Лекція 13

Arc<T> та Mutex<T>: Спільний стан

Thread-safe спільне володіння та синхронізація

Arc • Mutex • RwLock • Condvar • Barrier • Atomics

🤖 Приклади: спільний стан рою, координація агентів

Частина 1: Arc та Mutex глибоко

О. С. Бичков • 2025

План лекції (Частина 1)

1. Нагадування: проблема спільного стану
2. Arc<T> — внутрішня будова
3. Atomic operations
4. Arc методи детально
5. Weak<T> для Arc
6. Arc::make_mut та Arc::get_mut
7. Mutex<T> — внутрішня будова
8. Poisoned mutex
9. MutexGuard lifetime
10. Паттерни з Mutex
11. try_lock та timeout
12. 🤖 Спільний стан рою
13. 🤖 Реєстр агентів
14. 🤖 Command queue
15. Типові помилки з Arc/Mutex

Частина 2: RwLock, Condvar, Barrier, Atomics

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

// Rc<RefCell<T>> — працює в одному потоці
let counter = Rc::new(RefCell::new(0));

let counter_clone = Rc::clone(&counter);
thread::spawn(move || {
    *counter_clone.borrow_mut() += 1;
});

// ❌ Error: `Rc<RefCell<i32>>` cannot be sent between threads safely
// `Rc<RefCell<i32>>` cannot be shared between threads safely

// Причини:
// 1. Rc — non-atomic reference counting (!Send)
// 2. RefCell — non-thread-safe borrow checking (!Sync)
```

Рішення: Arc<Mutex<T>> — atomic + locking

```rust
// Спрощена структура Arc
pub struct Arc<T: ?Sized> {
    ptr: NonNull<ArcInner<T>>,
}

struct ArcInner<T: ?Sized> {
    strong: AtomicUsize,  // Atomic лічильник strong refs
    weak: AtomicUsize,    // Atomic лічильник weak refs
    data: T,              // Дані
}

// Arc::clone — atomic increment
impl<T> Clone for Arc<T> {
    fn clone(&self) -> Arc<T> {
        // Атомарно збільшуємо strong count
        self.inner().strong.fetch_add(1, Ordering::Relaxed);
        Arc { ptr: self.ptr }
    }
}

// Drop — atomic decrement, free якщо 0
impl<T> Drop for Arc<T> {
    fn drop(&mut self) {
        if self.inner().strong.fetch_sub(1, Ordering::Release) == 1 {
            // Останній strong — звільняємо
        }
    }
}
```

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

let counter = AtomicUsize::new(0);

// fetch_add — атомарно: читає, додає, записує
let old = counter.fetch_add(1, Ordering::SeqCst);  // old = 0
// counter тепер = 1

// load — атомарне читання
let value = counter.load(Ordering::SeqCst);  // 1

// store — атомарний запис
counter.store(42, Ordering::SeqCst);

// compare_exchange — атомарний CAS (compare-and-swap)
let result = counter.compare_exchange(
    42,     // expected
    100,    // new value
    Ordering::SeqCst,
    Ordering::SeqCst
);  // Ok(42) — успішно замінили
```

Memory Ordering — рівні синхронізації

Ordering	Опис	Коли
Relaxed	Тільки атомарність	Лічильники
Acquire	Синхронізує reads після	Lock acquire
Release	Синхронізує writes до	Lock release
AcqRel	Acquire + Release	Read-modify-write
SeqCst	Повний порядок	За замовчуванням, безпечно

Правило для початківців:
- SeqCst — завжди безпечно, трохи повільніше
- Relaxed — для простих лічильників
- Acquire/Release — для lock/unlock

Arc використовує:
- Relaxed для clone (просто лічильник)
- Release для drop, Acquire для останнього drop

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

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

// Clone — atomic increment, дешево
let arc2 = Arc::clone(&arc);  // ✓ Preferred style
let arc3 = arc.clone();        // Теж працює

// strong_count / weak_count — для дебагу
println!("Strong: {}", Arc::strong_count(&arc));  // 3

