Міністерство освіти і науки України

ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ

Конспект лекцій

для здобувачів першого (бакалаврського) рівня вищої освіти

освітньої програми «Комп’ютерна інженерія»

галузь знань 12 (F) Інформаційні технології

спеціальності 123 (F7) Комп’ютерна інженерія

денної та заочної форм навчання

Луцьк 2025

УДК 004.75(07)
П63

Рекомендовано до видання вченою радою факультету КІТ ЛНТУ,
протокол № _____ від «_____» _____ 20___ року.

Голова вченої ради факультету КІТ _____ Інна КОНДІУС

Електронна копія друкованого видання передана для внесення в репозитарій ЛНТУ

Директор бібліотеки _____ Наталія ПОЛІЩУК

Розглянуто і схвалено на засіданні кафедри комп'ютерної інженерії та безпеки ЛНТУ, протокол № _____ від « _____» _____ 20____ року.

Завідувач кафедри КІБ _____ Тарас ТЕРЛЕЦЬКИЙ

Укладач: _____ Катерина МЕЛЬНИК, кандидат технічних наук, доцент кафедри комп'ютерної інженерії та безпеки ЛНТУ

Рецензент: _____ Петро ПЕХ, кандидат технічних наук, доцент кафедри комп'ютерної інженерії та безпеки ЛНТУ

Відповідальний за випуск: _____ Тарас ТЕРЛЕЦЬКИЙ, кандидат технічних наук, доцент кафедри комп'ютерної інженерії та безпеки ЛНТУ

П63 **Паралельні та розподілені обчислення**: конспект лекцій для здобувачів першого (бакалаврського) рівня вищої освіти освітньої програми «Комп'ютерна інженерія» галузь знань 12 (F) Інформаційні технології спеціальності 123 (F7) Комп'ютерна інженерія денної та заочної форм навчання / уклад. К.В. Мельник. Луцьк: ЛНТУ, 2025. 75 с.

Конспект лекцій з дисципліни «**Паралельні та розподілені обчислення**» складений відповідно до діючої програми курсу.

Призначений для здобувачів вищої освіти спеціальності Комп'ютерна інженерія освітньої програми «Комп'ютерна інженерія».

ЗМІСТ

ВСТУП

Навчальна дисципліна «Паралельні та розподілені обчислення» охоплює методи обробки інформації в паралельних та розподілених обчислювальних системах, які є важливими для фахівців з інформаційних технологій. Програма курсу розподілена на два змістовні модулі, що забезпечують поступове поглиблення знань від багатопоточності до розподілених систем.

Змістовий модуль 1 «Паралельне програмування» охоплює основні концепції пов'язані з використанням багатьох обчислювальних ресурсів для одночасного виконання завдань з метою підвищення продуктивності. Студенти вивчають потоки та синхронізацію, застосовування гібридних конструкцій синхронізації, асинхронні моделі, технології TPL та OpenMP. Практична частина включає розробки програм з використанням багатопоточності та вирішенням проблем, що виникають при взаємодії потоків.

Змістовий модуль 2 «Розподілені обчислення» присвячений поглибленому вивченню технології MPI та розподіленому програмуванню в хмарі**.** Практична частина включає дослідження проблеми з масштабністю, взаємодією, різнорідністю, синхронізацією, відмовостійкістю і плануванням, тобто проблеми, які виникають при створенні хмарних програм

Курс спрямований на вивчення основних понять і методів організації паралельної та розподіленої обробки даних та проблем, що виникають при багатопотоковій обробці.

Тема 1. Огляд паралельних обчислень

1.1 Паралельні комп’ютерні архітектури

Архітектурою комп’ютера називають сукупність його властивостей та характеристик, які розглядаються з позицій користувача. Архітектура визначає принципи дії і взаємне поєднання основних пристроїв комп’ютера, систему команд і систему адресації, швидкодію процесора та обсяг пам’яті, програмне забезпечення і засоби інтерфейсу користувача [10-13].

Класична архітектура комп’ютера склалась в кінці 40-х – на початку 50-х років минулого століття. Суттєвий вплив на її становлення спричинили ідеї Джона фон Неймана [3].

Від самого початку комп’ютерної ери існувала необхідність в підвищенні продуктивності роботи обчислювальних машин. В основному це досягалось в результаті еволюції технології виробництва їх елементної бази, що давало можливість збільшувати швидкодію комп’ютера згідно із законом Мура: продуктивність процесорів має збільшуватись вдвічі кожні півтора–два роки. Більше сорока років обчислювальна потужність комп’ютерів дійсно зростала згідно з цим емпіричним законом. Але поступово зростання тактової частоти процесорів зупинилося і, відповідно, припинилось збільшення продуктивності роботи в рамках традиційної, фон-нейманівської, архітектури.

Подальший прогрес обчислювальної техніки став можливим тільки на основі паралельних обчислень, що змушує здійснити перехід на принципово нові комп’ютерні архітектури.

Звичайно, вже давно ведуться дослідження в сфері паралельної обробки, тому, коли у інженерів виникли проблеми зі зростанням продуктивності обчислень, у теоретиків вже був готовий великий вибір нових комп’ютерних архітектур.

Відомо багато різних класифікацій комп'ютерних архітектур [4], найпопулярнішою з них є класифікація Флінна, яка була запропонована ще у 1966 році. Ця класифікація використовує три основних компоненти: процесори, модулі пам'яті та мережу комутаторів, яка з'єднує між собою процесори та пам'ять. В основу роботи таких систем покладено поняття потоків команд та потоків даних, які обробляються процесорами. Залежно від кількості та співвідношення цих потоків можна виділити 4 архітектурних класи:

SISD (Single Instruction & Single Data – один потік команд & один потік даних),

SIMD (Single Instruction & Multiple Data – один потік команд & багато потоків даних),

MISD (Multiple Instruction & Single Data – багато потоків команд & один потік даних),

MIMD (Multiple Instruction & Multiple Data – багато потоків команд & багато потоків даних).

Суть паралельної обробки даних полягає в розподілі всієї обчислювальної роботи на окремі частини і їх одночасному виконанні, що в підсумку має дати виграш в часі виконання всієї роботи. Використовуючи математичну термінологію, кажуть про декомпозицію початкової обчислювальної задачі. Відомі такі способи декомпозиції:

– за даними,

– за функціями (підзадачами),

– за часом.

Відповідно можна розрізняти паралелізм за даними, паралелізм за функціями (підзадачами) та паралелізм за часом.

1.2 Процеси та потоки

Після розробки паралельного алгоритму розв'язання поставленої задачі необхідно записати у формі, яка сприймається комп'ютером [10-13]. Таку можливість забезпечують алгоритмічні мови програмування двох типів:

– спеціалізовані мови паралельного програмування,

– стандартні мови високого рівня, доповнені операторами длярозпаралелювання обчислень.

На практиці найчастіше використовуються мови програмування другого типу, їх і будемо в подальшому використовувати, зокрема, мови C++ та C#.

Після запису алгоритму за допомогою вибраної мови програмування буде отримано комп'ютерну програму, яка зберігається у вигляді файлів на деякому носії даних (зазвичай на магнітному чи оптичному диску). Далі виконується компіляція програми (наприклад, за допомогою пакета Microsoft Visual Studio) для отримання машинного коду програми. Програма в машинних кодах також зберігається у файлах на дисках, але іншого формату (найчастіше типу *.exe для операційної системи Windows).

На цьому завершується етап підготовки задачі до виконання. Далі програміст ініціює початок виконання комп'ютерної програми. Для пояснення всіх подальший дій будемо використовувати терміни «процес» і «потік» [10, 11].

Якщо програма – це статична послідовність команд і операторів, то процес – це програма на стадії її виконання, тобто в динаміці. Кожний раз після запуску на виконання файлу типу *.exe формується новий процес з окремим захищеним адресним простором в 4 Гбайт. Не тільки для різних, але і для однієї і тієї ж програми, викликаної кілька разів, створюється новий процес.

Кожний процес має набір атрибутів, зокрема:

– ідентифікатор процесу,

– базовий пріоритет,

– системні ресурси (квоти на машинний час, обсяг системної пам'яті та інші),

– інструменти міжпроцесної взаємодії,

– файли та системні бібліотеки.

Кожний блок процесу є представником цього процесу у Windows, в якому містяться всі атрибути процесу і покажчики на деякі структури даних.

В цілому процес можна розглядати як контейнер для всіх видів ресурсів, окрім одного – процесорного часу. Цей найважливіший ресурс розподіляється операційною системою між потоками.

Потік – це послідовність команд програми, які виконуються в процесорі протягом одного кванту процесорного часу (20 мс у Windows). Кожний процес починається з одного потоку – головного. Потік знаходиться в адресному просторі процесу, має пріоритет в межах базового пріоритету процесу і використовує його ресурси. Якщо в межах процесу динамічно створюються нові потоки, тоді всі ресурси процесу розподіляються між потоками цього процесу. Пріоритети потоків періодично змінюються, і тоді потоки з більшим пріоритетом витісняють потоки з меншим пріоритетом. Подібно процесу потік має свій набір атрибутів.

Процесор за допомогою апаратного таймера визначає момент закінчення чергового кванта, який був виділений даному потоку, і формує переривання. Далі процесор переміщає в структуру даних CONTEXT вміст всіх регістрів. Коли цей потік знову отримає квант машинного часу процесор відновить значення регістрів із його структури CONTEXT. Вся ця операція називається перемиканням контексту потоку.

Процеси і потоки мають ряд принципових відмінностей.

По-перше, потоки для свого створення, функціонування та ліквідації потребують набагато менше системних витрат, ніж процеси. Перемикання процесора з виконання одного потоку на виконання іншого потоку зводиться до однієї операції перемикання контексту: завантаження в регістри процесора нових значень.

По-друге, процеси «бачать» виділені їм пам’ять та інші ресурси, нічого не «знаючи» про існування інших процесів. Потоки одного процесу не тільки «знають» про існування один одного, але і конкурують між собою за спільні системні ресурси, використовуючи для цього різноманітні засоби синхронізації.

По-третє, дуже складно організувати обмін даними між процесами. Оскільки безпосередній обмін даними між ними заборонений, то для цього

необхідно використовувати спеціальні засоби взаємодії: черги, конвеєри та ін. Потоки можуть обмінюватись даними набагато швидше.

В перших операційних системах було реалізовано однозадачний режим роботи, тобто в один момент часу міг бути лише один процес з одним потоком. Тому навіть не розрізняли між собою процеси і потоки.

В сучасних багатозадачних операційних системах в комп’ютері можуть одночасно існувати сотні процесів та потоків. Відповідно системі потрібно паралельно організувати їх спільну роботу.

Розрізняють два різновиди багатозадачності: на основі процесів і на основі потоків. У зв'язку з цим важливо розуміти відмінності між ними.

Процес фактично являє собою виконувану програму. Тому багатозадачність на основі процесів – це засіб, завдяки якому накомп'ютері можуть паралельно виконуватися дві програми і більш.

Потік являє собою координовану одиницю виконуваного коду. Своїм походженням цей термін зобов'язаний поняттю «потік виконання». при організації багатозадачності на основі потоків у кожного процесу повинен бути принаймні один потік, хоча їх може бути і більше. Це означає, що в одній програмі одночасно можуть вирішуватися два завдання і більше.

Відмінності в багатозадачності на основі процесів і потоків можуть бути зведені до наступного: багатозадачність на основі процесів організується для паралельного виконання програм, а багатозадачність на основі потоків для паралельного виконання окремих частин однієї програми. Потік може перебувати в одному з декількох станів. В цілому, потік може бути що виконуються; готовим до виконання, як тільки він отримає час і ресурси ЦП; припиненим, тобто тимчасово що виконуються; відновленим в подальшому; заблокованим в очікуванні ресурсів для свого виконання; а також завершенням, коли його виконання закінчено і не може бути відновлено.

В середовищі .NET Framework визначені два різновиди потоків: пріоритетний і фоновий. За замовчуванням створюється потік автоматично стає пріоритетним, але його можна зробити фоновим. Єдина відмінність

пріоритетних потоків відфонових полягає в тому, що фоновий потік автоматично завершується, якщо в його процесі зупинені всі пріоритетні потоки.

1.3 Парадигми паралельного програмування

Не зважаючи на існування великої кількості паралельних програмних комплексів, в них використовується лише невелика кількість патернів або парадигм паралельного програмування. В роботі [10] виокремлено п’ять основних парадигм:

1) ітеративний паралелізм;

2) рекурсивний паралелізм;

3) «виробники та споживачі» (конвеєри);

4) «клієнти та сервери»;

5) взаємодіючі рівні (interacting peers).

З використанням однієї або кількох із цих парадигм і програмуються додатки. 5

Ітеративний паралелізм використовується, коли у програмі є кілька процесів (часто ідентичних), кожний із яких містить один або кілька циклів. Таким чином, кожний процес є ітеративною програмою. Процеси програми працюють сумісно над однією задачею; вони взаємодіють та синхронізуються з допомогою спільних змінних або передачі повідомлень. Ітеративний паралелізм частіше за все зустрічається в наукових обчисленнях, які виконуються на кількох процесорах.

Рекурсивний паралелізм може використовуватися у випадку наявності у програмі однієї або кількох рекурсивних процедур, виклики яких незалежні, тобто кожний із них опрацьовує свою частини загальних даних. Рекурсія часто застосовується у імперативних мовах програмування, особливо при реалізації алгоритмів типу «поділяй та пануй» або «перебір з поверненням» (backtracking). Рекурсія є однією з фундаментальних парадигм також і в символьних, логічних, та функціональних мовах програмування. Рекурсивний паралелізм

використовується для розв’язування таких комбінаторних проблем, як сортування, планування (задача комівояжера) та ігри (шахи та інші).

Виробники та споживачі – це взаємодіючі процеси. Вони часто організовані у конвеєр, через який проходить інформація. Кожний процес конвеєра є фільтром, який споживає вихідні дані свого попередника та продукує вхідні дані для свого користувача. Фільтри зустрічаються на рівні додатків в операційних системах типа ОС Unix, всередині самих операційних систем, прикладних програм, якщо один процес виробляє вихідні дані, які споживає (читає) інший процес.

Клієнти та сервери – найбільш поширена модель взаємодії в розподілених системах, від локальних мереж до World Wide Web. Клієнтський процес робить запит до сервісу та очікує відповіді. Сервер очікує запити від клієнтів, а потім діє відповідно до цих запитів. Сервер може бути реалізований у вигляді одиночного процесу, який не може обробляти одночасно кілька запитів клієнтів, або (при необхідності паралельного обслуговування запитів) як багатопотокова програма. Клієнти та сервери – паралельне програмне узагальнення процедур та їх викликів: сервер виконує роль процедури, а клієнт її викликає. Але якщо код клієнта та код сервера розташовані на різних машинах, звичайний механізм виклику процедур недоступний. Замість цього необхідно використовувати віддалений виклик процедури або рандеву.

Взаємодіючі рівні – остання парадигма взаємодії. Вона зустрічається в розподілених програмах, в яких кілька процесів для розв’язування задачі виконують один і той самий код та обмінюються повідомленнями. Взаємодіючі рівні використовуються для реалізації розподілених паралельних програм, особливо при ітеративному паралелізмі та децентралізованому прийняті рішень в розподілених системах.

1.3.1 Ітеративний паралелізм: множення матриць

Ітеративна послідовна програма використовує для обробки даних та обчислення результатів цикли типу for та while. Ітеративна паралельна програма містить кілька ітеративних процесів. Кожний процес обчислює результати для підмножини даних, а потім ці результати збираються разом.

В якості прикладу розглянемо задачу із галузі наукових обчислень. Припустимо, що задані дві -матриці a та b. Мета — обчислити c=a*b.

Матриці a, b, c є спільними глобальними змінними, індекси рядків та стовпців змінюються від 0 до n-1, елементи матриці є дійсними числами.

Добуток матриць можна знайти з використанням послідовної програми, яка має наступний псевдокод:

```
for [i = 0 to n-1]
 {
        for [j = 0 to n-1]
        {
                c[i,j] = 0;
                for [к = 0 to n-1]
                c[i,j] = c[i,j] + a[i,k]*b[k,j];
 }
    }
```

Множення матриць – приклад задачі з масовим паралелізмом, оскільки програма містить велику кількість операцій, які можуть виконуватися паралельно. Дві операції можуть виконуватися паралельно, якщо вони незалежні. Припустимо, що множина зчитування операції містить змінні, які вона читає, але не змінює, а множина запису – змінні, які вона змінює (і, можливо, зчитує). Дві операції є незалежними, якщо їх множини запису не перетинаються. Неформально кажучи, процеси завжди можуть безпечно читати змінні, які не змінюються. Однак двом процесам взагалі кажучи небезпечно змінювати значення однієї і тієї самої змінної або одному процесу зчитувати змінну, яка записується іншим процесом.

