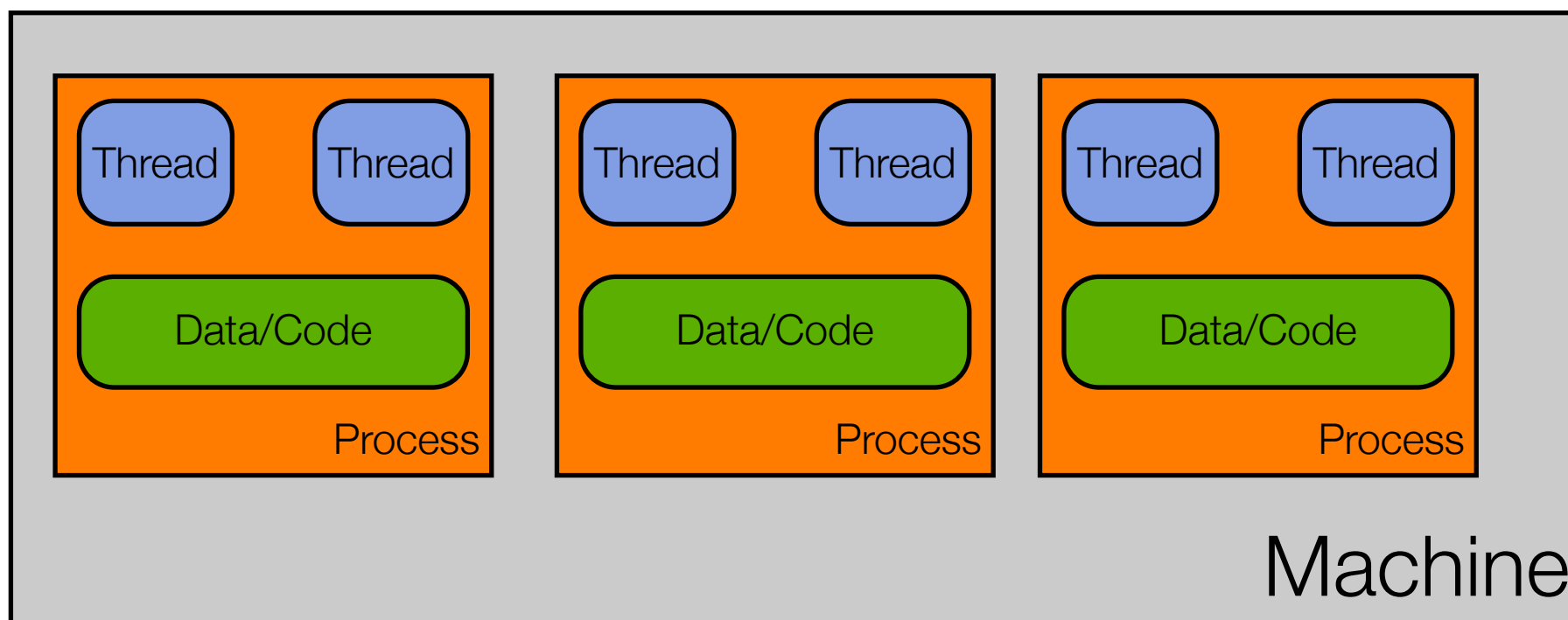
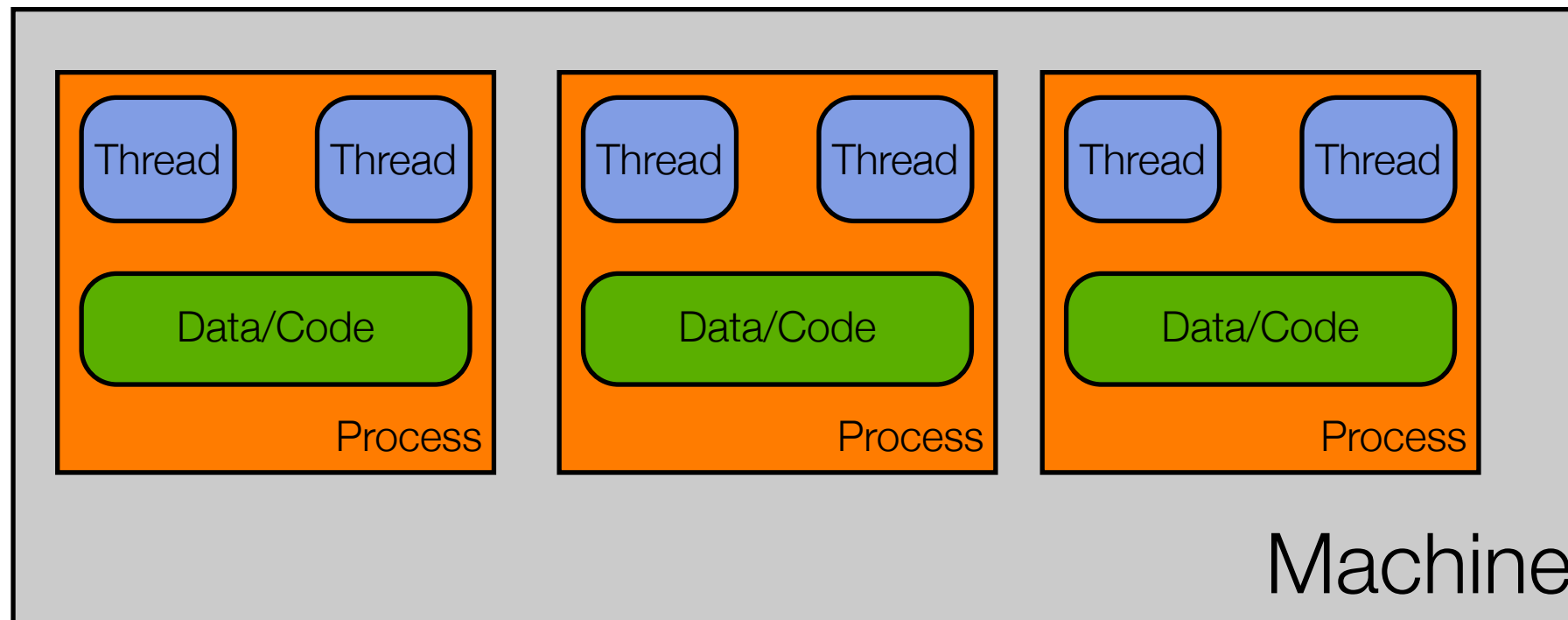


