Scaling Java Applications Through Concurrency

Patterns over Primitives

Josh Cummings

@jzheaux | tech.joshcummings.com

The complexity for minimum component costs has increased at a rate of roughly a factor of two per year.

- *Gordon Moore, 19 April 1965*

Moore’s Law Saves the Day

Jon develops software

Jon deploys to hardware that runs at X Ghz

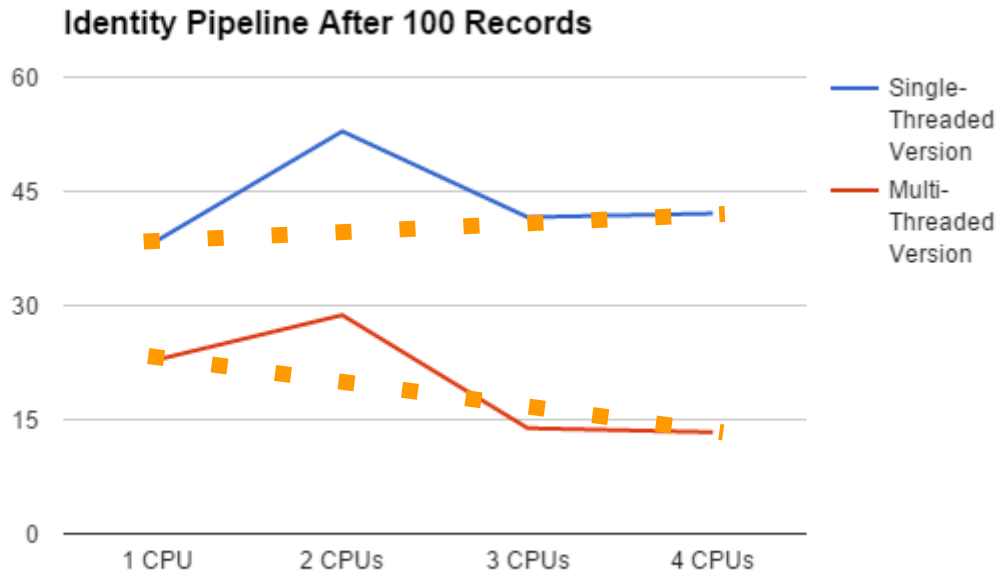
Jon’s software runs at speed Y

Moore’s Law increases hardware speeds ten-fold in six years

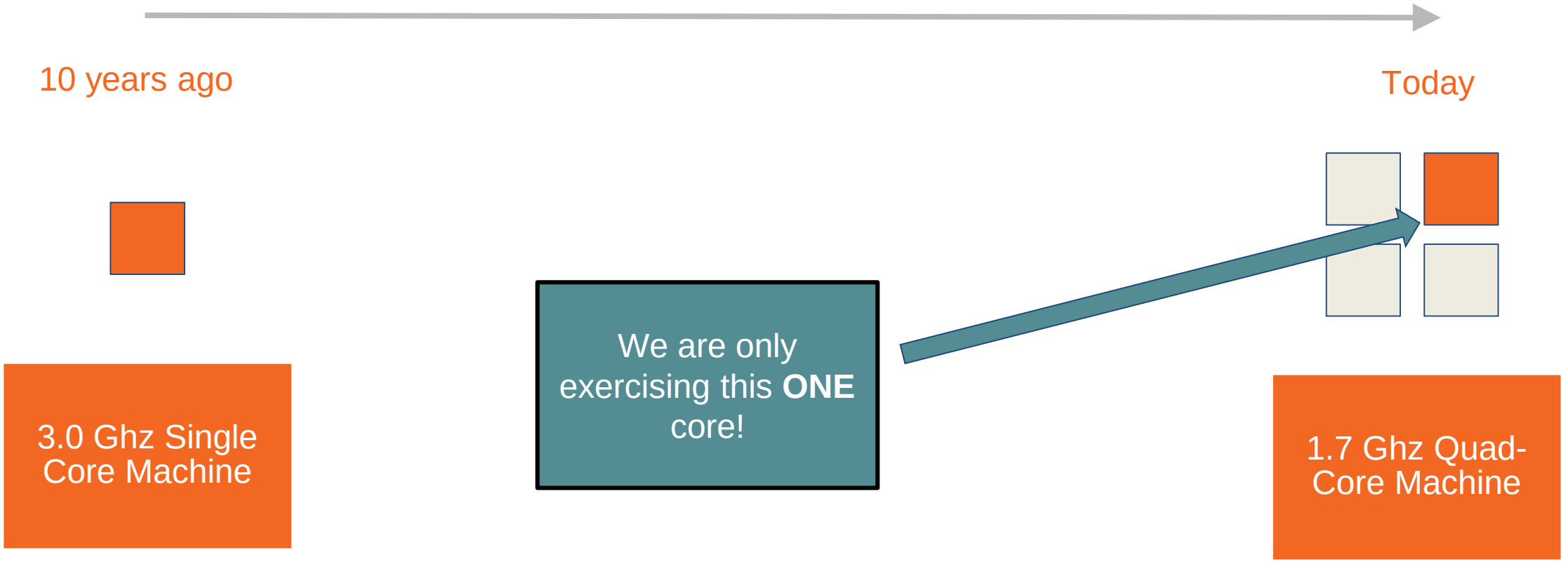
Jon deploys to hardware that runs at 10X Ghz

Jon’s software gets a “free” performance boost!

Moore’s Law No Longer Comes For Free



How does 10-year old hardware work for you?



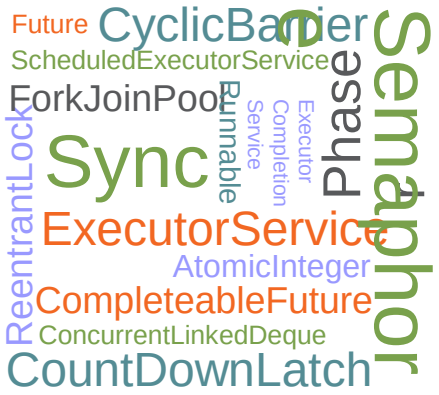
It could
actually run
SLOWER

I see Moore's Law dying out in the next decade or so.

- *Gordon Moore, 30 March 2015*

Low-level thread orchestration is challenging to get right and ought to be avoided if possible. Look for established coding patterns in order to stay out of the minutae.

There is hope...



...we have the Java Concurrency API!

Scaling Java Applications Through Concurrency

Running Processes in the Background

Josh Cummings

@jzheaux | tech.joshcummings.com

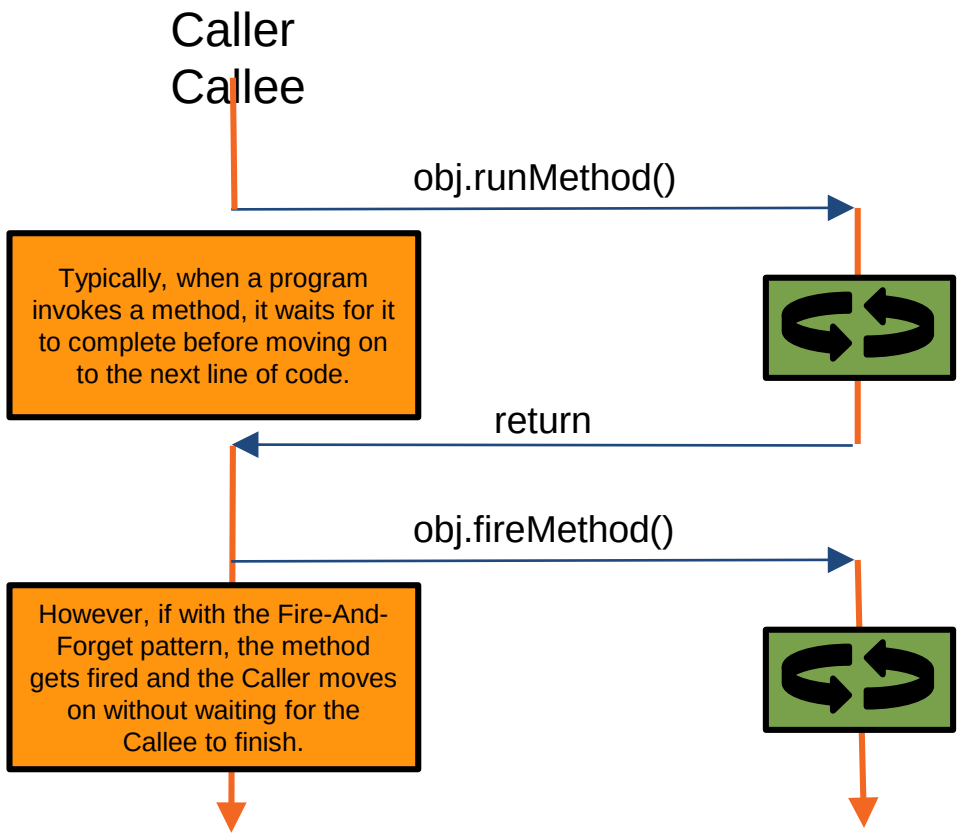
Identity Pipeline: An Integrated Example of Java Concurrency Patterns

