Лекція 29.
Багатопоточність.

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

План на сьогодні

1. Що таке процес?
2. Що таке потік?
3. Перемикання між потоками
4. Створення потоку в C++
5. Очікування завершення потоків
6. mutex, lock_guard, atomic, semaphore
7. Основні методи

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Що таке процес?

Що таке процес?

Процес — це незалежна програма, яка виконується, має унікальний ідентифікатор (Process ID) та власний адресний простір у пам’яті (код, дані, стек, купа).

- Кожен процес працює у власному адресному просторі, не має прямого доступу до інших.
- Обмін даними з іншими процесами відбувається через системні виклики (IPC - міжпроцесорна комунікація).
- Одна програма зазвичай запускається як один процес, але може створювати й кілька.
- Якщо процес аварійно завершується, це не впливає на інші.
- Кожен процес обов’язково містить щонайменше один потік, який виконує інструкції програми.

Приклад: Запуск браузера і текстового редактора створює два окремі процеси.

Що таке потік?

Що таке потік?

Потік — це частина процесу, яка виконує певну послідовність інструкцій і ділить спільний адресний простір з іншими потоками цього процесу.

- Кожен процес має щонайменше один потік — основний.
- Потоки розпаралелюють виконання завдань у межах одного процесу.
- Потоки мають спільний доступ до пам’яті процесу, змінних і ресурсів.
- На відміну від процесів, взаємодія потоків швидша і не вимагає системних викликів.
- Тому потоки використовуються частіше, ніж окремі процеси.

Приклад: У браузері: один потік обробляє інтерфейс, інший — мережу, третій — рендеринг сторінки.

```
Процес:
├── Код, Дані, Купа (спільні для всіх потоків)
├── Потік 1 (Головний)
│   └── Стек, регістри
├── Потік 2 (Додатковий)
│   └── Свій стек, регістри
└── Потік 3 (Додатковий)
    └── Свій стек, регістри
```

Відмінності між процесами та потоками

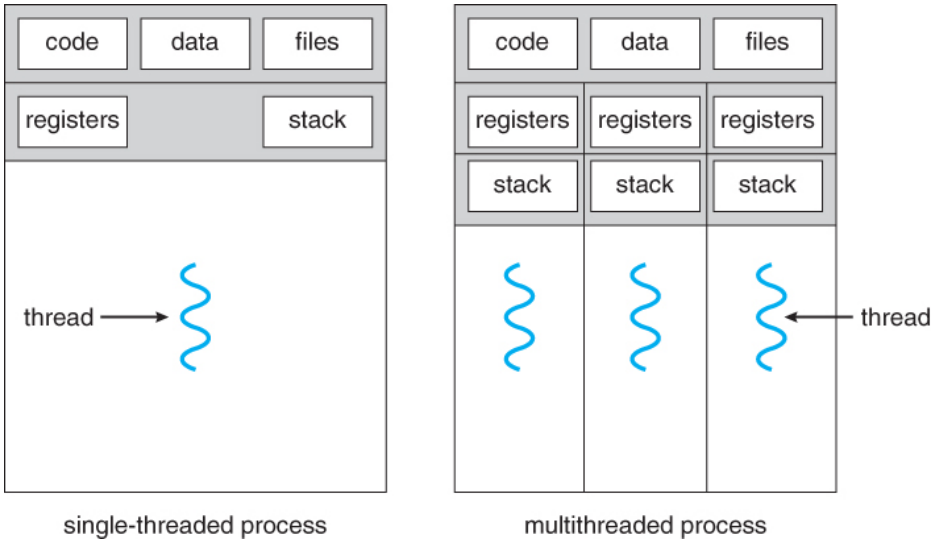
Процеси — це незалежні одиниці виконання, які мають власну пам’ять і контекст, тоді як **потоки** — це легковагові сегменти процесу, які працюють у спільному середовищі.

Процеси:

- Мають власну структуру PCB (Process Control Block), яка зберігає регістри, пріоритет, стан тощо.
- Ізольовані: не ділять пам'ять із іншими процесами.
- Можуть створювати дочірні процеси.
- Потребують більше часу для створення і завершення.
- Можливі стани: новий, готовий, запущено, очікує, завершено та призупинено.

