Лекція 12

Потоки: Базова багатопотоковість

Паралельне виконання з гарантіями безпеки

thread::spawn • JoinHandle • move • Send • Sync

🤖 Приклади: паралельні агенти, рій БПЛА

Частина 1: Основи потоків

О. С. Бичков • 2025

План лекції (Частина 1)

1. Чому багатопотоковість?
2. Процеси vs Потоки
3. Проблеми паралелізму
4. Fearless Concurrency у Rust
5. thread::spawn — створення потоку
6. JoinHandle — очікування завершення
7. move closures
8. Передача даних у потік
9. Send trait
10. Sync trait
11. Send + Sync правила
12. Типи та thread safety
13. 🤖 Паралельні агенти
14. 🤖 Незалежні патрульні
15. 🤖 Worker threads

Частина 2: Arc, Mutex, channels, синхронізація

Чому багатопотоковість?

Сучасні CPU мають багато ядер:
- Intel i7: 8-16 ядер
- AMD Ryzen: 8-64 ядра
- Apple M3: 8-12 ядер
- Серверні: 64-128 ядер

Однопотокова програма використовує лише 1 ядро!

Переваги паралелізму:
- Швидкість — паралельні обчислення
- Відгук — UI не блокується
- Ефективність — використання ресурсів
- Масштабування — більше ядер = швидше

🤖 MAC: кожен агент — окремий потік!

Рій БПЛА: 10 дронів = 10 потоків

- Кожен дрон приймає рішення незалежно
- Паралельна обробка сенсорів
- Одночасний рух без блокування
- Комунікація через повідомлення

Процеси vs Потоки

Процес (Process)

- Ізольована пам'ять
- Власні ресурси ОС
- Важкий для створення
- Комунікація через IPC
- Падіння одного не впливає на інші

Приклад: окремі програми

Потік (Thread)

- Спільна пам'ять процесу
- Легкий для створення
- Швидка комунікація
- Потрібна синхронізація!
- Падіння одного = падіння всього процесу

Приклад: паралельні задачі

Rust: потоки з гарантіями безпеки на етапі компіляції!

Проблеми паралелізму

Data Race

Два потоки одночасно:
• читають і пишуть
• або обидва пишуть
без синхронізації → UB

Deadlock

Потік A чекає на B,
потік B чекає на A.
Обидва заблоковані назавжди.

Race Condition

Результат залежить від порядку
виконання потоків.
Недетермінована поведінка.

Rust запобігає на компіляції!

• Data races — неможливі
• Race conditions — виявляються
• Deadlocks — логічна помилка

```
// C++ — data race
int counter = 0;
std::thread t1([&]{ counter++; });  // Race!
std::thread t2([&]{ counter++; });  // Race!

// Rust — не скомпілюється!
let mut counter = 0;
std::thread::spawn(|| counter += 1);  // ❌ Error!
// "closure may outlive the current function"
// "cannot borrow `counter` as mutable"
```

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

fn main() {
    // Створюємо новий потік
    thread::spawn(|| {
        for i in 1..10 {
            println!("Потік: {}", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    // Головний потік продовжує
    for i in 1..5 {
        println!("Головний: {}", i);
        thread::sleep(Duration::from_millis(1));
    }
}

// Вивід (приблизний — недетермінований):
// Головний: 1
// Потік: 1
// Головний: 2
// Потік: 2
// ... (потік може не завершитись!)
```

```rust
use std::thread;

fn main() {
    // spawn повертає JoinHandle
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("Потік: {}", i);
        }
        42  // Повертаємо значення
    });

    println!("Головний потік працює...");

    // join() — чекаємо завершення потоку
    let result = handle.join().unwrap();  // Блокує!

    println!("Потік повернув: {}", result);  // 42
}
```

join() блокує поточний потік до завершення іншого

```rust
// JoinHandle<T> — T це тип, що повертає потік
// join() -> Result<T, Box<dyn Any + Send>>
// Err якщо потік запанікував
```

