

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

**«Київський політехнічний інститут ім. І. Сікорського»**

**Інститут Прикладного Системного Аналізу**

**Кафедра Системного Проектування**

Лабораторна робота №3

З дисципліни «Паралельні обчислення»

Виконав:

Ст. гр. ДА-12

Кракович Павло

**Лабораторна робота №3**

**Дослідження базових операцій з потоками виконання**

**Мета роботи:** Розглянути поняття атомарності, навчитися працювати з атомарними змінними, а також ознайомитися з підходом написання паралельного коду без блокування.

**Завдання:**

1. Ознайомитися з визначенням: атомарна змінна, атомарна операція, неблокуючий алгоритм. Ознайомитися з деталями атомарності в обраній мові програмування.

2. Надати в протоколі роботи опис того, як саме досягається справжня атомарність операцій в обраній студентом мові програмування.

3. Виконати завдання за варіантом без використанням паралелізації. Заміряти час виконання завдання.

4. Виконати завдання за варіантом з використанням блокуючих примітивів синхронізації. Заміряти час виконання завдання.

5. Виконати завдання за варіантом з використанням атомарних змінних та CAS\CMPXCHNG операцій (більш високорівнені функції, що абстрагують дані операції, не приймаються для використання в даній роботі). Заміряти час виконання завдання.

6. Повторити пункти 2 – 4 з використанням різної розмірності даних та фіксованою кількістю потоків виконання.

7. В протокол занести отримані для пункту 5 результати у вигляді графіків залежності часу від кількості даних, надати порівняльний аналіз блокуючого та неблокуючого алгоритму з використання атомарних операцій.

8. Надати висновок, що повинен містити аналіз отриманих результатів.

Варіант роботи №7:

****

**Опис ключових моментів реалізації завдання:**

Атомарність – це властивість, що визначена для операцій, і говорить про те, що операція може мати лише один з двох станів – виконана, або не виконана, не допускаючи втручання інших процесів або потоків у середині виконання операції. Це забезпечує коректність результатів операцій, коли вони виконуються в паралельних середовищах.

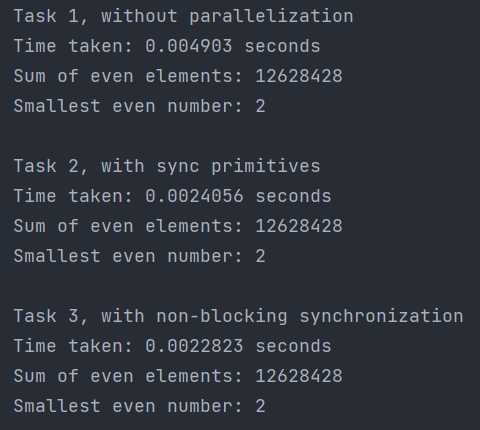
Коли операція є атомарною, це означає, що вона або виконалася повністю, або ще не виконувалася взагалі, і будь-які інші проміжні стани відсутні з точки зору стороннього наглядача (в даному випадку – інших потоків). Атомарність запобігає ситуаціям, коли стан операнда (змінної, над або за допомогою якої проводиться операція, чи системи в цілому) може змінитись під час виконання операції, порушуючи очікувану послідовність дій та приводячи донекоректних результатів.

Тож, справжня атомарність в C++ досягається за допомогою типу даних std::atomic. Цей тип даних гарантує, що операції над змінними атомарні, тобто вони відбуваються безперервно, без ризику переривання з боку інших потоків. Також, її забезпечує використання атомарних операцій, інтерфейсів бібліотеки atomic.

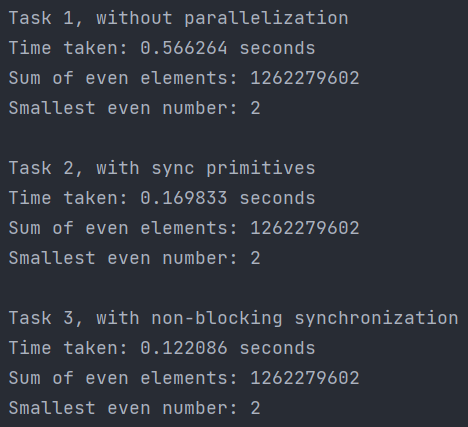
**Результати роботи програми за різної розмірності даних:**

Кількість потоків – 96. Для отримання більших затримок під час записування даних та кращого відображення різниці у швидкості виконання.

