O'REILLY®

Java Concurrency

About Me – Career

- New Relic, Principal Engineer
- jClarity, Co-founder
  - Sold to Microsoft
- Deutsche Bank
  - Chief Architect (Listed Derivatives)
- Morgan Stanley
  - Google IPO
- Sporting Bet
  - Chief Architect

About Me – Community

- Java Champion
- JavaOne Rock Star Speaker
- Java Community Process Executive Committee
- London Java Community
  - Organising Team
  - Co-founder, AdoptOpenJDK

Installing Java

- We will use AdoptOpenJDK - Java 8
  https://adoptopenjdk.net/
- We'll use "OpenJDK 8 (LTS)" and the HotSpot JVM (or Java 11)
- Download this repo from GitHub:
- **https://github.com/kittylyst/optimizing-java**
- The examples are in the repo
- My personal site (articles, links): **https://www.kittylyst.com**

Synchronization

- Simple Hardware Model
- Exercise: Effect of Hardware Caching
- The Need for Synchronization
- Exercise: Counters
- Java’s Original Thread & Synchronization API
- Synchronization Summary
- Q&A

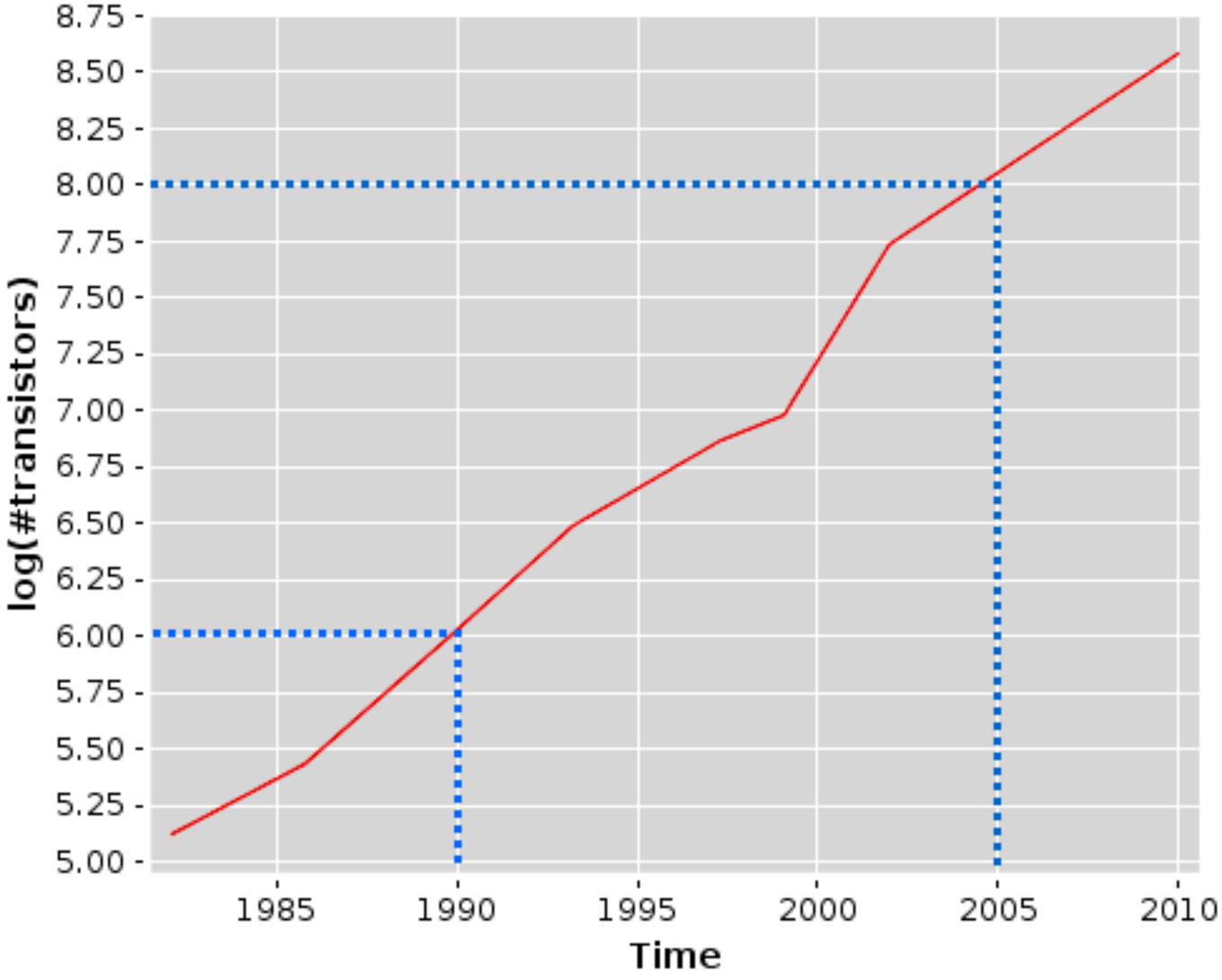
Simple Hardware Model

- Mechanical Sympathy
- Interesting Graphs and their Consequences
- The Modern CPU

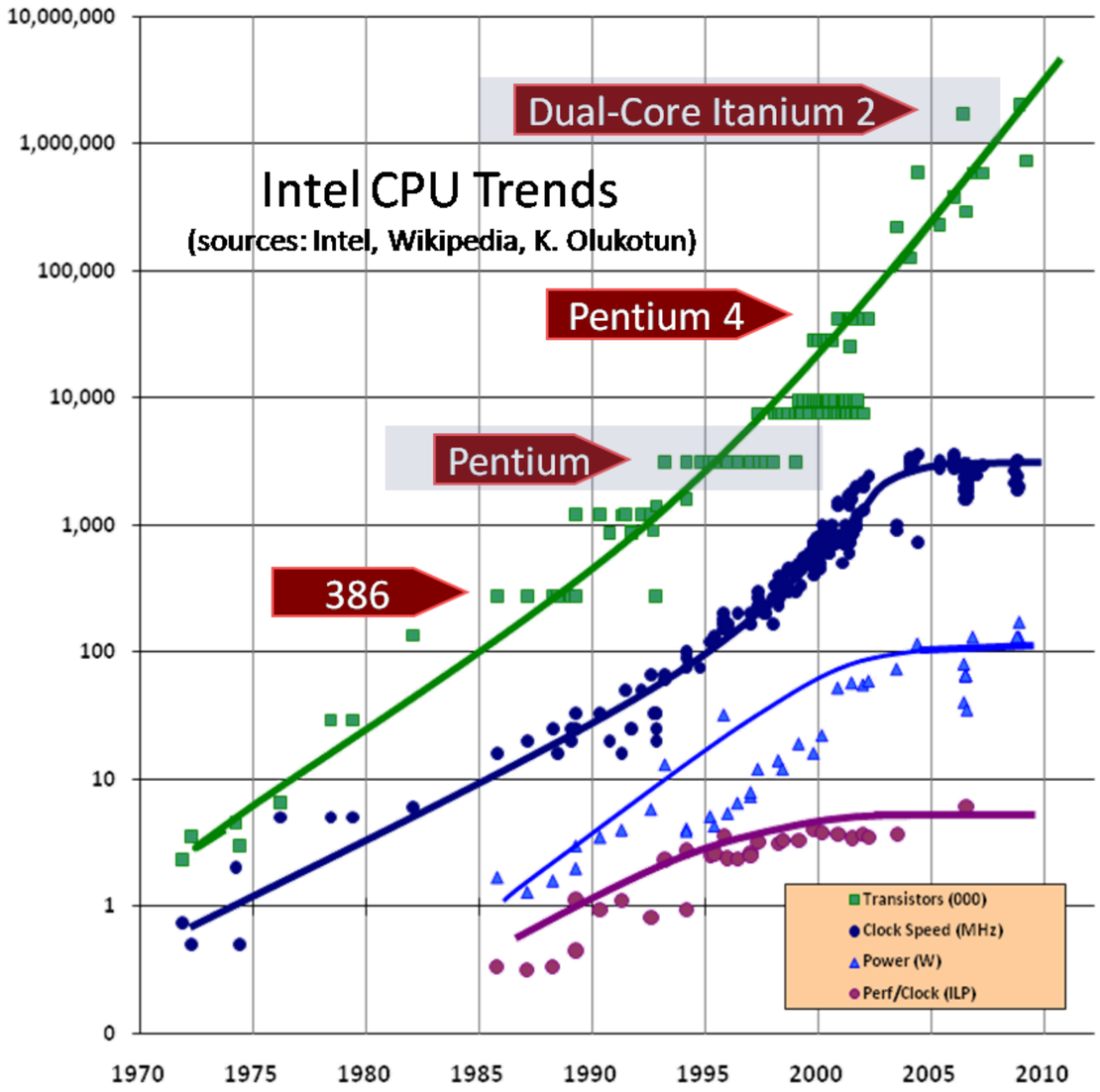
Mechanical Sympathy

- Named for a quote by Jackie Stewart (F1 driver)
- Understanding the underlying hardware helps
- Pioneered at LMAX by Martin Thompson
  - See also: Disruptor Pattern, Trisha Gee, Mike Barker
- Performance engineers need to understand hardware
  - At least, more than most developers do

