Лекція 17

Tokio: Async Runtime

Глибоке занурення в Tokio екосистему

Runtime • Tasks • Channels • Sync • I/O • Time

🤖 Production-ready асинхронний рій агентів

Частина 1: Runtime та Tasks

О. С. Бичков • 2025

План лекції (Частина 1)

1. Tokio — огляд екосистеми
2. Архітектура Tokio Runtime
3. Multi-thread vs Current-thread
4. Runtime Builder
5. Налаштування worker threads
6. tokio::spawn детально
7. JoinHandle та JoinSet
8. Task cancellation
9. Local tasks (spawn_local)
10. spawn_blocking детально
11. block_in_place
12. Task priorities
13. 🤖 Task pool для агентів
14. 🤖 Динамічне створення агентів
15. Типові помилки
16. Підсумок

Частина 2: Channels, Sync, Time, I/O

Tokio — екосистема

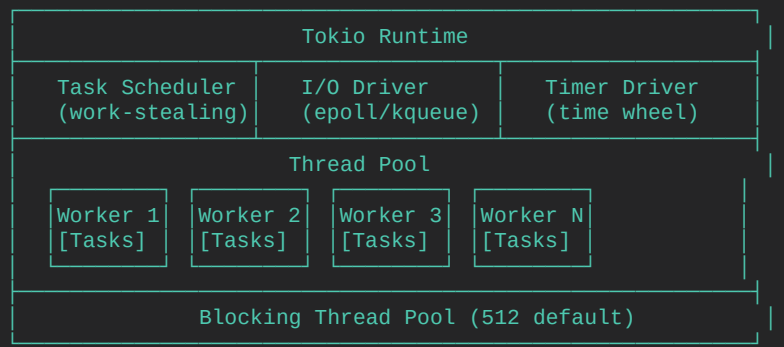
Tokio — найпопулярніший async runtime для Rust

Включає:
- Runtime — планувальник задач
- tokio::sync — async примітиви синхронізації
- tokio::time — таймери та інтервали
- tokio::io — async I/O
- tokio::net — TCP, UDP, Unix sockets
- tokio::process — async процеси

Крейт	Призначення
tokio	Core runtime та utilities
tokio-util	Додаткові utilities (codec, compat)
tokio-stream	Stream utilities
tokio-console	Runtime debugging tool

```toml
# Мінімальний набір
[dependencies]
tokio = { version = "1", features = ["rt", "macros"] }

# Повний набір (рекомендовано для розробки)
[dependencies]
tokio = { version = "1", features = ["full"] }

# Вибіркові features:
[dependencies]
tokio = { version = "1", features = [
    "rt-multi-thread",  # Multi-threaded runtime
    "rt",               # Current-thread runtime
    "sync",             # Channels, Mutex, etc.
    "time",             # Sleep, timeout, interval
    "io-util",          # AsyncRead/Write extensions
    "io-std",           # Stdin, stdout, stderr
    "net",              # TCP, UDP
    "fs",               # Async filesystem
    "macros",           # #[tokio::main], #[tokio::test]
    "signal",           # OS signals (Ctrl+C)
    "process",          # Async processes
    "parking_lot",      # Faster sync primitives
    "tracing",          # Integration with tracing
] }
```