Потоки:

- Є частиною процесу, можуть бути кількома в межах одного процесу.
- Ділять пам’ять і ресурси процесу з іншими потоками.
- Створюються і завершуються швидше, ніж процеси.
- Можливі стани: виконується, готовий, заблокований



Порівняльна таблиця

Критерій	Потік (Thread)	Процес (Process)
Означення	Найменша одиниця виконання в межах процесу	Програма, що виконується з власним середовищем
Контекст виконання	Ділить адресний простір з іншими потоками в процесі	Має власний адресний простір, ресурси, пам'ять
Споживання ресурсів	Легкий (менше ресурсів)	Важчий (більше ресурсів)
Швидкість створення	Швидше	Повільніше
Комунікація	Легка — через спільну пам'ять	Складніша — через канали, сокети, пайпи (IPC)
Незалежність	Залежний — креш одного потоку може вплинути на весь процес	Незалежний — креш одного процесу не впливає на інші
Приклад	std::thread у C++	Веб-браузер як окремий процес

Перемикання між потоками (context switch)

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Контекст потоку

Контекст потоку — це інформація, яка потрібна операційній системі для зупинки, відновлення або перемикання потоку виконання.

Компонент	Опис
Регістри процесора	Значення всіх основних регістрів: PC (Program Counter), SP (Stack Pointer), BP (Base Pointer), загальні регістри тощо
Лічильник команд (PC)	Вказує, яку інструкцію виконувати наступною
Стек потоку	Локальні змінні, адреси повернення з функцій, параметри
Прапори стану процесора	Результати попередніх операцій (нуль, переповнення тощо)
Ідентифікатор потоку (TID)	Унікальний ID потоку
Інформація про пріоритет/стан	Статус потоку: готовий, чекає, виконується
(Іноді) Сигнали або флаги переривань	Для асинхронного керування

Перемикання контексту потоку

Коли ОС перемикає CPU з одного потоку на інший:

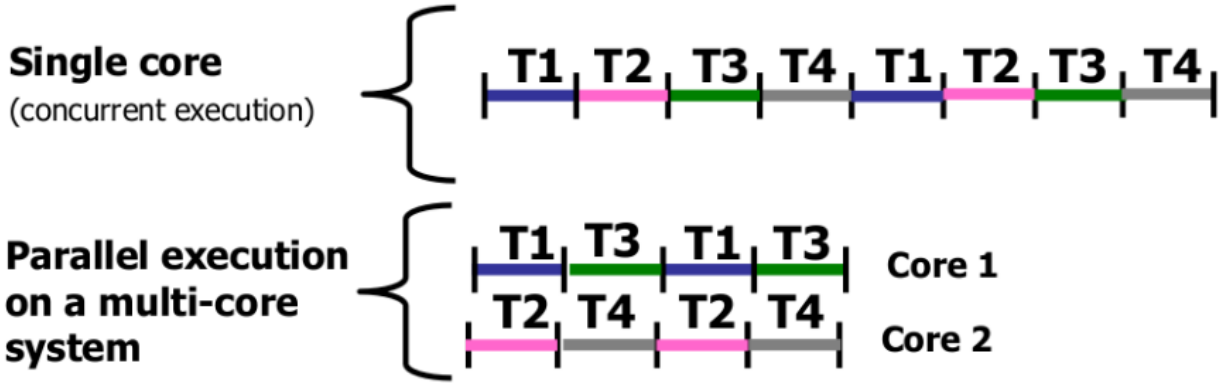
- Зберігає регістри поточного потоку в його контексті.
- Завантажує регістри нового потоку, щоб продовжити з того місця, де він зупинився.

Спрощена схема:

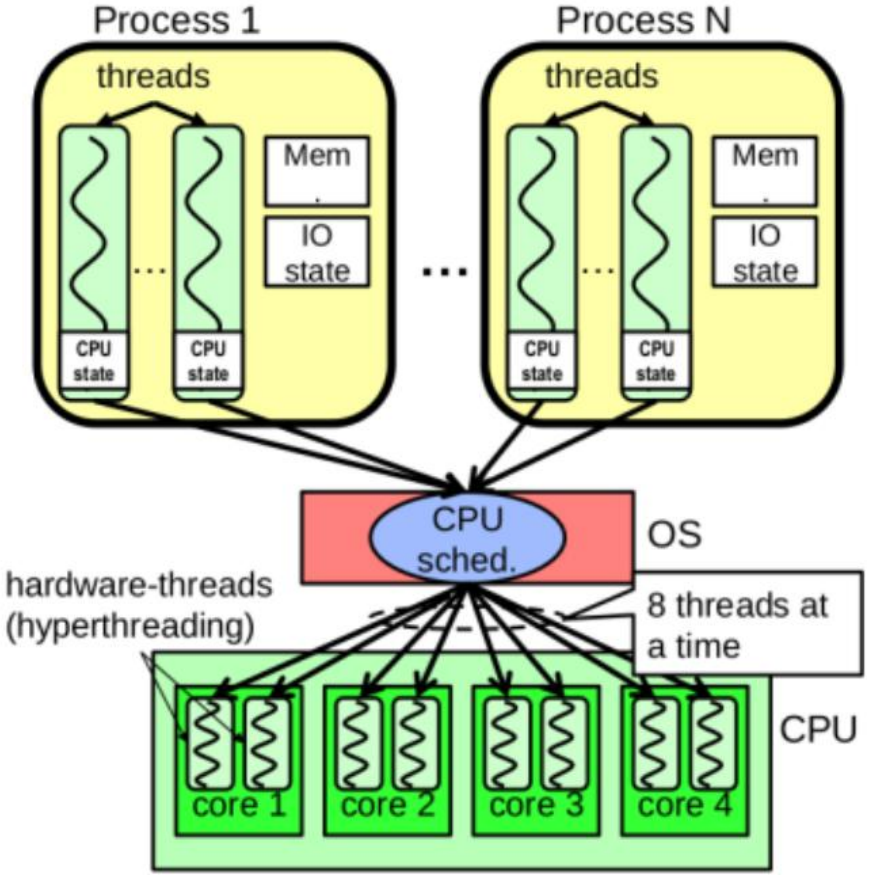
```
[CPU]
   |
   |--> Виконує Потік A → регістри з контекстом A
   |
   |--> Перемикання → зберігаємо регістри A, завантажуємо регістри B
   |
   |--> Виконує Потік B → з новим набором регістрів
```

Concurrency vs parallelism

- **Concurrency** - здатність системи керувати кількома завданнями одночасно, перемикаючись між ними
- **Parallelism** - здатність системи виконувати кілька завдань фізично одночасно, наприклад, на різних ядрах процесора.



Загальна архітектура



Створення потоку в C++

FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY

Створення потоку в C++

Багатопоточність — це здатність програми одночасно виконувати кілька частин коду (**потоків**) для ефективнішого використання процесора.

- До C++11 для потоків використовували POSIX (pthreads) — потрібно вручну звільняти ресурси, винятки не підтримуються.
- Починаючи з C++11, стандартна бібліотека надає клас std::thread (у <thread>).
- Потік запускається автоматично після створення об’єкта std::thread.

Синтаксис:

```
std::thread thread_object(callable);
```

callable — код, який виконується в потоці.

Можливі типи callable-об'єктів:

- Вказівник на функцію
- Лямбда-вираз
- Функціональний об’єкт
- Нестатична функція-член
- Статична функція-член

Приклади запуску потоків.

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Вказівник на функцію

Один із найпростіших способів запуску потоку — передати в std::thread звичайну функцію через її **вказівник**. Такий спосіб підходить, коли логіка обробки вже реалізована у вигляді функції.

```
#include <iostream>
#include <sstream>
#include <thread>
using namespace std;
void printMessage(string name, int count) {
    for (int i = 0; i < count; ++i) {
        stringstream ss;
        ss << "Hello from " << name << " (" << i + 1 << ")" << endl;
        cout << ss.str();
    }
}
int main() {
    thread threadObj(printMessage, "Thread A", 3);
    // Основний потік також виконує свою роботу
    printMessage("Main thread", 2);
    threadObj.join(); // Очікуємо завершення додаткового потоку
    return 0;
}
```

Можливий вивід:

```
Hello from Main thread (1)
Hello from Thread A (1)
Hello from Thread A (2)
Hello from Main thread (2)
Hello from Thread A (3)
```