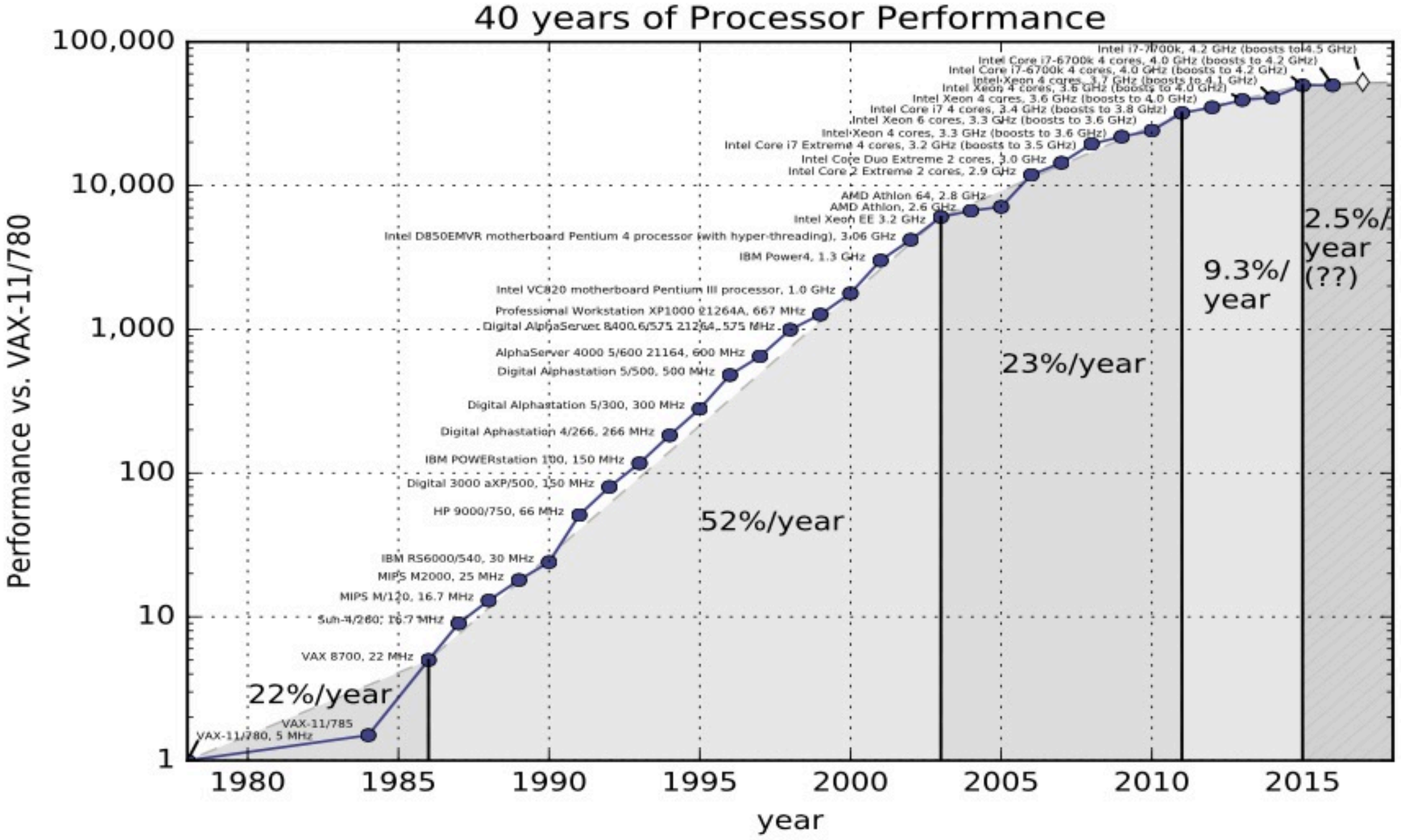
An Interesting Graph



A Scary Graph

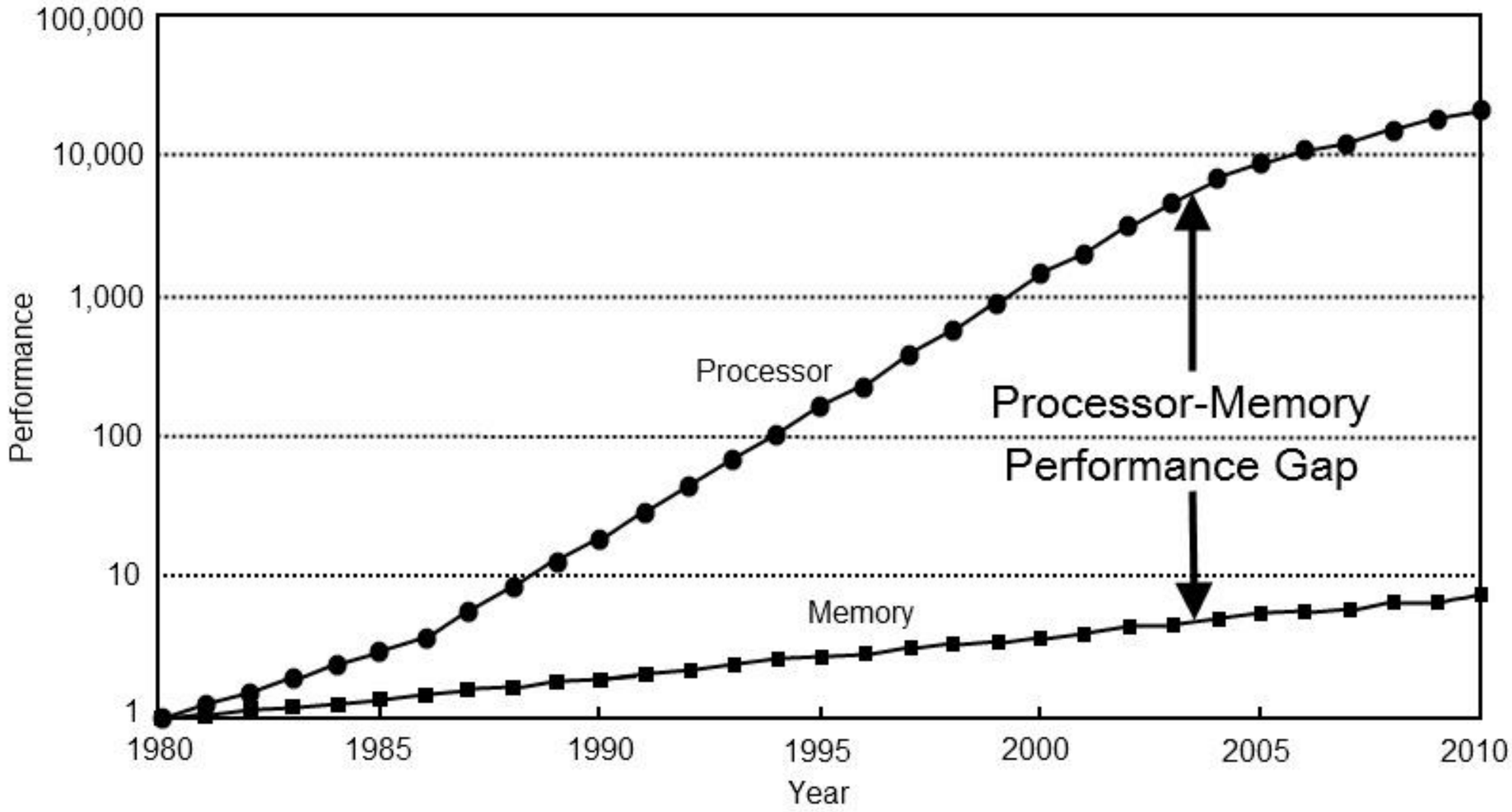


Another Interesting Graph



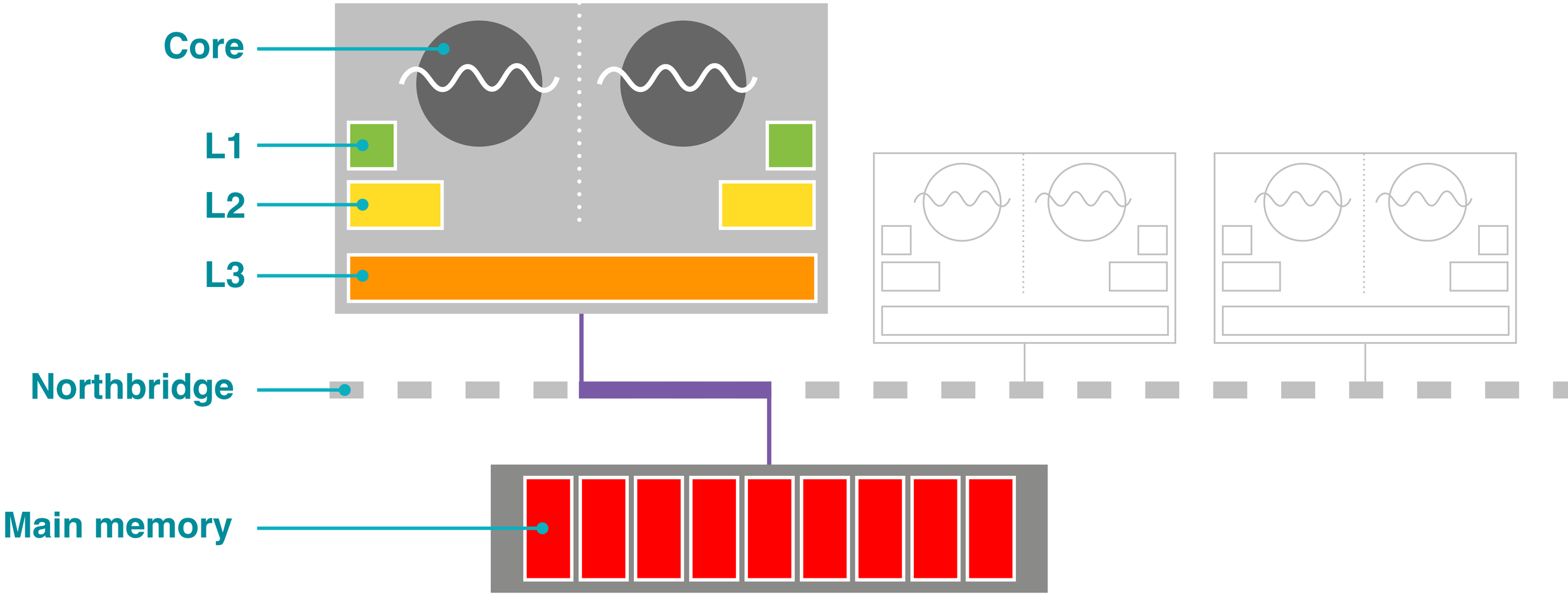
From: Computer Architecture: A Quantitative Approach by Hennessy, et al.

Another Interesting Graph

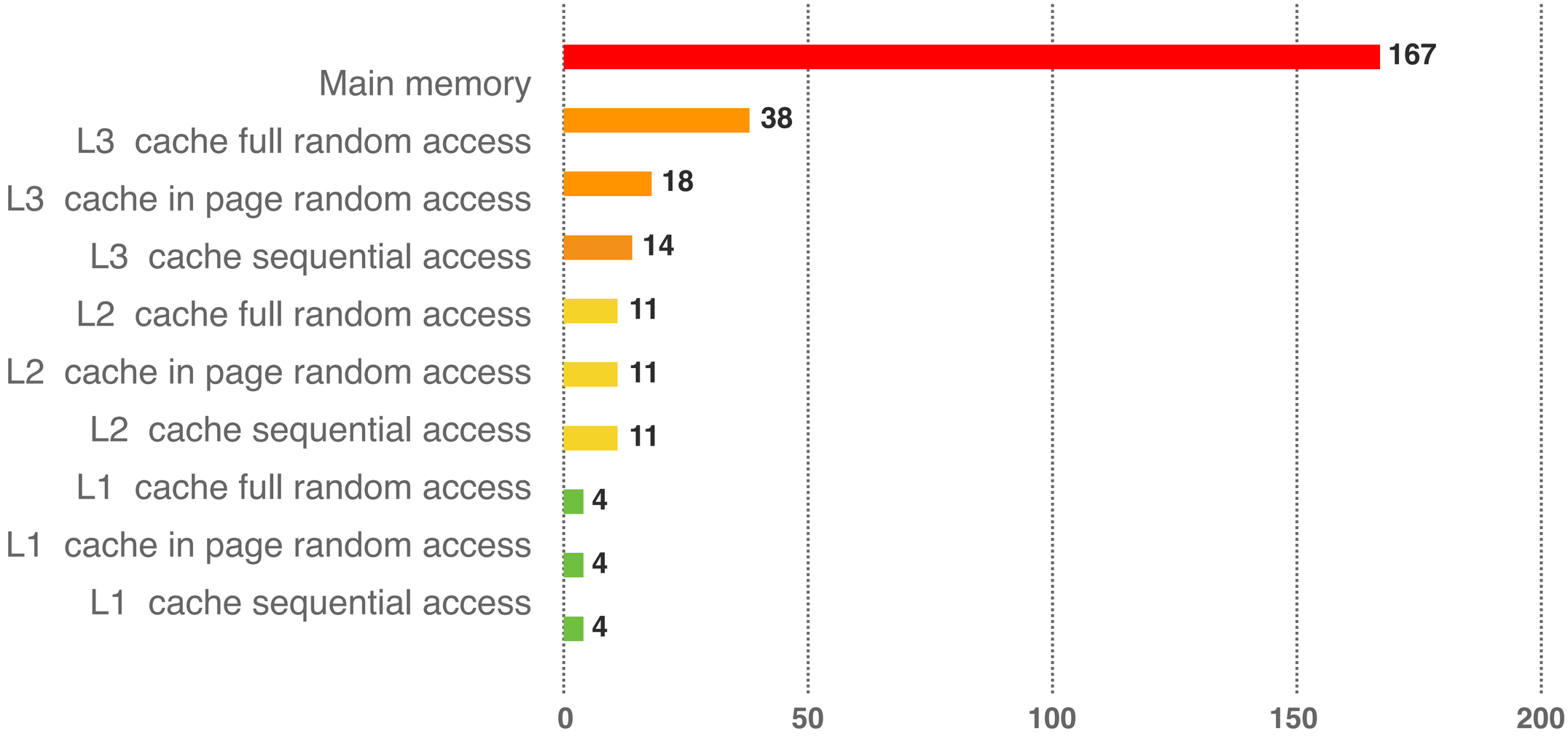


From: Computer Architecture: A Quantitative Approach by Hennessy, et al.

The CPU Landscape



The Speed of Memory



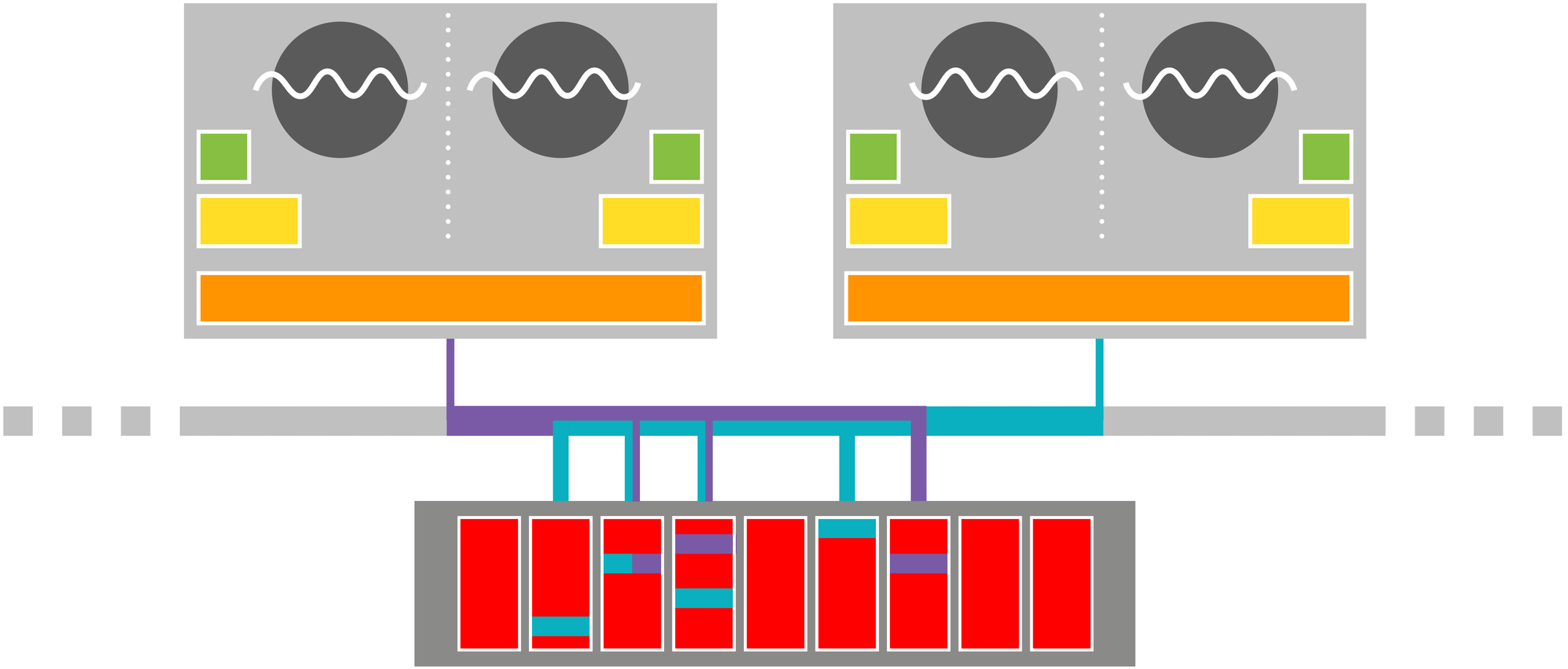
Exercise - Caching

EXERCISE

The Need for Synchronization

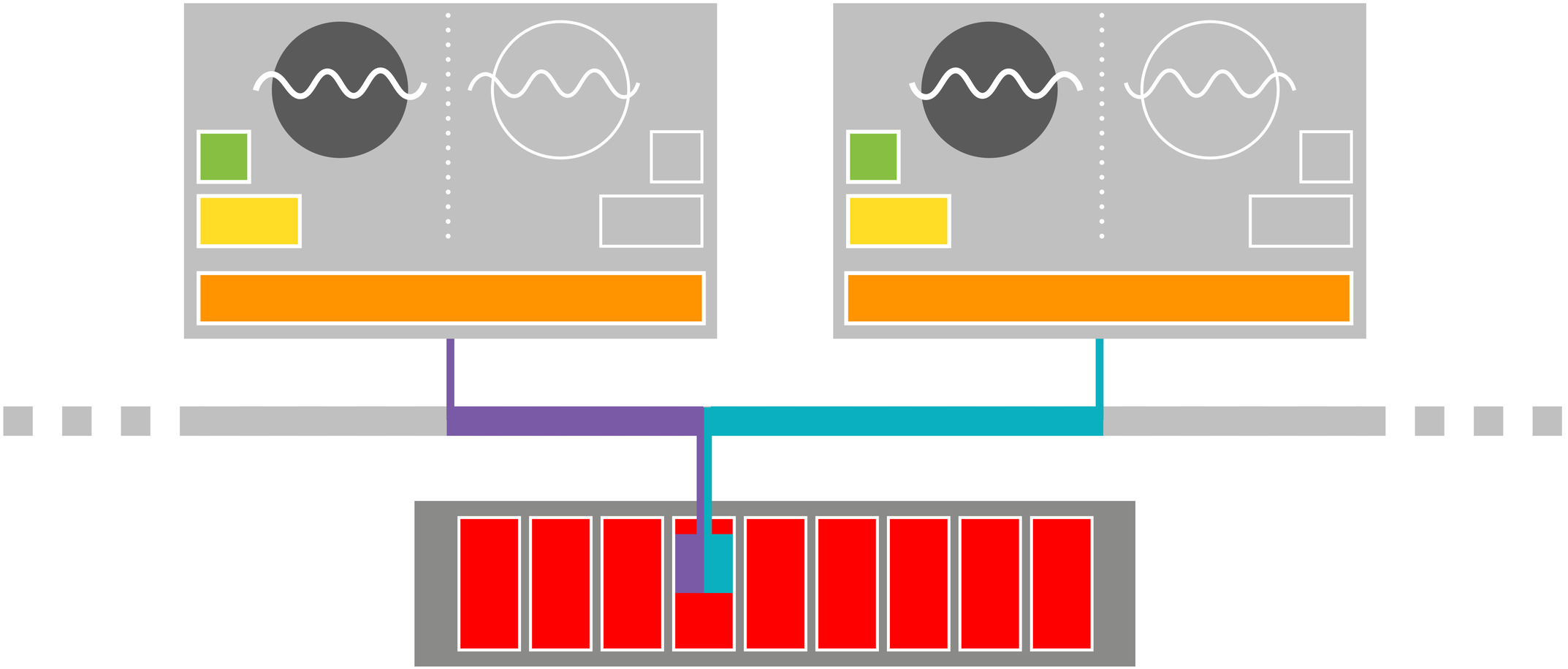
- Java's concurrency model
- Three fundamental concepts
- `synchronized` and `volatile`

