

# ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ

*Навчальний посібник  
Друге видання*

Жуков І.А., Корочкін О.В.



«Корнійчук»  
Київ  
2014

**УДК 681.3.06**

**ББК 32.973**

**Ж48**

Р е ц е н з е н т: В.П.Тарасенко, доктор технічних наук, професор, заві-  
дувач кафедри системного програмування і спеціалізованих  
комп'ютерних систем Національного технічного університету України  
– «Київський політехнічний інститут»

**Жуков І.А., Корочкін О.В.**

**Ж48 Паралельні та розподілені обчислення. Навч. посіб.**

– 2-ге вид. виправл. і доп. К.: «Корнійчук», 2014. – 284 с.

**ISBN 966-7599-01-9**

Подано теоретичний і практичний матеріал для вивчення засобів програмування для паралельних та розподілених комп'ютерних систем. Розглянуто засоби створення процесів (потоків), а також організації взаємодії процесів, що ґрунтуються на моделях спільніх змінних та посиланні повідомлень. Значну увагу приділено вивченю підходів до вирішення задач взаємного виключення та синхронізації процесів за допомогою механізмів семафорів, мютексів, подій, критичних секцій, моніторів.

Наведено багато прикладів програмування з використанням мов Java, Ада, С#, бібліотек PVM, MPI, WinAPI, OpenMP.

Може бути використаний під час вивчення дисциплін “Паралельні та розподілені обчислення”, “Паралельне програмування”, “Високопродуктивні обчислювальні системи”, “Системи реального часу”, «Програмне забезпечення комп’ютерних систем» студентами, які навчаються за напрямками 050102 «Комп’ютерна інженерія», 050103 «Програмна інженерія».

Для студентів, аспірантів, викладачів та фахівців з програмного забезпечення паралельних та розподілених комп’ютерних систем.

Рекомендовано Міністерством освіти і науки України як науково-пізнавальний посібник для студентів вищих навчальних закладів (один вид 04.07.05 № 14/18.2–1540).

**ББК 32.973**

**ISBN 966-7599-01-9**

© Жуков І.А., Корочкін О.В., 2014



## ЗМІСТ

<b>ВСТУП . . . . .</b>	<b>8</b>
<b>1. Структури комп'ютерних систем . . . . .</b>	<b>12</b>
1.1. Паралельні комп'ютерні системи . . . . .	12
1.2. Комп'ютерні системи з розподільною (спільною) пам'яттю . . . . .	13
1.3. Багатоядерні комп'ютерні системи . . . . .	16
1.4. Комп'ютерні системи з розподіленою (локальною) пам'яттю . . . . .	18
1.5. Розподілені комп'ютерні системи . . . . .	22
Запитання для модульного контролю . . . . .	24
Завдання для самостійної роботи . . . . .	24
<b>2. Процеси . . . . .</b>	<b>26</b>
2.1. Поняття процесу . . . . .	26
2.2. Стан процесу. Керування процесами. . . . .	28
2.3. Процеси в мові Java . . . . .	31
2.4. Процеси в мові Ада . . . . .	35
2.5. Процеси в бібліотеці WinAPI . . . . .	39
2.6. Процеси в мові C# . . . . .	42
2.7. Процеси в бібліотеці MPI . . . . .	46
2.8. Процеси в бібліотеці OpenMP . . . . .	50
Запитання для модульного контролю . . . . .	55
Завдання для самостійної роботи . . . . .	56
<b>3. Паралельні алгоритми . . . . .</b>	<b>58</b>
3.1. Аналіз паралельного алгоритму . . . . .	58
3.2. Ярусно-паралельна форма алгоритму . . . . .	61
3.3. Паралельні алгоритми для задач лінійної алгебри . . . . .	62

3.4 Розроблення паралельного алгоритму . . . . .	72
Запитання для модульного контролю . . . . .	75
Завдання для самостійної роботи . . . . .	76
<b>4. Взаємодія процесів, що ґрунтуються на спільних змінних . . . . .</b>	<b>78</b>
4.1. Взаємодія процесів . . . . .	78
4.1.1. Завдання взаємного виключення . . . . .	79
4.1.2. Завдання синхронізації процесів . . . . .	82
4.2 Змінні що поділяються . . . . .	82
4.3. Семафори . . . . .	91
4.3.1 Семафори в мові Ada . . . . .	91
4.3.2 Семафори в Win32. . . . .	92
4.3.3 Семафори і мютекси в мові C# . . . . .	99
4.4 Критичні секції . . . . .	100
4.5. Монітори . . . . .	102
4.5.1. Монітори в мові Ada . . . . .	104
4.5.2. Монітори в мові Java . . . . .	110
4.6. Вирішення завдання взаємного виключення . . . . .	111
4.6.1 Вирішення завдання взаємного виключення в мові Ada . . . . .	112
4.6.2. Вирішення завдання взаємного виключення в Win32 . . . . .	117
4.6.3. Вирішення завдання взаємного виключення в мові Java . . . . .	124
4.6.4. Вирішення завдання взаємного виключення в мові C# . . . . .	126
4.6.5. Вирішення завдання взаємного виключення в OpenMP . . . . .	127
4.7. Вирішення завдання синхронізації . . . . .	128
4.7.1. Вирішення завдання синхронізації в мові Ada . . . . .	128
4.7.2. Вирішення завдання синхронізації в Win32 .	132
4.7.3. Вирішення завдання синхронізації в мові Java . . . . .	143
4.7.4. Вирішення завдання синхронізації в мові C#	147
4.7.5. Вирішення завдання синхронізації в OpenMP та MPI . . . . .	147

Запитання для модульного контролю .....	149
Завдання для самостійної роботи .....	150
<b>5. Взаємодія процесів, що ґрунтуються на посиланні повідомлень .....</b>	<b>152</b>
5.1. Загальна схема .....	152
5.2. Мова Оккам .....	153
5.3. Ада.Механізм рандеву .....	157
5.4. Бібліотека PVM .....	162
5.5. Бібліотека MPI .....	168
Запитання для модульного контролю .....	178
Завдання для самостійної роботи .....	178
<b>6. Розподілені обчислення .....</b>	<b>180</b>
6.1. Організація розподілених обчислень .....	180
6.1.1. Сокети .....	181
6.1.2. Віддалені процедури .....	182
6.2. Java.Сокети .....	186
6.3. Java.RMI .....	192
6.4. Ада.RPC .....	195
Запитання для модульного контролю .....	202
Завдання для самостійної роботи .....	202
<b>7. Приклади програмування .....</b>	<b>204</b>
7.1. Програмування для комп'ютерних систем зі спільною пам'яттю .....	204
7.1.1. Ада.Семафори .....	204
7.1.2. Win32.Семафори і мютекси .....	210
7.1.3. Ада.Захищені модулі .....	213
7.1.4. Java.Монітори .....	218
7.1.5. Бібліотека OpenMP .....	223
7.2. Програмування для комп'ютерних систем з локальною пам'яттю .....	227
7.2.1. Ада.Рандеву .....	227
7.2.2. Бібліотека PVM .....	234
7.2.3. Бібліотека MPI .....	238
7.3. Програмування для розподілених комп'ютерних	

систем . . . . .	243
7.3.1. Java.Сокети . . . . .	244
7.3.2. Ада.Віддалені процедури . . . . .	251
Завдання для самостійної роботи . . . . .	256
<b>СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ . . . . .</b>	<b>258</b>
<b>ІНТЕРНЕТ-РЕСУРСИ . . . . .</b>	<b>264</b>
<b>ПРЕДМЕТНИЙ ПОКАЖЧИК . . . . .</b>	<b>266</b>
<b>ДОДАТКИ . . . . .</b>	<b>270</b>
<b>    А. Мова Java . . . . .</b>	<b>270</b>
А.1. Клас Thread . . . . .	270
А.2. Клас ThreadGroup . . . . .	273
А.3. Інтерфейс Runnable . . . . .	274
<b>    Б. Мова Ada . . . . .</b>	<b>275</b>
Б.1. Пакет Synchronous Task Control . . . . .	275
Б.2. Пакет Asynchronous Task Control . . . . .	275
<b>    В. Бібліотека WinAPI . . . . .</b>	<b>276</b>
В.1. Функції очікування . . . . .	276
В.2. Функції для роботи із сокетами . . . . .	277
<b>    Г. Бібліотека MPI . . . . .</b>	<b>278</b>
Г.1. Базові функції . . . . .	278
Г.2. Типи даних . . . . .	279
<b>    Д. Бібліотека PVM . . . . .</b>	<b>280</b>
Д.1. Базові функції . . . . .	280
<b>    Е. Мова C# і платформа .NET . . . . .</b>	<b>281</b>
Е.1. Базові члени класу AppDomain . . . . .	281
Е.2. Базові типи простору імен System.Threading . . . . .	281
Е.3. Базові члени класу Thread . . . . .	282

<b>Ж. Бібліотека OpenMP . . . . .</b>	<b>283</b>
Ж.1. Базові прагми . . . . .	283
Ж.2 Додаткові функції . . . . .	284



## В С Т У П

*Поведінка навіть дуже коротких паралельних програм може виявиться на диво складною. Той факт, що паралельна програма з конкретним набором даних виконувалась правильно один або сто разів, не гарантує, що завтра ця програма з тими ж входними даними не завершиться з помилкою.*

*Рей Беррідж II.*

Перед вами друге (доповнене та виправлене) видання навчального посібника з дисципліни «Паралельні та розподілені обчислення», яка згідно стандарту підготовки бакалавра з комп’ютерної інженерії (2011 року) є нормативною і вивчається в багатьох технічних університетах України.

Матеріали первого видання пройшли апробацію в багатьох навчальних закладах України, які здійснюють підготовку фахівців з напрямів 050102 «Комп’ютерна інженерія» та 050103 «Програмна інженерія». Це університети міст Києва (НТУУ- КПІ, НАУ), Кривого Рогу, Миколаєву, Одеси, Ужгороду та інші. Автори вдячні викладачам та студентам цих університетів за корисні поради, які знайшли своє відображення у другому виданні.

Друге видання складається з семи розділів. Кожен розділ містить питання для модульного контролю та завдання для самостійної роботи.

У першому розділі розглянуто структури сучасних комп’ютерних систем, які ґрунтуються на паралельній та розподіленій архітектурі, а також особливості організації обчислювальних процесів в таких системах. Описуються паралельні системи з різними системами пам’яті: загальною та локальною. У другому виданні перший розділ доповнений матеріалом, який пов’язаний з

комп'ютерними системами, що ґрунтуються на використанні багатоядерних процесорів.

У другому роздлі розглядаються поняття процесу (потоку). Визначено поняття стану процесу, базові операції над процесами. Описано можливі підходи до створення процесів. Наведено реалізацію процесів в сучасних мовах та бібліотеках паралельного програмування програмування. В другому виданні цей роздл доповнений матеріалом, який пов'язаний з використанням мови C# та бібліотеки OpenMP.

В третьому роздлі розглянута побудова та аналіз паралельних алгоритмів. Визначено поняття коефіцієнтів прискорення та ефективності для оцінювання паралельного алгоритму. Розглянуто представлення паралельного алгоритму у вигляді ярусно-паралельної форми. Значна увага в роздлі придлена прикладам формування паралельних алгоритмів для задач лінійної алгебри, що пов'язані з векторна - матричними операціями.

Четвертий та п'ятий роздлі присвячені питанням взаємодїї процесів і вирішенню проблем, які виникають при організації такої взаємодїї. Визначені дві основних моделі взаємодїї процесів: модель, що ґрунтуються на спільніх змінних і модель, що ґрунтуються на передаванні повідомлень.

У четвертому роздлі розглядається модель взаємодїї процесів, яка базується на використанні спільних змінних. Розглянуто постановку і загальні схеми вирішення фундаментальних завдань, які виникають при використання цієї моделі: завдання взаємного виключення та завдання синхронізації процесів. Детально розглянуті механізми, які використовуються для розв'язування цих завдань: атомарні змінні, семафори, мютекси, подїї, критичні секції, замки, монітори. Представлено реалізацію цих механізмів у мовах та бібліотеках. Наведено приклади їх застосування. Роздл доповнений матеріалом, що пов'язаний з використанням змінних, що поділяються (atomic змінних).

У п'ятому роздлі розглядається модель взаємодїї процесів, яка базується на використанні посилки повідомлень. Розглянута загальна схема моделі та реалізація моделі у мові Ада та бібліотеці MPI.

Шостий роздл присвячені питанням програмування для розподілених (клusterних) комп'ютерних систем. Розглянуто особли-

вості створення програмного забезпечення для таких систем. Основна увага приділена організації передавання даних між вузлами системи. Описано побудову, реалізацію та використання механізмів, які ґрунтуються на сокетах та віддалених процедурах.

Сьомий розділ відведено прикладам програмування для паралельних і розподілених систем. окрім розглянуто створення паралельних програм для систем зі спільною та локальною пам'яттю, а також для розподілених систем. Показано виконання всіх необхідних етапів створення паралельних (розподілених) програм, які включають побудову паралельного математичного алгоритму, розробку алгоритму кожного процесу, створення схеми взаємодії процесів, а також розробку та налагодження безпосередньо програми з використанням сучасних мов і бібліотек паралельного програмування.

У додатах наведено дані про базові засоби роботи з процесами в бібліотеках OpenMP, MPI, Pthreads, WinAPI, PVM та в мовах Java, Ада, C#.

**Дані про авторів:**

**ЖУКОВ Ігор Анатолійович** – д-р. техн. наук, професор, Заслужений винахідник України, завідувач кафедри комп’ютерних систем та мереж Національного авіаційного університету. Член науково-методичної комісії Міністерства освіти і науки України з комп’ютерної інженерії, експертної ради з комп’ютерних наук і технологій при Державній акредитаційній комісії України. Відповідальний редактор збірника наукових праць «Проблеми інформатизації та управління».

Спеціаліст у галузі обчислювальних систем та мереж, апаратно-програмних засобів підвищення їх продуктивності. Викладає дисципліни «Комп’ютерні системи» та «Комп’ютерні мережі».

[itt@nau.kiev.ua](mailto:itt@nau.kiev.ua)

**КОРОЧКІН Олександр Володимирович** – канд. техн. наук, доцент кафедри обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут». Учений секретар науково-методичної комісії Міністерства освіти і науки, молоді та спорту України з комп’ютерної інженерії. Член Європейської спілки Ada-Europe та спілки ACM.

Фахівець у галузі програмування для паралельних та розподілених систем, а також програмування мовою Ада. Викладає дисципліни «Паралельні та розподілені обчислення», «Технології розподілених обчислень», «Паралельне програмування».

[avcora@gmail.ua](mailto:avcora@gmail.ua)



## РОЗДІЛ 1.

# Структури комп'ютерних систем

- 1.1. Паралельні комп'ютерні системи.
- 1.2. Комп'ютерні системи з розподільною (спільною) пам'яттю.
- 1.3. Багатоядерні комп'ютерні системи
- 1.4. Комп'ютерні системи з розподіленою (локальною) пам'яттю.
- 1.5. Розподілені комп'ютерні системи.

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

### 1.1. Паралельні комп'ютерні системи

Загальну структуру комп'ютерної системи, що ґрунтується на паралельній організації, можна подати як сукупність *вузлів* та *системи взаємодії* вузлів (рис. 1.1). Вузли системи використовують для обробки та зберігання інформації, систему



Рис. 1.1. Загальна структура паралельної комп'ютерної системи

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 *повнозв'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 КС, що ґрунтуються на загальній структурі:

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

## 1.2. Комп'ютерні системи з розподільною (спільною) пам'яттю

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

Зв'язок процесорів в таких системах здійснюється за допомогою *спільної пам'яті*. Взаємодія програм (процесів), що виконуються в кожному процесорі (це обмін даними або синхронізація), здійснюється за допомогою *спільних змінних*, які в оголошуються як глобальні.

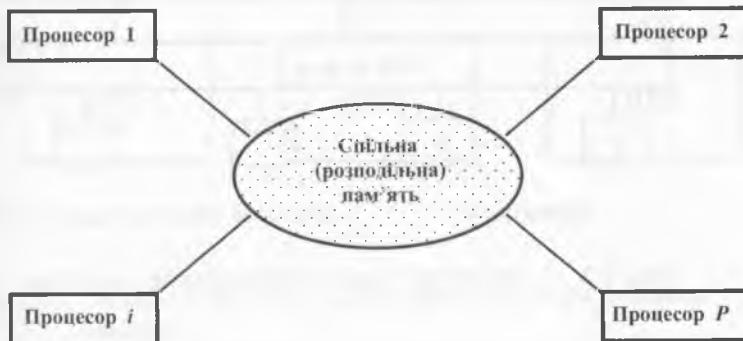


Рис. 1.2. Структура комп'ютерної системи з спільною пам'яттю

Така модель взаємодії отримала назву моделі 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. При програмної взаємодії один процес записує значення потрібної локальної змінної (яку треба передати в інший процес) у глобальну змінну, звідки інший процес читає 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 спроможність шини не дозволяє використовувати в системі багато процесорів. Симетричні мультипроцесорні системи мають назву однорідних (UMA – uniform memory access), оскільки час доступу кожного процесора до пам’яті та інших пристрійв системи одинаковий (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 uniform memory access) включають процесори, пам'ять яких організована за ієрархічним принципом. Така організація пам'яті дозволяє уникнути перева-

штажень на шини, але призводить до нерівномірного доступу до різних ділянок пам'яті.

На рис. 1.4 представлено загальну схему Р-процесої КС зі спільною пам'яттю.

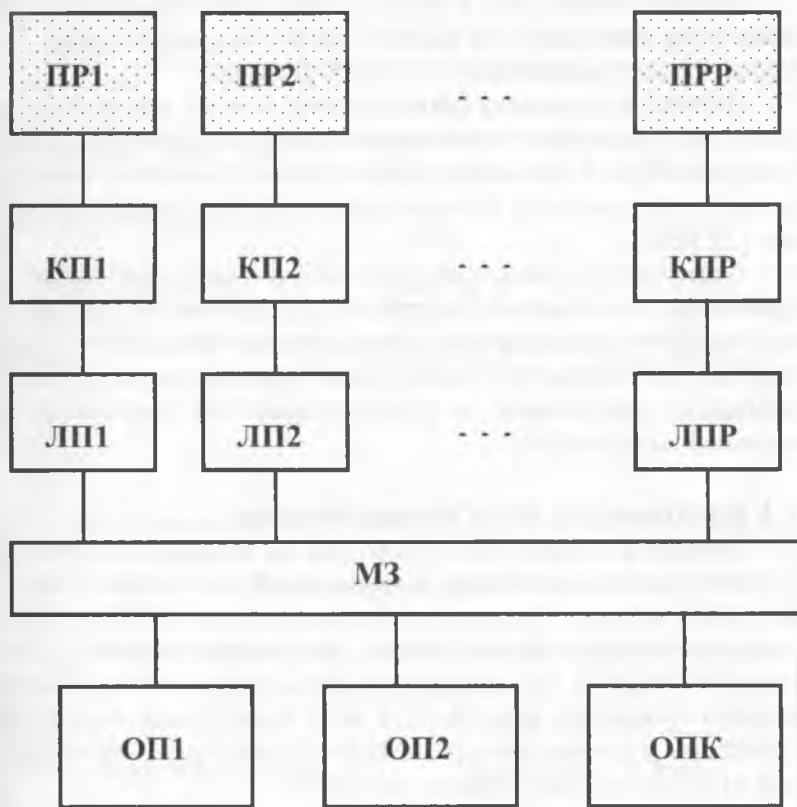


Рис. 1.4. Загальна структура комп'ютерної системи зі спільною пам'яттю

Вона включає :

- процесори (ПР1 - ПРР);
- кеш-пам'ять (КП1 - КПР);
- локальну пам'ять (ЛП1-ЛПР);
- блоки спільної пам'яті (ОП1 - ОПК);
- мережу зв'язку (МЗ).

Мережа зв'язку, яка з'єднує спільну пам'ять і процесори може бути побудована на підставі шини або кількох шин, для з'єднання яких використовуються комутатори.

Наявність в кожному процесорі кеш-пам'яті великого обсягу створює передумову для підвищення ефективності виконання програм через її ефективне використання. Результати експериментів з кеш-пам'яттю в паралельних системах наведено в працях [2], [54].

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.

### 1.3. Багатоядерні комп'ютерні системи

Сучасні паралельні КС будується на використанні багатоядерних процесорів. Основу багатоядерної комп'ютерної системи (БКС) складає процесор з кількома ядрами. Найбільш відомими виробниками багатоядерних процесорів є компанії Intel та AMD [70], [71]. На сьогодні кількість ядер в масових багатоядерних процесорах складає від 2 до 8. Крім кількості ядер таки процесори відрізняються організацією зв'язку ядер, використанням кеш-пам'яті різних рівнів та обсягів.

В таблиці 1.1 приведені технічні характеристики шести ядерних процесорів AMD та Intel.

На рис. 1.5 представлена структура 6-ї ядерного процесору Intel i7 - 980x. Процесор включає 6 ядер (типу Gulftown) з частотою 3,3 ГГц. Збільшена кеш-пам'ять L3 є спільною і може динамічно перерозподілятися між ядрами. Процесор підтримує всі передові технології Intel: Enhanced Halt State (C1E), Enhanced Intel Speedstep Technology, Hyper-Threading Technology, Execute Disable Bit, Intel

Таблиця 1.1

Шестиядерні процесори AMD і Intel

Процесори	AMD Phenom II X6 1100T	Intel Core i7-980x
Кількість ядер	6	6
Тактова частота GHz	3,3	3,3
Кеш-пам'ять рівня L1, КБ	124 x 6	64 x 6
Кеш-пам'ять рівня L2, КБ	512 x 6	256 x 6
Кеш-пам'ять рівня L3, КБ	6144	12288
Системна шина MHz	4,0 GT/s	6,4 GT/s
Технологія	45	32
Пам'ять	DDR-3/DDR-2	DDR-3/DDR-2

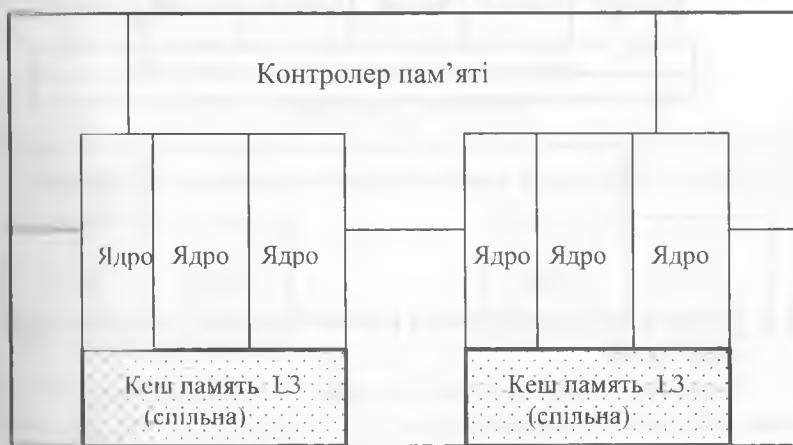


Рис. 1.5. Структура багатоядерного процесору (Intel i7 - 980x)

На рис. 1.6 представлена організація кеш-пам'яті в процесорі

Virtualization Technology , Intel Turbo Boost Technology, а також інструкції MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AES, EM64T.

Роботу з пам'яттю (до 24 ГБ) забезпечує трьох каналний контролер пам'яті з максимальної пропускною спроможністю 25,6 ГБ/с. Intel Core i7-980. Сумарний обсяг кеш-пам'яті всіх трьох рівнів (L1-L3) складає 13 518 Кб. Кеш-пам'ять рівня L3 є спільною і може використатися для взаємодії процесів через спільний змінний наряду з спільною пам'яттю.

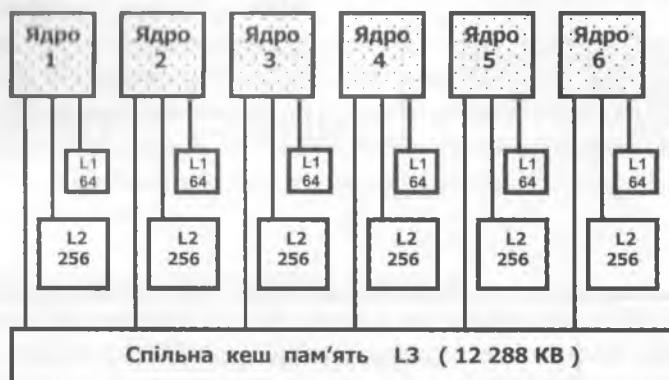


Рис. 1.6. Організація кеш-пам'яті в багатоядерних процесорах

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

Структуру комп'ютерної системи з розподіленою (локальною) пам'яттю показано на рис. 1.7. У вузлі системи розміщуються процесор (ПР) і локальна пам'ять (ЛП), для взаємодії вузлів використовується спеціальна система зв'язків, що зв'язує вузли за допомогою лінків (*link*) . Загальна структура КС С ЛП представлена на рис. 1.8.



Рис. 1.7. Структура комп'ютерної системи з локальною пам'яттю



Рис. 1.8. Загальна структура комп'ютерної системи з локальною пам'яттю

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

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

Реалізацію новного зв'язку процесорів у системах з розподіленою пам'яттю можна забезпечити лише для невеликої кількості процесорів. Для використання великої кількості процесорів їх сполучують за допомогою різних топологій: лінійних, кільцевих, деревоподібних, решітчастих (матричних), зіркових, гіперкубічних, повнозв'язаних та інших (рис. 1.9).

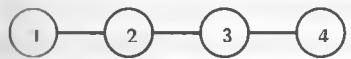
Кожна така структура характеризується визначеними топологічними параметрами:

- 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.

Топологічні характеристики структур комп'ютерних систем, які показано на рис. 1.9, наведено в табл. 1.2.

Таблиця 1.2. Топологічні характеристики КС

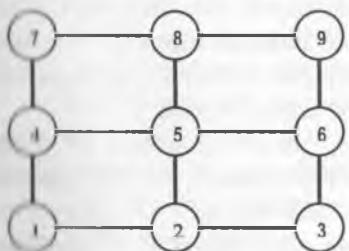
Структура	$P$	$D$	$R$	$S$
Лінійна	4	3	3	$\leq 2$
Кільцева	4	2	4	2
Решітка	9	4	12	$\leq 4$
Зірка	5	2	4	$\leq 4$
Гіперкуб	8	3	12	3
Повнозв'язна	4	1	6	3



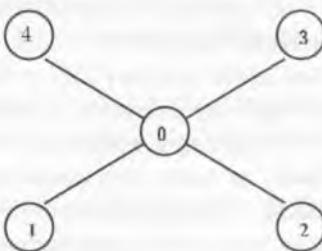
а) лінійна



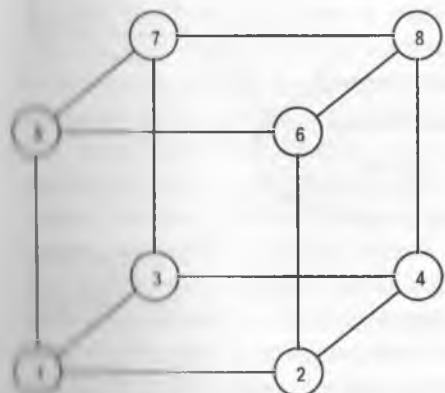
б) кільцева



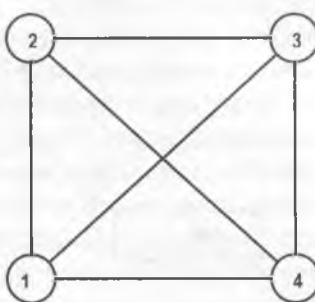
в) решітка



г) зірка



д) гіперкуб



е) повнозв'язна

Рис. 1.9. Структури КС з локальною пам'яттю:  
 а – лінійка; б – кільце; в – решітка; г – зірка;  
 д – гіперкуб; е – повнозв'язна

При програмуванні для КС важливим є діаметр ( $D$ ) системи, який визначає кількість необхідних дій по реалізації передавання даних між вузлами.

### 1.5. Розподілені комп'ютерні системи

Структуру розподіленої комп'ютерної системи (РКС) зображенено на рис. 1.10. 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.



Рис. 1.10. Структура розподіленої комп'ютерної системи

Модель взаємодії вузлів у РКС ґрунтуються на посиланні повідомлень, однак її реалізація відрізняється від прийнятої в системах з локальною пам'яттю, оскільки ґрунтуються на мережевих технологіях. Це потребує використання спеціальних програмних засобів – сокетів, віддалених методів, в яких задіяні протоколи, IP – адреси та ін. [12], [20], [22].

Розподілені комп'ютерні системи – 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 розширяються і побудовуються, 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 [25], [56].

Сучасні кластери системи будуються з використанням спеціального обладнання, яке повинне забезпечити на апаратному рівні рішення двох проблем:

- створення вузлів, що забезпечують виконання потужних обчислень;
- використання потужних систем, що забезпечує надійне передавання даних між вузлами.

Перша проблема вирішується за рахунок використання у вузлах багатоядерних процесорів, друга – через використання спеціалізованих мережених систем передавання даних (адаптерів, мереж, комутаторів), наприклад, системи *Infiniband*, які дозволяють значно прискорити передавання даних між вузлами системи, що традиційно є слабким місцем розподілених систем.

Багатоядерні кластерні системи забезпечують два рівня паралельної обробки. Перший рівень – це рівень вузла, де використання багатоядерного процесору дозволяє паралельну обробку на множині ядер. Другий рівень – це рівень вузлів, який дозволяє паралельну обробку на множині вузлів. Це ускладнює розробку програмного забезпечення для сучасних кластерних систем, яке потребує використання великої кількості взаємодіючих процесів у розподіленій програмі. Можливий підхід до покращення якості таких програм пов'язаний зі створенням відповідних математичних моделей [43].

Список (рейтинг) найбільш потужних КС світу *Top500* ([www.top500.org](http://www.top500.org)) вже кілька років обов'язково містить кластерні системи. У листопаді 2013 р. цей список (42-а редакція) налічував 82 кластерні системи з перших 100. Кластерна система DELL Stampede - PowerEdge C8220 займає сьоме місце, має 162 462 ядра (процесори Xeon E5-2680 8Cores, 2700Ghz), обсяг пам'яті 192 192 Гб, має максимальну продуктивність 8 520 ГПор/с. Систему взаємодії вузлів побудовано на *Infiniband FDR*.

Працює під ОС Linux з використанням MPI - бібліотеки *MVAPICH2*.

---

#### ➤ 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. 1.9, має найменший діаметр ?
9. В чому полягає відмінність організація кеш-пам'яті в багатоядерних процесорах Intel і AMD ?
10. Яка модель взаємодії процесів використовується в КС зі спільною пам'яттю?
11. Яка модель взаємодії процесів використовується в КС з локальною пам'яттю?

#### ➤ Завдання для самостійної роботи

1. Побудувати КС, що має зіркову структуру з  $P = 8$ .
2. Виконати модифікацію структури на рис. 1.9,  $\varepsilon$ , 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.1.9, з погляду надійності, у разі, якщо один з процесорів вийде з ладу.

6. Побудувати структури КС для гіперкуба, якщо значення ступеня гіперкуба дорівнюють 0, 1, 2 , 3, 4.
7. Розрахувати топологічні характеристики КС типу лінійки із  $P = 6$ .
8. Розрахувати топологічні характеристики КС типу кільця з  $P = 5$ .
9. Розрахувати топологічні характеристики КС типу решітки  $P = m \times m$ .
10. Розрахувати топологічні характеристики КС типу гіперкуб ступеня 4.
11. Побудувати повна зв'язану КС для  $P = 6$ . Визначити її топологічні параметри.



## РОЗДІЛ 2. Процеси

- 
- 2.1. Поняття процесу.
  - 2.2. Стан процесу. Керування процесами.
  - 2.3. Процеси в мові Java.
  - 2.4. Процеси в мові Ада.
  - 2.5. Процеси в бібліотеці WinAPI
  - 2.6. Процеси в мові C#
  - 2.7. Процеси в бібліотеці MPI.
  - 2.8. Процеси в бібліотеці OpenMP.
- 

В розділі розглянуто концепцію процесу, стани процесу й операції над процесами. Наведено приклади реалізацій процесів та операцій над ними, що застосовані в сучасних мовах бібліотеках паралельного програмування.

### 2.1. Поняття процесу

*Процес* – абстрактне поняття, що включає опис певних дій, пов’язаних з виконанням програми в комп’ютерній системі [12], [36]. При цьому процес оформлюється так, щоб система керування процесами в операційній системі (ОС) могла ефективно перерозподіляти ресурси системи (процесори, пам’ять, прилади введення-виведення, файли та інш.). 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 прикладною або системною програмою, що 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) [12]. Блок керування процесом – динамі-

чила структура даних, яка містить основну інформацію щодо процесу:

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

На рис. 2.1 представлено адресний простір процесу. Він містить три розділи: *текстовий* (для коду програми), *інформаційний* (для даних), *стековий* (для стеків програми) [68].

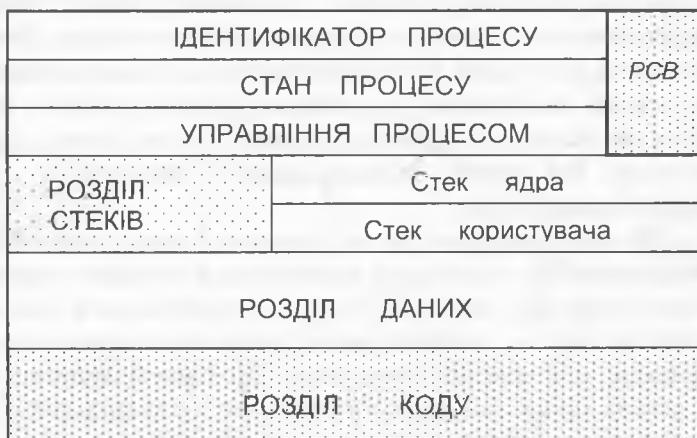


Рис. 2.1. Логічна структура процесу

Безліч програм, що виконуються в системі, визначають перший рівень паралелізму – рівень *процесів*. Це – багатозадачність. Другий рівень паралелізму - рівень *потоків*. Він визначає паралелізм всередині однієї програми. Це *багатопотоковість*.

Паралельне програмування пов'язано з використанням другого рівня паралелізму. *Потоки* - це процеси, що виконуються в паралельній програмі. Такі процеси отримали назву *легкі процеси* (*lightweight processes*). Традиційні процеси в ОС отримали назву *важких процесів* (*heavyweight processes*). Для позначення

легких процесів в різних мовах і бібліотеках паралельного програмування використовують різні терміни: потік (Java, C, C#), задача (Ада, MPI), процес.

В подальшому ми будемо розглядати виключно паралельне програмування і під терміном *процес* розуміти легкі процеси. Разом з терміном *процес* ми будемо використовувати також терміни *задача*, *потік*, *нитка* в залежності від конкретної реалізації процесу.

Тяжкі та легкі процеси мають багато спільного - ідентифікатор, стан, пріоритет та інші атрибути.

В чому полягає відмінність тяжких і легких процесів? Перш за все системні затрати в ОС на створення, підтримку і управління легкими процесами (потоками) значно нижче. Всі ресурси тяжкого процесу поділяються його потоками. Потоки не володіють ресурсами, вони конкурують за їх використання. Стежки потоків знаходяться у стековому розділі тяжкого процесу. Потік на відміну від тяжкого процесу *не має* свого адресного простору. Всі потоки тяжкого процесу знаходяться в одному адресному просторі.

Під час виконання тяжких процесів у комп'ютерній системі або звичайному комп'ютері відбувається постійне перемикання з одного процесу на інший. Те ж саме відбувається і для легких процесів. Але час, який витрачається на перемикання для легких процесів, менший, ніж для важких. Ще одна відмінність важких та легких процесів полягає в тому, що легкі процеси постійно взаємодіють, оскільки є частинами однієї програми (так звані *тісно зв'язані* процеси), в той час, як важкі процеси взаємодіють рідко (*слабко зв'язані* процеси).

## 2.2. Стан процесу. Керування процесами

Керування процесом включає виконання деяких операцій над ним, дозволяючи процесу пройти визначені стани (рис. 2.1). 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

- Виконання**
- звільнення процесора;
- Блокування**
- процес виконується на процесорі;
  - процес призупинений через очікування визначененої 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 введення-виведення та ін.

