Жуков І.А., Корочкін О.В.

Паралельні та розподілені обчислення

PVM•MPI•Java•Ada95•Win32•POSIX•PThreads

«Корнійчук»
Київ 2005

Міністерство освіти і науки України

Національний технічний університет
„Київський політехнічний інститут"
Національній авіаційний університет

І.А.Жуков, О.В.Корочкін

ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ

Рекомендовано Міністерством освіти і науки України
як навчальний посібник для студентів вищих навчальних
закладів, що навчаються за напрямом
0915 „Комп'ютерна інженерія"

Київ
„Корнійчук"
2005

УДК 681.3.06
ББК 32.973
Ж48

Рекомендовано Міністерством освіти і науки України як навчальний посібник для студентів вищих навчальних закладів (лист від 04.07.05 № 14/18.2– 1540).

Рецензент: В.П.Тарасенко, доктор технічних наук, професор, завідувач кафедри спеціалізованих комп'ютерних систем Національного технічного університету України «Київський політехнічний інститут»

Жуков І.А., Корочкін О.В.
Ж48 Паралельні та розподілені обчислення.Навч. посіб. – К.: «Корнійчук» , 2005. – 226 с.
ISBN 966-7599-36-1

Подано теоретичний і практичний матеріал з вивчення засобів програмування для паралельних та розподілених комп'ютерних систем. Детально розглянуто засоби програмування, що ґрунтуються на моделях спільних змінних та передавання повідомлень. Значну увагу приділено вивченню підходів до вирішення завдання взаємного виключення та синхронізації процесів за допомогою механізмів семафорів, мютексів, критичних секцій, моніторів.

Наведено багато прикладів програмування з використанням мов Java, Ада95, C, бібліотек PVM, MPI, Win32.

Може бути використаний під час вивчення дисциплін "Програмування", "Паралельні та розподілені обчислення", "Високопродуктивні обчислювальні системи", "Системи реального часу", «Програмне забезпечення комп'ютерних систем» студентами, які навчаються за напрямом 0915 «Комп'ютерна інженерія».

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

ББК 32.973
ISBN 966-7599-36-1
© Жуков І.А., Корочкін О.В., 2005

ЗМІСТ

ВСТУП 7

1. Структури комп'ютерних систем 10
1.1. Паралельні комп'ютерні системи 10
1.2. Комп'ютерні системи з розподільною (спільною) пам'яттю 11
1.3. Комп'ютерні системи з розподіленою (локальною) пам'яттю 13
1.4. Розподілені комп'ютерні системи 16
Запитання для модульного контролю 17
Завдання для самостійної роботи 17

2. Процеси 19
2.1. Поняття процесу 19
2.2. Стан процесу. Керування процесами. 20
2.3. Процеси в мові Java. 22
2.4. Процеси в мові Ада 27
2.5. Процеси в бібліотеках PVM і MPI. 30
2.6. Процеси в бібліотеці Win32 35
2.7. Паралельні алгоритми 37
2.7.1. Аналіз паралельного алгоритму 37
2.7.2. Розроблення паралельного алгоритму 45
Запитання для модульного контролю 48
Завдання для самостійної роботи 49

3. Взаємодія процесів, що грунтується на спільних змінних 50
3.1. Взаємодія процесів 50
3.1.1. Завдання взаємного виключення 51
3.1.2. Завдання синхронізації процесів 54

3.2. Семафори 54
3.2.1 Семафори в мові Ада 56
3.2.2 Семафори в бібліотеці Win32 58
3.3. Монітори 63
3.3.1. Монітори в мові Ада95 65
3.3.2. Монітори в мові Java 71
3.4. Вирішення завдання взаємного виключення 73
3.4.1. Вирішення завдання взаємного виключення в мові Ада95 73
3.4.2. Вирішення завдання взаємного виключення в Win32 78
3.4.3. Вирішення завдання взаємного виключення в мові Java 85
3.5. Вирішення завданя синхронізації 88
3.5.1. Вирішення завдання синхронізації в мові Ада . . 88
3.5.2. Вирішення завдання синхронізації процесів в бібліотеці Win32 91
3.5.3. Вирішення завдання синхронізації процесів в мові Java 102
Запитання для модульного контролю 105
Завдання для самостійної роботи 105

4. Взаємодія процесів, що грунтується посиланні повідомлень 107
4.1. Загальна схема 107
4.2. Мова Оккам 108
4.3. Ада95.Механізм рандеву 112
4.4. Бібліотека PVM 117
4.5. Бібліотека MPI 123
Запитання для модульного контролю 132
Завдання для самостійної роботи 132

5. Розподілені обчислення 134
5.1. Організація розподілених обчислень 134
5.1.1. Сокети 135
5.1.2. Віддалені процедури 136
5.2. Java.Сокети 140
5.3. Java.RMI 144

5.4. Ада95.RPC 147
Запитання для модульного контролю 154
Завдання для самостійної роботи 155

6. Приклади програмування 156
6.1. Програмування для комп'ютерних систем зі спільною пам'яттю 156
6.1.1. Ада95.Семафори 156
6.1.2. Win32.Семафори і мютекси 162
6.1.3. Ада95.Захищені модулі 165
6.1.4. Java.Монітори 170
6.2. Програмування для комп'ютерних систем з локальною пам'яттю 175
6.2.1. Ада95.Рандеву 175
6.2.2. Бібліотека PVM 181
6.3. Програмування для розподілених комп'ютерних систем 185
6.3.1. Java.Сокети 187
6.3.2. Ада95.Віддалені процедури 193
Завдання для самостійної роботи 198

СПИСОК ЛІТЕРАТУРИ 200

ІНТЕРНЕТ- РЕСУРСИ 205

ПРЕДМЕТНИЙ ПОКАЖЧИК 207

ДОДАТКИ 210
А. Мова Java 210
А.1. Клас Thread 210
А.2. Клас ThreadGroup 213
А.3. Інтерфейс Runnable 214
Б. Мова Ada95 214
Б.1. Пакет Synchronous_Task_Control 214
Б.2. Пакет Asynchronous_Task_Control 215
В. Бібліотека Win 32 215
В.1. Функції очікування 215

В.2. Функції для роботи із сокетами 217
Г. Бібліотека MPI 217
Г.1. Базові функції 217
Г.2. Типи даних 219
Д. Бібліотека PVM 219
Д.1. Базові функції 219
Д.2. Типи даних 220
Д.3. Функції пакування-розпакування 221
Е. Бібліотека Pthreads 221
Е.1. Базові функції 221
Ж. Мова C# і платформа .NET 222
Ж.1. Базові члени класу AppDomain 222
Ж.2. Базові типи простору імен SystemThreading. 222
Ж.3. Базові члени класу Thread 222

ВСТУП

Програмування для паралельних і розподілених комп'ютерних систем стало невід'ємною складовою загального мистецтва програмування. Сьогодні неможливо уявити мову програмування, яка не мала б механізмів роботи з процесами (Ада, Java). Засоби роботи з процесами реалізовано також у вигляді спеціальних бібліотек (PVM, MPI,) або є частиною операційної системи (Pthreads,Win32). Крім програмування для багатопроцесорних та розподілених комп'ютерних систем, процеси використовують також для розроблення програм для систем реального часу.

Уміння розробити паралельний алгоритм задачі, описати алгоритми паралельних процесів, організувати взаємодію процесів, вирішити завдання взаємного виключення та синхронізації процесів, передати дані між процесами, створити та налагодити паралельну або розподілену програму, розмістити та виконати її в реальній паралельній комп'ютерній системі – актуальне завдання для сучасного програміста.

Проблеми, що пов'язані з розробленням та виконанням паралельних програм, вдало сформульовані в одній з перших книжок, присвяченій програмуванню для паралельних систем [31]:

"Поведінка навіть дуже коротких паралельних програм може виявитись на диво складною. Той факт, що програма з конкретним набором даних виконувалась правильно один або сто разів, не гарантує, що завтра ця програма з цими ж вхідними даними не завершиться з помилкою".

Матеріали навчального посібника ґрунтуються на циклу лекцій, які кілька років читаються студентам Національного технічного університету України «Київський політехнічний інститут» (НТУУ «КПІ») і Національного авіаційного університету (НАУ) за напрямом 0915 «Комп'ютерна інженерія».

Основна мета цього видання – закласти основи для програмування паралельних і розподілених обчислень. Він містить шість розділів. Кожен розділ закінчується завданнями для модульного контролю та для самостійної роботи.

У першому розділі розглянуто структури сучасних комп`ютерних систем, які ґрунтуються на паралельній організації, а також особливості організації обчислювальних процесів у них. Це системи зі спільною і розподіленою пам'яттю, а також розподілені системи.

У другому розділі розглядаються поняття процесу, його стани, базові операції над процесами, їх реалізація в мовах програмування Java і Ада95, бібліотеках PVM, MPI та Win32, а також питання побудови й аналізу паралельних алгоритмів на прикладі задач лінійної алгебри.

Третій розділ присвячено питанням програмування для систем зі спільною (розподільною) пам'яттю. Особливу увагу приділено проблемі організації взаємодії процесів у таких системах, яка ґрунтується на використанні моделі спільних змінних. Розглянуто постановку і загальну схему вирішення завдань взаємного виключення і синхронізації процесів – фундаментальних завдань, які пов'язані з програмуванням для систем зі спільною пам'яттю, а також механізми, використовувані для розв'язування цих завдань: семафори, мютекси, події, критичні секції, монітори. Наведено приклади їх застосування.

У четвертому розділі приділено увагу питанням програмування для систем з розподіленою пам'яттю, а також можливим видам моделі посилання повідомлень – основного механізму організації взаємодії процесів у таких системах.

П'ятий розділі присвячено питанням програмування для розподілених комп'ютерних систем. Основну увагу приділено механізму сокетів і виклику віддалених процедур, та їх реалізації в мовах програмування (Java, Ада95).

Шостий розділ відведено прикладам програмування для паралельних і розподілених систем, у яких детально розглянуто всі етапи розроблення програм для подібних систем і виконано практичне застосування розглянутих засобів розроблення програм.

У додатках наведено дані про базові засоби роботи з процесами в бібліотеках PVM, MPI, Pthreads, Win32 та в мовах Java, Ада95, C#.

Дані про авторів:

ЖУКОВ Ігор Анатолійович – д-р. техн. наук, професор, Заслужений винахідник України, директор Інституту комп'ютерних технологій, завідувач кафедри обчислювальної техніки Національного авіаційного університету України. Член науково-методичної комісії Міністерства освіти і науки з комп'ютерної інженерії, експертної ради з комп'ютерних наук і технологій при Державній акредитаційній комісії України. Голова спеціалізованої вченої ради Д26.062.07 із захисту докторських дисертацій НАУ. Відповідальний редактор збірника наукових праць «Проблеми інформатизації та управління».

Спеціаліст у галузі обчислювальних систем та мереж і апаратно-програмних засобів підвищення їх продуктивності. Викладає дисципліни «Обчислювальні системи» та «Мережі ЕОМ».

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 з комп'ютерної інженерії. Член Європейської спілки Ada-Europe.

Спеціаліст у галузі програмування для паралельних та розподілених систем, а також програмування мовою Ада. Викладає дисципліни «Паралельне програмування», «Технології розподілених обчислень», «Програмування мовою Ада».

cora@comsys.ntu-kpi.kiev.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. Комп'ютерні системи з розподіленою (локальною) пам'яттю.
1.4. 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 взаємодії – для пе-

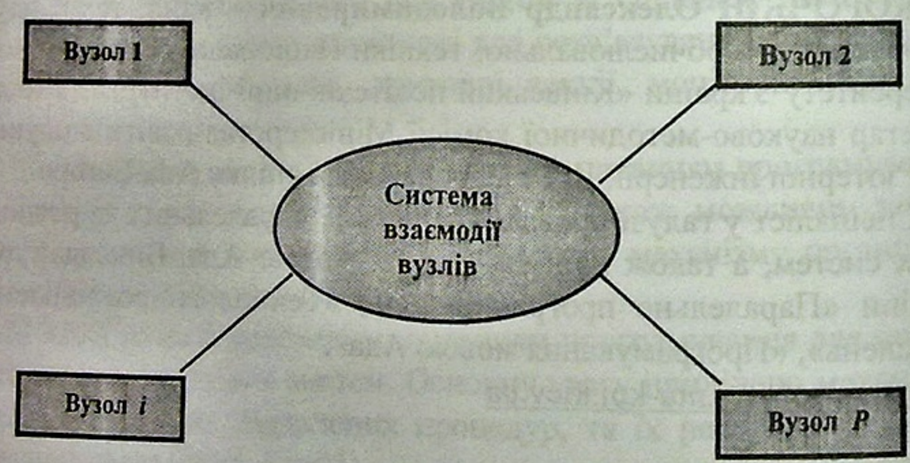


Рис. 1.1. Загальна структура КС з паралельною обробкою

редавання 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. Як вузлі в ній використовують процесори, а як систему взаємодії вузлів – спільну пам'ять .

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

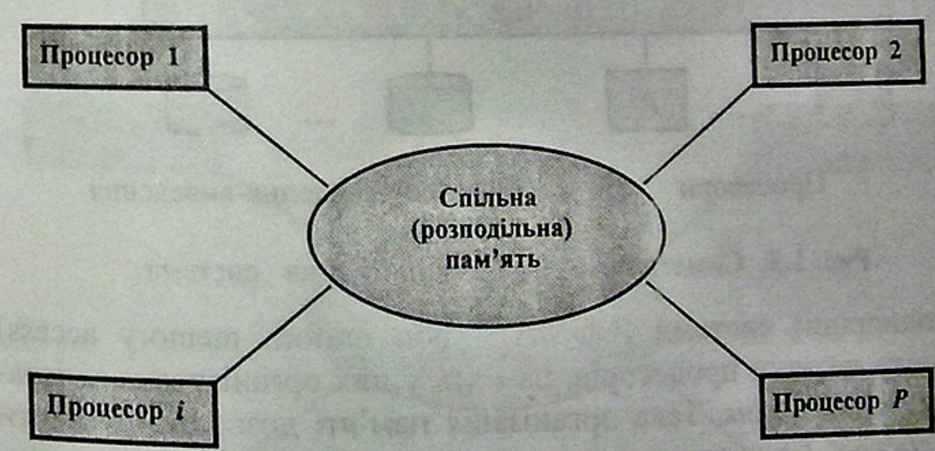


Рис. 1.2. Структура КС з розподільною пам'яттю

Така модель отримала назву взаємодії процесів, яка ґрунтується на *розподільних змінних (shared – variables based synchronization and*

communication). Система з розподільною пам'яттю є повнозв'язаною, тому що спільна пам'ять забезпечує безпосередній зв'язок кожного процесора з кожним. Для цього один процес записує значення потрібної локальної змінної (яку треба передати в інший процес) у глобальну змінну, звідки інший процес зчитує це значення (копіює в свою локальну змінну).

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

Рис. 1.3. Симетрична мультипроцесорна система

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

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

ної ємності кеш-пам'яті системи. 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.

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

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

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

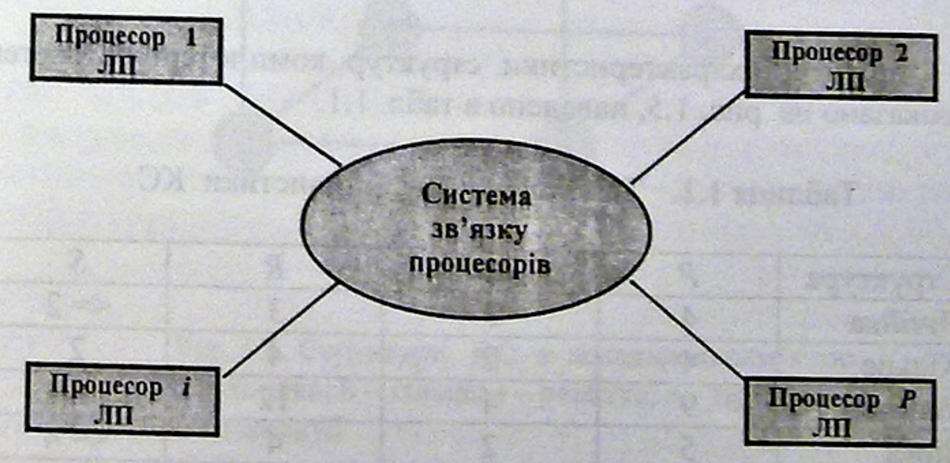


Рис. 1.4. Структура КС з локальною (розподіленою) пам'яттю

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

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

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

Кожна структура характеризується визначеними топологічними параметрами:

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

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

Таблиця 1.1. Топологічні характеристики КС

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

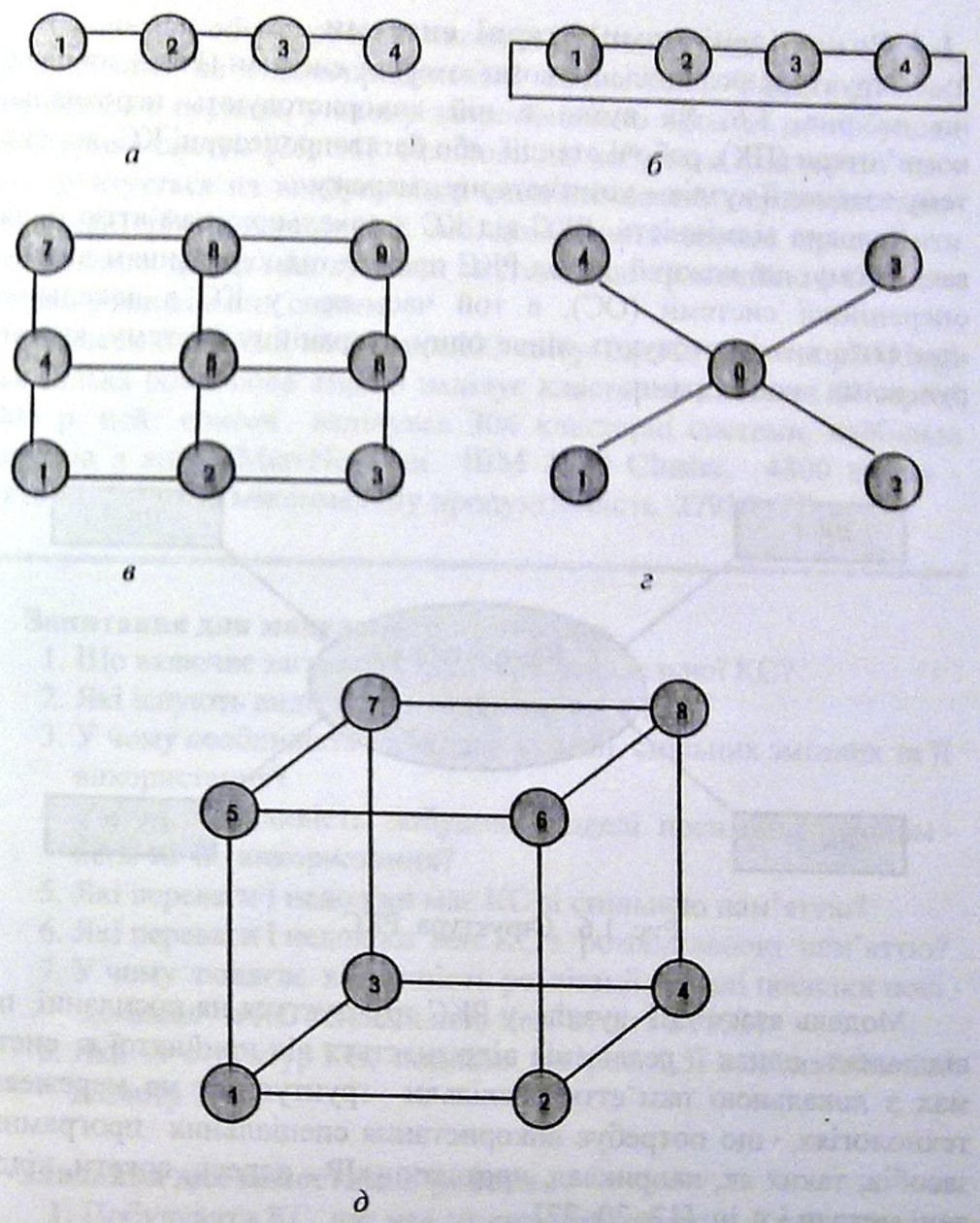


Рис. 1.5. Структури КС з локальною пам'яттю: *а* – лінійка; *б* – кільце; *в* – решітка; *г* – зірка; *д* – гіперкуб

1.4. Розподілені комп'ютерні системи

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

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

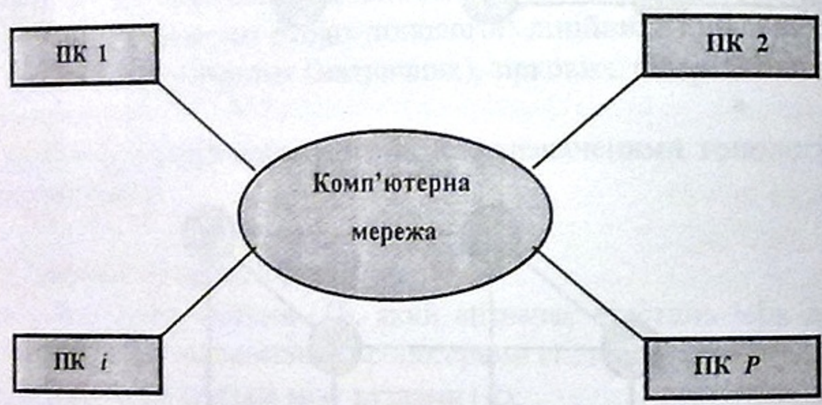


Рис. 1.6. Структура РКС

Модель взаємодії вузлів у РКС ґрунтується на посиланні повідомлень, однак її реалізація відрізняється від прийнятої в системах з локальною пам'яттю, оскільки ґрунтується на мережевих технологіях, що потребує використання спеціальних програмних засобів, таких як, наприклад, протоколи, IP - адреси, сокети, віддалені методи і т. ін. [12; 20; 22].

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

Кластерні обчислювальні системи – вид розподілених систем, орієнтованих на високопродуктивні обчислення. Проект *Beowulf* був одним з перших, у якому запропоновано підходи до побудови кластерних систем [25; 56]. Особливість побудови кластерних систем ґрунтується на використанні спеціалізованих мережевих засобів (адаптерів, комутаторів), які дозволяють значно прискорити передавання даних між вузлами системи, що традиційно є слабким місцем розподілених систем.

Список найбільш потужних КС світу *Top500* (www.top500.org) уже кілька років обов'язково налічує кластерні системи. У червні 2005 р. цей список налічував 304 кластерні системи, найбільш потужна з яких (MareNostrum IBM JS20 Cluster, 4800 вузлів - п'яте місце) мала максимальну продуктивність 27910 Tflop/s.

➢ **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 посилки повідомлень в КС з локальною пам'яттю і в РКС?
8. Яка із структур КС, поданих на рис. 1.5, має найменший діаметр ?

➢ **Завдання для самостійної роботи**

1. Побудувати КС, що має зіркову структуру з $P = 8$.
2. Виконати модифікацію структури на рис. 1.5, *г*, змінивши всі вузли, крім центрального, на топологію зірка з $P = 4$. Розрахувати топологічні характеристики такої КС.
3. Побудувати КС різної структури, які мають діаметр, що дорівнює 8.
4. Визначити діаметр КС, що має топологію решітки з кіль-

кістю процесорів, що дорівнює P.

5. Проаналізувати структури КС, що показані на рис.1.5, з погляду надійності, у разі, якщо один з процесорів вийде з ладу.
6. Побудувати структури КС для гіперкуба, якщо значення ступеня гіперкуба дорівнюють 0, 1, 2 , 3, 4.
7. Розрахувати топологічні характеристики КС типу лінійки із $P = 6$.
8. Розрахувати топологічні характеристики КС типу кільця із $P = 5$.
9. Розрахувати топологічні характеристики КС типу решітки.
10. Розрахувати топологічні характеристики КС типу гіперкуба ступеня 4.

Архитектоника мышления и нейроинтеллект

РОЗДІЛ 2. Процеси

2.1. Поняття процесу.
2.2. Стан процесу. Керування процесами.
2.3. Процеси в мові Java.
2.4. Процеси в мові Ада95.
2.5. Процеси в бібліотеці MPI та PVM.
2.6. Процеси в бібліотеці Win32.
2.7. Паралельні алгоритми.
2.8. Розроблення паралельного алгоритму.

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

2.1. Поняття процесу

Процес – абстрактне поняття, що включає опис певних дій, пов'язаних з виконанням програми в комп'ютерній системі [9; 35; 45; 48]. При цьому процес оформлюється так, щоб система керування процесами в ОС могла ефективно перерозподіляти ресурси системи (процесори, пам'ять, прилади введення-виведення, файли і т.ін.). Процес характеризується власним набором ресурсів і виділеною для нього ділянкою оперативної пам'яті.

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

Block) [9]. Блок керування процесом – динамічна структура даних, яка містить основну інформацію про процес:

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

Стосовно паралельного програмування процеси – це частини однієї програми користувача, які виконуються одночасно. Такі процеси отримали назву *легкі процеси* (*lightweight processes*). Для позначення легких процесів використовують також терміни *потік* (Java, C, C#) або *задача* (Ада, MPI). Тому традиційні процеси іноді називають *важкими процесами* (*heavyweight processes*).

Під час виконання кількох програм у комп'ютерній системі або звичайному комп'ютері відбувається постійне перемикання з одного процесу на інший. Те ж саме відбувається і для одиночної паралельної програми, якщо вона включає кілька легких процесів (потоків). 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 (*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, містять здебільшого такі стани:

Породження – створення процессу і підготовка до першого виконання в системі;

Готовності – процес готовий до виконання і чекає звільнення процесора;

Виконання – процес виконується на процесорі;

Блокування – процес призупинений через очікування визначеної події: завершення введення даних, завершення заданого часу очікування, сигналу від іншого процесу, звільнення ресурсу та ін.;

Завершення – нормальне або аварійне завершення виконання процесу.

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

Процес також може мати особливий стан, який отримав назву *тупик* (deadlock). Процес у тупиковому стані блокований і очікує на подію, яка ніколи не відбудеться. 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.

