Лекція 16

Async/Await: Концепції

Асинхронне програмування в Rust

async • await • Future • Poll • Pin • Runtime

🤖 Масштабування: тисячі агентів в одному процесі

Частина 1: Теорія та концепції

О. С. Бичков • 2025

План лекції (Частина 1)

1. Чому async? Проблема масштабування
2. Синхронний vs Асинхронний код
3. Threads vs Async — порівняння
4. Concurrency vs Parallelism
5. async fn та .await
6. Future trait
7. Poll enum
8. Lazy futures
9. State machine transformation
10. Pin<T> та чому потрібен
11. Unpin trait
12. async блоки
13. async closures
14. 🤖 Async агент
15. 🤖 Паралельні операції
16. Коли async, коли threads

Частина 2: Runtime, Tokio, практичні патерни

Проблема: масштабування потоків

Рій з 10,000 агентів = 10,000 потоків?

Проблеми з потоками:
- Кожен потік ~2-8 MB стека
- 10,000 потоків = 20-80 GB RAM тільки на стеки!
- Context switching overhead
- Обмеження ОС на кількість потоків

Потоків	RAM (стеки)	Context switches
100	200-800 MB	Низько
1,000	2-8 GB	Помітно
10,000	20-80 GB	Критично
100,000	200-800 GB	Неможливо

Рішення: Async/Await

Async дозволяє мати тисячі конкурентних задач
на ОДНОМУ або кількох потоках!

Як це працює?
- async fn не блокує потік при очікуванні
- Runtime переключається на іншу задачу
- Коли I/O готовий — продовжує виконання

Threads (OS)

```
Thread 1: [████████████]
Thread 2: [████████████]
Thread 3: [████████████]

3 потоки = 3 задачі
```

Async (Runtime)

```
Thread 1: [A][B][A][C][B][A][C]
          ↑ runtime переключає

1 потік = багато задач!
```

```rust
// Асинхронний код — НЕ блокує потік
async fn fetch_data_async(url: &str) -> String {
    let response = http_client.get(url).send().await;  // Відпускає потік!
    response.text().await  // Відпускає потік!
}

// Використання — паралельно:
let (data1, data2) = tokio::join!(
    fetch_data_async("url1"),
    fetch_data_async("url2"),
);
// Загалом: ~100ms (паралельно!)
```

Threads vs Async — детальне порівняння

Аспект	Threads	Async
Scheduling	OS (preemptive)	Runtime (cooperative)
Memory/task	2-8 MB стека	Кілька KB
Creation time	~20-50 мкс	~нс
Max tasks	~10K (OS limit)	~1M+
Best for	CPU-bound	I/O-bound
Complexity	Простіше	Складніше
Blocking calls	OK	Заборонено!
Cancellation	Складно	Просто (drop)

Правило: Threads для CPU-bound, Async для I/O-bound задач

Concurrency vs Parallelism

Concurrency (конкурентність)

Структура програми:
кілька задач можуть виконуватись
"одночасно" (перемикаючись)

Core 1: [A][B][A][B][A]

Один ресурс, багато задач
Async — про concurrency!

Parallelism (паралелізм)

Виконання програми:
кілька задач реально виконуються
в один момент часу

Core 1: [AAAAAAA]
Core 2: [BBBBBBB]

Багато ресурсів, багато задач
Threads — про parallelism!