```rust
use std::thread;

fn main() {
    let mut handles = vec![];

    // Створюємо 5 потоків
    for i in 0..5 {
        let handle = thread::spawn(move || {
            println!("Потік {} запущено", i);
            thread::sleep(std::time::Duration::from_millis(100));
            println!("Потік {} завершено", i);
            i * 2  // Повертаємо результат
        });
        handles.push(handle);
    }

    // Чекаємо на всі потоки
    let results: Vec<i32> = handles
        .into_iter()
        .map(|h| h.join().unwrap())
        .collect();

    println!("Результати: {:?}", results);  // [0, 2, 4, 6, 8]
}
```

```rust
use std::thread;

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

    // ❌ Без move — помилка!
    // let handle = thread::spawn(|| {
    //     println!("{:?}", data);  // Borrow data
    // });
    // Error: closure may outlive the current function

    // ✓ З move — ownership передається в потік
    let handle = thread::spawn(move || {
        println!("{:?}", data);  // data тепер належить потоку
    });

    // data більше недоступна тут!
    // println!("{:?}", data);  // ❌ Error: value moved

    handle.join().unwrap();
}

// move вимагається бо потік може пережити поточну функцію
```

```rust
use std::thread;

fn spawn_broken() {
    let data = vec![1, 2, 3];

    thread::spawn(|| {
        // Closure захоплює &data
        println!("{:?}", data);
    });  // ← потік може працювати після return

}  // ← data dropped тут!

// Якби це скомпілювалось:
// 1. spawn_broken() завершується
// 2. data звільняється
// 3. потік все ще працює з &data ← dangling reference!

// Rust не дозволяє!
// Error: `data` does not live long enough
//        may outlive borrowed value `data`

// Рішення: move переміщує data у потік
// Тепер потік ВОЛОДІЄ data, і вона живе поки живе потік
```

```rust
use std::thread;

fn main() {
    let name = String::from("Agent-42");
    let config = vec![1, 2, 3];

    // Передаємо кілька значень через move
    let handle = thread::spawn(move || {
        println!("Name: {}", name);
        println!("Config: {:?}", config);

        // Можемо мутувати — ми власники!
        let mut local_config = config;
        local_config.push(4);
        local_config
    });

    // name і config тут недоступні

    let result = handle.join().unwrap();
    println!("Result: {:?}", result);  // [1, 2, 3, 4]
}

// Якщо потрібно залишити дані — Clone
let name = String::from("shared");
let name_clone = name.clone();
thread::spawn(move || println!("{}", name_clone));
```

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

// Типи що НЕ Send:
let rc = Rc::new(5);

// ❌ Error: `Rc<i32>` cannot be sent between threads safely
// thread::spawn(move || {
//     println!("{}", rc);
// });

// Rc — не thread-safe (лічильник не atomic)
// Для потоків: Arc<T>

use std::sync::Arc;
let arc = Arc::new(5);
let arc_clone = Arc::clone(&arc);
thread::spawn(move || {
    println!("{}", arc_clone);  // ✓ Arc: Send
});
```

```rust
use std::cell::RefCell;

// RefCell — НЕ Sync (runtime borrow checking не thread-safe)
let cell = RefCell::new(5);

// Спроба передати &RefCell
// ❌ Error: `RefCell<i32>` cannot be shared between threads safely
// let cell_ref = &cell;
// thread::spawn(move || {
//     println!("{:?}", cell_ref);
// });

// Типи що НЕ Sync:
// RefCell<T>, Cell<T> — interior mutability не thread-safe
// Rc<T> — лічильник не atomic
// *mut T — raw pointers

// Для thread-safe interior mutability: Mutex<T>, RwLock<T>
```

