Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Кафедра мультимедійних систем

Факультет інформатики

Курсова робота

на тему: «**КЛАСИФІКАЦІЯ ТА ВДОСКОНАЛЕННЯ БАГАТОПОТОКОВИХ МОДЕЛЕЙ ПАТЕРНІВ ПРОЕКТУВАННЯ**»

Виконав: студент 1-го року навчання,

Освітньої програми «Інженерія програмного забезпечення», 121

Божко Владислав Вадимович

Керівник Бублик В.В.,_____
кандидат фіз.-мат. наук, доцент

Рецензент _____
(прізвище та ініціали)

Курсова робота захищена
з оцінкою _____

Секретар ЕК _____
«____» _____ 20____ р.

Київ – 2022

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри інформатики,

к. ф.-м. н. С. С. Гороховський

_____

(підпис)

“____”_____2021 р.

ЗАВДАННЯ

на курсову роботу

студента 1-го року навчання Божка Владислава Вадимовича

Тема: Класифікація та вдосконалення багатопотокових моделей патернів проектування

Зміст ТЧ до кваліфікаційної роботи:

Індивідуальне завдання

Календарний план

Зміст

Перелік умовних позначень

Вступ

Розділ 1: Огляд інструментарію для багатопотокової розробки

Розділ 2: Вдосконалення багатопотокової моделі патерну «Реактор»

Висновки

Перелік використаних джерел

Дата видачі “___” _____ 2021 р. Керівник _____

(підпис)

Завдання отримав _____

(підпис)

КАЛЕНДАРНИЙ ПЛАН ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Тема: Розробка застосування для проектування, відображення та аналізу ER-моделей

Календарний план виконання роботи:

№ п/п	Назва етапу курсового проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу.	12.10.2021	
2.	Ознайомлення з існуючою інформацією по темі	13.10.2021	
3.	Проектування прикладів	02.11.2021	
4.	Початок створення практичної частини	21.12.2021	
5.	Подання проміжної версії практичної частини	01.03.2022	
6.	Аналіз практичної частини; її корегування	26.04.2022	
7.	Початок написання теоретичної частини	01.05.2022	
8.	Подання проміжної версії текстової частини	15.05.2022	
9.	Остаточне завершення написання теоретичної частини роботи та корегування практичної частини;	26.05.2022	
10.	Створення презентації	01.06.2022	
11.	Захист курсової роботи	09.06.2022	

Студент Божко В. В.

Керівник Бублик В. В.

“_____” _____

ЗМІСТ

ПЕРЕЛІК ТЕРМІНІВ ТА УМОВНИХ ПОЗНАЧЕНЬ

- М’ютекс (англ. mutex - mutual exclusion) – примітив синхронізації, призначений для забезпечення одночасного доступу до певного об’єкту лише одним потоком
- HTTP (HyperText Transfer Protocol) – протокол передачі даних прикладного рівня
- Сокет – структура даних, яка описує мережеве з’єднання
- POSIX (Portable Operating System Interface) – набір стандартів, що описують контракти між операційною системою та програмою для UNIX-подібних систем
- Чанк (англ. chunk – шматок) – фрагмент певної інформації або файлу

ВСТУП

Процесори поступово досягають межі тактової частоти, за якої вони залишаються відносно енергоефективними. Виробники апаратного забезпечення пропонують нові архітектури, а технології спекулятивного виконання та розгалужені системи кешів дозволяють суттєво зменшити час, який процесор витрачає на очікування. Проте зростання потужності окремого ядра процесора вже суттєво не поспішає за зростанням складності програмного забезпечення, а отже, єдиним виходом для розробників є опанування багатопроцесорних систем – як відзначив у своїй статті Г. Саттер: «безкоштовний ланч закінчився» [7], маючи на увазі значне уповільнення звичного приросту швидкості процесорів.

Наприкінці 1990–х та початку 2000–х ця проблема стала очевидною, і перед інженерами постало питання адаптації ПЗ до багатопотокового середовища. Мова C++ та її стандартна бібліотека протягом довгого часу залишались осторонь багатопотокової розробки. Будь-яка робота із багатопотоковим середовищем могла бути реалізована лише за допомогою системних бібліотек, різних для кожної операційної системи. Це значно ускладнювало портування програм на інші системи, адже не лише програмні інтерфейси, а й поведінка таких програм могли сильно відрізнятись від однієї системи до іншої. Це створювало неабиякий попит на крос-платформену реалізацію багатопотокових примітивів, який задовольняли сторонні бібліотеки (напр., Boost).

У Unix-подібних системах базовим способом взаємодії між потоками є взірець fork-join. Він дозволяє створення нового потоку за допомогою функції fork() та очікування на виконання потоку за допомогою функції join(). Проте навіть створені таким чином потоки зазвичай потребують додаткової синхронізації, яка раніше імплементувалася або засобами, специфічними для конкретної операційної системи, або сторонніми бібліотеками на кшталт Boost. Саме для цього у стандарті C++11 було додано набір нових бібліотек та ключових слів, націлених на спрощення розробки багатопотокових програм.

Мета курсової роботи – виявити особливості механізмів синхронізації, наданих у стандартній бібліотеці C++ та вдосконалити з їх допомогою багатопотокові моделі взірців проектування

Завдання курсової роботи – дослідити вузькі місця існуючих багатопотокових моделей взірців проектування; вдосконалити ці моделі; протестувати отримані моделі на їх швидкодію.

Об'єкт дослідження – багатопотокові моделі взірців проектування

Предметом дослідження є особливості багатопотокових моделей взірців з точки зору швидкодії цих моделей у високонавантажених програмах

У рамках курсової роботи було досліджено та вдосконалено існуючу модель взірця «Реактор», та розглянуто і протестовано його модифікацію – взірець «Проактор». Були розглянуті недоліки та потенційні покращення існуючої моделі та було продемонстровано потенціал для її використання у високонавантажених застосунках.

Використане програмне забезпечення

Для розробки прикладів було використано:

- Середовище розробки Qt Creator
- Модуль Qt Network фреймворку Qt із крос-платформеними імплементаціями протоколів мережевої взаємодії
- Заголовні файли системних викликів ядра ОС Linux
- Набір інструментів збірки CMake

Робота складається з двох розділів.

У першому розділі роботи було розглянуто засоби синхронізації багатопотокових програм, надані у стандартній бібліотеці C++. Для цих механізмів були визначені способи їхнього застосування, переваги та обмеження.

У другому розділі, на прикладі минулорічної моделі взірця «Реактор» та прикладу «Веб-сервер», що її демонстрував, було описано застосування цих механізмів синхронізації в поєднанні із асинхронним доступом до файлової системи для вдосконалення цієї моделі та покращення її швидкодії.

РОЗДІЛ 1: ОГЛЯД ІНСТРУМЕНТАРІЮ ДЛЯ БАГАТОПОТОКОВОЇ РОЗРОБКИ

1.1 Механізми синхронізації, додані до C++11

Стандарт C++11 включає в себе бібліотеку універсальних абстракцій над потоками та примітивами синхронізації (м'ютексами, атомарними типами та ін.). Це значно спрощує портування програм між системами та дозволяє значно краще розуміти і керувати потоком виконання програми.

Для створення та керування потоками, стандарт C++ пропонує примітив std::thread. Він дозволяє створення нового потоку виконання та приєднання до нього. Для платформо-специфічних операцій можна скористатись методом std::thread::native_handle(), який надає доступ до низькорівневого представлення цього потоку.

1.1.1. М'ютекси

Стандартним та одним із найбільш відомих засобів синхронізації багатопотокових програм є м'ютекс. [1] Він дозволяє обмежити доступ до спільного ресурсу таким чином, що в кожен момент часу лише один потік має до нього доступ. Це може допомогти уникнути «стану гонитви» (англ. race condition), коли дані можуть бути пошкоджені або невірно отримані через конфліктуючі одночасні операції або невизначений їх порядок.

Блокування м'ютексів зазвичай відбувається за допомогою об'єктів-«замків» (std::lock_guard, std::unique_lock) – конструктори цих класів приймають на вхід м'ютекс, який вони можуть блокувати, та в залежності від їх імплементації, блокують отриманий м'ютекс, чи надають таку можливість. Так, std::lock_guard використовують для виконання доступу до захищеного спільного ресурсу в ідіомі RAII (Resource Acquisition Is Initialization) – м'ютекс одразу блокується за створення замку, та вивільняється за його видалення. Така поведінка дозволяє мінімізувати кількість коду, необхідного для синхронізації, та спрощує керування доступом до спільних ресурсів. Своєю чергою,

std::unique_lock не виконує блокування автоматично та надає гнучкіші засоби ручного контролю доступу: блокування та розблокування м’ютексу за допомогою методів lock() та unlock(), блокування з обмеженням у часі (try_lock_for(), try_lock_until()).

