Introduction to Concurrency

Kenneth M. Anderson
University of Colorado, Boulder

CSCI 5828 — Lecture 4 — 01/21/2010

Credit where Credit is Due

- Some text and images for this lecture come from the lecture materials provided by the publisher of the Magee/Kramer optional textbook. As such, some material is copyright © 2006 John Wiley & Sons, Ltd.

Lecture Goals

- Review material in Chapter 1 of the Breshears textbook
  - Who wants to go faster? Raise your hand if you want to go faster! :-)
  - Threading Methodologies
  - Parallel Algorithms (Intro)
- Cover remainder of Chapter 1 material in Lecture 6

Why worry?

- “Concurrency is hard and I’ve only ever needed single-threaded programs: Why should I care about it?”
- Answer: multi-core computers, increasing use of clusters
  - Growth rates for chip speed are flattening
    - “lets wait a year and our system will run faster!”: No longer!
  - Instead, chips are becoming “wider”
    - more cores, wider bus (more data at a time), more memory
- As chips are not getting faster (the same way they used to), a single-threaded, single process application is not going to see any significant performance gains from new hardware

New Model

- Instead, **software** will only see performance gains with new hardware if **they are designed to get faster the more processors they have available**
  - **This is not easy**: the computations that an application performs have to be amenable to parallelization
- Such an application will see noticeable speed improvements when run on machines with more processors
  - 2-cores, 4-cores, 8 cores becoming standard (Intel 80-cores!)
    - A system written for n-cores could potentially see an 80x speed-up when run on such a machine (very hard to achieve linear speed ups, however!)

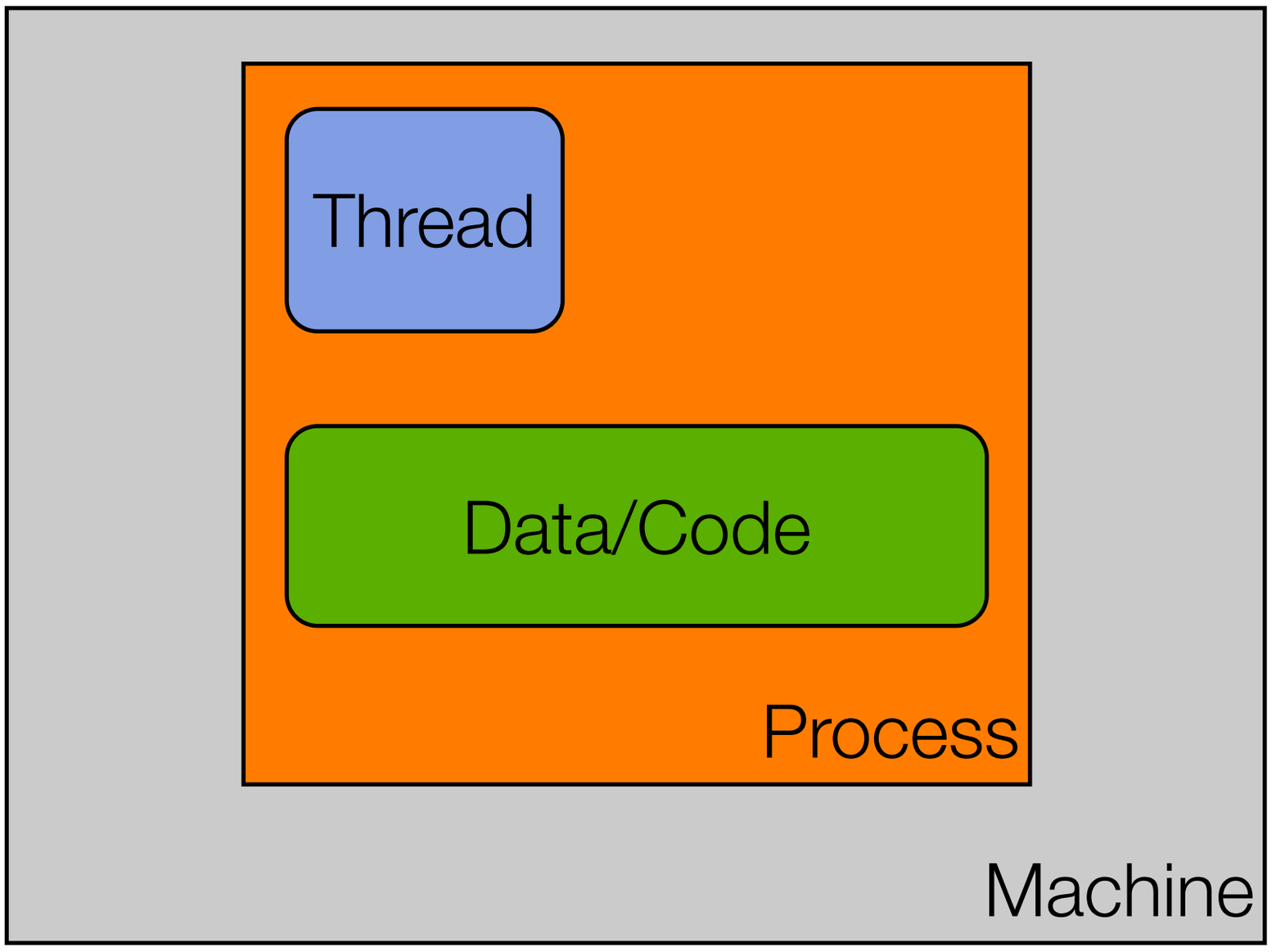
In addition…

- Concurrent programming is becoming hard to ignore
  - In addition to the increasing presence of multi-core computers there are lots of other 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: examples of multi-threaded GUI applications
    - without threads the UI would block as information is downloaded

