Lecture 28: Concurrency

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/5448 — Lecture 28 — 12/03/2009

Goals for this Lecture

- Briefly review concepts behind concurrency in software systems
    - See examples of how to make use of concurrency in OO systems
    - Look at some of the problems that occur
        - Look at one non-OO approach to concurrency that avoids some of the problems

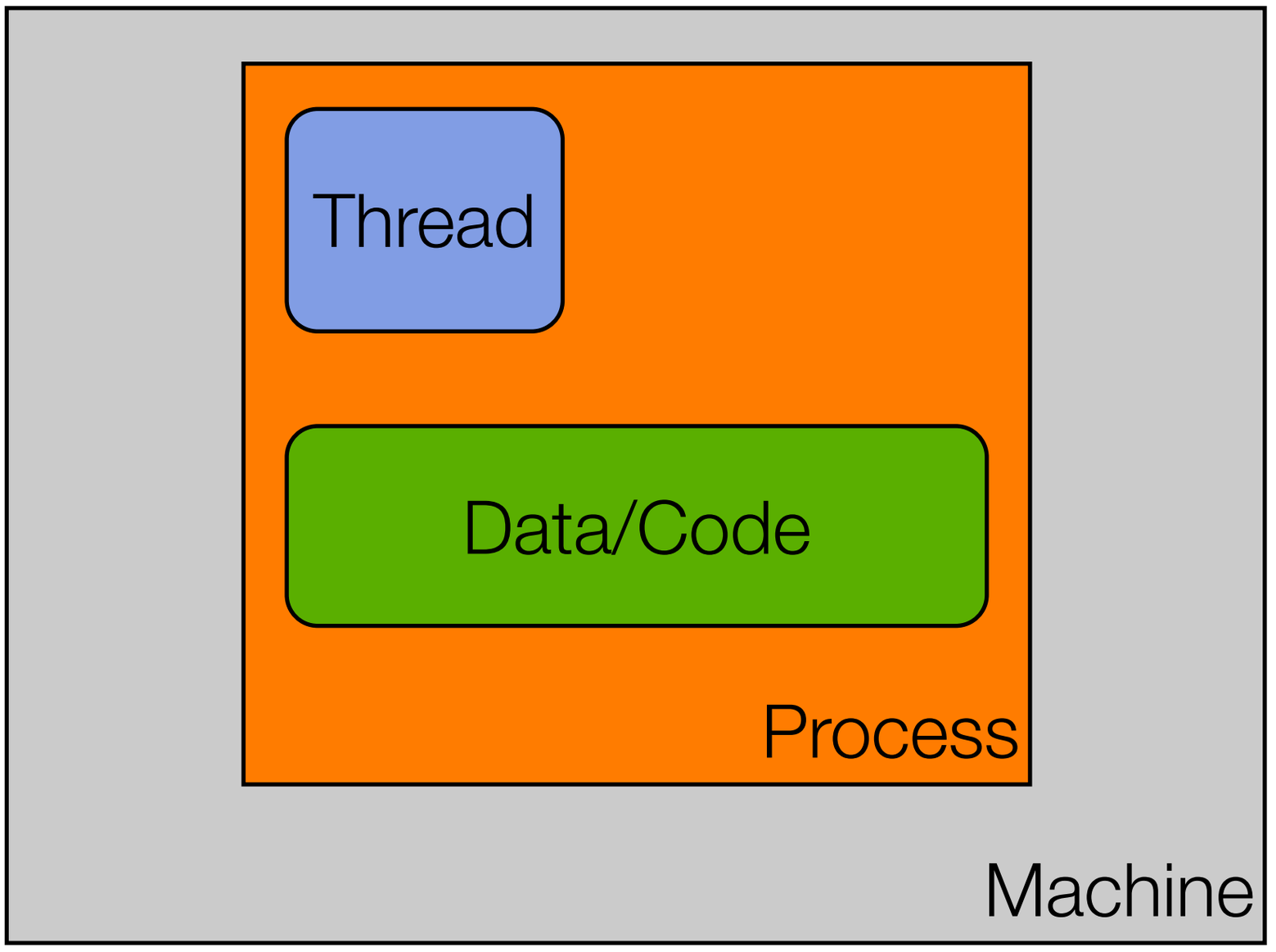
Why worry?

- Concurrency is hard and I've only ever needed single-threaded programs
    - Why should I care about it?
- Answer: multi-core computers
    - Growth rates for chip speed are flattening
        - You can no longer say "lets wait a year and our system will run faster!"
    - Instead, chips are becoming "wider"
        - more cores, wider bus (more data at a time), more memory on chip
- As chip 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

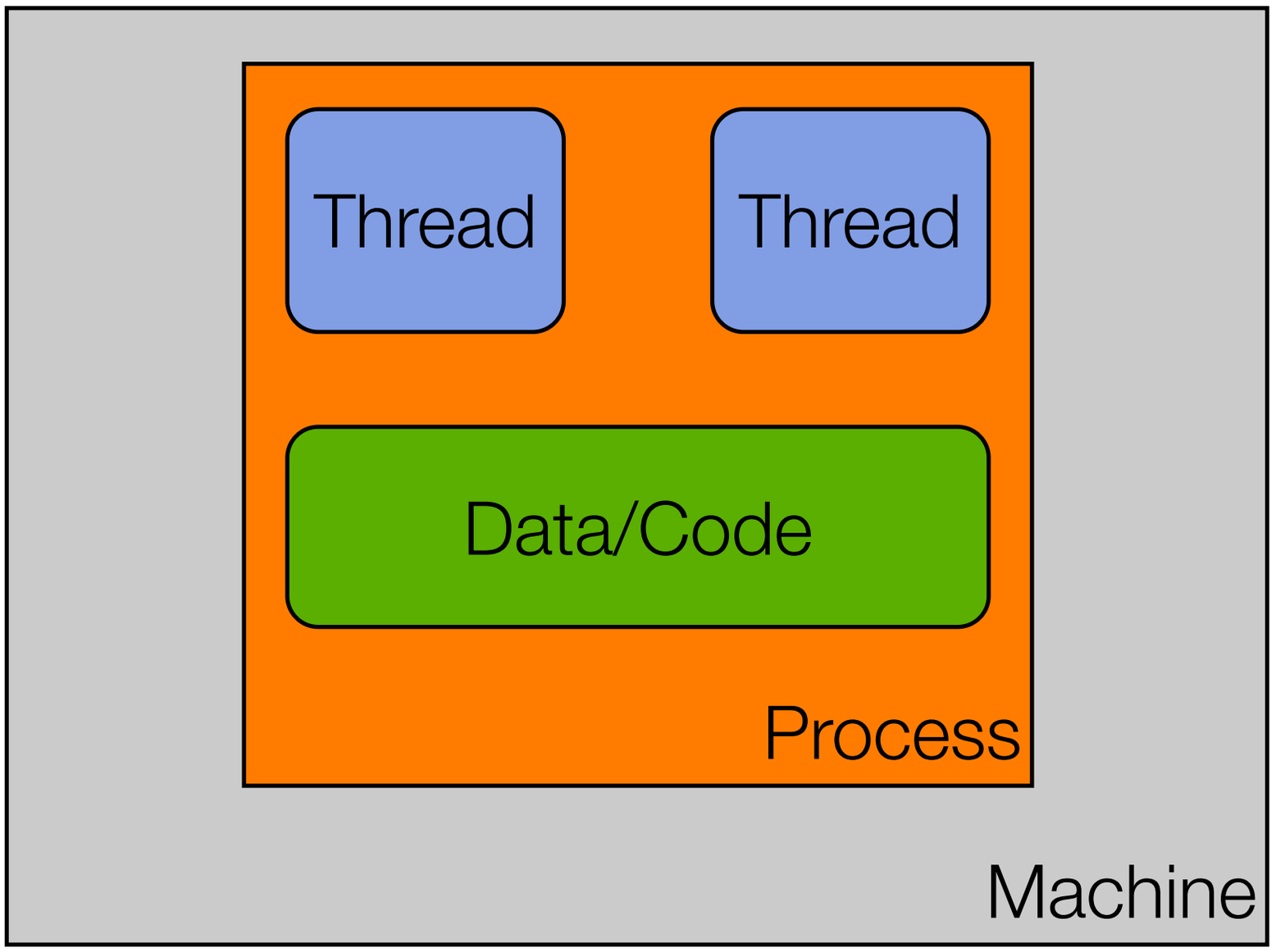
New Model

- Instead, the way in which software will see performance gains with new hardware is if they are designed to get faster the more processors they have available
    - This is not easy: the computations that an application performs has to be amenable to parallelization (that is, being split up into multiple parts that can be computed separately)
- If so, such an application will see noticeable speed improvements as it is put on machines with more and more processors.
    - Laptops currently have 2-cores, will soon have 4-cores, and for high-end machines Intel has an 80-core beast waiting in the wings
        - A system written for n-cores could potentially see an 80x speed-up when run on such a machine