- Reads in a stream of identity-related data
- Normalizes and verifies the contents of each identity
- Merges and persists the identities in an in-memory cache
- Records aggregate identity statistics
- Notifies an error queue if any of the above fails

Identity Pipeline: Background Processes

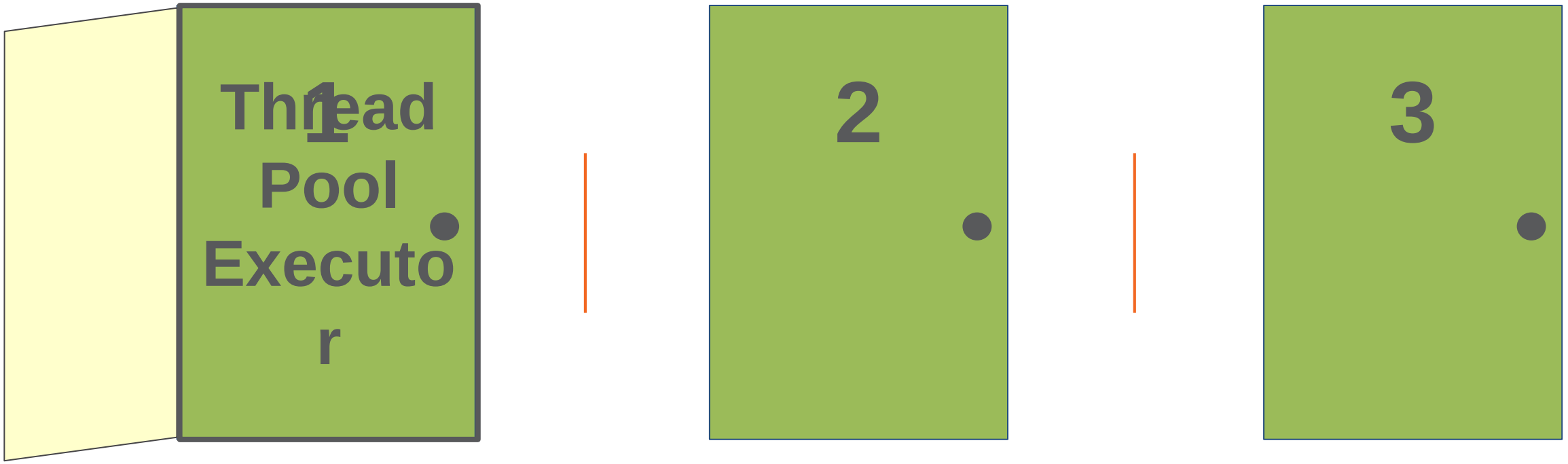
- Reads in a stream of identity-related data
- Normalizes and verifies the contents of each identity
- Merges and persists the identities in an in-memory cache
- Records aggregate identity statistics
- **Notifies an error queue if any of the above fails**

The Fire-And-Forget Pattern



- Operating on a single-thread bounds the application's performance to the slowest instructions
- Anticipating a reply doubles the work that needs to be done
- Fire-And-Forget addresses both

Options Found in the Concurrency API



ThreadPoolExecutor

- Maintains a pool of threads for rapid reuse
- Abstracts away the use of Thread
- Results may either be blocked on, ignored, or delegated
- Never blocks on task submission, even if the thread pool size is maxxed out

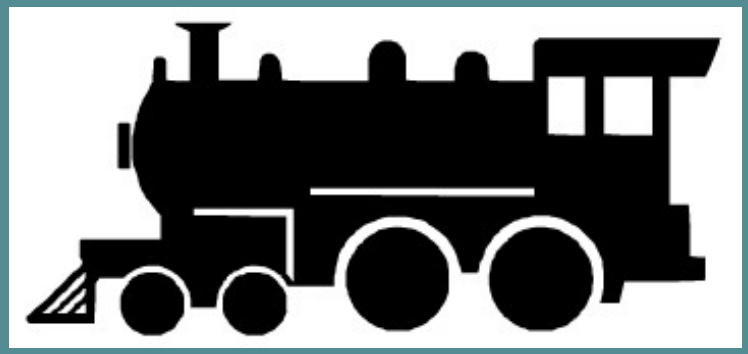
```
// each returns an appropriately
configured // instance of
ThreadPoolExecutor

ExecutorService pool =
Executors.newCachedThreadPool();

ExecutorService pool =
Executors.newFixedThreadPool(10);

ExecutorService single =
Executors.newSingleThreadExecutor();
```

