МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Київський національний університет будівництва і архітектури

О. Л. Соловей

ТЕХНОЛОГІЯ РОЗПОДІЛЕНИХ СИСТЕМ ТА ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

Конспект лекцій
для здобувачів першого (бакалаврського) рівня вищої освіти
за спеціальністю 122 Комп’ютерні науки

Київ 2024

УДК 681.3.06.(075)
Т

Затверджено на засіданні кафедри інформаційних технологій, протокол № 11 від 20 травня 2024 року.

В авторській редакції.

Соловей О.Л.

Т Технології розподілених систем та паралельних обчислень: конспект лекцій / Уклад. О. Л. Соловей. – Київ: КНУБА, 2024. – 98 с.

Розглянуто основні принципи технологій розподілених систем та паралельних обчислень.

Призначено для здобувачів спеціальності 122 «Комп’ютерні науки»

УДК 681.3.06.(075)
Т

Зміст

Вступ

Технології розподілених систем та паралельних обчислень є важливим елементом сучасної інформаційної технології, що дозволяє ефективно використовувати ресурси комп'ютерної системи для вирішення складних завдань. Ця дисципліна охоплює різноманітні концепції, алгоритми та практичні методи, спрямовані на підвищення продуктивності, масштабованості та надійності обчислювальних систем.

Метою вивчення технологій розподілених систем та паралельних обчислень є розуміння принципів побудови, розвитку та застосування розподілених та паралельних систем, а також набуття практичних навичок у розробці програмного забезпечення для таких середовищ.

У процесі вивчення цієї дисципліни здобувачі ознайомляться з рядом ключових понять:

1. Розподілені системи: системи, що складаються з різних компонентів, розташованих на різних комп'ютерах та взаємодіють між собою через мережу.
2. Паралельні обчислення: обчислення, що виконуються одночасно на кількох обчислювальних пристроях з метою підвищення швидкості виконання завдань.
3. Кластерні системи: група комп'ютерів, які спільно працюють для виконання обчислювальних завдань.
4. Системи розподіленого зберігання: системи, які забезпечують доступ до даних через мережу, дозволяючи ефективно зберігати та обробляти великі обсяги інформації.

Технології розподілених систем та паралельних обчислень застосовуються у різних галузях, включаючи наукові дослідження, фінансову аналітику, інтернет-сервіси, медичні додатки та багато інших. Вони дозволяють розв'язувати складні задачі швидко, ефективно та надійно, що робить їх невід'ємною частиною сучасного інформаційного середовища.

У наступних лекціях розглянено основні принципи побудови розподілених та паралельних систем, методи розподіленого програмування, а також практичні приклади використання цих технологій. Вивчення цієї дисципліни дозволить краще розуміти сучасні обчислювальні системи та забезпечить необхідні навички для роботи у сфері інформаційних технологій.

Лекція 1.
Поняття про паралельні та розподілені обчислення

1.1. Послідовні обчислення

У процесі розробки та використання програмного забезпечення (ПЗ) для *послідовних обчислень* (*serial computation*):

- Задача розбивається на дискретну послідовність інструкцій (операторів).
- Інструкції виконуються послідовно одна за одною.
- Код програми виконується на єдиному процесорі (single processor).
- В кожний момент часу може виконуватися тільки одна інструкція.

Схема послідовних обчислень наведена на рис. 1.1.

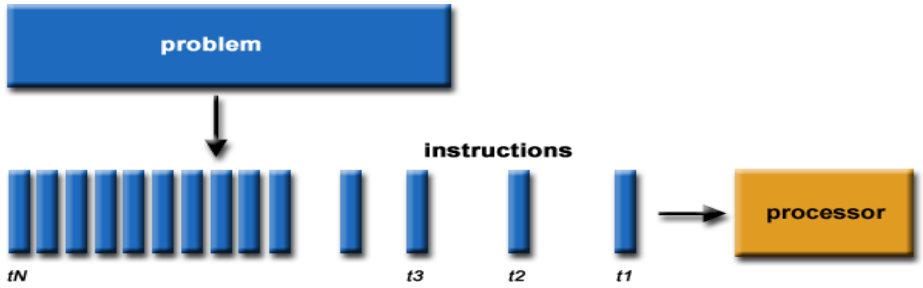


Рисунок 1.1. Схема послідовних обчислень

Приклад 1.1. Розглянемо процедуру нарахування заробітної плати працівників підприємства з використанням послідовного підходу. Відповідна схема наведена на рис. 1.2.

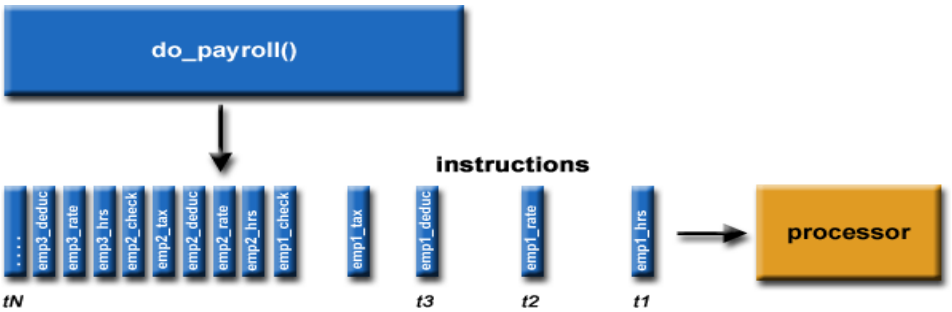


Рисунок 1.2. Схема обчислення зарплати

1.2. Паралельні обчислення

Паралельні обчислення (ПО) — одночасне використання кількох ресурсів ЕОМ для розв’язування обчислювальних задач:

- Задача розбивається на підзадачі, які можуть виконуватися у один і той самий момент часу.
- Кожна підзадача в свою чергу розбивається на послідовність інструкцій.
- Інструкції кожної підзадачі виконуються одночасно на різних процесорах.
- У процесі обчислень використовується загальний механізм контролю- координації.

Схема паралельних обчислень наведена на рис. 1.3. Обчислювальна задача має:

- Допускати розбиття на незалежні підзадачі, які можна виконувати одночасно (допускати паралелізм).
- З використанням кількох обчислювальних ресурсів, працюючих паралельно, розв'язуватися за коротший проміжок часу, ніж з використанням одного процесора.

Паралелізм — це сукупність математичних, алгоритмічних, програмних і апаратних засобів, що забезпечують можливість паралельного виконання задачі.

Типові обчислювальні ресурси, які використовують у ПО:

- Один комп'ютер з кількома процесорами.
- Довільна кількість комп'ютерів з'єднаних у мережу.

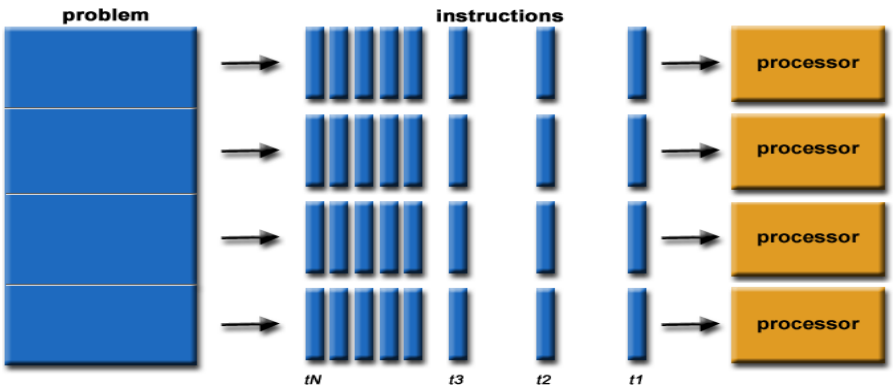


Рисунок 1.3. Схема паралельних обчислень

Приклад 1.2. Розглянемо процедуру нарахування заробітної плати працівників підприємства з використанням паралельного підходу. Відповідна схема наведена на рис. 1.4.

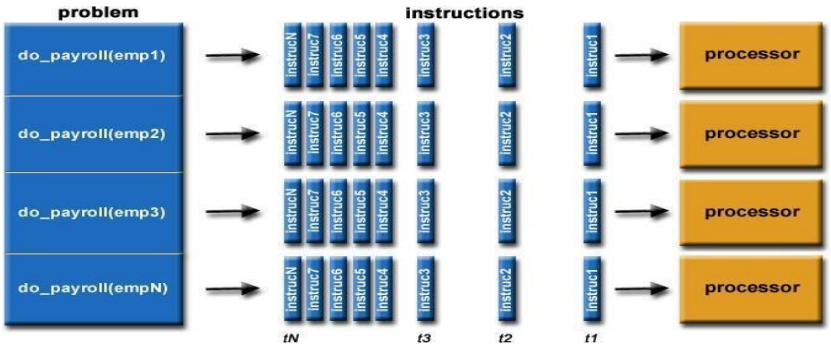


Рисунок 1.4. Схема паралельного обчислення зарплати

1.3. Засоби для здійснення паралельних обчислень

Засоби, які дають змогу втілити парадигму паралелізму, можна класифікувати таким чином:

- Апаратні:

✓ засоби для проведення обчислень (обчислювальна техніка):

✓ обчислювальна техніка, зібрана з стандартних комплектуючих;

✓ обчислювальна техніка, зібрана з спеціальних комплектуючих;

✓ засоби візуалізації;

✓ засоби для зберігання і обробки даних.

- Програмні:

✓ програмні засоби загального призначення (операційні системи, стандартні бібліотеки, мови програмування, компілятори, і т. п.);

✓ спеціальні програмні засоби: бібліотеки (PVM, MPI); засоби об’єднання ресурсів (Dynamite, Globus та інші).

1.4. Паралельні комп’ютери

Паралельні комп’ютери поділяються на:

Сучасні автономні комп’ютери. Вони є паралельними з точки зору їх внутрішньої будови: Наявність численних функціональних модулів (L1-кеш, L2-кеш, пристрої попереджувального вибору команд (prefetch), decode, floating-point, графічні процесори (GPU), модулі для цілої та дійсної арифметики тощо); Багатоядерність чи багатопроцесорність; Підтримка використання потоків (Multiple hardware threads). Кластери, які складаються із кількох робочих станцій, з’єднаних мережею (див рис. 1.5).

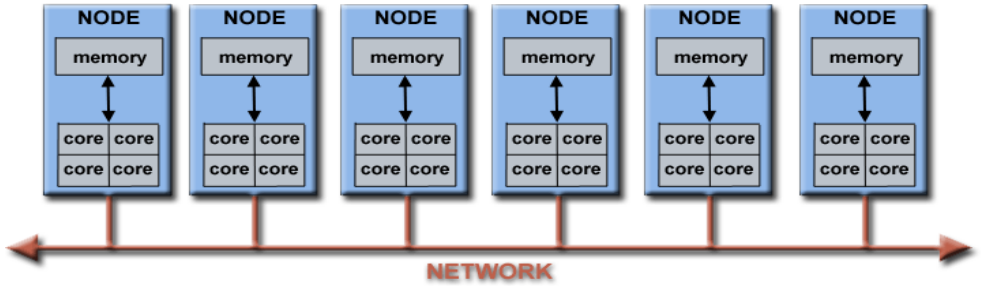


Рисунок 1.5. Схема кластера

Наприклад, на рис. 1.6 наведено схему типового кластера для паралельних обчислень, які використовуються в Lawrence Livermore National
Laboratory.

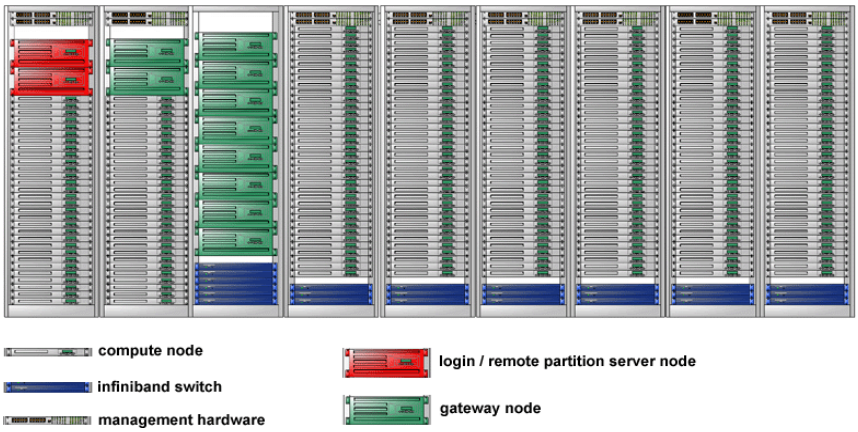


Рисунок 1.6. Схема будови кластера LLNL

Кожний обчислювальний вузол сам є багатопроцесорним комп'ютером. Вузли кластера з'єднані у Infiniband мережу. Кластер містить багатопроцесорні вузли спеціального призначення, які не використовуються для обчислень Дані про найбільші у світі паралельні комп'ютери (суперкомп'ютери), пере- важна більшість яких є кластерами, наведено на сайть *Top500.org*.

1.5. Актуальність використання паралельних обчислень

Важливість дослідження та використання ПО зумовлена наступними при- чинами:

а) *Явища у реальному світі відбуваються паралельно*.

Тому ПО є значно більш придатними для моделювання складних

взаємопов’язаних об’єктів, явищ, систем та процесів порівняно із послідовними обчисленнями. Приклади деяких таких об’єктів наведені на рисунку1.7.

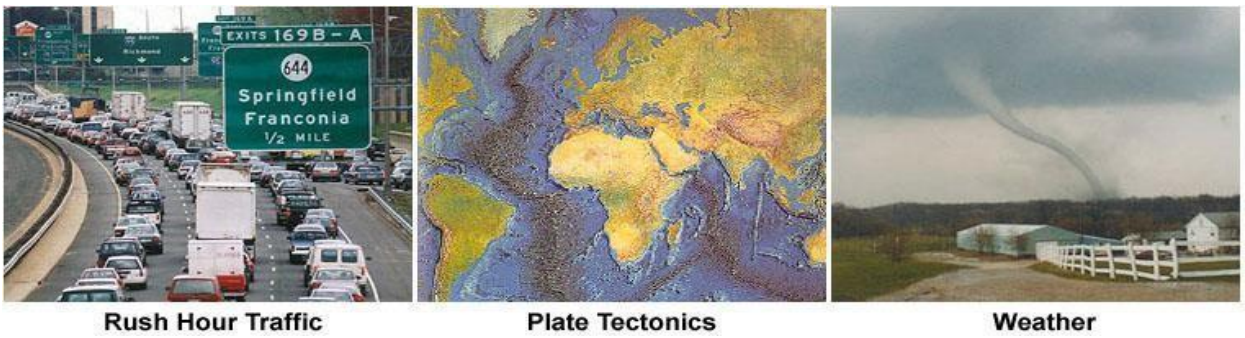
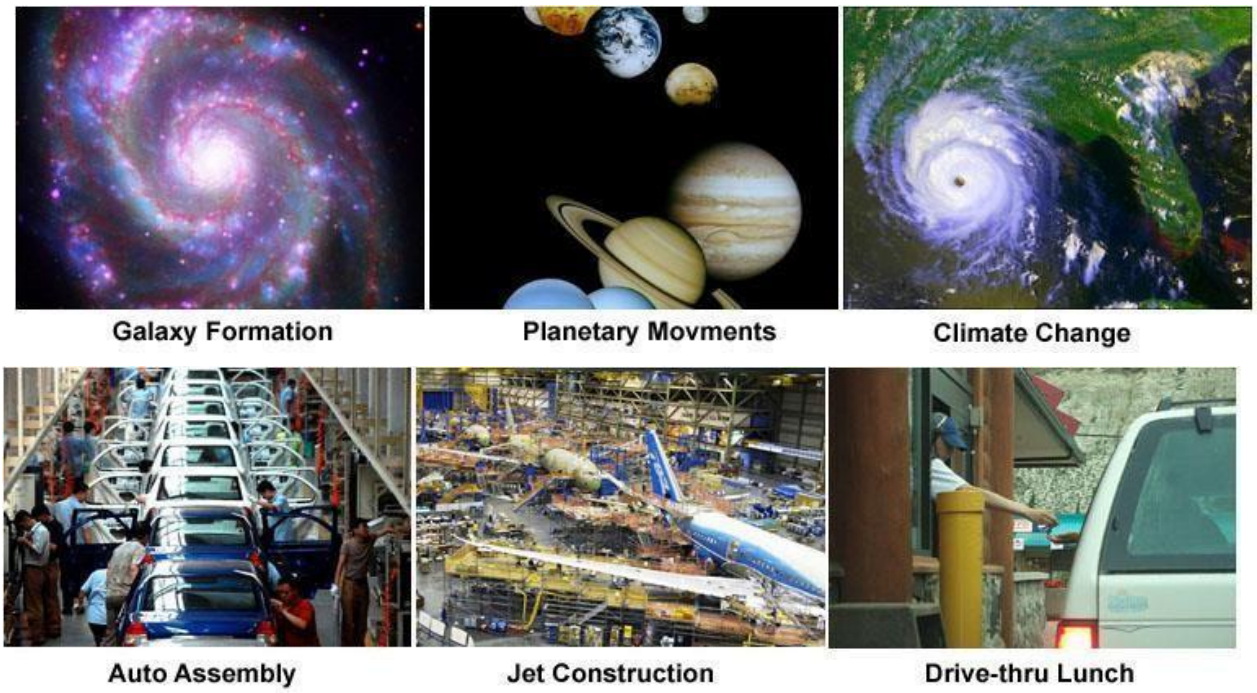


Рисунок 1.7. Складні процеси, моделювання яких вимагає застосування



паралельних обчислень

б) Економія часу та грошей. Використання значних обчислювальних ресурсів може пришвидшити процес знаходження розв’язків складних задач, причому виграш від оперативності часто перевищує вартість використання додаткових ресурсів.

в) Паралельні комп’ютери можуть бути побудовані із дешевих зручних компонентів.

г) Можливість знаходження розв’язків складних практичних та теоретичних задач. Добре відомо, що багато практично важливих задач мають настільки велику розмірність, що практично нереально їх розв’язати на одному персональному комп’ютері, особливо із обмеженою пам’яттю.

Прикладом таких задач є задачі із переліку "Grand Challenge Problems" (en.wikipedia.org/wiki/Grand_Challenge), які вимагають використання PetaFLOPS і PetaBytes комп'ютерних ресурсів.

Ще один приклад: Web-пошуковики, які виконують мільйони транзакцій що секунди, та задачі прогнозу погоди. Область розв'язку (атмосфера) розбивається на окремі просторові зони, причому для розв'язку часових змін обчислень в кожній зоні повторюється багато разів. Якщо об'єм зони рівний 1 км^3, то для моделювання 10 км шару атмосфери необхідно 5 х 10^8 таких зон. Припустимо, що обчислення в кожній зоні вимагає 200 операцій з плаваючою крапкою, тоді за один часовий крок необхідно виконати 10^{11} операцій з плаваючою крапкою. Для того, щоб провести розрахунок прогнозу погоди з передбаченням на 10 днів з 10-ти хвилинним кроком в часі, ЕОМ продуктивністю 100 Mflops (10^8 операцій з плаваючою крапкою за секунду) необхідно 10^7 секунд чи понад 100 днів. Для того, щоб провести розрахунок за 10 хв, необхідна ЕОМ продуктивністю 1.7.

д) Забезпечення одночасності багатьох дій (concurrency). Наприклад, корпоративні мережі дають можливість одночасно виконувати роботу багатьом працівникам.

е) Використання глобальних ресурсів: Використання ресурсів LAN або Internet у випадку, коли локальні обчислювальні ресурси недостатні.

є) Краще використання паралельного апаратного забезпечення:

Сучасні ПЕОМ та гаджети мають паралельну багатоядерну архітектуру. Паралельне ПЗ значно краще використовує можливості цих пристроїв у зв'язку із врахуванням багатопотоковості та багатоядерності, ніж програми, розроблені у межах «послідовного підходу».

1.6. Перспективи використання паралельних обчислень

Тренд останніх 20 років, який полягає у використанні швидких мереж, багатопроцесорних архітектур та розподілених архітектур, ясно вказує на те, що *майбутнє в технологіях комп'ютерних обчислень за паралелізмом.*

На рис. 1.8 можна переконатися, що за цей період спостерігається зростання продуктивності суперкомп'ютерів понад у 500000 разів. Є сподівання, що процес зростання продуктивності не буде значно сповільнятися у майбутньому.

Зараз мова йде про Exascale-обчислення (Exascale Computing), де Exaflop = 10^{18} обчислень у секунду.

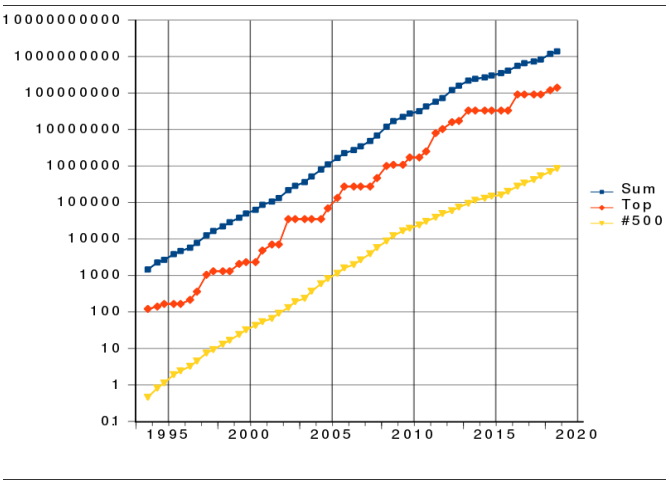


Рисунок 1.8. Зростання продуктивності суперкомп'ютерів (у Teraflops за даними Top500.org)

Станом на листопад 2020 року рейтинг Top500 мав такий вигляд (за тестом LINPACK benchmarks):

Fugaku (Японія) 442 петафлопс, 7 630 848 ядер, 5 087 232 Гб

Summit (США) — 144,8 петафлопс

Sierra (США) — 94,6 петафлопс

Sunway TaihuLight (КНР) — 93,0 петафлопс

Selene (США) — 63,4 петафлопс

Tianhe-2A (КНР) — 61,4 петафлопс.

Summit — суперкомп'ютер розроблений IBM, Nvidia та Mellanox для національної лабораторії Оук-Ридж, який у 2018 р. став найпотужнішим комп'ютером у світі. 4608 серверів IBM Power Systems AC922 суперкомп'ютера займають площу, еквівалентну площі двох тенісних кортів. До складу цих серверів входять 9 тисяч 22-ядерних процесорів IBM POWER9 і більше 27 тисяч графічних про- цесорів NVIDIA Tesla V100. Загальний об'єм зовнішньої пам'яті — 250 PB. Summit споживає 15 МВт енергії, якої вистачило б на постачання 8100 середньо- статистичних житлових будинків. Summit — перший комп'ютер, який подолав межу exaop (досягши 1,88 під час розв'язування задачі аналізу генома). Дослідники використовують Summit для розв'язування задач космології, медицини та кліматології. Сумарна вартість Summit та Sierra — 325 мільйонів доларів.

1.7. Сфери застосування паралельних обчислень

а) Наука та інженерія. Історично, паралельні обчислення розглядалися

як засіб для моделювання складних явищ та розв'язування різноманітних громіздких у таких галузях:

- науки про природу — моделювання атмосферних явищ, дослідження за- бруднення оточуючого середовища, задачі геології та сейсмології;
- фізика, астрономія та інженерія — розв'язування задач прикладної та ядерної фізики, електротехніки, фізики елементарних частинок, матеріалознавства, нанотехнологій, мікроелектроніки;
- біологія, хімія та медицина — дослідження у галузях біотехнологій, генетики, екології, популяційної біології, вірусології, фармакології та молекулярної хімії;
- математика та комп'ютерні науки.

б) Індустрія та комерція. Індустріальні та комерційні застосування є основним драйвером розвитку інформаційних технологій. Паралельні обчислення використовуються при розв'язуванні задач:

- розвідування корисних копалин;
- фінансового та економічного моделювання;
- обробки великих БД, Big Data, Data Mining;
- Web-пошуку, web-торгівлі;
- обробки медичних зображень та діагностування;
- обробки графіки, мультимедіа та віртуальна реальність.

2.1. Класифікація паралельності за рівнями

Класифікація паралельності за рівнями, що відрізняються показниками абстрактності розпаралелювання задач, наведена в табл. 2.1. Чим «глибше» рівень, який має ознаки паралельності, тим детальнішим буде розпаралелювання, що стосується елементів програми (інструкція, елементи інструкції тощо). Чим вище розміщено рівень абстракції, тим більші блоки має паралельність.

Кожний рівень має різні аспекти паралельної обробки. Методи і конструктиви рівня обмежуються тільки самим рівнем і не можуть бути поширені на інші рівні. Найбільший інтерес становить рівень процедур (великоблокова, асинхронна паралельність) та рівень арифметичних виразів (малоелементна, детальна або масивна синхронна паралельність).

Таблиця 2.1.