При обчислені добутку матриць обчислення проміжкових добутків є незалежними операціями. У рядках 3–5 попередньої програми виконується ініціалізація та обчислення елемента матриці c. Внутрішній цикл програми зчитує рядок матриці a та стовпець матриці b, а потім зчитує та записує один елемент матриці c. Множина зчитування для внутрішнього добутку — це рядок матриці a та стовпець матриці b, множина запису — елемент матриці c.

Оскільки множини запису внутрішніх добутків не перетинаються, їх можна виконувати паралельно. Можливими є варіанти, коли паралельно

обчислюються результуючі рядки, результуючі стовпці чи групи рядків та стовпців.

Для початку розглянемо паралельне обчислення рядків матриці c.

```
co [i = 0 to n-1]
 { # паралельне обчислення рядків
        for [j = 0 to n-1]
        {
        c[i,j] = 0;
        for [k = 0 to n-1]
        c[i,j] = c[i,j] + a[i,k]*b[k,j];
        }
}
```

Ця програма відрізняється від послідовного варіанта лише тим, що у зовнішньому циклі оператор for замінений оператором co. Але семантична різниця велика: оператор co вказує, що його тіло для кожного значення індексу i буде виконуватися паралельно (принаймні теоретично, в залежності від кількості наявних процесорів).

Другий спосіб паралельного множення матриць — паралельне обчислення стовпців матриці c:

```
co [j = 0 to n-1]
{ # паралельне обчислення стовпців
        for [i = 0 to n-1] {
        c[i,j] = 0;
        for [k = 0 to n-1]
        c[i,j] = c[i,j] + a[i,k]*b[k,j];
        }
}
```

У цій версії два зовнішні цикли (за i та за j) помінялися місцями (якщо тіла двох циклів незалежні, то їх можна безпечно міняти місцями).

Дві отримані версії програми по суті мають ідентичні обчислювальні властивості. Проте у випадку множення m×n та n×p прямокутних матриць числа m та p можуть сильно відрізнятися, а тому з огляду на параметри обчислювальної системи обчислення з паралелізмом за рядками у випадку m<p можуть виявитися значно ефективнішими, ніж обчислення з паралелізмом за стовпцями.

Програму для паралельного обчислення усіх проміжних добутків можна кількома способами. Можна використати один оператор co для двох індексів.

```
co [i = 0 to n-1, j = 0 to n-1]
```

```
{ # усі рядки та усі стовпці
        c[i,j] = 0;
        for [k = 0 to n-1]
        c[i,j] = c[i,j] + a[i,k]*b[k,j];
}
```

Тіло оператора co виконується паралельно для кожної комбінації значень індексів i та j, тому програма визначає n^2 процесів. (Чи будуть вони всі виконуватися паралельно, залежить від конкретної реалізації). Другий спосіб паралельного обчислення проміжних добутків полягає у використанні вкладених операторів co.

```
co [i = 0 to n-1] { # рядки паралельно, потім
    co [j = 0 to n-1] { # стовпці паралельно
        c[i,j] = 0;
        for [k = 0 to n-1]
            c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
}
```

Для кожного рядка (зовнішній оператор co) і потім для кожного стовпця (внутрішній оператор co) задається по одному процесу. Третій спосіб написати цю програму – поміняти місцями два рядки останньої програми. Результат усіх трьох програм однаковий: виконання внутрішнього циклу для усі n^2 комбінацій значень i та j. Різниця між ними – у визначенні процесів, а отже, і у часі їх створення.

Слід зазначити, що усі попередні паралельні програми були отримані заміною оператора for на co, причому це було зроблено тільки для індексів i та j. Виникає питання, як бути з внутрішнім циклом за індексом k? Чи можна цей оператор замінить оператором co? Відповідь – «ні», оскільки тіло внутрішнього циклу як зчитує, так і змінює значення змінної c[i, j]. Можна обчислити суму у циклі for за змінною k, використовуючи каскадну схему здвоєння, але як зазначено в [10], це не може забезпечити суттєвого прискорення для більшості машин.

Інший спосіб розпаралелити обчислення – використати ключове слово process замість оператора co (тобто використати декларацію процесу). По суті, process — це оператор co, який виконується у фоновому режимі. Наприклад, перша паралельна програма з наведених вище (з паралелізмом по рядкам результату) може бути записана наступним чином:

```
process row[i = 0 to n-1] { # рядки паралельно
       for [j = 0 to n-1] {
       c[i,j] = 0;
       for [k = 0 to n-1]
          c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
}
```

В програмі визначений масив процесів – row[1], row[2] і т.д. – по одному для кожного значення індексу i. Ці n процесів створюються і починають виконуватися, коли зустрічається рядок декларації процесу. Якщо за декларацією процесу йдуть оператори, то вони виконуються паралельно з процесом, тоді як оператори, записані після оператора co, не виконуються до його завершення.

В програмах, наведених вище, для кожного елементу, рядка або стовпця результуючої матриці використано по одному процесу. Припустимо, що число процесорів в системі менше за n (так зазвичай і буває, коли n велике). У цьому випадку є очевидний спосіб повного використання усіх процесорів: поділити матрицю на смуги (рядків чи стовпців) і для кожної смуги створити робочий процес, який обчислює результати для елементів своєї смуги. Припустимо, що є P процесорів і n кратне P. Тоді у випадку смуг рядків робочі процеси можна запрограмувати так:

```
process worker[w = 1 to P] {
    int first = (w-1) * n/P;     # перший рядок смуги
    int last = first + n/P - 1; # останній рядок смуги
    for [i = first to last] {
        for [j = 0 to n-1] {
            c[i,j] = 0;
            for [k = 0 to n-1]
                c[i,j] = c[i,j] + a[i,k]*b[k,j];
        }
    }
}
```

В програму додані оператори, необхідні для визначення першого та останнього рядка кожної смуги. Потім рядки смуги вказуються у циклі (за індексом i) для обчислення елементів матриці c.

Таким чином, суттєвою умовою розпаралелювання програм є наявність незалежних обчислень, тобто обчислень з множинами запису, що не перетинаються. Для добутку матриць незалежними обчисленнями є скалярні добутки рядків на стовпці, оскільки кожний із них записує (та читає) свій елемент

c[i,j]. Тому можна паралельно обчислювати усі скалярні добутки, рядки, стовпці або смуги з використанням оператора co або декларації процесу.

1.3.2 Рекурсивний паралелізм: адаптивна квадратура

Програма є рекурсивною, якщо вона містить процедури, які викликають самі себе — прямо або опосередковано. Рекурсія дуальна до ітерації, тобто рекурсивні програми можна перетворити у ітеративні і навпаки. У тілі багатьох рекурсивних процедур звертання до самої себе зустрічається більше одного разу. Наприклад, алгоритм quicksort розбиває масив на дві частини, а потім двічі викликає себе для окремого сортування лівої та правої частин. Значна кількість алгоритмів обробки дерев та графів мають подібну структуру.

Рекурсивну програму можна реалізувати за допомогою паралелізму, якщо вона містить кілька незалежних рекурсивних викликів. Два виклики процедури (або функції) є незалежними, якщо їх множини запису не перетинаються. Ця умова виконується, якщо:

1) процедура не звертається до глобальних змінних або тільки читає їх;

2) аргументи и результуючі змінні процедур — різні змінні.

Наприклад, якщо процедура не звертається до глобальних змінних і має тільки параметри-значення (за механізмом їх передачі), то будь-який її виклик буде незалежним. (процедура читає та змінює тільки локальні змінні, і кожний екземпляр процедури має свою локальну копію змінних).

Розглянемо задачу квадратури, яка полягає у наближеному обчисленні інтеграла неперервної функції. Припустимо, що це функція f (x). Як показано на рисунку 1.1, інтеграл функції f (x) від a до b – це площа фігури, обмеженої графіком f (x), віссю абсцис та прямими x = a і x = b.

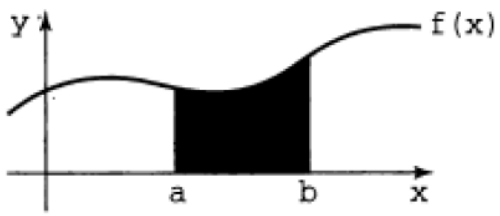


Рисунок 1.1 – Задача квадратури

Існує два основних способи апроксимації значення інтеграла. Перший – розбити інтервал від a до b на фіксоване число відрізків, а потім апроксимувати площу на кожному з них за правилом трапецій або за правилом Сімпсона.

```
double fleft = f(a), fright, area = 0.
double width = (b-a) / INTERVALS;
for [x = (a + width) to b by width]{
    fright = f(x);
    area = area + (fleft + fright) * width / 2;
    fleft = fright;
}
```

Кожна ітерація обчислює площа малої фігури за правилом трапецій і додає її до загального значення площі. Змінна width – ширина кожної трапеції, відрізки перебираються зліва направо, тому праве значення кожної ітерації стає лівим значенням наступної ітерації.

Другий спосіб апроксимації інтеграла – використовувати парадигму «поділяй і пануй» і змінне число інтервалів. Зокрема, спочатку обчислюють значення m – середину відрізка[a, b]. Потім апроксимують площу трьох областей під кривою f(x): від a до m, від m до b та від a до b. Якщо сума менших площ дорівнює більшій площі з деякою заданою точністю ε, то апроксимацію можна вважати достатньою [10]. Якщо ні, то задача ділиться на дві підзадачі: від a до m та від m до b, і процес повторюється. Цей спосіб називається адаптивною квадратурою, оскільки алгоритм «адаптується» до форми кривої. Його можна запрограмувати так.

```
double quad(double left,right,fleft,fright,lrarea){
    double mid = (left + right) / 2;
    double fmid = f(mid);
    double larea = (fleft+fmid) * (mid-left) / 2;
    double rarea = (fmid+fright) * (right-mid) / 2;
    if(abs((larea+rarea) - lrarea) > EPSILON){
        larea = quad(left, mid, fleft, fmid, larea);
        rarea = quad(mid, right, fmid, fright, rarea);
    }
    return (larea + rarea);
}
```

Інтеграл функції f (x) від a до b апроксимується таким викликом функції:

```
area = quad (a, b, f (a), f (b), (f (a) + f (b)) * (b-a) /
2);
```

У функції знову використовується правило трапеції. Значення функції f у крайніх точках відрізка і наближена площа цього інтервалу передаються в кожен виклик функції quad, щоб не обчислювати їх більше одного разу.

Ітеративну програму не можна розпаралелити (у наведеній формі), оскільки тіло циклу і зчитує, і записує значення змінної area. Проте в рекурсивної програмі виклики функції quad незалежні за умови, що обчислення функції f(x) не дає побічних ефектів. Зокрема, аргументи функції quad передаються за значенням, і в її тілі немає присвоювання глобальних змінних. Таким чином, для завдання паралельного виконання рекурсивних викликів функції можна використовувати оператор co.

```
co larea = quad (left, mid, fleft, fmid, larea);
// rarea = quad (mid, right, fmid, fright, rarea);
oc
```

Це єдина зміна, необхідне для того, щоб зробити цю програму паралельною. Оскільки оператор co не закінчується до тих пір, поки не будуть завершені обидва виклику функцій, значення змінних larea і rarea обчислюються до того, як функція quad поверне їх суму.

В операторах co програм множення матриць містяться списки інструкцій, які виконуються для кожного значення лічильників (i та j). В операторі co, наведеному вище, містяться два виклики функцій, відокремлених знаками //. Перша форма оператора co використовується для вираження ітеративного паралелізму, друга – рекурсивного.

Отже, програму з декількома рекурсивними викликами функцій можна легко перетворити в паралельну рекурсивну програму, якщо виклики незалежні. Проте існує чисто практична проблема: паралельно виконуваних операцій може стати занадто багато. Кожен оператор co в наведеній вище програмі створює два процеси, по одному для кожного виклику функції. Якщо глибина рекурсії велика, то виникне велика кількість процесів, можливо, занадто велика для паралельного виконання.

Вирішення цієї проблеми полягає в скороченні, або відтинанні, дерева рекурсії при занадто великій кількості процесів.

1.3.3 Виробники і споживачі: канали ОС Unix

Процес-виробник виконує обчислення і виводить потік результатів. Процес-споживач вводить і аналізує потік значень. Багато програм в тій чи іншій формі є виробниками та / або споживачами. Поєднання стає особливо цікавим,

якщо виробники і споживачі об'єднані в конвеєр – послідовність процесів, в якій кожен з них споживає дані виходу попередника і виробляє дані для подальшого процесу. Класичним прикладом є конвеєри в операційній системі Unix. Однією з найбільш потужних функцій, запропонованих в ОС Unix, була можливість прив’язки стандартних "пристроїв" введення-виведення до різних типів файлів. Зокрема, файли stdin або stdout можуть бути пов’язані з файлом даних або з "файлом" особливого типу, який називається каналом.

Канал – це буфер (черга типу FIFO) між процесом-виробником і процесом-спо-живачем, який містить зв’язану послідовність символів, до якої виробник може дописувати нові символи. Символи видаляються, коли процес-споживач зчитує їх з каналу. Процес-виробник очікує (при необхідності), поки в буфері з’явиться вільне місце, потім додає в кінець буфера новий рядок. Процес-споживач очікує (при необхідності), поки в буфері не з’явиться рядок даних, потім вибирає її з буфера. Такі буфери реалізовують з використанням спільних змінних (shared variables) і різних примітивів синхронізації (прапорів, семафорів та моніторів).

1.3.4 Клієнти і сервери: файлові системи

Між виробником і споживачем існує односпрямований потік інформації. Цей вид взаємодії між процесами часто зустрічається в паралельних програмах і не має аналогів в послідовних, оскільки в послідовній програмі тільки один потік управління, тоді як виробники і споживачі — незалежні процеси з власними потоками управління і власними швидкостями виконання.

Ще однією типовою схемою в паралельних програмах є взаємозв’язок типу клієнт-сервер. Процес-клієнт запитує сервіс, потім очікує обробки запиту. Процес-сервер багаторазово очікує запит, обробляє його, потім посилає відповідь. Як показано на рисунку 1.2, існує двонапрямлений потік інформації: від клієнта до сервера і назад.

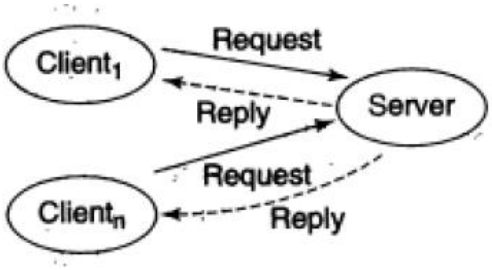


Рисунок 1.2 – Клієнти та сервери

Зв’язки між клієнтом і сервером в паралельному програмуванні аналогічні відносинам між програмою, що викликає підпрограму, і самої підпрограмою в послідовному програмуванні. Більш того, як підпрограма може бути викликана з кількох місць програми, так і у сервера зазвичай є багато клієнтів. Запити кожного клієнта повинні оброблятися незалежно, проте паралельно може оброблятися декілька запитів, подібно до того, як одночасно можуть бути активні кілька викликів однієї і тієї ж процедури.

Взаємодія типу клієнт-сервер зустрічається в операційних системах, об’єктно - орієнтованих системах, мережах, базах даних і багатьох інших програмах. Типовий приклад – читання і запис файлу. Для визначеності припустимо, що є модуль файлового сервера, що забезпечує дві операції з файлом: read (читати) і write (писати). Коли процес-клієнт хоче отримати доступ до файлу, він викликає операцію читання або запису у відповідному модулі файлового сервера.

На однопроцесорній машині або в іншій системі з пам’яттю файловий сервер зазвичай реалізується набором підпрограм (для операцій read, write і т.д.) і структурами даних, що зображають файли (наприклад, дескрипторами файлів). Отже, взаємодія між процесом-клієнтом і файлом зазвичай реалізується викликом відповідної процедури. Однак, якщо файл розділяється, важливо, щоб запис в нього вівся одночасно тільки одним процесом, а читатися він може одночасно кількома. Цей різновид задачі – приклад так званої задачі про "читачів і письменників", класичної задачі паралельного програмування.

1.3.5 Взаємодіючі рівні: розподілене множення матриць

Розглянемо два способи вирішення цього задачі з використанням процесів, взаємодіючих за допомогою пересилання повідомлень. Перша програма використовує керуючий процес і масив незалежних робочих процесів. У другій програмі робочі процеси рівні і їх взаємодія забезпечується круговим конвеєром. Мал. 4.3 ілюструє схему взаємодії цих процесів.