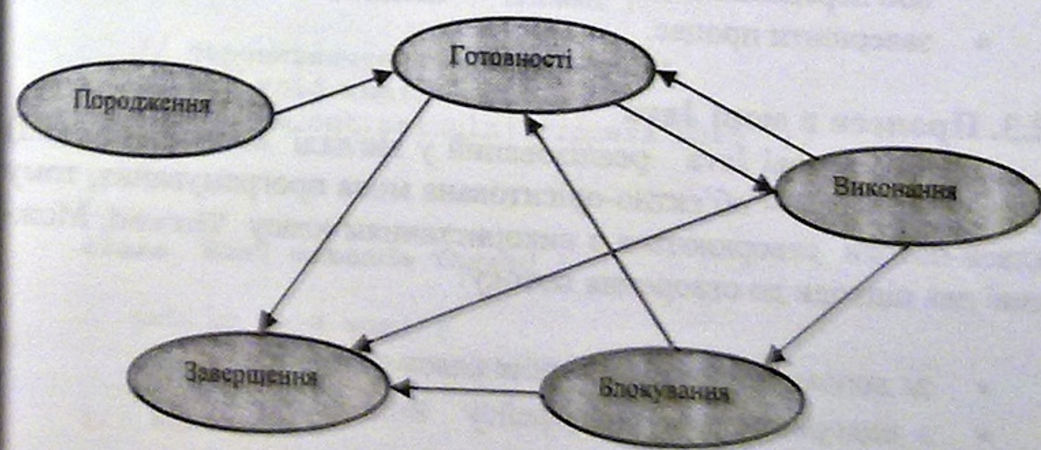


Рис. 2.1. Діаграма станів процесу

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 вивчаються взаємодії процесів.

Засоби роботи з процесами розподіляються на бібліотечні та мовні. Прикладами перших є засоби бібліотек Win32, PVM, MPI, OpenMP, POSIX, Pthread, які реалізовані у вигляді набору функцій. Вони дозволяють роботу з процесами в будь-яких мовах програмування, зокрема і в тих, які не мають вбудованих засобів роботи з процесами. Пізніші мови програмування, такі як C#, Java або Ада, безпосередньо мають засоби роботи з процесами, які вбудовані в мови, що забезпечує більш ефективне їх виконання та взаємодію.

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

- оголосити процес (групу процесів);
- установити пріоритет процесу;
- запустити процес на виконання,
- призупинити процес на визначений час;
- блокувати та розблокувати процес;
- організувати взаємодію з іншими процесами (синхронізації або передавання даних);
- завершити процес.

2.3. Процеси в мові Java

Процес в мові Java реалізований у вигляді *потоку* (thread) [26; 33; 45]. Java – об'єктно-орієнтована мова програмування, тому класи-потоки створюються з використанням класу `Thread`. Можливі два підходи до створення потоку:

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

Використання класу Thread. Клас `Thread` інкапсулює потік, 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 має перевизначити метод `run()`, що є точкою входу в потік. Необхідно також викликати метод `start()`, щоб почати виконання потоку.

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

Таблиця 2.1. Методи класу `Thread`

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

❖ **Приклад 2.1**

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

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

   class  West extends Thread {

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

   // головний потік
   class  MainThread {
```

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

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

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

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

    }// main
  }// MainThread
```

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

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

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

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

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

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

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


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

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

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

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

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

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

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

❖ Приклад 2.2

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

        String  Name;
        Thread  t;

        // Конструктор класу Denon
```

```
    Denon(String ім'я){
      Name = ім'я;
      t  = new  Thread(this, N);
      System.out.println("   New thread" + Name);
      t.start();
    }
    // точка входу в потік
    public void run(){

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

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

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

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


2.4. Процеси в мові Ада

Ада – відома мова програмування, яку було розроблено в 1983 році на замовлення Міністерства оборони США [16; 58]. Призначення мови – розроблення великих програмних систем, робота яких характеризується високою надійністю. Ада – одна з перших мов програмування, які мають вбудовані засоби роботи з процесами.

Процеси в мові Ада реалізовані у вигляді спеціальних модулів – *задач* (`task`). Задачний модуль `task` один із п'яти видів модулів мови, має стандартну форму у вигляді специфікації і тіла. Специфікація визначає інтерфейс задачі, де задається ім'я задачі, а також у разі потреби, – 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 *вироджена*. Тіло задачі визначає дії задачі під час виконання.

Задачний модуль не є одиницею компіляції в мові і тому задача має бути описаною в підпрограмі або пакеті. Мова немає явних засобів запуску задачі. Задача *автоматично* стартує під час запуску підпрограми або входячи в блок, у якому її описано. Якщо задачі розміщені в головній програмі, то із запуском основної програми задачі починають виконуватися одночасно з головною програмою, яка, в свою чергу, завжди розглядається як задача і виконується паралельно з вкладеними в ній задачами. Явний запуск задач можливо реалізувати через розміщення їх в окремій процедурі, виклик якої в потрібний момент спричинить запуск вкладених у ній задач. Крім того, запуск задач можна реалізувати за допомогою посильного (`access`) типу та генератора `new`.

❖ Приклад 2.3

```
-------------------------------------------------------------
--  Ада95.Оголошення та запуск задач                        --
-------------------------------------------------------------
procedure Lab23 is

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

    -- тіла задач A і B
    task body A is
    begin
```

```
    put_line("Process A started");
  end A;

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

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

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

❖ **Приклад 2.4**

```
--------------------------------------------------------
--   Ада95.Задачний тип з дискримінантом              --
--------------------------------------------------------
procedure Lab24 is

   -- задачний тип з дискримінантом
   task type  ЗадачнТип(Номер : integer);

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

   A: ЗадачнТип(1);  -- створення задач
   B: ЗадачнТип(2);
begin
   null;             -- 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 виборювати ресурси – процесор, прилади введення–виведення, а також взаємодіяти з іншими задачами або захищеними модулями. Тобто пріоритети працюють там, де з'являються черги, у яких процеси можуть знаходитись під час очікування. Формування таких черг та вибірка з них здійснюються з використанням пріоритетів задач.

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

❖ **Приклад 2.5**

```
--------------------------------------------------------
--Ада95.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;
```

Порівняльний аналіз засобів мов Java та Ада95 для роботи процесами наведено в праці [17].

2.5. Процеси в бібліотеках MPI та PVM

Бібліотеки MPI та PVM – найбільш поширені засоби програмування для комп'ютерних систем з розподіленою пам'яттю та розподілених комп'ютерних систем [2; 25; 42].

Бібліотека (інтерфейс) MPI (Message Passing Interface) містить набір функцій, що дозволять організувати роботу з процесами в мовах, які не мають вбудованих засобів програмування процесів. У разі застосування бібліотеки MPI використовується мова послідовного програмування типу C або Фортран, процеси в якій програмуються за допомогою засобів інтерфейсу MPI.

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

Загальна структура MPI програми має вигляд:

```
#include "mpi.h"

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

  /* Локальні описання */

  MPI_Init(&args, &argv);
```



```
  /* Тіло програми */

  MPI_Finalize();
return 0;
}
```

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

Таблиця 2.2. Основні функції MPI

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

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

Функція

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

створює групу.

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

Створюючи в додатку задачі, їх прив'язують до спільної ділянки зв'язку. Для цього використовують поняття *комунікатор* – опис ділянки зв'язку процесів, які об'єднані в групу, а також у різ-

ні групи. Програма може вміщувати кілька ділянок зв'язку. Нуме-
рація процесів всередині ділянки зв'язку незалежна. Комунікатор
може бути використаний як параметр MPI-функції і для обмеження
сфери її дії тільки заданою ділянкою зв'язку. Крім того, комуніка-
тор забезпечує вимоги безпеки. MPI автоматично створює комуні-
катор MPI_Comm_World, який є базовим для кожного додатка
створюється автоматично під час виклику функції MPI_Init().

Функції

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

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

❖ **Приклад 2.6.**

Створення процесів, кожний з яких виводить на дисплей по-
відомлення про старт процесу.

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

   /* Визначення розміру та буфера повідомлень  */
   #define розмір_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[розмір_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);

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

      MPI_Finalize();
      return 0;

   }/* main */
```

Бібліотека PVM (Parallel Virtual Machine) була розроблена в університеті штату Тенесі, США [25]. Дозволяє розробляти додатки для множини гетерогенних комп'ютерів, з'єднаних мережею, розглядаючи їх як одну велику паралельну комп'ютерну систему. Система містить великий набір засобів для керування процесами і ресурсами, пуску і завершення задач, засоби синхронізації задач, конфігурування PVM. Підтримується мовами Фортран, C, C++.

Підтримка різних гетерогенних комп'ютерів – основна особливість PVM. Програма, що написана для однієї архітектури, може виконуватися на іншій без модифікації. PVM додаток – це набір взаємодіючих задач (процесів), які забезпечують розв'язання однієї задачі користувача. Модель обчислень у PVM – набір взаємодіючих послідовних задач, кожна задача має особисту нить керування, взаємодія задач виконується за допомогою посилання повідомлень.

У PVM вузол системи з комп'ютерам має назву *хост*. PVM містить дві компоненти: PVM бібліотеку і демон, який має розмі-

щуватися в кожному хості PVM. Для запуску PVM додатка не-
хідно запустити PVM і сконфігурувати віртуальну машину.

PVM додаток – композиція послідовних програм, кожна з як
пов'язана з одним або кількома процесами в паралельній програ
Ці програми компілюються індивідуально для кожного хосту PV
Одна із задач (що ініціалізує) запускається вручну й активізує ін
задачі додатка. PVM додаток може мати кілька структур: зір
майстер-робітника та ін.

Задача в PVM запускається або вручну, або це виконується
іншої задачі. Динамічне створення задачі виконується за допом
гою PVM функції `pvm_spawn()`. Задача, що викликає функ
`pvm_spawn()`, є батьківською задачею, створювана задача – доч
ньою задачею. Створюючи дочірню задачу, потрібно вказати:

- машину, на якій буде виконуватися дочірня задача;
- шлях (`path`) до виконавчого файлу;
- кількість копій дочірньої задачі;
- масив аргументів.

Задачі в PVM мають унікальний ідентифікатор (`Tid`), як
отримують під час створення задачі. Ідентифікатор використо
ють для взаємодії задач.

Формат функції `pvm_spawn()`:

```
num = pvm_spawn(Child, Args, Flag,
                Where, HowMany,&Tids);
```

Параметри:

- `Child` – ім'я exec файлу, котрий визначає виконан
створюваної задачі. Має резидентно знаходитись на хос
де буде виконуватися.
- `Args` – покажчик на масив аргументів задачі. Має зн
чення NULL, якщо аргументів немає.
- `Flag` – застосовують для задання режимів виконання
процесів, що запускаються.

- `Where` – ім'я хосту або типу архітектури, де буде виконуватися задача, що створена залежно від значення змінної `Flag`; тобто визначається місце запуску процесу.
- `HowMany` – кількість ідентичних дочірніх задач для запуску.
- `Tids` – ідентифікатор (TID) дочірньої задачі.

Функція `pvm_spawn()` запускає копії виконуваного файлу `Child`.

Приклади створення процесів:

```
n1 = pvm_Spawn("user/rew/work", 0, 1, "Dina",
                                        2, &tid1);
n2 = pvm_Spawn("user/rew/work", 0, 1,
                                "Zond", 4, &tid2);
```

Дві створені задачі відповідають двом і чотирьом копіями програми "`work`" для двох хостів – `Dina` і `Zond`. Обидві машини мають містити файл `work.exe` в директоріях "`user/rew/`". Значення `n1` і `n2` – номери задач, TID задачі будуть розміщені в `tid1` і `tid2`.

2.6. Процеси в бібліотеці Win32

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

Для створення потоку в Win32 використовують функції `CreateThread()` і `CreateRemoteThread()`. Дані функції повертають ідентифікатор процесу, який є унікальним і ідентифікує його в системі. Під час створення потоку визначається початковий адрес коду, з якого має виконуватись потік. Зазвичай, це назва функції, яка буде виконуватися як процес.

Функція:

```
HANDLE Ім'я_Потоку = CreateThread(
    LPSECURITY_ATTRIBUTES  атр,   // атрибут безпеки
    SIZE_T    рс,                 // розмір стека
    LPTHREAD_START_ROUTUNE фп,    // функція потоку
    LPVOID  арг,          // аргумент функції потоку
```

```
    DWORD    пр,          // прапорець
    LPDWORD ід);          // ідентифікатор потоку
```

створює потік, для якого фактичні параметри визначають ім'я функції та її параметр, атрибути безпеки, початковий розмір стека потоку, прапорець створення.

❖ **Приклад 2.7**

Створення двох потоків, які будуть виконувати функцію Task Func:

```
void Task(void);

int main(void){

    DWORD TidA, TidB;
    HANDLE hThreadA, hThreadB;

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

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

// функція, яка виконується в потоці
void Task_Func(void){
    printf("Потік стартував ");
        . . .
    printf("Потік 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.

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.

2.7. Паралельні алгоритми

2.7.1. Аналіз паралельного алгоритму

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

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

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

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

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

До появи паралельних комп'ютерних систем використовувались виключно послідовні алгоритми, орієнтовані спочатку на людину, а потім на однопроцесорні комп'ютери. Проблема розроблення паралельних алгоритмів з'явилась з появою реальних паралельних систем. "Паралельна математика", яка вивчала методи і засоби побудови та аналізу паралельних алгоритмів бурхливо розвивалась у 60–70-х роках минулого століття [4; 7; 36]. Учені отримали цікаві результати про паралельні властивості цілого ряду задач. При цьому для окремих задач довелося відмовитися від традиційних підходів і розробити цілком нові, які дозволили побудувати ефективні паралельні алгоритми. З'ясувалося, що оцінка ефективності різних паралельних алгоритмів – важке завдання, тому що для цього треба враховувати різні додаткові фактори, наприклад, час передачі даних між процесорами.

Для спрощення аналізу паралельних алгоритмів було запропоновано *концепцію необмеженого паралелізму*, яка ґрунтувалась на таких поняттях [4; 7]:

- 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 і вибрати оптимальний алгоритм.

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

$$K_n = T_1 / T_p$$

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

$$K_e = T_1 / (T_p * P)$$

Для поєднання необмеженого паралелізму в концепції необмеженого паралелізму з обмеженою кількістю процесорів можна використати лему Брента [4]:

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

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

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

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

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

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

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

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

і на підставі теореми Мунро–Петерсона отримуємо

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

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

Ярусно-паралельна форма алгоритму. Подання паралельного алгоритму може бути виконано різними способами [7]. *Ярусно-паралельна форма* (ЯПФ) для паралельного алгоритму являє собою набір ярусів, у яких операції виконуються паралельно (рис. 2.2). Висота ЯПФ (H) – кількість ярусів, ширина i-го ярусу (R_i) – кількість операцій в ярусі, ширина ЯПФ (R) – найбільша з ширин ярусів. Виконання ЯПФ здійснюється за ярусами. Тому час виконання задачі, алгоритм який представлений ЯПФ, дорівнює висоті ЯПФ, а максимальна потрібна кількість процесорів – ширині ЯПФ.

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

Для ЯПФ, показаній на рис. 2.2, параметри дорівнюють: $H = 4$, $R_1 = 4$, $R_2 = 2$, $R_3 = 3$, $R_4 = 2$, $R = \max(4, 2, 3, 2) = 4$.

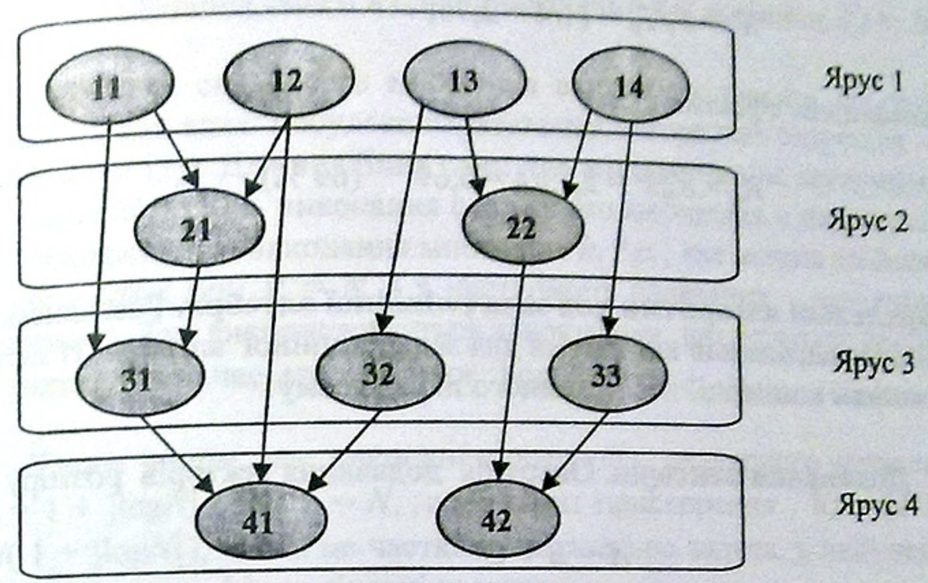


Рис. 2.2. Ярусно-паралельна форма алгоритму

Знання цих параметрів дозволяє розрахувати:

- час розв'язування задачі в однопроцесорній системі:

$$T_1 = R_1 + R_2 + R_3 + R_4 = 11;$$

- час розв'язування задачі в багатопроцесорній системі:

$$T_p = H = 4;$$

- кількість процесорів, потрібних на кожному кроку обчислення ЯПФ:

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

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

$$P = R = 4;$$

- коефіцієнт прискорення:

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

- коефіцієнт ефективності:

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

Паралельні алгоритми для задач лінійної алгебри. Розглянемо оцінимо паралельні алгоритми для задач лінійної алгебри із заст суванням концепції необмеженого паралелізму.

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

$$A = B + C$$

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

Час виконання задачі в однопроцесорній КС дорівнює $T_1 =$ час виконання в паралельній КС дорівнює $T_p = 1$ для $P = N$, к фіцієнт прискорення $K_n = P$, тобто це ідеальна для виконанн паралельній системі задача.

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

$$A_H = B_H + C_H,$$

де $H = N/P$; A_H – H елементів вектора A.

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

$$a = (B * C)$$

виконується за співвідношенням

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

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

Операцію скалярного множення векторів відносять до групи операцій, для яких побудова паралельних алгоритмів базується на *здвоюванні* [7]. Для подібних схем час виконання має логарифмічну залежність. Час виконання скалярного множення в паралельній КС складається із виконання множення $b_i * c_i$, яке можна здійснити за один крок при $P = N$, і з наступного додавання результатів множення. Тут використовується здвоювання, що дозволяє виконати множення за час, що дорівнює $]\log N[$.

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

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

$$a_i = (B_H * C_H), \qquad i = 1..P.$$
$$a = a + a_i$$

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

$$A = B * MC$$

виконується за співвідношенням

$$a_i = b_1 * c_{1i} + .. + b_i * c_{ii} + .. + b_n * c_{ni}, \qquad i = 1..N.$$

Час виконання задачі в однопроцесорній КС $T_1 = N*(N+(N-1))$. В паралельному алгоритмі використано схему здвоювання, тому час виконання в паралельній КС $T_p = 1 +]\log_2 N[$ якщо $P = N*N$, і коефіцієнт прискорення дорівнює $K_n = (2N*(N+(N-1))/(1 + \log_2 N)$, тобто це частково паралельна задача, в якій чергуються паралельні і послідовні частини.

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

$$A_н = (B * MC_н),$$

де $MC_н$ – H рядків (стовпців) матриці MC.

- **Додавання матриць.** Операцію додавання матриць

$$MA = MB + MC$$

виконуєть за співвідношенням

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

Час виконання задачі в однопроцесорній КС $T_1 = N*N$, час виконання в паралельній КС $T_p = 1$ для $P = N*N$, коефіцієнт прискорення $K_n = P$, тобто це теж ідеальна для виконання в паралельній КС задача, але слід звернути увагу на те, як збільшилася кількість процесорів в КС, що потрібно для розв'язання задачі за одиницю часу.

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

$$MA_н = MB_н + MC_н .$$

- **Множення матриць.** Операція множення матриць

$$MA = MB * MC$$

виконується за співвідношенням

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

Час виконання задачі в однопроцесорній КС $T_1 = N*N*(N + (N-1))$, час виконання в паралельній КС $Tp = 1 +]\log N[$ якщо

$P = N*N*N$, коефіцієнт прискорення $Kn = N*N*(N + (N-1)) / (1 +]\log N[ )$.

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

$$MA_н = MB * MC_н.$$

Кожен процес формує свою частину результату $MA_н$ (H рядків), використовуючи повну матрицю MB і частину матриці $MC_н$ (H стовпців).

2.7.2. Розроблення паралельного алгоритму

Розроблення алгоритму розв'язування завдання завжди була важливим і найбільш складним етапом розроблення програми [18; 21; 28; 53]. 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. 2.3.

1. **Декомпозиція.** Крок, на якому вхідне завдання аналізується і розкладається на частини – підзадачі, виконання яких може бути паралельним (на рис. 2.3 це підзадачі A .. P). Існують різні види декомпозиції : *декомпозиція за даними, функціональна декомпозиція, об'єктна декомпозиція* та ін. [7; 21] . 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 (завантаження процесорів) буде спрощеним, якщо підзадачі мають однаковий розмір. Щодо розміру підзадач, то він пов'язаний з поняттям «зерна» алгоритму, яке, в свою чергу, пов'язане з рівнем паралелізму і типом використовуваної паралельної системи. Дрібнозернистий паралелізм – це рівень команд, середньозернистий – рівень циклів, процедур і модулів (класів), великозернистий – рівень програм.

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

2. **Проектування взаємодії підзадач.** Визначаються дії і метод
які потрібні для організації взаємодії підзадач (передавання д
них, синхронізація, доступ до спільних даних та ін.) Види вз
модії можуть бути *локальними або глобальними* залежно від т
го, яку кількість зв'язків з іншими має кожна підзадача; сн
хронними або асинхронними залежно від виду виконання об
ну даних; *статичними або динамічними*, якщо вони можу
змінюватися в процесі виконання програми.

Організація взаємодії тісно пов'язана з типом архітекту
використовуваної КС. Для систем, де використовується вз
модія, що ґрунтується на моделі посилання повідомлень, оп
мальність цієї організації буде значною мірою визначати ч
виконання програми, оскільки в таких системах час пере
вання повідомлень можна порівнювати з часом безпосеред
обчислень і впливати на загальний час роботи програми. Ск
рочення часу на взаємодію процесів можна досягнути, як
зменшити загальну кількість взаємодій, а також обсяг даних,
передаються, і, по можливості, виконувати їх одночасно.

3. **Об'єднання.** Виконується об'єднання окремих підзадач у бі
ші задачі з метою зменшення обсягів даних, які потріб
передавати між задачами. Кінцева мета об'єднання – збільше
ня ефективності алгоритму за рахунок скорочення часу йо
виконання і складності реалізації.

Результат кроку об'єднання для рис. 2.3 – формуван
трьох задач *T1* (підзадачі *A, B, G, H*), *T2* (підзадачі *C,*
і *T3* (підзадачі *D, E, F, L, M, P*).

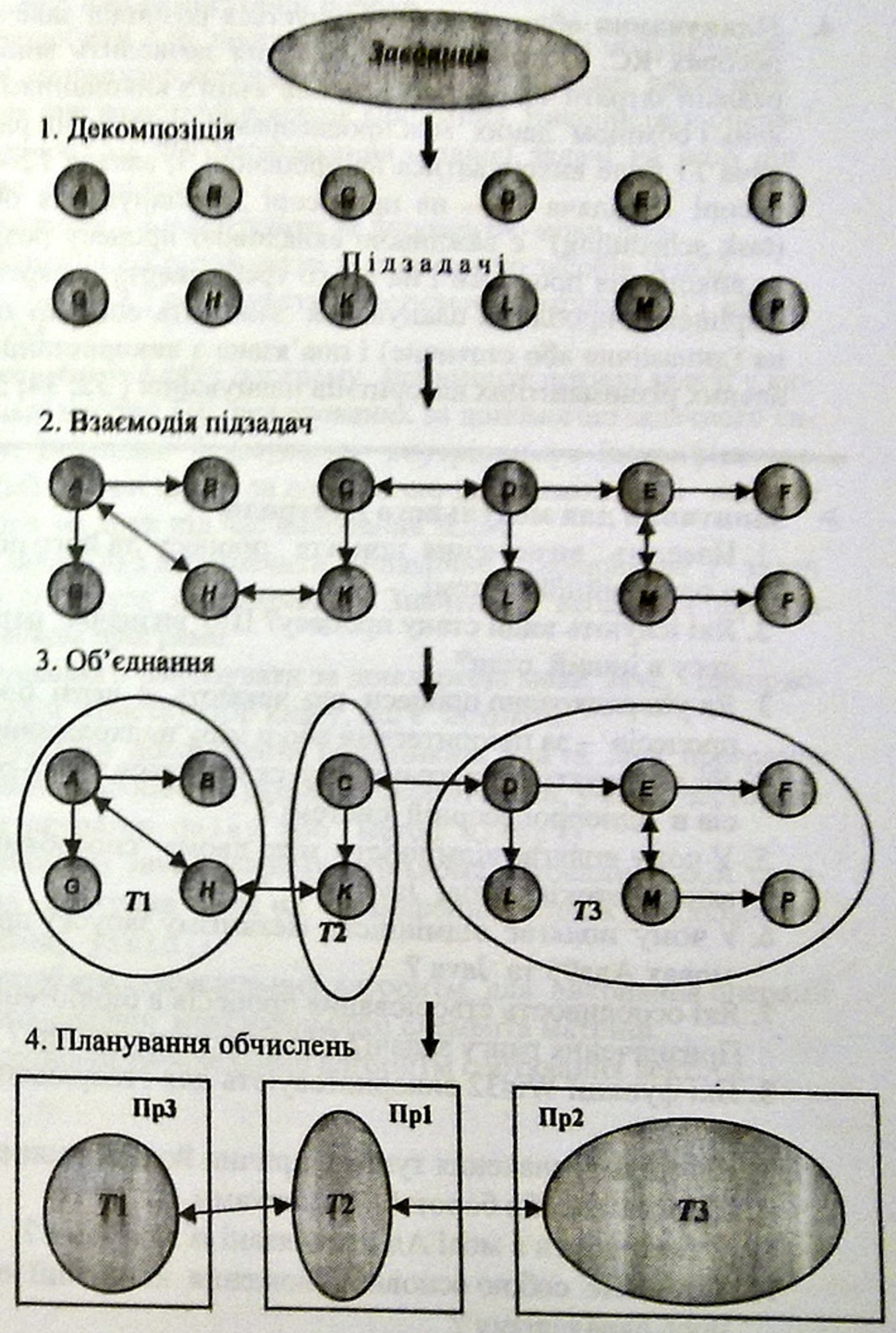


Рис. 2.3. Етапи розроблення паралельного алгоритму

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. 2.3 задача *T*1 буде виконуватися на процесорі 3, задача *T*2 – на процесорі 1, задача *T*3 – на процесорі 2. Планування обчислень (task scheduling) є важливою складовою процесу розроблення та виконання програми і на нього треба звертати окрему увагу. Вирішення проблеми планування залежить від виду планування (динамічне або статичне) і пов'язане з використанням спеціальних різноманітних алгоритмів планування [32; 34; 54].

