Rust: async

Hafta - 6

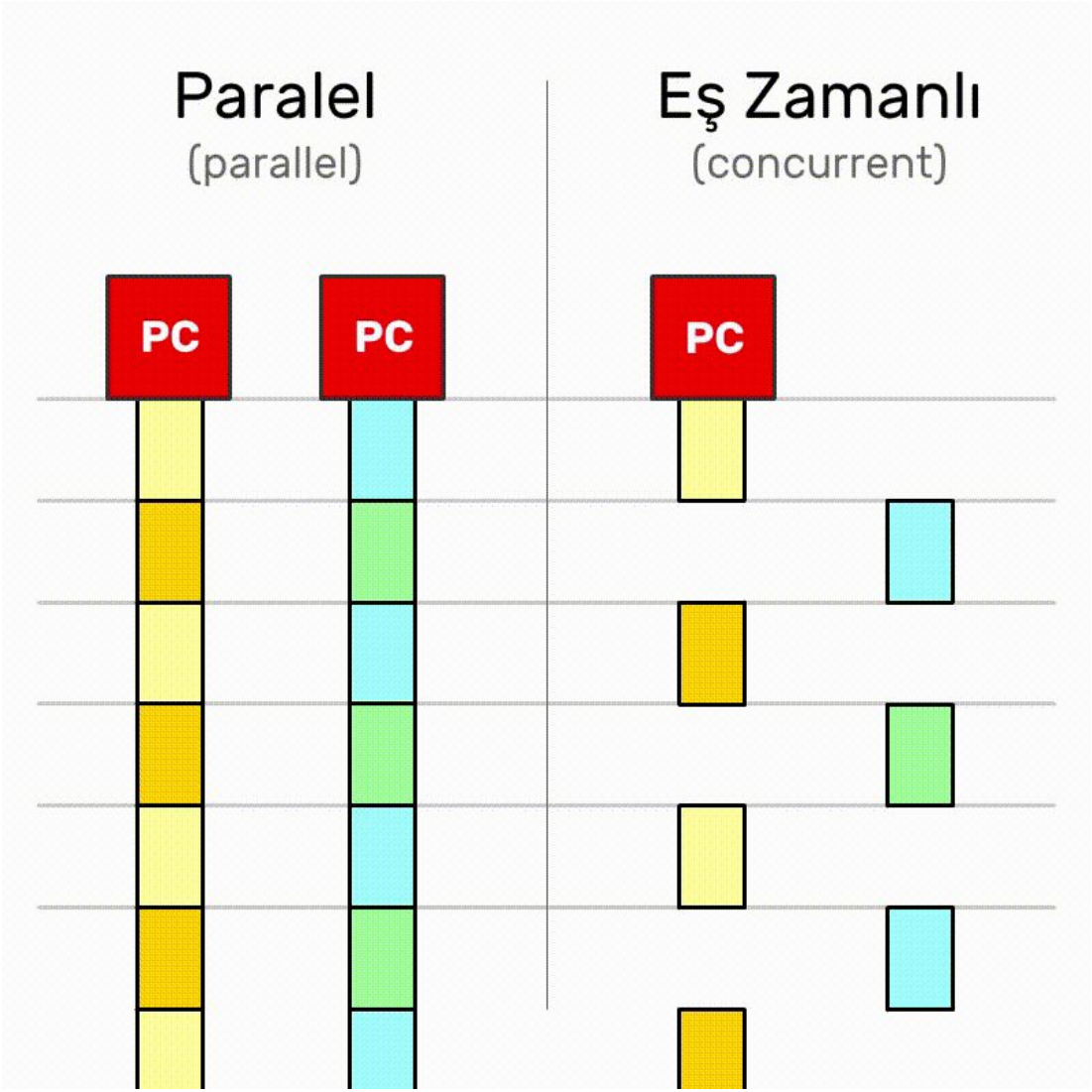
async nedir?

Parallel:

