Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

О.В. Корочкін, О.В Русанова

ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ

ВИБРАНІ РОЗДІЛИ

Навчальний посібник до кредитного модуля «Паралельні та розподілені обчислення» для студентів освітньої програми «Комп'ютерні системи та мережі», за спеціальністю 123 – "Комп'ютерна інженерія"

Київ
КПІ ім. Ігоря Сікорського
2020

УДК 681.3.06
Н72

Рецензент:
О.В.Тарасенко-Клятченко, канд. техн. наук, доц.,
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Відповідальний редактор
В. П. Сімоненко, доктор техн. наук, проф.,
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Корочкін О.В., Русанова О.В.
Н72 Паралельні та розподілені обчислення.Вибрані розділи: Навч. посібник. [Електронний ресурс] / О.В.Корочкін, Русанова О.В. – Електронні текстові дані (2 файла: 43,8 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2020. – 123 с.

Викладено засоби мови Ада для розробки програмного забезпечення для паралельних та розподілених систем. Мова використовується як базова в кредитному модулі "Паралельні та розподілені обчислення". Розглянуто засоби програмування для паралельних систем зі спільною пам'яттю: семафори, atomic-змінні, бар'єри, захищені типи. Наведено матеріал про можливості мови Ада для програмування для паралельних систем зі локальною пам'яттю: механізм рандеву. Розглянуті засоби мови Ада для програмування в розподілених комп'ютерних системах: сокети, виклик віддалених процедур (RPC).

Наведено приклади застосування мови для створення програмного забезпечення паралельних системах є різною структурною організацією.

Для студентів освітньої програми «Комп'ютерні системи та мережі» за спеціальністю 123 «Комп'ютерна інженерія», які вивчають проектування програмного забезпечення для паралельних комп'ютерних систем, а також буде корисним під час виконання курсових проектів, бакалаврських робіт та магістерських дисертацій, в яких використовуються паралельні комп'ютерні системи.

УДК 681.3.06

Паралельні та розподілені обчислення. Вибрані розділи: Навч. посібник до кредитного модуля «Паралельні та розподілені обчислення» для студентів освітньої програми «Комп'ютерні системи та мережі», за спеціальністю 123 – "Комп'ютерна інженерія" / Автори Корочкін О.В.. Русанова О.В. – К.: КПІ імені Ігоря Сікорського, 2020. – 123 с.

Гриф надано Методичною радою НТУУ «КПІ» 31.01.2020 протокол № 6

Навчальне видання

Паралельні та розподілені обчислення. Вибрані розділи

Навчальний посібник до кредитного модуля
"Паралельні та розподілені обчислення "
для студентів спеціальності
123 «Комп'ютерна інженерія»
ОПП "Комп'ютерні системи та мережі ",

Укладачі:
Корочкін Олександр Володимирович, канд. техн. наук, доц.
Русанова Ольга Веніамінівна, канд. техн. наук, доц.
Відповідальний редактор: В.П.Сімоненко, д.т.н., проф.
Рецензент: Тарасенко-Клятченко О.В. , канд. техн. наук, доц.

З М І С Т

В С Т У П

Навчальний посібник «Паралельні та розподілені обчислення. Вибрані розділи» пов'язаний з вивченням теорії і практики використанням засобів мови Ада для створення паралельних (розподілених) програм для паралельних комп'ютерних систем з різною архітектурою. Мова Ада є базовою у дисципліні « Паралельні та розподілені обчислення».

Головна мета посібника – отримання знань та умінь з практичного використання засов мови Ада для програмування паралельних і розподілених обчислень. Він містить три розділи. Кожен розділ закінчується завданнями для модульного контролю та самостійної роботи.

У першому розділі розглянуто практичне застосування засобів мови Ада для паралельних комп'ютерних систем за багатоядерною організацією (паралельних систем зі спільною пам'яттю). Розглянуто використання змінних, що поділяються (atomic, volatile), семафорів, бар'єрів, моніторів (захищених модулів).

У другому розділі розглянуто практичне застосування засобів мови Ада для паралельних комп'ютерних систем зі локальною пам'яттю. Ретельно розглянути особливості використання механізму рандеву (rendezvous mechanism), як ефективну реалізацію моделі взаємодії, що ґрунтується на посиланні повідомлень.

Третій розділ присвячено питанням програмування для розподілених комп'ютерних систем. Основну увагу приділено механізму виклику віддалених процедур (Remote Procedure Call, RPC), його реалізації та застосування.

Дані про авторів:

КОРОЧКІН Олександр Володимирович – канд. техн. наук, доцент кафедри обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського". Член Асоціації обчислювальної техніки (Association for Computing Machinery ACM), а також Європейської спілки Ada-Europe.

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

avcora@comsys.ntu-kpi.kiev.ua

РУСАНОВА Ольга Веніамінівна – канд. техн. наук, доцент кафедри обчислювальної техніки Національного технічного університету України "Київський політехнічний інститут імені Ігоря Сікорського".

Спеціаліст у галузі програмного забезпечення для паралельних комп'ютерних систем. Викладає дисципліни "Програмне забезпечення комп'ютерних систем", "Комп'ютерні системи".

avcora@comsys.ntu-kpi.kiev.ua

РОЗДІЛ 1.
Програмування для комп'ютерних систем зі спільною пам'яттю

Концепцію паралельного програмування в мові Ада реалізовано за допомогою спеціальних модулів. Процеси – модулі `task`, тому в посібнику для позначення процесів (потоків) використається термін "*задача*" [1-4, 12, 15, 17, 27, 29-31]. Задачні модулі дозволяють ефективно реалізувати всі необхідні операції: створення задачі або групи задач, внутрішню ідентифікацію задач через дискримінант задачного типу, призупинення задачі, встановлення пріоритету, визначення процесора (ядра) для виконання та інше. Головною проблемою, що пов'язана зі створенням паралельної програми, є організація взаємодії задач.

1.1. Взаємодія задач

Взаємодія задач пов'язана з двома основними завданнями:

- комунікацією задач;
- синхронізацією задач [7].

Комунікація задач передбачає обмін даними між задачами. При цьому інформація передається від однієї задачі до іншої в будь-якому напряму.

Синхронізація включає узгодження поведінки задач і залежність виконання однієї задачі від *подій*, що відбуваються в іншої задачі.

Реалізації взаємодії задач в багатоядерних комп'ютерних системах ґрунтується на використанні *моделі спільних змінних* .

У комп'ютерних системах зі спільною пам'яттю взаємодія задач на апаратному рівні виконується за допомогою спільної пам'яті. На програмному рівні комунікація і синхронізація задач виконується через *спільні змінні.*

Для передавання даних задача-відправник записує ці дані в спільні змінні, звідки їх зчитує задача-отримувач. Синхронізація задач виконується за допомогою відповідних спільних змінних,

які змінюються задаче, у якому видобувається подія, і зчитуються (аналізуються) задачею, яка чекає на подію.

Використання спільних змінних, що доступні будь-якої задачі , приводить до того, що в програмі може відбуватися одночасне звертання задач до спільних змінних, що призводить до конфлікту задач або некоректної роботи програми. Тому програмування з використанням моделі спільних змінних пов'язано з необхідністю вирішення специфічних проблем паралельного програмування, які загалом зводяться до розв'язання двох основних завдань – *взаємного виключення* і *синхронізації.*

Завдання взаємного виключення

Завдання взаємного виключення полягає в тому, що під час виконання двох і більше задач може виникнути одночасне звернення задач до одного і того ж *спільного ресурсу* (СР). Таке звернення зазвичай призводить до конфлікту задач, що полягає або в різкому уповільненні роботи паралельної програми, або в некоректній її роботі, якщо, наприклад, одна задача зчитує дані, а друга 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*) задачу, що звертається до спільного ресурсу, який вже використовується в цей момент іншою задачею. *Розблокування* задачі має бути виконано одразу після звільнення спільного ресурсу.

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

Класична схема організації такого контролю потребує використання операцій (примітивів) `ВХІДКД` і `ВИХІДКД`, що розміщу-

ються відповідно перед і після критичної ділянки, тобто обрамляють критичну ділянку, створюючи своєрідний "паркан" навколо неї (рис. 1.1).

Алгоритм виконання операції *ВХІДКД*:

1. Перевірити, чи знаходиться будь-який інша задача у критичний ділянці ?
2. Якщо знаходиться, то блокувати задачу, що виконує операцію *ВХІДКД*.
3. Якщо не знаходиться, то встановити заборону на входження всіх задач у свої критичної ділянки і дозволити цієї задачі увійти в критичну ділянку.

Алгоритм виконання операції *ВИХІДКД*:

1. Зняти заборону на входження задач в їх критичні ділянки.

Конкретна реалізація примітивів *ВХІДКД* і *ВИХІКД* у мові програмування Ада виконана у вигляді механізмів семафорів.

Другий підхід до вирішення завдання взаємного виключення передбачає контроль безпосередньо за спільним ресурсом (рис. 1.2). 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 задач до спільного ресурсу можливий тільки через виклик потрібної процедури монітора.

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

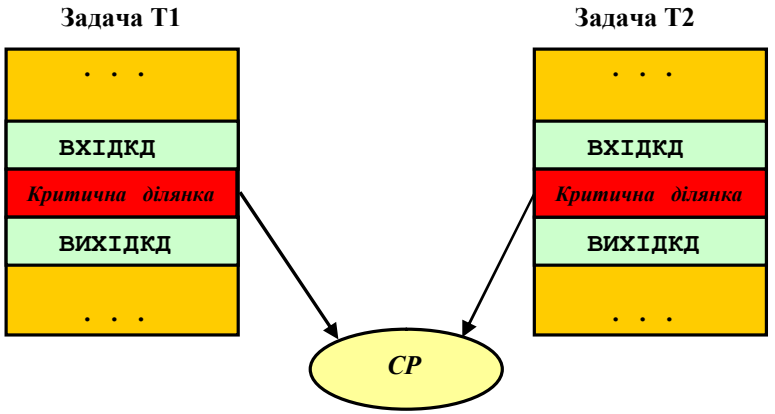

Рис. 1.1. Схема вирішення завдання взаємного виключення через на контроль задач

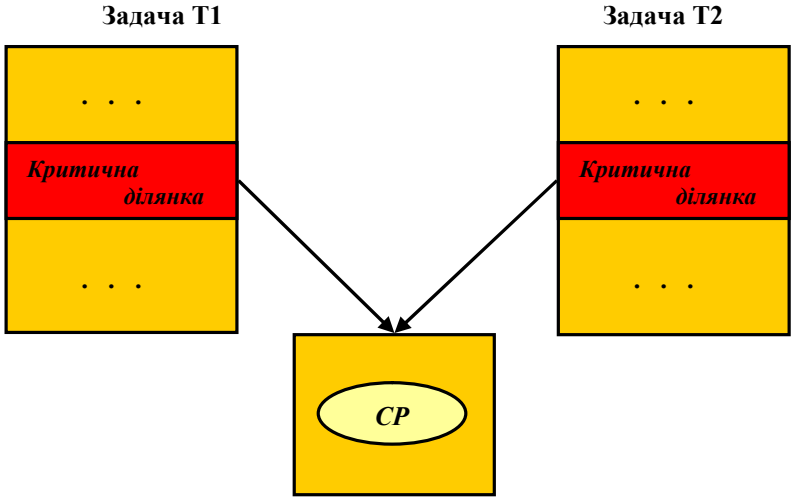

Рис. 1.2. Схема вирішення завдання взаємного виключення, через контролі спільного ресурсу

тільки однієї процедури. Якщо ввести термін «задача знаходиться в моніторі», під яким розуміти, що задача виконує будь-яку процедуру монітора, то можна стверджувати, що в моніторі може знаходитись тільки одна задача. Механізм моніторів реалізується у мові Ада за допомогою захищених модулів.

Завдання синхронізації

Завдання синхронізації двох задач полягає в тому, що в однієї задачі (*T1*) у визначеній точці (*точці події*) виконується подія (обчислення даних, уведення або виведення даних і та інше), а друга задача (*T2*) у визначеній точці (*точці очікування події*) блокується доти, доки ця подія не відбудеться і вона зможе продовжити своє виконання. Точка події і точка очікування – це *точки синхронізації*. Якщо задача `T1` вийшла на точку синхронізації, коли подія вже відбулася, то він не блокується і продовжує своє виконання.

Існує декілька схем синхронізації задач (рис. 1.3):

- одна задача очікує на подію в однієї задачі (*a*);
- декілька задач очікують на подію в однієї задачі (*b*);
- одна задача очікує на події в кількох задачах (*c*).

Точки `S` і `W` на рис. 1.3 – це точки синхронізації задач; `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. В мові Ада - це семафори та бар'єри.

1.2. Ада.Змінні, що поділяються

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

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


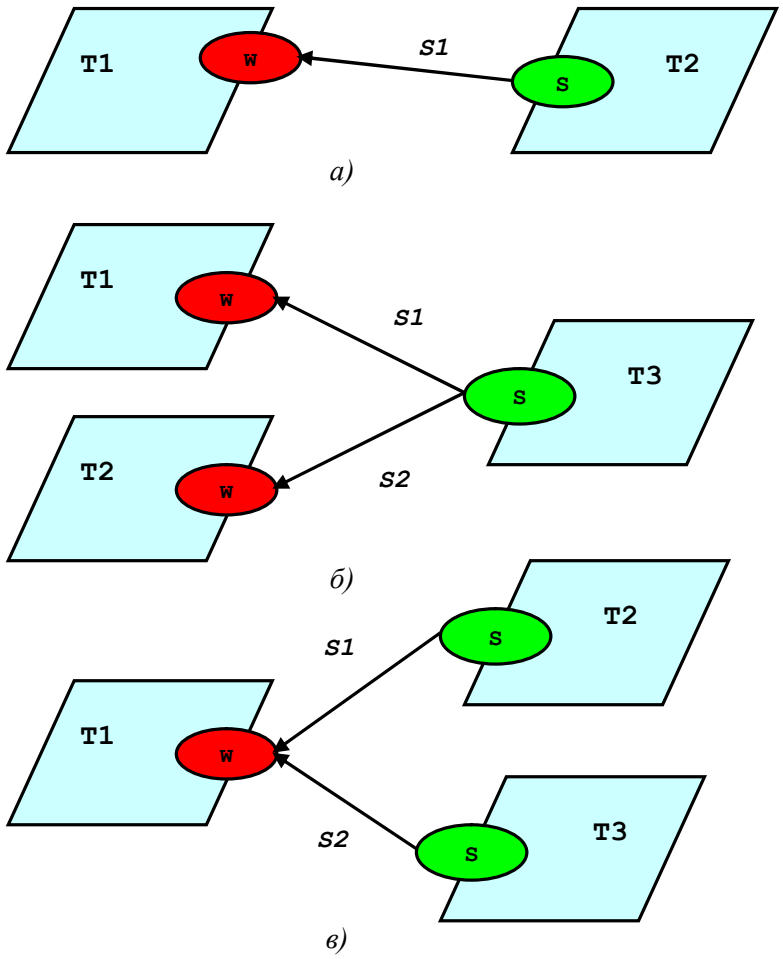

Рис. 1.3. Види синхронізації задач

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

❖ Приклад 1.1

```
------------------------------------------------------------
-- Pragma Atomic у завданні взаємного виключення  --
------------------------------------------------------------
procedure Lab1.1 is

   Буфер: integer:= 10;        -- спільний ресурс
   pragma Atomic(Буфер);       -- захищена (поділяєма)змінна

   task T1;
   task body T1 is
   begin
      put_line("Process T1 started");

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

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

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

      put_line("     Process T2 finished");
   end T2;

-- основна процедура
begin

   put_line("  Main procedure started ");

end Lab1.1;
```


❖ Приклад 1.2

```
------------------------------------------------------------
-- Pragma Atomic_Components у завданні взаємного виключення
------------------------------------------------------------
procedure Lab1.2 is

   type  Data is array (1..10) of integer;

   Буфер: Data;        -- спільний ресурс

   pragma Atomic_Components(Буфер);    -- захищена
                                       --(поділяєма)змінна

   task T1;
   task body T1 is
   begin
      put_line("Process T1 started");

      -- Операція зі спільним ресурсом
            Буфер(3) := Буфер(3) + 2;  -- критична ділянка

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

         -- Операція зі спільним ресурсом
         Буфер(4):= Буфер(3) - 25;     -- критична ділянка

      put_line("     Process T2 finished");
   end T2;

-- основна процедура
begin
   put_line("  Lab36 procedure started ");
end Lab1.2;
```

1.3 Ада.Бар'єри

Пакет Ada.Synchronous_Barriers надає тип Synchronous_Barrier, призначений для синхронізації виконання заданої кількості задач.

```
package Ada.Synchronous_Barriers is

   pragma Preelaborate(Synchronous_Barriers);

   subtype Barrier_Limit is Positive
                        range 1 .. implementation-defined;

   type Synchronous_Barrier (Release_Threshold :
                  Barrier_Limit) is limited private;

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

   -- not specified by the language

end Ada.Synchronous_Barriers;
```

Під час виклику процедури `Wait_For_Release` задача блокується на бар'єрі, поки не накопичитися вказана в дискримінанті бар'єра кількість задач, що очікують, і потім вони все будуть розблоковані і продовжать своє виконання.

