Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

Кваліфікаційна робота

освітній ступінь - бакалавр

на тему: **«ВПРОВАДЖЕННЯ ПАТЕРНУ СПІВПРОГРАМ ДЛЯ ПРОЕКТУВАННЯ ЕФЕКТИВНИХ ВЕБ-СЕРВЕРІВ»**

Виконав студент 4-го року
навчання спеціальності 122
«Комп'ютерні науки»
Цегельник Богдан Володимирович
Керівник: Бублик В. В.
Кандидат фіз.-мат. наук, доцент

Рецензент ______________________
(прізвище та ініціали)

Кваліфікаційна робота захищена
з оцінкою ______________________
Секретар ЕК ___________________
«____» _________________ 2024 р.

Київ 2024

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ
Викладач кафедри мультимедійних систем,
Кандидат фіз.-мат. наук, доцент
________ Бублик В.В.
„____” _____________ 2023р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на кваліфікаційну роботу
студенту Цегельнику Богдану Володимировичу
факультету інформатики 4 курсу бакалаврської програми

Тема: Впровадження патерну співпрограм для проектування ефективних веб-серверів
Вихідні дані:

- Веб-сервер з використанням співпрограм мовою С++
- Порівняльна характеристика ефективності сервера відносно інших реалізацій (синхронний, асинхронний із зворотними викликами)

Зміст ТЧ до кваліфікаційної роботи:

Зміст
Анотація
Вступ
1 Механізми взаємодії програмних елементів
2 Дослідження реалізації співпрограм в мові програмування С++
3 Проектування і тестування веб-сервера
Висновки
Список літератури
Додатки (за необхідністю)

Дата видачі „___” _________ 2023 р. Керівник__________________
(підпис)

Завдання отримав __________
(підпис)

Тема: Впровадження патерну співпрограм для проектування ефективних веб-серверів

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на кваліфікаційну роботу.	13.11.2023	
2.	Огляд літератури за темою роботи	16.02.2024	
3.	Написання програмного коду для проведення досліджень.	20.03.2024	
4.	Написання текстової частини.	26.04.2024	
5.	Надання роботи керівнику для перевірки	20.05.2024	
6.	Корегування роботи за результатами перевірки керівником	22.05.2024	
7.	Оформлення слайдів для доповіді.	23.05.2024	
8.	Захист роботи.	30.05.2024	

Студент Цегельник Б. В.

Керівник Бублик В. В.

«______»_______________

Зміст

Анотація 6

ВСТУП 7

РОЗДІЛ 1. МЕХАНІЗМИ ВЗАЄМОДІЇ ПРОГРАМНИХ ЕЛЕМЕНТІВ 9

1.1 Концепція співпрограм 9

1.2 Класифікація співпрограм 10

1.2.1 Механізм передачі керування 10

1.2.2 Об'єкт чи мовна конструкція 12

1.2.3 Збереження стеку 13

1.3 Відмінності співпрограм від потоків 14

1.4 Синхронні та асинхронні операції введення-виведення 17

1.5 Співпрограми як заміна зворотних викликів 18

1.6 Висновки до розділу 1 22

РОЗДІЛ 2. ДОСЛІДЖЕННЯ РЕАЛІЗАЦІЇ СПІВПРОГРАМ В МОВІ ПРОГРАМУВАННЯ C++ 24

2.1 Ключові компоненти співпрограм у C++20 24

2.1.1 Очікуваний об'єкт (awaitable) 26

2.1.2 Обробник співпрограми 27

2.1.3 Об'єкти обіцянки (promise) типу promise_type 28

2.2 Трансформація співпрограм компілятором 30

2.3 Співпрограми бібліотеки Boost Asio 32

2.4 Ефективність asio::awaitable в порівнянні з потоками виконання 33

2.5 Висновки до розділу 2 35

РОЗДІЛ 3. ПРОЕКТУВАННЯ І ТЕСТУВАННЯ ВЕБ-СЕРВЕРА 37

3.1 Підходи до проектування веб-сервера 37

3.2 Загальна архітектура 37

3.3 Синхронний сервер 40

3.4 Асинхронний сервер із зворотними викликами 42

3.5 Асинхронний сервер із співпрограмами 45

3.6 Тестування ефективності 47

3.7 Висновки до розділу 3 52

Висновки 53

Список літератури 55

Додаток А. Вихідний код програми для тестування максимальної кількості потоків 57

Додаток Б. Вихідний код для порівняння роботи asio::awaitable та потоків виконання 58

Додаток В. Вихідний код підрахунку середнього часу виконання 60

Анотація

У цій роботі досліджується використання співпрограм (корутин), як патерну проектування, для підвищення ефективності роботи веб-серверів. Робота аналізує теоретичні відомості про співпрограми та демонструє їх практичне застосування при розробці веб-серверів з великою кількістю асинхронних операцій. Основна увага приділена розробці веб-серверу мовою програмування C++ із застосуванням можливостей співпрограм стандарту C++20, а також порівнянню з альтернативними підходами до розробки.

Ключові слова:

Співпрограми, корутини, веб-сервери, асинхронне програмування, C++20

ВСТУП

Із зростанням вимог до обробки великих обсягів даних та збільшенням кількості користувачів, запити яких потрібно обробляти, постала необхідність в пошуку нових підходів до проектування та розробки веб-серверів.

Одним з перспективних напрямків у цьому є використання співпрограм. Вони дозволяють підвищити ефективність програмного коду за рахунок нелінійної організації виконання задач для уникнення простою системи під час очікування завершення операцій, незалежних від ресурсів процесора. Це особливо важливо для веб-серверів, де часто виникають такі операції, зокрема читання ти запис вхідних-вихідних запитів або ж звернення до баз даних.

Популярні мови програмування такі як Python, C++, Java, JavaScript, C#, Kotlin можуть використовувати співпрограми на рівні бібліотеки або мови. Підтримка співпрограм на рівні мови в C++ з’явилася не так давно, разом із стандартом мови C++20, що свідчить про зростання цікавості до співпрограм.

Виходячи з тенденцій, за **мету даної роботи** було поставлено дослідити поняття співпрограм як патерну багатозадачності, їх реалізацію в стандарті C++20 та переваги застосування на прикладі проектування веб-серверів, провести аналіз ефективності роботи сервера.

Завдання, які ставляться у даній роботі, включають:

1. Розглянути співпрограми як патерн багатозадачності, їх класифікацію, відмінності від потоків та переваги використання з асинхронними I/O операціями.
2. Дослідити співпрограми в стандарті C++20 та ефективність їх реалізації в бібліотеці Boost Asio в порівнянні з потоками.
3. Реалізувати веб-сервери з використанням різних підходів: синхронного, асинхронного із зворотними викликами та асинхронного із співпрограмами.
4. Провести тестування та порівняти ефективність роботи й складності в реалізації кожного підходу.

Робота складається з трьох розділів.

Перший розділ присвячено теоретичному вивченню співпрограм, їхнім основним характеристикам та класифікації. Порівнюються співпрограми з традиційними потоками, розглядаються переваги та недоліки синхронних та асинхронних операцій введення-виведення.

Другий розділ охоплює аналіз ключових компонентів співпрограм у C++20 та їхнє використання у бібліотеці Boost Asio. Розглядаються очікувані об'єкти (awaitable), обробники співпрограм та promise_type об'єкти.

Третій розділ зосереджений на експериментальному дослідженні ефективності різних підходів до проектування веб-серверів, включаючи синхронний підхід, асинхронний підхід із зворотними викликами та асинхронний підхід із співпрограмами. Проведено порівняльний аналіз продуктивності та зроблено висновки щодо ефективності кожного підходу.

РОЗДІЛ 1. МЕХАНІЗМИ ВЗАЄМОДІЇ ПРОГРАМНИХ ЕЛЕМЕНТІВ

1.1 Концепція співпрограм

Співпрограми (альтернативна назва - корутини), як концепція керування виконанням, були представлені ще в 1960-х роках. Проте, попри свій вік, підтримка співпрограм у сучасних високорівневих мовах програмування з'явилася досить пізно. Це зумовлено зокрема відсутністю точного визначення і єдиного погляду на термін співпрограми, а також наявністю більш зрозумілих механізмів, таких як потоки, для одночасного виконання декількох задач. Однак останнім часом популярні високорівневі мови, такі як Python, C++, JavaScript, C#, Kotlin, і Go, все ж таки впровадили підтримку співпрограм, дозволяючи більш ефективно писати асинхронний код.

Можна виділити дві фундаментальні властивості, які притаманні всім означенням співпрограми й відрізняють її від інших механізмів. Співпрограмою називається компонент програми, що задовольняє такі властивості [1]:

- Може зупинитися і передати керування іншій процедурі або співпрограмі, щоб в майбутньому відновити виконання з місця зупинки.
- Зберігає стан змінних між викликами.

На рисунку 1.1 можна побачити схему поведінки співпрограми та її відмінності в порівнянні з процедурою (функцією).

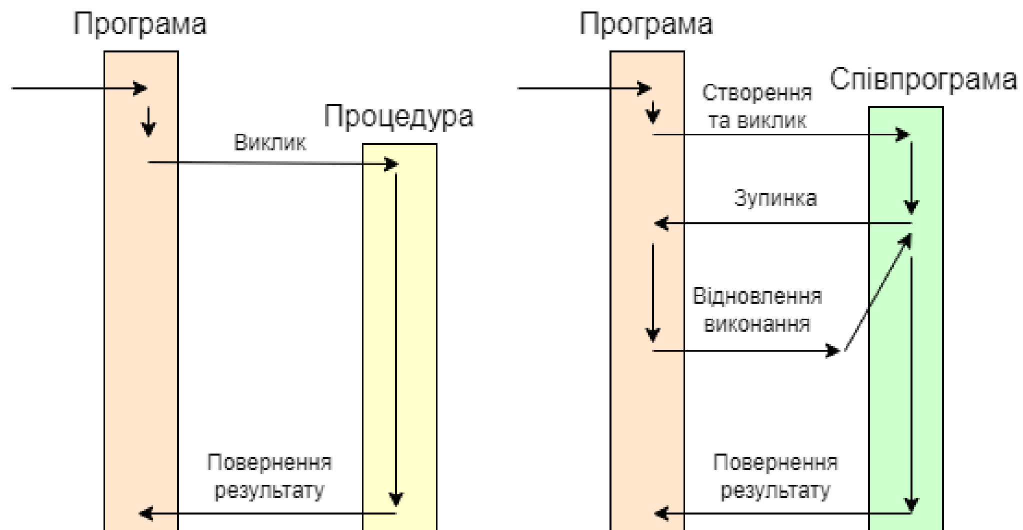

Рисунок 1.1 - Різниця між процедурою та співпрограмою

Окрім цього існують й інші властивості за якими класифікують співпрограми. Вони і вносять складність в однозначне тлумачення та реалізацію у високорівневих мовах програмування.

1.2 Класифікація співпрограм

Розглядаючи класифікацію співпрограм (корутин), можна виділити три основні властивості [1]:

- Механізм передачі керування: симетричні та асиметричні.
- Об’єкт, яким можна маніпулювати, чи мовна конструкція.
- Збереження стеку для зупинки у вкладених викликах.