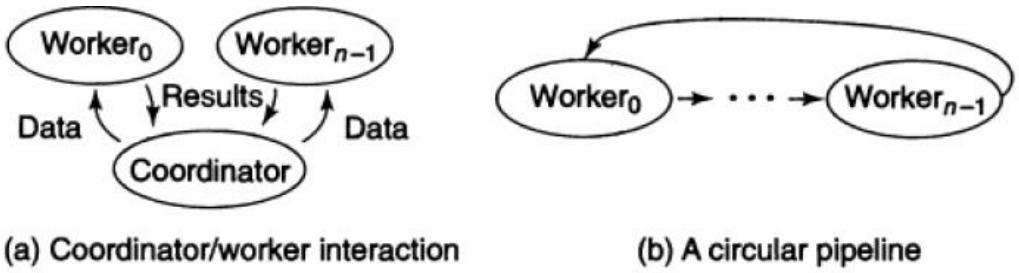


Рисунок 1.3 – Множення матриць з використанням передачі повідомлень

На машинах з розподіленою пам’яттю кожен процесор має доступ тільки до власної локальної пам’яті. Це означає, що програма не може використовувати глобальні змінні, тому будь-яка змінна повинна бути локальною для деякого процесу і може бути доступною тільки цього процесу або процедури. Отже, для взаємодії процеси повинні використовувати передачу повідомлень.

Припустимо, що нам необхідно знайти добуток квадратних матриць a і b, а результат помістити в матрицю c. Припустимо, що у системі є n процесорів. Можна використовувати масив з n робочих процесів, помістивши по одному на кожен процесор і змусивши кожен робочий процес обчислювати один рядок результуючої матриці c. Програма для робочих процесів буде виглядати наступним чином

```
process worker [i = 0 to n-1] {
    receive початкові значення вектора a і матриці b;
        # вектори a, c — i-ті рядки відповідних матриць
        for [j = 0 to n-1] {
            c[j] = 0;
            for [k = 0 to n-1]
                c[j] = c[j] + a[k] * b[k,j];
        }
        send вектор-результат c керуючому процесу;
    }
```

Робочий процес і обчислює і-й рядок результуючої матриці c. Щоб це зробити, він повинен отримати рядок і вихідної матриці a і всю вихідну матрицю b. Кожен робочий процес спочатку отримує ці значення від окремого керуючого процесу. Потім робочий процес обчислює свій рядок результатів і відсилає її назад керуючому.

Керуючий процес ініціює обчислення, збирає і виводить їх результати. Зокрема, спочатку керуючий процес посилає кожному робочому відповідну рядок матриці a і всю матрицю b. Потім керуючий процес очікує отримання рядків матриці від кожного робочого процесу. Схема керуючого процесу така.

```
process coordinator {
    double a[n, n]; # Вихідна матриця a
    double b[n, n]; # Вихідна матриця b
    double c[n, n]; # Результуюча матриця c
    # ініціалізувати a та b;
    for [i = 0 to n-1] {
        send рядок i матриці a процесу worker[i];
        send всю матрицю b процесу worker[i];
    }
    for [i = 0 to n-1]
        receive рядок i матриці c від процесу worker[i];
    вивести результат, який тепер в матриці c;
}
```

Оператори send та receive — це примітиви передачі повідомлень. Операція send надсилає повідомлення іншому процесу; операція receive чекає повідомлення від іншого процесу, а потім зберігає його в локальних змінних.

Припустимо, що кожен робочий процес отримує один рядок матриці a і повинен обчислити один рядок матриці c. Однак тепер припустимо, що у кожного процесу є тільки один стовпець, а не вся матриця b. Отже, в початковому стані робочий процес і має стовпець і матриці b. Маючи лише ці вихідні дані, робочий процес може обчислити тільки значення c[i,i]. Для того щоб робочий процес і міг вирахувати всю рядок матриці c, він повинен отримати всі стовпці матриці b. Для цього можна використовувати кругової конвеєр (див. рис. 1.3). Кожен робочий процес виконує послідовність раундів; в кожному раунді він відсилає свій стовпець матриці b наступному процесу і отримує інший її стовпець від попереднього. Програма має наступний вигляд:

```
process worker [i = 0 to n-1] {
    double a[n]; # Рядок i матриці a
    double b[n]; # Один стовпець матриці b
```

```
    double c[n]; # Рядок і матриці c
    double sum = 0; # Для проміжних добутків
    int nextCol = i; # Наступний стовпець результатів
    receive рядок і матриці a і стовпець і матриці b;
    # Обчислити c[i, i] = a[i,*] b[*,i]
    for [k = 0 to n-1]
        sum = sum + a[k] * b[k];
    c[nextCol] = sum;
    ((Пустити по колу стовпчики та обчислити інші c[i,*]
    for [j = 1 to n-1] {
        send мій стовпець матриці b наступного процесу;
        receive новий стовпець матриці b від попереднього;
        sum = 0;
        for [k = 0 to n-1]
            sum = sum + a[k] * b[k];
        if (nextCol == 0)
            nextCol = n-1;
        else
            nextCol = nextCol-1;
        c[nextCol] = sum;
    }
    send вектор-результат c керуючому процесу;
}
```

У програмі робочі процеси впорядковані відповідно до їх індексів. (Для процесу n − 1 наступним є процес 0, а попереднім для 0 про – n − 1.) Стовпці матриці b передаються по колу між робочими процесами, тому кожен процес врешті-решт отримає кожен стовпець. Змінна nextCol відстежує, куди у векторі c помістити черговий проміжний добуток. Як і в першому обчисленні, передбачається, що керуючий процес надсилає рядки матриці a і стовпці матриці b робочим процесам, а потім отримує від них рядки матриці c.

У попередній програмі використано відношення між процесорами, яке називається взаємодіючі рівні (interacting peers). Кожен робочий процес виконує один і той же алгоритм і взаємодіє з іншими робочими процесами, щоб обчислити свою частину необхідного результату.

У першій з наведених вище програм значення з матриці b дублюються в кожному робочому процесі. У другій програмі в будь-який момент часу у кожного процесу є один рядок матриці a і тільки один стовпець матриці b. Це знижує витрати пам’яті для кожного процесу, але друга програма виконується довше першої, оскільки на кожній її ітерації кожен робочий процес повинен відіслати повідомлення одному сусідові і отримати повідомлення від іншого.

Дані програми ілюструють класичне протиріччя між часом і простором в обчисленнях.

Контрольні питання

1. Для чого потрібний паралелізм. Задача критичної секції.
2. Класифікація обчислювальних систем. Аналіз ефективності паралельних обчислень.
3. Архітектура однопроцесорної машини. Мультікомп’ютери з розподіленою пам'яттю. кластер
4. Мультіпроцесор з розподіленою пам'яттю. Задача критичної секції. Рівні паралелізму в багатоядерній архітектурі.
5. Аналіз ефективності паралельних обчислень. Межі паралелізму. «Закон Амдала».
6. Проблеми розробки паралельних програм. Декомпозиція

Тема 2. Процеси та синхронізація

2.1 Процеси та синхронізація

У послідовних програмах часто використовуються колективні змінні, наприклад, для зберігання глобальних структур даних, але вважається, що краще обходитися без них. Разом з тим, паралельні програми цілком залежать від спільних компонентів, оскільки процеси можуть працювати над однією задачею, тільки взаємодіючи. А єдиний спосіб взаємодії – можливість для одного процесу записувати в щось, звідки інший процес читає. Цим чимось може бути колективна змінна або канал зв’язку. Тому взаємодія програмується як запис і читання спільних змінних або як передача і прийом повідомлень [15].

Взаємодія підвищує необхідність синхронізації. Існує два основних її типу: взаємне виключення і умовна синхронізація. Взаємне виключення зустрічається, коли два процеси повинні по черзі звертатися до таких спільних об’єктів, як записи в системі замовлення авіаквитків. Синхронізація умов відбувається, коли одному процесу доводиться чекати інший процес, наприклад, коли процес-споживач очікує дані від процесу-виробника.

В основу синхронізації покладено поняття блокування, за допомогою якої організовується управління доступом до кодовому блоку в об'єкті, коли об'єкт заблокований одним потоком, інші потоки не можуть отримати доступ до заблокованого кодовому блоку. Коли ж блокування знімається одним потоком, об'єкт стає доступним для використання в іншому потоці.

Підсумки використання блокування:

– Якщо блокування будь-якого заданого об'єкта отримана в одному потоці, то після блокування об'єкта вона не може бути отримана в іншому потоці.

– Іншим потокам, які намагаються отримати блокування того ж самого об'єкта, доведеться чекати до тих пір, поки об'єкт не виявиться в розблокованому стані.

– Коли потік виходить із заблокованого фрагмента коду, відповідний об'єкт розблокується.

2.2 Стан, дії, історія та властивості

Стан паралельної програми складається зі значень змінних програми в деякий момент часу. Змінні можуть бути описані явно певними програмістом або неявними (на кшталт програмного лічильника кожного процесу), що зберігають приховану інформацію про стан. Паралельна програма починає виконання в деякому вихідному стані. Кожен процес програми виконується незалежно, і в міру виконання він перевіряє і змінює стан програми.

Процес виконує послідовність операторів. Оператор, в свою чергу, реалізується послідовністю неподільних дій. Ці дії перевіряють чи змінюють стан програми неподільним чином. Прикладами неподільних дій є неперервні машинні інструкції, які завантажують і зберігають слова пам’яті.

Здійснення паралельної програми призводить до чергування послідовностей неподільних дій, вироблених кожним процесом. Конкретне виконання кожної програми може бути розглянуто як історія$S_0 \to S_1 \to \cdots \to S_n$, де S_0– початковий стан. Переходи між станами здійснюються неподільними

діями, що змінюють стан. Навіть паралельне виконання можна подати у вигляді лінійної історії, оскільки паралельна реалізація набору неподільних дій еквівалентна їх виконання в деякому послідовному порядку.

Мета синхронізації – виключити небажані історії паралельної програми. Взаємне виключення полягає в комбінуванні неподільних дій, що реалізуються безпосередньо апаратним забезпеченням у вигляді послідовностей дій, які називаються критичними секціями. Вони повинні бути неподільними, тобто їх не можна перервати діями інших процесів, які посилаються на ті ж змінні. Синхронізація за умовою (умовна синхронізація) означає, що дія буде здійснена, коли стан буде задовольняти заданій логічному умові. Обидві форми синхронізації можуть припиняти процеси, обмежуючи набір наступних неподільних дій.

Властивістю програми називається атрибут, який є істинним за будь-якої можливої історії програми і, отже, при всіх її виконаннях. Є два типи властивостей: безпека та живучість. Властивість безпеки полягає в тому, що програма ніколи не потрапляє в «поганий» стан (при якому деякі змінні можуть мати небажані значення).

2.3 Техніка виключення взаємоблокувань

Техніка виключення взаємоблокувань (Deadlock) – це набір методів, спрямованих на запобігання виникненню Deadlock до того, як система потрапить у небезпечний стан. Це один із трьох основних підходів до роботи з Deadlock (інші – запобігання та виявлення/відновлення).

Для того, щоб взаємоблокування стало можливим, повинні одночасно виконуватися чотири умови (Умови Коффмана):

– Взаємне Виключення (Mutual Exclusion): Ресурси не можуть використовуватися одночасно (режим ексклюзивного доступу).

– Утримання та Очікування (Hold and Wait): Процес, який утримує принаймні один ресурс, очікує на отримання додаткових ресурсів, які зараз утримуються іншими процесами.

– Відсутність Примусового Звільнення (No Preemption): Ресурси можуть бути звільнені процесом лише добровільно, після того, як процес завершив їх використання.

– Кругове Очікування (Circular Wait): Існує замкнений ланцюг процесів, де кожен процес у ланцюзі очікує ресурсу, утримуваного наступним процесом у ланцюзі.

Контрольні питання

1. Поясніть принципову різницю між процесом та потоком (thread) у термінах їхнього стану, адресного простору та накладних витрат на створення та перемикання контексту.
2. Чому виконання дій у критичній секції повинно бути атомарним (неподільним) по відношенню до інших процесів, і який механізм це забезпечує?
3. Уявіть проблему «Виробник-Споживач» з необмеженим буфером. Який примітив синхронізації (наприклад, м'ютекс чи семафор) є достатнім для захисту цілісності даних у цьому буфері, і чому він не запобігає голодуванню (starvation) споживача при порожньому буфері?
4. У чому полягає відмінність між блокуванням потоку (наприклад, за допомогою звичайного lock) та оператором очікування (наприклад, await в асинхронному програмуванні) з точки зору використання системних ресурсів?
5. Поясніть, чому взаємоблокування (Deadlock) є прикладом порушення властивості живучості.
6. Перелічіть чотири необхідні умови для виникнення взаємоблокування. Яку з цих умов найскладніше (або найменш бажано) порушувати в реальних системах, і чому?
7. Поясніть ключову відмінність між технікою запобігання взаємоблокуванням (Deadlock Prevention) та технікою виключення взаємоблокувань (Deadlock Avoidance) з точки зору накладних витрат та глибини попередньої інформації про процеси, необхідної системі.

Тема 3. Потоки в C#

3.1 Стан потоків

Кожен потік може бути в одному з кількох станів [16-17]. Потік, який готовий до виконання і очікує надання доступу до центрального процесора, знаходиться в стані «Готовий». Потік, який виконується в даний момент часу,

має статус «Виконується». При виконанні операцій введення-виведення або звернень до функцій ядра операційної системи потік знімається з процесора і перебуває в стані «Очікує». При завершенні операцій введення-виведення або повернення з функцій ядра потік міститься в чергу готових потоків. При перемиканні контексту потік вивантажується та поміщається у чергу готових потоків (рис.3.1).

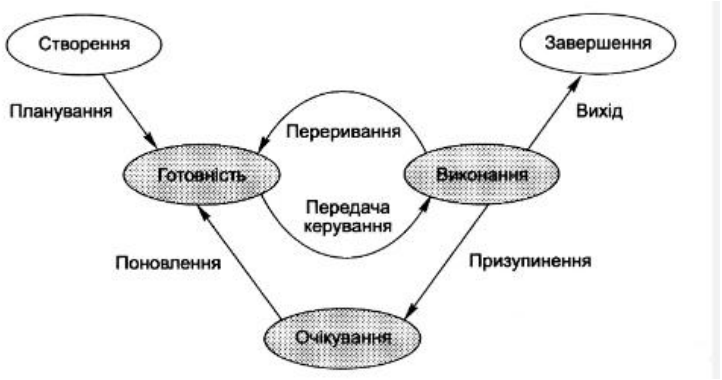


Рисунок 3.1 – Стан потоків

Перемикання контексту

1. Значення регістрів процесора для потоку, що виконується в даний момент, зберігаються в структурі контексту, яка розташовується в ядрі потоку.

2. З набору наявних потоків виділяється той, якому буде передано керування. Якщо вибраний потік належить до іншого процесу, Windows перемикає для процесора віртуальний адресний простір.

3. Значення вибраної структури контексту потоку завантажуються в регістри процесора.

3.2 Работа с потоками в C#

Класи, що підтримують багатопотокове програмування, визначені в просторі імен System.Threading. Система багатопотокового обробки грунтується на класі Thread, який інкапсулює потік виконання. Клас Thread є герметичним, тобто він не може успадковуватися.

Основні етапи роботи

```
// Ініціалізація потока
Thread oneThread = new Thread(Run);
// Запуск потока
oneThread.Start();
// Очікування завершення потоку
        oneThread.Join();
```

Як робочий елемент можна використовувати метод класу, делегат методу або лямбда-вираз. У наступному фрагменті створюються три потоки. Перший потік як робочий елемент приймає статичний метод LocalWorkItem. Другий потік ініціалізується за допомогою лямбда-вираження, третій потік зв'язується з методом загальнодоступного класу.

```
class Program
{
     static void LocalWorkItem()
     {
          Console.WriteLine("Hello from static method");
     }
     static void Main()
     {
          Thread thr1 = new Thread(LocalWorkItem);
          thr1.Start();
          Thread thr2 = new Thread(() =>
               {
                    Console.WriteLine("Hello from
                                        lambda-expression");
               });
          thr2.Start();
          ThreadClass thrClass = new  ThreadClass("Hello from
                                              thread-class");
          Thread thr3 = new Thread(thrClass.Run);
          thr3.Start();
     }
}
class ThreadClass
{
     private string greeting;
     public ThreadClass(string sGreeting)
     {
                    greeting = sGreeting;
               }
               public void Run()
               {
                    Console.WriteLine(greeting);
               }
          }
```

Для визначення моменту закінчення потоку можна скористатися значенням властивості IsAlive. Воно поверне false, коли потік завершиться. Крім того, можна скористатися спеціальним методів Join (), який дозволяє призупинити виконання основного потоку до моменту завершення тих вторинних потоків, для яких метод Join () був використаний.

