Dr Dinu Dragan

PARALELNE I DISTRIBUIRANE ARHITEKTURE I JEZICI (ČAS 9)

ŠTA RADIMO DANAS?

- Konkurentno programiranje

KONKURENTNO PROGRAMIRANJE

OSNOVNI POJMOVI/PRAVILA

- Ogroman broj problema u konkurentom programiranju nastaje iz ograničenja memorijskog modela okruženja u kojem se primenjuje

  In the long run it is not advisable to write large concurrent programs in machine-oriented languages that permit unrestricted use of store locations and their addresses. There is just no way we will be able to make such programs reliable (even with the help of complicated hardware mechanisms).

  —Per Brinch Hansen (1977)

- Postoji veliki broj idioma koji opisuju kako se treba ponašati i šta treba raditi (kako organizovati kod i delegirati poslove), ali ne rade svi uvek
- Treba izbegavati kompleksnost, tj. držati kod što je moguće jednostavnijim

OSNOVNI POJMOVI/PRAVILA – OSNOVNI IDIOMI

- **Background thread** – pozadinski proces koji se periodično budu da odradi neki zadatak
- **Worker pools** i **task queues** – skup radnih procesa koji obrađuju zadatke iz neke liste (reda)
- **Pipelines** – protočni sistem u kojem podaci teku kroz niti u nizu, pri čemu svaka nit odrađuje deo posla
- **Data parallelism** – pretpostavlja se da se ceo računar ponaša kao jedna računska mašina i gde se onda proces računanja (obrade) deli na **n** delova i izvršava u **n** procesa (niti) u paraleli (ideja je da se svaki izvršava na jednom do jezgara procesora u paraleli)

OSNOVNI POJMOVI/PRAVILA – OSNOVNI IDIOMI

- **Sea of synchronized objects** – više niti dele resurse (objekte) pri čemu se primenom *ad hoc* mehanizma zaključavanja (**locking scheme**) i jednostavnih primitiva (nalik **mutex** procedurama)
- **Atomic integer operations** – omogućuje većem broju jezgara da komunicira razmenom informacija veličine jedne mašinske reči (vrlo kompleksan mehanizam koji se svodi na razmenu pokazivača)

- Problem je u tome što u suštini možete slobodno kombinovati više pristupa (pri čemu to mogu da rade i drugi i onda dolazi do nekompatibilnosti)
- Programi koji se oslanjaju na niti i konkurentno programiranje su uglavnom puni nepisanih pravila (a kako nisu zapisani, samo ih autori znaju…)

OSNOVNI POJMOVI/PRAVILA – OSNOVNI IDIOMI

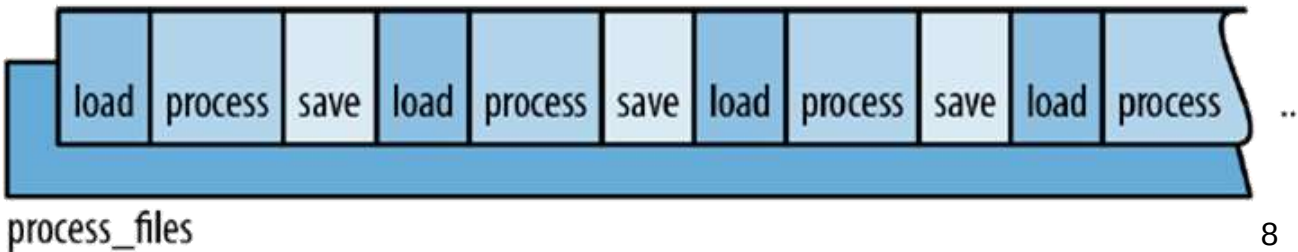
- Rust podržava konkurentno programiranje tako što podržava više stilova (naravno koja su potpuno sigurna iz Rust ugla sigurnosti), ali kroz pravila ugrađena u kompajler
- Rust omogućava pisanje sigurnih, brzih, konkurentnih programa
- Rust niti se mogu kreirati na tri načina:
  - **Fork-join parallelism**
  - **Channels**
  - **Shared mutable state**

FORK-JOIN PARALLELISM

- Najjednostavniji primer upotrebe konkurentnog programiranja je kada imamo nekoliko nezavisnih zadataka koje bi da odradimo od jednom
- Na primer, pretpostaviti obradu teksta primenom NLPa (natural language processing) nad velikim skupom dokumenata
- Single thread aplikacija bi imali sledeći izgled:

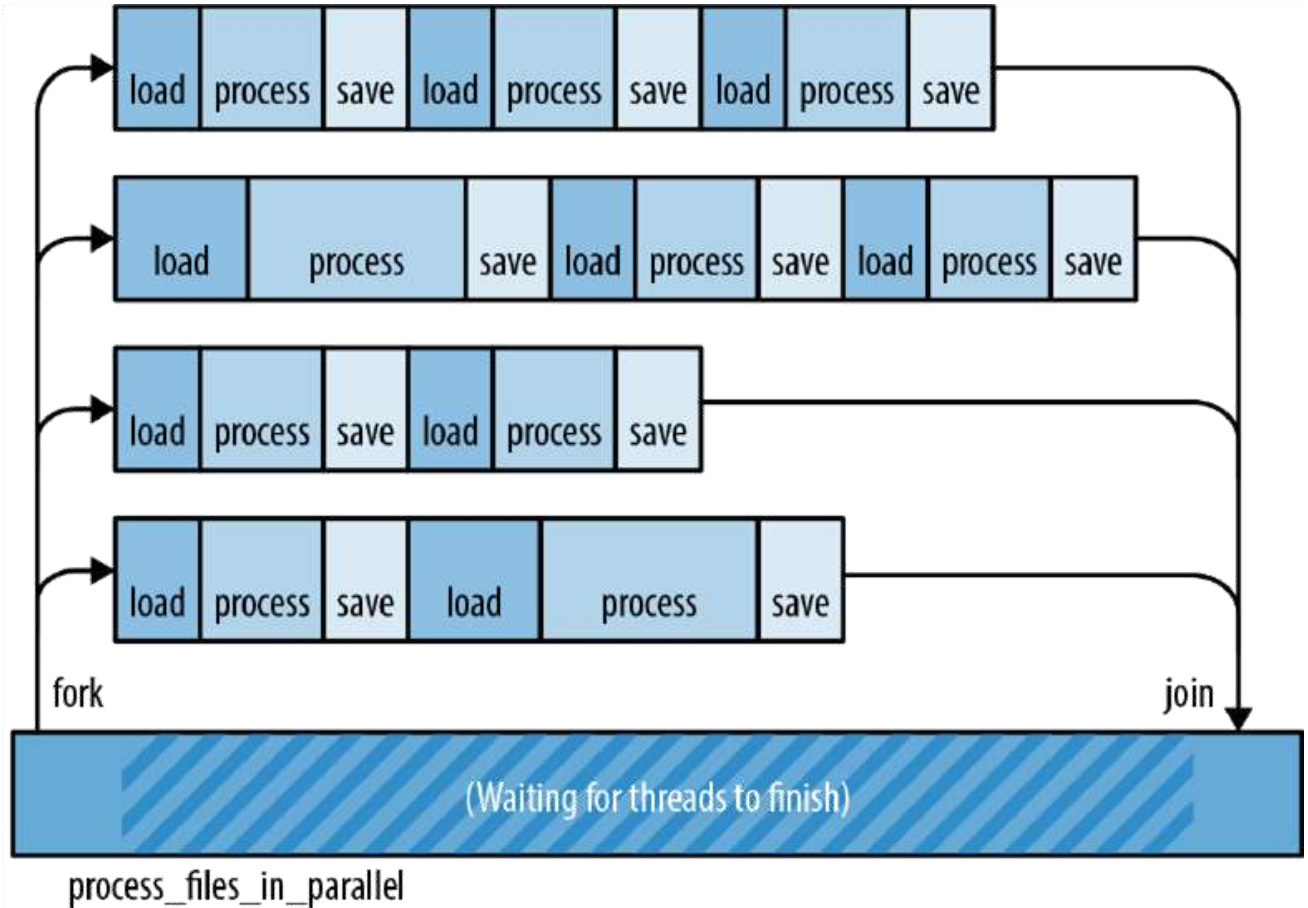
```
fn process_files(filenames: Vec<String>) -> io::Result<()> {
    for document in filenames {
        let text = load(&document)?;  // read source file
        let results = process(text);  // compute statistics
        save(&document, results)?;    // write output file
    }
    Ok(())
}
```