❖ Приклад 1.3

```
------------------------------------------------------------
-- Бар'єр у завданні взаємного синхронізації              --
------------------------------------------------------------
with Ada.Synchronous_Barriers;
use  Ada.Synchronous_Barriers;
procedure Lab1.3 is

   Бар'єр1: Synchronous_Barrier(2);    -- бар'єр

   task T1;
   task body T1 is
   begin
      put_line("Task T1 started");

      -- Операція зі спільним ресурсом
           Буфер(3) := Буфер(3) + 2;  -- критична ділянка
```



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

         -- Операція зі спільним ресурсом
         Буфер(4):= Буфер(3) - 25;     -- критична ділянка

      put_line("     Task T2 finished");
   end T2;

-- основна процедура
begin

   put_line("  Lab36 procedure started ");

end Lab1.3;
```

Зміні пріоритету задачі.

Пакет `Ada.Dynamic_Priorities` дозволяє визначати, а також змінити базовий пріоритет задачі під час виконання програми.

```
with System;
with Ada.Task_Identification;
package Ada.Dynamic_Priorities is

    pragma Preelaborate(Dynamic_Priorities);

    procedure Set_Priority(
         Priority : in System.Any_Priority;
         T : in Ada.Task_Identification.Task_Id :=
               Ada.Task_Identification.Current_Task);

    function Get_Priority (
         T : Ada.Task_Identification.Task_Id :=
             Ada.Task_Identification.Current_Task)
                        return System.Any_Priority;

 end Ada.Dynamic_Priorities;
```

Розподіл задач по CPU.

Під час опису задачі можна визначити процесор (або діапазон процесорів) для виконання.

```
task T5 with CPU=> 4 is
```

Таке призначення можна також виконувати динамічно.

1.4 Ада.Семафори

Механізм семафорів запропонував математик Е.Дейкстра [16]. 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. Бінарні 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 задачу, що виконує цю операцію.
3. Інакше `S:= S - 1` і дозволити входження задачі 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, так і для синхронізації (рис. 1.4 - 1.5).

.

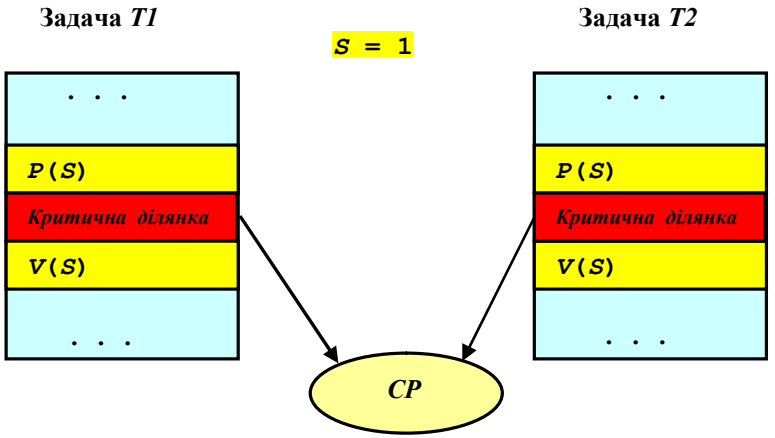

Рис. 1.4. Загальна схема вирішення завдання взаємного виключення з використанням семафорів

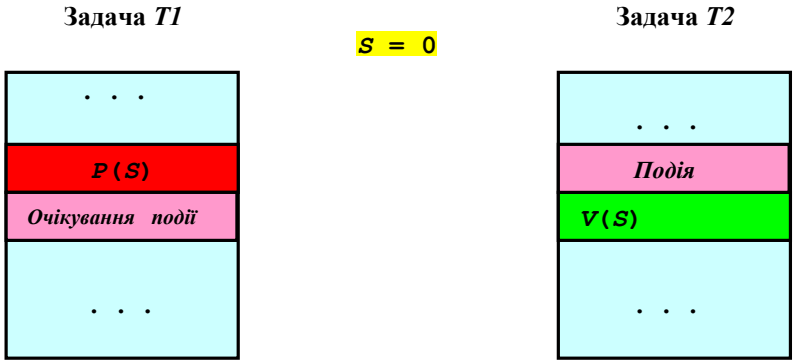

Рис. 1.5. Загальна схема вирішення завдання синхронізації з використанням семафорів

Механізм семафорів є універсальним, який може бути використаний як для вирішення завдання взаємного виключення, так і для синхронізації (рис. 1.4 - 1.5).

На рис. 1.6-1.7 надано використання Ада семафорів в завдання взаємного виключення та синхронізації.

Завдання взаємного виключення через Ада семафори

❖ **Приклад 1.4**

```
------------------------------------------------------------
--|Ада.Використання семафорів бібліотеки  WinAPI          --
--|в завданні взаємного виключення                        --
------------------------------------------------------------
with Ada.Synchronous_Task_Control;
use  Ada.Synchronous_Task_Control;
procedure Lab1.4 is

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

   -- створювання семафора
   Sem1: Suspension_Object;

   procedure Запуск_Задач is

      task T1;
      task T2;
      task T3;

      task body T1 is
         X1: integer;      -- локальна змінна
      begin
         put_line("Task T1 started");
             .  .  .
         --  Критична ділянка
         Suspend_Unil_True(Sem1);
            Ресурс := Ресурс + X1;     -- КД
         Set_True(Sem1);
             .  .  .
         put_line("Task T1 finished");

      end T1;

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



```
      task body T2 is
            X2: integer;      -- локальна змінна
         begin
            put_line("Task T2 started");
                .  .  .
            --  Критична ділянка
            Suspend_Unil_True(Sem1);
               Ресурс := Ресурс + X2;     -- КД
            Set_True(Sem1);

                .  .  .
            put_line("Task T2 finished");
         end T2;

         ----------------------------------------------

      task body T3 is
            X3: integer;      -- локальна змінна
         begin
            put_line("Task T3 started");
                .  .  .
            --  Критична ділянка
            Suspend_Unil_True(Sem1);
               Ресурс := Ресурс + X3;     -- КД
            Set_True(Sem1);

                .  .  .
            put_line("Task T3 finished");
         end T3;
      begin

         null;

      end Запуск_Задач;

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

      --  зміна початкового значення семафора на true
      Set_True(Sem1);

       -- виклик процедури для запуску задач
      Запуск_Задач;

end Lab1.4;
```

Синхронізація за допомогою Ада семафорів

❖ Приклад 1.5

```
-----------------------------------------------------------
-- Ада семафори в завданні синхронізації (рис. 1.3 а))
-----------------------------------------------------------
with Ada.Synchronous_Task_Control;
use  Ada.Synchronous_Task_Control;
procedure Lab1.5 is


   S1: Suspension_Object;        -- семафор (false)


   task T1;
   task body T1 is
   begin
      put_line("Process T1 started");
                . . .
      -- Точка очікування W
      Suspend_Util_True(S1);

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

         -- Подія

         Подія

         -- Сигнал про подію
         Set_True(S1);
                . . .
```

.

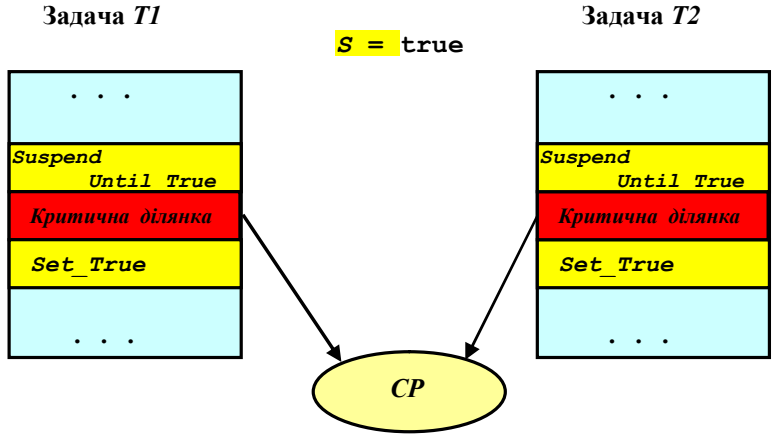

Рис. 1.6. Загальна схема вирішення завдання взаємного виключення з використанням Ада семафорів

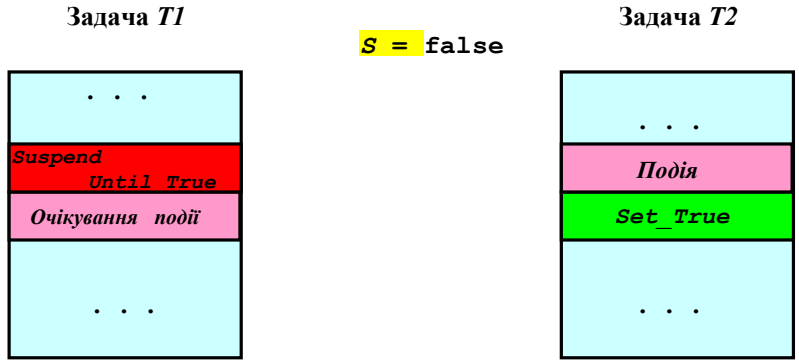

Рис. 1.7. Загальна схема вирішення завдання синхронізації з використанням Ада семафорів

```
      put_line("     Process T2 finished");
   end T2;

-- основна процедура
begin

   put_line("  Lab36 procedure started ");

end Lab1.5;
```

Приклад 1.9 ілюструє синхронізацію трьох задач, де одна задача (T3), в якої відбувається подія, надсилає сигнали двом іншим (T2 і T2). Через то, що в мові Ада присутні лише бінарні семафори (false, true), така синхронізація потребує наявність двох семафорів (S1 і S2). Реалізація подібної синхронізації через один бінарний семафор неможлива!

❖ Приклад 1.6

```
------------------------------------------------------------
-- Ада семафори в завданні синхронізації (рис. 1.3 б))
------------------------------------------------------------
with Ada.Synchronous_Task_Control;
use  Ada.Synchronous_Task_Control;
procedure Lab1.6 is


   S1, S2: Suspension_Object;       -- семафори (false)


   task T1;
   task body T1 is
   begin
      put_line(" Task T1 started");
               .  .  .
      -- Точка очікування  сигналу (W) від T3
      Suspend_Util_True(S1);


               .  .  .
      put_line("Task T1 finished");
   end T1;
   -------------------------------
   task T1;
   task body T1 is
   begin
      put_line(" Task T1 started");
               .  .  .
      -- Точка очікування сигналу  (W) від T3
```



```
      Suspend_Util_True(S2);

               .  .  .
      put_line(" Task T1 finished");
   end T1;
   -------------------------------

   task T3;
   task body T3 is
   begin
      put_line("     Task T3 started");

         -- Подія

         Подія

         -- Сигнали про подію в T1 і T2
         Set_True(S1);     -- для T1
         Set_True(S2);     -- для T2
               .  .  .
      put_line("     Task T3 finished");
   end T2;

-- основна процедура
begin

   put_line("  Lab1.9 procedure started ");

end Lab1.6;
```

❖ Приклад 1.7

```
------------------------------------------------------------
-- Ада семафори в завданні синхронізації (рис. 1.3 в))
------------------------------------------------------------
with Ada.Synchronous_Task_Control;
use  Ada.Synchronous_Task_Control;
procedure Lab.7 is


   S1, S2: Suspension_Object;       -- семафори


   task T1;
   task body T1 is
```

```
begin
   put_line("Task T1 started");
            .  .  .
   -- Точка очікування (W) сигналу від T2
   Suspend_Util_True(S1);

-- Точка очікування (W) сигналу від T3
   Suspend_Util_True(S2);


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

      -- Подія
```

Подія1

```
      -- Сигнал про подію
      Set_True(S1);  -- для T1


            .  .  .
   put_line("     Process T2 finished");
end T2;

------------------------------
task T3;
task body T3 is
begin
   put_line("     Task T3 started");

      -- Подія
```

Подія2

```
      -- Сигнал про подію
      Set_True(S2);  -- для T1
```



```
            .  .  .
      put_line("     Task T3 finished");
   end T3;

-- основна процедура
begin

   put_line("  Lab1.10 procedure started ");

end Lab1.7;
```

1.5. Використання WinAPI

Мова Ада має засоби (інтерфейси), які дозволяють використовувати ресурсі інших мов і бібліотек. В прикладі 1.8-1.9 показано використання семафорів, мютексів, критичних секцій бібліотеки WinAPI.

Завдання взаємного виключення.

❖ Приклад 1.8

```
-------------------------------------------------------------
--|Ада.Використання семафорів бібліотеки  WinAPI        --
--|в завданні взаємного виключення                       --
-------------------------------------------------------------
With Win32, Win32.winbase, Win32.winnt;
Use  Win32, Win32.winbase, Win32.winnt;
procedure Lab1.8 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("Task T1 started");
              .  .  .
         --  Критична ділянка
         Temp:= WaitForSingleObject(Sem1,Infinite);
            Ресурс := Ресурс + X1;      -- КД
         p1:= ReleaseSemaphore(Sem1,1,NULL);
              .  .  .
         put_line("Task T1 finished");
      end T1;
      ---------------
      task body T2 is
         X2: integer;      -- локальна змінна
      begin
         put_line("Task T2 started");
              .  .  .
         --  Критична ділянка
         Temp:= WaitForSingleObject(Sem1,Infinite);
            Ресурс := Ресурс + X2;      -- КД
         p1:= ReleaseSemaphore(Sem1,1, NULL);
              .  .  .
         put_line("Task T2 finished");
      end T2;
      --------------
      task body T3 is
         X3: integer;      -- локальна змінна
      begin
         put_line("Task T3 started");
              .  .  .
         --  Критична ділянка
         Temp:= WaitForSingleObject(Sem1,Infinite);
            Ресурс := Ресурс + X3;      -- КД
         p1:= ReleaseSemaphore(Sem1,1,NULL);
              .  .  .
         put_line("Task 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 Lab1.8;
```

❖ Приклад 1.9

```
-------------------------------------------------------------
--|Ада.Використання мютексів у бібліотеці  Win32          --
--|в завданні взаємного виключення                        --
-------------------------------------------------------------
procedure Lab1.9 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("Task T1 started");
              .  .  .
         --  Критична  ділянка
         Temp:= WaitForSingleObject(Mute,Infinite);
            Ресурс := Ресурс + X1;    -- КД
         p1:= ReleaseMutex(Mute);
              .  .  .
         put_line("Task T1 finished");
      end T1;
      ---------------
      task body T2 is
         X2: integer;      -- локальна  змінна
      begin
         put_line("Task T2 started");
```

```
            .  .  .
         --  Критична ділянка
         Temp:= WaitForSingleObject(Mute,Infinite);
            Ресурс := Ресурс + X2;    -- КД
         p1:= ReleaseMutex(Mute);
            .  .  .
         put_line("Task T2 finished");
      end T2;
      --------------
      task body T3 is
         X3: integer;      -- локальна  змінна
      begin
         put_line("Task T3 started");
            .  .  .
         --  Критична ділянка
         Temp:= WaitForSingleObject(Mute,Infinite);
            Ресурс := Ресурс + X3;    -- КД
         p1:= ReleaseMutex(Mute);
            .  .  .
         put_line("Task 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 Lab1.9;
```

❖ Приклад 1.10

```
-------------------------------------------------------------
--|Ада.Використання критичних секцій WinAPI              --
--|у задачі взаємного виключення                         --
-------------------------------------------------------------
procedure Lab1.10 is

   Ресурс: integer;  -- спільний ресурс
```



```
                    -- (глобальна змінна)

-- створювання критичної секції
CrCec: CRITICAL_SECTION;

 procedure Запуск_Задач is

    task T1;
    task T2;
    task T3;

    task body T1 is
       X1: integer:=20;          -- локальна змінна
    begin
       put_line("Task T1 started");
          .  .  .
       --  Критична ділянка
       EnterCriticalSection(CrSec);
          Ресурс := Ресурс + X1;    -- КД
       LeaveCriticalSEction(CrSec);
          .  .  .
       put_line("Task T1 finished");
    end T1;
    ---------------------------------------------
    task body T2 is
       X2: integer;             -- локальна змінна
    begin
       put_line("Task T2 started");
          .  .  .
       --  Критична ділянка
       EnterCriticalSection(CrSec);
          Ресурс := Ресурс + X2;    -- КД
       LeaveCriticalSEction(CrSec);
          .  .  .
       put_line("Task T2 finished");
    end T2;
    ---------------------------------------------
    task body T3 is
       X3: integer;             -- локальна змінна
    begin
       put_line("Task T3 started");
          .  .  .
       --  Критична ділянка
       EnterCriticalSection(CrSec);
          Ресурс := Ресурс + X3;    -- КД
       LeaveCriticalSEction(CrSec);
          .  .  .
```

```
         put_line("Task T3 finished");
      end T3;
   begin

      null;

   end Запуск_Задач;

begin     -- основна процедура

   put_line("  Main task started ");

   --  створення критичної секції
   InitializeCriticalSection(CrSec);

   -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab1.10;
```

Синхронізація

Застосування семафорів. У разі використання семафорів очікування події виконується за допомогою функції `WaitForSingleObject()`, а сигнал про подію, що відбулася, – через функцію `ReleaseSemaphore()`.