Accessing the NorthBridge

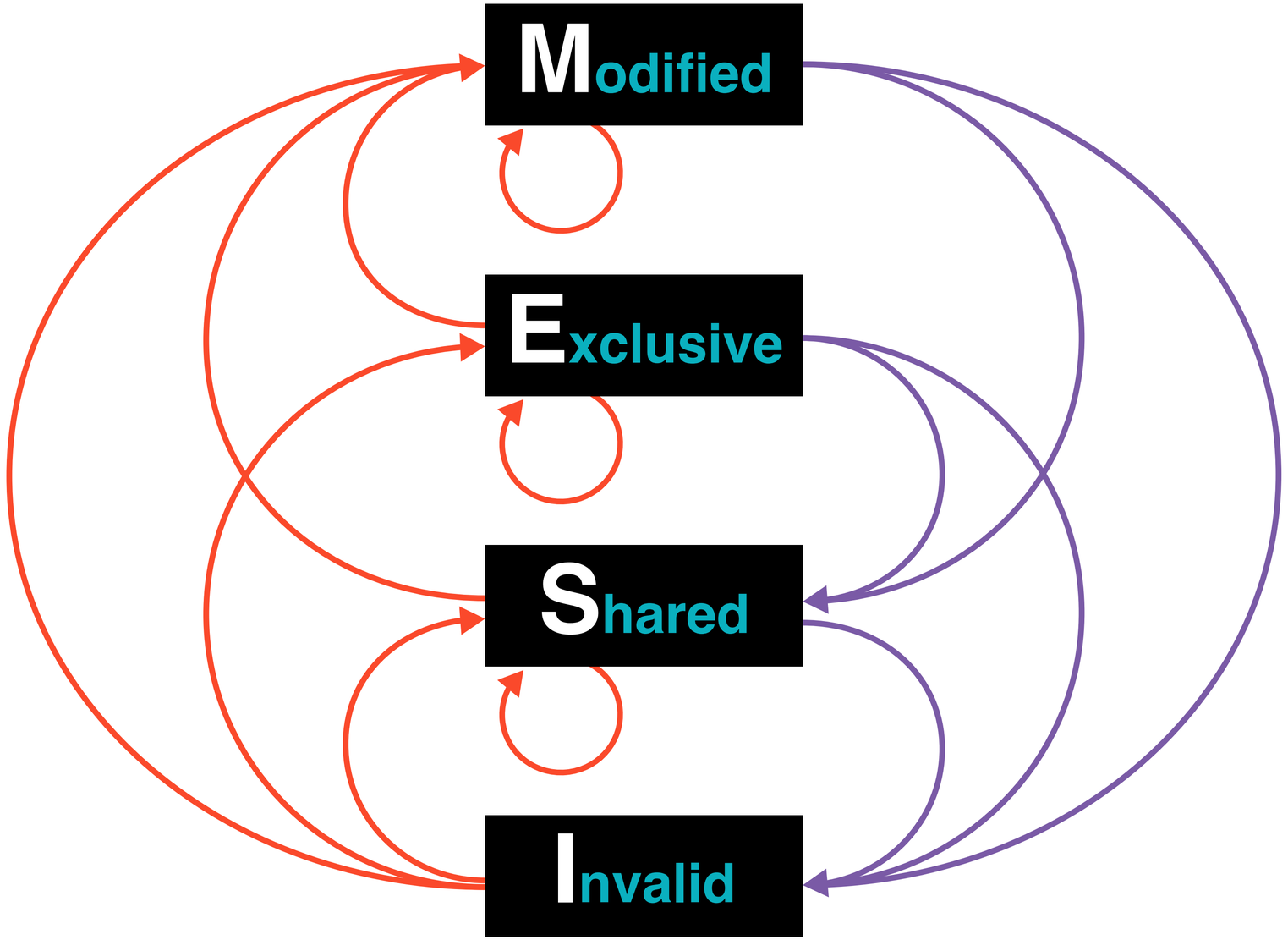


Cache Coherency

- If 2 cores access the same data, what is “true” view?



MESI



Fundamental Concepts of Java Concurrency

- Shared, visible-by-default mutable state
- Preemptive thread scheduling
- Objects must be locked to protect vulnerable state

Shared, Visible-by-Default Mutable State

- Objects are easily shared between all threads in a process
- Objects can be changed (“mutated”) by any thread
  - Provided the thread has a reference to the object

Preemptive Thread Scheduling

- What happens when a thread is swapped off a core?

Preemptive Thread Scheduling

- What happens when a thread is swapped off a core?
- Thread is suspended immediately
- It may be in an inconsistent state

Java's Concurrency model

- To protect against inconsistency, need to use exclusion
  - Only available mechanism
- All code using shared mutable state inconsistent must cooperate
  - Modifies **or reads**
- Code which does this is **Concurrently Safe**
- Remember: Scheduler can swap threads off cores at any time

Objects Must be Locked to Protect Vulnerable State

- Basis of “intrinsic concurrency” approach
- A Java thread is an OS thread
- **`Thread.start()`** calls to the OS to **`clone()`** a new thread
- A **`java.lang.Thread`** is just a metadata object
- OS monitors used to awake threads

Java's Concurrency Model

- What does `synchronized` do?

Java's Concurrency Model

- What does `synchronized` do?
- Why do we used the keyword `synchronized` for this?
- What is being synchronized?
- Why not `locked` or `critical` or similar?

Java's Concurrency Model

- What is being `synchronized`?
  - Main memory representation of object being locked

Java's Concurrency Model

- What is being **`synchronized`**?
  - Main memory representation of object being locked

- Two separate operations
  - Read from memory at start of synchronized block
  - Write-back to memory at end of block

Sequence of Events for Synchronized Block

- Thread needs to modify an object
  - Indicates it requires temporary exclusive access to the object

Sequence of Events for Synchronized Block

- Thread needs to modify an object
  - Indicates it requires temporary exclusive access to the object
- Thread acquires object monitor
  - May have had to block to acquire

Sequence of Events for Synchronized Block

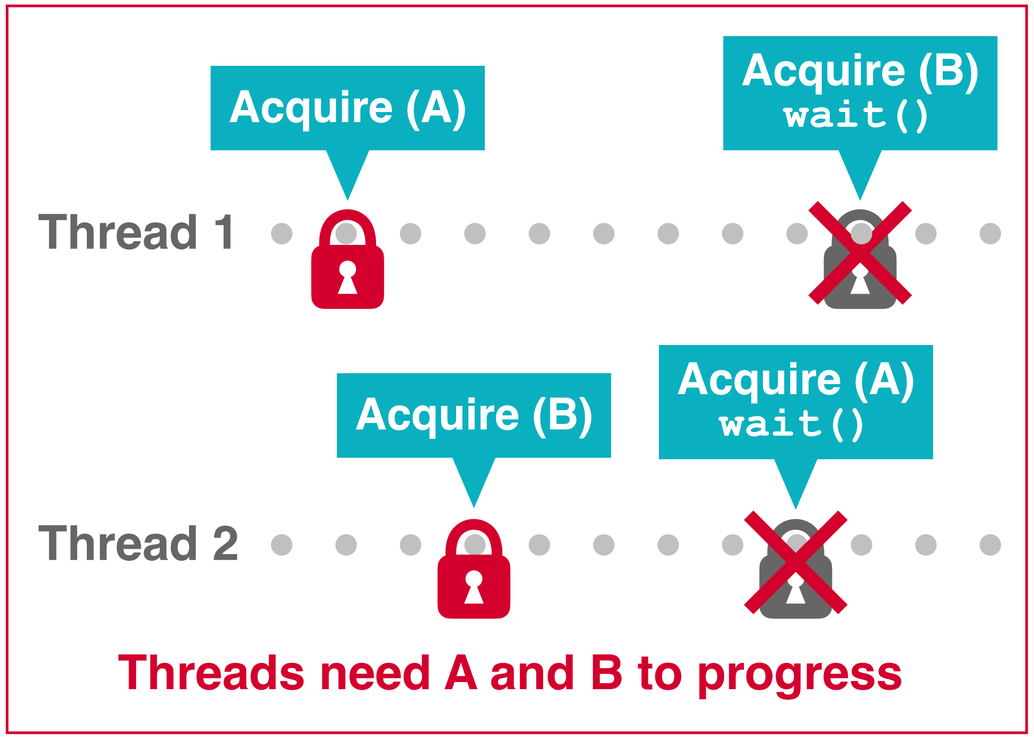
- Thread needs to modify an object
  - Indicates it requires temporary exclusive access to the object
- Thread acquires object monitor
  - May have had to block to acquire
- Thread modifies (or reads) the object
  - While locked, object may become briefly inconsistent
  - Leaves it in a consistent, legal state when done

Sequence of Events for Synchronized Block

- Thread needs to modify an object
  - Indicates it requires temporary exclusive access to the object
- Thread acquires object monitor
  - May have had to block to acquire
- Thread modifies (or reads) the object
  - While locked, object may become briefly inconsistent
  - Leaves it in a consistent, legal state when done
- Thread releases the monitor
  - Other threads can now acquire

Deadlocks

- When several threads are competing over locks
  - A Liveness hazard




```
Object a = new Object();
Object b = new Object();

Thread t1 = new Thread(new Runnable() {
   public void run() {
      synchronized(a) {
         try {
             Thread.sleep(100);
         } catch (InterruptedException e) { }
         synchronized(b) { compute(); }
      }
   }});

Thread t2 = new Thread(new Runnable() {
   public void run() {
      synchronized(b) {
         synchronized(a) { compute(); }
      }
   }});

t1.start();
t2.start();
```

Avoiding Deadlocks

- Fully-Synchronized Objects are immune
- Immutable Objects are immune
- For other object types
  - Always acquire locks in same order in all threads
  - Use the Lock construct in j.u.c
  - Use strategies with a timeout / backoff
- Be careful of partial solutions
  - Extra complexity can cause schedulers to make different choices
  - Some “fixes” are really this in disguise

`volatile`

- What does `volatile` do?

`volatile`

- What does `volatile` do?
  - Provides **1** operation to or from main memory
  - No locking is involved
  - Caches are flushed

`volatile`

- What does `volatile` do?
  - Provides **1** operation to or from main memory
  - No locking is involved
  - Caches are flushed

- The value is:
  - Re-read from main memory before use
  - **Or** written back to main memory

`volatile`

- Volatile variables cannot be used for state-dependent updates
  - e.g. can't do **`v++`** on a volatile int
  - For these, must use a synchronized block
  - Example of **`Counter`** – works the same way for volatile as non-volatile
- Volatiles cannot cause deadlocks
  - There are no locks

Exercise - Counters

EXERCISE

Fun with Counters

```
public final class UnprotectedCounter
    implements Counter {

        private int i = 0;

        public int increment() {
            return i = i + 1;
        }

        public int get() {
            return i;
        }
}
```

```
public int increment();
Code:
   0: aload_0
   1: aload_0
   2: getfield #2 // Field i:I
   5: iconst_1
   6: iadd
   7: dup_x1
   8: putfield #2 // Field i:I
  11: ireturn
```

Fun with Counters

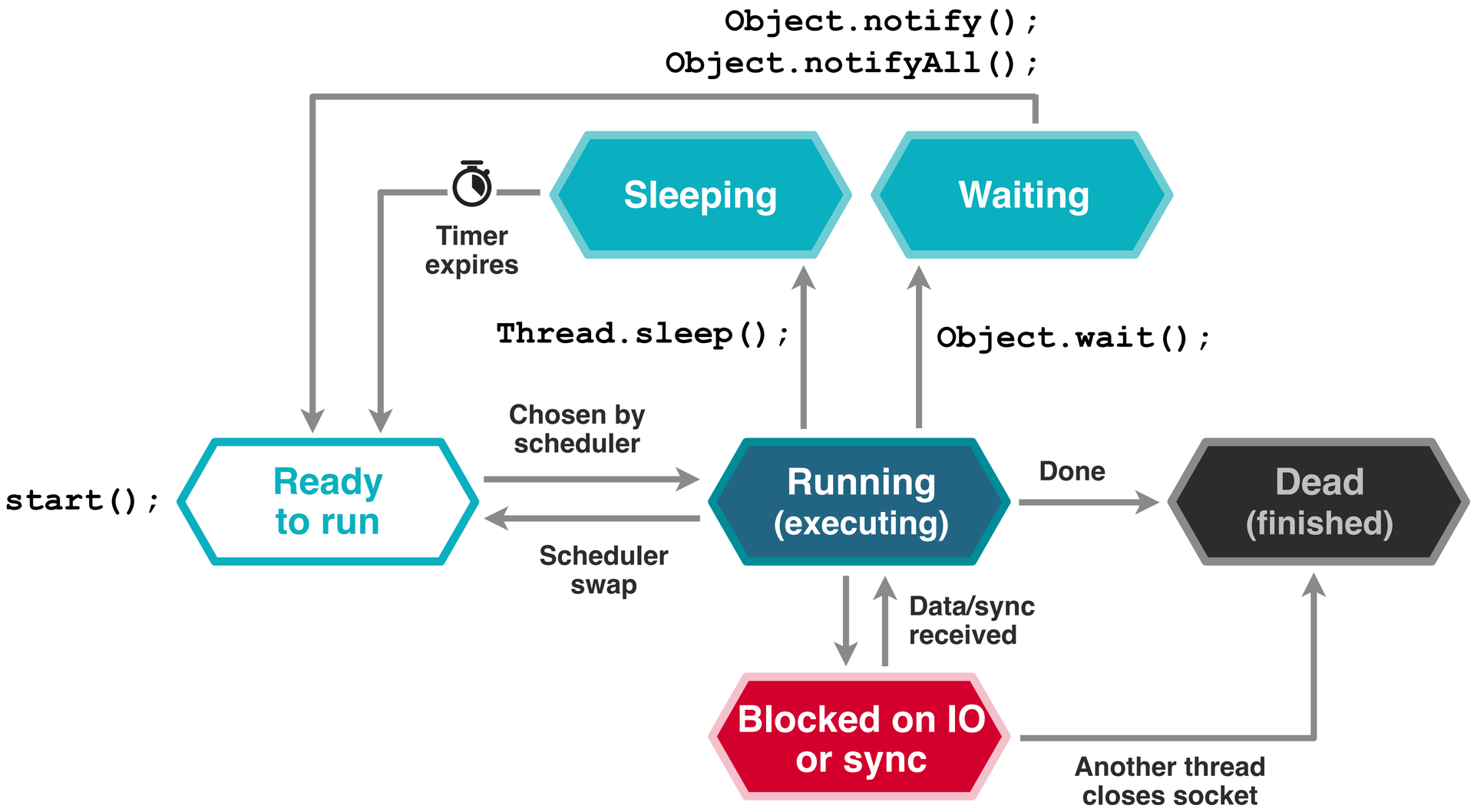
```
Counter c = new UnprotectedCounter(); // init to 0

Runnable r = () -> {
    for (int i = 0; i < REPS; i++)
        c.increment();
};

Thread tA = new Thread(r);
Thread tB = new Thread(r);
long start = System.currentTimeMillis();
tA.start();
tB.start();
tA.join();
tB.join();
long fin = System.currentTimeMillis();
int diff = c.increment() - (2 * REPS + 1);
System.out.println("Diff: " + diff);
System.out.println("Elapsed: " + (fin - start));
```

```
 A0: aload_0
 A1: aload_0
 A2: getfield #2 // Field i:I
 A5: iconst_1
 A6: iadd
 A7: dup_x1
 B0: aload_0
 B1: aload_0
 B2: getfield #2 // Field i:I
 B5: iconst_1
 B6: iadd
 B7: dup_x1
 A8: putfield #2 // Field i:I
A11: ireturn
 B8: putfield #2 // Field i:I
B11: ireturn
```

Java’s Original Thread & Synchronization API

- Scheduling Basics
- The Java Thread class
- Thread States
- Thread Methods

Thread Lifecycle



The Java Thread Class

- Thread is a lightweight unit of execution
  - Smaller than a process
  - Still capable of executing arbitrary Java code
- Each thread is a full unit of execution to OS
  - Own stack and program counter
  - Main address space (heap) shared between all threads in process
- Java has supported multithreaded programming since 1.0

Thread States

- NEW
  - Thread has been created but `start()` method has not been called
  - All threads start in this state.

