Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Факультет інформатики

Кафедра мультимедійних систем

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему **«Дослідження засобів потокового програмування, вільних від перешкод, блокувань і очікувань»**

Виконав: студент 4-го року навчання,

Освітньої програми «Інженерія програмного забезпечення», 121

Андрійченко Даниїл Сергійович

Керівник: Бублик Володимир Васильович

кандидат фіз.-мат. наук, доцент

Кваліфікаційна робота захищена

з оцінкою _____

Секретар ЕК _____

«____» _____ 20____ р.

Київ – 2023

Зміст

Вступ 3

Розділ 1: Огляд засобів управління пам'яттю, що застосовуються в багатопотоковому програмуванні 6

1.1 Бар'єри пам'яті 6

1.2 Семантики впорядкування пам'яті 10

Розділ 2: Розробка хеш таблиці, вільної від блокувань 12

2.1. Структура та інтерфейс класу хеш таблиці 12

2.2. Концепти для визначення умов які накладаються на шаблонні параметри ключів та значень 14

2.3. Реалізація операцій приєднання, видалення, входження і доступу 15

2.4. Застосування спеціалізації атомарних указників спільного використання для реалізації методу порівняння та обміну значень ... 18

Розділ 3: Порівняльний аналіз блокувальних та вільних від блокувань методів синхронізації та їх варіацій 21

3.1. Випробування та заміри швидкодії виконання змішаних операцій читання та запису 21

3.2 Порівняльний аналіз різних методів синхронізації залежно від кількості потоків та операцій 24

Висновки по роботі та рекомендації для подальших досліджень 28

Список використаної літератури 30

Вступ

Дизайн та реалізація структур данних – цікава та складна задача, особливо у випадку, коли з цією структурою працюватиме не один потік виконання.

Актуальність теми постійно зростає оскільки структури даних, вільні від блокувань, а деякі і від очікувань мають важливі переваги:

- Зі збільшенням кількості ядер у процесорах та багатопоточних застосунків, є велика потреба у структурах даних, які ефективно використовують паралельність без блокування.
- Ці структури даних масштабуються краще у відношенні до кількості потоків і ресурсів, порівняно з традиційними методами синхронізації, такими як м'ютекси чи семафори.
- Відсутність взаємних блокувань (deadlocks).
- В таких структурах даних, потоки можуть продовжувати виконувати операції навіть при збоях інших потоків, що підвищує відмовостійкість системи.
- Lock-free структури даних забезпечують низьку латентність операцій у порівнянні з традиційними структурами даних, що використовують блокування.

Усі ці переваги створюють великий сенс у тому, щоб розробляти алгоритми вільні від блокувань та очікувань там, де швидкодія системи є критично важливою складовою.

Структури даних, вільні від блокувань, поводять себе надзвичайно по різному, в залежності від архітектури системи, тому швидкодія тієї чи іншої реалізації міра дуже мінлива. Те, що буде швидко працювати на машині з однією архітектурою процесора, може працювати вкрай неефективно на машині з іншими архітектурами. Проте, незважаючи на цю мінливість, врахувавши особливості існуючих процесорних архітектур,

все ж існують закономірності у швидкості роботи структур даних, залежно від їх реалізації. І деякі рекомендації по дизайну та написанню такої все ж можна виділити.

Виходячи з цього, ***за мету данної роботи*** були поставлені розроблення та реалізація вільної від блокувань однієї з популярних структур данних, а саме хеш таблиці, а також порівняння програмної реалізації з іншими блокуючими методами захисту структури даних від конкурентного доступу.

Мета роботи зумовила наступне ***наукове завдання:***

- Виконати огляд засобів управління пам’яттю, що застосовуються в багатопотоковому програмуванні
- Зробити критичний огляд відомих методів захисту структур даних від конкурентних читань та записів, зокрема
    - синхронізація операцій у структурі за допомогою зовнішнього мьютексу
    - синхронізація операцій у структурі за допомогою застосування атомарних операцій, розглянувши семантики строгої гарантії послідовності виконань та більш послаблених
- Виконати порівняльний аналіз використаних методів на модельних (тестових) даних. Зробити висновки щодо можливостей застосування використаних методів.

Робота складається з 3-ьох розділів.

Перший розділ присвячено двум ключовим концепціям, пов’язаним з атомарними операціями та багатопотоковим програмуванням без перешкод: бар'єрам пам'яті та семантикам впорядкування пам'яті. Бар'єри пам'яті використовуються для впорядкування операцій над атомарними об'єктами, що забезпечують коректний обмін даними між потоками. Семантики впорядкування пам'яті, визначають порядок, в якому операції атомарних змінних мають бути виконані між різними потоками. Ми

розглянемо доступні семантики пам'яті в C++, а також те, як використовувати послаблені семантики для підвищення швидкодії програм.

У другому розділі описується побудова хеш таблиці, вільної від блокувань. Розібрано побудову інтерфейсу класу та реалізацію кожного окремого методу. Пояснено які засоби стандартної бібліотеки були використані, як досягається атомарність та синхронізація між операціями. Застосовано новітні можливості мови C++, такі як концепти та спеціалізація класу *std::atomic<std::shared_ptr>*.

Третій розділ присвячено тестуванню блокуючих реалізацій та побудованої у минолому розділі хеш таблиці. Наведено порівняльні таблиці та графіки.

