Loom: Continuations & Fibers

Ron Pressler

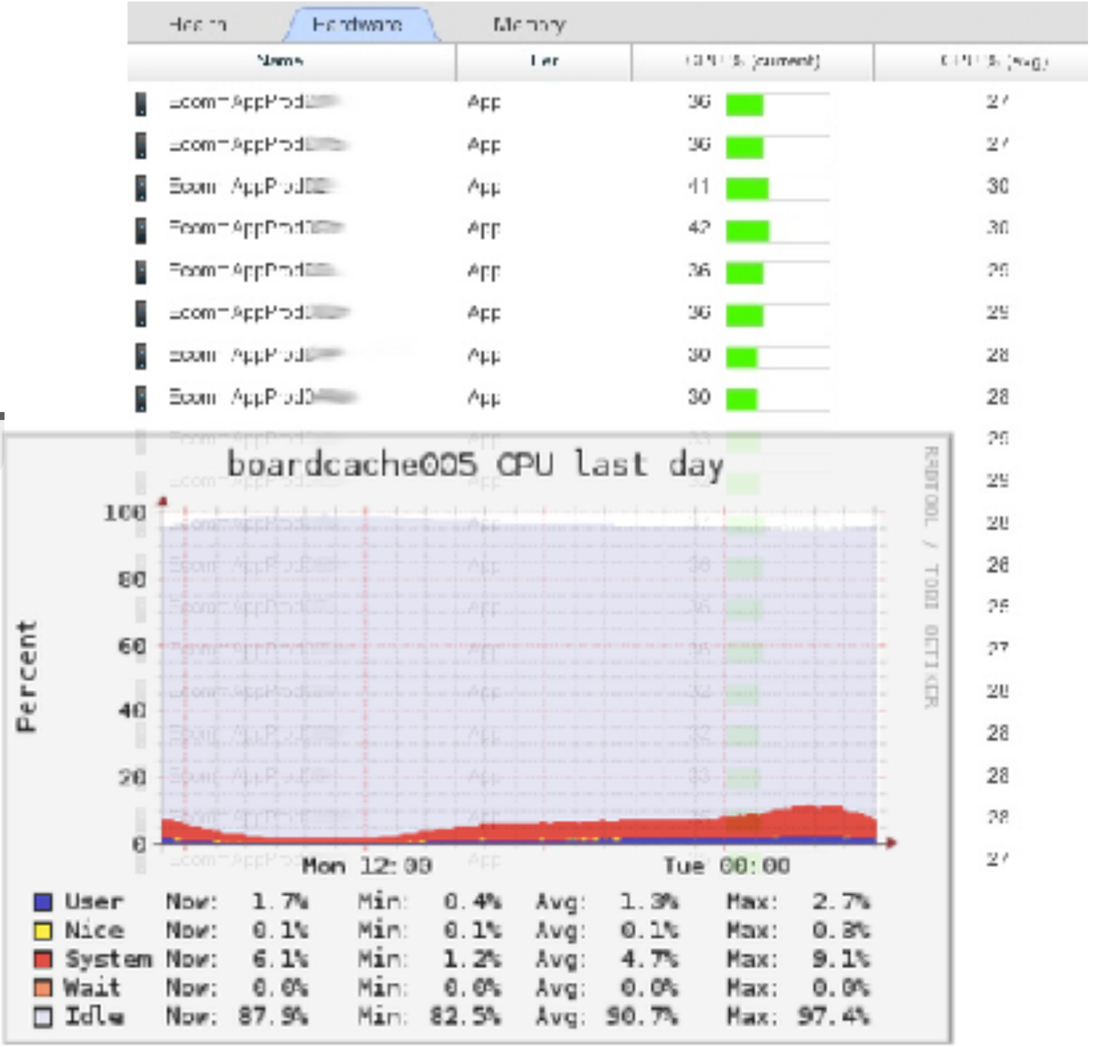
January 2018

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract.
It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

A fiber is a thread scheduled not by the OS but by the Java runtime/user code, i.e., a *user mode* thread. A fiber works like a thread, but you can have millions of them rather than thousands because of low footprint and negligible task-switching overhead.

