Dr Dinu Dragan

PARALELNE I DISTRIBUIRANE ARHITEKTURE I JEZICI (ČAS 10)

ŠTA RADIMO DANAS?

- Asinhrono programiranje

ASINHRONO PROGRAMIRANJE

ASINHRONI ZADACI

- Umesto da se, u nekoj konkurentnoj i paralelnoj implementaciji, za svaki novi zahtev kreira nova nit (što bi moglo da proizvode stotine hiljada niti), koriste se asinhroni zadaci (**asynchronous tasks**) koji na asinhroni način prepliću (kombinuju) mnoge nezavisne aktivnosti na jednoj niti ili grupu niti u protočnom sistemu
- Asinhroni zadaci su slični nitima, ali u odnosu na niti
  - brže se kreiraju
  - lakše i efikasnije se prebacuje kontrola između njih
  - preopterećenje memorije je značajno manje
- Generalno, asinhroni kod jako liči na kod koji proizvodi niti, ali se operacijama koje mogu dovesti do blokiranja (**I/O pozivi**, **muteksi**) rukuje drugačije
- Tretiranje ovih situacija drugačije, zapravo daje bolji uvid Rast okruženju u to kako će se kod ponašati i omogućuje bolju optimizaciju čitavog koda

ASINHRONI ZADACI

- Čet server bi se u režimu sa više niti implementirao na sledeći način:

```
use std::{net, thread};

let listener = net::TcpListener::bind(address)?;

for socket_result in listener.incoming() {
    let socket = socket_result?;
    let groups = chat_group_table.clone();
    thread::spawn(|| {
        log_error(serve(socket, groups));
    });
}
```

- Za svaku konekciju postoje dolazni paketi za parsiranje, odlazni paketi koji se moraju sastaviti, bezbednosni parametri za upravljanje, pretplate na čet grupe koje se moraju pratiti i sl.
- Za svaku konekciju, pokreće se nova nit koja izvršava funkcije servera koja je fokusirana na opsluživanje jedne konekcije

ASINHRONI ZADACI

- Čet server bi se u režimu asinhronih zadataka implementirao na sledeći način:

```
use async_std::{net, task};

let listener = net::TcpListener::bind(address).await?;

let mut new_connections = listener.incoming();
while let Some(socket_result) = new_connections.next().await {
    let socket = socket_result?;
    let groups = chat_group_table.clone();
    task::spawn(async {
        log_error(serve(socket, groups).await);
    });
}
```

- Primer pokazuje upotrebu **async_std** sanduka, njegovih modula za mrežnu komunikaciju (**net**) i asinhrone zadatke (**task**), kao **.await** metode koja se poziva da bi blokirala izvršavanje ako nema novih konekcija (u osnovi nema velike razlike u strukturi u odnosu na rešenje samo pomoću paralelnog programiranja)

SINRONO VS. ASINHRONO

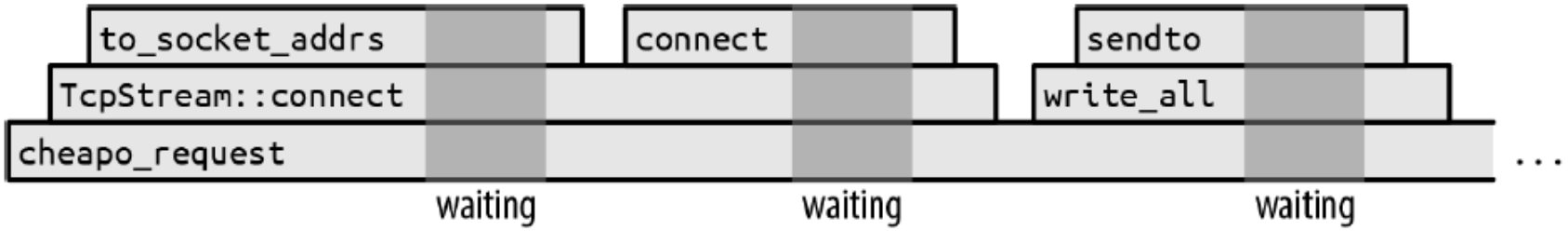
- Sinhronizovani primer HTTP redirektiranja:

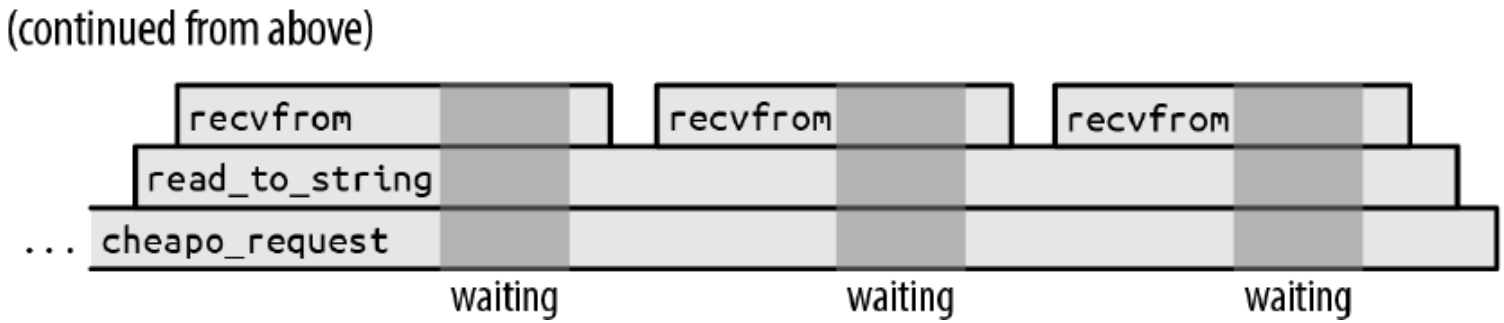
```
use std::io::prelude::*;
use std::net;

fn cheapo_request(host: &str, port: u16, path: &str)
                        -> std::io::Result<String>
{
    let mut socket = net::TcpStream::connect((host, port))?;

    let request = format!("GET {} HTTP/1.1\r\nHost: {}\r\n\r\n", path, host);
    socket.write_all(request.as_bytes())?;
    socket.shutdown(net::Shutdown::Write)?;

    let mut response = String::new();
    socket.read_to_string(&mut response)?;

    Ok(response)
}
```

- Bolje koristiti **surf** ili **reqwest** koji bi ovo uradili kako treba i asinhrono

SINRONO VS. ASINHRONO

- Poziv sistemskog steka u izvršavanju **cheapo_request** funkcije ima sledeći izgled

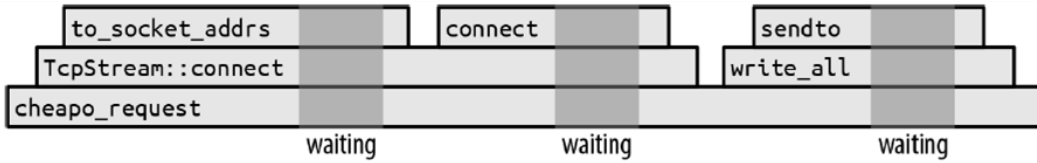


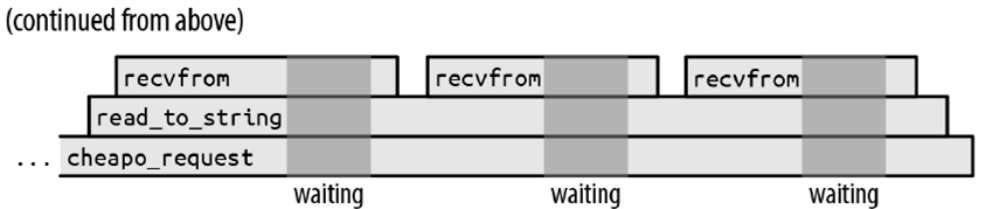


- Svaki poziv funkcije je predstavljen kao pravougaonik postavljen na funkciju koja ga poziva
- Funkcija **cheapo_request** se izvršava sve vreme