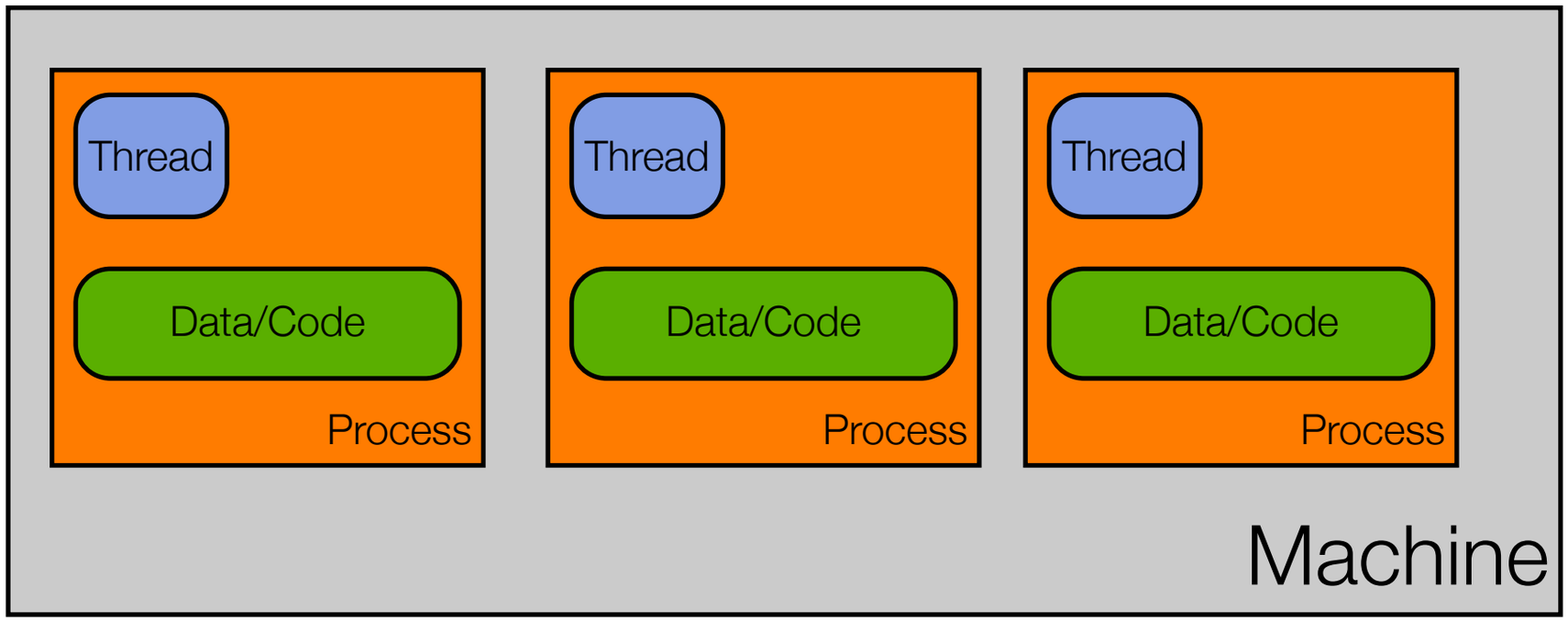
Basics: Single Thread, Single Process, Single Machine



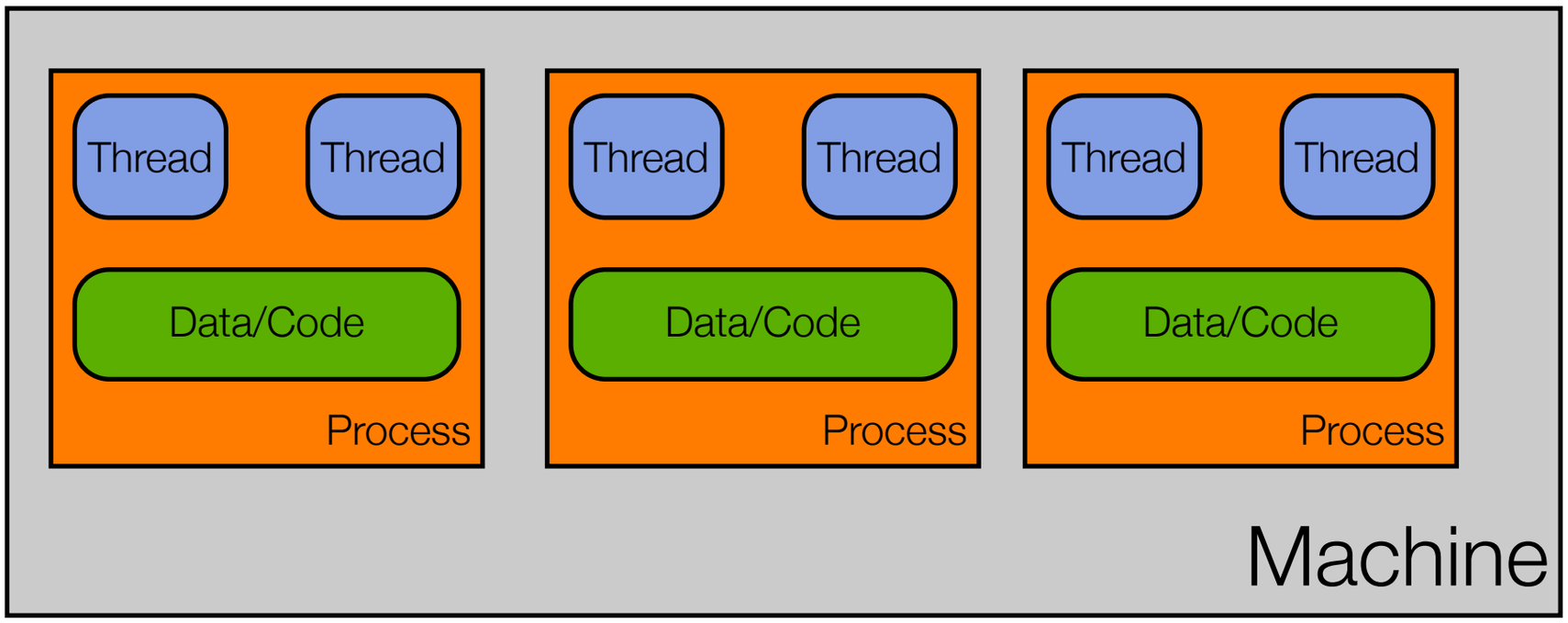
Basics: Multiple Thread, Single Process, Single Machine