Why Now

Sessions are getting longer (realtime push etc.) => servers experience more of them, but spend most of their time waiting for IO from DB / other services: 5-30% CPU utilization



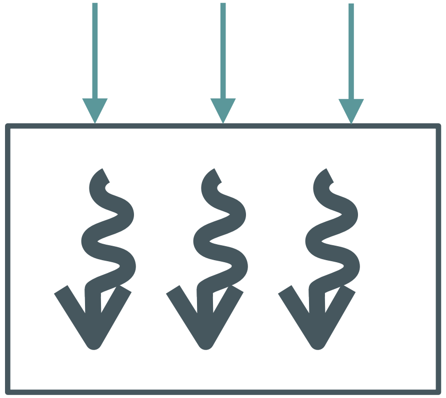
Servers are underutilized

Why Fibers

Today, developers are forced to choose between

Connections

App

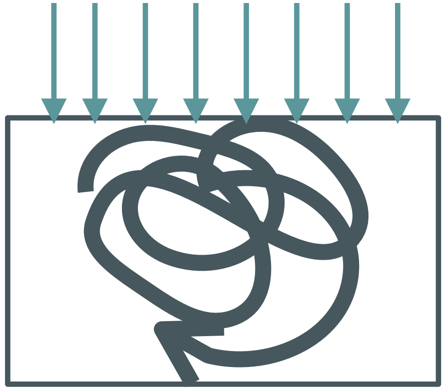


simple (blocking / synchronous),
but less scalable code (with threads)

and

Connections

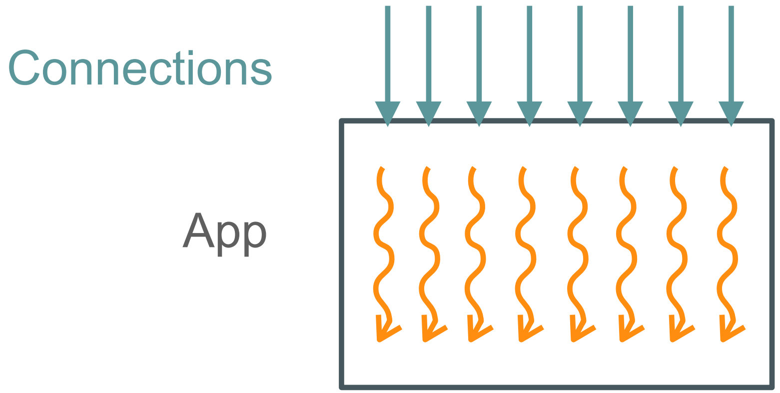
App



complex, non-legacy-interoperable,
but scalable code (asynchronous)

Why Fibers

With fibers, devs have *both*: simple, familiar, maintainable, interoperable code, that is also scalable



Fibers make even existing server applications consume fewer machines (by increasing utilization), significantly reducing costs