Cached

Light-weight asychronous tasks

Threads are automatically created if no thread is available for a task

Idle threads are re-used and will terminate after 60 seconds of inactivity



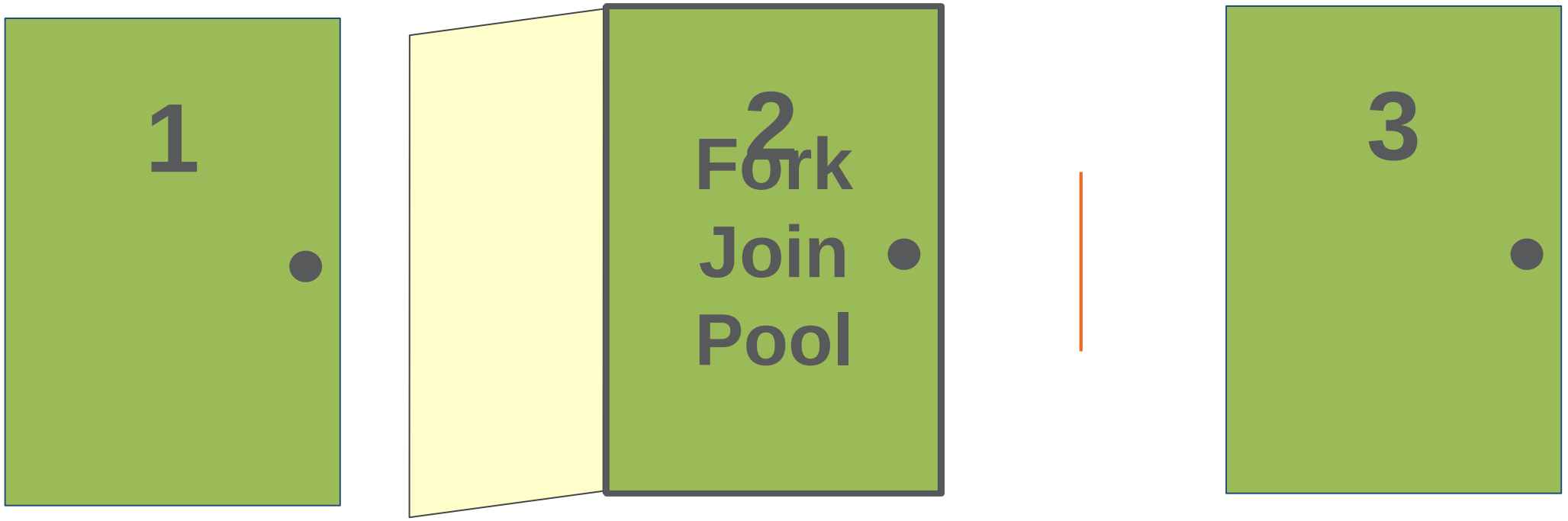
Fixed

Heavy, long-running use

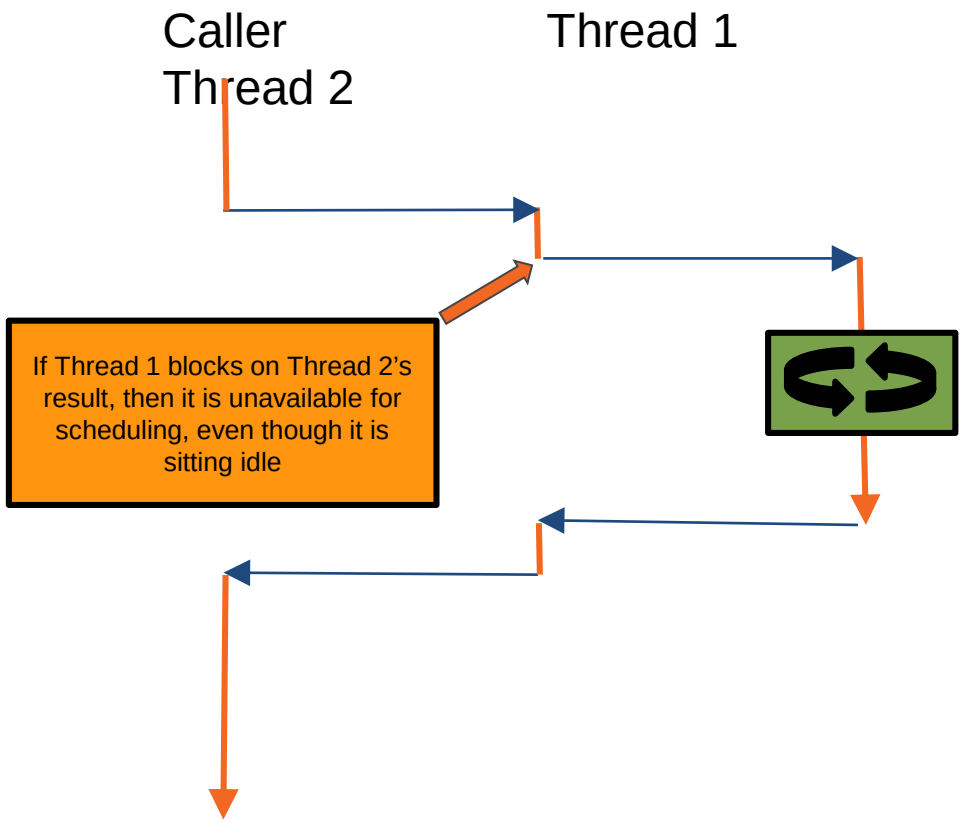
A fixed number of threads is constantly maintained regardless of workload

Threads will be kept active until they are explicitly shut down

Options Found in the Concurrency API



What to do about idling threads?



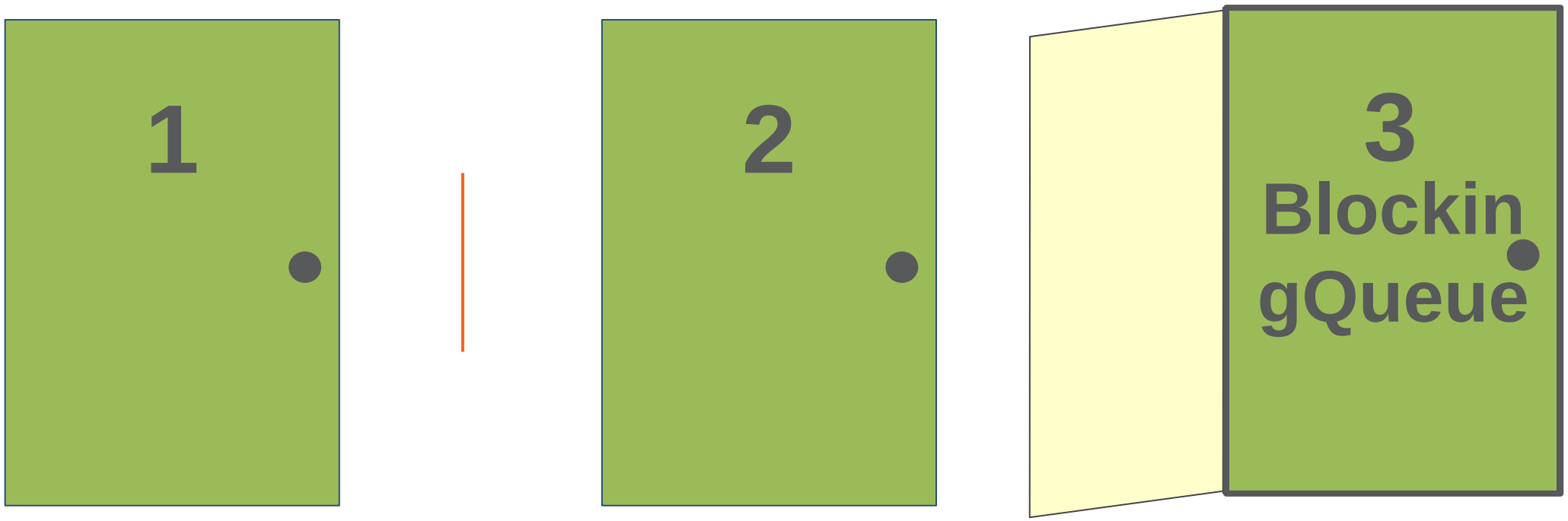
- asynchronous != non-blocking (consider the case of a parallelized sorting algorithm)
- Threads may need to block, waiting for other threads to finish
- In the case of thread pools, this is sub-optimal because a blocking thread cannot be scheduled even though it is not doing work

ForkJoinPool

- Identical benefits to ThreadPoolExecutor
- Processes are daemon threads, meaning that the main execution thread will not wait for them to complete when the JVM shuts down.
- Processes may steal work from one another, which is nice for threaded recursion, e.g. in the case of Divide and Conquer algorithms

```
// each returns an appropriately
configured // instance of ForkJoinPool

ExecutorService pool =
Executors.newWorkStealingPool();

ExecutorService pool =
ForkJoinPool.common();

// # in pool should be a power of two
ExecutorService single = new
ForkJoinPool(16);
```

Options Found in the Concurrency API



A Bit More on ForkJoinPool

If parallelizing...	… then extend ...
recursion with results (e.g. sorting, searching)	RecursiveTask
recursion with no results (e.g. fire-and-forget)	RecursiveAction

BlockingQueue

- Implementations of ExecutorService follow the producer-consumer paradigm, using a queue as the intermediate data structure

- BlockingQueue abstracts away direct use of wait and notify, avoiding common pitfalls

- BlockingQueue has a full complement of produce and consume methods that block, don't block, or throw exceptions

```
// a list of tasks to be completed at the
// disposal of any number of consumers

BlockingQueue<Task> tasks = new
LinkedBlockingQueue<>();

tasks.offer(task); // adds to the end
without blocking

tasks.take(); // blocks until there is
something in the queue
```

Pro Tip: Debugging Threads

- Name thread pools for easier debugging (use Google Guava)
- Name threads with debug information for more meaningful thread dumps

```
// name your thread pool

ExecutorService pool =
        Executors.newCachedThreadPool(
new
                ThreadFactoryBuilder()

        .setNameFormat("Malformed-%d")
                .build());


// provide context in thread name


Thread.currentThread().setName("Malform
ed-identity-" + identity.getId());
```

Review

- Fire-and-forget is a simple pattern that can be implemented by firing an independent process on a separate thread
- Implementations of **ExecutorService** abstract away thread management
- **ForkJoinPool** is useful for threaded recursion
- Java **Queues** can also be used to queue up tasks in a thread-safe fashion independent of a thread pool

Scaling Java Applications Through Concurrency

Sharing Resources Among Parallel Workers

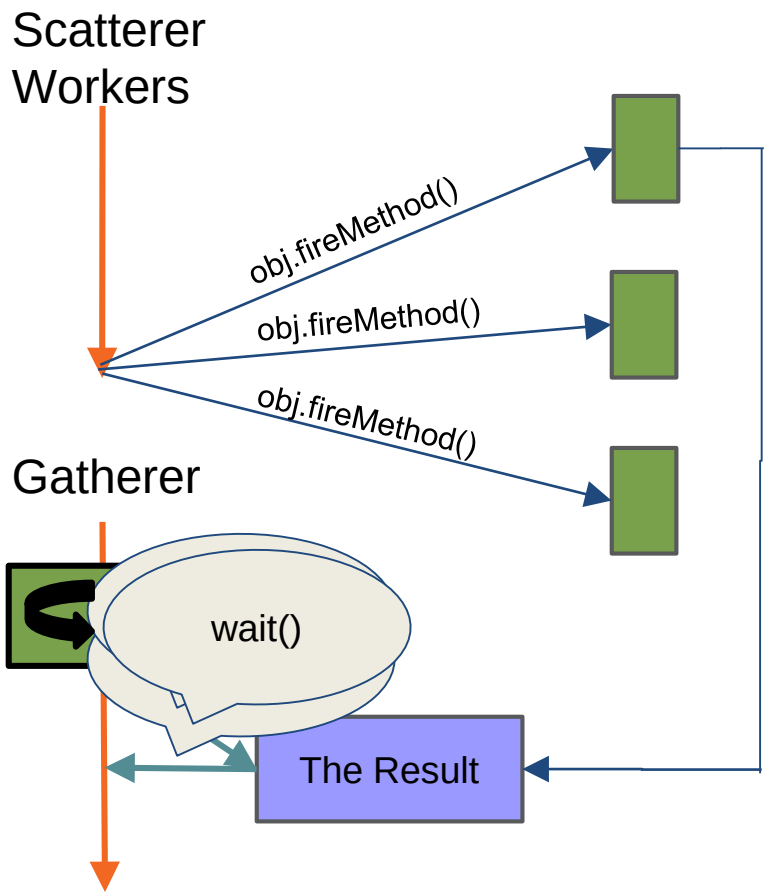
Josh Cummings

@jzheaux | tech.joshcummings.com

Identity Pipeline: Parallel Workers

- **Reads in a stream of identity-related data**
- Normalizes and verifies the contents of each identity
- Merges and persists the identities in an in-memory cache
- Records aggregate identity statistics
- Notifies an error queue if any of the above fails