У прикладі 3.16 задача `T1` інформує задачу `T2` про подію, що відбулася в задачі `T1` (`Подія1`), а задача `T3` чекає на подію, що відбудеться в задачі `T2` (`Подія2`).

❖ Приклад 1.11

```
------------------------------------------------------------
--|Ада.Використання семафорів бібліотеки  Win32          --
--|в завданні синхронізації                               --
------------------------------------------------------------
procedure Lab1.11 is

   -- HANDLE  змінні для створювання семафорів
   Sem12, Sem23: HANDLE;

   -- допоміжні змінні
   Temp: DWORD;
   p1, p2: Boolean;
```



```
   procedure Запуск_Задач is

      task T1;
      task T2;
      task T3;

      task body T1 is
      begin
         put_line("Process T1 started");
              .  .  .
         -- Подія1
         --  Сигнал задачі T2 про подію (Подія1)
         --  у задачі T1
         p1:= ReleaseSemaphore(Sem12,1,NULL);
              .  .  .
         put_line("Process T1 finished");
      end T1;
      ------------------------------------------------
      task body T2 is
      begin
         put_line("Process T2 started");
              .  .  .
         --  очікування на подію (Подія1) в T1
         Temp:= WaitForSingleObject(Sem12,Infinite);
                   .  .  .
         -- Подія2
         -- Сигнал задачі T3 про подію (Подія2) в задачі T2
         p1:= ReleaseSemaphore(Sem23,1,NULL);
              .  .  .
         put_line("Process T2 finished");
      end T2;
      ------------------------------------------------
      task body T3 is
      begin
         put_line("Process T3 started");
              .  .  .
         --  Очікування на подію (Подія2) в T2
         Temp:= WaitForSingleObject(Sem23,Infinite);
              .  .  .
         put_line("Process T3 finished");
      end T3;
   begin
      null;
   end Запуск_Задач;

begin -- основна процедура
```

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

   --  створення семафорів з початковим значенням 0
   Sem12:=  CreateSemaphore(NULL, 0,1,NULL);
   Sem23:=  CreateSemaphore(NULL, 0,1,NULL);

   -- виклик процедури для запуску задач
   Запуск_Задач;
end Lab1.11;
```

У прикладі 3.17 задача T3 чекає на події, що відбудуться в задачі T1 (Подія1) і задачі T2 (Подія2).

❖ Приклад 1.12

```
------------------------------------------------------------
--|Ада.Використання семафорів бібліотеки  Win32          --
--|у завданні синхронізації                              --
------------------------------------------------------------
procedure Lab1.12 is
   -- HANDLE  змінні для створювання семафорів
   Sem1_3, Sem2_3: HANDLE;
       -- допоміжні змінні
   Temp: DWORD;
   p1, p2: Boolean;
   procedure Запуск_Задач is

      task T1;
      task T2;
      task T3;

      task body T1 is
      begin
         put_line("Process T1 started");
             .  .  .
         -- Подія1
         -- Сигнал задачі T3 про подію (Подія1) в задачі T1
         p1:= ReleaseSemaphore(Sem1_3,1,NULL);
             .  .  .
         put_line("Process T1 finished");
      end T1;
      ---------------
      task body T2 is
      begin
         put_line("Process T2 started");
             .  .  .
         -- Подія2
```



```
         -- Сигнал задачі T3 про подію (Подія2) в задачі T2
         p1:= ReleaseSemaphore(Sem2_3,1,NULL);
             .  .  .
         put_line("Process T2 finished");
      end T2;
      --------------
      task body T3 is
      begin
         put_line("Process T3 started");
             .  .  .
         --  Очікування на подію (Подія1) в T1
         Temp:= WaitForSingleObject(Sem1_3,Infinite);
         --  Очікування на подію (Подія2) в T2
         Temp:= WaitForSingleObject(Sem2_3,Infinite);
             .  .  .
         put_line("Process T3 finished");
      end T3;

   begin
      null;
   end Запуск_Задач;

begin -- основна процедура

   put_line("  Main procedure started ");
   --  створення семафорів з початковими значеннями 0
   Sem1_3:=  CreateSemaphore(NULL, 0, 1,NULL);
   Sem2_3:=  CreateSemaphore(NULL, 0, 1,NULL);

   -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab1.12;
```

У прикладі 1.13 задача T1 інформує задачі T2 і T3 про подію, що відбулася в задачі T1. Це можливо виконати через два бінарні семафори, але в прикладі це виконано за допомогою одного множинного семафора Sem1_23, який набуває значення 0 .. 2.

❖ Приклад 1.13

```
------------------------------------------------------------
--|Ада.Використання семафорів бібліотеки  WinAPI         --
--|в завданні синхронізації                              --
------------------------------------------------------------
procedure Lab1.13 is
```

```
-- HANDLE  змінна для створювання богатозначного семафора
Sem1_23: HANDLE;

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

procedure Запуск_Задач is

   task T1;
   task T2;
   task T3;

   task body T1 is
   begin
      put_line("Process T1 started");
          .  .  .
      -- Подія
      --  Сигнали задачам T2 і T3 про подію в задачі T1
      p1:= ReleaseSemaphore(Sem1_23,2,NULL);
          .  .  .
      put_line("Process T1 finished");
   end T1;
   ------------------------------------------------
   task body T2 is
   begin
      put_line("Process T2 started");
          .  .  .
      --  Очікування на подію в T1
      Temp:= WaitForSingleObject(Sem1_23,Infinite);
          .  .  .
      put_line("Process T2 finished");
   end T2;
   ------------------------------------------------
   task body T3 is
   begin
      put_line("Process T3 started");
          .  .  .
      --  Очікування на подію в T1
      Temp:= WaitForSingleObject(Sem1_23,Infinite);
          .  .  .
      put_line("Process T3 finished");
   end T3;
begin
   null;
end Запуск_Задач;
```



```
begin -- основна процедура
   put_line("  Main procedure started ");
   --  створення семафора зі значеннями (0,1,2)  і
   --  початковим значенням 0
   Sem1_23:=  CreateSemaphore(NULL, 0, 2, NULL);

   -- виклик процедури для запуску задач
   Запуск_Задач;
end Lab1.13;
```

Застосування подій. Механізм подій призначений виключно для синхронізації процесів.

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

```
HANDLE CreateEvent(
   LPSECURITY_ATTRIBUTES    адреса_атрибутів_захисту
   BOOL     прапор_ручної_установки_події
   BOOL     прапор_початкового_стану
   LPCTSTR  адреса_об'єкта_події
);
```

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

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

Для роботи з подіями використовують також функції:

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

У прикладах 1.14 – 1.16 наведено вирішення за допомогою механізму повідомлень завдання синхронізації, які розглядалися в прикладах 1.11 – 1.13.

У прикладі 1.14 задача T1 інформує задачу T2 про подію, що відбулася в задачі T1 (Подія1), а задача T3 чекає на подію, що відбудеться в задачі T2 (Подія2).

❖ Приклад 1.14

```
------------------------------------------------------------
--|Ада.Використання подій бібліотеки  WinAPI              --
--|в завданні синхронізації                               --
------------------------------------------------------------
procedure Lab1.14 is

   -- HANDLE  змінна для створювання  подій
   Evn12, Evn23: HANDLE;

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

   procedure Запуск_Задач is

      task T1;
      task T2;
      task T3;

      task body T1 is
      begin
         put_line("Process T1 started");
             .  .  .
         -- Подія1
         -- Сигнал задачі T2 про подію (Подія1)в задачі T1
         p1:= SetEvent(Evn12);
             .  .  .
         put_line("Process T1 finished");
      end T1;
      ---------------
      task body T2 is
      begin
         put_line("Process T2 started");
             .  .  .
         --  очікування на подію (Подію1) в T1
         Temp:= WaitForSingleObject(Evn12,Infinite);
                  .  .  .
```



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

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

   --  створення подій
   Evn12:=  CreateEvent(NULL, 0,0, NULL);
   Evn23:=  CreateEvent(NULL, 0,0, NULL);

   -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab1.14;
```

У прикладі 3.20 задача T3 чекає на події, що відбудуться в задачі T1 (Подія1) і задачі T2 (Подія2). Для сигналізації про ці події застосовані події Evn1_3 та Evn2_3.

❖ Приклад 1.15

```
------------------------------------------------------------
--|Ада.Використання подій бібліотеки  WinAPI              --
--|в завданні синхронізації                               --
------------------------------------------------------------
procedure Lab1.15 is

   -- HANDLE  змінна для створювання  подій
```

```
Evn1_3, Evn2_3: HANDLE;

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

procedure Запуск_Задач is

   task T1;
   task T2;
   task T3;

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



```
   end Запуск_Задач;

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

   --  створення подій
   Evn1_3:=  CreateEvent(NULL, 0, 0, NULL);
   Evn2_3:=  CreateEvent(NULL, 0, 0, NULL);
   -- виклик процедури для запуску задач
   Запуск_Задач;

end Lab1.15;
```

У прикладі 1.16 задача T1 інформує задачі T2 і T3 про подію, що відбулася в задачі T1. Це можливо виконати через одну подію Evn1_23.

❖ Приклад 1.16

```
------------------------------------------------------------
--|Ада. Використання подій бібліотеки  WinAPI             --
--|в завданні синхронізації                               --
------------------------------------------------------------
procedure Lab1.16 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 Lab1.16;
```

1.6 Ада. Захищені модулі (типи)
Монітори

Монітор – програмний модуль, що містить спільні ресурси (захищені змінні) та процедури для роботи з ними, причому доступ до змінних можливий *тільки через процедури* монітора [28].

Монітор – ефективний засіб розподілу ресурсів і взаємодії процесів. Це призначення монітора реалізується за допомогою властивостей, якими наділені процедури монітора. Характерна особливість процедур монітора – *взаємне виключення*.

У будь-який момент часу може виконуватися лише *одна* процедура монітора.

Якщо процес викликав і виконує процедуру монітора, то жоден процес не може виконувати будь-які процедури цього монітора. За спроби виклику іншою задачею процедури, що виконується, або іншої процедури цього монітору, вона блокується і розміщується в *черзі блокованих задач* доти, доки активна задача не закінчить виконання вже розпочатої процедури монітора. Тобто в моніторі не може "знаходитись" більше однієї задачі. Така властивість процедур монітора забезпечує взаємне виключення задач, 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 синхронізовані 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  Лічильник;

    N: integer;        -- спільний ресурс

   procedure Додати_Значення(T: inout integer);
   procedure Зменшити_Значення(T:inout  integer);
```

```
   procedure Читати(T: out  integer);
   procedure Очистити(T: in out integer);

begin
      N:= 0; -- ініціалізація лічильника (СР)
end  Лічильник;
```

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).

Монітори в мові Ада

Механізм моніторів у другому стандарті мови Ада реалізований у вигляді спеціального програмного модуля (типа) – захищеного модуля (`protected units`) [5,25,31]. Призначення модулю – розширення можливості мови Ада для програмування задач, зокрема, для вирішення проблеми доступу до спільних ресурсів і синхронізації процесів. Захищені модулі також забезпечують підтримку різних парадигм систем реального часу, для розроблення яких мову Ада призначена в першу чергу.

Спільні ресурси (захищені елементи) і операції над ними (захищені операції) об'єднуються в захищеному модулі, аналогічно тому, як це робиться в інших модулях мови Ада. Доступ до спільних ресурсів можливий лише 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;
```

Види захищених операцій:

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

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

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

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

Приватна частина (private) 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 1.17

```
-------------------------------------------------------------
--  Ада.Захищений модуль.Приклади специфікації.            --
-------------------------------------------------------------
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 для формування бар'єра
   Стан: float;
```

```
end  Сенсор;
protected   Блок2(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  Бло2;
```

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 або *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 1.12 показано реалізацію тіла захищеного модуля `Сенсор`, специфікацію якого наведено раніше у прикладі 1.10:

❖ **Приклад 1.18**

```
------------------------------------------------------------
-- Ада. Тіло захищеного модуля                             --
------------------------------------------------------------
   protected  body   Сенсор  is
       -- тіло захищеного  входу з бар'єром
      entry   Чекати  when  Прп is
      begin
         Прп:= False;
      end  Чекати;

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

      procedure Зміна_Стану(x : in  float) is
      begin
         Стан := x;
      end Зміна_Стану;

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

   end  Сенсор;
```

Структуру захищеного модуля `Сенсор` показано на рис. 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 за-

хищеної 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.

Синхронізація через захищений модуль

На рис. 1.8 представлено структуру захищеного модуля для синхронізації двох задач.

Задача T1 чекає на подію, що видобувається в задачі T2. Для цього вона викликає вхід `ЧекатиНаПодію`. Вхід має бар'єр `F=1`. Як що значення F дорівнює 1, то вхід викритий, інакше - закритий. Початкове значення F дорівнює 0, тобто вхід спочатку закритий, задача T1 буде блокована і розміщені в черзі,що пов'язана з цим входом.

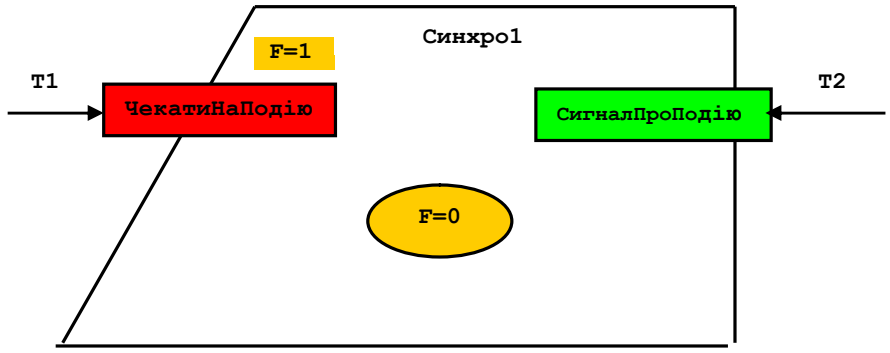

Рис. 1.8. Синхронізація двох задач через захищений модуль `Синхро1`

Задача T2 після події надсилає задаче T1 сигнал шляхом відклику процедури `СигналПроПодію`, яка змінює значення змінної F на одиницю. Зміна F в свою чергу змінює значення бар'єру входу `ЧекатиНаПодію`, який тепер відкритий. Якщо задача T1 раніше була блокована, то вона буде розблокована і буде виконувати тіло входу.

❖ Приклад 1.19

```
-------------------------------------------------------------
-- Ада.Захищений модуль.Синхронізація двох задач          --
-------------------------------------------------------------
procedure Lab1.19 is
    protected   Синхро1    is
        procedure  СигналПроПодію;
        entry      ЧекатиНаПодію;
    private
        F:  Integer:= 0;  -- прапор для формування бар'єра
    end Синхро1;

 -- тіло захищеного модуля
 protected  body  Синхро1  is
  procedure  СигналПроПодію is
     begin
          F:= 1;
     end СигналПроПодію;
     entry  ЧекатиНаПодію  when F=1 is
     begin
         null;
     end  ЧекатиНаПодію;
  end   Синхро1;
 ------------------------------------------------
  task T1;
  task body T1 is
  begin
      put_line("Process T1 started");
                 .  .  .
      -- Точка очікування
      Синхро1.ЧекатиНаПодію;
                 .  .  .
      put_line("Process T1 finished");
   end T1;
   -------------------------------
   task T2;
   task body T2 is
   begin
      put_line("     Task T2 started");
```

