Async Rust:
Portability and Interoperability

Nick Cameron, Microsoft
Rust Linz, April 2022

What is the Async WG?

What is async Rust?

```
async fn read_file(file_name: &Path) -> Result<String> {
  let mut f = File::open(file_name).await?;
  let mut result = String::new();
  f.read_to_string(&mut result).await?;
  Ok(result)
}

let contents = read_file("foo.txt".into()).await?;
```

```
async fn read_file(file_name: &Path) -> Result<String> {
  let mut f = File::open(file_name).await?;
  let mut result = String::new();
  f.read_to_string(&mut result).await?;
  Ok(result)
}

let contents = read_file("foo.txt".into()).await?;
```

```
async fn read_file(file_name: &Path) -> Result<String> {
  let mut f = File::open(file_name).await?;
  let mut result = String::new();
  f.read_to_string(&mut result).await?;
  Ok(result)
}

let contents = read_file("foo.txt".into()).await?;
```

```
async fn read_file(file_name: &Path) -> Result<String> {
  let mut f = File::open(file_name).await?;
  let mut result = String::new();
  f.read_to_string(&mut result).await?;
  Ok(result)
}

let contents = read_file("foo.txt".into()).await?;
```

```
async fn read_file(file_name: &Path) -> Result<String> {
  let mut f = File::open(file_name).await?;
  let mut result = String::new();
  f.read_to_string(&mut result).await?;
  Ok(result)
}

let contents = read_file("foo.txt".into()).await?;
```

```
async fn read_file(file_name: &Path) -> Result<String> {
  let mut f = File::open(file_name).await?;
  let mut result = String::new();
  f.read_to_string(&mut result).await?;
  Ok(result)
}

let contents = read_file("foo.txt".into()).await?;
```

```
async fn read_file(file_name: &Path) -> Result<String> {
  let mut f = File::open(file_name).await?;
  let mut result = String::new();
  f.read_to_string(&mut result).await?;
  Ok(result)
}

fn main() -> Result<()> {
  let contents = read_file("foo.txt".into()).await?;
                                             ^^^^^
}
```

```
error[E0728]: `await` is only allowed inside `async` functions and blocks
```

The Async Ecosystem

Rust currently provides only the bare essentials for writing async code. Importantly, executors, tasks, reactors, combinators, and low-level I/O futures and traits are not yet provided in the standard library. In the meantime, community-provided async ecosystems fill in these gaps.

The Async Foundations Team is interested in extending examples in the Async Book to cover multiple runtimes. If you're interested in contributing to this project, please reach out to us on Zulip.

Async Runtimes

Async runtimes are libraries used for executing async applications. Runtimes usually bundle together a *reactor* with one or more *executors*. Reactors provide subscription mechanisms for external events, like async I/O, interprocess communication, and timers. In an async runtime, subscribers are typically futures representing low-level I/O operations. Executors handle the scheduling and execution of tasks. They keep track of running and suspended tasks, poll futures to completion, and wake tasks when they can make progress. The word "executor" is frequently used interchangeably with "runtime". Here, we use the word "ecosystem" to describe a runtime bundled with compatible traits and features.

Community-Provided Async Crates

The Futures Crate

The `futures` crate contains traits and functions useful for writing async code. This includes the `Stream`, `Sink`, `AsyncRead`, and `AsyncWrite` traits, and utilities such as combinators. These utilities and traits may eventually become part of the standard library.

`futures` has its own executor, but not its own reactor, so it does not support execution of async I/O or timer futures. For this reason, it's not considered a full runtime. A common choice is to use utilities from `futures` with an executor from another crate.

Popular Async Runtimes

There is no asynchronous runtime in the standard library, and none are officially recommended. The following crates provide popular runtimes.

- Tokio: A popular async ecosystem with HTTP, gRPC, and tracing frameworks.
- async-std: A crate that provides asynchronous counterparts to standard library components.
- smol: A small, simplified async runtime. Provides the `Async` trait that can be used to wrap structs like `UnixStream` or `TcpListener`.
- fuchsia-async: An executor for use in the Fuchsia OS.

Rust currently provides only the bare essentials for writing async code. Importantly, executors, tasks, reactors, combinators, and low-level I/O futures and traits are not yet provided in the standard library. In the meantime, community-provided async ecosystems fill in these gaps.

The Async Foundations Team is interested in extending examples in the Async Book to cover multiple runtimes. If you're interested in contributing to this project, please reach out to us on Zulip.

Async Runtimes

Async runtimes are libraries used for executing async applications. Runtimes usually bundle together a *reactor* with one or more *executors*. Reactors provide subscription mechanisms for external events, like async I/O, interprocess communication, and timers. In an async runtime, subscribers are typically futures representing low-level I/O operations. Executors handle the scheduling and execution of tasks. They keep track of running and suspended tasks, poll futures to completion, and wake tasks when they can make progress. The word "executor" is frequently used interchangeably with "runtime". Here, we use the word "ecosystem" to describe a runtime bundled with compatible traits and features.

