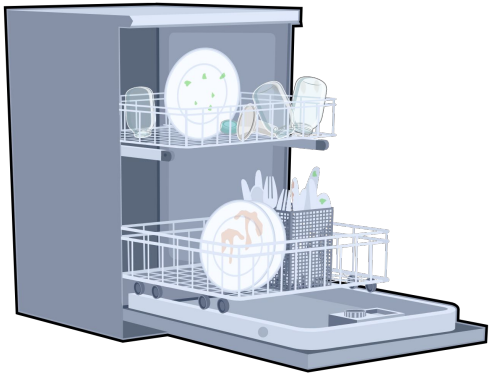
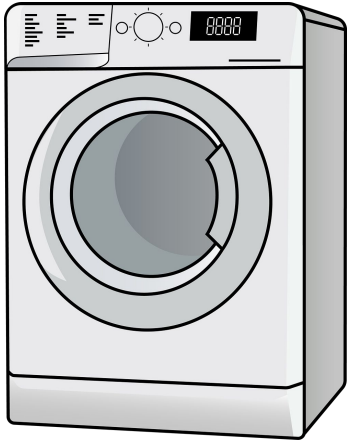
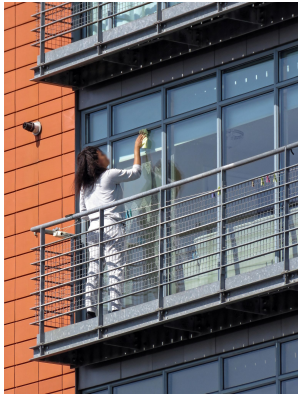
# Async Rust

