Collections & Concurrency

CON3531

Mike Duigou @mjduigou
Core Libraries Contributor

JavaOne ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Safe Harbour Statement & Recognition

- Mike is a **former** an Oracle employee
- Mike was a member of Java Platform Core Libraries team
- Mike is still an active contributor to OpenJDK
- Mike speaking today for nobody but himself

- This presentation was created while Mike worked at Oracle
- Chris Hegarty co-authored the material and previously co-presented this session in 2013 & 2014. Chris promises to return next year!

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Session Agenda

1. Introductory Session
2. Topic is understanding concurrent performance
3. Use of Collections is useful but incidental

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

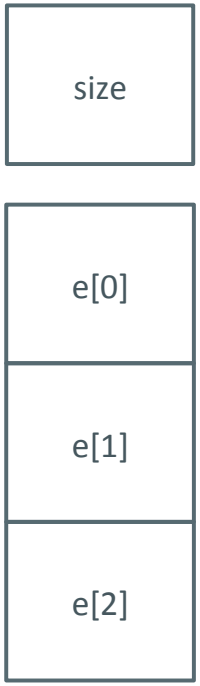
Once upon a thread

- Program flow used to be simple
  - Start, some looping, end
  - Easy to understand and analyze
  - Deterministic
- Program performance consisted of
  - **O** algorithmic complexity
  - Counting CPU cycles

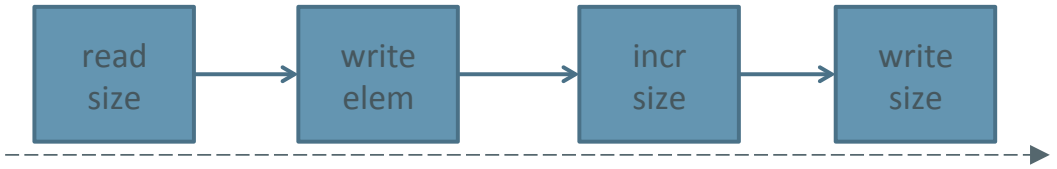
Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Add an element to ArrayList