// ptr_eq — чи вказують на ті самі дані
assert!(Arc::ptr_eq(&arc, &arc2));

// try_unwrap — спробувати отримати T якщо єдиний власник
let arc = Arc::new(42);
match Arc::try_unwrap(arc) {
    Ok(value) => println!("Got: {}", value),  // 42
    Err(arc) => println!("Still shared"),
}

// into_inner (nightly) — те саме, panic якщо не єдиний
```

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

// Arc::get_mut — отримати &mut T якщо єдиний власник
let mut arc = Arc::new(5);

if let Some(value) = Arc::get_mut(&mut arc) {
    *value = 10;  // Можемо мутувати!
}

let arc2 = Arc::clone(&arc);  // Тепер 2 власники
assert!(Arc::get_mut(&mut arc).is_none());  // None!

// Arc::make_mut — Clone-on-write семантика
let mut arc = Arc::new(vec![1, 2, 3]);
let arc2 = Arc::clone(&arc);

// make_mut клонує дані якщо є інші власники
Arc::make_mut(&mut arc).push(4);  // arc клонований!

println!("{:?}", arc);   // [1, 2, 3, 4]
println!("{:?}", arc2);  // [1, 2, 3] — оригінал

// Корисно для copy-on-write оптимізації
```

```rust
use std::sync::{Arc, Weak};

// Weak — не запобігає звільненню
let strong = Arc::new(42);
let weak: Weak<i32> = Arc::downgrade(&strong);

println!("Strong: {}", Arc::strong_count(&strong));  // 1
println!("Weak: {}", Arc::weak_count(&strong));    // 1

// upgrade() — спробувати отримати Arc
match weak.upgrade() {
    Some(arc) => println!("Value: {}", arc),  // 42
    None => println!("Already dropped"),
}

// Після drop strong
drop(strong);
assert!(weak.upgrade().is_none());  // Дані звільнені!
```

Weak не збільшує strong_count — дані можуть бути звільнені!

```rust
// Використання: уникнення циклічних посилань
// Parent -> Child (Arc)
// Child -> Parent (Weak)
```

```
// Спрощена структура Mutex
pub struct Mutex<T: ?Sized> {
    inner: sys::Mutex,     // OS-level mutex
    poison: Flag,          // Чи запанікував попередній власник
    data: UnsafeCell<T>,   // Interior mutability
}

// lock() повертає MutexGuard — RAII guard
pub struct MutexGuard<'a, T: ?Sized + 'a> {
    lock: &'a Mutex<T>,
}

// MutexGuard реалізує Deref/DerefMut
impl<T> Deref for MutexGuard<'_, T> {
    type Target = T;
    fn deref(&self) -> &T {
        unsafe { &*self.lock.data.get() }
    }
}

// Drop MutexGuard — розблоковує mutex
impl<T> Drop for MutexGuard<'_, T> {
    fn drop(&mut self) {
        unsafe { self.lock.inner.unlock(); }
    }
}
```

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

let mutex = Mutex::new(vec![1, 2, 3]);

let result = thread::spawn(|| {
    let mut guard = mutex.lock().unwrap();
    guard.push(4);
    panic!("Oops!");  // Паніка з утриманим lock!
}).join();

// mutex тепер "отруєний" — дані можуть бути неконсистентні
match mutex.lock() {
    Ok(guard) => println!("Data: {:?}", *guard),
    Err(poisoned) => {
        // PoisonError містить guard
        println!("Mutex was poisoned!");

        // Можемо все одно отримати дані
        let guard = poisoned.into_inner();
        println!("Data anyway: {:?}", *guard);  // [1, 2, 3, 4]
    }
}

// clear_poison() — очистити poison flag
mutex.clear_poison();
```

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

let mutex = Mutex::new(HashMap::new());

// ❌ ПОГАНО — тримаємо lock надто довго
{
    let mut guard = mutex.lock().unwrap();
    guard.insert("key", expensive_computation());  // Lock тримається!
    // Інші потоки чекають...
}

// ✓ ДОБРЕ — мінімізуємо час lock
{
    let value = expensive_computation();  // Без lock
    let mut guard = mutex.lock().unwrap();
    guard.insert("key", value);  // Швидко!
}  // Lock звільнено

// ✓ ДОБРЕ — отримати і відпустити
let value = {
    let guard = mutex.lock().unwrap();
    guard.get("key").cloned()  // Clone і відпустити
};  // Lock звільнено

// Тепер працюємо з value без lock
```