- RUNNABLE
  - The thread is running or is available to run
  - Operating system is responsible for scheduling it

Thread States

- BLOCKED
  - The thread is not running
  - It is waiting to acquire a lock or is in a system call

- WAITING
  - The thread is not running
  - It has called `Object.wait()` or `Thread.join()`

Thread States

- TIMED_WAITING
  - The thread is not running
  - It has called **`Thread.sleep()`** or **`Object.wait()`**

- TERMINATED
  - Thread has completed execution.
  - Its **`run()`** method has exited normally or by throwing an exception.

Java Thread Methods

- `getId()`
- `setName()` and `getName()`
- `getState()`
- `isAlive()`
- `start()`
- `interrupt()`
- `join()`

Java Thread Methods

- `getPriority()` and `setPriority()`
- `setDaemon()`
- `setUncaughtExceptionHandler()`

setUncaughtExceptionHandler

```
// This thread just throws an exception
Thread handledThread = new Thread(() -> {
    throw new UnsupportedOperationException(); });

// Giving threads a name helps with debugging
handledThread.setName("My Broken Thread");

// Here's a handler for the error
handledThread.setUncaughtExceptionHandler((t, e) -> {
    System.err.printf("Exception in thread %d '%s':" +
                      "%s at line %d of %s%n",
            t.getId(),    // Thread id
            t.getName(),  // Thread name
            e.toString(), // Exception name and message
            e.getStackTrace()[0].getLineNumber(),
            e.getStackTrace()[0].getFileName()); });
handledThread.start();
```

Deprecated Java Thread Methods

- `stop()`
  - Almost impossible to use without violating safety
- `suspend()`, `resume()` and `countStackFrames()`
  - The `suspend()` mechanism does not release any monitors
- `destroy()`
  - Never implemented
  - Would have suffered from the same races as `suspend()`

Native Methods of Object

```
static JNINativeMethod methods[] = {
    {"hashCode",    "()I",                    (void *)&JVM_IHashCode},
    {"wait",        "(J)V",                   (void *)&JVM_MonitorWait},
    {"notify",      "()V",                    (void *)&JVM_MonitorNotify},
    {"notifyAll",   "()V",                    (void *)&JVM_MonitorNotifyAll},
    {"clone",       "()Ljava/lang/Object;",   (void *)&JVM_Clone},
};
```

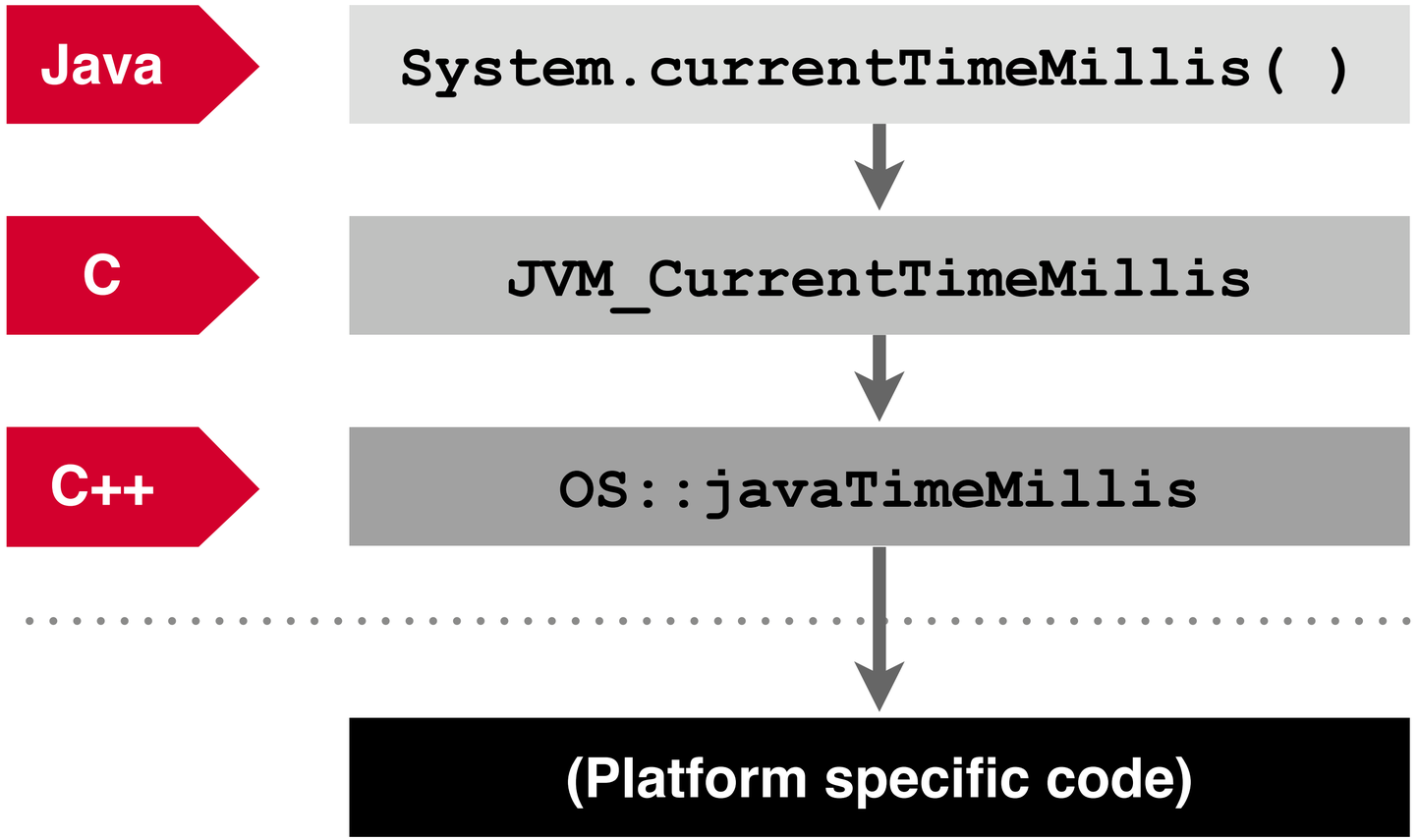
Native Code for an Object Wait

```
JVM_ENTRY(void, JVM_MonitorWait(JNIEnv* env, jobject handle, jlong ms))
    JVMWrapper("JVM_MonitorWait");
    Handle obj(THREAD, JNIHandles::resolve_non_null(handle));
    JavaThreadInObjectWaitState jtiows(thread, ms != 0);
    if (JvmtiExport::should_post_monitor_wait()) {
        JvmtiExport::post_monitor_wait((JavaThread *)THREAD, (oop)obj(), ms);
    }
    ObjectSynchronizer::wait(obj, ms, CHECK);
JVM_END
```

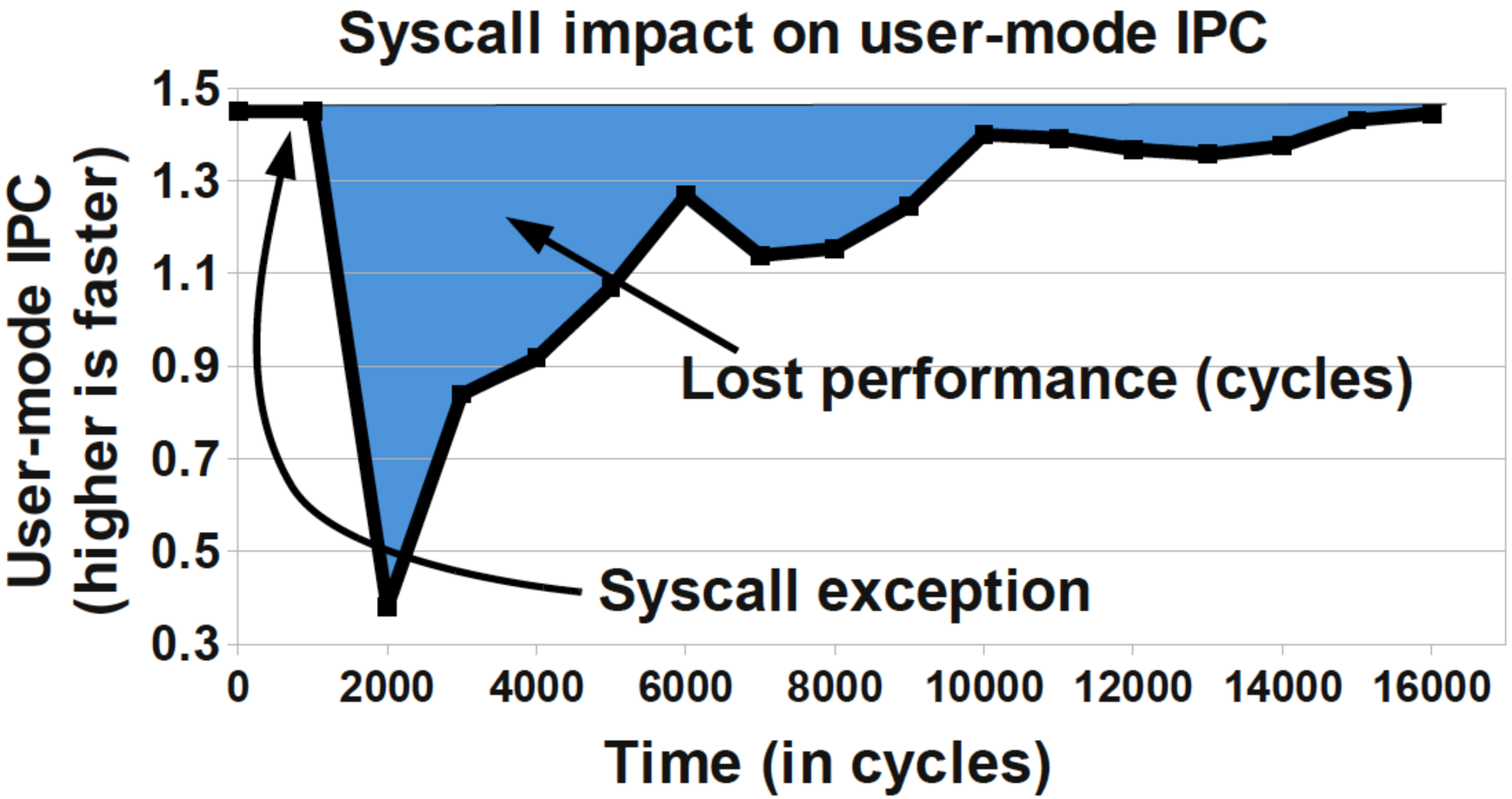
A Question of Time

- Schedulers depend on timing subsystems
- Modern hardware may have up to 4 timers
  - RTC, 8254, TSC and possibly HPET
- Also NTP
- `System.nanoTime()` vs `currentTimeMillis()`

A Question of Time – Accessing the OS



Impact of System Calls



From: FlexSC: Flexible System Call Scheduling with Exception-Less System Calls, by Livio Stoares & Michael Stumm

The Java Memory Model (JMM)

- Java has had a formal model of memory since 1.0
  - Heavily revised & fixed for Java 5 (JSR 133)
  - Continuing to evolve (new updates in Java 9)

- Increasingly important

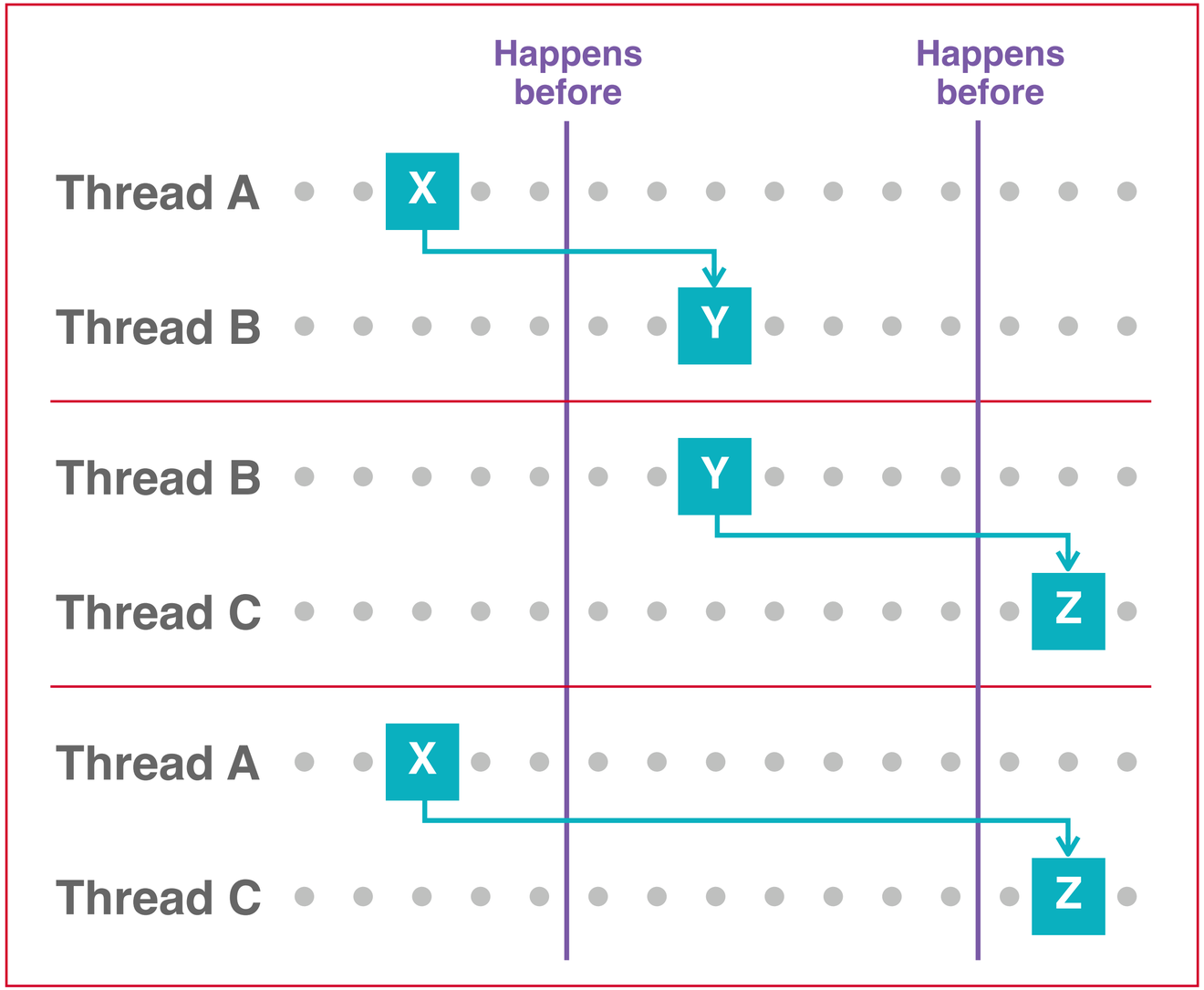
- What happens when two cores access same data?
  - When are they guaranteed to see the same value?
  - How do memory caches affect this?
  - Cache coherency protocols (MESI)

