МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

МИКОЛАЇВСЬКИЙ НАЦІОНАЛЬНИЙ АГРАРНИЙ УНІВЕРСИТЕТ

Факультет менеджменту

Кафедра економічної кібернетики, комп'ютерних наук та інформаційних технологій

ТЕХНОЛОГІЇ РОЗПОДІЛЕНИХ СИСТЕМ ТА ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

конспект лекцій

для здобувачів першого (бакалаврського) рівня вищої освіти ОПП «Комп'ютерні науки» за спеціальністю F3(122) «Комп'ютерні науки» денної форми здобуття вищої освіти

МИКОЛАЇВ
2026

УДК 004.75:004.272
Т38

Друкується за рішенням науково-методичної комісії факультету менеджменту Миколаївського національного аграрного університету від 19 березня 2026 року, протокол № 7.

Укладач:

О.О. Жебко — асистент кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій, Миколаївський національний аграрний університет

Рецензенти:

В.В.Базаренко — заступник начальника Миколаївської обласної військової адміністрації з питань цифрового розвитку, цифрових трансформацій і цифровізації (CDTO)

Д.Л.Кошкін — к.т.н., доцент, доцент кафедри електроенергетики, електротехніки та електромеханіки Миколаївського національного аграрного університету

Т38 **Технології** розподілених систем та паралельних обчислень : конспект лекцій для здобувачів першого (бакалаврського) рівня вищої освіти ОПП «Комп'ютерні науки» спец. F3 (122) «Комп'ютерні науки» денної форми здобуття вищої освіти / уклад. О. О. Жебко. Миколаїв : МНАУ, 2026. 141 с.

Конспект лекцій призначений для вивчення основ паралельних обчислень і технологій розподілених систем, ознайомлення з принципами організації та взаємодії обчислювальних процесів. Містить навчальні матеріали з основних тем курсу «Технології розподілених систем та паралельних обчислень», що передбачені освітньо-професійною програмою підготовки здобувачів першого (бакалаврського) рівня вищої освіти спеціальності F3 (122) «Комп'ютерні науки», галузі знань 12 «Інформаційні технології».

УДК 004.75:004.272

ЗМІСТ

Передмова

Курс дисципліни «Технології розподілених систем та паралельних обчислень» призначений для формування у здобувачів вищої освіти спеціальності 122 «Комп'ютерні науки» цілісного уявлення про принципи побудови, функціонування та організації сучасних систем обробки даних, що працюють у багатопроцесорному, багатопотоковому або розподіленому середовищі.

Предметом вивчення дисципліни є закономірності роботи паралельних і розподілених обчислювальних середовищ, моделі взаємодії процесів, механізми синхронізації.

Об'єктом вивчення є паралельні та розподілені обчислювальні процеси, потоки та процеси операційних систем.

Метою викладання дисципліни є підготовка висококваліфікованих фахівців, здатних ефективно проектувати, аналізувати та розробляти паралельні та розподілені програмні системи, розуміти принципи їхнього функціонування, забезпечувати коректну взаємодію процесів і оптимізувати обчислення для підвищення продуктивності та масштабованості програмних рішень.

Основними завданнями, що мають бути вирішені в процесі викладання дисципліни, є:

− ознайомлення студентів із базовими поняттями паралельних та розподілених обчислень і сучасними моделями паралельної обробки даних;

− формування знань щодо механізмів синхронізації, комунікації та управління потоками і процесами;

− навчання практичним методам розробки багатопотокових застосунків та розподілених систем із використанням інструментів Java та інших сучасних платформ;

− формування навичок аналізу та оптимізації паралельних алгоритмів,

моніторингу продуктивності та забезпечення коректності конкурентних процесів;

− розвиток умінь застосовувати набуті знання у моделюванні, тестуванні й побудові масштабованих обчислювальних рішень.

Конспект лекцій складено з урахуванням вимог чинної освітньо-професійної програми підготовки фахівців зі спеціальності «Комп’ютерні науки», а також на основі сучасних підручників, навчальних посібників, стандартів та нормативних документів, що регламентують галузь паралельних і розподілених обчислень.

ЗМІСТОВИЙ МОДУЛЬ 1
ВСТУП ДО БАГАТОПОТОЧНОГО (ПАРАЛЕЛЬНОГО) ПРОГРАМУВАННЯ

ТЕМА 1.1. ПОНЯТТЯ ПРО ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ

План

1. Послідовні обчислення
2. Паралельні обчислення
3. Засоби для здійснення паралельних обчислень
4. Паралельні комп’ютери
5. Актуальність та перспективи використання паралельних обчислень
6. Сфери застосування паралельних обчислень та рівні розпаралелювання

1. Послідовні обчислення

Послідовні обчислення являють собою класичну модель виконання програм, у межах якої всі інструкції виконуються строго одна за одною, відповідно до заздалегідь визначеної послідовності. У такій моделі існує лише один потік керування, а отже, кожна наступна операція починається лише після того, як завершена попередня. Такий спосіб виконання програм історично був основним у комп’ютерних системах з одним процесором і залишається актуальним для більшості базових алгоритмів та програм.

У послідовному режимі виконання кожен крок залежить від результату попереднього, тому порядок інструкцій є критично важливим. Це визначає ще одну характерну рису – детермінованість: однаковий вхід завжди породжує однаковий результат. Саме тому послідовні програми відносно прості для тестування, аналізу й відлагодження: програміст чітко бачить логіку переходів між операціями та легко відтворює помилки.

Послідовні обчислення добре працюють у задачах, де існує пряма залежність між етапами обробки. Наприклад, при поетапному зчитуванні даних із файлу, при покроковому моделюванні процесів чи в алгоритмах, де кожне наступне значення обчислюється на основі попереднього. Такі задачі важко або неможливо розпаралелити без зміни сутності самого алгоритму.

Прикладом послідовного виконання у Java може бути проста операція обчислення суми елементів масиву: цикл перебирає елементи один за одним, і нове значення суми з’являється лише після виконання конкретної операції додавання. Ніякі два кроки циклу не можуть виконуватися одночасно в межах одного потоку.

Попри простоту, послідовні обчислення мають низку обмежень, які стали особливо критичними в епоху багатоядерних процесорів. Головне з них – обмеження продуктивності. Програма, що працює послідовно, використовує лише одне ядро, а отже, не може повною мірою скористатися потенціалом сучасних апаратних засобів. Час виконання таких програм зростає пропорційно обсягу даних, а в багатьох випадках – навіть швидше. Якщо в алгоритмі є складні обчислення, вони просто блокують подальше виконання до повного завершення.

У контексті паралельних і розподілених обчислень важливо підкреслити, що будь-яка послідовність інструкцій є потенційним "вузьким місцем". Саме це формулює Закон Амдала, згідно з яким максимальне прискорення програми визначається тією частиною, яку неможливо розпаралелити. Навіть за умов наявності сотень процесорів частка послідовного коду обмежує загальний приріст продуктивності.

Попри це, послідовні обчислення мають фундаментальне значення у викладанні технологій паралельного програмування. Перш ніж говорити про прискорення, взаємодію потоків чи синхронізацію, необхідно розуміти класичну модель, у якій кожна операція залежить від попередньої. Саме аналіз послідовного алгоритму дає змогу визначити, які його частини можна виконувати паралельно, а які є критично залежними.

Отже, послідовні обчислення – це базова, але все ще надзвичайно важлива модель організації виконання програм. Вони забезпечують зрозумілу логіку, точне відтворення результатів і зручність розробки, але водночас накладають обмеження на швидкість виконання в умовах сучасних апаратних можливостей. У подальших лекціях саме від цієї моделі будемо відштовхуватися, аналізуючи можливості розпаралелювання та переходу до багатопотокового виконання програм у Java.

2. Паралельні обчислення

Паралелні обчислення – це модель виконання програм, у якій кілька операцій здійснюються одночасно, тобто паралельно, використовуючи можливості багатоядерних процесорів, багатопроцесорних систем або навіть розподілених обчислювальних середовищ. На відміну від послідовних обчислень, де існує лише один потік керування, паралельні обчислення передбачають наявність двох і більше потоків або навіть окремих процесів, які виконують частини однієї задачі одночасно.

Потреба у паралельних обчисленнях виникла, коли подальше підвищення продуктивності за рахунок збільшення частоти процесорів стало технічно обмеженим. Тому комп’ютерні архітектури почали переходити до концепції багатоядерності: замість одного дуже швидкого ядра використовується кілька ядер, які можуть виконувати різні частини програми незалежно одне від одного. Водночас, щоб скористатися цими можливостями, програмне забезпечення має бути розроблене так, щоб виконувати роботу паралельно.

У паралельних обчисленнях задача розбивається на кілька підзадач, які можуть виконуватися одночасно за умови, що між ними немає критичної взаємозалежності. Якщо частини алгоритму не залежать одна від одної або взаємодіють мінімально, вони можуть бути розміщені в окремих потоках або процесах і працювати паралельно. Таким чином, загальний час виконання скорочується, адже кілька елементів обробляються одночасно. Ефективність

паралельного виконання залежить від того, наскільки добре задача піддається розбиттю на незалежні частини.

Прикладом простої паралельної обробки у Java може бути одночасне обчислення частин суми великого масиву. Замінюючи один довгий цикл на кілька потоків, де кожен обробляє свій фрагмент масиву, можна зменшити загальний час виконання. Однак паралельність додає складності: необхідно правильно синхронізувати доступ до спільних змінних, уникати гонок даних, контролювати порядок завершення потоків та обробку результатів.

Паралельні обчислення не є лише технічним збільшенням швидкості – вони змінюють спосіб мислення програміста. Паралельні програми можуть бути недетермінованими: кожен запуск може давати однаковий результат, але порядок виконання частин програми змінюється щоразу, залежно від розкладу операцій процесором. Це ускладнює тестування та пошук помилок, особливо тих, що проявляються нерегулярно (так звані race conditions).

Важливо розуміти, що навіть за наявності великої кількості ядер розпаралелювати можна не все. Частина алгоритму неминуче має виконуватися послідовно, і саме ця частина стає обмеженням у прирості швидкодії. Закон Амдала показує, що навіть при ідеальному розпаралелюванні прискорення програми обмежене тією частиною, яку не можна виконати паралельно. Тому під час проєктування паралельних систем важливо визначати, які саме блоки роботи дають максимальний ефект при розподілі між потоками.

Паралельні обчислення широко застосовуються у виконанні наукових моделей, обробці великих даних, системах реального часу, графічних обчисленнях, машинному навчанні та серверних багатокористувацьких середовищах. Сучасні фреймворки та технології, такі як Java Streams, Fork/Join Framework, ExecutorService, надають засоби для зручного створення паралельного коду, абстрагуючи частину складності, пов’язаної із керуванням потоками.

Отже, паралельні обчислення є не просто альтернативою класичним послідовним алгоритмам, а цілком новою моделлю мислення, що дозволяє ефективно використовувати ресурси сучасних комп'ютерних систем. Вони дають можливість значно скорочувати час виконання складних задач, але водночас вимагають глибокого розуміння залежностей між частинами алгоритму, механізмів синхронізації та принципів роботи багатопотокових середовищ. Це робить паралельні обчислення важливою складовою сучасного програмування та ключовою темою курсу з технологій розподілених систем та паралельного виконання програм.

3. Засоби для здійснення паралельних обчислень

Засоби для здійснення паралельних обчислень – це сукупність апаратних і програмних механізмів, які забезпечують можливість одночасного виконання кількох операцій або потоків у межах однієї задачі. Вони формують інфраструктуру, на основі якої створюються паралельні програми, що використовують ресурси сучасних багатоядерних і багатопроцесорних систем. Без відповідних засобів паралельність у програмуванні була б надзвичайно складною або навіть неможливою, адже програмісту довелося б вручну контролювати кожну деталь керування потоками, синхронізацію та доступ до ресурсів.

Апаратною основою паралельних обчислень є багатоядерні процесори, багатопроцесорні системи, графічні процесори (GPU), обчислювальні кластери та розподілені інфраструктури. Багатоядерні CPU дають змогу запускати кілька потоків одночасно, тоді як GPU, маючи сотні й тисячі обчислювальних блоків, забезпечують масово-паралельну обробку даних. На рівні серверних систем важливу роль відіграють кластери й хмарні середовища, де паралельні обчислення реалізуються через взаємодію десятків або навіть сотень вузлів, об'єднаних мережею та координованих розподіленими алгоритмами.

На програмному рівні засоби паралельності можна умовно розділити на низькорівневі механізми, які забезпечують базове керування потоками, та

високорівневі фреймворки, що спрощують створення паралельних програм завдяки абстракціям. У низькорівневих механізмах програміст має самостійно створювати, запускати та завершувати потоки, синхронізувати доступ до спільних ресурсів і контролювати взаємодію між потоками. Вони надають максимальний контроль, але вимагають глибокого розуміння тонкощів багатопотокового виконання.

У мові Java традиційно використовується клас Thread, який дозволяє створювати нові потоки через успадкування або реалізацію інтерфейсу Runnable. Пакет java.util.concurrent розширює ці можливості завдяки пулу потоків (ExecutorService), синхронізаторам (Semaphore, CountDownLatch, CyclicBarrier), блокуючим чергам, атомарним змінним та іншим механізмам, що полегшують взаємодію між потоками. Ці засоби дають змогу ефективніше керувати ресурсами, зменшувати накладні витрати створення потоків і підвищувати стабільність паралельних програм.

Важливими програмними інструментами є також високорівневі фреймворки, які приховують більшість деталей роботи потоків і дозволяють програмісту зосередитися на алгоритмі. До таких засобів у Java належить Fork/Join Framework, що реалізує модель розбиття задачі на підзадачі й об’єднання результатів; паралельні Streams API, які забезпечують декларативний підхід до паралельної обробки колекцій; а також бібліотеки для роботи з GPU, розподіленими обчисленнями або великими даними, наприклад, OpenCL, Hadoop, Spark.

Окрему категорію становлять системи і мови паралельного програмування, що спеціально орієнтовані на паралельність. Це OpenMP, MPI, CUDA, які активно застосовуються в наукових обчисленнях, високопродуктивних системах і машинному навчанні. Вони дають змогу гнучко керувати потоками, процесами та мережевою взаємодією між вузлами.

Синхронізаційні механізми – блокування, монітори, атомарні операції, бар’єри – забезпечують коректний доступ потоків до спільних ресурсів і

запобігають помилкам типу гонок даних. Вони є обов'язковою частиною інструментарію, без якого паралельне виконання алгоритмів стало б хаотичним.

Отже, засоби для здійснення паралельних обчислень охоплюють широкий спектр технологій – від апаратної архітектури процесорів до потужних бібліотек і фреймворків. Вони дають можливість реалізувати ефективні паралельні програми та забезпечити значне прискорення обчислень за рахунок одночасного виконання незалежних частин алгоритму. Розуміння цих засобів є ключовим для програміста, який працює зі складними обчислювальними задачами, та основним елементом підготовки в контексті дисципліни з технологій розподілених і паралельних систем.

4. Паралельні комп'ютери

Паралельні комп'ютери – це обчислювальні системи, здатні виконувати багато операцій одночасно завдяки наявності декількох взаємодіючих процесорів або ядер. У таких системах фізична архітектура спеціально орієнтована на паралельну обробку даних, що дає змогу значно пришвидшувати виконання обчислювально складних задач та забезпечувати високу продуктивність при роботі з великими обсягами інформації. Поява паралельних комп'ютерів стала відповіддю на принципове обмеження зростання швидкодії одного ядра: замість нескінченного збільшення частоти процесора індустрія перейшла до об'єднання багатьох обчислювальних елементів у межах однієї системи.

Найпростішими паралельними комп'ютерами вважаються сучасні персональні комп'ютери з багатоядерними процесорами. Кожне ядро здатне виконувати свій потік інструкцій, тому операційна система розподіляє роботу так, щоб кілька задач могли виконуватися паралельно. Проте справжні паралельні комп'ютери охоплюють значно ширший спектр архітектур – від багато-процесорних серверів до високопродуктивних кластерів і суперкомп'ютерів, у яких сотні тисяч процесорів взаємодіють у межах однієї обчислювальної платформи.

З архітектурної точки зору паралельні комп’ютери найчастіше поділяють за класифікацією Флінна, що вирізняє чотири категорії, з яких найбільше практичне значення мають дві. Перша – це системи типу MIMD (multiple instruction – multiple data), де кожен процесор виконує свою інструкцію над власними даними. Такі системи є основою більшості сучасних паралельних серверів і кластерів. Друга – SIMD (single instruction – multiple data), у яких одна інструкція виконується одночасно над багатьма елементами даних; найбільш типовим прикладом таких систем є графічні процесори (GPU), що забезпечують масово-паралельні обчислення.

Окремим різновидом паралельних комп’ютерів є багатопроцесорні системи зі спільною пам’яттю, де всі ядра працюють у межах єдиного адресного простору. Це полегшує програмування, але створює виклики щодо узгодженості кешів і контролю доступу до пам’яті. На противагу їм існують розподілені системи, де кожен обчислювальний вузол має власну пам’ять і взаємодіє з іншими вузлами через мережу. Такі системи становлять основу кластерів і суперкомп’ютерів, адже вони краще масштабується та дають змогу створювати системи з тисячами процесорів.

Суперкомп’ютери – це вершина розвитку паралельних комп’ютерних систем. Вони складаються з великої кількості вузлів, кожен з яких містить багатоядерні процесори та нерідко також графічні або спеціалізовані прискорювачі. Паралельність у таких системах досягається не тільки на рівні окремих процесорів, а й на рівні міжвузлової комунікації, де застосовуються високошвидкісні мережі, спеціальні протоколи та оптимізовані алгоритми обміну даними.

Особливістю паралельних комп’ютерів є необхідність спеціального програмного забезпечення, яке здатне розподілити задачу між різними обчислювальними елементами. У випадку кластерів це можуть бути бібліотеки типу MPI; у випадку багатоядерних систем – технології OpenMP, CUDA чи Java-паралелізм. Саме програмні моделі визначають, яким чином

обчислення будуть синхронізуватися, як передаватимуться дані та яким чином оброблятимуться результати.

Паралельні комп'ютери використовуються в усіх сферах, де потрібні великі обчислювальні ресурси: у моделюванні фізичних процесів, біоінформатиці, обробці великих даних, криптографії, штучному інтелекті та машинному навчанні. Вони стають ключовим інструментом для науки та промисловості, оскільки дозволяють вирішувати задачі, які були б практично недосяжними для традиційної однопроцесорної техніки.

Таким чином, паралельні комп'ютери – це складні, багаторівневі системи, побудовані на принципі одночасного виконання численних обчислень. Їхня архітектура, програмні засоби і методи взаємодії між компонентами формують фундамент сучасних високопродуктивних обчислень і є найважливішим об'єктом вивчення в курсі технологій паралельних і розподілених систем.

5. Актуальність та перспективи використання паралельних обчислень

Актуальність та перспективи використання паралельних обчислень сьогодні визначаються тим, що обсяг даних, складність моделей та вимоги до швидкості обробки інформації стрімко зростають, тоді як традиційні послідовні обчислення вже не здатні забезпечити потрібний рівень продуктивності. Протягом багатьох років підвищення швидкодії комп'ютерів досягалося завдяки збільшенню тактової частоти процесорів, однак фізичні та енергетичні обмеження зробили цей шлях подальшого розвитку практично неможливим. У результаті індустрія перейшла до паралельних архітектур – багатоядерних процесорів, графічних прискорювачів, кластерів і розподілених платформ. Це призвело до того, що саме паралельні обчислення стали основою сучасних високопродуктивних технологій.

Актуальність паралельних обчислень особливо помітна у наукових і технічних сферах. Моделювання кліматичних змін, прогнозування природних явищ, оптимізація виробничих систем, біоінформатика, геноміка

та молекулярна динаміка – усі ці задачі потребують колосальних обчислювальних ресурсів. Паралельні комп’ютери дозволяють обробляти такі дані одночасно в сотнях або тисячах потоків, скорочуючи час, що раніше вимірювався місяцями, до лічених годин або навіть хвилин. У цій сфері паралельні обчислення стають не просто корисним інструментом, а єдиною можливою основою для виконання складних і ресурсомістких задач.

Окремий напрям, у якому паралельність стала критично важливою, – штучний інтелект і машинне навчання. Навчання нейронних мереж, особливо глибинних моделей, неможливе без використання GPU та спеціалізованих прискорювачів, які здатні виконувати мільярди операцій паралельно. Саме завдяки цим технологіям стали можливими сучасні системи розпізнавання зображень, мовлення, рекомендаційні алгоритми та генеративні моделі. Паралельні обчислення фактично стимулюють і визначають розвиток усієї галузі штучного інтелекту.

Паралельні технології мають також ключове значення для обробки великих даних. У світі, де обсяги інформації стрімко зростають, здатність розподіляти обчислення між багатьма вузлами – основа роботи систем аналітики, пошукових механізмів, обліково-транзакційних платформ та хмарних сервісів. Такі фреймворки, як Hadoop чи Spark, побудовані саме за принципами паралельної обробки, що дозволяє системам аналізувати петабайти даних у прийнятні терміни.

У перспективі роль паралельних обчислень лише зростатиме. Сучасні тенденції вказують на подальшу інтеграцію паралельних архітектур у всі рівні обчислювальних систем – від мобільних пристроїв до надпотужних суперкомп’ютерів. З'являються нові типи апаратних прискорювачів, оптимізованих під конкретні класи задач: штучний інтелект, криптографія, моделювання фізичних процесів. Поширення хмарних платформ робить паралельні ресурси доступними широкому колу користувачів, даючи можливість запускати великі обчислення без необхідності володіти дорогою інфраструктурою.

Ще одним перспективним напрямом є розвиток гетерогенних систем, де центральні процесори працюють разом із графічними, тензорними та іншими спеціалізованими ядрами. Такий підхід дає змогу оптимально розподіляти задачі залежно від їхньої природи, досягаючи максимальної продуктивності. Паралельні обчислення також відіграють важливу роль у таких нових галузях, як квантові обчислення, хоча вони базуються на іншій фізичній природі, але також використовують принципи одночасного виконання.

Таким чином, актуальність паралельних обчислень визначається не лише поточними потребами, а й загальною траєкторією розвитку обчислювальних технологій. Перехід до паралельних моделей став основним шляхом подолання обмежень традиційних процесорів, а перспективи їх використання охоплюють усі ключові сфери розвитку науки, техніки та інформаційного суспільства. Паралельні обчислення фактично формують основу майбутніх інновацій і залишаються необхідною компетенцією для сучасних фахівців у галузі ІТ та обчислювальних технологій.

6. Сфери застосування паралельних обчислень та рівні розпаралелювання

Сфери застосування паралельних обчислень та рівні розпаралелювання тісно пов’язані між собою, адже конкретні завдання диктують, який саме рівень паралельності буде ефективним. Сучасні обчислювальні системи настільки складні й різнорівневі, що паралельність може виникати на різних етапах – від внутрішньої роботи процесора до розподілу задач у глобальних обчислювальних мережах. Усі ці аспекти безпосередньо визначають сфери, де паралельні технології забезпечують найбільш відчутні переваги.

Паралельні обчислення застосовуються насамперед там, де обсяги даних або складність задач настільки великі, що послідовне виконання стає неприйнятно повільним. Однією з найважливіших сфер є наукові дослідження: моделювання кліматичних процесів, симуляції фізичних явищ, астрономічні обчислення, розрахунки у ядерній фізиці, хімічній кінетиці,

біомедичній інформатиці. У таких задачах паралельні комп'ютери дають змогу виконувати мільярди операцій за секунду та працювати з моделями, які в сотні разів перевищують можливості звичайних систем. Без паралельності значна частина сучасної науки була б просто недосяжною.

Ще однією ключовою сферою є машинне навчання, штучний інтелект і обробка великих даних. Навчання нейронних мереж, аналіз потокових даних, робота зі складними статистичними моделями потребують величезної обчислювальної потужності. GPU, TPU та інші прискорювачі підвищують швидкість таких обчислень у десятки або сотні разів завдяки масово-паралельному виконанню операцій. Паралельність тут стала не просто інструментом оптимізації, а фундаментальною необхідністю – без неї сучасні AI-технології були б практично неможливими.

Економіка, фінансове моделювання, кібербезпека та криптографія також активно використовують паралельні технології. Це може бути оптимізація портфелів, моделювання ризиків, розрахунок складних фінансових інструментів або аналіз мережевого трафіку в режимі реального часу. У промисловості паралельні обчислення застосовуються в оптимізації виробничих процесів, контролі систем автоматизації, робототехніці та інженерних симуляціях, які потребують швидкої обробки великих масивів сенсорних даних.

У сфері графіки та мультимедіа паралельність є невід'ємною частиною усіх операцій – від рендерингу зображень до обробки відео. Тут обробка кадрів або пікселів відбувається одночасно на тисячах маленьких обчислювальних ядер. У комп'ютерних іграх, VR-системах та анімації паралельні механізми дозволяють досягати реалістичних ефектів у реальному часі.

У контексті розподілених систем паралельні обчислення є основою роботи хмарних платформ, серверних рішень, пошукових систем, соціальних мереж та будь-яких високонавантажених сервісів. Тут паралельність

досягається за рахунок взаємодії великої кількості серверів, що одночасно обробляють незалежні запити користувачів.

Рівні розпаралелювання визначають, де саме відбувається одночасне виконання операцій. Найнижчий рівень – це апаратний паралелізм, або рівень інструкцій, реалізований у самому процесорі. Він включає в себе конвеєризацію, суперскалярність та векторні інструкції, які дозволяють обробляти кілька операцій за один такт.

На рівні програмування важливим є рівень потоків, де паралельність реалізується за допомогою багатопоточних програм у межах одного процесора. Це основа більшості паралельних алгоритмів у мовах програмування, зокрема в Java, де інтегровані механізми керування потоками.

Ще вищим є рівень процесів, де кілька окремих програм (або копій програми) можуть працювати паралельно та обмінюватися даними через міжпроцесну взаємодію. На цьому рівні працюють кластерні системи, бібліотеки типу MPI, хмарні обчислення та інші моделі розподілених систем.

Окремо виділяють рівень даних, коли одна операція застосовується одночасно до великої кількості елементів – цей підхід лежить в основі GPU-архітектур, SIMD-моделей і обробки великих даних.

Нарешті, існує рівень задач, де окремі частини програми або модулі можуть виконуватися одночасно, якщо вони логічно незалежні. Такий підхід характерний для паралельних фреймворків, систем потокової обробки та високорівневих інструментів оптимізації.

Таким чином, сфери застосування паралельних обчислень охоплюють практично всі галузі, що потребують високої продуктивності, а рівні розпаралелювання визначають, на якому саме етапі та з якою деталізацією досягається паралельність. У сукупності вони формують сучасну архітектуру високопродуктивних систем і задають вектор розвитку інформаційних технологій на найближчі десятиліття.

ТЕМА 1.2. ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЮВАЛЬНІ СИСТЕМИ

План

1. Способи обробки даних в обчислювальних системах.
2. Послідовна обробка даних.
3. Конвеєрна обробка даних.
4. Характеристики систем функціональних пристроїв.
5. Класифікація паралельних обчислювальних систем.

1. Способи обробки даних в обчислювальних системах

Способи обробки даних в обчислювальних системах визначають, яким чином інформація передається, зберігається та опрацьовується процесором або групою процесорів. Вони задають логіку роботи всієї обчислювальної системи – від найпростішого виконання інструкцій на одному ядрі до багаторівневих розподілених обчислень у сучасних високопродуктивних середовищах. Вибір способу обробки визначається типом задачі, вимогами до швидкодії, обсягами даних і архітектурою комп’ютера.

Традиційно виділяють три основних способи обробки даних: послідовний, конвеєрний та паралельний. Послідовний спосіб передбачає виконання операцій одна за одною, коли кожна інструкція починає виконуватись лише після завершення попередньої. Це найпростіша і найтрадиційніша модель обробки, яка добре підходить для алгоритмів із чіткими міжкроковими залежностями. Проте її продуктивність обмежена швидкодією одного ядра, що робить цей спосіб малоефективним для великих обсягів даних.

Конвеєрний спосіб обробки полягає у тому, що різні етапи виконання інструкцій можуть оброблятися одночасно, але кожен етап працює над своєю частиною задачі. Такий підхід нагадує конвеєр на виробництві, де одночасно можуть виконуватись різні частини одного й того самого процесу. У комп’ютерних процесорах це реалізується через поділ виконання інструкції

на кілька стадій – вибірку, декодування, обчислення тощо. Завдяки цьому забезпечується збільшення продуктивності навіть без повноцінної паралельності. Конвеєрна обробка є проміжною ланкою між строгою послідовністю та повним розпаралелюванням.

Паралельний спосіб обробки даних полягає в одночасному виконанні кількох операцій над різними частинами даних або навіть над різними наборами інструкцій. Залежно від архітектури це може бути паралельність на рівні окремих інструкцій, потоків, процесорів або вузлів обчислювального кластера. Паралельність дозволяє суттєво скоротити час виконання задач, які можуть бути розбиті на незалежні підзадачі. Саме на цій моделі базуються сучасні багатоядерні процесори, графічні прискорювачі, суперкомп'ютери та розподілені системи.

У сучасних обчислювальних системах також розрізняють пакетний та інтерактивний способи обробки даних, а також потокову (streaming) обробку. Пакетний спосіб використовується при обробці великих обсягів інформації, коли дані накопичуються й обробляються великими блоками – типово для статистичних, бухгалтерських чи аналітичних систем. Інтерактивний спосіб передбачає постійну взаємодію з користувачем, коли система реагує на запити в режимі реального часу.

Потокова обробка даних набуває особливої актуальності у сучасних інформаційних системах. Вона дає змогу опрацьовувати дані безперервно під час їх надходження, що необхідно в системах моніторингу, мережевій аналітиці, відеообробці, інтернеті речей та робототехніці. Потокова модель часто поєднується з паралельною обробкою, що дозволяє виконувати складні операції над даними в реальному часі.

Таким чином, способи обробки даних охоплюють широкий спектр підходів – від традиційних послідовних до високорівневих паралельних і потокових моделей. Розуміння цих способів є основою вивчення сучасних обчислювальних систем, оскільки вибір моделі обробки безпосередньо

впливає на ефективність, масштабованість та можливості програмних рішень.

2. Послідовна обробка даних

Послідовна обробка даних – це найкласичніший і найпростіший спосіб виконання обчислень, у якому всі операції над даними виконуються у строго визначеному порядку, одна за одною. У такій моделі поточна операція починається лише після того, як завершилась попередня, і жодні два кроки не можуть виконуватися одночасно. Це визначає лінійну, детерміновану структуру виконання програм, що робить послідовну обробку фундаментом для всіх інших моделей обчислень.

У послідовній моделі кожен крок обробки залежить від результатів попередніх операцій, тому час виконання напряму пов’язаний з довжиною алгоритму й обсягом даних. Якщо обсяг вхідної інформації зростає, то пропорційно збільшується і час роботи алгоритму, оскільки він змушений послідовно виконувати кожен крок над кожним елементом даних. Це природний, але доволі повільний спосіб обробки, особливо у випадку великих масивів даних або складних обчислень.

Послідовна обробка зручна тим, що забезпечує повну передбачуваність результатів. Оскільки усі операції виконуються у фіксованому порядку, програміст легко контролює логіку, відтворює структуру алгоритму та знаходить помилки. Цей спосіб ідеально підходить для задач, де кожен наступний крок неможливий без результату попереднього, наприклад у криптографічних алгоритмах, рекурсивних обчисленнях або операціях, пов’язаних з історичними залежностями у даних.

У прикладі обчислення суми елементів масиву, який зазвичай наводиться у лекціях, цикл перебирає елементи один за одним, і кожне нове значення підсумку залежить від попереднього. Подібні операції природно реалізуються лише в межах одного потоку, тобто вони не мають внутрішнього потенціалу до розпаралелювання без зміни алгоритмічної структури.

Попри свою простоту, послідовна обробка має важливе місце у сучасних системах. Велика частина програм працює саме у такому режимі, особливо якщо мова йде про прості обчислення, невеликі набори даних або рішення, де паралельність не виправдана ані з технічної, ані з економічної точки зору. Крім того, послідовна модель необхідна для тих частин програми, де необхідна строгість порядку виконання – наприклад, у фінансових транзакціях, локальному збереженні даних, логічних конструкціях або критичних секціях багатопотокових програм.

Однак недоліком послідовної обробки є її обмежена продуктивність. Навіть найпотужніший процесор не може прискорити виконання програми, якщо вона принципово побудована на лінійній послідовності інструкцій. Це обмеження стало ключовою причиною переходу до паралельних обчислень. Закон Амдала особливо яскраво демонструє, що навіть невелика частка послідовного коду може суттєво обмежити загальне прискорення програми, якщо інші її частини розпаралелено.

Таким чином, послідовна обробка даних є базовим способом виконання алгоритмів, необхідним для розуміння принципів програмування та роботи обчислювальних систем. Вона забезпечує передбачуваність, простоту та чіткість, але водночас накладає значні обмеження на швидкодію, що спонукає до пошуку альтернатив у вигляді конвеєрної та паралельної обробки.

3. Конвеєрна обробка даних

Конвеєрна обробка даних – це спосіб організації виконання обчислень, у якому послідовність операцій розбивається на окремі етапи (стадії), що можуть працювати одночасно над різними частинами даних. За своєю ідеєю конвеєрна модель нагадує промисловий конвеєр: поки перша стадія починає працювати над новою порцією даних, друга стадія вже обробляє попередню, третя – ще старішу, і так далі. У результаті підвищується пропускна здатність системи, адже кілька інструкцій перебувають у виконанні одночасно, хоча

фактично кожна окрема інструкція все ще проходить ті самі стадії, що й у послідовному режимі.

Основна ідея конвеєрної обробки полягає у поділі складної операції на набір простіших підоперацій, які можуть виконуватися незалежно й паралельно, якщо при цьому обробляють різні дані. Наприклад, при виконанні машинної інструкції процесор може підготувати наступну інструкцію до виконання, одночасно декодувати попередню та виконувати ще раніше вибрану. Таким чином, хоча кожна інструкція проходить ті самі кроки, пропускна здатність системи різко збільшується, оскільки в конвеєрі одночасно знаходяться інструкції різних етапів.

Конвеєрна обробка стала основою сучасної архітектури процесорів. У більшості процесорів інструкції виконуються у кілька стадій: вибірка, декодування, обчислення, доступ до пам’яті, запис результату. У послідовному режимі кожна інструкція повинна завершити всі ці етапи, перш ніж почнеться наступна; у конвеєрному – нова інструкція може пройти першу стадію вже тоді, коли попередня знаходиться, наприклад, на третій. Це дозволяє одночасно працювати кільком апаратним блокам процесора і не витрачати час простою.

Перевага конвеєрної обробки полягає у значному підвищенні продуктивності без потреби у збільшенні тактової частоти або додаванні нових процесорних ядер. Це робить її ефективною проміжною моделлю між повністю послідовним виконанням і справжньою паралельністю на рівні потоків чи процесів. Конвеєризація не змінює алгоритмічної сутності задачі, але оптимізує її виконання на рівні апаратури.

Однак конвеєрна обробка має й свої обмеження. Найпоширенішою проблемою є так звані “конфлікти конвеєра” або “конвеєрні небезпеки” – ситуації, коли одна інструкція залежить від результату іншої, що ще не завершила свій шлях через конвеєр. Такі залежності можуть призводити до затримок, порушення ритмічності конвеєра або необхідності вставляти “порожні цикли”, що знижує ефективність. Крім того, конвеєрність дуже

чутлива до умовних переходів: коли гілка виконання змінюється, кілька інструкцій у конвеєрі можуть виявитися непотрібними, і треба “скинути” конвеєр, що також призводить до втрат продуктивності.

Попри ці ризики, конвеєрна модель залишається одним із ключових механізмів оптимізації в архітектурі обчислювальних систем. Вона лежить в основі сучасних суперскалярних процесорів, графічних архітектур, засобів SIMD-обробки й багатьох високопродуктивних апаратних рішень. Конвеєрна обробка також активно використовується в мережевому обладнанні, потокових системах, графічних конвеєрах та інших сферах, де необхідно забезпечити безперервну обробку великих обсягів даних.

Таким чином, конвеєрна обробка даних є важливим способом підвищення ефективності обчислювальних систем за рахунок одночасного виконання різних етапів обробки. Вона поєднує лінійну логіку послідовних операцій із перевагами паралельності, забезпечуючи високу пропускну здатність та ефективне використання апаратних ресурсів.

4. Характеристики систем функціональних пристроїв

Характеристики систем функціональних пристроїв описують властивості, за якими оцінюється робота апаратних компонентів обчислювальної системи та їхня здатність ефективно виконувати різні типи операцій. Під функціональними пристроями зазвичай розуміють окремі елементи апаратної архітектури – арифметико-логічні пристрої, блоки керування, регістри, кеш-пам’ять, конвеєрні модулі, пристрої вводу-виводу, а також спеціалізовані прискорювачі. Сукупність характеристик визначає продуктивність, надійність і можливості комп’ютера в цілому, а також його здатність до паралельної та конвеєрної обробки даних.

Однією з ключових характеристик є продуктивність функціонального пристрою, яка визначається кількістю операцій, що можуть бути виконані за одиницю часу. На рівні арифметико-логічних пристроїв це часто вимірюється у FLOPS або GOPS; для пристроїв пам’яті – у пропускній здатності та затримці доступу; для інтерфейсів – у швидкості передачі даних.

Продуктивність тісно пов’язана з тактовою частотою, але не обмежується лише нею, адже апаратні оптимізації – суперскалярність, векторизація чи конвеєризація – можуть суттєво впливати на швидкодію.

Другою важливою характеристикою є пропускна здатність – здатність пристрою обробляти певний обсяг інформації за фіксований проміжок часу. Пропускна здатність визначає, наскільки швидко дані можуть переміщатися між окремими блоками системи: від пам’яті до процесора, між кешами різних рівнів, між ядрами або навіть між вузлами кластерів. У системах із паралельною обробкою пропускна здатність відіграє критичну роль, оскільки недостатньо швидкі канали обміну можуть перетворитися на вузькі місця та суттєво зменшити загальну ефективність.

Третя характеристика – затримка (латентність), тобто час, необхідний функціональному пристрою, щоб почати виконання операції або відповісти на запит. Для процесора затримка може означати час вибірки інструкції з пам’яті, для мережевих пристроїв – час передачі пакета, а для кеш-систем – час заповнення рядка кешу. Низька латентність особливо важлива у системах реального часу, інтерактивних сервісах та високочастотній торгівлі, де навіть мікросекундні затримки можуть впливати на результат.

До важливих характеристик належить і паралельність функціональних пристроїв, тобто їх здатність одночасно виконувати кілька операцій або обробляти кілька потоків даних. Наприклад, сучасні процесори мають декілька незалежних функціональних блоків, які дозволяють виконувати кілька інструкцій за один такт, а GPU мають тисячі елементарних ядер, здатних працювати над різними частинами даних одночасно. Високий рівень паралельності безпосередньо впливає на ефективність обробки великих масивів інформації.

Ще однією характеристикою є архітектурна узгодженість – те, наскільки добре індивідуальні функціональні пристрої інтегруються у роботу всієї системи. Навіть найпотужніший блок може виявитися малоефективним, якщо він не узгоджений із пропускною здатністю пам’яті або конфігурацією

шин. Це особливо актуально в умовах сучасних гетерогенних систем, де CPU, GPU, NPU та інші пристрої взаємодіють між собою.

Важливою є також енергетична ефективність, яка показує, скільки енергії витрачає функціональний пристрій на виконання однієї операції. У мобільних системах та дата-центрах це одна з ключових характеристик, оскільки енергоспоживання прямо впливає на тепловиділення, вартість експлуатації та стабільність системи. Саме тому сучасні архітектури часто переходять до спеціалізованих прискорювачів, які можуть забезпечувати значно кращу енергоефективність у вузьких класах задач.

Загалом, характеристики систем функціональних пристроїв визначають реальний рівень обчислювальних можливостей апаратної платформи. Вони впливають на те, наскільки ефективно система виконує послідовні, конвеєрні та паралельні обчислення, як вона масштабується при збільшенні обсягів даних і наскільки надійно працює в умовах інтенсивного навантаження. Саме від цих характеристик залежить здатність сучасних інформаційних технологій відповідати вимогам до високої продуктивності, низьких затримок та стабільної роботи у складних обчислювальних сценаріях.

5. Класифікація паралельних обчислювальних систем

Класифікація паралельних обчислювальних систем охоплює різні підходи до організації одночасного виконання обчислень і відображає, як у конкретній архітектурі поєднуються процесори, пам’ять, канали зв’язку та програмні моделі. Вона дає змогу розуміти, яким чином створюються високопродуктивні системи та які принципи визначають їхню здатність до масштабування й ефективної паралельної роботи. Найбільш відомі класифікації базуються на принципах Флінна та принципах організації пам’яті й взаємодії між обчислювальними елементами.

Найпоширенішою у теорії залишається класифікація Флінна, яка описує системи за кількістю потоків інструкцій і потоків даних. Найпростішим класом є SISD – система з одним потоком інструкцій і одним потоком даних. Це фактично традиційні послідовні машини, в яких процесор

виконує одну інструкцію над одним набором даних у даний момент часу. Вони не є паралельними, але слугують базою для порівняння.

Другий клас – SIMD, де одна інструкція одночасно виконується над багатьма елементами даних. Це модель масово-паралельної обробки, що лежить в основі GPU, векторних процесорів, графічних конвеєрів та засобів прискорення мультимедіа. SIMD особливо ефективний у задачах, де однакові операції потрібно виконувати над великими масивами даних, наприклад у лінійній алгебрі або обробці зображень.

Третій клас – MISD, модель, у якій багато інструкцій застосовуються до одного потоку даних. Це дуже рідкісний і специфічний тип систем, що майже не використовується у практиці, оскільки потребує особливої архітектури та нетипових задач. Теоретично такий підхід застосовується в деяких моделях відмовостійких систем, де один набір даних обробляється різними алгоритмами для підвищення надійності.

Найважливішим класом сучасних паралельних систем є MIMD, де багато потоків інструкцій працюють над багатьма потоками даних. Це універсальна паралельна модель, яка охоплює більшість сучасних багатоядерних процесорів, серверів, кластерів і суперкомп’ютерів. У таких системах кожне ядро або вузол може виконувати свою програму над власними даними, а взаємодія між ними відбувається через спільну пам’ять або мережевий обмін.

Окрім класифікації Флінна, важливою є класифікація за організацією пам’яті. За цим критерієм перший тип – це системи зі спільною пам’яттю, де всі процесори мають доступ до одного адресного простору. Такий підхід характерний для багатопроцесорних машин і багатоядерних процесорів. Перевагою є зручність програмування, але пропускна здатність пам’яті та проблема узгодженості кешів можуть стати критичними обмеженнями.

Другий тип – системи з розподіленою пам’яттю, у яких кожен вузол має свою локальну пам’ять, а обмін даними відбувається через мережу. Це основа кластерів і суперкомп’ютерів. Програмування в таких системах

складніше, але вони значно краще масштабуються і дозволяють створювати системи з тисячами або десятками тисяч вузлів.

Існують також гетерогенні системи, що поєднують різні типи обчислювальних пристроїв – CPU, GPU, тензорні прискорювачі, FPGA та інші спеціалізовані елементи. Такі системи забезпечують паралельність на різних рівнях і дозволяють оптимізувати виконання задач залежно від характеру навантаження.

У межах паралельних систем виділяють і класи за рівнем зв’язності, тобто тим, як саме організовано обмін даними. Це може бути слабкозв’язана архітектура (характерна для кластерів) або тісно зв’язана (внутрішньопроцесорні архітектури зі спільною шиною пам’яті). Зв’язність визначає швидкість комунікації між елементами та ефективність паралельного виконання.

Таким чином, класифікація паралельних обчислювальних систем дозволяє охопити широкий спектр архітектур – від SIMD-графічних прискорювачів до масивних MIMD-кластерів, від систем зі спільною пам’яттю до розподілених гетерогенних обчислювальних комплексів. Кожен тип характеризується своїми перевагами, обмеженнями та сферою застосування, а їхнє розуміння є необхідним для ефективного використання паралельних технологій у практиці програмування та системного моделювання.

ТЕМА 1.3. БАГАТОПОТОЧНЕ (ПАРАЛЕЛЬНЕ) ПРОГРАМУВАННЯ

План

1. Клас Thread
2. Створення, виконання потоків
3. Завершення та переривання потоку
4. Переваги та недоліки при використанні потоків
5. Спадкування від класу Thread та інтерфейс Runnable
6. Синхронізація потоків

1. Клас Thread

Клас Thread у Java є базовим засобом для створення та керування потоками виконання, тобто окремими незалежними послідовностями інструкцій, що можуть виконуватися паралельно в межах однієї програми. Він лежить у центрі моделі багатопотоковості мови Java й дозволяє реалізовувати паралельні обчислення на рівні процесів та задач, забезпечуючи програмісту можливість контролювати життєвий цикл, пріоритети, стан та поведінку окремих потоків.

Ідея класу Thread полягає в тому, що кожен потік – це окремий об’єкт, який інкапсулює логіку виконання певної задачі. Щоб створити новий потік, програміст може або успадкувати власний клас від Thread і перевизначити метод run(), в якому визначено дії, що має виконати потік; або реалізувати інтерфейс Runnable та передати його у конструктор Thread. Обидва способи забезпечують однаковий ефект, але реалізація Runnable вважається гнучкішою, оскільки дозволяє уникати множинного успадкування та краще розмежовувати логіку і механізм управління потоками.

Запустити потік можна за допомогою методу start(), який створює новий системний потік і передає йому керування. Виклик методу run() без start() не створює потік, а запускає метод у звичайному послідовному режимі. Це фундаментальна різниця, яку необхідно чітко усвідомлювати при

розробці мультипоточних додатків. Після запуску потік переходить у стан runnable і може бути запланований до виконання операційною системою.

Клас Thread включає механізми контролю за перебігом виконання. Потік можна тимчасово призупинити за допомогою методу sleep(), який переводить його в стан очікування на певний час; метод join() дає можливість одному потоку чекати завершення іншого; метод interrupt() – надсилає сигнал переривання, який може бути оброблений потоком, що виконується. Потоки також мають атрибут пріоритету, який може впливати на те, як операційна система розподіляє процесорний час між потоками, хоча на практиці це залежить від конкретної реалізації планувальника.

Thread надає інформацію про власний стан. Потоки можуть перебувати у різних статусах: new (створений, але не запущений), runnable (готовий до виконання), blocked (очікування ресурсу), waiting або timed_waiting (очікування подій або часу), terminated (завершений). Ці стани важливі для діагностики паралельних програм та розуміння їхньої поведінки у складних сценаріях взаємодії.

Однією з важливих особливостей Thread є те, що створення великої кількості потоків може бути неефективним. Кожен потік потребує окремого стеку, контексту та ресурсів операційної системи. Тому в сучасних Java-додатках для масової паралельності частіше застосовують ExecutorService, Fork/Join Framework та інші високорівневі інструменти. Проте Thread залишається фундаментальною конструкцією, без якої неможливо зрозуміти внутрішню логіку роботи паралельних механізмів.

У Java також існують засоби синхронізації – synchronized, об’єкти блокувань, монітори, атомарні змінні – які тісно пов’язані з роботою потоків і необхідні для забезпечення коректного доступу до спільних ресурсів. Потоки, які працюють із загальними даними без відповідної координації, можуть створювати конфлікти, гонки даних або неконсистентні стани програми.

Таким чином, клас Thread є базовим елементом паралельного програмування в Java. Він формує основу для створення потоків, керування їхнім виконанням і взаємодії між ними. Розуміння його принципів є обов'язковою складовою вивчення технологій багатопотоковості та паралельних обчислень, оскільки саме на цьому рівні програміст уперше стикається з реальними механізмами паралельного виконання алгоритмів.

2. Створення, виконання потоків

Створення та виконання потоків у Java є центральним елементом паралельного програмування, оскільки саме через потоки програма набуває здатності виконувати кілька завдань одночасно. Потік – це незалежна послідовність інструкцій, яка працює паралельно з іншими потоками у межах одного процесу. Створення потоку означає підготовку до запуску нової гілки виконання, а виконання – передачу їй управління для паралельної роботи.

У Java існує два основні способи створення нового потоку. Перший полягає в успадкуванні класу Thread і перевизначенні його методу run(), у якому описується логіка роботи потоку. Після створення об'єкта такого класу викликається метод start(), який ініціює реальне створення потокового ресурсу у операційній системі та переносить роботу run() у паралельний режим. Другий спосіб – реалізація інтерфейсу Runnable, що є гнучкішим, оскільки дозволяє поєднувати поточну логіку з іншими класами, не обмежуючи успадкування. У цьому випадку об'єкт Runnable передається в конструктор Thread, а запуск здійснюється так само через start().

Важливо розуміти, що виклик методу run() без використання start() не створює нового потоку. У цьому випадку метод виконується у тому ж потоці, з якого був викликаний, що повністю руйнує концепцію паралельності. Саме start() забезпечує створення нового системного потоку та передачу управління на рівні операційної системи.

Після запуску потік переходить у стан runnable, що означає, що він готовий до виконання, але не обов'язково виконується саме зараз – планувальник операційної системи визначає, коли надати йому процесорний

час. Коли потік отримує доступ до ядра процесора, він переходить у стан running і виконує інструкції методу run(). Виконання може тимчасово призупинятися через виклики sleep(), очікування ресурсів, вхід у блоковані секції або взаємодію з іншими потоками.

Керування моментами виконання здійснюється кількома ключовими методами. Використання sleep() дозволяє призупинити потік на певний час, що корисно для регулювання навантаження або емуляції тривалих операцій. Метод join() дає можливість одному потоку чекати завершення іншого – це зручно для того, щоб отримати результат або продовжити роботу лише після завершення певної паралельної задачі. Метод interrupt() служить для надсилання сигналу про переривання, яким потік може скористатися, щоб припинити свою роботу чи змінити поведінку.

Потоки можуть бути як звичайними, так і daemon-потоками. Daemon-потоки виконують допоміжну роботу у фоновому режимі (наприклад, збирач сміття) та завершуються автоматично, коли завершуються всі користувацькі потоки. Зміна статусу потоку здійснюється через метод setDaemon(true), але робити це можна лише перед запуском.

Життєвий цикл потоку включає кілька станів: new (створений), runnable (готовий до виконання), blocked або waiting (очікує ресурсу чи події), timed_waiting (очікує з таймером) та terminated (завершений). Розуміння цих станів є важливим для аналізу паралельної поведінки програми: воно дозволяє діагностувати зависання, блокування, взаємні очікування та інші типові помилки багатопоточності.

Створюючи потоки, слід пам’ятати про ефективність. Велика кількість потоків може призвести до перевантаження системи через витрати на перемикання контексту та створення стеків. Тому в сучасних системах часто використовують пул потоків (ExecutorService), який дозволяє багаторазово використовувати обмежену кількість потоків. Проте концепція ручного створення потоків через Thread залишається необхідною для глибокого розуміння механізмів паралельного виконання.

Отже, створення та виконання потоків у Java – це основа багатопоточності, яка забезпечує можливість розпаралелювати програму на рівні задач і дозволяє організувати одночасне виконання незалежних частин алгоритму. Розуміння цього процесу є фундаментом для роботи з більш складними механізмами паралельних обчислень та фреймворками високого рівня.

3. Завершення та переривання потоку

Завершення та переривання потоку – це важливі аспекти керування багатопоточними програмами, оскільки вони визначають, як саме потік припиняє роботу, як інші потоки можуть чекати його завершення та яким чином можна безпечно зупинити виконання потоку, що працює в автономному режимі. У Java завершення потоку може бути природним (коли метод run() доходить до кінця) або ініційованим іншими потоками через механізми переривання. Водночас безпосереднє «примусове» зупинення потоку не підтримується, оскільки це могло б призвести до неконсистентного стану програми.

Найприроднішим способом завершення потоку є повне відпрацювання методу run(), після чого потік переходить у стан terminated. Таке завершення є безпечним і передбачуваним. Якщо іншим потокам потрібно дочекатися цього моменту, використовується метод join(), який блокує потік, що викликає його, до завершення обраного потоку. Це дозволяє організувати правильну послідовність дій у програмах, де результати паралельної роботи повинні бути об’єднані або опрацьовані після завершення всіх потоків.

Переривання потоку в Java відбувається за допомогою спеціального механізму interrupt(), який не зупиняє потік миттєво, а лише надсилає йому сигнал про бажане припинення. Потік має самостійно реагувати на цей сигнал, перевіряючи прапорець переривання через Thread.interrupted() або isInterrupted(). Найчастіше така перевірка проводиться всередині циклів або у точках, де потік може безпечно завершити свою роботу. Програміст повинен передбачити логіку завершення і звільнення ресурсів у разі переривання.

Особливо важливою є взаємодія переривання з методами, що можуть блокувати потік – наприклад, sleep(), wait() або join(). Якщо під час виконання таких методів надійде interrupt(), блокована операція завершиться винятком InterruptedException. Це дає можливість одразу відреагувати на переривання та завершити роботу потоку у контрольованому режимі. Правильна обробка цього винятку є критичною, оскільки він сигналізує про зміну логіки виконання.

У старих версіях Java існували методи stop(), suspend() та resume(), однак вони визнані небезпечними й застарілими, оскільки могли зупиняти потік у довільний момент, не даючи можливості завершити критичні секції або звільнити ресурси. Саме тому їхнє використання заборонене, і сучасні багатопоточні програми мають покладатися лише на механізм переривання та власноруч створену логіку завершення.

Для плавного завершення потоків часто використовують так звані «прапорці зупинки» – звичайні булеві змінні, бажано оголошені як volatile, які потік періодично перевіряє. Якщо прапорець набуває значення false, потік коректно виходить із циклу та завершується. Такий спосіб має перевагу простоти і не призводить до небезпеки раптового зупинення потоку, однак не працює з блокуючими операціями, де все одно потрібен interrupt().

Переривання потоків у паралельних системах нерідко пов’язане з питаннями безпеки доступу до ресурсів. Потік, що завершується, повинен коректно звільнити зайняті об’єкти синхронізації, закрити файли, завершити роботу з мережею або базою даних. Саме тому завершення потоку – це не просто припинення виконання run(), а завершення його життєвого циклу згідно з усіма правилами консистентності програми.

Таким чином, завершення і переривання потоків у Java ґрунтуються не на примусовому зупиненні, а на контрольованому співробітництві між потоками. Потік має або природно завершити run(), або коректно відреагувати на interrupt(), повністю зберігаючи логічну цілісність програми.

Такий підхід забезпечує безпечну й передбачувану роботу багатопотокових систем, що є критично важливим у паралельних обчисленнях.

4. Переваги та недоліки при використанні потоків

Використання потоків у програмуванні відкриває широкі можливості для підвищення продуктивності та ефективності програм, однак водночас накладає додаткові вимоги до розробника та створює ризики, яких немає в однопоточних застосунках. Потоки є одним із базових механізмів паралелізму, і їхня роль особливо велика у системах, де важлива швидка реакція, багатозадачність або обробка великих обсягів даних. Тому оцінку використання потоків традиційно розглядають через аналіз переваг та недоліків, які вони приносять у програму.

Основна перевага використання потоків – можливість справжньої паралельності, коли кілька частин програми виконуються одночасно. У багатоядерних процесорах це означає, що різні потоки можуть фізично працювати на різних ядрах, підвищуючи загальну продуктивність. Це особливо корисно у задачах, що легко розбиваються на незалежні частини: обробка масивів даних, паралельні обчислення, фонові задачі або будь-які повторювані операції, що працюють автономно.

Другою важливою перевагою є покращення реактивності програм. Потоки дозволяють виконувати тривалі операції (ввід/вивід, обчислення, роботу з мережею) у фоновому режимі, не блокуючи основний потік. Це критично важливо для графічних інтерфейсів, серверів, сервісів обробки запитів та інших застосунків, де користувач або клієнт очікує швидкої відповіді.

Потоки також сприяють кращій структуризації програми, коли різні компоненти можуть працювати незалежно: один потік відповідає за логіку, інший – за обробку подій, ще один – за введення-виведення. Такий поділ підвищує модульність системи та дозволяє простіше масштабувати розв’язання задач.

Однак разом із перевагами потоковість має і низку суттєвих недоліків. Найважливіший із них – складність програмування та налагодження. У багатопотокових програмах з’являється недетермінізм: порядок виконання потоків не фіксований і залежить від системного планувальника. Це може призвести до помилок, які важко відтворити, наприклад, гонок даних, коли кілька потоків одночасно змінюють одну змінну без відповідної синхронізації.

Ще одним недоліком є небезпека взаємних блокувань (deadlocks), коли кілька потоків чекають на звільнення ресурсів один одним і таким чином «завмирають». Подібні ситуації часто виникають у складних системах, де застосовується багаторівнева синхронізація. Окрім deadlock’ів, існують також livelock’и, starvation і проблеми, пов’язані з неправильним використанням примітивів синхронізації.

Важливою проблемою є витрати на перемикання контексту. Кожне переключення між потоками потребує часу та ресурсів: система повинна зберегти стан потоку і завантажити стан іншого. Якщо потоків надто багато, ефективність падає, і програма може працювати навіть повільніше, ніж у послідовному режимі. Тому надмірне створення потоків вважається анти-патерном, а сучасні системи замість цього використовують пул потоків.

Потоки також створюють підвищені вимоги до пам’яті, оскільки кожен потік отримує власний стек. У програмах, що створюють сотні або тисячі потоків, це може призводити до швидкого вичерпання оперативної пам’яті.

Ще один недолік – небезпека непослідовного доступу до спільних ресурсів. Якщо потоки працюють із загальними змінними, структурами даних або файлами, потрібна синхронізація. Проте надмірне використання синхронізації може «перетворити» паралельний код на фактично послідовний, знижуючи продуктивність. Знайти баланс між безпекою й швидкодією – одне з ключових завдань розробника паралельного програмного забезпечення.

Таким чином, потоки є потужним інструментом, що дає змогу виконувати кілька задач одночасно, підвищувати продуктивність програм і забезпечувати плавність роботи користувацьких інтерфейсів та сервісів. Але водночас вони потребують уважного проектування, обережного керування ресурсами і глибокого розуміння механізмів синхронізації та взаємодії. Чітке усвідомлення як переваг, так і недоліків потоків є необхідною умовою створення стабільних, ефективних і надійних багатопотокових програм.

