CS5010: Lecture 10
More Concurrency

Fall 2022

Brian Cross, Northeastern University

Lots of credit: Ian Gorton adapted by Abi Evans

Before we get started...

Pull the latest code from Code_from_Lectures (course Github org)

https://github.ccs.neu.edu/cs5010seaF22/Code_from_Lectures

Then, open Lecture10 project in the Evening_Lectures if you'd like to follow along.

Administrivia

- Homework #4 Code Walks
  - Everyone did a video Codewalk that was due last night
  - Half the class will present their code for hw4 next week
  - 2nd Half will present their code for hw5 after Thanksgiving week.
  - Presenters picked at random.
  - Code walks happen 2nd half of class
  - We'll use teams to present to the class
- Homework #5 out tomorrow
  - Due Monday, November 28th @ 11:59pm
  - Thanksgiving week just before
- No Lab next week.

Agenda

- Thread pools
- Executors
- Loops/recursion and threads
- Readers & Writers problem
- Scalability
- Testing

Recap from last week

- If threads share state → must use locks to achieve thread safety
    - Synchronized methods and objects
    - `java.util.concurrent` thread-safe collections, variables
- If threads need to coordinate actions (e.g. do something only if there is data) →
    - Guarded conditions (wait/notify)
    - Beware of deadlocks

Reflection on Assignment 5

- Threads don't necessarily make programs faster.
- In this case → use threads to reduce the amount of the file that must be stored in memory at any one time, maybe speed up some parts of processing

Thread pools & the Executor framework

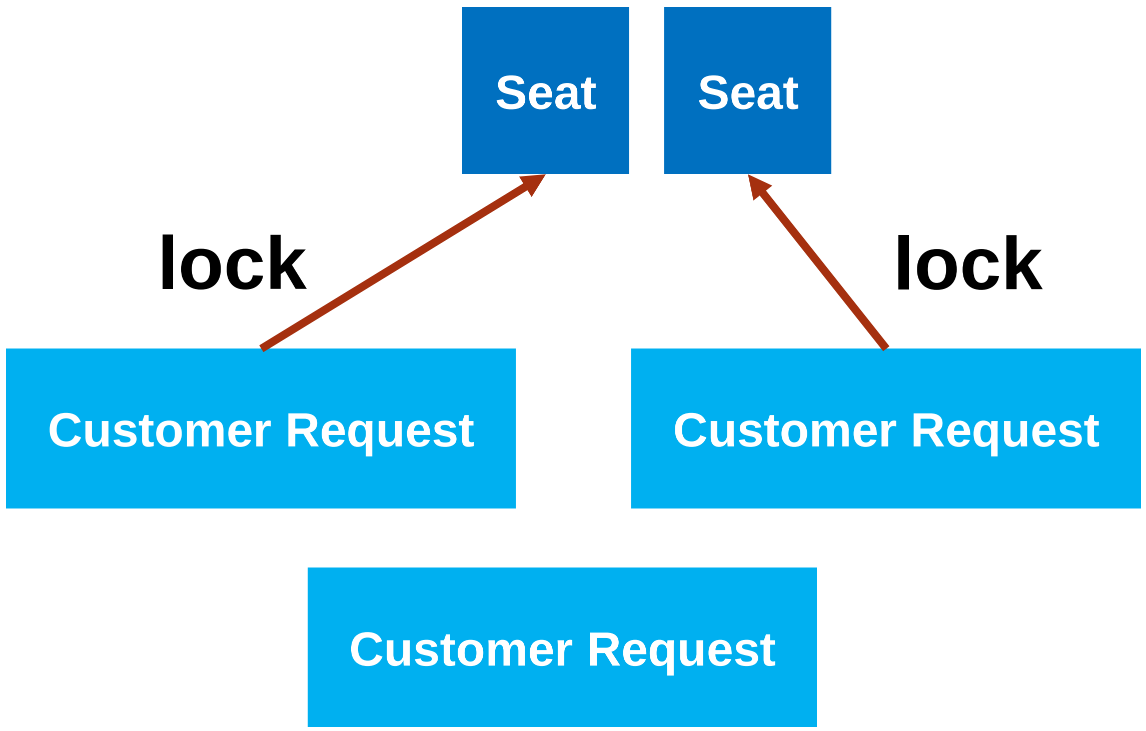
Scenario: ticketing system

- Seat availability stored in a database
- Must handle large volumes of concurrent read requests
  - Customer wants to see available tickets
- Must handle large volumes of concurrent write requests
  - Customer purchases tickets
- Can’t allow same seat to be sold multiple times

Scenario: ticketing system

Possible solution:

- Create a separate thread for each customer request
- Request holds lock on Seat until purchase completed (or customer changes their mind



must wait until lock available (or all seats taken → request denied)

Scenario: ticketing system

First attempt:

- Create a separate thread for each customer request
- Request holds lock on Seat until purchase completed (or customer changes their mind

Drawback?

- Threads use a lot of memory
- More time and resources spent on creating/destroying threads than processing

- > 70,000 seats
- Very high demand
- A lot to manage!

Thread pools

→ a **pool** (i.e. a group) of Threads that can be reused / queued

- Reduces overhead of thread management

In Java → Executor framework

`java.util.concurrent`

Utilities to simplify multithreaded programs:

- Atomic variables ☑
- Thread safe collections ☑
- **Executor framework**
- Lock objects

The Executor Framework

`public interface Executor`

- An object that executes submitted Runnable tasks
- Supports asynchronous task execution
- Decouples task submission from task executions
    - Supports different task execution policies
    - Provides task lifecycle support
    - Has hooks for statistics, management, monitoring
- Factory methods to create an Executor with desired policies

The Executor Framework

`public interface Executor`

- An object that executes submitted Runnable tasks

So far:

```
Thread t1
      = new Thread(new SomeRunnable());
Thread t2
      = new Thread(new SomeRunnable());
Thread t3
      = new Thread(new SomeRunnable());

t1.start();
t2.start();
t3.start();
```

With Executor:

```
Executor executor = anExecutor;

executor.execute(new SomeRunnable());
executor.execute(new SomeRunnable());
executor.execute(new SomeRunnable());
```

The Executor Framework

`public interface Executor`

- An object that executes submitted Runnable tasks

So far:

```
Thread t1
        = new Thread(new SomeRunnable());
Thread t2
        = new Thread(new SomeRunnable());
Thread t3
        = new Thread(new SomeRunnable());

t1.start();
t2.start();
t3.start();
```

With Executor:

```
Executor executor = anExecutor;

executor.execute(new SomeRunnable());
executor.execute(new SomeRunnable());
executor.execute(new SomeRunnable());
```

handles thread creation and start

Using the Executor framework

Step 1: declare an Executor

```
Executor mostSimple;
ExecutorService typical;
ScheduledExecutorService withTimer;
```

Using the Executor framework

Step 1: declare an Executor

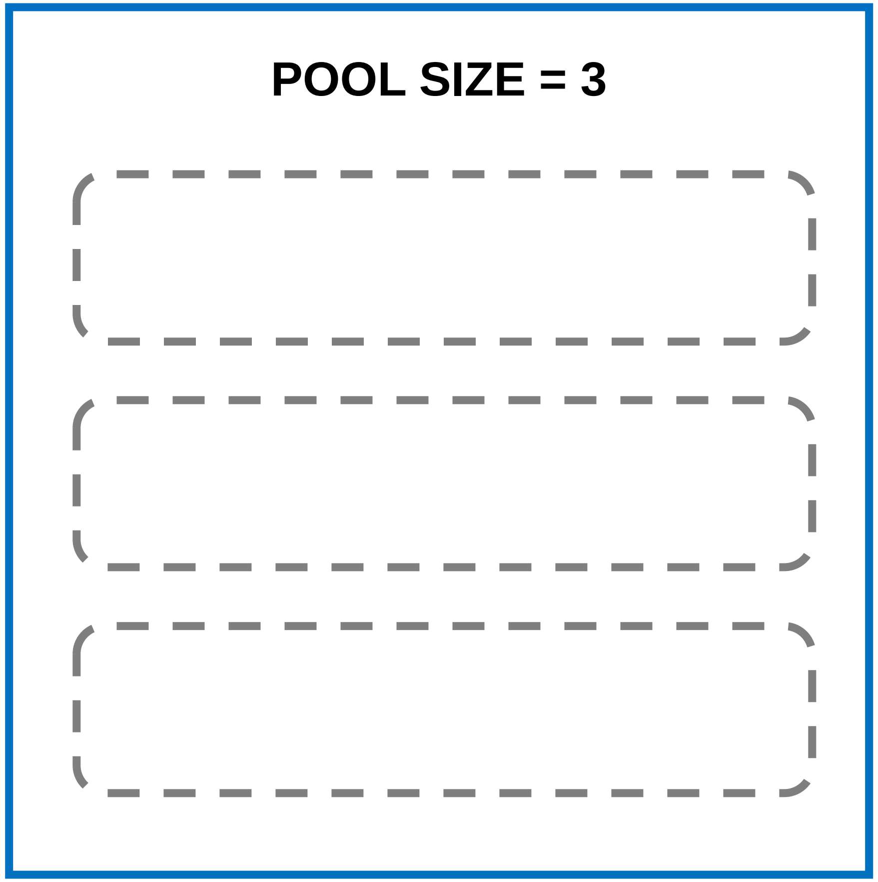
```
Executor mostSimple;
ExecutorService typical;
ScheduledExecutorService withTimer;
```