Basics: **Multi**-thread, **Multi**-Process, Single Machine



Note: You can have way more than just two threads per process.

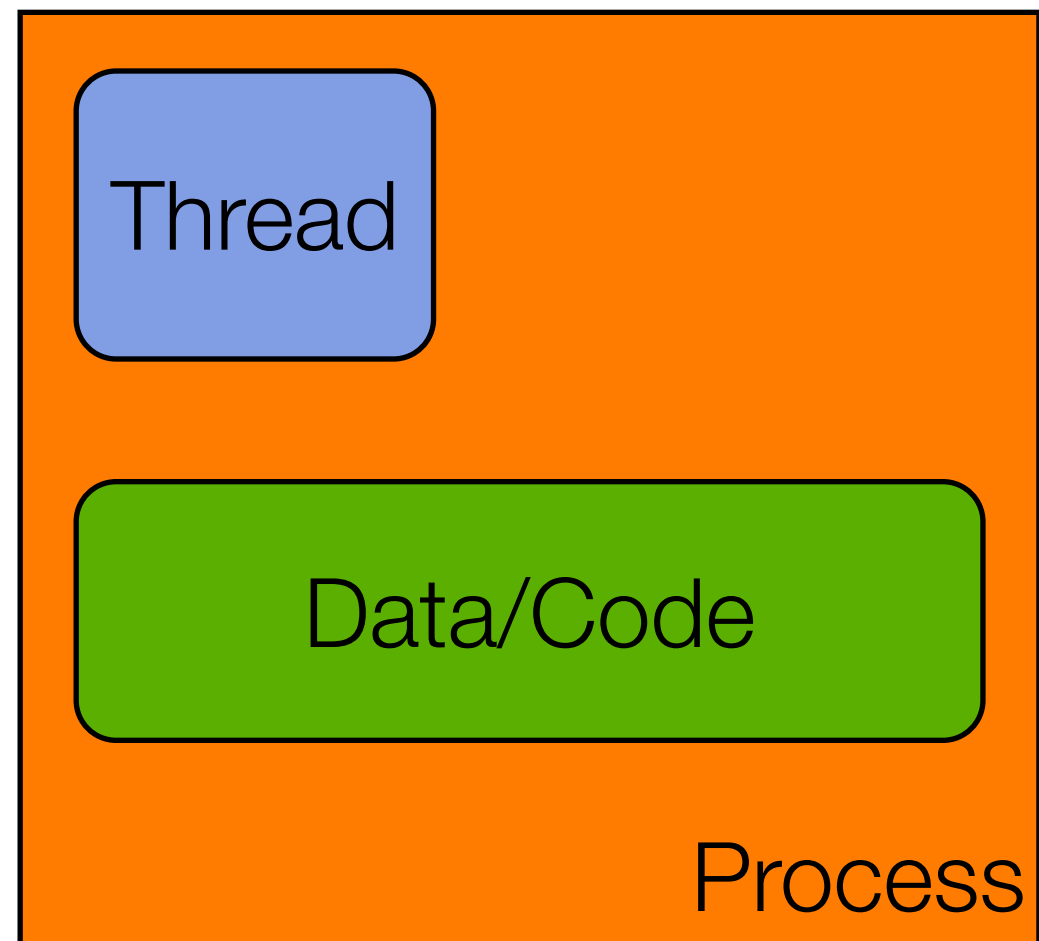
Basics: **Multi-everything**



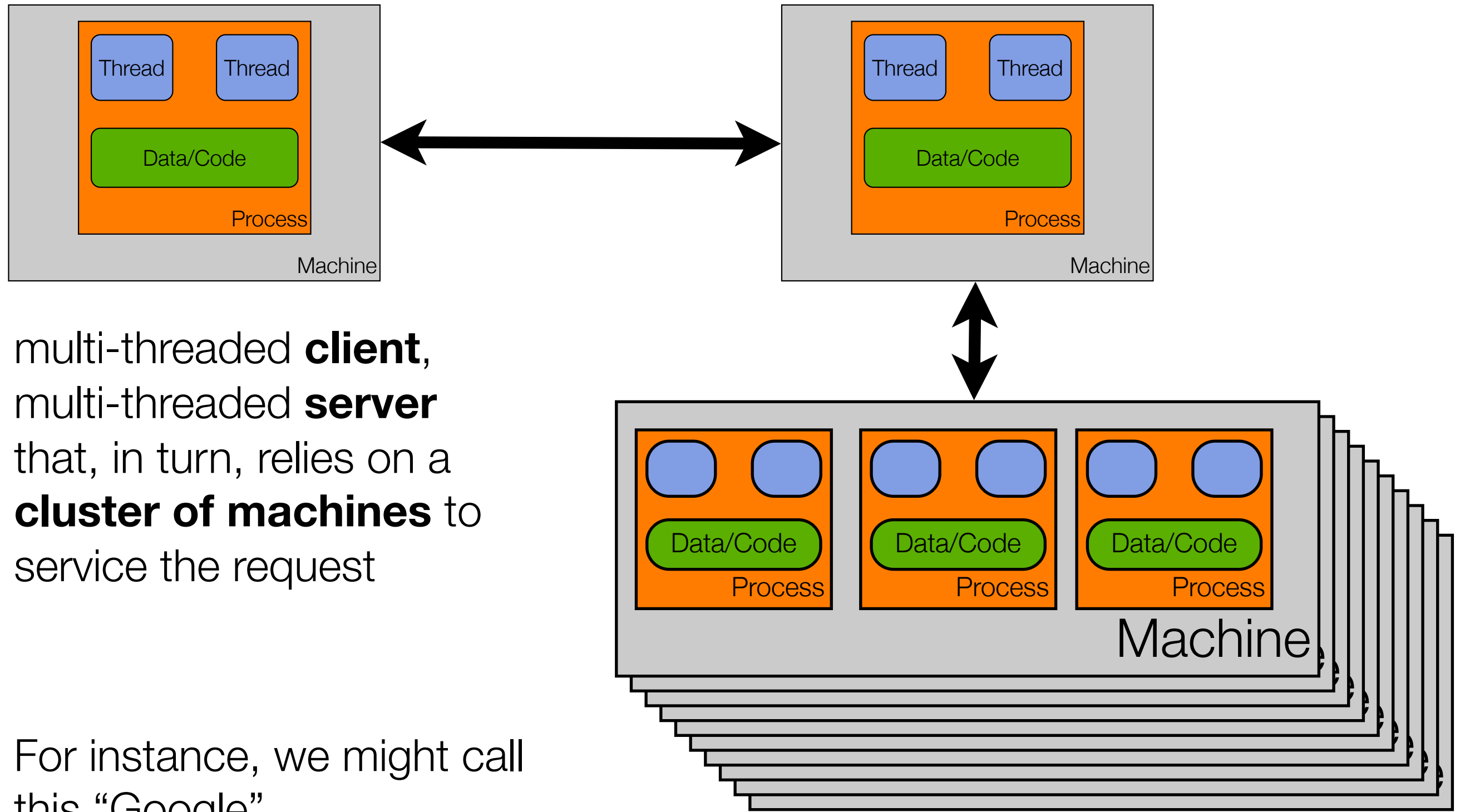
Applications are Dead! Long Live Applications!