Async з multi-threaded runtime = concurrency + parallelism!

```rust
// async fn — функція, що повертає Future
async fn hello() -> String {
    "Hello, async world!".to_string()
}

// Еквівалентно:
fn hello() -> impl Future<Output = String> {
    async {
        "Hello, async world!".to_string()
    }
}

// Виклик async fn НЕ виконує код!
let future = hello();  // Тільки створює Future
// Код всередині hello() ще не виконався!

// .await виконує Future
let result = future.await;  // Тепер виконується!
println!("{}", result);  // "Hello, async world!"

// Або одразу:
let result = hello().await;
```

```rust
async fn process_data() -> Result<Data, Error> {
    // .await можна використовувати тільки в async контексті!

    // Кожен .await — точка, де потік може переключитись
    let connection = connect_to_db().await?;   // Очікуємо з'єднання
    let data = connection.query("SELECT *").await?;  // Очікуємо запит
    let processed = transform(data).await?;    // Очікуємо обробку

    Ok(processed)
}

// .await:
// 1. Перевіряє чи Future готовий
// 2. Якщо готовий — отримує результат
// 3. Якщо не готовий — "відпускає" потік
// 4. Runtime запам'ятовує стан і продовжить пізніше

// ? працює з await:
let result = some_async_fn().await?;  // Propagate error
```

```rust
// Спрощений Future trait
pub trait Future {
    type Output;  // Тип результату

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

// Poll enum
pub enum Poll<T> {
    Ready(T),     // Результат готовий
    Pending,      // Ще не готовий, спробуйте пізніше
}

// Коли ви пишете:
async fn example() -> i32 {
    42
}

// Компілятор генерує щось на кшталт:
struct ExampleFuture { /* state */ }

impl Future for ExampleFuture {
    type Output = i32;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<i32> {
        Poll::Ready(42)
    }
}
```

```rust
// Коли Future повертає Pending, runtime має знати
// коли спробувати знову — це робить Waker

impl Future for MyFuture {
    type Output = String;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<String> {
        if self.is_ready() {
            Poll::Ready(self.get_result())
        } else {
            // Зберігаємо waker щоб "розбудити" коли готово
            let waker = cx.waker().clone();

            // Реєструємо callback (наприклад, для I/O)
            register_io_callback(move || {
                waker.wake();  // Сигнал runtime: "Я готовий!"
            });

            Poll::Pending  // Поки не готовий
        }
    }
}

// Runtime:
// 1. poll() -> Pending
// 2. Переходить до іншої задачі
// 3. Waker.wake() викликано
// 4. Runtime повертається і poll() знову
```

```rust
async fn important_work() -> i32 {
    println!("Starting work...");
    expensive_computation();
    println!("Work done!");
    42
}

fn main() {
    // ❌ Нічого не виведеться!
    let future = important_work();
    // Future створений, але код не виконувався!

    println!("Future created");

    // Без .await — робота ніколи не виконається
    // future просто буде dropped
}
```

Важливо: async fn без .await = нічого не робить!

```rust
// Output:
// Future created
// (нічого більше!)

// Rust futures є "lazy" — виконуються тільки при poll()
// На відміну від JavaScript Promises, які eager
```

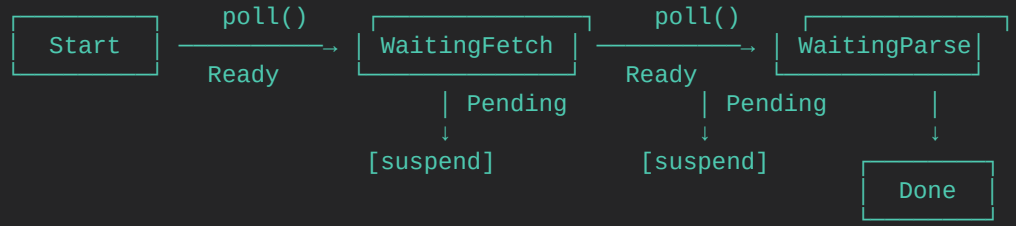
```
// Ваш код:
async fn example() -> i32 {
    let a = step1().await;
    let b = step2(a).await;
    a + b
}

// Компілятор перетворює в state machine:
enum ExampleStateMachine {
    Start,
    WaitingStep1 { step1_future: Step1Future },
    WaitingStep2 { a: i32, step2_future: Step2Future },
    Done,
}

impl Future for ExampleStateMachine {
    fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<i32> {
        loop {
            match self.state {
                Start => {
                    self.state = WaitingStep1 { step1_future: step1() };
                }
                WaitingStep1 { step1_future } => {
                    match step1_future.poll(cx) {
                        Poll::Ready(a) => self.state = WaitingStep2 { a, step2_future: step2(a) },
                        Poll::Pending => return Poll::Pending,
                    }
                }
                // ...
            }
```

