Lightweight Threads on the JVM

github.com/puniverse/quasar

Parallel Universe

Intro

- Ron Pressler
  `ron@paralleluniverse.co`
  `@puniverseco`

- Parallel Universe: OSS infrastructure for hardware-harmonious apps — storage, DB, concurrency, IO

- An Opinionated Guide to Modern Java

- Capsule — dead-simple deployment for JVM apps
  `github.com/puniverse/capsule`

The Problem with Threads

The concurrency level offered by the OS (threads) cannot match that which is required by modern servers (i.e. #conc. sessions >> #threads)

This is one of the main reasons for concurrency pain

The Problem with Threads

Benchmark

How to Not Block

- Callback (hell) – unclear flow, unclear concurrency
- Monads

```
CompletableFuture.supplyAsync(…).thenAccept(…)
   .thenAccept(…)…;
```

— Manual context, obscure concurrency, requires library support *and* API changes (methods need to return `CompletableFuture`).

Pull > Push

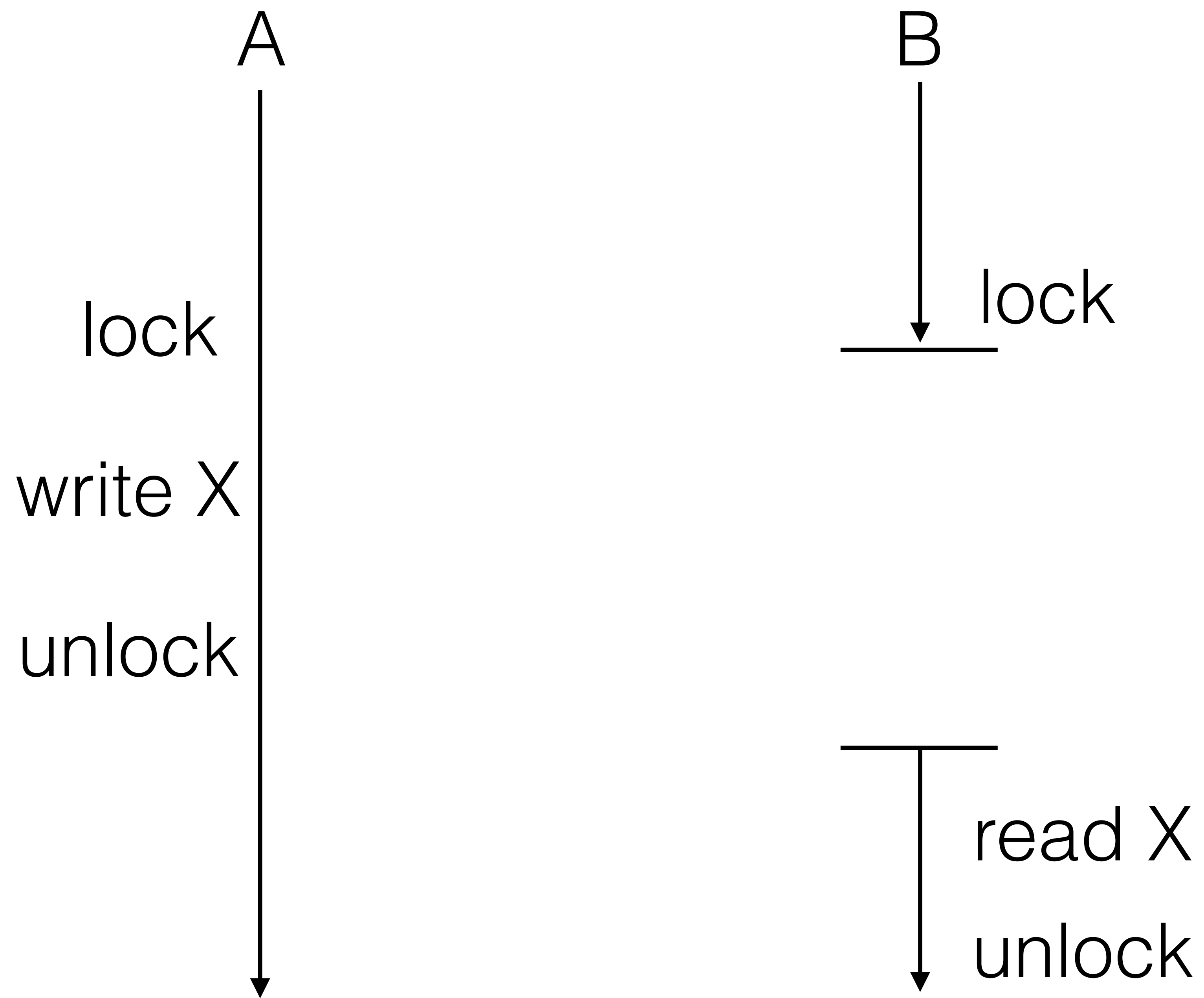
`Message m = receive()`

`onMessage(Message m)`

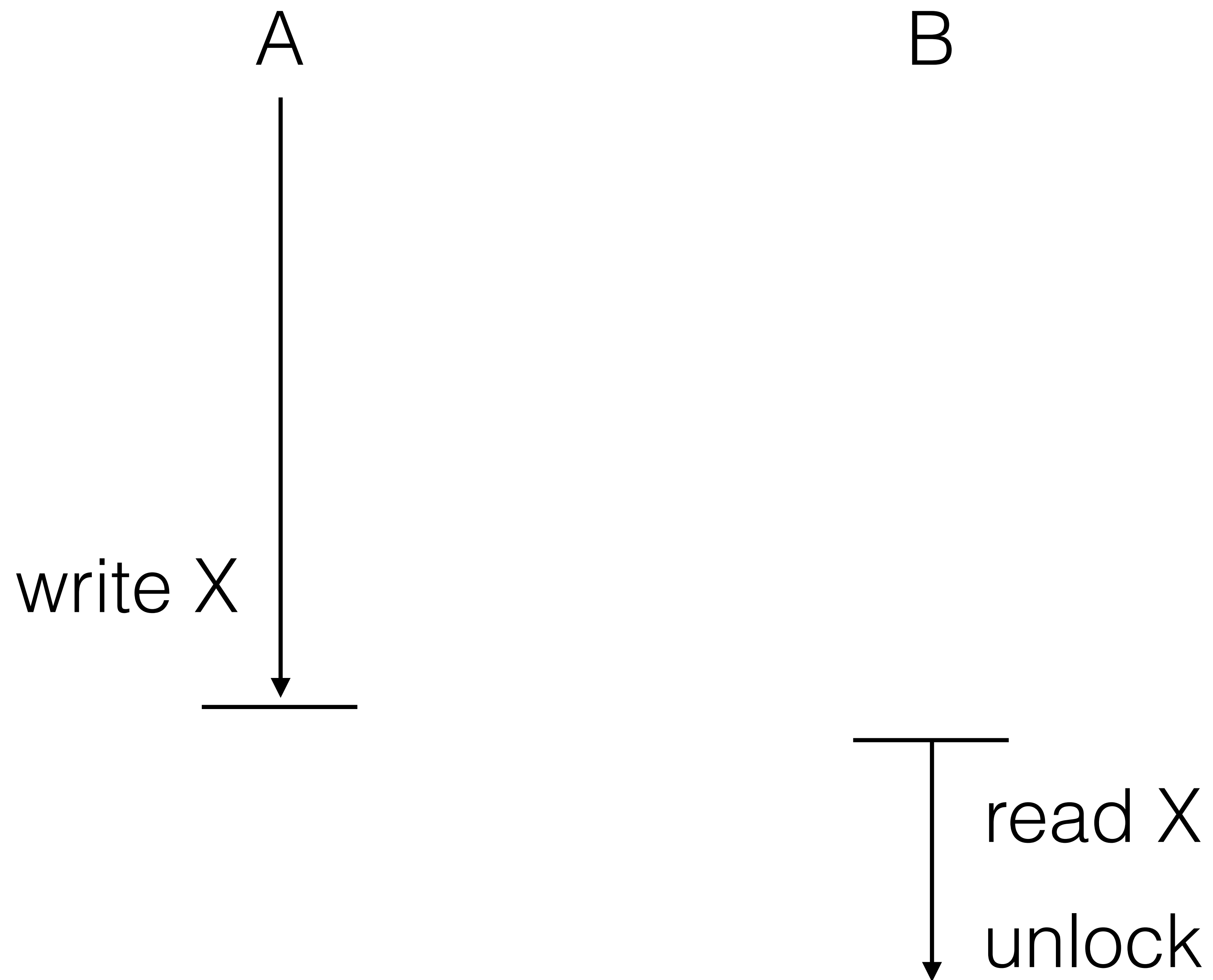
- **More general/powerful**
  - Pull -> Push is simpler than Push -> Pull (requires another queue)
  - OS already talks Pull (and already has a queue)
- **Conveys more information**
  - To programmer: threading is obvious (no need to specify in docs)
  - To the software: back pressure is implicit

