Лекція 3

БАГАТОПОТОКОВЕ ТА АСИНХРОННЕ ПРОГРАМУВАННЯ

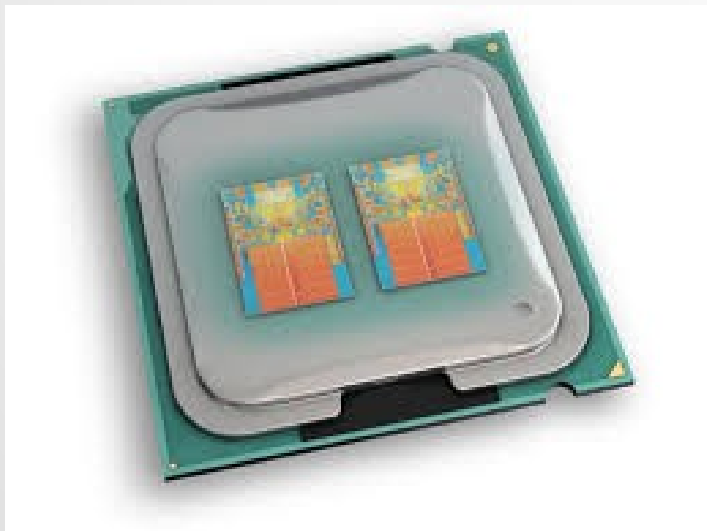
План

1. Загальне поняття багатопотоковості та паралелізму.
2. Проблеми паралелізму.
3. Асинхронні виклики.
4. Клас std::thread.
5. Клас std::mutex.
6. Клас std::atomic.

1. Загальне поняття багатопотоковості і паралелізму

Під **паралелізмом** розуміється одночасне виконання декількох операцій. В інформатиці це означає, що обчислювальна система виконує декілька незалежних операцій паралельно (одночасно), а не послідовно.

Поява і широке поширення комп’ютерів, обладнаних декількома процесорами або декількома ядрами на одному кристалі (**багатоядерними процесорами**), привело до необхідності розробки програм, що підтримують паралелізм.



Виробники апаратного забезпечення намагаються не нарощувати тактову частоту процесора (тобто його швидкодію), а збільшувати кількість його ядер. Таким чином, для підвищення ефективності застосування комп’ютеру необхідно створювати багатопотокові програми.

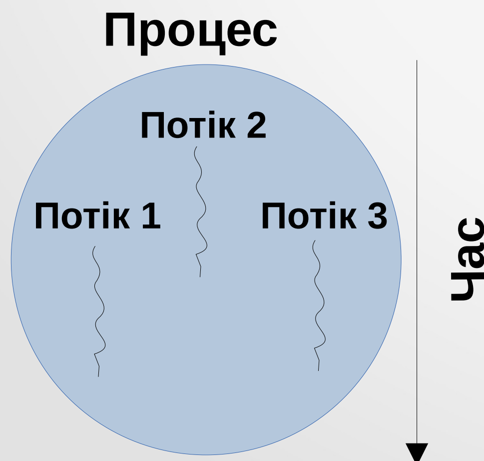
ВАЖЛИВО! Сучасні процесори навіть при наявності одного ядра можуть одночасно виконувати декілька команд. Це називається апаратним паралелізмом.

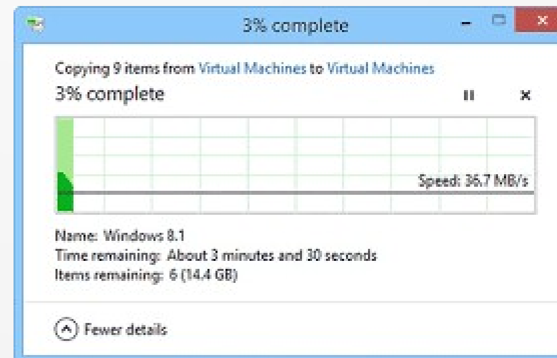
1. Загальне поняття багатопотоковості і паралелізму

Любий **обчислювальний процес** в сучасних операційних системах (Windows, Linux, ...) складається мінімум з одного потоку.

Потік (**thread**) – це одна з дій всередині процесу, яка є найменшою одиницею обробки, виконання якої може бути призначено **планувальником** ОС. Інакше кажучи, потік – це окремий шлях виконання програмного коду всередині програми, що виконується (процесу).

Процес може бути як **одно-**, так и **багатопотоковим**.