1. Розмір масиву – 500000



1. Розмір масиву – 50000000



**Аналіз результатів, порівняння блокуючого та неблокуючого механізму синхронізації:**

Кожен потік, що виконує функцію task2, блокує м'ютекси перед зміною глобальних змінних, що може призводити до затримок, особливо якщо багато потоків намагаються отримати доступ до ресурсів одночасно. Порівнючи з цим, використання атомарних операцій дозволяє уникнути блокування потоків і зменшити ймовірність виникнення гонок даних. Проте, це може призвести до більшої складності коду та можливої втрати продуктивності через постійну перевірку та повторні спроби виконання атомарних операцій.

1. Продуктивність:

* Блокуючий підхід (з м'ютексами) може мати більші затримки через блокування потоків у випадку конкуренції за ресурси.
* Неблокуючий підхід (з атомарними операціями) може мати менші затримки, оскільки уникне блокування, проте може збільшити обсяг коду та складність.

1. Безпека даних:

* Блокуючий підхід забезпечує простоту в реалізації та може бути менш схильний до помилок, але існує ризик гонок даних.
* Неблокуючий підхід зменшує ризик гонок даних, але потребує обережності в розробці та може вимагати додаткових зусиль для збереження коректності даних.

1. Простота реалізації:

* Блокуючий підхід зазвичай є простішим у реалізації через використання м'ютексів.
* Неблокуючий підхід може бути складнішим у реалізації через використання атомарних операцій та потребу уникнення гонок даних.

1. Масштабованість:

* Неблокуючий підхід може бути більш масштабованим, оскільки уникне блокування та дозволить більшу кількість потоків працювати паралельно.
* У вашому коді, неблокуючий підхід з атомарними операціями (з використанням std::atomic) може бути більш ефективним у випадку великої кількості потоків або великих обсягів даних.

**Висновок**

В ході реалізації завдання виявлено, що атомарність виконання операцій має велике значення для забезпечення коректності результатів у паралельних середовищах. Використання нового типу даних std::atomic в C++ надає можливість гарантованого виконання операцій без переривання з боку інших потоків, забезпечуючи атомарність дій.

Порівняння блокуючого та неблокуючого механізмів синхронізації показує, що використання атомарних операцій може зменшити затримки через уникнення блокування потоків. Однак це може призвести до більшої складності коду та можливої втрати продуктивності через постійну перевірку та повторні спроби виконання атомарних операцій.

Неблокуючий підхід з атомарними операціями може бути більш ефективним у випадку великої кількості потоків або великих обсягів даних, але вимагає обережності та додаткових зусиль для забезпечення коректності даних.

Таким чином, вибір між блокуючим та неблокуючим підходом залежить від конкретних вимог проекту щодо продуктивності, безпеки даних, простоти реалізації та масштабованості, і потребує уважного аналізу переваг та недоліків кожного методу з урахуванням контексту застосування.

**Додатки**

Код програми:

#include <iostream>  
#include <chrono>  
#include <atomic>  
#include <random>  
#include <thread>  
#include <vector>  
  
constexpr int arraySize = 500000;  
constexpr int numThreads = 12;  
int array[arraySize];  
  
  
static int getRandomValue(int lower\_bound, int upper\_bound) {  
 std::random\_device rd;  
  
 std::uniform\_int\_distribution<int> dist(lower\_bound, upper\_bound);  
  
 return dist(rd);  
}  
  
void fillArray() {  
 for (int & i : array) {  
 i = getRandomValue(0, 100);  
 }  
}  
  
void task1(int& sum, int& smallest) {  
 int localSum = 0;  
 int localSmallest = INT\_MAX;  
  
 for (int i : array) {  
 if (i % 2 == 0 && i != 0) {  
 localSum += i;  
 if (i < localSmallest) {  
 localSmallest = i;  
 }  
 }  
 }  
 sum += localSum;  
 if (localSmallest < smallest) {  
 smallest = localSmallest;  
 }  
}  
  
void task2(int& sum, int& smallest, int start, int end, std::mutex& sumMutex, std::mutex& smallestMutex) {  
 int localSum = 0;  
 int localSmallest = INT\_MAX;  
  
 for (int i = start; i < end; ++i) {  
 if (array[i] % 2 == 0 && array[i] != 0) {  
 localSum += array[i];  
 if (array[i] < localSmallest) {  
 localSmallest = array[i];  
 }  
 }  
 }  
 sumMutex.lock();  
 sum += localSum;  
 sumMutex.unlock();  
  
 smallestMutex.lock();  
 if (localSmallest < smallest) {  
 smallest = localSmallest;  
 }  
 smallestMutex.unlock();  
}  
  
void task3(std::atomic<int>& sum, std::atomic<int>& smallest, int start, int end) {  
 int localSum = 0;  
 int localSmallest = INT\_MAX;  
  