В кінці буде наведено висновки проведених у цій роботі досліджень та рекомендації щодо розробки структур даних, вільних від блокувань. Також буде висунуто вектор подальших досліджень для майбутніх робіт.

Розділ 1: Огляд засобів управління пам’яттю, що застосовуються в багатопотоковому програмуванні

1.1 Бар’єри пам’яті

Бар'єри пам'яті, також відомі як огорожі пам'яті, відіграють важливу роль у світі паралельного програмування, особливо коли йдеться про багатоядерні процесори. Щоб зрозуміти їх важливість, ми повинні спочатку дослідити концепцію слабко впорядкованих моделей пам'яті, а потім обговорити, навіщо потрібні бар'єри пам'яті, що вони роблять з кешем процесора і чим відрізняються сильніші та слабші бар'єри пам'яті.

Слабко впорядкована модель пам'яті - це тип моделі узгодженості пам'яті, в якій порядок операцій з пам'яттю не обов'язково зберігається. Це означає, що з точки зору декількох процесорів або потоків читання і запис у пам'ять можуть виконуватися невпорядковано. Слабко впорядкована модель пам'яті слугує основою для багатьох сучасних багатоядерних процесорів, оскільки вона забезпечує переваги у продуктивності, дозволяючи апаратному забезпеченню змінювати порядок операцій з пам'яттю для підвищення ефективності. Однак ця гнучкість може призвести до проблем синхронізації, коли декілька потоків одночасно отримують доступ до спільних даних.

Бар'єри пам'яті потрібні для того, щоб гарантувати збереження правильного порядку операцій з пам'яттю, коли цього вимагає логіка програми. У багатоядерному середовищі різні потоки можуть одночасно звертатися до спільної пам'яті, що призводить до непередбачуваних результатів, якщо немає механізму для забезпечення впорядкування. Якщо не підтримується належна синхронізація, це може призвести до ледь помітних помилок, станів гонитви даних та загальної ненадійної поведінки.

Бар'єри пам'яті є важливими інструментами для програмістів, які гарантують належне впорядкування операцій з пам'яттю у паралельному середовищі, вирішуючи проблеми, що виникають у слабко впорядкованих моделях пам'яті.

У сучасних процесорах кожне ядро має свою кеш-пам'ять, яка зберігає дані, до яких часто звертаються, щоб мінімізувати час, необхідний для отримання інформації з основної пам'яті. Однак цей механізм кешування може ускладнити ситуацію, коли декілька ядер повинні отримати доступ до спільних даних одночасно. Бар'єри пам'яті допомагають впоратися з цією складністю, гарантуючи, що кеші оновлюються та синхронізуються належним чином, коли це необхідно.

Бар'єри пам'яті можуть очищати кеш процесора, змушуючи відкладені записи фіксуватися в основній пам'яті і гарантуючи, що наступні зчитування побачать найсвіжіші дані. Вони також можуть перешкоджати процесору змінювати порядок операцій з пам'яттю через бар'єр, зберігаючи запланований порядок виконання.

Бар'єри пам'яті можна розділити на дві категорії: сильніші та слабші. Різниця між ними полягає у рівні примусу, який вони забезпечують з точки зору впорядкування операцій з пам'яттю.

Сильніші бар'єри пам'яті

Сильніші бар'єри пам'яті забезпечують суворий порядок виконання операцій читання та запису по обидва боки бар'єру. Цей тип бар'єрів часто використовується, коли дуже важливо підтримувати точний порядок виконання набору операцій з пам'яттю. Деякі приклади сильних бар'єрів пам'яті включають

a. Повний бар’єр: Цей бар'єр запобігає зміні порядку виконання будь-яких операцій з пам'яттю до і після бар'єру. Він гарантує, що всі попередні завантаження та збереження будуть завершені до початку

будь-яких наступних завантажень та збережень.

b. Бар'єр запису: Цей бар'єр впливає лише на операції запису і гарантує, що всі записи, які передують бар'єру, будуть завершені до початку будь-яких записів, що слідують за бар'єром.

Слабші бар'єри пам'яті

Слабші бар'єри пам'яті забезпечують менш жорсткі гарантії впорядкування, пропонуючи процесору більшу гнучкість у зміні порядку операцій з пам'яттю з метою оптимізації. Вони використовуються, коли точний порядок операцій з пам'яттю не є критичним. Приклади слабших бар'єрів пам'яті включають

a. Бар'єр читання: Цей бар'єр впливає лише на операції завантаження, гарантуючи, що всі завантаження, які передують бар'єру, будуть завершені перед будь-якими завантаженнями, що йдуть за ним.

b. Бар'єри отримання-звільнення: Ці бар'єри забезпечують більш м'які гарантії впорядкування і часто використовуються в поєднанні з атомарними операціями. Бар'єр отримання гарантує, що наступні операції з пам'яттю не будуть виконуватися перед бар'єром, тоді як бар'єр звільнення гарантує, що попередні операції з пам'яттю будуть завершені перед бар'єром.

Таким чином, бар'єри пам'яті є незамінними інструментами у паралельному програмуванні, допомагаючи підтримувати бажаний порядок операцій з пам'яттю у багатоядерних процесорних середовищах. Вирішуючи проблеми, пов'язані зі слабко впорядкованими моделями пам'яті, бар'єри пам'яті забезпечують коректний доступ до даних, що розділяються, та їх модифікацію в декількох потоках.

