Rust

Konkurentno programiranje

Konkurentno i paralelno programiranje

- Konkurentno - delovi programa se izvršavaju nezavisno
- Paralelno - različiti delovi programa se izvršavaju u isto vreme

Niti

- Podela programa na više niti kako bi se istovremeno izvršavalo više zadataka može da poboljša performanse, ali dodaje složenost.
- Pokreću se istovremeno, pa ne postoji garancija kojim redosledom će da se izvrše delovi vašeg programa. Ovo može da dovede do problema, kao što su:
  - Trka podataka - niti pristupaju podacima ili resursima u nedoslednom redosledu
  - Mrtve petlje - dve niti čekaju jedna drugu, sprečavajući obe niti da se nastave
  - Greške koje se dešavaju samo u određenim situacijama i koje je teško reprodukovati i pouzdano popraviti

Kreiranje niti

```rust
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
```

Join

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```

```
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 1 from the spawned thread!
hi number 3 from the main thread!
hi number 2 from the spawned thread!
hi number 4 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
```

Join

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

```
hi number 1 from the spawned thread!
hi number 2 from the spawned thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 3 from the main thread!
hi number 4 from the main thread!
```

Move i closure

- *Move* se koristi kako bi preneli vlasništvo nad vrednostima koje *closure* koristi iz okruženja u nit

Move i closure

```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

```
$ cargo run
   Compiling threads v0.1.0 (file:///projects/threads)
error[E0373]: closure may outlive the current function, but it borrows
 --> src/main.rs:6:32
  |
6 |     let handle = thread::spawn(|| {
  |                                ^^ may outlive borrowed value `v`
7 |         println!("Here's a vector: {:?}", v);
  |                                           - `v` is borrowed here
  |
note: function requires argument type to outlive `'static`
 --> src/main.rs:6:18
  |
6 |       let handle = thread::spawn(|| {
  |  __________________^
7 | |         println!("Here's a vector: {:?}", v);
8 | |     });
  | |______^
help: to force the closure to take ownership of `v` (and any other ref
  |
6 |     let handle = thread::spawn(move || {
  |                                ++++

For more information about this error, try `rustc --explain E0373`.
error: could not compile `threads` due to previous error
```

Move i closure

```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    drop(v); // oh no!

    handle.join().unwrap();
}
```

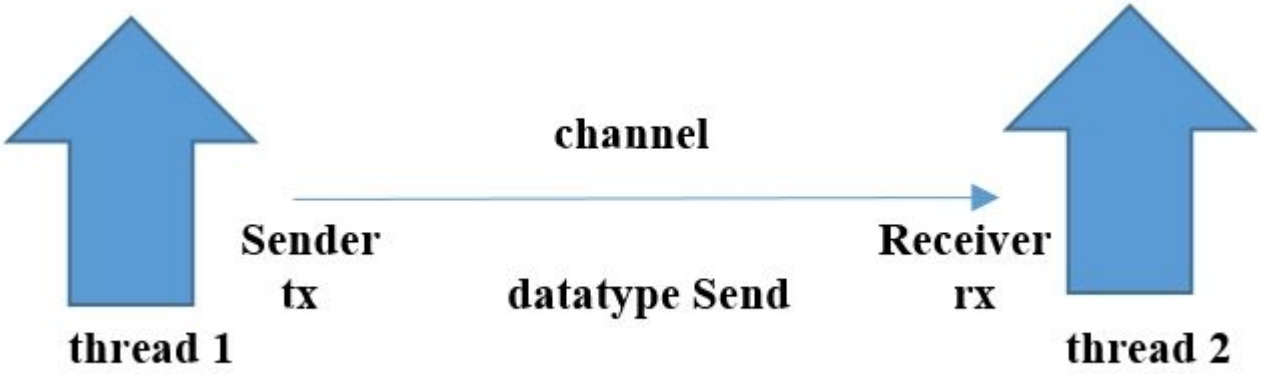
```
help: to force the closure to take ownership of `v` (and any other re
  |
6 |     let handle = thread::spawn(move || {
  |                                ++++
```

Move i closure

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

Slanje poruka za prenos podataka između niti

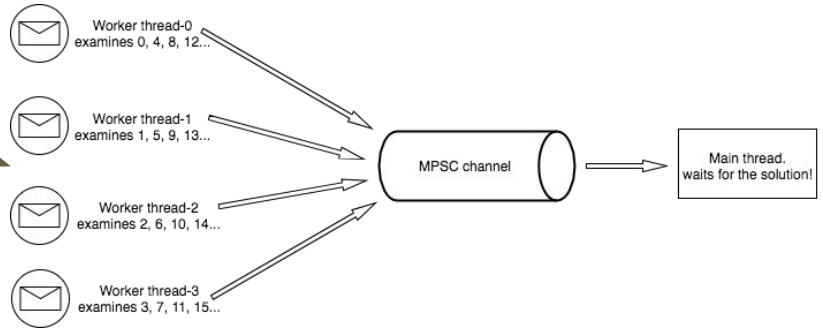
- Jedan od popularnijih pristupa obezbeđivanja bezbedne konkurentnosti
- Niti ili akteri komuniciraju tako što jedni drugima šalju poruke koje sadrže podatke
- Paralelno slanje poruka je moguće zbog implementacije ***kanala***



Kanali

```
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();
}
```

pošiljalac primalac



Kanali

```rust
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });
}
```

```rust
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

Kanali

- Prijemnik, ima dve metode:
  - *recv* - blokira izvršavanje glavne niti i čeka dok se vrednost ne pošalje niz kanal. Kada se vrednost pošalje, vratiće grešku da signalizira da vrednost više neće dolaziti.
  - *try_recv* - ne blokira nit, nego odmah vrati *Result<T, E>.* Korisna je kada niti ima drugog posla dok čeka na poruke.