- Program bi se izvršavao na sledeći način



FORK-JOIN PARALLELISM

- Kako su zadaci nezavisni (svaki dokument se zasebno procesira), lako je izdeliti problem na nezavisne podzadatke, svaki staviti u zasebnu nit izvršavanja, i na taj način ubrzati čitav postupak



FORK-JOIN PARALLELISM

- Ovaj proces se naziva **fork-join parallelism**
- **Fork** podrazumeva stvaranje i pokretanje novog procesa (nove niti)
- **Join** podrazumeva čekanje da se pokrenuta nit izvrši i spajanje čitavog procesa u kraj
- Zašto je ovo tako privlačno?
  1. Vrlo je lak za implementaciju i Rust u tome zdušno pomaže
  2. Izbegava se „usko grlo“ (bottleneck), niko nikoga ne čeka, jer nema deljenih resursa; jedino gde se niti čekaju je na kraju (čeka se kraj neke niti)
  3. Računanje performanse sistema je lako; u najboljem slučaju, ako pokrenemo četiri niti, završićemo posao za četvrtinu vremena (naravno, teško da možemo na idealan način da distribuiramo posao ili da ga na idealni način spojimo na kraju)

FORK-JOIN PARALLELISM

- Zašto je ovo tako privlačno?
    4. Lako je proveriti ispravnost programa; ovi programi su deterministički sve dok su niti izolovane u svom poslu i uvek završavaju sa istim rezultatom bez obzira na varijacije u brzini izvršavanja pojedinačnih niti, nema **race conditions**
- Mana ovog pristupa je što radi samo u specifičnim slučajevima (mora postojati izolovanost paralelnih poslova)
- Teško je da se može garantovati postojanje izolovanosti u svim slučajevima
- Rust podržava ovaj obrazac rada kroz **spawn** i **join**

FORK-JOIN PARALLELISM

- Nova nit se stvara pozivom funkcije **std::thread::spawn**

```
use std::thread;

thread::spawn(|| {
    println!("hello from a child thread");
});
```

- Ova funkcija prima kao argument ili funkciju ili FnOnce closure
- Rust stvara novu nit sa telom navedene funkcije ili closure anonimne funkcije
- Nova nit je zapravo nova nit operativnog sistema sa svojim stekom
- Primer sa NLPom se nalazi na sledećem slajdu

FORK-JOIN PARALLELISM


```rust
use std::{thread, io};

fn process_files_in_parallel(filenames: Vec<String>) -> io::Result<()> {
    // Divide the work into several chunks.
    const NTHREADS: usize = 8;
    let worklists = split_vec_into_chunks(filenames, NTHREADS);

    // Fork: Spawn a thread to handle each chunk.
    let mut thread_handles = vec![];
    for worklist in worklists {
        thread_handles.push(
            thread::spawn(move || process_files(worklist))
        );
    }

    // Join: Wait for all threads to finish.
    for handle in thread_handles {
        handle.join().unwrap()?;
    }

    Ok(())
}
```

FORK-JOIN PARALLELISM

- Zaglavlje funkcije **process_files_in_parallel** nije slulajno isto kao i zaglavlje **process_files**
- Posao je podeljen na nekoliko gomila upotrebom metode **split_vec_into_chunks** čija implementacija nije uključena
- Rezultat deljenja posla je vektor, **worklists**, čiji su elementi vektori; sadrži 8 delova jednake veličine iz originalnog vektora
- Za svaki **worklist** se kreira po jedan vektor
- Metoda **spawn()** vraća vrednost **JointHandle** za kasniju upotrebu i to se smešta u **thread_handles** vektor

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

fn process_files_in_parallel(filenames: Vec<String>) -> io::Result<()> {
    // Divide the work into several chunks.
    const NTHREADS: usize = 8;
    let worklists = split_vec_into_chunks(filenames, NTHREADS);

    // Fork: Spawn a thread to handle each chunk.
    let mut thread_handles = vec![];
    for worklist in worklists {
        thread_handles.push(
            thread::spawn(move || process_files(worklist))
        );
    }

    // Join: Wait for all threads to finish.
    for handle in thread_handles {
        handle.join().unwrap()?;
    }

    Ok(())
}
```

FORK-JOIN PARALLELISM

- Vektor **worklist** je kreiran i popunjen unutar for petlje
- Čim se napravi **move closure**, **worklist** se prebacuje u **closure**
- Metoda **spawn** zatim prebacuje **closure** i **worklist** vektor sa njim u novu nit koja se kreira
- Ovo prebacivanje je jeftino
- Sav kod koji je potreban niti, kao i podaci se nalaze u **closure** koja se prosleđuje (obratiti pažnju da je povratna vrednost **closure** funkcije funkcija **process_files**)
- Na kraju se poziva **.join()** metoda **JointHandle** za svaku nit da bi se sačekalo da svaka nit završi

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

fn process_files_in_parallel(filenames: Vec<String>) -> io::Result<()> {
    // Divide the work into several chunks.
    const NTHREADS: usize = 8;
    let worklists = split_vec_into_chunks(filenames, NTHREADS);

    // Fork: Spawn a thread to handle each chunk.
    let mut thread_handles = vec![];
    for worklist in worklists {
        thread_handles.push(
            thread::spawn(move || process_files(worklist))
        );
    }

    // Join: Wait for all threads to finish.
    for handle in thread_handles {
        handle.join().unwrap()?;
    }

    Ok(())
}
```

FORK-JOIN PARALLELISM

- Na kraju se poziva **.join()** metoda **JointHandle** za svaku nit da bi se sačekalo da svaka nit završi
- Spajanje je potrebno zbog provere korekntosti, kao i da bi se sve korektno završilo
- Rust program završava odmah kada se završi **main** funkcija
- To će se desiti odmah kada **main** vrati vrednost bez obzira da li su sve niti završili sa svojim izvršavanjem
- Destrukori se ne pozivaju, niti se prosto ubiju
- Da bi se to izbeglo, poziva se spajanje niti
- Obratite pažnju na greške

```rust
use std::{thread, io};

fn process_files_in_parallel(filenames: Vec<String>) -> io::Result<()> {
    // Divide the work into several chunks.
    const NTHREADS: usize = 8;
    let worklists = split_vec_into_chunks(filenames, NTHREADS);

    // Fork: Spawn a thread to handle each chunk.
    let mut thread_handles = vec![];
    for worklist in worklists {
        thread_handles.push(
            thread::spawn(move || process_files(worklist))
        );
    }

    // Join: Wait for all threads to finish.
    for handle in thread_handles {
        handle.join().unwrap()?;
    }

    Ok(())
}
```

FORK-JOIN PARALLELISM

- Interesantna stvar se dešava kada je potrebno podeliti deljenu nepromenljivu referencu između niti
- Na primer, u NLP primeru pretpostaviti da se koristi ogromni rečnik engleskog rečnika, pošto je velik, on bi se trebao preneti po referenci, nešto ovako:

```
// before
fn process_files(filenames: Vec<String>)

// after
fn process_files(filenames: Vec<String>, glossary: &GigabyteMap)
```

- Međutim, ovo neće raditi sa nitima kada se pokuša referenca prebaciti u **closure** (zbog životnog veka reference i niti)
- Nit može da traje doveka, a referenca ima svoj životni vek, tako je da **closure** ograničenog životnog veka

FORK-JOIN PARALLELISM

- Prosleđivanje reference u nit:

```
fn process_files_in_parallel(filenames: Vec<String>,
                             glossary: &GigabyteMap)
    -> io::Result<()>
{
    ...
    for worklist in worklists {
        thread_handles.push(
            spawn(move || process_files(worklist, glossary))  // error
        );
    }
    ...
}
```

