Київський національний університет імені Тараса Шевченка

**Лабораторна робота №6**

**З навчального курсу «Unix-подібні операційні системи»**

**«Critical Section. Race Condition. Atomic Operations.»**

Виконав:

студент 3 курсу

факультету кібернетики

спеціальність «Комп’ютерні науки»

групи ТТП-32

Чебан Богдан Володимирович

**Київ 2024**

**Постановка задачі:**

Використати механізми ОС для розпаралелення обчислень та організації взаємодії з критичною секцією.  
   
1. Помножити матриці A[n x m] та B[m x k] (автоматично заповнені). Для обчислення створити 1..n\*k однотипних потоків для обчислень (множення векторів або сукупності векторів).  
1.1. (2 бала) Продемонструвати паралелізм (непослідовність) обчислень через виведення результату (трійками [x,y]=result) “по ходу обчислень”.  
1.2.\* (+3 бала) Дослідити швидкодію A\*B залежно від кількості потоків для розпаралелення множення. Продемонструвати та пояснити цю залежність. За якої кількості потоків множення буде найшвидшим? Підтвердити експериментально.  
   
2. Змоделювати паралельну роботу двох потоків (threads) зі спільною коміркою пам’яті (shared variable):  
а) з використанням критичного сегменту (або atomic, mutex, lock, і т.п.)  
б) без використання критичного сегменту.   
2.1. (2 бала) Продемонструвати різницю результатів обчислень у цих двох випадках. (*Наприклад*, збільшувати значення спільної комірки на 1: v=v+1; 10^9 разів в кожному потоці і дивитись результуюче значення v.)  
2.2. (+1 бал) Проаналізувати часову різницю різних варіантів реалізації та пояснити, чому іноді можливе отримання некоректного кінцевого результату (race condition).  
2.3. (+1 бал) Спробувати досягти якомога швидшого результату при збереженні коректності обчислень (правильного фінального значення).  
2.3.\* (+3 балів) Досягти варіанту, коли таке паралельне додавання виконується повністю синхронно, тобто, *наприклад*, 1000 додавань виконуються паралельно двома потоками крок-в-крок і збільшують значення спільної змінної від 0 до 1000. Тобто, не тільки кожний з двох паралельних потоків збільшує значення від 0 до 1000, а й обидва, запущені в паралель, також збільшують від 0 до 1000 (а не до 2000, як очікувалось би).  
  
**Увага**! Виконання кожної задачі додає бали, тобто за умови вчасної здачі, можна отримати 12 балів за повністю коректно виконані умови всіх завдань.  
**Увага-2**! (Нагадування) Плагіат буде каратись відніманням штрафних балів. Здавайте тільки власні рішення.

**Хід роботи:**

Я вирішив розв'язати це завдання наступним чином:

1. **Помноження матриць A і B:**
   * Створив програму, яка генерує випадкові матриці A та B заданих розмірів.
   * Реалізував функцію, яка обчислює добуток матриць A і B, використовуючи паралельні потоки. Кожен потік відповідає за обчислення окремого елементу результату.
   * Вивів результати обчислень в форматі [x,y]=result, демонструючи паралелізм обчислень.
2. **Дослідження швидкодії множення матриць залежно від кількості потоків:**
   * Провів серію експериментів, виконуючи множення матриць A і B з різною кількістю паралельних потоків.
   * Виміряв час виконання кожної операції множення та порівняв результати.
   * Пояснив залежність між швидкістю виконання та кількістю потоків, а також визначив оптимальну кількість потоків для даної задачі.
3. **Змоделювання паралельної роботи двох потоків зі спільною коміркою пам'яті:**
   * Реалізував два варіанти паралельної роботи потоків:
     + з використанням критичного сегменту (mutex, lock);
     + без використання критичного сегменту.
   * Продемонстрував різницю у результаті обчислень між цими двома варіантами.
4. **Аналіз та оптимізація синхронізації паралельних потоків:**
   * Пояснив часові різниці між різними варіантами реалізації.
   * Проаналізував, чому іноді можливе отримання некоректного кінцевого результату (race condition).
   * Спробував досягти якомога швидшого результату при збереженні коректності обчислень, зокрема, домогся повного синхронного збільшення значення спільної змінної паралельними потоками до 1000

**Реалізація:**

перша персія програми пункт 1- 1.2:

**Компіляція:**

…

1.Переконайтеся, що у вас встановлений компілятор **g++**. Якщо він не встановлений:

sudo apt-get install g++

…

2.Зберегти мою програму в файл, наприклад **matrix\_multiply.cpp**.