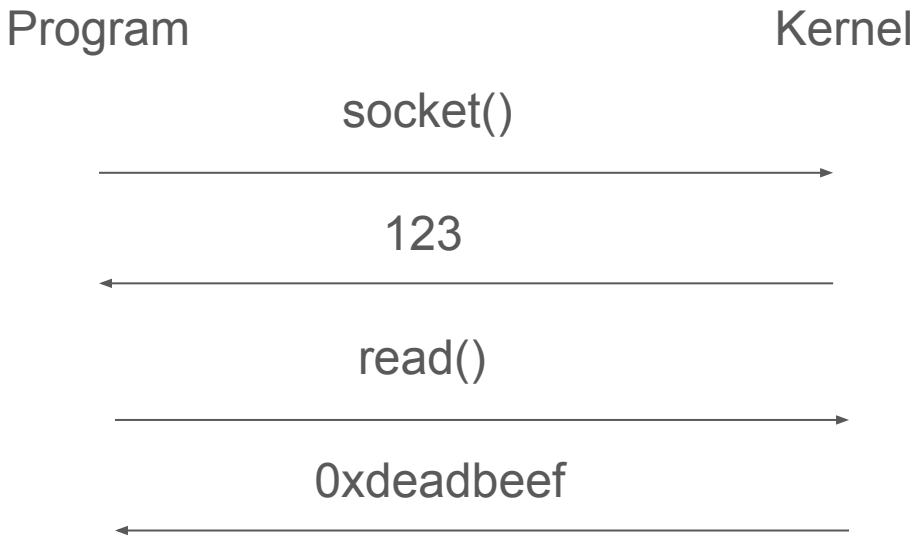
# BORN TO FRAG

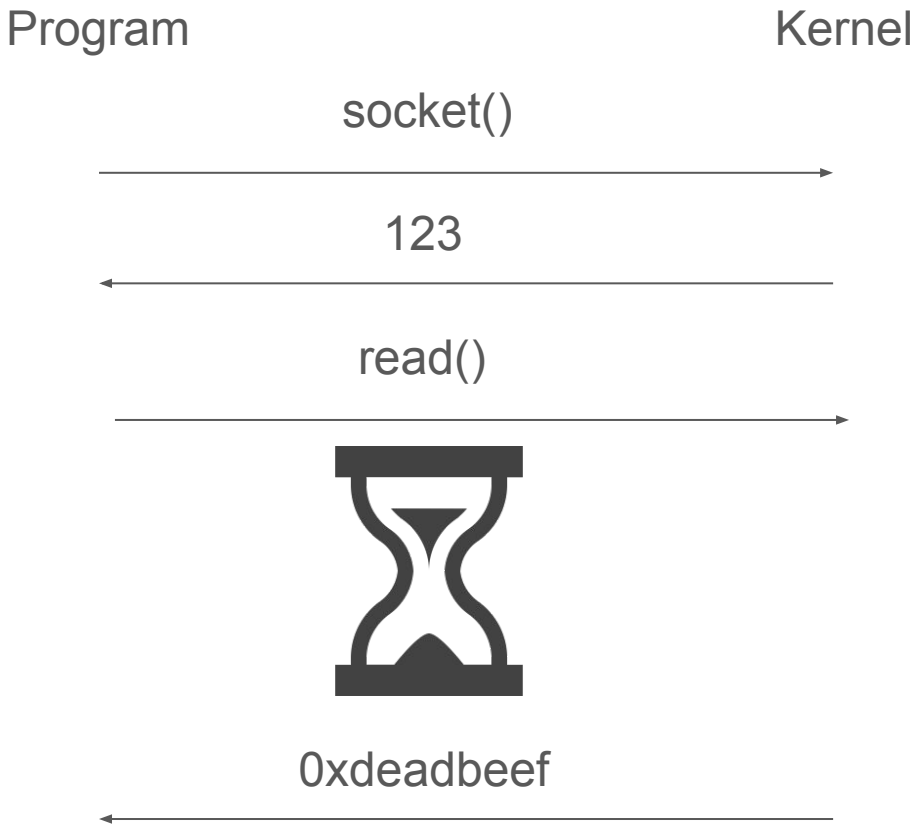

github.com/Kixunil

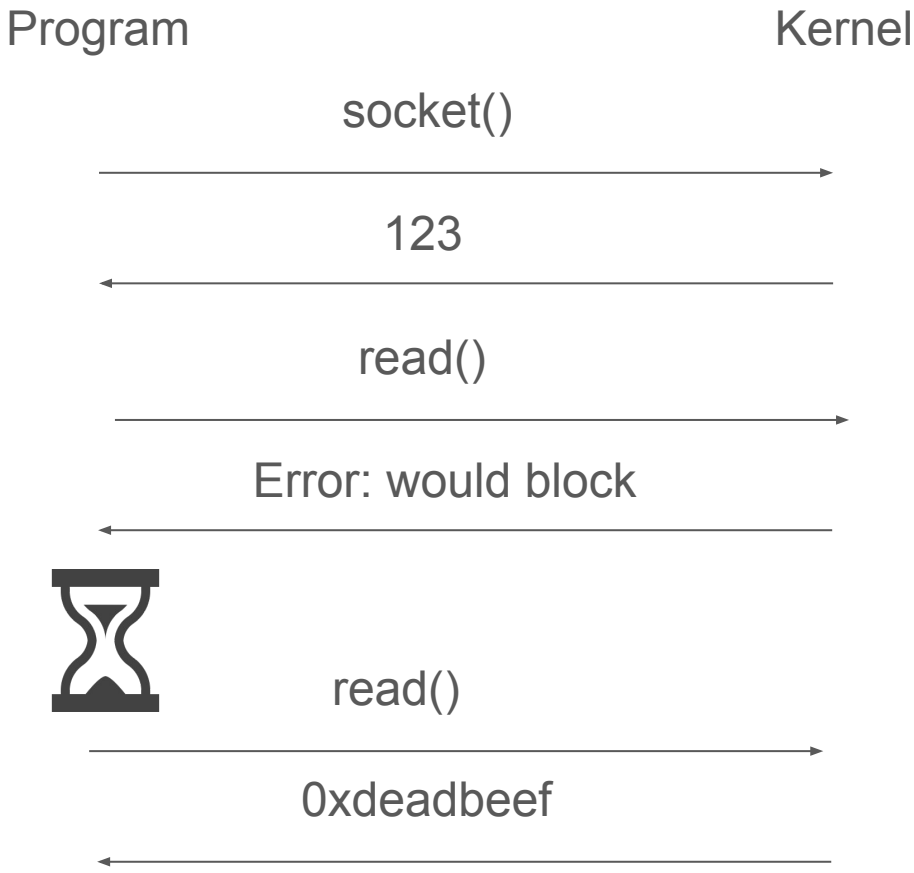


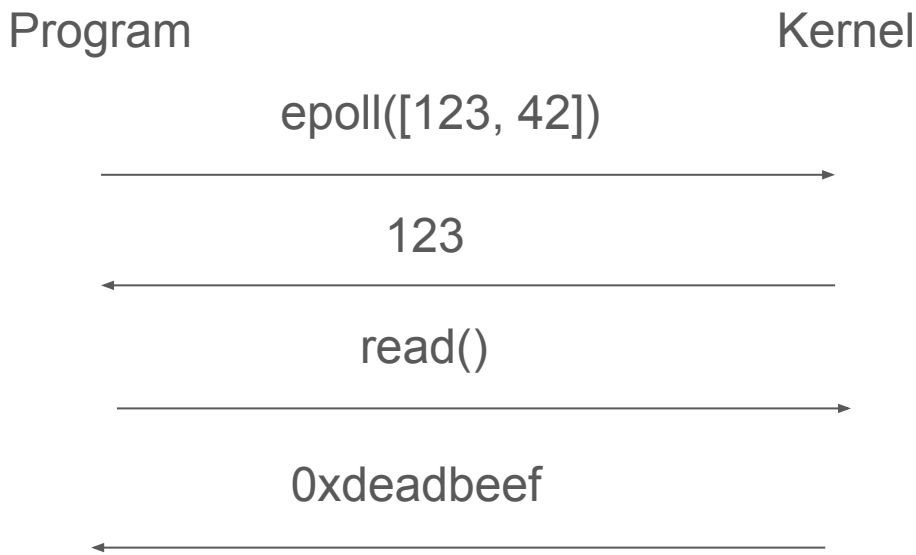




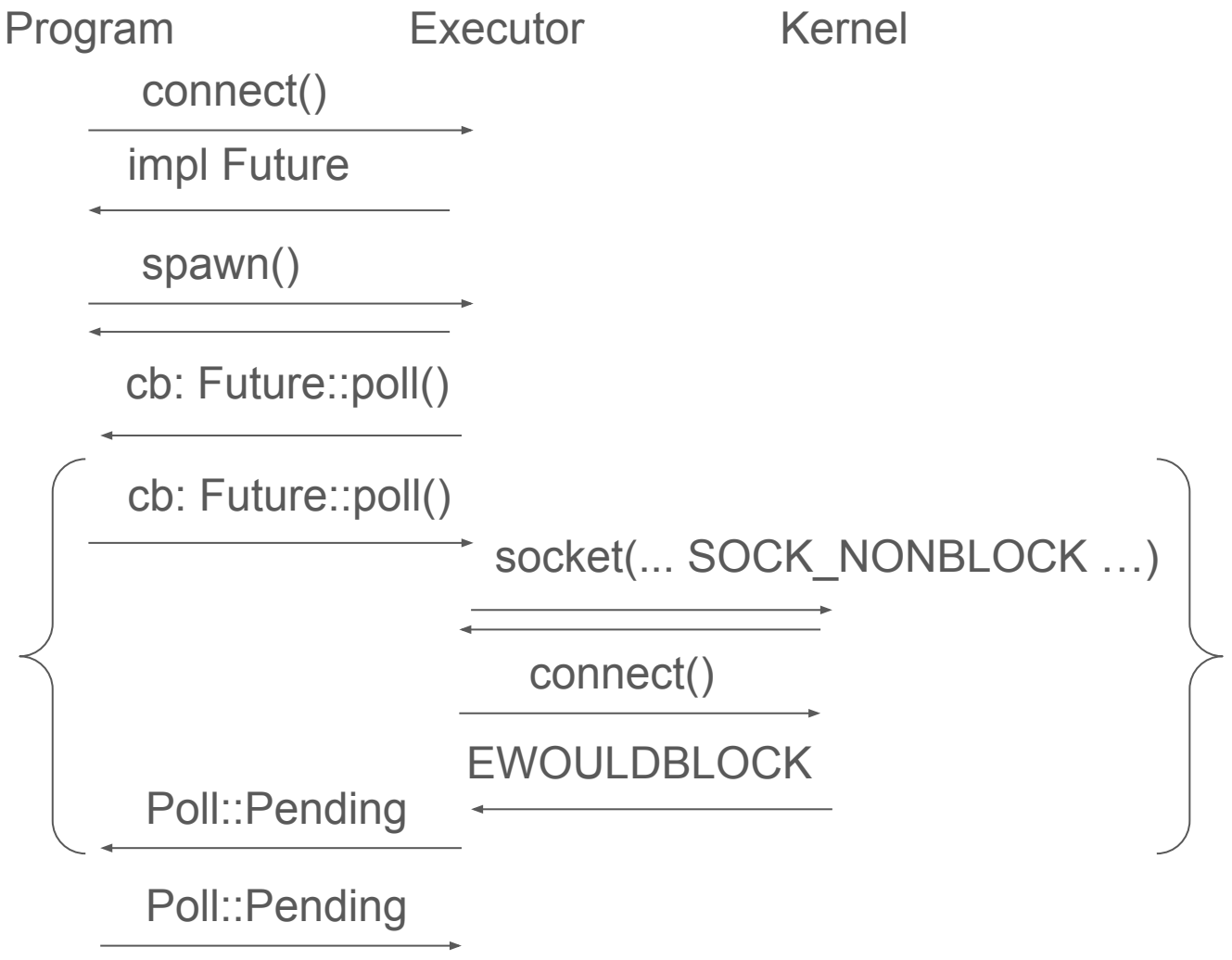

Program
Kernel
socket()
123
read()
0xdeadbeef

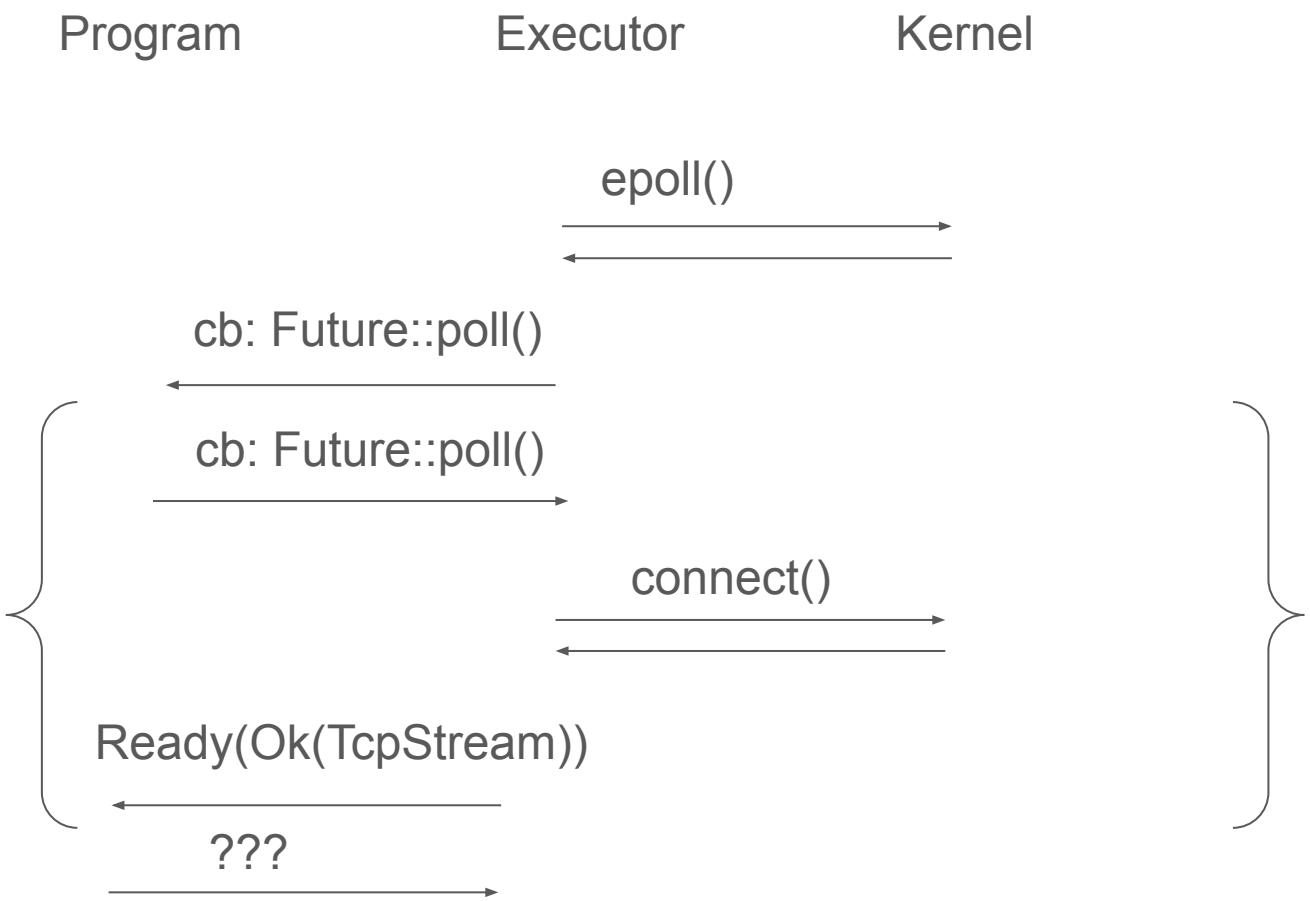

Program
Kernel
socket()
123
read()
0xdeadbeef


Program
Kernel
socket()
123
read()
Error: would block
read()
0xdeadbeef


Program
Kernel
epoll([123, 42])
123
read()
0xdeadbeef

mio


Program
Executor
Kernel
connect()
impl Future
spawn()
cb: Future::poll()