Візуалізація State Machine

```
async fn download_and_process(url: &str) -> Data {
    let response = fetch(url).await;      // State 1 → 2
    let data = parse(response).await;     // State 2 → 3
    process(data)                         // State 3 → Done
}
```



Кожен .await = потенційна точка призупинення та відновлення

```rust
async fn problematic() {
    let data = vec![1, 2, 3];
    let reference = &data[0];  // Посилання на data

    some_async_op().await;  // ← Тут структура може бути переміщена!

    println!("{}", reference);  // reference вказує на старе місце!
}

// State machine містить:
struct ProblematicFuture {
    data: Vec<i32>,
    reference: *const i32,  // Вказує на data!
}

// Якщо структуру перемістити в пам'яті:
// - data переїде на нову адресу
// - reference все ще вказує на стару!
// - Dangling pointer! 💥
```

Pin гарантує: "Ця структура не буде переміщена в пам'яті"

```rust
use std::pin::Pin;

// Pin<P> — обгортка над pointer P (Box, &mut, etc.)
// Гарантує, що значення за pointer не буде moved

// Створення pinned value:
let mut data = Box::pin(MyFuture::new());

// Pin<&mut T> — pinned mutable reference
let pinned: Pin<&mut MyFuture> = data.as_mut();

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

// Чому Pin а не просто &mut Self?
// Бо &mut T дозволяє std::mem::swap, std::mem::replace
// Які переміщують значення!

let mut a = MyFuture::new();
let mut b = MyFuture::new();
std::mem::swap(&mut a, &mut b);  // Переміщення!

// З Pin — swap неможливий (без unsafe)
```

```
// Unpin — marker trait: "Мене можна безпечно переміщувати навіть в Pin"

// Більшість типів — Unpin автоматично:
// i32: Unpin ✓
// String: Unpin ✓
// Vec<T>: Unpin ✓

// Типи що НЕ Unpin:
// - Futures створені async fn (можуть бути self-referential)
// - Типи з PhantomPinned

use std::marker::PhantomPinned;

struct NotUnpin {
    data: String,
    _pin: PhantomPinned,  // Робить структуру !Unpin
}

// Для Unpin типів Pin не дає обмежень:
let mut x: Pin<&mut i32> = Pin::new(&mut 42);
let value: &mut i32 = Pin::get_mut(x);  // Можна отримати &mut

// Для !Unpin — тільки unsafe:
let pinned_future: Pin<&mut MyFuture> = ...;
// let inner = Pin::get_mut(pinned_future);  // ❌ Error!
```

```rust
// async блок — створює Future inline
let future = async {
    let x = compute().await;
    x + 1
};

// Корисно для closures та inline futures:
let futures: Vec<_> = urls.iter()
    .map(|url| async move {
        fetch(url).await
    })
    .collect();

// async move — захоплює змінні по ownership
let data = vec![1, 2, 3];
let future = async move {
    // data moved into future
    process(data).await
};
// data більше недоступна тут

// async без move — захоплює по reference
let data = vec![1, 2, 3];
let future = async {
    // &data
    println!("{:?}", data);
};
// data все ще доступна
```

```rust
// Стабільний спосіб — повертати async block:
let fetch_url = |url: String| async move {
    reqwest::get(&url).await
};

// Або з явним типом:
fn make_fetcher() -> impl Fn(String) -> impl Future<Output = Response> {
    |url| async move {
        reqwest::get(&url).await.unwrap()
    }
}

// async closures (unstable, nightly):
// #![feature(async_closure)]
// let fetch = async |url: String| {
//     reqwest::get(&url).await
// };

// Для trait bounds:
fn take_async_fn<F, Fut>(f: F)
where
    F: Fn(i32) -> Fut,
    Fut: Future<Output = String>,
{
    // ...
}
```