The Java Memory Model (JMM)

- Two possible approaches
  - Strong memory model (all cores always see same values)
  - Weaker memory model (special cache instructions)
- The JMM has a weak memory model
  - Fits better with trend in processors
  - Makes porting easier
- A strong memory model
  - requires more implementation code on top of weak hardware
  - scales worse as number of cores increases
  - Cache invalidation swamps the memory bus

The JMM

- Mathematical description of memory
- Most impenetrable part of JLS
- JMM makes minimum guarantees
  - Real JVMs (and CPUs) may do more



JMM – Key concepts

- Happens-Before
  - One block of code fully completes before other can start
- Synchronizes-With
  - Action will synchronize its view of an object with main memory
- As-If-Serial
  - Instructions appear to execute in-order
- Release-Before-Acquire

Synchronizes-with

- Threads have their own description of an object's state
  - Must be flushed to main memory (& other threads)

- **`synchronized`** means that local view has been **synchronized with** with other threads

- Defines touch-points where threads must perform synching

Synchronization Summary

- Unsynchronized methods
  - Don’t look at or care about the state of any locks
  - Can progress while synchronized methods are running

Synchronization Summary

- Unsynchronized methods
  - Don’t look at or care about the state of any locks
  - Can progress while synchronized methods are running
- Only objects can be locked – not primitives.
- Locking an `Object[]` doesn’t lock the individual objects

Synchronization Summary

- Unsynchronized methods
  - Don’t look at or care about the state of any locks
  - Can progress while synchronized methods are running
- Only objects can be locked – not primitives.
- Locking an `Object[]` doesn’t lock the individual objects
- A synchronized method is equivalent to a `synchronized (this) { ... }` block that covers the entire method
- A static synchronized method locks the `Class` object, because there’s no instance object to lock.

Synchronization Summary

- Synchronization in an inner class is independent of the outer class
- When locking class object, two options

Synchronization Summary

- Synchronization in an inner class is independent of the outer class
- When locking class object, two options
  - Explicit lock on the class literal
  - Using `getClass()`
  - The approaches have different behaviour in a subclass

Synchronization Summary

- Synchronization in an inner class is independent of the outer class
- When locking class object, two options
  - Explicit lock on the class literal
  - Using `getClass()`
  - The approaches have different behaviour in a subclass
- `synchronized` isn't part of the method signature
  - It can't appear on a method declaration in an interface

Synchronization Summary

- Synchronization in an inner class is independent of the outer class
- When locking class object, two options
  - Explicit lock on the class literal
  - Using `getClass()`
  - The approaches have different behaviour in a subclass
- `synchronized` isn't part of the method signature
  - It can't appear on a method declaration in an interface
- Java's locks are reentrant
  - A recursive call to a synchronized method won't self-deadlock

Modern JDK Concurrency Classes

- Executors
- Concurrent Data Structures
- Exercise: Getting to Know ConcurrentHashMap
- Queues, Latches and Barriers
- Locks in java.util.concurrent
- Q&A

Execution & Executors

- Deferred Execution
- Executors & Threadpools

Deferred Execution

- A class is the smallest unit of functionality
  - Functionality is loaded & linked as classes
  - Must go through classloading
  - There’s no other way to do it
- Java has no “free functions”

Deferred Execution

- Representing Execution means we're deferring execution
  - Have a reference to some code which will be executed later
  - Callable: "Here's some code. Run it when we call `call()`"
  - Runnable: "Start this code up in a different thread (via a Thread ctor)"
  - Callbacks: "Here's some code - call it back when something happens"

- Deferring Execution comes up in other contexts
  - Closures / Lambda Expressions / Function pointers
  - Libraries (e.g. map / filter idioms)

Executors Factory

- `java.util.concurrent` has factories for making threadpools
  - In the Executors class
  - Another plurals class!

- Like this:

```
ScheduledExecutorService stpe =
    Executors.newScheduledThreadPool(2);
```

Don’t Roll Your Own!

- Greenspun’s Tenth Rule:

> “Any sufficiently complicated C or FORTRAN program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.”

Don’t Roll Your Own!

- Greenspun’s Tenth Rule:

> “Any sufficiently complicated C or FORTRAN program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.”

- There’s a similar law about multithreaded programs
  - If you aren’t using `java.util.concurrent`
  - Then you’ll end up reimplementing half of the classes in it
  - Badly, usually.

Ready-Made ThreadPools

- `Executors` has factory methods
  - `newFixedThreadPool(int nThreads)`
  - `newCachedThreadPool()`
  - `newSingleThreadExecutor()`
  - `newScheduledThreadPool(int corePoolSize)`

Fixed-Size ThreadPool

- Threads will be reused to run multiple tasks
- Prevents having to pay the cost of thread creation
- If threads are in use new tasks are stored in a queue
- Useful if task flow is stable and known

CachedThreadPool

- Will create new threads as required
- Will reuse threads where possible
- Created threads are kept for 60 seconds
  - After which they will be removed from the cache
- Can give better performance with small asynchronous tasks
  - Or bursty workloads

Single Thread Executor

- Backed by a single thread
- Newly submitted tasks are queued until the thread is available
- Can be useful to control number of tasks concurrently executed
- Also useful for testing

ScheduledThreadPoolExecutor

- Common choice of threadpool for many applications

```
ScheduledExecutorService stpe =
    Executors.newScheduledThreadPool(poolsize);
```

- Schedule events to execute at a fixed rate

```
stpe.scheduleAtFixedRate(msgReader, initialDelay,
    period, TimeUnit.MILLISECONDS);
```

  - Returns a ScheduledFuture for pending execution
  - Good for controlled concurrency work

`ScheduledThreadPoolExecutor`

- Schedule all threads to run
  - **`stpe.invokeAll()`**
  - Executes a bunch of tasks at once
  - Returns a collection of Future objects
  - Can be beneficial when you have sufficient hardware

- **`shutdown()`**, **`shutdownNow()`**
  - Used to shutdown threadpools

Concurrent Data Structures

- Atomic Data Structures
- Concurrent Maps
- Concurrent Lists and Sets

`AtomicInteger` & `AtomicLong`

- **`AtomicInteger`** & **`AtomicLong`** can be used as counters

- Can be used as a replacement for volatile variables
  - More flexible
  - Can do “state-dependent” updates (e.g. increment)
  - Still lock-free & can’t deadlock

`AtomicInteger` & `AtomicLong`

- **`AtomicInteger`** & **`AtomicLong`** can be used as counters
- Can be used as a replacement for volatile variables
  - More flexible
  - Can do “state-dependent” updates (e.g. increment)
  - Still lock-free & can’t deadlock
- Have composite operations
  - add or increment / decrement combined with get
  - Compare And Set (CAS) operations
  - These will execute atomically
  - Results are immediately seen by other threads

Atomic Classes

- Atomic classes don’t inherit from similarly named classes
  - **`AtomicBoolean`** can’t be used in place of a **`Boolean`**
  - **`AtomicInteger`** isn’t an **`Integer`**
  - Does extend **`Number`**

- Why Not?

Adders

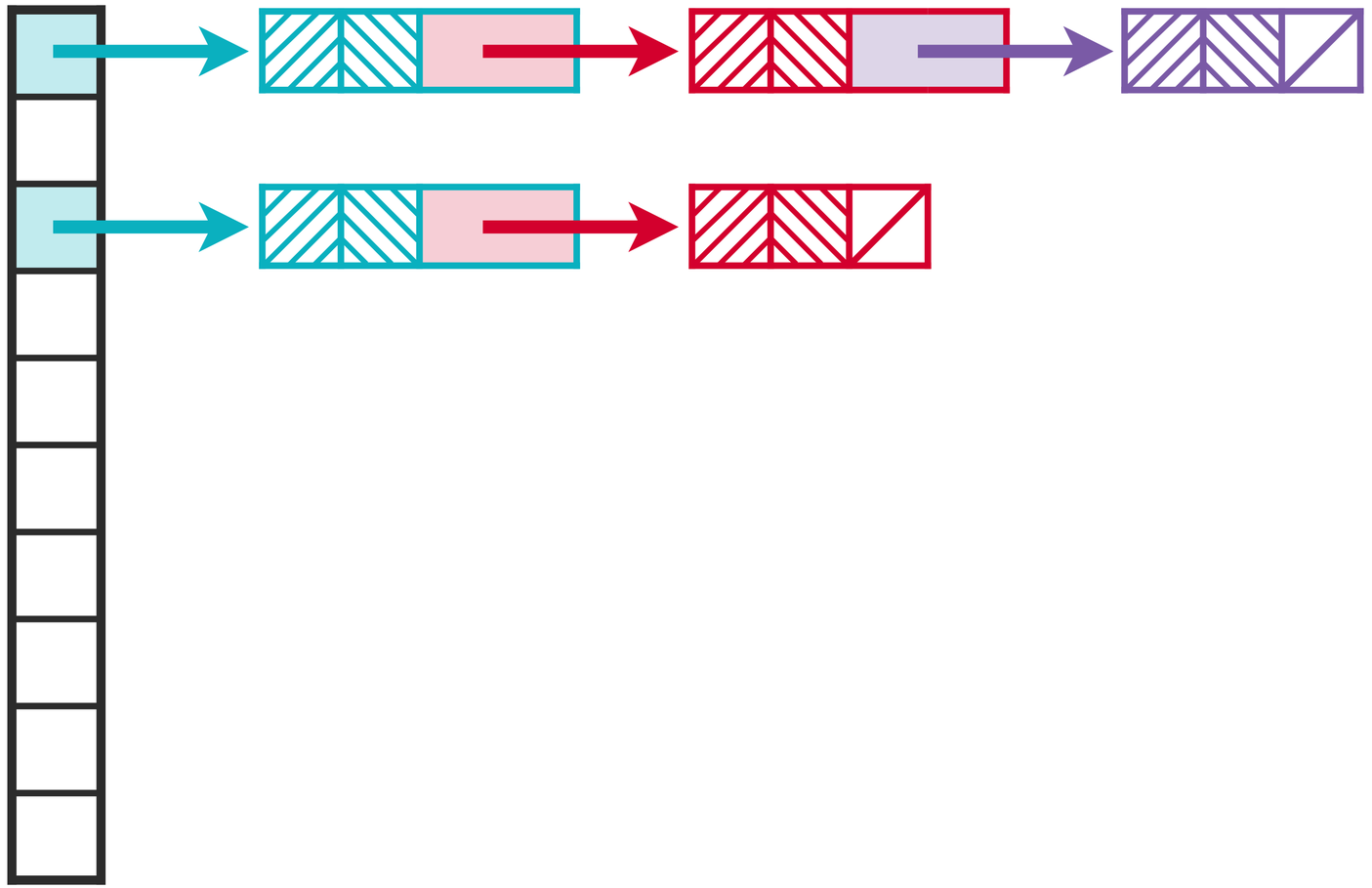
- Java 8 introduces scalable accumulators & adders
- Performance enhancement for contented shared state
  - As number of threads adding to an `AtomicLong` rises performance worsens
- `LongAdder` is designed to solve this problem
  - More scalable in the number of threads that are adding

Accumulators & Adders

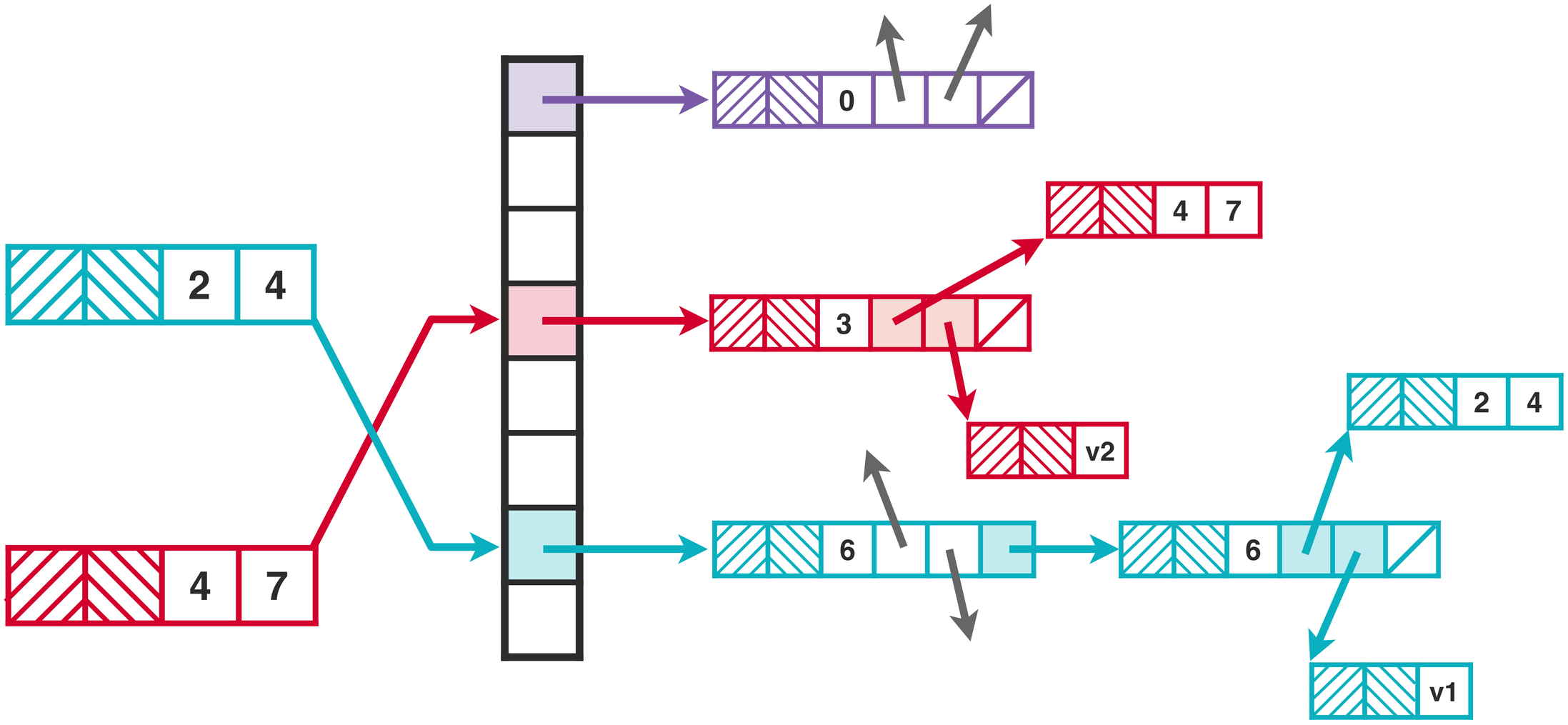
- Accumulators are a generalisation of adders
- Related to the reduce operations from functional code
  - `new DoubleAdder()` is just