Community-Provided Async Crates

The Futures Crate

The `futures` crate contains traits and functions useful for writing async code. This includes the `Stream`, `Sink`, `AsyncRead`, and `AsyncWrite` traits, and utilities such as combinators. These utilities and traits may eventually become part of the standard library.

`futures` has its own executor, but not its own reactor, so it does not support execution of async I/O or timer futures. For this

Rust currently provides only the bare essentials for writing async code. Importantly, executors, tasks, reactors, combinators, and low-level I/O futures and traits are not yet provided in the standard library. In the meantime, community-provided async ecosystems fill in these gaps.

The Async Foundations Team is interested in extending examples in the Async Book to cover multiple runtimes. If you're interested in contributing to this project, please reach out to us on Zulip.

Async Runtimes

Async runtimes are libraries used for executing async applications. Runtimes usually bundle together a *reactor* with one or more *executors*. Reactors provide subscription mechanisms for external events, like async I/O, interprocess communication, and timers. In an async runtime, subscribers are typically futures representing low-level I/O operations. Executors handle the scheduling and execution of tasks. They keep track of running and suspended tasks, poll futures to completion, and wake tasks when they can make progress. The word "executor" is frequently used interchangeably with "runtime". Here, we use the word "ecosystem" to describe a runtime bundled with compatible traits and features.

Community-Provided Async Crates

The Futures Crate

The `futures` crate contains traits and functions useful for writing async code. This includes the `Stream`, `Sink`, `AsyncRead`, and `AsyncWrite` traits, and utilities such as combinators. These utilities and traits may eventually become part of the standard library.

`futures` has its own executor, but not its own reactor, so it does not support execution of async I/O or timer futures. For this

Async Runtimes

Async runtimes are libraries used for executing async applications. Runtimes usually bundle together a *reactor* with one or more *executors*. Reactors provide subscription mechanisms for external events, like async I/O, interprocess communication, and timers. In an async runtime, subscribers are typically futures representing low-level I/O operations. Executors handle the scheduling and execution of tasks. They keep track of running and suspended tasks, poll futures to completion, and wake tasks when they can make progress. The word "executor" is frequently used interchangeably with "runtime". Here, we use the word "ecosystem" to describe a runtime bundled with compatible traits and features.

Community-Provided Async Crates

The Futures Crate

The `futures` crate contains traits and functions useful for writing async code. This includes the `Stream`, `Sink`, `AsyncRead`, and `AsyncWrite` traits, and utilities such as combinators. These utilities and traits may eventually become part of the standard library.

`futures` has its own executor, but not its own reactor, so it does not support execution of async I/O or timer futures. For this reason, it's not considered a full runtime. A common choice is to use utilities from `futures` with an executor from another crate.

Popular Async Runtimes

There is no asynchronous runtime in the standard library, and none are officially recommended. The following crates provide popular runtimes.

- Tokio: A popular async ecosystem with HTTP, gRPC, and tracing frameworks.

Async Runtimes

Async runtimes are libraries used for executing async applications. Runtimes usually bundle together a *reactor* with one or more *executors*. Reactors provide subscription mechanisms for external events, like async I/O, interprocess communication, and timers. In an async runtime, subscribers are typically futures representing low-level I/O operations. Executors handle the scheduling and execution of tasks. They keep track of running and suspended tasks, poll futures to completion, and wake tasks when they can make progress. The word "executor" is frequently used interchangeably with "runtime". Here, we use the word "ecosystem" to describe a runtime bundled with compatible traits and features.

Community-Provided Async Crates

The Futures Crate

The `futures` crate contains traits and functions useful for writing async code. This includes the `Stream`, `Sink`, `AsyncRead`, and `AsyncWrite` traits, and utilities such as combinators. These utilities and traits may eventually become part of the standard library.

`futures` has its own executor, but not its own reactor, so it does not support execution of async I/O or timer futures. For this reason, it's not considered a full runtime. A common choice is to use utilities from `futures` with an executor from another crate.

Popular Async Runtimes

There is no asynchronous runtime in the standard library, and none are officially recommended. The following crates provide popular runtimes.

- Tokio: A popular async ecosystem with HTTP, gRPC, and tracing frameworks.

Community-Provided Async Crates

The Futures Crate

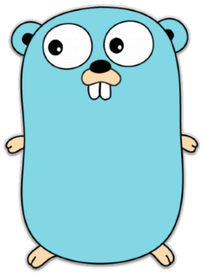
The `futures` crate contains traits and functions useful for writing async code. This includes the `Stream`, `Sink`, `AsyncRead`, and `AsyncWrite` traits, and utilities such as combinators. These utilities and traits may eventually become part of the standard library.