Step 2: create it with a factory method. Two possibilities among many:

```
ExecutorService e1
                = Executor.newFixedThreadPool(SIZE);
ExecutorService e2
                = Executor.newSingleThreadExecutor();
```

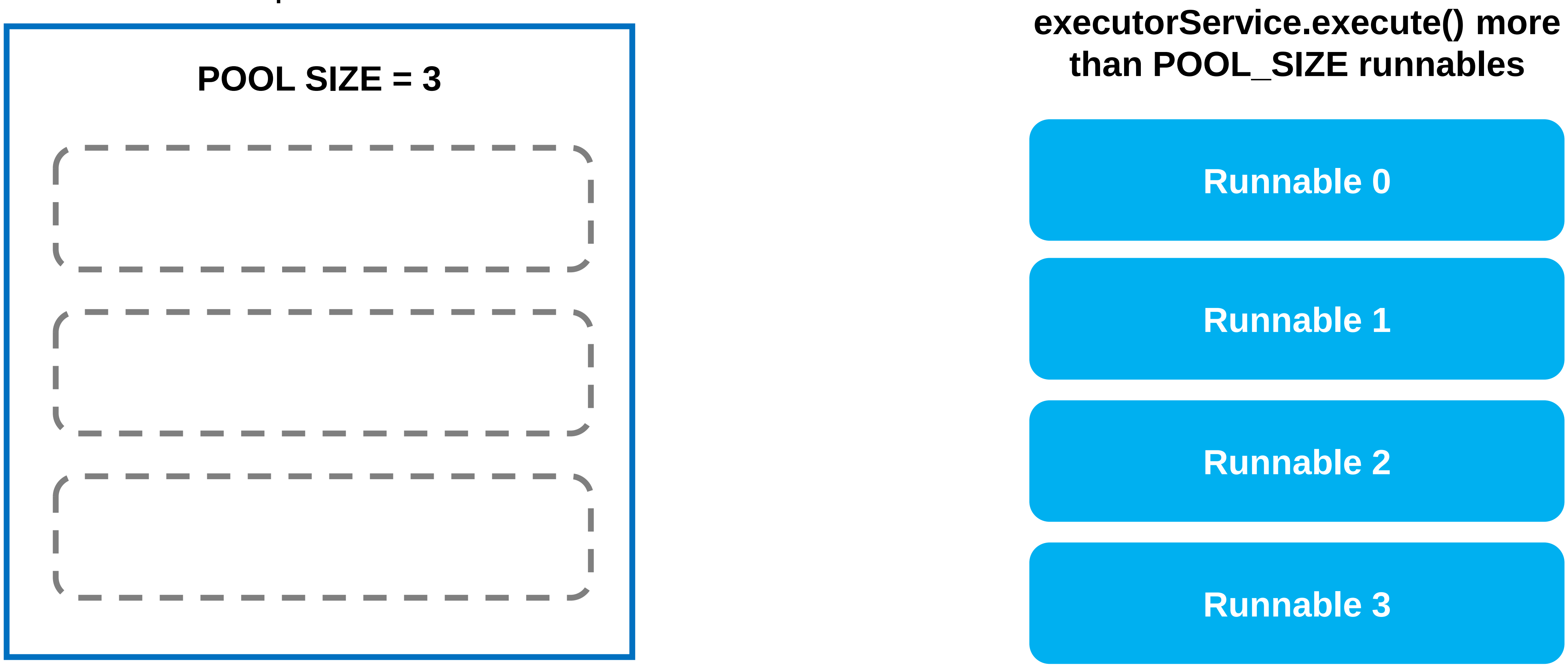
How thread pools work

When there are more threads to execute than the pool can hold, the extras are queued



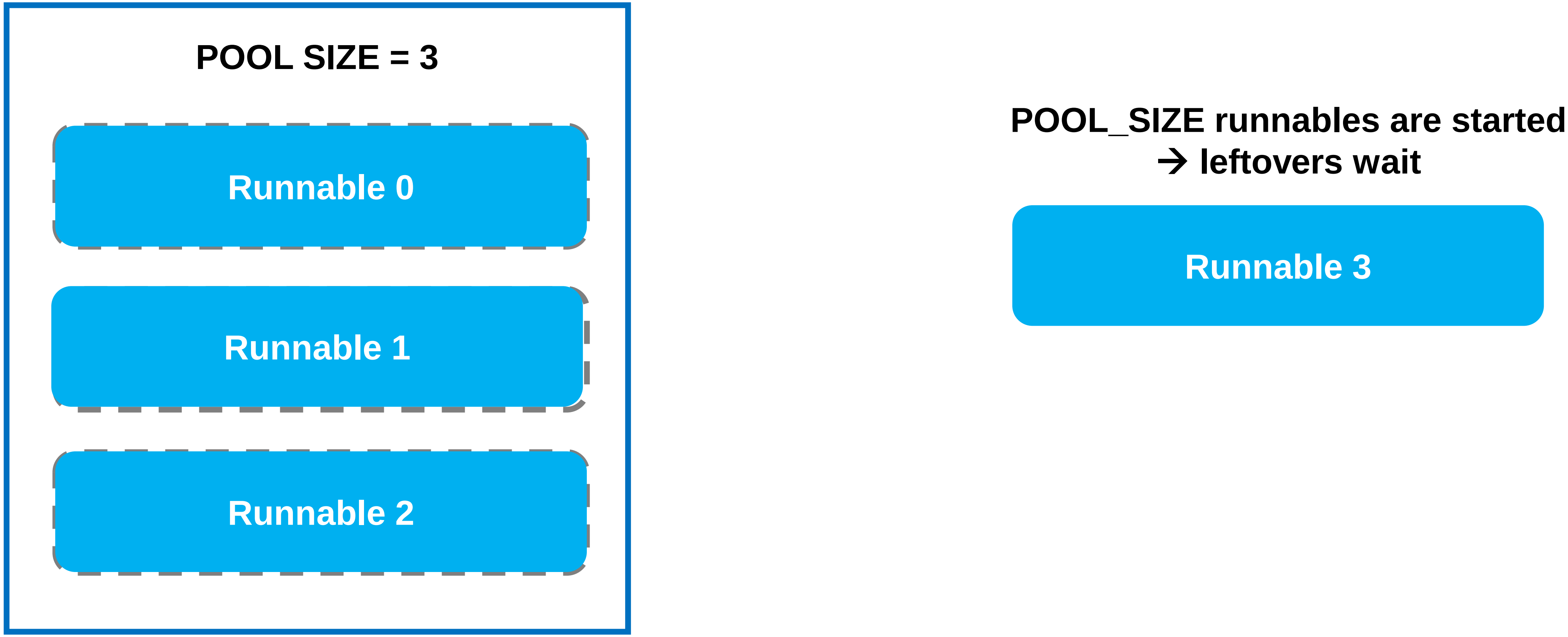
How thread pools work

When there are more threads to execute than the pool can hold, the extras are queued