Архітектура Tokio Runtime



```rust
#[tokio::main]  // За замовчуванням: multi-thread
async fn main() { }

#[tokio::main(flavor = "current_thread")]  // Single-thread
async fn main() { }

#[tokio::main(flavor = "multi_thread", worker_threads = 4)]
async fn main() { }
```

```rust
use tokio::runtime::{Builder, Runtime};

// Multi-thread runtime
let rt: Runtime = Builder::new_multi_thread()
    .worker_threads(4)          // Кількість worker threads
    .max_blocking_threads(128)  // Максимум blocking threads
    .thread_name("my-worker")   // Назва потоків
    .thread_stack_size(3 * 1024 * 1024)  // 3MB stack
    .enable_all()               // Всі drivers (io, time)
    .build()
    .expect("Failed to create runtime");

// Current-thread runtime
let rt = Builder::new_current_thread()
    .enable_all()
    .build()
    .unwrap();

// Запуск async коду
rt.block_on(async {
    println!("Hello from runtime!");
    some_async_function().await;
});

// Або отримати handle для spawn
let handle = rt.handle().clone();
handle.spawn(async { /* ... */ });
```

```rust
use tokio::runtime::Runtime;

let rt = Runtime::new().unwrap();

// block_on — запустити future до завершення
let result = rt.block_on(async {
    expensive_async_operation().await
});

// spawn — створити задачу
let handle = rt.spawn(async {
    background_work().await
});

// enter — увійти в контекст runtime
// Потрібно для tokio::spawn поза async контекстом
let _guard = rt.enter();
tokio::spawn(async {
    // Тепер можна spawn'ити
});

// Handle — легкий клон для передачі між потоками
let handle = rt.handle().clone();
std::thread::spawn(move || {
    handle.block_on(async {
        // Використовуємо runtime з іншого потоку
    });
});
```

```rust
use tokio::task::{self, JoinHandle};

// spawn створює нову задачу
// F: Future + Send + 'static
// F::Output: Send + 'static

#[tokio::main]
async fn main() {
    // Базовий spawn
    let handle: JoinHandle<i32> = tokio::spawn(async {
        heavy_computation().await;
        42
    });

    // await на JoinHandle
    match handle.await {
        Ok(value) => println!("Result: {}", value),
        Err(e) if e.is_cancelled() => println!("Task cancelled"),
        Err(e) if e.is_panic() => println!("Task panicked"),
        Err(e) => println!("Task failed: {:?}", e),
    }

    // Ігнорування результату (detached task)
    tokio::spawn(async {
        // Виконається у фоні
        background_logging().await;
    });
    // JoinHandle dropped — задача продовжує працювати!
}
```

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

let handle: JoinHandle<String> = tokio::spawn(async {
    tokio::time::sleep(Duration::from_secs(10)).await;
    "Done".to_string()
});

// abort() — скасувати задачу
handle.abort();

// is_finished() — перевірка завершення (non-blocking)
if handle.is_finished() {
    println!("Task completed");
}

// abort_handle() — отримати AbortHandle для скасування з іншого місця
let abort_handle = handle.abort_handle();
std::thread::spawn(move || {
    std::thread::sleep(Duration::from_secs(5));
    abort_handle.abort();  // Скасувати з іншого потоку
});

// id() — унікальний ідентифікатор задачі
println!("Task ID: {:?}", handle.id());

// await повертає Result<T, JoinError>
let result = handle.await;
```

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

#[tokio::main]
async fn main() {
    let mut set = JoinSet::new();

    // Додаємо задачі
    for i in 0..10 {
        set.spawn(async move {
            tokio::time::sleep(Duration::from_millis(100 * i)).await;
            i * 2
        });
    }

    // Отримуємо результати по мірі завершення
    while let Some(result) = set.join_next().await {
        match result {
            Ok(value) => println!("Task completed: {}", value),
            Err(e) => println!("Task failed: {:?}", e),
        }
    }

    // Або скасувати всі задачі
    set.abort_all();

    // len() — кількість активних задач
    println!("Active tasks: {}", set.len());

    // is_empty() — чи всі завершились
    println!("All done: {}", set.is_empty());
```

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

async fn fetch_all_users(ids: Vec<u32>) -> Vec<User> {
    let mut set = JoinSet::new();

    // Spawn задачу для кожного ID
    for id in ids {
        set.spawn(async move {
            fetch_user(id).await
        });
    }

    // Збираємо всі результати
    let mut users = Vec::new();
    while let Some(result) = set.join_next().await {
        if let Ok(user) = result {
            users.push(user);
        }
    }

    users
}

// JoinSet::spawn_on — spawn на конкретному runtime
let handle = tokio::runtime::Handle::current();
set.spawn_on(async { /* ... */ }, &handle);

// JoinSet::spawn_local — для !Send futures (current_thread)
// set.spawn_local(async { /* ... */ });
```

```rust
use tokio::select;
use tokio_util::sync::CancellationToken;