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

/// Async агент — не блокує потік при очікуванні
async fn agent_loop(id: u32) {
    println!("Agent {} started", id);

    loop {
        // Отримання команди (async)
        let command = receive_command().await;

        // Обробка команди
        match command {
            Command::MoveTo(pos) => {
                move_to(pos).await;  // Async рух
            }
            Command::Scan => {
                let results = scan_area().await;  // Async сканування
                report_results(results).await;
            }
            Command::Shutdown => break,
        }

        // Async затримка (не блокує потік!)
        tokio::time::sleep(Duration::from_millis(100)).await;
    }

    println!("Agent {} stopped", id);
}

// 1000 таких агентів можуть працювати на 1 потоці!
```

```rust
use futures::future::join_all;

/// Запуск багатьох агентів одночасно
async fn run_swarm(agent_count: usize) {
    // Створюємо futures для всіх агентів
    let agent_futures: Vec<_> = (0..agent_count)
        .map(|id| agent_loop(id as u32))
        .collect();

    // join_all — чекає на ВСІ futures
    join_all(agent_futures).await;

    println!("All agents completed");
}

/// Паралельне сканування кількома агентами
async fn parallel_scan(agents: &[Agent], areas: &[Area]) -> Vec<ScanResult> {
    let scan_futures: Vec<_> = agents.iter()
        .zip(areas.iter())
        .map(|(agent, area)| agent.scan(area))
        .collect();

    // Всі скани виконуються "паралельно"
    join_all(scan_futures).await
}

// 10,000 агентів на кількох потоках!
```

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

/// Агент чекає на команду АБО timeout
async fn agent_wait_for_command(rx: &mut Receiver<Command>) -> Option<Command> {
    select! {
        cmd = rx.recv() => cmd,
        _ = tokio::time::sleep(Duration::from_secs(5)) => {
            println!("Timeout waiting for command");
            None
        }
    }
}

/// Перший агент, що знайде ціль
async fn race_to_find_target(agents: Vec<Agent>) -> Option<Target> {
    // select! на динамічній кількості futures
    let mut futures: Vec<_> = agents.iter()
        .map(|a| Box::pin(a.search_for_target()))
        .collect();

    // futures::select_all — перший ready
    let (result, _index, _remaining) = futures::future::select_all(futures).await;
    result
}

// select! — виконує ПЕРШИЙ готовий, скасовує інші
```

Коли async, коли threads?

Async краще для:

- I/O-bound операції
  - Network requests
  - File I/O
  - Database queries
- Багато конкурентних задач
- Web servers
- Chat servers
- 🤖 MAC з тисячами агентів що чекають на події

Threads краще для:

- CPU-bound операції
  - Обчислення
  - Image processing
  - Crypto
- Blocking I/O (legacy APIs)
- Parallelism (multi-core)
- Простіший код
- 🤖 Важкі обчислення для кожного агента

Часто комбінація: async для I/O, spawn_blocking для CPU

Підсумок: Частина 1

Async/Await — cooperative concurrency:
• async fn повертає Future
• .await виконує Future
• Futures lazy — без await нічого не робиться

Future trait:
• poll() -> Poll::Ready(T) | Poll::Pending
• Waker сигналізує коли ready

State Machine:
• Компілятор перетворює async fn в state machine
• Кожен .await = точка призупинення

Pin<T>:
• Гарантує що значення не буде moved
• Потрібен для self-referential futures

→ Частина 2: Tokio runtime, практичні патерни

🤖 MAC: тисячі агентів на кількох потоках!