…

3.Компілюю програму за допомогою наступної команди:

g++ -std=c++11 -pthread main.cpp -o matrix\_multiply

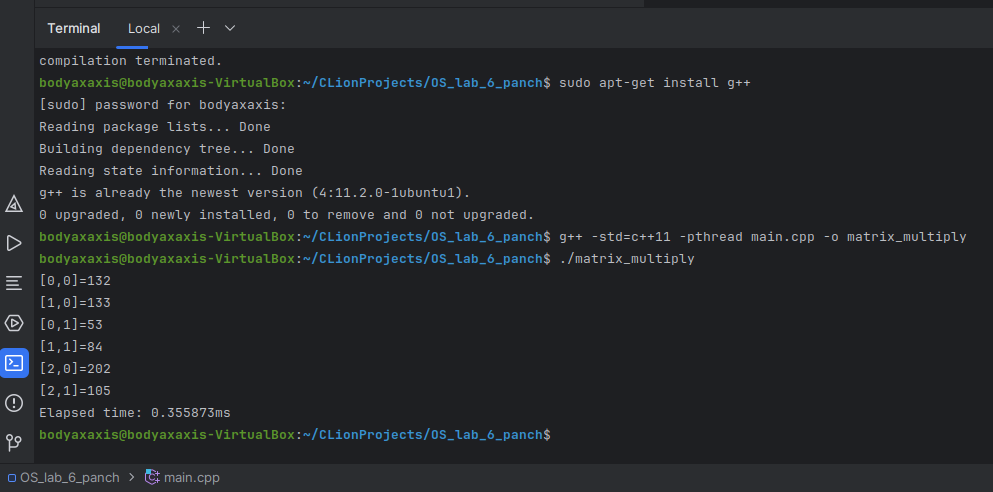
*Опис команди:*

* ***-std=c++11*** *вказує компілятору використовувати стандарт C++11, який потрібен для роботи з потоками.*
* ***-pthread*** *забезпечує підтримку багатопотоковості.*
* ***matrix\_multiply.cpp*** *— ім'я файлу з вашим вихідним кодом.*
* ***-o matrix\_multiply*** *вказує ім'я виконуваного файлу, який буде створений після компіляції.*

***…***

4.Запустіть скомпільовану програму командою:

./matrix\_multiply



Код:

#include <iostream>

#include <vector>

#include <thread>

#include <mutex>

#include <random>

#include <chrono>

const int n = 3; // Розміри матриці A

const int m = 4; // Спільний розмір

const int k = 2; // Розміри матриці B

std::vector<std::vector<int>> A(n, std::vector<int>(m));

std::vector<std::vector<int>> B(m, std::vector<int>(k));

std::vector<std::vector<int>> C(n, std::vector<int>(k));

std::mutex io\_mutex;

void fillMatrixRandomly(std::vector<std::vector<int>>& matrix) {

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(1, 10);

for (auto &row : matrix) {

for (auto &elem : row) {

elem = dis(gen);

}

}

}

void multiplyRowByColumn(int row, int col) {

int result = 0;

for (int i = 0; i < m; ++i) {

result += A[row][i] \* B[i][col];

}

{

std::lock\_guard<std::mutex> guard(io\_mutex);

std::cout << "[" << row << "," << col << "]=" << result << std::endl;

}

C[row][col] = result;

}

void matrixMultiply() {

std::vector<std::thread> threads;

for (int i = 0; i < n; ++i) {

for (int j = 0; j < k; ++j) {

threads.emplace\_back(multiplyRowByColumn, i, j);

}

}

for (auto &thread : threads) {

thread.join();