Рівні паралельності

Рівні	**Об'єкт обробки**	**Приклад системи**
Програмний	Робота/Задача	Мультизадачна ОС
Процедурний	Процес	MIMD-система
Рівень формул	Інструкція	SIMD-система
Біт-рівень	В межах інструкції	Машина фон Неймана

Кожний рівень має різні аспекти паралельної обробки. Методи і конструктиви рівня обмежуються тільки самим рівнем і не можуть бути поширені на інші рівні. Найбільший інтерес становить рівень процедур (великоблокова, асинхронна паралельність) та рівень арифметичних виразів (малоелементна, детальна або масивна синхронна паралельність).

Програмний рівень

На цьому найвищому рівню одночасно (або принаймні розподілено за часом) виконуються комплектні програми (рис. 2.1). Машина, що виконує ці програми, *не повинна бути паралельною ЕОМ*, досить того, що в ній наявна багатозадачна операційна система (наприклад, реалізована як система *розподіленого часу).* В цій системі кожному користувачеві відповідно до його пріоритету *планувальник* (scheduler) виділяє відрізок процесорного часу різної тривалості. Користувач одержує ресурси центрального процесорного блоку тільки впродовж короткого часу, а потім стає в чергу на обслуговування.

У тому випадку, коли в ЕОМ недостатня кількість процесорів для всіх користувачів (або процесів), що, як правило, найбільш імовірно, в системі моделюється паралельне обслуговування користувачів за допомогою «квазіпаралельних» процесів.

Рисунок 2.1. Паралельність на програмному рівні

Рівень процедур

На цьому рівні різні частини однієї і тієї самої програми мають виконуватися паралельно. Ці частини називаються «*процесами*» і приблизно відповідають послідовним процедурам. При проектуванні ПЗ задачі розбиваються на майже незалежні одна від інших частини так, щоб по можливості рідше виконувати операції обміну даними між процесами, які потребують відносно великих витрат часу. Цей рівень паралельності ні в якому разі не обмежується розпаралелюванням послідовних програм. Існує велика кількість задач, які потребують паралельних структур цього типу навіть тоді, коли так само, як і на програмному рівні, у користувача є тільки один процесор.

Основне застосування процедурного рівня — загальна паралельна обробка інформації, де застосовується поділ задач на паралельні підзадачі, які розв'язуються на багатопроцесорній системі з метою підвищення обчислювальної продуктивності. Відповідний приклад наведено на рис. 1.4.

Рівень формул

Арифметичні вирази виконуються паралельно покомпонентно, причому в значно простіших синхронних методах. Якщо, наприклад, йдеться про додавання 2x2-матриць, то воно синхронно розпаралелюється дуже просто тому, що кожному процесору відповідає один елемент матриці-результату.

При застосуванні n^2 процесорних елементів можна одержати суму двох матриць порядку n за час, необхідний для виконання однієї операції додавання (за винятком часу, потрібного на зчитування та запис даних). Цьому рівню притаманні засоби векторизації та так званої *паралельності даних*.

Рівень двійкових розрядів

На рівні розрядів (instruction-level parallelism) відбувається паралельне виконання двійкових операцій в межах одного машинного слова. Паралельність на рівні бітів присутня в будь-якому мікропроцесорі. Наприклад, у 64-розрядному арифметико-логічному пристрої побітова обробка виконується паралельними апаратними засобами.

2.2. Класифікація паралельних обчислювальних систем

З попереднього матеріалу зрозуміло, що існує багато різних способів організації паралельних обчислювальних систем. Серед найбільш розповсюдженої архітектури можна вказати векторноконвеєрні, масивно-

паралельні та матричні системи, спецпроцесори, кластери, комп'ютери із багатопотоковою архітектурою тощо.

У зв'язку з різноплановістю розроблених систем виникла потреба класифікувати паралельні системи.

Класифікація Флінна

Ця класифікація архітектур була запропонована в 1966 р. М. Флінном і вважається першою і найбільш розповсюдженою класифікацією. Класифікація Флінна заснована на понятті потоку, під яким мається на увазі послідовність команд або даних, які опрацьовує процесор. На основі кількості потоків команд та даних Флінн вирізняє чотири класи архітектури.

SISD (Single Instruction stream / Single Data stream) — одиничний потік команд та одиничний потік даних, наведений на рис. 2.2 (ПР — процесор, ПД — пам'ять даних, УУ — пристрій керування).

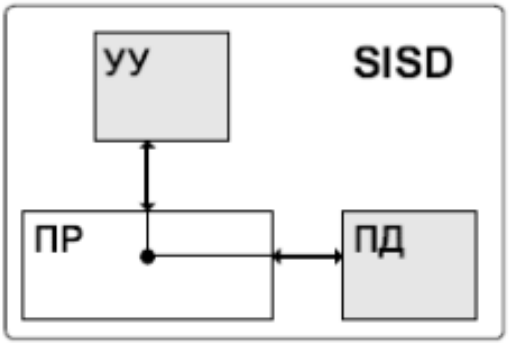


Рисунок 2.2. Клас SISD класифікації Флінна

До класу SISD належать, передусім, класичні послідовні машини із архітектурою фон Неймана, наприклад, PDP-11 або VAX 11/780. В таких машинах є тільки один потік команд, усі команди обробляються послідовно одна за одною і кожна з них породжує одну скалярну операцію. При цьому не важливо, що для збільшення швидкості обробки команд і швидкості арифметичних операцій може бути застосована конвеєрна обробка даних.

SIMD (Single Instruction stream / Multiple Data stream) — одиничний потік команд та множинний потік даних (рис. 2.3).

У подібній архітектурі зберігається один потік команд, який включає, на відміну від попереднього класу, векторні команди. Це дає змогу виконувати арифметичні операції відразу з багатьма даними, наприклад елементами вектора. Спосіб виконання строго не фіксується. Він може бути реалізований або з використанням процесорної матриці, як у ILLIAC IV, або

за допомогою конвеєра, як у машині Cray-1.

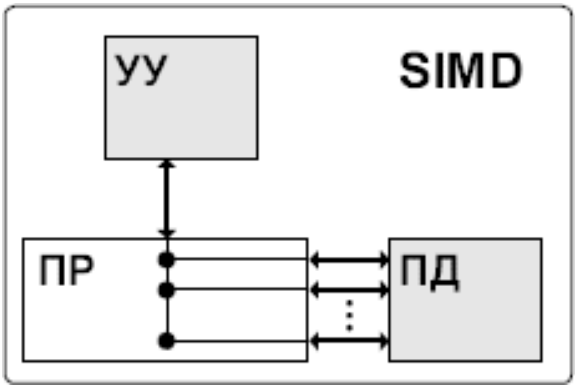


Рисунок 2.3. Клас SIMD класифікації Флінна

MISD (Multiple Instruction stream / Single Data stream) — множинний потік команд і одиночний потік даних (рис. 2.4).

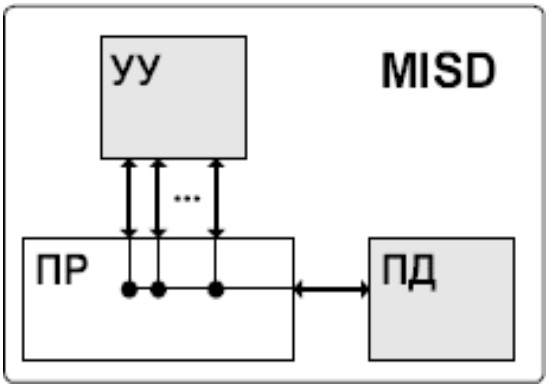


Рисунок. 2.4. Клас MISD класифікації Флінна

У визначенні мають на увазі, що наявність у архітектурі багатьох процесорів, які опрацьовують один і той самий потік даних.

MIMD (Multiple Instruction stream / Multiple Data stream) — множинний потік команд та множинний потік даних (див. рис. 2.5). Цей клас містить обчислювальні системи, які мають кілька пристроїв обробки даних. Він є надзвичайно широким і, зокрема, містить різноманітні мультипроцесорні системи: Cm*, C.mmp, Cray Y-MP, Intel Paragon тощо. Якщо конвеєрну обробку розглядати як виконання послідовності різних команд (стадій конвеєра) не над одиночно-векторним потоком даних, а над множинним скалярним потоком, то усі векторно-конвеєрні комп’ютери можна віднести до класу MIMD.

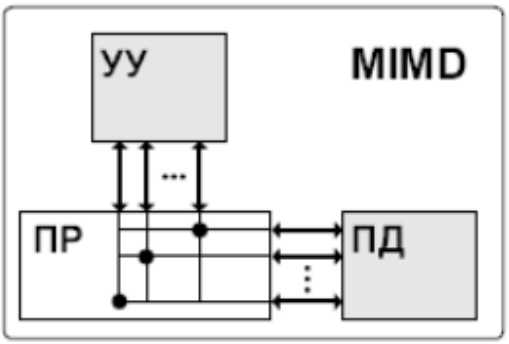


Рисунок. 2.5. Клас MIMD класифікації Флінна

Недоліком класифікації Флінна є те, що деякі важливі системи, наприклад dataflow та векторноконвеєрні машини, чітко не вписуються у дану класифікацію. Інший недолік — надмірна наповненість останнього класу MIMD. Цей недолік подолано у класифікації Р. Хокні, який провів більш ретельну класифікацію машин класу MIMD.

Класифікація Фенга

Принципові інший підхід до класифікації був запропонований Т. Фенгом у 1972 році. Згідно до цього підходу класифікація проводиться по двом простим характеристикам. Перша — число *n* бітів у машинному слові, які опрацьовуються паралельно при виконанні машинних інструкцій. Майже для усіх сучасних машин це число співпадає із довжиною машинного слова. Друга характеристика рівна числу слів *m*, які одночасно обробляє дана обчислювальна система.

Кожну обчислювальну систему можна описати парою чисел (*n*, *m*). Добуток визначає інтегральну характеристику потенціалу обчислювальної системи, яку Фенг назвав максимальною ступеню паралелізму обчислювальної системи. По суті, це не що інше, як пікова продуктивність, виражена у інших одиницях.

Покажемо обчислення характеристик Фенга на прикладі комп’ютера Advanced Scientific Computer фірми Texas Instruments (TI ASC). У основному режимі він обробляє 64-розрядне слово, причому усі розряди опрацьовуються паралельно. Арифметично-логічний пристрій має чотири одночасно працюючих 8-стадійних конвеєрів. При такій організації 4x8=32 слова можуть оброблятися одночасно, і отже комп’ютер TI ASC може бути поданий у вигляді (64, 32).

На основі запропонованої Фенгом класифікації можна відокремити чотири класи комп’ютерів:

1. Розрядно-послідовні, послівно-послідовні (n=1, m=1). У кожний момент часу на таких машинах обробляється тільки один двійковий розряд.

2. Розрядно-паралельні, послівно-послідовні (n>1, m=1). Більшість класичних послідовних комп'ютерів, так само як багато обчислювальних систем з описами (16,1) або (32,1) , які існували до ери багатоядерних машин.

3. Розрядно-послідовні, послівно-паралельні (n = 1, m > 1). Обчислювальні системи цього класу складаються із великої кількості однорозрядних процесорних елементів, кожний з яких працює незалежно від інших. Типовим прикладом є ICL DAP (1, 4096).

4. Розрядно-паралельні, послівно-паралельні (n > 1, m > 1). Переважна більшість паралельних обчислювальних систем, які опрацьовують n×m двійкових розрядів.

Запитання для самоперевірки

1. Дайте визначення поняттям послідовні та паралельні обчислення.
2. Назвіть апаратні та програмні засоби для здійснення паралельних обчислень.
3. Опишіть схему будови кластера Lawrence Livermore National Laboratory.
4. Назвіть сфери застосування паралельних обчислень.
5. Які задачі із переліку "Grand Challenge Problems" вимагають використання PetaFLOPS і PetaBytes комп'ютерних ресурсів?
6. Які рівні паралельності вам відомі? Які аспекти паралельної обробки на кожному рівні?
7. Назвіть для яких алгоритми варто використовувати паралельність на рівні розрядів (instruction-level parallelism)?
8. Дайте визначення класу SISD за класифікацією Фліна. Які складові включені в клас SISD?
9. Дайте визначення класу SIMD за класифікацією Фліна. Які складові включені в клас SIMD?
10. Дайте визначення класу MISD за класифікацією Фліна. Які складові включені в клас MISD?
11. Назвіть класи комп'ютерів за класифікацією Фенга.

Лекція 2.
Способи обробки даних в обчислювальних системах

Послідовна обробка даних

Припустимо, що потрібно знайти суму **c** двох векторів **a** та **b**, кожний з яких має 100 дійсних координат, з використанням обчислювального пристрою (або комп'ютера), який виконує додавання пари чисел за 5 тактів роботи і у процесі обчислень комп'ютер не може виконувати ніяких інших корисних дій. У таких умовах сума векторів може бути знайдена за 500 тактів.

Тепер припустимо, що є два така самі пристрої, які можуть працювати одночасно і незалежно один від іншого, і при цьому відсутні додаткові витрати ресурсів по отриманню пристроями вхідних даних та збереженням результатів. В такому випадку можна отримати шукану суму векторів вже за 250 тактів — тобто маємо подвійне прискорення.

У випадку використання 10 однакових пристроїв результат отримується за 50 тактів, а у загальному випадку система із N пристроїв витратить на обчислення суми приблизно $500 / N$ тактів.

Конвеєрна обробка даних

Розглянемо шляхи покращення ефективності роботи системи із попереднього параграфу. Для цього можна використати форму запису дійсних чисел в пам'яті комп'ютера. Додавання чисел пов'язане виконанням таких мікрооперацій як порівняння та вирівнювання порядків, додавання мантис, нормалізація та т. п. Суттєвим є те, що у процесі обробки кожна мікрооперація задіяна тільки один раз і завжди у тій самій послідовності одна за іншою. Це означає, що якщо перша мікрооперація виконала свою роботу і передала результат другій, то для обробки поточної пари дійсних чисел вона більше не знадобиться, і отже, цілком може бути використання для обробки наступної пари аргументів.

Виходячи із попередніх міркувань, можна сконструювати пристрій наступним чином. Кожну мікрооперацію виділимо у окрему частину пристрою і розташуємо їх у порядку виконання. Після виконання першої мікрооперації перша частина передає свій результат другій частині, а сама отримує для обробки нову пару. Коли вхідні аргументи пройдуть через усі етапи обробки, на виході пристрою з'явиться результат виконання операції.

Такий спосіб організації обчислень має назву *конвеєрної обробки*. Кожна частина пристрою називається *стадією конвеєра*, а загальна

кількість стадій — *довжиною конвеєра*.

Сучасні процесори мають багатоетапні конвеєри команд. Кожен етап (стадія) конвеєра відповідає іншій дії, що виконує процесор. Класичним прикладом процесора з конвеєром є процесор архітектури RISC (Reduced Instruction Set Computing), що має п'ять етапів: завантаження інструкції, декодування інструкції, виконання, доступ до пам'яті, та запис результату.

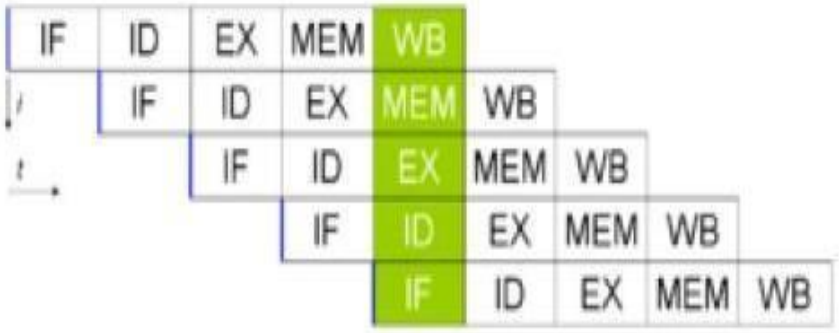


Рисунок 3.1. Стандартний п’ятикроковий конвеєр в машині RISC.

IF (Instruction Fetch), ID (Instruction Decode), EX (Execute), MEM (Memory access), WB (Register write back)

Процесор Pentium 4 використовує конвеєрну архітектуру з 20 стадіями. Основні стадії включають:

Prefetch/Fetch: Команда читається з кеш-пам'яті інструкцій або з основної пам'яті.

Decode: Інструкція перетворюється у мікро-операції.

Rename/Register Allocation: Відбувається перейменування регістрів для уникнення перекладу.

Scheduler/Dispatch: Розподіл мікро-операцій по виконавчим установкам.

Execute: Мікро-операція виконується.

Writeback: Результати виконання записуються назад у регістр.

Commit/Retirement: Остаточна стадія, де інструкції фіксуються як виконані. Цей процес може вплинути на продуктивність. Наприклад, якщо завдання не може бути розділені на незалежні частини, які можуть бути виконані одночасно, всі переваги конвеєрної обробки будуть втрачені. Проте в цілому конвеєрна архітектура дозволяє процесору працювати ефективніше, розподіляючи завдання по різних стадіях.

Припустимо, що для виконання операції додавання дійсних чисел спроектовано конвеєрний пристрій, який складається із п’яти стадій, які

спрацьовують за один такт. Час виконання операції на конвеєрному пристрої рівний сумі часів спрацьовування усіх стадій конвеєра. Це означає, що одна операція додавання двох чисел триває п’ять тактів, тобто так само довго, як і на послідовному пристрої у попередньому прикладі. Тепер розглянемо процес додавання двох векторів (рис. 3.2).

Як і раніше, через п’ять тактів отримано суму елементів першої пари. Проте слід зазначити, що поряд із першою парою пройшли часткову обробку (на різній кількості стадій) і інші пари аргументів. Кожний наступний такт на виході конвеєрного пристрою буде з’являтися сума чергової пари координат вектора **c**. На виконання усієї операції знадобиться 104 такти, замість 500 тактів при використанні послідовних пристроїв — виграш у часі приблизно у п’ять разів.

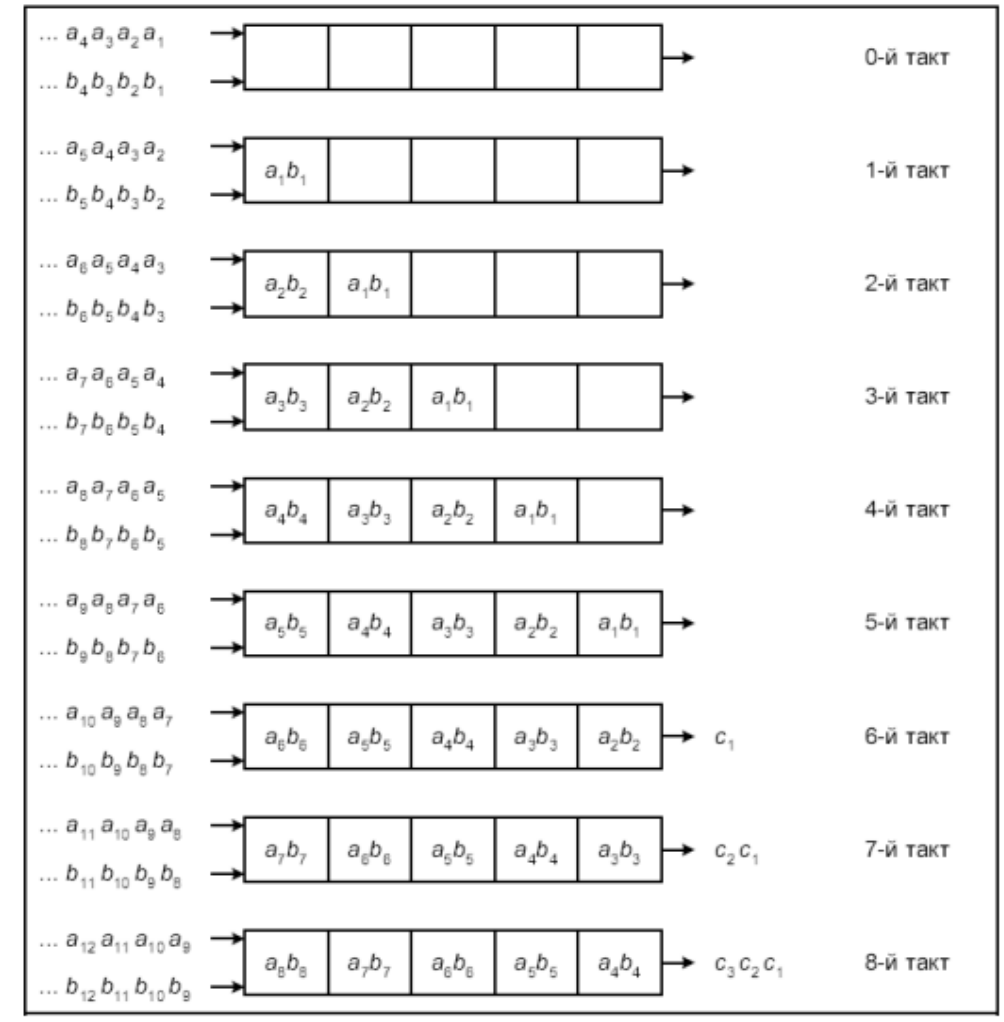


Рисунок 3.2 Знаходження суми **c = a** + **b** за допомогою 5-ти стадійного конвеєрного пристрою

Якщо конвеєрний пристрій є *l*-стадійним і обробка даних на кожній стадії триває один такт, то для виконання n послідовних операцій на цьому пристрої потрібно витратити $l+n-1$ тактів. Якщо ж цей пристрій використовувати у послідовному режимі, то кількість тактів буде рівна $l \cdot n$. При використанні векторних команд у формулі для тривалості обробки даних на конвеєрному пристрої додається ще один доданок σ — це час (у тактах), необхідний для ініціалізації векторної команди. Тому загальний час рівний $\sigma + l + n - 1$.

Оскільки ні σ, ні l не залежать від значення n, то із збільшенням довжини вхідних векторів *ефективність конвеєрної обробки даних зростає*.

Якщо під ефективністю обробки розуміти реальну продуктивність конвеєрного пристрою, рівну відношенню числа виконаних операцій n до часу їх виконання $t(n)$, то залежність продуктивності від довжини вхідних векторів визначається наступним співвідношенням:

$$\pi(n) = \frac{n}{t(n)} = \frac{n}{(l+n-1)\tau} = \frac{1}{(1+(l-1)/n)\tau},$$

де τ — це тривалість такту роботи комп'ютера.

На рис. 3.3 наведено приблизний вигляд графіка цієї залежності.

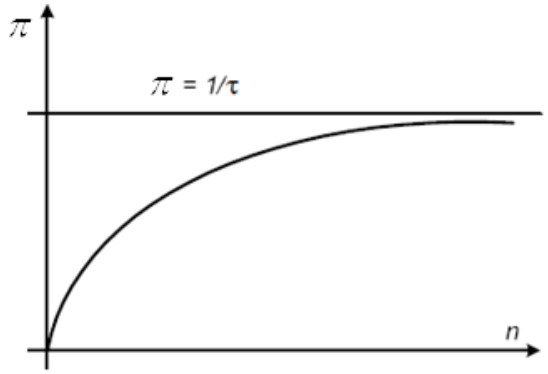


Рисунок 3.3. Залежність продуктивності конвеєрного пристрою від довжини вхідного набору

Із зростанням довжини вхідних даних реальна продуктивність конвеєрного пристрою все більше наближається до його пікової продуктивності. Однак *пікова продуктивність ніколи недосяжна на практиці*.

Прискорення $S(n)$ при виконанні n операцій, яке досягається за рахунок використання конвеєрного способу обробки даних, визначається величиною:

$$S(n) = \frac{n(t_1+\ldots+t_l)}{t_1+\ldots+t_l + (n-1)t_{max} + \sigma}$$

Граничне прискорення $S = S(n)$ рівне

$$S(n) = \frac{(t_1+\ldots+t_l)}{t_{max}}$$

Залежність продуктивності від довжини вхідних векторів визначається наступним співвідношенням:

$$\pi(n) = \frac{n}{(t_1+\ldots+t_l + (n-1)t_{max} + \sigma)\tau}$$

Приклад 2.1. Обробка даних на конвеєрному пристрої складається із 5 стадій, тривалості яких рівні 3, 5, 2, 6 та 4 такти відповідно. Виконати наступні завдання, вважаючи, що ініціалізація конвеєра потребує 2 тактів та тривалість одного такту складає 5 нс:

1) Обчислити кількість тактів, необхідну для виконання 1000 операцій обробки даних за умови, що пристрій працює:

а) у послідовному режимі;

б) у конвеєрному режимі.

2) Підрахувати пікову продуктивність системи.

3) Визначити найменшу кількість операцій, при виконанні яких у конвеєрному режимі досягається прискорення не менше за 90% від граничного прискорення.

