Project Loom

Ron Pressler, Alan Bateman

June 2018

Safe Harbor Statement

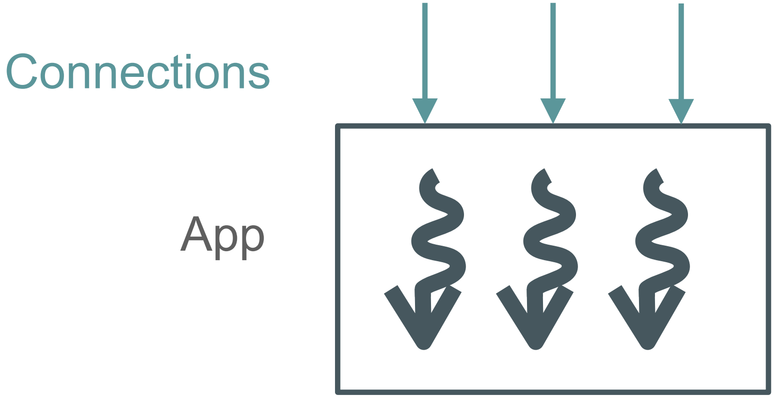
The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development,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

Project Loom

- Continuations
- Fibers
- Tail-calls

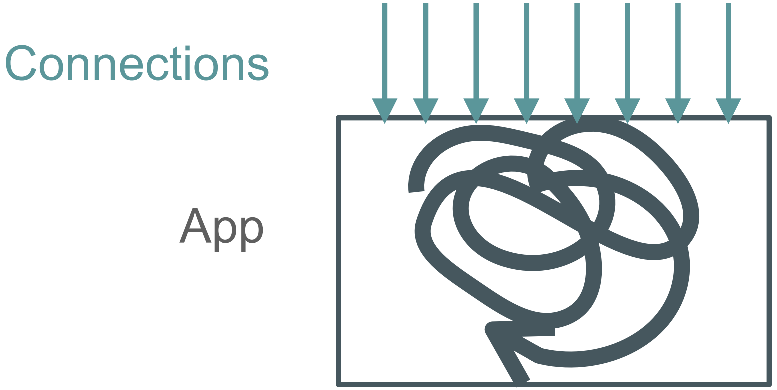
Why Fibers

Today, developers are forced to choose between



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

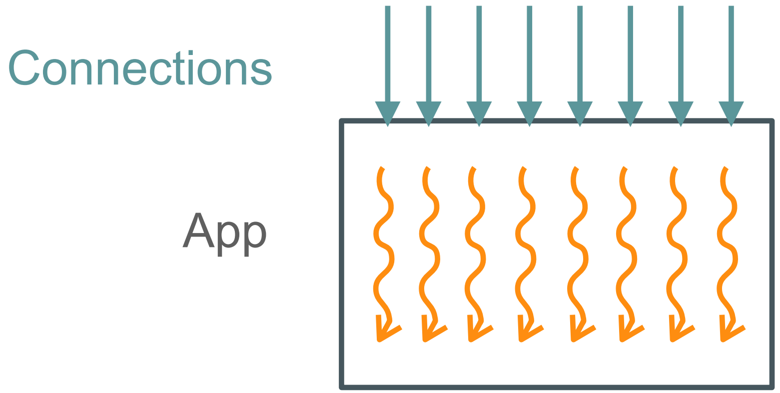
and



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

Continuations: The User Perspective

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).

Continuations: User Perspective

```
package java.lang;

public class Continuation implements Runnable {
  public Continuation(ContinuationScope scope, Runnable body);

  public final void run();
  public static void yield(ContinuationScope scope);
  public boolean isDone();

  protected void onPinned(Reason reason)
    { throw new IllegalStateException("Pinned: " + reason); }
}
```

Continuations: User Perspective

```
Continuation cont = new Continuation(SCOPE, () -> {
    while (true) {
        System.out.println("before");
        Continuation.yield(SCOPE);
        System.out.println("after");
    }
});

while (!cont.isDone()) {
    cont.run();
}
```

Fibers

What is a fiber?

- A *light weight* or *user mode thread*, scheduled by the Java virtual machine, not the operating system

- Fibers are low footprint and have negilgible task-switching overhead. You can have millions of them!

Why fibers?

- The runtime is well positioned to manage and schedule application threads, esp. if they interleave computation and I/O and interact very often (exactly how server threads behave)

- Make concurrency simple again

fiber = continuation + scheduler

fiber = continuation + scheduler

- A fiber wraps a task in a continuation
  - The continuation yields when the task needs to block
  - The continuation is continued when the task is ready to continue
- Scheduler executes tasks on a pool of *carrier* threads
  - java.util.concurrent.Executor in the current prototype
  - Default/built-in scheduler is a ForkJoinPool

User facing API

- Current focus is on the control flow and concepts, not the API
- Minimal `java.lang.Fiber` in current prototype that supports
  1. Starting a fiber to execute a task
  2. Parking/unparking
  3. Waiting for a fiber to terminate

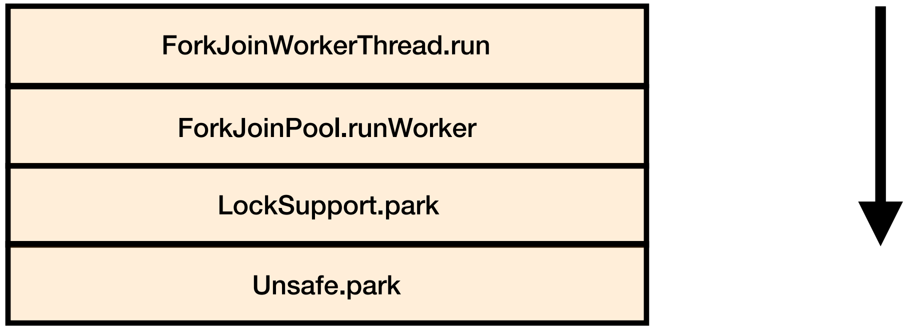
Implementing Fibers

- A fiber wraps a user’s task in a continuation
- The fiber task is submited to the scheduler to start or continue the continuation, essentially:

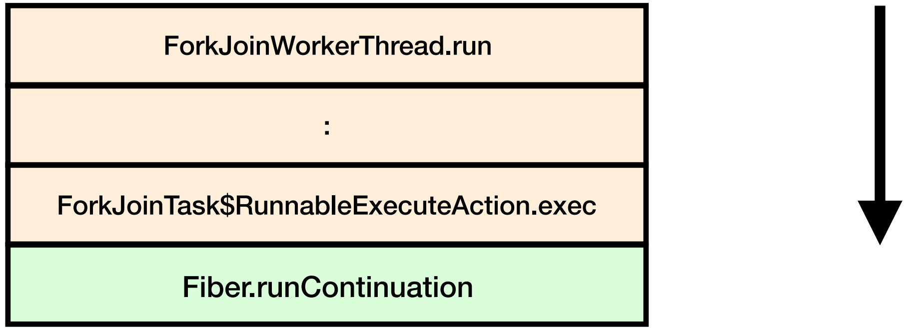
```
mount();
try {
    cont.run();
} finally {
    unmount();
}
```

```
Fiber f = Fiber.execute(() -> {
    out.println("Good morning");
    readLock.lock();
    try {
        out.println("Good afternoon");
    } finally {
        readLock.unlock();
    }
    out.println("Good night");
});
```

Carrier thread waiting for work



A fiber is scheduled on the carrier thread. The fiber task runs.



The fiber runs the continuation to run the user’s task.

```
Fiber.execute(() -> {
    out.println("Good morning");
    readLock.lock();
    try {
        out.println("Good afternoon");
    } finally {
        readLock.unlock();
    }
    out.println("Good night");
});
```

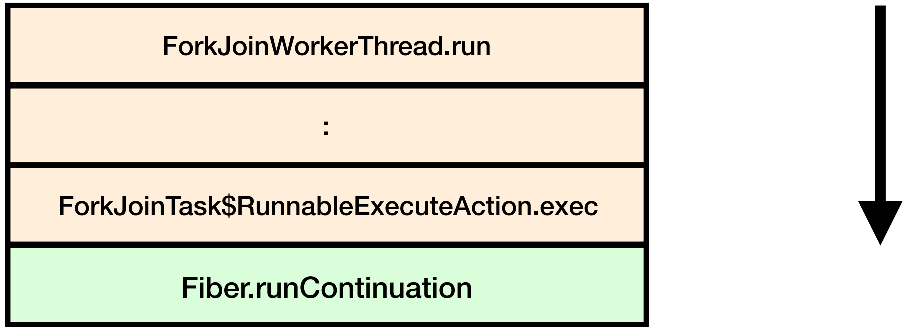
ForkJoinWorkerThread.run
:
ForkJoinTask$RunnableExecuteAction.exec
Fiber.runContinuation
Continuation.run
Continuation.enter
Continuation.enter0
Main.lambda$main$0

The task attempts acquire a lock which leads to the continuation yielding

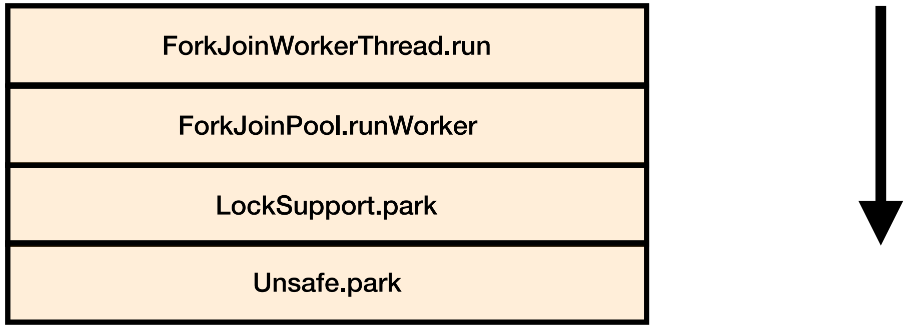
```
Fiber.execute(() -> {
    out.println("Good morning");
    readLock.lock();
    try {
        out.println("Good afternoon");
    } finally {
        readLock.unlock();
    }
    out.println("Good night");
});
```

ForkJoinWorkerThread.run
:
ForkJoinTask$RunnableExecuteAction.exec
Fiber.runContinuation
Continuation.run
Continuation.enter
Continuation.enter0
Main.lambda$main$0
java.util.concurrent.ReentrantLock.lock
:
java.util.concurrent.LockSupport.park
Fiber.park
Contuation.yield

The continuation stack is saved and control returns to the fiber's task at the instruction following the call to Continuation.run



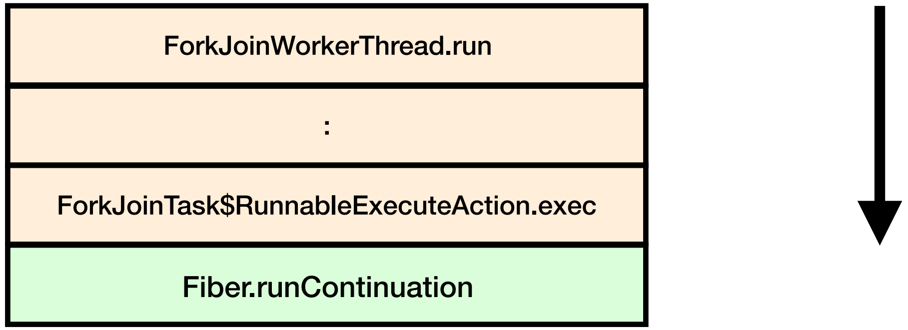
The fiber task terminates. The carrier thread goes back to waiting for work.



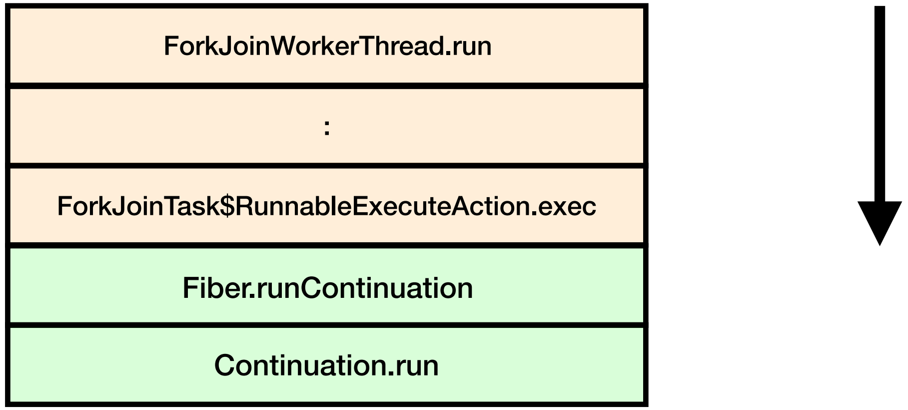
The owner of the lock releases it. This unparks the Fiber waiting to acquire the lock by scheduling its task to run again.

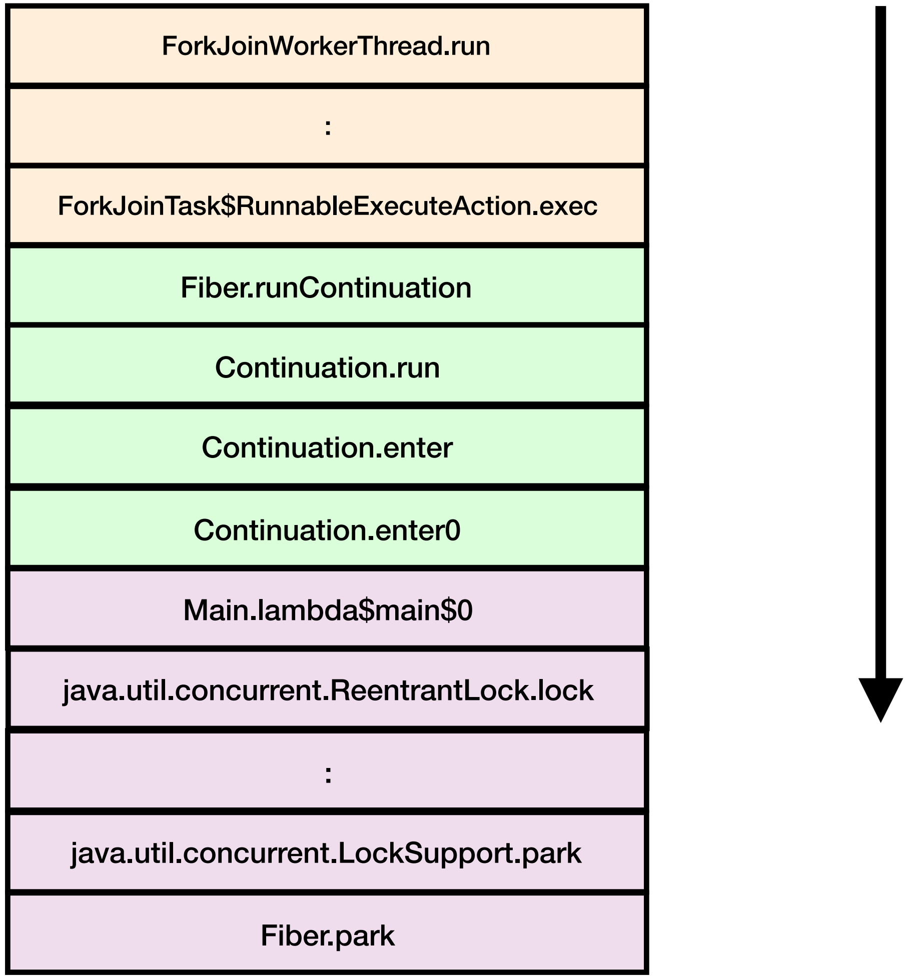
```
ReentrantLock.unlock
  LockSupport.unpark
    Fiber.unpark
      ForkJoinPool.execute
```

The fiber task runs again, maybe on a different carrier thread

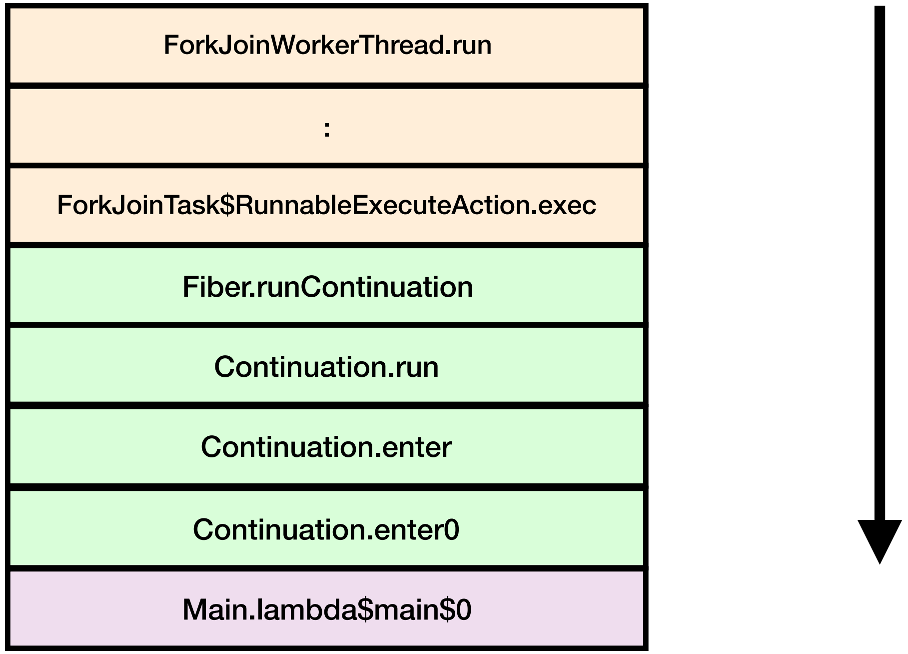


The fiber task invokes Continuation run (again) to continue it





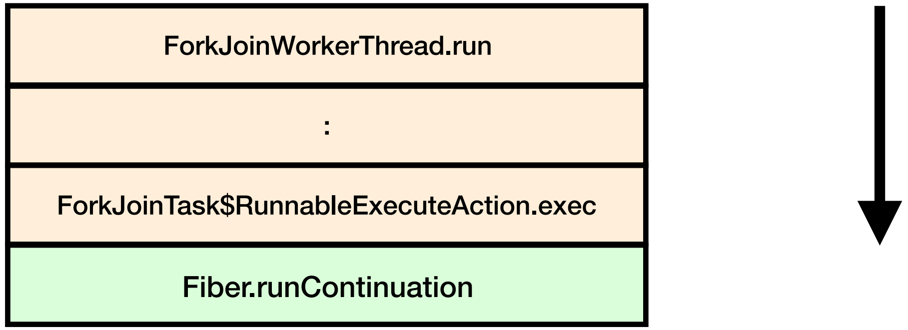
The stack is restored and control continues at the instruction following the call to Continuation.yield



The user’s task continues.

```
Fiber.execute(() -> {
    out.println("Good morning");
    readLock.lock();
    try {
        out.println("Good afternoon");
    } finally {
        readLock.unlock();
    }
    out.println("Good night");
});
```

```java
Fiber.execute(() -> {
    out.println("Good morning");
    readLock.lock();
    try {
        out.println("Good afternoon");
    } finally {
        readLock.unlock();
    }
    out.println("Good night");
});
```



The user's task completes and the continuation terminates. Control returns to the fiber's task at the instruction following the call to Continuation.run

How much existing code can fibers run?