➢ **Запитання для модульного контролю**

1. Наведіть визначення поняття процесу та його подання в операційній системі.
2. Які існують види стану процесу? Що визначає перехід процесу в інший стан?
3. Як упорядковано процеси, що чекають в черзі блокованих процесів – за пріоритетами або в міру надходження ?
4. Як виконується програма, яка складається з кількох процесів в однопроцесорній системі ?
5. У чому полягає відмінність між двома способами створювання потоків у мові Java ?
6. У чому полягає відмінність механізму запуску процесів у мовах Ада95 та Java ?
7. Які особливость створювання процесів в бібліотеці PVM ? 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. Що являє собою основні положення концепції необмеженого паралелізму ?
12. Що дає лема Брента ?

➢ **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 2.1 реалізувати за допомогою мови Java.
3. Завдання 2.1 реалізувати за допомогою засобів Win32.
4. Завдання 2.1 реалізувати за допомогою засобів MPI або PVM.
5. Розробити Ада95 програму, яка вміщує чотири задачі у вигляді масиву *T(i)*, реалізованих за допомогою задачного типу. Виконати формування внутрішнього ідентифікатора (*Tid*) кожної задачі за допомогою дискримінанта і вивести його на друк під час виконання задач.
6. У завданні 5 забезпечити динамічне створювання *P* задач та їх запуск на виконання. Значення *P* вводиться перед запуском програми.
7. Завдання 5 реалізувати за допомогою мови Java, використавши 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()`.
9. Створити Java програму з кількома потоками, у якій головна програма чекає на завершення потоків за допомогою методу `yield()`.
10. Розробити паралельний алгоритм для виконання операції знаходження максимального елемента матриці.
11. Розробити паралельний алгоритм сортування вектора.

РОЗДІЛ 3.
Взаємодія процесів, що ґрунтується на спільних змінних

3.1. Взаємодія процесів.
3.2. Семафори.
3.3. Монітори.
3.4. Вирішення завдання взаємного виключення.
3.5. Вирішення завдання синхронізації процесів.

3.1. Взаємодія процесів

Взаємодія процесів пов'язана з двома основними завданнями:

- комунікацією процесів;
- синхронізацією процесів.

Комунікація процесів передбачає обмін даними між процесами. При цьому інформація передається від одного процесу до іншого будь-якому напряму.

Синхронізація включає узгодження поведінки процесів і залежність виконання одного процесу від *подій*, що можуть бути в іншому процесі.

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

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

Використання спільних змінних, що доступні будь-якому процесу (це обов'язково глобальні змінні), приводить до того, що в програмі може відбуватися одночасне звертання процесів до спільних змінних, що призводить до конфлікту процесів або некоректної роботи програми. Тому програмування з використанням моделі спільних змінних пов'язано з необхідністю вирішення специфічних проблем, які загалом зводяться до розв'язання двох основних завдань – *взаємного виключення* і *синхронізації процесів*.

3.1.1. 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 *спільного ресурсу* (СР). Таке звернення зазвичай призводить до конфлікту процесів, що полягає або в різкому уповільненні роботи програми, або в некоректній її роботі, якщо, наприклад, один процес зчитує дані, а другий їх змінює в цей же час, або (в крайньому випадку) – до аварійного завершення програми.

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

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

Класична схема організації такого контролю потребує використання операцій (примітивів) *ВХІДКД* і *ВИХІДКД*, що розміщуються відповідно перед і після критичної ділянки, тобто обрам-

ляють критичну ділянку, створюючи "огорожу" навколо неї (рис. 3.1).

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 цьому процесу увійти в його критичну ділянку.

Алгоритм виконання операції *ВИХІДКД*:

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

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

Другий підхід до вирішення завдання взаємного виключення передбачає контроль безпосередньо за спільним ресурсом (рис. 3.2). Такий контроль може бути здійснений за допомогою оголошення належних змінних спільними ресурсами, а також за допомогою моніторів. З цією метою створюється програмний модуль (монітор), що містить в собі спільний ресурс, а також набір процедур для доступу до спільного ресурсу. Тепер доступ до спільного ресурсу з процесів можливий тільки через виклик потрібної процедури монітора.

Процедури монітора мають важливу властивість – вони *взаємно виключають* один одного. Тобто, якщо процес викликав і виконує будь-яку процедуру монітора, то виклик іншим процесом будь-якої процедури цього монітора приведе до блокування цього процесу до того часу, поки не закінчиться виконання вже розпочатої процедури монітора. Таким чином, монітор дозволяє виконання

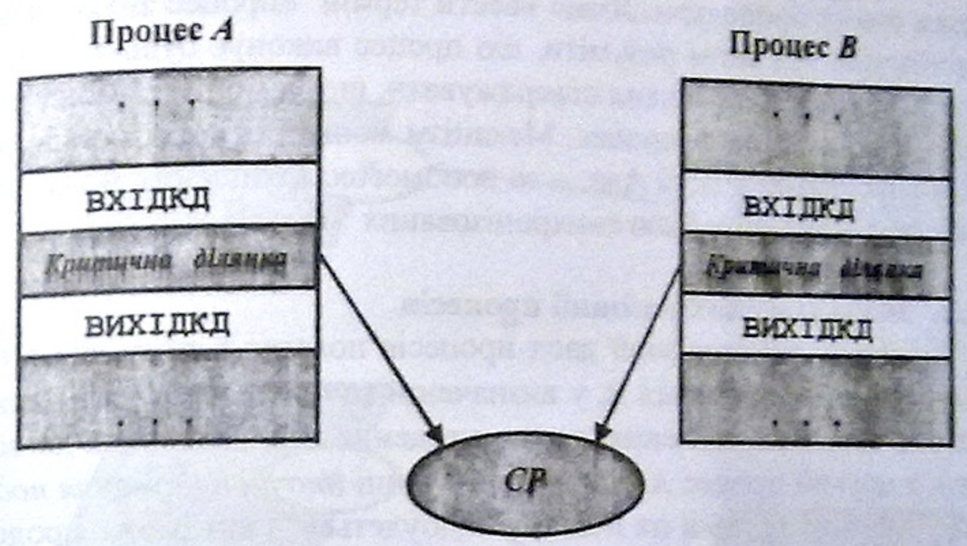


Рис. 3.1. Загальна схема вирішення завдання взаємного виключення, яка ґрунтується на контролі процесів

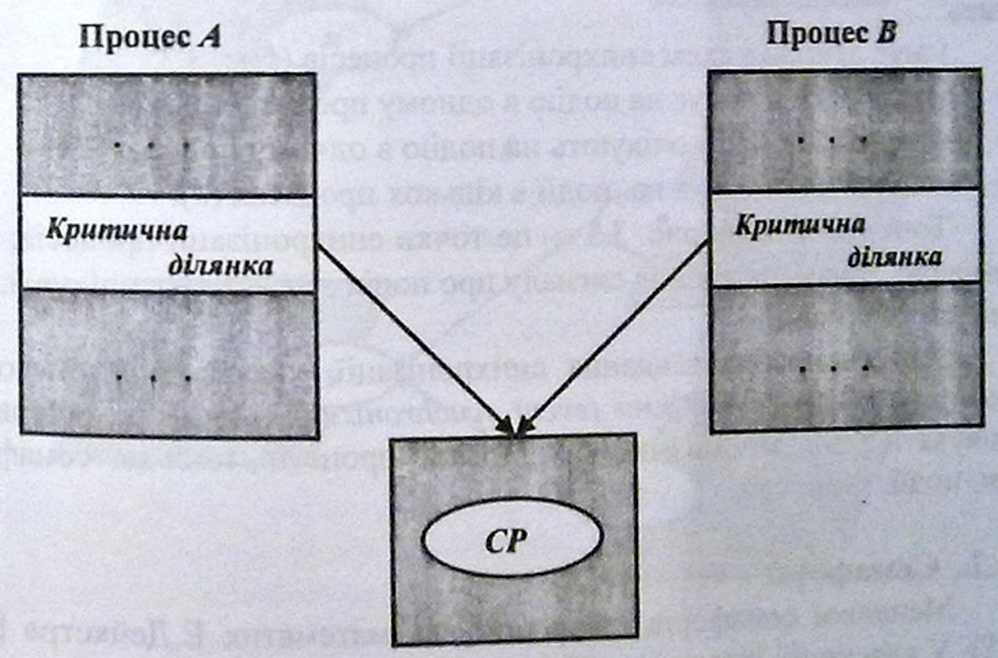


Рис. 3.2. Загальна схема вирішення завдання взаємного виключення, яка ґрунтується на контролі спільного ресурсу

тільки однієї процедури. Якщо ввести термін «процес знаходить 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, у мові Java – за допомогою синхронізованих методів.

3.1.2. Завдання синхронізації процесів

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

Існує декілька схем синхронізації процесів (рис. 3.3):

- один процес очікує на подію в одному процесі (*a*);
- декілька процесів очікують на подію в одному процесі (*b*);
- один процес очікує на події в кількох процесах (*c*).

Точки `S` і `W` на рис. 3.3 – це точки синхронізації процесів: `S` означає точку посилання сигналу про подію, `W` – точку очікування сигналу про подію.

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

3.2. Семафори

Механізм семафорів запропонував математик Е.Дейкстра [35]. У класичній інтерпретації механізм семафорів – це спеціальний захищений тип `Semaphore` та дві неподільні операції над змінною `S` цього типу: `P(S)` і `V(S)`. Неподільність операції означає, що її не можна переривати, поки не завершиться її виконання. Бінар

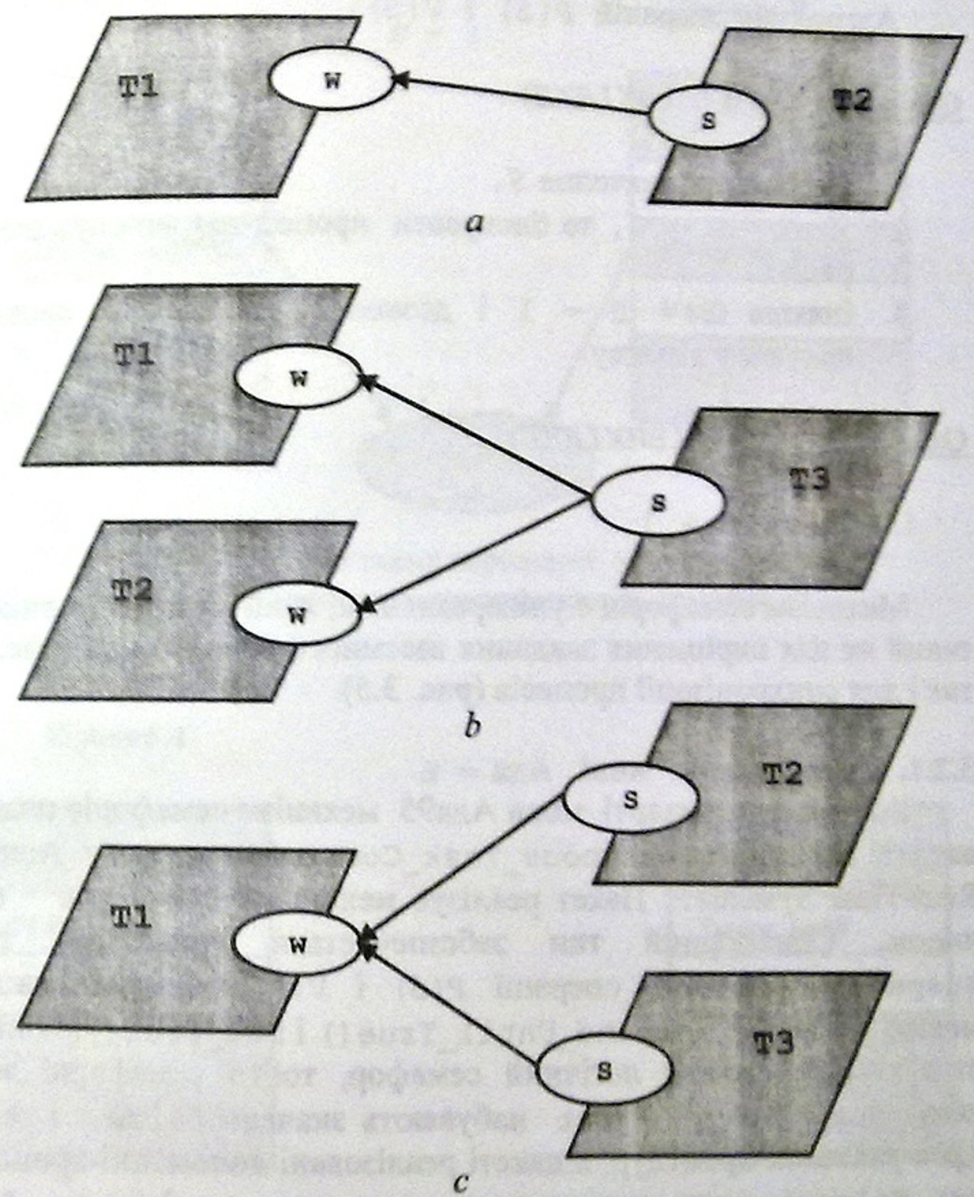


Рис. 3.3. Види синхронізації процесів

семафори набувають значень 0 і 1, багатозначні (множні) семафори – значень 0, 1, ..., *M*.

Алгоритми операцій `P(S)` і `V(S)`:

Операція P(S): (*ВХІДКД*)

1. Перевірити значення `S`.
2. Якщо `S = 0`, то блокувати процес, що виконує цю операцію.
3. Інакше `S:= S - 1` і дозволити входження процесу в критичну ділянку.

Операція V(S): (*ВИХІДКД*)

1. `S := S + 1`

Механізм семафорів є універсальним, який може бути використаний як для вирішення завдання взаємного виключення (рис. 3.4), так і для синхронізації процесів (рис. 3.5).

3.2.1. Семафори в мові Ада

У другому стандарті мови Ада95 механізм семафорів подано у вигляді пакета `Synchronous_Task_Control` в додатку Annex D: Real-Time Systems. Пакет реалізує механізм семафорів таким чином. Семафорний тип забезпечується приватним типом `Suspension_Object`, операції `P(S)` і `V(S)` реалізовані за допомогою процедур `Suspend_Until_True()` і `Set_True()`. Використовується бінарний логічний семафор, тобто семафорні змінні типу `Suspension_Object` набувають значень `false` і `true`. Крім указаних процедур, в пакеті реалізовані допоміжні процедури `Set_False()` для встановлення значення семафора в `false` і `Current_State()` для зчитування поточного значення семафора.

Специфікація пакета:

```
package Ada.Synchronous_Task_Control is
    type Suspension_Object  is limited private;
```

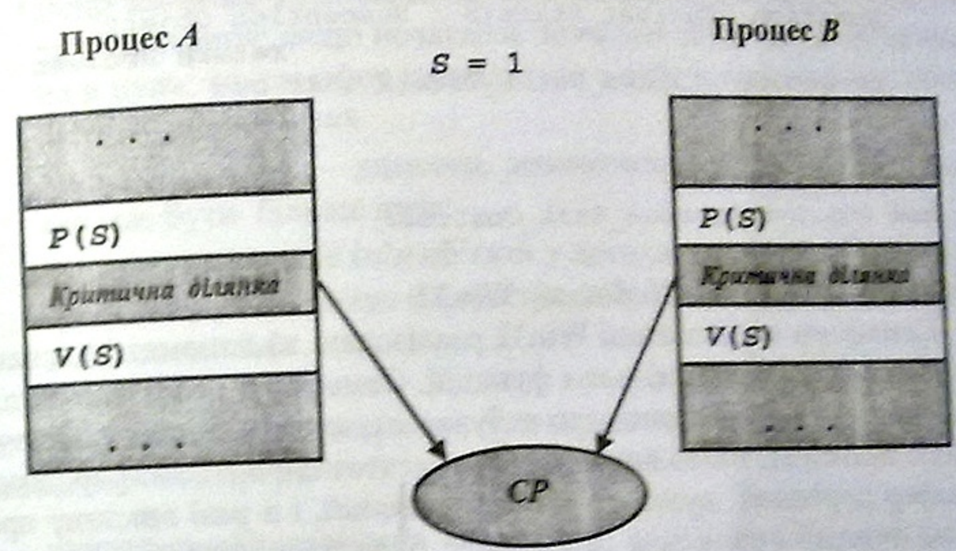



Рис. 3.4. Загальна схема вирішення завдання взаємного виключення з використанням семафорів

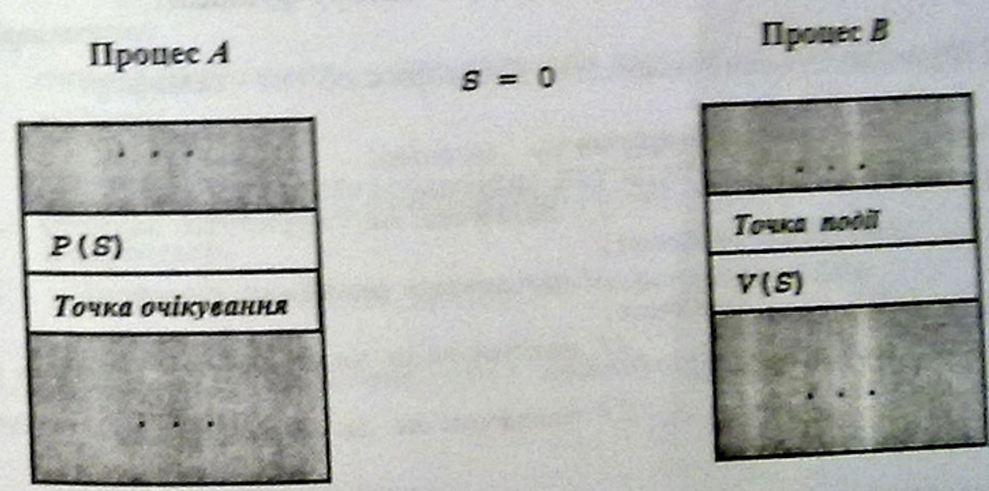


Рис. 3.5. Загальна схема вирішення завдання синхронізації з використанням семафорів

```
   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;
```

3.2.2. Семафори в бібліотеці 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 інший процес не змінить значення семафора (інкрементує лічильник).

Створення та використання семафорів у Win32 виконується за допомогою типу HANDLE, а також набору функцій:

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

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

Параметри:

- lpSemaphoreAttributes – покажчик на структуру SECURITY_ATTRIBUTES, яка визначає, чи може ідентифікатор, що повертається, бути успадкованим дочірнім процесом. Якщо покажчик установлено в NULL, то він не може бути успадкованим;

- lInitialCount – початкове значення лічильника семафора. Воно не може бути меншим за нуль або більшим за максимальне значення; якщо початкове значення лічильника встановлено в нуль, то семафор знаходиться в забороненому (закритому) стані;
- lMaximumCount – визначає максимальне значення семафора, яке має бути більше нуля;
- lpName – визначає ім'я об'єкта у вигляді рядка, який має закінчуватися нулем. Якщо параметр NULL, то створюється семафор, який не має ім'я.

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

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

Параметри:

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

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

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

Параметри:

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

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

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

Параметри:

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

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

- одиночні,
- множинні,
- попереджувальні.

Одиночні функції синхронізації `SignalObjectAndWait`, `WaitForSingleObject`, `WaitForSingleObjectEx` використовують ідентифікатор (типу `HANDLE`) об'єкта синхронізації і блоку-

ють процес, якщо об'єкт синхронізації знаходиться у забороненому стані; функції закінчують своє виконання, якщо виконується таке:

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

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

Формати виклику цих функцій:

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

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


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

);
```

Множини функцій `WaitForMultipleObjects`, `WaitForMultipleObjectsEx`, `MsgWaitForMultipleObjects`, `MsgWaitForMultipleObjectsEx` дозволяють викличному потоку визначати масив, який містить один або кілька ідентифікаторів

об'єктів синхронізації; функції закінчують своє виконання, якщо вони реалізують таке:

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

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

Формати виклику цих функцій

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

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

DWORD MsgWaitForMultipleObjects(
    DWORD    nCount,         //кількість об'єктів у масиві
    LPHANDLE pHandles,       // покажчик на масив
                             // ідентифікаторів
    BOOL  fWaitAll,          // чекати одного або всіх
    DWORD dwMilliseconds,    // тайм-аут
```



```
    DWORD dwWakeMask         // тип події уведення
                             // для очікування
);

DWORD MsgWaitForMultipleObjectsEx(
    DWORD    nCount,         //кількість об'єктів у масиві
    LPHANDLE pHandles,       // покажчик на масив
                             // ідентифікаторів
    DWORD dwMilliseconds,    // тайм-аут
    DWORD dwWakeMask,        // тип події уведення
                             // для очікування
    DWORD dwFlags            // прапор очікування
);
```

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

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

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

3.3. Монітори

Ідея монітора, яку запропонував Б. Хансен і розвинув С. Хоар, ґрунтується на об'єднанні змінних, що описують спільний ресурс, і дій, які визначають засоби доступу до спільного ресурсу [9; 35]. *Монітор – програмний модуль*, що містить змінні та процедури для роботи з ними, причому доступ до змінних можливий *тільки через процедури* монітора.

Монітор – засіб розподілу ресурсів і взаємодії процесів. Це призначення монітора реалізується за допомогою властивостей, якими наділені процедури монітора. Характерна особливість про-

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

Загальна структура монітора:

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

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

Приклад монітора:

```
monitor  Склад;

         Товар: Data;          -- спільний ресурс

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

begin

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


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

3.3.1. Монітори в мові Ада95

Концепцію моніторів у новому стандарті мови Ада95 реалізовано у вигляді спеціальних програмних модулів – захищених модулів (`protected units`) [16; 18; 45]. 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 захищеному 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;
```

Захищенні операції – це:

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

Захищені функції забезпечують доступ тільки до *читання* захищених елементів. Але дозволяють робити це *одночасно* всім про-

цесам автоматичним копіюванням елементів, які зчитуються. Це порушує головну властивість процедур монітора, яка дозволяє знаходитися в моніторі тільки одному процесу, але це "порушення" дозволяє скоротити час доступу до захищених елементів і не має будь-яких наслідків, оскільки зміна даних заборонена і не виконується.

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

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

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

❖ **Приклад 3.1**

```
-----------------------------------------------------------
--   Ада95.Захищений модуль.Специфікація.                --
-----------------------------------------------------------
protected   Контроль  is
   procedure   Включення;          -- захищені підпрограми
   function    Перемкнути(X :  Float);
end  Контроль;

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

protected   Блок232(Номер: in Positive)  is
   entry  Параметри(X: out  integer);
   procedure  ЗмінаПараметра(Y: in  integer);
private
   -- захищений елемент
```



```
   Об'єкт: array(1 .. Номер) of  integer;
end  Блок232;
```

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

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

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

Описання тіла захищеного входу:

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

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

В прикладі 3.2 показано реалізацію тіла захищеного модуля `Сенсор`, специфікацію якого наведено раніше у прикладі 3.1:

❖ Приклад 3.2

```
-------------------------------------------------------------
-- Ада95. 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. 3.6. Захищену функцію Замір_Стану() і процедуру Замір_Стану() зображено справа, захищені входи Чекати() і Сигнал() – зліва. Захищені елементи (Стан, Прп) зображено всередині захищеного модуля в овалі.

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

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

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

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

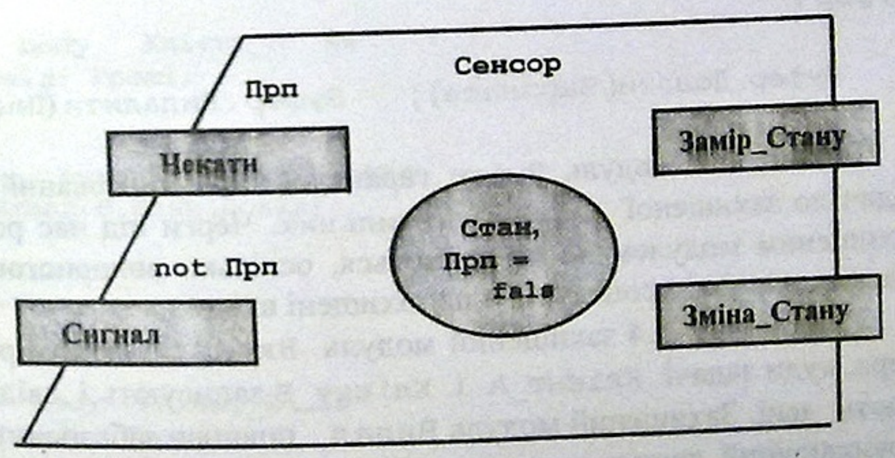


Рис. 3.6. Структура захищеного модуля

❖ Приклад 3.3

```
-------------------------------------------------------------
-- Ада95.Захищений модуль у завданні взаємного виключення --
-------------------------------------------------------------
protected   Буфер    is
    procedure  Додати  (Вклад:  out  Positive);
    procedure  Видалити(Вклад:  out  Positive);
private
```

```
    Лічильник:  Integer:= 0;
end Буфер;

-- тіло захищеного модуля
protected  body  Буфер  is

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

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

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

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

У прикладі 3.4 захищений модуль Вклад виконує роль буфера, куди задачі Клієнт_А і Клієнт_В записують і звідки зчитують дані. Захищений модуль Вклад повинен забезпечити взаємовиключний доступ задач до спільного ресурсу, яким є змінна Рахунок, а також синхронізацію процесів залежно від стану ресурсу.