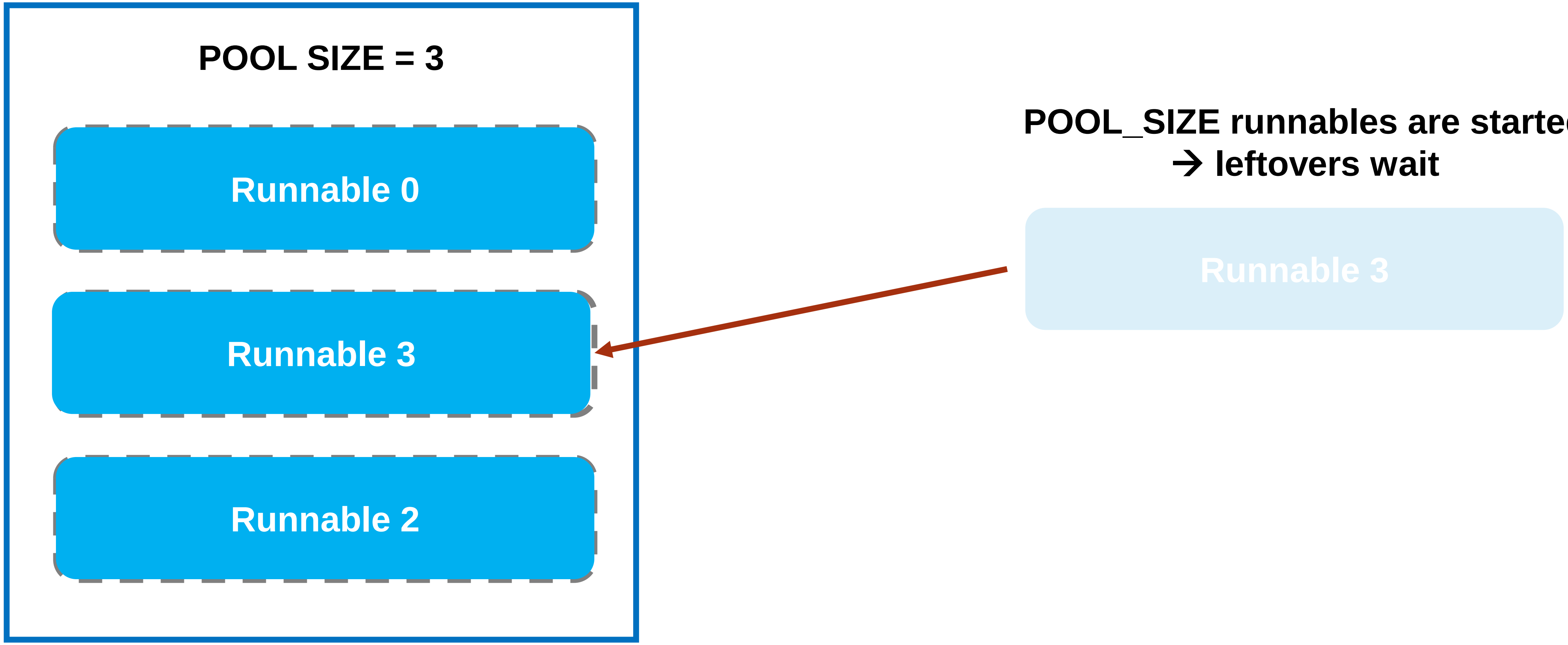
How thread pools work

When there are more threads to execute than the pool can hold, the extras are queued



How thread pools work

When a thread completes → exits the pool → queued thread takes its place



Thread pool in code

```
ExecutorService executor
                    = Executors.newFixedThreadPool(2);

executor.execute(new SomeRunnable());
executor.execute(new SomeRunnable());
executor.execute(new SomeRunnable());
executor.execute(new SomeRunnable());
```

All 4 runnables added to the pool queue → 2 at a time will run

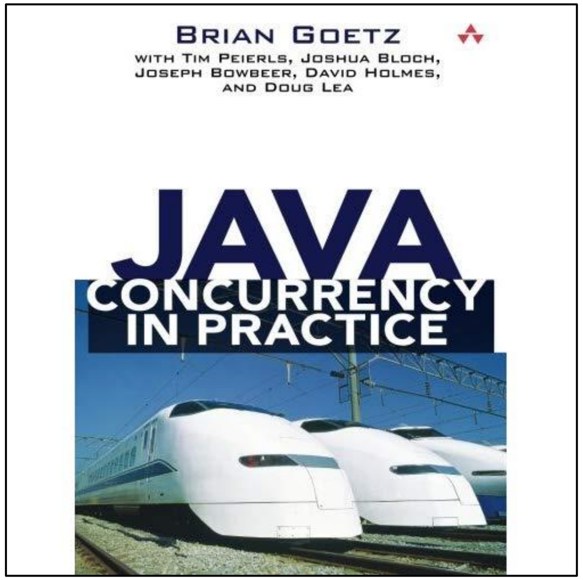
Why is a pool (often) better?

- Can restrict the number of threads created to the number that can meaningfully be used
- Reuses threads instead of creating new ones

→ less overhead

How big should the pool be?

It depends.



Brian Goetz’s formula:

```
available_cores * (1 + wait_time/service_time)
```

How many cores do you have?

```
Runtime.getRuntime().availableProcessors();
```

Wait time = time spent waiting for I/O (large files, HTTP requests)

Service time = time spent on tasks

Ticketing system with a thread pool

Original: start a new Thread for every incoming request

Revised: use a thread pool capped at some sensible size e.g. 5

Ticketing system with a thread pool

Original: start a new Thread for every incoming request

Revised: use a thread pool capped at some sensible size e.g. 5
→ **A LOT faster**

Some Executor limitations

- No way to obtain the result of a Runnable (e.g. was it successful?)
  - If necessary, often not
- Or find out when the Runnable has completed

To get results → Future and/or Callable

- Future: the future result of an asynchronous task
- Callable:
  - Like a Runnable but it *returns* something
  - Can only be used with an Executor

Using Future with a Runnable

`executorService.submit(runnable)`

```
Future future = executorService.submit(new Runnable()
{
  public void run() {...do stuff...}
});

future.get();
```

Using Future with a Runnable

`executorService.submit(runnable)`

```
Future future = executorService.submit(new Runnable()
{
  public void run() {...do stuff...}
});

future.get();
```

submit, not execute

Using Future with a Runnable

`executorService.submit(runnable)`

Stores the (eventual) result

```
Future future = executorService.submit(new Runnable()
{
   public void run() {...do stuff...}
});

future.get();
```

Using Future with a Runnable

`executorService.submit(runnable)`

```
Future future = executorService.submit(new Runnable()
{
  public void run() {...do stuff...}
});
```

```
future.get();
```

Blocks until the result is available

- Non-blocking: future.isDone()
- Result will be null

Using Future with a Callable

`executorService.submit(callable)`

```
Future future = executorService.submit(new Callable()
{
   public Object call() throws Exception {
      return “Result”;
   }
});

System.out.println(future.get());
```

Using Future with a Callable

`executorService.submit(callable)`

```
Future future = executorService.submit(new Callable()
{
   public Object call() throws Exception {
      return “Result”;
   }
});

System.out.println(future.get());
```

submit, not execute

Using Future with a Callable

`executorService.submit(callable)`

```
Future future = executorService.submit(new Callable()
{
   public Object call() throws Exception {
      return “Result”;
   }
});

System.out.println(future.get());
```

Using Future with a Callable

`executorService.submit(callable)`

```
Future future = executorService.submit(new Callable()
{
   public Object call() throws Exception {
      return “Result”;
   }
});

System.out.println(future.get());
```

Key differences:

- call() not run()
- returns something

Using Future with a Callable

`executorService.submit(callable)`

```
Future future = executorService.submit(new Callable()
{
   public Object call() throws Exception {
      return “Result”;
   }
});

System.out.println(future.get());
```

Using Future with a Callable

Use Generics

```
Future<String> future = executorService.submit(new Callable<String>() {
  public String call() throws Exception {
    return “Result”;
  }
});

System.out.println(future.get());
```