Лямбда-вираз

Окрім функцій, потік у C++ можна запускати за допомогою **лямбда-виразів**. Це зручно, коли логіку потоку не потрібно винести в окрему функцію, або вона є короткою й локальною.

```
int main() {
    string name = "Lambda Thread";
    thread threadObj([name]() {
        for (int i = 0; i < 3; ++i) {
            stringstream ss;
            ss << "Hello from " << name << " (" << i + 1 << ")" << endl;
            cout << ss.str();
        }
    });
    // Основний потік
    for (int i = 0; i < 2; ++i) {
        stringstream ss;
        cout << "Hello from Main thread (" << i + 1 << ")" << endl;
        cout << ss.str();
    }
    threadObj.join();
    return 0;
}
```

Можливий вивід:

```
Hello from Main thread (1)
Hello from Lambda Thread (1)
Hello from Lambda Thread (2)
Hello from Main thread (2)
Hello from Lambda Thread (3)
```

Фунтор

Функціональний об’єкт (або **функтор**) — це клас, який перевантажує оператор (). Такий об’єкт поводиться як функція і може бути використаний для запуску потоку.

```
class MessagePrinter {
    string name;
    int count;
public:
    MessagePrinter(string n, int c) : name(n), count(c) {}
    void operator()() {
        for (int i = 0; i < count; ++i) {
            stringstream ss;
            ss << "Hello from " << name << " (" << i + 1 << ")" << endl;
            cout << ss.str();
        }
    }
};
int main() {
    MessagePrinter printer("Functor Thread", 3);
    thread threadObj(printer); // або: thread threadObj(MessagePrinter("Functor Thread", 3));
    // Основний потік
    for (int i = 0; i < 2; ++i) {
        stringstream ss;
        ss << "Hello from Main thread (" << i + 1 << ")" << endl;
        cout << ss.str();
    }
    threadObj.join();
    return 0;
}
```

Можливий вивід:

```
Hello from Functor Thread (1)
Hello from Main thread (1)
Hello from Functor Thread (2)
Hello from Main thread (2)
Hello from Functor Thread (3)
```

Нестатична функція-член

У C++ можна запускати потік, викликаючи **нестатичну функцію-член** класу.

```
class Greeter {
public:
    void greet(string name, int count) {
        for (int i = 0; i < count; ++i) {
            stringstream ss;
            ss << "Hello from " << name << " (" << i + 1 << ")" << endl;
            cout << ss.str();
        }
    }
};
int main() {
    Greeter greeter;
    thread threadObj(&Greeter::greet, &greeter, "Member Function", 3);
    for (int i = 0; i < 2; ++i) {
        stringstream ss;
        ss << "Hello from Main thread (" << i + 1 << ")" << endl;
        cout << ss.str();
    }
    threadObj.join();
    return 0;
}
```

Можливий вивід:

```
Hello from Member Function (1)
Hello from Main thread (1)
Hello from Member Function (2)
Hello from Main thread (2)
Hello from Member Function (3)
```

Статична функція-член

Статичну функцію-члена класу можна передати в std::thread так само, як звичайну глобальну функцію — без об'єкта класу.

```
class Speaker {
public:
    static void sayHello(string name, int count) {
        for (int i = 0; i < count; ++i) {
            stringstream ss;
            cout << "Hello from " << name << " (" << i + 1 << ")" << endl;
            cout << ss.str();
        }
    }
};
int main() {
    thread threadObj(&Speaker::sayHello, "Static Thread", 3);
    for (int i = 0; i < 2; ++i) {
        stringstream ss;
        cout << "Hello from Main thread (" << i + 1 << ")" << endl;
        cout << ss.str();
    }
    threadObj.join();
    return 0;
}
```

Можливий вивід:

```
Hello from Main thread (1)
Hello from Static Thread (1)
Hello from Static Thread (2)
Hello from Main thread (2)
Hello from Static Thread (3)
```

Очікування завершення потоків

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Очікування завершення потоків

Після запуску потоку іноді потрібно дочекатися його завершення, перш ніж продовжувати роботу далі (наприклад, для ініціалізації графічного інтерфейсу чи завантаження ресурсів).