❖ **Приклад 3.4**

```
-------------------------------------------------------
-- Ада95.Захищений модуль у завданні взаємного виключення --
-- та синхронізації процесів                              --
-------------------------------------------------------
protected  Вклад   is
    entry  В_Банк(М  :  in    Гроші);
```



```
    entry  З_Банку(М :  out   Гроші);
private
    Рахунок: Гроші;                    -- спільний ресурс
    Прапор:  Boolean := False;
end  Вклад;
---------
protected  body    Вклад    is
    entry   В_Банк(М:  in   Гроші)
                            when  Прапор = False  is

    begin
        Рахунок:= М;
        Прапор:= True;
    end  В_Банк;
    entry  З_Банку(М:  out   Гроші)
                            when   Прапор = True  is

        М := Рахунок;
        Прапор := False;
    end  З_Банку;
end Вклад ;
---------
task  Клієнт_А;

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

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

3.3.2. 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 спільного ресурсу в класі-моніторі виконується за допомогою модифікатора private.

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

Приклади синхронізованих методів:

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

❖ **Приклад 3.5**

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

   private int Буфер;          // спільний ресурс

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

Створення екземпляра класу Монітор:

```
Монітор  ZZ = new Монітор();
```

Одночасне виконання методів ZZ.Читати() та ZZ.Писати() не можливо. Якщо процес (потік) B викликав і виконує, наприклад, метод ZZ.Читати(), то інший потік A 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 B.

3.4. Вирішення завдання взаємного виключення

Розглядаються приклади вирішення завдання взаємного виключення. При цьому припускається погодження про те, 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 виконання програми. В цілому питання попереднього копіювання спільних даних треба вирішувати у кожному окремому випадку, виконавши попередній аналіз необхідності створюення копій або відмови від нього у випадку, якщо виконання операцій в критичній ділянці не потребує значного часу.

3.4.1. Вирішення завдання взаємного виключення в мові Ада95

У мові Ада95 рішення завдання взаємного виключення можна здійснити за допомогою прагм Atomic і Volatile, механізму семафорів або механізму моніторів (захищених модулів).

Неподільні змінні. Простіший спосіб вирішення завдання взаємного виключення в мові Ада95 ґрунтується на використанні неподільних 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 і виконуються 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 3.6**

```
-------------------------------------------------------------
-- Ада95.Pragma Atomic у завданні взаємного виключення --
-------------------------------------------------------------
procedure Lab36 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 Lab36;
```

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

Використовуючи загальну схему вирішення завдання взаємного виключення за допомогою семафорів (рис. 3.4) необхідно:

- створити змінну – семафор з початковим значенням 0 або 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().(див. 3.3.1).

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

❖ **Приклад 3.7**

```
-------------------------------------------------------------
--  Ада95.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 Lab37 is

   Буфер: integer:= 10;          -- спільний ресурс
   Sem_Ku: Suspension_Object;    -- семафор

   procedure Старт_Задач is

      task A;

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

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

```
      Set_True(Sem_Ku);

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

   task B;

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

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

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

begin

   null;
end Старт_Задач;

-- основна процедура
begin

   put_line("  Main procedure started ");

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

   Старт_Задач;        -- запуск задач A і B

end Lab37;
```

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

Структуру захищеного модуля Захист показано на рис. 3.7. Модуль включає захищений елемент – змінну Буфер, що буде описана в приватній частині специфікації захищеного модуля. Для реалізації взаємовиключних дій над спільним ресурсом Буфер наведено дві процедури Операція_A() і Операція_B().

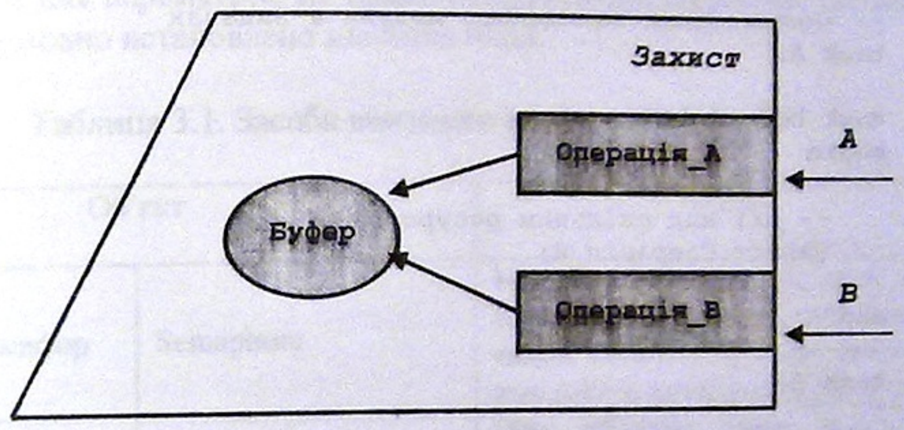


Рис. 3.7. Захищений модуль в завданні взаємного виключення

❖ **Приклад 3.8**

```
-------------------------------------------------------------
-- Ада95.Захищений модуль в                                 --
-- завданні взаємного виключення                            --
-------------------------------------------------------------

procedure Lab38 is
   -- 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 B is
begin
    . . .
    -- дії над спільним ресурсом
    Захист.Операція_B;
    . . .
end B;

-- головна процедура
begin

  null;

end Lab38;
```

3.4.2. Вирішення завдання взаємного виключення у Win32

Бібліотека Win32 містить декілька засобів, які можна використати для вирішення завдання взаємного виключення. Це реалізація механізму семафорів, мютекси та критичні секції (табл. 3.1). 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 3.1. Засоби взаємного виключення Win32

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

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

❖ **Приклад 3.9**

```
--------------------------------------------------------------
--|Ада95.Використання семафорів бібліотеки  Win32            --
--|в завданні взаємного виключення                           --
--------------------------------------------------------------
With Win32, Win32.winbase, Win32.winnt;
Use  Win32, Win32.winbase, Win32.winnt;
procedure Lab39 is

   Ресурс: integer;  -- спільний ресурс
```

```
                    -- (глобальна змінна)

-- HANDLE  змінна для створювання семафора
Sem1: HANDLE;

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

procedure Запуск_Задач is

   task T1;
   task T2;
   task T3;

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



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

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

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

      -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab39;
```

Мютекси. Мютекс (від *mutual exclusion* – взаємне виключення) – засіб, схожий на семафор, але на відміну від семафора він не має значення [45]. Мютекс належить тільки одному процесу і може передаватися від одного процесу до іншого (бути захопленим процесом). Вхід в критичну ділянку дозволяється тільки тому процесу, який теперь володіє мютексом. Після виходу з критичної ділянки процес звільнює мютекс, який захвачується іншим процесом.

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

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

❖ Приклад 3.10

```
-----------------------------------------------------------
--|Ада95.Використання мютексів у бібліотеці  Win32       --
--|в завдвнні взаємного виключення                        --
-----------------------------------------------------------
procedure Lab310 is

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

```
-- HANDLE змінна для створення мютексу
Mute: HANDLE;

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

procedure Запуск_Задач is

   task T1;
   task T2;
   task T3;

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



```
         p1:= ReleaseMutex(Mute);
            . . .
         put_line("Process T3 finished");
      end T3;
   begin

      null;
   end Запуск_Задач;

begin -- основна процедура

   put_line("  Main procedure started ");

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

   -- виклик процедури для запуску задач
   Запуск_Задач;
end Lab310;
```

Критичні секції. Механізм *критичних секцій* – це засіб, який об'єднує конструкції ВХІДКД і ВИХІДКД в *єдиний* оператор і позбавляє від помилок, які властиві семафорам (мютексам), коли на вході в критичну ділянку перевіряється один семафор, а на виході змінюється інший семафор і це ніяк не контролюється. 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 семафора в ньому використовується ім'я критичної ділянки, вхід у яку контролюється і дозволяється тільки одному процесу.

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

У разі використання критичних ділянок роль примітивів ВХІДКД і ВИХІДКД виконують функції `EnterCriticalSection()` та `LeaveCriticalSection()`. На відміну від семафорів і мютексів у механізмі критичних ділянок час очікування в операції входження в критичну ділянку `EnterCritical`

EnterCritical Section() не встановлюється, тобто він завжди є INFINITE.

❖ Приклад 3.11

```
-----------------------------------------------------------
--|Ада95.Використання критичних секцій бібліотеки Win32  --
--|у задачі взаємного виключення                          --
-----------------------------------------------------------
procedure Lab311 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 Lab311;
```

3.4.3. 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 методу), то всі інші потоки, які намагаються викликати будь-який синхронізований метод цього ж ек-

земпляра класу, мають чекати, поки завершиться його виконання. Тобто синхронізований метод має властивості процедури монітора.

Приклад 3.12 ілюструє реалізацію Java монітора, який містить спільний ресурс – змінну Буфер, і два методи Читати() і Писати() для доступу до спільного ресурсу.

❖ **Приклад 3.12**

```
/*-----------------------------------------------------------------
-- 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 блокований, якщо він спробує викликати будь-який метод класу Монітор, наприклад, Блок.Писати(), і зможе продовжити своє виконання тільки після завершення виконання методу Блок.Читати() у потоці В.

Синхронізовані блоки. Загальна форма оператора synchronized:

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

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

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

❖ **Приклад 3.13**

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

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

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

class Доступ  implements Thread{

    Робота Блок1;   // посилання на клас Робота

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

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

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

class Основний{

    public static void main(String [] args){

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

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

    }// main
}// Основний
```

```
      -- Подія2
      -- Сигнал задачі T3 про подію (Подія2) в задачі T2
      p1:= SetEvent(Evn2_3);
         . . .
      put_line("Process T2 finished");
   end T2;
   --------------
   task body T3 is
   begin
      put_line("Process T3 started");
         . . .
      --  Очікування на подію (Подія1) в T1
      Temp:= WaitForSingleObject(Evn1_3,Infinite);
      --  Очікування на подію (Подія2) в T2
      Temp:= WaitForSingleObject(Evn2_3,Infinite);
         . . .
      put_line("Process T3 finished");
   end T3;
begin
   null;
end Запуск_Задач;

begin -- основна процедура
   put_line("  Main procedure started ");
   --  створення подій
   Evn1_3:=  CreateEvent(NULL, 0, 0, NULL);
   Evn2_3:=  CreateEvent(NULL, 0, 0, NULL);
   -- виклик процедури для запуску задач
   Запуск_Задач;
end Lab320;
```

У прикладі 3.21 задача T1 інформує задачі T2 і T3 про подію, що відбулася в задачі T1. Це можливо виконати через одну подію Evn1_23.

❖ **Приклад 3.21**

```
--------------------------------------------------------------
--|Ада95.Використання подій бібліотеки  Win32                --
--|в завданні синхронізації                                  --
--------------------------------------------------------------
procedure Lab321 is

   -- HANDLE  змінна для створювання події
   Evn1_23: HANDLE;
```



```
   -- допоміжні змінні
   Temp: DWORD;
   p1: BOOL;
   procedure Запуск_Задач is

      task T1;
      task T2;
      task T3;

      task body T1 is
      begin
         put_line("Process T1 started");
            . . .
         --  Подія
         --  Сигнал задачам T2 і T3 про подію в задачі T1
         p1:= SetEvent(Evn1_23);
            . . .
         put_line("Process T1 finished");
      end T1;
      ---------------
      task body T2 is
      begin
         put_line("Process T2 started");
            . . .
         --  Очікування на подію в T1
         Temp:= WaitForSingleObject(Evn1_23,Infinite);
            . . .
         put_line("Process T2 finished");
      end T2;
      --------------
      task body T3 is
      begin
         put_line("Process T3 started");
            . . .
         --  Очікування на подію в T1
         Temp:= WaitForSingleObject(Evn1_23,Infinite);
            . . .
         put_line("Process T3 finished");
      end T3;

   begin
      null;
   end Запуск_Задач;

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

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

-- виклик процедури для запуску задач
Запуск_Задач;

end Lab321;
```

3.5.3. Вирішення завдання синхронізації процесів 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, оскільки вони спричинили побічні ефекти.

Методи `wait()`, `notify()` та `notifyAll()` оголошені в класі `Object`, доступні всім класам і мають вигляд:

```
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()` розблоковує потік, який блоковано методом `wait()`. На жаль, мова не визначає, який потік буде розблоковано, якщо потоків декілька. Метод `notifyAll()` розблоковує *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 3.22**

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 Снхр = new Synchro();

    // екземпляри потоків
    Потік1 П1 = new Потік1(Снхр);
    Потік2 П2 = new Потік2(Снхр);

    // старт потоків
    П1.start();
    П2.start();
  }// main
}// Синхронізація потоків
```


➢ **Запитання для модульного контролю**

1. Яки існують 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. У чому полягає віддмінність між семафорами та мютексами в Win32 ?
11. Яку структуру має захищений модуль ? Які ви знаєте види та яке призначення 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. У чому полягає відмінність в реалізації концепції монітора в мовах Java і Ада95 ?

➢ **Завдання для самостійної роботи**

1. Написати програму мовою Java, у 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 мовою Ада95, у 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 (чотири потоки чекають на подію в одному потоці).
5. Завдання 3 реалізувати за допомогою захищеного модулю мови Ада95.
6. Завдання 4 реалізувати за допомогою захищеного модуля мови Ада95 .
7. Виконати моделювання механізму богатозначних семафорів за допомогою захищеного модуля мови Ада95.
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 однієї і тієї же паралельної програми, де задача взаємного виключення вирішається за допомогою різних засобів синхронізації. Визначити також час виконання цієї програми у випадку, коли задача взаємного виключення взагалі не вирішується.

РОЗДІЛ 4.
Взаємодія процесів, що ґрунтується на посиланні повідомлень

4.1. Загальна схема.
4.2. Мова Оккам.
4.3. Ада95. Механізм рандеву.
4.4. Бібліотека PVM.
4.5. 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 .

4.1. Загальна схема

Загальна схема передачі повідомлень між процесами ґрунтується на використанні операцій `Send()` і `Receive()`[37; 42; 45]. За допомогою операції `Send()` процес відправляє дані іншому процесу, який отримує ці дані за допомогою операції `Receive()`:

```
Процес 1                    Процес 2
  . . .                       . . .
Send(Дані);                 Receive(Дані);
```

Існують різноманітні схеми використання операцій `Send()` і `Receive()`: іменовані, синхронні, асинхронні, з блокуванням або без блокування та ін.

Іменована схема потребує застосування імен взаємодіючих процесів в операціях `Send()` і `Receive()`:

```
Send    (Ім'я_Процесу_Одержувача,  Дані)
Receive(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:

```
Send    (Буфер, Дані)
Receive(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 кількома процесами;
- кілька процесів з кількома процесами;
- кілька процесів з одним процесом;
- один процес з одним процесом.

Передача повідомлень може бути також синхронною або асинхронною. Синхронна схема ґрунтується на тому, що процес, який виконує операцію `Send()` або `Receive()`, блокується, поки другий процес не вийде на приймання повідомлення (операція `Receive()`) або приймання повідомлення (операція `Send()`).

Оптимальною з погляду розробника є можливість використання різних видів операцій передавання-приймання повідомлень, як це зроблено в бібліотеці PVM, яка містить набір з кількох видів операцій `Send()` і `Receive()` [2].

4.2. 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 [11; 37; 46]. 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 посилання повідомлень. Оккам реалізує варіант механізму рандеву, який на відміну від рандеву в мові Ада95 є симетричним.

У мові Оккам процеси не іменовані, тому їх взаємодія ґрунтується на використанні спеціальної програмної конструкції – *каналу* (`channel`). Канал має ім'я і може бути використаний для запису даних у канал (операція **!**) або читання даних з каналу (операція **?**):

```
Chan1!X   - писати значення змінної X у канал Chan1
Chan3?Y   - читати значення змінної Y з каналу Chan3
```

Взаємодія процесів через канал (рис. 4.1) 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.

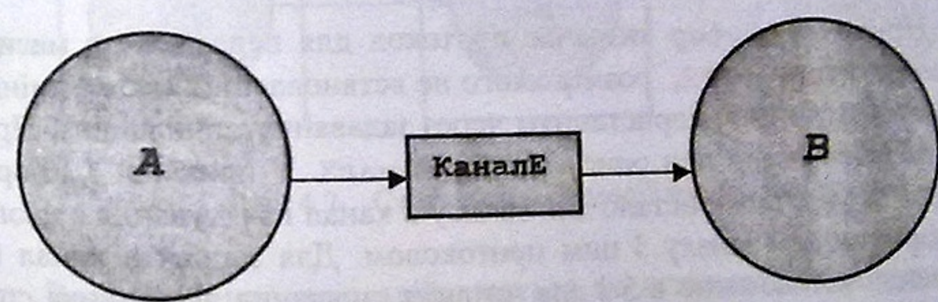


Рис. 4.1. Структура взаємодії процесів в Оккам

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

PAR          --   перший процес
  INT x = 100:

  КаналЕ ! x     -- запис у канал
```

```
PAR      --  другий процес
  INT y:
  КаналЕ ? y     -- читання з каналу
```

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

❖ **Приклад 4.1.** Протокол для передавання масивів різної довжини.

```
PROTOCOL Буфер IS INT::[]BYTE:
CHAN OF  Буфер КаналХ:
[8] BYTE Дані:
INT Розмір:
PAR
   КаналХ ! 5::"HELLO"
   КаналХ ? Розмір::Дані:
```

Протокол Буфер визначає протокол для передавання масиву елементів типу BYTE, розмір якого не встановлений і може змінюватися з кожним використанням через задавання лічильника. Протокол застосовано при опису каналу КаналХ. У прикладі 4.1 протокол Буфер використано для запису в канал і зчитування з нього. Використання каналу з цим протоколом. Для запису в канал це значення встановлено в 5 і для читання значення лічильника спочатку приймається в змінну Розмір.

❖ **Приклад 4.2.** Протокол для передавання даних різного типу.

```
PROTOCOL Порт IS INT;BYTE:
CHAN OF  Порт Регістр:
INT  Переривання:
BYTE Адреса:
PAR
   Регістр ! 500,'0123ОАВ'
   Регістр ? Переривання, Адреса
```


У мові Оккам також існує протокол з варіаціями, який дозволяє передавати по одному каналу повідомлення різного формату.

Розроблення програми мовою Оккам включає етап конфігурування, який пов'язаний з розподіленням процесів по трансп'ютерах (процесорам), а каналів – по лінках. Для цього використовують спеціальні оператори:

```
PLACED PAR
PROCESSOR номер тип.процесора
PLACE канал AT адрес
```

Застосування операторів конфігурування наведено в прикладі 4.3. 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. 4.2.

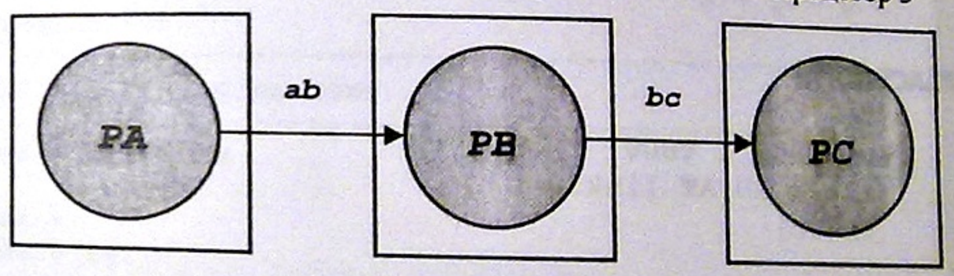


Рис. 4.2. Структура взаємодії процесів в Оккам

Взаємодія процесів здійснюється за допомогою каналів *ab* і *bc*, які використовують протокол Обмін. Для з'єднання процесорів 1 і 2 використано лінк-зв'язки link30out і link0in, а для з'єднання процесорів 2 і 3 – лінк-зв'язки link30out і link0in.

❖ **Приклад 4.3**

```
-- 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)
```

4.3. Ада95. 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 A і B, де задача A передає в задачу B ціле число x. Структурну схему взаємодії задач показано на рис. 4.3. Для взаємодії задач у специфікації задачі B описаний вхід `Data`, специфікація якого дозволяє приймати дані в задачі B. Посилання цілого числа зі задачі A виконано як виклик входу `Data` з підстановкою фактичного параметра.

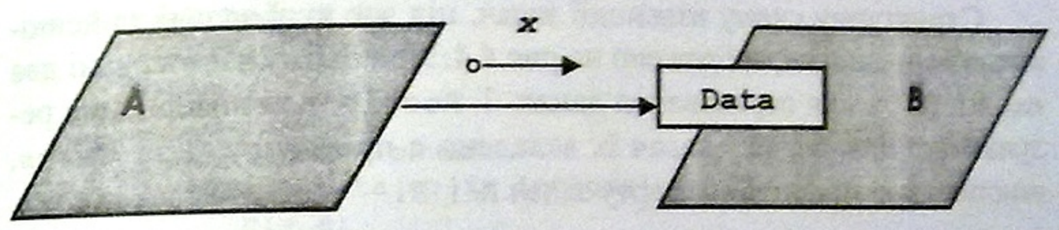


Рис. 4.3. Схема взаємодії задач (одностороння)

❖ Приклад 4.4

```
-------------------------------------------------------------
--  Ада95.Механізм рандеву                                  --
-------------------------------------------------------------
procedure Lab44 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 B
         . . .
   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 B;

            -- основна програма
begin
   null;     -- порожній оператор

end Lab44;
```

Структурну схему взаємодії задач, під час якої задачі здійснюють обмін даними, зображено на рис.4.4. 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.

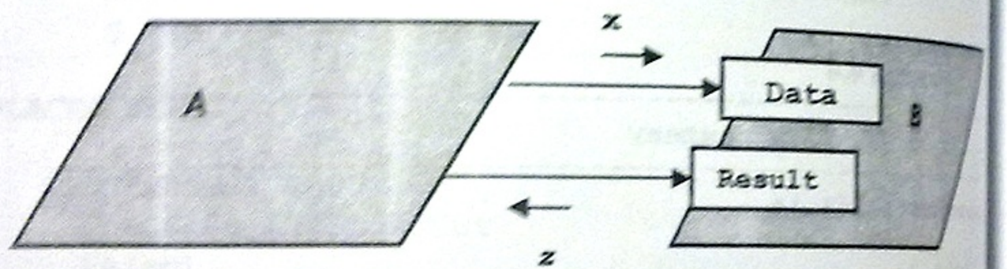


Рис. 4.4. Схема взаємодії задач (двостороння)

❖ Приклад 4.5

```
-----------------------------------------------------
-- Ада95.Механізм рандеву
-----------------------------------------------------
procedure Lab45 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 Lab45;
```

Структурну схему взаємодії трьох задач – T1, T2 і T3 зображено на рис. 4.5. Задача T3 приймає дані від задач T1 і T2 за допомогою входів: DataT1 і DataT2. Особливістю реалізації взаємодії задач є використання в тілі задачі T3 оператора select. Це дозволяє задачі T3 приймати виклик входів DataT1 та DataT2 в будь-якій послідовності в міру їх надходження.

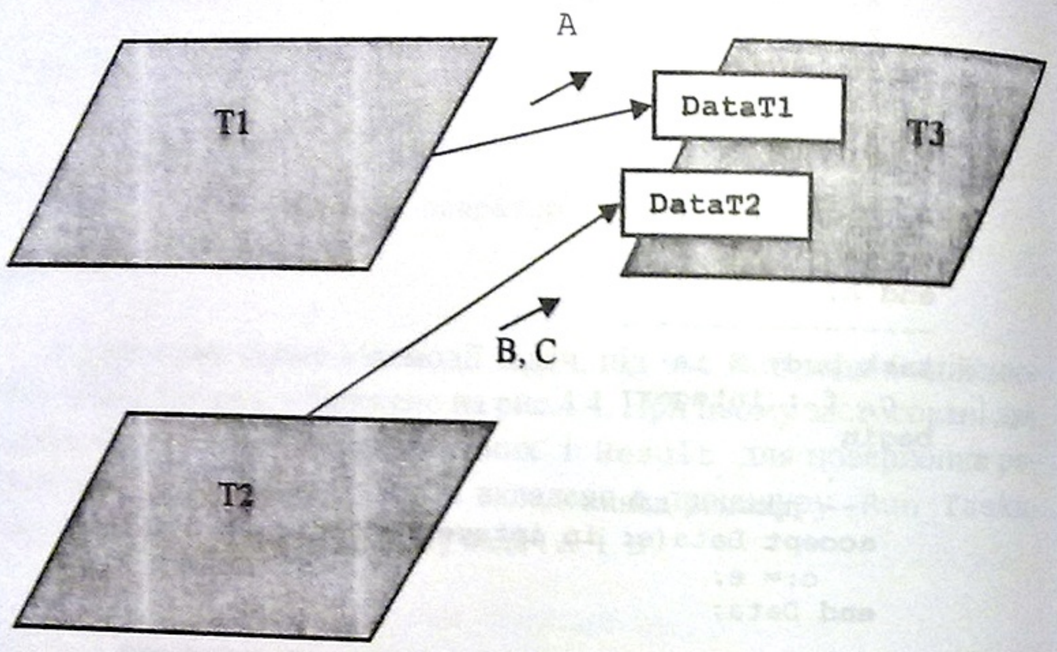


Рис. 4.5. Схема взаємодії задач

❖ Приклад 4.6

```
-------------------------------------------------------------
--  Ада95.Механізм рандеву                                  --
-------------------------------------------------------------
procedure Lab46 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 програма
   Run_Tasks;            -- явний запуск задач T1, T2, T3
end Lab46;
```

4.4. Бібліотека PVM.

Бібліотека (інтерфейс) PVM (Parallel Virtual Machine) визначає стандарт для програмування в комп'ютерних системах з розподіленою пам'яттю [25]. PVM містить набір утиліт і бібліотечних функ-

цій, 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 набуває 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` набуває таких значень:

- *PvmDataDefault* – автоматичне кодування різних даних, які приймаються і передаються;
- *PvmDataRaw* – без кодування даних для запису в буфер;
- *PvmDataInPlacet* – дані будуть упаковані і передані в буфер задачі, що приймає, минаючи буфер задачі, що передає.

Створення нового порожнього буфера для задачі, що передає:

```
int bufid = pvm_mkbuf(int encoding);
```

Знищення буфера з ідентифікатором `bufid`:


```
int info = pvm_freebuf(int bufid);
```

Повертає ідентифікатор активного буфера передавання:

```
int bufid = pvm_getsbuf(void);
```

Повертає ідентифікатор активного буфера приймання:

```
int bufid = pvm_getrbuf(void);
```

Переволить активність з буфера передавання `oldbuf`, на інший буфер передавання `bufid`:

```
int oldbuf = pvm_setsbuf(int bufid);
```

Перемикаає активність з буфера приймання `oldbuf` на інший буфер приймання `bufid`:

```
int oldbuf = pvm_setrbuf(int bufid);
```

Більшість схем взаємодії задач потребують в кожній задачі одного буфера приймання і одного буфера передавання. У цьому випадку достатньо використання тільки функції `pvm_initsend();`

Пакування повідомлень. Пакування даних у буфер передавання здійснюється за допомогою 12 функцій. Три параметри цих функцій:

- покажчик відповідного типу на перший елемент масиву, який пакується;
- розмір масиву;
- кількість елементів вказаного типу, що складає один елемент масиву.

Приклад функції для пакування даних типу `float`:

```
int info = pvm_pkcplx(float *cp,int n, int s);
```

Функції передавання повідомлень. Для *передавання* повідомлення необхідно надати йому ідентифікатор (параметр msgid), а також ідентифікатор задачі (параметр tid), яка приймає повідомлення.

Надає повідомленню ідентифікатор msgid і посилає це повідомлення задачі з ідентифікатором tid:

```
int info = pvm_send(int tid, int msgid);
```

Надає повідомленню ідентифікатор msgid і посилає це повідомлення задачам, які прораховані в елементах ntask масиву tids:

```
int info = pvm_mcast(int *tids, int ntask,
                     int msgid);
```

Пакує nt елементів масиву vd типу tp, надає повідомленню ідентифікатор msgid і посилає повідомлення задачі з ідентифікатором tid:

```
int info = pvm_send(int tid, int msgid,
                    void *vd, int nt, int tp);
```

Функції приймання повідомлень. Для функцій приймання можливі варіації значень параметрів: якщо tid = -1, то приймається будь-яке повідомлення з ідентифікатором msgid, яке надіслане задачі; якщо msgid = -1, то приймається будь-яке повідомлення, яке надіслане задачі від задачі з ідентифікатором tid; якщо tid = msgid = -1, то приймається будь-яке повідомлення від будь-якої задачі, що надіслане цій задачі.

Виконується блоковане приймання повідомлення, тобто очікування повідомлення з ідентифікатором msgid від задачі tid і створює приймальний буфер, посилає повідомлення задачі з ідентифікатором bufid і розміщує в ньому повідомлення, яке надійшло:

```
int info = pvm_recv(int tid, int msgid);
```


Виконується неблоковане приймання повідомлення, тобто якщо повідомлення немає, воно закінчується; якщо повідомлення з параметрами tid і msgid надійшло, то воно розміщується в приймальному буфері, ідентифікатор якого повертається через параметр bufid:

```
int bufid = pvm_nrecv(int tid, int msgid);
```

Виконується блоковане приймання повідомлення, якщо час очікування не перевищує значення tmout (задається у вигляді двох цілих полів – секунд і мілісекунд):

```
int bufid = pvm_trecv(int tid, int msgid,
                      struct timeval *tmout);
```

Виконується перевірка надходження повідомлення. Якщо воно надійшло, то повертається ідентифікатор приймального буфера bufid:

```
int bufid = pvm_probe(int tid, int msgid);
```

Розпакування повідомлень. Функції розпакування даних з приймального буфера аналогічні розглянутим вище функціям пакування для повідомлення, що надсилається.

Приклад функції для розпакування даних типу float:

```
int info = pvm_upkcplx(float *cp,int n, int s);
```

Перелік функцій PVM для пакування даних у повідомленні, яке надсилається, і розпакування повідомлень, що надійшли, наведено в дод. 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 4.7**

```
/*-----------------------------------------------------------------
-- 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).