Лекція 16 (продовження)

Async Runtime та практика

Tokio, spawn, join, select

Runtime • tokio::spawn • join! • select! • timeout

🤖 Асинхронний рій агентів: практична реалізація

Частина 2: Runtime та патерни

О. С. Бичков • 2025

План лекції (Частина 2)

1. Що таке Runtime?
2. Tokio — найпопулярніший runtime
3. #[tokio::main]
4. tokio::spawn — задачі
5. JoinHandle для async
6. join! — паралельне виконання
7. try_join! — з обробкою помилок
8. select! — перший готовий
9. tokio::time — таймери
10. timeout — обмеження часу
11. spawn_blocking — CPU tasks
12. Cancellation
13. 🤖 Async координатор
14. 🤖 Async агент повністю
15. 🤖 Async рій
16. Типові помилки

Що таке Async Runtime?

Future сам по собі нічого не робить!
Потрібен Runtime для виконання:

- Викликає poll() на futures
- Керує Wakers
- Планує задачі (scheduling)
- Інтегрується з OS для I/O events

Популярні runtimes:

- Tokio — найпопулярніший
- async-std — схожий API на std
- smol — мінімалістичний
- embassy — для embedded

Rust НЕ має runtime!

На відміну від Go, Node.js
Rust не включає runtime.

Ви обираєте або пишете свій.
Це дає гнучкість і контроль.

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

// Features:
// - rt: runtime
// - rt-multi-thread: multi-threaded runtime
// - sync: async synchronization primitives
// - time: timers, sleep, timeout
// - io-util: async I/O utilities
// - net: async TCP/UDP
// - fs: async file system
// - macros: #[tokio::main], #[tokio::test]
// - full: все вище
```

Tokio використовують:

- AWS, Microsoft, Discord, Cloudflare
- Більшість production async Rust

```rust
// #[tokio::main] створює runtime і запускає async main

#[tokio::main]
async fn main() {
    println!("Hello from async main!");

    // Тепер можна використовувати .await
    let result = some_async_function().await;
    println!("Result: {:?}", result);
}

// Еквівалентно:
fn main() {
    let rt = tokio::runtime::Runtime::new().unwrap();
    rt.block_on(async {
        println!("Hello from async main!");
        let result = some_async_function().await;
        println!("Result: {:?}", result);
    });
}

// Налаштування runtime:
#[tokio::main(flavor = "multi_thread", worker_threads = 4)]
async fn main() { /* ... */ }

#[tokio::main(flavor = "current_thread")]  // Single-threaded
async fn main() { /* ... */ }
```

```rust
use tokio::task::JoinHandle;

#[tokio::main]
async fn main() {
    // spawn — створює нову async задачу
    // Задача виконується КОНКУРЕНТНО з іншим кодом
    let handle: JoinHandle<i32> = tokio::spawn(async {
        println!("Task started");
        expensive_async_operation().await;
        println!("Task completed");
        42  // Повертаємо результат
    });

    // Головний код продовжує виконуватись
    println!("Main continues...");
    do_other_work().await;

    // Чекаємо на завершення задачі
    let result = handle.await.unwrap();  // Result<T, JoinError>
    println!("Task returned: {}", result);
}

// spawn вимагає 'static — задача не може borrowing!
// Використовуйте move або Arc для даних
```

```rust
// ❌ Помилка: data не 'static
let data = vec![1, 2, 3];
tokio::spawn(async {
    println!("{:?}", data);  // Error: borrowed data
});

// ✓ 3 move
let data = vec![1, 2, 3];
tokio::spawn(async move {
    println!("{:?}", data);  // OK: owned data
});

// ✓ 3 Arc для shared ownership
use std::sync::Arc;
let data = Arc::new(vec![1, 2, 3]);
let data_clone = Arc::clone(&data);

tokio::spawn(async move {
    println!("{:?}", data_clone);
});

