Rust: Thread, Mutex, Atomic, Channel

Hafta - 5

Fearless Concurrency

"Korkusuz Eşzamanlılık"

Rust güvenli bir dil olduğundan diğer dillerdeki multithreaded çalışma durumunda oluşabilecek sorunlar derleme esnasında çözülür.

Örneğin bir değişkeni birden fazla thread'de atomiklik veya mutex sağlamadan kullanamazsınız.

Yani Multithreaded yazdığınız bir program eğer derleniyorsa yüksek ihtimalle multithreaded'den kaynaklı bir sorun yoktur.

Elbette programınızda yine buglar olabilir. Rust bunları engelleyemez. :)

Thread

İş Parçacığı.
Birden fazla işlemci çekirdeğinde aynı anda kod çalıştırmayı sağlar.

Thread

```
use std::thread;

fn main() {
      thread::spawn(|| {
            println!("Thread'den merhaba!");
      }).join().unwrap();
}
```

```
> Thread'den merhaba
```

Thread

```
use std::thread;

fn main() {
      let th = thread::spawn(|| {
            println!("Thread'den merhaba!");
      });

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

```
> Thread'den merhaba
```

Thread

```rust
use std::thread;

fn main() {
      let mut merhaba = String::from("Merhaba");

      let th = thread::spawn(|| {
            println!("{}", merhaba);
      });

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

Merhaba değişkeninin yaşam süresi bulunduğu scope'ta biterken thread'de hala yaşamaya devam edebilir.
Bu da bellekten kaldırılmış bir referansı kullanma sorununa yol açar.
Çözüm: "merhaba"nın sahipliğini thread'e "move" ile aktarma. (veya 'static lifetime da verilebilir.)

```
error[E0373]: closure may outlive the current function, but it borrows `merhaba`, which is owned
 by the current function
 --> thread.rs:6:25
  |
6 |     let th = thread::spawn(|| {
  |                         ^^ may outlive borrowed value `merhaba`
7 |         println!("{}", merhaba);
  |                        ------- `merhaba` is borrowed here
  |
```

Thread

```
use std::thread;

fn main() {
      let mut merhaba = String::from("Merhaba");

      let th = thread::spawn(move || {
            println!("{}", merhaba);
      });

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

```
> Merhaba
```

Atomic

Atomik yani bölünemez değişken.

Yani üzerinde aynı anda birden fazla thread işlem yapamaz.

Her thread işlem yapabilmek için değişkendeki işlemin bitmesini bekler.

Dolayısıyla thread-safe.

Atomic

```
use std::thread;
use std::sync::atomic::AtomicU32;

fn main() {
      let mut sayi = AtomicU32::new(42);

      let th = thread::spawn(move || {
            let s:&mut u32 = sayi.get_mut();
            println!("{}", *s);
      });

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

```
> 42
```

Atomic

```
use std::thread;
use std::sync::atomic::AtomicU32;

fn main() {
      let mut sayi = AtomicU32::new(42);

      let th = thread::spawn(move || {
            let s:&mut u32 = sayi.get_mut();
            *s = 5;
            println!("{}", *s);
      });

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

```
> 5
```

Arc

Atomic Reference Counted
Birden fazla thread'in aynı referansı tutmasını sağlar.

Bu referansı elinde tutan her bir thread için referans sayısını 1 arttırır.

Dolayısıyla referans sayısı = 0 olduğunda da değişken artık silinebilir demektir.

Arc

```rust
use std::thread;
use std::sync::atomic::AtomicU32;

fn main() {
	let mut sayi = AtomicU32::new(42);

	let th = thread::spawn(move || {
		let s:&mut u32 = sayi.get_mut();
		*s = 5;
		println!("{}", *s);
	});
	th.join().unwrap();

	let th2 = thread::spawn(move || {
		let s:&mut u32 = sayi.get_mut();
		*s = 5;
		println!("{}", *s);
	});
	th2.join().unwrap();
}
```