Спочатку в середовищі .NET Framework не можна було передавати аргумент потоку, коли він починався, оскільки у методу, який служив в якості точки входу в потік, що не могло бути параметрів. Якщо ж потоку потрібно

передати якусь інформацію, то до цієї мети доводилося йти різними обхідними шляхами, наприклад, використовувати загальну змінну. Але цей недолік був згодом усунутий, і тепер аргумент може бути переданий потоку.

Для виклику методу без аргументів в звичайному потоці використовується делегат ThreadStart, а для передачі аргументів на потік, необхідно використовувати делегат ParameterizedThreadStart. З його допомогою в потік можна передати один аргумент типу object.

Для того щоб зробити потік фоновим, досить привласнити логічне значення true властивості IsBackground. А логічне значення false вказує на те, що потікє пріоритетним.

У кожного потоку є свій пріоритет, який певною мірою визначає, наскільки часто потік отримує доступ до ЦП. Взагалі кажучи, фонові потоки отримують доступ до ЦП рідше, ніж високопріоритетні. Слід мати на увазі, що, крім пріоритету, на частоту доступу потоку до ЦП впливають і інші фактори. так, якщо високопріоритетний потік очікує доступу до деякого ресурсу, наприклад, для введення з клавіатури, він блокується, а замість нього виконується фоновий потік. У подібній ситуації фоновий потік може отримувати доступ до ЦП частіше, ніж високопріоритетний потік протягом певного періоду часу. І нарешті, конкретне планування завдань на рівні операційної системи також впливає на час ЦП, що виділяється для потоку.

Пріоритет потоку можна змінити за допомогою властивості Priority, що є членом класу Thread. За замовчуванням для потоку встановлюється значення пріоритету ThreadPriority.Normal.

Багатопотоковий код може вести себе по-різному в різних середовищах, тому ніколи не слід покладатися на результати його виконання тільки в одному середовищі. Так було б помилкою вважати, що фоновий потік з наведеного вище прикладу буде завжди виконуватися лише протягом невеликого періоду часу до тих пір, поки не завершиться високопріоритетний потік. В іншому середовищі високопріоритетний потік може, наприклад, завершитися ще до того, як фоновий потік виконається хоча б один раз.

Коли використовується кілька потоків, то іноді доводиться координувати дії двох або більше потоків. Процес досягнення такої координації називається синхронізацією. Найпоширенішою причиною застосування синхронізації служить необхідність розділяти серед двох або більше потоків загальний ресурс, який може бути одночасно доступний тільки одному потоку.

Синхронізація організовується за допомогою ключового слова lock.

Ключове слово lock насправді служить в C # швидким способом доступу до засобам синхронізації, визначеним у класі Monitor, який знаходиться в просторі імен System.Threading. В цьому класі визначено, зокрема, ряд методів для управління синхронізацією. Наприклад, для отримання блокування об'єкта викликається метод Enter (), а для зняття блокування - метод Exit ().

Методи Wait (), Pulse () і PulseAll () визначені в класі Monitor і можуть викликатися тільки з заблокованого фрагмента блоку. Вони застосовуються у такий спосіб. Коли виконання потоку тимчасово заблоковано, він викликає метод Wait (). В підсумку потік переходить в стан очікування, а блокування з відповідного об'єкта знімається, що дає можливість використовувати цей об'єкт в іншому потоці. В Надалі очікує потік активізується, коли інший потік увійде в аналогічне стан блокування, і викликає метод Pulse () або PulseAll (). При виклику методу Pulse () поновлюється виконання першого потоку, що очікує своєї черги на отримання блокування. А виклик методу PulseAll () сигналізує про зняття блокування всім очікують потокам.

Найбільш небезпечними проблемами, з тих, що виникають під час паралельного виконання декількох потоків є проблеми взаимоблокировки і стану гонки. Взаємоблокування – це такий стан потоків, коли кожен чекає закінчення роботи інших і при цьому нічого не робить. Стан гонки – це спроби потоків отримати доступ до одного ресурсу без належної його блокування.

3.3 Пул потоків

Пул потоків призначений для спрощення багатопотокової обробки. Програміст виділяє фрагменти коду (робітники), які можна виконувати

паралельно. Планувальник (середовище виконання) оптимальним чином розподіляє робочі елементи по робочих потоках пулу. Таким чином, питання ефективного завантаження оптимальної кількості потоків вирішуються не програмістом, а планувальником (виконуючим середовищем). Ще однією перевагою застосування пулу є зменшення накладних витрат, пов'язаних з ручним створенням і завершенням потоків для кожного фрагмента коду, допускає розпаралелювання. Пул потоків використовується обробки завдань типу Task. Завдання мають ряд корисних вбудованих механізмів (очікування, скасування, продовження тощо). Тому для розпаралелювання робочих елементів рекомендується використовувати саме завдання чи шаблони класу Parallel. Безпосередня робота з пулом без явного визначення завдань може бути корисною, коли немає потреби у додаткових можливостях об'єкта Task.

Для додавання робочого елемента використовується метод

```
// Додавання методу без параметрів
ThreadPool.QuequeUserWorkItem(SomeWork);
// Додавання методу з параметром
        ThreadPool.QuequeUserWorkItem(SomeWork, data);
```

У наступному фрагменті проілюструємо основні особливості пулу потоків.

```
for(int i=0; i<10; i++)
{
     ThreadPool.QueueUserWorkIter((object o)=> {
          Console.WriteLine("i: {0}, ThreadId: {1},
               IsPoolThread: {2}",
               i, Thread.CurrentThread.ManagedThreadId,
               Thread.CurrentThread.IsThreadPoolThread);
          });
     Thread.Sleep(100);
}
```

Додаємо до пулу потоків 10 екземплярів безіменного делегата, оголошеного у вигляді лямбда-виразу. У робочому елементі здійснюється виведення значення індексу i, номер потоку та ознака того, що потік належить пулу.

```
i: 10, ThreadId: 3, IsPoolThread: True
i: 10, ThreadId: 4, IsPoolThread: True
i: 10, ThreadId: 4, IsPoolThread: True
i: 10, ThreadId: 4, IsPoolThread: True
i: 10, ThreadId: 4, IsPoolThread: True
i: 10, ThreadId: 3, IsPoolThread: True
i: 10, ThreadId: 3, IsPoolThread: True
i: 10, ThreadId: 3, IsPoolThread: True
i: 10, ThreadId: 3, IsPoolThread: True
i: 10, ThreadId: 4, IsPoolThread: True
```

Переконуємо, що справді всі робочі елементи виконувались потоками пулу (ознака IsPoolThread дорівнює true). Усього в обробці брало участь лише два потоки. Для цього коду можлива ситуація, коли всі робочі елементи будуть оброблятися в одному потоці пула. У всіх робочих елементах здійснюється висновок одного й того значення індексу i, рівного 10. Це пов'язано з асинхронністю запуску робочих елементів - основний потік продовжує обробляти цикл і збільшувати значення індексу, а робочі елементи фактично ще не виконуються. Щоб запобігти такій ситуації, необхідно використовувати індивідуальні копії для кожного елемента.

Контрольні питання

1. Перелічіть та коротко опишіть три основні стани життєвого циклу потоку в .NET (наприклад, Running, Suspended/Waiting, Stopped). Який із цих станів вимагає найбільше ресурсів для перемикання?
2. Яка принципова відмінність між фоновим потоком (IsBackground = true) та потоком переднього плану (Foreground Thread)? Як це впливає на завершення роботи всього процесу .NET?
3. Поясніть, що відбувається, коли ви викликаєте метод Thread.Interrupt(). Чому використання цього методу часто вважається небезпечним і замінюється механізмом токена скасування (CancellationToken)?
4. Коли в сучасному програмуванні .NET (після появи TPL) є виправданим явне створення нового об'єкта Thread замість використання пулу потоків? Наведіть приклад такого випадку.
5. Яку функцію виконує метод Thread.Join()? Що відбувається з потоком, що викликає, коли він викликає Join() на іншому потоці, і чому це вважається блокуючою операцією?
6. Яка головна мета використання Пулу Потоків (Thread Pool)? Як він допомагає мінімізувати накладні витрати (overhead) порівняно з постійним створенням нових потоків?
7. Як .NET керує мінімальною та максимальною кількістю потоків у пулі? Навіщо взагалі встановлювати максимальне обмеження?
8. Поясніть різницю між робочими потоками (Worker Threads) та потоками вводу/виводу (I/O Threads) у пулі. Який тип потоку використовується для обробки асинхронних операцій I/O-Bound (наприклад, мережеві запити)?

Тема 4. Синхронізація за допомогою об'єктів ядра і конструкцій користувацького режиму

4.1 Конструкції користувацького режиму (volatile та interlocked).

4.2 Конструкції синхронізації режиму ядра (AutoResetEvent і ManualResetEvent, Semaphore, Mutex)

4.3 Гібридні конструкції синхронізації потоків (Monitor, ManualResetEventSlim, SemaphoreSlim)

4.1 Конструкції рівня користувача

Синхронізація потоків є тією процедурою, яку по можливості потрібно уникати. Незалежно від типу блокуючих об'єктів завжди потрібно прагнути створювати код, в якому блокування утримується якомога менше часу. Якщо, наприклад, вам необхідно здійснити блокування ресурсу на довгий час, перевірте можливість створення локальної копії ресурсу для подальшої роботи з ним. якщо така можливість є, утримуйте блокування лише для того, щоб скопіювати ресурс і продовжити роботу з ним без установки тривалого блокування.

Власні бібліотеки класів слід будувати за таким зразком: Всі статичні методи слід зробити безпечними щодо потоків, а екземплярні – ні

Ключове слово volatile вказує, що поле може бути змінено декількома потоками, що виконуються одночасно. Поля, оголошені як volatile, що не проходять оптимізацію компілятором, яка передбачає доступ за допомогою окремого потоку. Це гарантує наявність найбільш актуального значення в поле в будь-який час.

Ключове слово volatile можна застосовувати до полів наступних типів:

– Силочні типи.

– Типи покажчиків (в небезпечному контексті). Зверніть увагу, що незважаючи на те, що сам покажчик може бути volatile, об'єкт, на який він вказує, таким бути не може. Іншими словами, не можна оголосити покажчик volatile.

– Типи, такі як sbyte, byte, short, ushort, int, uint, char, float і bool.

– Тип перерахування з одним з наступних базових типів: byte, sbyte, short, ushort, int або uint.

– Параметри універсальних типів, які є посилальними типами.

– Властивості IntPtr і UIntPtr.

Ключове слово volatile можна застосувати тільки до полів класу або структури. Локальні змінні не можуть бути оголошені як volatile.

Атомарні оператори

Бібліотека .NET 4.0 надає високоефективні атомарні оператори, реалізовані як статичні методи класу System.Threading.Interlocked. Атомарні оператори призначені для потокобезпечного неблокуючого виконання операцій над даними переважно целочисленного типу.

Таблиця 4.1 – Атомарні оператори

Оператор	**Метод**	**Типи даних**
Збільшення лічильника на одиницю	Increment	Int32, Int64
Зменьшення лічильника на одиницю	Decrement	Int32, Int64
Додавання	Add	Int32, Int64
Обмін значеннями	Exchange	Int32, Int64, double, single, object
Умовний обмін	CompareExchange	Int32, Int64, double, single, object
Читання 64-розрядного цілого	Read	Int64

Атомарність означає, що з виконанні оператора ніхто втрутиться у роботу потоку. Функціонально, атомарні оператори рівносильні критичній секції, виділеній за допомогою Lock, Monitor або інших засобів синхронізації.

```
lock (sync_obj)
  {
counter++;
  }
// можна виконати за допомогою атомарного оператора
        Interlocked.Increment(ref counter);
```

Атомарні оператори є неблокуючими – потік не вивантажується і не очікує, тому забезпечують високу ефективність. Виконання оператора Interlocked займає вдвічі менший час, ніж виконання критичної секції з блокуванням lock без конкуренції.

Оператор Interlocked.CompareExchange дозволяє атомарно виконати конструкцію «перевірити-привласнити»:

```
lock(LockObj)
  {
if(x == curVal)
x = newVal;
  }
```

oldVal = Interlocked.CompareExchange(ref x, newVal, curVal);

Якщо значення змінної x дорівнює значенню, що задається третім аргументом curVal, то змінній надається значення другого аргументу newVal. Значення, що повертається дозволяє встановити, чи здійснилася заміна значення.

4.2 Конструкції синхронізації режиму ядра

AutoResetEvent дозволяє потокам взаємодіяти один з одним шляхом передачі сигналів. Як правило, цей клас використовується, коли потокам потрібно винятковий доступ до ресурсу.

Виклик Set сигналізує події AutoResetEvent про необхідність звільнення очікує потоку. Подія AutoResetEvent залишається в сигнальному стані до звільнення одного очікує потоку, а потім повертається в несигнальному стан. Якщо немає очікують потоків, стан залишається сигнальним нескінченно. Якщо потік викликає метод WaitOne, а AutoResetEvent знаходиться в сигнальному стані, потік не блокується. AutoResetEvent негайно звільняє потік і повертається в несигнальному стан.

ManualResetEvent дозволяє потокам взаємодіяти один з одним шляхом передачі сигналів. Зазвичай це взаємодія стосується завдання, яку один потік повинен завершити до того, як інший продовжить роботу.

Коли потік починає роботу, яка повинна бути завершена до продовження роботи інших потоків, він викликає метод Reset для того, щоб помістити ManualResetEvent в несигнальному стан. Цей потік можна розуміти як контролюючий ManualResetEvent. Потоки, які викликають метод WaitOne в ManualResetEvent, будуть заблоковані, чекаючи сигналу. Коли контролюючий потік завершить роботу,він викличе метод Set для повідомлення про те, що очікують потоки можуть продовжити роботу. Всі очікують потоки звільняються.

Використовуйте клас Semaphore для управління доступом до пулу ресурсів. Потоки виробляють вхід в семафор, викликаючи метод WaitOne (), успадкований від класу WaitHandle, і звільняють семафор викликом методу Release ().

Лічильник на семафорі зменшується на одиницю кожного разу, коли в семафор входить потік, і збільшується на одиницю, коли потік звільняє семафор. коли лічильник дорівнює нулю, наступні запити блокуються, поки інші потоки не звільнять семафор. Коли семафор звільнений усіма потоками, лічильник має максимальне значення, задане при створенні семафора.

Гарантований порядок, в якому б блоковані потоки входили в семафор, наприклад, FIFO або LIFO, відсутня.

Семафори, як і м'ютекси, можуть бути локальними і системними (іменованими).

Коли двом або більше потокам одночасно потрібно доступ до загального ресурсу, системі необхідний механізм синхронізації, щоб забезпечити використання ресурсу тільки одним потоком одночасно. Mutex - примітив, який надає ексклюзивний доступ до загального ресурсу тільки одному потоку синхронізації. М'ютекс також зберігає інформацію про потік, який їм володіє і кількість блокувань який потік викликав на м'ютексів.

Якщо потік завершується, володіючи м'ютексів, то м'ютекс називається кинутим. Стан мьютекса задається сигнальним, і м'ютекс переходить у володіння наступного очікує потоку. Починаючи з версії 2.0 платформи .NET Framework, в наступному потоці, що отримав кинутий м'ютекс, видається виняток AbandonedMutexException. У версіях платформи .NET Framework 2.0 і раніших виняток не видавалась.

Кинутий системний м'ютекс може свідчити про раптовому припиненні виконання програми (наприклад, за допомогою диспетчера задач Windows). М'ютексів бувають двох типів: локальні м'ютекси (без імені), і іменовані системні м'ютекси. Локальний м'ютекс існує тільки всередині одного процесу. Він може використовуватися будь-яким потоком в процесі, який містить посилання на об'єкт Mutex, що представляє м'ютекс. Кожен неіменованого об'єкт Mutex представляє окремий локальний м'ютекс.

Іменовані системні м'ютекси доступні в межах всієї операційної системи і можуть бути використані для синхронізації дій процесів. можна створити об'єкт Mutex, що представляє іменований системний м'ютекс, використовуючи

конструктор з підтримкою імен. Об'єкт операційної системи може бути створений водночас, або існувати до створення об'єкта Mutex.

4.3 Гібридні конструкції синхронізації потоків

Конструкцію lock введено для зручності як аналог застосування об'єкта синхронізації Monitor.

Отже, конструкція

```
lock(sync_obj)
{
// Critical section
        }
```

аналогічна застосуванню об'єкта Monitor:

```
try
{
Monitor.Enter(sync_obj);
// Critical section
}
finally
{
Monitor.Exit(sync_obj);
        }
```

Блоки try-finally формуються для того, щоб гарантувати звільнення блокування (критичної секції) у разі виникнення будь-якого виключення всередині критичної секції.