Basics: Single Thread, Multiple Process, Single Machine

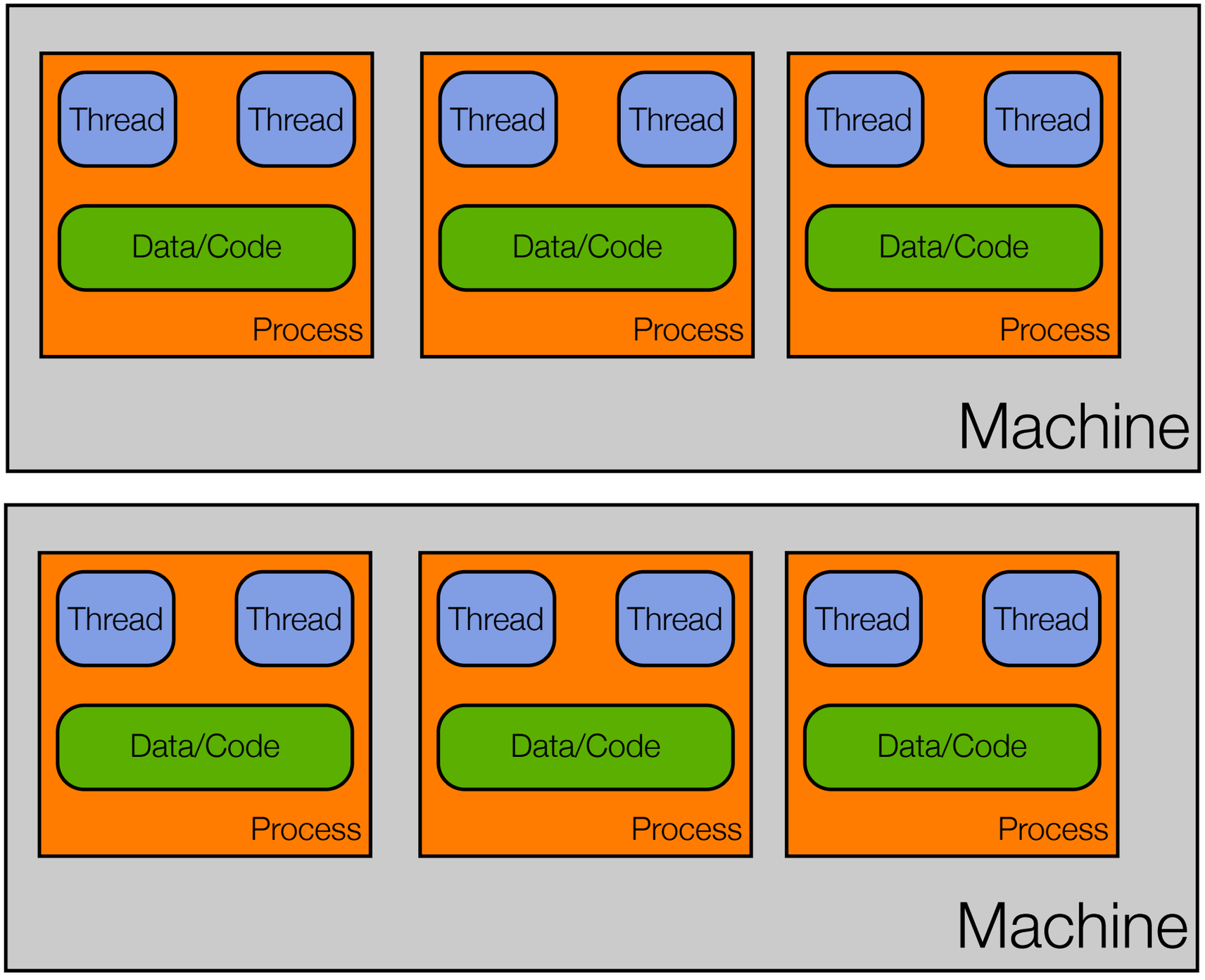


Basics: Multi-thread, Multi-Process, Single Machine



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

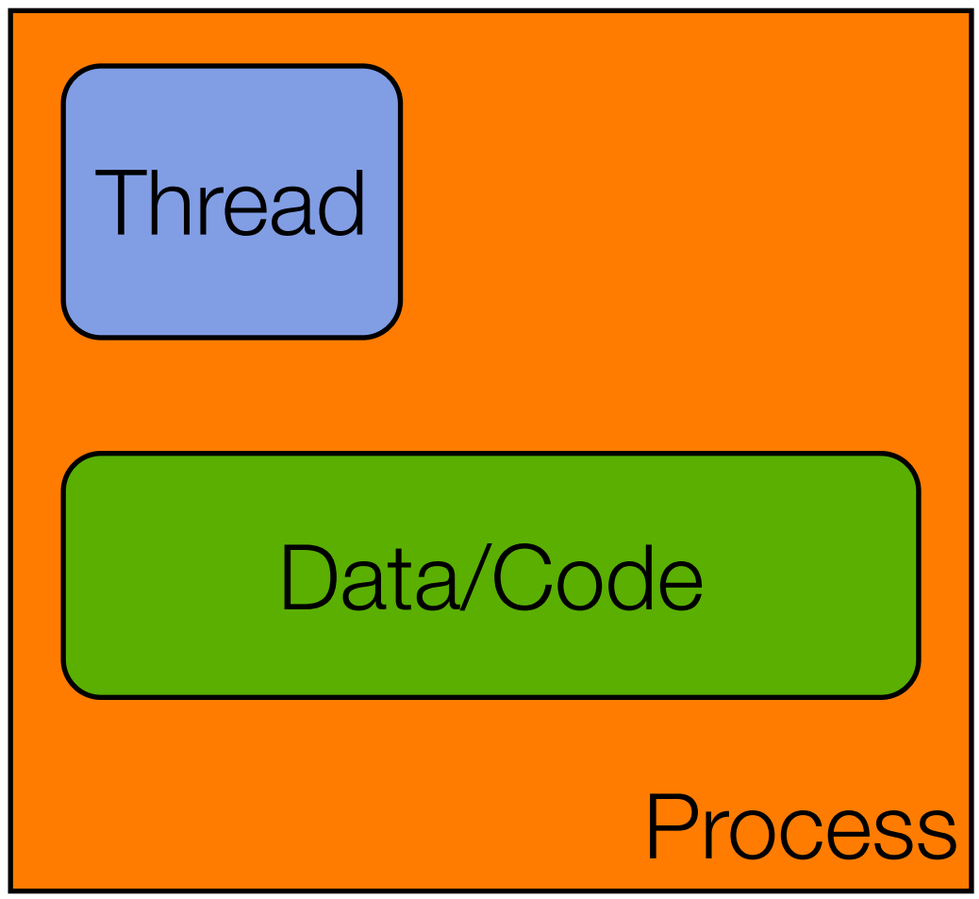
Basics: Multi-everything



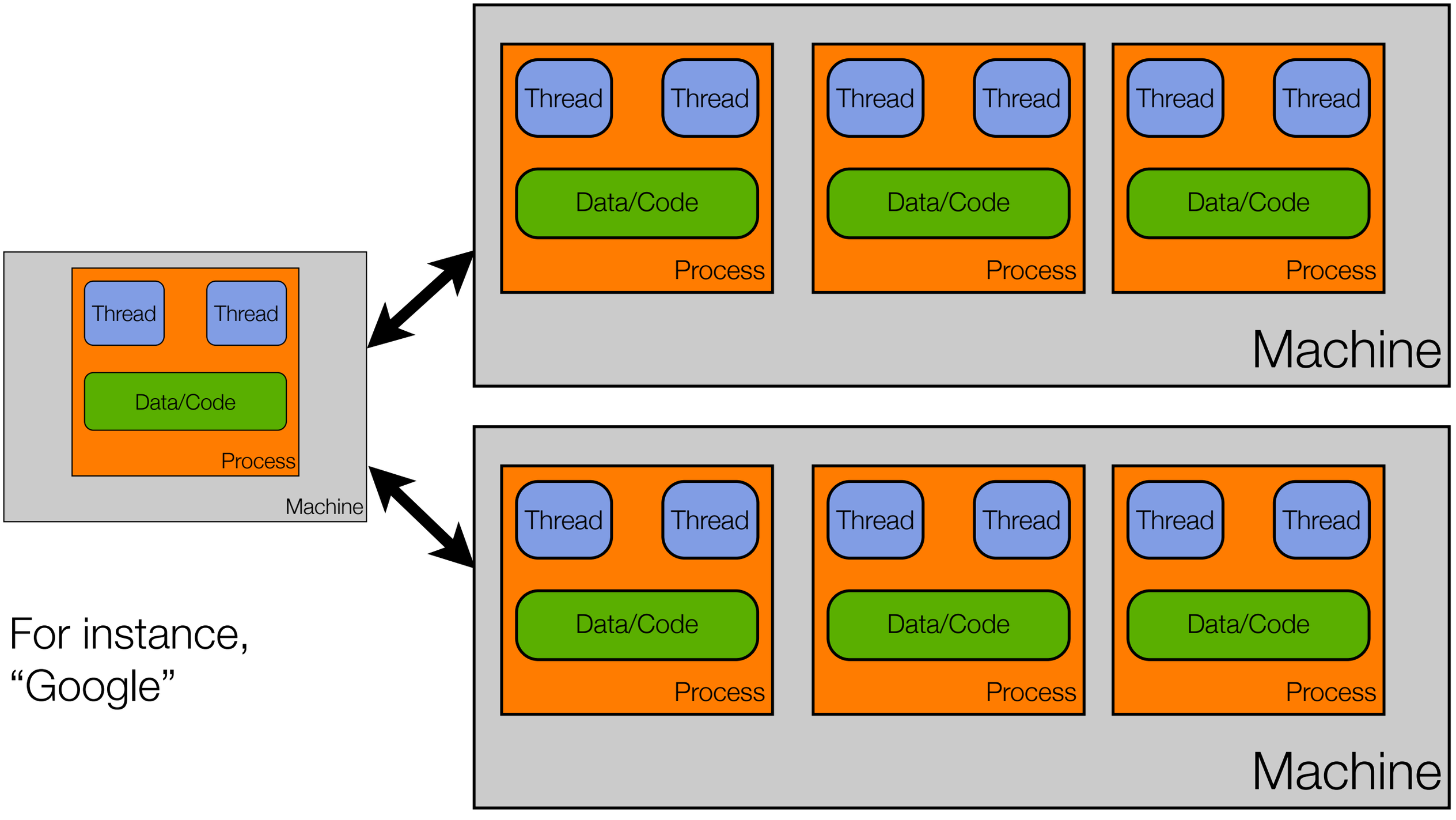
Applications are Dead! Long Live Applications!