```
        -- Подія
```

Подія

```
        -- Сигнал про подію
        Синхро1.СигналПроПодію;
                  . . .
     put_line("     Process T2 finished");
   end T2;
-- основна процедура
begin
   put_line("  Lab36 procedure started ");
end Lab1.19;
```

Синхронізація кількох задач.

Варіант 1.В прикладі 1.14 задача T1 чекає на події, що видобуваються в трьох задачах T2-T4. Розглянемо Синхро2 - модифікацію захищеного модуля Синхро1 для реалізації такої взаємодії задач T1-T5 (Рис.1.9).

Модифікація потребує двох змін:

- бар'єра у захищеному вході `ЧекатиНаПодію (when F=P-1)`, де P – кількість задач в програмі;
- процедури `СигналПроПодію` , де буде виконуватися дія F:=F+1 (накопичування сигналів від задач T2-T4).

На рис. 1 .9 представлено структуру захищеного Синхро2

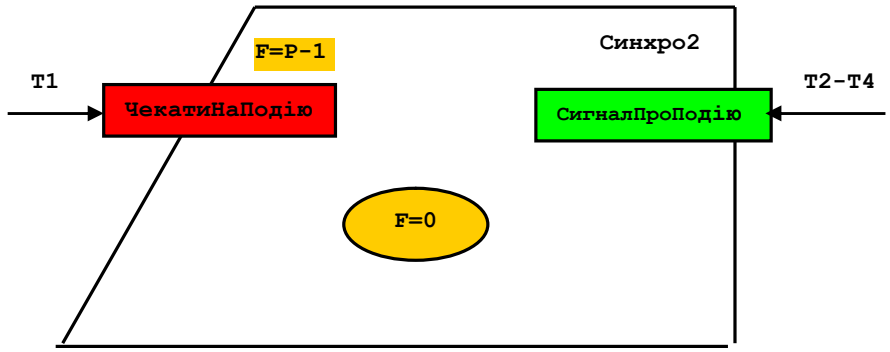

Рис. 1.9. Синхронізація двох задач через захищений модуль `Синхро2`

❖ Приклад 1.20

```
-------------------------------------------------------------
-- Ада.Захищений модуль.Синхронізація  задач T1-T4        --
-------------------------------------------------------------
 protected   Синхро2    is
     procedure  СигналПроПодію;
     entry      ЧекатиНаПодію;
 private
     F:  integer:= 0;     -- прапор для формування бар'єра
     P:  integer:= 4;     -- кількість задач (T1-T4)
 end Синхро2;

 -- тіло захищеного модуля
 protected  body  Синхро2  is

     procedure  СигналПроПодію is
     begin
          F:= F + 1;
     end СигналПроПодію;

     entry  ЧекатиНаПодію  when F=P-1 is
     begin
         null;
     end  ЧекатиНаПодію;
  end   Синхро2;
 ------------------------------------------

 task T1;
 task body T1 is
 begin
     put_line("Task T1 started");
               . . .
     -- Точка очікування
     Синхро2.ЧекатиНаПодію;

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

        -- Подія
```

Подія

```
        -- Сигнал про подію
        Синхро1.СигналПроПодію;
                . . .
    put_line("     Task T2 finished");
end T2;
------------------------------
task T3;
task body T3 is
begin
    put_line("     Task T3 started");
        -- Подія
```

Подія

```
        -- Сигнал про подію
        Синхро1.СигналПроПодію;
                . . .
    put_line("     Task  T3 finished");
end T3;
------------------------------
task T4;
task body T4 is
begin
    put_line("     Task T4 started");

        -- Подія
```

Подія

```
        -- Сигнал про подію
        Синхро1.СигналПроПодію;
                . . .
    put_line("     Process T4 finished");
end T4;
-- основна процедура
begin
   put_line("  Lab1.14 procedure started ");
end Lab1.20;
```


Захищений модуль Синхро2 ефективно реалізує розглянуту синхронізацію, прикладом якої є так звана синхронізація по завершенню обчислень (по виведенню результату). Тут одна задача чекає на завершення обчислень в інших і потім здійснює виведення повного результату виконання обчислень в задачах.

Варіант 2. В прикладі 1.15 задачі T2-T4 чекають на подію, що видобувається в задачі T1. Захищений модуль Синхро1 може бути використаний для такої реалізації взаємодії задач T1-T5 (Рис.1.8) без модифікації.

На рис. 1 .10 представлено взаємодія (синхронізація) задач T1-T5 через захищений модуль Синхро1.

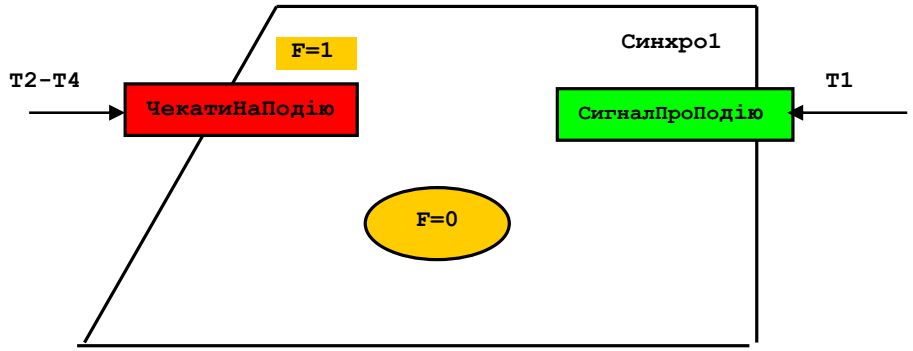

Рис. 1.10. Синхронізація задач через захищений модуль Синхро1. Версія 2

❖ Приклад 1.21

```
-------------------------------------------------------------
-- Ада.Захищений модуль.Синхронізація  задач T1-T4.Версія2-
-------------------------------------------------------------
Procedure Lab 1.21
   protected   Синхро1     is
      procedure  СигналПроПодію;
      entry      ЧекатиНаПодію;
   private
```

```
    F:  integer:= 0; -- прапор для формування бар'єра
  end Синхро1;

-- тіло захищеного модуля
protected  body  Синхро2  is

    procedure  СигналПроПодію is
    begin
         F:=1;
    end СигналПроПодію;

    entry  ЧекатиНаПодію  when F= 1 is
    begin
        null;
    end  ЧекатиНаПодію;
 end   Синхро1;
 ------------------------------------------------------------------

 task T2;
 task body T2 is
 begin
     put_line("Task T2 started");
                . . .
     -- Точка очікування
     Синхро2.ЧекатиНаПодію;
                . . .
     put_line("Task T2 finished");
  end T2;
  -------------------------------
  task T1;
  task body T1 is
  begin
     put_line("     Task T1 started");

        -- Подія

        Подія

        -- Сигнал про подію
        Синхро1.СигналПроПодію;
                . . .
     put_line("     Task T1 finished");
  end T1;
  -------------------------------
  task T3;
```



```
    task body T3 is
    begin
       put_line("     Task T3 started");
                . . .
       -- Точка очікування
       Синхро1.ЧекатиНаПодію;
                . . .
       put_line("     Task  T3 finished");
    end T3;
    -------------------------------
    task T4;
    task body T4 is
    begin
       put_line("     Task T4 started");
                . . .
       -- Точка очікування
       Синхро2.ЧекатиНаПодію;
                . . .
       put_line("     Process T4 finished");
    end T4;
-- основна процедура
begin
   put_line("  Lab1.14 procedure started ");
end Lab1.21;
```

Приклад 1.22 ефективно реалізує розглянуту синхронізацію, прикладом якої є так звана синхронізація по введенню даних. Тут задачі Т2-Т4 чекають на введення даних в задачі Т1, після чого вони розблокуються і зможуть розпочати обчислення.

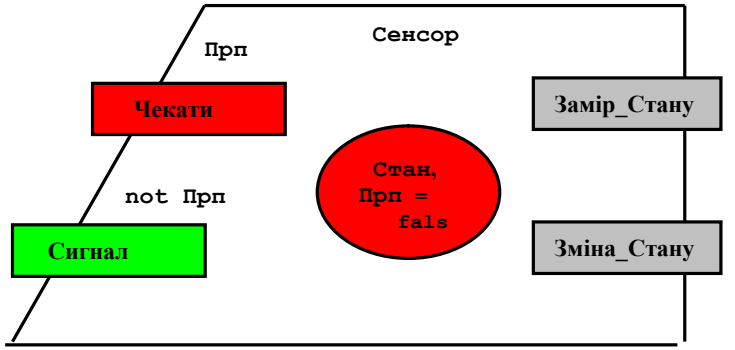

Рис. 1.11. Структура захищеного модуля Сенсор

❖ Приклад 1.22

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

 -- тіло захищеного модуля
 protected  body  Буфер  is

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

Задачі додають або зменшують значення змінної Лічильник, викликаючи процедури Додати і Видалити 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 1.23 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 1.23

```
-------------------------------------------------------------
-- Ада.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;
```

1.7 Ада в програмуванні для систем зі спільною пам'яттю

Використання семафорів, бар'єрів та атоміс змінних

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

Вхідні дані:

- комп'ютерна система зі спільною пам'яттю містить два процесори і два пристрої введення-виведення (ПВВ) (рис. 1.12),
- математичне завдання: операція додавання векторів та скалярним множенням векторів Z і R

  $A = B + C * (Z*R),$

  де *A, B, C ,Z, R* – вектори розміру *N;*
- введення векторів *B* і Z виконується в ПВВ1, введення векторів *C і R* – у ПВВ2, виведення результату вектор *A* – у ПВВ1.

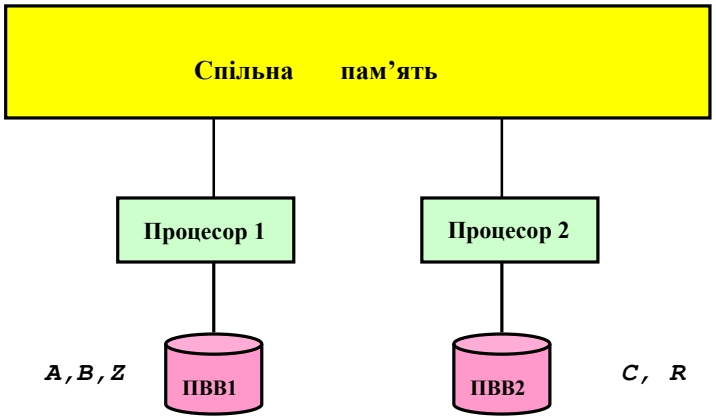

Рис. 1.12. Структурна схема двопроцесорної системи

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

$$\begin{array}{ll} 1 & x_i = (Z_н * R_н) \\ 2 & x = x + x_i \\ 3 & A_н = B_н + C_н * x \end{array} \qquad (1.1)$$

CP: x

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

Співвідношення (1.1) визначає дії в кожному процесорі КС під час виконанні обчислень.

Потрібно проаналізувати співвідношення (1.1) на наявність спільних ресурсів (спільних змінних), які можуть одночасно використовуватися процесами як для читання, так і для змінювання. Задачі використовують різні частини векторів, тому вони не є спільними ресурсами. Спільним ресурсом у співвідношенні (1.1) є скаляр x, відносно якого необхідно розв'язати завдання взаємного виключення виконанням обчислення 2 у (1.1) у критичної ділянці.

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

- *введення даних*, коли обидва процеси чекають на введення всіх даних і тільки після цього починають обчислення,
- завершення обчислення x,
- *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 *Z*	
2.	**Сигнал** задачі *T2* про введення *B* та *Z*	-- *S2-1*
3.	**Чекати** на введення *C, R* у задачі *T2*	-- *W2-1*
4.	$x_1 = (Z_н * R_н)$	
5.	$x = x + x_1$ -- Критична ділянка (КД)	
6.	**Сигнал** задачі *T2* завершення обчислень x	-- *S2-2*
7.	**Чекати** на завершення обчислень x в *T2*	-- *W2-2*
8.	$A_н = B_н + C_н * x$	
9.	**Чекати** на завершення обчислень в *T2*	-- *W2-3*
10.	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.	$x_2 = (Z_н * R_н)$	
5.	$x = x + x_2$ -- (КД)	
6.	**Сигнал** задачі *T1* завершення обчислень x	-- *S1-2*
7.	**Чекати** на завершення обчислень x в *T1*	-- *W1-2*
8.	$A_н = B_н + C_н * x$	
9.	**Сигнал** *T1* про завершення обчислень	-- *S1-3*

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 [5]. 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. 1.13) 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.

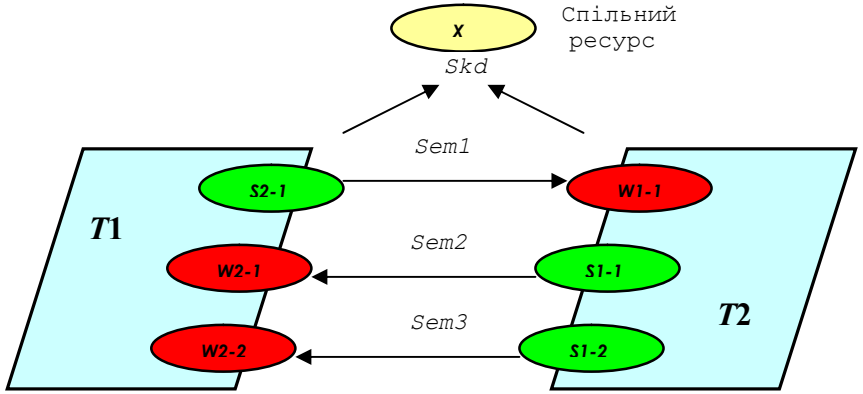

Рис. 1.13. Структурна схема взаємодії задач

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

```
------------------------------------------------------------
--|Ада.Використання семафорів                              --
--|Операція   A = B + C * (Z*R)                            --
------------------------------------------------------------
procedure Lab1.24 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,Z,R : Вектор;          -- глобальні змінні

   -- семафори
   Sem1, Sem2, Sem3, Sem4, Sem5, 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(" Task T2 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(" Task 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 Lab1.24;
```

Використання захищених модулів

Реалізація операції додавання векторів у двопроцесорній системі зі спільною пам'яттю (рис. 1.14) з використанням захищеного модуля мови Ада**.**

Етапи 1–3 не будуть відрізнятися від етапів, розглянутих раніше.

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. 1.12) включає три захищені елементи: `x`, `F1` і `F2`, а також набір захищених операцій:

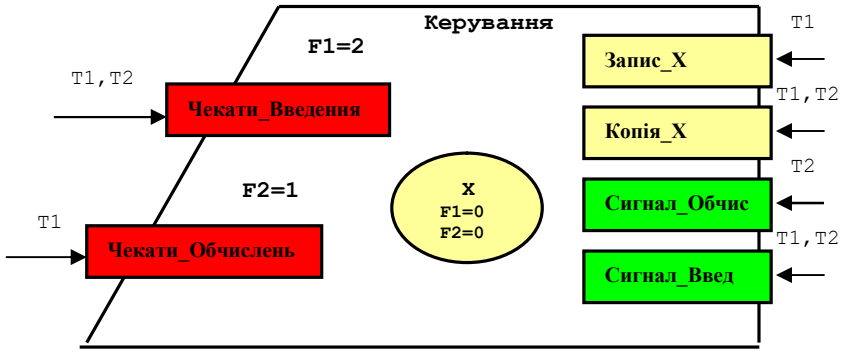

Рис. 1.14. Захищений модуль в операції `A = B + C * x`

- 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.