Scatter-Gather Pattern



- Ask several independent workers their vote or opinion on the same question, short-circuiting when a quorum is reached
- Segment work across several workers
- Ask several workers to perform the same operation for reliability or performance guarantee
- Workers operate in parallel for greater scalability

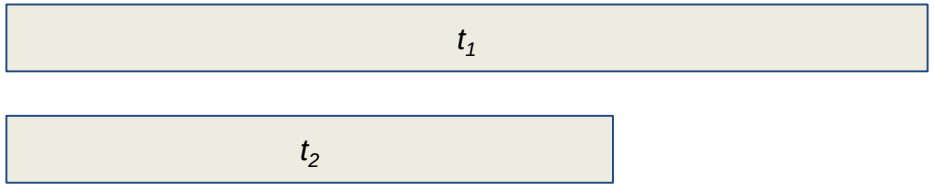
Future

- An Object that represents the result that a Worker will eventually provide
- Allows the submitter to block on Workers' results
- Futures can be returned to layers that are prepared to block on the response

```
// deploy a worker thread, storing a Future
// for later blocking

Future<Identity> identityHolder = pool.submit(() -> reader.read(source));

Identity identity = identityHolder.get(); // blocks until thread completes

List<Future<Identity>> identitiesHolder = pool.invokeAll(listOfCallables );
```

ExecutorCompletionService

- Adds completed tasks to a queue from which results can be retrieved in the order that they were completed
- Reduces real time execution



$t = t_1 + p_1 + p_2$ *(block on first)*

$t = t_2 + p_2 + (t_1 - t_2 - p_2) + p_1 = t_1 + p_1$ *(block on shortest)*

```
ExecutorCompletionService ecs = new
ExecutorCompletionService(pool);

// deploy several worker threads
for ( Task t : tasks ) {
    ecs.submit(() -> reader.read(source));
}

Identity identity = ecs.take().get(); //
blocks until first thread completes
```

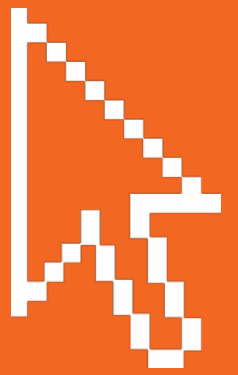
Identity Pipeline: Parallel Workers

- Reads in a stream of identity-related data
- Normalizes and verifies the contents of each identity
- Merges and persists the identities in an in-memory cache
- **Records aggregate identity statistics**
- Notifies an error queue if any of the above fails

Thread-safe Containers

NOT Thread Safe	Thread Safe
ArrayList	ConcurrentLinkedQueue, BlockingQueue
HashMap	ConcurrentHashMap
incrementation	AtomicInteger, LongAdder

Compound statements are not atomic ⇒ They are not naturally thread-safe



ReentrantLock

- Abstracts the monitor into an object, allowing acquisition and release patterns other than simple code blocks
- Allows for multiple Conditions to be created from a single lock for more sophisticated waiting patterns.
- Maintains the same atomicity and visibility semantics as **synchronized**
- May be faster than the **synchronized** keyword.

```
Lock lock = new ReentrantLock();
Condition circumstance =
lock.newCondition();

// wrapping an unsafe invocation in a lock
// identical to wrapping in synchronized
public void guardedInvocation() {
        lock.lock();
        try {

        obj.threadUnsafeInvocation();
        } finally {
                lock.unlock();
        }
}
```

Identity Pipeline: Parallel Workers

- Reads in a stream of identity-related data
- Normalizes and verifies the contents of each identity
- **Merges and persists the identities in an in-memory cache**
- Records aggregate identity statistics
- Notifies an error queue if any of the above fails

ReentrantLock#tryLock

- Requests lock, returning immediately if the lock is already held
- Usage may allow thread to “barge” past other threads currently waiting on lock() to return

```
ReentrantLock lock = new ReentrantLock();

// the underlying invocation will only occur
// if both locks can be acquired
public void complexInvocation() {
        if ( lock1.tryLock() && lock2.tryLock()
) {
                try {

        obj.threadUnsafeInvocation();
                } finally {
                        lock1.unlock();
                        lock2.unlock();
                }
        }
}
```

Review

- Scatter-Gather is a pattern for splitting work among several workers
- **ExecutorCompletionService** and **Future** can facilitate promise architectures
- Java **Concurrentxxx**, **Atomicxxx**, and **ReentrantLock** offer richer semantics than standard concurrency primitives

Parallel Streams

- Automatically distributes tasks across the common fork join pool
- Short-circuits on the first task to return

```
// deploy several workers
Optional<Identity> maybe =
readers.parallelStream()
        .map((reader) ->
reader.read(source))
// short-circuit on the first response
        .findAny();


Identity identity = maybe.get();
```

Scaling Java Applications Through Concurrency

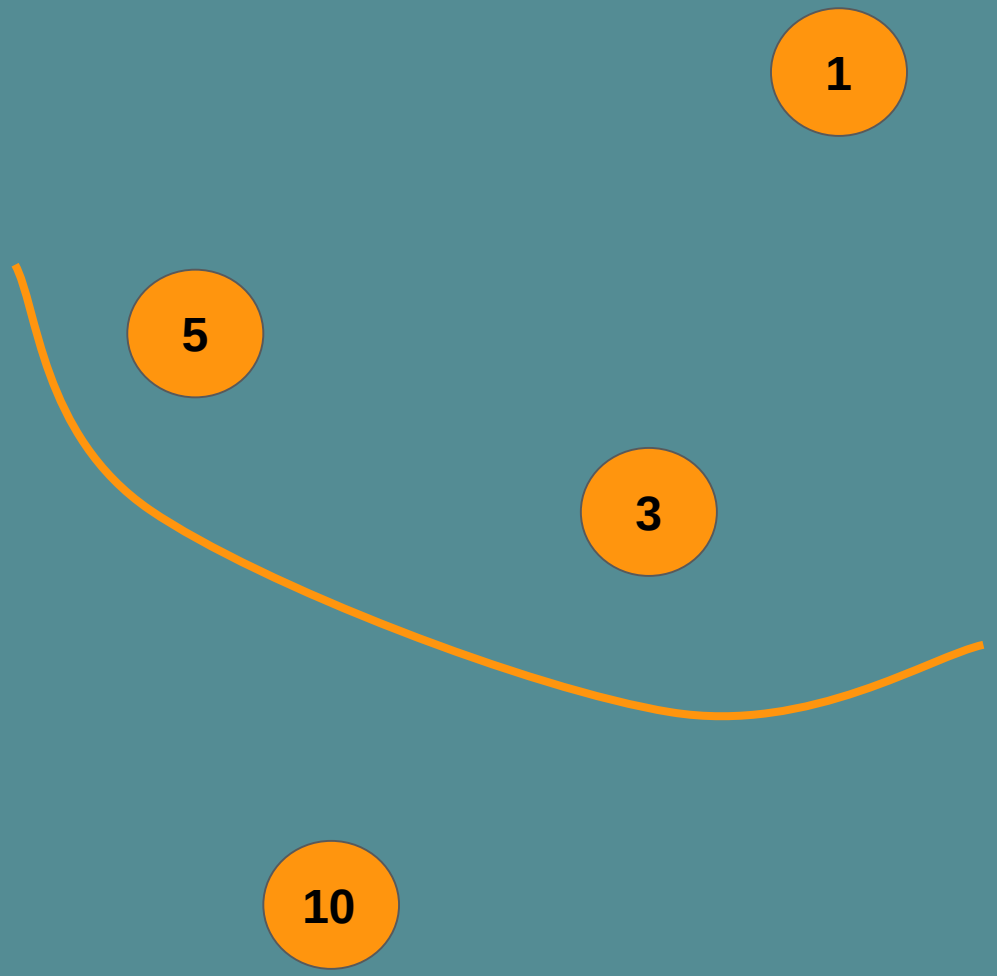
Coordinating Efforts Among Dependent Processes