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

// Pattern 1: Immediate scope
let data = Arc::new(Mutex::new(Vec::new()));
{
    data.lock().unwrap().push(1);  // Lock і unlock в одному виразі
}

// Pattern 2: Явний drop
let guard = data.lock().unwrap();
println!("{:?}", *guard);
drop(guard);  // Явно звільняємо до кінця scope
// Можемо робити інші речі

// Pattern 3: Conditional mutation
if let Ok(mut guard) = data.try_lock() {
    guard.push(2);
} else {
    println!("Mutex busy, skipping");
}

// Pattern 4: Read-then-write (обережно!)
let should_add = data.lock().unwrap().is_empty();
// ⚠️ Інший потік міг змінити між перевіркою і записом!
if should_add {
    data.lock().unwrap().push(1);
}
```

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

let mutex = Mutex::new(42);

// try_lock — не блокує
match mutex.try_lock() {
    Ok(guard) => println!("Got: {}", *guard),
    Err(TryLockError::WouldBlock) => println!("Mutex is locked"),
    Err(TryLockError::Poisoned(e)) => println!("Poisoned: {:?}", e),
}

// Для timeout — використовуйте parking_lot crate
// parking_lot::Mutex має try_lock_for та try_lock_until

// Або реалізуйте самі з loop + try_lock + sleep
fn lock_with_timeout<T>(mutex: &Mutex<T>, timeout: Duration)
    -> Option<std::sync::MutexGuard<T>>
{
    let start = std::time::Instant::now();
    loop {
        match mutex.try_lock() {
            Ok(guard) => return Some(guard),
            Err(_) if start.elapsed() > timeout => return None,
            Err(_) => std::thread::yield_now(),
        }
    }
}
```

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

/// Спільний стан всього рою
struct SwarmState {
    agents: HashMap<u32, AgentInfo>,
    world_map: WorldMap,
    active_missions: Vec<Mission>,
    global_stats: Stats,
}

/// Thread-safe обгортка
type SharedSwarmState = Arc<Mutex<SwarmState>>;

impl SwarmState {
    fn new() -> SharedSwarmState {
        Arc::new(Mutex::new(SwarmState {
            agents: HashMap::new(),
            world_map: WorldMap::new(),
            active_missions: Vec::new(),
            global_stats: Stats::default(),
        }))
    }
}

// Кожен агент отримує clone
fn spawn_agent(id: u32, state: SharedSwarmState) -> JoinHandle<()> {
    thread::spawn(move || {
        loop {
            let mut state = state.lock().unwrap();
```

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

struct AgentRegistry {
    agents: Mutex<HashMap<u32, AgentInfo>>,
    next_id: Mutex<u32>,
}

impl AgentRegistry {
    fn new() -> Arc<Self> {
        Arc::new(AgentRegistry {
            agents: Mutex::new(HashMap::new()),
            next_id: Mutex::new(1),
        })
    }

    fn register(&self, info: AgentInfo) -> u32 {
        let id = {
            let mut next = self.next_id.lock().unwrap();
            let id = *next;
            *next += 1;
            id
        };  // next_id lock released

        self.agents.lock().unwrap().insert(id, info);
        id
    }

    fn get(&self, id: u32) -> Option<AgentInfo> {
        self.agents.lock().unwrap().get(&id).cloned()
```

