Asynchroniczność w Rust

Daniel Olczyk



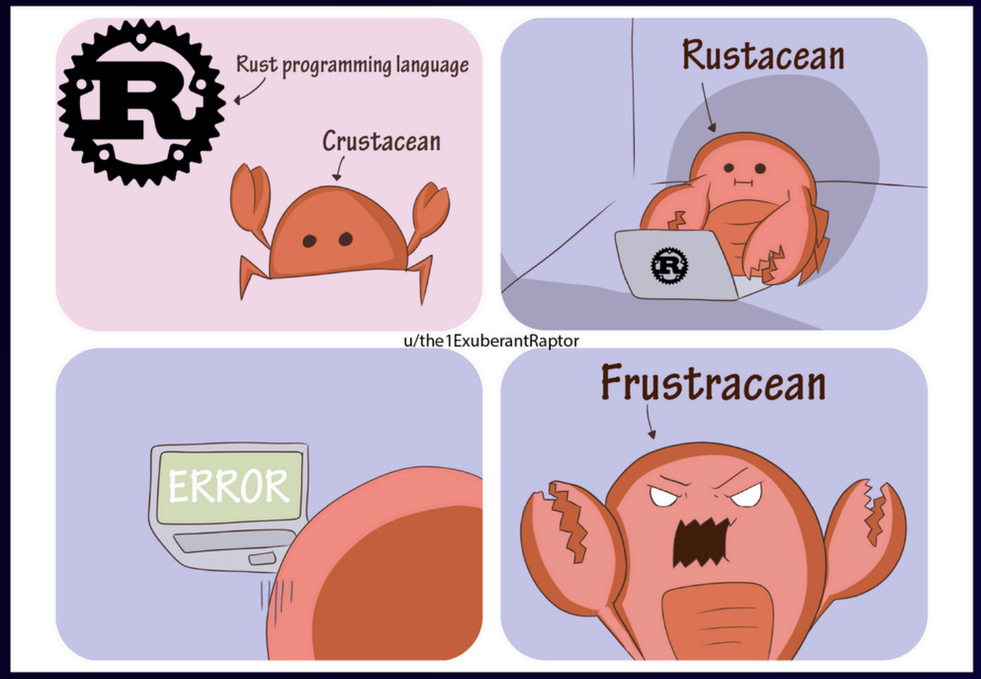
RUST LAB
SCIENCE CLUB
PJAIT

_

RUST LAB
SCIENCE CLUB
PJAIT

Co będzie?

- Co to jest ta cała asynchroniczność i jak działa
- Historia powstania asynca
- Użycie w ruście trait Future, runtime
- Tworzenie wątków
- Kontrolowanie współbieżności
- Operacje I/O
- Wykrywanie typowych błędów
- Praktyczne zadania do wykonania



RUST LAB
SCIENCE CLUB
PJAIT

O mnie słów kilka

- 3+ lat doświadczenia jako Software Engineer (.NET)
- 3 rok studiów I stopnia
- Od ponad roku działam w kole, a od 10 miesięcy jako sekretarz
- Pomagałem przy projekcie Hachi - emulator komputera CHIP-8