```
------------------------------------------------------------
--|Ада.Захищений модуль у задачі синхронізації           --
--|Операція  A = B + C * x                               --
------------------------------------------------------------
procedure Lab1.25 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 Lab1.25;
```

➢ **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. 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 Ада ?

➢ **Завдання для самостійної роботи**

1. Написати програму мовою Ада, 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 WinAPI.
2. Вирішити за допомогою семафорів WinAPI завдання синхронізації одного потоку з кількома іншими (потік чекає на події в трьох інших потоках).
3. Вирішити за допомогою засобів WinAPI завдання синхронізації кількох потоків з одним (чотири потоки чекають на подію в одному потоці).
4. Завдання 3 реалізувати за допомогою захищеного модулю мови Ада.
5. Завдання 4 реалізувати за допомогою захищеного модуля мови Ада .
6. Виконати моделювання механізму богатозначних семафорів за допомогою захищеного модуля мови Ада.
7. Провести дослідження часу виконання в ПКС 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 2.
Програмування для комп'ютерних систем зі локальною пам'ятю

Цей розділ присвячено питанням організації обчислень в комп'ютерних системах зі розподіленою пам'яттю та використанням *механізму рандеву* (rendezvous) мови Ада.

2.1 Ада. Механізм рандеву

Реалізація моделі посилання повідомлень в мові Ада здійснена через механізм *рандеву* [30]. Складовими механізму рандеву є :

- оператори входу `entry`,
- оператор прийняття виклику входу `accept`,
- оператор відбору `select`.

Рандеву підтримує модель клієнт-сервер, де задача-клієнт ініціює звертання до задачі-сервера, а задача-сервер забезпечує відгук на це звертання шляхом приймання (передавання) даних від задачі-клієнта з можливістю обробки цих даних під час рандеву.

Рандеву реалізує *асиметричну* модель взаємодії де задача-клієнт знає до кого вона звертається, а задача-сервер – ні. Також взаємодія задач в механізмі рандеву *синхронізована*, тобто вона відбувається лише тоді, коли задача-клієнт і задача-сервер одночасно "зустрічаються" на місці, де заплановано зустріч. В противному випадку задача, яка першою прийшла на "побачення", блокується и чекає на появу задачі, що спізнілася.

Головне призначення механізму рандеву – передавання *різноманітних* даних між задачами в будь якому напрямку з можливістю також обміну даними. Рандеву завжди відбувається між двома задачами. Якщо є звертання задачі-клієнта до задачі-сервера, яка вже взаємодіє з іншою задачею-клієнтом, то така клієнтська задача блокується до завершення вже розпочатого рандеву.

Приклад 2.1 ілюструє використання механізму рандеву для організації взаємодії задач `T1` і `T1`, де задача `T1` передає в задачу `T2` ціле число x. Схему взаємодії задач показано на рис. 2.1.

Для взаємодії задач у специфікації задачі `T2` описаний вхід `Data`, специфікація якого дозволяє приймати дані у вигляді одного цілого числа в задачу `T2`. Посилання цілого числа зі задачі `T1` виконано як виклик входу в `T1` входу `Data` з підстановкою фактичного параметра. Тобто існує схожість між входом та процедурою по опису та використанню. Опис входу через оператор `entry` задає *специфікацію* входу:

- назву входу,
- список формальних параметрів,
- зазначення типу параметрів,
- напряму передавання даних .

Тіло входу реалізується окремо через оператор `accept` в тілі задачі-сервера, а виклик входу відбувається в задачі-клієнті з застосуванням фактичних параметрів.

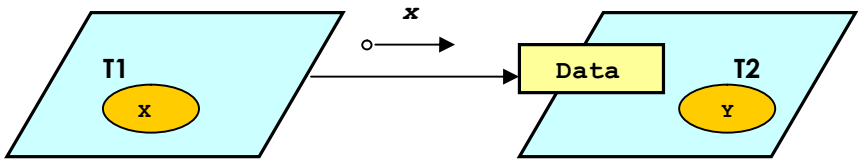

Рис. 2.1. Схема взаємодії задач. Версія 1.

❖ Приклад 2.1

```
------------------------------------------------------------
--  Ада.Механізм рандеву. Версія 1                        --
------------------------------------------------------------
procedure Lab2.1 is

   task T1;   -- вироджена специфікація задачі

   task T2 is
      entry Data(e: in integer);          -- опис входу
   end T2;

   task body T1 is
      x: integer := 25;        -- локальна  змінна
   begin
         . . .
      B.Data(x);    -- виклик входу Data задачі T2
```

```
                    -- для передавання x
          .  .  .
   end T1;

   task body T2 is
      y: integer;          -- локальна   змінна
   begin
         .  .  .
      -- оператор приймання виклику входу
      accept Data(e: in integer) do
         y:= e;  -- тіло оператора accept (прийманні x в y)
      end Data;

  end T2;

            -- основна програма
begin

   null;      -- порожній оператор

end Lab2.1;
```

Розміщення входу Data в задачі T2, тобто вибір задачі-сервера може бути змінено і вхід Data може бути описаний в задачі T1, яка автоматично стає сервером, а задача T2 – клієнтом (Рис. 2.2).

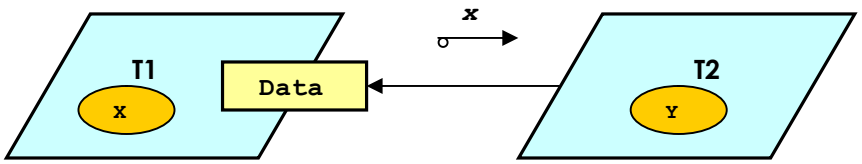

Рис. 2.2. Схема взаємодії задач. Версія 2

Виклик входу Data тепер відбувається в задачі T2. Виходячи з того, що напрям передавання даних (x) не змінився (від T1 до

T2), слід змініть модифікатор in на out в опису входу Data, бо він тепер використається для передавання даних з T1. Раніше вхід Data використався в задачі T2 для приймання даних.

❖ Приклад 2.2

```
-------------------------------------------------------------
--  Ада.Механізм рандеву. Версія 2                         --
-------------------------------------------------------------
procedure Lab2.2 is

   task T1 is
     entry Data(e: out integer);          -- опис вхіду
   end T1;

   task body T1 is
      x: integer:= 25;           -- локальна   змінна
   begin
         .  .  .
      -- оператор приймання виклику входу
      accept Data(e: out integer) do
         e:= x;    -- тіло оператора accept (відправка x)
      end Data;

   end T1;
   ---------------------------------------------
   task T2;   -- вироджена специфікація задачі

   task body T2 is
      y: integer;        -- локальна  змінна
   begin
         .  .  .
      T1.Data(y);  -- виклик входу Data задачі T1 для
                   -- приймання
         .  .  .
   end T2;
            -- основна програма
begin

   null;      -- порожній оператор

end Lab2.2;
```

Структурну схему взаємодії задач, під час якої задачі здійснюють обмін даними, зображено на рис.2.3. При цьому застосовані два

входи: Data для передавання і Result для приймання даних в задачу T2. Для запуску задач їх вкладено в процедуру Run_Tasks, виклик якої приведе до старту задач T1 і T2 в тілі головної процедури.

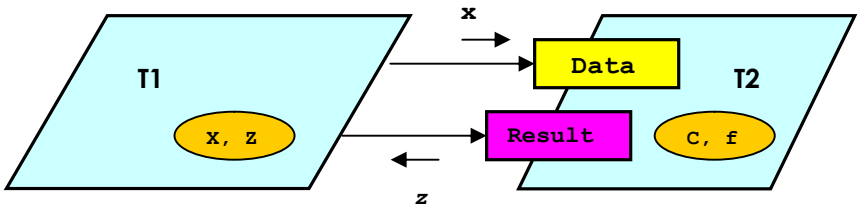

Рис. 2.3. Схема взаємодії задач (двостороння)

❖ Приклад 2.3

```
-------------------------------------------------------------
-- Ада.Механізм рандеву                                    --
-------------------------------------------------------------
procedure Lab2.3 is

   -- процедура із вкладеними задачами
   procedure Run_Tasks is
      task T1;

      task T2 is
         entry Data   (e: in  integer);
         entry Result (t: out integer);
      end T2;
      ------------------------------------
      task body T1 is
         x: integer := 25; -- для передавання
         z: integer;       -- для прийому
      begin
            .  .  .
         T2.Data(x);      -- передавання
            .  .  .
         T2.Result(z);    -- приймання
            .  .  .
      end T1;
      ------------------------------------
      task body T2 is
        c : integer;       -- для приймання
        f : integer:=200;  -- для передавання
      begin
            .  .  .
         -- прийом даних
         accept Data(e: in integer) do
            c:= e;
         end Data;

         -- повернення результату
         accept Result(t: out integer) do
            t:= f;
         end Data;
            .  .  .
      end T2;

   begin                -- основна процедура

      Run_Tasks;        -- запуск задач

   end Lab2.3;
```


Структурну схему взаємодії задач, під час якої задачі здійснюють обмін даними, зображено на рис.2.3. 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 T1 і T2.

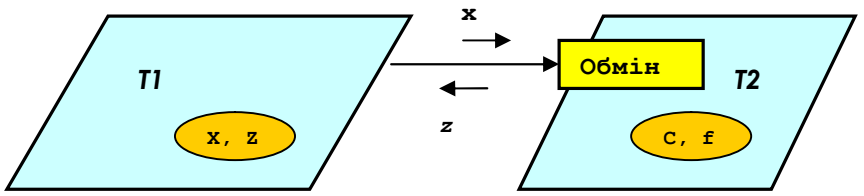

Рис. 2.4. Схема взаємодії задач (двостороння)

❖ Приклад 2.4

```
-------------------------------------------------------------
--Ада. Механізм рандеву. Обмін через один вхід           --
-------------------------------------------------------------
```

```
procedure Lab2.4 is

   -- процедура із вкладеними задачами
   procedure Run_Tasks is

      task T1;

      task T2 is
         entry Обмін(e: in integer, s:out integer);
      end T2;
      -------------------------------------
      task body T1 is
         x: integer := 25; -- для передавання
         z: integer;  -- для прийому
      begin
            .  .  .
         T2.Обмін(x, z);      -- передавання і приймання
            .  .  .
      end T1;
      -------------------------------------
      task body T2 is
        c : integer;        -- для приймання
        f : integer:=200;  -- для передавання
      begin
            .  .  .
         -- приймання і передавання даних
         accept Обмін(e: in integer, s:out integer) do
            c:= e;
            s:= f;
         end Data;
            .  .  .
      end T2;

   begin              -- основна процедура

      Run_Tasks;       -- запуск задач

   end Lab2.4;
```

Структурну схему взаємодії трьох задач – *T*1, *T*2 і *T*3 зображено на рис. 2.5. Задача T3 приймає дані від задач T1 і T2 за допомогою входів: DataT1 і DataT2. Особливістю реалізації взаємодії задач є використання в тілі задачі *T*3 оператора select. Це

дозволяє задачі *T*3 приймати виклик входів DataT1 та Datat2 в будь-якій послідовності в міру їх надходження.

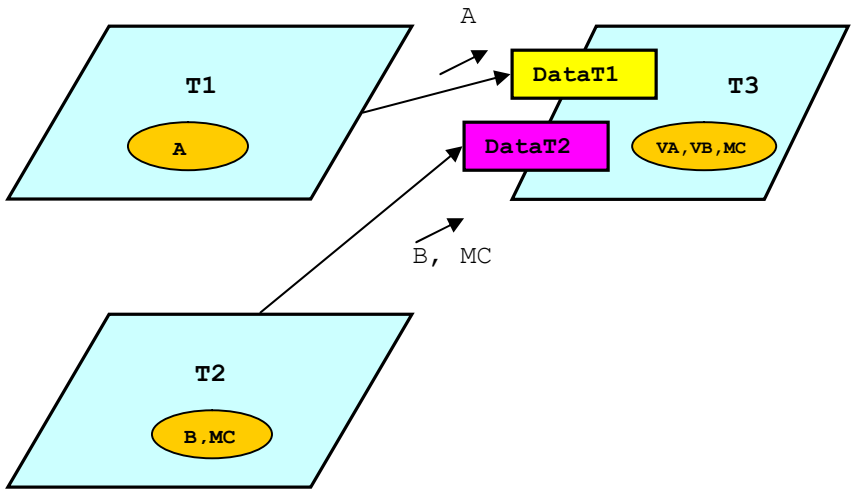

Рис. 2.5. Схема взаємодії задач T1-T3

❖ Приклад 2.5

```
-------------------------------------------------------------
--  Ада.Механізм рандеву                                   --
-------------------------------------------------------------
procedure Lab2.5 is

   N: integer:= 10;
   type Вектор  is array(1..N) of integer;
   type Матриця is array(1..N) of Вектор;

   procedure Run_Tasks is

      task T1;
      task T2;
      task T3 is
         entry DataT1(V:  in  Вектор);
         entry DataT2(V1: in  Вектор, MV1: in Матриця);
      end T3;
      ----------------------------
```

```
      task body T1 is
         A: Вектор;
      begin
          .  .  .
         T3.DataT1(A);
          .  .  .
      end T1;
      -------------------------
      task body T2 is
         B : Вектор;
         MC: Матриця;
      begin
          .  .  .
         T3.DataT2(B,MC);
          .  .  .
      end T2;
      -------------------------
      task body T3 is
         VA, VB: Вектор;
         MC: Матриця;
      begin
         .  .  .

          accept DataT1(V: in Вектор) do
             VA:= V;
          end Data;

          accept DataT2(V1: in Вектор; MV1: in Матриця) do
              VB:= V1;
              MC:= MV1;
          end Data;
             .  .  .
      end T3;
   begin

      null;

   end Run_Tasks;

begin              -- основна програма

   Run_Tasks;      -- явний запуск задач T1, T2, T3

end Lab2.5;
```

Оператор select

Приклад 2.5 задає жорстку послідовність взаємодії задач. В задачі T3 розміщення операторів accept визначає першим рандеву між задачами T1 і T3, а потім – між задачами T2 і T3. Якщо задача T2 буде раніше готова до рандеву, то вона все одно буде чекати коли відбудеться рандеву між T1 і T3.

Для ефективного програмування подобних видів взаємодії задач в рандеву передбачено використання оператора select.

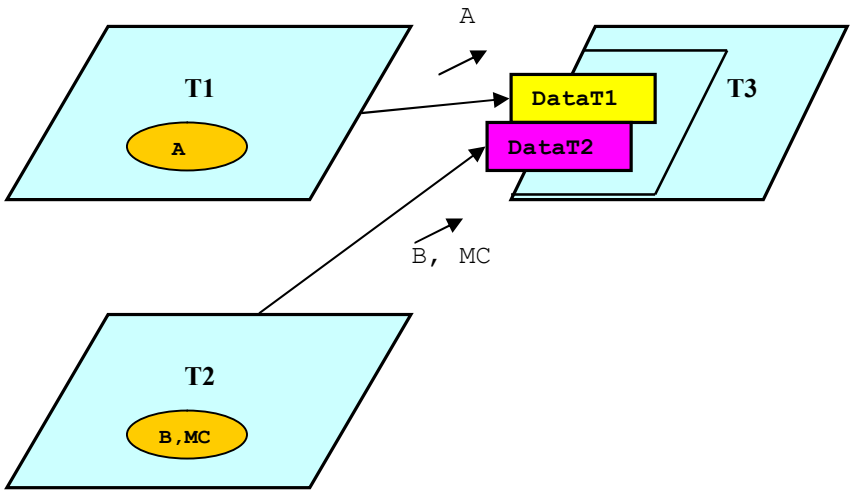

Рис. 2.6. Схема взаємодії задач T1-T3

Приклад 2.6 ілюструє використання оператора, який змінює порядок взаємодій задач в програмі 2.5. В задачі T3 обидва оператора accept DataT1 і accept DataT2 вкладено в оператор select. Коли задача T3 приходить на “місце побачення”, виконуватися буде той accept, який вже викликано. Якщо це виклик від задачі T1, то розпочинається рандеву меж T1 і T3, якщо це виклик від задачі T2, то розпочинається рандеву меж T2 і T3. У випадку, коли ніяких викликів немає, задача T3 блокується, чекає на виклики і готова до рандеву з будь якою задачею, чий виклик

прийде першим. Через наявність циклу для оператору select, задача T3 знову повертається до операторів accept і чекає на виклик наступної задачі.

❖ Приклад 2.6

```
------------------------------------------------------------
--  Ада.Механізм рандеву. Оператор Select                 --
------------------------------------------------------------
procedure Lab2.6 is

   N: integer:= 10;
   type Вектор  is array(1..N) of integer;
   type Матриця is array(1..N) of Вектор;

   procedure Run_Tasks is

      task T1;
      task T2;
      task T3 is
         entry DataT1(V:  in  Вектор);
         entry DataT2(V1: in  Вектор, MV1: in Матриця);
      end T3;
      ----------------------
      task body T1 is
         A: Вектор;
      begin
          .  .  .
         T3.DataT1(A);
          .  .  .
      end T1;
      ----------------------
      task body T2 is
         B : Вектор;
         MC: Матриця;
      begin
          .  .  .
         T3.DataT2(B,MC);
          .  .  .
      end T2;
      --------------------------
      task body T3 is
         VA, VB: Вектор;
         MC: Матриця;
      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 Lab2.6;
```

Існують де кілка видів оператору select:

- селективний accept ,
- часовий виклик входу,
- умовний виклик входу,
- асинхронний select.

Використання селективного оператора accept наведений в прикладі 2.6. Оператор accept додатково може мати *заглушку* (when умова = >) , яка відкриває (умова має значення true) або закриває (умова має значення true) альтернативу відбору:

```
task Box is
   entry DD1;
   entry DD2;
end Box;
task body Box is
      F: boolean:= false;
   loop
```

```
        select
           when F =>
             accept DD1 do
                 F:= true;
             end  DD1;
        or
           accept DD2 do
                 F:= false;
             end  DD2;
        or
            terminate;
        end select;
      end loop;
end Box;
```

Приклад використання часового виклику входу DataT1 задачі T1 з прикладу 2.6:

```
select
   T1.DataT1(B);
or
   delay 25.0; -- затримка задачі T1
end select;
```

Приклад використання умовного виклику входу Сигнал:

```
procedure Датчик(Y : in Тип_Задачі) is
begin
   loop
      select
        Y.Сигнал;  -- виклик входу
         return;
      else
         null;     -- очікування
      end select;
   end loop;
end;
```

Виникнення тупиків в механізмі рандеву

Потенційною причиною тупиків в рандеву є наявність системної операції блокування задачі, яка першою прибула на місце «по-

бачення» (наприклад, викликала вхід) і чекає на появу іншої задачі. Якщо задача, на яку очікують, ніколи не з'явиться (не викличе відповідний оператор accept), ми маємо тупикову ситуацію (dead-lock) через те, що першу задачу буде заблоковано назавжди.

Приклад 2.7 ілюструє появу тупика в прикладі під час взаємодії задач T1 і T2, якщо встановлений наступний порядок виконання операторів accept в задачі T2 (Таблиця 2.1). Тут порядок викликів входів Data і Result в T1 не відповідає порідку виконання операторів приймання викликів цих входів в T2. Це приводить к блокуванню T1 на операторі T2.Data(x), а також – к блокуванню задачі T2 на операторі accept Result.. end Result;

Таблиця 2.1. Тупикова ситуація в рандеву

T1	T2

`T2.Data(x);`
 	
`. . .`
 	
`T2.Result(z);`
	`accept Result(t: out integer) do`
	`  t:= f;`
	`end Result;`
	`. . .`
	
	`accept Data(e: in integer) do`
	`  c:= e;`
	`end Data;`

❖ **Приклад 2.7**

```
-------------------------------------------------------------
-- Ада.Механізм рандеву.Тупикова ситуація                  --
-------------------------------------------------------------
procedure Lab2.7 is

   -- процедура із вкладеними задачами
   procedure Run_Tasks is
      task T1;

      task T2 is
         entry Data   (e: in  integer);
         entry Result (t: out integer);
      end T2;
```

```
   -------------------------------------
   task body T1 is
      x: integer := 25; -- для передавання
      z: integer;       -- для прийому
   begin
        .  .  .
      T2.Data(x);      -- передавання
        .  .  .
      T2.Result(z);    -- приймання
        .  .  .
   end T1;
   -------------------------------------
   task body T2 is
     c : integer;       -- для приймання
     f : integer:=200;  -- для передавання
   begin
        .  .  .