`futures` has its own executor, but not its own reactor, so it does not support execution of async I/O or timer futures. For this reason, it's not considered a full runtime. A common choice is to use utilities from `futures` with an executor from another crate.

Popular Async Runtimes

There is no asynchronous runtime in the standard library, and none are officially recommended. The following crates provide popular runtimes.

- Tokio: A popular async ecosystem with HTTP, gRPC, and tracing frameworks.
- async-std: A crate that provides asynchronous counterparts to standard library components.
- smol: A small, simplified async runtime. Provides the `Async` trait that can be used to wrap structs like `UnixStream` or `TcpListener`.
- fuchsia-async: An executor for use in the Fuchsia OS.



```
func main() {
  go readFile(“foo.txt”)
}
```

Runtimes

Runtimes

different contexts

=>

different constraints

=>

different runtimes

Runtimes

Executor

Scheduler

Reactor

IO traits

IO implementation

Timers and timeouts

Channels, locks, barriers, …

Utilities

Portability and Interoperability

Portability and Interoperability

Many runtimes, one ecosystem

Portability and Interoperability

Easier to switch runtime

Portability and Interoperability

Easy to start

Portability and Interoperability

async stuff in std

executor abstraction

std executor?

ease of use improvements

```
use std::io::async::Read;

async fn read_file(file_name: &Path) -> Result<String> {
  let mut f = File::open(file_name).await?;
  let mut result = String::new();
  f.read_to_string(&mut result).await?;
  Ok(result)
}

async fn main() -> Result<()> {
  let contents = read_file("foo.txt".into()).await?;
}
```