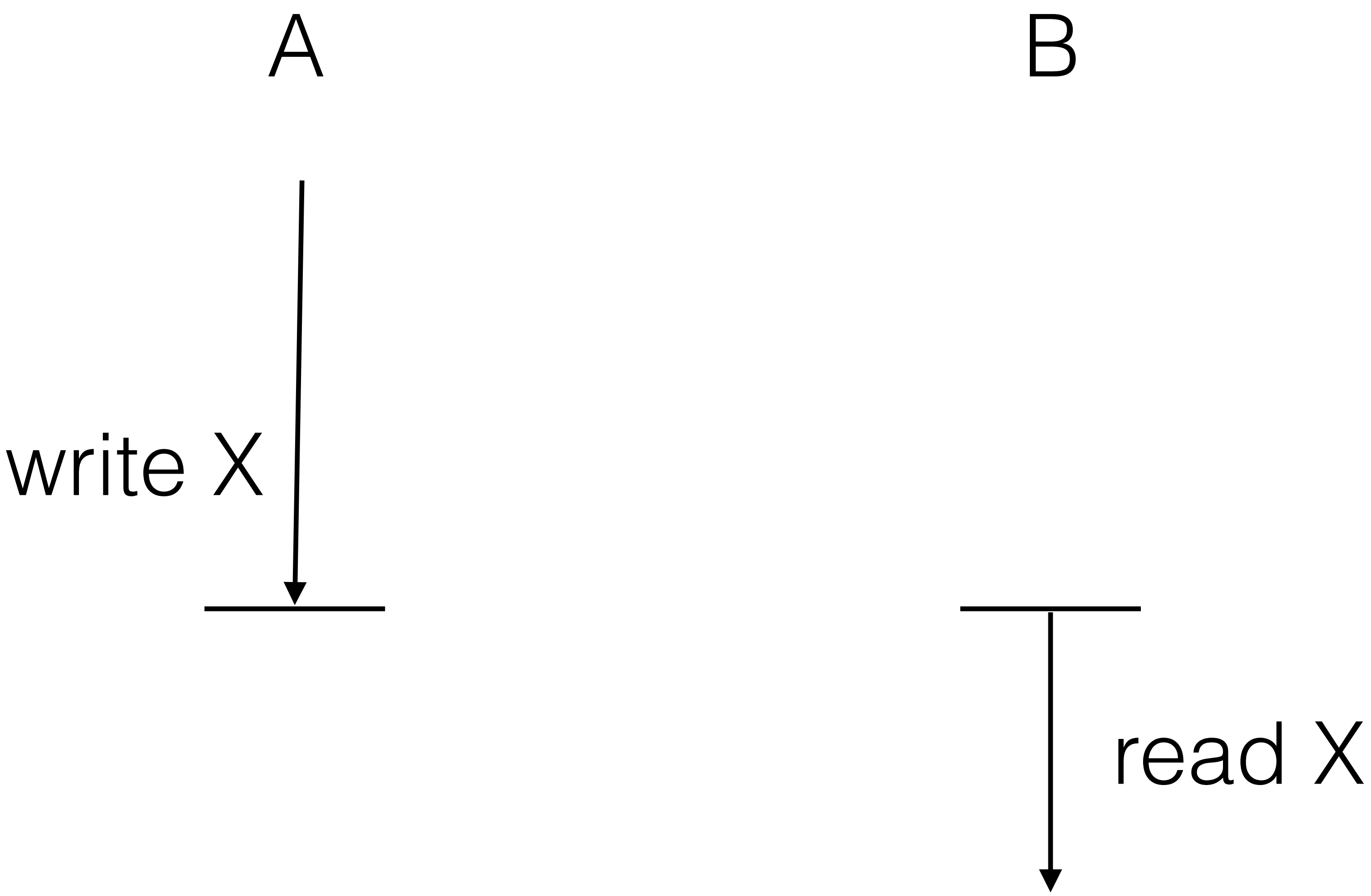
The main motivation of adding continuations to the JDK is **lightweight concurrency** (i.e. to implement fibers):

- Higher throughput/lower cost for ordinary Java code (incl. legacy)
- Devs shouldn't choose between performance and maintainability
- Enables new, modern programming styles

Why Fibers

Key idea: A language runtime is better positioned to manage and schedule application threads than the OS, esp. if they interleave IO and computation and interact often — exactly how server threads behave (also, UI elements)

Why Fibers



API: Option I

```
Thread(Runnable, ThreadScheduler)
```

API: Option II

`Strand` (abstract thread)

`Fiber` `Thread`

Fiber-based IO (just like regular blocking IO, only scalable)

```
ServerSocketChannel s = ServerSocketChannel.open().bind(new InetSocketAddress(8080));
new Fiber(() -> {
    try {
        while (true) {
            final SocketChannel ch = s.accept();
            new Fiber(() -> {
                try {
                    var buf = ByteBuffer.allocateDirect(1024);
                    var n = ch.read(buf);
                    String response = "HTTP/1.0 200 OK\r\n…";
                    n = ch.write(encoder.encode(CharBuffer.wrap(response)));
                    ch.close();
                } catch (IOException e) { … }
            }).start();
        }
    } catch (IOException e) { … }
}).start();
```