One step at a time




```
public void add(E elem) {
  ensureCapacity(size + 1);
  e[size++] = elem;
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Limits to Sequential

Pushing a rope (thread)

- Works great until there is more work than a single CPU core can handle
- Could just run multiple sequential instances
  - Yes, this is sometimes the answer
- Lots of CPU cores in modern systems, let's use them!
  - More resources means more performance

JavaOne ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Cores or Threads?
Apples or oranges

- Hardware has gotten surprisingly complicated
  - Multiprocessor ← increased density
  - Caching ← decreased latency
  - Multi-issue ← increased efficiency
  - Multi-core ← increased density
  - Virtual cores, Simultaneous multi-threading ← fight latency
- Threads are just software
  - Software abstraction for “run these instructions”
  - Threads run on a core, but are generally not attached to that core
  - Threads are frequently created and deleted

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Do we need threads?
Unnecessary clutter?

- Thread simplify modeling
  - Abstract the problem of scheduling cores
  - Alternative to queuing
- Threads enhance utilization
  - I/O wait and other latency
- Threads enhance “fairness”
  - Timeslice pre-emption means everybody gets to run
  - Resource hogs

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

These Modern Times

- Program flow is parallel and frequently concurrent
  - Each thread starts, some looping, end
  - May no longer be deterministic
- Program performance consists of
  - Everything that was important for serial
  - **TPS**
  - Throughput, latency, utilization

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Parallel or Concurrent?

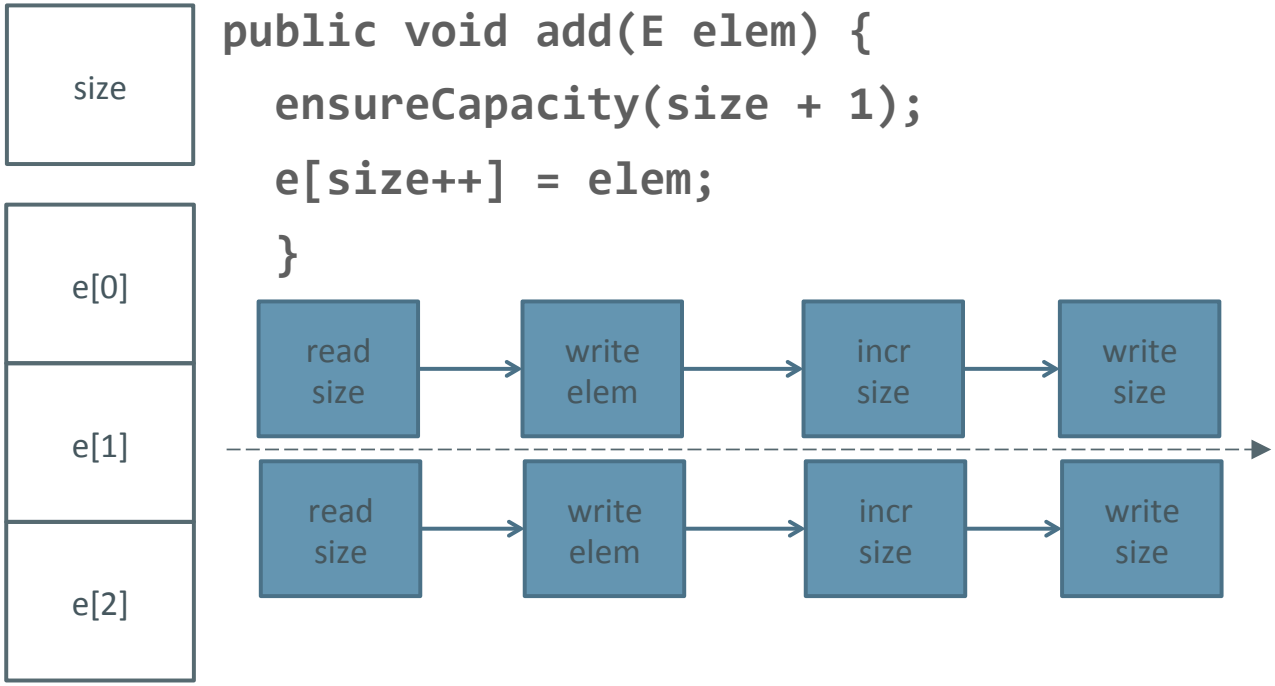
One lump or two?

- Sequential
  - A single thread performing a single task
  - No shared data. Blissfully isolated (almost entirely)
- Parallel
  - Multiple threads simultaneously performing multiple tasks
  - Read-only shared data. Blissfully isolated (mostly)
- Concurrent
  - Multiple threads collaborating on a single or multiple tasks
  - Mutable Shared data. Contention and coordination overhead for writes

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Add an element to ArrayList

Two threads is better than one

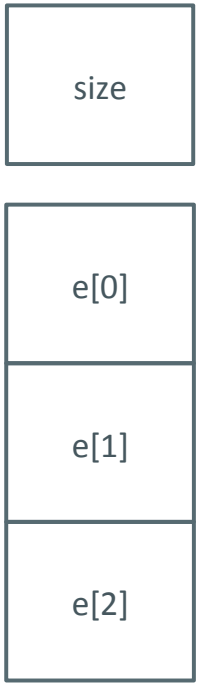


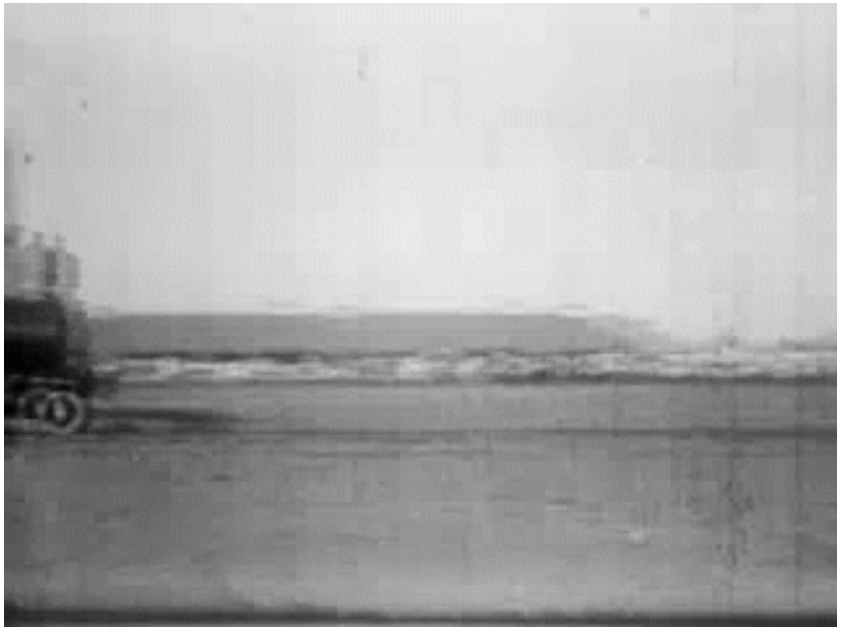
JavaOne ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Concurrency problem in detail

(not to scale)





public domain video via archive.org

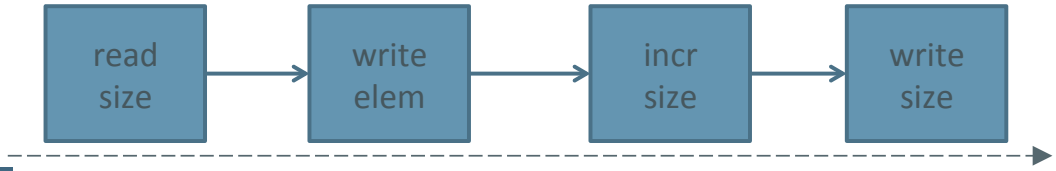
Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

`volatile` to the rescue?

Add some gasoline

