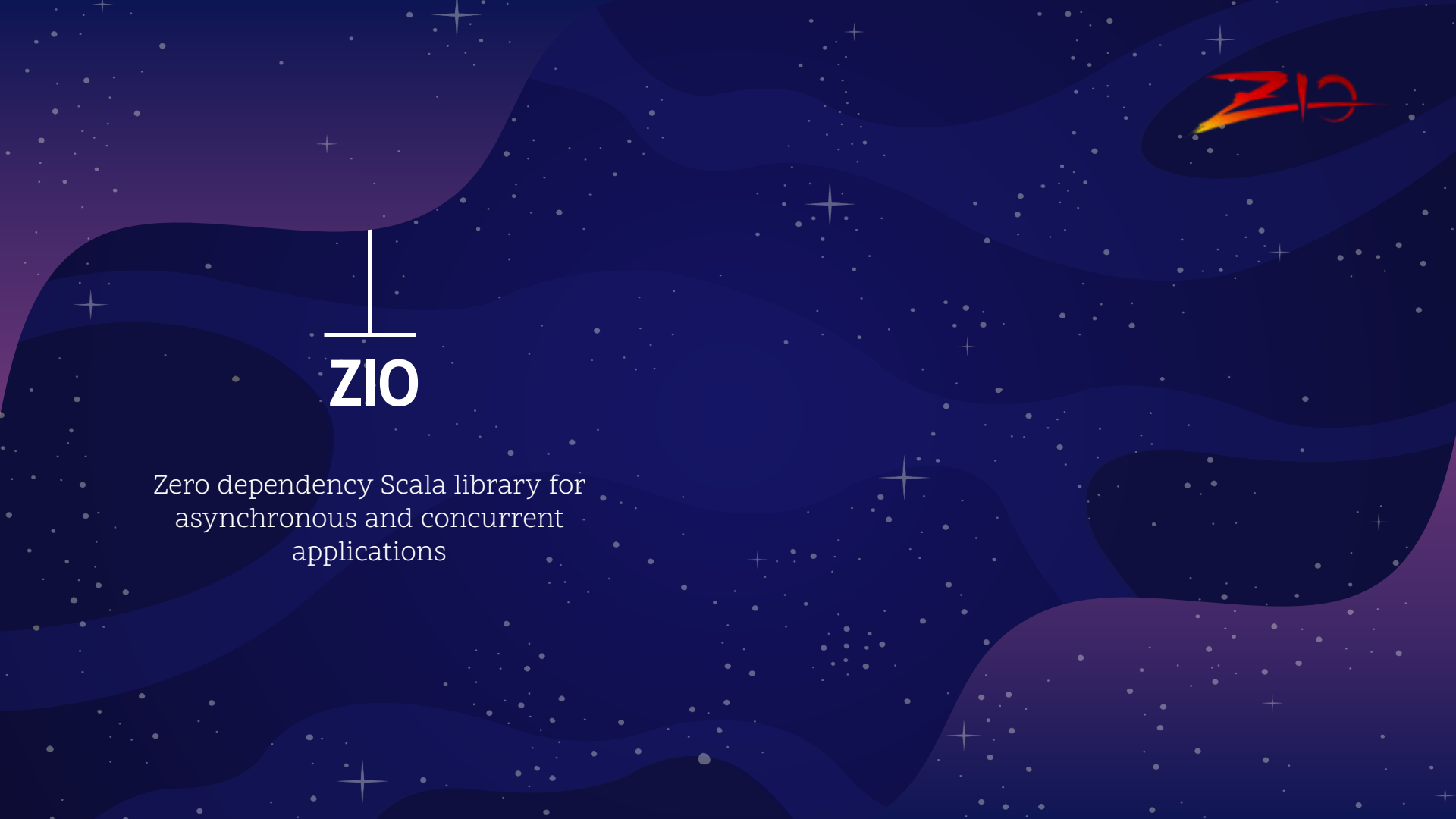
Fiber Supervision in ZIO

Adam Fraser & Wiem Zine Elabidine

@adamfraser

@wiemzin

Bringing Structured Concurrency to Scala
Using Fibers in ZIO


ZIO
Zero dependency Scala library for asynchronous and concurrent applications

ZIO[R, E, A]

Models effects

ZIO[R, E, A]

```
R => Either[E, A]
```

ZIO[R, E, A]

How to run an effect?

ZIO[R, E, A]

```
R => Either[E, A]
```

How to run an effect?

Fiber[E, A]

Models a running IO

How to run an effect?

```scala
object Main extends zio.App {
  def run(args: List[String]) =
    zio.fold(_ => 1, _ => 0)
}

zio.Runtime.default
   .unsafeRun(UIO(println("Hello ZIO!"))
```

What Is A Fiber ?

ZIO Fibers

Fibers are lightweight equivalents of threads:

- Can have hundreds of thousands of fibers at a time
- Semantically block but never block the underlying operating system thread
- Safely interruptible

ZIO Fibers

- An "in flight" computation
- Potentially executing concurrently with other fibers
- Will either succeed with an A or fail an E

```
sealed trait Fiber[+E, +A]
```

Forking Fibers

- Creates a new Fiber
- Fiber is a running computation so must be suspended in an effect
- Forking requires an environment R but forking itself cannot fail

```
sealed trait ZIO[-R, +E, +A] {
  def fork: ZIO[R, Nothing, Fiber[E, A]]
}
```