Async IO Traits

Async IO Traits

Read

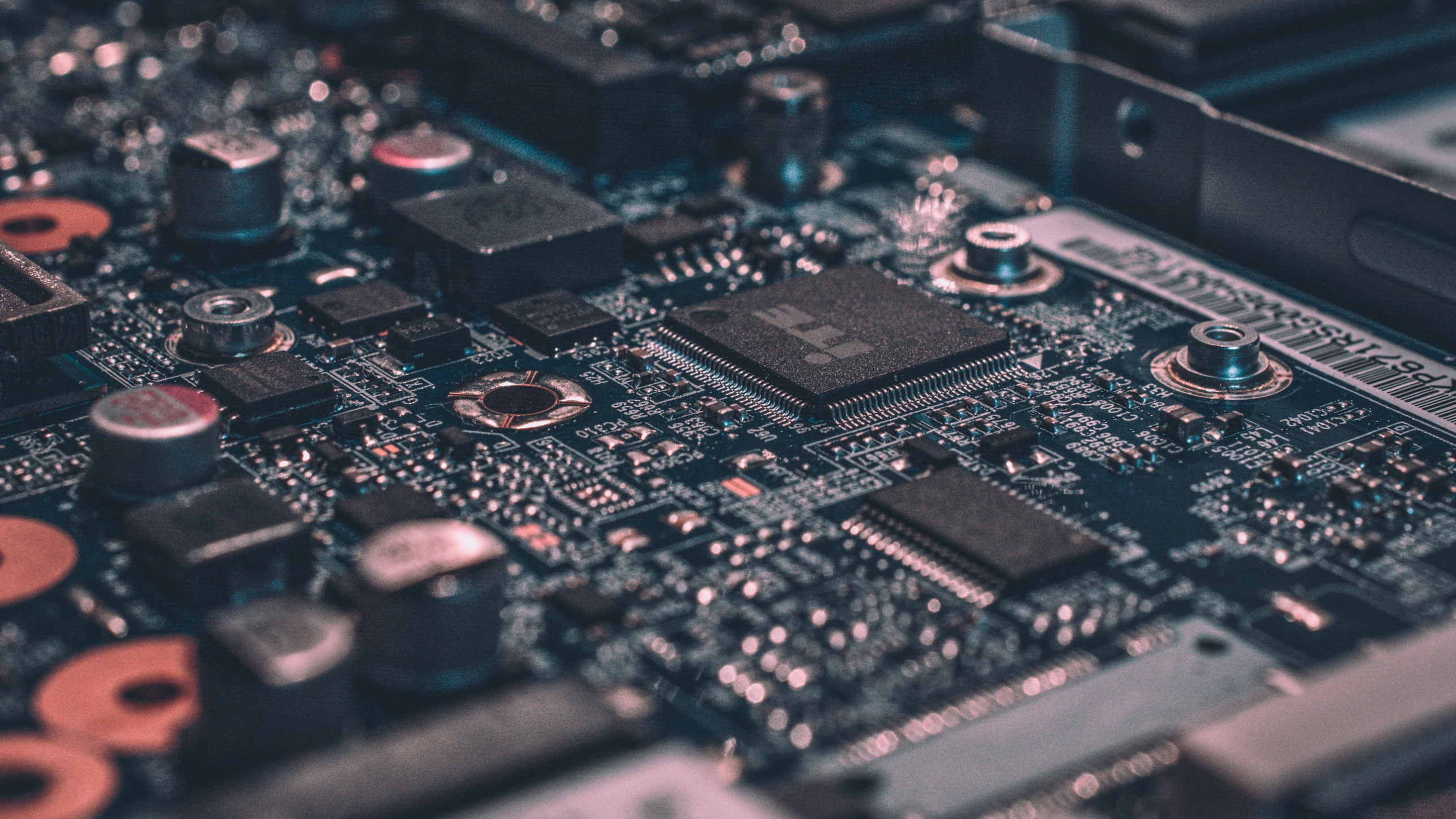
Write

BufRead

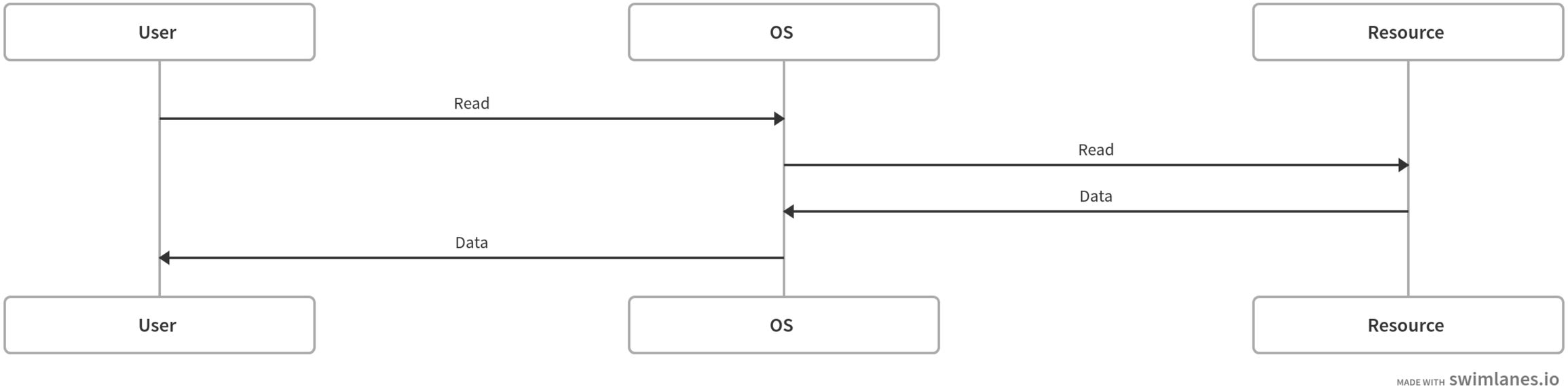
Seek

```
trait Read {
  fn read(&mut self, buf: &mut [u8]) -> Result<usize>;
}
```

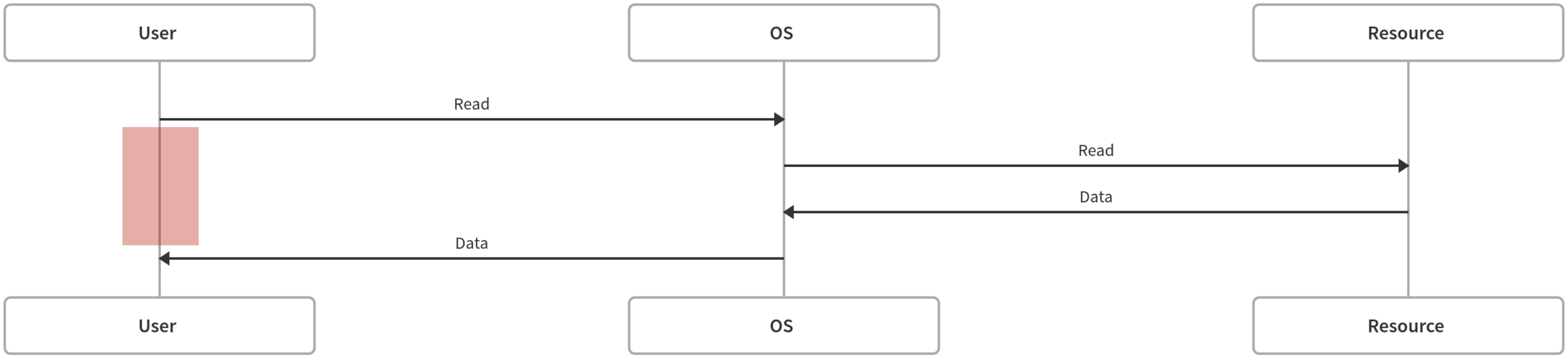
```
trait Read {
  async fn read(&mut self, buf: &mut [u8]) -> Result<usize>;
}
```



