Concurrency Factors

Many commonplace tasks can be parallelized

The existence of multi-core machines

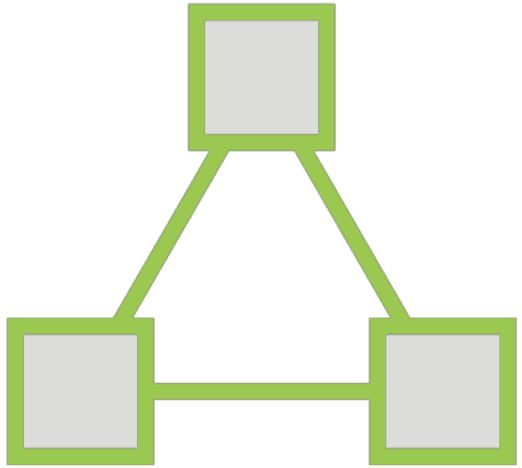
Java Concurrency Support

Thread

Initiating and managing threaded execution

Synchronized

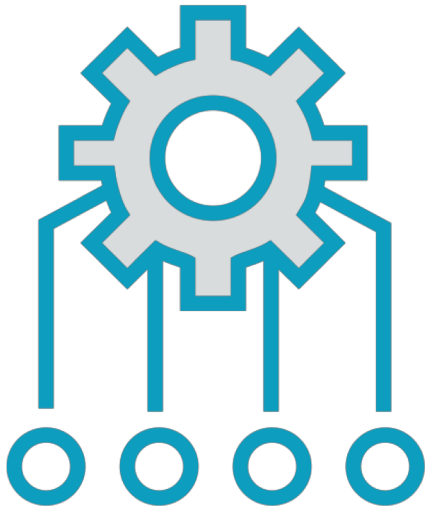
Obtaining and releasing locks



Wait/Notify/NotifyAll

Thread coordination

Java Concurrency Support

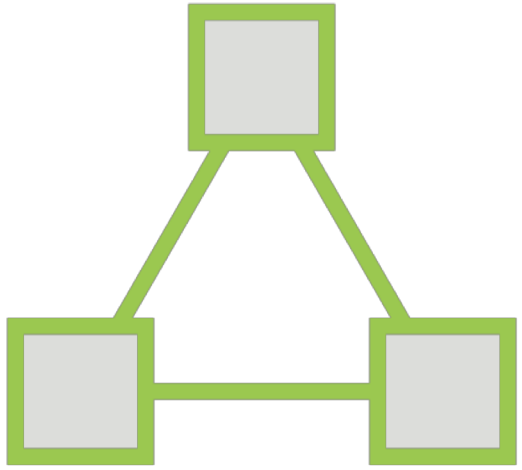


Executor Service

Initiating and managing threaded execution

Synchronized

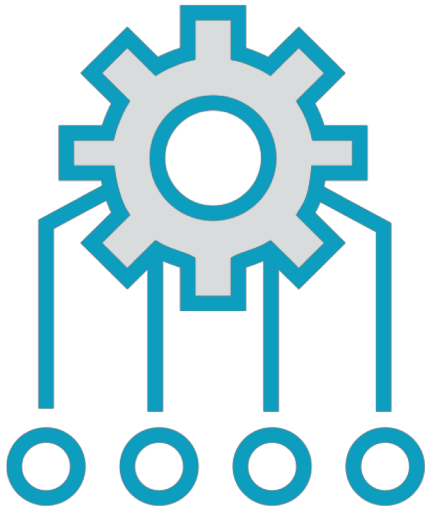
Obtaining and releasing locks



Wait/Notify/NotifyAll

Thread coordination

Java Concurrency Support

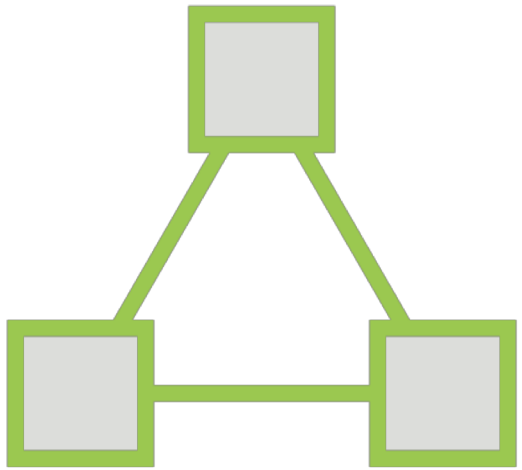


Executor Service

Initiating and managing threaded execution

Lock

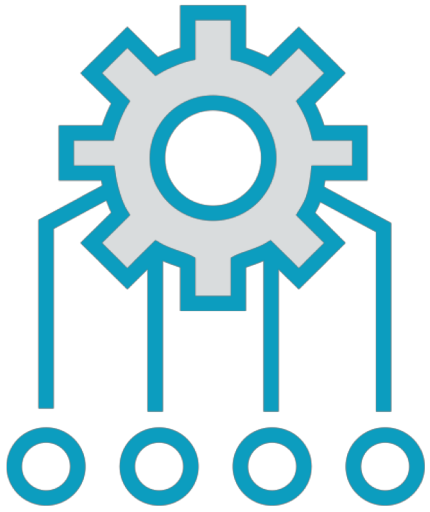
Obtaining and releasing locks



Wait/Notify/NotifyAll

Thread coordination

Java Concurrency Support



Executor Service

Initiating and managing threaded execution

Lock

Obtaining and releasing locks

Condition/Semaphore /CountDownLatch/ CyclicBarrier

Thread coordination

Concurrency Challenges

Even with the new tools, we can still write inefficient concurrent code

Challenges:

- Selecting the correct concurrency tool
- Knowing how to use and configure the tools
- Knowing how to avoid certain pitfalls

Overview

ThreadPoolExecutor Optimization

The ForkJoinPool

Reducing Lock Contention

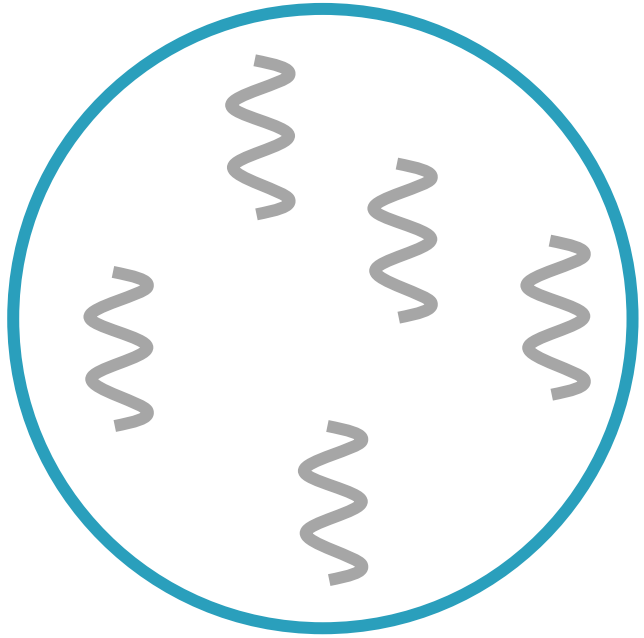
Atomic Variables and Concurrent Collections