- A big question, lots of trade-offs
  - Do we completely re-imagine threads?
  - Do we attempt to allow all existing code to run in the context of a fiber?
  - Likely to wrestle with this topic for a long time
- Current prototype can run existing code

  … but with some limitations, as we will see

Example using existing code/libraries

- Example uses Jetty and Jersey

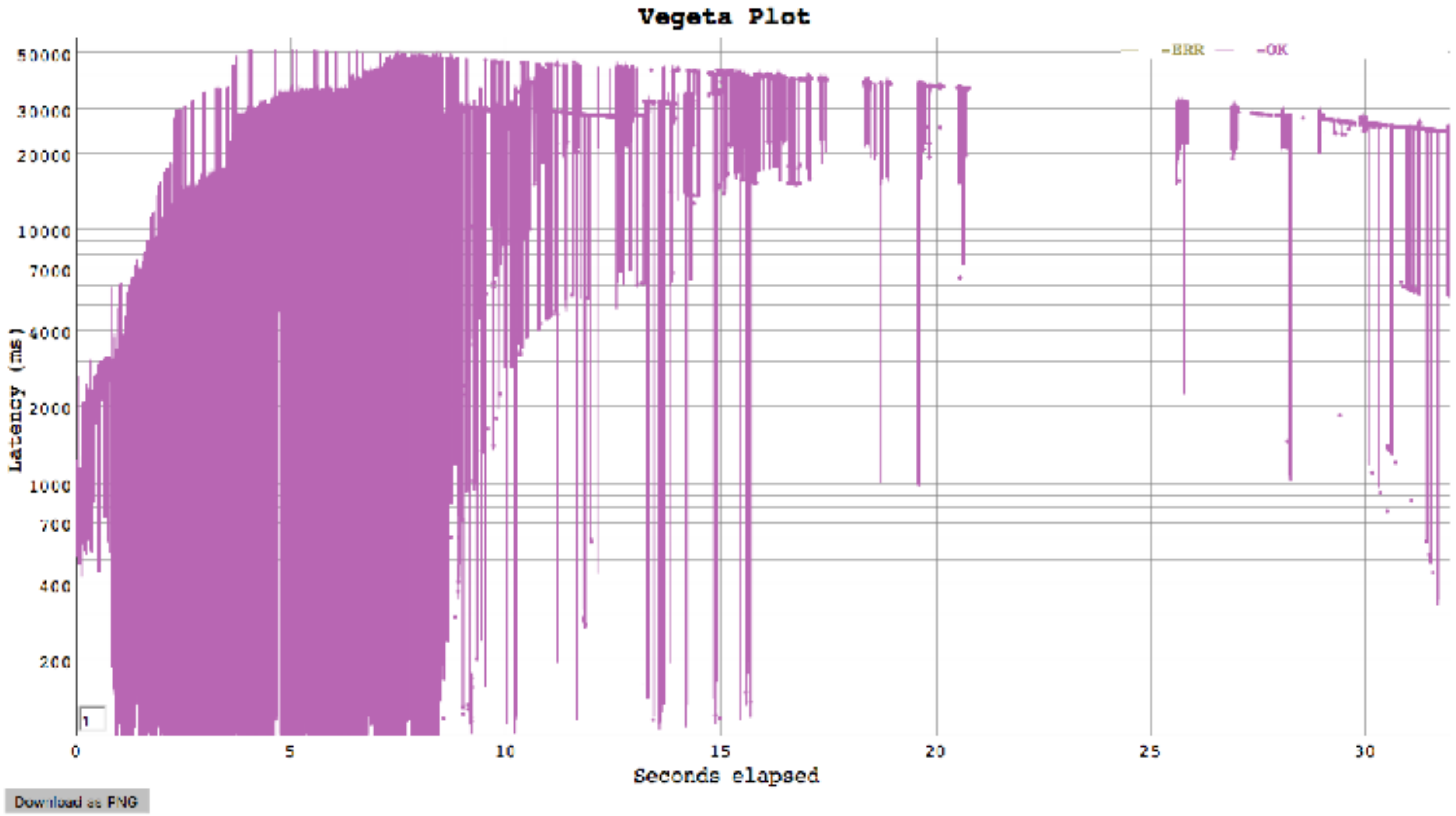
Example with existing code/libraries

- Assume servlet or REST service that spends a long time waiting

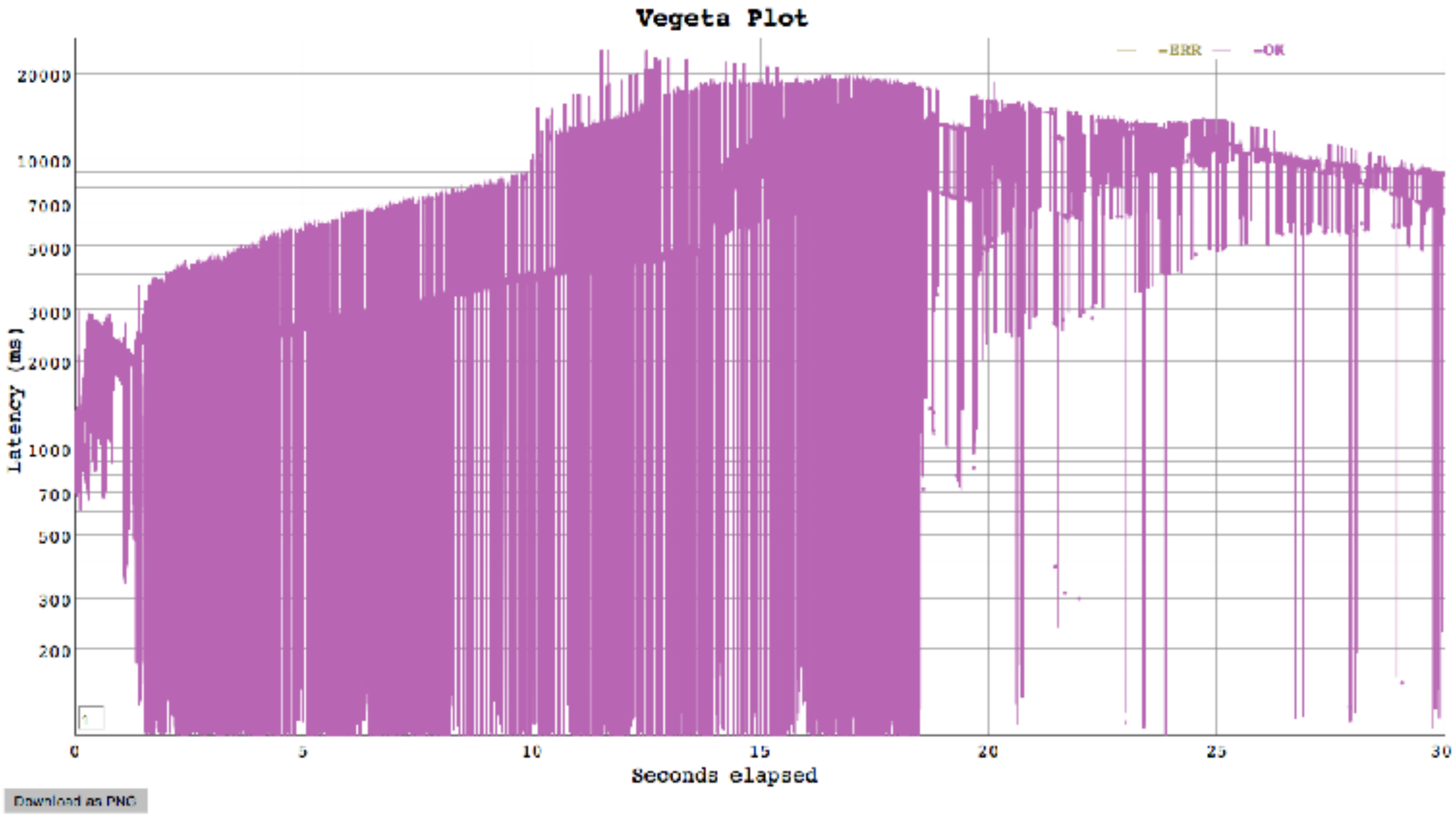
```
@GET
@Path("greeting")
@Produces(MediaType.APPLICATION_JSON)
public String greeting() {
    return "{ \"message\": \"" + computeValue() + "\" }";
}
```

assume this takes 100ms

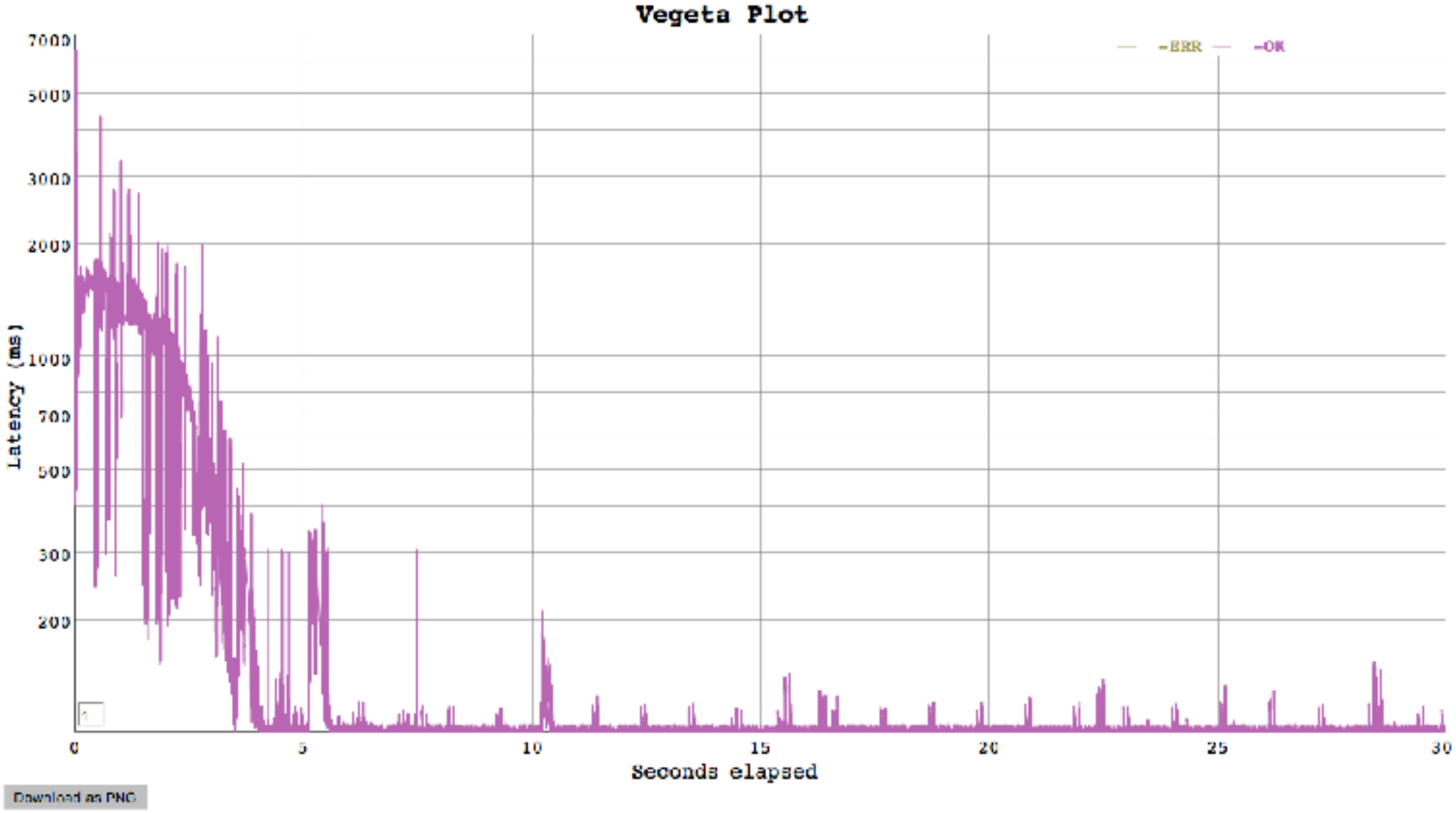
Default configuration (maxThreads = 200), load = 5000 HTTP request/s



maxThreads = 400, load = 5000 HTTP request/s



fiber per request, load = 5000 HTTP request/s



Limitations

- Can’t yield with native frames on continuation stack

```java
PrivilegedAction<Void> pa = () -> {
    readLock.lock();
    try {
        //
    } finally {
        readLock.unlock();
    }
    return null;
};
AccessController.doPrivileged(pa);
```

may park/yield

native method

Limitations

- Can't yield while holding or waiting for a monitor

```
synchronized (obj) {
    obj.wait();
}
```

may park carrier thread

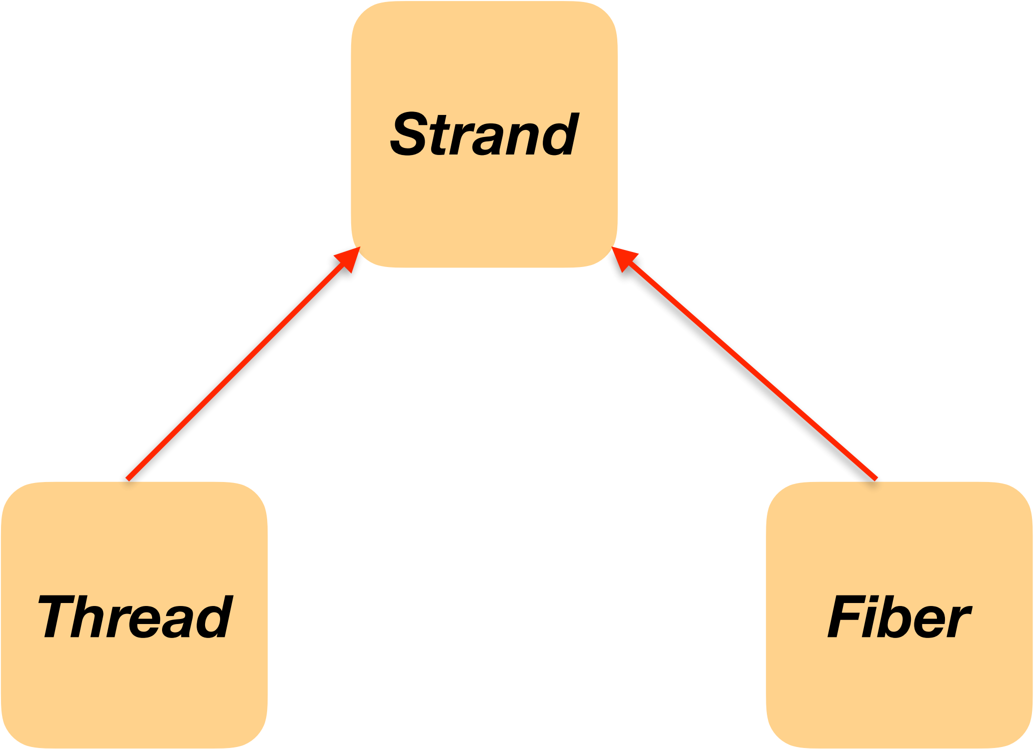
```
synchronized (obj) {
    socket.getInputStream().read();
}
```

may park carrier thread

Limitations

- Current limitations
  - Can’t yield with native frames on continuation stack
  - Can’t yield while holding or waiting for a monitor
  - In both cases, parking may pin the carrier thread
- What about the existing Thread API and Thread.currentThread() ?