Розв'язок:

Запишемо характеристики конвеєрного пристрою:

$$l = 5, \sigma = 2, \tau = 5 \cdot 10^{-9} c, t_1 = 3, t_2 = 5, t_3 = 2, t_4 = 6, t_5 = 4.$$

Тоді, $t_{max} = max\{t_1, \ldots, t_5\} = 6.$

Знайдемо шукану кількість тактів у випадку n =1000 у послідовному режимі:

$$t_s(n) = (t_1+\ldots+t_l) \cdot n$$
$$t_s(1000) = (3+5+2+6+4) \cdot 1000 = 2000$$

б) у конвеєрному режимі:

$t_c(n) = (t_1+\ldots+t_l) + (n-1)t_{max} + \sigma;$

$$t_c(1000) = 20 + 999 \cdot 6 + 2 = 6016.$$

Підрахуємо пікову продуктивність системи для обох режимів:

$\pi_S = \frac{1}{(t_1 + \ldots + t_l)\tau} = \frac{1}{20 \cdot 5 \cdot 10^{-9}} = 10^7.$

$\pi_c = \frac{1}{t_{max}\tau} = \frac{1}{6 \cdot 5 \cdot 10^{-9}} = 10^7.$

Визначимо шукану кількість операцій n, яка задовольняє умову $S(n) \geq 0{,}9S$:

$$\frac{20n}{20 + 6(n-1) + 2} \geq 0.9\frac{20}{6}$$

$$\frac{n}{6n + 16} \geq \frac{0.9}{6}$$

$$n \geq \frac{114}{6} = 249)$$

Відповідь: n = 24

Характеристики систем функціональних пристроїв

Будь-яка обчислювальна система являє собою сукупність деяких функціональних пристроїв (ФП). Для оцінки якості її роботи вводяться різні характеристики.

Нехай задана система відліку часу і задана деяка одиниця часу, наприклад, секунда. Будемо вважати, що всі спрацьовування одно і того самого ФП системи мають однакову тривалість.

Назвемо ФП *простим*, якщо ніяка наступна операція не може виконуватися на ньому до тих пір, поки не виконається попередня. Основна властивість простого ФП — він монопольне використовує своє обладнання для виконання кожної окремої операції.

На відміну від простого, *конвеєрний* ФП розподіляє своє обладнання для одночасного виконання кількох операцій. Дуже часто (але необов'язково) конвеєрні ФП конструюються як лінійні ланцюжки простих ФП (стадій). Простий ФП можна завжди вважати конвеєрним ФП з довжиною конвеєра, рівною 1.

Назвемо *вартістю операції* час її реалізації, а *вартістю роботи* — сумарну вартість усіх виконаних операцій. *Завантаженістю* пристрою *p* на даному проміжку часу будемо називати відношення вартості реально виконаної роботи до максимально можливої вартості. Наступні два

твердженню містять опис основних властивостей ФП та систем ФП.

Твердження 3.1. *Максимальна вартість, яку може виконати ФП за час T, рівна T для простого ФП та n T для конвеєрного ФП довжини n.*

Будемо називати *реальною продуктивністю* системи пристроїв кількість операцій, які реально виконуються у середньому за одиницю часу. *Піковою продуктивністю* називають максимальну кількість операцій, які могла би виконати система за одиницю часу у випадку відсутності зв'язків між її ФП. З визначення випливає, що *реальна (пікова) продуктивність системи рівна сумі реальних (пікових) продуктивностей ФП*, які входять до її складу.

Твердження 3.2. *Якщо система складається із l пристроїв, які мають пікові продуктивності $\pi_1, \dots \pi_l$ і працюють із завантаженістю $p_1, \dots p_l$, то реальна продуктивність системи r обчислюється за формулою*

$$r = \sum_{i=1}^{l} p_i \pi_i \tag{3.1}$$

Якщо пристрої мають пікові продуктивності $\pi_1, \dots \pi_l$ *і працюють із завантаженістю* $p_1, \dots p_l$, *то завантаженість системи є:*

$$p = \sum_{i=1}^{l} \alpha_i p_i \tag{3.2}$$

$$\text{Де } \alpha_i = \frac{\pi_i}{\sum_{i=1}^{l} \pi_i}$$

Справедлива наступна рівність:

$$r = p \cdot \pi \tag{3.3}$$

Велика кількість ФП, так само як і конвеєрні ФП, використовуються тоді, коли виникає потреба розв'язати задачу швидше. Для того, щоб зрозуміти, наскільки швидше це вдається зробити, потрібно увести у розгляд поняття «прискорення». Як і у випадку завантаженості це можна зробити по-різному. Розглянемо один із способів визначення прискорення. Будемо порівнювати швидкість роботи системи із швидкістю найпродуктивнішого пристрою системи. Відношення $S = \frac{r}{\pi_i}$ будемо називати *прискоренням реалізації алгоритму* на даній обчислювальній системі або просто *прискоренням*. Тобто,

$$S = \frac{\sum_{i=1}^{l} p_i \pi_i}{\pi_i} \quad (3.4)$$

Аналіз формули (3.5) показує, що прискорення обчислювальної системи, яка складається із l пристроїв, не може перевищувати l і може досягати l тоді і тільки тоді, коли усі пристрої системи мають однакові пікові продуктивності і є цілком завантаженими.

Твердження 3.3. *Якщо система складається із l пристроїв (простих чи конвеєрних), які мають однакові пікові продуктивності, то*

- *завантаженість системи рівна середньому арифметичному завантаженості усіх пристроїв;*
- *пікова продуктивність системи у l разів більша за продуктивність одного пристрою;*
- *прискорення системи рівне сумі завантаженості усіх пристроїв.*

Одним із основних питань теорії обчислювальних систем є питання досягнення високого рівня ефективності. Із (2.4) випливає, що для цього потрібно досягти високого рівня завантаженості системи. Цього у свою чергу можна досягти шляхом підвищення завантаженості окремих пристроїв. Проте залишається відкритим питання, як можна це зробити. Якщо пристрій не є завантаженим на 100%, то завантаженість можна завжди підвищити тільки у тому випадку, якщо він не пов'язаний із іншими пристроями. В іншому випадку ситуація не є очевидною.

Без обмеження загальності будемо вважати, що усі пристрої є простими, тому що довільний конвеєрний пристрій можна зобразити у вигляді ланцюжка простих пристроїв. Припустимо, що між пристроями встановлено направлені зв'язки, які не змінюються у процесі функціонування. Побудуємо орієнтований граф, вершини якого взаємно однозначно відповідають пристроям, а дуги — зв'язкам між ними. З вершини A проведемо дугу у вершину B тоді і тільки тоді, коли результат роботи пристрою, якому відповідає вершина A, передається у якості вхідного аргументу пристрою, якому ставиться у відповідність вершина B. Назвемо отриманий граф *графом системи*.

Твердження 3.4. Якщо система складається із l простих пристроїв, які мають пікові продуктивності $\pi_1, \dots \pi_l$, граф системи є слабко зв'язним (відповідний йому неорієнтований граф є зв'язним), то максимальна продуктивність системи виражається формулою:

π_i

В умовах твердження 3.4:

- *асимптотично усі пристрої виконують однакову кількість операцій;*
- *завантаженість кожного пристрою не перевищує завантаженість найменш продуктивного пристрою;*
- *якщо який-небудь пристрій завантажено повністю, то цей пристрій має найменшу продуктивність у системі;*

Закони Амдала

Перший закон Амдала. *Продуктивність обчислювальної системи, яка складається із пов'язаних між собою пристроїв, у загальному випадку визначається найменш продуктивним пристроєм.*

Кажучи, що система працює з максимальною можливою реальною продуктивністю, маємо на увазі те, що у системі забезпечується такий розклад команд, який мінімізує простій ФП системи.

Наслідок . *Нехай система утворена простими пристроями і має зв'язний граф. Тоді асимптотична продуктивність системи буде максимальною, якщо усі пристрої мають однакові пікові продуктивності.*

Максимальна продуктивність системи може досягатися при різних режимах роботи. Зокрема, вона досягається при синхронному режимі із тактом, обернено пропорційним продуктивності найповільнішого ФП системи. Із наслідку 2.3 можна зробити висновок, що продуктивність системи покращується, якщо усі пристрої системи мають однакову продуктивність.

Приклад 2.2. Граф системи ФП наведений на рис. 2.5. Пікові продуктивності пристроїв системи:

№	0	1	2	3	4	5	6	7	8	9	10	11	12
π	10	5	8	6	7	9	12	8	10	4	6	4	6

Знайти:

1. завантаженості усіх пристроїв системи;
2. реальну продуктивність системи;
3. завантаженість системи;
4. прискорення системи.

Розв'язок. Як видно з рис. 3.4, система складається з трьох незалежних

підсистем. Згідно до 1-го закону Амдала реальна продуктивність кожної із підсистем визначається продуктивністю найменш продуктивного пристрою цієї підсистеми.

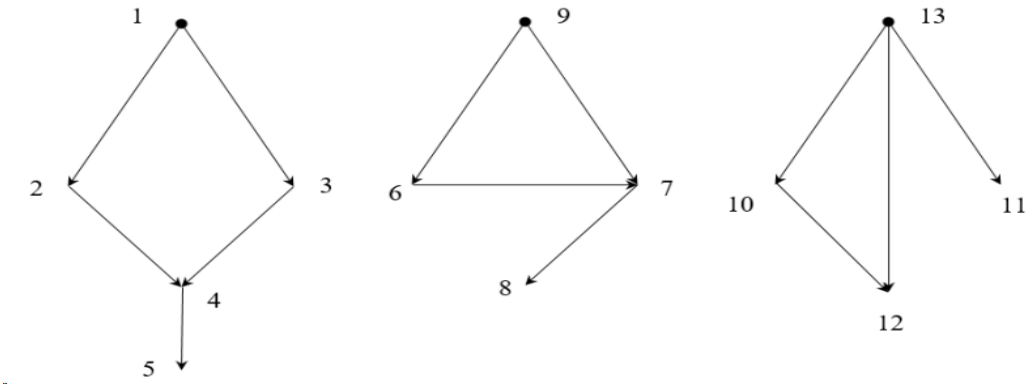


Рисунок 3.4 Граф системи ФП.

Нехай $\underline{r}^{(i)}$ - реальні продуктивності, з якими працюють усі пристрої *i*-системи, *i* =1,2,3. Тобто,

$r_1 = \ldots = r_5 = \underline{r}^{(1)}$,

$r_6 = \ldots = r_9 = \underline{r}^{(2)}$,

$r_{10} = \ldots = r_{13} = \underline{r}^{(3)}$.

Тоді,

$\underline{r}^{(1)} = 5, \underline{r}^{(2)} = 8, \underline{r}^{(3)} = 4$

Завантаженості p_i пристроїв першої підсистеми рівні $\underline{r}^{(1)}/\pi_i$, (i=1,…5):

$p_1 = \frac{5}{10},\ p_2 = 1; p_3 = \frac{5}{8}; p_4 = \frac{5}{6},\ p_5 = \frac{5}{7};$

Завантаженості p_i пристроїв другої підсистеми рівні $\underline{r}^{(2)}/\pi_i$, (i=6,…9):

$p_6 = \frac{8}{9},\ p_7 = \frac{2}{3}; p_8 = 1; p_9 = \frac{4}{5},$

Завантаженості p_i пристроїв третьої підсистеми рівні $\underline{r}^{(3)}/\pi_i$, (i=10,…13):

$p_{10} = 1, p_{11} = \frac{2}{3}; p_{12} = 1; p_{13} = \frac{2}{3},$

За першим законом Амдала (див. твердження 2.4) $r^{(i)} = l_i \cdot \underline{r}^{(1)}$, де l_i-кількість пристроїв і-ї підсистеми. Тому

$$\underline{r}^{(1)} = 5 \cdot 5 = 25, \underline{r}^{(2)} = 4 \cdot 8 = 32, \underline{r}^{(3)} = 4 \cdot 4 = 16$$

Отже, реальна продуктивність r = 25+32 + 16=73.

Для знаходження завантаженості системи використаємо формулу $r = p \cdot \pi$, де p — завантаженість, π — пікова продуктивність системи.

Тоді

$$\pi = \pi_1 + \ldots + \pi_{13} = 10 + 5 + 8 + 6 + 7 + 9 + 12 + 8 + 10 + 4 + 6 + 4 + 6 = 95$$

Тому $p = \frac{r}{\pi} = \frac{73}{95} = 0.77$

Для знаходження прискорення системи скористаємося формулою $S = r/\pi_i$,тоді S=73/12=6.08.

Припустимо, що усі системи є простими, універсальними (тобто на них можна виконувати різноманітні операції) та мають однакову продуктивність. Нехай у системі реалізується деякий алгоритм, а сама реалізація відповідає деякій його паралельній формі. Припустимо, що висота паралельної форми (кількість ярусів) рівна *m*, *ширина* (максимальна кількість вершин на одному ярусі) — q, а всього у алгоритмі виконується *N* операцій.

Твердження. *Для системи, яка задовольняє наведені вище умови, максимальне прискорення не більше за N / m.*

Наслідок. *Мінімальна кількість пристроїв системи, при якій може бути досягнуто максимально можливе прискорення, рівна ширині алгоритму.*

Припустимо, що у алгоритмі *n* операцій із *N* виконуються послідовно. Причини цього можуть бути різними. Наприклад, операції можуть бути пов’язані послідовними інформаційними зв’язками. Також цілком можливим є те, що при реалізації алгоритму просто не розпізнали паралелізм, наявний у відповідній його частині. Відношення $\beta = n/N$- назвемо *часткою послідовних обчислень*.

Другий закон Амдала. *Нехай система складається із l однакових простих універсальних пристроїв. Тоді максимальне можливе прискорення системи рівне*

$$S_l = \frac{l}{\beta l + (1 - \beta)}$$

Відповідно другому закону Амдала: "На процесорну потужність системи накладаються обмеження, зумовлені часткою програми, що не може бути паралельно виконаною". Доведення цього закону базується на

принципі скінченної паралелізації, що означає, що не всі частини програми можуть виконуватися паралельно. Отже, коли ми додаємо більше процесорів до системи, вони не зможуть бути використані ефективно, якщо є частина програми, яка не може бути виконана в паралелі. На прикладі: Якщо у нас є програма, яка на 75% може бути виконана паралельно, тоді навіть якщо ми підключимо нескінченну кількість процесорів, максимальне прискорення, яке можна отримати, становить 4 (тобто програма виконається у 4 рази швидше). Це тому, що наявний непаралельний розділ програми (25%) встановлює мінімум часу, необхідного для виконання програми, незалежно від кількості використовуваних процесорів.

Третій закон Амдала. *Нехай система складається із l однакових простих універсальних пристроїв. При будь-якому режимі роботи її прискорення менше за обернену величину до частки послідовних обчислень* $S_l < 1/\beta$.

Третій закон Амдала можна виразити наступним чином: якщо *S* – частка програми, що може бути зроблена паралельно (тобто цілком може бути розподілена між кількома процесорами), а *(1 - S)* – частка програми, що не може бути розподілена (тобто її потрібно виконати послідовно), тоді максимальне можливе покращення швидкості виконання дорівнює *1 / (1 - S)*. Таким чином, третій закон Амдала дає можливість дізнатися максимальну ефективність, яку можна отримати від додавання додаткових ядер або процесорів. Наприклад, програма виконується 12 годин, з яких 9 годин може бути розподілено (їх можна паралелізувати). Це означає, що *S = 9/12 = 0,75*; *(1 - S) = 1 - 0,75 = 0,25*. Тобто, навіть якщо у вас необмежене число процесорів для паралелізації 9 годин обчислень, максимальне покращення швидкості виконання буде *1 / (1 - S) = 1 / (1 - 0,75) = 4* рази.

Другий та третій закони Амдала проілюстровані на рис. 3.5. Величина Smax =1/ β називається граничним прискоренням системи для заданого алгоритму.

Для системи, яка складається із l однакових простих пристроїв величина $E_l = S_l/l$ називається ефективністю системи (при реалізації заданого алгоритму). Легко переконатися, що для розглядуваних систем ефективність співпадає із завантаженістю р. Максимальні значення прискорення та ефективності рівні l та 1, відповідно.

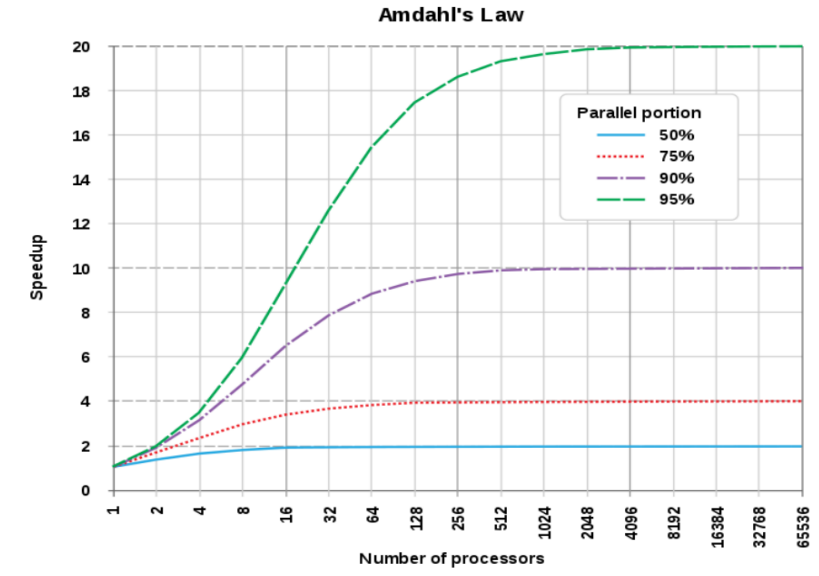


Рисунок. 3.5. Графік залежності прискорення від кількості пристроїв системи

Запитання для самоперевірки

1. Назвіть основні стадії процесора Pentium 4.
2. Скільки тактів знадобиться для обчислення суми 2-х чисел за допомогою 5-ти стадійного конвеєрного пристрою? Поясніть відповідь.
3. Яким відношенням визначається продуктивність конвеєрного пристрою?
4. Дайте визначення пікової продуктивності.
5. Як визначити завантаженість, реальну та пікову продуктивність системи, яка складається з l пристроїв, які мають однакові пікові продуктивності?
6. Як визначити завантаженість, реальну та пікову продуктивність системи, яка складається з l пристроїв, які мають різні пікові продуктивності?
7. Як визначити прискорення реалізації алгоритму?
8. Назвіть перший закон Амдала та його наслідок.
9. Поясніть другий закон Амдала.
10. Як визначити покращення швидкості при додаванні додаткових ядер для паралельного обчислення за третім законом Амдала?

Лекція 3.
Паралельна форма алгоритму. Концепція необмеженого паралелізму теорії паралельних обчислень. Процеси і потоки

Граф алгоритму

Будь-яка програма для «звичайного комп'ютера» описує деяку родину алгоритмів. Вибір конкретного алгоритму при її реалізації визначається тим, як «спрацьовують» умовні оператори, які залежать від вхідних даних. Тому «звичайний» комп'ютер завжди виконує деяку послідовність дій, яка *однозначно* визначається програмою та вхідними даними, причому в кожний момент часу виконується рівно одна дія.

Інакша ситуація в системах з паралельною архітектурою. Для них в кожний момент часу може виконуватися цілий набір операцій, які не залежать одна від іншої. На довільній конкретній паралельній системі ці набори та послідовність їх виконання однозначно визначаються програмою та вхідними даними. На різних системах ці набори та послідовності можуть бути *різними*. Тим не менш, для гарантування отримання однакового результату порядок виконання послідовності операцій має задовольняти певні умови.

Деякі операції алгоритму використовують результати виконання інших операцій і тому *обов'язково мають виконуватися* після цих операцій. Таким чином на множині операцій розробник програми явно чи неявно задає деякий «частковий порядок», який для довільних двох операцій вказує, яка з них має виконуватися раніше, або констатує, що операції можуть виконуватися незалежно.

Кожній операції алгоритму ставиться у відповідність *вершина графа*. Вершина графа *з'єднується дугою* з іншою вершиною, якщо перша вершина безпосередньо передує іншій (тобто результат першої операції є аргументом другої операції). Отриманий граф називається *графом алгоритму*.

У множині вершин графа також виділяють дві групи вершин: *вхідні вершини*, у які не входить жодна дуга, та *вихідні вершини*, з яких не виходять дуги.

Граф алгоритму майже завжди залежить від вхідних даних. Якщо у програмі відсутні умовні оператори, то він залежить від розмірів вхідних (вихідних) масивів, бо вони визначають загальну кількість операцій, а отже, і вершин графа. Таким чином на практиці майже завжди мають справу з алгоритмами, граф яких є пара метризованим. Від значення параметрів

залежить не тільки кількість вершин, а й уся сукупність дуг.

Якщо програма не містить умовних операторів, то її саму і також алгоритм, який вона реалізовує, називається *детермінованим.* Існує принципова відмінність між детермінованими та недетермінованими алгоритмами. Для детермінованого алгоритму завжди існує взаємно однозначна відповідність між всіма операціями та усіма вершинами графу алгоритму незалежно від значення вхідних параметрів. Для недетермінованого алгоритму такої відповідності нема. Надалі, будуть розглядатися лише детерміновані алгоритми. Їх аналіз простіший, ніж у загальному випадку. Крім того, багато недетермінованих алгоритмів є «майже» детермінованими, тобто зводяться до детермінованих.

Уведений у розгляд граф алгоритму є орієнтованим ациклічним мультиграфом. Його ациклічність випливає із того, що у довільних програмах реалізовують лише явні обчислення і ніяка величина не визначається сама через себе.

Твердження. Нехай $G=(V , E)$ — зв’язний ациклічний граф, який має n вершин. Тоді існує таке число $H < n$, що усі вершини графа можна так помітити одним із індексів що усі значення індексу задіяні і якщо дуга виходить із вершини з індексом i та входить у вершини з індексом j, то $i < j$. **Граф, отриманий у відповідності із твердженням 3.1,** називається *строгою паралельною формою* графа алгоритму. Додаткове визначення - граф називають строгою паралельною формою графа алгоритму, якщо він містить лише дуги з паралельним зв'язком та дуги з послідовним зв'язком і не містить дуг із залежним зв'язком. Отже, у такому графі всі вершини не мають прямих входів або прямих виходів, що дозволяє виконувати операції паралельно.

Група вершин, які мають однакові індекси називається *ярусом* паралельної форми, а кількість вершин у групі — *шириною* ярусу. Кількість ярусів у паралельній формі (без врахування 0-го ярусу) називається *висотою* паралельної форми (вона рівна числу H), максимальна ширина ярусу — її *шириною*. Відповідні «ботанічні» терміни застосовуються також і безпосередньо до алгоритмів.

Якщо для простоти вважати, що усі операції алгоритму виконуються за одиницю часу, то висота паралельної форми алгоритму рівна часу реалізації алгоритму. Якщо алгоритм реалізовується на «звичайному» комп’ютері, то усі яруси паралельної форми (крім, можливо, 0-го) містять одну вершину. Така паралельна форма називається лінійною.

Концепція необмеженого паралелізму

Поява перших паралельних обчислювальних систем в 60-х роках XX століття зумовила необхідність математичної концепції побудови *паралельних алгоритмів*, тобто алгоритмів, пристосованих до реалізації на таких системах. Швидкий розвиток елементної бази підказував дослідникам, що кількість обчислювальних пристроїв у системі невдовзі може стати дуже великим. Відповідна концепція отримала назву *концепції необмеженого паралелізму*. В основі концепції лежить припущення, що алгоритм реалізується на паралельній обчислювальній системі, яка не накладає на нього практично ніяких обмежень. Згідно до концепції вважається, що

- процесорів може бути як завгодно багато;
- усі процесори системи є універсальними;
- процесори працюють у синхронному режимі;
- усі запам'ятовуючі пристрої системи спільні;
- передавання інформації у системі виконується миттєво і без конфліктів.

Концепція необмеженого паралелізму є ідеалізованою математичною моделлю паралельної обчислювальної системи. Вона має як свої переваги, так і недоліки.

Для знаходження розв'язку однієї і тої самої задачі можуть використовуватися алгоритми різної паралельної складності. Серед них можуть бути і алгоритми найменшої висоти. Розглянемо класичний приклад, який демонструє принципи побудови алгоритмів «малої висоти».

Висота паралельної форми рівна 3, ширина рівна 4. Суттєве зменшення висоти зумовлене більшим завантаженням процесорів виконанням корисної ро- боти. Відповідні графи описаних алгоритмів наведені на рис. 4.1 (початкові вершини символізують ввід даних).

Остання схема очевидним чином розповсюджується на випадок довільного *n*. Для її реалізації потрібно на кожному ярусі виконувати максимально можливу кількість множень пар чисел, які отримані на попередньому ярусі (і не мають спільних множників). В загальному випадку висота паралельної форми рівна $\log_2 n$, така паралельна форма може бути реалізована на n/2 процесорах, при ступінь завантаженості процесорів зменшується від ярусу до ярусу.

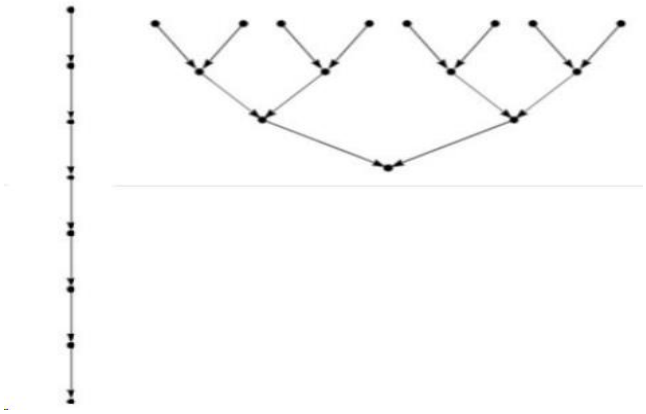


Рисунок 4.1. Граф послідовного алгоритму та алгоритму «здвоєння»

Якщо система містить *l* однакових простих універсальних процесорів, то прискорення реалізації алгоритму на цій системі $S_l(n)$ можна обчислити за формулою

$$S_l(n) = \frac{T_1(n)}{T_l(n)}$$

Де $T_1(n)$ - час, за який можна реалізувати алгоритм на одному процесорі, $T_l(n)$ - час реалізації алгоритму в системі (висота алгоритму), n — розмірність задачі.