Окрім «звичайного» входу в критичну секцію клас Monitor надає «умовні» входи:

```
b = Monitor.TryEnter(sync_obj);
if(!b)
{
// Выполняем полезную работу
DoWork();
// Снова пробуем войти в критическую секцию
Monitor.Enter(sync_obj);
}
// Критическая секция
ChangeData();
// Выходим
        Monitor.Exit(sync_obj);
```

Якщо критична секція виконується кимось іншим, то потік не блокується, а виконує корисну роботу. Після завершення всіх корисних робіт потік намагається увійти до критичної секції з блокуванням.

ManualResetEventSlim

ManualResetEventSlim – це гібридна конструкція синхронізації в .NET, що є високопродуктивною, «легшою» версією об'єкта синхронізації режиму ядра ManualResetEvent. Вона розроблена для сценаріїв, де час очікування потоку, ймовірно, буде коротким.

ManualResetEventSlim (MRES) поєднує в собі два механізми очікування:

1. Режим Користувача (Spin-Wait): Для короткого очікування MRES використовує цикли активного очікування (spinlock) та інші неблокуючі техніки. Це дозволяє уникнути дорогих перемикань у режим ядра, якщо сигнал очікується дуже швидко. Це ідеально підходить для високопродуктивних сценаріїв, де більшість очікувань триває кілька мілісекунд.

2. Режим Ядра (WaitHandle): Якщо потік змушений чекати довше, ніж заданий пороговий час (або якщо MRES містить понад 10 очікуючих потоків), він автоматично переходить у блокування режиму ядра, використовуючи внутрішній об'єкт ManualResetEvent. Це запобігає марнуванню часу процесора на активне очікування.

Таблиця 4.2 – Ключові Властивості

Властивість	**Опис**
`IsSet`	Повертає `true`, якщо подія перебуває в сигнальному (відкритому) стані.
`Wait()`	Блокує потік до тих пір, поки подія не буде встановлена у сигнальний стан.
`Set()`	Переводить подію у сигнальний стан. Усі потоки, які очікують, вивільняються.
`Reset()`	Переводить подію у несигнальний стан. Потоки, які викликають `Wait()`, блокуються.

SemaphoreSlim

SemaphoreSlim – це гібридна конструкція синхронізації в .NET, яка обмежує кількість потоків, що можуть одночасно отримати доступ до спільного ресурсу або розділу коду. Вона є легшою, швидшою та більш масштабованою

версією класичного об'єкта Semaphore і призначена виключно для використання в межах одного процесу.

SemaphoreSlim вирішує класичну проблему обмеження паралелізму (наприклад, дозволити лише трьом потокам одночасно обробляти запити до бази даних). Він підтримує внутрішній лічильник. Потік викликає метод Wait() або WaitAsync() для зменшення лічильника. Якщо лічильник стає нульовим, потік блокується, доки інший потік не викличе метод Release().

Сценарії використання:

- Обмеження кількості одночасних підключень до ресурсу (наприклад, до файлу або зовнішнього API).
- Контроль паралелізму в масиві асинхронних завдань (Task).

SemaphoreSlim є гібридним примітивом, що робить його високопродуктивним у більшості випадків:

- Режим Користувача (Spin-Wait): Для коротких періодів очікування він використовує активне очікування (SpinWait) без перемикання в режим ядра. Це дозволяє уникнути великих накладних витрат, якщо потік, який утримує семафор, швидко викликає Release().
- Режим Ядра (WaitHandle): Якщо потік очікує довше або якщо існує багато очікуючих потоків, SemaphoreSlim автоматично переходить у блокування режиму ядра, використовуючи внутрішній об'єкт WaitHandle. Це запобігає марнуванню часу процесора на тривале активне очікування.

Найважливіша перевага SemaphoreSlim – це підтримка асинхронного очікування через метод WaitAsync():

- await semaphore.WaitAsync(): Коли асинхронне завдання очікує семафора, воно не блокує фізичний потік. Замість цього, завдання "звільняється" з потоку пулу, і продовжується лише після того, як семафор стає доступним.
- Ця функція є життєво важливою для сучасних високомасштабованих веб-серверів та додатків, де блокування потоків є неприпустимим.

Контрольні питання

1. Що гарантує ключове слово volatile в C# (з точки зору пам'яті та кешування)? Чому воно ефективне лише для однорядкових операцій читання/запису і не може захистити від стану гонки при x++?
2. У чому основна перевага використання класу Interlocked порівняно з volatile?
3. Яка ключова функціональна можливість Mutex дозволяє використовувати його для міжпроцесної синхронізації, на відміну від Monitor?
4. Який основний недолік (з точки зору продуктивності) мають усі конструкції режиму ядра порівняно з режимом користувача, і чому вони все ж необхідні?
5. Як Monitor забезпечує, щоб при короткому очікуванні не відбувалося перемикання в режим ядра?
6. Чому використання SemaphoreSlim є кращим для внутрішньопроцесної синхронізації в сучасному .NET-додатку? Яку ключову функцію він підтримує, на відміну від класичного Semaphore?
7. Яким чином Mutex автоматично запобігає проблемі взаємоблокування (Deadlock) при аварійному завершенні процесу, на відміну від звичайного lock?

Тема 5. Асинхронна модель програмування

5.1 Основні поняття

Асинхронна модель – це парадигма програмування, яка дозволяє виконувати неблокуючі (non-blocking) операції, щоб основний потік програми міг продовжувати виконувати іншу роботу, поки очікувана операція (наприклад, мережевий запит або доступ до файлу) завершується. Вона є критично важливою для сучасних високомасштабованих додатків, особливо в хмарних середовищах.

Основна ідея асинхронності полягає у звільненні потоку під час очікування.

1. Блокуюча (синхронна) модель

У синхронній (блокуючій) моделі, коли потік викликає операцію вводу/виводу (I/O) — наприклад, читання великого файлу з диска – цей потік блокується і не робить абсолютно нічого, поки дані не будуть готові.

Наслідки: Марнування системних ресурсів. Якщо веб-сервер має тисячі користувачів, які чекають на I/O, йому знадобиться тисяча потоків, які простоюють.

2. Асинхронна (неблокуюча) модель

В асинхронній моделі, коли потік ініціює операцію I/O:

Ініціація: Потік відправляє запит на I/O до операційної системи.

Звільнення: Замість того, щоб чекати, потік повертається до пулу потоків і може обслуговувати інші запити або виконувати інший обчислювальний код.

Очікування: Операційна система (ОС) виконує I/O-операцію (наприклад, мережева карта чекає на відповідь) без участі основного потоку.

Продовження: Коли I/O-операція завершується, ОС надсилає повідомлення про завершення (наприклад, через I/O Completion Port). Потік із пулу бере це повідомлення і виконує продовження (Continuation) — код, який повинен виконатися після отримання результату.

5.2 Механіка асинхронних викликів методів

Класи, в яких є вбудована підтримка асинхронної моделі, мають пару асинхронних методів для кожного з синхронних методів. Ці методи починаються зі слів Begin і End. Наприклад, якщо ми хочемо скористатися асинхронним варіантом методу Read класу System.IO.Stream, нам потрібно використовувати методи BeginRead і EndRead цього ж класу.

Для використання вбудованої підтримки асинхронної моделі програмування потрібно викликати відповідний метод BeginOperation і вибрати модель завершення виклику. Виклик методу BeginOperation повертає об'єкт інтерфейсу IAsyncResult, за допомогою якого визначається стан виконання асинхронної операції.

Метод EndOperation застосовується для завершення асинхронного виклику в тих випадках, коли основному потоку необхідно виконати великий обсяг обчислень, що не залежать від результатів виклику асинхронного методу. Після того як основна робота зроблена і додаток потребує результатах виконання

асинхронного методу для подальших дій, викликається метод EndOperation. При цьому основний потік буде призупинено до завершення роботи асинхронного методу.

Спосіб завершення асинхронного виклику Callback використовується в тих випадках, коли потрібно запобігти блокуванню основного потоку. При використанні Callback ми запускаємо метод EndOperation в тілі методу, який викликається при завершенні методу, що працює в паралельному потоці. Сигнатура викликається методу повинна збігатися з сигнатурою делегата AsyncCallback.

Виклик асинхронних делегатів дозволяє неявно поміщати потоки в ThreadPool, тим самим позбавляючи програміста від необхідності працювати з ним безпосередньо.

Сигнатура методу BeginInvoke не відповідає методу Invoke. Це пояснюється тим, що потрібен певний спосіб ідентифікації певного елемента роботи, який тільки що був відкладений викликом Beginlnvoke. Таким чином, Beginlnvoke повертає посилання на об'єкт, який реалізує інтерфейс IAsyncResult. цей об'єкт подібний cookie-набору, який зберігається для ідентифікації виконується елемента роботи. Через методи інтерфейсу IAsyncResult можна перевіряти стан операції, наприклад, її готовність.

Коли потік, запитаний для виконання операції, завершить свою роботу, він викликає Endlnvoke на делегата. Однак, оскільки метод повинен мати спосіб ідентифікації асинхронної операції, результат якої потрібно отримати, йому повинен бути переданий об'єкт, отриманий з методу Beginlnvoke.

Якщо в процесі асинхронного виконання в пулі потоків цільового коду делегата буде згенеровано виняток, воно згенерує повторно, коли ініціює потік викличе EndInvoke.

Пул потоків має наступні переваги:

– Пул потоків управляє потоками ефективно, зменшуючи кількість створюваних, запускати і зупиняти потоків.

– Використовуючи пул потоків, можна зосередитися на вирішенні завдання, а не на інфраструктурі потоків додатки.

Ситуації, в яких переважно ручне управління потоками:

– Якщо потрібні потоки переднього плану, або повинен бути встановлений пріоритет потоку. Увага: Потоки з пулу завжди є фоновими з пріоритетом по замовчуванням (ThreadPriority.Normal).

– Якщо потрібно потік з фіксованою ідентичністю, щоб можна було переривати його або знаходити по імені.

Інтерфейс IAsyncResult реалізований за допомогою класів, що містять методи, які можуть працювати асинхронно. Об'єкт, який забезпечує роботу інтерфейсу IAsyncResult, зберігає в собі відомості про стан асинхронної операції і надає об'єкт синхронізації, що сигналізує потоку про завершення операції.

Для обробки результатів асинхронної операції в окремому потоці використовується делегат AsyncCallback. Делегат AsyncCallback представляє метод зворотного виклику, який викликається при завершенні асинхронної операції. Метод зворотного виклику приймає параметр IAsyncResult, який згодом використовується для отримання результатів асинхронної операції.

Клас AsyncResult використовується в поєднанні з асинхронними викликами методів з допомогою делегатів. IAsyncResult, повернутий з делегатському методу BeginInvoke, можна привести до AsyncResult. AsyncResult має властивість AsyncDelegate, що містить об'єкт делегата, до якого було направлено асинхронний виклик.

Контрольні питання

1. Яка ключова відмінність між викликом Delegate.Invoke() та Delegate.BeginInvoke()? Який із них повертає контроль негайно, і який об'єкт він повертає для управління операцією?
2. Поясніть призначення двох останніх параметрів методу BeginInvoke: AsyncCallback та object asyncState. Як ці параметри дозволяють реалізувати метод зворотного виклику (callback)?
3. Який метод делегата необхідно викликати після завершення асинхронної операції, і що він робить? Чому його виклик є критично важливим для уникнення витоку пам'яті та коректного отримання результату/винятків?
4. Як можна використовувати властивість IsCompleted об'єкта IAsyncResult для реалізації методу опитування (polling) результату асинхронного виклику делегата. Який недолік цього підходу в контексті продуктивності?

5. Коли ви викликаєте BeginInvoke, який механізм .NET (який ресурс) використовується для фактичного виконання синхронного цільового методу делегата асинхронно?

Тема 6. TPL. Паралельне програмування

6.1 Призначення TPL

Вищий рівень складається з двох API структурованого паралелізму даних: PLINQ та Parallel класу (рис. 6.1). Нижній рівень містить класи паралелізму завдань, а також набір додаткових конструкцій для допомоги в паралельному програмуванні [11-13].

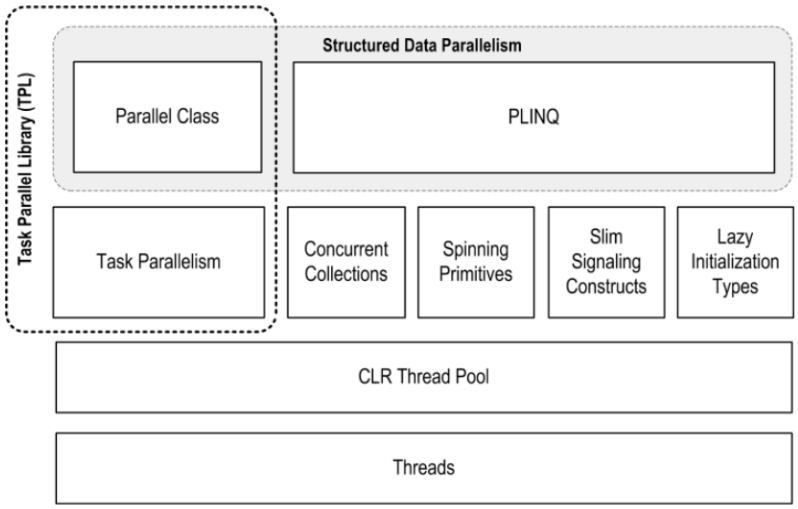


Рисунок 6.1 – Рівні функціональності

Задачі (tasks) є основним будівельним блоком бібліотеки Task Parallel Library. Задачі являють собою робочі елементи, які можуть виконуватися паралельно. В якості робочих елементів можуть використовуватися: методи, делегати, лямбда-вирази.

```
static void HelloWorld()
{
     Console.WriteLine("Hello, world!");
}
static void Main()
{
     // Используем обычный метод
     Task t1 = new Task(HelloWorld);
     // Используем делегат Action
     Task t2 = new Task(new Action(HelloWorld));
     // Используем безымянный делегат
     Task t3 = new Task(delegate
          {
               HelloWorld();
          });
     // Используем лямбда-выражение
     Task t4 = new Task(() => HelloWorld());
     // Используем лямбда-выражение
     Task t5 = new Task(() =>
          {
                    HelloWorld();
          });
     Task t6 = new Task(() =>
          {
                    Console.WriteLine("Hello, world!");
          });

     // Запускаем задачи
     t1.Start(); t2.Start(); t3.Start();
     t4.Start(); t5.Start(); t6.Start();
     // Дожидаемся завершения задач
     Task.WaitAll(t1, t2, t3, t4, t5, t6);
}
```

Робота з задачами, як правило, включає три основні операції: оголошення завдання, додавання завдання в чергу готових завдань, очікування завершення виконання задачі.

Робота з потоками:

```
Thread threadOne = new Thread(SomeWork);
        threadOne.Start();
        threadOne.Join();
```

Работа з задачами:

```
Task taskOne = new Task(SomeWork);
        taskOne.Start();
        taskOne.Wait();
```

Важлива відмінність полягає в тому, що виклик методу Start для задачі не створює новий потік, а поміщає задачу в чергу готових задач – пул потоків. Планувальник (TaskScheduler) відповідно до своїх правил розподіляє готові задачі по робочих потокам. Дії планувальника можна коригувати за допомогою параметрів задач. Момент фактичного запуску задачі в загальному випадку не визначений і залежить від завантаженості пулу потоків.

Для більш лаконічного «запуску» задачі існує шаблон, який замінює етапи оголошення і додавання задачі в пул потоків:

Task t = Task.Factory.StartNew(SomeWork);

Очікування завершення конкретної задачі здійснюється за допомогою методу Wait. Для очікування завершення декількох завдань існують статичні методи класу Task, які беруть в якості аргументу масив задач:

```
Task t1 = Task.Factory.StartNew(DoWork1);
Task t2 = Task.Factory.StartNew(DoWork2);
Task t3 = Task.Factory.StartNew(DoWork3);
// Дожидаемся завершения хотя бы одной задачи
int firstTask = Task.WaitAny(t1, t2, t3);
// Дожидаемся завершения всех задач
Task.WaitAll(t1, t2, t3);
```

Методи очікування WaitAll і WaitAny можуть приймати в якості аргументів, як масив задач (один параметр), так і самі задачі (довільне число параметрів). Виклик WaitAll блокує поточний потік до завершення всіх зазначених задач. Виклик WaitAny блокує поточний потік до завершення хоча б однієї із зазначених задач і повертає номер першої завершеної задачі.

6.2 Статуси задачі

Задачі може перебувати в кількох станах: Created, Running, WaitingToRun, Faulted, Canceled, RanToCompletion, WaitingForActivation, WaitingForChildrenToComplete. Переходи між основними станами зображені на рисунку 6.2.