Relationship between Fiber and Thread in current prototype



Thread.currentThread() and Thread API in current prototype

- Current prototype
    - First use of `Thread.currentThread()` in a fiber creates a *shadow Thread*
    - “unstarted” Thread from perspective of VM, no VM meta data
    - Shadow Thread implements Thread API except for *stop*, *suspend*, *resume*, and uncaught exception handlers
- Thread locals become fiber local (for now)
    - ThreadLocal and the baggage that is InheritableThreadLocal, context ClassLoader, ..
    - Special case ThreadLocal for now to avoid needing Thread object

Thread Locals

- Spectrum of uses
  - Container managed cache of connection or credentials context
  - Approximating processor/core local in lower level libraries
  - …
- Significant topic for later

Footprint

- Thread
  - Typically 1MB reserved for stack + 16KB of kernel data structures
  - ~2300 bytes per started Thread, includes VM meta data
- Fiber
  - Continuation stack: hundreds of bytes to KBs
  - 200-240 bytes per fiber in current prototype

APIs that potentially park

- Thread sleep, join
- java.util.concurrent and LockSupport.park
- I/O
  - Networking I/O: socket read/write/connect/accept
  - File I/O
  - Pipe I/O

Communication between fibers

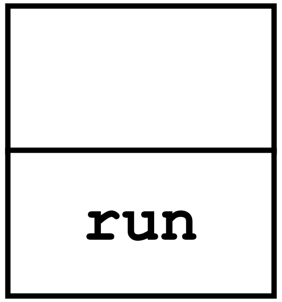
- Current prototype executes tasks as Runnable. Easy to use CompletableFuture too.
- j.u.concurrent *just works* so can share objects or share by communicating
- Not an explicit goal at this time to introduce new concurrency APIs but new APIs may emerge

Implementing Continuations

We need:

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

Native Stack



Continuation

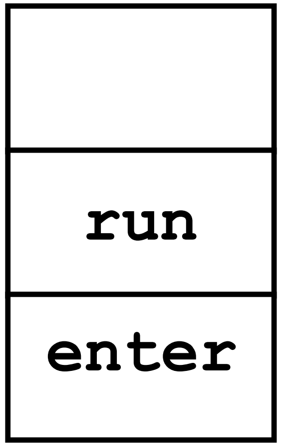
stack **refStack**

Native Stack

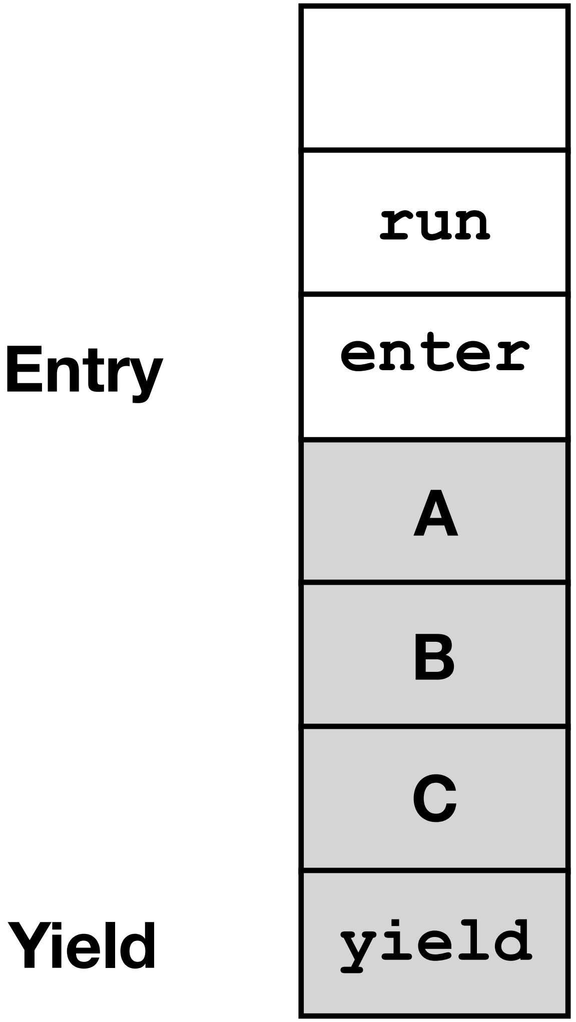
Continuation

stack **refStack**

Entry



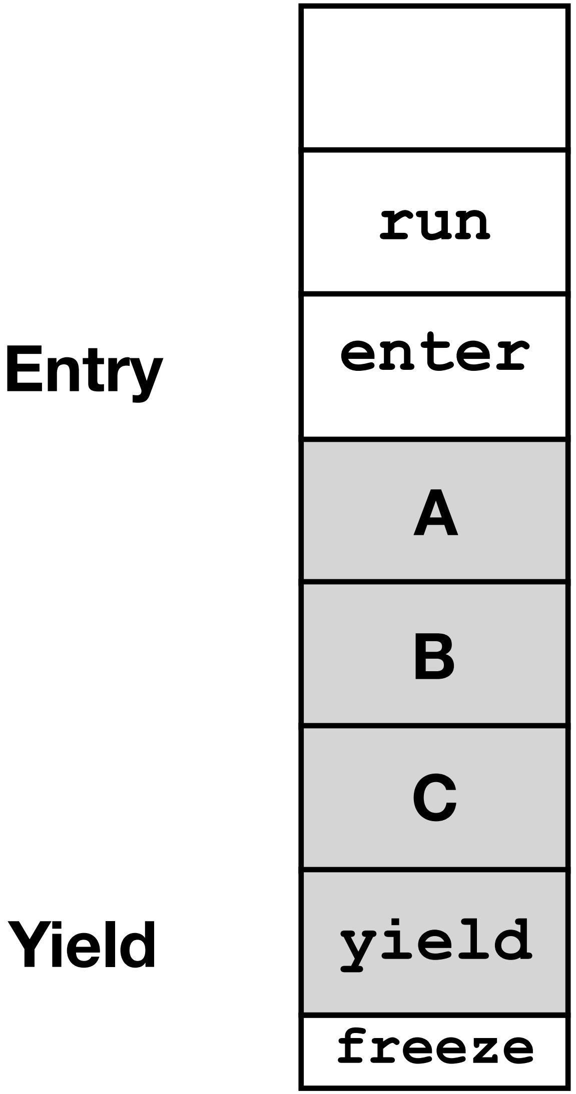
Native Stack



Continuation

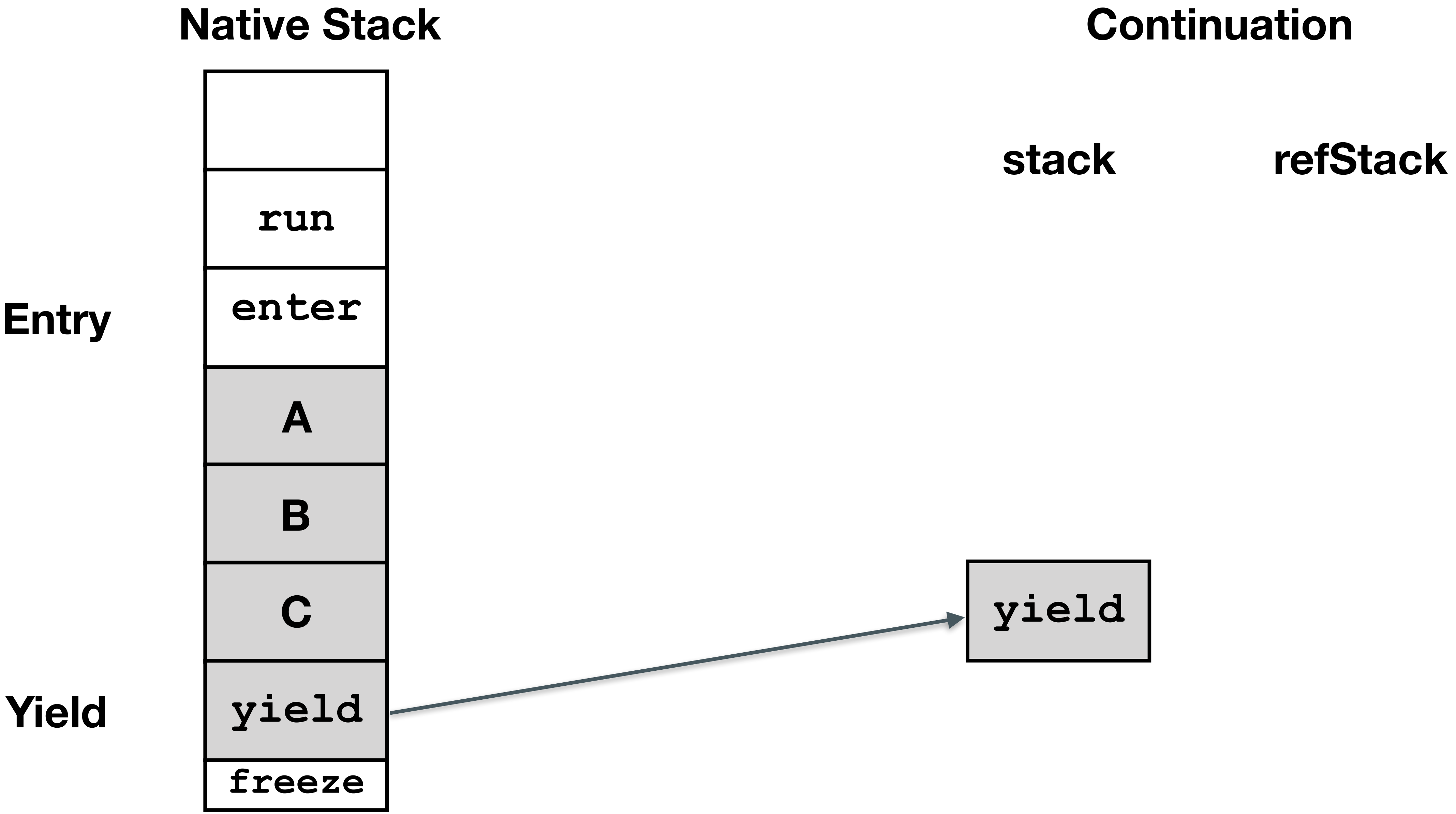
stack **refStack**

Native Stack



Continuation

stack **refStack**


Native Stack
Continuation
stack
refStack
run
Entry
enter
A
B
C
yield
Yield
yield
freeze