Birden fazla thread'in farklı işlemci çekirdeklerinde
aynı anda yürütülmesi

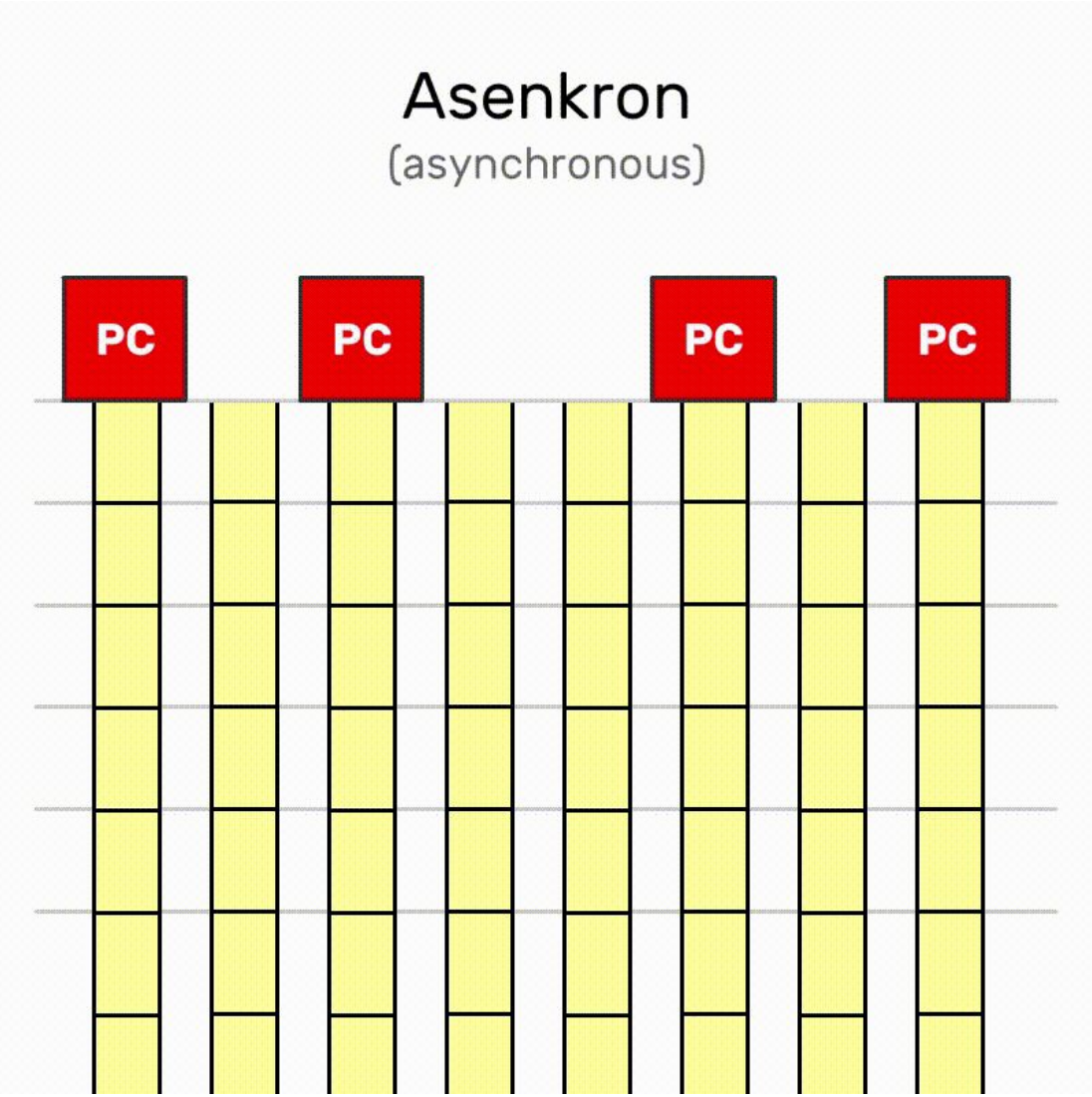
Concurrent (Eş Zamanlı):

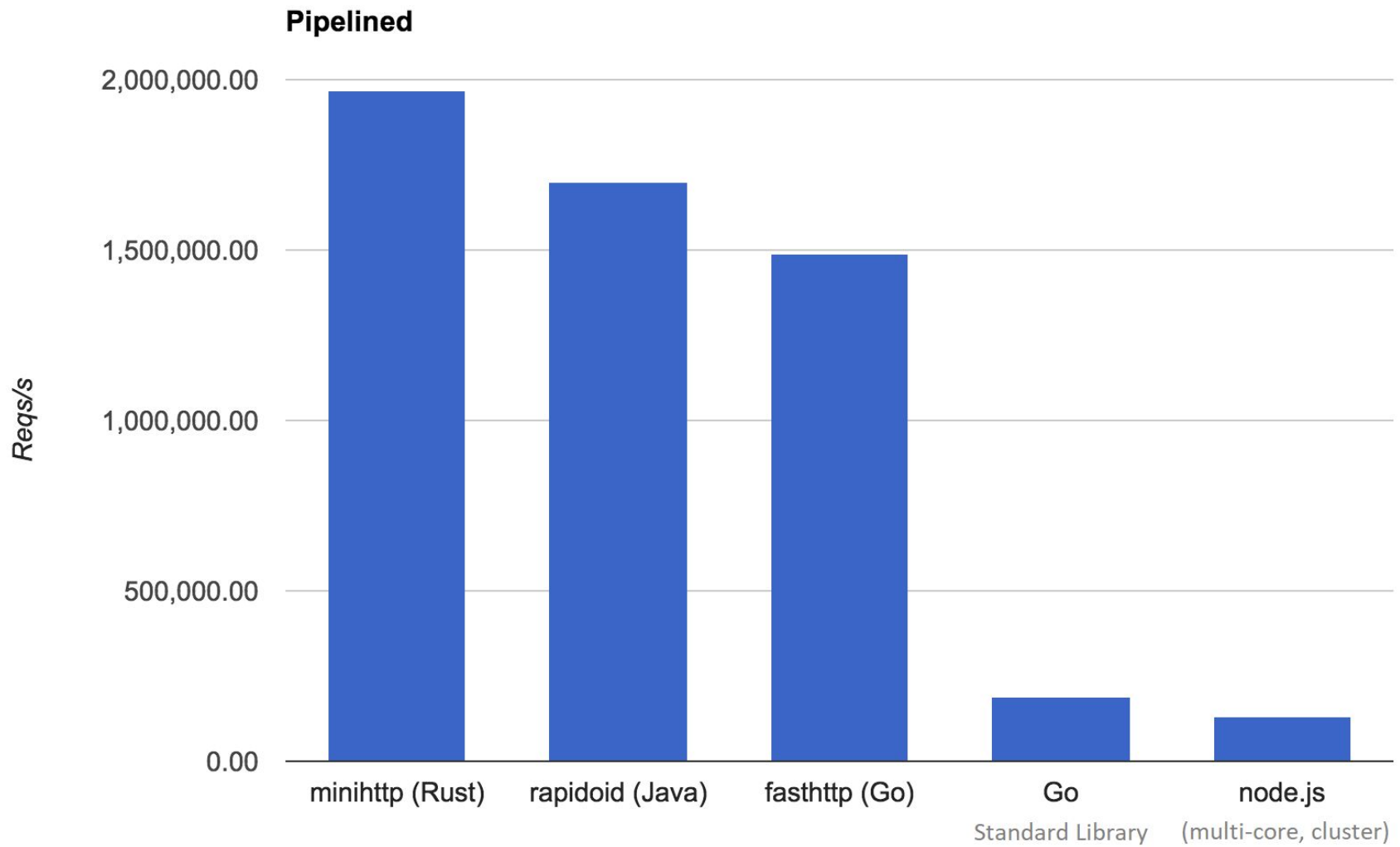
Tek thread ve tek çekirdek ile birden fazla
programı parça parça yürütme


Paralel
(parallel)
Eş Zamanlı
(concurrent)
PC
PC
PC

Asynchronous (Asenkron):

Programın Concurrent veya Parallel bir yapı sayesinde bloklanmadan yürütülmesi


Asenkron
(asynchronous)
PC
PC
PC
PC


Pipelined
2,000,000.00
1,500,000.00
1,000,000.00
500,000.00
0.00
Reqs/s
minihttp (Rust)
rapidoid (Java)
fasthttp (Go)
Go
Standard Library
node.js
(multi-core, cluster)

Future

Asenkron bir iş.

Future

```rust
async fn ornek() -> i32 {
    42
}
```

```rust
fn ornek() -> impl Future<Output=i32> {
    async {
        42
    }
}
```

İkisi aynı şeydir. İlki diğerine dönüştürülür.
“Syntax Sugar”

Executor

Future'ları alıp çalıştıran runtime.

Örneğin: async-std, tokio

Executor

```
use async_std::task;

async fn ornek() -> i32 {
    42
}

fn main() {
    task::block_on(async {
        println!("ornek() = {}", ornek().await);
    });
}
```

```
> ornek() = 42
```

Executor

```rust
use async_std::task;

async fn ornek() -> i32 {
    42
}

async fn async_main() {
    println!("ornek() = {}", ornek());
}

fn main() {
    task::block_on(async_main());
}
```

```
>  --> src/main.rs:8:27
   |
8  |     println!("ornek() = {}", ornek());
   |                              ^^^^^^^ `impl Future` cannot be formatted
with the default formatter
   |
```

Executor

Future'lar tek başına çalışmaz.

Bir Executor'a "al bunu çalıştır" diyerek verilmeli.

Bunu Future'u .await ederek yapabiliriz.

Executor

```
use async_std::task;

async fn ornek() -> i32 {
    42
}

async fn async_main() {
    println!("ornek() = {}", ornek().await);
}

fn main() {
    task::block_on(async_main());
}
```

```
> ornek() = 42
```

Executor

.await ile beklenen future'lar sonuçlanmadan program bir aşağıdaki satırı çalıştırmaz.

Executor, .await ile beklenen future'u yield edip(kenara koyup) başka bir future'u çalıştırmaya devam edebilir. Bu da asenkronluğu sağlar.

Örneğin siz bir dosya okuma işlemini .await ederken, dosya hazır olana kadar Executor başka bir .await ile beklenen future'u çalıştırmaya geçebilir.

Executor

```rust
use std::time::Duration;
use async_std::task;

async fn bekle() {
    task::sleep(Duration::from_millis(1000)).await;
    println!("Merhaba");
}

async fn async_main() {
    bekle().await;
    bekle().await;
}

fn main() {
    task::block_on(async_main());
}
```

```
- 1 saniye bekler -
> Merhaba
- 1 saniye bekler -
> Merhaba
```

Örnek

Terminal'den okuduğunu tek seferlik tekrar terminal'e yazma (echo)

```
use async_std::io::{WriteExt, self}; // std::io değil, async_std::io kullanılmalı.
use async_std::task;

async fn terminalden_oku() -> io::Result<String> {
    let mut buffer = String::new();

    io::stdin().read_line(&mut buffer).await?;

    Ok(buffer)
}


async fn async_main() -> io::Result<()> {
    let okunan = terminalden_oku().await?;
    io::stdout().write_all(okunan.as_bytes()).await?;

    Ok(())
}

fn main() -> io::Result<()>{
    task::block_on(async_main())
}
```

```
< selam
> selam
```

Örnek

Terminal'den okuduğunu tekrar terminal'e yazma (echo, sonsuz döngü kırılana kadar)

```
use async_std::io::{WriteExt, self}; // std::io değil, async_std::io kullanılmalı.
use async_std::task;

fn main() -> io::Result<()> {
    task::block_on(async {
        let stdin = io::stdin();
        let mut stdout = io::stdout();
        let mut line = String::new();

        loop {
            let n = stdin.read_line(&mut line).await?;

            // EOL
            if n == 0 {
                return Ok(());
            }

            stdout.write_all(line.as_bytes()).await?;
            stdout.flush().await?;
            line.clear();
        }
    })
}
```

```
< selam
> selam
```

Örnek

async TCP Server

```
use async_std::io::{WriteExt, ReadExt, self};
use async_std::task;
use async_std::net::{TcpListener, TcpStream};

async fn on_connection(mut stream: TcpStream) -> io::Result<()> {
    println!("New Connection: {}", stream.peer_addr()?);

    let mut buffer = [0u8; 1024];
    loop {
        let len = stream.read(&mut buffer).await?;
        if len > 0 {
            println!("{}", String::from_utf8_lossy(&buffer[..len]));

            stream.write(&buffer).await?;
        } else {
            println!("Disconnected: {}", stream.peer_addr()?);
            break
        }
    }

    Ok(())
}

async fn async_main() -> io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:8080").await?;
    println!("Listening on {}", listener.local_addr()?);

    loop {
        let (stream, addr) = listener.accept().await?;
        task::spawn(on_connection(stream));
    }
}

fn main() -> io::Result<()> {
    task::block_on(async_main())
}
```

Sıra Sizde

isteyen 1, isteyen 2 ve 3'ü yapabilir

1. `async-std crate'ini kullanarak bir async chat sunucusu yazın`

   - Bağlı kullanıcı listesini Arc<Mutex<>> ile tutabilirsiniz.

   - Herkesin yazdığı herkese gönderilmeli.

2. `Bir dizide verilmiş n değerlerini, n'inci asal sayıyı bulmak için bir thread oluşturup hesaplamayı yapın ve n. asal sayının ne olduğu sonucunu yazdırın.`

   - Örn: [10, 20, 300, 400, 5000], 10. asal sayıyı bir thread, 20.'yi bir thread, 5000. asal sayıyı da başka bir thread hesaplayacak.

3. `2. görevi MPSC channel'lar ile kullanarak yapın ve threadler buldukları sonuçları tek bir okuyucuya göndersin ve kaçıncı asal sayının değeri neymiş o consumer ekrana yazdırsın.`

Yardımcı linkler:

- https://doc.rust-lang.org/std/sync/atomic/struct.AtomicU64.html
- https://doc.rust-lang.org/std/sync/struct.Arc.html
- https://doc.rust-lang.org/std/sync/struct.Mutex.html
- https://doc.rust-lang.org/std/sync/mpsc/