Shutting an executor down

An executor must be shutdown

```
executorService.shutdown();
```

→ stops accepting new requests but does not shutdown immediately

To wait for all threads to complete after shutdown:

```
executorService.awaitTermination(time, unit);
```

Coordinating threads

CyclicBarrier

When a <u>known number</u> of threads have to execute before moving on

```
CyclicBarrier barrier
      = new CyclicBarrier(NUM, new OtherRunnable());
```

- After NUM threads have executed, run OtherRunnable()
  - 2nd parameter (action) is optional
- Cyclic – if there are more than NUM threads, OtherRunnable() will execute after every NUM threads

CyclicBarrier

When a known number of threads have to execute before moving on

```
CyclicBarrier barrier
    = new CyclicBarrier(NUM, new OtherRunnable());

for (int i = 0; i < NUM; i++) {
  Thread t = new Thread(new Runnable() {
    public void run() {
      barrier.await(); // Call at end
    }
  }).start();
}
```

The barrier counts the number of await() calls

Exercise: CyclicBarrier with a thread pool

Open cyclicbarrier > CyclicBarrierDeadlock.java

- Take a few minutes to make sure you understand the code

Then, try running the following conditions:

- barrier’s number of parties = POOL_SIZE; for loop: i < POOL_SIZE
- barrier’s number of parties = POOL_SIZE; for loop: i < POOL_SIZE * 2
- barrier’s number of parties = POOL_SIZE * 2; for loop as above

What do you observe?

Exercise: CyclicBarrier with a thread pool

Deadlock

- When barrier is waiting on await() calls > POOL_SIZE

Why?

barrier.await() blocks → runnable doesn't exit → pool space doesn't free up → next runnable in the queue can't take its place

Try printing something after await() to illustrate this

Loops / Recursion and Threads

Loops and threads

If we have a loop with completely independent iterations, can we use a thread to execute each iteration?

What are the effects on performance?

- Look at the drivers in looprunnables package
- (not in class) Try increasing the multiplying number of iterations by 10 and running → do this a few times
  - Few iterations → sequential (may be) faster
  - As iterations increase → concurrent becomes faster

Loops and threads

Parallelization of sequential loops works when:

- Each iteration **is completely independent** of others
- **Work done in each iteration is enough to offset cost** of thread management

Recursion and threads

Recursive algorithms often involve independent sequential loops → each iteration does not require results of recursive iterations it invokes.

Example:

- Depth-first tree traversal

Depth-First Tree Traversal: sequential

From https://jcip.net/listings.html (listing 8.11)

```
public <T> void sequentialRecursive(List<Node<T>> nodes,
                                    Collection<T> results) {
    for (Node<T> n : nodes) {
        results.add(n.compute());
        sequentialRecursive(n.getChildren(), results);
    }
}
```

Depth-First Tree Traversal: parallel

From https://jcip.net/listings.html (listing 8.11)

```
public <T> void parallelRecursive(final Executor exec,
                                  List<Node<T>> nodes,
                                  final Collection<T> results) {
    for (final Node<T> n : nodes) {
        exec.execute(new Runnable() {
            public void run() {
                results.add(n.compute());
            }
        });
        parallelRecursive(exec, n.getChildren(), results);
    }
}
```

Depth-First Tree Traversal: parallel results

From https://jcip.net/listings.html (listing 8.11)

```
public <T> Collection<T> getParallelResults(List<Node<T>> nodes)
        throws InterruptedException {

    ExecutorService exec = Executors.newCachedThreadPool();
    Queue<T> resultQueue = new ConcurrentLinkedQueue<T>();

    parallelRecursive(exec, nodes, resultQueue);

    exec.shutdown();
    exec.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);

    return resultQueue;
}
```

Readers & Writers Problem

Readers-writers problem

- Classic concurrency problem
- Multiple readers and writers to a shared database / file
- How to manage concurrent read/write requests?

Things to think about:

- Read/write requests could come in at any time and in any order
- Probably many more people looking at seat availability (readers) than making a purchase (writers)

Exercise: Readers & Writers

- **Spend a few minutes getting to know the code in ticketsystemdatabase**
- Run DatabaseDriver.java in each of the following conditions:
    - Readers: 1, writers: 1
    - Readers: 5, writers: 2
    - Readers: 10, writers: 3
- **What do you observe?**

ReadWriteLock

- Maintains a pair of associated locks
  - One for read-only operations
  - One for writing
- The read lock may be held simultaneously by multiple reader threads
  - If there are no writers
- The write lock is exclusive

Using a ReadWriteLock

```
private ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

public Object get() {
  readWriteLock.readLock().lock();
  try { return someObject; }
  finally { readWriteLock.readLock().unlock(); }
}

public void set() {
  readWriteLock.writeLock().lock();
  try { ...write/change something... }
  finally { readWriteLock.writeLock().unlock(); }
}
```

Using a ReadWriteLock

```
private ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

public Object get() {
  readWriteLock.readLock().lock();
  try { return someObject; }
  finally { readWriteLock.readLock().unlock(); }
}

public void set() {
  readWriteLock.writeLock().lock();
  try { ...write/change something... }
  finally { readWriteLock.writeLock().unlock(); }
}
```

Create a ReadWriteLock

- Will apply to instance of the current class

Using a ReadWriteLock

```
private ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
```

```
public Object get() {
  readWriteLock.readLock().lock();
  try { return someObject; }
  finally { readWriteLock.readLock().unlock(); }
}
```

Handles read operations

Needs to lock/unlock

`readWriteLock.readLock()`

```
public void set() {
  readWriteLock.writeLock().lock();
  try { ...write/change something... }
  finally { readWriteLock.writeLock().unlock(); }
}
```

Handles write operations

Needs to lock/unlock

`readWriteLock.writeLock()`

Using a ReadWriteLock

```
private ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

public Object get() {
  readWriteLock.readLock().lock();
  try { return someObject; }
  finally { readWriteLock.readLock().unlock(); }
}

public void set() {
  readWriteLock.writeLock().lock();
  try { ...write/change something... }
  finally { readWriteLock.writeLock().unlock(); }
}
```

1. Lock...

`readWriteLock.readLock()`

1. Lock...

`readWriteLock.writeLock()`

Using a ReadWriteLock

```
private ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

public Object get() {
  readWriteLock.readLock().lock();
  try { return someObject; }
  finally { readWriteLock.readLock().unlock(); }
}

public void set() {
  readWriteLock.writeLock().lock();
  try { ...write/change something... }
  finally { readWriteLock.writeLock().unlock(); }
}
```

2. Perform the read & return

`try { }`

2. Perform the write

`try { }`

Using a ReadWriteLock

```
private ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

public Object get() {
  readWriteLock.readLock().lock();
  try { return someObject; }
  finally { readWriteLock.readLock().unlock(); }
}

public void set() {
  readWriteLock.writeLock().lock();
  try { ...write/change something... }
  finally { readWriteLock.writeLock().unlock(); }
}
```

3. Unlock the read lock

`finally { }`

3. Unlock the write lock

`finally { }`

Exercise: Using a ReadWriteLock

```
private ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

public Object get() {
  readWriteLock.readLock().lock();
  try { return someObject; }
  finally { readWriteLock.readLock().unlock(); }
}

public void set() {
  readWriteLock.writeLock().lock();
  try { ...write/change something... }
  finally { readWriteLock.writeLock().unlock(); }
}
```

Add a ReadWriteLock to SeatDatabase.java

Run with 10 readers and 3 writers

Scalability

Scalability

The ability to improve performance / capacity with additional resources:

- CPUs
- Memory
- Storage
- Bandwidth
- etc.