    `new DoubleAccumulator((x, y) -> x + y, 0.0)`
- Same contention behaviour as adders

Hashing – for Data Structures



`HashMap` at Runtime



`SimpleDict` Internals

```
public class SimpleDict implements Map<String, String> {
    private final Node[] table; // Hash buckets (head of link-lists)
    private int size; // num of entries in the dict
```

https://github.com/kittylyst/optimizing-java

Aside: `SimpleDict` Internals

```
public String get(Object key) {
    // Null keys are not supported
    if (key == null)
        return null;

    int hash = key.hashCode();
    int i = indexFor(hash, table.length);
    for (Node e = table[i]; e != null; e = e.next) {
        String k;
        if (e.hash == hash
            && ((k = e.key) == key || key.equals(k)))
            return e.value;
    }

    return null;
}

private int indexFor(int h, int length) {
    return h & (length - 1);
}
```

```
private class Node implements
   Entry<String, String> {
      final int hash;
      final String key;
      String value;
      Node next;

      Node(int h, String k,
         String v, Node n) {
            hash = h;
            key = k;
            value = v;
            next = n;
   }
}
```

CHM v7

- Simple segmented model
- One `ReentrantLock` object per hash bucket
- Default concurrency level: 16
- So CHM v7 is memory intensive
  - ~ sizeof(HashMap) + 16 * sizeof(ReentrantLock)

Class	Size
`HashMap`	216
`ReentrantLock`	72
`ConcurrentHashMap`	2272

CHM v8

- In Java 8 implementation of **`ConcurrentHashMap`** changes
- Once hash chain crosses threshold
- Replace linked list with balanced tree
- Improves worst-case perf from O(n) to O(log n)
  - The case of high hash collisions
- Increased code complexity

New Methods on `Map`

- Java 8 introduced new methods on Map
  - Using the default methods feature
  - Add “missing” features from CHM
  - lambda expressions

New Methods on `Map`

default	V	compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)
default	V	computeIfAbsent(K key, Function<? super K,? extends V>
default	V	computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)
default	void	forEach(BiConsumer<? super K,? super V> action)
default	V	getOrDefault(Object key, V defaultValue)
default	V	merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)
default	V	putIfAbsent(K key, V value)
default	boolean	remove(Object key, Object value)
default	V	replace(K key, V value)
default	boolean	replace(K key, V oldValue, V newValue)
default	void	replaceAll(BiFunction<? super K,? super V,? extends V> function)

What about `ArrayList`?

- CHM intended to be a drop-in replacement for **`HashMap`**
- What about the sequential case (**`ArrayList`**)?
- Could just use **`Collections.synchronizedList()`**
  - Usually a lot poorer performance
- Instead, want a sequential data structure that is consistent
  - Will not throw a **`ConcurrentModificationException`**
- Uses a versioned data model
- Also **`CopyOnWriteArraySet`**

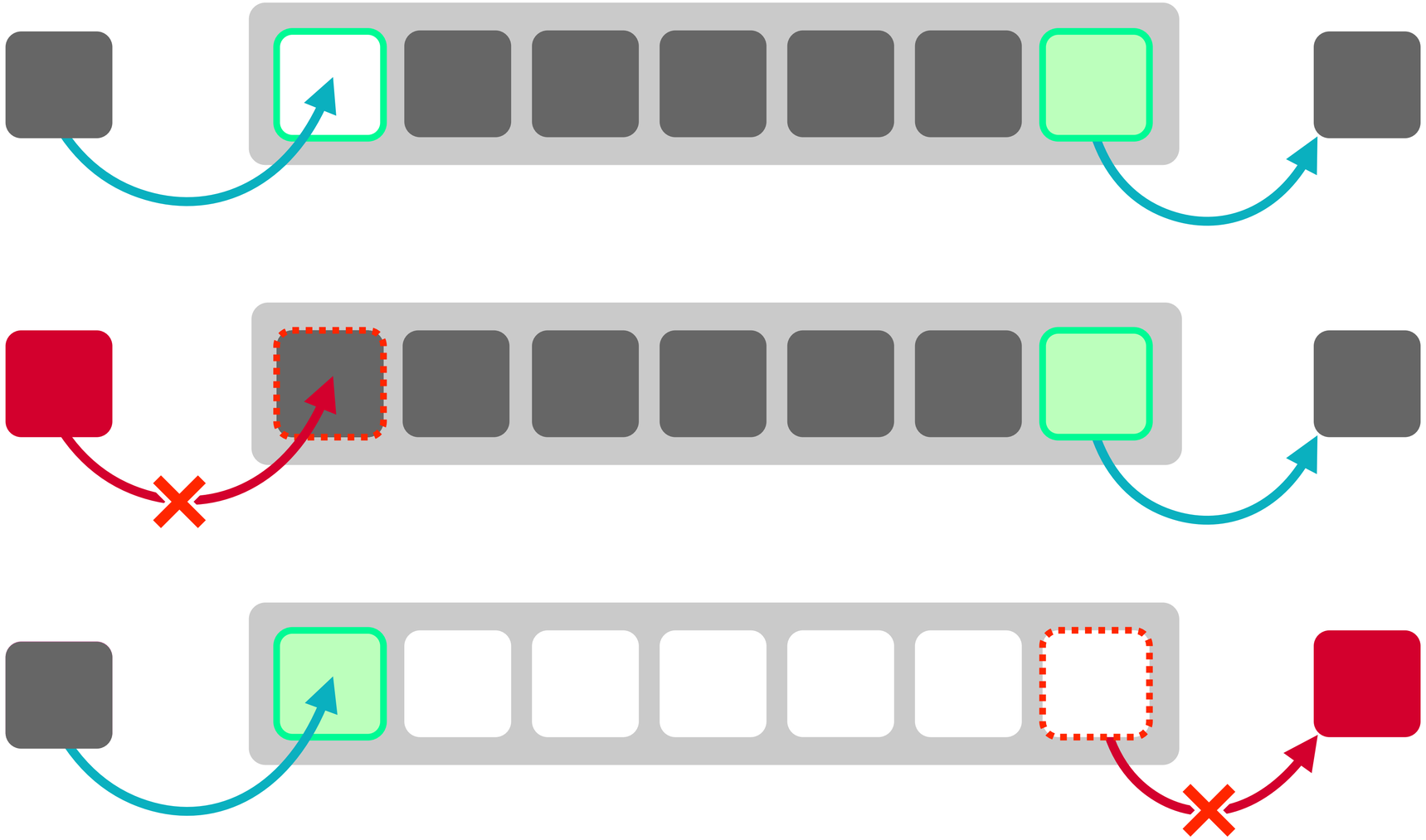
More Building Blocks

- Queues
- Latches
- Other Barriers

Blocking Queues

- Queues are often used to link execution together
- Efficient way to hand off work between threads
- **`BlockingQueue`** is the usual choice
  - Two basic impls – **`ArrayList`** and **`LinkedList`** backed
  - Java 7 introduces the new **`TransferQueue`**

Blocking Queue



`BlockingQueue`

- Offers several different ways to interact with it
  - **`add()`**, **`remove()`**, **`element()`** throw exceptions on failure
  - **`offer()`**, **`poll()`**, **`peek()`** return a special value on failure
  - **`put()`**, **`take()`** block until they succeed (note - no “examine” method)
- See Javadoc for full details

`BlockingQueue` methods

```
public interface BlockingQueue<E> extends Queue<E>
```

	Throws exception	Special value	Blocks	Times out
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examine	`element()`	`peek()`	not applicable	not applicable

`BlockingQueue`

- It’s best to pick one set of operations and stick to it
  - Mixing and matching can be confusing

- Blocking operations with timeouts can be very useful
  - e.g. For backoff / retry
  - `put()`, `take()` have overloads which allow for backoff

`LinkedBlockingQueue`

- Implemented using **`LinkedList`**
- Can specify a capacity
  - By default, is **`Integer.MAX_VALUE`**
  - Usually thought of as an unbounded queue
- Slightly better perf characteristics for low-latency apps
- A common default choice for producer-consumer apps

`ArrayBlockingQueue`

- Implemented using **`ArrayList`**
- Like **`LinkedBlockingQueue`**, has interesting special methods
  - e.g. **`drainTo(Collection T)`**
  - Drains all available elements in the queue into the collection
- Faster than **`LinkedBlockingQueue`** in single thread case
- Like **`LinkedBLockingQueue`**, is a FIFO implementation

`CountDownLatch`

- A group consensus construct
- Constructor takes an **`int`** (the count)
- **`countDown()`** decrements the count
- **`await()`** blocks until **`count == 0`**
  - i.e. consensus
  - Releases all waiting threads at once

CountDownLatch

CountDownLatch

- No way to increase the count - only decrease
- Use cases
  - Cache population & startup
  - Multithreaded testing
- countdown-and-await pattern common
  - Simulates Happens-Before edge
- Not the only way to use this

CountDownLatch Example

```
public class LatchExample implements Runnable {
    private final CountDownLatch latch;

    public LatchExample(CountDownLatch latch) {
        this.latch = latch;
    }

    @Override
    public void run() {
        //Call an API
        System.out.println(Thread.currentThread().getName() + " Done API Call");
        try {
            latch.countDown();
            latch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " Continue processing");
    }
```

`CyclicBarrier`

- **`CountDownLatch`** is 1-shot
- What if we want to reset or reuse the barrier?
- **`CyclicBarrier`**
  - Clean reset if the latch has been released by threads reaching it
  - Otherwise, throws a **`BrokenBarrierException`**

CyclicBarrier

```
public class ResynchingWorker implements Runnable {
    private final CyclicBarrier barrier;
    public ResynchingWorker(CyclicBarrier b) {
        barrier = b;
    }
    public void run() {
        try {
            // Wait until all Worker threads are ready
            barrier.await();
            // Simulate some work...
            Thread.sleep((int) (Math.random() * 1000));
            // Wait until all Worker threads are finished
            barrier.await();
        } catch (InterruptedException ignore) {
            // Ignore
        } catch (BrokenBarrierException bbe) {
            System.err.println("Barrier broken!");
        }
    }
}
```

`Phaser`

- Synchronization barrier similar to **`CyclicBarrier`**
  - More flexible
- Registered threads can change over time
  - Methods **`register()`** and **`bulkRegister(int)`**
- Can repeatedly await (like **`CyclicBarrier`**)

Phaser

- Methods **`arrive()`** & **`arriveAndDeregister()`** record arrival
- Non-blocking, return an associated arrival phase number
- When the final thread for a given phase arrives
  - An optional action is performed
  - The phase advances
- Wait for given phase to advance – **`awaitAdvance(int phase)`**
- Very flexible, but not often needed

Intrinsic Locks – Drawbacks

- There is only one type of lock
- Applies equally to all synchronized operations on locked object

Intrinsic Locks – Drawbacks

- There is only one type of lock
- Applies equally to all synchronized operations on locked object
- Lock acquired at the start of synchronized block or method
- Lock released at the end of block or method

Intrinsic Locks – Drawbacks

- There is only one type of lock
- Applies equally to all synchronized operations on locked object
- Lock acquired at the start of synchronized block or method
- Lock released at the end of block or method
- Lock is either acquired or the thread blocks
  - No other outcomes are possible
- We can imagine more general classes of locks

Reimagined Locks

- Different types of locks
  - e.g. reader and writer locks

Reimagined Locks

- Different types of locks
  - e.g. reader and writer locks
- Not restrict locks to blocks
  - Allow a lock in one method and unlock in another

Reimagined Locks

- Different types of locks
  - e.g. reader and writer locks
- Not restrict locks to blocks
  - Allow a lock in one method and unlock in another
- If a thread cannot acquire a lock allow back out – `tryLock()`
- Allow a thread attempted acquire with timeout

`Lock`

- The main interface for this reimagining is **`Lock`**
  - **`java.util.concurrent.locks.Lock`**

- Key methods
  - **`lock()`** - acquires the lock
  - **`newCondition()`** - creates a condition for wait / notify
  - **`tryLock()`** - acquire the lock (with optional timeout)
  - **`unlock()`** - release the lock

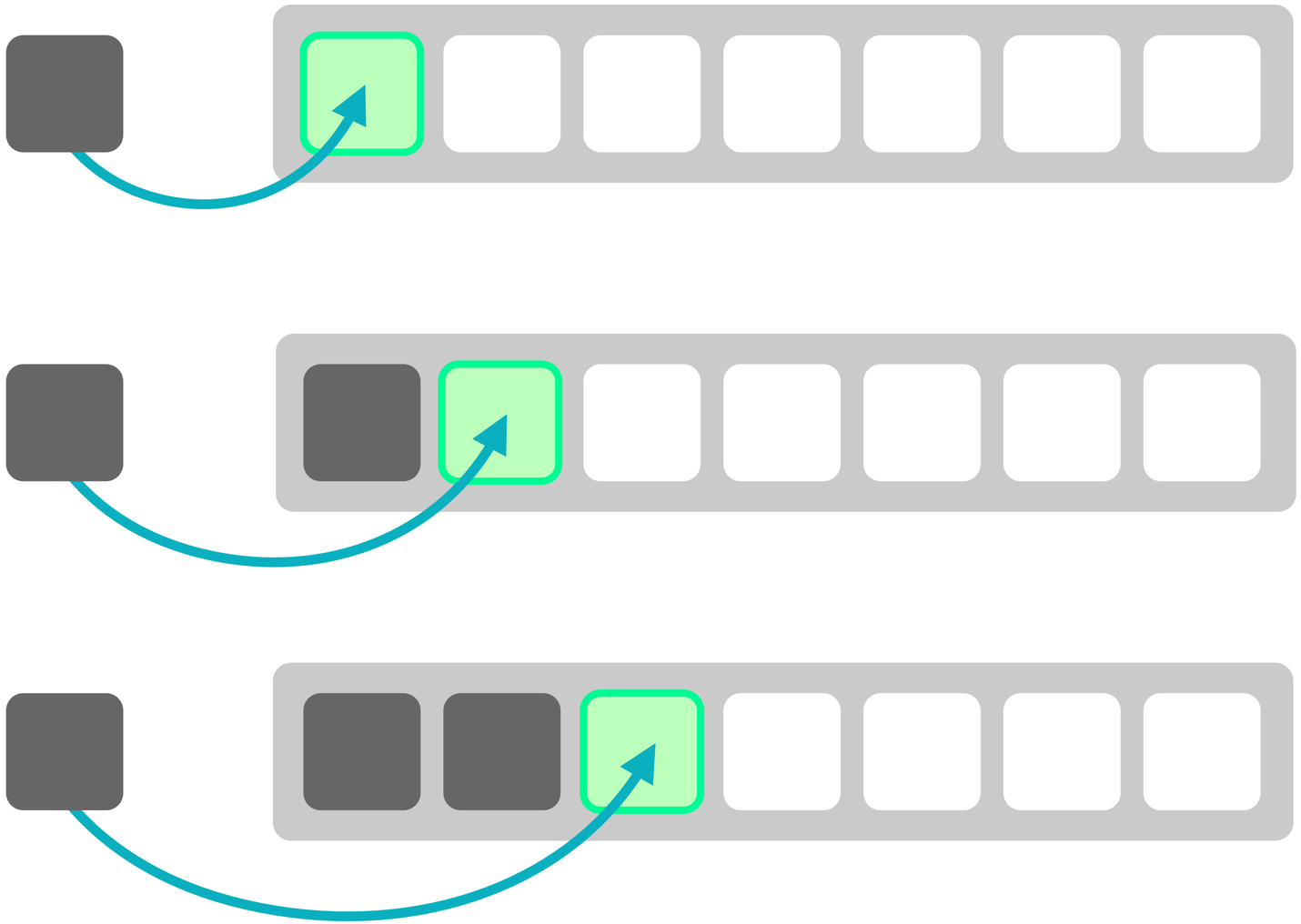
`Lock`

- Two main implementations
  - **`ReentrantLock`** - replicates the “synchronized” lock
  - **`ReentrantReadWriteLock`** - used for reader / writer locks
- Also the non-reentrant implementation
  - Very rarely seen in practice