// CancellationToken — cooperative cancellation
let token = CancellationToken::new();
let cloned_token = token.clone();

let handle = tokio::spawn(async move {
    loop {
        select! {
            _ = cloned_token.cancelled() => {
                println!("Task cancelled gracefully");
                break;
            }
            _ = do_work() => {
                println!("Work done");
            }
        }
    }
});

// Пізніше — скасовуємо
tokio::time::sleep(Duration::from_secs(5)).await;
token.cancel();  // Сигнал скасування

handle.await.unwrap();

// child_token() — ієрархія токенів
let child = token.child_token();
// Скасування parent скасовує всіх children
```

```rust
use tokio::task;
use std::rc::Rc;  // !Send тип

// spawn_local дозволяє !Send futures
// Працює ТІЛЬКИ з current_thread runtime або LocalSet

#[tokio::main(flavor = "current_thread")]
async fn main() {
    let local = task::LocalSet::new();

    local.run_until(async {
        let rc = Rc::new(42);  // Rc !Send

        task::spawn_local(async move {
            println!("Value: {}", rc);  // OK!
        }).await.unwrap();
    }).await;
}

// Або з multi-thread runtime
#[tokio::main]
async fn main() {
    let local = task::LocalSet::new();

    local.run_until(async {
        task::spawn_local(async {
            // !Send код тут
        }).await.unwrap();
    }).await;
}
```

```rust
use tokio::task;

// spawn_blocking — для CPU-bound або sync I/O
// Виконується на окремому thread pool (не async workers!)

#[tokio::main]
async fn main() {
    // CPU-bound робота
    let result = task::spawn_blocking(|| {
        // Цей код блокує потік — це ОК!
        heavy_cpu_computation()
    }).await.unwrap();

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

    // Sync library calls
    let data = task::spawn_blocking(move || {
        let client = blocking_http_client::Client::new();
        client.get("https://api.example.com").send()
    }).await.unwrap();
}

// Default blocking pool: 512 threads
// Налаштування: Builder::max_blocking_threads()
```

```rust
use tokio::task;

// block_in_place — блокує ПОТОЧНИЙ worker thread
// Інші workers продовжують працювати
// ТІЛЬКИ для multi-thread runtime!

#[tokio::main]
async fn main() {
    // block_in_place уникає context switch до blocking pool
    let result = task::block_in_place(|| {
        // Sync код тут
        heavy_computation()
    });

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

// spawn_blocking vs block_in_place:
// spawn_blocking:
//   - Новий потік з blocking pool
//   - Повертає JoinHandle (async)
//   - Для тривалих blocking операцій
//
// block_in_place:
//   - Використовує поточний worker
//   - Синхронно повертає результат
//   - Для коротких blocking операцій
//   - PANIC на current_thread runtime!
```

```rust
use tokio::task::JoinSet;
use std::collections::HashMap;

struct AgentPool {
    tasks: JoinSet<AgentResult>,
    agent_handles: HashMap<AgentId, tokio::task::AbortHandle>,
}

impl AgentPool {
    fn new() -> Self {
        AgentPool {
            tasks: JoinSet::new(),
            agent_handles: HashMap::new(),
        }
    }

    fn spawn_agent(&mut self, agent: Agent) {
        let id = agent.id;
        let abort_handle = self.tasks.spawn(async move {
            agent.run().await
        }).abort_handle();

        self.agent_handles.insert(id, abort_handle);
    }

    fn stop_agent(&mut self, id: AgentId) {
        if let Some(handle) = self.agent_handles.remove(&id) {
            handle.abort();
        }
    }
```

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

enum SwarmCommand {
    SpawnAgent { id: AgentId, config: AgentConfig },
    StopAgent { id: AgentId },
    StopAll,
}

async fn swarm_manager(
    mut commands: mpsc::Receiver<SwarmCommand>,
    shared_state: SharedState,
) {
    let mut pool = AgentPool::new();

    loop {
        tokio::select! {
            // Нові команди
            Some(cmd) = commands.recv() => {
                match cmd {
                    SwarmCommand::SpawnAgent { id, config } => {
                        let agent = Agent::new(id, config, shared_state.clone());
                        pool.spawn_agent(agent);
                        println!("Spawned agent {}", id);
                    }
                    SwarmCommand::StopAgent { id } => {
                        pool.stop_agent(id);
                    }
                    SwarmCommand::StopAll => {
                        pool.tasks.abort_all();
                        break;
```

```rust
use tokio_util::sync::CancellationToken;

struct Agent {
    id: AgentId,
    cancellation: CancellationToken,
    // ...
}

impl Agent {
    async fn run(self) -> AgentResult {
        println!("[Agent {}] Started", self.id);

        loop {
            tokio::select! {
                biased;  // Пріоритет cancellation

                _ = self.cancellation.cancelled() => {
                    println!("[Agent {}] Shutting down...", self.id);
                    self.cleanup().await;
                    return AgentResult::Cancelled;
                }

                _ = self.tick() => {
                    // Нормальна робота
                }
            }
        }
    }

    async fn cleanup(&self) {
```