Engineering concerns

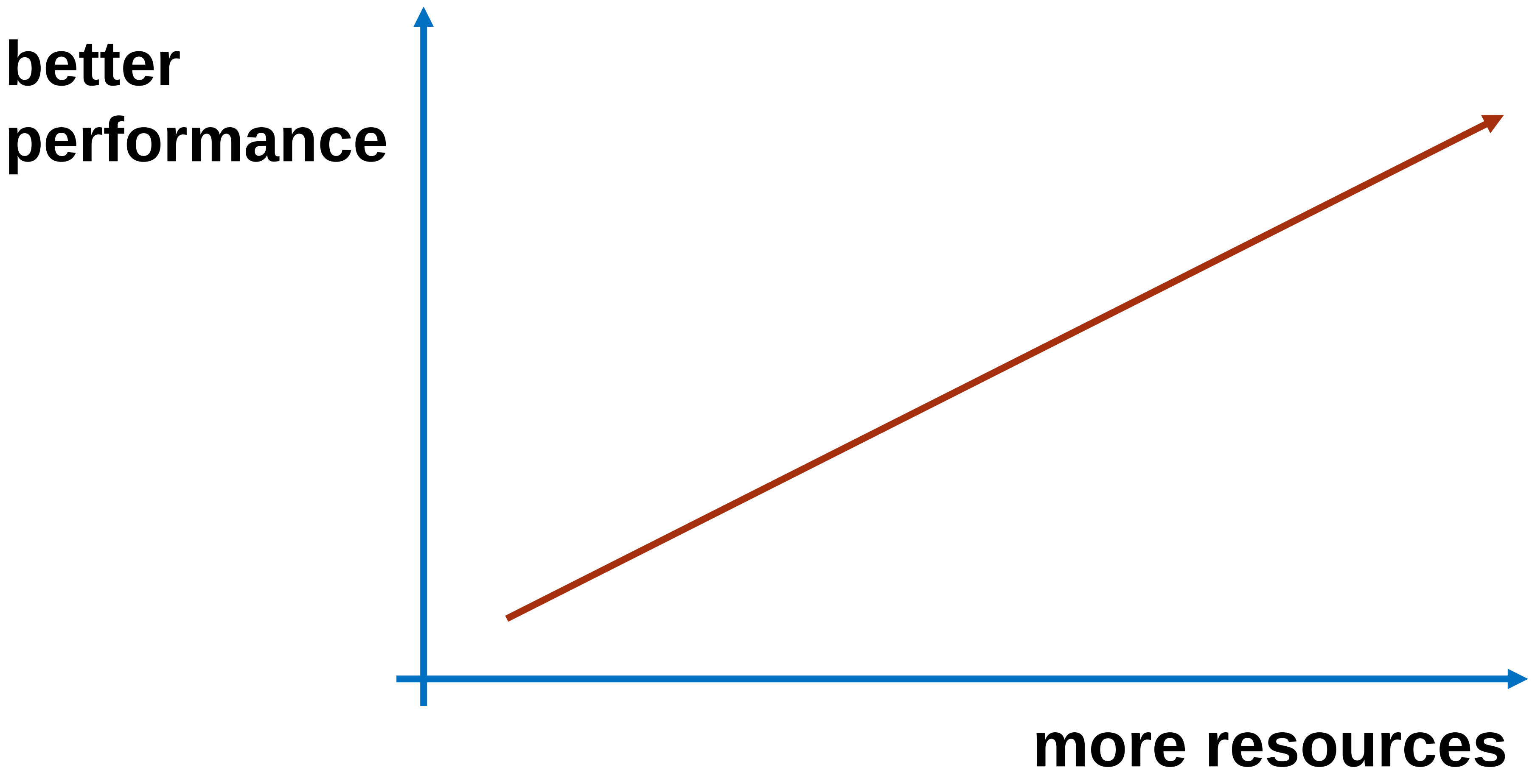
- Threads make it possible to better utilize resources
- But also introduce overheads:
  - creation, context switching, management, coordination

Terminology:

- Service time / latency / response time → how fast a piece of work happens
- Capacity / throughput → how much work can be performed with a given quantity of computing resources

More resources = better performance?

What we would like to happen:

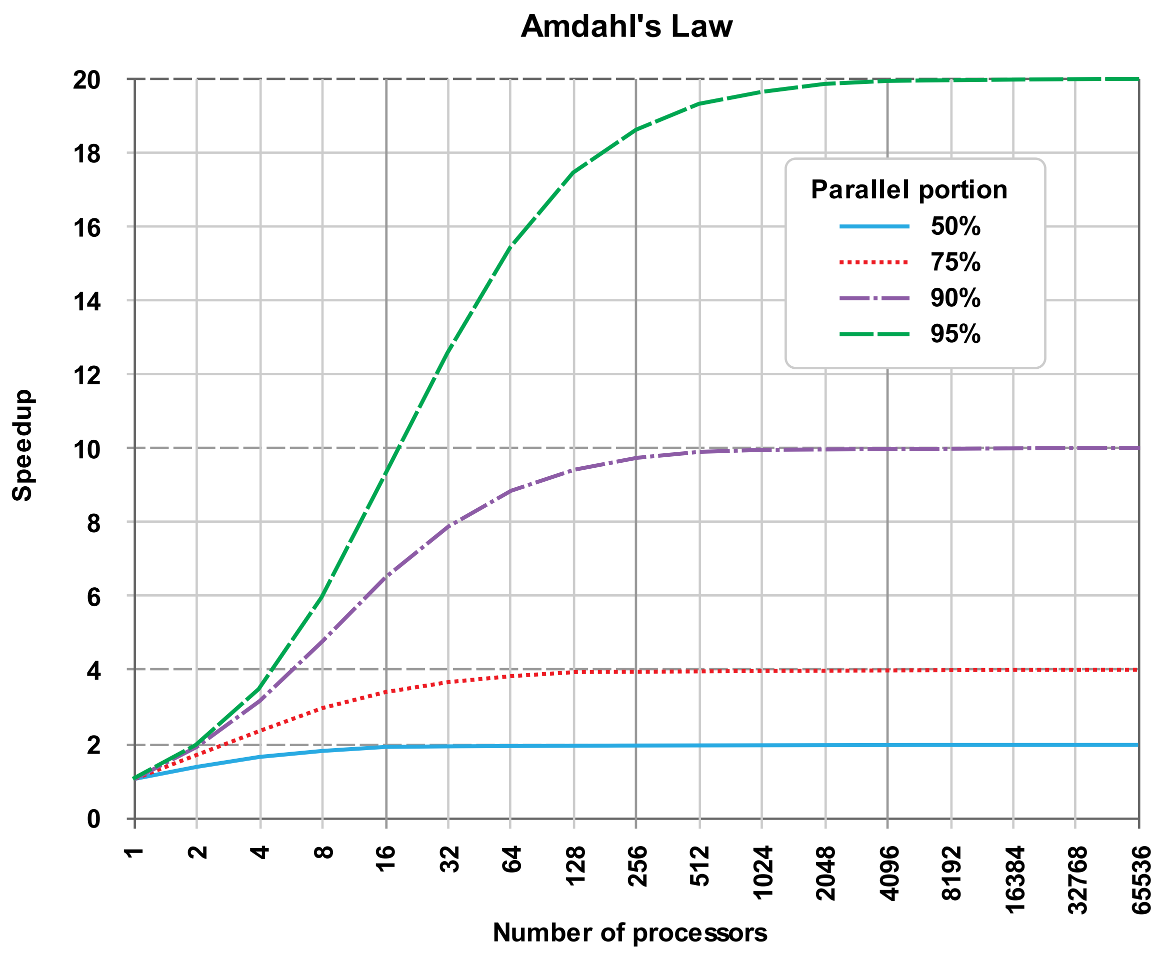


Amdahl’s Law

Formula for theoretical speedup with more processors

Speedup limited by serial part of program:

- e.g. if 95% of program can be parallelized, can be sped up 20 times with enough processors



Thread overheads

- Context switching has costs
- Every time a thread blocks it gets switched:
  - Blocking I/O
  - Contention over locks
  - Condition variables
- Frequently blocking threads reduce throughput

Reducing lock contention

- Serialization hurts scalability
- Context switching hurts performance
- Lock contention hurts both! e.g.:
  - Operation holds a lock for 2 mins
  - All threads must acquire this lock
  - What is the *maximum* throughput we can attain?
    - After a point, more processors make no difference!

Don’t lock for longer than necessary

all this has to be done

```
public class AttributeStore {
    private final Map<String, String>
    attributes = new HashMap<String, String>();

   public synchronized boolean userLocationMatches(String name,
                                                   String regexp) {
      String key = "users." + name + ".location";  // construct key
      String location = attributes.get(key);        // search hashmap
      if (location == null)
         return false;
      else
         return Pattern.matches(regexp, location);  // process results
   }
}
```

From Java Concurrency in Practice, p. 145 (classic book!)