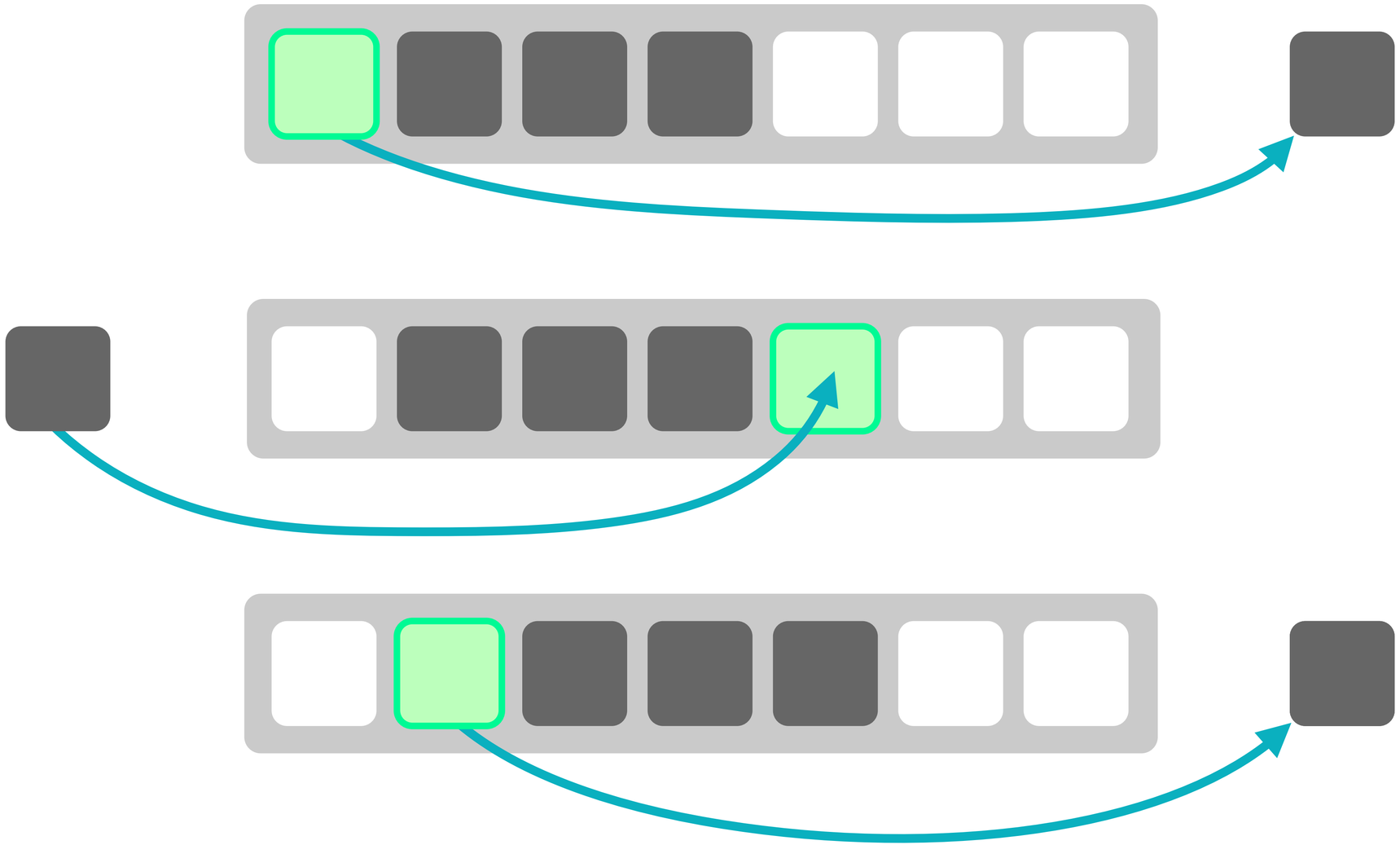
`Condition`

- Intrinsic concurrency only has 1 condition to wait on
- With j.u.c we can be more flexible
- **`Lock.newCondition()`** creates a **`Condition`** variable
  - Can represent any condition
  - Multiple conditions can be created on a lock
  - No “semantic” meaning as in intrinsic concurrency

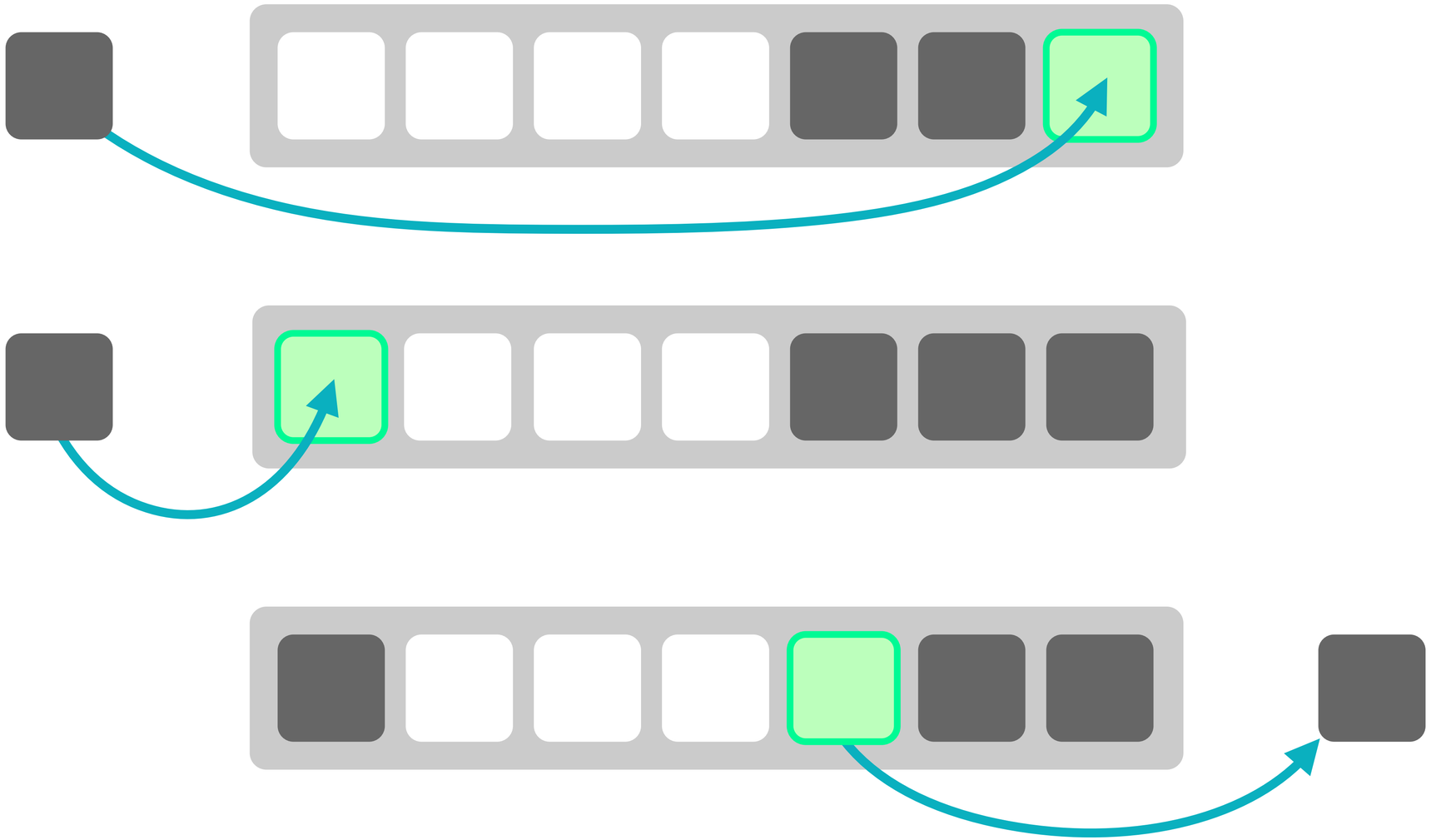
Example: Bounded Queue



Example: Bounded Queue



Example: Bounded Queue



Lock & Condition

```
Lock lock = new ReentrantLock();
Condition notFull  = lock.newCondition();
Condition notEmpty = lock.newCondition();
Object[] items = new Object[100];
int putptr, takeptr, count;
```

Lock & Condition

```
// Thread 1
public void put(Object x) throws InterruptedException {
    lock.lock();
    try {
        // If full, release lock & wait until we get it back
        while (count == items.length) notFull.await();

        items[putptr] = x;
        if (++putptr == items.length) putptr = 0;
        ++count;

        // signal notEmpty that it can try for the lock again
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}
```

Lock & Condition

```
// Thread 2
public Object take() throws InterruptedException {
    lock.lock();
    try {
        // If empty, release the lock & wait
        while (count == 0) notEmpty.await();

        Object x = items[takeptr];
        if (++takeptr == items.length) takeptr = 0;
        --count;

        // Signal notFull that it can stop waiting & try for the lock
        notFull.signal();
        return x;
    } finally {
        lock.unlock();
    }
}
```

Exercise – Intrinsic Lock vs `ReentrantLock`

- Take the BoundedQueue example
- Write it using intrinsic concurrency as well as j.u.c
- Compare the throughput in single-reader, single-writer case
- What about 2 writers, 1 reader? 2 readers, 1 writer?
- What happens if Biased Locking is switched off?

https://github.com/kittylyst/optimizing-java

Under the Hood

- Inside AtomicInteger (VarHandles and Unsafe)
- Exercise: Build your own Atomic
- Q&A

Implementing Atomics

- Lock-free
  - Use CAS hardware where available
  - On all modern platforms
  - Otherwise, fall back to software implementation (lock-based)

- Java Support provided via Unsafe
  - e.g. `Unsafe::compareAndSwapInt()`
  - See later…

CAS Operations

- Pass in 2 values
  - “Expected current value”
  - “Requested new value”

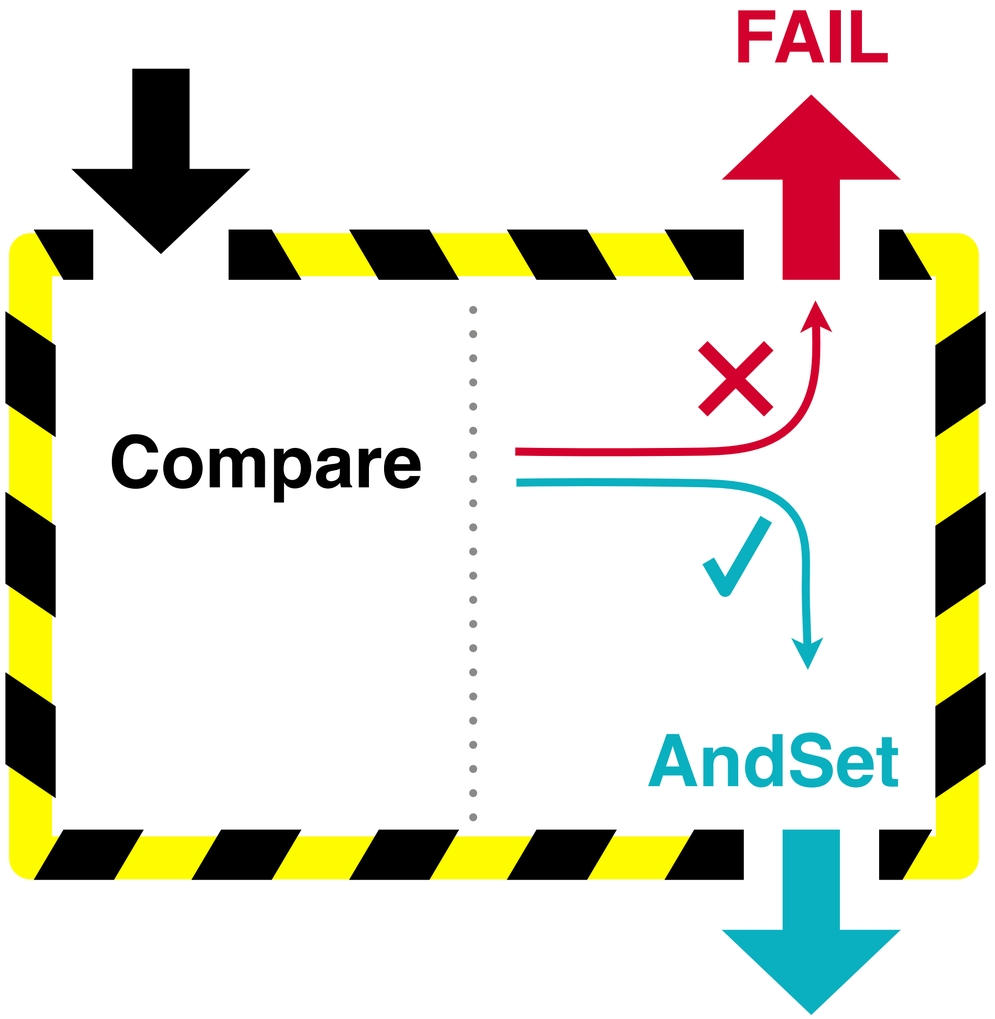
CAS Operations

- Pass in 2 values
  - “Expected current value”
  - “Requested new value”
- Compare expected current with actual current
- If they match, update to requested new value

CAS Operations

- Pass in 2 values
  - “Expected current value”
  - “Requested new value”
- Compare expected current with actual current
- If they match, update to requested new value
- If the match fails
  - Another thread has modified it
  - The update fails
- In both cases return the current value after the attempt
- Performed as a single operation