Send + Sync: Зведена таблиця

Тип	Send	Sync	Примітка
i32, f64, bool	✓	✓	Примітиви — безпечні
String, Vec<T>	✓*	✓*	*якщо T: Send/Sync
Box<T>	✓*	✓*	*якщо T: Send/Sync
Rc<T>	❌	❌	Non-atomic counter
Arc<T>	✓*	✓*	Atomic counter
Cell<T>, RefCell<T>	✓	❌	Not thread-safe
Mutex<T>	✓	✓	Thread-safe lock
RwLock<T>	✓	✓	Read-write lock
*mut T, *const T	❌	❌	Raw pointers

Compiler автоматично перевіряє Send/Sync для thread::spawn!

```
// Send і Sync реалізуються автоматично
// якщо всі поля типу — Send/Sync

struct MyData {
    value: i32,       // Send + Sync
    name: String,     // Send + Sync
    items: Vec<u8>,   // Send + Sync
}
// MyData автоматично Send + Sync!

struct NotSync {
    value: i32,
    cell: std::cell::Cell<i32>,  // !Sync
}
// NotSync: Send, але НЕ Sync!

// Можна явно заборонити:
struct NoSend {
    data: i32,
    _marker: std::marker::PhantomData<*const ()>,  // *const !Send
}
// NoSend: !Send

// Або unsafe impl (обережно!):
// unsafe impl Send for MyType {}
// unsafe impl Sync for MyType {}
```

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

struct Agent {
    id: u32,
    name: String,
}

impl Agent {
    fn run(self) {
        for tick in 1..=5 {
            println!("[Agent {}] {} tick {}", self.id, self.name, tick);
            thread::sleep(Duration::from_millis(100));
        }
        println!("[Agent {}] completed", self.id);
    }
}

fn main() {
    let agents = vec![
        Agent { id: 1, name: "Alpha".into() },
        Agent { id: 2, name: "Beta".into() },
        Agent { id: 3, name: "Gamma".into() },
    ];

    let handles: Vec<_> = agents
        .into_iter()
        .map(|agent| thread::spawn(move || agent.run()))
        .collect();
```

```
use std::thread;

#[derive(Clone)]
struct PatrolRoute {
    waypoints: Vec<(f64, f64)>,
}

struct Drone {
    id: u32,
    route: PatrolRoute,
}

impl Drone {
    fn patrol(self) -> u32 {
        println!("Drone {} starting patrol", self.id);
        for (i, (x, y)) in self.route.waypoints.iter().enumerate() {
            println!("Drone {} at waypoint {}: ({}, {})", self.id, i, x, y);
            std::thread::sleep(std::time::Duration::from_millis(50));
        }
        self.id  // Повертаємо ID завершеного дрона
    }
}

let routes = vec![/* ... */];
let drones: Vec<Drone> = routes.into_iter().enumerate()
    .map(|(i, route)| Drone { id: i as u32, route })
    .collect();

let handles: Vec<_> = drones.into_iter()
    .map(|d| thread::spawn(move || d.patrol()))
```

MAC Workbench

```rust
use std::thread;

/// Задача для обробки
enum Task {
    ProcessImage { id: u32, data: Vec<u8> },
    CalculatePath { from: (f64, f64), to: (f64, f64) },
    AnalyzeSensor { sensor_id: u32, readings: Vec<f64> },
}

/// Результат обробки
enum TaskResult {
    ImageProcessed { id: u32, features: Vec<f32> },
    PathCalculated { waypoints: Vec<(f64, f64)> },
    SensorAnalyzed { anomalies: Vec<usize> },
}

fn process_task(task: Task) -> TaskResult {
    match task {
        Task::ProcessImage { id, data } => {
            // Важкі обчислення...
            TaskResult::ImageProcessed { id, features: vec![] }
        }
        // ...
    }
}

// Запуск workers
let tasks: Vec<Task> = get_pending_tasks();
let results: Vec<TaskResult> = tasks.into_iter()
    .map(|t| thread::spawn(move || process_task(t)))
```