BUT…

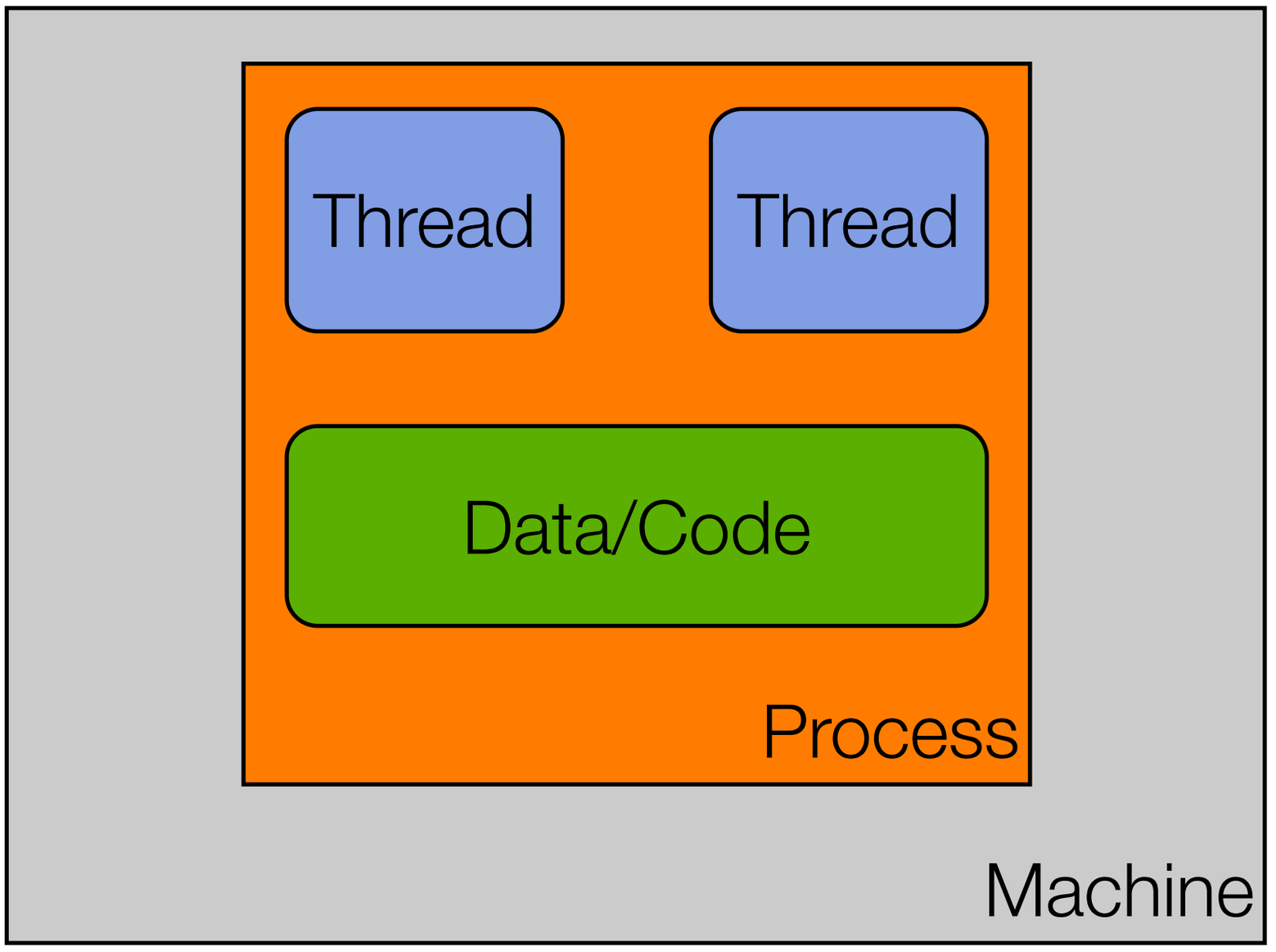
- While concurrency is widespread it is also error prone
  - Programmers trained on single-threaded programs face unfamiliar problems: synchronization, race conditions, deadlocks, etc.
- Example: Therac-25
  - Concurrent programming errors contributed to accidents causing death and serious injury
  - Mars Rover
    - Problems with interaction between concurrent tasks caused periodic software resets reducing availability for exploration

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



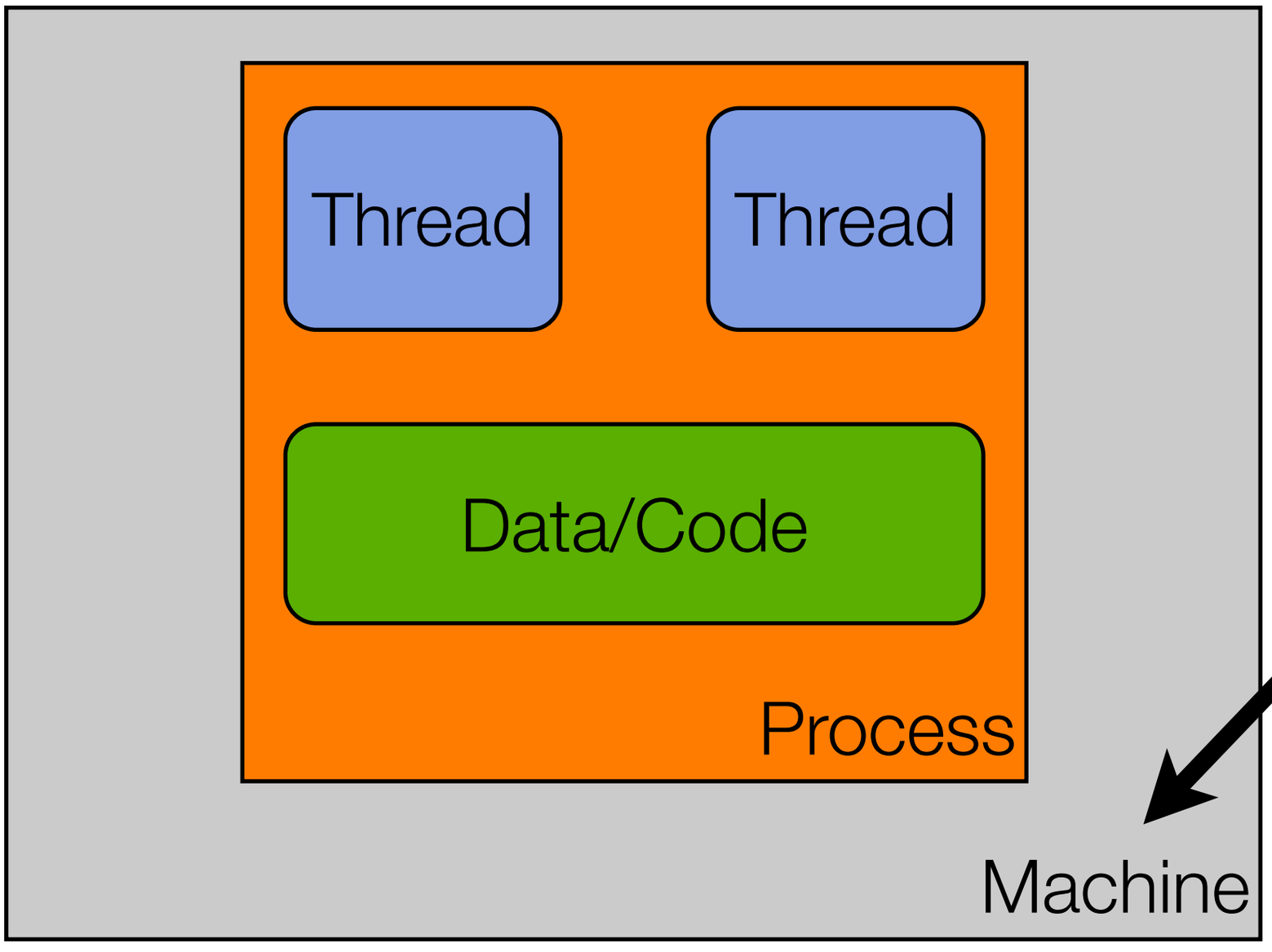
Sequential Program == Single Thread of Control