Due to the ability to have multiple threads, multiple processes, and multiple machines work together on a single problem, the notion of an application is changing. It used to be that:

Application ==



Now... we might refer to this as “an application”



multi-threaded **client**,
multi-threaded **server**
that, in turn, relies on a
cluster of machines to
service the request

For instance, we might call
this “Google”

Software Architecture Design Choices

- When designing a modern application, we now have to ask
 - How many machines are involved?
 - What software components will be deployed on each machine?
 - For each component
 - Does it need concurrency?
 - If so, how will we achieve that concurrency?
 - multiple threads?
 - multiple processes?
 - both?

Consider Chrome

- Google made a splash in 2008 by announcing the creation of a new web browser that was
 - multi-process (one process per tab) and
 - multi-threaded (multiple threads handle loading of content within each tab)
- They documented their engineering choices via a comic book
 - <http://www.google.com/googlebooks/chrome/index.html>

Chrome Advantages

- Some of the advantages they cite for this design
 - stability
 - single-process, multi-threaded browsers are vulnerable to having a crash in one tab bring down the entire browser
 - speed
 - multi-process browsers can be more responsive due to OS support
 - security
 - exploits in single-process browsers are easier if malware loaded in one tab can grab information contained in another tab; much harder to grab information across processes

Chrome Demo

- You can use an operating system's support for process monitoring to verify that Chrome is indeed multi-process and multi-threaded
 - Demo

Other benefits to multi-process design ‡

- Lots of existing applications that do useful things
 - Think of all the powerful command line utilities found in Unix-based platforms; You can take advantage of that power in your own application
 - Create a sub-process, execute the desired tool in that process, send it input, make use of its output
- Memory leaks in other programs are not YOUR memory leaks
 - As soon as the other program is done, kill the sub-process and the OS cleans up
- Flexibility: An external process can run as a different user, can run on a different machine, can be written in a different language, ...