```rust
use std::thread;

// Builder дозволяє налаштувати потік перед створенням
let builder = thread::Builder::new()
    .name("worker-1".to_string())  // Ім'я потоку
    .stack_size(4 * 1024 * 1024);  // Розмір стеку (4 MB)

let handle = builder.spawn(|| {
    // Отримати ім'я поточного потоку
    let name = thread::current().name().unwrap_or("unnamed");
    println!("Thread '{}' running", name);

    42
}).expect("Failed to spawn thread");

let result = handle.join().unwrap();
println!("Result: {}", result);

// Корисно для:
// - Дебагу (іменовані потоки в stack traces)
// - Великих стеків для глибокої рекурсії
// - Контролю ресурсів
```

```rust
use std::thread;

// thread::current() — інформація про поточний потік
let current = thread::current();
println!("Name: {:?}", current.name());
println!("ID: {:?}", current.id());

// thread::sleep — пауза
thread::sleep(std::time::Duration::from_millis(100));

// thread::yield_now — віддати час іншим потокам
thread::yield_now();

// thread::park / unpark — примітивна синхронізація
let handle = thread::spawn(|| {
    println!("Waiting to be unparked...");
    thread::park();  // Блокується
    println!("Unparked!");
});

thread::sleep(std::time::Duration::from_millis(100));
handle.thread().unpark();  // Розблокувати
handle.join().unwrap();

// available_parallelism — кількість CPU
let cpus = thread::available_parallelism().unwrap().get();
println!("CPUs: {}", cpus);
```

```rust
use std::thread;

// Panic в одному потоці НЕ вбиває інші
let handle = thread::spawn(|| {
    panic!("Something went wrong!");
});

// join() повертає Err якщо потік запанікував
match handle.join() {
    Ok(result) => println!("Success: {:?}", result),
    Err(e) => {
        // e: Box<dyn Any + Send>
        if let Some(msg) = e.downcast_ref::<&str>() {
            println!("Thread panicked: {}", msg);
        } else if let Some(msg) = e.downcast_ref::<String>() {
            println!("Thread panicked: {}", msg);
        } else {
            println!("Thread panicked with unknown error");
        }
    }
}

println!("Main thread continues!");

// Panic в main() завершує програму
// catch_unwind для ловлення паніки
```

Типові помилки

```
let rc = Rc::new(5);
thread::spawn(move || {
    println!("{}", rc); // ❌
});
```

✓ Arc замість Rc

```
let arc = Arc::new(5);
thread::spawn(move || {
```

Підсумок: Частина 1

Потоки в Rust — безпечні за замовчуванням:
• thread::spawn — створити потік
• JoinHandle — очікувати завершення
• move closure — передача ownership

Send + Sync traits:
• Send — можна передати в потік
• Sync — можна ділити &T між потоками
• Перевіряються на компіляції!

Типи:
• Rc → Arc для потоків
• RefCell → Mutex/RwLock
• Cell — залишається !Sync

→ Частина 2: Arc, Mutex, channels, синхронізація

🤖 MAC: кожен агент = окремий потік

Лекція 12 (продовження)

Arc, Mutex та синхронізація

Спільний стан та message passing

Arc • Mutex • RwLock • channels • mpsc

🤖 Приклади: спільна карта, координатор рою

Частина 2: Синхронізація

О. С. Бичков • 2025

План лекції (Частина 2)

1. Arc<T> — thread-safe Rc
2. Arc::clone vs Rc::clone
3. Mutex<T> — взаємне виключення
4. lock() та MutexGuard
5. Arc<Mutex<T>> pattern
6. Deadlocks
7. RwLock<T> — read-write lock
8. Коли Mutex, коли RwLock
9. Message passing — channels
10. mpsc::channel
11. Sender та Receiver
12. 🤖 Спільна карта світу
13. 🤖 Координатор рою
14. 🤖 Агенти з каналами
15. 🤖 Worker pool
16. Порівняння підходів

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

// Arc — thread-safe версія Rc
// A = Atomic (atomic operations для counter)

let data = Arc::new(vec![1, 2, 3]);

// Arc::clone — atomic increment, не deep copy!
let data1 = Arc::clone(&data);
let data2 = Arc::clone(&data);

let handle1 = thread::spawn(move || {
    println!("Thread 1: {:?}", data1);
});

let handle2 = thread::spawn(move || {
    println!("Thread 2: {:?}", data2);
});

handle1.join().unwrap();
handle2.join().unwrap();

// data все ще доступна (strong_count = 1)
println!("Main: {:?}", data);
```

```
// Правило: завжди починайте з Rc
// Переходьте на Arc тільки для потоків

use std::rc::Rc;     // Single-threaded
use std::sync::Arc;  // Multi-threaded

// Arc НЕ дає мутабельність!
// Arc<T> дозволяє &T з багатьох потоків
// Для мутабельності: Arc<Mutex<T>>
```