Комбінації цих властивостей визначають можливості корутин, їх використання та потребу в системних ресурсах.

1.2.1 Механізм передачі керування

Найвиразнішою відмінністю в реалізації співпрограм є механізм передачі керування виконанням. За цією властивістю їх розділяють на симетричні та асиметричні [2].

Симетричні корутини мають одну операцію - відновлення виконання. Співпрограма зупиняється, зберігає стан та відновлює роботу іншої співпрограми, а та в свою чергу може зробити те саме.

На рисунку 1.2 можна побачити приклад роботи симетричних корутин. Перша корутина зупиняється, зберігає стан змінних й відновлює виконання другої. Друга виконує обрахунки та передає виконання третій. Третя робить те саме й відновлює виконання першої.

Ця поведінка нагадує оператор goto, який є в більшості мов програмування, та рекомендується не використовувати. Проте, якщо правильно використовувати такі операції з корутинами, це навпаки може зробити код більш структурованим та зрозумілішим [2].

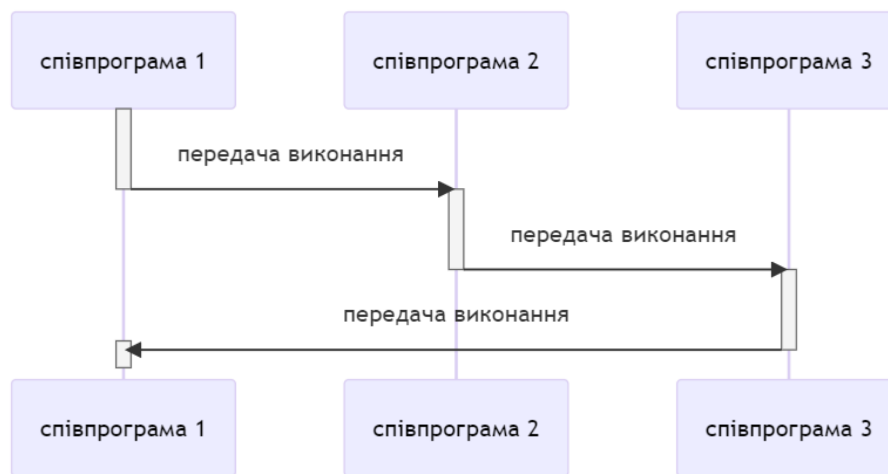

Рисунок 1.2 – Схема поведінки симетричних корутин

Іншою реалізацією є асиметричні корутини, які мають дві операції передачі керування. Одну для виклику та відновлення виконання, іншу для призупинення та передачі керування назад тому, хто її викликав. На рисунку 1.3 можна побачити схему такої роботи, яка більше схожа на поведінку процедур.

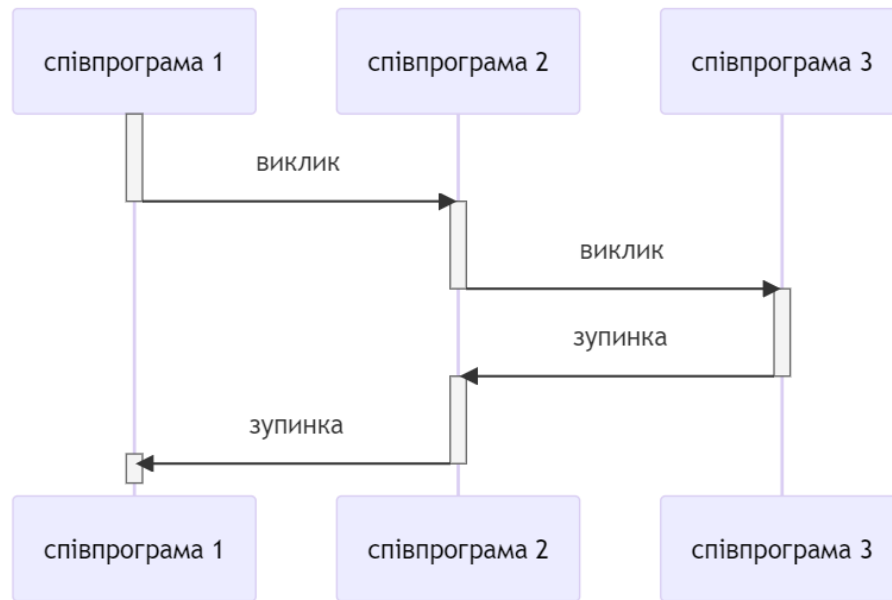

Рисунок 1.3 – Схема поведінки асиметричних корутин

Наявність двох видів механізму передачі управління зумовлено проблемами, для вирішення яких краще підходить та чи інша реалізація. Симетричні корутини зручніші для реалізації багатозадачності, де кожна корутина є незалежною обчислювальною задачею та існує планувальник, що керує їхнім виконанням. З іншого боку асиметричні можуть бути корисними, коли потрібно згенерувати або надати якесь проміжне значення. Наприклад, реалізувати генератор чисел або обійти вміст структури даних [2].

1.2.2 Об’єкт чи мовна конструкція

Другою характеристикою є реалізація корутини як об’єкта чи як конструкції у синтаксисі мови програмування.

Корутина як об’єкт означає рівні можливості в порівнянні з об’єктами інших типів. Проте варто розглядати сутність об’єкта з прив’язкою до конкретної мови програмування [2].

Така реалізація надає розробнику більшу гнучкість, оскільки існує можливість, наприклад, використовувати принципи об’єктно-орієнтованого

програмування: наслідування, інкапсуляція, поліморфізм, або ж передавати співпрограми як параметри.

В іншому випадку співпрограма може бути визначена лише в межах деякої мовної конструкції, що часто має певні обмеження у використанні для розробника.

1.2.3 Збереження стеку

Останньою, не менш важливою характеристикою є збереження кадрів стеку. Корутина може зберігати декілька кадрів стеку, що дозволяє зупинятися та продовжувати виконання у вкладених викликах. Таку співпрограму будемо називати стековою. Протилежною до неї є безстекова, що зберігає лише останній кадр стеку, тоді виконання повертається до попереднього рівня вкладеності і потребує складнішої каскадної обробки зупинки та відновлення [2].

На рисунку 1.4 зображена схема роботи стекової корутини із збереженням декількох кадрів стеку. Після відновлення співпрограми, виконання продовжиться всередині вкладеного виклику.

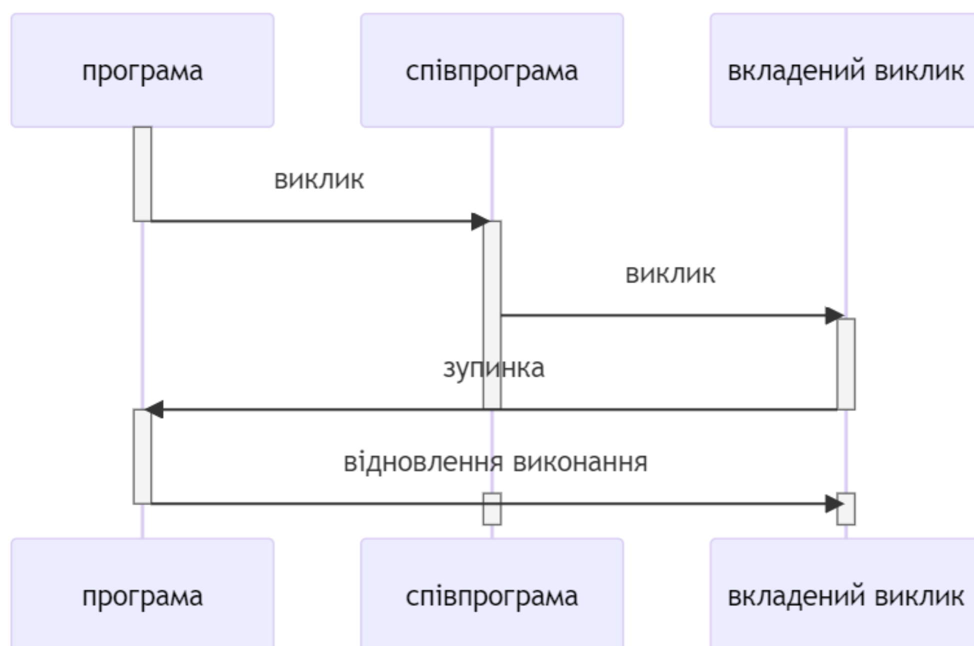

Рисунок 1.4 – Схема поведінки стекової корутини

На рисунку 1.5 зображена схема роботи із збереженням лише верхнього рівня стеку. Під час зупинки вкладений виклик повертає виконання співпрограмі, яка його викликала. Та в свою чергу має продовжити виконання або зупинитися й передати його ще на один рівень далі. Під час відновлення роботи співпрограма має відновити виконання вкладеного виклику, якщо це потрібно.

Збереження всього стеку є більш потужною властивістю, проте потребує більше системних ресурсів, зокрема пам’яті, для його збереження. Збереження лише останнього кадру стеку потребує менше пам’яті та краще підходить для пристроїв з обмеженою кількістю ресурсів.

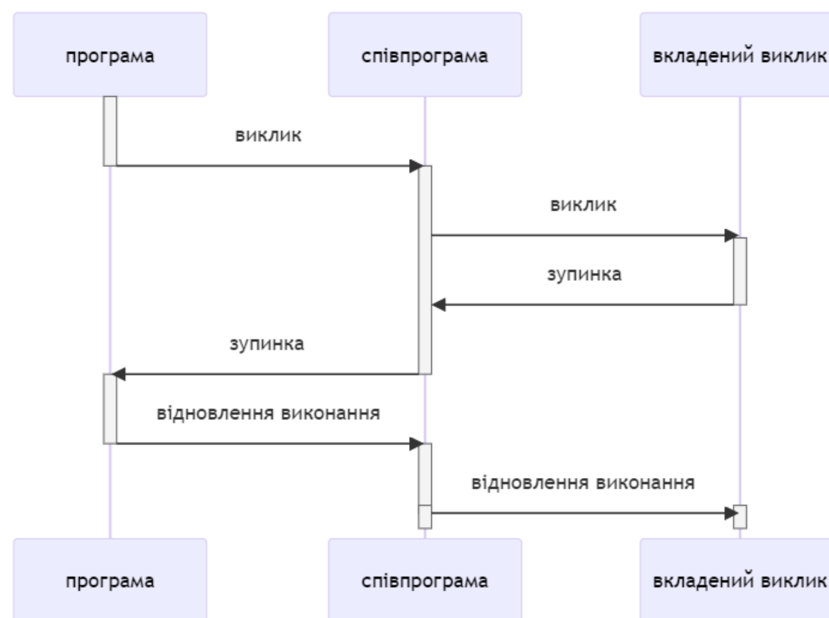

Рисунок 1.5 – Схема поведінки безстекової корутини

1.3 Відмінності співпрограм від потоків

Співпрограми часто розглядають як альтернативу потокам виконання, проте ці механізми мають певні відмінності.