Сильніші бар'єри пам'яті, такі як повні бар'єри та бар'єри запису,

забезпечують сувору впорядкованість операцій з пам'яттю, зберігаючи заплановану послідовність читання та запису. Ці бар'єри мають вирішальне значення, коли логіка програми вимагає точного впорядкування операцій з пам'яттю.

З іншого боку, слабші бар'єри пам'яті забезпечують менш суворі гарантії впорядкування, пропонуючи процесору більшу гнучкість у зміні порядку операцій з пам'яттю з метою оптимізації. Прикладами слабших бар'єрів пам'яті є бар'єри завантаження пам'яті та бар'єри отримання-звільнення. Ці бар'єри часто використовуються, коли точний порядок операцій з пам'яттю є менш важливим для правильного функціонування програми.

1.2 Семантики впорядкування пам’яті

Семантика впорядкування пам'яті для процесорів з послабленою моделью визначає порядок та видимість операцій у паралельному програмуванні. Стандарт C++11 вводить кілька обмежень щодо порядку пам'яті, надаючи детальний контроль над поведінкою атомарних операцій. Розуміння та належне використання цих семантик є важливим для розробки ефективних та коректних паралельних програм.

У C++ є шість порядків пам'яті:

1. *std::memory_order_relaxed*

   Цей порядок не накладає жодних обмежень на синхронізацію операції. Операції з розслабленим порядком можуть бути вільно переупорядковані компілятором та апаратним забезпеченням, що може призвести до покращення продуктивності. Однак це не дає гарантій видимості операцій між потоками, що робить його придатним лише в ситуаціях, коли немає залежностей між атомарними операціями.

2. *std::memory_order_consume*

   Цей порядок забезпечує часткове впорядкування між операцією та наступними операціями, що залежать від даних отриманих в першій. Однак, через його складність та обмежене практичне використання, часто рекомендується використовувати std::memory_order_acquire.

3. *std::memory_order_acquire*

   Порядок придбання гарантує, що наступні операції в поточному потоці не можуть бути переупорядковані перед операцією придбання. Він гарантує, що будь-які записи, зроблені іншим потоком перед відпусканням тієї самої атомарної змінної, будуть видимі поточному потоку після операції придбання.

4. *std::memory_order_release*

   Порядок відпускання гарантує, що попередні операції в поточному потоці не можуть бути переупорядковані після операції звільнення. Це забезпечує, що записи, зроблені поточним потоком перед операцією звільнення, будуть видимі іншим потокам, які виконають операції придбання на тій самій атомарній змінній.

5. *std::memory_order_acq_rel*

   Цей порядок є комбінацією порядків придбання та звільнення, гарантуючи властивості обох порядків пам'яті. Семантика придбання та звільнення зазвичай використовується на операціях які роблять і читання, і запис. Наприклад *compare-and-swap, fetch-and-add* і т.д.

6. *std::memory_order_seq_cst*

   Це найсильніший порядок пам'яті, який гарантує, що операції атомарних змінних виконуються в одному, узгодженому порядку між усіма потоками. Це є порядком пам'яті за замовчуванням для атомарних операцій. Однак, він може привести до зниження продуктивності через обмеження на впорядкування операцій.

Використання семантики пам'яті acquire/release в парі з розслабленими операціями може забезпечити кращу продуктивність порівняно з послідовно-узгодженою семантикою пам'яті. Послідовно-узгоджена семантика пам'яті гарантує повне впорядкування операцій між потоками, але це може привести до значного зниження продуктивності. Використання acquire/release і розслаблених операцій дозволяє розробникам керувати впорядкуванням операцій більш точно, сприяючи підвищенню продуктивності за рахунок додаткової складності розробки. Адже для успішного використання семантики пам'яті acquire/release та розслаблених операцій, розробники повинні досконало розуміти взаємодію потоків та механізми синхронізації.

Розділ 2: Розробка хеш таблиці, вільної від блокувань

2.1. Структура та інтерфейс класу хеш таблиці

Клас ***LockFreeHashMap*** розроблено як структуру даних, вільну від блокувань, яка підтримує одночасний доступ та модифікацію у багатопотоковому середовищі. Основна мета - мінімізувати конфліктні ситуації, уникаючи використання блокувань або інших механізмів синхронізації, забезпечуючи при цьому безпеку потоків та атомарність операцій.

Клас має два шаблонних параметрами: K і V. Ці параметри представляють типи ключа і значення для хеш таблиці. Це дозволяє класу бути гнучким та адаптованим для різних випадків використання.

Відкритий інтерфейс класу забезпечує основні операції з хеш таблицею, такі як вставка пари ключ-значення, видалення пари за допомогою ключа, перевірка наявності ключа, доступу до значення, та модифікація значення для заданого ключа. Інтерфейс розроблений таким чином, щоб бути простим і зручним у використанні, з чітко визначеними обов'язками для кожної функції:

```
template <typename K, typename V>
class LockFreeHashMap
{
public:
    explicit LockFreeHashMap(size_t numBuckets) :
                                        buckets(numBuckets) {}

    bool insert(const K& key, const V& value);
    bool remove(const K& key);
    bool contains(const K& key) const;
    std::optional<V> get(const K& key) const;
    bool modify(const K& key,
                    std::function<void(V&)>modificationFunction);

private:
    struct Node;
    std::vector<std::atomic<Node*>> buckets;
```

```
    size_t hash(const K& key) const;
};
```

При розробці конкурентних структур даних дуже важливо враховувати щоб всі функції відкритого інтерефейсу були атомарними. Саме тому метод *modify* приймає одним із своїх параметрів – функцію модифікація, що дозволяє користувачеві визначати власну логіку оновлення значення для будь яких типів, які будуть знаходитись у хеш таблиці , що робить метод більш універсальним.