- Для цього використовується метод join() з класу std::thread.
- Метод блокує поточний потік до завершення вказаного потоку.
- Якщо не викликати join() або detach(), об’єкт потоку в деструкторі призведе до аварійного завершення програми.

```
void greet(string name) {
    for (int i = 0; i < 3; ++i) {
        cout << "Hello from " << name << " (" << i + 1 << ")" << endl;
    }
}

int main() {
    thread t1(greet, "Worker Thread");
    // Очікуємо завершення потоку t1
    t1.join();
    cout << "Hello from Main thread (after join)" << endl;
    return 0;
}
```

Очікування завершення потоків

```
// Звичайна функція
void foo(int count) {
    for (int i = 0; i < count; ++i)
        stringstream ss;
        ss << "Hello from function pointer (" << i + 1 << ")\n";
        cout << ss.str();
}
// Функтор
class ThreadFunctor {
public:
    void operator()(int count) {
        for (int i = 0; i < count; ++i)
            stringstream ss;
            ss << "Hello from functor (" << i + 1 << ")\n";
            cout << ss.str();
    }
};
// Клас з методами
class Greeter {
public:
    void sayHi() {
        cout << "Hello from non-static member function\n";
    }
    static void sayHelloStatic() {
        cout << "Hello from static member function\n";
    }
};
```

```
int main() {
    cout << "Launching 5 threads using different callables...\n";
    thread t1(foo, 2);
    thread t2(ThreadFunctor(), 2);

    auto lambda = [](int count) {
        for (int i = 0; i < count; ++i)
            stringstream ss;
            ss<< "Hello from lambda (" << i + 1 << ")\n";
            cout << ss.str();
    };

    thread t3(lambda, 2);
    Greeter greeter;
    thread t4(&Greeter::sayHi, &greeter);
    thread t5(&Greeter::sayHelloStatic);

    // Очікування завершення всіх потоків
    t1.join();
    t2.join();
    t3.join();
    t4.join();
    t5.join();
    return 0;
}
```

Можливий вивід:

```
Hello from function pointer (1)
Hello from functor (1)
Hello from lambda (1)
Hello from non-static member function
Hello from static member function
Hello from function pointer (2)
Hello from functor (2)
Hello from lambda (2)
```

Доступ потоків до спільних ресурсів. Mutex.

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Доступ потоків до спільних ресурсів. Mutex

Race condition — це загальний термін, який описує ситуацію, коли результат виконання програми залежить від порядку або моменту доступу до спільних ресурсів (пам'яті, файлів тощо) кількома потоками чи процесами.
Data race — це підмножина race condition, виникає коли два або більше потоки одночасно читають і записують в одну й ту ж змінну, без належної синхронізації.

- Для синхронізації доступу використовується mutex (взаємне виключення).
- У C++ доступний клас std::mutex із заголовку <mutex>.
- Mutex блокує доступ до ресурсу, поки ним користується один потік.
- Після завершення критичної секції mutex розблоковується.

Синтаксис використання std::mutex

Для уникнення одночасного доступу до спільних ресурсів між потоками в C++ використовують механізм взаємного виключення — std::mutex. Його застосування складається з трьох простих кроків:

- Створення м’ютексу. Оголошується об’єкт типу std::mutex, який контролює доступ до ресурсу.
- Блокування доступу. Метод lock() дозволяє потоку увійти в критичну секцію та забороняє іншим потокам одночасно її виконувати.
- Розблокування. Метод unlock() відновлює доступ до заблокованого ресурсу, дозволяючи іншим потокам продовжити роботу.

*Якщо unlock() не буде викликано — інші потоки зависнуть у очікуванні, що може призвести до взаємного блокування (**deadlock**).*

Data race: приклад без mutex

Створимо глобальну змінну number і функцію increment(), яка збільшує її на 1 мільйон разів. Запускаємо два потоки, які одночасно виконують цю функцію. Теоретично, підсумкове значення має бути 2 000 000, але через відсутність синхронізації виникає умова змагання, тому реальний результат є непередбачуваним.

```
// Спільна змінна
int number = 0;

// Функція інкременту
void increment() {
    for (int i = 0; i < 1000000; ++i) {
        number++;
    }
}

int main() {
    thread t1(increment);
    thread t2(increment);

    t1.join();
    t2.join();

    cout << "Number after execution of t1 and t2 is " << number << endl;
    return 0;
}
```

Очікуване значення: 2000000
Реальне значення: змінне (наприклад: 1742398, 1887712...)