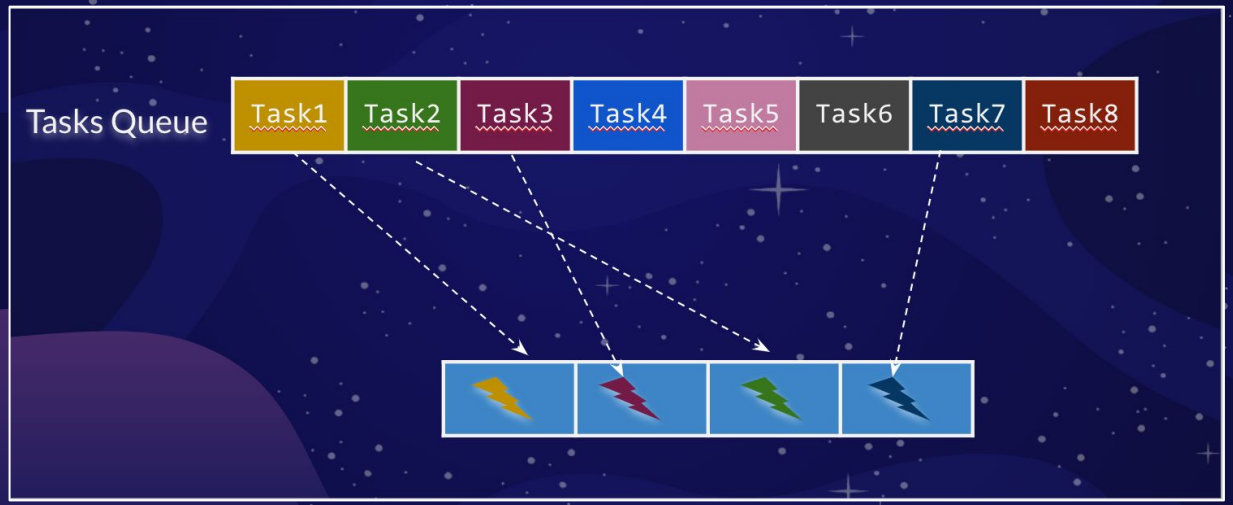
Implementation

- Every fiber executes a set of instructions in the ZIO "language"

```
val program: UIO[Fiber[Nothing, Unit]] =
  UIO(println("Hello world from a fiber."))
  .fork
```

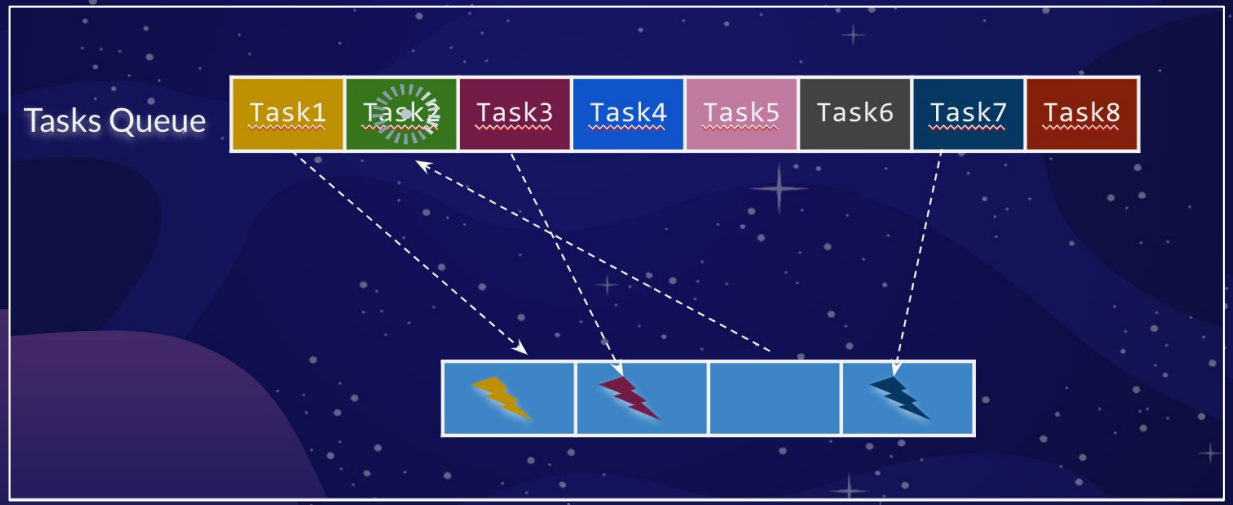
Implementation

- When a fiber has executed a specified number of instructions it yields to the ZIO runtime and enters a queue of running fibers



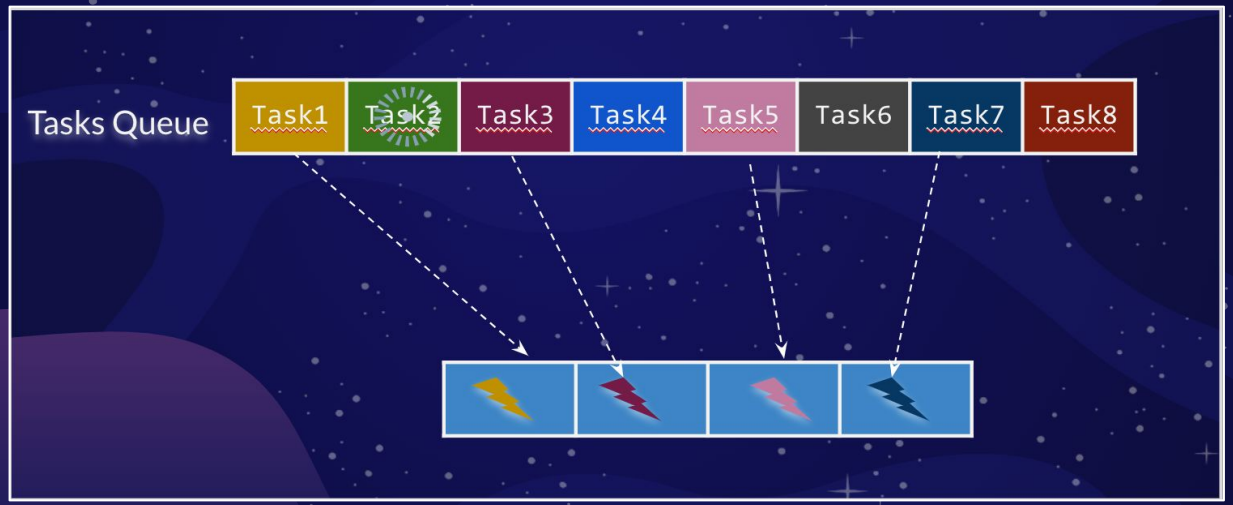
Implementation

- When a fiber has executed a specified number of instructions it yields to the ZIO runtime and enters a queue of running fibers



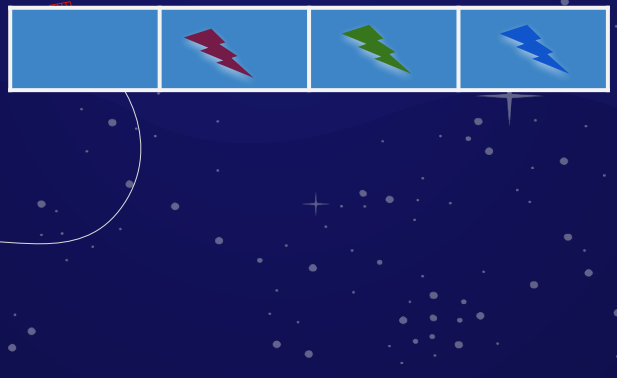
Implementation

- When a fiber has executed a specified number of instructions it yields to the ZIO runtime and enters a queue of running fibers