Ефективністю $E_l(n)$ реалізації алгоритму у паралельній системі називається відношення прискорення до кількості процесорів системи, тобто

$$E_l(n) = \frac{S_l(n)}{l},$$

Внутрішній паралелізм

У процесі тривалого використання послідовних комп'ютерів був накопичений значний багаж обчислювальних алгоритмів та програм. Поява паралельних комп'ютерів повинна була зумовити розробку нових ефективних паралельних методів. Але на практиці цього не відбулося. Тому постає питання, як тоді розв'язувати задачі на паралельних комп'ютерах?

Візьмемо довільний придатний алгоритм, записаний у вигляді математичних співвідношень, послідовних програм чи якимось іншим способом. Припустимо, що для цієї форми запису побудовано граф алгоритму. Припустимо також, що для цього графа знайдено паралельну форму із достатньою шириною ярусів. Тоді розглянутий алгоритм, принаймні принципово, можна реалізувати на паралельній обчислювальній системі. Важливо зауважити, що паралельна реалізація буде мати такі самі обчислювальні властивості, що й звичайна. Подібний паралелізм у

алгоритмах отримав назву *внутрішнього паралелізму.* Виявилося, що багато існуючих ефективних послідовних алгоритмів мають значний запас «внутрішнього паралелізму». Складність полягає лише у тому, як виявити цей паралелізм.

Процеси і потоки

Основою для побудови моделей функціонування програм, що реалізовують паралельні методи вирішення завдань, являється поняття процесу як конструктивної одиниці побудови паралельної програми. Опис програми у вигляді набору процесів, що виконуються паралельно на різних процесорах або на одному процесорі в режимі розподілу часу, дозволяє сконцентруватися на розгляді проблем організації взаємодії процесів, визначити моменти й способи забезпечення синхронізації та взаємовиключення процесів, вивчити умови виникнення або довести відсутність безвихідної ситуації в ході виконання програм (ситуацій, в яких усі або частина процесів не можуть бути продовжені при будь-яких варіантах продовження обчислень).

Для мультипроцесорних систем із загальною пам'яттю взаємодію процесів підрозділяють на дві категорії:

– взаємне виключенні процесів - коли вони «змагаються» за загальний ресурс та можуть володіти їм або мати доступ до нього тільки окремо;

– синхронізація процесів, коли вони «співпрацюють» при рішенні однієї задачі й повинні погоджувати, синхронізувати свої дії при операціях над загальними об'єктами.

Концепція процесу

Поняття процесу є одним із базових в теорії й практиці паралельного програмування. Трактування цього поняття є досить широким, але в цілому більшість визначень зводяться до розуміння процесу як «деякої послідовності команд програми, що претендує, нарівні з іншими процесами, на використання процесора для свого виконання». Конкретизація поняття процесу залежить від цілей дослідження паралельних програм. Для аналізу проблем організації взаємодії процесів процес можна розглядати як послідовність команд.

В ході свого виконання стан процесу може багаторазово змінюватися; можливі варіанти зміни станів показані на діаграмі переходів (рис. 5.1.).

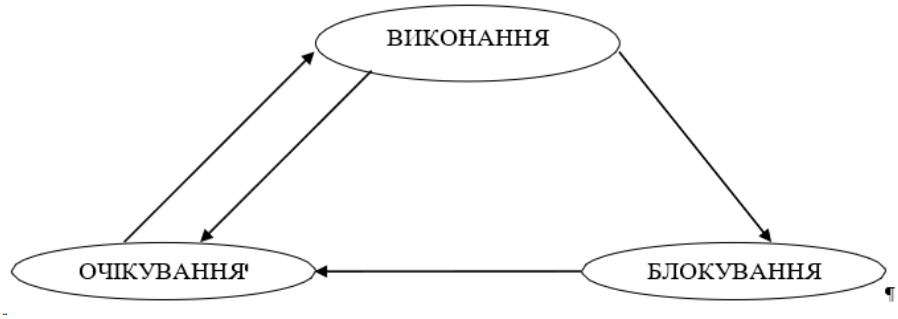


Рисунок 5.1. Діаграма переходів процесу із стану в стан

Поняття ресурсу

Поняття ресурсу зазвичай використовується для позначення будь-яких об'єктів обчислювальної системи, які можуть бути використані процесом для свого виконання. Як ресурс може розглядатися процесор, пам'ять, програми, дані тощо. За характером використання можуть розрізнятися наступні категорії ресурсів:

- ресурси, що використовуються монопольно (неперерозподільні), характеризуються тим, що виділяються процесам у момент їх виникнення та вивільняються тільки у момент завершення процесів;
- повторно розподільчі ресурси відрізняються можливістю динамічного запиту, виділення й вивільнення в ході виконання процесів (таким ресурсом є, наприклад, оперативна пам'ять);
- ресурси, що розділяються, особливість яких полягає в тому, що вони постійно залишаються в загальному використанні та виділяються процесам для використання в режимі розподілу часу (як, наприклад, процесор, файли, що розділяються тощо);
- багаторазово використовувані (реентерабельні) ресурси характеризуються можливістю одночасного використання декількома процесами (що може бути забезпечене, наприклад, при незмінності ресурсу під час його використанні; як приклади таких ресурсів можуть розглядатися реентерабельні програми, файли, що використовуються тільки для читання і так далі).

Слід зазначити, що тип ресурсу визначається не лише його конкретними характеристиками, але й залежить від способу використання. Так, наприклад, оперативна пам'ять може розглядатися як повторно розподільчій, так й ресурс, що розділяється; використання програм може

бути організоване у вигляді ресурсу будь-якого розглянутого типу.

Організація програм як системи процесів

Поняття процесу може бути використане як основний конструктивний елемент для побудови паралельних програм у вигляді сукупності взаємодіючих процесів. Така агрегація програми дозволяє отримати компактніші (що піддаються аналізу) обчислювальні схеми методів, що реалізовуються, приховати при виборі способів розпаралелювання несуттєві деталі програмної реалізації, забезпечує концентрацію зусиль на рішення основних проблем паралельного функціонування програм.

Існування декількох одночасно виконуваних процесів призводить до появи додаткових співвідношень, які повинні виконуватися для величин тимчасових траєкторій процесів.

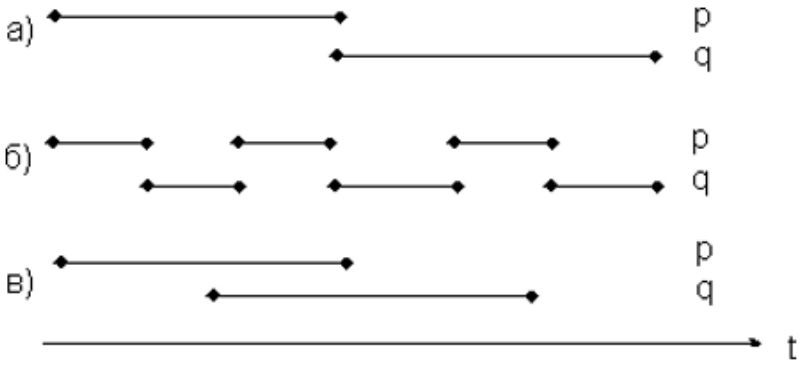


Рисунок 5.2. Варіанти взаєморозташування траєкторій одночасно виконуваних процесів (відрізки ліній - фрагменти командних послідовностей процесів)

Можливі типові варіанти таких співвідношень на прикладі двох процесів p та q полягають в наступному (рис. 5.2.);

Виконання процесів здійснюється строго послідовно, тобто процес q починає своє виконання тільки після повного завершення процесу p (однопрограмний режим роботи ЕОМ - рис. 5.2., а);

Виконання процесів може здійснюватися одночасно, але в кожен момент часу можуть виконуватися команди тільки одного або іншого процесу (режим розподілу часу або багатопрограмний режим роботи ЕОМ- рис. 5.2., б);

Паралельне виконання процесів, коли одночасно можуть виконуватися команди декількох процесів (цей режим виконання процесів можливий тільки за наявності в обчислювальній системі декількох процесорів рис. 5.2., в).

Приведені варіанти взаєморозташування траєкторій процесів визначаються не вимогами необхідних функціональних взаємодій процесів, а є лише наслідком технічної реалізації одночасної роботи декількох процесів. З іншого боку, можливість чергування за часом командних послідовностей різних процесів слід враховувати при реалізації процесів.

Виділені особливості одночасного виконання декількох процесів можуть бути сформульовані у вигляді ряду принципових положень, які повинні враховуватися при розробці паралельних програм:

- моменти виконання командних послідовностей різних процесів можуть чергуватися за часом;
- між моментами виконання команд різних процесів можуть виконуватися різні тимчасові співвідношення; характер цих співвідношень залежить від кількості й швидкодії процесорів та завантаження обчислювальної системи і тому не може бути визначений заздалегідь;
- тимчасові співвідношення між моментами виконання команд можуть розрізнятися при різних запусках програм на виконання, тобто одній і тій же програмі при одних і тих же початкових даних можуть відповідати різні командні послідовності внаслідок різних варіантів чергування моментів роботи різних процесів;
- доказ достовірності отримуваних результатів повинен проводитися для будь-яких можливих тимчасових співвідношень для елементів тимчасових траєкторій процесів;
- для виключення залежності результатів виконання програми від порядку чергування команд різних процесів потрібний аналіз ситуацій взаємовпливу процесів та розробка методів для їх виключення.

Перераховані моменти свідчать про істотне підвищення складності паралельного програмування в порівнянні з розробкою «традиційних» послідовних програм.

Взаємодія та взаємовиключення процесів

Однією з причин залежності результатів виконання програм від порядку чергування команд може бути розподіл одних й тих же даних між одночасно виконуваними процесами (наприклад, як це здійснюється у вище розглянутому прикладі). Ця ситуація може розглядатися як прояв загальної проблеми використання ресурсів (загальних даних, файлів, пристроїв тощо), що розділяються. Для організації розподілу ресурсів між декількома процесами необхідно мати можливість:

- визначення доступності затребуваних ресурсів (ресурс вільний

і може бути виділений для використання, ресурс вже зайнятий одним з процесів програми та не може використовуватися додатково яким-небудь іншим процесом);

- виділення вільного ресурсу одному з процесів, що запитали ресурс для використання;
- призупинення (блокування) процесів, що видали запити на ресурси, зайняті іншими процесами.

Головною вимогою до механізмів розподілу ресурсів є гарантоване забезпечення використання кожного ресурсу, що розділяється, тільки одним процесом від моменту виділення ресурсу цьому процесу до моменту звільнення ресурсу. Ця вимога називається взаємовиключенням процесів; командні послідовності процесів, в яких процес використовує ресурс, називається *критичною секцією процесу*. З використанням останнього поняття, умова взаємовиключення процесів може бути сформульована як вимога знаходження в критичних секціях процесу ресурсу, що розділяється.

Розглянемо декілька варіантів програмного рішення проблеми взаємовиключення (для запису програм використовується мова програмування C++). У кожному з варіантів пропонуватиметься деякий частковий спосіб взаємовиключення процесів з метою демонстрації усіх можливих ситуацій при використанні загальних ресурсів, що розділяються. Послідовне удосконалення механізму взаємовиключення при розгляді варіантів приведе до викладу алгоритму Деккера, що забезпечує взаємовиключення для двох паралельних процесів. Обговорення способів взаємовиключення завершується розглядом концепції семафорів Дейкстри, які можуть бути використані для загального вирішення проблеми взаємовиключення будь-якої кількості взаємодіючих процесів.

Варіант 1

```
intProcessNum = 1; // номер процесу для доступу до ресурсу Process_1()
{
while (1){
// повторювати, поки право доступу до ресурсу у процесу 2 while (
ProcessNum == 2 );
< Використання загального ресурсу >
// передача права доступу до ресурсу процесу 2 ProcessNum = 2;
}}
Process_2()
{
```

while (1){

// // повторювати, поки право доступу до ресурсу у процесу 1 while (ProcessNum == 1);

< Використання загального ресурсу >

// передача права доступу до ресурсу процесу 1 ProcessNum = 1;

}

}

Реалізований в програмі спосіб гарантує взаємовиключення, проте такому рішенню властиві два істотні недоліки:

- ресурс використовується процесами строго послідовно (по черзі) і, як результат, при різному темпі розвитку процесів загальна швидкість виконання програми визначатиметься найбільш повільним процесом;
- при завершенні роботи якого-небудь процесу інший процес не зможе скористатися ресурсом та може виявитися в постійно заблокованому стані.

Вирішення проблеми взаємовиключення відоме в літературі як спосіб жорсткої синхронізації.

Варіант 2

У цьому варіанті для відходу від жорсткої синхронізації використовуються дві керуючі змінні, які фіксують використання процесами ресурсу, що розділяється.

int ResourceProc1 = 0; // = 1 - ресурс зайнятий процесом 1 int ResourceProc2 = 0; // = 1 - ресурс зайнятий процесом 2 Process_1() {

while (1){

// повторювати, поки ресурс використовується процесом 2

while (ResourceProc2 == 1); ResourceProc1 = 1;

< Використання загального ресурсу > ResourceProc1 = 0;

}

}

Process_2()

{ while (1){

// повторювати, поки ресурс використовується процесом 1 while (ResourceProc1 == 1);

ResourceProc2 = 1;

< Використання загального ресурсу > ResourceProc2 = 0; }

}

Запропонований варіант розподілу ресурсів усуває недоліки жорсткої синхронізації, проте при цьому втрачається гарантія того, що при взаємовиключенні обидва процеси можуть виявитися одночасно у своїх критичних секціях (це може статися, наприклад, при перемиканні між процесами у момент завершення перевірки зайнятості ресурсу). Ця проблема виникає внаслідок відмінності моментів перевірки та фіксації зайнятості ресурсу.

Звідси слідує два важливі висновки:

- успішність одноразового виконання не може бути доказом правильності функціонування паралельної програми навіть при незмінних параметрах вирішуваної задачі;
- для виявлення помилкових ситуацій потрібна перевірка різних тимчасових траєкторій виконання паралельних процесів.

Варіант 3

Можливий варіант відновлення взаємовиключення може полягати в установці значень змінних управління перед циклом перевірки зайнятості ресурсу.

int ResourceProc1 = 0; // = 1 - ресурс зайнятий процесом 1 int ResourceProc2 = 0; // = 1 - ресурс зайнятий процесом 2 Process_1() {

while (1){

// встановити, що процес 1 намагається зайняти ресурс ResourceProc1 = 1;

// повторювати, поки ресурс зайнятий процесом 2 while (ResourceProc2 == 1);

< Використання загального ресурсу > ResourceProc1 = 0;

}

}

Process_2()

{

while (1){

// встановити, що процес 2 намагається зайняти ресурс ResourceProc2 = 1;

// повторювати, поки ресурс використовується процесом 1 while (ResourceProc1 == 1);

< Використання загального ресурсу > ResourceProc2 = 0;

}

}

Представлений варіант відновлює взаємовиключення проте при цьому виникає нова проблема - обидва процеси можуть виявитися заблокованими внаслідок нескінченного повторення циклів очікування звільнення ресурсів (відбувається при одночасній установці змінних управління в стан «зайнято»). Ця проблема відома під назвою ситуації безвиході («дедлок»). Виключення безвиході є одним з найбільш важливих завдань в теорії і практиці паралельних обчислень.

Варіант 4

Пропонований підхід для усунення безвиході полягає в організації тимчасового зняття значення зайнятості змінних процесів в циклі очікування ресурсу.

int ResourceProc1 = 0; // =1 - ресурс зайнятий процесом 1 int ResourceProc2 = 0; // =1 - ресурс зайнятий процесом 2 Process_1() {

while (1){

ResourceProc1 = 1; // процес 1 намагається зайняти ресурс

// повторювати, поки ресурс зайнятий процесом 2

while (ResourceProc2 == 1){

ResourceProc1 = 0; // зняття зайнятості ресурсу

< тимчасова затримка > ResourceProc1 = 1;

}

< Використання загального ресурсу > ResourceProc1 = 0; } }

Process_2()

{

while (1){

ResourceProc2 = 1; // процес 2 намагається зайняти ресурс

// повторювати, поки ресурс використовується процесом 1 while (ResourceProc1 == 1){

ResourceProc2 = 0; // зняття зайнятості ресурсу

< тимчасова затримка >ResourceProc2 = 1;

}

< Використання загального ресурсу > ResourceProc2 = 0;

}

}

Тривалість тимчасової затримки в циклах очікування повинна визначатися за допомогою деякого випадкового датчика. За таких умов реалізований алгоритм забезпечує взаємовиключення та виключає виникнення безвиході, але знову таки не позбавлений вагомого недоліку.

Проблема полягає в тому, що потенціальне вирішення питання про виділення може відкладатися до нескінченності (при синхронному виконанні процесів). Ця ситуація відома під найменуванням нескінченне відкладення (starvation).

Запитання для самоперевірки

1. Який граф називають строгою паралельною формою графа алгоритму?
2. Наведіть приклад графа, який відображає простий паралельний алгоритм, в якому задачі розподілені на дві гілки для паралельного виконання, і потім їх результати об'єднуються.
3. Назвіть параметри паралельної форми алгоритму.
4. Як визначити ширину ярусу паралельної форми графа алгоритму?
5. Назвіть характеристики паралельних алгоритмів. Як їх обчислити?
6. Поясніть ідею, яка лежить в основі концепції необмеженого паралелізму?
7. Дай визначення поняттю "внутрішній паралелізм". Що значить, коли програма або процес має запас "внутрішнього паралелізму"?
8. Поясніть паралелізм в алгоритмі множення матриць.
9. Дайте визначення поняття процесу.
10. Дайте визначення поняттю взаємовиключення процесів. Які способи синхронізації процесів вам відомі?
11. При якій ситуації можливе взаємнеблокування потоків?
12. Поясніть різницю між потоками на рівні користувача (UserLevel Threads) і потоками на рівні ядра (Kernel-Level Threads).

Лекція 4.
Процеси і потоки в ОС Windows

Процесом (process) називається екземпляр програми, завантаженої в пам'ять. Цей екземпляр може створювати потоки (thread), які є послідовністю інструкцій на виконання. Важливо розуміти, що виконуються не процеси, а саме потоки. Причому будь-який процес має хоча б один потік. Цей потік називається головним (основним) потоком додатку.

Оскільки практично завжди потоків значно більше, чим фізичних процесорів для їх виконання, то потоки насправді виконуються не

одночасно, а по черзі. (розподіл процесорного часу відбувається саме між потоками.) Але перемикання між ними відбувається так часто, що здається ніби вони виконуються паралельно. Залежно від ситуації потоки можуть знаходитися в трьох станах. По- перше, потік може виконуватися, коли йому виділений процесорний час, тобто він може знаходитися в стані активності. По-друге, він може бути неактивним та чекати виділення процесора, тобто бути в стані готовності. І є ще третій, теж дуже важливий стан - стан блокування. Коли потік заблокований, йому взагалі не виділяється час. Зазвичай блокування ставиться на час очікування якої-небудь події. При виникненні цієї події потік автоматично переводиться із стану блокування в стан готовності. Наприклад, якщо один потік виконує обчислення, а інший повинен чекати результатів, щоб зберегти їх на диск. Другий потік може заблокувати себе до тих пір, поки перший не встановить сигнальну подію, що читання закінчене. У системі виділяються два види потоків - інтерактивні, такі, що реалізують свій цикл обробки повідомлень (наприклад, головний потік програми), та робочі, що є простою функцією. У другому випадку потік завершується під час завершення виконання цієї функції. Заслуговує уваги також спосіб організації черговості потоків. Можна було б, звичайно, обробляти усі потоки за чергою але такий спосіб не завжди найефективніший. Набагато ефективнішим виявилося ранжувати усі потоки за пріоритетами. Пріоритет потоку позначається числом від 0 до 31 та визначається виходячи із пріоритету процесу, що «породив» потік й відносного пріоритету самого потоку. Таким чином, досягається найбільша гнучкість, кожен потік в ідеалі отримує стільки часу, скільки йому необхідно.

Іноді пріоритет потоку може змінюватися динамічно. Так інтерактивні потоки, що мають зазвичай клас пріоритету normal, система обробляє інакше та дещо підвищує фактичний пріоритет таких потоків, коли процес, який їх породив, знаходиться на передньому плані (foreground). Це зроблено для того, щоб застосування, з якому в даний момент працює користувач, швидше реагувало на його дії.

Створення потоків в Java

Платформа Java з самого початку підтримує паралельне програмування, з базовою підтримкою паралелізму в мові програмування Java і бібліотеках кла- сів Java. Java підтримує виконання програм, розроблених для паралельного виконання, як на кількох ядрах (декількох ядрах декількох процесорів), так і на одному ядрі (процесорі). У будь-якому

випадку набори операторів, що представляють логічні частини програми, які повинні працювати паралельно один з одним інкапсулюються в об'єкти, звані потоками виконання (threads) (далі для стислості будемо називати їх потоками), а така реалізація паралелізму називається багатопотоковою (multithreading). Клас Thread та інтерфейс Runnable розміщенні в основному пакеті java.lang, а у базовому класі java.lang.Object, від яко- го успадковуються всі інші класи, розміщені методи роботи з потоками: notify(), notifyAll(), wait(), wait(long timeout) та wait(long timeout, int nanos) (будуть розглянуті далі). В Java 5 був доданий пакет java.util.concurrent з високорівневими засобами Concurrency API багатопотокового програмування такими, як колекції, що ефективно працюють в багатопотоковому середовищі, виконувачі (Executors), використовувані для створення пулів потоків та планування роботи асинхронних завдань з отриманням результатів, синхронізатори (Synchronizers), які забезпечують зручну та ефективну синхронізацію потоків, та класи з підтримкою атомарних операцій (Atomics) над примітивними типами та посиланнями. Однопотокова програма (single-thread) має одну точку входу (метод main()) і одну точку виходу.

Багатопотокова програма (multithread) має початкову точку входу (метод main()) і кілька наступних точок входу і виходу (по кількості потоків), при цьому виконання одного потоку може бути перерва- но для виконання іншого (Рис. 6.1).

Потік може бути розподілений для виконання одному виконуючому блоку (процесору), але також може реалізовуватися в багатопроцесорному (багатоя- дерному) середовищі. Багатозадачні операційні системи зазвичай підтримують багатопоточність. Відмінною особливістю потоків є єдиний адресний простір, який поділяють потоки однієї програми (це обумовлює менші витрати продуктивності при перемиканні потоків і більш простий обмін даними між потоками). У той же час потоки не є самодостатньою програмою, а зазвичай виконуються в рамках в процесу (виконуваної програми).

У разі, коли завдання розділяється на кілька частин, сумарний час її виконання на паралельній системі не може бути менше часу виконання найдовшого фрагмента. Згідно з цим законом, прискорення виконання програми за рахунок розпаралелювання її інструкцій на безлічі обчислювачів обмежена часом, необхідним для виконання її послідовних інструкцій.

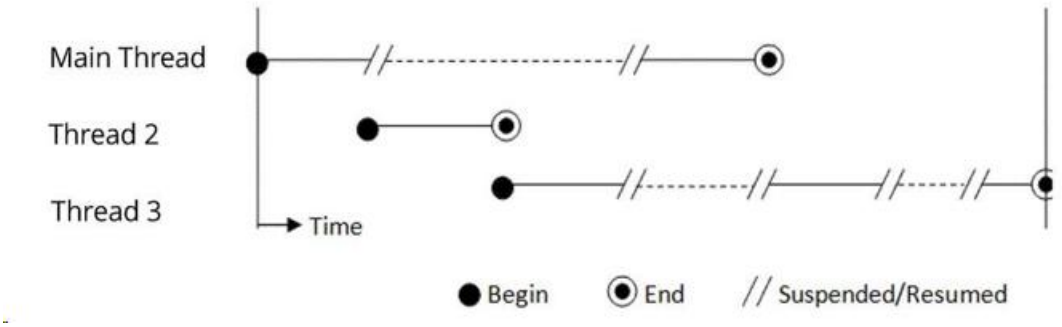


Рисунок 6.1. Багатопотокова програма

Існує два способи визначення послідовності операторів, які повинні виконуватись у потоці:

1) написання класу, що реалізовує інтерфейс java.lang.Runnable

2) написання класу користувача, що успадковує клас java.lang.Thread (останній огортає private Runnable target та реалізує метод run() інтерфейсу Runnable), і перевизначення методу run(), вказавши в ньому логічну послідовність операторів, які повинні виконуватись у потоці (в даному випадку клас користувача можна вважати потоком). Зазначимо, що запуск потоку виконується викликом метода start() класу Thread, а не безпосередньо метода run(), метод виконує додаткові функції і запускає метод run() потоку.

Таким чином, інтерфейс Runnable - є абстракцією над завданням, що виконується у потоці, і дозволяє розмежити виконання завдання від логіки управління потоками. Окрім того, використання інтерфейсу Runnable дозволяє класу користувача бути спадкоємцем іншого класу.

Клас java.lang. Thread визначає ряд методів, які використовуються для управління потоками. До них відносяться статичні методи, які надають інформацію про стан потоку або змінюють стан потоку. Інші методи викликаються з інших потоків і дозволяють управляти поточним потоком. Розглянемо використання деяких методів. Спочатку створим імплементацію Runnable:

public class MyRunnable **implements Runnable** { @Override

public void run() { System.out.println("MyRunnable start");

/*Отримання посилання на поточний потік*/ Thread thread = **Thread.currentThread()**; for (int i = 0; i < 3; i++) {

System.out.println("MyRunnable name=" +

thread.getName() + ", id="

+ **thread.getId());**}

System.out.println("MyRunnable end");

}

public static void main(String[] args) { System.out.println("main method start"); Thread thread = **Thread.currentThread()**;

System.out.println("main name=" + **thread.getName()**

+ ", id=" + **thread.getId()**); **MyRunnable tr1 = new MyRunnable();**

/*Об’єкт MyRunnable не запускатиметься у окремому потоці, оскільки він не потік, а завдання*/

// tr1.run();

Thread th1 = new Thread(tr1);

/*Запуск завдання MyRunnable у новому потоці*/

th1.start();

System.out.println("main method end");

}

}

Ми використовуємо статичний метод currentThread() класу java.lang.Thread для отримання посилання на поточний потік, після чого використовуємо нестатичні методи getName() та getId() для отримання унікальних ідентифікаторів потоку. Зазначимо, що безпосередній запуск об’єкта MyRunnable викликом з нього методу run() приводить довиконання завдання у потоці метода main без створення нового потоку. Тому правильно запускати об’єкт Runnable через створення об’єкту Thread з передачею його конструктору об’єкта Runnable.

Запуск програми виведе таку інформацію:

main method start main name=main, id=1 main method end MyRunnable start

MyRunnable name=Thread-0, id=12 MyRunnable name=Thread-0, id=12 MyRunnable name=Thread-0, id=12 MyRunnable end

Бачимо, що потік main запустився та завершився, а завдання MyRunnable ви конується у іншому потоці Thread-0.

Створимо користувацький потік успадкуванням від класу java.lang.Thread та перевизначенням його методу run():

public class MyThread **extends Thread** { @Override

public void run() { System.out.println("MyThread start"); for (int i = 0; i < 3; i++) {

System.out.println("MyThread name="+ **this.getName()**+ ", id=" + **this.getId()**);

}

System.out.println("MyThread end");

}

public static void main(String[] args) { System.out.println("main method start"); Thread thread = **Thread.currentThread()**;

System.out.println("main name=" + **thread.getName()**

+ ", id=" + **thread.getId()**); **MyThread th1 = new MyThread();**

/*Thread.run() запускає метод run() в поточному потоці, а Thread.start() створює новий потік і запускає в ньому MyRunnable.run()*/

// th1.run();

th1.start();

/*Спроба повторного запуску потоку виконання без створення нового об'єкта потоку призводить до виключення IllegalThreadStateException */

// th1.start(); System.out.println("main method end");

}

}

Зазначимо, що для повторного запуску потоку, потрібно знову створювати його об’єкт, у противному разі виникне IllegalThreadStateException.

Методи для призупинення виконання потоку – sleep(), join(), yield()

Статичний метод void sleep(long millis) змушує поточний потік призупинити виконання протягом зазначеного інтервалу часу (в мілісекундах). Виклик цього методу використовується для припинення поточного потоку і надання процесорного часу іншим потокам (наприклад, для дотримання черговості завершення потоків). Метод може викликати Interrupted Exception у разі, коли інший потік буде "переривати його сон" викликом методу void interrupt() з об’єкту потоку, який "спить".

public class MyThread extends Thread { @Override

public void run() {System.out.println("MyThread start"); for (int i = 0; i < 3; i++) {

System.out.println("MyThread name="+ this.getName()+ ", id=" + this.getId());

try {

System.out.println("Чекаємо 1 сек...");

/*Поточний потік призупиняється на 1 сек*/

Thread.sleep(1000);

} catch (InterruptedException ex) { ex.printStackTrace();

}

}

System.out.println("MyThread end");

}

public static void main(String[] args) { System.out.println("main method start"); MyThread th1 = new MyThread(); System.out.println("MyThread created");

th1.start(); //створюється потік виконання Thread-0 try {

/*Поточний потік main призупиняється на 0,5 сек*/

Thread.sleep(500);

/*Намагаємось перервати потік Thread-0, який спить 1 сек*/

th1.interrupt();

System.out.println("Чекаємо 3 сек...");

/*Поточний потік main призупиняється на 3 секунди, щоб main завершився після Thread-0*/

Thread.sleep(3000);

} catch (InterruptedException ex) { ex.printStackTrace();

}

System.out.println("main method end");

}

}

Ми бачимо, що виникло виключення InterruptedException і що потік main завершився після потоку користувача.

Існує перевантажена версія методу public static void sleep(long millis, int nanos), яка приймає в якості першого параметра мілісекунди, а в якості другого наносекунди. Однак точність часових інтервалів не гарантується, тому що вони залежать від реалізації таймера операційної системи.

Нестатичний метод public final void join() throws InterruptedException класу Thread дозволяє поточному потоку призупинитись доки не буде виконаний потік, з якого викликається цей метод, а після завершення роботи цього потоку "приєднатися" і завершити роботу поточного потоку. У попередньому прикладі ми переводили потік main у сон на певний час, щоб дати закінчитись потоку Thread-0 першому, вочевидь використання join() для такої мети зручніше, оскільки не потрібно вираховувати час, як для метода sleep. Наведемо приклад використання методу join():

public static void main(String[] args) { System.out.println("main method start"); MyRunnable run = new MyRunnable(); Thread th = new Thread(run); th.start();

try {

/*Припинити виконання потоку main до моменту закінчення роботи потоку th*/

th.join();

} catch (InterruptedException ex) { ex.printStackTrace();

}

System.out.println("main method end");

}

Бачимо, що потік main завершився після потоку Thread-0. Існують перевантажені версії методу public final synchronized void join(long millis) та public final synchronized void join(long millis, int nanos), які приймають як параметр час, протягом якого виконується очікування завершення потоку, з якого викликаний join. Якщо потік не завершується, то поточний потік відновлюється і завершує свою роботу, а потім завершується потік, з якого викликався join.

Статичний метод yield() класу Thread викликається тільки для поточного потоку і інформує планувальник потоків про те, що поточний потік готовий відмовитись від використання процесора на момент виклику метода, але бажав би, щоб його запланували якомога скоріше. Планувальник потоків може вільно дотримуватися або ігнорувати цю інформацію і насправді має різну поведінку в залежності від операційної системи. Метод yield() викликається тільки для поточного потоку і при його виклику планувальник потоків призупиняє роботу поточного потоку віддає квант часу іншому потоку, поточний потік переміщується вниз черги потоків з рівним пріоритетом. Насправді, метод yield() не гарантує що поточний потік буде призупинений, і навіть якщо це буде виконано, поточний потік може бути знову обраний для виконання.

Наступний фрагмент коду відображає два потоки, які після запуску поступаються процесором один одному:

public static void main(String[] args) { Runnable r = () -> {

int counter = 0; while(counter < 2){

System.out.println(Thread.currentThread().getName());

counter++;

Thread.yield();

```
}
};
new Thread(r).start(); new Thread(r).start();
}
```

Метод Thread.yield() може бути корисним, наприклад, коли потік очікує настання якої-небудь події і необхідно, щоб перевірка його настання відбувалася якомога частіше. В цьому випадку можна помістити перевірку події і метод Thread.yield() в цикл.

Стани потоків

Потік може існувати в одному з наступних станів (рис. 5.2):

NEW - об'єкт Thread створений, але потік ще не запущений. Метод isAlive(), викликаний об'єкта-потоку, повертає false.

RUNNABLE - Потік готовий виконуватися і знаходиться в пулі потоків пла- нувальника, але планувальник потоків не обрав його для виконання. Метод isAlive(), викликаний з об'єкта-потоку, повертає true.

RUNNINIG - Потік обраний планувальником потоків з пулу і виконується. Метод isAlive(), викликаний з об'єкта-потоку, повертає true.

BLOCKED - Робота потоку припинена доки не буде доступний монітор об'єкта, до якого забезпечується доступ з потоку. Метод isAlive(), викликаний з об'єкта-потоку, повертає true.

WAITING - Потік очікує певної дії іншого потоку невизначений час. Метод isAlive(), викликаний з об'єкта-потоку, повертає true.

TIMED_WAITING - Потік очікує певної дії іншого потоку протягом зазначеного часу. Метод isAlive(), викликаний з об'єкта-потоку, повертає true.

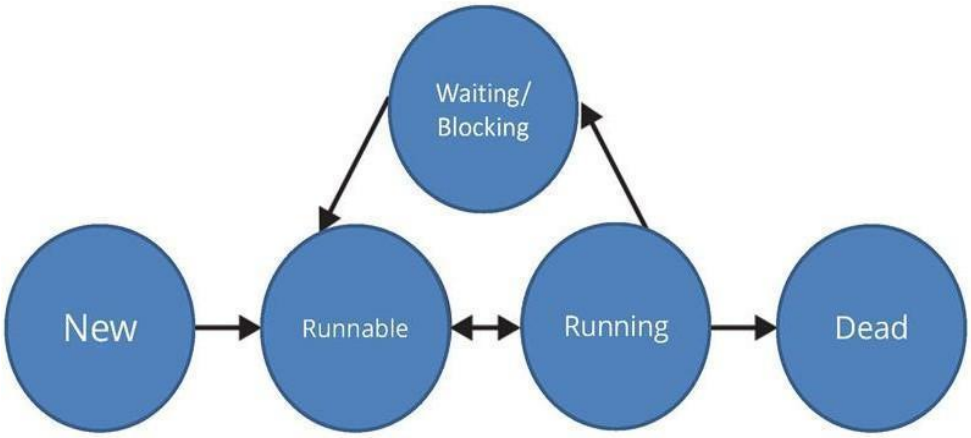


Рисунок 6.2. Стани потоків виконання

TERMINATED (DEAD) - Потік завершений. Метод isAlive(), викликаний з об'єкта-потоку, повертає false.

Наведемо код програми, яка відображає зміну станів потоку користувача (діагностичний метод printInfo виводить стани потоку та значення, що повертає метод isAlive()):

```
public class ThreadStatesTest extends Thread {
@Override
public void run() {try {
System.out.println(getName() + " sleep(50)"); Thread.sleep(50);
} catch (InterruptedException ex) { ex.printStackTrace();
}
System.out.println(getName() + " finished");
}
public static void main(String[] args) { try {
Thread t = new ThreadStatesTest(); System.out.println(t.getName() + " is created"); printInfo(1, t);
System.out.println(t.getName() + " start()"); t.start();
printInfo(2, t); System.out.println(Thread.currentThread().
getName() + " sleep(10)");
sleep(10); printInfo(3, t);
/*Призупиняє виконання поточного потоку main до завершення потоку t*/
System.out.println(t.getName() + " t.join()"); t.join();
printInfo(4, t);
} catch (InterruptedException ex) { ex.printStackTrace();
}
System.out.println(Thread.currentThread().getName()
+ " finished");
}

private static void printInfo(int count, Thread t) { System.out.println(String.valueOf(count) + ": "
+ t.getName() + ", State: " + t.getState()
+ ", isAlive=" + t.isAlive());
}
}
```

Зверніть увагу, що після старту потоку, метод getState() повертає для нього стан RUNNABLE, а не RUNNING. Річ у тому, що запущений потік може бути, а може і не бути фактично запущеним, оскільки він планується операційною системою і неможливо розрізнити ці два стани за допомогою Java. Тому метод getState() завжди повертає для запущеного потоку стан RUNNABLE [7]. Пере- хід до стану BLOCKED буде розглянутий далі при вивченні використання методів wait/notify.

Планувальник потоків JVM використовує багатозадачність з витисненням потоків і з пріоритетом потоків. Пріоритет можна розглядати як кількість тайм- слотів, що виділяються потоку при виклику його з пулу потоків для виконання (рис. 6.2).

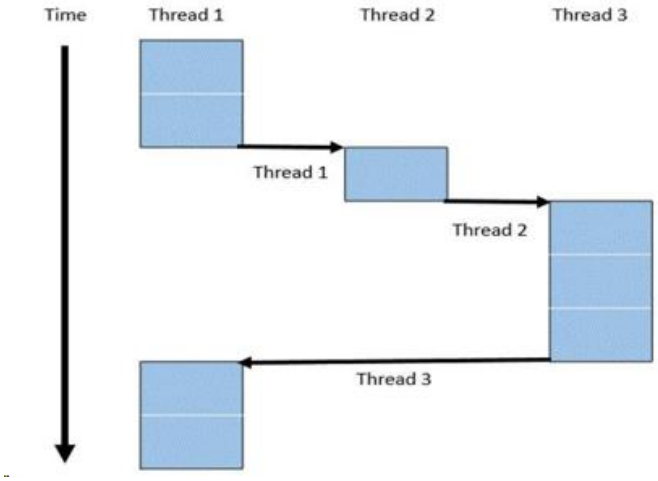


Рисунок 6.3. Пріоритети потоків

Зазвичай планувальник потоків працює наступним чином: при попаданні потоку в пул (зміні його стану на RUNNABLE), якщо його пріоритет вище за пріоритет інших потоків в пулі і вище пріоритету потоку, який виконується, виконуваний потік буде припинений і поміщений в пул (перейде в стан RUNNABLE), а доданий у пул потік з вищим пріоритетом буде запущений на виконання (перейде в стан RUNNING). Тобто, зазвичай, потік, що виконується, має вищий (або такий же) пріоритет, ніж потоки в пулі. На практиці обсяг часу процесора, який отримує потік, часто залежить від декількох факторів, окрім його пріоритету (наприклад, від того, як операційна система реалізує багатозадачність). Наведемо приклад коду, що дозволяє дослідити обрання планувальником потоків з різними пріоритетами для виконання:

public class MyTestThread extends Thread { private double d;

```
@Override
public void run() {
for (int i = 1; i < 10000000; i++) {                    //тяжке
//обчислювальне завдання d += (Math.PI + Math.E) / (double) i;
}
System.out.println("Thread :" + getName()
+ ", Priority=" + getPriority());
}
public static void main(String[] args) {
int numThreads = 8; //повинно бути парним MyTestThread[] threads =
new MyTestThread[numThreads]; for (int i = 0; i < numThreads; i = i + 2)
{
threads[i] = createThread(Thread.MIN_PRIORITY); threads[i + 1] =
createThread(Thread.MAX_PRIORITY);
}
for (MyTestThread thread : threads) { thread.start();
}
}
private static MyTestThread createThread(int priority) { MyTestThread th
= new MyTestThread(); th.setPriority(priority);
return th;
}
}
```

Запуск програми виводить таку (очікувану) інформацію:

Thread :Thread-7, Priority=10 Thread :Thread-3, Priority=10 Thread :Thread-5, Priority=10 Thread :Thread-1, Priority=10 Thread :Thread-2, Priority=1 Thread :Thread-0, Priority=1 Thread :Thread-6, Priority=1 Thread :Thread-4, Priority=1

Запитання для самоперевірки

1. Опишіть спосіб створення та запуску потоку із використанням інтерфейсу java.lang.Runnable.
2. Опишіть спосіб створення та запуску потоку із використанням інтерфейсу java.lang.Thread.
3. Поясніть, що являє собою пріоритет потоків і як він може встановлюватись. Як планувальник потоків враховує їх пріоритет?
4. В чому полягає різниця між інтерфейсами Runnable і Callable?

5. Які методи зупинки потоку вам відомі?
6. Опишить ситуацію коли варто викликати метод Thread.yield().

Лекція 5.
Механізми синхронного доступу до спільного ресурсу

Методи синхронизації

Отже, в Windows виконуються не процеси, а потоки. При створенні процесу автоматично створюється його основний потік. Цей потік в процесі виконання може створювати нові потоки, які, у свою чергу, теж можуть створювати потоки і так далі. Процесорний час розподіляється саме між потоками, і виходить, що кожен потік працює незалежно.

Усі потоки, що належать одному процесу, розділяють деякі загальні ресурси - такі, як адресний простір оперативної пам'яті або відкриті файли. Що станеться, якщо один потік ще не закінчив працювати з яким-небудь загальним ресурсом, а система перемкнулася на інший потік, що використовує той же ресурс? Відбудеться конфлікт й результат роботи цих потоків буде надзвичайно сильно відрізнятися від задуманого. Такі конфлікти можуть виникнути й між потоками, що належать різним процесам. Завжди, коли два або більше потоки використовують який-небудь загальний ресурс, виникає ця проблема.

Саме тому потрібний механізм, що дозволяє потокам погоджувати свою роботу із загальними ресурсами. Цей механізм отримав назву механізму синхронізації потоків (thread synchronization).

Це набір об'єктів операційної системи, які створюються і управляються програмно, є загальними для усіх потоків в системі (деякі - для потоків, що належать одному процесу) та використовуються для координування доступу до ресурсів. Ресурсами може виступати усе, що може бути загальним для двох й більше потоків - файл на диску, порт, запис в базі даних, об'єкт gdi, і навіть глобальна змінна програми (яка може бути доступна з потоків, що належать одному процесу).

Об'єктів синхронізації існує декілька, найважливіші з них – це

- критична секція (critical section);
- бар’єри;
- взаємовиключення (mutex);
- семафор (semaphore).

Кожен з цих об'єктів реалізує свій спосіб синхронізації. Також, як

об'єкти синхронізації можуть використовуватися самі процеси й потоки (коли один потік чекає завершення іншого потоку або процесу), а також файли, комунікаційні пристрої, консольне введення й повідомлення про зміну .

У чому сенс об'єктів синхронізації? Кожен з них може знаходитися в так званому сигнальному стані. Для кожного типу об'єктів цей стан має різний сенс. Потоки можуть перевіряти поточний стан об'єкту й/або чекати зміни цього стану і таким чином погоджувати свої дії. Гарантується, що коли потік працює з об'єктами синхронізації (створює їх, змінює стан) система не перерве його виконання, поки він не завершить цю дію. Таким чином, усі кінцеві операції з об'єктами синхронізації є атомарними (неподільними), такими що виконуються за один такт.

Важливо розуміти, що ніякого реального зв'язку між об'єктами синхронізації та ресурсами немає. Вони не зможуть запобігти небажаному доступу до ресурсу, вони лише підказують потокам, коли можна працювати з ресурсом, а коли треба почекати. Можна провести аналогію зі світлофорами - вони показують, коли можна їхати, але ж в принципі водій може й не звернути уваги на червоне світло.

Поняття критичної секції. Задача критичної секції.

Критичні секції забезпечують синхронізацію подібно м’ютексам за винятком того, що об'єкти, які представляють критичні секції, доступні в межах одного процесу. Події, м’ютекси та семафори також можна використовувати в «однопроцесорному» застосуванні, проте критичні секції забезпечують швидший та ефективніший механізм синхронізації. Подібно м’ютексам об'єкт, що представляє критичну секцію, може використовуватися тільки одним потоком в даний момент часу, що робить їх корисними при розмежуванні доступу до загальних ресурсів. Важко припустити що-небудь про порядок, в якому потоки будуть отримувати доступ до ресурсу, можна сказати лише, що система буде «справедливою» до усіх потоків. Критична секція (Critical Section) це ділянка коду, в якій потік (thread) дістає доступ до ресурсу (наприклад змінної), який доступний з інших потоків.

Задача критичної секції (КС) — це одна з класичних задач паралельного програмування. Вона була першою всебічно вивченою задачею, але інтерес до неї не згасає, оскільки критичні секції коду є в більшості паралельних програм. Крім того, розв’язок цієї задачі можна використовувати для реалізації операторів await.

У задачі критичної секції n процесів багаторазово виконують спочатку критичну, а потім некритичну секцію коду. Критичній секції передує протокол входу, а за нею йде протокол виходу. Таким чином, передбачається, що процес має наступний вигляд:

process CS [i = 1 to n] { while (true) {

протокол входу; критична секція; протокол виходу; некритична секція;

}

}

Кожна критична секція є послідовністю операторів, що мають доступ до деякого спільного об'єкта. Кожна некритична секція — це ще одна послідовність операторів. Передбачається, що процес, який увійшов в критичну секцію, обов'язково коли-небудь з неї вийде; таким чином, процес може завершитися тільки поза критичною секцією. Завдання — розробити протоколи входу і виходу, які задовольняють наступним властивостям:

(1) Взаємне виключення. У будь-який момент часу тільки один процес може виконувати свою критичну секцію.

(2) Відсутність взаємного блокування (живого блокування). Якщо кілька процесів намагаються увійти в свої критичні секції, хоча б один це здійснить.

(3) Відсутність зайвих затримок. Якщо один процес намагається увійти в свою критичну секцію, а інші виконують свої некритичні секції або завершені, першому процесу дозволяється вхід в критичну секцію.

(4) Можливість входу. Процес, який намагається увійти в критичну секцію, коли-небудь це зробить.

Перші три властивості є властивостями безпеки, четверта — властивістю живучості.

Тривіальний спосіб вирішення задачі критичної секції полягає в обмеженні кожної критичної секції кутовими дужками, тобто у використанні безумовних операторів await. З семантики кутових дужок відразу випливає умова (1) — взаємне виключення. Інші три властивості задовольнятимуться при безумовно справедливій стратегії планування, оскільки вона гарантує, що процес, який намагається виконати неподільну дію, відповідну його критичної секції, в кінці кінців це зробить, незалежно від дій інших процесів. Однак при такому «розв'язку» виникає питання про те, як реалізувати кутові дужки.

Всі чотири зазначених властивості важливі, однак найбільш істотним

є взаємне виключення. Для опису властивості взаємного виключення необхідно визначити, чи знаходиться процес у своїй критичній секції. Щоб спростити запис, розглянемо розв'язок задачі у випадку двох процесів, CS1 і CS2; він легко узагальнюється для n процесів.

Нехай in1 та in2 — логічні змінні з початковим значенням false. Якщо процес CS1 (CS2) знаходиться в своїй критичній секції, змінній in1 (in2) присвоюється значення true. Поганий стан, якого ми будемо намагатися уникнути, — якщо обидві змінні in1 та in2 є істинними. Таким чином, нам потрібно, щоб для будь-якого стану виконувалося заперечення умови поганого стану:

MUTEX: !(in1 && in2)

Предикат MUTEX має бути глобальним інваріантом. Для цього він повинен виконуватися в початковому стані і після кожного присвоювання змінним in1 та in2. Зокрема, перед тим, як процес CS1 увійде в критичну секцію, зробивши тим самим in1 істинною, він повинен переконатися, що in2 хибна. Це можна реалізувати так:

await (!in2) in1 = true;

Вхідний протокол процесу CS2 аналогічний. При виході з критичної секції затримуватися ні до чого, тому захищати оператори, які надають змінним in1 та in2 значення false, немає необхідності.

Розв'язок наведено у наступній програмі:

```
bool in1 = false, in2 = false; process CS1 {
while (true) {
await (!in2) in1 = true;        # вхід
критична секція;
in1 = false;   # вихід
некритична секція;
}
}

process CS2 { while (true) {
await (!in1) in2 = true;        # вхід критична секція;
in2 = false;   # вихід некритична секція;
}
}
```

За побудовою програма задовольняє умові взаємного виключення. Взаємне блокування тут не виникне: якби кожен процес був заблокований в

своєму протоколі входу, то обидві змінні, і in1, і in2, були б істинними, а це суперечить тому, що в даній точці коду обидві вони хибні. Зайвих затримок також немає, оскільки один процес блокується, тільки якщо інший перебуває в критичній секції, тому небажані паузи при виконанні програми не виникають.

Нарешті, розглянемо властивість живучості: процес, який намагається увійти в критичну секцію, в кінці кінців зможе це зробити. Якщо процес CS1 намагається увійти, але не може, то змінна in2 істинна, і процес CS2 знаходиться в критичній секції. За припущенням процес врешті-решт виходить з критичної секції, тому змінна in2 коли-небудь стане хибною, а змінна захисту входу процесу CS1 — істинної.

Якщо процесу CS1 вхід все ще не дозволено, це означає, що або диспетчер несправедливий, або процес CS2 знову досяг входу в критичну секцію. В останньому випадку описаний вище сценарій повторюється, так що коли-небудь змінна in2 стане хибною. Таким чином, або змінна in2 стає хибною нескінченно часто, або процес CS2 завершується, і змінна in2 приймає значення false і залишається в такому стані. Для того щоб процес CS1 в будь-якому випадку входив в критичну секцію, потрібно забезпечити справедливу в сильному сенсі стратегію планування.