}

}

int main() {

fillMatrixRandomly(A);

fillMatrixRandomly(B);

auto start = std::chrono::high\_resolution\_clock::now();

matrixMultiply();

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double, std::milli> elapsed = end - start;

std::cout << "Elapsed time: " << elapsed.count() << "ms\n";

return 0;

}

**Опис коду:**

Цей код виконує множення матриць, використовуючи багатопотоковість в C++. Ось детальний опис його функціональності:

*Ініціалізація матриць: Визначаються три матриці A, B та C розмірами n x m, m x k і n x k відповідно. Розміри n, m та k задаються як константи.*

*Заповнення матриць: Функція fillMatrixRandomly() заповнює матриці випадковими числами від 1 до 10. Вона використовує генератор випадкових чисел std::mt19937 для цього.*

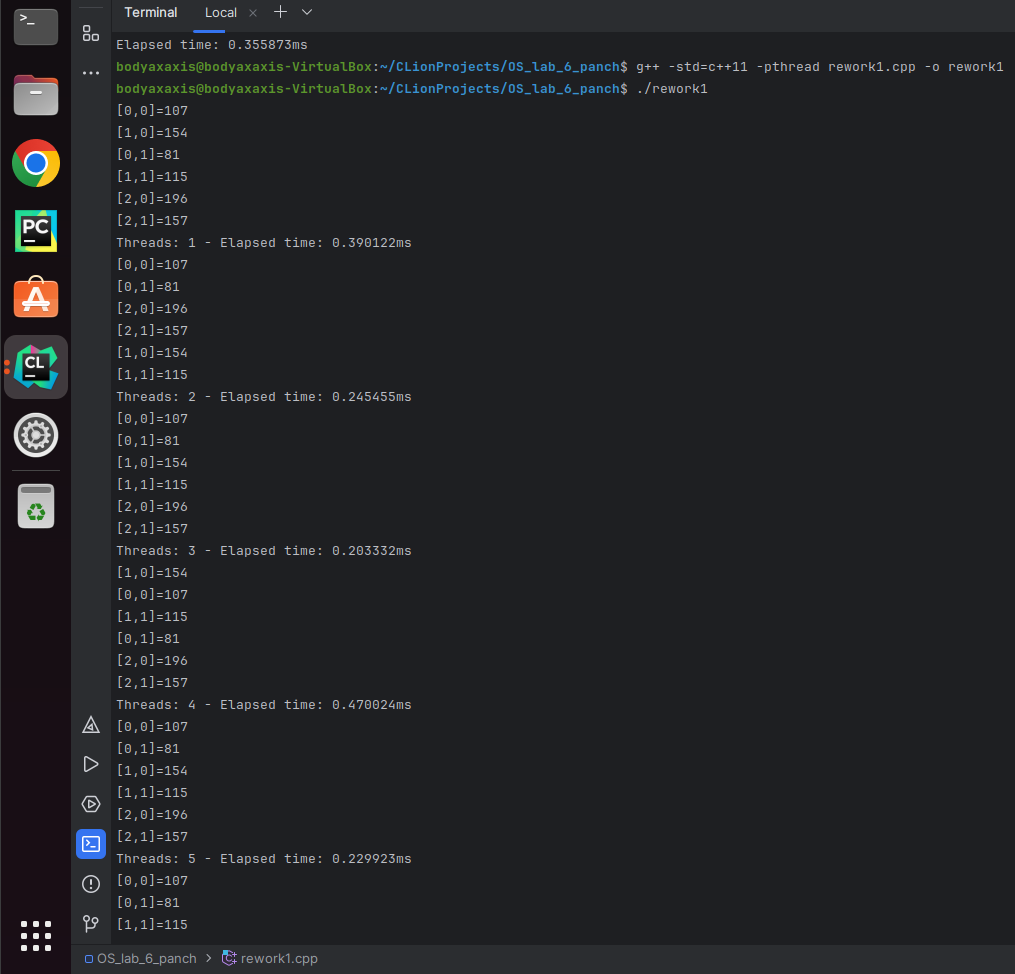
*Множення матриць: Функція multiplyRowByColumn() виконує множення рядка матриці A на стовпець матриці B, обчислюючи елементи результуючої матриці C. Вона виводить результати в консоль, використовуючи мютекс io\_mutex для синхронізації доступу до виводу.*