У наведеній реалізації атомарність функцій-членів досягається переважно за рахунок використання атомарних вказівників у структурі Node, таких як *std::atomic<Node*> next* та *std::atomic<std::shared_ptr<V>> value*. Ці атомарні вказівники гарантують, що основні операції, такі як обхід списку, оновлення вказівників, читання або запис значень, виконуються атомарно, запобігаючи перегонам даних.

Внутрішньо хеш таблиця використовує масив "кошиків" для зберігання пар ключ-значення. Кожен кошик відповідає за хеш-значення. Коли пара ключ-значення додається до таблиці, хеш-значення ключа визначає, в яке відро слід помістити цю пару. Використання декількох відер допомагає розподілити пари ключ-значення більш рівномірно, зменшуючи ймовірність зіткнень і підвищуючи продуктивність.

Пари ключ-значення в кожному з кошиків зберігаються у вигляді вузлів в однозв'язному списку. Кожен вузол містить ключ, значення (у вигляді атомарного *shared_ptr*) та атомарний вказівник на наступний вузол у списку. Використання атомарних вказівників дозволяє оновлювати та обходити список без блокування, гарантуючи, що операції залишаються паралельними та ефективними.

Хеш-функція є ключовим компонентом будь-якої хеш-структури даних, оскільки вона визначає, як ключі розподіляються між доступними кошиками. У цій реалізації хеш-функція є простою та використовує

стандартну *std::hash* для типу ключа:

```
size_t LockFreeHashMap<K, V>::hash(const K& key) const {
    return std::hash<K>{}(key) % buckets.size();
}
```

Ділення на модуль кількості кошиків гарантує, що отримане хеш-значення потрапляє в допустимий діапазон індексів.

2.2. Концепти для визначення умов які накладаються на шаблонні параметри ключів та значень

У C++20 можна використати концепти для визначення вимог до параметрів шаблону. Наприклад тип ключа K має підтримувати хешуваня і бути порівнювальним, а тип значення V - копіювальним. Це накладає умови на шаблонні параметри, і, наприклад, якщо користувач хоче використати власне-визначену структуру як ключ таблиці, йому потрібно буде визначити спеціалізацію функції *std::hash* та *operator==*. Для визначення концептів *Hashable* та *Comparable* у фігурних дужках наводимо вираз який має підтримуватись, а також використаємо *std::convertible_to*. Концепт *Copyable* можна визначити через *std::copy_constructible*.

Наведемо повну реалізацію з застосуванням *requires* до *LockFreeHashMap*:

```
template <typename T>
concept Hashable = requires(T a)
{
    { std::hash<T>{}(a) } -> std::convertible_to<size_t>;
};

template <typename T>
concept Comparable = requires(T a, T b)
{
    { a == b } -> std::convertible_to<bool>;
};

template <typename T>
concept Copyable = std::copy_constructible<T>;


template <typename K, typename V>
    requires Hashable<K> && Comparable<K> && Copyable<V>
```

```
class LockFreeHashMap
{
        // as in previous section
};
```

З цими обмеженнями компілятор забезпечить відповідність типів ключа та значення, наданих при створенні екземплярів класу LockFreeHashMap, вказаним вимогам, що призведе до кращої безпеки типів та більш інформативних повідомлень про помилки, коли ці вимоги не будуть виконані.

2.3. Реалізація операцій приєднання, видалення, входження і доступу

Методи *insert* та *remove* представляють операції запису, а методи *contains* і *get* представляють операції читання. Давайте опишемо їх, розберемо призначення і те, як зберігається атомарність.

Insert - вставити нову пару ключ-значення в хеш таблицю.

Метод використовує цикл, щоб гарантувати, що новий вузол буде вставлено атомарно з використанням compare_exchange_weak. Це запобігає виникненню перегонів і гарантує, що тільки один потік може вставити новий вузол.

```
bool LockFreeHashMap<K, V>::insert(const K& key,
                                   const V& value)
{
    size_t index   = hash(key);
    Node*  newNode = new Node(key, value);
    Node*  oldHead = nullptr;

    while (true)
    {
        oldHead = buckets[index].load(std::memory_order_acquire);
        newNode->next.store(oldHead, std::memory_order_relaxed);

        Node* current = oldHead;
        while (current)
        {
            if (current->key == key)
            {
                delete newNode;
                return false;
```

```
            }
            current = current->next.load(std::memory_order_acquire);
        }

        if (buckets[index].compare_exchange_weak(oldHead,
              newNode, std::memory_order_release))
        {
            return true;
        }
    }
}
```

Remove - видалення пари ключ-значення на основі заданого ключа.

Метод обходить зв'язаний список у відповідному кошику і використовує compare_exchange_strong для атомарного видалення вузла, коли його знайдено. Це запобігає виникненню перегонів і гарантує, що тільки один потік може видалити вузол.

```
bool LockFreeHashMap<K, V>::remove(const K& key)
{
    size_t index   = hash(key);
    Node*  current = buckets[index].load(std::memory_order_acquire);
    Node*  prev    = nullptr;

    while (current)
    {
        if (current->key == key)
        {
            Node* next = current->next
                                    .load(std::memory_order_acquire);
            if (!prev)
            {
                if (buckets[index].compare_exchange_strong(current,
                     next, std::memory_order_acq_rel))
                {
                    delete current;
                    return true;
                }
            }
            else
            {
                if (prev-
>next.compare_exchange_strong(current,                    next,
std::memory_order_acq_rel))
                {
                    delete current;
                    return true;
                }
            }
        }

        prev    = current;
```

```
        current = current->next.load(std::memory_order_acquire);
    }

    return false;
}
```

Contains - перевіряє, чи містить хеш таблиця заданий ключ.

Метод обходить зв'язаний список у відповідному кошику, використовуючи атомарне завантаженн. Це гарантує, що метод бачить найсвіжіше значення вказівників у багатопотоковому середовищі і запобігає виникненню умов гонки.

```
bool LockFreeHashMap<K, V>::contains(const K& key) const
{
    size_t index   = hash(key);
    Node*  current = buckets[index].load(std::memory_order_acquire);

    while (current)
    {
        if (current->key == key)
        {
            return true;
        }
        current = current->next.load(std::memory_order_acquire);
    }

    return false;
}
```

Get - отримати значення, пов'язане з заданим ключем.

Метод повертає std::optional, який вказує, чи було знайдено значення чи ні. Якщо значення знайдено, воно буде загорнуте в std::optional, інакше повертається std::nullopt.

Подібно до методу contains, метод get використовує атомарне завантаження під час обходу зв'язаного списку. Також, він гарантує, що буде прочитано найсвіжіше значення, використовуючи атомарне завантаження на shared_ptr значенні.

```
bool LockFreeHashMap<K, V>::contains(const K& key) const
{
    size_t index   = hash(key);
    Node*  current = buckets[index].load(std::memory_order_acquire);
```

```
    while (current)
    {
        if (current->key == key)
        {
            return true;
        }
        current = current->next.load(std::memory_order_acquire);
    }

    return false;
}
```

У цій реалізації атомарність зберігається завдяки використанню атомарних операцій типу *compare_exchange_weak*, *compare_exchange_strong* та атомарних *load* та *store* операціях.

2.4. Застосування спеціалізації атомарних указників спільного використання для реалізації методу порівняння та обміну значень

Метод *modify* призначений для атомарного оновлення значення, пов'язаного із заданим ключем. Атомарність досягається завдяки використанню спеціалізації C++20 std::atomic для std::shared_ptr та підходу порівняння і заміни (compare-and-swap). Наведемо код:

```
bool LockFreeHashMap<K, V>::modify(const K& key,
                    std::function<void(V&)> modificationFunction)
{
    size_t index   = hash(key);
    Node*  current = buckets[index].load(std::memory_order_acquire);

    while (current)
    {
        if (current->key == key)
        {
            std::shared_ptr<V> oldValue, newValue;
            do
            {
                oldValue =
                    current->value.load(std::memory_order_acquire);
                newValue = std::make_shared<V>(*oldValue);
                modificationFunction(*newValue);
            }
            while (!current->value.compare_exchange_weak(oldValue,
```

```
                    newValue, std::memory_order_acq_rel));
            return true;
        }
        current = current->next.load(std::memory_order_acquire);
    }

    return false;
}
```

Спочатку метод обраховує хеш ключа і шукає відповідний вузол у відповідному кошику. Під час цього обходу він використовує атомарні завантаження з семантикою acquire, щоб гарантувати, що найсвіжіші значення вказівників будуть видимими. Якщо цільовий вузол знайдено, метод переходить до модифікації значення, пов'язаного з ключем.

У даній реалізації значення зберігається як атомарний спільний вказівник (*std::atomic<std::shared_ptr<V>>*). Ця спеціалізація std::atomic гарантує, що спільні вказівники можна безпечно використовувати у паралельних контекстах без явного блокування. Це дозволяє виконувати атомарні операції, такі як *load, store, compare_exchange_weak, compare_exchange_strong* безпосередньо над спільними вказівниками.

Для досягнення атомарної модифікації метод використовує цикл з операцією *compare_exchange_weak*. Він починається з атомарного завантаження поточного значення спільного вказівника. Потім створюється новий спільний вказівник, копіюється поточне значення і до копії застосовується функція модифікації, надана користувачем. Після цього операція *CAS* застосовується для атомарного оновлення спільного вказівника у цільовому вузлі лише у тому випадку, якщо його значення не змінилося з моменту завантаження. Цей процес повторюється доти, доки операція *CAS* не завершиться успішно, гарантуючи, що оновлення буде атомарним, а інші одночасні модифікації не заважатимуть цій операції.

Використання CAS у методі modify гарантує, що якщо декілька потоків намагаються одночасно змінити одне і те ж значення, то тільки один з них зможе досягти успіху за один раз. Решті доведеться повторити операцію з

оновленим значенням. Такий підхід гарантує, що *LockFreeHashTable* залишається послідовною і безпечною для використання в багатопотокових середовищах.

Розділ 3: Порівняльний аналіз блокувальних та вільних від блокувань методів синхронізації та їх варіацій

3.1. Випробування та заміри швидкодії виконання змішаних операцій читання та запису

Для тестування можна відтворити одну із прикладних ситуацій застосування хеш таблиці, вільної від блокувань. Уявімо що ми будуємо багатокористувацьку гру, яка моделює ігровий світ на сервері і усі найбільш критичні дані зберігаються та ретельно обробляються саме на ньому. Користувачами ж будуть гравці, позицію у світі, здоров'я, рахунок яких потрібно розсилати по мережі усім клієнтам. Отже структура даних яка використовується на сервері повинна підтримувати конкурентний доступ, і ба більше, бути ефективною при наявності в грі багатьох гравців, які постійно і одночасно можуть намагатись ці дані читати чи змінювати.

Клас клієнту гравця максимально спрощеним можна подати у такому вигляді:

```
class PlayerClient
{
public:
    explicit PlayerClient(const int playerId): _id(playerId) { }
    int getId() const { return _id; }

private:
    int _id;
};
```

Структура з даними гравця:

```
struct PlayerData
{
    std::string nickName;
    int score{};
    bool bPlayingOffenseTeam{};
    // other specific data
};
```

Одразу визначимо для неї оператор виводу:

```
inline std::ostream& operator<<(std::ostream& os,
                                const PlayerData& playerData)
{
    os << "Nickname: " << playerData.nickName << ", score: " <<
playerData.score
        << ", offenseTeam = " << playerData.bPlayingOffenseTeam;
    return os;
}
```

Інтерфейс серверу від якого ми будемо наслідуватись та кожен нащадок буду представляти певний метод синхронізації виглядиме так:

```
class IServer
{
public:
    virtual bool connect(const PlayerClient& playerClient);
    virtual bool disconnect(const PlayerClient& playerClient);
    virtual std::optional<PlayerData> getPlayerData(int playerId)=0;
    virtual bool modifyPlayersData(int playerId, const
       std::function<bool(PlayerData&)>& modificationFunction) = 0;

    virtual ~IServer() = default;

private:
    virtual bool clientConnected(int playerId) const = 0;
};
```

Код тестування у головному потоці:

```
constexpr int PLAYERS_AMOUNT = 16;
constexpr int OPERATIONS_AMOUNT = 100'000;
std::mutex IOS_MUTEX;

int main()
{
    std::cout << std::boolalpha;

    // create server in main thread
    LockFreeServer server;
    for (int id = 0; id < PLAYERS_AMOUNT; ++id)
    {
        const PlayerClient playerClient(id);
        server.connect(playerClient);
    }

    // create player clients in new threads and let them connect and
register on server
    std::thread* playersThreads[PLAYERS_AMOUNT];
    for (int id = 0; id < PLAYERS_AMOUNT; ++id)
    {
        playersThreads[id] = new std::thread([&server, id]()
        {
            const std::chrono::steady_clock::time_point begin =
```

```
std::chrono::steady_clock::now();

            for (int op = 0; op < OPERATIONS_AMOUNT; ++op)
            {
                for (int i = 0; i < PLAYERS_AMOUNT; ++i)
                {
                    const int deltaScore = id % 2 == 0 ? 1 : -1;
                    server.modifyPlayersData(i,
                     [deltaScore](PlayerData& playerData)
                     {
                        playerData.score += deltaScore;
                        return true;
                     });
                }
            }

            const std::chrono::steady_clock::time_point end =
                     std::chrono::steady_clock::now();
            IOS_MUTEX.lock();
            std::cout << "Thread " << id << " finished in " <<
              std::chrono::duration_cast<std::chrono::milliseconds>(
                     end - begin).count() << " ms.\n";
            IOS_MUTEX.unlock();
        });
    }

    for (std::thread* playerThread : playersThreads)
    {
        playerThread->join();
    }

    for (int id = 0; id < PLAYERS_AMOUNT; ++id)
    {
        std::optional playersData = server.getPlayerData(id);
        if (!playersData.has_value()) continue;

        std::cout << "Id: " << id << ", " << playersData.value()
              << '\n';
    }

    return 0;
}
```

Спочатку ініціалізується сервер і підключається до нього кожен гравець-клієнт. Загальна кількість гравців визначається константою *PLAYERS_AMOUNT*.

Далі створюється декілька потоків для кожного гравця-клієнта, імітуючи одночасні з'єднання і запити до сервера. Кожен потік виконує заздалегідь визначену кількість операцій (визначається *OPERATIONS_AMOUNT*), де

він модифікує рахунок всіх гравців у хеш-таблиці сервера. Модифікація або збільшує, або зменшує рахунок на 1, в залежності від ідентифікатора гравця (парний чи непарний).

Після завершення всіх операцій вимірюється час виконання кожного потоку і виводиться на консоль разом з данними кожного гравця. Щоб запобігти виникненню гонки при виведенні результатів, використовується *std::mutex IOS_MUTEX*.

Запустивши тестер, ви можете спостерігати за поведінкою хеш-таблиці під високим навантаженням операцій читання і запису, а також переконатися, що реалізація працює належним чином у багатопотоковому середовищі. Надалі таке тестування дозволить нам провести порівняння методів синхронізації, вимірявши час виконання при різній кількості гравців (потоків) та операцій.

3.2 Порівняльний аналіз різних методів синхронізації залежно від кількості потоків та операцій

Умови експерименту будуть наступними:

- Кількість операцій читання та запису буде співвідноситись як 3:1 відповідно.
- Обрано та закріплено сумарну к-сть операцій яку виконають усі потоки, наприклад 32 000 000. Далі, оскільки нас цікавить поведінка при збільшенні конкурентності, будемо збільшувати кількість потоків і пропорціонально зменшувати кількість операцій щоб підтримувати час у розумних межах. Кількість операцій була обрана еврестично для конкретної машини і таким чином щоб час не був занадто малий і можна було покладатись на отримані результати.
- Для кожного методу синхронізації за кожного заданого співвідношення потоків та операцій, експеримент та заміри часу

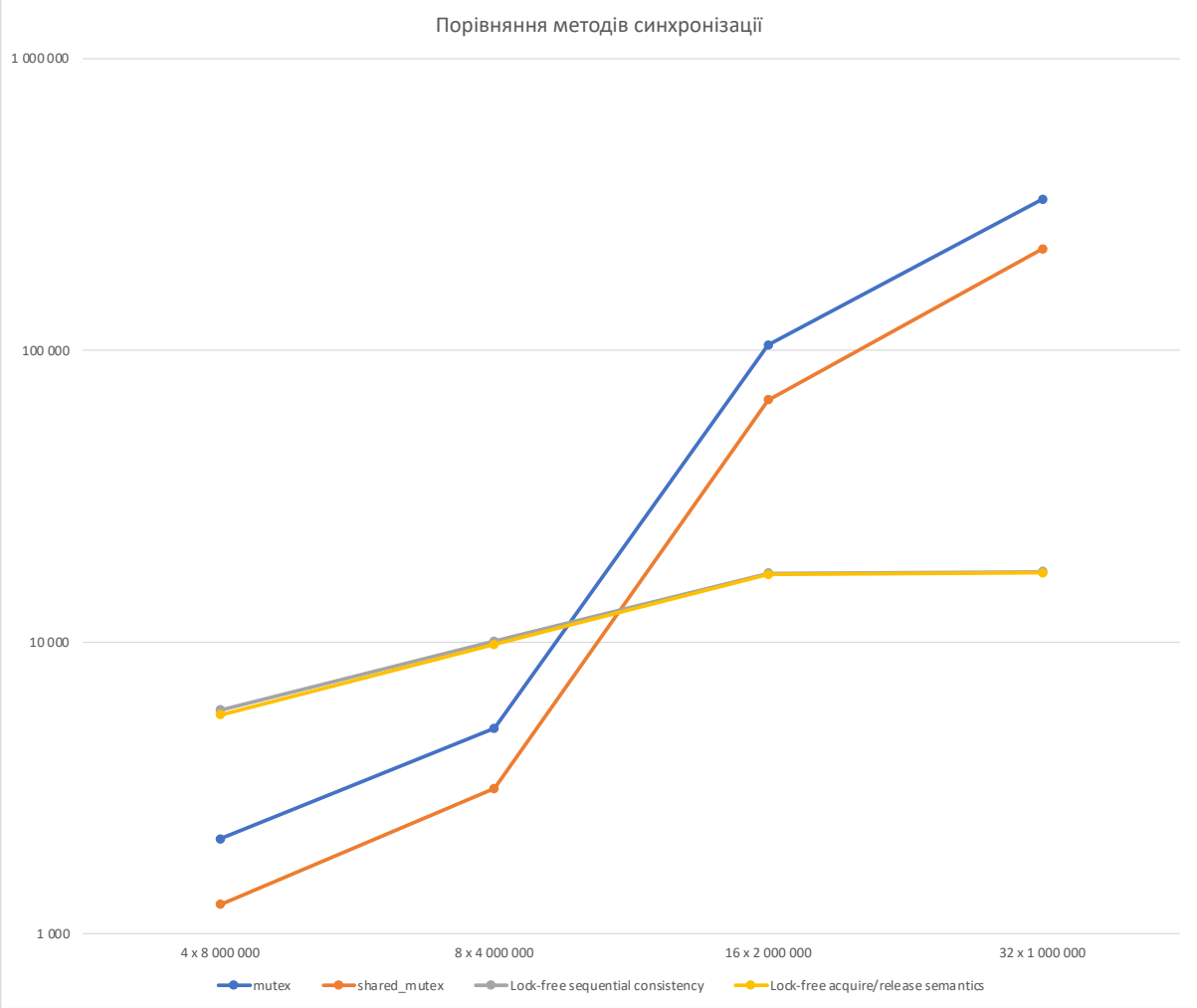
проведуться тричі для підвищення точності результатів. У таблиці буде наведено середній час виконання потоку.

Короткий опис методів синхронізації, які брали участь у випробуванні:

1. mutex – зовнішня синхронізація не конкурентної структури даних (у данному випробовуванні *std::unordered_map*) за допомогою *std::mutex*. Відповідно абсолютно усі операції, що читання, що запису є блокуючими.
2. shared_mutex – зовнішня синхронізація описана вище, проте використано *std::shared_mutex* з методами *lock_shared* для читання та ексклюзивний *lock* для запису. Операції читання не є блокуючими, операції запису блокуючі.
3. Lock-free acquire/release semantics – реалізована у попередньому розділі хеш таблиця, вільна від блокувань, з застосуванням acquire/release та relaxed семантик, де це можливо, для підвищення швидкодії.
4. Lock-free sequential consistency - реалізована у попередньому розділі хеш таблиця, вільна від блокувань, з застосуванням семантики за замовчуванням – sequential consistency.

Результати дослідження:

К-сть потоків та операцій / час виконання, ms	mutex	shared_mutex	Lock-free acquire/release semantics	Lock-free sequential consistency
4 x 8 000 000	2 112	1 257	5 623	5 852
8 x 4 000 000	5 061	3 148	9 804	10 063
16 x 2 000 000	104 162	67 524	17 035	17 180
32 x 1 000 000	329 353	222 324	17 231	17 400



За результатами дослідження, видно, що швидкість виконання для різних методів синхронізації змінюється по різному. Загалом він зростає, адже збільшується конкурентність. Проте для одних методів він зростає стрімкіше, для інших менш стрімко.

Mutex: Час виконання операцій із зростанням кількості потоків збільшується значно. Даний метод синхронізації менш ефективний при високому рівні конкуренції через блокування всіх операцій читання та запису.

Shared_mutex: Час виконання операцій також зростає зі збільшенням кількості потоків, але не так значно, як у випадку з простим мьютексом. Операції читання в даному методі не блокуються, тому він ефективніший при більш високому рівні конкуренції.

Lock-free acquire/release semantics: Час виконання операцій стабільно зростає, але не так стрімко, як у випадку з обидвами типами мьютексів. Даний метод синхронізації виявляється набагато ефективнішим при великій кількості потоків через відсутність блокувань.

Lock-free sequential consistency: Результати майже такі самі, як і за використання слабших семантик, але час виконання операцій зовсім трохи збільшився через використання семантики за замовчуванням - sequential consistency. У відсоткову співвідношенні різниця завжди складає менше 1%, а зазвичай є навіть меншою за 0.1%.

Загалом, можна зробити висновок, що методи синхронізації, вільні від блокувань (з застосуванням будь якої семантики) є найбільш ефективними при високому рівні конкуренції, тоді як блокуючі виявляються менш швидкими.

Висновки по роботі та рекомендації для подальших досліджень

Структури даних без блокування є потужним інструментом у паралельному програмуванні, оскільки вони можуть забезпечити значне покращення швидкодії системи порівняно з традиційними підходами, що базуються на блокуванні. Вони особливо цінні у сценаріях, де конкурентність висока, а очікування блокування може призвести до значних накладних витрат. Однак, розробка та реалізація таких структур вимагає глибоких знань та досвіду у паралельному програмуванні, моделях пам'яті та атомарних операціях.

Розробка структур даних без блокування є складним завданням, оскільки необхідно забезпечити коректність, узгодженість та швидкодію при паралельному доступі. Слід ретельно зважити компроміс між приростом продуктивності, який дають неблокуючі структури даних, і часом, витраченим на їх розробку. У багатьох випадках переваги підвищеної продуктивності можуть переважити додатковий час на розробку, особливо у високопродуктивних системах реального часу. Однак, щоб приймати обґрунтовані рішення щодо їх впровадження, необхідно мати глибоке розуміння основних принципів і методів, які лежать в їх основі.

Деякі загальні поради щодо розробки та впровадження структур даних без блокування включають наступне:

Глибоко вивчіть базові моделі апаратного забезпечення та пам'яті, а також атомарні операції та семантику впорядкування пам'яті, що надаються мовою програмування.

Завжди враховуйте конкретний варіант використання та вимоги до продуктивності при виборі між підходами на основі блокування та без блокування.

Ретельно протестуйте свою реалізацію без блокування, щоб забезпечити коректність та узгодженість при одночасному доступі.

У подальших роботах можна розглянути та дослідити програмування без очікування. Структури даних без очікування гарантують, що кожна операція завершиться за скінченну кількість кроків, незалежно від дій інших потоків. Це може призвести до ще кращої продуктивності та передбачуваності порівняно зі структурами даних без блокування, особливо в системах реального часу, де обмежений час відгуку є критично важливим. Однак програмування без очікування є ще більш складною і спеціалізованою областю паралельного програмування, і його практичне застосування та переваги слід ретельно вивчити, перш ніж приймати рішення про його використання в проекті.

Список використаної літератури

1. “Concurency in Action”, 2nd Edition, Anthony Williams [Друковане видання]
2. “Concurrency with Modern C++”, Rainer Grimm [Друковане видання]
3. Документація Concurrency support library [Електронний ресурс]
   Concurrency support library - cppreference.com
4. Документація про порядки пам’яті в C++ [Електронний ресурс]
   std::memory_order - cppreference.com
5. Цикл статтей про роботу процесорного кеша та бар’єри пам’яті [Електронний ресурс]
   Пам’ять монопроцесору
   Пам’ять мультипроцесору
   Бар’єри пам’яті. Оптимізуючі сутності
6. Протокол когерентності кешу MESI [Електронний ресурс]
   MESI protocol - Wikipedia
7. Цикл статтей про атомарність, бар’єри пам’яті, модель пам’яті C++ [Електронний ресурс]
   Lock-free структури даних. Атомарність та атомарні примітиви
   Lock-free структуры даних. Звідки пішли бар’єри пам’яті
   Lock-free структуры даних. Модель пам’яті