Потоки існують всередині процесів операційної системи, як менші одиниці виконання та поділяють між собою один адресний простір. Процес завжди має принаймні один потік. Декілька потоків дозволяють одночасно

виконувати декілька задач. Таке призначення схоже із співпрограми, проте плануванням виконання потоків займається операційна система, в той час як за обробку зупинки та відновлення роботи корутин відповідає розробник[3].

Іншою відмінністю є можливість паралельного виконання. В системі з одним фізичним ядром потоки почергово перемикаються, так що в момент часу операцію виконує лише один потік, як це роблять і корутини. Така операція називається зміною контексту. Проте в багатоядерних системах, вони можуть виконуватися паралельно. На рисунку 1.6 можна наглядно побачити різницю між виконанням у одноядерній та багатоядерній системі. Корутини виконуються всередині потоку і за семантикою знаходяться ближче до процедур, тому паралельне виконання не передбачають.

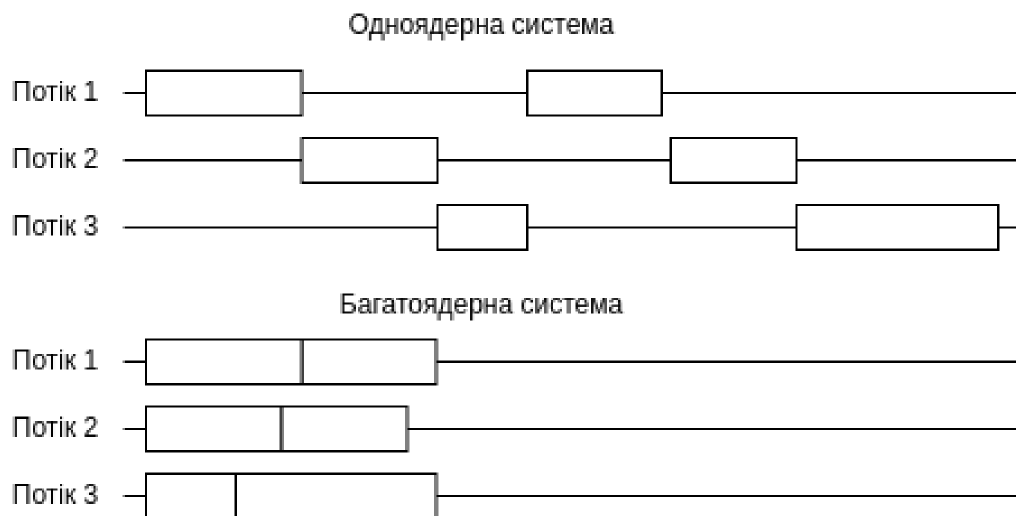

Рисунок 1.6 – Виконання потоків в різних системах

Співпрограми можуть виконуватися в одному потоці виконання. В цьому випадку однією з їх переваг є відсутність необхідності синхронізації спільних ресурсів, оскільки існує гарантія, що в певний момент часу виконується лише одна із них [4].

Найкращим рішенням є поєднання корутин для багатозадачності та потоків для паралельного виконання. На рисунку 1.7 можна побачити схему такого процесу. Вона є досить умовною, оскільки співпрограма не зовсім

належить потоку і в деяких реалізаціях може відновлювати виконання вже в іншому потоці. Прикладом цьому є реалізація корутин в стандарті C++20 [5].

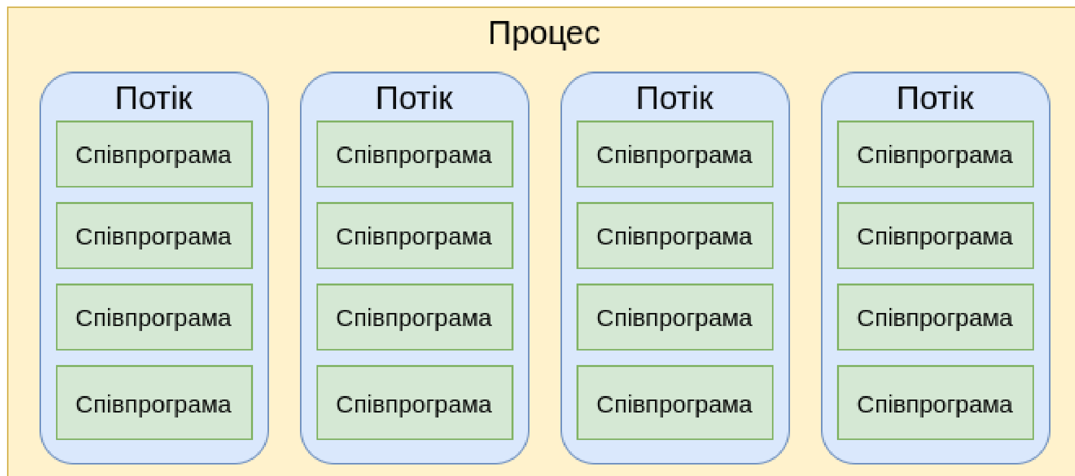

Рисунок 1.7 – Оптимальна структура програми

Останньою значною відмінністю є потреба у ресурсах, зокрема у пам’яті. Кожен потік має власний стек та набір регістрів, що заносяться до процесора кожного разу при зміні контексту [3].

Важливим моментом є розмір стеку. В операційних системах з ядром Linux максимальний розмір стеку можна дізнатися за допомогою утиліти ulimit[6]. В системі на якій проводяться досліди цей розмір становить 8 Мб, що означає: для тисячі потоків потрібно приблизно 8 Гб оперативної пам’яті. Це є досить значним розміром, проте насправді потоку виділяється віртуальна пам’ять, а споживання фізичної відбувається лише за потреби [3].

В зв’язку з цим операційна система обмежує кількість потоків, які користувач може створити. Для перевірки цього твердження написано програму мовою C++ (дивитися додаток А), яка намагається створити якнайбільше потоків виконання. Під час її запуску отримано такі повідомлення:

«Exception: Resource temporarily unavailable

Max number of threads created: 20875

terminate called without an active exception

Aborted(core dumped)»

Це означає, що тестова система не дозволила створити більше ніж двадцять тисяч вісімсот сімдесят п’ять потоків.

Співпрограмами не мають системних обмежень та не потребують створення окремого стеку та набору регістрів, тому вони використовують значно менше ресурсів і їх ініціалізація відбувається швидше. Крім цього відсутня необхідність змінювати регістри та стек у внутрішній пам’яті процесора, що робить зміну контексту швидшою, ніж це є у потоків.

1.4 Синхронні та асинхронні операції введення-виведення

Жодна реалізація сервера не може обійтися без операцій введення або виведення (I/O). Ці операції є критичними для взаємодії сервера з зовнішнім середовищем, зокрема мережею. Існують два основних типи I/O операцій: синхронні (блокуючі) та асинхронні (неблокуючі) [7].

Синхронні операції вимагають, щоб програма чекала завершення операції перед продовженням виконання наступного рядка коду. Це означає, що потік виконання блокується, поки дані не будуть повністю прочитані або записані. Такий підхід спрощує програмування, оскільки результати операцій доступні одразу після їх виконання і код програми виконується послідовно. Однак, при високих навантаженнях або тривалих операціях, це може знизити продуктивність сервера, так як потік виконання блокується і не виконує корисної роботи. Цього можна уникнути використовуючи більшу кількість потоків, проте, як було згадано раніше, їх максимальна кількість обмежена, а створення та перемикання є дороговартісними операціями. Окрім цього відсутня можливість ефективної роботи в однопотоковому середовищі.

На відміну від синхронних, асинхронні I/O операції дозволяють потоку продовжувати виконання без очікування їх завершення. Коли асинхронна операція починається, вона відразу повертає керування, що дозволяє серверу обробляти інші задачі або запити, не чекаючи завершення I/O. Після завершення, система надсилає програмі сигнал. Одним з найпоширеніших

засобів обробки сигналу є використання зворотних викликів. При отриманні сигналу потік перериває виконання поточного методу та запускає код зворотного виклику.

Асинхронне програмування є більш складним для розуміння і реалізації, а також має більші накладні витрати, оскільки вимагає додаткових механізмів для контролю стану. Через це, при відносно швидких операціях, синхронний підхід може демонструвати кращу продуктивність. Проте, в системах з великою кількістю тривалих та одночасних I/O запитів асинхронні операції показують себе краще. Тому вибір між синхронним та асинхронним введенням-виведенням значною мірою залежить від конкретних вимог та обмежень [7].

1.5 Співпрограми як заміна зворотних викликів

Співпрограми пропонують елегантне рішення проблем, які часто виникають у асинхронному програмуванні через інтенсивне використання зворотних викликів, які можуть ускладнити код, ведучи до так званого «пекла зворотних викликів», де велика кількість вкладень роблять код важким для читання та підтримки [8]. Співпрограми дозволяють писати асинхронний код, який виглядає та поводить себе як синхронний, що значно спрощує розуміння програми.

Основними недоліками зворотних викликів, які вирішуються корутинами є:

- **Складність керування:** коли асинхронні операції залежать одна від одної, код може швидко набути глибокої вкладеності і стати незрозумілим. Корутини дозволяють писати код, що виконується послідовно, навіть якщо він включає асинхронні операції. Це робить програму легшою для розуміння та тестування.
- **Ускладнення обробки помилок:** Керування помилками у вкладених викликах може бути непростим, оскільки кожний з них може мати власну логіку обробки помилок. З корутинами можна

використовувати звичайні конструкції для обробки помилок, такі як, наприклад, try/catch в мові програмування C++.

Використання співпрограм дозволяє розробникам ефективно вирішувати задачі, пов’язані з великою кількістю асинхронних викликів без використання глибокої вкладеності. Зокрема при розробці веб-сервера, асинхронні запити до баз даних або зовнішніх ресурсів можуть бути реалізовані з використанням корутин, що робить код більш зрозумілим і підтримуваним.

Розглянемо різницю на прикладі псевдокоду. Такий вигляд може мати обробка клієнтів на сервері з блокуючими синхронними операціями:

```
try
{
    socket sock = accept(sock);
    //...
    sock.read(buffer)
    //...
    connection conn = connect_database();
    //...
    conn.database_query(query);
    //...
    conn.disconnect_database();
    //...
    sock.write(buffer);
    //...
    sock.close();
}
catch (exception e)
{
    //...обробка помилок
}
```

Під час кожної операції потік виконання блокується до її завершення. Помилки можна зручно обробляти через конструкцію try/catch. Якщо переробити реалізацію через асинхронні операції із застосуванням зворотних викликів, код може мати такий вигляд:

```
async_accept((socket sock, error err){
    //...потреба обробки помилки
    //...
    sock.asycn_read(buffer, (error err){
        //...потреба обробки помилки
        //...
        async_connect_database((connection conn, error err){
            //...потреба обробки помилки
            //...
            conn.async_database_query(query, (query_result res, error err){
                //...потреба обробки помилки
                //...
                conn.disconnect_database();
                async_write(buffer, result, (error err){
                    //...потреба обробки помилки
                    //...
                    sock.close();
                });
            });
        });
    });
});
```

Цей код набагато складніше зрозуміти, до того ж потрібно обробляти помилки в декількох місцях. При збільшені кількості асинхронних операцій, наприклад, запитів до бази даних, вкладеність буде збільшуватися, що і призводить до так званого «пекла зворотних викликів». Іншим варіантом написання є поділ зворотних викликів на іменовані процедури. Це може виглядати так:

```
//глобальний/член класу buffer;
//глобальний/член класу socket;
//глобальний/член класу db_conn;

// program
{
    async_accept(async_accept_callback);
}

void async_accept_callback(socket sock, error err) {
    //...потреба обробки помилки
    //...
    sock.async_read(buffer, handleRead);
}

void handleRead(error err) {
    //...потреба обробки помилки
    //...
    async_connect_database(handleConnectDatabase);
}

void handleConnectDatabase(connection db_conn, error err) {
    //...потреба обробки помилки
    //...
    db_conn.async_database_query(query, handleDatabaseQuery);
}

void handleDatabaseQuery(query_result res, error err) {
    //...потреба обробки помилки
    //...
    db_conn.disconnect_database();
    async_write(buffer, result, handleWrite);
}

void handleWrite(error err) {
    //...потреба обробки помилки
    //...
    sock.close();
}
```

Такий підхід може бути більш зрозумілим, однак проблема обробки помилок залишається. Окрім цього, він може бути складнішим в реалізація, так як з’являється потреба збереження стану змінних, які можуть використовуватися в декількох зворотних викликах, наприклад, об’єкту socket. І все ще менш зрозуміло за синхронну реалізацію.

Співпрограми поєднують у собі переваги обох підходів. Вони дозволяють писати послідовний зрозумілий код, який буде працювати асинхронно. Код із застосуванням співпрограм може виглядати так:

```
try
{
    socket sock = co_await async_accept();
    //...
    co_await sock.async_read(buffer)
    //...
    connection conn = co_await async_connect_database();
    //...
    co_await conn.async_database_query(query);
    //...
    conn.disconnect_database();
    //...
    co_await sock.async_write(buffer);
    //...
    sock.close();
}
catch (exception e)
{
    //...обробка помилок
}
```

Як можемо побачити він майже не відрізняється від синхронного. Саме тому завдяки використанню корутин, розробники можуть підвищити читабельність та ефективність своїх серверів з великою кількістю I/O операцій, уникаючи комплексності та недоліків традиційних зворотних викликів.

1.6 Висновки до розділу 1

Перший розділ роботи зосереджений на теоретичному дослідженні співпрограм як однієї з моделей багатозадачності в теорії обчислень. Описано їхні основні характеристики та класифікація. Порівняно співпрограми з традиційними потоками. Розглянуто переваги й недоліки синхронних та асинхронних операцій введення-виведення. Виділено основні переваги співпрограм для асинхронних I/O операцій в порівнянні із зворотними викликами. Встановлено, що співпрограми забезпечують більшу ефективність

та чистоту коду, зокрема в складних програмних системах, демонструючи свою значущість як інструменти сучасного програмування.

РОЗДІЛ 2. ДОСЛІДЖЕННЯ РЕАЛІЗАЦІЇ СПІВПРОГРАМ В МОВІ ПРОГРАМУВАННЯ C++

2.1 Ключові компоненти співпрограм у C++20

Стандарт C++20 впроваджує можливість використання співпрограм, що є значним кроком вперед у розвитку можливостей асинхронного програмування в C++.

Корутини в новій версії є спеціальним видом функцій, які інакше обробляються компілятором. Вони мають асиметричний механізм передачі керування та поводяться як об’єкт. Окрім цього їх реалізація є безстековою: поточний стан зберігається в динамічно виділеній пам’яті. Це зроблено для задоволення вимог до дизайну, зокрема можливості створення тисяч чи навіть мільйонів співпрограм, що корисно для розробки веб-серверів, а також невеликі накладні витрати, які можна порівняти з викликом функції. Варто зазначити, що стандарт вводить лише каркас для створення співпрограм, а реалізацію їх роботи залишає розробнику [9].

Для розрізнення співпрограм від звичайних функцій вводяться нові ключові слова: co_await, co_yield, co_return. Без їх наявності компілятор не вважатиме код співпрограмою та не буде здійснювати ніяких трансформацій. Окрім цього, декларація співпрограми має повертати об’єкт з типом, який визначає її поведінку за допомогою реалізації класів та їх методів, оголошених стандартом. Нижче можна побачити приклад співпрограми:

```
CoroGenerator generateInts(int max) {
    for (int i = 1; i <= max; ++i) {
        co_yield i;
    }
}
```

Ключове слово co_yield дає компілятору знак, що це співпрограма, а не звичайна функція. Клас CoroGenerator визначає необхідні для її роботи методи

та дозволяє комунікувати з об'єктом стану. Такий виглядає має використання цієї корутини:

```
int main() {
    auto coro_generator = generateInts(5);
    while (coro_generator.next()) {
        std::cout << gen.value() << ' ';
    }
    return 0;
}
```

Виклик generateInts створює об'єкт стану та запускає співпрограму. Методи next() та value() дозволяють відновити виконання та дістати наступне значення відповідно. Програма буде мати такий результат:

«1 2 3 4 5 »

Як вже згадувалося раніше, стандарт надає лише каркас. Тому для роботи програмного коду наведеного вище потрібно реалізувати клас співпрограми – CoroGenerator.

Кожна співпрограма складається з трьох частин [5]:

1. **Внутрішній об'єкт обіцянки (promise)**, через який відбувається взаємодія з результатами роботи співпрограми або помилками, які виникають в процесі виконання.
2. **Обробник співпрограми**, який дозволяє керувати співпрограмою зовні, зокрема знищувати або відновлювати її виконання.
3. **Об'єкт стану**, що зберігається в динамічно виділеній пам'яті та містить в собі об'єкт обіцянки, значення скопійованих параметрів, локальних змінних та інформацію про точку зупинки.

Ще однією важливою деталлю є специфікації **очікуваного об'єкта (awaitable),** що використовується в парі з оператором co_await та керує зупинкою й відновленням співпрограми. В наступних підрозділах розглянемо кожну частину детальніше.

2.1.1 Очікуваний об'єкт (awaitable)

Для зупинки виконання співпрограми стандарт вводить унарний оператор co_await, який потрібно використовувати з виразом, що реалізує концепцію очікуваного об'єкта (awaitable). Ця концепція вимагає від об'єкту мати в наявності реалізацію таких методів [9]:

- **await_ready** – визначає чи готова співпрограма продовжити своє виконання. Якщо метод має результат true, то зупинки не відбувається, якщо false – відбувається зупинка та виклик методу await_suspend.
- **await_suspend** – приймає як параметр обробник поточної співпрограми за допомогою якого може задати умови її відновлення або знищення, наприклад, при отриманні певного сигналу. Якщо результатом методу є false або метод нічого не повертає, то зупинка відбувається. Якщо результатом є значення true, то співпрограма відразу відновлює виконання. Якщо ж метод повертає обробник іншої співпрограми, то виконання передається їй.
- **await_resume** – викликається після відновлення виконання. Результат цього методу є значенням всього виразу з co_await.

На рисунку 2.1 можна побачити діаграму поведінки під час виклику оператора co_await разом із очікуваним об'єктом.

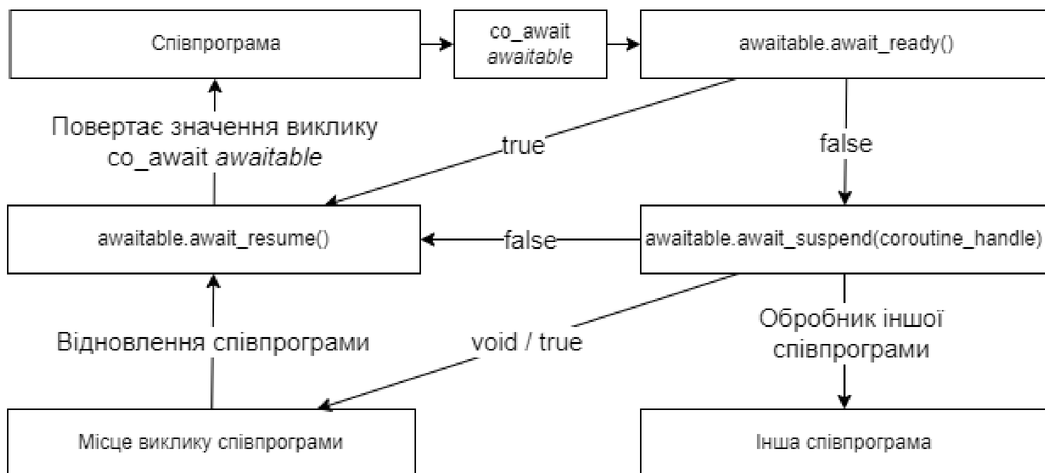

Рисунок 2.1 – Поведінка під час виклику оператора co_await

Стандартна бібліотека у версії C++20 містить дві конкретні тривіальні реалізації цієї концепції: std::suspend_never та std::suspend_always, які продовжують та зупиняють виконання співпрограми відповідно [10].

2.1.2 Обробник співпрограми

Обробник співпрограми є однією з ключових частин механізму співпрограм у C++20. Він представляє собою об’єкт за допомогою якого можна взаємодіяти із співпрограмою зовні, а саме керувати її життєвим циклом, зокрема відновлювати чи завершувати виконання.

Обробник є шаблонним класом стандартної бібліотеки – std::coroutine_handle<promise_type>, що параметризується типом обіцянки (promise) об’єкту співпрограми. Кожний обробник асоціюється з відповідним об’єктом, що створюється компілятором, та має метод promise() для доступу до нього. Окрім цього, маючи обіцянку об’єкту співпрограми, можна отримати її обробник за допомогою статичного методу coroutine_handle::from_promise(Promise &) [11]. Детальніше об’єкт обіцянки розглянемо в наступному підрозділі.

Окрім цього існує спеціалізація std::coroutine_handle<void>, яка не пов’язана з конкретним об’єктом обіцянки, та може використовуватися для загальних маніпуляцій.

Окрім вищезгаданих, обробник також має такі важливі методи [12]:

- **void resume()** – дозволяє відновити виконання співпрограми з місця її зупинки.
- **bool done()** – перевіряє чи закінчила співпрограма своє виконання.
- **void destroy()** – звільняє всі ресурси, які були виділені для співпрограми.

Прикладом використання обробника є згаданий в попередньому розділі метод await_suspend. Код наведений нижче показує його використання для

продовження роботи співпрограми після отримання сигналу про завершення асинхронної операції.

```
void Awaitable::await_suspend(std::coro_handler<> handler)
{
    async_operation([&handler](){
        handler.resume();
    })
}
```

Обробник співпрограми є ключовим інструментом для керування життєвим циклом співпрограм у C++20, забезпечуючи їх відновлення та керування ресурсами, проте він не дозволяє працювати з результатами їх роботи.

2.1.3 Об’єкти обіцянки (promise) типу promise_type

Об’єкт обіцянки (promise) є ще однією невід’ємною складовою механізму співпрограм C++20, який забезпечує зв’язок між результатами роботи співпрограми та її користувачем. Він визначає поведінку співпрограми на різних етапах її життєвого циклу та надає необхідні інтерфейси для керування її станом.

Обіцянка визначається всередині класу, що представляє об’єкт співпрограми, та за специфікацією має бути класом з назвою promise_type. Цього можна домогтися декількома шляхами, зокрема визначати вкладений клас promise_type або використати псевдонім за допомогою ключового слова using на клас, визначений в іншому місці.

Мінімально потрібний інтерфейс promise_type має такий вигляд:

```
struct promise_type
{
    CoroObject get_return_object();
    Awaitable initial_suspend();
    Awaitable final_suspend();
    void return_void();
    void unhandled_exception();
};
```

Розглянемо кожен з методів детальніше:

- **get_return_object()** – цей метод викликається при створенні співпрограми та повертає об’єкт, який буде використовуватися для взаємодії з нею.
- **initial_suspend()** – метод, який визначає, чи буде співпрограма призупинена відразу після створення. Він повертає об’єкт, який задовольняє концепцію очікуваного об’єкта (awaitable), реалізація якого і вирішує необхідність зупинки. Наприклад, при поверненні std::suspend_always, співпрограма відразу зупиняється та продовжує виконання лише за потреби. Така співпрограма називається лінивою. Протилежною до неї є активна співпрограма, що виконується відразу після створення, наприклад, при поверненні std::suspend_never з цього методу.
- **final_suspend()** – метод, схожий на попередній, проте викликається перед завершенням роботи й визначає, чи буде співпрограма призупинена перед тим як звільнити всі ресурси.
- **return_void()** – метод, який викликається при нормальному завершенні співпрограми, якщо вона не повертає ніякого значення за допомогою оператора co_return.
- **unhandled_exception()** – викликається при виникненні виключної ситуації всередині співпрограми.

Окрім цих методів, для можливості повернення даних із співпрограми за допомогою операторів co_yield та co_return потрібно визначити такі методи:

- **yield_value(ValueType value)** – викликається під час використання оператора co_yield із відповідним об’єктом типу ValueType. Вираз co_yield value є «синтаксичним цукром» та ідентичний виразу co_await promise.yield_value(value).
- **return_value(ValueType value)** – цей метод схожий до попереднього, проте виконується під час виклику виразу co_return value.

2.2 Трансформація співпрограм компілятором

Після ознайомлення з усіма важливими частинами, з яких складається співпрограма, перейдемо до того, які маніпуляції проводить компілятор для забезпечення їх роботи.

Коли у функції використовуються ключові слова co_await, co_yield, co_return компілятор вважає цей код співпрограмою та трансформує його для створення обробника, об'єкту обіцянки, стану та об'єкту через який буде здійснюватися взаємодія з цією співпрограмою. Цей процес включає генерацію необхідної інфраструктури для керування життєвим циклом співпрограми, забезпечуючи можливість її призупинення, відновлення та завершення.

Розглянемо її на прикладі співпрограми згаданої в попередньому підрозділі:

```
CoroGenerator generateInts(int max) {
    for (int i = 1; i <= max; ++i) {
        co_yield i;
    }
}
```

Код вище компілятор трансформує в щось схоже на такі рядки [13]:

```
{
    InnerState inner_state = new InnerState;
    inner_state.max = max;
    inner_state.promise = promise_type{};
    CoroGenerator return_object = inner_state.promise.get_return_object();
    co_await inner_state.promise.final_suspend();
    try {
        for (int i = 1; i <= inner_state.max; ++i) {
            co_await inner_state.promise.yield_value(i);
        }
        inner_state.promise.return_void();
        goto FinalSuspend;
    }
    catch (...) {
        inner_state.promise.unhandled_exception();
    }
FinalSuspend:
    co_await inner_state.promise.final_suspend();
    delete inner_state;
}
```

Спочатку в динамічній пам'яті створюється об'єкт стану, куди копіюються параметри передані при виклику співпрограми. Після цього викликається конструктор об'єкт обіцянки та за допомогою його метода get_return_object() створюється об'єкт самої корутини, що і буде результатом її виклику після першої зупинки. Після, разом з co_await викликається метод initial_suspend, який вирішує, чи потрібно призупиняти виконання. Код написаний розробником заноситься в конструкцію try/catch, де при виникненні необроблений виняткових ситуацій буде виконаний метод unhandled_exception. У місці використання co_return або в кінці співпрограми виконується метод return_void() або return_value(value), залежно від того чи має співпрограма повертати якийсь результат. Після цього виконання переходить до точки виконання final_suspend та видалення об'єкту стану.

Для зупинки та відновлення роботи співпрограми на місці кожного ключового слова co_await компілятор додає мітку. Під час зупинки значення цієї мітки зберігається в об'єкті стану. Під час виклику методу обробника resume, виконання переходить у місце мітки яка відповідає збереженому стану [13]. Тоді співпрограма стає схоже на щось представлене нижче:

```
{
    InnerState inner_state = new InnerState;
    inner_state.max = max;
    inner_state.promise = promise_type{};
    CoroGenerator return_object = inner_state.promise.get_return_object();
    co_await inner_state.promise.final_suspend();
suspend_point_0:
    try {
        for (int i = 1; i <= inner_state.max; ++i) {
            co_await inner_state.promise.yield_value(i);
suspend_point_1:
        }
        inner_state.promise.return_void();
        goto FinalSuspend;
    }
    catch (...) {
        inner_state.promise.unhandled_exception();
    }
FinalSuspend:
    co_await inner_state.promise.final_suspend();
```

```
suspend_point_2:
    delete inner_state;
}
```

А метод resume() обробника співпрограми міститиме таку конструкцію:

```
switch (inner_state.suspend_point) {
    case 0: goto suspend_point_0;
    case 1: goto suspend_point_1;
    case 2: goto suspend_point_2;
    default: std::unreachable();
}
```

Такий механізм має назву – машина станів (співпрограм) та є необхідною частиною для реалізації безстекових співпрограм [14].

2.3 Співпрограми бібліотеки Boost Asio

C++20 надає потужний, але базовий каркас для роботи зі співпрограмами, включаючи ключові слова co_await, co_yield, co_return і стандартний обробник корутин std::coroutine_handle. Він дозволяє розробнику створювати співпрограми для найрізноманітніших потреб, проте не має рішень поширених проблем. Тут на допомого приходять сторонні бібліотеки.

Однією з найпоширеніших бібліотек для роботи з асинхронними I/O операціями є Boost Asio. Вона має функціонал для роботи з мережею, потрібний для веб-сервера, планувальник операцій та починаючи з версії 1.18.0 (Boost 1.74) впровадила підтримку співпрограм стандарту C++20. Два важливі компоненти цієї підтримки є asio::awaitable та asio::co_spawn [15].

asio::awaitable - це шаблон класу, який реалізує згадані в попередніх підрозділах концепції та представляє об'єкт, що може використовуватися разом з ключовим словом co_await. Він інкапсулює асинхронну операцію, яку можна виконати в рамках корутини й дозволяє використовувати асинхронні функції у вигляді синхронного коду.

asio::co_spawn - це функція, яка запускає співпрограму в контексті заданого планувальника. Вона дозволяє запускати асинхронні операції у вигляді співпрограм та управляти їхнім життєвим циклом.

Розглянемо їх застосування на прикладі очікування сервером нових клієнтів:

```
void Server::run(io_context& io_context, acceptor acceptor)
{
    co_spawn(io_context, listen(move(acceptor)), detached)
}

awaitable<void> Server::listen(acceptor acceptor)
{
    while (true)
    {
        auto socket = co_await acceptor.async_accept(use_awaitable);
        co_spawn(socket.get_executor(), handleConnection(move(socket)), detached);
    }
}
```

У цьому прикладі функція Server::run використовує co_spawn, щоб запустити співпрограму listen, яка очікує нових клієнтів. Співпрограма listen у вічному циклі асинхронно приймає нові з'єднання за допомогою co_await acceptor.async_accept(use_awaitable), після чого додає до планувальника співпрограму handleConnection, передаючи їх з’єднання для обробки.

Таким чином реалізація співпрограм в бібліотеці Boost Asio спрощує розробку асинхронних серверів, дозволяючи писати код у більш природному стилі, який нагадує синхронний, але зберігає всі переваги асинхронності. Це значно зменшує кількість шаблонного коду та складних структур керування станами, що виникають при використанні традиційних методів асинхронного програмування, таких як зворотні виклики.

2.4 Ефективність asio::awaitable в порівнянні з потоками виконання

В попередньому розділі ми вже розглянули відмінності потоків виконання від співпрограм. Одним з підходів для написання веб-сервера є створення окремого потоку виконання для кожного клієнта. На противагу ньому розглядається підхід співпрограми для кожного клієнта. В цьому розділі ми проведемо симуляцію та виміряємо ефективність роботи кожного підходу із застосуванням потоків виконання та співпрограм класу asio::awaitable.

Тестування передбачає виконання певної задачі з декількома I/O операціями для кожного потоку/співпрограми. У якості I/O операції використовується таймер. Всього задача має п’ять таймерів, кожний по 3 мс, що симулює очікування завершення якоїсь операції. Для цього будемо використовувати asio::steady_time та його метод async_wait у співпрограмах та метод std::this_thread::sleep_for у потоках. Після виконання всіх задач, програма порахує час необхідний для їх виконання. Також програма передбачає задання розміру пулу потоків співпрограм. Повний код програми можна знайти в додатку Б.

Тестування буде проводитися в одноядерному та багатоядерному середовищі для різної кількості задач: 10, 100, 1000, 5000, 10000, 20000. Для одноядерного середовища використано утиліту taskset, яка дозволяє обмежити ядра процесора, на яких буде виконуватися програма [16]. Для проведення тестування використано скрипт мовою Python, який для кожної кількості задач десять разів запускає програму, та обраховує середню тривалість виконання. Програмний код можна знайти в додатку В.

Тестування проводилося на виділеному сервері з двохядерним процесором Intel(R) Xeon(R) Platinum 8280 CPU @ 2.70GHz. В таблиці 2.1 наведено результати роботи в багатоядерному середовищі.

Кількість задач	**Час виконання співпрограм (мс)**	**Час виконання потоків (мс)**
10	16071	15824
100	16162	17833
1000	21059	58209
5000	72828	248817
10000	149082	495625
20000	311070	980393

Таблиця 2.1 – Порівняння потоків та співпрограм asio::awaitable в багатоядерному середовищі

Бачимо, що при досить малій кількості задач потоки показують кращі результати. Проте при збільшенні кількості задач, ефективність співпрограм збільшується до трьох разів в порівнянні з потоками виконання.

В одноядерній системі, результати тестування якої наведені в таблиці 2.2, різниця відчутна вже на невеликій кількості задач.

Кількість задач	**Час виконання співпрограм (мс)**	**Час виконання потоків (мс)**
10	15346	15933
100	15986	18314
1000	20394	56272
5000	84628	240125
10000	172028	473724
20000	356456	944296

Таблиця 2.2 – Порівняння потоків та співпрограм asio::awaitable в одноядерному середовищі

В результаті тестування виявлено, що співпрограми мають набагато більшу ефективність створення та використання для I/O в порівнянні з потоками виконання та блокуючими операціями.

2.5 Висновки до розділу 2

У цьому розділі було проаналізовано ключові компоненти співпрограм у C++20, зокрема очікувані об'єкти (awaitable), обробники співпрограм, та promise_type об'єкти. Виявлено, що співпрограми забезпечують ефективну організацію асинхронного коду, дозволяючи призупиняти та поновлювати виконання програм на зручному для користувача рівні. Особливо важливою є роль promise_type об'єктів, які керують життєвим циклом співпрограм та передачею результатів.

Проведені експерименти та аналіз показали, що використання співпрограм у поєднанні з бібліотекою Boost Asio значно покращує продуктивність асинхронних програм, порівняно з потоками виконання. Співпрограми демонструють кращу масштабованість та зниження накладних витрат на створення та перемикання контексту, що робить їх перспективним інструментом для розробки високоефективних веб-серверів та інших асинхронних систем.

РОЗДІЛ 3. ПРОЕКТУВАННЯ І ТЕСТУВАННЯ ВЕБ-СЕРВЕРА

3.1 Підходи до проектування веб-сервера

При проектуванні веб-сервера, перед розробником постає питання у використанні синхронних чи асинхронних операцій.

Синхронний підхід використовує окремий потік виконання для кожного підключеного клієнта. Це дозволяє обробляти декілька клієнтів одночасно, оскільки під час очікування закінчення операції, операційна система перемикається на виконання іншого потоку. Такий підхід простіший в реалізації, проте має обмежену продуктивність, особливо при великій кількості I/O задач, що було перевірено в попередньому розділі.

Якщо говорити про асинхронний підхід, в цій роботі розглядається найбільш популярний варіант – використання зворотних викликів, та співпрограми як їх альтернатива. Використання асинхронних операцій може покращити продуктивність сервера, проте зворотні виклики ускладнюють його реалізацію. Використання співпрограм полегшує проектування до рівня синхронного сервера, проте залишає всі переваги асинхронних операцій.

Отже, в цьому розділі розглянемо та порівняємо три реалізації веб-сервера: синхронна з використанням потоків виконання, асинхронна з використанням зворотних викликів та асинхронна з використанням співпрограм. Кожна з них має спільні архітектурні елементи та використовує бібліотеки Boost Asio та Boost Beast для роботи з мережею та HTTP протоколом [15,17].

3.2 Загальна архітектура

При проектуванні веб-сервера, незалежно від обраного підходу (синхронного чи асинхронного), архітектура залишається схожою. Вона включає три основні компоненти: клас сервера, клас роутера та контролери.

Клас сервера відповідає за приймання вхідних з'єднань та управління ними. Основні обов'язки цього класу включають: ініціалізацію та конфігурацію мережевих сокетів, прийняття нових клієнтів та створення нових обробників для кожного з'єднання, управління життєвим циклом з'єднань, включаючи закриття сокетів та обробку помилок.

Клас роутера відповідає за маршрутизацію запитів до відповідних обробників. Основні функції цього класу включають: зберігання мапи маршрутів, де кожен маршрут асоціюється з певним обробником, аналіз вхідних запитів та визначення відповідного маршруту, виклик обробника, що відповідає конкретному маршруту. Роутер дозволяє легко додавати нові маршрути та обробники, що робить архітектуру сервера гнучкою та розширюваною.

Контролери реалізують логіку обробки запитів для кожного маршруту. Вони отримують запити від роутера, обробляють їх та формують відповіді. Основні обов'язки контролерів включають: валідацію та розбором вхідних даних, виконання бізнес-логіки та взаємодія з іншими компонентами системи (наприклад, базами даних), формування відповідей.

Для роботи з мережею та протоколом HTTP у цьому проекті використовуються бібліотеки Boost Asio та Boost Beast. Boost Asio забезпечує операції вводу-виводу та управління сокетами, а Boost Beast – інструменти для роботи з HTTP-протоколом.

Залежно від підходу, обробник запиту має мати таке оголошення:

- AnyResponse(Request&&) – для синхронного сервера, проте може використовуватися у всіх підходах
- Coro<AnyResponse>(Request&&) – для співпрограм
- void(Request&&, std::function<void(AnyResponse&&)>) – для асинхронності із зворотними викликами.

Варто зазначити, що Coro є псевдонімом до класу boost::asio::awaitable, а AnyResponse до класу boost::beast::http::message_generator. Тип Request

містить в собі «сирий» запит boost::beast::http::request<http::string_body>, шлях запиту, а також параметри, які були розібрані із шляху. Нижче можна побачити приклад одного з таких обробників.

```
AnyResponse TestController::noio(Request&& req) const
{
    TextResponse res{http::status::ok, req->version()};
    res.set(http::field::content_type, "application/json");
    res.keep_alive(req->keep_alive());
    nlohmann::json j = {
        {"result", "noio result"},
    };
    res.body() = j.dump();
    res.prepare_payload();
    return res;
}
```

Для того щоб використати цей обробник, його потрібно зареєструвати в роутера. Це можна зробити за допомогою одного з методів перелічених нижче.

```
void get(const std::string &route, Func &&func);
void post(const std::string &route, Func &&func);
void put(const std::string &route, Func &&func);
void patch(const std::string &route, Func &&func);
void del(const std::string &route, Func &&func);
void insert(http::verb verb, const std::string &route, Func &&func);
```

Назва відповідає методу HTTP запиту. Або ж можна скористатися функцією insert для задання менш поширених HTTP методів. В якості обробника може виступати як функція так і метод певного класу, наприклад, контролера, які задовольняють вищезгадані оголошення.

Для того, щоб сервер міг використовувати задані шляхи, потрібно передати заповнений об’єкт роутера, як параметр конструктору. Так виглядає створення сервера з двома шляхами:

```
Router router(io_context);
router.get(R"(/test/noio)", &TestController::noio);
router.get(R"(/test/io)", &TestController::ioCoro);
auto server = std::make_shared<HttpServerCoro>(Config::getPort(), io_context,
std::move(router));
```

Загальна архітектура веб-сервера, зображена на рисунку 3.1, виглядає наступним чином:

1. Сервер приймає з'єднання та створює обробник клієнта (потік/співпрограму/об'єкт сесії)
2. Обробник читає запит від клієнта та передає його роутеру.
3. Роутер визначає відповідний контролер та метод для обробки запиту.
4. Контролер обробляє запит та формує відповідь.
5. Обробник відправляє відповідь клієнту.

Рисунок 3.1 – Схема роботи сервера

Така архітектура забезпечує високу гнучкість та розширюваність, дозволяючи легко додавати нові функціональні можливості.

3.3 Синхронний сервер

Синхронний сервер використовує окремий потік виконання для кожного підключеного клієнта. При запуску сервера починається прослуховування нових клієнтів. Для кожного клієнта створюється потік, в якому запускається метод обробки. Це має такий вигляд:

```
void HttpServerSync::_listen()
{
    while (true)
    {
        tcp::socket socket{_ioContext};
        _acceptor.accept(socket);
        std::thread([this, socket = std::move(socket)]() mutable {
            _handleConnection(std::move(socket));
        }).detach();
```

```
    }
}

void HttpServerSync::_handleConnection(tcp::socket socket)
{
    tcp_stream stream(std::move(socket));
    beast::flat_buffer buffer;
    try
    {
        while (true)
        {
            stream.expires_after(std::chrono::seconds(60));

            http::request<http::string_body> req;
            http::read(stream, buffer, req);

            bool keep_alive = req.keep_alive();

            Request request;
            auto url = url::parse_origin_form(req.target());
            request.path = url->path();
            for (const auto &param : url->params())
            {
                request.query[param.key] = param.value;
            }
            request.req = move(req);

            auto msg = _router.route(std::move(request));

            beast::write(stream, std::move(msg));

            if (!keep_alive)
            {
                break;
            }
        }
    }
    catch (boost::system::system_error &se)
    {
        /* if (se.code() != http::error::end_of_stream)
            throw; */
    }
    catch (...)
    {
        throw;
    }

    beast::error_code ec;
    stream.socket().shutdown(tcp::socket::shutdown_send, ec);
}
```

Обробник послідовно читає від клієнта запит, передає його на роутер для отримання відповіді та надсилає клієнту. Ця реалізація синхронного сервера є простою та інтуїтивно зрозумілою, але може мати проблеми з продуктивністю при великій кількості одночасних з'єднань, оскільки кожне з'єднання використовує окремий потік, кількість яких обмежена.

3.4 Асинхронний сервер із зворотними викликами

Асинхронний сервер із зворотними викликами дозволяє більш ефективно використовувати ресурси системи, обробляючи множинні з'єднання у межах одного або декількох потоків виконання. Це досягається за допомогою асинхронних операцій, які не блокують потік під час виконання вводу-виводу. При запуску сервера запускається асинхронна операція очікування клієнта. Коли клієнт підключається, для нього створюється об’єкт сесії, який зберігає об’єкти, потрібні на різних етапах обробки, та запускається операція очікування нового клієнта. Програмний код сервера має такий вигляд:

```
void HttpServerAsync::run()
{
    net::dispatch(
        _acceptor.get_executor(),
        beast::bind_front_handler(
            &HttpServerAsync::_listen,
            shared_from_this()));
}

void HttpServerAsync::_listen()
{
    _acceptor.async_accept(
        net::make_strand(_ioContext),
        beast::bind_front_handler(
            &HttpServerAsync::_onAccept,
            shared_from_this()));
}
```

```
void HttpServerAsync::_onAccept(beast::error_code ec, tcp::socket socket)
{
    if(ec)
    {
        return;
    }
    else
    {
        std::make_shared<HttpServerAsync::Session>(std::move(socket), _router)-
>run();
    }
    _listen();
}

HttpServerAsync::Session::Session(tcp::socket&& socket, const Router& router) :
_stream(std::move(socket)), _router(router)
{
}

void HttpServerAsync::Session::run()
{
    net::dispatch(_stream.get_executor(),
                  beast::bind_front_handler(
                      &HttpServerAsync::Session::doRead,
                      shared_from_this()));
}

void HttpServerAsync::Session::doRead()
{
    _stream.expires_after(std::chrono::seconds(60));
    http::async_read(_stream, _buffer, _req,
            beast::bind_front_handler(
                &HttpServerAsync::Session::onRead,
                shared_from_this()));
}

void HttpServerAsync::Session::onRead(beast::error_code ec, std::size_t
bytes_transferred)
{
    if(ec)
    {
        return;
    }
    else
    {
        doHandleWrite();
    }
}

void HttpServerAsync::Session::doHandleWrite()
{
    bool keep_alive = _req.keep_alive();
```

```
    Request request;
    auto url = url::parse_origin_form(_req.target());
    request.path = url->path();
    for (const auto &param : url->params())
    {
        request.query[param.key] = param.value;
    }
    request.req = move(_req);

    auto self = this->shared_from_this();
    _router.route(std::move(request), [this, self, keep_alive](AnyResponse&& msg){
        beast::async_write(
            this->_stream,
            std::move(msg),
            beast::bind_front_handler(
                &HttpServerAsync::Session::onWrite, self, keep_alive));
  });
}

void HttpServerAsync::Session::onWrite(bool keep_alive, beast::error_code ec,
std::size_t bytes_transferred)
{
    if (ec)
    {
        return;
    }
    else
    {
        if(!keep_alive)
        {
            return doClose();
        }
        doRead();
    }
}

void HttpServerAsync::Session::doClose()
{
    beast::error_code ec;
    _stream.socket().shutdown(tcp::socket::shutdown_send, ec);
}
```

Як бачимо такий код набагато складніше зрозуміти й відповідно підтримувати. Окрім цього з’являється проблема підтримки життя ресурсів між зворотними викликами, а також складність обробки помилкових ситуацій. Також складнішою є маршрутизація запиту, оскільки нам потрібно

проектувати ланцюг із зворотних викликів, замість звичного тримання результату виконання обробника.

Тому такий підхід дозволяє ефективніше використовувати ресурси системи, зменшуючи кількість потоків та переключень контексту, проте використання зворотних викликів ускладнює код, що може призвести до зниження його читабельності. Складність розуміння та підтримки такого коду зростає зі збільшенням кількості зворотних викликів та рівнів вкладеності. Це робить асинхронний підхід із зворотними викликами менш зручним для розробників, особливо у великих проектах.

3.5 Асинхронний сервер із співпрограмами

Асинхронний сервер із співпрограмами (корутинами) дозволяє поєднати переваги асинхронних операцій з простотою та читабельністю синхронного коду. При підключенні, для кожного клієнта створюється окрема співпрограма обробки, як це було в синхронному сервері із створенням потоку. Проте співпрограма може виконуватися в одному або декількох потоках, що має покращити ефективність сервера при великій кількості одночасних клієнтів. Код асинхронного сервера дуже схожий на синхронний:

```
net::awaitable<void> HttpServerCoro::_listen(tcp::acceptor &acceptor)
{
    while (true)
    {
        auto socket = co_await acceptor.async_accept(net::use_awaitable);
        co_spawn(acceptor.get_executor(), _handleConnection(std::move(socket)),
net::detached);
    }
}

net::awaitable<void> HttpServerCoro::_handleConnection(tcp::socket socket)
{
    tcp_stream stream(std::move(socket));
    beast::flat_buffer buffer;
    try
    {
```

```
        while (true)
        {
            stream.expires_after(std::chrono::seconds(60));

            http::request<http::string_body> req;
            co_await http::async_read(stream, buffer, req);

            bool keep_alive = req.keep_alive();

            Request request;
            auto url = url::parse_origin_form(req.target());
            request.path = url->path();
            for (const auto &param : url->params())
            {
                request.query[param.key] = param.value;
            }
            request.req = move(req);

            auto msg = co_await _router.routeCoro(std::move(request));

            co_await beast::async_write(stream, std::move(msg),
net::use_awaitable);

            if (!keep_alive)
            {
                break;
            }
        }
    }
    catch (boost::system::system_error &se)
    {
        if (se.code() != http::error::end_of_stream)
            throw;
    }
    catch (...)
    {
        throw;
    }

    beast::error_code ec;
    stream.socket().shutdown(tcp::socket::shutdown_send, ec);
}
```

Використання співпрограм спрощує структуру коду, роблячи його більш читабельним та зрозумілим. Асинхронні операції виглядають як синхронні, що значно полегшує розробку та підтримку. Код з співпрограмами виглядає

лінійно, без вкладених зворотних викликів, що полегшує його розуміння. Лінійна структура коду зменшує ймовірність помилок та спрощує їх пошук.

Використання асинхронного підходу з співпрограмами дозволяє зберегти переваги асинхронних операцій, але значно спрощує структуру коду в порівнянні з традиційними зворотними викликами, роблячи його більш зручним для розробників, особливо у великих проектах.

3.6 Тестування ефективності

Після огляду реалізації кожного серверу можна перейти до їх тестування. Для оцінки продуктивності різних підходів до реалізації веб-сервера проведено серію тестів із використанням інструменту wrk на виділеному сервері, який має процесор Intel(R) Xeon(R) Platinum 8280 CPU @ 2.70GHz та 8 Гб оперативної пам’яті. wrk є високопродуктивним інструментом для генерації HTTP-навантаження, який дозволяє вимірювати продуктивність веб-серверів при великій кількості одночасних клієнтів [18].

Тестування проводилося із двома обробниками. Один з них не має ніяких додаткових I/O операцій, має шлях /test/noio та був показаний як приклад в попередньому підрозділі. При тестуванні цього шляху використовуються лише базові серверні операції: створення з’єднання з клієнтом, читання запиту, надсилання відповіді.

Інший шлях /test/io має логіку обробника з додатковими I/O операціями. Замість реальних операцій використовується таймер, як це було в тестуванні ефективності asio::awaitable. Він має різні обробники для кожної з реалізацій сервера, проте виконує одні й ті самі операції: три очікування по 10 мс.

Програмний код обробника синхронного сервера виглядає таким чином:

```
AnyResponse TestController::ioSync(Request&& req) const
{
    std::this_thread::sleep_for(10ms);
    std::this_thread::sleep_for(10ms);
    std::this_thread::sleep_for(10ms);
```

```
    TextResponse res{http::status::ok, req->version()};
    res.set(http::field::content_type, "application/json");
    res.keep_alive(req->keep_alive());
    nlohmann::json j = {
        {"result", "sync io result"},
    };
    res.body() = j.dump();
    res.prepare_payload();
    return res;
}
```

Він використовує метод стандартної бібліотеки sleep_for, щоб змусити потік виконання зупинитися на певний час. Асинхронні реалізації використовують таймер steady_timer із бібліотеки Boost Asio. Контролер з використанням зворотних викликів виглядає так:

```
void TestController::ioAsync(Request&& req, std::function<void(AnyResponse&&)>
handler) const
{
    auto timer = std::make_shared<steady_timer>(_io_context);
    auto self = shared_from_this();
    timer->expires_after(10ms);
    timer->async_wait([this, self, timer, req = std::move(req), handler =
std::move(handler)](const boost::system::error_code& ec) mutable {
        if (!ec) {
            timer->expires_after(10ms);
            timer->async_wait([this, self, timer, req = std::move(req), handler =
std::move(handler)](const boost::system::error_code& ec) mutable {
                if (!ec) {
                    timer->expires_after(10ms);
                    timer->async_wait([this, self, timer, req = std::move(req),
handler = std::move(handler)](const boost::system::error_code& ec) mutable {
                        if (!ec) {
                            TextResponse res{http::status::ok, req->version()};
                            res.set(http::field::content_type, "application/json")
                            res.keep_alive(req->keep_alive());
                            nlohmann::json j = {
                                {"result", "async io result"},
                            };
                            res.body() = j.dump();
                            res.prepare_payload();

                            handler(std::move(res));
                        }
```

```
                });
            }
        });
    }
});
}
```

Як було згадано раніше, використання зворотних викликів ускладнює код та вимагає слідкувати життям об'єктів, що використовуються. В цьому прикладі це об'єкт таймеру та кінцевого зворотного виклику, для підтримки життя яких, використовується «розумний» вказівник std::shared_ptr разом з методом shared_from_this. Він передається до списку захоплення лямбда-виразу, тим самим зберігаючи вказівник в об'єкті лямбди та зберігає життя об'єкту яким володіє.

Використання корутин спрощує його до рівня синхронного:

```
Coro<AnyResponse> TestController::ioCoro(Request&& req) const
{
    steady_timer timer(_io_context);

    timer.expires_after(10ms);
    co_await timer.async_wait(use_awaitable);
    timer.expires_after(10ms);
    co_await timer.async_wait(use_awaitable);
    timer.expires_after(10ms);
    co_await timer.async_wait(use_awaitable);

    TextResponse res{http::status::ok, req->version()};
    res.set(http::field::content_type, "application/json");
    res.keep_alive(req->keep_alive());
    nlohmann::json j = {
        {"result", "coro io result"},
    };
    res.body() = j.dump();
    res.prepare_payload();
    co_return res;
}
```

Цей приклад ще раз демонструє перевагу співпрограм над зворотними викликами.

Після огляду обробників повернемося до методології тестування. Тестування було проведено протягом однієї хвилини для різної кількості клієнтів: 100, 1000, 5000 та 10000. Спочатку воно проводилося без будь-яких додаткових параметрів запиту. Це означало, що wrk використовує одне й те саме підключення для надсилання декількох запитів та заголовок Connection: keep-alive [19]. В результаті тестування отримано результати представлені в таблицях 3.1 та 3.2.

	100	1000	5000	10000
Синхронний	14891	14388	13853	13645
Зворотні виклики	9662	9206	8920	8884
Співпрограми	10814	10141	9858	9838

Таблиця 3.1 – Тестування шляху без додаткових I/O операцій (Connection: keep-alive) запитів/секунду

	100	1000	5000	10000
Синхронний	3210	11804	11440	10935
Зворотні виклики	3195	8319	8059	7920
Співпрограми	3205	9413	9105	9057

Таблиця 3.2 – тестування шляху з додатковими I/O операціями (Connection: keep-alive) запитів/секунду

Як бачимо, при такому сценарії тестування синхронний сервер показує кращі результати. Це можу бути пов’язано з тим, що накладні витрати на асинхронні виклики більші за витрати на перемикання контексту потоків виконання. При використанні більшої кількості I/O операцій перевага залишається, проте вже не така велика. Також варто зазначити, що синхронний сервер не може тримати з’єднання з дуже великою кількістю клієнтів одночасно, оскільки створення потоків виконання обмежене, що було показано в попередньому розділі.

Іншим підходом до тестування є задання HTTP заголовку Connection: close. Він означає, що після надсилання відповіді сервер закриває з'єднання з клієнтом. Під час такого тестування для кожного запиту відкривається нове з'єднання. Цей підхід збільшує навантаження, ніби запити надсилає набагато більша кількість клієнтів за задану. Результати такого тестування представлені в таблицях 3.3 та 3.4.

	100	1000	5000	10000
Синхронний	5354	5255	5296	5200
Зворотні виклики	5491	5526	5569	5457
Співпрограми	5917	6000	5964	5865

Таблиця 3.3 – Тестування шляху без додаткових I/O операцій (Connection: close) запитів/секунду

	100	1000	5000	10000
Синхронний	2771	4737	4663	4534
Зворотні виклики	3167	5075	4960	4669
Співпрограми	3178	5467	5307	5068

Таблиця 3.4 – тестування шляху з додатковими I/O операціями (Connection: close) запитів/секунду

Оскільки постійне створення потоків для нових з'єднань більш затратне в порівнянні із перемиканням, в цьому тесті асинхронні підходи починають показувати кращі результати. Також варто зазначити, що у всіх тестах, сервер із застосуванням співпрограм показує кращі результати за сервер із використанням зворотних викликів.

Таким чином, результати тестувань показують, що вибір підходу до обробки запитів значною мірою залежить від конкретного сценарію використання. Для сценаріїв із невеликою кількістю потоків та I/O операцій підійде синхронний підхід. Натомість за умови великих навантажень та

потреби в масштабуванні у майбутньому, краще вибрати асинхронний підхід, до того ж із співпрограмами, оскільки вони не лише спрощують програмний код, а й мають кращу ефективність роботи.

3.7 Висновки до розділу 3

У цьому розділі було розглянуто різні підходи до проектування веб-серверів: синхронний підхід, асинхронний підхід із зворотними викликами та асинхронний підхід із співпрограмами. Було проведено всебічне тестування цих підходів з метою визначення їхньої ефективності в різних умовах навантаження та типах з'єднань.

Результати тестування показали, що синхронний підхід має переваги у сценаріях із постійними з'єднаннями (Connection: keep-alive) та невеликою кількістю клієнтів, де накладні витрати на перемикання контексту потоків виконання виявилися меншими, ніж витрати на асинхронні виклики. Проте, у сценаріях із великою кількістю клієнтів, зроблених за допомогою коротких з'єднань (Connection: close), асинхронні підходи показують кращу продуктивність, оскільки постійне створення потоків для кожного нового з'єднання є більш затратним.

Особливо слід відзначити, що асинхронний підхід із співпрограмами демонструє кращу продуктивність у порівнянні із підходом, що використовує зворотні виклики, у всіх тестових сценаріях. Це свідчить про ефективність співпрограм. Окрім цього співпрограми дозволяють уникнути вкладених викликів, роблячи програмний код і архітектуру сервера більш зрозумілою та простішою.

Висновки

У цій роботі було досліджена модель багатозадачності у вигляді співпрограм. Проаналізовано їх характеристики та класифікацію. Проведено порівняльний аналіз з іншою моделлю – потоками виконання. Проаналізовано переваги й недоліки синхронних та асинхронних операцій введення-виведення й способи їх реалізації. Аналіз показав, що співпрограми забезпечують більшу чистоту коду у порівнянні із зворотними викликами, зокрема у системах з великою кількістю таких операцій.

З'ясовані методи реалізації співпрограм мовою програмування C++ стандарту C++20. Досліджено використання співпрограм із бібліотекою для I/O операцій – Boost Asio. Проведено комп'ютерний експеримент для оцінки ефективності співпрограм з асинхронними операціями та потоків виконання із синхронними. Проведений експеримент підтвердив, що використання співпрограм значно покращує продуктивність I/O задач, порівняно з потоками виконання. Співпрограми демонструють кращу масштабованість та зниження накладних витрат, що робить їх перспективним інструментом для розробки високоефективних веб-серверів.

Досліджено різні підходи до проектування веб-серверів: синхронний з потоком виконання на кожного клієнта, асинхронний із зворотними викликами та асинхронний із співпрограмами. Для кожного підходу виконана реалізація веб-сервера, проведено тестування та порівняльний аналіз ефективності. Результати показали, що синхронний підхід має переваги у сценаріях із постійними з'єднаннями та невеликою кількістю клієнтів, тоді як асинхронні підходи, особливо із співпрограмами, показують кращу продуктивність при великій кількості клієнтів. Асинхронний підхід із співпрограмами демонструє кращу продуктивність у всіх тестових сценаріях порівняно з підходом, що використовує зворотні виклики, та дозволяє уникнути вкладених викликів, роблячи код і архітектуру сервера більш зрозумілими та простими.

На основі проведеного дослідження можна зробити висновок, що використання співпрограм при проектуванні веб-серверів є ефективним підходом, який забезпечує високу продуктивність сервера. Окрім цього співпрограми дозволяють розробникам створювати більш зрозумілий та масштабований програмний код, що є особливо важливим для великих проектів.

Список літератури

1. Moura A. L. D., Ierusalimschy R. Revisiting coroutines. ACM Transactions on Programming Languages and Systems. 2009. Т. 31, № 2. С. 1–31. URL: https://doi.org/10.1145/1462166.1462167 (дата звернення: 04.05.2024).
2. Steingrim D. H. Liberating Coroutines: Combining Sequential and Parallel Execution : Master thesis. 2006. URL: http://hdl.handle.net/10852/9401 (дата звернення: 04.05.2024).
3. Tanenbaum A. S., Bos H. Modern Operating Systems: Global Edition. Pearson Education, Limited, 2015. 1138 p.
4. Stadler L., Würthinger T., Wimmer C. Efficient coroutines for the Java platform. the 8th International Conference, м. Vienna, Austria, 15–17 верес. 2010 р. New York, New York, USA, 2010. URL: https://doi.org/10.1145/1852761.1852765 (дата звернення: 05.05.2024).
5. Coroutines (C++20). cppreference.com. URL: https://en.cppreference.com/w/cpp/language/coroutines (дата звернення: 05.05.2024).
6. ulimit man page. LinuxCommand.org: Learn The Linux Command Line. Write Shell Scripts. URL: https://linuxcommand.org/lc3_man_pages/ulimith.html (дата звернення: 05.05.2024).
7. Synchronous and Asynchronous I/O - Win32 apps. Microsoft Learn. URL: https://learn.microsoft.com/en-us/windows/win32/fileio/synchronous-and-asynchronous-i-o (дата звернення: 07.05.2024).
8. Callback Hell. Callback Hell. URL: http://callbackhell.com/ (дата звернення: 07.05.2024).
9. Jaud-Grimm B., Grimm R. C++20: Get the Details. Independently Published, 2021.
10. Coroutine support (C++20). cppreference.com. URL: https://en.cppreference.com/w/cpp/coroutine (дата звернення: 09.05.2024).

11. Sundaram M. S. C++ Coroutine ~ Deep dive. Medium. URL: https://maharajan-ses.medium.com/c-coroutine-deep-dive-dd3fc0d57708 (дата звернення: 09.05.2024).
12. std::coroutine_handle. cppreference.com. URL: https://en.cppreference.com/w/cpp/coroutine/coroutine_handle (дата звернення: 12.05.2024).
13. Baker L. C++ Coroutines: Understanding the Compiler Transform. Asymmetric Transfer. URL: https://lewissbaker.github.io/2022/08/27/understanding-the-compiler-transform (дата звернення: 13.05.2024).
14. Chen R. C++ coroutines: The mental model for coroutine promises. The Old New Thing. URL: https://devblogs.microsoft.com/oldnewthing/20210329-00/?p=105015 (дата звернення: 13.05.2024).
15. Boost.Asio - 1.85.0. Boost C++ Libraries. URL: https://www.boost.org/doc/libs/1_85_0/doc/html/boost_asio.html (дата звернення: 13.05.2024).
16. Kerrisk M. taskset(1) - Linux manual page. man7.org. URL: https://man7.org/linux/man-pages/man1/taskset.1.html (дата звернення: 15.05.2024).
17. Chapter 1. Boost.Beast - 1.85.0. Boost C++ Libraries. URL: https://www.boost.org/doc/libs/1_85_0/libs/beast/doc/html/index.html (дата звернення: 15.05.2024).
18. GitHub - wg/wrk: Modern HTTP benchmarking tool. GitHub. URL: https://github.com/wg/wrk (дата звернення: 18.05.2024).
19. Keep-Alive - HTTP | MDN. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Keep-Alive (дата звернення: 18.05.2024).

Додаток А. Вихідний код програми для тестування максимальної кількості потоків

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

using namespace std;

using namespace chrono_literals;

int main(int argc, char *argv[])
{
    const int numThreads = 1000000;
    vector<thread> threads;

    int counter = 0;
    try
    {
        for (counter = 1; counter <= numThreads; ++counter)
        {
            threads.emplace_back([]() {
                this_thread::sleep_for(10s);
            });
        }
        for (auto& t : threads)
        {
            if (t.joinable())
            {
                t.join();
            }
        }
    }
    catch (const exception& e)
    {
        cout << "Exception: " << e.what() << '\n';
    }
    cout << "Max number of threads created: " << counter << '\n';
    cout.flush();
}
```

Додаток Б. Вихідний код для порівняння роботи asio::awaitable та потоків виконання

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

#include <boost/asio.hpp>
#include <boost/asio/use_awaitable.hpp>

namespace asio = boost::asio;
using namespace std::chrono_literals;
using namespace std::chrono;

asio::awaitable<void> coro(asio::io_context& io) {
    for (int i = 0; i < 5; ++i)
    {
        asio::steady_timer timer(io, 3ms);
        co_await timer.async_wait(asio::use_awaitable);
    }
}

void sync() {
    for (int i = 0; i < 5; ++i)
    {
        std::this_thread::sleep_for(3ms);
    }
}

int main(int argc, char* argv[])
{
    if (argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <integer> <integer>" << std::endl;
        return 1;
    }

    int client_number = std::atoi(argv[1]);

    auto start = high_resolution_clock::now();
    {
        int pool_size = std::atoi(argv[2]);

        boost::asio::io_context io_context{pool_size};
        for (int i = 0; i < client_number; ++i)
        {
            asio::co_spawn(io_context, coro(io_context), asio::detached);
        }
```

```
        std::vector<std::thread> threadPool;
        threadPool.reserve(pool_size - 1);
        for (auto i = pool_size - 1; i > 0; --i)
        {
            threadPool.emplace_back(
                [&io_context]
                {
                    io_context.run();
                });
        }
        io_context.run();
        for (auto &th : threadPool)
            th.join();

    }
    auto end = high_resolution_clock::now();
    auto coro_duration = duration_cast<microseconds>(end - start).count();

    start = high_resolution_clock::now();
    {
        std::vector<std::thread> threads;
        for (int i = 0; i < client_number; ++i)
        {
            threads.emplace_back([]() {
                sync();
            });
        }
        for (auto& t : threads)
        {
            t.join();
        }
    }
    end = high_resolution_clock::now();
    auto thread_duration = duration_cast<microseconds>(end - start).count();

    std::cout << coro_duration << "," << thread_duration << "\n";

    return 0;
}
```

Додаток В. Вихідний код підрахунку середнього часу виконання

```
import subprocess
import statistics

num_runs = 10
executable_path = "./test"
user_counts = [10, 100, 1000, 5000, 10000, 20000]
one_thread = True

results = {user_count: {"coro": [], "thread": []} for user_count in user_counts}

taskset_cmd = ["taskset", "-c", "0"] if one_thread else []
thread_count = "1" if one_thread else "8"

for user_count in user_counts:
    for _ in range(num_runs):
        command = taskset_cmd + [executable_path, str(user_count), thread_count]
        result = subprocess.run(command, capture_output=True, text=True)
        output = result.stdout.strip()
        coro_duration, thread_duration = map(float, output.split(','))
        results[user_count]["coro"].append(coro_duration)
        results[user_count]["thread"].append(thread_duration)

print("Average Durations:")
for user_count in user_counts:
    average_coro_duration = statistics.mean(results[user_count]["coro"])
    average_thread_duration = statistics.mean(results[user_count]["thread"])
    print(f"User Count: {user_count} - Average Coro Duration: 
{average_coro_duration} ms, Average Thread Duration: {average_thread_duration} ms")
```