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



Concurrent Program == Multiple Threads of Control

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



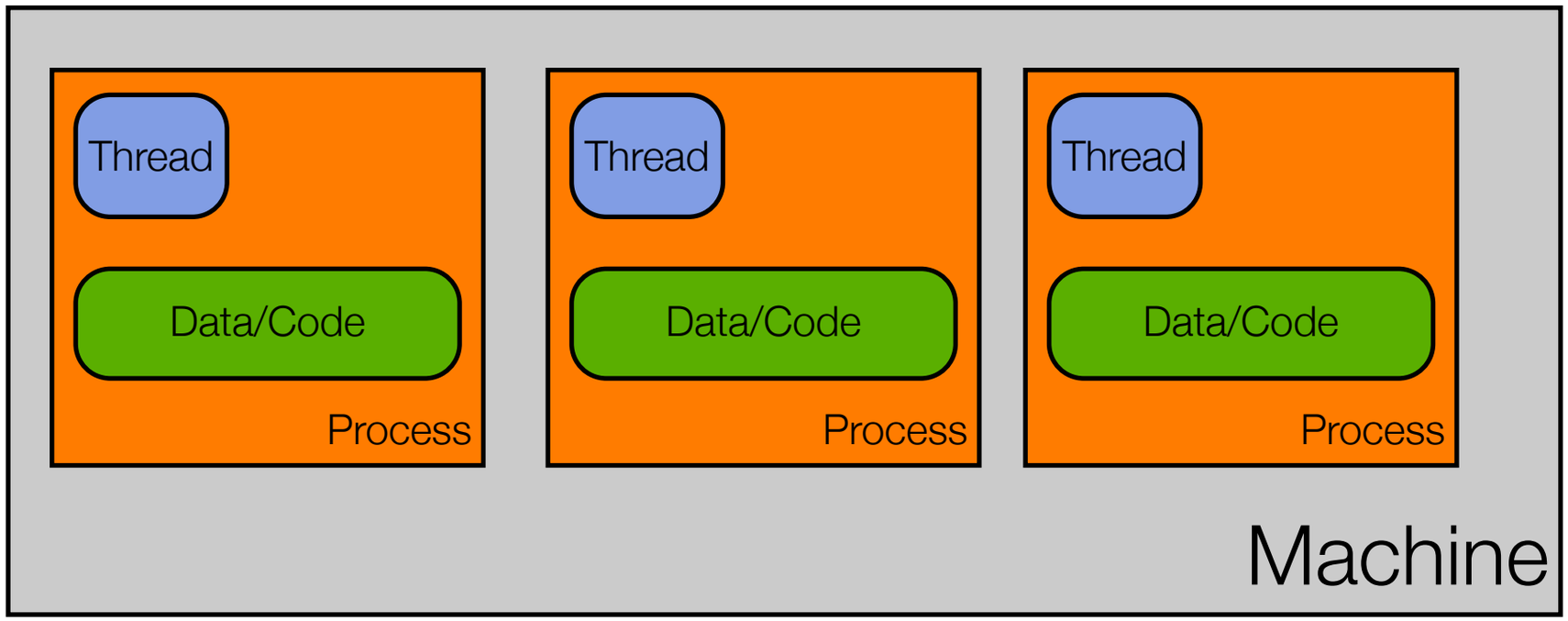
We may have multiple threads in this process, but we may not have events truly occurring in parallel. Why not?

It depends on the machine!

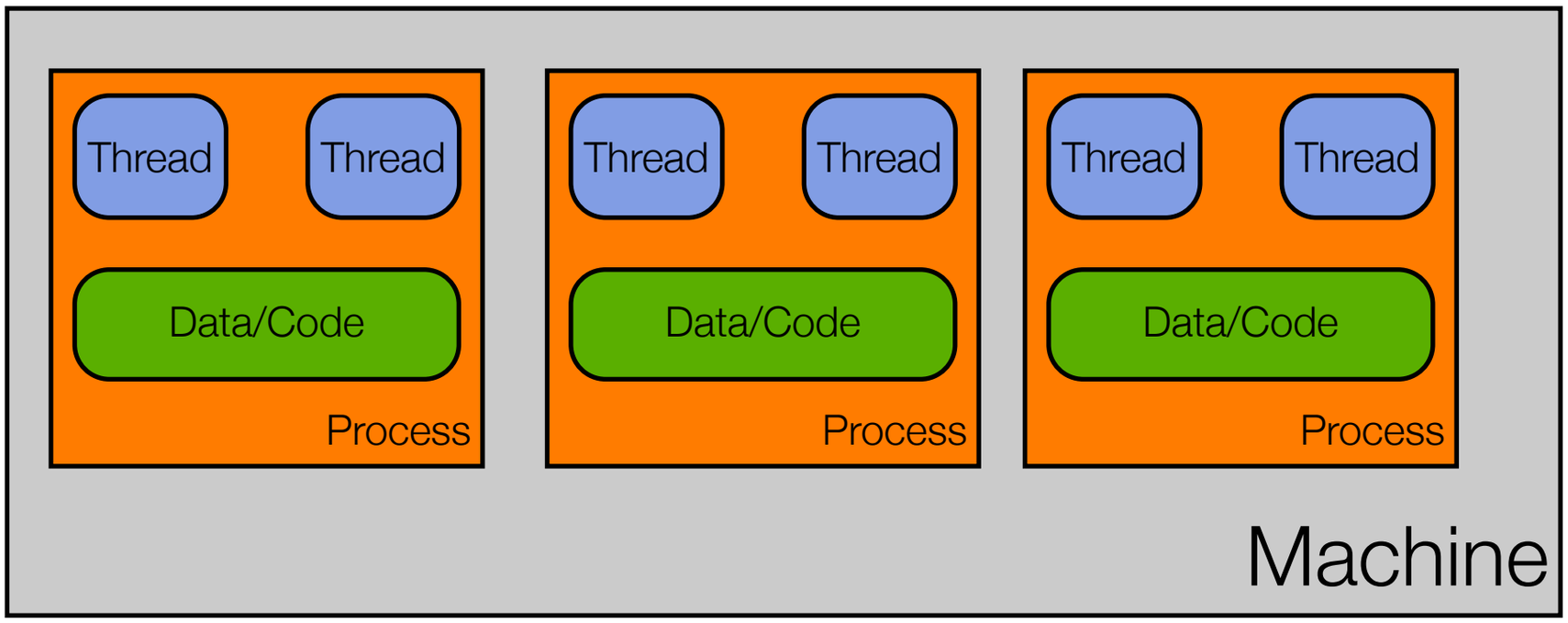
If the machine has multiple processors, then **true parallelism can occur**. Otherwise, parallelism is **simulated**