Josh Cummings

@jzheaux | tech.joshcummings.com

Identity Pipeline: Dependencies

- **Reads in a stream of identity-related data**
- **Normalizes and verifies the contents of each identity**
- Merges and persists the identities in an in-memory cache
- Notifies an error queue if any of the above fails
- Records aggregate identity statistics



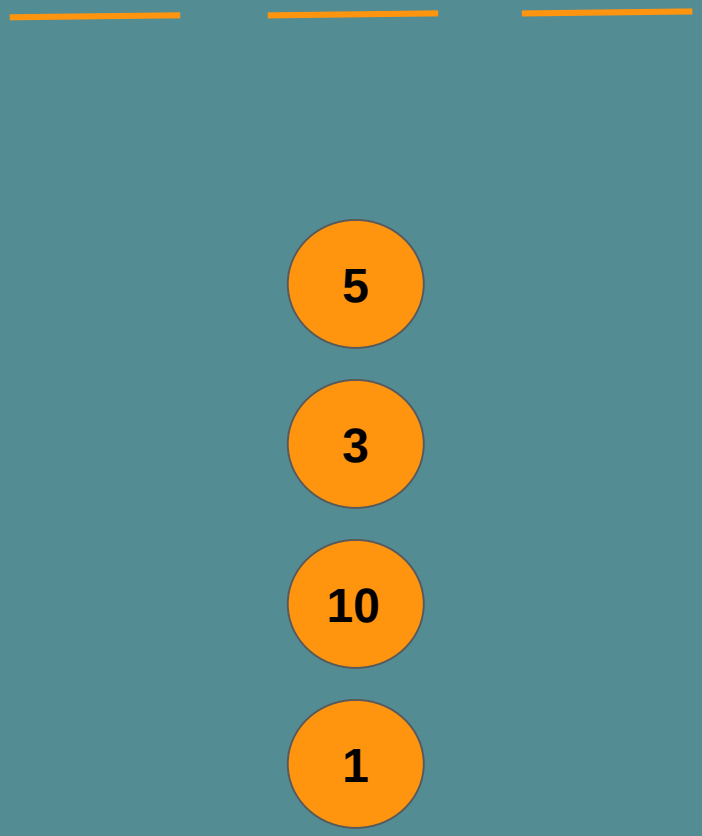

```
// In this case, order doesn't really matter;
// however it is implicit. verifyEmailAddress
// cannot be called until verifyPhoneNumber
// is finished


boolean pVerified =
verifyPhoneNumber(identity);
boolean eVerified =
verifyEmailAddress(identity);
boolean aVerified = verifyAddresses(identity);


// A simple way to multi-thread this is to
// introduce multiple workers


pool.submit(() -> verifyPhoneNumber(identity));
pool.submit(() -> verifyEmailAddress(identity));
pool.submit(() -> verifyAddresses(identity));
```

// How to make sure nothing happens until all

CountDownLatch

- Similar to, but richer semantics than **join**
- Threads wait for a programmatically decremented count down before continuing
- Helpful when one thread is dependent on the completion or arrival of several other threads before moving on

```
// guests arrive on their own and the waiter
// waits for everyone to arrive

CountDownLatch orchestrator = new
CountDownLatch(numberOfActors);


public void perform(Actor actor) {
        waiter.countDown(); // sing song,
etc...
}

public void sendNextActor()
                throws InterruptedException
{
        orchestrator.await(); // send now...
}
```




```
// In this case, order doesn't really matter;
// however it is implicit. verifyEmailAddress
// cannot be called until verifyPhoneNumber
// is finished


boolean pVerified =
verifyPhoneNumber(identity);
boolean eVerified =
verifyEmailAddress(identity);
boolean aVerified = verifyAddresses(identity);


// A simple way to multi-thread this is to
// introduce multiple workers


pool.submit(() -> verifyPhoneNumber(identity));
pool.submit(() -> verifyEmailAddress(identity));
pool.submit(() -> verifyAddresses(identity));
```

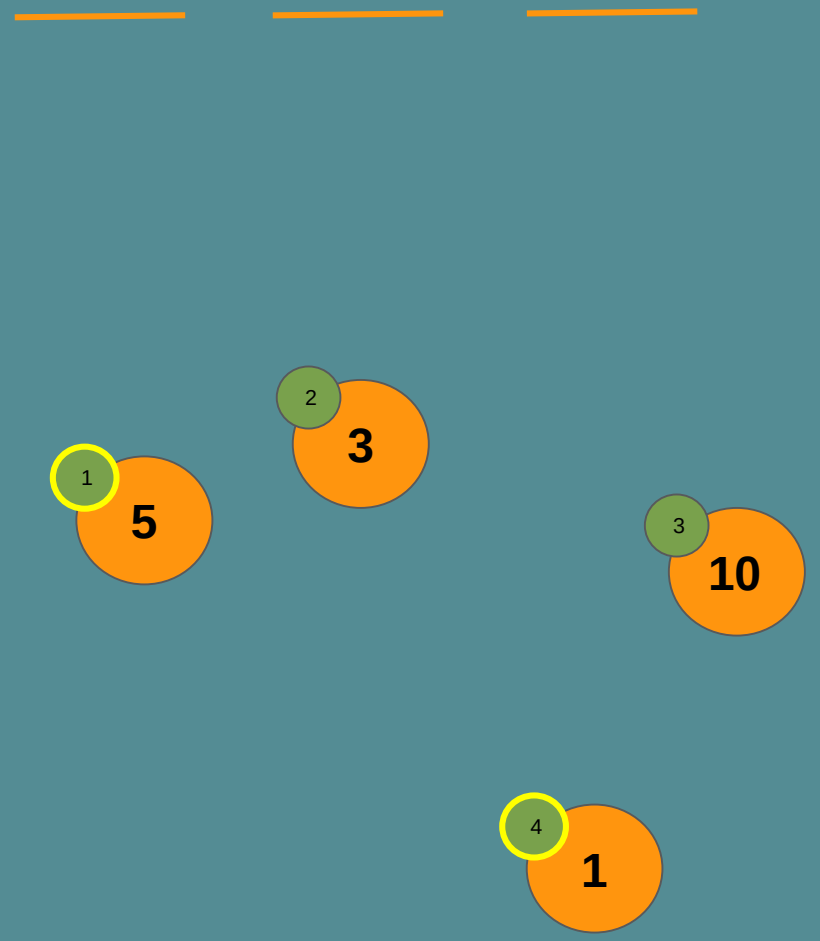
Why stay in line?