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”

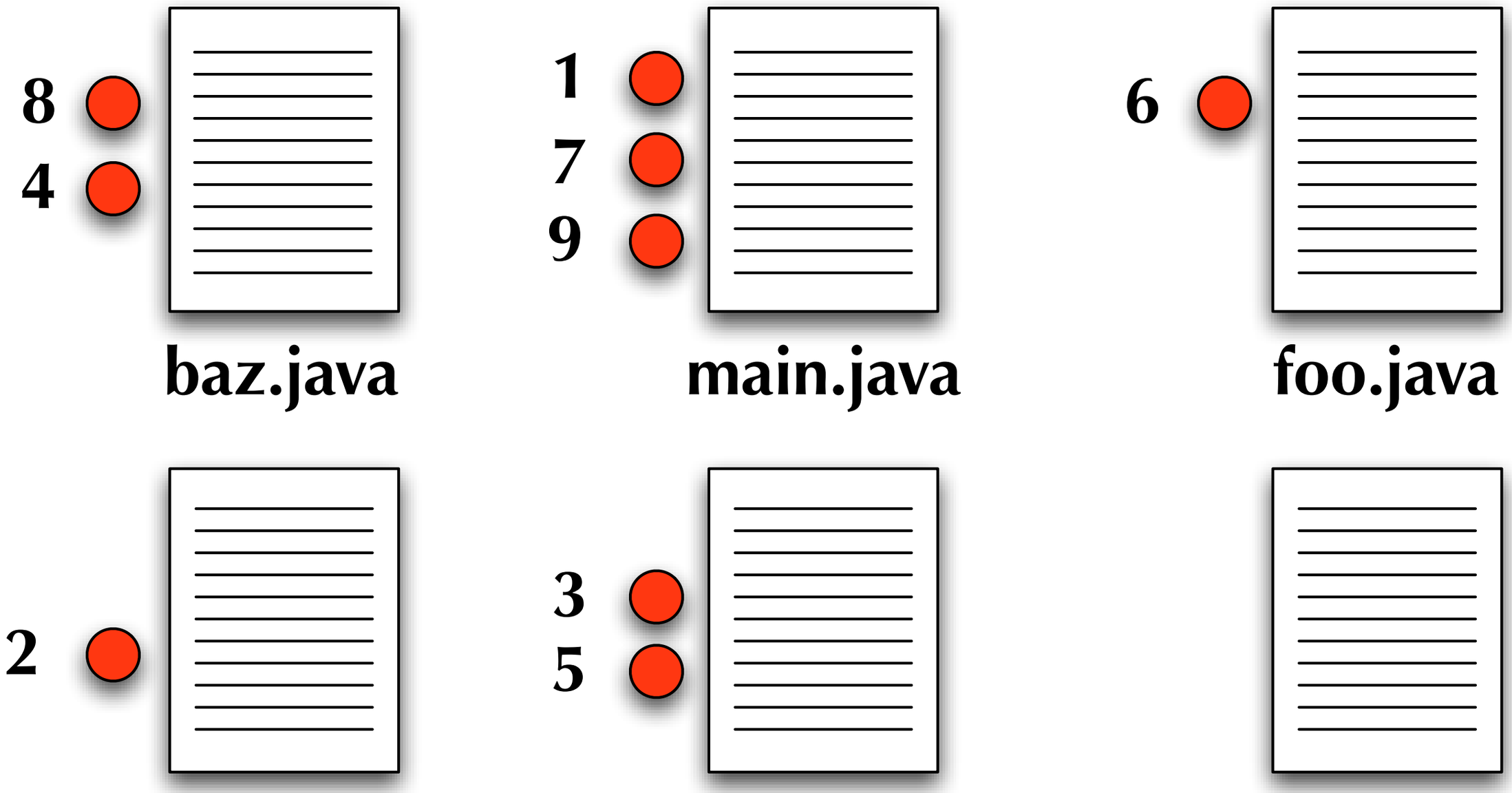


For instance,
“Google”