Concurrent Program == Multiple Threads of Control

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

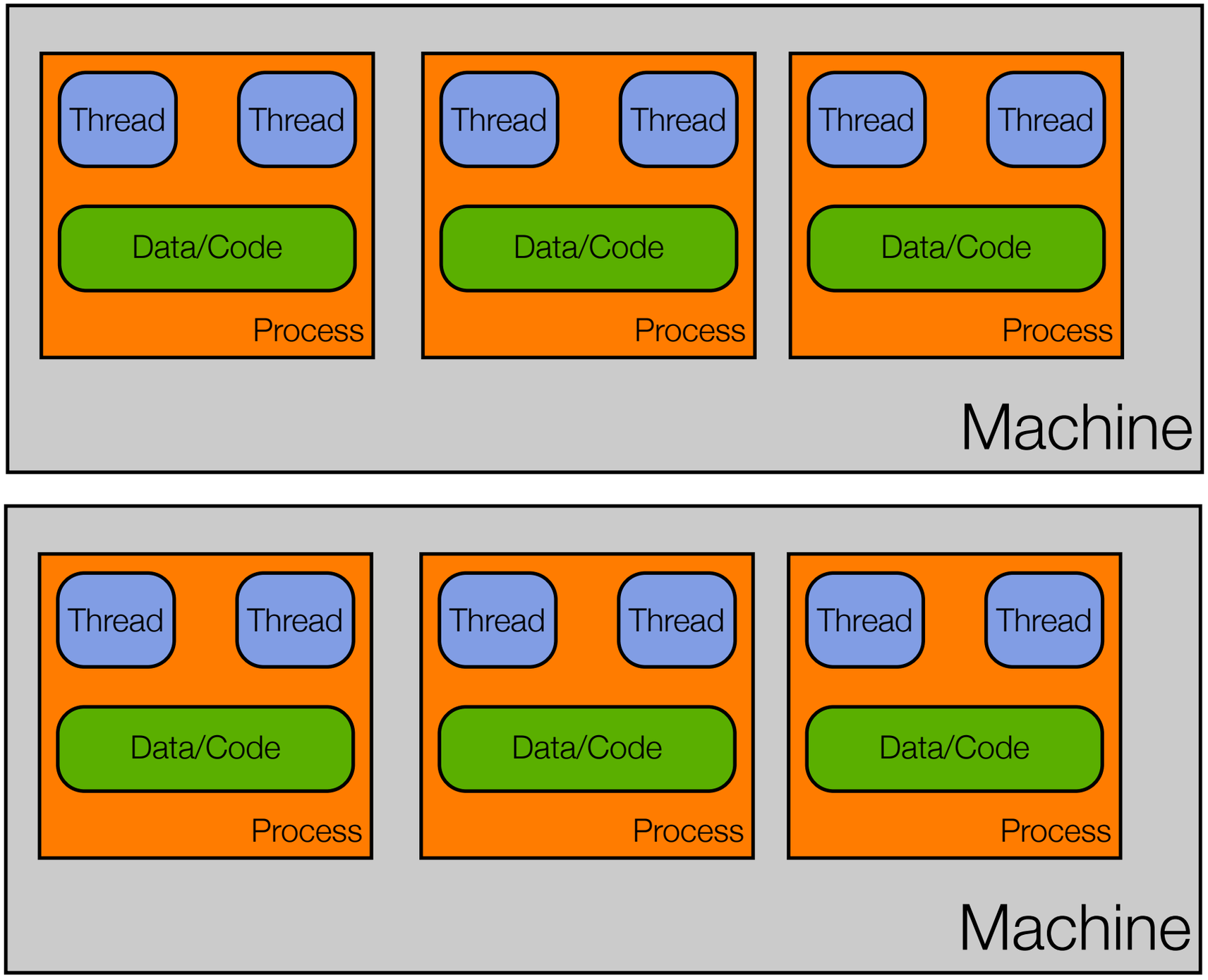


Basics: **Multi**-thread, **Multi**-Process, Single Machine



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

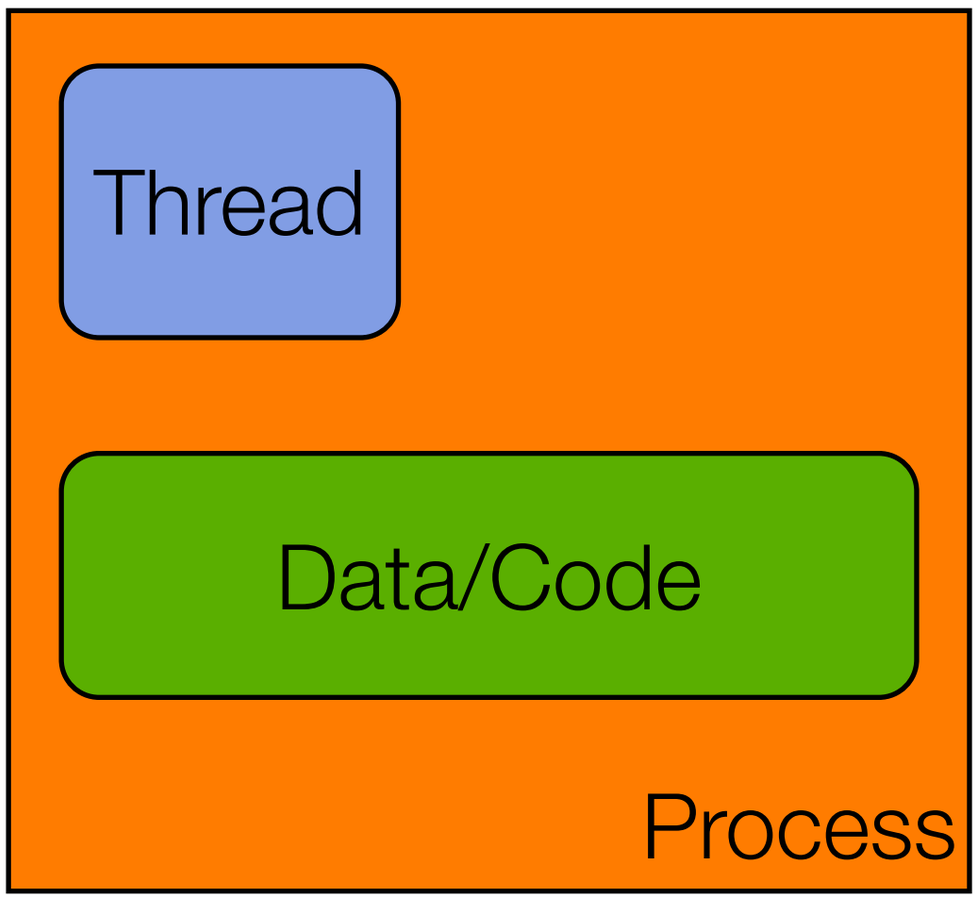
Basics: **Multi-everything**



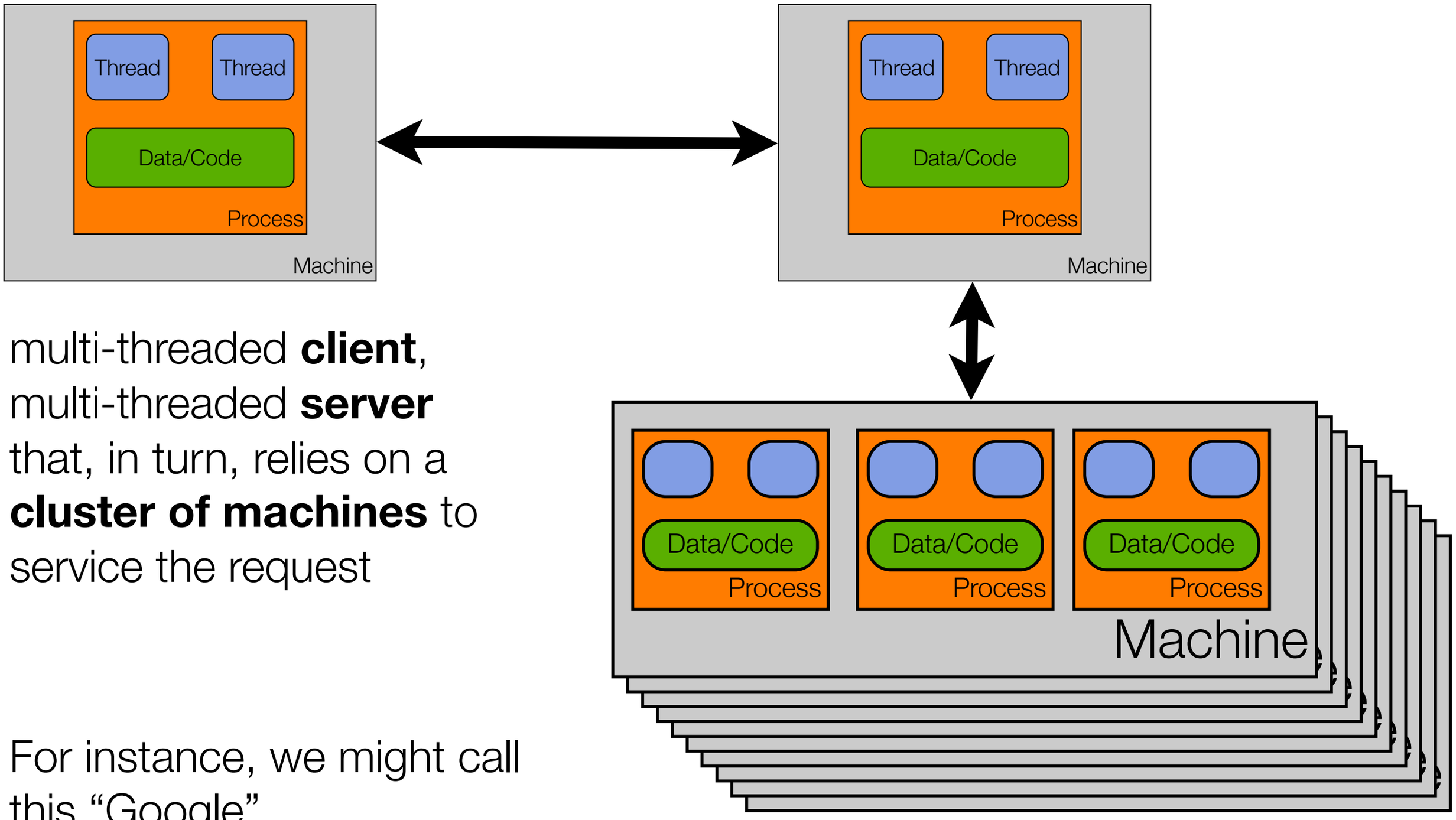
Applications are Dead! Long Live Applications!

