НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра технічної кібернетики

Звіти до комп’ютерних практикумів з кредитного модуля

“Parallel computing”

**Виконав**

**Студенти групи ІТ-02**

**Терешкович М.О.**

**Перевірив:**

Київ – 2023

**Комп‘ютерний практикум No 4**

**Розробка паралельних програм з використанням пулів потоків, екзекьюторів та ForkJoinFramework**

ЗАВДАННЯ

Зображення, що містить текст

Автоматично згенерований опис

Зображення, що містить текст

Автоматично згенерований опис

**ВИКОНАННЯ**

# Завдання 1

A screenshot of a computer

Description automatically generated with low confidenceFВ першому завданні в проекті присутні 4 файли:

A screenshot of a computer program

Description automatically generated with medium confidenceПочнемо з файлу Main:

Перша частина коду визначає два алгоритми: "SyncAlgo" і "ParallelAlgo", які знаходяться в іншому класі під назвою "Algorithms" – про нього буде далі. Ці алгоритми отримують шлях до текстового файлу як параметр і обробляють вміст файлу різними способами. "SyncAlgo" використовує синхронізовані методи для забезпечення безпеки потоків, тоді як "ParallelAlgo" використовує декілька потоків для одночасної обробки файлу.

Головний метод класу "Main" викликає метод "measureTime" двічі, передаючи кожен алгоритм і рядок, що вказує на тип алгоритму, який виконується. Метод "measureTime" приймає об'єкт, що виконується, і рядок як параметри. Він вимірює і виводить час виконання, що минув, разом з параметром рядка.

Метод "Print" отримує параметром об'єкт HashMap. Цей метод обчислює різні статистичні показники вмісту текстового файлу. Він підраховує загальну кількість слів і символів, обчислює середню довжину слова, дисперсію довжин слів і стандартне відхилення довжин слів. Потім вона виводить цю статистику на друк.

Константа "ITTER" має значення 500,000, що вказує на кількість ітерацій через які алгоритм ForkJoint розділиться на сабтаски. Константа "PATH" задає шлях до текстового файлу, який буде оброблено.

Тепер перейдемо до файлу Alghorithms:

В цьому файлі знаходяться два алгоритми, які згадувалися до цього – “ SyncAlgo” та "ParallelAlgo".



Цей метод, отримує шлях до каталогу на вході, переглядає всі файли в каталозі (включно з вкладеними підкаталогами) і обробляє вміст кожного файлу. Він використовує синхронізований підхід для забезпечення безпеки потоків.

Спочатку створюється порожній масив ArrayList з назвою "filePaths". Потім він використовує метод Files.walk для рекурсивного обходу каталогу і отримання списку всіх звичайних файлів у каталозі. Кожен шлях до файлу додається до масиву "filePaths".

Далі створюється об'єкт HashMap з назвою "result" для зберігання частот довжини слова. Потім алгоритм циклічно проходить по кожному шляху до файлу у списку "filePaths". Для кожного файлу він зчитує його вміст у рядкову змінну з назвою "content" за допомогою методу Files.readString. Вміст розбивається на масив слів за допомогою методу split з регулярним виразом "\s+", який відповідає одному або декільком пробілам.

Для кожного слова в масиві алгоритм обчислює довжину слова за допомогою методу length і збільшує значення відповідного ключа в "результуючій" HashMap. Якщо ключ не існує в HashMap, він додається зі значенням за замовчуванням 0.

Якщо під час читання файлу виникає виключення IOException, метод генерує виключення RuntimeException, причиною якого є виключення, що виникло.

Нарешті, "результуюча" HashMap передається методу Print класу Main для обчислення та друку статистичних показників вмісту файлів.

Далі йде метод ParallelAlgo:

A picture containing text, screenshot, font, software

Description automatically generated

Цей метод, отримує на вхід шлях до каталогу, створює екземпляр ForkJoinPool за допомогою методу ForkJoinPool.commonPool і викликає ForkJoinTask з назвою "DirectorySizeAnalysis" з вказаним шляхом до каталогу на вході. Про "DirectorySizeAnalysis" буде далі.

Задача "DirectorySizeAnalysis" є рекурсивною задачею, яка проходить через каталог і підкаталоги для отримання списку всіх файлів, а потім обчислює частоти довжини слів у файлах за тим же алгоритмом, що і в методі "SyncAlgo". Однак ця задача розподіляє робоче навантаження між декількома потоками в ForkJoinPool для підвищення продуктивності.

Метод "invoke" ForkJoinPool блокується до тих пір, поки завдання не завершиться, і повертає результат. Потім результат передається в метод "Print" класу "Main" для обчислення і друку статистичних показників вмісту файлів.

Нарешті, пул закривається за допомогою методу "shutdown".

A screen shot of a computer program

Description automatically generated with medium confidenceТепер перейдемо до файду FileSizeAnalysis:

A picture containing text, screenshot

Description automatically generated

Це клас FileSizeAnalysis, який розширює RecursiveTask<HashMap>, що вказує на те, що цей клас буде повертати HashMap з цілочисельними ключами та значеннями.

Клас має два конструктори. Перший отримує рядковий параметр filePath, який є шляхом до файлу, який буде проаналізовано. Другий отримує чотири параметри: filePath, список слів List і два цілих числа, які представляють початковий і кінцевий індекси підсписку слів, що аналізуються.

Метод compute є основним методом задачі. Якщо список слів не ініціалізовано, він викликає метод initWordsList для читання вмісту файлу і створення списку слів. Якщо підсписок слів для аналізу менший за визначену константу ITTER(тобто якщо менше за 500 000 символів), викликається метод getWordsData, який обробляє слова з підсписку і повертає HashMap з довжиною кожного слова і його частотою. Якщо підсписок більший за ITTER, метод розбиває підсписок на дві половини і створює дві підзадачі, по одній для кожної половини. Ліва підзадача розгалужується, а права обчислюється. Потім результат правої підзадачі об'єднується з результатом лівої підзадачі і повертається об'єднаний результат. І тепер два останні методи з цього файлу:

Це getWordsData та initWordsList.

* getWordsData - Цей метод отримує підсписок wordsList і обчислює довжину кожного слова у підсписку, зберігаючи підрахунок довжини кожного слова у HashMap. Він ітерує кожне слово у підсписку, обчислює його довжину за допомогою методу length() і відповідно оновлює лічильник у хеш-карті. Якщо хеш-карта вже містить довжину слова як ключ, вона збільшує його значення на 1, інакше вона додає довжину слова до хеш-карти з початковим значенням 1. Нарешті, вона повертає хеш-карту з підрахунком довжини слова.
* A picture containing text, screenshot, font

  Description automatically generatedinitWordsList - використовується для читання вмісту файлу, вказаного в filePath, розбиття вмісту на слова і присвоєння отриманого списку слів полю wordsList об'єкта FileSizeAnalysis. Метод використовує Files.readString() для читання вмісту файлу у змінну String, а потім List.of() для розбиття вмісту на слова і повернення їх у вигляді незмінного списку. Початкове та кінцеве поля встановлюються у 0 та розмір wordsList відповідно, що вказує на те, що має бути оброблено весь список. Якщо під час читання файлу виникає виключення вводу-виводу, на консоль виводиться повідомлення про помилку і генерується виключення часу виконання (RuntimeException).

Залишився останній файл DirectorySizeAnalysis:

Це клас, який розширює клас RecursiveTask, а це означає, що він призначений для виконання обчислень, які повертають результат, і ці обчислення можна розбити на менші завдання, які можна виконувати паралельно.

У цьому конкретному випадку клас DirectorySizeAnalysis відповідає за аналіз файлів у заданому каталозі і повертає хеш-мапу, яка відображає довжину кожного слова у відповідності до кількості разів, коли воно з'являється у каталозі.

Клас отримує параметр dirPath у своєму конструкторі, який є шляхом до каталогу, який потрібно проаналізувати. У конструкторі клас використовує метод Files.walk для отримання потоку всіх файлів у каталозі та його підкаталогах, відфільтровує будь-які каталоги у потоці та зіставляє кожен файл з його рядковим представленням. Отриманий потік збирається у список шляхів до файлів за допомогою методу collect.

Метод compute відповідає за виконання власне обчислень. Спочатку він створює список завдань FileSizeAnalysis, по одному для кожного файлу у каталозі. Потім він розгалужує кожне завдання, що означає, що завдання виконується асинхронно в окремому потоці. Метод розгалуження повертається негайно, а завдання продовжує виконуватися у фоновому режимі.

Після розгалуження всіх завдань, метод compute циклічно переглядає список завдань, приєднується до кожного завдання (тобто чекає на завершення завдання і повертає його результат) і об'єднує результат кожного завдання в єдину HashMap за допомогою методу merge. Метод злиття приймає три аргументи: ключ значення, яке потрібно об'єднати, значення, яке потрібно об'єднати, і функцію, яка об'єднує старе і нове значення, коли в обох картах з'являється однаковий ключ. У цьому випадку функція має вигляд Integer::sum, що означає, що старе і нове значення додаються разом.

Результатом обчислення є об'єднана HashMap.

A screenshot of a computer program

Description automatically generated with medium confidence

Тепер перейдем до демонстрації роботи програми. В проекті присутні декілька файлів формату .txt. Один з них, на якому проводяться тести 1 завдання має розмірність в 1.5 млн слів. Файл називається text2.txt і має 262 210 рядків с словами.

A screenshot of a computer

Description automatically generated

Приклад роботи програми:

Мною приведені два приклади роботи програми (1 – це найперший запуск, 2 – це подальші запуски). В статистиці приведені дані про довжину середню слова в тексті, про різницю в довжині слова, та стандартне відхилення.

Як бачимо, результат при використанні паралелізації за допомогою пулу ForkJoin у 549/362 = 1.5 рази швидший. Також зазначимо, що дана оптимізація має сенс лише при масивах даних великого розміру адже розбиття відбувається при умові того, що отримана кількість символів у підсписку = 500 000.

A screenshot of a computer

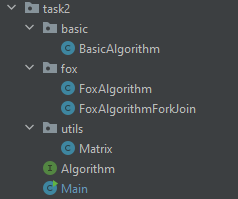
Description automatically generatedA screenshot of a computer

Description automatically generated

## Завдання 2

Для виконання завдання 2 був обраний алгоритм множення матриць методом Фокса.

Почнемо зі структури проекту Task2.



Програма має основний файл Main, клас Matrix, інтерфейс Algorithms, клас FoxAlgorithm та клас FoxAlgorithmForkJoint.

Для початку пройдемося по базовим файлам, методам та алгоритмам, які не мають основного відношення до завдання.

Файл IAlgorithms:

A screenshot of a computer program

Description automatically generated with medium confidence

Тут визначається інтерфейс з іменем "Algorithm" з сигнатурою методу "multiply()", який повертає об'єкт Matrix. Його мета - визначити набір методів, які клас повинен реалізувати.

Файл Matrix:

A screen shot of a computer program

Description automatically generated with medium confidence

Тут ми визначаємо клас з іменем Matrix, який представляє математичну матрицю та має наступні атрибути:

* "matrix": двовимірний масив з двійкових чисел, який представляє саму матрицю.
* "sizeAxis0": ціле число, яке представляє кількість рядків у матриці.
* "sizeAxis1": ціле число, що представляє кількість стовпців у матриці.

Клас має наступні методи:

* "Matrix(int sizeAxis0, int sizeAxis1)": конструктор матриці
* "print()": метод, який виводить матрицю на консоль.
* "getSizeAxis0()": метод, який повертає кількість рядків у матриці.
* "getSizeAxis1()": метод, який повертає кількість стовпців у матриці.
* "generateRandomMatrix()": метод, який заповнює матрицю випадковими значеннями

A screen shot of a computer program

Description automatically generated with medium confidenceФайл BasicAlgorithm:

Тут ми визначаємо клас з іменем BasicAlgorithm, який реалізує інтерфейс "Алгоритм", визначений у попередньому фрагменті коду.

Атрибути класу:

Дві матриці А і В, які потрібно перемножити.

Клас також реалізує метод multiply() інтерфейсу "Algorithm". Цей метод виконує операцію множення матриць між матрицями "A" та "B" за стандартним алгоритмом. Отримана матриця зберігається у новому об'єкті Matrix "C". Множення матриць виконується з використанням трьох вкладених циклів. Зовнішні цикли ітераційно перебирають рядки та стовпці результуючої матриці "C", тоді як внутрішній цикл виконує точковий добуток між відповідним рядком матриці "A" та стовпцем матриці "B" для обчислення кожного елемента результуючої матриці "C".

A screenshot of a computer program

Description automatically generated with medium confidenceФайл Foxalgorithm:

Для початку тут визначений клас з іменем "FoxAlgorithm", який розширює клас Java "RecursiveTask" і реалізує паралельний алгоритм множення матриць, відомий як "алгоритм Фокса".

Клас має наступні атрибути:

* Три матриці А, В, С – дві з яких будуть перемножуватися і записувати результат в матрицю С.
* "nBlocks": ціле число, яке представляє кількість блоків, на які потрібно розбити матриці.
* "step": ціле число, яке представляє розмір кожного блоку.
* "matrixOfSizesI": двовимірний масив цілих чисел, що представляє матрицю розмірів блоків у напрямку I.
* "matrixOfSizesJ": двовимірний масив цілих чисел, що представляє матрицю розмірів блоків у напрямку J.

A screenshot of a computer program

Description automatically generated with medium confidence

Це конструктор класу FoxAlgorithm, який отримує три аргументи: A та B типу Matrix та nBlocks типу int.

Спочатку конструктор ініціалізує матриці A, B і C (які будуть результатом перемноження матриць) і перевіряє, чи мають A і B однакову розмірність. Якщо вони не мають однакової розмірності, він генерує виключення і виходить з програми.

Далі конструктор знаходить найближчий цілочисельний дільник nBlocks, який ділить A.getSizeAxis0() (кількість рядків у матриці A) рівномірно. Це пов'язано з тим, що алгоритм Фокса вимагає, щоб матриця була розбита на блоки однакового розміру.

Потім конструктор ініціалізує дві матриці matrixOfSizesI та matrixOfSizesJ з nBlocks рядків та стовпців. Ці матриці будуть зберігати розміри підматриць, які будуть використовуватися при множенні матриць.

Нарешті, конструктор викликає метод preparing(), який заповнює матриці matrixOfSizesI та matrixOfSizesJ відповідними розмірами підматриць.

A picture containing text, screenshot, font

Description automatically generated

Цей метод використовується для знаходження найближчого цілочисельного дільника заданого числа, який є меншим або рівним заданій границі. Метод отримує два цілих числа, s і p, які представляють межу і число, яке потрібно розділити, відповідно.

Спочатку метод ініціалізує змінну i значенням s і входить у цикл while, який перевіряє, чи ділиться p на i. Якщо p ділиться на i, цикл виходить, і метод повертає значення i. Якщо p не ділиться на i, метод зменшує i на одиницю, якщо i менше s, і збільшує i на одиницю, якщо i більше або дорівнює s.

Якщо i стає меншим або рівним 1 або більшим за квадратний корінь з p, метод встановлює i в мінімальне значення між s і p/s мінус один, де / означає ціле ділення. Нарешті, метод повертає i, якщо i більше або дорівнює s, p/i, якщо i не дорівнює нулю, і p, якщо i дорівнює нулю.

Багато незрозумілого але цей метод використовується для знаходження оптимального значення кількості блоків, на які потрібно розбити матрицю для алгоритму Фокса, гарантуючи, що розміри блоків відносно рівні і мінімізуючи накладні витрати на зв'язок між процесорами.

A picture containing text, screenshot, software

Description automatically generated

Метод preparing() ініціалізує дві матриці matrixOfSizesI та matrixOfSizesJ розміром nBlocks по nBlocks. Ці матриці використовуються для відстеження початкового індексу кожного блоку для матриць A та B.

Наприклад, якщо nBlocks дорівнює 2, а розмір матриці A 4 на 4, то кожен блок A матиме розмір 2 на 2. matrixOfSizesI та matrixOfSizesJ будуть ініціалізовані наступним чином:

*matrixOfSizesI = [[0, 2], [0, 2]]*

*matrixOfSizesJ = [[0, 2], [0, 2]]*

Це означає, що лівий верхній блок A починається з індексу (0,0) і має розмір 2 на 2, правий верхній блок починається з індексу (0,2) і має розмір 2 на 2, і так далі.

Метод copyBlock() використовується для копіювання блоку матриці у новий об'єкт Matrix під час роботи алгоритму Фокса. Параметри i та j задають початковий індекс блоку, що копіюється, а розмір - розміром блоку.

A screen shot of a computer program

Description automatically generated with medium confidence

Це головний метод класу FoxAlgorithm. Він перевизначає метод compute() класу RecursiveTask, що дозволяє виконувати його в пулі ForkJoint за допомогою методу invoke() класу ForkJoinTask.

Метод compute() спочатку створює порожній список об'єктів RecursiveTask, який буде містити задачі, що будуть обчислювати матричне множення для кожного блоку результуючої матриці C. Потім він запускає три вкладені цикли для перебору всіх блоків матриць A і B. Для кожної пари блоків він визначає позицію результуючого блоку в матриці C і створює нову задачу FoxAlgorithmForkJoin з двома блоками матриць A і B і позицією результуючого блоку. Потім завдання додається до списку завдань і розгалужується для асинхронного виконання в пулі розгалужень.

Після того, як всі завдання були розгалужені, метод ітераційно переглядає список завдань за допомогою циклу for і викликає метод join() для кожного завдання, щоб дочекатися його завершення і отримати його результат. Результатом є об'єкт HashMap, який містить результуючу матрицю блоків та їх позицію в результуючій матриці C. Потім метод копіює отриманий блок у відповідну позицію матриці C за допомогою двох вкладених циклів і повертає результуючу матрицю C.

A screenshot of a computer program

Description automatically generated with medium confidenceТепер і сам файл FoxAlgorithmForkJoin:

Цей код реалізує множення двох матриць за допомогою алгоритму Фокса та фреймворку Fork-Join для паралельного виконання. Фреймворк Fork-Join - це функція Java, яка дозволяє паралельне виконання завдань шляхом створення пулу потоків для їх виконання. Клас FoxAlgorithmForkJoin розширює RecursiveTask<HashMap<String, Object>>, що дозволяє виконувати його асинхронно з використанням фреймворку Fork-Join. Конструктор отримує в якості параметрів дві матриці A і B, а також індекси блоку в C, який буде обчислюватися даною задачею, stepI і stepJ.

Метод multiplyBlock() виконує множення двох матриць A і B і повертає отриманий блок як об'єкт типу Matrix.

Метод compute() є перевизначеним методом класу RecursiveTask і викликається при виконанні задачі. Він викликає метод multiplyBlock для обчислення блоку, зберігає його в HashMap<String, Object> і повертає разом з індексами stepI і stepJ.

Фух, наче всі допоміжні класи, параметри та методи були розглянуті – тепер перейдемо до основного класу Main, який запускає весь код task2.

A screenshot of a computer program

Description automatically generated with medium confidenceФайл Main:

Основний метод класу Main створює дві матриці A та B розміром sizeAxis0 на sizeAxis1. Потім він ініціалізує A і B випадковими значеннями і створює екземпляр класу BasicAlgorithm для виконання множення матриць за послідовним алгоритмом. Час, витрачений на виконання множення, зберігається в currTimeBasic.

Далі створюється екземпляр класу FoxAlgorithm з параметрами A, B і 6, що представляє собою кількість блоків, на які розбиваються матриці. Потім клас ForkJoinPool використовується для виклику екземпляра FoxAlgorithm, який паралельно виконує множення матриць, використовуючи фреймворк Fork-Join. Час, витрачений на виконання множення, зберігається в currTimeFox.

Нарешті, виводяться матриці результатів, якщо параметр printMatrices рівний true, а також обчислюється і виводиться прискорення паралельного алгоритму порівняно з послідовним алгоритмом.

Вкінці ми порівнюємо час виконання алгоритмів, та вираховуємо прискорення ділячи час виконання подібного алгоритму з 2лр на його реалізацію зараз через ForkJointPool. Я опитав декількох одногрупників і визначив середній час виконання алгоритму Фокса з 2 лр (так як 2 лр я поки що не здав, через правки, які треба переробити). Отже середній час виконання Фокса в 2 лр був такий:

Матриця 500\*500 = 85 мс;

Матриця 1000\*1000 = 321 мс;

Матриця 2000\*2000 = 2973 мс;

Так як ми вже знаємо, що паралельні реалізації алгоритмів працюють краще на даних великої розмірності. Тому зразу перевіримо на матриці 2000\*2000 і порівняємо прискорення у відношенні базовий алгоритм/ФоксФоркДжоінт та Фокс2лр/ФоксФоркДжоінт.

A screenshot of a computer screen

Description automatically generated with medium confidence

Як бачимо послідовний алгоритм займає дуже багато часу в той час як алгоритм Фокса через пул потоків набагато швидший, а саме в 23.3 рази. А ось в порівнянні з алгоритмом Фокса з 2 лр, пришвидшення майже не помітне.

## Завдання 3

В цьому завданні нам також треба використати фреймворк Fork/Join для обробки декількох файлів і підрахунку загальної кількості слів у них. Ось короткий опис того, що відбувається:

Програма створює список об'єктів File, які представляють файли, що підлягають обробці. У цьому випадку програма обробляє два файли: "text/text.txt" і "text/LordOfTheRings.txt".

Програма створює об'єкт ForkJoinPool, який є пулом робочих потоків, що можуть виконувати завдання паралельно. У програмі створюється об'єкт MultiFileWordTask, який є підкласом RecursiveTask, що представляє задачу підрахунку слів у декількох файлах. Далі викликається метод invoke() об'єкта ForkJoinPool і передає йому об'єкт MultiFileWordTask. Це призводить до того, що завдання виконується паралельно робочими потоками в пулі. Так як програма невеличка – то і весь код в одному файлі.

A picture containing text, screenshot

Description automatically generatedA picture containing text, screenshot, font

Description automatically generated

Клас FileWordTask має єдину змінну екземпляра, яка є об'єктом File, що представляє файл, який потрібно обробити.

Метод compute() перевизначається для визначення задачі, яку представляє цей клас. Метод compute() повертає множину рядків, яка містить окремі слова у файлі. Він спочатку створює порожній HashSet, який буде зберігати окремі слова у файлі. Потім зчитує всі рядки з файлу за допомогою методу Files.readAllLines() і зберігає їх у Списку рядків. Після цього програма виконує ітерацію над кожним рядком файлу і розбиває його на масив слів за допомогою регулярного виразу \s+. Цей регулярний вираз розбиває рядок на лексеми, розділені одним або кількома пробілами. Потім додає всі токени до HashSet за допомогою методу addAll().

Якщо під час читання файлу виникає виключення вводу-виводу, воно перехоплюється і виводиться у стандартний потік помилок за допомогою методу e.printStackTrace(). Вкіні, метод compute() повертає HashSet, що містить окремі слова у файлі.

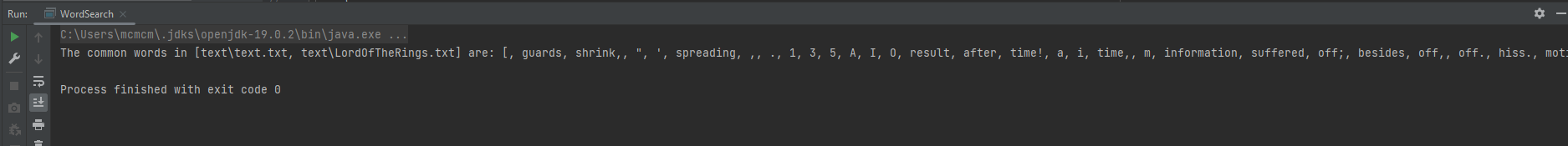
A screenshot of a computer program

Description automatically generated with medium confidence

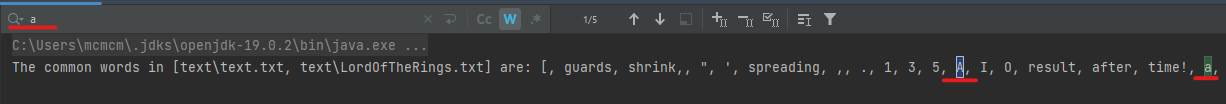
Клас MultiFileWordTask має єдину змінну екземпляра, яка є об'єктом List of File, що представляє файли, які потрібно обробити.

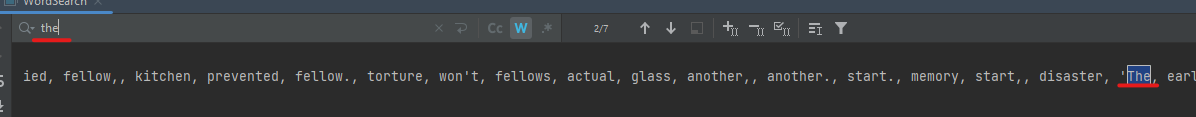
Метод compute() повертає Void, оскільки йому не потрібно повертати жодного результату; натомість він виводить у стандартний потік виводу спільні слова, знайдені у файлах, спочатку перевіряє, чи список файлів достатньо малий, щоб його можна було обробити безпосередньо у цій задачі. У цьому випадку він створює список об'єктів типу Set, кожен з яких містить окремі слова з одного файлу. Потім програма обчислює перетин усіх цих наборів за допомогою методу retainAll() першого набору, який модифікує його так, щоб він містив лише ті елементи, які також присутні в усіх інших наборах. Результатом є набір спільних слів у всіх файлах. Він виводить у стандартний потік виводу повідомлення про те, які файли було опрацьовано і які слова є спільними.

Якщо список файлів занадто великий для обробки безпосередньо цією задачею, метод compute() розбиває список на дві половини і створює два нових об'єкти MultiFileWordTask для їх рекурсивної обробки. Потім він викликає метод invokeAll() для паралельного виконання цих завдань за допомогою фреймворку Fork/Join. Ідея полягає в рекурсивному розбитті списку файлів на менші підсписки, поки кожен підсписок не стане достатньо малим, щоб його можна було обробити безпосередньо FileWordTask, а потім об'єднати результати для отримання спільних слів у всіх файлах.

Як вже зазначалося – шукати спільні слова програма буде у файлах "text/text.txt"(43 тис рядків слів) і "text/LordOfTheRings.txt"(4200 рядків слів).

В результаті бачимо перелік спільних слів. Але для удостовірення правильності роботи зробимо пошук слів таких як «а, the, I». Так як такі слова є в кожному тексті.





A screen shot of a computer

Description automatically generated with medium confidence

Як бачимо все працює правильно і коректно.

## Завдання 4

Реалізація завдання 4 має теоретично ідентичну реалізацію із завдання 3. Однак, для збереження інформації про факт знаходження ключового слова у тому чи іншому файлі, потрібно було відмовитися від розділення документів на підзадачі за лімітом довжини.

A screen shot of a computer program

Description automatically generated with medium confidence

A screen shot of a computer code

Description automatically generated with low confidence

Ця програма шукає файли у заданому каталозі та його підкаталогах, які містять будь-яке з вказаних слів у заздалегідь визначеному списку. Вона використовує фреймворк Fork/Join для розпаралелювання процесу пошуку, що може підвищити продуктивність на багатоядерних процесорах.

Програма починає з визначення списку слів для пошуку і створення екземпляра ForkJoinPool для управління паралельним виконанням завдань. Потім вона створює екземпляр класу FileSearchAction, який розширює клас RecursiveAction, що надається фреймворком Fork/Join.

Клас FileSearchAction отримує на вхід об'єкт File і перевизначає метод compute() для виконання пошуку файлу. Якщо файл є каталогом, він рекурсивно створює нові об'єкти FileSearchAction для кожного з підкаталогів і відправляє їх до ForkJoinPool за допомогою методу invokeAll(). Якщо файл не є директорією, він читає вміст файлу і перевіряє, чи містить він якесь із заданих слів. Якщо збіг знайдено, ім'я файлу виводиться на консоль.

Метод main() програми ініціалізує об'єкт File з шляхом до каталогу для пошуку, і передає новий екземпляр класу FileSearchAction до ForkJoinPool за допомогою методу invoke().

Для тестування буде обрано цілий список файлів, з попередніх завдань та нові також. Присутні папки в папках та файли лежачі в них.

A screenshot of a computer program

Description automatically generated with medium confidence

Зробимо пошук у файлах слів «Bob» та «GG». В результаті має вивести 3 файла – task2\_2.txt та text.txt та LordOfTheRings.txt бо в них вони присутні:

A screenshot of a computer screen

Description automatically generated with medium confidence

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

Запускаємо програму:

**A screenshot of a computer

Description automatically generated with medium confidence**

Як бачимо все працює справно та коректно.

**ВИСНОВОК**

В рамках роботи, ми на практиці познайомились з пулом потоків ForkJoin, його застосуванням, перевагами та недоліками.

Маючи декілька задач, ми змогли як покращити час виконання раніше реалізованих алгоритмів, за рахунок використання цього пулу потоків, так і створити оптимальні версії інших алгоритмів, задачі яких сприятливі для цього.