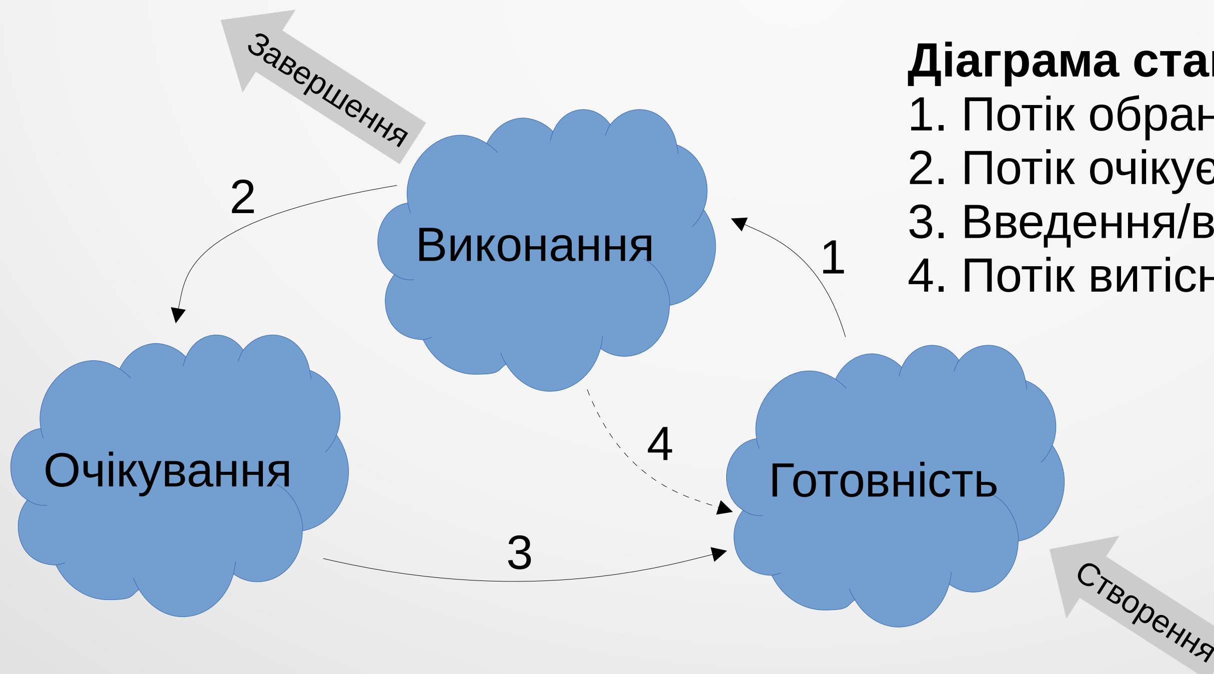


Приклад багатопотокового процесу: один потік копіює, інший – виконує анімацію

1. Загальне поняття багатопотоковості і паралелізму

Створення багатопотокових програм дає можливість досягти **реального паралелізму** в роботі комп’ютера, обладнаного декількома процесорами (або багатоядерним процесором).

Навіть на **однопроцесорному** комп’ютері використання багатопотокових програм дозволяє підвищити загальну **реактивність** системи за рахунок можливості обходу **блокувань** (процес, що очікує завершення операцій вводу/виводу, блокується системою, але використання декількох потоків в ньому дозволяє реагувати на команди користувача в інших потоках).



Діаграма стану потоку:
1. Потік обраний на виконання.
2. Потік очікує вводу/виводу.
3. Введення/виведення завершено.
4. Потік витісненний планувальником.

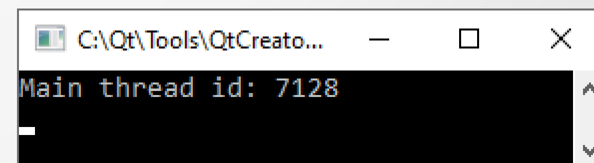
1. Загальне поняття багатопотоковості і паралелізму

В мові програмування C++ **головний потік** автоматично створюється при запуску функції main(). Він в любий час може створювати необхідну кількість **вторинних потоків** для виконання додаткової роботи.

В C++, починаючи зі стандарту C++11, реалізовано простір імен **std::this_thread**, який описано в заголовному файлі thread. В ньому реалізовано чотири методи:

- **std::this_thread::get_id()** – отримати ідентифікатор поточного потоку;
- **std::this_thread::yield()** – примусова передача управління планувальнику;
- **std::this_thread::sleep_for()** – блокування виконання потоку протягом певного періоду часу;
- **std::this_thread::sleep_until()** – ... до певного моменту.