- će izbaciti grešku:

```
error[E0621]: explicit lifetime required in the type of `glossary`
  --> src/lib.rs:75:17
   |
61 |     glossary: &GigabyteMap)
   |               ------------ help: add explicit lifetime `'static` to the
   |                              type of `glossary`: `&'static BTreeMap<String,
   |                              String>`
...
75 |                 spawn(move || process_files(worklist, glossary))
   |                 ^^^^^ lifetime `'static` required
```

FORK-JOIN PARALLELISM

- Uvek možemo da kloniramo reurs i podelimo njegove kopije, ali da li je to dobro rešenje? Rešenje je upotreba pametnih pokazivača, **Arc** referenca

```
use std::sync::Arc;

fn process_files_in_parallel(filenames: Vec<String>,
                             glossary: Arc<GigabyteMap>)
    -> io::Result<()>
{
    ...
    for worklist in worklists {
        // This call to .clone() only clones the Arc and bumps the
        // reference count. It does not clone the GigabyteMap.
        let glossary_for_child = glossary.clone();
        thread_handles.push(
            spawn(move || process_files(worklist, &glossary_for_child))
        );
    }
    ...
}
```

FORK-JOIN PARALLELISM

- Sada se samo povećava broj referenci na isti resurs
- Pozivalac mora proslediti **Arc<GigabyteMap>** što će dovesti do premeštanja reference na heap prilikom njenog kreiranja, **Arc::new(giga_map)**
- Poziv metode **glossary.clone()** će samo kopirati referencu i povećati broj referenci ali ne i klonirati resurs u memoriji
- Sada se program radi jer ne zavisi više od životnog veka reference
- Sve dok bilo koja nit poseduje neku Arc<GigabyteMap> referencu, sama mapa će ostati živa, neće biti obrisana čak i ako njena originalna nit prestane da se izvršava
- Naravno neće biti **data races**, jer je u pitanju nepromenljiva referenca

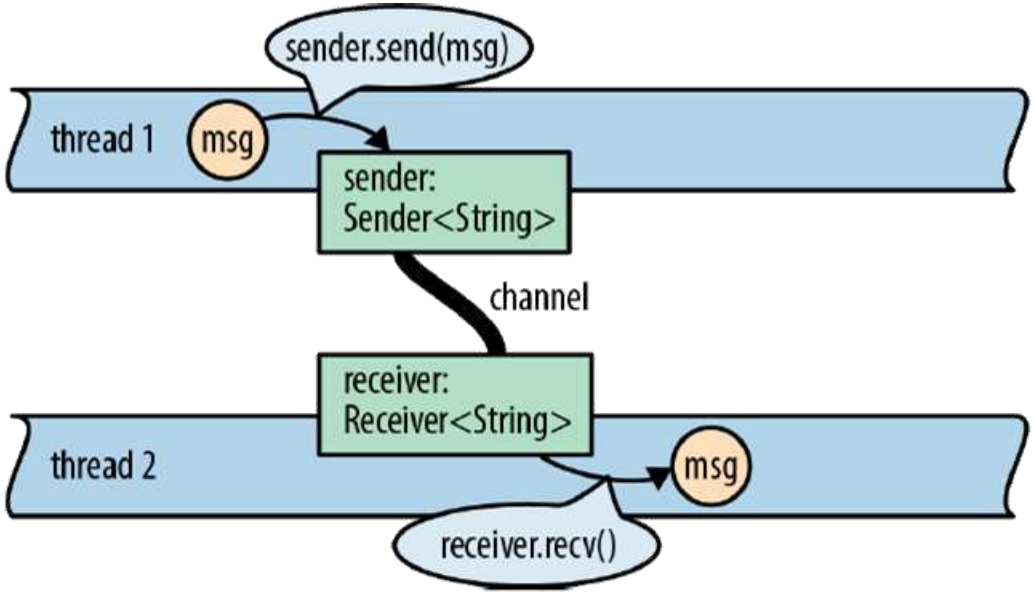
FORK-JOIN PARALLELISM

- **spawn** metoda je deo standardne biblioteke, ali nije dizajnirana samo za **fork-join** obrazac
- Na osnovu nje su izgrađeni sanduci sa boljom podrškom za **fork-join** obrazac
- Neke od njih su **Crossbeam** i **Rayon** biblioteke

```
use rayon::prelude::*;

// "do 2 things in parallel"
let (v1, v2) = rayon::join(fn1, fn2);

// "do N things in parallel"
giant_vector.par_iter().for_each(|value| {
    do_thing_with_value(value);
});
```

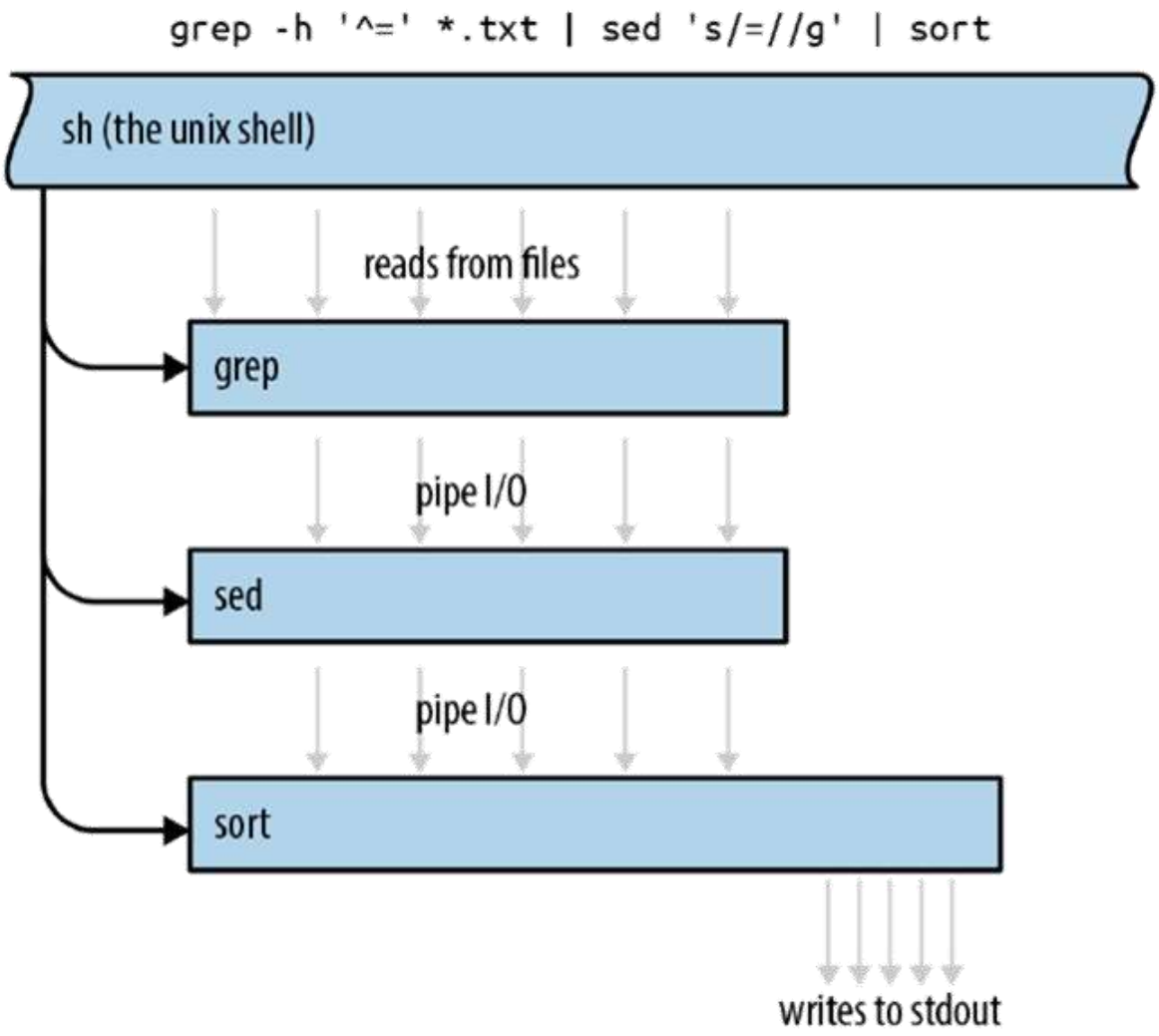
CHANNELS

- **Channels** predstavljaju mehanizam za sigurno jednosmerno prebacivanje vrednosti iz jedne niti u drugu
- Drugom rečju, to je siguran red za rad u konkurentnom okruženju (**thread-safe queue**)
- Oni se mogu zamisliti kao cev kroz koju se poruke šalju sa jednog kraja na drugi, uvek u istom smeru, pri čemu su krajevi obično u različitim nitima
- **Channels** služe za slanje Rust vrednosti
- Pošiljao stavlja jednu vrednost na kanal pomoću **sender.send(item)**