RUST LAB
SCIENCE CLUB
PJAIT

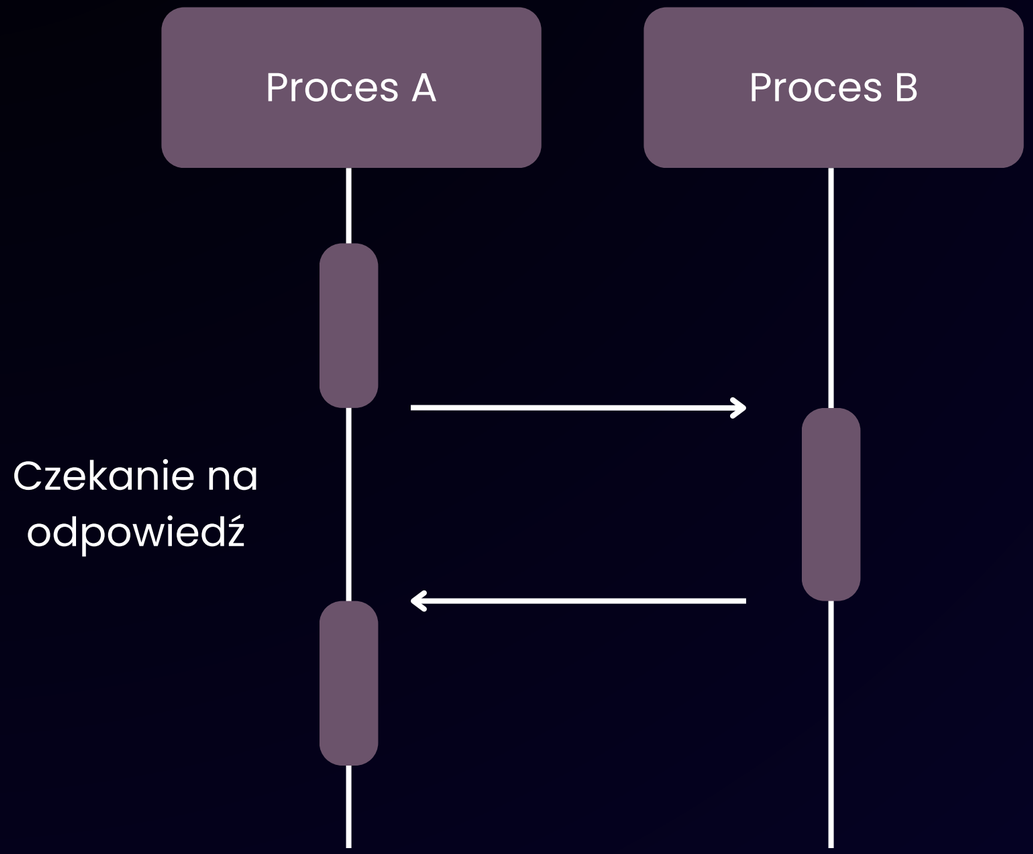
Co to jest asynchroniczność

Sposób wykonywania operacji, który nie blokuje dalszego działania programu, gdy jakaś czynność trwa dłużej niz zwykle.