Native Stack

Continuation

stack

refStack

Entry

run

enter

A

B

C

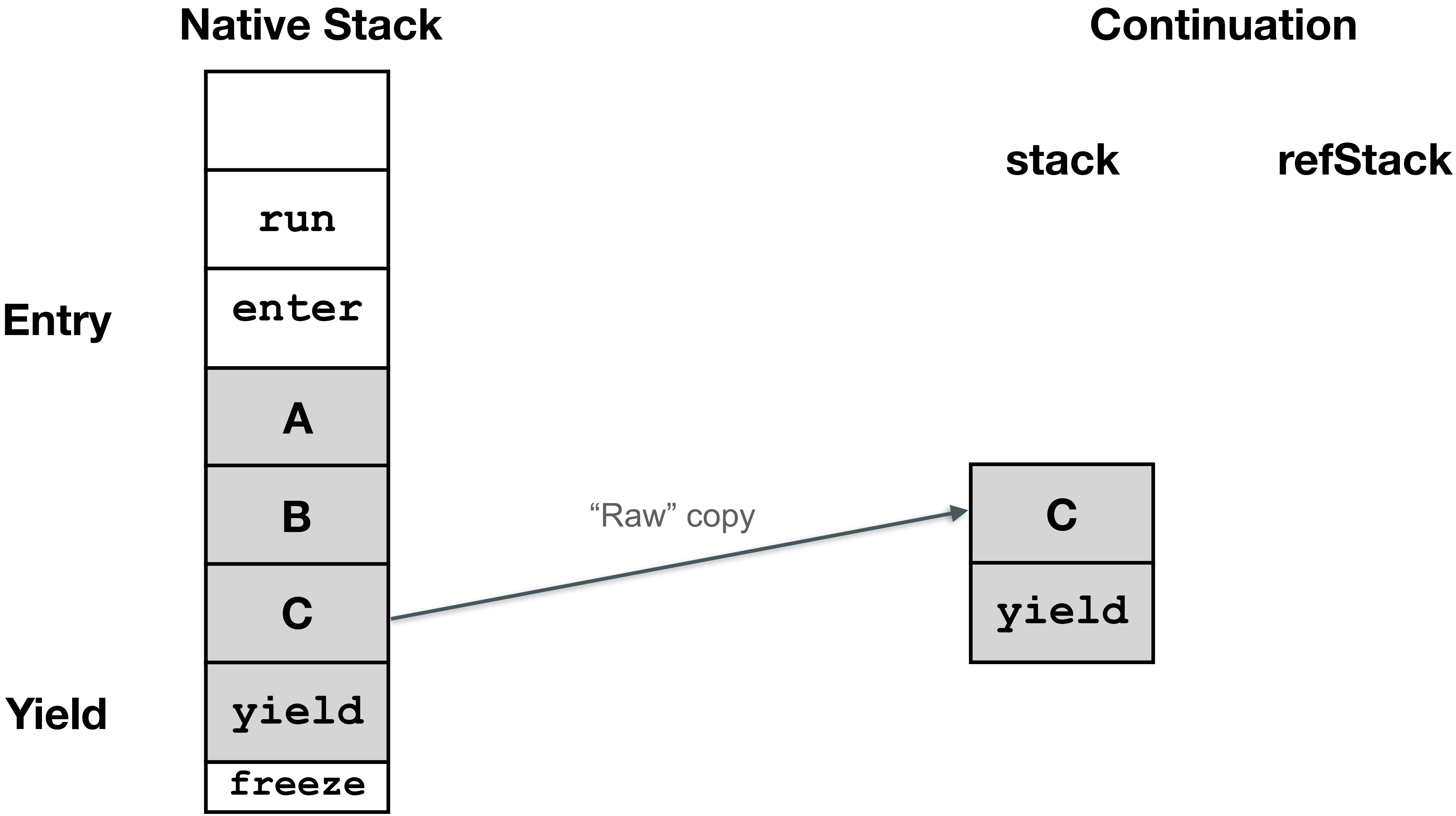
Yield

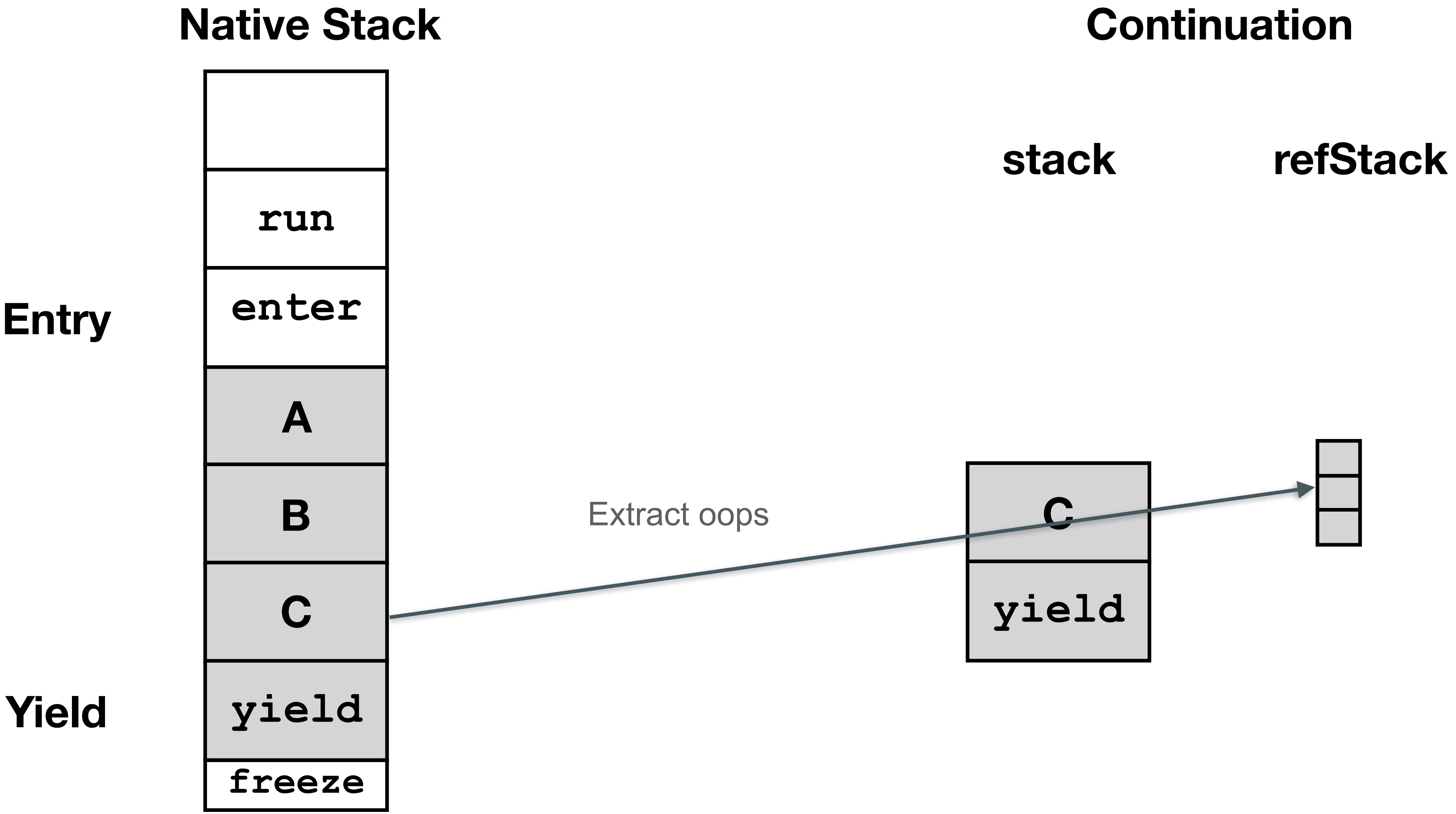
yield

freeze

Examine the frame for pinning

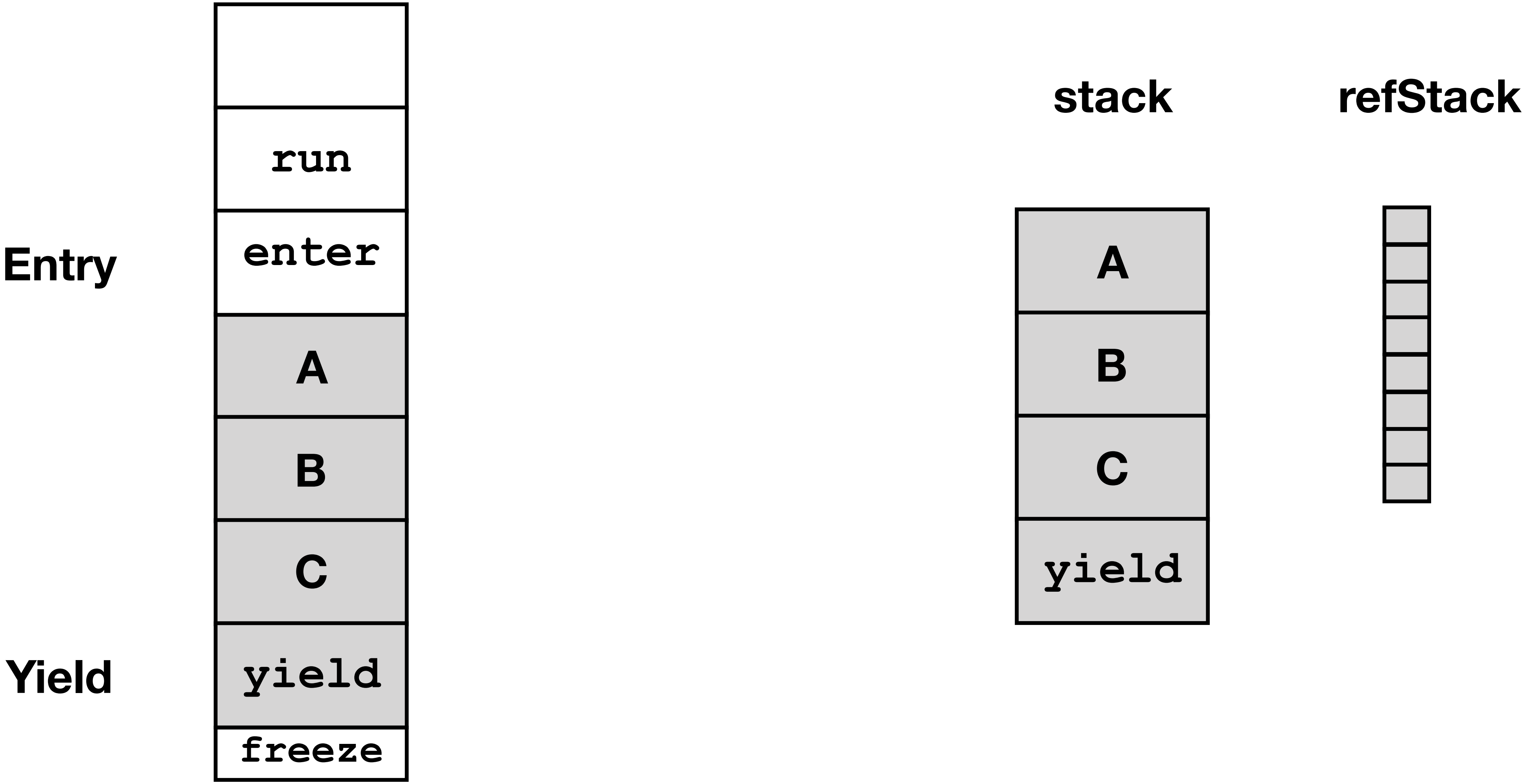
yield

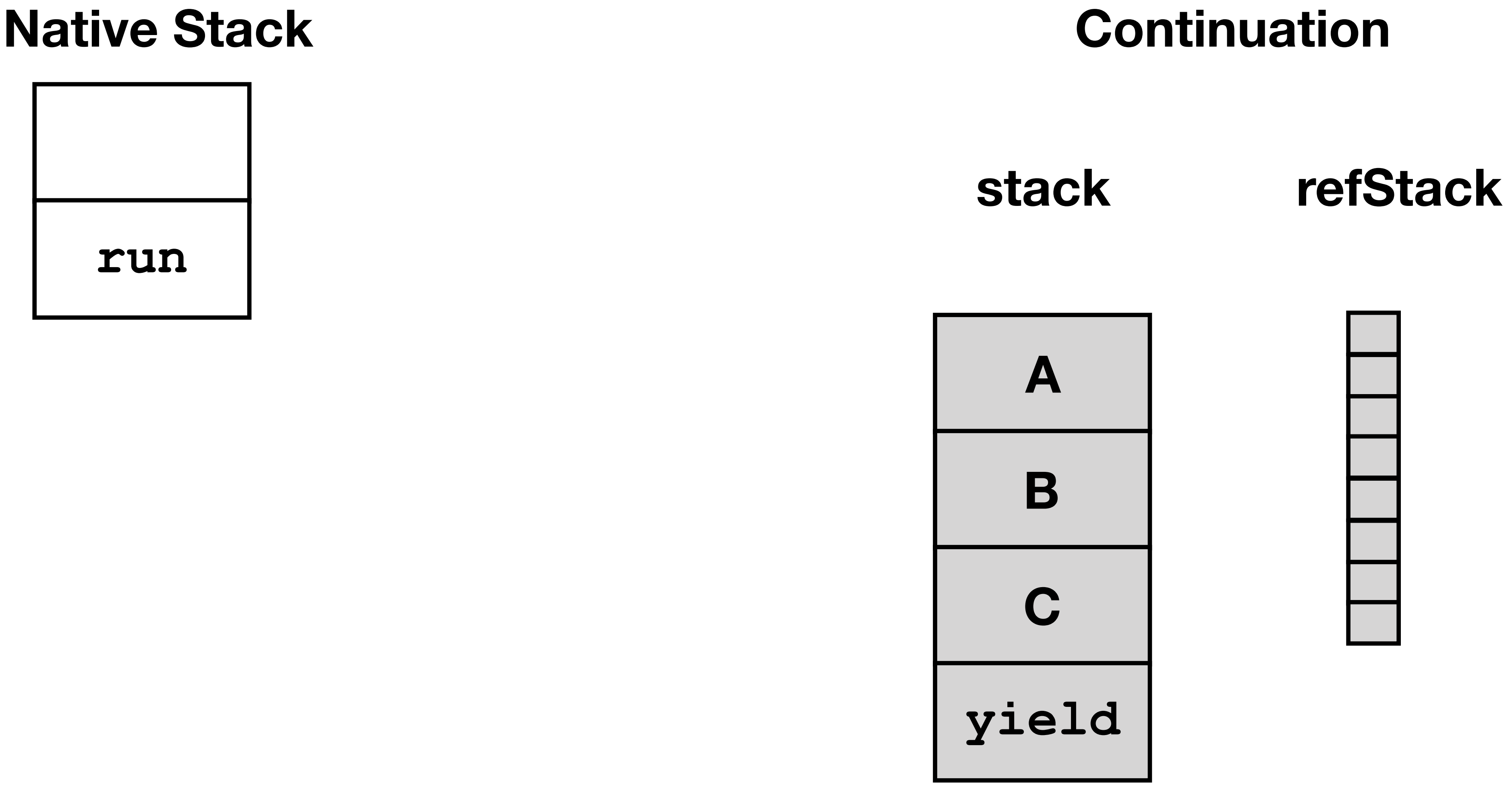

Native Stack
Continuation
stack
refStack
run
Entry
enter
A
B
“Raw” copy
C
C
yield
Yield
yield
freeze

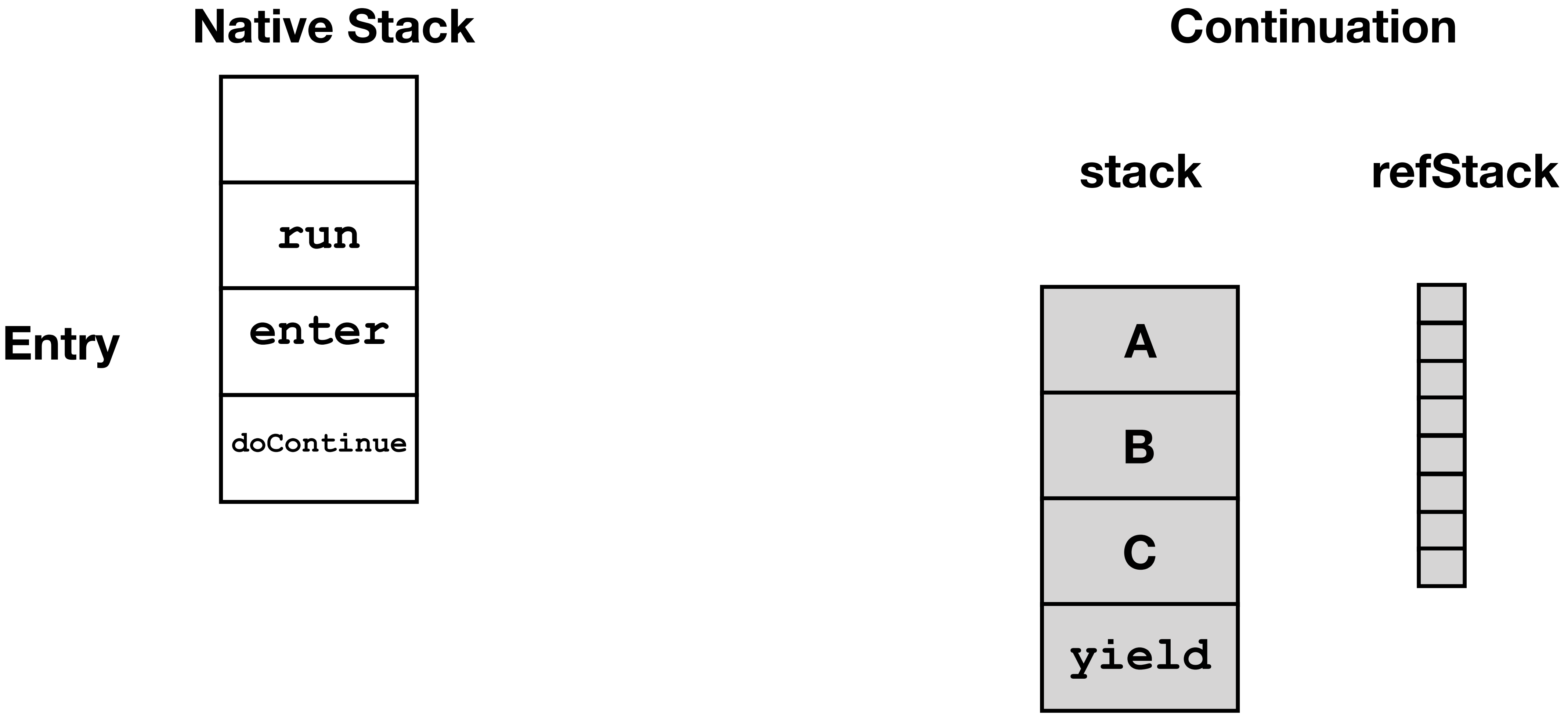

Native Stack
Continuation
stack
refStack
run
enter
Entry
A
B
C
Yield
yield
freeze
Extract oops
C
yield

