Async Programming

Rust, In Theory and in Practice

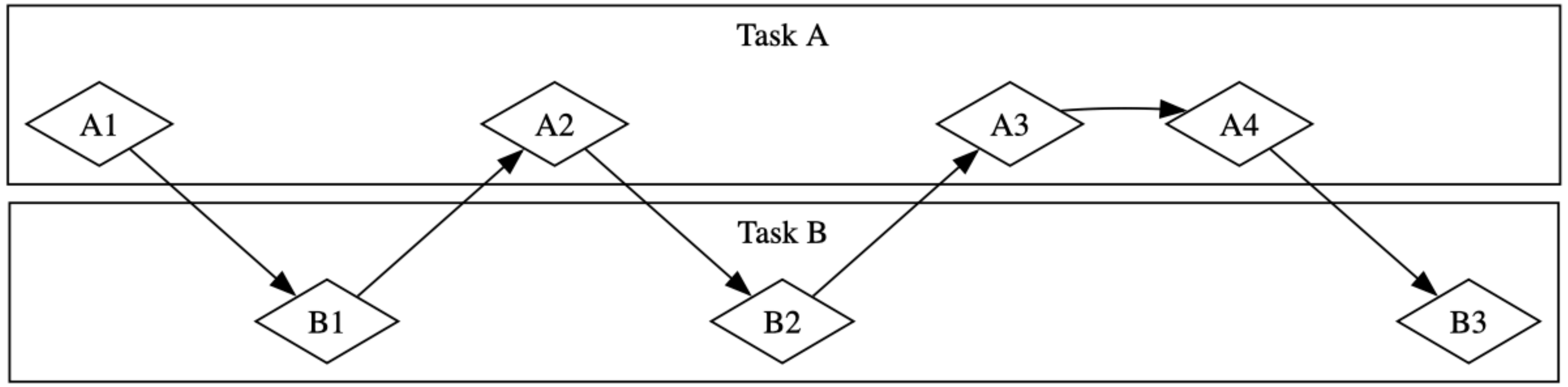
CAS CS 392 M1

Motivation

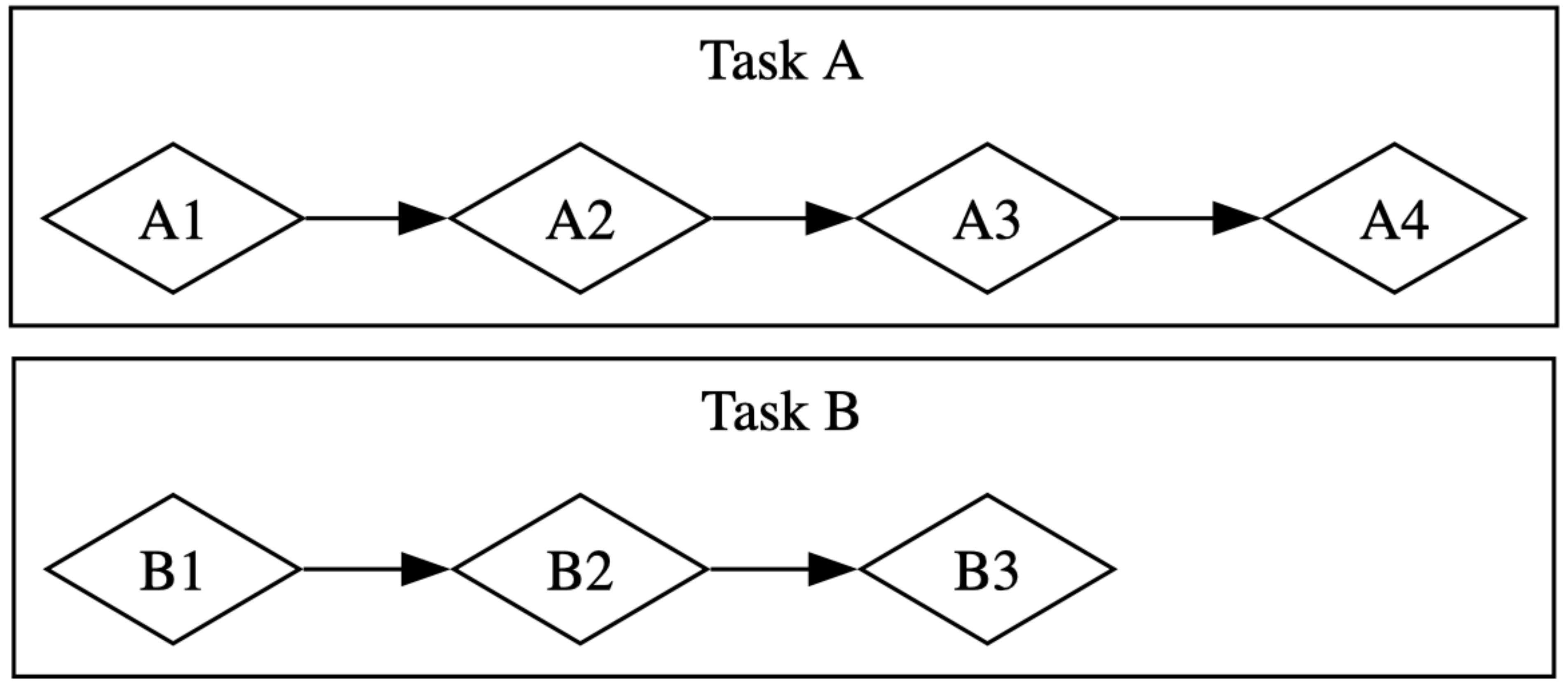
It would be no fun if our computer became useless every time we downloaded or exported something...