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

// Mutex обгортає дані
let m = Mutex::new(5);

{
    // lock() — заблокувати і отримати доступ
    let mut guard = m.lock().unwrap();

    // guard: MutexGuard<i32> — Deref до &mut i32
    *guard += 1;

    println!("Value: {}", *guard);  // 6
}  // guard dropped → mutex розблоковано

// Тепер інші можуть заблокувати
{
    let guard = m.lock().unwrap();
    println!("Value: {}", *guard);  // 6
}

// lock() повертає Result — може бути poisoned
```

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

let mutex = Mutex::new(String::from("hello"));

// lock() повертає LockResult<MutexGuard<T>>
// MutexGuard — RAII guard, unlock при drop

let result = mutex.lock();

match result {
    Ok(mut guard) => {
        // guard реалізує Deref<Target=T> та DerefMut
        guard.push_str(" world");
        println!("{}", *guard);

        // При виході guard dropped → unlock
    }
    Err(poisoned) => {
        // Mutex "отруєний" — попередній власник запанікував
        // Дані можуть бути в неконсистентному стані
        let guard = poisoned.into_inner();  // Отримати все одно
        println!("Recovered: {}", *guard);
    }
}

// try_lock() — не блокуюча спроба
if let Ok(guard) = mutex.try_lock() {
    println!("{}", *guard);
}
```

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

fn main() {
    // Arc для shared ownership, Mutex для мутабельності
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);  // Clone Arc, не Mutex!

        let handle = thread::spawn(move || {
            // Кожен потік блокує, інкрементує, розблоковує
            let mut num = counter.lock().unwrap();
            *num += 1;
            // num dropped тут → unlock
        });
        handles.push(handle);
    }

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

    println!("Result: {}", *counter.lock().unwrap());  // 10
}
```

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

// ❌ Deadlock scenario
let a = Arc::new(Mutex::new(1));
let b = Arc::new(Mutex::new(2));

let a1 = Arc::clone(&a);
let b1 = Arc::clone(&b);

let handle1 = thread::spawn(move || {
    let _lock_a = a1.lock().unwrap();  // Потік 1 блокує A
    thread::sleep(std::time::Duration::from_millis(10));
    let _lock_b = b1.lock().unwrap();  // Чекає на B...
});

let a2 = Arc::clone(&a);
let b2 = Arc::clone(&b);

let handle2 = thread::spawn(move || {
    let _lock_b = b2.lock().unwrap();  // Потік 2 блокує B
    thread::sleep(std::time::Duration::from_millis(10));
    let _lock_a = a2.lock().unwrap();  // Чекає на A...
});

// ❌ DEADLOCK! Обидва потоки чекають вічно
```

```
// Правильний порядок — сортуємо за адресою
let (first, second) = if &*a as *const _ < &*b as *const _ {
    (&a, &b)
} else {
    (&b, &a)
};
let _g1 = first.lock().unwrap();
let _g2 = second.lock().unwrap();
```