      -- повернення результату
      accept Result(t: out integer) do
         t:= f;
      end Result;
           . . .
       -- прийом даних
      accept Data(e: in integer) do
         c:= e;
      end Data;


        .  .  .
   end T2;

begin              -- основна процедура

   Run_Tasks;      -- запуск задач

end Lab2.7;
```

Уникнути подобної тупикової ситуації можна встановленням коректного виклику входів в задачі T1, якій відповідає порідку виконання відповідних операторів `accept`.

В прикладі 2.8 наведена модифікація приклада 2.7 шляхом застосування оператора `select` в задачі T2 для запобігання тупику для приклада 2.7

❖ Приклад 2.8

```
-------------------------------------------------------------
-- Ада.Механізм рандеву.Безупикова ситуація.Оператор select
-------------------------------------------------------------
procedure Lab2.8 is

   -- процедура із вкладеними задачами
   procedure Run_Tasks is

      task T1;
      task T2 is
         entry Data   (e: in  integer);
         entry Result (t: out integer);
      end T2;
      -------------------------------------
      task body T1 is
         x: integer := 25; -- для передавання
         z: integer;       -- для прийому
      begin
           .  .  .
         T2.Data(x);      -- передавання
           .  .  .
         T2.Result(z);    -- приймання
           .  .  .
      end T1;
      -------------------------------------

      task body T2 is
        c : integer;       -- для приймання
        f : integer:=200;  -- для передавання
      begin
           .  .  .
         for in 1..2 loop
            select
               -- повернення результату
               accept Result(t: out integer) do
                 t:= f;
               end Result;
            or
               -- прийом даних
               accept Data(e: in integer) do
                  c:= e;
               end Data;
            end select;
         end loop;

          .  .  .
```

```
    end T2;

begin                -- основна процедура

    Run_Tasks;       -- запуск задач

end Lab2.8;
```

2.2 Ада в програмуванні для комп'ютерних систем зі спільною памяттю

Реалізація операції додавання векторів $A = B + C^* x$ у трипроцесорній системі з локальною пам'яттю (три процесори П1, П2, П3 і два пристрої введення-виведення (ПВВ)) з використанням механізму рандеву мови Ада.

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

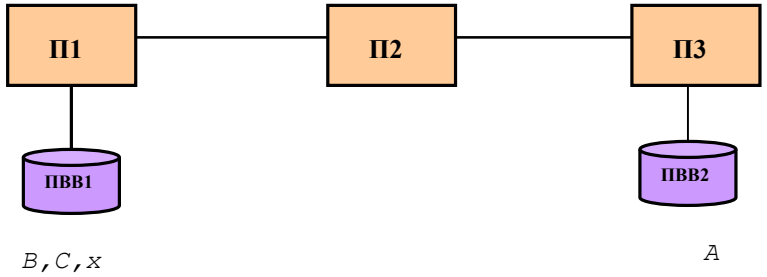

Рис. 2.7. Структура КС з локальною пам'яттю

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

$$A_н = B_н + C_н * x \qquad (2.1)$$

де $A_н$ – H елементів вектора A, ($H = N/3$)

Співвідношення (2.1) визначає дії задачіТ1-Т3 в кожному процесорі КС під час виконанні обчислень.

Етап 2. Розроблення алгоритмів роботи кожного процесу. Цей етап пов'язаний з розробленням алгоритмів роботи кожної задачі. 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 *Т1* частин вхідних даних $B_н$, $C_н$ та x по задачах *Т2* і *Т3*; оскільки задача *Т1* не пов'язана безпосередньо із задачею *Т3*, то пересилання даних між ними буде виконуватися через задачу *Т2;*
- збирання результату A в задачі *Т3.*

Розробляючи алгоритм задач, слід, звернути увагу на обсяги даних, що пересилаються між задачами. Під час розсилання вхідних даних із задачі *Т*1 в *Т*2 потрібно переслати дві частини вектора *B* і *C* розміром 2**H* (позначається як B_{2H} , C_{2H}), одна з яких призначена безпосередньо для Т2, а друга (B_{2H} , C_{2H}) – для пересилання далі в задачу *Т*3. 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 *Т1*

1. Введення *B, C* та *x*
2. **Передати** задачі *Т2* дані: $B_{2н}$, $C_{2н}$, *x*
3. Обчислення $A_н = B_н + C_н * x$
4. **Передати** в задачу *Т2* частину результату $A_н$

Задача *Т2*

1. **Прийняти** від задачі *Т1* дані: $B_{2н}$, $C_{2н}$, *x*
2. **Передати** в задачу Т3 дані $B_н$, $C_н$, *x*

3. Обчислення $Ан = Вн + Сн * x$
4. **Прийняти** від *Т1* результат – вектор *Ан*
5. **Передати** в *Т3* частини результату – підвектор *А2н*

Задача *Т3*

1. **Прийняти від** задачі *Т2* дані: *Вн, Сн, x*
2. Обчислення $Ан = Вн + Сн * x$
3. **Прийняти від** задачі *Т2* частину результат – підвектор *А2н*
4. Виведення результату – вектору А

Етап 3. Розроблення структурної схеми взаємодії задач. Структурна схема взаємодії задач ґрунтується на алгоритмах задач і дозволяє наочно показати передавання даних між задачами (рис.2.8).

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.

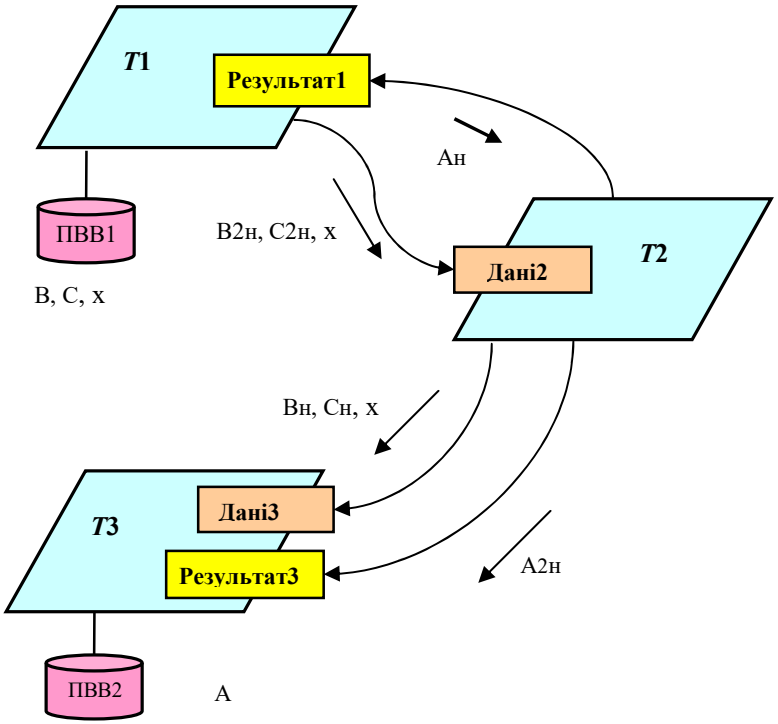

Рис. 2.8. Структурна схема взаємодії задач

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 *T*1 в *T*2 і потім з *T*2 в *T*3. Входи задач, що зазначені як `Результат()`, використані для збирання результату в задачі *T*3.

Для роботи з векторами також слід використовувати механізм відрізків, які дозволяють передавати (отримувати) різні частини векторів під час роботи з фактичними параметрами для виклику входу.

```
------------------------------------------------------------
--|Ада.Механізм рандеву Операція A = B + C * x           --
------------------------------------------------------------
procedure Lab2.9 is
   N: integer := 9;     -- розмір векторів
   P: integer := 3;     -- кількість процесорів
   H: integer := N/P;   -- розмір підвектора

   -- формування типів
   type Вектор  is array(integer range <>)
                                          of integer;
   subtype ВекторN  is Вектор(1..N);
   subtype Вектор2H is Вектор(1..2*H);
   subtype ВекторH  is Вектор(1..H);

   -- опис задач
   task T1 is
      entry Результат1(VA: out  ВекторH);
   end T1;

   task T2 is
      entry Дані2(VB,VC : in Вектор2H;
                          y : in integer);
   end T2;

   task T3 is
```



```
      entry Дані3(VB,VC    : in  ВекторH;
                           y    : in  integer);
      entry Результат3(VA : in  Вектор2H);
end T3;
-----------------------------------------
task body T1 is
    x1: integer;
    A1: ВекторH;
    B1, C1 : ВекторN;
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.Дані2(B1(H+1..N), C1(H+1..N), x1);

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

    -- передати в T2 Ан
    accept Результат1(VA: out  ВекторH) do
       VA:= A1;
    end Результат1;

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

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

   x2: integer;
   A2, B2, C2 : Вектор2H;
begin
   put_line("     Process T2 started");

    -- прийняти B2H, C2н і x  від T1
    accept Дані2(VB,VC : in Вектор2H;
                         y : in integer) do
       B2 := VB;
```

```
        C2 := VC;
        X2 := y;
    end  Дані3;

    -- передати  данні в T3
    T3.Дані3(B2(H+1..2*H), C2(H+1..2*H), x2);

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

    -- прийняти результат від T1
    T1.Результат1(A2(1..H));

    -- передати результат  A2H в T3
    T3.Результат3(A2(1..2*H));

   put_line("Process T2 finished");
end T2;
-----------------------------------
task body T3 is
   x3:      integer;
   B3, C3:   ВекторH;
   A3:       ВекторN;
begin

    put_line("     Process T3 started");

   -- прийняти від T2 BH, Cн і x
    accept Дані3(VB,VC : in ВекторH;
                 y : in integer) do
      B3 := VB;
      C3 := VC;
      X3 := y;
   end  Дані3;

    -- Обчислення
   for i in 2*H+1 .. N loop
       A3(i):= B3(i) + C3(i)*x3;
   end loop;

    -- прийяти  A2н від T2
   accept Результат3(VA:  in  Вектор2H) do
       A3(1..2*H):= VA;
   end Результат3;
```



```
       -- виведення результата
       put_line(" A = ");
       for i in 1 .. N loop
           put(A3(i),3);
       end loop;

       put_line("Process T3 finished");

   end T3;

--  тіло основної програми
Begin
   put_line("  Main procedure started ");
end Lab2.9;
```

➢ Запитання для модульного контролю

1. На чому ґрунтується механізм рандеву ?
2. Що включає опис входу задачі ?
3. Як здійснюється взаємодія задач під час рандеву ?
4. Що може бути джерелом тупику в механізмі рандеву ? Наведіть приклади виникнення тупику.
5. Як синхронізуються задачі під час виконання рандеву?
6. Яке призначення та які види оператора `select` ?
7. Яким чином визначаються обсяг та напрям передавання даних в механізмі рандеву ?
8. У чому полягає відмінність між реалізацією механізму рандеву у мовах Оккам і Ада ?

➢ 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. Виконати модифікацію завдання 3 для забезпечення передавання масиву будь-якого розміру.
4. У завданні 3 забезпечити передачу, крім масиву цілих, ще й масив типу BYTE.

РОЗДІЛ 3.
Програмування для розподілених комп'ютерних систем

Цей розділ присвячено питанням організації обчислень у розподілених комп'ютерних системах з використанням мови Ада. Розглянуто використання механізму сокєтів (WinAPI) та видалених процедур (RPC).

3.1 Розподілені обчислення в мові Ада

Стандарт мови Ада визначає поняття розподіленої системи та розподіленої програми [25, 29-31].

Розподілена система - це комп'ютерна система, що містить взаємозв'язані *вузлі обробки* та *вузли зберігання*. Вузол обробки (*processing node*) – це системний ресурс, що має як обчислювальні, так і запам'ятовуючі можливості. Вузол зберігання (*storage node*) - системний ресурс, який має лише можливості зберігання, з можливістю зберігання адреси однією або кількома обробками вузли (Рис. 3.1).

Розподілена програма (*distributed program*) містить один або кілька розділів (*partitions*), які виконуються незалежно (крім випадків, коли вони спілкуються) в розподіленій системі.

Кожен розділ має анонімну задачу (task) оточення, яка є неявною зовнішньою задачею, виконання якої опрацьовує бібліотечний елемент середовища декларативної частки, а потім викликає головну підпрограму, якщо така є. Виконання розділу є виконанням його задачі.

Процес відображення розділів програми до вузлів у розподіленій системі називається налаштуванням (конфігуруванням) розділів програми (*configuring the partitions of the program)*.

Розділи розподіленої програми класифікуються як *активні* або *пасивні*. Активний розділ - це розділ, що має власний потік управління. Пасивний розділ - це розділ, який не має власного потоку управління, бібліотечні підрозділи якого всі попередньо підготовлені, а дані та підпрограми доступні для одного або декількох активних розділів.

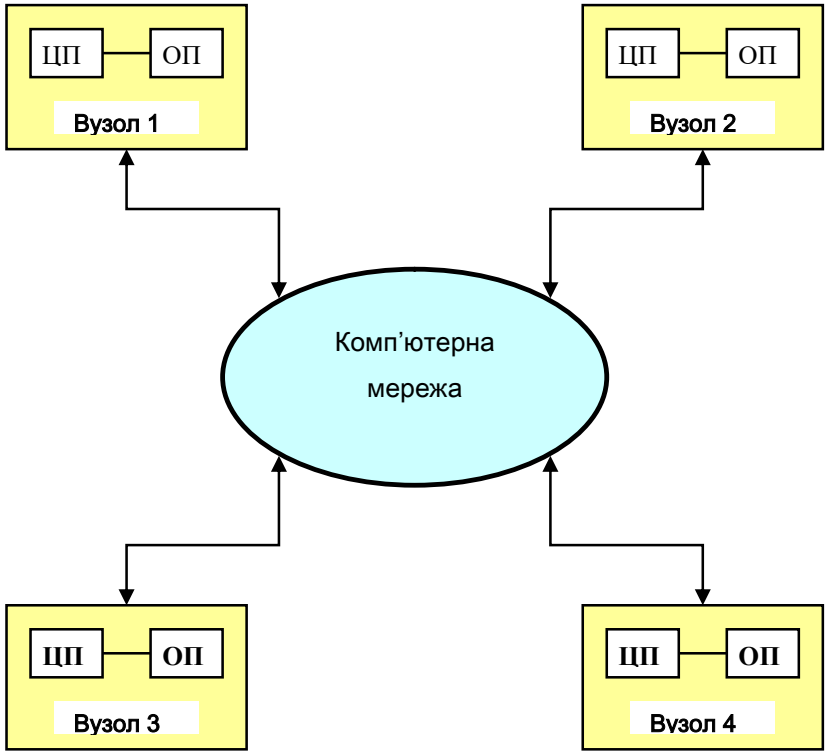

Рис.3.1 Розподілена комп'ютерна система . Апаратна складова

Пасивний розділ повинен містити лише бібліотечні елементи, які або оголошені `Pure`, або `Shared_Passive`. Зв'язок через пасивний розділ здійснюється за рахунок використання захищених (pritected) об'єктів, які оголошено як `Shared_Passive`.

При конфігуруванні активний розділ повинен бути налаштований на вузлі обробки, а пасивний розділ повинен бути налаштований або на вузол зберігання, або на вузол обробки.

Конфігурація розділів програми на розподіленій системі повинна відповідати можливості для посилань на дані або виклики між розділами, що мають на увазі їх семантичні залежності. Будь-яке посилання на дані або виклик підпрограми.

Реалізація розподіленої програми може дозволити конфігурувати кілька активних або пасивних розділів на одному вузлі обробки, а кілька пасивних розділів конфігурувати на одному вузлі зберігання. У цих випадках визначаються політики планування, обробка пріоритетів та управління спільними ресурсами між цими розділами.

Реалізація може дозволити конфігурувати окремі копії активного розділу на різних вузлах обробки, а також забезпечити відповідні взаємодії між копіями, щоб представити узгоджений стан розділу з іншими активними розділами.

Розділи розподіленої програми не потрібно завантажувати та опрацьовувати все одночасно; вони можуть бути завантажені та розроблені по черзі протягом тривалого періоду часу. Реалізація може забезпечити можливості для переривання та перезавантаження розділу під час виконання розподіленої програми.

На рисунку 3.2 наведений приклад розподіленої системи. Апаратна частина складається з чотирьох вузлів: Вузол1 – Вузол4. Вузли єднає комп'ютерна мережа.

Програмна складова системи включає розділі (partitions), які розміщуються в кожному вузлі. Розділи, в свою чергу, формуються з програмних модулів (units). Програмні модулі поділяються на пасивні (МП) або активні (МА). Система на рис.3.8 включає шість активних та три пасивних модуля. Вузол 4 є вузлом зберігання (storage node), він не містить активних модулів. Вузли 1-3 є вузлами обробки (processing nodes).

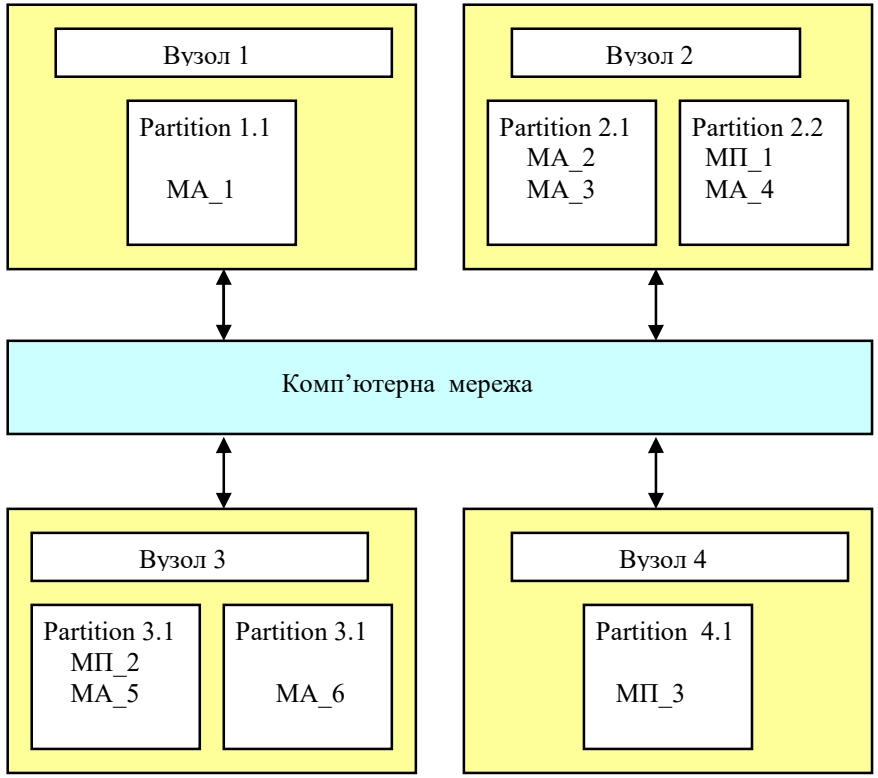

Рис. 3.2. Ada.Структура розподіленої системи .
Апаратна та програмна складові

3.2 Сокети

Мова Ада не має реалізації механізму сокетів. Через наявність в мові Ада інтерфейсу з WinAPI, є можливість використання сокетів бібліотеки WinAPI.

Створення сокету в Windows Sockets API (скорочено Winsock) відбувається за допомогою функції socket:

```
SOCKET socket (int P1, int P2, int P3);
```

Параметри функції:

- P1 визначає вид протоколу (AF_INET для інтернету),
- P2 визначає тип сокета (SOCK_STREAM для протоколу TCP, SOCK_DGRAM для протоколу UDP),
- P3 визначає вид транспортного протоколу (0 – для автоматичного вибору TCP або UDP).

Приклад створення TCP сокету Серверний_сокет :

```
Серверний_сокет = socket(AF_INET, SOCK_STREAM, 0);
```

Встановлення на боці клієнта з'єднання з сервером (віддаленим вузлом) за здійснюється допомогою функції connect:

```
int connect( SOCKETP P1,
             const struct sockaddr FAR* P2,
             int P3);
```

Параметри функції:

- P1 дескриптор сокета, що створено функцією socket,
- P2 вказівник на структуру даних типу sockaddr, яка містить адресу і порт серверного вузла,
- P3 розмір структури даних sockaddr;

Після виклику функції connect клієнт намагається встановити з'єднання із сервером. Повернення функцією socket ненульового значення означає неможливість такого з'єднання.

Структура даних sockaddr_in використається для формування мережевих параметрів (IP –адресів, портів):


```
struct sockaddr_in
{
   short         sin_family; // семейство протоколів
                             //(как правило AF_INET)
   u_short        sin_port;  // порт
   struct in_addr sin_addr;  // IP-адрес
   char           sin_zero[8];
};
```

Перш ніж сервер зможе використати сокет, він повинен зв'язати його з локальною адресою, де треба вказати IP адресу вузла и порт. Це виконується за допомогою виклику функції bind :

```
int bind (SOCKET P1,
          const struct sockaddr FAR* P2,
          int P3)
```

Параметри функції:

- P1 дескриптор сокета, що створено функцією socket,
- P2 вказівник на структуру даних типу sockaddr, яка містить адресу і порт серверного вузла,
- P3 розмір структури даних sockaddr;

Сервер викликає функцію listen, і переходить в режим очікування підключення клієнта

```
int listen (SOCKET P1, int P2)
```

Параметри функції:

- P1 дескриптор сокета, що створено функцією socket,
- P2 максимально допустимий розмір черги повідомлень.

За допомогою метода accept сервер контролює наявність в черзі запитів на з'єднання з боку клієнта:

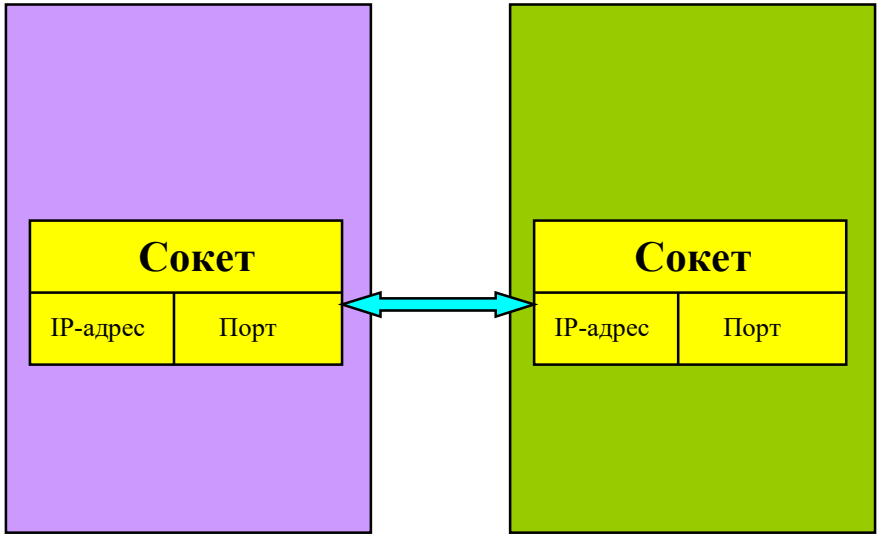

Рис.3.3 Сокети.Взаємодія сервера та клієнта


```
SOCKET accept(SOCKET P1,
              struct sockaddr FAR* addrP2,
              int FAR* addrlenP3)
```

Параметри функції:

- P1 дескриптор сокета, що створено функцією socket,
- P2 вказівник на структуру даних типу sockaddr, яка містить адресу і порт серверного вузла,
- P3 розмір структури даних sockaddr;

Функція accept створює новий сокет, виконує зв'язування, додає в структуру даних sockaddr відомості про клієнта, що підключився (його IP адресу и порт). Якщо черга пуста, то метод чекає на встановлення зв'язку з клієнтом.

Після встановлення зв'язку можлива взаємодія між сервером та клієнтом по передаванню і прийманню даних через сокет за допомогою функцій send і receive.

Функція send повертає управління одразу після виконання незалежно від того, отримала лі сторона, що приймає, дані. Вона повертає кількість переданих даних.

```
int send (SOCKET P1,
          const char FAR * P2,
          int P3,
          int P4)
```

Параметри функції:

- P1 дескриптор сокета, що створено функцією socket,
- P2 вказівник на буфер с даними для пересилання,
- P3 розмір буферу,
- P4 прапор управління.

Функція recv повертає управління одразу після виконання незалежно від того, отримала лі сторона, що приймає, дані. Вона повертає кількість переданих даних.

```
int recv (SOCKET P1,
          char FAR* P2,
          int P3,
          int P4)
```

Параметри функції:

- P1 дескриптор сокета, що створено функцією socket,
- P2 вказівник на буфер с даними для приймання,
- P3 розмір буферу,
- P4 прапор управління.

Закриття з'єднання і знищення сокету виконується за допомогою функції closesocket:

```
int closesocket (SOCKET s),
```

яка повертає нульове значення у випадку вдалого завершення операції.

В прикладі 3.1 наведено взаємодію через сокети WinSock клієнта та сервера. Клієнт передає серверу значення цілої змінної Buf_Clnt, яку сервер по отриманню розміщує в змінній Buf_Serv.

❖ **Приклад 3.1**

```
//-----------------------------------------------------
//-- C++. Сокети WinSock                             --
//-----------------------------------------------------
// сервер
#define PORT "567"

int_cdecl main(void){

   WSADATA wsaData;
   int r;

   SOCKET Cl_sock;--сокет приймання даних від клієнта
   SOCKET Lst_sock; --сокет прослуховування

   int BufServ; -- буфер для приймання
```


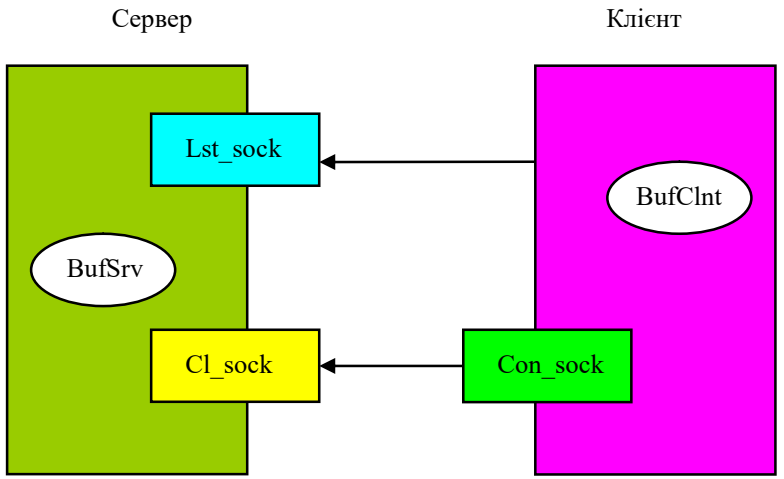

Рис.3.3 Приклад 3.1. Взаємодія сервера та клієнта

```
--   ініціалізація бібліотеки Winsock
  r = WSAStartup(MAKEWORD(2,2),&wsaData));

--  формування серверної IP адреси та порту
ZeroMemory(&hints, sizeof(hints)); 0
hints.ai_family   = AF_NET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
hints.ai_flags    = AI_PASSIVE;

 // формування IP адреси і порту
 r = getaddrinfo(NULL,PORT,&hints,&res);

 // створення сокету для з'єднання
 Lst_sock = socket(res->ai_family,
                   res->ai_socktype,
                   res->ai_protocol);

  -- зв'язування сокету із sockaddr
  r = bind(Lst_sock,
             res->ai_addr,
                (int)res->ai.addrlen);

  freeaddrinfo(res);

 // прослухування на підключення клієнта
 r = listen(Lst_sock,SOMAXCONN);

 // контроль підключення клієнта
 Cl_Sock = accept(Lst_sock, NULL,NULL);

 // закриття сокету
 closesocket(Lst_sock);

 // приймання даних  на сервері
 r = recv(Cl_sock,BufServ,1,0);

 // друк результату
 prinf("Результат  = ");
 prinf(BufServ);
```



```
  // вихід
  printf("Завершення взаємодії");
  closesocket(Cl_sock);
  WSACleanup();

  return 0;
}

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

// клієнт

#define PORT "567"
int_cdecl main(int argc, char **argv){

   WSADATA wsaData;
   SOCKET Con_sock;     -- сокет для з'єднання
                        -- із сервером

   struct addrinfo *res= NULL,
                   *ptr=Null,
                    hints;

   int BufClnt 9; -- данні для передавання
   int r;

   --   ініціалізація бібліотеки Winsock
   r = WSAStartup(MAKEWORD(2,2),&wsaData));

   --  формування серверної IP адреси та порту

   ZeroMemory(&hints, sizeof(hints));
   hints.ai_family   = AF_UNSPEC;
   hints.ai_socktype = SOCK_STREAM;
   hints.ai_protocol = IPPROTO_TCP;

  // формування IP адреси і порту
   r= getaddrinfo(argv[1],PORT,&hints,&res);

  // створення сокету для підключення до серверу

    Con_sock = socket(ptr->ai_family,
```

```
                    ptr->ai_socktype,
                    ptr->ai_protocol);

   // підключення до сервера
   r = connect(Con_sock,
                  ptr->ai_addr,
                      (int)ptr->ai_addrlen);

   freeaddrinfo(res);

   -- передавання даних  серверу
  r = send(Con_sock,BufClnt,1,0);