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”

Architecture Design Choices

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

Consider Chrome (I)

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

Consider Chrome (II)

- 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

- We can use process monitoring capabilities 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, …

* Taken from discussion in Cocoa Programming for Mac OS X by Aaron Hillegass

Example

- Developing our own concurrent applications
  - Let's look at the performance of
    - a single threaded program
    - a multi-threaded program
    - a multi-process program
  - all trying to perform the same task
- Searching for files that contain a particular search term

Background

- Our program will be searching 741 MB of data split across ~108,000 text files
- Files are stored as the leaves of a “tree” of folders
  - 11 folders at the top
    - In general, each contain 10 sub-folders, each with 1000 articles
- The articles themselves contain blog posts about the political turmoil that surrounded the recent election in Iran

Range of Solutions

- Developed 5 ruby programs to explore
  - single threaded, multi threaded & multi process
- approaches to solving this problem
  - Single threaded approach iterates over entire directory structure and maintains a count as it goes along
  - Multiprocess approach creates one process per top level directory -> invokes single threaded program on each
  - Multithreaded approach creates one thread per top level directory; each thread acts like single-threaded program
- Demo

Times

times in seconds	Single	MultiThreaded	MultiProcess
real	75.775	42.735	48.94
user	12.46	11.08	9.655
sys	17.95	13.53	10.735

Times are averages of two separate runs of each program

Underwhelming (I)

- If we take a look at the user and system times, there is almost no difference
  - Why?

Underwhelming (II)

- If we take a look at the user and system times, there is almost no difference
  - Lots of possible reasons
    - Shared Disk (primary)
    - Low processor utilization for each program
      - (waiting for the disk?)
    - Use of scripting language
      - (anyone want to write these programs in C?)

Mem vs. Disk?

- To test the theory that disk I/O dominated the results, I also created two programs that
  - read all articles into memory
  - and then perform the search
  - one program was single threaded
  - the other was multi process

Times

times in seconds	Single	MultiProcess
real	87.38	43.9
user	15.165	10.21
sys	20.455	11.105

Times are averages of two separate runs of each program

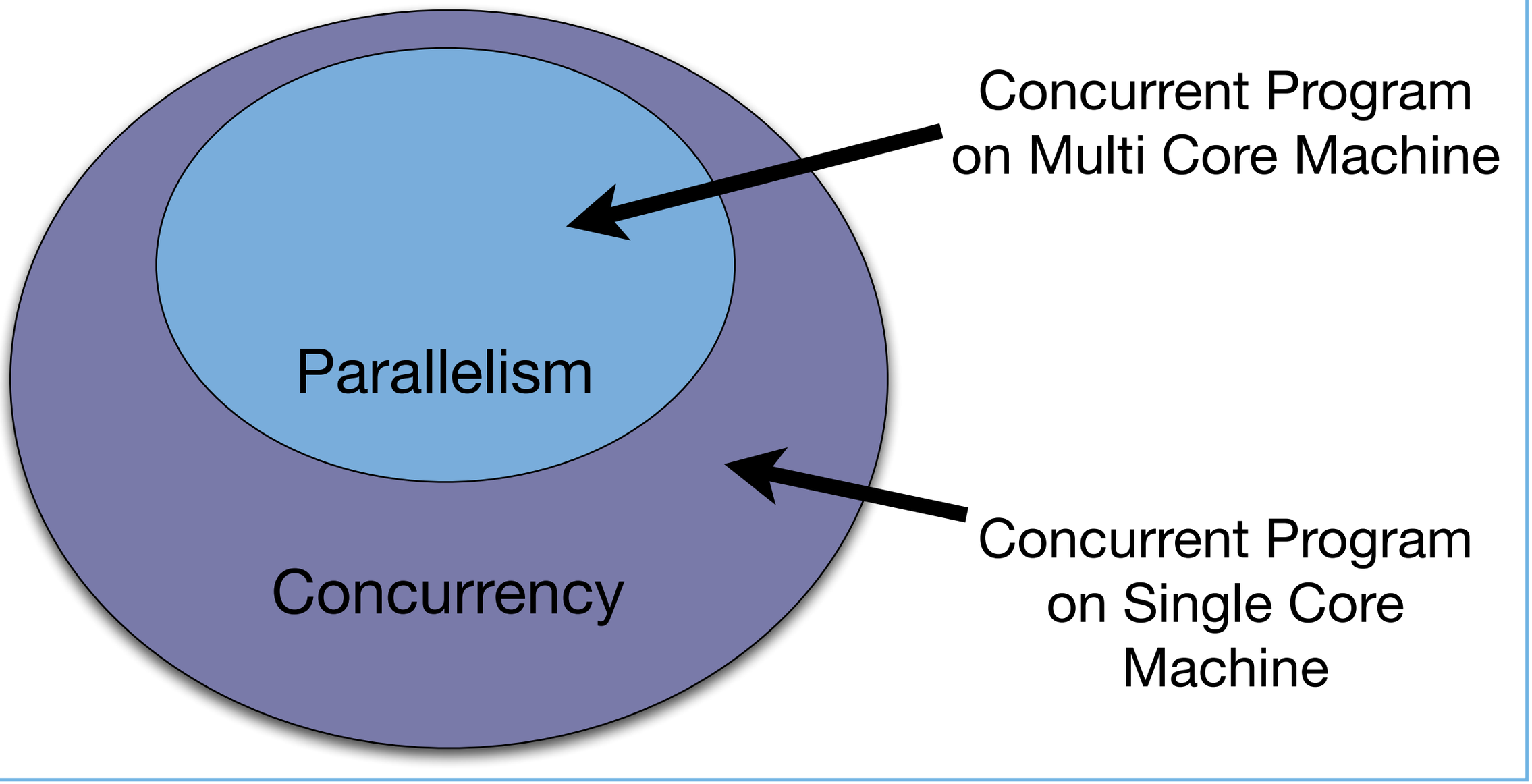
Slightly better

- However, my numbers between runs had high variability
  - so I can't be 100% confident in these numbers
- I have a new respect for the area of performance modeling!

On to the textbook...

- 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 that coordinates access to the single processor among the various processes
  - If a machine has more than a single processor, **then true parallelism can occur**: you can have N processes running simultaneously on a machine with N processors

Thus...


Concurrent Program
on Multi Core Machine
Parallelism
Concurrency
Concurrent Program
on Single Core
Machine

Potential Speed Up?

