МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

О.В.Корочкін, Ю.О.Кулаков, О.В. Русанова

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ВИСОКОПРОДУКТИВНИХ КОМП'ЮТЕРНИХ СИСТЕМ.
Конспект лекцій

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського як навчальний посібник для здобувачів ступеня бакалавра за освітньою програмою «Програмне забезпечення комп'ютерних систем» спеціальності 121 «Інженерія програмного забезпечення»,

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2024

УДК 681.3
Програмне забезпечення високопродуктивних комп'ютерних систем. Конспект лекцій [Електронний ресурс] : навч. посібн. для здобувачів ступеня бакалавра за освітньою програмою «Програмне забезпечення комп'ютерних систем» спеціальності 121 «Інженерія програмного забезпечення» / укладачі: О.В.Корочкін, Ю.О.Кулаков, О.В. Русанова; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 2,93 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2024. – 257 с.
Рецензент: Тарасенко-Клятченко О.В., к.т.н., доцент кафедри системного програмування та спеціалізованих комп'ютерних систем ФПМ КПІ ім. Ігоря Сікорського
Відповідальний редактор Писарчук О, д-р. техн. наук, проф.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол № 5 від 29.02.2024 р.) за поданням Вченої ради факультету інформатики та обчислювальної техніки (протокол № 8. від 19.02.2024 р.)

Навчальний посібник призначено для вивчення методів та засобів розробки програмного забезпечення сучасних комп'ютерних систем та використання при проведенні лекційних занять і самостійної роботи з дисципліни «Програмне забезпечення високопродуктивних комп'ютерних систем» при підготовки студентів за освітньою програмою «Програмне забезпечення комп'ютерних систем» спеціальності 121 «Інженерія програмного забезпечення». Розглядаються теоретичні та прикладні особливості розробки програмного забезпечення для комп'ютерних систем з використанням потоків в сучасних мовах і бібліотеках паралельного програмування: Java, C#, Ada, WinAPI, OpenMP, MPI.

Реєстр. № НП XX/XX-XXX. Обсяг 5,2 авт. арк.
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Перемоги, 37, м. Київ, 03056 https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

З М І С Т

В С Т У П

Дисципліна «Програмне забезпечення високопродуктивних комп'ютерних систем» є нормативною дисципліною підготовки бакалаврів за освітньою програмою «Інженерія програмного забезпечення комп'ютерних систем» за спеціальністю 121 «Інженерія програмного забезпечення».

Дисципліна "Програмне забезпечення високопродуктивних комп'ютерних систем" спрямована на опанування студентами методів та засобів побудови програмного забезпечення для високопродуктивних комп'ютерних систем (ВПКС) з використанням технологій паралельного програмування.

Дисципліна забезпечує наступні компетентності та програмні результати навчання освітньо-професійної програми :

- ФК17 Здатність розробляти та використовувати методи і алгоритми високопродуктивних обчислень
- ФК18 Здатність розробляти та використовувати програмне забезпечення для високопродуктивних комп'ютерних систем
- ПРН25 Знати програмне забезпечення високопродуктивних комп'ютерних систем
- ПРН27 Знати методи і алгоритми високопродуктивних обчислень.

Метою викладання дисципліни є отримання студентом знань та навичок у сфері створення програмного забезпечення для ВПКС, зокрема концепції потоку, реалізації потоків в сучасних мовах і бібліотеках паралельного програмування, методів та засобів організації взаємодії потоків, рішення завдання взаємного виключення та синхронізації потоків, засобів програмування взаємодії потоків: семафорів, мютексів, подій, критичних секцій, моніторів, повідомлень, аналізу та побудови паралельних алгоритмів.

Метою навчальної дисципліни є формування у студентів компетентності розробки програмного забезпечення для ВПКС з паралельною архітектурою на основі володіння засобами сучасних мов та бібліотек паралельного програмування на робочих місцях і підрозділах під час майбутньої професійної діяльності на першої посаді. За результатами вивчення дисципліни студент має бути здатними вирішувати професійні завдання та володіти таким::

оцінювати особливості ВПКС, що застосовуються;

використовувати методі створення та оцінювання паралельних алгоритмів;

використовувати засоби сучасних мов та бібліотек паралельного програмування для створення програмного забезпечення сучасних ВПКС,

володіти основними методами формування програм на основі концепції потоків.

ЗДАТНІСТЬ:

- розробки програмного забезпечення для комп'ютерних систем з паралельною або розподіленою архітектурою на основі володіння засобами сучасних мов та бібліотек паралельного програмування
- аналізувати структуру паралельної комп'ютерної системи - аналізувати паралельні властивості задачі, що вирішується
-розробляти та оцінювати паралельні алгоритми
-здійснювати організацію обчислень в паралельних та розподілених комп'ютерних системах

ЗНАННЯ:

- методів та засобів організації обчислень у високопродуктивних комп'ютерних системах
- структур сучасних паралельних комп'ютерних систем,
- організації пам'яті та зв'язку процесорів,
- методів аналізу, оцінювання та представлення паралельних алгоритмів, концепцій розробки паралельних програм
- методів програмування процесів, концепції потоку, операцій з потоками,
- методів та засобів організації взаємодії потоків ,
- засобів програмування процесів за допомогою сучасних мов та бібліотек паралельного програмування,
- моделей взаємодії процесів,
- постанови та вирішення завдання взаємного виключення та синхронізації процесів,
- методів налагодження паралельних програм.

УМІННЯ:

- організації обчислювальних процесів в сучасних комп'ютерних системах ,
- створювати програмне забезпечення для високопродуктивних комп'ютерних систем з різною архітектурою ,
- виконувати побудову паралельного алгоритму,

- аналізувати ефективність паралельного алгоритму, формувати алгоритми процесів,
- програмувати процеси (потоки) та створювати паралельну програму,
- організовувати ефективну взаємодію процесів в залежності від структури ВПКС,
- розміщувати процеси по процесорах ВПКС,
- виконувати налагодження та виконання паралельної програми.

ЛЕКЦІЯ 1
Структури високопродуктивних комп'ютерних систем

Розглядаються структури високопродуктивних комп'ютерних систем (ВПКС), що ґрунтуються на паралельній обробці інформації. Проаналізовано особливості організації процесорів, пам'яті та зв'язку процесорів (вузлів) системи.

1.1. Паралельні комп'ютерні системи

Загальну структуру високопродуктивної комп'ютерної системи (ВПКС), що ґрунтується на паралельній організації, можна подати як сукупність *вузлів* та *системи взаємодії* вузлів (рис. 1.1).

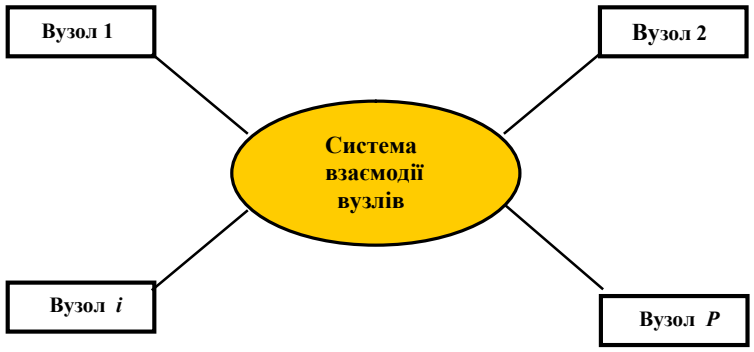


Рис. 1.1. Загальна структура ВПКС

Вузли системи використовують для обробки та зберігання інформації, систему взаємодії – для передавання даних між вузлами системи. Система взаємодії вузлів – це центральна ланка комп'ютерної системи, оскільки визначає її можливості щодо реалізації передавання даних між вузлами системи. Ідеальними з погляду організації взаємодії вузлів є *повно- зв'язані* системи, у яких система зв'язків реалізує можливість *прямої* взаємодії двох будь-яких вузлів системи, тобто передавання даних за один крок без використання проміжних вузлів.

Залежно від реалізації вузлів і системи зв'язків розрізняють три основні структури ВПКС, що ґрунтуються на загальній структурі:

- КС з *розподільною* (*спільною)* пам'яттю;
- КС з *розподіленою* (*локальною*) пам'яттю;
- *розподілені* КС.

1.2. Комп'ютерні системи зі спільною пам'яттю

Структуру ВПКС з розподільною пам'яттю зображено на рис. 1.2. В якості вузлів в ній використовують процесори, в якості системи взаємодії вузлів – спільну пам'ять .

Взаємодія процесорів у таких системах здійснюється за допомогою спільної пам'яті. Процеси (птоки), що виконуються в кожному процесорі, можуть обмінюватися даними і синхронізуватися за допомогою спільних змінних, які в програмі оголошуються як глобальні.

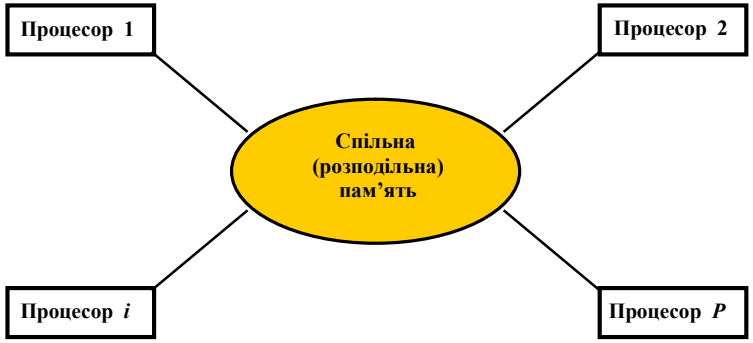


Рис. 1.2. Структура ВПКС з розподільною пам'яттю

Така модель отримала назву взаємодії процесів, яка ґрунтується на *розподільних змінних (shared – variables based synchronization and communication*). Система з розподільною пам'яттю є повнозв'язаною, тому що спільна пам'ять забезпечує безпосередній зв'язок кожного процесора з кожним. Для цього один процес записує значення потрібної локальної змінної (яку треба передати в інший процес) у глобальну змінну, звідки інший процес зчитує це значення (копіює в свою локальну змінну).

Реалізацію структури комп'ютерної системи зі спільною пам'яттю у вигляді симетричної мультипроцесорної системи (СМП системи), у якій використовується шинна організація, показано на рис. 1.3. Лімітована пропускна здатність шини дозволяє використовувати в системі не більше 8–16 процесорів. Симетричні мультипроцесорні системи мають назву однорідних (UMA – uniform memory access), оскільки час доступу кожного процесора до пам'яті та інших пристроїв системи однаковий (симетричний доступ). Поряд з шинами в СМП системах використовують також комутатори або комбінації шин і комутаторів.

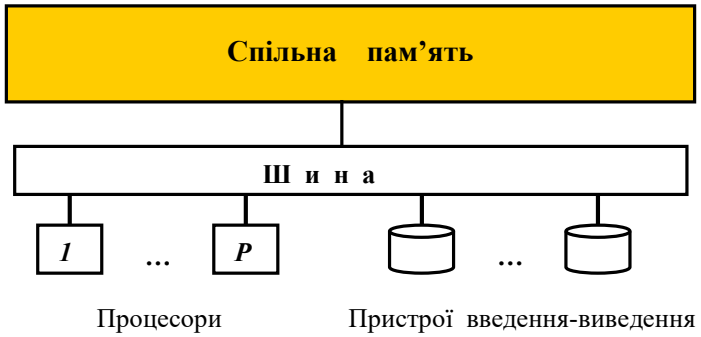


Рис. 1.3. Симетрична мультипроцесорна система

Неоднорідні системи (NUMA – non uniforn memory access) включають десятки процесорів, пам'ять у них організована за ієрархічним принципом. Така організація пам'яті дозволяє уникнути перевантажень на шини, але призводить до нерівномірного доступу до різних ділянок пам'яті.

Наявність в кожному процесорі кеш-пам'яті великої ємності створює передумову для підвищення ефективності виконання програми, якщо в системі забезпечено ефективне використання сумарної ємності кеш-пам'яті системи.

Наявність у процесорах кеш-пам'яті пов'язана також з проблемою *погодження* (когерентності) кеш-пам'яті, яка виникає за одночасного звертання процесорів до однієї ділянки кеш-пам'яті для читання або зміни даних. Проблема вирішується за допомогою реалізованих в кожному процесорі протоколів погодження кеш-пам'яті.

Проблема погодження існує і для спільної пам'яті. Її розв'язування ґрунтується на об'єднанні апаратного **погодження** звернень процесорів до пам'яті та програмного контролю доступу до спільних ділянок пам'яті, який покладається на програміста і який пов'язаний з використанням в програмі спеціальних синхронізованих операцій.

1.3. Багатоядерні комп'ютерні системи

З появою багатоядерних процесорів з'явилася можливість побудови ВПКС, основу яких складали різноманітні за структурою, кількістю ядер, організацією кеш-пам'яті різних рівнів та обсягу, систем зв'язку ядер процесори.

Для підтримки максимальної продуктивності комп'ютерних компанія Intel розробила лінійку процесорів Scalable Xeon. Лінійка містить моделі Intel Xeon Bronze, Silver, Gold и Platinum, яки відрізняються ціною та рівнем швидкодії.

Основні переваги процесорів Intel Xeon Scalable:

- Збільшена продуктивність і кількість ядер. Процесори серії 9200 підтримують до 56 ядер, що забезпечує можливість масштабування обчислювань.
- Підтримка до 48 ліній PCIe 3.0 дає можливість працювати з высоконавантаженними серверами без зниження швідкодії.
- Технологія Intel Ultra Path Interconnect забезпечує ідеальний баланс між єнєрго ефективністю і продуктивністю
- Процесори оптимізовано для роботи з навантаженням штучного інтелекту завдяки підтримки технології Intel Deep Learning Boost.

- Підвищений рівень безпеки на апаратному рівні досягається використанням бібліотек Intel Security Libraries for Data-Center. Бібліотеки мають модульну архітектуру і можуть використовуватися для побудови безпечної інфраструктури для роботи с критично важливим навантаженням.
- Інтегрована технологія QuickAssist Technology забезпечує прискорення роботи серверів
- Підтримка пам'яті Intel Optane DC дає можливість працювати з сучасними бізнес-додатками на якісно новому рівні.

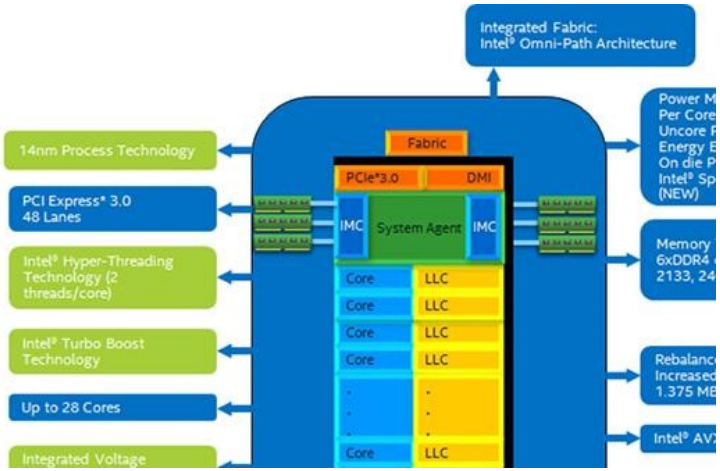


Рис.1.4 Схема архітектури сімейства процесорів Scalable Intel Xeon

Таблиця 1.1 Параметри процесорів сімейства Scalable Intel Xeon

Features	Intel® Xeon® processor E5 2600 product family	Intel® Xeon® processor E7 4600 product family	Intel® Xeon
Platform	Grantley	Brickland	
CPU TDP (with IVR)	55-145W, 160W WS only	115-165W	
Socket	Socket R3	Socket R1	
Scalability	2S	2S, 4S, 8S	
Cores	Up to 22C with Intel® HT Technology	Up to 24C with Intel® HT Technology	Up to 28C
Mid-Level Cache	256kb private	256kb private	
Last Level Cache	Up to 2.5MB/core (Inclusive)	Up to 2.5MB/core (Inclusive)	Up to 1.3
Memory	4 channels DDR4 per CPU RDIMM, LRDIMM	8 channels DDR4 per CPU with memory buffer RDIMM, LRDIMM	6 ch
	1DPC=up to 2400, 2DPC= up to 2400, 3DPC=up to 1866	DDR3/4 Performance Mode 1333, 1600 DDR3/4 Lockstep mode 1333, 1600, 1866	213
Point to Point Link	Intel® QPI: 2 v1.1 channels per CPU 9.6 GT/s max	Intel® QPI: 3 v1.1 channels per CPU 9.6 GT/s max	UPI: 2-3 cha
PCIe*	PCIe* 3.0 (2.5, 5.0, 8.0 GT/s)	PCIe* 3.0 (2.5, 5.0, 8.0 GT/s)	PCIe*
	40 lanes per CPU	32 lanes per CPU	Bifurca
PCH	Wellsburg: DMI2 – 4 lanes; Up to 6xUSB3, 8x USB2 ports,	Patsburg: 14x USB2 ports, 4x SATA2 ports, 2x SATA3 ports	Lewisburg: D Up to: 10xU

1.4. Комп'ютерні системи з розподіленою (локальною) пам'яттю

Структуру комп'ютерної системи з розподіленою (локальною) пам'яттю показано на рис. 1.4. У вузлі системи розміщуються процесор і локальна пам'ять (ЛП), для взаємодії вузлів використовується спеціальна система зв'язків, що зв'язує вузли за допомогою лінків (*link*) .

Рис. 1.4. Структура КС з локальною (розподіленою) пам'яттю

Потоки можуть обмінюватися даними і синхронізуватися передаванням (прийманням) даних між вузлами системи. Така модель отримала назву взаємодії процесів, що ґрунтується на *посиланні повідомлень* (*message-passing based synchronization and communication).*

Системи з розподіленою пам'яттю належать до категорії *тісно зв'язаних* (tightly couple) систем**.** Система ставить високі вимоги до пропускної здатності системи зв'язків, оскілки вона має забезпечувати швидке передавання даних від одного процесора до іншого.

Реалізацію повного зв'язку процесорів у системах з розподіленою пам'яттю можна забезпечити тільки для невеликої кількості процесорів. Для використання великої кількості процесорів їх сполучують за допомогою різних топологій: лінійних, кільцевих, деревоподібних, решітчастих (матричних), зіркових, гіперкубічних та інших (рис. 1.5).

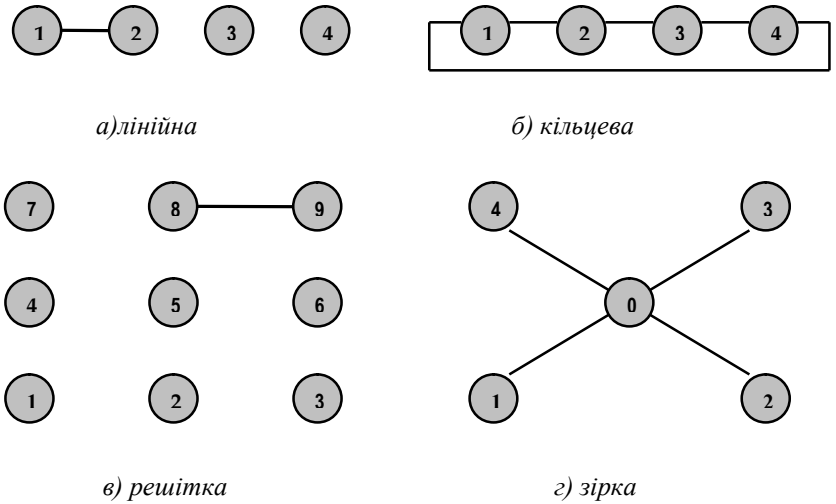
Кожна структура характеризується визначеними топологічними параметрами:

- кількістю процесорів (*P*);
- діаметром системи (*D*), який визначає відстань між двома найбільш віддаленими процесорами системи;
- кількістю зв'язків між вузлами (*R*);
- ступенем вузла (*S*), що задає кількість зв'язків цього вузла.

Тпологічні характеристики структур комп'ютерних систем, які показано на рис. 1.5, їх наведено в табл. 1.1.

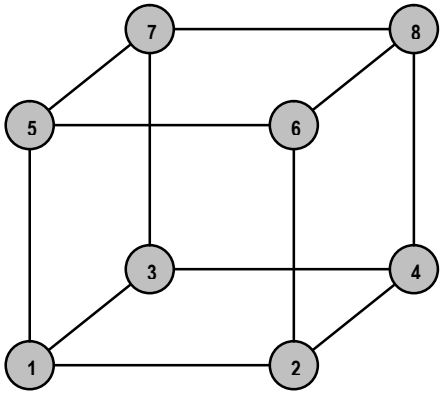
Таблиця 1.2 Топологічні характеристики КС

Структура	*P*	*D*	*R*	*S*
Лінійка	4	3	3	<= 2
Кільце	4	2	4	2
Решітка	9	4	12	<= 4
Зірка	5	2	4	<= 4
Гіперкуб	8	3	12	3



а)лінійна *б) кільцева*

в) решітка *г) зірка*



д) гіперкуб

Рис. 1.5. Структури КС з локальною пам'яттю:
а – лінійка; *б* – кільце; *в* – решітка; *г* – зірка;

1.5. Розподілені комп'ютерні системи

Структуру розподіленої комп'ютерної системи (РКС) зображено на рис. 1.6. Як вузли в ній використовують персональні комп'ютери (ПК), робочі станції або багатопроцесорні КС, як систему взаємодії вузлів – комп'ютерну мережу.

Головна відмінність РКС від КС з локальною пам'яттю полягає в тому, що кожний вузол РКС працює під керуванням власної операційної системи (ОС), в той час, як у КС з локальною пам'яттю використовують лише одну операційну систему, яка керує всіма процесорами.

Модель взаємодії вузлів у РКС ґрунтується на посиланні повідомлень, однак її реалізація відрізняється від прийнятої в системах з локальною пам'яттю, оскільки ґрунтується на мережевих технологіях, що потребує використання спеціальних програмних засобів, таких як, наприклад, протоколи, IP - адреси, сокети, віддалені методи і та ін.

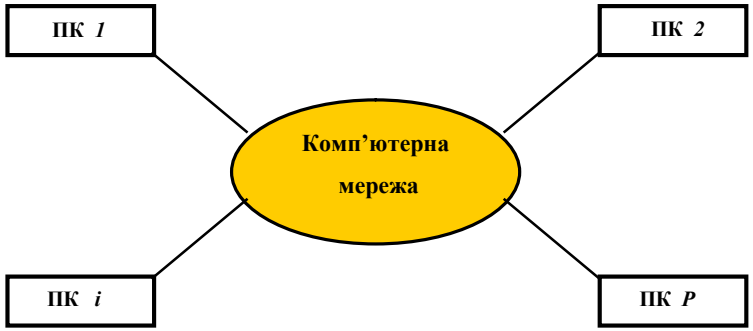


Рис. 1.6. Структура РКС

Розподілені комп'ютерні системи – недорогий і поширений засіб побудови високопродуктивних комп'ютерних систем. Можливість об'єднати десятки тисяч комп'ютерів дозволяє створювати потужні розподілені системи, які не поступаються багатопроцесорним системам за можливостями, легко розширюються і перебудовуються, а головне – значно дешевші.

Кластерні обчислювальні системи – вид розподілених систем, орієнтованих на високопродуктивні обчислення. Проект *Beowulf* був одним з перших, у якому запропоновано підходи до побудови кластерних систем. Особливість побудови кластерних систем ґрунтується на використанні *спеціалізованих* мережевих засобів (адаптерів, комутаторів), які дозволяють значно прискорити передавання даних між вузлами системи, що традиційно є слабким місцем розподілених систем.

Список найбільш потужних КС світу *Top500* (www.top500.org) уже кілька років обов'язково налічував кластерні системи. У травні 2023 р. цей список налічував кластерні системи, найбільш потужна з яких – Fugaku Fujitsy , 158976 вузлів, 7630848 ядер, Tofu Interconnect займає друге місце) мала максимальну продуктивність 442 Pflop/s.

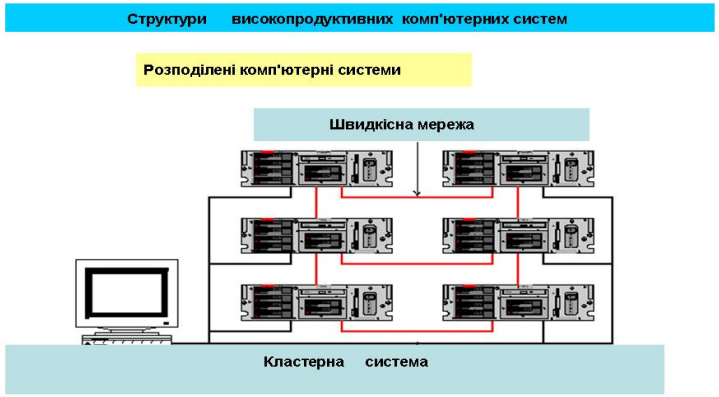


Рис. 1.7 Кластерні системи

Очолює рейтинг КС HPE Cray EX, яка має 8,699,904 ядер і продуктивність 1.194 Exaflops .

➢ **Запитання для модульного контролю**

1. Що включає загальна структура паралельної КС?
2. Які існують види організації пам’яті в КС?
3. У чому особливість побудови моделі спільних змінних та її використання?
4. У чому особливість побудови моделі посилання повідомлень та її використання?
5. Які переваги і недоліки має КС зі спільною пам’яттю?
6. Які переваги і недоліки має КС з розподіленою пам’яттю?
7. У чому полягає відмінність реалізації моделі посилання повідомлень у КС з локальною пам’яттю і в РКС?
8. Яка із структур КС, поданих на рис. 1.5, має найменший діаметр ?
9. В чому полягає відмінність організація кеш-пам’яті в багатоядерних процесорах Intel і AMD ?

➢ **Завдання для самостійної роботи**

1. Побудувати КС, що має зіркову структуру з $P = 8$.
2. Виконати модифікацію структури на рис. 1.5, *г*, змінивши всі вузли, крім центрального, на топологію «зірка» з $P = 4$. Розрахувати топологічні характеристики такої КС.
3. Побудувати КС різної структури, які мають діаметр, що дорівнює 8.
4. Визначити діаметр КС, що має топологію решітки з кількістю процесорів, що дорівнює P.
5. Проаналізувати структури КС, що показані на рис.1.5, з погляду надійності, у разі, якщо один з процесорів вийде з ладу.
6. Побудувати структури КС для гіперкуба, якщо значення ступеня гіперкуба дорівнють 0, 1, 2 , 3, 4.
7. Розрахувати топологічні характеристики КС типу лінійки із $P = 6$.
8. Розрахувати топологічні характеристики КС типу кільця із $P = 5$.
9. Розрахувати топологічні характеристики КС типу решітки.
10. Розрахувати топологічні характеристики КС типу гіперкуба ступеня 4.

ЛЕКЦІЯ 2
Потоки 1

2.1. Поняття потоку.
2.2. Стан потоку. Керування потоками.
2.3. Потоки в мові Java.
2.5. Потоки в мові Ada.

В лекції розглянуто концепцію потоку, стани потоку та операції над потоками. Наведено приклади реалізацій потоків та операцій над ними, що застосовані в сучасних мовах та бібліотеках паралельного програмування.

2.1. Поняття потоку

На рис. 2.1 показано відмінність послідовної і паралельної програм. Якщо послідовна програма будується на онові процедур (методів), то особливістю паралельної програми є використання *потоків*.

Розрізняють поняття процесу і потоку, між якими є багато спільного. *Процес* – абстрактне поняття, що включає опис певних дій, пов'язаних з виконанням програми в комп'ютерній системі. При цьому процес оформлюється так, щоб система керування процесами в операційної системі (ОС) могла ефективно перерозподіляти ресурси системи (процесори, пам'ять, прилади введення-виведення, файли і т.ін.). Процес характеризується власним набором ресурсів і виділеною для нього ділянкою оперативної пам'яті.

Поняття процесу вперше з'явилось в багатозадачних операційних системах і сьогодні є фундаментальним для кожної сучасної ОС. В ОС процес пов'язаний з кожною множиною прикладних або системних програм, які виконуються в комп'ютерній системі. Це дозволяє системі керування процесами ОС, яка розміщується в ядрі ОС, ефективно маніпулювати процесами за допомогою спеціального блоку керування процесом (БКП) або PCB (Process Control Block) [2].

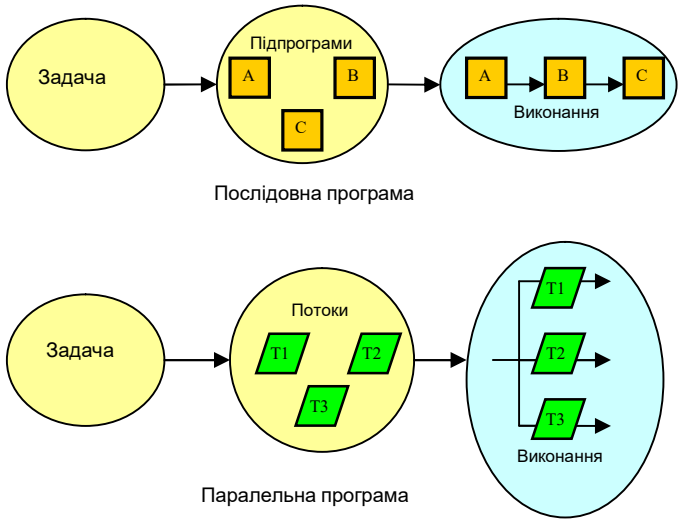


Рис. 2.1 Послідовна і паралельна програми

Блок керування процесом – динамічна структура даних, яка містить основну інформацію про процес:

- ім'я процесу;
- поточний стан процесу;
- пріоритет процесу;
- місце розміщення процесу в пам'яті;
- ресурси, що пов'язані з процесом та інше.

З появою паралельного програмування з'явилось поняття *потоку* – частини однієї програми користувача, які виконуються одночасно. Потоки отримали також назву *легкі процеси* (*lightweight processes*). Для позначення легких процесів використовують також

терміни: *потік* (Java, C, C#) або *задача* (Ada, MPI). Тому традиційні процеси ОС називають *важкими процесами* (*heavyweight processes*).

Під час виконання кількох програм у комп'ютерній системі або звичайному комп'ютері відбувається постійне перемикання з одного процесу на інший. Те ж саме відбувається і для паралельної програми, якщо вона включає кілька потоків. Але час, який витрачається на перемикання для потоків, менший, ніж для процесів. Ще одна відмінність потоків і процесів полягає в тому, що потоки постійно взаємодіють, оскільки є частинами однієї програми (так звані *тісно зв'язані* процеси), в той час, як процеси взаємодіють рідко (*слабко зв'язані* процеси).

Через те, що потоки є частинами однієї програми (одного процесу), вони використовують загальний адресний простір цього процесу. Це дозволяє потокам бачити і використовувати змінні інших потоків, надає додаткові можливості для організації взаємодії потоків, але ускладнює розробку та налагодження паралельної програми.

2.2. Стан потоку. Керування потоками

Керування процесом і потоком має багато спільного, воно включає виконання деяких операцій над ним, дозволяючи процесу (потоку) пройти визначені стани (рис. 2.2). Розглянемо стани процесу які характерні і для потоків. Види станів процесу визначаються конкретною ОС і системою керування процесами, але змістять здебільшого такі стани:

Породження – створення процессу і підготовка до першого виконання в системі;
Готовності – процес готовий до виконання і чекає звільнення процесора;
Виконання – процес виконується на процесорі;
Блокування – процес призупинений через очікування визначеної події: завершення введення даних, завершення заданого часу очікування, сигналу від іншого процесу, звільнення ресурсу та ін.;
Завершення – нормальне або аварійне завершення

виконання процесу.

Причинами переходів є операції, що виконуються ОС над процесом: породження і завершення процесу, блокування і розблокування, завершення кванта часу роботи процесора, операції уведення-виведення та ін.

Процес також може мати особливий стан, який отримав назву *тупик* (*deadlock*). Процес у тупиковому стані блокований і очікує на подію, яка ніколи не відбудеться (сигналу від іншого процесу, звільнення ресурсу, введення даних та інше). Це зумовлює неможливість його продовження і, як наслідок, – зависання програми в цілому. Тупики – одна з головних проблем, що виникають під час виконання паралельних програм.

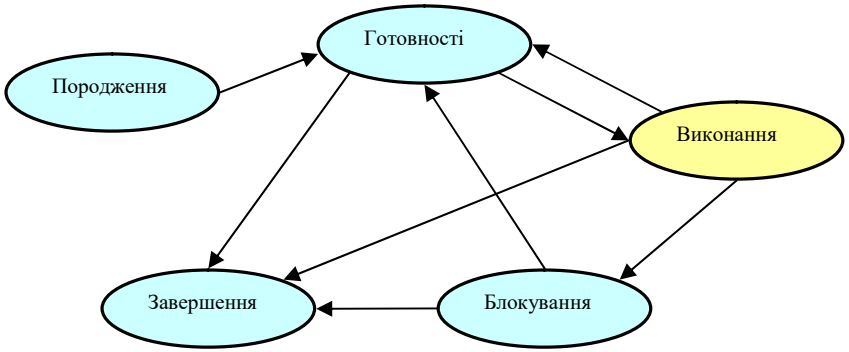


Рис. 2.2. Діаграма зміни станів процесу (потоку)

Боротьба з тупиками зводиться до таких дій:

- відвернення тупиків;
- запобігання тупикам;
- визначення тупика;
- ліквідація тупика.

У цілому тупики є істотною загрозою і боротьбі з ними слід приділяти особливу увагу під час розроблення та налагодження

паралельної програми. Приклади і методи запобігання тупикам будуть розглянуті в розділах, у яких вивчається взаємодії потоків.

Засоби роботи з потоками розподіляються на бібліотечні та мовні. Прикладами перших є засоби бібліотек WinAPI, PVM, MPI, OpenMP, Pthread, які реалізовані у вигляді набору ресурсів (функцій, змінних, типів). Вони дозволяють роботу з потоками в будь-яких мовах програмування, зокрема і тих, які не мають вбудованих засобів роботи з процесами.

Пізніші мови програмування, такі як C#, Java, Python або Ада, безпосередньо мають засоби роботи з потоками, які вбудовані в мови, що забезпечує ефективне їх виконання та взаємодію.

Незалежно від реалізації (бібліотечної або мовної) засоби роботи з потоками мають забезпечити розробнику паралельного додатка такі можливості:

- об'явити потік (групу потоків);
- встановити пріоритет потоку;
- запустити потік на виконання,
- призупинити потік на визначений час;
- блокувати та розблокувати потік;
- організувати взаємодію з іншими потоками (синхронізація або передавання даних);
- завершити потік.

Існує декілька шляхів до створення потоків:

- через використання спеціальних модулів (класів), яки дозволяють описати потік (Java, Ada);
- через так звані *потокові функції,* коли дії майбутнього потоку описуються за допомогою функції, яка потім використовується для створення процесу (WinAPI, C#);
- через створення копій програми (MPI, PVM);
- через визначення в програмі ділянок, які можуть виконуватися паралельно (Occam, OpenMP).

2.3. Потоки в мові Java

Потік в мові Java реалізований у вигляді *потоку* (`thread`). Java – об'єктно-орієнтована мова програмування, тому класи-

потоки створюються з використанням спеціального класу `Thread` [23]. Можливі два підходи до створення потоку:

- за допомогою наслідування класу `Thread;`
- з використанням інтерфейсу `Runnable.`

Використання класу Thread. Клас `Thread` інкапсулює потік і містить в собі набір методів для керування потоком (табл. 2.1). Для створення потоку потрібно оголосити підклас через розширення класу `Thread` і створити екземпляр цього підкласу. Підклас повинен перевизначити метод `run()`, що є точкою входу в потік. Необхідно також викликати метод `start()`, щоб почати виконання потоку.

Клас `Thread` має сім конструкторів. Простий конструктор не має параметрів, а інші конструктори дозволяють задавати як параметри: інтерфейс `Runnable`, ім'я потоку або ім'я групи потоків.

Таблиця 2.1. Методи класу `Thread`

Метод	Дія
`getName()`	Отримати ім'я потоку
`getPriority()`	Отримати пріоритет потоку
`isAlive()`	Визначити, чи виконується потік
`Join()`	Чекати завершення потоку
`run()`	Указати точку входу в потік
`Sleep()`	Призупинити потік на заданий інтервал часу
`Start()`	Запустити потік на виконання

❖ Приклад 2.1

```
/* ------------------------------------------------------
-- Java. Створення потоків Nord і West шляхом          --
--       розширення класу  Thread                      --
------------------------------------------------------*/
   class  Nord   extends   Thread {

      // перевизначення методу  run()
      public void run(){
         System.out.println("Process Nord ");
      }
   }// Nord  ------------------------
```

```
class  West extends Thread {

   public void run(){
      System.out.println("Process West ");
   }
}// West -------------------------

// головний потік
class  MainThread {

   // точка входу в основний клас
   public static void main(String  args []){

      // оголошення екземплярів  потоків
      Nord N = new  Nord();
      West W = new  West();

      // запуск потоків
      N.start();
      W.start();

      // продовження виконання головного потоку
      System.out.println("Process MainThread
                                      finished ");

   }// main
}// MainThread
```

Керувати порядком запуску потоків можна за допомогою пріоритету потоку. Для встановлення пріоритету потоку використовують метод `setPriority()` з класу `Thread`:

```
final void setPriority (int  Рівень);
```

де параметр `Рівень` визначає пріоритет потоку. Пріоритет вибирається з діапазону `MIN_PRIORITY` і `MAX_PRIORITY`, граничні значення якого відповідно дорівнюють 1 і 10. За умовчанням потік отримує пріоритет `NORM_PRIORITY`, що дорівнює 5.

Призупинити потік на зазначений відрізок часу можна за допомогою методу `sleep()`:


```
static void sleep(long Час) throws
                              InterruptedException;
```

де параметр `Час` задає час затримки потоку в мілісекундах. Метод може збуджувати виключення `InterruptedException`, що потребує обов'язкового створення блоків `try/catch` під час його використання.

Метод `yield()` викликає призупинення поточного потоку, після чого з черги готових до виконання потоків вибирається і запускається потік з більшим або однаковим пріоритетом:

```
final void yield();
```

Якщо в черзі готових до виконання немає потоків з таким пріоритетом, то потік продовжує своє виконання. Метод `yield()` дозволяє організувати більш "справедливе" виконання потоків.

Метод `join()` дозволяє організовувати очікування завершення викликаного потоку. Можливе використання в головному потоці, якщо потрібно, щоб він був завершений останнім – після завершення усіх запущених ним потоків:

```
final void join()
              throws InterruptedException;
```

Використання інтерфейсу Runnable. Потік можна створити також, визначивши клас, який реалізує інтерфейс `Runnable`. В інтерфейсі `Runnable` описано абстрактний метод `run()`, який необхідно визначити в створюваному класі - потоці:

```
public void run();
```

У методі `run()` слід описати код, який визначає дії створюваного потоку. Далі під час створення класу оголошується об'єкт типу `Thread`. При цьому можна використати конструктори, визначені в класі `Thread`. Після оголошення потокового об'єкта необхідно викликати його метод `start()`, який, в свою чергу, знаходить метод `run()`, визначений в об'єкті, і передає йому керування.

❖ Приклад 2.2

```
/* ------------------------------------------------------------
-- Java. Створення потоків використанням                    --
-- інтерфейсу    Runnable                                   --
------------------------------------------------------------*/
    class Denon implements Runnable{

        String  Name;
        Thread  t;

        // Конструктор класу Denon
        Denon(String ім'я){
           Name = ім'я;
           t  = new  Thread(this, N);
           System.out.println("   New thread" + Name);
           t.start();
        }

        // точка входу в потік
        public void run(){

           try{
              for (int i=1; i<5; i++){
                 System.out.println("Start of process "
                                                  + Name);
                 Thread.sleep(500);
              }
           }
           catch(InterruptedException e){
              System.out.println("Error in process");
           }
           System.out.println(Finish of process "
                                              + Name);
      }
    }// Denon
    // Головний потік, що використає клас Denon
    public class  Titan {

       // точка входу в основний клас
       public static void main(String  args[]){
          System.out.println("Process Titan started");

          // оголошення екземплярів класу Denon
          Denon A = new Denon("A");
          Denon B = new Denon("B");
```



```
          // чекати завершення потоку A і B
          try{
             A.t.join();
             B.t.join();
          }
          catch(InterruptedException e){
             System.out.println("Error in Titan
                                          process");
          }
          System.out.println("Process Titan finished ");

       }// main
    }// Titan
```

2.4. Потоки в мові Ada

Ada – відома мова програмування, яку було розроблено в 1983 році на замовлення Міністерства оборони США [2, 4, 5, 18, 22] . Призначення мови – розроблення великих програмних систем, робота яких характеризується високою надійністю. Ада – одна з перших мов програмування, які мають вбудовані засоби роботи з потоками.

Потоки в мові Ada реалізовані у вигляді спеціального модулю – *задачі* (`task`). Задачний модуль `task` один із п'яти видів модулів мови, має стандартну форму у вигляді специфікації та тіла. Специфікація визначає інтерфейс задачі, де задається ім'я задачі, а також у разі необхідності, – пріоритет задачі, засоби взаємодії з іншими задачами, місце розташування в пам'яті. Простіший вид специфікації задачі містить тільки ім'я задачі. Така специфікація має назву *виродженої.* Тіло задачі визначає дії задачі під час виконання.

Задачний модуль не є одиницею компіляції в мові і тому задача має бути описаною в підпрограмі або пакеті (в компільованому модулі). Мова немає явних засобів запуску задачі, як, наприклад, в мові Java. Задача *автоматично* стартує під час запуску підпрограми або входячи в блок, у яких вона описана. Якщо задачі розміщені в головній програмі, то при запуску основної програми задачі починають виконуватися одночасно з головною програмою, яка, в свою чергу, завжди розглядається як задача і виконується паралельно з вкладеними в ній задачами. Явний запуск задач можливо

реалізувати через розміщення задач в окремій процедурі, виклик якої в потрібний момент спричинить запуск вкладених у ній задач. Крім того, запуск задач можливо реалізувати за допомогою посилального (access) типу та генератора new.

❖ Приклад 2.3

```
------------------------------------------------------------
--  Ада.Оголошення та запуск задач                        --
------------------------------------------------------------
procedure Lab23 is

    -- специфікації задач A і B (вироджені)
    task A;
    task B;

    -- тіла задач A і B
    task body A is
    begin
       put_line("Process A started");
    end A;

    task body A is
    begin
       put_line("Process B started");
    end B;

-- основна програма
begin
       -- місце автоматичного запуску задач A і B
    put_line("Main procedure started ");
end Lab23;
```

Задачний тип (task type) дозволяє користувачу описати тип, об'єктами якого будуть задачі. Задачі, які створені з використанням задачного типу, ідентичні. Однак реалізація задачного типу за допомогою *дискримінанта* дозволяє внести необхідні особливості в кожну задачу для її ініціалізації. Механізм дискримінантів схожий на конструктори класів у мові Java.

❖ Приклад 2.4

```
 ------------------------------------------------------------
 --   Ада.Задачний тип з дискримінантом                    --
 ------------------------------------------------------------
```



```
 procedure Lab24 is
    -- задачний тип з дискримінантом
    task type  ЗадачнТип(Номер : integer);

    -- тіло задачного типу
    task body ЗадачнТип   is
       Номер_Задачі :  integer:= Номер;
    begin
       put(" Process  started  ");
       put(Номер_Задачі);
    end ЗадачнТип;

    A: ЗадачнТип(1);  -- створення задач
    B: ЗадачнТип(2);
begin
   null;              -- порожний оператор
end Lab24;
```

Пріоритет процесу задається в специфікації задачі за допомогою прагми Priority. Пріоритет задачі – ціле число в діапазоні 1 – 7. Більше значення відповідає більшій пріоритетності задачі. Пріоритет визначає спроможність задачі виборювати ресурси – процесор, прилади введення-виведення, а також взаємодіяти з іншими задачами або захищеними модулями. Тобто пріоритети працюють там, де з'являються черги, у яких процеси можуть знаходитись під час очікування. Формування таких черг та вибірка з них здійснюються з використанням пріоритетів задач.

Призупинення задачі на вказаний відрізок часу в мові виконується за допомогою оператора delay, у якому час очікування вказується в секундах (тип Duration із системного пакета Calendar). Під час виконання оператора delay задача блокується і керування передається іншій задачі, яка знаходиться у стані готовності. Оператор delay until T блокує задачу до вказаного часу T.