CHANNELS

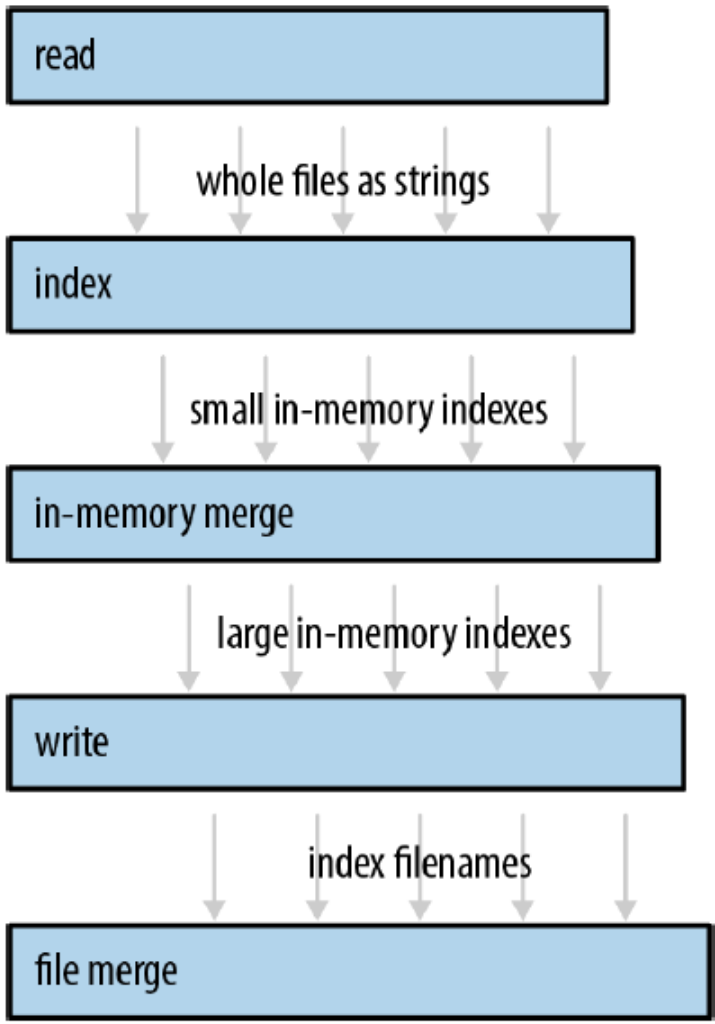
- Primer Unix pipeline:



- Rust-ovi kanali su brži od Unix cevi
- Slanje poruka, zapravo radi njihovo pomeranje (**move**)
- Ovo je brzo čak i kada su megabajti u pitanju

CHANNELS

- Opis kanala biće dan na primeru izgradnje obrnutog indeksa (**inverted index**), nešto što se koristi u pretraživanju dokumenata (pravi se šema koja se reč nalazi u kom dokumentu
- Program je konstruisan kao protočni sistem i biće prikazan deo koda samo za konkurentno programiranje
- U primeru će biti upotrebljeno 5 niti, svaka obavlja svoj deo posla
- Prva nit čita fajlove sa diska u memoriju, što će rezultovati dugačkim stringom, koji se prosleđuje sledećoj niti kroz String kanal



CHANNELS

- Program počinje pokretanjem niti koja čita fajlove

```
use std::{fs, thread};
use std::sync::mpsc;

let (sender, receiver) = mpsc::channel();

let handle = thread::spawn(move || {
    for filename in documents {
        let text = fs::read_to_string(filename)?;

        if sender.send(text).is_err() {
            break;
        }
    }
    Ok(())
});
```

- U vektoru **documents** tipa **Vec<PathBuf>** se nalaze putanje do fajlova

CHANNELS

- Kanali su deo **std::sync::mpsc** modula
- Linija **let (sender, receiver) = mpsc::channel();** dovodi do stvanja kanala, pri čemu nastaju dve vrednosti, **sender** i **receiver**
- U ovom konkretnom slučaju, pošiljalac je tipa **Sender<String>**, dok je primalac tipa **Receiver<String>**
- Tip primaoca i pošiljaoca se može eksplicitno postaviti, navođenjem **mpsc::channel::<String>()**, ali se u ovom slučaju ostavilo da Rust sam dedukuje
- I u ovom primeru se **std::thread::spawn** koristi da pokrene nit

```
use std::{fs, thread};
use std::sync::mpsc;

let (sender, receiver) = mpsc::channel();

let handle = thread::spawn(move || {
    for filename in documents {
        let text = fs::read_to_string(filename)?;

        if sender.send(text).is_err() {
            break;
        }
    }
    Ok(())
});
```

CHANNELS

- Vlasništvo nad **sender**-om se pozivom **move closure** funkcije prebacuje na novonastalu nit
- Ostatak koda čita tekst iz fajla
- Ako je čitanje bilo uspešno, tekst se prosleđuje primaocu pomoću **sender.send(text)**, ujedno se vlasništvo nad tekstom ovim putem premešta u primaoca
- Pošiljalac zapravo premešta tekst na kanal
- Koliko god da je tekst velik, sama operacija premešta svega tri mašinske reči (to je sve što se nalazi na steku)
- Isto će i poziv **receiver.recv()** kopirati tri mašinske reči

```
use std::{fs, thread};
use std::sync::mpsc;

let (sender, receiver) = mpsc::channel();

let handle = thread::spawn(move || {
    for filename in documents {
        let text = fs::read_to_string(filename)?;

        if sender.send(text).is_err() {
            break;
        }
    }
    Ok(())
});
```

CHANNELS

- Obe metode, i **send** i **receive** vraćaju rezultat tipa **Result**, tako da mora postojati rukovanje greškom
- Ako kojim slučajem dolazi do brisanja primaoca (**dropped**), pre nego što je vrednost pročitana sa kanala, **send** će generisati grešku (u suprotnom bi vrednost ostala zauvek u kanalu)
- Isto će i **receive** izazvati grešku ako na kanalu ne postoji vrednost, a pošiljalac je obrisan (u suprotnom bi primalac morao da čeka doveka)
- Uobičajeno je svaka strana na ovaj način prekine komunikaciju
- Šta god da se desi (svi fajlovi su pročitani ili je primalac prestao

```
use std::{fs, thread};
use std::sync::mpsc;

let (sender, receiver) = mpsc::channel();

let handle = thread::spawn(move || {
    for filename in documents {
        let text = fs::read_to_string(filename)?;

        if sender.send(text).is_err() {
            break;
        }
    }
    Ok(())
});
```

CHANNELS

- Uobičajeno je svaka strana prekine komunikaciju ako na drugoj strani nema više nikoga
- Šta god da se desi (svi fajlovi su pročitani ili je primalac prestao da postoji), u primeru se vraća **Result Ok()**
- Jedina mogućnost da pošiljalac bude neuspešan je upravo to, da primalac izađe ranije iz komunikacije
- Naravno, mogu postojati greške usled rada sa fajlovima, u tom slučaju pošiljalac odmah prekida sa svojim izvršavanjem

```
use std::{fs, thread};
use std::sync::mpsc;

let (sender, receiver) = mpsc::channel();

let handle = thread::spawn(move || {
    for filename in documents {
        let text = fs::read_to_string(filename)?;

        if sender.send(text).is_err() {
            break;
        }
    }
    Ok(())
});
```