Data race: приклад з mutex

Щоб уникнути умови змагання при доступі до спільної змінної з кількох потоків, використовують std::mutex. У цьому прикладі обидва потоки виконують функцію increment(), але доступ до змінної number синхронізований за допомогою блокування через mtx.lock() і mtx.unlock().

```
// М'ютекс для синхронізації
mutex mtx;

// Спільна змінна
int number = 0;

void increment() {
    mtx.lock();
    for (int i = 0; i < 1000000; ++i) {
        number++;
    }
    mtx.unlock();
}
```

```
int main() {
    thread t1(increment);
    thread t2(increment);
    t1.join();
    t2.join();
    cout << "Number after execution of t1 and t2 is " << number << endl;
    return 0;
}
```

Очікуваний вивід:
Number after execution of t1 and t2 is 2000000

Захист доступу до ресурсу за допомогою lock_guard.

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Захист доступу до ресурсу за допомогою std::lock_guard

Щоб не забути викликати unlock(), у C++ зручно використовувати обгортку std::lock_guard. Вона автоматично блокує м'ютекс при створенні об'єкта і розблокує його при виході з області видимості — навіть у разі виключення.

```
mutex mtx;
int number = 0;

void increment() {
    lock_guard<mutex> lock(mtx);
    for (int i = 0; i < 1000000; ++i) {
        number++;
    }
    // unlock() викликається автоматично
}

int main() {
    thread t1(increment);
    thread t2(increment);
    t1.join();
    t2.join();
    cout << "Number after execution of t1 and t2 is " << number << endl;
    return 0;
}
```

Очікуваний вивід:

Number after execution of t1 and t2 is 2000000

Синхронізація з використанням std::atomic

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Синхронізація з використанням std::atomic

У випадках, коли потоки працюють лише з простими типами даних, такими як int, ефективнішою альтернативою mutex є std::atomic. Цей клас гарантує безпечний доступ без блокування, що дозволяє уникнути надмірних витрат на синхронізацію.

На відміну від mutex, std::atomic не вимагає ручного блокування і розблокування. Операції на атомарних змінних виконуються як єдина, неподільна інструкція, що гарантує відсутність race condition.

```
atomic<int> number = 0;

void increment() {
    for (int i = 0; i < 1000000; ++i) {
        number++;
    }
}

int main() {
    thread t1(increment);
    thread t2(increment);
    t1.join();
    t2.join();
    cout << "Number after execution of t1 and t2 is " << number << endl;
    return 0;
}
```

Очікуваний вивід:
Number after execution of t1 and t2 is 2000000

Синхронізація з використанням semaphore

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Синхронізація з використанням semaphore

Семафор — це примітив синхронізації, який обмежує кількість потоків, що можуть одночасно отримати доступ до ресурсу. На відміну від mutex, який дозволяє лише одному потоку увійти в критичну секцію, семафор може пропускати кілька потоків залежно від значення лічильника.

У C++20 додано два типи: std::counting_semaphore (з довільним лічильником) та std::binary_semaphore (лічильник 0 або 1 — аналог mutex). Метод acquire() зменшує лічильник, а release() — збільшує. Якщо лічильник дорівнює нулю, нові потоки чекають.

```
// Семафор дозволяє лише одному потоку працювати зі змінною одночасно
counting_semaphore<1> sem(1);
// Спільна змінна
int number = 0;

void increment() {
    sem.acquire(); // заблокувати доступ
    for (int i = 0; i < 1000000; ++i) {
        number++;
    }
    sem.release(); // розблокувати доступ
}

int main() {
    thread t1(increment);
    thread t2(increment);
    t1.join();
    t2.join();
    cout << "Number after execution of t1 and t2 is " << number << endl;
    return 0;
}
```

Очікуваний вивід:
Number after execution of t1 and t2 is 2000000

binary_semaphore vs mutex

- binary_semaphore працює на основі сигналізації: потоки можуть повідомляти одне одного про готовність/завершення.
- mutex забезпечує взаємне виключення: лише один потік може входити в критичну секцію, і лише він може звільнити ресурс.
- Використовуй mutex, коли треба забезпечити жорстке взаємне виключення.
- Використовуй binary_semaphore, коли треба сигналізувати між потоками або розділяти контроль над ресурсом.