Avoiding Synchronization

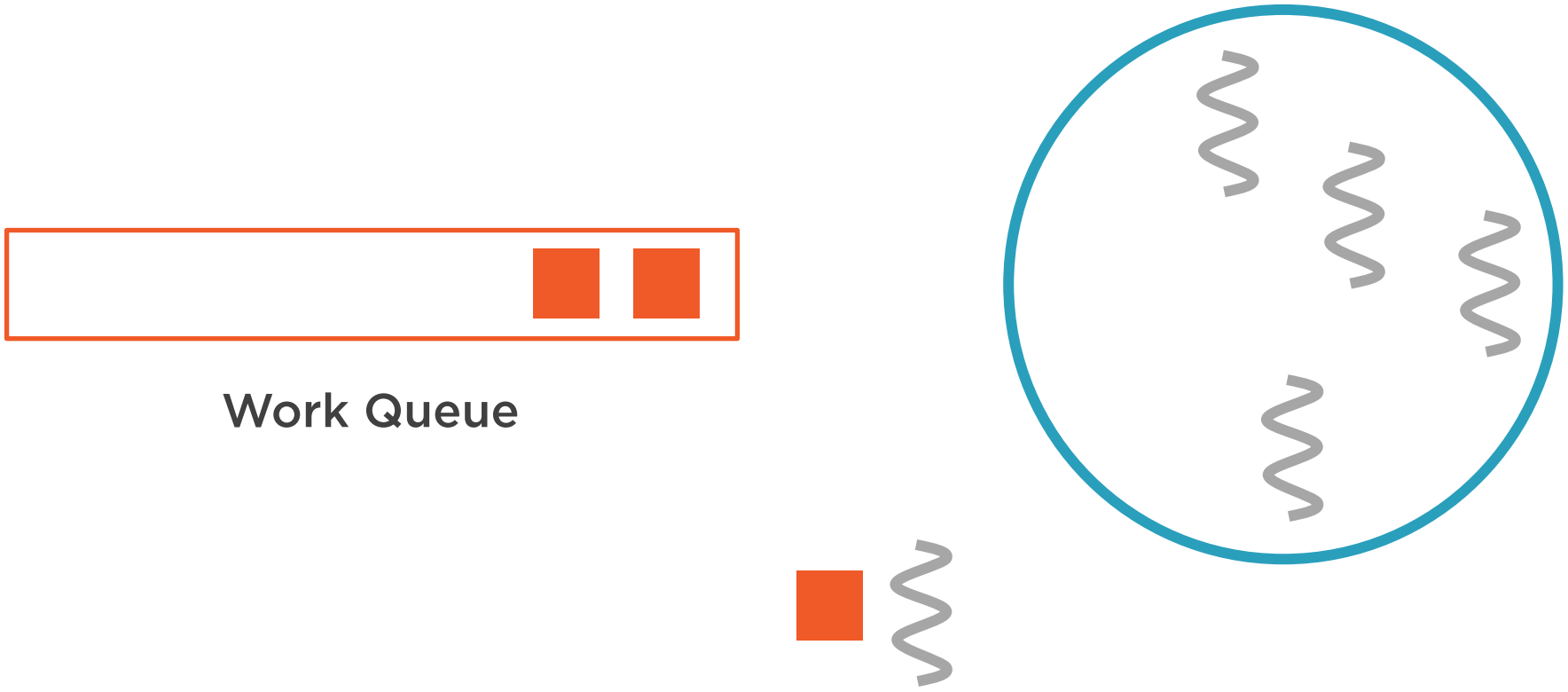
ThreadPoolExecutor Optimization

Thread Pools

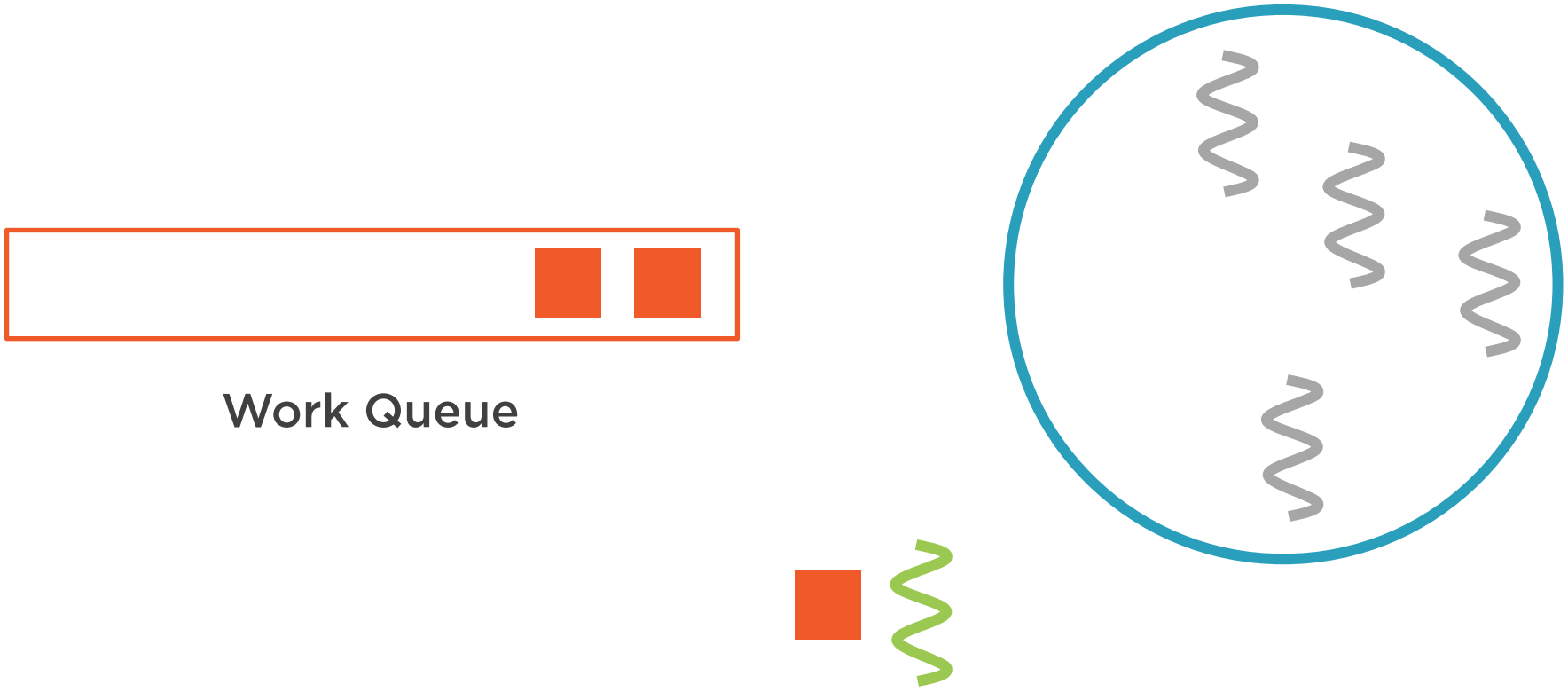
Work Queue



Thread Pools

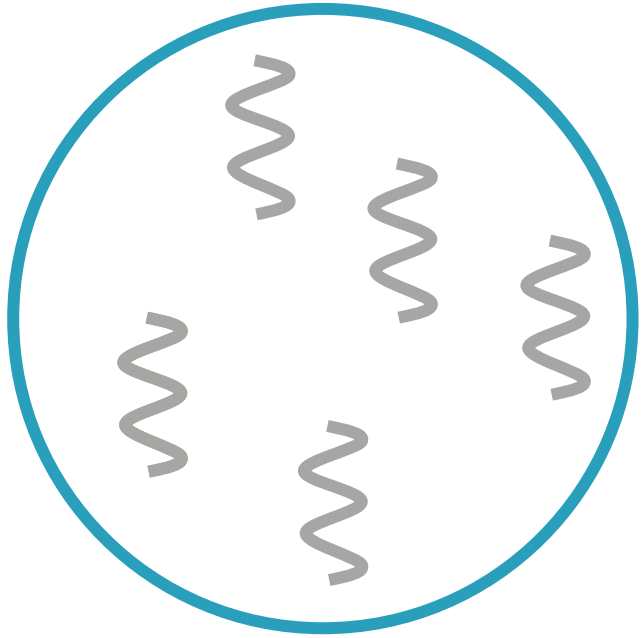


Thread Pools



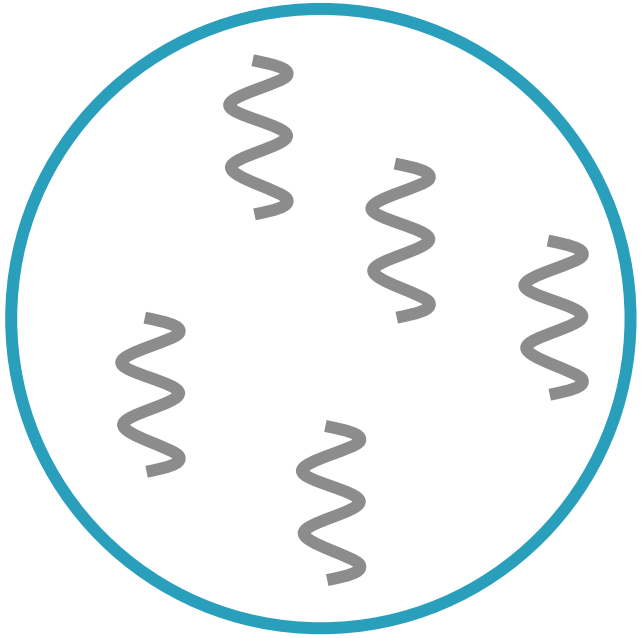
Thread Pools

Work Queue



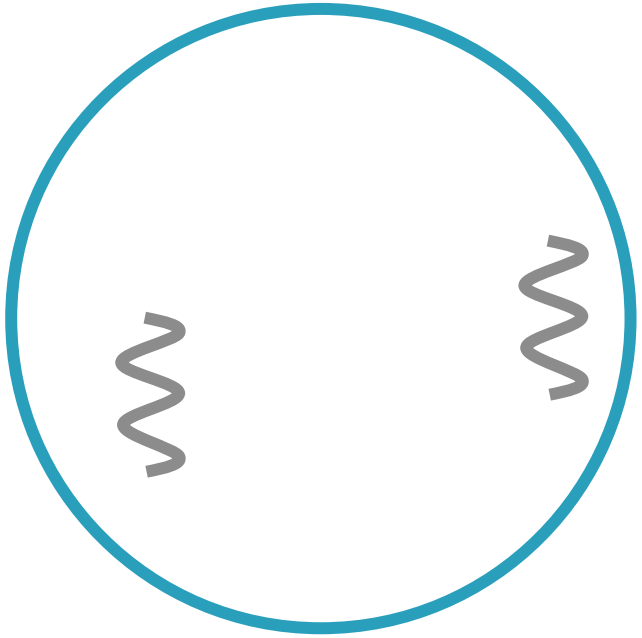
Thread Pools

Work Queue

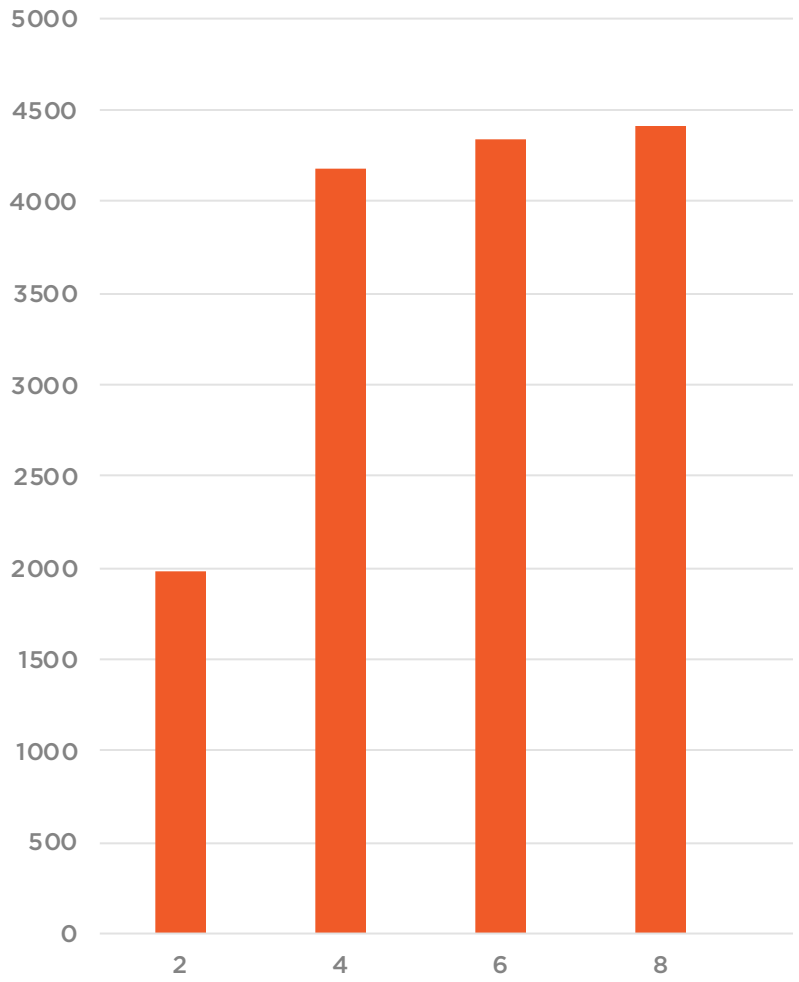


Thread Pools

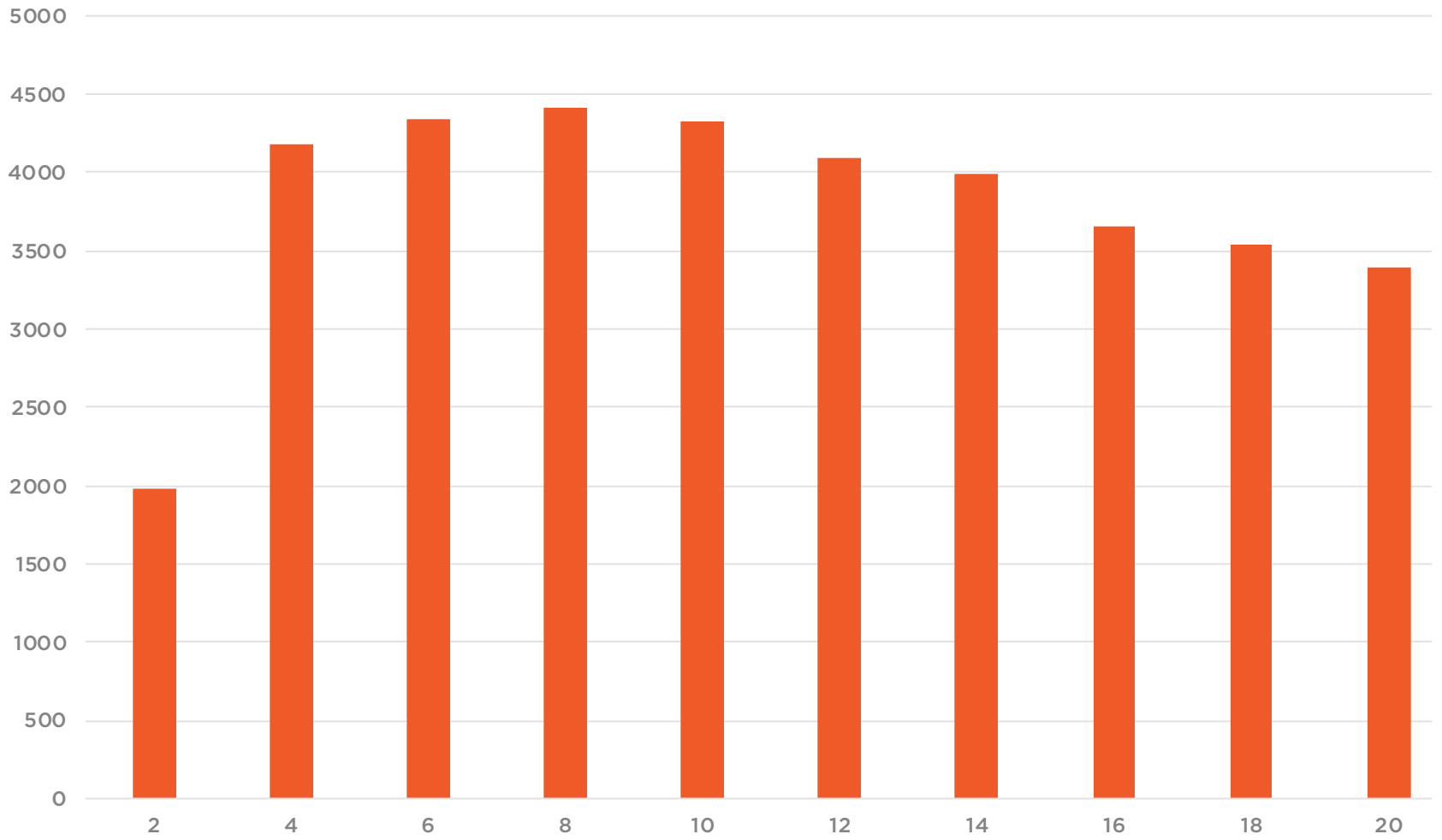
Work Queue



Thread Pool Size



Thread Pool Size



Drawbacks of Having Too Many Threads

The CPU has to dedicate more time to scheduling threads

Context switches require expensive context unloading and loading operations

Caches may be invalidated and data locality may be lost

Tuning the size of the thread pool is not an exact science.

Thread Pool Sizing Approaches

Model-based approach

Experimental approach

Thread Pool Sizing Approaches

Model-based approach

Experimental approach

$$N_{threads} = N_{cpu} * U_{cpu} * (1 + W/C)$$

N_{cpu} = number of CPU cores

U_{cpu} = desired CPU utilization, $0 < U_{cpu} <= 1$

W/C = ratio of wait time to compute time

$$N_{threads} = N_{cpu} * U_{cpu} * (1 + 0)$$

N_{cpu} = number of CPU cores

U_{cpu} = desired CPU utilization, $0 < U_{cpu} <= 1$

W/C = ratio of wait time to compute time

$$N_{threads} = N_{cpu} * 1 * (1 + 0)$$

N_{cpu} = number of CPU cores

U_{cpu} = desired CPU utilization, $0 < U_{cpu} <= 1$

W/C = ratio of wait time to compute time

$$N_{threads} = N_{cpu} * 1 * 1$$

N_{cpu} = number of CPU cores

U_{cpu} = desired CPU utilization, $0 < U_{cpu} <= 1$

W/C = ratio of wait time to compute time

$$N_{threads} = N_{cpu} * 0.6 * 1$$

N_{cpu} = number of CPU cores

U_{cpu} = desired CPU utilization, $0 < U_{cpu} <= 1$

W/C = ratio of wait time to compute time

$$N_{threads} = N_{cpu} * 1 * (1 + 75/25)$$

N_{cpu} = number of CPU cores

U_{cpu} = desired CPU utilization, $0 < U_{cpu} <= 1$

W/C = ratio of wait time to compute time

$$N_{threads} = N_{cpu} * 1 * (1 + 3)$$

N_{cpu} = number of CPU cores

U_{cpu} = desired CPU utilization, $0 < U_{cpu} <= 1$

W/C = ratio of wait time to compute time

Variations of the Model

Start with the model, then tweak the results

Tweak the formula itself and use different coefficients

Use a different model

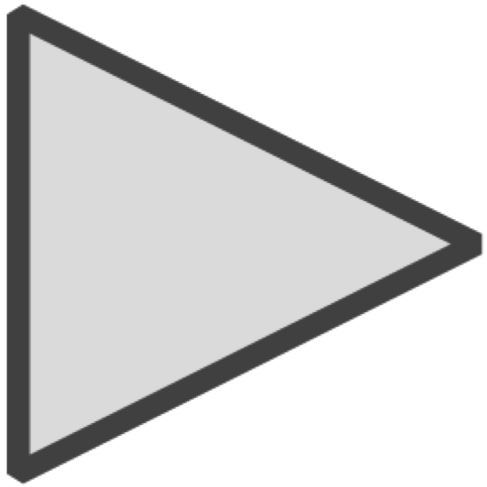
Other Factors Affecting Thread Pool Sizing