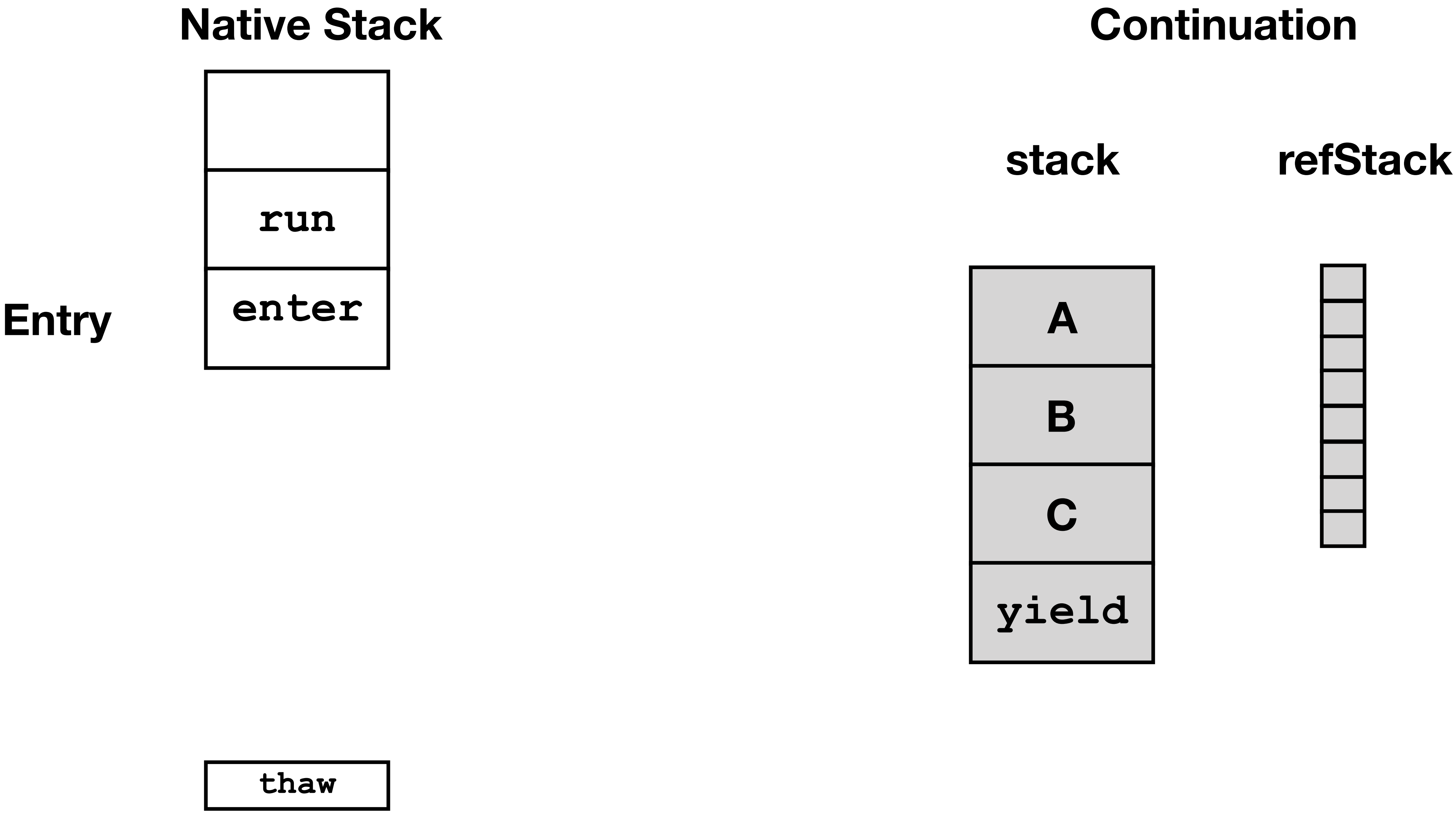
Native Stack

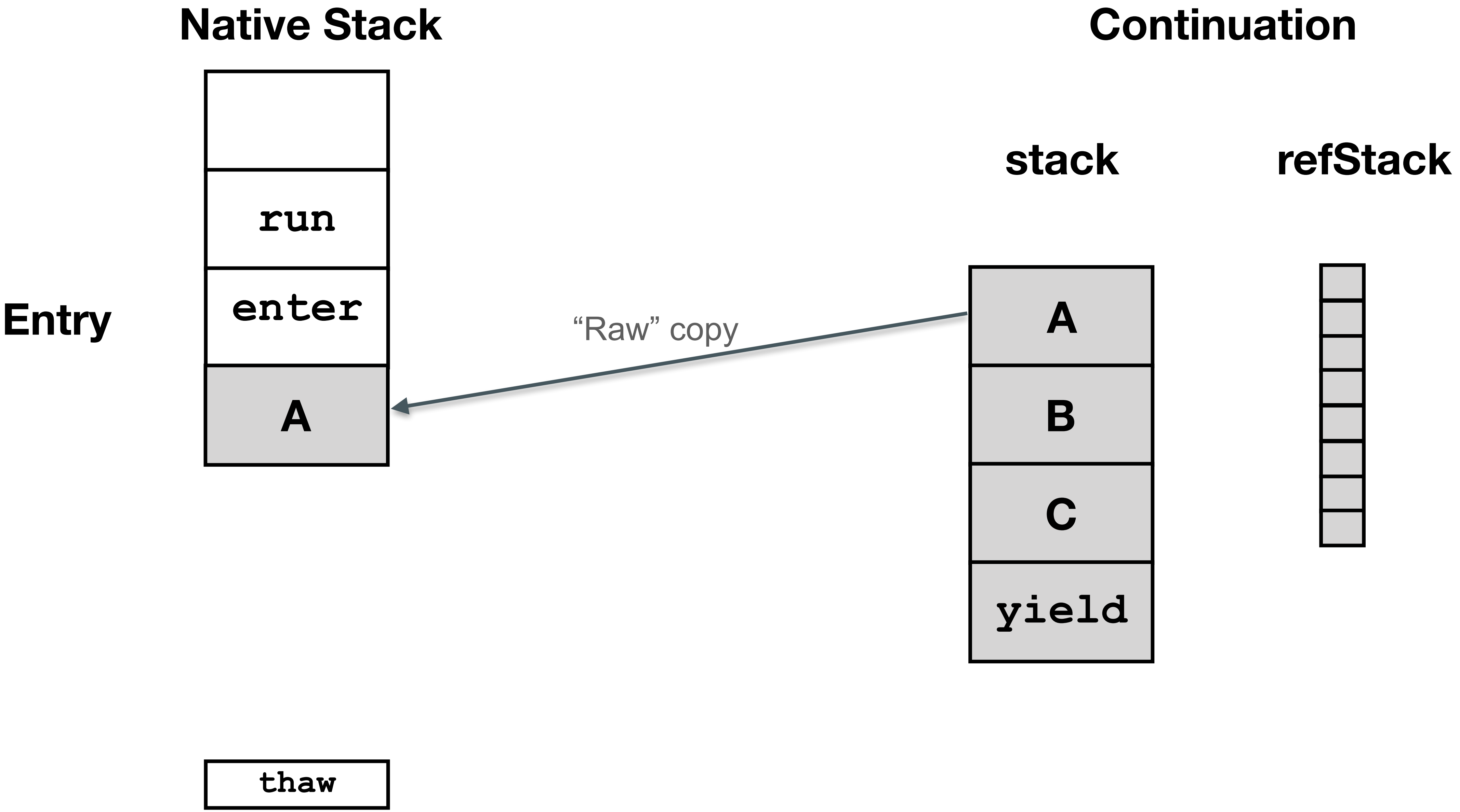
Continuation

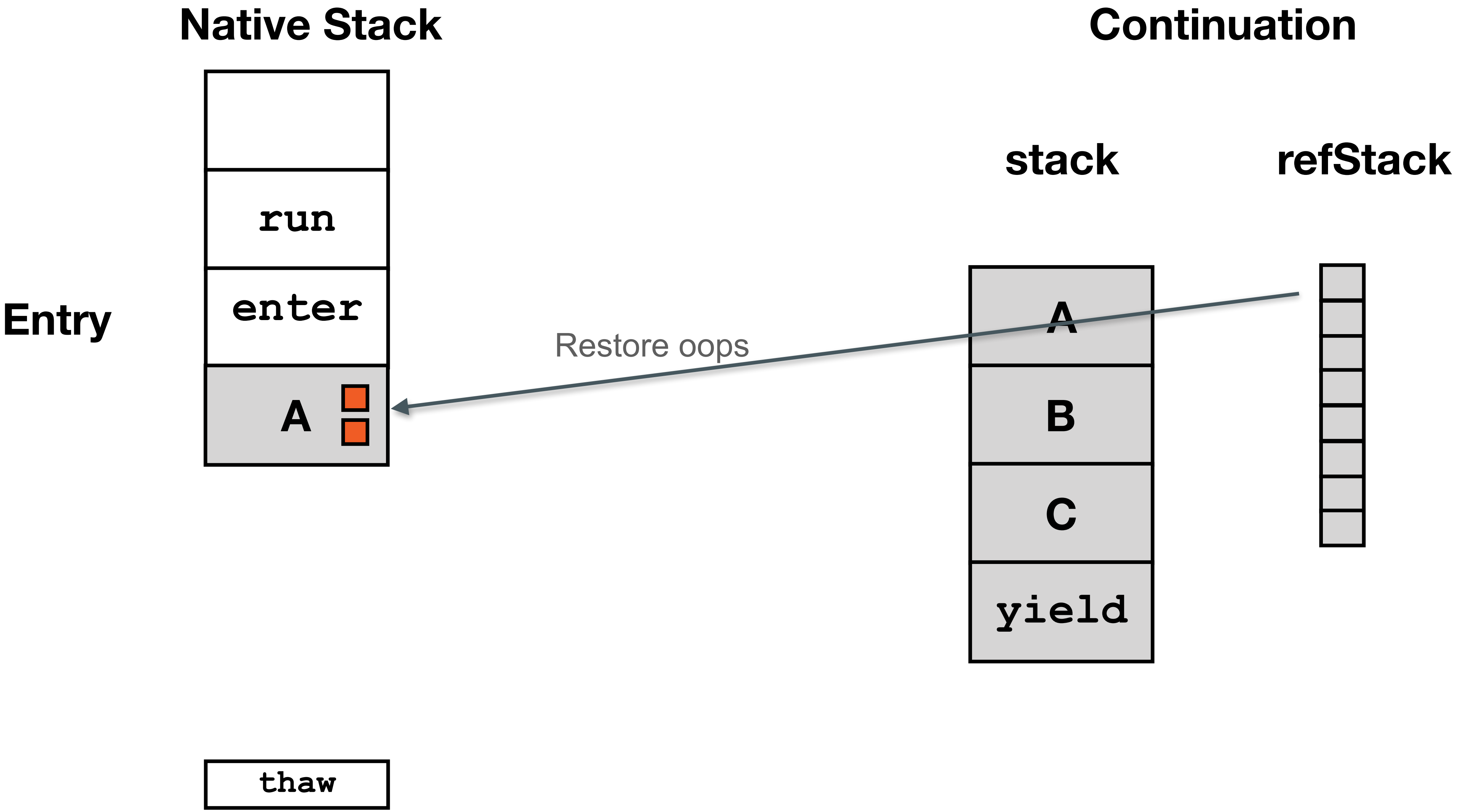


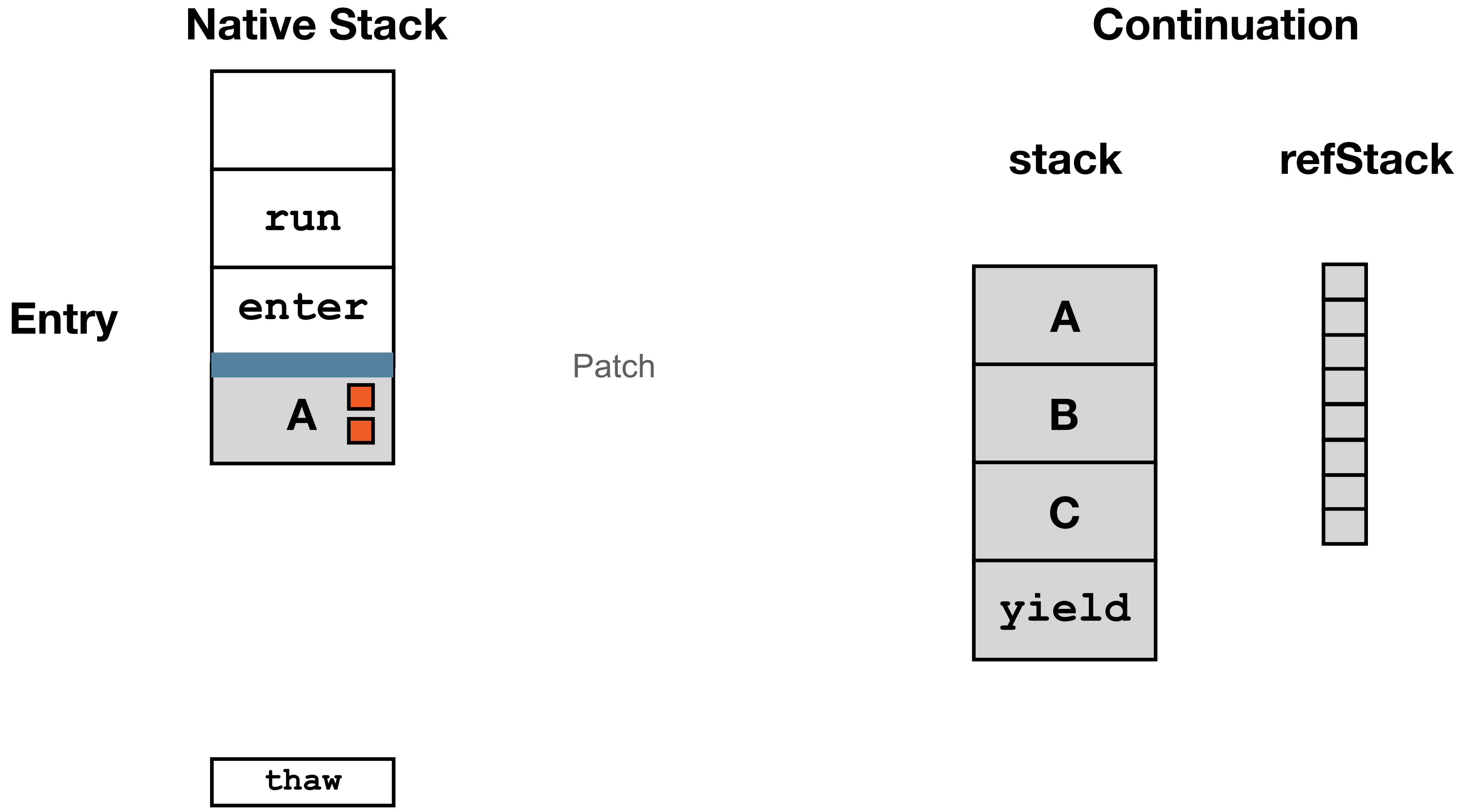

Native Stack
run
Continuation
stack
refStack
A
B
C
yield

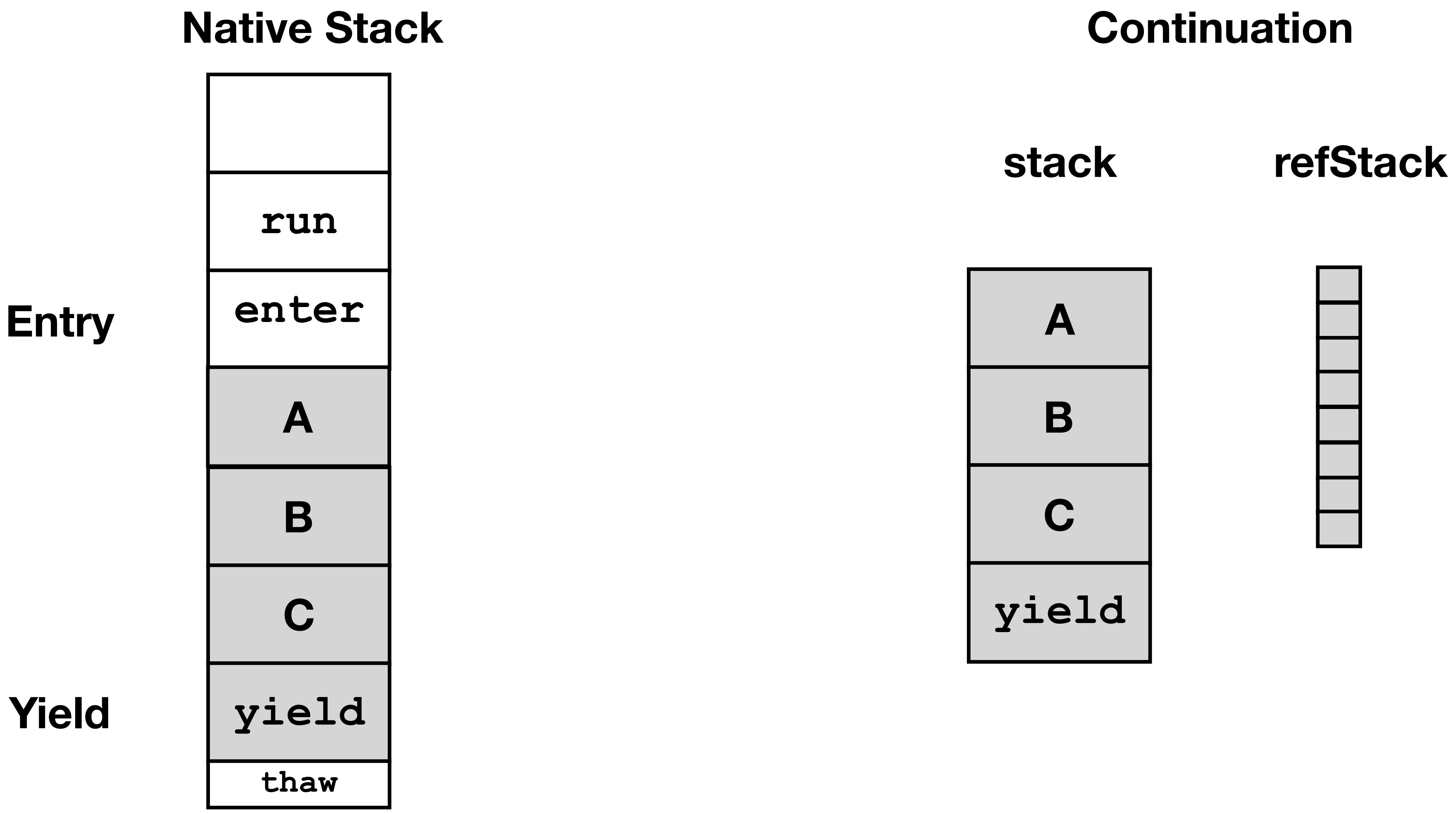

Native Stack
Continuation
stack
refStack
Entry
run
enter
doContinue
A
B
C
yield


Native Stack
Continuation
stack
refStack
run
enter
Entry
A
B
C
yield
thaw

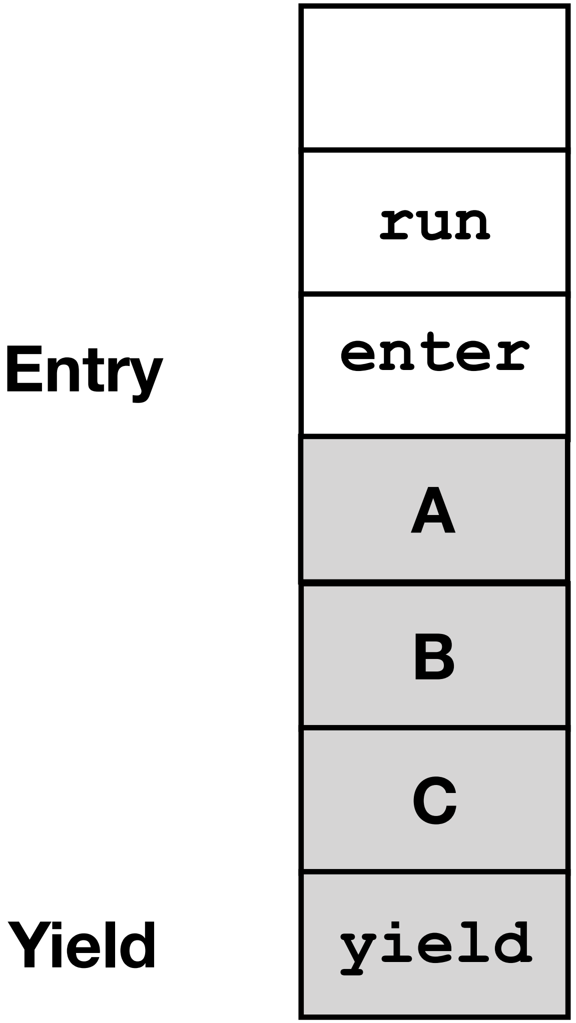

Native Stack
Continuation
stack
refStack
run
Entry
enter
"Raw" copy
A
A
B
C
yield
thaw


Native Stack
Continuation
stack
refStack
run
Entry
enter
Restore oops
A
A
B
C
yield
thaw


Native Stack
Continuation
stack
refStack
run
enter
Entry
Patch
A
A
B
C
yield
thaw


Native Stack
Continuation
stack
refStack
Entry
Yield
run
enter
A
B
C
yield
thaw
A
B
C
yield