Fiber-based IO

- Servlet
- JAX-RS

UNCHANGED!*

(* virtually)

IO: Async → Fiber-Blocking

```
class AsyncFoo {
    public void asyncFoo(FooCompletion callback);
}

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

IO: Async → Fiber-Blocking

```
abstract class AsyncFooToBlocking extends _AsyncToBlocking<String, FooException>
     implements FooCompletion {

  @Override
  public void success(String result) { _complete(result); }

  @Override
  public void failure(FooException exception) { _fail(exception); }
}
```

IO: Async → Fiber-Blocking

```
class SyncFoo {
    AsyncFoo foo = get instance;

    String syncFoo() throws FooException {
        new AsyncFooToBlocking() {
            @Override protected void register() { foo.asyncFoo(this); }
        }.run();
    }
}
```

Fibers open the door to a wealth of interesting new programming techniques that could be implemented in libraries (with no explicit support in the platform). Examples include:

- Channels
- Dataflow
- Actors
- Synchronous programming

Channels

(a-la Go, Clojure's core.async)

```
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();
```

Dataflow/Reactive

```
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();
```

Functional Reactive

```
ReceivePort<Integer> cout = Channels.transform(cin)
    .filter(x -> x % 2 != 0)
    .flatmap(x -> Channels.toReceivePort(Arrays.asList(x, x * 10, x*100)));
```

Fiber Serialization

Serialization of fibers opens a world of further possibilities:

- Tear down VMs while waiting for an event
- Code/data colocation for big data / novel databases
- Long-running (weeks, months) financial transactions

Part II

Continuations

What

A continuation (precisely: delimited continuation) is a program object representing a computation that may be suspended and resumed (also, possibly, cloned or even serialized).

```
class Foo implements Runnable {
    public void run() {
        … a(); …
    }

    void a() {
        System.out.println("111");
        b();
        System.out.println("222");
    }

    void b() {
        … Continuation.yield(_A); …
    }
}
```

```
class Bar {
  void f() {
      Continuation c = new Continuation(_A, new Foo());
      c.run(); 111
      c.run(); 222
  }
}
```

Cool Stuff You Can Do With Continuations: Generators

```
for (String x : new Generator(() -> {
      produce("a");
      Thread.sleep(100);
      produce("b");
      String c = Console.readline();
      produce(c);
  })) {
     System.out.println(x);
  }
```

Cool Stuff You Can Do With Continuations: Retry-able Exceptions

```
new Retry(() -> {
       findFile();
       writeToFile(); }) {
    @Override protected void handle(Exception e) throws Exception {
        if (e instanceof FileNotFoundException) {
            createFile();
            retry();
        } else throw e;
    }}).run();
```

Cool Stuff You Can Do With Continuations: Ambiguity

```
Ambiguity<Integer> amb = solve(() -> {
        int a = amb(1, 2, 3); // a is either 1, 2, or 3
        int b = amb(2, 3, 4); // b is either 2, 3, or 4

        assertThat(b < a);    // ... but we know that b < a
        return b;
    });

amb.run(); // returns 2
```

Cool Stuff You Can Do With Continuations: Ambiguity

```
Ambiguity<Integer> amb = solve(() -> {
        Iterable<Integer> a = new Generator<>(() -> {
            produce(amb(2, 1));
            produce(amb(3, 10)); });

        int sum = 0;
        for (int x : a) { sum += x;
                          assertThat(x % 2 == 0); }
        return sum;
    });

amb.run(); // => 12
```

Thread

=

Continuation + Scheduler

(we already have a great scheduler: **`ForkJoinPool`**)