❖ Приклад 2.5

```
------------------------------------------------------------
--Ада.Використання пріоритетів і оператора delay     --
------------------------------------------------------------
procedure Lab25 is

   -- специфікації задач A і B
```

```
   task A is
      pragma Priority(4);    -- пріоритет задачі A
   end  A;
   task B is
      pragma Priority(5);    -- пріоритет задачі B
   end B;

   -- тіло задач A і B
   task body A is
   begin
      put_line("Process A started");
      delay(4.5);     -- затримка задачі A на 4,5 с
      put_line("Process A finished");
   end A;
   --------------
   task body B is
   begin
      put_line("     Process B started");

      delay(7.4);     -- затримка задачі B на 7,4 с

      put_line("     Process B finished");
   end B;

-- основна програма
begin

   put_line("  Main procedure started ");

   delay(12.25);          -- затримка на 12,25 с

   put_line(" Main procedure finished  ");

end Lab25;
```

- **Запитання для модульного контролю**
  1. Наведіть визначення поняття процесу і потоку та його подання в операційній системі.
  2. Які існують види станів потоків ? Що визначає перехід потоку в інший стан ?
  3. Як упорядковано потоки, що чекають в черзі блокованих процесів – за пріоритетами або за надходженням ?

4. Як виконується програма, яка складається з кількох потоків в одно процесорній системі ?
5. У чому полягає відмінність між двома способами створювання потоків у мові Java ?
6. У чому полягає відмінність механізму запуску потоківів в мовах Ada та Java ?
7. Особливості створювання і керування запуском потоків в Java ?
8. Які прагми мови Ada використовують при створюванні потоків ?
9. Потоки в Java. Встановлення пріоритету?
10. Які атрибути в мові Ada пов'язані із задачами ?

ЛЕКЦІЯ 3
Потоки 2

3.1. Потоки в бібліотеці WinAPI

Бібліотека WinAPI (Windows API) входить до складу ОС Windows і містить набір функцій, які призначені для роботи з потоками [4, 25].

Для створення потоку в WinAPI використовують функцію `CreateThread()`. Функція повертає ідентифікатор потоку, який є унікальним і визначає його в системі. Під час створення потоку визначається початковий адрес коду, з якого має виконуватись потік. Зазвичай, це назва функції (*потокової функції*), яка вже створена і буде виконуватися як потік.

Функція:

```
HANDLE Ім'я_Потоку = CreateThread(
   LPSECURITY_ATTRIBUTES  атр,  // атрибут безпеки
   SIZE_T    рс,               // розмір стека
   LPTHREAD_STERT_ROUTUNE фп,   // функція потоку
   LPVOID  афт,         // аргумент функції потоку
   DWORD   пр,          // прапорець виконання
   LPDWORD іп);         // ідентифікатор потоку
```

створює потік, для якого фактичні параметри визначають ім'я потокової функції та її параметри, атрибути безпеки, початковий розмір стека потоку, прапорець виконання: негайного (0) або відкладеного (`Create_Susupended`), ім'я потоку.

❖ **Приклад 3.1**

Створення двох потоків, які будуть виконувати потокові функції `Task_ Func1` та `Task_ Func2` :

```
void Task(void);

int main(void){

   DWORD TidA, TidB;
   HANDLE hThreadA, hThreadB;

   // Створення потоків
   hThreadA = CreateThread(NULL, 0,
        (LPTHREAD_START_ROUTINE)Task_Func1,
                                    NULL,0, &TidA);
   hThreadB = CreateThread(NULL, 0,
        (LPTHREAD_START_ROUTINE)Task_Func2,
                                   NULL,0, &TidB);

   // Закриття потоків
   CloseHandle(hThreadA);
   CloseHandle(hThreadB);
}

// Потокова функція 1
void Task_Func1(void){
    printf("Потік 1 стартував ");
        .  .  .
    printf("Потік 1  завершився ");
}
// Потокова функція 2
void Task_Func2(void){
    printf("Потік 2 стартував ");
        .  .  .
    printf("Потік 2 завершився ");
}
```

Потік виконується доти, доки не відбудеться одна з таких подій:

- функція повертає значення потоку;
- потік викликає функцію `ExitThread();`
- інший потік викликає функцію `ExitProcess();`
- інший потік викликає функцію `TerminateThread()` з дескриптором потоку;

- інший потік викликає функцію `TerminateProcess()` з дескриптором процесу.

Бібліотека WinAPI надає багато ресурсів для роботи з потоками. За допомогою функції `GetExitCodeThread()` можна отримати значення стану завершення потоку. Під час виконання потік має стан `STILL_ACTIVE`.

Пріоритет потоку встановлюється за допомогою функції `SetThreadPriority()`. Отримати поточне значення пріоритету процесу можна за допомогою функції `GetThreadPriority()`.

Створюючись, потік отримує пріоритет, що дорівнює значенню `TRHREAD_PRIORITY_NORMAL`.

Потік залишається в системі, поки він не закінчить роботу і всі його дескриптори не будуть закритими за допомогою функції `CloseHandle()`.

Призупинення процесу виконується функціями `Suspend Thread()` і `ResumeThread()`. Функції `Sleep()` і `SleepEx()` блокують виконання процесу на заданий інтервал часу. Функція `SwitchToThread()` блокує потік і передає керування іншому потоку.

3.2. Потоки в мові C#

Мова C# забезпечує програмування потоків через класи-потоки (multithreading) . Програма мовою C# завжди є багатопотоковою, тому що виконання програми розпочинається з створення основного потоку (“main” потоку) і породження потім додаткових потоків [27-31].

Процес в C# має назві *потік* (`thread`). Потік виконується незалежно і паралельно з іншими потоками. Система управління потоками (CLR - common language runtime) зв’язує з кожним потоком *стек пам’яті* для розміщення коду потоку і локальних змінних.

Створення потоку в мові C# схоже на мову Java. Для цього використається спеціальний клас `Thread`, який містить ресурси для роботи з потоками. На відміну від мови Java цей клас не має методу `run()` для визначення дій майбутнього потоку. Дії потоку визначаються попередні за допомогою звичайного методу, який потім

передається як делегат класу `Thread` при створенні об’єкту-потоку.

❖ **Приклад 3.2**

Створення і запуск на виконання двох потоків `Td1` і `Td2`.

```
using System;
using System.Threading;
class Thread_27 {

   static void Main() {

      // Створення потоків
      Thread Td1 = new Thread(MT1);
      Thread Td2 = new Thread(MT2);

      // Запуск потоків
      Td1.Start();
      Td2.Start();

      // дії основного потоку  по
      // завершенню запуску потоків
      Console.Write("Main потік стартував");

   }// Main  ----------------------------

   // метод для виконання в першому потоці
   static void MT1() {
      Console.Write("Потік P1 стартував");
         .  .  .
      Console.Write("Потік P1 завершений");
   }//MT1

   // метод для виконання в другому потоці
   static void MT2() {
      Console.Write("Потік P2 стартував ");
        .  .  .
      Console.Write("Потік P2  завершений");
   }// MT2

}// Thread_37
```

У прикладі 3.2 клас `Thread_27` містить три методи: основний метод `Main()` та методи `MT1()`і `MT2()`. Головний потік з методом `Main()` створює потоки `Td1` і `Td2`, запускає потоки на виконання через виклик методів `Td1.Start()` і `Td2.Start()`, після чого продовжує своє виконання паралельно с потоками `Td1` і `Td1`. Метод `Start()` є точкою входу в потік.

При створенні потоків `Td1` і `Td2` використаний конструктор класу `Thread`, параметром якого є ім'я методів MT1() та MT2(), яки будуть виконуватися у якості тіл відповідних потоків.

Створення потоку і його запуск можна об'єднати:

```
new Thread(td.MT1).Start();
new Thread(td.MT2).Start();
```

При створенні потоку можна використовувати інші конструктори класу `Thread`, яки дозволяють працювати з делегатами, що забезпечують передавання посилань на функції через параметри конструктору.

Делегат `ThreadStart()` із `System.Threadig.ThreadStart` оголошується як

```
public delegate void ThreadStart();
```

Использование делегату `ThreadStart` при створенні потоку

```
Thread T = new Thread(new ThreadStart(MT4));
```

Іменовані потоки. Потік може отримати ім'я при створенні. Це забезпечить додаткові зручності при налагодженні програми.

❖ **Приклад 3.3**

Створення і запуск на виконання потоку .

```
class Thread_2 {

  static void Main() {
```



```
    Thread.CurrentThread.Name = "Потік 1";

    // формування і запуск нового потоку
    Thread  tw = new Thread(Обчислення);
    tw.Name = "Потік_B";
    tw.Start();

 }// Main

  // потокова функція
  static void Очислення() {
    Console.WriteLine ("Потік" +
                    Thread.CurrentThread.Name);
        . . .
  }// Обчислення
}// Thread_2
```

Передавання параметрів при запуску потоків. Делегат `ParameterizedThreadStart()` може передавати параметри при створенні потоку:

```
public delegate void
            ParameterizedThreadStart(object obj);
```

❖ **Приклад 3.4**

C#. Створення і запуск на виконання двох потоків `Td1` і `Td2`.

C#.Опис и запуск потоку з використанням делегата і параметрів

```
using System;
using System.Threading;
class Thread_Run {

   static void Main() {

      // Створення потоку
      Thread td = new Thread(ПотокA);
```

```
      // Запуск потоку
      td.Start(10);

   }// Main


   static void ПотокА(object o) {

      int zz = (int)(o);
      Console.Write("ПотокА стартував"+ zz);
   }
}// Thread_Run
```

Пріоритети потоків. Значення пріоритетів потоків в C# визначаються наступним типом:

```
enum ThreadPriority {Lowest, BelowNormal,
                     Normal, AboveNormal, Highest}
```

Потік, який створюється по замовченню отримує пріоритет `Normal`. Встановлення і контроль пріоритету потоку виконується за допомогою `Thread.Priority`.

Знищення потоку. Знищення потоку виконується викликом методу `Thread.Abort`. Середовище виконання завершує потік і генерує виключення `ThreadAbortException`, після чого виконується код із блоку `finally` потоку.

Призупинення потоку. Метод `Thread.Sleep()` блокує (призупиняє) потік на вказаний відрізок часу:

```
Thread.Sleep(0);     // переключення потоків
Thread.Sleep(1000); // призупинення на 1000 мсек
```

Блокування потоку. Метод `Join()` дозволяє блокувати потік до завершення іншого потоку.

❖ **Приклад 3.5**

Створення і запуск на виконання потоків із використанням методів `Sleep(0)` і `Join()`.

```
using System;
using System.Threading;
class Thread_210 {

   static void Main() {

      // Створення потоку
      Thread tx = new Thread(Печать);

      // Запуск потоку tx
      tx.Start();

      tx.Join(); // блокувати основний потік
                 // до завершення потоку tx

      Console.Write("Main потік продовжується
                     після  завершення потоку tx");

   }// Main -------------------------------


   static void Печать() {

      Console.Write("Поток стартовал");
           .  .  .
      Thread.Sleep(100); // призупинення на 100 мсек
           .  .  .
   }

}// Thread_210
```

3.3. Потоки в бібліотеці OpenMP

Бібліотека OpenMP створена з метою уніфікації обчислень в СМП системах. Широко підтримується компанією Intel як інстру-

мент для створення паралельних програм для систем з багатоядерними процесорами [1, 3, 29].

Особливість побудови паралельної програми в OpenMP полягає в тому, що вона формується на основі послідовної програми шляхом визначення в неї частин, які будуть виконуватися паралельно. Це – реалізація концепції fork/join (Рис.3.1).

Тобто паралельна програма в OpenMP являє собою низку послідовних та паралельних частин (рис. 3.2). Для створення паралельної програми бібліотека надає ресурси у вигляді директив (прагм), змінних, процедур. Всі таки ресурси розпочинаються з префіксу `OMP`.

Виконання програми розпочинається з потоку *Майстер* - основного потоку, який виконує послідовні частини програми, а також створює нові протоки при входженні у паралельні частини програми. Кожен потік при створенні отримує ідентифікатор (унікальний номер). При цьому потік *Майстер* завжди має номер 0.

Потоки, що створюються в паралельних частинах, можуть виконувати однакові або різні дії, які можливо запрограмувати через використання ідентифікатору потоку. Основний потіr завжди чекає на завершення паралельної частини програми і тільки потім продовжує своє виконання.

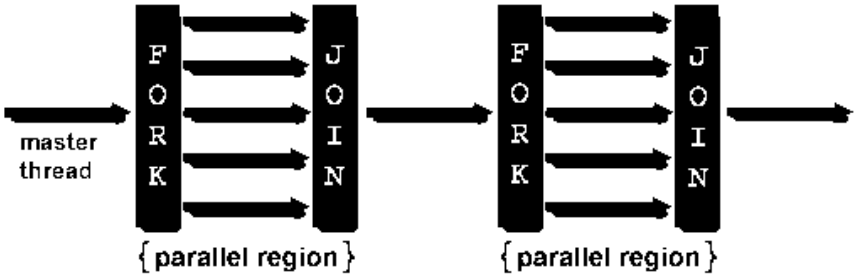


Рис. 3.1 Концепція Fork/Join

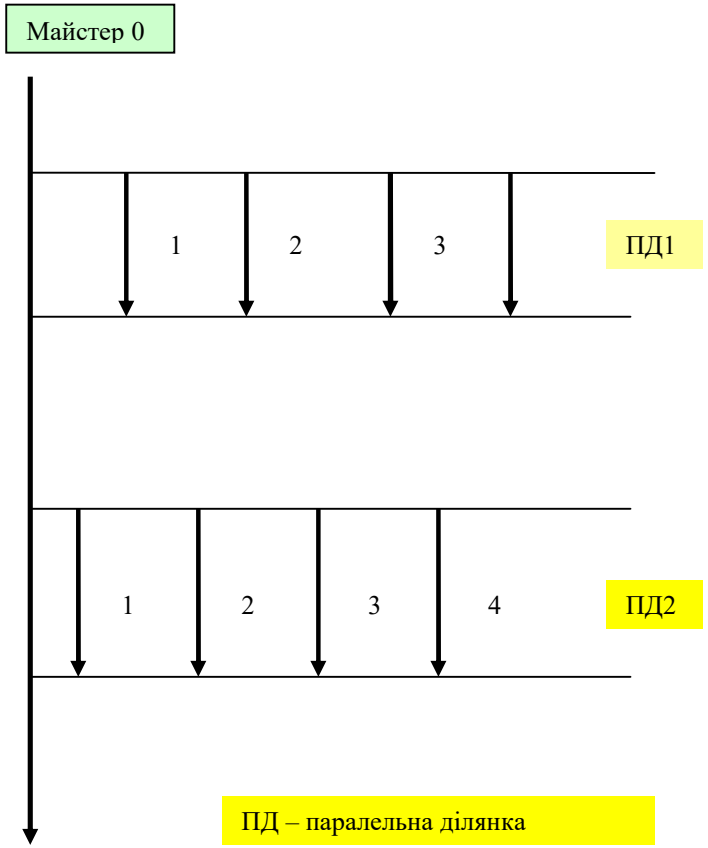


Рис 3.2 Структура програми в OpenMP

Основний потік породжує дочірні потоки. Програмування – шляхом вставки директив компілятора в потрібні місця вхідного коді послідовної програми.

Компілятор інтерпретує ці директиви і додає в програму бібліотечні ресурси для розпаралелювання ділянок коду.

Базовою є прагма `omp parallel` :

```
# pragma omp parallel  [ опції  ]  {

    . . .       тіло прагми

}
```

Прагма формує паралельну ділянку шляхом створення задану кількість *копій тіла*, які будуть виконуватися паралельно. По замовченню кількість копій визначається кількістю процесорів (ядер) в комп'ютерної системі.

Існує дві моделі виконання:

- динамічна, коли кількість потоків в програмі може змінюватися від одної паралельної ділянки до іншої
- статична, коли кількість потоків фиксована. Модель виконання контролюється за допомогою виклику функції `omp_set_dynamic()`.

Встановити потрібну кількість потоків в кожної паралельної ділянці можна через функцію `omp set_num_threads ()`

```
# omp  set_num_threads(2);
# pragma omp parallel {
          Calc();
  }
```

Потік (задача) може отримати свій ідентифікатор за допомогою методу `get_num_thread();`

```
int a, b, c;        //    глобальні змінні
#omp  set_num_threads (3);
# pragma omp parallel {
     int tid;        //    локальна змінна задачі
     tid:= get_num_thread();
```



```
     print(tid);
     a: = 100;
     b:=200;
     c:= 300;
 }
```

Дії потоків 1,2 і 3 – однакові. Можна зробити дії задач різними:

- через використання ідентифікаторів задач

```
int a, b, c;           //    глобальні змінні
#omp  set_num_threads(3);
#pragma omp parallel {
   int tid;   //    локальна змінна задачі
   tid:= get_num_thread();
   if(  tid=1 )    then    a: = 100;
   if ( tid=2 )    then    b: =200;
   if ( tid=3 )    then    c:= 300;
}
```

- через створення секцій

```
int a, b, c;               //    глобальні змінні
#omp  set_num_threads (3);
# pragma omp parallel {
  // тіло прагми
  # pragma omp sections{
  //  складається з окремих части  (секцій)
     # pragma omp section {a := 100; print(a) }
     # pragma omp section {b := 200; print(b) }
     # pragma omp section {c := 300; print(c) }
   }
}
```

Область бачення змінних в паралельної ділянці:

- `shared(var1, var2, ...)` змінні будуть спільними для задач.
- `private(var1, var2, ...)` змінні будуть локальними для задач, кожна задача буде працювати зі своєю копією

```
#pragma omp parallel shared(A), private(tid, x) {
    tid = omp_get_thread_num();
    x = work(tid);
```

```
    A[tid] = x;
}
```

Паралельне виконання циклів:

Прагма `for` дозволяє організувати паралельне виконання тіла циклу. Кожна задача виконує частину циклу.

```
#omp  set_num_threads(3);
# pragma omp parallel {
   # pragma omp  for
     for (int i = 0; i <120; i++)    {
          A[i] = B[i] + C[i];
     }
    }
```

3.4. Потоки в бібліотеці MPI

Бібліотека MPI наряду з PVM – найбільш поширений засіб програмування для комп'ютерних систем з розподіленою пам'яттю та розподілених комп'ютерних систем [7, 29].

Бібліотека (інтерфейс) MPI (Message Passing Interface) містить набір ресурсів (типів, функцій), що дозволять організувати роботу з потоками в мовах, які не мають вбудованих засобів програмування потоків. У разі застосування бібліотеки MPI використовується мова послідовного програмування типу С або Фортран, потоки в якій програмуються за допомогою засобів бібліотеки MPI.

Усі ресурси MPI бібліотеки (функції, типи, константи) розпочинаються з префікса `MPI_`.

Загальна структура MPI програми має вигляд:

```
#include "mpi.h"

/* Описання */
int main(int args, char * argv []){

  /* Локальні описання */

  MPI_Init(&args, &argv);

  /* Тіло програми */
```



```
  MPI_Finalize();
return 0;
}
```

Основні функції MPI бібліотеки, потрібні для створення потоків, наведено в таблиці 3.2:

Таблиця 3.2. Основні функції MPI

Функції	Дії
`MPI_Init()`	Підключити до MPI (параметри `args` і `argv` визначають аргументи програми). Завжди викликається першою
`MPI_Comm_size()`	Визначити кількість процесів, які потрібно запустити
`MPI_Comm_rank()`	Отримати ранг (номер) процесу в групі (в комунікаторі)
`MPI_Finalize()`	Завершити виконання програми

Потік у MPI має назву *задача.* Задачі в MPI можуть бути об'єднані в іменовану групу. Об'єкт - група дозволяє звертатися до групи як єдиного цілого (наприклад, відправляти повідомлення всім задачам групи), а також визначати дії, які виконуються тільки членами групи.

Функція

```
MPI_Group(MPI_Comm_World, &size);
```

створює групу.

У рамках групи всі задачі мають унікальні ідентифікатори (*ранки*) – впорядковані числа, які розпочинаються з нуля. Спочатку всі задачі належать до однієї базової групи, з якої потім формуються нові групи. MPI надає набір функцій для роботи з групами.

Створюючи в додатку задачі, їх прив'язують до спільної ділянки зв'язку. Для цього використовують поняття *комунікатор* – опис ділянки зв'язку потоків, які об'єднані в групу, а також у різні групи. Програма може вміщувати кілька ділянок зв'язку. Нумерація потоків всередині ділянки зв'язку незалежна. Комунікатор може

бути використаний як параметр MPI-функції і для обмеження сфери її дії тільки заданою ділянкою зв'язку. Крім того, комунікатор забезпечує вимоги безпеки. MPI автоматично створює комунікатор `MPI_Comm_World`, який є базовим для кожного додатка і створюється автоматично під час виклику функції `MPI_Init()`.

Функції

```
MPI_Comm_size(MPI_Comm_World, &size);
MPI_Comm_rank(MPI_Comm_World, &rank);
```

для комунікатора `MPI_Comm_World` повертають значення: `size` – розміру групи (кількість задач, що приєднані до ділянки зв'язку) і `rank` – порядковий номер задачі, яка викликає цю функцію.

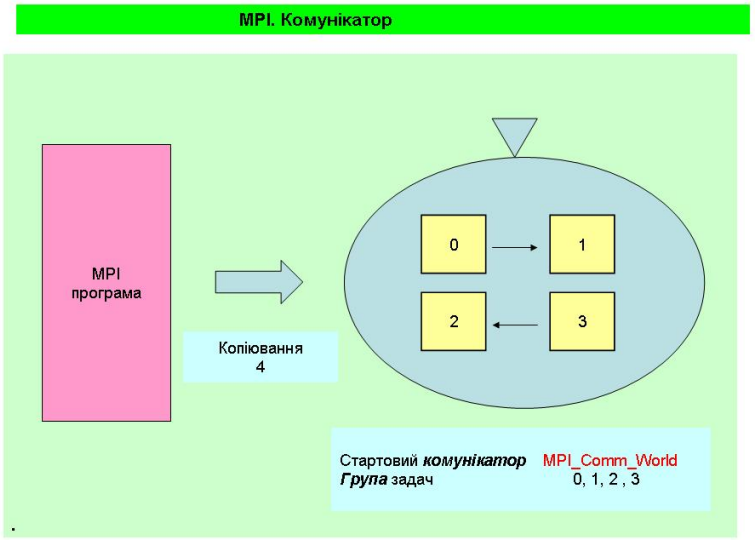


Рис. 3.3 MPI комунікатор

❖ Приклад 3.6

Створення потоків (задач), кожний з яких виводить на дисплей повідомлення про старт процесу. Кількість процесів задається при запуску програми. Використовується стандартний комунікатор `MPI_COMM_WORLD`.

```
/*-------------------------------------------------------------
```



```
-- MPI.Створення процесів                                    --
-------------------------------------------------------------*/
   #include <stdio.h>
   #include <string.h>
   #include <mpi.h>

   int main(int args, char *argv[]){

      int кількість_процесів;
      int мій_ранг;

      MPI_Status статус;

      /* Початок  роботи MPI */
      MPI_Init(&args, &argv);

      /* Отримати номер поточного процесу
                в групі в змінну   мій_ранг      */
      MPI_Comm_rank(MPI_COMM_WORLD, &мій_ранг);

      /* Отримати загальну кількість процесів,
        що старували  в змінну кількість_процесів */
      MPI_Comm_size(MPI_COMM_WORLD,
                    &кількість_процесів);

      /* Опис процесів програми */

      /* Виведення повідомлення
         на дисплей */
      if (мій_ранг != 0){
         sprintf("Start of process 0");
      }
      else{
         /* вивести на дисплей повідомлення з
            інших процесів */
         prinf("%s\n", мій_ранг);
          }
      }
      /* Завершення роботи  MPI */

      MPI_Finalize();
      return 0;
```

```
}/* main */
```

- **Запитання для модульного контролю**
  1. Які особливості створювання процесів в бібліотеці MPI ? Призначення рангу задачі? Що таке комунікатор ?
  2. Які функції WinAPI використовують для створювання потоків ?
  3. Засоби WinAPI для запуску потоків ?
  4. Засоби OpenMP для визначення паралельної ділянки коду?
  5. OpenMP. Функція встановлення кількості потоків ?
  6. OpenMP. Функція визначення ідентифікатору потоку ?
  7. OpenMP. Засоби організації паралельного виконання циклів ?
  8. C#. Створення потоку. Поняття потокової функції ?
  9. C#. Запуск потоку. Дії методу Start() ?
  10. MPI. Засоби створення комцнікатору?

- **Завдання для самостійної роботи**
  1. Розробити паралельну програму (бібліотека MPI), яка містить три потоки.
     За допомогою встановлення пріоритетів задач визначити для них жорсткий порядок виконання. Використати оператор затримки для перемикання з однієї задачі на іншу під час виконання.
  2. Завдання 1 реалізувати за допомогою засобів мови C#.
  3. Завдання 1 реалізувати за допомогою засобів бібліотеки OpenMP.
  4. Завдання 1 реалізувати за допомогою засобів бібліотеки MPI.
  5. В завданні 5 забезпечити динамічне створювання *P* задач та їх запуск на виконання. Значення *P* вводиться безносередне перед запуском програми.
  6. Завдання 5 реалізувати за допомогою мови Java, використовуючи конструктор класу, що є потоком.

  7. Виконати дослідження поведінки в Ада та Java програмах кількох процесів з різними пріоритетами, у яких застосовано оператор `delay` або метод `sleep()`.
  8. Створити Java програму з кількома потоками, у якій головна програма чекає на завершення потоків за допомогою методу `yield()`.

ЛЕКЦІЯ 4
Паралельні алгоритми

4.1 Паралельні алгоритми.
4.2 Паралельні алгоритми для задач лінійної алгебри

В лекції розглянуто елементи теорії паралельної математики, аналіз і побудова паралельних алгоритмів, поняття прискорення паралельного алгоритму (speedup), паралельні алгоритми для задач лінійної алгебри.

4.1. Аналіз паралельного алгоритму

Ефективна реалізація будь-якої задачі в паралельній системі можлива лише тоді, коли алгоритм її розв'язування поданий в паралельній формі, тобто буде розроблений паралельний алгоритм. З погляду паралельних властивостей (внутрішнього паралелізму) усі задачі можна поділити на три групи:

1) повністю паралельні задачі;
2) частково паралельні задачі;
3) задачі, які не допускають паралельної обробки.

Прикладом повністю паралельної задачі є операція додавання матриць, яку можна виконати за один крок, на якому здійснюється формування кожного елемента матриці.

Частково паралельною є задача, у якій чергуються паралельні та послідовні частини. Це найбільш поширений тип задач з погляду побудови паралельних алгоритмів. Приклад такої задачі − операція скалярного множення векторів.

Існують також задачі, для яких неможливо побудувати паралельний алгоритм. Наприклад, операція $a = b + c + d$ не може бути подано у вигляді паралельного алгоритму.

До появи паралельних комп'ютерних систем використовувались виключно послідовні алгоритми, орієнтовані спочатку на людину, а потім на одно процесорні комп'ютери. Проблема розроблення паралельних алгоритмів з'явилась з появою реальних паралельних систем. "Паралельна математика", яка вивчала методи і засоби побудови та аналізу паралельних алгоритмів бурхливо розвивалась у 60−70-х роках минулого століття . Науковці отримали цікаві результати про паралельні властивості цілого ряду задач. При цьому для окремих задач довелося відмовитися від традиційних підходів і розробити цілком нові, які дозволили побудувати ефективні паралельні алгоритми. З'ясувалося, що оцінка ефективності різних паралельних алгоритмів − важке завдання, тому що для цього треба враховувати різні додаткові фактори, наприклад, час передачі даних між процесорами [2].

Для спрощення аналізу паралельних алгоритмів було запропоновано *концепцію необмеженого паралелізму*, яка ґрунтувалась на таких поняттях :

- кожна операція виконується за одиницю часу;
- час передавання даних між процесорами системи не враховується;
- застосовується будь-яка потрібна (необмежена) кількість процесорів.

Зрозуміло, що концепція необмеженого паралелізму не відповідає дійсності щодо часу виконання паралельної програми в реальних комп'ютерних системах, оскільки не враховується більшість додаткових параметрів цієї системи, але вона дозволяє ще на етапі проектування паралельних алгоритмів оцінити для них час виконання і вибрати оптимальний алгоритм.

Концепція необмеженого паралелізму дозволяє спростити розрахунок *коефіцієнта прискорення* ($Kп$), який показує скорочення часу виконання паралельної програми в паралельній системі з P процесорами (Tp) в порівнянні з часом виконання послідовної програми в однопроцесорній системі (T_1):

$$Kп = T_1 / Tp$$

Поряд з коефіцієнтом прискорення використовують поняття коефіцієнта *ефективності* застосування комп'ютерної системи (Ke), який показує ступінь використання P процесорів системи:

$$Ke = T1 / (Tp * P)$$

Для поєднання необмеженого паралелізму в концепції необмеженого паралелізму з обмеженою кількістю процесорів можна використати лему Брента [2]:

Лема Брента. Якщо час розв'язання задачі, що включає n операцій в паралельній системі з необмеженою кількістю процесорів дорівнює t, то час tp виконання цієї задачі в системі з p процесорів буде не більше, ніж

$$tp = t + (n - t)/p .$$

Теорема, що наведена нижче, дозволяє оцінити час виконання бінарних операцій, алгоритм виконання яких можна подати бінарним деревом.

Теорема Мунро – Петерсена [4]. Якщо в комп'ютерній системі з p процесорів виконується обчислення скалярної величини, яке потребує m бінарних операцій, то необхідний час tp :

$$tp >= \begin{cases}]\log p[+] (m + 1 - 2^]\log p[/p [& \text{якщо } m >= 2^[\log p] \\ ] \log (m+1) [& \text{в інших випадках,} \end{cases}$$

де $]x[$ – найменше ціле число, більше, або таке, що, дорівнює x; логарифм береться за основою 2; ^ – означає ступінь.

За допомогою теореми Мунро–Петерсона можна оцінити час виконання операції множення матриць MA = MB * MC розміром $N*N$. Ця операція потребує N множень і N–1 додатків. Тобто $m = 2N - 1$. Звіди

$$\log (m+1) = \log (2N) = 1 + \log N$$

і на підставі теореми Мунро–Петерсона отримуємо

$$tp >= \begin{cases}] \log p [+] 2N - 2^] \log p[/p [& \text{якщо } 2N-1 >= 2^]\log p[\\ ] \log N [+ 1 & \text{в інших випадках.} \end{cases}$$

Якщо кількість процесорів необмежена, то оптимальний час для операції дорівнює] log N [+ 1. При цьому на першому кроці виконуються паралельно всі множення ($N * N * N$), а потім за] log N [кроків – паралельні додавання. Для $N = 100$ час дорівнює 8 одиницям, при цьому знадобиться 1000 000 процесорів.

Ярусно-паралельна форма алгоритму. Подання паралельного алгоритму може бути виконано різними способами [7]. *Ярусно-паралельна форма* (ЯПФ) для паралельного алгоритму являє собою набір ярусів, у яких операції виконуються паралельно (рис. 4.1). Висота ЯПФ (H) – кількість ярусів, ширина i-го ярусу (Ri) – кількість операцій в ярусі, ширина ЯПФ (R) – найбільша з ширини ярусів. Виконання ЯПФ здійснюється за ярусами. Тому час виконання задачі, алгоритм який представлений ЯПФ, дорівнює висоті ЯПФ, а максимальна потрібна кількість процесорів – ширині ЯПФ.

Одна й та сама задача може мати декілька паралельних алгоритмів, і відповідно, декілька ЯПФ, що відрізняються як висотою, так і шириною. Застосовуючи ЯПФ, можна вибрати алгоритм з необхідним часом виконання або із заданою кількістю процесорів.

Для ЯПФ, показаній на рис. 2.2, параметри дорівнюють: $H = 4$, $R1 = 4$, $R2 = 2$, $R3 = 3$, $R4 = 2$, $R = \max(4, 2, 3, 2) = 4$.

Знання цих параметрів дозволяє розрахувати:

- час розв'язування задачі в однопроцесорній системі:

$$T1 = R1 + R2 + R3 + R4 = 11;$$

- час розв'язування задачі в багатопроцесорній системі:

$$Tp = H = 4 ;$$

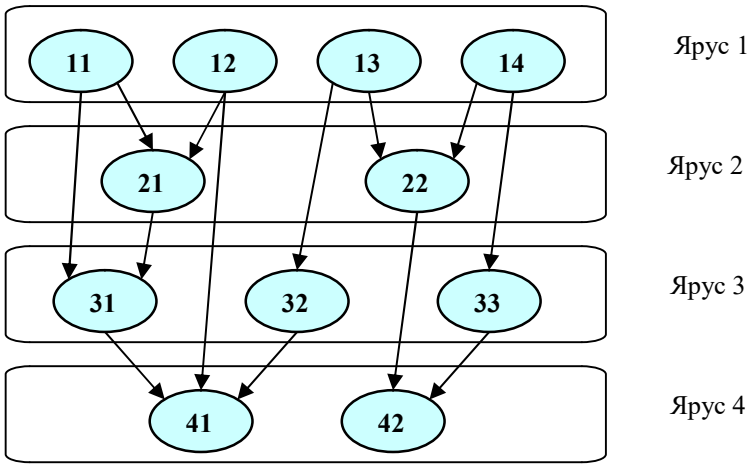


Рис. 4.1. Ярусно-паралельна форма алгоритму

- кількість процесорів, потрібних на кожному кроку обчислення ЯПФ:

$$P_1 = 4, \quad P_2 = 2, \quad P_3 = 3, \quad P_4 = 2 ;$$

- потрібну кількість процесорів, яка дозволяє розв'язати задачу за мінімальний час:

$$P = R = 4 ;$$

- коефіцієнт прискорення:

$$K_n = T_1/T_p = 11/4 = 2{,}75;$$

- коефіцієнт ефективності:

$$K_e = K_n/P = 2{,}75/4 = 0{,}69 \quad (69\ \%).$$

4.2 Паралельні алгоритми для задач лінійної алгебри.

Розглянемо й проаналізуємо паралельні алгоритми для задач лінійної алгебри із застосуванням концепції необмеженого паралелізму.

- **Додавання векторів.** Операція додавання векторів розміру N

$$A = B + C$$

виконується за співвідношенням $a_i = b_i + c_i$, де $i = 1..N$.

Час виконання задачі в одно процесорній комп'ютерної системі дорівнює $T_1 = N$, час виконання в паралельній системі дорівнює $T_p = 1$ для $P = N$, коефіцієнт прискорення $K_n = P$, тобто це задача має 100 відсотковий внутрішній паралелізм й ідеальна для виконання в паралельній системі.

Якщо кількість процесорів обмежена $P < N$, то паралельний алгоритм можна показати у вигляді

$$A_n = B_n + C_n,$$

де $H = N/P$; $A_n - H$ елементів вектора А.

- **Скалярне множення векторів.** Операція скалярного множення векторів

$$a = (B * C)$$

виконується за співвідношенням

$$a = b_1 * c_1 + \ldots + b_i * c_i + \ldots + b_n * c_n \qquad i = 1 .. N.$$

Час виконання задачі в однопроцесорній КС дорівнює $T_1 = N + (N - 1)$.

Операцію скалярного множення векторів відносять до групи операцій, для яких побудова паралельних алгоритмів базується на *здвоюванні* [6]. Для подібних схем час виконання має логарифмічну залежність. Час виконання скалярного множення в паралельній

КС складається із виконання множення $b_i * c_i$, яке можна здійснити за один крок при $P = N$, і з наступного додавання результатів множення. Тут використовується здвоювання, що дозволяє виконати множення за час, що дорівнює]log$N$[.

Таким чином, час виконання операції скалярного множення $Tp = 1 +]\log N[$ для $P = N$, коефіцієнт прискорення $Kn = (2*N - 1)/(1 +]\log N[)$, тобто це частково паралельна задача, у якій чергуються паралельні і послідовні частини.

Якщо кількість процесорів P обмежена і значно менша за N, то паралельний алгоритм можна показати у вигляді

1 $a_i = (B_н * C_н)$, $i = 1 .. P.$
2 $a = a + a_i$

- **Множення вектора на матрицю.** Операція множення вектора на матрицю

$$A = B * MC$$

виконується за співвідношенням

$$a_i = b1 * c_{1i} + . . + bi * c_{ii} + . . + bn * c_{ni}, \qquad i = 1 .. N.$$

Час виконання задачі в однопроцесорній КС $T1 = N*$ (N+ (N – 1)). В паралельному алгоритмі використано схему здвоювання, тому час виконання в паралельній КС $Tp = 1 +]\log_2 N[$ якщо $P = N*N$, і коефіцієнт прискорення дорівнює $Kn = (2N*(N+(N – 1))/(1 + + \log_2 N)$, тобто це частково паралельна задача, в якій чергуються паралельні і послідовні частини.

Якщо кількість процесорів обмежена і $P < N$, то паралельний алгоритм можна подати у вигляді

$$A_н = B * MC_н,$$

де $MC_н$ - H стовпців матриці MC.

- **Додавання матриць.** Операцію додавання матриць

$$MA = MB + MC$$

виконується за співвідношенням

$$a_{ij} = b_{ij} + c_{ij} , \qquad \text{де } i,j = 1.. N.$$

Час виконання задачі в однопроцесорній КС дорівнює $T1 = N*N$, час виконання в паралельній КС дорівнює $Tp = 1$ для $P = N*N$, коефіцієнт прискорення $Kn = P$, тобто це теж ідеальна для виконання в паралельній КС задача, але слід звернути увагу на те, як збільшилася кількість процесорів в КС, що потрібно для розв'язання задачі за одиницю часу.

Якщо кількість процесорів обмежена $P << N$, то паралельний алгоритм можна подати у вигляді

$$MA_н = MB_н + MC_н .$$

- **Множення матриць.** Операція множення матриць

$$MA = MB * MC$$

виконується за співвідношенням

$$a_{ij} = a_{ij} + b_{ik} * c_{kj} , \qquad \text{де } i,j,k = 1 .. N.$$

Час виконання задачі в однопроцесорній КС $T1 = N*N*(N + (N – 1))$, час виконання в паралельній КС $Tp = 1 +]\log N[$ якщо $P = N*N*N$, коефіцієнт прискорення $Kn = N*N*(N + (N – 1)) / (1 +]\log N[ )$.

Існує декілька паралельних алгоритмів множення матриць.

Якщо кількість потоків обмежена і значно менша за N, паралельний алгоритм множення матриць можна подати у вигляді

$$MA_н = MB * MC_н.$$

Кожен потік формує свою частину результату *МАн* (*Н* рядків), використовуючи повну матрицю *МВ* і частину матриці *МСн* (*Н* стовпців).

- **Сортування векторів**

$$A = sort(B)$$

Паралельній алгоритм сортування може базуватися на розбитті вектору на частини *Вн* і виконанні в кожному потоці (наприклад, чотирьох потоків Т1-Т4) локального сортування підвектру *Вн*. Потім потоки Т1 і Т3 виконують наступне сортування відсортованих частин *Вн* через спеціальний від сортування (*merge sort* - сортування через "злиття") , отримуючи в кожному потоці вже дві частини вектору *В2н*. На сам кінець остаточний результат формує потік Т1, який здійснює сортування через злиття вже відсортованих частин *В2н* (https://www.youtube.com/watch?v=jEw0rrckjnY).

1 Вн = sort(Вн) Т1 ... Т4
2 В2н = msort(Вн, Вн) Т1, Т3
3 А = msort(В2н, В2н) Т1

- **Пошук максимального значення елементів вектора**

$$a = max(B)$$

Паралельний алгоритм пошуку максимального значення елементів вектора буде складатися з двох послідовних кроків, де на першому кроці кожен потік шукає локальний максимум, а на другому - формується глобальний максимум, і може мати наступний вигляд:

1 ai = max(Вн) i=1 .. P
2 a = max(a, ai)

- **Паралельне обчислення складних векторно-матричних виразів**

$$A = x* C + (B*MD)*MR + (F*G)*Z \qquad (4.1)$$

Складні векторно-матричні вирази формуються з елементарних операцій над векторами і матрицями (додавання, множення, сортування, пошуку мінімального (максимального) елемента). Тому побудова паралельного алгоритму для складного виразу буде-ється на використанні паралельних алгоритмів для цих елементарних операцій.

Паралельний алгоритм для обчислення виразу (4.1) буде складатися з декількох послідовних кроків, де кожен крок описує паралельні обчислення в кожному потоці) і може мати наступний вигляд:

1 ei = (Fн Gн) i = 1 .. P*
2 e = e + ei
*3 R н = B*MDн*
4 Ан = x Сн + R*MRн + e*Zн*
Спільні ресурси(СР): e (крок 2)

Слід звернути увагу на те, що в наведеному паралельному алгоритму в потоках буде потрібна *додаткова* синхронізація між кроками 3 і 4 по формуванні повного вектора *R* для використання його на кроці 4. Це синхронізацію треба додати в алгоритми потоків (Етап2 розробки програми) наряду з синхронізацією по введенню даних і виведенню результату.

Розробка паралельного алгоритму

Розробка алгоритму розв'язування завдання завжди була важливим і найбільш складним етапом розроблення програми. Для паралельного алгоритму цей етап має свої особливості і складається з окремих кроків алгоритми : *декомпозиції, проектування взаємодії задач, об'єднання, планування обчислень.*

Етапи розроблення паралельного алгоритму показано на рис. 4.3.

1. **Декомпозиція.** Крок, на якому вхідне завдання аналізується і розкладається на частини – підзадачі, виконання яких може бути паралельним (на рис. 4.3 це підзадачі *A .. P*). Існують різні види декомпозиції : *декомпозиція за даними, функціональна декомпозиція, об'єктна декомпозиція* та ін. Кількість підзадач має перевищувати кількість процесорів КС, що дозволить оптимізувати наступні етапи розроблення алгоритму. При цьому також слід мінімізувати кількість обчислень, а також обсяг інформації, що пересилається між під задачами. Наступне планування обчислень (завантаження процесорів) буде спрощеним, якщо під задачі мають однаковий розмір. Щодо розміру під задач, то він пов'язаний з поняттям «зерна» алгоритму, яке, в свою чергу, пов'язане з рівнем паралелізму і типом використовуваної паралельної системи. Дрібнозернистий паралелізм – це рівень команд, середньозернистий – рівень циклів, процедур і модулів (класів), великозернистий – рівень програм.

   Декомпозиція – найважливіший крок розроблення алгоритму, який впливає на всі інші етапи. Помилки в декомпозиції мають найважчі наслідки для всієї програми в цілому, як за її коректністю, так і за часом її виконання. Тому слід приділяти особливу увагу виконанню декомпозиції, розглядати кількість можливих її варіантів, з яких після попереднього порівняльного аналізу (наприклад, за допомогою концепції необмеженого паралелізму) вибрати оптимальний.

2. **Проектування взаємодії під задач.** Визначаються дії і методи, які потрібні для організації взаємодії під задач (передавання даних, синхронізація, доступ до спільних даних та ін.) Види взаємодії можуть бути *локальними або глобальними* залежно від того, яку кількість зв'язків з іншими має кожна під задача; *синхронними або асинхронними* залежно від виду виконання обміну даних; *статичними або динамічними,* якщо вони можуть змінюватися в процесі виконання програми.

   Організація взаємодії задач тісно пов'язана з типом архітектури КС, що використається. Для систем, де використовується взаємодія, що ґрунтується на моделі посилання повідомлень

(КС з локальною пам'яттю), оптимальність цієї організації буде значною мірою визначати час виконання програми, оскільки в таких системах час передавання повідомлень можна порівнювати з часом безпосередніх обчислень і він буде суттєво впливати на загальний час роботи програми. Скорочення часу на взаємодію процесів можна досягнути, якщо зменшити загальну кількість взаємодій, а також обсяг даних, що передаються, і, по можливості, виконувати їх одночасно (паралельно).

3. **Об'єднання.** Виконується об'єднання окремих підзадач у більші задачі з метою зменшення обсягів даних, які потрібно передавати між задачами. Кінцева мета об'єднання – збільшення ефективності алгоритму за рахунок скорочення часу його виконання і складності реалізації.

   Результат кроку об'єднання для рис. 4.3 – формування трьох задач *T1* (підзадачі *A, B, G, H*), *T2* (підзадачі *C, K*) і *T3* (підзадачі *D, E, F, L, M, P*).