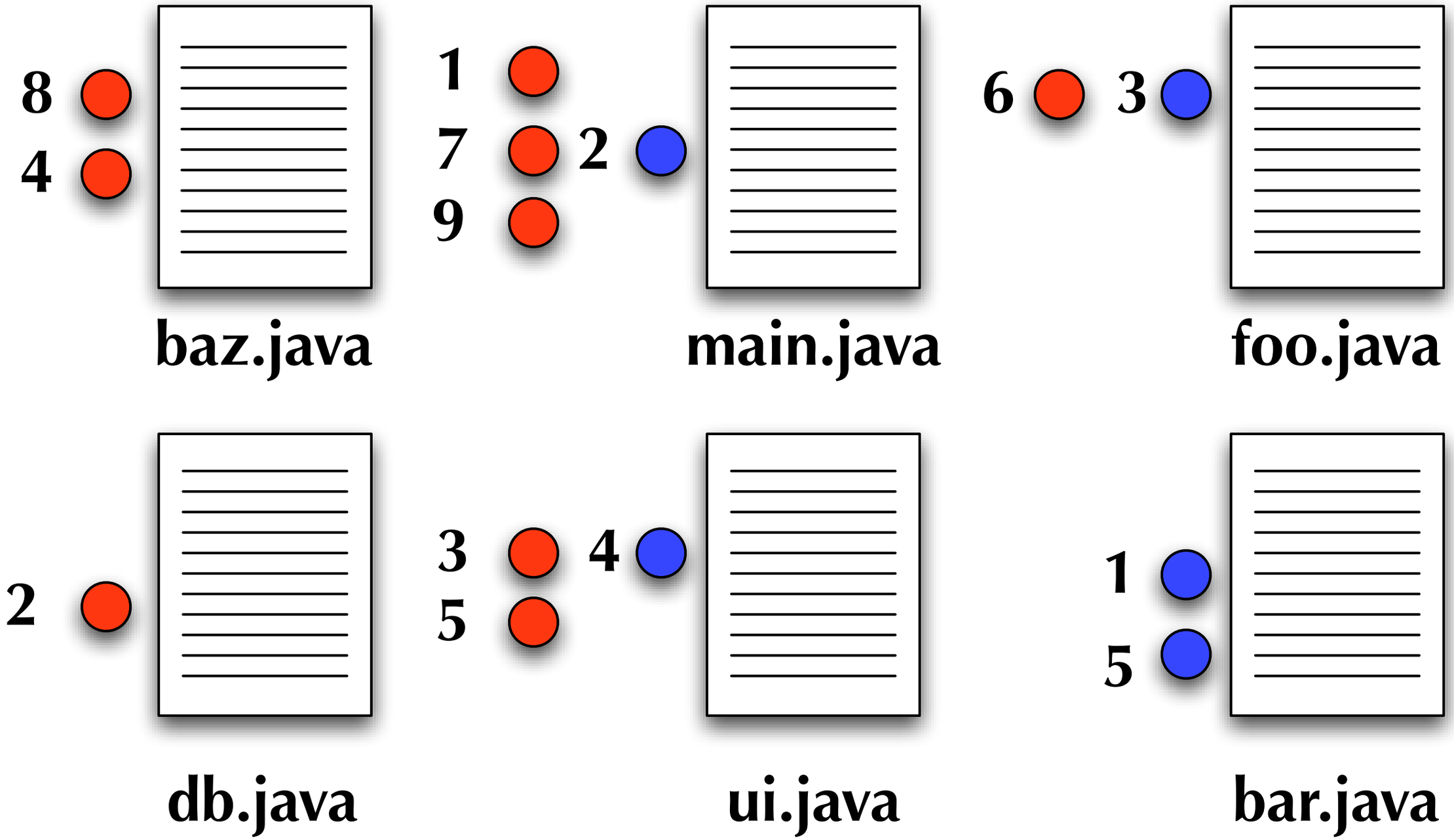
Terminology

- 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

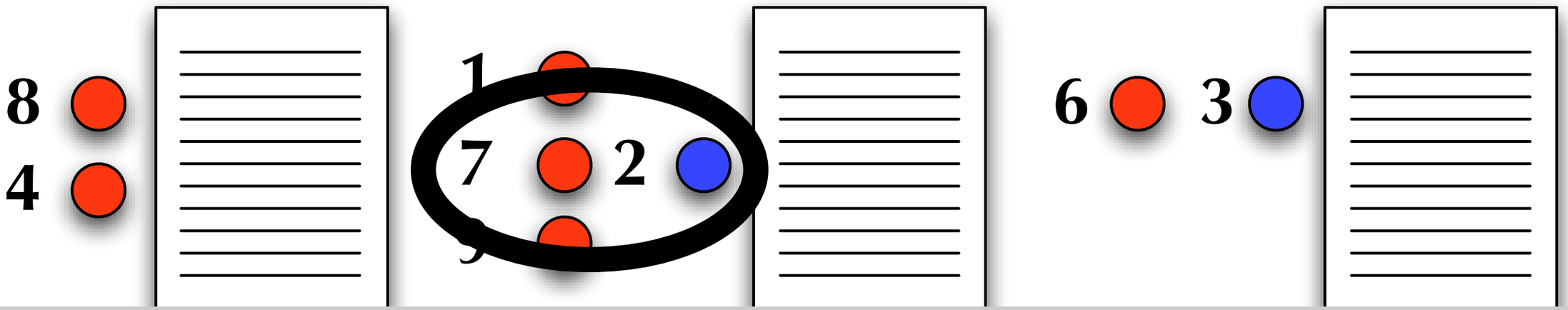
Another View: Sequential Program



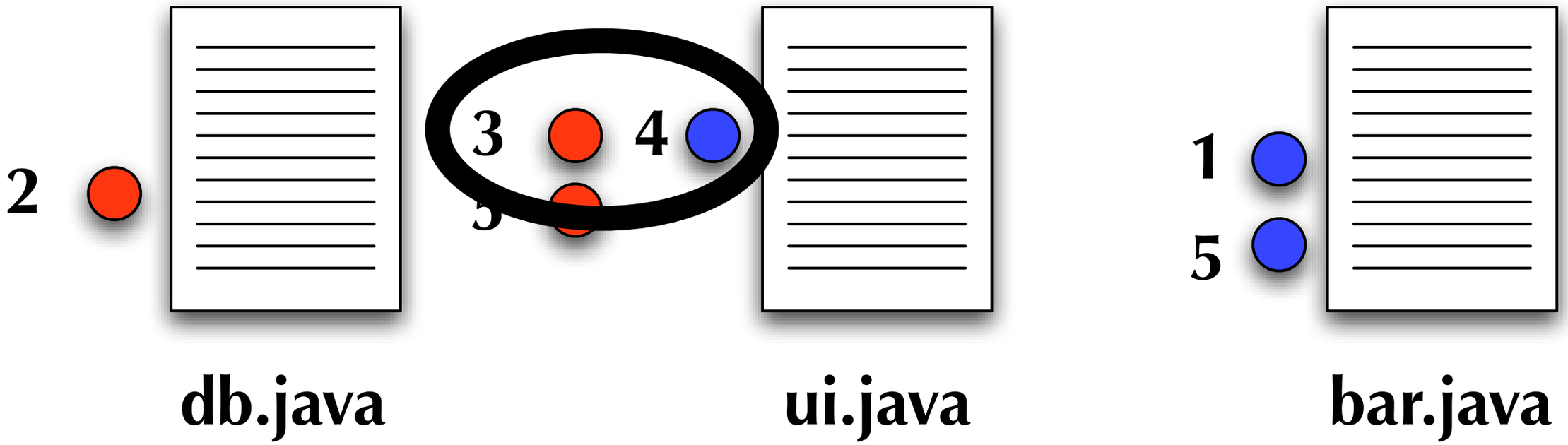
Another View: Concurrent Program



The problem with concurrency?



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



Output for Non Thread-Safe Singleton Code

- s9 = Singleton@45d068
- s8 = Singleton@45d068
- s3 = Singleton@45d068
- s6 = Singleton@45d068
- s1 = Singleton@45d068
- s0 = Singleton@ab50cd
- s5 = Singleton@45d068
- s4 = Singleton@45d068
- s7 = Singleton@45d068
- s2 = Singleton@45d068

Remember this slide from Lecture 22? (Program on next slide)

Thread 0 created an instance of the Singleton class at memory location ab50cd at the same time that another thread (we don't know which one) created an additional instance of Singleton at memory location 45d068!

Program to Test Thread Safety

```
1  public class Creator implements Runnable {
2
3      private int id;
4
5      public Creator(int id) {
6          this.id = id;
7      }
8
9      public void run() {
10         try {
11             Thread.sleep(200L);
12         } catch (Exception e) {
13         }
14         Singleton s = Singleton.getInstance();
15         System.out.println("s" + id + " = " + s);
16     }
17
18     public static void main(String[] args) {
19         Thread[] creators = new Thread[10];
20         for (int i = 0; i < 10; i++) {
21             creators[i] = new Thread(new Creator(i));
22         }
23         for (int i = 0; i < 10; i++) {
24             creators[i].start();
25         }
26     }
27
28 }
29
```

Creates a “runnable” object that can be assigned to a thread.

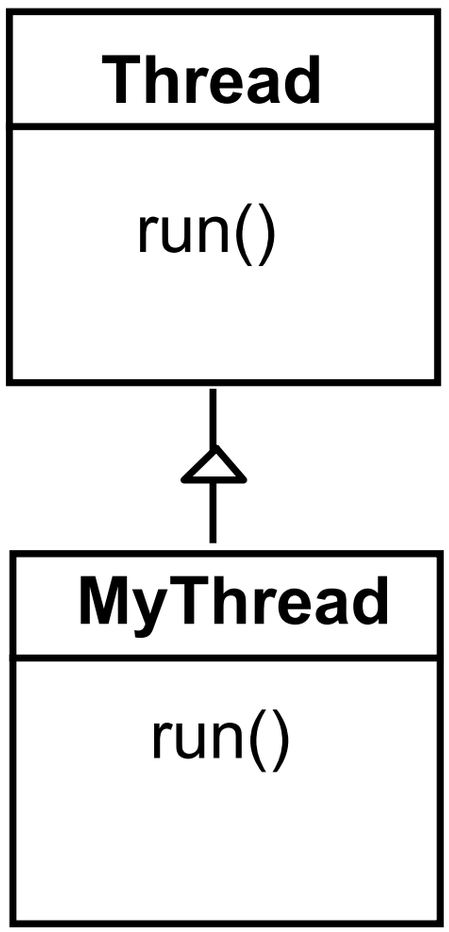
When its run, its sleeps for a short time, gets an instance of the Singleton, and prints out its object id.

The main routine, creates ten runnable objects, assigns them to ten threads and starts each of the threads

Since we didn’t protect Singleton.getInstance(), this program is not safe.

threads in Java

A Thread class manages a single sequential thread of control. Threads may be created and deleted dynamically.



The Thread class executes instructions from its method run(). The actual code executed depends on the implementation provided for run() in a derived class.

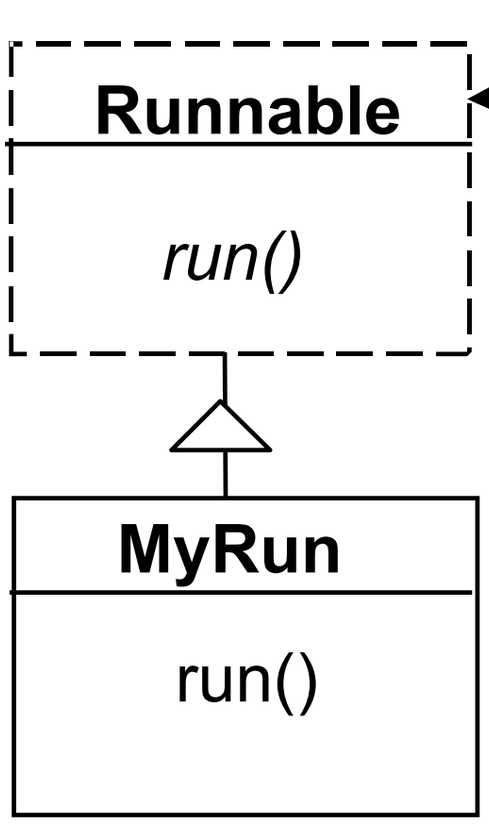
```
class MyThread extends Thread {
      public void run() {
            //......
      }
}
```

Creating a thread object:

```
Thread a = new MyThread();
```

threads in Java

Since Java does not permit multiple inheritance, we often implement the **run()** method in a class not derived from Thread but from the interface Runnable.



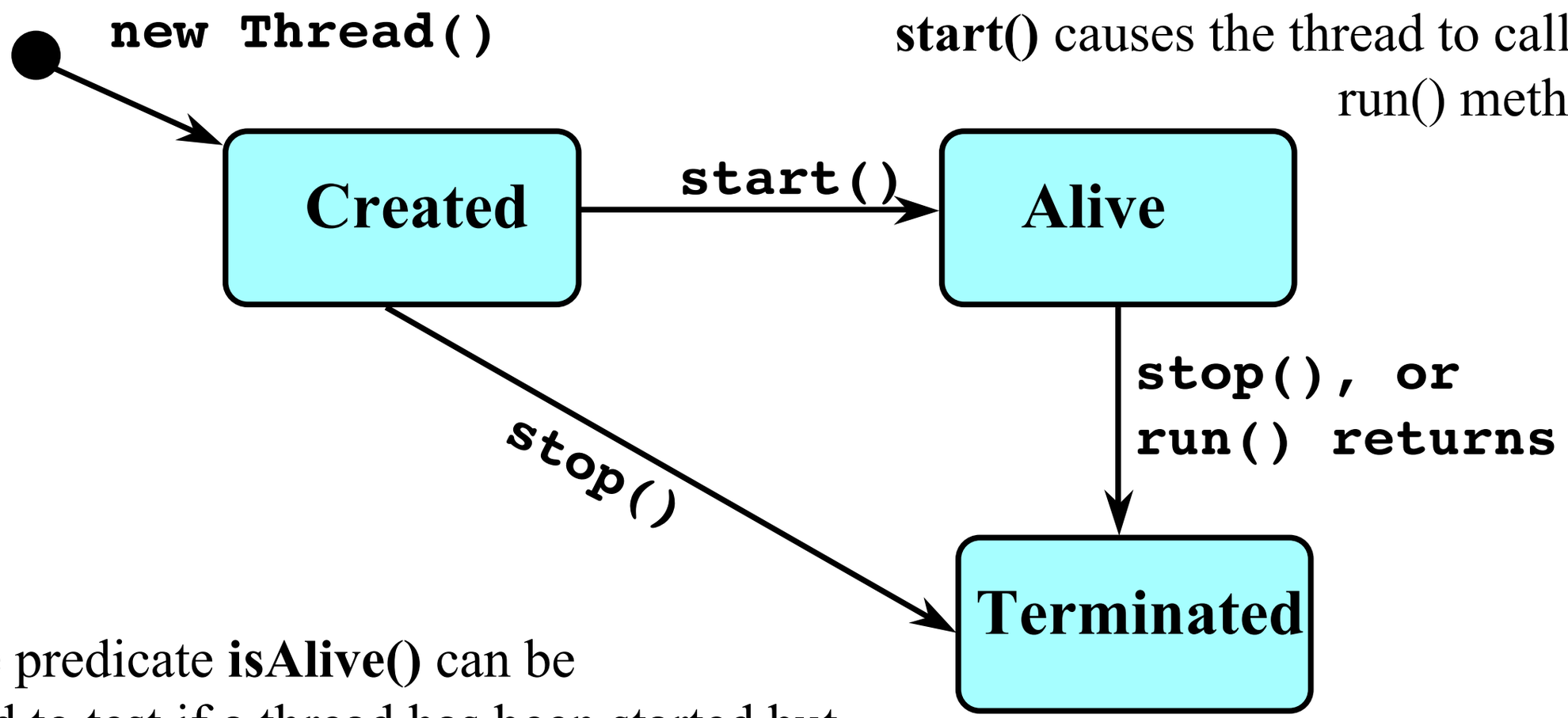

```
public interface Runnable {
    public abstract void run();
}

class MyRun implements Runnable{
    public void run() {
        //.....
    }
}
```