Sync


User
OS
Resource
Read
Read
Data
Data
User
OS
Resource
MADE WITH swimlanes.io

Sync

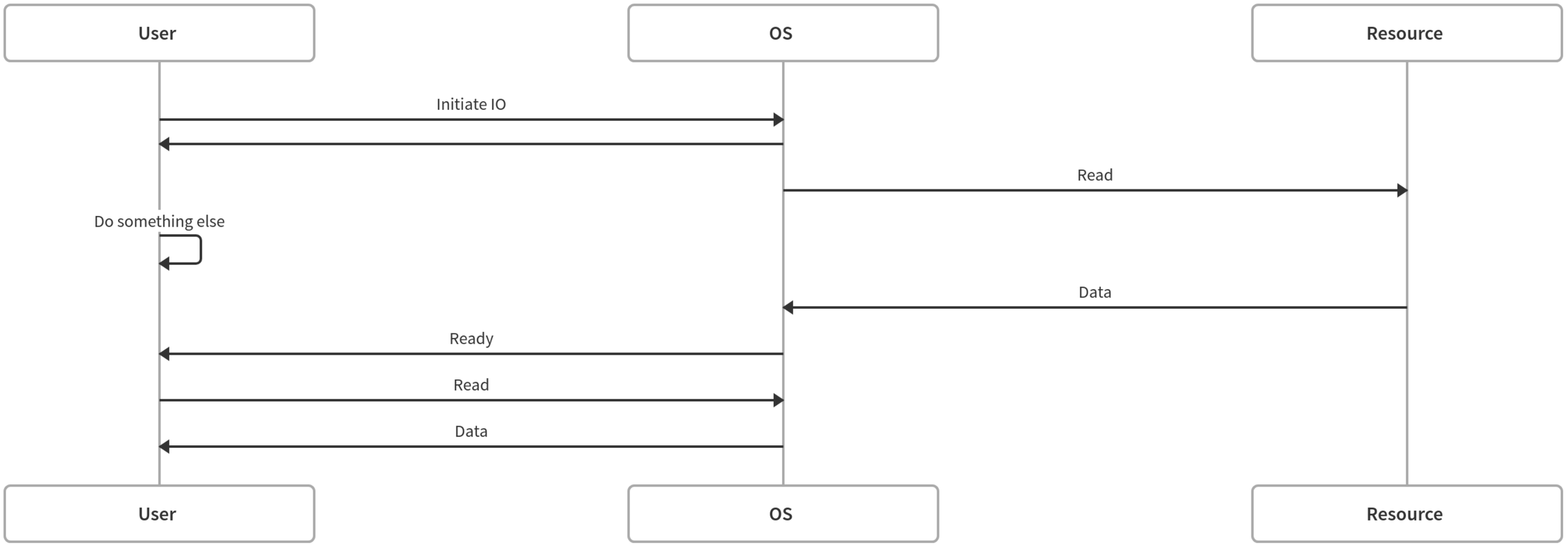

User
OS
Resource
Read
Read
Data
Data
User
OS
Resource
MADE WITH swimlanes.io

Async