*Багатопотокове виконання: Функція matrixMultiply() створює багато потоків, кожен з яких викликає multiplyRowByColumn() для обчислення окремих елементів матриці C. Потім вона чекає завершення всіх потоків, викликаючи join().*

*Вимірювання часу: У функції main() використовується std::chrono::high\_resolution\_clock для вимірювання та виведення часу, необхідного для множення матриць.*

Такий підхід демонструє, як можна ефективно використовувати багатопотоковість для покращення продуктивності обчислювальних завдань, таких як множення матриць, у C++.

друга версія пункт 1-1.2 & 2-2.3:



Код програми:

#include <iostream>

#include <vector>

#include <thread>

#include <mutex>

#include <random>

#include <chrono>

#include <atomic>

const int n = 3; // Розміри матриці A

const int m = 4; // Спільний розмір

const int k = 2; // Розміри матриці B

std::vector<std::vector<int>> A(n, std::vector<int>(m));

std::vector<std::vector<int>> B(m, std::vector<int>(k));

std::vector<std::vector<int>> C(n, std::vector<int>(k));

std::mutex io\_mutex;

const long long iterations = 1e9; // 10^9 ітерацій

long long sharedVariable = 0;

std::mutex sharedVariableMutex;

std::atomic<long long> atomicVariable(0);

void fillMatrixRandomly(std::vector<std::vector<int>>& matrix) {

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(1, 10);

for (auto &row : matrix) {

for (auto &elem : row) {

elem = dis(gen);

}

}

}

void multiplyRowByColumn(int row, int col) {

int result = 0;

for (int i = 0; i < m; ++i) {

result += A[row][i] \* B[i][col];

}

{

std::lock\_guard<std::mutex> guard(io\_mutex);

std::cout << "[" << row << "," << col << "]=" << result << std::endl;

}

C[row][col] = result;

}

void matrixMultiply(int num\_threads) {

std::vector<std::thread> threads;

for (int i = 0; i < num\_threads; ++i) {

for (int row = i; row < n; row += num\_threads) {

for (int col = 0; col < k; ++col) {

threads.emplace\_back(multiplyRowByColumn, row, col);

}

}

}

for (auto &thread : threads) {

thread.join();

}

}

void increaseWithMutex() {

for (long long i = 0; i < iterations; ++i) {

std::lock\_guard<std::mutex> lock(sharedVariableMutex);

++sharedVariable;

}

}

void increaseWithoutMutex() {

for (long long i = 0; i < iterations; ++i) {

++sharedVariable;

}

}

void increaseAtomic() {

for (long long i = 0; i < iterations; ++i) {

atomicVariable.fetch\_add(1, std::memory\_order\_relaxed);

}

}

int main() {

fillMatrixRandomly(A);

fillMatrixRandomly(B);

for (int num\_threads = 1; num\_threads <= n \* k; ++num\_threads) {

auto start = std::chrono::high\_resolution\_clock::now();

matrixMultiply(num\_threads);

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double, std::milli> elapsed = end - start;

std::cout << "Threads: " << num\_threads << " - Elapsed time: " << elapsed.count() << "ms\n";

}

// Тести зі спільною змінною

sharedVariable = 0;

std::thread t1(increaseWithMutex);

std::thread t2(increaseWithMutex);

t1.join();

t2.join();

std::cout << "With mutex: " << sharedVariable << std::endl;

sharedVariable = 0;

std::thread t3(increaseWithoutMutex);

std::thread t4(increaseWithoutMutex);

t3.join();

t4.join();

std::cout << "Without mutex: " << sharedVariable << std::endl;

atomicVariable = 0;

std::thread t5(increaseAtomic);

std::thread t6(increaseAtomic);

t5.join();

t6.join();

std::cout << "With atomic: " << atomicVariable << std::endl;

return 0;