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

#[derive(Clone)]
enum Command {
    MoveTo(Position),
    Scan(Area),
    Attack(TargetId),
    Return,
}

struct CommandQueue {
    queue: Mutex<VecDeque<Command>>,
}

impl CommandQueue {
    fn new() -> Arc<Self> {
        Arc::new(CommandQueue {
            queue: Mutex::new(VecDeque::new()),
        })
    }

    fn push(&self, cmd: Command) {
        self.queue.lock().unwrap().push_back(cmd);
    }

    fn pop(&self) -> Option<Command> {
        self.queue.lock().unwrap().pop_front()
    }
```

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

#[derive(Default, Clone)]
struct AgentStats {
    moves: u64,
    scans: u64,
    messages_sent: u64,
    targets_detected: u64,
}

struct StatsCollector {
    per_agent: Mutex<HashMap<u32, AgentStats>>,
    global: Mutex<AgentStats>,
}

impl StatsCollector {
    fn record_move(&self, agent_id: u32) {
        {
            let mut per = self.per_agent.lock().unwrap();
            per.entry(agent_id).or_default().moves += 1;
        }
        self.global.lock().unwrap().moves += 1;
    }

    fn get_agent_stats(&self, agent_id: u32) -> Option<AgentStats> {
        self.per_agent.lock().unwrap().get(&agent_id).cloned()
    }

    fn get_global_stats(&self) -> AgentStats {
        self.global.lock().unwrap().clone()
```

Типові помилки з Arc/Mutex

```
let guard = mutex.lock().unwrap();
do_async_work().await; // ❌
// Guard не Send!
```

✓ Drop перед await

```
let value = mutex.lock().unwrap().clone();
drop guard; // або scope
do_async_work().await; // ✓
```

Ще типові помилки

```
let g = mutex.lock().unwrap();
// Panic якщо poisoned!
```

✓ Обробка poison

```
let g = mutex.lock()
    .unwrap_or_else(|p| p.into_inner());
```

Arc<Mutex<T>> vs Rc<RefCell<T>>

	Rc<RefCell<T>>	Arc<Mutex<T>>
Thread-safe	❌	✓
Counter type	Non-atomic	Atomic
Borrow check	Runtime (panic)	OS lock (block)
Overhead	Менший	Більший
Use case	Single thread	Multi thread
Poison	Ні	Так

Правило:
- Один потік → Rc<RefCell<T>>
- Багато потоків → Arc<Mutex<T>> або Arc<RwLock<T>>

Interior Mutability: повне порівняння

Тип	Thread	Check	Block	T вимоги
Cell<T>	❌	—	❌	T: Copy
RefCell<T>	❌	Runtime	❌	—
Mutex<T>	✓	OS Lock	✓	—
RwLock<T>	✓	OS Lock	✓	—
AtomicT	✓	—	❌	Primitives

Block = чи блокує потік при конфлікті

- Cell/RefCell — panic при порушенні
- Mutex — блокує потік до звільнення
- Atomic — lock-free, ніколи не блокує

Підсумок: Частина 1

Arc<T> — Atomic Reference Counting:
• Thread-safe shared ownership
• Clone = atomic increment
• Weak<T> для уникнення циклів
• make_mut для copy-on-write

Mutex<T> — Mutual Exclusion:
• Тільки один потік має доступ
• MutexGuard — RAII, auto unlock
• Poisoned якщо panic з lock
• Мінімізуйте час lock!

Arc<Mutex<T>> — стандартний патерн для спільного мутабельного стану між потоками

→ Частина 2: RwLock, Condvar, Barrier, Atomics

Лекція 13 (продовження)

RwLock, Condvar та Atomics

Розширені примітиви синхронізації

RwLock • Condvar • Barrier • Once • Atomic types

🤖 Приклади: карта світу, події рою, синхронізація

Частина 2: RwLock та інші примітиви

О. С. Бичков • 2025

План лекції (Частина 2)

1. RwLock<T> — read-write lock
2. RwLockReadGuard / WriteGuard
3. Коли Mutex, коли RwLock
4. Reader starvation
5. Condvar — condition variable
6. Condvar::wait / notify
7. Barrier — синхронна точка
8. Once — одноразова ініціалізація
9. Atomic types
10. AtomicBool, AtomicUsize
11. Atomic operations
12. 🤖 Спільна карта (RwLock)
13. 🤖 Event system (Condvar)
14. 🤖 Phase synchronization
15. 🤖 Lock-free counter
16. parking_lot crate