Memory

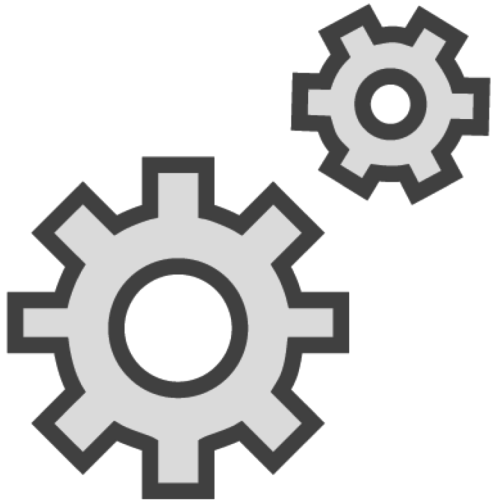
File / socket handles

Database connections

Experimental Approach to Thread Pool Sizing



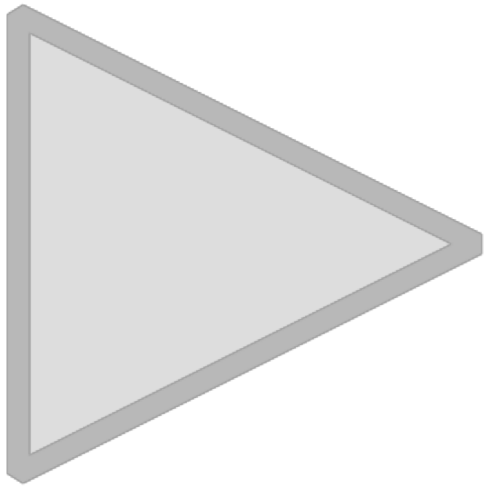
Send traffic to the application



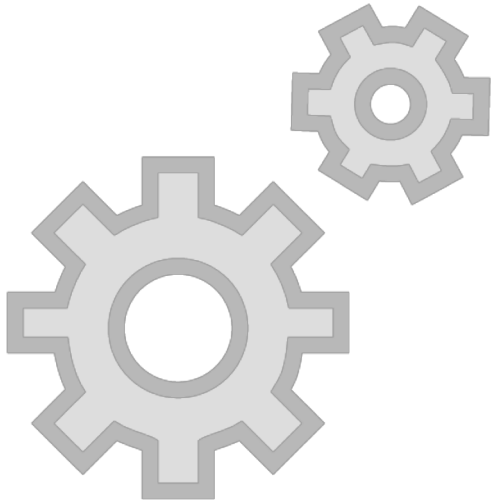
Vary the thread pool sizes

Monitor system performance and thread characteristics

Experimental Approach to Thread Pool Sizing



Send traffic to the application



Vary the thread pool sizes

Monitor system performance and thread characteristics

Thread Pool Sizing Best Practices

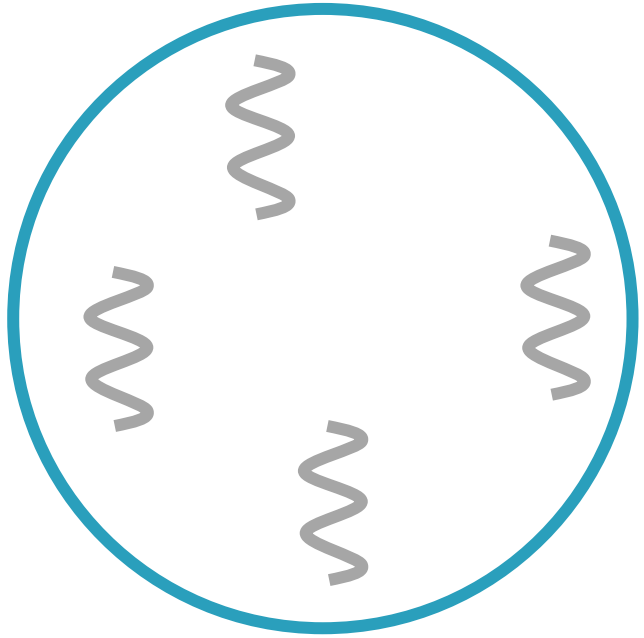
Instead of hard-coding thread pool sizes, make it configurable or calculate it dynamically

If plausible, split CPU and I/O work into separate thread pools that can be configured individually

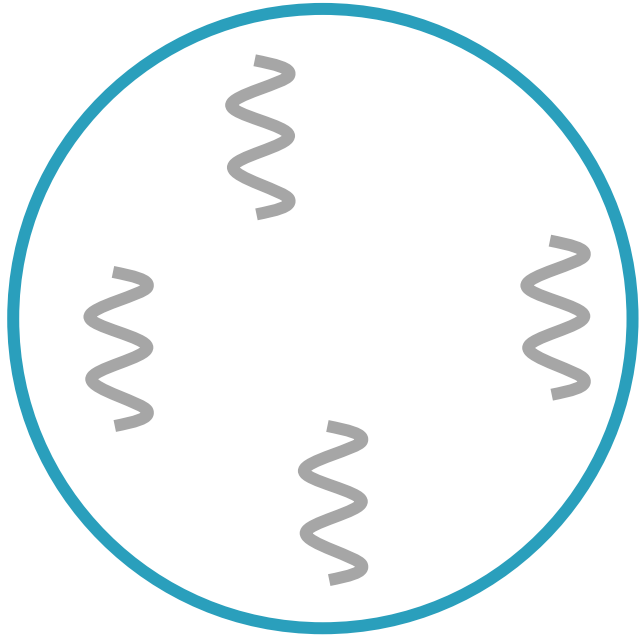
Focus first on tuning core thread pools before auxiliary ones

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue)
```

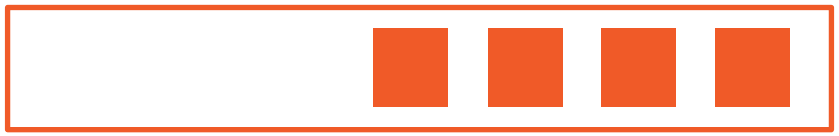
ThreadPoolExecutor



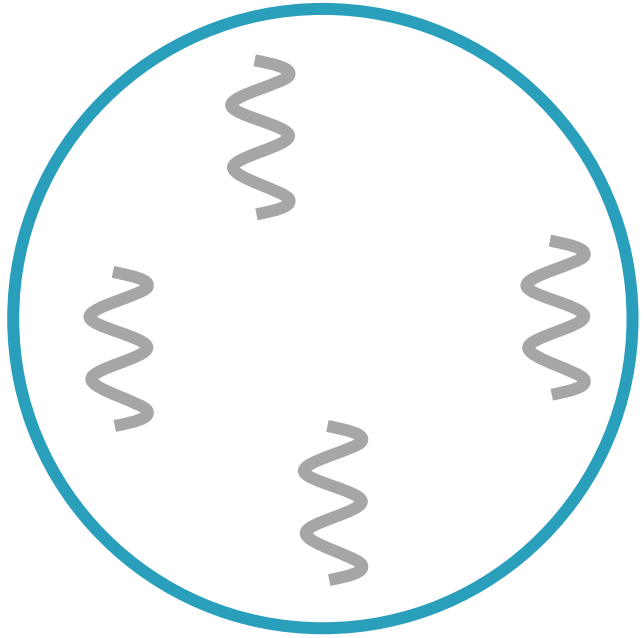
ThreadPoolExecutor



ThreadPoolExecutor

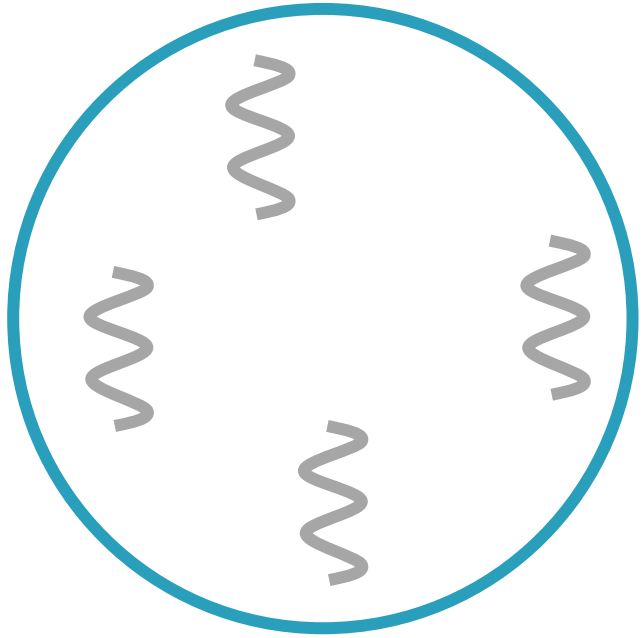


Unbounded / Bounded / Synchronous



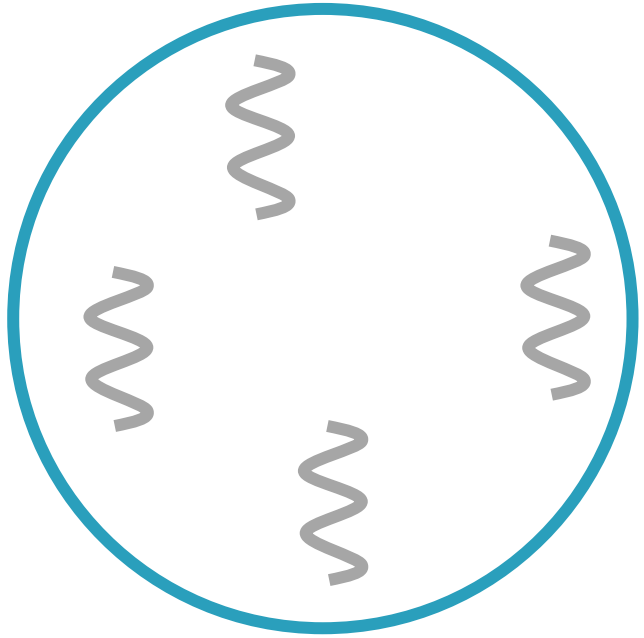
ThreadPoolExecutor

Unbounded



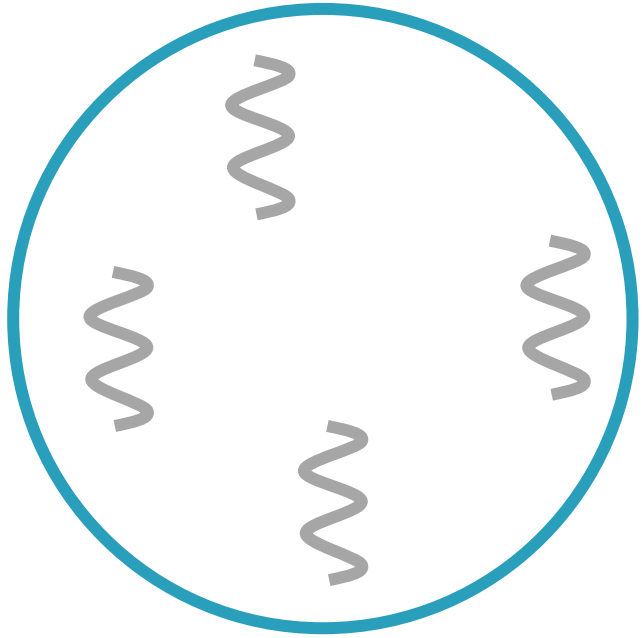
ThreadPoolExecutor

Unbounded



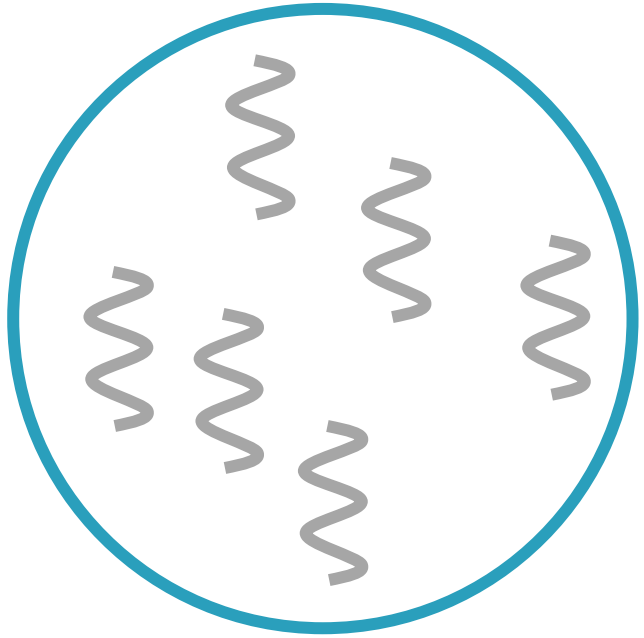
ThreadPoolExecutor

Bounded



ThreadPoolExecutor

Bounded



Server Application Best Practices

Determine optimal pool size

Use a bounded queue with a limit just large enough to handle a burst of requests

Set:

corePoolSize = maxPoolSize = `optimalSize`

Or:

corePoolSize = `optimalSizeMin`

maxPoolSize = `optimalSizeMax`

Work Queue Implementation

LinkedBlockingQueue, ArrayBlockingQueue

Tasks are started in the order they were received

PriorityBlockingQueue

Tasks are started in order of priority

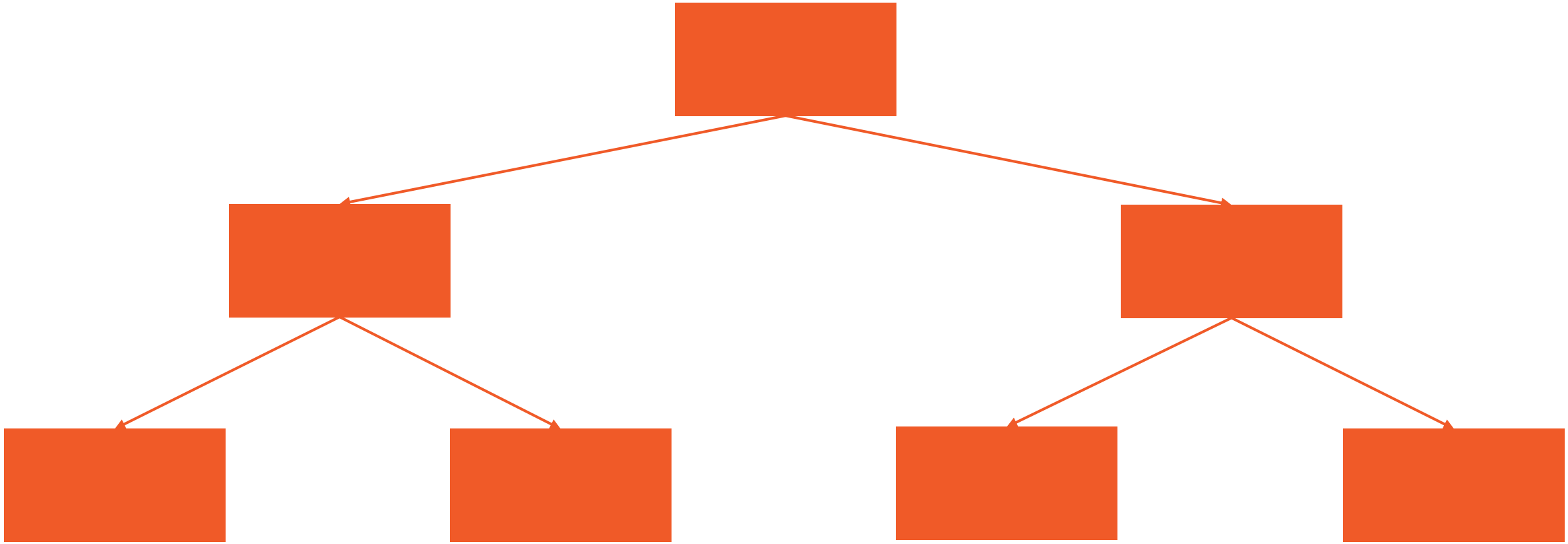
The ForkJoinPool

Recursive Model

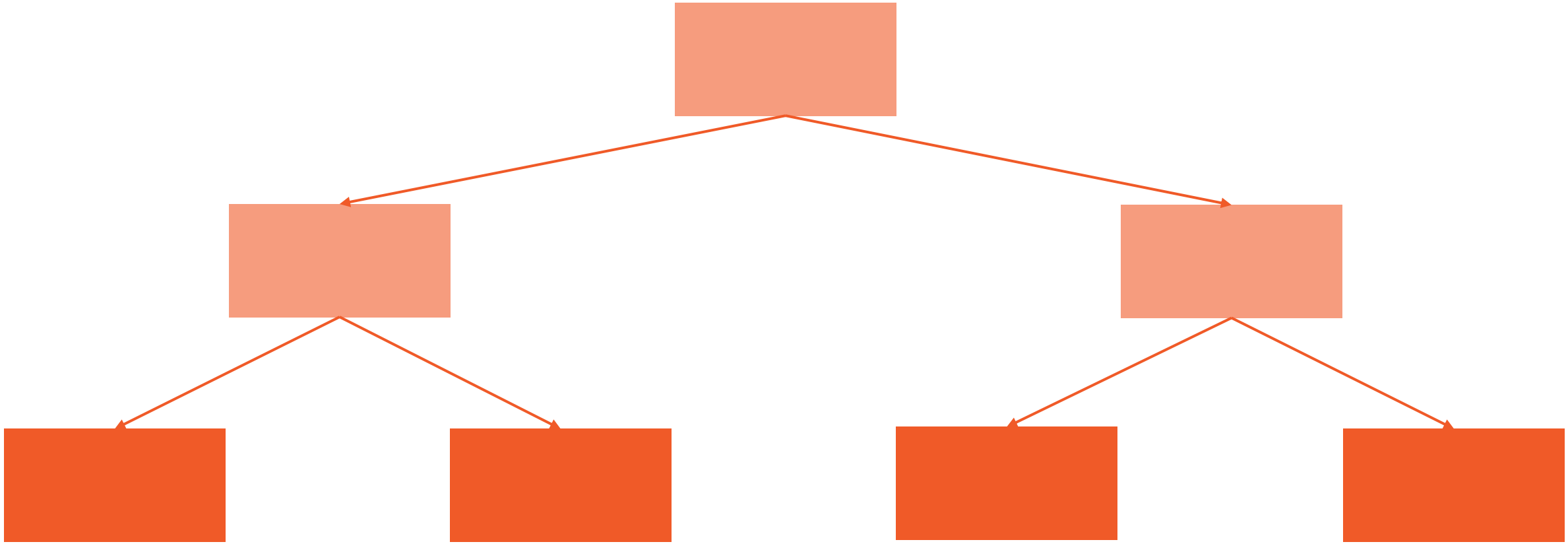
Recursive Model

Recursive Model

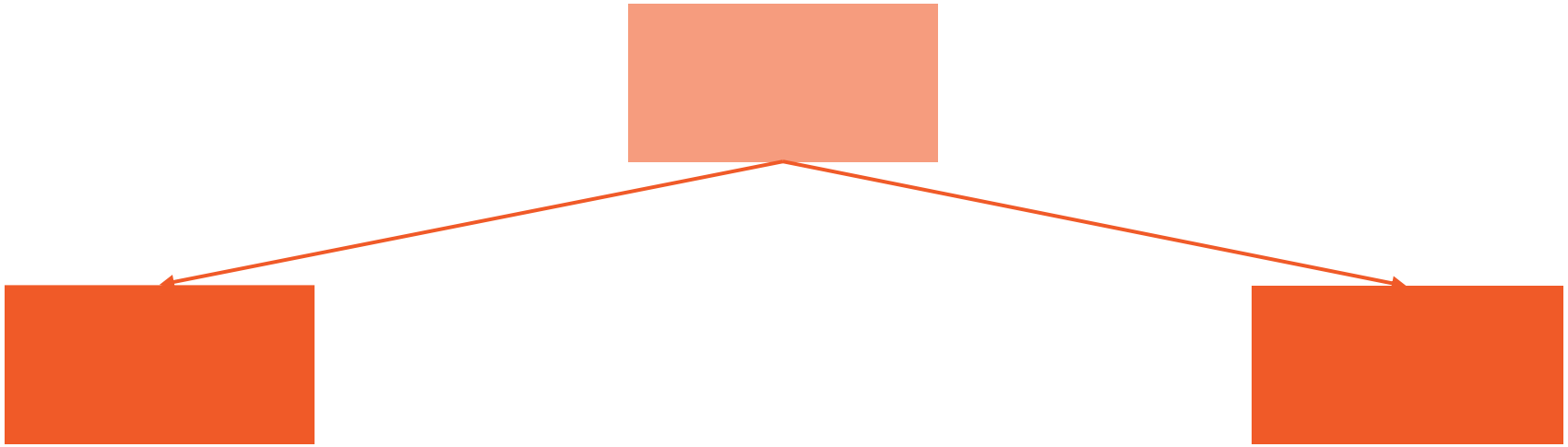
Recursive Model



Recursive Model



Recursive Model



Recursive Model