4. Використовуйте try_lock з timeout

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

// RwLock дозволяє:
// - Багато читачів одночасно (&T)
// - АБО один писач (&mut T)

let lock = RwLock::new(vec![1, 2, 3]);

// Читання — багато потоків одночасно
{
    let r1 = lock.read().unwrap();  // RwLockReadGuard
    let r2 = lock.read().unwrap();  // ✓ Ще один читач OK!
    println!("{:?} {:?}", *r1, *r2);
}  // guards dropped

// Запис — ексклюзивно
{
    let mut w = lock.write().unwrap();  // RwLockWriteGuard
    w.push(4);
    // Ніхто інший не може читати чи писати
}  // guard dropped

// try_read() / try_write() — неблокуючі версії
if let Ok(r) = lock.try_read() {
    println!("{:?}", *r);
}
```

Mutex vs RwLock — коли що

Критерій	Mutex	RwLock
Readers	1	Багато
Writers	1	1
Overhead	Менший	Більший
Fairness	FIFO	Reader bias можливий
Коли	Часті writes	Рідкі writes, часті reads

Правило вибору:
• Mutex — за замовчуванням (простіше, швидше)
• RwLock — якщо багато читачів і рідкі записи

🤖 MAC приклади:
• Mutex — лічильник повідомлень, черга команд
• RwLock — спільна карта світу (часте читання)

```rust
use std::sync::mpsc;  // Multi-Producer, Single-Consumer

// channel() створює (Sender, Receiver)
let (tx, rx) = mpsc::channel();

// tx — Sender, можна clone
// rx — Receiver, тільки один

tx.send(42).unwrap();     // Відправити
let msg = rx.recv().unwrap();  // Отримати (blocking)

println!("Received: {}", msg);  // 42
```

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

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

    thread::spawn(move || {
        let messages = vec!["hi", "from", "thread"];
        for msg in messages {
            tx.send(msg).unwrap();
            thread::sleep(std::time::Duration::from_millis(100));
        }
        // tx dropped тут — канал закритий
    });

    // Головний потік отримує
    // rx.recv() блокує поки є повідомлення
    // Повертає Err коли канал закритий
    while let Ok(msg) = rx.recv() {
        println!("Got: {}", msg);
    }

    println!("Channel closed");
}

// Вивід:
// Got: hi
// Got: from
// Got: thread
// Channel closed
```

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

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

    // Clone tx для кожного producer
    for i in 0..3 {
        let tx_clone = tx.clone();

        thread::spawn(move || {
            for j in 0..3 {
                let msg = format!("Producer {} message {}", i, j);
                tx_clone.send(msg).unwrap();
            }
            // tx_clone dropped
        });
    }

    // Важливо! Drop оригінальний tx
    drop(tx);  // Інакше rx.recv() буде чекати вічно

    // Отримуємо всі повідомлення
    for received in rx {
        println!("{}", received);
    }
}

// 9 повідомлень у недетермінованому порядку
```

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

// sync_channel — обмежений буфер
let (tx, rx) = mpsc::sync_channel(2);  // Буфер на 2 елементи

tx.send(1).unwrap();  // ✓ Буфер: [1]
tx.send(2).unwrap();  // ✓ Буфер: [1, 2]
// tx.send(3).unwrap();  // ⏳ БЛОКУЄ! Буфер повний

// Різниця:
// channel() — unbounded, ніколи не блокує send
// sync_channel(n) — bounded, блокує коли повний

// sync_channel(0) — rendezvous channel
// send блокує поки receive не готовий

let (tx, rx) = mpsc::sync_channel(0);

std::thread::spawn(move || {
    tx.send(42).unwrap();  // Блокує поки rx.recv()
});

std::thread::sleep(std::time::Duration::from_millis(100));
let val = rx.recv().unwrap();  // Розблоковує send
```