Concurrency and **Parallelism** allow our computers to multitask

Concurrency vs. Parallelism

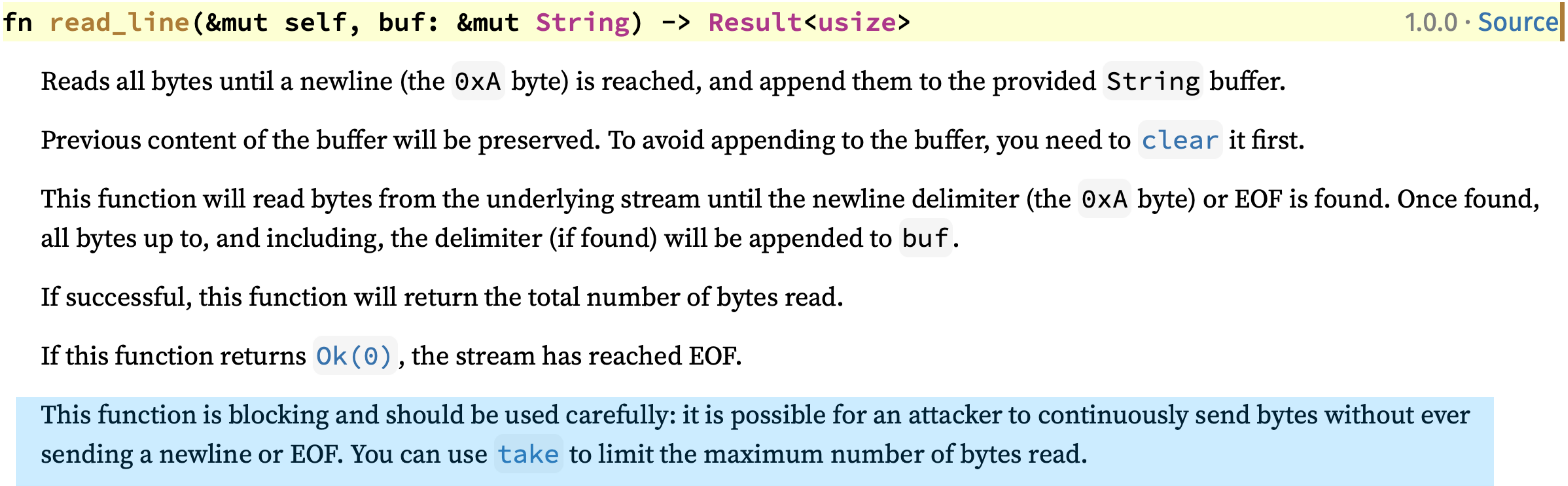


concurrent



parallel

Blocking Operations



```
fn read_line(&mut self, buf: &mut String) -> Result<usize>
```

Reads all bytes until a newline (the `0xA` byte) is reached, and append them to the provided `String` buffer.

Previous content of the buffer will be preserved. To avoid appending to the buffer, you need to `clear` it first.

This function will read bytes from the underlying stream until the newline delimiter (the `0xA` byte) or EOF is found. Once found, all bytes up to, and including, the delimiter (if found) will be appended to `buf`.