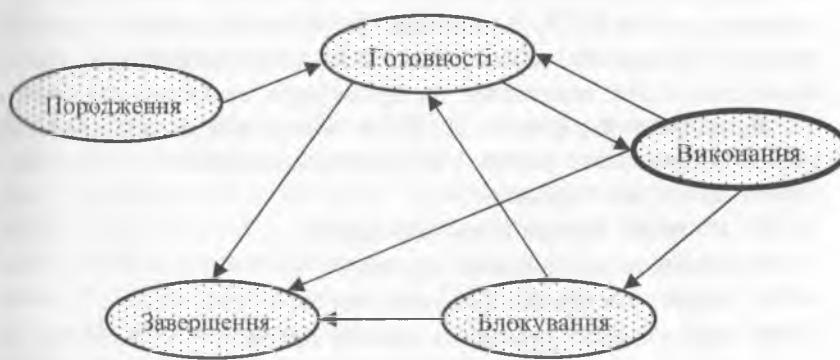


Рис. 2.1. Стани процесу

Процес також може мати особливий стан, який отримав на-шу *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.

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 взаємодія процесів.

Засоби роботи з процесами розподіляються на бібліотечні та мовні. Прикладами перших є засоби бібліотек WinAPI, PVM, MPI, OpenMP, Pthread, POSIX, які реалізовані у вигляді набору ресурсів (функцій, змінних, типів). Вони дозволяють роботу з процесами в будь-яких мовах програмування, зокрема і тих, які не мають вбудованих засобів роботи з процесами. Мови програмування, такі як C#, Java або Ада, безпосередньо мають засоби роботи з процесами, які вбудовані в мови, що забезпечує ефективне створення, виконання та організацію взаємодії процесів.

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

- об'явити процес (групу процесів);
- встановити пріоритет процесу;
- запустити процес на виконання,
- призупинити процес на визначений час;
- блокувати та розблокувати процес;
- організувати взаємодію з іншими процесами (синхронізація або передавання даних);
- завершити процес.

Існує декілька шляхів до створення процесу:

- через використання *спеціальних модулів* (класів), які дозволяють описати процес (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 *паралельних ділянок*, які можуть виконуватися паралельно (Occam, OpenMP).

В деяких мовах (бібліотеках) паралельного програмування використовуються комбінації наведених підходів. Так в мові C++ для створення процесу застосовують спеціальний модуль, якого визначаються через потокову функцію, а в бібліотеці OpenMP визначаються паралельні ділянки і процеси створюються шляхом копіювання окремих частин програми.

### 2.3. Процеси в мові Java

Процес в мові Java реалізований у вигляді *потоку* (thread) [27], [34], [45]. Java – об'єктно-орієнтована мова програмування, тому класи-потоки створюються з використанням спеціального класу Thread. 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 і буде визначати дії потоку. Щоб почати виконання потоку, необхідно викликати метод start(), який в свою чергу виклике відповідний метод run() потоку.

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

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.

### ❖ 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 ");
    }
}

class West extends Thread {
    public void run(){
        System.out.println("Process 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 ");
    }
}

```

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

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

Цей параметр Рівень визначає пріоритет потоку. Пріоритет відповідає 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;
```

Цей 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 “справедливe” 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 im'я){
        Name = im'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 А і В
        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. Процеси в мові Ада

Ада – відома мова програмування, яку було розроблено на замовлення Міністерства оборони США в 1980 році [16], [65]. Останній стандарт мови датується 2012 роком (Ада2012).

Призначення мови – розроблення великих програмних систем, робота яких характеризується високою надійністю. Це в першу чергу програмне забезпечення *систем реального часу*: лініяція (системи управління польотами (в більш ніж 40 країнах), літаки Be-200, TU-204, Boing – 737- 787, Airbus – 320, 330, 340, 380), космічні апарати (Ariane 4, 5, Atlas V, супутники, космічні станції), військові системи, системи управління атомними станціями, банківські і фінансові системи (Reuters, BNP), залізничний транспорт (European Train Control System (ETCS)), метро

(повністю автоматизована 14-а лінія в Парижі, Лондон, Каїр) та компільованому модулі). Мова немає явних засобів запуску задач, як, наприклад, в мові Java. Задача *автоматично стартує* при запуску підпрограми або при входженні в блок, у яких описана. Якщо задачі розміщені в основній програмі, то при запуску основної програми задачі починають виконуватися одночасно з головною програмою, яка, в свою чергу, завжди реалізується як задача і виконується паралельно з вкладеними 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 через *задачний тип* за допомогою посилального (access) типу та генератора new.

Підтримка і розвиток мови Ада забезпечується з боку Міністерства оборони США, підрозділу ACMSIGAda відомої організації Association for Computing Machinery (ACM) [75], [60] [62]. В розвитих європейських країнах існують організації ко- ристувачів мови Ада (Ada-Belgium, Ada-Denmark, Ada Deutschland, Ada-France, Ada-Spain, Ada-UK та інше), які з 1988 року було об'єднано в міжнародну організацію Ada-Europe [76]. [61].

Мова Ада знаходить застосування і в освіті [75], [61]. Деякі університети використають мову в якості першої мови програмування, а також в подальших курсах. Це Washington University (USA), Brighton University (UK), University of Stuttgart (Німеччина) та інші.

Університети, в яких використається мова Ада, було об'єднано у міжнародну спілку GAP (The Gnat Academia Program), яка налічує більш ніж 200 членів з 35 країн [77]. Програма підтримується компанією AdaCore - відомим розробником програмного забезпечення, яке пов'язане з мовою Ада, в тому числі компіляторів GNAT [78]. Найбільше представництво в GAP забезпечують університети США (69). Україна в GAP представляють чотири університети міст Київ (НТУУ-КПІ), Харків (ХНУ, НАУ-XAI), Чернівці (ЧНУ).

Ада – одна з перших мов програмування, яка має вбудовані засоби роботи з процесами.

Процес в мові Ада реалізований у вигляді *спеціального* модуля – *задача* (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 повинна бути описана в блоці, підпрограмі або пакеті (в

### Приклад 2.3

Ада. Опис 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 B 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 об'єктами якого будуть задачі. Задачі, які створені з використанням заданого типу, ідентичні. Однак реалізація задачного типу за допомогою *дискримінанта* дозволяє внести необхідні відмінності.

хідні особливості в кожну задачу для її ініціалізації. Механізм дискримінантів схожий на конструктори класів у мові Java.

### ❖ Приклад 2.4

```
-- Ада.Задачний тип з дискримінатором
procedure Lab24 is
    -- заданий тип з дискримінатором
    task type ЗадачнийТип (Номер : Integer);
    -- тіло заданиого типу
    task body ЗадачнийТип is
        Номер_Задачі : Integer := Номер;
    begin
        put ("Програвні підходи ");
        put (Номер_Задачі);
    end ЗадачнийТип;

    А : ЗадачнийТип (1); -- створення задачі
    В : ЗадачнийТип (2);
begin
    null; -- порожній 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 вибирювати 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` т 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;
```

## 2.5. Процеси в бібліотеці WinAPI

Бібліотека Win32 (Windows API) входить до складу ОС Windows і містить набір функцій, які призначені для роботи з процесами.

Для створення процесу в Win32 використовують функцію CreateThread(). Функція повертає ідентифікатор процесу, який є унікальним і визначає його в системі. Під час створення процесу визначається початковий адрес коду, з якого має виконуватись потік. Зазвичай, це назва функції (*потокової функції*), яка вже створена і буде виконуватися як процес.

Функція:

```
HANDLE Im'я_Потоку = CreateThread(
    LPSECURITY_ATTRIBUTES атр, // атрибут безпеки
    SIZE_T pc, // розмір стека
    LPTHREAD_START_ROUTINE 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: негайногого (0) або відкладеного (Create\_Suspended ), ім'я потоку.

#### ❖ Приклад 2.6

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 Win32 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 `SuspendThread()` і `ResumeThread()`. Функції `Sleep()` і `SleepEx()` блокують виконання процесу на заданий інтервал

часу. Функція `SwitchToThread()` блокує процес і передає керування іншому потоку.

## 2.6. Процеси в мові C#

Мова C# забезпечує програмування процесів через потоки (multithreading) [21]. Програма мовою C# завжди є багатопотоковою, тому що виконання програми розпочинається з створення основного потоку ("main" потоку) і породження додаткових потоків.

Процес в C# має назву *потоку* (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.

Процес створення потоку в мові C# схожий на створення потоків у мові Java. Для цього використовується спеціальний клас `Thread`, який містить ресурси для роботи з потоками. Але на відміну від мови Java цей клас не має методу `run()` для визначення дій майбутнього процесу. Дії процесу визначаються за допомогою *потокового методу*, який передається через конструктор класу `Thread` при створенні об'єкту-потоку.

### ❖ Приклад 2.7

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
    }

    static void MT1() {
        // дії потоку Td1
    }

    static void MT2() {
        // дії потоку Td2
    }
}
```

```
Console.WriteLine("Main потік стартував");  
}  
// Main -----  
  
// потокові методи  
// метод для виконання в першому потоці  
static void MT1() {  
    Console.WriteLine("Потік P1 стартував");  
  
    * * *  
    Console.WriteLine("Потік P1 завершений");  
}  
//MT1  
  
// метод для виконання в другому потоці  
static void MT2() {  
    Console.WriteLine("Потік P2 стартував ");  
  
    * * *  
    Console.WriteLine("Потік P2 завершений");  
}  
// MT2  
  
}// Thread_27
```

У прикладі 2.7 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 з потоками Td1 і Td1. Метод Start() є точкою входу в кожен потік. Він викликає відповідний потоковий метод (MT1() або MT2()).

При створенні потоків Td1 і Td2 використаний конструктор класу Thread, параметром якого є імена потокових методів MT1() або MT2(), які будуть виконуватись в 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, які дозволяють працювати з делегата-

ми, що забезпечують передавання посилань на функції через параметри конструктору.

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

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

Використання делегату `ThreadStart` при створенні потоку

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

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

#### ❖ Приклад 2.8

Створення і запуск на виконання потоку одночасно з основним методом.

```
class Thread28 {

    static void Main() {

        Thread.CurrentThread.Name = "Потік 1";

        // формування і запуск нового потоку
        Thread Td3 = new Thread(Обчислення);
        Td3.Name = "Потік_B";
        Td3.Start();

        // виконання методу Обчислення() в Main
        // одночасно з виконанням в потоці Td3
        // методу Обчислення();
    }
    // потоковий метод
    static void Обчислення() {
        Console.WriteLine ("Потік" +
                           Thread.CurrentThread.Name);
    }
} // Thread28
```

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

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

**Пріоритети потоків.** Значення пріоритетів потоків в C# визна-  
чаються наступним типом:

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

Потік, який створюється по замовченню отримує пріоритет Normal. Встановлення і контроль пріоритету потоку викону-  
ється за допомогою властивості Thread.Priority.

**Знищення потоку.** Знищенння 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 термін 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 2.9

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

}// Main -----
}

static void Печать() {
    Console.WriteLine("Поток стартував");
    . . .
    Thread.Sleep(100); // призупинення потоку
    . . .
}

}// Thread_29

```

## 2.7. Процеси 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 [15], [25], [38], [54].

Бібліотека (інтерфейс) MPI (Message Passing Interface) містить набір функцій, що дозволяють організовувати роботу з процесами в мовах, які не мають вбудованих засобів програмування процесів. У разі застосування бібліотеки MPI використовується мова послідовного програмування типу С або Фортран, процеси в якій програмуються за допомогою засобів інтерфейсу MPI. На сьогодні останньою версією інтерфейсу є MPI 3.0 [73].

Усі ресурси MPI бібліотеки (функції, типи, константи) починаються з префікса `mpi`.

Програма під MPI пишеться на любій мові програмування і включає необхідні ресурси бібліотеки 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;
}

```

Базові функції MPI бібліотеки, що потрібні для створення процесів, наведено в табл. 2.2:

Таблиця 2.2. Базові 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

Процес у MPI має назву *задача*. Задачі в MPI об'єднуються в іменовану групу - *комунікатор*. Комунікатор дозволяє звертатися до групи як до 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 комунікатору всі задачі мають унікальні ідентифікатори (*ранк*) – впорядковані числа, які розпочинаються з нуля (рис. 2.2). Спочатку всі задачі належать до одного базового комунікатору `mpi_Comm_World`, з якого потім формуються нові групи. MPI надає набір функцій для роботи з групами.

Програма може вмішувати кілька комунікаторів. Нумерація процесів всередині комунікатору 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()`.

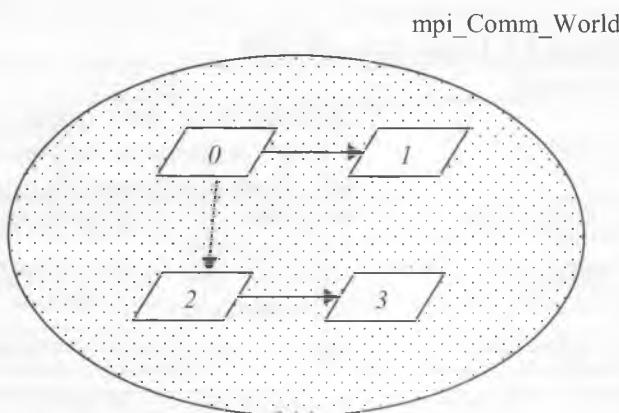


Рис. 2.2. MPI-комунікатор Взаємодія задач

Таким чином, створення процесів (задач) в MPI відбувається консольним викликом команди `mpr` системи, на вхід якої подається інформація про

- вхідну програму, яка містить `mpr` вкраплення (типи, змінні, функції)
- кількість процесів, що потрібне створити.

Під час створення паралельної програми відбувається:

- створення необхідної кількості копій заданої програми;
- створення комунікатора, де розміщаються копії (задачі);
- присвоєння кожній задачі унікального ідентифікатора (*Tid*) – номеру в комунікаторі.

Наступні функції дозволяють отримати з комунікатора необхідну інформацію про створені задачі:

```
mpi_Comm_size(MPI_Comm_World, &size),
mpi_Comm_rank(MPI_Comm_World, &rank),
```

де функція `Comm_size` – повертає розмір групи (кількість задач, що знаходяться в комунікаторі), а функція `Comm_rank` – ідентифікатор (*Tid*) задачі, в якої викликається ця функція.

Призначення комунікатора також полягає в забезпеченіні можливості взаємодії задач, які він об'єднує, через передавання повідомлень між задачами. Тому комунікатор також визначається як засіб (область) взаємодії задач.

### ❖ Приклад 2.10

Створення процесів, кожен з яких виводить на дисплей повідомлення про старт процесу. Кількість процесів задається під час запуску програми. Використовується стандартний комунікатор `mpi_Comm_World`.

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

int main(int argc, char *argv[]) {
    int кількість_процесів;
```

```

int мій_ранг;

mpi_Status статус;

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

/* Отримати номер поточного процесу
   з комунікатору в змінну мій_ранг      */
mpi_Comm_rank(MPI_COMM_WORLD, &мій_ранг);

/* Отримати загальну кількість процесів
   у комунікаторі в змінну
   кількість_процесів */
mpi_Comm_size(MPI_COMM_WORLD,
              &кількість_процесів);

/* Опис дій процесів програми */

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

mpi_Finalize();
return 0;

}/* main */

```

## 2.8. Процеси в бібліотеці OpenMP

Бібліотека OpenMP створена з метою уніфікації обчислень в СМП системах [68]. Широко підтримується компанією Intel як інструмент для створення паралельних програм для комп'ютерних систем з багатоядерними процесорами [69].

Особливість побудови паралельної програми в OpenMP полягає в тому, що вона формується на основі послідовної програми шляхом визначення в ній частин, які будуть виконуватися паралельно. Тобто паралельна програма в OpenMP являє собою низку послідовних та паралельних частин (рис. 2.3). Для створення паралельної програми бібліотека OpenMP надає ресурси у вигляді директив (прагм), змінних, процедур. Всі такі ресурси розпочинаються з префіксу `omp`.

Виконання програми розпочинається з потоку *Майстер* - основного потоку, який виконує послідовні частини програми, а також створює нові потоки 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 потік завжди чекає на завершення паралельної частини програми, яку він створює, і тільки потім продовжує своє виконання.

Базовою для визначення паралельних ділянок програми і створення потоків є директива `omp_parallel`:

```
#pragma omp_parallel [ опції ] { тіло }
```

При виконанні прагми `omp parallel` створюються копії *тіла* прагми. Кількість копій визначається за кількістю паралельних процесорів в системі (за замовчанням) або задається процедурою `set_num_threads(int num)` :

```
set_num_threads(3);
#pragma omp parallel
{
    Print("початок паралельної ділянки");
    b := 100;
    c := 250;
    d := 50;
}
```

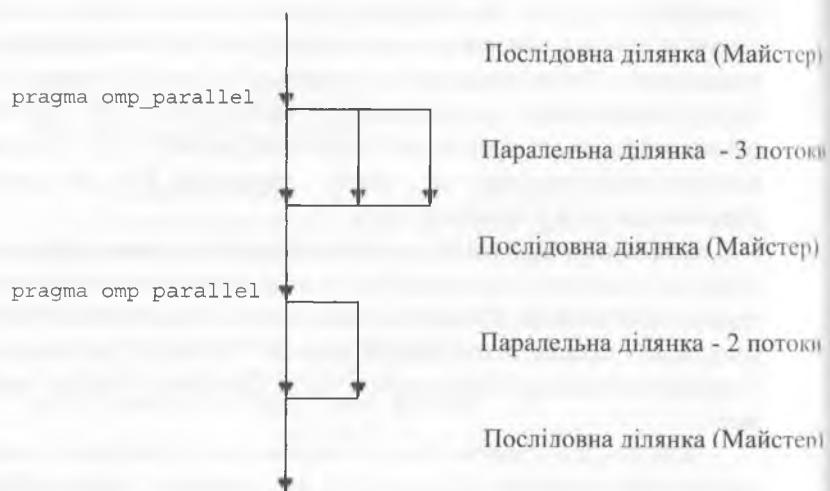


Рис. 2.3. Структура програми в OpenMP

Кожна копія отримує свій ідентифікатор, значення якого можна визначити за допомогою процедури `get_num_threads()`.

Копії є ідентичними, тому паралельні потоки що створено, виконують однакові дії. Використовуючи ідентифікатор кожного потоку можна змінювати дії окремих потоків:

```

set_num_threads(3);
#pragma omp parallel
{
    // дії для всіх потоків
    Tid:= get_num_thread(); //ідентифікатор потоку
    Print("початок паралельної ділянки потоку");
    Print(Tid);

    // дії для окремих потоків
    if(Tid=1)      b:= 100;
    if(Tid=2)      c:= 250;
    if(Tid=3)      d:= 50;
}

```

За допомогою прагми `sections` можна поділити тіло прагми `omp parallel` на *секції*, які будуть виконуватися в окремих потоках:

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        { b:= 100; }

        #pragma omp section
        { c:= 250;
            d:= 50;
        }
    }
}
```

Важливою особливістю бібліотеки є можливість паралельного виконання циклів. Директива `omp for` дозволяє організувати паралельне виконання тіла циклу, коли кожен процес послідовно виконує обчислення *частини* циклу. В прикладі кожен з двох процесів виконує по 50 додавань при формування вектору  $A$ .

```
set_num_threads(2);
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; I < 100; i++)
    {
        A[i] = B[i] + C[i];
    }
}
```

Директива `omp for` дозволяє організувати різні варіанти паралельного виконання тіла циклу. Загальна форма директиви:

```
pragma omp for [опція [, опція] ...]
-- цикл for
```

де опція це визначається як private, shared, firstprivate, lastprivate, reduction, ordered, schedule, nowait, if.

Наприклад, опція

```
schedule(тип [, Розмір блоку])
```

контролює розподіл роботи між потоками. Параметр *Розмір блоку* задає розмір кожного пакету на обробку потоком (кількість ітерацій циклу). Параметр *тип* може приймати наступні значення:

- static – ітерації циклу рівномірно розподіляються по потоках. В циклі із 100 ітерацій для 5 потоків кожен потік обробляє по 20 ітерацій;

- dynamic – робота розподіляється пакетами заданого розміру між потоками. Якщо потік закінчує обробку своєї порції даних, він захоплює наступну. Це дозволяє добитися кращого балансування завантаження між потоками;

- guided – даний тип розподілу роботи аналогічний dynamic, але розмір блоку змінюється динамічно в залежності від кількості необрблених ітерацій. Розмір блока поступово зменшується до вказаного значення.

Інші можливі значення параметру *тип* :

- private (для опису приватних змінних);
- shared (для опису спільних змінних);
- firstprivate, lastprivate (для формування приватних змінних при вході та виході з паралельної ділянки і зв'язування з глобальними змінними);
- reduction (для захисту спільних змінних);
- ordered (для визначення блоку в тілі циклу при виконанні в необхідному порядку);
- nowait (для мінімізації операцій синхронізації);
- if (паралельна робота ініціюється лише при виконанні вказаної умови).

### ❖ Приклад 2.11

Розпаралелювання циклу з використанням опцій.

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[])
{
    int A[100], B[100], C[100], i, n;

    // Формування вхідних масивів
    for (i = 0; i < 100; i++)
    {
        B[i] = i + 2;
        C[i] = 3 * i;
        A[i] = 0;
    }

    // Створювання паралельної ділянки
    #pragma omp parallel
            shared(A, B, C)
            private(i, n)
    {

        // Отримання номеру потоку
        n = omp_get_thread_num();