4. **Планування обчислень.** Виконується розподіл задач по процесорах КС. Оптимальне розміщення дозволить мінімізувати реальні затрати часу, які будуть пов'язані з виконанням обчислень і обміном даних між процесорами системи На рис. 4.3 задача *T*1 буде виконуватися на процесорі 3, задача *T*2 – на процесорі 1, задача *T*3 – на процесорі 2. Планування обчислень (task scheduling) є важливою складовою процесу розроблення та виконання програми і на нього треба звертати окрему увагу. Вирішення проблеми планування залежить від виду планування (динамічне або статичне) і пов'язане з використанням спеціальних різноманітних алгоритмів планування.

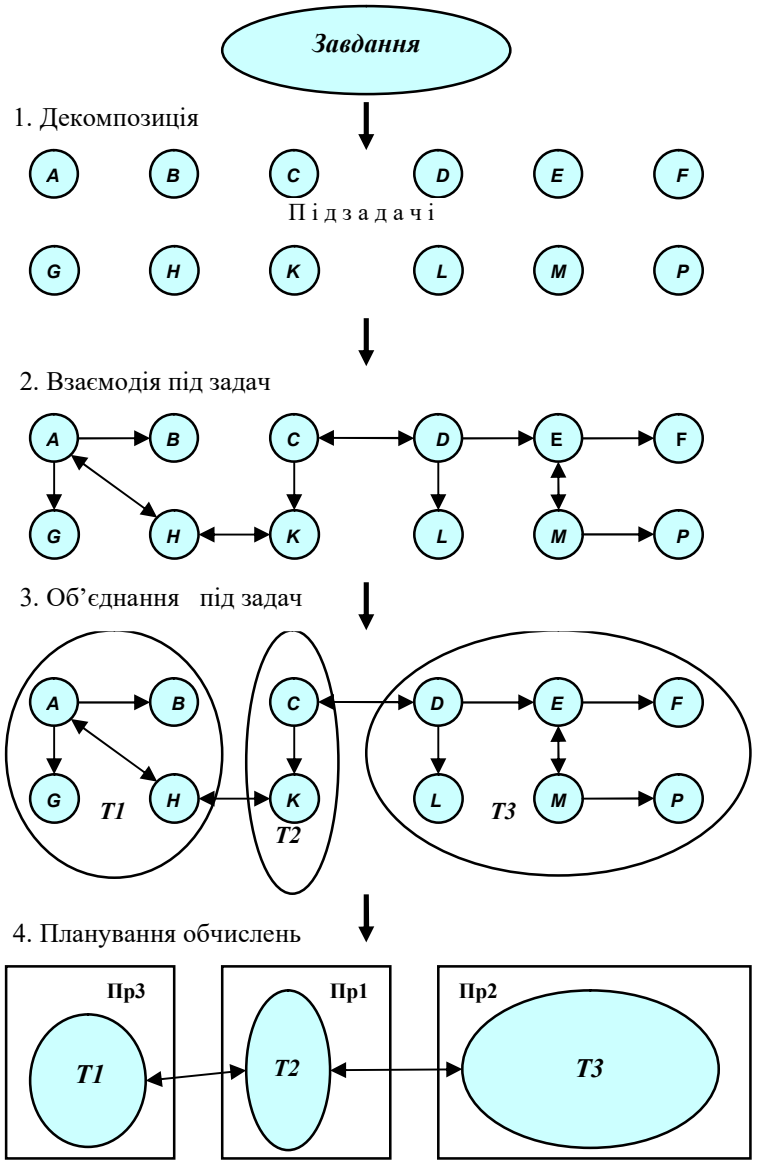


Рис. 4.3. Етапи розробки паралельного алгоритму

- **Запитання для модульного контролю**
  1. Класифікація задач за внутрішним паралелізмом
  2. Що являють собою основні положення концепції необмеженого паралелізму ?
  3 Що пов'язує лема Брента ?
  4. Як обчислюється коефіцієнт прискорення ? Що він показує?
  5. Для яких задач працює теорема Мунро-Петерсона?

- **Завдання для самостійної роботи**

1. Розробити паралельний алгоритм для виконання операції знаходження максимального елемента матриці.
2. Розробити паралельний алгоритм сортування вектора.
3. Розробити паралельний алгоритм для векторно-матричної операції A = d* (D*MC) + E*MR.
4. Розробити паралельний алгоритм для векторно-матричної операції MA = d* (MD*MC) + (ME*MR)
5. Розробити паралельний алгоритм для векторно-матричної операції A = d* (D*MC) + E*MR
6. Розробити паралельний алгоритм для векторно-матричної операції A = max(B)* D* + E*(MR*MX)
7. Розробити паралельний алгоритм для векторно-матричної операції A = (B*C)* (D*MC) + (X*ME)*g
8. Розробити паралельний алгоритм для векторно-матричної операції A = d* (D*MC) + Q*MR
10. Розробити паралельний алгоритм для виконання операції знаходження максимального елемента матриці.
11. Розробити паралельний алгоритм сортування вектору.

ЛЕКЦІЯ 5
Взаємодія потоків

5.1. Моделі взаємодії потоків

Взаємодія потоків пов'язана з двома основними завданнями:

- комунікацією потоків;
- синхронізацією потоків [18].

Комунікація потоків передбачає обмін даними між потоками. При цьому інформація передається від одного потоку до іншого в будь-якому напряму.

Синхронізація включає узгодження поведінки потоків і залежність виконання одного потоку від *подій*, що можуть бути в іншому потоці.

Механізм реалізації взаємодії потоків залежить від виду комп'ютерної системи і в загальному випадку ґрунтується або на використанні

- моделі *спільних змінних* (системи зі спільною пам'яттю),
- моделі *посилання повідомлень* (системи з локальною пам'яттю).

У системах зі спільною пам'яттю взаємодія потоків виконується за допомогою спільної пам'яті. Комунікація процесів і синхронізація процесів виконується через *спільні змінні*. Для передавання даних процес записує ці дані в спільні змінні, звідки їх зчитує інший процес. Синхронізація виконується за допомогою відповідних спільних змінних, які змінюються процесом, у якому видбувається подія, і зчитуються (аналізуються) процесом, що чекає на подію.

Використання спільних змінних, що доступні будь-якому процесу (це обов'язково глобальні змінні), приводить до того, що в

програмі може відбуватися одночасне звертання процесів до спільних змінних, що призводить до конфлікту процесів або некоректної роботи програми. Тому програмування з використанням моделі спільних змінних пов'язано з необхідністю вирішення специфічних проблем, які загалом зводяться до розв'язання двох основних завдань – *взаємного виключення* і *синхронізації процесів.*

5.2. Завдання взаємного виключення

Завдання взаємного виключення полягає в тому, що під час виконання двох і більше паралельних процесів може виникнути одночасне звернення процесів до одного і того ж *спільного ресурсу* (СР). Таке звернення зазвичай призводить до конфлікту процесів, що полягає або в різкому уповільненні роботи програми, або в некоректній її роботі, якщо, наприклад, один процес зчитає дані, а другий їх змінює в цей же час, або (в крайньому випадку) – до аварійного завершення програми.

Загальна схема вирішення завдання взаємного виключення ґрунтується на тому, що потрібно *призупинити* (*блокувати*) процес, який звертається до спільного ресурсу, який вже використовується в цей момент іншим процесом. *Розблокування* процесу має бути виконано одразу після звільнення спільного ресурсу.

Існують *два підходи* до вирішення завдання взаємного виключення. Перший підхід ґрунтується на *контролі процесів* і пов'язаний з виявленням у процесах ділянок, у яких вони звертаються до спільного ресурсу. Такі ділянки процесів отримали назву *критичних ділянок (КД)*. Для вирішення завдання взаємного виключення необхідно *не допустити* одночасного входження процесів у свої критичні ділянки. Якщо один процес уже знаходиться в критичній ділянці, то будь-який інший процес за намагання входження в свою критичну ділянку має бути блокований доти, доки перший процес не вийде зі своєї критичної ділянки.

Класична схема організації такого контролю потребує використання операцій (примітивів) **`ВХІДКД`** і **`ВИХІДКД`**, що розміщуються відповідно перед і після критичної ділянки, тобто обрамляють критичну ділянку, створюючи "огорожу" навколо неї (рис. 5.1).

Алгоритм виконання операції ***ВХІДКД*** :

1. Перевірити, чи знаходиться будь-який інший процес у своїй критичний ділянці ?
2. Якщо знаходиться, то блокувати процес, що виконує операцію *ВХІДКД*.
3. Якщо не знаходиться, то встановити заборону на входження всіх процесів у свої критичної ділянки і дозволити цьому процесу увійти в його критичну ділянку.

Алгоритм виконання операції ***ВИХІДКД***:

1. Зняти заборону на входи процесів в їх критичні ділянки.

Конкретна реалізація примітивів *ВХІДКД* і *ВИХІКД* у мовах та бібліотеках програмування виконана у вигляді механізмів семафорів, мютексів, критичних секцій.

Другий підхід до вирішення завдання взаємного виключення передбачає контроль безпосередньо за спільним ресурсом (рис. 5.2). Такий контроль може бути здійснений за допомогою оголошення належних змінних спільними ресурсами, а також за допомогою моніторів. З цією метою створюється програмний модуль (монітор), що містить в собі спільний ресурс, а також набір процедур для доступу до спільного ресурсу. Тепер доступ до спільного ресурсу з процесів можливий тільки через виклик потрібної процедури монітора.

Процедури монітора мають важливу властивість – вони *взаємно виключають* один одного. Тобто, якщо процес викликав і виконує будь-яку процедуру монітора, то виклик іншим процесом будь- якої процедури цього монітора приведе до блокування цього процесу до того часу, поки не закінчиться виконання вже розпочатої процедури монітора. Таким чином, монітор дозволяє виконання тільки однієї процедури. Якщо ввести термін «процес знаходиться в моніторі», під яким розуміти, що процес виконує будь-яку процедуру монітора, то можна стверджувати, що в моніторі може знаходитись тільки один процес.

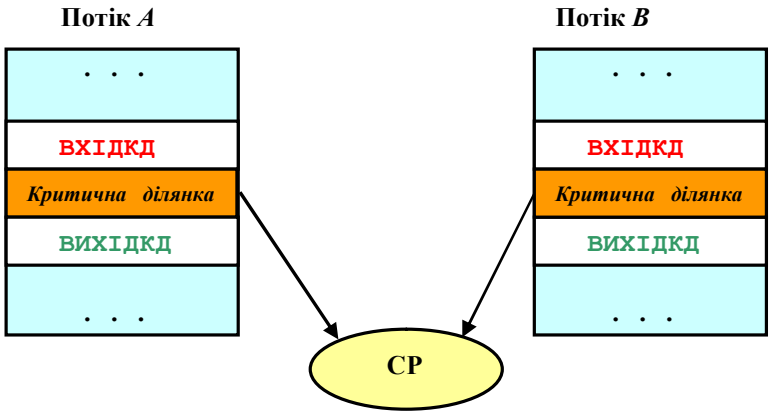


Рис. 5.1. Загальна схема вирішення завдання взаємного виключення, яка ґрунтується на *контролі потоків*

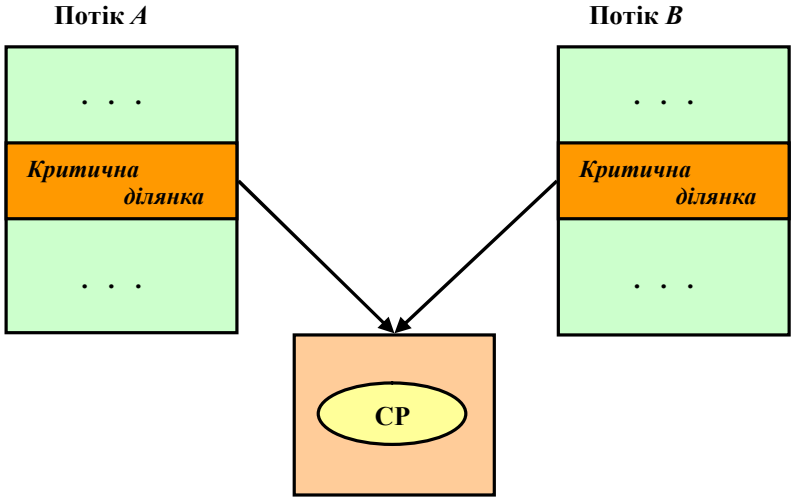


Рис. 5.2. Загальна схема вирішення завдання взаємного виключення, яка ґрунтується на *контролі спільного ресурсу*

Механізм моніторів реалізується різними способами: у мові Ada – за допомогою захищених модулів, в мові Java – за допомогою синхронізованих методів.

5.3. Завдання синхронізації процесів

Завдання синхронізації двох процесів полягає в тому, що в одному процесі, наприклад *B,* у визначеній точці (*точці події*) виконується подія (обчислення даних, уведення або виведення даних і т.ін.), а другий процес *A* у визначеній точці (*точці очікування події*) блокується доти, доки ця подія не відбудеться і він зможе продовжити своє виконання. Точка події і точка очікування – це *точки синхронізації.* Якщо процес *A* вийшов на точку синхронізації, коли подія вже відбулася, то він не блокується і продовжує виконуватись.

Існує декілька схем синхронізації процесів (рис. 5.3):

- один процес очікує на подію в одному процесі;
- один процес очікує на події в кількох процесах;
- декілька процесів очікують на подію в одному процесі.

Точки ***S*** і ***W*** на рис. 5.3 – це точки синхронізації процесів; *S* означає точку посилання сигналу про подію, *W* –точку очікування сигналу про подію.

Для вирішення завдання синхронізації, яке іноді називають *синхронізацією за подіями (event synchronization)*, можна використовувати різні механізми синхронізації потоків, таки як семафори, події, монітори.

Універсальним засобом рішення завдання синхронізації є ***механізм бар'єрів.*** Реалізований в мовах Ada, Java, C#, бібліотеках OpenMP, WinAPI.

Для групи потоків у кожному з них визначається точка синхронізація (бар'єр). По досягненню бар'єра потік блокується і чекає на досягнення цього бар'єру у всіх інших (або вказаної кількості) потоках групи, після чого потік розблокуються і продовжують своє виконання. В мові Ada для реалізації механізму бар'єрів запропоновано пакет:

```
package Ada.Synchronous_Barriers is

   pragma Preelaborate(Synchronous_Barriers);

   subtype Barrier_Limit is Positive range 1 ..
                                        implementation_defined;
```



```
   type Synchronous_Barrier (Release_Threshold :
                        Barrier_Limit) is limited private;

   procedure Wait_For_Release (The_Barrier : in out
            Synchronous_Barrier; Notified :  out Boolean );

private

   -- not specified by the language

end Ada.Synchronous_Barriers;
```

Приклад 5.1

```
   NT : integer constant:= 4;

   Bar : Synchronous_Barrier(NT);
   X : Boolean;

   task type WT;

   task body WT is
   begin
      loop
               .  .  .   -- дії до бар'єру

         -- бар'єр
         Wait_For_Release (Bar, X);

               .  .  .
       end loop;
   end WT;
```

5.4. Атомарні змінні

Концепція *поділяємих* (або *атомік*) змінних являє собою простіший механізм (з точки зору користувача), який забезпечує вирішення завдання взаємного виключення [2 , 23].

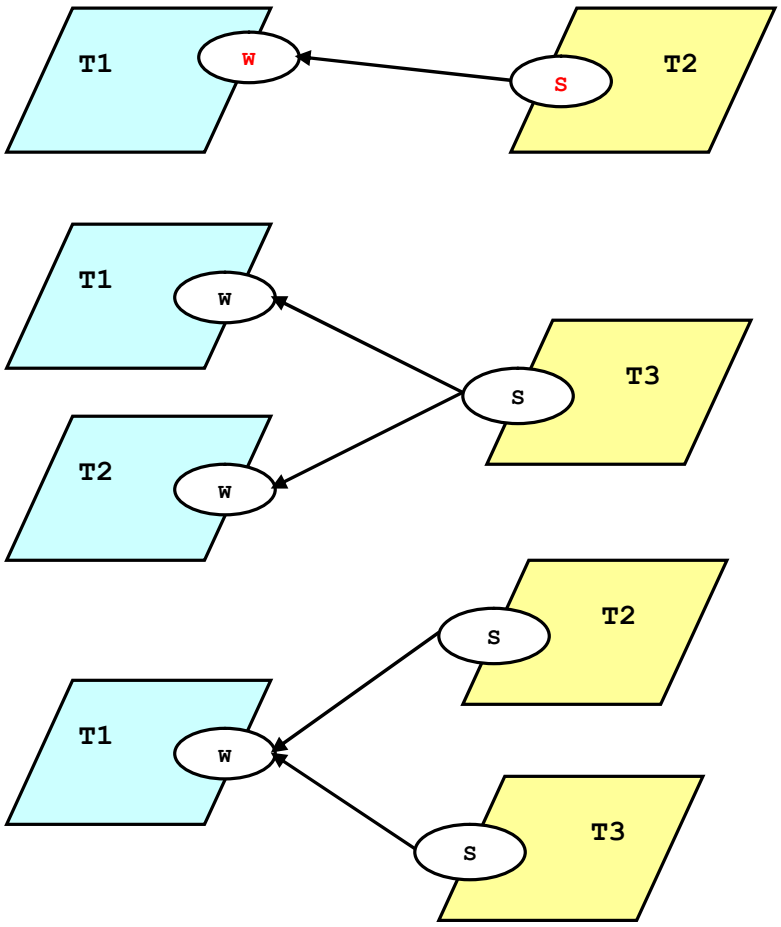


Рис. 5.3. Види синхронізації процесів

Механізм ґрунтується на можливості визначення і опису в програмі спільних ресурсів, після чого контроль за їх використанням бере на себе система керування потоками, яка автоматично вирішує завдання взаємного виключення стосовно цих ресурсів. Кожне звернення до поділяємої змінної розглядається як корична ділянка, вхід і вихід з якої контролюється тепер системою, яка управляє процесами.

Приклад 5.2 ілюструє концепцію поділяємих змінних. Для опису спільних ресурсів використовується конструкція типу `Shared_Variable`.

Концепцію поділяємих змінних реалізовано в мовах Ада, C#, Java, бібліотеках OpenMP.

Приклад 5.2

```
procedure Lab51 is

   Buf: integer; -- спільний ресурс

   -- опис атомарної (поділяємої) змінної
   Shared_Variable(Buf);

   task T1;
   task body T1 is
   begin
                        . . .

      Buf := Buf*4 +100;    --  критична ділянка в T1

                        . . .
   end T1;
  ------------------------------------------------------
   task T2;
   task body T2 is
   begin
                        . . .

      Buf := Buf + 200;       --  критична ділянка в T2
                        . . .
   end T2;
begin
         . . .
end Lab51;
```

Мова Ada. Механізм поділяємих змінних в мові Ada реалізований за допомогою атомарних (atomic) змінних. Оголошення атомарних змінних виконується з використанням прагм `Atomic` і `Volatile` [25].

Прагма `Atomic` дозволяє описати в якості поділяємої змінну простого типу. Поділяємі змінні складаного типу (масиви, записи) описуються за допомогою прагми `Atomic_Component`.

```
Pragma Atomic( )
Pragma Atomic_Component( )
```

Прагми `Volatile` і `Volatile_Component` схожі на `Atomic` прагми. Відмінність полягає в тому, що дії з поділяємими змінним мі виконуються безпосереднє в пам'яті, без використання регістрів та кеш-пам'яті.

```
Pragma Volatile( )
Pragma Volatile_Component( )
```

Приклад 5.2

```
procedure Lab52 is

   CP1:integer;  -- спільний ресурс1
   CP2: array (1..10) of float; --спільний ресурс 2

   -- опис атомік змінних
   pragma Atomic(CP1);
   pragma Atomic_Component(CP2);

   task T1;
   task body T1 is
   begin
                        .  .  .
      CP1 := CP1*4 +100; --  критична ділянка1 в T1
                        .  .  .
      OP2(2):= OP2(5) +123.33; критична ділянка2 в T1
                       .  .  .
   end T1;
   ------------------------------------------------
```



```
   task T2;
   task body T2 is
   begin
                        .  .  .
     CP1 := CP1  + 200; --  критична ділянка1 в T2

                        .  .  .

     CP2(4):= CP2(2)+2.55; -- критична ділянка2 в T2
                       .  .  .

    end T2;
begin
       null;
end Lab52;
```

Мова Java. Основні ресурси мови Java для створення багатопотокових додатків знаходяться в бібліотеці `JavaConcurrent` пакету `java.util.concurrent` [23]**:**

- `Concurrent Collections` – набір ресурсі для роботи у багатопотоковому середовищі. Замість базового `Collections.synchronizedList` з блокуванням доступу використовуються блокування сегментів даних або wait-free алгоритми для паралельного читання даних.
- `Queues` – неблокуючі та блокуючі черги. Неблокуючі черги зосереджені на швидкості та роботі без блокування потоків..
- `Synchronizers` – допоміжні утиліти для синхронізації потоків.
- `Executors` – фреймворк для створення пулів потоків
- `Locks` –додаткові механізми синхронізації потоків (до базових `synchronized, wait, notify, notifyAll`).
- `Atomics` – класи, які підтримують атомарні операції над примітивами та посиланнями.

В мові Java механізм поділяємих змінних реалізований через модифікатор `volatile`. Для змінної, яка описана через модифікатор `volatile` вводяться особливі правіла виконання операцій запис/читання. При цьому операції запис/читання виконуються безпосереднє в оперативної пам'яті без використання регістрів та кеш-пам'яті. Запис в змінну з модифікатором `volatile` повинен

закінчитися раніше, ніж розпочнетеся виконуватися наступне читання цієї змінної.

Змінна позначена, як volatile, доступна глобально. Якщо потік звертається до volatile змінної, він отримає його значення перед тим як використовувати її значення з кешу. Доступ здійснюється у відношенні по типу "виконується раніше". Якщо розібратися, то все, що буде видно потоку А, коли він звертався до volatile змінної, це змінна для потоку В. Тобто ти гарантовано не втратиш свої зміни з інших потоків.

Volatile змінні атомарні, тобто під час читання такої змінної використовується той самий самий ефект, що й при отриманні блокування: дані в пам'яті оголошуються недійсними або некоректними, і значення volatile змінної знову читається з пам'яті. При записі використовується ефект пам'яті, як і при звільненні блокування — volatile-поле записується у пам'ять.

Запис `volatile` змінної виконується в основу пам'ять, минуючи локальну. Читання `volatile` змінної виконується також із основної паняті. Таким чином, значення `volatile` змінної не зберігається в регістрах та локальної пам'яті потоку і операція читання гарантує повернення останнього значення `volatile` змінної:

```
public class TX{
   private volatile double R;
   public synchronized void AddR(){
         R = R + 1;
   }
}
```

`Volatile` гарантує, що всі потоки завжди будуть використати спільне вхідне значення R, а також - завжди бачити зміну значення R в іншому потоці. Потік, який читає поле R , побачить останнє записане значення R.

Атомарні класи з пакету java.util.concurrent.atomic:
https://javarush.com/ua/quests/lectures/ua.questservlets.level19.lecture02

- `AtomicInteger`,
- `AtomicBoolean`,
- `AtomicLong`,
- `AtomicIntegerArray`

- `AtomicLongArray`

Всі методі цих класів атомарні. Використовуються для операції типу x = x + a в потоках. Не потребують використання семафорів, а також дозволяють реалізувати *неблоковані* дії із спільним ресурсом.

Клас *java.util.concurrent.atomic.AtomicInteger*

Клас надає операції для int змінних, які можуть бути прочитані і записані атомарно, а також містить розширені атомарні операції.

Клас `AtomicInteger` (Таблиця 5.1) підтримує атомарні операції з базової змінної int. У нього є методи `get()` і `set`(), які працюють як читання і запис по змінними. Тобто набір має відношення «відбувається до» з будь-яким подальшим отриманням тієї ж змінної.

Таблиця 5.1 Методи класу `AtomicInteger`

Метод	Дії методу
`int addAndGet (int delta)`	Додає певне значення до поточного значення.
`boolean compareAndSet (очікуване int, оновлення int)`	Встановлює значення для оновленого значення, якщо поточне значення збігається з очікуваним значенням.
`int decrementAndGet()`	Зменшує поточне значення на одиницю.
`int getAndAdd (int delta)`	Додає це значення до поточного значення.
`int getAndDecrement()`	Зменшує поточне значення на одиницю.
`int getAndIncrement()`	Збільшує поточне значення на одиницю.
`int getAndSet (int newValue)`	Встановлює вказане значення та повертає старе значення.
`int incrementAndGet ()`	Збільшує поточне значення на одиницю.
`lazySet (int newValue)`	Зрештою встановлюється на задане значення.
`boolean weakCompareAndSet (очікуване, оновлення int)`	Встановлює значення для оновленого значення, якщо поточне значення збігається з очікуваним значенням.

У атомарного методу `compareAndSet`() є ці особливості узгодженості пам'яті.

```
public boolean compareAndSet ( X int,  Y int)
```

Атомарно встановлює значення для даного обновленого значення Y, якщо текуче значення співпадає зі значенням X, що очікується.

Мова C#. Мова C# забезпечує розробника різноманітними засобами, які реалізують концепцію поділяємих змінних [30]. Вони відносяться до групи неблокуючих конструкцій (), які базуються на використанні класу `Interlocked` та модифікатору `volatile`. Операції з подялємимі змінними в C# характеризуються високою швидкістю по відношенню до подібних операцій в інших мовах та бібліотеках.

Клас `Interlocked` (Додаток А). Методи класу допомагають захистити від помилок, коли потік оновлює змінну, до якої можуть отримати доступ інші потоки, або коли два потоки виконуються одночасно на окремих процесорах.

Методи `Increment` і `Decrement` збільшують або зменшують змінну та зберігають отримане значення за *одну операцію.* За звичай, збільшення змінної не є елементарною операцією, що вимагає таких кроків:

- завантажити значення змінної з памятті в регістр.
- збільшити або зменшити значення.
- зберігти значення в пам'яті.

Якщо не використовувати `Increment` і `Decrement` потік можна витягнути після виконання перших двох кроків. Потім інший потік може виконати всі три кроки. Коли перший потік відновлює виконання, він перезаписує значення в змінній екземпляра, і ефект збільшення або зменшення, виконаного другим потоком, втрачається.

Метод `Add` атомарно додає ціле значення до цілочисельної змінної та повертає нове значення змінної.

Метод `Exchange` атомарно обмінює значення вказаних змінних.

Метод `CompareExchange` поєднує дві операції: порівняння двох значень і збереження третього значення в одній із змінних на основі результату порівняння. Операції порівняння та обміну виконуються як атомарні операції.

Забезпечує атомарні операції для змінних, які використовуються потоками. Використання методів класу `Interlocked` більш ефективним ніж оператор `locked` через то, що використається неблокований доступ до об'єктам, що поділяються.

Приклади використання критичної секції (`lock`) і методу `Increment()` класу `Interlocked` для роботи зі спільним ресурсом x.

```
public void Add(){
    lock(x)  { x ++; }
}

public void Add(){
    int x = Interlocked.Increment(ref x)
}
```

Статичні методи дозволяють працювати з кеш-пам'яттю у синхронізованому режимі.

- **Запитання для модульного контролю**
  1. Назвіть види взаємодії потоків?
  2 Поняття комунікації потоків?
  3 Поняття синхронізації потоків?
  4. Що включає постановка та загальна схема вирішення завдання взаємного виключення ?
  5 Визначення критичної ділянки?
  6. Наведіть алгоритми примітивів ВХІДКД і ВИХІДКД ?
  7. У чому полягає головна відмінність двох підходів до рішення завдання взаємного виключення ?
  8. Концепція атоміс- змінних?
  9. Мова Ada. Реалізація концепції атоміс- змінних?
  10 Мова Java. Реалізація концепції атоміс- змінних?
  11. Мова C#. Реалізація концепції атоміс- змінних?

ЛЕКЦІЯ 6
Семафори

6.1 Концепція семафорів

Механізм семафорів запропонував математик Е.Дейкстра [2; 11]. У класичній інтерпретації механізм семафорів – це спеціальний захищений тип **`Semaphore`** та дві неподільні операції над змінною `S` цього типу: **`P(S)`** і **`V(S)`**. Неподільність операції означає, що операцію не можна переривати, поки не завершиться її виконання. Бінарні семафори набувають значень 0 і 1, багатозначні (множні) семафори – значень 0, 1, 2, .. M.

Алгоритми операцій `P(S)` і `V(S)`:

Операція P(S):

1. Перевірити значення `S`.
2. Якщо `S = 0`, то блокувати процес, що виконує цю операцію.
3. Інакше `S:= S - 1`

Операція V(S):

1. `S := S - 1`.

Механізм семафорів є універсальним, який може бути використаний як для вирішення завдання взаємного виключення (рис. 6.1), так і для синхронізації процесів (рис. 6.2).

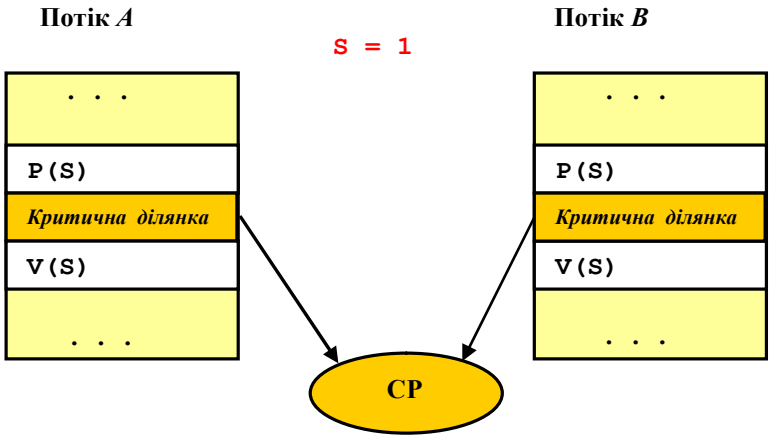


Рис. 6.1. Загальна схема вирішення завдання взаємного виключення з використанням семафорів

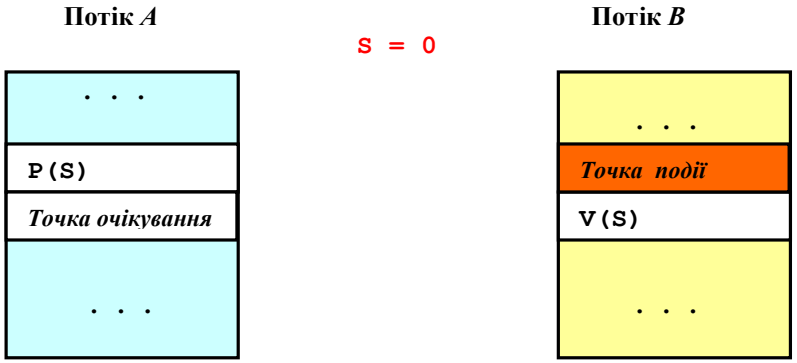


Рис. 6.2. Загальна схема вирішення завдання синхронізації з використанням семафорів

Семафори використовуються в потоках, описуються як глобальні змінні, є спільними ресурсами і потребують вирішення завдання

взаємного виключення. Тобто використання семафору в потоці є критичною ділянкою. Рішення завдання взаємного виключення відносно семафорів бере на себе система керування потоками (частиною ОС). Але це рішення спрощується через то, що дії з семафорами складаються лише з двох операцій P() і V(). На використання операцій P() і V() накладаться обмеження: виконання будь якої операції на семафором S дозволяється потоку T1 лише в тому випадку, колі інший потік T2 не виконує дій с цим семафором через операцію P(S) або V(S). Якщо семафор S вже використовується в T2 , то потік T1 блокується при намаганні виконати операцію з цим семафором. Розблокування потоку відбувається по завершенню в T2 дій з семафором S.

6.2. Семафори в мові Ada

У другому стандарті мови Ada механізм семафорів подано у вигляді пакету `Synchronous_Task_Control` в додатку Annex D: Real-Time Systems [25]. Пакет реалізує механізм семафорів наступним чином. Семафорний тип забезпечується приватним типом `Suspension_Object`, операції `P(S)` і `V(S)` реалізовані за допомогою процедур `Suspend_Until_True()` і `Set_True()`. Використовується бінарний логічний семафор, тобто семафорні змінні типу `Suspension_Object` набувають значень `false` і `true`. Крім указаних процедур, в пакеті реалізовані допоміжні процедури `Set_False()` для встановлення значення семафора в `false` і `Current_State()` для зчитування поточного значення семафора.

Специфікація пакета:

```
package Ada.Synchronous_Task_Control is

   type Suspension_Object  is limited private;

   procedure Set_True(S :  in out Suspension_Object);
   procedure Set_False(S : in out Suspension_Object);
   function  Current_State(S : Suspension_Object)
                                    return Boolean;
   procedure Suspend_Until_True(S : in out
                                Suspension_Object);
private
   .  .  .
end Ada.Synchronous_Task_Control;
```


6.3. Семафори в WinAPI

Семафори в бібліотеці WinAPI реалізовані за допомогою спеціального типу, а також кількох функцій [18]. Семафор в реалізації WinAPI визначається як лічильник, що набуває значення від нуля до визначеного значення, тобто використовує багатозначний семафор. Якщо семафор дорівнює нулю, то він заборонений, і в разі виклику процесом функції *очікування* цей процес буде блокований доти доки часу, поки інший процес не змінить значення семафора (інкрементує лічильник).

Створення та використання семафорів у WinAPI виконується за допомогою типу `HANDLE`, а також набору функцій:

1.Функція `CreateSemaphore()` створює об'єкт - семафор:

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
                    // покажчик на  атрибути захисту
  LONG lInitialCount,
                     // початкове значення семафора
  LONG lMaximumCount,
                    // максимальне значення семафора
  LPCTSTR lpName
                    // покажчик на ім'я об'єкта
);
```

Параметри:

- `lpSemaphoreAttributes` – покажчик на структуру `SECURITY_ATTRIBUTES`, яка визначає, чи може ідентифікатор, що повертається, бути успадкованим дочірнім процесом. Якщо покажчик установлено в `NULL`, то покажчик не може бути успадкованим;
- `lInitialCount` – початкове значення лічильника семафора. Воно не може бути меншим за нуль або більшим за максимальне значення; якщо початкове значення лічильника встановлено в нуль, то семафор знаходиться в забороненому (закритому) стані;
- `lMaximumCount` – визначає максимальне значення семафора, яке має бути більше нуля;

- lpName – визначає ім'я об'єкта у вигляді рядка, який має закінчуватися нулем. Якщо параметр NULL, то створюється семафор, який не має ім'я.

2. Функція OpenSemaphore() повертає ідентифікатор уже створюваного семафора:

```
HANDLE OpenSemaphore(
   DWORD dwDesiredAccess, // прапор доступу
BOOL bInheritHandle,      // прапор наслідування
   LPCTSTR lpName         // покажчик на ім'я об'єкта
);
```

Параметри:

- dwDesiredAccess – визначає вид доступу до об'єкта – семафора;
- bInheritHandle – визначає, чи буде ідентифікатор успадкованим. Якщо TRUE, створюваний процес, може успадкувати ідентифікатор;
- lpName – ім'я об'єкта в рядку, що закінчується нулем.

3.Функція ReleaseSemaphore() збільшує лічильник семафора на визначене значення.

```
BOOL ReleaseSemaphore(
  HANDLE hSemaphore,  //ідентифікатор об'єкта семафора
  LONG lReleaseCount, //  число, що додається до
                      //  поточного значення семафора
  LPLONG lpPreviousCount  // адрес попереднього
                          // лічильника
);
```

Параметри:

- hSemaphore – ідентифікатор об'єкта - семафора;
- lReleaseCount – визначає значення, на яке буде змінюватися значення семафора;
- lpPreviousCount – покажчик на змінну для отримання попереднього значення семафора.

4. Функція WaitForSingleObject() належить до групи функцій *очікування*. Вона перевіряє значення семафора. Якщо семафор дорівнює нулю, то процес блокується; інакше, значення семафора зменшується на одиницю.

```
DWORD WaitForSingleObject(
  HANDLE hHandle,             // ідентифікатор об'єкта
  DWORD dwMilliseconds,   //  тайм-аут в мілісекундах
  BOOL  bAlertable       //  прапор раннього виконання
);
```

Параметри:

- hHandle – ідентифікатор об'єкта - семафора;
- dwMilliseconds – визначає час очікування в мілісекундах. Якщо час очікування не визначений (процес буде чекати поки не зміниться значення семафора), то цей параметр встановлюється як INFINITE;
- bAlertable – визначає можливість раннього виконання функції.

Функції очікування. Існує три типи *функцій очікування*, які дозволяють потоку блокувати своє виконання залежно від результатів виконання перевірки стану об'єкта синхронізації:

- одиночні,
- множинні,
- попереджувальні.

Одиночні функції синхронізації SignalObjectAndWait, WaitForSingleObject, WaitForSingleObjectEx використовують ідентифікатор (типу HANDLE) об'єкта синхронізації і блокують процес, якщо об'єкт синхронізації знаходиться у забороненому стані; функції закінчують своє виконання, якщо виконується таке:

- указаний об'єкт знаходиться у дозволеному стані,
- вичерпаний час очікування. Час очікування може бути встановлений в INFINITE, що означає необмежений час очікування.

Функція **SignalObjectAndWait** дозволяє викличному потоку встановити стан одного об'єкта в дозволений і чекати дозволеного стану іншого об'єкта.

Формати виклику цих функцій:

```
BOOL SignalObjectAndWait(
  HANDLE hObjectToSignal,
                //ідентифікатор об'єкта, що сигналізує
  HANDLE hObjectToWaitOn,
                // ідентифікатор об'єкта-сигналу
  DWORD dwMilliseconds,
                //тайм-аут у мілісекундах
  BOOL bAlertable
                //прапор раннього виконання
);

DWORD WaitForSingleObject(
   HANDLE hHandle,
                    // ідентифікатор об'єкта,
                    // від якого очікується сигнал
   DWORD dwMilliseconds   // тайм-аут у мілісекундах
);


DWORD WaitForSingleObjectEx(
   HANDLE hHandle,  // ідентифікатор об'єкта,
                    // від якого очікується сигнал
   DWORD dwMilliseconds, // тайм-аут в мілісекундах
   BOOL  bAlertable      // прапор раннього виконання
                         //(для операцій В/В)
 );
```

Множинні функції WaitForMultipleObjects, WaitForMultipleObjectsEx, sgWaitForMultipleObjects, MsgWaitForMultipleObjectsEx дозволяють викличному потоку визначати масив, який містить один або кілька ідентифікаторів об'єктів синхронізації; функції закінчують своє виконання, якщо вони реалізують таке:

- стан одного або кількох (усіх) об'єктів дозволений – можна вказувати об'єкти для виклику функцій;
- вичерпаний час очікування. Час очікування може бути встановлений в INFINITE, що означає необмежений час очікування.

Функції MsgWaitForMultipleObjects і MsgWaitForMultipleObjectsEx дозволяють визначити об'єкт події введенням масивів ідентифікаторів об'єктів, для чого потрібно визначити тип уведення в черзі введення потоку. Наприклад, потік може використати MsgWaitForMultipleObjects для власного блокування доти, доки, поки стан об'єкта не буде встановлено і дозволено введення з маніпулятора «миша» в черзі введення потоку. Потік може використати функції GetMessage або PeekMessage для оброблення введення.

Формати виклику цих функцій

```
DWORD WaitForMultipleObjects(
   DWORD nCount,       // кількість об'єктів у масиві
   CONST HANDLE *lpHandles,
              // покажчик на масив ідентифікаторів
   BOOL bWaitAll,      // чекати одного або всіх
   DWORD dwMilliseconds // тайм-аут
);

DWORD WaitForMultipleObjectsEx(
   DWORD nCount, // кількість об'єктів у масиві
   CONST HANDLE *lpHandles,
                // покажчик на масив ідентифікаторів
   BOOL  bWaitAll,         // чекати одного або всіх
   DWORD dwMilliseconds,   // тайм-аут
   BOOL  bAlertable     // прапор раннього виконання
);

DWORD MsgWaitForMultipleObjects(
   DWORD    nCount,      //кількість об'єктів у масиві
   LPHANDLE pHandles,    // покажчик на масив
                         // ідентифікаторів
   BOOL  fWaitAll,       // чекати одного або всіх
   DWORD dwMilliseconds, // тайм-аут
   DWORD dwWakeMask      // тип події уведення
                         // для очікування
);

DWORD MsgWaitForMultipleObjectsEx(
    DWORD    nCount,      //кількість об'єктів у масиві
    LPHANDLE pHandles,    // покажчик на масив
                          // ідентифікаторів
    DWORD dwMilliseconds, // тайм-аут
```

```
    DWORD dwWakeMask,      // тип події уведення
                           // для очікування
    DWORD dwFlags          // прапор очікування
);
```

Попереджувальні функції `MsgWaitForMultipleObjectsEx`, `SignalObjectAndWait`, `WaitForMultipleObjectsEx`, `WaitForSingleObjectEx` можуть додатково виконувати попереджувальні операції. Вони також можуть закінчуватися з досягненням деякої умови, але також і в разі появи в системній черзі введення-виведення інформації, або віддаленого виклику процедур.

Функції очікування можуть змінювати стан об'єктів синхронізації, причому змін зазнають тільки ті об'єкти, від яких залежить вихід з функції очікування і розблокування потоку відповідно. Таким чином, функції змінюють такі типи об'єктів синхронізації:

- лічильник семафора зменшується на одиницю і досягає забороненого стану, якщо дорівнює нулю;
- об'єкти мютексу (`Mutex`), подія автоскидання встановлюються в заборонені;
- стан таймера синхронізації заборонений.

6.4. Семафори в мовах Java і C#

Механізм семафорів в мові Java підтриманий за допомогою класу `Semaphore` з пакету `java.util.concurrent`[2, 10].

Конструктори класу:

```
- Semaphore(int p1)
- Semaphore(int p1, boolean p2),
```

де параметр `p1` встановлює максимальне значення семафору Параметр `p2` визначає черговість розблокування потоків за цим семафором: якщо `p2=true`, то розблокування вибувається у порядку блокування, інакше цей порядок не визначений.

Для семафора визначені операції, аналогічні операціям `P(S)` і `V(S)`.

Операція `P(S)` реалізована через метод **`acquire()`**:

```
- void acquire() throws InterruptedException
```



```
- void acquire(int p) throws InterruptedException)
```

Параметр `p` задає

Операція `V(S)` реалізована через метод **`release()`**:

```
-  void release()
-  void release(int p)
```

Перший варіант метода виконує дію S = S +1, другий S = S + p .

Приклад 6.1 використання семафора в мові Java:

```
import java.util.concurrent.Semaphore;
public class Lab 61 {

   public static void main(String[] args) {

      // створення бінарного семафору
      Semaphore Sm = new Semaphore(1);

      // спільний ресурс
      ComResource Sr = new CommonResource();

      // створення потоків
      new Thread(new WorkThread(Sr, Sm)).start();
      new Thread(new WorkThread(Sr, Sm )).start();
    }

    class ComResource{
       int x = 0;
    }
    class WorkThread implements Runnable{

        ComResource Buf;
        Semaphore S;

     WorkThread(ComResource res, Semaphore sem {
         Buf=res;
```

```
        S=sem;
    }

      public void run(){
                . . .
        S.acquire();
          for (int i = 1; i < 10; i++){
             Buf.x = Buf.x + sin(i);
             Thread.sleep(10);
           }
        S.release();
             . . .
      }// run

    }// WorkThread

}// Lab61
```

Механізм семафорів в мові C# реалізований за допомогою класу `Semaphore`, який має три конструктори:

`Semaphore (int p1, int p2)`

- параметр `p1` задає початкове значення семафру, `p2` – максимальне значення семафору

`Semaphore (int p1, int p2, string name):`

- параметр `name` задає ім'я семафора

`Semaphore (int p1, int p2, string name, out bool p3)`

- параметр `p3` при значенні `true` вказує, що новий семафор був створений успішно.

Для роботи з потоками клас `Semaphore` має два основних методів

- `WaitOne()` – реалізація операції P(S)
- `Release()` – реалізація операції V(S)

Перший варіант метода виконує дію `S = S+1`, другий `S= S+p`.

Приклад 6.2 використання семафора в мові C#:

```
using System;
using System.Threading;
public class  Lab 62 {


    // спільний ресурс
    int Buf = 0;