Why replace a good, familiar, and natural **abstraction** simply because of an insufficient **implementation**?

General Purpose (Kernel) Threads



Transaction-Serving Threads



Definitions

- Thread – a plain Java thread, mapped 1:1 to a kernel thread
- Fiber – a user-mode, lightweight thread. M:N mapping to kernel threads
- Strand – an abstraction for a unit of concurrency, i.e. a thread or a fiber

Implementing Fibers

- Continuations (or user-scheduled kernel threads)
- Scheduler

JVM Continuations with Bytecode Instrumentation

```
class Stack {
    int[] method;           // PC, SP
    long[] dataLong;        // stack primitives
    Object[] dataObject;    // stack refs
}
```

JVM Continuations with Bytecode Instrumentation

```
bar() {
   baz();
   foo(); // sus
}
```

```
foo() {
   …
   Fiber.park();
   …
}
```

JVM Continuations with Bytecode Instrumentation

```
bar() {
    baz();
    foo(); // sus
}
```

```
foo() {
    …
    Fiber.park();
    …
}
```

JVM Continuations with Bytecode Instrumentation

```
bar() {
   int pc = isFiber ? s.pc : 0;
   switch(pc) {
   case 0:
     baz();
     if(isFiber) {
       s.pc = 1;
       // store locals -> s
     }
   case 1:
     if(isFiber)
       // load locals <- s
     foo(); // sus
  }
}
```

```
foo() {
   int pc = isFiber ? s.pc: 0;
   switch(pc) {
     …
     if(isFiber) {
       s.pc = 3;
       // store locals -> s
     }
     Fiber.park(); // thrw SE
   case 3:
     if(isFiber)
       // load locals <- s
     …
   }
}
```

Continuation Overhead

Benchmark	(DEPTH)	(STACK)	Mode	Score	Error	Units
baseline	3	16	avgt	17.505	1.383	ns/op
baseline	5	16	avgt	24.441	4.977	ns/op
baseline	10	16	avgt	40.672	3.638	ns/op
baseline	20	16	avgt	72.567	2.746	ns/op
fiber	3	16	avgt	260.138	58.866	ns/op
fiber	5	16	avgt	253.835	38.156	ns/op
fiber	10	16	avgt	311.743	99.902	ns/op
fiber	20	16	avgt	418.294	169.621	ns/op
fiberNoPark	3	16	avgt	138.126	56.220	ns/op
fiberNoPark	5	16	avgt	305.320	94.170	ns/op
fiberNoPark	10	16	avgt	301.045	216.137	ns/op
fiberNoPark	20	16	avgt	222.974	61.968	ns/op

- Negligible latency overhead (remember, we're blocking)
- But we would like to spend these cycles running app code

Continuation Issues

Suspendable Call Sites

- Interfaces/superclasses: iff have a suspendable implementation (dynamic proxies marked manually)
- Reflection, invoke dynamic: presumed suspendable (except lambdas)