Binary Semaphore	Mutex
Працює на основі сигналів	Працює на основі блокування
Може бути звільнений іншим потоком	Лише потік-власник може викликати unlock()
Не має власника	Має чітку прив'язку до потоку-власника
Кілька потоків можуть чекати/отримати доступ	Тільки один потік у критичній секції одночасно
Швидший у деяких сценаріях	Надійніший при взаємному виключенні
Добре підходить для **сигналізації між потоками**	Добре підходить для **захисту єдиного ресурсу**

Основні методи

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Основні методи std::thread

Метод	Призначення
`join()`	Очікує завершення потоку (блокує виклик).
`detach()`	Відокремлює потік — дозволяє йому виконуватись незалежно.
`joinable()`	Повертає true, якщо потік можна приєднати (тобто ще не join і не detach).
`get_id()`	Повертає унікальний ідентифікатор потоку.
`native_handle()`	Повертає об’єкт із нативним дескриптором ОС (для системно-залежних операцій).
`swap(thread& other)`	Обмінює внутрішні об’єкти двох потоків.
`operator=()`	Присвоює інший std::thread (з переносом володіння потоком).

Основні методи std::mutex

Метод	Опис
`lock()`	Блокує м'ютекс. Якщо вже заблокований — потік чекає, доки він звільниться.
`unlock()`	Розблоковує м'ютекс, дозволяючи іншим потокам доступ до ресурсу.
`try_lock()`	Намагається заблокувати м'ютекс. Якщо вже заблокований — не чекає, а повертає false.
`native_handle()`	Повертає нативний обробник м’ютексу (для системно-залежного коду).

Основні методи std::atomic

Метод	Опис
`store(value)`	Записує значення в атомарну змінну.
`load()`	Зчитує значення атомарної змінної.
`exchange(value)`	Замінює значення і повертає старе.
`compare_exchange_weak(expected, desired)`	Порівнює зі expected, якщо рівні — змінює на desired. Може хибно спрацьовувати.
`compare_exchange_strong(expected, desired)`	Те саме, але без хибних спрацювань — надійніше.
`fetch_add(value)`	Додає value і повертає попереднє значення.
`fetch_sub(value)`	Віднімає value і повертає попереднє значення.
`fetch_or(value)`	Побітове OR із value, повертає попереднє значення.
`fetch_and(value)`	Побітове AND із value, повертає попереднє значення.
`fetch_xor(value)`	Побітове XOR із value, повертає попереднє значення.
`is_lock_free()`	Повертає true, якщо операції над змінною виконуються без блокувань.

Основні методи std::semaphore

Метод	Опис
`acquire()`	Блокує потік, доки семафор не стане більшим за 0, після чого зменшує лічильник.
`try_acquire()`	Намагається негайно захопити ресурс. Повертає true, якщо вдалося.
`try_acquire_for(duration)`	Чекає заданий проміжок часу, щоб захопити ресурс.
`try_acquire_until(time)`	Чекає до певного моменту часу, щоб захопити ресурс.
`release()`	Збільшує лічильник семафора, дозволяючи іншим потокам продовжити виконання.
`max()` *(тільки для counting_semaphore)*	Повертає максимальне можливе значення лічильника.

Приклади

Вплив кількості потоків на ефективність

Розрахувати суму великого вектора цілих чисел, розділивши його на частини й обробляючи паралельно за допомогою різної кількості потоків.

```cpp
// Функція для обчислення суми елементів вектора з використанням кількох потоків
int parallel_sum(const vector<int>& data, int num_threads) {
    vector<thread> threads;                  // Вектор потоків
    vector<int> partial_sums(num_threads, 0); // Проміжні суми для кожного потоку
    int chunk_size = data.size() / num_threads; // Кількість елементів на потік

    // Функція, яку виконує кожен потік — підрахунок суми на своїй частині вектора
    auto sum_range = [&](int index) {
        int start = index * chunk_size;
        int end = start + chunk_size;
        partial_sums[index] = accumulate(data.begin() + start, data.begin() + end, 0);
    };

    // Створюємо потоки, кожен обробляє свою частину вектора
    for (int i = 0; i < num_threads; ++i)
        threads.emplace_back(sum_range, i);

    // Очікуємо завершення всіх потоків
    for (auto& t : threads)
        t.join();

    // Повертаємо загальну суму, яку отримуємо як суму всіх часткових
    return accumulate(partial_sums.begin(), partial_sums.end(), 0);
}
```