```
ef@monster-pardus:~/Belgeler/Rust/playground$ rustc thread.rs
error[E0382]: use of moved value: `sayi`
  --> thread.rs:15:26
   |
5  |     let mut sayi = AtomicU32::new(42);
   |         -------- move occurs because `sayi` has type `AtomicU32`, which
does not implement the `Copy` trait
6  |
7  |     let th = thread::spawn(move || {
   |                            ------- value moved into closure here
8  |         let a:&mut u32 = sayi.get_mut();
   |                          ---- variable moved due to use in closure
...
15 |     let th2 = thread::spawn(move || {
   |                             ^^^^^^^ value used here after move
16 |         let a:&mut u32 = sayi.get_mut();
   |                          ---- use occurs due to use in closure
```

Birden fazla thread aynı atomic değişkeni kullanamadı.
Çünkü değişkeni move ile taşıdık.

Bize başka bir yöntem lazım. Yani Arc :)

Arc

```rust
use std::thread;
use std::sync::atomic::{AtomicU32, Ordering};
use std::sync::Arc;

fn main() {
	let mut sayi = Arc::new(AtomicU32::new(42));

	let sayi1 = sayi.clone();
	let th = thread::spawn(move || {
		let a = sayi1.load(Ordering::SeqCst);
		println!("Loaded: {}", a);

		sayi1.store(a + 1, Ordering::SeqCst);
		println!("stored: {}", a+1);
	});

	let sayi2 = sayi.clone();
	let th2 = thread::spawn(move || {
		let a = sayi2.load(Ordering::SeqCst);
		println!("Loaded: {}", a);

		sayi2.store(a + 1, Ordering::SeqCst);
		println!("stored: {}", a+1);
	});

	th.join().unwrap();
	th2.join().unwrap();
}
```

```
> Loaded: 42
> stored: 43
> Loaded: 43
> stored: 44
```

Atomic

Atomic değişkenlerde load ve store iki ayrı atomik işlemdir.
Dolayısıyla ikisi arasında başka bir thread de çalışabilir.

Örneğin biz şöyle bir senaryo bekleriz:

```
Thread1 -> i = load()    // 1
Thread1 -> store(i+1)    // 2

Thread2 -> i = load()    // 2
Thread2 -> store(i+1)    // 3
```

Atomic

Fakat load ve store arasında bir boşluk olduğu için şöyle olabilir:

```
Thread1 -> x = load()    // 1

Thread2 -> y = load()    // 1
Thread2 -> store(y+1)    // 2

Thread1 -> store(x+1)    // 2
```

Görüleceği üzere iki thread bir atomik değişkeni sırayla 1 arttırmak istedi,
Fakat sonuç 3 yerine 2 oldu çünkü ikisi de değer 1 iken load yaptı ve 1 ekleyip 2 değerini kaydetti.

Atomic

Çözüm:
Birden fazla işlemin gerçekleşeceği kodlarda güvenli aralık belirlemek için Mutex kullanmak.

Veya

Bu senaryodaki gibi sadece sayı arttırmak gibi durumlarda tek bir operasyon olan
Atomic'in `fetch_add` fonksiyonunu kullanmak.

Mutex

Mutual Exclusion, yani Karşılıklı Dışlama.

Birden fazla threadin çalıştırdığı kodda güvenli bölge oluşturmaya yarar.

Güvenli bölgeye aynı anda sadece 1 thread erişebilir.

Kilitleme - Bırakma şeklinde çalışır.

Mutex

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

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

      for i in 0..100 {
            let mutex_clone = mutex.clone();
            threadler.push(thread::spawn(move || {
                  let mut sayi = mutex_clone.lock().unwrap(); // ┌──────┐
                                                              //   KİLİTLİ BÖLGE
                  *sayi += 1;                                 // └──────┘
            }));
      }

      for th in threadler {
            th.join().unwrap();
      }

      println!("final: {}", mutex.lock().unwrap());
}
```

```
> final: 100
```

Channel

Multi Producer Single Consumer(MPSC) yani çoklu üretici tekli tüketici kanallar.

Bu kanallar sayesinde birden fazla thread bir alıcıya veri gönderebilir.

Channel

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

fn main() {
	let (tx, rx) = mpsc::channel();
	let mut threadler = vec![];

	for i in 0..100 {
		let tx_clone = tx.clone();
		threadler.push(thread::spawn(move || {
			tx_clone.send(1).unwrap();
		}));
	}

	let mut sayac = 0;
	for i in 0..100 {
		sayac += rx.recv().unwrap();
	}

	for th in threadler {
		th.join().unwrap();
	}

	println!("final: {}", sayac);
}
```

```
> final: 100
```

Homeworks

1. Calculate nth prime number with threads.
    - Find every given prime number on an array &[u32] with spawning a thread for each of it, and send the result with std::sync::mpsc::Sender.