Benefits of promises

Order represented, but not rigid

Tickets can be obtained by one customer and then handed to another customer

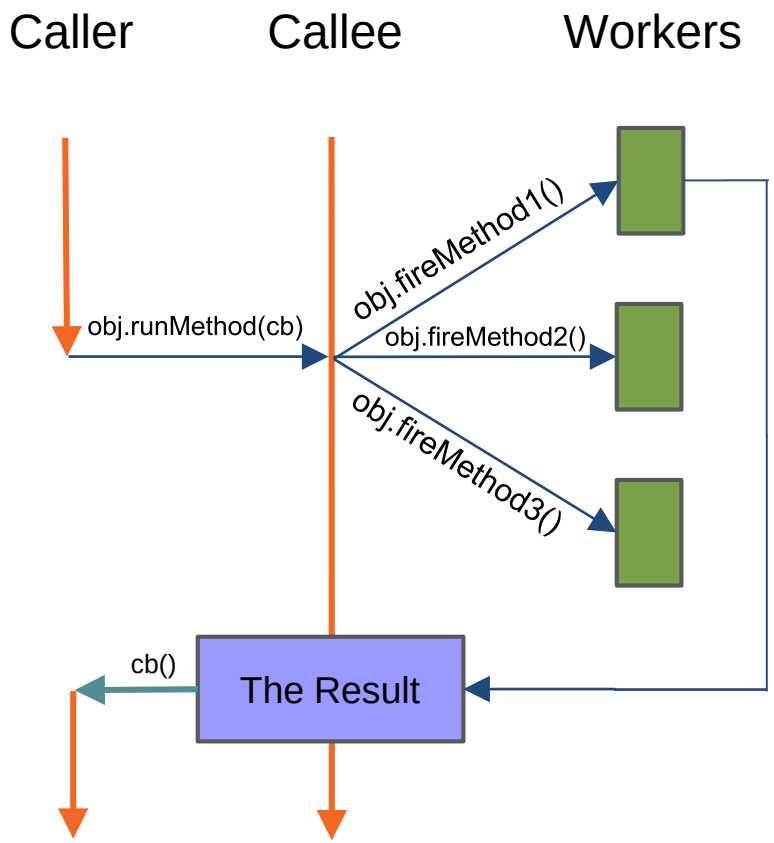
Customers can do something else while they wait (e.g. leave the waiting room and come back when it's their turn)




```
// Futures afford the abstraction of a callback
// instead in a return value


Future<Boolean> pResult = pool.submit(() ->
        verifyPhoneNumber(identity));
Future<Boolean> eResult = pool.submit(() ->
        verifyEmailAddress(identity));
Future<Boolean> aResult = pool.submit(() ->
        dresses(identity));

// A promise affords the possibility of waiting
// on results independent of execution order


if ( eResult.get() && aResult.get() &&
pResult.get() ) {
        // do something amazing here
}
```

The order of execution is independent from the completion order

Continuation-Passing Pattern



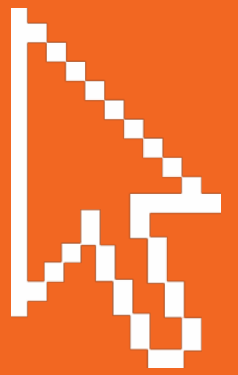
- Caller invokes method, providing a callback method
- Caller does not need to block on thread completion
- Callee can easily make the choice, even at runtime, to execute caller's request concurrently

Method References & Lambdas

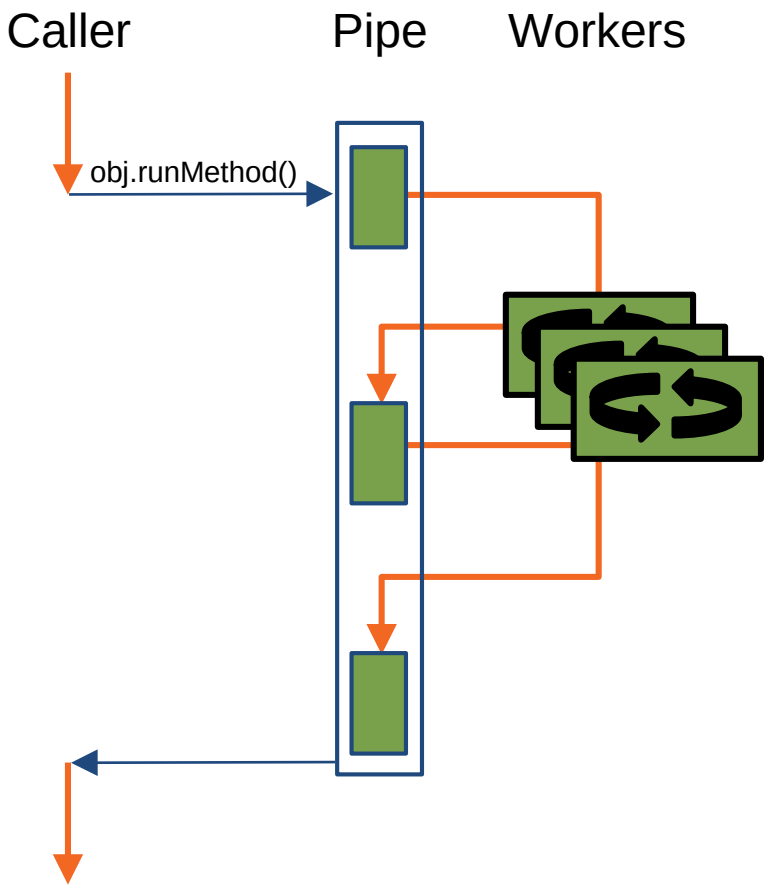
- Simple way to specify a callback to a downstream execution
- The caller can now do other things at the same time while waiting for the code to finish
- Passing more than one callback allows the caller to choreograph execution as more of a state transition diagram

```
// allow the callee to notify the caller when it
// is complete by using a callback

format(identity, this::fail, this::persist);

void fail(Identity identity, Exception e) {
  malformed.addIdentity(identity, e);
}


void persist(Identity identity) {
  doPersist(identity, this::fail, this::doStats);
}
```

Continuations typically return immediately, but that doesn't mean the work is done



Pipeline Pattern



- Flattens out otherwise deeply-nested callbacks inherent in the Continuation-Passing pattern
- Abstracts away dependencies between asynchronous tasks
- Affords the same benefits of ignoring, delegating, or blocking on the result as Futures

CompletableFuture

- Fluent API allows for chain of responsibility, similar to the Java Stream API
- Uses the ForkJoin common pool, though each method takes an **ExecutorService** as a parameter for easy configuration.
- Semantics similar to **Future**; fire-and-forget, block, or delegate the result

```
// run one task after another completes
// asynchronously

CompletableFuture c =
    CompletableFuture.runAsync(
            () -> str.toUpperCase())
            .thenAcceptAsync(
                    (upperCased) ->

            upperCased.replace("a", "b"), pool)
            .exceptionally(
                    (e) ->
                            log.error(e)
            );
```

Asynchronous Composition

Pattern	… using CompletableFuture
Fire-and-Forget	.runAsync
Foot-Race	.anyOf → .thenAcceptAsync
Scatter-Gather	.allOf → .thenAcceptAsync

Review

- Dependencies can be managed either from an orchestration or choreography standpoint
- CountDownLatch, Continuation-Passing, and CompletableFuture are all ways to think choreographically about concurrent dependencies
- Consider partitioning thread pools to increase throughput