cb: Future::poll()
socket(... SOCK_NONBLOCK …)
connect()
EWOULDBLOCK
Poll::Pending
Poll::Pending


Program
Executor
Kernel
epoll()
cb: Future::poll()
cb: Future::poll()
connect()
Ready(Ok(TcpStream))
???

```rust
fn get_data(address: &str) -> impl Future<Output=std::io::Result<u32>> {
    tokio::net::TcpStream::connect(address)
        .and_then(|stream| {
            let mut buf = [0; 4];
            stream.read_exact(&mut buf)
        })
        .and_then(|| {
            // ???
        })
}
```

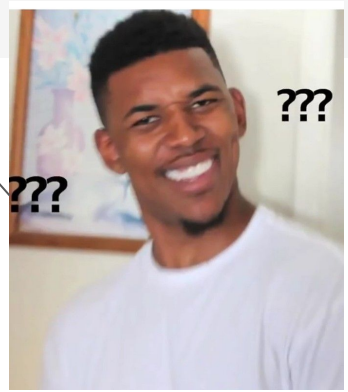
```rust
use tokio; // 1.37.0

use tokio::io::AsyncReadExt;

async fn get_data(address: &str) -> std::io::Result<u32> {
    let mut stream = tokio::net::TcpStream::connect(address).await?;
    let mut buf = [0; 4];
    stream.read_exact(&mut buf).await?;
    Ok(u32::from_be_bytes(buf))
}
```

```rust
use tokio; // 1.37.0

use tokio::io::AsyncReadExt;
use std::future::Future;

fn get_data(address: &str) -> impl Future<Output=std::io::Result<u32>> + Send + Sync + '_ {
    async move {
        let mut stream = tokio::net::TcpStream::connect(address).await?;
        let mut buf = [0; 4];
        stream.read_exact(&mut buf).await?;
        Ok(u32::from_be_bytes(buf))
    }
}
```

```rust
use tokio; // 1.37.0

use tokio::io::AsyncReadExt;
use std::future::Future;

fn get_data(address: &str) -> impl Future<Output=std::io::Result<u32>> + Send + Sync {
    let address = address.to_owned();
    async move {
        let mut stream = tokio::net::TcpStream::connect(&address).await?;
        let mut buf = [0; 4];
        stream.read_exact(&mut buf).await?;
        Ok(u32::from_be_bytes(buf))
    }
}
```

```rust
pub trait Future {
    type Output;

    // Required method
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

```rust
pub trait Future {
    type Output;