    // створення  бінарного семафору
    Semaphore S = new Semaphore(1,1);

    public void Func1(){
                . . .
         S.WaitOne();
            Buf = Buf + 10;    // КД
         S.Release();
              . . .
     }// F1

     public void Func2(){
                . . .
         S.WainOne();
            Buf = Buf + 20* Buf;    // КД
         S.Release();
              . . .
     }// F2

     public static void Main(String[] args) {
        // створення і запуск потоків
        Thread T1 = new Thread(Func1);
        Thread T2 = new Thread(Func2);
        T1.Start();
        T2.Start();

       Console.WriteLine("Main thread starts";
   }

}// Lab61
```

- **Запитання для модульного контролю**
  1. Концепція механізму семафорів ?
  2. Загальна схема вирішення завдання взаємного виключення за допомогою семафорів ?
  3. Наведіть алгоритми операцій P і V над семафорами ?
  4. Яке призначення мають багатозначні семафори ?
  5. Як застосують семафори в завданні взаємного виключення і в завданні синхронізації процесів ?
  6. Які процедури в Ада семафорах виконують роль операції `P()` і `V()` ?
  7. Яке призначення типу `HANDLE` в бібліотеці WinAPI ?
  8. В чому полягає різниця між семафорами та мютексами в WinAPI ?
  9. Реалізація семафорів в мові Java?
  10. Реалізація семафорів в мові C#?

ЛЕКЦІЯ 7
Мютекси

Механізм мютексів (Mutual Exclusion) є модифікацією механізму семафорів, яка направлена на рішення виключно завдання взаємного виключення [2, 6, 12]. Механізм мютексів – це спеціальний захищений тип `Mutex` та дві неподільні операції над змінною *M* цього типу: `P(M)` і `V(M)`. Неподільність операції означає, що операцію не можна переривати, поки не завершиться її виконання. На відміну від семафорів мютекси не мають значень, *а мають стани*

- *вільний*
- *зайнятий.*

Мютекс має стан зайнятий, якщо він захоплений потоком, при цьому мютексом може володіти лише ОДИН потік. Вхід в критичну ділянку потоку і, відповідно, працювати зі спільним ресурсом, може лише той потік, який захопив мютекс. Якщо мютекс не потрібний більше потоку, то потік змінює стан мютексу на вільний. Вільний мютекс може захопити інший потік.

Алгоритми операцій `P(M)` і `V(M)`:

Операція P(M):

4. Перевірити стан мютексу `M`.
5. Якщо `M` має стан зайнятий, то блокувати потік, що виконує цю операцію.
6. Інакше встановити для мютекса стан зайнятий і дозволити входження потоку в критичну ділянку.

Операція V(M):

2. Змінити стан мютекса на вільний

На рис.7.1 представлено схему використання мютексів для вирушення завдання взаємного виключення. Початковий стан мютексу *вільний* можна встановити або при створенні мютексу, або окремою операцією `V(M)`. При створенні мютексу потік може його захопити, встановив стан в *зайнятий.*

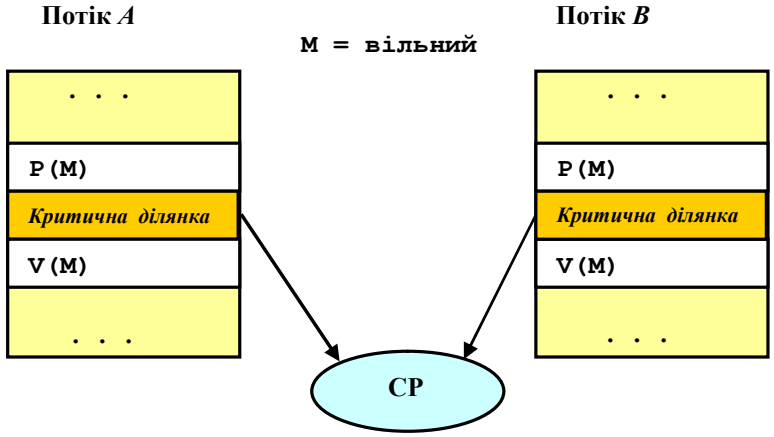


Рис. 7.1. Загальна схема вирішення завдання взаємного виключення з використанням мютексів

7.1. Мютекси в мові C#

Створення і використання мютексів в C# виконується за допомогою класу **`Mutex`** [2,31]:

```
public sealed class Mutex:
                    System.Threading.WaitHandle
```

Конструктори класа

`Mutex()`	Створення нового екземпляра класу зі стандартними властивостями.
`Mutex(Boolean)`	Створення нового екземпляра класу зі логічним значенням, яке вказує що він захоплює мютекс.

Основні методи

`Close()`	Звільнює всі ресурси, які удерживаемые поточним мютексом
`OpenExisting(String)`	Відкриває вказаний іменований мютекс
`ReleaseMutex()`	Звільнює мютекс (встановлює стан вільний).
`WaitOne()`	Блокує поточний потік
`WaitOne(Int32)`	Блокує поточний потік на вказаний інтервал часу в мілі секундах.

Створення мютексу здійснюється шляхом створення екземпляру класу Mutex:

```
private static Mutex M1 = new Mutex();
private static Mutex M2 = new Mutex(true);
```

Операція `P(M)`:

- метод **`public virtual bool WaitOne();`**

Блокує поточний потік до зміни стану мютекса на вільний

Операція `V(M)`:

- метод **`public void ReleaseMutex();`**

Використовується, коли потік звільнює мютекс шляхом встановлення в стан вільний

❖ Приклад 7.1

```
//-------------------------------------------------------
--|C#.Використання мютексів                            --
--|в завданні взаємного виключення                     --
-------------------------------------------------------//

using System;
using System.Threading;

class Lab72  {

    // спільний ресурс
```

```
    int X = 0;

  // створення мютексу
  private static Mutex Mut = new Mutex();

  // потокові методи
  private static void ThreadT1() {
            .  .  .
      Mut.WaitOne();
          X = X + 100;     --  КД
      Mut.ReleaseMutex();

  }

  private static void ThreadT2()  {
            .  .  .
      Mut.WaitOne();
          X = X *25 + 250;  --  КД
      Mut.ReleaseMutex();
            .  .  .
  }


  static void Main() {

          Thread T1 = new Thread(ThreadProcT1);
          Thread T2 = new Thread(ThreadProcT2);

          T1.Start();
          T2.Start();
   }
}
```

7.2. Мютекси в бібліотеці WinAPI

Створення мютексу виконується зі змінною типу `HANDLE` за допомогою операцій `CreateMutex()`, `OpenMutex()`, `InitMutex()` [13].

У разі використанні мютексів роль функцій `P(M)` і `V(M)` виконувати функції очікування `WaitForSingleObject()` та функція `ReleaseMutex()`.

1.Функція `CreateMutex()` створює об'єкт - мютекс:

```
HANDLE CreateMutex(
```



```
    LPSECURITY_ATTRIBUTES lpMutexAttributes,
                        // покажчик на  атрибути захисту
    BOOL lInitialOwner,
                        // початковий стан мютексу
    LPCTSTR lpName
                       // покажчик на ім'я об'єкта
);
```

Параметри:

- lpMutexAttributes – покажчик на структуру `SECURITY_ATTRIBUTES`, яка визначає, чи може ідентифікатор, що повертається, бути успадкованим дочірнім потоком. Якщо покажчик установлено в `NULL`, то мютекс отримує дескриптор безпеки за замовченням;
- `lInitialOwner` – початковий стан мютексу. Значення параметра `NULL` встановлює для мютекса стан *вільний*;
- `lpName` – визначає ім'я об'єкта у вигляді рядка, який має закінчуватися нулем. Якщо параметр `NULL`, то створюється мютекс, який не має ім'я.

Якщо функція виконується успішно, то вона повертає значення дескриптора створеного мютекса.

2. Функція `OpenMutex()` повертає ідентифікатор уже створюваного мютексу:

```
HANDLE OpenMutex(
   DWORD dwDesiredAccess, // прапор доступу
   BOOL bInheritHandle,   // прапор наслідування
   LPCTSTR lpName         // покажчик на ім'я об'єкта
);
```

Параметри:

- `dwDesiredAccess` – визначає вид доступу до об'єкта – мютекса;
- `bInheritHandle` – визначає, чи буде ідентифікатор успадкованим. Якщо `TRUE`, створюваний потік може успадкувати ідентифікатор;
- `lpName` – ім'я об'єкта в рядку, що закінчується нулем.

3.Функція `ReleaseMutex()` з встановлює для мютексу стан *вільний.*

```
BOOL ReleaseMutex(
  HANDLE hMutex,  //ідентифікатор об'єкта мютекса
);
```

Параметри:

- `hMutex` – ідентифікатор об'єкта - мютекса;

4. Функція `WaitForSingleObject()` належить до групи функцій *очікування*. Вона перевіряє стан мютексу. Якщо стан мютекса занятий, то потік блокується; інакше - стан мютекса встановлюється в зайнятий.

```
DWORD WaitForSingleObject(
  HANDLE hHandle,            // ідентифікатор об'єкта
  DWORD  dwMilliseconds      // час очікування
);
```

Параметри:

- `hHandle` – ідентифікатор об'єкта - мютекса;
- `dwMilliseconds` – визначає час очікування в мілісекундах. Якщо час очікування не визначений (потік буде чекати на зміну стану мютекса), то цей параметр встановлюється як `INFINITE;`

❖ Приклад 7.2

```
------------------------------------------------------------
--|Ada.Використання мютексів у бібліотеці  WinAPI       --
--|в завданні взаємного виключення                      --
------------------------------------------------------------
procedure Lab72 is

   Ресурс: integer;  -- спільний ресурс (глобальна змінна)

   -- HANDLE змінна для створення мютексу
   Mute: HANDLE;

   -- допоміжні змінні
   Temp: DWORD;
   p1:   Boolean;
```



```
procedure Запуск_Задач is

   task T1;
   task T2;
   task T3;

   task body T1 is
      X1: integer;      -- локальна змінна
   begin
      put_line("Process T1 started");
           .  .  .
      --  Критична  ділянка
      Temp:= WaitForSingleObject(Mute,Infinite);
         Ресурс := Ресурс + X1;
      p1:= ReleaseMutex(Mute);
           .  .  .
      put_line("Process T1 finished");
   end T1;
   ---------------------------------------
   task body T2 is
      X2: integer;      -- локальна  змінна
   begin
      put_line("Process T2 started");
           .  .  .
      --  Критична ділянка
      Temp:= WaitForSingleObject(Mute,Infinite);
         Ресурс := Ресурс + X2;
      p1:= ReleaseMutex(Mute);
           .  .  .
      put_line("Process T2 finished");
   end T2;
   ---------------------------------------------
   task body T3 is
      X3: integer;      -- локальна  змінна
   begin
      put_line("Process T3 started");
           .  .  .
      --  Критична ділянка
      Temp:= WaitForSingleObject(Mute,Infinite);
         Ресурс := Ресурс + X3;
      p1:= ReleaseMutex(Mute);
           .  .  .
      put_line("Process T3 finished");
   end T3;
begin
```

```
      null;

   end Запуск_Задач;

begin -- основна процедура

   put_line("  Main procedure started ");

   --  створення мютекса
   Mute:= CreateMutex(NULL,0,NULL);

   -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab72;
```

- **Запитання для модульного контролю**
  1. Концепція мютексів?
  2. Відміна мютексів від семафорів ?
  3. Стани мютексів ?
  4. Загальна схема застосування мютексів в завданні взаємного виключення ?
  5. Які функції в WinAPI мютексах виконують роль операції `P()` і `V()` ?
  6. Яке призначення типу `HANDLE` в бібліотеці WinAPI ?
  7. Як реалізовано механізм мютексів в WinAPI ?
  8. Реалізація мютексів в C#?
  9. Початковий стан мютексу в завданні взаємного виключення?
  10. Які методи в C# для мютексів виконують роль операції `P()` і `V()` ?

ЛЕКЦІЯ 8
Події

Механізм подій (events) є розвитком механізму семафорів з орієнтацією виключно на рішення завдання синхронізації потоків [2, 9, 12]. Механізм подій – це спеціальний захищений тип Event та дві неподільні операції над змінною *E* цього типу: *P*(*E*) і *V*(*E*). Неподільність операції означає, що операцію не можна переривати, поки не завершиться її виконання. На відміну від семафорів мютекси не мають значень, *а мають стани*

- *сигнальний*
- *не сигнальний.*

Подія має стан сигнальний, якщо подія відбулась. Якщо стан події не сигнальний, то подія ще не відбулась і потік, якому потрібна ця подія, буде блокований до настання події.

Алгоритми операцій `P(E)` і `V(E)`:

Операція P(E):

7. Перевірити стан події `E`.
8. Якщо `E` має стан не сигнальний, то блокувати потік, що виконує цю операцію.
9. Інакше встановити для події стан несигнальний.

Операція V(E):

1. Встановити подію в сигнальний стан та розблокувати потік, який що чекає на подію.

На рис.8.1 представлено схему використання події для вирушення завдання синхронізації. Початковий стан події *несигналь-*

ний можна встановити або при створенні події, або окремою операцією `V(E)`.

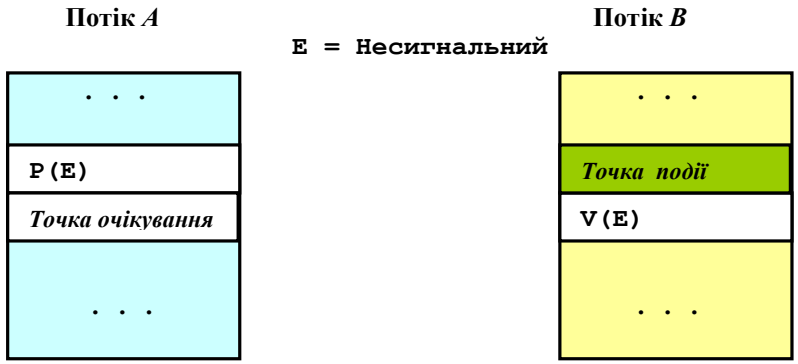


Рис. 8.1. Загальна схема вирішення завдання синхронізації з використанням подій

Після отримання сигналу про подію в потоці А, операція P(E) завжди змінює стан події на несигнальний (режим *автоматичного скидання сигналу*). Тобто сигнал про подію може отримати лише один потік.

В механізмі подій існує інший режим скидання (режим *ручного скидання сигналу*). Цей режим не змінює сигнальний стан події після отримання сигналу в операції P(E) і його можна використовувати, якщо сигнал про подію слід надати кільком потокам. Потрібний режим скидання для події задається при створення події.

8.1 Події в бібліотеці WinAPI

Створення події виконується зі змінною типу `HANDLE` за допомогою функції `CreateEvent()`:

```
HANDLE CreateEvent(
   LPSECURITY_ATTRIBUTES    адреса_атрибутів_захисту
   BOOL прапор_ручної_установки_події
   BOOL    прапор_початкового_стану // true - сигнальний
   LPCTSTR  адреса_об'єкта_події
);
```


Якщо параметр `прапор_ручної_установки_події` **TRUE**, то для події обирається ручной режим скидання події, який потребує використання методі ResenEvent() для зміни стану на несигнальний. Якщо – **FALSE,** то створюється подія з режимом автоматичного скидання.

У разі використання механізму подій очікування події виконується за допомогою однієї із функцій очікування, наприклад, `WaitForSingleObject()`, а сигнал про подію, що відбулася, – функцією `SetEvent()`:

```
BOOL SetEvent((
   HANDLE  Подія        // об'єкт - подія
);
```

Для роботи з подіями використовують також функції:

- `OpenEvent()` – повертає значення існуючого об'єкта-події;
- `PulseEvent()` – забезпечує установку стану події в сигнальне і наступне перемикання його в несигнальне після реалізації посилання сигналу очікуваним потокам;
- `ResetEvent()` – виконує скидання події (установлює стан події в несигнальний).

У прикладах 8.1 – 8.3 наведено вирішення за допомогою механізму повідомлень завдання синхронізації.

У прикладі 8.1 задача `T1` інформує задачу `T2` про подію, що відбулася в задачі `T1` (`Подія1`), а задача `T3` чекає на подію, що відбудеться в задачі `T2` (`Подія2`).

❖ Приклад 8.1

```
------------------------------------------------------------
--|Використання подій бібліотеки  WinAPI                  --
--|в завданні синхронізації                                --
------------------------------------------------------------
procedure Lab81 is

   -- HANDLE  змінна для створювання  подій
```

```
Evn12: HANDLE;
Evn23: HANDLE;

-- допоміжні змінні
Temp  : DWORD;
p1, p2: BOOL;

procedure Запуск_Задач is

   task T1;
   task T2;
   task T3;

   task body T1 is
   begin
      put_line("Process T1 started");
          .  .  .
      -- Подія1
      -- Сигнал задачі T2 про подію (Подія1)в задачі T1
      p1:= SetEvent(Evn12);
          .  .  .
      put_line("Process T1 finished");
   end T1;
   ------------------------------------------------

   task body T2 is
   begin
      put_line("Process T2 started");
          .  .  .
      --  очікування на подію (Подію1) в T1
      Temp:= WaitForSingleObject(Evn12,Infinite);
              .  .  .
      -- Подія2
      -- Сигнал задачі T3 про подію (Подія2)
      p1:= SetEvent(Evn23);
          .  .  .
      put_line("Process T2 finished");
   end T2;
   ------------------------------------------

   task body T3 is
   begin
      put_line("Process T3 started");
          .  .  .
      --  Очікування на подію (Подія2) в T2
      Temp:= WaitForSingleObject(Evn23,Infinite);
          .  .  .
```



```
      put_line("Process T3 finished");
   end T3;

   begin
      null;
   end Запуск_Задач;

begin -- основна процедура
   put_line("  Main procedure started ");

   --  створення подій
   Evn12:=  CreateEvent(NULL, FALSE,FALSE, NULL);
   Evn23:=  CreateEvent(NULL, FALSE,FALSE, NULL);

   -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab81;
```

8.3 Події в мові C#

Для створення подій в C# використовують клас **EventWaitHandle** і два його похідних підкласи [28]

- System.Threading.AutoResetEvent
- System.Threading.ManualResetEvent

Створення події:

```
EventWaitHandle E = new EwentWaitHandle( p1, p2 );
```

Параметри конструктора (прапори):

p1
- True, False для встановлення початкового стану

p2
- EventResetMode.AutoReset , для обрання автоматичного скідання події
- EventResetMode.ManualReset для обрання ручного скидання події)

Наприклад:

```
static EventWaitHandle E1=new EventWaitHandle(false,
```

```
                                EventResetMode.AutoReset);

 static EventWaitHandle E2 = new EventWait
            Handle(true, EventResetMode.ManualReset);
```

Методи (21)

`Set()` - встановлення сигнального стану події
`WaitOne()` - очікування на подію через перевірку .
`Reset()` - встановлення несигнального стану події.

Таблиця 8.1 Методи класу `EventWaitHandle`

`Close()`	Звільнює всі ресурси, яки утримуються поточним об'єктом `WaitHandle`.
`CreateObjRef(Type)`	Створює об'єкт, який містить всю необхідну інформацію для створення проксисервера, для взаємодії з віддаленим об'єктом.
`Dispose()`	Звільнює всі ресурси, яки утримуються поточним екземпляром класу `WaitHandle`.
`Dispose(Boolean)`	При перевизначені у похідному класі звільняє некеровані ресурси, використовувані об'єктом `WaitHandle`, і за необхідності звільняє керовані ресурси.
`Equals(Object)`	Визначає, чи дорівнює об'єкт поточному об'єкту.
`GetHashCode()`	Служить хеш-функцією за замовченням.
`GetType()`	Повертає об'єкт `Type` для поточного екземпляру.
`MemberwiseClone()`	Створює неповну копію об'єкта `Object`.
`OpenExisting(String)`	Відкриває зазначену іменовану подію синхронізації, якщо вона вже існує.

❖ **Приклад 8.2**

```
//-----------------------------------------------------------
--|C#.Використання подій(автоматичне скидання)            --
--|в завданні синхронізації                                --
-----------------------------------------------------------//
using System;
```



```
using System.Threading;

class Lab84  {

   // подія несигнальна, автоматичне скидання
   static EventWaitHandle E = new
         EventWaitHandle(false, EventResetMode.AutoReset);

   // потокові методи
   private static void ThreadT1() {
                              . . .
         ПОДІЯ;

      // сигнал  для   потока T2 про подію
      E.Set();

                              . . .
   }  // ThreadT1

   // ---------------------------------------------------
      private static void ThreadT2() {

                           . . .

            //  очікування на подію в T1
            E.WaitOne();

                           . . .
      }// ThreadT2

    static void Main() {

        Thread T1 = new Thread(ThreadProcT1);
        Thread T2 = new Thread(ThreadProcT2);

        T1.Start();
        T2.Start();

    }// Main

}// Lab84
```

В наступному прикладі потік, де видувається подія, надсилає сигнал двом іншим потокам.

❖ **Приклад 8. 3**

```
//-----------------------------------------------------------
--|C#.Використання подій( ручне скидання)                 --
--|в завданні синхронізації                                --
-----------------------------------------------------------//
using System;
using System.Threading;

class Lab85  {

   // подія несигнальна,  РУЧНЕ скидання
   static EventWaitHandle E =  new
       EventWaitHandle(false, EventResetMode.ManualReset);

  // потокові методи
  private static void ThreadT1 () {

              .  .  .

     ПОДІЯ;

     // сигнал про подію для T2 і T3
     E.Set( );

        .  .  .

  } // ThreadT1

  // ------------------------------------------------

  private static void ThreadT2() {

              .  .  .

      //  очікування на подію в T1
      E.WaitOne();

             .  .  .

  }// ThreadT2

  // ------------------------------------------------

  private static void ThreadT3() {

                 .  .  .

     //  очікування на подію в T1
```



```
     E.WaitOne();

                  .  .  .

  }// ThreadT3


   static void Main() {

      Thread T1 = new Thread(ThreadProcT1);
      Thread T2 = new Thread(ThreadProcT2);
      Thread T3 = new Thread(ThreadProcT3);

      T1.Start();
      T2.Start();
      T3.Start();

    }// Main
```

➢ Запитання для модульного контролю

1. Назвіть стани події?
2. Що включає постановка та загальна схема вирішення завдання синхронізації через події?
3. Наведіть алгоритми операцій з подіями `P(E)` і `V(E)`?
4. C#. Метод `WaitOne()` ?
5. Метод для створення події ? Його параметри?
6. WinAPI. Дії методу `SetEvent();`
7. WinAPI. Дії методу `ResetEvent();`
7. Дії методу `E:= CreateEvent(NULL,TRUE,FALSE,NULL)`?
8. Який тип має об'єкт - подія в бібліотеці WinAPI ?
9. Як реалізовано механізм подій в WinAPI ?
10. Засоби створення події в C# ?
11. C#. Як можна задавати режим скидання події?

ЛЕКЦІЯ 9
Критичні секції

Механізм *критичних секцій* – це засіб, який об'єднує конструкції `ВХІДКД` і `ВИХІДКД` в ***єдиний*** оператор виду

```
CriticalSection(X) {      }
```

де X – ідентифікатор критичної секції [2, 10].

Це позбавляє від помилок, які властиві семафорам (мютексам), коли на вході в критичну ділянку перевіряється один семафор, а на виході змінюється інший семафор і це ніяк не контролюється.

Структура оператора ***CriticalSection(X)***

```
CS(X)
        . . .        -- тіло оператора  (критична ділянка)
end CS;
```

Алгоритм ператора ***CriticalSection(X)***

1 Перевірити, чи знаходиться будь який потік в критичної секції з ідентифікатором X
2 Якщо знаходиться, то БЛОКУВАТИ потік
3 Якщо не знаходиться, то виконувати тіло оператора CS(), тобто дії в критичної ділянці

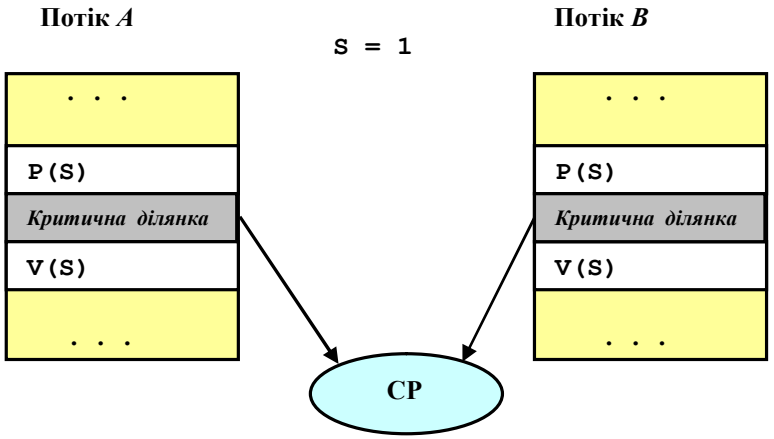


Рис. 9.1. Загальна схема вирішення завдання взаємного виключення з використанням семафорів

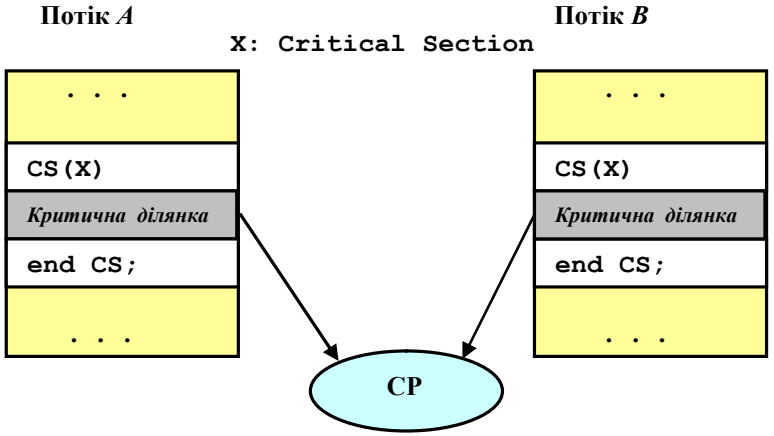


Рис. 9.2 Загальна схема вирішення завдання взаємного виключення з використанням критичних секцій

Механізм критичних секцій має більш високий рівень ніж семафори та мютекси. Надійність застосування критичних секцій крім об'єднання двох операцій P() і V() в єдиний оператор CS() забезпечується також посиленим контролем з боку операційної систем, яка бере на себе всі функції з перевірці стану критичної секції, блокуванню та розблокуванню потоків, що звертаються до критичної секції з намаганням виконати дії в критичної ділянці.

На малюнках 9.1 та 9.2 надано загальні схеми використання семафорів і критичних секцій при вирішенні завдання взаємного виключення.

Реалізація концепції критичних секцій виконано в бібліотеках WinAPI, OpenMP, мовах Java, C#

9.1 Критичні секції в WinAPI

Створення критичної секції виконується зі змінною типу `CRITICALSECTION` за допомогою операцій `Initialize CriticalSection()`, знищення – за допомогою операції `DeleteCriticalSection()`.

У разі використання критичних ділянок роль примітивів `ВХІДКД` і `ВИХІДКД` виконують функції `EnterCriticalSection()` та `LeaveCriticalSection()`. На відміну від семафорів і мютексів у механізмі критичних ділянок час очікування в операції входження в критичну ділянку `EnterCritical Section()` не встановлюється, тобто він завжди є `INFINITE`.

❖ Приклад 9.1

```
------------------------------------------------------------
--|Ada.Використання критичних секцій бібліотеки WinAPI   --
--|у задачі взаємного виключення                         --
------------------------------------------------------------
procedure Lab311 is

   Ресурс: integer;  -- спільний ресурс
                     -- (глобальна змінна)

   -- створювання критичної секції
   CrCec: CRITICAL_SECTION;

   procedure Запуск_Задач is
```



```
      task T1;
      task T2;
      task T3;

      task body T1 is
         X1: integer;           -- локальна змінна
      begin
         put_line("Process T1 started");
              .  .  .
         --  Критична ділянка
         EnterCriticalSection(CrSec);
            Ресурс := Ресурс + X1;
         LeaveCriticalSEction(CrSec);
              .  .  .
         put_line("Process T1 finished");
      end T1;
      ---------------
      task body T2 is
         X2: integer;            -- локальна змінна
      begin
         put_line("Process T2 started");
              .  .  .
         --  Критична ділянка
         EnterCriticalSection(CrSec);
            Ресурс := Ресурс + X2;
         LeaveCriticalSEction(CrSec);
              .  .  .
         put_line("Process T2 finished");
      end T2;
      --------------
      task body T3 is
         X3: integer;            -- локальна змінна
      begin
         put_line("Process T3 started");
              .  .  .
         --  Критична ділянка
         EnterCriticalSection(CrSec);
            Ресурс := Ресурс + X3;
         LeaveCriticalSEction(CrSec);
              .  .  .
         put_line("Process T3 finished");
      end T3;
   begin

      null;
```

```
   end Запуск_Задач;

begin     -- основна процедура

   put_line("  Main procedure started ");

   --  створення критичної секції
   InitializeCriticalSection(CrSec);

   -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab311;
```

9.2 Критичні секції в Java

Вирішення завдання взаємного виключення в Java програмах виконується за допомогою синхронізованих блоків та операторів `lock()/Unlock()`.

Синхронізовані блоки.

Загальна форма синхронізованого блоку

```
synchronized(X) {
        . . .
   //  оператори
        . . .
},
```

де **X** – це посилання на змінну, яка виконує функції ідентифікатору блоку і має тип Object.

Синхронізований блок гарантує, що виклик з потоків методу, що є членом об'єкта `Об'єкт`, буде виконуватися в режимі взаємного виключення.

❖ **Приклад 9.2**

Застосування синхронізованого блоку для взаємовиключного виклику методу `Зміна_Значення` з класу `Робота` в потоках – екземплярах класу `Доступ`.

```
/*-----------------------------------------------------------
-- Java. Застосування синхронізованого блоку             --
-----------------------------------------------------------*/
   class Робота{
```



```
      double Зміна_Значення(double e){
         return e*e;
      }
   }// Робота

   class  Доступ  implements Thread{

      Робота Блок1;   // посилання на клас Робота

      // конструктор
      Доступ(Блок  o){
         Блок1 = o;
      }

      public void run(){
         double x = 5;
         double y;

         synchronized(Блок1) {
            y = Блок1.Зміна_Значення(x);
         }
   }// Доступ

   class Основний{

      public static void main(String [] args){

         Робота Рб  = new Робота();
         Доступ  Д1 = new Доступ(Рб);
         Доступ  Д2 = new Доступ(Рб);

         Д1.start();
         Д2.start();

      }// main
   }// Основний
```

❖ **Приклад 9.3**

Застосування синхронізованого блоку для взаємовиключного доступу потоків до спільного ресурсу `Buf`.

```
/*-----------------------------------------------------------
Java. Застосування синхронізованого блоку
-----------------------------------------------------------*/
```

```
// створення  потокового класу
public class ПотікА  extends Thread {
   public void run(){
               . . .

         synchronized(X1){
             Buf= Buf + 1;      // КД
       }
               . . .
}// run
}// ПотікА
// ----------------------------------------

// створення  потокового класу
public class ПотікВ  extends Thread {
   public void run()   {
               . . .
         synchronized(X1){
             Buf = 3*Buf + 1200;    // КД
         }
               . . .
}// run
}// ПотікА

public class Lab2 {

    public static void main(String args[]){

       int Buf;   // спільний ресурс

       private final Object X1 = new Object();
  //  об'єкт для ідентифікації  синхронізованого блоку

       ПотікА     T1 =  new   ПотікА();
       ПотікВ     T2 =  new   ПотікВ();

          .  .  .
    }//  main
}// Lab2
```

Оператори lock()/Unlock()

Реалізація механізму критичних секцій через клас `ReentrantLock` та методи `Lock()/UnLock()`. Приклад використання методів надано нп рисунку 9.3

ReentrantLock CS = new ReentrantLock();

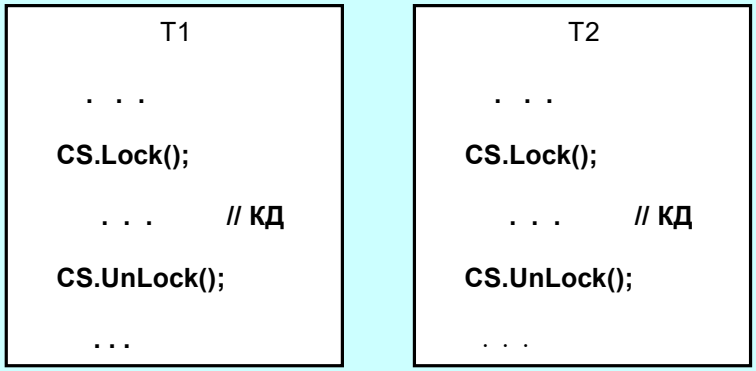


Рис 9.3 Критичні секції в мові Java

9.3 Критичні секції в C#

Механізм критичних секцій в C# реалізований через оператор lock [28]:

```
lock(X)
{
    . . .   // критична ділянка
}
```

де X - ідентифікатор критичної секції, має тип object

```
private readonly object Замок = new object();
```

```
T1
// потокова функція
public   void F1()
{
     . . .
    lock(Замок)
    {
       . . .  // КД
    }
     . . .
}// F1
```

```
T2
// потокова функція
 public void    F2()
{
     . . .
    lock(Замок)
    {
       . . .  // КД
    }
     . . .
}// F2
```

Рис 9.4 Критичні секції в мові C#

9.4 Критичні секції в OpenMP

Механізм критичних секцій в OpenMP реалізований через

- оператор `lock`.
- прагму `omp critical`

Оператор `lock`

Тип: `omp_lock_t`

Функції:

omp_init_lock();	InitializeCriticalSection();
omp_set_lock();	EnterCriticalSection().
omp_unset _lock();	LeaveCriticalSection()
omp_destroy_lock();	DeleteCriticalSection().

❖ **Приклад 9.4**

Критичні секції в OpenMP

```
omp_lock_t  CS;  //  критична секція
int n;
int Bufer;                //  спільний ресурс

omp_init_lock(&CS);       // ініціалізація КС

#pragma omp parallel private (n) shared (Bufer)
 {
        n = omp_get_thread_num();

        printf("Start work, thread %d\n", n);
                    .  .  .
        omp_set_lock (&CS))                 // P()
        {
             Bufer=Bufer*n + 100;           // КД
             printf(Bufer);
         }
        omp_unset_lock(&CS);                // V()
                        .  .  .

        printf("End work, thread %d\n", n);

 }

omp_destroy_lock(&CS);
```

Прагма `omp critical`

```
# pragma omp critical (Ім'я) {
            . . .     // КД
}
```

Приклад використання :

```
main(int argc, char *argv[])
{
    int x;
    x = 0;  // CP

    #pragma omp parallel shared(x) {
         int tid;
         tid=omp_get_num_thread();
         #pragma omp critical {
             x = x + tid*100;     //  КД
             printf(x);
         }
     }
}
```

- **Запитання для модульного контролю**

1. Головна відмінність механізмів критичних секцій від механізму семафорів?
2. За рахунок чого досягається більша надійність механізму критичних секцій ?
3. Що перевіряється при використанні критичних секцій? ?
4. Як здійснено реалізацію механізму критичних секцій в мові Java?
5. Як здійснено реалізацію механізму критичних секцій в мові C#?
6. Як здійснено реалізацію механізму критичних секцій в бібліотеці WinAPI?
7. Як здійснено реалізацію механізму критичних секцій в бібліотеці OpenMP?
8. Як формується ідентифікатор критичної секції в в мові Java?
9. Як формується ідентифікатор критичної секції в в мові C#?

ЛЕКЦІЯ 10
Монітори

Механізм моніторів реалізує концепцію рішення завдання взаємного виключення через *контроль спільного ресурсу.*

Ідея монітора, яку запропонував Б. Хансен і розвинув С. Хоар, ґрунтується на об'єднанні змінних, що описують спільний ресурс, і дій, які визначають засоби доступу до спільного ресурсу [21]. *Монітор – програмний модуль,* що містить змінні та процедури для роботи з ними, причому доступ до змінних можливий *тільки через процедури* монітора.

Монітор – засіб розподілу ресурсів і взаємодії процесів. Це призначення монітора реалізується за допомогою властивостей, якими наділені процедури монітора. Характерна особливість процедур монітора – *взаємне виключення* ними одне одного. У будь-який момент часу може виконуватися *тільки одна* процедура монітора. Якщо будь-який процес викликав і виконує процедуру монітора, то жоден процес не може виконувати будь-які процедури цього монітора. За спроби виклику іншим процесом процедури, що виконується, або іншої процедури монітора цей процес блокується і розміщується в черзі блокованих процесів доти, доки активний процес не закінчить виконання процедури монітора. Тобто в моніторі не може "знаходитись" більше одного процесу. Така властивість процедур монітора забезпечує взаємне виключення процесів, які працюють з монітором.

Загальна структура монітора:

```
monitor  Ім'я_Монітора;
       -- Опис локальних даних
       -- Опис процедур для доступу до даних
begin
       -- Ініціалізація локальних даних
```

```
end  Ім'я_Монітора;
```

У моніторі декларуються локальні змінні (спільні змінні), які захищені монітором, і процедури монітора. Значення локальних змінних можуть бути встановлені під час створення монітора. Далі значення цих змінних можуть бути прочитані або змінені процесами тільки за допомогою процедур, визначених у моніторі.

Приклад монітора:

```
monitor  Склад;

      Товар: Data;        -- спільний ресурс

      procedure На_Склад (T: in Data);
      procedure Зі_Складу(T: out Data);

begin

      Товар:= 0.0; -- ініціалізація спільного
                   -- ресурсу
end  Склад;
```

Властивості процедур монітора забезпечують вирішення завдання взаємного виключення за доступу до спільних ресурсів, об'явленими в моніторі. При цьому монітор формує чергу процесів, які викликали процедури монітора і є блокованими через зайнятість монітора (тобто спільного ресурсу).

10.1. Монітори в мові Ada

Концепцію моніторів у новому стандарті мови Ada реалізовано у вигляді спеціальних програмних модулів – захищених модулів (`protected units`) [2, 25]. Їх призначення – розширення можливості мови для програмування паралельних процесів, зокрема, для вирішення проблеми доступу до спільних ресурсів і синхронізації процесів. Крім того, захищені модулі забезпечують підтримку різних парадигм систем реального часу, для розроблення яких мову Ada використовують в першу чергу.

Спільні дані і операції над ними (захищені операції) об'єднуються в захищеному модулі, аналогічно тому, як це робиться в інших модулях мови Ada – пакетах. Доступ до спільних ресурсів можливий тільки через захищені операції, які мають властивості, що дозволяють вирішити завдання взаємного виключення під час роботи зі спільними ресурсами.

Як і всі модулі в мові, захищені модулі складаються зі специфікації і тіла.

Специфікація захищеного модуля:

```
PROTECTED [TYPE]   Ім'я_Захищеного_Модуля
                                  [Дискримінант]    IS
    --  Опис_Захищених_Операцій
[PRIVATE]
    --  Опис_Захищених_Елементів
END  Ім'я_Захищеного_Модуля;
```

Захищенні операції – це:

- захищені функції,
- захищені процедури,
- захищені входи.

Захищені функції забезпечують доступ тільки до *читання* захищених елементів. Але дозволяють робити це *одночасно* всім процесам автоматичним копіюванням елементів, які зчитуються. Це порушує головну властивість процедур монітора, яка дозволяє знаходитися в моніторі тільки одному процесу, але це "порушення" дозволяє скоротити час доступу до захищених елементів і не має будь-яких наслідків, оскільки зміна даних заборонена і не виконується.

Захищені процедури забезпечують ексклюзивний доступ до захищених елементів через читання і запис.

Захищені входи забезпечують ті самі функції, що й захищені процедури, додатково реалізуючи за допомогою *бар'єрів* ексклюзивний (умовний) доступ до тіла захищеного входу. Це дозволяє реалізувати за допомогою входів вирішення завдання синхронізації.

Приватна частина специфікації обмежує видимість захищених елементів: операцій і об'єктів, що описані в ній. Спільні дані, доступ до яких контролюється захищеним модулем, описуються в приватній частині його специфікації.

❖ **Приклад 10.1** Специфікації захищеного модуля

```
-----------------------------------------------------------
-- Ada. Захищений модуль                                 --
-----------------------------------------------------------
protected   Контроль  is
   procedure   Включення;        -- захищені підпрограми
   function    Перемкнути(X :  Float);
end  Контроль;

protected   type   Сенсор   is
   entry   Чекати;               -- захищені  входи
   entry   Сигнал;
   procedure Зміна_Стану(x : in  float);
   function  Замір_Стану return float;
private
   Прп: Boolean:= False;      -- Прапор
   Стан: float;
end  Сенсор;

protected   Блок232(Номер: in Positive)  is
   entry  Параметри(X: out  integer);
   procedure  ЗмінаПараметра(Y: in  integer);
private
   -- захищений елемент
   Об'єкт: array(1 .. Номер) of  integer;
end  Блок232;
```

Тіло захищеного модуля реалізує захищені операції, які об'явлені в його специфікації, використовуючи для цього локальні ресурси, які можуть бути об'явлені в тілі модуля.