Критична секція являє собою блок коду, який отримує доступ до загального ресурсу і не може виконуватися більш ніж одним потоком одночасно. Коли потік хоче отримати доступ до критичної секції, він використовує один з механізмів синхронізації, що надаються Java, щоб з'ясувати, чи виконує зараз критичну секцію будь-якої іншої потік. Якщо жоден з потоків не виконує критичну секцію, то потік входить в критичну секцію. В іншому випадку, потік призупиняється механізмом синхронізації, доки потік, виконуючий критичну секцію, не завершить її виконання. У разі, якщо більше одного потоку очікують завершення виконання критичної секції потоком, планувальник потоків обирає один з них (випадково, якщо потоки мають рівні пріоритети, або відповідно до пріоритетів), а інші чекають своєї черги.

У якості критичної секції часто вибирають метод, який виконує читання/запис загального ресурсу. У такому разі при оголошенні метода вказують модифікатор synchronized і метод називають *синхронізованим*. Модифікатор synchronized:

- не дозволяє виконувати тіло методу більш ніж одному потоку одночасно (запобігає взаємному впливу потоків);

- забезпечує збереження результуючих значень локальних змінних і ре зультату методу в пам'яті так, щоб вони були видні для інших потоків (запобігає неузгодженості пам'яті).

Наведемо приклад, коли синхронізація методу виправляє помилкову роботу програми. Нехай є клас-лічильник з несинхронізованим методом void increment(), який збільшує на 1 значення загального ресурсу - змінної cnt:

```
public class MyCounter {
private long cnt = 0; public void increment() {
cnt++;
}
public long getValue() { return cnt;
}
}
```

Також у класі є гетер для отримання значення змінної. Нехай до методу

increment() будуть мати доступ потоки, визначені у класі MyCounterThread, полями якого є посилання на об’єкт клас-лічильника та кількість викликів метода increment():

```
public class MyCounterThread extends Thread { MyCounter counter;
int n;
public MyCounterThread(MyCounter counter, int n) { this.counter = counter;
this.n = n;
}
@Override
public void run() {
for (int i = 0; i < n; i++) { counter.increment();
}
}
public static void main(String[] args) { int numOfThreads = 100;
int incrementCalls = 1000; MyCounter m = new MyCounter();
MyCounterThread[] tg =new MyCounterThread[numOfThreads]; for (int i = 0; i < numOfThreads; i++) {
tg[i] = new MyCounterThread(m, incrementCalls);
}
for (MyCounterThread t : tg) { t.start();
```

```
}
/*Розміщуємо всі створені потоки попереду потоку main*/
try {
for (MyCounterThread t : tg) { t.join();
}
} catch (InterruptedException ex) { ex.printStackTrace();
}
System.out.println(m.getValue());
System.out.println(Thread.currentThread().getName()+ " finished");
}
}
```

Запуск програми виведе:

92489

main finished

Бачимо, що результат інкременту менший за очікуваний (100 потоків по 1000 викликів методу повинні були дати 100000). Помилки для 100 потоків ви- дно зі 100 і більше викликів (залежить від комп’ютера, чим більше значення, тим більша різниця між отриманим та очікуваним результатом). Розбіжність результатів пояснюється тим, що можливе зчитування значення змінної- лічильника cnt іншим потоком перед збереженням в пам'ять його інкрементованого значення першим потоком. В результаті значення лічильника виявляється меншим загальної кількості інкрементів.

Додавання модифікатора synchronized в оголошенні методу increment() позначає його тіло як критичну секцію і забезпечує одночасний доступ до методу тільки для одного потоку.

```
public class MyCounter { private long cnt = 0;
public synchronized void increment() { cnt++;
}
public long getValue() { return cnt;
}
```

В результаті значення лічильника в точності дорівнює добутку кількості потоків та кількості викликів методу. Зазначимо, що метод getValue() не потребує синхронізації, тому що до нього йде звернення тільки з одного потоку main.

Додання модифікатора synchronized до нестатичного метода класу означає, що потік, який намагається викликати такий метод з об’єкту класу,

повинен захопити монітор цього об'єкта, перш ніж виконувати будь-який код цього методу.

Метод М'ютекс

Об'єкти взаємовиключення (м'ютекси, mutex - від mutual exclusion) дозволяють координувати взаємне виключення доступу до ресурсу, що розділяється. Сигнальний стан об'єкту (тобто стан «встановлений») відповідає моменту часу, коли об'єкт не належить жодному потоку і його можна «захопити». І навпаки, стан «скинутий» (не сигнальне) відповідає моменту, коли який-небудь потік вже володіє цим об'єктом. Доступ до об'єкту дозволяється, коли потік, що володіє об'єктом, звільнить його. Для того, щоб оголосити, взаємовиключення таким, що належить поточному потоку, потрібно викликати одну із очікуючих функцій. Потік, якому належить об'єкт, може його «захоплювати» повторно скільки завгодно раз (це не приведе до самоблокування), але стільки ж раз він повинен буде його звільняти за допомогою функції releasemutex. М'ютекс (взаємовиключення, mutex) - це об'єкт синхронізації, який встановлюється в особливий сигнальний стан, коли не зайнятий яким- небудь потоком. Тільки один потік володіє цим об'єктом у будь-який момент часу, звідси і назва таких об'єктів - одночасний доступ до загального ресурсу виключається. Наприклад, щоб виключити запис двох потоків в загальну ділянку пам'яті в один і той же час, кожен потік чекає, коли звільниться м'ютекс, стає його власником і тільки потім пише що- небудь в цю ділянку пам'яті. Після усіх необхідних дій м'ютекс звільняється, надаючи іншим потокам доступ до загального ресурсу. Два (чи більш) процеси можуть створити м'ютекс з одним і тим же ім'ям, викликавши метод CreateMutex. Перший процес дійсно створює м'ютекс, а наступні процеси отримують хендл вже існуючого об'єкту. Це дає можливість декільком процесам отримати хендл одного і того ж м'ютекса, звільняючи програміста від необхідності піклуватися про той, хто насправді створює м'ютекс. Якщо використовується такий підхід, бажано встановити прапор bInitialOwner у FALSE, інакше виникнуть певні труднощі при визначенні дійсного власника м'ютекса. Декілька процесів можуть отримати хендл одного і того ж м'ютекса, що робить можливою взаємодію між процесами. Mutex дозволяє проводити синхронізацію не лише між потоками (thread), але і процесами (process), тобто між застосуваннями. Цей об'єкт синхронізації реєструє доступ до ресурсів і може знаходитися в двох станах:

- встановлений

- скинутий

Mutex - встановлений в той момент коли ресурс вільний. Якщо до об'єкту є доступ, то говорять, що скинутий. Для роботи з Mutex є ряд функцій. Спочатку об'єкт треба створити CreateMutex(), для доступу OpenMutex(), а для звільнення ресурсу ReleaseMutex(). Для доступу до об'єкту Mutex використовується функція WaitForSingleObject(). Тонкість тут наступна. Кожна програма створює об'єкт Mutex за ім’ям. Тобто Mutex це іменований об'єкт. А якщо такий об'єкт синхронізації вже створила інша програма, то за викликом CreateMutex() ми отримаємо покажчик на об'єкт, який вже створила перша програма. Тобто у обох програм буде один і той же об'єкт. Ось це і дозволяє проводити синхронізацію

Захищені методи та захищені блоки

Синхронізованими можуть бути і статичні методи, при їх викликах потоки попередньо захоплюють монітор об’єкту класу Class з метаданими класу, у якому розміщений цей статичний метод.

Проте одночасне розташування синхронізованих статичних і нестатичних методів в одному класі розглядається як погана практика, оскільки можливий доступ двох потоків одночасно: одного до нестатичних, а одного до статичного синхронізованому методу, і, отже, можливі помилки читання-зміни стану, що зберігається і в змінних класу, і в змінних об'єкта.

Конструктори не можуть бути синхронізовані. Синхронізація конструкторів не має сенсу, тому що тільки потік, який створює об'єкт, повинен мати доступ до цього об'єкта.

Синхронізація методів призводить до деякого збільшення часу обчислень порівняно з несинхронізованими методами, але запобігає помилкам.

Синхронізовані блоки - це блоки коду (обрамляються фігурними дужками), які синхронізуються на якомусь об'єкті (використовують його монітор для синхронізації). Початок синхронізованого блоку коду помічається модифікатором synchronized з вказанням у дужках імені об’єкта, чий монітор використовується для синхронізації. В якості монітора (об'єкта, що забезпечує блокування) може використовуватись і поточний об'єкт у такому разі перед блоком використовується модифікатор synchronized(this). Якщо декілька синхронізованих блоків синхронізовані на тому ж самому об’єкті, то лише один потік може мати доступ до коду у всіх таких блоках одночасно.

Наведемо приклад використання синхронізованого блоку. Нехай є

клас, що описує банківський рахунок користувача зі змінною money, доступ до якої може виконуватися декількома потоками:

```
public class UserAccount { private int money;
public UserAccount(int money) { this.money = money;
}
public int getMoney() { return money;
}
public void setMoney(int money) { this.money = money;
}
}
```

Клас UserAction є потоком, який реалізує задачу зняття певної суми, вказаної змінною withdraw, з рахунку користувача:

```
public class UserAction extends Thread { private UserAccount acc;
private int withdraw;
public UserAction(UserAccount acc, int withdraw) { this.acc = acc;
this.withdraw = withdraw;
}
@Override
public void run() {
int has = acc.getMoney(); try {
sleep(1);
} catch (InterruptedException ex) { ex.printStackTrace();
}
if (has >= withdraw) { acc.setMoney(acc.getMoney() - withdraw);
System.out.println("Get " + withdraw
+ " from the account");
} else {
System.out.println("Not enough money");
}
}
}
```

Тоді в деякій програмі можемо організувати зняття певних сум, наприклад по 100 у.о., з рахунку кожним потоком:

```
public static void main(String[] args) {
/*Початкова сума - 500 у.о.*/ UserAccount acc = new UserAccount(500);
for (int i = 0; i < 5; i++) {
```

```
/*Створення потоків, які знімають по 100 у.о. з рахунку*/
UserAction act = new UserAction(acc, 100); act.start();
}
try {
Thread.sleep(1000);
} catch (InterruptedException ex) { ex.printStackTrace();
}
System.out.println("Баланс = " + acc.getMoney());
}
```

Хоча очікуваний баланс після зняття суми у 500 у.о. п’ятьома потоками по 100 у.о. дорівнює нулю, запуск програми показує більші нуля значення.

Get 100 from the account Get 100 from the account Get 100 from the account Get 100 from the account Get 100 from the account Баланс = 200

Причина - перший за часом потік не встигає записати у загальну зміну модифіковане значення до того, як її зчитує інший потік. Оскільки читання та запис загальної змінної потоками виконується у різних методах, потрібно дозволити доступ к обом методам тільки одному потоку одночасно. Це можна організувати створенням у методі run() класу UserAction синхронізованого блоку, у якому викликаються обидва методи, з використанням монітору об’єкту UserAccount:

```
@Override
public void run() {
synchronized(acc) {
int has = acc.getMoney(); try {
sleep(1);
} catch (InterruptedException ex) { ex.printStackTrace();
}
if (has >= withdraw) { acc.setMoney(acc.getMoney() - withdraw);
System.out.println("Get " + withdraw

+ " from the account");
} else {
System.out.println("Not enough money");
}
}
}
```

Тепер запуск програми виводить правильну інформацію:

Get 100 from the account Get 100 from the account Get 100 from the account Get 100 from the account Get 100 from the account Баланс = 0

Якщо порівняти синхронізовані методи та синхронізовані блоки то можна зазначити:

Синхронізований блок може зменшувати область блокованого коду. Оскільки область блокування зворотно пропорційна продуктивності, завжди краще блокувати лише критичний розділ коду.

Синхронізований блок може об’єднувати у області блокованого коду роботу з декількома методами. При оголошені цих методів синхронізованими не- має можливості забезпечити для потоку атомарність їх виконання.

Для синхронізованого блоку можна використовувати довільний об’єкт, що забезпечує блокування. З іншого боку, синхронізований метод завжди блокує або поточний об’єкт або об’єкт класу Class, якщо це статичний синхронізований метод.

Синхронізований блок може викидати NullPointerException, якщо об’єкт, який надається блоку як параметр, має значення null. Така ситуація неможлива для синхронізованих методів.

Використання синхронізованого блоку на відміну від синхронізованого методу дозволяє в одному класі для різних методів використовувати різні об'єкти (монітори).

Рекомендується переважне використання синхронізованих блоків.

Бар’єрна синхронізація. Спільний лічильник. Керуючі процеси.

Основною властивістю більшості паралельних ітераційних алгоритмів є залежність результатів кожної ітерації від результатів попередньої. Один із способів побудувати такий алгоритм — реалізувати тіло кожної ітерації, використовуючи оператори co. Якщо вважати, що на кожній ітерації виконується n задач, отримаємо такий загальний вигляд алгоритму:

```
while (true) {
co [i = 1 to n]
код розв’язку задачі i;
oc
}
```

На жаль, цей підхід досить неефективний, оскільки оператор co породжує n процесів на кожній ітерації. Створювати і знищувати процеси набагато дорожче, ніж реалізувати їх синхронізацію. Тому альтернативна

структура алгоритму робить його набагато ефективнішим — процеси створюються один раз на початку обчислень, а потім синхронізуються в кінці кожної ітерації.

```
process Worker[i = 1 to n] { while (true) {
код розв’язку задачі i; очікування завершення усіх задач;
}
}
```

Точка затримки в кінці кожної ітерації є бар’єром, якого для продовження роботи повинні досягти всі процеси, тому цей механізм називається бар’єрною синхронізацією. Бар’єри можуть знадобитися в кінці циклів або на проміжних стадіях. Нижче розглянуто кілька реалізацій бар'єрної синхронізації, що використовують різні способи взаємодії процесів.

Спільний лічильник

Найпростіший спосіб описати вимоги до бар’єра — використовувати лічильник count з нульовим початковим значенням. Припустимо, що є n робочих процесів, які повинні зібратися біля бар'єру. Коли процес доходить до бар'єру, він збільшує значення count. Коли значення count стане рівним n, всі процеси зможуть продовжити роботу:

```
process Worker[i = 1 to n] { while (true) {
код розв’язку задачі i;
count = count + 1;
await (count == n);
}
}
```

Проте цей код не повною мірою відповідає поставленому завданню. Складність полягає в тому, що значенням count повинен бути 0 на початку кожної ітерації, тобто count потрібно обнуляти кожного разу, коли всі процеси пройдуть бар’єр. Більш того, вона повинна мати значення 0 перед тим, як будь-який з процесів знову спробує її збільшити. Цю проблему можна вирішити за допомогою двох лічильників, один з яких збільшується до n, а інший зменшується до 0. Їх ролі міняються місцями після кожної стадії. Однак використання спільних лічильників призводить до чисто практичних труднощів. По-перше, збільшувати і зменшувати їх значення потрібно неподільним чином. По-друге, коли процес призупиняється, він безперервно перевіряє значення змінної count. У гіршому випадку n-1 процесів чекатимуть, поки останній процес досягне бар’єра. В результаті

виникне серйозний конфлікт звернення до пам’яті, якщо тільки програма не виконується на мультипроцесорній машині з узгодженою кеш-пам’яттю. Крім того, число n повинно бути відносно малим.

Керуючі процеси

Один із способів уникнути конфліктів звернення до пам’яті — реалізувати лічильник count за допомогою n змінних, значення яких додаються до одного і того ж значення. Нехай є масив цілих чисел arrive[1: n] з нульовими початковими значеннями. Замінимо операцію збільшення лічильника count в програмі так: arrive[i] = 1. Якщо елементи масиву arrive зберігаються в різних рядках кеш-пам'яті, то конфліктів звернення до пам’яті не буде. Залишилося реалізувати оператор await і обнулити елементи масиву arrive в кінці кожної ітерації. Оператор await можна записати в такому вигляді.

await ((arrive[l] + ... + arrive[n]) == n);

Але в такому випадку знову виникають конфлікти звернення до пам’яті, причому це рішення також неефективне, оскільки суму елементів arrive[i] тепер постійно обчислює кожний процес Worker, який очікує продовження.

Обидві проблеми можна вирішити, використовуючи додатковий набір

спільних значень і ще один процес, Coordinator. Нехай кожен процес Worker замість того, щоб додати усі елементи масиву arrive, чекає, поки не стане істинним логічне значення. Нехай continue[1: n] — додатковий масив цілих з нульовими початковими значеннями. Після того як Worker[i] присвоїть 1 елементу arrive[i], він повинен чекати, поки значенням змінної continue[i] не стане 1.

arrive[i] = 1; (await (continue[i] == 1);

Процес Coordinator очікує, поки всі елементи масиву arrive не стануть рівні 1, потім присвоює значення 1 всім елементам масиву continue.

for [i = 1 to n] □await (arrive[i] == 1);□ for [i = 1 to n] continue[i] = 1;

Оскільки для продовження процесів Worker повинні бути встановлені всі елементи arrive, процес Coordinator може перевіряти їх в будь-якому порядку. Конфліктів звернення до пам'яті тепер не буде, оскільки процеси очікують зміни різних змінних, кожна з яких може зберігатися в своєму рядку кеш- пам’яті. Змінні arrive та continue є прикладами так званого «прапорця». Його встановлює один процес, щоб повідомити інші про виконання умови синхронізації. Використовуються два основних правила синхронізації за допомогою прапорців:

а) прапорець синхронізації скидається тільки процесом, що очікує його установки;

б) прапорець не можна встановлювати до тих пір, поки невідомо точно, що він скинутий.

Перше правило гарантує, що прапорець не буде скинутий, поки процес не визначить, що він встановлений. Відповідно до цього правила прапорець continue[i] має скидатися процесом Worker[i], а обнуляти всі елементи масиву arrive має Coordinator. Згідно з другим правилом один процес не встановлює прапорець, поки він не скинутий іншим. В іншому випадку, якщо інший синхронізований процес надалі очікує повторного встановлення прапорця, можливе взаємне блокування. Це означає, що Coordinator повинен скинути arrive[i] перед установкою continue[i]. Coordinator може також скинути arrive[i] відразу після того, як дочекався його установки. Додавши код скидання прапорів, отримаємо бар'єр з керуючим процесом:

```
int arrive[l:n], continue[l:n]; process Worker[i = 1 to n] {
while (true) {
код розв'язку задачі i; arrive[i] = 1;
await (continue[i] == 1); continue[i] = 0;
}
}
process Coordinator { while (true) {
for [i = 1 to n] {
□await (arrive[i] == 1);□ arrive[i] = 0;
}
for [i = 1 to n] continue[i] = 1;
}
}
```

Хоча в програмі бар'єрна синхронізація реалізована так, що конфлікти звернення до пам'яті виключаються, у даного розв'язку є дві небажані властивості. По перше, потрібен додатковий процес. Синхронізація з активним очікуванням ефективна, якщо кожний процес виконується на окремому процесорі, так що процесу Coordinator потрібен свій власний процесор. Але, можливо, було б краще використовувати цей процесор для іншого робочого процесу. Другий недолік використання керуючого процесу полягає в тому, що час виконання кожної ітерації процесу Coordinator, і, отже, кожного екземпляра бар'єрної синхронізації пропорційно числу

процесів Worker. В ітераційних алгоритмах часто всі робочі процеси мають ідентичний код. Це означає, що якщо кожний робочий процес виконується на окремому процесорі, то всі вони підійдуть до бар'єра приблизно в один час. Таким чином, всі прапори arrive будуть встановлені практично одночасно. Проте процес Coordinator перевіряє прапорці в циклі, по черзі чекаючи, коли кожен з них буде встановлено. Обидві проблеми можна подолати, об'єднавши дії керуючого і робочих процесів так, щоб кожний робочий процес був одночасно і керуючим. Організуємо робочі процеси в дерево (рис. 7.1).

Сигнал про те, що процес підійшов до бар'єра (прапор arrive[i]), надсилається вгору по дереву, а сигнал про дозвіл продовження виконання (прапор continue[i]) — вниз. Вузол робочого процесу чекає, коли до бар'єра підійдуть його «діти», після чого повідомляє батьківський вузол про те, що він теж підійшов до бар'єра. Коли всі «діти» кореневого вузла підійшли до бар'єра, це означає, що все інші робочі вузли теж підійшли до бар'єра.

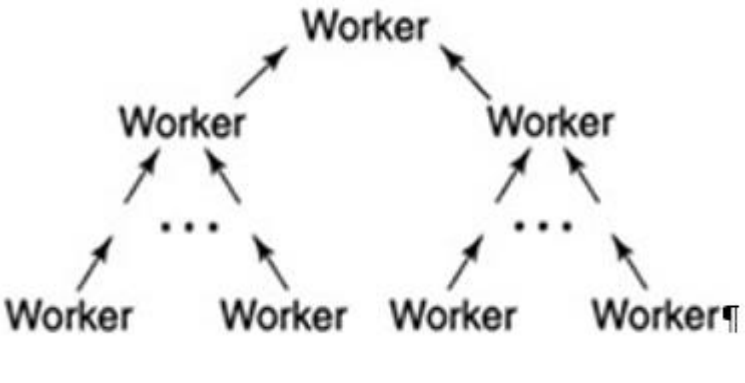


Рисунок 7.1. Деревоподібний бар'єр

Тоді кореневої вузол може повідомити нащадкам, що вони можуть продовжити виконання. Ті, в свою чергу, дозволяють продовжити виконання своїх синів, і так далі. Таким чином, отримуємо наступний розв'язок:

```
# Бар'єрна синхронізація за допомогою об'єднуючого дерева лист L:
arrive [L] = 1;
await (continue[L] == 1); continue[L] = 0;
проміжний вузол I:
await (arrive[left] == 1); arrive[left] = 0;
await (arrive[right] == 1) arrive[right] = 0;
arrive[I] = 1;
```

await (continue[I] == 1); continue[I] = 0;

continue[left] = 1; continue[right] = 1;

кореневий вузол R:

await (arrive[left] == 1) arrive[left] = 0;

await (arrive[right] == 1); arrive[right] = 0;

continue[left] = 1; continue[right] = 1;

Отримана реалізація називається бар'єром з об'єднуючим деревом, оскільки кожен процес об'єднує результати роботи своїх дочірніх процесів і відправляє батьківському.

Запитання для самоперевірки

1. Поясніть різницю між методами lock.lock() і lock.tryLock() класу ReentrantLock?
2. Як можна забезпечити справедливість між потоками під час блокування за допомогою ReentrantLock?
3. Поясніть, як повторне блокування може допомогти у створенні більш гнучкого структурованого коду блокування порівняно з використанням синхронізованих методів або операторів?
4. Коли корисно використовувати Condition.await()?
5. Наведіть методи класу CountDownLatch.
6. Що означає термін «Cyclic» в назві класу CyclicBarrier?
7. Що означає термін «Зламаний бар'єр»?
8. Наведіть методи класу CyclicBarrier.
9. Поясніть різницю між методами arriveAndAwaitAdvance() та arrive() класу Phaser()?
10. Поясніть навіщо потрібен метод register() в класі Phaser()?

Лекція 6.
Інструменти роботи в багатопотоковому середовищі Java

Засоби потокобезпечних колекцій Java Concurrent API

Concurrency API був введений в Java 5 з метою надання прикладним програмістам зручних інструментів побудови багатопотокових програм. Такі інструменти розміщені у пакеті java.util.concurrent і можуть бути поділені за функціональними ознаками на такі (Рис. 8.1):

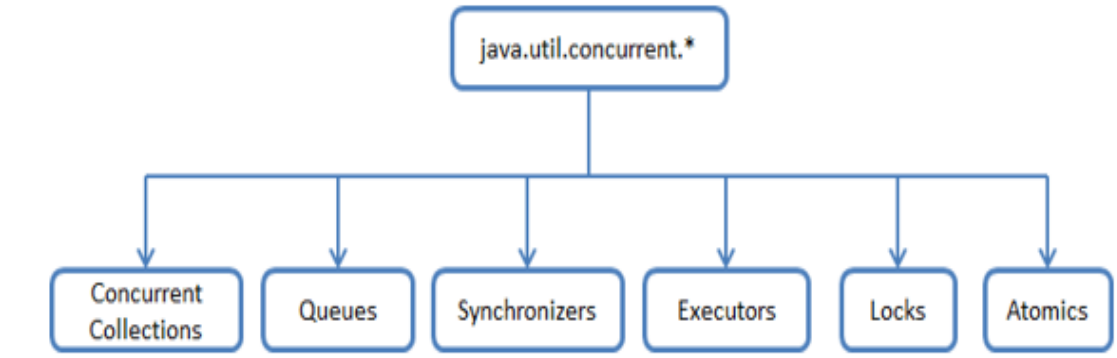


Рисунок 8.1. Інструменти Java Concurrency API

Concurrent Collections - набір колекцій, що більш ефективно працюють в багатопотоковому середовищі ніж стандартні універсальні колекції з java.util пакету. На відміну від засобів багатопотокового доступу стандартних колекцій, коли блокується доступ доступу до всієї колекції, використовуються блокування сегментів даних або ж оптимізується робота для паралельного читання даних.

Queues - блокуючі та неблокуючі черги з підтримкою багатопоточності. Блокуючі черги використовуються, коли потрібно «пригальмувати» потоки постачальника або споживача, якщо не виконані певні умови, наприклад, черга порожня або переповнена, або ж немає вільного споживача. Неблокуючі черги забезпечують швидкість роботи потоків.