```
volatile int size;
public void add(E elem) {
  ensureCapacity(size + 1);
  e[size++] = elem;
}
```

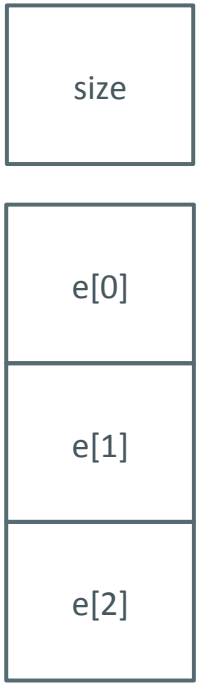
- **`volatile`** keyword ensures that value is not cached
  - This means no stale value seen by other threads
- Declaring **`size`** as **`volatile`** ensures reads & writes are consistent
- Doesn't fix our other problem
  - Multiple steps being done concurrent updates
- We need to have one thread updating at a time



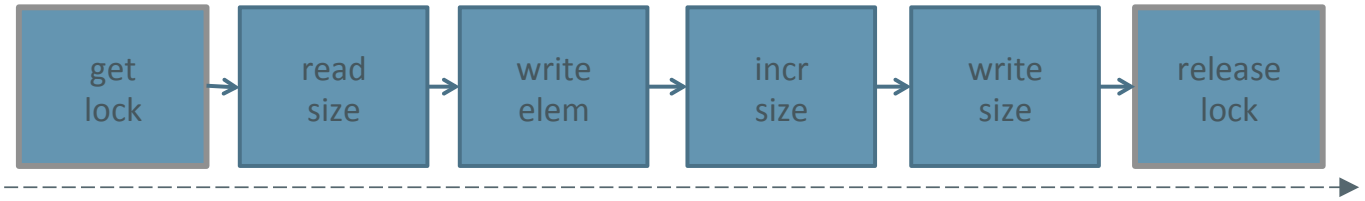
Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Add an element to ArrayList

We need some new steps




```
public synchronized void add(E elem) {
  ensureCapacity(size + 1);
  e[size++] = elem;
  }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Public Service Message

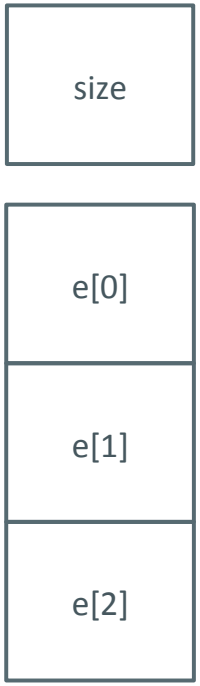
That's not a lock mate, ***this*** is a lock

photos CC BY-SA 3.0 from Wikipedia

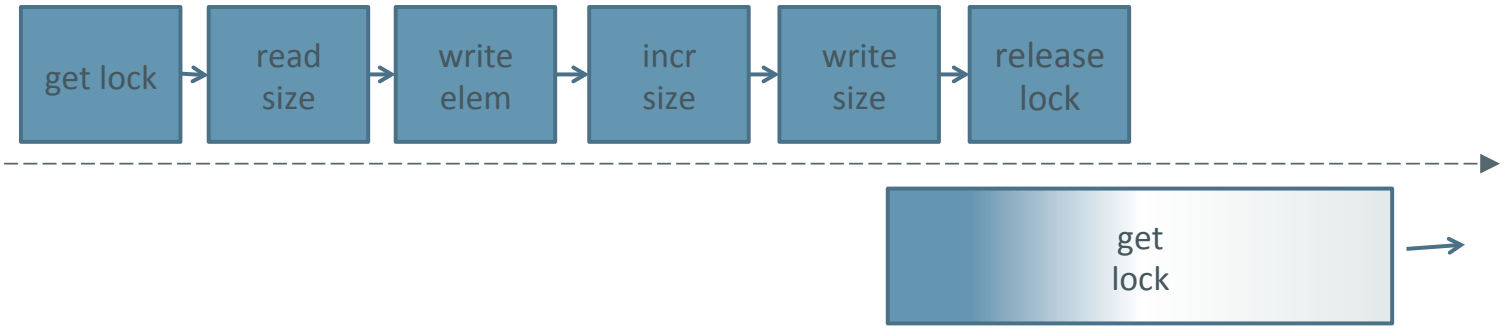
Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Add an element to ArrayList

One at a time please



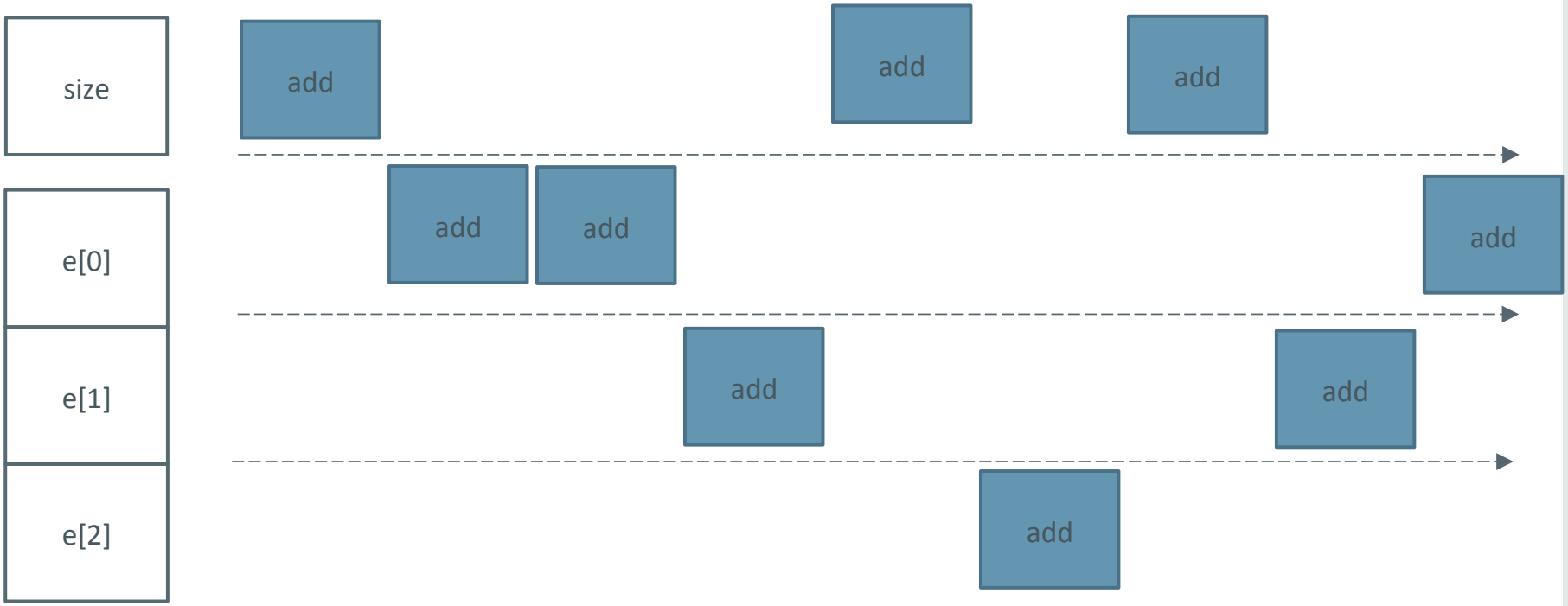

```
public synchronized void add(E elem) {
  ensureCapacity(size + 1);
  e[size++] = elem;
  }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Add an element to ArrayList

One at a time please



JavaOne ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Do we need `volatile` and `synchronized`?

I *really* want to use **`volatile`**

- Exiting **`synchronized`** block makes all writes visible
- Adding **`volatile`** to **`size`** is redundant for writes, slower for reads

