ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

КВАЛІФІКАЦІЙНА РОБОТА

на тему: « Розробка рекомендацій для вибору асинхронного чи паралельного програмування в бізнес-задачах »

на здобуття освітнього ступеня бакалавра
зі спеціальності 122 Комп'ютерні науки
(код, найменування спеціальності)
освітньо-професійної програми Комп'ютерні науки
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Роман КРЕНТОВСЬКИЙ
(підпис) *(Ім'я, ПРІЗВИЩЕ здобувача)*

Виконав:
здобувач вищої освіти Роман КРЕНТОВСЬКИЙ
група КНД-42

Керівник: Олег ІЛЬЇН_____
науковий ступінь, вчене звання ___д.т.н_____

Рецензент:
науковий ступінь, вчене звання _____
(Ім'я, ПРІЗВИЩЕ)

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерних наук

Ступінь вищої освіти Бакалавр

Спеціальність Комп'ютерні науки

Освітньо-професійна програма Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедрою Комп'ютерних наук

_____ Віктор ВИШНІВСЬКИЙ
«_____» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Крентовському Роману Сергійовичу
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: Розробка рекомендацій для вибору асинхронного чи паралельного програмування в бізнес-задачах

керівник кваліфікаційної роботи Олег ІЛЛІН, д.т.н., професор,
(Ім'я, ПРІЗВИЩЕ науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «27» 02.2024р. №36

2. Строк подання кваліфікаційної роботи «31» травня 2024р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, документація платформи NodeJS, бізнес-вимоги до швидкодії сервісів.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
Дослідження принципів роботи із системами вводу-виводу
Аналіз переваг та недоліків кожного із підходів
Розробка рекомендацій стосовно вибору архітектурного підходу
5. Перелік графічного матеріалу: *презентація*

1. Взаємодія між основним кодом та системою вводу-виводу
2. Поняття, що базуються на способах взаємодії. Методи, що реалізують ці способи.
3. Порівняння підходів на прикладі вирішення задач
4. Рекомендації
5. Висновки

6. Дата видачі завдання «23» лютого 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	23.02-26.02.24	
2	Вивчення матеріалів для аналізу основних підходів при роботі із системами вводу-виводу	27.02-05.03.24	
3	Дослідження схожих рис та відмінностей цих підходів	06.03-10.03.24	
4	Аналіз переваг та недоліків кожного із підходів	10.03-25.03.24	
5	Підбір задач для дослідження переваг та недоліків на практиці	25.03-30.03.24	
6	Формування рекомендацій для застосування кожного з підходів	30.03-10.04.24	
7	Оформлення роботи: вступ, висновки, реферат	10.04-25.04.24	
8	Розробка демонстраційних матеріалів	25.04-10.05.24	

Здобувач вищої освіти _____ Роман КРЕНТОВСЬКИЙ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Олег ІЛЛІН
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 52 стор., 32 рис., 6 джерел.

Мета роботи – дослідження, яким чином реалізовані асинхронний та паралельний програмні інтерфейси у платформі NodeJS та розробити стратегію використання в залежності від поставлених задач.

Об'єкт дослідження – процес взаємодії із системами вводу-виводу.

Предмет дослідження – сучасні архітектурні підходи, що застосовуються в промислових умовах для взаємодії із системами вводу-виводу.

Короткий зміст роботи: Проведено дослідження того, яким чином відбувається взаємодія між основним кодом програми та системами вводу-виводу.

Проаналізовані такі поняття як конкурентність (concurrency) та паралелізм(parallelism). Розглянуто методи роботи із системою вводу-виводу, які базуються на цих поняттях.

Розроблено програмне рішення підібраних задач для унаочнення переваг та недоліків кожного з методів. Демонстраційна частина створена за допомогою мови програмування JavaScript та платформи NodeJS.

Сформовано рекомендації щодо того, які методи роботи із системами вводу-виводу краще застосовувати у тих чи інших випадках.

КЛЮЧОВІ СЛОВА: СИСТЕМИ ВВОДУ-ВИВОДУ, АСИНХРОННІСТЬ, ПАРАЛЕЛІЗМ, МЕТОДИ, ПОТІК ВИКОНАННЯ

ABSTRACT

Text part of the bachelor's qualification work: 52 pages, 32 pictures, 6 sources.

The research task is to develop recommendations for choosing an architectural approach when working with input-output systems on the NodeJS platform.

The purpose of the study is to investigate how asynchronous and parallel software interfaces are implemented in the NodeJS platform and to develop a strategy for using them depending on the tasks.

The object of research is the process of interaction with input-output systems.

The subject of research is modern architectural approaches used in industrial environments for interaction with I/O systems.

Summary of the work: A study was conducted on how the interaction between the main code of the programme and input-output systems takes place.

Such concepts as concurrency and parallelism are analysed. The methods of working with the I/O system based on these concepts are considered.

A software solution to the selected tasks is developed to illustrate the advantages and disadvantages of each method. The demonstration part was created using the JavaScript programming language and the NodeJS platform.

Recommendations have been made on which methods of working with I/O systems are best used in certain cases.

KEYWORDS: I/O SYSTEMS, ASYNCHRONOUS, PARALLELISM, METHODS, THREAD OF EXECUTION.

ЗМІСТ

ВСТУП

В наш час слово “програмування" стало означати вирішення бізнес-задач шляхом використання методів інформаційних технологій. Іноді програмісту навіть не треба писати код, щоб покрити ту чи іншу проблему, яка виникла у замовника. Самі задачі також набувають трохи іншого вигляду. Здебільшого, лише невелика частина розробників займається написанням складної частини продукту - ядра. Інші ж спеціалісти зазвичай пишуть та налаштовують інфраструктуру навколо цих ядер, що вимагає набагато менше знань так званих “основ комп’ютерної науки”. На додаток до цього варто згадати про появу високорівневих інтерпретованих мов програмування. Такі мови дозволяють інженеру абстрагуватися від тієї частини системи, що відповідає за роботу з файлами, операційною системою, процесором, мережею тощо. Вони надають зручний та надійний інтерфейс для виконання вищезгаданих операцій. І це добре, адже програміст може більше зосередити свою увагу саме на тих потребах, які має замовник програмного забезпечення, передавши управління пам’яттю, потоками і подібними речами платформі, з допомогою якої він пише код.

Однак, час від часу, виникає ситуація, коли користувачів стає дедалі більше, обсяг даних, які ми повинні обробити, теж росте. І, якщо на це ніяк не реагувати, в один момент наша програма може перестати працювати. Тоді перед спеціалістом ставиться задача оптимізувати роботу програми, тобто переписати код таким чином, щоб він виконував ті ж самі задачі ефективніше. В цей момент програміст починає аналізувати, в який момент програма перестала виконувати відповідні функції за встановлений час, де знаходиться та частина коду, яка гальмує всі інші процеси тощо…

Хоча, частіше всього, проблема криється в тому, що програміст не до кінця знає особливості платформи, на базі якої пише код, або не має базових знань роботи тих чи інших алгоритмів та структур даних, інколи варто задуматися над

тим, щоб виконати якусь частину програми незалежно від інших частин. Це допоможе зберегти багато часу, ефективніше використовувати наявні ресурси пристрою та вирішити проблему швидкодії програми.

Як нам може допомогти знання асинхронного та паралельного підходів при написанні програм та проектуванні архітектури? За допомогою цих підходів ми зможемо організувати виконання нашого коду таким чином, аби одні його частини не заважали іншим. До прикладу, ми можемо зробити так, щоб запит до бази даних не заважав обчисленню складного відсотка, а введення користувачем якихось символів у текстове поле не блокувало анімації у веб-застосунку. Дослідивши відмінності цих підходів, ми зрозуміємо, де і коли який підхід краще застосовувати. Наприклад, ми зможемо обґрунтувати вибір архітектурного рішення, коли будемо проектувати новий http-сервер для відправки html-документів. Також, розуміючи принципи, що закладені в ці підходи, ми зможемо вирішувати більш низькорівневі задачі, що розширить спектр послуг, які ми можемо надавати. Саме тому так важливо розуміти, чим відрізняється асинхронне програмування від паралельного.

1 ОСНОВНІ ПІДХОДИ ПРИ РОБОТІ З ВВОДОМ-ВИВОДОМ

1.1 Поняття системи вводу-виводу

Сучасне програмне забезпечення рідко може покривати всю необхідну функціональність виключно за рахунок власного коду. Нерідко навіть невелика програма потребує інфраструктури для розгортання, для якої треба написати конфігурацію та в яку потім інтегрується програмний код. Програма може робити запити до бази даних, читати файли у системі, взаємодіяти з користувачами, містити чимало інтеграцій зі сторонніми сервісами і так далі… Також останнім часом з’явився тренд до міграції у мікросервісну архітектуру, що збільшило кількість потоків даних до модуля, який ми можемо розробляти як команда незалежно від інших програмістів. Одним словом, з часом кількість операцій вводу-виводу стає чимраз більшою. Взагалі, коли ми говоримо про блокування / неблокування, ми маємо на увазі роботу з вводом-виводом(I/O-операції). Операції вводу-виводу – операції взаємодії нашої програми із зовнішнім світом. Наприклад, обробка запитів. Зараз я пишу цей диплом за допомогою Google Docs, користуючись браузером, який можна назвати клієнтом. Десь там на стороні серверів Google є програмний код(сервер), який обробляє запити від мого браузера(клієнта). Для того, щоб почати взаємодію, клієнт та сервер встановлюють з’єднання один з одним. Ми не будемо заглиблюватися в семирівневу архітектуру, щоб зрозуміти, як саме це відбувається на кожного рівні. Основне, що ми повинні зрозуміти – це те, що клієнт та сервер мають спеціальні точки з’єднання(сокети), через які вони слухають один одного та передають інформацію на іншу сторону з’єднання. На кожен запит від клієнта сервер виконує якусь дію: обробляє запит, генерує дані для відповіді клієнту, перевіряє, чи клієнт є дійсно тим, за кого він себе видає і т.д. І якщо сервер виконує операції вводу-виводу більшість часу своєї роботи, то він, фактично, простоює, чекаючи на новий запит. Так ввід-вивід стає вузьким місцем(bottleneck) нашого програмного забезпечення. Виходить, нам як розробникам треба якось організувати цей процес вводу-виводу так, щоб він став

оптимальним. Для цього нам треба познайомитися із поняттями блокуючого та неблокуючого вводу-виводу[3, с. 6]. Є два шляхи, як організувати систему вводу-виводу: блокуючий та неблокуючий. Є також два типи операцій системи вводу-виводу: синхронний та асинхронний. Так створюється матриця моделей вводу-виводу (Рис. 1.1)

Рисунок 1.1 – Матриця організації процесу вводу-виводу

1.2 Блокуючий ввід-вивід

При такому підході процес виконання відбувається наступним чином: клієнт робить запит до серверу, утворюється сокет, який керує з'єднанням, та відповідний йому один потік, що виконує код утворення з'єднання. Цей сокет та потік заблоковані, тобто не можуть виконувати жодної іншої операції, крім як обробляти запит, допоки не завершиться обробка і дані не потраплять до клієнта[3, с. 7]. Висновок, до якого можна одразу дійти: ми не можемо обробити більше ніж одне

з’єднання, маючи один потік і таку модель вводу-виводу. Напишемо просту імплементацію такого клієнта та сервера. Клієнт буде просто викликати метод серверу, який буде через 5 секунд віддавати дані у JSON-форматі:

Клієнт:

```
const createClient = (server) => {
  return {
    clientId: Math.round(Math.random * 10_000),
    logData: () => log(server.getData()),
  };
};
```

Сервер:

```
const createServer = () => {
  return {
    getData: () => {
      const now = Date.now();
      while (Date.now() - now < 5000) {}
      return fs.readFileSync(path.resolve("src", "data.json"), {
        encoding: "utf-8",
      });
    },
  };
};
```

Викликаємо метод для перевірки і бачимо результат в консолі (Рис. 1.2):

```
createClient(createServer()).logData();
```

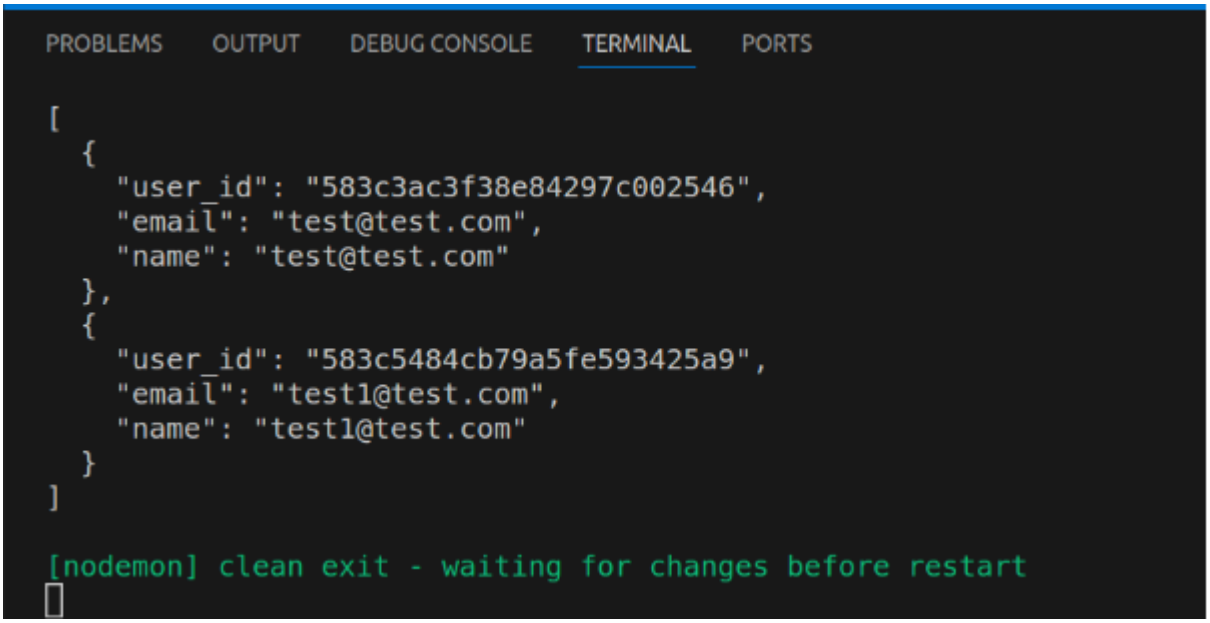


Рисунок 1.2 – Результат виконання коду

Здавалося б, усе працює. Але уявіть, якщо у нас з’явиться не один клієнт, а два. У тому разі можливий наступний сценарій: перший клієнт робить запит за даними, сервер починає обробку цього запиту(чекає 10 секунд, після чого читає файл та відправляє його зміст). В цей час другий клієнт теж хоче отримати необхідні дані. Він теж робить за ними запит. Але, так як ми маємо єдиний потік, який містить код серверу, що обробляє запити, другий клієнт муситиме чекати, поки перший клієнт отримає відповідь. Тільки після того сервер зможе взятися за обробку запиту від другого клієнта. В нашому випадку це може зайняти більш ніж 20 секунд. Не важко здогадатися, що це дуже неефективний спосіб роботи з вводом-виводом, якщо ми маємо більше, ніж одного клієнта. Так як більшу частину часу процесор просто “чекає”, не виконуючи при цьому жодної ефективної роботи.

1.3 Неблокуючий ввід-вивід

Різницю між попередньою моделлю роботи з операціями вводу-виводу та поточною видно із самої назви. На відміну від блокуючої моделі тут кожна операція буде повертати результат миттєво. Як це працює? Кожен запит буде ставитися в чергу і функція буде повертати результат, а фактичний ввід-вивід буде відбуватися

пізніше[3, с. 10]. Як це можна зробити? Перше, що спадає на думку – це використати прийом під назвою полінг(polling). Тобто ми будемо робити запити до тих пір, поки дані не стануть нам доступними. І, як тільки, дані можна отримати, ми будемо повертати їх у якості результату.

Додам функцію wait для того, щоб робити запит за даними не одразу після того, як ми отримали пустий результат, а через деякий час(до прикладу, через одну секунду):

```
const wait = async (time) => {
  return new Promise((res) => {
    setTimeout(() => {
      res();
    }, time);
  });
};
```

Трохи змінимо наш клієнт:

```
const createClient = (server) => {
  return {
    clientId: Math.round(Math.random * 10_000),
    async logData() {
      const res = server.getData();
      if (!res) {
        await wait(1000);
        this.logData();
        return;
      }
      log(res);
    },
  };
};
```

Також оновимо код серверу:

```
const createServer = () => {
  let isAvailable = false;
  return {
    getData: () => {
      setTimeout(() => {
        isAvailable = true;
      }, 5_000);

      if (!isAvailable) {
        return null;
      }

      return fs.readFileSync(path.resolve("src", "data.json"), {
        encoding: "utf-8",
      });
    },
  };
};
```

На останок створимо один сервер і два клієнти, щоб протестувати їхню роботу:

```
const server = createServer();
log("First client requests some data...");
createClient(server).logData();
log("Second client requests some data...");
createClient(server).logData();
```

Результат виконання можна побачити в терміналі (Рис. 1.3)

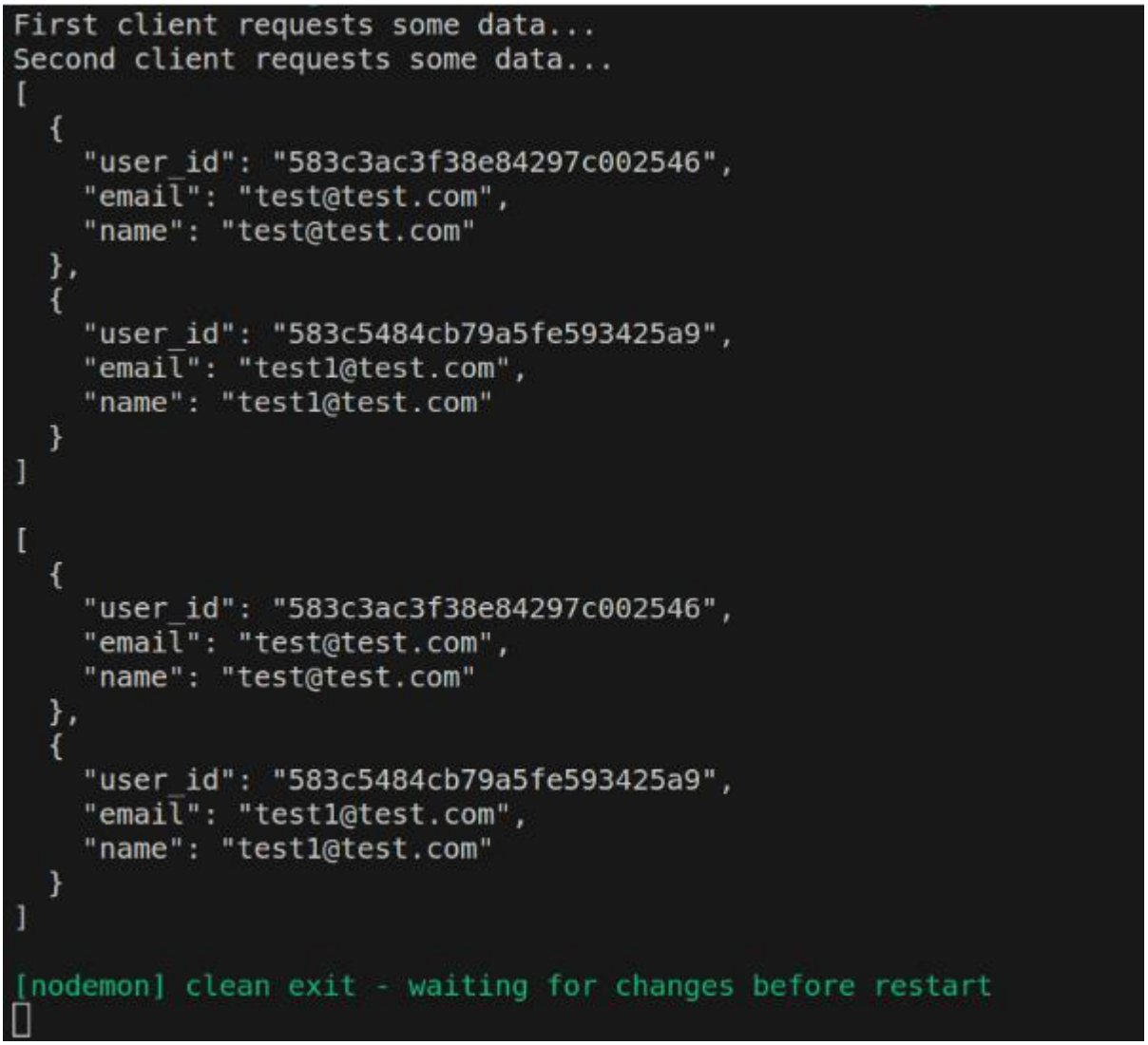


Рисунок 1.3 – Результат виконання коду

По-перше, як ми бачимо з консолі, спочатку відбулося логування обох запитів, а тільки потім прийшла відповідь на кожен запит(сервер відразу повернув керування в місце, з якого робився запит, а вже потім, коли дані стали доступними, повернув їх). По-друге, перший клієнт не блокував другого, тобто йому не довелося чекати десять секунд, аби отримати необхідні дані. Проблема такого підходу полягає в тому, що ми не знаємо напевне, скільки нам доведеться чекати, поки ми отримаємо відповідь на свій запит. Що буде робити клієнт, допоки не отримає відповідь? Буде “бомбардувати” сервер своїми запитати. Тобто сервер може отримати десятки запитів, на які муситиме відповісти, що дані поки-що відсутні, що теж не є оптимально, так як кількість таких запитів буде рости пропорційно

кількості клієнтів. В один момент сервер може просто “лягти” під таким навантаженням. Цей підхід може мати перевагу перед попереднім підходом лише у тих випадках, коли або кількість клієнтів невелика, або ми точно знаємо час затримки і можемо оптимально створювати нові запити до серверу.

Щоб позбутися неефективного потоку запитів, нам потрібно якось зрозуміти, чи готовий сервер віддати нам необхідні дані. У цьому випадку ми могли б опитувати готовність всіх запитів у черзі, і вони повідомляли б нам, який з них готовий до нової операції вводу/виводу, а який - ні не без явного запиту. Коли будь-який з запитів буде готовий, ми будемо виконувати операції у черзі, а потім зможемо повернутися у стан блокування , чекаючи на готовність серверу до наступної операції вводу/виводу.

Існує декілька механізмів реалізації цього підходу, які відрізняються за продуктивністю та деталями реалізації. Ми розглянемо такі механізми як конкурентність та паралелізм.

2 МЕХАНІЗМИ РЕАЛІЗАЦІЇ НЕБЛОКУЮЧОГО ВВОДУ-ВИВОДУ

2.1 Конкурентність(concurrency)

Конкурентність (concurrency) – це процес одночасного виконання декількох завдань у визначений проміжок часу, при цьому завдання можуть починатися незалежно одне від одного, тобто без очікування завершення попереднього завдання. Ці завдання можуть виконуватися паралельно, хоча це не обов'язково. Концепція конкурентності ґрунтується на принципі "переривання" – завдання можна розбити на підзадачі й виконувати їх по черзі[3, с. 13] (Рис. 2.1)

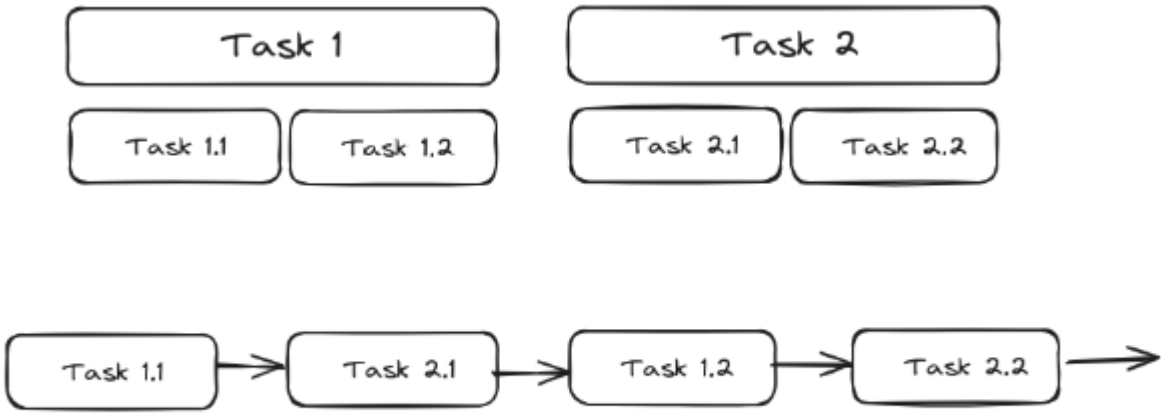


Рисунок 2.1 – Візуалізація поняття “Конкурентність”

Задачі діляться на підзадачі, які не пов’язані між собою. Тому нам не важливо, в якій послідовності вони будуть виконуватися і в якій повернуть результат виконання. Щоб краще це уявити, давайте розглянемо приклад з життя. Уявіть, що у вас є найманий працівник, який виконує всю хатню роботу: готує, прибирає, поливає квіти тощо. Майже весь час він прибирає, три рази на день готує, раз на тиждень поливає квіти. Це незалежні задачі, результати виконання яких не залежать одна від одної. Щоб почати готувати, працівнику треба припинити прибирати. Після закінчення процесу приготування їжу, працівник може знову

взятися за прибирання. Також він пам’ятає про квіти, які треба полити. Але виконає він цю задачу пізніше, коли прийде необхідний час.

2.2 Паралелізм(Parallelism)

Паралелізм (parallelism) – це буквальне одночасне виконання завдань. Сам термін вказує на те, що завдання виконуються одночасно. Паралелізм є одним із методів реалізації одночасного виконання, акцентуючи увагу на абстракції потоку або процесу[3, с. 15]. Для здійснення паралелізму необхідно мати щонайменше два обчислювальних ресурси (Рис. 2.2)

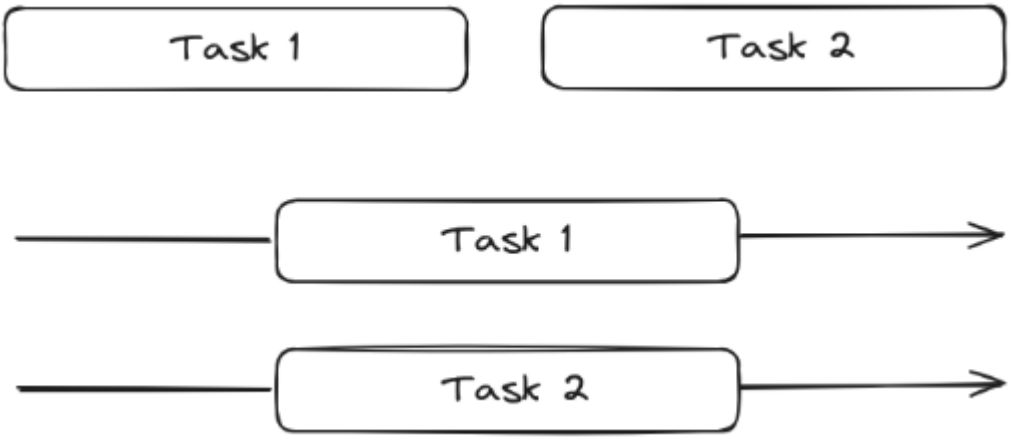


Рисунок 2.2 – Візуалізація поняття “Паралелізм”

Повернемося до нашого прикладу з хатнім робітником. Тепер ви маєте не одного робітника, а трьох. Кожен з них виконує одну конкретну задачу весь свій робочий час. Один працівник постійно готує, інший прибирає, а третій слідкує за часом та у відповідний момент поливає квіти. Різницю між цими підходами чітко показано на рисунку 2.3

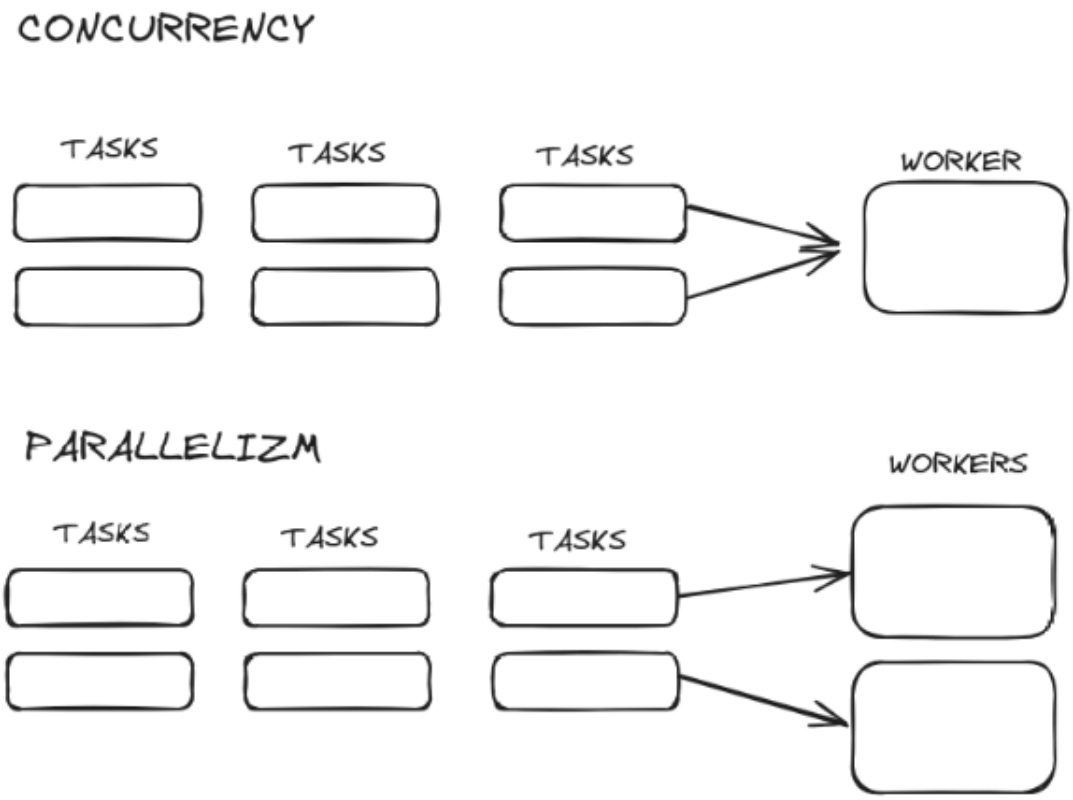


Рисунок 2.3 – Візуалізація різниці між конкурентністю та паралелізмом

Parallelism є підмножиною Concurrency, адже перед початком виконання задач таким чином, щоб початок одних не залежав від завершення інших, ці задачі треба правильно організувати[3, с. 15].

Отож, ми розглянули механізми побудови неблокуючого вводу-виводу. Тепер ми дослідимо способи імплементації цих механізмів.

3 ІМПЛЕМЕНТАЦІЯ МЕХАНІЗМІВ ПАРАЛЕЛІЗМУ ТА КОНКУРЕНТНОСТІ

3.1 Імплементація паралелізму

Для кращого розуміння способів імплементувати паралелізм, варто спуститися на рівень операційної системи та пригадати, яку роботу по відношенню до виконання програм робить вона. Основна задача операційних систем - виконання програм. Виділення пам’яті, обробка та таке інше - це все допоміжні засоби для вирішення основної задачі. Після компіляції програми вихідний код перетворюється в набір інструкцій, які може виконати процесор. Для виконання цього коду операційній системі треба виконати певний ряд дій. До прикладу, їй треба виділити пам’ять, необхідну для виконання програми, а потім завантажити саму програму в цю виділену пам’ять. Після того, як програма набула такого вигляду, в якому її можна виконувати, вона стала процесом[1, с. 70].

Процес - це програма у стані виконання. Кожен процес має свою адресу у віртуальній пам’яті та своє середовище виконання. Тобто кожен процес є ізольованим. З цієї властивості процесів випливає те, що кожен процес буде вимагатися чимало ресурсів комп’ютера. Але для більшості задач нам не потрібне ізольоване виконання. І тут на допомогу нам приходять потоки.

Поток – базова одиниця виконання програми в рамках поточного процесу[6, с. 20]. В рамках одного процесу може існувати декілька потоків, що забезпечує швидкий доступ до спільних даних. Створення потоків легше за створення процесів, адже немає потреби знову створювати адресний простір та іншу необхідну оболонку для запуску[2, с. 39]. Недоліком потоків є відсутність ізоляції даних. Тобто, якщо у нас є потреба часто читати та змінювати одні й ті ж дані, то нам треба буде додатково подумати над тим, як ми будемо організовувати управління нашими потоками, аби всі вони отримували актуальні та достовірні дані. Також потоки залежать від процесу, в рамках якого вони створені. Тобто, при

завершенні роботи процесу завершують свою роботу всі його потоки. Як ці знання можуть допомогти нам у роботі із вводом-виводом?

Перший варіант - керувати роботою вводу-виводу, створюючи новий процес на кожен запит. Ми залишаємося в рамках блокуючої системи вводу-виводу, але це блокування відбувається в рамках одного процесу. Тобто, якщо якась операція займає більше часу, ніж ми очікуємо, це вплине лише на ті запити, які обробляються в рамках одного процесу. А так як ми створюємо новий процес на кожен запит, то затримки в цьому процесі ніяк не будут впливати на час виконання інших запитів в інших процесах. Ми можемо створювати новий процес на кожен запит або ж одразу створити сталу кількість процесів та керувати розподіляти роботі із вводом-виводом між ними. Такий механізм використовують PostgreSQL та Apache[3, с. 17]. Цей варіант має декілька мінусів. По-перше, якщо нам все ж доведеться підтримувати якусь комунікацію між процесами, то зробити це буде не так просто, адже формально вони ізольовані. По-друге, якщо процесів буде більше, ніж запитів на операції вводу-виводу, то ми будемо витрачати багато ресурсів просто на очікування, що не є добре.

Другий варіант - використання потоків (threads). У цьому випадку ми також працюємо в блокуючій системі введення-виведення, блокуючи виконання в межах одного потоку. Операційна система автоматично розподіляє потоки між ядрами процесора. Оскільки потоки легші за процеси, ми можемо створити багато потоків для виконання завдань. Однак через відсутність ізоляції, аварійне завершення одного потоку може призвести до завершення всього процесу і, відповідно, всіх інших потоків. Ще одна проблема - керування спільною пам'яттю між потоками. Оскільки пам'ять спільна, необхідно знайти спосіб синхронізувати роботу потоків з нею. Цей спосіб має вирішувати дві основні проблеми. По-перше, синхронізація може призвести до тупиків (deadlocks). Тупик виникає, коли потік переходить у стан очікування, оскільки запитуваний ресурс зайнятий іншим потоком, що також очікує на ресурс, зайнятий іншим потоком. По-друге, відсутність синхронізації може призвести до конкурентного доступу до спільних даних, коли два потоки

змінюють дані одночасно. Такі програми важче рефакторити та дебажити[3, с. 19–20].

3.2 Імплементація конкурентності

Ми знаємо, що операційна система ефективно працює з процесами та потоками, здатна правильно ними керувати, своєчасно виділяти необхідні ресурси для їхньої роботи, перемикати контексти виконання та виконувати інші подібні задачі. Проте операційна система не має уявлення про специфіку роботи нашого програмного застосунку. Наприклад, вона може перемикати контекст виконання у невідповідний момент або виділити більше ресурсів, ніж це необхідно для роботи програми. Як розробники, ми знаємо, коли потрібно виконати операцію, яка вимагає значних обчислювальних ресурсів, а коли слід просто очікувати запит із системи введення-виведення. Ми також краще розуміємо, коли варто змінити контекст виконання для переходу до більш важливого завдання. У таких випадках нам на допомогу приходить поняття "багатозадачність у співпраці" (Cooperative multitasking).

З точки зору операційної системи, багатозадачність у співпраці – це виконання в одному потоці, в якому застосунок має можливість самостійно перемикати контекст виконання[3, с. 21]. Наприклад, якщо ми розробляємо застосунок, що виступає посередником між клієнтом і певним сервісом, після отримання даних від клієнта ми починаємо їх обробку. Завершивши обробку, ми надсилаємо дані до сервісу. Замість того, щоб просто зупинити будь-яку роботу і чекати на відповідь від сервісу, ми можемо продовжити приймати інші запити від клієнтів. Це і є "співпраця", оскільки всі завдання повинні взаємодіяти, щоб виконуватися у відповідний час. Вони переплітаються між собою, але в єдиному потоці керування, відомому як кооперативний планувальник, який відповідає за запуск завдань і надає їм можливість добровільно повернути керування назад.

Однопотокова асинхронна система завжди працює в режимі чергування навіть на багатоядерній системі. Складність написання таких програм полягає в

тому, що відповідальність за перемикання, підтримку контексту та організацію кожного завдання як послідовності менших кроків з перериваннями лягає на розробників. З іншого боку, ми отримуємо переваги в ефективності, оскільки уникатимемо зайвих перемикань контексту, таких як перемикання контексту процесора між потоками та процесами[3, с. 22]. Існують два способи реалізації багатозадачності у співпраці: функції зворотного виклику (callbacks) та кооперативна обробка потоків (cooperative threads).

Функції зворотного виклику - це функції, які викликаються після виконання певної дії або досягнення певної умови. Усі кооперативні операції означають, що дія відбудеться в майбутньому, і потік виконання повинен повернути результат, коли він буде готовий. Щоб отримати результат, ми повинні зареєструвати зворотний виклик: якщо запит або операція пройшли успішно, викликається одна функція, а якщо неуспішно - інша. Зворотний виклик є явним підходом, тобто розробник пише програми так, ніби не знає, коли буде викликана функція зворотного виклику. Це найпоширеніший варіант, оскільки він є очевидним і підтримується більшістю сучасних мов програмування. Перевага такого підходу полягає у відсутності проблем, пов’язаних з керуванням потоками. Недоліки включають можливе вкладання зворотних викликів, що призводить до так званого "пекла зворотних викликів" (callback hell), а також ускладнення процесу відлагодження таких програм[3, с. 23].

За принципом функції зворотнього виклику працюють таймери на платформі NodeJS. Ось приклад програми, яка виведе повідомлення в консоль через 5 секунд:

```
const logAfterFiveSeconds = (message) => {
  setTimeout(() => {
    console.log(message);
  }, 5_000);
};
console.log("Start working...");
logAfterFiveSeconds("Doing some useful stuff...");
console.log("Do some another useful stuff...");
```

Зауважте, що виведе нам консоль (Рис. 3.1)

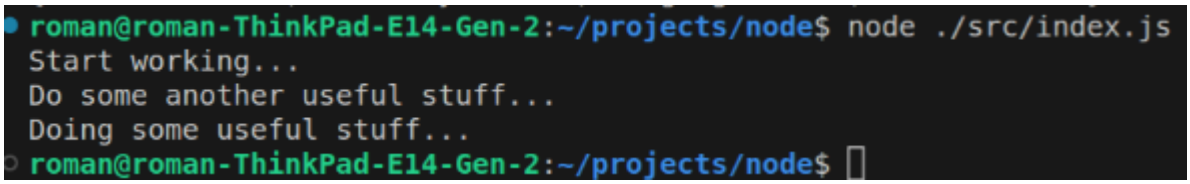


Рисунок 3.1 – Результат виконання програми

Повідомлення “Do some another useful stuff...” вивелося раніше, ніж "Doing some useful stuff..." Тобто, функція logAfterFiveSeconds не блокувала основний потік виконання програми. Якимось чином програма через API платформи відклала виконання команди console.log(message) на 5 секунд, і, не очікуючи результат, почала виконувати всі подальші команди. Потім, коли пройшов відповідний час, програма знову отримала управління та виконала згадану вище команду. Це і є багатозадачність у співпраці через функції зворотнього виклику.

Інший спосіб реалізації - користувацькі потоки. Цей метод часто називають неявним, оскільки розробники пишуть програму так, що здається, ніби кооперативної багатозадачності не існує. Існують різні реалізації цього підходу, такі як користувацькі потоки (так звані "зелені потоки") або підпрограми[3, с. 24]. Використовуючи "зелені потоки", ми можемо виконати операцію блокування так само, як раніше, і одразу очікувати результат, як якщо б операція не була блокуючою. Але "під капотом" фреймворк або мова програмування перетворює блокуючу операцію на неблокуючу і передає керування іншому потоку виконання, але не в сенсі потоку операційної системи, а логічного потоку (потоку користувача). Ці потоки виконуються в межах "звичайного" користувацького процесу, а не процесу операційної системи.

У підпрограмах програми пишуться так, щоб включати "контрольні точки", де функція може бути призупинена і відновлена. Виклик інших підпрограм дозволяє повернутися до початкової точки виклику. Підпрограми дуже схожі на потоки, але вони є кооперативно багатозадачними, тоді як потоки зазвичай використовують витісняючу багатозадачність. Підпрограми не потребують примітивів синхронізації, таких як м'ютекси чи семафори, і не вимагають

підтримки операційної системи. Перевагою цього підходу є контроль на рівні користувацького простору, а не ОС, а також схожість із синхронним програмуванням, що полегшує читання та підтримку коду. Недоліком є те, що розробники повинні передбачати точки, в яких безпечно перемикати контекст виконання[3, с. 24–25].

Ось приклад реалізації типової функції map (функціональний аналог циклу for, який приймає на вхід початковий масив і повертає новий масив з обробленими даними):

```
const map = (array, map, options) => {
  const interval = typeof options == "object" && options.hasOwnProperty("interval")
     ? Options.interval : 10;
  const results = [];
  const recurse = (startIndex) => {
    const startTime = Date.now();
    for (const a = startIndex, l = array.length; a < l; a++) {
      const item = array[a];
      if (Date.now() - startTime > interval) {
        return new Promise(function (resolve) {setImmediate(resolve)}).then(function () {
          return recurse(a);
        });
      }
      results.push(map(item, a));
    }
    return Promise.resolve();
  };
  return recurse(0).then(function () {return Promise.all(results)})};
```

Цей код працює, використовуючи NodeJS API setImmediate. Давайте протестуємо роботу коду, виконавши просту операцію: піднесення числа до квадрату.

```
const array = [1, 2, 3, 4, 5];
const mapper = (item) => item * item;
```

```
const options = {};
const main = async () => {
  let mappedResults = await map(array, mapper, options);
  console.log(mappedResults);
};
main();
```

Отримуємо наступний вивід у консоль (Рис. 3.2)

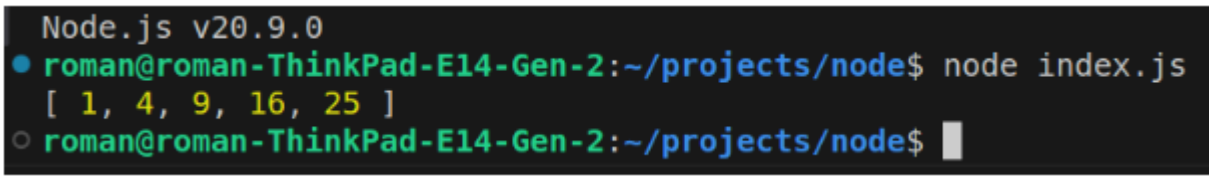


Рисунок 3.2 – Результат виконання програми

Якщо ми спробуємо виконати якусь звичайну операцію після виклику функції main, то вона поверне результат раніше, ніж виконається сама функція main:

```
main();
console.log("Starting...");
```

Результат (Рис. 3.3)

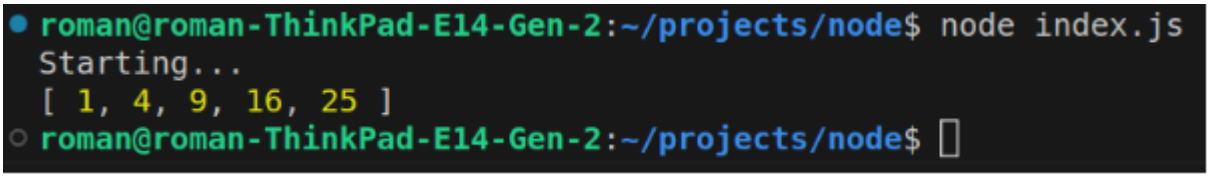


Рисунок 3.3 – Результат виконання програми

Це означає, що ми не блокуємо основний потік виконання програми і можемо виконувати якусь корисну роботу.

Зазвичай ці методи вже реалізовані на різних платформах для розробки. Найбільш основним і популярним є шаблон Реактор/Проактор[3, с. 26].

У кооперативній багатозадачності завжди є процесор, який відповідає за всю обробку введення/виведення. Цей процесор називається Реактором (Reactor) за назвою шаблону проектування. Завдання Реактора полягає в реагуванні на події введення/виведення шляхом делегування всієї обробки відповідному обробнику.

Обробники виконують обробку, тому немає необхідності блокувати введення/виведення, оскільки зареєстровані обробники або функції зворотного виклику обробляють події. Мета шаблону проектування Реактор - уникнути поширеної проблеми створення потоку для кожного повідомлення, запиту та з'єднання. Реактор отримує події від кількох обробників і послідовно розподіляє їх між відповідними обробниками подій. Таким чином, стандартний Реактор дозволяє запускати додаток з одночасними подіями, зберігаючи при цьому простоту однопотокової обробки. Для цього зазвичай використовується неблокуючий синхронний введення/виведення.

Більш цікавим є патерн Проактор (Proactor), який є асинхронною версією патерну Реактор. Він зазвичай використовує справжні асинхронні операції введення/виведення, що надаються операційною системою. Весь введення/виведення виконується на неблокуючих сокетах, які опитуються за допомогою найкращого доступного механізму на даній платформі. Як частина ітерації циклу, цикл блокує очікування активності введення/виведення на сокетах, доданих до опитувача, і виконує зворотній виклик із зазначенням стану сокета (читання, очікування на запис), щоб клієнтський код міг читати, записувати або виконувати необхідні операції введення/виведення.

Ці патерни використовуються у Nginx та платформі Node.js.

Тепер ми можемо перейти до безпосереднього розгляду асинхронності та розглянути, як асинхронність, що базується на патернах Реактор/Проактор, реалізована на платформі Node.js.

4 АСИНХРОННІСТЬ

4.1 Асинхронний стиль

Розробка додатків, які використовують кооперативну багатозадачність, має певні труднощі, оскільки відповідальність за перемикання контексту лежить на розробниках. Однак, цей підхід дозволяє досягти високої ефективності, уникаючи надмірних перемикань між контекстами виконання. Найбільш оптимальним рішенням є поєднання кооперативної багатозадачності з патернами Реактор/Проактор[3, с. 28].

Перейдемо до роботи із системами вводу-виводу. Основна проблема полягає в тому, що вони блокують основний потік і витрачають багато часу. Під час запису на жорсткий диск або роботи з мережею, процес фактично не виконує корисну роботу. Ба більше, можна було б обробляти кілька джерел вводу-виводу одночасно, не очікуючи результатів від конкретного джерела. При високому навантаженні на застосунок, оптимізація процесу вводу-виводу стає критично важливою. Стандартним рішенням є використання процесів або потоків. У синхронній моделі роботи з вводу-виводу, процес або потік прив'язується до завдання і починає його виконання. Після завершення поточного завдання він переходить до наступного, повторюючи цикл. При такому підході процес або потік не може частково виконати завдання і перейти до іншого. Це означає, що функція, яка виконується, не може бути перервана і буде завершена перед тим, як почнеться інша, що запобігає змінам даних, з якими працює поточна функція. Якщо система виконується в одному потоці і містить кілька завдань, вони будуть виконуватись послідовно, одне за одним[3, с. 32] (Рис. 4.1)

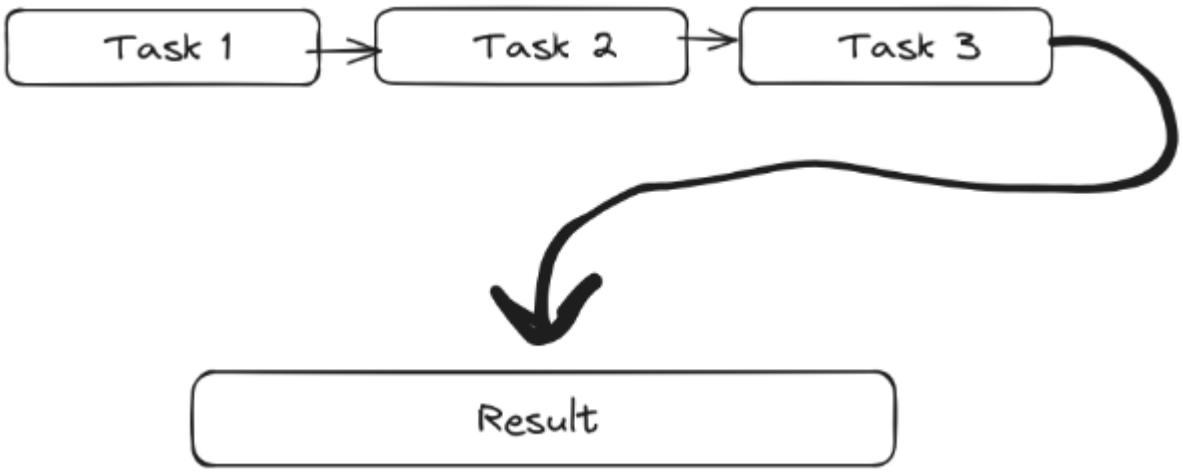


Рисунок 4.1 – Візуалізація синхронного виконання

У багатопотоковій системі діє принцип: один потік призначається для однієї задачі і працює над нею до завершення. Керування потоками здійснює операційна система. Потоки можуть виконуватися паралельно на системах з кількома процесорами або ядрами, або можуть бути мультиплексовані на одному процесорі. Отже, коли є декілька завдань і більше одного потоку, які повинні виконуватися одночасно, потік, завершивши одне завдання, може перейти до наступного[3, с. 33]. Процес показаний на рисунку 4.2.

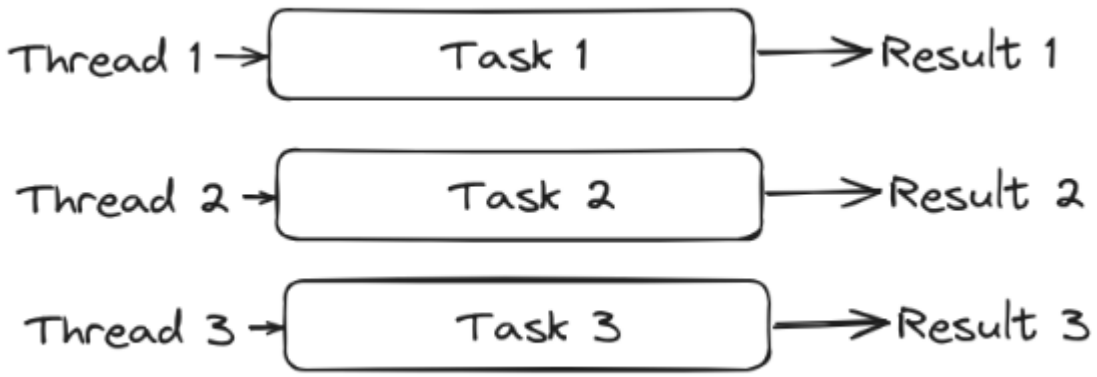


Рисунок 4.2 – Візуалізація багатопотокового / багатопроцесового виконання

Синхронізація доступу до даних є однією з перших проблем, з якою стикаються програмісти при роботі з багатопотоковими програмами. Низькорівневі мови, такі як C, не мають вбудованих примітивів для синхронізації

доступу, тому для цього потрібно використовувати семафори або створювати власні рішення. Проблема полягає в тому, що в багатьох мовах два потоки можуть одночасно читати або писати в одну і ту ж змінну без попередження. Якщо ці ситуації не контролювати, це може призвести до непередбачуваної поведінки програми. Загалом багатопотокові програми складніші та більш схильні до помилок, таких як умови перегонів, тупики і вичерпання ресурсів. Для вирішення цих проблем використовують додаткові механізми, такі як м'ютекси, семафори, тайм-аути тощо.

В асинхронному стилі виконання процес відбувається інакше. Більшість сучасних операційних систем надають підсистеми для роботи з подіями. Програма може попросити операційну систему спостерігати за сокетом і ставити події сповіщення в чергу, доки дані не будуть готові. Програма може перевіряти ці події у зручний для неї час, обробляючи інші задачі в проміжку. Це асинхронний підхід, оскільки програма переходить до виконання наступної команди, не чекаючи результату попередньої[3, с. 35–36] (Рис 4.3)

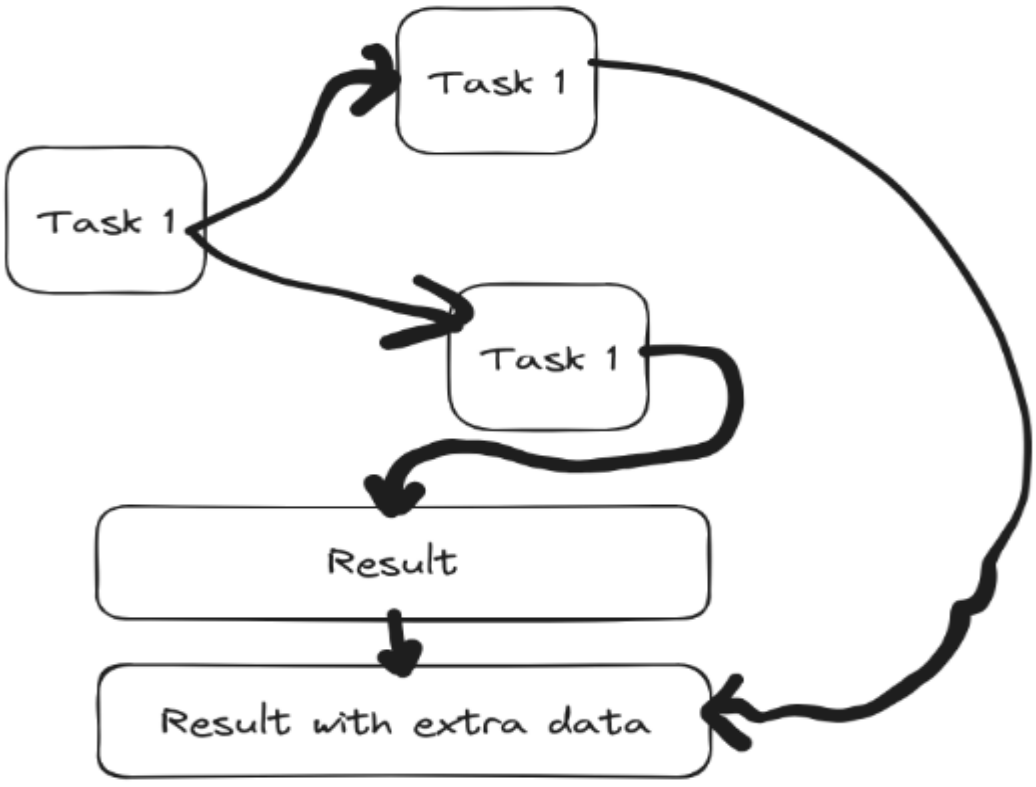


Рисунок 4.3 – Візуалізація асинхронного виконання

Асинхронний код переміщує блокуючу операцію з основного потоку програми, дозволяючи йому продовжувати роботу. Результат цієї операції буде отримано пізніше. Простіше кажучи, основний потік зберігає завдання і відкладе його виконання на більш пізній час.

4.2 Асинхронність та перемикання контексту

Хоча асинхронне програмування може прискорити виконання завдань вводу/виводу і вирішити проблеми синхронізації потоків, його основною метою було вирішення іншої проблеми — частого перемикання контексту процесора. Коли запущено кілька потоків, кожне ядро процесора може виконувати лише один потік одночасно. Щоб всі потоки і процеси могли спільно використовувати ресурси, процесор часто перемикає контекст. Для цього процесор зберігає всю інформацію про потік через певні інтервали часу і перемикається на інший потік. Однак потоки також є ресурсами і не безкоштовними. Часті завантаження даних потоку призводять до збоїв у низькорівневому кеші процесора, що може знизити продуктивність[3, с. 37].

Асинхронне програмування, по суті, є кооперативною багатозадачністю з потоковою обробкою на рівні користувача, де додаток керує потоками і перемикає контекст, а не операційна система[3, с. 37]. У асинхронному середовищі перемикання контексту відбувається лише в певних точках, а не через невизначені проміжки часу. На щастя, керування потоками та перемикання контексту здійснюють сервіси нижчого рівня, а не прикладний код. Це дозволяє відділити бізнес-логіку від таких абстракцій, як особливості операційної системи, платформи та фреймворків. Інженери, що пишуть низькорівневі абстракції, створили зручні інтерфейси для взаємодії з подіями, які відбуваються в процесі життєвого циклу програмного забезпечення. Наприклад, у Node.js використовується абстракція, відома як цикл подій (Event Loop).

4.3 Асинхронність та цикл подій

Як програма може зрозуміти, що з'явилася нова задача, якщо основний потік зайнятий обробкою іншої задачі? Це питання можна вирішити різними способами, в залежності від платформи. Деякі платформи використовують кооперативну багатозадачність, де нові задачі створюються як підпрограми, тоді як інші платформи розділяють перехоплення подій і їх обробку на різні потоки (як на рівні ОС, так і на рівні користувача)[3, с. 40].

Як керувати виконанням всіх подій, пов'язаних як з основною діяльністю, так і з вводу-виводом? Один зі способів - використання планувальника завдань або циклу подій. Ці механізми дозволяють організувати обробку подій таким чином, щоб основний потік міг реагувати на нові задачі і розподіляти ресурси ефективно, забезпечуючи одночасну обробку декількох задач без надмірного перемикання контексту(Рис. 4.4)

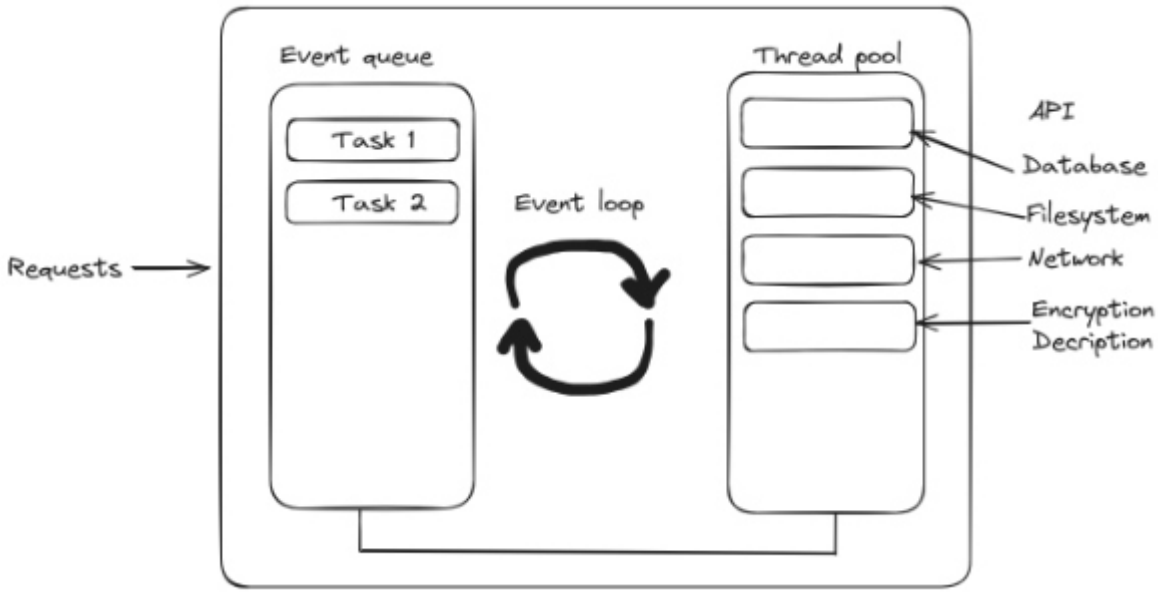


Рисунок 4.4 – Цикл подій

Існує черга подій, де зберігаються всі події, що відбулися, і цикл, який постійно витягує ці події з черги і викликає зворотні виклики (callback) для кожної події. Вся логіка виконання проходить через стек викликів. API надає функції для асинхронних викликів, таких як очікування відповіді від клієнта або бази даних

(наприклад, WebAPI, libuv API тощо)[3, с. 41]. У цьому потоці всі виклики функцій спочатку потрапляють до стеку викликів, а потім асинхронні команди виконуються через API. Після завершення цих команд зворотний виклик поміщається до черги завдань, а потім повертається до стеку викликів. Координація цього процесу відбувається в циклі обробки подій.

Цей підхід схожий на патерн Реактора, про який ми говорили раніше. Насправді, вони ідентичні. Коли цикл обробки подій формує центральний керуючий потік програми, його можна назвати головним циклом або основним циклом обробки подій. Така назва є доречною, оскільки цей цикл знаходиться на найвищому рівні управління всередині програми.

У подієво-орієнтованому програмуванні додаток підписується на певні події і реагує на них, коли вони відбуваються. Цикл обробки подій відповідає за збір подій від операційної системи або моніторинг інших джерел подій. Користувач може зареєструвати функції зворотного виклику, які будуть викликані при виникненні події. Цикл обробки подій зазвичай працює постійно, забезпечуючи безперервний обмін подіями та їх обробку[3, с. 41].

Давайте розглянемо приклад роботи циклу подій. Імпортуємо необхідні для роботи модулі: модуль для читання файлів, модуль для роботи з шляхом до файлів та модуль для роботи з CPU:

```
const fs = require("node:fs/promises");
const os = require("os");
const path = require("node:path");
```

Напишемо функцію, що буде виводити поточний час у стандартному форматі:

```
const logCurrentTime = () => {
  console.log(new Date());
};
```

Напишемо функцію, що буде читати поточний файл:

```
const readCurrentFile = () => {
  fs.readFile(path.resolve(__dirname, "./index.js"), "utf-8")
```

```
    .then(() => {
      console.log("Read");
    })
    .catch((err) => {
      console.log(err);
    });
};
```

Напишемо функцію. що буде виводити кількість ядер поточного процесора:

```
const logCoresCount = () => {
  setTimeout(() => {
    console.log(os.cpus().length);
  }, 10);
};
```

Викличемо наші функції у наступній послідовності та виведемо результат (Рис. 4.5):

```
logCoresCount();
readCurrentFile();
logCurrentTime();
```

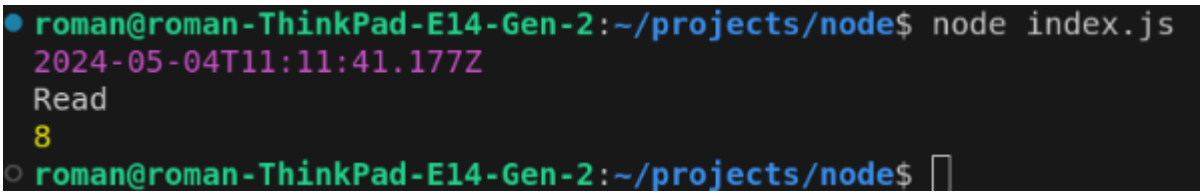


Рисунок 4.5 – Результат виконання коду

Спочатку в чергу виконання потрапить увесь вміст нашого файлу index.js. Далі ми дійдемо до функції logCoresCount, яка містить таймер setTimeout. Інтерпретатор створить нову задачу і помістить її в чергу виконання. Однак сама задача виконається лише тоді, коли поточні задачі з черги будуть виконані. А також час лічильника буде більшим або дорівнюватиме часу, що пройшов від моменту додвання задачі в чергу. Тож ми переходимо далі. Дійшовши до readCurrentFile, інтерпретатор запустить в окремому потоці читання файлу і покладе в чергу

виконання функцію зворотнього виклику, яка відпрацює тоді, коли файл буде прочитано або коли в процесі читання виникне помилка. Далі ми підходимо до функції logCurrentTime, в якій немає такого API, яке б потребувало нових потоків або створення нових задач в черзі. Тому ця функція виконується в рамках поточної задачі. Через це ми і бачимо результат виконання logCurrentTime першим. Як тільки файл прочитався - ми бачимо повідомлення “Read”. А після того ми виконуємо останню задачу із черги - logCoresCount, яка і повертає нам “8”. Бачимо, що ні читання файлу, ні таймер не заблокували нам операцію виводу кількості ядер. Тобто, якщо доведеться читати великий файл, у нас не виникне проблема із затримкою. Змінимо код функції читання файлу:

```
const readCurrentFile = () => {
  fs.readFile(
    path.resolve(__dirname, path.resolve("src", "data.json")),
    "utf-8"
  )
    .then((res) => {
      console.log(res.slice(0, 10));
    })
    .catch((err) => {
      console.log(err);
    });
};
```

У теці src створимо новий файл data.json із даними з відкритих джерел (Рис. 4.6)

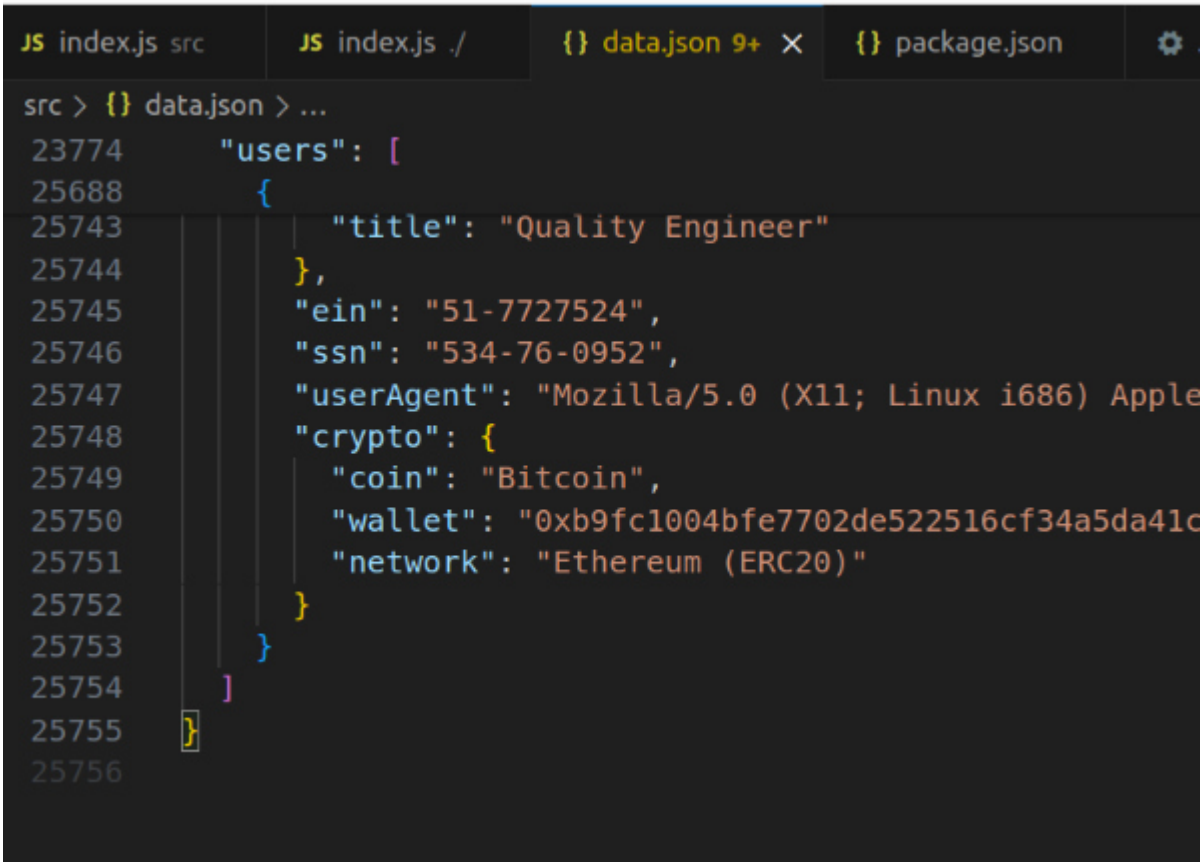


Рисунок 4.6 – Скріншот даних

Як бачимо, файл досить великий. Спробуємо знову запустити наш код (Рис. 4.7)

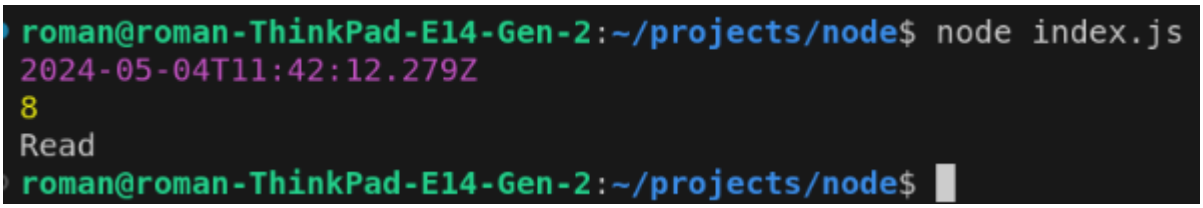


Рисунок 4.7 – Результат виконання коду

Так як розмір файлу збільшився, тепер функція зворотного виклику відпрацювала раніше. Як наслідок, жодна з операцій не блокувала одна одну.

4.4 Висновок

Використання асинхронних операцій у додатках може зробити їх ефективнішими і, головне, швидшими для кінцевого користувача. Потоки ОС

дешевші за процеси, але використання одного потоку для кожної задачі все одно залишається дорогим рішенням. Крім того, існують обмеження на кількість потоків, які можна створити. Повторне використання потоків є більш ефективним, і саме це забезпечує асинхронне програмування.

Асинхронна модель є одним з найважливіших підходів для оптимізації та масштабування додатків, що працюють з вводом/виводом (проте вона не допоможе у випадку завдань, пов'язаних з інтенсивною обробкою процесора). Однак, асинхронні програми можуть бути складними у написанні та виявленні помилок[3, с. 42].

5 ПОРІВНЯННЯ ПІДХОДІВ НА ПРИКЛАДІ ВИРІШЕННЯ ЗАДАЧ

5.1 Повернути число з ряду Фібоначчі

Нехай деякий елемент послідовності Фібоначчі - F(n), де n - його порядковий номер. Тоді F(n) = F(n-1) + F(n-2). Є декілька варіантів, як цю задачу можна описати у вигляді коду. Ми реалізуємо це наступним чином: створимо масив, в який помістимо перші два елементи: одиниці. Після базових перевірок напишемо цикл, що буде на кожній ітерації додавати в масив нове число, що є сумою двох попередніх елементів масиву. Як результат повернемо останній елемент масиву, який і буде розв'язком задачі.

```
const getFibonacciNumbers = (count) => {
  if (typeof count !== "number" || count < 0) {
    return -1;
  }
  const numbers = [1, 1];
  if (numbers[count]) {
    return 1;
  }
  for (let i = 2; i <= count; i++) {
    numbers.push(BigInt(numbers[i - 2]) + BigInt(numbers[i - 1]));
  }
  return numbers[count];
};
```

Тепер спробуємо відтворити ситуацію, коли на кожен запит користувача(виклик функції) ми будемо повертати необхідний елемент ряду Фібоначчі. Зробимо це у синхронному вигляді, багатопотоковому, багатопроцесорному та асинхронному.

Синхронний вигляд (Рис. 5.1):

```
console.time("First request");
```

```
console.time("Second request");
console.time("Third request");
getFibonacciNumbers(210000);
console.timeEnd("First request");
getFibonacciNumbers(210000);
console.timeEnd("Second request");
getFibonacciNumbers(210000);
console.timeEnd("Third request");
```

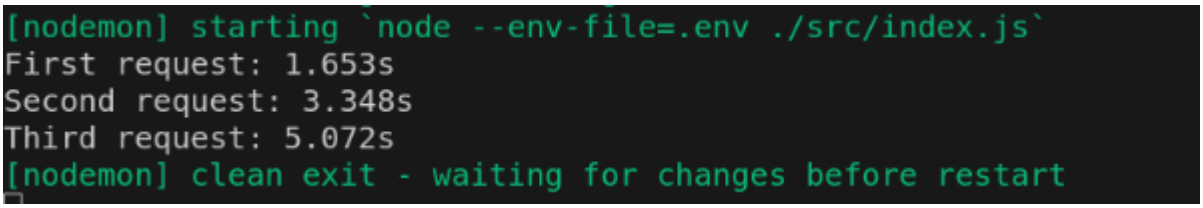


Рисунок 5.1 – Результат синхронної реалізації

Допоки ми передаємо у функцію числа невеликого порядку, все працює добре. Як тільки ми починаємо передавати п’ятирозрядні числа - з’являються затримки, бо процесору необхідно здійснити багато обрахунків, перш ніж функція поверне результат.

Тепер створимо декілька потоків, які будуть обробляти виклики цих функцій.

```
const NEEDED_NUMBER_INDEX = 210000;
const { Worker, isMainThread, parentPort } = require("worker_threads");
if (isMainThread) {
  const workerThreads = [];
  for (let i = 0; i < 4; i++) {
   workerThreads.push(new Worker(__filename));
  }
   workerThreads.forEach((worker) => {
   worker.postMessage({ task: NEEDED_NUMBER_INDEX });
  });
} else {
  parentPort.on("message", (message) => {
   console.log(`Worker ${process.pid}: Received task ${message.task}`);
```

```
        performTask(message.task);
  });
  function performTask(task) {console.time("Request finished"); getFibonacciNumbers(task);
    console.timeEnd("Request finished");
  }
}
```

Результат виведемо у консоль (Рис. 5.2):

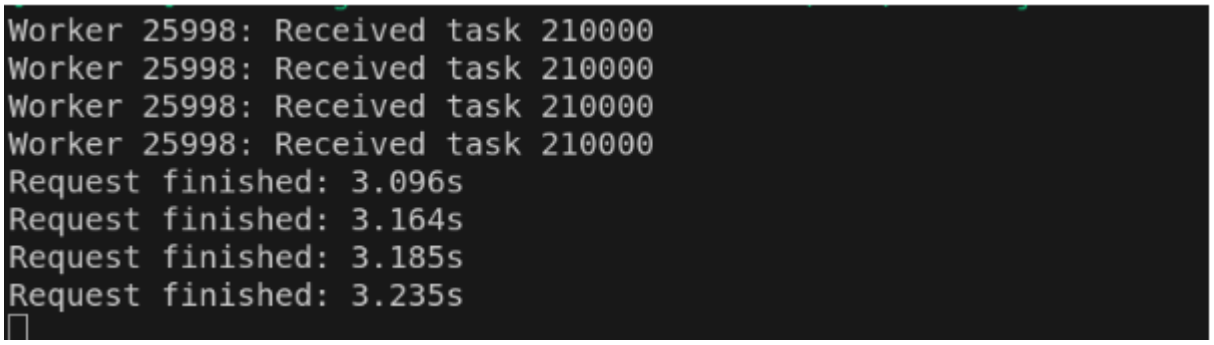


Рисунок 5.2 – Результат багатопотокової реалізації

Бачимо, що знадобився певний час, щоб запустити ці потоки, але потім всі користувачі практично миттєво отримали результати на свої запити, незважаючи на те, що складність обрахунків не зменшилася. Додамо ще пару викликів (Рис. 5.3):

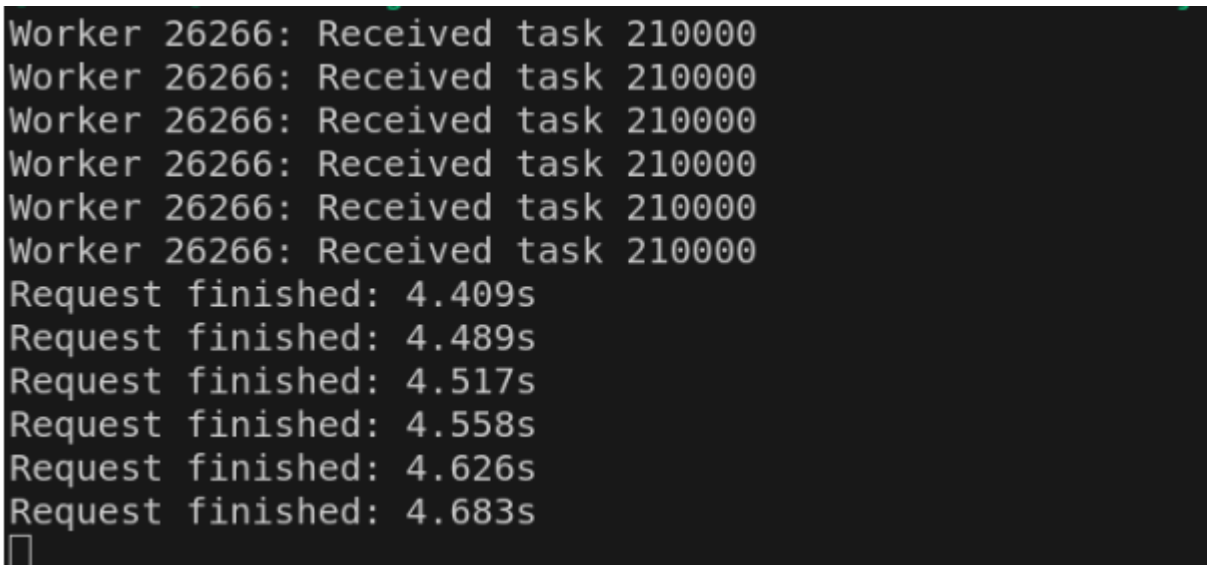


Рисунок 5.3 – Результат багатопотокової реалізації

Бачимо, що останній користувач отримав свій результат через 4,6 секунди. А що було б у синхронному варіанті? Якщо кожен результат приблизно обчислювався 2 секунди, то шостий запит запит отримав би відповідь через 11-12 секунд.

Тепер давайте створимо декілька процесів.

```
const { fork } = require("child_process");
console.time("Time 1");
console.time("Time 2");
console.time("Time 3");
console.time("Time 4");
for (let i = 0; i < 4; i++) {
  const subProcess = fork("src/fibo.js");
  subProcess.on("message", (message) => {
    console.timeEnd("Time " + (i + 1));
  });
}
```

Перенесемо логіку знаходження числа з ряду Фібоначчі в окремий файл і допишемо функцію сповіщення головного процесу, що дочірній процес завершився:

```
const main = () => {
  getFibonacciNumbers(210000);
  process.send("I have finished");
};
main();
```

Виведемо результат у консоль (Рис 5.4):

```
[nodemon] starting `node --env-file=.env ./src/index.js`
Time 3: 3.106s
Time 4: 3.121s
Time 1: 3.136s
Time 2: 3.161s
[nodemon] clean exit - waiting for changes before restart
```

Рисунок 5.4 – Результат багатопроцесорної реалізації

Як бачимо, багатопроцесорний підхід теж добре справляється із цією задачею. А що буде, якщо я збільшу кількість створених процесів? Не раджу це повторювати на своєму домашьому ПК, адже він припинить виконання зі схожою помилкою (Рис. 5.5):

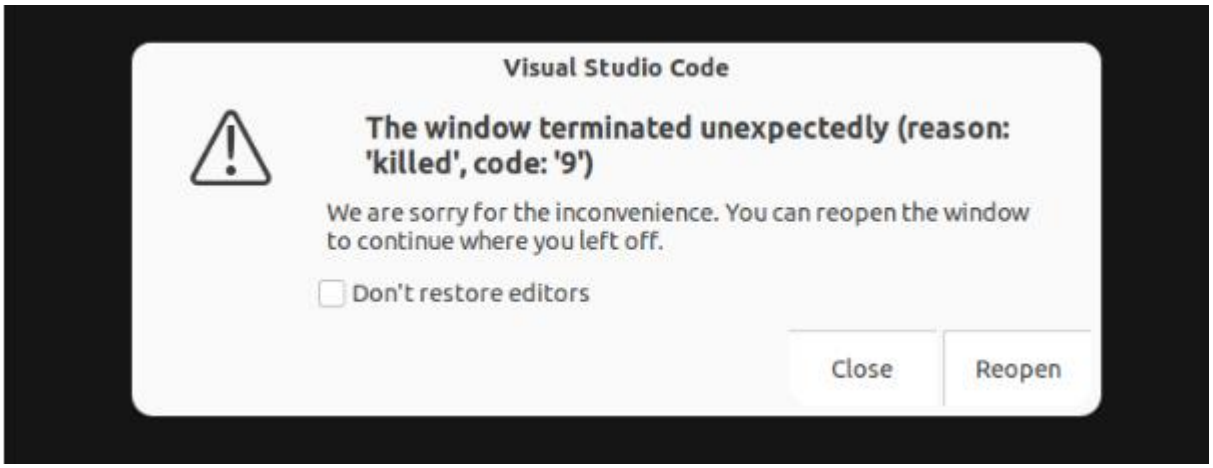


Рисунок 5.5 – Аварійне завершення роботи програми

Як ми вже згадували, процеси набагато дорожчі в обслуговуванні з точки зору ресурсів, тому їх потрібно створювати виключно при реальній необхідності в ізоляції. А як щодо асинхронного коду?

Огорнемо функцію getFibonacciNumbers в асинхронну обгортку:

```
const asyncWrapper = ({ callback, time }) => {
  return (...arguments) => {
    return new Promise((res, rej) => {
      setTimeout(() => {
        try {
          res(callback(...arguments));
        } catch (err) {
          rej(err);
        }
      }, time || 0);
    });
```

```
  };
};
```

Як і в попередніх прикладах, спочатку протестуємо результат на чотирьох викликах:

```
const getFibonacciNumbersAsync = asyncWrapper({
  callback: getFibonacciNumbers,
  time: 0,
});
console.time("First request");
console.time("Second request");
console.time("Third request");
console.time("Fourth request");
getFibonacciNumbersAsync().then(() => console.timeEnd("First request"));
getFibonacciNumbersAsync().then(() => console.timeEnd("Second request"));
getFibonacciNumbersAsync().then(() => console.timeEnd("Third request"));
getFibonacciNumbersAsync().then(() => console.timeEnd("Fourth request"));
console.log("Stared to work hard");
```

Виведемо результат у консоль (Рис. 5.6):

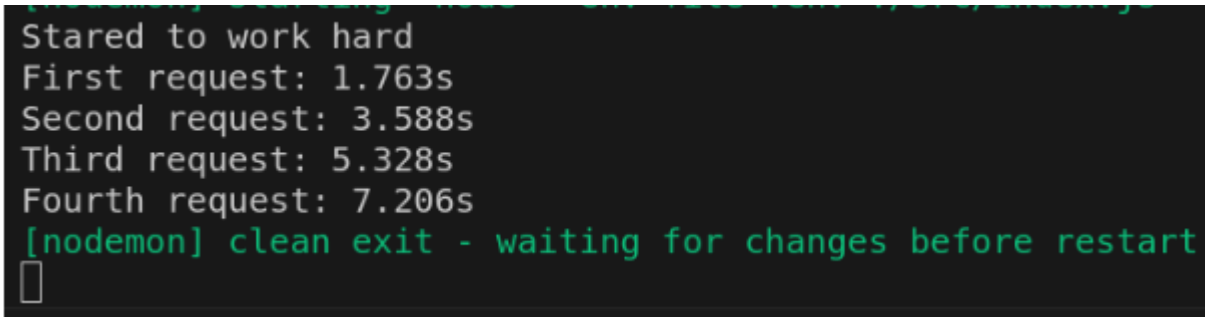


Рисунок 5.6 – Результат асинхронної реалізації

Хоча повідомлення "Stared to work hard" вивелося раніше за результати виконання нашої функції, вони відпрацювали послідовно, хоч і асинхронно, адже ми знаходилися в рамках одного потоку/процесу. Тобто в даному випадку асинхронність практично не показала жодних переваг перед синхронністю, зате сильно відстала від багатопроцесорності/багатопотоковості.

Висновок: у задачах, де основне навантаження йде на процесор, і систем вводу-виводу небагато, варто використовувати багатопроцесорний підхід.

5.2 Видати текстові файли документації для відділу маркетингу

Уявимо ситуацію: десь на диску лежать файли документації до деякого проекту. Час від часу відділ маркетингу, що складається з 8-ми осіб, хоче прочитати ці файли. На жаль, у нас немає користувацького інтерфейсу, щоб вирішити цю проблему. Але є сервер на NodeJS, який ми можемо використати. Напишемо програму, що буде приймати запит на читання файлу та повертатиме його вміст.

Створимо саму функцію читання файлу:

```
const { readFileSync } = require("node:fs");
const readFileFormDisk = (filePath) => {
  try {
    return readFileSync(filePath, "utf-8");
  } catch (err) {
    console.error(err);
    return null;
  }
};
```

Створимо імітацію серверу, який буде віддавати вміст файлів:

```
const createServer = () => {
  return {
    getFile(filePath) {
      const file = readFileFormDisk(filePath);
      if (!file) {
        return "File not found";
      }
```

```
      return file;
    },
  };
};
```

Тепер змоделюємо найгірший сценарій: усі вісім працівників відділу маркетингу захотіли прочитати якийсь файл.

Синхронний варіант:

```
const server = createServer();
for (let i = 0; i < 8; i++) {
  console.log(server.getFile("./src/data.txt"));
}
```

Вивід у консоль (Рис. 5.7):

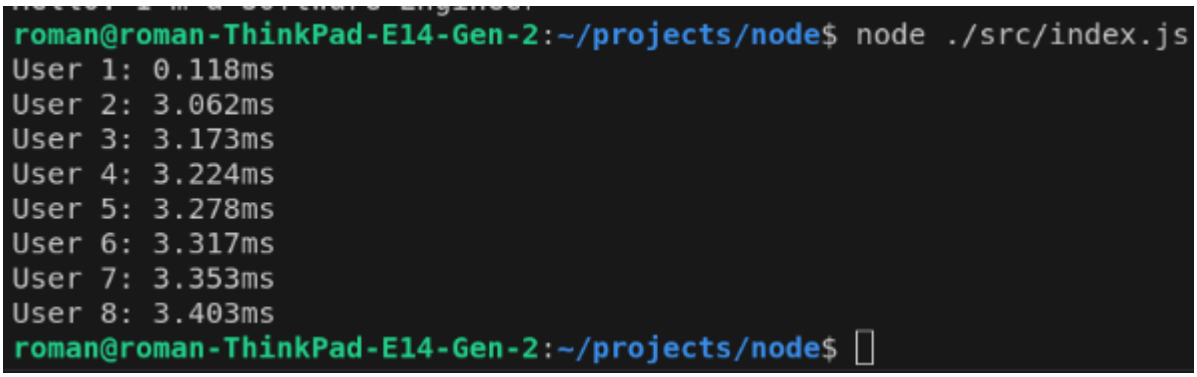


Рисунок 5.7 – Результат синхронної реалізації

Бачимо, що цей метод добре справився з маленький текстовим файлом. Давайте подивимося, що буде, якщо ми збільшимо ємність цього файлу (Рис. 5.8)

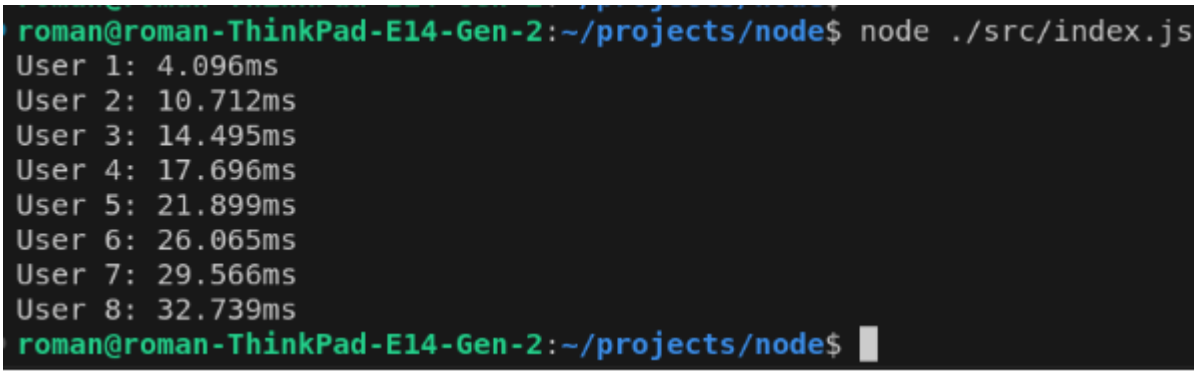


Рисунок 5.8 – Результат синхронної реалізації

Частка між першим і останнім користувачем склала 8 разів(відповідно до кількості користувачів).

Подивимося, як зміняться показники при багатопотоковому підході. Створимо вісім потоків, у яких будемо виконунавати функцію зчитування файлу:

```
const { Worker } = require("worker_threads");
for (let i = 0; i < 8; i++) {
  console.time(`Process ${i} has finished`);
  const worker = new Worker("./src/server.js", {
    workerData: {
      filePath: "./src/data.txt",
    },
  });
  worker.on("message", (result) => {
    console.timeEnd(`Process ${i} has finished`);
  });
  worker.on("error", (msg) => {
    console.log(msg);
  });
}
```

Перенесемо код створення нашого серверу в файл server.js та додамо наступні рядки для повернення результату в основний потік виконання:

```
const { parentPort, workerData } = require("worker_threads");
parentPort.postMessage(server.getFile(workerData.filePath));
```

Виведемо результат у консоль (Рис. 5.9):

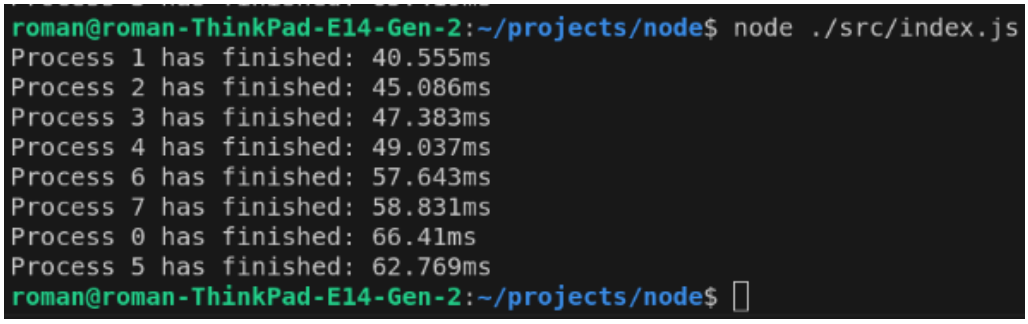


Рисунок 5.9 – Результат багатопотокової реалізації

Бачимо, що при такому ж об’ємі даних даних частка між першим та останнім користувачем становить 1.55, а не 8. Хоча, варто взяти до уваги той факт, що час першого запуску програми збільшився у 10 разів. Тобто цей метод можна брати до уваги лише при значних розмірах файлів, які ми будемо читати та передавати по мережі. З багатопроцесорних методом трохи складніше: на моєму поточному пристрої 8 ядер. Тобто я не можу запустити одночасно всі 8 процесів, адже в мене просто не вистачить ресурсів апаратного забезпечення. Проте, в конкретних умовах, ми можемо створити чотири процеси і приймати по два запити на кожен процес, що дає підстави вважати, що затримка між першим та останнім користувачем збільшиться. Створимо 4 процеси та змусимо їх обробляти по 2 користувача:

```
const { fork } = require("child_process");
console.time("Time 1"); // 1-8
const subProcess1 = fork("src/server.js"); // 1-4
subProcess1
  .on("message", () => {
    console.timeEnd("Time 1");
  })
  .on("message", () => {
    console.timeEnd("Time 5");
  }); // 1-4
```

Оновимо код нашого серверу:

```
const { readFileSync } = require("node:fs");
```

```
const readFileFormDisk = (filePath) => {
 try {
  return readFileSync(filePath, "utf-8");
 } catch (err) {
  console.error(err);
  return null;
 }
};
const createServer = () => {
 return {
  getFile(filePath) {
   const file = readFileFormDisk(filePath);
   if (!file) {return "File not found"}
   return file;
  },
 };
};
const server = createServer();
const main = () => { server.getFile("./src/data.txt"); process.send("I have finished")};
main();
```

Виведемо результат у консоль (Рис. 5.10):

```
roman@roman-ThinkPad-E14-Gen-2:~/projects/node$ node ./src/index.js
Time 1: 29.262ms
Time 5: 31.985ms
Time 2: 32.437ms
Time 6: 32.599ms
Time 4: 40.804ms
Time 8: 40.969ms
Time 3: 41.113ms
Time 7: 41.182ms
roman@roman-ThinkPad-E14-Gen-2:~/projects/node$
```

Рисунок 5.10 – Результат багатопроцесорної реалізації

Частка між першим і останнім запитом складає 1.4, тобто майже така ж, як і при багатопотоковому підході. Проте, варто пам’ятати, що при збільшенні

користувачів ми не зможемо збільшувати кількість процесів так само, як кількість потоків. Наостанок реалізуємо програму асинхронним методом.

```
const { readFileSync } = require("node:fs");
const readFileFormDisk = (filePath) => {return readFileSync(filePath, "utf-8")};
const createServer = () => {
 return {
  getFile(filePath) {
   return new Promise((res, rej) => {
    setTimeout(() => {try {res(readFileFormDisk(filePath))} catch {rej("File not found")}});
   });
  },
 };
};
const server = createServer();
const main = async (id) => {
 console.time("Finished " + id);
 await server.getFile("./src/data.txt");
 console.timeEnd("Finished " + id);
};
for (let i = 0; i < 8; i++) {main(i + 1)}
```

Виведемо результат у консоль (Рис. 5.11):

```
roman@roman-ThinkPad-E14-Gen-2:~/projects/node$ node ./src/server.js
Finished 1: 8.266ms
Finished 2: 18.472ms
Finished 3: 25.446ms
Finished 4: 31.67ms
Finished 5: 38.567ms
Finished 6: 45.41ms
Finished 7: 52.061ms
Finished 8: 60.266ms
roman@roman-ThinkPad-E14-Gen-2:~/projects/node$
```

Рисунок 5.11 – Результат асинхронної реалізації

Частка між першим і останнім запитами склала 7.5. Тобто, ми отримали такий же результат, як і при синхронному форматі. Єдина відмінність у тому, що у нам не заблокований основний потік і ми можемо робити інші операції, крім читання. Але ми застосували лише одну з можливостей асинхронного коду – функції

зворотнього виклику. Давайте замінимо функцію readFileSync на readFile, що дасть нам усі переваги патерну Проактор, який реалізований у платформі NodeJS. Поглянемо на результат тепер (Рис 5.12):

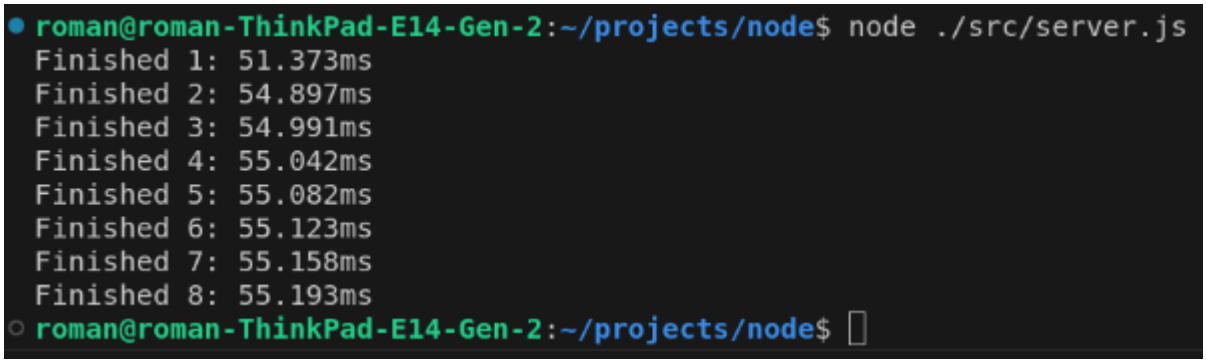


Рисунок 5.12 – Результат асинхронної реалізації

Перший результат ми віддали з певною затримкою. Але тепер частка між першим і останнім запитом складає всього 1.07, що є найкращим результатом.

Виснокок: щоб обрати в даній ситуації правильний метод написання роботи з вводом-виводом, варто ознайомитися з вимогами. Якщо нам краще віддавати результат якнайшвидше, то краще обрати багатопотоковість. Якщо нам важливо, скільки будуть чекати користувачі, які зробили запит останніми, то варто написати код в асинхронному стилі.

5.3 Реалізувати http-сервер, що буде повертати дані з БД

Синхронний варіант можна відкинути одразу, так як ми не знаємо, скільки буде користувачів. З цієї ж причини ми не можемо одразу обрати стратегію для створення процесів/потоків(створення на кожен новий запит чи розподіл навантаження між фіксованою кількістю). Також ми не знаємо, скільки часу буде займати запит до бази даних. Тому в цьому випадку варто одразу застосувати асинхронний підхід.

Створимо http-сервер:

```
const server = http.createServer((request, response) => {
  const { headers, method, url } = request; let body = [];
```

```
  request
    .on("error", (err) => {console.error(err)})
    .on("data", (chunk) => {body.push(chunk)})
    .on("end", async () => {body = Buffer.concat(body).toString();
      response.on("error", (err) => {console.error(err)});
      response.statusCode = 200;
      response.setHeader("Content-Type", "application/json");
     const responseBody = { headers, method, url, body: await getDataFromDB()};
      response.write(JSON.stringify(responseBody));
      response.end();
    });
});
server.listen(8000);
```

Тут хотів би звернути увагу на метод .on. Цей патерн наслідується з концепції Event-Driven Design. В поєднанні з патерном Proactor він дає можливість приймати запит від клієнта асинхронно, не блокуючи основний потік виконання. При чому, нам не важливо, скільки даних відправив користувач, адже вони будуть підвантажуватися частинами(чанками). Подія .on (“data”...) буде спрацьовувати кожного разу, коли підвантажиться наступна частина даних від користувача.

Тепер реалізуємо функцію-заглушку, що буде імітувати запит до бази даних:

```
const getDataFromDB = () => {
  return new Promise((res) => {
    setTimeout(() => {
      res("JavaScript");
    }, 3000);
  });
};
```

Додамо її у місце, яке відповідає за повернення результату користувачеві:

```
.on("end", async () => {
     body = Buffer.concat(body).toString();
```

```
    response.on("error", (err) => {
      console.error(err);
    });
    response.statusCode = 200;
    response.setHeader("Content-Type", "application/json");
    const responseBody = {headers, method, url, body: await getDataFromDB()};
    response.write(JSON.stringify(responseBody));
    response.end();
  });
```

Напишемо код, який буде робити запити на наш сервер.

Результат тестового запуску на рис. 5.13, а листинг програми у додатку А.

Рисунок 5.13 – Результат асинхронної реалізації

Все працює. Тепер збільшимо кількість запитів, пам’ятючи про те, що запит до бази даних займає три секунди. Для замірювання часу скористаємося конструкцією виду:

```
const time = Date.now();
…
console.log("Time: " + (Date.now() - time) / 1000);
```

Виведемо результат у консоль (Рис. 5.14):

```
roman@roman-ThinkPad-E14-Gen-2:~/projects/node$ node ./src/index.js
Time: 3.026
Time: 3.027
Time: 3.027
Time: 3.027
Time: 3.028
Time: 3.028
Time: 3.028
Time: 3.028
```

Рисунок 5.14 – Результат асинхронної реалізації

Як бачимо, запити одних користувачів не заважають запитам інших. Збільшимо кількість запитів до 100 (Рис. 5.15):

```
Time: 3.051
Time: 3.051
Time: 3.051
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.054
Time: 3.054
Time: 3.054
Time: 3.054
Time: 3.054
Time: 3.054
Time: 3.054
Time: 3.054
```

Рисунок 5.15 – Демонстрація роботи серверу із навантаженням у 100 користувачів

Висновок: асинхронна реалізація дозволяє нам не турбуватися про кількість користувачів, про час відповіді від бази даних, адже всю цю роботи ми скинули з основного потоку і готові виконувати реально важливі задачі.

6 РЕКОМЕНДАЦІЇ ЩОДО ВИКОРИСТАННЯ

Потоки можуть бути зручними, коли у вас стандартний веб-додаток з невеликою кількістю користувачів та коротким часом відгуку. Однак асинхронність стає вигідною, якщо ваш додаток проводить більше часу на операціях вводу/виводу, ніж на їх обробці. Наприклад, якщо ви працюєте з великою кількістю повільних запитів, таких як веб-сокети або довгий полінг, або з повільними зовнішніми бекендами, про які неможливо передбачити час завершення. Синхронне програмування - це простий спосіб почати розробку додатків, де виконання команд відбувається послідовно. У цьому підході розробники дбають про код так, щоб виконувати кожен крок за раз, навіть при використанні умовних операторів, циклів та функцій. У випадку асинхронного програмування система також працює крок за кроком, але вона не чекає завершення поточного кроку виконання. Для кращого розуміння цієї концепції можна намалювати діаграму (Рис. 6.1)

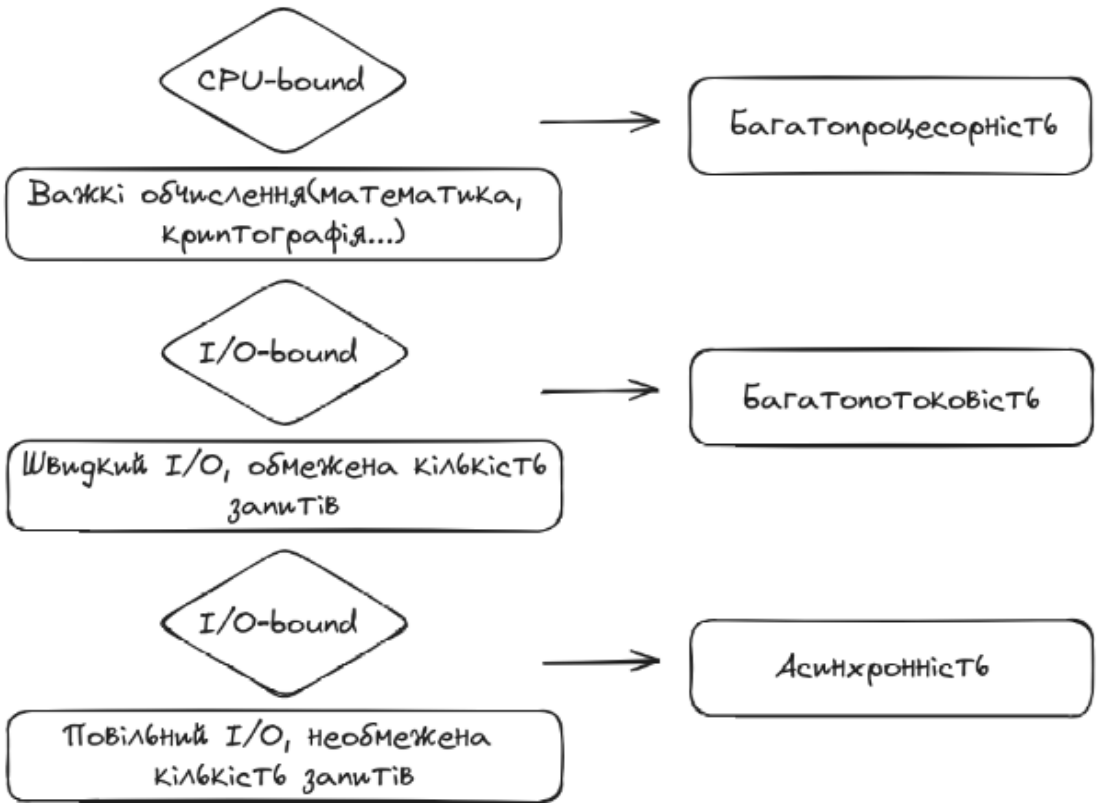


Рисунок 6.1 – Рекомендації по імплементації

ВИСНОВОК

Зі збільшенням інтеграцій та навантаження у промислових застосунках системи вводу та виводу даних стали невід'ємними складовими комп'ютеризованих інформаційних систем (ІС) будь-якої складності та архітектури: від систем, для роботи яких вистачає одного комп'ютера, до веб-орієнтованих та багатосерверних систем. Система вводу та виводу інформації відповідає за те, яким чином данні надходять до інформаційної системи від постачальників даних, та яким чином данні відправляються до них назад. Існує кілька розповсюджених схем обміну даними, серед яких основними є синхронна та асинхронна, багатопроцесорна та багатопотокові архітектури. Всі вони мають як переваги так і недоліки в контексті проектування конкретних інформаційних систем. Тому часто виникає проблема вибору системи вводу-виводу під час створення конкретної ІС для застосування в конкретних умовах. Основною метою дослідження було зробити є огляд та аналіз різних систем вводу-виводу та розробити рекомендації для використання кожної з них під час створення ІС різного призначення та для роботи в різних умовах.

Для виконання мети було проаналізовано всі базові принципи роботи із системами вводу/виводу, створено імплементацію на мові програмування JavaScript з використанням платформи NodeJS та перевірено швидкодію кожного методу на прикладі підібраних задач. Було переглянуто недоліки та переваги кожної імпементації. У результаті дослідження було сформовану рекомендації щодо впровадження кожного методу у конкретних ситуаціях. Було зроблено вимірювання та порівняння швикодії всіх базових імплементацій методів по роботі із системами вводу та виводу.

ПЕРЕЛІК ПОСИЛАНЬ

1. Maarten van Steen, Andrew S. Tanenbaum Distributed Systems: Principles and Paradigms // Maarten van Steen, Рік видання: 2023. – 70 с.

2. Andrew Tanenbaum, Herbert Bos Modern Operating Systems, 5th Global Edition // Pearson, Year: 2024 – 39 с.

3. Kiril Bobrov. Asynchronous Programming // Рік видання: 2021 – 4-63 с.

4. NodeJS official documentation // [Електронний ресурс]: https://nodejs.org/en/learn/asynchronous-work/asynchronous-flow-control
Дата звернення: 01.05.2024

5. JavaScript official documentation // [Електронний ресурс]: https://tc39.es/ecma262/
Дата звернення: 01.05.2024

6. Gudula Rünger, Thomas Rauber Parallel Programming: for Multicore and Cluster Systems // Рік видання: 2023. – 20 с.

ДОДАТОК А

Листинг програми

```
http
  .createServer((request, response) => {
    const { headers, method, url } = request;
    let body = [];
    request
      .on("error", (err) => {
        console.error(err);
      })
      .on("data", (chunk) => {
        body.push(chunk);
      })
      .on("end", async () => {
        body = Buffer.concat(body).toString();
        response.on("error", (err) => {
          console.error(err);
        });
        response.statusCode = 200;
        response.setHeader("Content-Type", "application/json");
        const data = await getDataFromDB();
        const responseBody = { headers, method, url, body: data };
        response.write(JSON.stringify(responseBody));
        response.end();
      });
  })
  .listen(8080, "localhost");

http
```

```
  .get("http://localhost:8000/", (res) => {
    const { statusCode } = res;
const contentType = res.headers["content-type"];

    let error;
    if (statusCode !== 200) {
      error = new Error("Request Failed.\n" + `Status Code: ${statusCode}`);
    } else if (!/^application\/json/.test(contentType)) {
      error = new Error(
        "Invalid content-type.\n" +
          `Expected application/json but received ${contentType}`
      );
    }
    if (error) {
      console.error(error.message);
      // Consume response data to free up memory
      res.resume();
      return;
    }

    res.setEncoding("utf8");
    let rawData = "";
    res.on("data", (chunk) => {
      rawData += chunk;
    });
    res.on("end", () => {
      try {
        const parsedData = JSON.parse(rawData);
        console.log(parsedData);
      } catch (e) {
```

```
      console.error(e.message);
    }
  });
})
.on("error", (e) => {
  console.error(`Got error: ${e.message}`);
});
```

Державний університет інформаційно-комунікаційних технологій

Розробка рекомендацій для вибору асинхронного чи паралельного програмування в бізнес-задачах

Виконав студент групи КНД-42
Крентовський Роман Сергійович
Київ, 2024

Актуальність

Системи вводу та виводу даних є невід’ємними складовими комп’ютеризованих інформаційних систем (ІС) будь-якої складності та архітектури: від систем, для роботи яких вистачає одного комп’ютера, до веб-орієнтованих та багатосерверних систем.

Існує кілька розповсюджених схем обміну даними, серед яких основними є синхронна та асинхронна, багатопроцесорна та багатопотокові архітектури. Всі вони мають як переваги так і недоліки в контексті проектування конкретних інформаційних систем. Тому часто виникає проблема вибору системи вводу-виводу під час створення конкретної ІС для застосування в конкретних умовах.

Мета

Огляд та аналіз різних систем вводу-виводу та розробка рекомендацій для використання кожної з них під час створення ІС різного призначення та для роботи в різних умовах.

Завдання

1. Дослідити, яким чином відбувається взаємодія між основним кодом програми та системою вводу-виводу.

2. Проаналізувати такі поняття як конкурентність(concurrency), паралелізм(parallelism). Розглянути методи роботи із системою вводу-виводу, які базуються на цих поняттях.

3. Розглянути на прикладах, які результати показує кожен із методів роботи із системами вводу-виводу.

4. Сформувати рекомендації по використанню кожного з методів

Взаємодія між основним кодом та системою вводу-виводу

Чотири способи взаємодії:
1) блокуючий синхронний;
2) блокуючий асинхронний;
3) неблокуючий синхронний;
4) неблокуючий асинхронний.

Поняття, що базуються на способах взаємодії.
Методи, що реалізують ці способи.

Поняття

- Паралелізм(Parallelism)
- Конкурентність(Concurrency)

Методи

- Багатопроцесорність
- Багатопотоковість
- Асинхронність

Порівняння підходів на прикладі вирішення задач

- Повернути число з ряду Фібоначчі.
- Сервер для читання та відправки текстових файлів.
- HTTP-сервер, що буде повертати дані з БД.

Задача 1. Числа Фібоначчі

Синхронний метод

```
[nodemon] starting `node --env-file=.env ./src/index.js`
First request: 1.653s
Second request: 3.348s
Third request: 5.072s
[nodemon] clean exit - waiting for changes before restart
```

Багатопотоковий метод

```
Request finished: 4.409s
Request finished: 4.489s
Request finished: 4.517s
Request finished: 4.558s
Request finished: 4.626s
```

Багатопроцесорний метод

```
[nodemon] starting `node --env-file=.env ./src/index.js`
Time 3: 3.106s
Time 4: 3.121s
Time 1: 3.136s
Time 2: 3.161s
```

Асинхронний метод

```
Stared to work hard
First request: 1.763s
Second request: 3.588s
Third request: 5.328s
Fourth request: 7.206s
```

Задача 2. Відправка текстових файлів

Синхронний метод

```
User 1: 4.096ms
User 2: 10.712ms
User 3: 14.495ms
User 4: 17.696ms
User 5: 21.899ms
User 6: 26.065ms
User 7: 29.566ms
User 8: 32.739ms
```

Багатопотоковий метод

```
Process 1 has finished: 40.555ms
Process 2 has finished: 45.086ms
Process 3 has finished: 47.383ms
Process 4 has finished: 49.037ms
Process 6 has finished: 57.643ms
Process 7 has finished: 58.831ms
Process 0 has finished: 66.41ms
Process 5 has finished: 62.769ms
```

Багатопроцесорний метод

```
Time 1: 29.262ms
Time 5: 31.985ms
Time 2: 32.437ms
Time 6: 32.599ms
Time 4: 40.804ms
Time 8: 40.969ms
Time 3: 41.113ms
Time 7: 41.182ms
```

Асинхронний метод

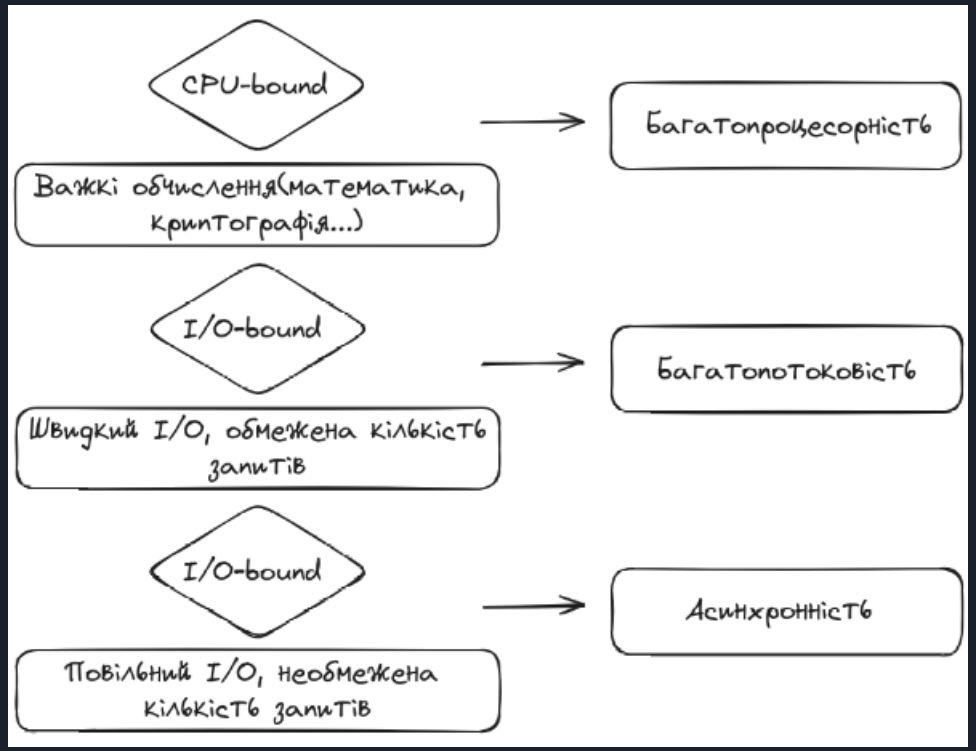
```
Finished 1: 51.373ms
Finished 2: 54.897ms
Finished 3: 54.991ms
Finished 4: 55.042ms
Finished 5: 55.082ms
Finished 6: 55.123ms
Finished 7: 55.158ms
Finished 8: 55.193ms
```

Задача 3. HTTP-сервер

Лише асинхронний метод

```
Time: 3.051
Time: 3.051
Time: 3.051
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.052
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.053
Time: 3.054
Time: 3.054
Time: 3.054
Time: 3.054
Time: 3.054
Time: 3.054
Time: 3.054
Time: 3.054
```

Рекомендації



Висновки

1. В бакалаврській роботі проведено аналіз методів роботи із системою вводу-виводу.
2. Досліджено імплементації цих методів на платформі NodeJS
3. Розглянуто переваги та недоліки кожної імплементації на конкретних задачах.
4. Сформовано рекомендації щодо застосування конкретної імплементації для вирішення типових бізнес-задач.