5. Спадкування від класу Thread та інтерфейс Runnable

Спадкування від класу Thread та реалізація інтерфейсу Runnable – це два основні способи створення потоків у Java, і кожен із них має свої особливості, переваги та зручність використання. Обидва підходи ведуть до єдиного результату – створення паралельної гілки виконання, але відрізняються тим, як організовується логіка програми та яким чином відбувається зв’язок між кодом і механізмом роботи потоку.

Перший спосіб передбачає спадкування класу Thread, що означає створення нового класу, який наслідує функціональність Thread і перевизначає його метод run(). У такому підході сам об’єкт класу Thread є і потоком, і носієм логіки його виконання. Потік створюється просто: достатньо оголосити клас, який extends Thread, перевизначити run() і викликати метод start(). Цей спосіб інтуїтивно зрозумілий і часто використовується для простих навчальних прикладів, коли логіка потоку невелика і не потребує складної структури. Однак він має суттєве обмеження – Java не підтримує множинного успадкування, тому якщо клас уже має батьківський клас, успадкувати Thread неможливо.

Другий спосіб – реалізація інтерфейсу Runnable – є більш гнучким і професійним. У цьому випадку логіка потоку розміщується в окремому класі, що реалізує метод run(), але сам потік не створюється автоматично. Для запуску потоку потрібен об’єкт класу Thread, у конструктор якого передається екземпляр Runnable. Такий підхід розділяє «що робити» (логіку) і «як виконувати» (механізм потоку), що набагато краще відповідає

принципам об’єктно-орієнтованого програмування. Крім того, клас, який реалізує Runnable, залишається вільним для успадкування від інших класів, що часто є важливим у реальних проєктах.

Спадкування від Thread також забезпечує доступ до методів керування виконанням потоку безпосередньо зсередини класу, оскільки об’єкт потоку і логіка виконання належать одному екземпляру. Це може спростити доступ до поточного стану потоку. Водночас при використанні Runnable всі дії над потоком – запуск, переривання, очікування – здійснюються через об’єкт Thread, у той час як Runnable містить лише виконувану логіку, не будучи самостійним елементом системи потоків.

У професійній практиці саме спосіб з Runnable вважається стандартом, оскільки він масштабований і сумісний із сучасними Java-фреймворками, такими як ExecutorService або Fork/Join. Ці високорівневі засоби очікують об’єктів Runnable або Callable, а не класів, що успадковують Thread. Крім того, підхід з Runnable полегшує повторне використання коду, тестування, інкапсуляцію та розширення функціоналу.

Водночас обидва способи по суті роблять одне й те саме: надають можливість визначити блок коду, який виконуватиметься паралельно. І спадкування від Thread, і реалізація Runnable зрештою використовують той самий механізм планування потоків, той самий метод run() та однакову логіку взаємодії з операційною системою.

Таким чином, спадкування від Thread є швидким, простим і наочним способом створення потоку, але менш гнучким і обмежує можливість спадкування. Реалізація Runnable забезпечує кращу архітектурну організацію програми, дозволяє розділити логіку й механізм виконання й є рекомендованим підходом у більшості реальних застосунків. Розуміння обох способів дає змогу повністю опанувати модель потоків Java і робить програміста готовим до роботи з більш складними паралельними механізмами.

6. Синхронізація потоків

Синхронізація потоків – це механізм координації доступу різних потоків до спільних ресурсів, який забезпечує коректність і передбачуваність багатопотокових програм. У паралельному середовищі декілька потоків можуть одночасно читати, змінювати або використовувати одні й ті самі дані. Якщо доступ не контролювати, виникають типові проблеми – гонки даних, неконсистентні стани, зіпсуті структури даних або навіть повне зависання програми. Саме тому синхронізація є ключовим елементом роботи з потоками і основою безпечного паралельного програмування.

Головна мета синхронізації – гарантувати, що лише один потік у певний момент часу може виконувати критичну секцію коду, тобто таку, яка працює з ресурсами, що можуть бути змінені. У Java базовим механізмом є ключове слово synchronized, яке працює на основі моніторів. Коли потік входить у синхронізований блок або метод, він отримує монітор (замок) об’єкта. Поки монітор утримує один потік, інші потоки блокуються, чекаючи на звільнення ресурсу. Такий підхід забезпечує атомарність – тобто виконання критичного коду від початку до кінця без переривання іншими потоками.

Окрім synchronized, Java пропонує низку розширених засобів із пакету java.util.concurrent, які дозволяють точніше контролювати доступ до ресурсів. Одним із таких механізмів є ReentrantLock, який працює аналогічно до synchronized, але має більше можливостей: спробу захоплення блокування без очікування, справедливе планування потоків та можливість примусового звільнення блокування. Інші механізми, такі як ReadWriteLock, дозволяють одночасне читання багатьма потоками та виключне записування одним – що підвищує продуктивність у задачах з домінуванням операцій читання.

Синхронізація часто пов’язана і з координацією – ситуаціями, коли один потік повинен чекати результатів іншого. Для цього Java пропонує такі інструменти, як wait(), notify(), notifyAll(), що працюють із внутрішнім монітором об’єкта. Вони дозволяють потокам "спати", очікуючи певної умови, і прокидатися, коли інший потік її змінить. У сучасних системах

частіше використовують такі засоби, як CountDownLatch, CyclicBarrier, Semaphore, які надають більш контрольовані та безпечні механізми взаємодії між потоками. Наприклад, CountDownLatch дозволяє одному потоку чекати завершення декількох інших, а CyclicBarrier – синхронізувати групи потоків так, щоб вони переходили до наступної стадії одночасно.

Особливо важливою частиною синхронізації є захист спільних даних. У багатопотокових програмах поля можуть бути змінені одночасно кількома потоками, що створює небезпеку отримання неконсистентних результатів. Java забезпечує безпечний доступ через атомарні класи (AtomicInteger, AtomicReference тощо), які дозволяють виконувати операції без блокувань, використовуючи низькорівневі механізми процесора. Це дає змогу досягти високої продуктивності без традиційних замків, але вимагає розуміння їхньої семантики.

Слід пам’ятати, що синхронізація хоч і забезпечує коректність, але має і свою ціну. Кожен замок – це потенційне вузьке місце програми. Якщо потоки часто змагаються за один і той самий замок, паралельність фактично перетворюється на послідовність. Це може призводити до блокувань, взаємних очікувань або значних затримок. Тому синхронізація повинна бути мінімальною, точно спроєктованою і застосованою лише там, де це дійсно необхідно.

Отже, синхронізація потоків – це фундаментальний механізм захисту спільних ресурсів, який дозволяє організувати безпечну взаємодію між потоками та уникнути типових помилок багатопоточності. Вона потребує уважного проєктування, оскільки неправильне використання може не лише не покращити, а й суттєво погіршити продуктивність та стабільність програми. Розуміння принципів синхронізації є одним із найважливіших елементів у вивченні паралельних обчислень.

ЗМІСТОВНИЙ МОДУЛЬ 2
ОРГАНІЗАЦІЯ МІЖПОТОКОВОЇ ВЗАЄМОДІЇ
ТЕМА 2.1. ОРГАНІЗАЦІЯ МІЖПОТОКОВОЇ ВЗАЄМОДІЇ ЗА ДОПОМОГОЮ МОНІТОРА

План

1. Загальне визначення монітору при організації міжпотокової взаємодії.
2. Клас Monitor, реалізація за допомогою м’ютекса та умовних змінних.
3. Блокуюча та неблокуюча умовні змінні.
4. Взаємне виключення.
5. Дисципліни сигналізації: оператор signal.

1. Загальне визначення монітору при організації міжпотокової взаємодії

Монітор при організації міжпотокової взаємодії – це програмна абстракція, яка забезпечує взаємне виключення під час доступу до спільних ресурсів та координацію роботи потоків. Це означає, що монітор гарантує виконання критичної секції коду лише одним потоком одночасно та надає механізми, за допомогою яких потоки можуть чекати на певні умови і повідомляти один одного про їх зміну.

У своїй сутності монітор складається з трьох ключових складових:

Блокування (lock / mutex) – механізм, який дозволяє лише одному потоку володіти монітором у певний момент часу. Потік, що захоплює монітор, отримує ексклюзивний доступ до критичного коду. Інші потоки очікують, доки монітор буде звільнено.

Умовні змінні (condition variables) – засоби, які дозволяють потокам тимчасово звільняти монітор і переходити в режим очікування, доки певна умова не стане істинною.

Черги очікування – потоки, які не можуть отримати монітор або чекають на умову, потрапляють до відповідної черги та продовжують виконання лише тоді, коли монітор звільниться або буде отримано сигнал про зміну стану.

У Java кожен об’єкт має вбудований монітор, а ключове слово synchronized дозволяє потокам взаємодіяти з ним. Методи wait(), notify() та notifyAll() реалізують координацію потоків у межах монітора: wait() тимчасово звільняє монітор і переводить потік у режим очікування, notify() пробуджує один потік, а notifyAll() – усі очікуючі потоки.

Таким чином, монітор – це структурований механізм синхронізації, який забезпечує керований, безпечний і послідовний доступ до спільних ресурсів у багатопотокових програмах. Він дозволяє уникати гонок даних, неконсистентних станів і хаотичної взаємодії між потоками.

2. Клас Monitor, реалізація за допомогою м’ютекса та умовних змінних

Клас Monitor – це узагальнена програмна конструкція, яка інкапсулює механізми взаємного виключення та координації потоків, дозволяючи безпечно керувати доступом до спільних ресурсів. Його фундаментальна ідея полягає в тому, що всі операції над ресурсом здійснюються через методи монітора, які автоматично виконуються атомарно завдяки внутрішньому блокуванню. Усередині монітора також існують спеціальні засоби очікування, що дозволяють потокам призупинятися та відновлюватися залежно від стану ресурсу.

У класичній реалізації монітор складається з м’ютекса (mutex) та однієї або кількох умовних змінних (condition variables).

М’ютекс – це примітив синхронізації, який гарантує, що лише один потік може володіти ним у певний момент часу. У контексті монітора м’ютекс працює як “двері”, які забезпечують ексклюзивний вхід у критичну секцію.

Основні властивості м’ютекса:

– Тільки один потік може захопити його одночасно.

– Потоки, які не можуть захопити м’ютекс, переходять у стан очікування.

– М’ютекс гарантує атомарність операцій у межах монітора.

Фактично м’ютекс є базовим механізмом, завдяки якому методи монітора стають взаємовиключними.

Умовні змінні (condition variables) та координація потоків. Однієї лише взаємної блокади недостатньо, оскільки іноді потік повинен не лише чекати на доступ до ресурсу, а й чекати на певний стан ресурсу. Для цього використовуються умовні змінні.

Умовні змінні дозволяють:

– Перевести потік у режим очікування, тимчасово звільнивши м’ютекс.

– Пробудити очікуючий потік, коли стан ресурсу змінився.

– Координувати роботу кількох потоків, забезпечуючи передбачуваний порядок дій.

Кожна умовна змінна має:

– wait-чергу – потоки, які чекають на певну умову;

– сигнали – notify або notifyAll, які пробуджують один або всі потоки відповідно.

Як м’ютекс і умовні змінні працюють разом у моніторі:

1. Потік входить у метод монітора → захоплює м’ютекс.

2. Перевіряє необхідну умову (наприклад, ресурс зайнятий).

3. Якщо умова не виконана – викликає wait():

– потік звільняє м’ютекс;

– переходить до черги очікування умовної змінної.

4. Інший потік змінює стан ресурсу й викликає notify() або notifyAll().

5. Потік прокидається, знову захоплює м’ютекс і продовжує виконання з моменту очікування.

6. Після завершення операції звільняє м’ютекс і виходить із монітора.

Така модель забезпечує одночасно взаємне виключення та керовану координацію.

Приклад реалізації класичного монітора в Java (через Lock та Condition). У Java низькорівневі засоби для реалізації моніторів – це ReentrantLock та Condition. Нижче наведено спрощений приклад:

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Monitor {
   private final Lock lock = new ReentrantLock();
   private final Condition condition = lock.newCondition();

   private int value;
   private boolean available = false;

   public void put(int newValue) throws InterruptedException {
      lock.lock();
      try {
         while (available) {
            condition.await();
         }
         value = newValue;
         available = true;
         condition.signalAll();
      } finally {
         lock.unlock();
      }
   }

   public int get() throws InterruptedException {
      lock.lock();
      try {
         while (!available) {
            condition.await();
         }
         available = false;
         condition.signalAll();
         return value;
      } finally {
         lock.unlock();
      }
   }
}
```

Ключові особливості такої реалізації:

– lock.lock() / lock.unlock() – аналог synchronized, але більш гнучкий;

– condition.await() – еквівалент wait(), звільняє м’ютекс і переводить потік у стан очікування;

– condition.signalAll() – еквівалент notifyAll();

– Можливість створювати кілька умовних змінних для одного м’ютекса (наприклад, notEmpty, notFull), що значно зручніше ніж wait/notify.

Класичний монітор у реалізації через м’ютекс і умовні змінні гарантує ексклюзивний доступ до ресурсу, забезпечує можливість організувати очікування певних умов, дозволяє потокам взаємодіяти у контрольованій і безпечній формі та є основою всіх сучасних засобів синхронізації, зокрема в Java.

3. Блокуючі та неблокуючі умовні змінні

У контексті моніторів умовні змінні відіграють ключову роль у координації потоків, проте характер їхньої поведінки може суттєво відрізнятися залежно від способу використання. Найпоширенішими є два підходи – блокуючий і неблокуючий – які визначають, як саме потоки реагують на невиконані умови та яким чином відбувається взаємодія із захопленим м’ютексом.

Блокуюча умовна змінна передбачає, що потік, який не може продовжити роботу через невідповідний стан ресурсу, переходить у режим очікування. У момент виклику операції очікування потік автоматично звільняє м’ютекс, надаючи можливість іншим потокам змінити стан ресурсу, і лише після отримання сигналу від пробуджувального потоку знову намагається захопити блокування, щоб продовжити роботу. Такий механізм характерний для класичних реалізацій моніторів і забезпечує чітку, строго контрольовану черговість дій: жоден потік не виконує зайвих обчислень, а переходить у стан "сплячки", очікуючи на сприятливий стан даних. Саме блокуючі умовні змінні лежать в основі традиційних патернів типу "виробник–споживач" та інших сценаріїв, де ресурс може бути тимчасово недоступним або неповним.

Неблокуюча умовна змінна працює інакше. Потік не переходить у призупинений стан і не звільняє м’ютекс, якщо умова не виконана. Натомість він продовжує виконання – часто повертається до повторної перевірки умови або застосовує альтернативну логіку, наприклад, виходить з методу чи виконує компенсаційні дії. Такий підхід використовується у високопродуктивних або низькозатримкових системах, де небажано будь-яке блокування потоку. Замість абсолютного очікування застосовується модель "активного опитування" (busy-waiting), яка дозволяє уникнути контекстного перемикання потоків. Через це неблокуючі умовні змінні менш безпечні з точки зору ресурсоспоживання, але забезпечують більш передбачувану швидкість реакції.

Узагальнюючи, блокуючі умовні змінні гарантують ефективну взаємодію потоків у середовищах, де допустимо чекати на подію, тоді як неблокуючі підходять для систем, де затримка є критичною, а блокування – небажаною. Обидва механізми є важливими складовими сучасних багатопотокових моделей, а вибір між ними визначається вимогами до продуктивності, стабільності та поведінки програми.

4. Взаємне виключення

Взаємне виключення є фундаментальним принципом у побудові багатопотокових програм, оскільки саме він забезпечує коректний і безпечний доступ до спільних ресурсів у середовищі, де декілька потоків можуть звертатися до одного й того самого об’єкта одночасно. Сутність взаємного виключення полягає у тому, що в будь-який момент часу лише один потік має право виконувати критичну секцію коду – тобто фрагмент, який працює зі спільними даними або несе потенційний ризик неконтрольованої зміни стану ресурсу.

У межах моніторів принцип взаємного виключення реалізується автоматично завдяки використанню внутрішнього м’ютекса. Коли потік входить у метод монітора, він фактично «захоплює» цей м’ютекс, закриваючи доступ усім іншим потокам до критичної секції. Лише після

завершення роботи або виходу з монітора м’ютекс звільняється, і наступний потік може отримати право продовжити виконання. Такий підхід унеможливлює взаємне перекручування даних, змагання потоків за ресурс або часткове оновлення об’єктів, які повинні оброблятися атомарно.

Не менш важливим є те, що взаємне виключення в моніторі створює передбачуваний та послідовний порядок роботи потоків. Незалежно від того, у якій черзі вони очікують або з якою швидкістю працюють, доступ до ресурсу завжди відбувається строго послідовно. Це дозволяє значно спростити логіку програми, позбавляючись необхідності вручну контролювати конкурентні конфігурації або додаткові перевірки стану даних.

Хоча механізм взаємного виключення суттєво підвищує безпеку виконання, він водночас створює певні обмеження щодо продуктивності, оскільки потоки змушені чекати на звільнення ресурсу. Проте ця затримка є необхідною умовою для забезпечення цілісності даних і коректної роботи всієї системи. У результаті монітор виступає універсальним механізмом, що об’єднує взаємне виключення з керованою координацією потоків, роблячи багатопотокові застосування більш надійними та передбачуваними.

5. Дисципліни сигналізації: оператор signal

У процесах синхронізації потоків у моніторах особливе значення має механізм сигналізації, який визначає порядок пробудження потоків, що перебувають у стані очікування. Оператор signal виступає центральним елементом цієї взаємодії, оскільки саме він сповіщає один із потоків у черзі умовної змінної про те, що необхідна умова стала істинною або ресурс перейшов у такий стан, який дозволяє продовжувати виконання. Дисципліна сигналізації визначає, як саме відбувається це пробудження, яким є порядок черговості та які гарантії отримує потік, що прокидається.

У класичних моніторах використання signal передбачає, що пробуджений потік не отримує негайного доступу до критичної секції, а переходить у так звану “готову чергу”. Оскільки м’ютекс монітора залишається зайнятим потоком, який викликає signal, пробуджений потік

повинен дочекатися моменту, коли монітор буде звільнений. Лише після цього він знову зможе конкурувати з іншими потоками за право увійти до монітора і продовжити виконання саме з того місця, де був призупинений. Така поведінка формує дисципліну сигналізації, яку часто називають сигналізацією з передачею контролю через чергу готовності.

Існують також альтернативні підходи, коли сигнальна подія безпосередньо передає керування потоку, що прокидається, а потік, який викликає signal, переходить у стан очікування. У подібних моделях, характерних для деяких теоретичних конструкцій моніторів, пробуджений потік отримує м’ютекс одразу, а тому семантика очікування є ще більш передбачуваною. Проте в більшості практичних реалізацій, наприклад у Java, використовується саме стандартна дисципліна, за якої signal дає лише право пробудитися, але не гарантує негайного доступу до ресурсу.

Оператор signal, незалежно від конкретної дисципліни виконання, забезпечує важливу властивість – контрольовану передачу інформації між потоками про зміну стану ресурсу. Потоки не діють хаотично та не перевіряють стан структури безперервно, натомість реагують на сигнали, що зменшує навантаження на процесор і робить поведінку системи більш передбачуваною. Саме цей механізм визначає ефективність монітора як синхронізаційної конструкції, оскільки дозволяє водночас підтримувати взаємне виключення і забезпечувати скоординоване, впорядковане пробудження потоків.

ТЕМА 2.2. ОРГАНІЗАЦІЯ МІЖПОТОКОВОЇ ВЗАЄМОДІЇ ЗА ДОПОМОГОЮ ЗАСУВОК

План

1. Загальне визначення засувок при організації міжпотокової взаємодії.
2. Інтерфейс Lock та клас ReentrantLock. Блокування.
3. Методи засувок. Нерекурсивна версія засувки. Рекурсивна версія засувки.
4. Ключове слово volatile в Java.

1. Загальне визначення засувок при організації міжпотокової взаємодії

Засувки (latches) у системах міжпотокової взаємодії слугують одним із найпростіших і водночас найефективніших засобів синхронізації потоків, коли необхідно забезпечити очікування певної події або завершення певного етапу роботи. Сутність засувки полягає у тому, що вона дозволяє одному або декільком потокам блокувати виконання, доки інший потік не змінить її стан, фактично "відкривши" шлях для продовження обчислень. Таким чином засувка працює як логічний перемикач, який спочатку перебуває у закритому стані й утримує очікуючі потоки, а після настання потрібної умови миттєво дозволяє їм перейти до наступного етапу.

На відміну від умовних змінних, де потік може багаторазово переходити у стан очікування та пробудження, засувка зазвичай має одноразову семантику: після відкриття вона не повертається у закритий стан. Така поведінка робить її зручною у сценаріях, де необхідно координувати початок виконання кількох потоків, забезпечувати завершення попередніх обчислень або організовувати поступовий запуск складних обчислювальних процесів. У більшості сучасних мов програмування засувки реалізуються як окремі синхронізаційні примітиви, що поєднують у собі блокування та механізм сигналізації, спрощуючи розробку багатопотокових систем і

зменшуючи ризик помилок, пов’язаних із некоректним доступом до спільних ресурсів.

Їхня простота, передбачуваність та чітка односпрямована модель переходу роблять засувки важливим інструментом у побудові надійних алгоритмів синхронізації, особливо там, де необхідно гарантувати початкову або проміжну узгодженість між потоками перед переходом до складніших етапів роботи.

2. Інтерфейс Lock та клас ReentrantLock. Блокування

У сучасних багатопотокових системах інтерфейс Lock у Java відіграє ключову роль як розширений механізм синхронізації, який забезпечує програмісту значно більшу гнучкість, ніж традиційний оператор synchronized. На відміну від внутрішніх моніторів, що автоматично керують захопленням і звільненням блокування, інтерфейс Lock вимагає явного керування, дозволяючи програмі самостійно визначати момент встановлення та зняття блокади. Такий підхід відкриває можливість тонкого контролю над потоками, дає змогу реалізовувати складніші синхронізаційні сценарії та ефективніше керувати доступом до спільних ресурсів.

Однією з найважливіших реалізацій інтерфейсу Lock є клас ReentrantLock. Його назва підкреслює властивість повторного входу: потік, який уже захопив блокування, може захопити його повторно без ризику самоблокування. Це особливо важливо для методів, що викликають один одного рекурсивно або потребують доступу до критичної секції з кількох рівнів вкладеності. ReentrantLock надає додаткові можливості, недоступні у synchronized, зокрема реалізацію справедливого доступу, контроль черги очікування, негайну або обмежену за часом спробу отримання блокування через метод tryLock(), а також можливість поєднання блокад із кількома умовними змінними.

Блокування, яке забезпечує ReentrantLock, робить взаємодію потоків більш передбачуваною і керованою. Потік, що захопив lock(), отримує повний і ексклюзивний доступ до критичної секції, а всі інші потоки

переходять у стан очікування доти, доки блокування не буде звільнено. На відміну від автоматичного звільнення блокади при виході з методу, характерного для synchronized, у випадку ReentrantLock відповідальність лежить на програмістові – він має явним чином викликати unlock(). Це підсилює гнучкість, але також вимагає дисципліни, оскільки помилки в парності lock()/unlock() можуть призвести до взаємних блокувань та зависань системи.

Завдяки поєднанню функціональної потужності та передбачуваної семантики ReentrantLock став природним вибором для задач, де необхідно будувати складні протоколи взаємодії між потоками, організовувати гнучкі стратегії очікування або уникати обмежень, притаманних класичним моніторам. Таким чином, він реалізує не лише механізм взаємного виключення, але й інструмент для керування поведінкою багатопотокових систем на більш детальному рівні.

3. Методи засувок. Нерекурсивна версія засувки. Рекурсивна версія засувки.

Засувка як синхронізаційний примітив має чітко визначений набір методів, які забезпечують її роботу та регулюють взаємодію потоків. Найчастіше засувки використовуються для блокування одного або кількох потоків до моменту, коли буде виконана певна умова, після чого всі залежні потоки отримують можливість продовжити обчислення. Основна логіка засувки полягає у механізмі зменшення внутрішнього лічильника, що визначає, чи залишаються події або операції, яких ще потрібно дочекатися. Коли цей лічильник досягає нуля, стан засувки переходить у “відкритий”, і потоки, що перебували в режимі очікування, миттєво пробуджуються.

Методи засувки зазвичай включають операції очікування та сигналізації. Метод очікування блокує потік до моменту, поки засувка не відкриється, при цьому потік може бути призупинений на довільний час. Метод зменшення або “рахування вниз”, навпаки, використовується потоками, що виконують певні частини попередньої роботи; кожен виклик

цього методу зменшує внутрішній лічильник і наближає момент розблокування очікуючих потоків. Такий механізм дозволяє гнучко структурувати багатопотокові обчислення: поки всі необхідні дії не завершені, засувка залишається закритою, а після завершення – відкривається автоматично.

Нерекурсивна версія засувки є базовою формою цього синхронізаційного примітиву, який застосовується для одноразового узгодження роботи потоків. Її сутність полягає в тому, що засувка визначає єдиний момент переходу з “закритого” стану в “відкритий”, після чого вона не повертається у вихідний стан і більше не може бути повторно використана в межах того самого циклу виконання. Така модель робить нерекурсивну засувку простим, але надзвичайно ефективним інструментом для синхронізації початкових або проміжних етапів роботи, де необхідно дочекатися завершення певної події або виконання певної кількості дій.

Нерекурсивна засувка зазвичай має внутрішній лічильник, що зменшується кожного разу, коли один із підготовчих потоків виконує свою частину роботи. Потоки, які мають продовжити виконання лише після завершення цих попередніх етапів, блокуються і перебувають у стані очікування, доки значення лічильника не досягне нуля. У цей момент засувка миттєво переходить у відкритий стан, і всі потоки, які чекали на її розблокування, продовжують виконання одночасно. Оскільки засувка не передбачає повернення у повторно “закритий” стан, процес очікування та пробудження є незворотним і відбувається лише один раз.

Цей тип засувки особливо зручний у завданнях, де необхідно забезпечити синхронний старт або узгоджене завершення певної кількості потоків. Наприклад, у паралельних обчисленнях кілька потоків можуть виконувати початкові підготовчі дії, після чого основний потік очікує на повне завершення цих дій, використовуючи нерекурсивну засувку як механізм сигналізації. Аналогічно, такий тип засувки часто застосовується у тестуванні багатопотокових структур, коли необхідно гарантувати

одночасний початок роботи великої групи потоків або зафіксувати момент завершення їхньої спільної діяльності.

У силу своєї одноразової природи нерекурсивна версія засувки є надзвичайно простою у використанні та не потребує складної логіки керування, оскільки після відкриття вона не вимагає жодного додаткового обслуговування. Водночас ця простота забезпечує високу надійність і передбачуваність, що робить нерекурсивну засувку одним із найпоширеніших засобів синхронізації у паралельних системах, де одноразові бар'єри узгодження є важливою частиною обчислювального процесу.

Рекурсивна версія засувки є розширеним варіантом класичного механізму синхронізації, який на відміну від нерекурсивної моделі допускає багаторазове використання в межах одного застосування. Якщо нерекурсивна засувка відкривається лише один раз і після цього назавжди переходить у стан, що дозволяє всім потокам продовжити виконання, то рекурсивна засувка здатна періодично повертатися у вихідний, "закритий" стан, забезпечуючи циклічну синхронізацію потоків. Це робить її особливо цінною у задачах, де робота виконується фазами, а між кожною фазою необхідно забезпечити узгодлення або очікування визначеної кількості подій.

У своїй основі рекурсивна засувка так само оперує внутрішнім лічильником, але особливістю є те, що після досягнення нуля, розблокування потоків та завершення синхронізаційної точки лічильник може бути переініціалізований до нового значення. Таким чином засувка знову переходить у "закритий" стан і починає новий цикл очікування. Це дозволяє організовувати повторювані бар'єри, за якими потоки переходять до наступного етапу лише після того, як усі учасники завершили попередній. Такий механізм є необхідним у багатьох паралельних алгоритмах, де виконання розподілене на послідовні логічні фази.

У практичному застосуванні рекурсивна засувка забезпечує гнучкий спосіб керувати груповою синхронізацією потоків, зберігаючи чітку

структуру обчислювального процесу. Наприклад, у симуляціях, паралельних чисельних розрахунках або в задачах обробки потоків даних часто виникає потреба синхронізувати завершення кожного етапу, перш ніж перейти до наступного. Рекурсивна засувка дозволяє організувати такий бар'єр природно та ефективно, уникаючи складних схем блокування і знижуючи ймовірність помилок.

Попри свою функціональну потужність, рекурсивна засувка вимагає більш обережного використання, ніж одноразовий варіант. Оскільки лічильник перевизначається багаторазово, важливо забезпечити коректне оновлення його значення і строго контролювати моменти переходу між фазами. Неправильна ініціалізація або передчасне відкриття засувки може призвести до розсинхронізації потоків, що негативно вплине на логіку багатопотокового алгоритму.

У підсумку рекурсивна версія засувки є універсальним механізмом циклічної синхронізації, який дозволяє ефективно структурувати багатоетапні паралельні обчислення. Вона забезпечує рівновагу між гнучкістю та визначеністю, даючи змогу створювати складні, фазово узгоджені системи, що працюють прогнозовано та стабільно.

У результаті обидві версії засувок – нерекурсивна та рекурсивна – відіграють важливу роль у синхронізації багатопотокових систем. Нерекурсивна підходить для одноразових точок узгодження, тоді як рекурсивна дозволяє багаторазово організовувати синхронізацію у складних алгоритмах, що складаються з послідовних фаз. Обираючи між ними, програміст має враховувати характер задачі, структуру обчислювальних потоків і потребу в повторному або одноразовому узгодженні виконання.

4. Ключове слово volatile в Java

Ключове слово volatile у Java використовується як спеціальний маркер для змінних, що беруть участь у міжпотоковій взаємодії, і відіграє важливу роль у забезпеченні коректного та передбачуваного доступу до пам'яті у багатопотоковому середовищі. Його основне призначення полягає у тому,

щоб гарантувати видимість змін, які один потік вносить у змінну, для всіх інших потоків без необхідності використання повноцінних механізмів блокування. У звичайних умовах кожний потік працює з локальними копіями даних, розміщених у кешах процесора, тому оновлення змінної в одному потоці може залишатися невідомим іншим потокам протягом певного часу. Позначення змінної як volatile усуває цю проблему, примушуючи JVM працювати з нею безпосередньо в основній пам’яті.

Особливість volatile полягає в тому, що воно гарантує не лише видимість змін між потоками, але й встановлює певний рівень упорядкованості виконання операцій. Завдяки цьому будь-які записані у volatile-змінну значення стають доступними для всіх потоків негайно після виконання запису, а всі операції, які передують цьому запису, також стають “видимими” для інших потоків. Це означає, що volatile створює своєрідний бар’єр пам’яті, усуваючи можливість небезпечного оптимізаційного переставляння інструкцій, яке могло б призвести до неузгоджених станів при конкурентному доступі.

Попри свою корисність, volatile не замінює повноцінні блокування, такі як synchronized або ReentrantLock. Воно не гарантує атомарності складних операцій, наприклад інкрементів або модифікацій структур даних, де потрібне захоплення м’ютекса. Змінна volatile гарантує лише атомарність операцій читання і запису, не поширюючи цю властивість на комплексні обчислення. Саме тому volatile найефективніше використовується у ситуаціях, де необхідно лише забезпечити синхронізовану видимість значень, а не захистити складну логіку зміни стану ресурсу.

У підсумку ключове слово volatile виступає легким і продуктивним механізмом синхронізації, який не створює блокувань і не уповільнює виконання програми, водночас забезпечуючи надійну видимість змін між потоками. Воно особливо корисне для реалізації сигналів завершення, маркерів стану або простих прапорців керування, де повноцінне блокування є надмірним, а гарантій видимості достатньо для правильної роботи системи.

ТЕМА 2.3. ОРГАНІЗАЦІЯ МІЖПОТОКОВОЇ ВЗАЄМОДІЇ ЗА ДОПОМОГОЮ СЕМАФОРІВ

План

1. Загальне визначення семафорів при організації міжпотокової взаємодії.
2. Клас Semaphore. Конструктори та методи класу Semaphore.
3. Простий семафор. Використання семафорів для сигналізації.
4. Рахуючий семафор. Обмежуючий семафор.

1. Загальне визначення семафорів при організації міжпотокової взаємодії

Семафори є одним із найстаріших і водночас найважливіших механізмів синхронізації потоків, які використовуються для керування доступом до спільних ресурсів у багатопотокових системах. Їх концепція була запропонована Едсгером Дейкстрою і стала фундаментом для розвитку сучасних підходів до організації конкурентного програмування. У своїй сутності семафор є інтегральним лічильником, що відображає кількість «дозволів» на виконання певної операції або на доступ до ресурсу. Потік, який хоче скористатися ресурсом, повинен отримати цей дозвіл, а у разі його відсутності – перейти в режим очікування до моменту, коли інший потік звільнить один із дозволів.

Семафори забезпечують природний і гнучкий механізм контролю, завдяки якому можна не лише гарантувати взаємне виключення, але й регулювати кількість потоків, яким одночасно дозволено працювати з одним і тим самим ресурсом. На відміну від традиційного блокування, де доступ обмежений одним потоком, семафор може дозволяти одночасне використання ресурсу декількома потоками – настільки, наскільки це передбачено значенням його лічильника. Це робить семафори універсальним засобом для реалізації шаблонів керування, таких як обмежений пул ресурсів,

контроль доступу до мережевих підключень, регулювання кількості одночасно активних завдань тощо.

У процесі роботи семафор може як зменшувати, так і збільшувати свій лічильник, реагуючи на дії потоків. Коли потік захоплює «дозвіл», значення лічильника зменшується, і якщо воно досягає нуля, подальші потоки змушені чекати. Після завершення роботи з ресурсом потік повертає дозвіл, збільшуючи лічильник і тим самим відкриваючи можливість іншим потокам продовжити роботу. Такий цикл створює чітку та передбачувану модель взаємодії, де стан ресурсу постійно контролюється за допомогою лічильника семафора.

Таким чином, семафори виступають гнучким інструментом організації міжпотокової взаємодії, який дозволяє ефективно поєднувати контроль доступу, регулювання паралелізму та забезпечення синхронності. Їхня універсальність і математична простота роблять їх надзвичайно важливою частиною сучасних систем конкурентного програмування.

2. Клас Semaphore. Конструктори та методи класу Semaphore

У Java клас Semaphore є повноцінною реалізацією класичної ідеї семафора, запропонованої Дейкстрою, і слугує універсальним інструментом для керування доступом до спільних ресурсів у багатопотокових системах. Він ґрунтується на внутрішньому лічильнику дозволів, які потоки можуть «отримувати» або «повертати», що дозволяє регулювати рівень паралелізму та визначати максимальну кількість потоків, які можуть одночасно користуватися певним ресурсом. На відміну від звичайних блокувань, які забезпечують взаємне виключення лише для одного потоку, семафор дає можливість створювати гнучкі механізми доступу, де кількість одночасних користувачів може бути більше одного.

Конструктори класу Semaphore дозволяють створити семафор з певною кількістю дозволів і, за необхідності, визначити політику справедливості доступу. Початкове значення лічильника задає максимально можливу кількість потоків, яким одночасно дозволено виконувати критичну секцію.

Режим справедливості визначає, чи отримуватимуть потоки доступ у порядку їхнього приходу, чи JVM самостійно обиратиме потоки – зазвичай з метою оптимізації продуктивності. Така гнучкість дає змогу адаптувати семафор до очікуваного сценарію: від високопродуктивних систем, де корисна несправедлива стратегія, до систем реального часу, де важлива строгість і передбачуваність черговості.

Методи класу Semaphore забезпечують основну логіку взаємодії потоків із семафором. Метод acquire() змушує потік чекати, доки не з'явиться вільний дозвіл, і лише тоді дозволяє продовжити виконання. Під час очікування потік автоматично блокується, не витрачаючи ресурси процесора. Існують також варіанти цього методу з тайм-аутом або без блокування, що дозволяє будувати більш чутливі до часу алгоритми. Повернення дозволу здійснюється методом release(), який збільшує лічильник та пробуджує один або кілька потоків, що перебувають у черзі очікування. Гарантії, які надає release(), забезпечують узгоджений, передбачуваний доступ до ресурсу навіть у складних конкурентних сценаріях.

Завдяки такій структурі клас Semaphore стає надзвичайно корисним інструментом у багатопотоковому програмуванні. Він дозволяє обмежувати кількість одночасних операцій, будувати пули ресурсів, контролювати навантаження на систему й ефективно керувати паралельною обробкою даних. Його простота та строгість математичної моделі роблять його одним із найнадійніших механізмів синхронізації у Java.

3. Простий семафор. Використання семафорів для сигналізації

Простий семафор є базовою та водночас надзвичайно важливою формою семафора, який використовують для синхронізації потоків у ситуаціях, де необхідно забезпечити контроль доступу або передати сигнал про настання певної події. Його структура визначається одним лічильником дозволів, значення якого або дорівнює одиниці (бінарний семафор), або може бути більшим, регулюючи кількість потоків, що отримують доступ до ресурсу одночасно. У своїй найпростішій формі такий семафор працює як

“пропускний квиток”: коли лічильник є додатним, потік може безперешкодно продовжувати виконання; коли ж його значення знижується до нуля, наступні потоки змушені чекати на відновлення дозволу.

Особливістю простого семафора є його здатність виконувати не лише функцію контролю доступу, але й роль засобу сигналізації. Потік, який виконує певний фрагмент роботи або очікує завершення операції з боку іншого потоку, може використовувати семафор як механізм сповіщення про те, що певна подія вже відбулася. У такому сценарії семафор працює подібно до засувки або прапорця, але має перевагу у тому, що забезпечує блокуюче очікування з повним звільненням процесорного часу. Потік, що викликає acquire(), переходить у стан очікування доти, доки інший потік, завершивши свою роботу, викличе release() і тим самим надішле сигнал про можливість продовження.

Це робить простий семафор особливо ефективним інструментом у моделюванні однонаправлених залежностей між потоками. Наприклад, один потік може виробляти дані, а інший – чекати на їхню готовність, при цьому семафор забезпечує строгий порядок дій без активного опитування та зайвих обчислень. Аналогічно семафори можна використовувати для синхронізації етапів у паралельних алгоритмах, де одне обчислення не може розпочатися до завершення попереднього.

У підсумку простий семафор є надзвичайно гнучким та ефективним механізмом, що поєднує в собі як можливість обмеження паралельного доступу, так і засіб надійної сигналізації між потоками. Його застосування дозволяє будувати прості та елегантні схеми координації, які зберігають передбачуваність виконання та забезпечують високий рівень контрольованості взаємодії між потоками.

Семафори, окрім класичного застосування для контролю доступу до ресурсів, часто використовуються як універсальний механізм сигналізації між потоками. У цьому контексті вони виконують роль передавача подій: один потік подає сигнал про те, що певний етап обчислення завершено або

ресурс перейшов у новий стан, тоді як інший потік очікує на появу цього сигналу, перш ніж продовжити власне виконання. Така модель забезпечує чітку та синхронізовану взаємодію, яка не потребує використання активного опитування, зменшуючи навантаження на процесор та підвищуючи загальну ефективність системи.

Коли семафор використовується як сигнал, його внутрішній лічильник фактично ототожнюється з кількістю доступних повідомлень. Потік, який очікує на подію, викликає операцію acquire() і переходить у стан блокування, якщо лічильник дорівнює нулю. Потік-ініціатор події, навпаки, викликає release(), чим збільшує значення лічильника та пробуджує один із потоків, що перебувають у черзі очікування. Таким чином семафор стає втіленням класичної схеми "очікування–сигнал", дозволяючи зафіксувати факт завершення обчислення без використання інших засобів синхронізації.

Перевага семафора над умовними змінними у цьому сценарії полягає в тому, що він може акумулювати сигнали. Якщо потік надішле кілька сигналів поспіль, а інший потік у цей час не виконує acquire(), усі ці сигнали будуть збережені у вигляді збільшеного лічильника. Це істотно відрізняє семафор від механізму notify, де сигнал, отриманий у момент відсутності очікуючого потоку, може бути втрачений. Завдяки цьому семафор надає більш стабільну та передбачувану модель взаємодії, особливо у випадках, коли важливо не пропустити жодної події.

Семафори широко використовуються для узгодження роботи потоків у ситуаціях, де певний фрагмент коду не повинен виконуватися, доки інший потік не завершив підготовчу роботу. Наприклад, головний потік може чекати, поки один або більше допоміжних потоків завершать ініціалізацію, або потік-споживач може блокуватися, доки потік-виробник не повідомить про готовність даних. У таких сценаріях семафор забезпечує інтуїтивний та ефективний спосіб зв’язку, який мінімізує ризики гонок даних і забезпечує строгий порядок виконання.

У підсумку використання семафорів для сигналізації надає гнучкий та надійний спосіб організації взаємодії між потоками, який поєднує блокуюче очікування, збереження сигналів та чітке управління залежностями. Завдяки простоті та математичній визначеності такий підхід залишається одним із найефективніших у ситуаціях, де необхідно забезпечити передбачувану та контрольовану передачу подій.

4. Рахуючий семафор. Обмежуючий семафор.

Рахуючий семафор є розширеною формою класичного семафора, яка дозволяє контролювати доступ до ресурсу не одного, а одразу кількох потоків. Його ключовою характеристикою є лічильник, що може набувати будь-якого невід'ємного значення і визначає максимальну кількість потоків, яким дозволено одночасно виконувати певну операцію або користуватися спільним ресурсом. На відміну від бінарного семафора, у якому лічильник обмежений значеннями 0 або 1, рахуючий семафор забезпечує значно ширший діапазон керування паралелізмом, роблячи його незамінним у задачах, де ресурс має обмежену, але не одиничну пропускну здатність.

Принцип роботи рахуючого семафора ґрунтується на поступовому зменшенні та збільшенні значення лічильника. Кожен потік, що звертається до ресурсу, спершу намагається зменшити лічильник, і якщо його значення залишається додатним, потік отримує дозвіл на виконання. Коли ж лічильник досягає нуля, це означає, що доступ до ресурсу вичерпаний, і потоки, які надходять пізніше, змушені чекати, доки інші потоки не звільнять дозволи, повернувши їх через відповідний метод. Така модель забезпечує природний спосіб регулювання кількості одночасних дій і дозволяє збалансувати навантаження у багатопотокових системах.

Рахуючий семафор особливо корисний у сценаріях, де ресурс має фіксовану місткість. Наприклад, його можна застосовувати для обмеження кількості одночасних запитів до сервера, регулювання розміру пулу потоків, контролю доступу до обмеженого набору підключень або управління кількістю одночасних обчислень у складних алгоритмах. Завдяки гнучкості

лічильника розробник може тонко налаштовувати рівень паралельності, забезпечуючи оптимальне використання ресурсів і уникаючи перевантаження системи.

Важливою особливістю рахуючого семафора є його здатність природним чином організовувати черги очікування. Потоки, які не отримали дозволу через недостатню кількість доступних ресурсів, не виконують зайвих операцій і не займають процесорний час, а переходять у стан блокування до моменту, коли дозволи знову стануть доступними. Завдяки цьому рахуючий семафор забезпечує ефективне, енергозберігаюче та передбачуване керування доступом, що особливо важливо у високонавантажених або ресурсно обмежених системах.

У підсумку рахуючий семафор є потужним інструментом для організації паралельної взаємодії між потоками, дозволяючи контролювати одночасний доступ до ресурсу з високою точністю і гнучкістю. Його використання сприяє створенню стабільних, збалансованих і керованих багатопотокових програмних систем.

Обмежуючий семафор є різновидом рахуючого семафора, який використовується для регулювання рівня паралелізму шляхом обмеження кількості потоків, що можуть одночасно виконувати певну дію або отримувати доступ до спільного ресурсу. Його головною характеристикою є те, що він встановлює чітку межу – максимальну кількість дозволів, які можуть бути доступні системі. Така межа попередньо визначається під час створення семафора і не може бути перевищена, що гарантує стабільність взаємодії та запобігає надмірному навантаженню на ресурс.

У межах роботи обмежуючого семафора потоки отримують дозволи через виклик операції acquire(), яка зменшує лічильник доступних ресурсів. Коли доступні дозволи вичерпуються, наступні потоки автоматично переходять у стан очікування, доки інші потоки не виконають release() і тим самим не звільнять дозвіл. Таким чином обмежуючий семафор створює природний механізм "запобіжника", не дозволяючи системі перевищувати

визначений рівень одночасної активності та захищаючи ресурс від перевищення своєї пропускної здатності.

Обмежуючі семафори найчастіше застосовуються в системах, де існує обмежена кількість одночасних операцій, як-от у пулах підключень до баз даних, файлових системах, мережевих серверах або багатопотокових обчислювальних алгоритмах, де одночасна робота великої кількості потоків може призвести до перевантаження або деградації продуктивності. Встановлення межі дозволів забезпечує контрольовану пропускну здатність і дозволяє системі працювати стабільно навіть під великим навантаженням.

Важливою властивістю обмежуючого семафора є його здатність зберігати інформацію про кількість наданих дозволів і жорстко гарантувати, що жоден потік не зможе обійти встановлені обмеження. Як наслідок, розробник може бути впевнений, що навіть у найскладніших сценаріях паралельної роботи ресурс ніколи не буде перевикористаний. Таким чином, обмежуючий семафор виступає невід’ємною частиною систем, де необхідно забезпечити баланс між високою продуктивністю і безпекою доступу до обмежених ресурсів.

ТЕМА 2.4. ОДНОЧАСНИЙ ЗАПУСК ПОТОКІВ ТА ВІДСТЕЖЕННЯ МОМЕНТУ ЗАКІНЧЕННЯ РОБОТИ ДЕКІЛЬКОХ ПОТОКІВ

План

1. Синхронізація робочих процесів. Клас CountDownLatch та його методи.
2. Принцип синхронізації робочих процесів за часом.
3. Алгоритми розподілених систем: централізований алгоритм, розподілений алгоритм, алгоритм Token Ring,
4. Об'єкт перерахування TimeUnit для переривання потоку.

1. Синхронізація робочих процесів. Клас CountDownLatch та його методи.

Синхронізація робочих процесів у багатопотокових системах є необхідною умовою для забезпечення узгодженого виконання завдань, які взаємозалежні або повинні починатися у строго визначеній послідовності. У таких випадках важливо мати механізм, який дозволяє одному або кільком потокам чекати, доки інші потоки завершать певний обсяг роботи. Саме для цього у Java існує клас CountDownLatch – спеціальний інструмент синхронізації, що реалізує концепцію одноразового бар’єра та дозволяє ефективно організувати взаємодію між потоками.

CountDownLatch працює на основі внутрішнього лічильника, значення якого задається під час створення об’єкта. Кожний потік, що виконує певне завдання, після його завершення викликає метод, який зменшує цей лічильник на одиницю. Потоки, що мають почати виконання лише після завершення всіх попередніх дій, викликають операцію очікування і залишаються заблокованими доти, доки лічильник не досягне нуля. У момент, коли всі необхідні події відбулися, CountDownLatch відкривається, і всі потоки, що перебували в режимі очікування, продовжують свою роботу без додаткової взаємодії.

Однією з найважливіших характеристик CountDownLatch є його одноразова природа. Після того як лічильник зменшився до нуля, засувка не може бути встановлена повторно. Це робить його аналогом нерекурсивної засувки – механізмом для одноразового узгодження фази роботи або завершення певної частини обчислень. Такий підхід ідеально підходить для ініціалізації системи, координації запуску групи потоків, або очікування завершення їхньої спільної роботи перед виконанням фінальних дій.

CountDownLatch надає два фундаментальні методи, що визначають його поведінку. Метод await() використовується потоками, які повинні чекати на настання визначеної події. Він переводить потік у стан блокування, з якого він виходить лише тоді, коли лічильник стане нульовим. Метод countDown(), навпаки, викликається потоками, що виконують частини підготовчої роботи. Кожен виклик цього методу зменшує значення лічильника на одиницю. Як тільки кількість викликів дорівнює початковому значенню лічильника, latch автоматично “спрацьовує”, і очікуючі потоки переходять до виконання.

CountDownLatch завдяки своїй простоті та передбачуваній семантиці стає одним із найпоширеніших інструментів координації потоків у Java. Його використання дозволяє організувати чіткі точки синхронізації між процесами виконання, забезпечити правильну послідовність дій та уникнути ситуацій, коли певний потік починає працювати до того, як усі попередні етапи були завершені. Завдяки одноразовому механізму “спрацювання” CountDownLatch особливо ефективний у сценаріях початкової ініціалізації, паралельної обробки й фінального узгодження результатів.

Одним із класичних прикладів використання CountDownLatch є синхронізований запуск групи потоків. У цьому випадку основний потік створює latch із лічильником, що дорівнює кількості потоків, яким потрібно стартувати одночасно. Кожний із потоків після свого створення викликає await(), тим самим очікуючи сигналу про старт. Коли всі потоки готові, головний потік викликає countDown() потрібну кількість разів або робить це

в межах окремого узгоджувального механізму. Як тільки лічильник доходить до нуля, усі очікуючі потоки виходять із блокування та фактично розпочинають роботу одночасно, що може бути особливо важливим у тестуванні паралельних алгоритмів або моделюванні високонавантажених систем.

Іншим поширеним сценарієм є очікування основним потоком завершення роботи допоміжних потоків. У складних застосуваннях часто виникає ситуація, коли необхідно дочекатися, доки всі паралельні задачі виконані, перш ніж перейти до подальшого обчислювального етапу або формування підсумкового результату. У такому випадку головний потік викликає await(), а кожен із робочих потоків після завершення своєї частини роботу викликає countDown(). Завдяки цьому програміст може бути впевнений, що продовження виконання відбувається лише після того, як усі задачі виконані повністю, і жоден із потоків не залишився “позаду”.

CountDownLatch також ефективно використовується для організації поетапного виконання складних алгоритмів, коли кожна фаза залежить від завершення попередньої. Потоки можуть бути розподілені за фазами обчислення, і після завершення першої з них весь процес синхронізується за допомогою latch. Після цього виконується наступна фаза, і так далі. Така модель дозволяє структурувати багатоступеневі обчислення та уникнути хаотичності, яка може виникати у середовищах з великою кількістю конкурентних потоків.

У всіх цих випадках CountDownLatch виступає універсальним інструментом у координації потоків, дозволяючи легко створювати контрольовані точки синхронізації та підтримувати чітку логіку виконання. Його використання значно підвищує надійність і передбачуваність роботи паралельних програм, зменшуючи ризик помилок, пов’язаних із несвоєчасним або неправильним узгодженням дій між потоками.

2. Принцип синхронізації робочих процесів за часом

Синхронізація робочих процесів за часом є одним із ключових підходів до організації багатопотокового виконання, коли взаємодія між потоками визначається не лише логічними залежностями, а й часовими обмеженнями. У такій моделі певні події або фази обчислення повинні відбутися у конкретний момент або після настання визначеної часової умови, що дозволяє координувати роботу потоків у більш передбачуваний і контрольований спосіб. Такий підхід широко застосовується у системах реального часу, високонавантажених сервісах, а також у задачах, де важливо гарантувати, що певний етап виконання не розпочнеться раніше або пізніше встановленого моменту.

Сутність синхронізації за часом полягає у тому, що потоки узгоджують початок своєї роботи або перехід між етапами відповідно до зовнішнього або внутрішнього таймера. Це можуть бути як реальні часові інтервали, так і відносні затримки, які гарантують, що всі учасники обчислювального процесу рухаються у синхронізованому ритмі. Часові бар'єри забезпечують можливість створювати симетрію у роботі потоків: усі вони "вирівнюються" у часі, перш ніж перейти до наступної фази, що усуває можливість ситуацій, коли окремі потоки випереджають або відстають від загального ходу алгоритму.

Подібний механізм дозволяє ефективно координувати робочі процеси у сценаріях, де важливо дотримуватися чітких часових рамок. Наприклад, в паралельних симуляціях або обчислювальних моделях кожен крок повинен виконуватися одночасно всіма потоками, щоб забезпечити узгодженість системи. У високонавантажених серверних застосуваннях синхронізація за часом дозволяє рівномірно розподіляти навантаження, уникати пікових перевантажень і забезпечувати реактивність системи. У задачах з періодичними операціями часова синхронізація дає змогу запускати робочі потоки в такт із визначеним графіком, що формує стабільний і ритмічний процес виконання.

Синхронізація за часом також часто поєднується з іншими засобами узгодження – бар'єрами, латчами або семафорами. Часові обмеження доповнюють структурні точки синхронізації, створюючи багаторівневу модель керування потоками. Це поєднання дозволяє уникнути ситуацій, коли потік може чекати надто довго або, навпаки, продовжити виконання занадто рано. Таким чином забезпечується баланс між строгістю виконання та гнучкістю у моделюванні робочих процесів.

У результаті принцип синхронізації робочих процесів за часом утворює основу для розробки систем, де передбачуваність і ритмічність виконання мають критичне значення. Він дозволяє структурувати роботу потоків у чітко визначеному часовому просторі, забезпечуючи стабільність і узгодженість усієї багатопотокової системи.

У контексті синхронізації робочих процесів за часом CountDownLatch відіграє особливо важливу роль, оскільки дозволяє поєднувати структурну й часову організації виконання. Сутність синхронізації за часом полягає в тому, щоб забезпечити перехід між фазами обчислення у визначені часові моменти або після проходження контрольованих затримок. CountDownLatch у такій моделі може виступати як інструмент, що узгоджує завершення робочих потоків до настання певного моменту часу. Наприклад, у симуляційних або високонавантажених системах кілька потоків можуть виконувати незалежні обчислення протягом певного інтервалу, після чого усі вони повинні синхронізуватися, щоб перейти до нового тактового кроку. Завдяки CountDownLatch можна гарантувати не лише логічне, а й часово узгоджене завершення фаз.

Оскільки CountDownLatch є одноразовим бар'єром, він особливо зручний у тих випадках, де важливо чітко зафіксувати передчасний або запізнілий перехід між етапами. Потоки, що виконали всі дії до встановленого моменту, зменшують лічильник і фактично сигналізують про готовність опрацювання до наступного кроку. Ті ж потоки, які приходять у точку синхронізації раніше, змушені очікувати, доки решта не завершить свої

завдання або не настане відповідний часовий бар'єр. Таким чином CountDownLatch допомагає формувати впорядковані, ритмічні структури виконання, у яких часові межі стають частиною загального алгоритму.

Об'єднання принципу синхронізації за часом із CountDownLatch дозволяє створювати системи, що працюють у такт із визначеною логікою і ритмом виконання. Це особливо важливо у задачах, де часові обмеження мають вирішальне значення: від симуляцій фізичних процесів і аналізу потокових даних до високонавантажених серверних систем, де кожна фаза роботи повинна починатися лише за умови, що всі попередні дії завершені й час для їхнього виконання вже настав. Такий підхід робить CountDownLatch не лише засобом синхронізації потоків, а й інструментом, що сприяє більш строгій та передбачуваній організації часу виконання всього застосування.

3. Алгоритми розподілених систем: централізований алгоритм, розподілений алгоритм, алгоритм Token Ring

Централізований алгоритм у розподілених системах є одним із найпростіших та водночас найінтуїтивніших способів організації взаємодії між вузлами, коли необхідно розв'язати задачу координації або розподілу спільних ресурсів. Його основна ідея полягає в тому, що вся відповідальність за прийняття рішень покладається на один спеціально виділений вузол – «центральний координатор». Саме він відповідає за керування доступом до критичних ресурсів, синхронізацію дій у системі або контроль за виконанням певних операцій, тоді як інші вузли виступають у ролі підлеглих компонентів, що звертаються до координатора з відповідними запитами.

У межах такого підходу координація відбувається через чітко визначений протокол обміну повідомленнями. Коли один із вузлів хоче отримати доступ до ресурсу або виконати операцію, що може конфліктувати з діями інших вузлів, він надсилає запит центральному координатору. Той перевіряє поточний стан ресурсу й ухвалює рішення, дозволити або відкласти виконання дії. Координатор записує всі активні використання ресурсів і гарантує, що в системі ніколи не виникнуть конфлікти або

порушення узгодженості. Завдяки цьому централізований алгоритм є надзвичайно прозорим: усі залежності та право доступу до ресурсів концентруються в одній точці, яка виступає «арбітром» для всієї розподіленої системи.

Однією з ключових переваг такого підходу є його простота. Координатор може ефективно керувати доступом, не вдаючись до складних схем обміну повідомленнями між усіма учасниками системи. Також це дозволяє уникати більшості помилок, пов’язаних із децентралізованими рішеннями, де кожен вузол повинен узгоджувати свої дії з іншими. Централізований алгоритм забезпечує високу узгодженість і передбачуваність, оскільки всі рішення походять із єдиного джерела.

Попри значну простоту, централізований алгоритм має й низку суттєвих недоліків. Найважливішим серед них є залежність усієї системи від одного вузла, який фактично стає «точкою відмови». У разі збою координатора вся система втрачає можливість ухвалювати рішення, що може призвести до повного припинення роботи. Також централізований підхід створює обмеження щодо продуктивності: із зростанням кількості запитів координатор може стати «вузьким місцем», уповільнюючи роботу всієї системи. Це робить централізований алгоритм менш придатним для масштабованих або високонавантажених середовищ.

Проте, незважаючи на ці обмеження, централізований алгоритм залишається важливим базовим шаблоном у розподілених системах, особливо там, де простота й передбачуваність важливіші за високу відмовостійкість або масштабованість. Він часто використовується в системах навчального призначення, невеликих кластерах, експериментальних середовищах або як перша модель синхронізації перед переходом до складніших, децентралізованих підходів. Завдяки своїй концептуальній ясності централізований алгоритм формує фундамент для розуміння більш складних механізмів координації у розподілених системах.

Розподілений алгоритм є фундаментальною концепцією у сфері розподілених систем, де обчислення виконуються на кількох незалежних вузлах, які взаємодіють один з одним через мережу. На відміну від централізованого підходу, де один координатор ухвалює рішення для всієї системи, розподілений алгоритм передбачає рівноправну участь усіх вузлів, кожен із яких може діяти автономно, виконувати локальні обчислення та обмінюватися повідомленнями з іншими учасниками. Основна ідея такого алгоритму полягає в тому, що складне завдання розбивається на окремі частини, які виконуються паралельно, а результат формується через узгоджену взаємодію між вузлами.

Розподілені алгоритми відзначаються тим, що вони не покладаються на єдиний центральний елемент керування, а тому здатні забезпечувати вищу відмовостійкість і масштабованість. Кожен вузол має власний набір знань про систему, який може бути неповним або застарілим, однак алгоритм повинен гарантувати, що система завжди досягне узгодженого стану, навіть за умов затримок, втрат мережевих пакетів або відмов окремих компонентів. Через це розподілені алгоритми зазвичай будуються на чітких правилах обміну повідомленнями, використанні лічильників, маркерів, часових міток або спеціальних механізмів логічного часу.

Одним із ключових завдань розподіленого алгоритму є координація дій між вузлами без централізованого контролю. Наприклад, у задачах взаємного виключення, де кілька вузлів можуть претендувати на доступ до спільного ресурсу, розподілений алгоритм повинен гарантувати, що доступ буде надано лише одному учаснику. Для цього використовуються схеми обміну запитами та підтвердженнями, які дозволяють усім вузлам узгоджувати свої дії і уникати гонок або неконсистентних станів. Аналогічно, у випадку вибору координатора розподілений алгоритм забезпечує, що всі вузли отримають однакову інформацію про те, хто є лідером, навіть якщо система зазнає мережевих збоїв або асинхронних затримок.

Особливістю розподілених алгоритмів є їхня здатність працювати в умовах неповної інформації. Кожен вузол має доступ лише до локального стану й обмежених відомостей про сусідні вузли, а глобальний стан системи не є доступним безпосередньо. Це створює низку теоретичних викликів, пов’язаних із досягненням узгодженості, уникненням тупиків і запобіганням конкурентних конфліктів. Задля цього розробляються складні моделі обміну повідомленнями, тайм-аутів, механізмів підтвердження доставки, а також логічних годинників, які дозволяють узгоджувати події в системі навіть за відсутності глобального фізичного часу.

Розподілені алгоритми становлять основу для роботи сучасних кластерних систем, хмарних сервісів, розподілених баз даних і високонадійних мережевих протоколів. Вони дозволяють будувати системи, що залишаються працездатними навіть за часткових збоїв, забезпечуючи рівновагу між узгодженістю, продуктивністю та масштабованістю. У сукупності ці алгоритми формують складну й водночас надзвичайно гнучку архітектуру, яка є основою сучасних розподілених технологій.

Алгоритм Token Ring є одним із класичних розподілених підходів до організації взаємного виключення в системах, де кілька вузлів повинні координувати свої дії без наявності центрального керуючого елемента. Основна ідея цього алгоритму полягає у передачі спеціального повідомлення, відомого як токен, між вузлами, що утворюють логічне кільце. Токен є єдиним маркером, який дозволяє вузлу увійти до критичної секції. Лише вузол, який володіє токеном, має право отримати ексклюзивний доступ до спільного ресурсу, після чого він повинен передати токен наступному учаснику кільця.

Вузли в алгоритмі Token Ring утворюють упорядковану структуру, де кожен вузол знає про свого наступника, і саме через цю топологію здійснюється передача токена. Якщо вузлу не потрібно входити до критичної секції, він просто переадресовує токен далі по кільцю. Якщо ж вузол має запит на доступ, він затримує токен на час виконання своєї критичної

операції, а після завершення одразу передає його наступному вузлу. Такий механізм забезпечує сувору впорядкованість та запобігає конфліктам, оскільки в будь-який момент часу в системі перебуває лише один токен.

Алгоритм Token Ring має низку важливих переваг. Він повністю усуває можливість виникнення гонок за ресурс, оскільки одночасний доступ просто неможливий. Крім того, алгоритм не потребує широкомасштабного обміну повідомленнями між усіма вузлами, оскільки комунікація відбувається лише між сусідніми учасниками логічного кільця. Це значно зменшує навантаження на мережу та забезпечує передбачуваність часу доступу до ресурсу. Кожен вузол у кільці гарантовано отримає токен рано чи пізно, що робить алгоритм справедливим і виключає можливість голодування.

Попри свою елегантність, алгоритм Token Ring має й певні недоліки. Найбільш очевидною проблемою є залежність від токена. Якщо токен буде втрачено – наприклад, унаслідок помилки на рівні мережі або збою вузла – система не зможе продовжувати роботу, доки не буде створено новий токен. Також продуктивність алгоритму погіршується, коли кількість вузлів велика, оскільки кожен учасник повинен дочекатися проходження токена через весь цикл кільця, навіть якщо лише один вузол активно потребує доступу до ресурсу. У деяких випадках це може призводити до затримок і зниження ефективності.

Незважаючи на ці обмеження, алгоритм Token Ring залишається важливою концепцією в теорії розподілених систем. Він лежить в основі цілого ряду протоколів і моделей, які використовуються у мережах, системах керування та спеціалізованих апаратних рішеннях. Його простота, справедливість і природний порядок передачі контролю роблять Token Ring фундаментальним прикладом децентралізованого алгоритму взаємного виключення, який і сьогодні застосовується у певних галузях, що потребують гарантованої послідовності та передбачуваності.

4. Об'єкт перерахування TimeUnit для переривання потоку

Перерахування TimeUnit у Java є спеціалізованим інструментом, який використовується для точного та уніфікованого задання часових інтервалів у багатопотокових програмах. Воно надає програмісту зручний спосіб працювати з одиницями вимірювання часу – від наносекунд до днів – і дозволяє явно визначати, у яких одиницях слід інтерпретувати затримки, тайм-аути або моменти очікування. Цей механізм набуває особливого значення у контексті переривання потоків, де точність і контрольованість часу є критичними для правильної координації роботи між компонентами системи.

TimeUnit фактично виступає прошарком абстракції, що позбавляє необхідності виконувати ручні обчислення або перетворення між різними одиницями часу. Коли потік повинен чекати певний проміжок часу, програміст може використати методи sleep(), await(), tryLock() чи інші операції синхронізації, що приймають параметр TimeUnit. Такий підхід дає змогу чітко визначати тривалість очікування з використанням інтуїтивно зрозумілих одиниць, наприклад milliseconds або seconds, що робить код як точнішим, так і легшим для читання.

У контексті переривання потоку TimeUnit відіграє ключову роль у визначенні того, як саме потік реагує на часові обмеження. Коли потік виконує операцію очікування з тайм-аутом, він блокується лише на визначений час, після чого може автоматично продовжити роботу або перейти до альтернативної логіки. Якщо в цей час потік буде перерваний, методи, що використовують TimeUnit, ініціюють обробку InterruptedException або повертають спеціальні значення, які дозволяють програмі коректно реагувати на зовнішній сигнал переривання. Завдяки цьому потік може завершити очікування без зайвих затримок і перевірити, чи слід припинити виконання або продовжити дію в новому стані.

Цей механізм особливо важливий у складних паралельних системах, де надмірне або неправильне блокування потоків може призвести до тупиків або

зниження продуктивності. Використання TimeUnit дає змогу точно регулювати поведінку потоків і дозволяє створювати алгоритми, що коректно реагують на тайм-аути, перевищення часу очікування або зовнішні запити на завершення. Таким чином, TimeUnit не лише полегшує роботу з часовими інтервалами, але й виступає важливою частиною механізму контролю потоку виконання.

Завдяки своїй універсальності та чіткості TimeUnit забезпечує передбачувану й безпомилкову інтерпретацію часу у багатопотокових програмах. Він дозволяє розробникам організовувати надійні схеми очікування і переривання, роблячи поведінку потоків стабільною, контрольованою й адаптованою до складних сценаріїв синхронізації.

ТЕМА 2.5. ВИКОРИСТАННЯ ОБМІННИКІВ

План

1. Обмін даними між потоками. Клас Exchanger та метод exchange.
2. Процес відправник та процес-отримувач, режим одночасного прийому та передачі даних.
3. Створення інтерфейсного класу для даних, які підлягають обміну та формату їх передачі.

1. Обмін даними між потоками. Клас Exchanger та метод exchange.

Обмін даними між потоками є однією з ключових складових багатопотокового програмування, оскільки саме через узгоджене передавання інформації паралельні задачі можуть взаємодіяти, доповнювати одна одну та формувати узгоджений результат. На відміну від однопотокових програм, де дані завжди перебувають у спільному контексті виконання, у багатопотокових системах кожен потік має свій власний стек і простір локальних змінних. Тому будь-яка взаємодія між потоками потребує доступу до спільних структур даних або спеціальних механізмів, які забезпечують коректність і передбачуваність такої взаємодії.

Основною проблемою при організації обміну даними є ризик одночасного доступу кількох потоків до одного і того ж ресурсу. Якщо два потоки змінюють спільний об’єкт паралельно, не узгоджуючи свої дії, це може призвести до неконсистентного стану, частково записаних значень, втрати інформації або логічних помилок у програмі. Саме тому обмін даними ніколи не зводиться лише до передавання значення – його необхідно поєднувати з механізмами синхронізації, які забезпечують взаємне виключення, упорядкованість доступу та видимість змін між потоками.

У найпростішому випадку обмін даними може відбуватися через спільний об’єкт, синхронізований за допомогою м’ютекса або ключового слова synchronized. У такій моделі кожний потік отримує ексклюзивний доступ до ресурсу, виконує запис або читання й звільняє блокування, що

гарантує атомарність операцій. Проте в більш складних сценаріях, де потоки повинні координувати не лише зміну даних, а й сам порядок взаємодії, застосовуються більш спеціалізовані механізми – черги блокування, умови очікування, семафори або засувки. Такі конструкції дозволяють створювати природні схеми виробник–споживач, односпрямовані канали або циклічні буфери, де дані не лише передаються, а й обробляються у визначеній послідовності.

Важливим аспектом обміну даними є забезпечення видимості змін між потоками. Навіть якщо два потоки послідовно виконують операції над одним і тим самим ресурсом, це не гарантує, що зміни, внесені одним потоком, відразу стануть доступними іншому. Кешування даних у локальних буферах процесора або оптимізації JVM можуть призвести до того, що другий потік працюватиме зі старою версією об’єкта. Використання volatile, пам’яттєвих бар’єрів або синхронізованих методів забезпечує правильну модель видимості, у якій усі оновлення ресурсів стають глобально доступними одразу після виконання операції.

У сучасній Java обмін даними часто реалізується через високорівневі структури з пакету java.util.concurrent. Такі конструкції, як BlockingQueue, ConcurrentMap, Exchanger або SynchronousQueue, забезпечують готові й безпечні засоби комунікації між потоками. Вони інкапсулюють усю складність синхронізації та надають розробникові простий інтерфейс, який дозволяє організувати передавання даних без ручного керування блокуваннями. Завдяки цьому значно зменшується ризик тупиків, гонок або неправильних послідовностей, а код стає стабільним і масштабованим.

У підсумку обмін даними між потоками є не просто технічною процедурою передавання інформації, а комплексним процесом, який вимагає дотримання правил узгодженості, видимості та синхронізації. Правильно організований обмін дозволяє потокам працювати спільно, формувати паралельні конвеєри обробки, підвищувати продуктивність програми та підтримувати логічну цілісність усієї системи. Безпечна передача даних є

наріжним каменем багатопотокового програмування, від якого залежить стабільність, ефективність і передбачуваність взаємодії між потоками.

Клас Exchanger у Java є спеціалізованим інструментом для організації безпосереднього та синхронізованого обміну даними між двома потоками. На відміну від інших механізмів комунікації, які передбачають посередників у вигляді черг, буферів або спільних змінних, Exchanger працює як двосторонній канал, у якому кожний із двох потоків передає свій об’єкт і водночас отримує об’єкт від партнера. Такий підхід створює природну модель взаємодії, де обмін відбувається лише тоді, коли обидві сторони до нього готові, що робить Exchanger ефективним засобом синхронізації у задачах, де дані рухаються попарно.

Принцип роботи Exchanger ґрунтується на ідеї зустрічної точки. Потік, який викликає метод exchange(), передає в нього об’єкт і блокується, очікуючи, доки другий потік не виконає ту саму дію зі свого боку. У момент, коли обидва потоки досягають цієї точки, об’єкти, передані ними, обмінюються місцями: перший потік отримує об’єкт другого, а другий – об’єкт першого. Якщо ж другий потік затримується, перший продовжує перебувати в режимі очікування, доки не спрацює тайм-аут або не буде надіслано сигнал переривання. Така модель забезпечує ідеально узгоджену координацію, де обмін не може відбутися частково або асиметрично – обидві сторони повинні бути одночасно готовими до синхронізації.

Метод exchange() надає можливість не лише передавати об’єкти, але й формувати структурні точки обміну, які стають частиною більшого алгоритмічного ланцюга. Наприклад, у задачах обробки даних два потоки можуть взаємодіяти у режимі “завантажувач–споживач”, де один генерує нову порцію даних, а інший надсилає назад результати обробки попередньої порції. Застосування Exchanger дозволяє уникнути надмірного блокування або створення додаткових структур даних, оскільки обмін відбувається безпосередньо і в строго визначені моменти часу.

Важливою особливістю Exchanger є те, що він органічно поєднує синхронізацію та обмін інформацією в єдину операцію. Це робить його надзвичайно корисним у сценаріях, де два потоки виконують паралельні частини однієї задачі та мають передавати дані один одному у певному ритмі. Наприклад, у паралельних алгоритмах, які працюють фазами або тактами, Exchanger дозволяє потокам синхронізувати завершення фази і водночас передати результати її виконання. Завдяки цьому механізм стає основою циклічного обміну, де кожна ітерація алгоритму супроводжується взаємною передачею даних.

Загалом Exchanger виступає простим, але потужним засобом організації двостороннього зв'язку між потоками, який забезпечує повну атомарність операції обміну, блокуюче очікування та автоматичну узгодженість дій. Він відкриває можливості для створення елегантних схем взаємодії, де потоки рухаються у спільному темпі та обмінюються даними точно в той момент, коли це необхідно для продовження обчислювального процесу.

2. Процес відправник та процес-отримувач, режим одночасного прийому та передачі даних.

У розподілених і багатопотокових системах взаємодія між процесами часто організовується через модель «відправник – отримувач», яка визначає чіткий напрямок руху даних і відповідальність кожної сторони. Процес-відправник є джерелом інформації: він формує повідомлення, структурує його у відповідному форматі й ініціює передачу. Процес-отримувач, у свою чергу, виступає кінцевою точкою, яка приймає передані дані, обробляє їх і може передавати результати далі або відповідати зустрічним сигналом. Така модель створює логічну комунікаційну пару, яка забезпечує впорядкованість обміну та можливість узгодженого виконання складних операцій.

У межах цієї взаємодії важливо розуміти, що передавання даних ніколи не є миттєвим або гарантованим. Відправник може надсилати повідомлення асинхронно, не очікуючи безпосередньої відповіді, або синхронно, коли він блокується до моменту отримання підтвердження. У синхронній моделі взаємозалежність процесів вища: відправник не може продовжити роботу, доки отримувач не підтвердить успішне прийняття даних. У той час як асинхронна модель дозволяє процесам рухатися незалежно, але потребує механізмів кешування, чергування або буферизації, щоб гарантувати, що дані не будуть втрачені в момент передачі.

Процес-отримувач завжди працює в умовах можливих затримок, неповної інформації й непередбачуваного часу надходження даних. Тому він повинен мати механізми обробки запитів, чергування або блокування на очікуванні, що дозволяє йому зберігати стабільність навіть тоді, коли потік повідомлень є нерівномірним. У багатопотокових системах це може бути реалізовано через блокуючі черги, семафори або засоби низькорівневої синхронізації. У розподілених мережевих системах функцію очікування виконують протоколи транспортного рівня або механізми перевірки коректності доставки.

Ключовим елементом взаємодії між відправником і отримувачем є узгодженість даних. Навіть якщо механізм передачі працює коректно, система повинна гарантувати, що отримувач бачить саме ті значення, які були передані, та у тому порядку, який є важливим для логіки застосування. Через це між процесами вводяться спеціальні протоколи підтвердження, маркери часу або індикатори стану, що дозволяють відправнику переконатися в тому, що його повідомлення було прийняте, а отримувачу – що воно належить до правильної частини робочої послідовності.

У підсумку модель «процес-відправник – процес-отримувач» є базовим шаблоном, який дозволяє структурувати взаємодію між незалежними обчислювальними компонентами. Вона створює передбачуваний канал комунікації, у якому кожен із процесів відіграє чітко окреслену роль,

забезпечуючи узгоджене передавання даних, контрольовану обробку та можливість масштабування складних паралельних або розподілених систем. Саме завдяки такій моделі вдається побудувати стабільні механізми обмену повідомленнями, що лежать в основі сучасних мережевих протоколів, паралельних алгоритмів і багатопотокових архітектур.

Режим одночасного прийому та передачі даних є важливою моделлю взаємодії в багатопотокових і розподілених системах, де процеси або потоки виконують обмін інформацією паралельно, не очікуючи завершення дій один одного. Такий режим дозволяє досягти більшої пропускної здатності, зменшити затримки та забезпечити плавний, безперервний рух даних у системах зі складною або інтенсивною комунікацією. Його сутність полягає у тому, що потік може виконувати операції надходження даних та їх передачі фактично незалежно, у той час як системні механізми синхронізації підтримують узгодженість і коректність цих операцій.

У середовищах зі значним мережевим або обчислювальним навантаженням режим одночасного прийому й передачі є критичним для досягнення високої ефективності. Коли процес отримує дані від одного джерела, він водночас може відправляти результати обробки іншому учасникові мережі, формуючи своєрідний конвеєр обчислень. Така модель усуває необхідність очікувати завершення одного етапу перед переходом до наступного, що робить взаємодію безперервною та ритмічною. На практиці це дозволяє значно скоротити час простою й забезпечити постійний рух інформаційних потоків.

Важливою особливістю цього режиму є те, що одночасність не означає хаотичності або відсутності структури. Навпаки, система повинна гарантувати, що потоки, які отримують та передають дані паралельно, оперують узгодженими станами і не заважають один одному. У Java така модель часто реалізується через окремі канали або буфери, що працюють незалежно: один обробляє вхідні дані, інший – вихідні. Потоки можуть використовувати неблокуючі структури, локальні черги або механізми з

лінійною синхронізацією, що дозволяє продовжувати роботу без зайвих затримок у разі тимчасової відсутності даних або коливань швидкості передавання.

У мережевих системах цей режим відповідає принципам повнодуплексного зв'язку, де канал дозволяє передавати й приймати інформацію одночасно, не розділяючи операції у часі. Таке рішення забезпечує більш плавний обмін повідомленнями, особливо в системах, де затримки можуть впливати на логіку обробки або точність часових залежностей. Паралельність дій дозволяє уникати проблем, пов'язаних із чергуванням, наприклад накопиченням неприйнятих пакетів або необхідністю відкладати обробку результатів через зайнятість каналу.

Одночасний прийом і передача даних створюють умови для реалізації більш складних алгоритмів паралельної обробки, коли система може розділяти інформаційні потоки залежно від їх призначення, пріоритетності або інтенсивності. У багатопотокових програмах це дозволяє створювати гнучкі архітектури, у яких кожний потік виконує свою роль, але водночас не блокує роботу інших компонентів. У результаті така організація забезпечує більш високу продуктивність, стабільність і передбачуваність навіть у складних або непередбачуваних робочих умовах.

Загалом режим одночасного прийому та передачі даних є фундаментальною моделлю для побудови ефективних систем паралельної взаємодії. Він забезпечує рівновагу між швидкістю, узгодженістю та адаптивністю, дозволяючи потокам виконувати свої функції незалежно, але у рамках загальної логіки роботи системи. Саме завдяки такому режиму сучасні мережеві й багатопотокові рішення здатні підтримувати стабільний, ритмічний і масштабований рух даних, що є ключовою умовою їх ефективності.

3. Створення інтерфейсного класу для даних, які підлягають обміну та формату їх передачі

У багатопотокових і розподілених системах ключову роль відіграє спосіб, у який організовується структура даних, що передаються між потоками або процесами. Для того щоб забезпечити передбачуваність, узгодженість і гнучкість у комунікації, дані часто описуються через спеціальні інтерфейсні класи, які визначають загальний контракт взаємодії. Інтерфейс задає те, якими властивостями повинні володіти об’єкти, що обмінюються між потоками, які методи вони повинні реалізовувати та в якому вигляді вони можуть бути передані або прийняті. Такий підхід дозволяє уніфікувати процес комунікації, уникнути залежності від конкретних реалізацій і забезпечити можливість легко замінювати або розширювати формати даних у майбутньому.

Створення інтерфейсного класу фактично задає абстрактний опис даних, незалежний від того, як саме ці дані будуть збережені у пам’яті або оброблені потоком. Потоки, які взаємодіють між собою, працюють не з конкретними типами, а з їхнім узагальненим представленням, що значно спрощує архітектуру всієї системи. Це дозволяє підключати різні реалізації інтерфейсу – від простих текстових повідомлень до складних об’єктів, що містять метадані, час мітки або структуровані набори значень. Завдяки цьому механізму обмін даними стає передбачуваним, а програміст має змогу керувати їхнім життєвим циклом, не змінюючи логіку потоків.

Окреме значення має питання формату передавання даних. Формат визначає, у якому вигляді об’єкт передається між потоками – як серіалізований байтовий потік, як JSON-структура, як внутрішній об’єкт JVM або як спеціально підготовлений контейнер. У межах однієї програми, де потоки працюють у спільній пам’яті, формат може бути максимально простим: достатньо внутрішнього об’єкта, який реалізує інтерфейс. У розподіленому середовищі формат стає критично важливим, оскільки дані мають бути коректно перетворені на структуру, яка може бути передана мережею, та відновлені на іншому боці. Саме інтерфейс задає правила

серіалізації та десеріалізації, що дозволяє досягти сумісності між різними учасниками обміну.

Інтерфейс також визначає межі відповідальності. Він описує, які операції дозволені над об’єктом, які атрибути є обов’язковими, у якому вигляді дані можуть бути змінені та які гарантії має забезпечити реалізація. Це особливо важливо у багатопотокових сценаріях, де різні потоки можуть одночасно працювати з отриманою структурою. Стандартизований інтерфейс дозволяє чітко описати очікувану поведінку даних, уникнути помилок, пов’язаних із невірною інтерпретацією, та забезпечити сумісність між різними частинами програми.

У підсумку створення інтерфейсного класу для даних, що підлягають обміну, є фундаментальним елементом побудови надійних систем міжпотокової взаємодії. Інтерфейс формує єдину точку абстракції, через яку визначаються властивості, формат і логіка передавання даних. Завдяки цьому система отримує структуровану й передбачувану модель комунікації, яка дозволяє масштабувати, розширювати й модифікувати механізми обміну без ризику втрати сумісності або виникнення помилок у взаємодії між потоками.

ЗМІСТОВНИЙ МОДУЛЬ 3
РОЗПОДІЛЕНЕ ПРОГРАМУВАННЯ НА МОВІ JAVA

ТЕМА 3.1. РОЗПОДІЛЕНЕ ПРОГРАМУВАННЯ З ВИКОРИСТАННЯМ ПОРТФЕЛЮ ЗАДАЧ

План

1. Парадигма портфелю задач: принципи взаємодії та синхронізації процесів при використанні портфеля задач.
2. Особливості програмування та застосування портфелю задач.
3. Приклад використання портфеля задач за допомогою ExecutorService та виклику Executors.newFixedThreadPool.

1. Парадигма портфелю задач: принципи взаємодії та синхронізації процесів при використанні портфеля задач

Парадигма портфеля задач є одним із найефективніших підходів до організації паралельних обчислень, коли велика робота розбивається на множину незалежних або частково пов’язаних підзадач, які регулюються спільним пулом виконавців. На відміну від класичних моделей, де кожен потік закріплюється за конкретною частиною алгоритму, портфель задач передбачає динамічний розподіл робіт: усі доступні задачі поміщаються у загальну структуру, з якої потоки-виконавці самостійно вибирають наступну задачу. Це створює гнучку, адаптивну модель взаємодії, у якій навантаження між потоками вирівнюється автоматично, а пам’ять і ресурси використовуються більш раціонально.

Сутність парадигми полягає в тому, що потоки не прив’язані до конкретної задачі або ролі, а працюють як універсальні «робітники». Вони взаємодіють не один із одним, а зі спільним портфелем задач – своєрідною чергою або набором робочих одиниць. Це мінімізує прямі залежності між потоками й усуває необхідність у складних узгодженнях. Комунікація відбувається опосередковано: додавання задачі до портфеля сигналізує про

необхідність її виконання, а вилучення задачи означає, що інший потік уже виконує цю частину роботи. Завдяки цьому потоки не блокують одне одного без потреби, а виконання відбувається у режимі постійного “самообслуговування”.

Важливим аспектом парадигми портфеля задач є синхронізація доступу до спільної структури, яка містить задачі. Оскільки кілька потоків можуть одночасно намагатися додати або взяти задачу, система повинна забезпечити атомарність цих операцій. У традиційних реалізаціях для цього застосовуються механізми взаємного виключення, блокуючі черги або спеціалізовані структури даних, такі як ConcurrentLinkedQueue чи BlockingQueue. Вони дозволяють організувати безпечний доступ до портфеля, гарантують коректний порядок обробки задач і запобігають можливим конфліктам або втратам даних.

У процесі виконання задачі можуть створювати нові підзадачі або змінювати структуру портфеля, що робить модель ще більш динамічною. Потоки працюють доти, поки у портфелі існують задачі, і припиняють роботу лише тоді, коли всі задачі вичерпані й жоден із потоків не створює нові. Цей механізм вимагає особливої уваги до синхронізації завершення роботи: система повинна правильно визначити момент, коли портфель порожній і більше не буде змінюватися. Зазвичай це реалізується через лічильники активних задач або спеціальні умови, які сигналізують про завершення формування нового набору робіт.

Парадигма портфеля задач демонструє свою особливу ефективність у нерівномірно розподілених обчисленнях, де окремі задачі можуть мати суттєво різну складність. Завдяки динамічному розподілу роботи потоки не простоюють, очікуючи завершення «важких» задач інших потоків. Замість цього кожен виконавець негайно бере наступну доступну задачу, що дозволяє використовувати ресурси процесора максимально повно. Такий підхід дозволяє істотно зменшити час виконання складних обчислень і підвищити масштабованість програми.

Синхронізація процесів у парадигмі портфеля задач ґрунтується на принципі узгодженого доступу до спільного набору робіт, який виступає центральною точкою координації між усіма потоками-виконавцями. У цій моделі потоки не взаємодіють безпосередньо один з одним, а комунікують опосередковано через портфель задач, тому коректність роботи системи визначається тим, наскільки узгоджено здійснюється доступ до цього спільного об’єкта. Система повинна забезпечити, щоб у будь-який момент часу задачі не втрачалися, не дублювалися і не виконувалися одночасно кількома потоками, що потребує надійних засобів синхронізації.

Основою такої синхронізації є атомарні операції додавання та вилучення задач із портфеля. Коли потік хоче взяти задачу, він повинен отримати ексклюзивний доступ до структури, у якій сформований список або черга задач. У разі, коли інший потік вже виконує аналогічну операцію, решта потоків чекають або перемикаються на інші роботи. Синхронізація тут не блокує виконання всього застосування, оскільки кожний потік працює незалежно, але гарантує, що результати взаємодії з портфелем залишаються послідовними та логічно узгодженими. Завдяки цьому виключається можливість виникнення гонок, унаслідок яких кілька потоків могли б виконувати одну і ту саму задачу.

Коли задачі виконуються нерівномірно, синхронізація відіграє ключову роль у тому, щоб забезпечити динамічний перерозподіл навантаження. Потоки, які завершили свою роботу раніше, негайно отримують доступ до портфеля й забирають наступну задачу. У той час як потоки, що ще виконують тривалі задачі, не заважають іншим і не потребують додаткової координації. Цей механізм природно усуває простої і формує рівномірний розподіл навантаження між виконавцями. Синхронізація тут не зводиться лише до блокування – вона визначає ритм взаємодії потоків із портфелем, запобігаючи перевантаженню структури та підтримуючи оптимальний баланс.

Завершення роботи системи також потребує точного синхронізаційного механізму. Оскільки задачі можуть створюватися динамічно – під час виконання інших задач – важливо правильно визначити момент, коли портфель остаточно спорожнів. Потоки повинні завершити роботу лише тоді, коли не тільки не залишилось задач у портфелі, але й не існує жодного потоку, що активно створює нові задачі. Для цього застосовуються лічильники активних задач або спеціалізовані сигнальні механізми, які дають можливість визначити стан завершення обчислювального процесу.

У результаті синхронізація в парадигмі портфеля задач формує стабільний і збалансований підхід до керування паралельним виконанням. Вона створює передбачуваний порядок доступу до задач, забезпечує рівномірний розподіл роботи та запобігає конфліктам, зберігаючи при цьому динамічність і масштабованість системи. Саме завдяки такій моделі синхронізації портфель задач залишається однією з найефективніших парадигм у сучасних системах паралельних обчислень.

У підсумку портфель задач створює адаптивну, збалансовану й масштабовану структуру паралельних обчислень, де задачі виконуються потоками незалежно один від одного, але в межах спільного узгодженого механізму. Взаємодія між процесами відбувається непрямо, через портфель, що зменшує складність синхронізації та запобігає виникненню тупиків або гонок даних. Саме завдяки такій архітектурі портфель задач широко застосовується у сучасних фреймворках, таких як ForkJoinPool, ExecutorService та різноманітні системи паралельних обчислень.

2. Особливості програмування та застосування портфелю задач

Парадигма портфелю задач у контексті Java набуває особливої виразності завдяки розвинутим засобам бібліотеки java.util.concurrent, яка надає цілісну інфраструктуру для організації пулів потоків, управління задачами та синхронізації між ними. У цій моделі програміст працює не з окремими потоками, а з абстракцією задачі – об’єктом, що реалізує інтерфейси Runnable або Callable, та передається до портфелю для

подальшого виконання одним із доступних робочих потоків. Таким чином Java створює природні умови для реалізації портфельного підходу, де задачі є центральними компонентами, а їх виконання – результатом взаємодії між системою та пулом виконавців.

Особливістю програмування портфелю задач у Java є те, що управління потоками приховане всередині інфраструктури, а програміст фактично формує лише логічний опис роботи. Створення пулу потоків через ExecutorService дозволяє передати відповідальність за планування, створення й перерозподіл виконання системі, яка адаптивно розподіляє задачі між доступними потоками. Це не лише мінімізує ризики неправильного керування потоками, а й гарантує, що програма зможе масштабуватися відповідно до апаратних можливостей – наприклад, автоматично використовувати більшу кількість ядер процесора без зміни логіки обчислень.

Важливим аспектом є спосіб розбиття роботи на задачі. У Java задачі зазвичай мають бути компактними, самодостатніми й вільними від довгих блокувань, оскільки виконавці пулу не очікують завершення операцій, що можуть затримати інші задачі. Надмірно великі задачі можуть призвести до нерівномірного використання потоків, тоді як надто дрібні – до значних накладних витрат на їх розподіл. Java надає гнучкі інструменти, такі як ForkJoinPool, які автоматично оптимізують розміри задач через механізм work-stealing, що дозволяє досягати високої продуктивності під час виконання глибоко розподілених або рекурсивних алгоритмів.

На етапі виконання задач Java приділяє велику увагу синхронізації та узгодженості результатів. Оскільки задачі можуть створювати нові підзадачі або оновлювати спільні ресурси, важливо правильно організувати комунікацію між ними, використовуючи блокуючі структури, атомарні змінні, семафори чи об’єкти синхронізації. Особливо поширеним у портфельних підходах є об’єднання результатів через механізм Future, який дозволяє спостерігати за завершенням задачі та отримувати результат лише

тоді, коли він готовий. Це створює природний спосіб комбінувати паралельні обчислення без необхідності блокування потоків на весь період виконання.

Окрему увагу в застосуванні портфелю задач на Java приділено завершенню системи. Пул потоків може функціонувати безперервно, приймаючи нові задачі, або ж існувати в межах певної обчислювальної сесії. У таких випадках програміст має враховувати, що завершення роботи через shutdown() або shutdownNow() не повинно відбуватися до моменту, коли всі задачі виконано або завершено контрольовано. Це гарантує правильність обчислень, узгодженість ресурсів і запобігає ситуаціям, коли задачі обриваються у проміжному стані.

Таким чином, особливості програмування портфелю задач у Java визначаються поєднанням високорівневої абстракції задач із гнучкими і ефективними засобами керування потоками. Java створює середовище, де паралельні обчислення стають логічно прозорими, а розподіл роботи – адаптивним і оптимізованим. Завдяки цьому парадигма портфелю задач перетворюється на універсальний інструмент для розробки масштабованих, надійних і високопродуктивних програмних систем, незалежно від складності і структури прикладних задач.

3. Приклад використання портфеля задач за допомогою ExecutorService та виклику Executors.newFixedThreadPool

Практичне застосування портфеля задач у Java найчастіше реалізується через інфраструктуру ExecutorService, яка дозволяє створити пул потоків і організувати обробку задач у режимі керованої паралельності. За допомогою методу Executors.newFixedThreadPool формується фіксований набір робочих потоків, кожен із яких виступає виконавцем задач зі спільного портфеля. У цій моделі задачі існують як автономні одиниці роботи, а потоки не закріплені ні за якою конкретною задачею, а виконують їх за мірою доступності, забезпечуючи природний і збалансований розподіл навантаження.

Уявімо ситуацію, коли програма повинна опрацювати велику кількість незалежних задач – наприклад, обробити набір даних, виконати серію розрахунків або здійснити кілька паралельних запитів. У традиційному підході довелося б створювати новий потік для кожної задачі, що призводить до суттєвих накладних витрат і сповільнює роботу програми. Використання ж портфеля задач з фіксованим пулом потоків дозволяє позбутися цього недоліку: задачі надходять у спільну чергу, а робочі потоки по черзі їх виконують, не створюючи додаткових об’єктів потоків. Це робить процес значно ефективнішим і контрольованішим.

У типовому сценарії програміст створює пул за допомогою виклику Executors.newFixedThreadPool(N), де N визначає кількість потоків, доступних для виконання задач. Після цього задачі подаються до ExecutorService у вигляді об’єктів, що реалізують Runnable або Callable, і система автоматично включає їх до загального портфеля. Потоки пулу постійно перевіряють наявність нових задач і виконують їх у міру надходження. Якщо задач більше, ніж доступних потоків, вони накопичуються в черзі, чекаючи своєї черги, що дозволяє виконанню відбуватися без перевантаження системи.

Під час виконання задач ExecutorService гарантує узгодженість роботи процесів і правильність завершення обчислювального циклу. Завдяки цій моделі програмісту не потрібно стежити за створенням, запуском або знищенням потоків – усі ці операції приховані всередині виконувального механізму. Після того як усі задачі передано, застосування може ініціювати впорядковане завершення роботи через виклик shutdown(), який дозволяє потоку-виконавцю завершити всі поточні задачі перед зупинкою. У такий спосіб програма завершує роботу передбачувано й без втрати результатів.

Цей підхід демонструє, що портфель задач у Java не є лише концептуальною моделлю, а являє собою робочу архітектуру, яка дозволяє ефективно керувати паралельними обчисленнями. Завдяки використанню ExecutorService та фіксованого пулу потоків задачі виконуються послідовно і впорядковано, при цьому ресурси системи залишаються оптимально

завантаженими. Це робить модель придатною для широкого спектра застосувань – від паралельної обробки даних до серверних рішень, де необхідно контролювати велику кількість однорідних операцій.

ТЕМА 3.2. РОЗПОДІЛЕНЕ ПРОГРАМУВАННЯ НА З ВИКОРИСТАННЯМ ГРАФА «ОПЕРАЦІЇ-ОПЕРАНДИ»

План

1. Модель обчислень у вигляді графа «операції-операнди».
2. Ациклічний орієнтований граф.
3. Граф алгоритму обчислення площі прямокутника.
4. Приклад паралельного алгоритму для обчислення площі прямокутника. ExecutorService та метод execute.

1. Модель обчислень у вигляді графа «операції-операнди»

Модель обчислень у вигляді графа «операції–операнди» є концептуальною основою для подання алгоритмів у паралельних і розподілених обчисленнях. У такій моделі весь процес обчислення розглядається як взаємодія двох ключових типів сутностей: операцій, які виконуються над даними, та операндів, що представляють самі дані або проміжні результати. Графова форма подання дозволяє наочно описати залежності між частинами обчислення, визначити можливості для паралелізації та забезпечити точне розуміння порядку виконання.

У межах графа вузли поділяються на ті, що відповідають за виконання конкретних операцій, і ті, що містять значення, потрібні для цих операцій. Зв’язки між вузлами мають напрямок і відображають потік даних або логічну послідовність обчислень. Коли операції залежать від кількох операндів, граф чітко вказує, які результати повинні бути доступні до початку виконання. Такий підхід дозволяє легко виявити незалежні підграфи, які можуть оброблятися паралельно, а також визначити точки синхронізації, де подальші обчислення можливі лише після завершення всіх попередніх.

Однією з ключових переваг графової моделі є її здатність відображати структуру обчислювального процесу без прив’язки до конкретної архітектури або способу виконання. Вона дає змогу декомпозувати складні задачі на елементарні операції та розмістити їх у просторі залежностей таким

чином, щоб визначити оптимальну послідовність виконання. У системах із паралельним виконанням граф виконує роль дорожньої карти, за якою диспетчер потоків визначає, які частини обчислень можуть бути запущені одночасно, а які повинні чекати на завершення залежних задач.

Граф «операції–операнди» формує природну основу для планування обчислень у різноманітних архітектурах – від SIMD-моделей до розгалужених конвеєрних структур. Він дозволяє аналізувати, які операції можуть бути переносені в інші гілки, які дані можуть бути обчислені заздалегідь, а де потрібно забезпечити сувору послідовність виконання. Такий підхід не лише підвищує продуктивність, а й забезпечує можливість адаптації алгоритмів під конкретні апаратні особливості, у тому числі багатоядерні процесори, кластери або графічні прискорювачі.

Крім того, графова модель дозволяє описувати розподілені обчислення, де різні частини графа можуть виконуватися на різних вузлах мережі. Завдяки явному поданню залежностей система може гарантувати, що дані передаються між вузлами лише тоді, коли це дійсно необхідно, що зменшує комунікаційні витрати і підвищує ефективність виконання. Навіть у випадках динамічних обчислень, коли структура задачі змінюється під час виконання, граф залишається гнучким інструментом, здатним відобразити зміну зв’язків між операціями.

У підсумку модель «операції–операнди» є універсальним способом подання обчислень, який робить структуру алгоритму прозорою та керованою. Вона дозволяє відокремити логіку обчислення від способу його реалізації, ефективно визначає можливості паралельності та забезпечує природну основу для оптимізації. Саме тому графові моделі широко застосовуються в компіляторах, системах паралельного програмування та багатьох сучасних фреймворках для високопродуктивних обчислень.

2. Ациклічний орієнтований граф

Ациклічний орієнтований граф, або DAG (Directed Acyclic Graph), є фундаментальною структурою, що широко використовується у паралельних,

розподілених та обчислювальних моделях. Його ключовою характеристикою є наявність напрямлених ребер між вершинами та відсутність циклів, що означає, що граф ніколи не може повернутися до вже пройденої вершини. У межах моделювання обчислювальних процесів DAG відіграє роль природного інструмента, який задає суворий порядок виконання операцій, відображає залежності між ними та визначає ті частини задачі, які можна виконувати паралельно.

Ациклічність гарантує, що між операціями існує чітка часткова впорядкованість. Жодна операція не може опиратися на результат самої себе через ланцюг залежностей, і це усуває можливість логічних тупиків, рекурсивних визначень або нескінченних циклів у процесі виконання. Такий порядок забезпечує однозначність обчислень: будь-який вузол DAG може виконуватися лише після того, як обчислені всі його попередники. Саме тому DAG є одним із найпоширеніших способів опису обчислювальних схем у компіляторах, планувальниках задач, системах обробки даних і розподілених середовищах.

Структура DAG дозволяє наочно й точно описати залежності між операціями. Якщо одна вершина спрямована у бік іншої, це означає, що друга операція потребує результату першої. Завдяки цьому можна легко визначити «критичний шлях» – послідовність операцій, що визначає мінімальний час виконання всього алгоритму. Ті вершини, які не входять до критичного шляху й не залежать одна від одної, можуть виконуватися паралельно, що дає можливість максимально використати ресурси багатоядерних або розподілених систем. DAG у цьому сенсі стає не лише формальною структурою, а й інструментом планування роботи.

Ациклічний орієнтований граф природно відповідає концепції потоку даних. Коли операції генерують результати, які передаються іншим операціям, DAG задає напрям руху цих даних і визначає умови та порядок їхнього використання. Завдяки цьому система може точно розуміти, які елементи обчислення готові до виконання, а які повинні чекати на

формування проміжних результатів. У складних обчислювальних моделях DAG використовується як основа для автоматичних планувальників, що визначають оптимальний порядок виконання задач на основі аналізу залежностей.

DAG також забезпечує прозорість і керованість розподілених обчислень. Коли окремі частини графа призначаються різним вузлам мережі, ациклічність гарантує, що обробка даних не потребує повторного проходження по колу або не призведе до взаємних блокувань. Це значно спрощує передавання результатів між вузлами, мінімізує комунікаційні витрати й робить поведінку системи прогнозованою. У середовищах, де алгоритм може генерувати нові підзадачі під час виконання, DAG дозволяє легко вставляти нові вузли без порушення загальної структури залежностей.

У підсумку ациклічний орієнтований граф є базовою моделлю подання обчислень, яка забезпечує логічну строгість, структурну прозорість і можливість паралельного виконання. Він дозволяє програмістам, компіляторам і системам планування визначити оптимальний порядок виконання, гарантувати узгодженість результатів і уникнути логічних помилок, пов’язаних із циклічними залежностями. Саме завдяки цим властивостям DAG став одним із ключових концептів сучасних моделей паралельних і розподілених обчислень.

3. Граф алгоритму обчислення площі прямокутника

Граф алгоритму обчислення площі прямокутника є простим, але водночас показовим прикладом використання моделі «операції–операнди» та побудови обчислювального процесу у вигляді ациклічного орієнтованого графа. Така форма подання дозволяє наочно відобразити залежності між вихідними даними, проміжними діями та кінцевим результатом, підкреслити логіку виконання та визначити можливості для паралельності навіть у найпростіших обчисленнях. Завдяки цьому граф виступає універсальною формою опису алгоритму, відокремлюючи суть обчислень від їхньої імперативної реалізації.

У задачі обчислення площі прямокутника вихідними операндами є значення довжини та ширини. У графі вони представлені у вигляді вузлів-даних, які не залежать один від одного, а отже можуть бути доступні одночасно. Алгоритм передбачає виконання єдиної ключової операції – множення цих двох величин, результатом якої є шукане значення площі. У графі ця операція відображається окремим вузлом-операцією, який отримує два вхідні ребра від вузлів операндів. Оскільки множення не потребує додаткових проміжних результатів, граф залишається мінімальним і має чітко визначений напрямок виконання: від отримання вхідних значень до формування кінцевого результату площі.

Структура графа відображає залежності між його елементами. Операція множення може бути виконана лише тоді, коли обидві вхідні величини – довжина і ширина – вже доступні. Це означає, що граф задає строго визначену точку синхронізації: навіть за наявності можливостей для паралельного виконання операцій підготовки даних, саме обчислення площі запускається лише за умови готовності обох операндів. Така властивість графа є важливою ознакою ациклічних орієнтованих структур, оскільки забезпечує логічну відповідність обчислювальному процесу та унеможливлює передчасне або некоректне виконання операції.

У більш загальному сенсі граф алгоритму обчислення площі прямокутника є прикладом того, як навіть найпростіші операції можуть бути структуровано описані через графову модель. Це особливо важливо у паралельних і розподілених обчисленнях, де алгоритм може містити сотні або тисячі операцій, але логіка їхнього виконання підкоряється тим самим принципам залежностей і впорядкованості. Граф дозволяє чітко бачити, які операції можна виконати незалежно, а які є критичними й вимагають попереднього отримання результатів інших дій. У результаті його використання забезпечує прозорість алгоритму та спрощує аналіз можливостей для оптимізації.

Граф обчислення площі прямокутника може розглядатися й як основа для складніших моделей, де множення входить до більшого обчислювального ланцюга. У такому разі простий граф розширюється додатковими вузлами, утворюючи частину більш складної обчислювальної схеми. Саме так формується загальний підхід до побудови обчислювальних DAG-структур, де кожен елементарний крок – включно з таким простим, як множення – стає частиною більшого процесу.

Отже, граф алгоритму обчислення площі прямокутника є ілюстративним прикладом того, як обчислення можуть бути представлені у вигляді структурованої і візуально зрозумілої моделі. Він демонструє важливість залежностей між операціями, відсутність циклів та можливість паралельного виконання вихідних дій. Незважаючи на свою простоту, цей граф є важливим кроком у розумінні того, як складні обчислювальні алгоритми організовуються, оптимізуються та реалізуються у вигляді ациклічних орієнтованих структур.

4. Приклад паралельного алгоритму для обчислення площі прямокутника. ExecutorService та метод execute

Хоча задача обчислення площі прямокутника є елементарною й складається з однієї операції множення, вона може слугувати наочним прикладом для демонстрації принципів паралельної організації обчислень у Java. У контексті багатопотокового програмування важливо не стільки саме числове обчислення, скільки спосіб структуризації роботи, спосіб передавання параметрів у потоки та організація взаємодії між ними. Використання ExecutorService у поєднанні з методом execute() дозволяє перенести навіть просту операцію в паралельне середовище, показавши, як задачі можуть бути подані до пулу потоків і виконані незалежно від основного процесу.

У паралельній реалізації обчислення площі прямокутника логіка алгоритму поділяється на дві частини. Перша частина пов’язана з підготовкою початкових даних – значень довжини та ширини. Друга частина

– це власне виконання операції множення, яке передається як задача до пулу потоків. Ключовим моментом є те, що основний потік не виконує множення безпосередньо, а формує окрему задачу у вигляді об’єкта, що реалізує інтерфейс Runnable, і делегує обчислення одному з доступних виконавців у пулі. Таким чином алгоритм набуває паралельної форми: обчислення площі відбувається в іншому потоці, а основна програма може продовжувати роботу або чекати на результат залежно від задачі.

При створенні пулу потоків за допомогою Executors.newFixedThreadPool() визначається кількість виконавців, які можуть обробляти задачі одночасно. Передача задачі через метод execute() не повертає результату, що робить модель асинхронною: задача починає виконуватися у фоновому режимі, а механізм синхронізації або обробки результатів покладається на програміста. У випадку обчислення площі це може означати запис отриманого значення у спільний об’єкт або виведення його безпосередньо всередині робочого потоку. Важливо, що використання execute() робить процес максимально легким і надає простий механізм передачі задачі до портфеля без потреби відстежувати її завершення через структуру Future.

Паралельна форма виконання розкриває додаткові можливості навіть у такій простій задачі. Наприклад, можна підготувати кілька наборів даних – площі різних прямокутників – і подати їх до пулу потоків у вигляді окремих незалежних задач. Пул забезпечить автоматичний розподіл цих задач, дозволяючи виконувати множення одночасно в кількох потоках. Це робить алгоритм масштабованим: при збільшенні числа доступних ядер процесора швидкість обчислення великої серії площ суттєво зростає. Водночас основний потік не бере участі в обчисленнях і може займатися іншими частинами програми.

Окреме значення має впорядкованість завершення роботи. Після передачі задач у пул необхідно коректно завершити роботу ExecutorService, використовуючи метод shutdown(), який дозволяє завершити обробку всіх

виконуваних задач без примусового переривання. Це забезпечує передбачуваність паралельного алгоритму і гарантує, що обчислення площі завершиться навіть тоді, коли ініціюючий потік вже не бере участі в процесі.

У підсумку паралельний приклад обчислення площі прямокутника демонструє, що навіть елементарні математичні операції можуть бути інтегровані в модель портфеля задач і виконані в контексті багатопотокової архітектури. Він підкреслює важливість правильного формування задачі, механізму її передачі до пулу потоків та організації завершення роботи. Завдяки ExecutorService і методу execute() програма отримує чітку структуру, у якій кожна задача виконується незалежним потоком виконавця, забезпечуючи масштабованість, читабельність та розширюваність обчислювального алгоритму.

ТЕМА 3.3. РОЗПОДІЛЕНЕ ПРОГРАМУВАННЯ З ВИКОРИСТАННЯМ БАР'ЄРНОЇ СИНХРОНІЗАЦІЇ

План

1. Засіб синхронізації бар’єр або точка синхронізації.
2. Принципи взаємодії та синхронізації при використанні бар'єрної синхронізації, особливості програмування та застосування.
3. Клас CyclicBarrier. Приклад використання бар'єрної синхронізації для задачі множення двох матриць.
4. Виключення BrokenBarrierException.

1. Засіб синхронізації бар’єр або точка синхронізації

Бар’єр, або точка синхронізації, є одним із ключових засобів координації у паралельних та розподілених обчисленнях, який дозволяє групі потоків або процесів узгоджувати свій рух у межах спільного алгоритму. Його основна ідея полягає в тому, що певна частина обчислення не може бути продовжена, доки всі учасники не досягнуть визначеної точки – бар’єра. Це створює чіткий ритм виконання, у якому кожний потік завершує свою частину роботи, чекає на інших, а після синхронного вирівнювання всі потоки переходять до наступного етапу. Таким чином бар’єр забезпечує глобальну узгодженість алгоритму та запобігає ситуаціям, коли одні потоки працюють над частинами, що потребують результатів інших.

Особливістю бар’єра є його здатність створювати структуровані, тактовані етапи обчислення. У багатьох паралельних алгоритмах робота виконується фазами, де кожна фаза потребує завершення всіх попередніх кроків. Наприклад, у фізичних симуляціях, обробці даних або задачах чисельного моделювання кожен крок є результатом узгоджених обчислень усіх потоків. Бар’єр виступає механізмом, який гарантує, що нова фаза почнеться лише після того, як усі попередні обчислення завершено, і жоден з потоків не продовжить роботу передчасно. Це забезпечує узгодженість проміжних результатів і правильність глобального алгоритму.

У багатопотоковому програмуванні бар’єр часто використовується тоді, коли потоки виконують різні частини однієї задачі, але залежать від проміжних результатів один одного. Ситуації, у яких один потік може випередити інші, потенційно призводять до логічних помилок або неконсистентності. Бар’єр усуває цю проблему, встановлюючи точку, у межах якої всі виконавці повинні зупинитися й синхронізувати свій прогрес. Лише після того як останній потік досягне бар’єра, група синхронно переходить до наступного кроку. Така поведінка гарантує передбачуваність і стабільність алгоритму навіть за умов нерівномірного навантаження на потоки.