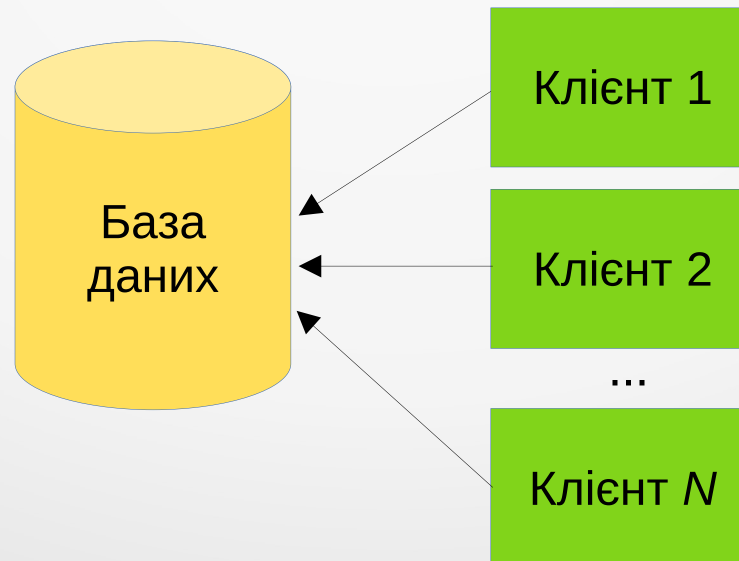
```
#include <iostream>
#include <thread>

int main()
{
    std::cout << "Main thread id: " << std::this_thread::get_id() << std::endl;
    return 0;
}
```



2. Проблеми паралелізму

Конкурентністю називається можливість виконання декількох потоків в перехресні періоди часу, що може призводити до **гонитви** – проблеми синхронізації їх доступу до деяких загальних ресурсів, наприклад даних (“гонитва за даними”).

Наприклад, якщо потік А ще не завершився, а потік В вже намагається оперувати його даними (які ще можливо не готові), то в цьому випадку поведінка потоків (і всього процесу) скоріше за все стане непередбачуваною і може призводити до дуже нетривіальних помилок.



2. Проблеми паралелізму

Розглянемо наступний приклад. Нехай в потоці виконується оператор інкременту мови C++:

x++; // збільшення значення цілочисельний змінної x на 1

Байт-код, що реалізує цей оператор, можна описати, наприклад, так:

```
load x into register
add 1 to register
store register in x
```

При виконанні цих операцій може виникнути гонитва за даними. Нехай, наприклад, x = 5. Тоді:

Шаг	Поток 1	Поток 2	x	register
1	load x into register		5	5
2	add 1 to register		5	6
3	store register in x		6	6
4		load x into register	6	6
5		add 1 to register	6	7
6		store register in x	7	7

2. Проблеми паралелізму

Так також можливо:

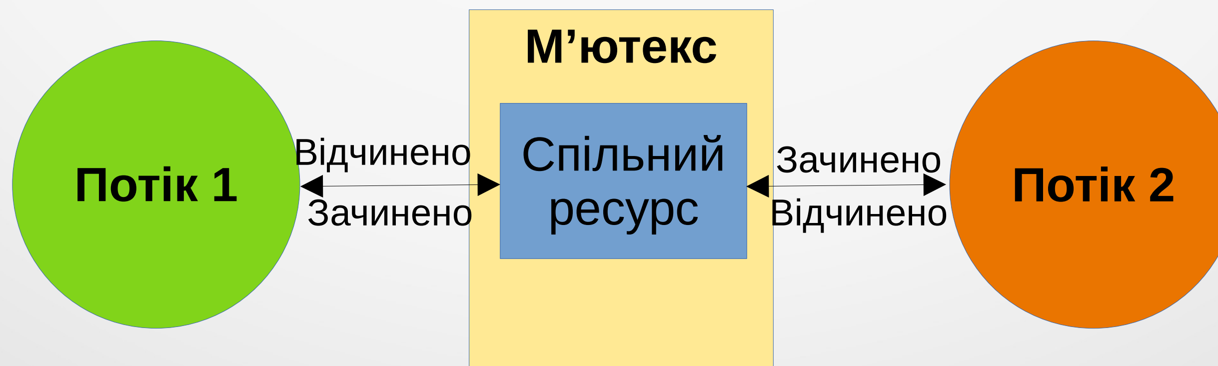
Шаг	Поток 1	Поток 2	x	register
1		load x into register	5	5
2		add 1 to register	5	6
3		store register in x	6	6
4	load x into register		6	6
5	add 1 to register		6	7
6	store register in x		7	7

А так вже ні:

Шаг	Поток 1	Поток 2	x	register
1	load x into register		5	5
2	add 1 to register		5	6
3		load x into register	5	5
4	store register in x		5	5
5		add 1 to register	5	6
6	store register in x		6	6