```
public class MyRecursiveTask extends RecursiveTask<Long> {
  private Task task;
  public MyRecursiveTask(Task task) {
    this.task = task;
  }
  protected Long compute() {
    if (this.task.size() > threshold) {
      Pair<RecursiveTask> subTasks = splitTask();
      subTasks.getLeft().fork();
      subTasks.getRight().fork();
      ...
    } else { ... }
  }
}
```

```
public class MyRecursiveTask extends RecursiveTask<Long> {
  private Task task;
  public MyRecursiveTask(Task task) {
    this.task = task;
  }
  protected Long compute() {
    if (this.task.size() > threshold) {
      Pair<RecursiveTask> subTasks = splitTask();
      subTasks.getLeft().fork();
      subTasks.getRight().fork();
      ...
    } else { ... }
  }
}
```

```
public class MyRecursiveTask extends RecursiveTask<Long> {
  ...
  protected Long compute() {
    if (this.task.size() > threshold) {
      Pair<RecursiveTask> subTasks = splitTask();
      subTasks.getLeft().fork();
      subTasks.getRight().fork();
      long result = 0;
      result += subTasks.getLeft().join();
      result += subTasks.getRight().join();
      return result;
    } else { ... }
  }
}
```

```
public class MyRecursiveTask extends RecursiveTask<Long> {
  ...
  protected Long compute() {
    if (this.task.size() > threshold) {
      Pair<RecursiveTask> subTasks = splitTask();
      ...
    } else {
      System.out.println(“Executing task myself”);
      return executeTask(task);
    }
  }
}
```

```
public class MyRecursiveTask extends RecursiveTask<Long> {
  ...
  protected void compute() {
    if (this.task.size() > threshold) {
      Pair<RecursiveTask> subTasks = splitTask();
      subTasks.getLeft().fork();
      subTasks.getRight().fork();
      long result = 0;
      result += subTasks.getLeft().join();
      result += subTasks.getRight().join();
      return result;
    } else { ... }
  }
}
```

```
public class MyRecursiveTask extends RecursiveAction {
  ...
  protected void compute() {
    if (this.task.size() > threshold) {
      Pair<RecursiveTask> subTasks = splitTask();
      subTasks.getLeft().fork();
      subTasks.getRight().fork();
      long result = 0;
      result += subTasks.getLeft().join();
      result += subTasks.getRight().join();
      return result;
    } else { ... }
  }
}
```

```
public class MyRecursiveTask extends RecursiveAction {
  ...
  protected void compute() {
    if (this.task.size() > threshold) {
      Pair<RecursiveTask> subTasks = splitTask();
      subTasks.getLeft().fork();
      subTasks.getRight().fork();




    } else { ... }
  }
}
```

```
MyRecursiveTask myRecurTask = new MyRecursiveTask(largeTask);
ForkJoinPool forkJoinPool = new ForkJoinPool(8);
forkJoinPool.invoke(myRecurTask);
```

```
MyRecursiveTask myRecurTask = new MyRecursiveTask(largeTask);
ForkJoinPool forkJoinPool = new ForkJoinPool(8);
forkJoinPool.invoke(myRecurTask);
```

```
MyRecursiveTask myRecurTask = new MyRecursiveTask(largeTask);
ForkJoinPool forkJoinPool = new ForkJoinPool(8);
forkJoinPool.invoke(myRecurTask);
```

```
public class UserRequest extends RecursiveTask<Result> {
  protected Result compute() {
    if (this.numItems < 100) {
      // directly execute UserRequest
    } else {
      // split into smaller UserRequest subtasks
    }
  }
}
```

```
public class UserRequest extends RecursiveTask<Result> {
  protected Result compute() {
    if (this.numItems < 100) {
      // directly execute UserRequest
    } else {
      // split into smaller UserRequest subtasks
    }
  }
}
```

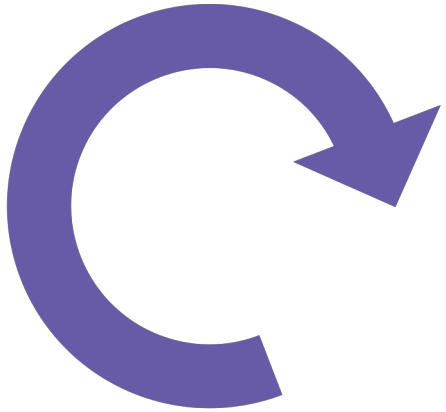
ForkJoinPool

Can handle a large amount of tasks with very few threads

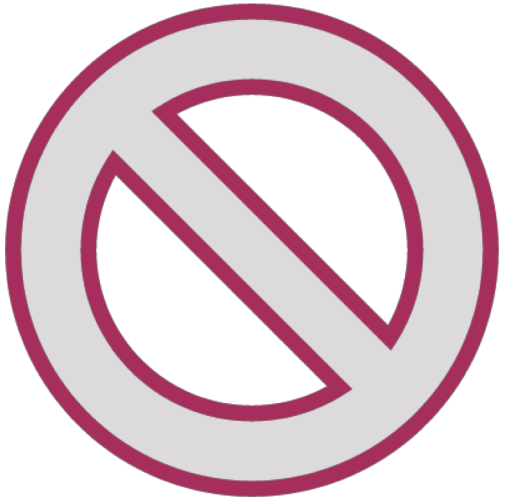
Used by the Java 8 Parallel Streams feature under the hood

Reducing Lock Contention

Blocking Mechanisms

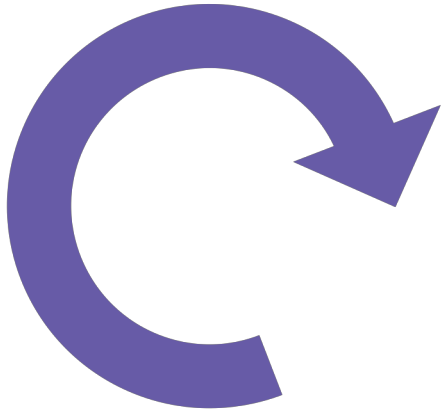


Spin-Wait



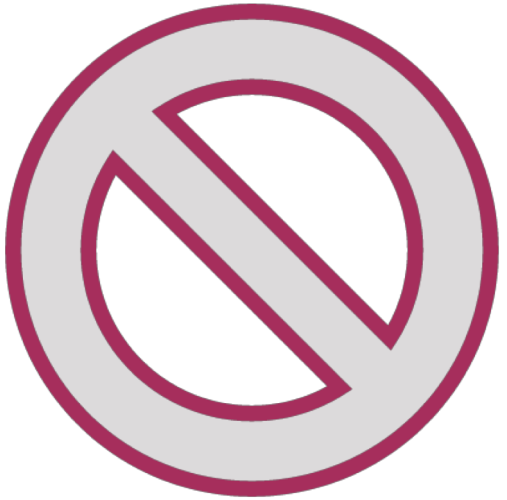
OS Suspension

Blocking Mechanisms



Spin-Wait

More efficient for short waits



OS Suspension

Preferable for longer waits

JVM Blocking Implementations

Spin-wait + suspend after X number of spins

Spin-wait or suspend, depending on profiling data

Default to suspension

Locks are not bad, lock contention is.

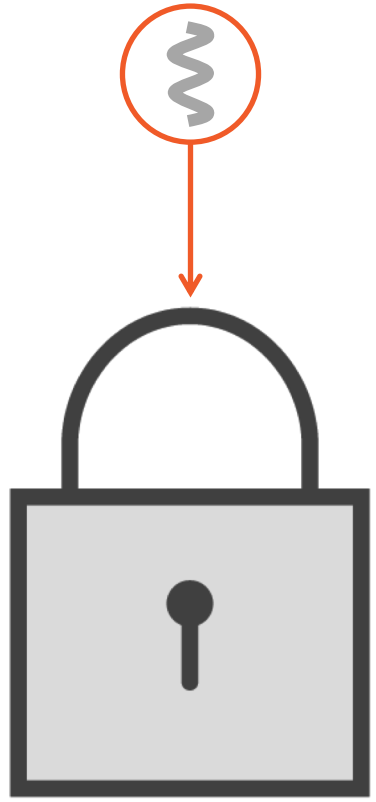
Performance Impacts of Lock Contention

Results in serial execution and reduces the scalability of the application

Leads to context switches which hurt performance

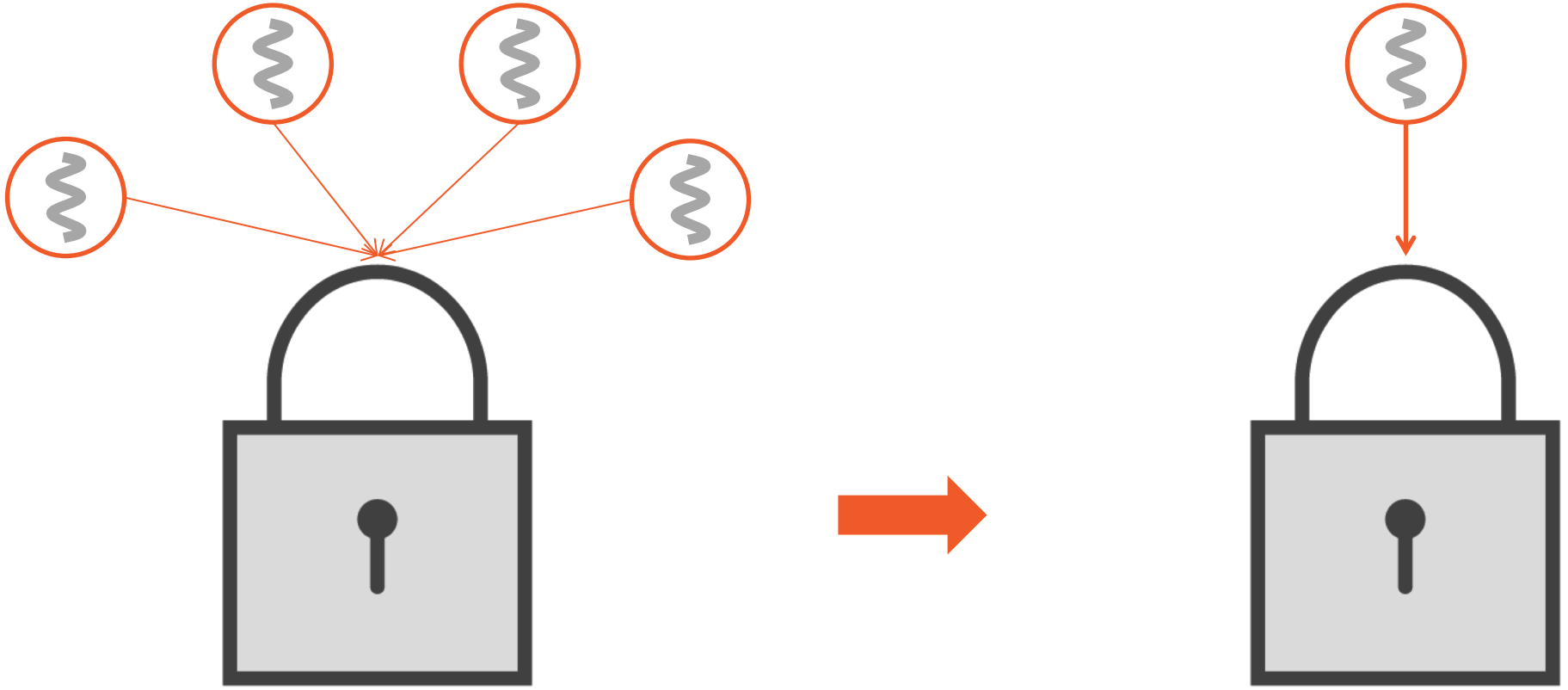
Lock Classification

High Contention



Low Contention

Lock Classification



High Contention

Low Contention

First make it right, then make it fast, if it's not already fast enough

Reducing Lock Contention

- Reduce the duration for which the lock is held
- Reduce the frequency at which the lock is requested
- Replace locks with alternative synchronization methods that allow for greater concurrency

Reducing the Duration for Which a Lock Is Held

Reduce the size of the critical block to the point where only the shared data access is protected