‡ — Also taken from Cocoa Programming For Mac OS X, 4th Edition by Aaron Hillegass and Adam Preble

Two Reasons for Using Concurrency

- Making applications more responsive
 - See example next slide
- Making applications faster
 - In particular, computationally intensive apps (compute bound) or data processing apps (I/O bound)

Example: SelfishWindow & ConsiderateWindow

- Background: In modern application frameworks, there is one thread devoted to updating the graphical user interface
 - if an event handler invokes a long running task, it will block the GUI; the long running task needs to be put in the background to keep the GUI responsive
 - DEMO
- Important to Understand
 - How many threads does Selfish Window have? Why does the GUI become blocked?
 - How many threads does Considerate Window have? Why did we have to use `SwingUtilities.invokeLaterAndWait()`?
 - Note how the complexity of our system went up as we fixed the problem and followed best practices to avoid having a non-GUI thread update the GUI

The Dangers of Concurrency (I)

- Starvation
 - All threads are active in the system but none of them are making progress
 - Thread A is waiting for an input that takes a long time to arrive
 - It's not blocked but it's not making progress
 - Generic solution:
 - **Timeouts:** have Thread A do something else once a timeout occurs
- Deadlock
 - Thread A is waiting for B to give up resource C
 - Thread B is waiting for A to give up resource D
 - Both are blocked and will never be unblocked
 - Generic solution: Have threads acquire resources in the same order

The Dangers of Concurrency (II)

- Race Conditions
 - If two threads access the same resource, we may have a race condition
 - In particular, if two threads have access to the same variable, they may attempt to change its value at the same time
 - this can result in the value getting set to an inconsistent state
 - You can also get into problems even when one thread is doing the writing and a second thread is doing the reading

The Dangers of Concurrency (III)

- First Example of Race Condition
 - threads A and B have access to an integer variable C
 - C currently equals 0 and then both A & B execute the code “ $C = C + 1$ ”
 - A reads the value 0 and gets suspended
 - then B reads the value 0 and updates it to 1
 - then A wakes up and updates the value to 1
- DEMO
 - Final value should have been 30; typically, the value was much less than that, due to the problem above; note: code had to be made way more complex than normal to make the problem appear consistently

The Dangers of Concurrency (IV)

- Second Example of Race Condition
 - Thread A is designed to loop until a boolean variable switches from true to false
 - Thread B creates A, then goes to sleep for a bit, then changes the boolean variable from true to false
 - Observed behavior: sometime the program works, sometimes it does not
 - See example in Chapter 1; DEMO
- The problem?
 - Thread A cached the value of the boolean variable and B's write never "crosses the memory barrier" to allow A to see the updated value

The memory barrier (I)

- The term “memory barrier” simply refers to transferring values from main memory into working memory and back again
 - Each thread has its own flow of execution and this means that it has
 - its own program stack
 - its own set of values for the machine’s registers
 - Each thread also shares access to the program’s heap and static data

The memory barrier (II)

- In order to make a program run faster, the compiler will look for ways to optimize memory reads/writes.
 - It may choose to cache a value stored in main memory into its set of registers
 - once that occurs, only certain types of operations will cause a change in the cached value to be synced back to its home in main memory
 - If the cached value corresponds to a value accessed by more than one thread, then problems can occur when changes made by one thread to shared memory are not made visible to another thread

The memory barrier (III)

- To avoid the race condition in our second example, we must do something to ensure that B's change to the shared boolean variable is made visible to A
 - Examples of techniques that cause the write to pass the memory barrier
 - Tagging the variable with the keyword “volatile”
 - Tagging methods that access the variable with the keyword “synchronized”
 - There are a few others that we will see later this semester
- As we have seen, having values that can be changed shared between multiple threads (a.k.a. shared mutability) can lead to incorrect behavior, unstable systems, crashes, locked apps, etc.

The Response: Avoid Shared Mutability

- To get around the problems associated with shared mutability, we must
 - AVOID SHARED MUTABILITY
- As our textbook author says “Shared mutability is pure evil. Avoid it!”
- In our upcoming lectures
 - we will look for ways to deal with shared mutability when we can’t avoid it
 - learn how to design systems to avoid shared mutability altogether

Summary

- Introduced the subject of concurrency in software systems
 - why it is important
 - why we cannot really avoid it
 - what problems occur when dealing with concurrency
 - starvation, deadlock, race conditions
 - learned about shared mutability and why to avoid it
 - learned about the “memory barrier” which is related to Java’s memory model and contributes to some of the problems of shared mutability

Coming Up Next

- Lecture 5: Introduction to Software Testing
- Lecture 6: Agile Methods and Agile Teams