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

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!
    println!("{:?}, {:?}", *r1, *r2);
}  // r1, r2 dropped

// Запис — ексклюзивно
{
    let mut w = lock.write().unwrap();  // RwLockWriteGuard
    // Ніхто не може читати чи писати поки w живий
    w.push(4);
}  // w dropped — інші можуть працювати
```

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

let lock = RwLock::new(String::from("hello"));

// RwLockReadGuard<T> — Deref до &T
{
    let read_guard = lock.read().unwrap();
    println!("Length: {}", read_guard.len());  // Deref до &String
    // read_guard.push_str("!");  // ❌ Error: no DerefMut!
}

// RwLockWriteGuard<T> — Deref та DerefMut до T
{
    let mut write_guard = lock.write().unwrap();
    write_guard.push_str(" world");  // ✓ DerefMut
    println!("{}", *write_guard);    // "hello world"
}

// try_read() / try_write() — non-blocking
if let Ok(r) = lock.try_read() {
    println!("{}", *r);
} else {
    println!("Lock busy");
}
```

Коли Mutex, коли RwLock

Критерій	Mutex	RwLock
Concurrent reads	1	N
Concurrent writes	1	1
Lock overhead	Менший	Більший
Implementation	Простіша	Складніша
Fairness	FIFO	Можливий bias
Best for	Часті writes	Рідкі writes

Правило вибору:
- Mutex — за замовчуванням (простіше, менший overhead)
- RwLock — якщо читань >> записів (>10:1)

⚠️ RwLock НЕ завжди швидший за Mutex!

```
// Проблема: Reader starvation або Writer starvation

// Сценарій 1: Writer starvation
// Багато читачів постійно, писач ніколи не отримує lock
loop {
    let r = lock.read().unwrap();  // Читачі завжди є
    // Writer чекає вічно...
}

// Сценарій 2: Reader starvation (write-preferring)
// Писач блокує всіх читачів
loop {
    let w = lock.write().unwrap();  // Writer завжди бере lock
    // Readers чекають вічно...
}

// std::sync::RwLock — OS-dependent поведінка
// Linux: зазвичай writer-preferring
// Windows: reader-preferring

// parking_lot::RwLock — fair, налаштовується
// Або: використовуйте Mutex якщо fairness критична
```

Порада: профілюйте реальний workload!

```
use std::sync::{Mutex, Condvar};

let pair = (Mutex::new(false), Condvar::new());
let (lock, cvar) = &pair;

// Потік 1: чекає на умову
let mut started = lock.lock().unwrap();
while !*started {
    // wait() атомарно: unlock mutex, sleep, re-lock при wake
    started = cvar.wait(started).unwrap();
}
println!("Started!");

// Потік 2: сигналізує
{
    let mut started = lock.lock().unwrap();
    *started = true;
}
cvar.notify_one();  // Розбудити один потік
// cvar.notify_all();  // Розбудити всі потоки
```

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

// Producer-Consumer з Condvar
let queue = Arc::new((Mutex::new(Vec::new()), Condvar::new()));

// Consumer thread
let q = Arc::clone(&queue);
thread::spawn(move || {
    let (lock, cvar) = &*q;
    loop {
        let mut queue = lock.lock().unwrap();

        // ВАЖЛИВО: while loop, не if!
        // Spurious wakeups можливі
        while queue.is_empty() {
            queue = cvar.wait(queue).unwrap();
        }

        let item = queue.pop().unwrap();
        println!("Consumed: {}", item);
    }
});

// Producer
let (lock, cvar) = &*queue;
lock.lock().unwrap().push(42);
cvar.notify_one();  // Розбудити consumer
```

```rust
use std::sync::{Mutex, Condvar};
use std::time::Duration;

let pair = (Mutex::new(false), Condvar::new());
let (lock, cvar) = &pair;

let mut guard = lock.lock().unwrap();

// wait_timeout — чекати з обмеженням часу
let result = cvar.wait_timeout(
    guard,
    Duration::from_secs(5)
).unwrap();

guard = result.0;  // Новий guard
let timed_out = result.1.timed_out();

if timed_out {
    println!("Timeout! Condition not met.");
} else {
    println!("Condition met: {}", *guard);
}