4.5. Бібліотека MPI.

Бібліотека (інтерфейс) MPI (Message Passing Interface) визначає стандарт для програмування в комп'ютерних системах з розподіленою пам'яттю [2; 25]. 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. Розрізняють процес-відправник і процес-одержувач.

У MPI повідомлення містить такі компоненти:

- блок даних (тип `void*`);
- тип даних (тип `MPI_Datatype`);
- кількість даних (тип `int`, кількість одиниць цього типу в блоці повідомлення.

Використовують таку інформацію про відправника і одержувача:

- комунікатор групи працюючих паралельних процесів (тип
- MPI_Comm);
- ранг відправника (номер процесу-одержувача в комунікаторі тип int);
- ранг одержувача (номер процесу-відправника в комунікаторі, тип int).

Крім того, для організації повідомлення використовують *тег* повідомлення – ціле число (тип int), що надається повідомленню відправником. Одержувач може вказати, що він буде приймати тільки ті повідомлення, які мають цей тег.

Обмін повідомлень має чотири форми:

1) **попарний** – повідомлення посилається одним процесом іншому одному процесу;
2) **колективний** – один процес передає повідомлення всім процесам його групи, що об'єднані за допомогою комунікатора;
3) **асинхронний** – процес-відправник не чекає, поки процес-одержувач отримує повідомлення, яке в цьому випадку копіюється в буфер;
4) **синхронний** – процес-відправник блокується, поки процес-одержувач не отримує повідомлення; процес-одержувач блокується, поки процес-відправник не буде готовий послати повідомлення.

Попарний обмін повідомленнями. Існує чотири види обміну для двох задач: синхронний, асинхронний, з блокуванням, без блокування. У MPI використовують чотири режими обміну (блокована і неблокована форми):

1) стандартне передавання; завершується одразу після відправлення повідомлення;
2) синхронне передавання; не завершується, поки не буде отримане повідомлення;
3) буферизований; завершується одразу же після копіювання повідомлення в системний буфер;

4) у міру готовності; передавання розпочинається тільки тоді, коли адресат ініціює приймання повідомлення.

Домовленість про ім'я функцій обміну двох задач.

Для передавання:

```
MPI_[I] [R,S,B] Send;
```

Префікс [I] означає неблокований режим; [R] – у міру готовності; [S] – синхронізований; [B] – буферизований. Стандартний обмін не потребує жодного префікса. Таки чином, існує вісім різновидів передавання повідомлень.

Для приймання:

```
MPI_[I] Recv;
```

Обмін із блокуванням процесів. Процес, який виконав посилання повідомлення блокується, поки інший процес не отримає це повідомлення. Процес, який приймає повідомлення, також буде блокований, поки він не отримує це повідомлення.

Надіслати повідомлення можна за допомогою функції **MPI_Send()**:

```
int MPI_Send(void *buffer, int count,
    MPI_Datatype tp, int rg, int tag, MPI_Comm cm);
```

де:

buffer – адреса буфера з даними;
count – кількість елементів у буфері;
tp – тип даних для елементів у буфері;
rg – ранг одержувача в групі;
tag – тег повідомлення;
cm – комунікатор.

Отримати повідомлення можна за допомогою функції **MPI_Recv()**:

```
int MPI_Recv(void *buffer, int count,
    MPI_Datatype tp, int rg, int tag, MPI_Comm cm,
    MPI_Status *st);
```

де:

- `buffer` – адреса буфера, де слід розмістити отримане повідомлення;
- `count` – максимальна кількість елементів у буфері;
- `tp` – тип даних для елементів у буфері;
- `sr` – ранг відправника в групі,
- `tag` – тег повідомлення;
- `cm` – комунікатор;
- `st` – інформація про отримання повідомлення.

Обмін без блокування процесів. Операції передавання (приймання) повідомлень виконуються без блокування. Тобто передавальний процес відправляє повідомлення і не чекає на його отримання іншим процесом.

Для неблокованої взаємодії використовують відповідно функції `MPI_ISend()` і `MPI_IReceive()`.

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 у комунікаторі (групі):

```
int MPI_Comm_size(MPI_Comm cm, int *res);
```

де:

- `cm` – коммунікатор;
- `res` – покажчик на результат (кількість процесів).

Номер (ранг) процесу в комунікаторі (групі):

```
int MPI_Comm_rank(MPI_Comm cm, int *rank);
```

де:

- `cm` – комунікатор;
- `rank` – покажчик результату (ранг).

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 (табл. 4.1);
root – ранг одержувача в групі cm;
cm – комунікатор.

Таблиця 4.1. Деякі функції бібліотеки MPI

Функція	Операція
MPI_Send()	Переслати повідомлення. Блокуватися, поки воно не буде отримано
MPI_Recv()	Отримати повідомлення. Блокуватися, поки воно не буде надіслано
MPI_ISend()	Переслати повідомлення без блокування
MPI_IRecv()	Отримати повідомлення без блокування
MPI_Probe()	З'ясувати, чи отримано повідомлення (блокуватися до отримання повідомлення)
MPI_IProbe()	З'ясувати, чи отримано повідомлення (без блокування)
MPI_Bcast()	Надіслати повідомлення всім процесам в групі
MPI_Reduce()	Отримати дані від усіх процесів у групі
MPI_Gather()	Зібрати дані від усіх процесів
MPI_Scatter()	Розсилати дані усім процесам у групі. Дані – одного розміру
MPI_Scatterv()	Розсилати дані усім процесам у групі. Дані можуть бути різного розміру
MPI_Alltoall()	Розсилати дані усім процесам в групі зі всіх процесів групи. Дані одного розміру
MPI_Alltoallv()	Розсилати дані усім 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. 4.2.

Таблиця 4.2. 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 у 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 у групі можна виконати за допомогою функції **MPI_Barrier()**:

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

де cm – комунікатор.

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

❖ **Приклад 4.8**

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 );

         prinf("%s\n", Повідомлення);

      }/* for */

   }

   /* Завершення роботи  MPI */
   MPI_Finalize();

   return 0;

}/* main */
```

➢ **Запитання для модульного контролю**

1. Яка загальна схема концепції взаємодії процесів через посилання повідомлень ?
2. Назвіть види операцій `send/receive`.
3. Як здійснюється синхронізація процесів під час передавання повідомлень ?
4. Що включає поняття каналу і протоколу у мові Оккам ?
5. Як оголошується канал у мові Оккам ?
6. На чому ґрунтується механізм рандеву в мові Ада ?
7. Що включає опис входу задачі ?
8. Як здійснюється взаємодія задач під час рандеву ?
9. Що може бути джерелом тупику в механізмі рандеву ? Наведить приклади виникнення тупику.
10. Як синхронізуються задачі під час виконання рандеву Ада?
11. Яке призначення та які види оператора `select` у мові Ада? Наведіть його аналог у мові Оккам.
12. Яким чином визначаються обсяг та напрям передавання даних в механізмі рандеву мови Ада95 ?
13. У чому полягає відмінність між реалізацією механізму рандеву у мовах Оккам і Ада ?
14. Які дії в PVM потрібні для відправлення повідомлення ?
15. Які дії в MPI пов'язані з відправленням повідомлення ?

➢ **Завдання для самостійної роботи**

1. Розробити програму мовою Ада, яка реалізує взаємодію трьох задач T1, T2 і T3. (Задача T1 передає в T2 два цілі числа x і z, одне з яких (x) потім передається в задачу T3).
2. Опрацювати завдання 1 з метою додаткового передавання із 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 розсилання вектора з однієї задачі в чотири інші.
8. Бібліотека PVM. Забезпечити передавання з однієї задачі в три інші задачі двох векторів типу `char` і `float`.

РОЗДІЛ 5.
Розподілені обчислення

5.1. Організація розподілених обчислень.
5.2. Java. Сокети.
5.3. Java. RMI.
5.4. Ада95.RPC.

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

5.1. Організація розподілених обчислень

Організація обчислень у розподілених комп'ютерних системах (розподілених обчислень) грунтується на мережевих технологіях, що визначає її особливість для використанні моделя посилання повідомлень.

Основна проблема, яка пов'язана з організацією обчислень 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 і засоби такої організації. Якщо Інтернет (який є прикладом розподіленої системи) не потребує надшвидкого передавання даних, то *кластерні розподілені системи*, які орієнтовані виключно на обчислення 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 мережевих адаптерів і комутаторів.

Програмне забезпечення, потрібне для програмування в розподілених комп'ютерних системах, ґрунтується на засобах, які тісно пов'язані з особливостями передавання даних у комп'ютерних

мережах. Такі засоби традиційно представлені у вигляді спеціальних бібліотек (Win32, PVM, MPI). Більш пізні мови програмування (Java, Ада95) 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. [8; 25; 42; 48].

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

Найпоширеніша для розроблення розподілених 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 задання, визначенням у ньому серверної і клієнтської частин, розроблення алгоритмів сервера і клієнта та організації взаємодії сервера і клієнта з використанням належних бібліотечних або мовних засобів.

5.1.1. Сокети

Сокети – низькорівневий механізм, що застосовується для організації розподілених обчислень. Парадигма сокетів була запропонована під час створення ОС UNIX у Каліфорнійському університеті в Берклі [42].

Сокет – своєрідний мережевий рознім 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. 5.1). Для створення сокета застосовують порт. *Порт* – абс-

практне 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. 5.2). При цьому оптимальний серверний додаток має бути реалізованим у вигляді кількох процесів, 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, неведено в табл. В.3 (дод. В.).

5.1.2. 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 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 рандеву полягає в тому, що останній потребує реалізацію програми сервера у вигляді активного модуля – процесу, в

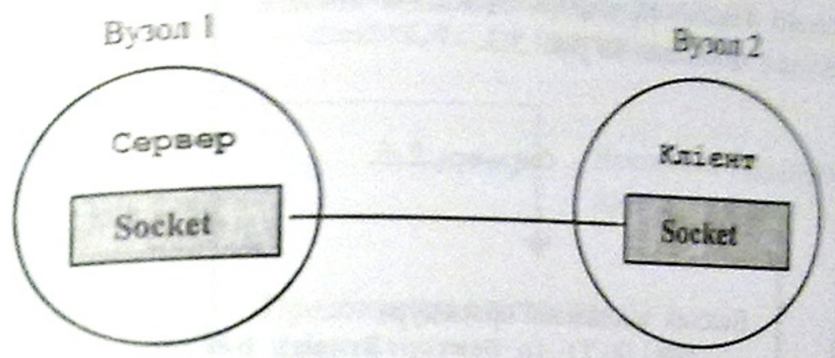


Рис. 5.1. Взаємодія вузлів за допомогою сокетів

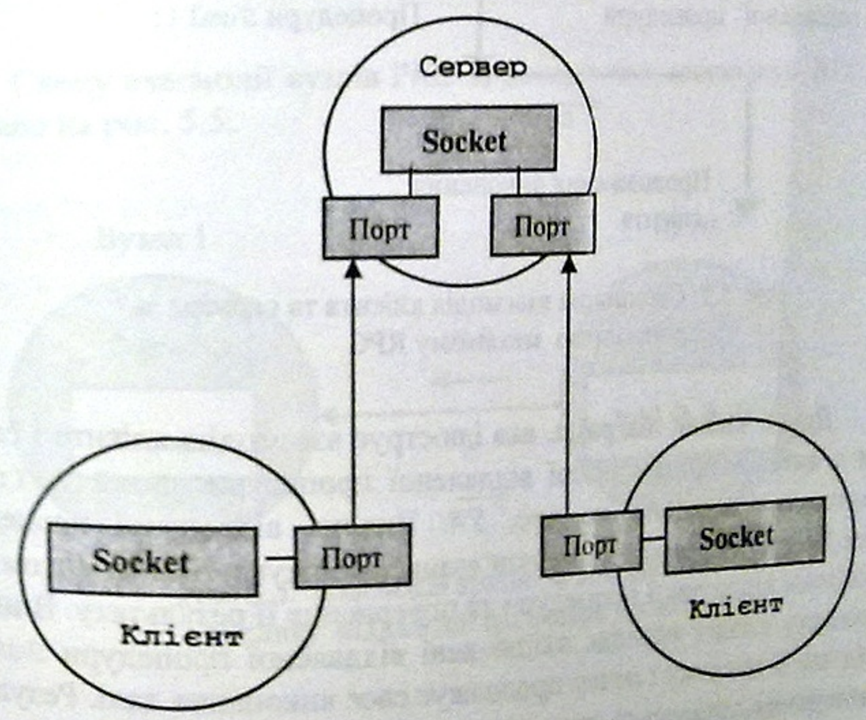


Рис. 5.2. Порти у взаємодії вузлів за допомогою сокетів

Часову діаграму, яка ілюструє взаємодію клієнта і сервера з використанням віддаленої процедури `Sum1(X,Y: in Вектор;Z: out Вектор)` показано на рис. 5.3.

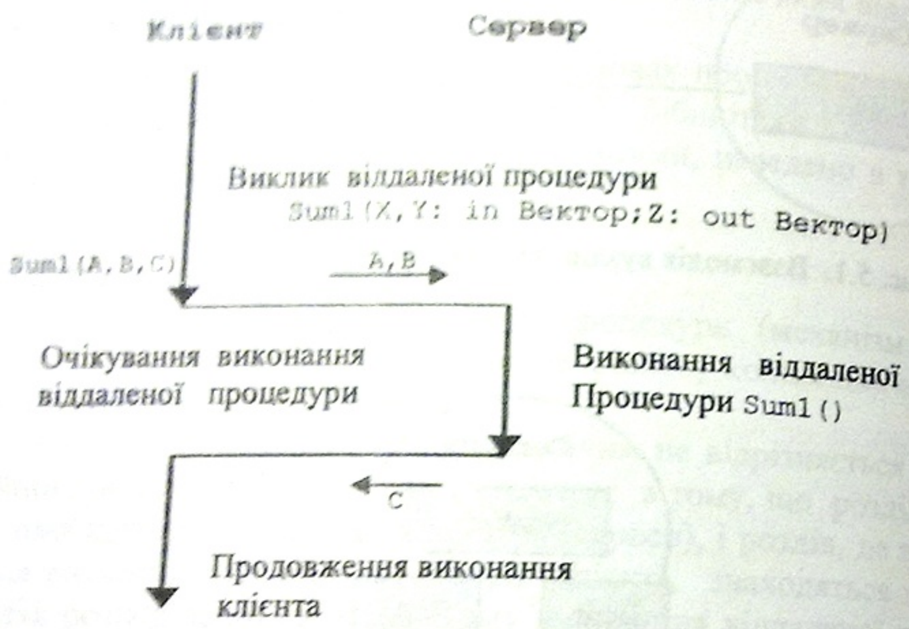


Рис. 5.3. Синхронна взаємодія клієнта та сервера за допомогою механізму RPC

Друга часова діаграма, яка ілюструє взаємодію клієнта і сервера з використанням іншої віддаленої процедури `Sum2(X,Y: in Вектор)`, показано на рис. 5.4. Виклик віддаленої процедури `Sum2()` асинхронний і не пов'язаний з очікуванням клієнтом виконання віддаленої процедури та повернення її результату. В цьому випадку клієнт передає вхідні дані віддаленої процедури `Sum2()` під час її виклику і потім продовжує своє виконання далі. Результат виконання віддаленої процедури `Sum2()` можна отримати від сервера пізніше за допомогою виклику іншої віддаленої процедури. Слід звернути увагу на то, що віддалена процедура, яка допускає асинхронний виклик, має бути описана як асинхронна і вона не *може мати вихідних параметрів* для повернення результату.

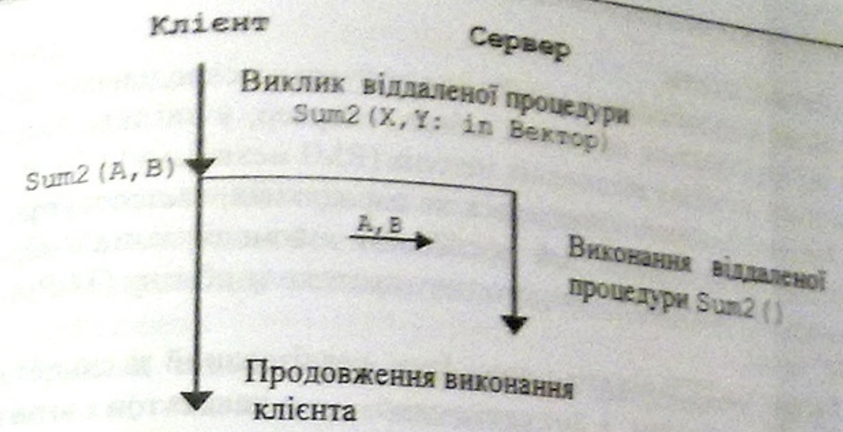


Рис. 5.4. Асинхронна взаємодія клієнта та сервера за допомогою механізму RPC

Схему взаємодії вузлів РКС за допомогою механізму RPC показано на рис. 5.5.

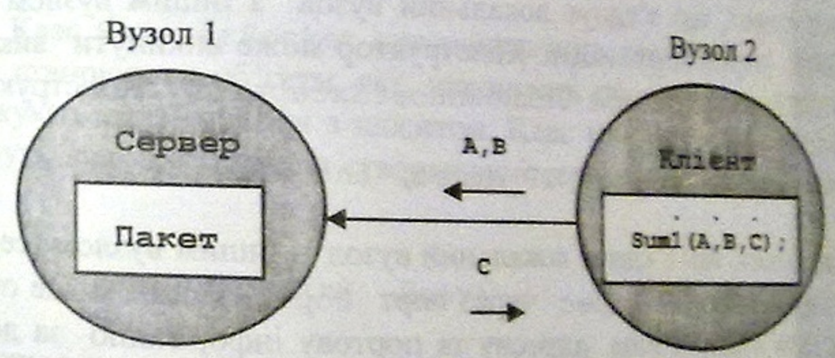


Рис. 5.5. Взаємодія вузлів за допомогою виклику віддаленої процедури `Sum1`

Серверний вузол (`Вузол 1`) містить пасивний модуль `Пакет`, у якому описано віддалену процедуру `Sum1()`. Клієнтський вузол (`Вузол 2`) містить активний модуль – процедуру `Клієнт`, у якому здійснюється виклик віддаленої процедури `Sum1()`.

Механізм RPC реалізовано багатьма мовами програмування, наприклад, мовами Java та Ада95.

5.2. Java.Сокети

Мова Java забезпечує засоби для побудови розподілених додатків, які ґрунтуються на моделі клієнт – сервер, у вигляді сокетів і механізму виклику віддалених методів (RMI механізму) [26; 41].

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

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

▪ Клас `Socket` використовують для клієнтів. Він дозволяє `Socket` об'єктам з'єднуватися із сервером і здійснювати обмін даними. 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 з іншим вузлом (сервером) 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 вхідним або вихідним потоком введення–виведення (`InputStream`, `OutputStream`).

Приклад створювання сокета і його застосування для запису і читання з потоків введення-виведення:

```
Socket  Сокет1  =  new Socket("126.01.23.18",80);
InputStream  in =  Сокет1.getInputStream();
OutputStream os =  Сокет1.getOutputStream();
```

Сокет потрібно закрити після завершення роботи з ним:

```
Сокет1.close();
```

▪ Клас `SereverSocket` використовують для серверів. Він дозволяє створювати об'єкти, які виконують прослуховування порту й очікують на з'єднання з клієнтом. 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 метод `accept()`, який дозволяє серверу блокуватися і чекати на підключення клієнта, після чого повертає `Socket` об'єкт, який буде використаний для зв'язку з клієнтом.

❖ Приклад створення та застосування серверного сокета:

```
try{
   Socket s = null;    // Socket об'єкт
   ServerSocket Сервер3 = new ServerSocket(65,18);
   }catch(IOException e){
}
//прослухування порту з очікуванням підключення клієнта,
//отримання Socket об'єкта
   s = Сервер3.accept();
```

Серверний сокет `Сервер3` підключено до порту з номером 65, до якого тепер може підключитися клієнт, використавши об'єкт `Socket` з належною IP адресою сервера і номером порту 65.

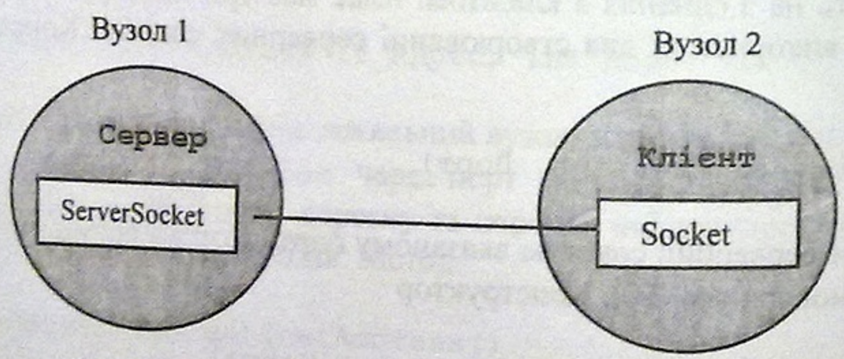


Рис. 5.6. Взаємодія вузлів за допомогою сокетів мови Java

По встановленню зв'язку між клієнтом і сервером здійснюється передавання даних за допомогою потоків введення-виведення і методів `getInputStream()` та `getOutputStream()` класу `Socket`.

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 Java існують класи `DatagramPacket` і `Datagram Socket`.

❖ **Приклад 5.1.**

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 адреса в клієнтському сокеті формується за допомогою методу `getLocalHost()`. В сервери використається порт з номером 65.

```
//-----------------------------------------------------------
//  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;
```

```
      ServerSocket  сервер = null;
      Socket  серверСокет  = null;
      OutputStream zz      = null;

      try{
         сервер  = ServerSocket(65, 10);
         серверСокет = сервер.accept();// прослуховування
         System.out.println("З'єднання на сервері");

         zz = серверСокет.getOutputStream();
         zz.write(x);                    // запис в сокет
      } catch(IOException e){  }
   }// run
}// Сервер

// Клієнт
class Клієнт implements Runnable{

   public void run(){
      int y;

      Socket  клієнтСокет  = null;
      InputStream  rr      = null;

      try{
         клієнтСокет  =  new Socket
                 (InetAddress.getLocalHost(), 65);

         rr = клієнтСокет.getInputStream();
         System.out.println(" З'єднання на клієнті");

         y = rr.read();// читання із сокета
         System.out.println(" y = " + y);

         } catch(UnknowHostException e){  }
      } catch(IOException e){  }

   }// run
} // Клієнт
```

5.3. 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 створення чотирьох класів:

1) віддаленого інтерфейсу, який розширює інтерфейс `Remote` з пакета `java.rmi` і в якому визначено віддалені методи;
2) класу, який є розширенням класу `UnicastRemoteObject` і реалізацією віддаленого інтерфейсу;
3) класу-сервера 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 сервер, та інформацію про сервер (IP – адреса або ім'я) і може викликати віддалені методи сервера.

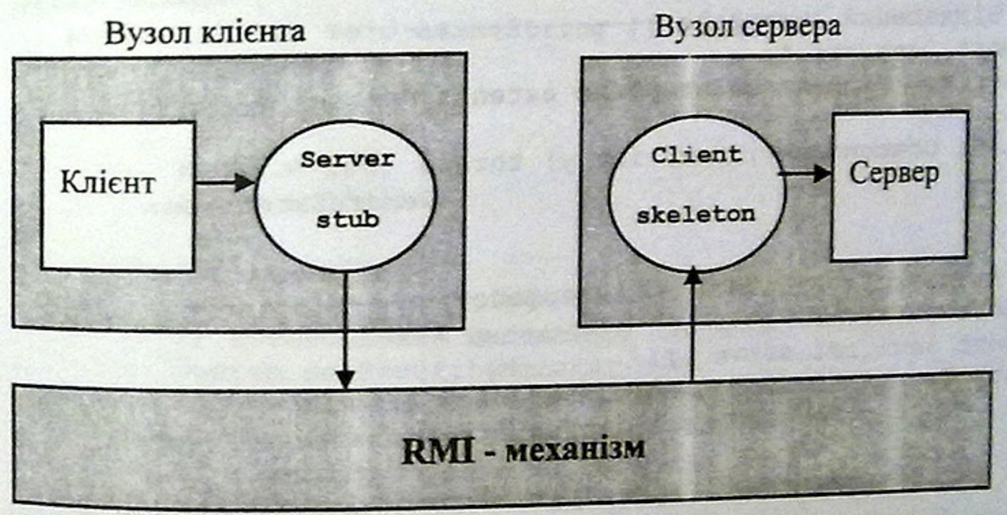


Рис. 5.7. Взаємодія сервера і клієнта в RMI під час виклику віддаленого методу

Взаємодія клієнта та сервера здійснюється за допомогою серверної заглушки (`server stub`) і клієнтського скелетона (`client skeleton`) (рис. 5.7). 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 віддалений метод на серверному вузлі.

❖ **Приклад 5.2**

Приклад ілюструє взаємодію за допомогою механізму RMI класів Сервер і Клієнт, яка полягає в тому, що клас Клієнт знаходить додаток значення змінних X і Y за допомогою використання віддаленого методу Обчислення() класа Сервер.

```
//-----------------------------------------------------------
//  Java.Застосування RMI механізму                          --
//-----------------------------------------------------------

// Віддалений інтерфейс -------------------
import java.rmi.*;
interface ВіддаленийІнтерфейс extends Remote{

   int Обчислення(int x, int y) throws
                                   RemoteException;
}

// 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

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

   public static void main(String [] args){

      try{
```



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

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

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


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

      try{

         int a = 100;
         int в = 150;

         int resoult;

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

         // отримати віддалений об'єкт
         ВіддаленийІнтерфейс  ві = new
                  (Віддалений інтерфейс)Naming.lookup();

         // виклик віддаленого методу
         Resoult = ві.Обчислення(a,в);

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

}// Клієнт
```

Кибернетика и нейрокибернетика мышления

5.4. Ада95. RPC

Механізм RPC лежить в основі моделі розподілених обчислень, яку запропоновано в другому стандарті мови Ада95 у додатку «Розподілені системи» (Annex E "Distributed Systems" - DSA) [58]. Додаток визначає поняття розподіленої комп'ютерної системи і розподіленої програми.

Розподілена комп'ютерна система – набір *вузлів* (*nodes*), які можуть зберігати й 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 конфігуровані 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 реалізовано в Ада95 за допомогою визначення в розділі віддалених процедур, до яких можливий віддалений доступ з інших розділів.

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

```
pragma  Shared_Passive();
pragma  Remote_Type();
pragma  Remote_Call_Intertface()
```

Приклад пакета Форт , у якому описано віддалену процедуру:

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


Застосовано прагму Remote_Call_Interface, яка визначає всі ресурси пакета Форт як такі, що допускають віддалений доступ.

❖ **Приклад 5.3**

Взаємодія двох розділів програми з використанням віддаленої процедури. Перший розділ містить пакет Ресурс, у якому описано віддалену процедуру Додавання_Матриць(). Другий розділ містить процедуру Клієнт, яка виконує віддалений виклик процедури Додавання_Матриц().

```
-----------------------------------------------------------
-- Ада95. 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 Додавання_Матриць;

end  Ресурс;
-----------
with Ресурс;
procedure Клієнт is
```

Архитектоника мышления и нейроинтеллект

```
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
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 її вхідних 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 має місце асинхронний виклик віддаленої процедури. В Ада95 асинхронний виклик описується за допомогою прагми Asynchronous.

Модифікація пакета Форт для забезпечення асинхронного виклику процедури P(...):

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

❖ **Приклад 5.4**

Це модифікація прикладу 5.3, 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.5.6.

```
-------------------------------------------------------------
-- Ада95. 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

   MT: Матриця; -- локальна змінна

    -- реалізація віддалених процедур
   procedure Додавання_Матриць(MB :  in   Матриця;
                               MC :  in   Матриця) is

   begin
```

```
      for i in 1..N loop
         for j in 1 .. N loop
            MT(i)(j):= MB(i)(j)  + MC(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
   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 (обючислення на сервері)
   Ресурс.Додавання_Матриць(MX,MY);
         . . .
   --  віддалений виклик (отримання результату)
   Ресурс.Отримати_МА(MZ);

   --  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 Ада95. Вони визначаються конкретною реалізацією DSA. Система GLADE – приклад такої реалізації. Вона працює з компілятором мови Ада95 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, грунтується 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), адресу вузла, на якому буде виконуватися кожний розділ і т.ін.

Приклад файлу конфігурації для програми з прикладу 5.6:

```
-- 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;
```

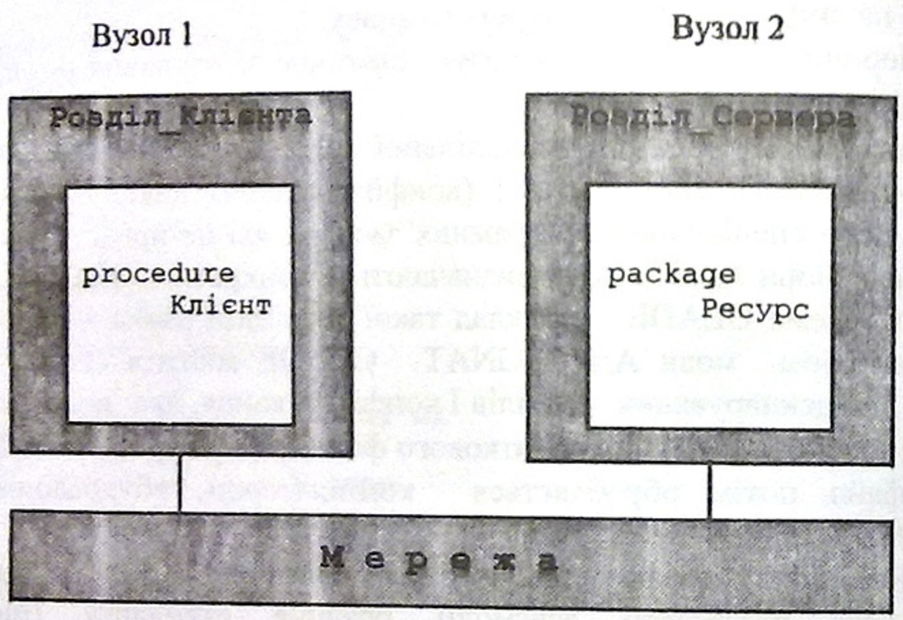


Рис. 5.8. Формування розділів

У формуванні розділів (рис. 5.8) 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 Ада95 наведено в роботі [56].

➢ **Запитання для модульного контролю**

1. У чому полягає особливість програмування для розподілених систем ? Які особливості має модель клієнт – сервер ?
2. Види сокетів, які використовуються в мові Java ?
3. Які функції виконує метод `accept()`?
4. Як у методі запису в потік визначається зв'язок з потрібним сокетом ?
5. Де здійснюється виконання віддаленої процедури ? Які дії виконуються піл час виклику віддаленої процедури ?
6. Які файли треба створити для RMI- додатку ?
7. Які поняття і призначення віддаленого інтерфейсу в RMI?
8. Яке призначення прагм категорії в механізмі RPC ? Як формується розділ ?
9. У чому полягає особливість RPC механізму мови Ада95 під час переходу від звичайної моделі до розподіленої ? Як це здійснюється ?
10. Ада RPC. Які складові має мова конфігурування ?

➢ **Завдання до самостійної роботи**

1. Розробити алгоритми сервера та клієнта для реалізації операції множення вектора на матрицю.
2. Розробити програму сортування вектора у розподіленій системі. Застосувати сокети (Java або Win32).
3. Розробити програму множення матриць у розподіленій системі. Застосувати MPI.
4. Розробити програму множення матриць у розподіленій системі. Застосувати PVM.
5. Розробити програму множення матриць у розподіленій системі. Застосувати RMI в Java.
6. Розробити програму множення матриць у розподіленій системі. Застосувати RPC Ада95.

РОЗДІЛ 6.
Приклади програмування

6.1. Програмування для комп'ютерних систем зі спільною пам'яттю.
6.2. Програмування для комп'ютерних систем з локальною пам'яттю.
6.3. Програмування для розподілених комп'ютерних систем.

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

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

6.1. Програмування для комп'ютерних систем зі спільною пам'яттю

6.1.1. Ада95. Семафори

Реалізація операції додавання векторів у двопроцесорній системі зі спільною пам'яттю з використанням семафорів мови Ада95.

Вхідні дані:

- комп'ютерна система зі спільною пам'яттю містить два процесори і два пристрої введення-виведення (ПВВ) (рис. 6.1),
- математичне завдання: операція додавання векторів,

$$A = B + C * x,$$

де A, B, C – вектори розміру N; x – скаляр,

- введення вектора B і скаляра x виконується в процесорі 1, введення вектора C – у процесорі 2, виведення результату вектор A – у процесорі 1.

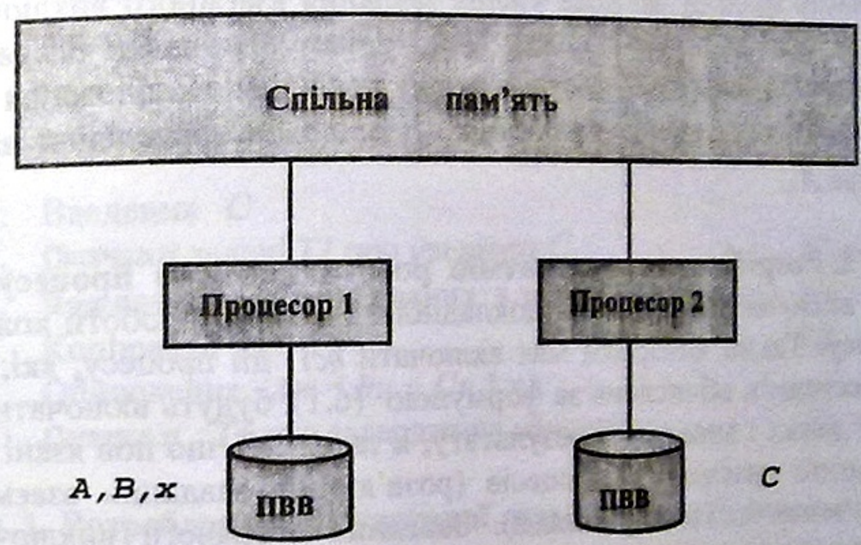


Рис. 6.1. Структурна схема двопроцесорної системи

Етап 1. Побудова паралельного алгоритму. Операція додавання векторів являє собою приклад ідеально паралельної операції. Паралельний алгоритм можна подати у вигляді

$$A_н = B_н + C_н * x \qquad (6.1)$$

де $A_н$ – H елементів вектора A, ($H = N/2$)

Співвідношення (6.1) визначає дії в кожному процесорі КС під час виконанні обчислень.

Потрібно проаналізувати співвідношення (6.1) 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 рівнянні (6.1) є скаляр x, відносно якого необхідно розв'язати завдання взаємного виключення. Можливі два варіанти її розв'язування: перший – виконанням обчислень (6.1) у критичної ділянці, другий – за допомогою попереднього створення копій змінної x (локальні змінні $x1$ і $x2$). Саме копіювання має виконуватися в критичній ділянці і розглядатися як завдання взаємного виключення.

Вибір підходу до розв'язання завдання взаємного виключення ґрунтується на порівняльному аналізу часу виконання операцій в критичних ділянках. У цьому випадку послідовне копіювання змінної x потребує менше часу, ніж послідовне обчислення частин вектора $A_н$.

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 (6.1), будуть включати введення даних і виведення результату, а також дії, що пов'язані з організацією взаємодії процесів (розв'язання завданнь взаємного виключення та синхронізації). Завдання взаємного виключення буде пов'язано з копіюванням змінної x, задачі синхронізації – із синхронізацією з :

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

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

Задача *T1*

		Точки синхронізації
1. Введення *B* і *x*		
2. **Сигнал** задачі *T2* про введення *B* та *x*	--	*S2-1*
3. **Чекати** на введення *C* у задачі *T2*	--	*W2-1*
4. Копіювати *x1: = x*	--	Критична ділянка (КД)
5. Обчислення *Ан = Вн + Сн * x1*		
6. **Чекати** на завершення обчислень в *T2*	--	*W2-2*
7. Виведення результату *A*		

Задача *T2*

1. Введення *C*		
2. **Сигнал** задачі *T1* про уведення *C*	--	*S1-1*
3. **Чекати** на уведення даних в задачі *T1*	--	*W1-1*
4. Копіювати *x2: = x*	--	КД
5. Обчислення *Ан = Вн + Сн * x2*		
6. **Сигнал** *T1* про завершення обчислень	--	*S1-2*

Етап 3. Розроблення структурної схеми взаємодії задач. Структурна схема взаємодії задач базується на алгоритмах задач і дозволяє наочно контролювати зв'язок належних точок синхронізації (***W*** і ***S***), а також ув'язати їх з реальними семафорами, які використовуються потім у програмі [53]. Графічне зображення взаємодії задач дозволяє виявити тупикові ситуації в програмі у випадку, коли точка синхронізації ***W*** не буде пов'язана з належною точкою синхронізації ***S***. Крім того, на структурній схемі уводяться також семафори, що відповідають за роботу зі спільними ресурсами, а потім використовуються в програмі.

На структурній схемі взаємодії задач (рис. 6.2) уведено такі семафори:

- ***Skd*** – для керування доступом до спільного ресурсу (x);
- ***Sem1*** – для синхронізації із завершення введення в *T1*;
- ***Sem2*** – для синхронізації із завершення введення в *T2*;
- ***Sem3*** – для синхронізації із завершення обчислень в *T2* і виведення результатів.

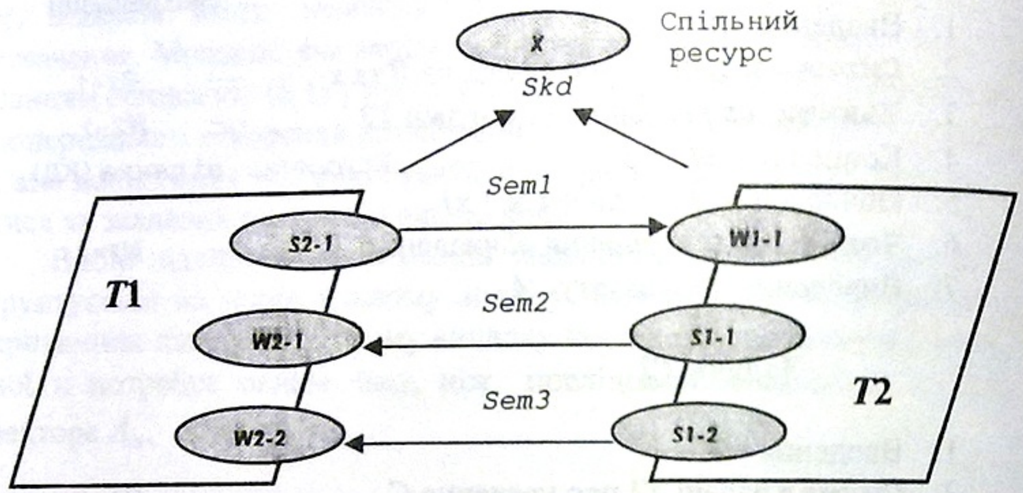


Рис. 6.2. Структурна схема взаємодії задач

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

```
-----------------------------------------------------------
--|Ада95.Використання семафорів                            --
--|Операція  A = B + C * x                                 --
-----------------------------------------------------------
procedure Lab61 is

   N: integer := 100;  -- розмір векторів
   P: integer := 2;    -- кількість процесорів
   H: integer := N/P;  -- розмір підвектора

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

   type Вектор is array (1.. N) of integer;

   A,B,C : Вектор;            -- глобальні змінні

      -- семафори
      Sem1, Sem2, Sem3, Skd:  Suspension_Object;

      procedure  Запуск_Задач  is

         task T1;
```



```
      task body T1 is
         x1: integer;          -- локальна змінна
      begin
         put_line("Process T1 started ");
         -- уведення B та x
         for i in 1.. N loop
            B(i):= 1;
         end loop;

         x:= 2;

         -- сигнал задачі T2 про введення B та x
         Set_True(Sem1);

         --  чекати на введення даних в задачі T2
         Suspend_Until_True(Sem2);

         -- копіювати  x в x1
         Suspend_Until_True(Skd);
            x1: = x;            -- критична ділянка
         Set_True(Skd);

         -- обчислення
         for i in 1 .. H loop
            A(i):= B(i) + C(i)*x1;
         end loop;

         -- Чекати на завершення обчислень в T2
         Suspend_Until_True(Sem3);

         -- виведення результату
         put_line(" A = ");
         for i in 1 .. N loop
            put(A(i),3);
         end loop;

         put_line("Process T1 finished");
      end T1;
      -------
      task T2;
      task body T2 is
         x2: integer;          -- локальна  змінна
      begin
         put_line(" Process B started ");
         -- уведення C
         for i in 1.. N loop
```

```
   C(i):= 2;
end loop;

-- сигнал задачі T1 про уведення C
Set_True(Sem2);

--  чекати на  уведення даних в задачі T1
Suspend_Until_True(Sem1);

--копіювати  x  в x2
Suspend_Until_True(Skd);
   x2: = x;          -- критична ділянка
Set_True(Skd);

-- обчислення
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 ");

   Set_True(Skd);  -- початкове значення семафора (true)

   Запуск_Задач;

end Lab61;
```

6.1.2. Win32. Семафори і мютекси

Реалізація операції додавання векторів у двопроцесорній системі зі спільною пам'яттю з використанням бібліотеки Win32. Для прикладу, що розглянутий в підрозділ. 6.1.1, для синхронізації процесів використовуємо семафори, а для взаємного виключення – мютекси.

Етапи 1– 3 не будуть відмінними від етапів, що розглянуті в прикладі 6.1.1. На структурній схемі (рис.6.2) семафор Skd, буде змінений на мютекс Mut.

Увага! При використанні функцій Win32 в системі програмування ObjectAda потрібне ручне підключення бібліотеки Win32lib для створення Ада-проекту і підключення до програми пакетів Win32, Win32.Base, Win32.Win32NT.

Етап 4. Розроблення програми

```
-----------------------------------------------------------
--|Ада95.Використання  бібліотеки  Win32.                --
--|Застосування семафорів та мютексів                    --
--|Операція A = B + C * x                                --
-----------------------------------------------------------
procedure Lab62 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");

         -- введення B і x
         for i in 1.. N loop
```

```
      B(i):= 1;
   end loop;

   x:= 2;

   -- сигнал задачі T2 про введення B і x
   p1 := ReleaseSemaphore(Sem1,1,NULL);

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

   -- копіювання  x в x1
   Temp:= WaitForSingleObject(Mut,Infinite);
      x1: = x;                -- критична ділянка
   p1:= ReleaseMutex(Mut);

   -- обчислення
   for i in 1 .. H loop
      A(i):= B(i) + C(i)*x1;
   end loop;

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

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

   --  чекати на введення B та x  у задачі T1
```



```
      Term := WaitForSingleObject(Sem1,Infinite);

      -- Копіювати  x  у x2
      Term := WaitForSingleObject(Mut, Infinite);
         x2: = x;                -- критична ділянка
      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 Lab62;
```

6.1.3. Ада95. Захищені модулі

Реалізація операції додавання векторів у двопроцесорній системі зі спільною пам'яттю (рис. 6.1) з використанням захищеного модуля мови Ада95.

Етапи 1–3 не будуть відрізнятися від етапів, розглянутих в прикладі в підрозділі 6.1.1.

Етап 3. Розроблення структурної схеми взаємодії задач. Етап пов'язаний с розробленням структури захищеного модуля, за допомогою якого реалізується взаємодія задач. Захищений модуль `Керування` (рис. 6.3) включає три захищені елементи: `x`, `F1` і `F2`, а також набір захищених операцій:

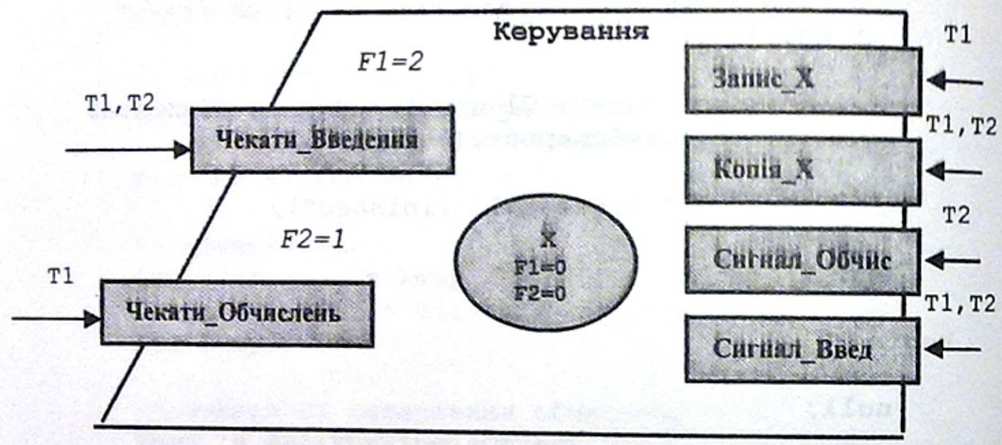


Рис. 6.3. Захищений модуль в операції `A = B + C * x`

- вхід `Чекати_Введення` для синхронізації з введення в задачах `T1` і `T2`;
- вхід `Чекати_Обчислень`, для синхронізації після завершення обчислень у задачі `T2`;
- функцію `Копія_X` для копіювання спільного ресурсу `x`;
- процедуру `Запис_X` для запису значення `x` у захищений модуль;
- процедуру `Сигнал_Обчис` для сигналу про завершення обчислень у задачі `T2`;
- процедуру `Сигнал_Введ` для сигналу про завершення уведення даних у задачах `T1` і `T2`.

Захищені змінні *F1* і *F2* використовують 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.

```
-----------------------------------------------------------
--|Ада95.Захищений модуль у задачі синхронізації          --
--|Операція  A = B + C * x                                --
-----------------------------------------------------------
procedure Lab63 is

   N: integer := 400;   -- розмір векторів
   P: integer := 2;     -- кількість процесорів
   H: integer := N/P;   -- розмір підвектора

   type Вектор is array (1.. N) of integer;

   A,B,C : Вектор;            -- глобальні змінні

   --  захищений модуль
   -- специфікація
   protected  Керування is

      entry  Чекати_Введення;
      entry  Чекати_Обчислень;
      function    Копія_X return integer;
      procedure   Запис_X(e: in integer);
      procedure   Сигнал_Введ;
      procedure   Сигнал_Обчис;
   private
      x : integer;  -- спільний ресурс
      F1: integer:= 0;
      F2: integer:= 0;

   end Керування;

   -- тіло
   protected body Керування is
      entry  Чекати_Введення when F1 = 2 is
      begin
         null;
      end Чекати_Введення;
      entry  Чекати_Обчислень when F2 = 1 is
      begin
         null;
      end Чекати_Обчислень;
      function  Копія_X return integer is
      begin
```

```
        return x;
    end Копія_X;
    procedure  Запис_X(e: in integer) is
    begin
       x:= e;
    end Запис_X;
    procedure  Сигнал_Введ is
    begin
       F1:= F1 + 1;
    end Сигнал_Введ;
    procedure  Сигнал_Обчис is
    begin
       F2:= 1;
    end  Сигнал_Обчис;

end Керування;
--------------
-- задачі
task T1;

task body T1 is
   x1, z : integer;    -- локальні змінні
begin
   put_line("Process T1 started");

   -- введення B та x
   for i in 1.. N loop
       B(i):= 1;
   end loop;
   z:= 2;

   -- запис x  в захищений модуль
   Керування.Запис_X(z);

   -- Сигнал задачі T2 про введення B та x
   Керування.Сигнал_Увед;

    --  чекати на введення C в задачі T2
    Керування.Чекати_Введення;

    -- Копіювати  x  в x1
    X1: = Керування.Копія_X;

    -- обчислення
    for i in 1 .. H loop
       A(i):= B(i) + C(i)*x1;
    end loop;
```



```
    -- Чекати на завершення обчислень у T2
    Керування.Чекати_Обчислень;

    -- виведення результату
    put_line(" A = ");
    for i in 1 .. N loop
       put(A(i),3);
    end loop;

    put_line("Process T1 finished");

end T1;
-------
task T2;
task body T2 is
   x2: integer; -- локальна змінна
begin
   put_line(" Process B started");

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

   -- сигнал задачі T1 про введення C
   Керування.Сигнал_Введ;

   -- чекати уведення B і x  у задачі T1
   Керування.Чекати_Введення;

   -- копіювати  x  в x2
   X2: = Керування.Копія_X;

   -- обчислення
   for i in H+1 .. N loop
      A(i):= B(i) + C(i)*x2
   end loop;

   -- сигнал  T1 про завершення обчислень
   Керування.Сигнал_Обчис;

   put_line(" Process T2 finished");

end T2;

--  тіло основної програми
begin
```

```
    put_line("    Main procedure started ");

end Lab63;
```

6.1.4. Java. Монітори

Реалізація операції скалярного множення векторів $a = (B*C)$ в двопроцесорній системі зі спільною пам'яттю (рис. 6.4) з використанням механізму моніторів у мові Java [53].

Етап 1. Побудова паралельного алгоритму. Паралельній алгоритм операції $a = (B*C)$ можна подати у вигляді

$$a_i = (B_н * C_н) \qquad (6.2)$$
$$a = a + a_i$$

Спільним ресурсом є скаляр **a**, відносно якого необхідно розв'язувати завдання взаємного виключення під час формування кінцевого результату.

Етап 2. Розроблення алгоритмів процесів

Потік *P1*

	Точки синхронізації	
1. Введення *B* і *C*		
2. **Сигнал** задачі T2 про введення *B* і *C*	--	*S2-1*
3. Обчислення a1 = (Вн * Сн)		
4. Обчислення a = a + a1	--	КД
5. **Чекати** на завершення обчислень в *T2*	--	*W2-1*
6. Виведення результату *A*		

Потік *P2*

1. **Чекати** на введення в *T1*	--	*W1-1*
2. Обчислення a2 = (Вн * Сн)		
3. Обчислення a = a + a2	--	КД
4. **Сигнал** *T1* про завершення обчислень	--	*S1-1*

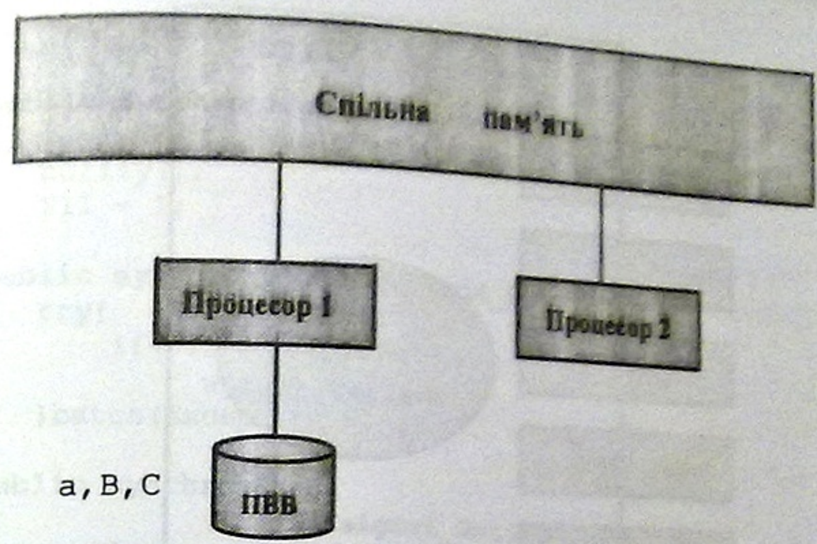


Рис. 6.4. Структурна схема комп'ютерної системи

Етап 3. Розроблення структурної схеми взаємодії задач. Етап пов'язаний з розробленням структури класу (монітора), за допомогою якого реалізується взаємодія задач. Клас `Data` (рис. 6.5) включає три поля: `a`, `Fl1` і `Fl2`, а також набір методів для організації взаємодії потоків:

- `wait_In()` – для очікування завершення введення даних;
- `signal_In()` – для сигналу про завершення введення 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 `Lab64` і пакета `Vector`, який включає три класи:

- `Data` – клас для синхронізації потоків;
- `aThread` – клас для потоку *P1*;
- `bThread` – клас для потоку *P2*.