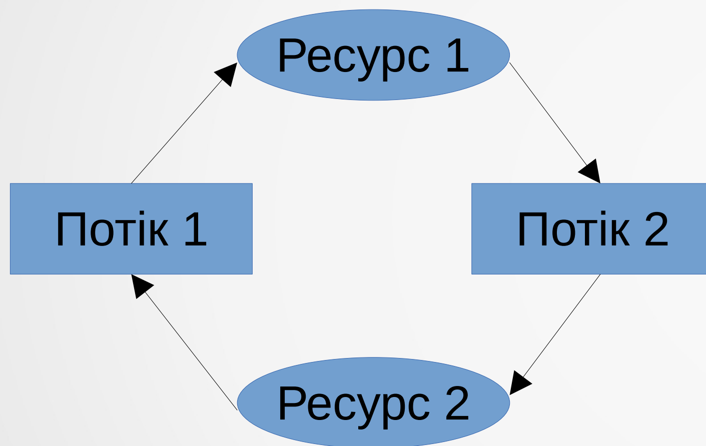
2. Проблеми паралелізму

Для усунення цієї проблеми (**синхронізації**) потоків використовують так звані **м'ютекси**, під якими розуміються спеціальні об'єкти (**семафори**), що мають можливість приймати два значення (наприклад, **відчинено/зачинено**). Якщо потік звертається до м'ютексу і він має значення "відчинено", то він встановлює його в значення "зачинено" і може монопольно використовувати **критичну область коду**. Всі інші потоки в цей момент блокуються. Таким чином, в один момент часу тільки один потік може володіти м'ютексом.
Після завершення своєї роботи з критичною областю, потік, що заблокував м'ютекс, встановлює його в значення "відчинено" і інші потоки можуть отримати до нього (а, відповідно, і до критичної області) доступ.



2. Проблеми паралелізму

На жаль, використання блокувань не завжди вирішує проблему синхронізації потоків, оскільки на практиці при реалізації багатопотокових програм можуть виникати так звані **взаємні блокування** (**клінчі**).



Приклад взаємного блокування двох потоків

Взаємне блокування – це ситуація, коли два потоки очікують закінчення роботи один одного і, таким чином, жоден з них не може завершити свою роботу. При наявності м'ютексів взаємне блокування відбувається, наприклад, тоді, коли потоку А потрібен м'ютекс, яким володіє потік В, і навпаки.

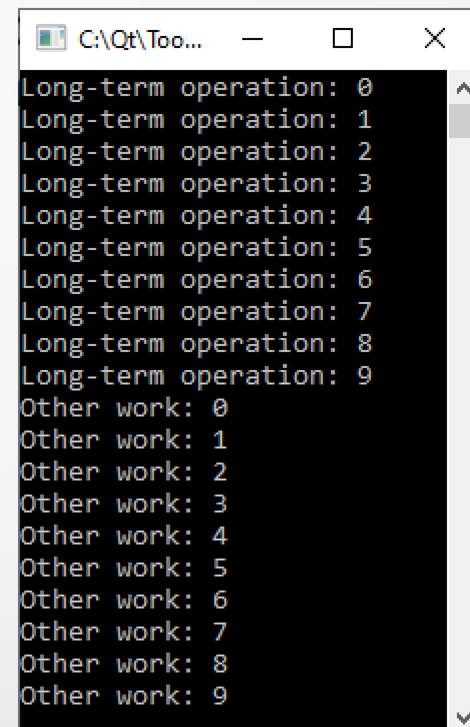
ВАЖЛИВО! Для профілактики клінчів потрібно правильним чином проектувати послідовність роботи потоків додатки.

3. Асинхронні виклики

Якщо однопотоковий процес виконує якусь тривалу операцію, то поки вона не закінчиться, програма не зможе виконувати ніяких інших задач (оновлювати свій графічний інтерфейс, реагувати на команди користувача і таке інше).

```
#include <iostream>
#include <thread>

void do_long_work(void)
{
    for (int i = 0; i < 10; i++)
        std::cout << "Long-term operation: " << i << '\n';
}

int main()
{
    // Виклик тривалої процедури
    do_long_work();
    // Виконання іншої роботи в головному потоці
    for (int i = 0; i < 10; i++)
        std::cout << "Other work: " << i << '\n';
    return 0;
}
```