Концепція бар’єра має й важливий вплив на розподілені обчислення. У системах, де різні вузли працюють паралельно та виконують частини спільного завдання, час доставки даних може бути непередбачуваним, а швидкість виконання – різною. Бар’єр у таких умовах стає необхідним засобом синхронізації, що дозволяє вирівнювати фази виконання та гарантувати, що жоден вузол не переходить до нового етапу, не отримавши актуальні результати від інших. Це робить алгоритм стійким до асинхронності та забезпечує глобальну узгодженість, навіть коли вузли працюють у гетерогенному середовищі.

У практиці Java бар’єр реалізується за допомогою таких інструментів, як CyclicBarrier або Phaser, що надають можливість створювати багаторазові точки синхронізації. Потоки, що досягають бар’єра, блокуються, доки їх кількість не досягне заданої межі. Після цього бар’єр «знімається», і всі потоки одночасно продовжують виконання. Такий механізм дозволяє легко структурувати паралельні алгоритми, працювати із складними фазовими моделями та створювати обчислювальні схеми, де важливо підтримувати ритмічність і узгодженість.

У підсумку бар’єр є фундаментальним засобом синхронізації, який дозволяє створювати чітку структуру виконання у паралельних та розподілених системах. Він забезпечує логічну послідовність, узгодженість

даних і стабільність роботи, формуючи точки, у межах яких усі потоки вирівнюють свій прогрес і переходять до наступного етапу в єдиному ритмі. Завдяки цьому бар’єри залишаються одним із найважливіших механізмів для побудови коректних і ефективних паралельних алгоритмів.

2. Принципи взаємодії та синхронізації при використанні бар'єрної синхронізації, особливості програмування та застосування

Принцип взаємодії у бар’єрній моделі полягає в тому, що потоки, досягнувши бар’єра, переходять у стан очікування, зберігаючи своє місце в загальному алгоритмі. Вони не виконують активного «кручення у циклі», а блокуються конструкцією синхронізації, даючи змогу системі ефективно використовувати ресурси. Коли останній з потоків доходить до бар’єра, точка синхронізації знімається, і всі потоки одночасно розблоковуються й переходять до наступної стадії алгоритму. У цьому полягає ключова особливість бар’єра: він не просто контролює порядок виконання, а гарантує, що глобальний стан системи є узгодженим, а всі проміжні результати доступні й завершені до переходу на новий етап.

Бар’єрна синхронізація вимагає ретельного програмного проєктування. Розробник має створити структуру алгоритму таким чином, щоб кожна фаза була логічно завершеною та не вимагала доступу до результатів, що ще не сформовані. Неправильне розташування бар’єрів може призвести до неочевидних помилок: якщо хоча б один потік ніколи не досягне бар’єра, усі інші залишаться в режимі очікування, що спричиняє повну зупинку системи (deadlock). Тому бар’єри застосовують лише тоді, коли впевнені, що всі потоки гарантовано пройдуть усі етапи алгоритму. У процесі програмування важливо планувати шлях потоків так, щоб жоден з них не виходив із синхронізованого потоку роботи передчасно.

Особливістю бар’єрної синхронізації є можливість її багаторазового використання. На відміну від одноразових механізмів, бар’єр, особливо у вигляді CyclicBarrier або більш сучасного Phaser у Java, може бути застосований у циклі, синхронізуючи потоки у кількох послідовних фазах. Це

робить бар’єр зручним засобом для реалізації ітераційних алгоритмів, таких як чисельні методи, паралельна фільтрація, симуляції фізичних процесів, обробка блоків даних чи координація взаємопов’язаних етапів аналізу.

У застосуванні бар’єрної синхронізації відображається ще одна суттєва перевага – передбачуваність стану системи. Оскільки всі потоки переходять до нової фази одночасно, легко підтримувати спільні змінні, буфери або структури даних на межах етапів. Після бар’єра можна гарантувати, що всі обчислення попередньої фази завершені коректно й жоден потік більше не змінює спільний ресурс. Це знижує потребу в додаткових механізмах блокування та спрощує структуру паралельних програм, що особливо цінно в алгоритмах, де перед кожною новою ітерацією потрібна узгодженість проміжних результатів.

У ширшому значенні бар’єрна синхронізація застосовується в моделях обчислень, що базуються на парадигмах SPMD (Single Program, Multiple Data), MapReduce, GPU-обчисленнях та алгоритмах розподіленого моделювання. У всіх цих випадках бар’єр виконує однакову роль – формує чітко виражену фазову структуру, де результати однієї фази доступні всім учасникам, а наступна фаза запускається лише за умови повної завершеності попередньої. Таке вирівнювання створює стійкі й передбачувані паралельні процеси, що здатні працювати з великими обсягами інформації та забезпечувати коректність незалежно від швидкості окремих потоків або вузлів.

Бар’єрна синхронізація у Java ґрунтується на ідеї узгодженого руху групи потоків крізь певну послідовність етапів, де кожна наступна фаза може розпочатися лише після того, як усі учасники завершать попередню. У такому підході взаємодія між потоками відбувається не через прямий обмін даними, а через досягнення спільної точки очікування – бар’єра, який виконує роль контрольного рубежу. Потоки незалежно виконують свої частини обчислення і зупиняються біля бар’єра, переходячи до стану блокування. Вони залишаються в очікуванні до того моменту, коли останній

потік приєднається, після чого бар’єр знімається і всі виконавці синхронно продовжують роботу. Такий механізм формує чітку фазову структуру алгоритму, що забезпечує рівномірний, узгоджений і контрольований хід виконання.

Сутність взаємодії при бар’єрній синхронізації полягає в тому, що жоден потік не може перейти до наступного етапу раніше за інші. Це створює природну модель паралельних обчислень, у якій проміжні результати є стабільними та консистентними після завершення кожної фази. Якщо алгоритм залежить від даних, які формуються іншими потоками, бар’єр гарантує, що він почне використовувати ці дані лише тоді, коли всі необхідні обчислення виконано. Таке узгодження важливе для чисельних симуляцій, багатокрокових моделей, алгоритмів аналізу даних, де кожен етап ґрунтується на результатах попереднього.

Java надає два основні засоби реалізації бар’єрної синхронізації – CyclicBarrier і Phaser. Кожен із них забезпечує здатність потоків зупинятися на узгоджених точках, але має свої особливості поведінки. CyclicBarrier створює регульований бар’єр для фіксованої кількості учасників і може використовуватися повторно після кожного циклу, що робить його ідеальним вибором для класичних ітераційних алгоритмів. Потоки викликають метод await(), і бар’єр автоматично розблоковує їх, коли кількість учасників буде повною. Phaser, зі свого боку, пропонує більш гнучку модель, яка підтримує різні набори учасників у різних фазах, дозволяє потокам приєднуватися та виходити з процесу і надає змогу будувати складні багаторівневі схеми синхронізації.

Особливості програмування бар’єрної синхронізації в Java визначаються двома важливими умовами. Перша полягає в тому, що всі потоки, які повинні досягти бар’єра, мають гарантовано з’являтися в кожній фазі. Якщо хоча б один потік не викликає await() – решта потоків залишиться заблокованою, що спричинить повну зупинку системи. Тому необхідно ретельно проєктувати логіку алгоритму, щоб жоден виконавець не виходив за

межі спільного ритму роботи. Друга особливість стосується розподілу навантаження між потоками. Оскільки час руху групи визначається найповільнішим потоком, бажано забезпечувати приблизно однаковий обсяг операцій для кожного учасника або здійснювати динамічний розподіл роботи, коли це можливо.

У Java застосування бар’єрів є найприроднішим у задачах, що виконуються у вигляді послідовних фаз. Наприклад, у паралельних симуляціях фізичних процесів необхідно на кожному кроці оновлювати стан системи, обчислюючи взаємодію між об’єктами, а потім синхронізувати потоки, щоб усі мали актуальну інформацію для наступної ітерації. Подібний принцип використовується у фільтраційних алгоритмах, де набір потоків обробляє окремі частини даних, а після завершення обробки чергового блоку синхронізується, щоб перейти до наступного. Бар’єри також ефективні у схемах поділу задач на підетапи, де кожен підетап виконується паралельно, але потребує узгодження результатів перед переходом до наступного.

Таким чином бар’єрна синхронізація в Java є засобом, що дає змогу створювати чітко структуровані паралельні алгоритми, де кожен етап виконується узгоджено всіма потоками. Вона забезпечує глобальну узгодженість, стабільність проміжних даних і можливість підтримувати правильну послідовність обчислень навіть у складних багатофазних процесах. Завдяки інструментам CyclicBarrier та Phaser Java надає високий рівень керованості й гнучкості, що робить бар’єри одним із ключових механізмів сучасного мультипотокового програмування.

3. Клас CyclicBarrier. Приклад використання бар'єрної синхронізації для задачі множення двох матриць

Клас CyclicBarrier у Java є одним із ключових інструментів для реалізації бар’єрної синхронізації між потоками. Його концепція полягає в тому, що певна кількість потоків незалежно виконують свою частину обчислення і зупиняються у спільній точці очікування, доки всі учасники не завершать поточну фазу. Після того як останній потік приєднується до

бар’єра, синхронізаційний механізм дозволяє всім потокам одночасно перейти до наступного етапу. Особливістю CyclicBarrier є його багаторазовість: після завершення циклу бар’єр автоматично «перезавантажується», що робить його корисним для алгоритмів, які складаються з кількох послідовних фаз або ітерацій.

У задачах множення двох матриць використання CyclicBarrier дозволяє організувати паралельне виконання, коли множення розбивається на незалежні фрагменти. Множення матриць є класичною операцією, де результат складається з великої кількості елементарних обчислень, що не залежать один від одного. Кожен елемент результуючої матриці визначається скалярним добутком відповідного рядка першої матриці та стовпця другої. Ця властивість дає можливість розподілити роботу між потоками так, щоб кожен із них обчислював певну кількість елементів або рядків результату. Проте після того як потік завершує свою частину, йому необхідно синхронізуватися з іншими потоками, щоб гарантувати цілісність загального етапу та запобігти передчасному переходу до наступної фази, наприклад, під час перевірки або подальшої обробки результатів.

Бар’єрна синхронізація у цьому контексті забезпечує чітке завершення кожного етапу обчислень. Кожен потік, обчисливши свій блок елементів результуючої матриці, викликає метод await(), який переводить його у стан блокування. Потоки накопичуються у точці синхронізації, доки останній з них не завершить обробку свого фрагмента. Після цього бар’єр знімається, і всі потоки одночасно переходять до наступної частини алгоритму. Такий підхід гарантує, що після завершення множення всіх частин матриць сформований результат є повністю коректним, а кожен наступний етап алгоритму може опиратися на узгоджені проміжні дані.

У Java подібний алгоритм реалізується через створення CyclicBarrier із кількістю учасників, що дорівнює числу потоків, задіяних у множенні. Кожен потік стартує окрему частину роботи, отримуючи доступ до фрагментів матриць або індексів, які він має обчислювати. Після виконання своєї

частини потік доходить до бар’єра й чекає на інші потоки, поки всі вони не завершать відповідний етап. Після зняття бар’єра потоки можуть або перейти до завершальної фази обробки результату, або почати нову ітерацію, якщо множення виконується блоками чи є частиною більшого ітераційного процесу.

У прикладі множення двох матриць CyclicBarrier виступає не лише засобом синхронізації, а й інструментом контролю цілісності обчислення. Оскільки результат формується поступово, важливо, щоб жоден потік не працював із неповними або ще не сформованими даними. Бар’єр створює чітку межу, яка відділяє паралельні обчислення від наступних кроків, забезпечуючи стабільність алгоритму. У великих обчислювальних системах такий підхід дозволяє ефективно використовувати ресурси процесора, рівномірно розподіляти навантаження і гарантувати коректність результатів у багатопотоковому середовищі.

У підсумку CyclicBarrier у задачі множення матриць виступає ідеальним механізмом для координації роботи потоків, який забезпечує чітку організацію паралельного алгоритму, контроль консистентності і можливість масштабування операції на велику кількість обчислювальних ядер. Він демонструє, як бар’єрні засоби синхронізації можуть бути інтегровані у реальні алгоритми, забезпечуючи ефективність і передбачуваність навіть у задачах з великими обсягами паралельної роботи.

4. Виключення BrokenBarrierException

Виключення BrokenBarrierException є характерною складовою механізму бар’єрної синхронізації в Java і відіграє особливу роль у підтриманні коректності та надійності роботи CyclicBarrier. Його природа пов’язана з тим, що бар’єр передбачає узгоджений рух кількох потоків: усі вони повинні дійти до точки синхронізації та чекати один на одного. Якщо ця узгодженість порушується – наприклад, один із потоків не може завершити свою частину роботи або переривається під час очікування – бар’єр

переходить у пошкоджений стан, і система сигналізує про це через BrokenBarrierException.

Ключовою особливістю цього виключення є те, що воно не описує помилку обчислення, а інформує про порушення логіки взаємодії потоків. Коли один потік переривається або завершує роботу раніше, ніж очікується, бар'єр стає «зламаним» і більше не може служити точкою синхронізації, оскільки він уже не здатний зібрати повну групу учасників. У такій ситуації Java автоматично переводить бар'єр у непрацездатний стан, щоб запобігти небезпечному або непередбачуваному продовженню роботи інших потоків, які могли б нескінченно чекати або працювати з неповними даними.

У момент, коли бар'єр зазнає порушення, усі потоки, що очікують у методі await(), негайно пробуджуються й отримують BrokenBarrierException. Це є механізмом захисту, оскільки така ситуація означає, що подальше узгоджене виконання неможливе і алгоритм потребує спеціальної обробки. Виключення дозволяє потокам вийти зі стану очікування не за звичайним сценарієм переходу до наступної фази, а через обробку аварійної ситуації, що дає змогу розробнику вжити заходів – від завершення всієї обчислювальної фази до ініціалізації нового циклу синхронізації.

Причиною виникнення BrokenBarrierException може бути не лише переривання одного з потоків. Бар'єр може стати «зламаним» також через логічні помилки, що змушують потік завершити виконання раніше або не досягти точки синхронізації. У таких випадках інші потоки, які акуратно виконують свою частину роботи й очікують при бар'єрі, зіштовхуються з тим, що група більше не є повною, і продовження алгоритму в нормальному режимі стає неможливим. Це особливо важливо у великомасштабних алгоритмах, де розподіл роботи між потоками може бути складним, і помилка в одному з них може вплинути на всю систему.

У контексті програмування на Java важливими є два моменти. По-перше, метод await() завжди повинен викликатися у межах конструкції обробки винятків, оскільки BrokenBarrierException є однією з двох можливих

аномалій – другою є InterruptedException. По-друге, після того як бар’єр стає «зламаним», його подальше використання вимагає або явної перевірки його стану, або створення нового екземпляра CyclicBarrier, оскільки повернути колишню працездатність об’єкту неможливо. Така поведінка має важливий сенс: вона гарантує, що потоки не продовжуватимуть роботу, яка втратила узгодженість, і тим самим запобігає логічним помилкам і некоректним результатам.

У підсумку BrokenBarrierException є механізмом контролю надійності бар’єрної синхронізації, що сигналізує про неможливість продовження спільного руху потоків через порушення узгодженості. Він забезпечує захист алгоритму від ситуацій, де один із потоків втрачається або завершує роботу передчасно, не дозволяючи іншим продовжувати обчислення на основі неповних або застарілих даних. Завдяки цьому виключення є важливою частиною архітектури паралельного програмування в Java, забезпечуючи стабільність, передбачуваність і коректність виконання складних багатопотокових алгоритмів.

ТЕМА 3.4. БЛОКУЮЧИ ЧЕРГИ

План

1. Блокуюча черга, як дієвий механізм синхронізації процесів. Клас BlockingQueue.
2. Відмінності між блокуючою та неблокуючою чергами.
3. Види черг: ArrayBlockingQueue, LinkedBlockingQueue, PriorityBlockingQueue, DelayQueue, LinkedTransferQueue, SynchronousQueue та ConcurrentLinkedQueue.
4. Приклад поелементного складання двох матриць за допомогою ArrayBlockingQueue.

1. Блокуюча черга, як дієвий механізм синхронізації процесів. Клас BlockingQueue

Блокуюча черга є одним із найефективніших і найбільш природних механізмів синхронізації між потоками в Java, оскільки вона поєднує в собі одразу два аспекти взаємодії – організоване передавання даних і автоматичну координацію потоків у моменти, коли ресурс недоступний. На відміну від класичних засобів синхронізації, де програміст повинен вручну керувати блокуваннями та умовами очікування, блокуюча черга сама забезпечує правильну поведінку, гарантуючи, що потоки діятимуть узгоджено у процесі додавання й вилучення елементів. Завдяки цьому вона виступає не лише контейнером даних, а й повноцінним засобом координації та взаємодії між потоками.

Основний принцип роботи блокуючої черги полягає в тому, що потік, який намагається додати елемент до повної черги, не отримує помилки й не продовжує виконання, а автоматично переходить у стан очікування. Він пробуджується лише тоді, коли інший потік вилучить елемент і звільнить місце. Аналогічно працює і механізм споживання даних: коли потік намагається отримати елемент з порожньої черги, він чекає, доки інший потік не додасть новий елемент. Така поведінка створює природну модель

виробник–споживач, де різні частини програми можуть працювати з різною швидкістю, але без ризику втрати даних або непередбачуваної конкуренції за ресурс.

У Java ця модель реалізована через інтерфейс BlockingQueue, який визначає набір операцій для безпечного додавання, взяття та керованої взаємодії з елементами черги. Його реалізації – такі як ArrayBlockingQueue, LinkedBlockingQueue, PriorityBlockingQueue, SynchronousQueue – надають різні стратегії організації черг: від фіксованого розміру до повністю динамічних структур. Незалежно від конкретної реалізації, усі блокуючі черги дотримуються однакового принципу: вони поєднують у собі механізм передачі даних і автоматичне блокування потоків тоді, коли відповідна операція неможлива.

Особливість блокуючої черги як механізму синхронізації полягає в тому, що вона інкапсулює всі необхідні примітиви взаємодії – блокування, умовні змінні, пробудження потоків – і приховує їх від програміста. Це звільняє розробника від ручного керування конструкціями типу wait(), notify() або складними схемами з м’ютексами. У моделі з блокуючою чергою сама структура гарантує, що взаємодія потоків відбувається коректно, без ризику гонок даних, взаємних блокувань чи інших поширених проблем синхронізації. У результаті код стає не лише простішим, а й більш надійним та прогнозованим у поведінці.

У паралельних і розподілених системах BlockingQueue часто використовується як канал комунікації, через який передається робота, дані або запити. Потоки-виробники додають завдання до черги, а потоки-споживачі отримують їх і обробляють. Такий підхід дозволяє створювати гнучкі, масштабовані системи, де кількість потоків-виконавців можна змінювати без зміни логіки передавання даних. Навіть у складних сценаріях блокуюча черга гарантує, що потоки синхронізуються природним чином, зберігаючи правильну послідовність взаємодії й забезпечуючи ефективне використання ресурсів процесора.

Завдяки своїм властивостям блокуюча черга стала одним із ключових механізмів у побудові багатопотокових програм у Java. Вона поєднує простоту використання, безпечне управління потоками та високий рівень апаратної ефективності, що дає змогу створювати стабільні й продуктивні алгоритми взаємодії між виробниками та споживачами. Саме тому BlockingQueue є незамінним інструментом у розробці серверних систем, потокових обробників даних, асинхронних обчислень та багатьох інших паралельних архітектур.

2. Відмінності між блокуючою та неблокуючою чергами

Блокуюча черга є однією з ключових абстракцій у багатопотоковому програмуванні, оскільки вона поєднує механізм упорядкованого зберігання даних із вбудованою логікою синхронізації між потоками. На відміну від звичайних колекцій, блокуюча черга автоматично контролює взаємодію між виробниками та споживачами, зупиняючи потік, який не може виконати операцію, і відновлюючи його лише тоді, коли це стає можливим. Завдяки цьому черга виконує роль надійного каналу передачі інформації між потоками, де синхронізація відбувається не через явні блокування або низькорівневі примітиви, а через саму структуру даних.

Сутність блокуючої черги полягає в її здатності змінювати поведінку залежно від поточного стану – якщо черга заповнена, спроба додати новий елемент призводить до автоматичного блокування потоку, а коли інший потік вилучає елемент і звільняє місце, заблокований виробник пробуджується й продовжує роботу. Аналогічно працює ситуація, коли черга порожня: спроба споживача отримати елемент призводить до очікування до моменту появи нового елемента. Таким чином формується природний «такт» взаємодії між потоками, у якому швидші потоки не випереджають повільніші, а сама система синхронізується без додаткових умовних змінних чи сигналів.

У моделі виробник–споживач блокуюча черга забезпечує ідеальні умови для безпечної й ефективної взаємодії потоків. Виробники можуть

генерувати завдання з будь-якою швидкістю, але система автоматично стримує їх, якщо споживачі не встигають обробляти інформацію. Це запобігає ситуаціям переповнення буфера, втраті даних або виникненню гонок доступу. У свою чергу споживачі отримують нові елементи без активного очікування чи перевірок стану – черга сама блокує їх, доки дані не будуть доступними. Завдяки цьому блокуюча черга зберігає стабільність і передбачуваність навіть у складних багатопотокових системах.

У Java механізм блокуючої черги реалізований через інтерфейс BlockingQueue, що надає засоби автоматичної синхронізації, вбудовані у саму структуру даних. Коли потік викликає методи put() або take(), він не працює з примітивами блокування явно – усі блокування, очікування та пробудження здійснюються всередині реалізації черги. Це робить код простішим і менш схильним до помилок, які неминуче виникають при ручному використанні м’ютексів, умовних змінних або методів wait() і notify().

Завдяки своїм властивостям блокуючі черги широко застосовуються в багатопотокових і серверних додатках. Вони формують основу високопродуктивних пулів потоків, механізмів обробки подій, транспортних каналів між асинхронними компонентами та систем, побудованих за принципом потокового або конвеєрного оброблення даних. Така природність взаємодії потоків через блокуючу чергу робить її незамінною у будь-який ситуації, де необхідно організувати чіткий порядок передачі інформації та водночас автоматично збалансувати роботу між продуктивними та повільнішими частинами системи.

У підсумку блокуюча черга є не просто структурою даних, а цілісним механізмом керованої синхронізації потоків. Її використання дозволяє позбутися великої кількості низькорівневих конструкцій, що робить паралельні програми простішими, надійнішими та легшими для супроводу. Саме завдяки такій поєднаній функціональності вона залишається одним із найважливіших інструментів у сучасному паралельному програмуванні й

продовжує відігравати ключову роль у побудові масштабованих та ефективних систем.

Неблокуюча черга є альтернативним підходом до організації взаємодії між потоками, який відмовляється від традиційних механізмів блокування на користь безблокових, або lock-free, алгоритмів. Її основна ідея полягає в тому, що потік ніколи не переходить у стан очікування, незалежно від того, чи може він виконати операцію додавання або вилучення елемента. Замість блокування неблокуюча черга використовує атомарні операції порівняння та обміну, які дозволяють потокам безпечно взаємодіяти із загальною структурою даних, не перешкоджаючи роботі один одного. Такий підхід мінімізує затримки, дозволяє уникнути блокувань і робить систему значно стійкішою до ситуацій конкуренції за ресурс.

Сутність неблокуючої черги полягає у відмові від примітивів, що змушують потоки чекати. Якщо структура зайнята або операцію неможливо завершити негайно, потік не блокується, а повторює спробу з використанням атомарної операції. Це створює поведінку, у якій потоки взаємодіють у режимі активного просування, не очікуючи на чиюсь чергу. У порівнянні з блокуючими структурами неблокуюча черга не використовує умовних змінних або блокувань, тому ризик виникнення взаємних блокувань, затримок або небажаних черг очікування повністю зникає. Навіть у випадках надмірного навантаження така структура демонструє передбачувану поведінку, реагуючи на зміни стану черги через повторні спроби виконання операцій.

У Java неблокуюча поведінка реалізована в таких структурах, як ConcurrentLinkedQueue, яка базується на алгоритмах, розроблених за принципом lock-free. Ця черга дозволяє одночасні операції додавання та вилучення без використання блокувань завдяки атомарним операціям compareAndSet(). Кожен потік взаємодіє з окремими фрагментами структури, і лише за потреби намагається оновити спільний стан. Якщо двом потокам одночасно потрібно змінити один і той самий елемент, переможе той, хто

першим виконає атомарну операцію, тоді як інший повторить спробу. Такий механізм, хоч і не гарантує повної відсутності повторних операцій, забезпечує просування всіх потоків без затримок і запобігає можливості «зависання» системи.

Неблокуюча черга є особливо корисною у високопродуктивних системах, де блокування може стати вузьким місцем. У серверних застосунках, мережевих процесорах, потокових системах обробки даних та ігрових рушіях неблокуючий доступ дозволяє суттєво зменшити час реакції та забезпечити плавний розподіл навантаження навіть за дуже великої кількості активних потоків. Такі черги добре масштабуються, оскільки збільшення кількості потоків не призводить до накопичення блокувань, а лише до збільшення кількості атомарних операцій, які апаратно виконуються надзвичайно швидко.

Незважаючи на численні переваги, неблокуючі черги вимагають ретельного проєктування. Вони не гарантують, що всі потоки будуть просуватися рівномірно, адже деякі з них можуть робити більше повторних спроб. Також безблокові алгоритми складніші для реалізації й потребують глибокого розуміння взаємодії між потоками на рівні апаратних інструкцій. Проте в Java ці складності приховані за інтерфейсом високого рівня, що дозволяє розробникам використовувати переваги неблокуючих структур без необхідності реалізовувати їх вручну.

У підсумку неблокуюча черга є важливим інструментом для побудови високопродуктивних, масштабованих і стійких до затримок паралельних систем. Вона демонструє зовсім іншу філософію взаємодії між потоками – не очікування, а постійне активне просування – і завдяки цьому забезпечує ефективність навіть у найскладніших обчислювальних середовищах. Її роль у сучасному паралельному програмуванні постійно зростає, адже дедалі більше систем переходять від блокувальних до безблокових моделей керування доступом.

3. Види черг: ArrayBlockingQueue, LinkedBlockingQueue, PriorityBlockingQueue, DelayQueue, LinkedTransferQueue, SynchronousQueue та ConcurrentLinkedQueue

У Java існує низка різновидів черг, які реалізують різні стратегії взаємодії між потоками та різні механізми організації зберігання елементів. Кожен тип черги призначений для конкретних сценаріїв, від моделі виробник–споживач до побудови складних асинхронних систем, де елементи можуть бути впорядковані за пріоритетом або затримані до певного моменту часу. Вибір конкретного типу черги визначає спосіб синхронізації, пропускну здатність системи та характер оброблення даних у багатопотоковому середовищі.

Однією з найпоширеніших є ArrayBlockingQueue, яка використовує масив фіксованої довжини. Її характерна риса – обмежена ємність, яка задається під час створення і не змінюється надалі. Завдяки тому, що структура побудована на масиві, доступ до елементів та їхнє переміщення є передбачуваними та ефективними. Така черга підходить для ситуацій, у яких необхідно чітко контролювати обсяг буфера і гарантувати рівновагу між виробниками та споживачами. У випадку переповнення потоки-виробники зупиняються, даючи можливість системі стабілізуватися.

На противагу цьому LinkedBlockingQueue використовує зв’язану структуру, що дозволяє динамічно збільшувати розмір черги у межах заданого або необмеженого значення. Така гнучкість робить її придатною для сценаріїв, де важко передбачити кількість елементів або інтенсивність надходження даних. Потоки не блокуються надто рано, оскільки черга може рости, і завдяки цьому зменшується ризик штучних затримок у роботі виробників.

Інший тип – PriorityBlockingQueue – упорядковує елементи не за часом надходження, а за їхнім пріоритетом. Це означає, що черга автоматично вибирає елемент з найвищим пріоритетом, незалежно від того, коли його було додано. Такий підхід широко використовується в системах планування

завдань, де важливіші операції повинні виконуватися першими, навіть якщо вони надійшли пізніше за інші. Черга в цьому випадку працює як внутрішній диспетчер, який забезпечує справедливе або критеріальне впорядкування потоків робіт.

Спеціальний різновид черги – DelayQueue – працює за принципом тимчасового блокування елементів. Кожен елемент має вбудований час затримки, після завершення якого він стає доступним для вилучення. До того моменту він «недоступний» для споживачів. Такий підхід є незамінним у задачах, де важливо, щоб певні дії виконувалися не відразу, а через визначений інтервал часу: планування завдань, оновлення кешу або відкладена передача повідомлень. Черга сама визначає момент, коли елемент готовий до обробки.

Ще один різновид – LinkedTransferQueue – поєднує властивості черги та механізму прямої передачі елементів між потоками. Якщо потік-виробник намагається передати елемент, а споживач уже чекає, передача здійснюється негайно, без додавання елемента до структури черги. Якщо ж споживача немає, елемент зберігається у черзі до появи потоку, який його забере. Такий механізм є особливо ефективним у великих потокових системах, де важливо мінімізувати затримки між частинами програми, що взаємодіють у режимі реального часу.

SynchronousQueue вирізняється тим, що взагалі не містить внутрішнього буфера. Будь-яке додавання до неї відбувається лише тоді, коли інший потік готовий негайно забрати елемент. Це найчистіша модель прямої передачі даних між потоками, де черга є лише точкою узгодженого обміну. Такий механізм підходить для систем, у яких виробник і споживач повинні працювати в одному ритмі, забезпечуючи ідеальну синхронізацію.

У протилежність усім попереднім структурам ConcurrentLinkedQueue реалізує неблокуючу поведінку і дозволяє потокам взаємодіяти без блокувань. Вона використовує атомарні операції й забезпечує високу пропускну здатність у середовищах, де велика кількість потоків одночасно

додають і вилучають елементи. Такий підхід усуває ризики блокування й дає змогу системі масштабуватися без створення черг очікування, що робить її однією з найефективніших структур для високонавантажених паралельних застосунків.

У підсумку всі ці типи черг демонструють різні моделі синхронізації між потоками – від жорстко контрольованих фіксованих буферів до безблокових структур з необмеженим ростом. Кожен різновид черги розв'язує свою категорію задач, надаючи програмісту інструменти для побудови ефективних, передбачуваних і масштабованих паралельних систем. Завдяки такій різноманітності Java забезпечує широкий спектр механізмів взаємодії між потоками, що дозволяє адаптувати синхронізацію до специфіки будь-якого алгоритму чи архітектури.

4. Приклад поелементного складання двох матриць за допомогою ArrayBlockingQueue

Поелементне складання двох матриць є однією з найпростіших операцій лінійної алгебри, проте саме завдяки своїй структурі вона добре підходить для демонстрації принципів взаємодії потоків через блокуючу чергу ArrayBlockingQueue. У такій моделі обчислення розділяється на два логічні процеси: перший потік виконує роль виробника, формуючи і надсилаючи у чергу пари елементів, які потрібно скласти, тоді як другий потік виступає споживачем, вилучаючи ці пари з черги, виконуючи обчислення та формуючи результат. Завдяки тому, що черга має фіксовану місткість, вона сама регулює швидкість взаємодії між потоками, створюючи природний баланс між виробленням і споживанням завдань.

У процесі поелементного складання дві матриці однакового розміру обробляються послідовно за індексами. Виробник здійснює ітерацію через рядки та стовпці, утворюючи для кожної позиції пару значень, яка представляє елементи з обох матриць. Ці пари передаються до ArrayBlockingQueue, яка виконує роль буфера між потоками. Коли черга заповнюється, виробник автоматично блокується у момент виклику методу

put(), і продовжує роботу лише тоді, коли споживач звільнить місце, вилучивши черговий елемент. Такий механізм дає змогу гарантувати, що виробник не випереджає споживача, а швидкість передачі даних синхронізується автоматично.

Споживач, зі своєї сторони, постійно вилучає пари елементів за допомогою методу take(), який блокує потік, якщо черга тимчасово порожня. Отримавши пару значень, споживач обчислює їхню суму й записує результат у відповідну позицію результуючої матриці. Завдяки блокуючій поведінці ArrayBlockingQueue споживач не потребує додаткових перевірок або умов – він працює в ритмі, заданому виробником. Навіть за ситуації, коли виробник тимчасово не встигає, споживач автоматично переходить у режим очікування й не створює зайвого навантаження на процесор.