CAS Operations



Implementing Atomics

```
public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // setup
// ...

   static {
// ...
   }

    private volatile int value;
```

Implementing Atomics

```
/**
 * Gets the current value.
 *
 * @return the current value
 */
public final int get() {
    return value;
}

/**
 * Sets to the given value.
 *
 * @param newValue the new value
 */
public final void set(int newValue) {
    value = newValue;
}
```

Implementing Atomics

```
public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // setup to use Unsafe.compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    private volatile int value;
```

Implementing Atomics

```
/**
 * Atomically sets to the given value and returns the old value
 *
 * @param newValue the new value
 * @return the previous value
 */
public final int getAndSet(int newValue) {
    return unsafe.getAndSetInt(this, valueOffset, newValue);
}
```

Implementing Atomics – `Unsafe`

```
public final int getAndSetInt(Object o, long offset, int newValue) {
    int v;
    do {
        v = getIntVolatile(o, offset);
    } while (!compareAndSwapInt(o, offset, v, newValue));
    return v;
}
```

Implementing Atomics – `Unsafe`

```
public native int getIntVolatile(Object o, long offset);
/**
 * Atomically update Java variable to <tt>x</tt> if it is currently
 * holding <tt>expected</tt>.
 * @return <tt>true</tt> if successful
 */
public final native boolean compareAndSwapInt(Object o, long offset,
                                              int expected,
                                              int x);
```

CAS Operation / Spinlock

- Pass in 2 values
  - “Expected current value”
  - “Requested new value”

```
locked:
  dd 0

spin_lock:
  mov eax, 1
  xchg eax, [locked]
  test eax, eax
  jnz spin_lock
  ret

spin_unlock:
  mov eax, 0
  xchg eax, [locked]
  ret
```

Removal of `Unsafe`

- Many frameworks/libraries unable to move to Java > 8
  - Without a replacement for some 'safe' sun.misc.Unsafe features
  - Indirectly impacts everyone using a wide range of frameworks
  - Basically every application in the ecosystem

- What can we do?
  - Give Unsafe a pass for now
  - Create new supported APIs for Java 11 & inform that migration

Exercise

EXERCISE

Practical Considerations

- Concurrency vs Parallelism
- Amdahl’s Law
- Q&A

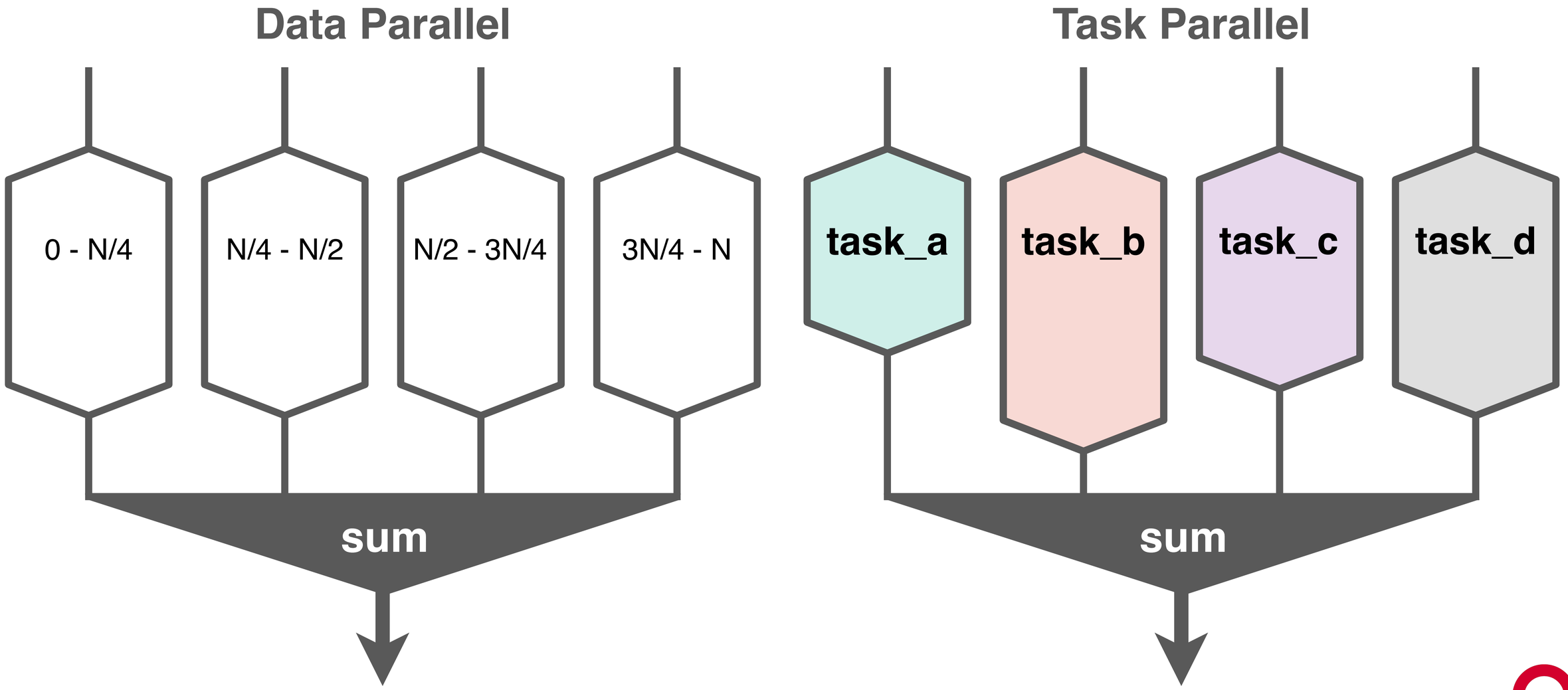
Concurrency vs Parallelism

- Concurrency
  - At least two threads are executing simultaneously
- Parallelism
  - At least two threads are making progress
  - Can include time-slicing
    - Including traditional UNIX timesharing

Parallelism

- Task
  - Distribute execution of operations over processes
  - Threads and Executors in Java
    - e.g. each thread services a user in Java EE App
- Data
  - Distribute data over different processes
  - Support built on top of Streams
    - e.g. process payroll and allocate each core 100 employees to process

Parallelism



Amdahl’s Law

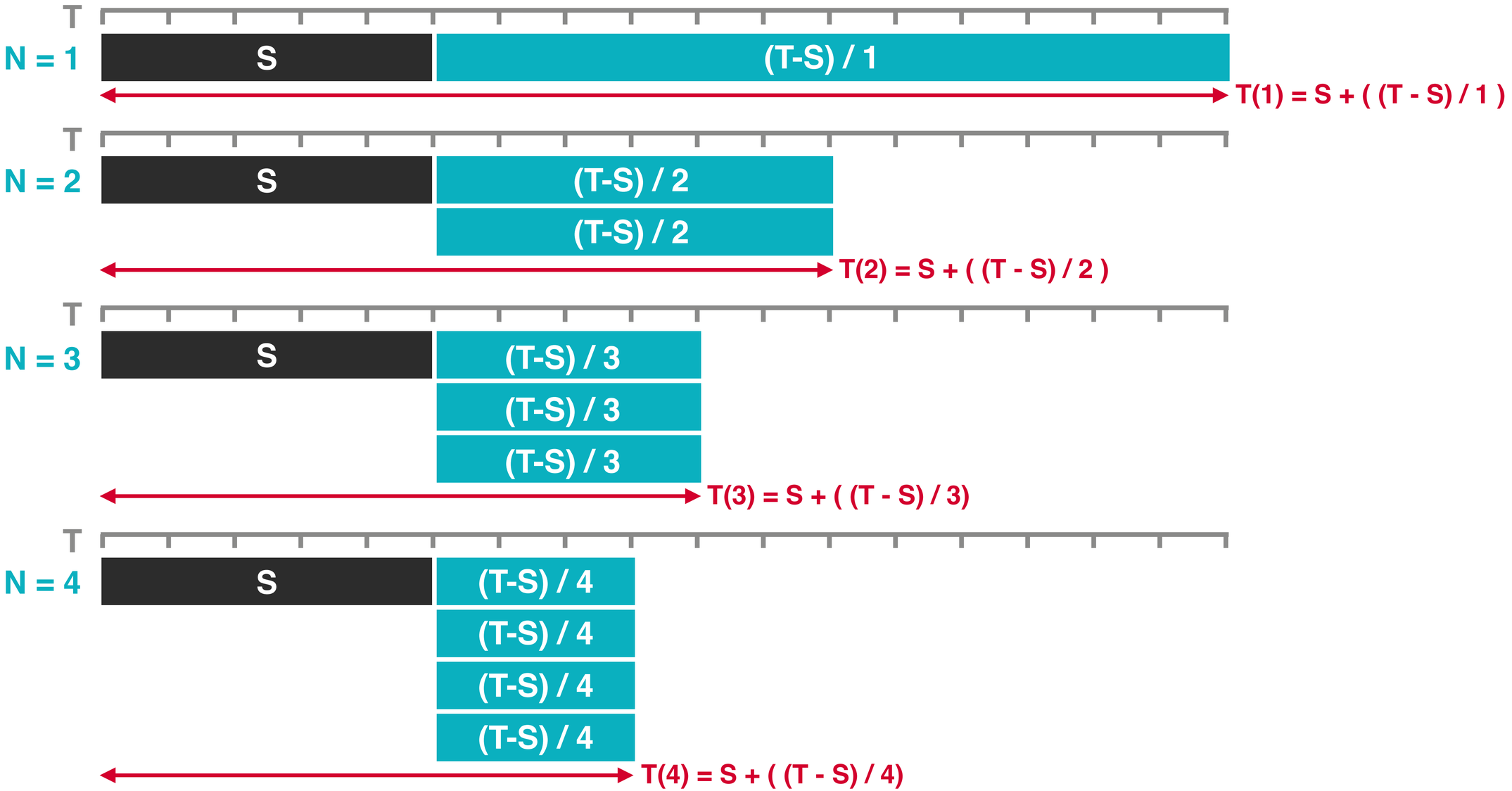
- Defines upper bound for parallel speedup

$$T(n) = T(1) * (s + (1 - s) /n)$$

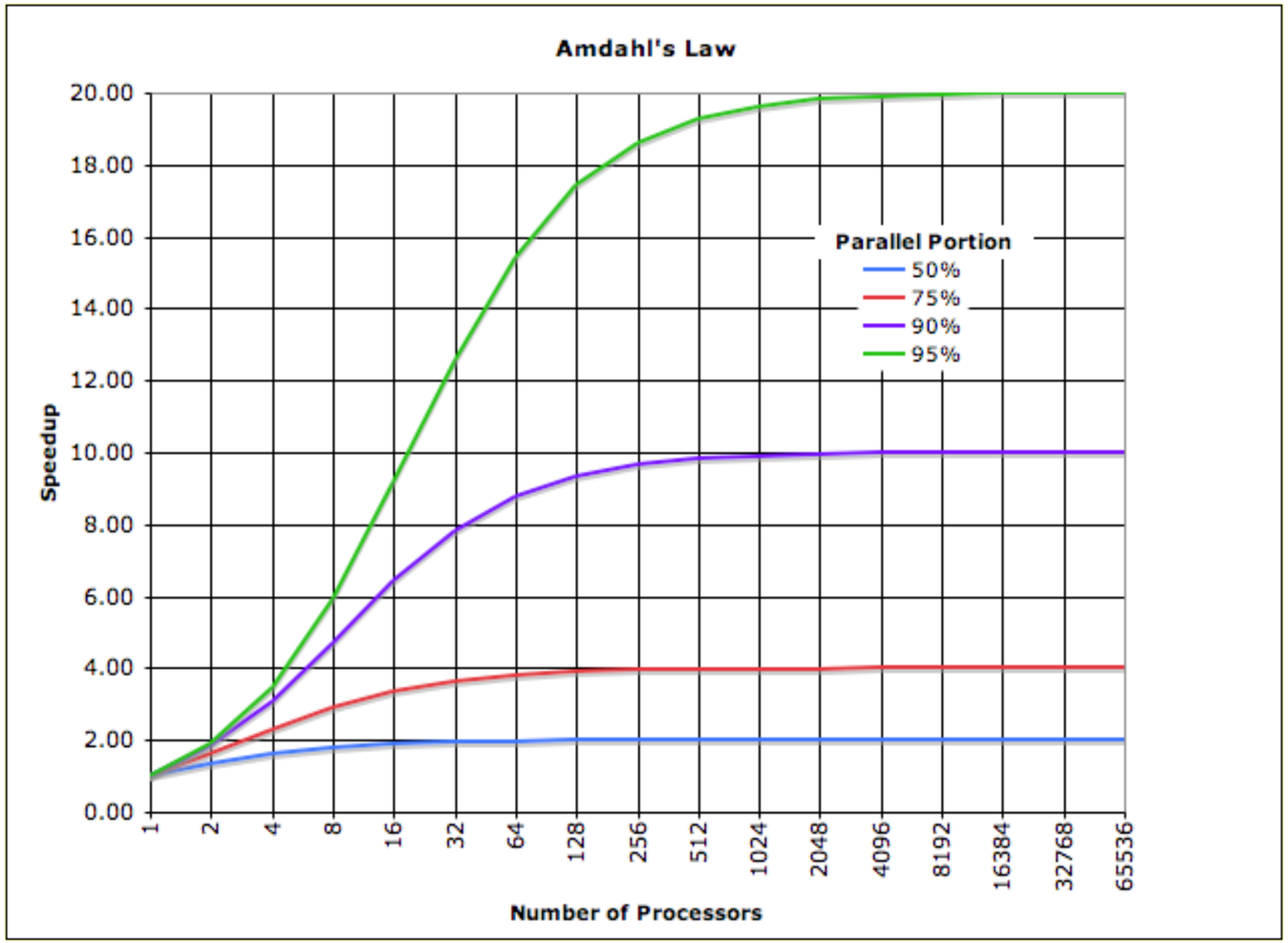
$$u * n = 1 / (s + (1 - s) /n)$$

- n = number of cores
- s = proportion of code that must be serial
- u = % utilisation (across all cores)

Amdahl’s Law


T
N = 1
S
(T-S) / 1
T(1) = S + ((T - S) / 1)
T
N = 2
S
(T-S) / 2
(T-S) / 2
T(2) = S + ((T - S) / 2)
T
N = 3
S
(T-S) / 3
(T-S) / 3
(T-S) / 3
T(3) = S + ((T - S) / 3)
T
N = 4
S
(T-S) / 4
(T-S) / 4
(T-S) / 4
(T-S) / 4
T(4) = S + ((T - S) / 4)

Parallelism



Amdahl's Law

- Invert Amadahl's Law
  - Calculate max s (serial overhead) for given utilisation
- E.g. To hit 70% utilisation across cores
  - s = 3 / (7 * (n-1))

#Cores (n)	Max Serial % (s)
1	100%
2	43%
4	14%
32	1.38%
48	<1%
400	< 0.1%

Concurrency Reading List

- **Java Concurrency in Practice** - B. Goetz et al.
- **Concurrent Programming in Java** - D. Lea
- **Optimizing Java** - B. Evans et al.
- **Mechanical Sympathy** (mailing list)
- **Concurrency Interest** (mailing list)