        // Паралельне виконання циклу
        #pragma omp for
            for (i = 0; i < 10; i++)
        {
            A[i] = B[i] + C[i];
        }
    }
}
```

### ➤ Запитання для модульного контролю

1. Наведіть визначення поняття процесу та його подання операційній системі.
2. Які існують види станів процесу? Що визначає перехід процесу в інший стан?

3. Як упорядковано процеси, 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 створення потоків у мові Java ?
6. У чому полягає відмінність механізму запуску процесів в мовах Ада та Java ?
7. Які особливості створювання процесів в бібліотеці MPI ? Призначення рангу задачі ? Що таке комунікатор ?
8. Які функції Win32 використовують для створювання процесів ?
9. Наведіть визначення тупика, причин його виникнення, наслідки та засоби боротьби з тупиками.
10. Які атрибути в мові Ада пов'язані із задачами ?
11. Алгоритм виконання функції `mpi_Comm_size()` ?
12. Алгоритм виконання функції `mpi_Comm_rank()` ?
13. Структура прагми `omp parallel` в бібліотеці OpenMP?
14. Наведіть засоби бібліотеки OpenMP для паралельної роботи з циклами.

➤ **Завдання для самостійної роботи**

1. Розробити Ада програму, яка містить три модулі - задачі. За допомогою встановлення пріоритетів задач визначити для них жорсткий порядок виконання. Використати оператор `delay` для перемикання з однієї задачі на іншу під час виконання.
2. Завдання 1 реалізувати за допомогою мови Java.
3. Завдання 1 реалізувати за допомогою засобів Win32.
4. Завдання 1 реалізувати за допомогою засобів MPI або PVM.
5. Розробити Ада-програму, яка містить чотири задачі у вигляді масиву  $T(i)$ , реалізованих за допомогою задачного типу. Виконати формування внутрішнього ідентифікатору ( $Tid$ ) кожної задачі за допомогою дискримінант інвестиції його на друк під час виконання задачі.

6. В завданні 5 забезпечити динамічне створювання  $P$  задач та їх запуск на виконання. Значення  $P$  вводиться безпосередньо перед запуском програми.
7. Завдання 5 реалізувати за допомогою мови Java, використовуючи конструктор класу, що є потоком.
8. 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()`.



## РОЗДІЛ 3.

# Паралельні алгоритми

- 3.1. Аналіз паралельного алгоритму.
- 3.2. Ярусно-паралельна форма алгоритму
- 3.3. Паралельні алгоритми для задач лінійної алгебри
- 3.4. Розроблення паралельного алгоритму

В розділі розглянуто принципи побудови та аналізу паралельних алгоритмів. Наведено приклади паралельних алгоритмів для задач лінійної алгебри.

### **3.1. Аналіз паралельного алгоритму**

Ефективна реалізація будь-якої задачі в паралельній системі можлива тільки тоді, коли алгоритм її розв'язання поданий в паралельній формі, тобто виконано розробку паралельного алгоритму рішення задачі. З погляду паралельних властивостей (внутрішнього паралелізму) усі задачі можна поділити на три групи:

- повністю паралельні задачі;
- частково паралельні задачі;
- задачі, які не допускають паралельної обробки.

Прикладом повністю паралельної задачі є операція додавання матриць, яку можна виконати за один крок, на якому паралельно 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, скалярна операція  $a = b + c + d$  не може бути поданою у вигляді паралельного алгоритму.

До появи паралельних комп'ютерних систем використовувались виключно послідовні алгоритми, орієнтовані спочатку на

людину, а потім на однопроцесорні комп'ютери. Проблема розроблення паралельних алгоритмів з'явилась з появою реальних паралельних систем. Паралельна математика, яка вивчає методи і засоби побудови та аналізу паралельних алгоритмів бурхливо почала розвиватися у 60–70-х роках минулого століття [6], [7], [28], [29]. Науковці отримали цікаві результати що до паралельних властивостей цілого ряду задач. При цьому для окремих задач довелося відмовитись від традиційних підходів до їх рішення і розробити цілком нові, які дозволили побудувати ефективні паралельні алгоритми. З'ясувалося, що оцінка ефективності різних паралельних алгоритмів – складне завдання, тому що для цього треба враховувати різні додаткові фактори, наприклад, час передавання даних між процесорами.

Для спрощення аналізу паралельних алгоритмів було запропоновано *концепцію необмеженого паралелізму*, яка ґрунтувалась на таких поняттях [23], [47]:

- 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 КС, оскільки не враховується більшість додаткових параметрів цієї системи, але вона дозволяє на етапі проектування паралельних алгоритмів оцінити для них час виконання, визначити кількість процесорів і обрати оптимальний алгоритм.

Концепція необмеженого паралелізму дозволяє спростити розрахунок *коєфіцієнта прискорення* ( $Kn$ ), який показує на скільки скорочується час виконання паралельної програми в паралельній системі з  $P$  процесорами ( $T_p$ ) в порівнянні з часом виконання послідовної програми в однопроцесорної системі ( $T_1$ ):

$$Kn = T_1 / T_p$$

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

$$Ke = T_1 / (Tp * P)$$

Для поєднання необмеженого паралелізму в концепції необмеженого паралелізму з обмеженою кількістю процесорів, що використається реально, можна застосувати лему Брента [5]:

**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 множини *бінарних* операцій, алгоритм виконання яких можна подати бінарним деревом.

**Теорема Мунро – Петерсена [5].** Якщо в комп'ютерній системі з  $p$  процесорів виконується обчислення скалярної величини, яке потребує  $m$  бінарних операцій, то необхідний час  $tp$  визначається як :

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

•  $\lceil x \rceil$  – найменше ціле число, більше, або таке, що, дорівнює  $x$ ; логарифм береться за основою 2;  $^{\wedge}$  – означає ступінь.

Із цієї теореми Мунро – Петерсона можна оцінити час паралельного виконання операції  $M1 = MB * MC$  множення

матриць розміром  $n \times n$ . Ця операція потребує  $n$  множень і  $n-1$  додатків для формування одного елемента матриці  $MA$ . Тобто  $m = 2n - 1$ .

Звідки

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

і на підставі теореми Мунро-Петersona отримуємо

$$t_p >= \begin{cases} \lceil \log p \rceil + \lceil (2n - 2 \wedge \lceil \log p \rceil) / p \rceil & \text{якщо } 2n-1 >= 2^{\lceil \log p \rceil} \\ \lceil \log n \rceil + 1 & \text{в інших випадках..} \end{cases}$$

Якщо кількість процесорів необмежена, то оптимальний час для операції дорівнює  $\lceil \log p \rceil + 1$ . При цьому на першому кроці виконуються паралельно всі множення ( $n * n * n$ ), а потім за  $\lceil \log n \rceil$  кроків – паралельні додавання. Для  $n = 100$  час  $t_p$  дорівнює 8, при цьому знадобиться 1 000 000 процесорів.

### 3.2 Ярусно-паралельна форма алгоритму.

Подання паралельного алгоритму може бути виконано різними способами. *Ярусно-паралельна форма* (ЯПФ) для представлення паралельного алгоритму являє собою набір ярусів, в яких операції виконуються паралельно (рис. 3.1) [5], [6]. Висота ЯПФ ( $H$ ) – кількість ярусів, ширина  $i$ -го ярусу ( $R_i$ ) – кількість операцій в ярусі, ширина ЯПФ ( $R$ ) – найбільша з ширини ярусів. Виконання ЯПФ здійснюється за ярусами. Тому час виконання задачі, алгоритм якої 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 обрати алгоритм з необхідним часом виконання або із заданою кількістю процесорів.

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

$$Ke = T_1 / (Tp * P)$$

Для поєднання необмеженого паралелізму в концепції необмеженого паралелізму з обмеженою кількістю процесорів, що використається реально, можна застосувати лему Брента [5]:

**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 множини *бінарних* операцій, алгоритм виконання яких можна подати бінарним деревом.

**Теорема Мунро – Петерсена [5].** Якщо в комп'ютерній системі з  $p$  процесорів виконується обчислення скалярної величини, яке потребує  $m$  бінарних операцій, то необхідний час  $tp$  визначається як :

$$tp \geq \begin{cases} \lceil \log p \rceil + \lceil (m+1 - 2^{\lceil \log p \rceil})/p \rceil & \text{якщо } m \geq 2^{\lceil \log p \rceil} \\ \lceil \log(m+1) \rceil & \text{в інших випадках,} \end{cases}$$

де  $\lceil x \rceil$  – найменше ціле число, більше, або таке, що, дорівнює  $x$ ; логарифм береться за основою 2;  ${}^{\wedge}$  – означає ступінь.

За допомогою теореми Мунро–Петерсона можна оцінити час паралельного виконання операції  $MA = MB * MC$  множення

матриць розміром  $n \times n$ . Ця операція потребує  $n$  множень і  $n-1$  додатків для формування одного елемента матриці  $MA$ . Тобто  $m = 2n - 1$ .

Звідки

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

і на підставі теореми Мунро-Петерсона отримуємо

$$t_p = \begin{cases} \lceil \log p \rceil + \lceil (2n - 2 \wedge \lceil \log p \rceil) / p \rceil & \text{якщо } 2n-1 >= 2^{\lceil \log p \rceil} \\ \lceil \log n \rceil + 1 & \text{в інших випадках..} \end{cases}$$

Якщо кількість процесорів необмежена, то оптимальний час для операції дорівнює  $\lceil \log p \rceil + 1$ . При цьому на першому кроці виконуються паралельно всі множення  $(n * n * n)$ , а потім за  $\lceil \log n \rceil$  кроків – паралельні додавання. Для  $n = 100$  час  $t_p$  дорівнює 8, при цьому знадобиться 1 000 000 процесорів.

### 3.2 Ярусно-паралельна форма алгоритму.

Подання паралельного алгоритму може бути виконано різними способами. *Ярусно-паралельна форма* (ЯПФ) для представлення паралельного алгоритму являє собою набір ярусів, в яких операції виконуються паралельно (рис. 3.1) [5], [6]. Висота ЯПФ ( $H$ ) – кількість ярусів, ширина  $i$ -го ярусу ( $R_i$ ) – кількість операцій в ярусі, ширина ЯПФ ( $R$ ) – найбільша з ширини ярусів. Виконання ЯПФ здійснюється за ярусами. Тому час виконання задачі, алгоритм якої 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 обрати алгоритм з необхідним часом виконання або із заданою кількістю процесорів.

Для ЯПФ, яка показана на рис. 3,1, параметри дорівнюють:

$$H = 4, R_1 = 4, R_2 = 2, R_3 = 3, R_4 = 2, R = \max(4, 2, 3, 2) = 4.$$

Знання цих параметрів дозволяє розрахувати:

- час розв'язування задачі в однопроцесорній системі:

$$T_l = R_1 + R_2 + R_3 + R_4 = 11;$$

- час розв'язування задачі в багатопроцесорній системі:

$$T_p = H = 4;$$

- кількість процесорів, потрібних на кожному кроці обчислення ЯПФ:

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

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

$$P = R = 4;$$

- коефіцієнт прискорення:

$$K_n = T_l/T_p = 11/4 = 2,75;$$

- коефіцієнт ефективності:

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

### 3.3 Паралельні алгоритми для задач лінійної алгебри.

Розглянемо та оцінимо паралельні алгоритми для деяких задач лінійної алгебри із застосуванням концепції необмеженого паралелізму.

- **Додавання векторів.** Операція додавання векторів розміру  $n$

$$A = B + C$$

виконується за спiввiдношенням  $a_i = b_i + c_i$ , де  $i = 1..n$ .

Час виконання задачі в однопроцесорній КС дорівнює  $T_1 = n$ , час виконання в паралельній КС дорівнює  $T_P = 1$  для  $P = n$ , коефіцієнт прискорення  $K_n = n$ , тобто це ідеальна для виконання в паралельній системі задача.

Якщо кількість процесорів обмежена  $P < n$ , то паралельний алгоритм можна представити у вигляді

$$A_H = B_H + C_H,$$

де  $H = n/P$ ;  $A_H$  –  $H$  елементів вектора  $A$  (підвектор). Кожен процес паралельно здійснює обчислення своєї частини результата  $A_H$ . (рис.3.2).

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

$$a = (B * C)$$

виконується за співвідношенням

$$a = b_1 * c_1 + \dots + b_i * c_i + \dots + b_n * c_n, \quad i = 1 \dots n.$$

Час виконання задачі в одно процесорній 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  $\lceil \log n \rceil$  (дів. теорему Мунро-Петersona).

Таким чином, час виконання операції скалярного множення  $T_P = 1 + \lceil \log n \rceil$  для  $P = n$ , коефіцієнт прискорення  $K_n = (2*n - 1)/(1 + \lceil \log n \rceil)$ , тобто це частково паралельна задача, в якій чергуються паралельні і послідовні частини.

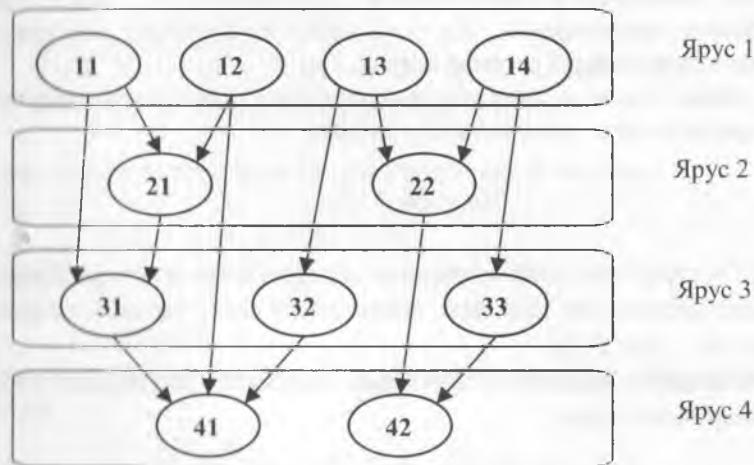


Рис. 3.1. Ярусно-паралельна форма алгоритму

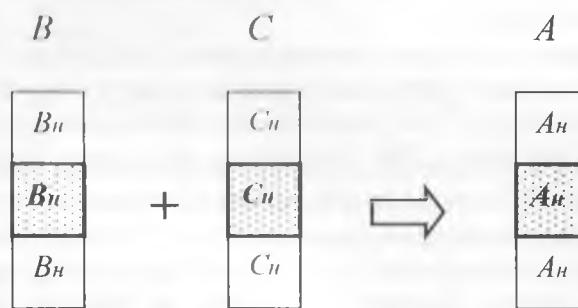


Рис. 3.2. Операція паралельного додавання векторів  $A = B + C$

Якщо кількість процесорів  $P$  обмежена і значно менша за  $n$ , то паралельний алгоритм можна надати у вигляді

$$a_i = (B_H * C_H), \quad i = 1 \dots P.$$

$$a = a + a_i$$

- **Множення вектора на матрицю.** Операція множення вектора з  $n$  елементів на матрицю  $n \times n$

$$A = B * MC$$

виконується за співвідношенням

$$a_i = b_1 * c_{1i} + \dots + b_i * c_{ii} + \dots + b_n * c_{ni} \quad i = 1 \dots n.$$

Час виконання задачі в однопроцесорній КС  $Tl = n * (2n - 1)$ . В паралельному алгоритмі використано схему здвоювання, тому час виконання в паралельній КС  $Tp = 1 + \lceil \log n \rceil$  якщо  $P = n * n$ , і коефіцієнт прискорення дорівнює  $Kn = (n * (2n - 1)) / (1 + \lceil \log n \rceil)$ , тобто це частково паралельна задача, в якій чергуються паралельні і послідовні частини.

Якщо кількість процесорів обмежена і  $P < n$ , то паралельний алгоритм можна представити у вигляді (рис. 3.3)

$$A_H = (B * MC_H),$$

де  $MC_H$  -  $H$  рядків матриці  $MC$ .

- **Додавання матриць.** Операція додавання матриць  $n \times n$

$$MA = MB + MC$$

виконується за співвідношенням

$$a_{ij} = a_{ij} + c_{ij}, \quad \text{де } i,j = 1..n.$$

Час виконання задачі в однопроцесорній КС дорівнює  $T_1 = n^2n$ , час виконання в паралельній КС дорівнює  $T_P = 1$  для  $P = n^2n$ , коефіцієнт прискорення  $K_P = n^2n$ , тобто це теж ідеальна для виконання в паралельній КС задача, але слід звернути увагу на те, як збільшилася кількість процесорів в КС, що необхідна для розв'язання задачі за одиницю часу.

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

$$MA_H = MB_H + MC_H,$$

де -  $MA_H$  –  $H$  рядків матриці  $MA$ .

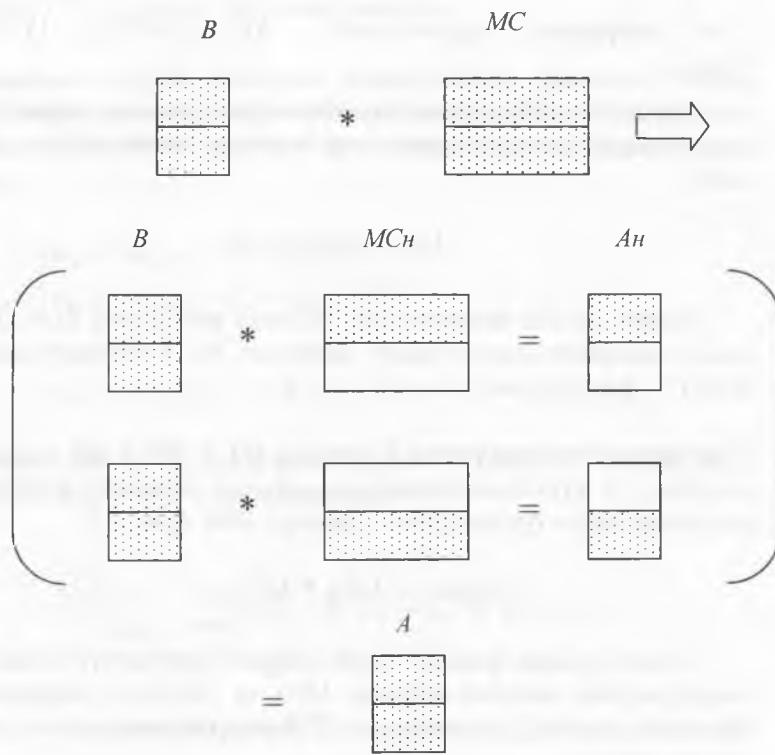
- Множення матриць.** Операція множення матриць є основною операцією лінійної алгебри і основою багатьох обчислювальних задач. Характеризується великою кількістю операцій ( $n^3$ ), де  $n \times n$  – розмір матриці. Існує багато різних паралельних алгоритмів множення матриць [ 29 ].

**Базовий алгоритм** Операція множення матриць

$$MA = MB * MC$$

виконується за співвідношенням

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

Рис. 3.3. Паралельний алгоритм операції  $A = B * MC$

Час виконання задачі в однопроцесорній КС  $T_1 = n * n * (2n - 1)$ , час виконання в паралельній КС  $T_p = 1 + \lceil \log n \rceil$  якщо  $P = n^3$ , коефіцієнт прискорення  $K_n = n * n * (2n - 1) / (1 + \lceil \log n \rceil)$ .

Якщо кількість процесорів обмежена і значно менша за  $n$ , паралельний алгоритм множення матриць можна подати у вигляді

$$MA_H = MB_H * MC.$$

Кожен процес формує свою частину результату  $MA_H$  ( $H$  рядків), використовуючи повну матрицю  $MC$  і частину матриці  $MB_H$  ( $H$  рядків) (рис.3.4).

**Стрічково-блокова схема.** Операція  $MA = MB * MC$  множення матриць за стрічково-блоковою схемою дозволяє формувати матрицю  $MA$  по блоках  $MA_{Hn}$  розміру  $H*H$  (рис. 3.5)

$$MA_{Hn} = MB_H * MC_n.$$

Кожен процес формує блок матриці результату  $MA_{Hn}$ , використовуючи частини матриць  $MB_H$  та  $MC_n$ , що відрізняє цю схему від базової, де матриця  $MC$  береться повністю і є спільним ресурсом.

В роботі [70] визначений час виконання матричного множення для послідовного та паралельного алгоритмів ( $T_{p1}$  - базового і  $T_{p2}$  - блочного) з врахуванням апаратних параметрів системи (оперативної пам'яті і кеш - пам'яті)

$$T_1 = n * n (2n - 1) + e(2 * n * n * n + n * n) * k + 64 * z$$

$$T_{p1} = (n * n / p) * 2n - 1 + n * n * n (1 + 1 / p) + n * n * k + 64 / z$$

$$T_{p2} = (n * n / p) * (2n - 1) + n * n * n (e + 1 / r) + n * n * k + 64 / z$$

де  $z$  - пропускна здатність доступу до оперативної пам'яті,  $e$  - частота виникнення кеш - промахів,  $k$  - латентність оперативної пам'яті,  $r * r$  - розмір блочної решітки.

- **Рішення систем лінійних алгебраїчних рівнянь. Метод Якобі (простих ітерацій).** Систему з  $n$  лінійних алгебраїчних рівнянь (СЛАУ)  $MA^*X = B$  можна представити наступним чином

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2, \\ \dots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n. \end{cases}$$

Існує багато методів розв'язання СЛАУ [28]. Ітераційний метод Якобі описується формулою

$$x_i^{k+1} = \frac{1}{a_{ii}} \left( -\sum_{j \neq i} a_{ij}x_j^k + b_i \right), \quad i = 1 \dots n, \quad k = 0, 1, \dots,$$

де  $a_{ii}$  - елементи матриці  $MA$ ,  $x_i$  - вектора  $X$ ,  $b_i$  - вектора  $B$ .

Метод Якобі дозволяє побудувати ефективний паралельний алгоритм, де кожна задача обчислює  $H$  елементів вектора  $X_H^{k+1}$  для наступної ітерації ( $k+1$ ), використовуючи значення вектора  $X^k$  попередньої ітерації:

$$X_H^{k+1} = (X^k * MA_H + B_H) / a_{ii}$$

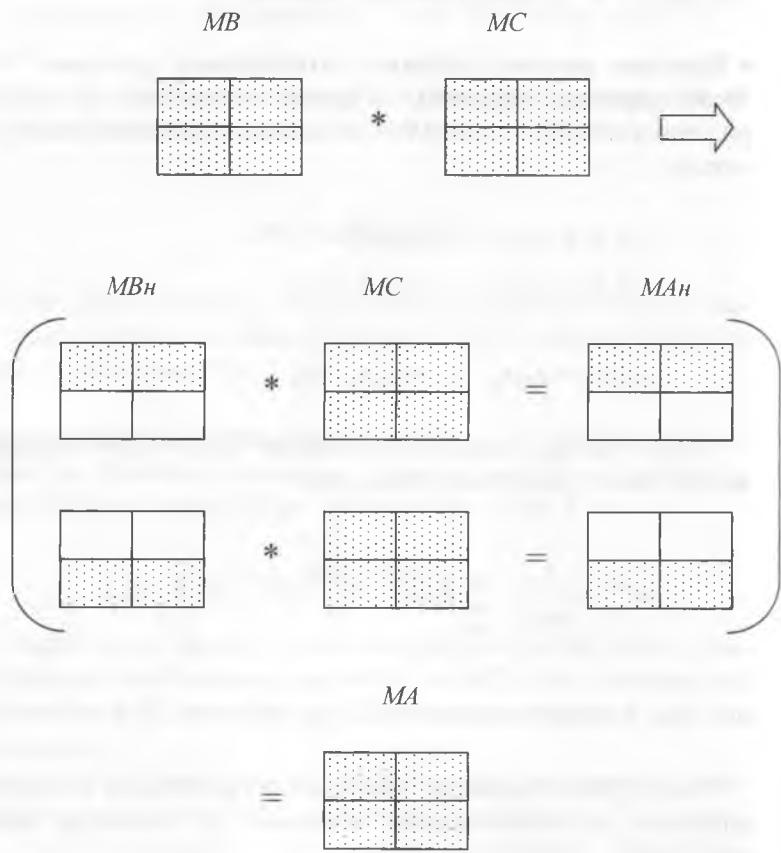


Рис. 3.4. Базовий паралельний алгоритм операції  $MA = MB * MC$

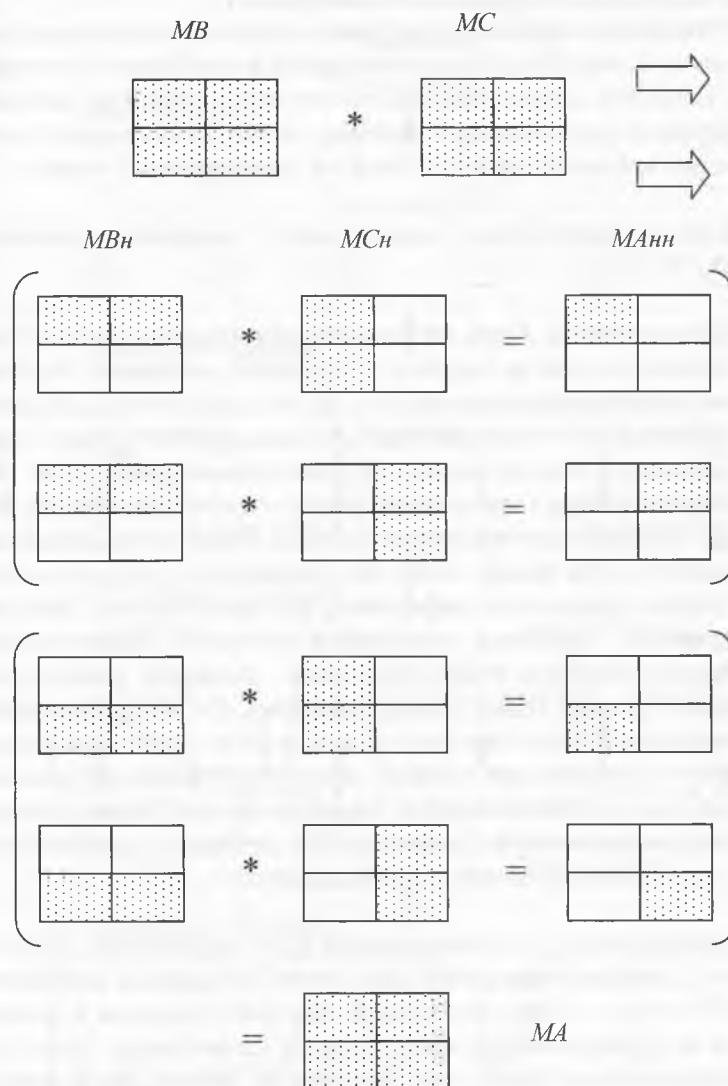


Рис. 3.5. Стрічково-блоковий паралельний алгоритм операції  $MA = MB * MC$

### 3.4. Розроблення паралельного алгоритму

Розроблення алгоритму розв'язування завдання завжди була важливим і найбільш складним етапом розроблення програми [18], [28], [48]. Для паралельного алгоритму цей етап має свої особливості і складається з окремих кроків: *декомпозиції, проектування взаємодії задач, об'єднання, планування обчислень.*

Етапи розроблення паралельного алгоритму показано на рис. 3.6.

- 1. Декомпозиція.** Крок, на якому вхідне завдання аналізується і розкладається на частини – підзадачі, виконання яких може бути паралельним (на рис. 3.6 це підзадачі  $A \dots P$ ). 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 підзадачами. Наступне планування обчислень (авантаження процесорів) буде спрощеним, якщо підзадачі мають одинаковий розмір. Щодо розміру підзадач, то він пов'язаний з поняттям «зерна» алгоритму, яке, в свою чергу, пов'язане з рівнем паралелізму і типом використованої паралельної системи. Дрібнозернистий паралелізм – це рівень команд, середньозернистий – рівень циклів, процедур і модулів (класів), велико зернистий – 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, розглядасти кількість можливих її варіантів, з яких після попереднього порівняльного аналізу (наприклад, за допомогою концепції необмеженого паралелізму) вибрati оптимальний.

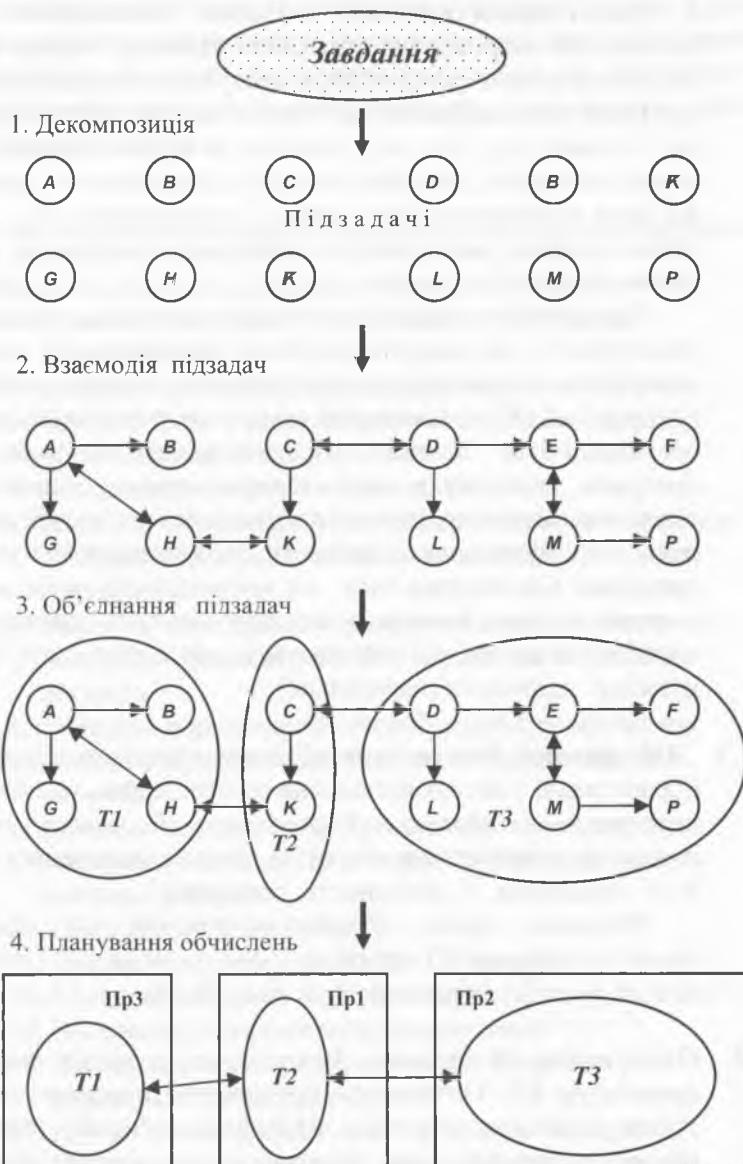


Рис. 3.6. Етапи розроблення паралельного алгоритму

**2. Проектування взаємодії підзадач.** Визначаються дії і методи, які потрібні для організації взаємодії підзадач (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 підзадача; 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 використовується. Для систем, де використовується взаємодія, що ґрунтуються 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. 3.6 – формування трьох задач  $T_1$  (підзадачі  $A, B, G, H$ ),  $T_2$  (підзадачі  $C, K$ ) і  $T_3$  (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. 3.6 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 .

---

➤ **Запитання для модульного контролю**

1. Класифікація задач з точки зору внутрішнього паралелізму?
2. Складові концепції необмеженого паралелізму?
3. Правила формування ЯПФ паралельного алгоритму?
4. Кількісні характеристики ЯПФ ?
5. Визначення коефіцієнтів  $K_e$  та  $K_i$ ?
6. Наведіть паралельний алгоритм скалярного множення векторів? Проаналізуйте його на наявність спільних ресурсів
7. Наведіть паралельний алгоритм множення вектора на матрицю. Проаналізуйте його на наявність спільних ресурсів
8. Наведіть паралельний базовий алгоритм множення матриць. Проаналізуйте його на наявність спільних ресурсів.
9. Наведіть паралельний стрічково-блоковий алгоритм множення матриць. Проаналізуйте його на наявність спільних ресурсів
10. Що пов'язує лема Брента ?
11. Призначення теореми Мунро-Петерсона?
12. Складові побудови паралельного алгоритму завдання?
13. Що виконується на етапи декомпозиції?
14. Що виконується на другому етапи розробки паралельної програми?
15. Для чого виконується етап об'єднання підзадач?
16. Що виконується на етапи планування обчислень?
17. Як здійснюється розміщення процесів по процесорам?

➤ Завдання для самостійної роботи

1. Розробити паралельний алгоритм для матричної операції

$$MA = MB * MC + MX * ME$$

2. Розробити паралельний алгоритм для векторно-матричної операції

$$A = B * MC + X * ME$$

3. Розробити паралельний алгоритм для матричної операції

$$MA = MB * (MC + MX * ME)$$

4. Розробити паралельний алгоритм для векторно-матричної операції

$$A = (B * MC) * (MM + MX * ME)$$

5. Розробити паралельний алгоритм для векторно-матричної операції  $MA = \max(Z) * (MM + MX * ME)$

6. Розробити паралельний алгоритм для векторно-матричної операції

$$x = \min(MM * d + MX * ME * a)$$

7. Побудувати паралельний алгоритм для обчислення заданого виразу, представити його через ярусно-паралельну форму, обчислити параметри ЯПФ, визначити час виконання ( $T_1, T_n$ ), коефіцієнти прискорення та ефективності ( $K_n, K_e$ )

$$a = b * c * d + (e + f / g) + k + l * (m + n)$$

8. Завдання 7 виконати для виразу

$$a = b*c + d*x + e*f + g*k + l*m + n*t$$

9. Завдання 7 виконати для виразу

$$a = (b*(c*d + e*f*g)) / (o*k + l*m)$$

10. Завдання 7 виконати для виразу

$$a = b + c*d * (e+f*g*s) + (k+l*m)*n$$

11. Завдання 7 виконати для виразу

$$a = b*(c*d + (e+f/g)) + k+l*m+n$$

12. Розробити паралельний алгоритм для виконання операції знаходження максимального елемента матриці.

13. Розробити паралельний алгоритм сортування вектора.



## РОЗДІЛ 4.

### Взаємодія процесів, що ґрунтуються на спільних змінних

---

- 4.1. Взаємодія процесів.
  - 4.2 Змінні, що поділяються
  - 4.3. Семафори.
  - 4.4 Критичні секції
  - 4.5. Монітори.
  - 4.6. Вирішення завдання взаємного виключення.
  - 4.7. Вирішення завдання синхронізації процесів.
- 

#### 4.1. Взаємодія процесів

Взаємодія процесів пов'язана з двома основними завданнями:

- комунікацією процесів;
- синхронізацією процесів.

*Комуна́кація* процесів передбачає обмін даними між процесами. При цьому інформація передається від одного процесу до іншого в будь-якому напрямку.

*Синхронізація* включає узгодження поведінки процесів і залежність виконання одного процесу від *подій*, що можуть бути в іншому процесі. Тому іноді використовується словосполучення “*синхронізація за подією*”.

Механізм реалізації взаємодії процесів залежить від виду використованої комп’ютерної системи і в загальному випадку ґрунтуються або на використанні моделі *спільних змінних* (системи зі спільною пам’яттю), або на використанні моделі *посилання повідомлень* (системи з локальною пам’яттю).

У системах зі спільною пам’яттю взаємодія процесів на фізичному (апаратному) рівні виконується за допомогою спільноПам’яті. На програмному рівні комунікація процесів і синхронізація процесів виконується через *спільні змінні*. Для передавання даних процес записує ці дані в спільні змінні, звідки їх читає інший процес. Синхронізація виконується за допомогою

відповідних спільних змінних, які змінюються процесом, у якому відбувається подія, і читаються (перевіряються) 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. Це може привести 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*.

#### 4.1.1. 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 *спільногого 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 читає дані, а другий їх змінює в цей же час, або до аварійного завершення програми.

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

Існують *два підходи* до вирішення завдання взаємного виключення. Перший підхід ґрунтуються на *контролі процесів* і пов'язаний з виявленням у процесах ділянок, у яких вони звертаються до спільногого 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 вже 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. 4.1).

Алгоритм виконання операції *ВХІДКД*:

1. Перевірити, чи знаходиться будь-який інший процес у своїй критичній 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 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 *ВИХІДКД* у мовах та бібліотеках програмування виконана у вигляді механізмів семафорів, мютексів, критичних секцій.

Другий підхід до вирішення завдання взаємного виключення передбачає безпосередній контроль спільногоресурсу (рис. 4.2). Такий контроль може бути здійснений за допомогою механізму змінних, що поділяються, або через механізм моніторів. З цією метою створюється програмний модуль (монітор), що містить в собі спільний ресурс, а також набір процедур для доступу до спільногоресурсу. Тепер доступ до спільногоресурсу 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 ж монітора приведе до блокування процесу до того часу, поки не закінчиться виконання вже розпочатої процедури монітора. Таким чином, монітор дозволяє виконання

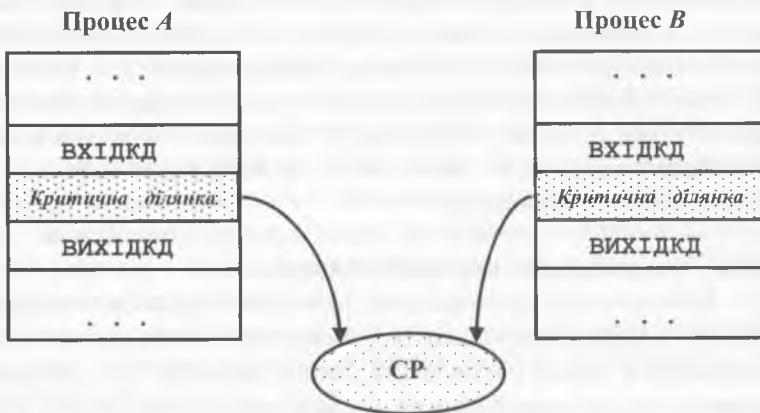


Рис. 4.1. Загальна схема вирішення завдання взаємного виключення, яка ґрунтується на контролі процесів

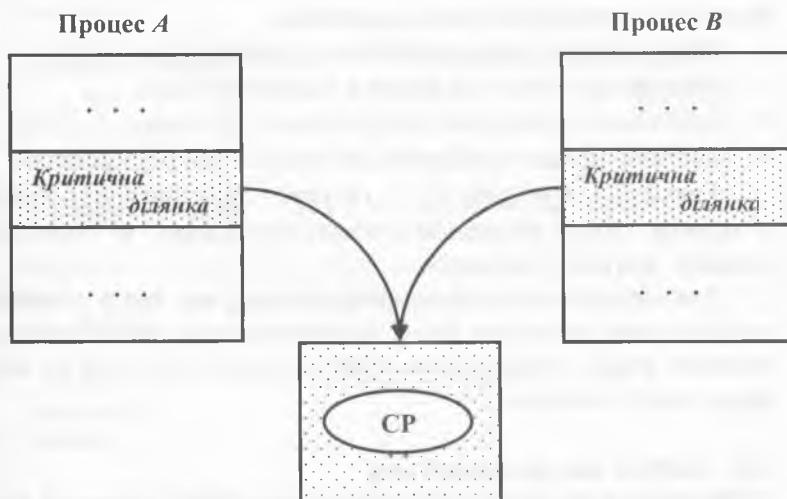


Рис. 4.2. Загальна схема вирішення завдання взаємного виключення, яка ґрунтується на контролі спільного ресурсу

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*. Механізм моніторів реалізується різними способами: у мові Ада – за допомогою захищених модулів, в мові Java – за допомогою класів з синхронізованими методами.

#### 4.1.2. Завдання синхронізації процесів

*Завдання синхронізації* двох процесів полягає в тому, що в одному процесі, наприклад *B*, у визначеній точці (*точці події*) видувається 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; він отримує сигнал від задачі *B* про завершення події і зможе продовжити своє виконання. Точка сигналу про подію і точка очікування сигналу – це *точки синхронізації*. Якщо процес *A* вийшов на точку синхронізації, коли подія вже відбулася, то він не блокується і продовжує свої виконання.

Існує декілька схем синхронізації процесів (рис. 4.3):

- 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. 4.3 – це точки синхронізації процесів; *S* означає точку посилання сигналу про подію, *W* – точку очікування сигналу про подію.

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

#### 4.2. Змінні, що поділяються

Концепція змінних, що поділяються (або *атомік*) являє собою простіший механізм, який забезпечує вирішення завдання взаємного виключення. Механізм ґрунтуються на можливості *визначення і опису* в програмі спільних ресурсів, після чого контроль цих ресурсів бере на себе система керування процесами,

яка автоматично вирішує завдання взаємного виключення стосовно цих ресурсів.

Кожне звернення до атомік -змінної розглядається як критична ділянка, вхід і вихід з якої контролюється тепер системою, яка управлює процесами. Підтримка атомік-змінних здійснюється на апаратно-системному рівні. Таки змінні не дозволяється переміщувати в кеш пам'ять процесорів, операції читання для них завжди виконуються після завершення операцій запису. Також операції з такими змінними можуть виконуватися безпосередньо в оперативній пам'яті.

Приклад програми, яка реалізує концепцію поділяємих змінних. Для опису спільних ресурсів як атомік змінних використовується конструкція Shared\_Variable.

#### ❖ Приклад 4.1

procedure Lab41

```
-- глобальна змінна (спільний ресурс)
Буфер: integer:= 1;
-- опис глобальної змінної що поділяється
Shared_Variable(Буфер);
process A
    . . .
    Буфер:= Буфер + 100;      -- критична ділянка
    . . .
end A;
process B
    . . .
    Буфер:= Буфер - 25;      -- критична ділянка
    . . .
end B;
begin
    parbegin
        A;
        B;                  -- запуск процесів А і В
    parend;
end Lab41;
```

У прикладі глобальна змінна Буфер за допомогою конструкції Shared\_Variable(Буфер) описується як змінна що поділяється. Тепер дії процесів з змінною Буфер виконуються у режимі взаємного виключення, тобто процеси не зможуть одночасно отримати доступ до змінної Буфер.

**84 Розділ 4. Взаємодія процесів, що ґрунтуються на спільних змінних**

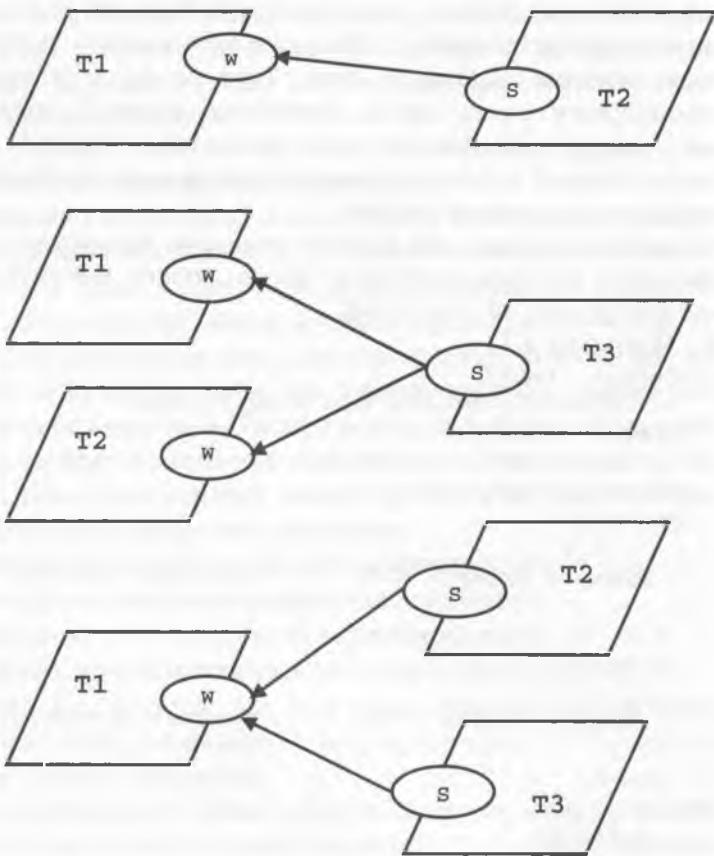


Рис. 4.3. Види синхронізації процесів

Концепцію змінних, що поділяються, реалізовано в мовах Ада, C#, Java, бібліотеці OpenMP.

### Мова Ада.

Механізм змінних що поділяються, в мові Ада реалізований за допомогою атомарних (atomic) змінних. Оголошення атомарних змінних виконується з використанням прагм Atomic і Volatile:

Прагма Atomic дозволяє описати в якості поділяємої змінної простого типу. Змінні складаного типу (масиви, записи) описуються за допомогою прагми Atomic\_Component.

Прагми Volatile і Volatile\_Component схожі на Atomic прагми. Відмінність полягає в тому, що дії з Volatile змінними виконуються безпосереднє в пам'яті, без використання реєстрів та кеш-пам'яті.

### ❖ Приклад 4.2

```
procedure Lab42 is

    -- спільні ресурси
    Ресурс1: integer:= 10;           -- простий тип
    Ресурс2: array(1..100) of float; -- складний тип

    -- опис змінних що поділяються
    Pragma Atomic(Ресурс1);
    Pragma Atomic_Component(Ресурс2);

procedure Старт_Задач is

    task A;

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

                . . .

                -- Операція з змінної Ресурс1
                Ресурс1:= Ресурс1 * 100;      -- КД
            end;
        end task;
    end task;
end;
```

## 86 Розділ 4. Взаємодія процесів, що ґрунтуються на спільних змінних

```
-- Операція з змінною Ресурс2
Ресурс2(10) := Ресурс2(50) + 100.2; -- КД

*
*
put_line("Process A finished");
end A;
-----
task B;

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

*
*
-- Операція з змінною Ресурс1
Ресурс1:= Ресурс1 + 200; -- КД

*
*
-- Операція з змінною Ресурс2
Ресурс2(50) := Ресурс2(34) * 3.14; -- КД

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

begin
    null;
end Старт_Задач;

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

    Старт_Задач; -- запуск задач А и В

end Lab42;
```

### Мова Java.

В мові Java механізм поділяємих змінних реалізований через модифікатор `volatile` [27], [55]. Для змінної, яка описана через модифікатор `volatile` вводяться особливі правила виконання операцій запис/читання. При цьому операції запис/читання виконуються безпосередньо в оперативній пам'яті без використання реєстрів та кеш-пам'яті. Запис в змінну з модифікатором `volatile` повинен закінчитися раніше, ніж почнеться виконуватися наступне читання цієї змінної.

Запис `volatile` змінної виконується в основу пам'ять, мінуючи локальну. Читання `volatile` змінної виконується також із основної пам'яті. Таким чином, значення `volatile` змінної не зберігається в реєстрах та локальній пам'яті потоку і операція читання гарантує повернення останнього значення `volatile` змінної.

#### ❖ Приклад 4.3

```
class Lab43 {  
    volatile int x = 0;  
  
    class ПотікA extends Thread {  
  
        public void run(){  
            for(int i=0; i++;i<6)  
                x = i;  
        }  
    }  
  
    class Потік2 extends Thread {  
        public void run(){  
            for(int i=0; i++;i<6)  
                System.out.println("x =" + x);  
        }  
    }  
  
    public static void main(String args[]){  
  
        ПотікA A = new ПотікA ();  
        ПотікB B = new ПотікB();  
  
        // запуск потоків  
        A.start();
```

```
B.start();  
  
    // продовження виконання головного потоку  
    System.out.println("Основний потік  
        завершено ");  
  
} // main  
  
} // Lab43
```

### **Мова C#.**

Мова C# забезпечує розробника різноманітними засобами, які реалізують концепцію атомарних змінних [21]. Вони відносяться до групи не блокованих конструкцій, які базуються на використанні класу **Interlocked** та модифікатору **volatile**. Операції з змінними в C# характеризуються високою швидкістю по відношенню до подібних операцій в інших мовах та бібліотеках.

Клас **Interlocked** забезпечує ресурси для простого і швидкого доступу до подільних через спеціальні методи:

```
Interlocked.Increment();  
Interlocked.Decrement()  
Interlocked.Add();  
Interlocked.Read();  
Interlocked.Exchange()  
Interlocked.CompareExchange();
```

Використання методів класу **Interlocked** є більш ефективним, ніж оператор **lock** через то, що реалізується неблокований доступ до об'єктів, що поділяються.

Статичні методи класу дозволяють працювати з кешем'яття у синхронізованому режимі.

#### **❖ Приклад 4.4**

```
class Lab44 {  
  
    static long Zr;      -- спільний ресурс  
  
    static void Main() {  
  
        // операції інкременту/декременту:  
        Interlocked.Increment(ref Zr);  
    }  
}
```

```

    Interlocked.Decrement(ref Zr);

    // операції додавання/зменшення:
    Interlocked.Add(ref Zr, 100);

    // читання 64-бітовго значення:
    Console.WriteLine(Interlocked.Read(ref Zr));

    // запис 64-бітовго значення з читанням
    // останнього значення:
    Console.WriteLine(Interlocked.Exchange(ref
        Zr, 10));

    // Порівняння і зміна значення:
    Interlocked.CompareExchange(ref Zr, 123, 10);

} // Lab44

```

**Volatile конструкції.** Статичні методи Thread.VolatileRead() і Thread.VolatileWrite() дозволяють працювати з кеш-пам'яттю у синхронізованому режимі. Метод VolatileRead() читає останнє значення змінної, метод VolatileWrite() записує безпосереднє в пам'ять.

Аналогічні дії можна реалізувати при опису змінної через модифікатор volatile:

```

class Доступ {
    volatile static int Q;
    ...
}

```

### Бібліотека OpenMP

Концепцію змінних, що поділяються, в бібліотеці OpenMP реалізовано за допомогою прагм omp atomic та omp reduction.

Директиви omp\_atomic розповсюджуються лише на операції наступного виду:

- X <бінарна операція> = <вираз>
- X++

- $++X$
- $X--$
- $--X$

де  $X$  - скалярна змінна,  $\langle$ вираз $\rangle$  - вираз скалярних типів, де відсутня змінна  $X$ ,  $\langle$ бінарна операція $\rangle$  - оператор виду  $+$ ,  $*$ ,  $-$ ,  $/$ ,  $&$ ,  $^$ ,  $|$ ,  $<<$ ,  $>>$ .

```
#pragma omp parallel for

for ( i = 0; i < 100; i++)
    Sum += A[i];
```

Цей варіант паралельного виконання циклу не буде коректним через то, що процеси будуть використають спільні елементи вектору  $A$ .

Використання прагми `omp_atomic` дозволить захистити спільні змінні від одночасного використання і уникнути помилок:

```
#pragma omp parallel for

for ( i = 0; i < 100; i++)
{
    #pragma omp atomic

    Sum += A[i];
}
```

Директива `omp reduction` має наступний формат:

```
omp reduction (оператор: список)
```

де можливі оператори : "+", "\*", "-", "&", "|", "^", "&&&", "||" є список - імена спільних змінних, що мають скалярний тип (`float`, `int` , `long` т. д).

Для кожної змінної створюються локальні копії в потоках, які ініціалізуються відповідно до типу оператора. Для аддитивних операцій - 0 , для мультіпликативних операцій - 1. Над ло-

кальними копіями після виконання всіх операторів в паралельної ділянці виконується заданий оператор.