   // Вихід
   printf("Завершення взаємодії");
   closesocket(Con_sock);
   WSACleanup();

    return 0;
}
```

3.3 Віддалені процедури

Механізм виклику віддаленої процедури (механізм RPC – Remote Procedure Call) реалізує високо ревеневу 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 той час, як RPC-механізм дозволяє використовувати для розміщення віддаленої процедури пасивні модулі, наприклад, пакети.

Часову діаграму, яка ілюструє взаємодію клієнта і сервера з використанням віддаленої процедури `SumS(X,Y: in Вектор;Z: out Вектор)` показано на рис. 3.5.

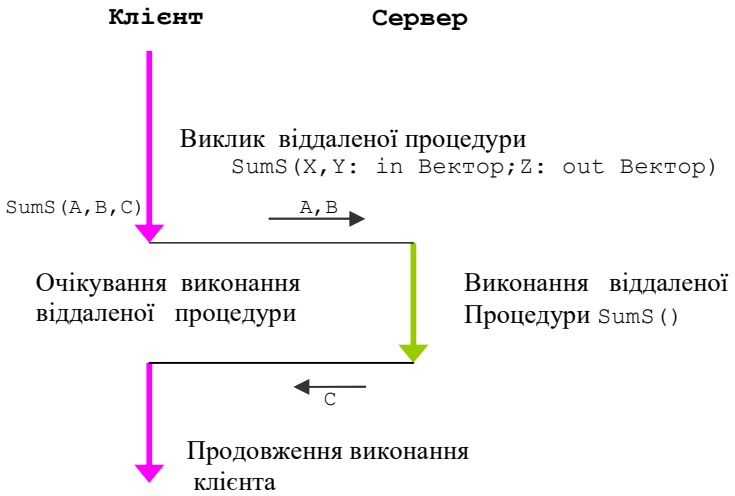

Рис. 3.4 Синхронна взаємодія клієнта та сервера за допомогою механізму RPC

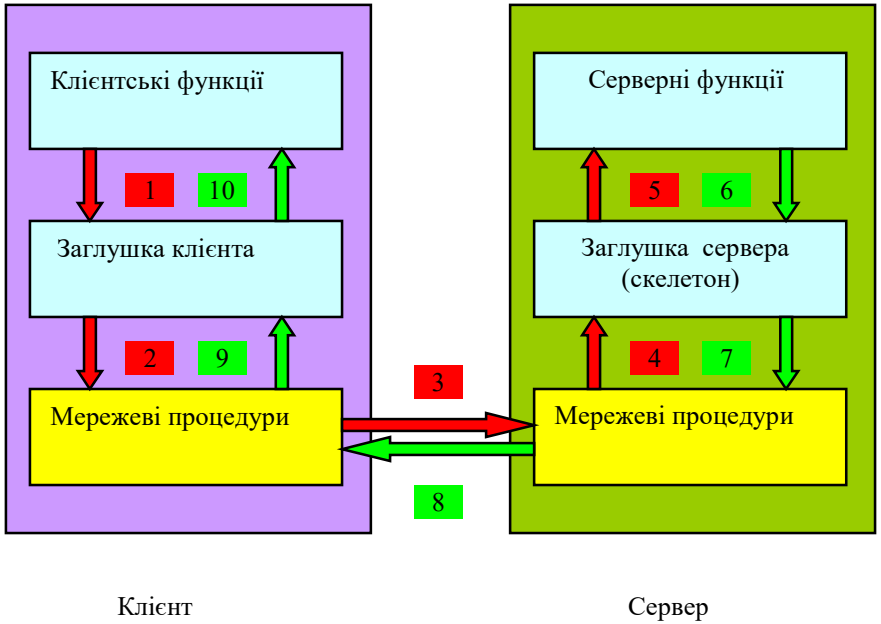

Рис.3.5 RPC.Взаємодія сервера та клієнта

Друга часова діаграма, яка ілюструє взаємодію клієнта і сервера з використанням іншої віддаленої процедури `SumA(X,Y: in Вектор)`, показано на рис. 3.6. Виклик віддаленої процедури `SumA()` асинхронний і не пов'язаний з очікуванням клієнтом виконання віддаленої процедури та повернення її результату. В цьому випадку клієнт передає вхідні дані віддаленої процедури `SumA()` під час її виклику і потім продовжує своє виконання далі.

Результат виконання віддаленої процедури `SumA()` 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.

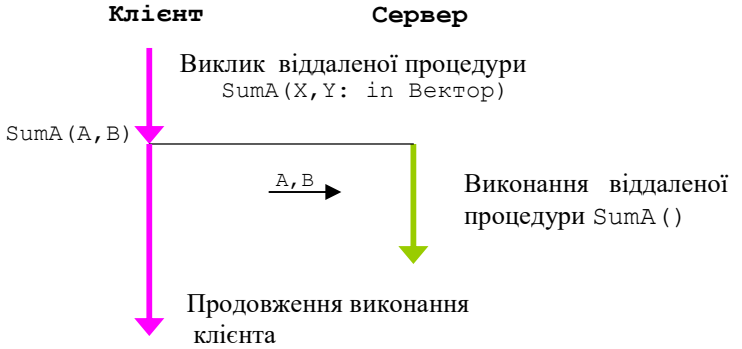

Рис. 3.6. Асинхронна взаємодія клієнта та сервера за допомогою механізму RPC

Схему взаємодії вузлів РКС за допомогою механізму RPC показано на рис. 3.7.

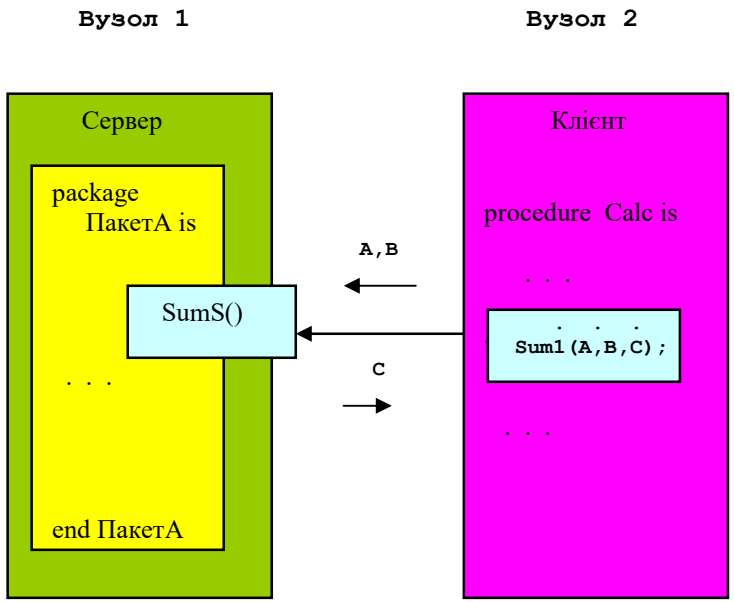

Рис. 3.7. Взаємодія вузлів за допомогою виклику віддаленої процедури `Sum1`

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

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

Ада. RPC механізм

Механізм RPC лежить в основі моделі розподілених обчислень, яку запропоновано в другому стандарті мови Ада у додатку «Розподілені системи» (Annex E “Distributed Systems” - DSA) [58].

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 Ада 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;
pragma  All_Calls_Remote;
```

Для прагм категорії існує правила видимості. Загальне правило залежності:

```
Remote_Call_Intertface >
                Remote_Type >
                    Shared_Passive >
                                Pure.
```

Це означає, що пакет `Remote_Call_Intertface` може зробити видимими в своєї специфікації тільки модулі `Remote_Types`, `Shared_Passive` та `Pure`.

`Pragma` `Shared_Passive(Ім’я_бібліотечного_модуля)`. Використається для опису бібліотечного модуля, який управляє глобальними даним, що поділяються активними розділами. Активні розділи взаємодіють через посилання на об’єкти, що об’явлень в бібліотечному модулі з прагмою `Shared_Passive`. Модули з прагмою `Shared_Passive` забезпечують прямий доступ до даних із різних розділів, підтримку і захист спільної пам’яті через `protected` об’єкти.

Pragma Remote_Type(Ім'я_бібліотечного модуля). Використається для опису бібліотечного модуля, який підтримує визначення типів для використанні при обміні даними між активними розділами. Модули з прагмою Remote_Type забезпечують опис розподілених об'єктів, дозволяють здійснювати динамічне зв'язування віддалених викликів через віданні посилальні типи.

Pragma Remote_Call_Intertface(Ім'я_бібліотечного модуля).**Pragma** Remote_Call_Intertface(Ім'я_ бібліотечного модуля). Модуль бібліотеки інтерфейсу віддалених викликів використається в якості інтерфейсі для віддалених викликів процедур (RPC) (віддалених функцій) між активними розділами. Визначає, що такий бібліотечний модуль є інтерфейсом віддаленого виклику (RCI). Підпрограма, яка об'явлена в видимої частині RCI бібліотечного модуля (або об'явлена таким бібліотечним модулем), має назву віддаленої підпрограми.

Модули з прагмою Remote_Call_Interface забезпечують віддалений виклик та виконання процедур, статичне та динамічне зв'язування віддалених викликів.

Приклад пакета Fox, у якому описано звичайні процедури Pr1 та Pr2 :

```
package Fox is

   procedure Prc1( ... );
   procedure Prc2( ... );

end Fox;
```

Пакет FoxDstr1 є модифікацією пакету Fox, в якому за допомогою прагми Remote_Call_Interface процедури Prc1 та Prc2 описано як віддалені:

```
package FoxDstr1 is

   pragma Remote_Call_Interface;
   procedure Prc1( ... );
```



```
   procedure Prc2( ... );

end FoxDstr1;
```

Прагма Remote_Call_Interface визначає всі ресурси пакета FoxDstr1 як такі, що допускають віддалений доступ. Це обидві процедури Prc1 та Prc2.

Віддалений виклик:

```
with FoxDistr1;
procedure Lab 3.1 is
   . . .
   FoxDistr1.P1(. . .);  -- віддалений виклик
  . . .
end Lab3.3;
```

Приклади застосування прагм категорії Pure.

```
package TypeD is
   pragma Pure;
   type Type_Server is new integer;
end TypeD;
```

❖ **Приклад 3.2**

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().

```
-------------------------------------------------------------
-- Ада. Віддалена процедура. Множення матриць              --
-------------------------------------------------------------
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

   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 мові Ада асинхронний виклик описується за допомогою прагми `Asynchronous`.

Процедура з директивою `Asynchronous`, дозволяє здійснювати асинхронне виконання статично і динамічно зв'язаних віддалених викликів. Виклик асинхронної процедури не чекає на завершення віддаленого виклику, що дозволяє клієнту, який викликає процедуру, продовжити своє виконання. Така віддвлена процедура повинна мати лише модифікатор `in` для параметрів.

Модифікація пакета `Fox` для забезпечення асинхронного виклику процедури `Pr4( ... )`:

```
package FoxDistr3 is

   pragma Remote_Call_Interface;

   procedure Pr4( ... );
   pragma Asynchronous(Pr4);

end FoxDistr3;
```

❖ Приклад 3.3

Це модифікація прикладу 3.3, 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_МА`, яка викликається в клієнті пізніше. Схему взаємодії розділів такої програми показано на рис.3.6.

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

   -- віддалений інтерфейс
   pragma Remote_Call_Interface;

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

   procedure Отримати_МА(МА :  out  Матриця);

   pragma Asynchronous(Додавання_Матриць);
   pragma Asynchronous(Отримати_МА);

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:= MT;
   end Отримати_МА;

end  Ресурс;
-------------------------------------------------
with Ресурс;
procedure Кліент 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 (обчислення 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 Кліент;
```

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.

В мові Ада існує підсистема взаємодії розділів (PCS - Partition Communication Subsystem), яка підтримує передавання та приймання даних між клієнтом та сервером через потік типу System.RPC.Params_Stream_Type

```
type Params_Stream_Type(Initial_Size:
          Ada.Streams.Stream_Element_Count) is
            new Ada.Streams.Root_Stream_Type
                                with private;
```

Тип є контейнером передавання даних між розділами. Кореневий тип - тип Root_Stream_Type, який визначає базовий тип потоку та дві абстрактні операції: Write для розміщення в потік, і Read для витягу з потоку об'єктів типу Stream_Element _Array, що є масивом байтів.

Ідентифікатор розділу визначається атрибутом P'*Partition_ID*

Система GLADE

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

Мова опису конфігурації, яку застосовано в GLADE, ґрунтується 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 3.6:

```
-------------------------------------------------------------------
-- GKADE. 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;
```

У формуванні розділів (рис. 3.8) розділ Розділ_Клієнта містить головну процедуру програми Клієнт, з якої починається її виконання, а розділ Розділ_Серверу містить RCI пакет Ресурс з віддаленою процедурою Додавання_Матриць. Вказується адреса вузла, на якому потрібно виконувати розділ Partition_2.

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

Експериментальні дослідження ефективності RPC механізму у мові Ада наведено в роботі [24].

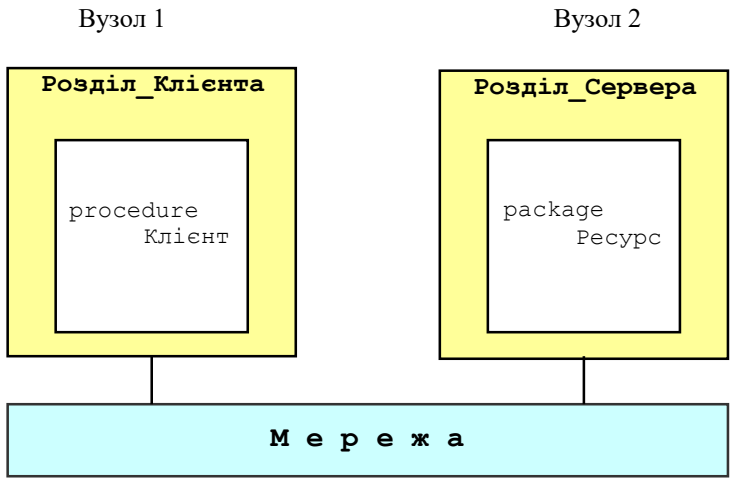

Рис. 3.8. Формування розділів

➢ **Запитання для модульного контролю**

1. У чому полягає особливість програмування для розподілених систем ? Які особливості має модель клієнт – сервер ?
2. Створення сокетів в бібліотеці WinSock?
3. WinSock.Призначення структури `sockaddr_in` ?
4. WinSock.Призначення функції `socket()`?
5. WinSock.Призначення функції `connect()`?
6. WinSock.Призначення функції `bind()`?
7. WinSock.Призначення функції `listen()`?
8. WinSock.Призначення функції `accept()`?
9. WinSock.Призначення функції `send()`?
10. WinSock.Призначення функції `recv()`?
11. Як у методі запису в потік визначається зв'язок з потрібним сокетом ?
12. Де здійснюється виконання віддаленої процедури ? Які дії виконуються піл час виклику віддаленої процедури ?

13. Яке призначення прагм категорії в механізмі RPC ? Як формується розділ ?
14. У чому полягає особливість RPC механізму мови Ада95 під час переходу від звичайної моделі до розподіленої ? Як це здійснюється ?
15. Ада RPC. Які складові має мова конфігурування ?

➢ **Завдання до самостійної роботи**

1. Розробити алгоритми сервера та клієнта для реалізації операції паралельного множення вектора на матрицю.
2. Розробити програму сортування вектора в розподіленій системі. Застосувати сокети (WinAPI).
3. Розробити програму множення матриць у розподіленій системі. Застосувати сокети (WinAPI).
4. Розробити програму множення матриць у розподіленій системі. Застосувати RPC.
5. Розробити програму множення матриць в розподіленій системі. Застосувати RPC .
6. Розробити програму множення матриць в розподіленій системі. Застосувати RPC мовиАда.

СПИСОК ЛІТЕРАТУРИ

1. Бар Р. Язык Ада в проектировании систем. – М.: Мир, 1988 – 320 с.
2. Василеску Ю. Прикладное программирование на языке Ада 95. – 1990. – 332 с.
3. Вегнер П. Программирование на языке Ада. – М.: Мир, 1983. – 78 с.
4. Джехани Н. Язык Ада. – М.: Мир, 1988. – 552 с.
5. Жуков І.А. Корочкін О.В. Паралельні та розподілені обчислення. Навч. посібн. 2-ге видання, К.: Корнійчук, 2014. – 284 с.
6. Корочкін О. Багатоядерне програмування на мові Ада. Гавю посібн. Част. 1. (англ.мовою). - К.:КПІ ім. І.Сікорського, 2018.- 226 с.
7. Корочкин А. В. Ада95: Введение в программирование – К.: Свит, 1999. – 260 с.
8. Касперски К. Секреты поваров компьютерной кухни или ПК: решение проблем. – BHV, 2003. – 560 с.
9. Корочкін О.В. Багатоядерне програмування на мові Ада. Навч. посібн. Част. І.(англ. мовою).- Київ, КПІ ім. І.Сікорського 2018.- 226 с.
10. Корочкин А., Мустафа Акрам Параллельные вычисления: Ада и Java. – Вісн. НТУУ "КПІ", Інформатика, управління та обчислювальна техніка, 1999, К.: – № 32, С. 13–17.
11. 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
9. Пайл Я. Ада – язык встроенных систем. – М.; Финансы и ста тистика, 1984. –120 с.
10. Перминов О.Н. Введение в язык программирования Ада. – М.: Радио и связь, 1991. – 228 с.
11. Русанова О.В. Программное обеспечение компьютерных систем. Особенности программирования и компиляции. – К.: Корнійчук, 2003. – 94 с.
12. Хоар Ч. Взаимодействующие последовательные процессы. – М.: Мир, 1989. – 180 с.

13. Языки программирования: Ада, Паскаль, Си.Сравнение и оценка / Под ред.Н. Джехани – М.: Радио и связь, 1989. – 386 с.
14. Burns A., Wellings A.Real-Time Systems and Programming Languages. Addison – Wesley, 2001, – 386 p.
15. Burns A., Wellings A. Concurrency in Ada. – Cambridge: Cambridge University Press, 1995. – 420 p.
16. Hoare C.A.R. Communicating Sequential Processes, Communications of ACM, vol.21, №. 8, Aug. 1978, pp. 666 – 667.
17. Hoare C.A.R. Communicating Sequential Processes,: Printice Hall, International Series in Computer Science, Englewood Cliffs NJ, 1985. – 186 p.
18. 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.

19 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;

20. 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.
21. 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.
22. 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.
23. Lutzky G., Korochkin O. Parallel Computing, Korneychuk, Kyiv, 2007, - 240 pp.
24. McCormick J.Singhoff F., Hugues J. Building Parallel, Embedded, and Real-Time Applications with Ada , Cambridge University Press,

2011, - 366 p.
25. Taft S., Duff R., Brukardt R., Ploedereder T.Consolidated Ada Refe– Reference Manual. Language and Standard Libraries, Springer, Ber– lin: 2001, – 562 p.

26. Ben-Ari M. Ada for Software Engineering, Cambridge University Press, 2011, - 366 p.
27. http://freecomputerbooks.com/langAdaBooks.html
28. http://www.ada-auth.org/standards/12rat/html/Rat12-TTL.html
29. http://www.ada-auth.org/standards/rationale12.html#update