// data все ще доступна в main
println!("Original: {:?}", data);

// Чому 'static? Бо spawn'ed task може пережити поточний scope
```

```rust
use tokio::join;

async fn fetch_user(id: u32) -> User { /* ... */ }
async fn fetch_posts(user_id: u32) -> Vec<Post> { /* ... */ }
async fn fetch_comments(user_id: u32) -> Vec<Comment> { /* ... */ }

#[tokio::main]
async fn main() {
    let user_id = 42;

    // ❌ Послідовно — повільно!
    // let user = fetch_user(user_id).await;       // 100ms
    // let posts = fetch_posts(user_id).await;     // 100ms
    // let comments = fetch_comments(user_id).await; // 100ms
    // Total: 300ms

    // ✓ Паралельно з join!
    let (user, posts, comments) = join!(
        fetch_user(user_id),
        fetch_posts(user_id),
        fetch_comments(user_id),
    );
    // Total: ~100ms (максимум з трьох)

    println!("User: {:?}", user);
    println!("Posts: {}", posts.len());
    println!("Comments: {}", comments.len());
}
```

```rust
use tokio::try_join;

async fn fetch_user(id: u32) -> Result<User, Error> { /* ... */ }
async fn fetch_posts(id: u32) -> Result<Vec<Post>, Error> { /* ... */ }

#[tokio::main]
async fn main() -> Result<(), Error> {
    let user_id = 42;

    // try_join! — повертає Result
    // Якщо БУДЬ-ЯКА задача поверне Err — весь try_join! = Err
    // Інші задачі СКАСОВУЮТЬСЯ!

    let (user, posts) = try_join!(
        fetch_user(user_id),
        fetch_posts(user_id),
    )?;  // Propagate error

    println!("User: {:?}, Posts: {}", user, posts.len());
    Ok(())
}

// join! vs try_join!:
// join!     — всі задачі завершуються, навіть з Err
// try_join! — перша Err скасовує інші
```

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

#[tokio::main]
async fn main() {
    let mut interval = tokio::time::interval(Duration::from_secs(1));

    loop {
        select! {
            // Біас: перша гілка перевіряється першою
            _ = interval.tick() => {
                println!("Tick!");
            }

            result = some_async_operation() => {
                println!("Operation completed: {:?}", result);
                break;
            }

            // Timeout branch
            _ = sleep(Duration::from_secs(10)) => {
                println!("Timeout!");
                break;
            }
        }
    }
}

// select! виконує ПЕРШИЙ готовий branch
// Інші futures СКАСОВУЮТЬСЯ (dropped)
```

```rust
use tokio::sync::mpsc;
use tokio::select;

async fn worker(
    mut commands: mpsc::Receiver<Command>,
    mut shutdown: mpsc::Receiver<()>,
) {
    loop {
        select! {
            // Biased — пріоритет shutdown
            biased;

            _ = shutdown.recv() => {
                println!("Shutdown received");
                break;
            }

            Some(cmd) = commands.recv() => {
                println!("Processing command: {:?}", cmd);
                process_command(cmd).await;
            }

            else => {
                // Всі канали закриті
                println!("All channels closed");
                break;
            }
        }
    }
}
```

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

#[tokio::main]
async fn main() {
    // sleep — async затримка (НЕ блокує потік!)
    sleep(Duration::from_secs(1)).await;

    // interval — періодичний таймер
    let mut interval = interval(Duration::from_millis(100));

    for _ in 0..5 {
        interval.tick().await;  // Чекаємо наступний tick
        println!("Tick at {:?}", Instant::now());
    }

    // Instant — час
    let start = Instant::now();
    some_operation().await;
    println!("Took: {:?}", start.elapsed());

    // sleep_until — до конкретного часу
    let deadline = Instant::now() + Duration::from_secs(5);
    tokio::time::sleep_until(deadline).await;
}