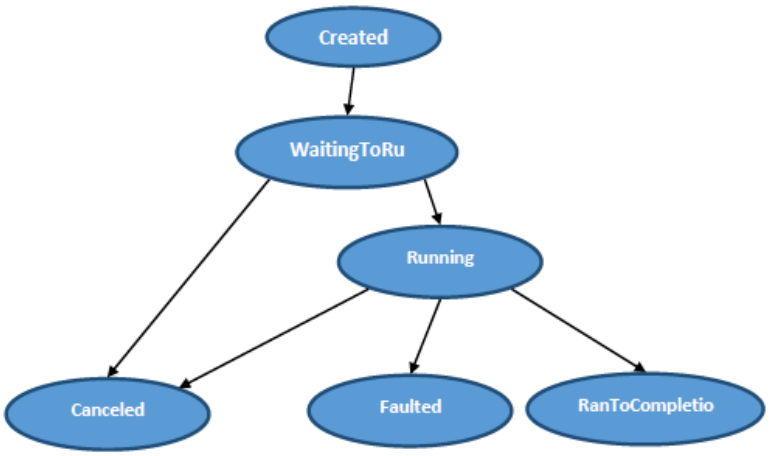


Рисунок 6.2 – Статуси задачі

При оголошенні задача отримує статус Created. Запуск задачі за допомогою методу Start або StartNew поміщає задачу в пул потоків зі статусом WaitingToRun. Задача, що виконується в даний момент, має статус Running. Статус Canceled відповідає задачі, скасованої за допомогою об'єкта CancellationTokenSource. Статус Faulted – при виконанні задачі сталася якась помилка, необроблена в коді задачі. При успішному завершенні задачі – статус RanToCompletion. Задачі-продовження починають зі статусу WaitingForActivation. При наявності дочірніх вкладених задач батьківська задача після завершення своєї роботи (Running) очікує завершення вкладених задач зі статусом WaitingChildrenToComplete.

У коді задачі можна запускати вкладені задачі, які можуть бути дочірніми та недочірніми (рис.6.3).

```
Task tParent = Task.Factory.StartNew( () =>
        {
            Console.WriteLine("Parent task starts");
            Task t1 = Task.Factory.StartNew( () =>
                Console.WriteLine("Inner task"));
            Task t2 = Task.Factory.StartNew( () =>
                Console.WriteLine("Child task"),
                TaskCreationOptions.AttachedToParent);
            Console.WriteLine("Parent task ends");
        });
   tParent.Wait();
   Console.WriteLine("Parent task really ends");
```

```
Parent task starts
Parent task ends
Child task
Parent task really ends
Inner task
```

Рисунок 6.3 – Вкладені задачі

Вбудований механізм узгодженої відміни задач дозволяє уніфікованим способом реалізовувати правильно дострокове завершення виконання задач.

Для реалізації механізму відміни необхідно виконати наступні кроки:

1. Створіть об’єкт CancellationTokenSource в області видимості методу, який породжує і запускає задачу.

2. Отримати об'єкт CancellationToken, через який здійснюється взаємодія із задачею.

3. Передайте об'єкт CancellationToken при запуску задачі.

4. Реалізувати в задачі процедуру відміни.

5. При необхідності відмінити метод Cancel().

Обробник скасування задачі можна реалізувати або в самій задачі, або призначити окремий делегат, який буде викликатись при виникненні сигналу про скасування задачі.

6.3 Задачі-продовження

Задачі-продовження призначені для планування запуску задач після завершення попередніх задач із тим чи іншим статусом завершення: OnlyOnRanToCompletion, OnlyOnCanceled, OnlyOnFaulted, NotOnCancelled, NotOnRanToCompletion.

```
// Основна задача, яка виконує розрахунок
Task t1 = Task<int>.Factory.StartNew(() => FindDecision());
// Вивід результатів в окремій задачі
Task t2 = t1.ContinueWith((prev) =>
            Console.WriteLine("Result: {0}", prev.Result),
            TaskContinuationOptions.OnlyOnRanToCompletion);
// Обробник помилок
Task t3 = t1.ContinueWith((prev) =>
            Console.WriteLine("Error: {0}",
            prev.Exception.InnerException.Message),
            TaskContinuationOptions.OnlyOnFaulted);
// Задача була відмінена
Task t4 = t1.ContinueWith((prev) =>
            Console.WriteLine("Task was cancelled"),
            TaskContinuationOptions.OnlyOnCanceled);
```

Перша задача здійснює основний розрахунок. Наступні завдання виконуються в залежності від статусу завершення першого завдання. Друге завдання виконується за успішного завершення. Третє завдання виконується у разі необробленого виключення. Четверте завдання виконується лише у разі скасування першої.

Завдання-продовження дозволяють без додаткових засобів синхронізації реалізувати критичну секцію та конструкцію бар'єру:

```
// Оголошуємо завдання, які можуть виконуватися паралельно
    Task[] tasks = new Task[3];
```

```
        tasks[0] = new Task(Work1);
        tasks[1] = new Task(Work2);
        tasks[2] = new Task(Work3);
// Плануємо виконання критичної секції
Task tCr = Task.Factory.ContinueWhenAll(tasks, (tt) => {
        // Критична секція
        });
// Паралельні задачі
        Task t5 = tCr.ContinueWith(Work5);
        Task t6 = tCr.ContinueWith(Work6);
// Запускаємо задачі
        tasks[0].Start(); tasks[1].Start(); tasks[2].Start();
// Очікуємо завершення останньої завдачі
        t6.Wait();
```

Контрольні питання

1. Поясніть різницю між Task (з TPL) та Thread (з класом System.Threading.Thread). Чому Task вважається абстракцією роботи, а не ресурсу, і як він використовує пул потоків?
2. У чому основна відмінність при використанні Task.Run() та Task.Factory.StartNew() для запуску задачі в пулі потоків? Який метод рекомендується для більшості випадків, і чому?
3. У якому кінцевому стані опиняється Task, якщо в ньому виникає необроблений виняток? Як можна безпечно отримати та обробити цей виняток у поточному потоці, що викликає?
4. Що таке задача-продовження (continuation task) і навіщо вона потрібна? Як вона вирішує проблему блокування потоків при необхідності виконати код після завершення попереднього завдання?
5. Опишіть, як використовується метод Task.ContinueWith(). Який аргумент дозволяє задачі-продовженню отримати результат або виняток із попередньої задачі?

Тема 7 Async і Await

7.1 Проблеми синхроних операцій.
7.3. Конструкція async – await.
7.2 Проблеми оновлення GUI з іншого потоку

7.1 Проблеми синхроних операцій

Для збільшення швидкості відгуку програми і збільшення продуктивності в програмуванні використовується асинхронність. Підтримка асинхронности виявляється особливо важливою для додатків з доступом до потоку

користувацького інтерфейсу, оскільки всі дії, пов'язані з оновленням GUI, зазвичай виконуються в одному потоці [14-15].

До виходу C # 5 способи створення асинхронного коду були громіздкими, важкими в написанні, налагодження та супровід. З виходом п'ятої версії C # розробник отримав інструментарій у вигляді ключових слів async і await.

Ключове слово async вказує компілятору, що метод є асинхронним. Ключове слово await вказує компілятору, що в цій точці необхідно дочекатися закінчення асинхронної операції (при цьому управління повертається викликав методу). Асинхронний метод зазвичай містить один або кілька входжень оператора await, але відсутність вираження await не викликає помилку компілятора. Метод, позначений ключовим словом async, і не містить жодного виразу з await не є асинхронним. За угодою назву асинхронного методу повинно закінчуватися словом Async.

Асинхронний метод може мати тип значення Task, Task <TResult> або void. Метод не може мати ref або out параметри, але може викликати методи, які мають такі параметри. Винятки «викидаються» в місці виклику асинхронної операції, а не Callback- методу.

Метод Task.WhenAll створює завдання, яка буде виконана після виконання всіх наданих завдань. Метод Task.WhenAny створює завдання, яка буде виконана після виконання будь-який з наданих завдань.

Асинхронність дозволяє винести окремі завдання з основного потоку в спеціальні асинхронні методи або блоки коду. Особливо це актуально в графічних програмах, де тривалі завдання можу блокувати інтерфейс користувача. І щоб цього не сталося, потрібно задіяти асинхронність. Також асинхронність несе вигоди в веб-додатках при обробці запитів від користувачів, при зверненні до баз даних або мережевих ресурсів. При великих запитах до бази даних асинхронний метод просто засне на час, поки не отримає дані від БД, а основний потік зможе продовжити свою роботу. У синхронному ж додатку, якби код отримання даних знаходився в основному потоці, цей потік просто б блокувався на час отримання даних.

7.2 Конструкція async – await

Ключовими для роботи з асинхронними викликами в C # є два ключових слова: async і await , мета яких спростити написання асинхронного коду. Вони використовуються разом для створення асинхронного методу.

Асинхоний метод володіє наступними ознаками:

- У заголовку методу використовується модифікатор async.
- Метод містить одне або кілька виразів await
- Як повертається типу використовується один з наступних:
    - Void;
    - Task;
    - Task<T>;
    - ValueTask<T>.

Асинхронний метод, як і звичайний, може використовувати будь-яку кількість параметрів або не використовувати їх взагалі. Однак асинхронний метод не може визначати параметри з модифікаторами out і ref.

Також варто відзначити, що слово async , яке зазначається в ухвалі методу, що не робить автоматично метод асинхронним. Воно лише вказує, що даний метод може містити одне або кілька виразів await .

Розглянемо приклад асинхронного методу:

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace HelloApp
{
   class Program
   {
      static void Factorial()
      {
         int result = 1;
         for(int i = 1; i <= 6; i++)
         {
            result *= i;
         }
         Thread.Sleep(8000);
         Console.WriteLine($"Факториал равен {result}");
      }
      // определение асинхронного метода
      static async void FactorialAsync()
```

```
        {
            Console.WriteLine("Начало метода FactorialAsync"); // выполняется синхронно
            await Task.Run(()=>Factorial());              // выполняется асинхронно
            Console.WriteLine("Конец метода FactorialAsync");
        }

        static void Main(string[] args)
        {
            FactorialAsync();   // вызов асинхронного метода

            Console.WriteLine("Введите число: ");
            int n = Int32.Parse(Console.ReadLine());
            Console.WriteLine($"Квадрат числа равен {n * n}");

            Console.Read();
        }
    }
}
```

Тут перш за все визначено звичайний метод підрахунку факторіала. Для імітації довгої роботи в ньому використовується затримка на 8 секунд за допомогою методу Thread.Sleep(). Умовно це деякий метод, який виконує деяку роботу тривалий час. Але для спрощення розуміння він просто підраховує факторіал числа 6.

Також тут визначено асинхронний метод FactorialAsync(). Асинхронним він є тому, що має у визначенні перед повертається типом модифікатор async , його повертається типом є void, і в тілі методу визначено вираз await .

Вираз await визначає завдання, яка буде виконуватися асинхронно. В даному випадку подібне завдання представляє виконання функції факторіалу:

await Task.Run(()=>Factorial());

За негласними правилами в назві асінхроннних методів прийнято використовувати суфікс Async, хоча в принципі це необов'язково робити.FactorialAsync().

Сам факторіал ми отримуємо в асинхронному методі FactorialAsync. Асинхронним він є тому, що він оголошений з модифікатором asyncі містить використання ключового слова await. І в методі Main ми викликаємо цей асинхронний метод. Подивимося, який у програми буде консольний висновок:

```
Початок методу FactorialAsync
Введіть число:
7
Квадрат числа дорівнює 49
```

```
Кінець методу Main
Факторіал дорівнює 720
Закінчення методу FactorialAsync
```

Розберемо поетапно, що тут відбувається:

1. Запускається метод Main, в якому викликається асинхронний метод FactorialAsync.

2. Метод FactorialAsync починає виконуватися синхронно аж до вираження await.

3. Вираз await запускає асинхронну завдання Task.Run(()=>Factorial())

4. Поки виконується асинхронна завдання Task.Run(()=>Factorial())(а вона може виконуватися досить продожітельность час), виконання коду повертається в викликає метод - тобто в метод Main. У методі Main нам буде запропоновано ввести число для обчислення квадрата числа.

В цьому і перевага асинхронних методів – асинхронна задача, яка може виконуватися досить довгий час, не блокує метод Main, і ми можемо продовжувати роботу з ним, наприклад, вводити і обробляти дані.

Коли асинхронна завдання завершила своє виконання (у разі вище – підрахувала факторіал числа), продовжує роботу асинхронний метод FactorialAsync, який викликав асинхронну завдання.

7.3 Проблеми оновлення GUI з іншого потоку

Проблема оновлення графічного інтерфейсу користувача (GUI) з іншого потоку є фундаментальною у багатопотоковому програмуванні та виникає через те, що більшість GUI-фреймворків (WPF, WinForms, Android, iOS) засновані на однопотоковій моделі для управління UI-елементами. Основна причина проблеми полягає в відсутності взаємного виключення для операцій візуалізації.

1. Потік UI (UI Thread): Елементи GUI (кнопки, текстові поля, панелі) створюються і контролюються єдиним, спеціальним потоком (часто називається потоком диспетчера або потоком інтерфейсу користувача). Цей потік відповідає за:

- Обробку подій користувача (натискання клавіш, кліки мишею).

- Перемальовування (рендеринг) елементів на екрані.
- Виконання бізнес-логіки, що викликається подіями.

2. Фоновий потік (Background Thread): Якщо фоновий потік (який виконує довгу операцію, наприклад, обчислення або I/O) намагається безпосередньо змінити властивість UI-елемента (наприклад, Button.Text або ProgressBar.Value), виникає стан гонки (Race Condition).

3. Порушення цілісності (Data Corruption): Це може призвести до невизначеної поведінки, помилок візуалізації, порушення інваріантів структури даних UI, або навіть до аварійного завершення (crash) програми. Наприклад, один потік може почати оновлювати список, а інший – його перемальовувати.

Єдиний безпечний спосіб оновити GUI з фонового потоку – це передати запит на оновлення до самого потоку UI і дозволити йому виконати цю дію.

У .NET це досягається за допомогою спеціальних механізмів:

1. Синхронна передача (Invoke / Dispatcher.Invoke)

Використовується для блокуючого виклику. Фоновий потік блокується доти, доки потік UI не виконає переданий делегат.

- WinForms: Control.Invoke(Delegate method).
- WPF: Dispatcher.Invoke(Action method).

2. Асинхронна передача (BeginInvoke / Dispatcher.BeginInvoke)

Використовується для неблокуючого виклику. Фоновий потік негайно продовжує свою роботу після розміщення делегата у черзі потоку UI.

- WinForms: Control.BeginInvoke(Delegate method)
- WPF: Dispatcher.BeginInvoke(Action method)

3. Сучасне Рішення: async та await

У сучасних .NET-додатках (використовуючи Task-based Asynchronous Pattern, TAP), проблема вирішується автоматично завдяки контексту синхронізації (SynchronizationContext):

- Коли асинхронний метод (позначений async) викликає await, система зберігає поточний SynchronizationContext.

– Після завершення асинхронної I/O-операції, виконання продовження (коду, що йде після await) автоматично повертається до оригінального потоку UI через збережений контекст.

– Це дозволяє писати код, який виглядає синхронно, але безпечно оновлює UI:

```
private async void Button_Click(...)
{
   // Асинхронна операція - потік UI звільняється.
   var data = await GetDataFromNetworkAsync();