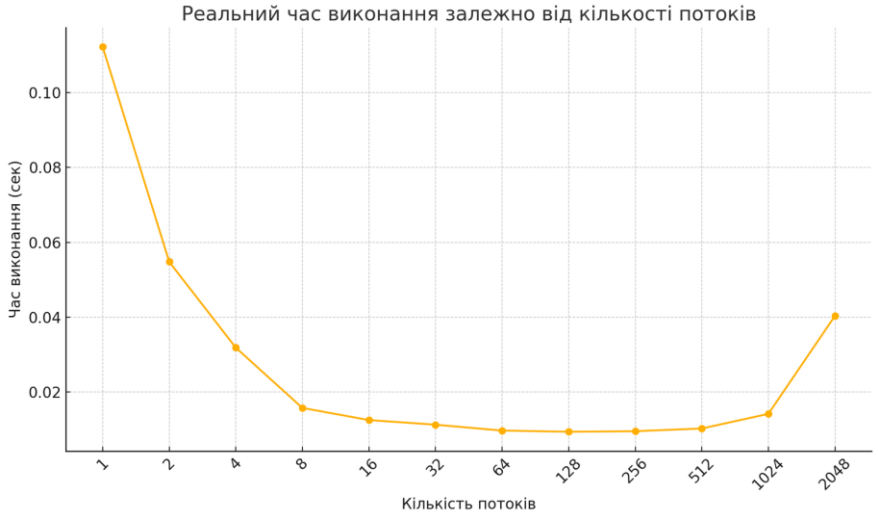
```cpp
int main() {
    // Створюємо великий вектор (2048 * 20000 = 40 960 000 елементів)
    vector<int> data(2048 * 20000, 1); // Очікувана сума: 40 960 000

    // Перевіряємо ефективність різної кількості потоків
    for (int threads : {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048}) {
        auto start = chrono::high_resolution_clock::now();

        long long total = parallel_sum(data, threads); // Паралельне обчислення

        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;

        // Вивід кількості потоків, суми та часу виконання
        cout << threads << " threads: sum = " << total
             << ", time = " << duration.count() << "s\n";
    }

    return 0;
}
```

Вплив кількості потоків на ефективність

Оптимальна кількість потоків: баланс між розпаралелюванням і перевантаженням.

- Розпаралелювання дозволяє значно скоротити час обчислень.
- Однак надмірна кількість потоків призводить до зниження продуктивності через системні накладні витрати.
- В даному прикладі оптимум досягнуто на 128 потоках — подальше збільшення кількості лише шкодить.

```
1 threads: sum = 40960000, time = 0.11219100s
2 threads: sum = 40960000, time = 0.05479420s
4 threads: sum = 40960000, time = 0.03188460s
8 threads: sum = 40960000, time = 0.01580630s
16 threads: sum = 40960000, time = 0.01255940s
32 threads: sum = 40960000, time = 0.01132740s
64 threads: sum = 40960000, time = 0.00975458s
128 threads: sum = 40960000, time = 0.00944637s
256 threads: sum = 40960000, time = 0.00958358s
512 threads: sum = 40960000, time = 0.01030880s
1024 threads: sum = 40960000, time = 0.01421220s
2048 threads: sum = 40960000, time = 0.04035840s
```



Примітка:

- Ці вимірювання залежать від обчислювальних ресурсів комп'ютера (процесора, кешу, ядер).
- На інших комп'ютерах результати можуть бути зовсім іншими: оптимальна кількість потоків може бути меншою або більшою.

Приклад з Deadlock

```
mutex mutex1;
mutex mutex2;

void threadA() {
    mutex1.lock();  // Захоплює перший м'ютекс
    this_thread::sleep_for(chrono::milliseconds(100)); // Імітація роботи
    mutex2.lock();  // Чекає на другий м'ютекс (може заблокуватись)

    cout << "Thread A finished\n";

    mutex2.unlock();
    mutex1.unlock();
}

void threadB() {
    mutex2.lock();  // Захоплює другий м'ютекс
    this_thread::sleep_for(chrono::milliseconds(100)); // Імітація роботи
    mutex1.lock();  // Чекає на перший м'ютекс (вже зайнятий threadA)

    cout << "Thread B finished\n";

    mutex1.unlock();
    mutex2.unlock();
}
```

```
int main() {
    thread t1(threadA);
    thread t2(threadB);

    t1.join();
    t2.join();

    return 0;
}
```

Що тут не так:

- Потік A захоплює mutex1, потім чекає mutex2.
- Потік B захоплює mutex2, потім чекає mutex1.
- Жоден не відпускає свій м’ютекс, бо чекає інший — програма зависає.

Дякую

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY