МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Київський національний університет будівництва і архітектури

Технології розподілених систем та паралельних обчислень

Методичні вказівки
до виконання практичних та лабораторних робіт
для студентів спеціальності
F3 (122) Комп’ютерні науки

Київ 2025

УДК 004.042

Укладачі: О.Л. Соловей, канд. техн. наук

Відповідальна за випуск Т.А. Гончаренко, доктор.тех.наук, професор

Затверджено на засіданні кафедри інформаційних технологій, протокол № 5 від 10 грудня 2025 року.

В авторській редакції.

Технології розподілених систем та паралельних обчислень: Методичні вказівки до виконання практичних та лабораторних робіт / Уклад. О.Л. Соловей. – Київ: КНУБА, 2025. – 32 с

Містять теоретичні відомості і рекомендації щодо виконання лабораторних робіт з дисципліни та вимоги до оформлення звіту. Спрямовані на організацію самостійної роботи студентів.

Призначені для студентів спеціальності F3 (122) Комп’ютерні науки для практичного використання при виконанні лабораторних робіт.

Зміст

Вступ

Лабораторні роботи є логічним продовженням лекційного курсу з дисципліни “Технології розподілених систем та паралельних обчислень” і є перехідною ланкою від теоретичного курсу до набуття практичних навичок з розробки програм мовою програмування Java з організацією одночасних обчислень в декількох потоках.

Кожна лабораторна робота містить наступні види робіт:

- аналіз умови задачі і розробка підходу до її розв’язку.
- покрокову розробку алгоритму розв’язку і його опис.
- обґрунтування алгоритму.
- написання програми, що реалізує цей алгоритм.
- демонстрація правильної роботи програми на обраному наборі тестів.
- складання і захист звіту.

Лабораторна робота №1.

На мові програмування Java розробити власну реалізацію черг ArrayBlockingQueue, DelayQueue, PriorityQueue, SynchronousQueue з бібліотеки java.util.concurrent.

Мета роботи: Здобути навики роботи чергами пакету java.util.concurrent.

Теоретичні відомості.

У багатопотокових програмах часто потрібно безпечно обмінюватися даними між потоками. Один з класичних підходів - використання черг (queues), де: producer (виробник) додає елементи в чергу, consumer (споживач) забирає елементи з черги.

Черга має бути потокобезпечною (thread-safe), тобто коректно працювати при одночасному доступі з кількох потоків. У конкурентних чергах зазвичай використовуються блокуючі методи:

1. put(E e) - додає елемент в чергу. Якщо немає місця (черга повна), метод блокує потік, доки місце не з’явиться.
2. take() - повертає й видаляє елемент з черги. Якщо черга порожня, метод блокує потік, доки не з’явиться новий елемент.

Таке блокуюче поводження дозволяє: не використовувати активне очікування (busy waiting), природно реалізувати патерн producer–consumer.

У власних реалізаціях блокуючих черг зазвичай використовують: synchronized + wait() / notify() / notifyAll(), або ReentrantLock + Condition (більш гнучкий і близький до java.util.concurrent).

Типи черг:

ArrayBlockingQueue - це обмежена за розміром черга з кільцевим буфером (ring buffer) на масиві. Особливості: Фіксована ємність, задається в конструкторі: capacity. Мета такої черги - ефективний обмін елементами між потоками з фіксованою кількістю місця.

PriorityQueue - черга з пріоритетами. У стандартній бібліотеці PriorityQueue - не потокобезпечна структура даних, але в задачі йдеться про конкурентну

чергу з пріоритетами (аналог PriorityBlockingQueue).Особливості: Елементи зберігаються за пріоритетом, а не в порядку вставки. Метод take() повинен повертати елемент з найвищим пріоритетом (наприклад, мінімальний або максимальний за значенням). Для потокобезпеки - спільний lock, блокування при порожній черзі. Такі черги використовуються, коли важливо обробляти елементи за важливістю, а не просто в порядку надходження.

DelayQueue - це черга, елементи якої стають доступними тільки після закінчення певного часу. Елементи реалізують інтерфейс Delayed, де є метод getDelay(TimeUnit unit). Метод take(): повертає елемент тільки тоді, коли його delay вичерпано (час очікування минув); якщо перший елемент ще «не дозрів» - take() блокується до завершення delay. Внутрішньо зазвичай використовують чергу з пріоритетами, де пріоритет - це момент часу готовності. Така структура підходить для: задач планування, таймерів, відкладеної обробки подій.

SynchronousQueue - це особливий тип черги, що не має буфера взагалі. Принцип роботи: Кожний put() чекає, поки якийсь потік виконає take() для цього ж елемента. Кожний take() чекає, поки з’явиться put(). Немає внутрішнього зберігання елементів - лише “рукопотискання” (hand-off) між producer і consumer. Ця модель корисна, коли: виробник не повинен «наперед» накопичувати дані, кожен елемент має бути негайно оброблений споживачем. У власній реалізації можна використати: wait() / notify() з одним спільним об’єктом і змінною стану, або Lock + Condition, де put() і take() взаємно «будять» одне одного.