- As an upper bound, performance improvements for concurrent applications over single-threaded applications
  - two cores → 200% speed up (runs in half the time)
  - four cores → 400% speed up (runs in quarter the time)
  - etc.
- Better than the typical increase of 20-30% provided by new single-core CPUs
- However, it is extremely difficult to achieve these speedups
  - Creating a scalable concurrent system is very hard

Why? (I)

- Overhead
  - Converting a non-concurrent program into a concurrent program adds overhead
    - You may need to rearrange data structures
    - You have to add code to manage threading
      - creating threads, waiting for them to end, querying them, passing information between them, etc.
      - The single threaded program had NONE of this extra code
  - In order to see a performance gain, there must be enough work for the threads to do to trivialize the overhead

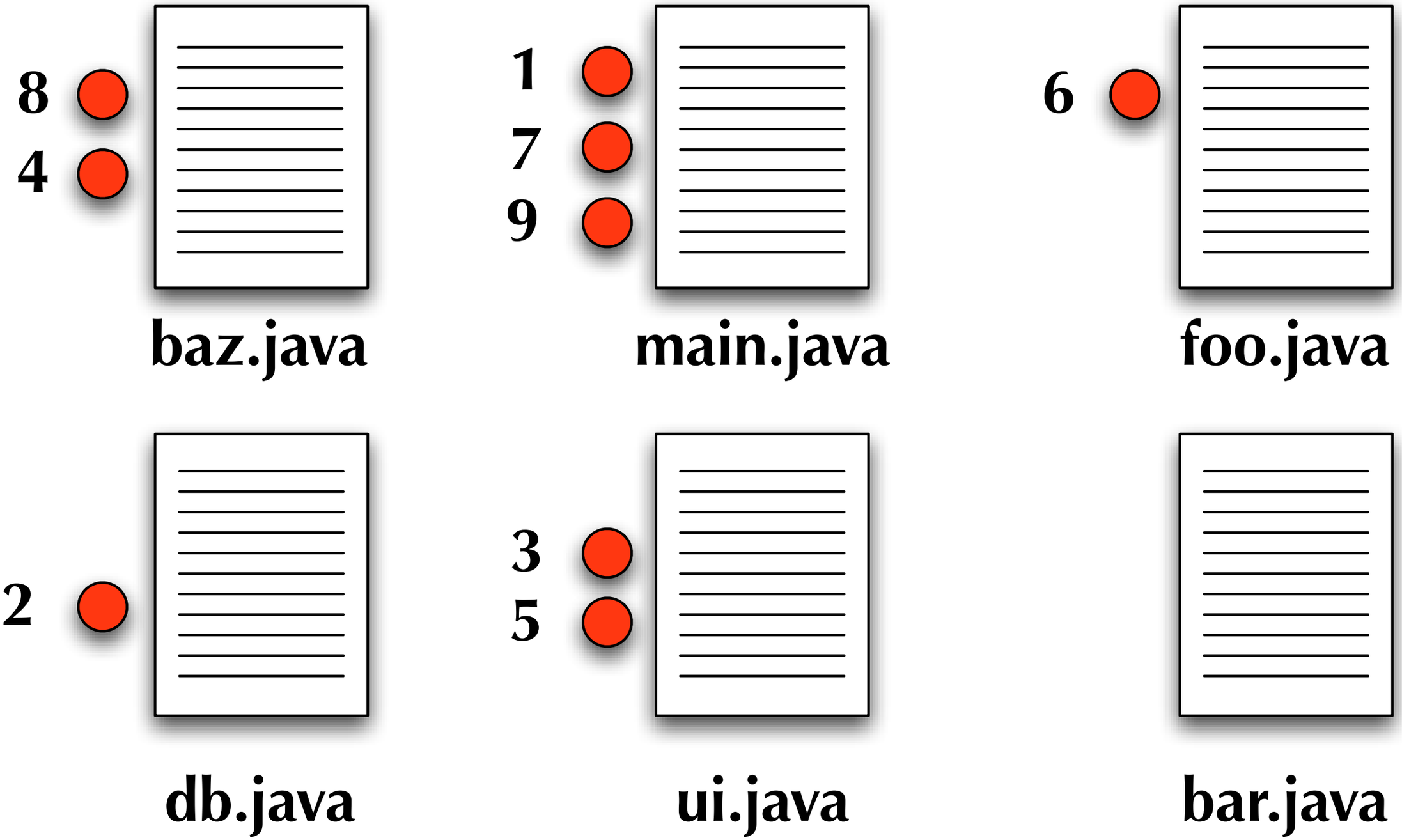
Why? (II)

- As we saw with the example program
  - It can be hard to determine why a concurrent program is or is not running faster than a single threaded program
    - The OS may not assign your program a high enough priority to run at top speed
    - The program may be I/O bound and spend most of its time waiting
      - More generally, the program may encounter bottlenecks that end up blocking any gains you might have seen from threads or multiple processes