// wait_timeout_while — чекати поки предикат false
let (guard, timeout) = cvar.wait_timeout_while(
    lock.lock().unwrap(),
    Duration::from_secs(5),
    |&mut ready| !ready  // Чекати поки ready == false
).unwrap();
```

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

// Barrier — всі потоки чекають поки всі не досягнуть точки
let barrier = Arc::new(Barrier::new(3));  // 3 потоки

let mut handles = vec![];

for i in 0..3 {
    let b = Arc::clone(&barrier);

    handles.push(thread::spawn(move || {
        println!("Thread {} before barrier", i);

        // Чекаємо поки всі 3 потоки досягнуть цієї точки
        b.wait();

        println!("Thread {} after barrier", i);
    }));
}

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

// Вивід:
// Thread 0 before barrier
// Thread 1 before barrier
// Thread 2 before barrier
// (всі чекають...)
// Thread 0 after barrier
// Thread 1 after barrier
```

```rust
use std::sync::Once;

static INIT: Once = Once::new();
static mut CONFIG: Option<Config> = None;

fn get_config() -> &'static Config {
    // call_once гарантує виконання closure ОДИН раз
    // Навіть якщо кілька потоків викличуть одночасно
    INIT.call_once(|| {
        println!("Initializing config...");
        unsafe {
            CONFIG = Some(Config::load());
        }
    });

    unsafe { CONFIG.as_ref().unwrap() }
}

// Безпечніша альтернатива — OnceLock (Rust 1.70+)
use std::sync::OnceLock;

static CONFIG: OnceLock<Config> = OnceLock::new();

fn get_config() -> &'static Config {
    CONFIG.get_or_init(|| {
        println!("Initializing...");
        Config::load()
    })
}
```

```rust
use std::sync::atomic::{AtomicBool, AtomicUsize, AtomicI32, Ordering};

// Атомарні типи — thread-safe без lock
let flag = AtomicBool::new(false);
let counter = AtomicUsize::new(0);
let value = AtomicI32::new(42);

// load — атомарне читання
let f = flag.load(Ordering::SeqCst);
let c = counter.load(Ordering::SeqCst);

// store — атомарний запис
flag.store(true, Ordering::SeqCst);

// fetch_add/sub — атомарний інкремент/декремент
let old = counter.fetch_add(1, Ordering::SeqCst);  // Повертає старе

// compare_exchange — атомарний CAS
let result = flag.compare_exchange(
    false,              // expected
    true,               // new
    Ordering::SeqCst,   // success ordering
    Ordering::SeqCst    // failure ordering
);
// Ok(false) якщо було false і стало true
// Err(actual) якщо було не false
```

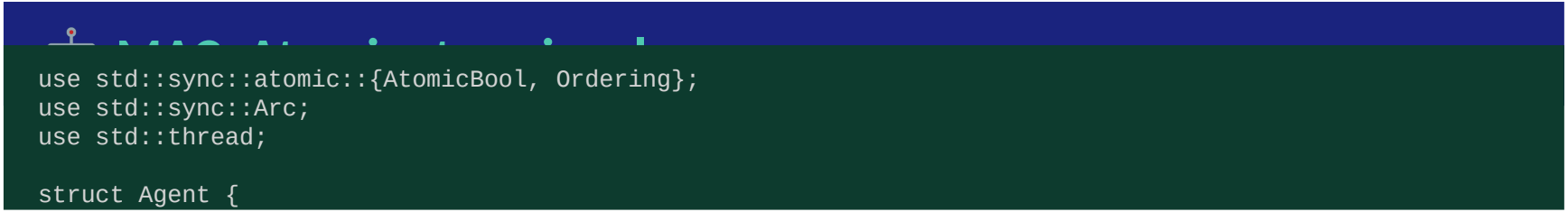
```rust
use std::sync::atomic::{AtomicUsize, AtomicBool, Ordering};

// Pattern 1: Counter (most common)
static COUNTER: AtomicUsize = AtomicUsize::new(0);

fn increment() {
    COUNTER.fetch_add(1, Ordering::Relaxed);
}

// Pattern 2: Flag (stop signal)
static RUNNING: AtomicBool = AtomicBool::new(true);

fn worker_loop() {
    while RUNNING.load(Ordering::Relaxed) {
        // do work...
    }
}

fn stop_workers() {
    RUNNING.store(false, Ordering::Relaxed);
}

// Pattern 3: Spin lock (зазвичай краще Mutex)
struct SpinLock {
    locked: AtomicBool,
}

impl SpinLock {
    fn lock(&self) {
        while self.locked.compare_exchange(
```