CHANNELS

- Zadatak primaoca u ovom primeru je da iščitava kanal i obrađuje pristigle vrednosti
- On će to raditi sve dok je primalac aktivan i dok ima vrednosti na kanalu
- Onog trena kad se pošiljalac obriše i kanal se isprazni, briše se i primalac
- U osnovi primalac radi u petlji koja može biti **while** ili **for**

```
while let Ok(text) = receiver.recv() {
    do_something_with(text);
}
```

```
for text in receiver {
    do_something_with(text);
}
```

- U oba slučaja, ako je kanal prazan kada se dođe do vrha petlje, primalac blokira i čeka sledeću vrednost
- Kada je kanal prazan a pošiljalac obrisan, izlazi se iz petlje normalno
- Nit dobija promenljivu **sender** kroz poziv **closure** funkcije

CHANNELS

- Drugi deo protočnog sistema ima sledeći izgled:

```
fn start_file_indexing_thread(texts: mpsc::Receiver<String>)
    -> (mpsc::Receiver<InMemoryIndex>, thread::JoinHandle<()>)
{
    let (sender, receiver) = mpsc::channel();

    let handle = thread::spawn(move || {
        for (doc_id, text) in texts.into_iter().enumerate() {
            let index = InMemoryIndex::from_single_document(doc_id, text);
            if sender.send(index).is_err() {
                break;
            }
        }
    });

    (receiver, handle)
}
```

CHANNELS

- Funkcija pokreće nit koja prima String vrednosti iz jednog kanala (**texts**) i koja šalje InMemoryIndex vrednosti drugom kanalu (**sender/receiver**)
- Zadatak ove druge niti je da preuzme fajl učitan u prvoj niti i da ga pretvori **inverted index** fajl u memoriji
- Sav posao oko pravljenja obrnutog indeksa se nalazi u metodi **InMemoryIndex::from_single_document**
- Ovde ne postoji I/O operacija, tako da nema ni grešaka koje to obrađuju

```
fn start_file_indexing_thread(texts: mpsc::Receiver<String>)
    -> (mpsc::Receiver<InMemoryIndex>, thread::JoinHandle<()>)
{
    let (sender, receiver) = mpsc::channel();

    let handle = thread::spawn(move || {
        for (doc_id, text) in texts.into_iter().enumerate() {
            let index = InMemoryIndex::from_single_document(doc_id, text);
            if sender.send(index).is_err() {
                break;
            }
        }
    });

    (receiver, handle)
}
```

CHANNELS

- Preostala tri elementa u protočnom sistemu su vrlo sličnog dizajna
- Svaka sledeća nit konzumira primaoca kanala nastalog u prethodnoj niti
- Cilj ostatka sistema je da se svi mali indeksni fajlovi spoje u jedan veliki fajl na disku
- To se može realizovati u 3 faze (koje neće biti ovde prikazane detaljno)
- U trećoj fazi se indeksi spajaju u memoriji sve dok ne dosegnu veličinu suviše veliku da se njome manipuliše u memoriji

```
fn start_in_memory_merge_thread(file_indexes: mpsc::Receiver<InMemoryIndex>)
    -> (mpsc::Receiver<InMemoryIndex>, thread::JoinHandle<()>)
```

- U četvrtoj fazi se veliki indeks fajl iz memorije beleži na disk

```
fn start_index_writer_thread(big_indexes: mpsc::Receiver<InMemoryIndex>,
                             output_dir: &Path)
    -> (mpsc::Receiver<PathBuf>, thread::JoinHandle<io::Result<()>>)
```

CHANNELS

- U petoj fazi se veliki indeks fajl sa diska spajaju u jedan veliki indeksni fajl na disku upotrebom algoritama za spajanje fajlova

```
fn merge_index_files(files: mpsc::Receiver<PathBuf>, output_dir: &Path)
    -> io::Result<()>
```

- Kako se peta faza nalazi na kraju protočnog sistema, ona ne vraća primaoca, jer posle nje ne ide nikakva nova nit, te nema ni deljenog kanala
- Ona kreira jedan veliki fajl na disku
- Kod koji pokreće čitav protočni sistem i koji proverava i rukuje greškama je opisan na sledećem slajdu

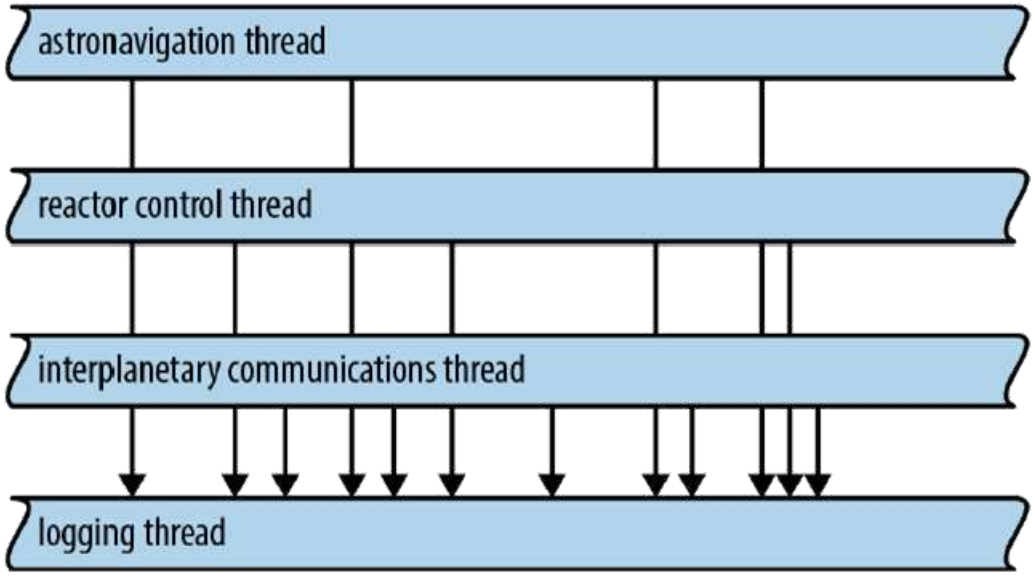
CHANNELS

- Kod koji pokreće čitav protočni sistem

```rust
fn run_pipeline(documents: Vec<PathBuf>, output_dir: PathBuf)
    -> io::Result<()>
{
    // Launch all five stages of the pipeline.
    let (texts,   h1) = start_file_reader_thread(documents);
    let (pints,   h2) = start_file_indexing_thread(texts);
    let (gallons, h3) = start_in_memory_merge_thread(pints);
    let (files,   h4) = start_index_writer_thread(gallons, &output_dir);
    let result = merge_index_files(files, &output_dir);

    // Wait for threads to finish, holding on to any errors that they encounter.
    let r1 = h1.join().unwrap();
    h2.join().unwrap();
    h3.join().unwrap();
    let r4 = h4.join().unwrap();

    // Return the first error encountered, if any.
    // (As it happens, h2 and h3 can't fail: those threads
    // are pure in-memory data processing.)
    r1?;
    r4?;
    result
}
```

CHANNELS

- **mpsc** deo modula **std::sync::mpsc** odnosi se na **multiproducer, single-consumer**, koji zapravo opisuje kakvu komunikaciju pružaku Rust kanali
- U primeru smo imali jedan izvor i jednog konzumenta, jer se vrednost prenosila od jednog pošiljaoca ka jednom primaocu
- Rust omogućava i da postoji više pošiljaoca u slučajevima kada jedan primalac obrađuje poruke iz više različitih izvora
- Primer može biti jedna nit za beleženje logova