Implementation

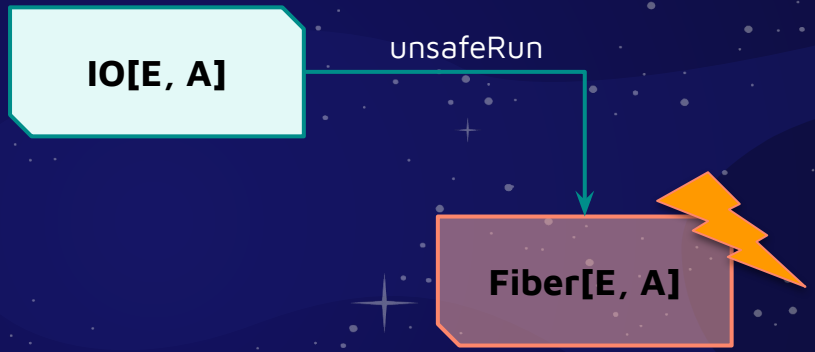
- The ZIO runtime continually takes fibers from the queue and executes them using underlying operating system threads.



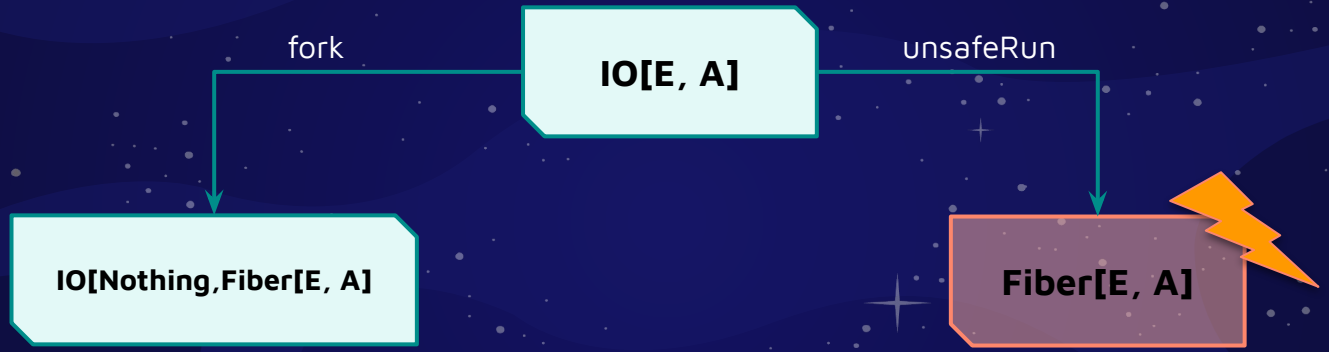
Summary

- Allows concurrency even in single threaded environments
- One thread may execute many fibers and one fiber may be executed by multiple threads during its life

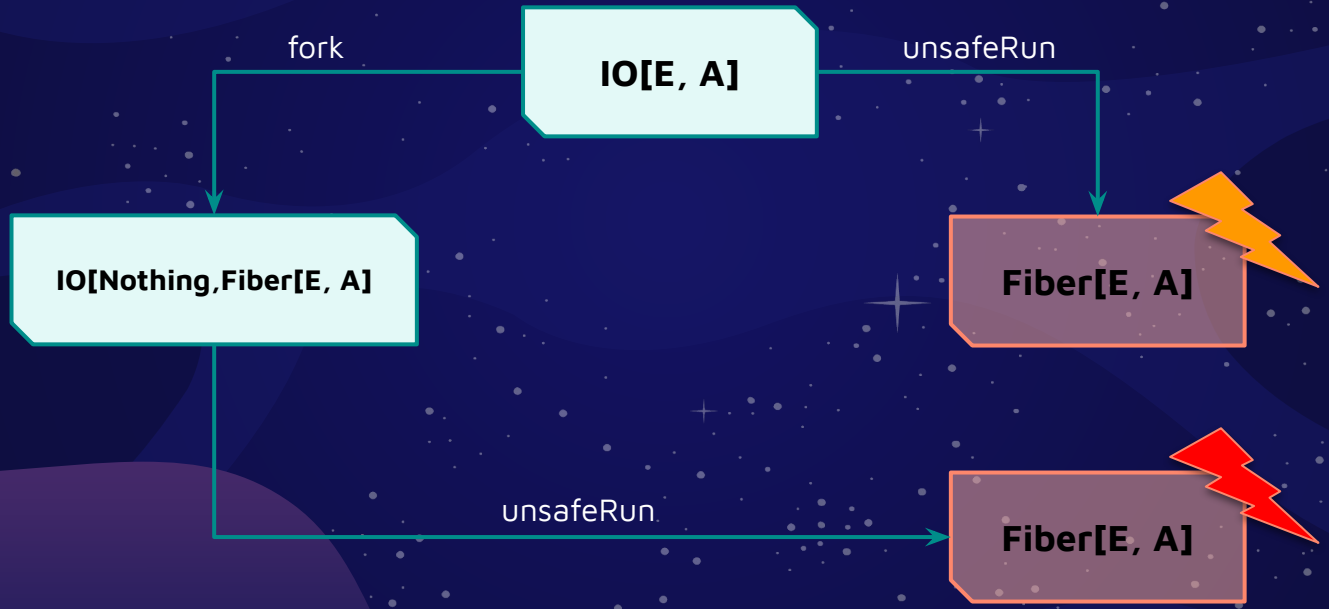
Forking Fibers



Forking Fibers



Forking Fibers



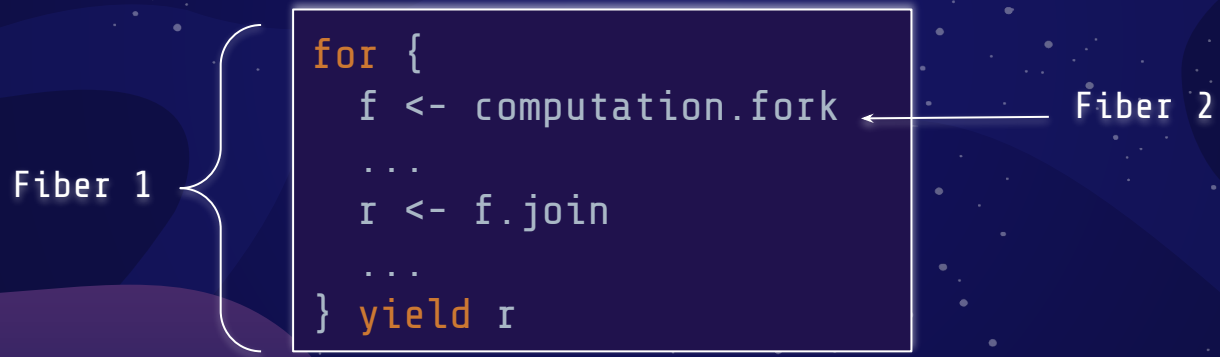
Joining Fibers

- Suspends until the fiber has terminated and returns its result
- If the fiber completed with a failure the result will be a failure
- If the fiber was interrupted joining will return immediately

```
sealed trait Fiber[+E, +A] {
  def join: ZIO[Any, E, A]
}
```

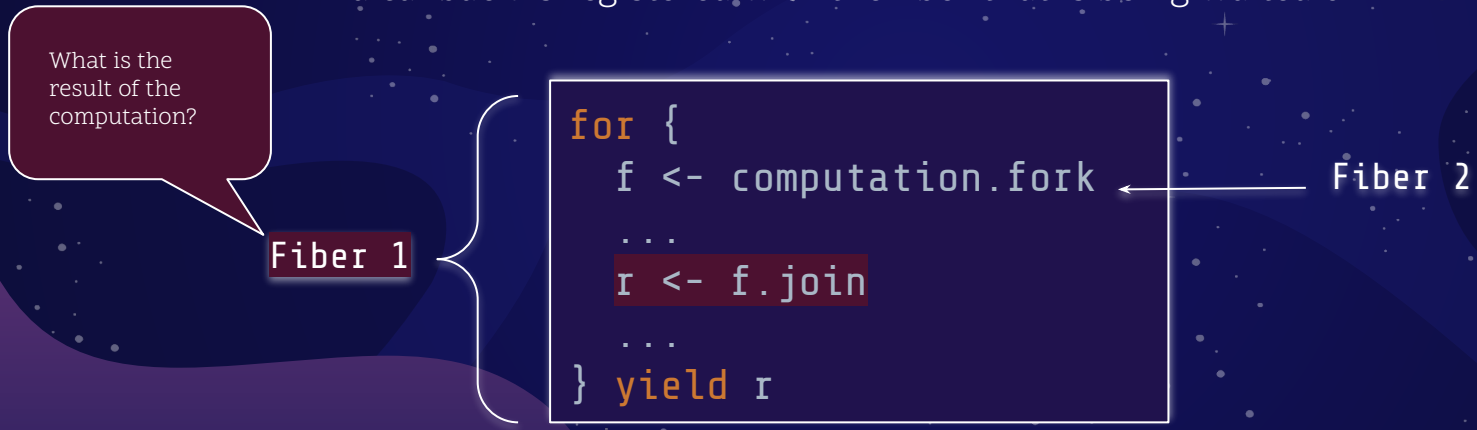
Implementation

- When one fiber "waits" on another fiber it enters a suspended state and a callback is registered with the fiber that is being waited on



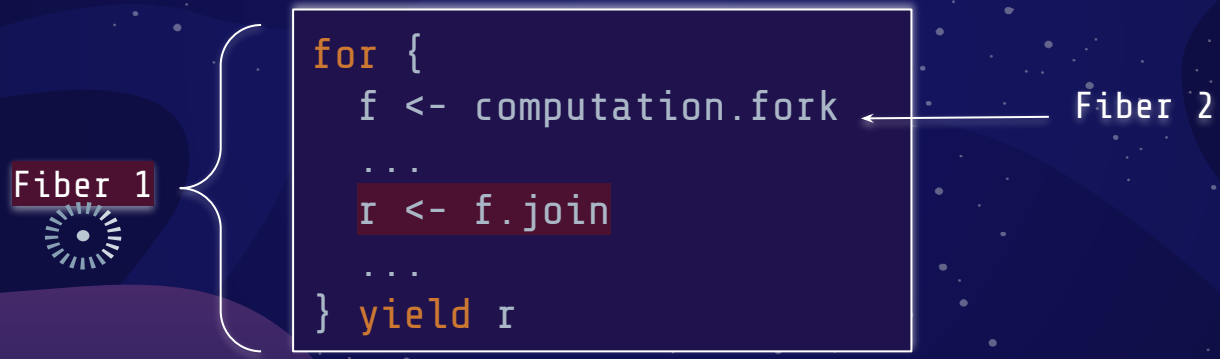
Implementation

- When one fiber "waits" on another fiber it enters a suspended state and a callback is registered with the fiber that is being waited on



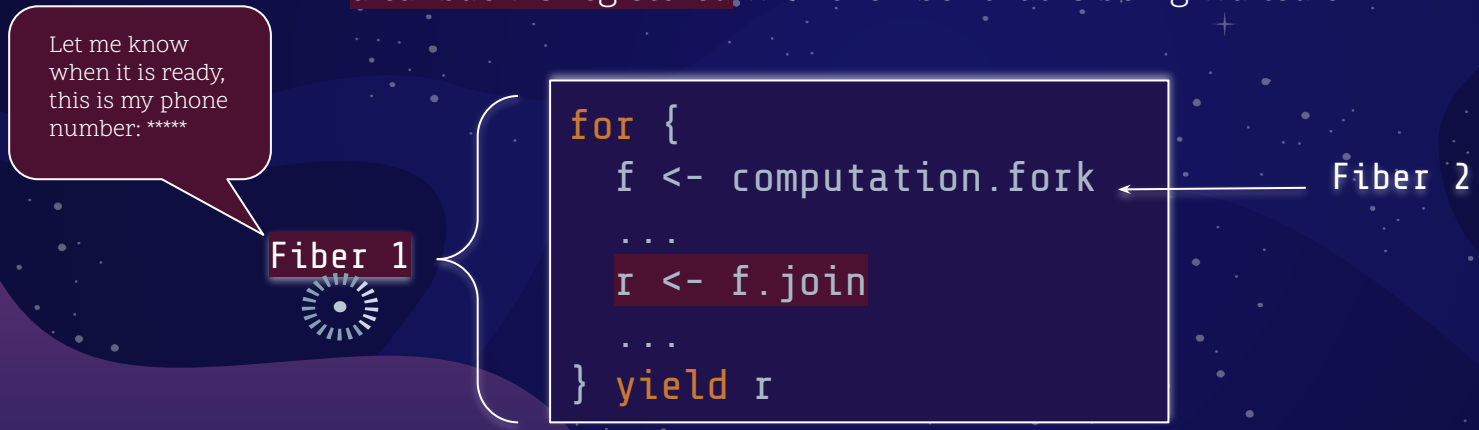
Implementation

- When one fiber "waits" on another fiber it enters a suspended state and a callback is registered with the fiber that is being waited on



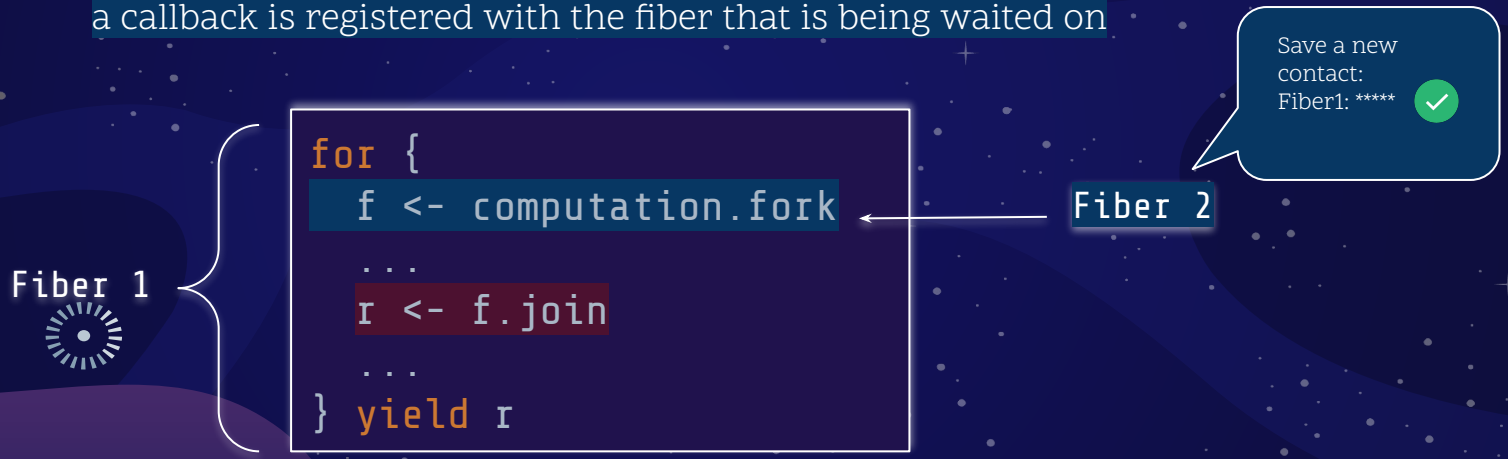
Implementation

- When one fiber "waits" on another fiber it enters a suspended state and a callback is registered with the fiber that is being waited on



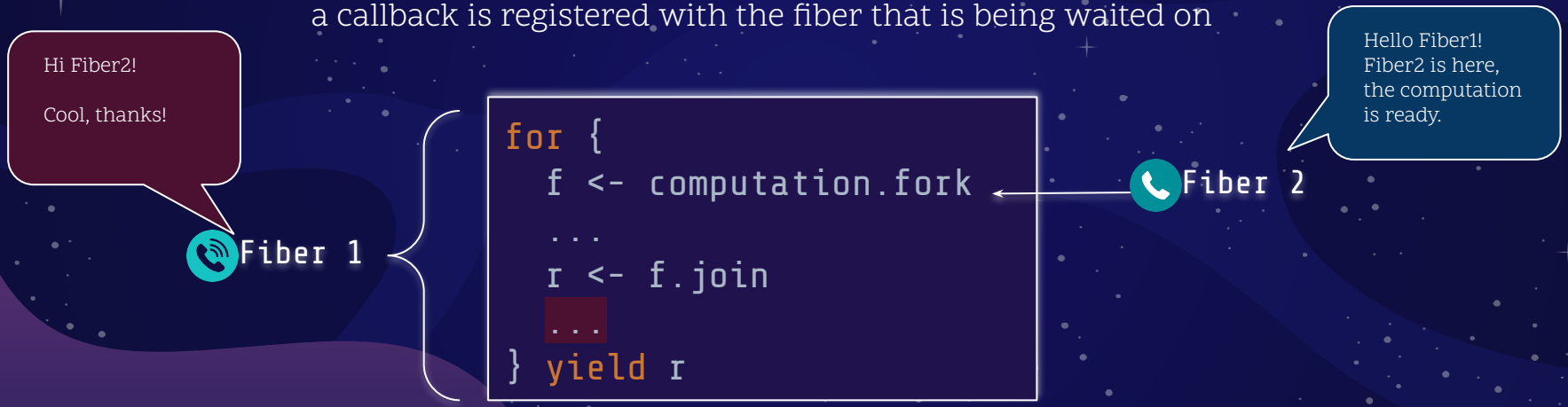
Implementation

- When one fiber "waits" on another fiber it enters a suspended state and a callback is registered with the fiber that is being waited on



Implementation

- When one fiber "waits" on another fiber it enters a suspended state and a callback is registered with the fiber that is being waited on



Summary

- Allows waiting on completion of a fiber without ever blocking operating system threads

```
for {
  f <- computation.fork
  ...
  r <- f.join
  ...
} yield r
```

Interrupting Fibers

- Attempts to immediately stop execution of further computations by this fiber
- Suspends until the fiber has been successfully interrupted
- Returns the result of the fiber (success, failure, or interruption)

```
sealed trait Fiber[+E, +A] {
  def interrupt: UIO[Exit[E, A]]
}
```

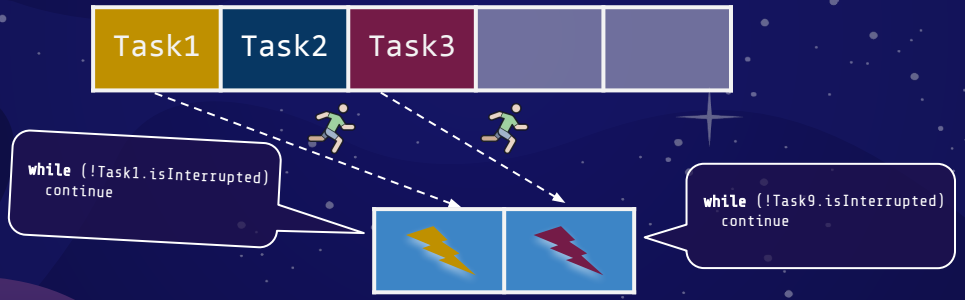
Finalizers

- Adds a finalizer to the specified effect
- Finalizer will be run even if the fiber running this effect is interrupted
- Critical for safe resource usage

```
trait ZIO[-R, +E, +A] {
  def ensuring[R1 <: R](finalizer: URIO[R1, Any]): ZIO[R1, E, A]
}
```

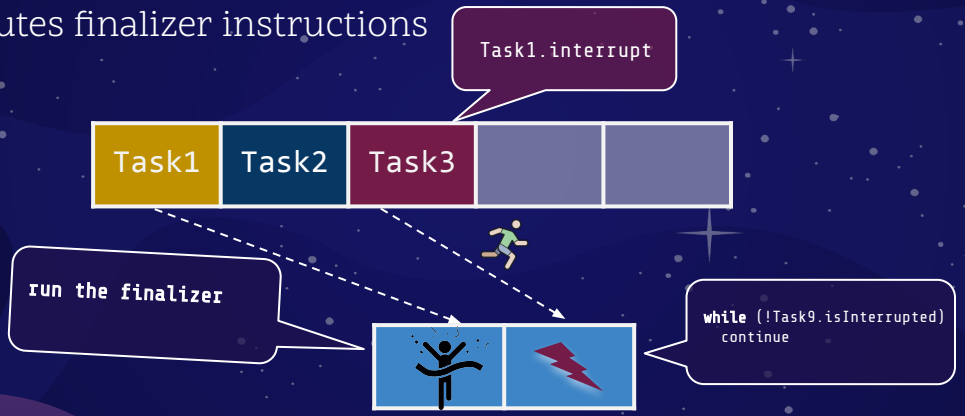
Implementation

Fibers check for interruption between executing each instruction.



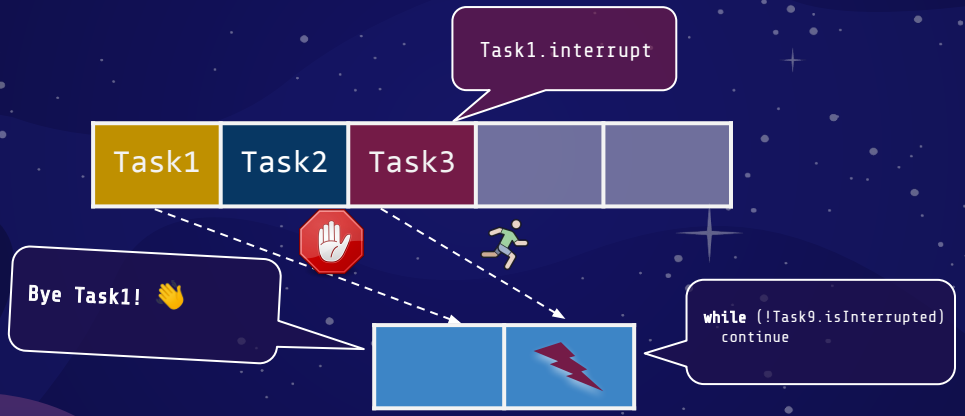
Implementation

- If a fiber is interrupted it skips executing normal instructions and only executes finalizer instructions



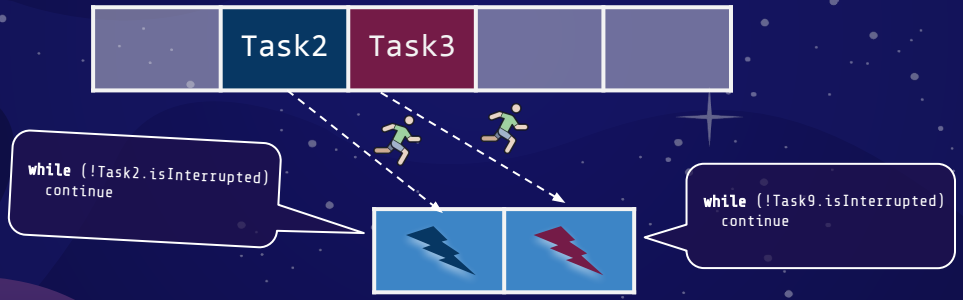
Implementation

- Fiber is then removed from active fibers.



Implementation

- No operating system threads are interrupted



Interruptibility

- Effects are interruptible by default but can be marked as uninterruptible
- For example, finalizers run uninterruptibly
- If a fiber is executing code that is uninterruptible it continues until it reaches a point when it is interruptible

```
trait ZIO[-R, +E, +A] {
  def interruptible: ZIO[R, E, A]
  def uninterruptible: ZIO[R, E, A]
}
```

Implementation

- Each fiber contains a variable indicating whether it is currently interruptible.
- Special instructions in the ZIO language turn interruptibility off and on.
- When checking for interruption, only stop executing instructions if interrupted and interruptible.

What Is Fiber Supervision ?

Fiber supervision

Fiber supervision describes how supervised fibers will be handled when the fiber that supervises them terminates:

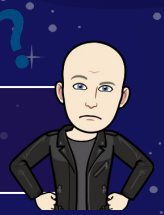
```
def john(topics: List[Topic]) =
  wiem(topics).fork
```

```
def wiem(topics: List[Topic]) = {
  val (topics1, topics2) =
    topics.partition(_.priority == HIGH)
  for {
    adam <- research(topics2).fork
    result <- research(topics1)
    ...
  } yield result
}
```

Fiber supervision

Fiber supervision describes how supervised fibers will be handled when the fiber that supervises them terminates:

> On success



```
def john(topics: List[Topic]) =
  wiem(topics).fork
```

```
def wiem(topics: List[Topic]) = {
  val (topics1, topics2) =
    topics.partition(_.priority == HIGH)
  for {
    adam <- research(topics2).fork
    result <- research(topics1)
    ...
  } yield result
}
```

Fiber supervision

Fiber supervision describes how supervised fibers will be handled when the fiber that supervises them terminates:

> On Failure

```
def john(topics: List[Topic]) =
  wiem(topics).fork
```


```
def wiem(topics: List[Topic]) = {
  val (topics1, topics2) =
    topics.partition(_.priority == HIGH)
  for {
    adam <- research(topics2).fork
    result <- research(topics1)
    ...
  } yield result
}
```

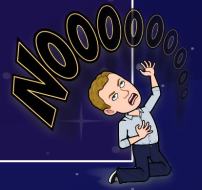
Fiber supervision

Fiber supervision describes how supervised fibers will be handled when the fiber that supervises them terminates:

> On interruption

```
def john(topics: List[Topic]) =
  wiem(topics).fork
```

```
def wiem(topics: List[Topic]) = {
  val (topics1, topics2) =
    topics.partition(_.priority == HIGH)
  for {
    adam <- research(topics2).fork
    result <- research(topics1)
    ...
  } yield result
}
```



Unsupervised Fibers

```
object CatsExample extends IOApp {
  val child: IO[Unit] =
    IO.sleep(5.seconds) *> IO(println("Hello from a child fiber!"))
  val parent: IO[Unit] =
    child.start *> IO.sleep(3.seconds) *> IO(println("Hello from a parent fiber!"))
  override def run(args: List[String]): IO[ExitCode] =
    for {
      fiber <- parent.start
      _     <- IO.sleep(1.second)
      _     <- fiber.cancel
      _     <- IO.sleep(10.seconds)
    } yield ExitCode(0)
 }
```

Problems

- **Resource Leak:** We tried to cancel this entire computation but child is still running, potentially doing expensive work and acquiring unnecessary resources!
- **Global Reasoning:** The fact that parent decided to fork child for efficiency has changed the meaning of our program!

Fork Join Identity

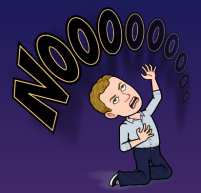
- Forking a fiber to perform a computation and then joining it back should be the same as just doing the computation.
- This seems obvious but it is going to be critically important.
- It means we can reason about our code locally. If I have some expensive computation I can refactor to fork some fibers to do parts of the work, join them all back, and my program will still work the same way.

```
effect.fork.flatMap(_.join) <-> effect
```

Supervised Fibers

```
object ZIOExample extends App {
 val child: ZIO[ZEnv, Nothing, Unit] =
   ZIO.sleep(5.seconds) *> putStrLn("Hello from a child fiber!")
 val parent: ZIO[ZEnv, Nothing, Unit] =
   child.fork.delay(3.seconds) *> putStrLn("Hello from a parent fiber!")
 override def run(args: List[String]): ZIO[ZEnv, Nothing, Int] =
   for {
     fiber <- parent.fork
     _     <- ZIO.sleep(1.second)
     _     <-.fiber.interrupt
     _     <- ZIO.sleep(10.seconds)
   } yield 0
}
```

Supervised Fibers



- `parent` "supervises" `child` when it forks it.
- On interruption or failure of `parent`, `child` is immediately interrupted.
- Ensures resource safety because no unnecessary work is done.
- Allows local reasoning because supervision is hierarchical.

Daemon Fibers

- If we don't want this supervision behavior we can disown the fiber, making it its own "root fiber".
- Will not be interrupted if previous parent is interrupted.
- Can continue indefinitely beyond the life of the previous parent fiber.
- Be careful with this, fiber supervision is generally there to help you!

```
trait ZIO[-R, +E, +A] {
  def forkDaemon: URIO[R, Fiber.Runtime[E, A]]
}
```

Normal Termination

- Interruption and failure of supervising fiber are the "easy case". Supervised fibers should clearly be interrupted when their supervisor fails or is interrupted.
- What if the supervisor completes normally before the supervised fiber has completed?
- Ideally to preserve fork join identity we would like to "transplant" the supervised fiber to the supervisor's supervisor.
- However, this is very challenging in practice because it requires synchronizing on the state of two different fibers.

Release Candidate 17

- On supervisor termination, supervised fibers are transplanted to supervisor's supervisor.
- Honors fork join identity.
- However, led to concurrency issues with transplanting fibers.
- Also created risk of accidentally leaking resources since fibers would always be transplanted when in some cases they should not outlive supervisor.

Release Candidate 18

- On supervisor termination, supervised fibers are immediately terminated.
- Very strong resource safety guarantees.
- However, violates fork join identity.
- Could lead to forked effects being terminated "too early" and encouraged use of `forkDaemon` which resulted in fibers being completely unsupervised.

Explicit Scopes

- Stepping back, problem is really one of scopes.
- Ideally, every fiber would be explicitly either joined or interrupted at some point.
- But we have no way to enforce this with the type system.
- We can do the next best thing through concept of explicit scopes.

Scopes

- Every fiber has a scope.
- By default a fiber's scope ends when it is terminated and it handles fibers it is supervising at that point.
- `scoped` creates a new scope and merges child scopes into the parent scope

```
object ZIO {
  def  scoped[R, E, A](zio: ZIO[R, E, A]): ZIO[R, E, A]
}
```

Implications

- Preserves fork join identity within the context of a scope.
- Maintains high degree of resource safety since the default scope is the life of the supervising fiber.
- Allows the caller to extend the scope by using scoped.

How Does It Help Us Write Safer Concurrent Code?

Welcome to Fiber Club!

Fiber Club


THE FIRST RULE OF FIBER CLUB
YOU DO NOT USE FIBERS DIRECTLY
imgflip.com

Common Concurrency Combinators in ZIO

```
trait ZIO[R, E, A]  {
  def race(that: ZIO[R, E, A]): ZIO[R, E, A]
  def raceAll(ios: Iterable[ZIO[R, E, A]]): ZIO[R, E, A]
  def zipPar(that: ZIO[R, E, B]): ZIO[R, E, (A, B)]
  def on(ec: ExecutionContext): ZIO[R, E, A]
  ...
}

object ZIO {
 def foreachPar(as: Iterable[A])(fn: A => ZIO[R, E, B]): ZIO[R, E, List[B]]
...
}
```

Racing effects

Racing effects

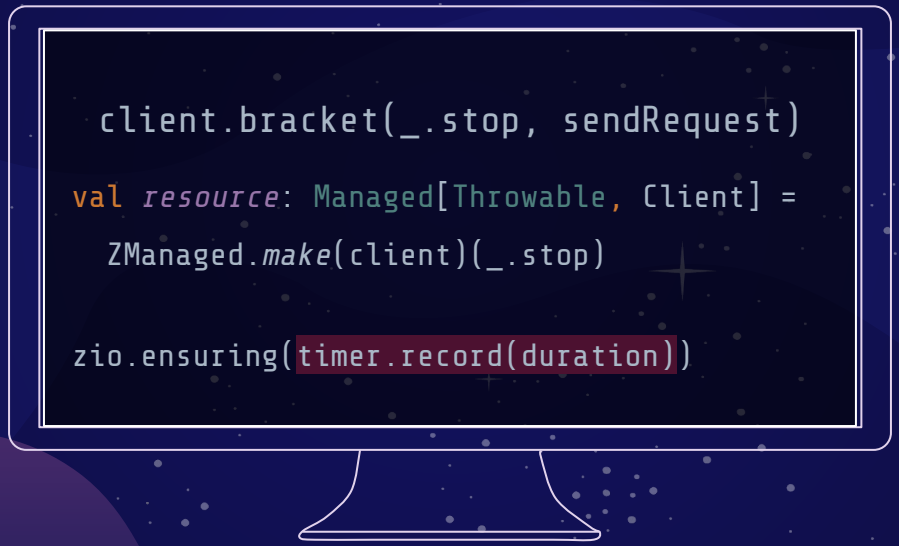
Parallel computation

Parallel computation

Using Resources

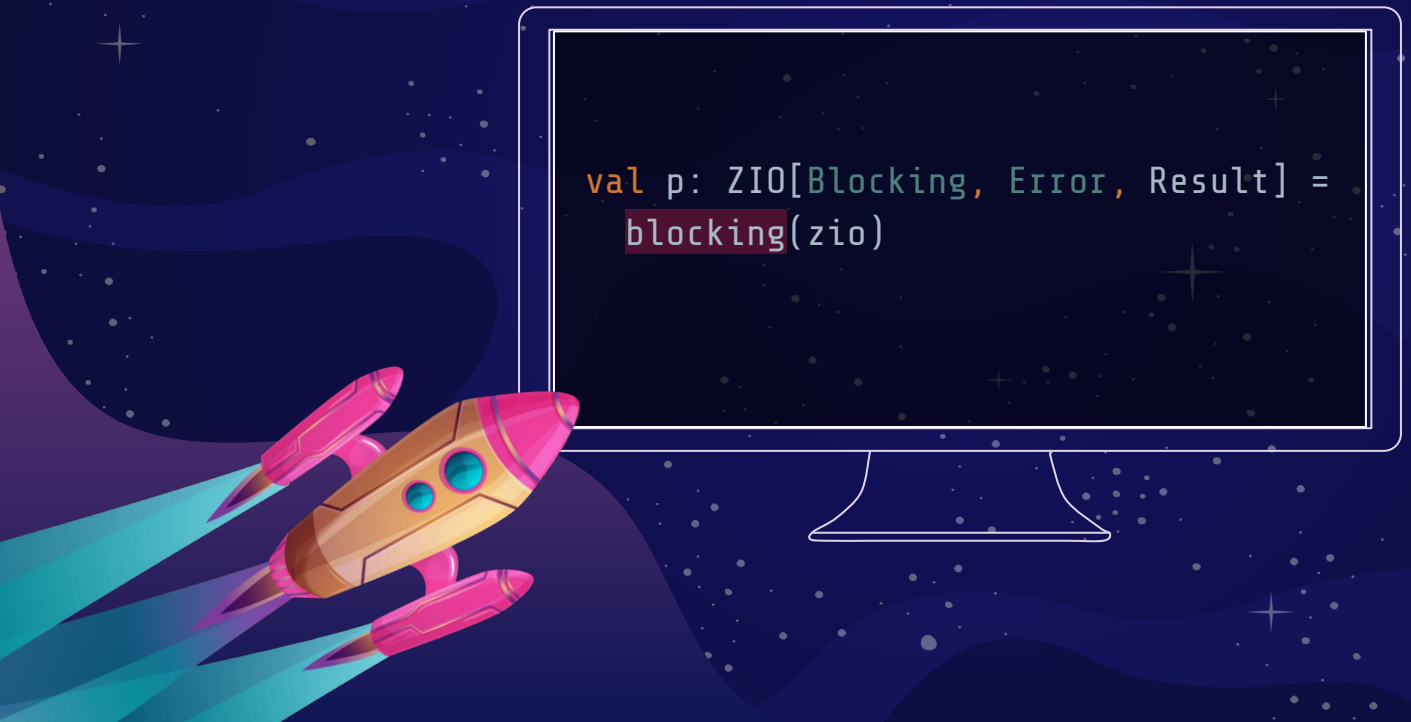
Using Resources

Using Resources



Controlling thread pools

Controlling thread pools



Summary

- Fibers are the building blocks for writing safe, high performance concurrent code
- Fiber supervision helps us do the “right thing” by default
- Variety of combinators and data types build on top of fibers for most use cases
- Fibers give you the ability to implement your own combinators when you need to

zio documentation

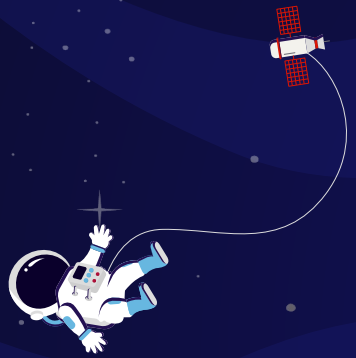
zio.dev

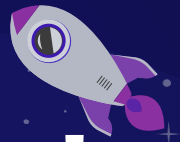
zio project

https://github.com/zio/zio

discord

https://discord.com/channels/629491597070827530/630498701860929559





Thanks!

Does anyone have any questions?

@adamfraser @wiemzin

Credits

This is where you give credit to the ones who are part of this project.
Did you like the resources on this template? Get them for free at our other websites.

- Presentation template by Slidesgo
- Icons by Flaticon
- Images & infographics by Freepik
- Author introduction slide photo created by Freepik
- Text & Image slide photo created by Freepik.com
- Big image slide photo created by Freepik.com


slidesgo