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

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

struct CellInfo {
    terrain: Terrain,
    last_seen: u64,
    seen_by: Vec<u32>,
}

struct Agent {
    id: u32,
    world_map: WorldMap,
}

impl Agent {
    /// Читання карти — багато агентів одночасно
    fn check_cell(&self, pos: (i32, i32)) -> Option<CellInfo> {
        let map = self.world_map.read().unwrap();
        map.get(&pos).cloned()
    }

    /// Оновлення карти — ексклюзивно
    fn update_cell(&self, pos: (i32, i32), info: CellInfo) {
        let mut map = self.world_map.write().unwrap();
        map.insert(pos, info);
    }

    /// Масове читання — один lock
```

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

#[derive(Clone)]
enum Event {
    TargetDetected { pos: Position },
    AgentDown { id: u32 },
    MissionComplete { mission_id: u32 },
}

struct EventBus {
    events: Mutex<Vec<Event>>,
    notifier: Condvar,
}

impl EventBus {
    fn publish(&self, event: Event) {
        self.events.lock().unwrap().push(event);
        self.notifier.notify_all();  // Розбудити всіх підписників
    }

    fn wait_for_event(&self) -> Event {
        let mut events = self.events.lock().unwrap();
        while events.is_empty() {
            events = self.notifier.wait(events).unwrap();
        }
        events.remove(0)
    }

    fn try_get_event(&self) -> Option<Event> {
        let mut events = self.events.lock().unwrap();
```

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

/// Симуляція з синхронними фазами
struct SwarmSimulation {
    agents: Vec<Agent>,
    phase_barrier: Arc<Barrier>,
}

impl SwarmSimulation {
    fn new(agent_count: usize) -> Self {
        SwarmSimulation {
            agents: (0..agent_count).map(Agent::new).collect(),
            phase_barrier: Arc::new(Barrier::new(agent_count)),
        }
    }

    fn run_tick(&self) {
        let handles: Vec<_> = self.agents.iter().enumerate().map(|(i, agent)| {
            let barrier = Arc::clone(&self.phase_barrier);
            let agent = agent.clone();

            thread::spawn(move || {
                // Фаза 1: Сприйняття
                agent.perceive();
                barrier.wait();  // Всі чекають

                // Фаза 2: Рішення
                agent.decide();
                barrier.wait();  // Всі чекають
```

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

/// Lock-free глобална статистика
struct SwarmStats {
    total_moves: AtomicU64,
    total_scans: AtomicU64,
    total_messages: AtomicU64,
    active_agents: AtomicU64,
}

impl SwarmStats {
    fn new() -> Arc<Self> {
        Arc::new(SwarmStats {
            total_moves: AtomicU64::new(0),
            total_scans: AtomicU64::new(0),
            total_messages: AtomicU64::new(0),
            active_agents: AtomicU64::new(0),
        })
    }

    fn record_move(&self) {
        self.total_moves.fetch_add(1, Ordering::Relaxed);
    }

    fn agent_joined(&self) {
        self.active_agents.fetch_add(1, Ordering::SeqCst);
    }

    fn agent_left(&self) {
```



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

struct Agent {
    id: u32,
    stop_flag: Arc<AtomicBool>,
}

impl Agent {
    fn run(&self) {
        println!("Agent {} started", self.id);

        while !self.stop_flag.load(Ordering::Relaxed) {
            // Основний цикл агента
            self.do_work();
            thread::sleep(std::time::Duration::from_millis(100));
        }