Scaling Java Applications Through Concurrency

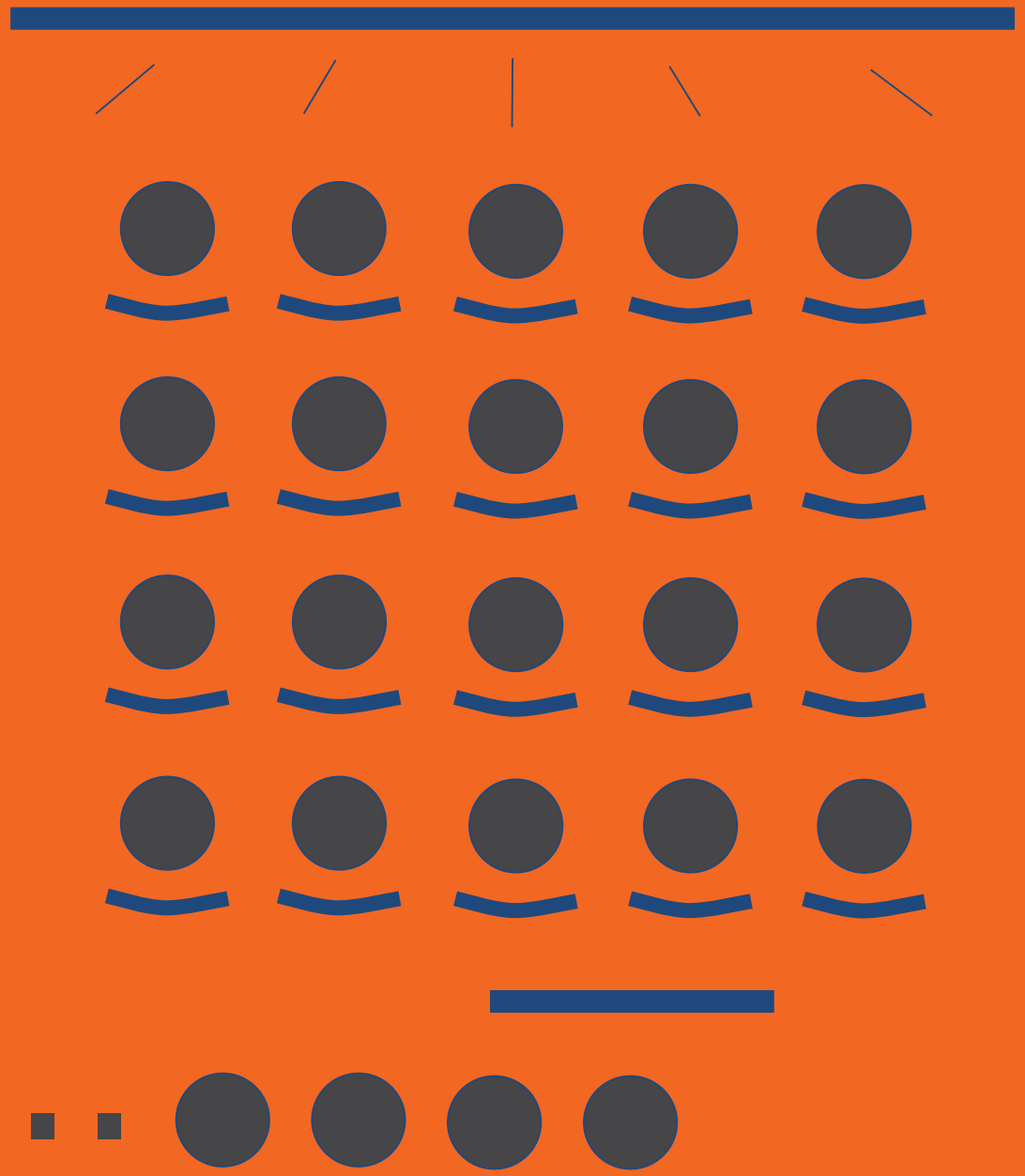
Throttling Incoming Work

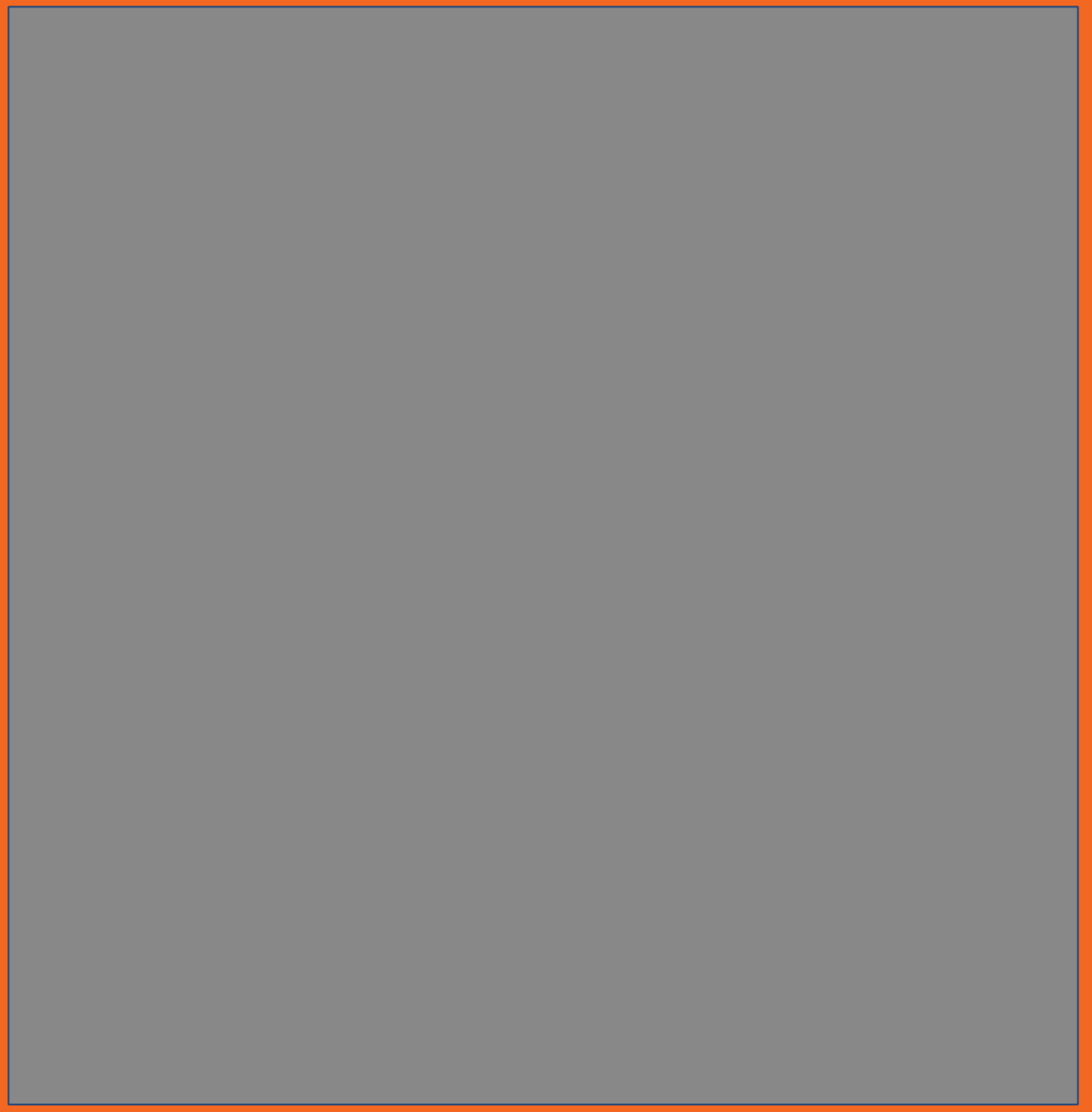
Josh Cummings

@jzheaux | tech.joshcummings.com

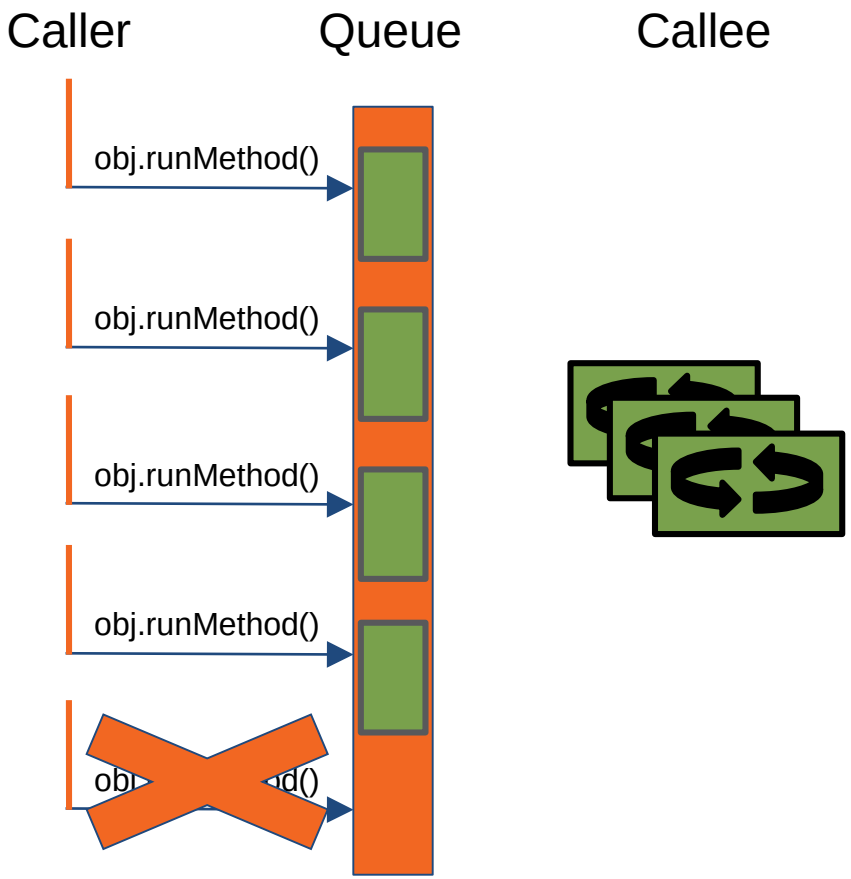
Identity Pipeline: Throttling Work

- Reads in a stream of identity-related data
- **Normalizes and verifies the contents of each identity**
- Merges and persists the identities in an in-memory cache
- Notifies an error queue if any of the above fails
- Records aggregate identity statistics





The Backpressure Pattern



- Service rejects requests after bounded queue fills up
- Client decides what to do at that point; try again or failover to a different service
- Queueing Theory provides a mathematical model (see Little's Law)