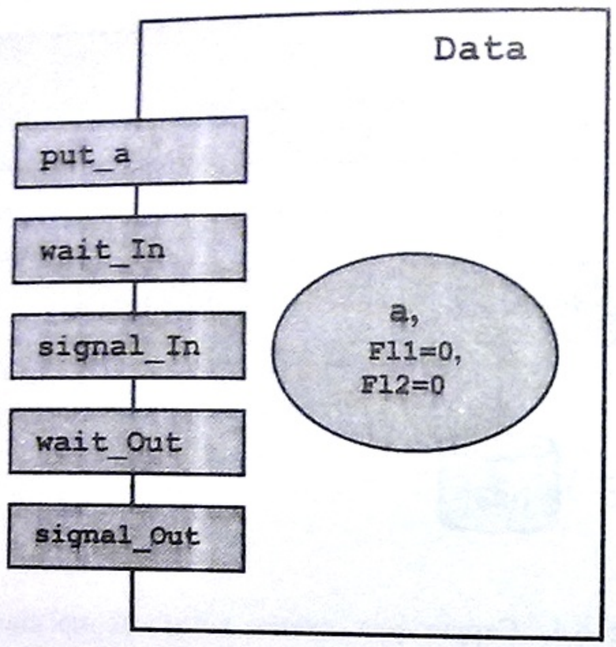


Рис. 6.5. Структура класу Data

```
/*-----------------------------------------------------
-- Java. Синхронізовані методи. Операція a = (B*C)    --
-----------------------------------------------------*/
package Vector;
public class Data {

   int a = 0;        // результат
   int N = 1000;
   int H = N/2;

   int B[] = new int [N];
   int C[] = new int [N];

   private int Fl1 = 0;
   private int Fl2 = 0;

   public synchronized void
      put_a(int x){ a = a + x;}

   public synchronized void wait_In(){
      try{
         if (Fl1 = = 0)
```



```
            wait();
      }catch(Exception e) {}
   }
   public synchronized
                void signal_In(){
      notify();
      Fl1 = 1;
   }
   public synchronized void wait_Out(){
      try{
         if(Fl2 = = 0)
            wait();
      }catch(Exception e) {}
   }
   public synchronized
                void signal_Out(){
      notify();
      Fl2 = 1;
   }
}

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          //S2-1
      Z.signal_In();

      // обчислення
      for (int i = 0; i < Z.H; i++)
            a1 = a1 + Z.B[i]*Z.C[i];

      // формування результату      // Критична ділянка
      Z.put_a(a1);
```

```
        //Очікування на завершення обчислень в P2
        Z.wait_Out();                                   // W2-1

        // Виведення результату
        Z.geta();
    }
}

public class bThread extends Thread{

    int   a2 = 0;
    Data  Z;

    public bThread(Data q) {
        Z = q;
    }
    public void run(){

        // очікування на уведення в P1
        Z.wait_In();                                    // W1-1

        //  обчислення
        for (int i= Z.H; i< Z.N; i++)
            a2 = a2 + Z.B[i]*Z.C[i];

        // формування результату
        Z.put_a(a2);                    // Критична ділянка

        //  сигнал про завершення обчислень
        Z.signal_Out();                                 // S1-1
    }
}

// Основний клас
import Vector.*;
public class Lab64 {

    public static void main (String [] args){
        Data D = new Data();
        aThread  P1 = new  aThread(D);
        bThread  P2 = new  bThread(D);
        P1.start();
        P2.start();
    }
}//Lab64
```


6.2. Програмування для комп'ютерної системи із локальною пам'яттю

6.2.1. Ада95. Рандеву

Реалізація операції додавання векторів $A = B + C * x$ у трипроцесорній системі з локальною пам'яттю (три процесори Пр.1, Пр.2, Пр.3 і два пристрої введення-виведення (ПВВ)) з використанням механізму рандеву мови Ада95.

Вхідні дані (вектори B та C, скаляр x) уводяться в ПВВ1, що зв'язаний з процесором 1, результат – вектор A виводиться в ПВВ2.

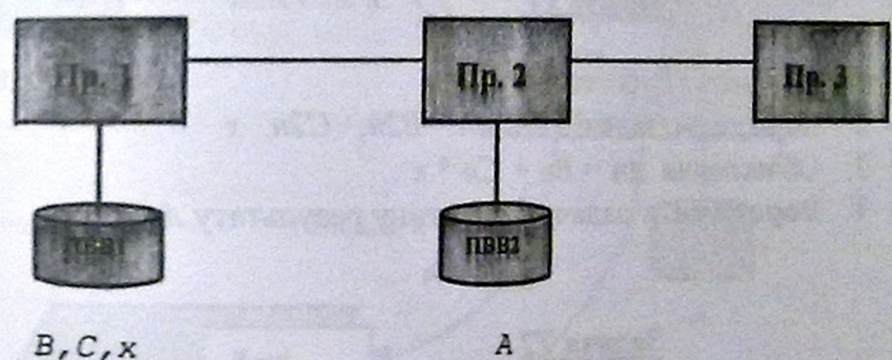


Рис. 6.6. Структура КС з локальною пам'яттю

Етап 1. Побудова паралельного алгоритму. Паралельний алгоритм реалізації цієї операції буде аналогічним алгоритму, розглянутому в прикладі в підрозділі 6.1.1.

Етап 2. Розроблення алгоритмів роботи кожного процесу. Цей етап пов'язаний з розробленням алгоритмів роботи кожного процесу. Крім безпосередніх обчислень, введення вхідних даних і виведення результату, алгоритми мають включати дії, що пов'язані із взаємодією задач. Така взаємодія включає:

- розсилання із задачі *T1* вхідних даних по задачах *T2* і *T3*; оскільки задача *T1* не пов'язана безпосередньо із задачею *T3*, то пересилання даних між ними буде виконуватися через задачу *T2*;
- збирання результату в задачі *T2*.

Розробляючи алгоритм задач, слід, звернути увагу на обсяги даних, що пересилаються між задачами. Під час розсилання вхідних даних із задачі *T1* в *T2* потрібно переслати дві частини вектора *B* і *C* розміром 2**H* (позначається як *B2H* і *C2H*), одна з яких призначена безпосередньо для T2, а друга (*Bн, Cн*) – для пересилання далі в задачу *T3*. Пересилання повних векторів зумовлено значним збільшенням часу виконання програми, оскільки операції пересилання даних між процесами в КС з локальною пам'яттю є досить довгі операції порівняно з операціями обчислень і необхідно їх мінімізувати як за кількістю пересилок, так і за обсягом інформації, що передається.

Задача *T1*

1. **Введення** *B, C* та *x*
2. **Передати** задачі *T2* дані: *B2н, C2н, x*
3. Обчислення *Aн = Bн + Cн * x*
4. **Передати** в задачу *T2* частину результату *Aн*

Задача *T2*

1. **Прийняти** від задачі *T1* дані: *B2н, C2н, x*
2. **Передати** в задачу T3 дані *Bн, Cн, x*
3. Обчислення *Aн = Bн + Cн * x*
4. **Прийняти** від *T1* результат – вектор *Aн*
5. **Прийняти** від *T3* результат – вектор *Aн*
6. Виведення результату – вектор *A*

Задача *T3*

1. **Прийняти від** задачі *T2* дані: *Bн, Cн, x*
2. Обчислення *Aн = Bн + Cн * x*
3. **Передати** в задачу *T2* результат – *Aн*

Етап 3. Розроблення структурної схеми взаємодії задач. Структурна схема взаємодії задач ґрунтується на алгоритмах задач і дозволяє наочно показати передавання даних між задачами (рис.6.7).

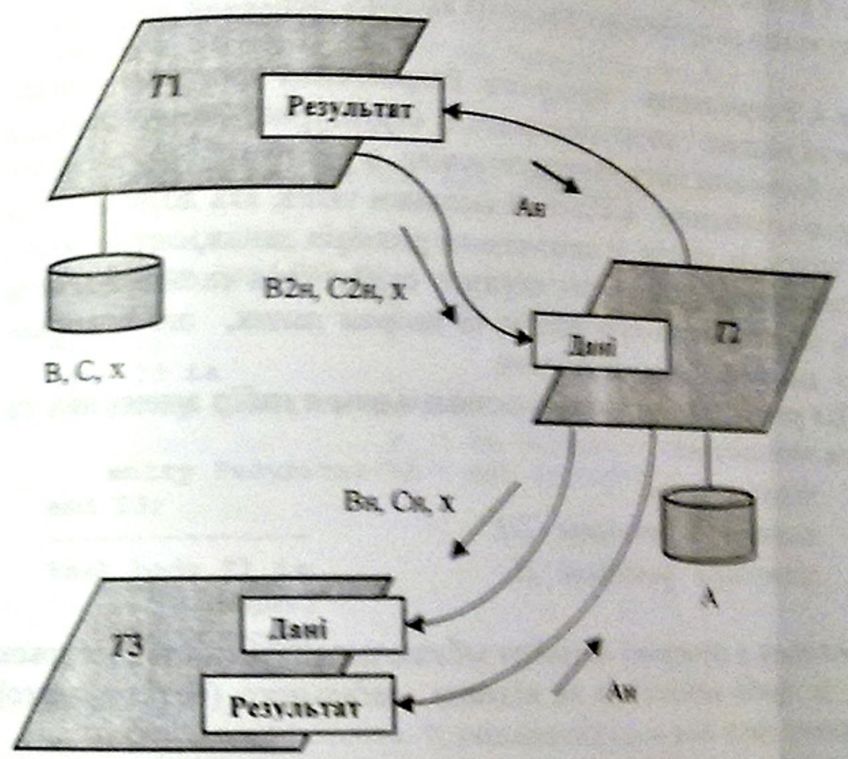


Рис. 6.7. Структурна схема взаємодії задач

На структурній схемі визначаються входи задач, їх розміщення в належних задачах і протоколи взаємодії для них, які задають напрям і обсяг передачі даних між задачами. Для подання взаємодії задач використовують дві напрямлені лінії. Одна лінія визначає виклик входу належної задачі, друга, коротка – напрям передавання даних, яке буде виконуватися за цим викликом. Біля короткої стрілки вказується інформація, що передається, наприклад, два вектори і матриця. Ця інформація потім використовується для оголошення специфікацій входів задач у програмі.

Графічне зображення взаємодії задач дозволяє також виявити можливі *тупикові ситуації* в програмі у випадку, якщо вхід не має виклику або між задачами має місце циклічний виклик входів. Щоб позбавитись таких ситуацій, слід розміщувати входи в одній задачі, а не в різних задачах. При цьому слід контролювати порядок виклику входів та приймання викликів входів.

Етап 4. Розроблення програми. Розроблення програми виконується на підставі алгоритмів задач і структурної схеми взаємодії задач. Формуючи типи, використовувані в програмі, слід звернути увагу на оголошення достатньої множини типів, яка 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 оптимальним є набір векторних типів для передавання:

- цілого вектора;
- 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.

```
-------------------------------------------------------------
--|Ада95.Механізм рандеву Операція A = B + C * x            --
-------------------------------------------------------------
procedure Lab65 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 задач
   task T1 is
      entry Результат(VA: out  ВекторH);
   end T1;
   task T2 is
      entry Дані(VB,VC : in Вектор2H;
                    y : in integer);
   end T2;
   task T3 is
      entry Дані(VB,VC  : in  ВекторH;
                    y  : in integer);
      entry Результат(VA : out ВекторH);
   end T3;
   ---------------
   task body T1 is
      x1: integer;
      A1: ВекторH;
      B1, C1 : Вектор;
   begin
      put_line("Process T1 started");

      -- уведення B,C і x
      for i in 1.. N loop
         B1(i):= 1;
         C1(i):= 2;
      end loop;
      x1:= 2;

      -- передати B2H, C2H  і x в  задачу T2
      T2.Дані(B1(H+1..N), C1(H+1..N), x1);

      -- обчислення
      for i in 1 .. H loop
         A1(i):= B1(i) + C1(i)*x1;
      end loop;

      -- передати в T2 Ан
```

```
   accept Результат(VA: out  ВекторН) do
      VA:= A1;
   end Результат;

   put_line("Process T1 finished");
end T1;

---------------
task body T2 is

   x2: integer;
   A2: Вектор;   -- для виведення повного результату
   B2, C2 : Вектор2Н;
begin
   put_line("     Process T2 started");

    -- прийняти B2H, C2н і x  від T1
    accept Дані(VB,VC : in Вектор2Н;
                    y : in integer) do
       B2 := VB;
       C2 := VC;
       X2 := y;
    end  Дані;

    -- передати  данні в T3
    T3.Данні(B2(H+1..2*H), C2(H+1..2*H), x2);

    -- обчислення
    for i in 1 .. H loop
       A2(H+i):= B2(i) + C2(i)*x2
    end loop;

    -- прийняти результат від T1
    T1.Результат(A2(1..H));

    -- прийняти результат від T3
    T3.Результат(A2(2*H+1..N));

    -- виведення результата
    put_line(" A = ");
    for i in 1 .. N loop
       put(A2(i),3);
    end loop;
    put_line("Process T2 finished");
end T2;
```



```
   x3: integer;
   A3, B3, C3 : ВекторН;
Begin

   put_line("     Process T3 started");

   -- прийняти від T2 BH, Cн і x
   accept Дані(VB,VC : in ВекторН;
                    y : in integer) do
      B3:= VB;
      C3 := VC;
      X3 := y;
   end  Дані;

   -- Обчислення
   for i in 1 .. H loop
      A3(i):= B3(i) + C3(i)*x3;
   end loop;

   -- передати Ан в T2
   accept Результат(VA: out  ВекторН) do
      VA:= A3;
   end Результат;

   put_line("Process T3 finished");

end T3;

--  тіло основної програми
Begin

   put_line("  Main procedure started ");

end Lab65;
```

6.2.2. Бібліотека PVM

Реалізація операції множення матриць $MA=MB * MC$ у п'яти процесорній системі з локальною пам'яттю (рис.6.8) з використанням бібліотеки PVM та мови C.

Етап 1. Побудова паралельного алгоритму. Паралельний алгоритм для операції множення матриць:

$$MA_н = MB_н * MC$$

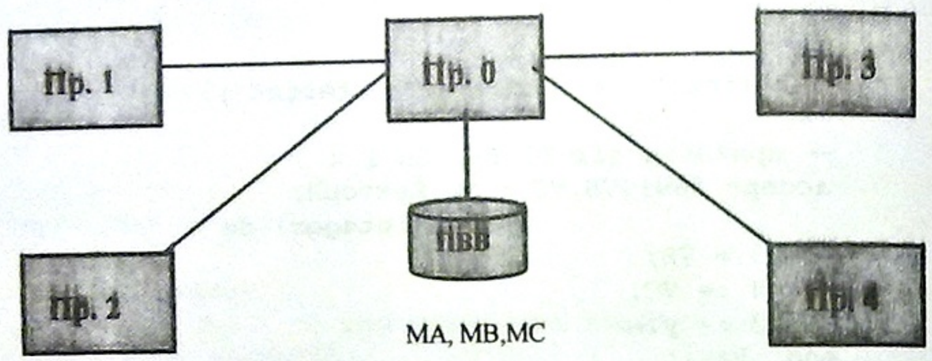


Рис. 6.8. Структура КС з локальною пам'яттю

Етап 2. Розроблення алгоритмів роботи кожного процесу. Цей етап пов'язаний з розробленням алгоритмів роботи процесу Майстер (процесор 0) та Робочий (процесори 1–4).

Майстер

1. Уведення *MB, MC*
2. Активізація процесів Робочий
3. **Розсилання** всім процесам Робочий матриці *MC*
4. **Передати** *MBн* кожному процесу Робочий
5. Обчислення *MAн = MBн *MCн*
6. **Прийняти** *MAн* від кожного процесу Робочий
7. Виведення результату – матриці *MA*

Робочий

1. **Прийняти** матрицю *MC* від процесу Майстер
2. **Прийняти** *MBн* від процесу Мастер
3. Обчислення *MAн = MBн *MCн*
4. **Передати** MAн до процесу Майстер

Етап 3. Розроблення структурної схеми взаємодії задач

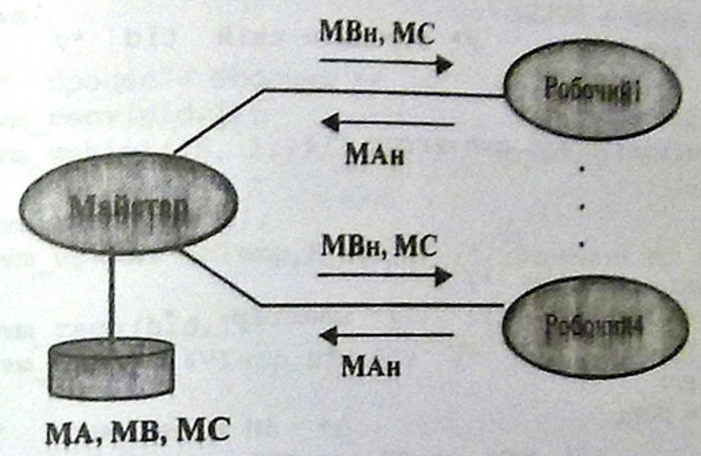


Рис. 6.9. Взаємодія Майстер–Робочий

Етап 4. Розроблення програми

```
/*-----------------------------------------------------------
 PVM. Lab66. Операція MA=MB*MC
-----------------------------------------------------------*/
#include "pvm3.h"
#include <stdio.h>
#include <string.h>

#define N 5
#define P 5
#define H 1

#define SPAWN_TYPE   PvmaTsks
#define SPAWN_WHERE  "Work"

int MA[N][N];
int MB[N][N] = (1, .., 1);
int MC[N][N] = (2, .., 2);

int MTemp[N][N];
int VTemp[N];
int MAH[N];

main(int argc, char **argv){

     int Child[P];  /* масив з tid дочірніх процесів */
     int myTid;
```

```
int pid;    /*   ідентифікатор процесу */
int n;      /*   розмір масиву */

char ** agvc = NULL;
myTid = pvm_tid();     /* отримати свій  tid  */

if (mytid<0){
  pvm_perror("Start pvm error ");
  exit(-1);
}

/* Процес - Майстер  */
if pid == PvmNoParent){     /* майстер    */
   int i;
   int rc;
   num = NUM;

/* запуск дочірніх процесів */
rc = pvm_spwm(argv[0],argvc, SPAWN_TYPE,
                    SPAWN_WHERE, P-1, Child+1);

/* переслати всім дочірнім процесам розмір матриць */
pvm_initsend(PvmDataDefault);
pvm_pkint(&N,1,1);
pvm_mcast(Child + 1, P-1,1);

/* переслати всім дочірнім процесам  матрицю MC  */
pvm_initsend(PvmDataDefault);
pvm_pkint(MC, P*P,1);
pvm_mcast(Child + 1, P-1,1);

/* переслати всім дочірнім процесам частину матриці
 MBh  (один стовпець) */
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);
}
```

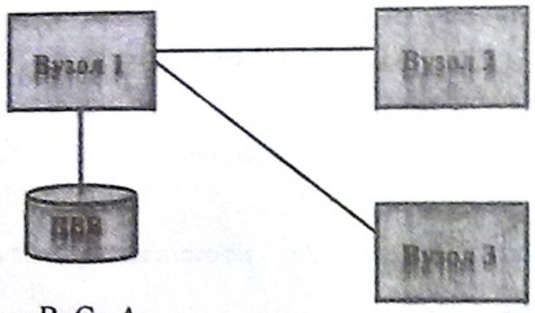


```
/* виведення  MA  */
ДрукМасиву("MA  = ", MA);

}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 MBh  */

/*  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){
     .  .  .
}
```

6.3. Програмування для розподілених комп'ютерних систем

У цьому розділі наведено приклади реалізації операції додавання векторів $A = B + C$ в розподіленій комп'ютерній системі (рис. 6.10) з трьома вузами і одним пристроєм введення–виведення (ПВВ) мовою Java (використання механізму сокетів) та мовою Ада95 (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 однаковими.



B, C, A

Рис. 6.10. Структура розподіленої системи

Етап 1. Побудова паралельного алгоритму. Паралельний алгоритм реалізації цієї операції буде мати вигляд

$$A_н = B_н + C_н, \qquad (6.3)$$

де $A_н$ – H елементів вектора A.

Співвідношення (6.3) визначає дії вузлів 2 і 3 розподіленої системи під час виконанні обчислень.

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. Введення *B, C*
2. **Передати** *Клієнт 1* дані: $B_н$, $C_н$
3. **Передати** *Клієнт 2* дані: $B_н$, $C_н$
4. **Прийняти** від *Клієнт 1* результат – вектор $A_н$
5. **Прийняти** від *Клієнт 2* результат – вектор $A_н$

6. Виведення результату – вектора *A*

Клієнт 1

1. **Прийняти** від *Сервер* вектори $B_н$, $C_н$
2. Обчислення $A_н = B_н + C_н$
3. **Передати** *Сервер* результат - вектор $A_н$

Клієнт 2

1. **Прийняти** від *Сервер* вектори $B_н$, $C_н$
2. Обчислення $A_н = B_н + C_н$
3. **Передати** *Сервер* результат – вектор $A_н$.

Наступні етапи розроблення програм будуть відрізнятися, оскільки використовуються різні механізми реалізації посилання повідомлень: сокети в мові Java та віддалені процедури в мові Ада.

6.3.1. Java. Сокети

Етап 3. Розроблення структурної схеми взаємодії розділів. Передавання (приймання) даних у *Сервері* виконується в Java за допомогою серверних сокетів, у *Клієнті* – 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 у 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 зі своїм *Клієнтом*, де створені потоки `MyClient` (рис. 6.11).

Етап 4. Розроблення програми. Розроблення програми виконується на підставі алгоритмів потоків і структурної схеми взаємодії класів:

```
/*-----------------------------------------------------
--  Lab67.Java.Сокети. Операція A = B + C              --
-----------------------------------------------------*/

import java.io.*;
import java.net.*;
```

```
public class  Lab67 {
   Lab67(){

      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();}

}// Lab67

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 Fl = 1;            // Прапор 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 (Fl==1) wait();
      }catch(Exception e){}
   }

    synchronized void Signal(){
      try{
         notify();
         Fl = Fl -1;
```

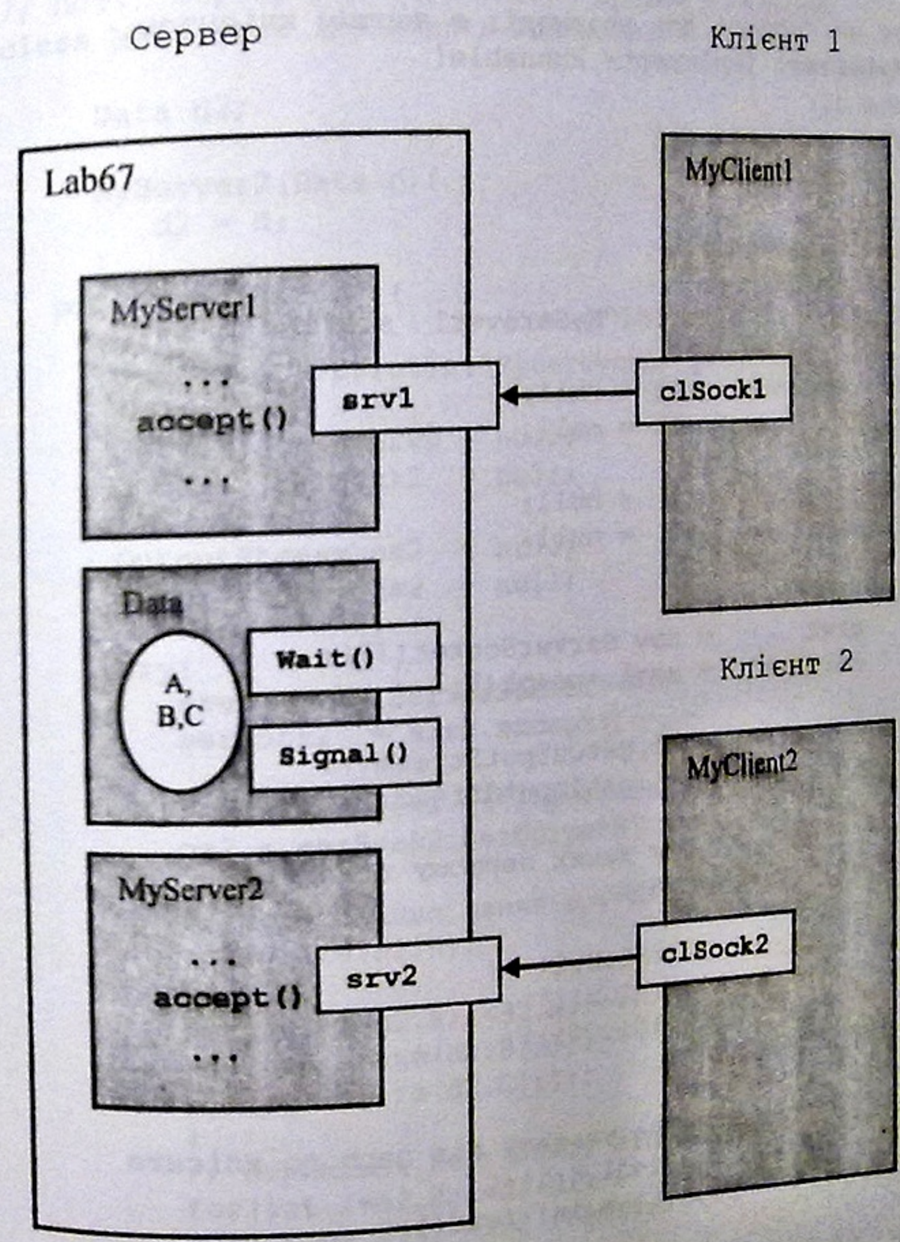



Рис. 6.11. Реалізація моделі клієнтсервер

```
      }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("MySerever1  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();

         // 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("MySerever1 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("MySerever2  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 клієнта
         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 клієнте
         for(int i=d2.H;i<d2.N;i++)
                d2.A[i] =is2.read();

         // синхронізація
         d2.Signal();
         System.out.println("MySerever2 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){ }
   }
}

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 =  clSock2.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){   }
   }

}// Client2
```

Розглянутий приклад орієнтований на використання в одному комп'ютері. При використанні кількох комп'ютерів слід звернути увагу на формування IP - адреси серверу.