Synchronizers - допоміжні утиліти для синхронізації потоків, що дозволяють розробнику управляти та/або обмежувати роботу декількох потоків.

Executors - засоби створення пулів потоків і планування роботи асинхрон- них завдань з отриманням результатів.

Locks - являє собою альтернативні і більш гнучкі механізми синхронізації потоків в порівнянні з базовими synchronized, wait, notify, notifyAll.

Atomics - класи з підтримкою атомарних операцій над примітивами і посиланнями.

Пул потоків

Створення потоку для кожного завдання вимагає значних комп'ютерних ресурсів. Певним виходом з такої ситуації є використання

пулів потоків (Thread pool). Переваги використання пулів потоків:

1. підвищення ефективності програми за рахунок повторного використання потоків (зниження витрат процесорного часу і пам'яті);
2. кращий дизайн програми, що дозволяє зосередитися на логіці роботи програми, а не на технічних засобах організації її паралельного виконання.

Основна ідея використання пулу потоків полягає в наступному (рис. 8.1): коли програмі необхідно виконати деяку задачу, замість створення потоку для виконання цього завдання і передачі йому завдання, вона просто розміщує завдання у черзі (Job queue); а один з працюючих в нескінченному циклі потоків з пулу потоків (Thread pool) вибирає завдання з черги і виконує його.

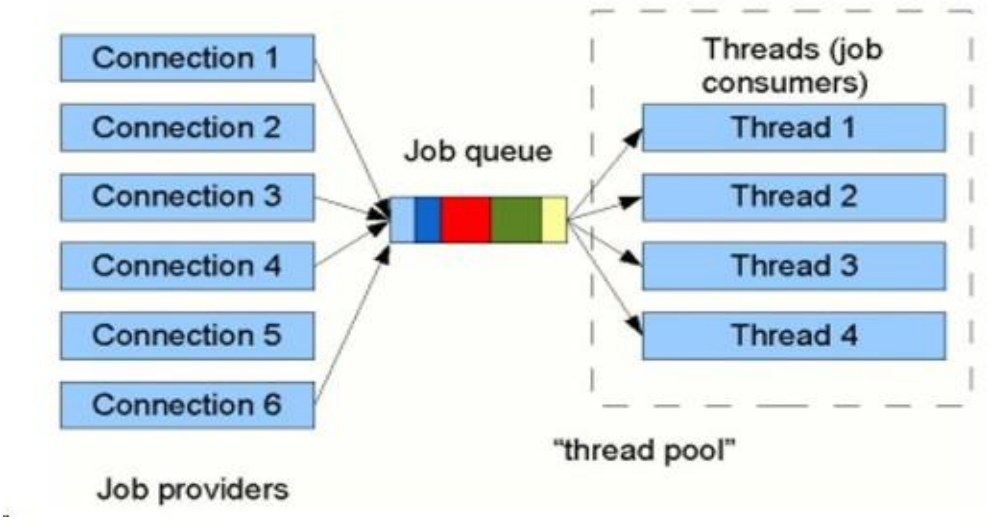


Рисунок 8.2. Використання пулу потоків

Пакет java.util.concurrent визначає три інтерфейси з функціональністю пулів потоків (Рис. 8.2):

Executor - простий інтерфейс, який підтримує запуск нових завдань.

ExecutorService - успадковує Executor, додаючи функції, що забезпечують управління життєвим циклом, як індивідуальних завдань, так і самого виконавця.

ScheduledExecutorService - успадковує ExecutorService, даний інтерфейс додає можливість запускати відкладені завдання.

Як правило, об'єкти реалізацій виконавців оголошуються як об'єкти одного з цих трьох типів інтерфейсів, а не як об'єкти класів-реалізацій.

Інтерфейс Executor містить єдиний метод void execute(Runnable r). Об'єкт Executor буде використовувати існуючий робочий потік з пулу потоків для запуску завдання r, або помістить r в чергу у разі, якщо жоден робочий потік недоступний.

Інтерфейс ExecutorService містить методи, які беруть в якості параметра як Runnable об'єкти, так і Callable об'єкти (останні дозволяють повернути з завдання результат його виконання). Найбільш часто використовуваними є методи Future<?>submit(Runnable task), <T> Future<T> submit(Runnable task, T result), <T> Future<T> submit(Callable<T> task), які повертають об'єкт Future, що використовується для отримання результату виконаного у потоці завдання або для отримання статусу завдання.

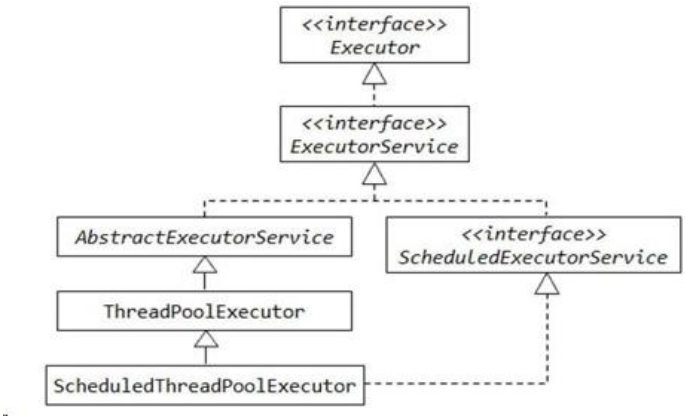


Рисунок 8.3. Ієрархія виконавців пулів потоків

Callable<V> - аналог інтерфейсу Runnable для завдань, які повертають значення, отримане при виконанні завдання у потоці (нагадаємо, що метод void run() інтерфейсу Runnable нічого не повертає). Єдиним методом Callable<V> є метод V call(), що дозволяє повертати типізоване значення. Callable- завдання, як і Runnable-завдання можуть бути передані об’єктам ExecutorService. як аргументи методу submit. Але як тоді отримати результат, який вони повертають? Оскільки метод submit не чекає завершення завдання, об’єкт ExecutorService не може повернути результат завдання безпосередньо. Замість цього він повертає спеціальний об'єкт Future, у якого можливо запросити результат виконання завдання (таке виконання називають асинхронним).

Future<V> - інтерфейс для отримання результатів виконання завдання

у потоках пулу. Основними методами є V get() та V get(long, TimeUnit), які очікують завершення завдання та повертають значення результату виконаного завдання (другий метод чекає максимум впродовж часу, вказаного як аргумент). Також існують методи для скасування виконання завдання boolean cancel(boolean mayInterruptIfRunning) і перевірки чи виконане завдання boolean isDone() та чи скасоване виконання завдання до його завершення boolean isCancelled(). Як імплементацію інтерфейсу Future<V> часто використовують клас FutureTask<V>.

Інтерфейс ExecutorService

Методи інтерфейсу ExecutorService <T> List<Future<T>> invokeAll (Collection<extends Callable<T>> tasks) та <T> List<Future<T>> invokeAll(Collection<extends Callable<T>> tasks, long timeout, TimeUnit unit) працюють зі списками завдань з блокуванням потоку до завершення всіх завдань в переданому списку або до закінчення заданого часу очікування, вони повертають список об’єктів Future з результатами та статусами відповідних завдань.

Методи <T> T invokeAny (Collection<extends Callable<T>> tasks), <T> T invokeAny (Collection<extends Callable<T>> tasks, long timeout, TimeUnit unit) блокують потік до завершення будь-якого з переданих завдань або до закінчення заданого часу очікування, вони повертають результат виконаного завдання.

Інтерфейс також містить метод void shutdown(), який завершить роботу ExecutorService після завершення усіх завдань, які виконувались у момент виклику shutdown(), та метод List<Runnable> shutdownNow(), який намагається зупинити усі завдання, що виконуються та очікують виконання та повертає список завдань, які буди у стані очікування. Після виклику методів shutdown об’єкт ExecutorService більше не братиме завдання, кидаючи RejectedExecutionException при спробі додати завдання. Також інтерфейс містить методи boolean isShutdown(), що повертає true, якщо поточний ExecutorService вимкнений, та boolean isTerminated(), що повертає true, якщо всі завдання були завершені після вимкнення ExecutorService. Метод boolean awaitTermination(long timeout, TimeUnit unit) блокує завершення ExecutorService після отримання команди shutdown(), поки всі завдання не будуть завершені або не сплине заданий як параметр час очікування або поточний потік не буде перерваний методом interrupt(), залежно від того, що трапиться раніше. Як імплементацію інтерфейсу ExecutorService часто використовують клас ThreadPoolExecutor.

ScheduledExecutorService успадковує ExecutorService та додає можливість запускати завдання через заданий як аргумент проміжок часу методами <V> ScheduledFuture <V> schedule (Callable<V> callable, long delay, TimeUnit unit) та (ScheduledFuture <> schedule (Runnable command, long delay, TimeUnit unit)). Також є методи ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit) та ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit), які виконують періодичний запуск завдань - перший після первинної затримки з фіксованим періодом, а другий - після первинної затримки з фіксованим проміжком часу між завершенням попереднього завдання та початком наступного. Для організації виконання завдання у потоці, що управляється об'єктом.

ExecutorService, необхідно:

1) реалізувати задачу в методі void run() об'єкта Runnable;

2) створити об'єкт Executor, керуючий пулом потоків, безпосередньо (в цьому випадку буде доступно більше опцій при створенні об'єкта) або з використанням одного з фабричних методів класу Executors. Пул потоків виконавця може мати один потік, фіксовану кількість потоків або необмежене число потоків (буде продемонстровано далі);

3) створити (отримати) робочий потік, виконуючий завдання шляхом виклику методу void execute(Runnable r), що додає завдання в чергу пулу потоків. Завдання буде заплановане і виконається доступним робочим потоком пулу.

Нехай є завдання вивести на екран деяку інформацію про це завдання:

```
public class MyTask implements Runnable { String taskInfo;
public MyTask(String taskInfo) { this.taskInfo = taskInfo;
}
@Override
public void run() { System.out.println(taskInfo);
}
}
```

Тоді у деякому класі:

```
public static void main(String[] args) { ThreadPoolExecutor tpe = new ThreadPoolExecutor(5, 10,
30L, TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());
MyTask[] tasks = new MyTask[25];
```

```
for (int i = 0; i < tasks.length; i++) { tasks[i] = new MyTask("Task-" + i);
tpe.execute(tasks[i]);
}
tpe.shutdown();
}
```

Параметри конструктора ThreadPoolExecutor: розмір ядра пула потоків corePoolSize = 5, максимальний розмір пула maximumPoolSize = 10. Коли нове завдання передається в метод execute, і в цей час кількість потоків в пулі менше corePoolSize, створюється новий потік для цього завдання, навіть якщо інші потоки простоюють. Якщо кількість потоків в пулі більше corePoolSize, але менше maximumPoolSize, новий потік буде створений, лише якщо черга заповнена повністю. Період життя надлишкових потоків keepAliveTime = 30 seconds - якщо в пулі є більш corePoolSize потоків, надлишкові потоки будуть знищені, якщо вони не використовується протягом проміжку часу, більш ніж KeepAliveTime. Останній параметр конструкотора - посилання на чергу для завдань workQueue (використовується колекція- черга, до якої можуть мати доступ декілька потоків LinkedBlockingQueue). Запуск програми виведе 25 рядків з інформацією про завдання, як правило, не в порядку їх запуску, оскільки планування потоків пов’язано з випадковістю:

Task-0 Task-2 Task-1 Task-7 Task-8…

Запитання для самоперевірки

1. Дайте визначення пулу потоків.
2. Які реалізації пулу потоків включені в пакет java.util.concurrent?
3. Назвіть типи черг які включають в реалізації пулу потоків?
4. Поясніть різницю між методами add(), offer(), put().
5. Поясніть, з якою метою використовують метод getDelay?
6. Поясніть різницю між методами offerFirst() та takeFirst()?
7. Поясніть різницю між методами transfer() та tryTransfer()?
8. Які переваги та недоліки має кожен з класів які реалізують пул потоків.
9. Поясніть алгоритм крадіжки роботи (work-stealing algorithm), який використовує Fork/Join фреймворк.
10. В чому різниця між абстрактними класами RecursiveTask та RecursiveAction?
11. Які статуси може мати задача в ForkJoinPool?

12. Назвіть методи класу AtomicInteger.
13. В чому полягає різниця між інтерфейсами Runnable і Callable?

Лекція 7.
Організація взаємодії потоків

Організація контролю доступу до ресурсу. Клас Семафор.

Під семафором S зазвичай розуміється змінна особливого типу, значення якої може опитуватися та змінюватися тільки за допомогою спеціальних операцій P(S) й V(S), що реалізовуються відповідно до наступних алгоритмів: операція P(S)

якщо S>0

то S = S - 1

інакше

< чекати S >

операція V(S)

якщо < один або декілька процесів чекають S >

то < зняти очікування у одного із очікуючих процесів > інакше S = S + 1

Принциповим в розумінні семафорів є те, що операції P(S) й V(S) передбачаються неподільними, що гарантує взаємовиключення при використання загальних семафорів (забезпечення неподільності операції обслуговування семафорів зазвичай реалізуються засобами операційної системи).

На відміну від об’єктів Lock, які надають потоку, в якому викликаний метод lock() цього об’єкту (або його варіації), ексклюзивний доступ до критичної секції, об'єкт класу java.util.concurrent.Semaphore підтримує певну кількість дозволів на доступ до критичної секції, яка вказується як параметр конструктора при створенні семафора.

Semaphore semaphore = new Semaphore(5);

Створювані конструктором семафори не забезпечують справедливість при наданні доступу потокам до критичної секції. Справедливе блокування (FairSync) - це коли потоки отримують блокування в тому порядку, в якому вони його запитували, несправедливе блокування (NonfairSync) може допускати "безлад" (barging), коли потік може отримати блокування раніше іншого, який запитував її першим. При передачі в конструктор другого параметра true - буде забезпечуватися справедливе блокування (семафор буде використовувати FIFO чергу).

Semaphore sem = new Semaphore(5, true);

Кожен виклик методу void acquire() з семафора в потоці зменшує на 1 кількість дозволів на доступ до критичної секції, кожен виклик методу void release() - збільшує число дозволів на 1. Таким чином, кількість потоків, що одночасно виконують критичну секцію коду, обмежується кількістю дозволів семафора, а критична секція починається викликом acquire() і закінчується викликом release(). Група методів boolean tryAcquire(), boolean tryAcquire(long timeout, TimeUnit unit), boolean tryAcquire(int permits) та tryAcquire(int permits, long timeout, TimeUnit unit), які отримують дозвіл (або permits дозволів) з семафору тільки, якщо цей дозвіл наявний в момент виклику методу (або впродовж проміжку часу, вказаного як параметр). Вони повертають true, якщо дозвіл отримано, та false - в іншому випадку. Такі методи дозволяють уникати (або зменшувати час) очікування по- токів, дозволяючи їм виконувати іншу роботу. Семафор з кількістю дозволів, що дорівнює 1, називається бінарним семафором або взаємовиключаючим семафором (Mutex-ом - Mutually Exclusive Semaphore).

Наведемо приклад використання семафору. Нехай є завдання, доступ до якого обмежується для граничної кількості потоків (вказаної в RunTime) за допомогою семафора.

```
public class Task implements Runnable {
Semaphore semaphore;
public Task(Semaphore semaphore) { this.semaphore = semaphore;
}
@Override
public void run() { boolean permit = false; try {
permit = semaphore.tryAcquire(3000,
TimeUnit.MILLISECONDS);
if (permit) {
System.out.println(Thread.currentThread(). getName() + ": Permit acquired");
sleep(5000);
} else { System.out.println(Thread.currentThread().
getName() + ": Could not acquire permit");
}
} catch (InterruptedException ex) {
} finally {
if (permit) {
```

```
semaphore.release();
System.out.println(Thread.currentThread().

getName() + ": Permit released");}}}}
```

Тоді у деякому класі:

```
public static void main(String[] args)
throws InterruptedException {
ExecutorService executor =
Executors.newFixedThreadPool(10); Semaphore semaphore = new Semaphore(3);
Task task = new Task(semaphore); for (int i = 0; i < 10; i++) {
executor.submit(task);
/*Потрібно підбирати час, щоб побачити роботу семафору*/
Thread.sleep(500);
}
executor.shutdown();
}
```

Запуск програми демонструє роботу семафору:

pool-1-thread-1: Permit acquired pool-1-thread-2: Permit acquired pool-1-thread-3: Permit acquired

pool-1-thread-4: Could not acquire permit pool-1-thread-1: Permit released

pool-1-thread-5: Permit acquired pool-1-thread-2: Permit released pool-1-thread-6: Permit acquired pool-1-thread-3: Permit released pool-1-thread-7: Permit acquired

pool-1-thread-8: Could not acquire permit pool-1-thread-9: Could not acquire permit pool-1-thread-10: Could not acquire permit pool-1-thread-5: Permit released

pool-1-thread-6: Permit released pool-1-thread-7: Permit released

Бачимо, що після отримання трьох дозволів потоками 1, 2 та 3, потік 4 не

дочекався вивільнення дозволу, а потоки 5, 6 та 7 - дочекались і отримали його. Інші потоки (8, 9 та 10) не дочекались дозволу.

Циклічні бар'єри

Об’єкти циклічного бар'єру (CyclicBarrier) є точкою синхронізації, в якій вказана кількість паралельних потоків зустрічається і блокується. Як тільки всі потоки прибули, виконується опціонна операція (або не виконується, якщо бар'єр був ініціалізований без неї), і, після виконання

операції, бар'єр ламається і потоки, що очікують, «звільняються». В конструктор бар'єру (CyclicBarrier(int parties) і CyclicBarrier(int parties, Runnable barrierAction)) передається кількість сторін, які повинні «зустрітися», і, опціонально, дія, яка має відбутися, коли сторони зустрілися, але перед тим коли вони будуть «відпущені». Метод int await() об’єкту CyclicBarrier вказує потоку, з якого був викликаний цей метод, що він «підійшов» до бар’єра. Цей потік переводиться в стан очікування до моменту, коли всі інші потоки, кількість яких вказана у конструкторі як параметр parties досягнуть бар’єру (в них буде викликаний метод int await() об’єкту CyclicBarrier). Існує метод int await(long timeout, TimeUnit unit) об’єкту CyclicBarrier, який переводить поточний потік в стан очікування доки всі інші потоки не досягнуть бар’єру або не спливе проміжок часу, вказаний як параметр.

Розглянемо роботу об’єкта CyclicBarrier у програмі, яка емулює паромну переправу. Паром може переправляти одночасно по три автомобіля. Щоб не ганяти паром зайвий раз, потрібно відправляти його, коли біля переправи збереться мінімум три автомобілі. Є клас-завдання Car, що описує автомобіль та клас- завдання FerryBoat, що описує паром:

```
public class Car implements Runnable { private int carNumber;
public Car(int carNumber) { this.carNumber = carNumber;
}
@Override
public void run() { try {
System.out.printf("Car №%d drove up
to the ferry.\n", carNumber);
Ferry.BARRIER.await();
System.out.printf("Car №%d continued to move.\n", carNumber);
} catch (Exception e) {
}
}
}

public class FerryBoat implements Runnable {
@Override
public void run() { try {
Thread.sleep(500); System.out.println("FerryBoat ferrying cars!");
} catch (InterruptedException e) {
```

```
}
}
}
```

Клас Car містить виклик методу await() з об’єкту CyclicBarrier, який створюється в RunTime-класі Ferry. В його конструкторі параметрами вказуються 3 потоки (автомобілі), які будуть очікувати зняття бар’єру, та дія - створення об’єкту FerryBoat (парому), який буде перевозити автомобілі:

```
public class Ferry {
static final CyclicBarrier BARRIER =
new CyclicBarrier(3, new FerryBoat());
public static void main(String[] args)
throws InterruptedException { for (int i = 0; i < 9; i++) {
new Thread(new Car(i)).start(); Thread.sleep(400);} }}
```

Запуск програми виводить таку інформацію:

Car №0 drove up to the ferry. Car №1 drove up to the ferry. Car №2 drove up to the ferry. Car №3 drove up to the ferry. FerryBoat ferrying cars!

Car №2 continued to move. Car №1 continued to move. Car №0 continued to move. Car №4 drove up to the ferry. Car №5 drove up to the ferry. Car №6 drove up to the ferry. FerryBoat ferrying cars!

Car №3 continued to move. Car №5 continued to move. Car №4 continued to move. Car №7 drove up to the ferry. Car №8 drove up to the ferry. FerryBoat ferrying cars!

Car №8 continued to move. Car №7 continued to move. Car №6 continued to move.

Бачимо що паром перевозить по 3 автомобілі, а інші чекають. CyclicBarriers корисні в програмах, що включають фіксовану групу потоків, які час від часу повинні чекати один одного. Бар'єр називається циклічним, оскільки його можна використовувати повторно після випуску потоків, що очікували.

Запитання для самоперевірки

1. Які основні методи в класі Semaphore у Java?
2. Яка різниця між бінарним семафором та об'єктом потокового блокування?
3. Як використовується семафор для управління доступом до обмежених ресурсів?
4. Чому семафори є важливими в оперативних системах?

5. Які проблеми у багатопотоковому програмуванні може вирішити використання семафорів?
6. Яка різниця між семафорами та м'ютексами?
7. Як працює метод acquire() в семафорах?
8. Як працює метод release() в семафорах?
9. Які проблеми можуть виникнути при використанні семафорів, і як їх уникнути?

Лекція 8.
Монітори

Семафори є фундаментальним механізмом синхронізації. Їх використання полегшує програмування взаємного виключення і сигналізації. Однак семафори низькорівневий механізм; користуючись ними, легко наробити помилок. Програміст повинен стежити за тим, щоб випадково не пропустити виклики операцій P і V або задати їх більше, ніж потрібно. Семафори глобальні по відношенню до всіх процесів, тому, щоб розібратися, як використовується семафор або інша колективна змінна, необхідно переглянути всю програму. Нарешті, при використанні семафорів взаємне виключення і умовна синхронізація програмуються однієї і тієї ж парою примітивів. Через це важко зрозуміти, для чого призначені конкретні P і V, не подивившись на інші операції з даними семафорами. Взаємне виключення і умовна синхронізація це різні поняття, тому і програмувати їх краще різними способами.

Концепція монітора була розроблена Ч. Хоаром.

Монітори це програмні модулі, які забезпечують більшу структурованість, ніж семафори, хоча реалізуються так само ефективно. В першу чергу, монітори є механізмом абстракції даних. Монітор інкапсулює поняття абстрактного об’єкта і забезпечує набір операцій, тільки за допомогою яких він обробляється. Монітор містить змінні, що зберігають стан об’єкта, і процедури, що реалізують операції над ним. Процес отримує доступ до змінних в моніторі тільки шляхом виклику процедур цього монітора. Взаємне виключення забезпечується неявно тим, що процедури в одному моніторі не можуть виконуватися паралельно. Умовна синхронізація в моніторах забезпечується явно за допомогою умовних змінних (condition variables). Вони аналогічні семафорам, але мають істотні відмінності у визначенні та, отже, в використанні для сигналізації.

Паралельна програма, яка використовує монітори для взаємодії і синхронізації, містить два типи модулів: активні процеси і пасивні монітори. За умови, що всі колективні змінні знаходяться всередині моніторів, два процеси взаємодіють, викликаючи процедури одного і того ж монітора. Отримана модульність має два важливих переваги.

Перше — процес, що викликає процедуру монітора, може не знати про конкретну реалізації процедури; роль грають лише видимі результати виклику процедури.

Друге — програміст монітора може не піклуватися про те, де і як використовуються процедури монітора, і вільно змінювати його реалізацію. Ці переваги дозволяють розробляти процеси і монітори відносно незалежно, що полегшує створення та розуміння паралельної програми.

Синтаксис та семантика

Монітор використовується, щоб згрупувати зображення і реалізацію спільного ресурсу (класу). Він складається з інтерфейсу та тіла. Інтерфейс визначає надані ресурсом операції (методи). Тіло містить змінні, що описують стан ресурсу, і процедури, що реалізують операції інтерфейсу.

Надалі будемо вважати, що монітор є статичним об’єктом, а його тіло і інтерфейс описані так:

monitor mname {

оголошення постійних змінних оператори ініціалізації процедури

}

Процедури реалізують видимі операції. Постійні змінні (permanent variables) поділяються всіма процедурами тіла монітора. Вони називаються постійними, оскільки існують і зберігають своє значення, поки існує монітор. У процедурах можна використовувати локальні змінні, копії яких створюються для кожного виклику функції.

Монітор як представник абстрактних типів даних має три властивості.

- По - перше, поза монітором видно тільки імена процедур. Щоб змінити стан ресурсу, процес повинен викликати одну з процедур монітора. Виклик процедури монітора має такий вигляд.

mname.opname(arguments)

Тут mname — ім’я монітора, opname — ім’я однієї з його операцій (процедур), що викликається з аргументами arguments.

- По-друге, оператори всередині монітора (в оголошеннях і процедурах) не можуть звертатися до змінних, оголошених поза монітором.
- По-третє, постійні змінні ініціалізуються до виклику його

процедур.