Creating a thread object:

```
Thread b = new Thread(new MyRun());
```

thread life-cycle in Java

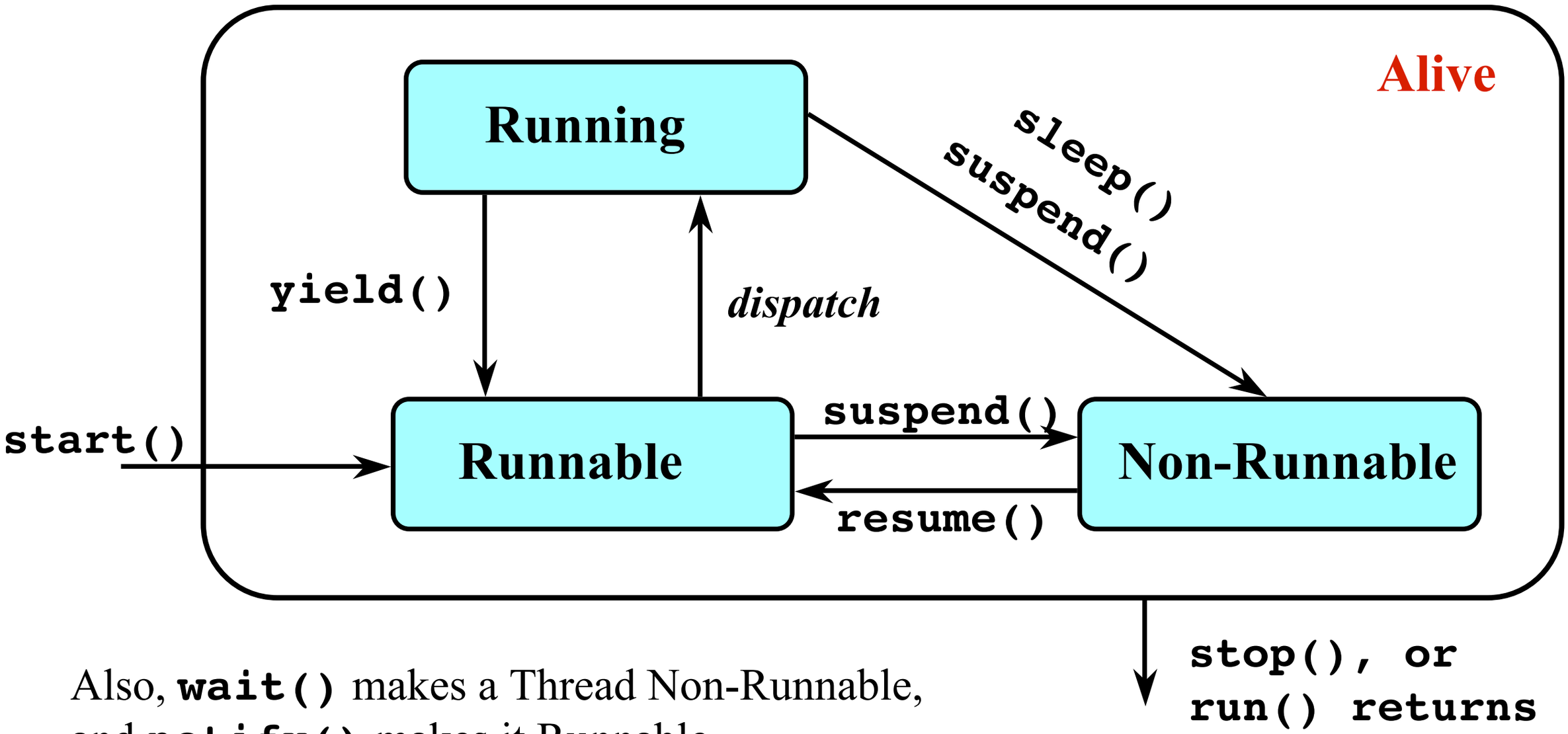
An overview of the life-cycle of a thread as state transitions:



The predicate **isAlive()** can be used to test if a thread has been started but not terminated. Once terminated, it cannot be restarted (cf. mortals).

thread alive states in Java

Once started, an **alive** thread has a number of substates :

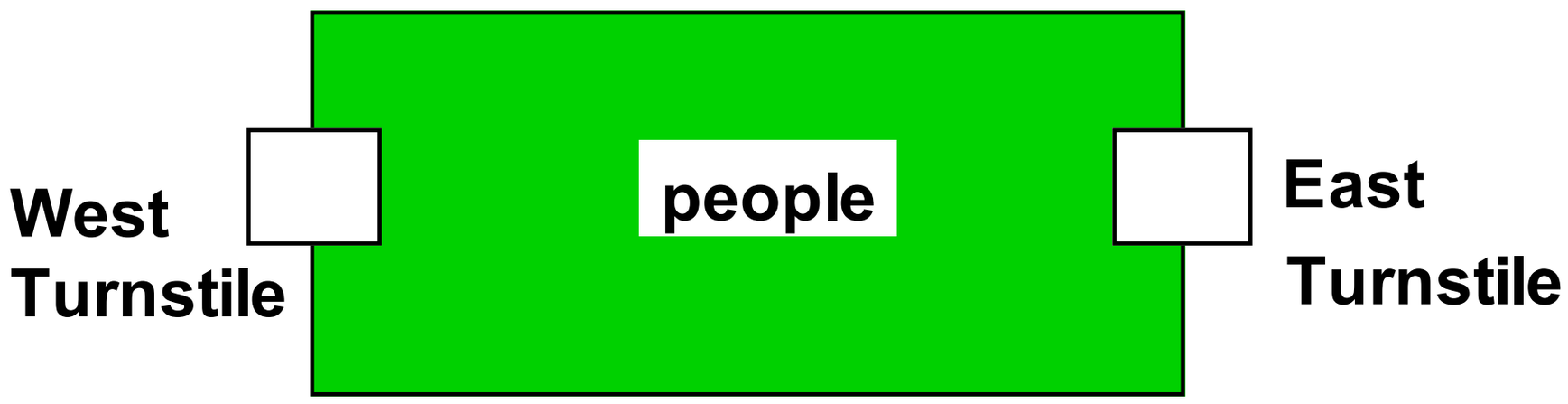


Also, **`wait()`** makes a Thread Non-Runnable,
and **`notify()`** makes it Runnable
(used in later chapters).

4.1 Interference

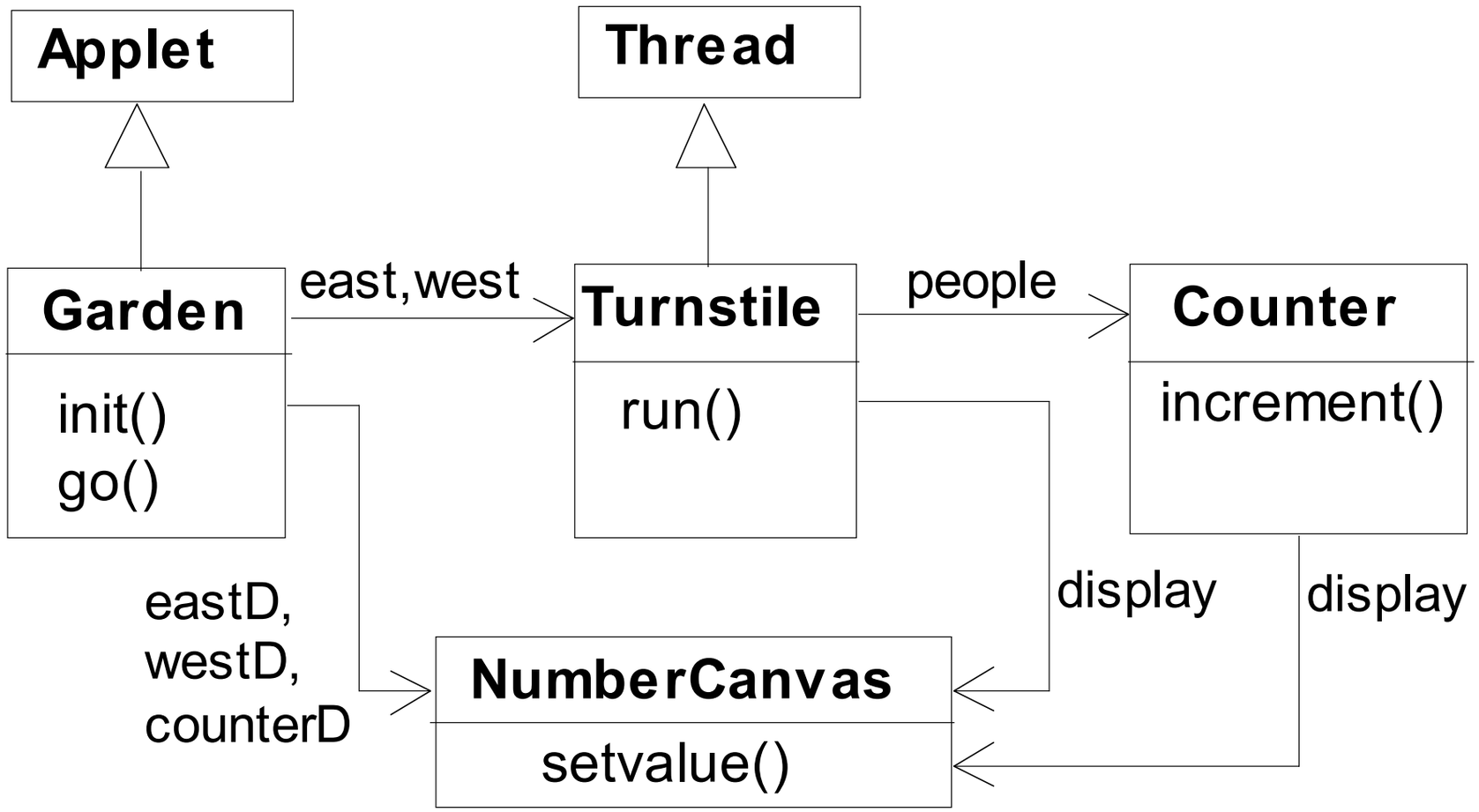
Ornamental garden problem:

People enter an ornamental garden through either of two turnstiles. Management wants to know how many people are in the garden at any time.



The concurrent program consists of two concurrent threads and a shared counter object.

ornamental garden Program - class diagram



The **`Turnstile`** thread simulates the periodic arrival of a visitor to the garden every second by sleeping for a second and then invoking the **`increment()`** method of the counter object.

ornamental garden program

The **`Counter`** object and **`Turnstile`** threads are created by the **`go()`** method of the Garden applet:

```
    private void go() {
      counter = new Counter(counterD);
      west = new Turnstile(westD,counter);
      east = new Turnstile(eastD,counter);
      west.start();
      east.start();
    }
```

Note that **`counterD`**, **`westD`** and **`eastD`** are objects of **`NumberCanvas`** used in chapter 2.

Turnstile class

```
class Turnstile extends Thread {
  NumberCanvas display;
  Counter people;

  Turnstile(NumberCanvas n,Counter c)
    { display = n; people = c; }

  public void run() {
    try{
      display.setvalue(0);
      for (int i=1;i<=Garden.MAX;i++){
        Thread.sleep(500); //0.5 second between arrivals
        display.setvalue(i);
        people.increment();
      }
    } catch (InterruptedException e) {}
  }
}
```

The **run()** method exits and the thread terminates after **Garden.MAX** visitors have entered.

Counter class

```
class Counter {
  int value=0;
  NumberCanvas display;

  Counter(NumberCanvas n) {
    display=n;
    display.setvalue(value);
  }

  void increment() {
    int temp = value;     //read value
    Simulate.HWinterrupt();
    value=temp+1;         //write value
    display.setvalue(value);
  }
}
```

Hardware interrupts can occur at **arbitrary** times.

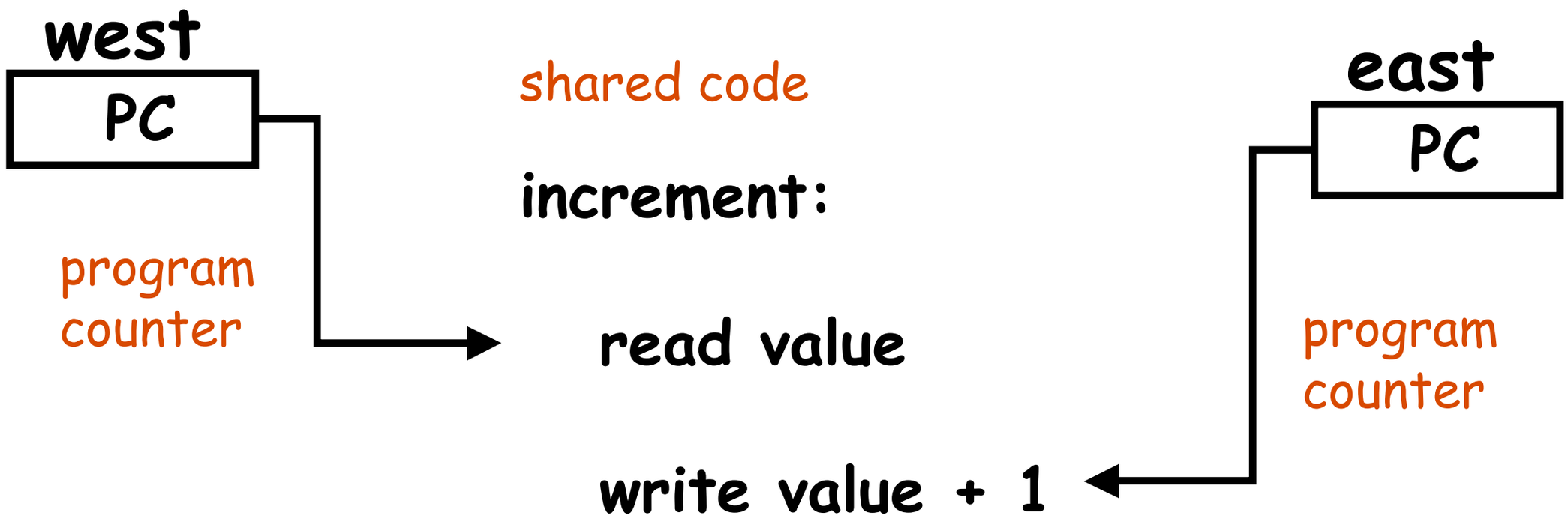
The **`counter`** simulates a hardware interrupt during an **`increment()`**, between reading and writing to the shared counter **`value`**. Interrupt randomly calls **`Thread.sleep()`** to force a thread switch.

ornamental garden program - display

After the East and West turnstile threads have each incremented its counter 20 times, the garden people counter is not the sum of the counts displayed. Counter increments have been lost. ***Why?***

concurrent method activation

Java method activations are not atomic - thread objects **east** and **west** may be executing the code for the increment method at the same time.



Interference and Mutual Exclusion

Destructive update, caused by the arbitrary interleaving of read and write actions, is termed ***interference***.

Interference bugs are extremely difficult to locate. The general solution is to give methods *mutually exclusive* access to shared objects. Mutual exclusion can be modeled as atomic actions.

4.2 Mutual exclusion in Java

Concurrent activations of a method in Java can be made mutually exclusive by prefixing the method with the keyword **`synchronized`**, which uses a lock on the object.

We correct **`COUNTER`** class by deriving a class from it and making the increment method **synchronized**:

```
class SynchronizedCounter extends Counter {

  SynchronizedCounter(NumberCanvas n)
     {super(n);}

   synchronized void increment() {
        super.increment();
   }
}
```

acquire lock

release lock

mutual exclusion - the ornamental garden

Java associates a ***lock*** with every object. The Java compiler inserts code to acquire the lock before executing the body of the synchronized method and code to release the lock before the method returns. Concurrent threads are blocked until the lock is released.

Java synchronized statement

Access to an object may also be made mutually exclusive by using the **`synchronized`** statement:

```
synchronized (object) { statements }
```

A less elegant way to correct the example would be to modify the **`Turnstile.run()`** method:

```
synchronized(people) {people.increment();}
```

Why is this "less elegant"?

To ensure mutually exclusive access to an object, **all object methods** should be synchronized.

If synchronized is so great, what's the downside?

Performance

Lots of synchronized methods can significantly impact a program's performance, as time is spent acquiring and releasing locks; if there is no thread contention that time is simply wasted.

You only want to incur this penalty when it is likely that that multiple threads will stomp on shared data.