SINRONO VS. ASINHRONO

- Funkcija **cheapo_request** poziva funkcije iz standardne Rast biblioteke **TcpStream::connect** i **TcpStream** implementacije **write_all** i **read_to_string**
- Ove funkcije pozivaju druge funkcije, ali na kraju program realizuje i sistemske pozive: npr. otvori TCP konekciju, pročitaj ili zapiši neke podatke
- Tamna siva pokazuje kad sve program čeka na OS (u praksi, gotovo sve vreme prolazi u čekanju OS funkcija)
- Dok se čeka na OS, program (nit izvršavanja) je suštinski blokiran
- Kako kod može zauzimati poprilično memorije, to znači da je to parče memorije zaključano nizašta





SINRONO VS. ASINHRONO

- Tako da, čak i u paralelnoj implementaciji, imali bi veliki broj niti koje zauzimaju poprilično memorije, a čekaju blokirane od strane OS
- Rešenje?
- Napisati program tako, da niti izvršavanja rade nešto drugo dok se čeka na izvršavanje OS poziva
- Naravno, to se ne može realizovati kroz pozive ovih funkcija, jer su one po svojoj prirodi sinhrone

```
use std::io::prelude::*;
use std::net;

fn cheapo_request(host: &str, port: u16, path: &str)
                  -> std::io::Result<String>
{
    let mut socket = net::TcpStream::connect((host, port))?;

    let request = format!("GET {} HTTP/1.1\r\nHost: {}\r\n\r\n", path, host);
    socket.write_all(request.as_bytes())?;
    socket.shutdown(net::Shutdown::Write)?;

    let mut response = String::new();
    socket.read_to_string(&mut response)?;

    Ok(response)
}
```

FUTURE

- Podrška za asinhrono programiranje svodi se na reimplementaciju osobine **std::future::Future**:

```
trait Future {
    type Output;

    // For now, read `Pin<&mut Self>` as `&mut Self`.
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}

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

- **Future** predstavlja operaciju koja se može proveravati da li se izvršila (**test for completion**)
- Metoda **poll** nikada ne čeka da se operacija izvrši, ona odmah vraća rezultat

FUTURE

- Ako je operacija izvršena, metoda **poll** vraća **Poll::Ready(output)**, gde je **output** finalni rezultat

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

- U suprotnom, **poll** vraća **Pending**
- Ako se ikada desi da je sigurno i u redu da ponovo povuče (**polling**), **poll** metoda se obavezuje da probudi metodu (**waker**) koja je prosleđena kroz parametar **Context**
- Ovo se zove pinjata model (**piñata model**) asinhronog programiranja, jer sve što može da se uradi sa **Future** procesom je da ga odalamite sa **poll** metodom dok vrednost ne otpadne
- Svi savremeni OS imaju odgovarajuće sistemske pozive (interfejs) za implementaciju **poll** modela

FUTURE

- Asinhrona implementacija **read_to_string** bi imala sledeće zaglavlje:

```
fn read_to_string(&mut self, buf: &mut String)
    -> impl Future<Output = Result<usize>>;
```

- **Jedina razlika je zapravo u povratnoj vrednosti funkcije**
- Asinhrona verzija vraća **Future** rezultata **Result<usize>**, bukvalno se može pročitati kao **obećavam ti rezultat u budućnosti** ili vraćam ti buduću vrednost rezultata (**impl** znači da funkcija implementira **Future**)
- Razlika postoji i u načinu pozivu ove metode, jer se ona sada mora pozivati kroz **poll** mehanizam, tj. upotrebom **poll** metode
- Ovaj **Future** se mora pozivati sve dok se ne dobije rezultat
- Svaki put kada se desi povlačenje (**poll**) **read_to_string** izvrši se onoliko koliko može
- Krajnji rezultat će vratiti ili uspešno pročitanu vrednost ili grešku

FUTURE

- Ovo je generalni obrazac da bi se neka metoda napravila asinhronom, svi argumenti ostaju isti kao kod sinhronog poziva, jedino se povratna vrednost transformiše u odgovarajući **Future**
- Poziv **read_to_string** metode neće ništa izvršiti, već će konstruisati i vratiti **Future** koji će odraditi pravi posao kada se povuče (**poll**)
- Ovaj **Future** mora sadržati sve što je neophodno da se realizuje budući poziv, u ovom konkretnom slučaju, mora da se zapamti ulazni tok iz kojeg se čitaju podaci i string na koji će se dodati pročitani podaci
- Zbog toga je pravilan zapis ovo funkcije onaj u kojem se naglašava da je životni vek **Future** jednak životnom veku ulaznog toka i stringa na koji se vrednost dodaje (**self** i **buf**) i ne može ih nadživeti

```
fn read_to_string<'a>(&'a mut self, buf: &'a mut String)
    -> impl Future<Output = Result<usize>> + 'a;
```

FUTURE

- Rast sanduk **async-std** implementira asinhrone verzije svih **std I/O** funkcionalnosti uključujući i **Read** osobinu, pa samim tim i metodu **read_to_string**
- **async-std** pažljivo prati dizajn **std**, pa koristi (**reusing**) tipove, greške, rezultate, mrežne adrese i gotovo sve druge podatke kompatibilne i u asinhronom i sinhronom prostoru
- Pravilo **Future** osobine je da jednom kada se desi **Poll::Ready**, više neće doći do pozivanja (**it will never be polled again**)
- Neke implementacije će vratiti **Poll::Pending** ako su prozvane (**overpolled**) nakon što su vratile **Poll::Ready**, neke će paničiti ili zauvek visiti (**hang**)
  - Naravno neće dozvoliti greške u memoriji ili ugroziti sigurnost niti
- Rast ima mehanizam pomoću kojeg sve i dalje drži jednostavnim, jer **poll** mehanizam može da postane prekompleksan

ASYNC FUNKCIJE I AWAIT IZRAZI

- Asinhrona verzija **cheapo_request** metode:

```
use async_std::io::prelude::*;
use async_std::net;

async fn cheapo_request(host: &str, port: u16, path: &str)
                                -> std::io::Result<String>
{
    let mut socket = net::TcpStream::connect((host, port)).await?;

    let request = format!("GET {} HTTP/1.1\r\nHost: {}\r\n\r\n", path, host);

    socket.write_all(request.as_bytes()).await?;
    socket.shutdown(net::Shutdown::Write)?;

    let mut response = String::new();
    socket.read_to_string(&mut response).await?;

    Ok(response)
}
```

ASYNC FUNKCIJE I AWAIT IZRAZI

- Nije velika razlika u kodu (svega tri čekanja)
- Funkcija počinje naredbom **async fn** umesto **fn**
- Koristi se asinhrone verzije **TcpStream::connect**, **write_all** i **read_to_string** iz **async_std**; sve vraćaju **Future** kao rezultat
- Nakon svakog poziva koji vraća **Future** kod uvek poziva **.await**
  - iako liči kao referenca na polje strukture koja se zove **await**, to je **deo sintakse** ugrađene u Rast jezik koji specifikuje čekanje dok **Future** nije spreman
  - **await** izraz evaluira finalni rezultat koji vraća **Future**
  - Na ovaj način funkcija dobija rezultat iz **connect**, **write_all** i **read_to_string**
- Za razliku od običnih funkcija kada se poziva asinhrona funkcija, kod se ne zaustavlja, već se funkcija vraća odmah (pre izvršavanja svog tela) i kod nastavlja dalje

ASYNC FUNKCIJE I AWAIT IZRAZI

- Kako funkcija vraća odmah (pre izvršavanja svog tela), konačni rezultat nije izračunat, već se vraća **Future** za kasniju upotrebu
- Ako se kod **cheapo_request** funkcije pozove na sledeći način:

```
let response = cheapo_request(host, port, path);
```

- Povratna vrednost funkcije će biti **Future** od **std::io::Result<String>**, a telo **cheapo_request** funkcije nije ni započelo sa izvršavanjem
- Nije potrebno prilagoditi tip povratne vrednosti asinhrone funkcije, Rast automatski tretira **async fn f(...) -> T** kao funkciju koja vraća **Future** tipa **T**, a ne direktno **T**
- **Future** koji vraća asinhrona funkcija obuhvata sve informacije potrebne za izvršavanje tela funkcije: argumente funkcije, prostor za lokalne promenljive i ostalo
- Tako da u primeru, povratna vrednost sadrži i vrednosti prosleđene kroz **host**, **port** i **path** argumente (jer su potrebno telu funkcije)

ASYNC FUNKCIJE I AWAIT IZRAZI

- Specifični **Future** tip za datu situaciju generiše kompajler automatski bazirano na telu funkcije i argumentima
- Tip nema ime, sve što programer zna o njemu je da implementira **Future<Output=R>**, gde je **R tip** povratne vrednosti asinhrone funkcije
- **Future** asinhrone funkcije liči na **closures** (u smislu anonimnosti)
- Kada se prvi put prozove **Future** (**poll the future**) koji vrati **cheapo_request**, izvršavanje započinje od početka tela funkcije i izvršava se do prvog **await** dela **Future-a** koji vraća **TcpStream::connect**
- **await** izraz proziva povezani **Future** i ako on nije spreman, vraća **Poll::Pending** svom pozivaocu: **cheapo_request Future** ne može da nastavi delje preko prvog **await** poziva sve dok prozivanje **TcpStream::connect Future-a** ne vrati **Poll::Ready**

ASYNC FUNKCIJE I AWAIT IZRAZI

- Kako to izgleda u suštini za **TcpStream::connect(...).await**

```
{
    // Note: this is pseudocode, not valid Rust
    let connect_future = TcpStream::connect(...);
    'retry_point:
    match connect_future.poll(cx) {
        Poll::Ready(value) => value,
        Poll::Pending => {
            // Arrange for the next `poll` of `cheapo_request`'s
            // future to resume execution at 'retry_point.
            ...
            return Poll::Pending;
        }
    }
}
```

- U osnovi **await** izraz preuzima vlasništvo nad **Future-om** i onda ga proziva

ASYNC FUNKCIJE I AWAIT IZRAZI

- U osnovi **await** izraz preuzima vlasništvo nad **Future-om** i onda ga proziva
- Ako je spreman, onda je finalna vrednost **await** izraza i nastavlja se sa izvršavanjem pozivaoca, a ako nije, onda vraća **Poll::Pending** pozivaocu
- Ono što je ovde važno, je da sledeće prozivanje **Future-a** vezanog za **cheapo_request** funkciju, ne kreće od početka tela funkcije, već nastavlja (**resumes**) sa mesta na kom je prozvan **Future** koji pripada **connect** metodi
- Ne nastavlja se dalje dok **Future** koji pripada **connect** metodinije spreman
- Kako se proziva **Future** vezan za **cheapo_request** funkciju, tako se postupno prolazi kroz telo funkcije od jednog do drugo **await** izraza, prelazeći dalje tek kad je odgovarajući **pod-Future** spreman

ASYNC FUNKCIJE I AWAIT IZRAZI

- Koliko će se puta **cheapo_request** funkcija prozivati zavisi od broja i ponašanja **pod-Future-a** i toka same funkcije
- **cheapo_request Future** prati kad treba da se desi sledeće prozivanje i stanje lokalnih promenljivih, argumenata, kao i privremenih promenljivih a koje se koriste za nastavak rada tela funkcije
- Mogućnost da se suspenduje izvršavanje funkcije i potom nastavi njeno izvršavanje je **jedinstveno za asinhrone funkcije** – kada se sinhrona funkcija završi, njen stek se briše, dok se kod asinhronih funkcija čuva
- Ipak, kako se **await** izraz oslanja na mogućnost da nastavi nad sačuvanim stanjem steka, ono se može koristiti samo sa asinhronim funkcijama, tj. unutar asinhronih funkcija
- Za sada Rast ne dozvoljava osobinama da imaju asinhrone metode, samo slobodne funkcije i funkcije koje implementiraju specifične ugrađene osobine koje mogu biti asinhrone

ASYNC FUNKCIJE I AWAIT IZRAZI

- Iako ona sama može da čeka finalne vrednosti svojih **Future-a**, asinhrona funkcija, u osnovi, prosleđuje vrednosti svih **Future-a** koji su u njoj, dalje svom pozivaocu
- Ultimativno, **neko mora da čeka finalnu vrednost** (mora da sačeka sve te **await** pozive)
- Asinhrona funkcija se može pozivati iz sinhrone funkcije upotrebom **task::block_on** funkcije iz **async_std**

```
fn main() -> std::io::Result<()> {
    use async_std::task;

    let response = task::block_on(cheapo_request("example.com", 80, "/"))?;
    println!("{}", response);
    Ok(())
}
```

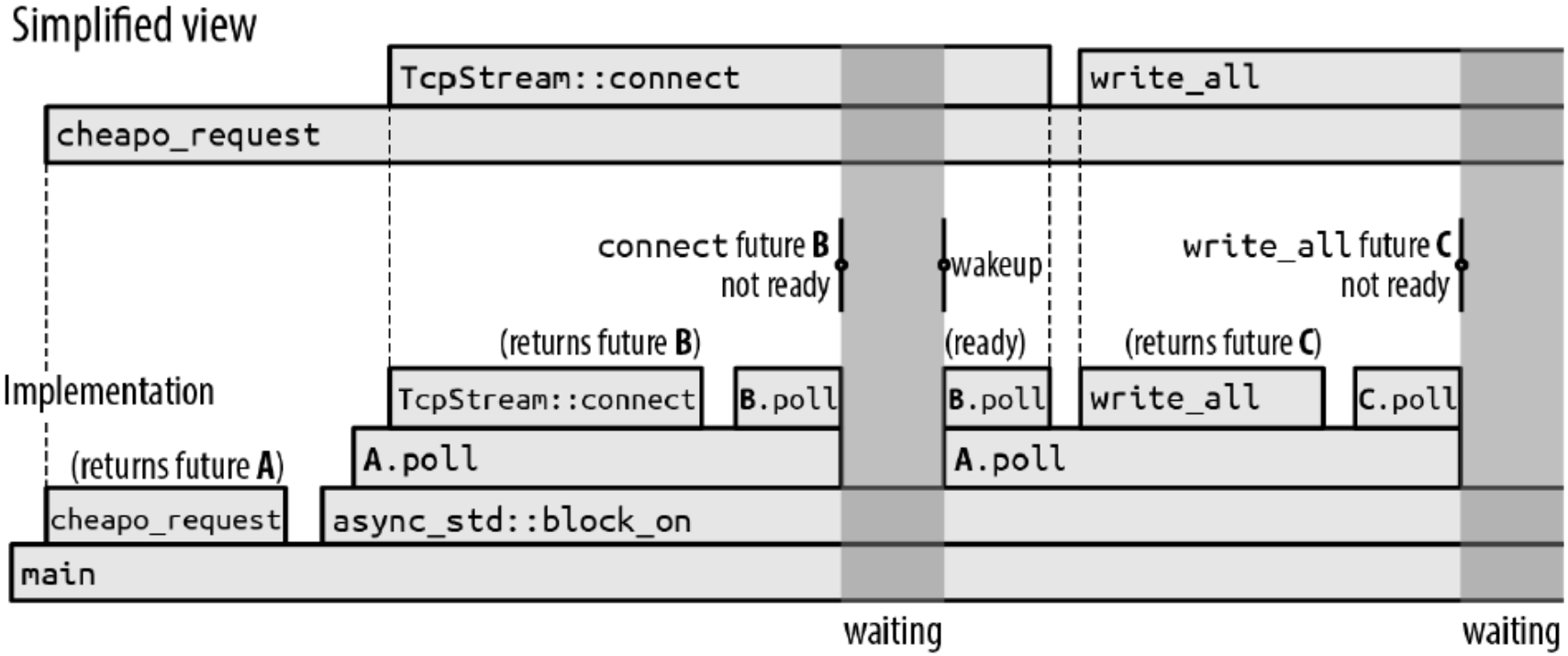
- **task::block_on** funkcija preuzime **Future** i proziva je sve dok ne proizvede vrednost

ASYNC FUNKCIJE I AWAIT IZRAZI

- **task::block_on** funkcija se može smatrati ako adapter između asinhronog i sinhronog sveta
- Ipak, **sama funkcija ima blokirajući karakter**, što znači da se ne može (tačnije, ne bi trebalo) koristiti unutar neke asinhrone funkcije, jer će blokirati izvršavanje cele niti dok vrednost nije spremna
- Ako vam treba, onda je najbolje koristi **await** u asinhronim funkcijama
- Jedan mogući način izvršavanja dat je na sledećem slajdu
- On predstavlja pojednostavljen pogled na mejn i apstraktan pogled na asinhrone pozive
- Funkcija **cheapo_request** prvo poziva **TcpStream::connect** kako bi dobila soket (**socket**) a onda poziva **write_all** i **read_to_string** nad tim soketom, zatim izlazi
- Ovo vrlo liči na sinhronu verziju

ASYNC FUNKCIJE I AWAIT IZRAZI

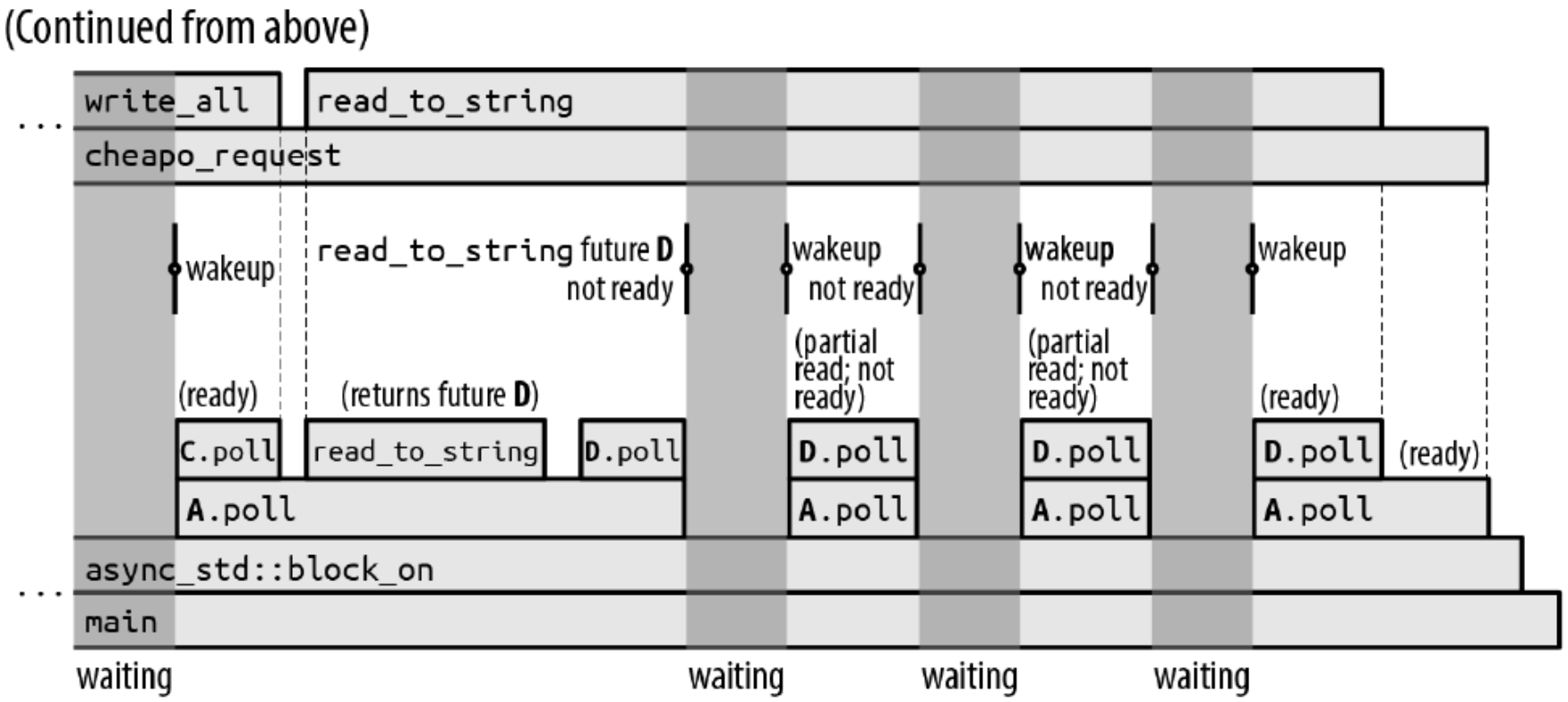
- Jedan mogući način izvršavanja:



- koji se nastavlja na sledećem slajdu

ASYNC FUNKCIJE I AWAIT IZRAZI

- Nastavak izvršavanja:



- Pozivi asinhronih metoda i samo izvršavanje se sastoji od nekoliko koraka

ASYNC FUNKCIJE I AWAIT IZRAZI

- Prvo, **main** poziva **cheapo_request**, koja vraće **future A** kao svoj finalni rezultat, dok **main** prosleđuje **future A async_std::block_on** funkciji koja proziva taj **future**
- Prozivanje **future A** omogućuje izvršavanje tela **cheapo_request** funkcije, koja poziva **TcpStream::connect** i dobavlja **future B** sa soketom i čeka na to (zapravo kako **TcpStream::connect** može da naiđe na grešku, **B** je **future** od **Result<TcpStream, std::io::Error>**
- **Future B** proziva **await** (kako mrežna konekcija još uvek nije uspostavljena, **B.poll** će vratiti **Poll::Pending**, ali će urediti da se pozivajući zadatak probudi jednom kad je soket spreman)
- Kako **future B nije** spreman, **A.poll** vraća **Poll::Pending** svom pozivaocu, **block_on**
- Kako **block_on** nema druga posla, on se uspavljuje, a cela nit je sada blokirana

ASYNC FUNKCIJE I AWAIT IZRAZI

- Kada je konekcija **future-a B** spremna sa korišćenje, budi se zadatak koji ju je prozvao, što dovodi do buđenja **block_on** funkcije koja ponovo pokuša da prozove **future A**
- Prozivanje **A** dovodi do toga da **cheapo_request** nastavi dalje od počev od prvog **await-a**, gde se sada ponovo proziva **B**
- Ovog puta **B** je spremno: kreiranje soketa je završeno, tako da ovo prozivanje (**A.poll**) vraća **Poll::Ready(Ok(socket))**
- Asinhroni poziv **TcpStream::connect** je kompletirano, vrednost izraza **TcpStream::connect(...).await** je **Ok(socket)**
- Izvršavanje tela **cheapo_request** nastavlja normalnim tokom, gradi se request string upotrebom **format!** makroa i to se prosleđuje **socket.write_all**
- Kako je i **socket.write_all** asinhrona funkcija, njen poziv vraća **future C** kao rezultat koji će **cheapo_request** opet da sačeka (**await**)

ASYNC FUNKCIJE I AWAIT IZRAZI

- Procedura za **future C** je slična kao i za **future B**
- Future nastao pozivom **socket.read_to_string** će biti prozvan čak četiri puta jer svaki poziv čita nešto od podataka iz soketa koji se čekaju a koji se moraju svi iščitati i to traje nekoliko operacija
- Nije teško napisati petlju koja realizuje prozivanje (**poll** pozive), ali za razliku od običnog koda **sync_std::task::block_on** zna kada da se uspava, odnosno neće uradili ogroman broj **poll** poziva

```
use async_std::io::prelude::*;
use async_std::net;

async fn cheapo_request(host: &str, port: u16, path: &str)
                                -> std::io::Result<String>
{
    let mut socket = net::TcpStream::connect((host, port)).await?;

    let request = format!("GET {} HTTP/1.1\r\nHost: {}\r\n\r\n", path, host);
    socket.write_all(request.as_bytes()).await?;
    socket.shutdown(net::Shutdown::Write)?;

    let mut response = String::new();
    socket.read_to_string(&mut response).await?;

    Ok(response)
}
```

GENERISANJE ASYNC ZADATAKA

- Ograničenje **async_std::task::block_on** je što blokira nit u kojoj se poziva
- Ideja je da nit kada naiđe na posao koji blokira, preskoči na nešto drugo što može da radi
- Za to se može iskoristiti **async_std::task::spawn_local** funkcija
- Ona uzima **Future** i dodaje ga u **pul** (**pool**) odakle će ga **block_on** funkcija prozvati
- Tako da ako se tamo prosledi gomila **future-a block_on** će ih prozivati konkurentno sve dok finalni rezultat nije spreman (tj. dok nisu svi završili svoj posao)
- **spawn_local** je dostupna unutar **async-std** samo ako se eksplicitno omogući

```
async-std = { version = "1", features = ["unstable"] }
```

GENERISANJE ASYNC ZADATAKA

- **spawn_local** je asinhrona implementacija standardne funkcije **spawn**
- **std::thread::spawn(c)** će uzeti **closure c** i započeti nit a vratiće **std::thread::JoinHandle** čija join metoda čeka da se nit izvrši i vraća šta god da je **c** vratila
- **async_std::task::spawn_local(f)** će uzeti **future f** i dodati ga pulu (**pool**) odakle će biti prozvan kada trenutna nit pozove **block_on**; **spawn_local** će vratiti **async_std::task::JoinHandle** tip, koji je sam po sebi **future** koji se koristi kako bi se preuzela finalna vrednost od **f**
- Voditi računa da u ovom slučaju asinhrone funkcije moraju živeti onoliko dugo koliko i parametri koji se prosleđuju (tako da se ne mogu tek tako prosleđivati reference i vršiti pozajmljivanje)
- Npr. sledeći kod će izazvati grešku

GENERISANJE ASYNC ZADATAKA

- Primer koji uključuje metodu koja obrađuje više HTTP zahteva konkurentno

```
pub async fn many_requests(requests: Vec<(String, u16, String)>)
                           -> Vec<std::io::Result<String>>
{
    use async_std::task;

    let mut handles = vec![];
    for (host, port, path) in requests {
        handles.push(task::spawn_local(cheapo_request(&host, port, &path)));
    }

    let mut results = vec![];
    for handle in handles {
        results.push(handle.await);
    }

    results
}
```

GENERISANJE ASYNC ZADATAKA

- Funkcija poziva **cheapo_request** nad svakim elementom requests, prosleđuje **future** svakog poziv u **spawn_local**
- Rezultujući **JoinHandles** se smeštaju u vektor i čeka se **await** izraz svakog od njih
- Svejedno je u kom se redosledu čekaju, kako je svaki zahtev već prozvan, njihovi **future-i** će biti prozvani kadgod nit poziva **block_on** i kada nema šta bolje da radi
- Svi zahtevi se izvršavaju konkurentno
- Kada svi završe, **many_requests** će vratiti finalni rezultat pozivaocu
- Ipak **borrow checker** će ovda baciti grešku za **host** i **path**, jer **future** kojem su dodeljene, ne sme da nadživi ove reference
- Način da se ovo reši je da se kreira asinhrona funkcija koja će da preuzme te vrednosti u svoje vlasništvo

GENERISANJE ASYNC ZADATAKA

- Način da se ovo reši je da se kreira asinhrona funkcija koja će da preuzme te vrednosti u svoje vlasništvo

```
async fn cheapo_owning_request(host: String, port: u16, path: String)
                                    -> std::io::Result<String> {
    cheapo_request(&host, port, &path).await
}
```

- Sada je moguće generisati sve zahteve upotrebom funkcije **cheapo_owning_request**:

```
for (host, port, path) in requests {
    handles.push(task::spawn_local(cheapo_owning_request(host, port, path)));
}
```

- Sama **many_requests** se može pozivati iz sinhronizovane **main** funkcije sa **block_on** metodom

GENERISANJE ASYNC ZADATAKA

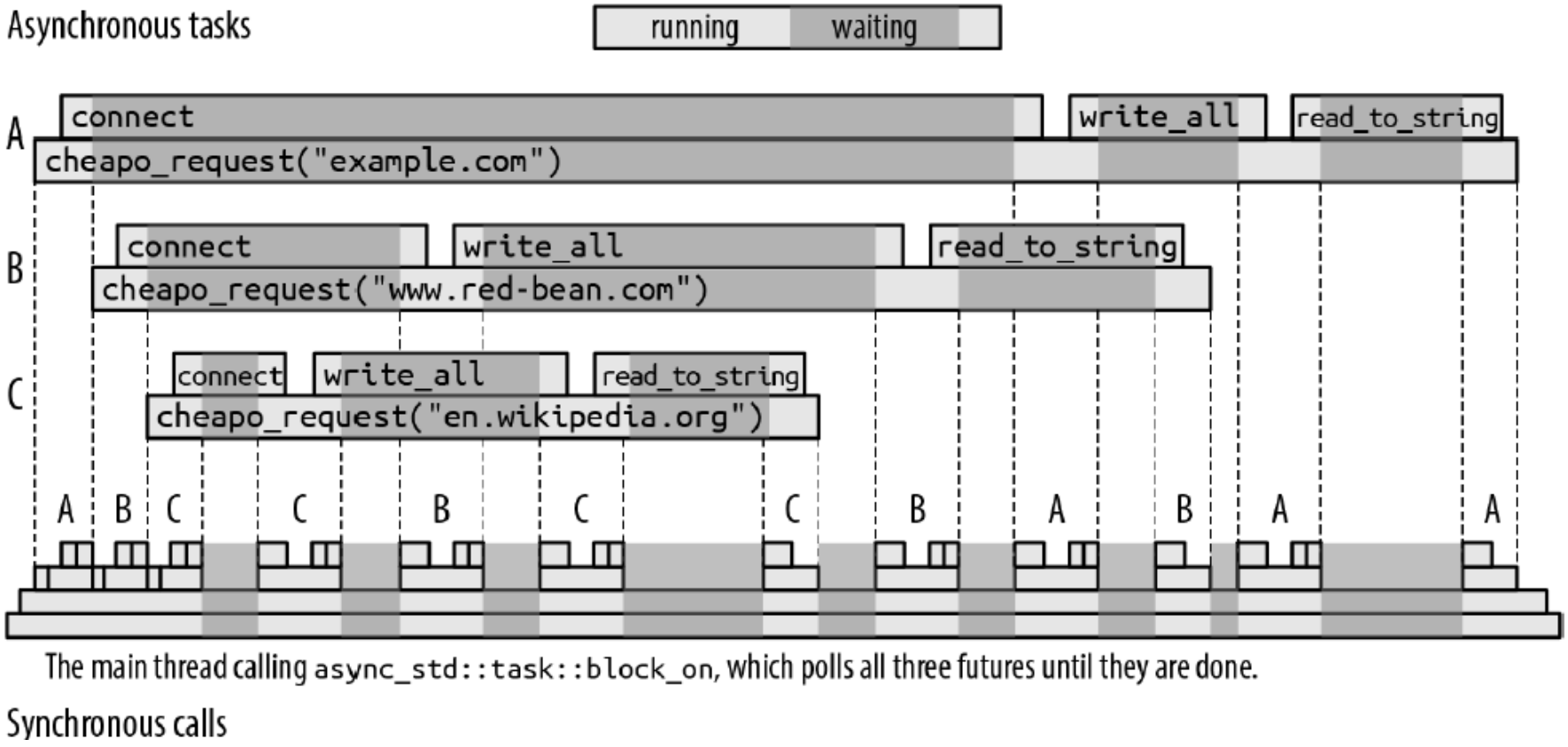
- Sama **many_requests** se može pozivati iz sinhronizovane main funkcije sa **block_on** metodom

```
let requests = vec![
    ("example.com".to_string(),      80, "/".to_string()),
    ("www.red-bean.com".to_string(), 80, "/".to_string()),
    ("en.wikipedia.org".to_string(), 80, "/".to_string()),
];

let results = async_std::task::block_on(many_requests(requests));
for result in results {
    match result {
        Ok(response) => println!("{}", response),
        Err(err) => eprintln!("error: {}", err),
    }
}
```

- Ovaj kod izvršava sva tri zadatka konkurentno unutar jednog **block_on**
- Svaki od njih se izvršava kada se ukaže prilika, dok su drugi blokirani

GENERISANJE ASYNC ZADATAKA

- Primer kako bi se prethodni kod mogao izvršavati:



- Poziv **many_requests** se zbog jednostavnosti ne prikazuje

GENERISANJE ASYNC ZADATAKA

- Poziv **many_requests** je stvorio (**spawned**) tri asinhrona zadatka, označeni kao **A**, **B**, **C**
- **block_on** počinje prozivanjem **future A,** koji započinje povezivanje sa (**connecting to**) **example.com**
- Čim **future A** vrati **Poll::Pending**, **block_on** se prebacuje na sledeći generisani zadatak i proziva **future B** i započinje povezivanje sa njegovim serverom
- Zatim se prebacuje na **future C** i započinje povezivanje sa njegovim serverom
- Kada svi **future-i** vrate **Poll::Pending**, **block_on** se uspavljuje sve dok jedan od **TcpStream::connect future-a** ne signalizira da se njegov zadatak može ponovo prozvati

GENERISANJE ASYNC ZADATAKA

- U primeru, najbrže se ostvarila konekcija ka **en.wikipedia.org**, tako da će zadatak koji odgovara **future c** završiti prvi
- Kada je zadatak završio, on beleži svoj rezultat u njegovom **JoinHandle** i označava da je spreman, tako da **many_requests** može da nastavi dalje
- Eventualno će i drug pozivi **cheapo_request** ili završiti uspešno ili vratiti grešku i sama funkcija **many_requests** može da završi i vrati rezultat
- Konačno, **main** dobija vektor sa rezultatima iz **block_on**
- Obratite pažnju da se ovo sve dešava nad jednom niti i da su tri poziva **cheapo_request** pomešana kroz uspešna prozivanja njihovih **future-a**
- Prebacivanje između asinhronih zadataka se dešava samo na mestima gde se pojavljuje **await** izraz i kada future koji se proziva vrati **Poll::Pending** (ako funkcija traje dugo, može da uzurpira vreme ostalih)

ASYNC BLOCKS

- Pored asinhronih funkcija, Rast podržava i asinhrone blokove
- Dok standardni blok vraća vrednost poslednjeg izraza ili (), asinhroni blok vraća **future** vrednosti poslednjeg izraza
- To znači da se **await** izraz može koristiti sa asinhronim blokom
- Asinhroni blok izgleda kao običan, jedino što mu prethodi ključna reč **async**

```
let serve_one = async {
    use async_std::net;

    // Listen for connections, and accept one.
    let listener = net::TcpListener::bind("localhost:8087").await?;
    let (mut socket, _addr) = listener.accept().await?;

    // Talk to client on `socket`.
    ...
};
```

ASYNC BLOCKS

- Ovaj kod će inicijalizovati **future serve_one** koji kada se proziva opslužuje jednu TCP konekciju

```
let serve_one = async {
    use async_std::net;

    // Listen for connections, and accept one.
    let listener = net::TcpListener::bind("localhost:8087").await?;
    let (mut socket, _addr) = listener.accept().await?;

    // Talk to client on `socket`.
    ...
};
```

- Blok neće krenuti sa izvršavanjem sve dok se ne prozove (**poll**)
- Ako se iza asinhronog bloka stavi operator **?**, u slučaju greške, neće doći do povratka iz funkcije koja okružuje blok, već samo do izlaska iz bloka, što znači da će **?** vratiti finalnu verziju **serve_one** bloka
- Slično, ako se zapiše **return** u asinhronom bloku, ono neće izaći iz funkcije, već će samo vratiti finalnu vrednost bloka

ASYNC BLOCKS

- Ako blok koristi promenljive iz okruženja van bloka, onda on hvata njihove vrednosti, slično kao **closure**, i pravi njihove **future**
- Takođe, slično kao i **closure**, ako se napiše **async move** u definiciji bloka, doći će do preuzimanja vlasništva svih tih promenljivih
- Asinhroni blokovi predstavljaju mehanizam da se na koncizan način napiši asinhroni delove koda, kao i da se oni na vrlo vidljiv način odvoje od ostatka koda
- Isti efekat koji se dobio prebacivanjem koda u zasebnu asinhronu funkciju kako bi se preuzelo vlasništvo nad promenljivama:

```
async fn cheapo_owning_request(host: String, port: u16, path: String)
                                    -> std::io::Result<String> {
    cheapo_request(&host, port, &path).await
}
```

- može se ostvariti i primenom asinhronog bloka, tako što se metoda pozove unutar asinhronog bloka koji je preuzeo vlasništvo nad promenljivama

ASYNC BLOCKS

- Poziv asinhrone funkcije iz asinhronog bloka:

```
pub async fn many_requests(requests: Vec<(String, u16, String)>)
                           -> Vec<std::io::Result<String>>
{
    use async_std::task;

    let mut handles = vec![];
    for (host, port, path) in requests {
        handles.push(task::spawn_local(async move {
            cheapo_request(&host, port, &path).await
        }));
    }
    ...
}
```

- Kako je ovde sada u pitanju **async move** blok, njen future je preuzeo vlasništvo nad **String** vrednostima **host** i **path**, a zatim preneo njihove reference **cheapo_request** funkciji
- Ovde sada nema problema sa vlasništvom

ASYNC BLOCKS

- Jedan nedostatak je što se za asinhroni blok ne može definisati tip povratne vrednosti (ne postoji sintaksa za to), to može izazvati određene probleme kada se koristi **?** operator

```
let input = async_std::io::stdin();
let future = async {
    let mut line = String::new();

    // This returns `std::io::Result<usize>`.
    input.read_line(&mut line).await?;

    println!("Read line: {}", line);

    Ok(())
};
```

- Na prvi pogled nije jasno šta je problem, ali kako se iz bloka može izaći i na kraju bloka ali i na mestu na kome se nalazi **await**, blok može vratiti dve različite vrednosti (čiji se tip razlikuje)

ASYNC BLOCKS

- Iz bloka se može izaći i na kraju bloka ali i na mestu na kome se nalazi **await**, blok može vratiti dve različite vrednosti (čiji se tip razlikuje)

```
error: type annotations needed
   |
42 |     let future = async {
   |         ------ consider giving `future` a type
...
46 |         input.read_line(&mut line).await?;
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ cannot infer type
```

- Metoda **read_line** vraća vrednost **Result<(), std::io:Error>**, a **?** operator koristi **From** osobinu da pretvori povratnu vrednost u bilo koji tip koji se očekuje i može vratiti bilo koji **E** za **Result<(), E>**, ali ovde ne zna koji se tip **E** očekuje, pa baca grešku
- Pretpostavlja se da će naredne iteracije Rasta ispraviti ovo i dodati podršku u samoj sintaksi
- Za sada postoji način da se ovo popravi

ASYNC BLOCKS

- Način da se nedostatak specifikacije tipa asinhronog bloka popravi je da se eksplicitno navede tip povratne vrednosti samog bloka

```
let future = async {
    ...
    Ok::<(), std::io::Error>(())
};
```

- Kako je **Result** generički tip koji očekuje **success** i **error** tipove kao svoje parametre, ti parametri se mogu definisati upotrebom **Ok** ili **Err** kako je gore pokazano

ASYNC FUNKCIJE IZ ASYNC BLOKOVA

- Asinhroni blokovi omogućuju kreiranje asinhroni funkcija, ali na način koji je fleksibilniji u odnosu na standardni način definisanja funkcije
- Zašto je fleksibilniji?
- Pogledajte reimplementaciju **cheapo_request** as asinhronim blokom

```
use std::io;
use std::future::Future;

fn cheapo_request<'a>(host: &'a str, port: u16, path: &'a str)
    -> impl Future<Output = io::Result<String>> + 'a
{
    async move {
        ... function body ...
    }
}
```

- Ova funkcija odmah po svom završetku vraća **future** isto kao i originalna implementacija, jedino što mora eksplicitno da se napiše **impl Future** da se označi da to funkcija implementira

ASYNC FUNKCIJE IZ ASYNC BLOKOVA

- Osim toga sa **impl Future**, iz ugla pozivaoca ova funkcija se ponaša kao asinhrona, ali sa jednom bitnom razlikom
- Deo funkcije koji se ne nalazi u asinhronom bloku se izvršava odmah, ne mora da se čeka da telo funkcije dođe na red da bude prozvano

```
fn cheapo_request(host: &str, port: u16, path: &str)
    -> impl Future<Output = io::Result<String>> + 'static
{
    let host = host.to_string();
    let path = path.to_string();

    async move {
        ... use &*host, port, and path ...
    }
}
```

- Asinhroni blok hvata sve promenljive i preuzima vlasništvo, nema problema sa pozajmljivanjem i životnim vekom promenljivih, te se može koristiti sa **spawn_local**

ASYNC FUNKCIJE IZ ASYNC BLOKOVA

- Može se koristiti sa **spawn_local**

```
let join_handle = async_std::task::spawn_local(
    cheapo_request("areweasyncyet.rs", 80, "/")
);

... other work ...

let response = join_handle.await?;
```

ASINHRONI ZADACI I PARALELNO IZVRŠAVNJE

- Asinhroni zadaci se mogu tako organizovati, da se njihov izvršavanja preraspodeli nad više niti
- Oni se organizuju u kolekcije radnih niti (**pool of worker threads**) koje izvlače spremne **future** i konkurentno ih obrađuju, tako da ako se desi da su dva ili više **future-a** spremni i ako ima slobodnih niti, ne moraju da se čekaju
- Za kreiranje kolekcije radnih niti koristi se **async_std::task::spawn**

```
use async_std::task;

let mut handles = vec![];
for (host, port, path) in requests {
    handles.push(task::spawn(async move {
        cheapo_request(&host, port, &path).await
    }));
}
...
```

ASINHRONI ZADACI I PARALELNO IZVRŠAVNJE

- Poput **spawn_local**, **spawn** vraća **JoinHandle** vrednost koja čeka (**await**) da dobije finalnu vrednost nekog **future-a**
- Za razliku od **spawn_local**, **future** ne mora da čeka da programer pozove **block_on** da bi bila prozvana, već čim je neka nit iz kolekcije spremna pokušaće da je prozove
- U praksi se **async_std::task::spawn** koristi daleko više nego **spawn_local**, jer se ovako blokiranje i resursi balansirano koriste
- Ono o čemu treba da se vodi računa kada se koristi **spawn** je da sistem teži da kolekcija radnih niti bude stalno aktivna, što znači da će **future** biti prozvan od strane prve radne niti koja je slobodna
- Tako da neki future može započeti u jednoj niti, doći do mesta koje ga blokira, uspava se, a da sledeći put kada je spreman, taj **future** preuzme neka druga nit

SEND I SPAWN

- Jedno ograničenje upotrebe **spawn** funkcije je da sada **future** mora implementirati **Send** osobinu
- Pričali smo o **Send** osobini kada smo radili konkurentno programiranje
- **Future** je **Send** samo ako su i vrednosti koje sadrži **Send**: svi argumenti funkcije, lokalne promenljive, pa čak i privremene vrednosti moraju biti bezbedne za pomeranje u drugu nit (**safe to move to another thread**)
- Ovo može vrlo lako da izazove grešku, npr:

```
use async_std::task;
use std::rc::Rc;

async fn reluctant() -> String {
    let string = Rc::new("ref-counted string".to_string());

    some_asynchronous_thing().await;

    format!("Your splendid string: {}", string)
}
```

SEND I SPAWN

- Izaziva sledeću grešku:

```
error: future cannot be sent between threads safely
    |
17  |     task::spawn(reluctant());
    |     ^^^^^^^^^^ future returned by `reluctant` is not `Send`
    |


    |
127 | T: Future + Send + 'static,
    |             ---- required by this bound in `async_std::task::spawn`
    |
    = help: within `impl Future`, the trait `Send` is not implemented
            for `Rc<String>`
note: future is not `Send` as this value is used across an await
    |
10  |         let string = Rc::new("ref-counted string".to_string());
    |             ------ has type `Rc<String>` which is not `Send`
11  |
12  |         some_asynchronous_thing().await;
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                    await occurs here, with `string` maybe used later
...
15  |     }
    |     - `string` is later dropped here
```

SEND I SPAWN

- Problem se može rešiti na dva načina
- Jedan je da se ograniči opseg promenljive koja ne implementira **Send** i da ona nije u dosegu **await** izraza, pa se samim tim ne mora čuvati za kasniju upotrebu (unutar **future**)

```
async fn reluctant() -> String {
    let return_value = {
        let string = Rc::new("ref-counted string".to_string());
        format!("Your splendid string: {}", string)
        // The `Rc<String>` goes out of scope here...
    };

    // ... and thus is not around when we suspend here.
    some_asynchronous_thing().await;

    return_value
}
```

- Drugo rešenje je da se umesto **Rc** upotrebi **std::sync::Arc** (koja koristi atomične operacija koje su sigurne za konkurentni rad)

YIELD_NOW I SPAWN_BLOCKING

- Da bi čitav sistem funkcionisao efikasno, potrebno je da je **future** takav da kada se prozove brzo završi svoj posao i što brže omogući prozivanje drugih **future**-a
- Naravno, ovo nije uvek moguće, te procesi koji dugo traju, mogu blokirati ostale spremne **future**
- Jedan način da se ovo izbegne je da se namesti da se **future** proziva samo povremeno, tj. ređe od ostalih
- Za to se koristi **async_std::task::yield_now**

```
while computation_not_done() {
    ... do one medium-sized step of computation ...
    async_std::task::yield_now().await;
}
```

- Prvi put kada se **yield_now future** prozove, ono vraća **Poll::Pending**, koje kaže da vredi da se ubrzo ponovo prozove

YIELD_NOW I SPAWN_BLOCKING

- Efekat **yield_now** je da se asinhroni poziv odriče svog prava da se izvrši u korist nekih drugih future-a, ali da će se ono svakako uskoro ponovo prozvati
- Sledeći put kada se **yield_now** prozove, ono će vratiti **Poll::Ready(())**
- Ovaj pristup ne radi uvek, moguće je da se koristi kod koji je manje prilagođen asinhronom izvršavanju (tuđe biblioteke npr. ili nešto što je suviše komplikovano menjati), u tim situacijama se može koristiti **async_std::task::spawn_blocking**
- U ovoj kombinaciji funkcija uzima **closure**, prebacuje ga na zasebnu nit i tu pokreće, a vraća **future** od njegove povratne vrednosti – asinhroni kod može lepo da nastavi da radi svoj posao i čeka da ovaj proces kad tad završi, dok se kontrola da taj posao radi u asinhronom i paralelnom režimu prebacuje na OS

YIELD_NOW I SPAWN_BLOCKING

- Na primer, pretpostaviti da je potrebno proveriti vrednost prosleđene lozinke i uporediti je sa heširanim vrednostima u bazi podataka
- Zbog sigurnosti ova obrada mora biti kompleksna (**computationally intensive**)
- **argonautica** sanduk upravo pruža funkcije za ovaj posao, ali je potreban dobar deo sekunde da bi se ovo izračunalo i verifikovalo
- Primer je na narednom slajdu
- Kod će vratiti **Ok(true)** ako se lozinka poklapa (tj. njen ključ se poklapa sa heširanom vrednošću)
- Verifikacija se realizuje u closure funkciji koja se prosleđuje **spawn_blocking**, čime se kompleksno i vremenski dugo procesiranja prebacuje na sopstvenu nit izvršavanja, i time se obezbeđuje da ovo procesiranje neće uticati na druge korisničke zahteve

YIELD_NOW I SPAWN_BLOCKING

- **argonautica** primer

```
async fn verify_password(password: &str, hash: &str, key: &str)
                         -> Result<bool, argonautica::Error>
{
    // Make copies of the arguments, so the closure can be 'static.
    let password = password.to_string();
    let hash = hash.to_string();
    let key = key.to_string();

    async_std::task::spawn_blocking(move || {
        argonautica::Verifier::default()
            .with_hash(hash)
            .with_password(password)
            .with_secret_key(key)
            .verify()
    }).await
}
```

KAKO TO RADI U DRUGIM JEZICIMA

- Rast mehanizam za asinhrono programiranje jako liči na to kako to radi u drugim programskim jezicima
- Međutim prozivanje (**polling**) je implementirano malo drugačije
- Kod drugih programskim jezika asinhrone funkcije započinju odmah sa izvršavanjem (čim se pozovu) i postoji globalna petlja na nivou sistemske biblioteke čiji je zadatak da budi zadatke koje čekaju na neke vrednosti kada te vrednosti postanu dostupne
- Kod Rasta nema te globalne petlje, i funkcije počinju sa izvršavanjem i prozivanjem tek kad se eksplicitno pozove **block_on**, **spawn** ili **spawn_local** funkcije (**executors**) koje brinu o prozivanju i izvršavanju asinhronih funkcija do njihovog kraja
- Prednost je u tome, što imate fleksibilnost da po potrebi, napišete sami svoju **executor** funkciju (**tokio** sanduk) i onda po potrebi da koristite onu **executor** funkciju koja najviše odgovara potrebama vaše implementacije

ASINHRONI HTTP KLIJENT

- Primer jednostavne implementacije HTTP klijenta pomoću **surf** sanduka
- Prvo je potrebno uneti zavisnosti u **Cargo.toml** fajl

```
[dependencies]
async-std = "1.7"
surf = "1.0"
```

- Onda se definiše **many_requests**

```
pub async fn many_requests(urls: &[String])
                           -> Vec<Result<String, surf::Exception>>
{
    let client = surf::Client::new();

    let mut handles = vec![];
    for url in urls {
        let request = client.get(&url).recv_string();
        handles.push(async_std::task::spawn(request));
    }
```

ASINHRONI HTTP KLIJENT

- Onda se definiše **many_requests**

```
pub async fn many_requests(urls: &[String])
                                -> Vec<Result<String, surf::Exception>>
{
    let client = surf::Client::new();

    let mut handles = vec![];
    for url in urls {
        let request = client.get(&url).recv_string();
        handles.push(async_std::task::spawn(request));
    }

    let mut results = vec![];
    for handle in handles {
        results.push(handle.await);
    }
    results
}
```

ASINHRONI HTTP KLIJENT

- I sam **main**

```
fn main() {
    let requests = &["http://example.com".to_string(),
                     "https://www.red-bean.com".to_string(),
                     "https://en.wikipedia.org/wiki/Main_Page".to_string()];

    let results = async_std::task::block_on(many_requests(requests));
    for result in results {
        match result {
            Ok(response) => println!("*** {}\n", response),
            Err(err) => eprintln!("error: {}\n", err),
        }
    }
}
```

- Upotreba **surf::Client** omogućava da se jedna HTTP konekcija ka serveru upotrebi za sve zahteve ka tom serveru
- Nema potrebe za asinhronim blokom, jer **recv_string** je sama po sebi asinhrona metoda koja vraća **Send + 'static future**, te možemo taj future poslati direktno **spawn** metodi