```
public synchronized void clear() {
  for(int i = 0; i < size; i++) {
    e[i] = null;
  }
  size = 0;
}
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

How much slower is `synchronized`?

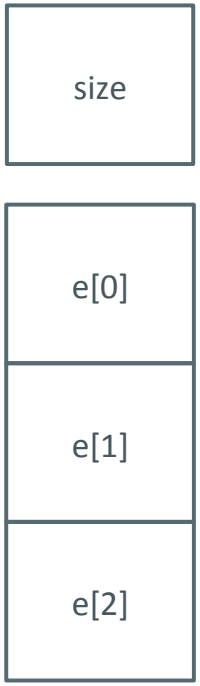
Not the end of the world

- It depends
- JVM can do lots of optimization. Some include:
  - General Optimizations
    - Lock coarsening – grouping actions on same lock
  - Uncontended Optimizations
    - Lock Elision – simple lock for first user
    - Biased locking – repeated locking by the same thread is optimized
  - Contented Optimization
    - Spin locking – hot waiting
    - Lazy Wait Queues – delay until second waiter.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Working with a Queue

Push me, pull you



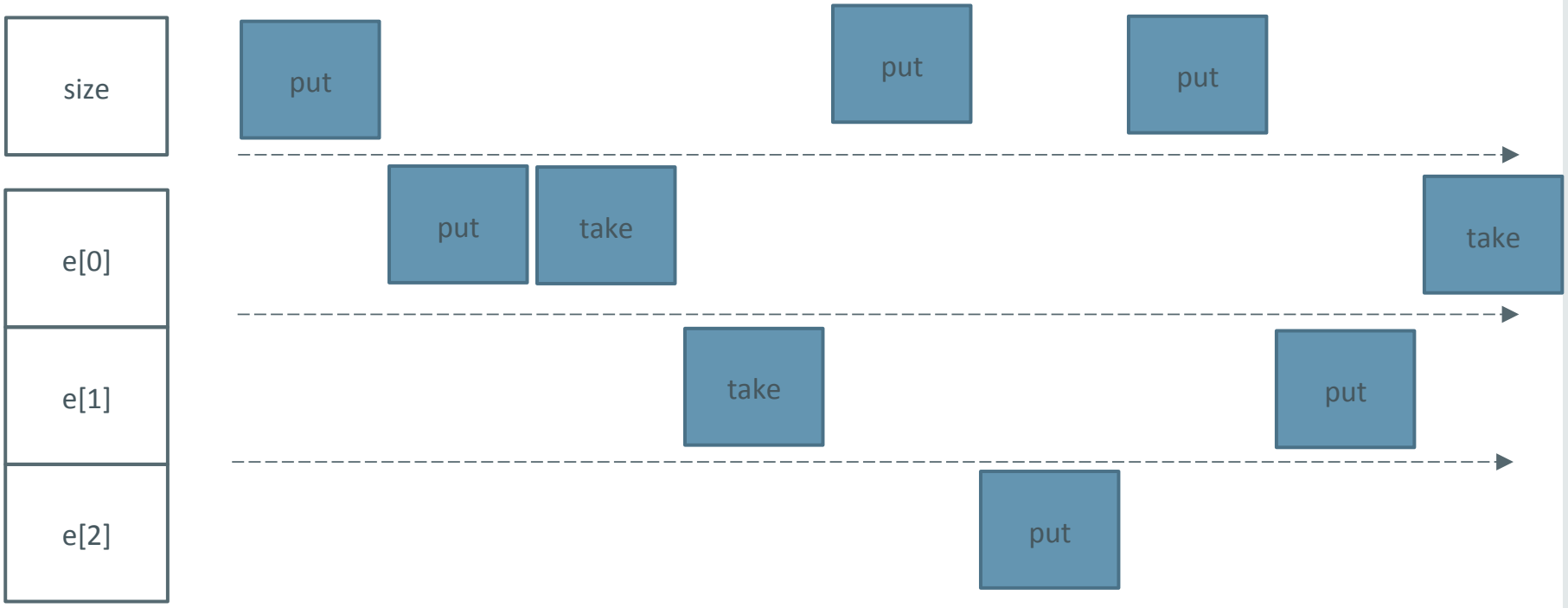

```
public synchronized void put(E elem) {
    while (size == e.length)
        wait();
    add(elem);
}
```

```
public synchronized E take() {