CHANNELS

- **Sender<T>** implementira **Clone** osobinu
- Da bi se napravio kanal sa više pošiljalaca, potrebno je samo napraviti običan regularan kanal i klonirati ga onoliko puta koliko je to potrebno
- Svaka **Sender** vrednost može se pomeriti u drugu nit
- **Receiver<T>** se ne može klonirati
- Ako je potrebno implementirati više primaoca koji primaju neku poruku, onda se mora koristiti **mutex** mehanizam, **Mutex** osobina
- Rad sa kanalima u Rust-u je optimizovan, kompleksnost kanala raste sa brojem poruka koje se šalju i brojem izvora poruka (menja se konkretna implementacija reda u zavisnosti od kompleksnosti)
- Sam proces se svodi na par atomičnih operacija + heap alokacija + pomeranje
- Sistemski pozivi su potrebni samo kada nit mora na spavanje (a to se dešava kada je red prazan)

CHANNELS

- Jedna od čestih grešaka (koja negativno utiče na performansu) jeste kada pošiljalac generiše poruke brže nego što primalac može da obradi
- To dovodi do stvaranja reda neobrađenih poruka (**backlog**) na kanalu, pri čemu broj poruka konstantno raste
- U primeru, faza 1 generiše daleko više fajlova nego što faza 2 može da procesira i indeksira (do čega ovo dovodi?)
- Ovo ume da košta protočni sistem u vidu memorije, ali i same niti lokalno
- U trenutnoj implementaciji pošiljalac nastavlja da radi, pa samim tim i on troši resurse a njegov rezultat nema ko da obradi
- Rešenje?

CHANNELS

- Jedno rešenje je da se pošiljalac uspori, backpressure u Unix-u
- Veličina kanala se ograniči na tačno određen broj poruka
- Rust podržava ovaj mehanizam – kanali sa sinhronizovanom razmenom poruka (**synchronous channel**)

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

let (sender, receiver) = mpsc::sync_channel(1000);
```

- Metoda **sender.send(value)** može da blokira pošiljaoca ako je u trenutku slanja kanal prepunjen porukama
- U primeru ograničavanje kanala na 32 poruke je smanjilo korišćenje memorije za dve trećine uz nikakav uticaj na protok čitavog sistema

CHANNELS

- Ne mogu se sve vrednosti na siguran način deliti između niti
- Zbog toga Rust predviđa dva načina rada
- Vrednosti koje implementiraju **std::marker::Send** mogu se bezbedno prenositi po vrednosti u drugu nit
- Vrednosti koje implementiraju **std::marker::Sync** mogu da se bezbedno prenose nepromenljive reference u drugu nit
- U primeru, Vec<String> se prenosi bezbedno između niti
  - I vektor i string su kreirani i alocirani u roditeljskoj niti, a oslobađaju se novonastaloj niti
- **Vec<String>** implementira **Send** kao API obavezu tako da je sve **OK**
- Većina Rust tipova je implementira i **Send** i **Sync**
- Struct i enum su po automatizmu **Send** ako su sva njihova polja **Send** tipa, isto važi i za **Sync**

CHANNELS

- Neki tipovi su **Sync**, ali ne i **Send**
- To je gotovo uvek urađeno namerno, kao na primer sa mpsc::Receiver gde se onda garantuje da će primalac biti korišćen u samo jednoj niti
- Tipovi koji nisu ni **Send**, ni **Sync** u suštini nisu sigurni za korišćenje u konkurentnim programima (uglavnom njihova promenljiva forma nije sigurna za rad sa nitima)

- Čitav mehanizam za pravljenje kanala dozvoljava kreiranje iteratora i pakovanje čitavog koda u samo par povezanih linija, nešto ovog tipa:

```
documents.into_iter()
    .map(read_whole_file)
    .errors_to(error_sender)   // filter out error results
    .off_thread()              // spawn a thread for the above work
    .map(make_single_file_index)
    .off_thread()              // spawn another thread for stage 2
    ...
```

SHARED MUTABLE STATE

- Osnova konkurentnog programiranja jesu i deljeni resursi, resursi kojima više niti pristupa
- Rust za to ima čitav mehanizam koji se oslanja na sledeće:
  - mutex
  - read/write locks,
  - condition variables,
  - atomic integers
- Sve ćemo ih pogledati
- Kao primer uzećemo igricu u kojoj postoji lista igrača kojoj pristupa više niti
- Igra počinje kada se prijavi 8 igrača

MUTEX

- Mutex ili lock, je mehanizam za zaključavanje resursa
- Koristi se kako bi se više niti nateralo da pristupa deljenom resursu u nekom redosledu
- Kako Mutex radi u C++?
- Kritičan kod se nalazi između **mutex.Acquire()** i **mutex.Release()**
- Ako je neka nit u tom stanju, ostali su blokirani na **mutex.Acquire()**
- Koji je limit kod C++?

```
// C++ code, not Rust
void FernEmpireApp::JoinWaitingList(PlayerId player) {
    mutex.Acquire();

    waitingList.push_back(player);

    // Start a game if we have enough players waiting.
    if (waitingList.size() >= GAME_SIZE) {
        vector<PlayerId> players;
        waitingList.swap(players);
        StartGame(players);
    }

    mutex.Release();
}
```

MUTEX

- **Mutex** su korisni iz nekoliko razloga
- Sprečavaju trke u podacima (**data races**)
  - U velikom broju programskih jezika (**Go**, **C++**) ovo izaziva undefined ponašanje
  - Mnogi programski jezici tvrde da ovo neće izazvati pad (**Java**, **C#**), ali je rezultat u mnogim situacijama ipak čista glupost
- Čak i kada da se trka u podacima nikada ne dešava (a dešava se), nekontrolisan pristup deljenom resursu, izazvao bi vrlo nepredvidivo ponašanje i rezultat bi bio miks različitih vrednosti
- Koriste za održavanje određenih nepromenljivih stanja ili pravila (**invairants**) vezanih za neke promenljive
  - **Mutex** brine da su ta pravila stalno poštuju
- Iako rešavaju puno problema, ako se ne koristi na pravi način, **mutex** može biti izvor problema

MUTEX

- U većini programskih jezika mutex je odvojen od same promenljive
- U C++ **mutex** i objekat su odvojene stvari, tako da morate u komentarima da zamolite da se koriste zajedno

```
class FernEmpireApp {
    ...

private:
    // List of players waiting to join a game. Protected by `mutex`.
    vector<PlayerId> waitingList;

    // Lock to acquire before reading or writing `waitingList`.
    Mutex mutex;
    ...
};
```

- Što znači da poštovanje pravila zapravo nije obavezujuće i zavisi od samog programera, pravilo nije nametnuto kompajlerom
- Ovo važi čak i za Javu gde postoji određena veza između objekta i mutex-a, ali ta veza nije čvrsta

MUTEX<T>

- U primeru, svaki igrač ima svoj ID, **PlayerId**

```
type PlayerId = u32;
```

- Broj igrača je ograničen

```
const GAME_SIZE: usize = 8;

/// A waiting list never grows to more than GAME_SIZE players.
type WaitingList = Vec<PlayerId>;
```

- Lista igrača na čekanju, **WaitingList**, smešta se unutar strukture, FernEmpireApp, singleton koji se stavlja u pametar pokazivač, Arc, tokom pokretanja servera
- Struktura sadrži uglavnom read-only polja i služi da se u nju stave konfiguracioni parametri
- Kako je **WaitingList** i promenljiva i deljiva, mora biti zaštićena muteksom

MUTEX<T>

- Za razliku od C++, u Rust-u se promenljiva koja se štiti smešta unutar muteksa

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

/// All threads have shared access to this big context struct.
struct FernEmpireApp {
    ...
    waiting_list: Mutex<WaitingList>,
    ...
}
```

- Muteks se u inicijalizuje na sledeći način:

```
use std::sync::Arc;

let app = Arc::new(FernEmpireApp {
    ...
    waiting_list: Mutex::new(vec![]),
    ...
});
```

MUTEX<T>

- Metoda koja koristi muteks i dodaje novog korisnika u listu:

```
impl FernEmpireApp {
    /// Add a player to the waiting list for the next game.
    /// Start a new game immediately if enough players are waiting.
    fn join_waiting_list(&self, player: PlayerId) {

            // Lock the mutex and gain access to the data inside.
            // The scope of `guard` is a critical section.
            let mut guard = self.waiting_list.lock().unwrap();

            // Now do the game logic.
            guard.push(player);
            if guard.len() == GAME_SIZE {
                let players = guard.split_off(0);
                self.start_game(players);
            }
        }
    }
```