6.3.2. Ада95. Віддалені процедури

Етап 3. Розроблення структурної схеми взаємодії розділів. Структурна схема взаємодії розділів ґрунтується на алгоритмах задач і дозволяє наочно показати передавання даних між розділами. На структурній схемі (рис. 6.12) 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.

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.

```
--------------------------------------------------------------
--|Lab68.Ада95.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 ВекторH   is Вектор_Загальний (1..H);

   -- віддалена процедура
   procedure Додавання_Векторів(VB,VC: in  ВекторH;
```

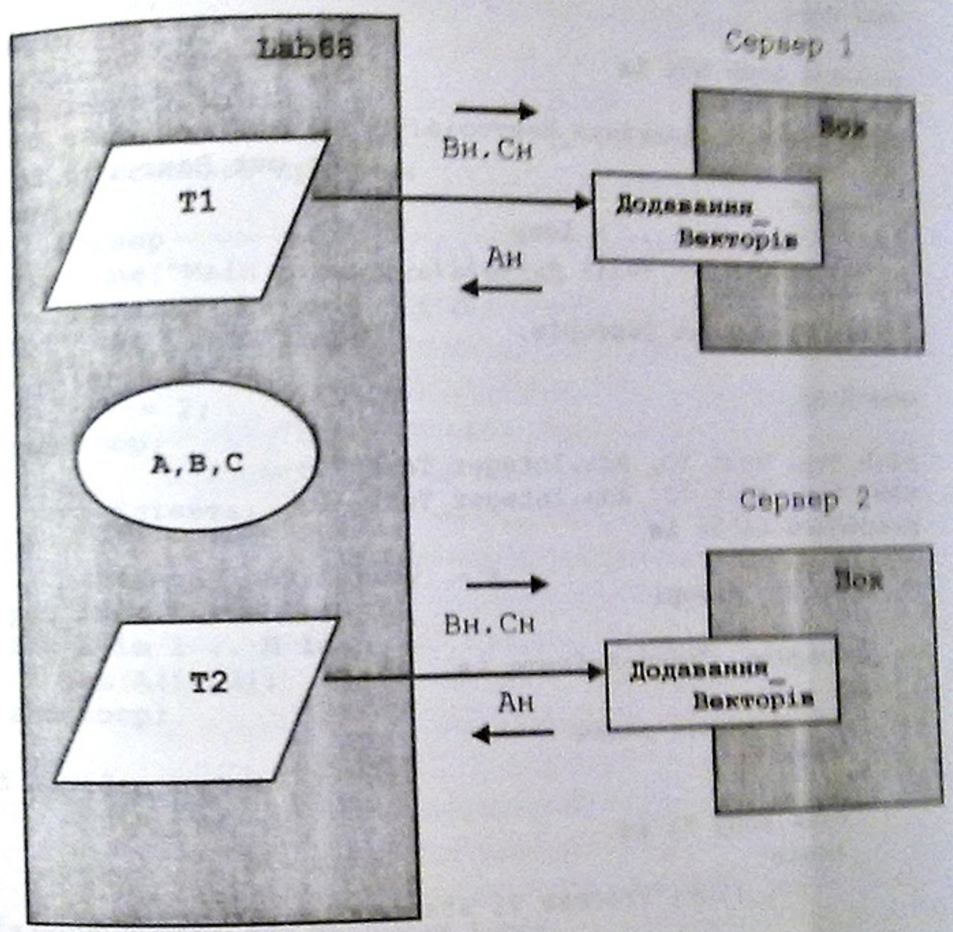



Рис. 6.12. Реалізація взаємодії сервера з клієнтами

```
                                    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 Lab68 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(B(1..H),C(1..H),A(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(B(H+1..2*H),C(H+1..2*H),
                                                A(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 ");
   -- уведення векторів B і C
   for i in 1.. N loop
      B(i):= 1;
      C(i):= 2;
   end loop;

   Старт_Клієнта;

   -- виведення результату  - A
   put_line(" A = ");
   for i in 1 .. N loop
      put(A(i),3);
   end loop;

end Lab68;
```

Файл конфігурації для програми Lab68:

```
configuration MyConfig is

   Partititon_Client: Partition := ();
   Procedure  Main is in Partititon_Client;
   Partition_Serever1: Partition;
   Partition_Serever2: 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 заданій структури.

Розглядаються такі векторно-матричні операції:

- скалярне множення векторів;
- множення вектора на матрицю;
- множення матриць;
- сортування векторів;
- пошук мінімального (мінімального) елемента вектора.

Виконуючи 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 залежно 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 4–8 процесори, 2–3 пристрої введення-виведення. Використовують мови Java, C#, C, Ада95, бібліотеки Win32, Pthread. Засоби взаємодії процесів: 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 (див. рис. 1.5)

- лінійні,
- кільцеві,
- зірка,
- решітка,
- гіперкуб.

Кількість процесорів системи і розміщення пристроїв уведення-виведення визначається в завданні. Засоби, які використовуються: мови C, Ада95; 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, C, Ада95. 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 вузли використано дво- або чотирипроцесорні СМП КС. Програмуючи клієнтські вузли, треба враховувати, що обчислення в них слід виконувати як в системі зі спільною пам'яттю, тобто передбачити створення двох-чотирьох процесів для кожного процесора КС у вузлі, а також у разі потреби – вирішення завдань взаємного виключення і синхронізації за допомогою відповідних засобів використовуваних мов або бібліотек.

СПИСОК ЛІТЕРАТУРИ

1. Бар Р. Язык Ада в проектировании систем. – М.: Мир, 1988. – 320 с.
2. Богачев К.Ю. Основы параллельного программирования. – М.: БИНОМ. Лаб. знаний, 2003. – 342 с .
3. Бройнль Т. Паралельне програмування. Початковий курс: Навч. посіб. – К.: Вища шк, 1997. – 358 с.
4. Валях Е. Последовательно - параллельные вычисления. – М.: Мир, 1985. – 456 с.
5. Василеску Ю. Прикладное программирование на языке Ада 95. – 1990. – 332 с.
6. Вегнер П. Программирование на языке Ада. – М.: Мир, 1983. – 78 с.
7. Воеводин В.В. Математические модели и методы в параллельных процессах. – М.: Наука, 1984. – 296 с.
8. Гома Х. UML. Проектирование систем реального времени, параллельных и распределенных приложений. – М.: ДМК Пресс, 2002. – 704 с.
9. Дейтел Д. Введение в операционные системы. – М.: Мир,1989. – 360 с.
10. Джехани Н. Язык Ада. – М.: Мир, 1988. – 552 с.
11. Джоунз Г. Программирование на языке ОККАМ. – М.: Мир, 1989. – 246 с.
12. Жуков І.А., Гуменюк В.О., Алтьман І.Є. Комп'ютерні мережі та технології: Навч. посібн. – К.: НАУ, 2004. – 276 с.
13. Жуков И.А. Методология синтеза параллельных вычислительных систем для задач большой размерности // Проблеми інформатизації та керування. – К.: КМУЦА, 1997. – Вип. 2. – С. 7 – 10.
14. Жуков І.А., Кременецький Г.А. Апаратні та програмні засоби підготовки та розповсюдження електронних конфіденційних новин серед розподілених комп'ютерних мереж // Захист інформації. – К.: НАУ, 2002. – Вип. 2. – С. 60 – 67.

15. Жуков И.А., Іванкевич О.В. Распараллеливание операций по обработке конфиденциальных баз данных в многопроцессорных вычислительных системах // Захист інформації. – К.:НАУ, 2003. – Вип. 1. – С. 34 – 42.
16. Корочкин А. В. Ада95: Введение в программирование – К.: Свит, 1999. – 260 с.
17. Корочкин А., Мустафа Акрам Параллельные вычисления: Ада и Java. – Вісн. НТУУ "КПІ", Інформатика, управління та обчислювальна техніка, 1999, К.: – № 32, С. 13–17.
18. Корочкин А., Жужель М., Авдеев А., Корочкин Д. Защищенный модуль как универсальное средство синхронизации процессов. – Вісн. НТУУ"КПІ", Інформатика, управління та обчислювальна техніка, 2001, К. : – № 34, С. 137 – 145.
19. Корочкин А.,Тулинов А.Экспериментальный анализ эффективности PVM и MPI в распределенных вычислительных системах – Вісн. НТУУ "КПІ", Інформатика, управління та обчислювальна техніка, 2002, К. : – № 38, С. 154 – 58.
20. Кулаков Ю.О., Луцький Г.М. Комп'ютерні мережі. – К.: Юніор, 2003. – 400 с.
21. Лисков Б., Гатэг Дж. Использование абстракций и спецификаций при разработке программ : Пер. с англ. – М.: Мир, 1989. – 424 с.
22. Луцкий Г.М., Кулаков Ю.А. Локальные сети. – К.: Юниор, 1998. – 336 с.
23. Миренков Н.Н. Параллельное программирование для многомодульных вычислительных систем. – М.: Радио и связь, 1989. – 320 с.
24. Нагорный Л.Я., Жуков И.А. Многоуровневые алгоритмы и структуры распраллеливания решения систем уравнений большой размерности // Комп'ютери у Європі. Минуле, сучасне та майбутнє: Праці міжнар. симпозіуму з історії створення перших ЕОМ та внеску європейців в розвиток комп'ютерних технологій, Україна. – К.: НАУ,1998. С. 468 – 472.
25. Немнюгин С., Стесик О. Параллельное программирование для многопроцессорных вычислительных систем. – СПб.: БХВ – Петербург, 2002. – 400 с.
26. Ноутон П., Шилдт Г. Java2: Пер. с англ. – СПб.: БХВ – Петер –

бург, 2000. – 1072 с.
27. Ортега Дж. Введение в параллельные и векторные методы решения линейных систем. – М.: Радио и связь, 1989, – 280 с.
28. Параллельные вычисления / Под ред. Г.Родрига – М.: Наука, 1986. – 376 с.
29. Пайл Я. Ада – язык встроенных систем. – М.; Финансы и статистика, 1984. –120 с.
30. Перминов О.Н. Введение в язык программирования Ада. – М.: Радио и связь, 1991. – 228 с.
31. Программирование на параллельных вычислительных системах: Пер. с англ. / Р. Бэбб, Дж. Мак – Гроу и др.; Под ред Бэбба II. – М.: Мир, 1991. – 376 с.
32. Русанова О.В. Программное обеспечение компьютерны систем. Особенности программирования и компиляции. – К.: Корійчук, 2003. – 94 с.
33. Симкин С., Барлетт Н., Лесли А. Программирование на Java. Путеводитель – К.: НИПФ "ДиаСофт Лтд.", 1996. – 736 с.
34. Симоненко В.П. Организация вычислительных процессов в ЭВМ, комплексах, сетях и системах. – К.: ВЕК+, 1997. – 304 с.
35. Соловьев Г.Н., Никитин В.Д . Операционные системы ЭВМ. – М.: Высш. школа., 1989. – 255 с.
36. Системы параллельной обработки: Пер. с англ./Под ред.Д.Ивенса. – М: Мир, 1985. – 416 с.
37. Траспьютеры.Архитектура и программное обеспечение: Пер. с англ./Под ред.Г.Харпа. – М.: Радио и связь, 1993. – 304 с.
38. Троелсон Є. C# и платформа.NET. Библиотека программиста – СПб.: Питер, 2004. – 796 с.
39. Хоар Ч. Взаимодействующие последовательные процессы. – М.: Мир, 1989. – 180 с.
40. Хокни Р., Джессхоул К. Параллельные ЭВМ. Пер. с англ. – М.: Радио и связь, 1986. – 240 с.
41. Юткин А. Обьектные технологии в распределенных системах. Открытые системы, – № 3(11). – 1995. – С. 23–28.
42. Эндрюс Г. Основы многопоточного, параллельного и распределенного програмирования.: Пер. с англ. – М.: Изд. Дом «Вильямс», 2003. – 512 с.
43. Элементы параллельного программирования / В.А.Вальковский

, В.Е.Котов, А.Г.Марчук / Под ред. В. Е. Котова – М.: Радио и связь, 1983. – 240 с.
44. Языки программирования: Ада, Паскаль, Си.Сравнение и оценка / Под ред.Н. Джехани – М.: Радио и связь, 1989. – 386 с.
45. Burns A., Wellings A.Real-Time Systems and Programming Languages. Addison – Wesley, 2001, – 386 p.
46. Burns A., Programming in Occam2. Reading. Addison – Wesley, 1995, – 326 p.
47. Burns A., Wellings A. Concurrency in Ada. – Cambridge: Cambridge University Press, 1995., – 420 p.
48. El–Rewini H, Lewis T.,Distributed and Parallel Computing. – Manning Pub. Co.1998, – 430 p.
49. Hyde P. Java Thread Programming. – Indiapolis, In: Sams Publishing, 1999, – 324 p.
50. Hoare C.A.R. Monitors: An Operating System Structuring Concept, Communications of ACM, Vol.17, №.10, Oct.1974, pp. 549–557 (9)
51. Hoare C.A.R. Communicating Sequential Processes, Communications of ACM, vol.21, №. 8, Aug. 1978, pp. 666 – 667.
52. Hoare C.A.R. Communicating Sequential Processes,: Printice Hall, International Series in Computer Science, Englewood Cliffs NJ, 1985. – 186 p.
53. Korochkin D., Korochkin S. Experimental Performance Analysis of the Ada95 and Java Program on SMP System. In Proceeding of the ACM Annual Conference (SIGADA'02)(The Houston, Texas, USA, December 8 – 12, 2002) ACM Press, New York, NY, 2002, pp. 48 – 54.
54. 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;
55. 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'099), (Santander, Spain, June 7 – 11, 1999), Lecture Notes in Computer Science , – № 1622, Springer, 1999, pp. 62 – 70.
56. Korochkin A.,V.Rovto, M.Zuzel, W.Sinchuk. Experimental Performance Analysis of the Ada95 Program on Distributed Systems. In

Proceeding of the1-st International Conference (ACSN-2003) (The Lviv, Ukraine, September 24 – 26, 2003) Publishing House of LvivPNU, Lviv, 2003. pp. 86-92.

57. 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.
58. Lea D. Concurrent Programming in Java: Design Principles and Patterns. Reading, MA: Addison – Wesley, 1999, – 344 p.
59. Taft S., Duff R., Brukardt R., Ploedereder T.Consolidated Ada Refe-Reference Manual. Language and Standard Libraries, Springer, Berlin: 2001, – 562 p.

ІНТЕРНЕТ- РЕСУРСИ

У цьому розділі наведено дані про Інтернет-ресурси, за допомогою яких можна знайти інформацію, що пов'язана з паралельними і розподіленими обчисленнями, а також знайти програмне забезпечення, яке можна використати для розроблення програм для паралельних і розподілених комп'ютерних систем і яке вільно поширується.

1. Система програмування *Object Ada7.2.2 SE* (Aonix):
   `http://www.act-europe.com`

2. Система програмування *GNATAda 3.15p* (AdaCore Technology):
   `http://www.act-europe.fr`
   `ftp://cs.ny.edu/pub/gnat`
   `http://www.gnat.com`

3. Реалізація DSA - *GLADE 3.15p* (AdaCore Technology):
   `http://www.act-europe.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. Ада95 - ресурси:
   `http://www.adaword.com`

http://www.ada-europe.org
http://www.adapower.com
http://www.act-europe.com
http://www.acm.org/ada

8. 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

ПРЕДМЕТНИЙ ПОКАЖЧИК

A
Ada95, мова 27
Atomic, прагма 73

B
Beowulf, проект, 17

C
CORBA, технологія, 135
Critical Section, критична секція, 78
C#, мова , 222

D
DCOM, технологія, 135
DSA, додаток, 147
Deadlock, 21

E
Entry, вхід, 113
Event, подія, 78

G
GLADE, система програмування, 153
GNAT, система програмування, 153

J
Java, мова, 22

L
Link, зв'язок, 108

M
Mutex, засіб, 78
Message-passing, інтерфейс 30
MPI, бібліотека 30

N
NET , технологія, 222

O
Occam, мова, 108
OpenMP, бібліотека, 22

P
Partition, розділ, 148
POSIX, ОС, 22
Pthreads, бібліотека, 22
PVM, бібліотека, 30
P(), операція, 55
PCB, блок, 19

R
Rendezvous, механізм, 113
RPC, віддалений виклик процедури, 136
RMI, бібліотека, 148

S
SMP симетрична мультіпроцесорна система, 12

T
Task, задача, 27
Thread, потік, 22

Tid, ідентифікатор задачі, 34

V

Volatile, прагма, 73
V(), операція, 55

W

Win32, бібліотека, 35

А

Алгоритм паралельний, 38

Б

Бар'єр, 67
Блокування, 20
Брент, лема 40
Блок керування процесу, 19

В

Віддалений інтерфейс, 144
Ввіддалений об'єкт, 135
Віддалена процедура, 136
Віддалений метод, 144
Взаємне виключення, 51
Вхід, 113

Г

Гіперкуб, 14

Д

Дідлок, тупик, 21
Діаметр системи, 14
Дискримінант, 27

З

Задача, 27
Задачний тип, 27
Захищений модуль, 65
Захищена операція, 65
Захищена функція, 66
Захищена процедура, 66
Захищений вхід, 66

І

Інтерфейс ввіддалений, 144

К

Канал, 108
Кеш-пам'ять, 12
Кластерні системи, 134
Критична ділянка, 51
Критична секція, 78

Л

Лінк, 108

М

Мова
Ада95, 27
Java, 22
Оккам, 108

Модель клієнт-сервер, 135
Монітор, 78
Мунро-Петерсон, теорема, 40
Мютекс, 78

Методи 23
wait()
notify()
notifyAll()
sleep()
suspend()

resume()
yield()
join()

О

Обчислення
паралельні, 50
розподілені, 134
Оккам, мова, 108
Оператор
Accept, 113
Delay, 29
Requeue, 113
select 113
Операція
Send, 107
Receive, 107

П

Пам'ять
розподілена, 13
спільна, 11
поділяєма, 11
Паралельна програма, 22
Передача повідомлення, 107
Потік, 20, 22
Подія, 78
Повідомлення, 107
Прагма синхронізації, 73
Протокол, 110
Процес, 19

Прагма
Atomic, 73
Volatile, 73

Р

Рандеву, механізм, 113
Розділ, 148
Розподільна змінна, 73
Розподілена програма, 134
Ресурс (поділяємий), 51

С

Семафор, 55
Синхронізація, 55
Синхронізовані блоки, 86
Синхронізовані методи, 85
Спільний ресурс, 51
Стан процесу, 20

Т

Точка синхронізації, 158
Трансп'ютер, 108
Тупик, 21
Топологія системи, 14

Ф

Функції очікування, 60

Х

Хоар, концепція, 63

ДОДАТКИ

Додаток А. Мова Java.
Додаток Б. Мова Ada95.
Додаток В. Бібліотека Win32.
Додаток Г. Бібліотека MPI.
Додаток Д. Бібліотека PVM.
Додаток Е. Бібліотека Pthreads.
Додаток Ж. Мова C# і платформа .NET.

Додаток А

Класи Java

А.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();

ThreedGroup 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 nanjs)
                    throws InterruptedException;

  private native void resume()
            throws SecurityException; // DEPREACTED

  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; // виключений

  private static void yield();
```



```
    public String toString();

  public void interrupt() throws SecurityException;
}
```

А.2. Клас ThreadGroup

```
package  java.lang;
public   class  ThreadGroup{

  // Конструктори
  public  class ThreadGroup(String name)
                              throws SecurityException;
  public  class ThreadGroup(ThraedGroup parent,
                  String name)throws SecurityException;

  // Методи
  publicic 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);

}
```

А.3. Інтерфейс Runnable

```
package java.lang;
public  interface Runnable {
   public abstract void run();
}
```


Додаток Б
Специфікація пакетів Ada95

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 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 на всіх
   DWORD dwMilliseconds,  // тайм-аут
   DWORD dwWakeMask       // тип події уведення
                          // для очікування
);

DWORD MsgWaitForMultipleObjectsEx(
   DWORD    nCount,       //кількість об'єктів в масиві
   LPHANDLE pHandles,     // покажчик на масив
                          // ідентифікаторів
   DWORD dwMilliseconds,  // тайм-аут
   DWORD dwWakeMask,      // тип події уведення
                          // для очікування
   DWORD dwFlags          // прапор очікування
);
```

В.2. Функції для роботи із сокетами

Функція	Дія
accept()	Прийняти з'єднання по сокету
bind()	Присвоїти локальне ім'я сокету
closesocket()	Закрити сокет
connect()	Ініціювати з'єднання через визначений сокет
getpeername()	Отримати ім'я порту, який з'єднується з сокетом
getsockname()	Отримати поточне ім'я для специфікованого сокету
recv()	Отримати дані від підключеного сокету
recvfrom()	Отримати дані від будь-якого сокету
send()	Послати дані в підключений сокет
sendto()	Послати дані в будь-якій сокет

Додаток Г

Бібліотека MPI

Г.1. Базові функції

Функція	Дія
MPI_Init()	Ініціалізація MPI
MPI_Finalize()	Завершення MPI
MPI_Comm_size()	Визначення розміру області взаємодії
MPI_Comm_rank()	Визначення номера процесу
MPI_Send()	Послати повідомлення (блоковане)
MPI_Recv()	Прийняти повідомлення (блоковане)
MPI_Irecv()	Прийняти повідомлення (не блоковане)
MPI_Ssend()	Послати повідомлення (блоковане, синхронізоване)
MPI_Isend()	Послати повідомлення (не блоковане, стандартне)
MPI_Issend()	Послати повідомлення (не блоковане, синхронізоване)
MPI_Bsend()	Послати повідомлення (буферизоване)
MPI_Ibsend()	Послати повідомлення (не блоковане, буферизоване)
MPI_Irsend()	Послати повідомлення не блоковане, в міру готовності)
MPI_Sendrecv()	Блоковане приймання і посилання
MPI_Sendrecv_replace()	Посилання і приймання в блокованому режимі
MPI_Buffer_attach()	Створення буфера
MPI_Buffer_detach()	Вімкнення буфера
MPI_Rsend()	Послати повідомлення (в міру готовності)
MPI_Probe()	Перевірити доставляння повідомлення (блоковане)
MPI_Iprobe()	Перевірити доставляння повідомлення (не блоковане)
MPI_Wait()	Блокування процесу до завершення приймання (передавання) повідомлення

Функція	Дія
MPI_Test()	Неблокована перевірка завершення доставляння 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

Ідентифікатор типу	Тип
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	Байтовий

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 передавання масиву даних зазначеної задачі
pvm_initsend()	Створення буфера передавання
pvm_recv()	Блоковане приймання повідомлення від зазначеної задачі
pvm_reduce()	Зведення 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 зазначеної задачі
pvm_tasks()	Інформація про задачі, яки виконуються в PVM
pvm_pstat()	Інформація про задачі з указаним ідентифікатором
pvm_pkbyte()/ pvm_upkbyte()	Пакування/Розпакування для типу byte (див. дод. Д.3)

Д.2. Типи даних

Ідентифікатор типу	Тип
pvm_str	Символьний
pvm_short	Короткий цілий
pvm_int	Цілий
pvm_cplx	Комплексний
pvm_float	З плаваючою точкою
pvm_double	Подвійна точність
pvm_long	Довгий цілий
pvm_byte	Байтовий
pvm_ushort	Короткий цілий
pvm_dcplx	Подвійний комплексний
pvm_ulong	Довгий цілий
pvm_uinit	Цілий

Д.3. Функції пакування-розпакування повідомлень

Ідентифікатор функції		Тип даних
Пакування	Розпакування	
pvm_pkcplx()	pvm_upkcplx()	float
pvm_pkbyte()	pvm_upkbyte()	byte
pvm_pkdcplx()	pvm_upkdcplx()	double
pvm_pkdouble()	pvm_upkdouble()	double
pvm_pkfloat()	pvm_upkfloat()	float
pvm_pkint()	pvm_upkint()	integer
pvm_pklong()	pvm_upklong()	long
pvm_pkshort()	pvm_upkshort()	short
pvm_pkstr()	pvm_upkstr()	char

Додаток Е

Бібліотека Pthreads

Е.1. Базові функції

Ідентифікатор функції	Функція
pthread_attr_init()	Ініціалізація атрибута потоку
pthread_attr_setscope()	Ініціалізація атрибутів потоку
pthread_attr_create()	Створення потоку
pthread_attr_exit()	Завершення потоку
pthread_attr_join()	Очікування завершення дочірнього потоку
sem_init()	Ініціалізація семафора
sem_wait()	Перевірка семафора
sem_post()	Перемикання семафора
pthread_mutex_init()	Ініціалізація мютекса
pthread_mutex_lock()	Захват мютекса
pthread_mutex_unlock()	Звільнення мютекса
pthread_cond_init()	Ініціалізація умовній змінній
pthread_cond_wait()	Блокування за умовною змінною
pthread_cond_signal()	Розблокування за умовною змінною

Додаток Ж

Мова C# і платформа .NET

Ж.1. Базові члени класу AppDomain

Ідентифікатор функції	Функція
CreateDomain()	Створення нового домену додатка в поточному процесі
GetCurreantThreadId()	Визначення ідентифікатора поточного потоку
Unload()	Вивантаження вказаного домену додатка
CreateIstance()	Створення екземпляра вказаного типу
ExecuteAssembly()	Виконання збирання
GetAssembly	Повернення списку збирання
Load()	Завантаження збирання в домен додатка

Ж.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

Ж.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

Навчальне видання

Жуков Ігор Анатолійович
Корочкін Олександр Володимирович

ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ

ISBN 966-7599-36-1

Редактор *Шульженко Р.М.*
Комп'ютерна верстка *Корочкін С.О.*

Видавництво «Корнійчук», 04116, Київ-116, а/с 4
Свідоцтво про внесення до Державного реєстру суб'єктів видавничої справи № 424 від 18.04.2001

Підп. до друку 17.10.2005 р. Формат 60x84 1/16.
Папір офсетний. Друк офсетний. Обл.-вид. арк.14,0.
Наклад 900. Зам. № 5-1790

ЗАТ «Віпол». ДК №15
03151, м. Київ, вул. Волинська, 60