```
public class AggregateScore {
  private static final double MAX_SCORE_PER_SCORER = 100;
  private double score;

  public void synchronized updateScore(Scorer scorer) {
    if (scorer.isApplicable(this)) {
      double scorerScore = scorer.calculateScore();
      this.score += scorerScore < MAX_SCORE_PER_SCORER ?
         scorerScore : MAX_SCORE_PER_SCORER;
    }
  }
}
```

```
public class AggregateScore {
  private static final double MAX_SCORE_PER_SCORER = 100;
  private double score;

  public void synchronized updateScore(Scorer scorer) {
    if (scorer.isApplicable(this)) {
      double scorerScore = scorer.calculateScore();
      this.score += scorerScore < MAX_SCORE_PER_SCORER ?
         scorerScore : MAX_SCORE_PER_SCORER;
    }
  }
}
```

```
public class AggregateScore {
  private static final double MAX_SCORE_PER_SCORER = 100;
  private double score;

  public void synchronized updateScore(Scorer scorer) {
    if (scorer.isApplicable(this)) {
      double scorerScore = scorer.calculateScore();
      this.score += scorerScore < MAX_SCORE_PER_SCORER ?
         scorerScore : MAX_SCORE_PER_SCORER;
    }
  }
}
```

```
public class AggregateScore {
  private static final double MAX_SCORE_PER_SCORER = 100;
  private double score;

  public void updateScore(Scorer scorer) {
    if (scorer.isApplicable(this)) {
      double scorerScore = scorer.calculateScore();
      synchronized(this) {
        this.score += scorerScore < MAX_SCORE_PER_SCORER ?
            scorerScore : MAX_SCORE_PER_SCORER; }
    }
  }
}
```

Reducing the Demand for a Lock

Lock splitting:

- Takes a single lock that is guarding different independent shared state variables and splits it into multiple locks for each variable

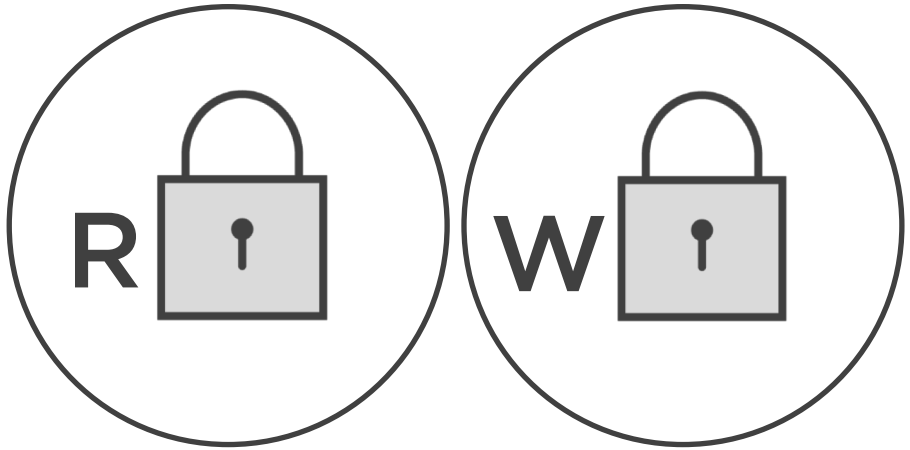
```
public class ServerStatistics {
  private long numActiveUsers;
  private long numDBQueries;

  public void synchronized incrementNumActiveUsers(int increment) {
    numActiveUsers += increment;
  }
  public void synchronized incrementNumDBQueries(int increment) {
    numDBQueries += increment;
  }
  public long synchronized getNumActiveUsers { ... }
  public long synchronized getNumDBQueries { ... }
}
```

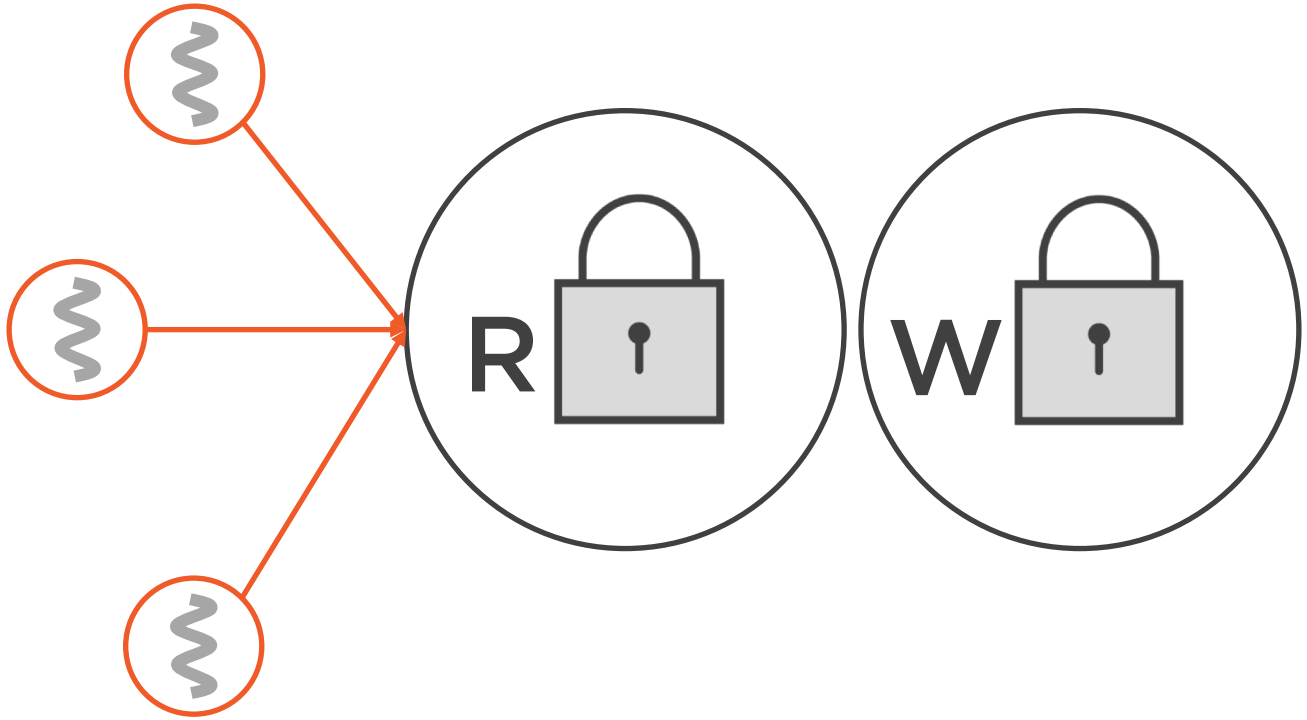
```
public class ServerStatistics {
  private long numActiveUsers;
  private long numDBQueries;

  public void synchronized incrementNumActiveUsers(int increment) {
    numActiveUsers += increment;
  }
  public void synchronized incrementNumDBQueries(int increment) {
    numDBQueries += increment;
  }
  public long synchronized getNumActiveUsers { ... }
  public long synchronized getNumDBQueries { ... }
}
```

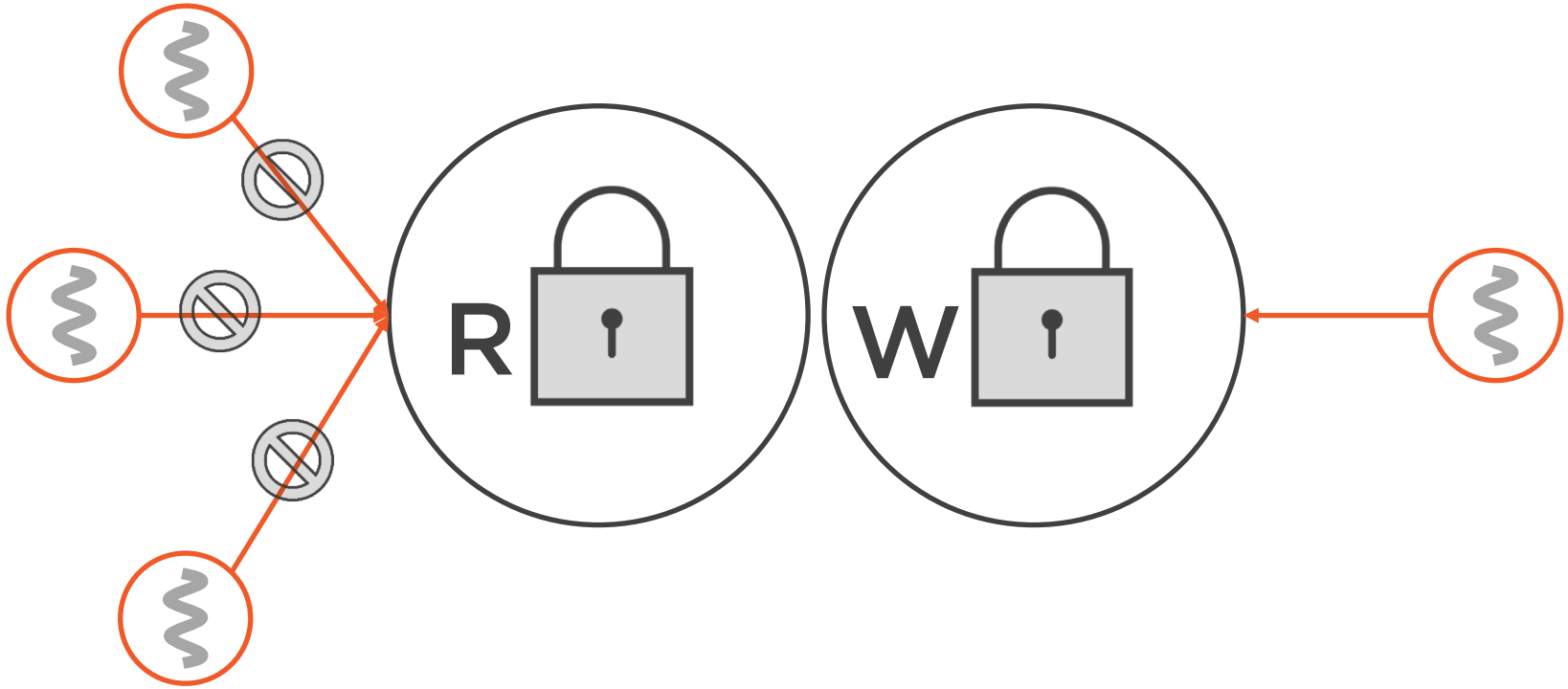
```
public class ServerStatistics {
  private long numActiveUsers;
  private long numDBQueries;
  private Object activeUsersLock = new Object();
  private Object dbQueriesLock = new Object();