Типові помилки з Tasks

```
tokio::spawn(async {
    std::thread::sleep(dur); // Блокує!
});
```

✓ spawn_blocking

```
task::spawn_blocking(|| {
    std::thread::sleep(dur);
}).await;
```

Ще типові помилки

```
#[tokio::main(flavor = "current_thread")]
block_in_place(|| { }); // PANIC!
```

✓ spawn_blocking

```
#[tokio::main(flavor = "current_thread")]
spawn_blocking(|| { }).await; // OK!
```

Best Practices: Tasks

✓ Використовуйте JoinSet для груп задач
✓ CancellationToken для graceful shutdown
✓ spawn_blocking для CPU-bound та sync I/O
✓ Завжди обробляйте JoinError (panic, cancel)
✓ Встановіть timeout для зовнішніх операцій
✓ Логуйте lifecycle задач для debugging

❌ Не ігноруйте JoinHandle (результат втрачено)
❌ Не блокуйте в async tasks
❌ Не використовуйте Rc в spawned tasks
❌ Не використовуйте block_in_place з current_thread

🤖 Для MAC:
• JoinSet для управління пулом агентів
• CancellationToken ієрархія для shutdown
• spawn_blocking для path planning

Підсумок: Частина 1

Tokio Runtime:
- Multi-thread — production, parallelism
- Current-thread — CLI, tests, !Send
- Builder для детального налаштування

Tasks:
- spawn — async задачі (Send + 'static)
- JoinHandle — результат та контроль
- JoinSet — група задач
- spawn_local — для !Send futures
- spawn_blocking — CPU-bound
- block_in_place — коротке blocking

Cancellation:
- CancellationToken — cooperative
- abort() — force cancel
- Drop JoinHandle — detach (не cancel!)

→ Частина 2: Channels, Sync, Time, I/O

Лекція 17 (продовження)

Tokio: Channels, Sync, Time

Async примітиви синхронізації та утиліти

mpsc • broadcast • watch • oneshot • Mutex • RwLock • Semaphore

🤖 Комунікація та координація асинхронного рою

Частина 2: Sync та Time

О. С. Бичков • 2025

План лекції (Частина 2)

1. tokio::sync — огляд
2. mpsc — Multi-producer Single-consumer
3. mpsc bounded vs unbounded
4. broadcast — Multi-producer Multi-consumer
5. watch — Single value broadcast
6. oneshot — One-time channel
7. tokio::sync::Mutex
8. tokio::sync::RwLock
9. Semaphore — обмеження concurrency
10. Notify — async notification
11. Barrier — sync point
12. tokio::time — таймери
13. 🤖 Комунікація в рої
14. 🤖 Rate limiting агентів
15. 🤖 Повний async рій
16. Підсумок

tokio::sync — async синхронізація

Примітив	Опис	Producers	Consumers
mpsc	Канал повідомлень	Multi	Single
broadcast	Broadcast канал	Multi	Multi (clone)
watch	Single value	Single	Multi
oneshot	One-time channel	Single	Single
Mutex	Async mutex	—	—
RwLock	Async read-write lock	—	—
Semaphore	Permit-based sync	—	—
Notify	Async notification	—	—
Barrier	Sync point	—	—

Всі примітиви async-first: .await замість blocking!