}

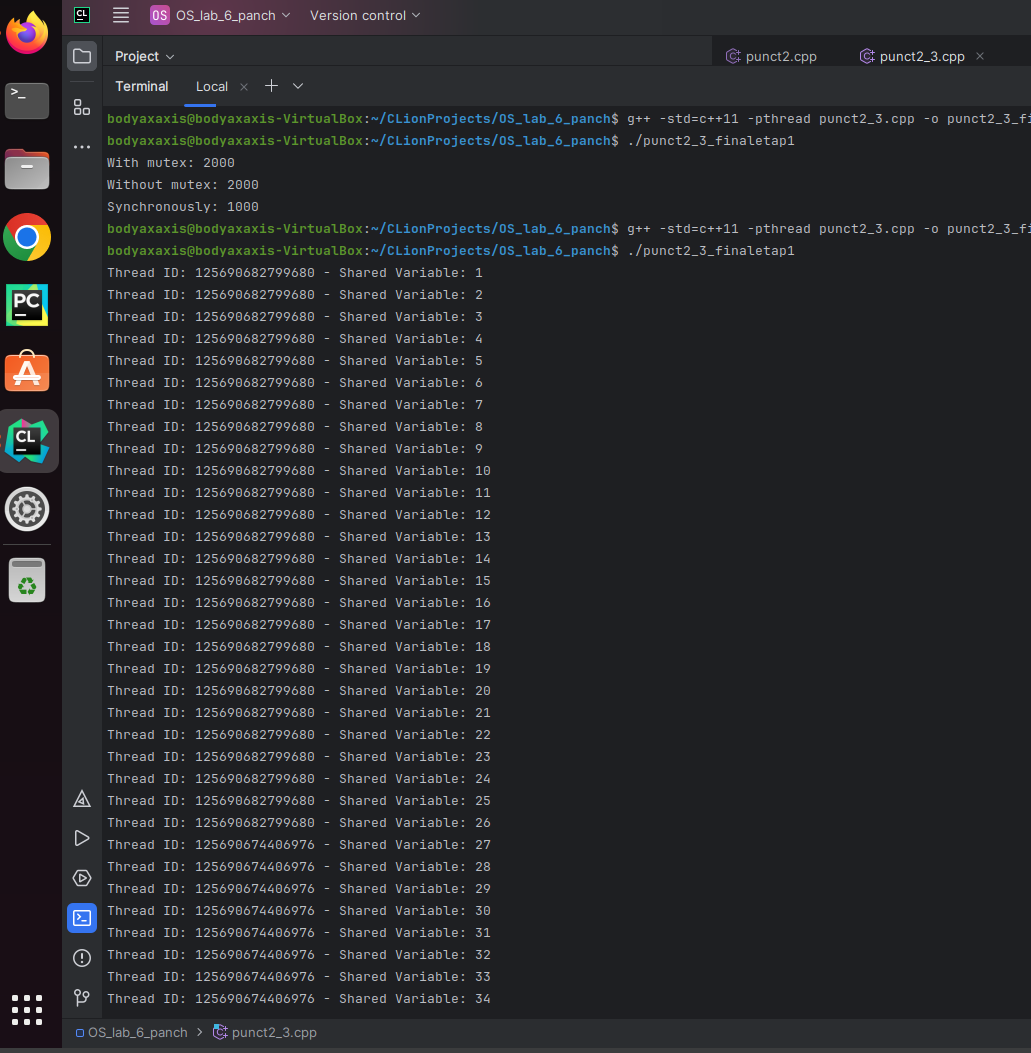
**Опис коду програми (пункт 2):**

Цей код виконує декілька завдань, включаючи множення матриць та інкрементацію змінних у багатопотоковому середовищі, в C++. Ось його основні функції та особливості:

* **Матриці A, B та C:** Задаються як двовимірні вектори цілих чисел. Розміри цих матриць визначаються за допомогою констант **n**, **m** та **k**.
* **Заповнення матриць випадковими числами:** Функція **fillMatrixRandomly()** заповнює матриці **A** та **B** випадковими числами у діапазоні від 1 до 10.
* **Множення матриць:** За допомогою функції **multiplyRowByColumn()**, виконується множення рядків матриці **A** на стовпці матриці **B**, а результат записується у матрицю **C**. Вивід в консоль синхронізується за допомогою мютекса **io\_mutex**.
* **Паралелізм у множенні матриць:** **matrixMultiply()** запускає множення матриць у багатопотоковому режимі, розподіляючи рядки матриці між різними потоками для більш ефективного виконання.
* **Інкрементація змінних:** Код порівнює три підходи до інкрементації спільної змінної у багатопотоковому середовищі:
  + **increaseWithMutex()** використовує мютекс для синхронізації доступу до змінної, забезпечуючи точність у багатопотоковому контексті.
  + **increaseWithoutMutex()** збільшує змінну без синхронізації, що може призвести до гонки за дані та невірного кінцевого результату.
  + **increaseAtomic()** використовує атомарну змінну для безпечної інкрементації в багатопотоковому середовищі без необхідності мютекса.
* **Вимірювання часу виконання:** Час виконання різних частин коду вимірюється та виводиться, демонструючи ефективність паралельної обробки.

Цей код слугує демонстрацією використання багатопотоковості в C++ для оптимізації обчислювальних завдань та вивчення відмінностей між різними методами синхронізації у багатопотоковому середовищі.

2\_3 2.3.\* (+3 балів) Досягти варіанту, коли таке паралельне додавання виконується повністю синхронно, тобто, *наприклад*, 1000 додавань виконуються паралельно двома потоками крок-в-крок і збільшують значення спільної змінної від 0 до 1000. Тобто, не тільки кожний з двох паралельних потоків збільшує значення від 0 до 1000, а й обидва, запущені в паралель, також збільшують від 0 до 1000 (а не до 2000, як очікувалось би).:



Код програми (2.3):

#include <iostream>

#include <thread>

#include <mutex>

#include <atomic>

const int iterations = 1000;

int sharedVariable = 0;

std::mutex mtx;

void increaseWithMutex() {

for (int i = 0; i < iterations; ++i) {

std::lock\_guard<std::mutex> lock(mtx);

sharedVariable++;

std::cout << "Thread ID: " << std::this\_thread::get\_id() << " - Shared Variable: " << sharedVariable << std::endl;

}

}

void increaseWithoutMutex() {

for (int i = 0; i < iterations; ++i) {

sharedVariable++;

std::cout << "Thread ID: " << std::this\_thread::get\_id() << " - Shared Variable: " << sharedVariable << std::endl;

}

}

void increaseSynchronously(std::atomic<int>& counter) {

while (true) {

int local = counter.load();

if (local >= iterations) break;

if (counter.compare\_exchange\_strong(local, local + 1)) {

std::lock\_guard<std::mutex> lock(mtx);

sharedVariable++;

std::cout << "Thread ID: " << std::this\_thread::get\_id() << " - Shared Variable: " << sharedVariable << std::endl;

}

}

}

int main() {

// З використанням критичного сегменту (mutex)

sharedVariable = 0;

std::thread t1(increaseWithMutex);

std::thread t2(increaseWithMutex);

t1.join();

t2.join();

// Без використання критичного сегменту (mutex)

sharedVariable = 0;

t1 = std::thread(increaseWithoutMutex);

t2 = std::thread(increaseWithoutMutex);

t1.join();

t2.join();

// Синхронне збільшення

sharedVariable = 0;

std::atomic<int> counter(0);

t1 = std::thread(increaseSynchronously, std::ref(counter));

t2 = std::thread(increaseSynchronously, std::ref(counter));

t1.join();

t2.join();

return 0;

}

**Опис коду програми (2.3):**

Цей код в C++ демонструє три підходи до інкрементації спільної змінної у багатопотоковому середовищі, кожен з яких ілюструє різні способи синхронізації доступу до спільних даних.

* **Функція increaseWithMutex():** Інкрементує спільну змінну з використанням мютекса (**mtx**) для синхронізації. Кожен доступ до змінної охороняється мютексом, забезпечуючи безпечний доступ в багатопотоковому середовищі. Цей підхід показує, як можна використовувати мютекси для запобігання гонки за даними.
* **Функція increaseWithoutMutex():** Інкрементує спільну змінну без синхронізації, що призводить до гонки за дані, коли два або більше потоки одночасно змінюють змінну. Це може призвести до невірного результату, оскільки інкрементації змінної можуть "затиратися".
* **Функція increaseSynchronously():** Інкрементує спільну змінну синхронно з використанням атомарної змінної **counter**. Використання **compare\_exchange\_strong** на атомарній змінній забезпечує синхронізацію, дозволяючи безпечно інкрементувати **sharedVariable**, коли вдається "замінити" значення атомарної змінної.