Don't lock for longer than necessary

a lot less to do...

```
public class BetterAttributeStore {
        private final Map<String, String>
        attributes = new HashMap<String, String>();

    public boolean userLocationMatches(String name, String regexp) {
        String key = "users." + name + ".location";
        String location;
        synchronized (this) {
            location = attributes.get(key);
        }
        if (location == null)
            return false;
        else
            return Pattern.matches(regexp, location);
    }
}
```

From Java Concurrency in Practice, p. 145 (classic book!)

Easier—use thread safe collections

```
public class BetterAttributeStore {
        private final ConcurrentHashMap<String, String>
        attributes = new ConcurrentHashMap<String, String>();

    public boolean userLocationMatches(String name, String regexp) {
        String key = "users." + name + ".location";
        String location;

        location = attributes.get(key);

        if (location == null)
            return false;
        else
            return Pattern.matches(regexp, location);
    }
}
```

From Java Concurrency in Practice, p. 145 (classic book!)

Lock splitting & striping

Splitting:

- Using different locks for different functionality within a class
  - e.g. locking methods, individual operations

Striping:

- Using different locks for different parts of the data
  - e.g. breaking a large data structure into smaller parts, use different locks for each part (remember ConcurrentHashMap?)

Lock striping example

```
public class StripedMap {
    // Synchronization policy: buckets[n] guarded by locks[n%N_LOCKS]
    private static final int N_LOCKS = 16;
    private final Node[] buckets;
    private final Object[] locks;

    private static class Node { ….} // stuff missing

    public StripedMap(int numBuckets) {
        buckets = new Node[numBuckets];
        locks = new Object[N_LOCKS];
        for (int i = 0; i < N_LOCKS; i++)
            locks[i] = new Object();
    }
```

each lock guards a portion of the map

From Java Concurrency in Practice, p. 148

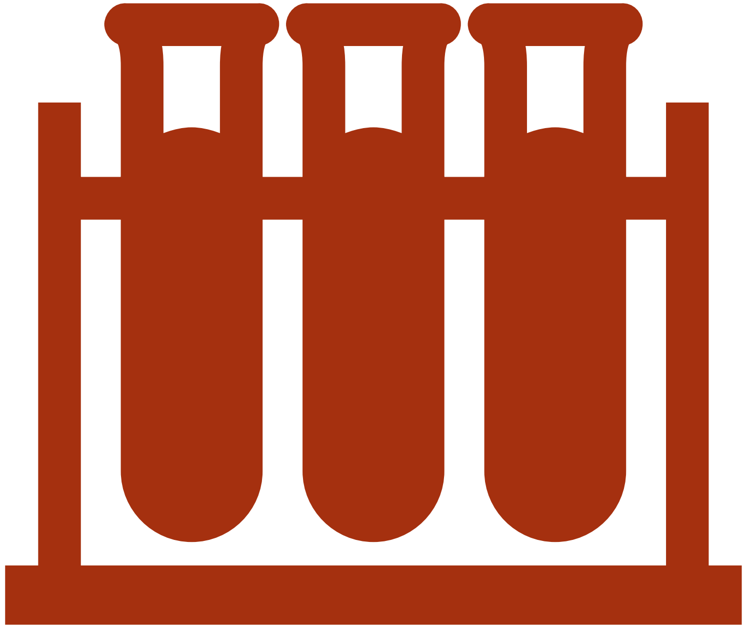
Lock striping example continued

operations can take advantage of locks

```
private final int hash(Object key) {
        return Math.abs(key.hashCode() % buckets.length);
    }
    public Object get(Object key) {
        int hash = hash(key);
        synchronized (locks[hash % N_LOCKS]) {
            for (Node m = buckets[hash]; m != null; m = m.next)
                if (m.key.equals(key))
                    return m.value;
        }
        return null;
    }
    public void clear() { // non atomic clear
        for (int i = 0; i < buckets.length; i++) {
            synchronized (locks[i % N_LOCKS]) {
                buckets[i] = null;
            }
        }
    } }
```

From Java Concurrency in Practice, p. 148

Testing



Testing concurrent programs

- Tricky in the face of non-determinism
- Larger number of potential interactions and failure cases
- Test suites have to be more extensive and run for longer

Testing concurrent programs

Most tests are testing for

- Safety:
  - Nothing bad ever happens
  - Test invariants usually hold
- Liveness:
  - Something good eventually happens
  - Trickier – e.g. testing for deadlocks, race conditions
  - Also throughput, response times, scalability

Testing for correctness

- Similar to testing sequential code
- Identify post conditions and invariants
  - **Post conditions**: state of system after an action has happened
  - **Invariants**: rules that must hold true for the life cycle of an object e.g. you have two boolean fields but only one of them should be true at a time

Example: testing a shared buffer

If buffer has methods for:

- adding and removing items
- checking if it's full /empty

Test:

- New buffer identifies itself as empty
- New buffer identifies itself as not full
- Insert N items into a buffer with capacity N → buffer should be full
- Insert N items into a buffer with capacity N → buffer should be not empty

Bounded buffer tests

```
public class TestBoundedBuffer extends TestCase {

    void testIsEmptyWhenConstructed() {
        SemaphoreBoundedBuffer<Integer> bb = new SemaphoreBoundedBuffer<Integer>(10);
        assertTrue(bb.isEmpty());
        assertFalse(bb.isFull());
    }

    void testIsFullAfterPuts() throws InterruptedException {
        SemaphoreBoundedBuffer<Integer> bb = new SemaphoreBoundedBuffer<Integer>(10);
        for (int i = 0; i < 10; i++)
            bb.put(i);
        assertTrue(bb.isFull());
        assertFalse(bb.isEmpty());
    }
```

From https://jcip.net/listings.html, (listing 12.2)

Testing blocking operations

E.g. a consumer thread tries to take from an empty buffer → should be blocked until the buffer is no longer empty

- Test outcome: thread should not succeed in taking while empty

Testing steps:

1. Try to take an element from an empty buffer
2. Thread should be blocked → put it to sleep for a bit
3. After sleep, if still blocked → interrupt the taker thread
4. If take() succeeds → test fails
5. If thread exits without taking → test succeeds

From the buffer test example

```
void testTakeBlocksWhenEmpty() {
    final SemaphoreBoundedBuffer<Integer> bb = new SemaphoreBoundedBuffer<Integer>(10);
    Thread taker = new Thread() {
        public void run() {
            try {
                int unused = bb.take();
                fail(); // if we get here, it's an error
            } catch (InterruptedException success) { // thread exits
            }
        }
    };
    try {
        taker.start();
        Thread.sleep(LOCKUP_DETECT_TIMEOUT);
        taker.interrupt();
        taker.join(LOCKUP_DETECT_TIMEOUT);
        assertFalse(taker.isAlive());  // verify join returned successfully as thread will be dead
    } catch (Exception unexpected) {
        fail();
    }
}
```

From https://jcip.net/listings.html, (listing 12.3)

From the buffer test example

```
void testTakeBlocksWhenEmpty() {
    final SemaphoreBoundedBuffer<Integer> bb = new SemaphoreBoundedBuffer<Integer>(10);
    Thread taker = new Thread() {
        public void run() {
            try {
                int unused = bb.take();
                fail(); // if we get here, it's an error
            } catch (InterruptedException success) { // thread exits
            }
        }
    };
    try {
        taker.start();
        Thread.sleep(LOCKUP_DETECT_TIMEOUT);
        taker.interrupt();
        taker.join(LOCKUP_DETECT_TIMEOUT);
        assertFalse(taker.isAlive());  // verify join returned successfully as thread will be dead
    } catch (Exception unexpected) {
        fail();
    }
}
```

empty buffer

From https://jcip.net/listings.html, (listing 12.3)