Lots of work on making this easier

- Intense focus on multi-core programming in research settings and industry
- Apple's Grand Central: <http://www.apple.com/macosx/snowleopard/>
- "New" languages being developed
  - Erlang for instance is a functional programming language developed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications
    - To avoid the problems we've seen today, Erlang programs consist of processes that communicate by passing messages. There is no mutable state (functional programming mainstay), no shared memory, as a result no need for locks!
    - An Erlang program run on one processor can potentially become **n** times faster if run on **n** processors
      - as long as it has lots of (Erlang) processes, that don't interact much and there are no significant sequential bottlenecks
- Other languages: Go, Scala, Clojure (the latter two build on the JVM)

Alternative Approaches

- As a result of these concerns, computer scientists have searched for other ways to exploit concurrency
    - in particular using techniques from functional programming
- Functional programming is an approach to programming language design in which functions are
    - first class values (with the same status as int or string)
        - you can pass functions as arguments, return them from functions and store them in variables
    - and have no side effects
        - they take input and produce output
        - this typically means that they operate on immutable values

Example (I)

- In python, strings are immutable
  - a = “Ken @@@”
  - b = a.replace(“@”, “!”)
  - b
    - 'Ken !!!'
  - a
    - 'Ken @@@'
- Replace is a function that takes an immutable value and produces a new immutable value with the desired transformation ; it has no side effects

Example (II)

- Functions as values (in python)
  - def Foo(x, y):
    - return x + y
  - add = Foo
    - add(2, 2)
      - 4
- Here, we defined a function, stored it in a variable, and then used the “call syntax” with that variable to invoke the function that it pointed at

Example (III)

- continuing from previous example
  - def DoIt(fun, x, y): return fun(x,y)
  - DoIt(add, 2, 2)
    - 4

- Here, we defined a function that accepts three values, some other function and two arguments
  - We then invoked that function by passing our add function along with two arguments ;
  - DoIt() is an example of higher-order functions: functions that take functions as parameters
  - Higher-order functions is a common idiom in func. prog.

Relationship to Concurrency?

- How does this relate to concurrency?
    - It offers a new model for designing concurrent systems
        - Each thread operates on immutable data structures using functions with no side effects
        - A thread's data structures are not shared with other threads
        - Work is performed by passing messages between threads
            - If one thread requires data from another that data is copied and then sent
- Such an approach allows each thread to act like a single-threaded program; no danger of interference

Map, Filter, Reduce

- Three common higher order functions are map, filter, reduce
- map(fun, list) -> list
    - Applies fun() to each element of list; returns results in new list
- filter(fun, list) -> list
    - Applies boolean fun() to each element of list; returns new list containing those members of list for which fun() returns True
- reduce(fun, list) -> value
    - Returns a value by applying fun() to successive members of list (total = fun (list[0], list[1]); total = fun(total, list[2]); …)

Examples

- list = [10, 20, 30, 40, 50]

- def double(x): return 2 * x
- def limit(x): return x > 30
- def add(x,y): return x + y

- map(double, list) returns [20, 40, 60, 80, 100]
- filter(limit, list) returns [40, 50]
- reduce(add, list) returns 150

Implications

- map is very powerful
    - especially when you consider that you can pass a list of functions to it and then pass a higher-order function as the function to be applied
        - for example
            - def DoIt(x): return x()
            - map(DoIt, [f(), g(), h(), i(), j(), k()])
- But the real power, with respect to concurrency is that map is simply an abstraction that can, in turn, be implemented in a number of ways

Single Threaded Map

- We could for instance implement map() like this:
    - def map(fun, list):
        - results = []
        - for item in list:
            - results.append(fun(item))
- This would implement map in a single threaded fashion

Multi-threaded Map

- We could also implement map like this (pseudocode):
  - def Mapper(Thread):
    - def __init__(… fun, list): …
    - def run():
      - self.results = map(fun, list)
  - def xmap(fun, list):
    - split list into N parts where N = number of cores
    - create N instances of Mapper(fn, list_i)
    - wait for each thread to end (in order) and grab results

Note: threads can complete in any order since each computation is independent

Super Powerful Map

- We could also implement map like this:
  - def supermap(fun, list):
    - divide list into N parts where N equals # of machines
    - send list_i to machine i which then invokes xmap
    - wait for results from each machine
    - combine into single list and return
- Given this implementation, you can apply a very complicated function to a very large list and have (potentially) thousands of machines leap into action to compute the answer

Google

- Indeed, this is what Google does when you submit a search query:
  - def aboveThreshold(x): return x > 0.5 <-- just making this up
  - def probabilityDocumentRelatedToSearchTerm(doc): …

- searchResults =
  - filter(aboveThreshold,
    - map(probabilityDocumentRelatedToSearchTerm,
      - [<entire contents of the Internet]))

Difference between map and xmap?

- The team behind Erlang published results concerning the difference between map and xmap
    - They make a distinction between
        - CPU-bound computations with little message passing vs.
        - lightweight computations with lots of message passing
- With the former, xmap provides linear speed-up (10 CPUs provides a 10x speed-up, then declining) over map
    - the latter less so (10 CPUs provided 4x speed-up)
    - Indeed, xmap’s performance in the latter case tends to max out at 4x no matter how many CPUs were added

Linear speed-up: Hard to achieve!

- On my machine a program to double each member of a large list actually runs faster in single threaded mode!!
  - When using map, you are building just one results list and do not incur any overhead with respect to threading
  - When using xmap, three lists are being created (one per thread, one to collect the results) and
  - you incur overhead to
    - create each thread
    - wait for each one to start running
    - wait for each one to join the main thread

Demo

Agent Model

- The functional language Erlang is credited with creating an approach to concurrency known as the agent model
  - A concurrent program consists of a set of agents
    - Each agent has its own set of data structures that are not shared with other agents
- Agents can perform computations and send messages
- Messages sit in an actor’s mailbox until it is ready to process them; they are always processed one at a time
  - An actor does not block when sending a message
  - An actor is not interrupted when a message arrives

Examples

- Examples will be presented in Scala
  - Scala is a language which nicely combines both the imperative and functional programming styles
  - It is implemented on top of Java and thus is cross platform
  - I won’t spend much time explaining Scala; I’ll just focus on the agent model

Example 1

- import scala.actors._
- object SillyActor extends Actor {
  - def act() {
    - for (i <- 1 to 5) {
      - println(“I’m acting!”)
      - Thread.sleep(1000)
      - }
    - }
  - }
- }

- object SeriousActor extends Actor {
  - def act() {
    - for (i <- 1 to 5) {
      - println(“To be or not to be”)
      - Thread.sleep(1000)
    - }
  - }
- }

Running Example 1

- SillyActor.start() ; SeriousActor.start()
- Demo
  - From this example we can see that Actor is a class that can be sub-classed (just like Thread in Java)
  - You start an actor by calling start()
  - At some point, the scheduler calls the actor’s act() method
    - The actor will be active until that method returns
    - This is just like Thread’s run() method, only the name has changed

Processing Messages

- To process a message, an actor must call either receive or react
  - react is a special case of receive that we'll discuss below
- You can think of receive as a "switch" statement that specifies the structure of the different type of messages it wants to receive
- When an actor calls receive, it looks at the mailbox and attempts to find a waiting message that matches one of the branches of the "switch" statement
  - it processes the first match that it finds

Example

- val echoActor = actor {
  - while (true) {
    - receive {
      - case msg =>
        - println(“received message: “ + msg)
    - }
  - }
- }
- This actor loops forever and prints out any message it receives

A message is sent with the ! operator:

echoActor ! “hi there”
echoActor ! 25

Demo

Conserve Threads

- When an act() method calls receive(), it tells the scala run-time system that this actor needs its own thread
    - The actor may be spending its time switching between processing messages and performing a long computation
- Since threads in Java are not cheap, scala provides the react keyword to tell the runtime that all this thread does is react to messages
    - This means it spends most of its time blocked
        - Scala uses this information to assign “react actors” to a single thread, thus conserving threads in the overall system

Example

- object NameResolver extends Actor {
  - ...
  - def act() {
    - react {
      - case (name: String, actor: Actor) =>
        - actor ! getIp(name)
        - act()
      - case “EXIT” =>
        - println(“quitting”)
- }
- ...

Note: no explicit loop; that’s because react doesn’t return (enables sharing of multiple actors on a single thread)

instead, react must call act() if it wants to keep waiting for messages

Results

- To test Scala's claim that react helps conserve threads
  - I wrote a program that can create a specified number of NameResolvers that either
    - use receive or
    - use react
- Results: when creating 100 NameResolvers
  - using receive: 104 threads created
  - using react: 7 threads created (!)

Returning to the Ornamental Garden

- With the Agent model of concurrency, you can easily avoid interference problems
    - Here’s an example of the ornamental garden problem
        - No need for mutual exclusion: create two agents that act as turnstiles and have them send increment messages to a shared counter agent
        -

Wrapping Up

- We have looked at a few alternative models to the “locks and shared data” model of concurrency that
  - draw on functional programming techniques
  - do not allow threads to share data
    - allow threads to communicate via asynchronous messages
- Deadlock and Race conditions are still possible in this model but harder to achieve
  - However, interference is simply not possible in this model
- Functional techniques seem like a promising method for tackling concurrency on multi-core hardware