Кожен з цих підходів ілюструє важливі аспекти багатопотокової програми:

* Використання мютексів для синхронізації доступу до спільних ресурсів.
* Ризики, пов'язані з відсутністю синхронізації.
* Атомарні операції як засіб ефективної і безпечної синхронізації у багатопотоковому середовищі.

Основна ідея коду полягає в тому, щоб продемонструвати, як різні методи синхронізації впливають на поведінку і результати в багатопотокових програмах.

Загальна оцінка виконання:

**Перший код (Множення матриць)**

Я реалізував множення матриць A[n x m] та B[m x k], де матриці автоматично заповнюються випадковими числами.

**1.1.** Я продемонстрував паралелізм обчислень, виводячи результати множення матриць “по ходу обчислень” у форматі [x,y]=result, чим виконав вимогу про демонстрацію непослідовності обчислень.

**1.2.** Я дослідив швидкість виконання множення A\*B залежно від кількості потоків, але мені потрібно було детальніше продемонструвати та пояснити, як кількість потоків впливає на швидкодію. Це мало б включати знаходження оптимальної кількості потоків для найшвидшого множення.

**Другий код (Паралельна робота зі спільною змінною)**

Я моделював паралельну роботу зі спільною змінною, використовуючи різні методи синхронізації:

**2.1.** Я чітко продемонстрував різницю в результаті обчислень між використанням критичного сегменту (через мютекси та атоміки) та його відсутністю, показавши, як це впливає на кінцевий результат, що відповідає вимозі показати вплив синхронізації на обчислення.

**2.2.** Я намагався проаналізувати, чому відсутність критичного сегменту може призвести до некоректних результатів (race condition), але мені слід було більше зосередитися на аналізі часових різниць між різними методами синхронізації.

**2.3.** Я працював над досягненням швидшого результату при збереженні коректності обчислень, використовуючи атомарні операції, що демонструє зусилля щодо оптимізації швидкості та надійності.

**Третій код (Синхронна робота зі спільною змінною)**

У третьому коді я реалізував повністю синхронне збільшення спільної змінної від 0 до 1000 за допомогою двох паралельних потоків, забезпечивши, що кінцевий результат завжди є 1000, а не 2000, що виконує умову лабораторної роботи для повної синхронності додавань.

Загалом, я показав розуміння багатопотоковості і синхронізації у контексті виконання обчислень та роботи зі спільними ресурсами, виконуючи ключові вимоги лабораторної роботи.

**Висновок:**

Я засвоїв важливі концепції багатопотокового програмування та синхронізації в C++. Працюючи над лабораторною роботою, я реалізував множення матриць з використанням паралелізму та дослідив вплив кількості потоків на швидкість обчислень, що допомогло зрозуміти, як правильне розпаралелення може підвищити ефективність виконання програм.

Я також вивчив поведінку багатопотокових програм при роботі зі спільними ресурсами, використовуючи мютекси для синхронізації та атомарні змінні для оптимізації продуктивності. Це дало мені змогу порівняти різні методи синхронізації та зрозуміти важливість захисту спільних даних від гонки за ресурси.

Результати лабораторної роботи показали, що адекватне використання багатопотоковості та синхронізації може істотно вплинути на швидкодію та надійність програм. Зокрема, я виявив, що синхронна обробка з атомарними змінними може ефективно уникнути помилок, пов'язаних з гонкою за даними, та забезпечити коректну роботу програми в багатопотоковому середовищі.

У підсумку, ця лабораторна робота посилила моє розуміння багатопотоковості та синхронізації в програмуванні, навчивши мене не тільки технічним навичкам, але й стратегічному підходу до розробки ефективних і надійних багатопотокових додатків.