From the buffer test example

```
void testTakeBlocksWhenEmpty() {
    final SemaphoreBoundedBuffer<Integer> bb = new SemaphoreBoundedBuffer<Integer>(10);
    Thread taker = new Thread() {
        public void run() {
            try {
                int unused = bb.take();
                fail(); // if we get here, it's an error
            } catch (InterruptedException success) { // thread exits
            }
        }
    };
    try {
        taker.start();
        Thread.sleep(LOCKUP_DETECT_TIMEOUT);
        taker.interrupt();
        taker.join(LOCKUP_DETECT_TIMEOUT);
        assertFalse(taker.isAlive());  // verify join returned successfully as thread will be dead
    } catch (Exception unexpected) {
        fail();
    }
}
```

”Taker” thread

From https://jcip.net/listings.html, (listing 12.3)

From the buffer test example

```
void testTakeBlocksWhenEmpty() {
    final SemaphoreBoundedBuffer<Integer> bb = new SemaphoreBoundedBuffer<Integer>(10);
    Thread taker = new Thread() {
        public void run() {
            try {
                int unused = bb.take();
                fail(); // if we get here, it's an error
            } catch (InterruptedException success) { // thread exits
            }
        }
    };
    try {
        taker.start();
        Thread.sleep(LOCKUP_DETECT_TIMEOUT);
        taker.interrupt();
        taker.join(LOCKUP_DETECT_TIMEOUT);
        assertFalse(taker.isAlive());  // verify join returned successfully as thread will be dead
    } catch (Exception unexpected) {
        fail();
    }
}
```

The buffer should block this

From https://jcip.net/listings.html, (listing 12.3)

From the buffer test example

```
void testTakeBlocksWhenEmpty() {
    final SemaphoreBoundedBuffer<Integer> bb = new SemaphoreBoundedBuffer<Integer>(10);
    Thread taker = new Thread() {
        public void run() {
            try {
                int unused = bb.take();
                fail(); // if we get here, it's an error
            } catch (InterruptedException success) { // thread exits
            }
        }
    };
    try {
        taker.start();
        Thread.sleep(LOCKUP_DETECT_TIMEOUT);
        taker.interrupt();
        taker.join(LOCKUP_DETECT_TIMEOUT);
        assertFalse(taker.isAlive());  // verify join returned successfully as thread will be dead
    } catch (Exception unexpected) {
        fail();
    }
}
```

Bad

From https://jcip.net/listings.html, (listing 12.3)

From the buffer test example

```
void testTakeBlocksWhenEmpty() {
    final SemaphoreBoundedBuffer<Integer> bb = new SemaphoreBoundedBuffer<Integer>(10);
    Thread taker = new Thread() {
        public void run() {
            try {
                int unused = bb.take();
                fail(); // if we get here, it's an error
            } catch (InterruptedException success) { // thread exits
            }
        }
    };
    try {
        taker.start();
        Thread.sleep(LOCKUP_DETECT_TIMEOUT);
        taker.interrupt();
        taker.join(LOCKUP_DETECT_TIMEOUT);
        assertFalse(taker.isAlive()); // verify join returned successfully as thread will be dead
    } catch (Exception unexpected) {
        fail();
    }
}
```

Good

From https://jcip.net/listings.html, (listing 12.3)

From the buffer test example

```
void testTakeBlocksWhenEmpty() {
      final SemaphoreBoundedBuffer<Integer> bb = new SemaphoreBoundedBuffer<Integer>(10);
      Thread taker = new Thread() {
         public void run() {
            try {
               int unused = bb.take();
               fail(); // if we get here, it's an error
            } catch (InterruptedException success) { // thread exits
            }
         }
      };
      try {
         taker.start();
         Thread.sleep(LOCKUP_DETECT_TIMEOUT);
         taker.interrupt();
         taker.join(LOCKUP_DETECT_TIMEOUT);
         assertFalse(taker.isAlive());  // verify join returned successfully as thread will be dead
      } catch (Exception unexpected) {
         fail();
      }
   }
```

Run the taker → should be blocked

From https://jcip.net/listings.html, (listing 12.3)

From the buffer test example

```
void testTakeBlocksWhenEmpty() {
    final SemaphoreBoundedBuffer<Integer> bb = new SemaphoreBoundedBuffer<Integer>(10);
    Thread taker = new Thread() {
        public void run() {
            try {
                int unused = bb.take();
                fail(); // if we get here, it's an error
            } catch (InterruptedException success) { // thread exits
            }
        }
    };
    try {
        taker.start();
        Thread.sleep(LOCKUP_DETECT_TIMEOUT);
        taker.interrupt();
        taker.join(LOCKUP_DETECT_TIMEOUT);
        assertFalse(taker.isAlive()); // verify join returned successfully as thread will be dead
    } catch (Exception unexpected) {
        fail();
    }
}
```

sleep a little → taker should still be blocked

From https://jcip.net/listings.html, (listing 12.3)

From the buffer test example

```
void testTakeBlocksWhenEmpty() {
    final SemaphoreBoundedBuffer<Integer> bb = new SemaphoreBoundedBuffer<Integer>(10);
    Thread taker = new Thread() {
        public void run() {
            try {
                int unused = bb.take();
                fail(); // if we get here, it's an error
            } catch (InterruptedException success) { // thread exits
            }
        }
    };
    try {
        taker.start();
        Thread.sleep(LOCKUP_DETECT_TIMEOUT);
        taker.interrupt();
        taker.join(LOCKUP_DETECT_TIMEOUT);
        assertFalse(taker.isAlive());  // verify join returned successfully as thread will be dead
    } catch (Exception unexpected) {
        fail();
    }
}
```

interrupt taker

From https://jcip.net/listings.html, (listing 12.3)

From the buffer test example

```
void testTakeBlocksWhenEmpty() {
    final SemaphoreBoundedBuffer<Integer> bb = new SemaphoreBoundedBuffer<Integer>(10);
    Thread taker = new Thread() {
        public void run() {
            try {
                int unused = bb.take();
                fail(); // if we get here, it's an error
            } catch (InterruptedException success) { // thread exits
            }
        }
    };
    try {
        taker.start();
        Thread.sleep(LOCKUP_DETECT_TIMEOUT);
        taker.interrupt();
        taker.join(LOCKUP_DETECT_TIMEOUT);
        assertFalse(taker.isAlive());  // verify join returned successfully as thread will be dead
    } catch (Exception unexpected) {
        fail();
    }
}
```

wait some more to ensure operation completes

From https://jcip.net/listings.html, (listing 12.3)

From the buffer test example

```
void testTakeBlocksWhenEmpty() {
    final SemaphoreBoundedBuffer<Integer> bb = new SemaphoreBoundedBuffer<Integer>(10);
    Thread taker = new Thread() {
        public void run() {
            try {
                int unused = bb.take();
                fail(); // if we get here, it's an error
            } catch (InterruptedException success) { // thread exits
            }
        }
    };
    try {
        taker.start();
        Thread.sleep(LOCKUP_DETECT_TIMEOUT);
        taker.interrupt();
        taker.join(LOCKUP_DETECT_TIMEOUT);
        assertFalse(taker.isAlive());  // verify join returned successfully as thread will be dead
    } catch (Exception unexpected) {
        fail();
    }
}
```

check that taker is dead

From https://jcip.net/listings.html, (listing 12.3)

Testing for race conditions

- Tricky
- Tests need to be multi-threaded, can be complex
- Tests should affect non-determinism as little as possible:
  - Adding synchronization impacts order of execution → can obscure deadlocks

Testing the buffer for race conditions

Test goal:

- Everything put into the buffer comes out (and nothing else)

Basic approach:

- Each producer calculates a checksum for all data it produces
- Each consumer calculates a checksum for all data it receives
- When all producers/consumers complete → checksums should be equal

Testing the buffer for race conditions

Test data should be generated randomly

- Minimizes chances of tests accidentally passing

Example: test by putting integers in the buffer

- Use your own random number generator to generate data to put into the buffer. Why?
  - Built in generators are thread-safe → impact synchronization
  - If each producer generates own random data → doesn't need to be thread safe
  - Create data based on time → different values every test
- We'll see this in code soon...

Testing the buffer for race conditions

To introduce more randomness, coordinate starting and termination of threads:

- Ensure sequential thread operation doesn't introduce some determinism
- Ensure testing of checksums is done after all threads finished
- Use CyclicBarrier to coordinate start and end behavior

Exercise: testing the buffer for race conditions

Spend a few minutes looking at

http://jcip.net/listings/PutTakeTest.java

Easier to find: search "JCIP listings" → scroll to 12.6.