MUTEX<T>

- Jedini način da se dođe do podataka je da se dohvati ključ pozivom **.lock( )** metode

```
let mut guard = self.waiting_list.lock().unwrap();
```

- **self.waiting_list.lock()** će blokirati izvršavanje, sve dok se ne dohvati muteks, tj. dok se promenljiva ne oslobodi za korišćenje
- **MutexGuard<WaitingList>** vrednost koju vraća ova metoda je vreper oko **&mut WaitingList**
- Sada se **WaitingList** metode direktno pozivaju preko **MutexGuard** instance

```
guard.push(player);
```

- **MutexGuard** omogućuje da dođemo direktno do referenci na podatke koji se nalaze ispod
- Životni vek u Rust-u obezbeđuje da reference ne mogu da nadžive **MutexGuard**

MUTEX<T>

- Kada **MutexGuard** izađe iz opsega i kada se on obriše (**dropped**), muteks se otključava
- Uobičajeno je da se to desi na kraju bloka, ali se može i eksplicitno pozvati

```
if guard.len() == GAME_SIZE {
    let players = guard.split_off(0);
    drop(guard);  // don't keep the list locked while starting a game
    self.start_game(players);
}
```

MUTEX NIJE SVEMOGUĆ

- Iako je Rust muteks moćan, i neće dovesti do trke u podacima, ne može zaštiti od drugih negativnih situacija koje mogu nastati sa deljenim resursima
- Neće sprečiti dolazak do drugih situacija utrkivanja (**race conditions**)
  - Ove situacije nastaju kada postoje tempirane akcije koje zavisi od njihovog redosleda izvršavanja u vremenu
  - Upotreba muteksa na nestrukturiran načinmože izazvati ovo stanje
- Upotreba deljenih resursa utiče na projektovanje softverskog rešenja i arhitekturu samog programa
  - Dodavanje muteksa dozvoljeno je bilo gde u kodu, bez jasnog razgraničavanja konkurentnog od nekonkurentno dela programa
  - Lako je mešati nekonkurentne delove koda i konkurentne delove koda što može dovesti do monolitne hrpe povezanog koda
- Upotreba muteksa nije tako jednostavna

DEADLOCK

- Upotreba muteksa može dovesti do zaključavanje koje se ne može otkočiti (**deadlock**)
- Najjednostavniji način da dođe do dedloka je da pokušate da zaključate resurs koji ste već zaključali

```
let mut guard1 = self.waiting_list.lock().unwrap();
let mut guard2 = self.waiting_list.lock().unwrap();  // deadlock
```

- Ovde je dedlok očigledan, ali vrlo verovatno može da se desi da se u istoj niti upotrebi isti muteks u dve različite metode
- Slično dedlok može nastati kada imate više niti sa više metoda koja uzimaju više muteksa na takav način da se međusobno isključuju i zaključavaju da dođe do dedloka
- Rustov mehanizam pozajmljivanja ne štiti od dedloka, jedino zdrav razum, držati kod koji radi sa muteksom malim kompaktnim – uzmete mutekts, obavite posao, oslobodite muteks

DEADLOCK

- Moguće je napraviti dedlok i kada se koriste kanali
- Dve niti mogu da se blokiraju kada jedna nit čeka na poruku od druge niti, dok druga nit čeka na poruku od prve niti
- Dobar dizajn programa će ovo izbeći

ZATROVANI MUTEKSI

- **Mutex::lock()** vraća **Result** iz istog razloga iz kog **JoinHandle::join()** to radi, kako bi došlo do gracioznog gašenja ako se druga nit uspaničila
- Kada napišete **handle.join().unwrap()**, kažete Rust da propagira paniku iz jedne niti u drugu; sintaksa **mutex.lock().unwrap()** radi sličnu stvar
- Ako se nit uspaniči dok drži muteks, za taj muteks se kaže da je zatrovan
- Svaki pokušaj da se zaključa zatrovani muteks izazvaće grešku, što će dovesti do toga da .unwrap() u zaključavanju muteksa izazove paniku
- Ovo nije nužno loše, ali je odabrano ovakvo ponašanje kako bi se izbeglo da su podaci slučajno ostavljeni u polustanju u niti koja se uspaničila dok je držala muteks
- Zatrovan muteks se može ipak koristiti upotrebom izraza **PoisonError::into_inner()**

MULTICONSUMER CHANNELS

- Rust kanal dozvoljava da imamo više pošiljaoca ali samo jednog primaoca, ali ako se primalac stavi u muteks, je moguće podeliti ga

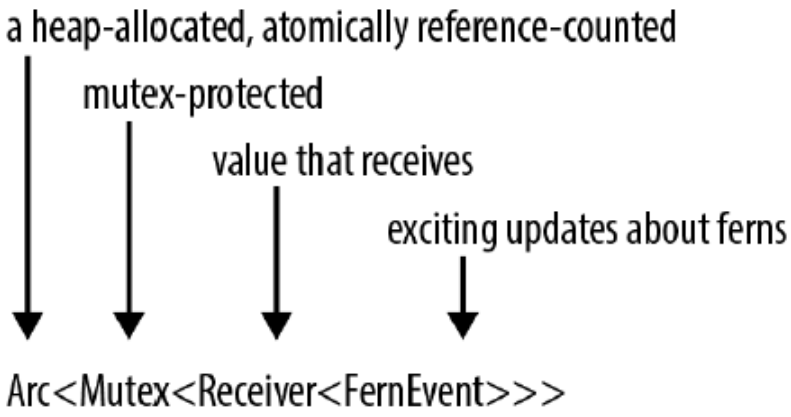
```
mod shared_channel {
use std::sync::{Arc, Mutex};
use std::sync::mpsc::{channel, Sender, Receiver};

/// A thread-safe wrapper around a `Receiver`.
#[derive(Clone)]
pub struct SharedReceiver<T>(Arc<Mutex<Receiver<T>>>);

impl<T> Iterator for SharedReceiver<T> {
    type Item = T;

    /// Get the next item from the wrapped receiver.
    fn next(&mut self) -> Option<T> {
        let guard = self.0.lock().unwrap();
        guard.recv().ok()
    }
}
```

MULTICONSUMER CHANNELS

- Koristi se Arc<Mutex<Receiver<T>>>

```
    /// Create a new channel whose receiver can be shared across threads.
    /// This returns a sender and a receiver, just like the stdlib's
    /// `channel()`, and sometimes works as a drop-in replacement.
    pub fn shared_channel<T>() -> (Sender<T>, SharedReceiver<T>) {
        let (sender, receiver) = channel();
        (sender, SharedReceiver(Arc::new(Mutex::new(receiver))))
    }
}
```

- Na slici se vidi tačno šta se dešava



SINHRONIZACIJA

- **std::sync** je deo standardne biblioteke koji sadrži metode koje se koriste za sinhronizaciju
- Muteks dozvoljava da u datom trenutku postoji samo jedna nit koja radi sa resursom, bilo da se radi o pisanju, bilo da se radi o čitanju
- Ako postoji situacija u kojoj imamo da je čitanje sigurno, onda se može upotrebiti **RwLock** tzv. čitaj/piši zaključavanje (**read/write lock**)
- Ovo se razlikuje od muteksa, što dok god niti samo čitaju resurs (pristupaju metodom **RwLock::read**), mogu neograničeno pristupati resursu
- Onog trena kada neko pristupa resursu za pisanje (preko metode **RwLock::write**), ono se ekskluzivno zaključava
- RwLock::write metoda je sličan Mutex::lock metodi

SINHRONIZACIJA

- U onom primeru, moguće je da postoji struktura za konfiguraciju

```
use std::sync::RwLock;

struct FernEmpireApp {
    ...
    config: RwLock<AppConfig>,
    ...
}
```

- Metode koje žele da pročitaju konfiguraciju koriste **RwLock::read()**

```
/// True if experimental fungus code should be used.
fn mushrooms_enabled(&self) -> bool {
    let config_guard = self.config.read().unwrap();
    config_guard.mushrooms_enabled
}
```