 for (int i = start; i < end; ++i) {  
 if (array[i] % 2 == 0 && array[i] != 0) {  
 localSum += array[i];  
 if (array[i] < localSmallest) {  
 localSmallest = array[i];  
 }  
 }  
 }  
 int expectedSum = sum.load();  
 while (!sum.compare\_exchange\_weak(expectedSum, expectedSum + localSum)) {}  
  
 int expectedSmallest = smallest;  
 while (localSmallest < expectedSmallest) {  
 if (smallest.compare\_exchange\_weak(expectedSmallest, localSmallest)) {  
 break;  
 }  
 }  
}  
  
  
int main() {  
 fillArray();  
  
 int sum\_task\_1 = 0;  
 int smallest\_task\_1 = INT\_MAX;  
 auto start\_task\_1 = std::chrono::high\_resolution\_clock::now();  
 task1(sum\_task\_1, smallest\_task\_1);  
 auto end\_task\_1 = std::chrono::high\_resolution\_clock::now();  
 std::chrono::duration<double> duration\_task\_1 = end\_task\_1 - start\_task\_1;  
 std::cout << "Task 1, without parallelization " << std::endl;  
 std::cout << "Time taken: " << duration\_task\_1.count() << " seconds" << std::endl;  
 std::cout << "Sum of even elements: " << sum\_task\_1 << std::endl;  
 std::cout << "Smallest even number: " << smallest\_task\_1 << std::endl << std::endl;  
  
  
 int sum\_task\_2 = 0;  
 int smallest\_task\_2 = INT\_MAX;  
 std::mutex sumMutex;  
 std::mutex smallestMutex;  
 std::vector<std::thread> threads\_task\_2;  
 threads\_task\_2.reserve(numThreads);  
 int chunkSize = arraySize / numThreads;  
 int remaining = arraySize % numThreads;  
 int startIdx\_task2 = 0;  
 int endIdx\_task2 = 0;  
  
 auto start\_task\_2 = std::chrono::high\_resolution\_clock::now();  
 for (int i = 0; i < numThreads; ++i) {  
 startIdx\_task2 = endIdx\_task2;  
 endIdx\_task2 = startIdx\_task2 + chunkSize + (i < remaining ? 1 : 0);  
 threads\_task\_2.emplace\_back(task2, std::ref(sum\_task\_2), std::ref(smallest\_task\_2), startIdx\_task2, endIdx\_task2, std::ref(sumMutex), std::ref(smallestMutex));  
 }  
  
 for (std::thread & t : threads\_task\_2) {  
 t.join();  
 }  
 auto end\_task\_2 = std::chrono::high\_resolution\_clock::now();  
  
 std::chrono::duration<double> duration\_task\_2 = end\_task\_2 - start\_task\_2;  
 std::cout << "Task 2, with sync primitives" << std::endl;  
 std::cout << "Time taken: " << duration\_task\_2.count() << " seconds" << std::endl;  
 std::cout << "Sum of even elements: " << sum\_task\_2 << std::endl;  
 std::cout << "Smallest even number: " << smallest\_task\_2 << std::endl << std::endl;  
  
  
 std::vector<std::thread> threads\_task\_3;  
 threads\_task\_3.reserve(numThreads);  
 std::atomic<int> sum\_task\_3(0);  
 std::atomic<int> smallest\_task\_3(INT\_MAX);  
 int startIdx\_task3 = 0;  
 int endIdx\_task3 = 0;  
  
 auto start\_task\_3 = std::chrono::high\_resolution\_clock::now();  
 for (int i = 0; i < numThreads; ++i) {  
 startIdx\_task3 = endIdx\_task3;  
 endIdx\_task3 = startIdx\_task3 + chunkSize + (i < remaining ? 1 : 0);  
 threads\_task\_3.emplace\_back(task3, std::ref(sum\_task\_3), std::ref(smallest\_task\_3), startIdx\_task3, endIdx\_task3);  
 }  
  
 for (std::thread & t : threads\_task\_3) {  
 t.join();  
 }  
 auto end\_task\_3 = std::chrono::high\_resolution\_clock::now();  
  
 std::chrono::duration<double> duration\_task\_3 = end\_task\_3 - start\_task\_3;  
 std::cout << "Task 3, with non-blocking synchronization" << std::endl;  
 std::cout << "Time taken: " << duration\_task\_3.count() << " seconds" << std::endl;  
 std::cout << "Sum of even elements: " << sum\_task\_3 << std::endl;  
 std::cout << "Smallest even number: " << smallest\_task\_3 << std::endl;  
  
  
 return 0;  
}