Доки не закінчиться тривала операція, програма не може виконувати іншу роботу



3. Асинхронні виклики

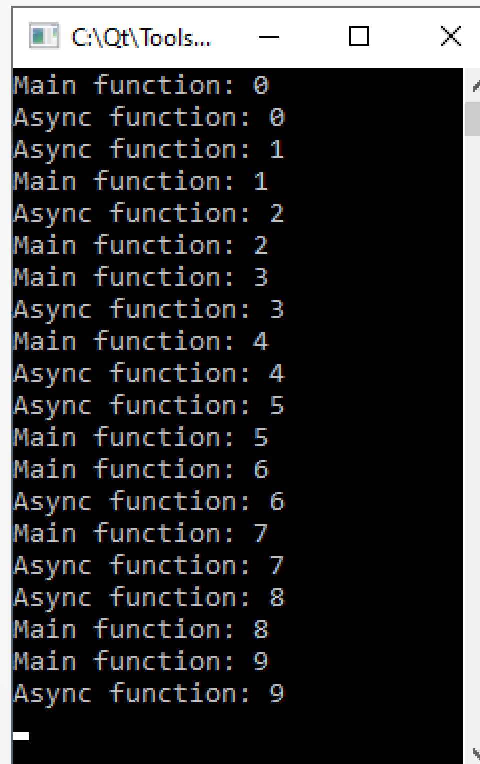
Для виправлення цієї ситуації використовується механізм **асинхронності** – можливості незалежного (асинхронного) виконання різних частин коду програми.

Для цього в бібліотеці STL реалізовано спеціальну шаблонну функцію **std::async()**, яка оголошена в заголовному файлі future.

```
#include <iostream>
#include <future>

using namespace std;

// Асинхронна функція
void async_func(void)
{
    for (auto i = 0; i < 10; i++)
    {
        cout << "Async function: " << i << '\n';
        this_thread::sleep_for(0.1s);
    }
}

int main()
{
    // Асинхронний запуск функції
    auto res = async(async_func);

    for (auto i = 0; i < 10; i++)
    {
        cout << "Main function: " << i << '\n';
        this_thread::sleep_for(0.1s);
    }
    return 0;
}
```

3. Асинхронні виклики

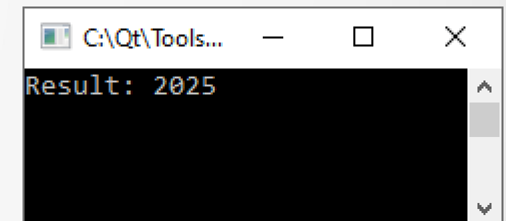
Результат роботи асинхронної програми



3. Асинхронні виклики

Функція std::async() повертає об’єкт класу **std::future**, що призначений для збереження майбутнього результату (який буде повернено асинхронною функцією після її завершення). Найбільш часто вживаним методом класу std::future є **get()**, який повертає результат роботи асинхронної функції, коли він буде готовий.

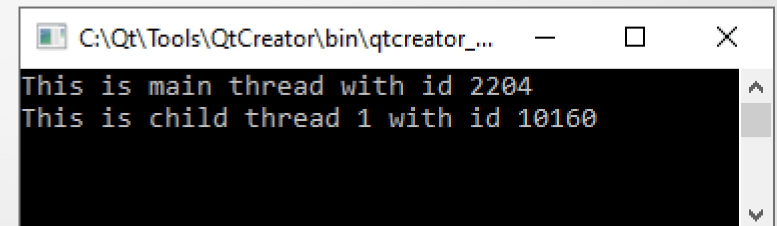
```
#include <iostream>
#include <future>

auto async_func(void)
{
    auto sum = 0;

    for (auto i = 0; i < 10; i++)
        sum += i;
    return sum;
}

int main()
{
    auto res1 = async(async_func);
    auto res2 = async(async_func);

    cout << "Result: " << res1.get() * res2.get() << endl;
    return 0;
}
```



4. Клас std::thread

Для створення нового потоку в мові програмування C++, починаючи зі стандарту C++11, використовується клас std::thread, який описано у заголовному файлі thread.