SINHRONIZACIJA

- Metode koje žele da pročitaju konfiguraciju koriste **RwLock::write()**

```
fn reload_config(&self) -> io::Result<()> {
    let new_config = AppConfig::load()?;
    let mut config_guard = self.config.write().unwrap();
    *config_guard = new_config;
    Ok(())
}
```

- Opisano ponašanje **RwLock** je bukvalno ponašanje mehanizma pozajmljivanja
- **self.config.read()** vraća **config_guard** koji omogućuje nemutabilni (deljeni) pristup **AppConfig**
- **self.config.write()** returns vraća **config_guard** koji omogućuje mutabilni (ekskluzivni) pristup **AppConfig**

CONDVAR

- U nekim slučajevima (koji su četi kod konkurentnog programiranja) niti moraju da čekaju da se ispune neki uslovi za njihov dalji rad
  - Prilikom gašenja glavna nit mora da sačeka da sve ostale završe sa poslom
  - Kada radna nit nema šta da radi, mora da čeka da naiđe posao
  - Nit koja implementira konsenzus protokol mora da sačeka da svi čvorovi odgovore
- **JoinHandle::join** je ugrađena u Rust i rešava prvu situaciju, ali za preostale nema ugrađenog mehanizma, već se mora napraviti
- To se može rešiti pomoću promenljivih uslova (**condition variables**)
- U Rust-u **std::sync::Condvar** tip implementira promenljivu uslova
- Condvar sadrži metode **.wait()** i **.notify_all()**; **.wait()** blokira izvršavanje niti sve dok neka druga nit ne pozove **.notify_all()**

CONDVAR

- Malo je kompleksnije od pukog korišćenja **Condvar**, jer je ono obično povezano sa true-false stanjem neke **Mutex** vrednosti
- U osnovi, kada je uslov ispunjen poziva se **Condvar::notify_all** ili **Condvar::notify_one** da se probude uspavane niti

```
self.has_data_condvar.notify_all();
```

- Da bi se nit blokirala nad Condvar promenljivom koristi se **Condvar::wait()**

```
while !guard.has_data() {
    guard = self.has_data_condvar.wait(guard).unwrap();
}
```

- Ako se pogleda **Condvar::wait()**, ono uzima **MutexGuard** objekat, konzumira ga i vraća novi ako je operacija uspela

ATOMICS

- **std::sync::atomic** modul sadrži atomične tipove koji omogućuju konkurentno programiranje bez potreba za zaključavanjem resursa
- U suštini su isti kao standardni C++ atomični tipovi uz dodatke
  - **AtomicIsize** i **AtomicUsize** su deljivi integer tipovi koji korespondiraju single-thread **isize** i **usize** tipovima
  - **AtomicI8**, **AtomicI16**, **AtomicI32**, **AtomicI64** i njihove neoznačene varijante poput **AtomicU8** su deljivi integer tipovi koji korespondiraju single-thread **i8**, **i16** tipovima
  - **AtomicBool** je deljiv **bool** tip
  - **AtomicPtr<T>** je deljiva vrednost verzija nesigurnog mutabilnog pokazivača tipa ***mut T**
- Moguće je da više niti pristupa promenljivama ovog tipa konkurentno a da pri tome ne dolazi do trke u podacima

ATOMICS

- U radu sa atomičnim promenljivama se ne koriste standardni operatori već odgovarajuće atomične metode
- Ovo je važno, jer čak i da druga nit pristupa baš tom delu memorije, atomična operacija se neće prekinuti, tj. završiće se pre nego što ova druga počne
- Inkrementiranje promenljive **atom** tipa **AtomicIsize**:

```
use std::sync::atomic::{AtomicIsize, Ordering};

let atom = AtomicIsize::new(0);
atom.fetch_add(1, Ordering::SeqCst);
```

- Ove metode se mogu kompajlirati u specijalizovane instrukcije mašinskog jezika
- **Ordering::SeqCst** definše da se mora ispratiti redosled u memoriji (**memory ordering**), npr. da uzrok uvek mora prethoditi posledici i sl.

ATOMICS

- Jedan primer upotrebe atomika je prekid (**cancellation**)
- Pretpostaviti da postoji nit za koju se očekuje da će nešto jako dugo procesirati i da želimo da ubacimo mogućnost da se to procesiranje prekine asinhrono
- Problem je kako to ubaciti u nit koja je zauzeta procesiranjem? Preko deljenje **AtomicBool** vrednosti

```
use std::sync::Arc;
use std::sync::atomic::AtomicBool;

let cancel_flag = Arc::new(AtomicBool::new(false));
let worker_cancel_flag = cancel_flag.clone();
```

- Napravljena su dva pametna pokazivača, **Arc<AtomicBool>**, na istu atomičnu promenljivu na heapu i vrednost ove promenljive je **false**
- **cancel_flag** promenljiva ostaje u glavnoj niti, dok će se **worker_cancel_flag** pomeriti u nit koja se želi prekinuti

ATOMICS

- Ovako izgleda worker

```
use std::thread;
use std::sync::atomic::Ordering;

let worker_handle = thread::spawn(move || {
    for pixel in animation.pixels_mut() {
        render(pixel); // ray-tracing - this takes a few microseconds
        if worker_cancel_flag.load(Ordering::SeqCst) {
            return None;
        }
    }
    Some(animation)
});
```

- Nakon što nacrta svaki piksel, metoda provera status atomika pozivom **.load()** metode

ATOMICS

- U slučaju da se želi prekinuti rad worker niti, samo treba postaviti vrednost atomika na **true** u glavnoj niti izvršavanja i sačekati da worker nit završi (ovo se ne mora desiti trenutno)

```
// Cancel rendering.
cancel_flag.store(true, Ordering::SeqCst);

// Discard the result, which is probably `None`.
worker_handle.join().unwrap();
```

- Ista stvar se mogla realizovati i preko kanala i **Mutex<bool>**, ali je rešenje preko atomičnih promenljivih najmanje zahtavan i najjednostavniji je
- Nikada ne koriste sistemske pozive a **load** i **store** se često svode na jednu CPU instrukciju

GLOBALNE PROMENLJIVE

- Ako se sećate, Rust nema podršku za globalne promenljive, i statičke promenljive i konstante su read-only, tj. nepromenljive
- Ipak u situacijama kada je neophodno, globalne promenljive se najjednostavnije mogu realizovati preko atomičnih promenljivih
- Pretpostaviti da želite u nekom serveru da brojite koliko je paketa poslato/usluženo, to može atomični intidžer:

```
use std::sync::atomic::AtomicUsize;

static PACKETS_SERVED: AtomicUsize = AtomicUsize::new(0);
```

- Kad se promenljiva deklariše, posle se inkrementiranje lako radi:

```
use std::sync::atomic::Ordering;

PACKETS_SERVED.fetch_add(1, Ordering::SeqCst);
```

- Globalne promenljive bazirane na atomicima su limitirane na intidžere i bool

GLOBALNE PROMENLJIVE

- Ako želite globalnu promenljivu bilo kog drugog tipa, morate rešiti dva problema
- Prvi je da promenljiva mora biti sigurna za rad u konkurentnom režimu (thread-safe)
  - Ovo prosto rešava Mutex i RwLock
- Drugi je da ih mogu inicijalizovati samo konstantne funkcije, tj. funkcije čije je ponašanje determinističko (zavisi isključivo od vrednosti argumenata funkcije, a ne od I/O ili nekog drugog stanja)
  - Kompajler ovo može da koristi kao compile-time constant
- Konstante funkcije se u Rust-u prave tako što se dodaje prefiks **const**
- Vodite računa da će Rust ograničiti šta ova funkcije može da radi, neće dozvoliti nedeterminističke rezultate, ne može imati generičke argumente i ne dozvoljava alokaciju memorije ili rad sa pokazivačima

GLOBALNE PROMENLJIVE

- Dozvoljene su aritmetičke operacije, logičke operacija bez skraćivanja i druge konstantne funkcije