    while (size == 0)
        wait();
    return remove(0);
}
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Working with a Queue

One at a time please



Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

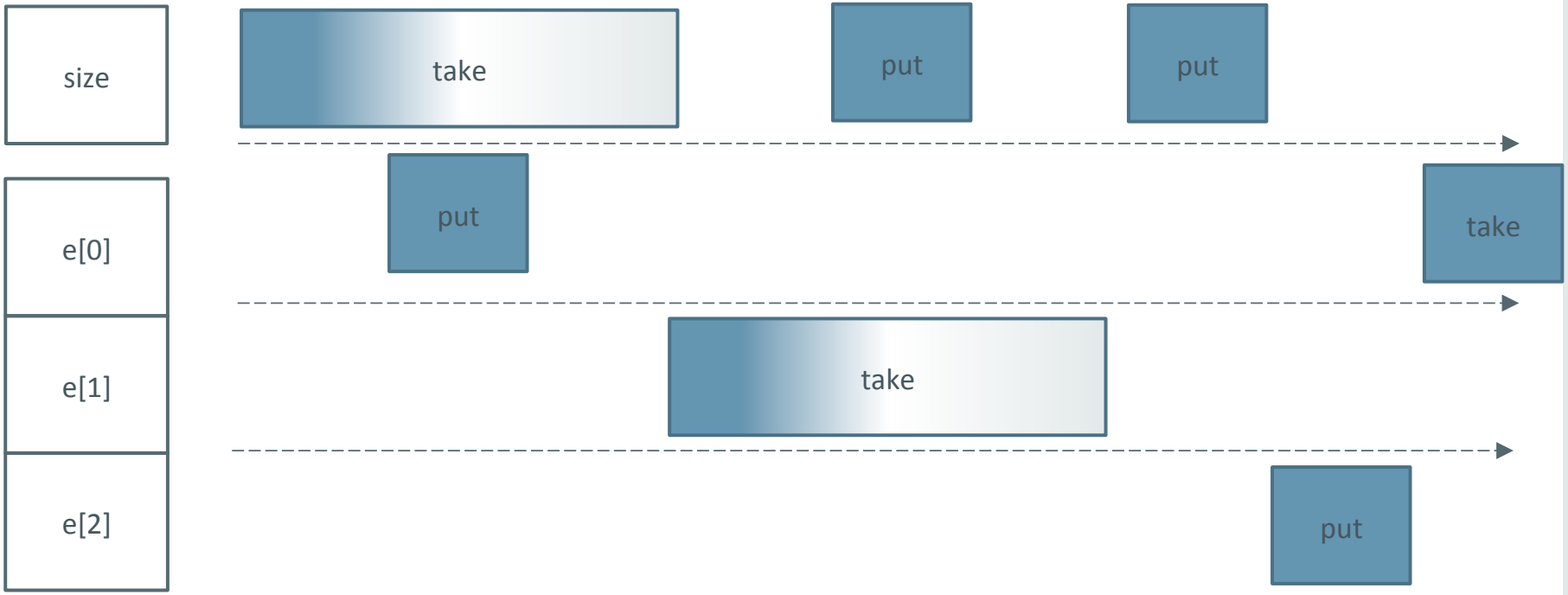
Two kinds of waiting
One at a time please

- Contention
  - Waiting to acquire the lock
  - The overhead you were warned about
  - Important to measure this
- Starvation
  - Waiting for free space/element
  - Entirely natural
  - Can be caused by contention
  - Also important to measure

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Working with a Queue

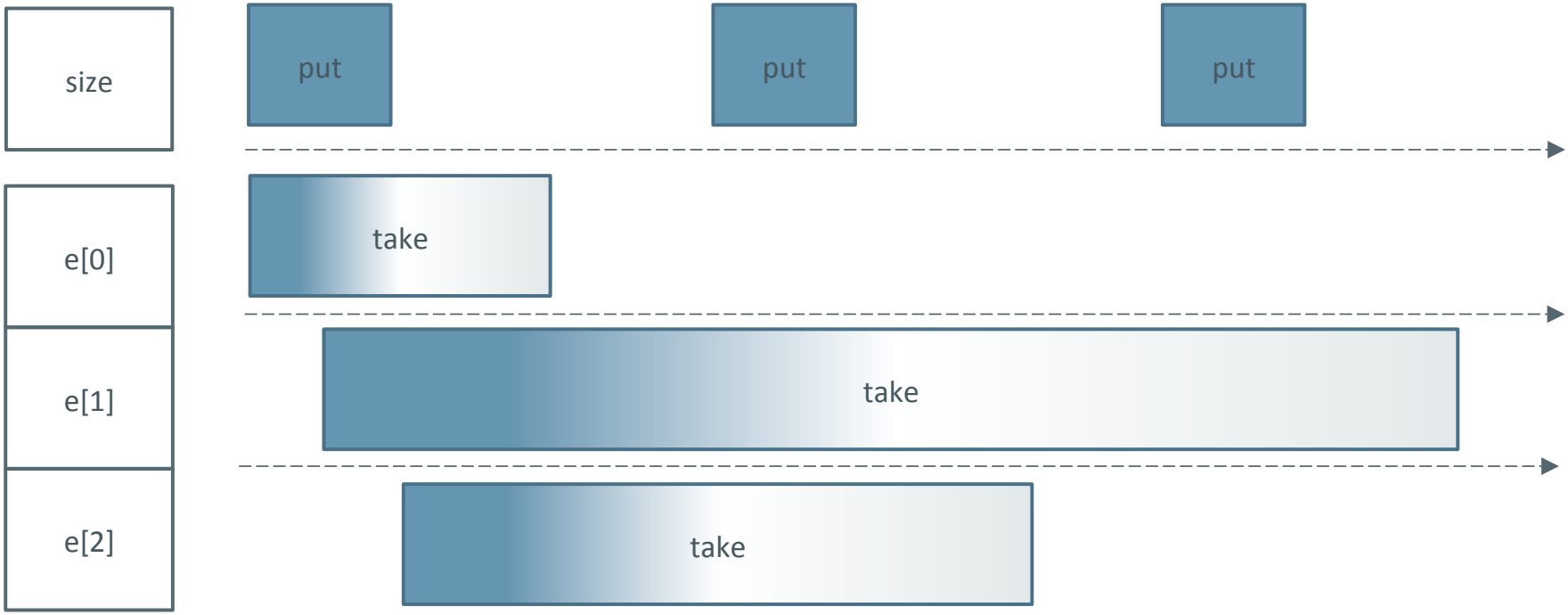
Higher demand than expected



Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Working with a Queue

Take turns please

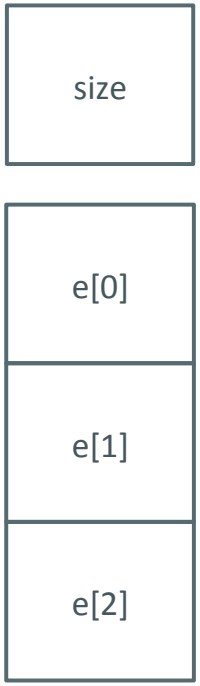


JavaOne ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Working with a Queue

Taking matters into our own hands