```
#pragma omp parallel for reduction(+:sum)
for ( i = 0; i < 100; i++)
    Sum += A[i];
```

### 4.3. Семафори

Механізм семафорів запропонував математик Е.Дейкстра [12],[36]. У класичній інтерпретації механізм семафорів – це спеціальний захищений тип *Semaphore* та дві неподільні операції над змінною  $S$  цього типу:  $P(S)$  і  $V(S)$ . Неподільності 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, ...,  $M$ , де  $M > 1$ .

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. 4.4), так і для синхронізації процесів (рис. 4.5).

#### 4.3.1. Семафори в мові Ада

У другому стандарті мови Ада механізм семафорів реалізований у вигляді пакету *Ada.Synchronous\_Task\_Control* в додатку Annex D: Real-Time Systems [65]. Пакет реалізує механізм семафорів наступним чином. Семафорний тип забезпе-

чується 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 вказаних 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;
```

#### **4.3.2. Семафори в Win32**

Семафори в бібліотеці Win32 реалізовані за допомогою спеціального типу, а також кількох функцій. Семафор в реалізації Win32 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 Win32 виконується за допомогою типу `HANDLE`, а також набору функцій:

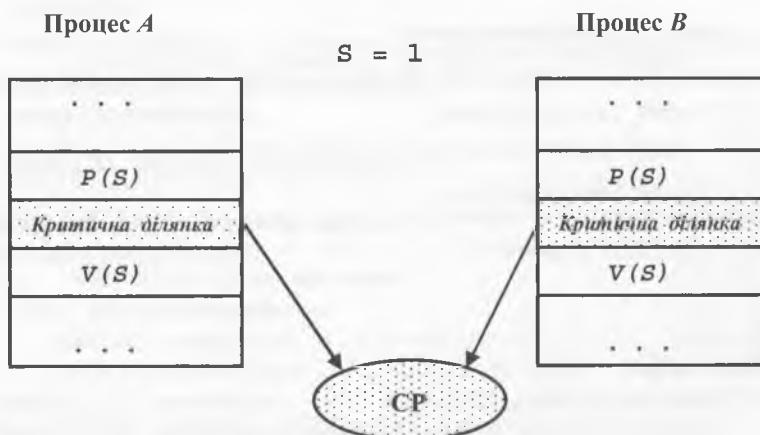


Рис. 4.4. Загальна схема вирішення завдання взаємного виключення з використанням семафорів

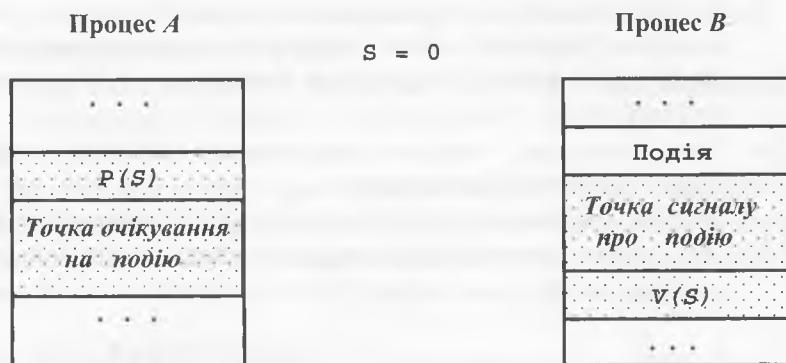


Рис. 4.5. Загальна схема вирішення завдання синхронізації з використанням семафорів

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 знаходитьться 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 створованого 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 вказане значення.

```
BOOL ReleaseSemaphore(
    HANDLE hSemaphore, // ідентифікатор об'єкта семафора
    LONG lReleaseCount, // число, що додається до
                        // поточного значення семафора
    LPVOID 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 сигнального об'єкту
    HANDLE hObjectToWaitOn,
                    // ідентифікатор об'єкта-сигналу
    DWORD dwMilliseconds,
                    // тайм-аут у мілісекундах
    BOOL bAlertable
                    // пропор 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,

```

## **98 Розділ 4. Взаємодія процесів, що ґрунтуються на спільних змінних**

```
// 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
);
```

Попереджуvalні функції :

MsgWaitForMultipleObjectsEx, SignalObjectAndWait,  
WaitForMultipleObjectsEx, WaitForSingleObjectEx  
можуть додатково виконувати попереджуvalні 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.

#### 4.3.3. Семафори і мютекси в мові C#

**Семафори.** Механізм семафорів в мові C# підтриманий за допомогою класу Semaphore, який забезпечує створення семафору та виконання належних операцій. Має чотири конструктори. Належить до простору імен System.Threading.

Семафор створюється як об'єкт класу Semaphore:

```
Semaphore S = new Semaphore(p1, p2),
```

де параметри обраного конструктора визначають: p1 - поточне (початкове) , p2 - максимальне значення семафору S.

Для семафора визначені двадцять один метод. Методи

```
S.WaitOne(),
S.Resume(),
```

аналогічні операціям P(S) і V(S) .

**Мютекси .** Мютекс створюється як об'єкт класу Mutex (існують п'ять конструкторів класу) :

```
Mutex M = new Mutex(false),
```

де параметр обраного конструктору визначає початковий стан мютексу.

Для мютексу в MSDN визначені 21 метод. Операції, аналогічні операціям P(M) і V(M) , визначено як

```
S.WaitOne(),
S.ResumeMutex(),
```

#### 4.4. Критичні секції

Вирішення завдання взаємного виключення за допомогою низькорівневих механізмів типу семафорів (мютексів) має недоліки. Вони полягають в недостатньо надійному контролю за їх застосуванням, який повністю покладається на розробника. «Огорожа» навколо критичної ділянки створюється з двох операцій, що може привести до тупикової ситуації у випадку якщо відсутня одна з операцій «огорожі» або в них використовуються різні семафори.

Ідея механізму критичних секцій базується на об'єднанні конструкцій ВХІДКД та ВИХІДКД в єдиний оператор

```
КРИТИЧНА_СЕКЦІЯ( Ім'я_КС ) do
    <Критична ділянка>
end КС;
```

Вхід в критичну секцію дозволяється лише одному процесу. При виконанні оператору *КРИТИЧНА\_СЕКЦІЯ* відбувається перевірка – вільна критична секція чи ні. Якщо вільна, процес може заходити до критичної секції і виконувати дії, що описано в тілі оператору, тобто виконувати критичну ділянку процесу. Якщо критична секція зайнята (в неї знаходитьться інший процес), то процес, що виконує оператор *КРИТИЧНА\_СЕКЦІЯ* блокується. Він буде автоматично розблокований коли критична секція звільниться (рис. 4.6).

Реалізацію механізму критичних секцій виконано в мовах Java, C#, а також у бібліотеці OpenMP.

**Мова Java.** Механізм критичних секцій у мові Java реалізований за допомогою створення синхронізованого блоку через модифікатор *synchronized*. Загальна форма синхронізованого блоку:

```
synchronized(Об'єкт Object) {
    // оператори
}
```

де Об'єкт – це посилання на об'єкт, який визначає ім'я синхронізованого блоку. Вхід до синхронізованого блоку дозволяється лише одному процесу. Намагання процесу війти в

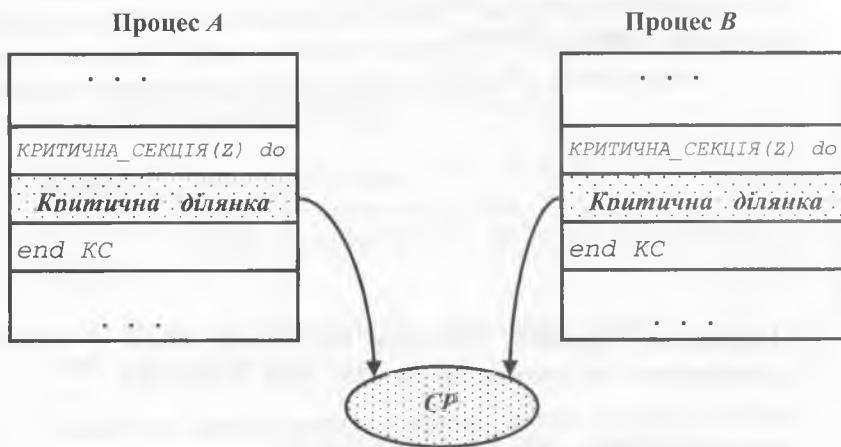


Рис. 4.6. Вирішення завдання взаємного виключення через критичні секції

синхронізований блок, де вже знаходитьться інший процес, приводить до його блокування. Синхронізований блок гарантує, що виклик з потоків методу, що є членом об'єкта Об'єкт, буде виконуватися в режимі взаємного виключення.

**Мова C#.** Механізм критичних секцій у мові C# реалізований за допомогою оператору `lock` (замок). В тілі оператору описаний блок коду, виконання якого дозволяється лише одному процесу

```
lock(Object o) {
    < <Блок коду >
}
```

Якщо процес виконує оператор `lock`, то він буде блокований у випадку, коли інший процес вже розпочав виконання блоку коду з оператору `lock` з однимаковим параметром `Object o`:

```
class Рахунок {
    int N;
    private Object Замок = new Object();
```

```

public void Вивід(int x) {
    lock (Замок) {
        if (x > N) {
            throw new Exception(" Помилка ");
        }
        N = N - x;
    }
}

```

**Бібліотека OpenMP.** Механізм критичних секцій у OpenMP реалізований за допомогою прагми `omp critical`:

```

#pragma omp critical(ім'я){
    < Блок коду >
}

```

Прагма відокремлює частину процесу < Блок коду>, доступ до якої синхронізований, і цей блок не може виконуватися процесами одночасно. Параметр *ім'я* задає ім'я критичної секції.

## 4.5. Монітори

Ідея монітора, яку запропонував Б. Хансен і розвинув Ч. Хоар, ґрунтуються на об'єднанні змінних, що описують спільний ресурс, і дій, які визначають засоби доступу до спільногого ресурсу [12], [36]. *Монітор* – програмний модуль, що містить (ховає) змінні та надає 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 викликає 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 Зи_Складу (T: out Data);

begin
    Товар:= 0.0; -- ініціалізація спільногого
                  -- 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 спільногого ресурсу).

#### 4.5.1. Монітори в мові Ада

Концепцію моніторів у стандарті мови Ада реалізовано у вигляді спеціальних програмних модулів – захищених модулів (*protected units*) [16], [51], [65]. 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 Ада використовують в першу чергу.

Спільні дані і операції над ними (захищені операції) об’єднуються в захищенному модулі, аналогічно тому, як це робиться в інших модулях мови Ада – 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;
```

Захищені 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 читаються. 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 широнеподійна 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 умовної 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 4.5 Специфікації захищеного модуля

```
-- Ада. 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 Змін_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'ект: 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.

## 106 Розділ 4. Взаємодія процесів, що ґрунтуються на спільних змінних

```
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_Захищеного_Входу;
```

Конструкція **When** Умова – це бар'єр, де Умова – логічний вираз, який визначає *відкритий* або *закритий* вхід. Переїврка 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 зчинений 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 4.6 наведено реалізацію тіла захищеного модуля Сенсор, специфікацію якого подано раніше у прикладі 4.1:

### ❖ Приклад 4.6

```
-- Ада. 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. 4.7. Захищенну функцію Замір\_Стану() і процедуру Замір\_Стану() зображенено справа, захищені входи Чекати() і Сигнал() – зліва. Захищені елементи (Стан, Прп) зображені всередині захищеного модуля в овалах.

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.

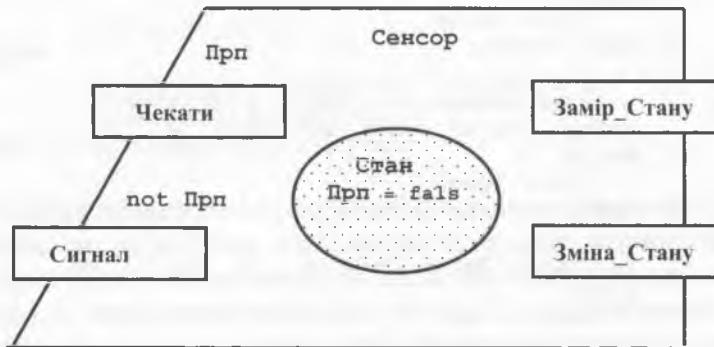


Рис. 4.7. Структура захищеного модуля

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.

#### ❖ Приклад 4.7

-- Ада. 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 захищено-го 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 використовують тільки захищені процедури, а не захищені входи.

У прикладі 4.8 захищений модуль Вклад виконує роль буфера, куди задачі Клієнт\_А і Клієнт\_В записують і звідки читують дані. Захищений модуль Вклад повинен забезпечити взаємовиключний доступ задач до спільногого ресурсу, яким є змінна Рахунок, а також синхронізацію процесів залежно від стану ресурсу.

#### ❖ Приклад 4.8

---

```
-- Ада.Захищений модуль у завданні взаємного виключення
-- та синхронізації процесів
```

---

```

protected Вклад is
    entry В_Банк(M : in    Проші);
    entry З_Банку(M : out   Гроши);
private
    Рахунок: Гроши;           -- спільний ресурс
    Прапор: Boolean := False;

end Вклад;
-----
protected body Вклад is
    entry В_Банк(M: in    Гроши)
                           when Прапор = False is
        begin
            Рахунок:= M;
            Прапор:= True;
        end В_Банк;

```

## **110 Розділ 4. Взаємодія процесів, що ґрунтуються на спільних змінних**

```
entry З_Банку(М: out Гроши)
when Прапор = True is
    М := Рахунок;
    Прапор := False;
end З_Банку;
end Вклад;

-----
task Клієнт_A;
task body Клієнт_A is
    Дохід: Гроши;
begin
    * * * виклик входу В_Банк
    Вклад.В_Банк(Дохід);

end Клієнт_A;

-----
task Клієнт_B;
task body Клієнт_B is
    Сплата: Гроши;
begin
    * * * виклик входу З_Банку
    Вклад.З_Банку(Сплата);

end Клієнт_B;
```

### **4.5.2. Монітори в мові Java**

Мова Java не має моніторів, подібних до захищених модулів у мові Ада, але вона дозволяє створювати класи, методи яких будуть мати основну властивість процедур монітора – виконуватися в режимі взаємного виключення. Такі методи в Java повинні мати модифікатор `synchronized`. Інкапсуляція спільногоресурсу 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 4.9 Клас, що виконує функції монітора**

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

    private int Буфер;           // спільний ресурс

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

Створення екземпляра класу Монітор:

```
Монітор zz = new Монітор();
```

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

**4.6. Вирішення завдання взаємного виключення**

Розглядаються приклади вирішення завдання взаємного виключення. При цьому припускається погодження про то, що змінна є спільним ресурсом незалежно від того, читається вона або змінюється. Перевага надається попередньому копіюванню спільних змінних до виконання операцій з ними в критичній ділянці, які зазвичай потребують більше часу, ніж створювання копій. Застосування реальних СМП систем показало, що відмова від попереднього копіювання спільних змінних (особливо це стосується великих масивів), які тільки читаються, призводить до хаотичної конкуренції процесів, які одночасно використовують спільний ресурс, і це пов'язано зі збільшенням часу вико-

нання програми. В цілому питання попереднього копіювання спільних даних треба вирішувати у кожному окремому випадку, виконавши попередній аналіз необхідності створювання копій або відмови від нього у випадку, якщо виконання операцій в критичній ділянці не потребує значного часу.

#### 4.6.1. Вирішення завдання взаємного виключення в мові Ада

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

**Змінні, що поділяються.** Простіший спосіб вирішення завдання взаємного виключення в мові Ада ґрунтуються на використанні неділимих змінних (атомік об'єктів). Їх описують через спеціальні прагми Atomic і volatile, які контролюють використання спільних змінних. Є чотири форми цих 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 як неподільні операції. Тобто, якщо один процес уже виконує дії з атомік-об'єктом, то інший може отримати доступ до нього лише в разі звільнення об'єкта. Отже, всі дії процесів з атомік-об'єктами виконуються лише послідовно і ніколи – одночасно.

##### ❖ Приклад 4.10

```
-- Ada.Pragma Atomic у завданні взаємного виключення --
```

```
procedure Lab410 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 Lab410;

```

**Семафори.** Задачі А і В 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 (рис. 4.4) необхідно:

- створити змінну – семафор з початковим значенням false
- розмістити операції P(S) і V(S) навколо критичної ділянки в кожному процесі

У мові Ада ці дії виконуються за допомогою типу *Suspension\_Object* і підпрограм *Suspend\_Until\_True()* та *Set\_True()*. (дів. 4.3.1).

У зв'язку з тим, що при створенні семафора (*Sem\_Ku*) він автоматично набуває значення false, потрібно змінити його на true перед запуском задач, щоб початковий вхід у критичну ділянку був відкритим для обох процесів. Явний запуск задач А і В виконується через виклик процедурі *Старт\_Задач*, де описані обидві задачі.

❖ **Приклад 4.11**

```
-- 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 Lab411 is
    Ресурс: 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

    Старт_Задач;      -- запуск задач A і B

end Lab411;
```

**Захищений модулі.** Розглянемо використання захищеного модуля для вирішення завдання синхронізації процесів, що було розглянуто в попередньому прикладі.

Структуру захищеного модуля Захист показано на рис. 4.8.

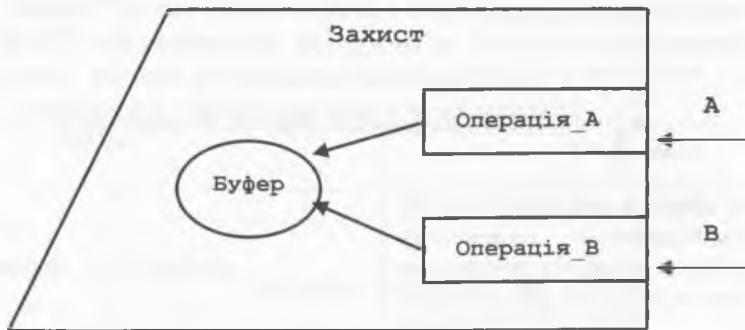


Рис. 4.8. Захищений модуль в завданні взаємного виключення

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().

#### ❖ Приклад 4.12

```
-- Ада. Захищений модуль в
-- завданні взаємного виключення

procedure Lab412 is
```

## 116 Розділ 4. Взаємодія процесів, що ґрунтуються на спільних змінних

```
-- захищений модуль
protected Захист is
    procedure Операція_A;
    procedure Операція_B;
private
    Буфер : integer:= 10;          -- спільний ресурс
end Захист;

-- тіло захищеного модуля
protected body Захист is
    procedure Операція_A is
        begin
            Буфер := Буфер + 2;
        end Операція_A;

    procedure Операція_B is
        begin
            Буфер := Буфер - 25;
        end Операція_B;

    end Захист;

-- використання захищеного модуля в задачах
task A;

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

task body B is
begin
    *
    * *
    -- дії над спільним ресурсом
    Захист.Операція_B;
    *
    *
end B;
-- головна процедура
begin
    null;
end Lab412;
```

#### 4.6.2. Вирішення завдання взаємного виключення в Win32

Бібліотека Win32 містить декілька засобів, які можна використати для вирішення завдання взаємного виключення. Це реалізація механізму семафорів, мютексі та критичні секції (табл. 4.1). У наступних підрозділах розглядаються приклади вирішення завдання взаємного виключення для трьох задач T1, T2 і T3 за допомогою засобів Win32. 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 процесів за допомогою “створювання огорожі” з належних операцій Win32, які виконують функції примітивів ВХІДКД і ВИХІДКД.

**Увага!** Під час використання в наступних прикладах функцій Win32 для параметрів, які прямо не стосуються керування процесами, умовно встановлено значення NULL.

Таблиця 4.1. Засоби взаємного виключення Win32

Об'єкт		Опис
Семафор	Semaphore	Містить лічильник від нуля до визначеного значення, які визначають кількість потоків, що мають доступ до спільног ресурсу
Мю текст	Mutex	Цим об'єктом може володіти тільки один процес у поточний момент часу, що дозволяє цьому процесу ексклюзивний доступ до спільног 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 4.13

```
-- | Ада. Використання семафорів бібліотеки Win32
-- | в завданні взаємного виключення

With Win32, Win32.winbase, Win32.winnt;
Use Win32, Win32.winbase, Win32.winnt;
procedure Lab413 is

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

    -- HANDLE змінна для створювання семафора
    Sem1: HANDLE;

    -- допоміжні змінні

    Temp: DWORD;
    pl: 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;
            pl := 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");
        end T2;
    end Запуск_Задач;
```

```

-- Критична ділянка
Temp := WaitForSingleObject(Sem1, Infinite);
Pecupsr := Pecupsr + 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);
    Pecupsr := Pecupsr + 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 Lab413;

```

**Мютекси.** Мютекс (від *mutual exclusion* – взаємне виключення) – засіб, схожий на семафор, але на відміну від семафора він має не значення, а стан (*вільний, зайнятий*) [45]. Мютекс може належити тільки одному процесу і може передаватися від одного процесу до іншого (бути захопленим процесом). Вхід в критичною ділянку дозволяється тільки тому процесу, який тепер 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 4.14**

```
-- | Ада. Використання мютексів у бібліотеці Win32          --
-- | в завданні взаємного виключення                           --
-----
procedure Lab414 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");
        end T2;
    end Запуск_Задач;
```

```

    . . .
    -- 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 Lab414;

```

**Критичні секції.** Механізм критичних секцій в Win32 реалізований за допомогою типу CRITICAL\_SECTION і спеціальних операцій, що дозволяють створити критичну секцію, а також контролювати входження і виходження процесу в критичну секцію. На жаль, реалізація механізму критичних секцій у Win32 виконано подібно до механізмів семафорів і мютексів у вигляді двох примітивів замість одного. Тобто цей механізм є різновидом механізму семафорів з тією відмінністю, що замість сема-

фора в ньому використовується ім'я критичної ділянки, вхід у яку контролюється системою і дозволяється тільки одному процесу.

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

На 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 EnterCriticalSection() не встановлюється, тобто він завжди є INFINITE.

### ❖ Приклад 4.15

```
-- |Ада. Використання критичних секцій бібліотеки Win32
-- |у задачі взаємного виключення
```

```
procedure Lab415 is

    Ресурс: integer;      -- спільний ресурс
                           -- (глобальна змінна)
    -- створювання критичної секції
    CrSec: 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 Lab415;
```

### 4.6.3. Вирішення завдання взаємного виключення в мові Java

Вирішення завдання взаємного виключення в Java програмах виконується за допомогою *синхронізованих методів* і блоків.

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

#### ❖ Приклад 4.16

```
/*
-- 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) В викликав і виконує, наприклад, з екземпляра `Блок` класу `Монітор` метод `Блок.Читати()`, то інший потік А буде блокуваний, 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 В.

**Синхронізовані блоки.** Приклад 4.17 демонструє використання синхронізованого блоку для вирішення завдання взаємного виключення.

#### ❖ Приклад 4.17

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;

        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
```

#### 4.6.4 Вирішення завдання взаємного виключення в мові C#

В мові C# для вирішення завдання взаємного виключення передбачено використання атомарних змінних, семафорів, мутексів, критичних секцій.

❖ **Приклад 4.18.** Застосування мутексів мови C#:

```
class Lab418 {

    static int Buffer;      // спільний ресурс

    static Mutex MTK = new Mutex(false); // мутекс

    static void Main() {

        // створення потоків
        Thread PT1 = new Thread(new ThreadStart(T1));

        Thread PT2 = new Thread(new ThreadStart(T2));

        // Запуск потоків
        PT1.Start();

        PT2.Start();

    } // Main

    // потік 1 -----
    static void T1() {

        . . .

        MTK.WaitOne();
        Buffer = Buffer - 110; //критична ділянка
        MTK.ReleaseMutex();

        . . .

    } // T1

    // потік 2-----
    static void T2() {
```

```

    MTK.WaitOne();
    Buffer = Buffer + 20; //критична ділянка
    MTK.ReleaseMutex();

}

}

// T2

}// Lab418

```

#### 4.6.5 Вирішення завдання взаємного виключення в OpenMP

В бібліотеці OpenMP для вирішення завдання взаємного виключення передбачено прагми `omp atomic` і `omp critical`. Вони відповідно реалізують концепції атомік – змінних та критичних секцій.

Прагма `omp atomic` має наступну форму

```

#pragma omp atomic{
    < Блок коду>
}

```

Забезпечує синхронізоване виконання частини програми, що описано у розділі < Блок коду>.

```

int buf
int x
#pragma omp parallel sections
                    shared(buf) private(x) {
    section{
        #pragma omp atomic{
            buf = buf + x;
        }
    section{
        #pragma omp atomic{
            buf = buf + x;
        }
    }
}

```

**Критична секція.** Наступний приклад ілюструє організацію взаємовиключного доступу до виконання оператору `buf = buf + x`. Змінна `buf` описана через прагму `shared` і є спільним ресурсом. Синхронізований доступ до змінної `buf` відбувається в критичної секції, позначеної як `cs1`:

```
int buf
int x
#pragma parallel sections
    shared(buf) private(x) {

        section{
            #pragma omp critical(cs1){
                buf = buf + x;
            }
        }

        section{
            #pragma omp critical(cs1){
                buf = buf + x;
            }
        }
    }
}
```

## 4.7. Вирішення завдання синхронізації

### 4.7.1. Вирішення завдання синхронізації в мові Ада

Для вирішення завдання синхронізації в мові Ада можна застосувати механізм семафорів, а також захищенні модулі.

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

### ❖ Приклад 4.19

```
-- Ада. Синхронізація задач
-----
procedure Синхронізація is

    X : integer; -- глобальна змінна

    СигналПроПодію: Suspension_Object; -- семафор

    -- задача, що чекає на подію
    task A;
    task body A is
        begin
            . . .
            -- точка очікування події
            Suspend Until True(СигналПроПодію);
        end A;

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

                Set_True(СигналПроПодію); -- 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: вход Чекати\_Введення() і процедура СигналПроВведення(), а також допоміжна захищена змінна (прапор) F1, яка буде використову-

ватись у бар'єрі захищеного входу. Структуру захищеного модуля Контроль подано на рис. 4.9.

Очікування на подію (введення X) у задачі А виконано як виклик входу Чекати\_Уведення(), а посилання сигналу про подію в задачі В – 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.

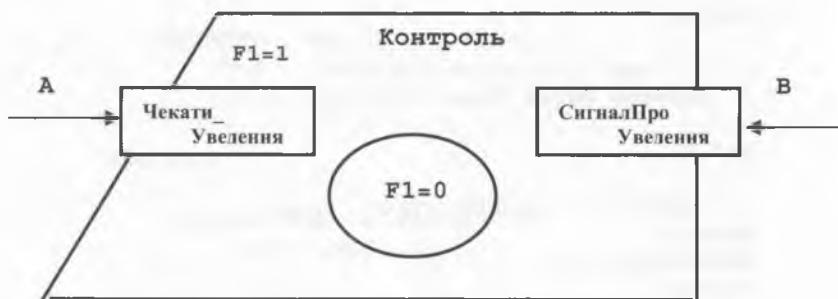


Рис. 4.9. Захищений модуль в завданні синхронізації

Таким чином, початкове значення прапора  $F1=0$  дозволяє блокувати процес А в операції виклику входу Чекати\_Уведення() до того часу, поки процес В не змінить значення бар'єра за допомогою операції СигналПроУведення() після того, як подія буде мати місце.

#### ❖ Приклад 4.20

-- Ада.Захищений модуль ии

-- захищений модуль

```

protected Контроль is
    entry Чекати_Уведення;
    procedure СигналПроУведення;
private
    F1 : integer:= 0;
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
        F1 := 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 А з кількома задачами (В, С, D), де виконується введення, необхідна модифікація захищеного модуля Контроль, яка пов'язана з тим, що бар'єр входу Чекати\_Уведення буде мати вигляд when F1 = 3, а процедура СигналПроУведення буде змінювати пропор F1:= F1 + 1. Тепер задача А під час виклику входу Чекати\_Уведення продовжить своє виконання тільки після того, як кожна задача

В, С і D викличе процедуру СигналПроУведення і прапор F1 набуде значення 3.

#### 4.7.2 Вирішення завдання синхронізації в Win32.

Завдання синхронізації в Win32 можна вирішити за допомогою семафорів, подій або таймерів. При цьому використовуються функції очікування (табл. 4.2). Докладний опис функцій бібліотеки Win32 наведено в дод. А.

Таблиця 4.2. Засоби синхронізації у Win32

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 `WaitForSingleObject()`, а сигнал про подію, що відбулася, – через функцію `ReleaseSemaphore()`.

У прикладі 4.21 задача Т1 інформує задачу Т2 про подію, що відбулася в задачі Т1 (Подія1), а задача Т3 чекає на подію, що відбудеться в задачі Т2 (Подія2).

#### ❖ Приклад 4.21

---

-- | Ада. Використання семафорів бібліотеки Win32  
-- | в завданні синхронізації

---

```
procedure Lab421 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 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);
                        *
                        *
                    end T3;
                end Запуск_Задач;
```

```

        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 Lab421;

```

У прикладі 4.22 задача Т3 чекає на події, що відбудеться в задачі Т1 (Подія1) і задачі Т2 (Подія2).

### ❖ Приклад 4.22

---

```
-- |Ада. Використання семафорів бібліотеки Win32
-- |у завданні синхронізації
```

---

```

procedure Lab422 is
    -- 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 Т3 про подію в задачі Т1
            p1:= ReleaseSemaphore(Sem1_3,1,NULL);
        end;
    end;

```

```

        put_line("Process T1 finished");
end T1;
-----
task body T2 is
begin
    put_line("Process T2 started");

    -- Подія2
    -- Сигнал задачі T3 про подію 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 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 Lab422;

```

У прикладі 4.23 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 4.23**

```
-- | Ада. Використання семафорів бібліотеки Win32
-- | в завданні синхронізації
-----
procedure Lab423 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 Т1
    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 Lab423;

```

**Застосування подій.** Механізм подій призначений виключно для синхронізації процесів.

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

```

HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES    адреса_атрибутів_захисту
    BOOL        пралор_ручної_установки_події
    BOOL        пралор_початкового_стану
    LPCTSTR    адреса_об'єкта_події
);

```

У разі використання механізму подій очікування подій 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 очікуванням потокам;
- `ResetEvent()` – виконує скидання події (установлює стан події в несигнальний).

У прикладах 4.24 – 4.26 наведено вирішення за допомогою механізму повідомлень завдання синхронізації, які розглядалися в прикладах 4.21 – 4.23.

У прикладі 4.24 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 4.24

---

```
-- |Ада. Використання подій бібліотеки Win32
-- |в завданні синхронізації
```

---

```
procedure Lab424 is

    -- HANDLE змінна для створювання подій
    Evn12, Evn23: 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 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 (Подія1) 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 Lab424;

```

У прикладі 4.25 задача Т3 чекає на події, що відбудуться в задачі Т1 (Подія1) і задачі Т2 (Подія2). Для сигналізації про ці події застосовані події Evn1\_3 та Evn2\_3.

#### ❖ Приклад 4.25

```

-- |Ада. Використання подій бібліотеки Win32
-- |в завданні синхронізації
-----
procedure Lab425 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 Т3 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 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 Lab425;
```

## **142 Розділ 4. Взаємодія процесів, що ґрунтуються на спільних змінних**

У прикладі 4.26 задача T1 інформує задачі T2 і T3 про подію, що відбулася в задачі T1. Це можливо виконати через одну подію Evn1\_23.

### **❖ Приклад 4.26**

```
-- |Ада. Використання подій бібліотеки Win32
-- |в завданні синхронізації
-----
procedure Lab426 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");
                .
                .
            end T3;
    end;
-----
```

```

-- 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
Evn1_23:= CreateEvent(NULL, 1, 0, NULL);

-- виклик процедури для запуску задач
Запуск_Задач;

end Lab426;

```

#### 4.7.3. Вирішення завдання синхронізації в мові Java

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()` роз-

блокує 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 відбулася 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 4.27

Клас `Synchro` 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 Synchro {

    private int Прапор;      // додаткова 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
    }
} //Synchro
```

```

class Потік1 extends Thread {
    Synchro O1;           // Посилання на клас Synchro
    // конструктор
    Потік1(Synchro 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
        Synchro Синхр = new Synchro();
        // екземпляри потоків
    }
}

```

```

Потік1 П1 = new Потік1(Снхр);
Потік2 П2 = new Потік2(Снхр);

// старт потоків
П1.start();
П2.start();

}// main

}// Синхронізація потоків

```

#### 4.7.4. Вирішення завдання синхронізації в мові C#

Синхронізація процесів в мові C# виконується за допомогою механізмів семафорів і подій. Множні семафори дозволяють реалізувати різні схеми взаємодії потоків: один потік чекає на сигнал від одного потоку, кілька потоків чекають на сигнал від одного (дів. 4.3.4).

Механізм подій реалізований за допомогою набору класів EventWaitHandle, AutoResetEvent, ManualResetEvent, що є наслідками абстрактного класу WaitHandle. Два останніх класи призначено для реалізації автоматичного або ручного управління станом об'єкта-події під час його перевірки. Використання класу ManualResetEvent дозволяє процесу не скидати подію в несигнальний стан при отриманні сигналу, як це робиться зазвичай. Це дозволяє ефективно реалізувати схему синхронізації, коли процес через один об'єкт – подію може сигналізувати кільком процесам про подію, що відбулася.

Сигнал про подію передається за допомогою методу Set(), очікування на сигнал здійснюється через метод WaitOne(). Метод Reset() дозволяє перевести об'єкт-подію в несигнальний стан.

Під час виклику методу Set() він переводить подію в сигнальний стан доти, доки процес, що чекає на подію, не викличе метод WaitOne(), який змінює стан події на несигнальний (використається автоматичне скидання), або залишає його в сигнальному стані (використається ручне скидання).

❖ **Приклад 4.28** В прикладі наведено використання механізму подій мови C# для синхронізації двох потоків:

```

class Lab428 {

    // створення об'єкту - події
    static EventWaitHandle СНГ = new
        AutoResetEvent(false);

    static void Main() {
        new Thread(СигнПотік).Start();
        new Thread(ЧекПотік).Start();
    }

    } // Main

    static void ЧекПотік() {
        . . .
        СНГ.WaitOne(); // очікування на сигнал
    }

    static void СигнПотік() {
        . . .
        СНГ.Set(); // посилання сигналу
    }
}

} // Lab428

```

#### 4.7.5. Вирішення завдання синхронізації в OpenMP і MPI

**Бібліотека OpenMP.** Синхронізація процесів в бібліотеці OpenMP здійснюється неявно при виході програми з паралельної ділянки - потік майстер очікує на завершення всіх потоків паралельної ділянки і тільки потім продовжує своє виконання.

Явна синхронізація процесів в бібліотеці OpenMP здійснюється через прагму `omp barrier`. Прагма блокує кожен процес в паралельної ділянці поки всі процеси її не досягнуть:

```

#pragma omp parallel {
    < Дія 1 >
#pragma omp barrier

```

```
< Дія 2 >
}
```

В прикладі кожен процес паралельно виконує блок коду

< Дія 1 >, блокується по досягненню прагми `omp barrier` і розпочинає виконання блоку коду < Дія 2 > тільки по завершенню всіма процесами блоку < Дія 1 >.

Синхронізація процесів також здійснюється при паралельному виконанні циклів. В наступному прикладі виконання блоку коду < Дія 2 >, який розміщений після прагми `omp for`, розпочинається тільки по завершенню всіма процесами паралельного виконання циклу блоку < Дія 1 >. Тобто в прагмі `omp for` неявно присутня і прагма `omp barrier`.

```
#pragma omp parallel {
    #pragma omp for{
        < Дія 1 >
    }
    < Дія 2 >
}
```

Неявну синхронізацію для прагми `omp for` можна скасувати за допомогою параметру `nowait` :

```
#pragma omp parallel {
    #pragma omp for nowait{
        < Дія 1 >
    }
    < Дія 2 >
}
```

В цьому випадку процес, що закінчив паралельне виконання своєї частини циклу, не очікує на закінчення всіма процесами циклу і негайно розпочинає виконання блока коду < Дія 2 >.

**Бібліотеках MPI.** Явна синхронізація процесів в бібліотеці MPI здійснюється через функцію `mpi barrier()` :

```
int MPI_Barrier(MPI_COMM KM),
```

де параметр `KM` задає ім'я комунікатора.