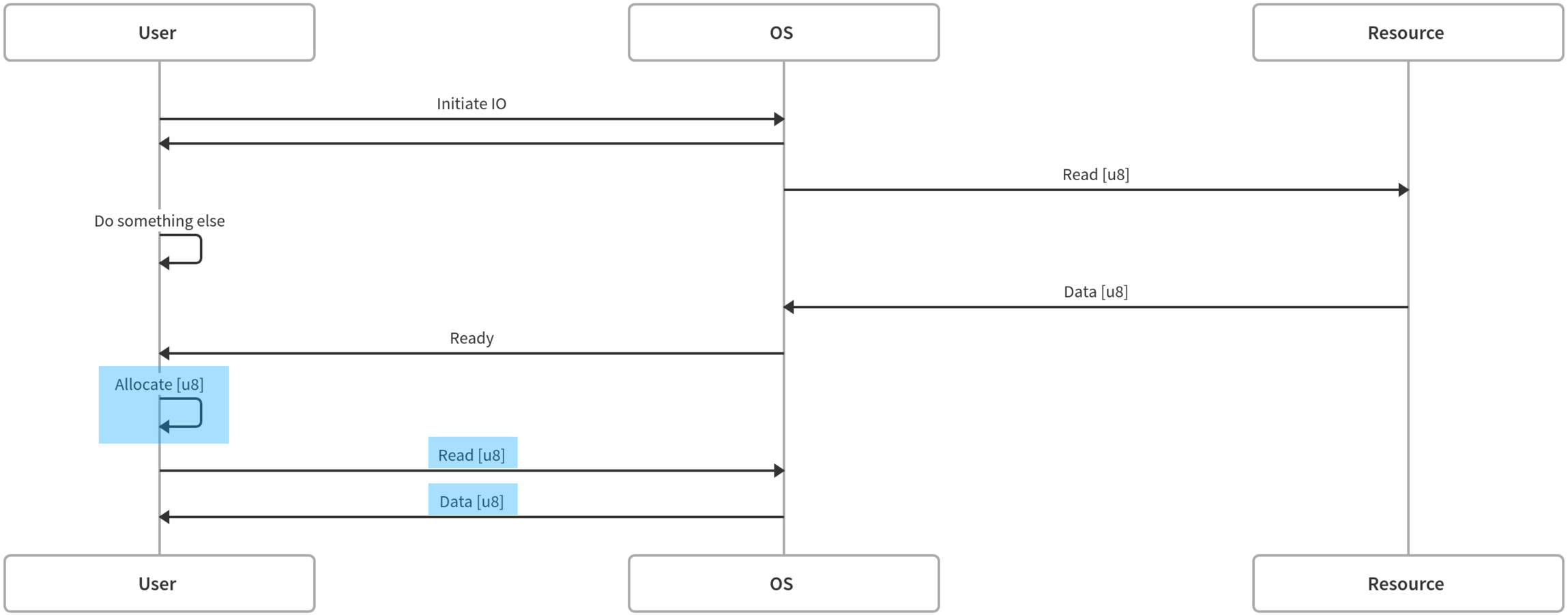
Readiness

Completion

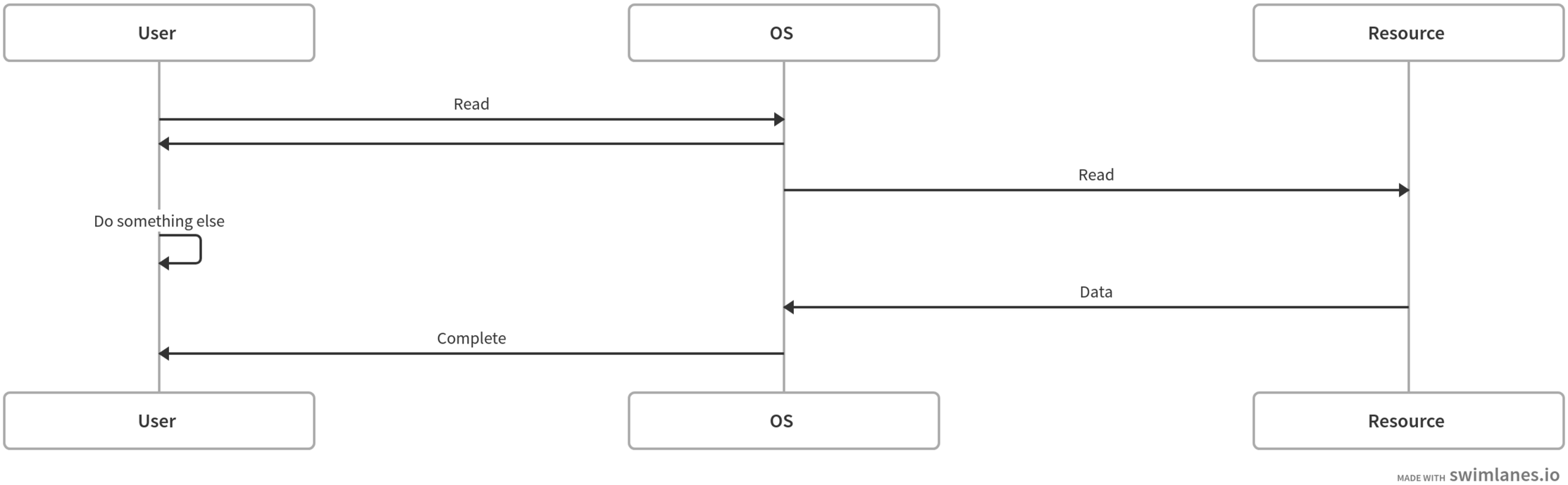
Readiness


User
OS
Resource
Initiate IO
Read
Do something else
Data
Ready
Read
Data
User
OS
Resource
MADE WITH swimlanes.io