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

#[tokio::main]
async fn main() {
    // Bounded channel — з backpressure
    let (tx, mut rx) = mpsc::channel::<String>(100);  // buffer 100

    // Sender можна клонувати
    let tx2 = tx.clone();

    tokio::spawn(async move {
        tx.send("Hello".to_string()).await.unwrap();
    });

    tokio::spawn(async move {
        tx2.send("World".to_string()).await.unwrap();
    });

    // Отримання
    while let Some(msg) = rx.recv().await {
        println!("Got: {}", msg);
    }
}

// send().await блокує async якщо буфер повний!
// recv().await повертає None коли всі Sender dropped
```

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

let (tx, mut rx) = mpsc::channel::<i32>(10);

// send() — async, чекає якщо буфер повний
tx.send(42).await?;

// try_send() — non-blocking, помилка якщо повний
match tx.try_send(42) {
    Ok(()) => println!("Sent!"),
    Err(mpsc::error::TrySendError::Full(val)) => {
        println!("Buffer full, value {} not sent", val);
    }
    Err(mpsc::error::TrySendError::Closed(val)) => {
        println!("Channel closed");
    }
}

// send_timeout() — з timeout (tokio >= 1.22)
use tokio::time::Duration;
match tx.send_timeout(42, Duration::from_secs(1)).await {
    Ok(()) => println!("Sent!"),
    Err(mpsc::error::SendTimeoutError::Timeout(val)) => {},
    Err(mpsc::error::SendTimeoutError::Closed(val)) => {},
}

// is_closed() — перевірка чи закритий
if tx.is_closed() { println!("Receiver dropped"); }
```

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

// Unbounded — без ліміту буфера
// send() НЕ async — ніколи не блокує!
let (tx, mut rx) = mpsc::unbounded_channel::<String>();

// UnboundedSender::send() — синхронний!
tx.send("Hello".to_string()).unwrap();  // Без .await!

// Отримання — як звичайно
while let Some(msg) = rx.recv().await {
    println!("{}", msg);
}

// ⚠️ Небезпека: необмежений буфер може призвести до OOM!
// Використовуйте тільки коли:
// - Точно знаєте що producer не швидший за consumer
// - Потрібно уникнути async в sender

// bounded vs unbounded:
// bounded:   send().await, backpressure, safe
// unbounded: send() sync,  no backpressure, OOM risk
```

Рекомендація: завжди використовуйте bounded якщо можливо!

```rust
use tokio::sync::broadcast;

#[tokio::main]
async fn main() {
    // broadcast — кожен subscriber отримує КОЖНЕ повідомлення
    let (tx, _) = broadcast::channel::<String>(16);

    // Створюємо subscribers
    let mut rx1 = tx.subscribe();
    let mut rx2 = tx.subscribe();

    // Sender
    tx.send("Hello everyone!".to_string()).unwrap();

    // Обидва отримають
    assert_eq!(rx1.recv().await.unwrap(), "Hello everyone!");
    assert_eq!(rx2.recv().await.unwrap(), "Hello everyone!");

    // Lagging receiver — якщо пропустив повідомлення
    for _ in 0..20 {
        tx.send(format!("msg")).unwrap();
    }

    match rx1.recv().await {
        Ok(msg) => println!("Got: {}", msg),
        Err(broadcast::error::RecvError::Lagged(n)) => {
            println!("Missed {} messages", n);
        }
        Err(broadcast::error::RecvError::Closed) => {},
    }
```

```rust
use tokio::sync::watch;

#[tokio::main]
async fn main() {
    // watch — single value, many observers
    // Тільки останнє значення зберігається!
    let (tx, rx) = watch::channel("initial".to_string());

    // Читання поточного значення
    println!("Current: {}", *rx.borrow());

    // Очікування зміни
    let mut rx2 = rx.clone();
    tokio::spawn(async move {
        loop {
            // changed() чекає на НОВЕ значення
            rx2.changed().await.unwrap();
            println!("Value changed: {}", *rx2.borrow());
        }
    });

    // Оновлення значення
    tx.send("updated".to_string()).unwrap();
    tx.send("final".to_string()).unwrap();

    // send_modify — update in place
    tx.send_modify(|val| val.push_str(" modified"));
}

// Ідеально для: config updates, state observation, shutdown signals
```

```rust
use tokio::sync::oneshot;