  public void incrementNumActiveUsers(int increment) {
    synchronized (activeUsersLock) { numActiveUsers += increment; }
  }
  public void incrementNumDBQueries(int increment) {
    synchronized (dbQueriesLock) { numDBQueries += increment; }
  }
  ...
}
```

ReadWriteLock

Ideal when; a collection or data structure is written to infrequently but reads happen at a high frequency


R
W

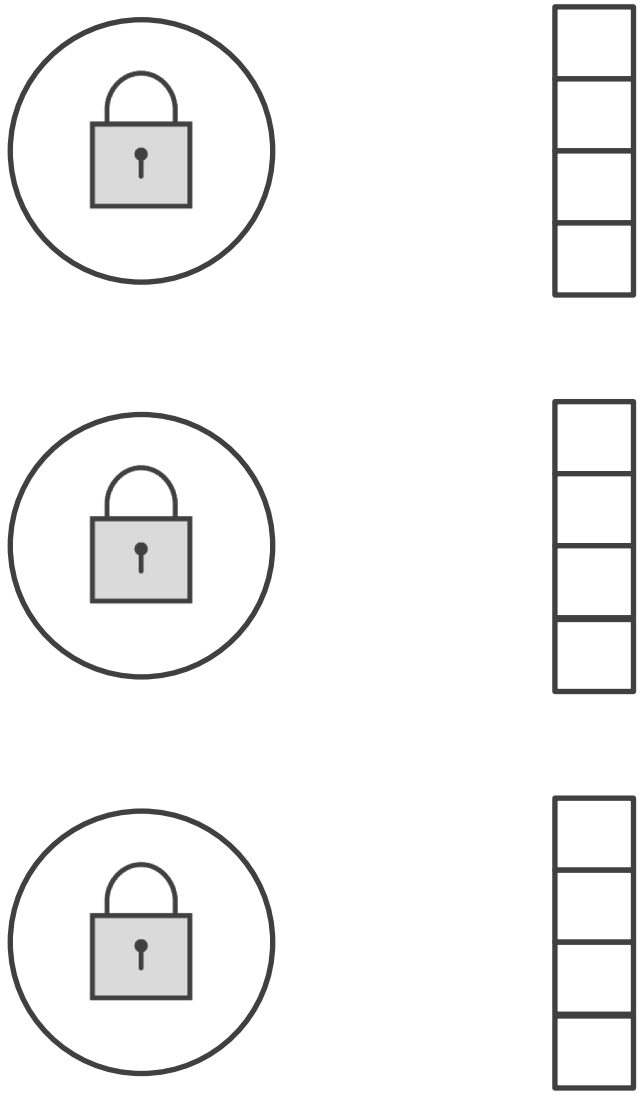

R
W

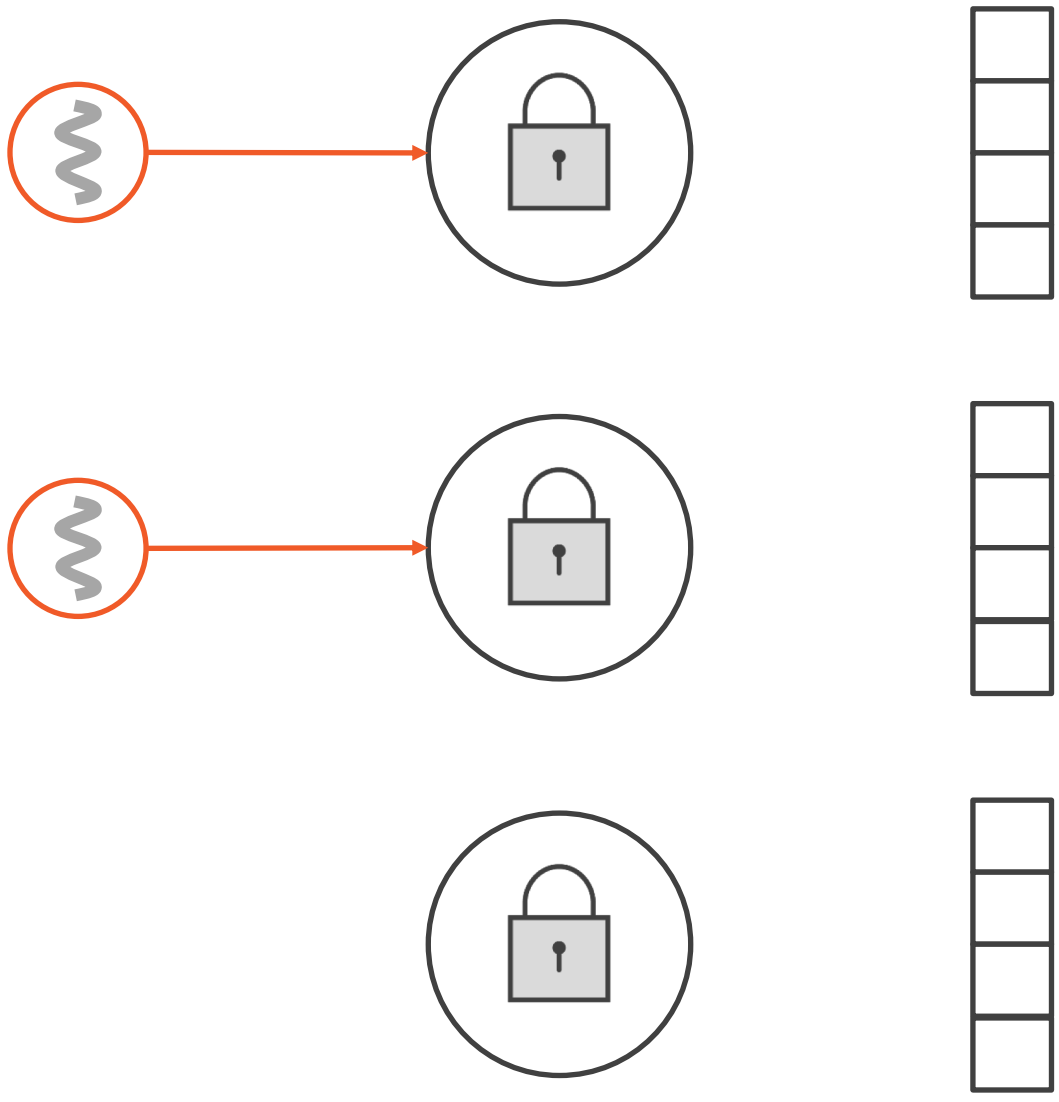

R
W

Lock Striping

Reduces the granularity at which a lock is applied

Applicable when you have a variable sized collection of independent objects





```
public class Users {
  private Object[] locks = new Object[16]; ...
  private Object getLock(long id) {
    return locks[id % locks.length];
  }

  public void updateUserLocation(long userId, float lat, float lon) {
    synchronized (getLock(userId)) {
      User user = db.getUser(userId);
      user.setLat(lat);
      user.setLon(lon);
    }
  }
}
```

```
public class Users {
  private Object[] locks = new Object[16]; ...
  private Object getLock(long id) {
    return locks[id % locks.length];
  }

  public void updateUserLocation(long userId, float lat, float lon) {
    synchronized (getLock(userId)) {
      User user = db.getUser(userId);
      user.setLat(lat);
      user.setLon(lon);
    }
  }
}
```

```
public class Users {
  private Object[] locks = new Object[16]; ...
  private Object getLock(long id) {
    return locks[id % locks.length];
  }

  public void updateUserLocation(long userId, float lat, float lon) {
    synchronized (getLock(userId)) {
      User user = db.getUser(userId);
      user.setLat(lat);
      user.setLon(lon);
    }
  }
}
```

Lock Class Features

- Ability to interrupt a thread that's waiting for a lock
- Ability to set a timeout on the amount of time to wait for a lock
- More scalable under high contention than the synchronized implementation

Atomic Variables and Concurrent Collections

Java Atomic Classes

AtomicBoolean

AtomicInteger, AtomicIntegerArray

AtomicLong, AtomicLongArray

AtomicReference, AtomicReferenceArray

Atomic Classes

- Enables atomic updates
- Enables atomic arithmetic operations in the case of AtomicInteger and AtomicLong objects
- Atomicity prevents data corruption
- Uses the Compare-And-Swap instruction that is provided by modern CPUs

```
public class AggregateScore {
  private static final double MAX_SCORE_PER_SCORER = 100;
  private double score;

  public void updateScore(Scorer scorer) {
    if (scorer.isApplicable(this)) {
      double scorerScore = scorer.calculateScore();
      synchronized(this) {
        this.score += scorerScore < MAX_SCORE_PER_SCORER ?
            scorerScore : MAX_SCORE_PER_SCORER; }
    }
  }
}
```

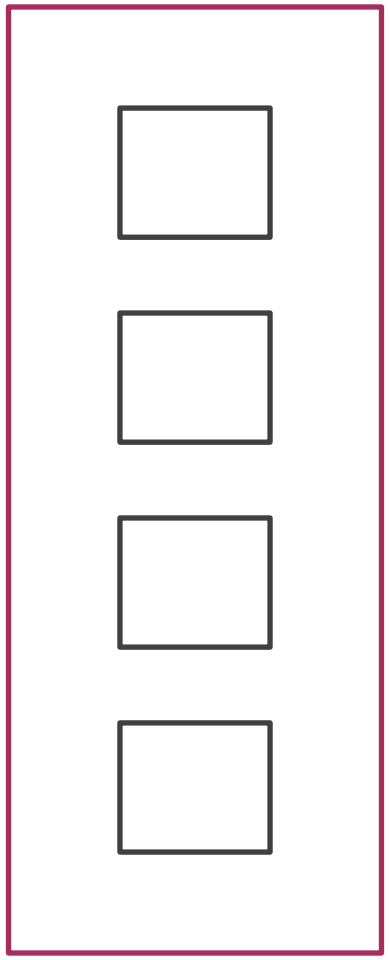
```
public class AggregateScore {
  private static final double MAX_SCORE_PER_SCORER = 100;
  private AtomicLong score = new AtomicLong();

  public void updateScore(Scorer scorer) {
    if (scorer.isApplicable(this)) {
      double scorerScore = scorer.calculateScore();
      synchronized(this) {
        this.score += scorerScore < MAX_SCORE_PER_SCORER ?
            scorerScore : MAX_SCORE_PER_SCORER; }
    }
  }
}
```

```
public class AggregateScore {
  private static final double MAX_SCORE_PER_SCORER = 100;
  private AtomicLong score = new AtomicLong();

  public void updateScore(Scorer scorer) {
    if (scorer.isApplicable(this)) {
      double scorerScore = scorer.calculateScore();
      cappedScore = scorerScore < MAX_SCORE_PER_SCORER ?
           scorerScore : MAX_SCORE_PER_SCORER;
      this.score.getAndAdd(Double.doubleToLongBits(cappedScore));
  }
}
```

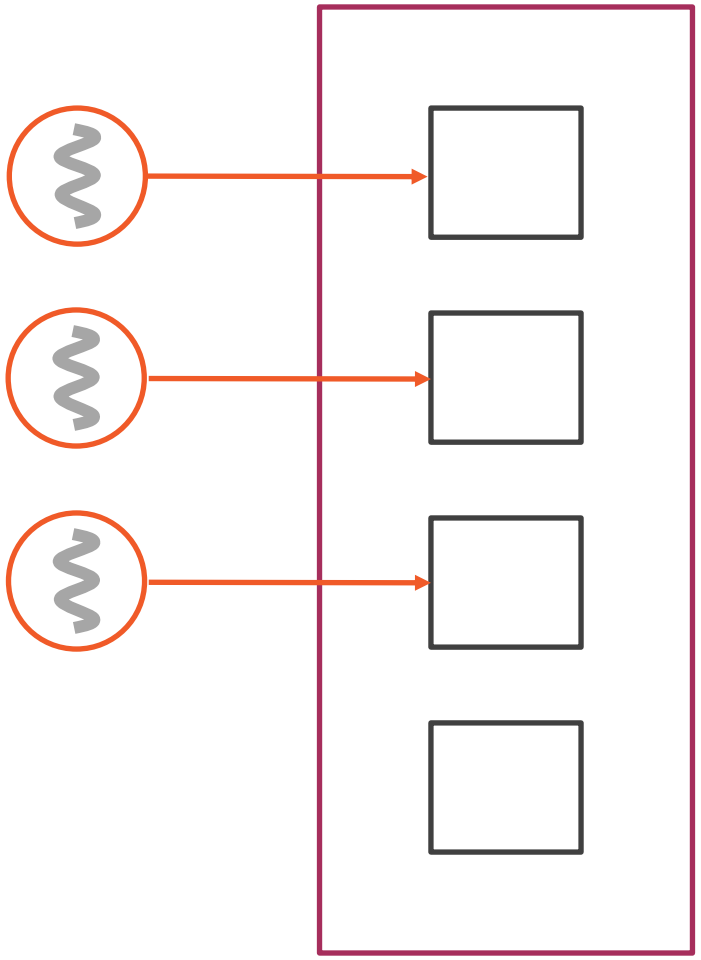
Even Faster Atomics

When there are frequent simultaneous writes by multiple threads

- LongAdder
- LongAccumulator
- Double Adder
- DoubleAccumulator



LongAdder


LongAdder

```
public class ServerStatistics {
  private long numActiveUsers;
  private long numDBQueries;
  private Object activeUsersLock = new Object();
  private Object dbQueriesLock = new Object();

  public void incrementNumActiveUsers(int increment) {
    synchronized (activeUsersLock) { numActiveUsers += increment; }
  }
  public void incrementNumDBQueries(int increment) {
    synchronized (dbQueriesLock) { numDBQueries += increment; }
  }
  ...
}
```

```
public class ServerStatistics {
  private LongAdder numActiveUsers = new LongAdder();
  private LongAdder numDBQueries = new LongAdder();