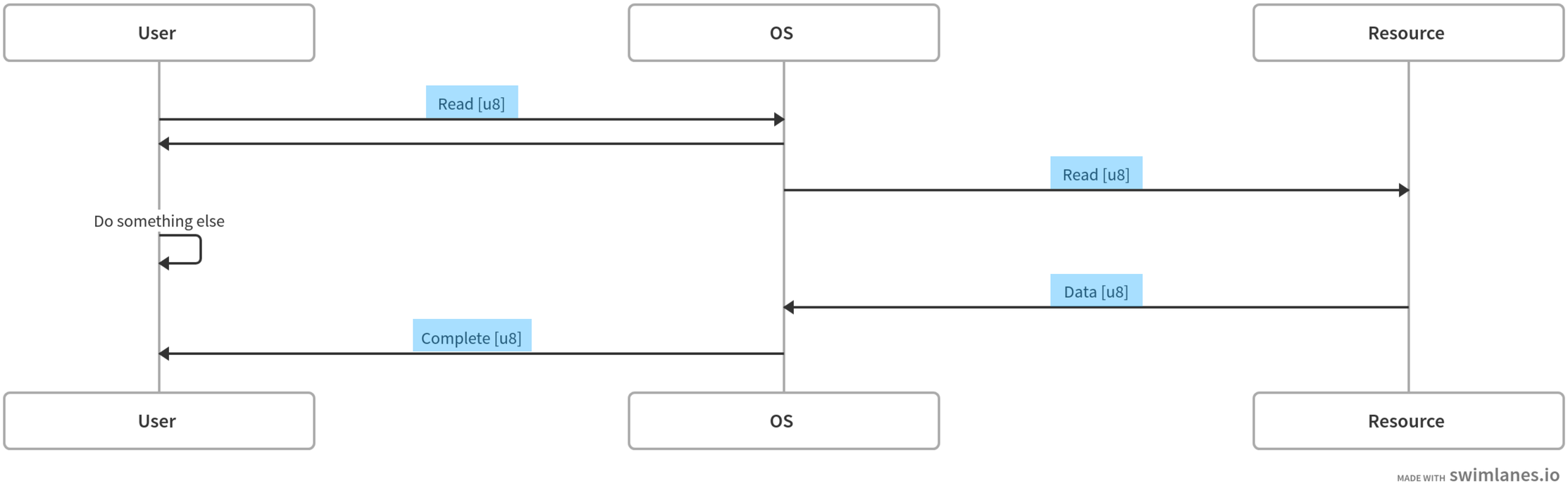
Readiness


User
OS
Resource
Initiate IO
Read [u8]
Do something else
Data [u8]
Ready
Allocate [u8]
Read [u8]
Data [u8]
User
OS
Resource
MADE WITH swimlanes.io

Completion


User
OS
Resource
Read
Read
Do something else
Data
Complete
User
OS
Resource
MADE WITH swimlanes.io

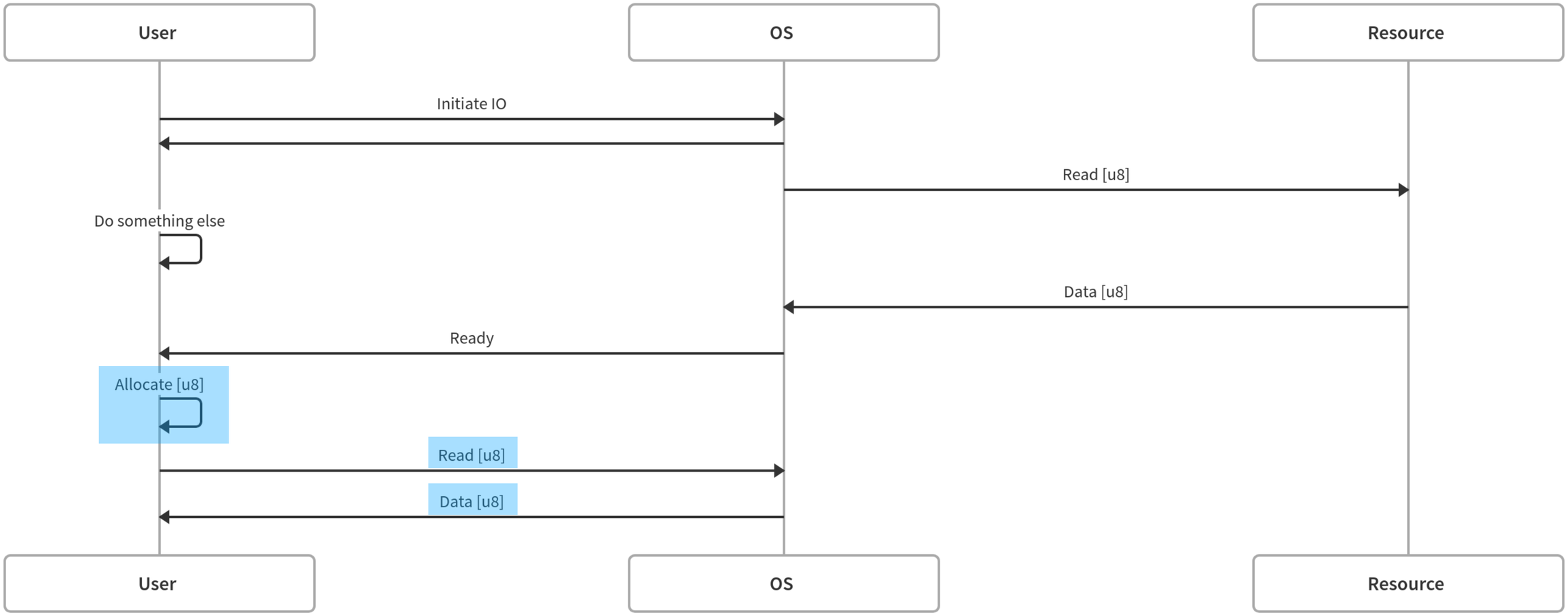
Completion


User
OS
Resource
Read [u8]
Read [u8]
Do something else
Data [u8]
Complete [u8]
User
OS
Resource
MADE WITH swimlanes.io