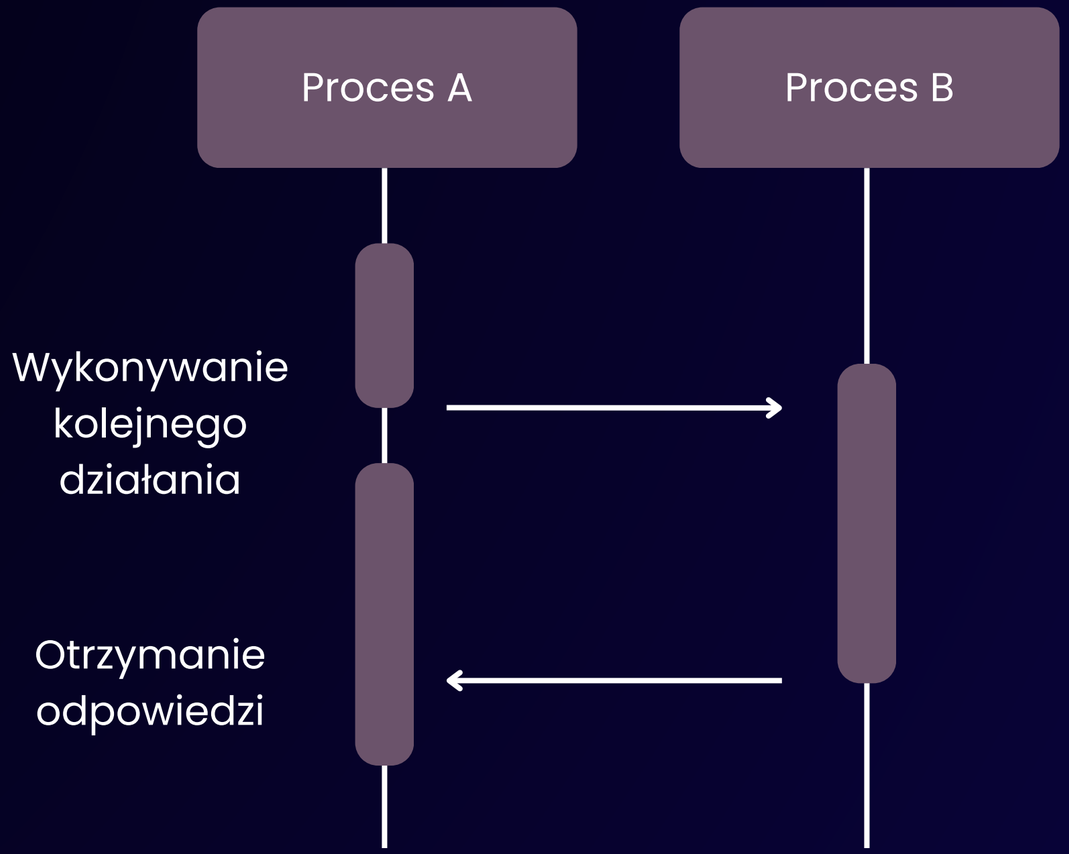


RUST LAB
SCIENCE CLUB
PJAIT

Synchroniczność



Asynchroniczność





RUST LAB
SCIENCE CLUB
PJAIT

asynchroniczność != wielowątkowość



RUST LAB
SCIENCE CLUB
PJAIT

asynchroniczność != wielowątkowość

Współbieżność



RUST LAB
SCIENCE CLUB
PJAIT

1963
Wydanie pracy wyjaśniającej “Coroutine”

2005
Microsoft wydał AJAXa, który działał w JS

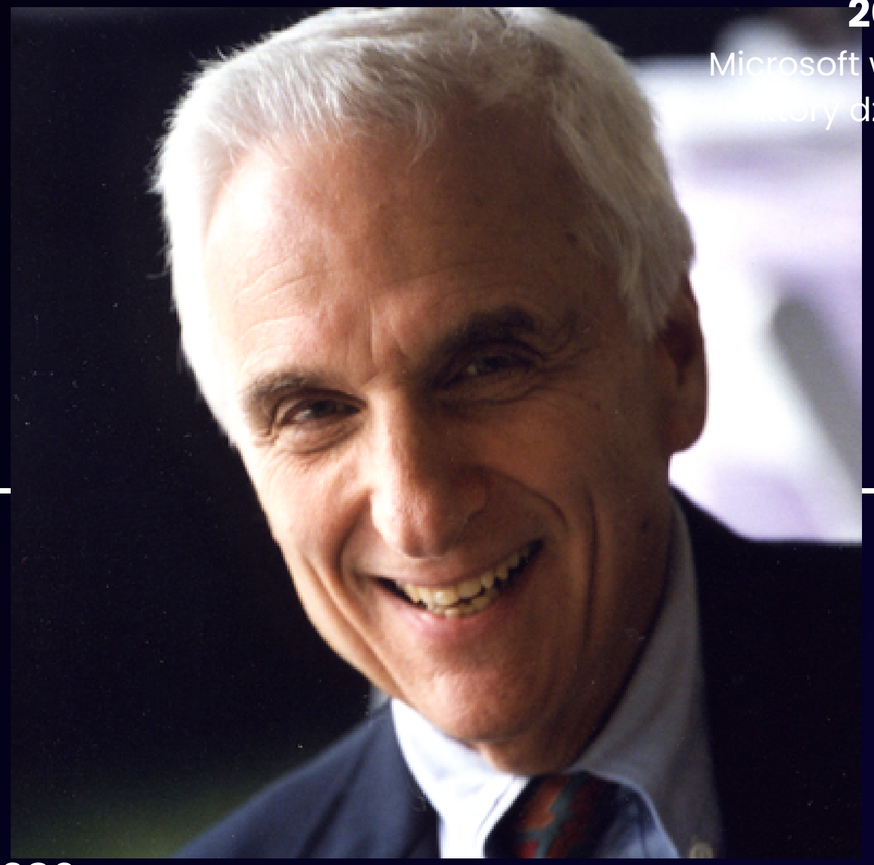
1958
Melvin Conway ukuł termin “Coroutine“