```
PROTECTED  BODY   Ім'я_Захищеного_Модуля   IS
    --  Локальні_Описи
BEGIN
    --  Реалізація  захищених операцій і
    --  захищених елементів
END  Ім'я_Захищеного_Модуля;
```

Захищені процедури і функції реалізуються в тілі захищеного модуля, як це робиться в тілі пакета. На відміну від модулів - задач, реалізація захищеного входу в тілі захищеного модуля не пов'язана з оператором приймання `accept`, а виконується за допомогою тіла входу, у якому обов'язково використовується *бар'єр.*

Описання тіла захищеного входу:

```
ENTRY  Ім'я_Захищеного_Входу  WHEN   Умова  IS
BEGIN
   .  .  .     -- Послідовність_Операторів
END    Ім'я_Защищеного_Входу;
```

Конструкція **When** `Умова` – це бар'єр, де `Умова` – логічний вираз, який визначає *відкритий* або *закритий* вхід. Перевірка умови в бар'єрі виконується під час виклику захищеного входу. Якщо значення виразу `Умова` дорівнює `true`, то вхід відкритий і виконується тіло захищеного входу, інакше вхід є зачинений і виконання процесу, який викликав цей вхід, блокується до того часу, поки значення виразу `Умова` в бар'єрі не буде змінено в `true` іншою задачею за допомогою захищеної процедури або іншого захищеного входу.

В прикладі 10.2 наведено реалізацію тіла захищеного модуля `Сенсор`, специфікацію якого подано раніше у прикладі 10.1:

❖ **Приклад 10.2**

```
-----------------------------------------------------------
-- Ada. Тіло захищеного модуля                           --
-----------------------------------------------------------
   protected  body   Сенсор  is

      -- тіло захищеного  входу з бар'єром
      entry   Чекати  when  Прп is

      begin
         Прп:= False;
      end  Чекати;

      -- тіло захищеного  входу з бар'єром
      entry  Сигнал  when  not  Прп  is
      begin
         Прп:= True;
      end  Сигнал;

      procedure Зміна_Стану(x : in  float) is
      begin
```

```
      Стан := x;
   end Зміна_Стану;

   function  Замір_Стану return float is
   begin
      return Стан;
   end  Замір_Стану;

end  Сенсор;
```

Структуру захищеного модуля `Сенсор` показано на рис. 10.6. Захищену функцію `Замір_Стану()` і процедуру `Замір_Стану()` зображено справа, захищені входи `Чекати()`і `Сигнал()` – зліва. Захищені елементи (`Стан, Прп`) зображено всередині захищеного модуля в овалі.

Виклик захищеної функції дозволяє процесу зчитувати дані із захищеного модуля. Кілька процесів можуть виконувати таке читання *одночасно*, викликаючи потрібні функції. Під час виконання читання в тілі захищеної функції заборонено зміну даних. Тіло захищеної функції може містити виклик іншої захищеної функції, але не виклик захищеної процедури.

Виклик захищеної процедури дозволяє процесу як читати, так і змінювати інформацію в захищеному модулі. На відміну від захищеної функції під час виконання захищеної процедури дозволяється змінювати дані. Якщо кілька процесів виконують виклик захищених процедур, то тільки один з них отримує можливість роботи з викликаною процедурою. У тілі захищеної процедури дозволено виклик як захищеної функції, так і захищеної процедури.

Виклик закритого захищеного входу приводить до блокування процесу до того часу, поки вхід не стане відкритим, тобто умовний вираз в бар'єрі набуде значення `True`. Це може статися в разі виконання іншим процесом потрібної захищеної операції, пов'язаної зі змінними, використаними в бар'єрі.

Блокований процес розміщується в черзі, яка пов'язана із входом, а також зі змінними, що використовуються в бар'єрі. Якщо вхід відкритий, то виконується тіло входу.

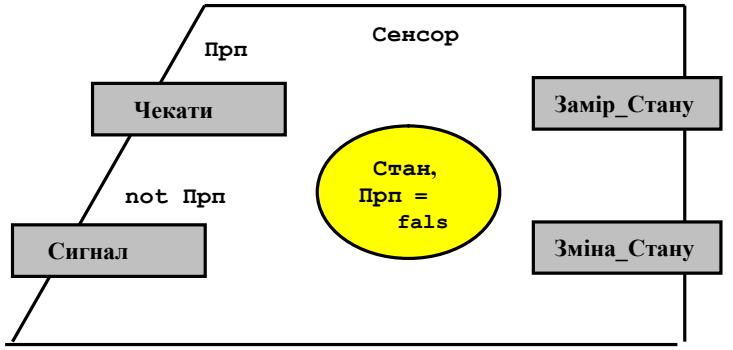


Рис. 10.6. Структура захищеного модуля

❖ Приклад 10.3

```
-------------------------------------------------------------
-- Ada.Захищений модуль у завданні взаємного виключення -
-------------------------------------------------------------
 protected   Буфер    is
     procedure  Додати  (Вклад:  out  Positive);
     procedure  Видалити(Вклад:  out  Positive);
 private
     Лічильник:  Integer:= 0;
 end Буфер;

 -- тіло захищеного модуля
 protected  body  Буфер  is

     procedure  Додати(Вклад: out  Positive) is
     begin
         Лічильник := Лічильник + 1;
         Вклад := Лічильник;
     end Додати;
     procedure  Видалити(Вклад: out Positive) is
     begin
         Лічильник := Лічильник - 1;
         Вклад := Лічильник;
     end  Видалити;
  end   Буфер;
```

Задачі додають або зменшують значення змінної Лічильник, викликаючи процедури Додати і Видалити захищеного модуля Буфер :

```
Буфер. Додати( Зарплата) ;     Буфер . Видалити (Плата ) ;
```

Захищений модуль Буфер гарантує синхронізований доступ задач до захищеної змінної Лічильник. Черги під час роботи із захищеним модулем не створюються, оскільки використовуються тільки захищені процедури, а не захищені входи.

У прикладі 10.4 захищений модуль Вклад виконує роль буфера, куди задачі Клієнт_А і Клієнт_В записують і звідки зчитують дані. Захищений модуль Вклад повинен забезпечити взаємовиключний доступ задач до спільного ресурсу, яким є змінна Рахунок, а також синхронізацію процесів залежно від стану ресурсу.

❖ Приклад 10.4

```
-------------------------------------------------------------
-- Ada.Захищений модуль у завданні взаємного виключення -
-- та синхронізації процесів                                --
-------------------------------------------------------------
 protected  Вклад    is

     entry  В_Банк(М  :  in     Гроші);
     entry  З_Банку(М :  out    Гроші);
 private
     Рахунок: Гроші;            -- спільний ресурс
     Прапор:  Boolean := False;

 end  Вклад;

 ----------

 protected  body     Вклад     is
     entry   В_Банк(М:  in    Гроші)
                           when  Прапор = False  is
     begin
        Рахунок:= М;
        Прапор:= True;
     end  В_Банк;
     entry  З_Банку(М:  out    Гроші)
```



```
                                when   Прапор = True  is
        М := Рахунок;
        Прапор := False;
    end  З_Банку;
end Вклад ;
-------------------------
 --  потоки
 task  Клієнт_А;

 task  body   Клієнт_А  is
    Дохід: Гроші;
 begin
          .  .  .
     --  виклик  входу В_Банк
     Вклад.В_Банк(Дохід);
          .  .  .
 end  Клієнт_А;
 ---------------------------
 task  Клієнт_В;

 task  body  Клієнт_В  is
    Сплата: Гроші;
 begin
          .  .  .
     --  виклик  входу  З_Банку
     Вклад.З_Банку(Сплата);
          .  .  .
 end  Клієнт_В;
```

10.2. Монітори в мові Java

Мова Java не має моніторів, подібних до захищених модулів у мові Ada, але вона дозволяє створювати класи, методи яких будуть мати основну властивість процедур монітора – виконуватися в режимі взаємного виключення [23]. Такі методи в Java повинні мати модифікатор synchronized. Інкапсуляція спільного ресурсу в класі-моніторі виконується за допомогою модифікатора private.

Метод, описаний як синхронізований, допускає виконання тільки в одному потоці. Якщо інший потік викликає синхронізований метод, який вже виконується, то такий потік буде блокований доти, доки не завершиться виконання синхронізованого методу.

Приклади синхронізованих методів:

```
   void   synchronized  Додати(double x);
   int    synchronized  Обсяг (int x, int y);
```

❖ **Приклад 10.5** Клас, який виконує функції монітора

```
/*-----------------------------------------------------
--  Реалізація монітора в мові Java                  --
----------------------------------------------------- */
class Монітор{

    private int Буфер;       // спільний ресурс

    // синхронізовані процедури доступу до
    // спільного ресурсу Буфер
    synchronized int Читати(){
       return Буфер;
    }
    synchronized void Писати(int x){
       Буфер = X;
    }
}// Монітор
```

Створення екземпляра класу `Монітор`:

```
Монітор  Z2 = new Монітор();
```

Одночасне виконання методів `Z2.Читати()` та `Z2.Писати()` не можливо. Якщо процес (потік) `B` викликав і виконує, наприклад, метод `Z2.Читати()`, то інший потік `A` буде блокований, якщо він спробує викликати будь-який метод екземпляра `Z2` класу `Монітор`, і зможе продовжити своє виконання тільки після завершення виконання методу `Z2.Читати()` в потоці `B`.

➢ **Запитання для модульного контролю**

1. Призначення моніторів ?
2 Особливості методів моторів?
3. Яку структуру має захищений модуль ? Види та призначення захищених операцій ?
4. Яку роль виконує бар'єр у захищеному модулі ?

5. Як виконується синхронізація за допомогою захищеного модуля ?
6. Яка особливость використання захищених функцій ?
7. Які черги формуються при використанні захищеного модуля ?
8. В чому полягає відмінність в реалізації концепції монітора в мовах Java і Ada ?
9. Особливості реалізації концепції моніторів в мові Java?
10. Мова Java. Призначення модифікатору synchronized?

ЛЕКЦІЯ 11
Завдання взаємного виключення

11.1. Вирішення завдання взаємного виключення в мові Ada

В мові Ада рішення завдання взаємного виключення можна здійснити за допомогою прагм `Atomic` і `Volatile`, механізму семафорів або з використанням механізму моніторів (захищених модулів) [2].

Змінні що поділяються. Простіший спосіб вирішення завдання взаємного виключення в мові Ада ґрунтується на використанні поділяємих змінних (атомік об'єктів). Їх описують через спеціальні прагми `Atomic` і `Volatile`, які контролюють використання спільних змінних. Існує чотири форми прагм:

```
pragma Atomic(Ім'я_Змінної);
pragma Volatile(Ім'я_Змінної);
pragma Atomic_Components(Ім'я_Масиву);
pragma Volatile_Components(Ім'я_Масиву);
```

Операції читання або зміни атомік об'єктів розглядаються і виконуються як неподільні операції. Тобто, якщо один потік уже виконує дії з атомік-об'єктом, то інший може отримати доступ до нього лише в разі звільнення об'єкта. Отже, всі дії потоків з атомік-об'єктами виконуються лише послідовно і ніколи – одночасно.

❖ Приклад 11.1

```
-------------------------------------------------------------
-- Ada.Pragma Atomic у завданні взаємного виключення  --
-------------------------------------------------------------
procedure Lab111 is
```



```
   Буфер: integer:= 10;          -- спільний ресурс

   pragma Atomic(Буфер);         -- захищена змінна

   task A;
   task body A is
   begin
      put_line("Process A started");

      -- Операція зі спільним ресурсом
            Буфер:= Буфер + 2;  -- критична ділянка

      put_line("Process A finished");
   end A;

   ------------------------------------------
   task B;
   task body B is
   begin
      put_line("     Process B started");

      -- Операція зі спільним ресурсом
         Буфер:= Буфер - 25;    -- критична ділянка

      put_line("     Process B finished");
   end B;

-- основна процедура
begin

   put_line("  Main procedure started ");

end Lab111;
```

Семафори. Задачі `A` і `B` виконують деякі дії над загальним ресурсом – цілою змінною **Буфер**. Ці операції мають виконуватись у взаємовиключному режимі, який можна забезпечити за допомогою семафорів.

Використовуючи загальну схему вирішення завдання взаємного виключення за допомогою семафорів необхідно:

- створити змінну – семафор з початковим значенням `false`

- розмістити операції P(S) і V(S) навколо критичної ділянки в кожному процесі

У мові Ada ці дії виконуються за допомогою типу Suspension_Object і підпрограм Suspend_Until_True() та Set_True().

У зв'язку з тим, що при створенні семафора (Sem_Ku) він автоматично набуває значення false, потрібноно змінити його на true перед запуском задач, щоб початковий вхід у критичну ділянку був відкритим для обох потоків. Явний запуск задач A і B виконується через виклик процедури Старт_Задач, де описані обидві задачі.

❖ Приклад 11.2

```
-------------------------------------------------------------
--  Ada.Семафори в завданні взаємного виключення          --
-------------------------------------------------------------
with Ada.Synchronous_Task_Control,Text_IO,Integer_Text_IO;
use  Ada.Synchronous_Task_Control,Text_IO,Integer_Text_IO;
procedure Lab112 is

   Буфер: integer:= 10;          -- спільний ресурс

   Sem_Ku: Suspension_Object;    -- семафор

   procedure Старт_Задач is

      task A;

      task body A is
      begin
         put_line("Process A started");

         -- Операція зі спільним ресурсом
         Suspend_Until_True(Sem_Ku);
            Буфер := Буфер + 2;  -- критична ділянка
         Set_True(Sem_Ku);

         put_line("Process A finished");
      end A;

      ------------------------------------------------
```



```
      task B;

      task body B is
      begin
         put_line("     Process B started");

         -- Операція зі спільним ресурсом
         Suspend_Until_True(Sem_Ku);
            Буфер:= Буфер - 25;     -- критична ділянка
         Set_True(Sem_Ku);

         put_line("     Process B finished");
      end B;

   begin

      null;

   end Старт_Задач;

-- основна процедура
begin

   put_line("  Main procedure started ");

   Set_True(Sem_Ku); -- начальне значення семафора  (true)

   Старт_Задач;      -- запуск задач A і B

end Lab112;
```

Захищені модулі. Розглянемо використання захищеного модуля для вирішення завдання синхронізації процесів, що було розглянуто в попередньому прикладі [22].

Структуру захищеного модуля Захист показано на рис. 11.1. Модуль включає захищений елемент – змінну Буфер, що буде описана в приватній частині специфікації захищеного модуля. Для реалізації взаємовиключних дій над спільним ресурсом Буфер наведено дві процедури Операція_A() і Операція_B().

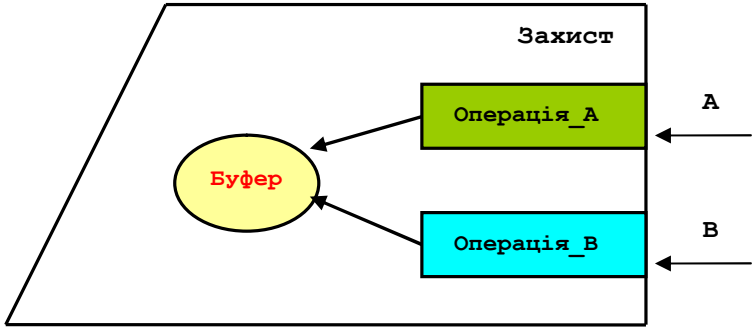


Рис. 11.1. Захищений модуль в завданні взаємного виключення

❖ Приклад 11.3

```
------------------------------------------------------------
-- Ada.Захищений модуль в                                 --
-- завданні взаємного виключення                          --
------------------------------------------------------------
procedure Lab113 is
   -- захищений модуль
   protected Захист is
      procedure Операція_А;
      procedute Операція_В;
   private
      Буфер : integer:= 10;       -- спільний ресурс
   end Захист;

-- тіло захищеного модуля
   protected  body Захист is

      procedure Операція_А is
      begin
         Буфер := Буфер + 2;
      end Операція_А;

      procedure Операція_В is
      begin
         Буфер := Буфер - 25;
      end Операція_В;
```



```
   end Захист;

   -- використання захищеного модуля в задачах
   task A;

   task body A is
   begin
         .  .  .
      -- дії над спільним ресурсом
      Захист.Операція_А;
       .  .  .
   end A;
   -------------------------------------
   task B;

   task B is
   begin
        .  .  .
      -- дії над спільним ресурсом
      Захист.Операція_В;
       .  .  .
   end B;

-- головна процедура
begin

   null;

end Lab113;
```

11.2. Вирішення завдання взаємного виключення в WinAPI

Бібліотека WinAPI містить декілька засобів, які можна використати для вирішення завдання взаємного виключення. Це реалізація механізму семафорів, мютекси та критичні секції (табл. 11.1) [2]. У наступних підрозділах розглядаються приклади вирішення завдвння взаємного виключення для трьох задач `T1`, `T2` і `T3` за допомогою засобів WinAPI. Кожна задача бере участь у формуванні спільної змінної `Ресурс` через додавання значення. Операція над змінною `Ресурс` розглядається в кожному процесі як критична ділянка, яку треба захищати від одночасного входження в неї проце-

сів за допомогою “створювання огорожі” з належних операцій WinAPI, які виконують функції примітивів ВХІДКД і ВИХІДКД.

Під час використання в наступних прикладах функцій WinAPI для параметрів, які прямо не стосуються керування процесами, умовно встановлено значення NULL.

Таблиця 11.1. Засоби взаємного виключення WinAPI

Об’єкт		Опис
Семафор	Semaphore	Містить лічильник від нуля до визначеного значення, які визначають кількість потоків, що мають доступ до спільного ресурсу
Мютекс	Mutex	Цим об’єктом може володіти тільки один процес у поточний момент часу, що дозволяє цьому процесу ексклюзивний доступ до спільного ресурсу
Критична секція	Critical Section	Містить критичну ділянку, вхід у яку дозволяється тільки одному процесу
Функції очікування	WaitForSingleObject, WaitForMultipleObject	Перевіряють умову, залежно від якої процес може увійти в критичну ділянку. Якщо це неможливо, то процес блокується і чекає на її звільнення, після чого входить в критичну ділянку

Семафори. У разі використанні семафорів роль примітивів ВХІДКД і ВИХІДКД виконують функції WaitForSingleObject() та ReleaseSemaphore().

❖ Приклад 11.4

```
-------------------------------------------------------------
--|Використання семафорів бібліотеки  WinAPI              --
--|в завданні взаємного виключення                        --
-------------------------------------------------------------
With Win32, Win32.winbase, Win32.winnt;
Use  Win32, Win32.winbase, Win32.winnt;
procedure Lab114 is
```



```
   Ресурс: integer;  -- спільний ресурс
                     -- (глобальна змінна)

  -- HANDLE  змінна для створювання семафора
  Sem1: HANDLE;

  -- допоміжні змінні
  Temp: DWORD;
  p1: Boolean;

  procedure Запуск_Задач is

     task T1;
     task T2;
     task T3;

     task body T1 is
        X1: integer;      -- локальна змінна
     begin
        put_line("Process T1 started");
             .  .  .
        --  Критична ділянка
        Temp:= WaitForSingleObject(Sem1,Infinite);
           Ресурс := Ресурс + X1;
        p1:= ReleaseSemaphore(Sem1,1,NULL);
             .  .  .
        put_line("Process T1 finished");
     end T1;
     ----------------------------------------
     task body T2 is
        X2: integer;      -- локальна змінна
     begin
        put_line("Process T2 started");
             .  .  .
        --  Критична ділянка
        Temp:= WaitForSingleObject(Sem1,Infinite);
           Ресурс := Ресурс + X2;
        p1:= ReleaseSemaphore(Sem1,1, NULL);
             .  .  .
        put_line("Process T2 finished");
     end T2;
     ----------------------------------------
     task body T3 is
        X3: integer;      -- локальна змінна
     begin
        put_line("Process T3 started");
             .  .  .
```

```
        --  Критична ділянка
        Temp:= WaitForSingleObject(Sem1,Infinite);
           Ресурс := Ресурс + X3;
        p1:= ReleaseSemaphore(Sem1,1,NULL);
             .  .  .
        put_line("Process T3 finished");
     end T3;
  begin
     null;
  end Запуск_Задач;

begin -- основна процедура
  put_line("  Main procedure started ");

  --  створення семафора з початковим значенням 1
  Sem1:=  CreateSemaphore(NULL,0,1,NULL);

     -- виклик процедури для запуску задач
  Запуск_Задач;

end Lab114;
```

Мютекси. Мютекс (від *mutual exclusion* – взаємне виключення) – засіб, схожий на семафор, але на відміну від семафора він не має значення. Мютекс належить тільки одному процесу і може передаватися від одного процесу до іншого (бути захопленим процесом). Вхід в критичну ділянку дозволяється тільки тому процесу, який теперь володіє мютексом. Після виходу з критичної ділянки процес звільнює мютекс, який захвачується іншим процесом.

Створення мютексу виконується зі змінною типу `HANDLE` за допомогою операцій `CreateMutex()`, `OpenMutex()`, `InitMutex()`.

У разі використанні мютексів роль примітивів `ВХІДКД` і `ВИХІДКД` можуть виконувати функції `WaitForSingleObject()` та `ReleaseMutex()`.

❖ Приклад 11.5

```
-------------------------------------------------------------
--|Використання мютексів у бібліотеці  WinAPI              --
--|в завдвнні взаємного виключення                         --
-------------------------------------------------------------
procedure Lab115 is
```



```
  Ресурс: integer;  -- спільний ресурс (глобальна змінна)

-- HANDLE змінна для створення мютексу
Mute: HANDLE;

-- допоміжні змінні
Temp: DWORD;
p1: Boolean;

procedure Запуск_Задач is

   task T1;
   task T2;
   task T3;

   task body T1 is
      X1: integer;      -- локальна змінна
   begin
      put_line("Process T1 started");
           .  .  .
      --  Критична  ділянка
      Temp:= WaitForSingleObject(Mute,Infinite);
         Ресурс:= Ресурс + X1;    -- КД
      p1:= ReleaseMutex(Mute);
           .  .  .
      put_line("Process T1 finished");

   end T1;
   ------------------------------------------
   task body T2 is
      X2: integer;      -- локальна  змінна
   begin
      put_line("Process T2 started");
           .  .  .
      --  Критична ділянка
      Temp:= WaitForSingleObject(Mute,Infinite);
         Ресурс:= Ресурс + X2;    -- КД
      p1:= ReleaseMutex(Mute);
           .  .  .
      put_line("Process T2 finished");

   end T2;
   ------------------------------------------------
   task body T3 is
      X3: integer;      -- локальна  змінна
   begin
      put_line("Process T3 started");
```

```
            .  .  .
         --  Критична ділянка
         Temp:= WaitForSingleObject(Mute,Infinite);
            Ресурс:= Ресурс + X3;
         p1:= ReleaseMutex(Mute);
            .  .  .
         put_line("Process T3 finished");
      end T3;
   begin

      null;
   end Запуск_Задач;

begin -- основна процедура

   put_line("  Main procedure started ");

   --  створення мютекса
   Mute:= CreateMutex(NULL,0,NULL);

   -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab115;
```

Критичні секції. Механізм *критичних секцій* – це засіб, який об'єднує конструкції ВХІДКД і ВИХІДКД в *єдиний* оператор.

Створення критичної секції виконується зі змінною типу CRITICALSECTION за допомогою операцій Initialize CriticalSection(), знищення – за допомогою операції DeleteCriticalSection().

У разі використання критичних ділянок роль примітивів ВХІДКД і ВИХІДКД виконують функції EnterCriticalSection() та LeaveCriticalSection(). На відміну від семафорів і мютексів у механізмі критичних ділянок час очікування в операції входження в критичну ділянку EnterCritical Section() не встановлюється, тобто він завжди є INFINITE.

❖ Приклад 11.6

```
------------------------------------------------------------
--|Використання критичних секцій бібліотеки WinAPI       --
--|у задачі взаємного виключення                          --
```



```
------------------------------------------------------------
procedure Lab116 is

   Ресурс: integer;  -- спільний ресурс
                     -- (глобальна змінна)

   -- створювання критичної секції
   CrCec: CRITICAL_SECTION;

   procedure Запуск_Задач is

      task T1;
      task T2;
      task T3;

      task body T1 is
         X1: integer;          -- локальна змінна
      begin
         put_line("Process T1 started");
            .  .  .
         --  Критична ділянка
         EnterCriticalSection(CrSec);
            Ресурс:= Ресурс + X1;   -- КД
         LeaveCriticalSEction(CrSec);
            .  .  .
         put_line("Process T1 finished");
      end T1;
      ---------------------------------------
      task body T2 is
         X2: integer;            -- локальна змінна
      begin
         put_line("Process T2 started");
            .  .  .
         --  Критична ділянка
         EnterCriticalSection(CrSec);
            Ресурс:= Ресурс + X2;  -- КД
         LeaveCriticalSEction(CrSec);
            .  .  .
         put_line("Process T2 finished");
      end T2;
      -------------------------------------
      task body T3 is
         X3: integer;            -- локальна змінна
      begin
         put_line("Process T3 started");
            .  .  .
         --  Критична ділянка
```

```
            EnterCriticalSection(CrSec);
               Ресурс := Ресурс + X3;    -- КД
            LeaveCriticalSEction(CrSec);
                 .  .  .
            put_line("Process T3 finished");
        end T3;
    begin

       null;

    end Запуск_Задач;

begin      -- основна процедура

    put_line("  Main procedure started ");

    --  створення критичної секції
    InitializeCriticalSection(CrSec);

    -- виклик процедури для запуску задач
    Запуск_Задач;

end Lab116;
```

11.3. Вирішення завдання взаємного виключення в мові Java

Вирішення завдання взаємного виключення в Java програмах виконується за допомогою семафорів, критичних секцій (синхронізованих блоків та замків) [5, 23].

Семафори. Реалізовані за допомогою класу `Semapore` з пакету `Java.util.concurrent`. Семафори – множні, значення яких задаються при створенні семафора:

```
Semapore S1 = new Semaphore(1);  -- бінарний
Semapore S2 = new Semaphore(4);
```

Методи класу:

- `acquire()` реалізація операції P(S)
- `release()` реалізація операції V(S)

❖ **Приклад 11.7**

```
------------------------------------------------------------
--|Java.Семафори в завданні взаємного виключення
```



```
------------------------------------------------------------

import java.util.concurrent.*;

class Lab117 {

    public static void main(String args[]) {

        Semaphore Sm = new Semaphore(1);

        new T1(Sm);
        new T2(Sm);

    }
}// Lab117

class SharedResource {

    static int SR = 0;  // Спільний ресурс

}

// ------------------------------

class T1 implements Runnable {

    Semaphore S;

    T1(Semaphore s) {
        S = s;
        new Thread(this).start();
    }

    public void run() {

        System.out.println("Потік T1 старт");
                    .  .  .
            S.acquire();
               SharedResourse.SR++;   // КД
            S.release();

         System.out.println("Потік T1 фініш ");

    }
}// T1

// ------------------------------------------------
```

```
class T2 implements Runnable {

        Semaphore S;

    T1(Semaphore s,) {
        S = s;
        new Thread(this).start();
    }

    public void run() {

        System.out.println("Потік Т2 старт");
                    . . .
            S.acquire();
               SharedResource.SR++;    // КД
            S.release();
                    . . .
         System.out.println("Потік Т2 фініш ");

    }
}// Т2
```

Синхронізовані методи. Синхронізованим є метод, що має модифікатор `synchronized`. Якщо потік виконує синхронізований метод (знаходиться всередині метода), то всі інші потоки, які намагаються викликати будь-який синхронізований метод цього ж екземпляра класу, мають чекати, поки завершиться його виконання. Тобто синхронізований метод має властивості процедури монітора.

Приклад 11.7 ілюструє реалізацію Java монітора, який містить спільний ресурс – змінну `Буфер`, і два методи `Читати()` та `Писати()` для доступу до спільного ресурсу.

❖ Приклад 11.8

```
/*-----------------------------------------------------------
-- Java. Реалізація монітора для взаємного виключення   --
-----------------------------------------------------------*/
   class Монітор{

       private int Буфер; -- спільний ресурс

       synchronized int Читати(){
          return Буфер;
       }
```



```
       synchronized void Писати(int x){
          Буфер = x;
       }
   }// Монітор
```

Одночасне виконання методів `Читати()` та `Писати()` одного екземпляра класу `Монітор` неможливо. Якщо процес (потік) `B` викликав і виконує, наприклад, з екземпляра `Блок` класу `Монітор` метод `Блок.Читати()`, то інший потік `A` буде блокований, якщо він спробує викликати будь-який метод класу `Монітор`, наприклад, `Блок.Писати()`, і зможе продовжити своє виконання тільки після завершення виконання методу `Блок.Читати()` у потоці `B`.

Синхронізовані блоки. Загальна форма блоку `synchronized`:

```
synchronized(Об'єкт) {

     //  оператори ( тіло блоку)

},
```

де `Об'єкт` – це посилання на об'єкт, доступ до якого треба синхронізувати.

Синхронізований блок гарантує, що виконання тіла блоку, який пов'язаний з одним параметорм `Об'єкт`, буде виконуватися в режимі взаємного виключення.

❖ Приклад 11.9

Застосування синхронізованого блоку для взаємовиключного виклику методу `Зміна_Значення` з класу `Робота` в потоках – екземплярах класу `Доступ`.

```
/*-----------------------------------------------------------
-- Java. Застосування синхронізованого блоку            --
-----------------------------------------------------------*/
   class Робота{

      double Зміна_Значення(double e){
         return e*e;
```

```
    }
}// Робота

class  Доступ  implements Thread{

   Робота Блок1;   // посилання на клас Робота

   // конструктор
   Доступ(Блок  о){
      Блок1 = о;
   }

   public void run(){

      double x = 5;
      double y;

      //  синхронізований блок
      synchronized(Блок1) {
           y = Блок1.Зміна_Значення(x);
      }
}// Досту

class Основний{

   public static void main(String [] args){

      Робота Рб  = new Робота();
      Доступ  Д1 = new Доступ(Рб);
      Доступ  Д2 = new Доступ(Рб);

      Д1.start();
      Д2.start();

   }// main
}// Основний
```

Інтерфейс *Lock*

Інтерфейс `Lock` існує з Java 1.5. Він визначений у пакеті `java.util.concurrent.lock` і забезпечує широкий спектр операцій для рішення завдання взаємного виключення..

Загальна форма використання:


```
Lock Lck = ReentrantLock();

Lck.lock();

   try {

       // критична ділянка

         .  .  .

   } finally {

Lck.unlock();


}
```

Методи Lock і ReentrantLock

Інтерфейс Java Lock містить такі основні методи:

```
lock()
lockInterruptibly()
tryLock()
tryLock(час очікування, TimeUnit timeUnit)
unlock()
```

Клас Java ReentrantLock також має кілька цікавих публічних методів:

```
getHoldCount()
getQueueLength()
hasQueuedThread(потік)
hasQueuedThreads()
isFair()
isHeldByCurrentThread()
isLocked()
```

Розглянемо кожен із цих методів більш детально:

lock()

Метод `lock()` блокує екземпляр Lock, якщо це можливо. Якщо екземпляр `Lock` уже заблоковано, потік, що викликає `lock()`, блокується, доки `Lock` не буде розблоковано.

lockInterruptibly()

Метод lockInterruptibly() блокує блокування, якщо потік, що викликає цей метод, не було перервано. Крім того, якщо потік заблоковано, очікуючи на блокування `Lock` за допомогою цього методу, і він переривається, він виходить із викликів цього методу.

tryLock()

Метод `tryLock()` намагається негайно заблокувати екземпляр `Lock`. Він повертає `true`, якщо блокування вдалось, і false, якщо `Lock` уже заблоковано. Цей метод ніколи не блокується.

trylock(тривалий час очікування, TimeUnit timeUnit)

Метод `tryLock(long timeout, TimeUnit timeUnit)` працює як метод `tryLock()`, за винятком того, що він очікує завершення заданого часу очікування, перш ніж припинити спробу заблокувати блокування.

unlock()

Метод `unlock()` розблоковує екземпляр `Lock`. Як правило, реалізація блокування дозволяє лише потоку, який заблокував блокування, викликати цей метод. Інші потоки, які викликають цей метод, можуть призвести до неперевіреного винятку `(RuntimeException)`.

getHoldCount()

Метод Java `ReentrantLock getHoldCount`() повертає кількість разів, коли даний потік блокував цей екземпляр `Lock`. Потік може блокувати блокування кілька разів через повторний вхід блокування.

getQueueLength()

Метод `ReentrantLock getQueueLength()` повертає кількість потоків, які очікують на блокування блокування.

hasQueuedThread()

Метод `ReentrantLock hasQueuedThread(Thread thread)` приймає `Thread` як параметр і повертає true, якщо цей `Thread` стоїть у черзі в очікуванні блокування `Lock, i false,` якщо ні.

hasQueuedThreads()

Метод `ReentrantLock hasQueuedThreads()` повертає true, якщо будь-які потоки стоять у черзі, очікуючи на блокування цього блокування, і `false`, якщо ні.

isFair()

Метод `ReentrantLock isFair`() повертає true, якщо це блокування гарантує справедливість серед потоків, які очікують його блокування, і `false,` якщо ні. Щоб отримати додаткові відомості про справедливість блокування, див.

isHeldByCurrentThread()

Метод `ReentrantLock isHeldByCurrentThread()` повертає true, якщо блокування утримується (заблоковано) потоком, який викликає `isHeldByCurrentThread()`, і `false,` якщо ні.

д

isLocked()

Метод `ReentrantLock isLocked()` повертає true, якщо блокування наразі заблоковано, і `false`, якщо ні.

Java Lock і synchronized

Відмінності між Java `Lock` і `synchronized`:

1. Java Lock API забезпечує більшу видимість і параметри для блокування, на відміну від синхронізації, де потік може закінчитися нескінченним очікуванням блокування, ми можемо

використовувати `tryLock()`, щоб переконатися, що потік чекає лише певний час.

2. Код синхронізації набагато чистіший і його легко підтримувати, тоді як із `Lock` ми змушені мати блок `try-finally`, щоб переконатися, що `Lock` звільнено, навіть якщо між викликами методів `lock()` і `unlock()` виникає якийсь виняток.

3. Блоки або методи синхронізації можуть охоплювати лише один метод, тоді як ми можемо отримати блокування в одному методі та звільнити його в іншому методі за допомогою `Lock` API.

4. Ключове слово synchronized не забезпечує справедливості, тоді як ми можемо встановити справедливості на true під час створення об'єкта `ReentrantLock`, щоб потік, який довго чекав, отримував блокування першим.

5. Ми можемо створювати різні умови для `Lock` і різні потоки можуть `await()` для різних умов.

➢ **Запитання для модульного контролю**

1. Засоби рішення завдання взаємного виключення в мові Ada ?
2. Засоби рішення завдання взаємного виключення в мові Java ?
3. Засоби рішення завдання взаємного виключення в WinAPI?
4. Особливості реалізації семафорів в мові Ада ?
5. Ada. Призначення прагми Atomic_Components ?
6. Порівняння в Java інтерфейсу Lock і модифікатору synchronized?
7. Особливості реалізації семафорів в мові Java ?
8. WinAPI. Реалізація мютексів ?
9. WinAPI. Реалізація критичних секцій?
10. Як реалізовано механізм критичних секцій в Java ?

ЛЕКЦІЯ 12
Завдання синхронізації потоків

12.1. Вирішення завдання синхронізації потоків в мові Ада

Для вирішення завдання синхронізації в мові Ada можна застосувати механізм семафорів, бар'єрів, а також захищені модулі [5].

Застосування семафорів. Розглянемо приклад, у якому задача `A` чекає на подію, яку має містити задачі `B`, наприклад, введення даних (змінної `x`). Для синхронізації процесів з уведення використовуємо семафор `СігналПроПодію` з початковим значенням `false`, яке встановлюється автоматично під час створювання семафора. Очікування події в процесі `A` реалізовано за допомогою операції `Suspend_Until_True(СігналПроПодію)`, а посилання сигналу про подію в задачі `B` реалізовано за допомогою операції `Set_True(СігналПроПодію)`.

❖ **Приклад 12.1**

```
------------------------------------------------------------
--Ada.Синхронізація задач. Семафори                       --
------------------------------------------------------------
procedure Синхронізація is

   X : integer;  --  глобальна  змінна

   СігналПроПодію: Suspension_Object;        -- семафор

    -- задача, що чекає на подію
    task A;
```

```
     task body A is
     begin
             .  .  .
        -- точка очікування  події
        Suspend_Until_True(СігналПроПодію);
             .  .  .
     end A;

     -- задача, де  відбувається  подія
     task B;
     task body B is
     begin
             .  .  .
        get(X); -- уведення даних(подія, на
                -- яку чекає задача A)
        Set_True(СігналПроПодію);     -- сигнал задачі A
             .  .  .
     end B;
begin

     null;

end Синхронізація;
```

Застосування захищеного модуля. Розглянемо застосування захищеного модуля для вирішення завдання синхронізації задач `A` і `B`, що поданої у попередньому прикладі. Синхронізація задач реалізується за допомогою захищеного модуля `Контроль`, у якому визначені дві захищені операції: вхід `Чекати_Введення()` і процедура `СигналПроВведення()`, а також допоміжна захищена змінна (прапор) `F1`, яка буде використовуватись у бар'єрі захищеного входу. Структуру захищеного модуля `Контроль` подано на рис. 12.1.

Очікування на подію (введення `X`) у задачі `A` виконано як виклик входу `Чекати_Уведення()`, а посилання сигналу про подію в задачі `B` – як виклик процедури `СигналПроУведення()`, яка змінює значення прапору `F1` на `1`. Вхід `Чекати_Уведення()` має бар'єр, який закритий, якщо значення прапора `F1=0`, і відкритий, якщо `F1=1`. Таким чином, початкове значення прапора `F1=0` дозволяє блокувати процес `A` в операції виклику входу `Чекати_Уведення()` до того часу, поки процес `B` не змінить значення

бар'єра за допомогою операції `СигналПроУведення()` після того, як подія буде мати місце.

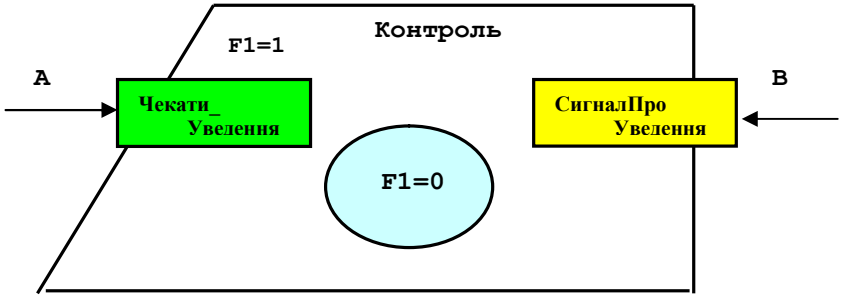


Рис. 12.1. Захищений модуль в завданні синхронізації

❖ Приклад 12.2

```
-------------------------------------------------------------
--  Ada. Синхронізація задач.Захищений модуль              --
-------------------------------------------------------------
-- захищений модуль
  protected  Контроль is

     entry  Чекати_Уведення;
     procedure СигналПроУведення;
  private

     F1 : integer:= 0;  -- прапор керування входом

  end Контроль;

  -- тіло захищеного модуля
  protected  body Контроль is

     entry  Чекати_Уведення  when  F1=1 is
     begin
         null;
     end Чекати_Уведення;

     procedure СигналПроУведення is
     begin
```

```
        Fl := 1;
    end СигналПроУведення;

end Контроль;

-- використання захищеного модуля в задачах
task A;
task B;

task body A is
begin
        .  .  .
    -- точка синхронізації(очікування уведення)
    Контроль.Чекати_Уведення;
        .  .  .
end A;

task body  B is
begin
        .  .  .
    -- сигнал про подію
    Контроль.СигналПроУведення;
        .  .  .
 end B;
```

Для синхронізації задачі A з кількома задачами (B,C, D), де виконується введення, необхідна модифікація захищеного модуля Контроль, яка пов'язана з тим, що бар'єр входу Чекати_Уведеня буде мати вигляд when Fl = 3, а процедура СигналПроУведення буде змінювати прапор Fl:= Fl + 1. Тепер задача A під час виклику входу Чекати_Уведення продовжить своє виконання тільки після того, як кожна задача B,C і D викличе процедуру СигналПроУведення і прапор Fl набуде значення 3.

12.2 Вирішення завдання синхронізації процесів в WinAPI

Завдання синхронізації в WinAPI можна вирішити за допомогою семафорів, подій або таймерів. При цьому використовуються функції очікування (табл. 12.2).

Таблиця 12.1. Засоби синхронізації у WinAPI

Об'єкт		Опис
Подія	Event	Повідомляє одному або кільком потокам, що чекають, про подію
Семафор	Semaphore	Містить лічильник від нуля до визначеного значення, який за значення нуль блокує процес, а зі зміною значення розблокує процес
Таймер	Timer	Визначає один або кілька потоків, що очікують настання визначеного часу
Функції очікування	WaitForSingleObject, WaitForMultipleObject	Перевіряють умову залежно від якої процес блокується і чекає на подію. Якщо подія вже відбулася, то процес не блокується

Застосування семафорів. У разі використання семафорів очікування події виконується за допомогою функції WaitForSingle Object(), а сигнал про подію, що відбулася, – через функцію ReleaseSemaphore().

У прикладі 12.3 задача T1 інформує задачу T2 про подію, що відбулася в задачі T1 (Подія1), а задача T3 чекає на подію, що відбудеться в задачі T2 (Подія2).

❖ Приклад 12.3

```
------------------------------------------------------------
--|Використання семафорів бібліотеки  WinAPI              --
--|в завданні синхронізації                                --
------------------------------------------------------------
procedure Lab123 is

   -- HANDLE  змінні для створювання семафорів
   Sem12, Sem23: HANDLE;

   -- допоміжні змінні
   Temp: DWORD;
   p1, p2: Boolean;
```

```
   procedure Запуск_Задач is

      task T1;
      task T2;
      task T3;

      task body T1 is
      begin
         put_line("Process T1 started");
              .  .  .
         -- Подія1
         --  Сигнал задачі T2 про подію (Подія1)
         --  у задачі T1
         p1:= ReleaseSemaphore(Sem12,1,NULL);
              .  .  .
         put_line("Process T1 finished");
      end T1;
      ----------------------------------------
      task body T2 is
      begin
         put_line("Process T2 started");
              .  .  .
         --  очікування на подію (Подія1) в T1
         Temp:= WaitForSingleObject(Sem12,Infinite);
                   .  .  .
         -- Подія2
         -- Сигнал задачі T3 про подію (Подія2) в задачі T2
         p1:= ReleaseSemaphore(Sem23,1,NULL);
              .  .  .
         put_line("Process T2 finished");
      end T2;
      --------------
      task body T3 is
      begin
         put_line("Process T3 started");
              .  .  .
         --  Очікування на подію (Подія2) в T2
         Temp:= WaitForSingleObject(Sem23,Infinite);
              .  .  .
         put_line("Process T3 finished");
      end T3;
   begin
      null;
   end Запуск_Задач;

begin -- основна процедура
```



```
   put_line("  Main procedure started ");

   --  створення семафорів з початковим значенням 0
   Sem12:=  CreateSemaphore(NULL, 0,1,NULL);
   Sem23:=  CreateSemaphore(NULL, 0,1,NULL);

   -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab123;
```

У прикладі 12.4 задача T3 чекає на події, що відбудуться в задачі T1 (Подія1) і задачі T2 (Подія2).

❖ Приклад 12.4

```
------------------------------------------------------------
--|Використання семафорів бібліотеки  WinAPI              --
--|у завданні синхронізації                               --
------------------------------------------------------------
procedure Lab124 is  317

   -- HANDLE  змінні для створювання семафорів
   Sem1_3, Sem2_3: HANDLE;

       -- допоміжні змінні
   Temp: DWORD;
   p1, p2: Boolean;
   procedure Запуск_Задач is

      task T1;
      task T2;
      task T3;

      task body T1 is
      begin
         put_line("Process T1 started");
              .  .  .
         -- Подія1
         -- Сигнал задачі T3 про подію (Подія1) в задачі T1
         p1:= ReleaseSemaphore(Sem1_3,1,NULL);
              .  .  .
         put_line("Process T1 finished");
      end T1;
      ---------------
      task body T2 is
      begin
```

```
         put_line("Process T2 started");
              .  .  .
         -- Подія2
         -- Сигнал задачі T3 про подію (Подія2) в задачі T2
         p1:= ReleaseSemaphore(Sem2_3,1,NULL);
              .  .  .
         put_line("Process T2 finished");
      end T2;
      --------------
      task body T3 is
      begin
         put_line("Process T3 started");
              .  .  .
         --  Очікування на подію (Подія1) в T1
         Temp:= WaitForSingleObject(Sem1_3,Infinite);

         --  Очікування на подію (Подія2) в T2
         Temp:= WaitForSingleObject(Sem2_3,Infinite);
              .  .  .
         put_line("Process T3 finished");
      end T3;

   begin
      null;
   end Запуск_Задач;

begin -- основна процедура

   put_line("  Main procedure started ");

   --  створення бінарних семафорів з початковими значеннями 0
   Sem1_3:=  CreateSemaphore(NULL, 0, 1,NULL);
   Sem2_3:=  CreateSemaphore(NULL, 0, 1,NULL);

   -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab317;
```

У прикладі 12.5 задача `T1` інформує задачі `T2` і `T3` про подію, що відбулася в задачі `T1`. Це можливо виконати через два бінарні семафори, але в прикладі це виконано за допомогою одного множинного семафора `Sem1_23`, який набуває значення `0 .. 2`.

❖ Приклад 12.5

```
-------------------------------------------------------------
--|Використання семафорів бібліотеки  WinAPI               --
--|в завданні синхронізації                                --
-------------------------------------------------------------
procedure Lab125 is

   -- HANDLE  змінна для створювання богатозначного семафора
   Sem1_23: HANDLE;

   -- допоміжні змінні
   Temp: DWORD;
   p1, p2: Boolean;

   procedure Запуск_Задач is

      task T1;
      task T2;
      task T3;

      task body T1 is
      begin
         put_line("Process T1 started");
              .  .  .
         -- Подія
         --  Сигнали задачам T2 і T3 про подію в задачі T1
         p1:= ReleaseSemaphore(Sem1_23,2,NULL);
              .  .  .
         put_line("Process T1 finished");
      end T1;
      ---------------
      task body T2 is
      begin
         put_line("Process T2 started");
              .  .  .
         --  Очікування на подію в T1
         Temp:= WaitForSingleObject(Sem1_23,Infinite);
              .  .  .
         put_line("Process T2 finished");
      end T2;
      --------------
      task body T3 is
      begin
         put_line("Process T3 started");
              .  .  .
         --  Очікування на подію в T1
         Temp:= WaitForSingleObject(Sem1_23,Infinite);
```

```
            .  .  .
         put_line("Process T3 finished");
      end T3;
   begin
      null;
   end Запуск_Задач;

begin -- основна процедура
   put_line("  Main procedure started ");

   --  створення семафора зі значеннями (0,1,2)  і
   --  початковим значенням 0
   Sem1_23:=  CreateSemaphore(NULL, 0, 2, NULL);

   -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab318;
```

Застосування подій. Механізм подій призначений виключно для синхронізації процесів.

У прикладі 12.6 задача T1 інформує задачу T2 про подію, що відбулася в задачі T1 (Подія1), а задача T3 чекає на подію, що відбудеться в задачі T2 (Подія2).

❖ Приклад 12.6

```
------------------------------------------------------------
--|Використання подій бібліотеки  WinAPI                  --
--|в завданні синхронізації                                --
------------------------------------------------------------
procedure Lab319 is

   -- HANDLE  змінна для створювання  подій
   Evn12, Evn23: HANDLE;