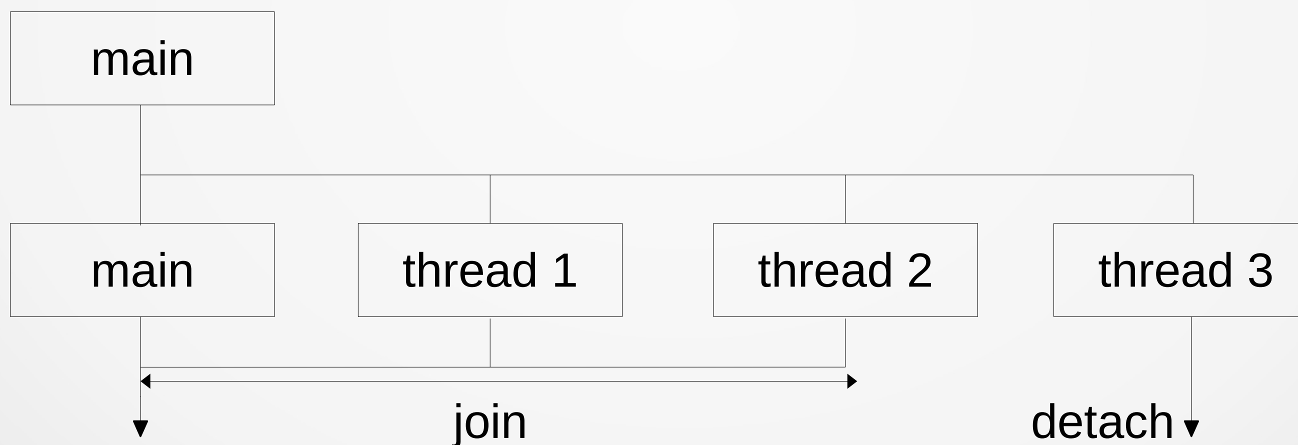
```
#include <iostream>
#include <thread>

// Потокова функція
void thread_func(int no)
{
    std::cout << "This is child thread " << no << " with id " <<
              std::this_thread::get_id() << '\n';
}

int main()
{
    // Створення і запуск дочірнього потоку з параметром "1"
    std::thread thr(thread_func, 1);

    std::cout << "This is main thread with id " <<
              std::this_thread::get_id() << '\n';
    thr.join();
    return 0;
}
```



4. Клас std::thread

Для синхронізації дочірніх потоків з батьківським (головним) в класі std::thread використовуються два методи **join()** і **detach()**.

Метод **join()** викликає блокування батьківського потоку до завершення дочірнього. Метод **detach()** викликає від'єднання дочірнього потоку від батьківського.



ВАЖЛИВО! Метод detach() застосовується у тому випадку, коли головний потік не буде очікувати завершення роботи дочірнього. Після від'єднання потоку отримати результат його роботи вже неможливо.

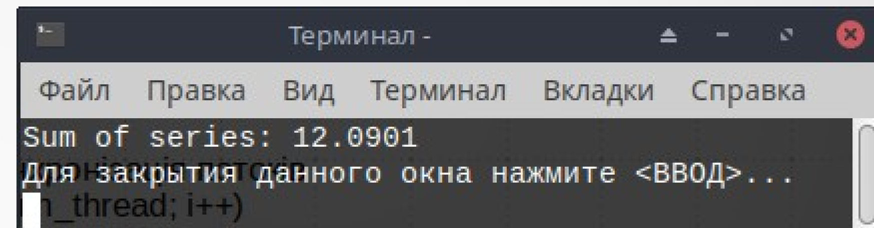
4. Клас std::thread

Приклад багатопотокового розрахунку часткової суми гармонічного ряду.

```
#include <iostream>
#include <thread>
#include <vector>
#include <algorithm>

// Потокова функція
void sum_series(int begin, int end, double &sum)
{
    double local_sum{0};

    for (auto i = begin; i < end; i++)
        local_sum += 1.0 / double(i + 1);
    sum += local_sum; // Можлива гонитва за даними
}
```