Kanali i vlasništvo

```rust
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
        println!("val is {}", val);
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

```
$ cargo run
   Compiling message-passing v0.1.0 (file:///projects/message-passing)
error[E0382]: borrow of moved value: `val`
  --> src/main.rs:10:31
   |
8  |         let val = String::from("hi");
   |             --- move occurs because `val` has type `String`, which does not impleme
9  |         tx.send(val).unwrap();
   |                 --- value moved here
10 |         println!("val is {}", val);
   |                                ^^^ value borrowed here after move
   |
   = note: this error originates in the macro `$crate::format_args_nl` (in Nightly build

For more information about this error, try `rustc --explain E0382`.
error: could not compile `message-passing` due to previous error
```

Slanje više vrednosti i primaoc koji čeka

```rust
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
}
```

```
Got: hi
Got: from
Got: the
Got: thread
```

Kreiranje više pošiljalaca kloniranjem pošiljaoca

```
// --snip--

    let (tx, rx) = mpsc::channel();

    let tx1 = tx.clone();
    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx1.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    thread::spawn(move || {
        let vals = vec![
            String::from("more"),
            String::from("messages"),
            String::from("for"),
            String::from("you"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }

    // --snip--
```

```
Got: hi
Got: more
Got: from
Got: messages
Got: for
Got: the
Got: thread
Got: you
```

Deljeno stanje konkurentnosti

- Drugi metod za rukovanje paralelnošću - pristup istim deljenim resursima
- Deljenje memorije je kao višestruko vlasništvo - više niti može da pristupi istoj memorijskoj lokaciji u isto vreme.

Mutex

- Uzajamno isključivanje, dozvoljava samo jednoj niti da pristupi nekim podacima u bilo kom trenutku.
- Pravila:
  - Morate pokušati da preuzmete zaključavanje pre upotrebe podataka
  - Kada završite sa podacima koje mutex čuva, morate otključati podatke kako bi druge niti mogle da zaključaju podataka i obave svoj posao.

Mutex API

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

    println!("m = {:?}", m);
}
```

Deljenje mutex-a između više niti

```rust
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    for _ in 0..10 {
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

```
$ cargo run
   Compiling shared-state v0.1.0 (file:///projects/shared-state)
error[E0382]: use of moved value: `counter`
  --> src/main.rs:9:36
   |
5  |     let counter = Mutex::new(0);
   |         ------- move occurs because `counter` has type `Mutex<i32>`, which does nc
...
9  |         let handle = thread::spawn(move || {
   |                                    ^^^^^^^ value moved into closure here, in previ
10 |             let mut num = counter.lock().unwrap();
   |                           ------- use occurs due to use in closure

For more information about this error, try `rustc --explain E0382`.
error: could not compile `shared-state` due to previous error
```

Višestruko vlasništvo sa više niti

```rust
use std::rc::Rc;
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Rc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Rc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

```
$ cargo run
   Compiling shared-state v0.1.0 (file:///projects/shared-state)
error[E0277]: `Rc<Mutex<i32>>` cannot be sent between threads safely
   --> src/main.rs:11:22
    |
11  |           let handle = thread::spawn(move || {
    |  ______________________^^^^^^^^^^^^^_-
    | |                      |
    | |                      `Rc<Mutex<i32>>` cannot be sent between threads safely
12  | |             let mut num = counter.lock().unwrap();
13  | |
14  | |             *num += 1;
15  | |         });
    | |_________- within this `[closure@src/main.rs:11:36: 15:10]`
    |
    = help: within `[closure@src/main.rs:11:36: 15:10]`, the trait `Send` is not impleme
    = note: required because it appears within the type `[closure@src/main.rs:11:36: 15:
note: required by a bound in `spawn`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `shared-state` due to previous error
```

Atomično brojanje referenci sa *Arc*

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

Omogućavanje prenosa vlasništva između niti upotrebom Send

- Vlasništvo nad vrednostima tipa koji implementira Send može se preneti između niti.
- Skoro svaki tip u Rust-u je Send, osim Rc
  - Ne može da bude Send zato što ako ga klonirate i pokušate da prenesete vlasništvo nad klonom na drugu niti, obe niti mogu ažurirati broj referenci u isto vreme

Omogućavanje pristupa iz više niti sa Sync

- Sync ukazuje da je bezbedno da tip koji primenjuje Sync bude referenciran iz više niti. - referenca se može bezbedno poslati drugoj niti