Функція блокує процес, що її викликає, до того як всі процеси з комунікатору  $KM$  не здійснять виклик функції  $\text{mp}i$   $\text{barrier}()$  з цим же параметром  $KM$ . Функція  $\text{MPI\_Barrier}()$  гарантує, що код процесу, що розміщений після виклику цієї функції ( $< \text{Action2} >$ ), всі процеси вказаного комунікатору розпочнуть одночасно:

```
int Rank, N;

MPI_Comm_rank(MPI_COMM_WORLD, &Rank);
MPI_Comm_size(MPI_COMM_WORLD, &N);

for(int j=0; j< N; ++j) {
    if(Rank == j) {
        < Action 1 >
    }

    MPI_Barrier(MPI_COMM_WORLD);
    < Action 2 >
}
```

#### ➤ Запитання для модульного контролю

1. Навидить види взаємодії процесів ?
2. Що включає постановка та загальна схема вирішення завдання взаємного виключення ?
3. Наведіть алгоритми примітивів ВХІДКД і ВИХІДКД ?
4. У чому полягає головна відмінність двох підходів до рішення завдання взаємного виключення ?
5. Яке призначення мають багатозначні семафори ?
6. Як застосують семафори в завданні взаємного виключення і  
в завданні синхронізації процесів ?
7. Які процедури в Ада семафорах виконують роль операції  $P()$  і  $V()$  ?
8. Яке призначення типу HANDLE в бібліотеці Win32 ?
9. Як реалізовано механізм критичних секцій в Win32 ?
10. В чому полягає різниця між семафорами та мютексами в Win32 ?
11. Яку структуру має захищений модуль ? Види та призна-

- чення захищених операцій ?
12. Яку роль виконує бар'єр у захищеному модулі ?
  13. Як виконується синхронізація за допомогою захищеного модуля ?
  14. Яка особливість використання захищених функцій ?
  15. Які черги формуються при використанні захищеного модуля ?
  16. В чому полягає відмінність в реалізації концепції монітору в мовах Java і Ада ?
  17. Як створюється потік у мові Java?
  18. Засоби мови Ада для створення множини задач?
  19. Поняття потокової функції? Де використається ця функція?
  20. Що спільногого в концепції створення процесу, що використовуються в бібліотеках OpenMP та MPI ?

➤ **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) синхронізовані методи,
  - б) синхронізовані блоки.
2. Написати програму на мові Ада, 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 Win32.
3. Вирішити за допомогою семафорів Win32 завдання синхронізації одного потоку з кількома іншими (потік чекає на події в трьох інших потоках).
4. Вирішити за допомогою засобів Win32 завдання синхронізації кількох потоків з одним (четири потоки чекають на подію в одному потоці).
5. Завдання 3 реалізувати за допомогою захищеного модулю мови Ада.

6. Завдання 4 реалізувати за допомогою захищеного модуля мови Ада .
7. Виконати моделювання механізму багатозначних семафорів за допомогою захищеного модуля мови Ада.
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 завдання взаємного виключення вирішується за допомогою різних засобів синхронізації. Визначити також час виконання цієї програми у випадку, коли задача взаємного виключення взагалі не вирішується.



## РОЗДІЛ 5.

### Взаємодія процесів, що ґрунтуються на посиланні повідомлень

- 5.1. Загальна схема.
- 5.2. Мова Оккам.
- 5.3. Ада. Механізм рандеву.
- 5.4. Бібліотека PVM.
- 5.5. Бібліотека MPI.

Цей розділ присвячено питанням організації обчислень в комп’ютерних системах з розподіленою пам’яттю на підставі посилання повідомлень. Розглянуто реалізації механізму повідомлень в мовах Оккам і Ада, бібліотеках PVM і MPI.

#### 5.1. Загальна схема

Загальна схема передачі повідомлень між процесами ґрунтуються на використанні операцій Передати() і Прийняти()[38], [41], [44]. За допомогою операції Передати() процес відправляє дані іншому процесу, який отримує ці дані за допомогою операції Прийняти():

<u>Процес1</u>	<u>Процес2</u>
• • •	• • •
<b>Передати</b> (Дані) ;	<b>Прийняти</b> (Дані) ;

Існують різноманітні схеми операцій **Послати()** і **Прийняти()**: іменовані, синхронні, асинхронні, з блокуванням або без блокування та інше.

Іменована схема потребує застосування імен взаємодіючих процесів в операціях **Передати()** і **Прийняти()**:

**Передати**(Ім’я\_Процесу\_Одержанувача, Дані)  
**Прийняти**(Ім’я\_Процесу\_Відправника, Дані)

У разі застосування непрямої схеми взаємодії процесів із використанням поштової скриньки, буфера, каналу, лінка або труби (*pipe*) треба вказувати ім'я засобу, що застосовується:

**Передати** (Буфер, Дані)  
**Прийняти** (Буфер, Дані)

Схема, у якій вказані імена обох процесів – відправника і одержувача, має назву *симетричної*. Асиметрична схема потребує вказати ім'я лише одного процесу, до якого звертаються. Асиметрична схема вигідна в моделі *клієнт–сервер*, коли процес–сервер не знає і не повинен знати, з яким процесом–клієнтом він буде взаємодіяти.

Існує кілька варіантів непрямої взаємодії процесів:

- один процес з одним процесом;
- один процес з кількома процесами;
- кілька процесів з одним процесом;
- кілька процесів з кількома процесами;

Передача повідомлень може бути також синхронною або асинхронною. Синхронна схема ґрунтуються на тому, що процес, який виконує операцію *Передати()* або *Прийняти()*, блокується, поки другий процес не вийде на приймання повідомлення (операція *Прийняти()*) або на передавання повідомлення (операція *Передати()*).

Оптимальною з погляду розробника є можливість використання різних видів операцій передавання–приймання повідомлень, як це зроблено в бібліотеці MPI, яка містить набір з кількох видів операцій *Передати()* і *Прийняти()* [2].

## 5.2. 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 багатопроцесорних системах [39]. Трансп’ютер має процесор, локальну пам’ять та засоби для зв’язку з іншими трансп’ютерами – вісім двонаправлених *лінків* (*link*). За допомогою лінків можлива побудова трансп’ютерних

систем різноманітної структури з розподіленою пам'яттю. Оккам підтримує комунікацію і синхронізацію процесів через посилення повідомлень. Оккам реалізує варіант механізму рандеву, який на відміну від рандеву в мові Ада є симетричним.

У мові Okcam процеси не іменовані, тому їх взаємодія ґрунтуються 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. 5.1) 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.

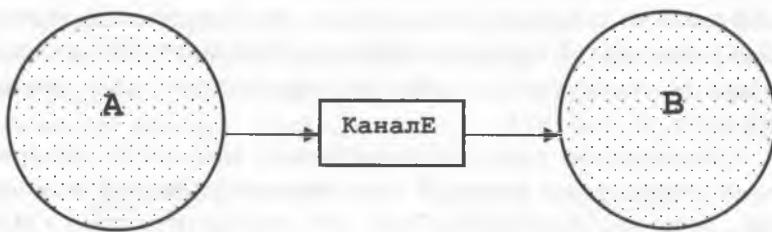


Рис. 5.1. Схема взаємодії процесів в Okcam

**CHAN OF INT** КаналЕ: -- оголошення каналу

**PAR** -- перший процес  
**INT** x = 100:

КаналЕ ! x -- запис у канал

```

PAR      -- другий процес
INT у:
КаналE ? у      -- читання з каналу

```

Канал у мові Оккам визначається за допомогою типу CHAN. Для каналу необхідно визначити тип даних, які він буде пересилати. Канал КаналE може бути застосований для передачі змінних типу INT. Для більш складних типів даних канал потребує використання поняття *протоколу каналу*.

- ❖ **Приклад 5.1.** Протокол для передавання масивів різної довжини.

```

PROTOCOL Буфер IS INT::[] BYTE:
CHAN OF Буфер КаналX:
[8] BYTE Дані:
INT Розмір:
PAR
КаналX ! 5::"HELLO"
КаналX ? 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 КаналX. В прикладі 5.1 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 з змінну Розмір.

- ❖ **Приклад 5.2.** 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 АТ адрес

Застосування операторів конфігурування наведено в прикладі 5.3. 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. 5.2.

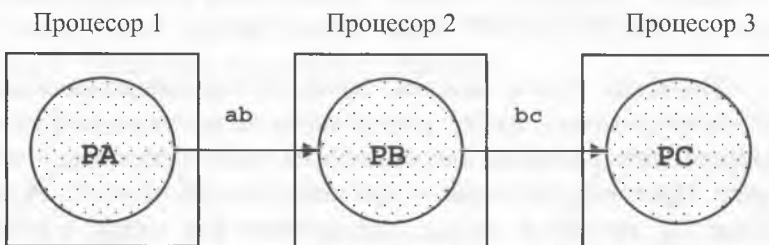


Рис. 5.2. Структура взаємодії процесів в Оккам

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 5.3

---

-- 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)
```

### 5.3. Ада. Механізм рандеву

Механізм рандеву визначає реалізацію взаємодії задач у мові Ада через посилання повідомлень. Оператори входу entry, прийняття виклику входу accept, оператор відбору select дозволяють ефективно реалізувати різноманітні можливості моделі рандеву.

Приклад використання механізму рандеву для організації взаємодії задач А і В, де задача А передає в задачу В ціле число  $x$ . Структурну схему взаємодії задач показано на рис. 5.3. Для взаємодії задач у специфікації задачі В описаний вхід Data, специфікація якого дозволяє приймати дані в задачі В. Посилання цілого числа зі задачі А виконано як виклик входу Data з підстановкою фактичного параметра.

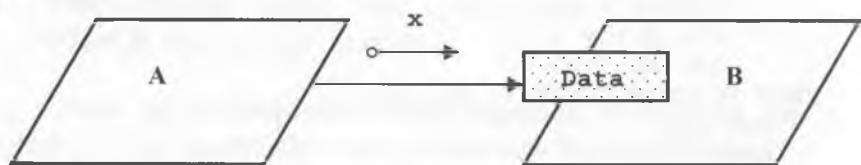


Рис. 5.3. Схема взаємодії задач (одностороння)

#### ❖ Приклад 5.4

-- Ада. Механізм рандеву

```

procedure Lab54 is

    task A;
    task B is
        entry Data(e: in integer);           -- вхід
    end B;

    task body A is
        x: integer := 25;                  -- локальна змінна
    begin
        . . .
        B.Data(x);                      -- виклик входу Data задачі В
        . . .
    end A;

    task body B is
        c: integer;                      -- локальна змінна
    begin
        . . .
        -- оператор приймання виклику входу
        accept Data(e: in integer) do
            c:= e;                      -- тіло оператора приймання
        end Data;
    end;

```

```

end B;

begin          -- основна програма
null;          -- порожній оператор

end Lab54;

```

Структурну схему взаємодії задач, під час якої задачі здійснюють обмін даними, зображенено на рис.5.4. При цьому застосовані два входи: Data для передавання даних і Result для повернення результату. Для запуску задач їх вкладено в процедуру Run\_Tasks, виклик якої приведе до старту задач A і B.

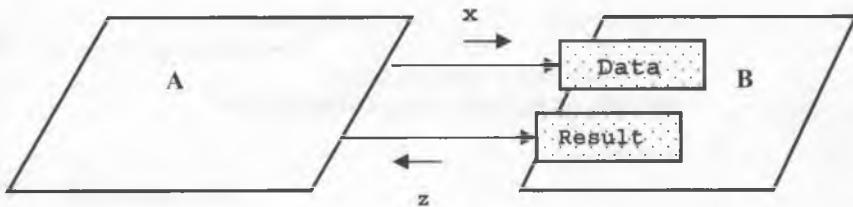


Рис. 5.4. Схема взаємодії задач (двостороння)

### ❖ Приклад 5.5.

---

-- Ада.Механізм рандеву

---

procedure Lab55 is

```

temp: integer;
-- 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
    temp := 124;
    Run_Tasks;      -- запуск задач
end Lab55;

```

Структурну схему взаємодії трьох задач – Т1, Т2 і Т3 зображенено на рис. 5.5. Задача Т3 приймає дані від задач Т1 і Т2 за допомогою входів: DataT1 і DataT2. Особливістю реалізації взаємодії задач є використання в тілі задачі Т3 оператора select. Це дозволяє задачі Т3 приймати виклик входів DataT1 та DataT2 в будь-якій послідовності в міру їх надходження.

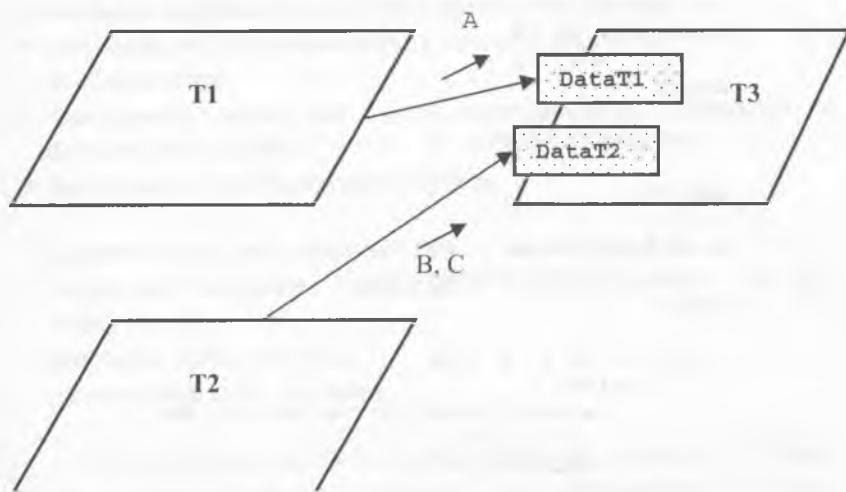


Рис. 5.5. Схема взаємодії задач

#### ❖ Приклад 5.6.

-- Ада. Механізм рандеву

procedure Lab56 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
            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 программа
    Run_Tasks;      -- явний запуск задач T1, T2, T3
end Lab56;

```

#### 5.4. Бібліотека PVM.

Бібліотека (інтерфейс) PVM (Parallel Virtual Machine) визначає стандарт для програмування в комп'ютерних системах з розподіленою пам'яттю [54]. 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;
- *PvmDataInPlace* – 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);
```

Перемикає 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_pkcp1x(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 розміщує-

ється 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 повідомень.** 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_upkcp1x(float *cp, int n, int s);
```

Перелік функцій PVM для пакування даних у повідомленні, яке надсилається, і розпакування повідомень, 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 5.7.

---

```
/*
-- 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).

### 5.5. Бібліотека MPI.

Бібліотека (інтерфейс) MPI (Message Passing Interface) визначає стандарт для програмування в комп’ютерних системах з розподіленою пам’яттю [2], [15]. 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. Одержанувач 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 копірується 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) буферізоване передавання; завершується одразу же після копіювання повідомлення в системний буфер;
- 4) передавання по 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 посилення 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) повідомень 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()`.

**Колективний обмін повідомленнями.** 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 - комунікатор;
- 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).

Колективний 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. 5.1);
- `root` - ранг одержувача в групі `cm`;
- `cm` - комунікатор.

Таблиця 5.1. Деякі функції бібліотеки MPI

Функція	Операція
<code>MPI_Send()</code>	Переслати повідомлення. Блокуватися, поки воно не буде отримано
<code>MPI_Recv()</code>	Отримати повідомлення. Блокуватися, поки воно не буде надіслано
<code>MPI_ISend()</code>	Переслати повідомлення без блокування
<code>MPI_IRecv()</code>	Отримати повідомлення без блокування
<code>MPI_Probe()</code>	З'ясувати, чи отримано повідомлення (блокуватися до отримання повідомлення)
<code>MPI_IProbe()</code>	З'ясувати, чи отримано повідомлення (без блокування)
<code>MPI_Bcast()</code>	Надіслати повідомлення всім процесам в групі
<code>MPI_Reduce()</code>	Отримати дані від усіх процесів у групі
<code>MPI_Gather()</code>	Зібрати дані від всіх процесів
<code>MPI_Scatter()</code>	Розсилати дані усім процесам у групі. Дані - одного розміру.
<code>MPI_Scatterv()</code>	Розсилати дані всім процесам у групі. Дані можуть бути різного розміру.
<code>MPI_Alltoall()</code>	Розсилати дані всім процесам в групі зі всіх процесів групи. Дані одного розміру.
<code>MPI_Alltoallv()</code>	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. 5.2.

Таблиця 5.2. 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 (див. табл. 4.1);
- cm - комунікатор.

Результат формується в буфері в усіх процесорах групи cm.

Синхронізацію процесів в групі можно виконати за допомогою функції MPI\_Barrier() :

```
int MPI_Barrier(MPI_Comm cm);
```

де cm - комунікатор.

Ця функція блокує процес, що її викликає, доти, доки всі процеси групи з ідентифікатором `cm` не викличуть цю функцію.

#### ❖ Приклад 5.8.

Створення процесів, кожний з яких передає головному процесу (ранк 0) свій ранк, який виводиться на дисплей. Цей приклад - розширення прикладу 2.6.

```
#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 );

        printf("%s\n", Повідомлення);

    }/* for */
}

/* Завершення роботи MPI */
MPI_Finalize();

return 0;
}/* main */

```

#### ❖ Приклад 5.9

Приклад ілюструє застосування колективної операції MPI\_Bcast для розсилання даних зі задачі 6 до всіх інших задач з комунікатором MPI\_COMM\_WORLD

```

#include <stdio.h>
#include <string.h>
#include <mpi.h>

int main(int args, char *argv[]){
    int data = 548;
    int мій_ранг;

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

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

    if (мій_ранг == 6) {
        /* Процес 6
           Відправити повідомлення всім процесам */

        MPI_Bcast(&data, 1, MPI_Init, 6, MPI_COMM_WORLD);
    }
    else{
        /* всі процеси крім процесу 6
           Отримати повідомлення */

        MPI_Bcast(&data, 1, MPI_Init, 6, MPI_COMM_WORLD);
    }

    /* Завершення роботи MPI */
    MPI_Finalize();
    return 0;
}

}/* main */

```

### ➤ Запитання для модульного контролю

1. Яка загальна схема концепції взаємодії процесів через посилання повідомень ?
2. Назвіть види операцій send/receive ?
3. Як здійснюється синхронізація процесів при передаванні повідомень ?

4. Як виконується передавання даних в мові Оккам?
5. Що включає поняття каналу і протоколу в мові Оккам?
6. Яке призначення та які види оператора `select` існує у мові Ада? Наведіть його аналог у мові Оккам?
7. Складові механізму рандеву в мові Ада?
8. Яким чином визначаються обсяг та напрям передавання даних в механізмі рандеву мови Ада?
9. В чому полягає відмінність між реалізацією механізму рандеву у мовах Оккам і Ада?
10. Які дії в PVM потрібні для відправлення повідомлення?
11. Які дії в MPI пов'язані з відправленням повідомлення?
12. Поняття комунікатору в MPI?
13. Як в MPI задається кількість необхідних процесів?
14. MPI. Призначення типу `MPI_Comm`?
15. Призначення MPI функції `Bcast()`?
16. Призначення MPI функції `MPI_Gather()`?
17. Призначення MPI функції `MPI_Scatter()`?
18. MPI. Параметри функції `Send()`?
19. MPI. Параметри функції `Recv()`?
20. MPI. Призначення колективних операцій?
21. Що виконує ця функція  
`MPI_Bcast(&data, 1, MPI_Init, 6, MPI_COMM_WORLD)`?

#### ➤ Завдання для самостійної роботи

1. Розробити програму мовою Ада, яка реалізує взаємодію трьох задач T1, T2 і T3. (Задача T1 передає в T2 два цілих числа x і z, одне з яких (x) потім передається в задачу T3).
2. Опрацювати завдання 1 з метою додаткового передавання задачі T3 через задачу T2 в задачу T1 вектора X з десяти елементів.
3. Розробити програму мовою Оккам, у якій задача T1 приймає від задачі T2 масив цілих чисел розміром 10.
4. Виконати модифікацію завдання 3 для забезпечення передавання масиву будь-якого розміру.
5. У завданні 3 забезпечити передачу крім масиву цілих, ще й масив типу `BYTE`.
6. Бібліотека MPI. Забезпечити передавання з однієї задачі в дві інші задачі двох векторів типу `int` і `float`.

- Розглянути передавання і приймання даних з блокуванням і без блокування.
7. Бібліотека MPI. Організувати розслання вектора з однієї за дачі в чотири інші.
  8. Бібліотека PVM. Забезпечити передавання з однієї задачі в три інші задачі двох векторів типу char і float.



## РОЗДІЛ 6.

### Розподілені обчисlenня

- 6.1. Організація розподілених обчисlenь.
- 6.2. Java. Сокети.
- 6.3. Java. RMI.
- 6.4. Ада.RPC.

Цей розділ присвячено питанням організації обчисlenь у розподілених комп'ютерних системах (РКС). Розглянуто механізми сокетів і видалених процедур та їх реалізацію і застосування в мовах Java і Ада.

#### **6.1. Організація розподілених обчисlenь**

Організація обчисlenь у розподілених комп'ютерних системах (розподілених обчисlenь) ґрунтуються на мережевих технологіях, що визначає її особливість при використанні моделі посилання повідомлень.

Основна проблема, яка пов'язана з організацією обчисlenь у розподілених комп'ютерних системах, – це проблема організації взаємодії вузлів системи. Використання комп'ютерних мереж як засобу передавання даних між вузлами накладає певні обмеження на методи та засоби такої організації. Якщо Інтернет (який є прикладом розподіленої системи) не потребує надшвидкого передавання даних, то *кластерні розподілені системи*, які орієнтовані виключно на обчисlenня великих обсягів даних, потребують супершвидкого передавання між вузлами системи. Тому в кластерних системах використовують спеціальне обладнання, орієнтоване на значне скорочення часу обміну інформацією в розподілених системах. Це стосується передусім мережених адаптерів і комутаторів.

Програмне забезпечення, потрібне для програмування в розподілених комп'ютерних системах, ґрунтуються на засобах, які тісно пов'язані з особливостями передавання даних у комп'ютерних мережах. Такі засоби традиційно представлені у

вигляді спеціальних бібліотек (WinAPI, PVM, MPI). Більш пізні мови програмування (Java, Ада, C#) мають вбудовані засоби програмування для розподілених систем. Зрештою, поширення потреби в розробленні розподілених додатків привело до необхідності створення проміжного (middleware) програмного забезпечення, яке виконує функції інтерфейсу між розподіленою програмою і розподіленою системою, і яке дозволяє спростити (автоматизувати) процес його розроблення і виконання (CORBA, COM/DCOM). Тому гама засобів розроблення розподілених програм сьогодні досить велика і безперервно поширюється, включаючи *сокети*, *віддалені процедури*, *віддалені об'єкти* та ін. [10], [11], [38], [42].

*Розподілена програма* – це набір *розділів*, з яких складається програма і які розміщаються та виконуються на різних вузлах розподіленої комп’ютерної системи.

Найпоширеніша для розроблення додатків модель *клієнт – сервер* вже стала класичною для розподілених обчислень. *Сервер* – об’єкт, що надає послуги (або спільні ресурси) іншим об’єктам (обчислювальні сервери, сервери друку, файл-сервери, Web-сервери). *Клієнт* – це об’єкт, що звертається до сервера за послугами. Тому розроблення розподіленого додатка пов’язано з побудовою паралельного алгоритму розв’язання завдання, визначенням у ньому серверної і клієнтської частин, розробленням алгоритмів сервера і клієнта та організації взаємодії сервера і клієнта з використанням належних бібліотечних або мовних засобів.

### 6.1.1. Сокети

Сокети – низькорівневий механізм, що застосовується для організації розподілених обчислень. Парадигма сокетів була запропонована під час створення ОС UNIX у Каліфорнійському університеті в Берклі [47].

*Сокет* – своєрідний мережевий роз’їм на логічному рівні, який дозволяє передавати і приймати дані між комп’ютерами, з’єднаними мережею. Сокет дозволяє серверу обслуговувати клієнта, чекаючи на його підключення, а клієнту – звертатися до сервера (рис. 6.1). Для створення сокета застосовують порт. *Порт* – абстрактне поняття, що є пронумерованим сокетом на

конкретному вузлі системи. Сервер “прослуховує” порт доти, доки клієнт не з’єднається з ним. Сервер може обслуговувати кількох клієнтів (рис. 6.2). При цьому оптимальний серверний додаток має бути реалізованим у вигляді кількох процесів (*багатопотоковий сервер*), кожен з яких відповідає за зв’язок зі своїм клієнтом.

Механізму сокетів реалізовано в мовах програмування (Java) і бібліотеках (Win32). Базові функції бібліотеки Win32, які використовуються для роботи із сокетами, наведено в таблиці В.3.

### 6.1.2. Віддалені процедури

Механізм виклику віддаленої процедури (механізм RPC – Remote Procedure Call) реалізує високорівневу концепцію взаємодії розділів розподіленої програми.

Зовнішнє віддалена процедура нічим не відрізняється від звичайної процедури. Її особливість полягає в тому, що розділ, у якому знаходиться віддалена процедура (сервер), і розділ, де здійснюється виклик віддаленої процедури (клієнт), знаходяться в різних вузлах розподіленої системи. Тому виклик та виконання віддаленої процедури пов’язано з трьома діями:

- 1) передавання вхідних параметрів віддаленої процедури з вузла клієнта у вузол сервера;
- 2) виконання віддаленої процедури на вузли сервера, де вона знаходиться;
- 3) повертання результату виконання віддаленої процедури з вузла сервера на вузол клієнта за допомогою вихідних параметрів.

Таким чином, аргументи віддаленої процедури передаються в мережі у вигляді повідомлень між вузлами розподіленої системи.

Клієнт ініціалізує виконання віддаленої процедури створюванням і запуском нового процесу на вузлі сервера і чекає його завершення, після чого клієнт продовжує своє виконання. Відмінність від механізму randevu полягає в тому, що останній потребує реалізацію програми сервера у вигляді активного модуля – процесу, в той час, як RPC-механізм дозволяє викорис-

товувати для розміщення віддаленої процедури пасивні модулі, наприклад, пакети.

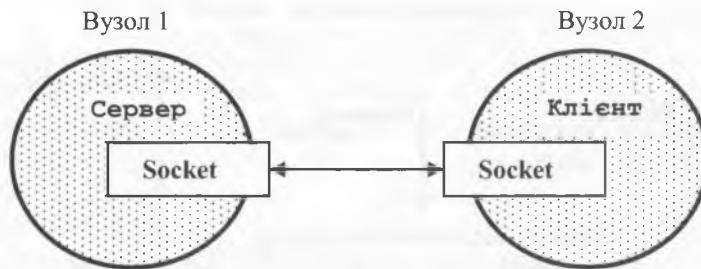


Рис. 6.1. Взаємодія розділів за допомогою сокетів

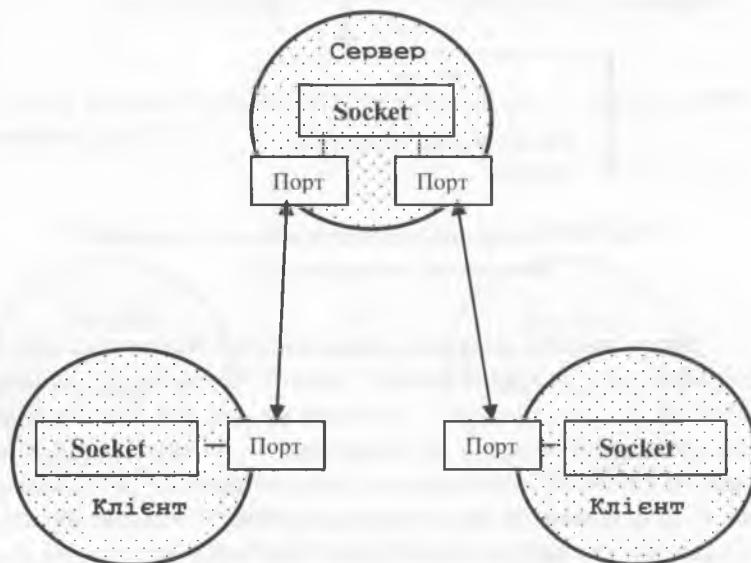


Рис. 6.2. Порти у взаємодії вузлів за допомогою сокетів

Часову діаграму, яка ілюструє взаємодію клієнта і сервера з використанням віддаленої процедури  $\text{Sum1}(X, Y: \text{in Вектор}; Z: \text{out Вектор})$ , показано на рис. 6.3.



Рис. 6.3. Синхронна взаємодія клієнта та сервера за допомогою механізму RPC

Друга часова діаграма, яка ілюструє взаємодію Клієнта і Сервера з використанням іншої віддаленої процедури  $\text{Sum2}(X, Y: \text{in Вектор})$ , показано на рис. 6.4. Виклик віддаленої процедури  $\text{Sum2}()$  асинхронний і не пов'язаний з очікуванням клієнтом виконання віддаленої процедури та повернення її результату. В цьому випадку клієнт передає входні дані віддаленої процедури  $\text{Sum2}()$  під час її виклику і потім продовжує своє виконання далі. Результат виконання віддаленої процедури  $\text{Sum2}()$  можна отримати від сервера пізніше за допомогою виклику іншої віддаленої процедури. Слід звернути увагу на то, що віддалена процедура, яка допускає асинхронний ви-

клик, має бути описана як асинхронна і вона *не може мати вихідні параметри* для повернення результату.

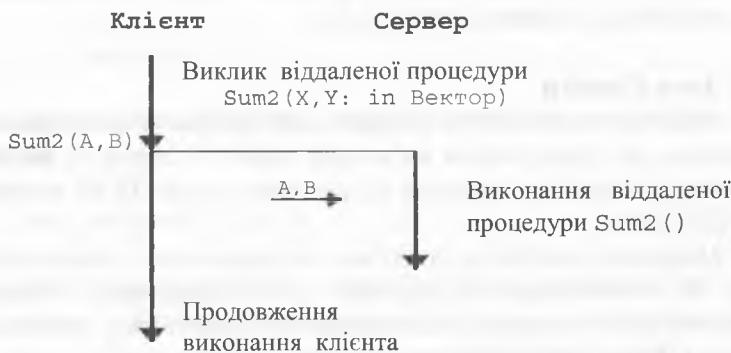


Рис. 6.4. Асинхронна взаємодія клієнта та сервера за допомогою механізму RPC

Схему взаємодії вузлів РКС за допомогою механізму RPC показано на рис. 6.5.

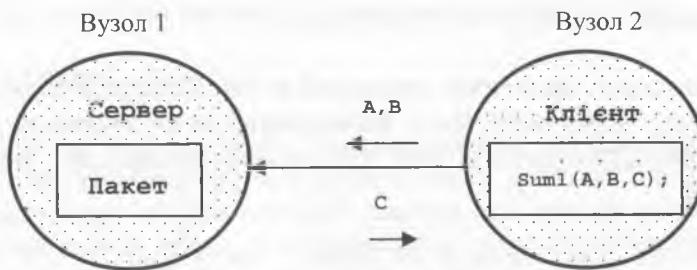


Рис. 6.5. Взаємодія вузлів за допомогою виклику віддаленої процедури `Sum1`

Серверний вузол (Вузол 1) містить пасивний модуль Пакет, у якому описано віддалену процедуру `Sum1()`. Клієнтський вузол (Вузол 2) містить активний модуль – процедуру Клі-

ент, у якому здійснюється виклик віддаленої процедури `Sum1()`.

Механізм RPC реалізовано багатьма мовами програмування, наприклад, мовами Java та Ада.

## 6.2. Java. Сокети

Мова Java забезпечує ресурси для побудови розподілених додатків, які ґрунтуються на моделі клієнт – сервер, у вигляді сокетів і механізму виклику віддалених методів (RMI механізму) [27], [59].

Механізм сокетів ґрунтується на парадигмах, застосовуваних у комп’ютерних мережах для організації взаємодії комп’ютерів, і залежить від використованого типу протоколу обміну (TCP, UDP, ICMP та ін.).

Механізм TCP сокетів у мові Java реалізований за допомогою класів `Socket` і `ServerSocket` з пакету `java.net`.

Клас `Socket` використовують для серверних і клієнтських частин розподіленого додатку. Він дозволяє створити `Socket` об’єкти, за допомогою яких клієнт може з’єднуватися із сервером, а потім здійснювати обмін даними між сервером та клієнтом. Клас має два конструктори, які використовуються для створювання сокетів. Конструктор

```
Socket(String Хост, int Порт)
```

створює сокет, що з’єднує локальний вузол з іншим вузлом на ім’я `Хост` через порт `Порт`. Конструктор може викинути виключення `IOException` або `UnknownHostException`. Конструктор

```
Socket(InetAddress IP_Адрес, int Порт)
```

створює сокет, що з’єднує локальний вузол із іншим вузлом з адресою `IP_Адрес` через порт `Порт`. Сокет може отримувати пов’язану з ним адресну та портову інформацію за допомогою спеціальних методів. Метод

```
InetAddress getInetAddress()
```

повертає InetAddress об'єкт, що пов'язаний з Socket об'єктом.

Метод

```
int getPort()
```

повертає віддалений порт, з яким з'єднаний Socket об'єкт.

Метод

```
int getLocalPort()
```

повертає локальний порт, з яким з'єднаний Socket об'єкт.

Після створення Socket об'єкта його можна зв'язувати з входним або вихідним потоком уведення—виведення (InputStream, OutputStream) для передавання (приймання) даних з іншого вузла.

Приклад створювання сокета Сокет1 і потоків уведення—виведення In і Os для запису і читання даних через сокет:

```
Socket Сокет1 = new Socket(12.12.12.11, 45);
InputStream In = Сокет1.getInputStream();
OutputStream Os = Сокет1.getOutputStream();
```

Сокет потрібно закрити після завершення роботи з ним:

```
Сокет1.close();
```

Клас ServerSocket використовують для створення сокетів на серверної частини розподіленого додатку. Він дозволяє створювати об'єкти (серверні сокети), які використаються для встановлення зв'язку з клієнтом. Для цього сервер виконує прослуховування порту, який визначений у сокеті, та очікує на підключення клієнта. Клас має три конструктори, які будуть використані для створюванні серверних сокетів. Конструктор

```
ServerSocket(int Порт)
```

створює серверний сокет на вказаному порті Порт з розміром черги за умовчанням (50). Конструктор

```
ServerSocket(int Порт, int Черга)
```

створює серверний сокет на вказаному порті Порт з максимальним розміром черги згідно з параметром Черга. Конструктор

```
ServerSocket(int Порт, int Черга,
             InetAddress ЛокальнаАдреса)
```

створює серверний сокет на вказаному порту Порт з максимальним розміром черги згідно з параметром Черга. На груповому хост-вузлі параметр ЛокальнаАдреса визначає IP адресу, з якою цей сокет пов'язаний.

Клас ServerSocket має ключовий метод accept(), який дозволяє серверу виконувати з'єднання з клієнтом:

```
СКС = СС.accept(),
```

де СКС та СС – звичайний та серверний сокети, що створені на серверної частині додатку,

Це з'єднання ініціює клієнт при створенні свого сокету. В сервері метод accept() виконує прослуховування порту, який визначено у серверному сокеті СС. Якщо зв'язку з боку клієнта немає, метод accept() блокує сервер і чекати на підключення клієнта. При підключені клієнта метод accept() отримує від клієнта службові данні (IP-адресу клієнту), який записує в звичайний сокет (Socket об'єкт) СКС, що створений на боці сервера. Цей об'єкт буде використаний далі сервером для зв'язку з клієнтом безпосередньо при передаванні основних даних клієнту. Тобто, на боці сервера створюється два сокета: серверний СС (ServerSocket об'єкт) для отримання службової інформації та звичайний СКС (Socket об'єкт) для обміну даними.

❖ Приклад створення та застосування сокетів на боці сервера і клієнта (рис. 6.6):

```
// сервер-----
try{
    ServerSocket СерверСк1 = new ServerSocket(65);
    Socket СерверСк2 = null;
```

```

        } catch(IOException e){
    }
    // прослухування порту з очікуванням
    // підключення клієнта за допомогою методу accept()
    СерверСк2 = СерверСк1.accept();
// клієнт-----
try{
    Socket КієнтСк = new Socket(10.12.121.12,65);
} catch(IOException e){
}

```

Серверний сокет СерверСк1 підключено до порту з номером 65, до якого тепер може підключитися клієнт, використавши об'єкт КлієнтСк з належною IP адресою сервера (10.12.121.12) і номером порту 65, який створено на боці клієнта. Після підключення до серверу клієнт передає серверу значення своєї IP – адреси (10.12.121.15), яка записується методом СерверСк1.accept() в об'єкт СерверСк2 на боці сервера. Подальший обмін даними між сервером та клієнтом буде здійснюватися виключно через звичайні сокети: СерверСк2 (на сервері) та КлієнтСк (на клієнті).