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

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

// recv() — блокує поки є повідомлення
let msg = rx.recv()?;  // Result<T, RecvError>

// try_recv() — не блокує
match rx.try_recv() {
    Ok(msg) => println!("Got: {}", msg),
    Err(mpsc::TryRecvError::Empty) => println!("No message yet"),
    Err(mpsc::TryRecvError::Disconnected) => println!("Channel closed"),
}

// recv_timeout() — блокує з timeout
match rx.recv_timeout(Duration::from_secs(1)) {
    Ok(msg) => println!("Got: {}", msg),
    Err(mpsc::RecvTimeoutError::Timeout) => println!("Timeout!"),
    Err(mpsc::RecvTimeoutError::Disconnected) => println!("Closed"),
}

// iter() — ітератор по повідомленнях
for msg in rx.iter() {
    println!("{}", msg);  // Поки канал відкритий
}
```

```rust
use std::sync::{Arc, RwLock};
use std::collections::HashMap;

type WorldMap = Arc<RwLock<HashMap<(i32, i32), CellType>>>;

fn create_agents(map: WorldMap, count: usize) -> Vec<JoinHandle<()>> {
    (0..count).map(|id| {
        let map = Arc::clone(&map);

        thread::spawn(move || {
            loop {
                // Читання — багато агентів одночасно
                {
                    let map_read = map.read().unwrap();
                    let cell = map_read.get(&(id as i32, 0));
                    // Аналіз...
                }  // read lock released

                // Запис — ексклюзивно
                {
                    let mut map_write = map.write().unwrap();
                    map_write.insert((id as i32, 0), CellType::Explored);
                }  // write lock released

                thread::sleep(Duration::from_millis(100));
            }
        })
    }).collect()
}
```

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

#[derive(Debug)]
enum AgentMessage {
    Position { id: u32, x: f64, y: f64 },
    TargetDetected { id: u32, target: TargetInfo },
    LowBattery { id: u32, level: u8 },
    RequestCommand { id: u32 },
}

struct Coordinator {
    rx: mpsc::Receiver<AgentMessage>,
    agents: HashMap<u32, AgentInfo>,
}

impl Coordinator {
    fn run(&mut self) {
        while let Ok(msg) = self.rx.recv() {
            match msg {
                AgentMessage::Position { id, x, y } => {
                    self.agents.get_mut(&id).unwrap().position = (x, y);
                }
                AgentMessage::TargetDetected { id, target } => {
                    self.assign_agents_to_target(target);
                }
                AgentMessage::LowBattery { id, level } => {
                    self.recall_agent(id);
                }
                AgentMessage::RequestCommand { id } => {
                    self.send_next_command(id);
```

```rust
use std::sync::mpsc::{self, Sender, Receiver};

struct Agent {
    id: u32,
    to_coordinator: Sender<AgentMessage>,
    from_coordinator: Receiver<Command>,
}

impl Agent {
    fn run(self) {
        loop {
            // Надіслати позицію
            self.to_coordinator.send(AgentMessage::Position {
                id: self.id,
                x: self.current_x(),
                y: self.current_y(),
            }).unwrap();

            // Перевірити команди (неблокуюче)
            if let Ok(cmd) = self.from_coordinator.try_recv() {
                self.execute_command(cmd);
            }

            // Основна логіка агента...
            thread::sleep(Duration::from_millis(50));
        }
    }
}

// Створення агента з каналами
```

MAC Worker Pool

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

struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Job>,
}

type Job = Box<dyn FnOnce() + Send + 'static>;

impl ThreadPool {
    fn new(size: usize) -> Self {
        let (sender, receiver) = mpsc::channel();
        let receiver = Arc::new(Mutex::new(receiver));

        let workers: Vec<_> = (0..size)
            .map(|id| Worker::new(id, Arc::clone(&receiver)))
            .collect();