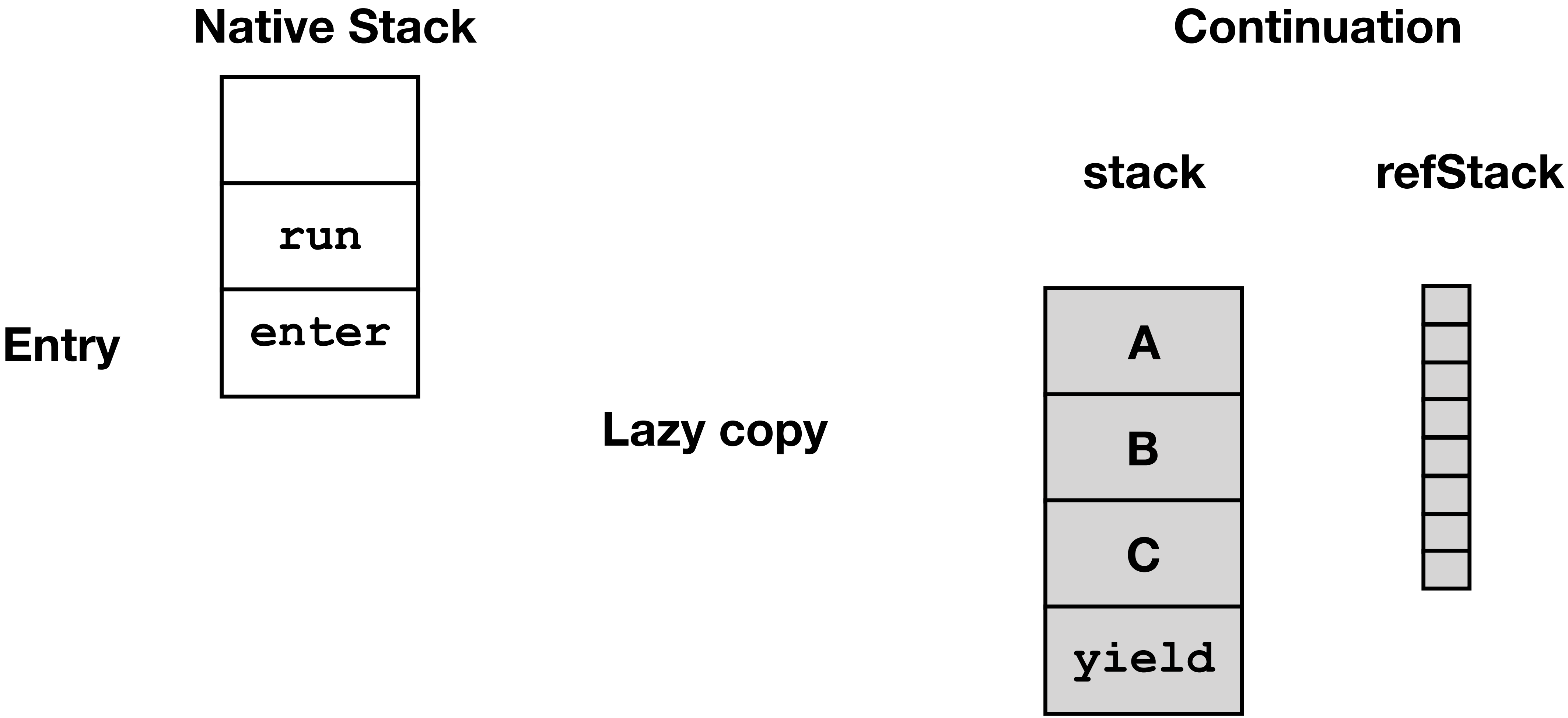
Native Stack

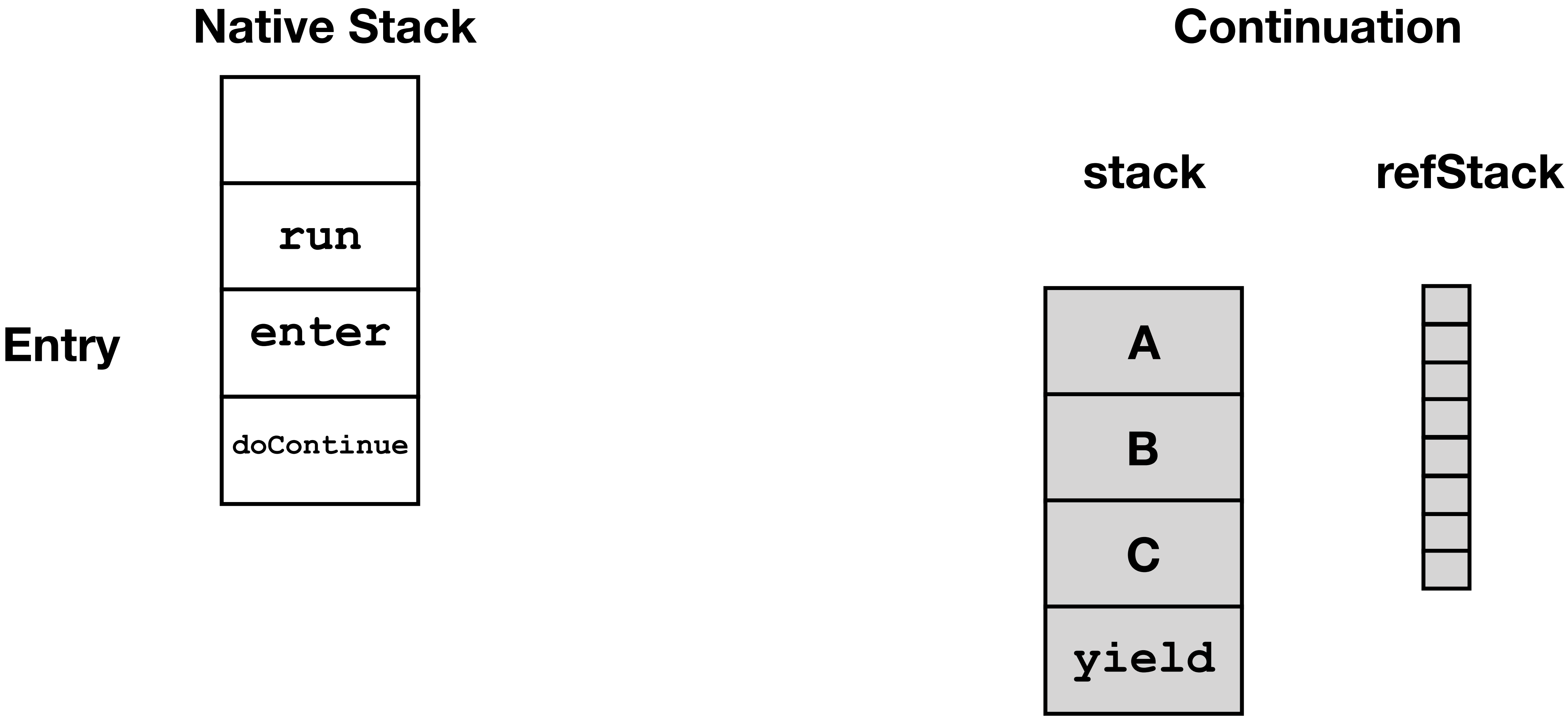


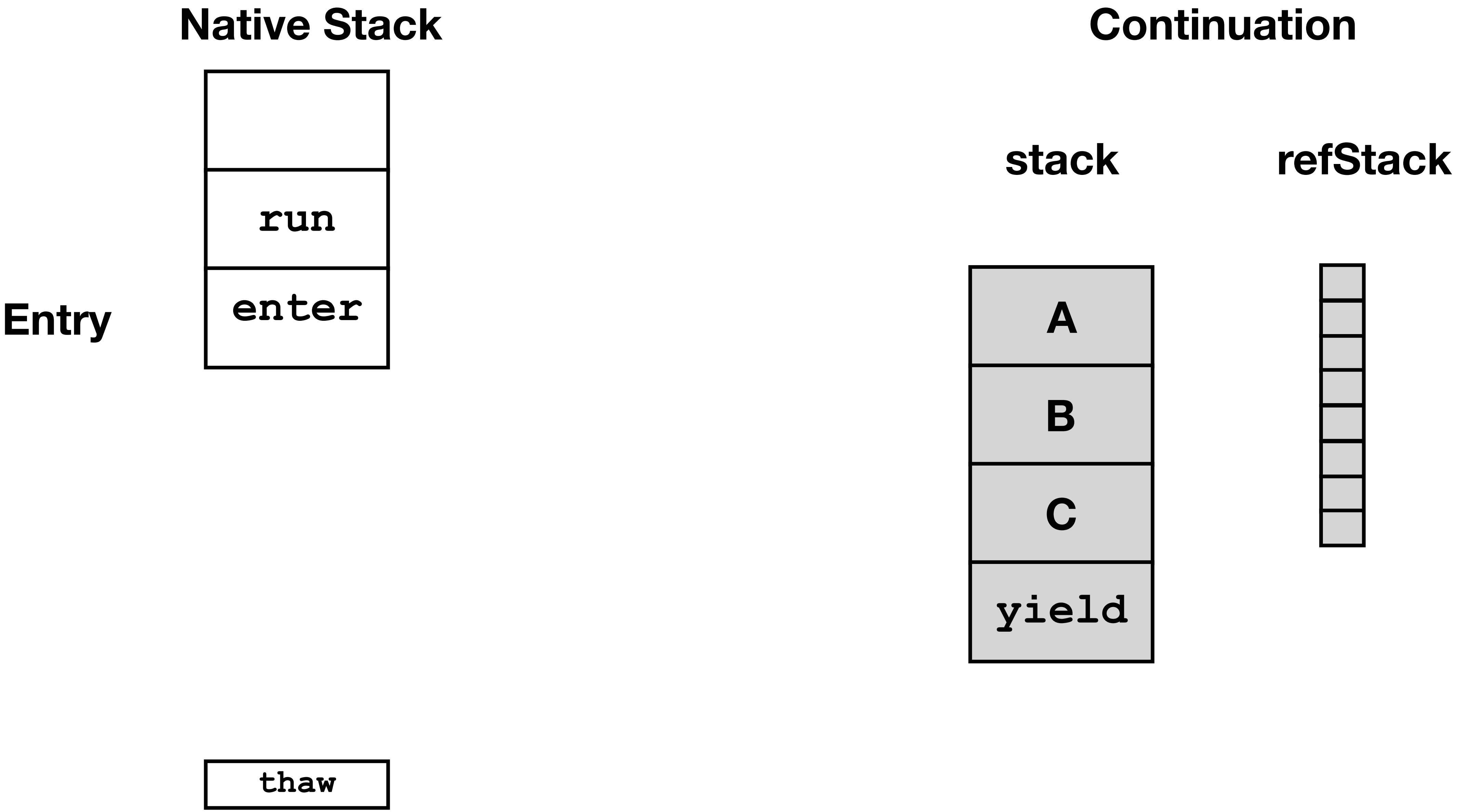
Continuation

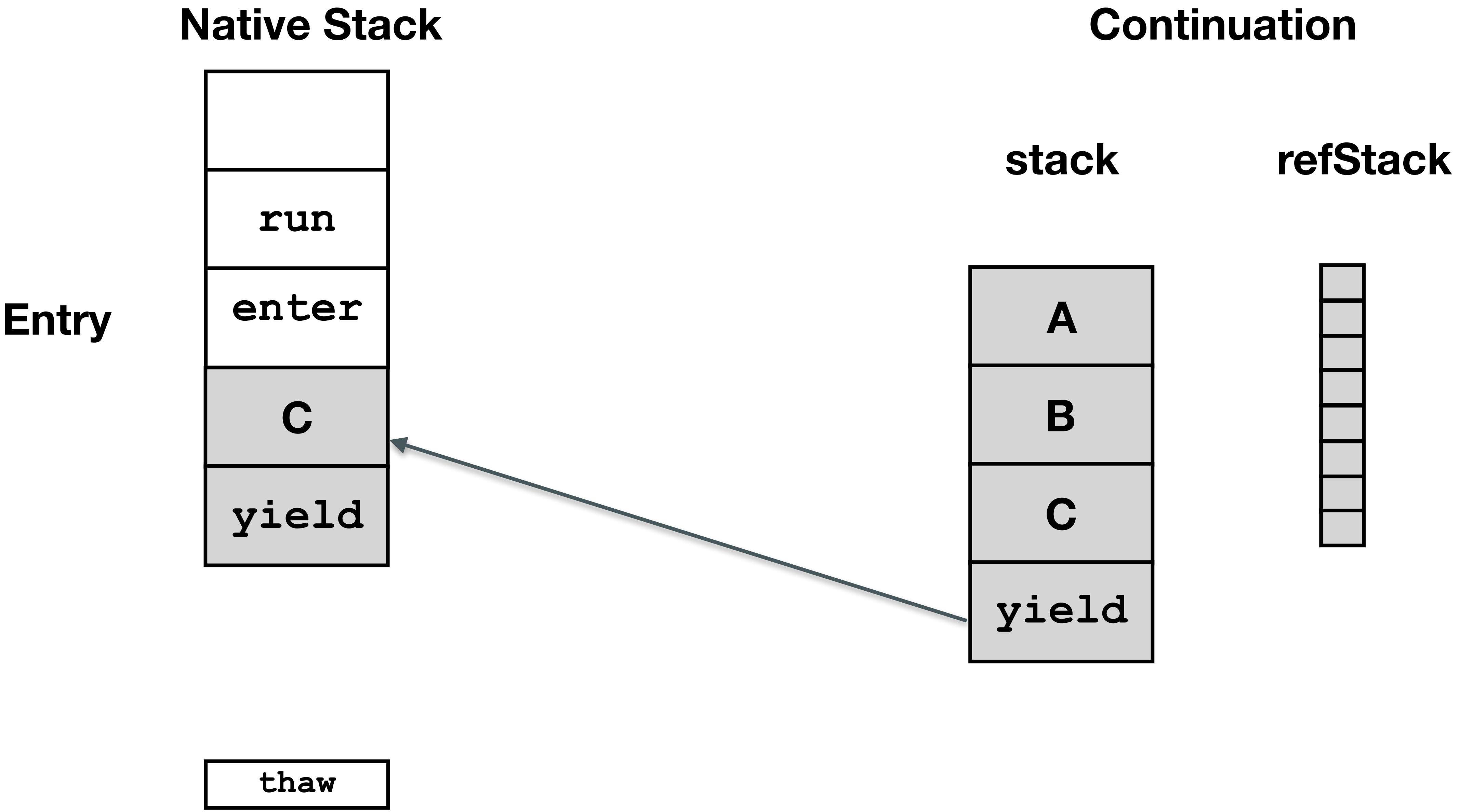
stack

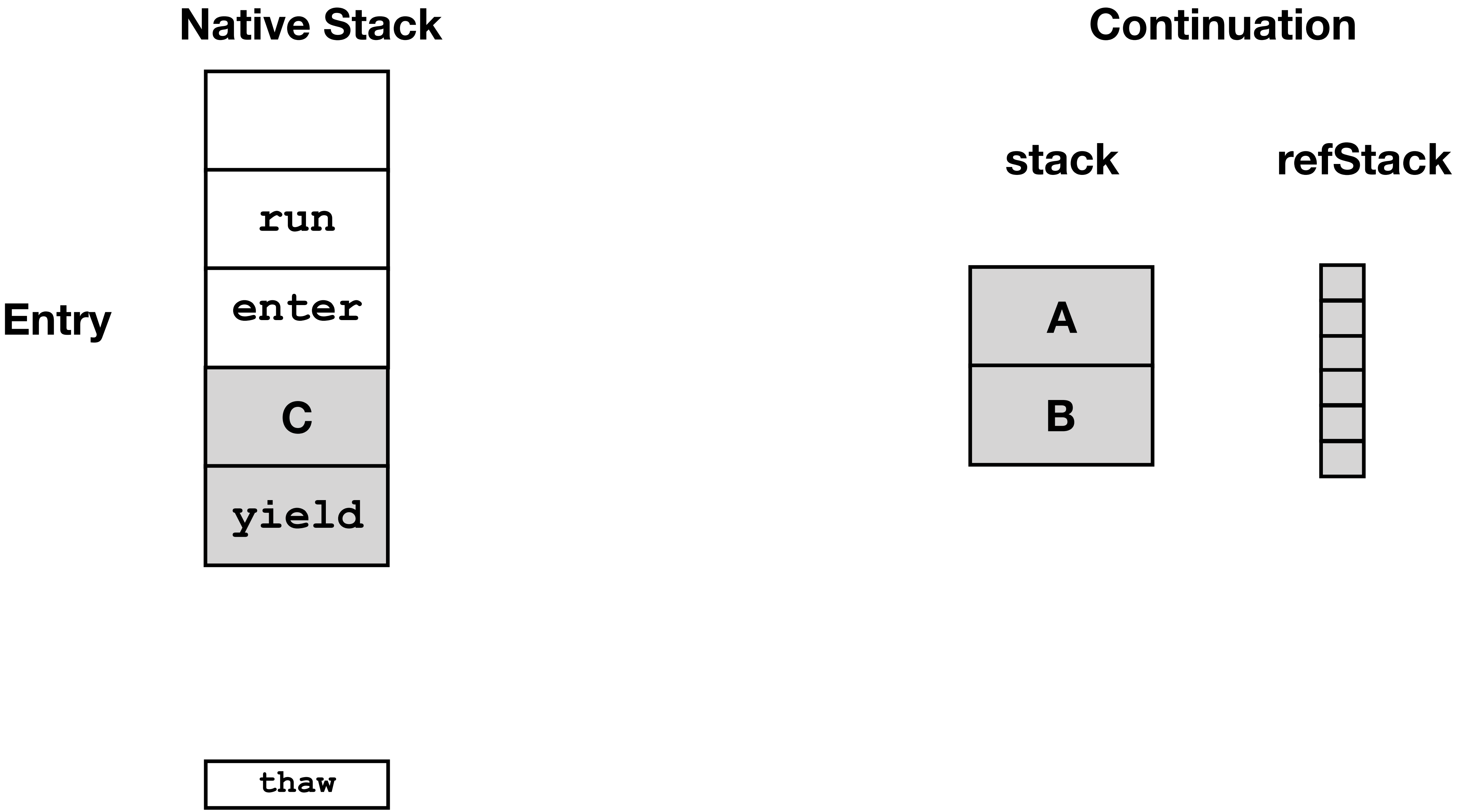
refStack


Native Stack
Continuation
stack
refStack
run
enter
Entry
Lazy copy
A
B
C
yield

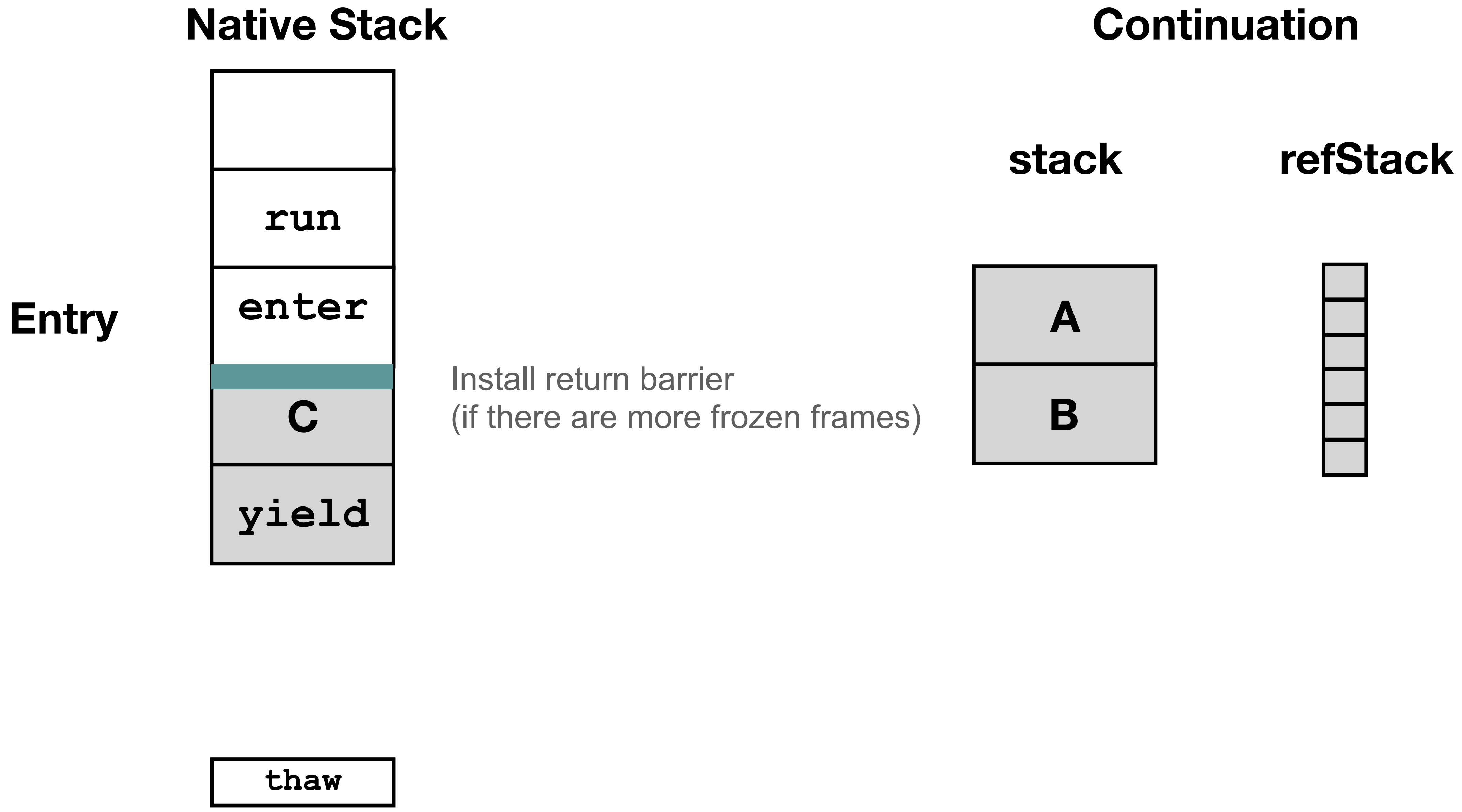

Native Stack
Continuation
stack
refStack
Entry
run
enter
doContinue
A
B
C
yield

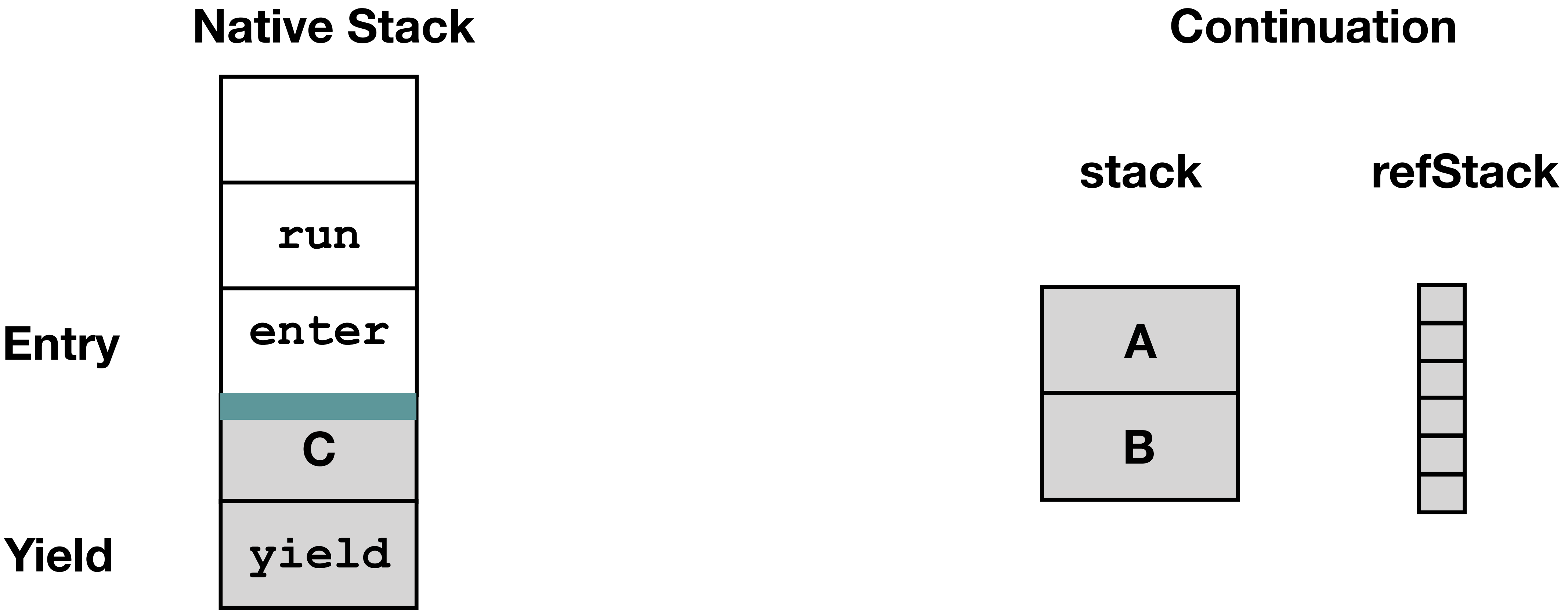

Native Stack
run
enter
Entry
thaw
Continuation
stack
refStack
A
B
C
yield


Native Stack
Continuation
stack
refStack
Entry
run
enter
C
yield
A
B
C
yield
thaw

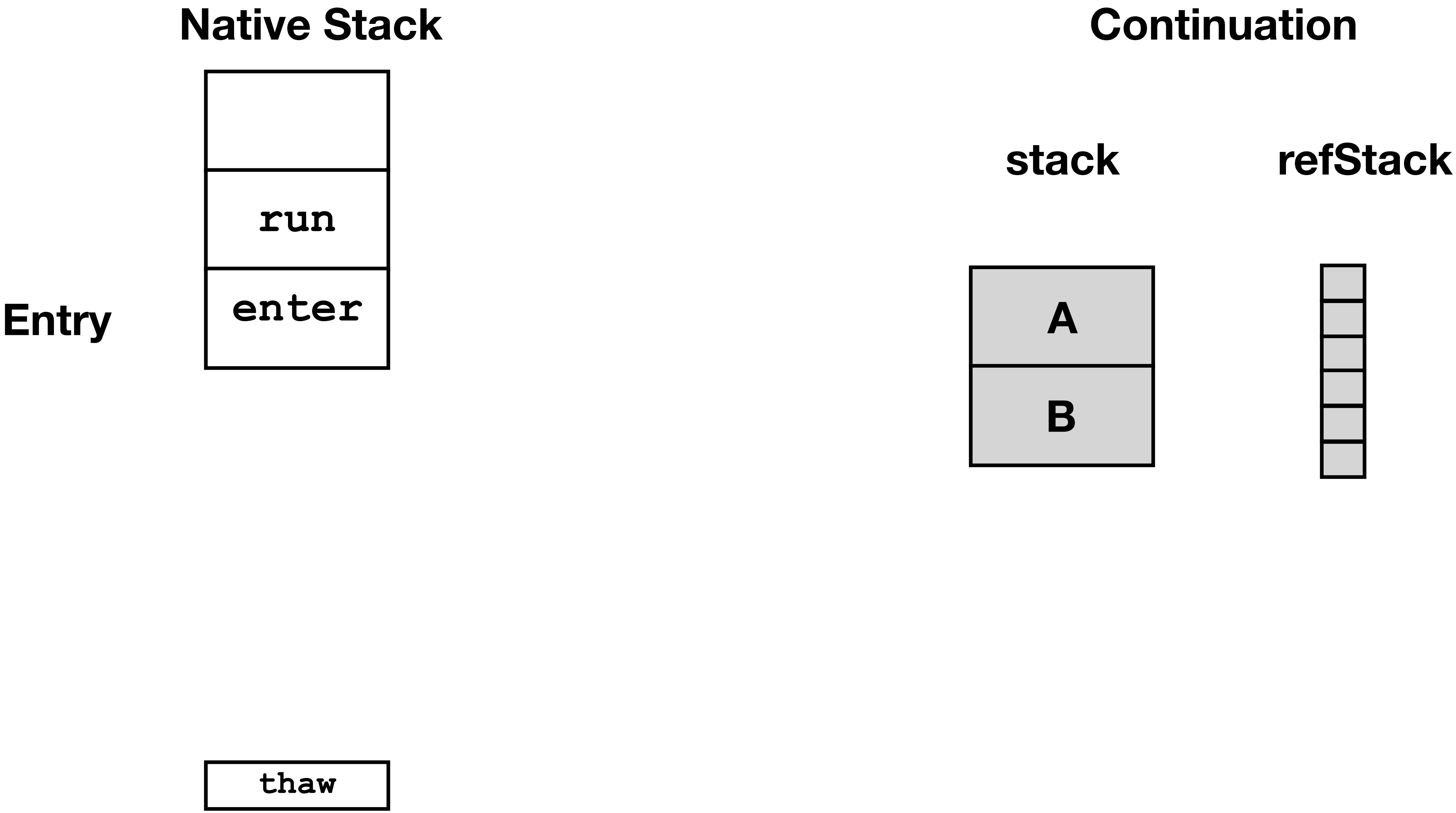


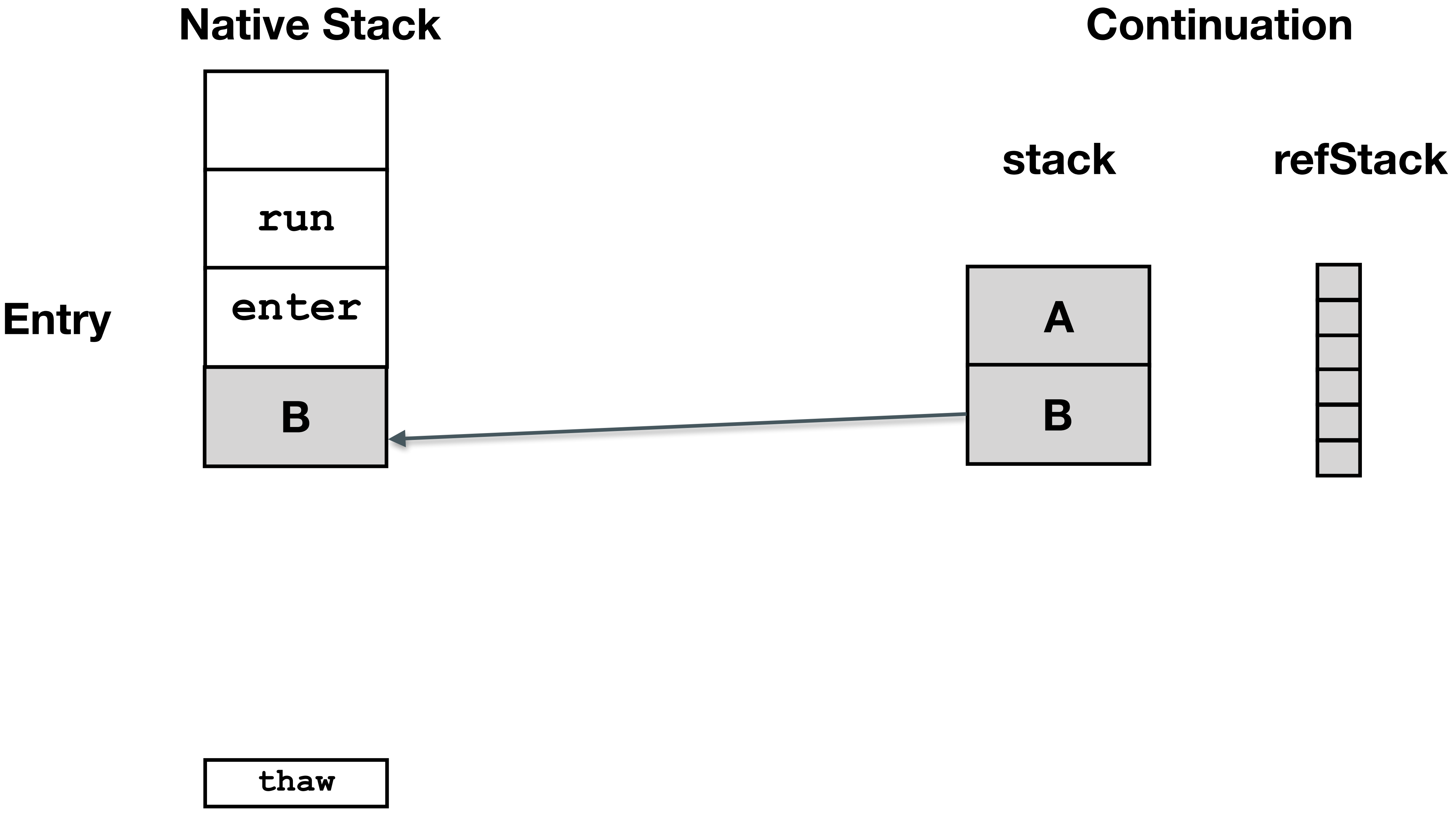
Native Stack
Continuation
stack
refStack
run
Entry
enter
C
yield
A
B
thaw

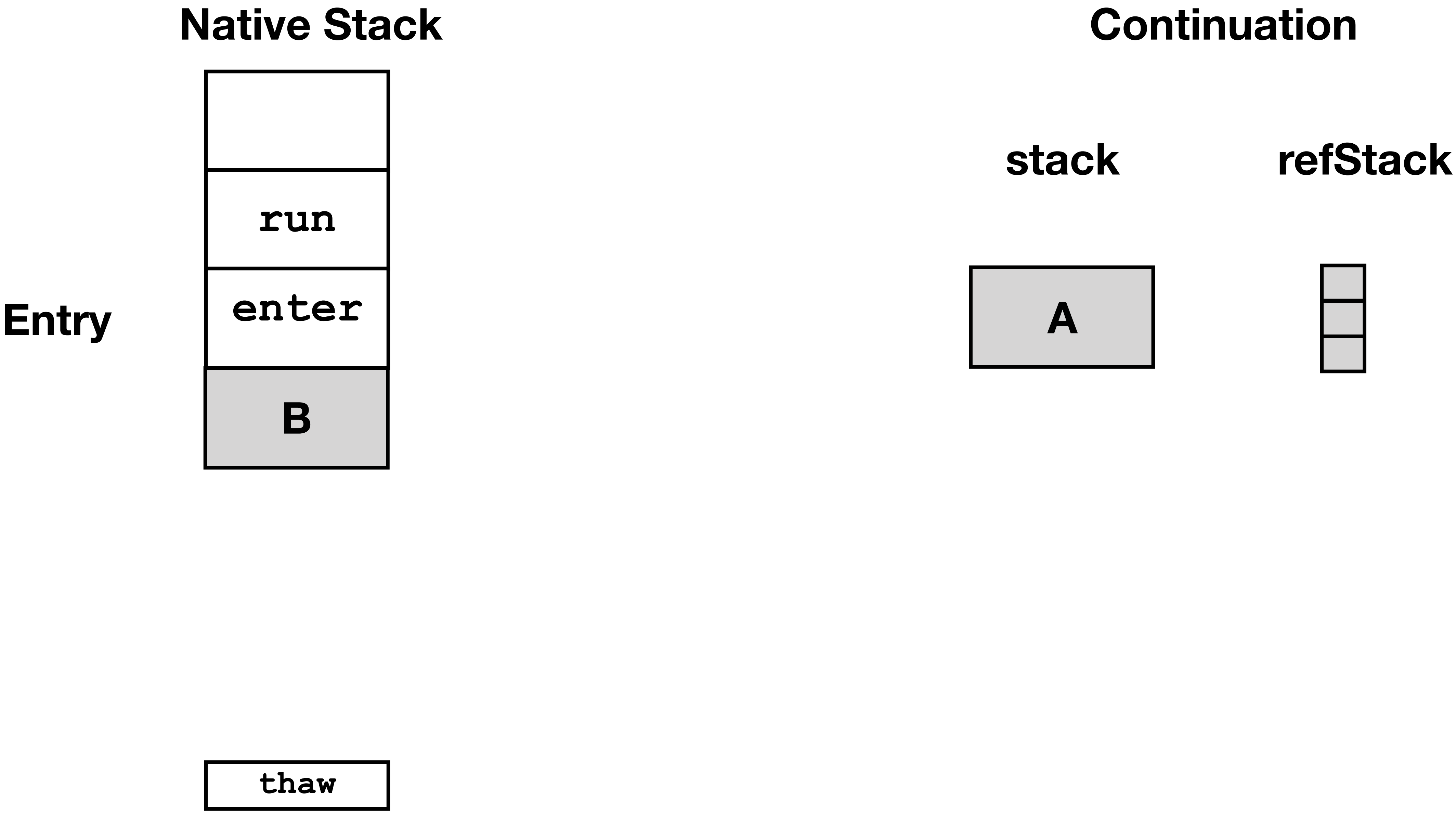

Native Stack
Continuation
stack
refStack
run
Entry
enter
A
Install return barrier
(if there are more frozen frames)
C
B
yield
thaw

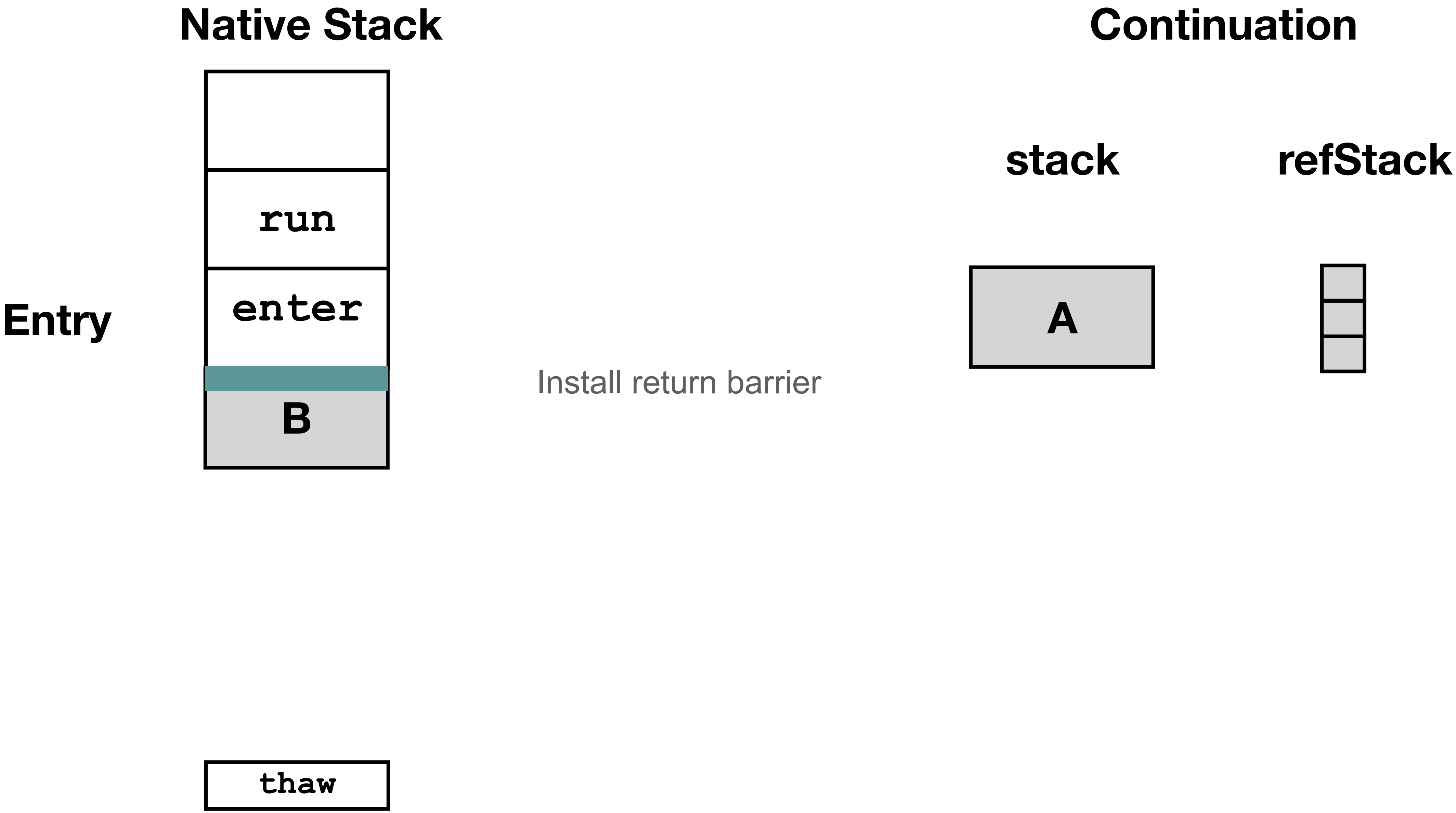

Native Stack
run
enter
C
yield
Entry
Yield
Continuation
stack
refStack
A
B


Native Stack
Continuation
stack
refStack
run
enter
Entry
C
A
B

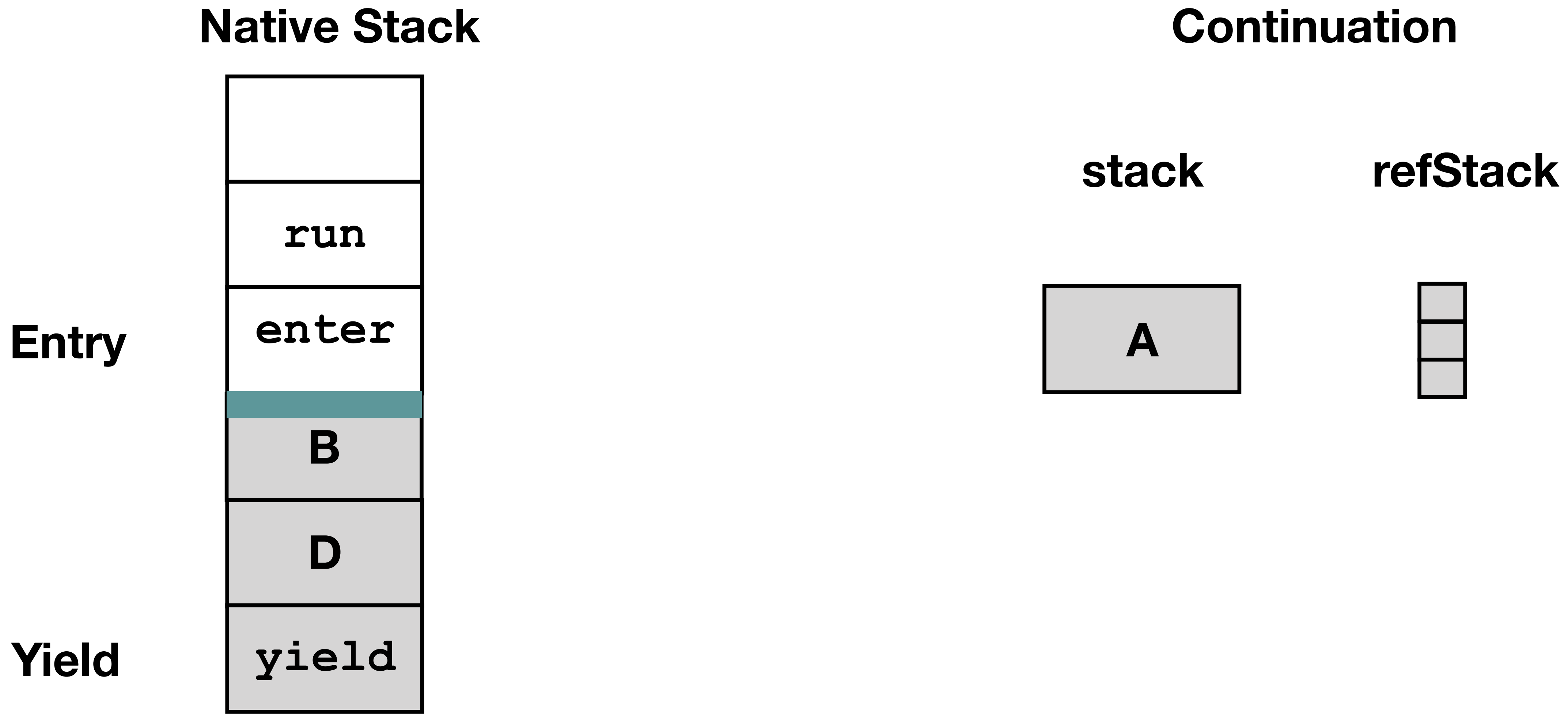

Native Stack
Continuation
stack
refStack
run
enter
Entry
A
B
thaw

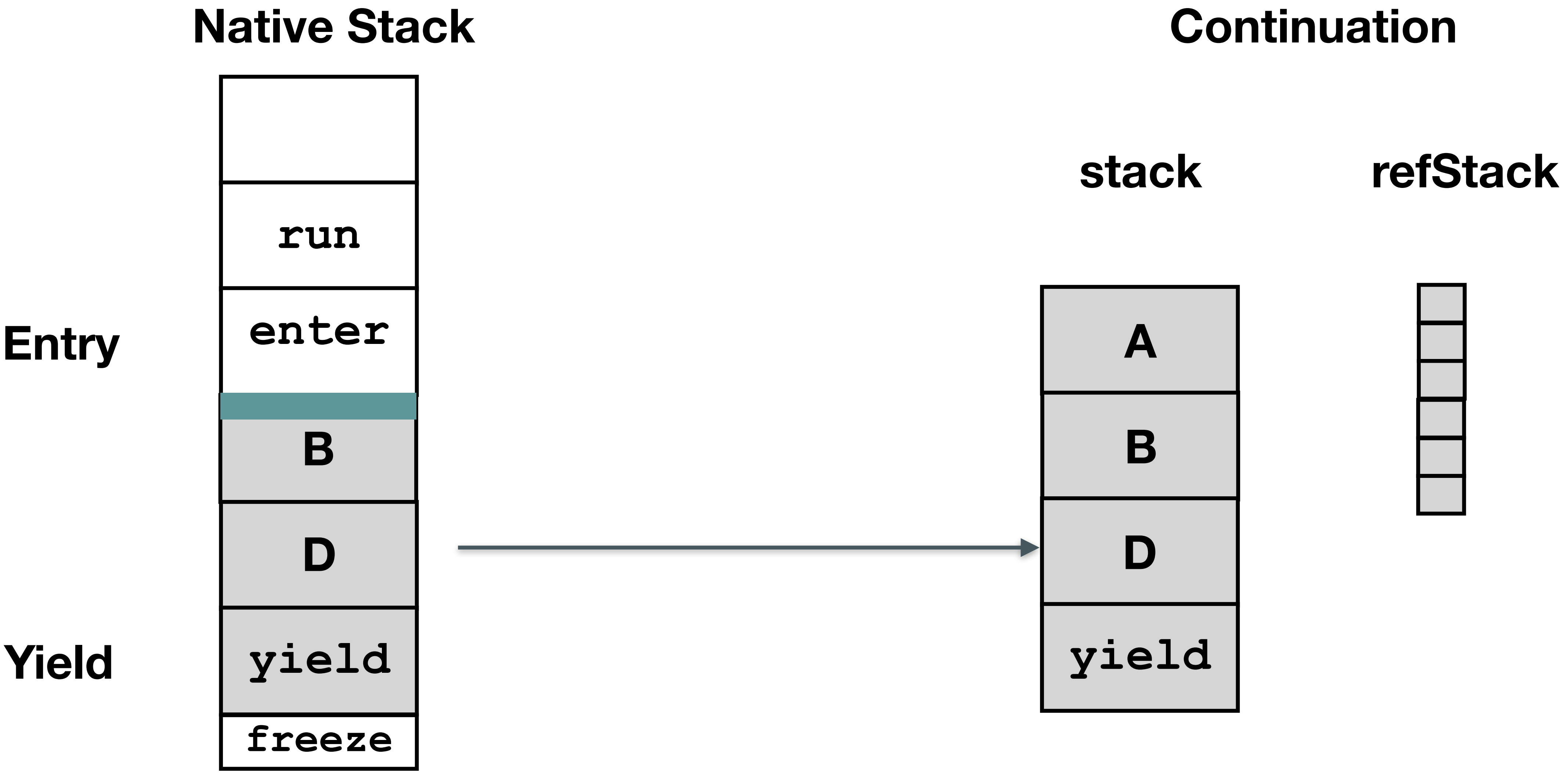

Native Stack
Continuation
stack
refStack
run
enter
Entry
B
A
B
thaw


Native Stack
Continuation
stack
refStack
run
Entry
enter
A
B
thaw

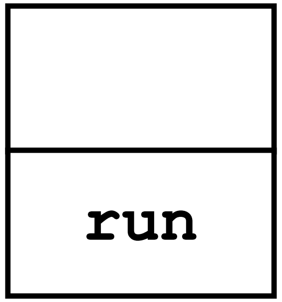


Install return barrier

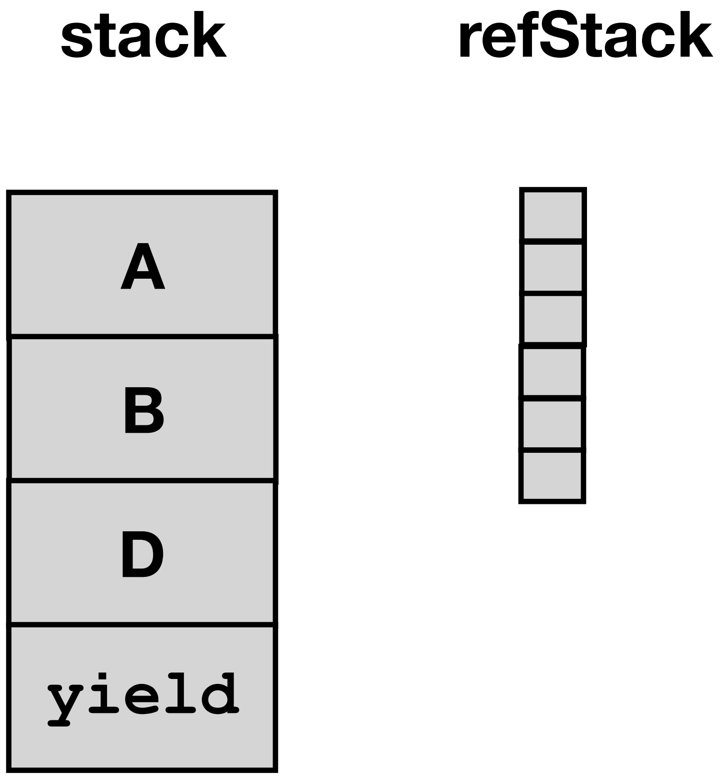

Native Stack
Continuation
stack
refStack
run
Entry
enter
A
B
D
Yield
yield


Native Stack
Continuation
stack
refStack
run
enter
Entry
B
D
Yield
yield
freeze
A
B
D
yield

Native Stack



Continuation



Epilogue

Features not in current prototype

- Serialization and cloning
- JVM TI and debugging support for fibers
- Tail calls

Next Steps

- Design behavior and API
- Add missing features
- Improve performance

More information

- Project Loom page: http://openjdk.java.net/projects/loom/
- Mailing list: loom-dev@openjdk.java.net
- Repo: http://hg.openjdk.java/net/loom/loom

Java™

ORACLE®