1986
Powstanie języka programowania erlang

2015
Większość popularnych języków ma już zaimplementowany model asynchroniczny

RUST LAB
SCIENCE CLUB
PJAIT

1963
Wydanie pracy wyjaśniającej “Coroutine”



2005
Microsoft wydał AJAXa, który działał w JS

1958
Melvin Conway ukuł termin “Coroutine“

1986
Powstanie języka programowania erlang

2015
Większość popularnych języków ma już zaimplementowany model asynchroniczny



RUST LAB
SCIENCE CLUB
PJAIT

1963
Wydanie pracy wyjaśniającej “Coroutine”

2005
Microsoft wydał AJAXa, który działał w JS

1958
Melvin Conway ukuł termin “Coroutine“

1986
Powstanie języka programowania erlang

2015
Większość popularnych języków ma już zaimplementowany model asynchroniczny

RUST LAB
SCIENCE CLUB
PJAIT

1963
Wydanie pracy wyjaśniającej “Coroutine”

2005
Microsoft wydał AJAXa, działał w JS



1958
Melvin Conway ukuł termin “Coroutine“

Powst
programowania erlang

2015
Większość popularnych języków ma już zaimplementowany model asynchroniczny



RUST LAB
SCIENCE CLUB
PJAIT

1963
Wydanie pracy wyjaśniającej “Coroutine”

2005
Microsoft wydał AJAXa, który działał w JS

1958
Melvin Conway ukuł termin “Coroutine“

1986
Powstanie języka programowania erlang

2015
Większość popularnych języków ma już zaimplementowany model asynchroniczny

RUST LAB
SCIENCE CLUB
PJAIT

Rodzaje modeli asynchronicznych

Task based: C#, F#
Event loop: JS, Python
Corountines: Java, Kotlin, Go
Actor/message: Erlang



RUST LAB
SCIENCE CLUB
PJAIT



Użycie w Ruście

Domyślnie rust nie narzuca pod sobą runtime, lecz udostępnia **trait Future**, gdzie zarządza się wykonywaniem kawałku kodu. Istnieje biblioteka **futures**, która rozszerza bardziej podstawowe operacje asynchroniczne.



RUST LAB
SCIENCE CLUB
PJAIT

Future w Rust

```rust
struct Countdown {
    remaining: u32,
}

impl Future for Countdown {
    type Output = String;

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        if self.remaining > 0 {
            println!("Pozostało: {}", self.remaining);
            self.remaining -= 1;

            cx.waker().wake_by_ref();

            Poll::Pending
        } else {
            Poll::Ready("Start!".to_string())
        }
    }
}
```



RUST LAB
SCIENCE CLUB
PJAIT

Najważniejsze pojęcia Future

Struktura	Co oznacza?	Ważne informacje struktury
Pin	Zarezerowanie danych w pamięci	-
Poll	Sprawdza status zadania	Poll::Ready - zadanie jest gotów, aby wykonać Poll::Pending - zadanie nie jest gotowe
Context	Mechanizm uruchamiania egzekutora	-



RUST LAB
SCIENCE CLUB
PJAIT

Rodzaje runtimemów

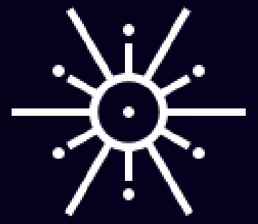
Tokio

Embassy

glommio

smol





Tokio



RUST LAB
SCIENCE CLUB
PJAIT

```toml
[dependencies]
tokio = { version = "1", features = ["full", ... ] }
```

RUST LAB
SCIENCE CLUB
PJAIT

Przykładowy kod

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

#[tokio::main]
async fn main() {
    let future = fetch_data();
    let result = future.await;
    println!("{}", result);
}
```

Czy ten kod zadziała???



RUST LAB
SCIENCE CLUB
PJAIT

Przykładowy kod

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

#[tokio::main]
async fn main() {
    let future = fetch_data();
    let result = future.await;
    println!("{}", result);
}
```

Czy ten kod zadziała???

NIE



RUST LAB
SCIENCE CLUB
PJAIT

Tworzenie wątków

```rust
tokio::spawn(async move {
    println!("Nowy wątek!");
});
```

RUST LAB
SCIENCE CLUB
PJAIT

Tworzenie wątków

std::spawn	**tokio::spawn**
Tworzy wątek systemowy	Tworzy wątek runtime
Wysokie koszty czasu/pamięci	Mniejsze koszty czasu/pamięci
Zarządzane przez kernel systemu operacyjnego	Zarządzanie przez cały runtime
Sprawdza się przy bardziej intensywnych operacjach obliczeniowych	Obliczenia I/O (Systemy plików, Sieciowe)



RUST LAB
SCIENCE CLUB
PJAIT

join!()

- Uruchamia kilka Future jednocześnie
- Czeka na wszystkie
- Zachowuje kolejność wyników

```rust
use tokio::{join, time::{Duration, sleep}};

async fn prepare_meal() -> &'static str {
    sleep(Duration::from_secs(5)).await;
    "jajecznica"
}

async fn prepare_drink() -> &'static str {
    sleep(Duration::from_secs(1)).await;
    "herbata"
}

#[tokio::main]
async fn main() {
    let (meal, drink) = join!(prepare_meal(), prepare_drink());
    println!("{} z {}", meal, drink);
}
```



RUST LAB
SCIENCE CLUB
PJAIT

select!()

- Czeka na pierwsze działanie, które się zakończy
- Reszta działań jest porzucona