Can you understand how it works?

- We'll walk through it shortly

PutTakeTest.java

```
public class PutTakeTest extends TestCase {
    protected static final ExecutorService pool = Executors.newCachedThreadPool();
    protected CyclicBarrier barrier;
    protected final SemaphoreBoundedBuffer<Integer> bb;
    protected final int nTrials, nPairs;
    protected final AtomicInteger putSum = new AtomicInteger(0);
    protected final AtomicInteger takeSum = new AtomicInteger(0);

    public static void main(String[] args) throws Exception {
        new PutTakeTest(10, 10, 100000).test(); // sample parameters
        pool.shutdown();
    }

    public PutTakeTest(int capacity, int npairs, int ntrials) {
        this.bb = new SemaphoreBoundedBuffer<Integer>(capacity);
        this.nTrials = ntrials;
        this.nPairs = npairs;
        this.barrier = new CyclicBarrier(npairs * 2 + 1); // initialize the barrier  +1 for main thread
    }
```

cached pool

PutTakeTest.java

```
public class PutTakeTest extends TestCase {
    protected static final ExecutorService pool = Executors.newCachedThreadPool();
    protected CyclicBarrier barrier;
    protected final SemaphoreBoundedBuffer<Integer> bb;
    protected final int nTrials, nPairs;
    protected final AtomicInteger putSum = new AtomicInteger(0);
    protected final AtomicInteger takeSum = new AtomicInteger(0);

    public static void main(String[] args) throws Exception {
        new PutTakeTest(10, 10, 100000).test(); // sample parameters
        pool.shutdown();
    }

    public PutTakeTest(int capacity, int npairs, int ntrials) {
        this.bb = new SemaphoreBoundedBuffer<Integer>(capacity);
        this.nTrials = ntrials;
        this.nPairs = npairs;
        this.barrier = new CyclicBarrier(npairs * 2 + 1); // initialize the barrier  +1 for main thread
    }
```

to coordinate start/temination

PutTakeTest.java

```
public class PutTakeTest extends TestCase {
    protected static final ExecutorService pool = Executors.newCachedThreadPool();
    protected CyclicBarrier barrier;
    protected final SemaphoreBoundedBuffer<Integer> bb;
    protected final int nTrials, nPairs;
    protected final AtomicInteger putSum = new AtomicInteger(0);
    protected final AtomicInteger takeSum = new AtomicInteger(0);

    public static void main(String[] args) throws Exception {
        new PutTakeTest(10, 10, 100000).test(); // sample parameters
        pool.shutdown();
    }

    public PutTakeTest(int capacity, int npairs, int ntrials) {
        this.bb = new SemaphoreBoundedBuffer<Integer>(capacity);
        this.nTrials = ntrials;
        this.nPairs = npairs;
        this.barrier = new CyclicBarrier(npairs * 2 + 1); // initialize the barrier  +1 for main thread
    }
```

to calculate checksums

PutTakeTest.java

```
public class PutTakeTest extends TestCase {
    protected static final ExecutorService pool = Executors.newCachedThreadPool();
    protected CyclicBarrier barrier;
    protected final SemaphoreBoundedBuffer<Integer> bb;
    protected final int nTrials, nPairs;
    protected final AtomicInteger putSum = new AtomicInteger(0);
    protected final AtomicInteger takeSum = new AtomicInteger(0);

    public static void main(String[] args) throws Exception {
        new PutTakeTest(10, 10, 100000).test(); // sample parameters
        pool.shutdown();
    }

    public PutTakeTest(int capacity, int npairs, int ntrials) {
        this.bb = new SemaphoreBoundedBuffer<Integer>(capacity);
        this.nTrials = ntrials;
        this.nPairs = npairs;
        this.barrier = new CyclicBarrier(npairs * 2 + 1); // initialize the barrier  +1 for main thread
    }
```

we know exactly how many threads there will be

PutTakeTest.java

The test entry point (main thread):

```
void test() {
      try {
         for (int i = 0; i < nPairs; i++) {   //  create the threads
            pool.execute(new Producer());
            pool.execute(new Consumer());
         }
         barrier.await(); // wait for all threads to be ready
         barrier.await(); // wait for all threads to finish
         assertEquals(putSum.get(), takeSum.get());
      } catch (Exception e) {
         throw new RuntimeException(e);
      }
   }
```

2 await() calls in main thread

PutTakeTest.java: Producer

```
class Producer implements Runnable {
      public void run() {
          try {
              int seed = (this.hashCode() ^ (int) System.nanoTime());
              int sum = 0;
              barrier.await();
              for (int i = nTrials; i > 0; --i) {
                  bb.put(seed);
                  sum += seed;
                  seed = xorShift(seed);
              }
              putSum.getAndAdd(sum);
              barrier.await();
          } catch (Exception e) {
              throw new RuntimeException(e);
          }
      }
  }
```

executed nPair times by main thread

2 await calls per Producer thread

PutTakeTest.java: Producer

```
class Producer implements Runnable {
    public void run() {
        try {
            int seed = (this.hashCode() ^ (int) System.nanoTime());
            int sum = 0;
            barrier.await();
            for (int i = nTrials; i > 0; --i) {
                bb.put(seed);
                sum += seed;
                seed = xorShift(seed);
            }
            putSum.getAndAdd(sum);
            barrier.await();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

seed for simple custom random number generator

- uses system time to ensure randomness

PutTakeTest.java: Producer

```
class Producer implements Runnable {
      public void run() {
          try {
              int seed = (this.hashCode() ^ (int) System.nanoTime());
              int sum = 0;
              barrier.await();
              for (int i = nTrials; i > 0; --i) {
                  bb.put(seed);
                  sum += seed;
                  seed = xorShift(seed);
              }
              putSum.getAndAdd(sum);
              barrier.await();
          } catch (Exception e) {
              throw new RuntimeException(e);
          }
      }
  }
```

generate nTrials random numbers and put them in the buffer

PutTakeTest.java: Producer

```
class Producer implements Runnable {
    public void run() {
        try {
            int seed = (this.hashCode() ^ (int) System.nanoTime());
            int sum = 0;
            barrier.await();
            for (int i = nTrials; i > 0; --i) {
                bb.put(seed);
                sum += seed;
                seed = xorShift(seed);
            }
            putSum.getAndAdd(sum);
            barrier.await();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

track the sum of all random data added to the buffer

PutTakeTest.java: Consumer

```
class Consumer implements Runnable {
        public void run() {
            try {
                barrier.await();
                int sum = 0;
                for (int i = nTrials; i > 0; --i) {
                    sum += bb.take();
                }
                takeSum.getAndAdd(sum);
                barrier.await();
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
```

executed nPair times by main thread

2 await calls per Consumer thread

PutTakeTest.java: Consumer

```
class Consumer implements Runnable {
    public void run() {
        try {
            barrier.await();
            int sum = 0;
            for (int i = nTrials; i > 0; --i) {
                sum += bb.take();
            }
            takeSum.getAndAdd(sum);
            barrier.await();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

take out nTrials data

track all the received data

PutTakeTest.java

Back to the main thread...

```
void test() {
    try {
      for (int i = 0; i < nPairs; i++) {   //  create the threads
        pool.execute(new Producer());
        pool.execute(new Consumer());
      }
      barrier.await(); // wait for all threads to be ready
      barrier.await(); // wait for all threads to finish
      assertEquals(putSum.get(), takeSum.get());
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }
```

the actual test

- data in matches data out

Wrap up: testing concurrent code

- Test cases can be more complex than code
  - See previous example
- Other things to test:
  - Resource management (see testLeak in http://jcip.net/listings, 12.2)
  - Performance
    - Add timing information
  - Scalability
    - Requires large volume tests, coordination of more resources

Want to learn more?

- Review testing code listings at http://jcip.net/listings
- Take Dr. Gorton’s Distributed Systems class

Summary

- Threads can be used to parallelize independent loop iterations & independent recursive calls
- Scalability requires careful design & is always limited by serialization
- Testing concurrent systems is tricky due to large amount of failure modes & non-determinism