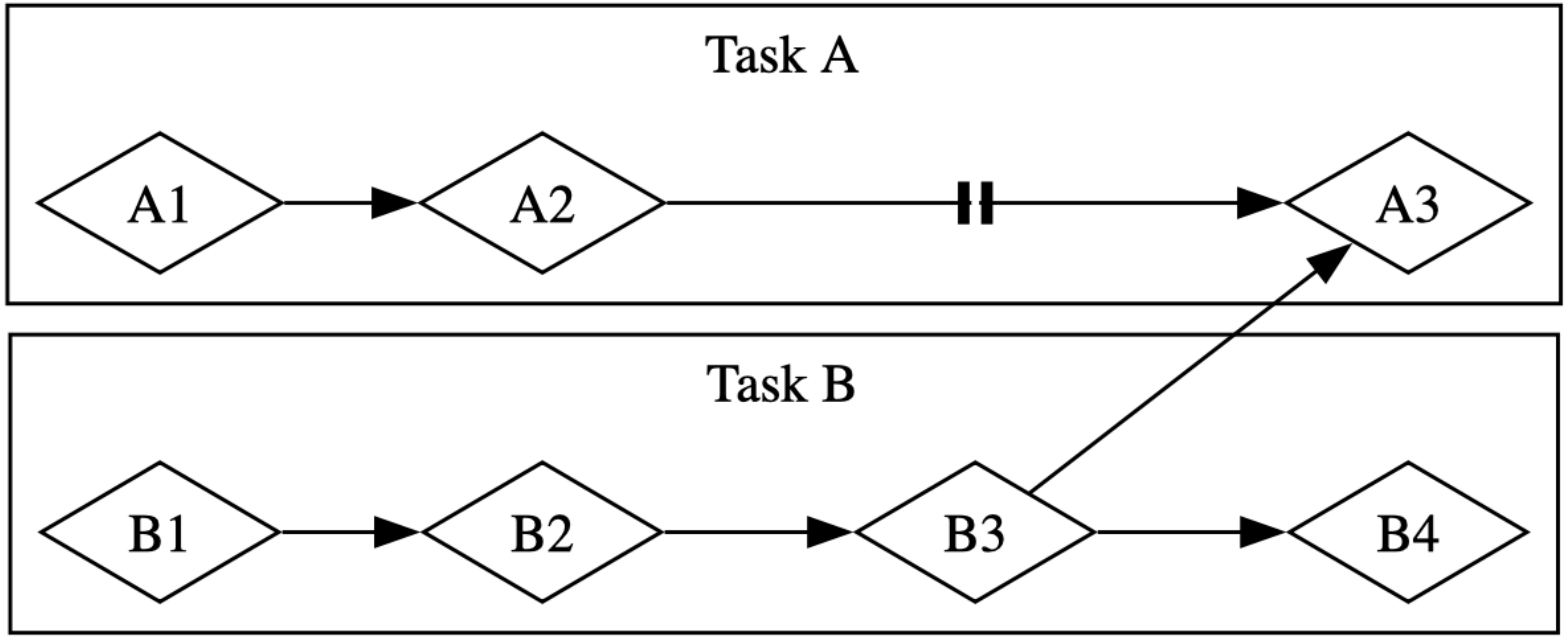
If successful, this function will return the total number of bytes read.

If this function returns `Ok(0)`, the stream has reached EOF.

This function is blocking and should be used carefully: it is possible for an attacker to continuously send bytes without ever sending a newline or EOF. You can use `take` to limit the maximum number of bytes read.

An operation is **blocking** if a program cannot make progress until the operation is over

Blocking Operations (A Picture)



The process A3 is blocked by B3 in this parallel workflow

Asynchronous Programming

```
async fn get_page(url: &str) -> Option<String>{
    reqwest::get(url)
        .await
        .ok()?
        .text()
        .await
        .ok()
}
```

The async abstraction gives us a way to call functions in a non-blocking way

Futures

"A **future** is a value that may not be ready now but will become ready at some point in the future."

Futures are implemented via the **Future** *trait*

```rust
pub trait Future {
    type Output;

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

Polling

```
pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

The **poll** function is used by asynchronous runtimes to determine whether or not a future is ready to be used

We rarely interact directly with **Future::poll**

Crates for Asynchronous Programming

```toml
[dependencies]
reqwest = "0.12"
tokio = { version = "1", features = ["full"] }
futures = "0.3"
```

async blocks

```
async {
    // within an async block we
    // can use the await keyword

    // remember: blocks are expressions

    // an async block evaluates to a Future

}
```

The await keyword

```
async {
     for i in 1..10 {
         println!("first task: {i}");
         tokio::time::sleep(
             tokio::time::Duration::from_millis(500)
         ).await;
     }
 }
```

await is used within an **async** block in order to wait for a future

In the we *join* futures, the **await** keywords tells the runtime "I'm waiting, I cede my time until it's ready"

Async Runtimes

```
let rt = tokio::Runtime::new().unwrap();
rt.block_on(future)
```

Futures are *lazy*. This means that the computation associated with a future is not run until it is given to a **runtime**

Rust has many async runtimes, we'll be using **tokio** for the examples

Cheatsheet

tokio::task::spawn	spawn an asychronous task that starts running immediately
futures::future::join	run two futures until they both finish
futures::future::select	run two futures until one finishes, and then determine what to do in each case
tokio::sync::mpsc::unbounded_channel	create a channel for passing messages between futures

demo

The "Invisible State Machine" (Rabbit Hole)

```
enum PageTitleFuture<'a> {
    Initial { url: &'a str },
    GetAwaitPoint { url: &'a str },
    TextAwaitPoint { response: Response },
}
```

Since futures are lazy, there's a bunch of data they keep track of

In particular, we can think of each **await** as triggering a transition in a state machine

Pinning (Rabbit Hole)

```rust
pub struct Pin<Ptr> {
    pointer: Ptr,
}
```

Sometimes that extra data can lead to *self-referential structures*, i.e., structures that contain pointers to their own data

This is why we to **pin** our types when working with futures

(pinning is bizarre, we won't talk much about it)

Workshop

Tasks

Implement a function that gets an webpage and also prints a message describing how much time has elapsed (maybe 1 message per second)

Implement a function **timeout** function, that run a future for a given time (from the text). *Challenge.* Take a closure instead (note: I haven't battle tested this one)

Build a **Join** structure for the joining two futures (this meaning using **poll**)