// НЕ використовуйте std::thread::sleep в async!
// Це ЗАБЛОКУЄ потік runtime!
```

```rust
use tokio::time::{timeout, Duration};

#[tokio::main]
async fn main() {
    // timeout — обгортає future з обмеженням часу
    let result = timeout(
        Duration::from_secs(5),
        slow_operation()
    ).await;

    match result {
        Ok(value) => println!("Got: {:?}", value),
        Err(_) => println!("Operation timed out!"),
    }

    // Або з ?
    let value = timeout(Duration::from_secs(5), slow_operation())
        .await
        .map_err(|_| MyError::Timeout)?;

    // timeout_at — до конкретного часу
    use tokio::time::Instant;
    let deadline = Instant::now() + Duration::from_secs(10);
    let result = tokio::time::timeout_at(deadline, operation()).await;
}

// Якщо timeout — внутрішній future СКАСОВУЄТЬСЯ (dropped)
```

```rust
use tokio::task;

#[tokio::main]
async fn main() {
    // ❌ НЕ робіть CPU-bound роботу в async!
    // let result = heavy_computation();  // Блокує runtime!

    // ✓ Використовуйте spawn_blocking
    let result = task::spawn_blocking(|| {
        // Виконується на окремому thread pool
        heavy_computation()
    }).await.unwrap();

    println!("Result: {:?}", result);

    // spawn_blocking для sync I/O:
    let contents = task::spawn_blocking(|| {
        std::fs::read_to_string("large_file.txt")  // Sync I/O
    }).await.unwrap()?;

    // block_in_place — для поточного потоку (multi-thread runtime)
    let result = task::block_in_place(|| {
        heavy_computation()
    });
}

// spawn_blocking pool за замовчуванням = 512 threads
```

```rust
use tokio::task::JoinHandle;

#[tokio::main]
async fn main() {
    let handle: JoinHandle<()> = tokio::spawn(async {
        loop {
            println!("Working...");
            tokio::time::sleep(Duration::from_secs(1)).await;
        }
    });

    // Чекаємо трохи
    tokio::time::sleep(Duration::from_secs(3)).await;

    // Скасовуємо задачу
    handle.abort();  // Задача буде скасована

    // Або просто drop handle — задача продовжиться у фоні
    // drop(handle);  // Detached task

    // Перевірка результату після abort
    match handle.await {
        Ok(_) => println!("Task completed"),
        Err(e) if e.is_cancelled() => println!("Task was cancelled"),
        Err(e) => println!("Task panicked: {:?}", e),
    }
}

// Cancellation у Rust — cooperative
// Задача скасовується на наступному .await
```

```rust
use tokio::sync::mpsc;
use tokio::select;

struct AsyncCoordinator {
    agents: HashMap<AgentId, AgentHandle>,
    from_agents: mpsc::Receiver<AgentMessage>,
    shutdown: tokio::sync::broadcast::Receiver<()>,
}

impl AsyncCoordinator {
    async fn run(mut self) {
        println!("[Coordinator] Started");

        loop {
            select! {
                biased;

                _ = self.shutdown.recv() => {
                    println!("[Coordinator] Shutdown");
                    self.broadcast_shutdown().await;
                    break;
                }

                Some(msg) = self.from_agents.recv() => {
                    self.handle_message(msg).await;
                }

                else => break,
            }
        }
```

```
struct AsyncAgent {
    id: AgentId,
    position: Position,
    to_coordinator: mpsc::Sender<AgentMessage>,
    from_coordinator: mpsc::Receiver<Command>,
}

impl AsyncAgent {
    async fn run(mut self) {
        let mut tick_interval = tokio::time::interval(Duration::from_millis(100));

        loop {
            select! {
                _ = tick_interval.tick() => {
                    self.tick().await;
                }