  public void incrementNumActiveUsers(int increment) {
    numActiveUsers.add(increment);
  }
  public void incrementNumDBQueries(int increment) {
    numDBQueries.add(increment);
  }
  ...
}
```

Concurrent Collections

ConcurrentHashMap

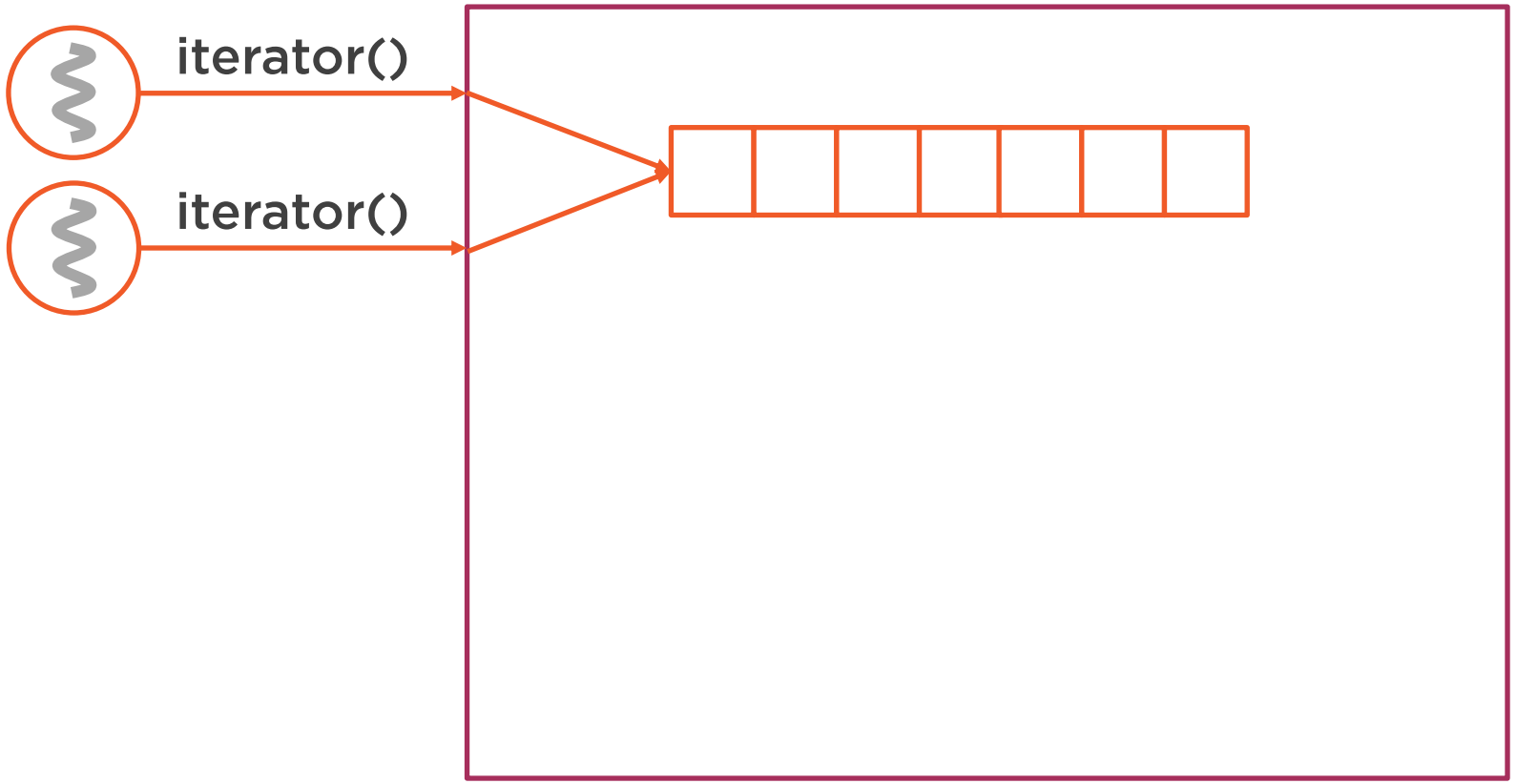
CopyOnWriteArrayList

ConcurrentHashMap

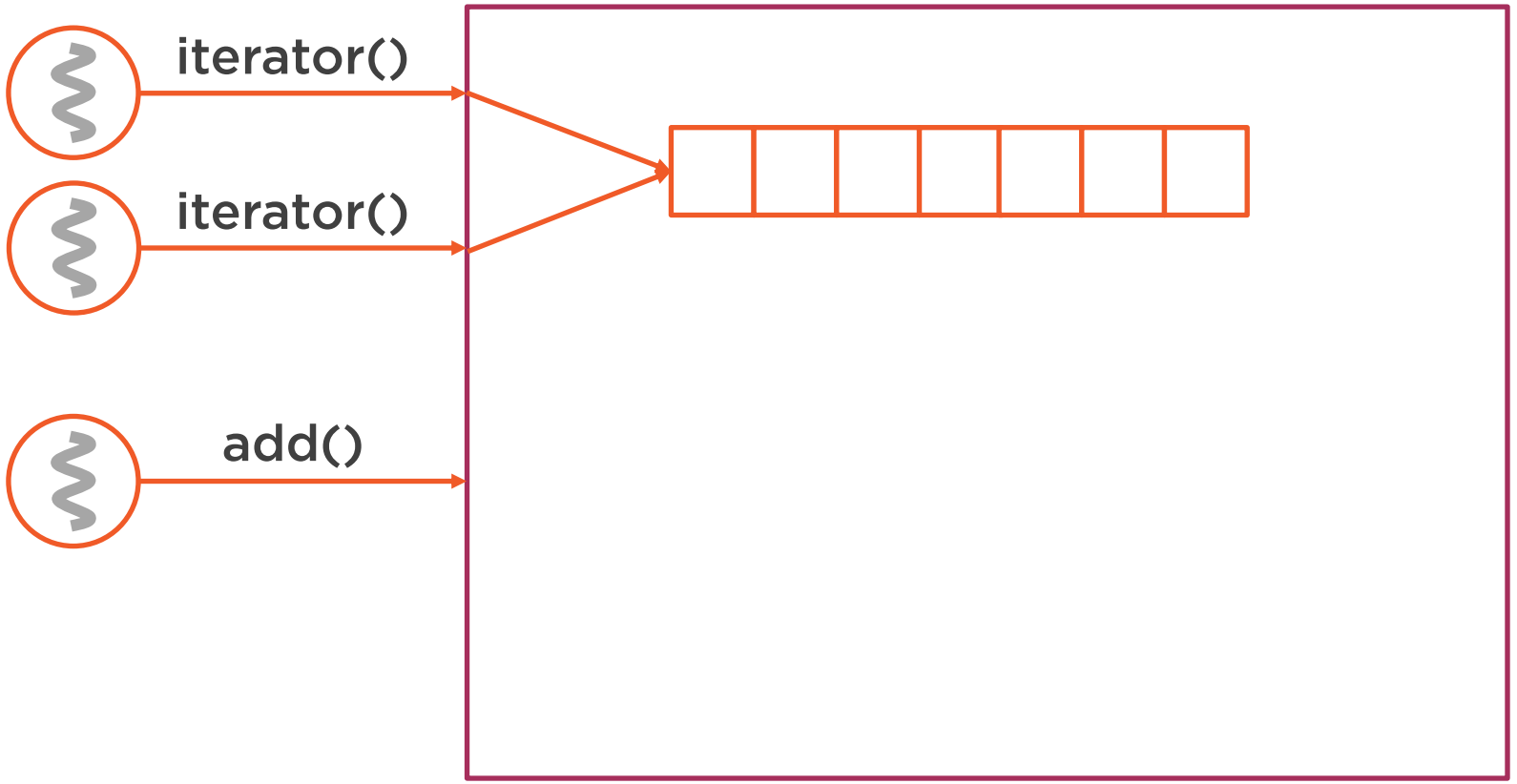
- Supports highly concurrent updates and fully concurrent retrievals
- Earlier implementations used lock striping to enable high concurrency
- Current implementations use a combination of Compare-And-Swap operations and volatile puts and gets
- Drop-in replacement for synchronized maps

CopyOnWriteArrayList

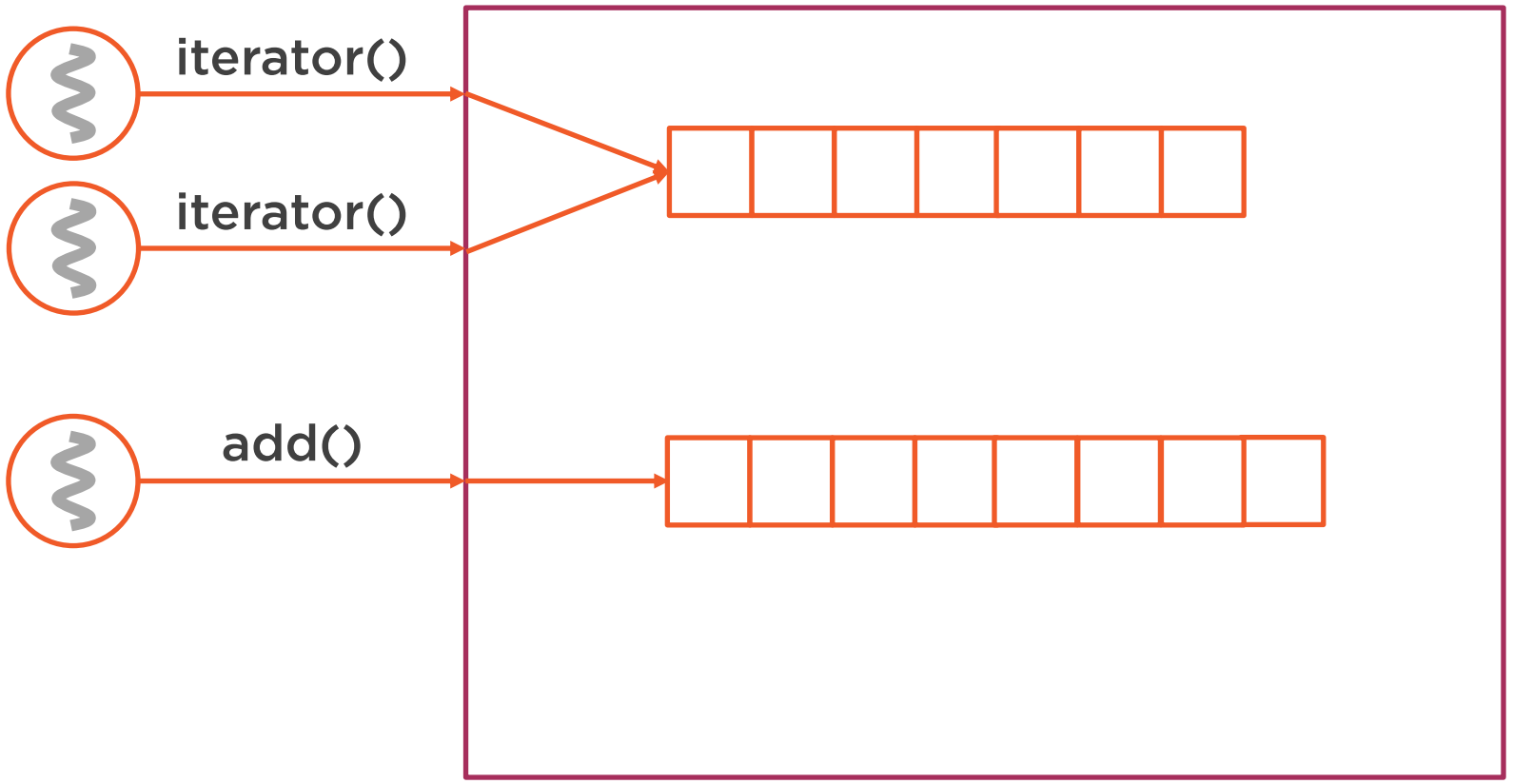
CopyOnWriteArrayList



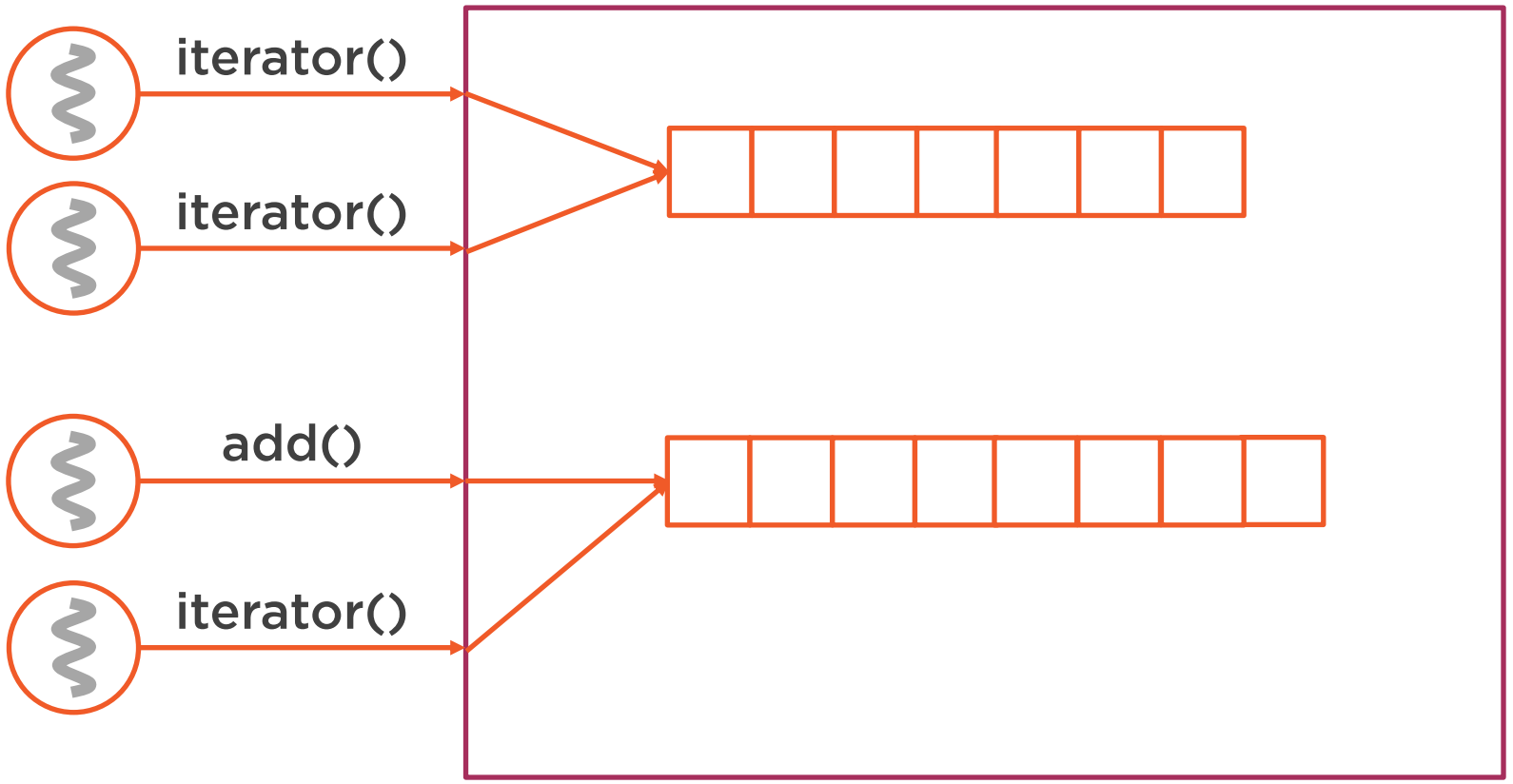
CopyOnWriteArrayList



CopyOnWriteArrayList



CopyOnWriteArrayList



CopyOnWriteArrayList Alternative

CopyOnWriteArrayList doesn't fit your use case

Would like to avoid using a single lock over the entire list

Don't need index based access or ability to insert in the middle of the list

ConcurrentLinkedQueue

```
ConcurrentLinkedQueue<String> list = new
ConcurrentLinkedQueue<>();


// adding to the queue

list.add(“Some String”)


// iterating the queue

for (String str : list) {

  System.out.println(str);