   // Код тут автоматично виконується в потоці UI.
   MyTextBox.Text = data; // Безпечне оновлення UI
```

Контрольні питання

1. Поясніть, що таке «блокування (зависання) інтерфейсу користувача (UI)r» у контексті однопотокової моделі. Як синхронна, довготривала операція (наприклад, мережевий запит) призводить до цього ефекту?
2. Наведіть приклад, де синхронна операція заважає обробці простих подій (наприклад, натисканню кнопки)
3. Сформулюйте основне правило в таких фреймворках, як WPF, WinForms чи Android: чому елементи GUI можна оновлювати лише з потоку, який їх створив (потоку UI)?
4. Які два ключові механізми (або класи/методи, наприклад, Dispatcher.Invoke або Control.Invoke) використовуються у WinForms чи WPF для безпечної передачі завдання оновлення UI з фонового потоку до потоку UI?
5. Поясніть призначення ключових слів async та await. Який із них позначає метод, а який призупиняє виконання цього методу, не блокуючи потік?

Тема 8 Технологія OpenMP

8.1 Призначення OpenMP

OpenMP (Open specifications for Multi-Processing) - це набір специфікацій для паралелізації програм у середовищі із загальною пам'яттю. Інтерфейс OpenMP задуманий як стандарт для програмування на масштабованих SMP-

системах (SSMP, ccNUMA) в моделі загальної пам'яті (shared memory model). У стандарт OpenMP входять специфікації набору директив компілятора, процедур і змінних середовища [3].

OpenMP дозволяє легко і швидко створювати багатопоточні додатки на алгоритмічних мовах Fortran і C/C++. При цьому директиви OpenMP аналогічні директивам препроцесора для мови C/C++ і є аналогом коментарів у алгоритмічній мові Fortran. Це дозволяє в будь-який момент розробки паралельної реалізації програмного продукту при необхідності повернутися до послідовного варіанту програми.

OpenMP - це стандартна модель для паралельного програмування в середовищі зі спільною пам'яттю. У даній моделі всі процеси спільно використовують загальний адресний простір, до якого вони асинхронно звертаються із запитами на читання і запис. У таких моделях для управління доступом до загальної пам'яті використовуються різні механізми синхронізації типу семафорів та блокування процесів. Перевага цієї моделі з точки зору програмування полягає в тому, що поняття монопольного володіння даними відсутнє, отже, не потрібно явно задавати обмін даними між потоками, що їх задають, та потоками, що їх використовують. Ця модель, з одного боку, спрощує розробку програми, але, з іншого боку, ускладнює розуміння і управління локальністю даних, написання детермінованих програм. В основному вона використовується при програмуванні для архітектур зі спільною пам'яттю.

Прикладами систем зі спільною пам'яттю, які мають велике число процесорів, можуть служити суперкомп'ютери Cray Origin2000 (до 128 процесорів), HP 9000 V-class (до 32 процесорів в одному вузлі, а в конфігурації з 4 вузлів - до 128 процесорів), Sun Starfire (до 64 процесорів).

8.2 Стандарт OpenMP

Розробкою стандарту займається організація OpenMP ARB (ARchitecture Board), до якої увійшли представники найбільших компаній - розробників SMP-архітектур і програмного забезпечення. Перша версія специфікації OpenMP (www.openmp.org) з'явилася в 1997 році і призначалася для мови програмування Фортран. Біля витоків OpenMP стояли такі відомі компанії, як IBM, Intel, Sun і

Hewlett-Packard. У 1998 році з'явилися варіанти OpenMP для мов C/C++, і на даний момент останньою є версія 3.0 [5].

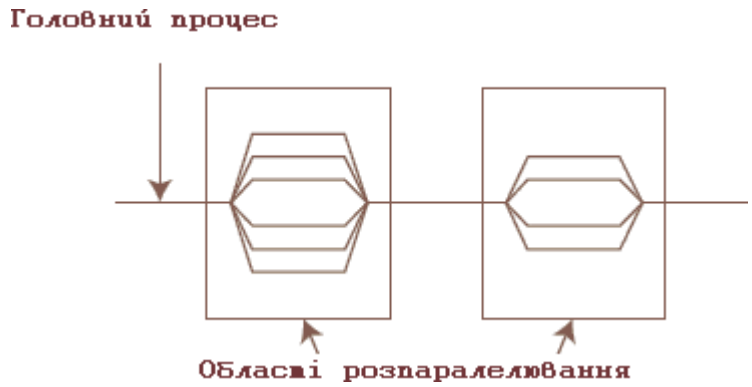
В даний час OpenMP підтримується більшістю розробників паралельних обчислювальних систем: компаніями Intel, Hewlett-Packard, Silicon Graphics, Sun, IBM, Fujitsu, Hitachi, Siemens, Bull і іншими. Багато відомих компаній в галузі розробки системного програмного забезпечення також приділяють значну увагу розробці системного програмного забезпечення з OpenMP. Серед цих компаній відзначимо Intel, KAI, PGI, PSR, APR, Absoft і деякі інші. Значне число компаній і науково-дослідних організацій, які розробляють прикладне програмне забезпечення, в даний час використовує OpenMP при розробці своїх програмних продуктів. Серед цих компаній і організацій відзначимо ANSYS, Fluent, Oxford Molecular, NAG, DOE ASCI, Dash, Livermore Software.

Компілятори gcc і gfortran мають вбудовану підтримку паралелізації OpenMP починаючи з версії 4.2. Для використання цієї можливості слід додати ключ -fopenmp при компіляції. OpenMP версії 3.0 підтримується починаючи з версії gcc 4.4. OpenMP 2.5 підтримується також компілятором Microsoft Visual C ++ 2005 (слід використовувати ключ /openmp) і компіляторами Intel C або Intel Fortran починаючи з версії 10.1 (ключ -openmp).

8.3 Основи OpenMP

Будь-яка програма, послідовна або паралельна, складається з набору областей двох типів: послідовних областей і областей розпаралелювання. При виконанні послідовних областей породжується тільки один головний потік виконання (процес). У цьому потоці ініціюється виконання програми, а також відбувається її завершення. У послідовної програми цей потік є єдиним протягом виконання всієї програми. У паралельній програмі в областях розпаралелювання породжується цілий ряд паралельних потоків. Породжені паралельні потоки можуть виконуватися як на різних процесорах, так і на одному процесорі обчислювальної системи. В останньому випадку паралельні процеси (потоки) конкурують між собою за доступ до процесора. Управління конкуренцією

здійснюється планувальником операційної системи за допомогою спеціальних алгоритмів. В операційній системі Linux планувальник завдань здійснює обробку процесів за допомогою стандартного карусельного (round-robin) алгоритму. При цьому тільки адміністратори системи мають можливість змінити або замінити цей алгоритм системними засобами. Таким чином, у паралельних програмах в областях розпаралелювання виконується ряд паралельних потоків. Принципова схема паралельної програми зображена на рис. 8.1.



Рисуно. 8.1 – Принципова схема паралельної програми

При виконанні паралельної програми робота починається з ініціалізації і виконання головного потоку (процесу), який у міру необхідності створює і виконує паралельні потоки виконання, передаючи їм необхідні дані. Паралельні потоки з однієї паралельної області програми можуть виконуватися як незалежно один від одного, так і з пересиланням та отриманням повідомлень від інших паралельних потоків. Остання обставина ускладнює розробку програми, оскільки в цьому випадку програмістові доводиться займатися плануванням, організацією і синхронізацією посилки повідомлень між паралельними потоками. Таким чином, при розробці паралельної програми бажано виділяти такі області розпаралелювання, в яких можна організувати виконання незалежних паралельних потоків. Для обміну даними між паралельними процесами (потоками) в OpenMP використовуються загальні змінні. При зверненні до загальних змінних у різних паралельних потоках можливе виникнення конфліктних ситуацій при доступі до даних. Для запобігання конфліктів можна скористатися процедурою синхронізації. При цьому треба

мати на увазі, що процедура синхронізації - дуже дорога операція по тимчасових витратах і бажано по можливості уникати її або застосовувати якомога рідше. Для цього необхідно дуже ретельно продумувати структуру даних програми.

Виконання паралельних потоків в паралельній області програми починається з їх ініціалізації. Вона полягає у створенні дескрипторів породжуваних потоків і копіюванні всіх даних з області даних головного потоку в області даних паралельних потоків, що створюються. Ця операція надзвичайно трудомістка - вона еквівалентна приблизно трудомісткості не менше 1000 машинних команд. Ця оцінка надзвичайно важлива при розробці паралельних програм с допомогою OpenMP, оскільки її ігнорування веде до створення неефективних паралельних програм, які виявляються часто повільнішими ніж їх послідовні аналоги. Справді: для того щоб отримати виграш у швидкодії паралельної програми, необхідно, щоб трудомісткість паралельних процесів в областях розпаралелювання програми істотно перевершувала б трудомісткість породження паралельних потоків. В іншому випадку ніякого виграшу за швидкодією отримати не вдасться, а часто можна опинитися навіть і в програші.

Після завершення виконання паралельних потоків управління програмою знову передається головному потоку. При цьому виникає проблема коректної передачі даних від паралельних потоків головного. Тут важливу роль грає синхронізація завершення роботи паралельних потоків, оскільки в силу цілого ряду обставин час виконання навіть однакових за трудомісткістю паралельних потоків непередбачувано (воно визначається як історією конкуренції паралельних процесів, так і поточним станом обчислювальної системи). При виконанні операції синхронізації паралельні потоки, вже завершили своє виконання, простоюють і чекають завершення роботи самого останнього потоку. Природно, при цьому неминуча втрата ефективності роботи паралельної програми. Крім того, операція синхронізації має трудомісткість, порівнянну з трудомісткістю ініціалізації паралельних потоків.

8.4 Директиви OpenMP

У найзагальнішому вигляді формат директив OpenMP може бути представлений у такому вигляді:

#pragma omp <ім'я_директиви> [<параметр>[[,] <параметр>]...]

Початкова частина директиви (#pragma omp) є фіксованою, вид директиви визначається її ім'ям (ім'я_директиви), кожна директива може супроводжуватися довільною кількістю параметрів (англійською мовою для параметрів директиви OpenMP використовується термін clause).

Для ілюстрації наведемо приклад директиви:

#pragma omp parallel default(shared)

\ private (beta, pi)

Приклад показує також, що для завдання директиви може бути використано декілька рядків програми – ознакою наявності продовження є знак зворотного слішу «\».

Для виділення паралельних фрагментів програми слід використовувати директиву parallel:

#pragma omp parallel [<параметр> ...] <блок_програми>

Для блоку (як і для блоків всіх інших директив OpenMP) має виконуватися правило «один вхід - один вихід», тобто передача управління ззовні до блоку та з блоку за межі блоку не допускається.

Директива Parallel є однією з основних директив OpenMP. Правила, що визначають дії директиви, полягають у наступному:

- Директива parallel (основна директива OpenMP).
- Коли основний потік виконання досягає директиви parallel, створюється набір (team) потоків; вхідний потік є основним потоком цього набору (master thread) та має номер 0.
- Код області дублюється або розділяється між потоками для паралельного виконання.
- Наприкінці області забезпечується синхронізація потоків – виконується очікування завершення обчислень всіх потоків; далі всі потоки

завершуються – подальші обчислення продовжує виконувати лише основний потік.

Контрольні питання

1. Яке головне призначення стандарту OpenMP у паралельному програмуванні? З яким типом паралельної архітектури він працює (спільна чи розподілена пам'ять)?
2. Що є базовою моделлю паралелізму, яку використовує OpenMP?
3. Який термін використовується в OpenMP для позначення блоку коду, що виконується паралельно декількома потоками?
4. Опишіть модель «Fork-Join», яка використовується OpenMP. Який механізм відповідає за створення команди потоків (Fork), і що відбувається після виконання паралельної секції (Join)?
5. Яке значення за замовчуванням має змінна, оголошена до паралельного регіону, якщо вона не вказана у директиві \#pragma omp parallel?
6. Поясніть різницю між атрибутами shared та private для змінних у паралельній області OpenMP. Як ці атрибути впливають на безпеку та ефективність доступу до даних?
7. Яка директива OpenMP використовується для гарантування, що блок коду буде виконаний лише одним потоком, що входить до паралельного регіону, без гарантії того, що це буде головний потік?
8. Яку функцію виконує директива reduction у OpenMP?
9. Чому використання директиви barrier може погіршити продуктивність паралельної програми OpenMP?
10. Яка директива є основним будівельним блоком у OpenMP для визначення області, в якій будуть працювати декілька потоків?
11. Що означає, що програма OpenMP використовує ітераційний паралелізм?

Тема 9. Технологія MPI

9.1 Призначення MPI

Комунікаційна бібліотека MPI стала загальновизнаним стандартом у паралельному програмуванні із застосуванням механізму передачі повідомлень. Повний і строгий опис середовища програмування MPI можна знайти в авторському описі розробників [8;9]. На жаль, дотепер немає перекладу цього

документа українською мовою. Пропонований увазі читача опис MPI не є повним, проте містить достатньо матеріалу для написання досить складних програм. Мета даного посібника полягає в тому, щоб, по-перше, ознайомити читача з функціональними можливостями цієї комунікаційної бібліотеки і, по-друге, розглянути набір підпрограм, достатній для програмування будь-яких алгоритмів. Приклади паралельних програм з використанням комунікаційної бібліотеки MPI, наведені в кінці даної частини, перевірені на різних багатопроцесорних системах (Linux-кластері, 2-процесорній системі Xeon).

9.2 Загальна організація MPI

MPI-програма є набір незалежних процесів, кожний з яких виконує свою власну програму (не обов'язково одну й ту ж), написану на мові C або FORTRAN. З'явилися реалізації MPI для C++, проте розробники стандарту MPI за них відповідальності не несуть. Процеси MPI-програми взаємодіють один з одним за допомогою виклику комунікаційних процедур. Як правило, кожен процес виконується у своєму власному адресному просторі, проте допускається і розділення пам'яті. MPI не специфікує модель виконання процесу — це може бути як послідовний процес, так і багатопотоковий. MPI не надає ніяких засобів для розподілу процесів по обчислювальних вузлах і для запуску їх на виконання. Ці функції покладаються або на операційну систему, або на програміста. Зокрема, на кластерах - спеціальна команда mpiexec, яка припускає, що скомпільовані програми вже якимось чином розподілені по комп'ютерах кластера. Описаний стандарт MPI 1.1 не містить механізмів динамічного створення і знищення процесів під час виконання програми. MPI не накладає будь-яких обмежень на те, як процеси будуть розподілені по процесорах, зокрема, можливий запуск MPI-програми з декількома процесами на звичайній однопроцесорній системі.

Для ідентифікації наборів процесів вводиться поняття групи, що об'єднує всі або якусь частину процесів. Кожна група утворює область зв'язку, з яким зв'язується спеціальний об'єкт - комунікатор області зв'язку. Процеси всередині

групи нумеруються цілим числом в діапазоні 0..groupsize-1. Всі комунікаційні операції з деяким комунікатором виконуватимуться тільки всередині області зв'язку, що описується цим комунікатором. У процесі ініціалізації MPI створюється область зв'язку, що містить всі процеси MPI-програми, з якою зв'язується комунікатор MPI_COMM_WORLD. У більшості випадків на кожному процесорі запускається один окремий процес, і тоді терміни “процес” і “процесор” стають синонімами, а величина groupsize стає рівною NPROCS – числу процесорів, виділених завданню. У подальшому обговоренні будемо мати на увазі саме таку ситуацію і не будемо строго дотримуватися термінології.

Отже, якщо сформулювати коротко, MPI - це бібліотека функцій, що забезпечує взаємодію паралельних процесів за допомогою механізму передачі повідомлень. Це достатньо об’ємна і складна бібліотека, що складається приблизно з 130 функцій, до числа яких входять:

- функції ініціалізації і закриття MPI-процесів;
- функції реалізації комунікаційних операцій типу точка-точка;
- функції реалізації колективних операцій;
- функції для роботи з групами процесів і комунікаторами;
- функції для роботи зі структурами даних;
- функції формування топології процесів.

Набір функцій бібліотеки MPI виходить далеко за межі набору функцій, мінімально необхідного для підтримки механізму передачі повідомлень, описаного в першій частині. Проте складність цієї бібліотеки не повинна лякати користувачів, оскільки вся ця безліч функцій призначена для полегшення розробки ефективних паралельних програм. Врешті-решт, користувачу належить право самому вирішувати, які засоби з арсеналу, що надається, використовувати, а які ні. У принципі, будь-яка паралельна програма може бути написана з використанням всього 6-MPI функцій, а достатньо повне і зручне середовище програмування створює набір з 24 функцій.

Кожна з MPI-функцій характеризується способом виконання:

1. Локальна функція – виконується усередині конкретного процесу. Її завершення не вимагає комунікацій.