#[tokio::main]
async fn main() {
    // oneshot — один send, один receive
    let (tx, rx) = oneshot::channel::<String>();

    tokio::spawn(async move {
        // tx.send() споживає tx — можна викликати лише раз
        tx.send("Result".to_string()).unwrap();
    });

    // Очікування результату
    let result = rx.await.unwrap();
    println!("Got: {}", result);
}

// Request-Response pattern
async fn request_response(service: &Service) -> Response {
    let (tx, rx) = oneshot::channel();

    service.send(Request {
        data: "query",
        response: tx,  // Канал для відповіді
    }).await;

    rx.await.unwrap()  // Чекаємо відповідь
}

// closed() — перевірити чи receiver dropped
```

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

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

    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        handles.push(tokio::spawn(async move {
            // lock().await — async!
            let mut guard = counter.lock().await;
            *guard += 1;
            // guard dropped — unlock
        }));
    }

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

    println!("Result: {}", *counter.lock().await);
}

// tokio::sync::Mutex vs std::sync::Mutex:
// tokio: lock().await — НЕ блокує потік
// std:   lock().unwrap() — БЛОКУЄ потік

// Коли що:
// - Короткі критичні секції → std::sync::Mutex (швидше)
```

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

#[tokio::main]
async fn main() {
    let data = Arc::new(RwLock::new(vec![1, 2, 3]));

    // Багато читачів одночасно
    let read_handle = {
        let data = Arc::clone(&data);
        tokio::spawn(async move {
            let guard = data.read().await;  // Read lock
            println!("Data: {:?}", *guard);
        })
    };

    // Один писач
    let write_handle = {
        let data = Arc::clone(&data);
        tokio::spawn(async move {
            let mut guard = data.write().await;  // Write lock
            guard.push(4);
        })
    };

    read_handle.await.unwrap();
    write_handle.await.unwrap();
}

// Методи: read(), write(), try_read(), try_write()
```

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

// Semaphore — обмежує кількість одночасних операцій
#[tokio::main]
async fn main() {
    // Максимум 3 одночасні операції
    let semaphore = Arc::new(Semaphore::new(3));

    let mut handles = vec![];

    for i in 0..10 {
        let sem = Arc::clone(&semaphore);
        handles.push(tokio::spawn(async move {
            // Отримуємо permit (чекаємо якщо немає вільних)
            let _permit = sem.acquire().await.unwrap();

            println!("Task {} started", i);
            tokio::time::sleep(Duration::from_secs(1)).await;
            println!("Task {} done", i);

            // permit dropped — звільняється автоматично
        }));
    }

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

// OwnedSemaphorePermit — для передачі між tasks
let permit = semaphore.clone().acquire_owned().await.unwrap();
```

```rust
use tokio::sync::Semaphore;

// Pattern 1: Rate limiting
struct RateLimiter {
    semaphore: Arc<Semaphore>,
}

impl RateLimiter {
    fn new(max_concurrent: usize) -> Self {
        RateLimiter {
            semaphore: Arc::new(Semaphore::new(max_concurrent)),
        }
    }

    async fn acquire(&self) -> SemaphorePermit<'_> {
        self.semaphore.acquire().await.unwrap()
    }
}

// Pattern 2: Connection pool
struct ConnectionPool {
    connections: Mutex<Vec<Connection>>,
    semaphore: Semaphore,
}

impl ConnectionPool {
    async fn get(&self) -> PooledConnection {
        let permit = self.semaphore.acquire().await.unwrap();
        let conn = self.connections.lock().await.pop().unwrap();
        PooledConnection { conn, permit }
```

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

#[tokio::main]
async fn main() {
    let notify = Arc::new(Notify::new());

    // Waiter
    let notify2 = Arc::clone(&notify);
    let waiter = tokio::spawn(async move {
        println!("Waiting for notification...");
        notify2.notified().await;
        println!("Notified!");
    });

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

    // Notifier
    notify.notify_one();   // Розбудити одного
    // notify.notify_waiters();  // Розбудити всіх

    waiter.await.unwrap();
}

// Корисно для:
// - Signaling "work available"
// - Wake up idle workers
// - Event notification