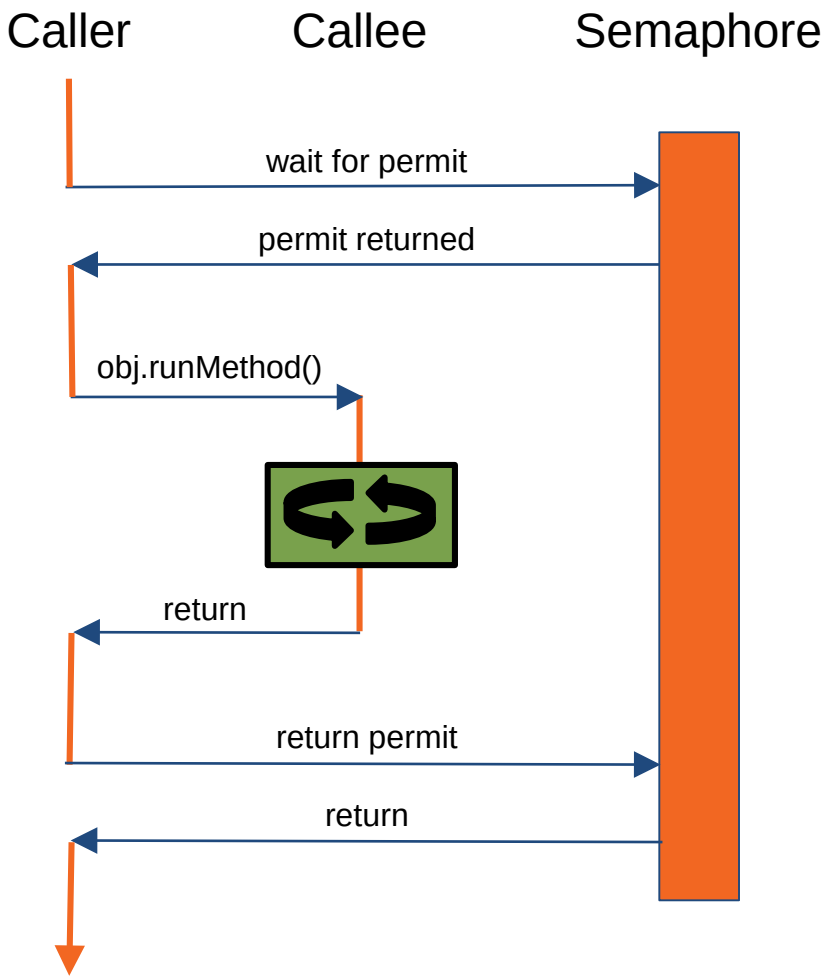
ThreadPoolExecutor, part II

- Configurable for backpressure by supplying an ArrayBlockingQueue where the capacity is the in-flight max
- Defaults to throw a RejectedExecutionException if no handler is provided
- Max and core pool size can be throttled at runtime

```
// limit the length of the pool's internal queue
// and configure a rejection handler for when
// backpressure is applied

ThreadPoolExecutor pool =
        new ThreadPoolExecutor(5, 5, -1,
                TimeUnit.MILLISECONDS,
                new
ArrayBlockingQueue<>(capacity),
                new
RejectionExecutionHandler() {
                        public void
rejectedExecution(...) {
                                // fail/try
another service
                        }
                });
```

Demo

The Semaphore Pattern



- Semaphore has as a given number of permits that can be issued on its lock
- Each client blocks at a “ticket booth”, waiting for a permit
- Only a given number of callers can invoke the method at a time

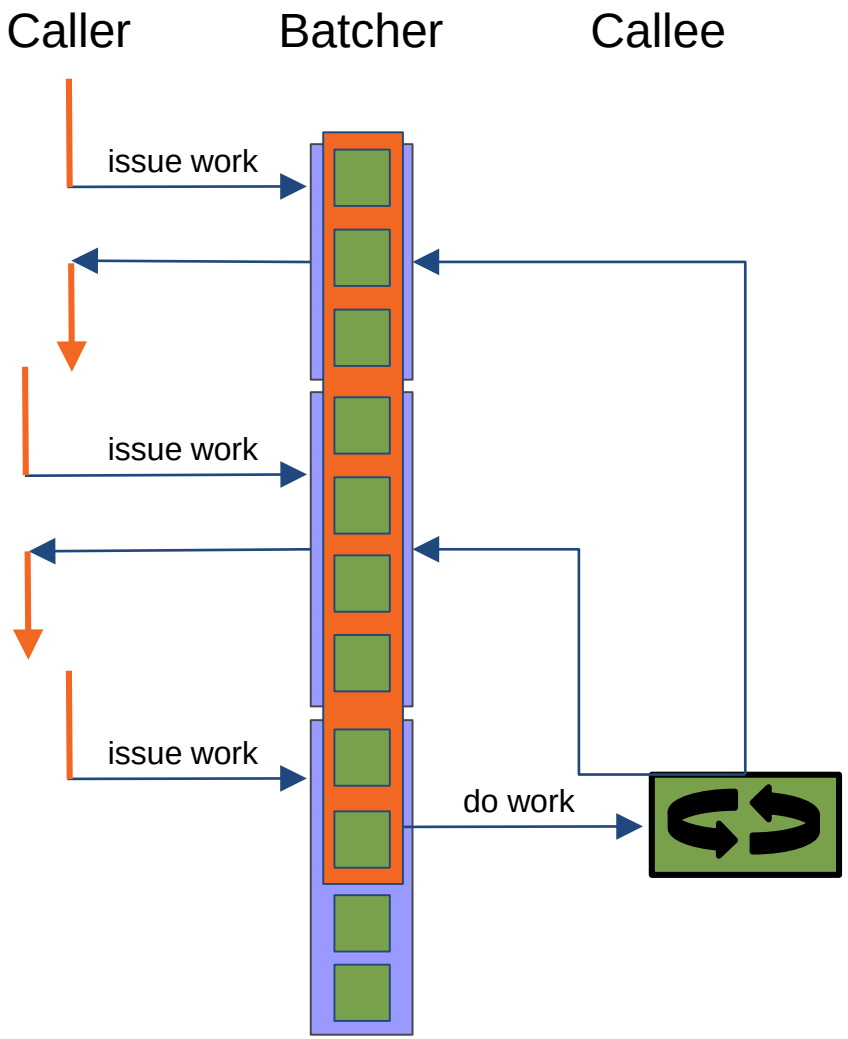
Semaphore

- Similar *happens-before* semantics to ReentrantLock, just allowing more than one thread to obtain the lock at once
- Similar acquisition and release patterns to ReentrantLock, like trial locking and timed waits
- Number of available permits can be changed at runtime if Semaphore is subclassed

```
// ask for a permit to call the method, and
then
// call it

Semaphore ticketBooth = new
Semaphore(5);


public void verifyAddresses(List addresses) {
        ticketBooth.acquire();
        try {

        delegate.verifyAddresses(addresses)
;
        } finally {
                ticketBooth.release();
        }
}
```

Demo

The Batching Pattern



- Batch according to a specific size with a timeout
- Callers can ignore, block, or provide callback to continue once their payload has been processed
- Like Backpressure, model the traffic in order to get optimal batch size and timeout

CyclicBarrier

- Like CountDownLatch, but reusable
- New arrivals should block
- A runnable can be specified to run each time the barrier is broken
- Useful when repeated groups of multiple threads must wait for one another

```
// guests explore the museum, the guide
// responds once all are done

CyclicBarrier tour = new
CyclicBarrier(numberOfGuests,this::alertGuid
e);


public void arrive(Guest guest) {
            // explore room
      tour.await(3000); // wait for others
}

public void alertGuide() {
            guide.wakeup(this); // guide to next
room
}           }
}
```

Demo

CyclicBarrier

CountDownLatch

Phaser

Phaser

- Arrivals may choose to block and wait or simply announce their arrival
- Separates the concepts of registration and arrival to allow for myriad sophisticated completion patterns
- Supports tiers of Phasers to reduce contention in cases of a large number of registrants

```
// guests arrive at each table, the waiter
// responds as each table fills up

Phaser waiter =
        new Phaser(numberOfGuests + 1);


public void arrive(Guest guest) {
        waiter.arrive(); // sit at table, etc...
}

public void arrive(Waiter waiter) {
        waiter.arriveAndAwaitAdvance();
    // now take orders...
}
```

```
        }
}
```

Demo

Review

- Applications can be made more resilient by considering arrival rate, concurrency limits, and bulk requests
- The Java Concurrency API offers several classes that can cap arrival rates in order to increase overall throughput
- Whatever solution is chosen, modeling actual traffic levels is critical