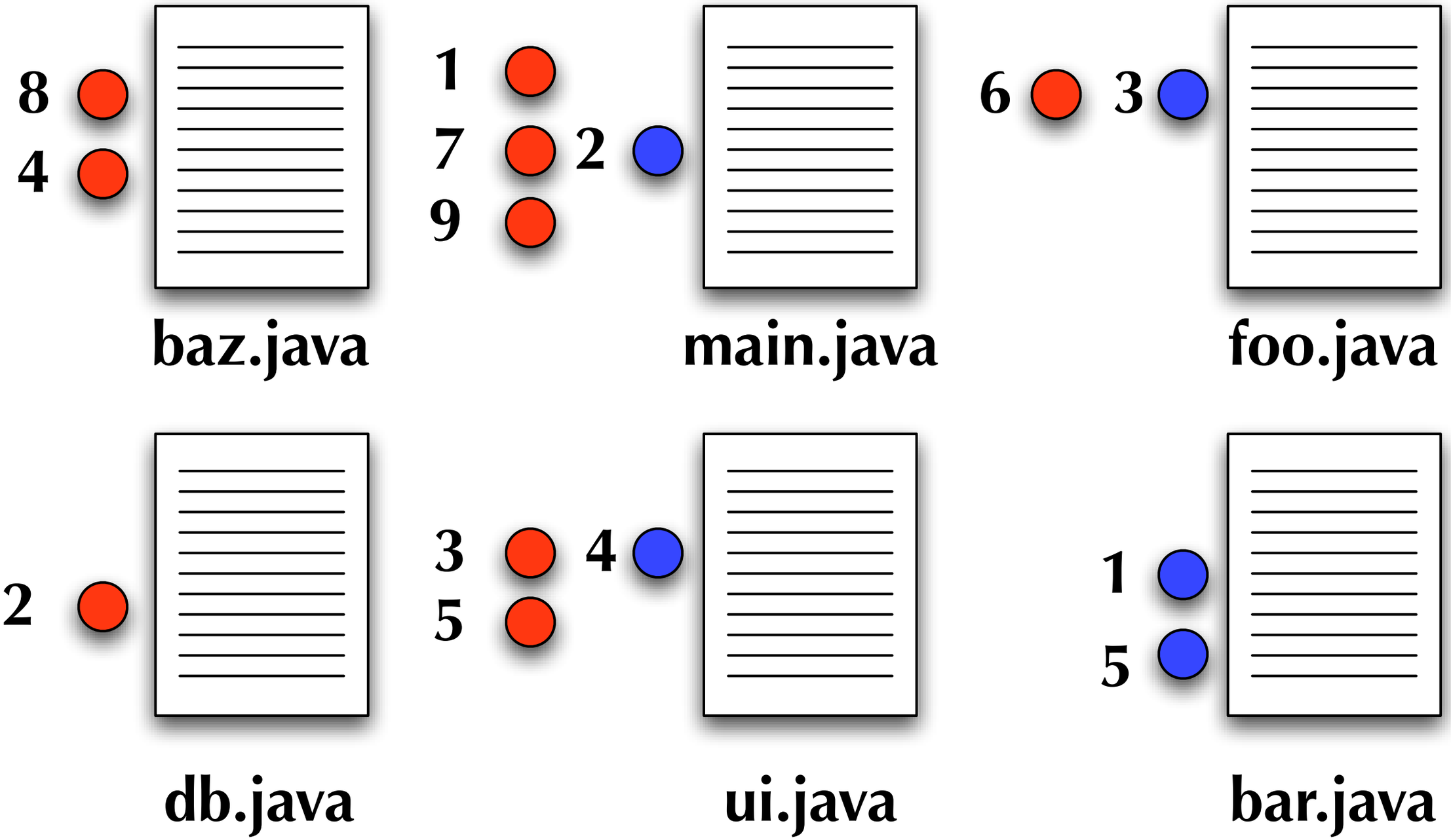
In addition…

- Designing, implementing & testing concurrent programs is hard
  - Much harder than testing sequential programs due to
    - **interference**: two threads accessing shared data inappropriately
    - **race conditions**: behaviors that appear in one configuration but don't appear in other configurations
    - **deadlock**: threads block waiting for each other
- To guard against these problems, you need synchronization to protect shared memory, which slows programs down

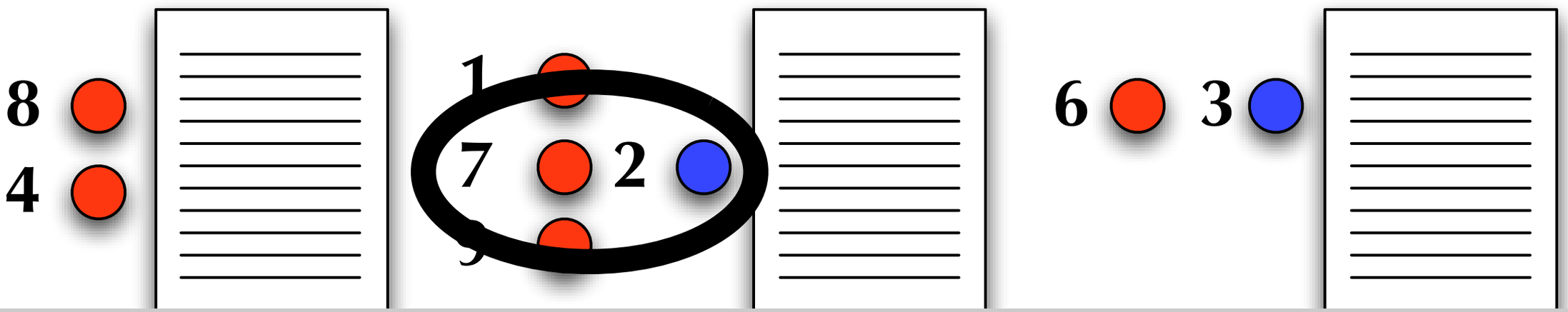
Graphically: Sequential Program



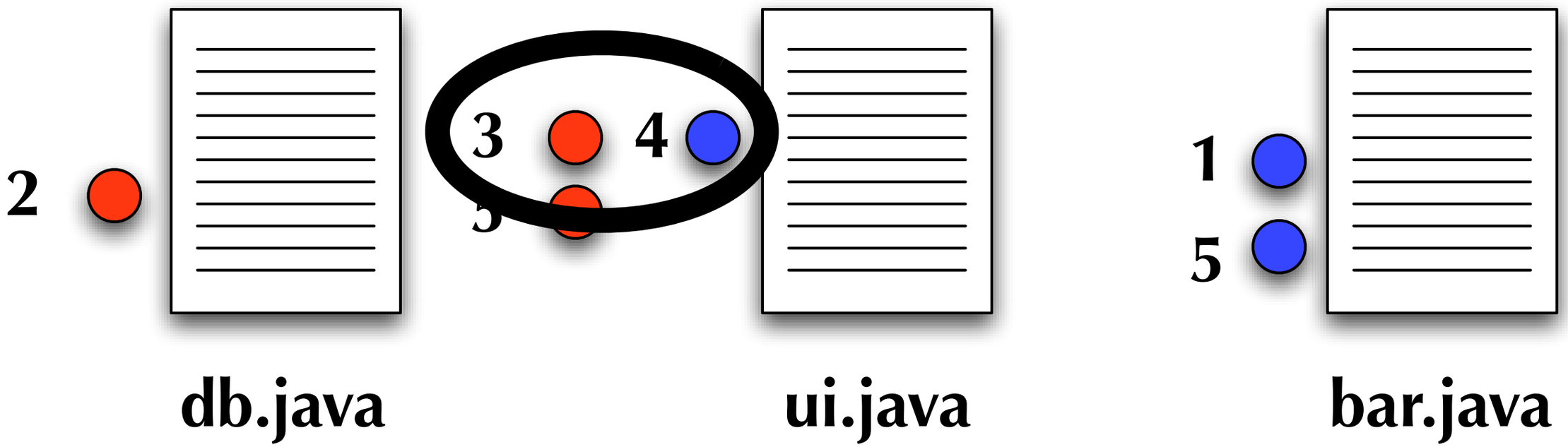
Graphically: Concurrent Program



Example of Interference



The potential for interactions… two threads hitting the same method at the same time, potentially corrupting a shared data structure



Other problems…

- With serial programs, execution takes a predictable path
  - Logic errors can be tracked down systematically and with good tool support
- With concurrent programs, developer must keep track of multiple execution paths whose instructions can arbitrarily interleave
  - Two threads with four instructions each: 70 different ways to interleave their instructions!
  - Tool support is minimal and errors do not behave predictably
    - Indeed, adding debug statements can “fix” problems

Benefits of Concurrent Programming?

- Performance gain from multi-core hardware
  - True parallelism
- Increased application throughput
  - an I/O call need only block one thread
- Increased application responsiveness
  - high priority thread for user requests
- More appropriate structure
  - for programs which interact with the environment, control multiple activities, and handle multiple events
  - by partitioning the application's thread/process structure to match its external conditions (e.g. one thread per activity)

Threading Methodology

- Breshears presents a threading methodology
  - First produce a tested single-threaded program
    - Use reqs./design/implement/test/tune/maintenance steps
  - Then to create a concurrent system from the former, do
    - Analysis: Find computations that are independent of each other
      - AND take up a large amount of serial execution time (80/20 rule)
    - Design and Implement: straightforward
    - Test for Correctness: Verify that concurrent code produces correct output
    - Tune for performance: once correct, find ways to speed up