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

fn main() {
    let stop_flag = Arc::new(AtomicBool::new(false));

    let handles: Vec<_> = (0..5).map(|id| {
        let agent = Agent { id, stop_flag: Arc::clone(&stop_flag) };
        thread::spawn(move || agent.run())
    }).collect();
```

```
// parking_lot — популярна альтернатива std::sync
// Переваги:
// - Менший розмір (Mutex: 1 byte vs 40+ bytes)
// - Швидший для uncontended case
// - Не має poison
// - Додаткові можливості

use parking_lot::{Mutex, RwLock, Condvar};

let mutex = Mutex::new(42);
let guard = mutex.lock();  // Не повертає Result!
println!("{}", *guard);

// Додаткові можливості:
let rwlock = RwLock::new(vec![1, 2, 3]);

// Upgradeable read lock — можна апгрейднути до write
let read = rwlock.upgradable_read();
if read.is_empty() {
    let mut write = parking_lot::RwLockUpgradableReadGuard::upgrade(read);
    write.push(1);
}

// Fair lock
let fair_mutex = parking_lot::FairMutex::new(0);
```

Cargo.toml: parking_lot = "0.12"

Порівняння примітивів синхронізації

Примітив	Use Case	Blocking	Overhead
Mutex	General purpose	✓	Low
RwLock	Read-heavy	✓	Medium
Condvar	Wait for condition	✓	Low
Barrier	Phase sync	✓	Low
Once	Init once	✓	Very low
Atomic*	Counters, flags	❌	Lowest

Правило вибору:
- Atomic — для простих лічильників/прапорців
- Mutex — загальний випадок
- RwLock — якщо reads >> writes
- Condvar — очікування умови
- Barrier — фазова синхронізація

Best Practices

✓ Мінімізуйте scope lock'ів
✓ Уникайте вкладених locks (deadlock)
✓ Використовуйте RwLock тільки якщо вимірювання показують виграш
✓ Atomic для простих лічильників/прапорців
✓ parking_lot для production коду

❌ Не тримайте lock через await
❌ Не використовуйте spin locks без причини
❌ Не ігноруйте poisoned mutex

🤖 Для MAC:
• Arc<RwLock<WorldMap>> — карта світу
• Arc<Mutex<CommandQueue>> — черга команд
• AtomicBool — stop signal
• Condvar — очікування подій
• Barrier — фазова симуляція

Deadlock Prevention Strategies

1. Lock Ordering
   Завжди беріть locks в однаковому порядку

2. Lock Hierarchy
   Визначте рівні: L1 < L2 < L3
   Ніколи не беріть lower lock тримаючи higher

3. try_lock з timeout
   Відпустіть все і спробуйте знову

4. Single Lock
   Один mutex для всієї критичної секції

5. Lock-free
   Atomic operations замість locks

Підсумок лекції

RwLock<T>:
• Багато читачів АБО один писач
• Для read-heavy workloads

Condvar:
• Очікування умови
• wait() + notify_one()/notify_all()

Barrier:
• Синхронна точка для N потоків

Once / OnceLock:
• Одноразова ініціалізація

Atomic types:
• Lock-free операції
• Найшвидші для простих випадків

→ Наступна лекція: Channels — Message Passing

🤖 MAC: комбінація примітивів для різних задач

Завдання для самостійної роботи

1. Thread-safe Cache:
   - Arc<RwLock<HashMap<K, V>>>
   - get, insert, remove методи
   - Benchmark Mutex vs RwLock

2. Producer-Consumer з Condvar:
   - Bounded queue
   - Producers чекають якщо повний
   - Consumers чекають якщо порожній

3. Barrier-based simulation:
   - 10 агентів
   - 3 фази на tick: perceive, decide, act
   - Синхронізація між фазами

4. Lock-free Statistics:
   - Atomic counters для всіх метрик
   - snapshot() без locks

5. Event Bus:
   - publish(Event)
   - subscribe() → wait_for_event()
   - Timeout support

Дякую за увагу!

Arc • Mutex • RwLock • Condvar • Atomics

Питання?