   -- допоміжні змінні
   Temp  : DWORD;
   p1, p2: BOOL;

   procedure Запуск_Задач is

      task T1;
      task T2;
      task T3;
```



```
      task body T1 is
      begin
         put_line("Process T1 started");
              .  .  .
         -- Подія1
         -- Сигнал задачі T2 про подію (Подія1)в задачі T1
         p1:= SetEvent(Evn12);
              .  .  .
         put_line("Process T1 finished");
      end T1;
      ---------------
      task body T2 is
      begin
         put_line("Process T2 started");
              .  .  .
         --  очікування на подію (Подію1) в T1
         Temp:= WaitForSingleObject(Evn12,Infinite);
                   .  .  .
         -- Подія2
         -- Сигнал задачі T3 про подію (Подія2)
         p1:= SetEvent(Evn23);
              .  .  .
         put_line("Process T2 finished");

      end T2;
      --------------
      task body T3 is
      begin
         put_line("Process T3 started");
              .  .  .
         --  Очікування на подію (Подія2) в T2
         Temp:= WaitForSingleObject(Evn23,Infinite);
              .  .  .
         put_line("Process T3 finished");
      end T3;
   begin
      null;
   end Запуск_Задач;

begin -- основна процедура
   put_line("  Main procedure started ");

   --  створення подій
   Evn12:=  CreateEvent(NULL, 0,0, NULL);
   Evn23:=  CreateEvent(NULL, 0,0, NULL);

   -- виклик процедури для запуску задач
```

```
   Запуск_Задач;

end Lab126;
```

У прикладі 12.7 задача T3 чекає на події, що відбудуться в задачі T1 (Подія1) і задачі T2 (Подія2). Для сигналізації про ці події застосовані події Evn1_3 та Evn2_3.

❖ Приклад 12.7

```
------------------------------------------------------------
--|Використання подій бібліотеки  WinAPI                  --
--|в завданні синхронізації                                --
------------------------------------------------------------
procedure Lab127 is

   -- HANDLE  змінна для створювання  подій
   Evn1_3, Evn2_3: HANDLE;

   -- допоміжні змінні
   Temp: DWORD;
   p1, p2: BOOL;

   procedure Запуск_Задач is

      task T1;
      task T2;
      task T3;

      task body T1 is
      begin
         put_line("Process T1 started");
              .  .  .
         -- Подія1
         --  Сигнал задачі T3 про подію (Подія1)
         p1:= SetEvent(Evn1_3);
              .  .  .
         put_line("Process T1 finished");
      end T1;
      ---------------
      task body T2 is
      begin
         put_line("Process T2 started");
              .  .  .
         -- Подія2
         -- Сигнал задачі T3 про подію (Подія2) в задачі T2
```



```
         p1:= SetEvent(Evn2_3);
              .  .  .
         put_line("Process T2 finished");
      end T2;
      --------------
      task body T3 is
      begin
         put_line("Process T3 started");
              .  .  .
         --  Очікування на подію (Подія1) в T1
         Temp:= WaitForSingleObject(Evn1_3,Infinite);

         --  Очікування на подію (Подія2) в T2
         Temp:= WaitForSingleObject(Evn2_3,Infinite);
              .  .  .
         put_line("Process T3 finished");
      end T3;
   begin
      null;
   end Запуск_Задач;

begin -- основна процедура
   put_line("  Main procedure started ");

   --  створення подій
   Evn1_3:=  CreateEvent(NULL, 0, 0, NULL);
   Evn2_3:=  CreateEvent(NULL, 0, 0, NULL);

   -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab127;
```

У прикладі 12.8 задача T1 інформує задачі T2 і T3 про подію, що відбулася в задачі T1. Це можливо виконати через одну подію Evn1_23, для якої обирається режим ручного скидання події.

❖ Приклад 12.8

```
------------------------------------------------------------
--|Використання подій бібліотеки  WinAPI                  --
--|в завданні синхронізації                                --
------------------------------------------------------------
procedure Lab321 is

      -- HANDLE  змінна для створювання події
```

```
Evn1_23: HANDLE;

-- допоміжні змінні
Temp: DWORD;
p1: BOOL;

procedure Запуск_Задач is

   task T1;
   task T2;
   task T3;

   task body T1 is
   begin
      put_line("Process T1 started");
          .  .  .
      --  Подія
          .  .  .
      --  Сигнал задачам T2 і T3 про подію в задачі T1
      p1:= SetEvent(Evn1_23);
          .  .  .
      put_line("Process T1 finished");

   end T1;
   ---------------
   task body T2 is
   begin
      put_line("Process T2 started");
          .  .  .
      --  Очікування на подію в T1
      Temp:= WaitForSingleObject(Evn1_23,Infinite);
          .  .  .
      put_line("Process T2 finished");

   end T2;
   --------------
   task body T3 is
   begin
      put_line("Process T3 started");
          .  .  .
      --  Очікування на подію в T1
      Temp:= WaitForSingleObject(Evn1_23,Infinite);
          .  .  .
      put_line("Process T3 finished");
   end T3;

begin
```



```
      null;
   end Запуск_Задач;

begin -- основна процедура
   put_line("  Main procedure started ");

   --  створення події з ручним скіданням
   Evn1_23:=  CreateEvent(NULL, 1, 0, NULL);

   -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab128;
```

12.3 Вирішення завдання синхронізації потоків в мові Java

Рішення завдання синхронізації в Java може бути здійснено за допомогою створення класу – монітору, або механізму бар'єрів.

Монітори. Синхронізація процесів в мові Java виконується за допомогою методів `wait()`, `notify()` або `notifyAll()`. У першій версії мови з цією метою також були використані методи `suspend()/resume()`, але згодом від них відмовились, оскілки вони спричинили побічні ефекти.

Методи `wait()`, `notify()` та `notifyAll()` об'явлені в класі `Object`, доступні всім класам і мають форму:

```
final void  wait()  throws  InterruptedException
final void  notify()
final void  notifyAll()
```

Виконання методу `A.wait()` у потоці зумовлює блокування потоку, поки інший процес не виконає метод `A.notify()` або `A.notifyAll()`.

Метод `notify()` розблокує потік, який блоковано методом `wait()`. На жаль, мова не визначає, який потік буде розблоковано, якщо потоків декілька. Метод `notifyAll()` розблокує *усі* потоки, які блоковані методом `wait()` в одному об'єкті. Якщо на момент виконання методів `notify()` і `notifyAll()` не існує блокованих потоків, то методи ігноруються. Тому, якщо треба методом `notify()` обов'язково повідомити процес про подію, яка відбула-

ся до того, як він буде її чекати, необхідно застосувати додаткову загальну змінну, значення якої треба перевіряти поряд з використанням методу `wait()`.

Звернімо увагу, що методи `wait()`, `notify()` і `notifyAll()` взаємодіють тільки тоді, коли вони прив'язані до одного і того ж об'єкта. Крім того, всі три методи можуть бути викликані тільки всередині синхронізованого методу.

❖ Приклад 12.9

Клас - монітор `MSynchro` містить два синхронізовані методи `ЧекатиНаПодію()` та `СигналПроПодію()`, за допомогою яких виконується синхронізація потоків: у потоці `Потік1` відбувається подія, на котру чекає потік `Потік2`.

```
/* ------------------------------------------------------
-- Java. Синхронізація  двох потоків                   --
------------------------------------------------------*/
class MSynchro {

   private int Прапор;     // ддодаткова змінна

   synchronized void ЧекатиНаПодію(){

      if(Прапор = = 0){
        try{
           wait();         // очікування події

        }catch(InterruptedException){
            System.out.println("Потік П1 завершений");
        }
      }
   }

   synchronized void СигналПроПодію(){

      Прапор ++;
      notify();            // розблокування потоку
   }

}//MSynchro

// ------------------------------------------------------
class Потік1 extends  Thread {
```



```
   Synchro O1;            // Посилання на клас Synchro
   // конструктор
   Потік1(MSynchro s){
      O1 = s;
   }
   public  void run(){

         . . .

      // подія
         . . .

      // сигнал про подію для Потоку2
      O1.СигналПроПодію();
         . . .

      System.out.println("Потік П1 завершений");

    }// run
   }// Потік1

// ------------------------------------------------------
class Потік2 extends  Thread {

   Synchro O2;         // Посилання на клас  Synchro

   // конструктор
   Потік2(Synchro s){
      O1 = s;
   }

   public void run(){
         . . .

      // очікування сигналу про подію від Потоку 1
      O2.ЧекатиНаПодію();

         . . .
      System.out.println("Потік П2 завершений");

   }// run

}// Потік2

class СинхронізаціяПотоків {
```

```
   public static void main(String args []){

      // екземпляр монітора
      MSynchro Снхр = new MSynchro();

      // екземпляри  потоків
      Потік1 П1 = new Потік1(Снхр);
      Потік2 П2 = new Потік2(Снхр);

      // старт потоків
      П1.start();
      П2.start();

   }// main

}// Синхронізація потоків
```

Бар'єри. Механізм бар'єрів в мові Java реалізований за допомогою класу **`CyclicBarrier`**. Ресурси класу дозволяют набору (групі) потоків очікувати один одного в точці, яка є бар'єром і містить виклик методу `await()`.

Клас `CyclicBarrier` має два конструктори:

```
- CyclicBarrier(int Кількість_потків)
-    CyclicBarrier(int Кількість_потків,Runnable дія).
```

Параметри конструкторів визначають кількість потоків очікування в групі, а також можливі дії (потік) по завершенню бар'єрної синхронізації.

Метод `await()` має дві форми:

```
- int await() throws InterruptedException,
                                BrokenBarrierException
- int await(long Очікування, TimeUnit Час)
         throws InterruptedException,
            BrokenBarrierException, TimeoutException
```

❖ Приклад 12.10

Потоки П2 і П3 чкають на введення даних в потоці П1. Використано бар'єр `Bar`, в точках синхронізації потоків розміщено метод `await()` класу `CyclicBarrier`.

```
/* -------------------------------------------------------
```



```
-- Java. Бар'єри. Синхронізація  трьох потоків          --
-------------------------------------------------------*/
class BSynchro {

  // створення барєру
  public static CyclicBarrier Bar = new CyclicBarrier(3);

  // -------------------------------------------------------
  class Потік1 extends  Thread {

     public  void run(){

           . . .

        // подія
           Введення_Даних();

        // Бар'єр  синхронізації по введенню даних
        Bar.await();

              . . .

        System.out.println("Потік П1 завершений");

    }// run
   }// Потік1

// -------------------------------------------------------
class Потік2 extends  Thread {

   public void run(){
         . . .


        // Бар'єр  по введенню даних в Т1
        Bar.await();

         . . .

      System.out.println("Потік П2 завершений ");

   }// run

}// Потік2


// -------------------------------------------------------
```

```
class Потік3 extends  Thread {

   public void run(){
         . . .

        // Бар'єр  по введенню даних в Т1
        Bar.await();

         . . .

      System.out.println("Потік П3 завершений ");

   }// run

}// Потік3

public static void main(String args []){

      // екземпляри  потоків
      Потік1 П1 = new Потік1();
      Потік2 П2 = new Потік2();

      // старт потоків
      П1.start();
      П2.start();

   }// main

}// BSynchro
```

- **Запитання для модульного контролю**
  1. Які засоби мови Ada можна використовувати для програмування синхронізації потоків?
  2. Мова Ada. Які складові захищеного модуля слід застосувати для ефективної синхронізації потоків?
  3 Мова Ada. Яка складова захищеного входу захищеного модуля дозволяє керувати синхронізацією кількома потоками?
  4. WinAPI. Засоби програмування синхронізації потоків ?
  5. WinAPI. Події. Ручне і автоматичне скидання події. Концепція і реалізація?

6. Багатозначні семафори в синхронізації потоків?
7. Концепція барєрів для синхронізації потоків ?
8. Реалізація механізму бар'єрів в мові Java?
9. Java. Засоби програмування синхронізації потоків ?
10. Java. Методи і конструктори класу `CyclicBarrier` ?

ЛЕКЦІЯ 13
Взаємодія потоків за допомогою повідомлень

Лекція присвячено питанням організації обчислень в комп'ютерних системах з розподіленою пам'яттю на підставі посилання повідомлень. Розглянуто загальну схему механізму повідомлень та її реалізації механізму повідомлень в мовах Оккам, бібліотеці PVM.

13.1. Загальна схема

Модель взаємодії потоків орієнтована на паралельні комп'ютерні системи з локальною пам'яттю (ПКС ЛП). В таких системах відсутня спільна пам'ять, тому неможлива організація взаємодії потоків через модель, що базується на спільних змінних Вузол ПКС ЛП містить процесор і локальну пам'ять, фізичні зв'язки між вузлами реалізовано з допомогою спеціальних ліній зв'язку (лінків) [2, 5, 21]. Потоки взаємодіють шляхом формування повідомлень і відправки їх через лінки (Рис. 13.1).

Загальна схема передачі повідомлень між потоками ґрунтується на використанні операцій `Send()` і `Receive()`.

За допомогою операції `Send()` потік відправляє дані іншому потоку, який отримує ці дані за допомогою операції `Receive()`:

```
Потік1                    Потік2
  . . .                     . . .
Send(Дані);               Receive(Дані);
```

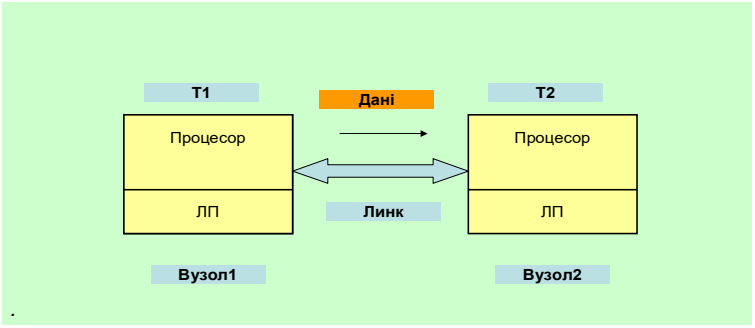



Рис. 13.1 Взаємодія потоків в ПКС ЛП

Існують різноманітні схеми використання операцій `Send()` і `Receive()`: іменовані, синхронні, асинхронні, з блокуванням або без блокування та ін.

Іменована схема потребує застосування імен взаємодіючих потоків в операціях `Send()` і `Receive()`:

```
Send    (Ім'я_Процесу_Одержувача,  Дані)
Receive(Ім'я_Процесу_Відправника, Дані)
```

У разі застосування непрямої схеми взаємодії потоків із використанням поштової скриньки, буфера, каналу, лінка або труби (pipe) треба вказувати ім'я засобу, що застосовується:

```
Send    (Буфер, Дані)
Receive(Буфер, Дані)
```

Схема, у якій вказані імена обох потоків – відправника і одержувача, має назву *симетричної. Асиметрична* схема потребує вка-

зати ім'я лише одного потоку, до якого звертаються. Асиметрична схема вигідна в моделі *клієнт–сервер*, коли потік–сервер не знає і не повинен знати, з яким потоком–клієнтом він буде взаємодіяти.

Існує кілька варіантів взаємодії потоків:

- один потік з кількома потоками;
- кількаа потоків з кількома потоками;
- кілька потоків з одним потоком;
- один потік з одним потоком (Рис. 13.2).

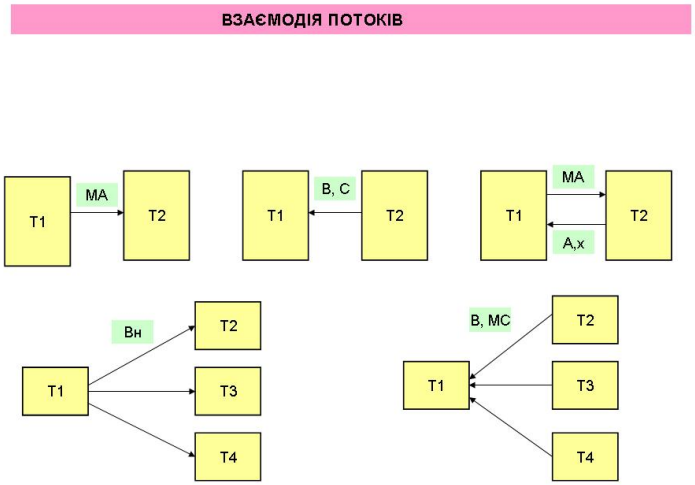


Рис. 13.2 Види взаємодії потоків

Передача повідомлень може бути також синхронною або асинхронною. Синхронна схема ґрунтується на тому, що потік, який виконує операцію `Send()` або `Receive()`, блокується, поки інший потік не вийде на приймання повідомлення (операція `Receive())` або отримання повідомлення (операція `Receive()`).

Оптимальною з погляду розробника є можливість використання різних видів операцій передавання-приймання повідомлень, як

це зроблено в бібліотеці PVM, яка містить набір з кількох видів операцій `Send()` і `Receive()`.

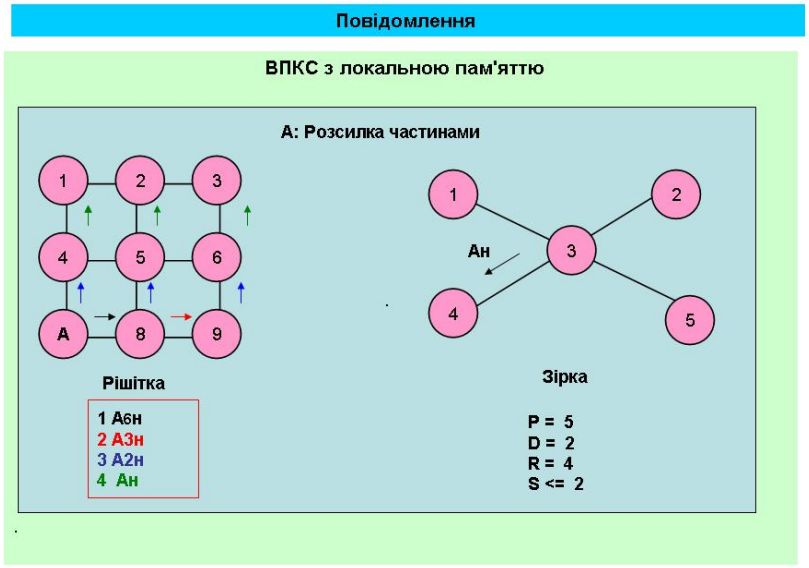


Рис.13.3 Розсилка частин вектора

13.2. Мова Оккам

Мову Оккам було створено для програмування в комп'ютерних системах, які будувались за допомогою *трансп'ютерів* – спеціальних мікропроцесорів, спроектованих виключно для використання в ПКС з локальною пам'яттю [19, 26].

Трансп'ютер має процесор, локальну пам'ять та засоби для зв'язку з іншими трансп'ютерами – вісім двонаправлених *лінків* (link) (Рис. 13.4). За допомогою лінків можлива побудова трансп'ютерних систем різноманітної структури з розподіленою пам'яттю (Рис 13.5).

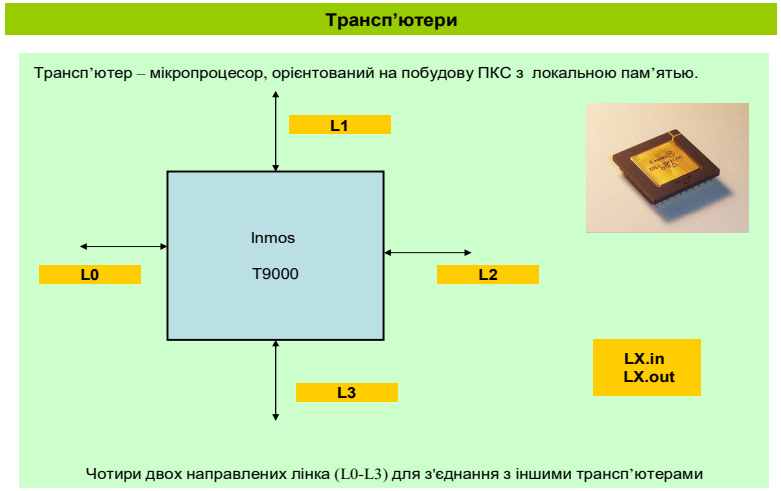


Рис.13.4 Структура трансп'ютера

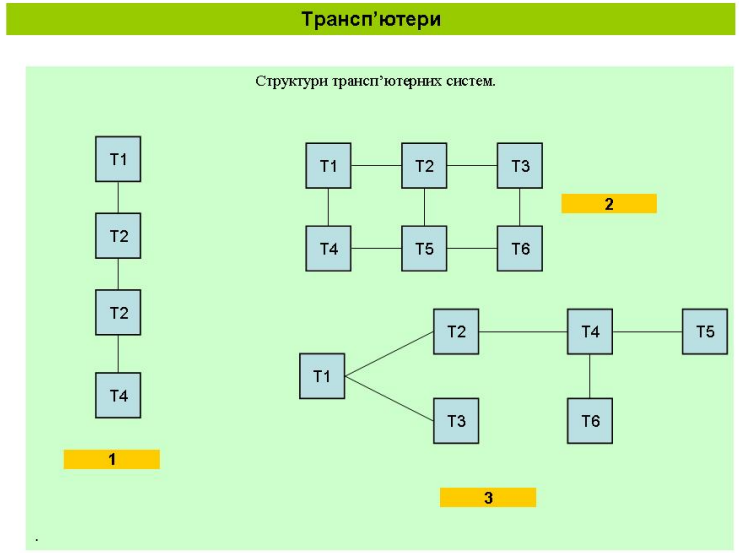


Рис.13.5 Структури трансп'ютерних систем

Оккам підтримує комунікацію і синхронізацію процесів через посилання повідомлень. Оккам реалізує варіант механізму рандеву, який на відміну від рандеву в мові Ada є симетричним.

У мові Оккам процеси не іменовані, тому їх взаємодія ґрунтується на використанні спеціальної програмної конструкції – *каналу* (`channel`). Канал має ім'я і може бути використаний для запису даних у канал (операція **!**) або читання даних з каналу (операція **?**):

```
Chan1!X   - писати  значення змінної X у канал  Chan1
Chan3?Y   - читати  значення змінної Y з каналу Chan3
```

Взаємодія процесів через канал (рис. 13.6) синхронізована згідно із загальною моделлю механізму рандеву. Процес, що готовий записати дані в канал `КаналЕ` (операція `КаналЕ!x`), чекає, поки другий процес зможе зчитати ці дані з каналу `КаналЕ` (операція `КаналЕ?y`) і навпаки.

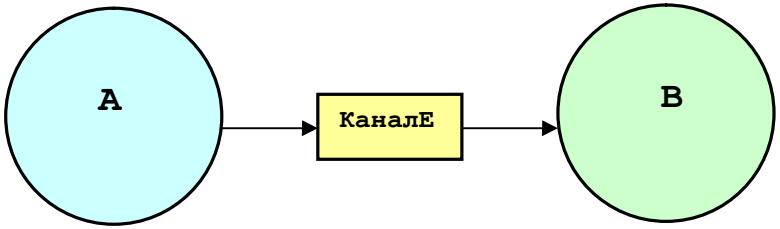


Рис. 13.6. Структура взаємодії задач в Оккам через канал

```
CHAN OF INT  КаналЕ:          --  оголошення каналу

PAR       --   перший процес
   INT x = 100:

   КаналЕ ! x     -- запис у канал

PAR       --  другий процес
```

```
INT y:
КаналЕ ? y     -- читання з каналу
```

Канал у мові Оккам визначається за допомогою типу CHAN. Для каналу необхідно визначити тип даних, які він буде пересилати. Канал КаналЕ може бути застосований для передачі змінних типу INT. Для більш складних типів даних канал потребує використання поняття *протоколу каналу.*

❖ **Приклад 13.1.** Протокол для передавання масивів різної довжини.

```
PROTOCOL Буфер IS INT::[]BYTE:
CHAN OF  Буфер КаналХ:
[8] BYTE Дані:
INT Розмір:
PAR
   КаналХ ! 5::"HELLO"
   КаналХ ? Розмір::Дані:
```

Протокол Буфер визначає протокол для передавання масиву елементів типу BYTE, розмір якого не встановлений і може змінюватися з кожним використанням через задавання лічильника. Протокол застосовано при опису каналу КаналХ. В прикладі 13.1 протокол Буфер використано для запису в канал і зчитування з нього. Використання каналу з цим протоколом. Для запису в канал це значення встановлено в 5 і для читання значення лічильника спочатку приймається в змінну Розмір.

❖ **Приклад 13.2.** Протокол для передавання даних різного типу.

```
PROTOCOL Порт IS INT;BYTE:
CHAN OF  Порт Регістр:
INT  Переривання:
BYTE Адреса:
PAR
   Регістр ! 500,'01F0A6'
   Регістр ? Переривання, Адреса
```


У мові Оккам також існує протокол з варіаціями, який дозволяє передавати по одному каналу повідомлення різного формату.

Розроблення програми мовою Оккам включає етап конфігурування, який пов'язаний з розподіленням процесів по трансп'ютерах (процесорам), а каналів – по лінках. Для цього використовують спеціальні оператори:

```
PLACED PAR
PROCESSOR номер тип.процесора
PLACE канал AT адрес
```

Застосування операторів конфігурування наведено в прикладі 13.3. Три задачі виконують відповідно три процедури PA, PB і PC і передають результати їх виконання одна одній. Розподіл процесів по трансп'ютерах показано на рис. 13.7.

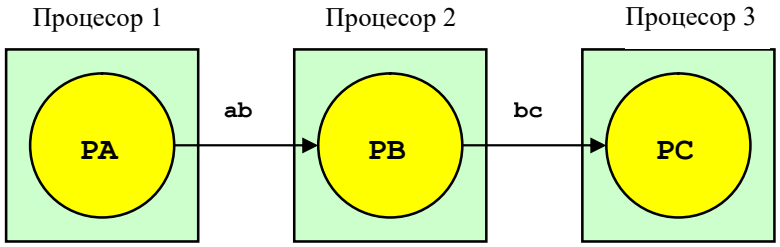


Рис. 13.7. Структура взаємодії процесів в Оккам

Взаємодія процесів здійснюється за допомогою каналів ab і bc, які використовують протокол Обмін. Для з'єднання процесорів 1 і 2 використано лінк-зв'язки link3Out і link0in, а для з'єднання процесорів 2 і 3 – лінк-зв'язки link3Out і link0in.

❖ **Приклад 13.3**

```
--------------------------------------------------------------------------------
-- Оккам. Взаємодія трьох процесів                                            --
--------------------------------------------------------------------------------
PROTOCOL Обмін IS INT;INT:
CHAN OF  Обмін ab:
```

```
CHAN OF  Обмін bc:
PROC PA(CHAN OF Обмін Лінія )
   INT x:
   SEQ
      x = 100
      Лінія ! x
PROC PB(CHAN OF Обмін Лінія1, Лінія2)
   INT y,z:
   SEQ
      z = 25
      Лінія1 ? y
      z = z + y
      Лінія2 ! z
PROC PC(CHAN OF Обмін Лінія))
   INT e,g:
   SEQ
      g = 8
      Лінія ? e
      g = g + e

PLACED PAR

   PROCESSOR 1 T800
     PLACE AB AT link30ut
     PA(ab)

   PROCESSOR 2 T800
     PLACE AB AT link30in
     PLACE BC AT link30ut
     PB(ab,bc)

   PROCESSOR 2 T800
     PLACE BC AT link0in
     PC(bc)
```

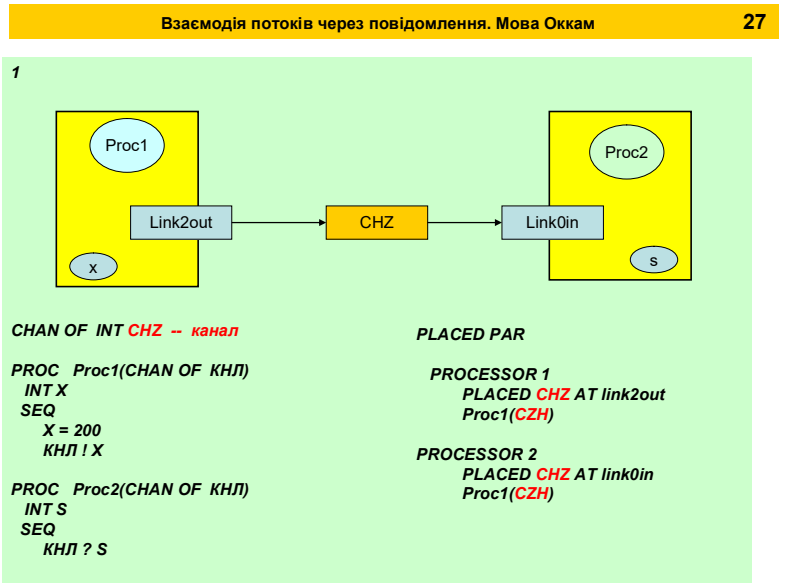



Рис. 13.8 Оккам. Взаємодія задач

13.3. Бібліотека PVM.

Бібліотека (інтерфейс) PVM (Parallel Virtual Machine) визначає стандарт для програмування в комп'ютерних системах з розподіленою пам'яттю [2, 7, 29]. PVM містить набір утиліт і бібліотечних функцій, що дозволяє проектувати паралельні додатки для мов, які не мають вбудованих засобів роботи з процесами.

Обмін повідомленнями. Процеси в PVM взаємодіють через посилання повідомлень. Дії посилання повідомлень:

- ініціалізація буфера, який буде використаний для повідомлення;
- упакування даних, які будуть передаватися, відповідно до формату їх подання;
- пересилання повідомлення в буфер.

Прийняття повідомлення пов'язано з такими діями:

- визначити алгоритм приймання (з блокуванням або без блокування);
- прийняти повідомлення;
- розпакувати повідомлення.

Під час формування повідомлення воно отримує ідентифікатор, який приймає ціле значення. Його призначення таке саме, як і ідентифікатор задачі (`Tid`).

Робота з буфером. Для роботи з буфером у PVM використовують низку функцій:

```
int bufid = pvm_initsend(int encoding)
```

де параметр `encoding` приймає такі значення:

- *PvmDataDefault* – автоматичне кодування різних даних, які приймаються і передаються;
- *PvmDataRaw* – без кодування даних для запису в буфер;
- *PvmDataInPlacet* – дані будуть упаковані і передані в буфер задачі, що приймає, минаючи буфер задачі, що передає.

Створення нового порожнього буфера для задачі, що передає:

```
int bufid = pvm_mkbuf(int encoding);
```

Знищення буфера з ідентифікатором `bufid`:

```
int info = pvm_freebuf(int bufid);
```

Повертає ідентифікатор активного буфера передавання:

```
int bufid = pvm_getsbuf(void);
```

Повертає ідентифікатор активного буфера приймання:

```
int bufid = pvm_getrbuf(void);
```


Переволить активність з буфера передавання `oldbuf`, на інший буфер передавання `bufid`:

```
int oldbuf = pvm_setsbuf(int bufid);
```

Перемикаає активність з буфера приймання `oldbuf` на інший буфер приймання `bufid`:

```
int oldbuf = pvm_setrbuf(int bufid);
```

Більшість схем взаємодії задач потребують в кожній задачі одного буфера приймання і одного буфера передавання. У цьому випадку достатньо використання тільки функції `pvm_initsend();`

Пакування повідомлень. Пакування даних у буфер передавання здійснюється за допомогою 12 функцій. Три параметри цих функцій:

- покажчик відповідного типу на перший елемент масиву, який пакується;
- розмір масиву;
- кількість елементів вказаного типу, що складає один елемент масиву.

Приклад функції для пакування даних типу `float`:

```
int info = pvm_pkcplx(float *cp,int n, int s);
```

Функції передавання повідомлень. Для *передавання* повідомлення необхідно надати йому ідентифікатор (параметр `msgid`), а також ідентифікатор задачі (параметр `tid`), яка приймає повідомлення.

Надає повідомленню ідентифікатор `msgid` і посилає це повідомлення задачі з ідентифікатором `tid`:

```
int info = pvm_send(int tid, int msgid);
```

Надає повідомленню ідентифікатор msgid і посилає це повідомлення задачам, які прораховані в елементах ntask масиву tids:

```
int info = pvm_mcast(int *tids, int ntask,
                                    int msgid);
```

Пакує nt елементів масиву vd типу tp, надає повідомленню ідентифікатор msgid і посилає повідомлення задачі з ідентифікатором tid:

```
int info = pvm_send(int tid, int msgid,
                        void *vd, int nt, int tp);
```

Функції приймання повідомлень. Для функцій приймання можливі варіації значень параметрів: якщо tid = -1 , то приймається будь-яке повідомлення з ідентифікатором msgid, яке надіслане задачі; якщо msgid = -1, то приймається будь-яке повідомлення, яке надіслане задачі від задачі з ідентифікатором tid; якщо tid = msgid = -1, то приймається будь-яке повідомлення від будь-якої задачі, що надіслане цій задачі.

Виконується блоковане приймання повідомлення, тобто очікування повідомлення з ідентифікатором msgid від задачі tid і створює приймальний буфер, посилає повідомлення задачі з ідентифікатором bufid і розміщує в ньому повідомлення, яке надходить:

```
int info = pvm_recv(int tid, int msgid);
```

Виконується неблоковане приймання повідомлення, тобто якщо повідомлення немає, воно закінчується; якщо повідомлення з параметрами tid і msgid надійшло, то воно розміщується в приймальному буфері, ідентифікатор якого повертається через параметр bufid:

```
int bufid = pvm_nrecv(int tid, int msgid);
```


Виконується блоковане приймання повідомлення, якщо час очікування не перевищує значення tmout (задається у вигляді двох цілих полів – секунд і мілісекунд):

```
int bufid = pvm_trecv(int tid, int msgid,
                            struct timeval *tmout);
```

Виконується перевірка надходження повідомлення. Якщо воно надійшло, то повертається ідентифікатор приймального буфера bufid:

```
int bufid = pvm_probe(int tid, int msgid);
```

Розпакування повідомлень. Функції розпакування даних з приймального буфера аналогічні розглянутим вище функціям пакування для повідомлення, що надсилається.

Приклад функції для розпакування даних типу float:

```
int info = pvm_upkcplx(float *cp,int n, int s);
```

Перелік функцій PVM для пакування даних у повідомленні, яке надсилається, і розпакування повідомлень, що надійшли, наведено в додатку Г.

Групові функції. Групові функції дозволяють об'єднувати задачі в групи з ідентифікацією групи. Це дає змогу організовувати всередині групи синхронізацію задач-членів групи, обмін повідомленнями, координацію з іншими задачами.

❖ **Приклад 13.4.**

```
/*-------------------------------------------------------------------------
-- PVM. Взаємодія задач                                                 --
-------------------------------------------------------------------------*/
   /* Основна програма Майстер  */
   #include "pvm3.h"
   main(){
      int Помилка, Tid, ТегПовідомлення;

      char Буфер(50);
```

```
   printf("Майстер стартував  Tid = ", pvm_mytid());

   Помилка = pvm_spawn("Новий потік",
                          (char**),0,0,"",1,&Tid);

   if (Помилка==1) {

      ТегПовідомлення = 1;

      /* Отримати повідомлення
      pvm_recv(Tid, ТегПовідомлення);

      /* Розпакувати повідомлення
      pvm_upkstr(Буфер);

      printf("Отримано від задачі Робітник з Tid ",
                                        Tid, Буфер;
   }
   else
      printf("Помилка при запуску задачі  ");

   pvm_exit();

}// Майстер

/* Задача Робітник  */--------------------------------
#include "pvm3.h"

main(){

   int PTid, ТегПовідомлення;

   char Буфер(50);

   PTid = pvm_parent();

   strcpy(Буфер, "Повідомлення від задачі Робітник");

   ТегПовідомлення = 1;

   /* Створення буфера
   pvm_initsend(PvmDataDefault);

   /* Пакування даних
   pvm_ptstr(Буфер);
```



```
      /* Відправлення  повідомлення
      pvm_send(PTid, ТегПовідомлення);

      pvm_exit();

   }// Робітник
```

У прикладі застосовано дві задачі – `Майстер` і `Робітник`. Задача `Майстер` запускає задачу `Робітник` і приймає від неї повідомлення (строку символів).

➢ **Запитання для модульного контролю**

1. Загальна схема концепції взаємодії процесів через посилання повідомлень ?
2. Назвіть види операцій `send/receive` ?
3. Як здійснюється синхронізація процесів при передаванні повідомлень ?
4. Що включає поняття каналу і протоколу в Оккам ?
5. Мова Оккам. Засоби взаємодії потоків ?
6. Структура трансп'ютера ?
7. Структури трансп'ютерних систем ?
8. В чому полягає відмінність між реалізацією механізму рандеву у мовах Оккам і Ada ?
9. Які засоби PVM потрібні для відправлення повідомлення ?
10. Які дії в PVM пов'язані з відправленням повідомлення ?

ЛЕКЦІЯ 14
Механізм рандеву в мові Ada

14.1.Механізм рандеву

Механізм рандеву визначає реалізацію взаємодії задач у мові Ada через посилання повідомлень [http://www.ada-auth.org/arm.html].

Оператори: входу `entry`, прийняття виклику входу `accept`, оператор відбору `select` дозволяють ефективно реалізувати різноманітні можливості моделі рандеву [2, 4, 5, 22].

В рандеву реалізовано концепція модель *клієнт-сервер*, в якої задач сервер містить вхід, за допомогою якого задач клієнт звертається до сервера через виклик цього входу з метою пересилання або отримання даних. Рандеву реалізує синхронізацію задач, передача даних відбувається лише тоді, коли обидві задачі готові це робити, тобто "заявилися на місце побачення ". Якщо одна з задач зробила це раніше, то вона блокується і чекає на появу іншої.

Напрям і обсяг передачі даних задається за допомогою оператора `entry`, який розміщується в специфікації серверної задачі. Будь яка задача клієнт може звертатися до серверної задачі через виклик потрібного входу. Дії сервера під час рандеву описуються в операторі `accept`, розміщеному в тілі серверної задачі

Приклад використання механізму рандеву для організації взаємодії задач `A` і `B`, де задача `A` передає в задачу `B` ціле число x. Структурну схему взаємодії задач надано на рис. 14.1. Для взаємодії задач у специфікації задачі `B` описаний вхід `Data`, специфікація якого дозволяє приймати дані (скаляр x) в задачі `B`. Пересилання цілого числа зі задачі `A` виконано як виклик входу `Data` з підстановкою фактичного параметра x.

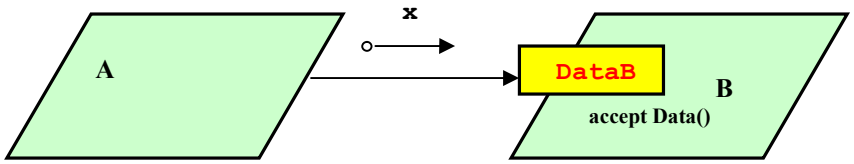


Рис. 14.1. Схема 1 взаємодії задач (одностороння)

❖ Приклад 14.1

```
------------------------------------------------------
--  Ada.Механізм рандеву                             --
------------------------------------------------------
procedure Lab141 is

   task A;
   task B is
       entry DataB(e: in integer);          -- вхід
   end B;

   task body A is

       x: integer := 25;        -- локальна  змінна
   begin
         .  .  .
       B.DataB(x);     -- виклик входу Data задачі B
         .  .  .
   end A;

    ---------------------------------------------
   task body B is

       c: integer;           -- локальна   змінна
   begin
         .  .  .
       -- оператор приймання виклику входу
       accept DataB(e: in integer) do
          c:= e;            -- тіло оператора приймання
       end DataB;

  end B;

  -- основна програма
Begin
```

```
   null;      -- порожний оператор

end Lab141;
```

Можлива інша організація взаємодії задач A і B, де вхід `Data` розміщений в задачі A, тобто серверної буде вже задача A. Структурну схему взаємодії задач надано на рис. 14.2. Перенесення входу `DataA` в задачу A потребує зміни в ньому модифікатора напрямку `in` на `out`. Виклик входу `DataA` з тепер відбувається в клієнтської задачі B.

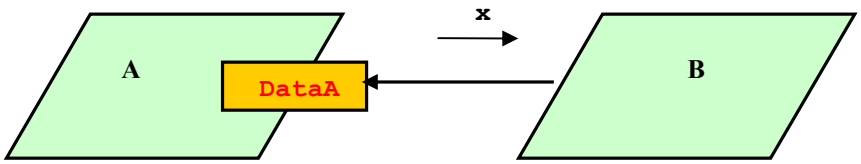


Рис. 14.2. Схема 2 взаємодії задач (одностороння)

❖ Приклад 14.2

```
-------------------------------------------------------
--  Ada.Механізм рандеву                              --
-------------------------------------------------------
procedure Lab142 is

   task B;

   task A is
      entry DataA(e: out integer);          -- вхід
   end A;
    ------------------------------------
   task body B is
      c: integer;        -- локальна  змінна
   begin
         .  .  .
      A.DataA(c);     -- виклик входу Data задачі A
         .  .  .
   end B;
  ---------------------------------------
   task body A is
      x: integer := 25;          -- локальна   змінна
```



```
   begin
         .  .  .
      -- оператор приймання виклику входу
      accept DataA(e: out integer) do

         e:= x;            -- тіло оператора приймання

      end DataA;

  end A;

            -- основна програма
Begin

   null;      -- порожний оператор

end Lab14.4;
```

Структурну схему взаємодії задач, під час якої задачі здійснюють обмін даними, зображено на рис. 14.3. При цьому застосовані два входи: `Data` для пересилання даних і `Result` для повернення результату. Для запуску задач їх вкладено в процедуру `Run_Tasks`, виклик якої приведе до старту задач `A` і `B`.

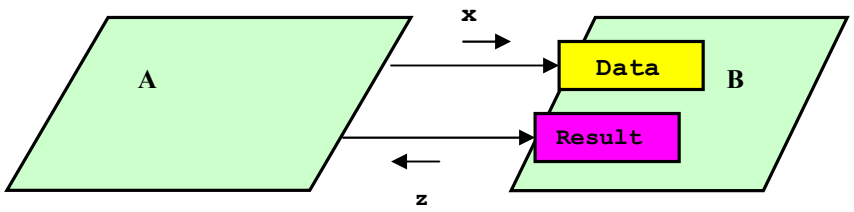


Рис. 14.3. Схема взаємодії задач (двостороння)

❖ Приклад 14.3.

```
-------------------------------------------------------------
-- Ada.Механізм рандеву. Обмін даниви. Два входи           --
-------------------------------------------------------------
procedure Lab143 is

   -- процедура із вкладеними задачами
```

```
procedure Run_Tasks is
    task A;
    task B is
       entry Data   (e: in  integer);
       entry Result (t: out integer);
    end B;

    task body A is
       x: integer := 25;
       z: integer;
    begin
         .  .  .
       B.Data(x);
         .  .  .
       B.Result(z);
         .  .  .
    end A;
    -----------------
    task body B is
      c, f : integer;
    begin
         .  .  .
       -- прийом даних
       accept Data(e: in integer) do
          c:= e;
       end Data;

       f:= c*c;           -- обчислення

       -- повернення результату
       accept Result(t: out integer) do
          t:= f;
       end Data;
         .  .  .
    end B;
begin               -- основна процедура

    Run_Tasks;      -- запуск задач

end Lab143;
```

Структурну схему взаємодії задач, під час якої задачі здійснюють обмін даними, зображено на рис. 14.4. При цьому застосовані один вхід: DataAB для пересилання і отримання даних. Для запуску задач їх вкладено в процедуру Run_Tasks, виклик якої приведе до старту задач A і B.

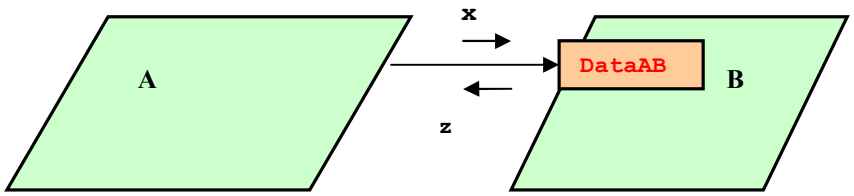


Рис. 14.4. Схема взаємодії задач (обмін)

❖ Приклад 14.4.