Continuation Issues

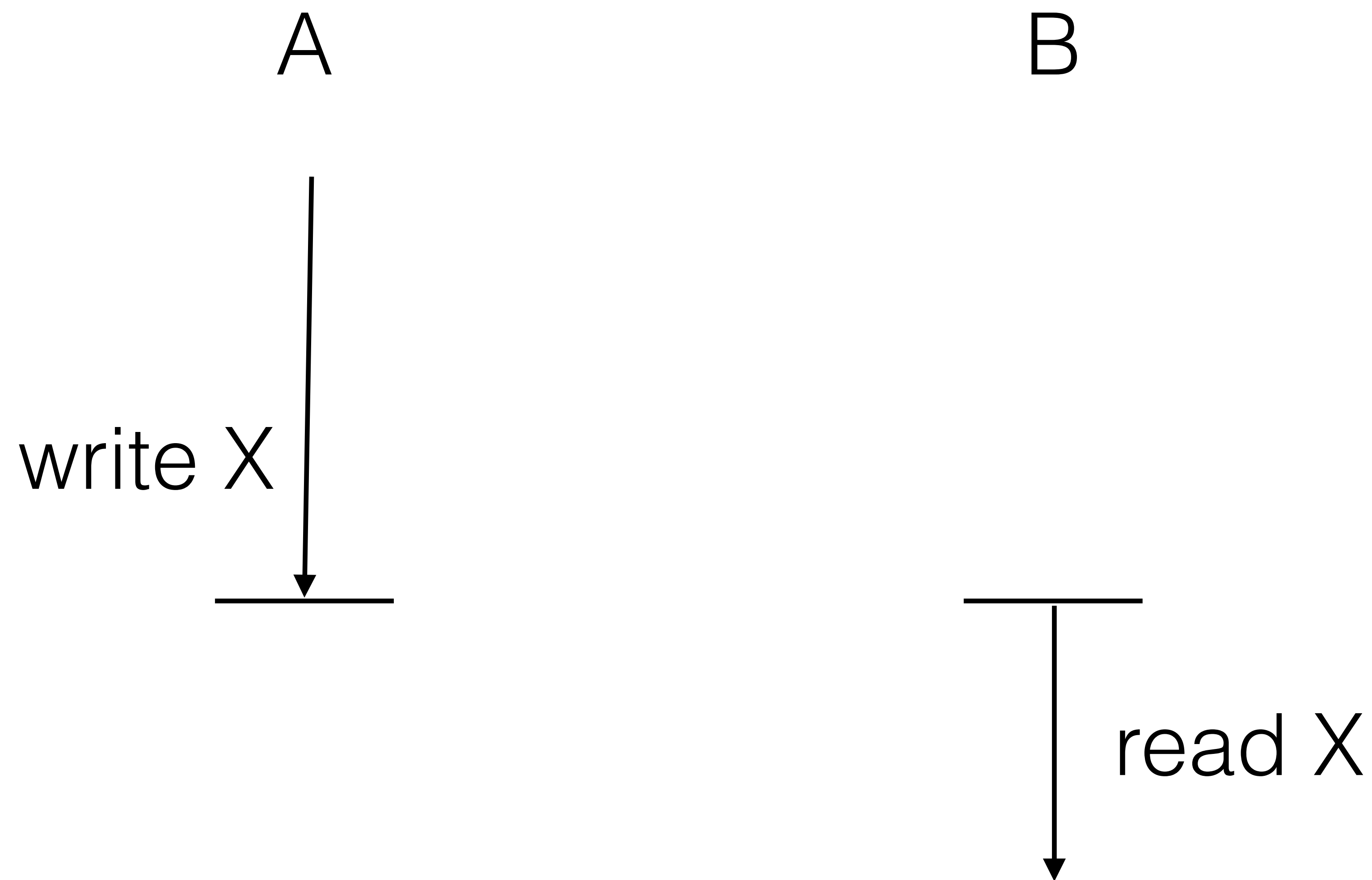
- Which methods to instrument? (manual/graph analysis)
- Other JVM languages (a function is compiled into multiple methods)
- Instrumentation "infection": if we're conservative and automatic, a lot of methods must be instrumented (e.g. consider `Runnable` being suspendable — and all it's callers…).

Can we please have native JVM continuations?

Scheduling

- Currently: stock `ForkJoinPool` in async mode by default, or any `Executor` (even on EDT)
- Work-stealing is great for "transaction processing" fibers
- Core methods: `Fiber.park()`, `Fiber.unpark()`
- State (lease) CAS circumvents FJ's "fork once" rule.

Transaction-Serving Fibers



Scheduling

- Every fiber increments a "run counter"
- Fiber state is fenced
- Occasionally, all pool's threads are examined (read state for store-load fence); same fiber encountered twice in a row with the same counter is reported as a runaway fiber.