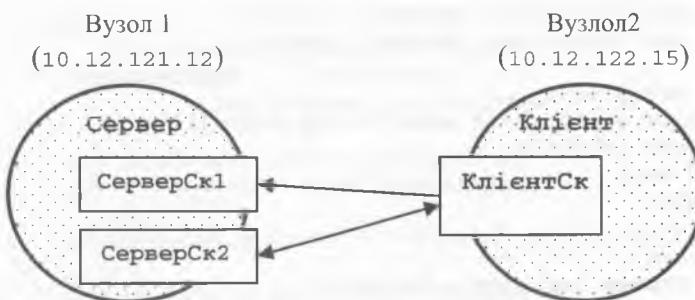


Рис. 6.6. Взаємодія вузлів за допомогою сокетів мови Java

Сокети TCP/IP забезпечують надійний зв'язок вузлів РКС. При цьому вартість зв'язку велика, оскільки, що TCP протокол використовує складні алгоритми для реалізації.

Поряд з TCP/IP сокетами Java дозволяє застосовувати для організації розподілених обчислень дейтаграмні сокети, які працюють в протоколах UDP. Дейтаграми – невеликі пакети даних, що транспортуються між вузлами розподіленої системи. Це менш надійний але більш швидкий механізм сокетної взаємодії. Для використання дейтаграм у мові Java існують класи DatagramPacket і DatagramSocket.

#### ❖ Приклад 6.1.

Приклад ілюструє взаємодію за допомогою сокетів класів Сервер і Клієнт, яка полягає в тому, що клас Сервер передає значення змінної X класу Клієнт, де воно розміщується у змінній Y. Приклад розроблено для використання на одному комп’ютері, тому потрібна IP - адреса в клієнтському сокеті формується за допомогою методу getLocalHost().

```
-----
// Java. Застосування сокетів
-----
import java.io.*;
import java.net.*;
class Lab51{
    // конструктор
    Lab51(){
        new Thread(new Сервер()).start();
        new Thread(new Клієнт()).start();
    }
    // метод main
    public static void main(String args[]){
        new Lab51();
    } // main
} // Lab51

// Сервер-----
class Сервер implements Runnable{

    public void run(){
        int X = 100;

        // створення сокетів
        ServerSocket сервер = null;
        Socket серверСокет = null;

        // створення потоку виведення
        OutputStream zz = null;
```

```
try{
    // формування серверного сокета
    сервер = ServerSocket(34, 10);

    // очікування на підключення клієнта
    серверСокет = сервер.accept();
    System.out.println("З'єднання на сервері");

    // формування потоку виведення zz
    zz = серверСокет.getOutputStream();

    // передавання даних клієнту
    zz.write(X);                                // запис в сокет
} catch(IOException e){ }
} // run
} // Сервер

// Клієнт -----
class Клієнт implements Runnable{
    public void run(){
        int Y;
        // створення сокета
        Socket клиентСокет = null;
        // створення потоку введення
        InputStream rr = null;

        try{
            // створення клієнтського сокету
            клиентСокет = new Socket
                (InetAddress.getLocalHost(), 34);

            // створення потоку введення rr
            rr = клиентСокет.getInputStream();

            System.out.println(" З'єднання на клієнти");

            // отримання даних від сервера
            Y = rr.read(); // читання із сокета
            System.out.println(" y = " + Y);

        } catch(UnknownHostException e){ }
        } catch(IOException e){ }
    } // run

} // Клієнт
```

### 6.3. Java.RMI

Реалізацію в Java механізму виклику віддалених процедур виконано за допомогою засобів, які отримали назву RMI (Remote Method Invocation) – *виклик віддаленого методу*. Він підтриманий пакетами `java.rmi` і `java.server`.

Реалізація механізму RMI потребує дій по створенню чотирьох класів:

- 1) *віддаленого інтерфейсу*, який розширяє інтерфейс `Remote` з пакета `java.rmi` і в якому визначено віддалені методи;
- 2) класу, який є реалізацією віддаленого інтерфейсу через розширення класу `UnicastRemoteObject` і ;
- 3) класу-серверу з методом `main()`; екземпляр сервера фіксується за допомогою служби реєстрації в RMI реєстрі і методу `Naming.rebind()`;
- 4) класу-клієнта, який визначає взаємодію з клієнтом; за допомогою методу `Naming.lookup` об'єкт отримує від служби реєстрації об'єкт сервера, та інформацію про сервер (IP – адреса або ім'я) і може викликати віддалені методи сервера.

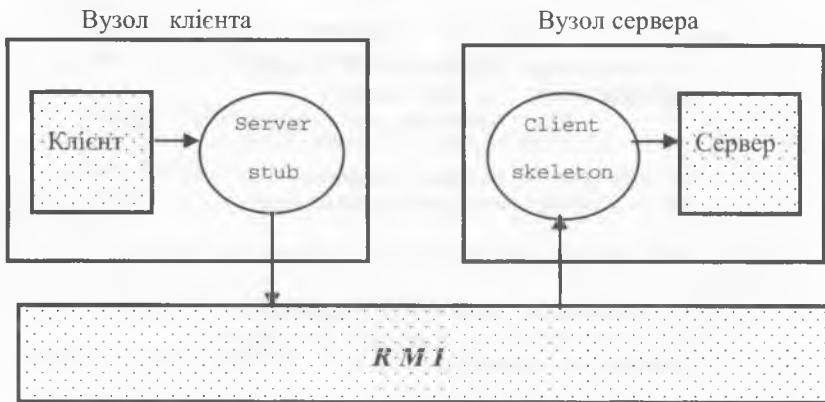


Рис. 6.7. Взаємодія сервера і клієнта в RMI під час виклику віддаленого методу

Взаємодія клієнта та сервера здійснюється за допомогою серверної заглушки (`server stub`) і клієнтського скелетона

(client skeleton) (рис. 6.7). Серверна заглушка – це Java об'єкт, який постійно знаходиться на машині клієнта. Клієнтський скелетон – це Java об'єкт, який постійно знаходиться на машині сервера. Він не використовується в Java2, але потрібний для сумісності Java 1.1 і Java 2. Генерація заглушки і скелетону здійснюється за допомогою RMI компілятора.

Виклик віддаленого методу від клієнта відправляється до заглушки, звідки за допомогою механізму RMI по комп'ютерній мережі з'єднається з клієнтським скелетоном на вузлі сервера, який знаходить і виконує віддалений метод на серверному вузлі.

### ❖ Приклад 6.2.

Приклад ілюструє взаємодію за допомогою механізму RMI класів Сервер і Клієнт, яка полягає в тому, що клас Клієнт знаходить додаток змінних  $X$  і  $Y$  ( $R = X + Y$ ) за допомогою використання віддаленого методу Обчислення() із класу Сервер.

```
//-----
// Java. Вастосування RMI механізму
//-----

// 1. Віддалений інтерфейс -----
import java.rmi.*;
interface ВіддаленийІнтерфейс extends Remote{
    int Обчислення(int x, int y) throws
                                         RemoteException;
}

// 2. Реалізація віддаленого інтерфейсу -----
import java.rmi.*;
import java.rmi.server.*;
class РеалізаціяВіддаленогоІнтерфейсу extends
    UnicastRemoteObject implements ВіддаленийІнтерфейс{
    int Обчислення(int x, int y) throws RemoteException
    {
        return(x + y);
    } // Обчислення
} // РеалізаціяВіддаленогоІнтерфейсу
```

```

// 3.Сервер -----
import java.net.*;
import java.rmi.*;
class Сервер{

    public static void main(String [] args){

        try{

            // Створення об'єкта
            РеалізаціяВіддаленогоІнтерфейсу  рві =
                new РеалізаціяВіддаленогоІнтерфейсу();

            // Регістрація об'єкта в rmiregistry
            Naming.rebind("Поліс",  рві);

            }catch(Exception e){
                System.out.println("Виключення " + e);
            }
        }// main

    }// Сервер

// 4.Клієнт -----
import java.rmi.*;
class Клієнт {
    public static void main(String [] args){

        try{
            int a = 100;
            int b = 150;
            int r;

            // встановлення диспетчера безпеки
            System.setSecurityManager(new
                RMISecurity Manager);

            // отримання віддаленого об'єкту
            ВіддаленийІнтерфейс  ві = new
                (Віддалений інтерфейс)Naming.lookup();

            // обчислення через виклик віддаленого методу
            r = ві.Обчислення(a,b);

            System.out.println("Результат = " + r);
        }
        catch(Exception e){
            ("Виключення :" + e)
        }
    }
}

```

## **Розділ 6. Розподілені обчислення**

```
    }  
}  
} // Клієнт
```

### **6.4. Ада. RPC**

Механізм RPC лежить в основі моделі розподілених обчислень, яка запропонована в другому стандарті мови Ада у додатку «Розподілені системи» (Annex E “Distributed Systems” - DSA) [65]. Додаток визначає поняття розподіленої комп’ютерної системи і розподіленої програми.

Розподілена комп’ютерна система – набір *вузлів* (*nodes*), які можуть зберігати та обробляти дані. Вузли можуть бути вузлами обробки (*processing nodes*) або вузлами зберігання (*storage nodes*).

Розподілена програма – набір *розділів* (*partitions*), які виконуються незалежно і взаємодія яких здійснюється на підставі механізму RPC. Розділи формуються з бібліотечних програмних модулів. Розміщення розділів на вузлах розподіленої системи отримало назву конфігурування розділів.

Розділи розподіленої програми поділяють на *активні* та *пасивні*. Пасивні розділи, на відміну від активних, не мають власної ниті керування. Активні розділи можуть бути конфігурковані на вузли обробки, пасивні – на вузли зберігання та вузли обробки.

*Віддалений доступ* (*remote access*) – це звернення до даних або виклик підпрограми, який здійснюється між розділами. При цьому визначаються розділ, що викликає, і розділ, якого викликають. Механізм RPC реалізований в мові Ада за допомогою визначення в розділі віддалених процедур, до яких можливий віддалений доступ з інших розділів.

Віддалені процедури мають бути оголошені в бібліотечно-му модулі (пакеті) з використанням спеціальних прагм категорії:

```
pragma Shared_Passive();  
pragma Remote_Type();  
pragma Remote_Call_Interface()
```

Приклад пакета Форт , у якому описано віддалену процедуру:

```
package Форт is
    pragma Remote_Call_Interface;
    procedure Work( ... );
end Форт;
```

Застосовано прагму `Remote_Call_Interface`, яка визна-  
чає всі ресурси пакета Форт як такі, що допускають віддалений  
доступ. В даному випадку – це процедура `Work`.

### ❖ Приклад 6.3.

Взаємодія двох розділів програми з використанням відда-  
леної процедури. Перший розділ містить пакет Ресурс, у  
якому описано віддалену процедуру Додавання\_Матриць.  
Другий розділ містить процедуру Клієнт, яка виконує віддален-  
ний виклик процедури Додавання\_Матриць.

```
-- Ада. Віддалена процедура. Операція MA=MB*MC
-----
package Ресурс is

    -- віддалений інтерфейс
    pragma Remote_Call_Interface;

    -- віддалена процедура
    procedure Додавання_Матриць (MB : in Матриця;
                                    MC : in Матриця;
                                    MA : out Матриця) ;
end Ресурс;

-- тіло пакета
package body Ресурс is

    -- реалізація віддаленої процедури
    procedure Додавання_Матриць (MB : in Матриця;
                                    MC : in Матриця;
                                    MA : out Матриця) is
begin
    for i in 1..N loop
        for j in 1 .. N loop
            MA(i)(j) := MB(i)(j) + MC(i)(j);
        end loop;
    end loop;
end;
```

```

end Додавання_Матриць;

end Ресурс;

-----
with Ресурс;
procedure Клієнт is

  MX,MY,MZ : Матриця;

  -- формування матриці MX
  for i in 1..N loop
    for j in 1 .. N loop
      MX(i)(j) := 1;
    end loop;
  end loop;

  -- формування матриці MY
  for i in 1..N loop
    for j in 1 .. N loop
      MY(i)(j) := 2;
    end loop;
  end loop;

  -- віддалений виклик (MZ = MX + MY)
  Ресурс.Додавання_Матриць (MX,MY,MZ);

  -- виведення результату
  for i in 1..N loop
    for j in 1 .. N loop
      put(MY(i)(j),4);
    end loop;
  end loop;

end Клієнт;

```

Під час віддаленого виклику процедури Додавання\_Матриць значення входних параметрів MX, MY передаються з розділу, що викликає, в розділ, який викликається; віддалена процедури Додавання\_Матриць виконується на вузлі, де розміщено пакет Ресурс і результат повертається в процедуру Клієнт через параметр MZ. При цьому під час виконання віддаленої процедури процес, що викликає процедуру, буде блокований і зможе продовжити своє виконання тільки після отримання результату від виконання процедури Додавання\_Матриць.

В окремих випадках розділ, що викликає, може не чекати на завершення виконання віддаленої процедури і розпочинати виконання наступного оператора, тобто мас місце асинхронний виклик віддаленої процедури. В мові Ада асинхронний виклик описується за допомогою прагми `Asynchronous`.

Модифікація раніше розробленого пакету Форт для забезпечення асинхронного виклику віддаленої процедури `R( ... )`:

```
package ФортM is
    pragma Remote_Call_Interface;
    procedure Work( ... );
    pragma Asynchronous (Work);
end ФортM;
```

Слід звернути увагу на те, що процедура `Work` тепер не може повернати результат і не повинна використовувати вихідні (`out`) параметри.

#### ❖ Приклад 6.4.

Це модифікація прикладу 6.3, яка забезпечує асинхронний виклик процедури `Додавання_Матриць()`. Вона потребує зміни параметрів процедури, тому що в ній не дозволяється використовувати вихідні параметри з модифікатором `out`. Для отримання результатів виконання віддаленої процедури в пакеті Ресурс описується допоміжна процедура `Отримати_MA`, яка викликається в клієнті пізніше. Схему взаємодії розділів такої програми показано на рис.6.8.

```
-- Ада. Асинхронна віддалена процедура
-----
package Ресурс is

    -- віддалений інтерфейс
    pragma Remote_Call_Interface;

    -- віддалені процедури
    procedure Додавання_Матриць (MB : in Матриця;
                                    MC : in Матриця);

    procedure Отримати_MA (MA : out Матриця);

end Ресурс;
```

```

-- тіло пакета
package body Ресурс is

    МТ: Матриця; -- локальна змінна

    -- реалізація віддалених процедур
    procedure Додавання_Матриць(МВ : in Матриця;
                                    МС : in Матриця) is
    begin
        for i in 1..N loop
            for j in 1 .. N loop
                МТ(i)(j) := МВ(i)(j) + МС(i)(j);
            end loop;
        end loop;
    end Додавання_Матриць;

    procedure Отримати_МА(МА : out Матриця) is
    begin
        МА:= МТ;
    end Отримати_МА;

end Ресурс;
-----
with Ресурс;
procedure Клієнт is
    МХ,МY,МZ : Матриця;

    -- формування матриці МХ
    for i in 1..N loop
        for j in 1 .. N loop
            МХ(i)(j) := 1;
        end loop;
    end loop;

    -- формування матриці МY
    for i in 1..N loop
        for j in 1 .. N loop
            МY(i)(j) := 2;
        end loop;
    end loop;

    -- віддалений виклик (обчислення на сервері)
    Ресурс.Додавання_Матриць(МХ,МY);

    . . .
    -- віддалений виклик (отримання результату)
    Ресурс.Отримати_МА(МZ);

```

```
-- виведення результату
for i in 1..N loop
    for j in 1 .. N loop
        put (MZ(i)(j),4);
    end loop;
end loop;

end Клієнт;
```

Віддалений виклик в мові Ада здійснюється трьома способами:

- прямим викликом підпрограми, яка оголошена як віддалена;
- непрямим викликом через значення віддаленого типу, що посилається на підпрограму;
- диспетчерським викликом через значення типу, що посилається на віддалений тип широкого класу.

Перший вид виклику реалізує статичне зв'язування розділів, другий і третій – динамічне.

Формування розділів розподіленої програми і зв'язування з вузлами розподіленої системи (конфігурування) виконуються за допомогою спеціальних програмних засобів, які не представлені в стандарті мови Ада. Вони визначаються конкретною реалізацією DSA. Система GLADE – приклад такої реалізації. Вона працює з компілятором мови Ада GNAT. GLADE містить спеціальну мову для декларування розділів і конфігурування, яка використовується для створювання додаткового файлу конфігурації (\*.cfg). Цей файл потім обробляється компілятором, вбудованим в GLADE.

Мова опису конфігурації, яку застосовано в GLADE, ґрунтуються на підсистемі взаємодії розділів стандарту (пакет System.RPC). Мова дозволяє описати розділи, зв'язати їх зі створюваними програмними модулями, вказати місце розташування цих модулів в РКС, спосіб запуску розподіленої програми (ручний або автоматичний), адрес вузла, на якому буде виконуватися кожний розділ і т.ін.

Приклад файлу конфігурації для програми з прикладу 6.3:

```
-- файл конфігурації    RR.cfg
configuration  RR is

    -- формування двох розділів
    Розділ_Клієнта : Partition := ();
    procedure Клієнт is in Розділ_Клієнта;

    Розділ_Сервера : Partition := (Ресурс);

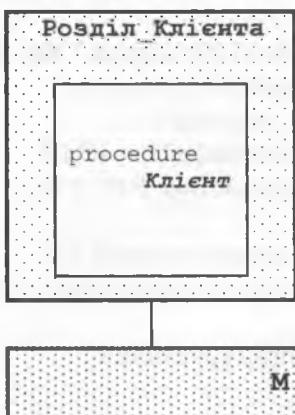
    -- зв'язування розділів з вузлами
    for Розділ_Клієнта'Host use "foo.bar.com";

    for Розділ_Сервера'Storage_Dir use
        "/usr/you/test/bin";

    -- визначення виду запуску програми
    pragma Starter(Method => Ada);

end RR;
```

Вузол 1



Вузол 2

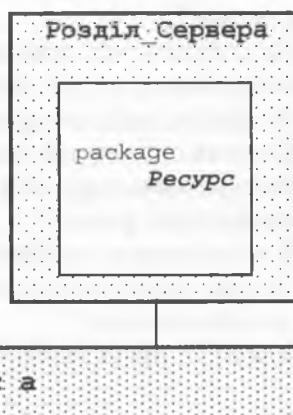


Рис. 6.8. Формування розділів

При формуванні розділів (рис. 6.8) розділ Розділ\_Клієнта містить головну процедуру програми Клієнт, з якої починається її виконання, а розділ Розділ\_Серверу містить RCI пакет Ресурс з віддаленою процедурою Додавання\_Матриць. Вказується адреса вузла, на якому потрібно виконувати розділ Partition\_2.

Прагма Starter (Method => Ada) визначає запуск програми з головної процедури.

Експериментальні дослідження ефективності RPC механізму у мові Ада наведено в роботі [62].

---

#### ➤ Запитанні для модульного контролю

1. У чому полягає особливість програмування для розподілених систем ? Які особливості має моделі клієнт – сервер ?
2. Види сокетів, які використаються в мові Java ?
3. Які функції виконує метод accept () ?
4. Яка службова інформація передається між сокетами в Java?
5. Які види сокетів використаються на боці серверу?
6. Як в методі запису в потік визначається зв'язок з потрібним сокетом ?
7. Де здійснюється виконання віддаленої процедури ? Які дії виконуються під час виклику віддаленої процедури ?
8. Які файли треба створити для RMI- додатку ?
9. Поняття і призначення віддаленого інтерфейсу в RMI?
10. Яке призначення прагм категорій в механізмі RPC ? Як формується розділ ?
11. В чому полягає особливість RPC механізму мови Ада під час переходу від звичайної моделі до розподіленої ? Як це здійснюється ?
12. Ада RPC. Які складові має мова конфігурування ?

#### ➤ Завдання до самостійної роботи

1. Розробити алгоритми серверу та клієнта для реалізації операції множення вектора на матрицю.
2. Розробити програму сортування вектора в розподіленій

- системі. Застосувати сокети (Java або Win32).
3. Розробити програму множення матриць в розподіленій системі. Застосувати MPI.
  4. Розробити програму множення матриць в розподіленій системі. Застосувати PVM.
  5. Розробити програму множення матриць в розподіленій системі. Застосувати RMI мови Java.
  6. Розробити програму множення матриць в розподіленій системі. Застосувати RPC мови Ада.



## РОЗДІЛ 7.

### Приклади програмування

- 7.1. Програмування для комп’ютерних систем зі спільною пам’яттю.
- 7.2. Програмування для комп’ютерних систем з локальною пам’яттю.
- 7.3. Програмування для розподілених комп’ютерних систем.

У цьому розділі розглянуто приклади розв’язання задач у комп’ютерних системах із спільною та локальною пам’яттю, а також в розподілених системах. Для обраних векторно-матричних задач виконано повний цикл створювання програм, який включає розроблення паралельного математичного алгоритму, алгоритмів кожного паралельного процесу, схеми взаємодії процесів, програми.

Наведено практичне використання семафорів, мютексів, захищених модулів, механізму randevu, а також сокетів і видалених підпрограм.

#### **7.1. Програмування для комп’ютерних систем зі спільною пам’яттю**

##### **7.1.1. Ада.Семафори**

Реалізація операції додавання векторів у двох процесорній системі зі спільною пам’яттю з використанням семафорів мови Ада.

Вхідні дані:

- комп’ютерна система зі спільною пам’яттю включає два процесори і два пристрої введення-виведення (рис. 7.1),
- 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 вектора  $A$  – у процесорі 1.



Рис. 7.1. 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

$$A_h = B_h + C_h * x \quad (7.1)$$

де  $A_h$  –  $H$  елементів вектора  $A$ .

Співвідношення (7.1) визначає дії в кожному процесорі системи під час виконанні паралельних обчислень.

Співвідношення (7.1) 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 співвід-

ношенні (7.1) є скаляр  $x$ , відносно якого необхідно розв'язати завдання взаємного виключення. Можливі два варіанти її розв'язування. Перший – виконанням обчислень (7.1) у критичної ділянці, другий – за допомогою попереднього створення копій змінної  $x$  (локальні змінні  $x_1$  і  $x_2$ ) для кожного процесу. Копіювання буде виконуватися в критичній ділянці і розглядається як завдання взаємного виключення.

Вибір підходу до розв'язування завдання взаємного виключення ґрунтуються на порівняльному аналізі часу виконання операцій в критичних ділянках. У нашому випадку послідовне копіювання змінної  $x$  потребує менше часу, ніж послідовне обчислення частин вектора  $AH$ , тому оберемо копіювання.

**Етап 2. Розроблення алгоритмів роботи кожного процесу.** Цей етап включає розроблення докладного алгоритму роботи кожного процесу. Такий алгоритм має включати *всі* дії процесу, які,крім безпосередніх обчислень за формулою (7.1), будуть включати введення даних і виведення результату, а також дії, що пов'язані з організацією взаємодії процесів (розв'язування завдання взаємного виключення та синхронізації). Завдання взаємного виключення буде пов'язана з копіюванням змінної  $x$ , завдання синхронізації – зі синхронізацією по:

- *введенню даних*, коли обидва процеси чекають на введення всіх даних і тільки після цього починають обчислення,
- *завершенню обчислень (виведенню результату)*, коли перший процес повинен дочекатися на завершення обчислень у другому процесі і тільки після цього виводити кінцевий результат.

В алгоритмі кожного процесу синхронізація (у *точках синхронізації*) буде реалізована діями **Чекати** і **Сигнал**. Точка синхронізації, яка пов'язана з очікуванням на подію, позначається як  $W_j, k$ , де  $j$  – номер задачі, у якій відбудеться подія;  $k$  – порядковий номер взаємодії цієї задачі та задачі  $Tj$  (якщо взаємодії декілька). Analogічно позначається точка синхронізації  $Sj, k$ , що пов'язана з сигналом до задачі  $Tj$  про подію, яка відбулася.

Задача T1

Точки  
синхронізації

1. Уведення  $B$  і  $x$
2. **Сигнал** задачі  $T2$  про введення  $B$  та  $x$  --  $S_2, 1$
3. **Чекати** на уведення даних в задачі  $T2$  --  $W_2, 1$
4. Копіювати  $x1 := x$  -- Критична ділянка (КД)
5. Обчислення  $A_n = B_n + C_n * x1$
6. **Чекати** на завершення обчислень в  $T2$  --  $W_2, 2$
7. Виведення результату  $A$

Задача T2

1. Уведення  $C$
2. **Сигнал** задачі  $T1$  про уведення  $C$  --  $S_1, 1$
3. **Чекати** на уведення даних в задачі  $T1$  --  $W_1, 1$
4. Копіювати  $x2 := x$  -- КД
5. Обчислення  $A_n = B_n + C_n * x2$
6. **Сигнал**  $T1$  про завершення обчислень --  $S_1, 2$

**Етап 3. Розроблення структурної схеми взаємодії 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 будуть використовуватися 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 з'являються семафори, що відповідають за роботу зі спільними ресурсами і потім використовуються в програмі.

На структурній схемі взаємодії задач (рис. 7.2) задіяні такі семафори:

- **Skd** – для керування доступом до спільногого 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;

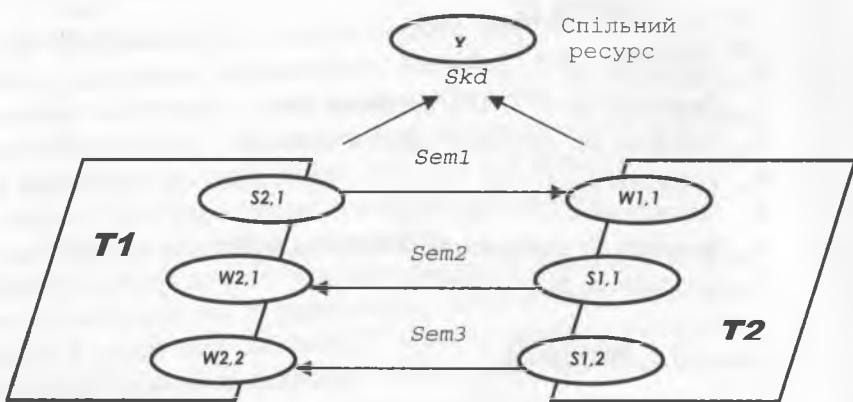


Рис. 7.2. Структурна схема взаємодії задач

**Етап 4. Розроблення програми.** Розроблення програми виконують на підставі алгоритмів задач і структурної схеми взаємодії задач:

```

--| Ада. Використання семафорів
--| Операція A = B + C * x
-----
procedure Lab71 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
            xl: integer;           -- локальна змінна
        
```

```

begin
    put_line("Process T1 started ");
    -- уведення В та x
    for i in 1.. N loop
        B(i) := 1;
    end loop;

    x := 2;

    -- сигнал задачі T2 про уведення В 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
    Set_True(Skd);

    -- обчислення AH
    for i in 1 .. N 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 С
    for i in 1.. N loop
        C(i) := 2;
    end loop;

```

```

-- сигнал задачі T1 про уведення С
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
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 Lab71;

```

### 7.1.2. Win32. Семафори і мютекси

Реалізація операції додавання векторів у двох процесорній системі із спільною пам'яттю з використанням бібліотеки Win32. Для приклада, що розглянутий в 7.1.1, для синхронізації процесів використовуємо семафори, а для взаємного виключення – мютекси.

Етапи 1– 3 не будуть відмінними від етапів, що розглянуті в прикладі 7.1. На структурній схемі (Рис.7.2) семафор Skd, буде змінений на мютекс Mut.

Для використання функцій Win32 в системі програмування ObjectAda потрібне ручне підключення бібліотеки Win32lib при створенні Ада-проекту і підключення до програми пакетів Win32, Win32.Base, Win32.Win32NT.

#### Етап 4. Розроблення програми

```
--| Ада.Використання бібліотеки Win32
--| Застосування семафорів та мютексів
--| Операція A = B + C * x
-----
procedure Lab72 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, Sem3, 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 В і x
p1 := ReleaseSemaphore(Sem1,1,NULL);

-- чекати на введення С у задачі T2
Temp:= WaitForSingleObject(Sem2,Infinite);

-- копіювання x в xl
Temp:= WaitForSingleObject(Mut,Infinite);
xl := x; -- критична ділянка
p1:= ReleaseMutex(Mut);

-- обчислення
for i in 1 .. N loop
    A(i):= B(i) + C(i)*xl;
end loop;

-- чекати на завершення обчислень у T2
Temp := WaitForSingleObject(Sem3, 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 С
    for i in 1.. N loop
        C(i):= 2;
    end loop;

    -- сигнал задачі T2 про введення С
    P2:= ReleaseSemaphore(Sem2,1,NULL);

    -- чекати на введення В та x у задачі T1
    Temp := WaitForSingleObject(Sem1,Infinite);

```

```

-- Копіювати x у x2
Term := WaitForSingleObject(Mut, Infinite);
x2:= x; -- критична ділянка
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:= ReleaseSemaphore(Sem3, 1, NULL);

put_line(" Process T2 finished");

end T2;

begin
    null;
end Запуск_Задач;

begin
    put_line(" Main procedure started ");

    -- створення семафорів і мютексу
    Sem1:= CreateSemaphore(NULL, 0, 1, NULL);
    Sem2:= CreateSemaphore(NULL, 0, 1, NULL);
    Sem3:= CreateSemaphore(NULL, 0, 1, NULL);
    Mut := CreateMutex(NULL, 0, NULL);

    -- виклик процедури для запуску задач
    Запуск_Задач;

end Lab72;

```

### 7.1.3. Ада. Захищений модулі

Реалізація операції додавання векторів у двох процесорній системі із спільною пам'яттю (рис. 7.1) з використанням захищеного модуля мови Ада.

Етапи 1–3 не будуть відрізнятися від етапів, розглянутих в прикладі в 7.1.

**Етап 3. Розроблення структурної схеми взаємодії задач.** Етап пов'язаний з розробкою структури захищеного модуля, за допомогою якого реалізується взаємодія задач. Захищений модуль Керування (рис. 7.3) включає три захищені елементи:  $x$ ,  $F1$  і  $F2$ , а також набір захищених операцій:



Рис. 7.3. Захищений модуль в операції  $A = B + C * x$

- вхід Чекати\_Введення для синхронізації з введення в задачах  $T1$  і  $T2$ ;
- вхід Чекати\_Обчислень, для синхронізації по завершенню обчислень у задачі  $T2$ ;
- функцію Копія\_Х для копіювання спільногоресурсу  $x$ ;
- процедуру Запис\_Х для запису значення  $x$  У захищений модуль;
- процедуру Сигнал\_Обчислень для сигналу про завершення обчислень У задачі  $T2$ ;
- процедуру Сигнал\_Введення для сигналу про завершення уведення даних у задачах  $T1$  і  $T2$ .

Захищені змінні  $F1$  і  $F2$  використовуються в бар'єрах входів Чекати\_Введення і Чекати\_Обчислень.

**Етап 4. Розроблення програми.** Розроблення програми виконується на підставі алгоритмів задач і структурної схеми взаємодії задач.

```
--|Ада.Захищений модуль
--|Операція A = B + C * x
-----
procedure Lab73 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_Введення;
procedure Сигнал_Обчислень;
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_Введення is
begin
    F1:= F1 + 1;
end Сигнал_Введення;
procedure Сигнал_Обчисень is
begin
    F2:= 1;
end Сигнал_Обчисень;

end Керування;
-----
-- задачі
task T1;

task body T1 is
    x1, z : integer; -- локальні змінні
begin
    put_line("Process T1 started");

    -- введення В та x
    for i in 1.. N loop
        B(i):= 1;
    end loop;
    z:= 2;

    -- запис х в захищений модуль
    Керування.Запис_X(z);

    -- Сигнал задачі T2 про введення В та x
    Керування.Сигнал_Введення;

    -- чекати на введення С в задачі T2
    Керування.Чекати_Введення;

    -- Копіювати x в x1
    x1:= Керування.Копія_X;

    -- обчислення
    for i in 1 .. N 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_Введення;

    -- 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_Обчислень;

    put_line(" Process T2 finished");

end T2;

-- тіло основної програми
begin
```

```

put_line("    Main procedure started ");
end Lab73;

```

#### 7.1.4. Java. Монітори

Реалізація операції скалярного множення векторів  $a = (B*C)$  в двох процесорній системі зі спільною пам'яттю (рис. 7.4) з використанням механізму моніторів у мові Java [59].

**Етап 1. Побудова паралельного алгоритму.** Паралельний алгоритм операції  $a = (B*C)$  можна подати у вигляді

$$\begin{aligned} ai &= (B_n * C_n) \\ a &= a + ai \end{aligned} \tag{7.2}$$

Спільним ресурсом є скаляр **a**, відносно якого необхідно розв'язувати завдання взаємного виключення під час формування кінцевого результату. Спільний ресурс **a** в (7.2) 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 процесу треба виконувати всю операцію  $a = a + ai$ .

#### Етап 2. Розроблення алгоритмів процесів

##### Задача P1

- |  | Точки<br>синхронізації |
|--|------------------------|
| 1. Уведення $B$ і $C$                      | ---                    |
| 2. Сигнал задачі P2 про введення $B$ і $C$ | $S_{2,1}$              |
| 3. Обчислення $a1 = (B_n * C_n)$           | ---                    |
| 4. Обчислення $a = a + a1$                 | КД                     |
| 5. Чекати на завершення обчислень в P2     | $W_{2,1}$              |
| 6. Виведення результату $a$                | ---                    |

##### Задача P2

- |                                  |           |
|----------------------------------|-----------|
| 1. Чекати на уведення в P1       | ---       |
| 2. Обчислення $a2 = (B_n * C_n)$ | $W_{1,1}$ |

3. Обчислення  $a = a + a_2$  -- КД  
 4. Сигнал  $P1$  про завершення обчислень --  $S1, 1$



Рис. 7.4. Структурна схема комп'ютерної системи

**Етап 3. Розроблення структурної схеми взаємодії задач.** Етап пов'язаний з розробленням структури класу (монітора), за допомогою якого реалізується взаємодія задач. Клас Data (рис. 7.5) включає три поля:  $a$ ,  $F11$  і  $F12$ , 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 Lab74 і пакета Vector, який включає три класи:

- Data – клас для синхронізації потоків;
- aThread – клас для потоку P1;
- bThread – клас для потоку P2.

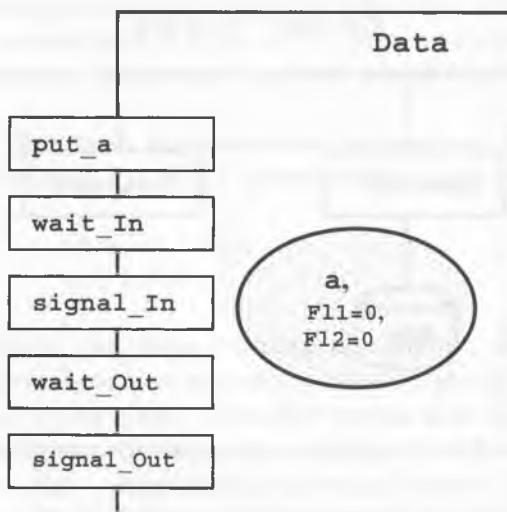


Рис. 7.5. Java. Структура класу-монітору Data

```

/*
-- Java. Синхронізовані методи. Операція a = (B*C)
*/
package Vector;

// монітор
public class Data {

    int N = 1000;
    int H = N/2;

    // змінні
    int a = 0;
    int B[] = new int [N];
    int C[] = new int [N];
}

```

```
private int F11 = 0;
private int F12 = 0;

public synchronized void
    put_a(int x){ a = a + x; }

public synchronized void wait_In(){
    try{
        if (F11 == 0)
            wait();
    }catch(Exception e) {}
}

public synchronized
    void signal_In(){
    notify();
    F11 = 1;
}

public synchronized void wait_Out(){
    try{
        if(F12 == 0)
            wait();
    }catch(Exception e) {}
}

public synchronized
    void signal_Out(){
    notify();
    F12 = 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 Р2
Z.wait_Out();                                // W2,1

// Виведення результату
Z.get_a();
}

// 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 Р1
        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 Lab74 {

```

```

public static void main (String [] args) {
    Data D = new Data();
    aThread P1 = new aThread(D);
    bThread P2 = new bThread(D);
    P1.start();
    P2.start();
}
}//Lab74

```

### 7.1.5. Бібліотека OpenMP

Розглянемо реалізацію задачі скалярного множення (7.1.4) для КС із спільною пам'яттю (рис. 7.6) з використанням засобів бібліотеки OpenMP. В системі використається два пристрой введення-виведення, тому зміняться алгоритми процесів (Етап 2). Етап 1 не змінюється (співвідношення 7.2)



Рис. 7.6. Структурна схема комп'ютерної системи

## Етап 2. Розроблення алгоритмів потоків

### Потік P1

	Точки синхронізації
1. Введення $B$	
2. <b>Сигнал</b> задачі $P2$ про введення $B$	$S2, 1$
3. Чекати на введення в $P2$	$W2, 1$
3. Обчислення $a1 = (Bn * Cn)$	
4. Обчислення $a = a + a1$	КД
5. <b>Чекати</b> на завершення обчислень в $P2$	$W2, 2$
6. Виведення результату $a$	

### Потік P2

1. Введення $C$		
2. <b>Сигнал</b> задачі $P1$ про введення $C$	$S2, 1$	
3. <b>Чекати</b> на уведення в $P1$	$W1, 1$	
4. Обчислення $a2 = (Bn * Cn)$		
5. Обчислення $a = a + a2$	КД	
6. <b>Сигнал</b> $P1$ про завершення обчислень	$S1, 2$	

## Етап 3. Розробка схеми взаємодії процесів

Взаємодія потоків  $P1$  і  $P2$  пов'язана з вирішенням наступних завдань:

- синхронізації по введенню даних ( $B$  і потоці  $P1$  і  $C$  в потоці  $P2$ );
- синхронізації по завершенню обчислень (потік  $P1$  чекає на завершення обчислень в  $P2$ );
- завдання взаємного виключення при роботі потоків із спільним ресурсом (змінною  $a$ ) .