        ThreadPool { workers, sender }
    }

    fn execute<F: FnOnce() + Send + 'static>(&self, f: F) {
        self.sender.send(Box::new(f)).unwrap();
    }
}

struct Worker { id: usize, thread: JoinHandle<()> }

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Self {
```

```rust
use std::sync::atomic::{AtomicU64, Ordering};
use std::sync::Arc;

/// Thread-safe лічильник без Mutex
struct SwarmStats {
    messages_sent: AtomicU64,
    messages_received: AtomicU64,
    tasks_completed: AtomicU64,
}

impl SwarmStats {
    fn new() -> Self {
        SwarmStats {
            messages_sent: AtomicU64::new(0),
            messages_received: AtomicU64::new(0),
            tasks_completed: AtomicU64::new(0),
        }
    }

    fn record_sent(&self) {
        self.messages_sent.fetch_add(1, Ordering::Relaxed);
    }

    fn summary(&self) -> String {
        format!("Sent: {}, Received: {}, Tasks: {}",
            self.messages_sent.load(Ordering::Relaxed),
            self.messages_received.load(Ordering::Relaxed),
            self.tasks_completed.load(Ordering::Relaxed))
    }
}
```

Shared State vs Message Passing

Shared State (Arc<Mutex<T>>)

✓ Прямий доступ до даних
✓ Низький overhead для дрібних ops
✓ Простіше для простих випадків

✗ Deadlock можливий
✗ Складно масштабувати
✗ Контролювати порядок важко

Message Passing (channels)

✓ Немає shared state
✓ Ownership чіткий
✓ Масштабується добре
✓ Легше тестувати

✗ Копіювання даних
✗ Складніша архітектура
✗ Можливий channel bloat

🤖 MAC: часто комбінація — channels для команд, Arc<RwLock> для карти

Практичні поради

✓ Починайте з channels — простіше рефакторити
✓ Мінімізуйте scope lock'ів
✓ Уникайте вкладених locks
✓ Clone перед spawn (дешевше ніж Arc часом)
✓ Використовуйте try_lock/try_recv для responsive коду

❌ Не тримайте lock через await (async)
❌ Не ігноруйте poisoned mutex
❌ Не забувайте drop(tx) для закриття каналу

🤖 Для MAC:
• Координатор — окремий потік з Receiver
• Агенти — потоки з Sender до координатора
• Спільні дані — Arc<RwLock<T>> для читання

Типові помилки

```
let guard = mutex.lock().unwrap();
// ... багато коду ...
// guard живе надто довго
```

✓ Мінімальний scope

```
let value = {
    let guard = mutex.lock().unwrap();
```

Підсумок лекції

Shared State:
- Arc<T> — thread-safe shared ownership
- Mutex<T> — один потік має доступ
- RwLock<T> — багато читачів АБО один писач

Message Passing:
- mpsc::channel — multi-producer, single-consumer
- Sender можна clone, Receiver — ні
- sync_channel — bounded з backpressure

🤖 MAC архітектура:
- Координатор з Receiver для команд
- Агенти з Sender до координатора
- Arc<RwLock<WorldMap>> для спільної карти
- AtomicU64 для лічильників

→ Наступна лекція: Async/Await

Завдання для самостійної роботи

1. Counter: 10 потоків інкрементують Arc<Mutex<u32>>.
   Виведіть фінальне значення.

2. Producer-Consumer: 3 producers, 1 consumer.
   Producers генерують числа, consumer сумує.

3. Worker Pool: Реалізуйте пул з 4 workers.
   Виконайте 20 задач паралельно.

4. MAC Simulation:
   - 5 агентів (потоків)
   - 1 координатор
   - Агенти надсилають позиції
   - Координатор оновлює Arc<RwLock<HashMap>>
   - Агенти читають позиції сусідів

5. Chat: Multi-user chat з channels.
   Broadcast повідомлення всім учасникам.

Дякую за увагу!

Threads • Arc • Mutex • Channels

Питання?