2. Нелокальна функція – для її завершення потрібне виконання MPI-процедури іншим процесом.

3. Глобальна функція - процедуру повинні виконувати всі процеси групи. Недотримання цієї умови може призводити до зависання завдання.

4. Блокуюча функція - повернення керування з процедури гарантує можливість повторного використання параметрів, що беруть участь у виклику. Ніяких змін у стані процесу, що викликав блокуючий запит, до виходу з процедури не буде.

5. Неблокувальна функція - повернення з процедури відбувається негайно, без очікування закінчення операції і до того, як буде дозволене повторне використання параметрів, що беруть участь у запиті. Завершення неблокувальних операцій здійснюється спеціальними функціями.

Контрольні питання

1. Яке головне призначення стандарту MPI? З яким типом паралельної архітектури він працює (спільна чи розподілена пам'ять)?
2. Яку модель програмування реалізує MPI? Як вона відрізняється від моделі спільної пам'яті (наприклад, OpenMP) з точки зору доступу до даних?
3. Як MPI забезпечує переносимість (портативність) паралельних програм? Що дозволяє програмі, написаній за допомогою MPI, працювати на різних кластерах та суперкомп'ютерах?
4. Що таке комунікатор в MPI, і чому він є фундаментальним елементом організації? Яке призначення має комунікатор MPI_COMM_WORLD?
5. Що таке ранг (rank) процесу в MPI? Що визначає розмір (size) комунікатора, і як ці два параметри використовуються для ідентифікації процесів?
6. Опишіть модель SPMD (Single Program, Multiple Data). Чому ця модель є типовою для програм, написаних за допомогою MPI?
7. Поясніть принципову різницю між блокуючим (MPI_Send, MPI_Recv) та неблокуючим (MPI_Isend, MPI_Irecv) режимами передачі повідомлень. Коли доцільно використовувати неблокуючий режим?
8. Навіщо MPI вимагає від розробника вказувати тип даних (наприклад, MPI_INT, MPI_DOUBLE) при відправці та отриманні повідомлень?
9. Для чого використовується колективна операція MPI_Reduce? Яку проблему (наприклад, обчислення загальної суми) вона вирішує, і чому вона ефективніша, ніж поєднання операцій Send та Recv?
10. Що робить колективна операція MPI_Bcast? Скільки процесів-джерел беруть участь у цій операції?

Тема 10. Розподілене програмування в хмарі

10.1 Моделі хмарного програмування

Розробка хмарних програм (тобто проектування та впровадження програмних систем, які успішно використовують можливості масово розподілених обчислювальних ресурсів) представляє величезні виклики. Труднощі виникають через безліч можливих логічних взаємодій і тимчасових чергувань численних програмних і апаратних компонентів. Помилки програм може бути важко відтворити, а через недетерміновану продуктивність деяких хмарних програм аналіз і обґрунтування поведінки системи може перевищити здібності людини [1-2, 7].

У міру того, як дослідники та практики поступово набули кращого розуміння цієї проблеми, вони розробили моделі програмування та обчислень, які зменшують притаманну складність хмарних систем. Ці моделі, які втілені в програмних/апаратних системах, стоять між розробником і основними обчислювальними ресурсами, надаючи програмісту стилізовані шаблони проектування, відносно простіший спосіб мислення про розподілене програмування та гнучкий інтерфейс до програм, даних і ресурсів.

Сучасне покоління моделей хмарного програмування базується на класичних попередниках, які підтримують міжпроцесний зв’язок на основі спільної пам’яті та передачі повідомлень. Незважаючи на те, що ці попередні версії надають базові можливості для взаємодії між розподіленими завданнями, вони не мають можливості розпаралелювати та автоматично розподіляти завдання та відновлюватися після збоїв. Сучасні нащадки цих версій, включаючи Hadoop MapReduce, Pregel і GraphLab, забезпечують більшу складність і спеціально відповідають вимогам розподіленого програмування та обчислень у хмарних середовищах. Крім інших переваг, ці поточні моделі позбавляють

розробників проблем із багатьма складними аспектами розподіленого програмування та дозволяють програмістам зосередитися на послідовних частинах алгоритмів своїх програм [8-9].

Повторюючи проект Беббіджа 1837 року для першого програмованого комп’ютера, ми відрізняємо сучасні моделі хмарного програмування, називаючи їх механізмами розподіленої аналітики.

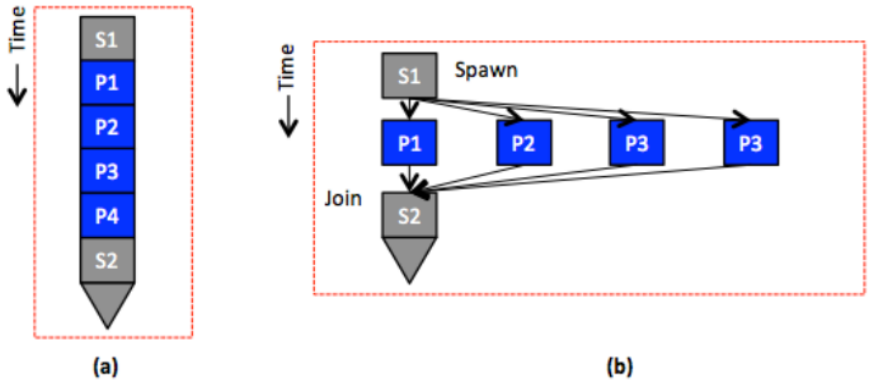


Рисунок 10.1 – (a) Послідовна програма з послідовними (S1) і паралельними (P1) частинами. (b) Паралельна або розподілена програма, яка відповідає послідовній програмі в (a), завдяки якій паралельні частини можуть бути розподілені по машинах або працювати одночасно на одному комп'ютері.

Розподілені програми також знайшли широкі програми за межами науки, такі як пошукові системи, веб-сервери та бази даних. Одним з прикладів є проект Folding@Home, який використовує розподілені обчислення на всіх типах систем, від суперкомп'ютерів, до персональних комп'ютерів для виконання моделювання молекулярної динаміки динаміки білка. Без паралелізації Folding@Home не зможе отримати доступ до майже стільки обчислювальних ресурсів. Наприклад, запуск програми Hadoop MapReduce2 в одному екземплярі віртуальної машини не такий ефективний, як запуск його на великомасштабному кластері екземплярів VM. Звичайно, виконання завдань раніше в хмарі призводить до зниження вартості, що є ключовою метою для хмарних користувачів.

Розподілені програми також допомагають полегшити вузькі місця підсистеми. Наприклад, пристрої вводу-виводу, наприклад диски та картки

мережевого інтерфейсу, зазвичай являють собою основні вузькі місця з точки зору смуги пропускання, продуктивності та/або пропускної здатності. Розподіляючи роботу на різних комп'ютерах, дані можуть надаватися з кількох дисків одночасно, пропонуючи збільшену агрегатну смугу пропускання вводу-виводу, підвищення продуктивності та максимальну пропускну здатність. У підсумку розподілені програми відіграють важливу роль у швидкому вирішенні різних обчислювальних проблем і ефективному пом'якшенні вузьких місць ресурсів. Ця дія підвищує продуктивність, підвищує пропускну здатність і знижує витрати, особливо в хмарі.

10.2 Розподілене програмування в хмарі

Розподілені програми працюють на мережевих комп'ютерах. Мережі комп'ютерів є повсюдними. Інтернет, високопродуктивні обчислювальні кластери (HPC), мобільні телефони та автомобільні мережі, серед інших, представляють поширені приклади. Багато мереж комп'ютерів вважаються розподіленими системами. Ми визначаємо розподілену систему як таку, у якій мережеві комп'ютери спілкуються за допомогою передачі повідомлень і/або спільної пам'яті та координуємо свої дії, щоб вирішити певну проблему або запропонувати певну послугу. Оскільки хмара визначається як набір інтернет-програмного забезпечення, платформи та інфраструктурних служб, які пропонуються через кластер (або кластери) мережевих комп'ютерів (наприклад, центрів обробки даних), хмара , таким чином, є розподіленою системою. Ще одним наслідком нашого визначення є те, що розподілені програми (проти послідовних або паралельних) будуть нормою в хмарах. Зокрема, розподілені програми в розділі визначаються як паралельні програми, які працюють на окремих процесорах на різних комп'ютерах. Таким чином, єдиний спосіб взаємодії завдань у розподілених програмах за допомогою розподіленої системи – надсилання та отримання повідомлень явно або читанням і написанням від/до спільної розподіленої пам'яті, що підтримується базовою розподіленою

системою (наприклад, за допомогою розподіленої спільної пам'яті [DSM] апаратної архітектури).

10.3 Ефективність хмарних програм

Ефективність хмарних програм залежить від способу їх розроблення, реалізації та виконання. Процес розробки має відповідати кільком міркуванням:

– Яка базова модель програмування є найбільш доцільною, передача повідомлень або спільна пам'ять?

– Чи краще програма відповідає синхронній або асинхронній обчислювальній моделі?

– Який найкращий спосіб настроїти дані для обчислювальної ефективності: за допомогою паралелізму даних або паралелізму графіків?

– Яка архітектурно-управлінська структура найбільше підвищить складність програми, ефективність і масштабованість: головний підпорядкований або одноранговий?

Для хмарних програм, зокрема, кілька питань, які охоплюють проектування, впровадження, налаштування та обслуговування, потребують особливої уваги:

– Обчислювальної масштабованості важко досягти у великих системах (наприклад, у хмарах) з кількох причин, включаючи неможливість повністю паралелізувати алгоритми, високу ймовірність дисбалансу навантаження та неминучість синхронізації та накладних витрат на зв'язок.

– Зв'язок, який використовує локалізацію даних і мінімізує мережевий трафік, може бути складним, особливо в (загальнодоступних) хмарах, де зазвичай приховуються мережеві topologies.

– Дві поширені хмарні реалії – віртуальні середовища та різноманітність компонентів центру обробки даних – запроваджують неоднорідність, яка ускладнює планування завдань і маскує апаратні та програмні відмінності між хмарними вузлами.

– Щоб уникнути взаємоблокування та транзитних закриттів і гарантувати взаємовиключний доступ, які є дуже бажаними можливостями в розподілених настройках, базова система повинна забезпечити, і дизайнер повинен використовувати, ефективні механізми синхронізації.

– Оскільки ймовірність відмови збільшується з хмарним масштабом, системні конструкції повинні використовувати механізми відмовостійкості, зокрема стійкість завдань, розподілені контрольні точки та журналювання повідомлень.

– Для ефективного та ефективного виконання планувальники завдань і завдань мають підтримувати контроль за локалізмом завдань, паралелізмом і еластичністю, а також цілями на рівні обслуговування (SLOS).

Вирішення всіх цих міркувань щодо розвитку та хмарних питань накладає велике навантаження на програмістів. Розробка, розробка, перевірка та налагодження всіх (або навіть деяких) цих можливостей є за своєю суттю складними проблемами та може унеможливити значні проблеми з правильністю та продуктивністю, на додачу до значного часу та ресурсів [10].

Сучасні засоби розподіленої аналітики обіцяють полегшити розробникам ці обов'язки. Ці обробники надають інтерфейси програмування програм (API), які дають змогу користувачам представляти свої програми як прості послідовні функції. Потім обробники автоматично створюють, паралелізують, синхронізуються та заплануют завдання та завдання. Вони також обробляють помилки, не вимагаючи залучення користувачів. Наприкінці цієї одиниці ми детально описуємо, як розподілені засоби аналітики ефективно абстрагуються та вирішують проблеми розробки хмарних програм. Однак у наступному розділі ми вперше представляємо дві традиційні моделі розподіленого програмування: спільна пам'ять і передача повідомлень. По-друге, ми обговорюємо обчислювальні моделі, які можуть використовувати хмарні програми. Зокрема, ми пояснюємо синхронні та асинхронні обчислювальні моделі. По-третє, ми представляємо дві основні категорії паралелізму хмарних програм, паралелізм даних і паралелізм графіків. Нарешті, ми описуємо архітектурні моделі, які

зазвичай можуть використовувати хмарні програми: підпорядковані майстру та однорангові архітектури.

Контрольні питання

1. Визначення розподілених систем і позначення зв'язку між розподіленими системами та хмарами
2. Визначення розподілених моделей програмування
3. Опишіть, як завдання можуть спілкуватися за допомогою моделі програмування, що передає повідомлення
4. Структурування різниці між синхронними та асинхронними програмами
5. Пояснення групової синхронної паралельної моделі (BSP)
6. Розрізняти ці розподілені програми: одну програму, кілька даних (SPMD); і кілька програм, кілька даних (MPMD)
7. Два основних методів, які можна включити в розподілені програми, щоб вирішити проблему зв'язку в хмарі
8. Визначення неоднорідних і однорідних хмар і визначення основних причин, які викликають неоднорідність у хмарі.
9. Коли та чому потрібна синхронізація в хмарі.
10. Визначення основного методу, який можна використовувати для переносу несправностей у хмарах.

РЕКОМЕНДОВАНІ ДЖЕРЕЛА ІНФОРМАЦІЇ

1. Ford, N., Richards, M., Sadalage, P., & Dehghani, Z. Software architecture: The hard parts. «O'Reilly Media, Inc.», 2021. 462 с.

2. Heather Adkins, Betsy Beyer, Paul Blankinship, Piotr Lewandowski, Ana Oprea, and Adam Stubblefield. Building Secure and Reliable Systems Published by O’Reilly Media, 2020. 557 с.

3. Home – OpenMP. OpenMP. URL: http://www.openmp.org/ (date of access: 16.05.2025).

4. Khrystynets N., Melnyk K., Lavrenchuk S., Miskevych O., Kostiuchko S. Multiprocessing as a Way to Optimize Queries. Advances in Transdisciplinary Engineering, 2024, №48, pp. 455 – 464 / URL: https://doi.org/10.3233/ATDE231357.

5. Multithreading and Parallel Programming in C#. http://surl.li/mfdjf. (date of access: 16.05.2025)

6. Parallel Patterns Library (PPL). URL: https://cutt.ly/gr1uBNh (date of access: 16.05.2025).

7. Carnell, J., & Sánchez, I. H. (2021). Spring microservices in action. Simon and Schuster.

8. В.М. Мельник, К.В. Мельник, О.І. Кузьмич, Н.В. Багнюк, О.Р. Кравець. Підвищення параметрів швидкодії зв’язку на кластері комунікуючих віртуальних машин. // Програмовані логічні інтегральні схеми та мікропроцесорна техніка в освіті і виробництві: збірник тез міжнародного науково-практичного семінару молодих вчених та студентів (12-13 травня 2020 р.) / відп. ред. П.А. Пех. Луцьк, 2020. С. 41-43.

9. Дослідження покращення внутрішніх та зовнішніх параметрів швидкодії зв’язку на кластері комунікуючих віртуальних машин / В. Мельник та ін. Computer-integrated technologies: education, science, production. 2020. № 39. С. 162–174.

10. Корочкін О.В.. Русанова О.В. Паралельні та розподілені обчислення. Вибрані розділи: Навч. посібник до кредитного модуля «Паралельні та розподілені обчислення» для студентів освітньої програми «Комп’ютерні

системи та мережі», за спеціальністю 123 «Комп’ютерна інженерія» К.: КПІ імені Ігоря Сікорського, 2020. 123 с.

11. Коцовський В. М. К75 Теорія паралельних обчислень: навчальний посібник. Ужгород: ПП «АУТДОР-Шарк», 2021. 188 с.

12. Кузьма К.Т., Мельник О.В. Паралельні та розподілені обчислення: навчальний посібник для вищих закладів освіти. Миколаїв: ФОП Швець В.М., 2020. 172 с.

13. Лисенко В. Ф. Паралельні та розподілені обчислення: навч. посіб. Кропивницький: Видавець, 2021. 153 с.

14. Мельник К., Мельник В., Григоришин А. Автоматичний збір інформації (парсинг) в мережі. Computer-integrated technologies: education, science, production. 2020. № 39. С. 151–156.

15. Наконечна О. А., Ярмоленко Т. А., Алексеєнко В. В., Якимчук Б. М. Інструктивно-методичні рекомендації з дисципліни «Технології розподілених систем та паралельних обчислень. Житомир: Вид-во ЖДУ ім. Івана Франка, 2023. 74 с.

16. Паралельні та розподілені обчислення. Методичні вказівки до виконання самостійної роботи для здобувачів для здобувачів першого (бакалаврського) рівня вищої освіти освітньо-професійної програми «Комп’ютерна інженерія» галузь знань 12 Інформаційні технології спеціальності 123 Комп'ютерна інженерія та 125 Кібербезпека денної та заочної форм навчання / уклад.К.В.Мельник, В.М. Мельник. Луцьк : Луцький НТУ, 2020. 16 с.

17. Паралельні та розподілені обчислення. Методичні вказівки до лабораторних занять для здобувачів першого (бакалаврського) рівня вищої освіти освітньої програми «Комп'ютерна інженерія» галузь знань 12 Інформаційні технології спеціальності 123 Комп'ютерна інженерія денної та заочної форм навчання / уклад. Катерина МЕЛЬНИК. Луцьк: ЛНТУ, 2025. 48 с.

П63 **Паралельні та розподілені обчислення**: конспект лекцій для здобувачів першого (бакалаврського) рівня вищої освіти освітньої програми «Комп'ютерна інженерія» галузь знань 12 (F) Інформаційні технології спеціальності 123 (F7) Комп'ютерна інженерія денної та заочної форм навчання / уклад. К.В. Мельник. Луцьк: ЛНТУ, 2025. 75 с.

Конспект лекцій з дисципліни «**Паралельні та розподілені обчислення**» складений відповідно до діючої програми курсу.

Призначений для здобувачів вищої освіти спеціальності Комп'ютерна інженерія освітньої програми «Комп'ютерна інженерія».

Комп'ютерний набір К.В. Мельник

Редактор К.В. Мельник

Підп. до друку «___» _____ 2025р.
Формат 60х84/16. Папір офс. Гарнітура Таймс.
Ум. друк. арк. ____. Тираж 10 прим. Зам. ____

Відділ іміджу та промоцій
Луцького національного технічного університету
43018, м. Луцьк, вул. Львівська, 75