Проте, хоча м’ютекс – це дуже зручний та простий у розумінні механізм, він може призвести до логічних помилок у програмі, найрозповсюдженішою з яких є взаємне блокування (англ. deadlock). Взаємне блокування виникає в ситуації, коли потокам необхідно одночасно отримати доступ до декількох ресурсів, захищених м’ютексами, наприклад, безпечно отримати елемент зі стеку і додати його до іншої колекції. В такому разі, якщо порядок доступу до таких ресурсів не є жорстко контрольованим, може виникнути ситуація, в якій два потоки заблокували м’ютекси для двох ресурсів, і обидва з них намагаються отримати доступ до іншого. В такому разі, припинити виконання цих двох потоків стає неможливим, і програма зупиняється в нескінченному очікуванні.

Гарним тоном у багатопотоковій розробці є повне уникнення ситуацій, в яких потоку може знадобитися блокуючий доступ одразу до декількох ресурсів. Але інколи уникнути таких ситуацій неможливо, і саме для таких випадків, у стандарт C++17 було включено ще один механізм блокування м’ютексів – std::scoped_lock. Він може приймати на вхід декілька м’ютексів, та гарантує, що у разі блокування інших м’ютексів за допомогою цього ж механізму, взаємного блокування буде уникнено.

1.1.2. Умовні змінні

Часто у багатопотоковій програмі виникає ситуація, коли потоку виконання необхідно дочекатись настання певної події – додавання елементу в колекцію, досягнення певного значення лічильника, та ін. За допомогою м’ютексів реалізація такого механізму є нетривіальною. Для таких випадків у стандарті C++11 описано іще один механізм синхронізації потоків – умовні змінні, представлені класом std::condition_variable.

Принцип роботи умовних змінних наступний: спершу створюється умовна змінна. Потім, потік, який планує очікувати на настання певної умови, стає в очікування на настання цієї умови, передаючи в умовну змінну об’єкт-замок, за допомогою якого буде виконуватись блокування, та викликаний булевий вираз, в якому перевіряється виконання чи невиконання необхідної умови. Потім, за настання цієї умови, м’ютекс розблоковується, та потік продовжує виконання.

Проте, якщо лише потік не перевіряє настання умови в нескінченному циклі, він не може самостійно дізнатись про те, що умова настала, і виконання можна продовжувати. Для цього в умовних змінних застосовано механізм сповіщень – інші потоки, що виконують операції, які можуть вплинути на виконання умови (напр., додають елемент у колекцію або змінюють значення спільної змінної), можуть сповістити очікуючі потоки про те, що виконання умови необхідно перевірити. Для цього в класі std::condition_variable наявні два методи: notify_one() та notify_all(). Вони використовуються для, відповідно, сповіщення лише першого потоку в черзі, або усіх потоків, що очікують на цю змінну. Сам механізм сповіщень працює наступним чином: в методі очікування виконується цикл, що перевіряє виконання умови, та, якщо вона виконується, цикл завершується, а якщо ні – блокується наданий замок. При сповіщенні, замок розблоковується, та, відповідно, очікуючим потоком виконується ще одна ітерація циклу перевірки.

1.1.3. Future та promise

Механізм умовних змінних є дуже універсальним, та надає розробникам можливість реалізувати багато сценаріїв очікування даних чи подій. Проте, для деяких випадків, створення окремої умовної змінної та відповідного об’єкту-замка може виявитись невиправдано складним та об’ємним. Це стосується, наприклад, очікувань на результати виконання окремих асинхронних обчислень. Саме для цього, до стандарту C++11, було додано механізми Future та Promise, представлені класами std::future, std::shared_future, std::packaged_task, та std::promise.

Об’єкти класу std::future використовують для очікування на певні дані. Для цього використовують його методи wait() та get(). Метод wait() (або wait_for() для очікування протягом певного часу) – блокуючий виклик, який повертає потоку виконання, коли необхідні дані стають доступними. Після цього, за допомогою методу get() ці дані можна отримати. Варто зазначити, що у класу std::future немає копіювального конструктора, лише пересувний. Для очікування на одні й ті самі дані з декількох потоків пропонується використання клас std::shared_future, об’єкт якого можна створити із std::future, і який уже можна копіювати. У стандартній бібліотеці C++ є декілька механізмів створення або отримання об’єктів std::future: за допомогою функції std::async(), і класів std::packaged_task та std::promise.

Функція std::async() – найпростіший спосіб створити Future. Вона приймає на вхід виконуваний вираз, та повертає std::future, який буде містити в собі дані, коли переданий вираз виконається. Для обчислення виконуваного виразу, всередині std::async створюється новий потік, в якому асинхронно відбуваються обчислення.

```
std::future future(std::async([]() {
    return 42;
}));
future.wait();
future.get(); // Поверне 42
```

Приклад 1.1 – Створення Future за допомогою std::async

Цей механізм дозволяє з мінімальними зусиллями асинхронно виконати необхідні обчислення, або блокуюче очікування, проте створення окремого потоку на обчислення далеко не завжди є раціональним. Через великі накладні витрати на створення потоку та переключення контексту, створення окремого потоку на обчислення має сенс лише у випадку дійсно довготривалих обчислень. А додаткові витрати пам’яті на стек виконання потоку (~1 МБ на сучасній

системі) суттєво обмежують кількість потоків, які можна одночасно створити, а отже ці кількість таких обчислень має бути невеликою.

Ще один спосіб створити Future – за допомогою класу std::packaged_task. Конструктор цього класу приймає будь-який виконуваний вираз і створює «обгортку» навколо нього. Об’єкт цього класу також є виконуваним виразом – його можна виконати в необхідний час в потрібному потоці – а результат виконання отримати за допомогою об’єкту std::future, повернутого методом get_future().

Для випадків, коли створення окремого потоку не є бажаним, гарним варіантом для виконання таких обчислень є взірець Пул Потоків (англ. Thread Pool). [8] А для взаємодії Future із власними механізмами виконання існує клас std::promise. Об’єкт цього класу дозволяє за допомогою методу get_future() очікуючому потоку отримати об’єкт класу std::future, а потоку, який виконує обчислення – передати значення в цей future за допомогою методу set_value().

```
std::promise<int> promise;

// Thread 1
std::this_thread::sleep_for(std::chrono::seconds(1));
promise.set_value(16);

// Thread 2
auto future(promise.get_future());
future.wait();
future.get(); // Поверне 16
```

Приклад 1.2 – Використання std::promise для передачі даних між потоками

1.2 Експериментальні включення до стандартної бібліотеки C++17

Описані вище механізми Future та Promise значно спрощують очікування на обрахунок результату асинхронної операції, проте вони є досить обмеженими у своєму застосуванні – так, у розробників немає зручної опції керувати

очікуванням на одразу декілька розрахунків. Так, якщо нам необхідно дочекатись одразу усіх обчислень, ми можемо в циклі викликати метод wait() для всіх об’єктів Future. Проте, якщо ми хочемо отримувати результати багатьох обчислень одразу по мірі їх готовності, ми не можемо зробити цього з блокуючими викликами, і повинні в циклі перевіряти готовність усіх об’єктів Future, що значно вплине на швидкодію, і може в багатьох випадках нівелювати приріст ефективності, отриманий від асинхронного виконання операцій.

Саме для подолання цих обмежень робоча група C++ запропонувала стандарт ISO/IEC TS 19571:2016 – Technical specification for C++ extensions for concurrency (Технічні специфікації розширень C++ для паралельності). Цей стандарт пропонував:

- Тимчасове включення розширених версій класів std::future та std::promise у простір імен std::experimental
- Додавання функцій std::experimental::when_any() та std::experimental::when_all() для одночасного очікування на кілька асинхронних обчислень
- Додавання додаткових механізмів синхронізації: Latch та Barrier

До класу std::experimental::future було додано механізм продовжень (англ. continuations) за допомогою методу then(). За допомогою цього методу пропонувалось виконувати додаткову обробку результатів асинхронних обчислень та їх очікування.

```
std::experimental::future<int> input_future;
auto output_future(input_future.then(
    [](int i) { return std::to_string(i); }
);
output_future.wait();
output_future.get(); // Поверне рядок
```

Приклад 1.3 – Використання методу std::experimental::future::then()

Також цей стандарт мав запропонувати способи очікування багатьох Future за допомогою функцій std::experimental::when_all() та std::experimental::when_any(). Ці функції могли бути використані для очікування виконання усіх Future, та найпершого з них, відповідно.

Проте цей стандарт було зрештою відхилено підгрупою SG1 комітету C++ Committee – розширення для std::future та std::promise було відкладено до наступних версій стандарту, а механізми синхронізації Latch та Barrier у стандарті C++20 було додано у набір стандартної бібліотеки.

1.2.1. Майбутнє Future та Promise

Для вирішення проблем примітивів Future та Promise, підгрупа SG1 запропонувала повністю переосмислити виконання асинхронних операцій та отримання їх результатів у стандартній бібліотеці C++. Для цього ними було внесено пропозицію до стандарту P0443R12 – A Unified Executors Proposal for C++. [4] Автори цієї пропозиції пропонують виділити три примітивні інтерфейси роботи з асинхронними операціями: виконавці (executors), передавачі (senders) та приймачі (receivers). За допомогою цих інтерфейсів можна описати роботу як класів Future та Promise, так і складніших алгоритмів із ними та механізмів синхронізації. Декомпозиція виконання асинхронних операцій і відправки та отримання їх результатів дозволить значно гнучкіше керувати окремими їх аспектами – наприклад, обирати пріоритет виконання або механізм блокування. Також це дозволить створити платформо-незалежну імплементацію алгоритмів when_all та when_any в межах самої стандартної бібліотеки. А універсальний інтерфейс виконавців асинхронних подій дозволив авторам включити у пропозицію до стандартної бібліотеки також імплементацію взірця «Пул потоків».

1.3 Атомарні типи

Хоча м’ютекси та інші механізми синхронізації потоків є зручними та зрозумілими для розробника, і дозволяють гарантувати одночасний доступ лише одного потоку до будь-яких даних, додаткові витрати на їх використання роблять їх часте застосування недоцільним або навіть шкідливим. Але, додаткова синхронізація може бути потрібна не лише громіздким структурам даних – класичним прикладом є числа з рухомою крапкою подвійної точності. На деяких системах, таке число займає в пам’яті два машинних слова, а отже операція його запису не може бути виконана за один такт процесору. У разі паралельного запису в змінну такого типу з декількох потоків, може виникнути ситуація гонитви, коли одна половина числа буде записана одним потоком, а друга – іншим, і дані в пам’яті залишаться пошкодженими. Для уникнення такої ситуації, модель пам’яті C++ пропонує атомарні типи – спеціальні відповідники звичайних типів даних, усі операції над якими є атомарними – тобто, такими, які ніколи не виявляються напіввиконаними для будь-якого з потоків – інший потік може отримати значення змінної атомарного типу лише або до, або після операції.

Стандартні атомарні типи включають у себе булевий тип, усі числові типи та указники. Усі ці типи перебувають у заголовному файлі <atomic> та мають префікс atomic_ перед їхньою назвою (напр., atomic_llong як відповідник long long) Крім того, C++ надає ще один атомарний тип, що немає стандартного відповідника – атомарний прапорець (std::atomic_flag).

1.3.1. Операції над атомарними типами

Варто зазначити, що стандарт C++ не гарантує, що імплементація атомарних типів на кінцевій системі буде ефективною. Справа в тому, що далеко не всі архітектури процесорів підтримують атомарні операції на рівні набору команд. І якщо відповідні набори команд відсутні, то імплементація атомарних типів замінюється на значно менш ефективну імплементацію на основі

блокувань. Задля перевірки наявності нативної реалізації під час компіляції, стандартні атомарні типи надають булеву константу часу компіляції is_always_lockfree(). Єдиний тип, для якого стандарт C++ гарантує реалізацію без блокувань – це атомарний прапорець.

Атомарний прапорець схожий на булеву змінну у тому значенні, що він може набувати лише двох значень: прапорець або встановлений, або ні. Прапорець завжди ініціалізується невстановленим, беручи за початкове значення константу ATOMIC_FLAG_INIT. Після ініціалізації, ми можемо змінювати його значення за допомогою двох атомарних операцій: clear() щоб атомарно його зняти, або test_and_set() – встановити прапорець та дізнатись його попереднє значення. Набір операцій над прапорцем виглядає примітивним, але це дозволяє гарантувати його ефективність на будь-якій системі, на яку можна скомпілювати C++11 код. Для переважної більшості алгоритмів прапорця буде недостатньо, а отже необхідно буде скористатись атомарними відповідниками стандартних типів.

Атомарні типи у C++ пропонують методи для читання значення із атомарної змінної – load(), та для запису – store() для атомарної заміни значення, exchange() для заміни значення та отримання попереднього, та compare_exchange_weak() або compare_exchange_strong() для заміни значення з перевіркою. Останні два методи є особливо важливими для алгоритмів, що працюють з атомарними значеннями. Справа в тому, що атомарні типи зазвичай не підтримують увесь набір операцій з присвоєнням (напр., цілочисельні атомарні типи у C++ не підтримують /=). А отже, для виконання ділення з присвоєнням над змінною атомарного типу необхідно виконати дві атомарні операції, для яких без додаткової синхронізації неможливо гарантувати, що вони виконаються безпосередньо одна за одною, без втручання інших потоків. І саме для таких ситуацій можна використати операцію заміни з перевіркою (англ. compare-and-swap), яка у C++ представлена методами compare_exchange_*. Ця операція дозволяє виконати зміну значення всередині змінної лише після перевірки її

початкового значення, із поверненням результату операції (успіх чи невдача). Наприклад, якщо ми отримуємо зі змінної значення 10 та ділимо його на 2, ми хочемо записати туди значення 5 лише якщо там все ще зберігається 10. Якщо між читанням та діленням значення змінної змінилося, нам скоріш за все необхідно перевиконати ділення та записати туди вже новий результат:

```
std::atomic_int a;
int value(a.load()); // a = 10
// result == true, якщо у змінну було записано 5
bool result(a.compare_exchange_strong(value, value/2));
```

Приклад 1.4 – використання compare-and-swap операції для ділення з присвоєнням

Така операція теж є атомарною, і підтримується більшістю сучасних процесорів, а отже може бути використана у високоефективних неблокуючих алгоритмах. Для ситуацій, коли попереднє значення не має впливу на наступне, розробник може скористатись методами store() та exchange().

1.4 Механізми синхронізації, використані у цій роботі

Оскільки робота розглядає різні аспекти застосування взірців проектування у багатопотоковому середовищі, виникла необхідність у використанні різних механізмів синхронізації. У цій роботі було застосовано:

- М’ютекси для загальних випадків синхронізації між потоками
- Умовні змінні для очікувань на настання подій
- Future та Promise для передачі результатів асинхронних операцій із файловою системою
- Атомарні змінні для забезпечення потокобезпечності черги Майкла-Скотта

РОЗДІЛ 2: ВДОСКОНАЛЕННЯ БАГАТОПОТОКОВОЇ МОДЕЛІ ПАТЕРНУ «РЕАКТОР»

2.1 Існуюча модель та її обмеження

Минулого року нами була представлена модель взірця «Реактор». [3] Ідея цього взірця полягає в обробці великого обсягу завдань одним потоком з використанням черги подій та асинхронних операцій. За рахунок мінімізації перемикань контексту між потоками та очікувань усередині основного потоку, цей взірець має значну перевагу у швидкодії за виконання великої кількості паралельних задач, особливо якщо такі задачі пов’язані із операціями введення-виведення.

2.1.1. Приклад «Веб-сервер»

Приклад застосування цього взірця, розглянутий у цій роботі – веб-сервер, який надає доступ на читання до певної директорії по протоколу HTTP.

2.2 Методологія тестування швидкодії

Для вимірювання швидкодії веб-серверу було вирішено вимірювати кількість запитів, які застосування може обробити за секунду. Оскільки ця метрика виміряється ззовні, вона дозволяє заміряти реальну швидкість роботи кінцевого застосування і підкреслити вузькі місця реалізації. Також, одночасне виконання багатьох запитів може слугувати як перевірка веб-серверу навантаженням та допомогти у виявленні витоків пам’яті або некоректного вивільнення ресурсів.

Для вимірювання кількості запитів на секунду, було використано утиліту ApacheBench, яка постачається разом із веб-сервером Apache, але може бути використана для перевірки будь-якого серверу, що підтримує протокол HTTP. Ця утиліта дозволяє зазначити загальну кількість необхідних запитів та кількість одночасних підключень до серверу, що дозволяє змоделювати різні умови його

використання. Задля мінімізації впливу мережевого середовища на результати тестування, воно проводилось на тому ж комп’ютері, на якому був запущений сервер.

Через залежність більш ефективної імплементації файлової системи на заголовні файли операційної системи Linux, тести виконувались на комп’ютерах двох конфігурацій, на одній з яких – у віртуальній машині.

Конфігурація 1, віртуальна машина:

- Операційна система хоста: macOS 10.15
- Операційна система гостя: Manjaro Linux, ядро 5.15
- Процесор: Intel Core i7-9750H, 6 фізичних, 12 логічних ядер, 8 логічних ядер надано віртуальній машині
- Оперативна пам’ять: 32 ГБ DDR4, 2667 МГц, 10 ГБ надано віртуальній машині
- Сховище: Apple SSD nVME, 40 ГБ надано віртуальній машині

Конфігурація 2, виділений сервер:

- Операційна система: Manjaro Linux, ядро 5.4
- Процесор: AMD Ryzen 5 3600, 6 фізичних, 12 логічних ядер
- Оперативна пам’ять: 16 ГБ DDR4, 3733 МГц
- Сховище: 256 ГБ nVME SSD, 2 ТБ 5400 rpm HDD

Дві різні конфігурації дозволяють перевірити ефективність отриманої реалізації веб-серверу на процесорах різної архітектури (AMD та Intel), у віртуальному середовищі та безпосередньо на апаратному забезпеченні, та при роботі як із твердотільним сховищем, так і з жорстким диском. Це краще дасть зрозуміти про потенційні вузькі місця імплементації та варіанти її покращення.

2.3 Вхідні дані

Минулорічна реалізація не використовувала всі потенційні переваги патерну «Реактор». Так, Через недоступність у фреймворку Qt засобів асинхронного доступу до файлової системи, на момент написання попередньої роботи було вирішено змоделювати асинхронний доступ за допомогою синхронного доступу та взірця «Пул потоків». Хоча такий підхід давав пристойний результат в порівнянні із багатопотоковою обробкою запитів, він суттєво програвав реалізаціям зі справжнім асинхронним доступом до файлової системи. Для порівняння було вирішено обрати веб-сервер nginx – один з найпопулярніших та найшвидших веб-серверів, який є загальновизнаним стандартом індустрії. Для створення рівних умов тестування, у конфігурації сервера nginx було відімкнено опцію sendfile – ця опція дозволяє застосування однойменного системного виклику ОС Linux, який дозволяє передати файл із файлової системи в TCP сокет засобами операційної системи напряму, без участі веб-серверу. Оскільки ефективність доступу до файлової системи є одним із головних чинників, що впливають на швидкодію серверу, використання цього налаштування є недоцільним під час тестування.

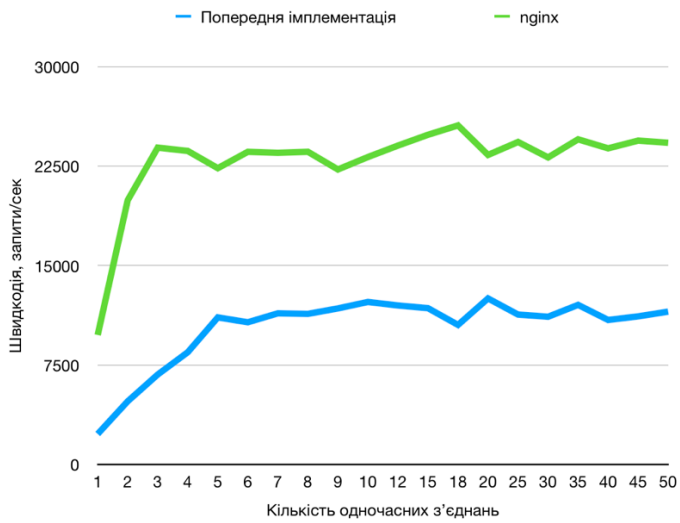

Рисунок 2.1 – Порівняння швидкості обробки запитів попередньої реалізації та веб-серверу nginx в залежності від кількості одночасних з'єднань. Тестувалось на конфігурації 2

Проте це – достатньо оптимістичний сценарій для імплементації на основі пулу потоків. Зі збільшенням розміру отримуваних файлів, цей розрив стає ще більш суттєвим та перевищує чотирикратний.

Рисунок 2.2 – Порівняння швидкості обробки запитів попередньої реалізації та веб-серверу nginx в залежності від кількості одночасних з'єднань на файлах великого розміру. Зелені лінії – файли з розміром 25 КБ, червоні – 100 КБ. Світліші лінії – попередня власна реалізація, темніші – nginx

Враховуючи, що сервер має оперувати і значно більшими файлами, реалізація, ефективність якої залежить від їх об'єму, не є придатною до використання в реальному середовищі і потребує доопрацювання.

2.4 Можливі оптимізації та покращення

В минулорічній роботі першочерговим напрямом оптимізації було названо взаємодію із файловою системою. Попередня реалізація мала одразу декілька проблем, що впливали на її ефективність:

- Доступ до файлової системи виконувався синхронно в інших потоках, що додавало накладні витрати на перемикання контексту між

багатьма потоками, додатковий необхідний об’єм оперативної пам’яті на їх створення та навантаження на процесор.

- Через специфіку емуляції асинхронної файлової системи, виникала необхідність виділяти оперативну пам’ять під увесь файл, щоб передати його зміст між потоками. Це суттєво обмежувало максимальний розмір файлів, підтримуваних сервером та зменшувало швидкодію.

Ці два основні недоліки потребують обов’язкового вирішення в межах цієї роботи і їх вирішення має суттєво покращити ефективність розглянутого веб-серверу.

2.4.1. Робота із файловою системою

Перш за все, для того щоб позбутись громіздкого механізму емуляції асинхронної роботи із файловою системою на основі пулу потоків, необхідно перейти на справжній асинхронний доступ. На жаль, ані стандартна бібліотека C++, ані використаний фреймворк Qt не надають засобів для асинхронної роботи із файловою системою. А операційна система macOS, на якій виконувались розробка і тестування попередньої імплементації веб-серверу, не надає ефективних механізмів асинхронної роботи, пропонуючи лише використання неефективних POSIX-сумісних системних викликів, реалізація яких в операційній системі використовує схожий, хоча і ефективніший, механізм емуляції асинхронного доступу.

Для розв’язання цієї проблеми було вирішено перенести розробку та тестування застосунку на ОС Linux. Linux пропонує асинхронний доступ до файлової системи як за допомогою POSIX-сумісних системних викликів (із тими ж нюансами, що і в macOS), так і за допомогою системних викликів Linux, що мають ефективнішу імплементацію в ядрі ОС. Варто зазначити, що асинхронний доступ до файлової системи має багато обмежень (напр., у дозволених режимах доступу до файлів, обмеження запису та ін.), а його ефективність суттєво залежить від використаного у системі драйверу файлової системи. Проте, за

сприятливих умов, такий доступ може суттєво покращити ефективність роботи ПЗ. У цій роботі для асинхронного доступу до файлової системи було використано POSIX-сумісні системні виклики aio_read(), aio_suspend() та aio_return(), а також системні виклики Linux io_setup(), io_submit() та io_getevents(). [6] Для відкриття файлів на читання та їх звільнення використовуються стандартні системні виклики open() та close().

Робота із POSIX-сумісними асинхронними викликами виконується так:

- Файл відкривається на читання за допомогою виклику open()
- Створюється запит на читання із файлу у вигляді об’єкту структури aiocb
- Цей запит відправляється файловій системі за допомогою виклику aio_read()
- Масив усіх поточних запитів передається у системний виклик aio_suspend(). Цей виклик – блокуючий, і він повертає виконання, коли хоча б один із запитів виконується операційною системою
- Статус виконання запиту та кількість прочитаних байтів отримується за допомогою виклику aio_return()
- Зміст файлу отримується із буфера, переданого разом із запитом на читання.
- Файл закривається викликом close()

Робота із системними викликами Linux є дуже схожою за своєю природою, проте дещо відрізняється:

- Створюється об’єкт структури io_context_t із налаштуваннями асинхронного доступу. Цей об’єкт ініціалізується за допомогою системного виклику io_setup()
- Файл так само відкривається на читання за допомогою виклику open()
- Створюється запит на читання із файлу, цього разу – у вигляді об’єкту структури **iocb**, з дещо іншим набором полів
- Масив усіх поточних запитів передається у системний виклик io_submit(). Цей виклик – неблокуючий

- Потік стає на очікування настання асинхронних подій за допомогою системного виклику io_getevents(). На відміну від aio_suspend(), цей виклик дозволяє зазначити мінімальну та максимальну кількість очікуваних подій, що може стати у нагоді за оптимізації роботи із файловою системою, за рахунок групування декількох відповідей і зменшення кількості необхідних системних викликів. Цей виклик повертає масив виконаних запитів до файлової системи.
- Зміст файлів отримується із буферів, переданих разом із виконаними запитами до файлової системи.

Хоча обидва ці варіанти схожі між собою, системні виклики Linux пропонують більшу гнучкість у роботі за рахунок можливості налаштування кількості очікуваних запитів. Також, за рахунок явної передачі масиву усіх виконаних запитів, зникає необхідність ручної перевірки статусу виконання усіх надісланих запитів, що дозволяє зменшити кількість системних викликів і трохи спростити код. Проте в цілому, обидва ці механізми надають можливість достатньо зручного асинхронного доступу до файлової системи і можуть бути застосовані у покращеній власній реалізації веб-серверу.

2.4.2. Виділення пам’ятті

Друга проблема, частково пов’язана із першою – значні витрати на виділення пам’яті. Через особливості синхронного доступу до файлової системи, сервер вимушений спочатку прочитати увесь файл в пам’ять, і лише після цього надіслати його через сокет користувачу. Виділення значних об’ємів пам’яті – важка процедура, до того ж, такий механізм передачі даних накладає суттєві обмеження на максимальний розмір файлів та кількість одночасно під’єднаних клієнтів.

Одним із способів позбутися необхідності виділяти оперативну пам’ять у великій кількості – зчитувати файли частково, зберігаючи в пам’яті невеликі чанки, які будуть одразу ж передаватись клієнту. Проте такий підхід потребує значної переробки механізму мережевої взаємодії із клієнтами.

2.4.3. Комунікація із клієнтською частиною

Робота із цілими файлами в оперативній пам’яті сприяла спрощенню імплементації протоколу HTTP. Наприклад, протокол HTTP вимагає, щоб сервер на початку своєї відповіді надіслав розмір очікуваного файлу. Тримаючи файл в оперативній пам’яті, це виконується тривіально, адже розмір вмісту файлу вже визначено. Проте, із частковим зчитуванням, виникає потреба робити окремий запит до файлової системи, аби дізнатися розмір файлу. Також, це вимагатиме переробки механізму роботи реактору, збільшивши кількість подій, які він опрацьовує, та змінивши їх формат. До того ж, реалізація сокетів, надана фреймворком Qt накладає декілька важливих обмежень на роботу з мережею – нові з’єднання повинні оброблятись лише у тому ж потоці, який створив об’єкт веб-серверу, що суттєво обмежує можливості роботи із ним. Перехід на реалізацію сокетів на основі системних викликів Linux може теж покращити швидкодію веб-серверу.

2.5 Вдосконалення багатопотокової моделі

Очевидно, що попередня реалізація має багато потенційних варіантів удосконалення. В першу чергу було вирішено сфокусуватись на роботі із файловою системою, і змінити асинхронний доступ до неї на синхронний. Для взаємодії із асинхронною файловою системою було вирішено застосувати механізми Future та Promise.

2.5.1. Реалізація файлової системи на основі std::future

Для першої ітерації покращення файлової системи було вирішено позбутись емуляції файлової системи на основі пулу потоків, і надати механізм справжнього асинхронного доступу на основі Future. Механізм Future дозволяє зручно розділити ті ділянки коду, які виконують асинхронні обчислення, та ті, що обробляють результати їх обчислень, що дозволить вдосконалити архітектуру проєкту. Задля зменшення зв’язаності коду, механізм роботи із

файловою системою було вирішено винести в окремий абстрактний інтерфейс для класу-обгортки над нею.

```
class IAsyncReadFilesystem
{
private:
    virtual std::future<std::string> doRead(const char * const path)
        = 0;

public:
    std::future<std::string> read(const char * const path);
    virtual ~IAsyncReadFilesystem();
};
```

Приклад 2.1 – Абстрактний інтерфейс IAsyncReadFilesystem

Цей інтерфейс абстрагує операцію асинхронного читання із файлової системи. Імплементація методу read() має приймати на вхід рядок із шляхом до файлу та повертати об’єкт класу std::future, що буде містити зміст файлу у вигляді рядка, коли операція читання буде виконаною. Оскільки на основі Future проблематично побудувати часткову передачу файлу між потоками, в цій імплементації було вирішено передавати файли цілими. Це також дозволяє спростити інтерфейс та реалізацію файлової системи за рахунок відсутності необхідності наперед дізнатись розмір файлу. Сам інтерфейс побудований із використанням ідіоми NVI (non-virtual interface, англ. не віртуальний інтерфейс), яка полегшує відлагодження та розуміння коду.

Оскільки в межах цієї роботи розглядається лише робота асинхронної файлової системи в ОС Linux, було б корисно забезпечити принаймні працездатність програми і в інших операційних системах. Для цього, перед початком роботи із асинхронною файловою системою, на цей інтерфейс було перенесено минулорічний крос-платформений механізм синхронної роботи із файловою системою на основі патерну «Пул Потоків». Клас із цією

імплементацією був названий FakeAsyncReadFilesystem, вказуючи на те, що асинхронний доступ не є справжнім, а емулюється.

Наступною була розроблена вже справжня асинхронна реалізація файлової системи. Оскільки вона є специфічною для операційної системи Linux, її клас був названий LinuxAsyncReadFilesystem. Для виконання асинхронного доступу в цій реалізації було використано POSIX-сумісний набір системних викликів. Механізм роботи цієї імплементації показаний на Рисунку 2.3:

- Спершу викликається метод read() зі шляхом до потрібного файлу. Всередині обгортки створюється об’єкт класу std::promise, який буде використано для передачі результату виконання. Метод повертає об’єкт класу std::future, який необхідно зберегти
- Потім, відповідний екземпляр LinuxAsyncReadFilesystem за допомогою системного виклику надсилає запит до файлової системи на читання першого чанку файлу. Потім клас-обгортка стає в очікування на виконання всіх надісланих запитів.
- Щойно виконується необхідний запит, отриманий вміст файлу зберігається у тимчасовий буфер.
- Перевіряється, чи було досягнуто кінець файлу:
    - Якщо кінця не досягнуто, відправляється ще один запит на отримання наступного чанку файлу.
    - Якщо кінець файлу було досягнуто, тимчасовий буфер конвертується у рядок, який за допомогою Promise передається отримувачу

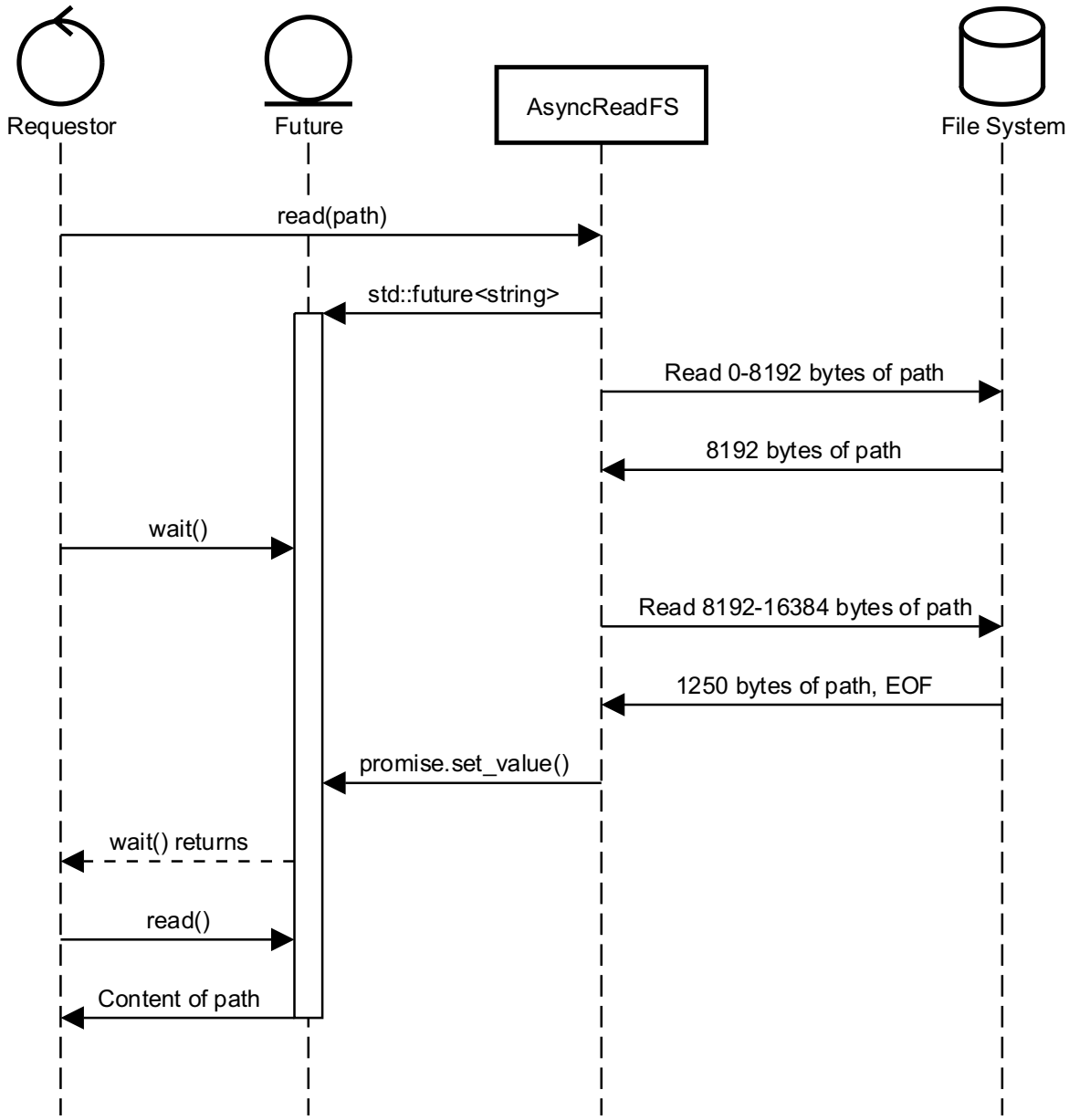

Рисунок 2.3 – Діаграма послідовності роботи класу LinuxAsyncReadFilesystem

Така обгортка над файловою системою потребує можливості тимчасово зберігати вміст прочитаних файлів, аби об’єднати результати усіх запитів та надіслати отримувачу вже цілий файл. Також необхідно зберігати Promise, які будуть використані для відправки даних. Для цього, у класі асинхронної файлової системи визначено дві мапи, які зберігають відповідні об’єкти (класів std::stringstream для збереження файлів та std::promise<string>) за ідентифікатором файлу, який повретає системний виклик open().

Оскільки в нас є дві реалізації файлової системи (блокуюча кросплатформена і асинхронна для Linux), варто було б надати користувачам програми механізм автоматичного створення необхідної обгортки, аби програма працювала на усіх підтримуваних системах. Для цього, за взірцем «Абстрактна Фабрика» [2], було створено клас MultiplatformAsyncReadFilesystemFactory. Цей клас, на основі макросів та констант компіляції, створює екземпляр необхідної файлової системи.

```
MultiPlatformAsyncReadFilesystemFactory::MultiPlatformAsyncReadFiles
      ystemFactory()
    :
      _conditionVariable(std::make_shared<std::condition_variable>())
{
    #ifdef __linux
    _filesystem =
      std::make_shared<LinuxAsyncReadFilesystem>(_conditionVariable);
    #elif
    _filesystem =
      std::make_shared<FakeAsyncReadFilesystem>(_conditionVariable);
    #endif
}
```

Приклад 2.2 – Конструктор класу абстрактної фабрики файлової системи

За допомогою цього класу та узагальненої імплементації взірця «Синглтон», розглянутої у минулорічній роботі, розробники мають змогу працювати із файловою системою на основі одного інтерфейсу, незалежно від цільової операційної системи.

2.5.1.1 Обмеження використання std::future та std::promise для навантажених систем

Хоча такий інтерфейс обгортки над файловою системою непогано використовує механізм асинхронної роботи із нею та надає відповідний інтерфейс, у такої імплементації є декілька суттєвих недоліків:

1. Файл так само зберігається в пам’яті цілим, а отже обмеження, пов’язані із виділенням пам’ятті, залишаються
2. У разі необхідності одночасного читання декількох файлів, постає питання одночасного очікування на декілька Future, що додатково ускладнює роботу із файловою системою

Теоретично, для очікування на кілька Future нам би підійшов запропонований стандартний алгоритм std::experimental::when_any(). Він надає можливість заблокувати виконання потоку доти, доки принаймні один із декількох наданих Future не стане доступним до читання. Це дозволяє обробити отримані дані, відправити їх користувачу, та знову стати в очікування на результат решти Future. Проте, як було зазначено в розділі 1.3, цю пропозицію, на жаль, було відхилено, і відповідно, експериментальна імплементація цього алгоритму більше недоступна для використання. Отже, механізм очікування на кілька Future необхідно розробляти самостійно.

Для такого механізму в цій роботі було вирішено застосувати умовні змінні, адже вони саме пристосовані до очікування на настання певних подій, в нашому разі – отримання вмісту файлу одним із Future. Цей механізм працює так:

- Об’єкт обгортки над файловою системою ініціалізується об’єктом умовної змінної, яка буде використовуватись для синхронізації
- Потік, що виконує запити до файлової системи, зберігає всі отримані від файлової системи Future
- Коли настає час стати на очікування файлів, цей потік стає в очікування умовної змінної, викликаючи її метод wait()
- Коли вміст будь-якого із файлів був прочитаний об’єктом-обгорткою та збережений у відповідний Future, в умовної змінної викликається метод notify_all(), який пробуджує усі очікуючі потоки
- Очікуючий потік перевіряє усі наявні Future, та отримує дані із тих, що мають їх, та стає в очікування знову, поки усі Future не будуть виконані

Такий механізм справді дозволяє встати на очікування всіх необхідних Future. Проте, через часте перемикання контексту, додаткові витрати на таку синхронізацію є суттєвими та можуть перевищити виграш від використання асинхронного доступу до файлової системи.

2.5.2. Тестування швидкодії

Після початкового тестування було виявлено, що така реалізація суттєво програє навіть минулорічній – для 10 одночасних з'єднань та файлу розміром 4 КБ, сервер зміг обробляти всього 451 запит на секунду (для довідки, минулорічна реалізація в цих умовах обробляла більше дванадцяти тисяч запитів).

Це сталось через декілька факторів, основними з яких є необхідність виділення значних об'ємів оперативної пам'яті та значний об'єм додаткових перевірок та очікувань. Так, через відсутність стандартного механізму очікування на декілька Future, програмі доводиться кожного разу, коли справджується один із них, перевіряти усі наявні об'єкти на наявність даних. Те саме відбувається і на рівні файлової системи – кожного разу необхідно за допомогою системного виклику перевірити, чи виконався кожен запит до файлової системи, що значно уповільнює роботу із нею. А додаткові витрати на перемикання контексту через використання умовних змінних ще більше ускладнюють ситуацію.

Отже, якщо ми хочемо побудувати справді ефективну реалізацію веб-серверу, нам необхідно повністю переробити існуючу реалізацію, а саме:

- Використати новий ефективніший механізм передачі даних між потоками
- Повністю позбутись необхідності виділяти багато оперативної пам'ті
- Перейти на Linux-специфічну реалізацію роботи із асинхронною файловою системою, адже вона пропонує кращий інтерфейс роботи без додаткових системних викликів

2.5.3. Патерн «Проактор»

Для вдосконалення існуючої реалізації, механізм передачі даних має накладати якомога менше додаткових витрат на синхронізацію та виділення пам’яті. Отже, необхідно зменшити кількість проміжних етапів при передачі даних від файлової системи до клієнта та зменшити кількість перемикань контексту. Саме для цього може стати в нагоді вдосконалена версія патерну «Реактор» – взірець «Проактор». [6]

В основі цього взірця лежить перерозподіл відповідальності між учасниками Реактору та спрощення роботи із асинхронними ресурсами, такими як файлова система. Основними елементами цього взірця є:

- Проактивний ініціатор – потік або сутність у програмі, яка ініціює виконання асинхронної операції
- Виконавець асинхронних операцій – компонент, що надає можливість виконувати асинхронні операції
- Обробник виконання – компонент, що оброблює виконання асинхронної операції, зазвичай – зворотній виклик
- Диспетчер виконання – компонент, що викликає відповідний обробник виконання, коли асинхронна операція виконується

За рахунок розділення диспетчера виконання та проактивного ініціатора асинхронних операцій, ці компоненти працюють окремо один від одного та не потребують перемикання контексту або синхронізації між собою. А представлення обробника виконання у вигляді зворотного виклику також прибирає необхідність ще одного перемикання між потоками для обробки події. Варто зазначити, що для досягнення гарних результатів обробник виконання має виконувати мінімальний об’єм обчислень та не блокувати диспетчер виконання.

Отже, цей взірець відповідає нашим умовам щодо вдосконалення імплементації веб-серверу та може бути застосований для її покращення.

2.5.4. Реалізація файлової системи на основі зворотних викликів

Для запобігання збільшення зв’язаності коду, реалізацію обгортки над файловою системою також було вирішено винести в окремий клас та визначити абстрактний інтерфейс для його потенційної заміни. Такий інтерфейс має забезпечити передачу даних між компонентами веб-серверу з мінімальними витратами оперативної пам’яті та надати можливість наперед дізнаватись розмір файлу, аби забезпечити коректну імплементацію протоколу HTTP. Інтерфейс асинхронної файлової системи на основі зворотних викликів, що відповідає цим вимогам, був названий ICallbackReadFilesystem.

```
class ICallbackReadFilesystem
{
private:
    virtual void doRead(const char * const path,
      std::function<void(std::string, bool)> fileCallback,
      std::function<void(size_t)> sizeCallback) = 0;

public:
    void read(const char * const path,
      std::function<void(std::string, bool)> fileCallback,
      std::function<void(size_t)> sizeCallback);
    virtual ~ICallbackReadFilesystem();
};
```

Приклад 2.3 – Інтерфейс ICallbackReadFilesystem

Імплементація цього інтерфейсу для операційної системи Linux була названа LinuxCallbackReadFilesystem. Метод read(), що використовується для читання файлу, тут приймає три параметри: шлях до файлу і два зворотні виклики: один для отримання вмісту файлу, та другий для отримання його розміру. Коли чанк файлу був отриманий від файлової системи, обгортка над файловою системою виконує відповідний зворотній виклик, передаючи два парамери: безпосередньо вміст чанку у вигляді рядку, та булевий прапорець

isFinal, що вказує на те, чи був досягнений кінець файлу, чи читання ще буде продовжуватись. В термінології патерну Проактор, обгортка над файловою системою – це диспетчер виконання, а зворотній виклик, що передається в метод read() – обробник виконання. Для виконання необхідних зворотних викликів, об’єкт класу LinuxCallbackReadFilesystem містить мапу, що зберігає відповідності між ідентифікаторами файлів та відповідними зворотними викликами. А для передачі запитів на читання від Ініціатора до Диспетчера виконання використовується потокобезпечна черга запитів.

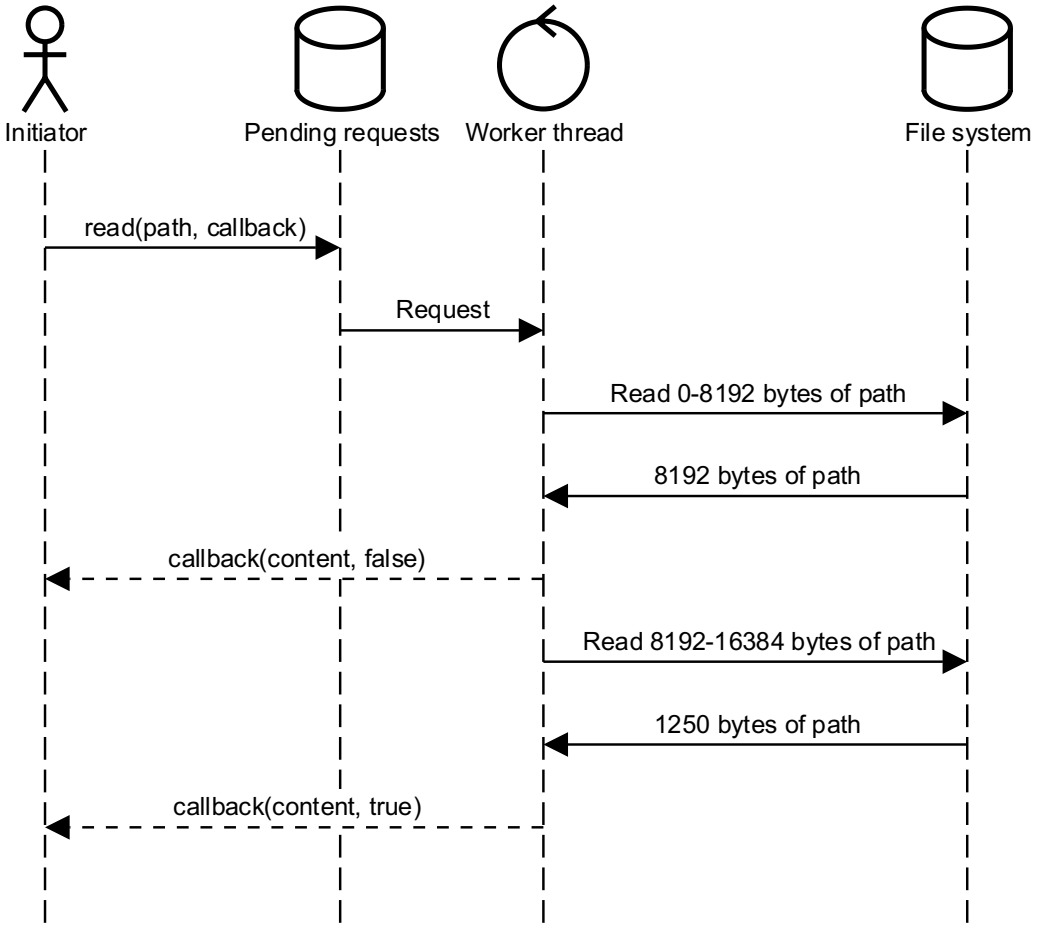

Рисунок 2.4 – Діаграма послідовності роботи файлової системи на основі зворотних викликів

Обробники виконання відповідають за створення події про читання файлу і додавання її у загальну потокобезпечну чергу подій, яка обробляється окремим

потоком разом із іншими подіями. Це дозволяє та мінімізувати навантаження на Диспетчер, так і розділити його і Ініціатора.

Така імплементація файлової системи, хоча і значно відрізняється від попередньої, була без особливих складностей інтегрована у існуючу реалізацію веб-серверу та була відлагоджена на основі реального прикладу веб-серверу.

2.5.4.1 Тестування швидкодії

Для тестування імплементації файлової системи на основі зворотніх викликів, було проведено той самий набір тестів, що і для минулорічної реалізації та веб-серверу nginx. Так, перше тестування було проведене на файлах розміром у 4 КБ, для різної кількості одночасних підключень, від одного до п’ятидесяти.

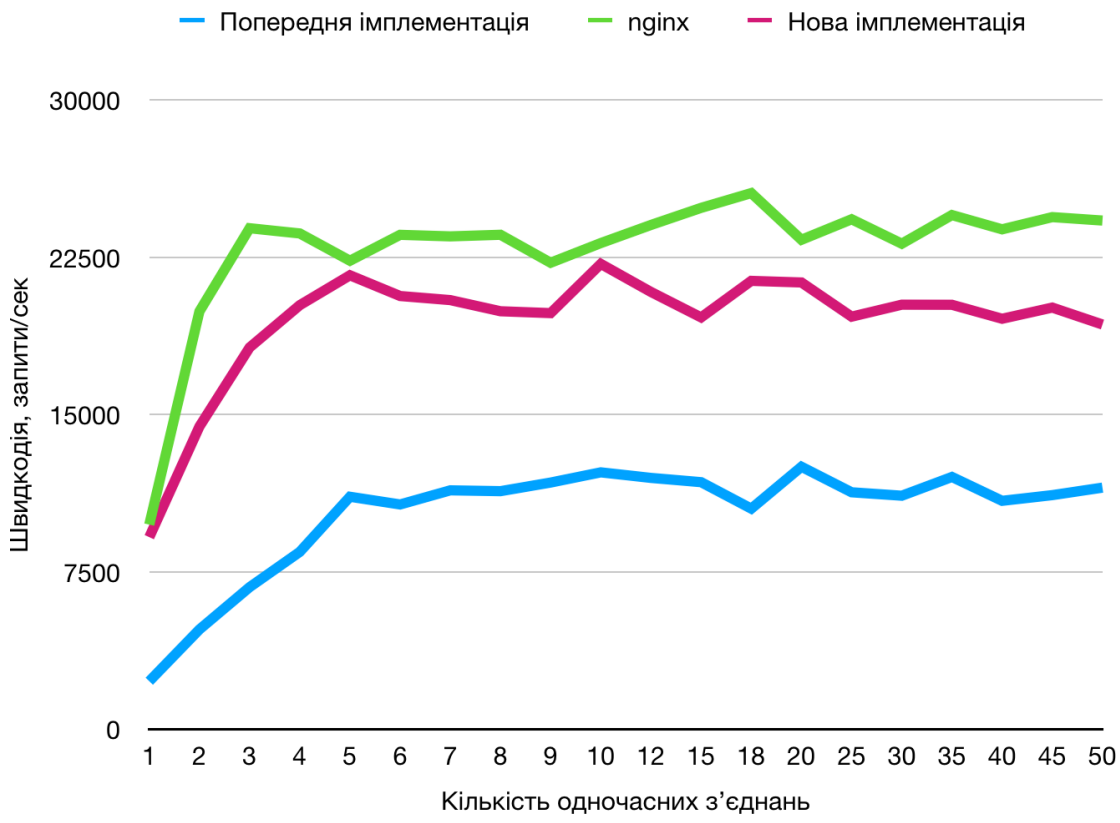

Рисунок 2.5 – Графік залежності швидкодії веб-серверу в залежності від кількості одночасних підключень

Як можна побачити із графіку, швидкодія нової імплементації, позначена рожевою лінією, значно наблизилась до еталонної, практично наздоганяючи її. Хоча зріст швидкості обробки запитів у нової імплементації дещо повільніший ніж в nginx, виходячи на плато, починаючи із п’яти одночасних підключень проти трьох у nginx, нова імплементація є значно ефективнішою для одного одночасного підключення і практично досягає швидкості nginx.

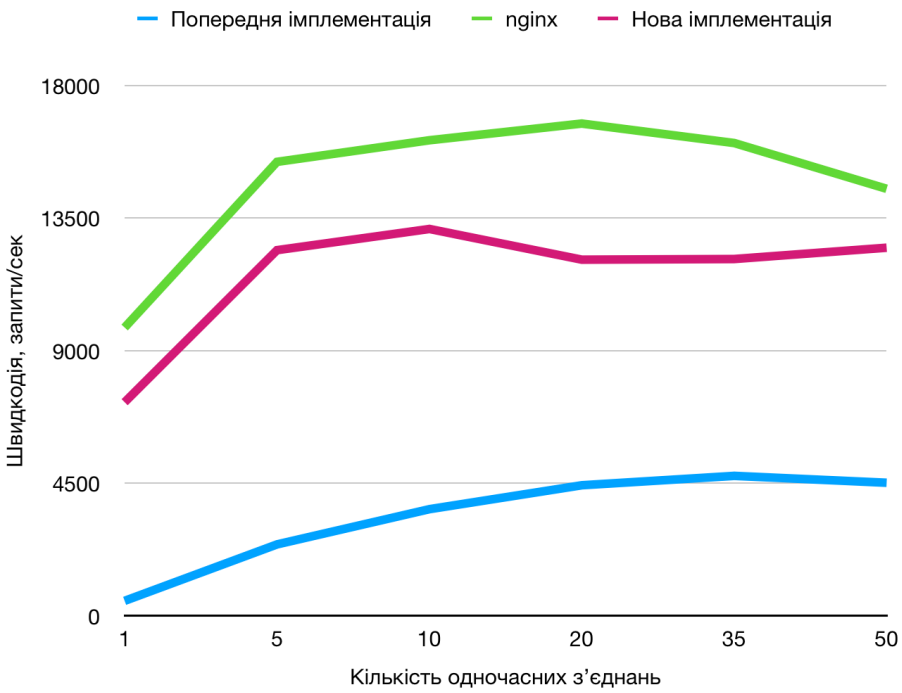

Рисунок 2.6 – Графік залежності швидкодії веб-серверу в залежності від кількості одночасних підключень для файлу розміром 25 КБ

Для файлів більшого розміру ця тенденція зберігається. За рахунок почанкового надсилання файлів клієнтам, веб-сервер тепер не обмежений максимальним обсягом пам’яті, який йому доступний для зберігання файлів, а обмежений лише загальною пропускною здатністю. Варто зазначити, що як і nginx, нова імплементація є ефективнішою для великих файлів, адже сталі накладні витрати на відкриття сокету та файлу стають меншими відносно витрат на безпосередньо читання та відправку файлу. Так, під час запитів на файл розміром 4 КБ за 20 одночасних підключень, веб-сервер відправляє 110 МБ/сек, а для файлу розміром 25 КБ – вже 804 МБ/сек.

2.6 Подальші оптимізації

Хоча результати кінцевої імплементації вже суттєво наблизились до результатів еталонної (nginx), вона має потенціал для додаткових покращень.

По-перше, може виявитись ефективним заміна крос-платформеної реалізації TCP-сокетів, наданої фреймворком Qt, на специфічної для кожної платформи. Використана реалізація сокетів містить багато нерелеватного коду, специфічного для використання із іншими компонентами фреймворку, і накладає багато додаткових обмежень на її використання в багатопотокових програмах. Заміна її на стандартну допоможе спростити програми та може позитивно вплинути на її швидкодію.

По-друге, покращити результати програми може уніфікація обробників виконання асинхронних операцій. Зараз, на кожен запит на читання, створюється окремий зворотній виклик, який до того ж зберігається в мапі в класі-обгортці до кінця читання файлу. Якщо ініціатор буде надавати один зворотній виклик, що зможе обробляти усі виконані асинхронні операції, це зменшить кількість створених зворотніх викликів та спростить код обробки запитів користувачів.

По-третє, отримана реалізація містить багато припущень про оптимальні розміри буферів читання, час очікування на події файлової системи та інші. Гранулярна оптимізація цих значень теж може вплинути на швидкодію та покращити результати веб-серверу.

ВИСНОВКИ

В першу чергу, була виконана спроба застосувати механізм Future для передачі результатів асинхронних операцій. Проте, як виявилось, цей механізм погано підходить для високонавантажених програм, які можуть виконувати велику кількість одночасних запитів до файлової системи. Через відсутність ефективного механізму очікування на одразу декілька Future, який на даний момент існує лише у вигляді пропозиції до стандарту, область застосування цього та інших дотичних механізмів суттєво звужується.

Швидкодію цієї імплементації було виміряно, та після отримання незадовільних результатів, які були гірші навіть за минулорічні, було визначено вузькі місця існуючої реалізації та обраних механізмів синхронізації, і розроблено план наступних покращень. Для другої спроби було обрано патерн «Проактор», як вдосконалення патерну «Реактор», пристосованого до справді асинхронних операцій.

Для реалізації моделі цього взірця, було імплементовано клас-обгортку для роботи із файловою системою на основі зворотних викликів. Механізм зворотніх викликів дозволив суттєво зменшити кількість перемикань контексту між потоками при передачі даних, а також, на противагу Future, дозволив повертати на один запит на читання одразу декілька результатів, що є абсолютно необхідним для організації почанкового доступу та відправки файлів користувачам.

Тестування швидкодії цієї імплементації показало гарні результати – вона практично наздоганяє швидкодію дуже ефективного веб-серверу nginx, та значно переважає минулу реалізацію, і зі збільшенням розмірів передаваних файлів цей розрив дедалі збільшується. Ці результати наочно демонструють переваги «Проактору» над «Реактором» та його кращу придатність до роботи із масовими асинхронними запитами. Також, не зважаючи на гарні результати, були виділені потенційні вузькі місця та можливості до подальшого покращення цієї моделі.

Результати цієї роботи можуть бути використані для розробки та оптимізації застосувань, що значно опираються на асинхронні операції, такі як мережеві запити, доступ до файлової системи та інші. Описані у цій роботі спроби побудови ефективного механізму роботи із асинхронною файловою системою можуть стати у нагоді розробникам та допомогти із вибором ефективних механізмів синхронізації.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. C++ Concurrency in Action, Second Edition / Anthony Williams – Shelter Island, NY, Manning Publications Co.: 2019
2. Design Patterns: Elements of Reusable Object-Oriented Software / Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides – Boston, Addison-Wesley: 1994
3. Reactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events / Douglas C. Schmidt – St. Louis, MO: Washington University, Department of Computer Science, 1995
4. A Unified Executors Proposal for C++ | P0443R12 / Jared Hoberock та інші: SG1 – Concurrency and Parallelism, 2020
5. Proactor – An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events – Irfan Pyarali, Tim Harrison, Douglas C. Schmidt – St. Louis, MO: Washington University, Department of Computer Science, 1997
6. Linux System Programming, First Edition / Robert Love – Sebastopol, CA, O'Reilly Media, 2007
7. The Free Lunch is Over / Herb Sutter – Dr. Dobb's Journal, 30(3): 2005
8. Thread Pool Design Pattern / Pete Laker, Benoit Jester: Microsoft TechNet, 2012

ДОДАТОК А

(довідниковий)

UML-діаграма класів, що відповідають за роботу із файловою системою