                Some(cmd) = self.from_coordinator.recv() => {
                    if matches!(cmd, Command::Shutdown) {
                        break;
                    }
                    self.handle_command(cmd).await;
                }
            }
        }
        println!("[Agent {}] Stopped", self.id);
    }

    async fn tick(&mut self) {
        // Async операції: сканування, рух
```

```rust
use tokio::task::JoinSet;

#[tokio::main]
async fn main() {
    println!("=== Async Swarm ===");

    let (coord_tx, coord_rx) = mpsc::channel(1000);
    let (shutdown_tx, _) = tokio::sync::broadcast::channel(1);

    // JoinSet — колекція задач
    let mut tasks = JoinSet::new();

    // Spawn координатора
    let coordinator = AsyncCoordinator::new(coord_rx, shutdown_tx.subscribe());
    tasks.spawn(coordinator.run());

    // Spawn агентів
    for i in 0..1000 {  // 1000 агентів!
        let (cmd_tx, cmd_rx) = mpsc::channel(100);
        let agent = AsyncAgent::new(
            AgentId(i),
            coord_tx.clone(),
            cmd_rx,
        );
        tasks.spawn(agent.run());
    }

    // Graceful shutdown через 30 секунд
    tokio::time::sleep(Duration::from_secs(30)).await;
    let _ = shutdown_tx.send(());
```

🤖 MAC: Порівняння: Threads vs Async

Метрика	Threads	Async
100 агентів	~400 MB	~10 MB
1,000 агентів	~4 GB	~50 MB
10,000 агентів	~40 GB (!)	~500 MB
Context switches	High	Low
Latency	Higher	Lower
Throughput	Limited by threads	Limited by CPU

Async дозволяє масштабувати до сотень тисяч агентів
на звичайному сервері!

Типові помилки в async коді

```
async fn bad() {
    some_async_fn(); // Нічого!
}
```

✓ 3 .await

```
async fn good() {
    some_async_fn().await;
}
```

Ще типові помилки

```
trait MyTrait {
    async fn method(&self); // ❌
}
```

✓ async_trait crate

```
#[async_trait]
trait MyTrait {
    async fn method(&self);
```

Best Practices

✓ Використовуйте tokio для production
✓ spawn_blocking для CPU-bound та sync I/O
✓ timeout для всіх зовнішніх операцій
✓ select! для graceful shutdown
✓ Мінімізуйте scope async mutex guards
✓ JoinSet для керування багатьма задачами

❌ Не блокуйте в async коді (thread::sleep, sync I/O)
❌ Не забувайте .await (compiler warning допомагає)
❌ Не тримайте locks через await points
❌ Не ігноруйте JoinHandle (задача стане detached)

🤖 Для MAC:
• Async для I/O-bound агентів (networking, sensing)
• spawn_blocking для CPU-bound (path planning)

Підсумок лекції

Runtime — виконує futures:
• Tokio — індустріальний стандарт
• #[tokio::main] — точка входу

Базові операції:
• spawn — створити задачу
• join! — паралельно, всі
• select! — паралельно, перший
• timeout — обмеження часу
• spawn_blocking — CPU tasks

Cancellation:
• Drop future = скасування
• handle.abort() — явне скасування

→ Наступна лекція: Tokio детально

🤖 MAC: 1000+ агентів на кількох потоках з мінімальним memory footprint!

Завдання для самостійної роботи

1. Hello Async:
   - Напишіть async функцію, що чекає 1 секунду
   - Виведіть час виконання

2. Parallel Fetch:
   - 5 async операцій (імітація з sleep)
   - Виконайте паралельно з join!
   - Порівняйте час з послідовним

3. Timeout:
   - Async операція що "зависає"
   - Обмежте timeout 2 секунди

4. Async Agent:
   - Агент з tick() кожні 100ms
   - Приймає команди через channel
   - Graceful shutdown

5. Async Swarm:
   - 100 async агентів
   - 1 координатор
   - select! для shutdown

Async/Await опановано!

Future • Tokio • spawn • join! • select!

Питання?