<pattern> = <async expression> (, if <precondition>)? => <handler>,

else => <expression>

```rust
use tokio::{select, time::{Duration, sleep}};

struct PrinterType {
    name: &'static str,
    delay: u64
}

async fn print_page(printer_type: &str, delay: u64) {
    println!("Drukowanie rozpoczete: drukarka {}", printer_type);
    sleep(Duration::from_secs(delay)).await;
}

#[tokio::main]
async fn main() {
    let printers = vec![
        PrinterType { name: "atramentowa", delay: 10 },
        PrinterType { name: "laserowa", delay: 5 },
    ];

    select! {
        _ = print_page(printers[0].name, printers[0].delay) => {
            println!("Wydruk zakończony: {}", printers[0].name);
        }
        _ = print_page(printers[1].name, printers[1].delay)=> {
            println!("Zadanie zakończone: {}", printers[1].name);
        }
    }
}
```



RUST LAB
SCIENCE CLUB
PJAIT

try_join!()

- Uruchamia kilka Future jednocześnie
- Czeka na wszystkie
- Zachowuje kolejność wyników
- Posiada obsługę błędów

```rust
use tokio::try_join;

async fn fetch_user(id: u32) -> Result<String, String> {
    Ok(format!("Użytkownik #{id}"))
}

async fn fetch_orders(_id: u32) -> Result<Vec<String>, String> {
    Ok(vec!["Zamówienie #1".into(), "Zamówienie #2".into()])
}

async fn fetch_bonuses(_id: u32) -> Result<Vec<String>, String> {
    Err("Błąd podczas pobierania bonusów".into())
}

#[tokio::main]
async fn main() -> Result<(), String> {
    let (
        user,
        orders,
        bonuses)
        = try_join!(
        fetch_user(1),
        fetch_orders(1),
        fetch_bonuses(1)
    )?;

    println!("Użytkownik: {user}");
    println!("Zamówienia: {orders:?}");
    println!("Bonusy: {bonuses:?}");

    Ok(())
}
```

RUST LAB
SCIENCE CLUB
PJAIT

Arc i Mutex

Struktura	Co oznacza?	Gdzie użyć?
Arc<T>	Wielu właścicieli / wiele wątków	Gdy potrzebujemy przerzucać strukturę do wielu funkcji
Mutex<T>	Możliwość modyfikacji na wielu wątkach	Kiedy chcemy wykonywać opecjację zapisywania przy wielu wątkach
Arc<Mutex<T>>	Połączenie dwóch wyżej struktur	Standardowy sposób na bezpieczną modyfikację danych z wielu wątków.

RUST LAB
SCIENCE CLUB
PJAIT

Przykład kodu

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

use tokio::{select, sync::Mutex, time::{Duration, sleep}};

async fn increment(val: Arc<Mutex<i32>>) {
    loop {
        let mut v = val.lock().await;
        *v += 1;
    }
}

#[tokio::main]
async fn main() {
    let val = Arc::new(Mutex::new(0));

    select! {
        _ = increment(val.clone()) => {
            println!("Inkrementacja zwiększona");
        }
        _ = sleep(Duration::from_secs(3)) => {
            println!("Czas minął");
        }
    }

    println!("Wynik: {}", val.lock().await);
}
```



RUST LAB
SCIENCE CLUB
PJAIT

Operacje I/O

- **AsyncRead**, **AsyncWrite**- traity, które mają zaimplementowaną podstawową logikę pod operację asynchroniczne (niskopoziomowe)
- **AsyncReadExt**, **AsyncWriteExt** - trai basujący na **AsyncRead, AsyncWrite**, który ma podstawowe funkcje do operowania pod różne implementacje źródeł takie jak: **read, write etc.** (wysokopoziomowe)
- Przykładowe implementacje: File, Channel, HttpClient



RUST LAB
SCIENCE CLUB
PJAIT

Przykładowy kod I/O

```rust
// https://tokio.rs/tokio/tutorial/io
use tokio::fs::File;
use tokio::io::{self, AsyncReadExt};

#[tokio::main]
async fn main() -> io::Result<()> {
    let mut f = File::open("foo.txt").await?;
    let mut buffer = [0; 10];

    // read up to 10 bytes
    let n = f.read(&mut buffer[..]).await?;

    println!("The bytes: {:?}", &buffer[..n]);
    Ok(())
}
```

```rust
// https://tokio.rs/tokio/tutorial/io
use tokio::io::{self, AsyncWriteExt};
use tokio::fs::File;

#[tokio::main]
async fn main() -> io::Result<()> {
    let mut file = File::create("foo.txt").await?;

    let n = file.write(b"some bytes").await?;

    println!("Wrote the first {} bytes of 'some bytes'.", n);
    Ok(())
}
```



RUST LAB
SCIENCE CLUB
PJAIT

Zabawmy się w “Gdzie jest haczyk?”



RUST LAB
SCIENCE CLUB
PJAIT

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

#[tokio::main]
async fn main() {
    let task1 = tokio::spawn(async {
        println!("Zadanie 1: Start (ciężka praca)");
        thread::sleep(Duration::from_secs(3));
        println!("Zadanie 1: Koniec");
    });

    let task2 = tokio::spawn(async {
        println!("Zadanie 2: Start (szybkie)");
        tokio::time::sleep(Duration::from_secs(1)).await;
        println!("Zadanie 2: Koniec");
    });

    let _ = tokio::join!(task1, task2);
}
```



RUST LAB
SCIENCE CLUB
PJAIT

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

#[tokio::main]
async fn main() {
    let task1 = tokio::spawn(async {
        println!("Zadanie 1: Start (ciężka praca)");
        thread::sleep(Duration::from_secs(3));
        println!("Zadanie 1: Koniec");
    });

    let task2 = tokio::spawn(async {
        println!("Zadanie 2: Start (szybkie)");
        tokio::time::sleep(Duration::from_secs(1)).await;
        println!("Zadanie 2: Koniec");
    });

    let _ = tokio::join!(task1, task2);
}
```

NIE

W task1 używa on sleep z std::thread co do końca nie zadziała z naszym kodem asynchronicznym



RUST LAB
SCIENCE CLUB
PJAIT

```rust
use std::sync::{Arc, Mutex};
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
    let counter = Arc::new(Mutex::new(0));

    let counter_cloned = counter.clone();
    tokio::spawn(async move {
        let mut lock = counter_cloned.lock().unwrap();

        *lock += 1;

        sleep(Duration::from_secs(1)).await;

        println!("Nowa wartość: {}", *lock);
    });

    sleep(Duration::from_secs(2)).await;
}
```



RUST LAB
SCIENCE CLUB
PJAIT

```rust
use std::sync::{Arc, Mutex};
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
    let counter = Arc::new(Mutex::new(0));

    let counter_cloned = counter.clone();
    tokio::spawn(async move {
        let mut lock = counter_cloned.lock().unwrap();

        *lock += 1;

        sleep(Duration::from_secs(1)).await;

        println!("Nowa wartość: {}", *lock);
    });

    sleep(Duration::from_secs(2)).await;
}
```

NIE

Licznik jest zablokowany w jednym wątku, co uniemożliwia modyfikację w innych wątkach



RUST LAB
SCIENCE CLUB
PJAIT

```rust
struct Database {
    tags: Mutex<Vec<String>>,
}

impl Database {
    async fn add_record(&self, tag: String) {
        let mut lock = self.tags.lock().await;
        lock.push(tag);
    }

    async fn add_tags(&self, tags: Vec<String>) {
        let _lock = self.tags.lock().await;
        println!("Stan aktualny tagów: {:?}", *_lock);

        for t in tags {
            self.add_record(t).await;
        }

        println!("Stan po dodaniu wiele tagów: {:?}", *_lock);
    }
}
```



RUST LAB
SCIENCE CLUB
PJAIT

```rust
struct Database {
    tags: Mutex<Vec<String>>,
}

impl Database {
    async fn add_record(&self, tag: String) {
        let mut lock = self.tags.lock().await;
        lock.push(tag);
    }

    async fn add_tags(&self, tags: Vec<String>) {
        let _lock = self.tags.lock().await;
        println!("Stan aktualny tagów: {:?}", *_lock);

        for t in tags {
            self.add_record(t).await;
        }

        println!("Stan po dodaniu wiele tagów: {:?}", *_lock);
    }
}
```

NIE

Próba odblokowania Mutexa w add_record(), gdy próbujemy wywołać z funkcji add_tags() powoduje, że program się blokuję tworząc tzw. Deadlock



RUST LAB
SCIENCE CLUB
PJAIT

- Skalowalność
- Responsywność w UI
- Możliwość łączenia wielu wyników operacji

Gdzie należy używać



RUST LAB
SCIENCE CLUB
PJAIT

Gdzie nie należy używać

- Operacje CPU-bound
- Systemy RT
- Inicjalizacja/konfiguracja systemu

Dzięki!



RUST LAB
SCIENCE CLUB
PJAIT

_