```java
class Fiber {
  private final Continuation c;
  private final Executor scheduler;
  public Fiber(Executor scheduler, Runnable target) {
    this.c = new Continuation(_FiberScope, target);
    this.scheduler = scheduler;
  }

  public static void park() ≅ { Continuation.yield(_FiberScope); }
  public void unpark() ≅ { scheduler.execute(c); }
}
```

Forced (time-slice) Preemption

- May be unnecessary
- If we add it — use safepoints

Implications

The following components would need to be continuation/fiber aware:

- Debuggers and profilers (JFR, JVMTI)
- JDK constructs
  - `synchronized`, `Object.wait()`
  - IO (NIO, old-IO?)
  - `java.util.concurrent`

Part III

Hotspot Implementation

We need:

- Millions of continuations (=> low RAM overhead)
- Fast task-switching (=> no stack copying)

Stacks

1. Contiguous virtual memory
   - Pros: Native methods (non-goal), smallest change
   - Cons: RAM overhead, address-space overhead (32 bit)

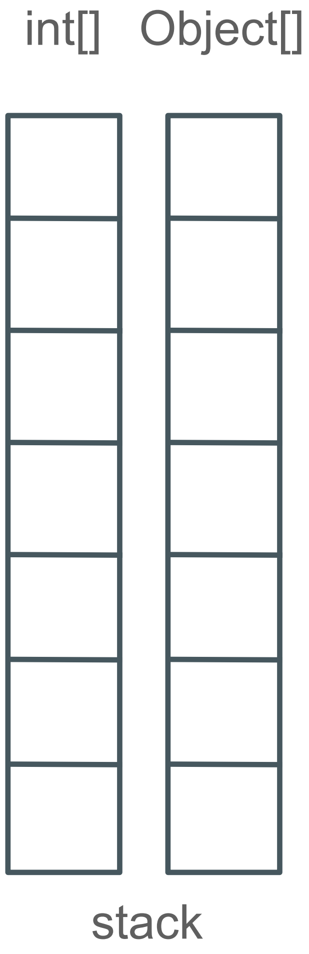
2. Stacklets (constant-size, linked, C heap)
   - Pros: Native methods (non-goal), non-relocating & treated as ordinary stacks by GC
   - Cons: Management, RAM overhead

3. Horizontal stacks (contiguous, Java heap)
   - Pros: least RAM, rely on GC for management
   - Cons: More VM (interpreter, compiler) tricks (stacks relocate), new oopMaps

4. Native Horizontal stacks (contiguous, C-heap)
   - Pros: least RAM, less GC pressure
   - Cons: Management

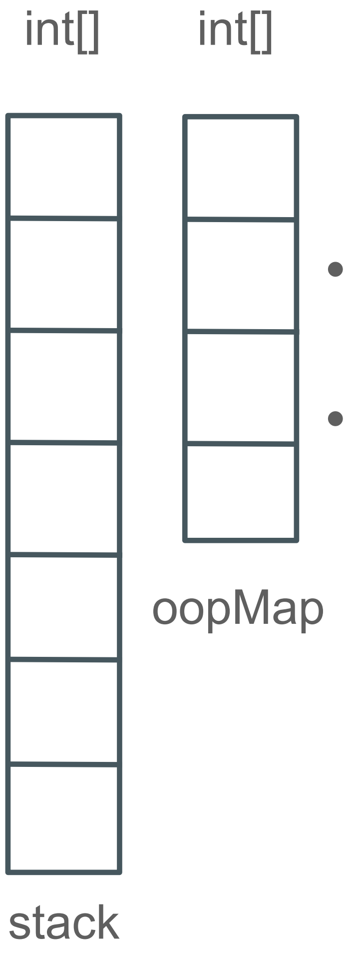
Horizontal Stacks

Split



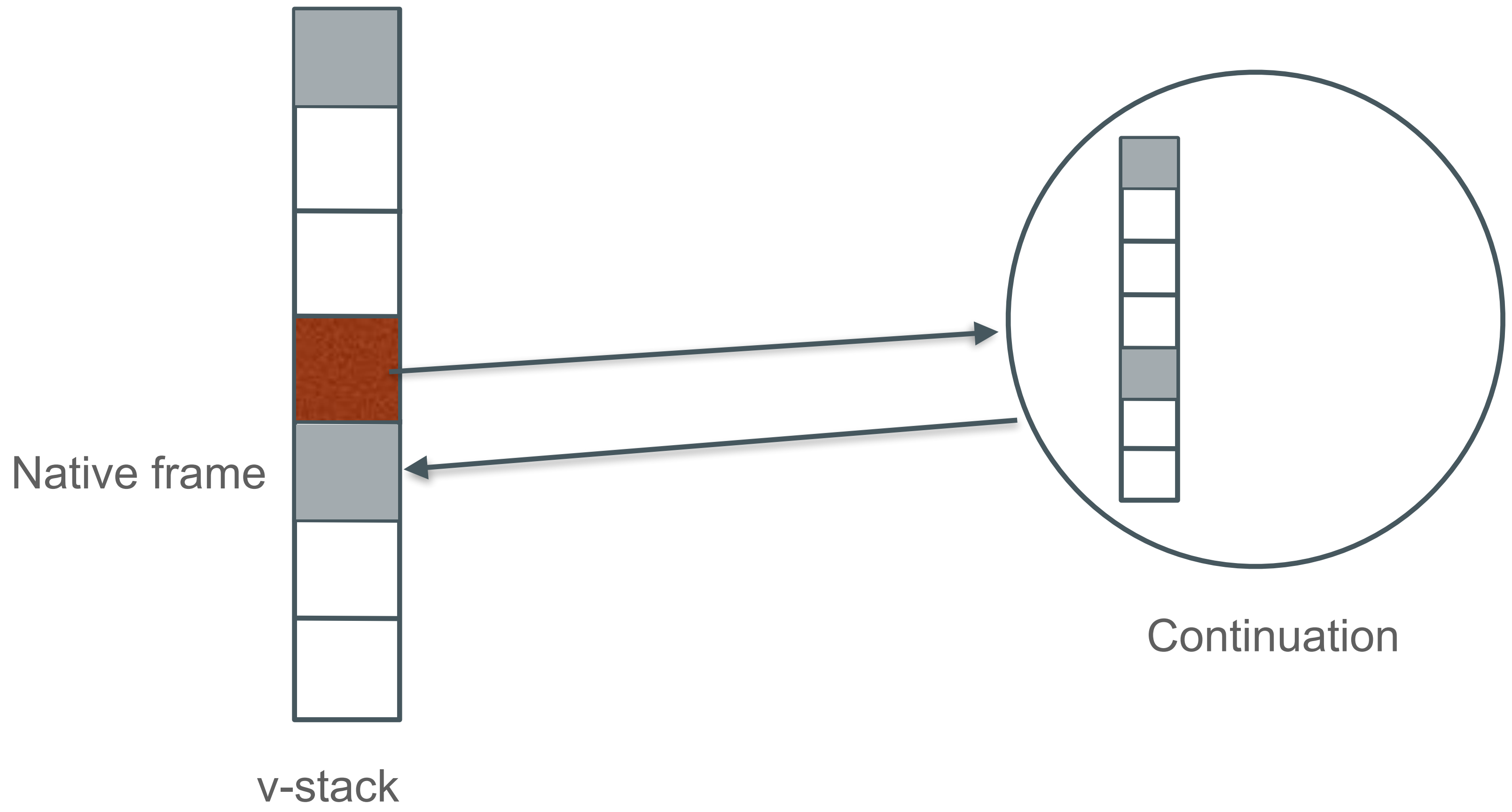
- Pros: GC just works
- Cons: More specialized/less efficient code, more waste

Simple



- Pros: More efficient/less specialized code, less waste
- Cons: How to do GC?

Native Call



GC

Invariant: continuation stacks mutate only when mounted

Java™

ORACLE®