Побудову паралельної програми можна виконати різними способами. В нашому випадку використаються прагми `parallel`, `sections`, `section` (рис. 7.7). Для вирішення завдання взаємного виключення використаємо механізм *критичних секцій*, для синхронізації – *бар’єри*.



Рис. 7.7 OpenMP.Операція  $a=(B*C)$ .Структура програми

#### Етап 4. Розробка програми

```

// -----
// OpenMP. Операція a = (B*C)
// -----
#include <omp.h>

#define N 1000

int main(int argc, char *argv[]){
    int B[N], C[N];
    int a = 0;
    int H = N/2;

    // Перша паралельна ділянка
    // Паралельне введення даних
    #pragma omp parallel num_threads(2)
  
```

```

sections{
// ----- P1
#pragma omp section{
    // Потік P1
    printf("Start P1\n");
    // Введення B
    for (int i = 0; i<N; i++) {
        B[i] = 1;
    }
}
// ----- P2
#pragma omp section{
    // Потік P2
    printf("Start P2\n");
    // Введення C
    for (int i = 0; i<N; i++) {
        C[i] = 2;
    }
}
} // закінчення першої паралельної ділянки

// Друга паралельна ділянка
// Паралельні обчислення
#pragma omp parallel num_thread(2)
    shared (B,C a) sections{

// ----- P1
#pragma omp section{
    // Потік P1
    int a1=0;
    for (int i = 0; i<N; i++) {
        a1 = a1 + B[i]*C[i];
    }

#pragma omp critical(CS1){
    a = a + a1;      -- КД
}

// синхронізація по обчисленню
#pragma omp barrier;

// виведення результату
printf(" a = \n", a);

}
// ----- P2
#pragma omp section{

```

```

// Потік P2
int a2=0;
for (int i = H; i<N; i++){
    a2 = a2 + B[i]*C[i];
}
#pragma omp critical(CS1){
    a = a + a2;      -- КД
}
// синхронізація по обчисленню
#pragma omp barrier;

}
} // закінчення другої паралельної ділянки
printf(" Завершення програми \n", a);
}//main

```

## 7.2. Програмування для комп'ютерної системи з локальною пам'яттю

### 7.2.1. Ада. Рандеву

Реалізація операції додавання векторів  $A = B + C^*x$  у трьох процесорній системі з локальною пам'яттю (три процесори і два пристрой введення-виведення (ПВВ)) з використанням механізму рандеву мови Ада.

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 цієї операції буде аналогічним алгоритму, розглянутому в прикладі в 7.1.

**Етап 2. Розроблення алгоритмів роботи кожного процесу.** Цей етап пов'язаний з розробленням алгоритмів роботи кожного процесу. 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:

- 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  $T_1$ ;
- збирання результату в задачі  $T_2$ .

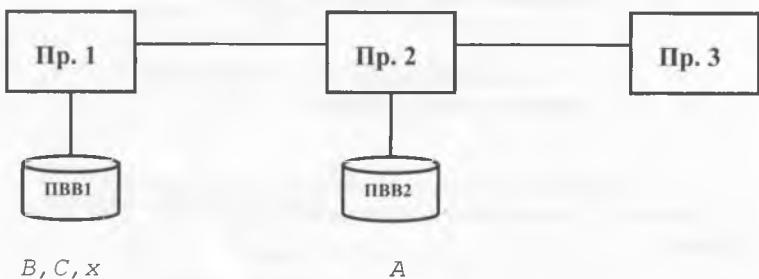


Рис. 7.8. Структура КС з локальною пам'яттю

Розробляючи алгоритм задач слід, звернути увагу на обсяги даних, що пересилаються між задачами. Під час розсилання вхідних даних із задачі  $T_1$  в  $T_2$  потрібно переслати дві частини вектора  $B$  і  $C$  розміром  $2^*H$  (позначається як  $B_{2H}$  і  $C_{2H}$ ), одна з яких призначена безпосередньо для  $T_2$ , а друга ( $B_h$ ,  $C_h$ ) – для пересилання далі в задачу  $T_3$ . 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 $T_1$

- Уведення  $B$ ,  $C$  та  $x$
- Передати** задачі  $T_2$  дані:  $B_{2H}$ ,  $C_{2H}$ ,  $x$
- Обчислення  $A_h = B_h + C_h * x$
- Передати** в задачу  $T_2$  частину результату  $A_h$

**Задача Т2**

1. **Прийняти** від задачі *T1* дані:  $B_{2n}$ ,  $C_{2n}$ ,  $x$
2. **Передати** в задачу *T3* дані  $B_n$ ,  $C_n$ ,  $x$
3. Обчислення  $A_n = B_n + C_n * x$
4. **Прийняти** від *T1* результат – вектор  $A_n$
5. **Прийняти** від *T3* результат – вектор  $A_n$
6. Виведення результату – вектор  $A$

**Задача Т3**

1. **Прийняти від** задачі *T2* дані:  $B_n$ ,  $C_n$ ,  $x$
2. Обчислення  $A_n = B_n + C_n * x$
3. **Передати** в задачу *T2* результат –  $A_n$

**Етап 3. Розроблення структурної схеми взаємодії задач.** Структурна схема взаємодії задач ґрунтуються на алгоритмах задач і дозволяє наочно показати передавання даних між задачами (рис. 7.9 ).

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 *две* напрямлені лінії. Одна лінія визначає виклик входу належної задачі, друга, коротка – напрям передавання даних, яка буде виконуватися за цім викликом. Біля короткої стрілки вказується інформація, що передається, наприклад, два вектори і матриця. Ця інформація потім використовується для опису входів задач у програмі.

Графічне зображення взаємодії задач дозволяє також виявити можливі *туникової ситуації* в програмі у випадку, якщо вход 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.

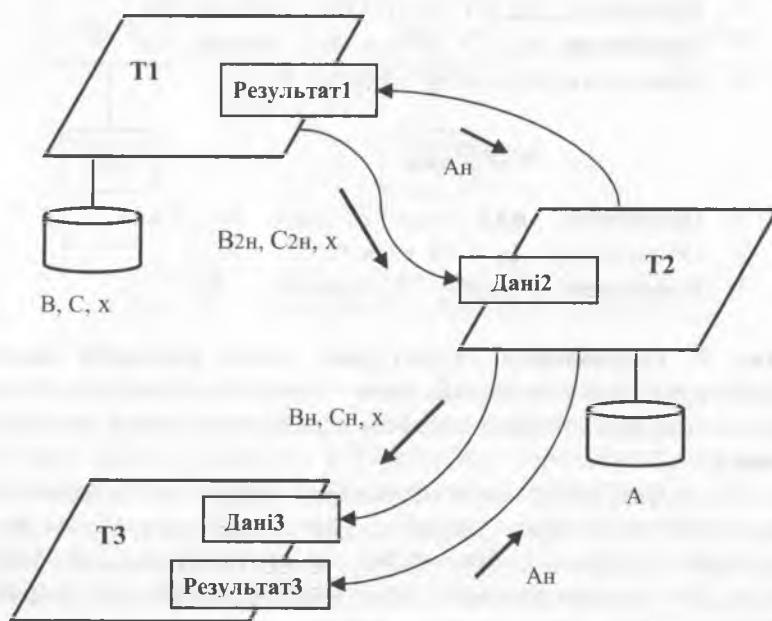


Рис. 7.9. Структурна схема взаємодії задач

**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.

```
-- | Ада.Механізм рандеву
-- | Операція A = B + C * x

procedure Lab75 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 Результат1(VA: out ВекторН);
end T1;
task T2 is
    entry Дані2(VB,VC : in Вектор2Н;
                 Y : in integer);
end T2;
task T3 is
    entry Дані3(VB,VC : in ВекторН;
                 Y : in integer);
    entry Результат3(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 B2Н, C2Н, x в задачу T2
    T2.Дані2(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 Результат1(VA: out ВекторН) do
        VA := A1;
    end Результат1;

    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 B2Н, C2Н і x від Т1
    accept Дані2(VB,VC : in Вектор2Н;
                  y : in integer) do
        B2 := VB;
        C2 := VC;
        X2 := Y;
    end Дані2;

    -- передати данні в Т3
    Т3.Дані3(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 Т1
    Т1.Результат1(A2(1..H));

    -- прийняти результат від Т3
    Т3.Результат3(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 Т2 ВН, СН і x
    accept Дані3(VB,VC : in ВекторН;

```

```

        y : in integer) do
      B3 := VB;
      C3 := VC;
      X3 := y;
    end Дани3;

    -- Обчислення АН
    for i in 1 .. N loop
      A3(i) := B3(i) + C3(i)*x3;
    end loop;

    -- передати АН в Т2
    accept Результат3(VA: out ВекторН) do
      VA := A3;
    end Результат3;

    put_line("Process T3 finished");

  end Т3;

  -- тіло основної програми
begin
  put_line(" Main procedure started ");
end Lab75;

```

### 7.2.2. Бібліотека PVM

Реалізація операції множення матриць  $MA=MB * MC$  в п'яти процесорній системі з локальною пам'яттю (рис.7.10) з використанням бібліотеки PVM та мови С.

**Етап 1. Побудова паралельного алгоритму.** Паралельний алгоритм для операції множення матриць:

$$MA_H = MB_H * MC$$

**Етап 2. Розроблення алгоритмів роботи кожного процесу.** Цей етап пов'язаний з розробленням алгоритмів роботи процесу Майстер (процесор 0) та Робочий (процесори 1–4).

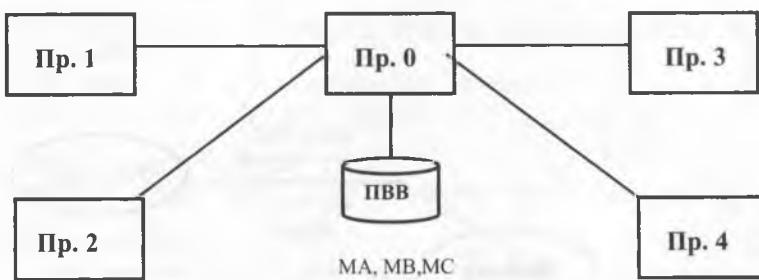


Рис. 7.10 Структура КС з локальною пам'яттю

**Майстер**

1. Уведення  $MB$ ,  $MC$
2. Активізація процесів Робочий
3. Розсилання всім процесам Робочий матриці  $MC$
4. Передати  $MB_n$  кожному процесу Робочий
5. Обчислення  $MA_n = MB_n * MC$
6. Прийняти  $MA_n$  від кожного процесу Робочий
7. Виведення результату – матриці  $MA$

**Робочий**

1. Прийняти матрицю  $MC$  від процесу Майстер
2. Прийняти  $MB_n$  від процесу Мастер
3. Обчислення  $MA_n = MB_n * MC$
4. Передати  $MA_n$  до процесу Майстер

### Етап 3. Розроблення структурної схеми взаємодії задач

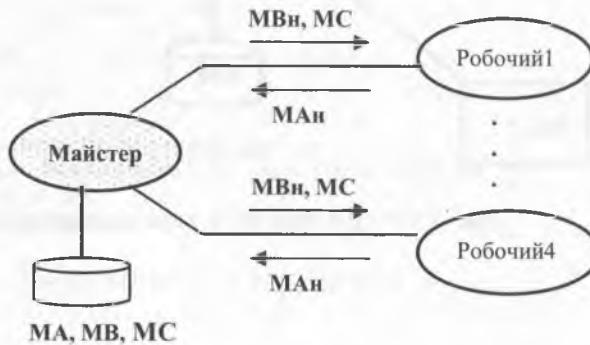


Рис. 7.11. PVM. Взаємодія задач. Операція  $MA=MB \cdot MC$

### Етап 4. Розроблення програми [54]

```
/*
-----*
 PVM. 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],agvc, 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
         MBn (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 MAH */
    }
}

```

```

for(i=1; i<P; i++){
    pvm_recv(Child[i],1);
    pvm_upkint(&MA[i][0], N,1);
}

/* виведення MA */
ДрукМасиву("MA = ", MA);

}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 MC */

pvm_recv(pid,1);
pvm_upkint(&VTemp,N*N,1); /* отримати MB */

/* обчислення MA */
МноженняМатриць(MTemp, VTemp, MAH, N);

/* відправлення результату в процес Майстер */
pvm_initsend(PvmDataDefault);
pvm_pkint(MAH, 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){

}

```

### 7.2.3. Бібліотека MPI

Реалізація операції множення матриць  $MA=MB * MC$  в п'яти процесорній системі з локальною пам'яттю (рис.7.11) з використанням бібліотеки MPI та мови С.

**Етап 1. Побудова паралельного алгоритму.** Паралельний алгоритм для операції множення матриць:

$$MA_n = MB_n * MC$$

**Етап 2. Розроблення алгоритмів роботи кожного процесу.** Цей етап пов'язаний з розробленням алгоритмів роботи процесу T0 (процесор 0) та T1-T4 (процесори 1-4).

### T0

1. Уведення  $MB$ ,  $MC$
2. **Розсилання** процесам T1-T4 значення  $H$
3. **Розсилання** процесам T1-T4 матриці  $MC$
4. **Розсилання** процесам T1-T4 матриці  $MB_n$
5. Обчислення  $MA_n = MB_n * MC$
6. **Прийняти**  $MA_n$  від процесів T1-T4
7. Виведення результату – матриці  $MA$

### T1-T4

1. **Прийняти** матрицю  $MC$  від процесу T0
2. **Прийняти**  $MC$  від процесу T0
3. **Прийняти**  $MB_n$  від процесу T0
4. Обчислення  $MA_n = MB_n * MC$
5. **Передати**  $MA_n$  процесу T0

**Етап 3. Розроблення схеми взаємодії процесів**

Взаємодія задачі T0 з задачами T1-T4 може здійснюватися за допомогою операцій точка-точка (рис. 7.12):

- Send() передавання H, матриць  $MC$ ,  $MB_n$
- Recv() приймання результату - частини матриці  $MA_n$ .

Можливе також застосування колективних операцій для розсилання даних і збирання результату: Bcast(), Scatter(), Gather().

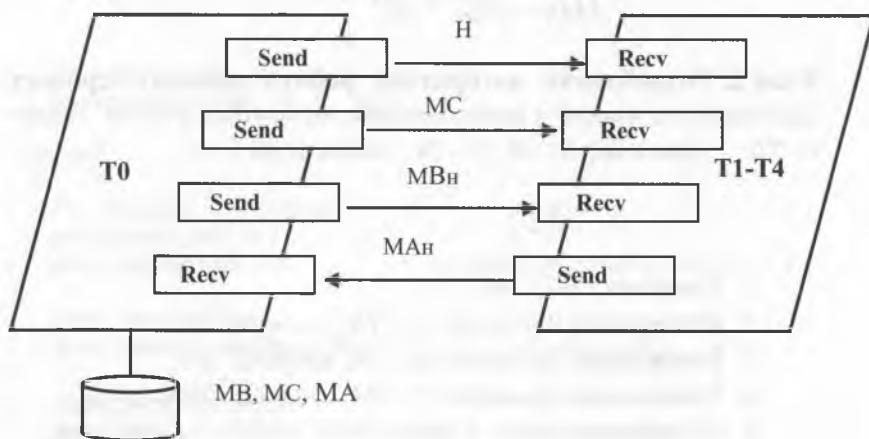


Рис. 7.12. MPI. Структурна схема взаємодії процесів

#### Етап 4. Розроблення програми

```
/*
MPI. Операція MA = MB*MC
*/
#include "mpi.h"
#include <stdio.h>
#include <string.h>

#define N 5
#define P 5
#define H 1

int main(int argc, char **argv) {
    MPI_Status status;

    int MA[N][N];
    int MB[N][N];
    int MC[N][N];

    /* тимчасові масиви */

```

```

int MAT[H][N];
int MBT[H][N];
int MCT[N][N];

int myTid;      /* ідентифікатор процесу */
int tag;        /* таги повідомлення */

MPI_Init(&argc, &argv);

/* отримати свій tid */
MPI_Comm_rank(MPI_Comm_World, &myTid);

/* Процес Т0 ----- */
if (myTid == 0){
    printf("Старт задачі Т0");

    /* введення матриць MB і MC */
    for (int i = 0; i<N; i++){
        for (int j = 0; j<N; j++) {
            MB[i][j] = 1;
            MC[i][j] = 2;
        }
    }

    /* переслати процесам Т1-Т4 1-4 розмір підматриці H */
    for (int i = 1; i<5; i++)
        MPI_Send(H, MPI_int, 1, i, tag, MPI_Comm_World);

    /* переслати процесам Т1-Т4 матрицю MC */
    for (int i = 1; i<P; i++)
        MPI_Bcast(MC[0][0], N*N, MPI_int, i, tag,
                  MPI_Comm_World);

    /* переслати процесам Т1-Т4 частини матриці MBH */
    for (int i = 1; i<P; i++)
        MPI_Send(MB[H*I][0], MPI_int, H*N, i,
                  tag, MPI_Comm_World);

    /* обчислення MAH у Т0 */
    for (int i = 0; i<H; i++)
        for (int j = 0; j<N; j++)
            MA[i][j] = 0;
        for (int k = 0; k<N; j++)
            MA[i][j] = MA[i][j]+MB[i][k]*MC[k][j];

    /* прийняти результату MAH від задач Т1-Т4 */
    for (int i = 1; i<P; i++)

```

```

MPI_Recv(MA[H*i][0], H*N, MPI_INT,
           i, tag, MPI_Comm_World, &status);

/* виведення MA */

ДрукМасиву(MA);

printf("Завершення задачі T0");

}else{

/* Процеси T1-T4 -----*/
printf("Старт задачі", myTid);

/* отримати розмір масиву H від Т0 */
mpi_Recv(H, 1, MPI_INT, 0, tag,
            MPI_Comm_World, &status);

/* прийняти від Т0 матрицю MC */
MPI_Recv(MCT[0][0], N*N, MPI_INT, 0, tag,
            MPI_Comm_World, &status);

/* прийняти від Т0 частину матриці MBn */
MPI_Recv(MBT[0][0], H*N, MPI_int, 0, tag,
            MPI_Comm_World, &status);

/* обчислення MAh у T1-T4 */
МноженняМатриць(MAT, MBT, MCT);

/* передати результат MAh задачі Т0 */
MPI_Send(MAT[0][0], H*N, MPI_INT, 0,
            MPI_Comm_World);

printf("Завершення задачі", myTid);
}

mpi_Finalize(); /* завершення і вихід з mpi */

exit(0);
}

/* процедура обчислення MAh = MBn * MC */
МноженняМатриць(. . .) {
}

```

```

ДрукМасиву( . . . . ) {
}

```

### 7.3. Програмування для розподілених комп'ютерних систем

У цьому розділі наведено приклади реалізації операції додавання векторів  $A = B + C$  в розподіленій комп'ютерній системі (рис. 7.13) з трьома вузлами і одним пристроєм введення–виведення (ПВВ) мовою Java (використання механізму сокетів) та мовою Ада (використання RPC механізму).

Обидва приклади використовують один і той же паралельний алгоритм для додавання векторів і ґрунтуються на моделі клієнт – сервер. Тому етапи 1 та 2 для них будуть одинаковими.

**Етап 1. Побудова паралельного алгоритму.** Паралельний алгоритм реалізації цієї операції буде мати вигляд

$$A_H = B_H + C_H, \quad (7.3)$$

де  $A_H$  –  $H$  елементів вектора  $A$ .

Співвідношення (7.3) визначає дії вузлів 2 і 3 розподіленої системи під час виконанні обчислень.

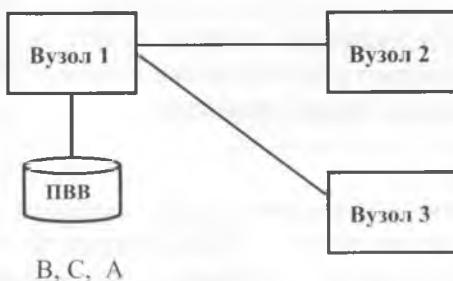


Рис. 7.13. Структура розподіленої системи

**Етап 2. Розроблення алгоритмів роботи розділів для кожного вузла.** Цей етап пов'язаний з розробленням алгоритмів роботи сервера та клієнта. Сервер буде виконуватися на першому вузлі, клієнти – на другому і третьому.

### Сервер

1. Уведення  $B, C$
2. **Передати** Клієнту1 дані:  $B_n, C_n$
3. **Передати** Клієнту2 дані:  $B_n, C_n$
4. **Прийняти** від Клієнта1 результат – вектор  $A_n$
5. **Прийняти** від Клієнта2 результат – вектор  $A_n$
6. Виведення результату – вектора  $A$

### Клієнт 1

1. **Прийняти** від Сервера вектори  $B_n, C_n$
2. Обчислення  $A_n = B_n + C_n$
3. **Передати** Серверу результат – вектор  $A_n$

### Клієнт 2

1. **Прийняти** від Сервера вектори  $B_n, C_n$
2. Обчислення  $A_n = B_n + C_n$
3. **Передати** Серверу результат – вектор  $A_n$ .

Наступні етапи розроблення програм будуть відрізнятися, оскільки використовуються різні механізми реалізації посилання повідомлень: сокети та віддалені процедури.

#### 7.3.1. Java. Сокети

**Етап 3. Розроблення структурної схеми взаємодії розділів.** Передавання (приймання) даних у Сервері виконується в Java за допомогою серверних сокетів, в Клієнти – за допомогою звичайних сокетів. У Сервері після створення серверного сокета виконується його прослуховування через метод `accept()` з метою очікування підключення клієнтів і подальшого читання

або запису даних в потоки введення–виведення. Для незалежної (паралельної) взаємодії з кожним клієнтом у *Сервері* створюється два потоки MyServer1 та MyServer2, кожен з яких і буде безпосередньо підтримувати зв'язок із своїм *Клієнтом*, де створені потоки MyClient (рис. 7.14). СІ– службова інформація, що необхідна для створення сокету SerSock1 і яка передається до Серверу при першому підключені Клієнта.

**Етап 4. Розроблення програми.** Розроблення програми виконується на підставі алгоритмів потоків і структурної схеми взаємодії класів.

```
/*
-- Lab77.Java.Сокети. Операція A = B + C
*/
import java.io.*;
import java.net.*;
public class Lab77 {
    Lab77() {
        Data x = new Data();
        new Thread(new MyServer1(x)).start();
        new Thread(new MyServer2(x)).start();
        new Thread(new MyClient1()).start();
        new Thread(new MyClient2()).start();
    }

    public static void main(String[] args) {
        new Lab67();
    }
}

// Lab77

class Data{
    int N = 4;           // Розмір векторів
    int Cl=2;           // Кількість клієнтів
    int H = N/Cl;

    // Описування масивів
    int [] A = new int[N];
    int [] B = new int[N];
    int [] C = new int[N];

    int F1 = 1;          // Пралор для синхронізації
```

```

// Конструктор
Data(){
    B[0]=1; B[1]=1;B[2]=1; B[3]=1;
    C[0]=2; C[1]=2;C[2]=2; C[3]=2;
}

// Методи для синхронізації
synchronized void Wait(){
    try{
        if (F1==1) wait();
    }

    }catch(Exception e){}
}

synchronized void Signal(){
    try{
        notify();
        F1 = F1 -1;
    }catch(Exception e){}
}
}// Data

// Потік на Сервері для взаємодії з першим клієнтом
class MyServer1 implements Runnable{
    Data d1;
    MyServer1(Data d) {

d1 = d;
    }

    public void run(){

System.out.println("MyServer1 started ");

ServerSocket srv1 = null;
Socket serSock1 = null;

OutputStream os1 = null;
InputStream is1 = null;

try{
    srv1      = new ServerSocket(77);
    serSock1 = srv1.accept();

    os1 = serSock1.getOutputStream();
    is1 = serSock1.getInputStream();
}

```

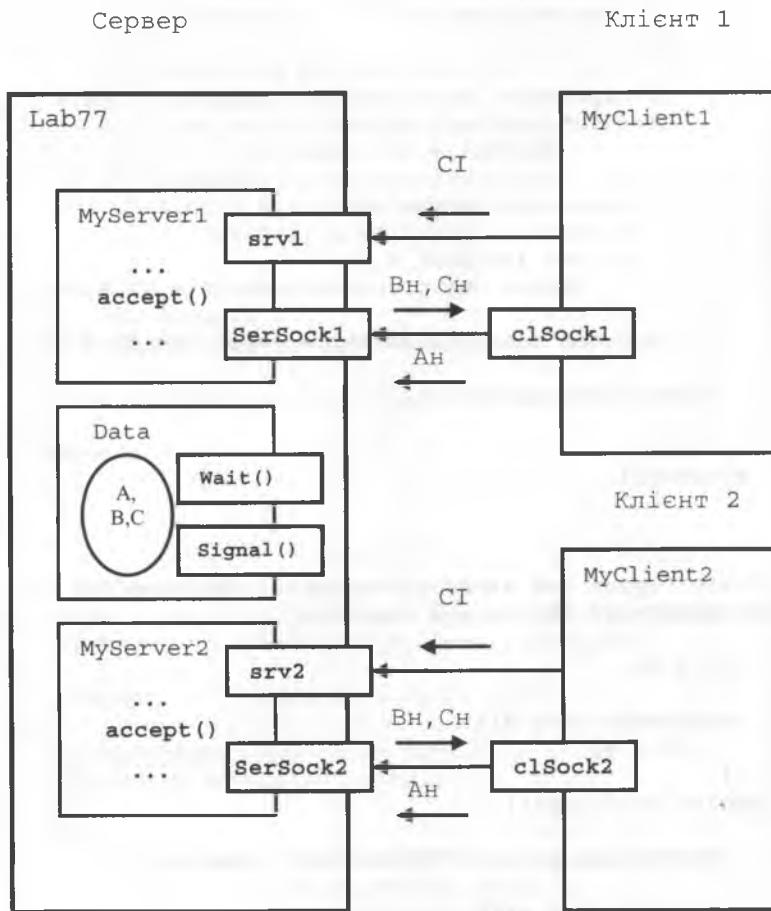


Рис. 7.14. Реалізація моделі клієнт-сервер

```

        // передавання даних першому клієнту
        os1.write(d1.H);

        for(int i=0;i<d1.H;i++){
            os1.write(d1.B[i]);
            os1.write(d1.C[i]);
        }

        // отримання результату від першого клієнта
        for(int i=0;i<d1.H;i++)
            d1.A[i] = is1.read();

        // виведення результату
        d1.Wait();
        for(int i=0;i<d1.N;i++)
            System.out.println(" A = " + d1.A[i]);

        System.out.println("MyServer1 Finished ");

    }catch(IOException e){      }
}

}// MyServer1 -----

```

// Потік Сервер для взаємодії з другим клієнтом

```
class MyServer2 implements Runnable{
```

```

    Data d2;

    MyServer2(Data d){
        d2 = d;
    }
    public void run(){

        System.out.println("MyServer2 started ");

        ServerSocket srv2 = null;
        Socket serSock2 = null;

        OutputStream os2 = null;
        InputStream is2 = null;

        try{
            srv2 = new ServerSocket(78);
            serSock2 = srv2.accept();

```

```
    is2 = serSock2.getInputStream();
    os2 = serSock2.getOutputStream();

    // передавання даних другому клієнту
    os2.write(d2.H);

    for(int i=d2.H;i<d2.N;i++){
        os2.write(d2.B[i]);
        os2.write(d2.C[i]);
    }
    // отримання результату від другого клієнта
    for(int i=d2.H;i<d2.N;i++)
        d2.A[i] =is2.read();

    // синхронізація
    d2.Signal();
    System.out.println("MyServer2 Finished ");

} catch(IOException e) {
}
}// MyServer2 -----
```

```
class MyClient1 implements Runnable{
    public void run(){
        System.out.println("MyClient1 started");

        Socket      clSock1 = null;

        InputStream is      = null;
        OutputStream os11   = null;

        try{
            InetAddress IA;
            IA = InetAddress.getLocalHost();
            clSock1 = new Socket(IA,77);

            is = clSock1.getInputStream();
            int H1 = is.read();

            int [] zA= new int[H1];
            int [] zB= new int[H1];
            int [] zC= new int[H1];
        }
```

```

    // отримання даних
    for(int i=0;i<H1;i++){
        zB[i] = is.read();
        zC[i] = is.read();
    }

    // обчислення
    for(int i=0;i<H1;i++){
        zA[i] = zB[i] + zC[i];
    }
    os11= clSock1.getOutputStream();

    for(int i=0;i<H1;i++){
        os11.write(zA[i]);
    }

    System.out.println("Client1 Finished ");

}catch(UnknownHostException e){ }
catch(IOException e){ }
}
}// MyClient1 -----

```

```

class MyClient2 implements Runnable{

    public void run(){
        System.out.println("MyClient2 started");

        Socket      clSock2   = null;
        InputStream is         = null;
        OutputStream os22     = null;

        try{
            InetAddress IA;
            IA = InetAddress.getLocalHost();

            clSock2 = new Socket(IA,78);
            is = clSock2.getInputStream();

            int H2 = is.read();
            int [] zA = new int[H2];
            int [] zB = new int[H2];
            int [] zC = new int[H2];

            // отримання даних

```

```
for(int i=0;i<H2;i++) {
    zB[i] = is.read();
    zC[i] = is.read();
}

// обчислення
for(int i=0;i<H2;i++) {
    zA[i] = zB[i] + zC[i];
}
os22 = c1Sock2.getOutputStream();

// повернення результату
for(int i=0;i<H2;i++) {
    os22.write(zA[i]);
}

System.out.println("Client2 Finished ");
}catch(UnknownHostException e) {
}catch(IOException e){ }
}

}// MyClient2
```

### 7.3.2. Ада. Віддалені процедури

**Етап 3. Розробка структурної схеми взаємодії розділів.** Структурна схема взаємодії розділів ґрунтуються на алгоритмах задач і дозволяє наочно показати передавання даних між розділами. На структурній схемі (рис. 7.15) формуються розділи і визначається організація взаємодії між ними.

Програма містить два розділи:

- розділ Клієнт, який включає процедуру з двома задачами – T1 і T2;
- розділ Сервер, який включає пакет Box з віддаленою процедурою Додавання\_Векторів.

Розділ Сервер буде розміщений на вузлі 1, розділ Клієнт буде розміщений на вузлах 2 і 3. Це виконується за допомогою формування конфігураційного файлу.

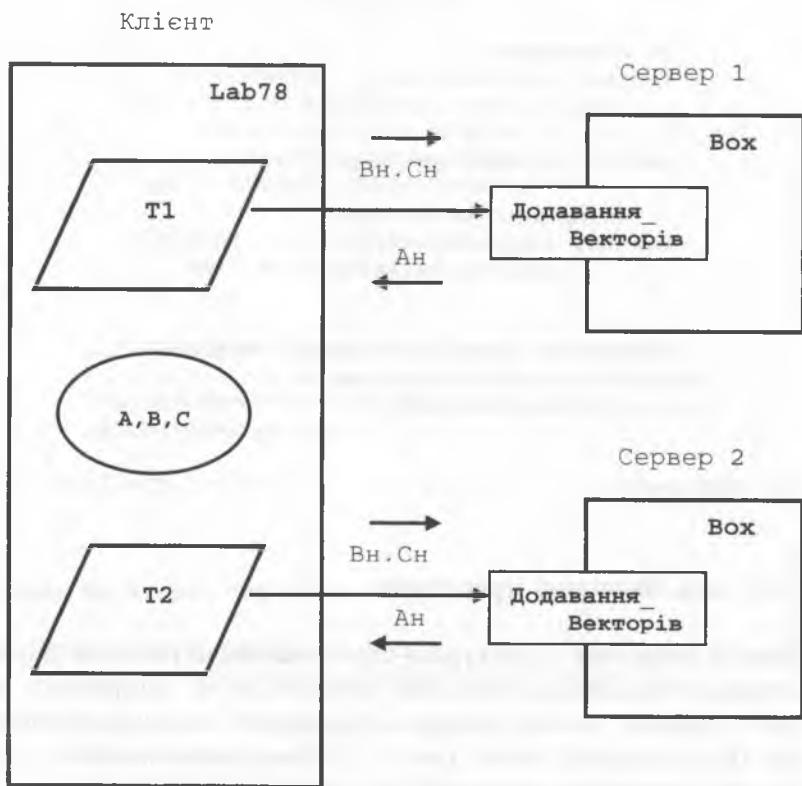


Рис. 7.15. Реалізація взаємодії сервера з клієнтами

**Етап 4. Розроблення програми.** Програми розробляється на підставі алгоритмів сервера і клієнтів і структурної схеми взаємодії задач. Формуючи типи, що використовуються в програмі, слід звернути увагу на оголошення типів, які дозволяють описати протоколи всіх входів.

```
--|Lab78.Ада.Механізм віддалених підпрограм
--|Операція A = B + C
-----
package Box is

    N: integer := 60;      -- розмір векторів
    P: integer := 2;       -- кількість клієнтів
    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 ВекторН is Вектор_Загальний (1..H);

    -- віддалена процедура
    procedure Додавання_Векторів(VB, VC: in ВекторН;
                                    VA: out ВекторН);

    -- прагма категорії
    pragma Remote_Procedure_Call(Додавання_Векторів);

end Box;
-----
package body Box is

    procedure Додавання_Векторів(VB, VC: in ВекторН;
                                    VA: out ВекторН) is
        begin
            for i in 1 .. H loop
                VA(i) := VB(i) + VC(i);
            end loop;
    end Додавання_Векторів;

end Box;
```

```

with Box, Text_IO, Ada.Integer_Text_IO;
use Box, Text_IO, Ada.Integer_Text_IO;
procedure Lab78 is

    A,B,C : Вектор;

    procedure Старт_Клієнта is

        -- описання задач
        task T1;

        task body T1 is
            begin
                put_line("Process T1 started");

                -- передати ВН, СН в Клієнт1,
                -- виконати обчислення
                -- і повернути результат

                Box.Додавання_Векторів(В(1..H),С(1..H),А(1..H));

                put_line("Process T1 finished");
            end T1;

            -----
            task T2;
            task body T2 is
                begin
                    put_line("      Process T2 started");

                    -- передати ВН, СН в Клієнт2,
                    -- виконати обчислення
                    -- і повернути результат
                    Box.Додавання_Векторів(В(H+1..2*H),С(H+1..2*H),
                                         А(H+1..2*H));

                    put_line("Process T2 finished");
            end T2;

            begin
                null;
            end Старт_Клієнта;

        -- тіло основної програми
        begin
            -- Сервер
            put_line("Main procedure started ");
        end;
    end;

```

```
-- уведення векторів В і С
for i in 1.. N loop
    B(i) := 1;
    C(i) := 2;
end loop;

Старт_Клієнта;

-- виведення результату - А
put_line(" A = ");
for i in 1 .. N loop
    put(A(i),3);
end loop;

end Lab78;
```

Файл конфігурації для програми Lab78:

```
configuration MyConfig is

    Partititon_Client: Partition := ();
    Procedure Main is in Partititon_Client;
    Partition_Server1: Partition;
    Partition_Server2: Partition;

    for Partititon_Client'Host use "18.12.00.01"
    for Partititon_Client'Directory use "/DSA/Box;"

    for Partititon_Server1'Host use ""
    for Partititon_Server1'Directory use "/DSA/"

    for Partititon_Server2'Host use ""
    for Partititon_Server2'Directory use "/DSA/"
    for Partition'Filter use "ZIP"

    pragma Starter(Method => Ada);

begin
    Partititon_Server1 := (task1);
    Partititon_Server2 := (task2);
end MyConfig;
```

### ➤ Завдання до самостійної роботи

Завдання до самостійної роботи (лабораторні та розрахунково–графічні роботи, курсові роботи) пов’язані з розробленням програм для виконання векторно–матричної операції в паралельних (розподілених) комп’ютерних системах заданий структури.