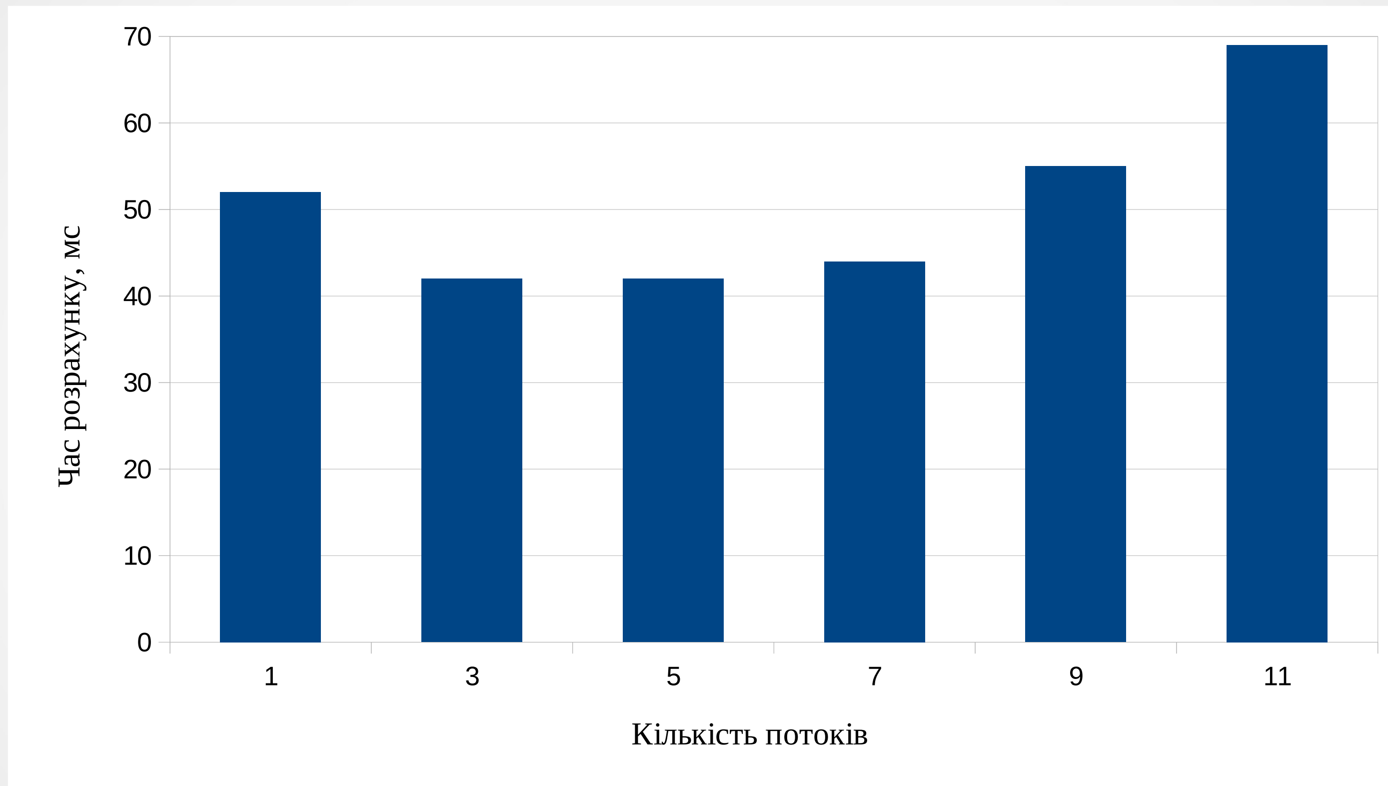
4. Клас std::thread

Приклад багатопотокового розрахунку часткової суми гармонічного ряду.




```
int main()
{
    auto n{100000}, // Кількість ітерацій
         num_thread{8}, // Кількість потоків
         step = n / num_thread;
    auto sum{0.};
    std::vector<std::thread> thr(num_thread);

    // Створення та синхронізація потоків
    for (auto i = 0; i < num_thread; i++)
        thr[i] = std::thread(sum_series, i * step,
                    (i == num_thread - 1) ? n : (i + 1) * step, std::ref(sum));
    for_each(thr.begin(), thr.end(), [](auto &it) { it.join(); });
    // Виведення результату
    std::cout << "Sum of series: " << sum << '\n';
    return 0;
}
```

4. Клас std::thread



Залежність часу обчислення суми ряду від кількості використаних потоків (на процесорі з 16 потоками)

5. Клас std::mutex

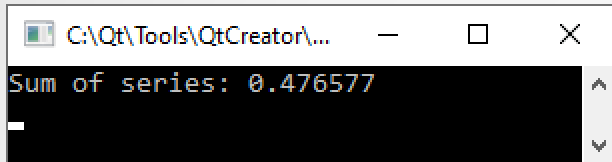
Розглянемо наступний приклад. Змінимо код потокової функції sum_series() попереднього прикладу наступним чином:

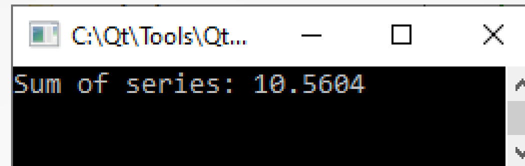
```
void sum_series(int begin, int end, double &sum)
{
    double local_sum{0}; // Локальна змінна потоку

    for (auto i = begin; i < end; i++)
        local_sum += 1.0 / double(i + 1);
    sum += local_sum; // Критична область
}
```

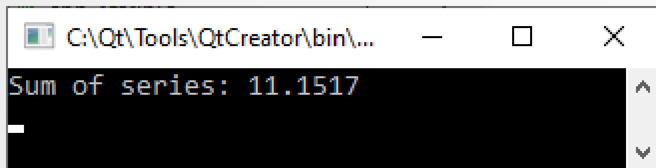
```
void sum_series(int begin, int end, double &sum)
{
    for (auto i = begin; i < end; i++)
        sum += 1.0 / double(i + 1); // Критична область
}
```