Організація Producer–Consumer - класичний спосіб організації багатопотокових програм: Producer: генерує дані (наприклад, числа, повідомлення), викликає queue.put(value). Consumer: забирає дані value = queue.take(), обробляє їх (вивід у консоль, обчислення тощо)

Завдання

1. На мові програмування Java розробити власну реалізацію черг ArrayBlockingQueue, DelayQueue, PriorityQueue, SynchronousQueue з бібліотеки java.util.concurrent.
2. Програма має реалізовувати методи: put(), take().
3. У головній програмі написати сценарії для тестування, які включатимуть потоки producer і consumer для кожного типу черги.
4. Відповісти на контрольні запитання, підготувати звіт.

Контрольні запитання

1. Чим відрізняється блокуюча черга від неблокуючої? Наведіть приклади сценаріїв, де блокування у put() / take() є необхідним.
2. Поясніть, як працює кільцевий буфер (ring buffer) в реалізації ArrayBlockingQueue. Для чого потрібні індекси head і tail?
3. Чому в DelayQueue зручно використовувати пріоритетну чергу всередині? Як саме момент часу «готовності» елемента впливає на take()?
4. У чому різниця між FIFO-чергою та чергою з пріоритетами? Чи можна в PriorityQueue гарантувати порядок вставки для елементів з однаковим пріоритетом?
5. Чому SynchronousQueue не має внутрішнього буфера? Які переваги й недоліки такого підходу в контексті producer–consumer?
6. Як зміниться поведінка вашої PriorityQueue, якщо кілька потоків одночасно додають елементи? Які поля й операції потрібно захистити блокуванням?
7. Опишіть, як би ви тестували коректність роботи своєї SynchronousQueue. Які сценарії потрібно перевірити, щоб упевнитись у відсутності deadlock’ів і втрати даних?

Лабораторна робота №2.

На мові програмування Java розробити власну реалізацію політик обробки відхилених задач (RejectedExecutionHandler) з бібліотеки java.util.concurrent.

Мета роботи: Здобути навики роботи з методами класу ThreadPoolExecutor з пакету java.util.concurrent.

Теоретичні відомості.

У багатопотокових додатках створення нового Thread для кожної задачі - дорого й неефективно, щоб зменшити ці витрати, використовують пули потоків (thread pools): ThreadPoolExecutor - головна реалізація пулу потоків у Java (java.util.concurrent). Завдання класу ThreadPoolExecutor це створити обмежену кількість потоків, які повторно використовуються для виконання великої кількості задач (Runnable / Callable).Таким чином, пул потоків дозволяє:

1. контролювати кількість паралельних потоків;
2. керувати чергою задач;
3. задавати політику відмови, коли задач занадто багато;
4. логічно розділяти місця створення задач та їх реальне виконання.

Клас ThreadPoolExecutor має ключові параметри (у конструкторі):

corePoolSize - кількість «основних» потоків (постійний мінімум).

maximumPoolSize - максимальна кількість потоків.

workQueue - черга задач (наприклад, BlockingQueue<Runnable>).

ThreadFactory - фабрика для створення нових потоків. ThreadFactory визначає як саме створюються нові потоки для пулу.

```
public interface ThreadFactory {
   Thread newThread(Runnable r);
}
```

RejectedExecutionHandler – це інтерфейс, який слугує обробником для завдань (Runnable), які не можуть бути виконані об'єктом ThreadPoolExecutor.

Інтерфейс містить єдиний абстрактний метод: void rejectedExecution(Runnable r, ThreadPoolExecutor executor)

«r»: це запитане завдання (Runnable), яке не вдалося виконати.

«executor»: це пул потоків (ThreadPoolExecutor), який намагався виконати завдання.

Метод rejectedExecution викликається об'єктом ThreadPoolExecutor у тих випадках, коли метод execute не може прийняти завдання. Це може статися через дві основні причини:

1. Коли відсутні вільні потоки або вільні слоти в черзі завдань, оскільки їхні визначені межі були перевищені.
2. Зупинка Executor: У разі зупинки (shutdown) Executor.

Коли обробник викликається, він повинен визначити, як поводитися з відхиленим завданням. Якщо обробник не має іншого виходу або альтернативи для обробки завдання, він може згенерувати неперевірений виняток RejectedExecutionException. Цей виняток буде поширений до потоку, який викликав метод execute.

У пакеті java.util.concurrent надано чотири стандартні політики (класи), які реалізують інтерфейс RejectedExecutionHandler, як вбудовані політики ThreadPoolExecutor:

1. ThreadPoolExecutor.AbortPolicy
2. ThreadPoolExecutor.CallerRunsPolicy
3. ThreadPoolExecutor.DiscardOldestPolicy
4. ThreadPoolExecutor.DiscardPolicy

Завдання

1. Реалізуйте власний ThreadPoolExecutor, який включатиме ваші реалізації методів:

a. ThreadFactory – створює потоки з іменем CustomThread-ID.

b. CallerRunsPolicy – виконує задачу у потоці, який викликав execute().

c. DiscardPolicy – просто відкидає задачу без повідомлення.

d. DiscardOldestPolicy – видаляє найстарішу задачу з черги, а потім додає нову.

e. AbortPolicy – обробник відмови, що генерує виняток RejectedExecutionException.

2. В головному потоці розробіть сценарії для тестування поведінок 1.a-1.e.
3. Відповісти на контрольні запитання, підготувати звіт.

Контрольні запитання

1. Що таке ThreadPoolExecutor і які основні переваги використання пулу потоків порівняно зі створенням нового Thread для кожної задачі?
2. Яку роль відіграє ThreadFactory у ThreadPoolExecutor? Навіщо задавати власні імена потоків (наприклад, CustomThread-ID)?
3. У яких випадках викликається RejectedExecutionHandler? Наведіть приклад ситуації, коли це відбудеться у вашому тестовому сценарії.
4. Поясніть, як працює політика CallerRunsPolicy. Яким чином вона реалізує механізм «back pressure» для потоку, який генерує задачі?
5. Які ризики має використання DiscardPolicy? У яких системах, на вашу думку, це може бути прийнятним, а в яких - категорично небажаним?
6. Чим відрізняється DiscardOldestPolicy від DiscardPolicy? Наведіть приклад прикладної задачі, де DiscardOldestPolicy буде кращим вибором.
7. Що відбувається при використанні AbortPolicy, коли пул не може прийняти нову задачу? Як ви запропонували б обробляти виняток RejectedExecutionException у головному коді?

8. Як за допомогою вашого власного ThreadPoolExecutor можна експериментально переконатися, що CallerRunsPolicy дійсно виконує задачу в потоці-викликачі?

Лабораторна робота №3.

Розробити програму алгоритму «Сортування злиттям» методами ForkJoinPool.

Мета роботи: Набути навичок роботи з фреймворком ForkJoinPool при реалізації рекурсивних алгоритмів.

Теоретичні відомості.

Merge Sort - це алгоритм сортування типу «розділяй і володарюй» (divide and conquer). Його основні етапи: Рекурсивне розбиття масиву на дві половини, доки не залишиться масиви довжиною 1. Сортування кожної половини (також рекурсивне). Злиття (merge) двох відсортованих підмасивів в один великий. Часова складність алгоритму O(n log n) у найгіршому, середньому та кращому випадках. Просторова складність O(n) через допоміжний масив у merge-фазі.

ForkJoin Framework - це механізм Java для розпаралелювання задач, особливо рекурсивних. Для цього використовуються два базових класи:

RecursiveAction - для задач, що не повертають результат (void).

RecursiveTask<V> - для задач, що повертають значення.

Оскільки сортування виконує модифікацію масиву на місці і не мусить повертати значення, використовують RecursiveAction.

Клас MergeSortTask наслідується від класу extends RecursiveAction, таким чином успадковує метод compute() який містить логіку: якщо підмасив достатньо малий → виконати сортування; інакше: знайти середину, створити дві нові задачі: leftSortTask та rightSortTask, викликати invokeAll(leftSortTask, rightSortTask) (або fork() і join()), виконати merge() після того, як підзадачі завершились.

Метод compute() має визначати умову коли підмасив дуже малий, наприклад:

```
if (right - left <= threshold)
   Arrays.sort(array, left, right + 1);
```

Метод merge(int[] array, int left, int mid, int right) відповідає за злиття двох відсортованих підмасивів:
Перший підмасив: array[left .. mid]
Другий підмасив: array[mid+1 .. right]
Злиття відбувається шляхом:створення тимчасового масиву, покрокового порівняння елементів обох підмасивів, копіювання найменшого елемента у тимчасовий масив, перенесення результату назад у array.

У паралельному сортуванні важливо уникати надмірного створення задач. Часто вводять константу threshold, при якій перехід на звичайне Arrays.sort() швидший за паралельне розбиття.

Клас ForkJoinTask – абстрактний клас для опису об'єкта «задача», яка виконуватиметься у ForkJoinPool. Задача починає виконуватися у ForkJoinPool, щойно її надіслано методом submit(). Якщо задача надсилається до спільного пулу, тоді використовують метод ForkJoinPool.commonPool(). Також ForkJoinTask може бути виконана у ForkJoinPool викликом двох методів – fork() та join(). Метод fork() організовує асинхронне виконання підзадач. Метод join() не повертає керування, доки не буде отримано результат виконання задачі. Таким чином, метод join() є аналогом Future.get().

Статуси задачі: isDone() – задача завершена (включно з випадком, коли її було скасовано - cancelled). isCompletedNormally() – задача виконана без виникнення виняткових ситуацій. isCancelled() – задача була скасована (cancelled). isCompletedAbnormally() – задача завершилась некоректно: була скасована або під час виконання сталася помилка (виняток).

Завдання

Розробити клас MergeSortTask, який наслідуватиме клас RecursiveAction і реалізовуватиме методи: @Override protected void compute() private void

merge(int[] array, int left, int mid, int right) для сортування масиву елементів за алгоритмом «Сортування злиттям».

Підготувати звіт, який включатиме: лістинг коду, демонстрацію роботи програми, відповіді на контрольні запитання.

Контрольні запитання

1. Поясніть алгоритм крадіжки роботи (work-stealing algorithm), який використовує Fork/Join фреймворк.
2. У чому різниця між абстрактними класами RecursiveTask та RecursiveAction?
3. Які типи конструкторів містить клас ForkJoinPool?
4. Які функції виконує ForkJoinPool.ForkJoinWorkerThreadFactory?
5. Які статуси може мати задача у ForkJoinPool?
6. У чому різниця між методами public <T> T invoke(ForkJoinTask<T> task) і public void execute(ForkJoinTask<?> task)
7. Що таке threshold у паралельному сортуванні? Чому його використання покращує продуктивність?
8. Що робить метод invokeAll()? У чому різниця між підходами fork/join та invokeAll?
9. Чому паралельний Merge Sort не завжди швидший за послідовний? Наведіть приклади.

Лабораторна робота №4.

Розробити програму для алгоритму трьохканального злиття з використанням методу get() інтерфейсу Future.

Мета роботи: Набути навичок використання методів інтерфейсу Future під час реалізації алгоритмів сортування.

Теоретичні відомості.

В основі методу зовнішнього сортування збалансованим багатоканальним злиттям є розподіл серій вхідного файлу по m допоміжних файлів C1, C2, ..., Cm і т. д., доки в B1 або C1 не утвориться одна серія. Відсортувати файл, використовуючи трьохканальне злиття (рис. 1). Як видно з рис. 1, по мірі, збільшення довжини серій допоміжні файли, з великими номерами (починаючи з номера n) перестають використовуватися, оскільки їм „не дістається" жодної серії.

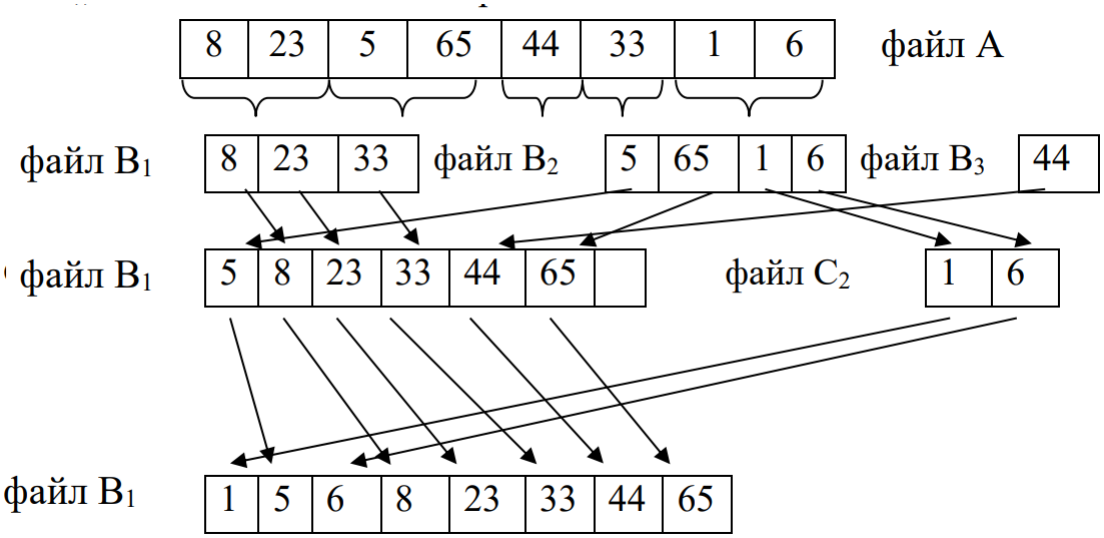

Рис. 1. Трьохканальне злиття

Інтерфейс Future<V> надає методи для управління асинхронним завданням, включаючи перевірку його статусу, очікування завершення та отримання результату.

• V: Це параметр типу, який визначає тип результату, що повертається методом get() цього Future. Якщо необхідно використовувати Future лише заради можливості скасування (cancellability), але без повернення

корисного результату, можна оголосити типи у формі Future<?> та повертати null як результат базового завдання.

Інтерфейс Future визначає п'ять основних методів:

1. Отримання результату (get)

◦ V get(): Очікує (блокує потік, якщо необхідно), поки обчислення не завершиться, а потім отримує його результат.

◦ V get(long timeout, TimeUnit unit): Очікує щонайбільше заданий час на завершення обчислення, а потім отримує його результат, якщо він доступний. Якщо час очікування вичерпано до завершення, генерується виняток TimeoutException.

2. Перевірка статусу

◦ boolean isDone(): Повертає true, якщо завдання завершилося. Завершення може бути наслідком нормального виконання, винятку або скасування - у всіх цих випадках метод повертає true.

◦ boolean isCancelled(): Повертає true, якщо завдання було скасовано до того, як воно завершилося нормально.

3. Скасування завдання

◦ boolean cancel(boolean mayInterruptIfRunning): Робить спробу скасувати виконання цього завдання. Ця спроба буде невдалою, якщо завдання вже завершилося, вже було скасовано або не може бути скасовано з іншої причини. Якщо скасування успішне і завдання ще не розпочалося, воно ніколи не повинно бути виконане. Якщо завдання вже розпочалося, параметр mayInterruptIfRunning визначає, чи слід перервати потік, який виконує завдання, намагаючись його зупинити. Після повернення цього методу, подальші виклики isDone() завжди повертатимуть true. Якщо метод cancel повернув true, подальші виклики isCancelled() також завжди повертатимуть true.

Методи get() можуть генерувати три основні винятки:

• CancellationException: Якщо обчислення було скасовано.

• ExecutionException: Якщо обчислення згенерувало виняток.

• InterruptedException: Якщо поточний потік був перерваний під час очікування.

Класи, які реалізують інтерфейс Future, включають CompletableFuture, ForkJoinTask, FutureTask та інші. FutureTask — це реалізація Future, яка також реалізує інтерфейс Runnable, тому вона може бути виконана за допомогою Executor. Наприклад, використання ExecutorService.submit() повертає Future, але можна також створити FutureTask і передати його до executor.execute().

Завдання

Розробити клас ThreeChannelMergeSort з методом private static int[] mergeThreeChannels(int[] array1, int[] array2, int[] array3), який приймає три відсортовані послідовності та повертає одну відсортовану послідовність чисел. Виконання методу mergeThreeChannels має відбуватися у пулі потоків, створеному за допомогою ExecutorService. Результат отримати шляхом виклику методу Future.get().

Підготувати звіт, який включатиме: лістинг коду, демонстрацію роботи програми, відповіді на контрольні запитання.

Контрольні запитання

1. Які ключові методи має інтерфейс Future?
2. Як слід викликати метод isDone, щоб перевіряти статус завдання через визначені проміжки часу?
3. Поясніть різницю між викликом методу Future.cancel(true) та Future.cancel(false)?
4. З якими значеннями параметрів слід викликати метод scheduleWithFixedDelay, щоб пауза між завданнями для виконання становила 5 секунд, а початкова затримка була 1 секунда?
5. У чому різниця між Runnable та Callable? Чому в даній роботі зручніше використовувати Callable?

6. Які винятки можуть виникнути при виклику future.get()? У яких випадках вони виникають?
7. Що станеться, якщо не викликати shutdown() чи shutdownNow() у ExecutorService? Чому це може бути проблемою для програми?

Лабораторна робота №5.

Розробити власну реалізацію алгоритму LockFreeQueue.

Мета роботи: Набути навичок використання методів java.util.concurrent.atomic.

Теоретичні відомості.

Пакет java.util.concurrent.atomic надає набір класів, які підтримують потокобезпечне програмування без використання блокувань (lock-free) для одиночних змінних. Ці класи, по суті, розширюють концепцію значень, полів та елементів масивів, оголошених як volatile.

Ключові механізми та операції

1. Умовне атомарне оновлення (compareAndSet): Основним механізмом, який використовується в класах цього пакету, є атомарна операція умовного оновлення: boolean compareAndSet(expectedValue, updateValue). Цей метод атомарно встановлює змінну на значення updateValue, лише якщо вона наразі містить expectedValue, повертаючи true у разі успіху. Специфікації цих методів дозволяють реалізаціям використовувати ефективні атомарні інструкції на рівні машини, доступні на сучасних процесорах. Хоча вони є високоефективними, методи не мають суворої гарантії неблокування, оскільки на деяких платформах підтримка може вимагати певної форми внутрішнього блокування.

2. Слабке умовне оновлення (weakCompareAndSet): Пакет також містить операцію умовного атомарного оновлення - weakCompareAndSet. Ключова відмінність полягає в тому, що weakCompareAndSet може повертати false помилково (spuriously), навіть якщо очікуване значення збігається з поточним. Повернення false означає лише те, що операцію можна повторити. Крім того, weakCompareAndSet не створює жодних гарантій порядку happens-before,. Коли потік бачить оновлення атомарної змінної, спричинене weakCompareAndSet, він не обов'язково бачить оновлення будь-яких інших змінних, що відбулися до weakCompareAndSet.

3. Доступи та оновлення атомарних змінних загалом відповідають правилам для volatile змінних.

◦ Метод get має ефекти пам'яті читання volatile змінної.

◦ Метод set має ефекти пам'яті запису volatile змінної.

◦ Операції compareAndSet та всі інші операції читання та оновлення (наприклад, getAndIncrement) мають ефекти пам'яті як читання, так і запису volatile змінних.

◦ Метод lazySet дозволяє змінювати порядок виконання з наступними діями пам'яті, які самі по собі не накладають обмежень на зміну порядку звичайних не-volatile записів.

Пакет java.util.concurrent.atomic включає категорії класів:

AtomicBoolean, AtomicInteger, AtomicLong, AtomicReference<V> - надають доступ та оновлення однієї змінної відповідного типу. Наприклад, AtomicLong та AtomicInteger надають атомарні методи інкременту, які можуть використовуватися для генерації порядкових номерів.

AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray<E> - розширюють підтримку атомарних операцій на масиви. Ці класи примітні тим, що надають семантику доступу volatile для своїх елементів масиву, що не підтримується для звичайних масивів.

AtomicIntegerFieldUpdater<T>, AtomicLongFieldUpdater<T>, AtomicReferenceFieldUpdater<T,V> - це утиліти які дозволяють атомарно оновлювати вибрані volatile поля призначених класів. Вони використовуються переважно в атомарних структурах даних, де кілька volatile полів одного вузла незалежно підлягають атомарним оновленням.

Спеціалізовані посилання AtomicMarkableReference<V> - підтримує посилання на об'єкт разом із бітом позначки (mark bit), що може використовуватися, наприклад, для позначення логічно видаленого об'єкта всередині структури даних.

AtomicStampedReference<V> - підтримує посилання на об'єкт разом із цілим числом ("штампом"), яке може використовуватися для представлення номерів версій, що відповідають серії оновлень.

Акумулятори - LongAdder, DoubleAdder, LongAccumulator, DoubleAccumulator, які використовуються для підтримки суми або поточного значення, яке оновлюється за допомогою наданої функції.

Атомарні класи розроблені в першу чергу як будівельні блоки для реалізації неблокуючих структур даних та пов'язаних інфраструктурних класів.

Завдання

Розробити клас LockFreeQueue з методами: public void enqueue(T value) public T dequeue() які забезпечують роботу черги за алгоритмом «без блокування».

Підготувати звіт, який включатиме: лістинг коду, демонстрацію роботи програми, відповіді на контрольні запитання.

Контрольні запитання

1. Поясніть та наведіть приклад роботи з методом public final int getAndAddInt(Object o, long offset, int delta)?
2. Яку атомарну операцію умовного оновлення надають класи пакету java.util.concurrent.atomic?
3. Чому операції в атомарних класах (наприклад, compareAndSet) не мають суворої гарантії неблокування, незважаючи на їхню ефективність, що базується на машинних інструкціях?
4. Які ефекти пам'яті (memory effects) має метод lazySet порівняно зі звичайним методом set, і для яких сценаріїв він може бути корисним?
5. Які класи в пакеті java.util.concurrent.atomic використовуються для розширення підтримки атомарних операцій на масиви?
6. У чому полягає ключова відмінність операції weakCompareAndSet від compareAndSet щодо можливості помилкового повернення false?

7. Чому weakCompareAndSet не створює жодних гарантій порядку happens-before щодо інших змінних, і коли це може бути прийнятно?,
8. Для чого використовуються класи оновлювачів на основі рефлексії (наприклад, AtomicIntegerFieldUpdater), і які поля вони дозволяють атомарно оновлювати?
9. Яку додаткову інформацію, окрім посилання на об'єкт, зберігають класи AtomicMarkableReference та AtomicStampedReference, і яка функція цієї інформації?
10. Чому атомарні класи (наприклад, AtomicInteger) не є універсальною заміною для стандартних класів Java (наприклад, java.lang.Integer)?

Лабораторна робота №6.

Реалізувати «задачу філософів, що обдають» методами класу Phaser.

Мета роботи: Вивчити методи класу Phaser для організації фазового контролю над потоками.

Теоретичні відомості.

Клас java.util.concurrent.Phaser є багаторазовим бар'єром синхронізації, який за функціональністю схожий на CyclicBarrier та CountDownLatch, але пропонує більш гнучке використання.

Цей клас був доданий у Java Platform SE 7. На відміну від більшості інших бар'єрів, кількість учасників (parties), зареєстрованих для синхронізації на Phaser, може змінюватися з часом. Завдання можуть реєструватися у будь-який час за допомогою методів register() (для одного учасника) або bulkRegister(int) (для кількох учасників). Ці методи додають нових учасників до Phaser.

Учасник може бути дерегістрований після прибуття за допомогою методу arriveAndDeregister(). Дереєстрація зменшує кількість учасників, необхідних для просування фази в майбутньому.

Якщо Phaser має батьківський елемент (parent), і його кількість зареєстрованих учасників змінюється з нуля на ненульове значення (або навпаки), відбувається автоматична реєстрація або дереєстрація цього дочірнього Phaser у батьківському.

Кожне покоління Phaser має пов'язаний номер фази:

1. Початкова фаза: номер фази починається з нуля.

2. Просування фази: Номер фази збільшується, коли всі зареєстровані учасники прибувають до Phaser. Після досягнення Integer.MAX_VALUE номер фази зациклюється на нуль.

Методи arrive() та arriveAndDeregister() фіксують прибуття. Ці методи не блокують потік, але повертають номер фази, до якої застосовується прибуття (arrival phase number).

Метод arriveAndAwaitAdvance() фіксує прибуття та очікує, поки прибудуть інші учасники, що призведе до просування фази. Ефект цього методу аналогічний CyclicBarrier.await.

Метод awaitAdvance(int phase) вимагає аргументу, що вказує номер фази прибуття, і повертається, коли Phaser просувається до іншої фази,. На відміну від схожих конструкцій з CyclicBarrier, awaitAdvance продовжує чекати, навіть якщо потік, що очікує, перервано. Існують також версії з можливістю переривання (awaitAdvanceInterruptibly) та таймаутом.

Коли прибуває останній учасник для даної фази, виконується необов'язкова дія, і фаза просувається.

Метод onAdvance: Ця дія виконується учасником, який спричинив просування фази. Вона налаштовується шляхом перевизначення захищеного методу onAdvance(int phase, int registeredParties), який також контролює завершення роботи Phaser. Якщо onAdvance повертає true, Phaser переходить у кінцевий стан завершення. Стандартна реалізація onAdvance повертає true, якщо дереєстрація призвела до того, що кількість зареєстрованих учасників стає нульовою. Стан завершення можна перевірити за допомогою методу isTerminated(). При завершенні всі методи синхронізації негайно повертаються без очікування, про що свідчить негативне значення, що повертається. Метод forceTermination() доступний для примусового переведення Phaser у стан завершення, що негайно звільняє потоки, які очікують.

Для зменшення суперечності синхронізації при великій кількості учасників (Phaser обмежує максимальну кількість учасників до 65535) їх можна організовувати в деревоподібні структури (tiered).

Поточний стан Phaser може контролюватися будь-яким викликаючим потоком, незалежно від того, чи він зареєстрований. Методи моніторингу включають:

• getPhase(): Повертає поточний номер фази.

• getRegisteredParties(): Повертає загальну кількість зареєстрованих учасників.

• getArrivedParties(): Повертає кількість зареєстрованих учасників, які вже прибули до поточної фази.

• getUnarrivedParties(): Повертає кількість учасників, які ще не прибули до поточної фази (при їхньому прибутті фаза просувається).

• toString(): Повертає знімок цих запитів стану у зручному для моніторингу вигляді.

Завдання

Є кілька філософів, які сидять за круглим столом і по черзі думають та їдять. Для того, щоб поїсти, кожен філософ має два ресурси - виделки, що знаходяться між сусідніми філософами. Проблема полягає в тому, що філософи повинні одночасно використовувати дві виделки, і вони не повинні здійснювати конфлікти або блокувати один одного. Необхідно розробити програму яка розв’яже проблему. Phaser дозволяє синхронізувати потоки, а також дає можливість розділяти операції на етапи. Кожен філософ буде виконувати кілька етапів: Думати - філософ не потребує жодних ресурсів. Брати виделки - філософ має забрати дві виделки з двох сусідніх філософів. Їсти - після того, як філософ взяв виделки, він може поїсти. Повернути виделки - після того, як філософ поїв, він повертає виделки. Ви можете бути «автором» свого алгоритму реалізації завдання, може реалізувати один з алгоритмів з сайту Проблема філософів, що обідають - Задача філософів, що обідають - Вікіпедія

Контрольні запитання

1. У чому полягає ключова відмінність Phaser від інших бар'єрів, таких як CyclicBarrier або CountDownLatch, стосовно кількості учасників (parties)? Які методи дозволяють змінювати цю кількість динамічно?
2. Як працює механізм фаз (phase number) у Phaser, і що відбувається з номером фази після досягнення Integer.MAX_VALUE?

3. Опишіть функціональні відмінності між методами arrive(), arriveAndDeregister() та arriveAndAwaitAdvance()?

4. Яку роль відіграє захищений метод onAdvance(int phase, int registeredParties), і яким чином він контролює завершення роботи Phaser?

5. Які гарантії надає метод awaitAdvance(int phase) щодо переривання потоку (interruption), і чим це відрізняється від поведінки, наприклад, awaitAdvanceInterruptibly?

6. Що таке багаторівневість (Tiering) у контексті Phaser, і чому вона може знадобитися для Phaser з великою кількістю учасників?

7. Яким чином Phaser переходить у стан завершення (termination state)? Що відбувається з методами синхронізації (наприклад, arriveAndAwaitAdvance), коли Phaser перебуває в цьому стані, і яке значення вони повертають?

8. Як реалізовано автоматичне управління реєстрацією та дереєстрацією дочірніх Phaser у батьківському елементі при багаторівневій структурі?

9. Які методи моніторингу стану, окрім getPhase(), може використовувати будь-який викликаючий потік (незалежно від реєстрації) для перевірки поточної кількості зареєстрованих та прибулих учасників?

10. Коли може бути корисним метод forceTermination(), і який вплив він має на зареєстрованих учасників та очікуючі потоки?

Лабораторна робота №7.

Робота з Apache Kafka.

Мета роботи: Ознайомитись із принципами роботи системи обміну повідомленнями Apache Kafka. Навчитись створювати продюсерів (Producers) та консьюмерів (Consumers), що взаємодіють з Kafka topics з різною кількістю розділів (partitions).

Теоретичні відомості.

Apache Kafka використовується для широкого спектра завдань обробки даних у реальному часі:

Kafka є ефективною заміною традиційним брокерам повідомлень (наприклад, ActiveMQ або RabbitMQ). У порівнянні з більшістю систем обміну повідомленнями, Kafka забезпечує кращу пропускну здатність, вбудоване розділення (partitioning), реплікацію та відмовостійкість, що робить його хорошим рішенням для великомасштабної обробки повідомлень. Для цих випадків часто потрібна низька наскрізна затримка та сильні гарантії довговічності.

Початковим сценарієм використання Kafka було відновлення конвеєра відстеження активності користувачів як набору каналів публікації/підписки в реальному часі. Активність сайту (перегляди сторінок, пошукові запити) публікується в центральні топіки. Ці канали є високооб'ємними, оскільки генерується багато повідомлень активності на кожен перегляд сторінки користувачем.

Kafka використовується як заміна рішенням для агрегації логів (наприклад, Scribe або Flume). Він абстрагує деталі фізичних файлів логів і надає абстракцію даних подій або логів як потоку повідомлень. Це забезпечує нижчу затримку обробки, простішу підтримку розподіленого споживання даних, а також пропонує міцніші гарантії довговічності завдяки реплікації.

Kafka часто застосовується для оперативних даних моніторингу, агрегуючи статистику з розподілених додатків для створення централізованих потоків операційних даних.

Багато користувачів використовують Kafka для створення багатостадійних конвеєрів обробки, де вхідні дані споживаються з топіків Kafka, а потім агрегуються, збагачуються або трансформуються в нові топіки.

Kafka підтримує додатки, побудовані у стилі Event Sourcing, де зміни стану записуються як часопослідовна послідовність записів. Завдяки підтримці дуже великих обсягів збережених даних логів, Kafka може виступати як зовнішній журнал комітів (commit-log) для розподіленої системи, допомагаючи реплікувати дані між вузлами та відновлювати дані для вузлів, що вийшли з ладу.

Kafka Streams - це клієнтська бібліотека, доступна в Apache Kafka починаючи з версії 0.10.0.0, призначена для обробки та аналізу даних, що зберігаються в Kafka.

Ключові концепції Kafka Streams включають:

• Належне розрізнення часу події (event time) та часу обробки (processing time).

• Підтримку віконності (windowing support).

• Семантику обробки "рівно один раз" (exactly-once processing semantics).

• Ефективне управління станом додатку.

Kafka Streams має низький поріг входу (low barrier to entry): можна швидко створити прототип на одній машині, а для масштабування на великі робочі навантаження достатньо запустити додаткові екземпляри програми на кількох машинах. Kafka Streams прозоро керує балансуванням навантаження між екземплярами однієї програми, використовуючи модель паралелізму Kafka.

Завдання

1. Створити клас ProducerSinglePartition, який надсилає повідомлення до Kafka topic, що має один розділ. Назва топіку повинна включати ваше прізвище, наприклад: messages_kovalenko_1p.
2. Створити клас ProducerMultiPartition, який надсилає повідомлення до Kafka topic з трьома розділами. Назва топіку повинна включати ваше прізвище, наприклад: messages_kovalenko_3p.
3. Створити клас ConsumerSinglePartition, який читає повідомлення з Kafka topic, що має один розділ (використовується топік з п.1).
4. Створити клас ConsumerMultiPartition, який читає повідомлення з Kafka topic з трьома розділами (використовується топік з п.2).
5. Забезпечити, щоб Consumer міг працювати з кількома потоками або у складі Consumer Group.

Вимоги до реалізації:

1. Повідомлення повинні містити інформацію у форматі: ID: | Повідомлення: (наприклад: ID: 5 | Повідомлення: Тестове повідомлення).

2. Для Producer реалізуйте логіку надсилання щонайменше 10 повідомлень з інтервалом 1 секунда.

3. Для Consumer реалізуйте вивід отриманих повідомлень у консоль.

4. Усі Kafka topics повинні бути створені вручну або програмно перед запуском Producer/Consumer.

5. Код має бути реалізований на мові програмування Java

Контрольні запитання

1. Як впливає кількість розділів (partitions) у topic на споживання
2. повідомлень?
3. Як визначається, у який розділ (partition) потрапляє повідомлення від
4. Producer?
5. Що станеться, якщо два Consumer-и читають з одного розділу одного topic
6. без використання Consumer Group?

7. Яким чином можна забезпечити доставку повідомлень у Kafka у правильному порядку?
8. Які основні переваги має Apache Kafka порівняно з традиційними брокерами повідомлень (наприклад, ActiveMQ або RabbitMQ) у контексті великомасштабних застосувань для обробки повідомлень?
9. Опишіть, що таке Event Sourcing (Проектування на основі подій), і як Kafka підтримує додатки, побудовані в цьому стилі?
10. Що таке Kafka Streams, і з якої версії Apache Kafka ця клієнтська бібліотека доступна?,
11. Назвіть принаймні три ключові концепції потокової обробки, які підтримує бібліотека Kafka Streams?

Список літератури

1. Лісовенко І.Д., Яковлєва І. Д. Навчальний посібник «Паралельні та розподілені обчислення». Чернівці: ЧНУ, 2022. 120 с.
2. Минайленко Р.М. Паралельні та розподілені обчислення : навч. посіб. — Кропивницький: Видавець Лисенко В. Ф., 2021. 153 с.
3. Коцовський В. М. Теорія паралельних обчислень: навчальний посібник. /В. М. Коцовський - Ужгород: ПП «АУТДОР-Шарк», 2021. - 188 с.
4. Малашонок Г. І., Сідько А. А. Паралельні обчислення на розподіленій пам’яті: OpenMPI, Java, Math Partner : підручник. / Г. І. Малашонок, А. А. Сідько. – Київ : НаУКМА, 2020. – 266 с.
5. Корочкін О.В. Паралельні та розподілені обчислення. Вибрані розділи: Навч. посібник. / О.В. Корочкін, О.В. Русанова.– Київ : КПІ ім. Ігоря Сікорського, 2020. – 123 с.
6. Solovei, O., Honcharenko, T. and Solovei, B., 2025, May. A Discrete Bayesian Network Model For Diagnosing Latency Growth In Apache Kafka Cluster Within Information Systems For Building Construction Projects. In 2025 IEEE 5th International Conference on Smart Information Systems and Technologies (SIST) (pp. 1-7). IEEE.
7. Solovei, O. and Honcharenko, T., Leveraging Sensitivity Analysis for Configurable Kafka Clusters: A Multi-objective Model to Minimize Latency.

Інформаційні ресурси в Інтернет

1. Java специфікація класу Циклічний бар’єр. Режим доступу CyclicBarrier (Java Platform SE 8) (oracle.com)
2. Java специфікація класу Семафор. Режим доступу Semaphore (Java Platform SE 8) (oracle.com)
3. Java специфікація станів потоку. Режим доступу Thread.State (Java Platform SE 8) (oracle.com)

4. Java специфікація інтерфейсу Lock. Режим доступу Lock (Java SE 11 & JDK 11) (oracle.com)
5. Java специфікація інтерфейсу Executor. Режим доступу Executor (Java Platform SE 8) (oracle.com)
6. Java специфікація інтерфейсу Executor. Режим доступу ExecutorService (Java Platform SE 8) (oracle.com)