```
-------------------------------------------------------------
-- Ada.Механізм рандеву Обмін даними. Один вхід            --
-------------------------------------------------------------
procedure Lab144 is

   -- процедура із вкладеними задачами
   procedure Run_Tasks is
       task A;
       task B is
          entry DataAB(e: in  integer;t: out integer);
       end B;

       task body A is
          x: integer := 25;
          z: integer;
       begin
            .  .  .
          B.DataAB(x, z);
            .  .  .

       end A;
       -----------------
       task body B is
         c, f : integer;
       begin
            .  .  .
         -- прийом і передача даних
         accept DataAB(e:in integer;t: out integer)  do
            c:= e;
```

```
            t:= f;
         end DataAB;

            .  .  .
      end B;
   begin                -- основна процедура

      Run_Tasks;        -- запуск задач

   end Lab144;
```

14.2 Оператор `select`

Структурну схему взаємодії трьох задач – `T1`, `T2` і `T3` зображено на рис. 14.5. Задача `T3` приймає дані від задач `T1` і `T2` за допомогою входів: `DataT1` і `DataT2`. Особливістю реалізації взаємодії задач є використання в тілі задачі `T3` оператора `select`. Це дозволяє задачі `T3` приймати виклик входів `DataT1` та `Datat2` в будь-якій послідовності незалежно від порядку їх надходження.

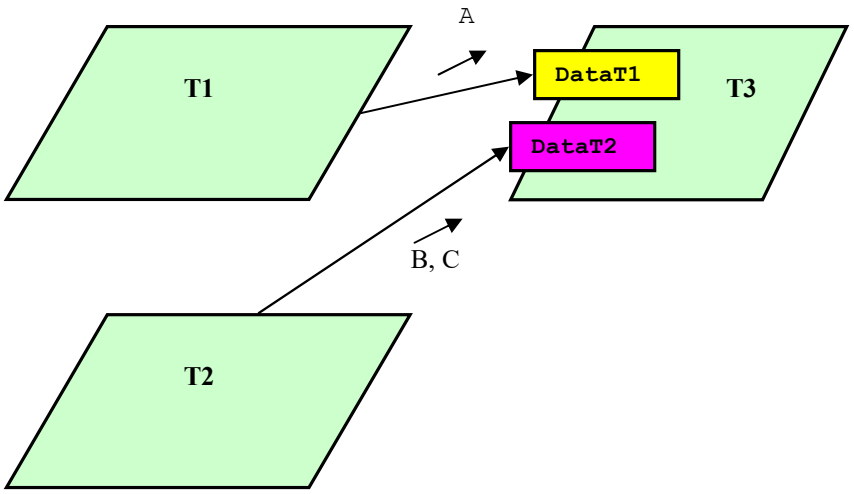


Рис. 14.5 Схема взаємодії задач. Оператор `select`

❖ Приклад 14.5.

```
-------------------------------------------------------------
--  Ada.Механізм рандеву. Оператор select                  --
-------------------------------------------------------------
procedure Lab145 is

   N: integer:= 10;
   type Вектор is array(1..N) of integer;

   procedure Run_Tasks is

      task T1;
      task T2;
      task T3 is
         entry DataT1(V:       in  Вектор);
         entry DataT2(V1, V2: in  Вектор);
      end B;
      --------------
      task body T1 is
         A: Вектор;
      begin
          .  .  .
         T3.DataT1(A);
          .  .  .
      end T1;
      --------------
      task body T2 is
         B,C: Вектор;
      begin
          .  .  .
         T3.DataT2(B,C);
          .  .  .
      end T2;
      ---------------
      task body T3 is
         VA, VB, VC: Вектор;
      begin
         .  .  .
         for i in 1..2 loop
            --  використання select
            select
               accept DataT1(V: in Вектор) do
                  VA:= V;
               end Data;
            or
               accept DataT2(V1, V2 in Вектор) do
                  VB:= V1;
```

```
                VC:= V2;
              end Data;
            end select;
          end loop;
              .  .  .
       end T3;

   begin

       null;

   end Run_Tasks;

begin                -- основна програма

   Run_Tasks;        -- явний запуск задач T1, T2, T3

end Lab145;
```

Існують декілька видів оператора `select`:
- селективний
- за часом
- умовний
- асинхронний.

Це дозволяє ефективно реалізувати за допомогою механізму рандеву різноманітні види поведінки задач при посиланні повідомлень.

➢ **Запитання для модульного контролю**

1. Призначення механізму рандеву?
2. Оператори для реалізації механізму рандеву ?
3. Призначення оператора `entry`?
4. Як в рандеву задається напрям передавання даних?
5. Призначення оператору accept?
6. Як реалізована в механізмі рандеву модель клієнт-сервер ?
7. Синхронізація потоків під час рандеву?
8. Призначення і види оператору `select`?
9. Де описуються дії задач під час рандеву?
10. Чи можливий обмін даними між задачами під час рандеву?

ЛЕКЦІЯ 15.
Бібліотека MPI

15.1. Реалізація концепції повідомлень в MPI

Бібліотека (інтерфейс) MPI (Message Passing Interface) визначає стандарт для програмування в комп'ютерних системах з розподіленою пам'яттю [7, 29]. MPI містить набір утиліт і бібліотечних функцій, що дозволяє проектувати паралельні додатки для мов, які не мають вбудованих засобів роботи з процесами.

Повідомлення. Процеси в MPI взаємодіють через посилання повідомлень. При цьому можливий обмін даними і синхронізація процесів. Розділяють процес-відправник і процес-одержувач.

У MPI повідомлення містить такі компоненти:
- блок даних (тип `void*`);
- тип даних (тип `MPI_Datatype`);
- кількість даних (тип `int` , кількість одиниць цього типу в блоці повідомлення.

Використовують таку інформацію про відправника і одержувача:
- комунікатор групи працюючих паралельних процесів (тип
- `MPI_Comm`);
- ранг відправника (номер процесу-одержувача в комунікаторі тип `int`);
- ранг одержувача (номер процесу-відправника в комунікаторі, тип `int`).

Крім того, для організації повідомлення використовують *тег* повідомлення – ціле число (тип `int`), що надається повідомленню відправником. Одержувач може вказати, що він буде приймати тільки ті повідомлення, які мають цей тег.

Обмін повідомлень має чотири форми:
1) попарний – повідомлення посилається одним процесом іншому одному процесу;
2) колективний – один процес передає повідомлення всім процесам його групи, що об'єднані за допомогою комунікатора;
3) асинхронний – процес-відправник не чекає, поки процес-одержувач отримує повідомлення, яке в цьому випадку копіюється в буфер;
4) синхронний – процес-відправник блокується, поки процес–одержувач не отримує повідомлення; процес-одержувач блокується, поки процес-відправник не буде готовий послати повідомлення.

Попарний обмін повідомленнями. Існує чотири види обміну для двох задач: синхронний, асинхронний, з блокуванням, без блокування. У MPI використовують чотири режими обміну (блокована і неблокована форми):
1) стандартне передавання; завершується одразу після відправлення повідомлення;
2) синхронне передавання; не завершується, поки не буде отримане повідомлення;
3) буферизований; завершується одразу же після копіювання повідомлення в системний буфер;
4) у міру готовності; передавання розпочинається тільки тоді, коли адресат ініціює приймання повідомлення.

Домовленість про ім'я функцій обміну двох задач.
Для передавання:

```
MPI_[I] [R,S,B] Send;
```

Префікс `[I]` означає неблокований режим; `[R]` – в міру готовності; `[S]` – синхронізований; `[B]` – буферизований. Стандартний обмін не потребує жодного префікса. Таки чином, існує вісім різновидів передавання повідомлень.

Для приймання:

```
MPI_[I] Recv;
```

Обмін із блокуванням процесів. Процес, який виконав посилання повідомлення блокується, поки інший процес не отримає це повідомлення. Процес, який приймає повідомлення, також буде блокований, поки він не отримує це повідомлення.

Надіслати повідомлення можна за допомогою функції **`MPI_Send():`**

```
int MPI_Send(void *buffer, int count,
    MPI_Datatype tp, int rg, int tag, MPI_Comm cm);
```

де:

`buffer` – адреса буфера з даними;
`count` – кількість елементів у буфері;
`tp` – тип даних для елементів у буфері;
`rg` – ранг одержувача в групі;
`tag` – тег повідомлення;
`cm` – комунікатор.

Отримати повідомлення можно за допомогою функції **`MPI_Recv():`**

```
int MPI_Recv(void *buffer, int count,
    MPI_Datatype tp, int rg, int tag, MPI_Comm cm,
    MPI_Status *st);
```

де:

`buffer` – адреса буфера, де слід розмістити отримане повідомлення;
`count` – максимальна кількість елементів у буфері;
`tp` – тип даних для елементів у буфері;
`sr` – ранг відправника в групі,

tag – тег повідомлення;
cm – комунікатор;
st – інформація про отримання повідомлення.

Обмін без блокування процесів. Операції передавання (приймання) повідомлень виконуються без блокування. Тобто передавальний процес відправляє повідомлення і не чекає на його отримання іншим процесом.

Для неблокованої взаємодії використовують відповідно функції **MPI_ISend()** і **MPI_IReceive()**.

15.2.Колективні операції

Для реалізації колективного обміну застосовують поняття *комунікатор* – ідентифікатор групи процесів, які можуть обмінюватися повідомленнями. Ідентифікатор комунікатора використовують як аргумент у всіх функціях, які здійснюють обмін повідомленнями.

Комунікатор дозволяє організовувати колективні обміни всередині групи, ізолювати обміни різних груп процесів, ураховувати особливості структури розподіленої комп'ютерної системи.

Функції для роботи з комунікатором:

- Кількість процесів в комунікаторі (групі):

```
int MPI_Comm_size(MPI_Comm cm, int *res);
```

де:
cm – коммунікатор;
res – покажчик на результат (кількість процесів).

- Номер (ранг) процесу в комунікаторі (групі):

```
int MPI_Comm_rank(MPI_Comm cm, int *rank);
```

де:
cm – комунікатор;
rank – покажчик результату (ранг).

Колективній обмін повідомленнями дозволяє оптимізувати організацію кількох попарних обмінів.

Для колективного розсилання даних з одного процесу в групі всім іншим процесам в групі використовують функцію **MPI_Bcast()**:

```
int MPI_Bcast(void *buffer, int count,
          MPI_Datatype tp, int root, MPI_Comm cm);
```

де:
buffer – адреса буфера з відправленими даними для процесу з номером root; адрес буфера з даними, де слід розміщувати отримане повідомлення для інших процесів групи cm;
count – кількість елементів у буфері для процесу з номером root; максимальна кількість елементів у буфері з даними для інших процесів групи cm;
tp – тип даних для елементів у буфері;
root – ранг відправника в групі cm;
cm – комунікатор.

Для колективного збору даних з кількох процесів групи в один процес групи використовують функцію **MPI_Reduce()**:

```
int MPI_Reduce(void *sbuffer, void *rbuffer,
               int count, MPI_Datatype tp,
               MPI_Op op, int root, MPI_Comm cm);
```

де:
sbuffer – адреса буфера;
rbuffer – покажчик буфера, де треба отримати результат; використовується тільки для процесу з номером root у групі cm;
count – кількість елементів у буфері;
tp – тип даних для елементів у буфері;
op – ідентифікатор операції (таб. 4.1);
root – ранг одержувача в групі cm;
cm – комунікатор.

Таблиця 15.1. Деякі функції бібліотеки MPI

Функція	Операція
MPI_Send()	Переслати повідомлення. Блокуватися, поки воно не буде отримано
MPI_Recv()	Отримати повідомлення. Блокуватися, поки воно не буде надіслано
MPI_ISend()	Переслати повідомлення без блокування
MPI_IRecv()	Отримати повідомлення без блокування
MPI_Probe()	З'ясувати, чи отримано повідомлення (блокуватися до отримання повідомлення)
MPI_IProbe()	З'ясувати, чи отримано повідомлення (без блокування)
MPI_Bcast()	Надіслати повідомлення всім процесам в групі
MPI_Reduce()	Отримати дані від усіх процесів у групі
MPI_Gather()	Зібрати дані від всіх процесів
MPI_Scatter()	Розсилати дані усім процесам у групі. Дані – одного розміру.
MPI_Scatterv()	Розсилати дані всім процесам у групі. Дані можуть бути різного розміру.
MPI_Alltoall()	Розсилати дані всім процесам в групі зі всіх процесів групи. Дані одного розміру.
MPI_Alltoallv()	Розсилати дані всім процесам у групі зі всіх процесів групи. Дані можуть бути різного розміру.

Операції, які можна виконувати в процесі-одержувачі над даними, що збираються функцією MPI_Reduce(), наприклад, формувати суму отриманих даних, наведено в табл. 15.2.

Таблиця 15.2. Операції в функції MPI_Reduce()

Код	Операція
MPI_Max()	Максимум
MPI_Min()	Мінімум
MPI_SUM()	Сума
MPI_PROD()	Додаток
MPI_LAND()	Логічне «І»
MPI_BAND()	Побітове «І»
MPI_LOR()	Логічне «Або»
MPI_BOR()	Побітове «Або»
MPI_LXOR()	Логічне «Виключне Або»
MPI_BXOR()	Побітове «Виключне Або»
MPI_MAXLOC()	Максимум та його позиція
MPI_MINLOC()	Мінімум та його позиція

Якщо треба, щоб результат виконання функції MPI_Reduce() був отриманий не тільки одним процесом в групі (root), а всіма процесами у групі, то для цього використовують функцію **MPI_Allreduce():**

```
int MPI_Allreduce(void *sbuffer, void *rbuffer,
    int count, MPI_Datatype tp, MPI_Op op,
    MPI_Comm cm);
```

де:

sbuffer – адреса буфера з даними;
rbuffer – покажчик буфера, де треба отримати результат;
count – кількість елементів у буфері;
tp – тип даних для елементів у буфері;
op – ідентифікатор операції;
cm – комунікатор.

Результат формується в буфері в усіх процесорах групи cm.

Синхронізацію задач в групі можна виконати за допомогою функції **MPI_Barrier():**

```
int MPI_Barrier(MPI_Comm cm);
```

де cm – комунікатор.

Ця функція блокує задачу, що її викликав, доти, доки всі задачі групи з ідентифікатором cm не викличуть цю функцію.

❖ **Приклад 15.1.**

Створення процесів, кожний з яких передає головному процесу (ранк 0) свій ранк , який виводиться на дисплей.

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>

/* Визначення довжини буфера повідомлень  */
#define довжина_буфера 256

int main(int args, char *argv[]){

   int кількість_процесів;
   int мій_ранг;
   int ранг_відправника;
   int ранг_отримувача;
   int тег_повідомлення;

   MPI_Status статус;
   char буфер[довжина_буфера];

   /* Початок  роботи MPI */
   MPI_Init(&args, &argv);

   /* Отримати номер поточного процесу в групі */
   MPI_Comm_rank(MPI_COMM_WORLD, &мій_ранг);

   /* Отримати загальну кількість процесів,
      що старували */
   MPI_Comm_size(MPI_COMM_WORLD,
                 &кількість_процесів);

   /* Процеси 1 - .  .  .
      Відправити повідомлення процесу 0 для виведення
      на дисплей */
   if (мій_ранг != 0){

      /* Створення повідомлення */
      sprintf(Повідомлення, "Start of process %d!",
              мій_ранг);
```



```
      /* Відправити повідомлення процесу 0  */
      ранг_отримувача = 0;

      MPI_Send(Повідомлення, strlen(Повідомлення)+1,
               MPI_CHAR, ранг_отримувача,
               тег_повідомлення, MPI_COMM_WORLD);
   }
   else{
      /* процес 0  Отримати повідомлення з інших
         процесів  і вивести на дисплей */

      for(ранг_відправника = 1; ранг_відправника <
            кількість_процесів; ранг_відправника++){

         MPI_Recv(Повідомлення, довжина_буфера,
                  MPI_CHAR, ранг_відправника,
                  тег_повідомлення, MPI_COMM_WORLD,
                  &статус );

         prinf("%s\n", Повідомлення);

      }/* for */

   }

   /* Завершення роботи  MPI */

   MPI_Finalize();
   return 0;

}/* main */
```

15.3.Віртуальні топології

Механізм віртуальних топології MPI дозволяє формувати додаткову систему адресації для задач. Віртуальна топологія є логічною і не пов'язана з топологією фізичного середовища передачі даних.

В MPI передбачено два типи топології:

- декартова топологія (прямокутна решітка довільного розміру);
- топологія графа.

Декартова топологія.

Функція MPI_Cart_create створює новий комуніктор **Decart** з декартовою топологією:

```
int MPI_Cart_create(MPI_Comm com_Base, int nd,
          int *dm, int *pr, int rdr,
                                  MPI_Comm *Decart)
```

де параметры:

- com_Base – вхідний комунікатор;
- nd - розмір декартової решітки;
- dm – масив з nd елементів, визначає кількість задач в кожному виміру;
- pr – логічний масив з dm елементів, який визначає періодичність решітки вздовж кожного виміру;
- rdr - при значенні параметра "true" дозволяється зміна порядку нумерації задач.

Інформація о структуре декартової топології міститься в параметрах nd, dm і pr. Дозволяє описати багатомірні решітки.

Функція MPI_Cart_create є колективною операцією, викликається всіма задачами з комунікатору Decart з декартовой топологией, що створюється.

```
#include "mpi.h";
#include <stdio.h>;
int main(int argc,char *argv[])  {

  MPI_Comm grid_comm;

  int dims[2];
  int periodic[2];
  int reorder = 1;
  int q = 5;
  int ndims = 2;
  int maxdims = 2;
  int coordinates[2];
  int my_grid_rank;
  int coords[2];

MPI_Comm row_comm;
dims[0] = dims[1] = q;
periodic[0] = periodic[1] = 1;
```



```
coords[0] = 0; coords[1] = 1;
MPI_Init(&argc, &argv);
MPI_Cart_create(MPI_COMM_WORLD, ndims, dims,
                  periodic, reorder, &grid_comm);
MPI_Comm_rank(grid_comm, &my_grid_rank);
MPI_Cart_coords(grid_comm, my_grid_rank,
                           maxdims, coordinates);
printf("Process rank %i has coordinates %i %i\n",
    my_grid_rank, coordinates[0], coordinates[1]);

MPI_Finalize();
return 0;
```

Топологія графа.

Для топології задається кількість вершин і матриця інцідентності
Приклад топології “зірка” (рис.15.4).

Кількість вершин графа – 5,
матриця звязків - (таблиця 15.1)
порядки вершин приймають значення (4,1,1,1,1),

Таблиця 15.1 Матриця зв’язків

Вершина	Зв’язки
0	1,2,3,4
1	-
2	-
3	-
4	-

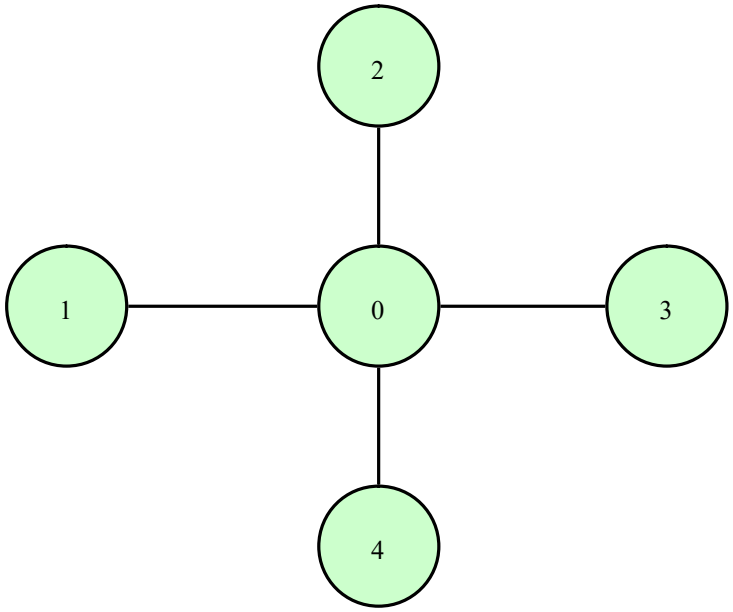


Рис 15.4 Топологія графа «зірка»

Для створення топології з графом “зірка” слід визначити наступні масиви :

```
int index[ ] = { 4,1,1,1,1 };
int edges[ ] = { 1,2,3,4,0,0,0,0 };
```

і побудувати комунікатор:

```
MPI_Comm Star;
MPI_Graph_create(MPI_COMM_WORLD, 5,
                          index, edges, 1, &Star);
```

➢ **Запитання для модульного контролю**

1. Яка загальна схема взаємодії потоків через посилання повідомлень в MPI?

2. MPI.Назвіть види операцій `send/receive` ?
3. MPI. Роль комунікатора при взаємодії задач ?
4. Призначення функції `MPI_Comm_rank()` ?
5. Призначення функції `MPI_Comm_size()` ?
6. Можливості колективних операцій?
7. Призначення операції `MPI_Bcast()`?
8. Призначення операції `MPI_Reduce()`?
9. Призначення операції `MPI_Barrier()`?
10. Види віртуальної топології ?

ЛЕКЦІЯ 16
Програмування для комп'ютерних систем зі спільною пам'яттю. Низькорівневі засоби

У цьому розділі розглянуто приклади розв'язання задач у комп'ютерних системах із спільною та локальною пам'яттю, а також в розподілених системах. Для низки векторно-матричних задач виконано повний цикл створювання програм, який включає розроблення паралельного математичного алгоритму, алгоритмів кожного паралельного процесу або алгоритмів сервера та клієнта, структури взаємодії процесів, програми.

Наведено практичне використання семафорів, мютексів, захищених модулів, механізму рандеву, а також сокетів і видалених підпрограм [2, 8,10, 12, 13].

16.1. Програмування для комп'ютерних систем із спільною пам'яттю

16.1 Ada.Семафори

Реалізація операції додавання векторів у двох процесорній системі із спільною пам'яттю з використанням семафорів мови Ada**.**

Вхідні дані:

- комп'ютерна система зі спільною пам'яттю включає два процесори і два пристрої введення-виведення (рис. 16.1),
- математичне завдання: операція додавання векторів,

$$A = B + C * x,$$

де A, B, C – вектори розміру N, x – скаляр,

- уведення вектора B і скаляра x виконується в процесорі 1, уведення вектора C – у процесорі 2, виведення результату вектор A – у процесорі 1.

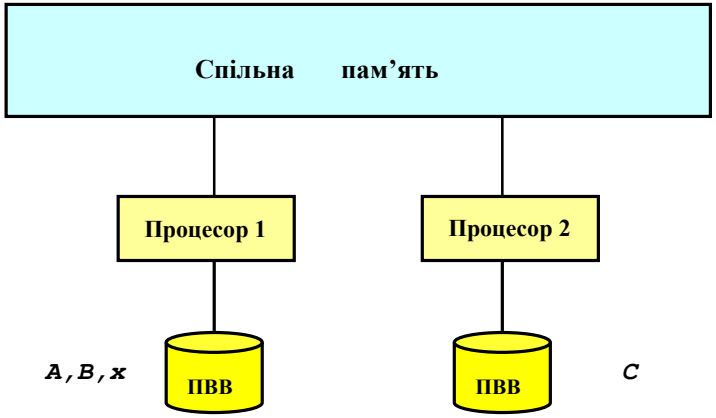


Рис. 16.1. Структурна схема двох процесорної системи

Етап 1. Побудова паралельного алгоритму. Операція додавання векторів являє собою приклад ідеально паралельної операції. Паралельний алгоритм можна представити у вигляді

$$A_н = B_н + C_н * x \qquad (16.1)$$

де $A_н$ – H елементів вектора A.

Співвідношення (16.1) визначає дії в кожному процесорі системи під час виконанні паралельних обчислень.

Співвідношення (16.1) потрібно проаналізувати на наявність спільних ресурсів (спільних змінних), які можуть одночасно використовуватися процесами як для читання, так і для змінювання. Задачі використовують різні частини векторів B і C, тому вони не є спільними ресурсами. Спільним ресурсом у співвідношенні (16.1) є скаляр x, відносно якого взагалі необхідно розв'язати завдання взаємного виключення. Але спільний ресурс x є константою, яка не

змінюється потоками. Її одночасне використання потоками не впливає на кінцевий результат. З метою оптимізації часу обчислень створимо в кожному потоці копію змінної *x* (локальні змінні *x1* і *x2*) з можливістю наступного розміщення копій в кеш - пам'яті процесору. Само копіювання має виконуватися в критичній ділянці кожного потоку і розглядатися як завдання взаємного виключення.

Етап 2. Розроблення алгоритмів роботи кожного потоку. Цей етап включає розроблення докладного алгоритму роботи кожного потоку. Такий алгоритм має включати *всі* дії потокуу, які, крім безпосередніх обчислень за формулою (16.1), будуть включати введення даних і виведення результату, а також дії, що пов'язані з організацією взаємодії потоків (розв'язування завдання взаємного виключення та синхронізації). Завдання взаємного виключення буде пов'язана з копіюванням змінної *x*, задачі синхронізації – із синхронізацією по:

- *введенню даних*, коли обидва потоки чекають на введення всіх даних і тільки після цього починають обчислення,
- *виведенню результату*, коли перший потік повинен дочекатися на завершення обчислень у другому потоції і тільки після цього виводити кінцевий результат.

В алгоритмі кожного потоку синхронізація (у *точках синхронізації*) буде реалізована діями **Чекати** і **Сигнал**. Точка синхронізації, яка пов'язана з *очікуванням* на подію, позначається як ***Wj,k,*** де ***j*** – номер потоку, у якій відбудеться подія; ***k*** – порядковий номер взаємодії цього потоку та потоку ***Tj*** (якщо взаємодій декілька). Аналогічно позначається точка синхронізації ***Sj,k***, що пов'язана з *сигналом* до потоку ***Tj*** про подію, яка відбулася.

Потік *T1*

		Точки синхронізації
1.	Уведення *B* і *x*	
2.	**Сигнал** задачі *T2* про введення *B* та *x*	-- *S2,1*
3.	**Чекати** на уведення даних в задачі *T2*	-- *W2,1*
4.	Копіювати *x1: = x*	-- Критична ділянка(КД)

5.	Обчислення *Ан = Вн + Сн * x1*	
6.	**Чекати** на завершення обчислень в *T2*	-- *W2,2*
7.	Виведення результату *A*	

Потік *T2*

1.	Уведення *C*	
2.	**Сигнал** задачі *T1* про уведення *C*	-- *S1,1*
3.	**Чекати** на уведення даних в задачі *T1*	-- *W1,1*
4.	Копіювати *x2: = x*	-- КД
5.	Обчислення *Ан = Вн + Сн * x2*	
6.	**Сигнал** *T1* про завершення обчислень	-- *S1,2*

Етап 3. Розроблення структурної схеми взаємодії потоків (задач). Структурна схема взаємодії задач базується на алгоритмах задач і дозволяє наглядно контролювати зв'язок належних точок синхронізації (***W*** і ***S***), а також ув'язати їх з реальними семафорами, які використовуються потім в програмі. Графічне зображення взаємодії задач дозволяє виявити тупикові ситуації в програмі у випадку, коли точка синхронізації ***W*** не буде пов'язана з належною точкою синхронізації ***S.*** Крім того, на структурній схемі уводяться також семафори, що відповідають за роботу зі спільними ресурсами і потім використовуються в програмі.

На структурній схемі взаємодії задач (рис. 16.2) уведено таки семафори:

- ***Skd*** – для керування доступом до спільного ресурсу (*x*);
- ***Sem1*** – для синхронізації з завершення уведення в *T1;*
- ***Sem2*** – для синхронізації з завершення уведення в *T2;*
- ***Sem3*** – для синхронізації з завершення обчислень в *T2* і виведення результатів;

Етап 4. Розроблення програми. Розроблення програми виконують на підставі алгоритмів задач і структурної схеми взаємодії задач:

```
------------------------------------------------------------
--|Ada.Використання семафорів                               --
--|Операція  A = B + C * x                                  --
```

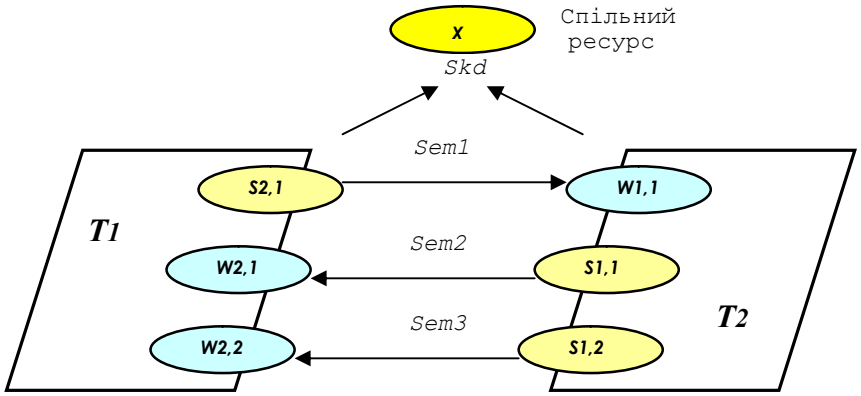


Рис. 16.2. Ada.Структурна схема взаємодії потоків (задач)

```
procedure Lab161 is

   N: integer := 100;  -- розмір векторів
   P: integer := 2;    -- кількість процесорів
   H: integer := N/P;  -- розмір підвектора

   x: integer;  -- спільний ресурс (глобальна змінна)

   type Вектор is array (1.. N) of integer;

   A,B,C : Вектор;          -- глобальні змінні

      -- семафори
      Sem1, Sem2, Sem3, Skd:  Suspension_Object;

      procedure  Запуск_Задач  is

         task T1;

         task body T1 is
            x1: integer;       -- локальна змінна
         begin
            put_line("Process T1 started ");
            -- уведення B та x
            for i in 1.. N loop
```



```
               B(i):= 1;
            end loop;

            x:= 2;

            -- сигнал задачі T2 про уведення B та x
            Set_True(Sem1);

            --  чекати на уведення даних в задачі T2
            Suspend_Until_True(Sem2);

            -- копіювати  x в x1
            Suspend_Until_True(Skd);
                x1: = x;          -- критична ділянка КД
            Set_True(Skd);

            -- обчислення  AH
            for i in 1 .. H loop
               A(i):= B(i) + C(i)*x1;
            end loop;

            -- Чекати на завершення обчислень в T2
            Suspend_Until_True(Sem3);

            -- виведення результату
            put_line(" A = ");
            for i in 1 .. N loop
               put(A(i),3);
            end loop;

         put_line("Process T1 finished");
      end T1;
      ---------------------------------------
      task T2;
      task body T2 is
         x2: integer;       -- локальна  змінна
      begin
         put_line(" Process B started ");
         -- уведення C
         for i in 1.. N loop
            C(i):= 2;
         end loop;

         -- сигнал задачі T1 про уведення C
         Set_True(Sem2);

         --  чекати на  уведення даних в задачі T1
```

```
            Suspend_Until_True(Sem1);

            --копіювати  x  в x2
            Suspend_Until_True(Skd);
               x2: = x;          -- критична ділянка КД
            Set_True(Skd);

            -- обчислення Ан
            for i in H+1 .. N loop
               A(i):= B(i) + C(i)*x2
            end loop;

            -- сигнал  T1 про завершення обчислень
            Set_True(Sem3);

            put_line(" Process T2 finished");
         end T2;
      begin
         null;
      end Запуск_Задач;

--  тіло основної програми
begin

   put_line("  Main procedure started ");

   -- встановлення початкового значення семафора Skd
   Set_True(Skd);

   Запуск_Задач;

end Lab161;
```

16.2. WinAPI. Семафори, мютекси, події

Реалізація операції додавання векторів у двох процесорній системі із спільною пам'яттю з використанням бібліотеки WinAPI. Для приклада, що розглянутий в 16.1., для синхронізації процесів використовуємо семафори, а для взаємного виключення – мютекси.

Етапи 1– 3 не будуть відмінними від етапів, що розглянуті в прикладі 16.1. На структурній схемі (Рис.16.3) семафор `Skd`, буде змінений на мютекс `Mut`, семафор `Sem3` – на подію `Ev1`.

Для використання функцій `WinAPI` в системі програмування ObjectAda потрібне ручне підключення бібліотеки `Win32lib` при

створенні Ада-проекту і підключення до програми пакетів `Win32, Win32.Base, Win32.Win32NT`.

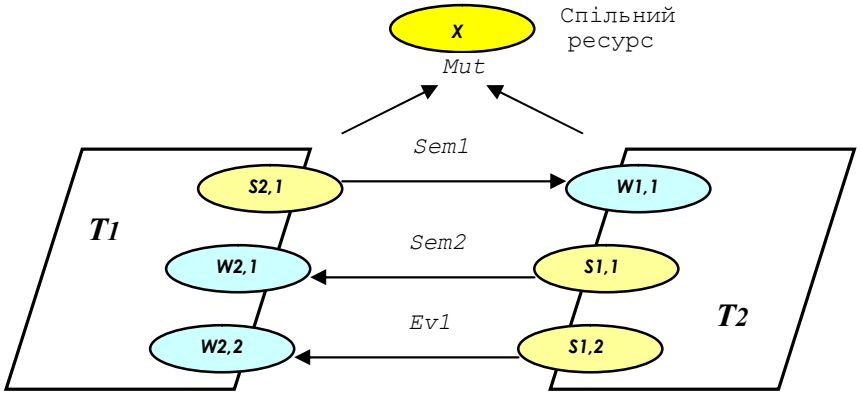


Рис. 16.3. WinAPI.Структурна схема взаємодії потоків

Етап 4. Розроблення програми

```
-------------------------------------------------------------
--|Ada.Використання  бібліотеки  WinAPI                    --
--|Застосування   семафорів,мютексів, подій                --
--|Операція A = B + C * x                                  --
-------------------------------------------------------------
procedure Lab162 is

   N: integer := 100;  -- розмір векторів
   P: integer := 2;    -- кількість процесорів
   H: integer := N/P;  -- розмір підвектора

   X: integer;   -- спільний ресурс(глобальна змінна)

   type Вектор is array (1.. N) of integer;
   A,B,C : Вектор;     -- глобальні змінні

  -- HANDLE  змінні
  Sem1, Sem2, Ev1, Mut: Handle;

   -- допоміжні змінні
   Temp: DWORD;
   p1, p2,p3,p4 : BOOL;
```

```
procedure Запуск_Задач is

    task T1;
    task T2;

    task body T1 is
       x1: integer;      -- локальна змінна
    begin
       put_line("Process T1 started");

       -- уведення B i x
       for i in 1.. N loop
          B(i):= 1;
       end loop;

       x:= 2;

       -- сигнал задачі T2 про введення B i x
       p1 := ReleaseSemaphore(Sem1,1,NULL);

       --  чекати на введення C у задачі T2
      Temp:= WaitForSingleObject(Sem2,Infinite);

       -- копіювання  x в x1
       Temp:= WaitForSingleObject(Mut,Infinite);
          x1: = x;             -- критична ділянка
       p1:= ReleaseMutex(Mut);

       -- обчислення
       for i in 1 .. H loop
          A(i):= B(i) + C(i)*x1;
       end loop;

       -- чекати на завершення обчислень у T2
       Temp := WaitForSingleObject(Ev1, Infinite);

       -- виведення результату
       put_line(" A = ");
       for i in 1 .. N loop
          put(A(i),3);
       end loop;

       put_line("Process T1 finished");
    end T1;
    ----------------------------------------
    task body T2 is
       x2: integer;        -- локальна змінна
```



```
    begin
       put_line(" Process T2 started");

       -- уведення C
       for i in 1.. N loop
          C(i):= 2;
       end loop;

       -- сигнал задачі T2 про введення C
       P2:= ReleaseSemaphore(Sem2,1,NULL);

       --  чекати на введення B та x  у задачі T1
       Term := WaitForSingleObject(Sem1,Infinite);

       -- Копіювати  x  у x2
       Term := WaitForSingleObject(Mut, Infinite);
          x2: = x;             -- критична ділянка КД
       p3:= ReleaseMutex(Mut);

       -- обчислення
       for i in H+1 .. N loop
          A(i):= B(i) + C(i)*x2
       end loop;

       -- сигнал задачі T1 про завершення обчислень
       P2:= SetEvent(Ev1);

       put_line(" Process T2 finished");

    end T2;

 begin

    null;

 end Запуск_Задач;

begin
   put_line("  Main procedure started ");

   --  створення семафорів, мютексу, події
   Sem1:= CreateSemaphore(NULL, 0,1, NULL);
   Sem2:= CreateSemaphore(NULL, 0,1, NULL);
   Ev1 := CreateEvent(NULL, 0,0, NULL);
   Mut := CreateMutex(NULL, 0, NULL);

   -- виклик процедури для запуску задач
```

```
   Запуск_Задач;

end Lab162;
```

- **Запитання для модульного контролю**
  1. Види низкорівневих засобів для рішення завдання взаємного виключення ?
  2. Види низкорівневих засобів для рішення завдання синхронізації?
  3. Події в WinAPI. Керування вилами скидання стану події?
  4. Події в C#. Керування вилами скидання стану події?
  5. Використання багатозначних семафорів ?
  6. Java. Застосування семафорів в завданні взаємного виключення і синхронізації процесів ?
  7. Ada. Особливості реалізації семафорів
  8. Яке призначення типу `HANDLE` в бібліотеці WinAPI ?
  9. Як реалізовано механізм критичних секцій в C# ?
  10. В чому полягає різниця між семафорами та мютексами в WinAPI ?

- **Завдання для самостійної роботи**
  1. Написати програму мовою Java, в якій вирішується завдання взаємного виключення для трьох потоків, що мають доступ до спільної змінної `Ресурс.` Використати:
     а) семафори,
     б) критичні секції (синхронізовані блоки).
  2. Написати програму на мові Ada, в якій вирішується завдання взаємного виключення для чотирьох потоків, які мають доступ до спільної змінної `Ресурс.`
     Використати:
     а) семафори з пакета `Synchronous_Task_Control`,
     б) прагми,
     в) засоби бібліотеки WinAPI (семафори, мютекси, події, критичні секції).
  3. Вирішити за допомогою семафорів WinAPI завдання синхронізації одного потоку з кількома іншими (потік чекає на

події в трьох інших потоках).
  4. Вирішити за допомогою засобів WinAPI завдання синхронізації кількох потоків з одним (чотири потоки чекають на подію в одному потоці).
  5. Завдання 3 реалізувати за допомогою мови C#.
  6. Завдання 4 реалізувати за допомогою захищеного модуля мови C# .
  7. Виконати моделювання механізму богатозначних семафорів за допомогою захищеного модуля мови Ada.
  8. Завдання 3 реалізувати за допомогою засобів мови Java.
  9. Завдання 4 реалізувати за допомогою засобів мови Java.
  10. Провести дослідження часу виконання в ПКС одної і тої ж паралельної програми, де задача взаємного виключення вирішується за допомогою різних засобів синхронізації. Визначити також час виконання цієї програми у випадку, коли задача взаємного виключення взагалі не вирішується.

ЛЕКЦІЯ 17
Програмування для комп'ютерних систем зі спільною пам'яттю. Монітори

У лекції розглянуто приклади розв'язання задач у комп'ютерних системах із спільною пам'яттю з використанням моніторів. Для низки векторна-матричних задач виконано повний цикл створювання програм, який включає розроблення паралельного математичного алгоритму, алгоритмів кожного паралельного процесу, структури монітору і взаємодії потоків через монітор, програми.

Розглянуто практичне використання захищених модулів мови Ada, а також моніторів мови Java [4, 5].

17.1 Ada. Захищені модулі

Реалізація операції додавання векторів $A = B + C * x$, у двох процесорній системі із спільною пам'яттю (рис. 16.1) з використанням захищеного модуля мови Ада.

Етапи 1–3 не будуть відрізнятися від етапів, розглянутих в прикладі в Розділі 16.

Етап 3. Розроблення структурної схеми взаємодії задач. Етап пов'язаний с розробкою структури захищеного модуля, за допомогою якого реалізується взаємодія задач. Захищений модуль `Керування` (рис. 17.1) включає три захищені елементи: `x`, `F1` і `F2`, а також набір захищених операцій:

- вхід `Чекати_Введення` для синхронізації з введення в задачах `T1` і `T2`;

- вхід `Чекати_Обчислень`, для синхронізації по завершенню обчислень у задачі `T2`;

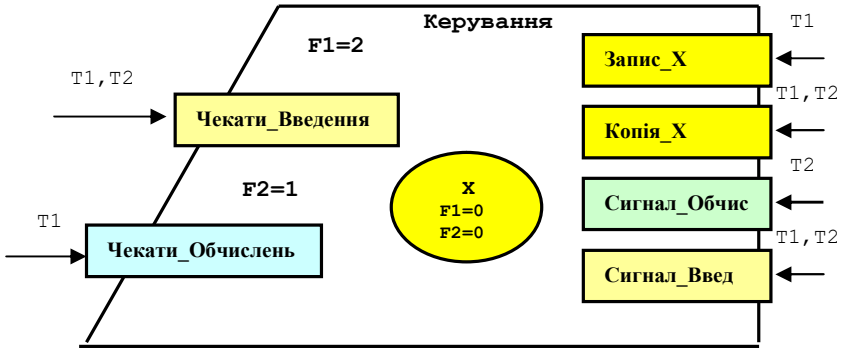


Рис. 17.1. Ada.Захищений модуль в операції `A = B + C * x`

- функцію `Копія_X` для копіювання спільного ресурсу `x;`
- процедуру `Запис_X` для запису значення `x` У захищений модуль;
- процедуру `Сигнал_Обчис` для сигналу про завершення обчислень У задачі `T2`;
- процедуру `Сигнал_Введ` для сигналу про завершення уведення даних у задачах `T1` і `T2`.

Захищені змінні `F1` і `F2` використовуються в бар'єрах входів `Чекати_Введ` і `Чекати_Обчис`.

Етап 4. Розроблення програми. Розроблення програми виконується на підставі алгоритмів задач і структурної схеми взаємодії задач.

```
------------------------------------------------------------
--|Ada.Захищений модуль                                     --
--|Операція  A = B + C * x                                  --
------------------------------------------------------------
procedure Lab171 is

   N: integer := 400;  -- розмір векторів
   P: integer := 2;    -- кількість процесорів
```

```
H: integer := N/P;  -- розмір підвектора

type Вектор is array (1.. N) of integer;

A,B,C : Вектор;          -- глобальні змінні

--  захищений модуль
-- специфікація
protected  Керування is

    entry  Чекати_Введення;
    entry  Чекати_Обчислень;
    function   Копія_X return integer;
    procedure  Запис_X(e: in integer);
    procedure  Сигнал_Введ;
    procedure  Сигнал_Обчис;

private
    x : integer;  -- спільний ресурс
    F1: integer:= 0;
    F2: integer:= 0;

end Керування;

-- тіло
protected body Керування is
   entry  Чекати_Введення when F1 = 2 is
   begin
       null;
   end Чекати_Введення;

   entry  Чекати_Обчислень when F2 = 1 is
    begin
        null;
    end Чекати_Обчислень;

    function  Копія_X return integer is
    begin
         return x;
    end Копія_X;

    procedure  Запис_X(e: in integer) is
    begin
         x:= e;
      end Запис_X;

    procedure  Сигнал_Введ is
```



```
    begin
        F1:= F1 + 1;
    end Сигнал_Введ;

  procedure  Сигнал_Обчис is
    begin
       F2:= 1;
    end  Сигнал_Обчис;

end Керування;
-----------------------------------------
-- задачі
task T1;

task body T1 is
    x1, z : integer;   -- локальні змінні
begin
   put_line("Process T1 started");

   -- введення B та x
   for i in 1.. N loop
       B(i):= 1;
   end loop;
   z:= 2;

   -- запис x  в захищений модуль
   Керування.Запис_X(z);

   -- Сигнал задачі T2 про введення B та x
   Керування.Сигнал_Увед;

    --  чекати на введення C в задачі T2
    Керування.Чекати_Введення;

    -- Копіювати  x  в x1
    X1: = Керування.Копія_X;

    -- обчислення
    for i in 1 .. H loop
       A(i):= B(i) + C(i)*x1;
    end loop;
 -- Чекати на завершення обчислень у T2
Керування.Чекати_Обчислень;

-- виведення результату
put_line(" A = ");
for i in 1 .. N loop
```

```
        put(A(i),3);
     end loop;

     put_line("Process T1 finished");

  end T1;
  ------------------------------------------
  task T2;
  task body T2 is
     x2: integer; -- локальна змінна
  begin
     put_line(" Process B started");

     -- введення C
     for i in 1.. N loop
        C(i):= 2;
     end loop;

     -- сигнал задачі T1 про введення C
     Керування.Сигнал_Введ;

     -- чекати уведення B і x  у задачі T1
     Керування.Чекати_Введення;

     -- копіювати  x  в x2
     X2: = Керування.Копія_X;

     -- обчислення
     for i in H+1 .. N loop
        A(i):= B(i) + C(i)*x2
     end loop;

     -- сигнал  T1 про завершення обчислень
     Керування.Сигнал_Обчис;

     put_line(" Process T2 finished");

  end T2;

--  тіло основної програми
begin

    put_line("    Main procedure started ");

end Lab171;
```


17.2. Java. Монітори

Реалізація операції скалярного множення векторів $a = (B*C)$ в двох процесорній системі зі спільною пам'яттю (рис. 17.2) з використанням механізму моніторів у мові Java**.**

Етап 1. Побудова паралельного алгоритму. Паралельній алгоритм операції $a = (B*C)$ можна подати у вигляді

$$a_i = (B_н * C_н) \qquad (17.2)$$
$$a = a + a_i$$

Спільним ресурсом є скаляр ***a,*** відносно якого необхідно розв'язувати завдання взаємного виключення під час формуванні кінцевого результату. Спільний ресурс ***a*** в (17.2) використається зліва і справа від знаку рівно, тобто і читається і змінюється процесом. Це – «справжній» спільний ресурс. Його не можна копіювати і в КД кожного потоку треба виконувати всю операцію $a = a + a_i$.

Етап 2. Розроблення алгоритмів потоів

Задача *P1*

		Точки синхронізації
1.	Уведення *B* і *C*	
2.	**Сигнал** задачі P2 про введення *B* і *C*	-- *S2,1*
3.	Обчислення a1 = (Bн * Cн)	
4.	Обчислення a = a + a1	-- КД1
5.	**Чекати** на завершення обчислень в *P2*	-- *W2,1*
6.	Виведення результату *a*	

Задача *P2*

1.	**Чекати** на уведення в *P1*	-- *W1,1*
2.	Обчислення a2 = (Bн * Cн)	
3.	Обчислення a = a + a2	-- КД1
4.	**Сигнал** *P1* про завершення обчислень	-- *S1,1*

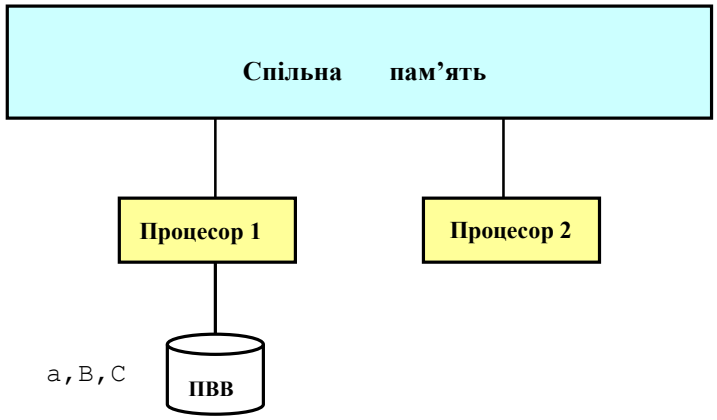


Рис. 17.2. Структурна схема комп'ютерної системи

Етап 3. Розроблення структурної схеми взаємодії потоків. Етап пов'язаний з розробленням структури класу (монітора), за допомогою якого реалізується взаємодія задач. Клас `Data` (рис. 17.3) включає три поля: `a`, `Fl1` і `Fl2`, а також набір методів для організації взаємодії потоків:

- `wait_In()` - для очікування завершення уведення даних;
- `signal_In()` - для сигналу про завершення уведення даних;
- `signal_Out()` - для сигналу про завершення обчислень;
- `wait_Out()` - для очікування на завершення обчислень;
- `put_a()` – для роботи зі спільним ресурсом.

Етап 4. Розроблення програми. Програма реалізована у вигляді основного класу `Lab172` і пакета `Vector`, який включає три класи:

- `Data` – клас для синхронізації потоків;
- `aThread` – клас для потоку P1;
- `bThread` – клас для потоку P2.

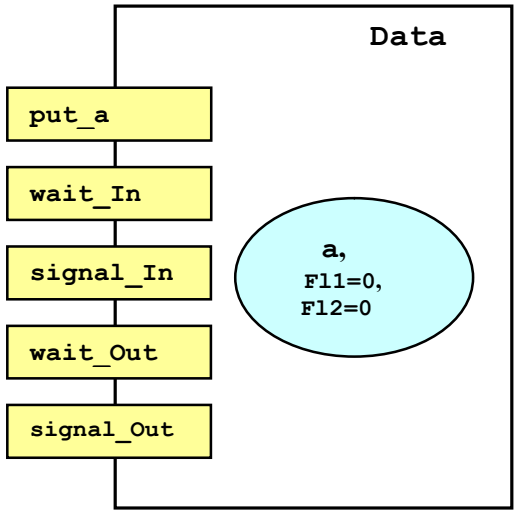


Рис. 17.3. Структура класу-монітору `Data`

```
/*-------------------------------------------------
-- Java. Монітори. Операція a = (B*C)                --
-------------------------------------------------*/
package Vector;

// монітор
public class Data {


   // змінні
   int private a = 0;

   private int Fl1 = 0;
   private int Fl2 = 0;

   public synchronized void
      put_a(int x){ a = a + x;}

   public synchronized void wait_In(){
      try{
         if (Fl1 = = 0)
              wait();
```

```
      }catch(Exception e) {}
   }
   public synchronized
                  void signal_In(){
      notify();
      Fl1 = 1;
   }
   public synchronized void wait_Out(){
      try{
          if(Fl2 = = 0)
                        wait();
      }catch(Exception e) {}
   }
   public synchronized
                  void signal_Out(){
      notify();
      Fl2 = 1;
   }
}

// потік 1 ---------------------------------
public class aThread extends Thread{

   int  a1 = 0;
   Data Z;

   public aThread(Data q){
      Z = q;
   }

   public void run(){

      // уведення  B,C
      for(int i = 0; i< Z.N; i++){
         Z.B[i] = 1;
         Z.C[i] = 1;
      }
      // сигнал P2 про уведення даних
      Z.signal_In();                          //S2,1

      // обчислення
      for (int i = 0; i < Z.H; i++)
            a1 = a1 + Z.B[i]*Z.C[i];

      // формування результату
      Z.put_a(a1);                // Критична ділянка
```



```
      //Очікування на завершення обчислень в P2
      Z.wait_Out();                           // W2,1

      // Виведення результату
      Z.geta();
   }
}

// потік 2 ---------------------------------
public class bThread extends Thread{

   int   a2 = 0;
   Data  Z;

   public bThread(Data q) {
      Z = q;
   }
   public void run(){

      // очікування на уведення в P1
      Z.wait_In();                            // W1,1

      //  обчислення
      for (int i= Z.H; i< Z.N; i++)
         a2 = a2 + Z.B[i]*Z.C[i];

      // формування результату
      Z.put_a(a2);                // Критична ділянка

      //  сигнал про завершення обчислень
      Z.signal_Out();                         // S1,1
   }
}

// Основний клас ---------------------------------
import Vector.*;
public class Lab172 {

   public static void main (String [] args){

      int N = 1000;
      int H = N/2;

      int B[] = new int [N];
      int C[] = new int [N];

      Data D = new Data();
```

```
        aThread  P1 = new  aThread(D);
        bThread  P2 = new  bThread(D);
        P1.start();
        P2.start();
    }
}//Lab172
```

- **Запитання для модульного контролю**
  1. Структура захищеного модуля для синхронізації?
  2. Структура захищеного модуля для завдання взаємного виключення?
  3. Засоби синхронізація за допомогою захищеного модуля ?
  4. Особливість використання захищених функцій ?
  5. Засоби Java – монітору для синхронізації?
  6. Засоби Java – монітору для завдання взаємного виключення?
  7. Java. Призначення методів `notify/notifAll`
  8. Java. Призначення методу `wait ?`
  9. Синхронізовані методи при побудові Java-монітору?
  10. В чому полягає відмінність в реалізації концепції монітора в мовах Java і Ada

ЛЕКЦІЯ 18
Програмування для комп'ютерних систем з локальною пам'яттю

У лекції розглянуто приклади розв'язання задач у комп'ютерних системах з розподіленою пам'яттю з використанням моделі повідомлень. Для низки векторна-матричних задач виконано повний цикл створювання програми, який включає розроблення паралельного математичного алгоритму, алгоритмів кожного паралельного потоку, взаємодії потоків через формування повідомлень.

Розглянуто практичне використання мови Ada, а також бібліотек MPI [2, 4, 29].

18.1. Ada. Рандеву

Реалізація операції додавання векторів $A = B + C * x$ для трьох процесорній системі з локальною пам'яттю (три процесори і два пристрої введення-виведення (ПВВ)) з використанням механізму рандеву мови Ada.

Вхідні дані (вектори B та C, скаляр x) уводяться в ПВВ1, що зв'язаний з процесором 1, результат – вектор A виводиться в ПВВ2.

Етап 1. Побудова паралельного алгоритму. Паралельний алгоритм реалізації операції:

$$Ан = Вн + Сн * x$$

Етап 2. Розроблення алгоритмів роботи кожного потоку (задачі). Цей етап пов'язаний з розробленням алгоритмів роботи кожного потоку. Крім безпосередніх обчислень, уведення вхідних

даних і виведення результату, алгоритми мають включати дії, що пов'язані із взаємодією задач через посилання даних. Така взаємодія включає:

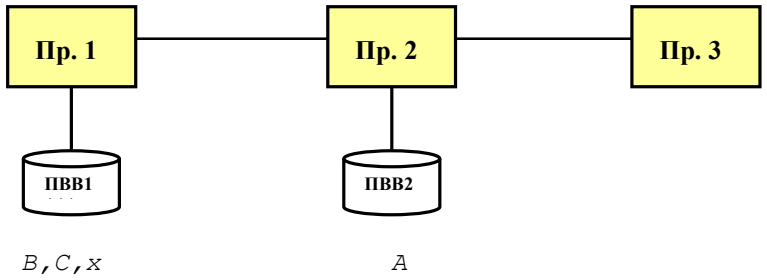


Рис. 18.1. Структура ВПКС з локальною пам'яттю

- розсилання із задачі *T1* вхідних даних в задачі *T2* і *T3*; оскільки задача *T1* не пов'язана безпосередньо із задачею *T3*, то пересилання даних між ними буде виконуватися через задачу *T;*,
- збирання результату в задачі *T2*.

Розробляючи алгоритм задач слід, звернути увагу на обсяги даних, що пересилаються між задачами. Під час розсилання вхідних даних із задачі *T1* в *T2* потрібно переслати дві частини вектора *B* і *C* розміром 2**H* (позначається як B_{2H} і C_{2H}), одна з яких призначена безпосередньо для Т2, а друга ($B_н$, $C_н$) – для пересилання далі в задачу *T3*.

Пересилання повних векторів приводить до значного збільшення часу виконання програми, оскільки операції пересилання даних між процесами в КС з локальною пам'яттю є досить довгими операціями порівняно з операціями обчислень і необхідно їх мінімізувати як за кількістю пересилок, так і за обсягом інформації, що передається.

Задача *T1*

1. Уведення *B, C* та *x*
2. **Передати** задачі *T2* дані: $B_{2н}$, $C_{2н}$, *x*
3. Обчислення $A_н = B_н + C_н * x$
4. **Передати** в задачу *T2* частину результату $A_н$

Задача *T2*

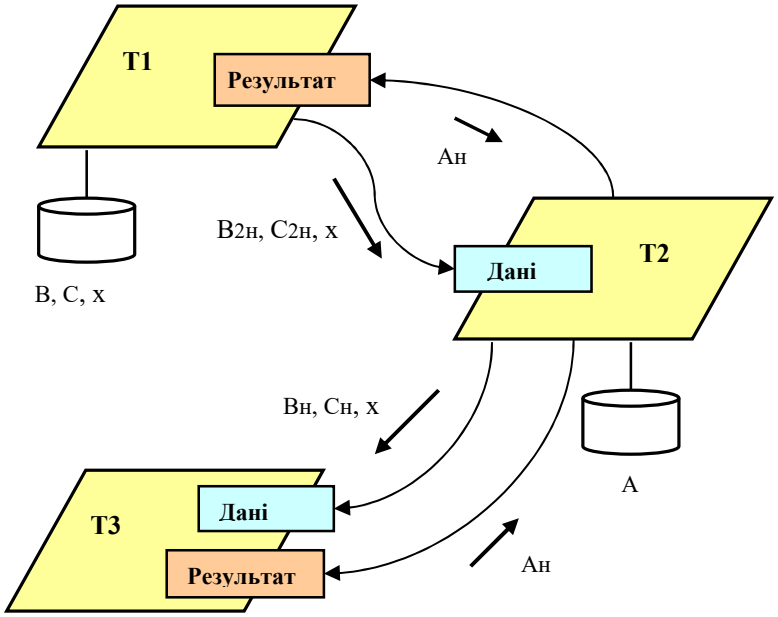
1. **Прийняти** від задачі *T1* дані: $B_{2н}$, $C_{2н}$, *x*
2. **Передати** в задачу Т3 дані $B_н$, $C_н$, *x*
3. Обчислення $A_н = B_н + C_н * x$
4. **Прийняти** від *T1* результат – вектор $A_н$
5. **Прийняти** від *T3* результат – вектор $A_н$
6. Виведення результату – вектор *A*

Задача *T3*

1. **Прийняти від** задачі *T2* дані: $B_н$, $C_н$, *x*
2. Обчислення $A_н = B_н + C_н * x$
3. **Передати** в задачу *T2* результат – $A_н$

Етап 3. Розроблення структурної схеми взаємодії задач. Структурна схема взаємодії задач ґрунтується на алгоритмах задач і дозволяє наочно показати передавання даних між задачами (рис. 18.2).

На структурній схемі визначаються задачі, входи задач, їх розміщення в належних задачах і протоколи взаємодії для них, які задають *напрям* і *обсяг* даних, що передаються між задачами. Для подання взаємодії задач використовують *дві* напрямлені лінії. Одна лінія визначає виклик входу належної задачі, друга, коротка – напрям передавання даних, яка буде виконуватися за цим викликом. Біля короткої стрілки вказується інформація, що передається, наприклад, два вектори і матриця. Ця інформація потім використовується для опису входів задач у програмі.



18.2. Рандеву.Структурна схема взаємодії задач

Графічне зображення взаємодії задач дозволяє також виявити можливі *тупикові ситуації* в програмі у випадку, якщо вхід не має виклику або між задачами має місце циклічний виклик входів. Щоб позбавитись таких ситуацій, слід розміщувати входи в одній задачі, а не в різних задачах. При цьому слід також контролювати порядок виклику входів.

Етап 4. Розроблення програми. Розроблення програми виконується на підставі алгоритмів задач і структурної схеми взаємодії задач. Формуючі типи, що використовувані в програмі, слід звернути увагу на оголошення достатньої кількості типів, що дозволяє

описати протоколи входів із зазначенням розмірів даних, оптимальних для передавання. Джерелом служить структурна схема взаємодії задач, за якою визначено обсяги та напрям даних, що пересилаються для кожної взаємодії задач.

Для розроблення програми, що розглядається. оптимальним є набір векторних типів для передавання:

- цілого вектора розміром *N*;
- підвектора розміром *2H;*
- підвектора розміром *H.*

Оскільки в програмі потрібно забезпечити сумісність цих трьох типів, їх треба оголосити як підтипи загального (батьківського) векторного типу `Вектор_Загальний`.

Усі змінні в програмі мають бути *локальними* та описаними безпосередньо в кожній задачі. Входи задач, зазначені як `Дані()`, використаються для розсилання вхідних даних із задачі *T1* в *T2* і потім з *T2* в *T3*. Входи задач, що зазначені як `Результат()`, використані для збирання результату в задачі *T2*.

Для роботи з векторами також слід використовувати механізм відрізків, які дозволяють передавати (отримувати) різні частини векторів під час роботи з фактичними параметрами для виклику входу.

```
------------------------------------------------------------
--|Ada.Механізм рандеву                                     --
--|Операція A = B + C * x                                   --
------------------------------------------------------------
procedure Lab181 is
   N: integer := 6;     -- розмір векторів
   P: integer := 3;     -- кількість процесорів
   H: integer := N/P;   -- розмір підвектора

   -- формування типів
   type Вектор_Загальний  is array(integer range <>)
                                                 of integer;
      subtype Вектор   is Вектор_Загальний (1..N);
      subtype Вектор2H is Вектор_Загальний (1..2*H);
      subtype ВекторH  is Вектор_Загальний (1..H);

   -- опис специфікацій задач
   task T1 is
      entry Результат(VA: out  ВекторH);
```

```
end T1;
task T2 is
   entry Дані(VB,VC : in Вектор2Н;
                  y : in integer);
end T2;
task T3 is
   entry Дані(VB,VC  : in  ВекторН;
                  y  : in integer);
   entry Результат(VA : out ВекторН);
end T3;
----------------------------------------
-- опис тіл задач
task body T1 is
   x1: integer;
   A1: ВекторН;
   B1, C1 : Вектор;
begin
   put_line("Process T1 started");

   -- уведення B,C і x
   for i in 1.. N loop
      B1(i):= 1;
      C1(i):= 2;
   end loop;
   x1:= 2;

   -- передати B2H, C2H  і x в  задачу T2
  T2.Дані(B1(H+1..N), C1(H+1..N), x1);

   -- обчислення
  for i in 1 .. H loop
      A1(i):= B1(i) + C1(i)*x1;
  end loop;

   -- передати в T2 Ан
  accept Результат(VA: out  ВекторН) do
     VA:= A1;
  end Результат;

   put_line("Process T1 finished");
end T1;

-------------------------------------------------
task body T2 is

   x2: integer;
   A2: Вектор;   -- для виведення повного результату
```



```
   B2, C2 : Вектор2Н;
begin
   put_line("     Process T2 started");

    -- прийняти B2H, C2н і x  від T1
    accept Дані(VB,VC : in Вектор2Н;
                   y : in integer) do
       B2 := VB;
       C2 := VC;
       X2 := y;
    end  Дані;

    -- передати  данні в T3
    T3.Данні(B2(H+1..2*H), C2(H+1..2*H), x2);

    -- обчислення
    for i in 1 .. H loop
       A2(H+i):= B2(i) + C2(i)*x2
    end loop;

    -- прийняти результат від T1
    T1.Результат(A2(1..H));

    -- прийняти результат від T3
    T3.Результат(A2(2*H+1..N));

    -- виведення результата
    put_line(" A = ");
    for i in 1 .. N loop
       put(A2(i),3);
    end loop;
    put_line("Process T2 finished");
end T2;
 ------------------------------------------------------
task body T3 is
   x3: integer;
    A3, B3, C3 : ВекторН;
Begin

    put_line("     Process T3 started");

    -- прийняти від T2 BH, Сн і x
    accept Дані(VB,VC : in ВекторН;
                   y : in integer) do
       B3:= VB;
       C3 := VC;
       X3 := y;
```

```
      end  Дані;

       -- Обчислення    Ан
      for i in 1 .. H loop
          A3(i):= B3(i) + C3(i)*x3;
      end loop;

       -- передати Ан в Т2
      accept Результат(VA: out  ВекторН) do
          VA:= A3;
      end Результат;

       put_line("Process T3 finished");

   end T3;

--  тіло основної програми
Begin

   put_line("  Main procedure started ");

end Lab55;
```

18.2. Бібліотека PVM

Реалізація операції множення матриць $MA=MB * MC$ в п'яти процесорній системі з локальною пам'яттю (рис.18.3) з використанням бібліотеки PVM та мови С [20].

Етап 1. Побудова паралельного алгоритму. Паралельний алгоритм для операції множення матриць:

$$MAн = MBн * MC$$

Етап 2. Розроблення алгоритмів роботи кожного процесу. Цей етап пов'язаний з розробленням алгоритмів роботи процесу `Майстер` (процесор 0) та `Робочий` (процесори 1–4).

Майстер

1. Уведення *MB, MC*
2. Активізація процесів `Робочий`

3. **Розсилання** всім процесам `Робочий` матриці *MC*
4. **Передати** *MBн* кожному процесу `Робочий`
5. Обчислення *MAн = MBн *MCн*
6. **Прийняти** *MAн* від кожного процесу `Робочий`
7. Виведення результату – матриці *MA*

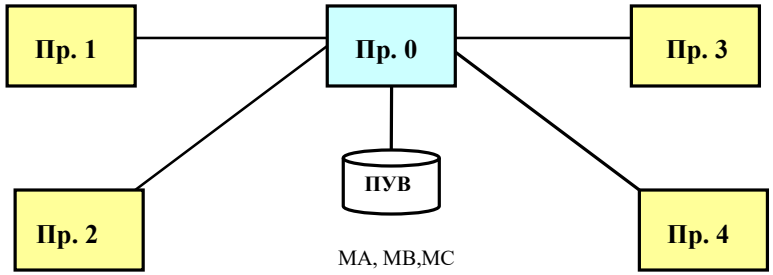


Рис. 18.3. Структура ВПКС з локальною пам'яттю

Робочий

1. **Прийняти** матрицю *MC* від процесу `Майстер`
2. **Прийняти** *MBн* від процесу `Мастер`
3. Обчислення *MAн = MBн *MCн*
4. **Передати** `MAн` до процесу `Майстер`

Етап 3. Розроблення структурної схеми взаємодії задач

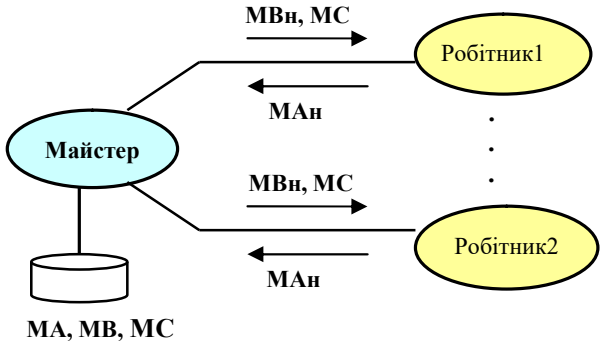


Рис. 18.4. Взаємодія задач Майстер–Робітник

Етап 4. Розроблення програми

```
/*-------------------------------------------------------
 PVM. Lab18. Операція MA=MB*MC
-------------------------------------------------------*/
#include "pvm3.h"
#include <stdio.h>
#include <string.h>

#define N 5
#define P 5
#define H 1

#define SPAWN_TYPE   PvmaTsks
#define SPAWN_WHERE  "Work"

int MA[N][N];
int MB[N][N] = (1, .., 1);
int MC[N][N] = (2, .., 2);

int MTemp[N][N];
int VTemp[N];
int MAH[N];

main(int argc, char **argv){

     int Child[P];  /* масив з tid  дочірніх процесів */
     int myTid;
     int pid;   /*   ідентифікатор процесу */
```



```
     int n;     /*   розмір масиву */

     char ** agvc = NULL;
     myTid = pvm_tid();    /* отримати свій  tid  */

     if (mytid<0){
       pvm_perror("Start pvm error ");
       exit(-1);
     }

     /* Процес - Майстер  */
     if pid == PvmNoParent){    /* майстер   */
        int i;
        int rc;
        num = NUM;

     /* запуск дочірніх процесів */
     rc = pvm_spwm(argv[0],argvc, SPAWN_TYPE,
                          SPAWN_WHERE, P-1, Child+1);

     /* переслати всім дочірнім процесам розмір матриць */
     pvm_initsend(PvmDataDefault);
     pvm_pkint(&N,1,1);
     pvm_mcast(Child + 1, P-1,1);

     /* переслати всім дочірнім процесам  матрицю MC  */
     pvm_initsend(PvmDataDefault);
     pvm_pkint(MC, P*P,1);
     pvm_mcast(Child + 1, P-1,1);

     /* переслати всім дочірнім процесам частину матриці
      MBн  (один стовпчик) */
     for(i=1; i<P; i++){
       pvm_initsend(PvmDataDefault);
       pvm_pkint(&MB[i][0], N,1);
       pvm_send(Child[i], 1);
     }
     /*  обчислення MA  */
     МноженняМатриць(&MA[0][0], MB[0][0], MC, N);

     /* прийняти  результат MAн  */
     for(i=1; i<P; i++){
        pvm_recv(Child[i],1);
        pvm_upkint(&MA[i][0], N,1);
     }

    /* виведення  MA  */
```

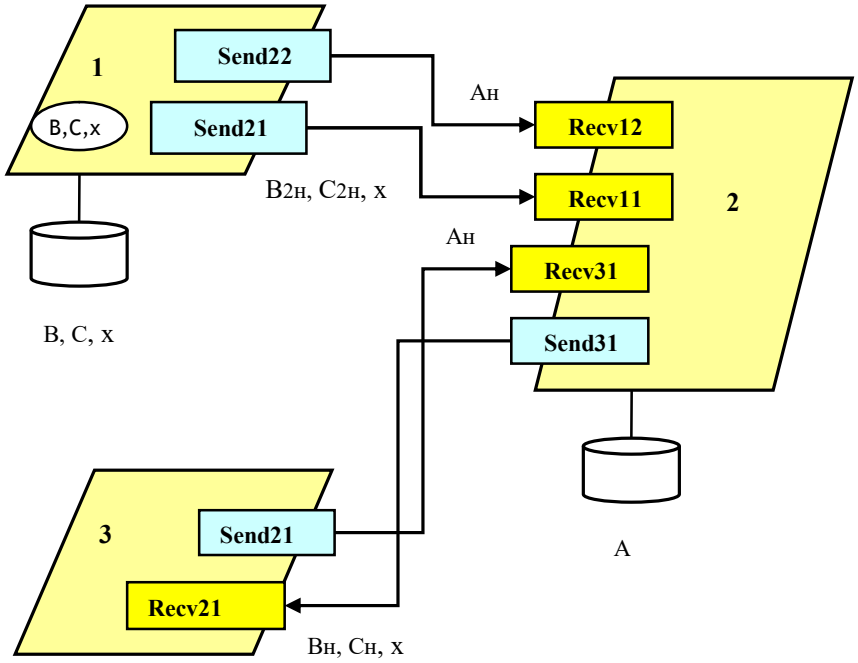
```
    ДрукМасиву("МА  = ", МА);

  }else
  {

    /*  Процес - Робочий */
    pvm_recv(pid,1);
    pvm_upkint(&N, 1,1);       /*  отримати розмір масиву */

     pvm_recv(pid,1);
     pvm_upkint(&MTemp,N*N,1);  /*  отримати МС    */

     pvm_recv(pid,1);
     pvm_upkint(&VTemp,N*N,1);  /*  отримати МВh   */

    /*  обчислення МА  */
    МноженняМатриць(MTemp, VTemp, МАН, N);

    /*  відправлення результату в процес Майстер */
    pvm_initsend(PvmDataDefault);
    pvm_pkint(МАН, N,1);
    pvm_send(pid,1);
  }

pvm_exit();  /*  завершення   і вихід з pvm */
exit(0);
}

/*  допоміжні процедури */
МноженняМатриць(int * x, int * y, int * d,  int s){
      .  .  .
}

ДрукМасиву(char * l, int  * M){
      .  .  .
}
```

18. 3 Бібліотека MPI

Розглядається завдання з розділу 18.1. Етапи 1 і 2 не змінюються, тому увагу слід приділити організації взаємодії потоків (створенню схеми взаємодії) та розробці програми.

Етап 3. Розроблення структурної схеми взаємодії задач

Особливістю схеми взаємодії потоків (задач) в MPI є використання функцій `Send` і `Receive`. На відміну від використання механізму рандеву на схемі не потрібно вказувати напрям передавання даних через те, що він задається відповідною функцією: для `Send` - це передача даних з задачі, для `Receive` – отримання даних в задачі.



18.5. MPI. Структурна схема взаємодії задач

Розсилання вхідних даних від задачі 1.

Після введення в задачі 1 даних (x, B. C) для виконання обчислень в задачі 2 і 3 слід переслати відповідні частини векторів Вн і Сн, а також скаляр x. Через відсутність зв'язку задачі 1 і 3 передача даних здійснюється через задачу 2.

*Передача даних з задачі 1 до задачі 2. В*икористовується одна функція Send21, в буфер якої записуються скаляр x, а також по дві частині (2Н) векторів В і С.

*Передача даних з задачі 2 до задачі 3. В*икористовується одна функція Send31, в буфер якої записуються скаляр x, а також по одної частині (Н) векторів В і С.

Збирання результату в задачі 2

Задача 2 отримує першу частину результату Ан від задачі 1 за допомогою функції Recv12, а від задачі 3 – третю частину – через функції Recv31. Особливість збирання результат на боці задачі 3 є коректне розміщення частин Ан, отриманих від задач 1 і 3, в повному векторі результат А за номерами задач.

Етап 4. Розроблення програми. Розроблення програми виконується на підставі алгоритмів задач 1-3 і структурної схеми взаємодії задач.

При формуванні параметрів функцій `Send` і `Recv` слід звернути увагу на побудову буферів для відправки і отримання повідомлень в задачах.

Роздача даних x задачі 1.

Бажано відправляти дані разом однім `Send`, в буфер якого потрібні упакувати частини векторів B, C і скаляр x.

Задача 0. Буфер Buf0 повинен мати розмір 4*H + 1 для розміщення и передачі даних B2н, C2н і x до задачі 1.

Задача 1: Буфер Buf1 повинен мати розмір 2*H +1 для розміщення и передачі даних Вн, Сн і x до задачі 1.

Збірка результату (Ан) в задачі 2.

Результат (вектор А) збирається в задачі 1 частинами Ан від задач 1 і 3 і виводиться.

Номера частин векторів при роздачі і збірки повіни співпадають.


```
const int P = 3;
const int N = 6;
const int H = N / P;
void fillVector(int vector[], int number);
void printVector(int vector[], int size);

int main(int* argc, char** argv) {
   int numtasks, rank;
   MPI_Init(argc, &argv);
   MPI_Status status;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

   if (rank == 1)     {
      int B[N];
      int C[N];
      int x;
      int A1[H];

      // введення даних
       fillVector(B, N);
       fillVector(C, N);
       x = 1;

       //   формування буферу
       int Buf1[4*H + 1];
       Buf1[0] = x;
       Buf1[1..2*H]     = B[H+1..N];
       Buf1[2*H+1.. N] = C[H+1..N];

       // відправка даних в задачу 2 через
       MPI_Send(Buf1, 4*H+1, MPI_INT, 2, 1,
                                    MPI_COMM_WORLD);

        //обчислення першої частини Ан
       for (int i = 1; i < H+1; i++)
          A1[i] =  B[i] + C[i]*x;

       //  відправка результату АН в задачу 1 через
       MPI_Send(A1, H, MPI_INT, 2, 1, MPI_COMM_WORLD);
    } //  задача 1
   //=============================================================
    if (rank == 2)        {
       int B2[2*H];
       int C2[2*H];
       int A2[H];
       int x2;
```

```
    int A[N]; --  результат
    int Buf21[4*H +1];// буфер для прийому від задачі 1
    int BufAH[H]; // буфер для збирання  результату

     // отримання даних  від  задачі 0
     MPI_Recv(Buf21, 4*H + 1, MPI_INT, 1, 1,
                              MPI_COMM_WORLD, &status);
    x2 = Buf2[0];
    B2[1..H] = Buf21[1..H]
    C2[1..H] = Buf21[2*H +1 .. 3*H];

    // формування  буферадля передачі даних до задачі 2
    int Buf22[2*H +1];
    Buf22[0]        = x2;
    Buf22[1..H]     = Buf21[H+1 .. 2*H];
    Buf22[H+i..2*H] = Buf21[3*H+1.. 4*H];

    // відправка даних в задачу  Send31
   MPI_Send(Buf22, 2*H+1, MPI_INT,3,1,MPI_COMM_WORLD);

    //обчислення  другої частини Ан
    for (int i = 1; i < H+1; i++)
          A2[i] =  B2[i] + C2[i]*x2;

    // формування  результату
    A[H+1 ..2*H] = A2;

    // збірка результату ( вектора А)
    // отримання частини результату Ан від задачі 1
    MPI_Recv(BufAH,H,MPI_INT,
                      1,1,MPI_COMM_WORLD,&status);
    A[1..H] = BufAH;

    // отримання частини результату  Ан від задачі 2
     MPI_Recv(BufAH, ,MPI_INT,3,1,
                              MPI_COMM_WORLD,&status);
     A[2*H+1..N] =  BufAH;

    // виведення результату
    printVector(A, N);
}// задача 2
//============================================================
if (rank == 3)  {
  int B3[H];
  int C3[H];
  int A3[H];
  int x3;
```



```
    int Buf3 [2*H +1]

    // отримання даних  від  задачі 2
     MPI_Recv(Buf3, 2*H + 1, MPI_INT, 1, 1,
                              MPI_COMM_WORLD, &status);
    X3 = Buf3[0];
    B3[1..H] = Buf3[1..H]
    C3[1..H] = Buf3[H +1 .. 2*H];

    //обчислення  третьої  частини Ан
    for (int i = 1; i < H+1; i++)
              A3[i] =  B3[i] + C3[i]*x3;

    // відправка результату АН в задачу 2
     MPI_Send(A3, H, MPI_INT, 2, 1, MPI_COMM_WORLD);

  }//  задача 3
  MPI_Finalize();
  return 0;
}
```

Додаткові засоби MPI для організації взаємодії потоків забезпечуються колективними операціями та використанням віртуальних топологій.

- **Запитання для модульного контролю**
  1. Особливості програмування для ВПКС зі локальною пам'яттю. ?
  2. Засоби організації повідомлень в мові Ada ?
  3. Засоби організації повідомлень в бібліотеці PVM ?
  4. Засоби організації повідомлень в бібліотеці MPI ?
  5. Ada. Оператори `entry` і `accept` ?
  6. Ada. Призначення оператору `select` ?
  7. PVM. Засоби формування повідомлень?
  8. MPI.Види функцій `Send/Receive` ?
  9. MPI.Концепція і реалізація колективних операцій для повідомлень?
  10. MPI. Віртуальні топології ?

СПИСОК ЛІТЕРАТУРИ

1. Вербіцький В. В. Паралельне програмування з використанням технології OpenMP : метод. вказівки / В. В. Вербіцький, А. Л. Максимов. – Одеса : Одес. нац. ун-т ім. І. І. Мечникова, 2022 – 48 с. [Електронний ресурс].
http:// dspace.onu.edu.ua/8080/handle/123456789/34441
2. Жуков І., Корочкін О. Паралельні та розподілені обчислення. Навч. посібник – Київ: Корнейчук, 2016 - 240 с.
3 Малашонок Г.І., Сідько А.А.. Паралельні обчислення на розподіленої памяті: OpenMP, Java, Math Patner: підручник. – Київ: НаУКМА, 2022.- 266 с.
4. Багатоядерне програмування на мові Ада: Англ мовою [Електронний ресурс] : навч. посіб.для студ. спеціальності 123 «Комп'ютерна інженерія», спеціалізації «Комп'ютерні системи та мережі», «Технології програмування для комп'ютерних системи та мереж» О.В.Корочкін ; КПІ ім. Ігоря Сікорського. – Електронні текстові данні (1 файл: 2,44 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2018. – 114 с. Гриф надано Методичною радою КПІ ім. Ігоря Сікорського.
5. Паралельні та розподілені обчислення. Вибрані розділи: Навч. посібник для здобувачів ступеня бакалавр за спеціальністю 123 «Комп'ютерні системи та мережі» / Корочкін О.В., Русанова О.В. – Київ : КПІ ім. Ігоря Сікорського, 2020. – 123с Електронний ресурс. Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол № 6 від 31.01.2020 р.) за поданням Вченої ради факультету інформатики та обчислювальної техніки (протокол № 4 від 25.11.2019 р.) http://ela.kpi.ua/handle /123456789/48224
6. Семеренко, В. П. Технології паралельних обчислень : навчальний посібник. – Вінниця : ВНТУ, 2018. – 104 с. ([Електронний ресурс] - Режим доступу:
https://www.researchgate.net/publication/334710599_V_P_Semerenko
7. Паралельне програмування: лабораторний практикум з дисципліни «Паралельне програмування» : [Електронний ресурс] : навч. посіб. для студ. спеціальності 123 – «Комп'ютерна інженерія» / О. І. Марченко, О. О. Марченко ; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 3,46 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2021. – 80 с.
8. Наконечна О. А., Ярмоленко Т. А., Алексеєнко В. В., Якимчук Б. М. Інструктивно-методичні рекомендації з дисципліни «Технології розподілених систем та паралельних обчислень» / уклад.: Оксана Наконечна, Тетяна Ярмоленко, Вікторія Алексеєнко, Богданна Якимчук. Житомир: Житомир: Вид-во ЖДУ ім. Івана Франка, 2023. -74 с.
9. Аксак Н. Г. Паралельні та розподілені обчислення: підруч. / Н. Г. Аксак, О. Г. Руденко, А.М. Гуржій. – Х.: Компанія СМІТ, 2009. – 480 с.
10. Паралельні та розподілені обчислення: навчальний посібник для вищих закладів освіти / К.Т. Кузьма, О.В. Мельник. – Миколаїв: ФОП Швець В.М., 2020. – 172 с.
11. Минайленко Р.М. М 61 Паралельні та розподілені обчислення: навч. посіб. — Кропивницький: Видавець Лисенко В. Ф., 2021. — 153 с.
12. Качко О.Г. Паралельне програмування. – Харків. нац. ун-т радіоелектроніки. – Харків : ХНУРЕ, 2016. – 403 с.
13. Коцовський В. М. Теорія паралельних обчислень : навчальний посібник. Ужгород : ПП «АУТДОР-Шарк», 2021. 188 с.
14. Минайленко Р.М. Паралельні та розподілені обчислення : навч. посіб. — Кропивницький: Видавець Лисенко В. Ф., 2021. 153 с.
15. Восс М., Асенхо Р., Рейндерс Дж. Паралельне програмування на C++ з допомогою бібліотеки TBB. К : ДМК Прес, 2020. 674 с.
16. Сандерс Дж., Кэндрот Ед. Технологія CUDA в прикладах. Введення в програмування графічних процесорів. К : ДМК Прес, 2017. 272с.
17. Малашонок Г., Сідько А. Паралельні обчислення на розподіленої пам'яті: OpenMP, Java, Math Partner : підручник. – Київ, НаУКМА, 2020.- 266 с.
18. Burns A., Wellings A. Real-Time Systems and Programming Languages. Addison – Wesley, 2001, – 386 p.
19. Burns A., Programming in Occam2. Reading. Addison–Wesley, 1995, – 326 p.
20. El – Rewini H, Lewis T., Distributed and Parallel Computing. – Manning Pub. Co.1998, – 430 p.
21. Hoare C.A.R. Communicating Sequential Processes,: Printice Hall,

International Series in Computer Science, Englewood Cliffs NJ, 1985. – 186 p.
22. Korochkin A. Ada95 as a Foundation Language in Computer Engineering Education in Ukraine – In Proceeding of the Ada-Europe International Conference on Reliable Software Technologies (Ada-Europe'99), (Santander, Spain, June 7 – 11, 1999), Lecture Notes in Computer Science , – № 1622, Springer, 1999, pp. 62 – 70.
23. Lea D. Concurrent Programming in Java: Design Principles and Patterns. Reading, MA: Addison – Wesley, 1999, – 344 p.
24. Loutsky G., Zhukov I., Korochkin A. Parallel Computing. – Kyiv, Kornechuk, 2007. -216 pp.
25. Taft S., Duff R., Brukardt R., Ploedereder T. Consolidated Ada Reference Manual. Language and Standard Libraries, Springer, Berlin: 2022, – 562 p.
26. Hyde D. Introduction to the Programming Language Occam. Bucknell University-1995.[Електронний ресурс] https://www.eg.bucknell.edu/~cs366/occam.pdf
27. Ringler R. C# Multithreaded and Parallel Programming. Birmingham: Packt Publishing, 2014. – 323с. ([Електронний ресурс] – Режим доступу: https://www.pdfdrive.com/c-multithreaded- and-parallel-programming-develop-powerful-c-applications-to-take-advantage-of-todays-multicore-hardware-e187077213.html
28. Terrell R. Concurrency in .NET. Shelter Island: Manning, 2018. – 534с. ([Електронний ресурс] - Режим доступу: https://www.pdfdrive.com/concurrency-in-net-modern-patterns-of-concurrent...
29. Quinn M.J. Parallel Programming in C with MPI and OpenMP. McGrawHill, 2004. – 544 p.
30. Himschoot P. Microsoft Blazor: Building Web Applications in .NET. – Apress, 2020.- 276 с. ([Електронний ресурс] - Режим доступу: https://www.pdfdrive.com/blazor-revealed-building-web-applications-in-ne...).
31. Asad A., Hamza A. The C# Programmer's Study Guide (MCSD). - Apress, 2017. – 475 с. ([Електронний ресурс] - Режим доступу: https://www.pdfdrive.com/the-c-programmers-study-guide-mcsd-exam-70-483-...).

Інтернет-ресурси

1 OpenMP Tutorial Part 1: A Core Elevents of OpenMP [Електроний ресурс] https://www.powershow.com/view1/1a8710-ZDc1Z/OpenMP_Tutorial_Part_1_The_Core_Elements_of_OpenMP_powerpoint_ppt_presentation
2 OpenMP Tutorial Part 2: Advanced OpenMP [Електроний ресурс https://www.powershow.com/viewfl/f25ff-ZDc1Z/OpenMP_ Tutorial_Part_2_Advanced_OpenMP_powerpoint_ppt_presentation
3. High Performance Parallel Programming - PowerPoint PPT Presentation [Електроний ресурс] https://www.powershow.com/view1/1f7b74-ZDc1Z/
4 Thw List of Top 500 Supercomputers Sites. TOP 500. URL : http://www.top500.org (date of access: 20.08.2022).
5. Message Passing Interface Tutorials. Lawrence Livermore National Laboratory. URL : https://hpc-tutorials.llnl.gov/mpi/ (date of access: 20.08.2022).
6. The OpenMP API specification for parallel programming. OpenMP. URL : www.openmp.org (date of access: 20.08.2022).
7. Програми, дослідження, обчислення. Nvidia. URL : https://www.nvidia.com (дата звернення: 20.08.2022).
8. Using MPI with C. GitHub. URL : https://curc.readthedocs.io/en/latest/programming/MPI-C.html. (date of access: 20.08.2022).
9. MPI-resourses GitHub. GitHub. URL : https://github.com/mpitutorial/mpitutorial (date of access: 20.08.2022).
10. Курс лекцій з паралельних обчислень. URL : https://sites.google.com/site/parcomp2012/lectures (дата звернення: 20.08.2022).
11. Steen M., Tanenbaum A. Distributed-systems – Pearson Education. – 2017. – 596 с. ([Електронний ресурс] - Режим доступу: https://www.pdfdrive.com/distributed-systems-3rd-edition-d189433770.html)
12. Burns B. Designing Distributed Systems. — O'Reilly Media, 2018. — 149 с. ([Електронний ресурс] - Режим доступу: https://www.pdfdrive.com/designing-distributed-systems-patterns-and-para...)

13 Troelsen A., Japikse P. Pro C# 7: With .NET and .NET Core. - APress. Inc.,2017.- 1372 с. ([Електронний ресурс] - Режим доступу: https://www.pdfdrive.com/pro-c-7-with-net-and-net-core-d183552783.html)
14.Ringler R. C# Multithreaded and Parallel Programming. Birmingham: Packt Publishing, 2014. – 323с. ([Електронний ресурс] - Режим доступу: https://www.pdfdrive.com/c-multithreaded-and-parallel-programming-develo...
15.Terrell R. Concurrency in .NET. Shelter Island: Manning, 2018. – 534с. ([Електронний ресурс] - Режим доступу: https://www.pdfdrive.com/concurrency-in-net-modern-patterns-of-concurren....
16.Stoecker M., Stein S. MTCS Exam 70-505: Microsoft .NET Framework 3.5 – Windows Forms Application Development. Microsoft Press, 2009. – 746с. ([Електронний ресурс] - Режим доступу: https://www.pdfdrive.com/mcts-self-paced-training-kit-exam-70-505-micros...
17. Horstmann C.S. Core Java. Volume 1- Fundamentals. – Pearson Education, 2015. – 864с. ([Електронний ресурс] - Режим доступу: https://www.pdfdrive.com/core-java-volume-ifundamentals-e34499118.html).
18. Freeman A. Essential Angular for ASP.NET Core MVC 3: A Practical Guide to Successfully Using Both in Your Projects. – Apress, 2019. – 350 с. ([Електронний ресурс] - Режим доступу: https://www.pdfdrive.com/essential-angular-for-aspnet-core-mvc-e48499390...).