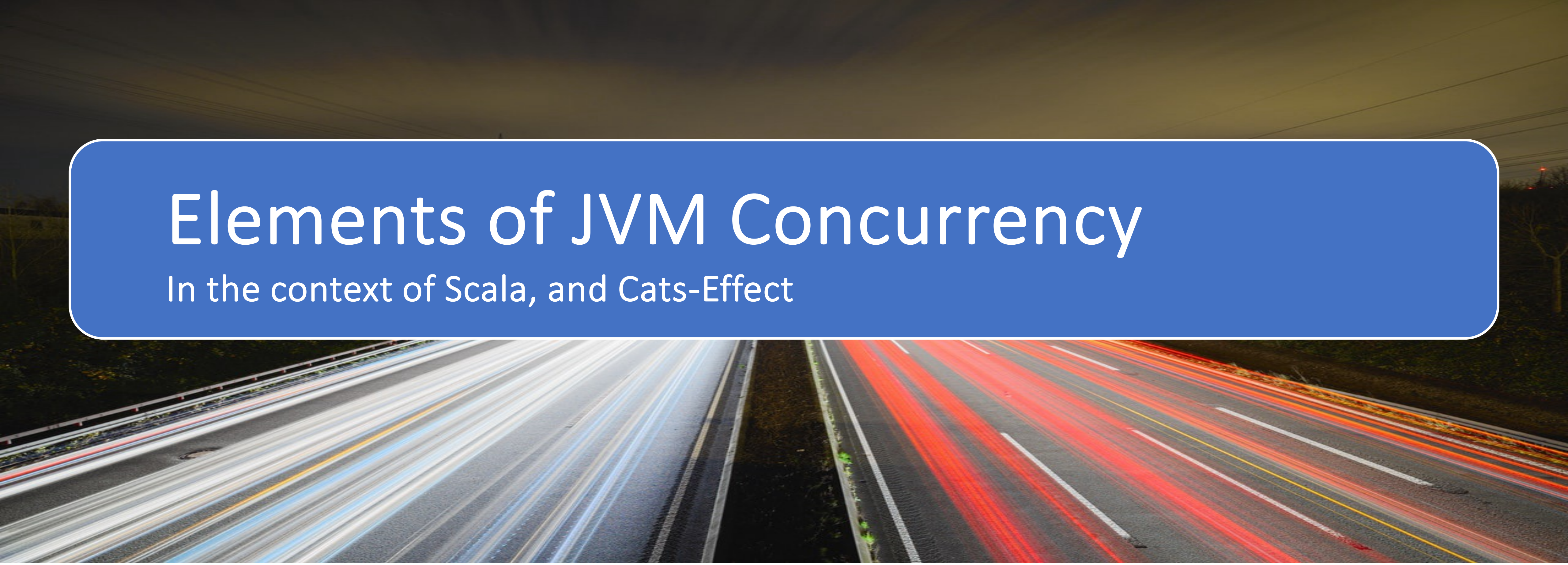


Elements of JVM Concurrency