    // Required method
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```



```
enum FutureState {
    Init { address: String },
    Connect { address: String, future: ConnectFuture<'???> },
    Read { buf: [u8; 4], future: ReadFuture<'???> },
    Done,
}
```

```rust
fn busy_poll<F: Future>(future: F) -> F::Output {
    use std::task::Poll;

    let mut future = future;
    let mut future = unsafe { std::pin::Pin::new_unchecked(&mut future) };
    loop {
        match future.as_mut().poll(&mut std::task::Context::from_waker(&futures::task::noop_waker())) {
            Poll::Ready(value) => break value,
            Poll::Pending => (),
        }
    }
}
```

```rust
fn busy_poll<F: Future>(future: F) -> F::Output {
    use std::task::Poll;

    let mut future = future;
    let mut future = std::pin::pin!(future);
    loop {
        match future.as_mut().poll(&mut std::task::Context::from_waker(&futures::task::noop_waker())) {
            Poll::Ready(value) => break value,
            Poll::Pending => (),
        }
    }
}
```

```rust
fn busy_poll<F: Future>(future: F) -> F::Output {
    use std::task::Poll;

    let mut future = Box::pin(future);
    loop {
        match future.as_mut().poll(&mut std::task::Context::from_waker(&futures::task::noop_waker())) {
            Poll::Ready(value) => break value,
            Poll::Pending => (),
        }
    }
}
```

```rust
fn a_number(high: bool) -> std::pin::Pin<Box<dyn Future<Output=u32>>> {
    if high {
        Box::pin(async { 21000000 })
    } else {
        Box::pin(async { 42 })
    }
}
```

```rust
use futures::StreamExt;

async fn process_stream<S: futures::Stream>(stream: S) where S::Item: std::fmt::Display {
    let common_state = format!("foo");
    let common_state = std::sync::Arc::new(common_state);
    stream.for_each_concurrent(None, move |item| {
        let common_state = common_state.clone();
        async move {
            println!("{} in context {}", item, common_state);
        }
    }).await
}
```

```rust
use tokio; // 1.37.0

async fn concurrent() {
    let future0 = async { 42 };
    let future1 = async { 21000000 };
    tokio::spawn(future0);
    tokio::spawn(future1);
}
```

```rust
use tokio; // 1.37.0

async fn concurrent() {
    let future0 = async { 42 };
    let future1 = async { 21000000 };
    let handle0 = tokio::spawn(future0);
    let handle1 = tokio::spawn(future1);
    handle0.await;
    handle1.await;
}
```

```
async fn concurrent() {
    let future0 = async { 42 };
    let future1 = async { 21000000 };
    futures::future::join(future0, future1).await;
}
```

```rust
pub trait Stream {
    type Item;

    // Required method
    fn poll_next(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Option<Self::Item>>;

    // Provided method
    fn size_hint(&self) -> (usize, Option<usize>) { ... }
}
```

```rust
async fn display_all<S: futures::Stream>(stream: S) where S::Item: std::fmt::Display {
    let mut stream = std::pin::pin!(stream);
    while let Some(item) = stream.next().await {
        println!("{}", item);
    }
}
```

```rust
use futures::stream::FuturesUnordered;

async fn many_concurrent() {
    let future0 = async { 42 };
    let future1 = async { 21000000 };
    let future2 = async { 37 };

    let mut all = FuturesUnordered::new();
    all.push(Box::pin(future0) as std::pin::Pin<Box<dyn std::future::Future<Output=i32>>>);
    all.push(Box::pin(future1));
    all.push(Box::pin(future2));

    while let Some(item) = all.next().await {
        println!("{}", item);
    }
}
```

```rust
async fn two_streams<S0: futures::Stream, S1: futures::Stream<Item=S0::Item>>(s0: S0, s1: S1)
    where S0::Item: std::fmt::Display
{
    let mut s0 = std::pin::pin!(s0.fuse());
    let mut s1 = std::pin::pin!(s1.fuse());
    loop {
        futures::select! {
            i0 = s0.next() => {
                match i0 {
                    Some(item) => println!("item from stream 0: {}", item),
                    None => println!("stream 0 ended"),
                }
            },
            i1 = s1.next() => {
                match i1 {
                    Some(item) => println!("item from stream 1: {}", item),
                    None => println!("stream 1 ended"),
                }
            }
        }
    }
}
```

```rust
use futures::future::Either;

async fn two_streams<S0: futures::Stream, S1: futures::Stream<Item=S0::Item>>(s0: S0, s1: S1)
    where S0::Item: std::fmt::Display
{
    let mut stream = std::pin::pin!(futures::stream::select(s0.map(Either::Left), s1.map(Either::Right)));
    while let Some(item) = stream.next().await {
        match item {
            Either::Left(item) => println!("item from stream 0: {}", item),
            Either::Right(item) => println!("item from stream 1: {}", item),
        }
    }
}
```