```
public E take() {
    lock.lock();
    try {
       while (size== 0)
           signal.await();
       E result = remove(0);
       signal.signalAll();
       return result;
    } finally { lock.unlock(); }
}
```

JavaOne ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Lock Options

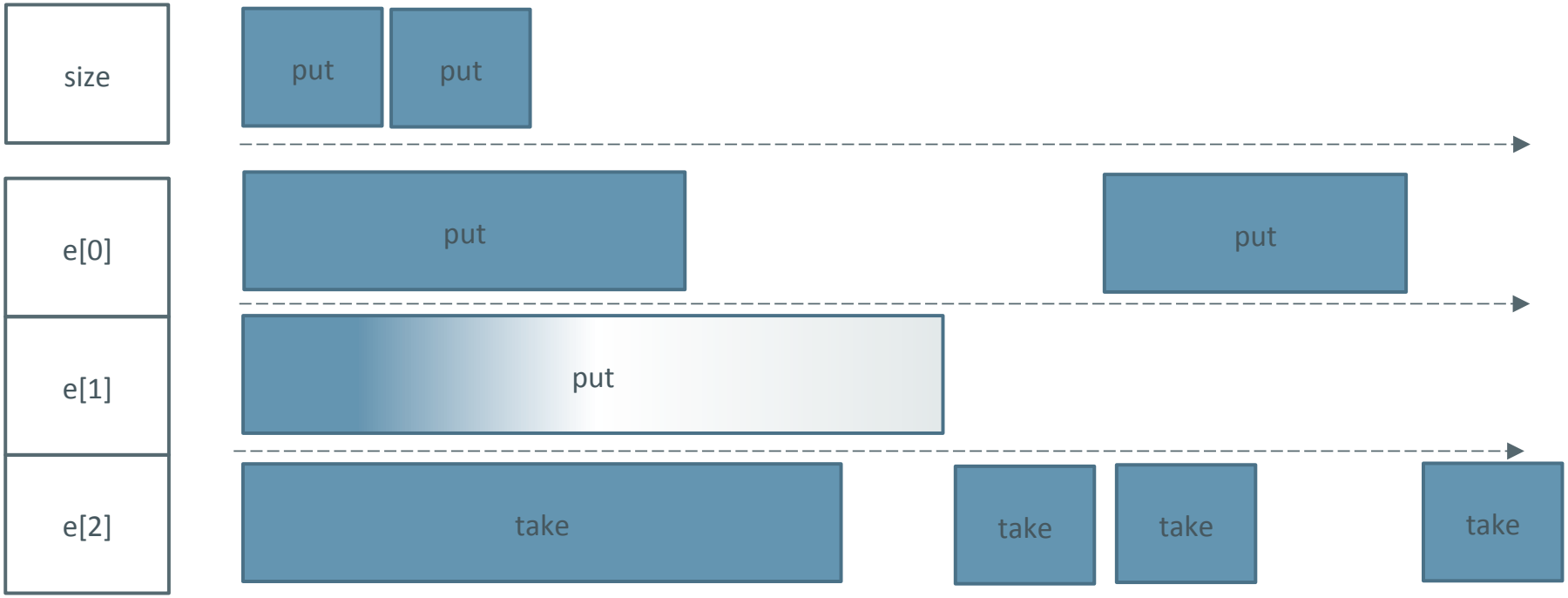
The price of fairness

- **`Lock lock = new ReentrantLock();`**
  - Result is very similar to synchronized
  - OS may schedule next thread in any order
- **`Lock lock = new ReentrantLock(true);`**
  - Longest waiter (generally) goes next
- Fairness is not free
  - Does fairness matter more than throughput?

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Working with a Queue

Trying to be fair can bring everyone down



JavaOne ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Condition Signalling

The thread who cried wolf!

- One signal, multiple conditions
  - Space available!
  - Element available!
- Not every waiter is looking for same condition
- Right now we have to wake everyone for EVERY signal
- Let's separate these conditions
- Producers wait for notFull, signal notEmpty
- Consumers wait for notEmpty and signal notFull

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Lock Conditions

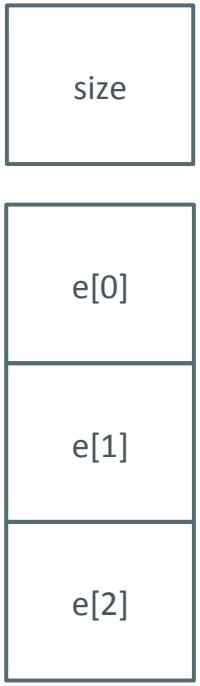
What condition my condition is in

- `Lock lock = new ReentrantLock();`
- `Condition notFull = lock.newCondition();`
- `Condition notEmpty = lock.newCondition();`

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Working with a Queue

Alert the media

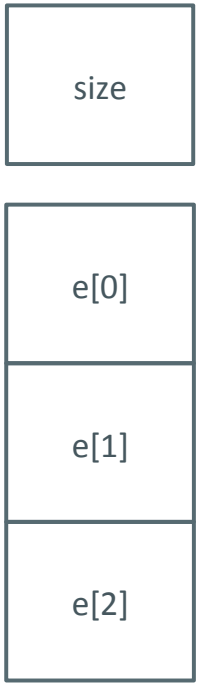