Такий підхід демонструє перевагу блокуючої черги як засобу синхронізації, особливо в задачах, де потоки працюють з різною швидкістю або мають неоднакове навантаження. ArrayBlockingQueue поєднує у собі роль буфера і механізму синхронізації, що дозволяє уникнути необхідності використовувати м’ютекси, умовні змінні або ручні схеми блокування. Черга природним чином вирівнює роботу потоків і гарантує, що обробка елементів матриці буде виконуватися послідовно та узгоджено.

У ширшому сенсі приклад поелементного складання матриць є демонстрацією того, як прості математичні операції можуть бути розділені між потоками за допомогою блокуючої черги. Така модель легко поширюється на складніші задачі, у яких об’єктами передачі можуть бути не лише окремі числа, але й рядки матриці, блоки даних або великі структури. Незмінною залишається поведінка ArrayBlockingQueue, яка забезпечує стабільний, керований і передбачуваний обмін даними між потоками, роблячи її одним із найзручніших інструментів для реалізації паралельних алгоритмів у Java.

ТЕМА 3.5. ОБРАХУНОК ВИЗНАЧЕНОГО ІНТЕГРАЛУ З ВИКОРИСТАННЯМ МЕХАНІЗМУ СИНХРОНІЗАЦІЇ COUNTDOWNLATCH

План

1. Клас CountDownLatch та його метод countDown().
2. Принципи взаємодії та часової синхронізації процесів під час використання класу CountDownLatch, особливості програмування та застосування.
3. Визначений інтеграл. Метод прямокутників та метод трапецій. Приклад реалізації ітеративного обрахунку визначеного інтеграла з використанням лівосторонніх прямокутників.

1. Клас CountDownLatch та його метод countDown()

Клас CountDownLatch у Java є одним із базових засобів синхронізації, який забезпечує можливість затримати виконання певного потоку до того моменту, коли визначена кількість подій або дій буде завершена. Цей механізм використовують для організації узгодженого старту процесів, очікування завершення групи потоків, а також для побудови багатофазних алгоритмів, де перехід до наступного етапу можливий лише після виконання певного набору операцій. Основна ідея полягає в тому, що лічильник, закладений у CountDownLatch, задає кількість умов, які повинні бути виконані, і лише коли лічильник досягає нуля, заблоковані потоки можуть продовжити свою роботу.

У центрі роботи CountDownLatch лежить початкове значення лічильника, яке встановлюється в момент створення об’єкта. Цей лічильник відображає кількість очікуваних подій – наприклад, завершення декількох потоків, виконання певної кількості завдань або підготовку компонентів системи. Поки лічильник не досягне нуля, потоки, що викликають метод await(), залишаються заблокованими. Вони не продовжать виконання, доки кожна з подій не буде позначена як завершена.

Метод countDown() є ключовою операцією, за допомогою якої поток повідомляє про виконання своєї частини роботи. Кожен виклик цього методу зменшує значення лічильника на одиницю. Коли останнє зменшення доводить лічильник до нуля, усі потоки, що чекали в режимі await(), одразу розблоковуються і продовжують виконання. Цей механізм дозволяє створювати чітко визначені точки синхронізації, у межах яких виконання одного процесу залежить від завершення іншого. countDown() не блокує потік, який його викликає – він лише сигналізує, що одна з попередніх умов завершена.

Принципова відмінність CountDownLatch від інших засобів синхронізації полягає в його одноразовості. Після того як лічильник досягає нуля, його значення більше не можна змінити, і лічильник не відновлюється. Об’єкт CountDownLatch у цьому сенсі є одноразовим бар’єром, який працює лише один цикл, але робить це з максимальною надійністю й передбачуваністю. Завдяки такій властивості його широко застосовують у випадках, коли потрібно дочекатися завершення великої кількості підготовчих дій перед запуском основної частини алгоритму або завершити основний потік лише після того, як усі допоміжні потоки виконають свої завдання.

У багатопотокових системах CountDownLatch дає змогу створити чітку і прозору модель взаємодії, де один або кілька потоків виступають у ролі очікувачів, а інші – у ролі виконавців, які поступово наближають систему до моменту, коли виконання може бути продовжене. Завдяки методам await() та countDown() розробник може керувати рухом потоків без необхідності використовувати складні конструкції ручного блокування. Це робить CountDownLatch одним із найпоширеніших і найзручніших інструментів у задачах синхронізації, де важливо забезпечити контрольоване й передбачуване завершення групи паралельних дій.

2. Принципи взаємодії та часової синхронізації процесів під час використання класу CountDownLatch, особливості програмування та застосування

Використання CountDownLatch у багатопотокових програмних системах ґрунтується на ідеї створення чіткої точки синхронізації, у межах якої один або кілька потоків очікують завершення визначеного набору дій. Цей підхід дає змогу впорядковувати виконання паралельних обчислень і забезпечувати розподіл роботи таким чином, щоб жоден процес не розпочинав наступний етап раніше, ніж будуть виконані всі необхідні попередні операції. У реальному часі CountDownLatch працює як своєрідний часовий замок: потоки можуть рухатися тільки тоді, коли відпрацьована кумулятивна кількість подій, яку було встановлено на початку.

Механізм взаємодії між потоками, що використовують CountDownLatch, побудований на очікуванні та сигналізації. Потоки, яким необхідно дочекатися певного стану системи, викликають метод await() і переходять у пасивний режим, не використовуючи процесорного часу і не заважаючи роботи іншим потокам. У цей час інші потоки виконують свої завдання, і кожне їхнє завершення супроводжується викликом countDown(). У момент, коли кількість таких сигналів досягає нуля, усі потоки, що очікували, одночасно розблоковуються і починають подальше виконання. Так створюється момент синхронізованого переходу всієї системи до нового етапу обчислень.

Принцип часової синхронізації в CountDownLatch є особливо важливим у ситуаціях, де різні частини системи мають працювати у строго визначеному порядку. Наприклад, важливою є гарантія того, що основний процес не розпочне обчислення, доки не завершаться всі підготовчі операції; або навпаки – що завершальний етап не стартує доти, доки всі робочі потоки не виконають свої частини задачі. Значення CountDownLatch полягає у можливості природно узгодити роботу потоків, які можуть мати різну швидкість, різний обсяг роботи й навіть різні алгоритмічні підходи.

Незалежно від цього, синхронізація залишається стабільною та передбачуваною.

Особливості програмування з використанням CountDownLatch пов’язані з його одноразовим характером. Після того як лічильник досягає нуля, об’єкт втрачає свою синхронізаційну функцію, і повторно використати його неможливо. Це означає, що у складних системах, які складаються з багатьох етапів, необхідно створювати окремий екземпляр CountDownLatch для кожної фази. Водночас одноразовість підсилює передбачуваність: відсутній ризик того, що лічильник буде випадково змінено або перезапущено під час роботи.

Застосування CountDownLatch є надзвичайно широким. Цей механізм використовують для організації одночасного старту групи робочих потоків, коли важливо, щоб усі вони почали роботу у строго визначений момент. Його застосовують також у зворотному сценарії, де основний потік має дочекатися завершення всіх допоміжних потоків перед тим, як завершити програму або перейти до нової фази обчислень. У тестових системах і стрес-тестуванні CountDownLatch дозволяє створювати штучний одночасний запуск великої кількості потоків, що дає можливість перевірити поведінку програми в умовах значного паралельного навантаження.

Система взаємодії через CountDownLatch забезпечує потужний інструментарій для побудови алгоритмів, у яких важливе значення має саме момент переходу між етапами. Його здатність координувати процеси без активного циклічного очікування, а через механізм блокування, робить його ефективним і легким у використанні. Він дозволяє будувати алгоритми, які працюють у строго визначеній часовій логіці, контролюючи складні взаємозв’язки між процесами і забезпечуючи коректність у багатопотоковому середовищі. Саме тому CountDownLatch залишається одним із ключових інструментів синхронізації у Java, поєднуючи простоту реалізації та високу надійність у складних паралельних системах.

3. Визначений інтеграл. Метод прямокутників та метод трапецій. Приклад реалізації ітеративного обрахунку визначеного інтеграла з використанням лівосторонніх прямокутників.

Розрахунок визначеного інтеграла у чисельній формі, зокрема за допомогою методу прямокутників або методу трапецій, набуває нового змісту, коли процес інтегрування переходить у багатопотоковий режим. У традиційному підході інтеграл оцінюється на основі розбиття відрізка на підінтервали та послідовного додавання площ елементарних фігур. Однак у багатопотокових системах цей же принцип дає можливість природного розподілу обчислень між кількома потоками, оскільки кожен підінтервал може бути оброблений незалежно від інших. Власне, інтегральна сума через свою дискретну структуру є однією з найзручніших задач для паралельного виконання.

Метод прямокутників, а саме варіант із лівими точками, у паралельному середовищі працює за тією ж математичною ідеєю: площа на кожному підінтервалі наближується прямокутником, висота якого визначається значенням функції у лівій точці, а ширина – довжиною підінтервалу. Але замість одного потоку, який послідовно обчислює площі всіх прямокутників, робота розподіляється між кількома потоками. Кожен з них отримує певну частину відрізка, обчислює свою часткову інтегральну суму і передає результат до спільного накопичувача або повертає його у вигляді власного локального значення. Оскільки значення підінтервалів не перетинаються, а обчислення всередині кожної частини незалежні, паралельна схема майже повністю усуває необхідність блокування та синхронізації, за винятком моменту зведення результатів.

У типовій реалізації процес починається з поділу відрізка [a, b] на n рівних частин. Потім підінтервали групуються у більші ділянки, кожна з яких відповідає одному потоку. Усі потоки працюють одночасно, кожен застосовуючи метод лівосторонніх прямокутників до своєї частини відрізка. Математична природа задачі гарантує, що незалежні часткові суми можуть

бути додані без ризику порушення точності або коректності. Завдяки цьому розпаралелювання інтегрування стає практично лінійно масштабованим – збільшення кількості потоків зазвичай веде до пропорційного скорочення часу виконання, особливо коли n є великим.

Організаційно найпростіша схема базується на пулі потоків (ExecutorService) та задачах, кожна з яких відповідає за окремий підфрагмент інтегралу. Потоки виконують свої обчислення незалежно, а результат їхньої роботи повертається через механізм Future або додається до спільної суми через безпечну структуру даних, таку як AtomicDouble чи блокування короткої тривалості. Узгодження потоків вимагає мінімальної синхронізації, оскільки операції над кожним підінтервалом є повністю локальними, а єдине місце взаємодії – це фінальне підсумовування.

Метод трапецій аналогічним чином підходить для розпаралелювання. Він використовує значення функції в обох кінцях підінтервалу, а отже, також може бути обчислений незалежно на кожному фрагменті відрізка. Єдиною особливістю є те, що спільні межі підінтервалів мають бути використані лише один раз, але навіть ця деталь добре узгоджується у багатопотоковій моделі за допомогою акуратного розподілу точок або простого узгодження у момент підсумовування.

Паралельна реалізація інтегрування методом лівосторонніх прямокутників є прикладом того, як проста математична операція може стати ефективною та масштабованою в умовах багатопотокової обробки. Незалежність локальних обчислень, відсутність спільних змінних у робочих частинах алгоритму та простота поділу задачі роблять цей підхід майже ідеальним для демонстрації переваг паралельного програмування. Завдяки розпаралелюванню інтегральні задачі великого обсягу виконуються значно швидше, а архітектура алгоритму зберігає свою прозорість, надійність і математичну строгість.

ТЕМА 3.6. ВИРІШЕННЯ ОБЧИСЛЮВАЛЬНИХ ЗАДАЧ МЕТОДОМ МОНТЕ-КАРЛО

План

1. Методика програмування у функціональному стилі програмування.
2. Генерація, фільтрація та підрахунок даних. Пакет java.util.stream.
3. Функціональні потоки.
4. Приклад обрахунку співвідношення об'ємів куба та вписаної у нього сфери методом Монте-Карло

1. Методика програмування у функціональному стилі програмування

Програмування у функціональному стилі ґрунтується на ідеї розгляду обчислення як послідовності застосувань функцій, що перетворюють дані, а не як маніпуляції зі станами чи змінними. Такий підхід суттєво відрізняється від традиційних імперативних методик, у яких значення змінюється крок за кроком, а сам алгоритм описує, як досягти результату. У функціональній парадигмі основний акцент робиться на тому, що саме потрібно обчислити, і це дозволяє створювати програми, які є більш передбачуваними, математично строгими і придатними до паралельної або розподіленої обробки.

Однією з ключових ідей функціонального стилю є незмінність даних. Замість зміни існуючих структур створюються нові, які відображають результат перетворення. Це забезпечує відсутність побічних ефектів і робить функції чистими, тобто такими, що залежать лише від вхідних параметрів і завжди повертають однаковий результат при однакових даних. Чисті функції легко тестувати, комбінувати та розподіляти між потоками, оскільки вони не модифікують зовнішній стан і не вимагають синхронізації доступу до спільних ресурсів. Саме тому функціональний стиль природним чином підсилює коректність алгоритмів і дозволяє уникати багатьох типових помилок, пов’язаних зі зміною змінних у багатопотокових середовищах.

Функціональний підхід також передбачає широке використання вищих функцій – тобто функцій, які приймають як аргументи інші функції або повертають їх як результат. Такий механізм дозволяє формувати цілі композиції обчислень, де кожне перетворення є окремою функцією, а їх поєднання дає складну логіку. Цей спосіб програмування не лише робить код компактним і декларативним, а й дозволяє динамічно формувати алгоритми, що адаптуються до умов виконання. У мовах, які підтримують лямбда-вирази, композицію функцій, карування або часткове застосування параметрів, функціональний стиль набуває особливої виразності та зручності.

Завдяки відсутності змінюваного стану функціональні програми легко оптимізуються компілятором або виконуючим середовищем. Кешування результатів, ліниве обчислення, паралельне виконання та інші оптимізації стають природними, оскільки чисті функції не залежать від контексту. Це дозволяє створювати програми, які працюють значно ефективніше при обробці великих масивів даних або потокових структур. Крім того, саме функціональний стиль лежить в основі сучасних підходів до обробки даних – від MapReduce і потокових API до реактивних систем, де дані перетворюються послідовно через набір функцій.

Методика функціонального програмування також впливає на стиль мислення розробника. Вона пропонує відмову від традиційних циклів на користь рекурсії та функцій перетворення колекцій, таких як map, filter, reduce, які описують кінцеву мету обробки, а не спосіб її досягнення. Це робить код ближчим до математичного опису задачі й водночас значно зменшує можливість помилок, пов’язаних із неправильним контролем індексів, меж масиву або станів змінних.

Узагальнюючи, функціональний стиль програмування є не просто альтернативною технікою структурування програм, а цілісною методологією, яка забезпечує високу модульність, передбачуваність, стійкість до помилок та легкість паралелізації. Він пропонує спосіб опису обчислень, що наближає

програму до математичної моделі й відкриває можливість створення гнучких, масштабованих та надійних систем. Саме тому функціональний стиль набуває дедалі більшої популярності, особливо в середовищах, де важливими є паралельність, реактивність і робота з великими обсягами даних.

2. Генерація, фільтрація та підрахунок даних. Пакет java.util.stream

Пакет java.util.stream у сучасній Java є фундаментальним інструментом для роботи з даними у функціональному стилі. Потоки даних – або Streams – пропонують модель обробки, у якій масиви, списки та інші колекції перестають бути об’єктами покрокового імперативного обходу і перетворюються на джерела даних, що проходять через послідовність операцій-трансформацій. У цьому підході розробник описує не процес, а логіку обробки, зосереджуючись на тому, які перетворення потрібно отримати. Потік дозволяє природно поєднувати генерацію даних, їх фільтрацію, агрегацію та підрахунок, а виконання відбувається лише тоді, коли це дійсно необхідно – у момент виклику кінцевої операції.

Генерація даних у потоках може здійснюватися різними шляхами. Потік може створюватися з колекції, масиву, діапазону чисел, або навіть за допомогою функції-генератора, що формує потенційно нескінченну послідовність. На відміну від традиційного циклу, який ітерує над структурую даних, Stream існує як абстракція, що не зберігає елементи, а лише описує їх джерело і спосіб отримання. Це дозволяє створювати детерміновані конвеєри обробки даних, які можуть бути оптимізовані JVM – аж до автоматичного розпаралелювання, якщо використовується паралельний потік.

Фільтрація є одним із ключових етапів обробки у потоці. Вона дозволяє виокремлювати лише ті елементи, які відповідають певній умові, визначеній функціональним предикатом. Завдяки цьому фільтрація стає природним інструментом для очищення даних, вибору релевантних елементів або побудови складних логічних фільтрів. Особливістю потоків є те, що фільтрація відбувається ліниво: до моменту виклику кінцевої операції

елементи не обробляються, а лише описуються правила їх перетворення. Це не лише підвищує ефективність, але й дозволяє оптимізувати виконання на рівні JVM, уникнувши зайвих обчислень.

Підрахунок даних у потоках реалізується через термінальні операції, серед яких count() є однією з найпростіших та найпоширеніших. Вона повертає кількість елементів, що пройшли всі попередні фільтри й трансформації. На відміну від імперативного підходу, де цикл повинен містити змінну-лічильник і контроль індексів, метод count() абстрагує цей процес повністю, дозволяючи розглядати підрахунок як логічну операцію над даними. У випадку паралельних потоків підрахунок може виконуватися одночасно на кількох ядрах процесора, що робить цю операцію особливо ефективною для великих наборів даних.

Пакет java.util.stream поєднує в собі декларативність функціонального стилю та ефективність оптимізованого виконання. Потоки дозволяють створювати обчислювальні конвеєри з послідовних і паралельних операцій, які автоматично зливаються JVM у мінімальну кількість проходів над даними. Це суттєво підвищує продуктивність та робить код значно компактнішим і зрозумілішим порівняно з імперативним стилем. У таких конвеєрах генерація, фільтрація та підрахунок стають частинами єдиного логічного процесу, який описує суть задачі у високоабстрактній формі.

Завдяки своїм властивостям java.util.stream є ключовим елементом функціонального програмування в Java. Він дозволяє працювати з даними як з математичними послідовностями, легко комбінувати операції, уникати побічних ефектів і масштабувати обчислення. У результаті розробник отримує потужний, гнучкий і виразний інструмент, який ідеально підходить для обробки колекцій, моделювання обчислювальних процесів і побудови сучасних багатопотокових застосунків.

3. Функціональні потоки

Функціональні потоки в Java є продовженням концепції функціонального програмування, перенесеної в контекст обробки даних. У

своїй суті потік (Stream) являє собою абстракцію над послідовністю елементів, яка дозволяє представляти обробку даних як формування ланцюжка функціональних перетворень. На відміну від традиційних циклів, де програміст повинен явно керувати індексами, перевірками меж та проміжними змінними, функціональний потік фокусується виключно на логіці трансформації, не вимагаючи вказівки того, як саме здійснюється ітерація.

Основною властивістю функціональних потоків є їхня декларативність: розробник описує бажаний набір операцій – таких як відбір, перетворення, сортування або агрегація – а виконання цих операцій делегується внутрішньому механізму потоку. Це створює алгоритмічний стиль, який ближчий до математичних описів, ніж до класичних імперативних інструкцій. Завдяки цьому програма стає не лише компактнішою, а й краще структурованою, оскільки логіка обробки даних виражена чітко та прозоро через ланцюжок функцій.

Особливістю функціональних потоків є ліниве виконання. Проміжні операції, такі як map(), filter() або sorted(), не виконують фактичної обробки одразу. Замість цього вони формують план обчислень – своєрідний конвеєр, який активується лише тоді, коли викликається термінальна операція, наприклад collect(), count() або forEach(). Такий підхід дозволяє JVM оптимізувати процес обробки, поєднуючи кілька операцій у єдиний прохід над даними, що суттєво знижує обчислювальні витрати.

Функціональні потоки також підтримують паралельність, надаючи можливість виконувати обробку даних одночасно на кількох ядрах процесора. Перехід від звичайного потоку до паралельного (parallelStream()) відбувається практично миттєво й не потребує внесення змін до логіки алгоритму. Це робить функціональні потоки особливо привабливими для задач, пов’язаних з обробкою великих масивів даних, коли класичні методи стають надто повільними або ускладненими через необхідність ручного керування потоками, блокуваннями чи синхронізацією.

Іще однією важливою властивістю є можливість композиції. Функціональні потоки дозволяють природно поєднувати функції у вигляді довгих ланцюгів, де кожне перетворення є окремим кроком загальної логіки. Це підсилює модульність і дозволяє повторно використовувати функції для різних обчислень. Фактично потік стає формою декларативного програмування, у межах якого алгоритм можна розглядати як послідовність логічних операцій, а не як набір керуючих структур.

Функціональні потоки також позбавлені внутрішнього стану, що робить їх безпечними для використання у паралельних системах. Жодна операція не змінює самих елементів або структури даних, якщо розробник не використовує операції з побічними ефектами. Це дає можливість уникнути конфліктів доступу, некоректних модифікацій і потреби у синхронізації. Замість цього усі перетворення відбуваються у вигляді створення нових елементів, що відповідає принципу незмінності у функціональному програмуванні.

У версії 8 мови програмування Java введено новий пакет java.util.stream, якій містить класи, що можуть бути використані при програмуванні у функціональному стилі.

Функціональні потоки:

– **IntStream** - поток даних типу Integer.

– **LongStream** - поток даних типу Long.

– **DoubleStream** - поток даних типу Double.

Основні методи функціональної обробки даних:

– **generate(IntSupplier s)** - генерація потоку даних за допомогою постачальника s.

– **of(int ... values)** - побудова потоку даних з переліку елементів.

– **iterate(int seed, IntUnaryOperator f**) - ітеративна генерація потоку даних.

– **skip(long n)** - відкидання (пропуск) перших n елементів даних потока.

– **limit(long maxSize)** - обмеження генерації даних у потоці до maxSize елементів.

– **range(int startInclusive, int endExclusive)** - вібір з потоку певної послідовності даних за їх номерами від *startInclusive* до *endExclusive*.

– **parallel()** - розпаралелювання потока даних.

– **sequential()** - об'єднання потоків даних.

– **filter(IntPredicate predicate)** - фільтрація даних згідно зі значенням предиката.

– **sorted()** - сортування потоку даних.

– **count()** - підрахунок кількості даних, що надійшли.

– **min()** - повернення мінімального значення з потоку даних.

– **max()** - повернення максимального значення з потоку даних.

– **sum()** - підрахунок суми.

– **average()** - підрахунок середнього значення.

– **reduce(IntBinaryOperator op)** - об'єднання даних за певним законом або правилом.

– **map(IntUnaryOperator mapper)** - Мапінг, або ремапінг даних - процес, при якому одні дані замінюються іншими за певним законом або правилом.

– **mapToInt(DoubleToIntFunction mapper)**

– **mapToLong(IntToLongFunction mapper)**

– **mapToDouble(IntToDoubleFunction mapper)**

– **mapToObj(IntFunction mapper)**

– **peek(IntConsumer action)** - виконання дії action над кожним елементом потоку. Проміжна обробка, результати видаються у вихідний потік і у подальшому можуть бути оброблені іншим фільтром.

– **forEach(IntConsumer action**) - виконання дії action над кожним елементом потоку. Фінальна обробка.

– **findFirst()** - пошук першоко елемента даних.

– **findAny()** - пошук будь-якого елемента даних.

– **toArray()** - перетворення потоку даних у масив

У підсумку функціональні потоки стають потужним інструментом для створення програм, які є одночасно лаконічними, ефективними та добре структурованими. Вони поєднують у собі ідеї функціонального програмування з можливостями оптимізованої обробки даних, що робить їх ідеальними для сучасних програмних систем, особливо тих, що працюють з великими масивами інформації або потребують високого рівня паралелізації. Саме тому поняття функціональних потоків є невід'ємною частиною сучасної Java і одним із базових інструментів для побудови елегантних і продуктивних алгоритмів.

4. Приклад обрахунку співвідношення об'ємів куба та вписаної у нього сфери методом Монте-Карло

Метод Монте-Карло є одним із найвідоміших статистичних підходів до наближених обчислень, особливо у випадках, коли прямі аналітичні методи є складними або незручними. Його сила полягає в тому, що він використовує випадковість для оцінювання інтегралів, площ або об'ємів, моделюючи ймовірнісний процес, який наближає реальну геометричну чи фізичну задачу. Однією з класичних демонстрацій цього методу є обчислення співвідношення між об'ємом куба та об'ємом сфери, вписаної у цей куб. Така задача є простою геометрично, але водночас чудово ілюструє принципи Монте-Карло й демонструє його збіжність до точного результату за достатньої кількості експериментів.

Розглянемо куб зі стороною довжини a. У цей куб вписується сфера радіусом r = a / 2, центр якої збігається з центром куба. Геометрично очевидно, що об'єм сфери менший за об'єм куба, а їхнє точне співвідношення визначається аналітичними формулами. Проте метод Монте-Карло дозволяє знайти це співвідношення експериментальним шляхом, без використання формул, що є особливо корисним для задач у вищих розмірностях або для складніших геометричних фігур. Суть методу полягає у

випадковому “влучанні” точок у тривимірний простір куба та перевірці, чи належать вони внутрішній області сфери.

Процес обчислення починається з генерації великої кількості випадкових точок, рівномірно розподілених усередині куба. Кожна точка має координати, які випадково вибираються з інтервалу від −r до r, тобто в межах геометричного простору куба. Для кожної точки перевіряється умова належності до сфери: якщо квадрат відстані до центру є меншим або рівним квадрату радіуса, точка вважається такою, що потрапила всередину сфери. Збільшуючи кількість таких експериментальних точок, наближення стає усе точнішим, адже частка точок, що потрапили у сферу, прямує до справжнього співвідношення їхніх об’ємів.

У цьому підході оцінка співвідношення базується на ймовірності того, що випадкова точка належить сфері. Якщо кількість точок, що потрапили всередину сфери, позначити як k, а загальну кількість згенерованих точок як N, то відношення k / N стає наближеним значенням частки об’єму сфери в об’ємі куба. Ця частка, помножена на об’єм куба, дає оцінку об’єму сфери. Хоча метод є апроксимаційним, збільшення N робить результат дедалі ближчим до точного. З математичної точки зору метод Монте-Карло демонструє закон великих чисел: частота попадань випадкових точок у сферу прямує до істинної ймовірності цього випадку.

У застосуванні до задачі про куб і вписану сферу метод Монте-Карло має особливу педагогічну цінність. Він показує, що складні геометричні величини можна оцінювати через прості й інтуїтивно зрозумілі процедури випадкового моделювання. Потік випадкових точок відображає статистичну природу вимірювання, а перевірка належності до сфери перетворює задачу на просту перевірку нерівності між радіусом і відстанню. Такий підхід дає змогу легко узагальнювати метод для багатовимірних інтегралів, областей довільної форми або функцій, що не піддаються аналітичному інтегруванню.

У практичній реалізації метод Монте-Карло не потребує складних обчислень і є надзвичайно ефективним при паралельному виконанні.

Генерація випадкових точок, перевірка умов та підрахунок попадань є незалежними операціями, а тому можуть виконуватися одночасно у кількох потоках. Це робить метод особливо привабливим для реалізації у високопродуктивних системах або в розподіленому середовищі, де обсяг ітерацій може сягати мільйонів або мільярдів. Розпаралелювання дозволяє з мінімальними витратами часу отримувати високоточні результати.

Таким чином, обрахунок співвідношення об’ємів куба та вписаної сфери методом Монте-Карло є класичним прикладом застосування стохастичних методів у чисельному аналізі. Він поєднує інтуїтивну простоту випадкового моделювання та строгість математичної збіжності, демонструючи можливість отримати якісну оцінку геометричних величин навіть у тих випадках, коли формальне інтегрування є складним або недоступним. Метод Монте-Карло, попри свою імовірнісну природу, залишається одним із найефективніших інструментів для наближених розрахунків у задачах геометрії, фізики, статистики та моделювання складних систем.

Список використаних джерел

Основна література

1. Архітектура обчислювальних систем : навчальний посібник до виконання розрахунково-графічної роботи / уклад. В. Г. Артюхов та ін. Київ : КПІ ім. Ігоря Сікорського, 2023. 85 с. URL: https://ela.kpi.ua/server/api/core/bitstreams/f0a58483-7a52-4bd5-9ee5-66831d54c445/content
2. Бородкіна І. Л. Теорія алгоритмів : посібник. Київ : ЦУЛ, 2022. 184 с.
3. Гороховатський В. О., Творошенко І. С. Методи інтелектуального аналізу та оброблення даних : навч. посібник. Харків : ХНУРЕ, 2021. 92 с. URL: https://openarchive.nure.ua/server/api/core/bitstreams/2e55d639-52fd-48d9-b7b7-14989f49f291/content
4. Корочкін О. В., Русанова О. В. Паралельні та розподілені обчислення. Вибрані розділи : навч. посібник. Київ : КПІ ім. Ігоря Сікорського, 2020. 123 с.
5. Коцовський В. М. Теорія паралельних обчислень : навчальний посібник. Ужгород : ПП «АУТДОР-Шарк», 2021. 188 с. URL: http://surl.li/seeca
6. Кузьма К. Т., Мельник О. В. Паралельні та розподілені обчислення : навчальний посібник для вищих закладів освіти. Миколаїв : ФОП Швець В.М., 2020. 172 с. URL: http://surl.li/pnljs
7. Малашонок Г. І., Сідько А. А. Паралельні обчислення на розподіленій пам’яті: OpenMPI, Java, Math Partner : підручник. Київ : НаУКМА, 2020. 266 с.
8. Минайленко Р. М. Паралельні та розподілені обчислення : навч. посіб. Кропивницький : ЦНТУ, 2021. 153 с. URL: https://dspace.kntu.kr.ua/server/api/core/bitstreams/396e02d2-725b-47b5-a1c0-ae07a9bec326/content
9. Паулін О. М. Розподілені обчислення : навчальний посібник. Одеса : Одеська політехніка, 2022. 62 с.
10. Юрчишин В. Я. Проєктування сучасних високопродуктивних обчислювальних систем : навч. посіб. Київ : КПІ ім. Ігоря Сікорського, 2022. 279 с. https://ela.kpi.ua/server/api/core/bitstreams/781c88bf-0f18-4af9-aed2-c92d952e3f01/content
11. Луцків А. М., Лупенко С. А., Пасічник В. В. Паралельні та розподілені обчислення : посібник. Львів : ПП "Магнолія 2006", 2025. 566 с.
12. Бочкарьов О. Ю. Паралельне програмування в ОС Linux : навчальний посібник. Львів : ПП "Магнолія 2006", 2024. 201 с.

13. Минайленко Р. М. Паралельні та розподілені обчислення: навч. посіб. Кропивницький: ЦНТУ, 2021. 153 с.

Додаткова література

1. Argonne National Laboratory, Center for Computational Science and Technology. URL: http://www.mcs.anl.gov

2. Czarnul P. Parallel Programming for Modern High Performance Computing Systems. CRC Press, 2018. 304p.

3. Kurgalin S., Borzunov S. A Practical Approach to High-Performance Computing. Springer, 2019. 206 p

4. Parallel Computing. Architectures, Algorithms and Applications / Bischof C., Bücker M., Gibbon P., Joubert G.R., Lippert T., Mohr B., Peters F.

5. Richard Gerber, Aart J.C. Bik, Kevin B. Smith, and Xinmin Tian The Software Optimization Cookbook, Second Edition.Intel Press, 404p.

6. S. Akhter, J. Roberts. Multi-Core Programming. – Intel Press, 344p.

7. Synchronization of Parallel Programs / Andre J., Herman D., Verjus J.-P. Oxford: North Oxford Academic Publishing Company Limited, 1985. 110 p.

8. Аксак Н. Г. Руденко О. Г., Гуржій А. М. Паралельні та розподілені обчислення: підручник. Харків : Компанія СМІТ, 2009. 480с.

9. Дорошенко А.Ю. Паралельні обчислювальні системи : методичний посібник і конспект лекцій. Київ : Видавничий дім «КМ Академія», 2013. 46 с.

10. Жуков І., Корочкін О. Паралельні та розподілені обчислення : навчальний посібник. Київ : Корнійчук, 2014. 284 с.

11. Кузьменко Б. В., Чайковська О. А. Технологія розподілених систем та паралельних обчислень. Конспект лекцій, частина 1. Розподілені об’єктні системи, паралельні обчислювальні системи та паралельні обчислення, паралельне програмування на основі MPI : навчальний посібник. Київ : Видавничий центр КНУКІМ, 2011. 126 с.

12. Лазарович І. М. Паралельні обчислювальні середовища. Лабораторний практикум. Івано-Франківськ : Видавництво ПНУ імені Василя Стефаника, 2014. 65 с.

13. Організація паралельних обчислень : навчальний посібник / уклад. Є. Ваврук, О. Лашко. Львів : Національний університет “Львівська політехніка”, 2018. 70 с.

14. Паралелізм в Java. URL: https://uk.wikipedia.org/wiki/Паралелізм_в_Java

15. Паралельна обробка і паралелізм в .NET Framework. URL: http://msdn.microsoft.com/ruru/library/hh156548(v=vs.110).aspx

16. Рольщиков В. Б. Технології розподілених систем та паралельних обчислень : конспект лекцій. Одеса : ОДЕКУ 2016.155 с.

17. Сайт Української команди розподілених обчислень. URL: http://distributed.org.ua/.

Навчальне видання

ТЕХНОЛОГІЇ РОЗПОДІЛЕНИХ СИСТЕМ ТА ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

Конспект лекцій

Укладач: **Жебко** Олександр Олегович

Формат 60х84 1/16. Ум. друк. арк. 2.94.
Наклад 50 прим. Зам. № _____

Надруковано у видавничому відділі
Миколаївського національного аграрного університету
54020, м. Миколаїв, вул. Георгія Гонгадзе, 9

Свідоцтво суб'єкта видавничої справи ДК № 4490 від 20.02.2013