Взаємне виключення

Синхронізацію найпростіше зрозуміти і запрограмувати, якщо взаємне виключення і умовна синхронізація виконуються різними способами. Найкраще, якщо взаємне виключення відбувається неявно, чим автоматично усувається взаємний вплив. Крім того, програми легше читати, оскільки в них немає явних протоколів входу в критичні секції і виходу з них.

На відміну від взаємного виключення, умовну синхронізацію потрібно програмувати явно, оскільки різні програми вимагають різних умов синхронізації. Хоча часто простіше синхронізувати за допомогою логічних умов, як в операторах await, низькорівневі механізми можна реалізувати набагато ефективніше. Відповідно до цих зауваженнями взаємне виключення в моніторах забезпечується неявно, а умовна синхронізація програмується за допомогою так званих умовних змінних.

Зовнішній процес викликає процедуру монітора. Поки певний процес виконує оператори процедури, вона активна. У будь-який момент часу може бути активним тільки один екземпляр тільки однієї процедури монітора, тобто одночасно не можуть бути активними ні два виклики різних процедур, ні два виклики однієї і тієї ж процедури.

Процедури моніторів за визначенням виконуються із взаємним виключенням. Воно забезпечується реалізацією мови, бібліотекою або операційною системою, але не програмістом, який використовує монітори. На практиці взаємне виключення в мовах і бібліотеках реалізується за допомогою блокування та семафорів.

Умовні змінні

Умовна змінна використовується для припинення роботи процесу, безпечне виконання якого неможливо до переходу монітора в стан, що задовольняє деяку логічну умову. Умовні змінні також застосовуються для запуску призупинених процесів, коли певна логічна умова стає істинною. Умовна змінна оголошується наступним чином:

cond cv;

Таким чином, cond — це новий тип даних. Умовні змінні можна оголошувати і використовувати тільки в межах моніторів. Значенням умовної змінної cv є черга призупинених процесів (черга затримки). Спочатку вона порожня. Програміст не може безпосередньо звертатися до значення змінної cv. Замість цього він отримує непрямий доступ до черги за

допомогою декількох спеціальних операцій, описаних нижче. Процес може запросити стан умовної змінної за допомогою виклику empty(cv)

Якщо черга змінної cv порожня, ця функція повертає значення true, інакше false.

Процес блокується на умовній змінній cv за допомогою виклику wait(cv)

Операція wait змушує працюючий процес затриматися в кінці черги змінної cv. Щоб інший процес міг врешті-решт увійти в монітор для запуску призупиненого процесу, виконання операції wait відбирає у процесу, що викликав її, винятковий доступ до монітора. Процеси, заблоковані на умовних змінних, запускаються операторами signal. При виконанні виклику signal(cv) перевіряється черга затримки змінної cv. Якщо вона порожня, ніякі дії не відбуваються. Однак, якщо призупинені процеси є, оператор signal запускає процес у початку черги. Таким чином, операції wait та signal забезпечують порядок сигналізації FIFO: процеси припиняються в порядку викликів операції wait, а запускаються в порядку викликів операції signal.

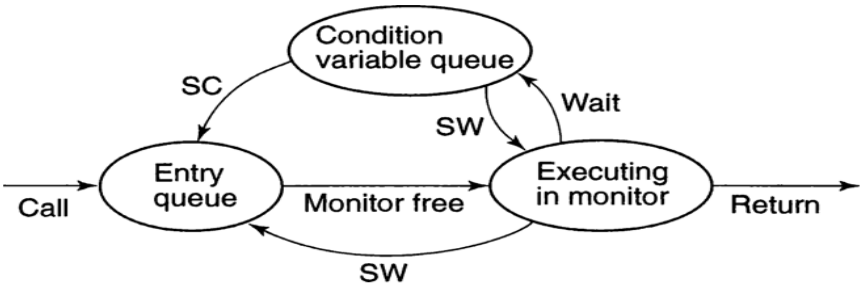


Рисунок 10.1. Діаграма станів

Виконуючи оператор signal, процес працює в моніторі і, отже, може керувати блокуванням, неявно пов'язаним із монітором. В результаті виникає дилема. Якщо операція signal запускає інший процес, то виходить, що могли б виконуватися два процеси: той, що викликав операцію signal та запущений нею. Але наступним може виконуватися тільки один з них, оскільки лише один процес може мати винятковий доступ до монітора. Таким чином, можливі два варіанти:

- сигналізувати і продовжити: сигналізатор продовжує роботу, а процес, що отримав сигнал, виконується пізніше;
- сигналізувати і очікувати: сигналізатор чекає деякий час, а процес, який отримав сигнал, виконується відразу.

Дисципліна (порядок) «сигналізувати і продовжити» не переривається обслуговування. Процес, що виконує операцію signal, зберігає винятковий доступ до монітора, а процес, що запускається, почне роботу трохи пізніше, коли отримає винятковий доступ до монітора. По суті, операція signal просто вказує процесу, який запускається, на можливість виконання, після чого він повертається в чергу процесів, що очікують.

Порядок «сигналізувати і очікувати» має властивість переривання обслуговування. Процес, що виконує операцію signal, передає управління блокуванням монітора процесу, який запускається, тобто переривається робота процесу - сигналізатора. В цьому випадку сигналізатор переходить в чергу процесів, які очікують на звільнення монітора.

Діаграма станів

Діаграма станів на рис. 10.1 ілюструє роботу синхронізації в моніторах. Викликаючи процедуру монітора, процес поміщається у вхідну чергу, якщо в моніторі виконується ще один процес; в іншому випадку процес, який викликав операцію, негайно починає виконання в моніторі. Коли монітор звільняється (після повернення з процедури або виконання операції wait), один процес з вхідної черги може перейти до роботи в моніторі. Виконуючи операцію wait(cv), процес переходить від роботи в моніторі в чергу, пов’язану з умовною змінною. Якщо процес виконує операцію signal(cv), то при порядку

«Сигналізувати і продовжити» (Signal and Continue — SC) процес з початку черги умовної змінної переходить до вхідної. При порядку «сигналізувати і очікувати» (Signal and Wait — SW) процес, що виконується в моніторі, переходить до вхідної черги, а процес з початку черги умовної змінної переходить до виконання в моніторі.

Запитання для самоперевірки

1. Яка різниця між моніторами та семафорами?
2. Як використовується монітор для синхронізації декількох потоків у багатопотоковому програмуванні?
3. Як працює система умовного контролю в моніторах?
4. Що таке пробудження потоку в контексті моніторів?
5. Як монітори допомагають в управлінні взаємовиключенням?
6. Як монітор може допомогти в управлінні взаємодією потоків?
7. Що включає в себе внутрішній механізм моніторів для управління потоками?

8. Які переваги використання моніторів порівняно з іншими механізмами синхронізації, такими як замки (locks) або семафори?
9. Чи є у моніторів якісь недоліки або обмеження в контексті багатопотокового програмування?

Лекція 9.
Сучасні інтерфейси для паралельного програмування

Засоби паралельного програмування застосовуються для різної архітектури мультипроцесорних систем та різних процесорних платформ з використанням різних мов програмування. При такій великій різноманітності з метою уніфікації й можливості перенесення паралельного програмного забезпечення йдуть двома основними шляхами:

1) Розширення існуючих мов програмування і створення нових компіляторів для них (наприклад, від C до C++).

2) Створення бібліотек функцій та програмних оболонок, які дають можливість, не змінюючи компілятора з базової мови програмування, забезпечувати стандартний інтерфейс програмування.

Другий підхід має перевагу, оскільки робить можливим використання існуючого програмного забезпечення. Проте обидва підходи не виключають, а доповнюють один одного. Прикладом першого з них являється поява нової мови програмування Java, яка базується на сучасних концепціях об'єктного орієнтування, багатоплатоформленій реалізації й підтримки паралелізму. Прикладами другого підходу є сучасні інтерфейси паралельного програмування, які є надбудовами над стандартними компіляторами та бібліотеки функцій, за допомогою яких паралельні алгоритми можуть програмуватися єдиним способом на багатьох мультипроцесорних системах. Матеріал цієї лекції присвячений останньому з названих підходів.

Існує декілька відомих бібліотек інтерфейсних функцій для паралельного програмування. Найвідомішими з них є PVM (Parallel Virtual Machine) та MPI (Message Passing Interface).

Parallel Virtual Machine

PVM - розробка кінця 80-х - початку 90-х рр., місце розробки - США. Головною метою було забезпечення можливості паралельного програмування на різнорідних (гетерогенних) мережах комп'ютерів, сполучених протоколом TCP/IP. PVM - це пакет програм, написаних мовою

програмування С, що дає можливість розглядати гетерогенну мережу Unix-машин як єдиний паралельний комп'ютер з розподіленою пам'яттю (віртуальну машину). До складу віртуальної машини можуть входити як послідовні, так і паралельні комп'ютери.

Основною одиницею роботи в PVM є завдання (task), яке, як правило, відповідає поняттю процесу в Unix. Завдання можуть обмінюватися повідомленнями та утворювати таким чином застосування (application), яке може бути написане мовою С.

PVM складається з двох частин: клієнтської та серверної. Серверна частина є множиною образів (демонів) програм, розподілених між комп'ютерами віртуальної машини. Клієнтська частина утворена бібліотечними функціями PVM, які знаходяться у розпорядженні користувача та призначені для підтримки обміну повідомленнями, координації її завдань й зміни конфігурації віртуальної машини. Щоб скористатися можливостями PVM, програма застосування має бути пов'язана з бібліотекою PVM.

На одному і тому ж устаткуванні мережі може існувати декілька віртуальних машин, які одночасно здатні виконувати декілька застосувань.

Для виконання застосування користувач спочатку породжує демон на своєму комп'ютері, після чого застосування може стартувати з будь-якого комп'ютера віртуальної машини. У разі виходу з ладу одного з комп'ютерів PVM автоматично реєструє цю подію і виводить цей комп'ютер із складу віртуальної машини. Застосування може передбачити заміну машини користувача резервною.

Модель взаємодії завдань у PVM не накладає обмежень ні на кількість, ні на об'єм повідомлень, якими обмінюються завдання. Обмін повідомленнями може відбутися між будь-якими завданнями у віртуальному комп'ютері. Передбачені такі режими обміну:

- блокована передача - виконується відразу після отримання буфера для передачі;
- асинхронна блокована передача - не залежить від виклику відповідного прийому повідомлення;
- неблокований прийом - завершується відразу з альтернативним результатом: або прийняте повідомлення, або ознака того, що повідомлення ще не поступило.

Окрім цих парних взаємодій можливі також групові операції:

«група – одному» (multicast), «один – групі» (broadcast).

Модель обміну повідомленнями PVM передбачає також збереження порядку проходження повідомлень. Тобто якщо завдання 1 передало повідомлення A і B завданню 2 в порядку їх переліку, то в тому ж порядку вони будуть прийняті завданням 2.

Отже, особливостями PVM є:

- підтримка гетерогенних мереж;
- динамічне управління ресурсами віртуальної машини.

Ці особливості є перевагами PVM, проте вони тягнуть і головний недолік PVM - порівняно невисоку ефективність паралельних обчислень через великі накладні витрати на підтримку своїх переваг.

Message Passing Interface

Однією із найпоширеніших сучасних технологій організації паралельних обчислень є MPI. MPI (Message Passing Interface) — інтерфейс обміну повідомленнями (інформацією) між одночасно працюючими обчислювальними процесами. Він широко використовується для створення паралельних програм для обчислювальних систем з розподіленою пам'яттю (кластерів).

Так само, як і в PVM, в MPI реалізується підхід розширення відомих мов за рахунок бібліотеки функцій, які забезпечують засоби взаємодії паралельних процесів. Проте метою MPI є забезпечення високої продуктивності і можливості перенесення застосувань на різні мультипроцесорні платформи. На теперішній час MPI є машинно-незалежним стандартом для паралельного програмування, який включає понад 100 бібліотек функцій [16].

На відміну від PVM, яку можна розглядати як розподілену операційну систему для гетерогенних мереж, MPI є тільки інтерфейсом, розрахованим на однорідні мультипроцесорні системи.

Однією із найпоширеніших сучасних технологій організації паралельних обчислень є MPI. MPI (Message Passing Interface) — інтерфейс обміну повідомленнями (інформацією) між одночасно працюючими обчислювальними процесами. Він широко використовується для створення паралельних програм для обчислювальних систем з розподіленою пам'яттю (кластерів).

MPICH — найвідоміша реалізація MPI, створена в Арагонській національній лабораторії (США). Існують версії цієї бібліотеки для усіх популярних операційних систем. До того ж, вона безкоштовна. Перераховані чинники роблять MPICH ідеальним варіантом для того, щоб

почати практичне освоєння MPI.

MPICH2 - це не версія програмного забезпечення, а номер того стандарту MPI, який реалізований у бібліотеці. MPICH2 відповідає стандарту MPI 2.0, звідси і назва. Мета створення MPICH2 наступна:

1) Надати реалізацію MPI, яка ефективно підтримує різні обчислювальні і комунікаційні платформи, включаючи загальнодоступні кластери (настільні системи, системи із загальною пам'яттю, багатоядерна архітектура), високошвидкісні мережі (Ethernet 10 ГБіт/с, InfiniBand, Myrinet, Quadrics) та ексклюзивні обчислювальні системи (Blue Gene, Cray, SiCortex).

2) Зробити можливими передові дослідження технології MPI за допомогою легко розширюваної модульної структури для створення похідних реалізацій.

В якості кластера можна використовувати комп'ютерну мережу навчального класу. За відсутності комп'ютерної мережі можна запускати усі обчислювальні процеси і на одному комп'ютері. Якщо комп'ютер одноядерний, то, природно, приросту швидкодії ви не отримаєте, - тільки уповільнення. Якщо багатоядерний - то можна завантажити всі обчислювальні ядра.

В якості середовища розробки доцільно використовувати Visual Studio 2008 або 2010 (можливе використання безкоштовної версії Express).

MPICH для Windows складається з наступних компонентів:

1) Менеджер процесів smpd.exe, який є системною службою (сервісне застосування). Менеджер процесів веде список обчислювальних вузлів системи, і запускає на цих вузлах MPI-програми, надаючи їм необхідну інформацію для роботи і обміну повідомленнями.

2) Заголовні файли (.h) і бібліотеки компіляції (.lib), необхідні для розробки MPI -програм.

3) Бібліотеки виконання (.dll), необхідні для роботи MPI -програм.

4) Додаткові утиліти (.exe), необхідні для налаштування MPICH і запуску MPI -програм.

Усі компоненти, окрім бібліотек виконання, встановлюються по замовчуванню в теку C:\Program Files\MPICH2; dll-бібліотеки встановлюються в C:\Windows\System32.

Менеджер процесів є основним компонентом, який має бути встановлений і налагоджений на усіх комп'ютерах мережі (бібліотеки часу виконання можна, в крайньому випадку, копіювати разом з MPI

програмою). Інші файли потрібно для розробки MPI-програм і налаштування деякого «головного» комп'ютера, з якого здійснюватиметься їх запуск.

Менеджер працює у фоновому режимі і чекає запитів до нього з мережі з боку «головного» менеджера процесів (за умовчанням використовується мережевий порт 8676). Щоб захистити себе від хакерів і вірусів, менеджер вимагає пароль при зверненні до нього. Коли один менеджер процесів звертається до іншого менеджера процесів, він передає йому свій пароль. Звідси витікає, що треба вказувати один і той же пароль при установці MPICH на комп'ютери мережі.

Запуск MPI-програми робиться таким чином:

Користувач за допомогою програми Mpirun (чи Mpiexec, при використанні MPICH2 під Windows) вказує ім'я виконуваного файлу MPI - програми і необхідне число процесів. Крім того, можна вказати ім'я користувача і пароль: процеси MPI-програми запускатимуться від імені цього користувача.

Mpirun передає відомості про запуск локальному менеджерові процесів, у якого є список доступних обчислювальних вузлів.

Менеджер процесів звертається до обчислювальних вузлів за списком, передаючи запущеним на них менеджерам процесів вказівки по запуску MPI - програми.

Менеджери процесів запускають на обчислювальних вузлах декілька копій MPI-програми (можливо, по декілька копій на кожному вузлі), передаючи програмам необхідну інформацію для зв'язку один з одним.

Дуже важливим моментом є те, що перед запуском MPI -програма не копіюється автоматично на обчислювальні вузли кластера. Замість цього менеджер процесів передає вузлам шлях до виконуваного файлу програми точно в тому вигляді, в якому користувач вказав цей шлях програмі Mpirnn. Це означає, що якщо ви, наприклад, запускаєте програму C :.exe, то усі менеджери процесів на обчислювальних вузлах намагатимуться запустити файл C :.exe. Якщо хоч би на одному з вузлів такого файлу не виявиться, станеться помилка запуску MPI -програми.

Щоб кожного разу не копіювати вручну програму і усі необхідні для її роботи файли на обчислювальні вузли кластера, зазвичай використовують загальний мережевий ресурс. В цьому випадку користувач копіює програму і додаткові файли на мережевий ресурс, видимий усіма вузлами кластера, і вказує шлях до файлу програми на цьому ресурсі. Додатковою зручністю

такого підходу є те, що за наявності можливості запису на загальний мережевий ресурс запущені копії програми можуть записувати туди результати своєї роботи.

Робота MPI програми відбувається таким чином:

1) Програма запускається та ініціалізує бібліотеку часу виконання MPICH шляхом виклику функції MPI_Init.

2) Бібліотека отримує від менеджера процесів інформацію про кількість і місце розташування інших процесів програми, і встановлює з ними зв'язок.

3) Після цього запущені копії програми можуть обмінюватися один з одним інформацією за допомогою бібліотеки MPICH. З точки зору операційної системи бібліотека є частиною програми (працює в тому ж процесі), тому можна вважати, що запущені копії MPI-програми обмінюються даними безпосередньо один з одним, як будь-які інші застосування, передавальні дані по мережі.

4) Консольне уведення-виведення усіх процесів MPI-програми перенаправляється на консоль, на якій запущена Mpirun. Перенаправленням введення-виведення займаються менеджери процесів, оскільки саме вони запустили копії MPI-програми, і тому можуть отримати доступ до потоків введення-виведення програм.

5) Перед завершенням усі процеси викликають функцію MPI_Finalize, яка коректно завершує передачу і прийом усіх повідомлень, і відключає MPICH.

OpenMP

OpenMP задуманий як стандарт для програмування на масштабованих SMP-системах в моделі загальної пам'яті (shared memory model). У стандарт OpenMP входять специфікації набору директив компілятора, процедур і змінних середовища. Розробкою стандарту займається організація OpenMP ARB (Architecture Board), до якої увійшли представники найбільших компаній – розробників SMP-архітектур і програмного забезпечення..

До появи OpenMP не було відповідного стандарту для ефективного програмування на SMP-системах.

POSIX-інтерфейс для організації ниток(Pthreads) підтримується широко (практично на всіх UNIX-системах), однак з багатьох причин не підходить для практичного паралельного програмування: немає підтримки Fortrana, занадто низький рівень, немає підтримки паралелізму за даними, механізм ниток спочатку розроблявся не для цілей організації паралелізму.

OpenMP можна розглядати як високорівневу надбудову над Pthreads (чи аналогічними бібліотеками). Багато постачальників SMP-архітектур (Sun, HP, SGI) у своїх компіляторах підтримують спеціальні директиви для розпаралелювання циклів. Однак ці набори директив, як правило дуже обмежені, несумісні між собою; в результаті чого розробникам доводиться розпаралелювати застосування окремо для кожної платформи. OpenMP є багато в чому узагальненням і розширенням згаданих наборів директив. OpenMP надає розробнику наступні переваги:

1) За рахунок ідеї «інкрементального розпаралелювання» OpenMP ідеально підходить для розробників, що бажають швидко розпаралелити свої обчислювальні програми з великими паралельними циклами. Розробник не створює нову паралельну програму, а просто послідовно додає в текст послідовної програми OpenMP-директиви.

2) При цьому OpenMP – досить гнучкий механізм, що надає розробнику більші можливості контролю над поведінкою паралельного застосування.

3) Передбачається, що OpenMP-програма на багатопроцесорній платформі може бути використана в якості послідовної програми. Директиви OpenMP просто ігноруються послідовним компілятором, а для виклику процедур OpenMP можуть бути підставлені заглушки (stubs).

4) Однією з переваг OpenMP його розробники вважають підтримку так званих «orphan» (відірваних) директив, тобто директиви синхронізації і розподілу роботи можуть не входити безпосередньо в лексичний контекст паралельної області.

На даний момент технологія OpenMP підтримується більшістю компіляторів мови C/C++. Дещо гірше справа йде з інструментами тестування паралельних OpenMP програм. Інструменти аналізу, перевірки та оптимізації паралельних програм хоча й існують давно, до недавнього часу були мало задіяні при розробці прикладного програмного забезпечення. Тому вони часто є менш зручними, ніж інші інструментальні засоби розробки. Найбільш повно процес розробки паралельних OpenMP програм підтриманий у пакеті Intel Parallel Studio. Є інструмент попереднього аналізу коду, для виявлення ділянок коду, які потенційно можна ефективно розпаралелити. Є добре оптимізований компілятор з підтримкою OpenMP. Додатково можна виділити інструмент VivaMP, що входить до складу PVS-Studio. Це статичний аналізатор коду, спеціалізований на виявленні помилок у OpenMP програмах на етапі їх

написання.

Робота OpenMP-застосування починається з єдиного потоку – основного. У застосуванні можуть міститися паралельні регіони, входячи в які, основний потік створює групи потоків (що включають основний потік). Наприкінці паралельного регіону групи потоків зупиняються, а виконання основного потоку триває. У паралельний регіон можуть бути вкладені інші паралельні регіони, в яких кожен потік первісного регіону стає основним для своєї групи потоків. Вкладені регіони можуть у свою чергу включати регіони більш глибокого рівня вкладеності.

Запитання для самоперевірки

1. Які є основні функції інтерфейсу обміну повідомленнями (MPI)?
2. Як PVM або MPI використовуються для реалізації паралельних обчислень?
3. Які типи проблем можуть бути вирішені за допомогою PVM або MPI?
4. Які особливості працюють з MPI для обробки одночасного обміну повідомленнями?
5. Як PVM та MPI працюють із розподіленою пам'яттю та процесами?
6. Що таке колективні операції в MPI?
7. Як відбувається обробка помилок в PVM та MPI?
8. Як можна оптимізувати продуктивність використовуючи MPI?
9. В чому полягає різниця між блокуючими та неблокуючими операціями в MPI?
10. Які варіанти існують для персоналізації процесу розсилання повідомлень в PVM або MPI?

Список літератури

1. Лісовенко І.Д., Яковлєва І. Д. Навчальний посібник «Паралельні та розподілені обчислення». Чернівці: ЧНУ, 2022. 120 с.
2. Минайленко Р.М. Паралельні та розподілені обчислення : навч. посіб. — Кропивницький: Видавець Лисенко В. Ф., 2021. 153 с.
3. Коцовський В. М. Теорія паралельних обчислень: навчальний посібник. /В. М. Коцовський - Ужгород: ПП «АУТДОР-Шарк», 2021. - 188 с.
4. Малашонок Г. І., Сідько А. А. Паралельні обчислення на розподіленій пам’яті: OpenMPI, Java, Math Partner : підручник. / Г. І. Малашонок, А. А. Сідько. – Київ : НаУКМА, 2020. – 266 с.
5. Корочкін О.В. Паралельні та розподілені обчислення. Вибрані розділи: Навч. посібник. / О.В. Корочкін, О.В. Русанова.– Київ : КПІ ім. Ігоря Сікорського, 2020. – 123 с.

Інформаційні ресурси в Інтернет

1. Java специфікація класу Циклічний бар’єр. Режим доступу CyclicBarrier (Java Platform SE 8) (oracle.com)
2. Java специфікація класу Семафор. Режим доступу Semaphore (Java Platform SE 8) (oracle.com)
3. Java специфікація станів потоку. Режим доступу Thread.State (Java Platform SE 8) (oracle.com)
4. Java специфікація інтерфейсу Lock. Режим доступу Lock (Java SE 11 & JDK 11) (oracle.com)
5. Java специфікація інтерфейсу Executor. Режим доступу Executor (Java Platform SE 8) (oracle.com)
6. Java специфікація інтерфейсу Executor. Режим доступу ExecutorService (Java Platform SE 8) (oracle.com)

Навчальне видання

О. Л. Соловей

ТЕХНОЛОГІЯ РОЗПОДІЛЕНИХ СИСТЕМ ТА ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

Конспект лекцій
для здобувачів першого (бакалаврського) рівня вищої освіти
за спеціальністю 122 Комп’ютерні науки

Редагування та коректура В.С. Ясінської
Комп’ютерне верстання А.П. Морозюк

Підписано до друку 23.09.2016 Формат 60 □ 84 1/ 16
Ум. друк. арк. 6,74 Обл.-вид. арк. 7.25
Електронний документ. Вид. No 10/І-16 Зам. 40/1-16

Видавець і виготовлювач
Київський національний університет будівництва і архітектури

Повітрофлотський проспект, 31, Київ, Україна, 03680

Свідоцтво про внесення до Державного реєстру суб’єктів
видавничої справи ДК No 808 від 13.02.2002 р.