```
public E take() throws InterruptedException {
    lock.lock();
    try {
       while (size== 0)
           notEmpty.await();
       E result = remove(0);
       notFull.signal();
       return result;
    } finally { lock.unlock(); }
}
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Working with a Queue

Taking matters into our own hands

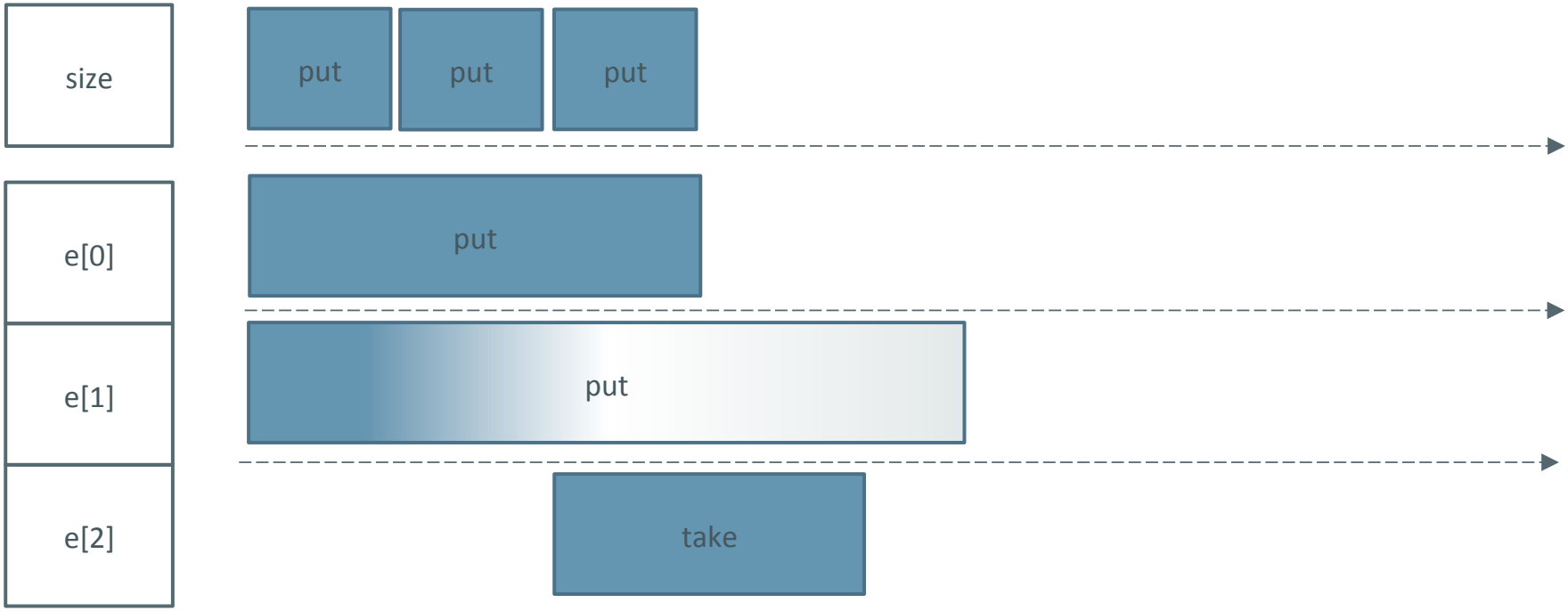



```
public void put(E e) throws InterruptedException {
    lock.lock();
    try {
        while (size == e.length)
            notFull.await();
        add(e);
        notEmpty.signal();
    } finally { lock.unlock(); }
}
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Working with a Queue

Eliminated from Contention



Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Lock alternatives?
Travelling light

- A lock can be too heavyweight for
  - Low-moderate write contention
  - Read-mostly application
  - Guarding a simple operation
- Alternative – Atomic Compare And Swap
  - Reading value is same as unsynchronized volatile read
  - Write is an atomic conditional replacement
    - Compare current value against some value
      - Replace current value if matched

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Going Atomic

Insanity is expecting a different result

```
final AtomicInteger size = new AtomicInteger();
public void incrementAndGet() {
 int curr, next;
  do {
    curr = size.get(); next = curr + 1;
  } while(!size.compareAndSet(curr, next));
  return next;
}
```

JavaOne ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

I was promised `volatile`!

Hoisted with one's own petard (look it up, it **really** applies)

- Unsynchronized read does save time
- **`volatile`** ensures that JIT does not hoist (cache) value
- Write is going to be slower than AtomicInteger
- Under contention will be even worse

```
private volatile int size;
public int get() { return size; }
public synchronized int incrementAndGet() {
    return size++;
}
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Where to Begin?

I thought we were almost done?

- Immutable
- Safe-racey
- `Volatile`
- `java.util.concurrent.atomic.*`
- `Synchronized`
- `java.util.concurrent.Lock`
- …

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Immutable

Always safe

- Data never observed to change
- Use **`final`**, unmodifiable wrappers, gentleman's agreement
- Design objects for immutability
  - Even more important in Java 10(?) with Project Valhalla
- Exercise: make a `Point` class with fields `x` and `y`

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Safe-racey

Play nice and nobody gets hurt

- Often used as a lazy initialization pattern

```
String cachedToString;
public String toString() {
   String result = cachedToString;
   if(result == null) {
      result = cachedToString = makeToString();
   }

   return result;
}
```

- Might call `makeToString()` on multiple threads, but that's OK
- Better than making `toString()` synchronized
- Exercise : Convert an existing class to use this pattern

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

volatile

Playing with gasoline

- Readers want current value
- Writers don't care about current value
- Check-then-act is not possible*
- Can be combined with **`synchronized`**
- Exercise: Benchmark earlier **`synchronized`**/**`volatile`** counter vs **AtomicInteger**
-

(*) You can do safe-racey like prior example.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

java.util.concurrent.atomic

Harness the power of the atom

- Readers want current value (like `volatile`)
- Writers either don't care about current value or want to do something based on it
- Limited check-then-act is possible, Compare and Swap (CAS)
- How much work is reasonable between initial read and CAS?
- Exercise: Convert existing synchronized counter to Atomic

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

`synchronized`

Safe and steady

- Necessary for multi-value state without tearing
- Readers want coherent view of state
- Writers must update all values comprising state atomically
- There are perils in holding more than one lock at a time
- You have little control over use of `synchronized` by other classes
- Exercise: Replace synchronized with one of these other techniques

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

`java.util.concurrent.Lock`

Bells and Whistles

- Needed for multiple conditions
- Needed for **`tryLock`**
- Needed for fair locking
- If you don't need it don't use it
  - More JVM optimization around **`synchronized`**
  - **`java.util.concurrent`** converting some **`Lock`** -> **`synchronized`** in Java 9
- Exercise: Benchmark performance vs synchronized with/without fairness

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

…

You don't start here, you end up here

- **`Semaphore`**
  - Because why build your own?
- **Phaser, CountdownLatch, CyclicBarrier, Exchanger, SynchronousQueue, Disruptor, …**
  - Designed to solve problems (scalability, performance) with simpler approaches
- Exercise: Read the JavaDoc and think if these would have solved any previous performance problems

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Something simpler? Faster?

Travelling light

- All this concurrency stuff is…
  - rocket science, black magic, brain surgery, voodoo???
- Less coordination, more processing!
  - Concurrent: multiple threads doing a single task
  - Parallel: multiple threads doing multiple tasks
- Sequential/Concurrent -> Parallel : divide up the task!
  - Not applicable to all problems (indivisible, realtime)
- Division doesn't even have to be complete
  - Recursive decomposition is actually more efficient

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Parallel is what we want!

Full Streams ahead

- For parallel operation we need
  - Immutable input
  - Decomposable problem
  - Coordination to divide/combine sub-tasks
- Java 7 provides Fork/Join for the extremely macho/desperate
- Java 8 provides new Streams library to make this easy

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Java 8 Streams/Lambda

Full Streams ahead

- Library handles coordination
  - Our code focuses on “what” not “how”
  - Scalability, decomposition, aggregation all handled
- Opt-in parallelism – agree to constraints
  - Immutability, non-interference

```
double highestGrade = students.parallelStream()
    .filter(s -> s.isEnrolled())
    .mapToDouble(s -> s.getGrade())
    .max().orElse(0.0);
```

JavaOne ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

Resources

- Java 8
  - Streams/Lambda
- Java Flight Recorder/Mission Control (OracleJDK)
  - Measurement and diagnostics
- Brian Goetz ***Java Concurrency in Practice***
  - Indispensible guide to Java concurrency
- Doug ***Lea Concurrent Programming in Java, 2nd Edition***
  - The gory details
- Charlie Hunt ***Java Performance***
  - **The** Java performance book

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2015 Mike Duigou

JavaOne™

ORACLE®