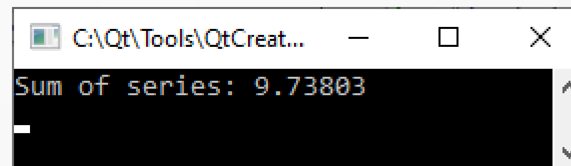
5. Клас std::mutex

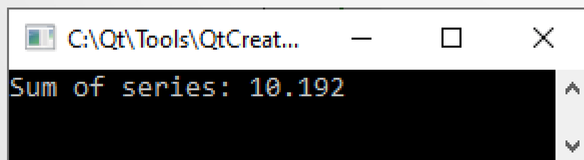
Серія запусків програми призводить до різних результатів:

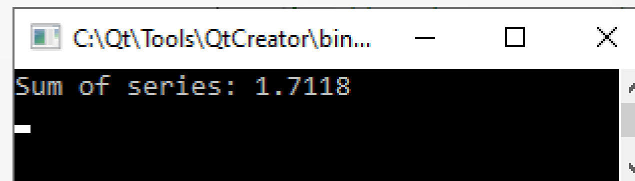












5. Клас std::mutex

До такого розкиду результатів очевидно призводить гонитва за даними – змінною **sum**, яка містить підсумкову суму ряду.

Для усунення цієї проблеми скористаємося м’ютексом. В бібліотеці STL для інкапсуляції поняття м’ютексу реалізовано клас **std::mutex**, який описано в заголовному файлі mutex.

Базовий конструктор цього класу не вимагає параметрів, тому створити м’ютекс можна, наприклад, таким чином.

```
#include <mutex>

// ...
std::mutex my_mutex;
// ...
```

5. Клас std::mutex

Клас std::mutex містить декілька методів, найбільш часто вживаними серед яких є **lock()** і **unlock()**.

```
// ...
my_mutex.lock(); // Захоплення м’ютексу

// Критична область
// ...
my_mutex. unlock(); // Звільнення м’ютексу
```

ВАЖЛИВО! Повторне захоплення в потоці м’ютексу може привести до зависання програми.

5. Клас std::mutex

Для запобігання гонитві за даними функцію sum_series() можна переписати із застосуванням м’ютексу, наприклад, наступним чином.

```
// ...
#include <mutex>

std::mutex my_mutex;

// Потокова функція
void sum_series(int begin, int end, double &sum)
{
    for (auto i = begin; i < end; i++)
    {
        my_mutex.lock();
        sum += 1.0 / double(i + 1);
        my_mutex.unlock();
    }
}

// ...
```

5. Клас std::mutex

Така реалізація функції sum_series() не є оптимальною. Наступна реалізація буде більш швидкою.

```
// ...
void sum_series(int begin, int end, double &sum)
{
    auto local_sum{0.};

    for (auto i = begin; i < end; i++)
        local_sum += 1.0 / double(i + 1);
    my_mutex.lock();
    sum += local_sum;
    my_mutex.unlock();
}
// ...
```

6. Клас std::atomic

Ще одним засобом синхронізації доступу потоків до спільних даних є застосування узагальненого класу **std::atomic**, який описано в заголовному файлі atomic.

Об’єкти цього типу є неділимими (атомарними) з точки зору виконання потокових операцій над ними. Тобто ні один потік не може побачити проміжний стан змінної такого типу, що гарантує коректність роботи з ними в багатопотокових процесах.

На жаль, клас std::atomic можна застосовувати не завжди, а лише для обмеженої кількості стандартних типів, таких, як **bool**, **char**, **int** тощо. Крім того не всі операції в цьому класі перезавантажено.

Попередній приклад підрахунку часткової суми ряду можна переписати таким чином.

6. Клас std::atomic

```
#include <iostream>
#include <thread>
#include <vector>
#include <algorithm>
#include <atomic>

void sum_series(int begin, int end, std::atomic<double> &sum)
{
    for (auto i = begin; i < end; i++)
        sum = sum + 1.0 / double(i + 1); // sum += ... дає помилку
}

int main()
{
    auto n{100000}, // Кількість ітерацій
            num_thread{8}, // Кількість потоків
            step = n / num_thread;
    std::atomic<double> sum{0.};
    std::vector<std::thread> thr(num_thread);
```

6. Клас std::atomic

```
    // Створення та синхронізація потоків
    for (auto i = 0; i < num_thread; i++)
        thr[i] = std::thread(sum_series, i * step,
            (i == num_thread - 1) ? n : (i + 1) * step, std::ref(sum));
    for_each(thr.begin(), thr.end(), [](auto &it) { it.join(); });
    // Виведення результату
    std::cout << "Sum of series: " << sum << '\n';
    return 0;
}
```