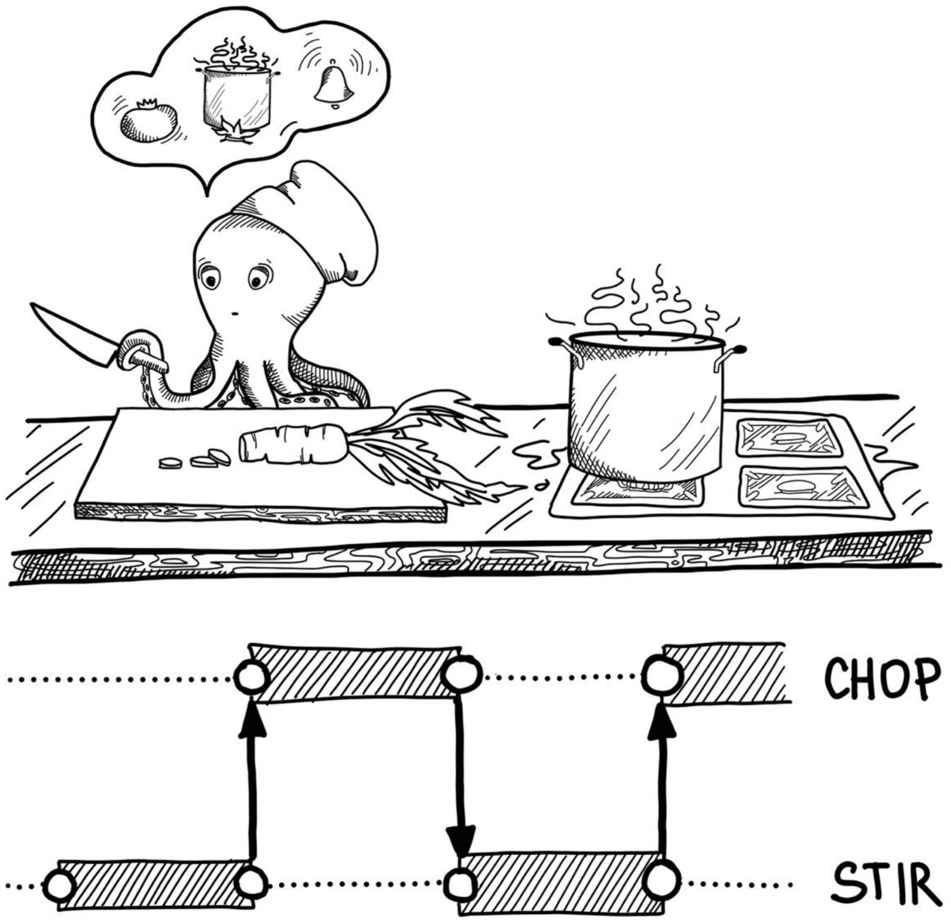
In the context of Scala, and Cats-Effect

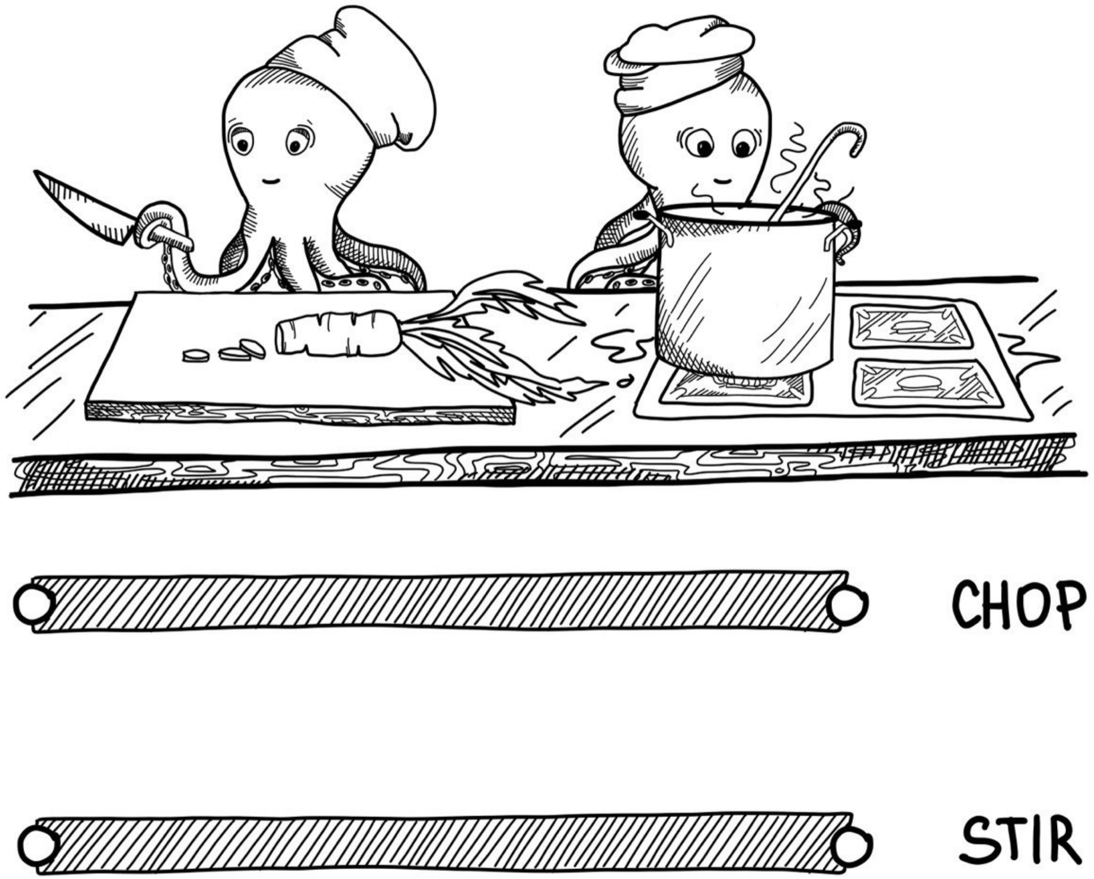
Alexandru Nedelcu
alexn.org

Plan

- Synchronization
  - Monitors, atomic references, Ref, volatiles
- Interruption
  - Java, Cats-Effect
- Goal: confidence boost

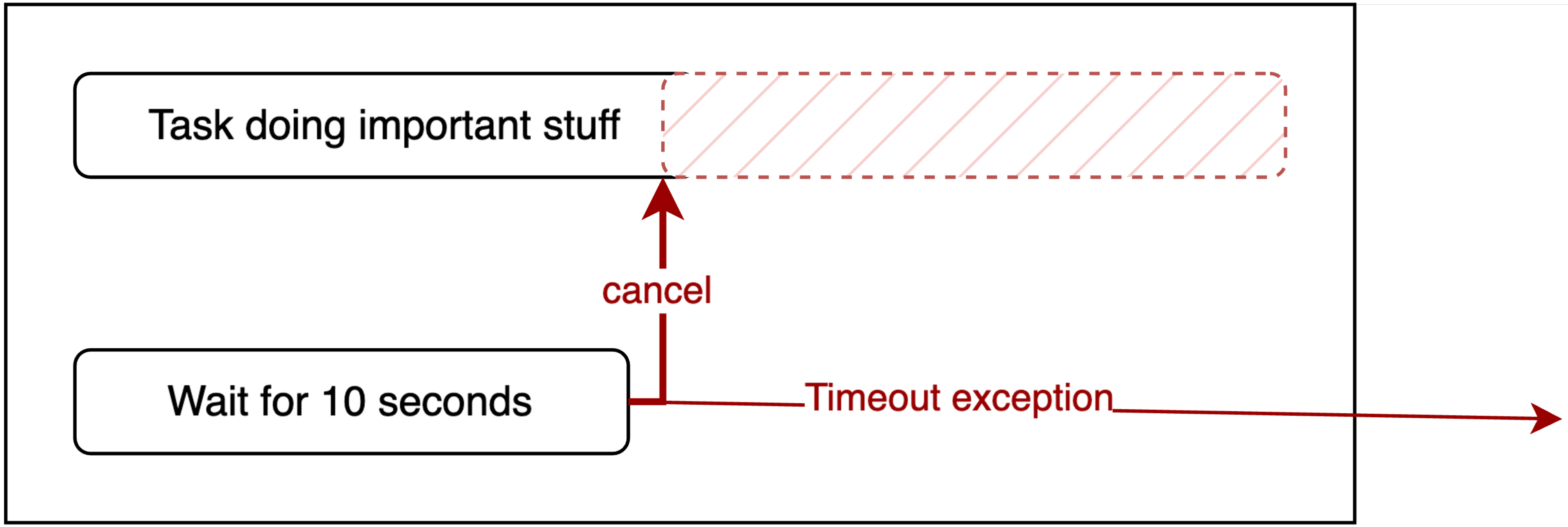
Concurrency vs. Parallelism





Source: https://freecontent.manning.com/concurrency-vs-parallelism/

Interruption



```
val th = new Thread:
    while !Thread.interrupted() do
        process()

th.start()
//...
th.interrupt()
```

```
val th = new Thread:
    while true do
        process()
        Thread.sleep(1000)

th.start()
//...
th.interrupt()
```

Java interruption is usable, but error prone

1. Must own the thread, or do expensive synchronization;
2. Code can catch and ignore interruption;
3. Interruption is too cooperative.

Just use **cats.effect.IO**

```
val task =
    IO.interruptible:
        while !Thread.interrupted() do
            process()
            Thread.sleep(1000)

    .timeout(30.seconds)
```

```scala
trait ConcurrentQueue[A]:
    def push(a: A): Unit
    def pop(): A
```

```
ref.synchronized {
    //...
}
```

atomic references

```scala
import cats.effect.IO

class AtomicQueueVer1[A]:
    private val ref = new AtomicReference(immutable.Queue.empty[A])

    def push(event: A): IO[Unit] =
        IO.defer:
            val current = ref.get
            val update = current.enqueue(event)
            if ref.compareAndSet(current, update) then
                IO.unit
            else // RETRY!
                push(event)
```

```scala
import cats.effect.IO

class AtomicQueueVer1[A]:
    private val ref = new AtomicReference(immutable.Queue.empty[A])

    def push(event: A): IO[Unit] =
        IO.defer:
            val current = ref.get
            val update = current.enqueue(event)
            if ref.compareAndSet(current, update) then
                IO.unit
            else // RETRY!
                push(event)
```

```scala
import cats.effect.IO

class AtomicQueueVer1[A]:
    private val ref = new AtomicReference(immutable.Queue.empty[A])

    def push(event: A): IO[Unit] =
        IO.defer:
            val current = ref.get
            val update = current.enqueue(event)
            if ref.compareAndSet(current, update) then
                IO.unit
            else // RETRY!
                push(event)
```

```scala
import cats.effect.IO

class AtomicQueueVer1[A]:
    private val ref = new AtomicReference(immutable.Queue.empty[A])

    def push(event: A): IO[Unit] =
        IO.defer:
            val current = ref.get
            val update = current.enqueue(event)
            if ref.compareAndSet(current, update) then
                IO.unit
            else // RETRY!
                push(event)
```

```
f: State => (State, Return)
```

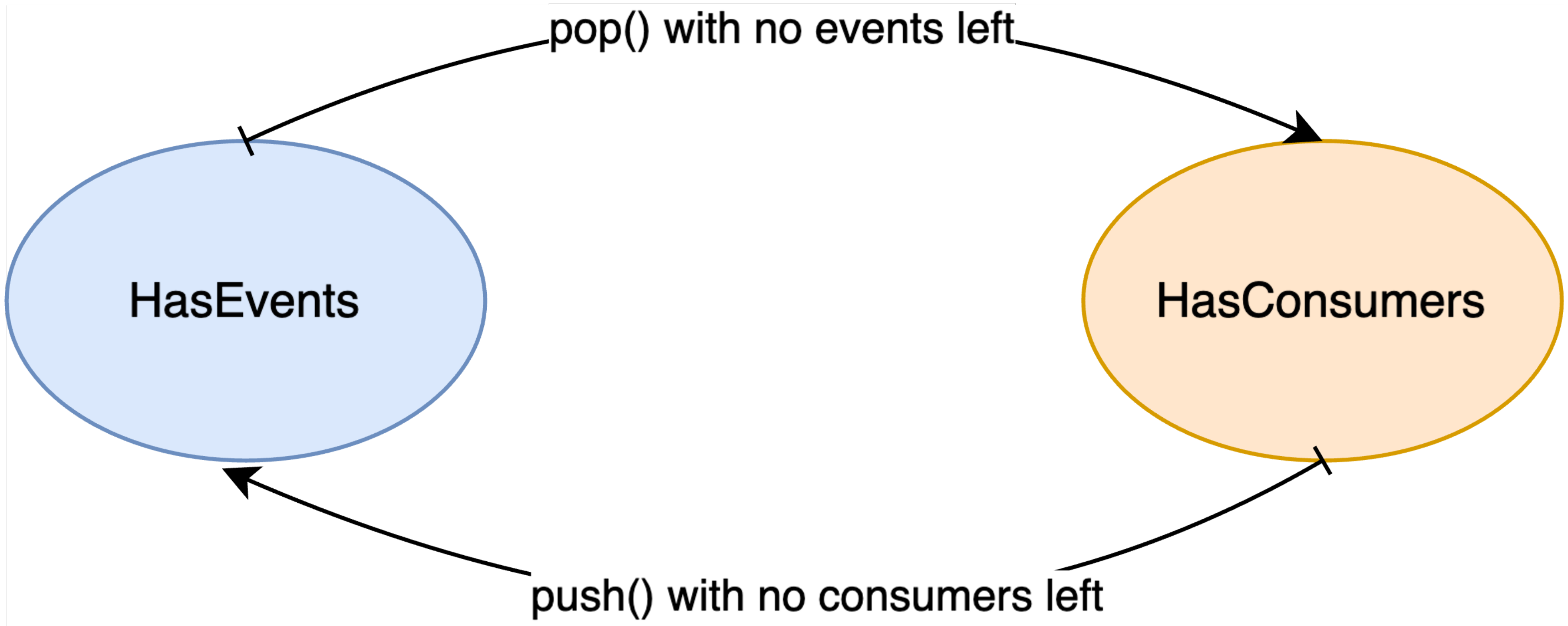
```scala
extension [State](self: AtomicReference[State])
  def modify[Return](f: State => (State, Return)): IO[Return] =
    IO.defer:
      val current = self.get
      val (updated, returnValue) = f(current)
      if (self.compareAndSet(current, updated))
        IO.pure(returnValue)
      else // RETRY!
        modify(f)
```

```scala
class AtomicQueueVer1[A]:
    private val ref = new AtomicReference(immutable.Queue.empty[A])

    def push(event: A): IO[Unit] =
        ref.modify: queue =>
            (queue.enqueue(event), ())
```

```scala
class AtomicQueueVer1[A]:
  private val ref = new AtomicReference(immutable.Queue.empty[A])

  def pop: IO[A] =
    ref.modify: queue =>
      queue.dequeueOption match
      case Some((event, rest)) =>
        (rest, IO.pure(event))
      case None =>
        val retry = IO.sleep(500.millis) *> pop
        (queue, retry)
    .flatten
```

Interruptible

State Machines, FTW!


pop() with no events left
HasEvents
HasConsumers
push() with no consumers left

```scala
import scala.collection.immutable.Queue as ScalaQueue

type Callback[-A] = Either[Throwable, A] => Unit

enum State[A]:
  case HasEvents(
    events: ScalaQueue[A]
  )
  case HasConsumers(
    callbacks: ScalaQueue[Callback[A]]
  )
```

```
f: State => (State, IO[Return])
```

```scala
enum State[A]:
  //...
  def pushEvent(event: A): (State[A], IO[Unit]) =
    this match
    case HasEvents(events) =>
      (HasEvents(events.enqueue(event)), IO.unit)
    case HasConsumers(consumers) =>
      consumers.dequeueOption match
      case None =>
        (HasEvents(ScalaQueue(event)), IO.unit)
      case Some((callback, rest)) =>
        (HasConsumers(rest), IO(callback(Right(event))))
```

```scala
enum State[A]:
    //...
    def popEvent(cb: Callback[A]): (State[A], IO[Unit]) =
        this match
        case HasEvents(events) =>
            events.dequeueOption match
            case None =>
                (HasConsumers(ScalaQueue(cb)), IO.unit)
            case Some((event, rest)) =>
                (HasEvents(rest), IO(cb(Right(event))))
        case HasConsumers(callbacks) =>
            (HasConsumers(callbacks.enqueue(cb)), IO.unit)
```

```
enum State[A]:
  //...
  def cancelPopAwait(cb: Callback[A]): State[A] =
    this match
    case ref: HasEvents[A] => ref
    case HasConsumers(callbacks) =>
      HasConsumers(callbacks.filterNot(_ == cb))
```

```scala
import cats.effect.*

class RefQueue[A](ref: Ref[IO, State[A]]):

    def push(event: A): IO[Unit] =
        ref.flatModify(_.pushEvent(event))

    def pop: IO[A] =
        IO.async: cb =>
            for
                _ <- ref.flatModify(_.popEvent(cb))
            yield Some(
                // Execute this in case of interruption
                IO(ref.update(_.cancelPopAwait(cb)))
            )
```

PROS:

- pure functions, immutable data structures
- cheap, often easy to implement
- async-friendly
- lock free

CONS:

- can have poor performance, sometimes

Plot Twist

Synchronization

is

Ordering!

```scala
class Queue[A]:
    var hasEvent = false
    var event: A | Null = null

    def push(a: A): Unit = ???

    def pop(): A = ???
```

```scala
class Queue[A]:
  var hasEvent = false
  var event: A | Null = null

  def push(a: A): Unit =
    while hasEvent do
      Thread.onSpinWait()
    event = a
    hasEvent = true
```

```scala
class Queue[A]:
    var hasEvent = false
    var event: A | Null = null

    def pop(): A =
        while !hasEvent do
            Thread.onSpinWait()
        val ret = event.asInstanceOf[A]
        hasEvent = false
        ret
```

```
class Queue[A]:
    var hasEvent = false
    var event: A | Null = null

    def pop(): A =
        while !hasEvent do
            Thread.onSpinWait()
        hasEvent = false                    // RE-ORDERED!!!
        val ret = event.asInstanceOf[A]
        ret
```

```scala
class Queue[A]:
  var hasEvent = false
  var event: A | Null = null

  def push(a: A): Unit =
    while hasEvent do
      Thread.onSpinWait()
    hasEvent = true          // RE-ORDERED!
    event = a
```

```
// ...
// producer thread
queue.push("value")

// consumer thread
println(queue.hasEvent, queue.event)

//=> (false, null)
//=> (false, "value")
//=> (true, "value")
//=> (true, null)
```

@volatile

```scala
class SpScQueue[A]:
  @volatile var hasEvent = false
  var event: A | Null = null

  def push(a: A): Unit =
    while hasEvent do
      Thread.onSpinWait()
    event = a
    hasEvent = true
```

```scala
class SpScQueue[A]:
    @volatile var hasEvent = false
    var event: A | Null = null

    def pop(): A =
        while !hasEvent do
            Thread.onSpinWait()
        val ret = event.asInstanceOf[A]
        hasEvent = false
        ret
```

Reading from a volatile	Acquiring a lock
Writing to a volatile	Releasing a lock

- acquisition and release needs to happen on the same monitor
- volatile reading and writing needs to happen on the same variable

- acquisition and release needs to happen in pair
- reading and writing from a volatile needs to happen in pair

- atomic references and monitor locks have the same visibility guarantees, due to volatile semantics

Takeaways

- State machines, FTW
- Synchronization is ordering
- Meet concurrency issues with the confidence of Zuzi