Розглядаються такі векторно–матричні операції:

- скалярне множення векторів;
- множення вектора на матрицю;
- множення матриць;
- сортування векторів;
- пошук мінімального (мінімального) елементу вектора.

Виконуючі розрахунково-графічні роботи або курсові роботи треба розглядати програмування для масштабованих КС, де кількість процесорів (вузлів) може змінюватися.

У разі застосування реальних СМП або кластерних систем потрібно виконати дослідження поведінки коефіцієнтів прискорення в залежності від розміру векторів (матриць) та кількості використовуваних процесорів (вузлів) системи.

**Системи зі спільною пам’яттю.** Для систем зі спільною пам’яттю використовують СМП системи, які містять 4–6 процесори, 2–3 пристрої введення-виведення. Використовують мови Java, C#, C, Ада, бібліотеки Win32, Pthread, OpenMP. Засоби взаємодії процесів: атомарні змінні, семафори, мютекси, події, критичні ділянки, монітори, захищені модулі. Виконати розроблення паралельного алгоритму, алгоритмів процесів, структурну схему взаємодії процесів. Розробити та налагодити програму.

**Системи з локальною пам’яттю.** Розглядаються п’ять типів систем з розподіленою пам’яттю (Рис. 1.5)

- лінійні,
- кільцеві,
- зірка,
- решітка,

- гіперкуб,
- повнозвязна.

Кількість процесорів системи і розміщення пристройівуведення-виведення визначається в завданні. Засоби, які використовуються: мови C, C#, Ада; бібліотеки MPI, PVM, Win32. Виконати розроблення паралельного алгоритму, алгоритмів процесів, структурну схему взаємодії процесів. Розробити та налагодити програму.

**Розподілені системи.** Розглядаються розподілені системи різної структури, побудовані на базі кількох робочих станцій і комутаторів. Визначаються вузли, які безпосередньо пов'язані з пристроями введення-виведення.

Мови програмування: Java, С, Ада. Засоби організації взаємодії вузлів: сокети, RMI, RPC. Розробити паралельний алгоритм, алгоритми процесів з використанням моделі клієнт-сервер, структурну схему взаємодії процесів. Розробити та налагодити програму.

У завданні до розрахунково-графічних або курсових робіт потрібно розглядати масштабовані системи, де як вузлі використано двох або чотирьох процесорні СМП КС. Програмуючи клієнтські вузлі, треба враховувати, що обчислення в них слід виконувати як в системі зі спільною пам'яттю, тобто передбачити створення двох-чотирьох процесів для кожного процесору КС у вузлі, а також у разі потреби – вирішення завдань взаємного виключення і синхронізації за допомогою відповідних засобів мов або бібліотек паралельного програмування.



## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Антонов А.С. Параллельное программирование с использованием технологий OpenMP: Учебное пособие".-М.: Изд-во МГУ, 2009. - 77 с.
2. Богачев К.Ю. Основы параллельного программирования. – М.: БИНОМ. Лаб. знаний, 2003. – 342 с .
3. Брайант Р. Компьютерные системы: архитектура и программирование. - BHV-СПб, 2005. - 1186 с.
4. Брайнль Т. Паралельне програмування. Початковий курс: Навч. посіб. – К.: Вища шк, 1997. – 358 с.
5. Валях Е. Последовательно-параллельные вычисления. – М.: Мир, 1985. – 456 с.
6. Воеводин В.В. Математические модели и методы в параллельных процессах. – М.: Наука, 1984. – 296 с.
7. Воеводин В., Воеводин В. Параллельные вычисления. БХВ-Петербург, 2002. 608 стр.
8. Гергель В.П. Теория и практика параллельных вычислений. - Интернет-университет информационных технологий – ИНТУ ИТ.ру, БИНОМ. Лаборатория знаний, 2007. - 424 с.
9. Гергель В. Высокопроизводительные вычисления для много процессорных многоядерных систем. . М.: Изд. МГУ.- 2010. – 544 с.
10. Гома Х. UML. Проектирование систем реального времени, параллельных и распределенных приложений. – М.: ДМК Пресс, 2002. – 704 с.
11. Гофф Макс К. Сетевые распределенные вычисления. Достижения и проблемы. – М.:Кудиц-Образ, 2005. - 320 с.
12. Дейтел Д. Введение в операционные системы. – М.: Мир,1989. – 360 с.
13. Джехани Н. Язык Ада. – М.: Мир, 1988. – 552 с.
14. Джоунз Г. Программирование на языке ОККАМ. – М.: Мир, 1989. – 246 с.
15. Корнеев В.Д. Параллельное программирование в MPI. – Москва-Ижевск: "Институт компьютерных исследований",

2003. - 303 с.
16. Корочкин А.В. Ада95: Введение в программирование. – К.: Світ, 1999. – 260 с.
  17. Линев А., Боголевов Д. Технологии параллельного программирования для процессоров новых архитектур. М.: Изд. МГУ.- 2010. – 160 с.
  18. Лисков Б., Гатэг Дж. Использование абстракций и спецификаций при разработке программ : Пер. с англ. – М.: Мир, 1989. – 424 с.
  19. Лупин С., Посыпкин М. Технологии параллельного программирования. М.: ИД Форум, 2011. - 208 с.
  20. Луцкий Г.М., Кулаков Ю.А. Локальные сети. – К.: Юниор, 1998. – 336 с.
  21. Мак-Дональд М., Шлушта М. Microsoft ASP.NET 2.0 с примерами на C# 2005 для профессионалов.: Пер. с англ. – М.: Изд. дом «Вильямс», 2006. - 1408 с.
  22. Малышкин В.Э., Корнеев В.Д. Параллельное программирование мульткомпьютеров. - НГТУ, 2006. - 296 с.
  23. Миллер Р.,Боксер Л. Последовательные и параллельные алгоритмы: Общий подход. – М.: Лаборатория Базовых Знаний, 2004. – 406 с.
  24. Миренков Н.Н. Параллельное программирование для много модульных вычислительных систем. – М.: Радио и связь, 1989. – 320 с.
  25. Немнюгин С., Стесик О. Параллельное программирование для многопроцессорных вычислительных систем. – СПб.: БХВ – Петербург, 2002. – 400 с.
  26. Немнюгин С. А. Модели и средства программирования для многопроцессорных вычислительных систем. С.Петербургский ГУ, 2010. - 150 с.
  27. Ноутон П., Шилдт Г. Java2: Пер. с англ. – СПб.: БХВ – С. Петербург, 2000. – 1072 с.
  28. Ортега Дж. Введение в параллельные и векторные методы решения линейных систем. – М.: Радио и связь, 1989, – 280 с.
  29. Параллельные вычисления / Под ред. Г.Родрига – М.: Наука, 1986. – 376 с.
  30. Пайл Я. Ада – язык встроенных систем. – М.; Финансы и статистика, 1984. –120 с.

31. Перминов О.Н. Введение в язык программирования Ада. – М.: Радио и связь, 1991. – 228 с.
32. Программирование на параллельных вычислительных системах/ Пер. с англ./ Р.Бэбб, Дж. Мак – Гроу и др.; под ред.Бэбба II. – М.: Мир, 1991. – 376 с.
33. Русанова О.В. Программное обеспечение компьютерных систем. Особенности программирования и компиляции. – К.: Корнійчук, 2003. – 94 с.
34. Симкин С., Барлетт Н., Лесли А. Программирование на Java. Путеводитель – К.: НИПФ “ДиаСофт Лтд.”, 1996. – 736 с.
35. Симоненко В.П. Организация вычислительных процессов в ЭВМ, комплексах, сетях и системах.–К.: ВЕК+, 1997.– 304 с.
36. Соловьев Г.Н., Никитин В.Д. Операционные системы ЭВМ. – М.: Высш. школа., 1989. – 255 с.
37. Системы параллельной обработки: Пер. с англ./Под ред. Д.Ивенса. – М: Мир, 1985. – 416 с.
38. Стіренко С. Г., Грибенюк Д. В., Зіценко А. І., Михайленко А. В. Засоби паралельного програмування. - К., 2011. - 181 с.
39. Траспьютеры.Архитектура и программное обеспечение: Пер. с англ./Под ред.Г.Харпа. – М.: Радио и связь, 1993. – 304 с.
40. Троелсон Є. С# и платформа.NET. Библиотека программиста - СПб.: Питер, 2004. – 796 с.
41. Уильямс Э. Параллельное программирование на C++ в действии. ДМК, - 2012. - 672 с.
42. Хьюз К, Хьюз Т. Параллельное и распределенное программирование в C++. Пер. с англ.- М.: Радио и связь, 1986. – 240 с.
43. Хоар Ч. Взаимодействующие последовательные процессы. - М.: Мир, 1989. – 180 с.
44. Хокни Р., Джессхоул К. Параллельные ЭВМ. Пер. с англ.- М.: Радио и связь, 1986. – 240 с.
45. Шилд Г. Полный справочник по C#.: Пер. с англ. – М.: Изд. дом «Вильямс», 2007. – 752 с.
46. Шамим Э., Джейсон Р. Многоядерное программирование Питер, 2010. - 180 с.
47. Эндрюс Г. Основы многопоточного, параллельного и распределенного программирования.: Пер. с англ. – М.: Изд. дом «Вильямс», 2003. – 512 с.

48. Элементы параллельного программирования / В.А.Вальковский , В.Е.Котов, А.Г.Марчук / Под ред. В.Е.Котова – М.: Радио и связь, 1983. – 240 с.
49. Языки программирования: Ада, Паскаль, Си. Сравнение и оценка / Под ред.Н. Джехани – М.: Радио и связь, 1989. – 386 с.
50. Breshears C. The Art of Concurrency. O'Reilly Media, 2009. – 280 p.
51. Burns A., Wellings A. Real-Time Systems and Programming Languages. Addison – Wesley, 2001, – 386 p.
52. Burns A., Programming in Occam2. Reading. Addison-Wesley, 1995, – 326 p.
53. Burns A., Wellings A. Concurrency in Ada. – Cambridge: Cambridge University Press, 1995., – 420 p.
54. El – Rewini H, Lewis T., Distributed and Parallel Computing. – Manning Pub. Co.1998, – 430 p.
55. Hyde P. Java Thread Programming. – Indianapolis, IN: Sams Publishing, 1999, – 324 p.
56. Hoare C.A.R. Monitors: An Operating System Structuring Concept, Communications of ACM, Vol.17, №.10, Oct.1974, pp. 549–557
57. Hoare C.A.R. Communicating Sequential Processes, Communications of ACM, vol.21, №. 8, Aug. 1978, pp. 666 – 667.
58. Hoare C.A.R. Communicating Sequential Processes.; Printice Hall, International Series in Computer Science, Englewood Cliffs NJ, 1985. – 186 p.
59. Goetz B. Java Concurrency in Practice.- Addison-Wesley Professional, 2006, – 384 p.
60. Korochkin A., Rusanova O. Scheduling Problems for Parallel and Distributed Systems– In Proceeding of the ACM Annual Conference (SIGADA'99) (The Redondo Beach, CA, USA, October 17–21, 2001) ACM Press, New York, NY, 1999, pp. 182– 190;
61. 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.
62. Korochkin A., Salah I., Korochkin D. Experimental Analyze Ada Program in Cluster System – In Proceeding of the ACM Annual Conference (SIGADA'05) (Atalanta, Georgia, USA, November 13–21, 2005) ACM Press, New York, NY, 2005, pp. 126 – 134.
63. Lea D. Concurrent Programming in Java: Design Principles and Patterns. Reading, MA: Addison – Wesley, 1999, – 344 p.
64. Mattson T., Sanders B., Massingill B. Patterns For Parallel Programming. Addison-Wesley, 2005. – 246 p.
65. Taft S., Duff R., Brukardt R., Ploedereder T. Consolidated Ada Reference Manual. Language and Standard Libraries, Springer, Berlin: 2001, – 562 p.
66. Chapman B. Using OpenMP: portable shared memory parallel programming. Massachusetts Institute of Technology, London, 2008, - 350 p.
67. Butenhof D. Programming with POSIX Threads. Addison-Wesley, 1997. – 300 p.
68. Pacheco P. An Introduction to Parallel programming. Elsevier Inc., Amsterdam , 2011, – 310 p.
69. Ros A. Parallel and Distributed Computing. IN-TECH.-2010, - 290 p.
70. McKenney P. Is Parallel Programming Hard, And, If So, What Can You Do About It? [Електронний ресурс]. Режим доступу: [tps://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/](http://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/)
71. Downey A. The Little Book of Semaphores. ? [Електронний ресурс]. Режим доступу: <http://freecomputerbooks.com/The-Little-Book-of-Semaphores.html>
72. TOP500. [ Електронний ресурс ]. Режим доступу: [http://parallel.ru/computers/ top500.list42.html](http://parallel.ru/computers/top500.list42.html)
73. Intel Developer Zone [Електронний ресурс]. Режим доступу: <http://software.intel.com/ru-ru/articles/more-work-sharing-with-openmp>
74. Параллельные методы матричного умножения [Електронний ресурс]. Режим доступу: [www.hpc.unn.ru/file.php?id=426](http://www.hpc.unn.ru/file.php?id=426)
75. Офіційний сайт компанії Інтел. [Електронний ресурс]. Режим доступу: [www.intel.com](http://www.intel.com)
76. Офіційний сайт компанії AMD. [Електронний реурс]. Режим доступу: [www.amd.com](http://www.amd.com)

77. Форум MPI. [Електронний реурс]. Режим доступу:  
<http://www mpi-forum.org/docs/docs.html>
78. Who's Using Ada? Real-World Projects Powered by the Ada Programming Language. [Електронний ресурс]. Режим доступу: <http://www.seas.gwu.edu/~mfeldman/ada-project-summary.html>
79. The Special Interest Group on Ada (ACM's SIGAda). [Електронний ресурс]ю Режим доступу:  
<http://www.sigada.org/>
- 80 Офіційний сайт організації Ada-Europe. [Електронний ресурс], Режим доступу: <http://www.ada-europe.org/>
81. Офіційний сайт проекту GAP. [Електронний ресурс]. Режим доступу:  
<http://www.adacore.com/academia/universities/>
82. Офіційний сайт OpenMP. [Електронний ресурс], Режим доступу: <http://openmp.org>



## ІНТЕРНЕТ-РЕСУРСИ

---

У цьому розділі наведені дані про Інтернет-ресурси, за допомогою яких можна знайти інформацію, що пов'язана з паралельними і розподіленими обчисленнями, а також знайти доступне (вільне) програмне забезпечення, яке можна використати для розробки програм для паралельних і розподілених комп'ютерних систем.

1. Система програмування *ObjectAda* (Atego):

<http://www.atego.com/products/atego-objectada/>

2. Система програмування *GNATAda* (AdaCore Technology):

<http://www.adacore.com>  
<ftp://cs.nyu.edu/pub/gnat>  
<http://www.gnat.com>

3. Реалізація DSA - *GLADE* (AdaCore Technology):

<http://www.libre.adacora.com>

4. MPI-ресурси:

<ftp://info.mcs.anl.gov/pub/mpi>  
<ftp://ftp.epcc.ed.ac.uk/pub/chimp/realease>  
<http://www.mpi.org>  
<http://www.mpi-forum.org>  
<http://www.lam-mpi.org>

5. PVM-ресурси:

<http://www.epm.ornl.gov/pvm/>  
<http://www.netlib.org/pvm3/>

6. Java-ресурси:

<http://sun.com>

7. Ада - ресурси:

<http://www.adaword.com>  
<http://www.ada-europe.org>

<http://www.adapower.com>  
<http://www.adacora.com>  
<http://www.acm.org/ada>

**8. OpenMP - ресурси:**

<http://www.adaword.com>  
<http://www.ada-europe.org>  
<http://www.adapower.com>  
<http://www.act-europe.com>  
<http://www.acm.org/ada>

**9. Паралельні та розподілені обчислення:**

<http://www.paralel.ru>  
<http://www.osp.ru>  
<http://www.beowulf.org>  
<http://www.topcluster.org>  
<http://www.csa.oru>  
<http://www.top500.org>  
<http://www.OpenMP.org>



## ПРЕДМЕТНИЙ ПОКАЖЧИК

### A

Ada, мова 35

Atomic, прагма 85

### L

Link, зв'язок, 153

### C

CORBA, технологія, 181

Critical Section,  
критична секція, 117

C#, мова, 42

### M

Mutex, засіб, 99, 119

Message-passing, інтерфейс 46

MPI, бібліотека 30

### D

DCOM, технологія, 181

DSA, додаток, 195

Deadlock, 29

### N

NET, технологія, 283

### E

Entry, вхід, 157

Event, подія, 132

### O

Occam, мова, 153

OpenMP, бібліотека, 50

### G

GLADE, система програмування, 200

GNAT, система програмування, 200

### P

Partition, розділ, 195

POSIX, ОС, 30

Pthreads, бібліотека, 30

PVM, бібліотека, 234

P(), операція, 91

PCB, блок, 26

### R

Rendezvous, механізм, 159

RPC, віддалений виклик

процедури, 195

RMI, пакет, 192

### I

Infinityband, технологія, 23

### J

Java, мова, 31

### S

SMP, симетрична

мультипроцесорна система, 14

### T

Task, задача, 36

Thread, потік, 31

Tid, ідентифікатор задачі, 48

### V

Volatile, прагма, 85

V(), операція, 91

### W

Win32, бібліотека, 39

### A

Алгоритм паралельний, 58

### B

Багатоядерні системи, 16

Бар'єр, 147

Блокування, 29

Брент, лема 60

Блок керування процесу, 26

### B

Віддалений інтерфейс, 192

Ввіддалений об'єкт, 135

Віддалена процедура, 136

Віддалений метод, 144

Взаємне виключення, 79

Взаємодія процесів, 78

Вхід, оператор 157

ВХІДКД, приймитів 80

ВИХІДКД, приймитів 80

### G

Гіперкуб, структура 21

### D

Декомпозиція, 72

Дідлок, тупик, 21

Діаметр системи, 20

Дискримінант, 38

### Z

Завдання взаємного

виключення, 79

Завдання синхронізації, 82

Задача, 36

Задачний тип, 37

Замок, 101

Захищений модуль, 65

Захищена операція, 65  
 Захищена функція, 104  
 Захищена процедура, 104  
 Захищений вхід, 104  
 Зірка, структура 21  
 Змінні, що поділяються, 82

**I**

Інтерфейс віддалений, 192

**K**

Канал, 153  
 Кеш-пам'ять, 16  
 Кільцева структура, 21  
 Кластерні системи, 23  
 Комунікатор, 47  
 Критична ділянка, 79  
 Критична секція, 100

**L**

Лінк, 153  
 Лінійна структура, 21

**M**

Методи, 31  
 wait()  
 notify()  
 notifyAll()  
 sleep()  
 suspend()  
 resume()  
 yield()  
 join()

Мова  
 Ада, 35  
 Java, 31  
 Оккам, 153  
 C#, 42

Модель взаємодії  
 omp parallel 51

через спільні змінні, 78  
 через посилання повідомлень, 78  
 Модель клієнт-сервер, 135  
 Монітор, 102  
 Мунро-Петerson, теорема, 60  
 Мютекс, 99

**O**

Обчислення  
 паралельні, 204  
 розподілені, 243  
 Оккам, мова, 153  
 Оператор  
 accept, 157  
 delay, 28  
 requeue, 157  
 select 157  
 Операція  
 Send, 170  
 Receive, 170

**P**

Пам'ять  
 розподілена, 22  
 спільна, 13  
 поділяєма, 13  
 Паралельна програма, 22  
 Передача повідомлення, 152  
 Потік, 20, 22  
 Подія, 132  
 Повідомлення, 78  
 Повнозв'язна система, 21  
 Прагма синхронізації, 147  
 Протокол, 110  
 Процес, 26  
 Прагма  
 Atomic, 85  
 Volatile, 85

omp sections 53

## Р

Рандеву, механізм, 113  
Розділ, 148  
Розподільна змінна, 82  
Розподілена програма, 134  
Ресурс (поділяємий), 51  
Решітка, структура, 21

## С

Семафор, 91  
Синхронізація, 78  
Синхронізовані блоки, 100  
Синхронізовані методи, 100  
Сокет серверний, 245  
Сокет, 187  
Спільній ресурс, 79  
Стан процесу, 26

## Т

Точка синхронізації,  
 $W_{ij}$  206  
 $S_{ij}$  206  
Трансп'ютер, 157  
Тупик, 29  
Топологія системи, 20

## Ф

Функції очікування, 96

## Х

Хоар, концепція, 102

## Я

Ядро, 16  
Ярусно-паралельна форма, 61



## ДОДАТКИ

- Додаток А. Мова Java
- Додаток Б. Мова Ada
- Додаток В. Бібліотека WinAPI (Win32)
- Додаток Г. Бібліотека MPI
- Додаток Д. Бібліотека PVM
- Додаток Е. Мова C# і платформа .NET
- Додаток Ж. Бібліотека OpenMP

### Додаток А

#### Класи Java.

##### A.1. Клас Thread

```

package java.lang;
public class Thread extends Object
    implements Runnable {

    // Поля
    static int MIN_PRIORITY = 1;
    static int NORM_PRIORITY = 5;
    static int MAX_PRIORITY = 10;

    // Конструктори
    public Thread();

    public Thread(Runnable target);

    public Thread(Runnable target, String name);

    public Thread(String name);

    public Thread(ThreadGroup group, String name)
        throws SecurityException, IllegalThreadStateException;
}

```

```
public Thread(ThreadGroup group, Runnable target)
    throws SecurityException, IllegalThreadStateException;

public Thread(ThreadGroup group, Runnable target,
               String name)
    throws SecurityException, IllegalThreadStateException;

// Методи
static int activeCount();

void checkAccess()
    throws SecurityException;

int countStackFrames()
    throws IllegalThreadStateException;

public static Thread currentThread();

public void destroy()
    throws SecurityException;

static void dumpStack();

static int enumerate(Thread [] tarrarrey)
    throws SecurityException;

ClassLoader getContextClassLoader()
    throws SecurityException;

String getName();

int getPriority();

ThreadGroup getTresdGroup();

public void interrupt() throws SecurityException;

static boolean interrupted();

public final boolean isAlive();

public final boolean isDaemon();

boolean isInterrupted();
```

```

public final void join()
                      throws InterruptedException;

public final void join(long millis)
                      throws InterruptedException;

public final void join(long millis, int nanos)
                      throws InterruptedException;

private native void resume()
                      throws SecurityException; // DEPRACTED

public void run();

public void setContextClassLoader(ClassLoader cl)
                      throws SecurityException;

public final void setDaemon() throws
                      SecurityException, IllegalThreadStateException;

public void setName(String name)
                      throws SecurityException;

private native void setPriority(int newPriority)
                      throws IllegalArgumentException,
                           InterruptedException;

public static void sleep(long millis)
                      throws InterruptedException;

public static void sleep(long millis, int nanos)
                      throws IllegalArgumentException,
                           InterruptedException;

public synchronized void start()
                      throws IllegalThreadStateException;

private final void stop()
                      throws SecurityException; // виключений

private final void stop(Throwable t) // виключений
                      throws SecurityException,
                           NullPointerException;

private native void suspend()
                      throws SecurityException; // виключе-

```

```
private static void yield();

public String toString();

public void interrupt() throws SecurityException;
}
```

## A.2. Клас ThreadGroup

```
package java.lang;
public class ThreadGroup{

    // Конструктори
    public class ThreadGroup(String name)
        throws SecurityException;
    public class ThreadGroup(ThreadGroup parent,
                           String name) throws SecurityException;

    // Методи
    public int activeCount();

    public int activeGroupCount();

    public boolean allowThreadSuspension(boolean a);

    private final void checkAccess()
        throws SecurityException;

    private final void destroy()
        throws IllegalThreadStateException,
               SecurityException;

    public int enumerate(Thread [] list)
        throws SecurityException;

    public int enumerate(Thread [] list, boolean b)
        throws SecurityException;

    public int enumerate(ThreadGroup [] list)
        throws SecurityException;

    public int enumerate(ThreadGroup [] list, boolean b)
        throws SecurityException;

    public int getMaxPrioritet();

    public String getName();
```

```

public final ThreadGroup getParent()
    throws SecurityException;

public final boolean isDaemon();

public synchronized boolean isDestroyed();

public void list();

public final boolean parentOf(ThreadGroup d)();

public void resume(); // виключений

public final void setDaemon(boolean daemon)
    throws SecurityException;

public void setMaxPriority(int pr)
    throws SecurityException;

private final void stop()
    throws SecurityException; // виключений

private native void suspend()
    throws SecurityException; // виключений

public String toString();

public void uncaughtException(Thread t, Throwable a);

}

```

### A.3. Інтерфейс Runnable

```

package java.lang;
public interface Runnable {
    public abstract void run();
}

```

## Додаток Б.

### Специфікації пакетів мови Ada

#### **Б.1. Пакет Synchronous\_Task\_Control**

```

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

    ...
    --    не визначено в мові

end Ada.Synchronous_Task_Control;

```

#### **Б.2. Пакет Asynchronous\_Task\_Control**

```

with      Ada.Task_Identification;
package Ada.Asynchronous_Task_Control is

    procedure Hold(T : in
                  Ada.Task_Identification.Task_ID);

    procedure Continue(T : in
                      Ada.Task_Identification.Task_ID);

    function Is_Held(T:
                      Ada.Task_Identification.Task_ID)
                    return Boolean;

end Ada.Asynchronous_Task_Control;

```

**Додаток В****Бібліотека WinAPI (Win32).****В.1. Функції очікування**

```

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, сигнал
    // якого очікується
    DWORD dwMilliseconds // тайм-аут у мілісекундах
);

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

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 на 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 на одного або всіх
    DWORD dwMilliseconds, // тайм-аут
    DWORD dwWakeMask     // тип події уведення
                           // для очікування
);
;

DWORD MsgWaitForMultipleObjectsEx(
    DWORD nCount,        //кількість об'єктів в масиві
    LPHANDLE pHandles,   // вказівник на масив
                           // ідентифікаторів
    DWORD dwMilliseconds, // тайм-аут
    DWORD dwWakeMask,    // тип події уведення
                           // для очікування
    DWORD dwFlags        // прапор очікування
);
;

```

## B.2. Функції для роботи із сокетами

Функція	Дія
accept()	Прийняти з'єднання по сонету
bind()	Присвоїти локальне ім'я сонету
closesocket()	Закрити сонет
connect()	Ініціювати з'єднання через визначений сонет
getpeername()	Отримати ім'я порту, який з'єднається з сонетом
getsockname()	Отримати поточне ім'я для специфікованого сонету
recv()	Отримати дані від підключенного сокету
recvfrom()	Отримати дані від будь-якого сокету
send()	Послати дані в підключений сокет
sendto()	Послати дані в будь-який сонет

## Додаток Г

### Бібліотека MPI

#### Г.1. Базові функції

Функція	Дія
<code>MPI_Init()</code>	Ініціалізація MPI
<code>MPI_Finalize()</code>	Завершення MPI
<code>MPI_Comm_size()</code>	Визначення розміру області взаємодії
<code>MPI_Comm_rank()</code>	Визначення номера процесу
<code>MPI_Send()</code>	Послати повідомлення (блоковане)
<code>MPI_Recv()</code>	Прийняти повідомлення (блоковане)
<code>MPI_Irecv()</code>	Прийняти повідомлення (не блоковане)
<code>MPI_Ssend()</code>	Послати повідомлення (блоковане, синхронізоване)
<code>MPI_Isend()</code>	Послати повідомлення (не блоковане, стандартне)
<code>MPI_Issend()</code>	Послати повідомлення (не блоковане, синхронізоване)
<code>MPI_Bsend()</code>	Послати повідомлення (буферизоване)
<code>MPI_Ibsend()</code>	Послати повідомлення (не блоковане, буферизоване)
<code>MPI_Irsend()</code>	Послати повідомлення не блоковане, в міру готовності)
<code>MPI_Sendrecv()</code>	Блоковане приймання і посилання
<code>MPI_Sendrecv_replace()</code>	Посилка і приймання в блокованому режимі
<code>MPI_Buffer_attach()</code>	Створення буфера
<code>MPI_Buffer_detach()</code>	Відключення буфера
<code>MPI_Rsend()</code>	Послати повідомлення (в міру готовності)
<code>MPI_Probe()</code>	Перевірити доставляння повідомлення (блоковане)
<code>MPI_Iprobe()</code>	Перевірити доставляння повідомлення (не блоковане)
<code>MPI_Wait()</code>	Блокування процесу до завершення приймання (передавання) повідомлення
Функція	Дія

MPI_Test()	Неблокована перевірка завершення доставлення або приймання повідомлення
MPI_Waitall()	Перевірити завершення всіх обмінів повідомлення (блоковане)
MPI_Testall()	Перевірити завершення всіх обмінів повідомлення (не блоковане)
MPI_Waitany()	Перевірити завершення заданої кількості обмінів повідомлення (блоковане)
MPI_Testany()	Перевірити завершення будь-якого ініціалізованого обміну
MPI_Waitsome()	Визначити кількість завершених обмінів
MPI_Bcast()	Послати повідомлення іншим процесам у групі
MPI_Barrier()	Синхронізація процесів

## Г.2. Типи даних

<i>Ідентифікатор типу</i>	<i>Tip</i>
MPI_CHAR	Символьний
MPI_SHORT	Короткий цілий
MPI_INT	Цілий
MPI_LONG	Довгий цілий
MPI_UNSIGNED_CHAR	Символьний
MPI_UNSIGNED_SHORT	Короткий цілий
MPI_UNSIGNED_INT	Цілий
MPI_UNSIGNED_LONG	Довгий цілий
MPI_FLOAT	З плаваючою точкою
MPI_DOUBLE	Подвійна точність
MPI_LONG_DOUBLE	Довгий двійний цілий
MPI_BYTE	Байтів

## Додаток Д

## Бібліотека PVM

## Д.1. Базові функції

Ідентифікатор Функції	Функція
pvm_mytid()	Визначення ідентифікатора задачі
pvm_gettid()	Отримання ідентифікатора задачі в групі
pvm_spawn()	Запуск вказаної кількості задач (процесів)
pvm_exit()	Завершення роботи процесу
pvm_send()	Передавання повідомлення з активного буфера
pvm_sendsig()	Посилання сигналу указаної задачі
pvm_psend()	Пакування і передача масиву даних за-значеної задачі
pvm_initrecv()	Створення буфера передавання
pvm_recv()	Блоковане приймання повідомлення від зазначененої задачі
pvm_reduce()	Приведення в групі
pvm_mcast()	Групове розсилання
pvm_barrier()	Синхронізація групова
pvm_mkbuf()	Створення порожнього буфера передавання з визначенням для нього типу кодування
pvm_bcast()	Широке розсилання (для кількох задач)
pvm_config()	Інформація про віртуальну машину
pvm_precv()	Об'єднання блокованого приймання і розпакування
pvm_nrecv()	Неблоковане приймання від зазначеного процесу
pvm_bufinfo()	Інформація про буфер повідомлення
pvm_kill()	Завершення зазначененої задачі
pvm_tasks()	Інформація про задачі, яки виконуються в PVM
pvm_pstat()	Інформація про задачі з вказаним ідентифікатором
pvm_pkbyte() / pvm_upkbyte()	Пакування/Розпакування для типу byte (див. додаток Д.3)

## Додаток Е

### Мова C# і платформа .NET

#### E.1. Базові члени класу AppDomain

Ідентифікатор функції	Функція
CreateDomain()	Створення нового домену додатка в поточному процесі
GetCurreaantThreadId()	Визначення ідентифікатора поточного потоку
Unload()	Вивантаження вказаного домену Додатка
CreateInstance()	Створення екземпляра вказаного типу
ExecuteAssembly()	Виконання збирання
GetAssembly	Повернення списку збирання
Load()	Завантаження зборки в домен додатка

#### E.2. Базові типи простору імен System.Threading

Тип	Призначення
Interlocked	Синхронізований доступ до спільних даних
Monitor	Синхронізація потоків за допомогою блокування і керування очікуванням
Mutex	Синхронізація процесів
Thread	Потік, який виконується у середовищі .NET
ThreadPool	Керування набором потоків, які взаємно пов'язані
WaitHandle	Об'єкти, які дозволяють багаторазове очікування
ThreadStart	Делегат, який посилається на метод, що має бути виконаний перед запуском потоку

### E.3. Базові члени класу Thread

Ідентифікатор функції	Функція
CurrentThread()	Посилання на потік, який виконується
GetData() / SetData()	Повернення/встановлення значення для вказаного слота в поточному потоці
GetDomain() / GetDomainID()	Посилання на домен додатка (або ідентифікатор домену), де виконується потік
Sleep()	Призупинити поточний потік на вказаний час
IsAlive()	Перевірити стан потоку
IsBackground()	Отримання (встановлення) значення для фонового потоку
Name()	Установлення дружнього текстового імені потоку
Priority()	Отримання (встановлення) пріоритету потоку
ThreadState()	Інформація про стан потоку
Interrupt()	Переривання поточного потоку
Join()	Очікування на появу нового потоку
Resume()	Продовжити роботу потоку після призупинення
Suspend()	Призупинити потік
Start()	Виконання потоку

## Додаток Ж

### Бібліотека OpenMP

Загальна форма директиви компілятору OpenMP (прагми):

#pragma omp директива <> опція [ [ [,] опція] . . . ]

#### Ж.1. Базові прагми

Ідентифікатор прагми	Дія
Parallel	Задає паралельну ділянку програми для виділення блоку коду, яка використається потім для побудові паралельного фрагменту програми шляхом її копіювання
Sections	Описує структурування блоку коду в прагми parallel через створення секцій, які описує прагма section
Section	Описує блок коду, який виконується один раз одним потоком
Task	Визначає окрему незалежну задачу
Taskwait	Гарантує завершення в точці старту у всіх задач, що запущені
Master	Опис ділянки коду для виконання лише ниттю майстер
Schedule	Задає розподіл ітерацій циклу між нитями через вид алгоритму планування, числовий параметр (розмір блоку простору ітерацій)
For	Для розподілу ітерацій циклу між нитями
Taskyield	Блокування текучої задачі на користь інших задач з більшим пріоритетом
Single	Опис ділянки колу, яка буде виконуватися лише один раз
Critical	Опис критичної ділянки, яку дозволяється виконувати лише одному потоку
barrier	Засіб для синхронізації потоків. Потоки блокується на бар'єрі, поки всі воїни не досягнуть його
Atomic	Опис спільніх змінних з метою забезпечення синхронізованого доступу

## Ж.2. Додаткові функції

Функція	Дія
<code>set_num_threads(int num)</code>	Визначення кількості нитей в паралельної ділянці через змінну середовища <code>omp num threads</code>
<code>omp_set_dynamic(int num)</code>	Зміна змінної <code>omp_nested</code> для динамічної зміни кількості нитей
<code>omp_get_dynamic(void)</code>	Отримання значення змінної <code>omp nested</code>
<code>omp_get_max_threads(void)</code>	Отримання максимально можливої кількості нитей для використання у наступної паралельної ділянки
<code>omp_get_num_procs(void)</code>	Керування вкладеними паралельними ділянками за допомогою змінної середовища <code>omp nested</code>
<code>omp_set_nested(int nested)</code>	Дозволяє або вкладений паралелізм
<code>omp_get_nested(void)</code>	Отримати значення змінної <code>omp nested</code>
<code>omp_in_parallel(void)</code>	Повертає 1, якщо є викликано з паралельної ділянки програми
<code>omp_get_wtime(void)</code>	Повертає значення часу, що затрачено