Async Read Traits

```
trait Read {
  async fn read(&mut self, buf: &mut [u8]) -> Result<usize>;
}
```

Readiness

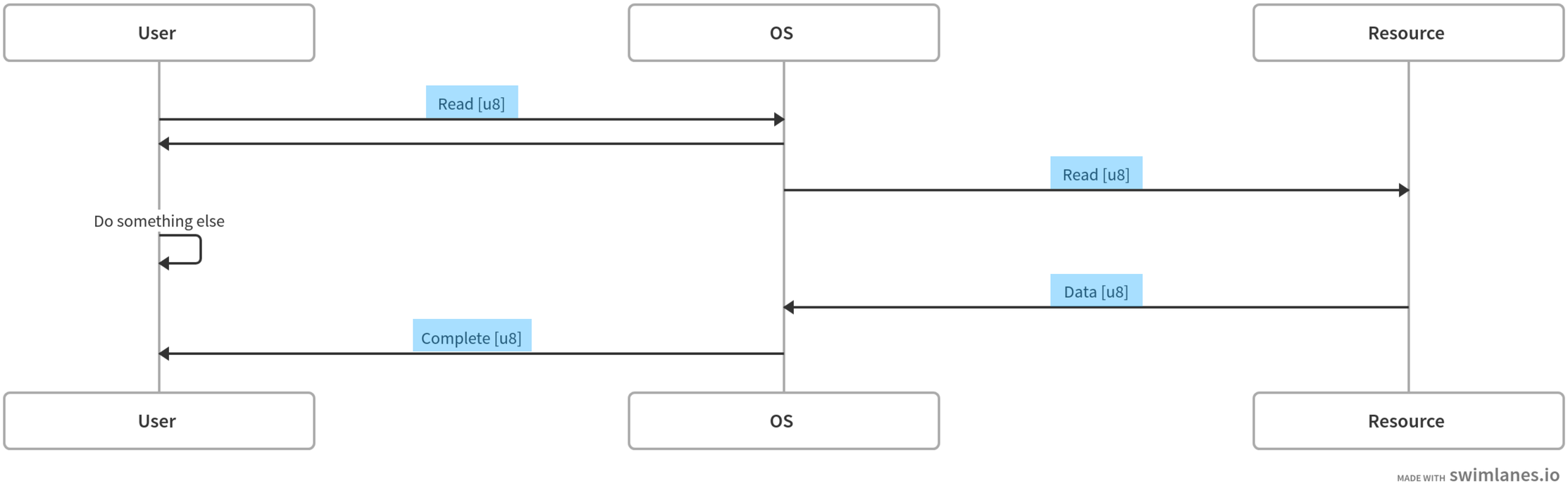

User
OS
Resource
Initiate IO
Read [u8]
Do something else
Data [u8]
Ready
Allocate [u8]
Read [u8]
Data [u8]
User
OS
Resource
MADE WITH swimlanes.io

```
trait Ready {
  async fn ready(&mut self, …) -> Result<…>;
}

trait Read: Ready {
  async fn read(&mut self, buf: &mut [u8]) -> Result<usize>;

  fn non_blocking_read(&mut self, buf: &mut [u8]) -> Result<…>;
}
```

Completion


User
OS
Resource
Read [u8]
Read [u8]
Do something else
Data [u8]
Complete [u8]
User
OS
Resource
MADE WITH swimlanes.io

```
trait BufRead: Read {
  async fn fill_buf(&mut self) -> Result<&[u8]>;
  fn consume(&mut self, amt: usize);
}
```

```
trait OwnedRead {
  async fn read<T>(&mut self, buf: T) -> Result<T>
  where
    T: OwnedReadBuf;
}
```

Summary

```
async fn read_file(file_name: &Path) -> Result<String> {
  let mut f = File::open(file_name).await?;
  let mut result = String::new();
  f.read_to_string(&mut result).await?;
  Ok(result)
}

let contents = read_file("foo.txt".into()).await?;
```

Portability and Interoperability

Many runtimes, one ecosystem

Easier to switch runtime

Easy to start

Async IO Traits

Read

Write

BufRead

Seek

Thank you!

 Nick Cameron
 @nick_r_cameron
 @nrc
nrc@ncameron.org