}
```

Avoiding Synchronization

The `volatile` Keyword

Let's the compiler and Java runtime know that this variable is shared

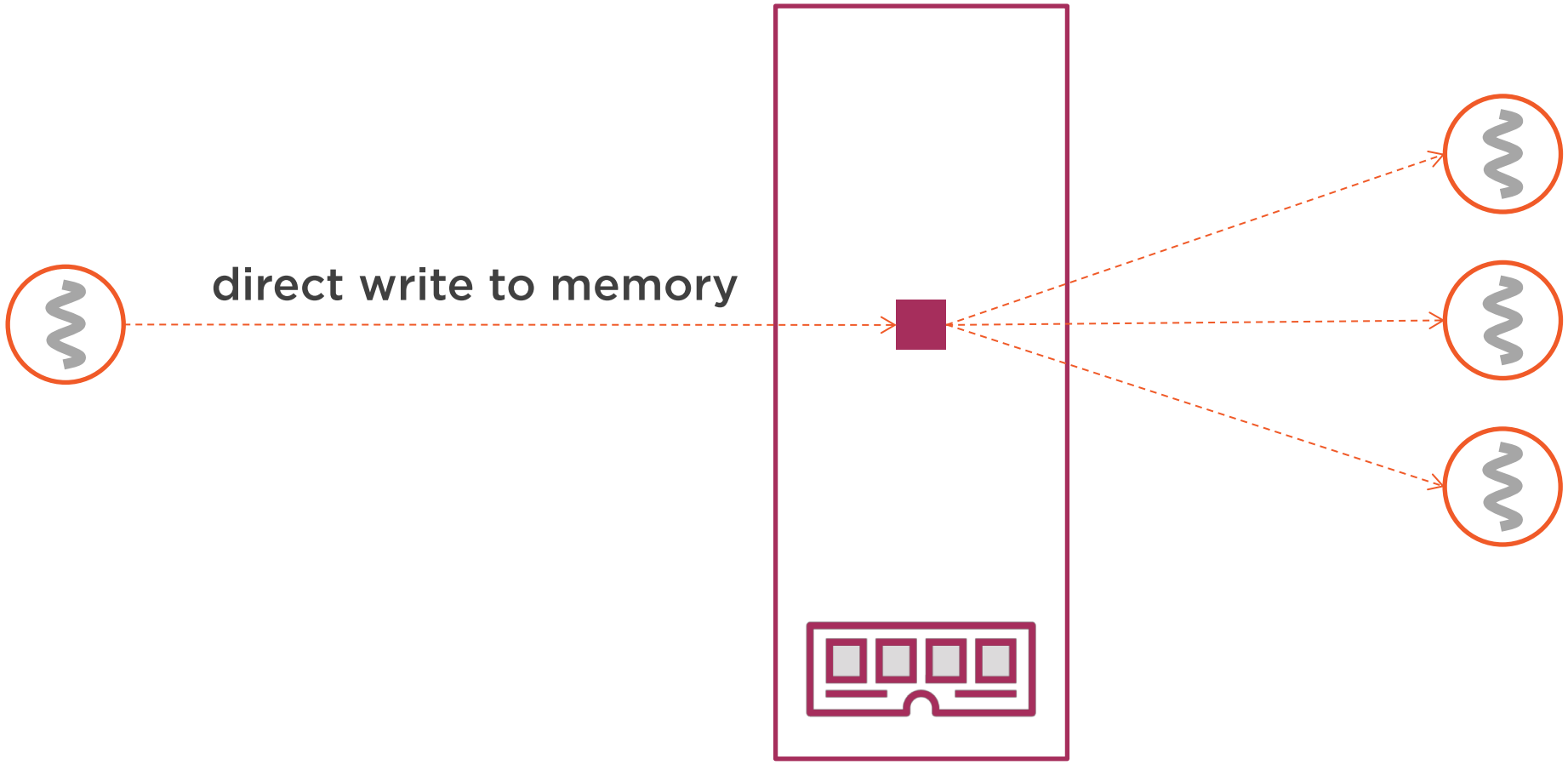
Compiler:

- Will not re-order operations surrounding the variables

Runtime:

- Volatiles are directly read from and written to memory and are not cached

- When a thread writes to a volatile variable, all other variables visible to it are flushed to memory


direct write to memory

Volatile Variables

Most commonly used as a completion, interruption or status flag

Can be used when:

- Updates to a single variable are only being done by one thread

- You have multiple writer threads but writes to the variable do not depend on its current value

Immutable Objects

Do not need synchronized access

Once constructed, their state cannot be changed

The more you use immutable classes when writing concurrent code, the less synchronization and locking that's needed

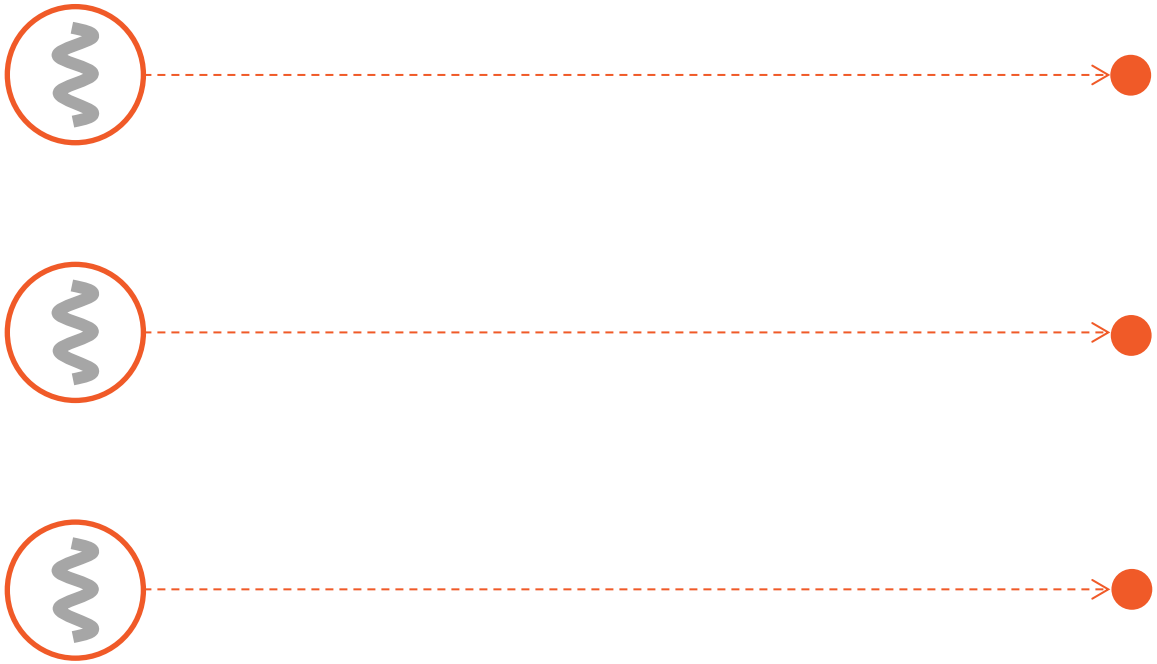
ThreadLocal

Instead of sharing, each thread has their own copy of an object

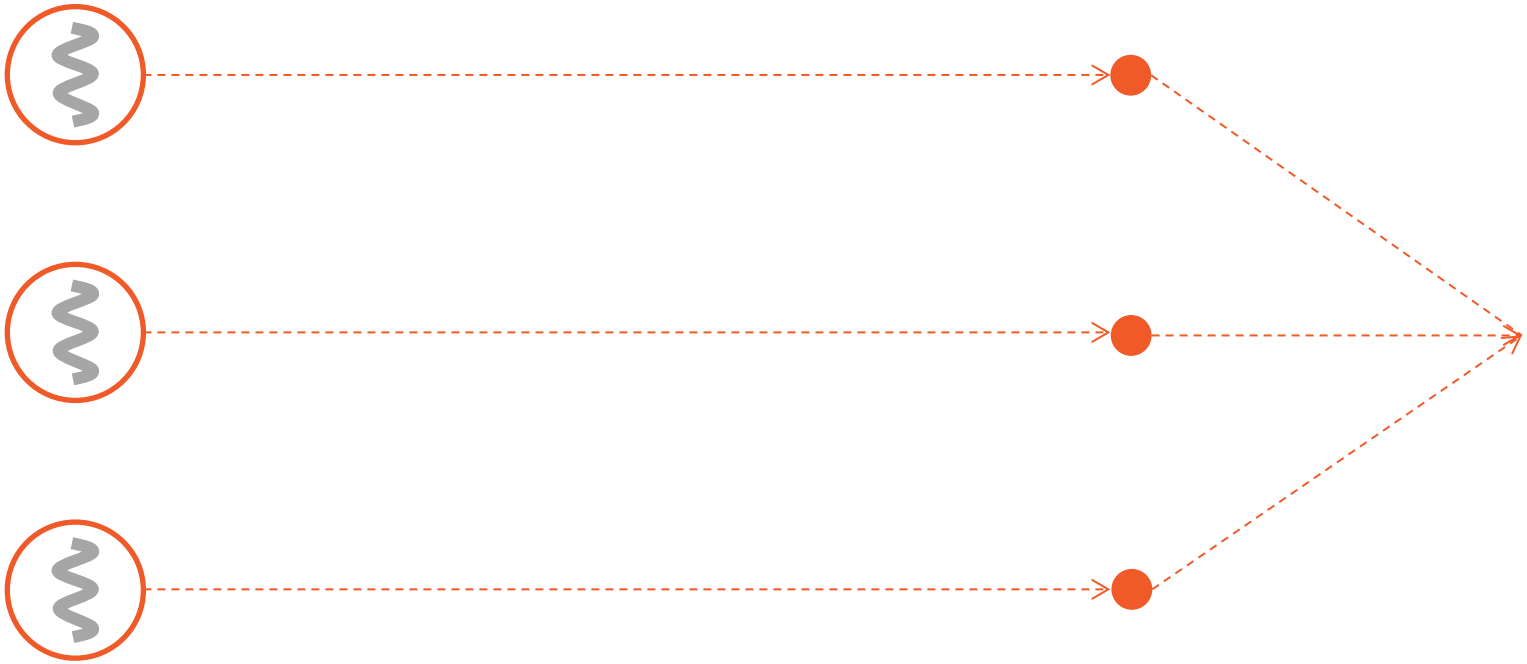
```
public class MySerializer {
  Kyro kryoSerializer = new Kryo(); // shared unsafe object
  kryoSerializer.register(SomeClass.class);

  public synchronized byte[] serializer(SomeClass object) {
    Output output =
        new Output(new ByteArrayOutputStream(), 100);
    kryoSerializer.writeObject(output, object);
    return output.toBytes();
  }
}
```

```
public class MySerializer {
  ThreadLocal<Kryo> kryoSerializers = new ThreadLocal<Kryo>() {
        protected Kryo initialValue() {
            Kryo kryoSerializer = new Kryo();
            kryoSerializer.register(SomeClass.class);
            return kryo; };
  };
  public byte[] serializer(SomeClass object) {
    Output output = new Output(new ByteArrayOutputStream(), 100);
    kryoSerializers.get().writeObject(output, object);
    return output.toBytes();
  }
}
```

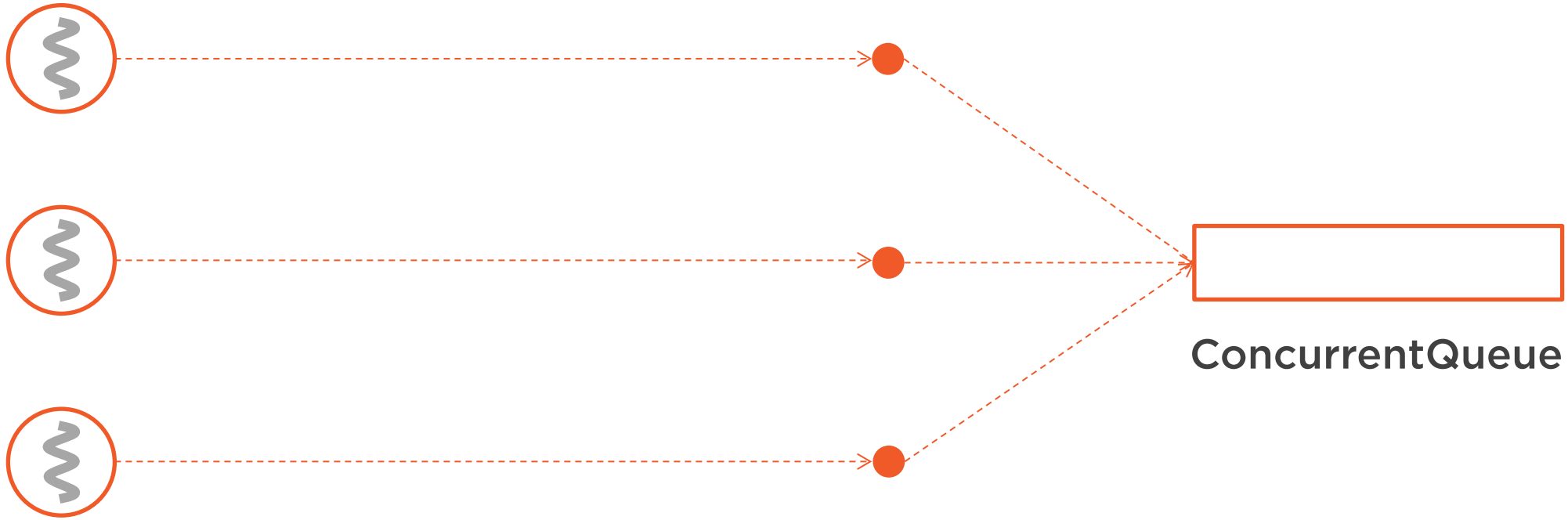
```
public class MySerializer {
  ThreadLocal<Kryo> kryoSerializers = new ThreadLocal<Kryo>() {
        protected Kryo initialValue() {
            Kryo kryoSerializer = new Kryo();
            kryoSerializer.register(SomeClass.class);
            return kryo; };
  };
  public byte[] serializer(SomeClass object) {
    Output output = new Output(new ByteArrayOutputStream(), 100);
    kryoSerializers.get().writeObject(output, object);
    return output.toBytes();
  }
}
```



Parallel shared-nothing execution



Parallel shared-nothing execution



Parallel shared-nothing execution

Concurrent Queues

Non-Blocking Queues

ConcurrentLinkedQueue

ConcurrentLinkedDeque

Blocking Queues

ArrayBlockingQueue

LinkedBlockingQueue

LinkedBlockingDeque

Concurrent Queues

Non-Blocking Queues

ConcurrentLinkedQueue

ConcurrentLinkedDeque

Blocking Queues

ArrayBlockingQueue

LinkedBlockingQueue

LinkedBlockingDeque

Use when:

Concurrent Queues

Non-Blocking Queues

ConcurrentLinkedQueue

ConcurrentLinkedDeque

Blocking Queues

ArrayBlockingQueue

LinkedBlockingQueue

LinkedBlockingDeque

Use when:

- You want the consuming thread to wait for a period of time for a message to arrive

Concurrent Queues

Non-Blocking Queues

ConcurrentLinkedQueue

ConcurrentLinkedDeque

Blocking Queues

ArrayBlockingQueue

LinkedBlockingQueue

LinkedBlockingDeque

Use when:

- You want the consuming thread to wait for a period of time for a message to arrive

- You want to control the growth rate of the queue

Resources

Online articles and blog posts

Java Concurrency in Practice

Java Performance: The Definitive Guide