// Різниця з channel:
```

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

#[tokio::main]
async fn main() {
    let barrier = Arc::new(Barrier::new(3));  // 3 учасники

    let mut handles = vec![];

    for i in 0..3 {
        let b = Arc::clone(&barrier);
        handles.push(tokio::spawn(async move {
            println!("Task {} before barrier", i);

            // Всі чекають поки всі 3 досягнуть цієї точки
            let result = b.wait().await;

            // is_leader() — один task отримує true
            if result.is_leader() {
                println!("Task {} is the leader", i);
            }

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

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

// Ідеально для фазової синхронізації
```

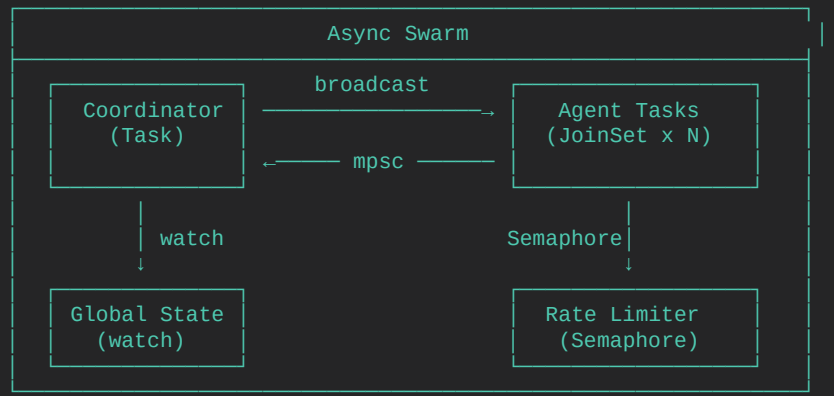
```rust
use tokio::time::{sleep, interval, timeout, Duration, Instant};

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

    // interval — періодичний таймер
    let mut interval = interval(Duration::from_millis(100));
    for _ in 0..5 {
        interval.tick().await;
        println!("Tick!");
    }

    // timeout — обмеження часу
    match timeout(Duration::from_secs(5), slow_operation()).await {
        Ok(result) => println!("Result: {:?}", result),
        Err(_) => println!("Timeout!"),
    }

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

    // sleep_until — до конкретного моменту
    let deadline = Instant::now() + Duration::from_secs(10);
    tokio::time::sleep_until(deadline).await;
}
```

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

#[tokio::main]
async fn main() {
    // Базовий interval
    let mut interval = interval(Duration::from_millis(100));

    // Перший tick() повертається ОДРАЗУ!
    interval.tick().await;  // Instant return
    interval.tick().await;  // 100ms later

    // interval_at — починати з конкретного часу
    let start = Instant::now() + Duration::from_secs(1);
    let mut interval = interval_at(start, Duration::from_millis(100));

    // MissedTickBehavior — що робити якщо пропустили tick
    let mut interval = interval(Duration::from_millis(100));
    interval.set_missed_tick_behavior(MissedTickBehavior::Skip);
    // Burst — викликати пропущені tick'и швидко (default)
    // Delay — затримати наступний tick
    // Skip — пропустити пропущені

    // period() — отримати період
    println!("Period: {:?}", interval.period());

    // reset() — скинути таймер
    interval.reset();
}
```

```rust
use tokio::sync::{mpsc, broadcast, watch};

struct SwarmCommunication {
    // Агенти → Координатор (mpsc)
    agent_to_coord: mpsc::Sender<AgentMessage>,

    // Координатор → Всі агенти (broadcast)
    coord_broadcast: broadcast::Sender<BroadcastCommand>,

    // Глобальний стан (watch)
    global_state: watch::Sender<GlobalState>,
}

struct AgentChannels {
    // Надсилання до координатора
    to_coordinator: mpsc::Sender<AgentMessage>,

    // Отримання broadcast
    from_broadcast: broadcast::Receiver<BroadcastCommand>,

    // Спостереження за станом
    state_watch: watch::Receiver<GlobalState>,

    // Персональні команди (mpsc)
    personal_commands: mpsc::Receiver<Command>,
}

// Coordinator надсилає:
// - broadcast для всіх: "All agents retreat!"
// - watch для стану: updated world map
```

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

struct SwarmRateLimiter {
    // Максимум одночасних сканувань
    scan_limit: Arc<Semaphore>,
    // Максимум одночасних атак
    attack_limit: Arc<Semaphore>,
    // Максимум повідомлень на секунду
    message_rate: Arc<Semaphore>,
}

impl SwarmRateLimiter {
    fn new() -> Self {
        SwarmRateLimiter {
            scan_limit: Arc::new(Semaphore::new(10)),    // 10 одночасних
            attack_limit: Arc::new(Semaphore::new(3)),   // 3 одночасних
            message_rate: Arc::new(Semaphore::new(100)), // 100/sec
        }
    }
}

impl Agent {
    async fn scan(&self) -> ScanResult {
        let _permit = self.rate_limiter.scan_limit.acquire().await.unwrap();
        // Тепер можемо сканувати
        self.do_scan().await
    }

    async fn attack(&self, target: TargetId) {
```

MAC: Архітектура async рою



```
struct AsyncAgent {
    id: AgentId,
    channels: AgentChannels,
    rate_limiter: Arc<SwarmRateLimiter>,
    cancellation: CancellationToken,
}

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

        loop {
            select! {
                biased;

                _ = self.cancellation.cancelled() => break,

                Ok(cmd) = self.channels.from_broadcast.recv() => {
                    self.handle_broadcast(cmd).await;
                }

                Some(cmd) = self.channels.personal_commands.recv() => {
                    self.handle_command(cmd).await;
                }

                Ok(_) = self.channels.state_watch.changed() => {
                    let state = self.channels.state_watch.borrow().clone();
                    self.update_local_state(state);
                }
```

Порівняння tokio каналів

Канал	Producers	Consumers	Use Case
mpsc	Multi	Single	Agent → Coordinator
broadcast	Multi	Multi	Global announcements
watch	Single	Multi	State observation
oneshot	Single	Single	Request-Response

🤖 Рекомендації для MAC:
- mpsc: агенти надсилають звіти координатору
- broadcast: координатор надсилає команди всім
- watch: глобальний стан рою (карта світу)
- oneshot: request-response патерн

Комбінуйте канали для різних типів комунікації!

Best Practices: Sync

✓ Bounded channels для backpressure
✓ watch для shared state observation
✓ Semaphore для rate limiting
✓ CancellationToken для graceful shutdown
✓ std::sync::Mutex для коротких критичних секцій
✓ tokio::sync::Mutex якщо .await всередині

❌ Не використовуйте unbounded без причини (OOM)
❌ Не тримайте MutexGuard через .await (std::sync)
❌ Не забувайте про lagging receivers (broadcast)

🤖 Патерни для MAC:
• mpsc (bounded, 1000) для messages
• broadcast для commands
• watch для global state
• Semaphore для resource limits

Підсумок лекції

tokio::sync — async синхронізація:
- mpsc — multi-producer, bounded/unbounded
- broadcast — multi-consumer broadcasting
- watch — single value observation
- oneshot — one-time response
- Mutex, RwLock — async locks
- Semaphore — concurrency limiting
- Notify, Barrier — coordination

tokio::time — таймери:
- sleep, interval, timeout
- MissedTickBehavior для intervals

🤖 MAC: комбінація каналів та примітивів для масштабованої async комунікації!

→ Наступна лекція: Async Channels та синхронізація

Завдання для самостійної роботи

1. mpsc Chat:
   - Кілька "клієнтів" надсилають повідомлення
   - Один "сервер" обробляє
   - Bounded channel з backpressure

2. broadcast Announcements:
   - Coordinator надсилає broadcast
   - 10 агентів отримують
   - Handle lagging receivers

3. Rate Limiter:
   - Semaphore з 5 permits
   - 20 tasks намагаються отримати permit
   - Вимірюйте час очікування

4. Async Agent:
   - select! на tick + commands + shutdown
   - CancellationToken для graceful stop
   - interval 100ms

5. Full Swarm:
   - Coordinator + 100 agents
   - mpsc + broadcast + watch
   - Rate limiting
   - Graceful shutdown

Tokio опановано!

Runtime • Tasks • Channels • Sync • Time

Питання?