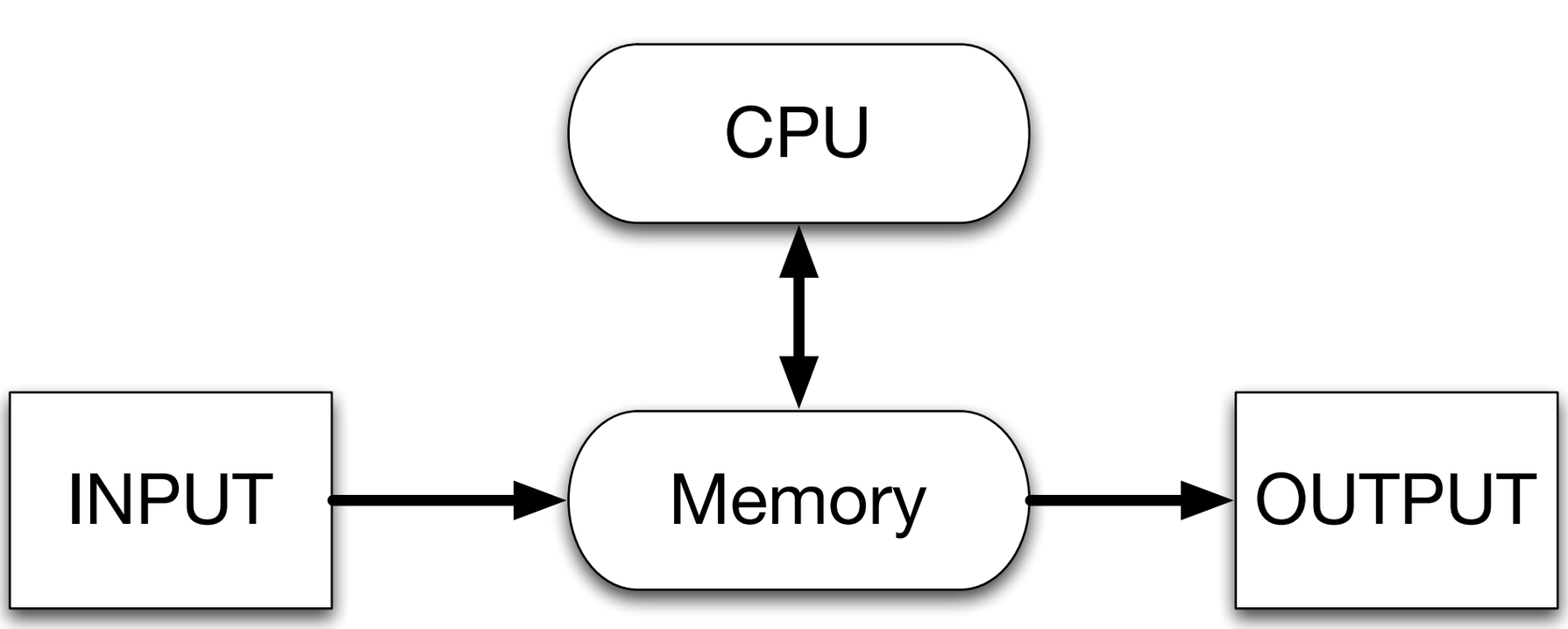
Note: does not recommend going straight to concurrency!

Performing Tuning

- Tuning threaded code typically involves
  - identifying sources of contention on locks (synchronization)
  - identifying work imbalances across threads
  - reducing overhead
- Testing and Tuning
  - Whenever you tune a threaded program, you must test it again for correctness
- Going back further: if you are unable to tune system performance, you may have to re-design and re-implement

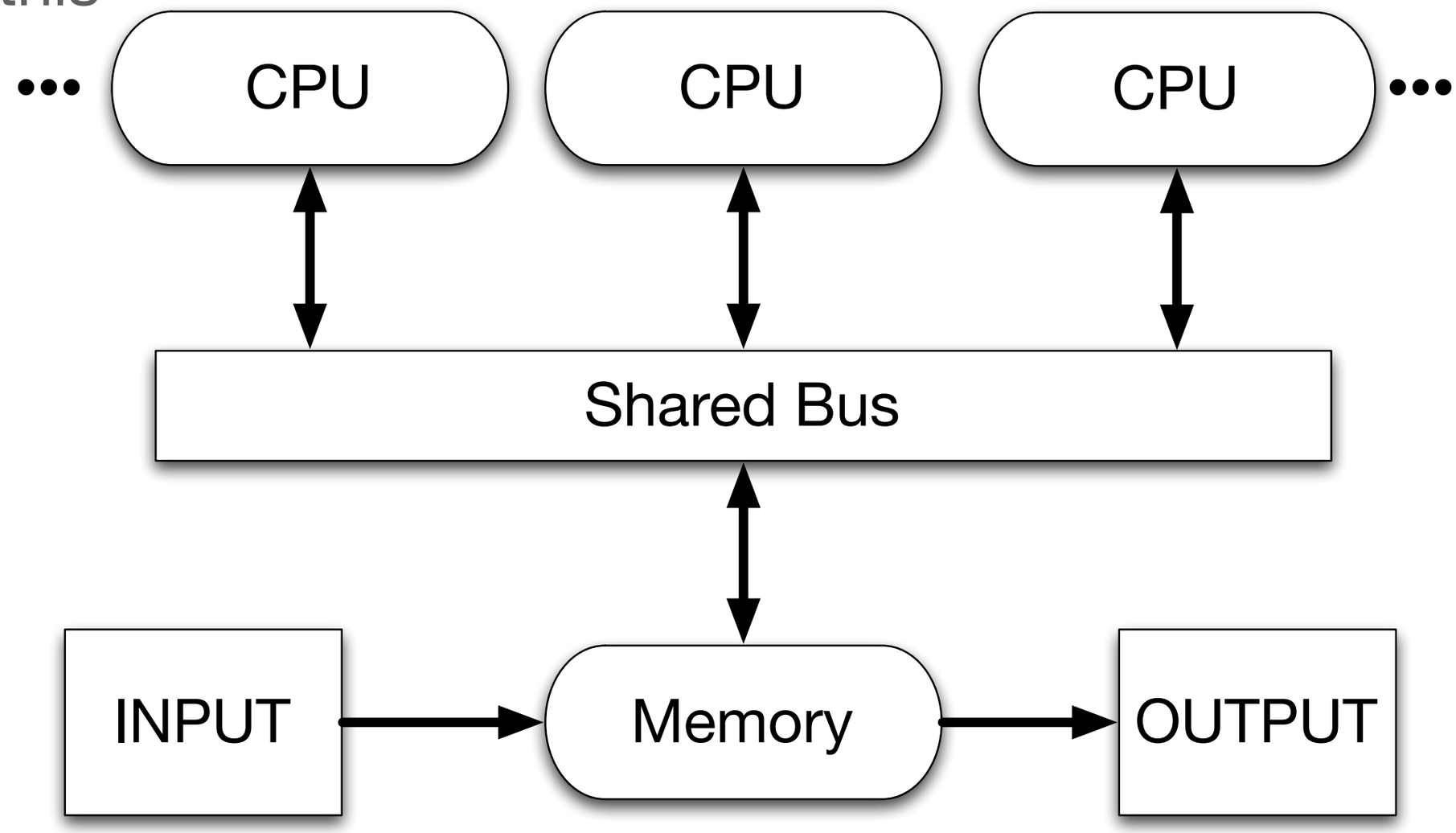
Parallel Algorithms (Intro)

- In looking at the development of parallel algorithms, the standard Von Neumann architecture is modified, from this



Parallel Algorithms (Intro)

- to this



Wrapping Up

- Concepts
  - Introduced notion of concurrency systems
    - And the related notion of parallelism
- Presented an example single threaded system along with multi-threaded and multi-process variants
- Discussed benefits and problems of concurrent programming
- Reviewed Breshears recommended threading methodology

Coming Up Next

- Lecture 5: Gathering Requirements
  - Chapter 2 of Pilone & Miles
- Lecture 6: Concurrent or Not Concurrent
  - Remainder of Chapter 1 material
  - Begin look at Chapter 2 of Breshears