Fiber Benchmark

Fiber Usage: CSP

```
Channel<String> ch = Channels.newChannel(0);

new Fiber(() -> {
    String m = ch.receive();
    System.out.println("Received: " + m);
}).start();

new Fiber(() -> {
    ch.send("a message");
}).start();
```

Fiber Usage: Dataflow

```
Val<Integer> a = new Val<>();
Var<Integer> x = new Var<>();
Var<Integer> y = new Var<>(() -> a.get() * x.get());
Var<Integer> z = new Var<>(() -> a.get() + x.get());
Var<Integer> r = new Var<>(() -> {
    int res = y.get() + z.get();
    System.out.println("res: " + res);
    return res;
});

Strand.sleep(2000);
a.set(3);
```

```
new Fiber<Void>(() -> {
    for (int i=0; i<200; i++) {
        x.set(i);
        Strand.sleep(100);
    }
}).start();
```

Fiber Usage

- Actors
- Fiber serialization

Fiber Issues: Integration

- Requires library support
- Must take care instrumenting all blocking methods (but not if the JVM has continuations)
- Tooling: Debuggers

But:

- Fibers and threads work great together
- No API changes
- Concurrency primitives – we've got you covered
- IO – an app uses no more than 5 IO libraries

Strands: Abstracting Threads and Fibers

```
Strand.currentStrand();
Strand.park();
Strand.parkNanos(long nanos);
Strand.unpark(Strand s);
Strand.sleep(long millis);
Strand.join();
Strand.getStackTrace();
```

```
class FIFOMutex {
   private AtomicBoolean locked = new AtomicBoolean(false);
   private Queue<Thread> waiters = new ConcurrentLinkedQueue<Thread>();

   public void lock() {
     Thread current = Thread.currentThread();
     waiters.add(current);

     // Block while not first in queue or cannot acquire lock
     while (waiters.peek() != current ||
            !locked.compareAndSet(false, true)) {
        LockSupport.park(this);
     }
   }

   public void unlock() {
     locked.set(false);
     LockSupport.unpark(waiters.peek());
   }
 }
```

```java
class FIFOMutex {
  private AtomicBoolean locked = new AtomicBoolean(false);
  private Queue<Strand> waiters = new ConcurrentLinkedQueue<Strand>();

  public void lock() {
    Strand current = Strand.currentStrand();
    waiters.add(current);

    // Block while not first in queue or cannot acquire lock
    while (waiters.peek() != current ||
           !locked.compareAndSet(false, true)) {
      Strand.park(this);
    }
  }

  public void unlock() {
    locked.set(false);
    Strand.unpark(waiters.peek()); // waiter.fjTask.fork()
  }
}
```

Callback → Fiber Blocking

```
class Foo {
   public void asyncOp(FooCompletion callback);
}

interface FooCompletion {
  void success(String result);
  void failure(FooException exception);
}
```

Callback → Fiber Blocking

```
class FooAsync extends FiberAsync<String, FooException>
     implements FooCompletion {
  @Override
  public void success(String result) {
    asyncCompleted(result); // unpark fiber
  }
  @Override
  public void failure(FooException exception) {
    asyncFailed(exception); // unpark fiber
  }
}
```

Callback → Fiber Blocking

```
String op() {
    new FooAsync() {
      @Override // called after fiber is parked
      protected void requestAsync() {
        Foo.asyncOp(this);
      }
    }.run();
}
```

Fiber Blocking NIO

```
FiberServerSocketChannel s = FiberServerSocketChannel.open().bind(new InetSocketAddress(PRT));
new Fiber((SuspendableRunnable)() -> {
    try {
        while (true) {
            final FiberSocketChannel ch = s.accept();
            new Fiber(() -> {
                try {
                    ByteBuffer buf = ByteBuffer.allocateDirect(1024);
                    int n = ch.read(buf);
                    String response = "HTTP/1.0 200 OK\r\n…";
                    n = ch.write(charset.newEncoder().encode(CharBuffer.wrap(response)));
                    ch.close();
                } catch (IOException e) { … }
          }).start();
        }
    } catch (IOException e) { … }
}).start();
```

Fibers buy us the same performance benefits as async, while keeping a simple, familiar and intuitive abstraction.

Future of Concurrency

- Lightweight Threads
- Domain-specific STM

Thank You

ron@paralleluniverse.co

@puniverseco